ESP32-S3

技术参考手册 版本 1.8



www.espressif.com

关于本文档

ESP32-S3 技术参考手册面向使用 ESP32-S3 系列产品进行底层软件开发的人员，介绍了 ESP32-S3 系列产品中内置的硬件模块，包括概述、功能列表、硬件架构、编程指南、寄存器列表等信息。

本文档中的跳转

在本文档中实现跳转，请参考以下建议：

- 发布进度速览（下一页）罗列了本文档中的所有章节，您可以从这里快速跳转至某个具体章节。
- 您还可以通过文档左侧的**书签**，从文中的任何位置直接跳转至另一个章节。注意，本文档已设置默认打开**书签**功能，但一些 PDF 阅读器或浏览器会忽略此设置。因此，如果您无法找到**书签**功能，请尝试以下方法：
  - 在您的浏览器中安装 PDF 阅读器拓展；
  - 下载本文档，使用本地 PDF 阅读器进行浏览；
  - 配置您的 PDF 阅读器，使其默认打开**书签**功能。
- 大多数 PDF 阅读器均支持跳转功能，允许您借助按钮、菜单选项或快捷键进行跳转（**向上**、**向下**、**向前**、**向后**、**后退**、**前进**及**前往页面**）等。
- 此外，您还可以使用本文档内置的 **GoBack** 按钮（每页右上角）快速后退至跳转之前的位置。注意，本功能仅适用于 Acrobat 系列的 PDF 阅读器（比如 Acrobat Reader 和 Adobe DC）以及内置 Acrobat 系列 PDF 阅读器或拓展的浏览器（比如 Firefox）。

发布进度速览

No.	章节	最新进度	No.	章节	最新进度
第 I 卷：微处理器和主机			20	RSA 加速器 (RSA)	已发布
1	处理器指令拓展 (PIE)	已发布	21	HMAC 加速器 (HMAC)	已发布
2	超低功耗协处理器 (ULP-FSM, ULP-RISC-V)	已发布	22	数字签名 (DS)	已发布
3	通用 DMA 控制器 (GDMA)	已发布	23	片外存储器加密与解密 (XTS_AES)	已发布
第 II 卷：存储器组织结构			24	时钟毛刺检测	已发布
4	系统和存储器	已发布	25	随机数发生器 (RNG)	已发布
5	eFuse 控制器 (EFUSE)	已发布	**第 V 卷：通讯接口**		
第 III 卷：系统组件			26	UART 控制器 (UART)	已发布
6	IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)	已发布	27	I2C 控制器 (I2C)	已发布
7	复位和时钟	已发布	28	I2S 控制器 (I2S)	已发布
8	芯片 Boot 控制	已发布	29	LCD 与 Camera 控制器 (LCD_CAM)	已发布
9	中断矩阵 (INTERRUPT)	已发布	30	SPI 控制器 (SPI)	已发布
10	低功耗管理 (RTC_CNTL)	已发布	31	双线汽车接口 (TWAI®)	已发布
11	系统定时器 (SYSTIMER)	已发布	32	USB OTG (USB)	已发布
12	定时器组 (TIMG)	已发布	33	USB 串口/JTAG 控制器 (USB_SERIAL_JTAG)	已发布
13	看门狗定时器 (WDT)	已发布	34	SD/MMC 主机控制器 (SDHOST)	已发布
14	XTAL32K 看门狗定时器 (XTWDT)	已发布	35	LED PWM 控制器 (LEDC)	已发布
15	权限控制 (PMS)	已发布	36	电机控制脉宽调制器 (MCPWM)	已发布
16	World 控制器 (WCL)	已发布	37	红外遥控 (RMT)	已发布
17	系统寄存器 (SYSTEM)	已发布	38	脉冲计数控制器 (PCNT)	已发布
第 IV 卷：加密和安全组件			**第 VI 卷：模拟信号处理**		
18	SHA 加速器 (SHA)	已发布	39	片上传感器与模拟信号处理	已发布
19	AES 加速器 (AES)	已发布			

说明：

点击链接或扫描二维码确保您使用的是最新版本的文档：

https://www.espressif.com/documentation/esp32-s3_technical_reference_manual_cn.pdf

目录

I 微处理器和主机 38

1 处理器指令拓展 (PIE) 39
1.1 概述 39
1.2 主要特性 39
1.3 结构概述 39
1.3.1 向量寄存器组 40
1.3.2 ALU 41
1.3.3 QACC 累加寄存器 41
1.3.4 ACCX 累加寄存器 41
1.3.5 地址单元 41
1.4 符号介绍 41
1.4.1 比特及字节序 41
1.4.2 指令域定义 43
1.5 扩展指令集组件 44
1.5.1 寄存器 44
1.5.1.1 通用寄存器 45
1.5.1.2 特殊寄存器 46
1.5.2 快速 GPIO 端口 47
1.5.2.1 GPIO_OUT 47
1.5.2.2 GPIO_IN 47
1.5.3 数据格式及对齐 47
1.5.4 数据溢出及饱和处理 48
1.6 扩展指令简介 48
1.6.1 读内存指令 50
1.6.2 写内存指令 51
1.6.3 数据交换指令 52
1.6.4 运算指令 53
1.6.5 比较指令 55
1.6.6 按位逻辑操作指令 56
1.6.7 移位指令 56
1.6.8 FFT（快速傅立叶变换）专用指令 57
1.6.9 GPIO 控制指令 58
1.6.10 处理器控制指令 58
1.7 指令性能 60
1.7.1 数据冒险 60
1.7.2 资源冒险 68
1.7.3 控制冒险 69
1.8 扩展指令功能描述 70

2 超低功耗协处理器 (ULP-FSM, ULP-RISC-V) 295
2.1 概述 295

2.2 特性 295
2.3 编程流程 298
2.4 协处理器的睡眠和唤醒流程 298
2.5 ULP-FSM 300
2.5.1 特性 300
2.5.2 指令集 300
2.5.2.1 ALU - 算术与逻辑运算 301
2.5.2.2 ST - 存储数据至内存 303
2.5.2.3 LD - 从内存加载数据 306
2.5.2.4 JUMP - 跳转至绝对地址 306
2.5.2.5 JUMPR - 跳转至相对地址（基于 R0 寄存器判断） 307
2.5.2.6 JUMPS - 跳转至相对地址（基于阶段计数器寄存器判断） 307
2.5.2.7 HALT - 结束程序 308
2.5.2.8 WAKE - 唤醒芯片 308
2.5.2.9 WAIT - 等待若干个周期 309
2.5.2.10 TSENS - 对温度传感器进行测量 309
2.5.2.11 ADC - 对 ADC 进行测量 309
2.5.2.12 REG_RD - 从外设寄存器读取 310
2.5.2.13 REG_WR - 写入外设寄存器 311
2.6 ULP-RISC-V 311
2.6.1 特性 311
2.6.2 乘除法器 311
2.6.3 ULP-RISC-V 中断 312
2.6.3.1 概述 312
2.6.3.2 中断控制器 312
2.6.3.3 中断处理 313
2.6.3.4 中断相关指令 313
2.6.3.5 RTC 外设中断 314
2.7 RTC I2C 控制器 315
2.7.1 连接 RTC I2C 信号 315
2.7.2 配置 RTC I2C 控制器 315
2.7.3 使用 RTC I2C 316
2.7.3.1 I2C 指令编码格式 316
2.7.3.2 I2C_RD - I2C 读流程 316
2.7.3.3 I2C_WR - I2C 写流程 317
2.7.3.4 检测错误条件 318
2.7.4 RTC I2C 中断 318
2.8 地址映射 319
2.9 寄存器列表 319
2.9.1 ULP (ALWAYS_ON) 寄存器列表 319
2.9.2 ULP (RTC_PERI) 寄存器列表 320
2.9.3 RTC I2C (RTC_PERI) 寄存器列表 320
2.9.4 RTC I2C (I2C) 寄存器列表 320
2.10 寄存器 321
2.10.1 ULP (ALWAYS_ON) 寄存器 321
2.10.2 ULP (RTC_PERI) 寄存器 324

2.10.3 RTC I2C (RTC_PERI) 寄存器 328
2.10.4 RTC I2C (I2C) 寄存器 330

3 通用 DMA 控制器 (GDMA) 345
3.1 概述 345
3.2 特性 345
3.3 架构 346
3.4 功能描述 347
3.4.1 链表 347
3.4.2 外设到存储及存储到外设的数据传输 348
3.4.3 存储到存储数据传输 348
3.4.4 通道 Buffer 349
3.4.5 启动 GDMA 349
3.4.6 读链表 350
3.4.7 数据传输结束标志 350
3.4.8 访问内部 RAM 350
3.4.9 访问外部 RAM 351
3.4.10 访问外部 RAM 的权限管理 351
3.4.11 内部及外部 RAM 数据无缝访问 352
3.4.12 仲裁 352
3.5 GDMA 中断 353
3.6 编程流程 353
3.6.1 GDMA 时钟与复位配置流程 353
3.6.2 GDMA TX 通道配置流程 353
3.6.3 GDMA RX 通道配置流程 354
3.6.4 GDMA 存储器到存储器配置流程 354
3.7 寄存器列表 355
3.8 寄存器 361

II 存储器组织结构 382

4 系统和存储器 383
4.1 概述 383
4.2 主要特性 383
4.3 功能描述 384
4.3.1 地址映射 384
4.3.2 内部存储器 385
4.3.3 外部存储器 387
4.3.3.1 外部存储器地址映射 387
4.3.3.2 高速缓存 387
4.3.3.3 Cache 操作 388
4.3.4 GDMA 地址空间 389
4.3.5 模块/外设地址空间 390
4.3.5.1 模块/外设地址空间列表 390

5 eFuse 控制器 (EFUSE) 393

5.1 概述 393
5.2 主要特性 393
5.3 功能描述 393
5.3.1 结构 393
5.3.1.1 EFUSE_WR_DIS 398
5.3.1.2 EFUSE_RD_DIS 398
5.3.1.3 数据存储方式 398
5.3.2 烧写参数 400
5.3.3 用户读取参数 401
5.3.4 eFuse VDDQ 时序 403
5.3.5 硬件模块使用参数 403
5.3.6 中断 403
5.4 寄存器列表 404
5.5 寄存器 408

III 系统组件 449

6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX) 450
6.1 概述 450
6.2 特性 450
6.3 结构概览 451
6.4 通过 GPIO 交换矩阵的外设输入 452
6.4.1 概述 452
6.4.2 信号同步 452
6.4.3 功能描述 453
6.4.4 简单 GPIO 输入 454
6.5 通过 GPIO 交换矩阵的外设输出 454
6.5.1 概述 454
6.5.2 功能描述 455
6.5.3 简单 GPIO 输出 455
6.5.4 Sigma Delta 调制输出 (SDM) 456
6.5.4.1 功能描述 456
6.5.4.2 配置方法 456
6.6 IO MUX 的直接输入输出功能 457
6.6.1 概述 457
6.6.2 功能描述 457
6.7 RTC IO MUX 的低功耗性能和模拟输入输出功能 457
6.7.1 概述 457
6.7.2 低功耗性能描述 457
6.7.3 模拟功能描述 457
6.8 Light-sleep 模式管脚功能 458
6.9 管脚 Hold 特性 458
6.10 GPIO 管脚供电和电源管理 459
6.10.1 GPIO 管脚供电 459
6.10.2 电源管理 459
6.11 GPIO 交换矩阵外设信号列表 459

6.12 IO MUX 管脚功能列表 471
6.13 RTC IO MUX 管脚功能列表 472
6.14 寄存器列表 474
6.14.1 GPIO 交换矩阵寄存器列表 474
6.14.2 IO MUX 寄存器列表 475
6.14.3 SDM 寄存器列表 477
6.14.4 RTC IO MUX 寄存器列表 477
6.15 寄存器 478
6.15.1 GPIO 交换矩阵寄存器 478
6.15.2 IO MUX 寄存器 490
6.15.3 SDM 寄存器 492
6.15.4 RTC IO MUX 寄存器 493

7 复位和时钟 503
7.1 复位 503
7.1.1 概述 503
7.1.2 结构图 503
7.1.3 特性 503
7.1.4 功能描述 504
7.2 时钟 505
7.2.1 概述 505
7.2.2 结构图 506
7.2.3 特性 506
7.2.4 功能描述 507
7.2.4.1 CPU 时钟 507
7.2.4.2 外设时钟 507
7.2.4.3 Wi-Fi 和 Bluetooth LE 时钟 509
7.2.4.4 RTC 时钟 509

8 芯片 Boot 控制 510
8.1 概述 510
8.2 Boot 模式控制 511
8.3 ROM 日志打印控制 512
8.4 VDD_SPI 电压控制 513
8.5 JTAG 信号源控制 513

9 中断矩阵 (INTERRUPT) 515
9.1 概述 515
9.2 主要特性 515
9.3 功能描述 515
9.3.1 外部中断源 515
9.3.2 CPU 中断 520
9.3.3 分配外部中断源至 CPU*x* 外部中断 521
9.3.3.1 分配一个外部中断源 Source_Y 至 CPU*x* 外部中断 521
9.3.3.2 分配多个外部中断源 Source_Y*n* 至 CPU*x* 外部中断 521
9.3.3.3 关闭 CPU*x* 外部中断源 Source_Y 521

9.3.4 关闭 CPUx 的 NMI 类型中断 521
9.3.5 查询外部中断源当前的中断状态 522
9.4 寄存器列表 522
9.4.1 CPU0 中断寄存器列表 523
9.4.2 CPU1 中断寄存器列表 526
9.5 寄存器 531
9.5.1 CPU0 中断寄存器 531
9.5.2 CPU1 中断寄存器 536

10 低功耗管理 (RTC_CNTL) 541
10.1 概述 541
10.2 主要特性 541
10.3 功能描述 541
10.3.1 功耗管理单元 543
10.3.2 低功耗时钟 544
10.3.3 定时器 546
10.3.4 调压器 547
10.3.4.1 数字调压器 547
10.3.4.2 低功耗调压器 547
10.3.4.3 flash 调压器 548
10.3.4.4 欠压检测器 549
10.4 功耗模式管理 550
10.4.1 电源域 550
10.4.2 RTC 状态 551
10.4.3 预设功耗模式 552
10.4.4 唤醒源 553
10.4.5 拒绝睡眠 554
10.5 Retention DMA 554
10.6 RTC Boot 555
10.7 寄存器列表 557
10.8 寄存器 559

11 系统定时器 (SYSTIMER) 605
11.1 概述 605
11.2 特性 605
11.3 时钟源选择 606
11.4 功能描述 606
11.4.1 计数器 606
11.4.2 比较器和报警 607
11.4.3 同步操作 608
11.4.4 中断 608
11.5 编程示例 609
11.5.1 读取当前计数器的值 609
11.5.2 在单次报警模式下配置一次性报警 609
11.5.3 在周期报警模式下配置周期性报警 609
11.5.4 唤醒后时间补偿 610

11.6 寄存器列表 610
11.7 寄存器 612

12 定时器组 (TIMG) 625
12.1 概述 625
12.2 功能描述 626
12.2.1 16 位预分频器与时钟选择器 626
12.2.2 54 位时基计数器 626
12.2.3 报警产生 626
12.2.4 定时器重新加载 627
12.2.5 RTC 慢速时钟 (RTC_SLOW_CLK) 频率计算 628
12.2.6 中断 628
12.3 配置与使用 628
12.3.1 定时器用作简单时钟 628
12.3.2 定时器用于一次性报警 629
12.3.3 定时器用于周期性报警 629
12.3.4 RTC_SLOW_CLK 频率计算 630
12.4 寄存器列表 631
12.5 寄存器 633

13 看门狗定时器 (WDT) 643
13.1 概述 643
13.2 数字看门狗定时器 644
13.2.1 主要特性 644
13.2.2 功能描述 645
13.2.2.1 时钟源与 32 位计数器 645
13.2.2.2 阶段与超时动作 646
13.2.2.3 写保护 646
13.2.2.4 flash 引导保护 647
13.3 模拟看门狗定时器 647
13.3.1 主要特性 647
13.3.2 SWD 控制器 647
13.3.2.1 结构 647
13.3.2.2 工作流程 648
13.4 中断 648
13.5 寄存器 648

14 XTAL32K 看门狗定时器 (XTWDT) 649
14.1 主要特性 649
14.1.1 XTAL32K 看门狗定时器的中断及唤醒 649
14.1.2 BACKUP32K_CLK 649
14.2 功能描述 649
14.2.1 工作流程 650
14.2.2 BACKUP32K_CLK 实现原理 650
14.2.3 BACKUP32K_CLK 分频因子配置方法 650

15 权限控制 (PMS) 651
15.1 概述 651
15.2 主要特性 651
15.3 片内存储器的权限管理 652
15.3.1 ROM 的访问权限管理 652
15.3.1.1 地址范围 652
15.3.1.2 权限配置 652
15.3.2 SRAM 的权限管理 652
15.3.2.1 地址范围 653
15.3.2.2 Internal SRAM0 的权限配置 653
15.3.2.3 Internal SRAM1 权限配置 654
15.3.2.4 Internal SRAM2 权限配置 658
15.3.3 RTC 快速内存 (FAST Memory) 的权限管理 659
15.3.3.1 地址范围 659
15.3.3.2 权限配置 659
15.3.4 RTC 慢速内存 (SLOW Memory) 的权限管理 659
15.3.4.1 地址范围 659
15.3.4.2 权限配置 660
15.4 外设权限管理 660
15.4.1 外设空间权限控制 660
15.4.2 自定义地址段权限管理 662
15.5 片外存储器权限管理 663
15.5.1 外部存储器实地址空间划分 663
15.5.2 外部存储器的权限配置 664
15.5.3 GDMA 权限管理 664
15.6 非法访问与中断 665
15.6.1 IBUS 总线非法访问中断 666
15.6.2 DBUS 总线非法访问中断 666
15.6.3 外部存储器中断 666
15.6.4 GDMA 中断 667
15.6.5 PIF 外设总线中断 667
15.6.6 非字对齐访问检查 668
15.7 CPU VECBASE 寄存器保护 669
15.8 寄存器锁 670
15.9 寄存器列表 673
15.10 寄存器 678

16 World 控制器 (WCL) 768
16.1 概述 768
16.2 主要特性 768
16.3 功能描述 768
16.4 CPU 的世界切换 769
16.4.1 安全世界切换到非安全世界 769
16.4.2 非安全世界切换到安全世界 771
16.4.3 清除 write_buffer 771
16.5 世界切换记录表 772

16.5.1 世界切换记录表寄存器的组成 772
16.5.2 世界切换记录表寄存器的更新 773
16.5.3 世界切换记录表寄存器的读取 774
16.5.4 中断嵌套 775
16.5.4.1 处理方法 775
16.5.4.2 编程指南 775
16.6 NMI 中断屏蔽 776
16.7 寄存器列表 778
16.8 寄存器 779

17 系统寄存器 (SYSTEM) 787
17.1 概述 787
17.2 主要特性 787
17.3 功能描述 787
17.3.1 系统和存储器寄存器 787
17.3.1.1 内部存储器 787
17.3.1.2 片外存储器 788
17.3.1.3 RSA 存储器 788
17.3.2 时钟配置寄存器 789
17.3.3 中断信号寄存器 789
17.3.4 低功耗管理寄存器 789
17.3.5 外设时钟门控和复位寄存器 789
17.3.6 CPU 控制寄存器 791
17.4 寄存器列表 792
17.5 寄存器 793

IV 加密和安全组件 806

18 SHA 加速器 (SHA) 807
18.1 概述 807
18.2 主要特性 807
18.3 工作模式简介 807
18.4 功能描述 808
18.4.1 信息预处理 808
18.4.1.1 附加填充比特 808
18.4.1.2 信息解析 809
18.4.1.3 哈希初始值 (Initial Hash Value) 809
18.4.2 哈希运算流程 810
18.4.2.1 Typical SHA 模式下的运算流程 810
18.4.2.2 DMA-SHA 模式下的运算流程 812
18.4.3 信息摘要存储 813
18.4.4 中断 814
18.5 寄存器列表 814
18.6 寄存器 816

19 AES 加速器 (AES) 820

19.1 概述 820
19.2 主要特性 820
19.3 工作模式简介 820
19.4 Typical AES 工作模式 821
19.4.1 密钥、明文、密文 821
19.4.2 字节序 822
19.4.3 Typical AES 工作模式的流程 824
19.5 DMA-AES 工作模式 825
19.5.1 密钥、明文、密文 825
19.5.2 字节序 826
19.5.3 标准增量函数 826
19.5.4 块个数 827
19.5.5 初始向量 827
19.5.6 DMA-AES 工作模式的流程 827
19.6 存储器列表 828
19.7 寄存器列表 829
19.8 寄存器 830

20 RSA 加速器 (RSA) 834
20.1 概述 834
20.2 主要特性 834
20.3 功能描述 834
20.3.1 大数模幂运算 834
20.3.2 大数模乘运算 836
20.3.3 大数乘法运算 836
20.3.4 控制加速 837
20.4 存储器列表 838
20.5 寄存器列表 838
20.6 寄存器 839

21 HMAC 加速器 (HMAC) 843
21.1 主要特性 843
21.2 功能描述 843
21.2.1 上行模式 843
21.2.2 下行 JTAG 启动模式 844
21.2.3 下行数字签名模式 844
21.2.4 烧写 HMAC 密钥 844
21.2.5 HMAC 功能初始化 845
21.2.6 调用 HMAC 流程（详细说明） 845
21.3 HMAC 算法细节 847
21.3.1 附加填充比特 847
21.3.2 HMAC 算法结构 848
21.4 寄存器列表 850
21.5 寄存器 852

22 数字签名 (DS) 858

22.1 概述 858
22.2 主要特性 858
22.3 功能描述 858
22.3.1 概述 858
22.3.2 私钥运算子 859
22.3.3 软件需要做的准备工作 859
22.3.4 硬件工作流程 860
22.3.5 软件工作流程 861
22.4 存储器列表 862
22.5 寄存器列表 863
22.6 寄存器 864

23 片外存储器加密与解密 (XTS_AES) 867
23.1 概述 867
23.2 主要特性 867
23.3 模块结构 867
23.4 功能描述 868
23.4.1 XTS 算法 868
23.4.2 密钥 Key 868
23.4.3 目标空间 869
23.4.4 数据填充 869
23.4.5 手动加密模块 870
23.4.6 自动加密模块 870
23.4.7 自动解密模块 871
23.5 软件流程 871
23.6 寄存器列表 873
23.7 寄存器 874

24 时钟毛刺检测 877
24.1 概述 877
24.2 功能描述 877
24.2.1 时钟毛刺检测 877
24.2.2 复位 877

25 随机数发生器 (RNG) 878
25.1 概述 878
25.2 主要特性 878
25.3 功能描述 878
25.4 编程指南 879
25.5 寄存器列表 879
25.6 寄存器 879

V 通讯接口 880

26 UART 控制器 (UART) 881
26.1 概述 881

26.2 主要特性 881
26.3 UART 架构 882
26.4 功能描述 883
26.4.1 时钟与复位 883
26.4.2 UART RAM 884
26.4.3 波特率产生与检测 885
26.4.3.1 波特率产生 885
26.4.3.2 波特率检测 885
26.4.4 UART 数据帧 886
26.4.5 AT_CMD 字符格式 887
26.4.6 RS485 887
26.4.6.1 驱动控制 888
26.4.6.2 转换延时 888
26.4.6.3 总线侦听 888
26.4.7 IrDA 888
26.4.8 唤醒 889
26.4.9 回环功能 889
26.4.10 流控 890
26.4.10.1 硬件流控 890
26.4.10.2 软件流控 891
26.4.11 GDMA 模式 891
26.4.12 UART 中断 892
26.4.13 UCHI 中断 893
26.5 编程流程 893
26.5.1 寄存器类型 893
26.5.1.1 同步寄存器 893
26.5.1.2 静态寄存器 894
26.5.1.3 立即寄存器 895
26.5.2 具体步骤 895
26.5.2.1 UART*n* 模块初始化 896
26.5.2.2 UART*n* 通信配置 897
26.5.2.3 启动 UART*n* 897
26.6 寄存器列表 898
26.6.1 UART 寄存器列表 898
26.6.2 UHCI 寄存器列表 899
26.7 寄存器 901
26.7.1 UART 寄存器 901
26.7.2 UHCI 寄存器 919

27 I2C 控制器 (I2C) 935
27.1 概述 935
27.2 主要特性 935
27.3 I2C 架构 936
27.4 功能描述 938
27.4.1 时钟配置 938
27.4.2 滤除 SCL 和 SDA 噪声 938

27.4.3 SCL 时钟拉伸 938
27.4.4 SCL 空闲时产生 SCL 脉冲 939
27.4.5 同步 939
27.4.6 漏级开路输出 940
27.4.7 时序参数配置 940
27.4.8 超时控制 942
27.4.9 指令配置 942
27.4.10 TX/RX RAM 数据存储 943
27.4.11 数据转换 944
27.4.12 寻址模式 944
27.4.13 10 位寻址的读写标志位检查 944
27.4.14 启动控制器 945
27.5 编程示例 945
27.5.1 I2C 主机写入从机，7 位寻址，单次命令序列 945
27.5.1.1 场景介绍 946
27.5.1.2 配置示例 946
27.5.2 I2C 主机写入从机，10 位寻址，单次命令序列 947
27.5.2.1 场景介绍 948
27.5.2.2 配置示例 948
27.5.3 I2C 主机写入从机，7 位双地址寻址，单次命令序列 949
27.5.3.1 场景介绍 949
27.5.3.2 配置示例 950
27.5.4 I2C 主机写入从机，7 位寻址，多次命令序列 950
27.5.4.1 场景介绍 951
27.5.4.2 配置示例 952
27.5.5 I2C 主机读取从机，7 位寻址，单次命令序列 953
27.5.5.1 场景介绍 953
27.5.5.2 配置示例 953
27.5.6 I2C 主机读取从机，10 位寻址，单次命令序列 954
27.5.6.1 场景介绍 955
27.5.6.2 配置示例 955
27.5.7 I2C 主机读取从机，7 位双寻址，单次命令序列 956
27.5.7.1 场景介绍 957
27.5.7.2 配置示例 957
27.5.8 I2C 主机读取从机，7 位寻址，多次命令序列 958
27.5.8.1 场景介绍 959
27.5.8.2 配置示例 960
27.6 中断 961
27.7 寄存器列表 963
27.8 寄存器 965

28 I2S 控制器 (I2S) 983
28.1 概述 983
28.2 术语 983
28.3 特性 984
28.4 系统架构 985

28.5 I2S*n* 模块支持的音频协议 986
28.5.1 TDM Philips 标准模式 987
28.5.2 TDM MSB 对齐标准模式 987
28.5.3 TDM PCM 标准模式 988
28.5.4 PDM 标准模式 988
28.6 TX/RX 模块时钟 989
28.7 I2S*n* 模块复位 991
28.8 I2S*n* 主/从机模式 991
28.8.1 主/从机发送模式 991
28.8.2 主/从机接收模式 992
28.9 发送数据 992
28.9.1 数据格式控制 992
28.9.1.1 通道有效数据位宽 992
28.9.1.2 通道有效数据字节序 993
28.9.1.3 A 率/μ 率压缩/解压缩 993
28.9.1.4 通道发送数据位宽 994
28.9.1.5 通道数据比特顺序 994
28.9.2 通道模式控制 994
28.9.2.1 TDM 模式下 I2S*n* 通道模式 995
28.9.2.2 PDM 模式下 I2S*n* 通道模式 995
28.10 接收数据 997
28.10.1 通道模式控制 997
28.10.1.1 TDM 模式下 I2S*n* 通道模式 998
28.10.1.2 PDM 模式下 I2S*n* 通道模式 998
28.10.2 数据格式控制 999
28.10.2.1 通道数据比特顺序 999
28.10.2.2 通道储存数据位宽 999
28.10.2.3 通道接收数据位宽 999
28.10.2.4 通道储存数据字节序 1000
28.10.2.5 A 率/μ 率压缩/解压缩 1000
28.11 软件配置流程 1000
28.11.1 软件配置 I2S*n* 发送流程 1000
28.11.2 软件配置 I2S*n* 接收流程 1001
28.12 I2S*n* 中断 1002
28.13 寄存器列表 1002
28.14 寄存器 1003

29 LCD 与 Camera 控制器 (LCD_CAM) 1020
29.1 概述 1020
29.2 特性 1020
29.3 功能描述 1020
29.3.1 功能框图 1020
29.3.2 信号描述 1021
29.3.3 LCD_CAM 模块时钟 1022
29.3.3.1 LCD 时钟 1022
29.3.3.2 Camera 时钟 1023

29.3.4 LCD_CAM 模块复位 1024
29.3.5 LCD_CAM 数据格式控制 1024
29.3.5.1 LCD 数据格式控制 1024
29.3.5.2 Camera 数据格式控制 1025
29.3.6 YUV-RGB 数据格式转换 1026
29.3.6.1 YUV 时序 1026
29.3.6.2 格式转换模块配置流程 1027
29.4 软件配置流程 1028
29.4.1 软件配置 LCD（RGB 格式）发送流程 1028
29.4.2 软件配置 LCD（I8080/MOTO6800 格式）发送流程 1030
29.4.3 软件配置 Camera 接收流程 1031
29.5 LCD_CAM 中断 1032
29.6 寄存器列表 1033
29.7 寄存器 1033

30 SPI 控制器 (SPI) 1047
30.1 概述 1047
30.2 术语 1047
30.3 特性 1048
30.4 架构概览 1050
30.5 功能描述 1050
30.5.1 数据模式 1050
30.5.2 FSPI 总线信号和 SPI3 总线信号描述 1051
30.5.3 数据位读/写顺序控制 1054
30.5.4 传输方式 1056
30.5.5 CPU 控制的数据传输 1056
30.5.5.1 CPU 控制的主机模式 1056
30.5.5.2 CPU 控制的从机模式 1057
30.5.6 DMA 控制的数据传输 1057
30.5.6.1 GDMA 配置 1058
30.5.6.2 GDMA TX/RX Buffer 长度控制 1059
30.5.7 GP-SPI 主机模式和从机模式下的数据流控制 1059
30.5.7.1 GP-SPI 功能块图 1059
30.5.7.2 主机模式下的数据流控制 1060
30.5.7.3 从机模式下的数据流控制 1061
30.5.8 GP-SPI 主机模式 1061
30.5.8.1 主机模式状态机 1062
30.5.8.2 状态控制和位模式控制寄存器 1064
30.5.8.3 主机全双工通信（仅支持 1-bit 模式） 1067
30.5.8.4 主机半双工通信（支持 1/2/4/8-bit 模式） 1068
30.5.8.5 DMA 控制的分段配置传输 1070
30.5.9 GP-SPI 从机模式 1074
30.5.9.1 可配置的通信格式 1075
30.5.9.2 半双工通信支持的 CMD 值 1076
30.5.9.3 从机单次传输和从机连读传输 1078
30.5.9.4 配置从机单次传输模式 1078

30.5.9.5 配置半双工模式下从机连续传输 1079
30.5.9.6 配置全双工模式下从机连续传输 1080
30.6 CS 建立时间和保持时间控制 1080
30.7 GP-SPI 时钟控制 1081
30.7.1 时钟相位和极性 1082
30.7.2 主机模式下的时钟控制 1083
30.7.3 从机模式下的时钟控制 1084
30.8 GP-SPI 时序补偿 1084
30.9 GP-SPI2 和 GP-SPI3 功能差异 1085
30.10 中断 1086
30.11 寄存器列表 1089
30.12 寄存器 1090

31 双线汽车接口 (TWAI®) 1121
31.1 概述 1121
31.2 主要特性 1121
31.3 功能性协议 1122
31.3.1 TWAI 性能 1122
31.3.2 TWAI 报文 1122
31.3.2.1 数据帧和远程帧 1123
31.3.2.2 错误帧和过载帧 1124
31.3.2.3 帧间距 1127
31.3.3 TWAI 错误 1127
31.3.3.1 错误类型 1127
31.3.3.2 错误状态 1127
31.3.3.3 错误计数 1128
31.3.4 TWAI 位时序 1129
31.3.4.1 名义位 1129
31.3.4.2 硬同步与再同步 1129
31.4 结构概述 1130
31.4.1 寄存器模块 1131
31.4.2 位流处理器 1131
31.4.3 错误管理逻辑 1131
31.4.4 位时序逻辑 1132
31.4.5 接收滤波器 1132
31.4.6 接收 FIFO 1132
31.5 功能描述 1132
31.5.1 模式 1132
31.5.1.1 复位模式 1132
31.5.1.2 操作模式 1132
31.5.2 位时序 1133
31.5.3 中断管理 1133
31.5.3.1 接收中断 (RXI) 1134
31.5.3.2 发送中断 (TXI) 1134
31.5.3.3 错误报警中断 (EWI) 1134
31.5.3.4 数据溢出中断 (DOI) 1134

31.5.3.5 被动错误中断 (TXI) 1134
31.5.3.6 仲裁丢失中断 (ALI) 1135
31.5.3.7 总线错误中断 (BEI) 1135
31.5.3.8 总线状态中断 (BSI) 1135
31.5.4 发送缓冲器与接收缓冲器 1135
31.5.4.1 缓冲器概述 1135
31.5.4.2 帧信息 1136
31.5.4.3 帧标识符 1136
31.5.4.4 帧数据 1137
31.5.5 接收 FIFO 和数据溢出 1137
31.5.6 接收滤波器 1138
31.5.6.1 单滤波模式 1138
31.5.6.2 双滤波模式 1139
31.5.7 错误管理 1139
31.5.7.1 错误报警限制 1140
31.5.7.2 被动错误 1141
31.5.7.3 离线状态与离线恢复 1141
31.5.8 错误捕捉 1141
31.5.9 仲裁丢失捕捉 1142
31.6 寄存器列表 1144
31.7 寄存器 1145

32 USB OTG (USB) 1157
32.1 概述 1157
32.2 特性 1157
32.2.1 通用特性 1157
32.2.2 设备模式 (Device mode) 特性 1157
32.2.3 主机模式 (Host mode) 特性 1158
32.3 功能描述 1158
32.3.1 控制器内核与接口 1158
32.3.2 存储器布局 1159
32.3.2.1 控制 & 状态寄存器 (CSR) 1159
32.3.2.2 FIFO 访问 1160
32.3.3 FIFO 和队列组织 1160
32.3.3.1 主机模式 FIFO 和队列 1160
32.3.3.2 设备模式 FIFO 1161
32.3.4 中断层次结构 1161
32.3.5 DMA 模式和 Slave 模式 1162
32.3.5.1 Slave 模式 1162
32.3.5.2 缓冲 DMA 模式 1163
32.3.5.3 Scatter/Gather DMA 模式 1163
32.3.6 事务和传输级操作 1164
32.3.6.1 DMA 模式下的事务和传输级操作 1164
32.3.6.2 Slave 模式下的事务和传输级操作 1164
32.4 OTG 1165
32.4.1 OTG 接口 1166

32.4.2 ID 管脚检测 1167
32.4.3 会话请求协议 (SRP) 1167
32.4.3.1 A 设备 SRP 1167
32.4.3.2 B 设备 SRP 1167
32.4.4 主机协商协议 (HNP) 1168
32.4.4.1 A 设备 HNP 1168
32.4.4.2 B 设备 HNP 1169
32.5 寄存器 1170

33 USB 串口/JTAG 控制器 (USB_SERIAL_JTAG) 1171
33.1 概述 1171
33.2 特性 1171
33.3 功能描述 1173
33.3.1 USB 串口/JTAG 主机连接 1173
33.3.2 CDC-ACM USB 接口描述 1174
33.3.3 CDC-ACM 固件接口描述 1175
33.3.4 USB-JTAG 接口 1175
33.3.5 JTAG 命令处理器 1176
33.3.6 USB-JTAG 接口：CMD_REP 使用示例 1176
33.3.7 USB-JTAG 接口：响应捕捉单元 1177
33.3.8 USB-JTAG 接口：控制传输请求 1177
33.4 操作建议 1178
33.4.1 内部/外部 PHY 选择 1178
33.4.2 运行操作 1179
33.5 寄存器列表 1181
33.6 寄存器 1182

34 SD/MMC 主机控制器 (SDHOST) 1195
34.1 概述 1195
34.2 主要特性 1195
34.3 SD/MMC 外部接口信号 1195
34.4 功能描述 1197
34.4.1 SD/MMC 主机控制器结构 1197
34.4.1.1 总线接口单元 (BIU) 1198
34.4.1.2 卡接口单元 (CIU) 1198
34.4.2 命令通路 1198
34.4.3 数据通路 1199
34.4.3.1 数据发送 1199
34.4.3.2 数据接收 1200
34.5 CIU 操作的软件限制 1200
34.6 收发数据 RAM 1201
34.6.1 TX RAM 模块 1201
34.6.2 RX RAM 模块 1201
34.7 DMA 链表环 1202
34.8 DMA 链表结构 1202
34.9 初始化 1204

34.9.1 DMA 控制器初始化 1204
34.9.2 DMA 控制器数据发送初始化 1204
34.9.3 DMA 控制器数据接收初始化 1205
34.10 时钟相位选择 1205
34.11 中断 1206
34.12 寄存器列表 1207
34.13 寄存器 1209

35 LED PWM 控制器 (LEDC) 1233
35.1 主要特性 1233
35.2 功能描述 1233
35.2.1 架构 1233
35.2.2 定时器 1234
35.2.2.1 时钟源 1234
35.2.2.2 时钟分频器配置 1234
35.2.2.3 14 位计数器 1235
35.2.3 PWM 生成器 1236
35.2.4 占空比渐变 1237
35.2.5 中断 1238
35.3 寄存器列表 1239
35.4 寄存器 1241

36 电机控制脉宽调制器 (MCPWM) 1247
36.1 概述 1247
36.2 主要特性 1247
36.3 模块 1249
36.3.1 概述 1249
36.3.1.1 预分频器模块 1249
36.3.1.2 定时器模块 1249
36.3.1.3 操作器模块 1250
36.3.1.4 故障检测模块 1251
36.3.1.5 捕获模块 1252
36.3.2 PWM 定时器模块 1252
36.3.2.1 PWM 定时器模块的配置 1252
36.3.2.2 PWM 定时器工作模式和定时事件生成 1252
36.3.2.3 PWM 定时器影子寄存器 1256
36.3.2.4 PWM 定时器同步和锁相 1257
36.3.3 PWM 操作器模块 1257
36.3.3.1 PWM 生成器模块 1257
36.3.3.2 死区生成器模块 1267
36.3.3.3 PWM 载波模块 1271
36.3.3.4 故障处理器模块 1274
36.3.4 捕获模块 1275
36.3.4.1 介绍 1275
36.3.4.2 捕获定时器 1275
36.3.4.3 捕获通道 1276

36.4 寄存器列表 1277
36.5 寄存器 1280

37 红外遥控 (RMT) 1330
37.1 概述 1330
37.2 特性 1330
37.3 功能描述 1330
37.3.1 架构 1331
37.3.2 RAM 1331
37.3.2.1 RAM 结构 1331
37.3.2.2 RAM 使用说明 1332
37.3.2.3 RAM 访问方式 1333
37.3.3 时钟 1333
37.3.4 发射器 1334
37.3.4.1 普通发送模式 1334
37.3.4.2 乒乓发送模式 1334
37.3.4.3 发送加载波 1334
37.3.4.4 持续发送模式 1335
37.3.4.5 多通道同时发送 1335
37.3.5 接收器 1335
37.3.5.1 普通接收模式 1335
37.3.5.2 乒乓接收模式 1336
37.3.5.3 接收滤波 1336
37.3.5.4 接收去载波 1336
37.3.6 配置参数更新 1336
37.4 中断 1337
37.5 寄存器列表 1338
37.6 寄存器 1340

38 脉冲计数控制器 (PCNT) 1353
38.1 主要特性 1353
38.2 功能描述 1354
38.3 应用实例 1356
38.3.1 通道 0 独自递增计数 1356
38.3.2 通道 0 独自递减计数 1357
38.3.3 通道 0 和通道 1 同时递增计数 1357
38.4 寄存器列表 1359
38.5 寄存器 1360

VI 模拟信号处理 1366

39 片上传感器与模拟信号处理 1367
39.1 概述 1367
39.2 电容式触摸传感器 1367
39.2.1 术语 1367
39.2.2 概述 1367

39.2.3 主要特性 1368
39.2.4 电容触摸管脚 1369
39.2.5 触摸传感器工作原理和工作信号 1370
39.2.6 Touch FSM 1371
39.2.6.1 测量过程 1372
39.2.6.2 测量操作的触发源 1372
39.2.6.3 SCAN 模式 1372
39.2.7 触摸检测 1373
39.2.7.1 采样值 1373
39.2.7.2 硬件触摸检测 1374
39.2.8 噪声检测 1375
39.2.9 接近模式 1375
39.2.10 防潮功能和遇水保护功能 1376
39.2.10.1 防潮功能 1376
39.2.10.2 遇水保护功能 1376
39.3 SAR ADC 1376
39.3.1 概述 1376
39.3.2 主要特性 1377
39.3.3 SAR ADC 架构 1378
39.3.4 输入信号 1380
39.3.5 ADC 转换和衰减 1381
39.3.6 RTC ADC 控制器 1381
39.3.7 DIG ADC 控制器 1382
39.3.7.1 DIG ADC 控制器时钟 1382
39.3.7.2 DMA 支持 1382
39.3.7.3 DIG ADC FSM 1382
39.3.7.4 样式表结构 1383
39.3.7.5 多通道扫描配置示例 1384
39.3.7.6 DMA 数据格式 1384
39.3.7.7 ADC 滤波器 1385
39.3.7.8 阈值监控 1385
39.3.8 SAR ADC2 仲裁器 1385
39.4 温度传感器 1386
39.4.1 概述 1386
39.4.2 主要特性 1386
39.4.3 功能描述 1387
39.5 中断 1388
39.6 寄存器列表 1388
39.6.1 SENSOR (ALWAYS_ON) 寄存器列表 1388
39.6.2 SENSOR (RTC_PERI) 寄存器列表 1389
39.6.3 SENSOR (DIG_PERI) 寄存器列表 1390
39.7 寄存器 1391
39.7.1 SENSOR (ALWAYS_ON) 寄存器 1391
39.7.2 SENSOR (RTC_PERI) 寄存器 1397
39.7.3 SENSOR (DIG_PERI) 寄存器 1415

VII 附录 1426

相关文档和资源 1427

词汇列表 1428
外设相关词汇 1428
寄存器相关缩写 1428
寄存器的访问类型 1429

如何配置寄存器的保留域 1431
概述 1431
如何配置保留域 1431

中断配置寄存器 1432

修订历史 1433

表格

1.4-1 指令域名及含义 44
1.5-1 ESP32-S3 扩展指令集寄存器集合 45
1.5-2 数据格式及对齐 47
1.6-1 指令简介 49
1.6-2 读内存指令 51
1.6-3 写内存指令 52
1.6-4 数据交换指令 52
1.6-5 向量加法运算指令 53
1.6-6 运算指令 53
1.6-7 运算指令 54
1.6-8 运算指令 54
1.6-9 运算指令 55
1.6-10 运算指令 55
1.6-11 比较指令 56
1.6-12 按位逻辑操作指令 56
1.6-13 移位指令 56
1.6-14 蝶形运算指令 57
1.6-15 比特反转指令 57
1.6-16 实数 FFT 指令 58
1.6-17 GPIO 控制指令 58
1.7-1 5 级 Xtensa 处理器流水线 60
1.7-2 扩展指令流水级数 61

2.2-1 超低功耗协处理器特性比较 297
2.5-1 对寄存器数值的 ALU 运算 302
2.5-2 对指令立即值的 ALU 运算 302
2.5-3 对阶段计数器寄存器的 ALU 运算 303
2.5-4 数据存储格式-地址自增模式 304
2.5-5 数据存储格式-地址指定模式 306
2.5-6 ADC 指令的输入信号 310
2.6-1 乘除法指令效率 312
2.6-2 ULP-RISC-V 中断列表 312
2.6-3 ULP-RISC-V 的中断寄存器 312
2.6-4 ULP-RISC-V 的外设中断列表 315
2.8-1 地址映射 319
2.8-2 ULP 协处理器可访问的外设寄存器 319

3.4-1 配置寄存器与外设选择关系表 348
3.4-2 访问内部 RAM 的链表描述符参数对齐要求 350
3.4-3 访问外部 RAM 的链表描述符参数对齐要求 351
3.4-4 配置寄存器、块大小和对齐方式的关系表 351

4.3-1 内部存储器地址映射 385
4.3-2 外部存储器地址映射 387

4.3-3 模块/外设地址空间映射表 390

5.3-1 BLOCK0 参数 394
5.3-2 密钥用途数值对应的含义 397
5.3-3 BLOCK1-10 参数 397
5.3-4 寄存器信息 402
5.3-5 VDDQ 默认时序参数配置 403

6.8-1 IO MUX Light-sleep 管脚功能控制寄存器 458
6.11-1 GPIO 交换矩阵外设信号 461
6.12-1 IO MUX 管脚功能 471
6.13-1 RTC IO MUX 管脚的 RTC 功能 473
6.13-2 RTC IO MUX 管脚模拟功能 473

7.1-1 复位源 505
7.2-1 CPU_CLK 时钟源选择 507
7.2-2 CPU_CLK 时钟频率 507
7.2-3 外设时钟 508
7.2-4 APB_CLK 时钟 509
7.2-5 CRYPTO_PWM_CLK 时钟 509

8.1-1 Strapping 管脚默认上拉/下拉 510
8.2-1 系统启动模式 511
8.3-1 UART0 的 ROM 日志打印控制 512
8.3-2 USB Serial/JTAG 控制器的 ROM 日志打印控制 513
8.5-1 JTAG 信号源控制 514

9.3-1 CPU 外部中断配置寄存器、外部中断状态寄存器、外部中断源 517
9.3-2 CPU 中断 520

10.3-1 低功耗时钟 546
10.3-2 RTC 定时器的触发条件 546
10.4-1 RTC 状态转换 552
10.4-2 预设功耗模式 552
10.4-3 唤醒源 553

11.4-1 UNIT*n* 配置控制位 607
11.4-2 报警触发条件 608
11.4-3 同步操作 608

12.2-1 可逆计数器向上计数时的报警触发场景 627
12.2-2 可逆计数器向下计数时的报警触发场景 627

15.3-1 ROM 地址范围 652
15.3-2 ROM 的权限配置 652
15.3-3 SRAM Block 地址范围 653
15.3-4 Internal SRAM0 的使用权限配置 653
15.3-5 Internal SRAM0 的访问权限配置 654
15.3-6 Internal SRAM1 的区域分割 655

15.3-7 Internal SRAM1 的访问权限控制 – 指令空间 657
15.3-8 Internal SRAM1 的访问权限控制 – 数据空间 657
15.3-9 Internal SRAM2 的使用权限配置 658
15.3-10 Internal SRAM2 的访问权限配置 658
15.3-11 地址范围 659
15.3-12 RTC 快速内存的分割 659
15.3-13 RTC 快速内存的权限配置 659
15.3-14 地址范围 660
15.3-15 RTC 慢速内存的分割 660
15.3-16 RTC 慢速内存的权限管理 660
15.4-1 外设权限管理寄存器 661
15.4-2 针对地址段的权限管理 663
15.5-1 外部存储器的区域范围配置 664
15.5-2 外部存储器的权限配置 664
15.5-3 外部 SRAM 的区域范围配置（DMA 访问） 665
15.5-4 外部 SRAM 区域的权限配置 665
15.6-1 iBUS 总线的非法访问中断寄存器 666
15.6-2 dBUS 总线的非法访问中断寄存器 666
15.6-3 外部存储器的非法访问中断寄存器 667
15.6-4 DMA 非法访问中断寄存器 667
15.6-5 PIF 总线的非法访问中断寄存器 668
15.6-6 非字对齐访问外设情况汇总 668
15.6-7 非字对齐中断寄存器 669
15.8-1 寄存器锁保护配置 670

17.3-1 内部存储器控制位 788
17.3-2 外设时钟门控与复位控制位 789

18.3-1 工作模式选择 808
18.3-2 运算标准选择 808
18.4-4 不同运算标准信息摘要的寄存器占用情况 814

19.3-1 工作模式 821
19.3-2 密钥长度和加解密方向 821
19.4-1 状态返回值 821
19.4-2 Typical AES 文本字节序 822
19.4-3 AES-128 密钥字节序 822
19.4-4 AES-256 密钥字节序 823
19.5-1 块模式选择 825
19.5-2 状态返回值 825
19.5-3 TEXT-PADDING 826
19.5-4 DMA AES 存储字节序 826

20.3-1 加速效果 838

21.2-1 HMAC 功能及配置数值 844

23.4-1 根据 Key_A、Key_B、Key_C 生成 Key 值 869

23.4-2 目标空间与寄存器堆的映射关系 870

26.5-1 UART*n* 同步寄存器 894
26.5-2 UART*n* 静态寄存器 895

27.4-1 I2C 同步寄存器 939

28.4-1 模块信号描述 986
28.9-1 通道有效数据位宽控制 993
28.9-2 通道有效数据字节序控制 993
28.9-3 PDM 模式下 I2S*n* 取数逻辑 996
28.9-4 PDM 模式下 I2S*n* 通道模式控制 996
28.9-5 PCM 转 PDM 输出模式 996
28.10-1 PDM 转 PCM 输入模式 999
28.10-2 通道储存数据位宽控制 999
28.10-3 通道储存数据字节序控制 1000

29.3-1 信号描述 1021
29.3-2 LCD 数据格式控制 1025
29.3-3 Camera 数据格式控制 1026
29.3-4 转换模式配置 1027

30.5-1 GP-SPI2 和 GP-SPI3 支持的数据模式 1050
30.5-2 FSPI/SPI3 总线信号功能描述 1051
30.5-3 各种 SPI 模式下使用到的 FSPI 总线信号 1052
30.5-4 各种 SPI 模式下使用到的 SPI3 总线信号 1053
30.5-5 GP-SPI 主机模式和从机模式下的数据位控制 1055
30.5-6 主机模式和从机模式下支持的传输方式 1056
30.5-7 GP-SPI 从机模式下数据传输中断触发条件 1058
30.5-8 1/2/4/8-bit 模式下状态控制寄存器 1065
30.5-9 命令值的发送顺序 1067
30.5-10 地址值的发送顺序 1067
30.5-11 CONF 阶段 BM 位图 1072
30.5-12 传输事务 *i* 中 CONF buffer*i* 配置示例 1074
30.5-13 BM 位图与待更新的寄存器 1074
30.5-14 GP-SPI 从机 SPI 模式支持的 CMD 值 1077
30.5-14 GP-SPI 从机 SPI 模式支持的 CMD 值 1078
30.5-15 GP-SPI 从机 QPI 支持的 CMD 值 1078
30.7-1 主机模式下的时钟相位和极性配置 1083
30.7-2 从机模式下的时钟相位和极性配置 1084
30.9-1 对 GP-SPI3 无效的字段 1086
30.10-1 GP-SPI 主机模式中断表 1087
30.10-2 GP-SPI 从机模式中断表 1088

31.3-1 SFF 和 EFF 中的数据帧和远程帧 1124
31.3-2 错误帧 1125
31.3-3 过载帧 1125
31.3-4 帧间距 1127

31.3-5 名义位时序中包含的段 1129
31.5-1 TWAI_BUS_TIMING_0_REG 的 bit 信息 (0x18) 1133
31.5-2 TWAI_BUS_TIMING_1_REG 的 bit (0x1c) 1133
31.5-3 SFF 与 EFF 的缓冲器布局 1135
31.5-4 TX/RX 帧信息 (SFF/EFF)； TWAI 地址 0x40 1136
31.5-5 TX/RX 标识符 1 (SFF); TWAI 地址 0x44 1136
31.5-6 TX/RX 标识符 2 (SFF); TWAI 地址 0x48 1136
31.5-7 TX/RX 标识符 1 (EFF); TWAI 地址 0x44 1137
31.5-8 TX/RX 标识符 2 (EFF); TWAI 地址 0x48 1137
31.5-9 TX/RX 标识符 3 (EFF); TWAI 地址 0x4c 1137
31.5-10 TX/RX 标识符 4 (EFF); TWAI 地址 0x50 1137
31.5-11 TWAI_ERR_CODE_CAP_REG 中的位信息 (0x30) 1141
31.5-12 SEG.4 - SEG.0 的位信息 1142
31.5-13 TWAI_ARB LOST CAP_REG 中的位信息 (0x2c) 1143

32.3-1 Slave 模式下的 IN 和 OUT 事务级操作 1165
32.4-1 UTMI OTG 接口 1166

33.3-1 标准 CDC-ACM 控制请求 1174
33.3-2 CDC-ACM 中 RTS 和 DTR 的设置 1175
33.3-3 半字节中的命令 1176
33.3-4 USB-JTAG 控制请求 1177
33.3-5 JTAG 功能描述符 1178
33.4-1 内部/外部 PHY 选择与相应 eFuse 配置 1179
33.4-2 USB-OTG Download 下载模式下芯片初始化后 IO 焊盘状态 1179
33.4-3 复位芯片进入下载模式 1180
33.4-4 复位 SoC 进行启动 1180

34.3-1 SD/MMC 信号描述 1197
34.8-1 DES0 单元描述 1203
34.8-2 DES1 单元描述 1203
34.8-3 DES2 单元描述 1204
34.8-4 DES3 单元描述 1204
34.10-1 SDHOST 时钟相位选择 1206

35.2-1 常用配置频率及精度 1236

36.3-1 操作器模块的配置参数 1251
36.3-2 PWM 生成器中的所有定时事件 1258
36.3-3 PWM 定时器递增计数时，定时事件的优先级 1259
36.3-4 PWM 定时器递减计数时，定时事件的优先级 1259
36.3-5 控制死区时间生成器开关的字段 1268
36.3-6 死区生成器的典型操作模式 1269

37.3-1 更新配置参数 1336

38.2-1 控制信号为低电平时输入脉冲信号上升沿的计数模式 1355
38.2-2 控制信号为高电平时输入脉冲信号上升沿的计数模式 1355

38.2-3 控制信号为低电平时输入脉冲信号下降沿的计数模式 1355
38.2-4 控制信号为高电平时输入脉冲信号下降沿的计数模式 1355

39.2-1 ESP32-S3 电容式触摸传感器的管脚 1369
39.2-2 平滑触摸数据算法 1374
39.2-3 基准数据算法 1374
39.2-4 噪声算法 1374
39.2-5 迟滞算法 1375
39.3-1 SAR ADC 的信号输入 1380
39.4-1 温度测量范围与温度偏移 1388

39.7-4 ENA/RAW/ST 寄存器的配置 1432

插图

1.3-1 PIE 内部描述 (MAC) 40
1.4-1 指令 EE.ZERO.QACC 小端比特序 42
1.4-2 指令 EE.ZERO.QACC 大端比特序 42
1.4-3 指令 EE.ZERO.QACC 小端字节序 42
1.4-4 指令 EE.ZERO.QACC 大端字节序 42
1.7-1 指令操作数依赖导致内部锁 61
1.7-2 硬件资源冒险 69
1.7-3 控制冒险 69

2.1-1 超低功耗协处理器概图 295
2.2-1 超低功耗协处理器基本架构 297
2.3-1 编程流程图 298
2.4-1 协处理器睡眠和唤醒流程 299
2.4-2 ULP 程序框图 300
2.5-1 ULP-FSM 协处理器的指令格式 301
2.5-2 指令类型 - 对寄存器数值的 ALU 运算 301
2.5-3 指令类型 - 对指令立即值的 ALU 运算 302
2.5-4 指令类型 - 对阶段计数器寄存器的 ALU 运算 303
2.5-5 指令类型 - ST 303
2.5-6 指令类型 - 地址自增模式的基地址偏移 (ST-OFFSET) 304
2.5-7 指令类型 - 地址自增模式的数据存储 (ST-AUTO-DATA) 304
2.5-8 MEM[Rdst + Offset] 写全字 305
2.5-9 指令类型 - 指定地址模式的数据存储 305
2.5-10 指令类型 - LD 306
2.5-11 指令类型 - JUMP 306
2.5-12 指令类型 - JUMPR 307
2.5-13 指令类型 - JUMPS 307
2.5-14 指令类型 - HALT 308
2.5-15 指令类型 - WAKE 308
2.5-16 指令类型 - WAIT 309
2.5-17 指令类型 - TSENS 309
2.5-18 指令类型 - ADC 309
2.5-19 指令类型 - REG_RD 310
2.5-20 指令类型 - REG_WR 311
2.6-1 标准 R-type 指令格式 313
2.6-2 中断指令 - getq rd, qs 313
2.6-3 中断指令 - setq qd, rs 314
2.6-4 中断指令 - retirq 314
2.6-5 中断指令 - Maskirq rd rs 314
2.7-1 I2C 读操作 317
2.7-2 I2C 写操作 318

3.1-1 具有 GDMA 功能的模块和 GDMA 通道 345
3.3-1 GDMA 引擎的架构 346

3.4-1 链表结构图 347
3.4-2 通道 Buffer 示意图 349
3.4-3 链表关系图 349
3.4-4 外部 RAM 的权限区域划分 352

4.2-1 系统结构与地址映射结构 384
4.3-1 Cache 系统框图 388
4.3-2 具有 GDMA 功能的外设 390

5.3-1 移位寄存器电路图（前 32 字节） 399
5.3-2 移位寄存器电路图（后 12 字节） 399

6.3-1 IO MUX、RTC IO MUX 和 GPIO 交换矩阵结构框图 451
6.3-2 焊盘内部结构 452
6.4-1 GPIO 输入经 APB 时钟上升沿或下降沿同步 453
6.4-2 GPIO 输入信号滤波时序图 453

7.1-1 四种复位等级 503
7.2-1 系统时钟 506

9.2-1 中断矩阵结构图 516

10.3-1 低功耗管理原理图 542
10.3-2 电源管理单元的主要工作流程 544
10.3-3 RTC 时钟 545
10.3-4 Wireless 时钟 545
10.3-5 数字调压器 547
10.3-6 低功耗调压器 548
10.3-7 flash 调压器 548
10.3-8 欠压检测器 549
10.3-9 欠压检测器处理方式 550
10.4-1 RTC 状态 551
10.6-1 ESP32-S3 启动流程图 556

11.1-1 系统定时器结构图 605
11.4-1 系统定时器生成报警 606

12.1-1 定时器组 625
12.2-1 定时器组架构 626

13.1-1 看门狗定时器概览 643
13.2-1 ESP32-S3 的看门狗定时器 645
13.3-1 SWD 控制器结构 647

14.0-1 XTAL32K 看门狗定时器 649

15.3-1 Internal SRAM1 的区域分割示意图 654
15.3-2 分割线 Category 分配示意图 655
15.5-1 外部存储器访问路径示意图 663

16.4-1 安全世界切换到非安全世界 769
16.4-2 非安全世界切换到安全世界 771
16.5-1 世界切换记录表 772
16.5-2 中断嵌套流程示意图 - 9 号入口 773
16.5-3 中断嵌套流程示意图 - 1 号入口 773
16.5-4 中断嵌套流程示意图 - 4 号入口 774

21.3-1 HMAC 附加填充比特示意图 847
21.3-2 HMAC 结构示意图 848

22.3-1 软件准备工作与硬件工作流程 859

23.3-1 片外存储器加解密工作配置 867

24.2-1 XTAL_CLK 脉宽 877

25.3-1 噪声源 878

26.3-1 UART 架构概况 882
26.3-2 UART 基本架构图 882
26.4-1 UART 共享 RAM 图 884
26.4-2 UART 控制器分频 885
26.4-3 UART 信号下降沿较差时序图 886
26.4-4 UART 数据帧结构 886
26.4-5 AT_CMD 字符格式 887
26.4-6 RS485 模式驱动控制结构图 888
26.4-7 SIR 模式编解码时序图 889
26.4-8 IrDA 编解码结构图 889
26.4-9 硬件流控图 890
26.4-10 硬件流控信号连接图 891
26.4-11 GDMA 模式数据传输 892
26.5-1 UART 编程流程 896

27.3-1 I2C 主机基本架构 936
27.3-2 I2C 从机基本架构 936
27.3-3 I2C 协议时序（引自 The I2C-bus specification Version 2.1 Fig.31） 937
27.3-4 I2C 时序参数（引自 The I2C-bus specification Version 2.1 Table5） 937
27.4-1 I2C 时序图 940
27.4-2 I2C 命令寄存器结构 942
27.5-1 I2C 主机写 7 位寻址的从机 946
27.5-2 I2C 主机写 10 位寻址的从机 948
27.5-3 I2C 主机写 7 位双地址寻址从机 949
27.5-4 I2C 主机分段写 7 位寻址的从机 951
27.5-5 I2C 主机读 7 位寻址的从机 953
27.5-6 I2C 主机读 10 位寻址的从机 955
27.5-7 I2C 主机从 7 位寻址从机的 M 地址读取 N 个数据 957
27.5-8 I2C 主机分段读 7 位寻址的从机 959

28.4-1 ESP32-S3 I2S 系统框图 985

28.5-1 时序图 – TDM Philips 标准 987
28.5-2 时序图 – TDM MSB 对齐标准 988
28.5-3 时序图 – TDM PCM 标准 988
28.5-4 时序图 – PDM 标准 989
28.6-1 I2S*n* 时钟 989
28.9-1 TX 数据格式控制 994
28.9-2 TDM 通道控制 995
28.9-3 PDM 通道控制 997

29.3-1 LCD_CAM 功能框图 1021
29.3-2 LCD 时钟 1022
29.3-3 Camera 时钟 1023
29.4-1 LCD 视频帧结构 1029
29.4-2 LCD 时序（RGB 格式） 1029
29.4-3 LCD 时序（I8080 格式） 1030

30.4-1 SPI 模块概览 1050
30.5-1 CPU 控制的传输中使用的数据 Buffer 1056
30.5-2 GP-SPI 功能块图 1059
30.5-3 主机模式下的数据流控制 1060
30.5-4 从机模式下的数据流控制 1061
30.5-5 GP-SPI 主机模式状态机 1063
30.5-6 GP-SPI2 主机使用全双工模式与 SPI 从机通信框图 1068
30.5-7 4-bit 模式下 GP-SPI2 与 flash 以及外部 RAM 的连接方式 1070
30.5-8 GP-SPI2 发送到 flash 的 SPI Quad I/O Read 命令序列 1070
30.5-9 主机模式下 DAM 控制的分段配置传输 1071
30.6-1 GP-SPI 访问外部 RAM 时推荐的 CS 时序配置 1081
30.6-2 GP-SPI 访问 flash 时推荐的 CS 时序配置 1081
30.7-1 SPI 时钟模式 0 和时钟模式 2 1082
30.7-2 SPI 时钟模式 1 和时钟模式 3 1083
30.8-1 GP-SPI 主机模式下时序补偿控制图 1084
30.8-2 GP-SPI2 主机模式下时序补偿 1085

31.3-1 数据帧和远程帧中的位域 1123
31.3-2 错误帧中的位域 1125
31.3-3 过载帧中的位域 1125
31.3-4 帧间距中的域 1127
31.3-5 位时序构成 1129
31.4-1 TWAI 概略图 1130
31.5-1 接收滤波器 1138
31.5-2 单滤波模式 1139
31.5-3 双滤波模式 1140
31.5-4 错误状态变化 1140
31.5-5 丢失仲裁的 bit 位置 1143

32.3-1 OTG_FS 系统架构 1158
32.3-2 内核地址映射 1159

32.3-3 主机模式 FIFO 1161
32.3-4 设备模式 FIFO 1162
32.3-5 OTG_FS 中断层次结构图 1163
32.3-6 Scatter/Gather DMA 链表结构 1164
32.4-1 A 设备 SRP 1167
32.4-2 B 设备 SRP 1168
32.4-3 A 设备 HNP 1169
32.4-4 B 设备 HNP 1170

33.2-1 USB Serial/JTAG 高层框图 1172
33.2-2 USB Serial/JTAG 框图 1172
33.3-1 USB 串口/JTAG 与 USB-OTG 内部/外部 PHY 连接图 1173
33.3-2 JTAG 信号传输 1174

34.2-1 SD/MMC 控制器连接的拓扑结构 1196
34.3-1 SD/MMC 控制器外部接口信号 1197
34.4-1 SDIO 主机结构框图 1197
34.4-2 命令通路状态机 1199
34.4-3 数据发送状态机 1199
34.4-4 数据接收状态机 1200
34.7-1 链表环结构 1202
34.8-1 链表结构 1202
34.10-1 时钟相位选择 1206

35.1-1 LED PWM 控制器架构 1233
35.2-1 定时器和 PWM 生成器功能块 1234
35.2-2 LEDC_CLK_DIV 非整数时的分频 1235
35.2-3 LED PWM 输出信号图 1237
35.2-4 输出信号占空比渐变图 1237

36.2-1 MCPWM 外设概览 1247
36.3-1 预分频器模块 1249
36.3-2 定时器模块 1249
36.3-3 操作器模块 1250
36.3-4 故障检测模块 1251
36.3-5 捕获模块 1252
36.3-6 递增计数模式波形 1253
36.3-7 递减计数模式波形 1253
36.3-8 递增递减循环模式波形，同步事件后递减 1254
36.3-9 递增递减循环模式波形，同步事件后递增 1254
36.3-10 递增模式中生成的 UTEP 和 UTEZ 1255
36.3-11 递减模式中生成的 DTEP 和 DTEZ 1255
36.3-12 递增递减模式中生成的 DTEP 和 UTEZ 1256
36.3-13 PWM 操作器的子模块 1257
36.3-14 递增递减模式下的对称波形 1260
36.3-15 递增计数模式，单边不对称波形，PWM*x*A 和 PWM*x*B 独立调制–高电平 1261
36.3-16 递增计数模式，脉冲位置不对称波形，PWM*x*A 独立调制 1262

36.3-17 递增递减循环计数模式，双沿对称波形，在 PWMxA 和 PWMxB 上独立调制–高电平有效 1263
36.3-18 递增递减循环计数模式，双沿对称波形，在 PWMxA 和 PWMxB 上独立调制–互补 1264
36.3-19 NCI 在 PWMxA 输出上软件强制事件示例 1265
36.3-20 CNTU 在 PWMxB 输出上软件强制事件示例 1266
36.3-21 死区模块的开关拓扑 1268
36.3-22 高电平有效互补 (AHC) 死区波形 1269
36.3-23 低电平有效互补 (ALC) 死区波形 1270
36.3-24 高电平有效 (AH) 死区波形 1270
36.3-25 低电平有效 (AL) 死区波形 1271
36.3-26 PWM 载波操作的波形示例 1272
36.3-27 载波模块的第一个脉冲和之后持续的脉冲示例 1273
36.3-28 PWM 载波模块中持续脉冲的 7 种占空比设置 1274

37.3-1 RMT 结构框图 1331
37.3-2 RAM 中脉冲编码结构 1332

38.0-1 PCNT 框图 1353
38.2-1 PCNT 单元基本架构图 1354
38.3-1 通道 0 递增计数图 1356
38.3-2 通道 0 递减计数图 1357
38.3-3 双通道递增计数图 1357

39.2-1 触摸传感器 1368
39.2-2 触摸传感器工作原理 1370
39.2-3 触摸传感器的内部结构 1370
39.2-4 Touch FSM 的内部结构 1371
39.2-5 TOUCH SCAN 时序图 1373
39.2-6 感测面积示意图 1375
39.3-1 SAR ADC 概图 1377
39.3-2 SAR ADC 的功能概况 1379
39.3-3 RTC SAR ADC 的功能概况 1381
39.3-4 APB_SARADC_SAR1_PATT_TAB1_REG 与样式 0 - 3 1383
39.3-5 APB_SARADC_SAR1_PATT_TAB2_REG 与样式 4 - 7 1383
39.3-6 APB_SARADC_SAR1_PATT_TAB3_REG 与样式 8 - 11 1383
39.3-7 APB_SARADC_SAR1_PATT_TAB4_REG 与样式 12 - 15 1383
39.3-8 样式表中的样式结构 1384
39.3-9 SAR ADC1 cmd0 配置示例 1384
39.3-10 SAR ADC1 cmd1 配置示例 1384
39.3-11 DMA 数据格式 1385
39.4-1 温度传感器结构 1387

第 I 卷

微处理器和主机

该部分介绍构成芯片系统的基本要素，包括超低功耗协处理器架构、CPU 的扩展指令、直接内存访问 (DMA) 控制器等。

第 1 章

处理器指令拓展 (PIE)

1.1 概述

为了提高特定 AI 和 DSP (Digital Signal Processing) 算法的运算效率，在 ESP32-S3 中新增了一组扩展指令。该扩展指令集使用 TIE (Tensilica Instruction Extension) 语言设计。新增了大位宽的通用寄存器、特殊寄存器、处理器端口。该扩展指令集主要采用单指令多数据 (SIMD, Single Instruction Multiple Data) 设计思想，支持 8-bit、16-bit 以及 32-bit 的向量运算，能显著提高数据运算效率；对于乘法、移位、累加等运算指令，在其进行数据计算的同时完成数据搬运操作，进一步提升单条指令的执行效率。

1.2 主要特性

处理器指令拓展 (PIE, Processor Instruction Extensions) 具有如下特性：

- 增加 128-bit 位宽通用寄存器
- 支持 128-bit 位宽的向量数据操作，包括：乘法、加法、减法、累加、移位、比较等
- 合并数据搬运指令与运算指令
- 支持非对齐 128-bit 带宽的向量数据
- 支持取饱和操作

1.3 结构概述

本章节有助于理解指令列表、指令可能性以及指令限制，但不提供硬件细节描述。

PIE 内部结构中有关乘法累加 (multiplication-accumulation, MAC) 指令的概述如下文所示。

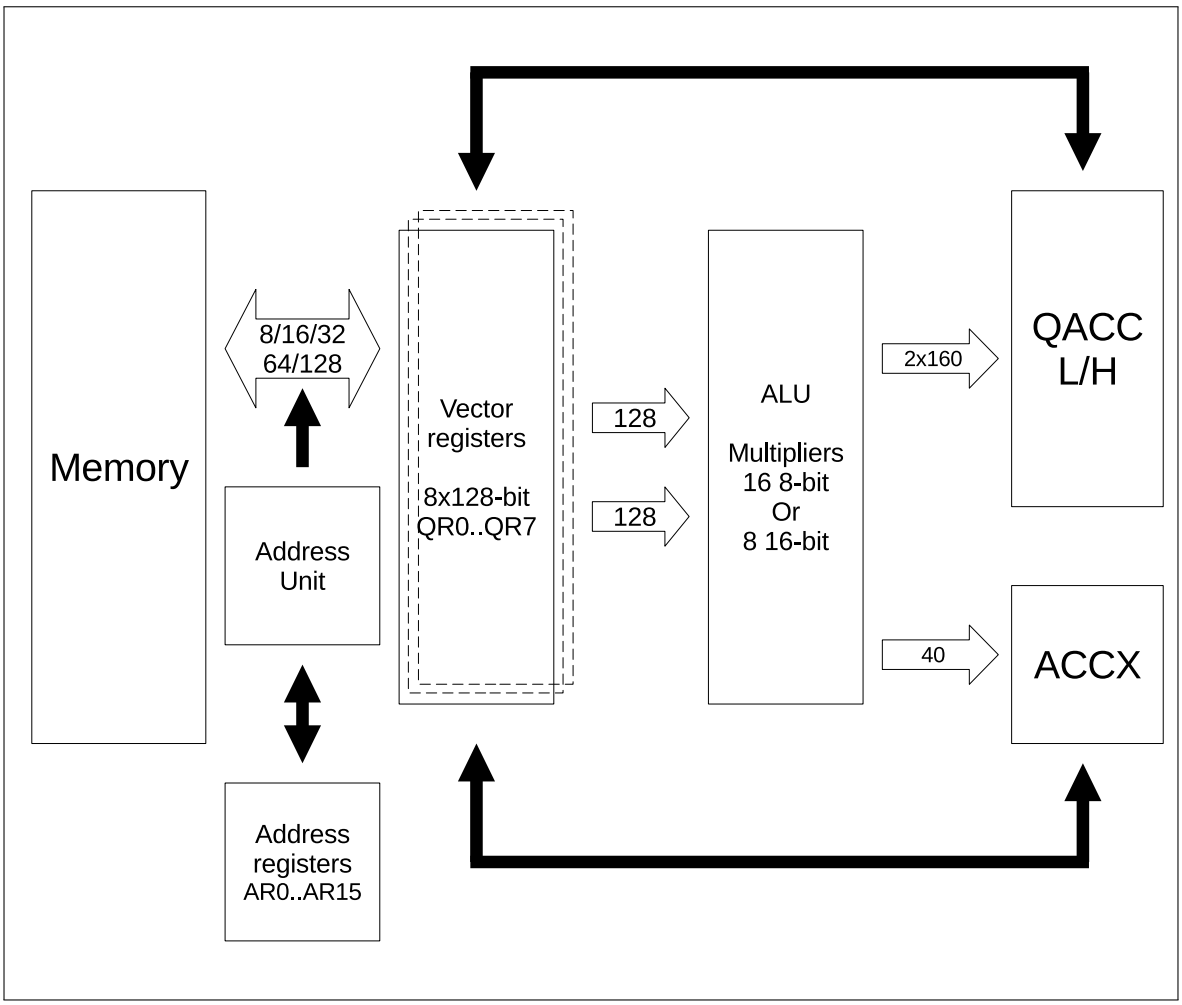


图 1.3-1. PIE 内部描述 (MAC)

上图展示了数据通路和 PIE 组件。

PIE 模块包含：

- 可读取 8/16/32/64/128-bit 对齐数据的地址单元
- 8 个 128-bit 向量 QR 的寄存器组
- 运算逻辑单元 (arithmetic logic unit, ALU) 支持
  - 16 个 8-bit 乘法器
  - 8 个 16-bit 乘法器
- QACC_H/QACC_L：2 个 160-bit 累加寄存器
- ACCX：40-bit 累加寄存器

1.3.1 向量寄存器组

向量寄存器组包含 8 个向量寄存器 (QR)。每个寄存器可以表示为 16 x 8-bit、8 x 16-bit 或 4 x 32-bit 的数据向量。根据使用的指令，处理器将按照 8-bit、16-bit 或 32-bit 数据格式进行相应的运算。

1.3.2 ALU

运算逻辑单元 (arithmetic logic unit, ALU) 可以作为 8-bit ALU 处理 8-bit 输入数据，也可作为 16-bit ALU 处理 16-bit 输入数据，或者作为 32-bit ALU 处理 32-bit 输入数据。例如 8-bit 乘法 ALU 包含 16 个 8-bit 乘法器，在一条指令中单个周期可完成 16 次乘法。考虑到对各种应用场景的支持，ESP32-S3 的扩展指令集中乘法运算几乎可以和任何其他运算进行组合，例如，FFT 指令在一条指令中包含乘法、加法和减法运算。此外，还支持 AND、OR、移位等 ALU 逻辑运算。
ALU 的输入操作数来自 QR 寄存器，运算结果可以保存到 QR 寄存器或特殊的累加寄存器（ACCX、QACC）。

1.3.3 QACC 累加寄存器

QACC 累加寄存器可用于 8-bit 或 16-bit 数据的乘累加运算。对于 8-bit 数据，QACC 由 16 个 20-bit 宽度的累加寄存器组成。对于 16-bit 运算，QACC 由 8 个 40-bit 宽度的累加寄存器组成。下文描述了 8-bit 数据运算的情况，对于 16-bit 数据运算，逻辑类似。
两个向量 QR 寄存器完成 16 次 8-bit 乘法和加法运算后，16 次运算的结果将写入 16 个 20-bit 累加器。
QACC 累加寄存器由由两部分组成：160-bit QACC_H 和 160-bit QACC_L。前者存储 QACC 寄存器高 160 比特的数据，后者存储 QACC 寄存器低 160 比特数据。若需将 QACC 累加寄存器的数据存储到 QR 存储器中，可以通过右移操作将 20-bit 数据片段转换为 8-bit。对于 16-bit 乘累加运算，则是右移 40-bit 数据片段得到 16-bit。
同时也支持从内存载入数据到 QACC 寄存器或将其数据重置为 0。

1.3.4 ACCX 累加寄存器

某些应用需要对所有乘法器的结果做累加操作。这种情况下，可以使用 ACCX 累加寄存器。
ACCX 是一个 40-bit 寄存器，其存储的数据既可以是 8-bit 数据的乘累加结果，也可以是 16-bit 数据的乘累加结果，取决于使用的指令。
同时也支持从内存载入数据到 ACCX 寄存器或将其数据重置为 0。

1.3.5 地址单元

PIE 中的大多数指令可以在一个周期内并行地从 128-bit Q 寄存器加载或存储数据。大多数情况下，数据应该为 128-bit 对齐，地址的低 4-bit 被忽略。地址单元可并行操作地址寄存器，从而节省更新地址寄存器的时间。
可对地址寄存器进行的操作有 AR + 有符号常量、ARx + ARy 以及 AR + 16。
地址单元在指令处理完成后再对寄存器进行操作，即所有对地址寄存器的操作都是在指令完成后完成的。

1.4 符号介绍

本章节将介绍指令描述时会涉及到的指令编码排序以及指令描述中出现的符号含义。

1.4.1 比特及字节序

指令编码的顺序根据颗粒度分为按字节排序和按比特排序两种。另外根据低位比特或者字节表示在左边或者右边分为：大端序和小端序。综上，常见的指令编码类型主要有小端比特序、大端比特序、小端字节序、大端字节序。

- 小端比特序：指令编码时按照比特顺序，将低位的比特放在右边。

- 大端比特序：指令编码时按照比特顺序，将低位的比特放在左边。
- 小端字节序：指令编码时按照字节顺序，将低位字节放在右边。
- 大端字节序：指令编码时按照字节顺序，将低位字节放在左边。

其中，采用小端字节序和小端比特序得到的指令编码比特序列是完全一致的。以指令编码长度为 24-bit 的指令 EE.ZERO.QACC 为例，图 1.4-1，1.4-2，1.4-3，1.4-4 分别给出了该指令在小端比特序、大端比特序、小端字节序以及大端字节序下的编码内容。

ESP32-S3 的 PIE 文档中所有指令以及寄存器的描述都是采用的小端比特序，即指令编码的最低位比特存放在地址的最低比特位置。

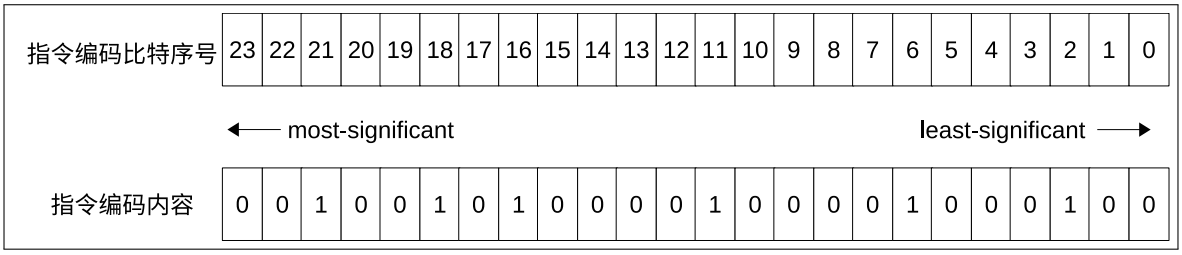


图 1.4-1. 指令 EE.ZERO.QACC 小端比特序

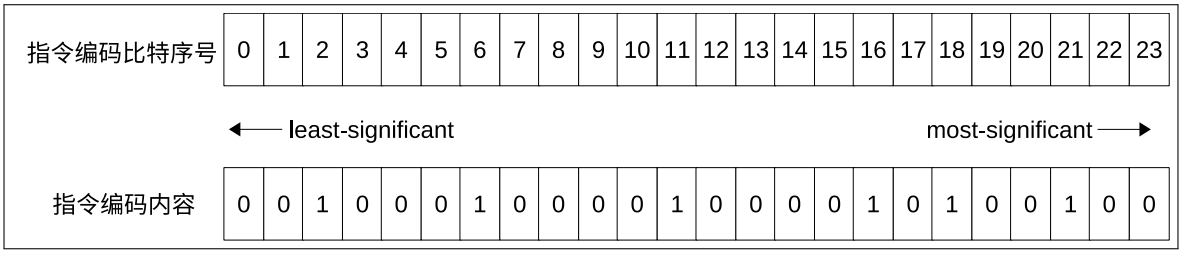


图 1.4-2. 指令 EE.ZERO.QACC 大端比特序

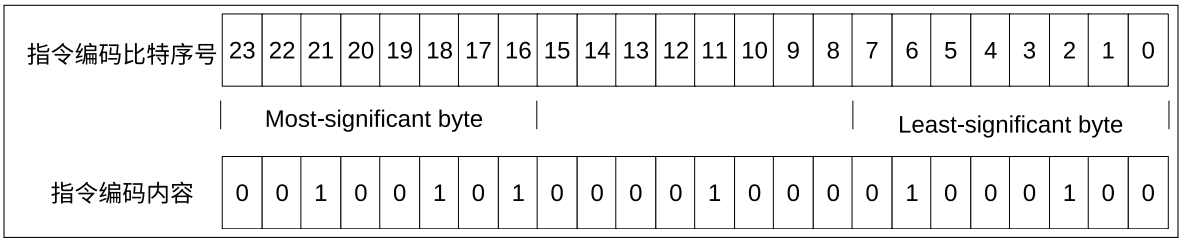


图 1.4-3. 指令 EE.ZERO.QACC 小端字节序

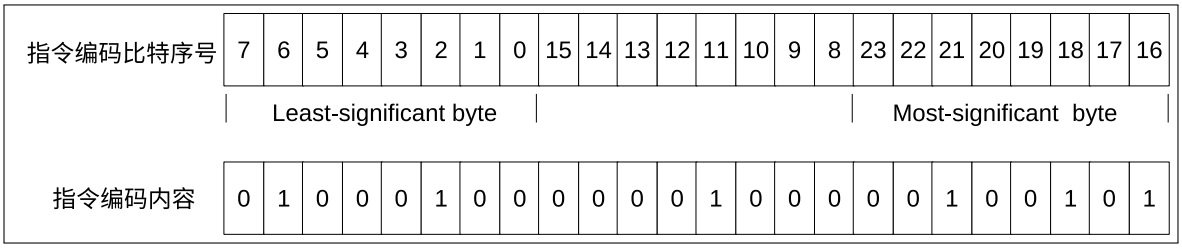


图 1.4-4. 指令 EE.ZERO.QACC 大端字节序

1.4.2 指令域定义

表 1.4-1 介绍了指令描述中涉及到的符号所代表的含义。可以在章节 1.8 中找到相关符号及其所表示的操作数的用途。

表 1.4-1. 指令域名及含义

域名			含义
a*	32-bit 通用寄存器	as	in-out 类型（同时作为输入输出操作数）。用于存放读写操作的地址信息，该地址信息在读写操作完成后被更新
		at	in-out 类型。在指令 EE.FFT.AMS.S16.ST.INCP 中暂存计算结果同时构成写入内存数据的一部分。
		ad	in 类型（用作输入操作数）。用于存放用于更新地址信息的数据。
		av	in 类型。用于存放待写入到内存的数据。
		ax,ay	in 类型。用于存放参与指令运算的数据，比如移位数值、乘数等。
		au	out 类型（用作输出操作数）。用于存放指令的运算结果。
q*	128-bit 通用寄存器	qs	in 类型。存放用于拼接操作的 128-bit 数据。
		qa、qx、qy、qm	in 类型。用于存放向量运算的数据。
		qz	out 类型。用于存放向量运算的结果。
		qu	out 类型。用于存放从内存中读取的数据。
		qv	in 类型。用于存放待写入到内存的数据。
fu			32-bit 通用浮点寄存器，用于存放从内存读取的浮点数据。
fv			32-bit 通用浮点寄存器，用于存放待写入内存的浮点数据。
sel2			1-bit 立即数，范围 0 ~ 1。作为选择信号使用。
sel4、upd4			2-bit 立即数，范围 0 ~ 3。作为选择信号使用。
sel8			3-bit 立即数，范围 0 ~ 7。作为选择信号使用。
sel16			4-bit 立即数，范围 0 ~ 15。作为选择信号使用。
sar2			1-bit 立即数，范围 0 ~ 1。用于表示移位大小。
sar4			2-bit 立即数，范围 0 ~ 3。用于表示移位大小。
sar16			4-bit 立即数，范围 0 ~ 15。用于表示移位大小。
imm1			7-bit 无符号立即数，可取的数值范围 0 ~ 127，间隔为 1。用于表示更新读写操作地址的数值大小。
imm2			7-bit 无符号立即数，可取的数值范围 0 ~ 254，间隔为 2。用于表示更新读写操作地址的数值大小。
imm4			8-bit 有符号立即数，可取的数值范围 -256 ~ 252，间隔为 8。用于表示更新读写操作地址的数值大小。
imm16			8-bit 有符号立即数，可取的数值范围 -2048 ~ 2032，间隔为 16。用于表示更新读写操作地址的数值大小。
imm16f			4-bit 有符号立即数，可取的数值范围 -128 ~ 112，间隔为 16。用于表示更新读写操作地址的数值大小。

对于存在多个同样功能操作数的指令，通过在域名后增加数字编号加以区分。例如：指令 EE.LDF.128.IP 使用了 4 个 fu 寄存器，分别命名为 fu0 ~ 3，用于存放从内存中读取 128-bit 数据。

1.5 扩展指令集组件

1.5.1 寄存器

本章节介绍了 ESP32-S3 扩展指令集所有涉及到的寄存器，包括 Xtensa 初始定义的寄存器以及自定义的寄存器。寄存器信息如表 1.5-1 所示。

表 1.5-1. ESP32-S3 扩展指令集寄存器集合

寄存器助记符	数量	比特位宽	访问权限	类型
AR	16[1]	32	可读可写	通用寄存器
FR	16	32	可读可写	通用寄存器
QR	8	128	可读可写	自定义通用寄存器
SAR	1	6	可读可写	特殊寄存器
SAR_BYTE	1	4	可读可写	自定义特殊寄存器
ACCX	1	40	可读可写	自定义特殊寄存器
QACC_H	1	160	可读可写	自定义特殊寄存器
QACC_L	1	160	可读可写	自定义特殊寄存器
FFT_BIT_WIDTH	1	4	可读可写	自定义特殊寄存器
UA_STATE	1	128	可读可写	自定义特殊寄存器

[1] Xtensa 处理器内部共有 64 个 AR 寄存器。其在设计上采用了寄存器窗口技术，任意时刻软件都只能获取 64 个 AR 寄存器中 16 个的数值内容。通过旋转窗口，替换函数调用以及异常触发时保存寄存器的行为，能够有效提升程序性能。

1.5.1.1 通用寄存器

用户在使用通用寄存器作为操作数的指令时，需要显性地声明所要指派的通用寄存器编号。例如：

$$EE.VADDS.S8 \quad q2, q0, q1$$

上述指令使用 0 号和 1 号 QR 寄存器作为输入向量。向量加法的结果存放到 2 号 QR 寄存器中。

AR

指令中每有一个 AR 寄存器操作数，都将占用 4-bit 的编码长度。用户可以选择 16 个 AR 寄存器中任意一个作为操作数，4-bit 编码数值表明了用户选取编号。表 1.4-1 中的 “a*” 行列出了 AR 寄存器在扩展指令集中各种用途，既可以用于存储访问地址，也可以用于存储数据。

FR

指令中每有一个 FR 寄存器操作数，都将占用 4-bit 的编码长度。用户可以选择 16 个 FR 寄存器中任意一个作为操作数，4-bit 编码数值表明了用户选取编号。在 ESP32-S3 扩展指令集中，浮点数据只有读写内存指令，凭借 128-bit 访存带宽，相比 Xtensa 处理器自带的 32-bit 浮点数读写指令，效率提升 4 倍。

QR

为了提高程序的执行效率，通常将操作的数据对象存储在通用寄存器中来实现数据的快速读写。Xtensa 自带的 AR 寄存器只有 32-bit 位宽，而 ESP32-S3 一次数据访存的位宽为 128-bit，因此 AR 寄存器只能发挥现有数据总线 1/4 的带宽能力。为此 ESP32-S3 新增了 8 个 128-bit 自定义通用寄存器（QR 寄存器）。QR 寄存器主要用来存储 128-bit 位宽数据总线对内存进行读/写操作所获取/使用的数据，也被用来暂存 128-bit 带宽的数据操作所产生的的运算结果。

在处理器执行指令的过程中，根据该指令定义的向量运算位宽，将单个 QR 寄存器视作 16 个 8-bit 或者 8 个 16-bit 或者 4 个 32-bit 的操作数，从而实现单条指令完成多个操作数的运算操作。

1.5.1.2 特殊寄存器

相比通用寄存器，特殊寄存器是在特定指令中被隐性地调用。用户无需也无法在执行指令时指定某个特殊寄存器参与运算。例如：

$$EE.VMUL.S16 \quad q2, q0, q1$$

上述向量乘法指令使用 q0 和 q1 通用寄存器作为输入，在内部运算过程中要对中间 32-bit 乘法结果进行右移处理，再截取移位后结果的低 16-bit 组成 128-bit 输出到 q2。中间右移大小就是由 SAR 特殊寄存器内数值决定，且 SAR 特殊寄存器并不会出现在指令操作数列表中。

SAR

移位数值寄存器 (Shift Amount Register) 存储以比特为单位的移位的位数大小。ESP32-S3 扩展指令集中会用 SAR 寄存器的指令有两类。一类是向量数据移位指令，包括 EE.VSR.32、EE.VSL.32。前者使用 SAR 低 5 比特数值大小作为右移数值；后者使用 SAR 低 5 比特数值大小作为左移数值。另一类是乘法运算指令，包括 EE.VMUL.*、EE.CMUL.*、EE.FFT.AMS.*、EE.FFT.CMUL.*。这一类指令使用 SAR 寄存器内数值作为中间乘法运算结果右移的数值，数值的大小决定了最终计算结果的精度。

SAR_BYTE

以字节为单位的移位数值寄存器。该特殊寄存器被设计用来处理数据非 128-bit 对齐的问题（章节 1.5.3 中介绍了数据非对齐问题）。对于向量运算指令而言，通过扩展指令读取或存储的数据都是强制 16-字节对齐的，但实际应用中往往无法保证使用的数据地址一定是 16-字节对齐的。

EE.LD.128.USAR.IP 和 EE.LD.128.USAR.XP 指令在从内存读取 128-bit 数据的同时会将代表非对齐的数据的访存地址寄存器的低 4-bit 数值内容写入 SAR_BYTE。

ESP32-S3 扩展指令集中会用 SAR_BYTE 寄存器的指令有两类。一类是专门用于处理 QR 内数据不对齐，包括 EE.SRCQ.*, EE.SRC.Q*。这类指令将分别从非对齐地址前后两个对齐地址读取出两个 16-字节数据进行拼接，再右移 SAR_BYTE 寄存器内数值的字节大小，从而得到一个来自非对齐地址的 128-bit 数据。另一类是在执行运算的同时进行非对齐数据的处理工作，该类指令的名称后缀带有 “.QUP”。

ACCX

乘累加寄存器 (multiplier-accumulator)。调用该寄存器的指令包括 EE.VMULAS.*.ACCX*, EE.SRS.ACCX。前者利用 ACCX 寄存器存储累加上两个 QR 寄存器向量乘的结果；后者对 ACCX 寄存器进行右移处理。

QACC_H,QACC_L

按片分区的连续乘累加寄存器。调用该寄存器的指令包括 EE.VMULAS.*.QACC*, EE.SRCMB.*.QACC。该寄存器的主要应用场景是将两个 QR 寄存器向量乘的结果按照片区分别累加到 QACC_H 和 QACC_L 寄存器的对应片区中，其中 16-bit 的向量乘结果会分别累加到对应 16 个 20-bit 片区，32-bit 的向量乘结果则累加到对应 8 个 40-bit 片区。

FFT_BIT_WIDTH

EE.BITREV 指令专用特殊寄存器。该寄存器内数值用于指示 EE.BITREV 的工作模式。数值范围 0 ~ 7 分别代表 3-bit ~ 10-bit 工作模式，具体指令功能请查阅 EE.BITREV。

UA_STATE

EE.FFT.AMS.S16.LD.INCP.UAUP 指令专用特殊寄存器。该寄存器用于存储从内存中读取的非对齐 128-bit 数据，在下次调用该指令时通过与新读取的非对齐数据拼接后移位来获取 128-bit 对齐的数据。

1.5.2 快速 GPIO 端口

ESP32-S3 的 Xtensa 处理器新增了两组端口信号：GPIO_OUT 和 GPIO_IN。用户可以通过 GPIO Matrix 模块将上述两组端口信号配置到芯片特定的 GPIO 管脚上。

1.5.2.1 GPIO_OUT

8-bit 处理器输出端口。用户先通过 GPIO Matrix 将上述 8-bit 端口信号配置到指定的芯片管脚。core0 对应的 8 个输出端口号为 pro_alonegpio_out0~7；core1 对应的 8 个输出端口号为 core1_gpio_out0~7。随后可以通过指令 EE.WR_MASK_GPIO_OUT, EE.SET_BIT_GPIO_OUT 将 GPIO_OUT 某些比特位置置 1，也可以通过指令 EE.CLR_BIT_GPIO_OUT 将某些比特位的数据置 0，进而驱动相应的芯片管脚为高或低电平。相比通过配置寄存器来驱动芯片管脚，用户调用上述指令驱动芯片 GPIO 管脚可以获得更快的响应。

1.5.2.2 GPIO_IN

8-bit 处理器输入端口。用户先通过 GPIO Matrix 将上述 8-bit 端口信号配置到指定的芯片管脚。core0 对应的 8 个输入端口号为 pro_alonegpio_in0~7；core1 对应的 8 个输入端口号为 core1_gpio_in0~7。随后可以通过指令 EE.GET_GPIO_IN 读取 8 个芯片 GPIO 管脚的电平值到 AR 寄存器。相比读取寄存器来获取芯片管脚的状态，通过上述指令可以更快的获取并处理芯片 GPIO 管脚上的电平变化。

1.5.3 数据格式及对齐

当前扩展指令集支持字节、2-字节、4-字节、8-字节、16-字节的数据格式。同时还存在 20-字节的数据格式：QACC_H 和 QACC_L，这两个特殊寄存器无法直接与内存进行数据交换。用户可以通过 5 个 4-字节（AR 寄存器）或者 2 个 16-字节 (QR) 寄存器与 QACC_H 和 QACC_L 进行数据传输。

下表 1.5-2 列出每种数据格式的比特位宽和对齐信息（"x" 表示该比特数据位可以是"0" 值也可以是"1" 值）。Xtensa 处理器使用字节作为所有数据格式存储在内存中地址的最小单位。字节顺序采用小端字节序，0 号字节存放在最低位（最右边），如图 1.4-3 所示。

表 1.5-2. 数据格式及对齐

数据格式	长度	内存中对齐地址
1-字节	8 bits	xxxx
2-字节	16 bits	xxx0
4-字节	32 bits	xx00
8-字节	64 bits	x000
16-字节	128 bit	0000

若数据格式在内存中的存放位置是非对齐的地址，此时直接对该数据存储在内存中地址进行访问面临被拆分成二次访存的情况，进而影响到代码的性能。例如，用户期望从内存中读取一个 16-字节的数据，如表 1.5-2 所示，对齐的情况下，该数据在内存中的存储地址地位为 0000。实际情况下该数据的地址低位可能是 0000 ~ 1111（二进制）中的任意一种情况。假设其存放在内存的地址低位是 0_0100，处理器会将对该数据的一次访存操作

变成两次，分别是对 0_0000 和 1_0000 地址的访问。随后处理器从得到的 2 个 16-字节中拼接出所需要的 16-字节。

为了避免上述非对齐访存操作导致的代码性能降低的情况发生，扩展指令集中所有的访存指令发出的访问地址都被强制对齐了，即地址低位将被 0 值替换。例如，对 0x3fc8_0024 发起 128-bit 数据读取操作，访问地址的低 4-bit 将被强制置 0，最终读取到的是地址 0x3fc8_0020 处存放的 128-bit 数据。同理进行 64-bit 数据访问的地址的低 3-bit 将被置 0；进行 32-bit 数据访问的地址的低 2-bit 将被置 0；进行 16-bit 数据访问的地址的低 1-bit 将被置 0。

上述设计特性要求用户保证代码中发起的访存操作的地址都是对齐的，否则读取到数据并不是期望的内容。用户需要显性地声明该变量或者数组在内存中的对齐方式。16 字节对齐能够满足绝大部分应用场景的需求。

考虑到某些应用场景下，参与运算的数据的内存地址是不确定的。ESP32-S3 扩展指令集提供了 SAR_BYTE 特殊寄存器及 EE.LD.128.USAR.*、EE.SRC.* 等相关指令来处理非对齐的数据。

假设通用寄存器 a8 中存储着 128-bit 非对齐数据地址。可以通过下列代码将该 128-bit 数据读取到指定 QR 寄存器（例子中以 q2 代替）：

$EE.LD.128.USAR.IP \quad q0, \quad a8, \quad 16$

$EE.VLD.128.IP \quad q1, \quad a8, \quad 16$

$EE.SRC.Q \quad q2, \quad q0, \quad q1$

1.5.4 数据溢出及饱和处理

数据溢出是指运算结果的数值大小超出了结果暂存寄存器所能存储的数值范围。以指令 EE.VMUL.S8 为例，两个 8-bit 乘数的结果是 16-bit，进行符号右移后的结果应该仍是 16-bit，但最终的结果将被存储到 8-bit 寄存器空间中，此时便存在数据溢出风险。

在 ESP32-S3 扩展指令的设计中，对于数据溢出，有两种处理方式：取饱和、截取低有效位。前者根据结果寄存器所能存储的数值范围对计算结果进行约束，超过结果寄存器所能表示的最大数值时取最大值；小于结果寄存器所能表示的最小数值时取最小值。这部分会在指令的描述中显性的表示出来，比如 EE.VADDS.* 系列指令对加法结果就进行了取饱和操作。对于更多指令内部运算结果的数据溢出处理，统一采用截断的方式处理，即只截取结果数据中与结果寄存器位宽一致的低位数据存入结果寄存器中。

请用户注意：对于指令描述中没有提及取饱和处理的指令，数据溢出将按照截断处理。

1.6 扩展指令简介

表 1.6-1 列出了 ESP32-S3 扩展指令集包含的指令类别及各类别包括的指令信息。本章节将对各指令类别做简要介绍。

表 1.6-1. 指令简介

扩展指令类别	指令[1]	参考章节
读内存指令	LD.QR	1.6.1
	EE.VLD.128.[XP/IP]	
	EE.VLD.[H/L].64.[XP/IP]	
	EE.VLDBC.[8/16/32].[-/XP/IP]	
	EE.VLDHBC.16.INCP	
	EE.LDF.[64/128].[XP/IP]	
	EE.LD.128.USAR.[XP/IP]	
	EE.LDQA.[U/S][8/16].128.[XP/IP]	
	EE.LD.QACC_[H/L].[H.32/L.128].IP	
	EE.LD.ACCX.IP	
	EE.LD.UA_STATE.IP	
	EE.LDXQ.32	
写内存指令	ST.QR	1.6.2
	EE.VST.128.[XP/IP]	
	EE.VST.[H/L].64.[XP/IP]	
	EE.STF.[64/128].[XP/IP]	
	EE.ST.QACC_[H/L].[H.32/L.128].IP	
	EE.ST.ACCX.IP	
	EE.ST.UA_STATE.IP	
	EE.STXQ.32	
数据交换指令	MV.QR	1.6.3
	EE.MOVI.32.A	
	EE.MOVI.32.Q	
	EE.VZIP.[8/16/32]	
	EE.VUNZIP.[8/16/32]	
	EE.ZERO.Q	
	EE.ZERO.QACC	
	EE.ZERO.ACCX	
	EE.MOV.S8.QACC	
	EE.MOV.S16.QACC	
	EE.MOV.U8.QACC	
	EE.MOV.U16.QACC	
运算指令	EE.VADDS.S[8/16/32].[-/LD.INCP/ST.INCP]	1.6.4
	EE.VSUBS.S[8/16/32].[-/LD.INCP/ST.INCP]	
	EE.VMUL.[U/S][8/16].[-/LD.INCP/ST.INCP]	
	EE.CMUL.S16.[-/LD.INCP/ST.INCP]	
	EE.VMULAS.[U/S][8/16].ACCX.[-/LD.IP/LD.XP]	
	EE.VMULAS.[U/S][8/16].QACC.[-/LD.IP/LD.XP/LDBC.INCP]	
	EE.VMULAS.[U/S][8/16].ACCX.[LD.IP/LD.XP].QUP	
	EE.VMULAS.[U/S][8/16].QACC.[LD.IP/LD.XP/LDBC.INCP].QUP	
	EE.VSMULAS.S[8/16].QACC.[-/LD.INCP]	
	EE.SRCMB.S[8/16].QACC	

Con't on next page

表 1.6-1 – con't from previous page

扩展指令类别	指令[1]	参考章节
	EE.SRS.ACCX	
	EE.VRELU.S[8/16]	
	EE.VPRELU.S[8/16]	
比较指令	EE.VMAX.S[8/16/32].[-/LD.INCP/ST.INCP]	1.6.5
	EE.VMIN.S[8/16/32].[-/LD.INCP/ST.INCP]	
	EE.VCMP.[EQ/LT/GT].S[8/16/32]	
按位逻辑操作指令	EE.ORQ	1.6.6
	EE.XORQ	
	EE.ANDQ	
	EE.NOTQ	
移位指令	EE.SRC.Q	1.6.7
	EE.SRC.Q.QUP	
	EE.SRC.Q.LD.[XP/IP]	
	EE.SLCI.2Q	
	EE.SLCXXP.2Q	
	EE.SRCI.2Q	
	EE.SRCXXP.2Q	
	EE.SRCQ.128.ST.INCP	
	EE.VSR.32	
	EE.VSL.32	
FFT 专用指令	EE.FFT.R2BF.S16.[-/ST.INCP]	1.6.8
	EE.FFT.CMUL.S16.[LD.XP/ST.XP]	
	EE.BITREV	
	EE.FFT.AMS.S16.[LD.INCP.UAUP/LD.INCP/LD.R32.DECP/ST.INCP]	
	EE.FFT.VST.R32.DECP	
GPIO 控制指令	EE.WR_MASK_GPIO_OUT	1.6.9
	EE.SET_BIT_GPIO_OUT	
	EE.CLR_BIT_GPIO_OUT	
	EE.GET_GPIO_IN	

[1] 这些指令的详细介绍请参考 1.8 章节。

1.6.1 读内存指令

读内存指令根据存储有访存地址信息的 AR 寄存器，让处理器发出虚地址访问内存。对于大部分指令而言，都是采用读内存后更新访存地址的策略。指令 EE.LDXQ.32 是特例，该指令先通过立即数选择 QR 寄存器中某片 16-bit 数据，加到访存地址上，随后再发出访存操作。

由于不对齐的地址访问会导致访存行为变慢，扩展指令集中的所有读内存指令发出的虚地址都是根据数据格式强制对齐后的地址。根据访存数据格式的大小，内存返回相应位宽的数据：1-字节，2-字节，4-字节，8-字节或者 16-字节。当强制对齐后读取到的数据与预期不一致时，可以通过 EE.SRC.Q 等指令从多个 QR 寄存器中提取出想要的数据。

下表对读内存指令的访存行为进行了简要说明，更多信息参见 1.8。

表 1.6-2. 读内存指令

指令	定义
LD.QR	将 16-字节数据加载至 QR 寄存器。
EE.VLD.128.XP	读取 16-字节数据，随后访存地址加上 AR 寄存器内数值。
EE.VLD.128.IP	读取 16-字节数据，随后访存地址加上立即数。
EE.VLD.[H/L].64.XP	读取 8-字节数据，随后访存地址加上 AR 寄存器内数值。
EE.VLD.[H/L].64.IP	读取 8-字节数据，随后访存地址加上立即数。
EE.VLDBC.[8/16/32]	读取 1-字节/2-字节/4-字节数据。
EE.VLDBC.[8/16/32].XP	读取 1-字节/2-字节/4-字节数据，随后访存地址加上 AR 寄存器内数据。
EE.VLDBC.[8/16/32].IP	读取 1-字节/2-字节/4-字节数据，随后访存地址加上立即数。
EE.VLDHBC.16.INCP	读取 16-字节数据，随后访存地址加上 16。
EE.LDF.[64/128].XP	读取 8-字节/16-字节数据，随后访存地址加上 AR 寄存器内数值。
EE.LDF.[64/128].IP	读取 8-字节/16-字节数据，随后访存地址加上立即数。
EE.LD.128.USAR.XP	读取 16-字节数据，随后访存地址加上 AR 寄存器内数值。
EE.LD.128.USAR.IP	读取 16-字节数据，随后访存地址加上立即数。
EE.LDQA.U8.128.[XP/IP]	读取 16-字节数据，按照 1-字节大小分片，随后 0 扩展为 20-bit 数据写入 QACC_H 和 QACC_L 寄存器中，最后在访存地址上加上 AR 寄存器内数值/立即数。
EE.LDQA.U16.128.XP	读取 16-字节数据，按照 2-字节大小分片，随后 0 扩展为 40-bit 数据写入 QACC_H 和 QACC_L 寄存器中，最后在访存地址上加上 AR 寄存器内数值/立即数。
EE.LDQA.S8.128.XP	读取 16-字节数据，按照 1-字节大小分片，随后符号位扩展为 20-bit 数据写入 QACC_H 和 QACC_L 寄存器中，最后在访存地址上加上 AR 寄存器内数值/立即数。
EE.LDQA.S16.128.XP	读取 16-字节数据，按照 1-字节大小分片，随后符号位扩展为 40-bit 数据写入 QACC_H 和 QACC_L 寄存器中，最后在访存地址上加上 AR 寄存器内数值/立即数。
EE.LD.QACC_[H/L].H.32.IP	读取 4-字节数据，随后访存地址加上立即数。
EE.LD.QACC_[H/L].L.128.IP	读取 16-字节数据，随后访存地址加上立即数。
EE.LD.ACCX.IP	读取 8-字节数据，随后访存地址加上立即数。
EE.LD.UA_STATE.IP	读取 16-字节数据，随后访存地址加上立即数。
EE.LDXQ.32	先更新访存地址，随后读取 4-字节数据。

1.6.2 写内存指令

写内存指令根据存储有访存地址信息的 AR 寄存器，让处理器发出虚地址访问内存。对于大部分指令而言，都是采用写内存后更新访存地址的策略。指令 EE.STXQ.32 是特例，该指令先通过立即数选择 QR 寄存器中某片 16-bit 数据，加到访存地址上，随后再发出访存操作。

由于不对齐的地址访问会导致访存行为变慢，扩展指令集中的所有写内存指令发出的虚地址都是根据数据格式强制对齐后的地址。根据访存数据格式的大小，向内存写入相应位宽的数据：1-字节，2-字节，4-字节，8-字节或者 16-字节。当待写入内存的数据位宽小于访存位宽时，需要对待写入数据进行 0 扩展或者符号位扩展。

下表对写内存指令的访存行为进行了简要说明，更多信息参见章节 1.8。

表 1.6-3. 写内存指令

指令	定义
ST.QR	将 QR 寄存器中的 16-字节数据写入内存中。
EE.VST.128.XP	将 16-字节数据写入内存，随后访存地址加上 AR 寄存器内数值。
EE.VST.128.IP	将 16-字节数据写入内存，随后访存地址加上立即数。
EE.VST.[H/L].64.XP	将 8-字节数据写入内存，随后访存地址加上 AR 寄存器内数值。
EE.VST.[H/L].64.IP	将 8-字节数据写入内存，随后访存地址加上立即数。
EE.STF.[64/128].XP	将 8-字节/16-字节数据写入内存，随后访存地址加上 AR 寄存器内数值。
EE.STF.[64/128].IP	将 8-字节/16-字节数据写入内存，随后访存地址加上立即数。
EE.ST.QACC_[H/L].H.32.IP	将 4-字节数据写入内存，随后访存地址加上立即数。
EE.ST.QACC_[H/L].L.128.IP	将 16-字节数据写入内存，随后访存地址加上立即数。
EE.ST.ACCX.IP	将 ACCX 寄存器内数值 0 扩展为 8-字节数据后写入内存，随后访存地址加上立即数。
EE.ST.UA_STATE.IP	将 16-字节数据写入内存，随后访存地址加上立即数。
EE.STXQ.32	先更新访存地址，随后将 4-字节数据写入内存。

1.6.3 数据交换指令

数据交换指令主要用于交换不同寄存器之间的数据信息。考虑到交换数据之间的寄存器位宽不匹配的情况，一方面通过增加立即数作为选择信号；另一方面提供 0 扩展和符号位扩展指令。多种数据交换指令，可以满足用户多种场景下的数据交换要求。

更多有关数据交换指令的详细信息，请参见章节 1.8。

表 1.6-4. 数据交换指令

指令	定义
MV.QR	将原 QR 寄存器中的值移动到目标 QR 寄存器中。
EE.MOVI.32.A	将 QR 寄存器中某片 32-bit 数据赋值给 AR 寄存器。
EE.MOVI.32.Q	将 AR 寄存器内数据赋值给 QR 寄存器中某片 32-bit 数据空间。
EE.VZIP.[8/16/32]	按照 1-字节/2-字节/4-字节为处理单位对两个 QR 寄存器进行编码。
EE.VUNZIP.[8/16/32]	按照 1-字节/2-字节/4-字节为处理单位对两个 QR 寄存器进行解码。
EE.ZERO.Q	将指定 QR 寄存器清零。
EE.ZERO.QACC	将 QACC_H 和 QACC_L 寄存器清零。
EE.ZERO.ACCX	将指定 ACCX 寄存器清零。
EE.MOV.S8.QACC	按照 1-字节将 QR 寄存器分片，随后按符号位扩展成 20-bit 数据，赋值给 QACC_H 和 QACC_L 寄存器。
EE.MOV.S16.QACC	按照 2-字节将 QR 寄存器分片，随后按符号位扩展成 40-bit 数据，赋值给 QACC_H 和 QACC_L 寄存器。
EE.MOV.U8.QACC	按照 1-字节将 QR 寄存器分片，随后按 0 扩展成 20-bit 数据，赋值给 QACC_H 和 QACC_L 寄存器。
EE.MOV.U16.QACC	按照 2-字节将 QR 寄存器分片，随后按 0 扩展成 40-bit 数据，赋值给 QACC_H 和 QACC_L 寄存器。

1.6.4 运算指令

运算指令主要采用 SIMD（单指令多数据）设计原则，均是针对向量数据的运算指令，主要包括：向量加法、向量乘法、向量复数乘法、向量乘累加、向量和标量乘累加以及其他。

向量加法

ESP32-S3 提供 1-字节、2-字节、4-字节为数据处理单位的向量加减法指令。考虑到向量运算所需的输入输出操作数都是存储在内存中，为了减少额外执行读取内存操作，提升代码执行速度，还设计了在向量运算同时执行 16-字节访存的向量加法指令，并在访存结束后，地址寄存器内数值增加 16，从而直接指向连续的下一个 16-字节的内存地址。用户可以根据实际算法需求，选择相应的指令。
此外向量加法指令还对加减法结果进行了取饱和操作以保证运算精度。

表 1.6-5. 向量加法运算指令

指令	定义
EE.VADDS.S[8/16/32]	1-字节/2-字节/4-字节向量加法。
EE.VADDS.S[8/16/32].LD.INCP	1-字节/2-字节/4-字节向量加法同时从内存读取 16-字节数据。
EE.VADDS.S[8/16/32].ST.INCP	1-字节/2-字节/4-字节向量加法同时将 16-字节数据写入内存。
EE.VSUBS.S[8/16/32]	1-字节/2-字节/4-字节向量减法。
EE.VSUBS.S[8/16/32].LD.INCP	1-字节/2-字节/4-字节向量减法同时从内存读取 16-字节数据。
EE.VSUBS.S[8/16/32].ST.INCP	1-字节/2-字节/4-字节向量减法同时将 16-字节数据写入内存。

向量乘法

ESP32-S3 提供 1-字节、2-字节为数据处理单位的向量乘法指令，并且支持无符号和有符号两种向量乘法。
考虑到向量运算所需的输入输出操作数都是存储在内存中，为了减少额外执行读取内存操作，提升代码执行速度，还设计了在向量运算同时执行 16-字节访存的向量乘法指令，并在访存结束后将访存地址增加 16，使其指向下一个 16-字节的内存地址。用户可以根据实际算法需求，选择相应的指令。

表 1.6-6. 运算指令

指令	定义
EE.VMUL.U[8/16]	无符号 1-字节/2-字节向量乘法。
EE.VMUL.S[8/16]	有符号 1-字节/2-字节向量乘法。
EE.VMUL.U[8/16].LD.INCP	无符号 1-字节/2-字节向量乘法同时从内存读取 16-字节数据。
EE.VMUL.S[8/16].LD.INCP	有符号 1-字节/2-字节向量乘法同时从内存读取 16-字节数据。
EE.VMUL.U[8/16].ST.INCP	无符号 1-字节/2-字节向量乘法同时将 16-字节数据写入内存。
EE.VMUL.S[8/16].ST.INCP	有符号 1-字节/2-字节向量乘法同时将 16-字节数据写入内存。

向量复数乘法

ESP32-S3 提供 2-字节为数据处理单位的向量复数乘法指令。指令运算操作数都是按照符号数来执行的。

考虑到向量运算所需的输入输出操作数都是存储在内存中，为了减少额外执行读取内存操作，提升代码执行速度，还设计了在向量运算同时执行 16-字节访存的指令，并在访存结束后将访存地址增加 16，使其指向下一个 16-字节的内存地址。用户可以根据实际算法需求，选择相应的指令。

表 1.6-7. 运算指令

指令	定义
EE.CMUL.S16	2-字节向量复数乘法指令。
EE.CMUL.S16.LD.INCP	2-字节向量复数乘法指令同时从内存读取 16-字节数据。
EE.CMUL.S16.ST.INCP	2-字节向量复数乘法指令同时将 16-字节数据写入内存。

向量乘累加

ESP32-S3 提供两种类型的向量乘累加。一种是基于 ACCX 寄存器，将多个向量乘法的结果累加到一个 40-bit ACCX 寄存器中。一种是基于 QACC_H 和 QACC_L 寄存器，将向量乘法的结果分别累加到 QACC_H 和 QACC_L 对应的比特片区上。上述两种乘累加指令都支持 1-字节和 2-字节分片大小的乘累加。
为了减少额外执行读取内存操作，提升代码执行速度，还设计了在向量运算同时执行 16-字节访存的指令，并在访存结束后将访存地址增加 AR 寄存器内数值或者立即数大小。
除此之外，向量乘累加指令中带有"QUP" 后缀的指令还支持从非对齐地址中提取出 16-byte 对齐数据的功能。

表 1.6-8. 运算指令

指令	定义
EE.VMULAS.[U/S][8/16].ACCX	有符号/无符号按 1-字节/2-字节分片区执行向量乘累加，结果暂存到 ACCX 寄存器。
EE.VMULAS.[U/S][8/16].ACCX.LD.IP	有符号/无符号按 1-字节/2-字节分片区执行向量乘累加，结果暂存到 ACCX 寄存器。同时从内存读取 16-字节数据。并对地址增加立即数大小。
EE.VMULAS.[U/S][8/16].ACCX.LD.XP	有符号/无符号按 1-字节/2-字节分片区执行向量乘累加，结果暂存到 ACCX 寄存器。同时从内存读取 16-字节数据。并对地址增加 AR 寄存器内数值大小。
EE.VMULAS.[U/S][8/16].ACCX.LD.IP.QUP	有符号/无符号按 1-字节/2-字节分片区执行向量乘累加，结果暂存到 ACCX 寄存器。同时从内存读取 16-字节数据，并处理出一个 16-字节对齐数据。并对地址增加立即数大小。
EE.VMULAS.[U/S][8/16].ACCX.LD.XP.QUP	有符号/无符号按 1-字节/2-字节分片区执行向量乘累加，结果暂存到 ACCX 寄存器。同时从内存读取 16-字节数据，并处理出一个 16-字节对齐数据。并对地址增加 AR 寄存器内数值大小。
EE.VMULAS.[U/S][8/16].QACC	有符号/无符号按 1-字节/2-字节分片区执行向量乘，随后将乘法结果累加到 QACC_H 和 QACC_L 寄存器相应的比特片区。
EE.VMULAS.[U/S][8/16].QACC.LD.IP	有符号/无符号按 1-字节/2-字节分片区执行向量乘，随后将乘法结果累加到 QACC_H 和 QACC_L 寄存器相应的比特片区。同时从内存读取 16-字节数据。并对地址增加立即数大小。
EE.VMULAS.[U/S][8/16].QACC.LD.XP	有符号/无符号按 1-字节/2-字节分片区执行向量乘，随后将乘法结果累加到 QACC_H 和 QACC_L 寄存器相应的比特片区。同时从内存读取 16-字节数据。并对地址增加 AR 寄存器内数值大小。

指令	定义
EE.VMULAS.[U/S][8/16].QACC.LDBC.INCP	有符号/无符号按 1-字节/2-字节分片区执行向量乘，随后将乘法结果累加到 QACC_H 和 QACC_L 寄存器相应的比特片区。同时从内存读取 1-字节/2-字节数据广播到 128-bit QR 寄存器。并对地址增加 16。
EE.VMULAS.[U/S][8/16].QACC.LD.IP.QUP	有符号/无符号按 1-字节/2-字节分片区执行向量乘，随后将乘法结果累加到 QACC_H 和 QACC_L 寄存器相应的比特片区。同时从内存读取 16-字节数据，并处理出一个 16-字节对齐数据。并对地址增加立即数大小。
EE.VMULAS.[U/S][8/16].QACC.LD.XP.QUP	有符号/无符号按 1-字节/2-字节分片区执行向量乘，随后将乘法结果累加到 QACC_H 和 QACC_L 寄存器相应的比特片区。同时从内存读取 16-字节数据，并处理出一个 16-字节对齐数据。并对地址增加 AR 寄存器内数值大小。
EE.VMULAS.[U/S][8/16].QACC.LDBC.INCP.QUP	有符号/无符号按 1-字节/2-字节分片区执行向量乘，随后将乘法结果累加到 QACC_H 和 QACC_L 寄存器相应的比特片区。同时从内存读取 1-字节/2-字节数据广播到 128-bit QR 寄存器，并处理出一个 16-字节对齐数据。并对地址增加 16。

向量和标量乘累加

该类指令的功能与基于 QACC_H 和 QACC_L 寄存器的向量乘累加指令相似，区别在于双目操作数中一个是向量，一个是标量。另外同样包含在向量运算的同时执行访存操作的指令。

表 1.6-9. 运算指令

指令	定义
EE.VSMULAS.S[8/16].QACC	有符号 1-字节/2-字节分片的向量与标量乘累加。
EE.VSMULAS.S[8/16].QACC.LD.INCP	有符号 1-字节/2-字节分片的向量与标量乘累加。同时从内存读取 16-字节数据。并对地址增加 16。

其他

这部分指令主要包含对 QACC_H、QACC_L 以及 ACCX 中乘累加结果进行算术右移处理的指令。用户通过设置移位数值来获得预期精度范围内的乘累加结果。
此外还包含条件使能的向量乘法指令。

表 1.6-10. 运算指令

指令	定义
EE.SRCMB.S[8/16].QACC	对 QACC_H, QACC_L 寄存器内数值按片区进行符号右移。
EE.SRS.ACCX	对 ACCX 寄存器内数值进行符号右移。
EE.VRELU.S[8/16]	条件使能的向量与标量乘法指令。
EE.VPRELU.S[8/16]	条件使能的向量与向量乘法指令。

1.6.5 比较指令

向量比较指令，支持 1-字节，2-字节以及 4-字节为数据比较单位的多种比较指令。包括有取两者间较大值、取两者间较小值、前者等于后者置 1 反之 0、前者大于后者置 1 反之 0、前者小于后者置 1 反之 0。

考虑到向量运算所需的输入输出操作数都是存储在内存中，为了减少额外执行读取内存操作，提升代码执行速度，还设计了在向量运算同时执行 16-字节访存的指令，并在访存结束后增加 16 到访存地址，使其指向下一个 16-字节的内存地址。用户可以根据实际算法需求，选择相应的指令。

表 1.6-11. 比较指令

指令	定义
EE.VMAX.S[8/16/32]	取两个 1-字节/2-字节/4-字节数据间较大值。
EE.VMAX.S[8/16/32].LD.INCP	取两个 1-字节/2-字节/4-字节数据间较大值。同时从内存读取 16-字节数据。
EE.VMAX.S[8/16/32].ST.INCP	取两个 1-字节/2-字节/4-字节间数据较大值。同时将 16-字节数据写入内存。
EE.VMIN.S[8/16/32]	取两个 1-字节/2-字节/4-字节数据间较小值。
EE.VMIN.S[8/16/32].LD.INCP	取两个 1-字节/2-字节/4-字节数据间较小值。同时从内存读取 16-字节数据。
EE.VMIN.S[8/16/32].ST.INCP	取两个 1-字节/2-字节/4-字节数据间较小值。同时将 16-字节数据写入内存。
EE.VCMP.EQ.S[8/16/32]	两个 1-字节/2-字节/4-字节数据，前者等于后者将 1-字节/2-字节/4-字节所有比特位置 1 反之置 0。
EE.VCMP.LT.S[8/16/32]	两个 1-字节/2-字节/4-字节数据，前者小于后者将 1-字节/2-字节/4-字节所有比特位置 1 反之置 0。
EE.VCMP.GT.S[8/16/32]	两个 1-字节/2-字节/4-字节数据，前者大于后者将 1-字节/2-字节/4-字节所有比特位置 1 反之置 0。

1.6.6 按位逻辑操作指令

包含按位逻辑或、按位逻辑与、按位逻辑异或和按位取反指令。

表 1.6-12. 按位逻辑操作指令

指令	定义
EE.ORQ	按位逻辑或 $qa = qx \| qy$
EE.XORQ	按位逻辑异或 $qa = qx \hat{} qy$
EE.ANDQ	按位逻辑与 $qa = qx \& qy$
EE.NOTQ	按位取反 $qa = \sim qx$

1.6.7 移位指令

包含 4-字节为数据处理单位的向量左移和向量右移指令，16-字节拼接后左移指令和右移指令。前者移位数值由 SAR 寄存器决定。后者移位数值既可以由 SAR_BYTE 寄存器决定，也可以由立即数决定，还可以由 AR 寄存器低位比特数据决定。用户可以根据应用需求选择最合适的移位指令。

上述指令都是按照有符号数移位来执行移位操作。

表 1.6-13. 移位指令

指令	定义
EE.SRC.Q	16-字节拼接后逻辑右移，移位数值由 SAR_BYTE 寄存器决定。

指令	定义
EE.SRC.Q.QUP	16-字节拼接后逻辑右移，移位数值由 SAR_BYTE 寄存器决定。同时保存高 8-字节数据。
EE.SRC.Q.LD.XP	16-字节拼接后逻辑右移，移位数值由 SAR_BYTE 寄存器决定。同时从内存读取 16-字节数据，并对读地址进行寄存器加法操作。
EE.SRC.Q.LD.IP	16-字节拼接后逻辑右移，移位数值由 SAR_BYTE 寄存器决定。同时从内存读取 16-字节数据，并对读地址进行立即数加法操作。
EE.SLCI.2Q	16-字节拼接后逻辑左移，移位数值由立即数决定。
EE.SLCXXP.2Q	16-字节拼接后逻辑左移，移位数值由 AR 寄存器内数值决定。
EE.SRCI.2Q	16-字节拼接后逻辑右移，移位数值由立即数决定。
EE.SRCXXP.2Q	16-字节拼接后逻辑右移，移位数值由 AR 寄存器内数值决定。
EE.SRCQ.128.ST.INCP	16-字节拼接后逻辑右移，并将移位后得到的 16-字节写入内存。
EE.VSR.32	4-字节向量算术右移。
EE.VSL.32	4-字节向量算术左移。

1.6.8 FFT（快速傅立叶变换）专用指令

FFT（Fast Fourier Transform, 快速傅立叶变换）专用指令，包括蝶形运算指令，比特反转指令以及实数 FFT 指令。

蝶形运算指令

支持基-2 蝶形运算。

表 1.6-14. 蝶形运算指令

指令	定义
EE.FFT.R2BF.S16.	基-2 蝶形运算指令。
EE.FFT.R2BF.S16.ST.INCP	基-2 蝶形运算同时将其中 16-字节计算结果写入内存。
EE.FFT.CMUL.S16.LD.XP	基-2 蝶形复数运算指令。同时从内存读取 16-字节数据。
EE.FFT.CMUL.S16.ST.XP	基-2 蝶形复数运算指令。同时将 16-字节数据（由运算结果、QR 寄存器部分片区数值组成）写入内存。

比特反转指令

FFT_BIT_WIDTH 寄存器数值大小决定比特反转指令的反转位宽。

表 1.6-15. 比特反转指令

指令	定义
EE.BITREV	比特反转指令

实数 FFT 指令

单条实数 FFT 可以完成一系列复杂运算：加法、乘法、移位等。

表 1.6-16. 实数 FFT 指令

指令	定义
EE.FFT.AMS.S16.LD.INCP.UAUP	完成复杂运算的同时从内存读取 16-字节数据并处理出 16-字节对齐数据。
EE.FFT.AMS.S16.LD.INCP	完成复杂运算的同时从内存读取 16-字节数据。并对地址增加 16。
EE.FFT.AMS.S16.LD.R32.DECP	完成复杂运算的同时从内存读取 16-字节数据（翻转字顺序）。并对地址增加 16。
EE.FFT.AMS.S16.ST.INCP	完成复杂运算的同时将 16-字节数据（由 AR 寄存器、QR 寄存器部分片区数值组成）写入内存。
EE.FFT.VST.R32.DECP	将 QR 寄存器按 2-字节分片区后移位结果组成的 16-字节写入内存。

1.6.9 GPIO 控制指令

包含驱动 GPIO_OUT 以及获取 GPIO_IN 状态的指令。

表 1.6-17. GPIO 控制指令

指令	定义
EE.WR_MASK_GPIO_OUT	通过掩码的方式置位 GPIO_OUT 比特位。
EE.SET_BIT_GPIO_OUT	置位 GPIO_OUT 比特位。
EE.CLR_BIT_GPIO_OUT	清除 GPIO_OUT 比特位。
EE.GET_GPIO_IN	获取 GPIO_IN 状态。

1.6.10 处理器控制指令

如章节 1.5.1.2 所述，ESP32-S3 处理器内部有多种特殊寄存器。为了方便用户读写这些特殊寄存器的数值，提供如下几种处理器控制指令实现特殊寄存器与 AR 寄存器之间的数据传输。

RSR.* (Read Special register)

指令可以读取处理器内自带的特殊寄存器内的数值到 AR 寄存器，“*”为特殊寄存器名，仅包括 SAR 寄存器。

WSR.*（Write Special register）

指令可以通过 AR 寄存器修改处理器内自带的特殊寄存器内的数值，“*”为特殊寄存器名，仅包括 SAR 寄存器。

XSR.*（Exchange Special register）

指令交换 AR 寄存器与特殊寄存器内的数值，“*”为特殊寄存器名，仅包括 SAR 寄存器。

RUR.* (Read User-defined register)

指令可以读取处理器内自定义的特殊寄存器内的数值到 AR 寄存器，“*”为特殊寄存器名，包括 SAR_BYTE、ACCX、QACC_H、QACC_L、FFT_BIT_WIDTH、UA_STATE 寄存器。

WUR.*（Write User-defined register）

指令可以通过 AR 寄存器修改自定义的特殊寄存器内的数值，“*”为特殊寄存器名，包括 SAR_BYTE、ACCX、QACC_H、QACC_L、FFT_BIT_WIDTH、UA_STATE 寄存器。

对于位宽超过 32-bit 特殊寄存器，通过“_n”后缀区分对同一个特殊寄存器的不同 32-bit 片区数据进行读写的指令。以读取 ACCX 寄存器数值指令为例，有两条 RUR.* 指令：RUR.ACCX_0，RUR.ACCX_1。前者读取 ACCX 低 32-bit 数据写入 AR 寄存器；后者读取 ACCX 余下的高 8-bit 数据 0 扩展后写入 AR 寄存器。相应地，QACC_H 和 QACC_L 通过 5 个 AR 寄存器进行数据传输。

1.7 指令性能

对于采用流水线设计的处理器而言，理想情况下，每个处理器周期发出一条指令到流水线上。ESP32-S3 Xtensa 处理器采用 5 级流水线技术，包含 I（取指令）、R（译码）、E（执行）、M（访存）、W（写回）。处理器在每个流水级工作内容如表 1.7-1 所示。

表 1.7-1. 5 级 Xtensa 处理器流水线

流水级	数字编号	执行操作
I	-	指令对齐（同时支持 24-bit、32-bit 指令）
R	0	读取通用寄存器 AR、QR 指令译码、内部锁检测、操作数转发
E	1	对于运算指令，ALU 单元（加、减、乘等）工作 对于读取内存指令，生成访存虚地址 对于分支跳转指令，进行跳转地址选择
M	2	发出读写内存访问
W	3	计算结果及读取内存数据写回寄存器

一条指令的所有操作数以及操作所需要的硬件资源都已经准备完成后才能被处理器发出到流水线上。但实际程序运行过程中存在如下几种冒险问题，导致流水线停顿，指令被延迟执行。

1.7.1 数据冒险

当指令 A 写结果到寄存器 X（包括显性的通用寄存器与隐性的特殊寄存器），而另一条指令 B 需要用到寄存器 X 作为输入操作数时。此时称指令 B 依赖于指令 A。如果指令 A 是在 SA 流水级（这一流水级的结尾）准备好写入寄存器 X 的结果，指令 B 是在 SB 流水级（这一流水级的开始）读取寄存器 X 中的数据。那么指令 A 必须早于指令 B D=max(SA - SB+1, 0) 个周期发出。

如果处理器在指令 A 后小于 D 周期数内取到指令 B，处理器会延迟发出指令 B 直到满足 D 个周期数之后。处理器因为流水线相互作用延迟某条指令的行为被称之为内部锁。

假设指令 A 的 SA 流水级为 W，指令 B 的 SB 流水级为 E。如图 1.7-1 所示，指令 B 最终发出到流水线上的时刻，晚于指令 A D=max(2-1+1, 0)=2 个周期。

当指令的输出操作数被设计成在某个流水级的结尾时可获取时，意味着该指令的运算结束。通常情况下，对该结果数据有所依赖的指令必须等到输出操作数被写入相应的寄存器后，才能再从对应寄存器中获取更新后的数据。Xtensa 处理器支持“bypass”操作，它会检测指令的输入操作数在哪条指令的哪个流水级上被生成，不需要等该数据被写入寄存器，便可直接从生成流水级转发到需要用到的流水级上。

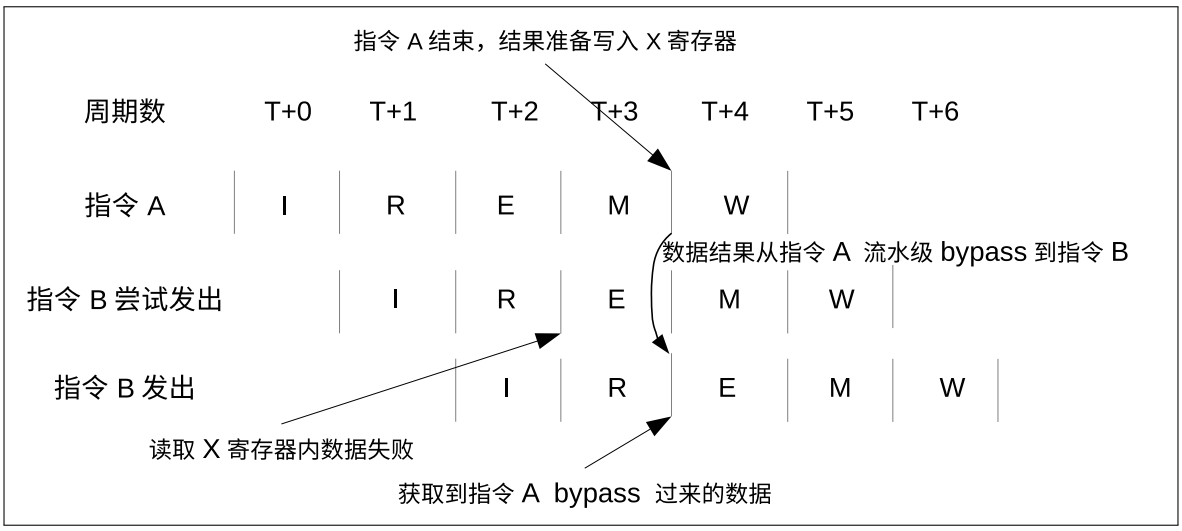


图 1.7-1. 指令操作数依赖导致内部锁

指令之间的数据依赖关系取决于操作数之间的依赖关系以及读写所处的流水级。表 1.7-2 列出了 ESP32-S3 扩展指令所有操作数，包括隐性的特殊寄存器的写 (def)，读 (use) 流水级信息。

表 1.7-2. 扩展指令流水级数

指令	操作数流水级		特殊寄存器流水级	
	Use	Def	Use	Def
EE.ANDQ	qx 1, qy 1	qa 1	—	—
EE.BITREV	ax 1	qa 1, ax 1	FFT_BIT_WIDTH 1	—
EE.CLR_BIT_GPIO_OUT	—	—	GPIO_OUT 1	GPIO_OUT 1
EE.CMUL.S16	qx 1, qy 1	qz 2	SAR 1	—
EE.CMUL.S16.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qz 2	SAR 1	—
EE.CMUL.S16.ST.INCP	qv 2, as 1, qx 1, qy 1	as 1, qz 2	SAR 1	—
EE.FFT.AMS.S16.LD.INCP	as 1, qx 1, qy 1, qm 1	qu 2, as 1, qz 2, qz1 2	SAR 1	—
EE.FFT.AMS.S16.LD.INCP.UAUP	as 1, qx 1, qy 1, qm 1	qu 2, as 1, qz 2, qz1 2	SAR 1, SAR_BYTE 1, UA_STATE 1	UA_STATE 1
EE.FFT.AMS.S16.LD.R32.DECP	as 1, qx 1, qy 1, qm 1	qu 2, as 1, qz 2, qz1 2	SAR 1	—
EE.FFT.AMS.S16.ST.INCP	qv 2, as0 1, as 1, qx 1, qy 1, qm 1	qz1 2, as0 2, as 1	SAR 1	—
EE.FFT.CMUL.S16.LD.XP	as 1, ad 1, qx 1, qy 1	qu 2, as 1, qz 2	SAR 1	—
EE.FFT.CMUL.S16.ST.XP	qx 1, qy 1, qv 2, as 1, ad 1	as 1	SAR 1	—
EE.FFT.R2BF.S16	qx 1, qy 1	qa0 1, qa1 1	—	—
EE.FFT.R2BF.S16.ST.INCP	qx 1, qy 1, as 1	qa0 1, as 1	—	—

EE.FFT.VST.R32.DECP	qv 2, as 1	as 1	—	—
EE.GET_GPIO_IN	—	au 1	GPIO_IN 1	—
EE.LD.128.USAR.IP	as 1	qu 2, as 1	—	SAR_BYTE 1
EE.LD.128.USAR.XP	as 1, ad 1	qu 2, as 1	—	SAR_BYTE 1
EE.LD.ACCX.IP	as 1	as 1	—	ACCX 2
EE.LD.QACC_H.H.32.IP	as 1	as 1	QACC_H 1	QACC_H 2
EE.LD.QACC_H.L.128.IP	as 1	as 1	QACC_H 1	QACC_H 2
EE.LD.QACC_L.H.32.IP	as 1	as 1	QACC_L 1	QACC_L 2
EE.LD.QACC_L.L.128.IP	as 1	as 1	QACC_L 1	QACC_L 2
EE.LD.UA_STATE.IP	as 1	as 1	—	UA_STATE 2
EE.LDF.128.IP	as 1	fu3 2, fu2 2, fu1 2, fu0 2, as 1	—	—
EE.LDF.128.XP	as 1, ad 1	fu3 2, fu2 2, fu1 2, fu0 2, as 1	—	—
EE.LDF.64.IP	as 1	fu1 2, fu0 2, as 1	—	—
EE.LDF.64.XP	as 1, ad 1	fu1 2, fu0 2, as 1	—	—
EE.LDQA.S16.128.IP	as 1	as 1	—	QACC_L 2, QACC_H 2
EE.LDQA.S16.128.XP	as 1, ad 1	as 1	—	QACC_L 2, QACC_H 2
EE.LDQA.S8.128.IP	as 1	as 1	—	QACC_L 2, QACC_H 2
EE.LDQA.S8.128.XP	as 1, ad 1	as 1	—	QACC_L 2, QACC_H 2
EE.LDQA.U16.128.IP	as 1	as 1	—	QACC_L 2, QACC_H 2
EE.LDQA.U16.128.XP	as 1, ad 1	as 1	—	QACC_L 2, QACC_H 2
EE.LDQA.U8.128.IP	as 1	as 1	—	QACC_L 2, QACC_H 2
EE.LDQA.U8.128.XP	as 1, ad 1	as 1	—	QACC_L 2, QACC_H 2
EE.LDXQ.32	qs 1, as 1	qu 2	—	—
EE.MOV.S16.QACC	qs 1	—	—	QACC_L 1, QACC_H 1
EE.MOV.S8.QACC	qs 1	—	—	QACC_L 1, QACC_H 1
EE.MOV.U16.QACC	qs 1	—	—	QACC_L 1, QACC_H 1
EE.MOV.U8.QACC	qs 1	—	—	QACC_L 1, QACC_H 1
EE.MOVI.32.A	qs 1	au 1	—	—
EE.MOVI.32.Q	as 1	qu 1	—	—
EE.NOTQ	qx 1	qa 1	—	—

EE.ORQ	qx 1, qy 1	qa 1	—	—
EE.SET_BIT_GPIO_OUT	—	—	GPIO_OUT 1	GPIO_OUT 1
EE.SLCI.2Q	qs1 1, qs0 1	qs1 1, qs0 1	—	—
EE.SLCXXP.2Q	qs1 1, qs0 1, as 1, ad 1	qs1 1, qs0 1, as 1	—	—
EE.SRC.Q	qs0 1, qs1 1	qa 1	SAR_BYTE 1	—
EE.SRC.Q.LD.IP	as 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1	—
EE.SRC.Q.LD.XP	as 1, ad 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1	—
EE.SRC.Q.QUP	qs0 1, qs1 1	qa 1, qs0 1	SAR_BYTE 1	—
EE.SRCI.2Q	qs1 1, qs0 1	qs1 1, qs0 1	—	—
EE.SRCMB.S16.QACC	as 1	qu 1	QACC_H 1, QACC_L 1	QACC_H 1, QACC_L 1
EE.SRCMB.S8.QACC	as 1	qu 1	QACC_H 1, QACC_L 1	QACC_H 1, QACC_L 1
EE.SRCQ.128.ST.INCP	qs0 1, qs1 1, as 1	as 1	SAR_BYTE 1	—
EE.SRCXXP.2Q	qs1 1, qs0 1, as 1, ad 1	qs1 1, qs0 1, as 1	—	—
EE.SRS.ACCX	as 1	au 1	ACCX 1	ACCX 1
EE.ST.ACCX.IP	as 1	as 1	ACCX 1	—
EE.ST.QACC_H.H.32.IP	as 1	as 1	QACC_H 1	—
EE.ST.QACC_H.L.128.IP	as 1	as 1	QACC_H 1	—
EE.ST.QACC_L.H.32.IP	as 1	as 1	QACC_L 1	—
EE.ST.QACC_L.L.128.IP	as 1	as 1	QACC_L 1	—
EE.ST.UA_STATE.IP	as 1	as 1	UA_STATE 1	—
EE.STF.128.IP	fv3 1, fv2 1, fv1 1, fv0 1, as 1	as 1	—	—
EE.STF.128.XP	fv3 1, fv2 1, fv1 1, fv0 1, as 1, ad 1	as 1	—	—
EE.STF.64.IP	fv1 1, fv0 1, as 1	as 1	—	—
EE.STF.64.XP	fv1 1, fv0 1, as 1, ad 1	as 1	—	—
EE.STXQ.32	qv 1, qs 1, as 1	—	—	—
EE.VADDS.S16	qx 1, qy 1	qa 1	—	—
EE.VADDS.S16.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VADDS.S16.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VADDS.S32	qx 1, qy 1	qa 1	—	—
EE.VADDS.S32.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VADDS.S32.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VADDS.S8	qx 1, qy 1	qa 1	—	—
EE.VADDS.S8.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—

EE.VADDS.S8.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VCMP.EQ.S16	qx 1, qy 1	qa 1	—	—
EE.VCMP.EQ.S32	qx 1, qy 1	qa 1	—	—
EE.VCMP.EQ.S8	qx 1, qy 1	qa 1	—	—
EE.VCMP.GT.S16	qx 1, qy 1	qa 1	—	—
EE.VCMP.GT.S32	qx 1, qy 1	qa 1	—	—
EE.VCMP.GT.S8	qx 1, qy 1	qa 1	—	—
EE.VCMP.LT.S16	qx 1, qy 1	qa 1	—	—
EE.VCMP.LT.S32	qx 1, qy 1	qa 1	—	—
EE.VCMP.LT.S8	qx 1, qy 1	qa 1	—	—
EE.VLD.128.IP	as 1	qu 2, as 1	—	—
EE.VLD.128.XP	as 1, ad 1	qu 2, as 1	—	—
EE.VLD.H.64.IP	as 1	qu 2, as 1	—	—
EE.VLD.H.64.XP	as 1, ad 1	qu 2, as 1	—	—
EE.VLD.L.64.IP	as 1	qu 2, as 1	—	—
EE.VLD.L.64.XP	as 1, ad 1	qu 2, as 1	—	—
EE.VLDBC.16	as 1	qu 2	—	—
EE.VLDBC.16.IP	as 1	qu 2, as 1	—	—
EE.VLDBC.16.XP	as 1, ad 1	qu 2, as 1	—	—
EE.VLDBC.32	as 1	qu 2	—	—
EE.VLDBC.32.IP	as 1	qu 2, as 1	—	—
EE.VLDBC.32.XP	as 1, ad 1	qu 2, as 1	—	—
EE.VLDBC.8	as 1	qu 2	—	—
EE.VLDBC.8.IP	as 1	qu 2, as 1	—	—
EE.VLDBC.8.XP	as 1, ad 1	qu 2, as 1	—	—
EE.VLDHBC.16.INCP	as 1	qu 2, qu1 2, as 1	—	—
EE.VMAX.S16	qx 1, qy 1	qa 1	—	—
EE.VMAX.S16.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VMAX.S16.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VMAX.S32	qx 1, qy 1	qa 1	—	—
EE.VMAX.S32.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VMAX.S32.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VMAX.S8	qx 1, qy 1	qa 1	—	—
EE.VMAX.S8.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VMAX.S8.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VMIN.S16	qx 1, qy 1	qa 1	—	—
EE.VMIN.S16.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VMIN.S16.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VMIN.S32	qx 1, qy 1	qa 1	—	—

EE.VMIN.S32.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VMIN.S32.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VMIN.S8	qx 1, qy 1	qa 1	—	—
EE.VMIN.S8.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VMIN.S8.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VMUL.S16	qx 1, qy 1	qz 2	SAR 1	—
EE.VMUL.S16.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qz 2	SAR 1	—
EE.VMUL.S16.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qz 2	SAR 1	—
EE.VMUL.S8	qx 1, qy 1	qz 2	SAR 1	—
EE.VMUL.S8.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qz 2	SAR 1	—
EE.VMUL.S8.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qz 2	SAR 1	—
EE.VMUL.U16	qx 1, qy 1	qz 2	SAR 1	—
EE.VMUL.U16.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qz 2	SAR 1	—
EE.VMUL.U16.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qz 2	SAR 1	—
EE.VMUL.U8	qx 1, qy 1	qz 2	SAR 1	—
EE.VMUL.U8.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qz 2	SAR 1	—
EE.VMUL.U8.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qz 2	SAR 1	—
EE.VMULAS.S16.ACCX	qx 1, qy 1	—	ACCX 2	ACCX 2
EE.VMULAS.S16.ACCX.LD.IP	as 1, qx 1, qy 1	qu 2, as 1	ACCX 2	ACCX 2
EE.VMULAS.S16.ACCX.LD.IP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, ACCX 2	ACCX 2
EE.VMULAS.S16.ACCX.LD.XP	as 1, ad 1, qx 1, qy 1	qu 2, as 1	ACCX 2	ACCX 2
EE.VMULAS.S16.ACCX.LD.XP.QUP	as 1, ad 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, ACCX 2	ACCX 2
EE.VMULAS.S16.QACC	qx 1, qy 1	—	QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S16.QACC.LD.IP	as 1, qx 1, qy 1	qu 2, as 1	QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S16.QACC.LD.IP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S16.QACC.LD.XP	as 1, ad 1, qx 1, qy 1	qu 2, as 1	QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S16.QACC.LD.XP.QUP	as 1, ad 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2

EE.VMULAS.S16.QACC.LDBC.INCP	as 1, qx 1, qy 1	qu 2, as 1	QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S16.QACC.LDBC.INCP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S8.ACCX	qx 1, qy 1	—	ACCX 2	ACCX 2
EE.VMULAS.S8.ACCX.LD.IP	as 1, qx 1, qy 1	qu 2, as 1	ACCX 2	ACCX 2
EE.VMULAS.S8.ACCX.LD.IP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, ACCX 2	ACCX 2
EE.VMULAS.S8.ACCX.LD.XP	as 1, ad 1, qx 1, qy 1	qu 2, as 1	ACCX 2	ACCX 2
EE.VMULAS.S8.ACCX.LD.XP.QUP	as 1, ad 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, ACCX 2	ACCX 2
EE.VMULAS.S8.QACC	qx 1, qy 1	—	QACC_L 2, QACC_H 2	QACC_L 2, QACC_H 2
EE.VMULAS.S8.QACC.LD.IP	as 1, qx 1, qy 1	qu 2, as 1	QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S8.QACC.LD.IP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S8.QACC.LD.XP	as 1, ad 1, qx 1, qy 1	qu 2, as 1	QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S8.QACC.LD.XP.QUP	as 1, ad 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S8.QACC.LDBC.INCP	as 1, qx 1, qy 1	qu 2, as 1	QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.S8.QACC.LDBC.INCP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.U16.ACCX	qx 1, qy 1	—	ACCX 2	ACCX 2
EE.VMULAS.U16.ACCX.LD.IP	as 1, qx 1, qy 1	qu 2, as 1	ACCX 2	ACCX 2
EE.VMULAS.U16.ACCX.LD.IP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, ACCX 2	ACCX 2
EE.VMULAS.U16.ACCX.LD.XP	as 1, ad 1, qx 1, qy 1	qu 2, as 1	ACCX 2	ACCX 2
EE.VMULAS.U16.ACCX.LD.XP.QUP	as 1, ad 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, ACCX 2	ACCX 2
EE.VMULAS.U16.QACC	qx 1, qy 1	—	QACC_L 2, QACC_H 2	QACC_L 2, QACC_H 2
EE.VMULAS.U16.QACC.LD.IP	as 1, qx 1, qy 1	qu 2, as 1	QACC_L 2, QACC_H 2	QACC_L 2, QACC_H 2

EE.VMULAS.U16.QACC.LD.IP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.U16.QACC.LD.XP	as 1, ad 1, qx 1, qy 1	qu 2, as 1	QACC_L 2, QACC_H 2	QACC_L 2, QACC_H 2
EE.VMULAS.U16.QACC.LD.XP.QUP	as 1, ad 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.U16.QACC.LDBC.INCP	as 1, qx 1, qy 1	qu 2, as 1	QACC_L 2, QACC_H 2	QACC_L 2, QACC_H 2
EE.VMULAS.U16.QACC.LDBC.INCP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.U8.ACCX	qx 1, qy 1	—	ACCX 2	ACCX 2
EE.VMULAS.U8.ACCX.LD.IP	as 1, qx 1, qy 1	qu 2, as 1	ACCX 2	ACCX 2
EE.VMULAS.U8.ACCX.LD.IP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, ACCX 2	ACCX 2
EE.VMULAS.U8.ACCX.LD.XP	as 1, ad 1, qx 1, qy 1	qu 2, as 1	ACCX 2	ACCX 2
EE.VMULAS.U8.ACCX.LD.XP.QUP	as 1, ad 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, ACCX 2	ACCX 2
EE.VMULAS.U8.QACC	qx 1, qy 1	—	QACC_L 2, QACC_H 2	QACC_L 2, QACC_H 2
EE.VMULAS.U8.QACC.LD.IP	as 1, qx 1, qy 1	qu 2, as 1	QACC_L 2, QACC_H 2	QACC_L 2, QACC_H 2
EE.VMULAS.U8.QACC.LD.IP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.U8.QACC.LD.XP	as 1, ad 1, qx 1, qy 1	qu 2, as 1	QACC_L 2, QACC_H 2	QACC_L 2, QACC_H 2
EE.VMULAS.U8.QACC.LD.XP.QUP	as 1, ad 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VMULAS.U8.QACC.LDBC.INCP	as 1, qx 1, qy 1	qu 2, as 1	QACC_L 2, QACC_H 2	QACC_L 2, QACC_H 2
EE.VMULAS.U8.QACC.LDBC.INCP.QUP	as 1, qx 1, qy 1, qs0 1, qs1 1	qu 2, as 1, qs0 1	SAR_BYTE 1, QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VPRELU.S16	qx 1, qy 1, ay 1	qz 2	—	—
EE.VPRELU.S8	qx 1, qy 1, ay 1	qz 2	—	—
EE.VRELU.S16	qs 1, ax 1, ay 1	qs 2	—	—
EE.VRELU.S8	qs 1, ax 1, ay 1	qs 2	—	—
EE.VSL.32	qs 1	qa 1	SAR 1	—

EE.VSMULAS.S16.QACC	qx 1, qy 1	—	QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VSMULAS.S16.QACC.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1	QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VSMULAS.S8.QACC	qx 1, qy 1	—	QACC_L 2, QACC_H 2	QACC_L 2, QACC_H 2
EE.VSMULAS.S8.QACC.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1	QACC_H 2, QACC_L 2	QACC_H 2, QACC_L 2
EE.VSR.32	qs 1	qa 1	SAR 1	—
EE.VST.128.IP	qv 1, as 1	as 1	—	—
EE.VST.128.XP	qv 1, as 1, ad 1	as 1	—	—
EE.VST.H.64.IP	qv 1, as 1	as 1	—	—
EE.VST.H.64.XP	qv 1, as 1, ad 1	as 1	—	—
EE.VST.L.64.IP	qv 1, as 1	as 1	—	—
EE.VST.L.64.XP	qv 1, as 1, ad 1	as 1	—	—
EE.VSUBS.S16	qx 1, qy 1	qa 1	—	—
EE.VSUBS.S16.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VSUBS.S16.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VSUBS.S32	qx 1, qy 1	qa 1	—	—
EE.VSUBS.S32.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VSUBS.S32.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VSUBS.S8	qx 1, qy 1	qa 1	—	—
EE.VSUBS.S8.LD.INCP	as 1, qx 1, qy 1	qu 2, as 1, qa 1	—	—
EE.VSUBS.S8.ST.INCP	qv 1, as 1, qx 1, qy 1	as 1, qa 1	—	—
EE.VUNZIP.16	qs0 1, qs1 1	qs0 1, qs1 1	—	—
EE.VUNZIP.32	qs0 1, qs1 1	qs0 1, qs1 1	—	—
EE.VUNZIP.8	qs0 1, qs1 1	qs0 1, qs1 1	—	—
EE.VZIP.16	qs0 1, qs1 1	qs0 1, qs1 1	—	—
EE.VZIP.32	qs0 1, qs1 1	qs0 1, qs1 1	—	—
EE.VZIP.8	qs0 1, qs1 1	qs0 1, qs1 1	—	—
EE.WR_MASK_GPIO_OUT	as 1, ax 1	—	GPIO_OUT 1	GPIO_OUT 1
EE.XORQ	qx 1, qy 1	qa 1	—	—
EE.ZERO.ACCX	—	—	—	ACCX 1
EE.ZERO.Q	—	qa 1	—	—
EE.ZERO.QACC	—	—	—	QACC_L 1, QACC_H 1

1.7.2 资源冒险

当多条指令同时调用相同的硬件资源时，处理器只能允许其中一条指令占用硬件资源，其余指令将被延迟。例如，处理器内只有 8 个 16-bit 乘法器。指令 C 在 M 流水级上需要用到这 8 个 16-bit 乘法器，指令 D 在它的 E

流水级上需要用到 4 个 16-bit 乘法器。如图 1.7-2 所示，指令 C 在第 T+0 周期发出，指令 D 在第 T+1 周期发出，有 4 个 16-bit 乘法器在第 T+3 周期同时被它们申请占用。此时处理器会延迟一个周期发出指令 D 到流水线上来避免与指令 C 的冲突。

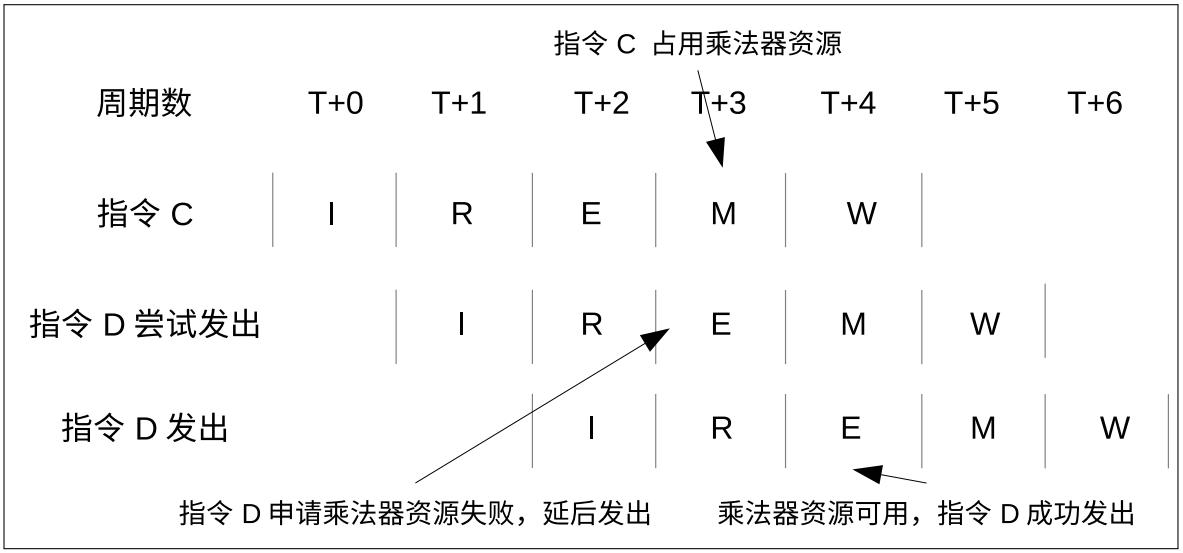


图 1.7-2. 硬件资源冒险

1.7.3 控制冒险

数据冒险、资源冒险可以通过调整代码顺序来优化，而控制冒险却很难优化。程序代码通常有许多条件选择语句，根据条件满足与否，执行不同的代码。编译器会将上述条件语句处理成分支跳转指令：条件满足就跳到目标地址执行相应的代码；条件不满足就按照顺序处理后续的指令。当条件满足时，如图 1.7-3 所示，处理器将从新的目标地址重新取指，此时流水线上处于 R、E 流水级的指令将被刷新掉，相当于流水线停滞了 2 个周期。

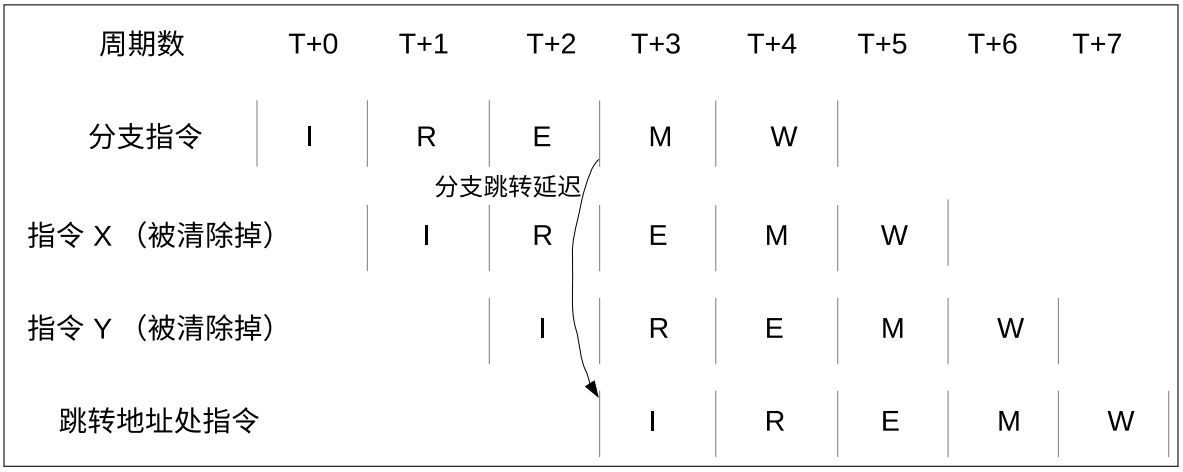


图 1.7-3. 控制冒险

1.8 扩展指令功能描述

建议在阅读本章节前，先阅读一下表 1.4-1，该表介绍了指令编码中指令域名及其含义。

[N:M] 是下文中采用的域的表达方式。比特上下限 N 和 M 都包含在内，表示域的位宽为 (N-M+1)。以 qa[2:0] 为例，qa 共有 3 (2-0+1) 个比特，分别是 bit0, bit1 和 bit2。qa[1] 表示 bit1 的值。

本章节按照指令名称中首字母大小顺序描述了 1.6 章节中提到的所有指令。每条指令的编码序列都采用小端比特序，如图 1.4-1 所示。

1.8.1 EE.ANDQ

Instruction Word

11	qa[2:1]	1101	qa[0]	011	qy[2:1]	00	qx[2:1]	qy[0]	qx[0]	0100

Assembler Syntax

EE.ANDQ qa, qx, qy

Description

该指令对 qx 和 qy 寄存器进行按位与操作，并将逻辑运算的结果写入 qa 寄存器。

Operation

```
qa = qx & qy
```

1.8.2 EE.BITREV

Instruction Word

11	qa[2:1]	1101	qa[0]	1111011	as[3:0]	0100

Assembler Syntax

EE.BITREV qa, as

Description

该指令根据特殊寄存器 FFT_BIT_WIDTH 的数值内容，调换不同的位宽数据的比特顺序。比较调换顺序后数据与原始数据的大小，取其中较大的数值高位补 0，填充成 16-bit 数据写入 qa 寄存器的对应数据段。下文中 Switchx 表示处理比特位宽为 x 的数据按比特反转的函数。Switch5 (0b10100) = 0b00101，此处的 0b10100 表示 5-bit 二进制数据。

Operation

```
  temp0[15:0] = as[15:0]
  temp1[15:0] = as[15:0] + 1
  temp2[15:0] = as[15:0] + 2
  temp3[15:0] = as[15:0] + 3
  temp4[15:0] = as[15:0] + 4
  temp5[15:0] = as[15:0] + 5
  temp6[15:0] = as[15:0] + 6
  temp7[15:0] = as[15:0] + 7
if FFT_BIT_WIDTH==3:
  Switch3(X[2:0]) = X[0:2]
  qa[ 15:  0] = {13'h0, max(tmp0[2:0], Switch3(tmp0[2:0]))}
  qa[ 31: 16] = {13'h0, max(tmp1[2:0], Switch3(tmp1[2:0]))}
  qa[ 47: 32] = {13'h0, max(tmp2[2:0], Switch3(tmp2[2:0]))}
  qa[ 63: 48] = {13'h0, max(tmp3[2:0], Switch3(tmp3[2:0]))}
  qa[ 79: 64] = {13'h0, max(tmp4[2:0], Switch3(tmp4[2:0]))}
  qa[ 95: 80] = {13'h0, max(tmp5[2:0], Switch3(tmp5[2:0]))}
  qa[127: 96] = 0
if FFT_BIT_WIDTH==4:
  Switch4(X[3:0]) = X[0:3]
  qa[ 15:  0] = {12'h0, max(tmp0[3:0], Switch4(tmp0[3:0]))}
  qa[ 31: 16] = {12'h0, max(tmp1[3:0], Switch4(tmp1[3:0]))}
  qa[ 47: 32] = {12'h0, max(tmp2[3:0], Switch4(tmp2[3:0]))}
  qa[ 63: 48] = {12'h0, max(tmp3[3:0], Switch4(tmp3[3:0]))}
  qa[ 79: 64] = {12'h0, max(tmp4[3:0], Switch4(tmp4[3:0]))}
  qa[ 95: 80] = {12'h0, max(tmp5[3:0], Switch4(tmp5[3:0]))}
  qa[111: 96] = {12'h0, max(tmp6[3:0], Switch4(tmp6[3:0]))}
  qa[127:112] = {12'h0, max(tmp7[3:0], Switch4(tmp7[3:0]))}
  ...
if FFT_BIT_WIDTH==10:
  Switch10(X[9:0]) = X[0:9]
  qa[ 15:  0] = {6'h0, max(tmp0[9:0], Switch10(tmp0[9:0]))}
  qa[ 31: 16] = {6'h0, max(tmp1[9:0], Switch10(tmp1[9:0]))}
  qa[ 47: 32] = {6'h0, max(tmp2[9:0], Switch10(tmp2[9:0]))}
  qa[ 63: 48] = {6'h0, max(tmp3[9:0], Switch10(tmp3[9:0]))}
  qa[ 79: 64] = {6'h0, max(tmp4[9:0], Switch10(tmp4[9:0]))}
  qa[ 95: 80] = {6'h0, max(tmp5[9:0], Switch10(tmp5[9:0]))}
```

```
37  qa[111: 96] = {6'h0, max(tmp6[9:0], Switch10(tmp6[9:0]))}
38  qa[127:112] = {6'h0, max(tmp7[9:0], Switch10(tmp7[9:0]))}
39
40  as[31:0] = as[31:0] + 8
```

1.8.3 EE.CLR_BIT_GPIO_OUT

Instruction Word

011101100100	imm256[7:0]	0100

Assembler Syntax

EE.CLR_BIT_GPIO_OUT 0..255

Description

该指令为 CPU GPIO 专用指令。功能为清零 GPIO_OUT 某些比特。清零的内容取决于 8-bit 立即数 imm256。

Operation

```
GPIO_OUT[7:0] =  (GPIO_OUT[7:0] & ~imm256[7:0])
```

1.8.4 EE.CMUL.S16

Instruction Word

10	qz[2:1]	1110	qz[0]	qy[2]	0	qy[1:0]	qx[2:0]	00	sel4[1:0]	0100

Assembler Syntax

EE.CMUL.S16 qz, qx, qy, 0..3

Description

该指令实现了 16-bit 有符号复数乘法。立即数 sel4 的范围为 0 ~ 3，指定两个 QR 寄存器 qx、qy 中 32-bit 进行复数乘法运算。复数的实部和虚部分别存储在 32-bit 的高 16-bit 和低 16-bit。计算得到的实部、虚部结果存储到 qz 寄存器对应位置的 32-bit 中。

Operation

```
if sel4 == 0:
  qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0] - qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
  qz[ 31: 16] = (qx[ 15:  0] * qy[ 31: 16] + qx[ 31: 16] * qy[ 15:  0]) >> SAR[5:0]
  qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32] - qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
  qz[ 63: 48] = (qx[ 47: 32] * qy[ 63: 48] + qx[ 63: 48] * qy[ 47: 32]) >> SAR[5:0]
if sel4 == 1:
  qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64] - qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
  qz[ 95: 80] = (qx[ 79: 64] * qy[ 95: 80] + qx[ 95: 80] * qy[ 79: 64]) >> SAR[5:0]
  qz[111: 96] = (qx[111: 96] * qy[111: 96] - qx[127:112] * qy[127:112]) >> SAR[5:0]
  qz[127:112] = (qx[111: 96] * qy[127:112] + qx[127:112] * qy[111: 96]) >> SAR[5:0]
if sel4 == 2:
  qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0] + qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
  qz[ 31: 16] = (qx[ 15:  0] * qy[ 31: 16] - qx[ 31: 16] * qy[ 15:  0]) >> SAR[5:0]
  qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32] + qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
  qz[ 63: 48] = (qx[ 47: 32] * qy[ 63: 48] - qx[ 63: 48] * qy[ 47: 32]) >> SAR[5:0]
if sel4 == 3:
  qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64] + qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
  qz[ 95: 80] = (qx[ 79: 64] * qy[ 95: 80] - qx[ 95: 80] * qy[ 79: 64]) >> SAR[5:0]
  qz[111: 96] = (qx[111: 96] * qy[111: 96] + qx[127:112] * qy[127:112]) >> SAR[5:0]
  qz[127:112] = (qx[111: 96] * qy[127:112] - qx[127:112] * qy[111: 96]) >> SAR[5:0]
```

1.8.5 EE.CMUL.S16.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	000	qu[0]	qz[2:0]	qx[1:0]	qy[2:1]	11	sel4[1:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.CMUL.S16.LD.INCP qu, as, qz, qx, qy, 0..3

Description

该指令实现了 16-bit 有符号复数乘法。立即数 sel4 的范围为 0 ~ 7，指定两个 QR 寄存器 qx、qy 中 32-bit 进行复数乘法运算。复数的实部和虚部分别存储在 32-bit 的高 16-bit 和低 16-bit。计算得到的实部、虚部结果存储到 qz 寄存器对应位置的 32-bit 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
if sel4 == 0:
  qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0] - qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
  qz[ 31: 16] = (qx[ 15:  0] * qy[ 31: 16] + qx[ 31: 16] * qy[ 15:  0]) >> SAR[5:0]
  qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32] - qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
  qz[ 63: 48] = (qx[ 47: 32] * qy[ 63: 48] + qx[ 63: 48] * qy[ 47: 32]) >> SAR[5:0]
if sel4 == 1:
  qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64] - qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
  qz[ 95: 80] = (qx[ 79: 64] * qy[ 95: 80] + qx[ 95: 80] * qy[ 79: 64]) >> SAR[5:0]
  qz[111: 96] = (qx[111: 96] * qy[111: 96] - qx[127:112] * qy[127:112]) >> SAR[5:0]
  qz[127:112] = (qx[111: 96] * qy[127:112] + qx[127:112] * qy[111: 96]) >> SAR[5:0]
if sel4 == 2:
  qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0] + qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
  qz[ 31: 16] = (qx[ 15:  0] * qy[ 31: 16] - qx[ 31: 16] * qy[ 15:  0]) >> SAR[5:0]
  qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32] + qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
  qz[ 63: 48] = (qx[ 47: 32] * qy[ 63: 48] - qx[ 63: 48] * qy[ 47: 32]) >> SAR[5:0]
if sel4 == 3:
  qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64] + qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
  qz[ 95: 80] = (qx[ 79: 64] * qy[ 95: 80] - qx[ 95: 80] * qy[ 79: 64]) >> SAR[5:0]
  qz[111: 96] = (qx[111: 96] * qy[111: 96] + qx[127:112] * qy[127:112]) >> SAR[5:0]
  qz[127:112] = (qx[111: 96] * qy[127:112] - qx[127:112] * qy[111: 96]) >> SAR[5:0]

  qu[127:0] = load128({as[31:4],4{0}})
  as[31:0] = as[31:0] + 16
```

1.8.6 EE.CMUL.S16.ST.INCP

Instruction Word

11100100	qy[0]	qv[2:0]	0	qz[2:0]	qx[1:0]	qy[2:1]	00	sel4[1:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.CMUL.S16.ST.INCP qv, as, qz, qx, qy, sel4

Description

该指令实现了 16-bit 有符号复数乘法。立即数 sel4 的范围为 0 ~ 7，指定两个 QR 寄存器 qx、qy 中 32-bit 进行复数乘法运算。复数的实部和虚部分别存储在 32-bit 的高 16-bit 和低 16-bit。计算得到的实部、虚部结果存储到 qz 寄存器对应位置的 32-bit 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将 16-byte qv 寄存器数据写入到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
if sel4 == 0:
  qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0] - qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
  qz[ 31: 16] = (qx[ 15:  0] * qy[ 31: 16] + qx[ 31: 16] * qy[ 15:  0]) >> SAR[5:0]
  qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32] - qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
  qz[ 63: 48] = (qx[ 47: 32] * qy[ 63: 48] + qx[ 63: 48] * qy[ 47: 32]) >> SAR[5:0]
if sel4 == 1:
  qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64] - qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
  qz[ 95: 80] = (qx[ 79: 64] * qy[ 95: 80] + qx[ 95: 80] * qy[ 79: 64]) >> SAR[5:0]
  qz[111: 96] = (qx[111: 96] * qy[111: 96] - qx[127:112] * qy[127:112]) >> SAR[5:0]
  qz[127:112] = (qx[111: 96] * qy[127:112] + qx[127:112] * qy[111: 96]) >> SAR[5:0]
if sel4 == 2:
  qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0] + qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
  qz[ 31: 16] = (qx[ 15:  0] * qy[ 31: 16] - qx[ 31: 16] * qy[ 15:  0]) >> SAR[5:0]
  qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32] + qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
  qz[ 63: 48] = (qx[ 47: 32] * qy[ 63: 48] - qx[ 63: 48] * qy[ 47: 32]) >> SAR[5:0]
if sel4 == 3:
  qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64] + qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
  qz[ 95: 80] = (qx[ 79: 64] * qy[ 95: 80] - qx[ 95: 80] * qy[ 79: 64]) >> SAR[5:0]
  qz[111: 96] = (qx[111: 96] * qy[111: 96] + qx[127:112] * qy[127:112]) >> SAR[5:0]
  qz[127:112] = (qx[111: 96] * qy[127:112] - qx[127:112] * qy[111: 96]) >> SAR[5:0]

  qv[127:0] => store128({as[31:4],4{0}})
  as[31:0] = as[31:0] + 16
```

1.8.7 EE.FFT.AMS.S16.LD.INCP

Instruction Word

110100	sel2[0]	qz1[2]	qz[0]	qy[2:0]	qz1[1]	qm[2:0]	qx[1:0]	qz[2:1]	qz1[0]	qu[2:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.FFT.AMS.S16.LD.INCP qu, as, qz, qz1, qx, qy, qm, sel2

Description

该指令为 FFT 专用指令。对 16-bit 数据段进行加减、乘、加减、移位操作。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
  temp0[15:0] = qx[ 47: 32] + qy[ 47: 32]
  temp1[15:0] = qx[ 63: 48] - qy[ 63: 48]

if sel2==0:
  temp2[15:0] = ((qx[ 47: 32] - qy[ 47: 32]) * qm[ 47: 32] - (qx[ 63: 48] + qy[ 63: 48]) * qm[ 63: 48]) >> SAR
  temp3[15:0] = ((qx[ 47: 32] - qy[ 47: 32]) * qm[ 63: 48] + (qx[ 63: 48] + qy[ 63: 48]) * qm[ 47: 32]) >> SAR
if sel2==1:
  temp2[15:0] = ((qx[ 63: 48] + qy[ 63: 48]) * qm[ 63: 48] + (qx[ 47: 32] - qy[ 47: 32]) * qm[ 47: 32]) >> SAR
  temp3[15:0] = ((qx[ 63: 48] + qy[ 63: 48]) * qm[ 47: 32] - (qx[ 47: 32] - qy[ 47: 32]) * qm[ 63: 48]) >> SAR

  qz[47: 32] = temp0[15:0] + temp2[15:0]
  qz[63: 48] = temp1[15:0] + temp3[15:0]
  qz1[47: 32] = temp0[15:0] - temp2[15:0]
  qz1[63: 48] = temp3[15:0] - temp1[15:0]
  qu = load128({as[31:4],4{0}})
  as[31:0] = as[31:0] + 16
```

1.8.8 EE.FFT.AMS.S16.LD.INCP.UAUP

Instruction Word

110101	sel2[0]	qz1[2]	qz[0]	qy[2:0]	qz1[1]	qm[2:0]	qx[1:0]	qz[2:1]	qz1[0]	qu[2:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.FFT.AMS.S16.LD.INCP.UAUP qu, as, qz, qz1, qx, qy, qm, sel2

Description

该指令为 FFT 专用指令。对 16-bit 数据段进行加减、乘、加减、移位操作。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据。将载入数据与特殊寄存器 UA_STATE 拼接成 32-byte 数据，右移 SAR_BYTE 数值乘以 8 的结果，并将移位后结果的低 16-byte 赋值给 qu 寄存器。同时用载入的 16-byte 数据更新 UA_STATE 寄存器的数据内容。访存结束后，as 寄存器内数值增加 16。

Operation

```
  temp0[15:0] = qx[ 15:  0] + qy[ 15:  0]
  temp1[15:0] = qx[ 31: 16] - qy[ 31: 16]

if sel2==0:
  temp2[15:0] = ((qx[ 15:  0] - qy[ 15:  0]) * qm[ 15:  0] - (qx[ 31: 16] + qy[ 31: 16]) * qm[ 31: 16]) >> SAR
  temp3[15:0] = ((qx[ 15:  0] - qy[ 15:  0]) * qm[ 31: 16] + (qx[ 31: 16] + qy[ 31: 16]) * qm[ 15:  0]) >> SAR
if sel2==1:
  temp2[15:0] = ((qx[ 31: 16] + qy[ 31: 16]) * qm[ 31: 16] + (qx[ 15:  0] - qy[ 15:  0]) * qm[ 15:  0]) >> SAR
  temp3[15:0] = ((qx[ 31: 16] + qy[ 31: 16]) * qm[ 15:  0] - (qx[ 15:  0] - qy[ 15:  0]) * qm[ 31: 16]) >> SAR

  dataIn[127:0] = load128({as[31:4],4{0}})
  qz[15:  0] = temp0[15:0] + temp2[15:0]
  qz[31: 16] = temp1[15:0] + temp3[15:0]
  qz1[15: 0] = temp0[15:0] - temp2[15:0]
  qz1[31:16] = temp3[15:0] - temp1[15:0]
  qu[127: 0] = {dataIn[127:0], UA_STATE[127:0]} >> {SAR_BYTE[3:0] << 3}
  UA_STATE[127:0] = dataIn[127:0]
  as[31:0] = as[31:0] + 16
```

1.8.9 EE.FFT.AMS.S16.LD.R32.DECP

Instruction Word

110110	sel2[0]	qz1[2]	qz[0]	qy[2:0]	qz1[1]	qm[2:0]	qx[1:0]	qz[2:1]	qz1[0]	qu[2:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.FFT.AMS.S16.LD.R32.DECP qu, as, qz, qz1, qx, qy, qm, sel2

Description

该指令为 FFT 专用指令。对 16-bit 数据段进行加减、乘、加减、移位操作。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，将载入数据按 word 大端序存入 qu 寄存器，即载入数据的 [127: 96] 段存入 qu 寄存器的 [31:0] 段。访存结束后，as 寄存器内数值减少 16。

Operation

```
  temp0[15:0] = qx[ 79: 64] + qy[ 79: 64]
  temp1[15:0] = qx[ 95: 80] - qy[ 95: 80]

if sel2==0:
  temp2[15:0] = ((qx[ 79: 64] - qy[ 79: 64]) * qm[ 79: 64] - (qx[ 95: 80] + qy[ 95:
      80]) * qm[ 95: 80]) >> SAR
  temp3[15:0] = ((qx[ 79: 64] - qy[ 79: 64]) * qm[ 95: 80] + (qx[ 95: 80] + qy[ 95:
      80]) * qm[ 79: 64]) >> SAR
if sel2==1:
  temp2[15:0] = ((qx[ 95: 80] + qy[ 95: 80]) * qm[ 95: 80] + (qx[ 79: 64] - qy[ 79:
      64]) * qm[ 79: 64]) >> SAR
  temp3[15:0] = ((qx[ 95: 80] + qy[ 95: 80]) * qm[ 79: 64] - (qx[ 79: 64] - qy[ 79:
      64]) * qm[ 95: 80]) >> SAR

  qz[79: 64] = temp0[15:0] + temp2[15:0]
  qz[95: 80] = temp1[15:0] + temp3[15:0]
  qz1[79:64] = temp0[15:0] - temp2[15:0]
  qz1[95:80] = temp3[15:0] - temp1[15:0]
  {qu[31: 0], qu[63: 32], qu[95: 64], qu[127: 96]} = load128({as[31:4],4{0}})
  as = as - 16
```

1.8.10 EE.FFT.AMS.S16.ST.INCP

Instruction Word

10100	sel2[0]	qz1[2:1]	qm[0]	qv[2:0]	qz1[0]	qx[2:0]	qy[1:0]	qm[2:1]	as[3:0]	at[3:0]	111	qy[2]

Assembler Syntax

EE.FFT.AMS.S16.ST.INCP qv, qz1, at, as, qx, qy, qm, sel2

Description

该指令为 FFT 专用指令。对 16-bit 数据段进行加减、乘、加减、移位操作。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值与 at 寄存器拼接成的 16-byte 数据存储到内存。访存结束后，as 寄存器内数值增加 16。并用运算结果更新 at 寄存器内数值。

Operation

```
  temp0[15:0] = qx[111: 96] + qy[111: 96]
  temp1[15:0] = qx[127:112] - qy[127:112]

if sel2==0:
  temp2[15:0] = ((qx[111: 96] - qy[111: 96]) * qm[111: 96] - (qx[127:112] + qy
      [127:112]) * qm[127:112]) >> SAR[5:0]
  temp3[15:0] = ((qx[111: 96] - qy[111: 96]) * qm[127:112] + (qx[127:112] + qy
      [127:112]) * qm[111: 96]) >> SAR[5:0]
  {qv[ 95: 80] >> 1, qv[ 79: 64] >> 1, qv[ 63: 48] >> 1, qv[ 47: 32] >> 1, qv[ 31:
      16] >> 1, qv[ 15:  0] >> 1, at[31: 16] >> 1, at[15:0] >> 1} => store128({as
      [31:4],4{0}})
if sel2==1:
  temp2[15:0] = ((qx[127:112] + qy[127:112]) * qm[127:112] + (qx[111: 96] - qy[111:
      96]) * qm[111: 96]) >> SAR[5:0]
  temp3[15:0] = ((qx[127:112] + qy[127:112]) * qm[111: 96] - (qx[111: 96] - qy[111:
      96]) * qm[127:112]) >> SAR[5:0]
  {qv[ 95: 64], qv[ 63: 32], qv[ 31:  0], at[31:0]} => store128({as[31:4],4{0}})

  temp4[16:0] = temp1[15:0] + temp3[15:0]
  temp5[16:0] = temp0[15:0] + temp2[15:0]
  qz1[111: 96] = temp0[15:0] - temp2[15:0]
  qz1[127:112] = temp3[15:0] - temp1[15:0]
  at = {temp4[15:0], temp5[15:0]}
  as[31:0] = as[31:0] +16
```

1.8.11 EE.FFT.CMUL.S16.LD.XP

Instruction Word

110111	sel8[2:1]	qu[0]	qy[2:0]	sel8[0]	qz[2:0]	qx[1:0]	qu[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.FFT.CMUL.S16.LD.XP qu, as, ad, qz, qx, qy, sel8

Description

该指令运算部分与 EE.CMUL.S16 相似，都是实现了 16-bit 有符号复数乘法。区别在于调换了 qx 与 qy 寄存器顺序。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
if sel8 == 0:
  qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0] + qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
  qz[ 31: 16] = (qx[ 31: 16] * qy[ 15:  0] - qx[ 15:  0] * qy[ 31: 16]) >> SAR[5:0]
if sel8 == 1:
  qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0] - qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
  qz[ 31: 16] = (qx[ 31: 16] * qy[ 15:  0] + qx[ 15:  0] * qy[ 31: 16]) >> SAR[5:0]
if sel8 == 2:
  qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32] + qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
  qz[ 63: 48] = (qx[ 63: 48] * qy[ 47: 32] - qx[ 47: 32] * qy[ 63: 48]) >> SAR[5:0]
if sel8 == 3:
  qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32] - qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
  qz[ 63: 48] = (qx[ 63: 48] * qy[ 47: 32] + qx[ 47: 32] * qy[ 63: 48]) >> SAR[5:0]
if sel8 == 4:
  qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64] + qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
  qz[ 95: 80] = (qx[ 95: 80] * qy[ 79: 64] - qx[ 79: 64] * qy[ 95: 80]) >> SAR[5:0]
if sel8 == 5:
  qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64] - qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
  qz[ 95: 80] = (qx[ 95: 80] * qy[ 79: 64] + qx[ 79: 64] * qy[ 95: 80]) >> SAR[5:0]

  qu[127:0] = load128({as[31:4],4{0}})
  as[31:0] = as[31:0] + ad[31:0]
```

1.8.12 EE.FFT.CMUL.S16.ST.XP

Instruction Word

10101	sar4[1:0]	upd4[1]	sel8[0]	qy[2:0]	upd4[0]	qv[2:0]	qx[1:0]	sel8[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.FFT.CMUL.S16.ST.XP qx, qy, qv, as, ad, sel8, upd4, sar4

Description

该指令运算部分与 EE.CMUL.S16 相似，都是实现了 16-bit 有符号复数乘法。区别在于调换了 qx 与 qy 寄存器顺序。

根据立即数 upd4，将上述运算操作的计算结果与 qx、qv 寄存器内的部分数据段拼接成 16-byte，随后写入存储器空间。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
if sel8 == 6:
  temp[15:0] = (qx[111: 96] * qy[111: 96] + qx[127:112] * qy[127:112]) >> SAR[5:0]
  temp[31:16] = (qx[127:112] * qy[111: 96] - qx[111: 96] * qy[127:112]) >> SAR[5:0]
if sel8 == 7:
  temp[15:0] = (qx[111: 96] * qy[111: 96] - qx[127:112] * qy[127:112]) >> SAR[5:0]
  temp[31:16] = (qx[127:112] * qy[111: 96] + qx[111: 96] * qy[127:112]) >> SAR[5:0]

if upd4 == 0:  // normal radix 2
  {temp[31:0], qv[ 95:  0]} => store128({as[31:4],4{0}})
if upd4 == 1:  // radix2 last second stage
  {
       temp[31:0],
       qv[ 95: 64],
       qx[ 63: 48] >> sar4,
       qx[ 47: 32] >> sar4,
       qx[ 31: 16] >> sar4,
       qx[ 15:  0] >> sar4
  } => store128({as[31:4],4{0}})
if upd4 == 2:  // radix2 last stage
  {
       temp[31:0],
       qx[ 63: 48] >> sar4,
       qx[ 47: 32] >> sar4,
       qv[ 95: 64],
       qx[ 31: 16] >> sar4,
       qx[ 15:  0] >> sar4
  } => store128({as[31:4],4{0}})

  as[31:0] = as[31:0] + ad[31:0]
```

1.8.13 EE.FFT.R2BF.S16

Instruction Word

11	qa0[2:1]	1100	qa0[0]	qa1[2:0]		qy[2:1]	0	sel2[0]	qx[2:1]	qy[0]	qx[0]	0100

Assembler Syntax

EE.FFT.R2BF.S16 qa0, qa1, qx, qy, sel2

Description

对 qx、qy 寄存器内 16-byte 数值进行基-2 蝶形运算，数据位宽为 16-bit。一部分计算结果被写入 qa0 寄存器，另一部分计算结果被写入 qa1 寄存器。

Operation

```
if sel2==0:
  op_a[127:  0] = {qy[ 63:  0], qx[ 63:  0]}
  op_b[127:  0] = {qy[127: 64], qx[127: 64]}
if sel2==1:
  op_a[127:  0] = {qy[ 95: 64], qy[ 31:  0], qx[ 95: 64], qx[ 31:  0]}
  op_b[127:  0] = {qy[127: 96], qy[ 63: 32], qx[127: 96], qx[ 63: 32]}

  qa0[ 15:  0] = op_a[ 15:  0] + op_b[ 15:  0]
  qa0[ 31: 16] = op_a[ 31: 16] + op_b[ 31: 16]
  qa0[ 47: 32] = op_a[ 47: 32] + op_b[ 47: 32]
  qa0[ 63: 48] = op_a[ 63: 48] + op_b[ 63: 48]
  qa0[ 79: 64] = op_a[ 15:  0] - op_b[ 15:  0]
  qa0[ 95: 80] = op_a[ 31: 16] - op_b[ 31: 16]
  qa0[111: 96] = op_a[ 47: 32] - op_b[ 47: 32]
  qa0[127:112] = op_a[ 63: 48] - op_b[ 63: 48]

  qa1[ 15:  0] = op_a[ 79: 64] + op_b[ 79: 64]
  qa1[ 31: 16] = op_a[ 95: 80] + op_b[ 95: 80]
  qa1[ 47: 32] = op_a[111: 96] + op_b[111: 96]
  qa1[ 63: 48] = op_a[127:112] + op_b[127:112]
  qa1[ 79: 64] = op_a[ 79: 64] - op_b[ 79: 64]
  qa1[ 95: 80] = op_a[ 95: 80] - op_b[ 95: 80]
  qa1[111: 96] = op_a[111: 96] - op_b[111: 96]
  qa1[127:112] = op_a[127:112] - op_b[127:112]
```

1.8.14 EE.FFT.R2BF.S16.ST.INCP

Instruction Word

11101000	sar4[0]	qy[2:0]	1	qa0[2:0]	qx[1:0]	0	sar4[1]	0100	as[3:0]	111	qx[2]

Assembler Syntax

EE.FFT.R2BF.S16.ST.INCP qa0, qx, qy, as, 0..3

Description

对 qx、qy 寄存器内 16-byte 数值进行基-2 蝶形运算，数据位宽为 16-bit。一部分计算结果被写入 qa0 寄存器，另一部分执行算术移位后写入 as 地址所指示的内存中。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa0[ 15:  0] = qx[ 15:  0] - qy[ 15:  0]
qa0[ 31: 16] = qx[ 31: 16] - qy[ 31: 16]
qa0[ 47: 32] = qx[ 47: 32] - qy[ 47: 32]
qa0[ 63: 48] = qx[ 63: 48] - qy[ 63: 48]
qa0[ 79: 64] = qx[ 79: 64] - qy[ 79: 64]
qa0[ 95: 80] = qx[ 95: 80] - qy[ 95: 80]
qa0[111: 96] = qx[111: 96] - qy[111: 96]
qa0[127:112] = qx[127:112] - qy[127:112]

{
  (qx[127:112] + qy[127:112]) >> sar4,
  (qx[111: 96] + qy[111: 96]) >> sar4,
  (qx[ 95: 80] + qy[ 95: 80]) >> sar4,
  (qx[ 79: 64] + qy[ 79: 64]) >> sar4,
  (qx[ 63: 48] + qy[ 63: 48]) >> sar4,
  (qx[ 47: 32] + qy[ 47: 32]) >> sar4,
  (qx[ 31: 16] + qy[ 31: 16]) >> sar4,
  (qx[ 15:  0] + qy[ 15:  0]) >> sar4
} => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.15 EE.FFT.VST.R32.DECP

Instruction Word

11	qv[2:1]	1101	qv[0]	0110	sar2[0]	11	as[3:0]	0100

Assembler Syntax

EE.FFT.VST.R32.DECP qv, as, sar2

Description

该指令为 FFT 专用指令。对 qv 寄存器按照 16-bit 分成的 8 个数据段进行算术右移，右移数值大小为 0 或 1，取决于立即数 sar2。最后根据 word 大端序，将上述移位得到的 8 个 16-bit 数据写入 as 寄存器指示的内存。访存结束后，as 寄存器内数值减少 16。

Operation

```
{
  qv[ 31: 16] >> sar2,
  qv[ 15:  0] >> sar2,
  qv[ 63: 48] >> sar2,
  qv[ 47: 32] >> sar2,
  qv[ 95: 80] >> sar2,
  qv[ 79: 64] >> sar2,
  qv[127:112] >> sar2,
  qv[111: 96] >> sar2
} => store128({as[31:4],4{0}})
as = as - 16
```

1.8.16 EE.GET_GPIO_IN

Instruction Word

0110010100001000	au[3:0]	0100

Assembler Syntax

EE.GET_GPIO_IN au

Description

该指令为 CPU GPIO 专用指令。将 GPIO_IN 数据内容赋值给 au 寄存器的低 8-bit。

Operation

```
au = {24'h0, GPIO_IN[7:0]}
```

1.8.17 EE.LD.128.USAR.IP

Instruction Word

1	imm16[7]	qu[2:1]	0001	qu[0]	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.LD.128.USAR.IP qu, as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。同时将 as 地址寄存器中低 4-bit 数值存入到特殊寄存器 SAR_BYTE 中。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
qu[127:0] = load128({as[31:4],4{0}})
SAR_BYTE = as[3:0]
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.18 EE.LD.128.USAR.XP

Instruction Word

10	qu[2:1]	1101	qu[0]	000	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.LD.128.USAR.XP qu, as, ad

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。同时将 as 地址寄存器中低 4-bit 数值存入到特殊寄存器 SAR_BYTE 中。访存结束后，as 寄存器内数值增加 ad 寄存器数值。

Operation

```
qu[127:0] = load128({as[31:4],4{0}})
SAR_BYTE = as[3:0]
as[31:0] = as[31:0] + ad[31:0]
```

1.8.19 EE.LD.ACCX.IP

Instruction Word

0	imm8[7]	0011100	imm8[6:0]	as[3:0]	0100

Assembler Syntax

EE.LD.ACCX.IP as, -1024..1016

Description

将寄存器 as 内访存地址低 3 比特强制为 0，随后从内存中载入 64-bit 数据，将数据的低 40-bit 存入特殊寄存器 ACCX。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 3 位的值。

Operation

```
ACCX[39:0] = load64({as[31:3],3{0}})
as[31:0] = as[31:0] + {21{imm8[7]},imm8[7:0],3{0}}
```

1.8.20 EE.LD.QACC_H.H.32.IP

Instruction Word

0	imm4[7]	0111100	imm4[6:0]	as[3:0]	0100

Assembler Syntax

EE.LD.QACC_H.H.32.IP as, -512..508

Description

将寄存器 as 内访存地址低 2 比特强制为 0，随后从内存中载入 32-bit 数据，并存入特殊寄存器 QACC_H[159:128]。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 2 位的值。

Operation

```
QACC_H[159:128] = load32({as[31:2],2{0}})
as[31:0] = as[31:0] + {22{imm4[7]},imm4[7:0],2{0}}
```

1.8.21 EE.LD.QACC_H.L.128.IP

Instruction Word

0	imm16[7]	0001100	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.LD.QACC_H.L.128.IP as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入特殊寄存器 QACC_H[127:0]。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
QACC_H[127:  0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.22 EE.LD.QACC_L.H.32.IP

Instruction Word

0	imm4[7]	0101100	imm4[6:0]	as[3:0]	0100

Assembler Syntax

EE.LD.QACC_L.H.32.IP as, -512..508

Description

将寄存器 as 内访存地址低 2 比特强制为 0，随后从内存中载入 32-bit 数据，并存入特殊寄存器 QACC_L[159:128]。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 2 位的值。

Operation

```
QACC_L[159:128] = load32({as[31:2],2{0}})
as[31:0] = as[31:0] + {22{imm4[7]},imm4[7:0],2{0}}
```

1.8.23 EE.LD.QACC_L.L.128.IP

Instruction Word

0	imm16[7]	0000000	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.LD.QACC_L.L.128.IP as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入特殊寄存器 QACC_L[127:0]。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
QACC_L[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.24 EE.LD.UA_STATE.IP

Instruction Word

0	imm16[7]	0100000	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.LD.UA_STATE.IP as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，存入特殊寄存器 UA_STATE。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
UA_STATE[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.25 EE.LDF.128.IP

Instruction Word

10000	fu3[3:1]	fu0[3:0]	fu3[0]	fu2[3:1]	fu1[3:0]	imm16f[3:0]	as[3:0]	111	fu2[0]

Assembler Syntax

EE.LDF.128.IP fu3, fu2, fu1, fu0, as, -128..112

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，按照数据低位到高位顺序依次存入浮点寄存器 fu0、fu1、fu2、fu3 中。访存结束后，as 寄存器内数值增加指令编码段中 4-bit 符号位扩展常数左移 4 位的值。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
fu3 = dataIn[127: 96]
fu2 = dataIn[ 95: 64]
fu1 = dataIn[ 63: 32]
fu0 = dataIn[ 31:  0]
as[31:0] = as[31:0] + {24{imm16f[3]},imm16f[3:0],4{0}}
```

1.8.26 EE.LDF.128.XP

Instruction Word

10001	fu3[3:1]	fu0[3:0]	fu3[0]	fu2[3:1]	fu1[3:0]	ad[3:0]	as[3:0]	111	fu2[0]

Assembler Syntax

EE.LDF.128.XP fu3, fu2, fu1, fu0, as, ad

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，按照数据低位到高位顺序依次存入浮点寄存器 fu0、fu1、fu2、fu3 中。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
fu3 = dataIn[127: 96]
fu2 = dataIn[ 95: 64]
fu1 = dataIn[ 63: 32]
fu0 = dataIn[ 31:  0]
as[31:0] = as[31:0] + ad[31:0]
```

1.8.27 EE.LDF.64.IP

Instruction Word

111000	imm8[7:6]	fu0[3:0]	imm8[5:2]	fu1[3:0]	010	imm8[0]	as[3:0]	111	imm8[1]

Assembler Syntax

EE.LDF.64.IP fu1, fu0, as, -1024..1016

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后从内存中载入 64-bit 数据，按照数据低位到高位顺序依次存入浮点寄存器 fu0、fu1 中。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 3 位的值。

Operation

```
dataIn[63:0] = load64({as[31:3],3{0}})
fu1 = dataIn[63:32]
fu0 = dataIn[31: 0]
as[31:0] = as[31:0] + {21{imm8[7]},imm8[7:0],3{0}}
```

1.8.28 EE.LDF.64.XP

Instruction Word

fu0[3:0]	0110	fu1[3:0]	ad[3:0]	as[3:0]	0000

Assembler Syntax

EE.LDF.64.XP fu1, fu0, as, ad

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后从内存中载入 64-bit 数据，按照数据低位到高位顺序依次存入浮点寄存器 fu0、fu1 中。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
dataIn[63:0] = load64({as[31:3],3{0}})
fu1 = dataIn[63:32]
fu0 = dataIn[31: 0]
as[31:0] = as[31:0] + ad[31:0]
```

1.8.29 EE.LDQA.S16.128.IP

Instruction Word

0	imm16[7]	0000010	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.LDQA.S16.128.IP as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将内存中载入的 16-byte 数据拆成 8 个 16-bit，将每个 16-bit 符号位扩展成 40-bit，随后将 8 个 40-bit 数据分别存入 160-bit 特殊寄存器 QACC_L 和 QACC_H 中。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
QACC_L[ 39:  0] = {24{dataIn[15]}, dataIn[ 15:  0]}
QACC_L[ 79: 40] = {24{dataIn[31]}, dataIn[ 31: 16]}
...
QACC_H[ 39:  0] = {24{dataIn[79]}, dataIn[ 79: 64]}
QACC_H[ 79: 40] = {24{dataIn[95]}, dataIn[ 95: 80]}

as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.30 EE.LDQA.S16.128.XP

Instruction Word

011111100100	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.LDQA.S16.128.XP as, ad

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将内存中载入的 16-byte 数据拆成 8 个 16-bit，将每个 16-bit 符号位扩展成 40-bit，随后将 8 个 40-bit 数据分别存入 160-bit 特殊寄存器 QACC_L 和 QACC_H 中。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
QACC_L[ 39:  0] = {24{dataIn[15]}, dataIn[ 15:  0]}
QACC_L[ 79: 40] = {24{dataIn[31]}, dataIn[ 31: 16]}
...
QACC_H[ 39:  0] = {24{dataIn[79]}, dataIn[ 79: 64]}
QACC_H[ 79: 40] = {24{dataIn[95]}, dataIn[ 95: 80]}
as[31:0] = as[31:0] + ad[31:0]
```

1.8.31 EE.LDQA.S8.128.IP

Instruction Word

0	imm16[7]	0100010	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.LDQA.S8.128.IP as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将内存中载入的 16-byte 数据拆成 16 个 8-bit，将每个 8-bit 符号位扩展成 20-bit，随后将 16 个 20-bit 数据分别存入 160-bit 特殊寄存器 QACC_L 和 QACC_H 中。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
QACC_L[19:0] = {12{dataIn[7]}, dataIn[7:0]}
QACC_L[39:20] = {12{dataIn[15]}, dataIn[15:8]}
...
QACC_H[159:140] = {12{dataIn[127]}, dataIn[127:120]}
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.32 EE.LDQA.S8.128.XP

Instruction Word

011100010100	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.LDQA.S8.128.XP as, ad

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将内存中载入的 16-byte 数据拆成 16 个 8-bit，将每个 8-bit 符号位扩展成 20-bit，随后将 16 个 20-bit 数据分别存入 160-bit 特殊寄存器 QACC_L 和 QACC_H 中。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
QACC_L[19:0] = {12{dataIn[7]}, dataIn[7:0]}
QACC_L[39:20] = {12{dataIn[15]}, dataIn[15:8]}
...
QACC_H[159:140] = {12{dataIn[127]}, dataIn[127:120]}
as[31:0] = as[31:0] + ad[31:0]
```

1.8.33 EE.LDQA.U16.128.IP

Instruction Word

0	imm16[7]	0001010	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.LDQA.U16.128.IP as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将内存中载入的 16-byte 数据拆成 8 个 16-bit，将每个 16-bit 高位补 0 扩展成 40-bit，随后将 8 个 40-bit 数据分别存入 160-bit 特殊寄存器 QACC_L 和 QACC_H 中。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
QACC_L[ 39:  0] = {24{0}, dataIn[ 15:  0]}
QACC_L[ 79: 40] = {24{0}, dataIn[ 31: 16]}
QACC_L[119: 80] = {24{0}, dataIn[ 47: 32]}
QACC_L[159:120] = {24{0}, dataIn[ 63: 48]}
QACC_H[ 39:  0] = {24{0}, dataIn[ 79: 64]}
QACC_H[ 79: 40] = {24{0}, dataIn[ 95: 80]}
QACC_H[119: 80] = {24{0}, dataIn[111: 96]}
QACC_H[159:120] = {24{0}, dataIn[127:112]}
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.34 EE.LDQA.U16.128.XP

Instruction Word

011110100100	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.LDQA.U16.128.XP as, ad

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将内存中载入的 16-byte 数据拆成 8 个 16-bit，将每个 16-bit 高位补 0 扩展成 40-bit，随后将 8 个 40-bit 数据分别存入 160-bit 特殊寄存器 QACC_L 和 QACC_H 中。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
QACC_L[ 39:  0] = {24{0}, dataIn[ 15:  0]}
...
QACC_H[159:120] = {24{0}, dataIn[127:112]}
as[31:0] = as[31:0] + ad[31:0]
```

1.8.35 EE.LDQA.U8.128.IP

Instruction Word

0	imm16[7]	0101010	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.LDQA.U8.128.IP as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将内存中载入的 16-byte 数据拆成 16 个 8-bit，将每个 8-bit 高位补 0 扩展成 20-bit，随后将 16 个 20-bit 数据分别存入 160-bit 特殊寄存器 QACC_L 和 QACC_H 中。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
QACC_L[ 19:  0] = {12{0}, dataIn[  7:  0]}
QACC_L[ 39: 20] = {12{0}, dataIn[ 15:  8]}
QACC_L[ 59: 40] = {12{0}, dataIn[ 23: 16]}
...
QACC_L[159:140] = {12{0}, dataIn[ 63: 56]}
QACC_H[ 19:  0] = {12{0}, dataIn[ 71: 64]}
QACC_H[ 39: 20] = {12{0}, dataIn[ 79: 72]}
QACC_H[ 59: 40] = {12{0}, dataIn[ 87: 80]}
...
QACC_H[159:140] = {12{0}, dataIn[127:120]}
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.36 EE.LDQA.U8.128.XP

Instruction Word

011100000100	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.LDQA.U8.128.XP as, ad

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将内存中载入的 16-byte 数据拆成 16 个 8-bit，将每个 8-bit 高位补 0 扩展成 20-bit，随后将 16 个 20-bit 数据分别存入 160-bit 特殊寄存器 QACC_L 和 QACC_H 中。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
QACC_L[ 19:  0] = {12{0}, dataIn[  7:  0]}
...
QACC_L[159:140] = {12{0}, dataIn[ 63: 56]}
QACC_H[ 19:  0] = {12{0}, dataIn[ 71: 64]}
...
QACC_H[159:140] = {12{0}, dataIn[127:120]}
as[31:0] = as[31:0] + ad[31:0]
```

1.8.37 EE.LDXQ.32

Instruction Word

1110000	sel4[1]	sel8[0]	111	sel4[0]	qu[2:0]	qs[1:0]	sel8[2:1]	1101	as[3:0]	111	qs[2]

Assembler Syntax

EE.LDXQ.32 qu, qs, as, 0..3, 0..7

Description

根据立即数 sel8 从 qs 的 8 个 16-bit 数据段中选择一个作为加数。用地址寄存器 as 的数值与前面的加数左移 2 位后相加得到的结果作为访问地址，且进行 32-bit 对齐（将访存地址的低 2 比特置 0）。根据立即数 sel4 的数值，将载入的 32-bit 数据存入 qu 寄存器某个 32-bit 数据段中。

Operation

```
  vaddr0[31:0] = as[31:0] + qs[ 15:  0] * 4
  vaddr1[31:0] = as[31:0] + qs[ 31: 16] * 4
  vaddr2[31:0] = as[31:0] + qs[ 47: 32] * 4
  vaddr3[31:0] = as[31:0] + qs[ 63: 47] * 4
  vaddr4[31:0] = as[31:0] + qs[ 79: 64] * 4
  vaddr5[31:0] = as[31:0] + qs[ 95: 80] * 4
  vaddr6[31:0] = as[31:0] + qs[111: 96] * 4
  vaddr7[31:0] = as[31:0] + qs[127:112] * 4

if sel8 == 0:
  dataIn[31:0] = load32({vaddr0[31:2],2{0}})
if sel8 == 1:
  dataIn[31:0] = load32({vaddr1[31:2],2{0}})
if sel8 == 2:
  dataIn[31:0] = load32({vaddr2[31:2],2{0}})
if sel8 == 3:
  dataIn[31:0] = load32({vaddr3[31:2],2{0}})
if sel8 == 4:
  dataIn[31:0] = load32({vaddr4[31:2],2{0}})
if sel8 == 5:
  dataIn[31:0] = load32({vaddr5[31:2],2{0}})
if sel8 == 6:
  dataIn[31:0] = load32({vaddr6[31:2],2{0}})
if sel8 == 7:
  dataIn[31:0] = load32({vaddr7[31:2],2{0}})

  qu[32*sel4+31:32*sel4] = dataIn[31:0]
```

1.8.38 EE.MOV.S16.QACC

Instruction Word

11	qs[2:1]	1101	qs[0]	111111100100100

Assembler Syntax

EE.MOV.S16.QACC qs

Description

将 qs 寄存器内 8 个 16-bit 数据段按符号位扩展成 40-bit。随后将得到的 8 个 40-bit 数据写入特殊寄存器 QACC_H 和 QACC_L 中。

Operation

```
QACC_L[ 39:  0] = {24{qs[15]}, qs[ 15:  0]}
QACC_L[ 79: 40] = {24{qs[31]}, qs[ 31: 16]}
QACC_L[119: 80] = {24{qs[47]}, qs[ 47: 32]}
QACC_L[159:120] = {24{qs[63]}, qs[ 63: 48]}
QACC_H[ 39:  0] = {24{qs[79]}, qs[ 79: 64]}
QACC_H[ 79: 40] = {24{qs[95]}, qs[ 95: 80]}
QACC_H[119: 80] = {24{qs[79]}, qs[111: 96]}
QACC_H[159:120] = {24{qs[95]}, qs[127:112]}
```

1.8.39 EE.MOV.S8.QACC

Instruction Word

11	qs[2:1]	1101	qs[0]	111111100110100

Assembler Syntax

EE.MOV.S8.QACC qs

Description

将 qs 寄存器内 16 个 8-bit 数据段按符号位扩展成 20-bit。随后将得到的 16 个 20-bit 数据写入特殊寄存器 QACC_H 和 QACC_L 中。

Operation

```
QACC_L[ 19:  0] = {12{qs[7]}, qs[  7:  0]}
QACC_L[ 39: 20] = {12{qs[15]}, qs[ 15:  8]}
...
QACC_H[159:140] = {12{qs[127]}, qs[127:120]}
```

1.8.40 EE.MOV.U16.QACC

Instruction Word

11	qs[2:1]	1101	qs[0]	111111101100100

Assembler Syntax

EE.MOV.U16.QACC qs

Description

将 qs 寄存器内 8 个 16-bit 数据段高位补 0 扩展成 40-bit。随后将得到的 8 个 40-bit 数据写入特殊寄存器 QACC_H 和 QACC_L 中。

Operation

```
QACC_L[ 39:  0] = {24{0}, qs[ 15:  0]}
QACC_L[ 79: 40] = {24{0}, qs[ 31: 16]}
QACC_L[119: 80] = {24{0}, qs[ 47: 32]}
QACC_L[159:120] = {24{0}, qs[ 63: 48]}
QACC_H[ 39:  0] = {24{0}, qs[ 79: 64]}
QACC_H[ 79: 40] = {24{0}, qs[ 95: 80]}
QACC_H[119: 80] = {24{0}, qs[111: 96]}
QACC_H[159:120] = {24{0}, qs[127:112]}
```

1.8.41 EE.MOV.U8.QACC

Instruction Word

11	qs[2:1]	1101	qs[0]	111111101110100

Assembler Syntax

EE.MOV.U8.QACC qs

Description

将 qs 寄存器内 16 个 8-bit 数据段高位补 0 扩展成 20-bit。随后将得到的 16 个 20-bit 数据写入特殊寄存器 QACC_H 和 QACC_L 中。

Operation

```
QACC_L[ 19:  0] = {12{0}, qs[  7:  0]}
QACC_L[ 39: 20] = {12{0}, qs[ 15:  8]}
QACC_L[ 59: 40] = {12{0}, qs[ 23: 16]}
...
QACC_L[159:140] = {12{0}, qs[ 63: 56]}
QACC_H[ 19:  0] = {12{0}, qs[ 71: 64]}
QACC_H[ 39: 20] = {12{0}, qs[ 79: 72]}
QACC_H[ 59: 40] = {12{0}, qs[ 87: 80]}
...
QACC_H[159:140] = {12{0}, qs[127:120]}
```

1.8.42 EE.MOVI.32.A

Instruction Word

11	qs[2:1]	1101	qs[0]	111	sel4[1:0]	01	au[3:0]	0100

Assembler Syntax

EE.MOVI.32.A qs, au, 0..3

Description

根据立即数 sel4 从 qs 寄存器中选择 1 个 32-bit 数据段，赋值给 au 寄存器。

Operation

```
if sel4 == 0:
  au = qs[ 31:  0]
if sel4 == 1:
  au = qs[ 63: 32]
if sel4 == 2:
  au = qs[ 95: 64]
if sel4 == 3:
  au = qs[127: 96]
```

1.8.43 EE.MOVI.32.Q

Instruction Word

11	qu[2:1]	1101	qu[0]	011	sel4[1:0]	10	as[3:0]	0100

Assembler Syntax

EE.MOVI.32.Q qu, as, 0..3

Description

根据立即数 sel4，将 as 寄存器中数值赋值给 qu 寄存器中的某一个 32-bit 数据段。

Operation

```
if sel4 == 0:
  qu[ 31:  0] = as
if sel4 == 1:
  qu[ 63: 32] = as
if sel4 == 2:
  qu[ 95: 64] = as
if sel4 == 3:
  qu[127: 96] = as
```

1.8.44 EE.NOTQ

Instruction Word

11	qa[2:1]	1101	qa[0]	1111111	qx[2:1]	0	qx[0]	0100

Assembler Syntax

EE.NOTQ qa, qx

Description

该指令将 qx 寄存器按位取反的结果写入 qa 寄存器。

Operation

```
qa = ~qx
```

1.8.45 EE.ORQ

Instruction Word

11	qa[2:1]	1101	qa[0]	111	qy[2:1]	00	qx[2:1]	qy[0]	qx[0]	0100

Assembler Syntax

EE.ORQ qa, qx, qy

Description

该指令对 qx 和 qy 寄存器进行按位或操作，并将逻辑运算的结果写入 qa 寄存器。

Operation

```
qa = qx | qy
```

1.8.46 EE.SET_BIT_GPIO_OUT

Instruction Word

011101010100	imm256[7:0]	0100

Assembler Syntax

EE.SET_BIT_GPIO_OUT 0..255

Description

该指令为 CPU GPIO 专用指令。功能为置位 GPIO_OUT 某些比特。赋值内容取决于 8-bit 立即数 imm256。

Operation

```
GPIO_OUT[7:0] =  (GPIO_OUT[7:0] | imm256[7:0])
```

1.8.47 EE.SLCI.2Q

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	0110	sar16[3:0]	0100

Assembler Syntax

EE.SLCI.2Q qs1, qs0, 0..15

Description

该指令对寄存器 qs0、qs1 拼接成的 32-byte 数据执行左移操作，低位补零。左移结果的 32-byte 数据的高 16-byte 更新到 qs1 寄存器，左移结果的低 16-byte 更新到 qs0 寄存器。左移数值为 sar16 加 1 结果乘以 8。

Operation

```
{qs1[127:  0], qs0[127:  0]} = {qs1[127:  0], qs0[127:  0]} << ((sar16[3:0]+1)*8)
```

1.8.48 EE.SLCXXP.2Q

Instruction Word

10	qs1[2:1]	0110	qs1[0]	qs0[2:0]	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.SLCXXP.2Q qs1, qs0, as, ad

Description

该指令对寄存器 qs0、qs1 拼接成的 32-byte 数据执行左移操作，低位补零。左移结果的 32-byte 数据的高 16-byte 更新到 qs1 寄存器，左移结果的低 16-byte 更新到 qs0 寄存器。左移数值为 as 寄存器低 4-bit 数值加 1 结果乘以 8。上述操作结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
{qs1[127:  0], qs0[127:  0]} = {qs1[127:  0], qs0[127:  0]} << ((as[3:0]+1)*8)
as[31:0] = as[31:0] + ad[31:0]
```

1.8.49 EE.SRC.Q

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	00110	qa[2:0]	0100

Assembler Syntax

EE.SRC.Q qa, qs0, qs1

Description

该指令对保存着两个连续对齐地址载入数据的寄存器 qs0、qs1 拼接成的 32-byte 数据执行算术右移操作，从而得到非对齐的 16-byte 数据，该数值被写入 qa 寄存器。右移数值为 SAR_BYTE 乘以 8。

Operation

```
qa[127:  0]  = {qs1[127:  0], qs0[127:  0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.50 EE.SRC.Q.LD.IP

Instruction Word

111000	imm16[7:6]	imm16[2]	qs1[2:0]	imm16[5]	qu[2:0]	qs0[1:0]	imm16[4:3]	00	imm16[1:0]	as[3:0]	111	qs0[2]

Assembler Syntax

EE.SRC.Q.LD.IP qu, as, -2048..2032, qs0, qs1

Description

该指令对保存着两个连续对齐地址载入数据的寄存器 qs0、qs1 拼接成的 32-byte 数据执行算术右移操作，从而得到非对齐的 16-byte 数据，该数值被写入 qs0 寄存器。右移数值为 SAR_BYTE 乘以 8。

执上述操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.51 EE.SRC.Q.LD.XP

Instruction Word

111010000	qs1[2:0]	0	qu[2:0]	qs0[1:0]	00	ad[3:0]	as[3:0]	111	qs0[2]

Assembler Syntax

EE.SRC.Q.LD.XP qu, as, ad, qs0, qs1

Description

该指令对保存着两个连续对齐地址载入数据的寄存器 qs0、qs1 拼接成的 32-byte 数据执行算术右移操作，从而得到非对齐的 16-byte 数据，该数值被写入 qs0 寄存器。右移数值为 SAR_BYTE 乘以 8。

执上述操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
qs0[127:0] = {qs1[127: 0], qs0[127: 0]} >> {SAR_BYTE[3:0] << 3}
qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.52 EE.SRC.Q.QUP

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	01110	qa[2:0]	0100

Assembler Syntax

EE.SRC.Q.QUP qa, qs0, qs1

Description

该指令对保存着两个连续对齐地址载入数据的寄存器 qs0、qs1 拼接成的 32-byte 数据执行算术右移操作，从而得到非对齐的 16-byte 数据，该数值被写入 qa 寄存器。右移数值为 SAR_BYTE 乘以 8。同时将 qs1 寄存器的数值更新到 qs0 寄存器中。

Operation

```
qa[127:  0] = {qs1[127:  0], qs0[127:  0]} >> {SAR_BYTE[3:0] << 3}
qs0 = qs1
```

1.8.53 EE.SRCI.2Q

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	1010	sar16[3:0]	0100

Assembler Syntax

EE.SRCI.2Q qs1, qs0, sar16

Description

该指令对寄存器 qs0、qs1 拼接成的 32-byte 数据执行逻辑右移操作，高位补零。右移结果的 32-byte 数据的高 16-byte 更新到 qs1 寄存器，右移结果的低 16-byte 更新到 qs0 寄存器。右移数值为 sar16 加 1 结果乘以 8。

Operation

```
{qs1[127:  0], qs0[127:  0]} = {qs1[127:  0], qs0[127:  0]} >> ((sar16+1)*8)
qs1[127:127-8*sar16] = 0
```

1.8.54 EE.SRCMB.S16.QACC

Instruction Word

11	qu[2:1]	1101	qu[0]	1110	010	as[3:0]	0100

Assembler Syntax

EE.SRCMB.S16.QACC qu, as, 0

Description

从特殊寄存器 QACC_H 和 QACC_L 中提取出 8 个 40-bit 数据段，分别进行右移操作。将右移后结果重新写回 QACC_H 和 QACC_L 的同时对结果进行 16-bit 有符号数取饱和，并将取饱和后得到的 8 个 16-bit 数据写入 qu 寄存器。

Operation

```
temp0[39:0] = QACC_L[ 39:  0]
...
temp3[39:0] = QACC_L[159:120]
temp4[39:0] = QACC_H[ 39:  0]
...
temp7[39:0] = QACC_H[159:120]

temp_shf0[39:0] = temp0[39:0] >> as[5:0]
temp_shf1[39:0] = temp1[39:0] >> as[5:0]
...
temp_shf7[39:0] = temp7[39:0] >> as[5:0]

QACC_L[ 39:  0] = temp_shf0[39:0]
...
QACC_L[159:120] = temp_shf3[39:0]
QACC_H[ 39:  0] = temp_shf4[39:0]
...
QACC_H[159:120] = temp_shf7[39:0]

qu[ 15:  0] = min(max(temp_shf0[39:0], -2^{15}), 2^{15}-1)
...
qu[ 63: 48] = min(max(temp_shf3[39:0], -2^{15}), 2^{15}-1)
qu[ 79: 64] = min(max(temp_shf4[39:0], -2^{15}), 2^{15}-1)
...
qu[127:112] = min(max(temp_shf7[39:0], -2^{15}), 2^{15}-1)
```

1.8.55 EE.SRCMB.S8.QACC

Instruction Word

11	qu[2:1]	1101	qu[0]	1111	010	as[3:0]	0100

Assembler Syntax

EE.SRCMB.S8.QACC qu, as, 0

Description

从特殊寄存器 QACC_H 和 QACC_L 中提取出 16 个 20-bit 数据段，分别进行右移操作。将右移后结果重新写回 QACC_H 和 QACC_L 的同时对结果进行 8-bit 有符号数取饱和，并将取饱和后得到的 16 个 8-bit 数据写入 qu 寄存器。

Operation

```
temp0[19:0] = QACC_L[ 19:  0]
temp1[19:0] = QACC_L[ 39: 20]
...
temp7[19:0] = QACC_L[159:140]
temp8[19:0] = QACC_H[ 19:  0]
temp9[19:0] = QACC_H[ 39: 20]
...
temp15[19:0] = QACC_H[159:140]

temp_shf0[19:0] = temp0[19:0] >> as[4:0]
temp_shf1[19:0] = temp1[19:0] >> as[4:0]
...
temp_shf15[19:0] = temp15[19:0] >> as[4:0]

QACC_L[ 19:  0] = temp_shf0[19:0]
...
QACC_L[159:140] = temp_shf7[19:0]
QACC_H[ 19:  0] = temp_shf8[19:0]
...
QACC_H[159:140] = temp_shf15[19:0]

qu[  7:  0] = min(max(temp_shf0[19:0], -2^{7}), 2^{7}-1)
qu[ 15:  8] = min(max(temp_shf1[19:0], -2^{7}), 2^{7}-1)
...
qu[ 63: 56] = min(max(temp_shf7[19:0], -2^{7}), 2^{7}-1)
qu[ 71: 64] = min(max(temp_shf8[19:0], -2^{7}), 2^{7}-1)
qu[ 79: 72] = min(max(temp_shf9[19:0], -2^{7}), 2^{7}-1)
...
qu[127:120] = min(max(temp_shf15[19:0], -2^{7}), 2^{7}-1)
```

1.8.56 EE.SRCQ.128.ST.INCP

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	1110	as[3:0]	0100

Assembler Syntax

EE.SRCQ.128.ST.INCP qs0, qs1, as

Description

该指令对寄存器 qs0、qs1 拼接成的 32-byte 数据执行算术右移操作。随后将右移结果的低 16-byte 存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
{qs1[127:  0], qs0[127:  0]} >> {SAR_BYTE[3:0] << 3}  => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.57 EE.SRCXXP.2Q

Instruction Word

11	qs1[2:1]	0110	qs1[0]	qs0[2:0]	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.SRCXXP.2Q qs1, qs0, as, ad

Description

该指令对寄存器 qs0、qs1 拼接成的 32-byte 数据执行逻辑右移操作，高位补零。右移结果的 32-byte 数据的高 16-byte 更新到 qs1 寄存器，右移结果的低 16-byte 更新到 qs0 寄存器。右移数值为 as 寄存器低 4-bit 数值加 1 结果乘以 8。上述操作结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
{qs1[127:  0], qs0[127:  0]} = {qs1[127:  0], qs0[127:  0]} >> ((as[3:0]+1)*8)
qs1[127:127-8*as[3:  0]] = 0
as[31:0] = as[31:0] + ad[31:0]
```

1.8.58 EE.SRS.ACCX

Instruction Word

011111100	001	au[3:0]	as[3:0]	0100

Assembler Syntax

EE.SRS.ACCX au, as, 0

Description

对特殊寄存器 ACCX 进行右移操作。将右移后结果重新写回 ACCX 的同时对该结果进行 32-bit 有符号数取饱和，并将取饱和后得到的 32-bit 数据写入 au 寄存器。

Operation

```
temp_shf[39:0] = ACCX[39:0] >> as[5:0]
ACCX = temp_shf[39:0]
au = min(max(temp_shf[39:0], -2^{31}), 2^{31}-1)
```

1.8.59 EE.ST.ACCX.IP

Instruction Word

0	imm8[7]	0000100	imm8[6:0]	as[3:0]	0100

Assembler Syntax

EE.ST.ACCX.IP as, -512..508

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后将特殊寄存器 ACCX 高位补 0 扩展成 64-bit 存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 3 位的值。

Operation

```
{24{0},ACCX[39:0]} => store64({as[31:3],3{0}})
as += imm8
```

1.8.60 EE.ST.QACC_H.H.32.IP

Instruction Word

0	imm4[7]	0100100	imm4[6:0]	as[3:0]	0100

Assembler Syntax

EE.ST.QACC_H.H.32.IP as, -512..508

Description

将寄存器 as 内访存地址低 2 比特强制为 0，然后将特殊寄存器 QACC_H 高 32-bit 数据存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 2 位的值。

Operation

```
QACC_H[159:128] => store32({as[31:2],2{0}})
as[31:0] = as[31:0] + {23{imm4[7]},imm4[7:0],2{0}}
```

1.8.61 EE.ST.QACC_H.L.128.IP

Instruction Word

0	imm16[7]	0011010	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.ST.QACC_H.L.128.IP as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将特殊寄存器 QACC_H 低 16-byte 数据存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
QACC_H[127:  0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + {23{imm4[7]},imm4[7:0],2{0}}
```

1.8.62 EE.ST.QACC_L.H.32.IP

Instruction Word

0	imm4[7]	0111010	imm4[6:0]	as[3:0]	0100

Assembler Syntax

EE.ST.QACC_L.H.32.IP as, -512..508

Description

将寄存器 as 内访存地址低 2 比特强制为 0，然后将特殊寄存器 QACC_L 高 32-bit 数据存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 2 位的值。

Operation

```
QACC_L[159:128] => store32({as[31:2],2{0}})
as[31:0] = as[31:0] + {23{imm4[7]},imm4[7:0],2{0}}
```

1.8.63 EE.ST.QACC_L.L.128.IP

Instruction Word

0	imm16[7]	0011000	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.ST.QACC_L.L.128.IP as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将特殊寄存器 QACC_L 低 16-byte 数据存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
QACC_L[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + {23{imm4[7]},imm4[7:0],2{0}}
```

1.8.64 EE.ST.UA_STATE.IP

Instruction Word

0	imm16[7]	0111000	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.ST.UA_STATE.IP as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将特殊寄存器 UA_STATE 的 16-byte 数据存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
UA_STATE[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.65 EE.STF.128.IP

Instruction Word

10010	fv3[3:1]	fv0[3:0]	fv3[0]	fv2[3:1]	fv1[3:0]	imm16f[3:0]	as[3:0]	111	fv2[0]

Assembler Syntax

EE.STF.128.IP fv3, fv2, fv1, fv0, as, -128..112

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后按照数据低位到高位顺序，将浮点寄存器 fu0、fu1、fu2、fu3 拼接成的 16-byte 数据存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 4-bit 符号位扩展常数左移 4 位的值。

Operation

```
{fv3,fv2,fv1,fv0} => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + {24{imm16f[3]},imm16f[3:0],4{0}}
```

1.8.66 EE.STF.128.XP

Instruction Word

10011	fv3[3:1]	fv0[3:0]	fv3[0]	fv2[3:1]	fv1[3:0]	ad[3:0]	as[3:0]	111	fv2[0]

Assembler Syntax

EE.STF.128.XP fv3, fv2, fv1, fv0, as, ad

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后按照数据低位到高位顺序，将浮点寄存器 fu0、fu1、fu2、fu3 拼接成的 16-byte 数据存储到内存。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
{fv3,fv2,fv1,fv0} => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.67 EE.STF.64.IP

Instruction Word

111000	imm8[7:6]	fv0[3:0]	imm8[5:2]	fv1[3:0]	011	imm8[0]	as[3:0]	111	imm8[1]

Assembler Syntax

EE.STF.64.IP fv1, fv0, as, imm8

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后按照数据低位到高位顺序，将浮点寄存器 fu0、fu1 拼接成的 64-bit 数据存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 3 位的值。

Operation

```
{fv1,fv0} => store64({as[31:3],3{0}})
as[31:0] = as[31:0] + {21{imm8[7]},imm8[7:0],3{0}}
```

1.8.68 EE.STF.64.XP

Instruction Word

fv0[3:0]	0111	fv1[3:0]	ad[3:0]	as[3:0]	0000

Assembler Syntax

EE.STF.64.XP fv1, fv0, as, ad

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后按照数据低位到高位顺序，将浮点寄存器 fu0、fu1 拼接成的 64-bit 数据存储到内存。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
{fv1,fv0} => store64({as[31:3],3{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.69 EE.STXQ.32

Instruction Word

1110011	sel4[1]	sel8[0]	qv[2:0]	sel4[0]	000	qs[1:0]	sel8[2:1]	0000	as[3:0]	111	qs[2]

Assembler Syntax

EE.STXQ.32 qv, qs, as, sel4, sel8

Description

根据立即数 sel4 从 qv 寄存器的四个 32-bit 数据段中挑选出一个 32-bit 数据，写入内存。根据立即数 sel8 从 qs 寄存器 8 个 16-bit 数据段中挑出一个加数并左移 2 位，随后与地址寄存器 as 相加，得到的数值结果取 32-bit 对齐（将访存地址的低 2 比特置 0），从而计算出访问内存的地址。

Operation

```
  vaddr0[31:0] = as[31:0] + qs[ 15:  0] * 4
  vaddr1[31:0] = as[31:0] + qs[ 31: 16] * 4
  vaddr2[31:0] = as[31:0] + qs[ 47: 32] * 4
  vaddr3[31:0] = as[31:0] + qs[ 63: 47] * 4
  vaddr4[31:0] = as[31:0] + qs[ 79: 64] * 4
  vaddr5[31:0] = as[31:0] + qs[ 95: 80] * 4
  vaddr6[31:0] = as[31:0] + qs[111: 96] * 4
  vaddr7[31:0] = as[31:0] + qs[127:112] * 4

if sel8 == 0:
  qv[32*sel4+31:32*sel4] =>store32({vaddr0[31:2],2{0}})
if sel8 == 1:
  qv[32*sel4+31:32*sel4] =>store32({vaddr1[31:2],2{0}})
if sel8 == 2:
  qv[32*sel4+31:32*sel4] =>store32({vaddr2[31:2],2{0}})
if sel8 == 3:
  qv[32*sel4+31:32*sel4] =>store32({vaddr3[31:2],2{0}})
if sel8 == 4:
  qv[32*sel4+31:32*sel4] =>store32({vaddr4[31:2],2{0}})
if sel8 == 5:
  qv[32*sel4+31:32*sel4] =>store32({vaddr5[31:2],2{0}})
if sel8 == 6:
  qv[32*sel4+31:32*sel4] =>store32({vaddr6[31:2],2{0}})
if sel8 == 7:
  qv[32*sel4+31:32*sel4] =>store32({vaddr7[31:2],2{0}})
```

1.8.70 EE.VADDS.S16

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	0	qy[1:0]	qx[2:0]	01100100

Assembler Syntax

EE.VADDS.S16 qa, qx, qy

Description

该指令执行数据位宽为 16-bit 的向量加法。qx、qy 寄存器是两个加数，对计算得到 8 个数据结果取饱和后写入 qa 寄存器。

Operation

```
qa[ 15:  0] = min(max(qx[ 15:  0] + qy[ 15:  0], -2^{15}), 2^{15}-1)
qa[ 31: 16] = min(max(qx[ 31: 16] + qy[ 31: 16], -2^{15}), 2^{15}-1)
...
```

1.8.71 EE.VADDS.S16.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	010	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1101	as[3:0]	111	qx[2]

Assembler Syntax

EE.VADDS.S16.LD.INCP qu, as, qa, qx, qy

Description

该指令执行数据位宽为 16-bit 的向量加法。qx、qy 寄存器是两个加数，对计算得到 8 个数据结果取饱和后写入 qa 寄存器。执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 15:  0] = min(max(qx[ 15:  0] + qy[ 15:  0], -2^{15}), 2^{15}-1)
qa[ 31: 16] = min(max(qx[ 31: 16] + qy[ 31: 16], -2^{15}), 2^{15}-1)
...

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.72 EE.VADDS.S16.ST.INCP

Instruction Word

11100100	qy[0]	qv[2:0]	1	qa[2:0]	qx[1:0]	qy[2:1]	0000	as[3:0]	111	qx[2]

Assembler Syntax

EE.VADDS.S16.ST.INCP qv, as, qa, qx, qy

Description

该指令执行数据位宽为 16-bit 的向量加法。qx、qy 寄存器是两个加数，对计算得到 8 个数据结果取饱和后写入 qa 寄存器。执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 15:  0] = min(max(qx[ 15:  0] + qy[ 15:  0], -2^{15}), 2^{15}-1)
qa[ 31: 16] = min(max(qx[ 31: 16] + qy[ 31: 16], -2^{15}), 2^{15}-1)
...

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.73 EE.VADDS.S32

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	0	qy[1:0]	qx[2:0]	01110100

Assembler Syntax

EE.VADDS.S32 qa, qx, qy

Description

该指令执行数据位宽为 32-bit 的向量加法。qx、qy 寄存器是两个加数，对计算得到 4 个数据结果取饱和后写入 qa 寄存器。

Operation

```
qa[ 31:  0] = min(max(qx[ 31:  0] + qy[ 31:  0], -2^{31}), 2^{31}-1)
qa[ 63: 32] = min(max(qx[ 63: 32] + qy[ 63: 32], -2^{31}), 2^{31}-1)
...
```

1.8.74 EE.VADDS.S32.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	011	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1101	as[3:0]	111	qx[2]

Assembler Syntax

EE.VADDS.S32.LD.INCP qu, as, qa, qx, qy

Description

该指令执行数据位宽为 32-bit 的向量加法。qx、qy 寄存器是两个加数，对计算得到 4 个数据结果取饱和后写入 qa 寄存器。执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 31:  0] = min(max(qx[ 31:  0] + qy[ 31:  0], -2^{31}), 2^{31}-1)
qa[ 63: 32] = min(max(qx[ 63: 32] + qy[ 63: 32], -2^{31}), 2^{31}-1)
...

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.75 EE.VADDS.S32.ST.INCP

Instruction Word

11100100	qy[0]	qv[2:0]	1	qa[2:0]	qx[1:0]	qy[2:1]	0001	as[3:0]	111	qx[2]

Assembler Syntax

EE.VADDS.S32.ST.INCP qv, as, qa, qx, qy

Description

该指令执行数据位宽为 32-bit 的向量加法。qx、qy 寄存器是两个加数，对计算得到 4 个数据结果取饱和后写入 qa 寄存器。执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 31:  0] = min(max(qx[ 31:  0] + qy[ 31:  0], -2^{31}), 2^{31}-1)
qa[ 63: 32] = min(max(qx[ 63: 32] + qy[ 63: 32], -2^{31}), 2^{31}-1)
...

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.76 EE.VADDS.S8

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	0	qy[1:0]	qx[2:0]	10000100

Assembler Syntax

EE.VADDS.S8 qa, qx, qy

Description

该指令执行数据位宽为 8-bit 的向量加法。qx、qy 寄存器是两个加数，对计算得到 16 个数据结果取饱和后写入 qa 寄存器。

Operation

```
qa[  7:  0] = min(max(qx[  7:  0] + qy[  7:  0], -2^{7}), 2^{7}-1)
qa[ 15:  8] = min(max(qx[ 15:  8] + qy[ 15:  8], -2^{7}), 2^{7}-1)
...
qa[127:120] = min(max(qx[127:120] + qy[127:120], -2^{7}), 2^{7}-1)
```

1.8.77 EE.VADDS.S8.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	001	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1100	as[3:0]	111	qx[2]

Assembler Syntax

EE.VADDS.S8.LD.INCP qu, as, qa, qx, qy

Description

该指令执行数据位宽为 8-bit 的向量加法。qx、qy 寄存器是两个加数，对计算得到 16 个数据结果取饱和后写入 qa 寄存器。执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[  7:  0] = min(max(qx[  7:  0] + qy[  7:  0], -2^{7}), 2^{7}-1)
qa[ 15:  8] = min(max(qx[ 15:  8] + qy[ 15:  8], -2^{7}), 2^{7}-1)
...

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.78 EE.VADDS.S8.ST.INCP

Instruction Word

11100100	qy[0]	qv[2:0]	1	qa[2:0]	qx[1:0]	qy[2:1]	0010	as[3:0]	111	qx[2]

Assembler Syntax

EE.VADDS.S8.ST.INCP qv, as, qa, qx, qy

Description

该指令执行数据位宽为 8-bit 的向量加法。qx、qy 寄存器是两个加数，对计算得到 16 个数据结果取饱和后写入 qa 寄存器。执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[  7:  0] = min(max(qx[  7:  0] + qy[  7:  0], -2^{7}), 2^{7}-1)
qa[ 15:  8] = min(max(qx[ 15:  8] + qy[ 15:  8], -2^{7}), 2^{7}-1)
...

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.79 EE.VCMP.EQ.S16

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	0	qy[1:0]	qx[2:0]	10010100

Assembler Syntax

EE.VCMP.EQ.S16 qa, qx, qy

Description

该指令用于处理 16-bit 向量数据的大小比较。比较 qx、qy 寄存器中 8 个 16-bit 数据段的数值大小，若数值相等，将 0xFFFF 写入 qa 寄存器对应的 16-bit 数据段中。否则将 0 值写入 qa 寄存器对应的 16-bit 数据段中。

Operation

```
qa[ 15:  0] = (qx[ 15:  0]==qy[ 15:  0]) ? 0x{}FFFF : 0
qa[ 31: 16] = (qx[ 31: 16]==qy[ 31: 16]) ? 0x{}FFFF : 0
...
qa[127:112] = (qx[127:112]==qy[127:112]) ? 0x{}FFFF : 0
```

1.8.80 EE.VCMP.EQ.S32

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	0	qy[1:0]	qx[2:0]	10100100

Assembler Syntax

EE.VCMP.EQ.S32 qa, qx, qy

Description

该指令用于处理 32-bit 向量数据的大小比较。比较 qx、qy 寄存器中 4 个 32-bit 数据段的数值大小，若数值相等，将 0xFFFFFFFF 写入 qa 寄存器对应的 32-bit 数据段中。否则将 0 值写入 qa 寄存器对应的 32-bit 数据段中。

Operation

```
qa[ 31:  0] = (qx[ 31:  0]==qy[ 31:  0]) ? 0x{}FFFFFFFF : 0
qa[ 63: 32] = (qx[ 63: 32]==qy[ 63: 32]) ? 0x{}FFFFFFFF : 0
...
qa[127: 96] = (qx[127: 96]==qy[127: 96]) ? 0x{}FFFFFFFF : 0
```

1.8.81 EE.VCMP.EQ.S8

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	0	qy[1:0]	qx[2:0]	10110100

Assembler Syntax

EE.VCMP.EQ.S8 qa, qx, qy

Description

该指令用于处理 8-bit 向量数据的大小比较。比较 qx、qy 寄存器中 16 个 8-bit 数据段的数值大小，若数值相等，将 0xFF 写入 qa 寄存器对应的 8-bit 数据段中。否则将 0 值写入 qa 寄存器对应的 8-bit 数据段中。

Operation

```
qa[  7:  0] = (qx[  7:  0]==qy[  7:  0]) ? 0x{}FF : 0
qa[ 15:  8] = (qx[ 15:  8]==qy[ 15:  8]) ? 0x{}FF : 0
...
qa[127:120] = (qx[127:120]==qy[127:120]) ? 0x{}FF : 0
```

1.8.82 EE.VCMP.GT.S16

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	0	qy[1:0]	qx[2:0]	11000100

Assembler Syntax

EE.VCMP.GT.S16 qa, qx, qy

Description

该指令用于处理 16-bit 向量数据的大小比较。比较 qx、qy 寄存器中 8 个 16-bit 数据段的数值大小，若前者的数值大于后者，将 0xFFFF 写入 qa 寄存器对应的 16-bit 数据段中。否则将 0 值写入 qa 寄存器对应的 16-bit 数据段中。

Operation

```
qa[ 15:  0] = (qx[ 15:  0]>qy[ 15:  0]) ? 0x{}FFFF : 0
qa[ 31: 16] = (qx[ 31: 16]>qy[ 31: 16]) ? 0x{}FFFF : 0
...
qa[127:112] = (qx[127:112]>qy[127:112]) ? 0x{}FFFF : 0
```

1.8.83 EE.VCMP.GT.S32

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	0	qy[1:0]	qx[2:0]	11010100

Assembler Syntax

EE.VCMP.GT.S32 qa, qx, qy

Description

该指令用于处理 32-bit 向量数据的大小比较。比较 qx、qy 寄存器中 4 个 32-bit 数据段的数值大小，若前者的数值大于后者，将 0xFFFFFFFF 写入 qa 寄存器对应的 32-bit 数据段中。否则将 0 值写入 qa 寄存器对应的 32-bit 数据段中。

Operation

```
qa[ 31:  0] = (qx[ 31:  0]>qy[ 31:  0]) ? 0x{}FFFFFFFF : 0
qa[ 63: 32] = (qx[ 63: 32]>qy[ 63: 32]) ? 0x{}FFFFFFFF : 0
...
qa[127: 96] = (qx[127: 96]>qy[127: 96]) ? 0x{}FFFFFFFF : 0
```

1.8.84 EE.VCMP.GT.S8

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	0	qy[1:0]	qx[2:0]	11100100

Assembler Syntax

EE.VCMP.GT.S8 qa, qx, qy

Description

该指令用于处理 8-bit 向量数据的大小比较。比较 qx、qy 寄存器中 16 个 8-bit 数据段的数值大小，若前者的数值大于后者，将 0xFF 写入 qa 寄存器对应的 8-bit 数据段中。否则将 0 值写入 qa 寄存器对应的 8-bit 数据段中。

Operation

```
qa[  7:  0] = (qx[  7:  0]>qy[  7:  0]) ? 0x{}FF : 0
qa[ 15:  8] = (qx[ 15:  8]>qy[ 15:  8]) ? 0x{}FF : 0
...
qa[127:120] = (qx[127:120]>qy[127:120]) ? 0x{}FF : 0
```

1.8.85 EE.VCMP.LT.S16

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	0	qy[1:0]	qx[2:0]	11110100

Assembler Syntax

EE.VCMP.LT.S16 qa, qx, qy

Description

该指令用于处理 16-bit 向量数据的大小比较。比较 qx、qy 寄存器中 8 个 16-bit 数据段的数值大小，若前者的数值小于后者，将 0xFFFF 写入 qa 寄存器对应的 16-bit 数据段中。否则将 0 值写入 qa 寄存器对应的 16-bit 数据段中。

Operation

```
qa[ 15:  0] = (qx[ 15:  0]<qy[ 15:  0]) ? 0x{}FFFF : 0
qa[ 31: 16] = (qx[ 31: 16]<qy[ 31: 16]) ? 0x{}FFFF : 0
...
qa[127:112] = (qx[127:112]<qy[127:112]) ? 0x{}FFFF : 0
```

1.8.86 EE.VCMP.LT.S32

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	00000100

Assembler Syntax

EE.VCMP.LT.S32 qa, qx, qy

Description

该指令用于处理 32-bit 向量数据的大小比较。比较 qx、qy 寄存器中 4 个 32-bit 数据段的数值大小，若前者的数值小于后者，将 0xFFFFFFFF 写入 qa 寄存器对应的 32-bit 数据段中。否则将 0 值写入 qa 寄存器对应的 32-bit 数据段中。

Operation

```
qa[ 31:  0] = (qx[ 31:  0]<qy[ 31:  0]) ? 0x{}FFFFFFFF : 0
qa[ 63: 32] = (qx[ 63: 32]<qy[ 63: 32]) ? 0x{}FFFFFFFF : 0
...
qa[127: 96] = (qx[127: 96]<qy[127: 96]) ? 0x{}FFFFFFFF : 0
```

1.8.87 EE.VCMP.LT.S8

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	00010100

Assembler Syntax

EE.VCMP.LT.S8 qa, qx, qy

Description

该指令用于处理 8-bit 向量数据的大小比较。比较 qx、qy 寄存器中 16 个 8-bit 数据段的数值大小，若前者的数值小于后者，将 0xFF 写入 qa 寄存器对应的 8-bit 数据段中。否则将 0 值写入 qa 寄存器对应的 8-bit 数据段中。

Operation

```
qa[  7:  0] = (qx[  7:  0]<qy[  7:  0]) ? 0x{}FF : 0
qa[ 15:  8] = (qx[ 15:  8]<qy[ 15:  8]) ? 0x{}FF : 0
...
qa[127:120] = (qx[127:120]<qy[127:120]) ? 0x{}FF : 0
```

1.8.88 EE.VLD.128.IP

Instruction Word

1	imm16[7]	qu[2:1]	0011	qu[0]	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.VLD.128.IP qu, as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.89 EE.VLD.128.XP

Instruction Word

10	qu[2:1]	1101	qu[0]	010	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.VLD.128.XP qu, as, ad

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.90 EE.VLD.H.64.IP

Instruction Word

1	imm8[7]	qu[2:1]	1000	qu[0]	imm8[6:0]	as[3:0]	0100

Assembler Syntax

EE.VLD.H.64.IP qu, as, -1024..1016

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后从内存中载入 64-bit 数据，并存入 qu 寄存器的高 64-bit 区域。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 3 位的值。

Operation

```
qu[127: 64] = load64({as[31:  3],3{0}})
as[31:0] = as[31:0] + {21{imm8[7]},imm8[7:0],3{0}}
```

1.8.91 EE.VLD.H.64.XP

Instruction Word

10	qu[2:1]	1101	qu[0]	110	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.VLD.H.64.XP qu, as, ad

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后从内存中载入 64-bit 数据，并存入 qu 寄存器的高 64-bit 区域。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
qu[127: 64] = load64({as[31:  3],3{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.92 EE.VLD.L.64.IP

Instruction Word

1	imm8[7]	qu[2:1]	1001	qu[0]	imm8[6:0]	as[3:0]	0100

Assembler Syntax

EE.VLD.L.64.IP qu, as, -1024..1016

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后从内存中载入 64-bit 数据，并存入 qu 寄存器低高 64-bit 区域。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 3 位的值。

Operation

```
qu[ 63:  0] = load64({as[31:3],3{0}})
as[31:0] = as[31:0] + {21{imm8[7]},imm8[7:0],3{0}}
```

1.8.93 EE.VLD.L.64.XP

Instruction Word

10	qu[2:1]	1101	qu[0]	011	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.VLD.L.64.XP qu, as, ad

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后从内存中载入 64-bit 数据，并存入 qu 寄存器的低 64-bit 区域。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
qu[ 63:  0] = load64({as[31:3],3{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.94 EE.VLDBC.16

Instruction Word

11	qu[2:1]	1101	qu[0]	1110011	as[3:0]	0100

Assembler Syntax

EE.VLDBC.16 qu, as

Description

将寄存器 as 内访存地址低 1 比特强制为 0，然后从内存中载入 16-bit 数据，并广播到 qu 寄存器中 8 个 16-bit 段内。

Operation

```
qu[127:0] = {8{load16({as[31:1],1{0}})}}
```

1.8.95 EE.VLDBC.16.IP

Instruction Word

10	qu[2:1]	0101	qu[0]	imm2[6:0]	as[3:0]	0100

Assembler Syntax

EE.VLDBC.16.IP qu, as, 0..254

Description

将寄存器 as 内访存地址低 1 比特强制为 0，然后从内存中载入 16-bit 数据，并广播到 qu 寄存器中 8 个 16-bit 段内。访存结束后，as 寄存器内数值增加指令编码段中 7-bit 无符号常数左移 1 位的值。

Operation

```
qu[127:0] = {8{load16({as[31:1],1{0}})}}
as[31:0] = as[31:0] + {24{0},imm2[6:0],0}
```

1.8.96 EE.VLDBC.16.XP

Instruction Word

10	qu[2:1]	1101	qu[0]	100	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.VLDBC.16.XP qu, as, ad

Description

将寄存器 as 内访存地址低 1 比特强制为 0，然后从内存中载入 16-bit 数据，并广播到 qu 寄存器中 8 个 16-bit 段内。访存结束后，as 寄存器内数值增加 ad 寄存器内数值

Operation

```
qu[127:0] = {8{load16({as[31:1],1{0}})}}
as  = as + ad[31:0]
```

1.8.97 EE.VLDBC.32

Instruction Word

11	qu[2:1]	1101	qu[0]	1110111	as[3:0]	0100

Assembler Syntax

EE.VLDBC.32 qu, as

Description

将寄存器 as 内访存地址低 2 比特强制为 0，然后从内存中载入 32-bit 数据，并广播到 qu 寄存器中 4 个 32-bit 段内。

Operation

```
qu[127:0] = {4{load32({as[31:2],2{0}})}}
```

1.8.98 EE.VLDBC.32.IP

Instruction Word

1	imm4[7]	qu[2:1]	0010	qu[0]	imm4[6:0]	as[3:0]	0100

Assembler Syntax

EE.VLDBC.32.IP qu, as, -256..252

Description

将寄存器 as 内访存地址低 2 比特强制为 0，然后从内存中载入 32-bit 数据，并广播到 qu 寄存器中 4 个 32-bit 段内。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 2 位的值。

Operation

```
qu[127:0] = {4{load32({as[31:2],2{0}})}}
as[31:0] = as[31:0] + {22{imm4[7]},imm4[7:0],2{0}}
```

1.8.99 EE.VLDBC.32.XP

Instruction Word

10	qu[2:1]	1101	qu[0]	001	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.VLDBC.32.XP qu, as, ad

Description

将寄存器 as 内访存地址低 2 比特强制为 0，然后从内存中载入 32-bit 数据，并广播到 qu 寄存器中 4 个 32-bit 段内。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
qu[127:0] = {4{load32({as[31:2],2{0}})}}
as  = as + ad[31:0]
```

1.8.100 EE.VLDBC.8

Instruction Word

11	qu[2:1]	1101	qu[0]	0111011	as[3:0]	0100

Assembler Syntax

EE.VLDBC.8 qu, as

Description

根据访存寄存器 as 中指示的地址，从内存中载入 8-bit 数据，并广播到 qu 寄存器中 16 个 8-bit 段内。

Operation

```
qu[127:0] = {16{load8(as[31:0])}}
```

1.8.101 EE.VLDBC.8.IP

Instruction Word

11	qu[2:1]	0101	qu[0]	imm1[6:0]	as[3:0]	0100

Assembler Syntax

EE.VLDBC.8.IP qu, as, 0..127

Description

根据访存寄存器 as 中指示的地址，从内存中载入 8-bit 数据，并广播到 qu 寄存器中 16 个 8-bit 段内。访存结束后，as 寄存器内数值增加指令编码段中 7-bit 无符号常数值。

Operation

```
qu[127:0] = {16{load8(as[31:0])}}
as[31:0] = as[31:0] + {25{0},imm1[6:0]}
```

1.8.102 EE.VLDBC.8.XP

Instruction Word

10	qu[2:1]	1101	qu[0]	101	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.VLDBC.8.XP qu, as, ad

Description

根据访存寄存器 as 中指示的地址，从内存中载入 8-bit 数据，并广播到 qu 寄存器中 16 个 8-bit 段内。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
qu[127:0] = {16{load8(as[31:0])}}
as  = as + ad[31:0]
```

1.8.103 EE.VLDHBC.16.INCP

Instruction Word

11	qu[2:1]	1100	qu[0]	qu1[2:0]	0010	as[3:0]	0100

Assembler Syntax

EE.VLDHBC.16.INCP qu, qu1, as

Description

将寄存器 as 内访存地址低 4 比特强制为 0，随后从内存中载入 16-byte 数据，随后按照如下特定方式将 16-byte 数据扩展成 32-byte 数据，并赋值给 qu、qu1 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
dataIn[127:0] = load128({as[31:4],4{0}})
qu  = {2{dataIn[ 63: 48]}, 2{dataIn[ 47: 32]}, 2{dataIn[ 31: 16]}, 2{dataIn[ 15: 0]} }
qu1 = {2{dataIn[127:112]}, 2{dataIn[111: 96]}, 2{dataIn[ 95: 80]}, 2{dataIn[ 79: 64]} }
as[31:0] = as[31:0] + 16
```

1.8.104 EE.VMAX.S16

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	00100100

Assembler Syntax

EE.VMAX.S16 qa, qx, qy

Description

该指令用于处理 16-bit 向量数据的大小比较。qx、qy 寄存器中 8 个 16-bit 数据段的数值大小。将其中数值较大的数据段写入 qa 寄存器对应的 16-bit 数据段中。

Operation

```
qa[ 15:  0] = (qx[ 15:  0]>=qy[ 15:  0]) ? qx[ 15:  0] : qy[ 15:  0]
qa[ 31: 16] = (qx[ 31: 16]>=qy[ 31: 16]) ? qx[ 31: 16] : qy[ 31: 16]
...
qa[127:112] = (qx[127:112]>=qy[127:112]) ? qx[127:112] : qy[127:112]
```

1.8.105 EE.VMAX.S16.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	001	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1101	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMAX.S16.LD.INCP qu, as, qa, qx, qy

Description

该指令用于处理 16-bit 向量数据的大小比较。qx、qy 寄存器中 8 个 16-bit 数据段的数值大小。将其中数值较大的数据段写入 qa 寄存器对应的 16-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 15:  0] = (qx[ 15:  0]>=qy[ 15:  0]) ? qx[ 15:  0] : qy[ 15:  0]
qa[ 31: 16] = (qx[ 31: 16]>=qy[ 31: 16]) ? qx[ 31: 16] : qy[ 31: 16]
...
qa[127:112] = (qx[127:112]>=qy[127:112]) ? qx[127:112] : qy[127:112]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.106 EE.VMAX.S16.ST.INCP

Instruction Word

11100100	qy[0]	qv[2:0]	1	qa[2:0]	qx[1:0]	qy[2:1]	0011	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMAX.S16.ST.INCP qv, as, qa, qx, qy

Description

该指令用于处理 16-bit 向量数据的大小比较。qx、qy 寄存器中 8 个 16-bit 数据段的数值大小。将其中数值较大的数据段写入 qa 寄存器对应的 16-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 15:  0] = (qx[ 15:  0]>=qy[ 15:  0]) ? qx[ 15:  0] : qy[ 15:  0]
qa[ 31: 16] = (qx[ 31: 16]>=qy[ 31: 16]) ? qx[ 31: 16] : qy[ 31: 16]
...
qa[127:112] = (qx[127:112]>=qy[127:112]) ? qx[127:112] : qy[127:112]

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.107 EE.VMAX.S32

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	00110100

Assembler Syntax

EE.VMAX.S32 qa, qx, qy

Description

该指令用于处理 32-bit 向量数据的大小比较。qx、qy 寄存器中 4 个 32-bit 数据段的数值大小。将其中数值较大的数据段写入 qa 寄存器对应的 32-bit 数据段中。

Operation

```
qa[ 31:  0] = (qx[ 31:  0]>=qy[ 31:  0]) ? qx[ 31:  0] : qy[ 31:  0]
qa[ 63: 32] = (qx[ 63: 32]>=qy[ 63: 32]) ? qx[ 63: 32] : qy[ 63: 32]
...
qa[127: 96] = (qx[127: 96]>=qy[127: 96]) ? qx[127: 96] : qy[127: 96]
```

1.8.108 EE.VMAX.S32.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	001	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1110	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMAX.S32.LD.INCP qu, as, qa, qx, qy

Description

该指令用于处理 32-bit 向量数据的大小比较。qx、qy 寄存器中 4 个 32-bit 数据段的数值大小。将其中数值较大的数据段写入 qa 寄存器对应的 32-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 31:  0] = (qx[ 31:  0]>=qy[ 31:  0]) ? qx[ 31:  0] : qy[ 31:  0]
qa[ 63: 32] = (qx[ 63: 32]>=qy[ 63: 32]) ? qx[ 63: 32] : qy[ 63: 32]
...
qa[127: 96] = (qx[127: 96]>=qy[127: 96]) ? qx[127: 96] : qy[127: 96]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.109 EE.VMAX.S32.ST.INCP

Instruction Word

11100101	qy[0]	qv[2:0]	0	qa[2:0]	qx[1:0]	qy[2:1]	0000	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMAX.S32.ST.INCP qv, as, qa, qx, qy

Description

该指令用于处理 32-bit 向量数据的大小比较。qx、qy 寄存器中 4 个 32-bit 数据段的数值大小。将其中数值较大的数据段写入 qa 寄存器对应的 32-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 31:  0] = (qx[ 31:  0]>=qy[ 31:  0]) ? qx[ 31:  0] : qy[ 31:  0]
qa[ 63: 32] = (qx[ 63: 32]>=qy[ 63: 32]) ? qx[ 63: 32] : qy[ 63: 32]
...
qa[127: 96] = (qx[127: 96]>=qy[127: 96]) ? qx[127: 96] : qy[127: 96]

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.110 EE.VMAX.S8

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	01000100

Assembler Syntax

EE.VMAX.S8 qa, qx, qy

Description

该指令用于处理 8-bit 向量数据的大小比较。qx、qy 寄存器中 16 个 8-bit 数据段的数值大小。将其中数值较大的数据段写入 qa 寄存器对应的 8-bit 数据段中。

Operation

```
qa[  7:  0] = (qx[  7:  0]>=qy[  7:  0]) ? qx[  7:  0] : qy[  7:  0]
qa[ 15:  8] = (qx[ 15:  8]>=qy[ 15:  8]) ? qx[ 15:  8] : qy[ 15:  8]
...
qa[127:120] = (qx[127:120]>=qy[127:120]) ? qx[127:120] : qy[127:120]
```

1.8.111 EE.VMAX.S8.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	001	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1111	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMAX.S8.LD.INCP qu, as, qa, qx, qy

Description

该指令用于处理 8-bit 向量数据的大小比较。qx、qy 寄存器中 16 个 8-bit 数据段的数值大小。将其中数值较大的数据段写入 qa 寄存器对应的 8-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[  7:  0] = (qx[  7:  0]>=qy[  7:  0]) ? qx[  7:  0] : qy[  7:  0]
qa[ 15:  8] = (qx[ 15:  8]>=qy[ 15:  8]) ? qx[ 15:  8] : qy[ 15:  8]
...
qa[127:120] = (qx[127:120]>=qy[127:120]) ? qx[127:120] : qy[127:120]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.112 EE.VMAX.S8.ST.INCP

Instruction Word

11100101	qy[0]	qv[2:0]	1	qa[2:0]	qx[1:0]	qy[2:1]	0000	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMAX.S8.ST.INCP qv, as, qa, qx, qy

Description

该指令用于处理 8-bit 向量数据的大小比较。qx、qy 寄存器中 16 个 8-bit 数据段的数值大小。将其中数值较大的数据段写入 qa 寄存器对应的 8-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[  7:  0] = (qx[  7:  0]>=qy[  7:  0]) ? qx[  7:  0] : qy[  7:  0]
qa[ 15:  8] = (qx[ 15:  8]>=qy[ 15:  8]) ? qx[ 15:  8] : qy[ 15:  8]
...
qa[127:120] = (qx[127:120]>=qy[127:120]) ? qx[127:120] : qy[127:120]

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.113 EE.VMIN.S16

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	01010100

Assembler Syntax

EE.VMIN.S16 qa, qx, qy

Description

该指令用于处理 16-bit 向量数据的大小比较。qx、qy 寄存器中 8 个 16-bit 数据段的数值大小。将其中数值较小的数据段写入 qa 寄存器对应的 16-bit 数据段中。

Operation

```
qa[ 15:  0] = (qx[ 15:  0]<=qy[ 15:  0]) ? qx[ 15:  0] : qy[ 15:  0]
qa[ 31: 16] = (qx[ 31: 16]<=qy[ 31: 16]) ? qx[ 31: 16] : qy[ 31: 16]
...
qa[127:112] = (qx[127:112]<=qy[127:112]) ? qx[127:112] : qy[127:112]
```

1.8.114 EE.VMIN.S16.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	010	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1110	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMIN.S16.LD.INCP qu, as, qa, qx, qy

Description

该指令用于处理 16-bit 向量数据的大小比较。qx、qy 寄存器中 8 个 16-bit 数据段的数值大小。将其中数值较小的数据段写入 qa 寄存器对应的 16-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 15:  0] = (qx[ 15:  0]<=qy[ 15:  0]) ? qx[ 15:  0] : qy[ 15:  0]
qa[ 31: 16] = (qx[ 31: 16]<=qy[ 31: 16]) ? qx[ 31: 16] : qy[ 31: 16]
...
qa[127:112] = (qx[127:112]<=qy[127:112]) ? qx[127:112] : qy[127:112]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.115 EE.VMIN.S16.ST.INCP

Instruction Word

11100101	qy[0]	qv[2:0]	0	qa[2:0]	qx[1:0]	qy[2:1]	0001	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMIN.S16.ST.INCP qv, as, qa, qx, qy

Description

该指令用于处理 16-bit 向量数据的大小比较。qx、qy 寄存器中 8 个 16-bit 数据段的数值大小。将其中数值较小的数据段写入 qa 寄存器对应的 16-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 15:  0] = (qx[ 15:  0]<=qy[ 15:  0]) ? qx[ 15:  0] : qy[ 15:  0]
qa[ 31: 16] = (qx[ 31: 16]<=qy[ 31: 16]) ? qx[ 31: 16] : qy[ 31: 16]
...
qa[127:112] = (qx[127:112]<=qy[127:112]) ? qx[127:112] : qy[127:112]

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.116 EE.VMIN.S32

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	01100100

Assembler Syntax

EE.VMIN.S32 qa, qx, qy

Description

该指令用于处理 32-bit 向量数据的大小比较。qx、qy 寄存器中 4 个 32-bit 数据段的数值大小。将其中数值较小的数据段写入 qa 寄存器对应的 32-bit 数据段中。

Operation

```
qa[ 31:  0] = (qx[ 31:  0]<=qy[ 31:  0]) ? qx[ 31:  0] : qy[ 31:  0]
qa[ 63: 32] = (qx[ 63: 32]<=qy[ 63: 32]) ? qx[ 63: 32] : qy[ 63: 32]
...
qa[127: 96] = (qx[127: 96]<=qy[127: 96]) ? qx[127: 96] : qy[127: 96]
```

1.8.117 EE.VMIN.S32.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	011	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1110	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMIN.S32.LD.INCP qu, as, qa, qx, qy

Description

该指令用于处理 32-bit 向量数据的大小比较。qx、qy 寄存器中 4 个 32-bit 数据段的数值大小。将其中数值较小的数据段写入 qa 寄存器对应的 32-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 31:  0] = (qx[ 31:  0]<=qy[ 31:  0]) ? qx[ 31:  0] : qy[ 31:  0]
qa[ 63: 32] = (qx[ 63: 32]<=qy[ 63: 32]) ? qx[ 63: 32] : qy[ 63: 32]
...
qa[127: 96] = (qx[127: 96]<=qy[127: 96]) ? qx[127: 96] : qy[127: 96]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.118 EE.VMIN.S32.ST.INCP

Instruction Word

11100101	qy[0]	qv[2:0]	1	qa[2:0]	qx[1:0]	qy[2:1]	0001	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMIN.S32.ST.INCP qv, as, qa, qx, qy

Description

该指令用于处理 32-bit 向量数据的大小比较。qx、qy 寄存器中 4 个 32-bit 数据段的数值大小。将其中数值较小的数据段写入 qa 寄存器对应的 32-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 31:  0] = (qx[ 31:  0]<=qy[ 31:  0]) ? qx[ 31:  0] : qy[ 31:  0]
qa[ 63: 32] = (qx[ 63: 32]<=qy[ 63: 32]) ? qx[ 63: 32] : qy[ 63: 32]
...
qa[127: 96] = (qx[127: 96]<=qy[127: 96]) ? qx[127: 96] : qy[127: 96]

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.119 EE.VMIN.S8

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	01110100

Assembler Syntax

EE.VMIN.S8 qa, qx, qy

Description

该指令用于处理 8-bit 向量数据的大小比较。qx、qy 寄存器中 16 个 8-bit 数据段的数值大小。将其中数值较小的数据段写入 qa 寄存器对应的 8-bit 数据段中。

Operation

```
qa[  7:  0] = (qx[  7:  0]<=qy[  7:  0]) ? qx[  7:  0] : qy[  7:  0]
qa[ 15:  8] = (qx[ 15:  8]<=qy[ 15:  8]) ? qx[ 15:  8] : qy[ 15:  8]
...
qa[127:120] = (qx[127:120]<=qy[127:120]) ? qx[127:120] : qy[127:120]
```

1.8.120 EE.VMIN.S8.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	010	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1111	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMIN.S8.LD.INCP qu, as, qa, qx, qy

Description

该指令用于处理 8-bit 向量数据的大小比较。qx、qy 寄存器中 16 个 8-bit 数据段的数值大小。将其中数值较小的数据段写入 qa 寄存器对应的 8-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[  7:  0] = (qx[  7:  0]<=qy[  7:  0]) ? qx[  7:  0] : qy[  7:  0]
qa[ 15:  8] = (qx[ 15:  8]<=qy[ 15:  8]) ? qx[ 15:  8] : qy[ 15:  8]
...
qa[127:120] = (qx[127:120]<=qy[127:120]) ? qx[127:120] : qy[127:120]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.121 EE.VMIN.S8.ST.INCP

Instruction Word

11100101	qy[0]	qv[2:0]	0	qa[2:0]	qx[1:0]	qy[2:1]	0010	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMIN.S8.ST.INCP qv, as, qa, qx, qy

Description

该指令用于处理 8-bit 向量数据的大小比较。qx、qy 寄存器中 16 个 8-bit 数据段的数值大小。将其中数值较小的数据段写入 qa 寄存器对应的 8-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[  7:  0] = (qx[  7:  0]<=qy[  7:  0]) ? qx[  7:  0] : qy[  7:  0]
qa[ 15:  8] = (qx[ 15:  8]<=qy[ 15:  8]) ? qx[ 15:  8] : qy[ 15:  8]
...
qa[127:120] = (qx[127:120]<=qy[127:120]) ? qx[127:120] : qy[127:120]

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.122 EE.VMUL.S16

Instruction Word

10	qz[2:1]	1110	qz[0]	qy[2]	1	qy[1:0]	qx[2:0]	10000100

Assembler Syntax

EE.VMUL.S16 qz, qx, qy

Description

该指令执行数据位宽为 16-bit 的有符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 8 个 32-bit 数据结果进行算术右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 16-bit 数据，写入 qz 寄存器对应的数据段。

Operation

```
qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0]) >> SAR[5:0]
qz[ 31: 16] = (qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32]) >> SAR[5:0]
qz[ 63: 48] = (qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64]) >> SAR[5:0]
qz[ 95: 80] = (qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
qz[111: 96] = (qx[111: 96] * qy[111: 96]) >> SAR[5:0]
qz[127:112] = (qx[127:112] * qy[127:112]) >> SAR[5:0]
```

1.8.123 EE.VMUL.S16.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	011	qu[0]	qz[2:0]	qx[1:0]	qy[2:1]	1111	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMUL.S16.LD.INCP qu, as, qz, qx, qy

Description

该指令执行数据位宽为 16-bit 的有符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 8 个 32-bit 数据结果进行算术右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 16-bit 数据，写入 qz 寄存器对应的数据段。执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0]) >> SAR[5:0]
qz[ 31: 16] = (qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32]) >> SAR[5:0]
qz[ 63: 48] = (qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64]) >> SAR[5:0]
qz[ 95: 80] = (qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
qz[111: 96] = (qx[111: 96] * qy[111: 96]) >> SAR[5:0]
qz[127:112] = (qx[127:112] * qy[127:112]) >> SAR[5:0]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.124 EE.VMUL.S16.ST.INCP

Instruction Word

11100101	qy[0]	qv[2:0]	1	qz[2:0]	qx[1:0]	qy[2:1]	0010	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMUL.S16.ST.INCP qv, as, qz, qx, qy

Description

该指令执行数据位宽为 16-bit 的有符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 8 个 32-bit 数据结果进行算术右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 16-bit 数据，写入 qz 寄存器对应的数据段。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0]) >> SAR[5:0]
qz[ 31: 16] = (qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32]) >> SAR[5:0]
qz[ 63: 48] = (qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64]) >> SAR[5:0]
qz[ 95: 80] = (qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
qz[111: 96] = (qx[111: 96] * qy[111: 96]) >> SAR[5:0]
qz[127:112] = (qx[127:112] * qy[127:112]) >> SAR[5:0]

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.125 EE.VMUL.S8

Instruction Word

10	qz[2:1]	1110	qz[0]	qy[2]	1	qy[1:0]	qx[2:0]	10010100

Assembler Syntax

EE.VMUL.S8 qz, qx, qy

Description

该指令执行数据位宽为 8-bit 的有符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 16 个 16-bit 数据结果进行算术右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 8-bit 数据，写入 qz 寄存器对应的数据段。

Operation

```
qz[  7:  0] = (qx[  7:  0] * qy[  7:  0]) >> SAR[5:0]
qz[ 15:  8] = (qx[ 15:  8] * qy[ 15:  8]) >> SAR[5:0]
qz[ 23: 16] = (qx[ 23: 16] * qy[ 23: 16]) >> SAR[5:0]
qz[ 31: 24] = (qx[ 31: 24] * qy[ 31: 24]) >> SAR[5:0]
qz[ 39: 32] = (qx[ 39: 32] * qy[ 39: 32]) >> SAR[5:0]
qz[ 47: 40] = (qx[ 47: 40] * qy[ 47: 40]) >> SAR[5:0]
qz[ 55: 48] = (qx[ 55: 48] * qy[ 55: 48]) >> SAR[5:0]
qz[ 63: 56] = (qx[ 63: 56] * qy[ 63: 56]) >> SAR[5:0]
qz[ 71: 64] = (qx[ 71: 64] * qy[ 71: 64]) >> SAR[5:0]
qz[ 79: 72] = (qx[ 79: 72] * qy[ 79: 72]) >> SAR[5:0]
qz[ 87: 80] = (qx[ 87: 80] * qy[ 87: 80]) >> SAR[5:0]
qz[ 95: 88] = (qx[ 95: 88] * qy[ 95: 88]) >> SAR[5:0]
qz[103: 96] = (qx[103: 96] * qy[103: 96]) >> SAR[5:0]
qz[111:104] = (qx[111:104] * qy[111:104]) >> SAR[5:0]
qz[119:112] = (qx[119:112] * qy[119:112]) >> SAR[5:0]
qz[127:120] = (qx[127:120] * qy[127:120]) >> SAR[5:0]
```

1.8.126 EE.VMUL.S8.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	100	qu[0]	qz[2:0]	qx[1:0]	qy[2:1]	1100	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMUL.S8.LD.INCP qu, as, qz, qx, qy

Description

该指令执行数据位宽为 8-bit 的有符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 16 个 16-bit 数据结果进行算术右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 8-bit 数据，写入 qz 寄存器对应的数据段。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qz[  7:  0] = (qx[  7:  0] * qy[  7:  0]) >> SAR[5:0]
qz[ 15:  8] = (qx[ 15:  8] * qy[ 15:  8]) >> SAR[5:0]
qz[ 23: 16] = (qx[ 23: 16] * qy[ 23: 16]) >> SAR[5:0]
qz[ 31: 24] = (qx[ 31: 24] * qy[ 31: 24]) >> SAR[5:0]
qz[ 39: 32] = (qx[ 39: 32] * qy[ 39: 32]) >> SAR[5:0]
qz[ 47: 40] = (qx[ 47: 40] * qy[ 47: 40]) >> SAR[5:0]
qz[ 55: 48] = (qx[ 55: 48] * qy[ 55: 48]) >> SAR[5:0]
qz[ 63: 56] = (qx[ 63: 56] * qy[ 63: 56]) >> SAR[5:0]
qz[ 71: 64] = (qx[ 71: 64] * qy[ 71: 64]) >> SAR[5:0]
qz[ 79: 72] = (qx[ 79: 72] * qy[ 79: 72]) >> SAR[5:0]
qz[ 87: 80] = (qx[ 87: 80] * qy[ 87: 80]) >> SAR[5:0]
qz[ 95: 88] = (qx[ 95: 88] * qy[ 95: 88]) >> SAR[5:0]
qz[103: 96] = (qx[103: 96] * qy[103: 96]) >> SAR[5:0]
qz[111:104] = (qx[111:104] * qy[111:104]) >> SAR[5:0]
qz[119:112] = (qx[119:112] * qy[119:112]) >> SAR[5:0]
qz[127:120] = (qx[127:120] * qy[127:120]) >> SAR[5:0]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.127 EE.VMUL.S8.ST.INCP

Instruction Word

11100101	qy[0]	qv[2:0]	0	qz[2:0]	qx[1:0]	qy[2:1]	0011	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMUL.S8.ST.INCP qv, as, qz, qx, qy

Description

该指令执行数据位宽为 8-bit 的有符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 16 个 16-bit 数据结果进行算术右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 8-bit 数据，写入 qz 寄存器对应的数据段。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qz[  7:  0] = (qx[  7:  0] * qy[  7:  0]) >> SAR[5:0]
qz[ 15:  8] = (qx[ 15:  8] * qy[ 15:  8]) >> SAR[5:0]
qz[ 23: 16] = (qx[ 23: 16] * qy[ 23: 16]) >> SAR[5:0]
qz[ 31: 24] = (qx[ 31: 24] * qy[ 31: 24]) >> SAR[5:0]
qz[ 39: 32] = (qx[ 39: 32] * qy[ 39: 32]) >> SAR[5:0]
qz[ 47: 40] = (qx[ 47: 40] * qy[ 47: 40]) >> SAR[5:0]
qz[ 55: 48] = (qx[ 55: 48] * qy[ 55: 48]) >> SAR[5:0]
qz[ 63: 56] = (qx[ 63: 56] * qy[ 63: 56]) >> SAR[5:0]
qz[ 71: 64] = (qx[ 71: 64] * qy[ 71: 64]) >> SAR[5:0]
qz[ 79: 72] = (qx[ 79: 72] * qy[ 79: 72]) >> SAR[5:0]
qz[ 87: 80] = (qx[ 87: 80] * qy[ 87: 80]) >> SAR[5:0]
qz[ 95: 88] = (qx[ 95: 88] * qy[ 95: 88]) >> SAR[5:0]
qz[103: 96] = (qx[103: 96] * qy[103: 96]) >> SAR[5:0]
qz[111:104] = (qx[111:104] * qy[111:104]) >> SAR[5:0]
qz[119:112] = (qx[119:112] * qy[119:112]) >> SAR[5:0]
qz[127:120] = (qx[127:120] * qy[127:120]) >> SAR[5:0]

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.128 EE.VMUL.U16

Instruction Word

10	qz[2:1]	1110	qz[0]	qy[2]	1	qy[1:0]	qx[2:0]	10100100

Assembler Syntax

EE.VMUL.U16 qz, qx, qy

Description

该指令执行数据位宽为 16-bit 的无符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 8 个 32-bit 数据结果进行逻辑右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 16-bit 数据，写入 qz 寄存器对应的数据段。

Operation

```
qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0]) >> SAR[5:0]
qz[ 31: 16] = (qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32]) >> SAR[5:0]
qz[ 63: 48] = (qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64]) >> SAR[5:0]
qz[ 95: 80] = (qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
qz[111: 96] = (qx[111: 96] * qy[111: 96]) >> SAR[5:0]
qz[127:112] = (qx[127:112] * qy[127:112]) >> SAR[5:0]
```

1.8.129 EE.VMUL.U16.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	101	qu[0]	qz[2:0]	qx[1:0]	qy[2:1]	1100	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMUL.U16.LD.INCP qu, as, qz, qx, qy

Description

该指令执行数据位宽为 16-bit 的无符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 8 个 32-bit 数据结果进行逻辑右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 16-bit 数据，写入 qz 寄存器对应的数据段。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0]) >> SAR[5:0]
qz[ 31: 16] = (qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32]) >> SAR[5:0]
qz[ 63: 48] = (qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64]) >> SAR[5:0]
qz[ 95: 80] = (qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
qz[111: 96] = (qx[111: 96] * qy[111: 96]) >> SAR[5:0]
qz[127:112] = (qx[127:112] * qy[127:112]) >> SAR[5:0]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.130 EE.VMUL.U16.ST.INCP

Instruction Word

11100101	qy[0]	qv[2:0]	1	qz[2:0]	qx[1:0]	qy[2:1]	0011	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMUL.U16.ST.INCP qv, as, qz, qx, qy

Description

该指令执行数据位宽为 16-bit 的无符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 8 个 32-bit 数据结果进行逻辑右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 16-bit 数据，写入 qz 寄存器对应的数据段。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qz[ 15:  0] = (qx[ 15:  0] * qy[ 15:  0]) >> SAR[5:0]
qz[ 31: 16] = (qx[ 31: 16] * qy[ 31: 16]) >> SAR[5:0]
qz[ 47: 32] = (qx[ 47: 32] * qy[ 47: 32]) >> SAR[5:0]
qz[ 63: 48] = (qx[ 63: 48] * qy[ 63: 48]) >> SAR[5:0]
qz[ 79: 64] = (qx[ 79: 64] * qy[ 79: 64]) >> SAR[5:0]
qz[ 95: 80] = (qx[ 95: 80] * qy[ 95: 80]) >> SAR[5:0]
qz[111: 96] = (qx[111: 96] * qy[111: 96]) >> SAR[5:0]
qz[127:112] = (qx[127:112] * qy[127:112]) >> SAR[5:0]

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.131 EE.VMUL.U8

Instruction Word

10	qz[2:1]	1110	qz[0]	qy[2]	1	qy[1:0]	qx[2:0]	10110100

Assembler Syntax

EE.VMUL.U8 qz, qx, qy

Description

该指令执行数据位宽为 8-bit 的无符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 16 个 16-bit 数据结果进行逻辑右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 8-bit 数据，写入 qz 寄存器对应的数据段。

Operation

```
qz[  7:  0] = (qx[  7:  0] * qy[  7:  0]) >> SAR[5:0]
qz[ 15:  8] = (qx[ 15:  8] * qy[ 15:  8]) >> SAR[5:0]
qz[ 23: 16] = (qx[ 23: 16] * qy[ 23: 16]) >> SAR[5:0]
qz[ 31: 24] = (qx[ 31: 24] * qy[ 31: 24]) >> SAR[5:0]
qz[ 39: 32] = (qx[ 39: 32] * qy[ 39: 32]) >> SAR[5:0]
qz[ 47: 40] = (qx[ 47: 40] * qy[ 47: 40]) >> SAR[5:0]
qz[ 55: 48] = (qx[ 55: 48] * qy[ 55: 48]) >> SAR[5:0]
qz[ 63: 56] = (qx[ 63: 56] * qy[ 63: 56]) >> SAR[5:0]
qz[ 71: 64] = (qx[ 71: 64] * qy[ 71: 64]) >> SAR[5:0]
qz[ 79: 72] = (qx[ 79: 72] * qy[ 79: 72]) >> SAR[5:0]
qz[ 87: 80] = (qx[ 87: 80] * qy[ 87: 80]) >> SAR[5:0]
qz[ 95: 88] = (qx[ 95: 88] * qy[ 95: 88]) >> SAR[5:0]
qz[103: 96] = (qx[103: 96] * qy[103: 96]) >> SAR[5:0]
qz[111:104] = (qx[111:104] * qy[111:104]) >> SAR[5:0]
qz[119:112] = (qx[119:112] * qy[119:112]) >> SAR[5:0]
qz[127:120] = (qx[127:120] * qy[127:120]) >> SAR[5:0]
```

1.8.132 EE.VMUL.U8.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	110	qu[0]	qz[2:0]	qx[1:0]	qy[2:1]	1100	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMUL.U8.LD.INCP qu, as, qz, qx, qy

Description

该指令执行数据位宽为 8-bit 的无符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 16 个 32-bit 数据结果进行逻辑右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 8-bit 数据，写入 qz 寄存器对应的数据段。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qz[  7:  0] = (qx[  7:  0] * qy[  7:  0]) >> SAR[5:0]
qz[ 15:  8] = (qx[ 15:  8] * qy[ 15:  8]) >> SAR[5:0]
qz[ 23: 16] = (qx[ 23: 16] * qy[ 23: 16]) >> SAR[5:0]
qz[ 31: 24] = (qx[ 31: 24] * qy[ 31: 24]) >> SAR[5:0]
qz[ 39: 32] = (qx[ 39: 32] * qy[ 39: 32]) >> SAR[5:0]
qz[ 47: 40] = (qx[ 47: 40] * qy[ 47: 40]) >> SAR[5:0]
qz[ 55: 48] = (qx[ 55: 48] * qy[ 55: 48]) >> SAR[5:0]
qz[ 63: 56] = (qx[ 63: 56] * qy[ 63: 56]) >> SAR[5:0]
qz[ 71: 64] = (qx[ 71: 64] * qy[ 71: 64]) >> SAR[5:0]
qz[ 79: 72] = (qx[ 79: 72] * qy[ 79: 72]) >> SAR[5:0]
qz[ 87: 80] = (qx[ 87: 80] * qy[ 87: 80]) >> SAR[5:0]
qz[ 95: 88] = (qx[ 95: 88] * qy[ 95: 88]) >> SAR[5:0]
qz[103: 96] = (qx[103: 96] * qy[103: 96]) >> SAR[5:0]
qz[111:104] = (qx[111:104] * qy[111:104]) >> SAR[5:0]
qz[119:112] = (qx[119:112] * qy[119:112]) >> SAR[5:0]
qz[127:120] = (qx[127:120] * qy[127:120]) >> SAR[5:0]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.133 EE.VMUL.U8.ST.INCP

Instruction Word

11101000	qy[0]	qv[2:0]	1	qz[2:0]	qx[1:0]	qy[2:1]	0000	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMUL.U8.ST.INCP qv, as, qz, qx, qy

Description

该指令执行数据位宽为 8-bit 的无符号数向量乘法。qx、qy 寄存器分别是乘数和被乘数，对计算得到 16 个 16-bit 数据结果进行逻辑右移，特殊寄存器 SAR 内存储右移数值大小。随后截取移位后的结果的低 8-bit 数据，写入 qz 寄存器对应的数据段。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qz[  7:  0] = (qx[  7:  0] * qy[  7:  0]) >> SAR[5:0]
qz[ 15:  8] = (qx[ 15:  8] * qy[ 15:  8]) >> SAR[5:0]
qz[ 23: 16] = (qx[ 23: 16] * qy[ 23: 16]) >> SAR[5:0]
qz[ 31: 24] = (qx[ 31: 24] * qy[ 31: 24]) >> SAR[5:0]
qz[ 39: 32] = (qx[ 39: 32] * qy[ 39: 32]) >> SAR[5:0]
qz[ 47: 40] = (qx[ 47: 40] * qy[ 47: 40]) >> SAR[5:0]
qz[ 55: 48] = (qx[ 55: 48] * qy[ 55: 48]) >> SAR[5:0]
qz[ 63: 56] = (qx[ 63: 56] * qy[ 63: 56]) >> SAR[5:0]
qz[ 71: 64] = (qx[ 71: 64] * qy[ 71: 64]) >> SAR[5:0]
qz[ 79: 72] = (qx[ 79: 72] * qy[ 79: 72]) >> SAR[5:0]
qz[ 87: 80] = (qx[ 87: 80] * qy[ 87: 80]) >> SAR[5:0]
qz[ 95: 88] = (qx[ 95: 88] * qy[ 95: 88]) >> SAR[5:0]
qz[103: 96] = (qx[103: 96] * qy[103: 96]) >> SAR[5:0]
qz[111:104] = (qx[111:104] * qy[111:104]) >> SAR[5:0]
qz[119:112] = (qx[119:112] * qy[119:112]) >> SAR[5:0]
qz[127:120] = (qx[127:120] * qy[127:120]) >> SAR[5:0]

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.134 EE.VMULAS.S16.ACCX

Instruction Word

000110100	qy[2]	0	qy[1:0]	qx[2:0]	10000100

Assembler Syntax

EE.VMULAS.S16.ACCX qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后分别计算出 8 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

Operation

```
add0[31:0] = qx[ 15:  0] * qy[ 15:  0]
add1[31:0] = qx[ 31: 16] * qy[ 31: 16]
...
add7[31:0] = qx[127:112] * qy[127:112]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[31:0] + ad[31:0]d1[31:0] + ... + ad[31:0]d
    7[31:0]

ACCX[39:0] = min(max(sum[40:0], -2^{39}), 2^{39}-1)
```

1.8.135 EE.VMULAS.S16.ACCX.LD.IP

Instruction Word

1111	imm16[5:4]	qu[2:1]	qy[0]	000	qu[0]	000	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S16.ACCX.LD.IP qu, as, -512..496, qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后分别计算出 8 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

Operation

```
add0[31:0] = qx[ 15:  0] * qy[ 15:  0]
add1[31:0] = qx[ 31: 16] * qy[ 31: 16]
...
add7[31:0] = qx[127:112] * qy[127:112]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[31:0] + ad[31:0]d1[31:0] + ... + ad[31:0]d
    7[31:0]
ACCX[39:0] = min(max(sum[40:0], -2^{39}), 2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.136 EE.VMULAS.S16.ACCX.LD.IP.QUP

Instruction Word

0000	imm16[5:4]	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S16.ACCX.LD.IP.QUP qu, as, imm16, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后分别计算出 8 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
add0[31:0] = qx[ 15:  0] * qy[ 15:  0]
add1[31:0] = qx[ 31: 16] * qy[ 31: 16]
...
add7[31:0] = qx[127:112] * qy[127:112]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[31:0] + ad[31:0]d1[31:0] + ... + ad[31:0]d
    7[31:0]
ACCX[39:0] = min(max(sum[40:0], -2^{39}), 2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.137 EE.VMULAS.S16.ACCX.LD.XP

Instruction Word

111100	qu[2:1]	qy[0]	001	qu[0]	000	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S16.ACCX.LD.XP qu, as, ad, qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后分别计算出 8 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
add0[31:0] = qx[ 15:  0] * qy[ 15:  0]
add1[31:0] = qx[ 31: 16] * qy[ 31: 16]
...
add7[31:0] = qx[127:112] * qy[127:112]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[31:0] + ad[31:0]d1[31:0] + ... + ad[31:0]d
    7[31:0]
ACCX[39:0] = min(max(sum[40:0], -2^{39}), 2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.138 EE.VMULAS.S16.ACCX.LD.XP.QUP

Instruction Word

101100	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S16.ACCX.LD.XP.QUP qu, as, ad, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后分别计算出 8 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
add0[31:0] = qx[ 15:  0] * qy[ 15:  0]
add1[31:0] = qx[ 31: 16] * qy[ 31: 16]
...
add7[31:0] = qx[127:112] * qy[127:112]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[31:0] + ad[31:0]d1[31:0] + ... + ad[31:0]d
    7[31:0]
ACCX[39:0] = min(max(sum[40:0], -2^{39}), 2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
qs0[127:0] = {qs1[127: 0], qs0[127: 0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.139 EE.VMULAS.S16.QACC

Instruction Word

000110100	qy[2]	1	qy[1:0]	qx[2:0]	10000100

Assembler Syntax

EE.VMULAS.S16.QACC qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

Operation

```
QACC_L[ 39:  0] = min(max(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], -2^{39}),
    2^{39}-1)
QACC_L[ 79: 40] = min(max(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], -2^{39}),
    2^{39}-1)
QACC_L[119: 80] = min(max(QACC_L[119: 80] + qx[ 47: 32] * qy[ 47: 32], -2^{39}),
    2^{39}-1)
QACC_L[159:120] = min(max(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], -2^{39}),
    2^{39}-1)
QACC_H[ 39:  0] = min(max(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], -2^{39}),
    2^{39}-1)
QACC_H[ 79: 40] = min(max(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], -2^{39}),
    2^{39}-1)
QACC_H[119: 80] = min(max(QACC_H[119: 80] + qx[111: 96] * qy[111: 96], -2^{39}),
    2^{39}-1)
QACC_H[159:120] = min(max(QACC_H[159:120] + qx[127:112] * qy[127:112], -2^{39}),
    2^{39}-1)
```

1.8.140 EE.VMULAS.S16.QACC.LD.IP

Instruction Word

1111	imm16[5:4]	qu[2:1]	qy[0]	000	qu[0]	001	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S16.QACC.LD.IP qu, as, imm16, qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

Operation

```
QACC_L[ 39:  0] = min(max(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], -2^{39}),
    2^{39}-1)
QACC_L[ 79: 40] = min(max(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], -2^{39}),
    2^{39}-1)
QACC_L[119: 80] = min(max(QACC_L[119: 80] + qx[ 47: 32] * qy[ 47: 32], -2^{39}),
    2^{39}-1)
QACC_L[159:120] = min(max(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], -2^{39}),
    2^{39}-1)
QACC_H[ 39:  0] = min(max(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], -2^{39}),
    2^{39}-1)
QACC_H[ 79: 40] = min(max(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], -2^{39}),
    2^{39}-1)
QACC_H[119: 80] = min(max(QACC_H[119: 80] + qx[111: 96] * qy[111: 96], -2^{39}),
    2^{39}-1)
QACC_H[159:120] = min(max(QACC_H[159:120] + qx[127:112] * qy[127:112], -2^{39}),
    2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {22{imm16[5]},imm16[5:0],4{0}}
```

1.8.141 EE.VMULAS.S16.QACC.LD.IP.QUP

Instruction Word

0001	imm16[5:4]	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S16.QACC.LD.IP.QUP qu, as, imm16, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 39:  0] = min(max(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], -2^{39}),
    2^{39}-1)
QACC_L[ 79: 40] = min(max(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], -2^{39}),
    2^{39}-1)
QACC_L[119: 80] = min(max(QACC_L[119: 80] + qx[ 47: 32] * qy[ 47: 32], -2^{39}),
    2^{39}-1)
QACC_L[159:120] = min(max(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], -2^{39}),
    2^{39}-1)
QACC_H[ 39:  0] = min(max(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], -2^{39}),
    2^{39}-1)
QACC_H[ 79: 40] = min(max(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], -2^{39}),
    2^{39}-1)
QACC_H[119: 80] = min(max(QACC_H[119: 80] + qx[111: 96] * qy[111: 96], -2^{39}),
    2^{39}-1)
QACC_H[159:120] = min(max(QACC_H[159:120] + qx[127:112] * qy[127:112], -2^{39}),
    2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {22{imm16[5]},imm16[5:0],4{0}}
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.142 EE.VMULAS.S16.QACC.LD.XP

Instruction Word

111100	qu[2:1]	qy[0]	001	qu[0]	001	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S16.QACC.LD.XP qu, as, ad, qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
QACC_L[ 39:  0] = min(max(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], -2^{39}), 2^{39}-1)
QACC_L[ 79: 40] = min(max(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], -2^{39}), 2^{39}-1)
QACC_L[119: 80] = min(max(QACC_L[119: 80] + qx[ 47: 32] * qy[ 47: 32], -2^{39}), 2^{39}-1)
QACC_L[159:120] = min(max(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], -2^{39}), 2^{39}-1)
QACC_H[ 39:  0] = min(max(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], -2^{39}), 2^{39}-1)
QACC_H[ 79: 40] = min(max(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], -2^{39}), 2^{39}-1)
QACC_H[119: 80] = min(max(QACC_H[119: 80] + qx[111: 96] * qy[111: 96], -2^{39}), 2^{39}-1)
QACC_H[159:120] = min(max(QACC_H[159:120] + qx[127:112] * qy[127:112], -2^{39}), 2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.143 EE.VMULAS.S16.QACC.LD.XP.QUP

Instruction Word

101101	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S16.QACC.LD.XP.QUP qu, as, ad, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 39:  0] = min(max(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], -2^{39}), 2^{39}-1)
QACC_L[ 79: 40] = min(max(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], -2^{39}), 2^{39}-1)
QACC_L[119: 80] = min(max(QACC_L[119: 80] + qx[ 47: 32] * qy[ 47: 32], -2^{39}), 2^{39}-1)
QACC_L[159:120] = min(max(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], -2^{39}), 2^{39}-1)
QACC_H[ 39:  0] = min(max(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], -2^{39}), 2^{39}-1)
QACC_H[ 79: 40] = min(max(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], -2^{39}), 2^{39}-1)
QACC_H[119: 80] = min(max(QACC_H[119: 80] + qx[111: 96] * qy[111: 96], -2^{39}), 2^{39}-1)
QACC_H[159:120] = min(max(QACC_H[159:120] + qx[127:112] * qy[127:112], -2^{39}), 2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.144 EE.VMULAS.S16.QACC.LDBC.INCP

Instruction Word

100	qu[2]	0111	qu[1]	qy[2]	qu[0]	qy[1:0]	qx[2:0]	as[3:0]	0100

Assembler Syntax

EE.VMULAS.S16.QACC.LDBC.INCP qu, as, qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 1 比特强制为 0，随后从内存中载入 16-bit 数据，并广播到 qu 寄存器中 8 个 16-bit 数据段内。访存结束后，as 寄存器内数值增加 2。

Operation

```
QACC_L[ 39:  0] = min(max(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], -2^{39}), 2^{39}-1)
QACC_L[ 79: 40] = min(max(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], -2^{39}), 2^{39}-1)
QACC_L[119: 80] = min(max(QACC_L[119: 80] + qx[ 47: 32] * qy[ 47: 32], -2^{39}), 2^{39}-1)
QACC_L[159:120] = min(max(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], -2^{39}), 2^{39}-1)
QACC_H[ 39:  0] = min(max(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], -2^{39}), 2^{39}-1)
QACC_H[ 79: 40] = min(max(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], -2^{39}), 2^{39}-1)
QACC_H[119: 80] = min(max(QACC_H[119: 80] + qx[111: 96] * qy[111: 96], -2^{39}), 2^{39}-1)
QACC_H[159:120] = min(max(QACC_H[159:120] + qx[127:112] * qy[127:112], -2^{39}), 2^{39}-1)

qu[127:0] = {8{load16({as[31:1],1{0}})}}
as[31:0] = as[31:0] + 2
```

1.8.145 EE.VMULAS.S16.QACC.LDBC.INCP.QUP

Instruction Word

111000	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	1000	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S16.QACC.LDBC.INCP.QUP qu, as, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 1 比特强制为 0，随后从内存中载入 16-bit 数据，并广播到 qu 寄存器中 8 个 16-bit 数据段内。访存结束后，as 寄存器内数值增加 2。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 39:  0] = min(max(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], -2^{39}), 2^{39}-1)
QACC_L[ 79: 40] = min(max(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], -2^{39}), 2^{39}-1)
QACC_L[119: 80] = min(max(QACC_L[119: 80] + qx[ 47: 32] * qy[ 47: 32], -2^{39}), 2^{39}-1)
QACC_L[159:120] = min(max(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], -2^{39}), 2^{39}-1)
QACC_H[ 39:  0] = min(max(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], -2^{39}), 2^{39}-1)
QACC_H[ 79: 40] = min(max(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], -2^{39}), 2^{39}-1)
QACC_H[119: 80] = min(max(QACC_H[119: 80] + qx[111: 96] * qy[111: 96], -2^{39}), 2^{39}-1)
QACC_H[159:120] = min(max(QACC_H[159:120] + qx[127:112] * qy[127:112], -2^{39}), 2^{39}-1)

qu[127:0] = {8{load16({as[31:1],1{0}})}}
as[31:0] = as[31:0] + 2
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.146 EE.VMULAS.S8.ACCX

Instruction Word

000110100	qy[2]	0	qy[1:0]	qx[2:0]	11000100

Assembler Syntax

EE.VMULAS.S8.ACCX qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后分别计算出 16 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

Operation

```
add0[15:0] = qx[  7:  0] * qy[  7:  0]
add1[15:0] = qx[ 15:  8] * qy[ 15:  8]
...
add15[15:0] = qx[127:120] * qy[127:120]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[15:0] + ad[31:0]d1[15:0] + ... + ad[31:0]d
    15[15:0]

ACCX[39:0] = min(max(sum[40:0], -2^{39}), 2^{39}-1)
```

1.8.147 EE.VMULAS.S8.ACCX.LD.IP

Instruction Word

1111	imm16[5:4]	qu[2:1]	qy[0]	000	qu[0]	010	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S8.ACCX.LD.IP qu, as, -512..496, qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后分别计算出 16 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

Operation

```
add0[15:0] = qx[  7:  0] * qy[  7:  0]
add1[15:0] = qx[ 15:  8] * qy[ 15:  8]
...
add15[15:0] = qx[127:120] * qy[127:120]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[15:0] + ad[31:0]d1[15:0] + ... + ad[31:0]d
    15[15:0]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.148 EE.VMULAS.S8.ACCX.LD.IP.QUP

Instruction Word

0010	imm16[5:4]	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S8.ACCX.LD.IP.QUP qu, as, imm16, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后分别计算出 16 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
add0[15:0] = qx[  7:  0] * qy[  7:  0]
add1[15:0] = qx[ 15:  8] * qy[ 15:  8]
...
add15[15:0] = qx[127:120] * qy[127:120]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[15:0] + ad[31:0]d1[15:0] + ... + ad[31:0]d
    15[15:0]

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.149 EE.VMULAS.S8.ACCX.LD.XP

Instruction Word

111100	qu[2:1]	qy[0]	001	qu[0]	010	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S8.ACCX.LD.XP qu, as, ad, qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后分别计算出 16 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
add0[15:0] = qx[  7:  0] * qy[  7:  0]
add1[15:0] = qx[ 15:  8] * qy[ 15:  8]
...
add15[15:0] = qx[127:120] * qy[127:120]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[15:0] + ad[31:0]d1[15:0] + ... + ad[31:0]d
    15[15:0]
ACCX[39:0] = min(max(sum[40:0], -2^{39}), 2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.150 EE.VMULAS.S8.ACCX.LD.XP.QUP

Instruction Word

101110	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S8.ACCX.LD.XP.QUP qu, as, ad, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后分别计算出 16 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
add0[15:0] = qx[  7:  0] * qy[  7:  0]
add1[15:0] = qx[ 15:  8] * qy[ 15:  8]
...
add15[15:0] = qx[127:120] * qy[127:120]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[15:0] + ad[31:0]d1[15:0] + ... + ad[31:0]d
    15[15:0]
ACCX[39:0] = min(max(sum[40:0], -2^{39}), 2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
qs0[127:0] = {qs1[127: 0], qs0[127: 0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.151 EE.VMULAS.S8.QACC

Instruction Word

000110100	qy[2]	1	qy[1:0]	qx[2:0]	11000100

Assembler Syntax

EE.VMULAS.S8.QACC qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

Operation

```
QACC_L[ 19:  0] = min(max(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], -2^{19}), 2^{19}-1)
QACC_L[ 39: 20] = min(max(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], -2^{19}), 2^{19}-1)
QACC_L[ 59: 40] = min(max(QACC_L[ 59: 40] + qx[ 23: 16] * qy[ 23: 16], -2^{19}), 2^{19}-1)
QACC_L[ 79: 60] = min(max(QACC_L[ 79: 60] + qx[ 31: 24] * qy[ 31: 24], -2^{19}), 2^{19}-1)
QACC_L[ 99: 80] = min(max(QACC_L[ 99: 80] + qx[ 39: 32] * qy[ 39: 32], -2^{19}), 2^{19}-1)
QACC_L[119:100] = min(max(QACC_L[119:100] + qx[ 47: 40] * qy[ 47: 40], -2^{19}), 2^{19}-1)
QACC_L[139:120] = min(max(QACC_L[139:120] + qx[ 55: 48] * qy[ 55: 48], -2^{19}), 2^{19}-1)
QACC_L[159:140] = min(max(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], -2^{19}), 2^{19}-1)
QACC_H[ 19:  0] = min(max(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], -2^{19}), 2^{19}-1)
QACC_H[ 39: 20] = min(max(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], -2^{19}), 2^{19}-1)
QACC_H[ 59: 40] = min(max(QACC_H[ 59: 40] + qx[ 87: 80] * qy[ 87: 80], -2^{19}), 2^{19}-1)
QACC_H[ 79: 60] = min(max(QACC_H[ 79: 60] + qx[ 95: 88] * qy[ 95: 88], -2^{19}), 2^{19}-1)
QACC_H[ 99: 80] = min(max(QACC_H[ 99: 80] + qx[103: 96] * qy[103: 96], -2^{19}), 2^{19}-1)
QACC_H[119:100] = min(max(QACC_H[119:100] + qx[111:104] * qy[111:104], -2^{19}), 2^{19}-1)
QACC_H[139:120] = min(max(QACC_H[139:120] + qx[119:112] * qy[119:112], -2^{19}), 2^{19}-1)
QACC_H[159:140] = min(max(QACC_H[159:140] + qx[127:120] * qy[127:120], -2^{19}), 2^{19}-1)
```

1.8.152 EE.VMULAS.S8.QACC.LD.IP

Instruction Word

1111	imm16[5:4]	qu[2:1]	qy[0]	000	qu[0]	011	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

Assembler Syntax

EE.VMULAS.S8.QACC.LD.IP qu, as, imm16, qx, qy

Description

Operation

```
QACC_L[ 19:  0] = min(max(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], -2^{19}),
    2^{19}-1)
QACC_L[ 39: 20] = min(max(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], -2^{19}),
    2^{19}-1)
QACC_L[ 59: 40] = min(max(QACC_L[ 59: 40] + qx[ 23: 16] * qy[ 23: 16], -2^{19}),
    2^{19}-1)
QACC_L[ 79: 60] = min(max(QACC_L[ 79: 60] + qx[ 31: 24] * qy[ 31: 24], -2^{19}),
    2^{19}-1)
QACC_L[ 99: 80] = min(max(QACC_L[ 99: 80] + qx[ 39: 32] * qy[ 39: 32], -2^{19}),
    2^{19}-1)
QACC_L[119:100] = min(max(QACC_L[119:100] + qx[ 47: 40] * qy[ 47: 40], -2^{19}),
    2^{19}-1)
QACC_L[139:120] = min(max(QACC_L[139:120] + qx[ 55: 48] * qy[ 55: 48], -2^{19}),
    2^{19}-1)
QACC_L[159:140] = min(max(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], -2^{19}),
    2^{19}-1)
QACC_H[ 19:  0] = min(max(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], -2^{19}),
    2^{19}-1)
QACC_H[ 39: 20] = min(max(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], -2^{19}),
    2^{19}-1)
QACC_H[ 59: 40] = min(max(QACC_H[ 59: 40] + qx[ 87: 80] * qy[ 87: 80], -2^{19}),
    2^{19}-1)
QACC_H[ 79: 60] = min(max(QACC_H[ 79: 60] + qx[ 95: 88] * qy[ 95: 88], -2^{19}),
    2^{19}-1)
QACC_H[ 99: 80] = min(max(QACC_H[ 99: 80] + qx[103: 96] * qy[103: 96], -2^{19}),
    2^{19}-1)
QACC_H[119:100] = min(max(QACC_H[119:100] + qx[111:104] * qy[111:104], -2^{19}),
    2^{19}-1)
QACC_H[139:120] = min(max(QACC_H[139:120] + qx[119:112] * qy[119:112], -2^{19}),
    2^{19}-1)
QACC_H[159:140] = min(max(QACC_H[159:140] + qx[127:120] * qy[127:120], -2^{19}),
    2^{19}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {22{imm16[5]},imm16[5:0],4{0}}
```

1.8.153 EE.VMULAS.S8.QACC.LD.IP.QUP

Instruction Word

0011	imm16[5:4]	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S8.QACC.LD.IP.QUP qu, as, imm16, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 19:  0] = min(max(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], -2^{19}), 2^{19}-1)
QACC_L[ 39: 20] = min(max(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], -2^{19}), 2^{19}-1)
QACC_L[ 59: 40] = min(max(QACC_L[ 59: 40] + qx[ 23: 16] * qy[ 23: 16], -2^{19}), 2^{19}-1)
QACC_L[ 79: 60] = min(max(QACC_L[ 79: 60] + qx[ 31: 24] * qy[ 31: 24], -2^{19}), 2^{19}-1)
QACC_L[ 99: 80] = min(max(QACC_L[ 99: 80] + qx[ 39: 32] * qy[ 39: 32], -2^{19}), 2^{19}-1)
QACC_L[119:100] = min(max(QACC_L[119:100] + qx[ 47: 40] * qy[ 47: 40], -2^{19}), 2^{19}-1)
QACC_L[139:120] = min(max(QACC_L[139:120] + qx[ 55: 48] * qy[ 55: 48], -2^{19}), 2^{19}-1)
QACC_L[159:140] = min(max(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], -2^{19}), 2^{19}-1)
QACC_H[ 19:  0] = min(max(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], -2^{19}), 2^{19}-1)
QACC_H[ 39: 20] = min(max(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], -2^{19}), 2^{19}-1)
QACC_H[ 59: 40] = min(max(QACC_H[ 59: 40] + qx[ 87: 80] * qy[ 87: 80], -2^{19}), 2^{19}-1)
QACC_H[ 79: 60] = min(max(QACC_H[ 79: 60] + qx[ 95: 88] * qy[ 95: 88], -2^{19}), 2^{19}-1)
QACC_H[ 99: 80] = min(max(QACC_H[ 99: 80] + qx[103: 96] * qy[103: 96], -2^{19}), 2^{19}-1)
QACC_H[119:100] = min(max(QACC_H[119:100] + qx[111:104] * qy[111:104], -2^{19}), 2^{19}-1)
QACC_H[139:120] = min(max(QACC_H[139:120] + qx[119:112] * qy[119:112], -2^{19}), 2^{19}-1)
QACC_H[159:140] = min(max(QACC_H[159:140] + qx[127:120] * qy[127:120], -2^{19}), 2^{19}-1)

```

```
qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {22{imm16[5]},imm16[5:0],4{0}}
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.154 EE.VMULAS.S8.QACC.LD.XP

Instruction Word

111100	qu[2:1]	qy[0]	001	qu[0]	011	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S8.QACC.LD.XP qu, as, ad, qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
QACC_L[ 19:  0] = min(max(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], -2^{19}), 2^{19}-1)
QACC_L[ 39: 20] = min(max(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], -2^{19}), 2^{19}-1)
QACC_L[ 59: 40] = min(max(QACC_L[ 59: 40] + qx[ 23: 16] * qy[ 23: 16], -2^{19}), 2^{19}-1)
QACC_L[ 79: 60] = min(max(QACC_L[ 79: 60] + qx[ 31: 24] * qy[ 31: 24], -2^{19}), 2^{19}-1)
QACC_L[ 99: 80] = min(max(QACC_L[ 99: 80] + qx[ 39: 32] * qy[ 39: 32], -2^{19}), 2^{19}-1)
QACC_L[119:100] = min(max(QACC_L[119:100] + qx[ 47: 40] * qy[ 47: 40], -2^{19}), 2^{19}-1)
QACC_L[139:120] = min(max(QACC_L[139:120] + qx[ 55: 48] * qy[ 55: 48], -2^{19}), 2^{19}-1)
QACC_L[159:140] = min(max(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], -2^{19}), 2^{19}-1)
QACC_H[ 19:  0] = min(max(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], -2^{19}), 2^{19}-1)
QACC_H[ 39: 20] = min(max(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], -2^{19}), 2^{19}-1)
QACC_H[ 59: 40] = min(max(QACC_H[ 59: 40] + qx[ 87: 80] * qy[ 87: 80], -2^{19}), 2^{19}-1)
QACC_H[ 79: 60] = min(max(QACC_H[ 79: 60] + qx[ 95: 88] * qy[ 95: 88], -2^{19}), 2^{19}-1)
QACC_H[ 99: 80] = min(max(QACC_H[ 99: 80] + qx[103: 96] * qy[103: 96], -2^{19}), 2^{19}-1)
QACC_H[119:100] = min(max(QACC_H[119:100] + qx[111:104] * qy[111:104], -2^{19}), 2^{19}-1)
QACC_H[139:120] = min(max(QACC_H[139:120] + qx[119:112] * qy[119:112], -2^{19}), 2^{19}-1)
QACC_H[159:140] = min(max(QACC_H[159:140] + qx[127:120] * qy[127:120], -2^{19}), 2^{19}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.155 EE.VMULAS.S8.QACC.LD.XP.QUP

Instruction Word

101111	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S8.QACC.LD.XP.QUP qu, as, ad, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 19:  0] = min(max(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], -2^{19}), 2^{19}-1)
QACC_L[ 39: 20] = min(max(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], -2^{19}), 2^{19}-1)
QACC_L[ 59: 40] = min(max(QACC_L[ 59: 40] + qx[ 23: 16] * qy[ 23: 16], -2^{19}), 2^{19}-1)
QACC_L[ 79: 60] = min(max(QACC_L[ 79: 60] + qx[ 31: 24] * qy[ 31: 24], -2^{19}), 2^{19}-1)
QACC_L[ 99: 80] = min(max(QACC_L[ 99: 80] + qx[ 39: 32] * qy[ 39: 32], -2^{19}), 2^{19}-1)
QACC_L[119:100] = min(max(QACC_L[119:100] + qx[ 47: 40] * qy[ 47: 40], -2^{19}), 2^{19}-1)
QACC_L[139:120] = min(max(QACC_L[139:120] + qx[ 55: 48] * qy[ 55: 48], -2^{19}), 2^{19}-1)
QACC_L[159:140] = min(max(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], -2^{19}), 2^{19}-1)
QACC_H[ 19:  0] = min(max(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], -2^{19}), 2^{19}-1)
QACC_H[ 39: 20] = min(max(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], -2^{19}), 2^{19}-1)
QACC_H[ 59: 40] = min(max(QACC_H[ 59: 40] + qx[ 87: 80] * qy[ 87: 80], -2^{19}), 2^{19}-1)
QACC_H[ 79: 60] = min(max(QACC_H[ 79: 60] + qx[ 95: 88] * qy[ 95: 88], -2^{19}), 2^{19}-1)
QACC_H[ 99: 80] = min(max(QACC_H[ 99: 80] + qx[103: 96] * qy[103: 96], -2^{19}), 2^{19}-1)
QACC_H[119:100] = min(max(QACC_H[119:100] + qx[111:104] * qy[111:104], -2^{19}), 2^{19}-1)
QACC_H[139:120] = min(max(QACC_H[139:120] + qx[119:112] * qy[119:112], -2^{19}), 2^{19}-1)
QACC_H[159:140] = min(max(QACC_H[159:140] + qx[127:120] * qy[127:120], -2^{19}), 2^{19}-1)

```

```
qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.156 EE.VMULAS.S8.QACC.LDBC.INCP

Instruction Word

101	qu[2]	0111	qu[1]	qy[2]	qu[0]	qy[1:0]	qx[2:0]	as[3:0]	0100

Assembler Syntax

EE.VMULAS.S8.QACC.LDBC.INCP qu, as, qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，根据访存寄存器 as 中指示的地址，从内存中载入 8-bit 数据，并广播到 qu 寄存器中 16 个 8-bit 数据段内。访存结束后，as 寄存器内数值增加 1。

Operation

```
QACC_L[ 19:  0] = min(max(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], -2^{19}), 2^{19}-1)
QACC_L[ 39: 20] = min(max(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], -2^{19}), 2^{19}-1)
QACC_L[ 59: 40] = min(max(QACC_L[ 59: 40] + qx[ 23: 16] * qy[ 23: 16], -2^{19}), 2^{19}-1)
QACC_L[ 79: 60] = min(max(QACC_L[ 79: 60] + qx[ 31: 24] * qy[ 31: 24], -2^{19}), 2^{19}-1)
QACC_L[ 99: 80] = min(max(QACC_L[ 99: 80] + qx[ 39: 32] * qy[ 39: 32], -2^{19}), 2^{19}-1)
QACC_L[119:100] = min(max(QACC_L[119:100] + qx[ 47: 40] * qy[ 47: 40], -2^{19}), 2^{19}-1)
QACC_L[139:120] = min(max(QACC_L[139:120] + qx[ 55: 48] * qy[ 55: 48], -2^{19}), 2^{19}-1)
QACC_L[159:140] = min(max(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], -2^{19}), 2^{19}-1)
QACC_H[ 19:  0] = min(max(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], -2^{19}), 2^{19}-1)
QACC_H[ 39: 20] = min(max(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], -2^{19}), 2^{19}-1)
QACC_H[ 59: 40] = min(max(QACC_H[ 59: 40] + qx[ 87: 80] * qy[ 87: 80], -2^{19}), 2^{19}-1)
QACC_H[ 79: 60] = min(max(QACC_H[ 79: 60] + qx[ 95: 88] * qy[ 95: 88], -2^{19}), 2^{19}-1)
QACC_H[ 99: 80] = min(max(QACC_H[ 99: 80] + qx[103: 96] * qy[103: 96], -2^{19}), 2^{19}-1)
QACC_H[119:100] = min(max(QACC_H[119:100] + qx[111:104] * qy[111:104], -2^{19}), 2^{19}-1)
QACC_H[139:120] = min(max(QACC_H[139:120] + qx[119:112] * qy[119:112], -2^{19}), 2^{19}-1)
QACC_H[159:140] = min(max(QACC_H[159:140] + qx[127:120] * qy[127:120], -2^{19}), 2^{19}-1)

qu[127:0] = {16{load8(as[31:0])}}
as[31:0] = as[31:0] + 1
```

1.8.157 EE.VMULAS.S8.QACC.LDBC.INCP.QUP

Instruction Word

111000	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	1001	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.S8.QACC.LDBC.INCP.QUP qu, as, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的有符号乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，根据访存寄存器 as 中指示的地址，从内存中载入 8-bit 数据，并广播到 qu 寄存器中 16 个 8-bit 数据段内。访存结束后，as 寄存器内数值增加 1。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 19:  0] = min(max(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], -2^{19}), 2^{19}-1)
QACC_L[ 39: 20] = min(max(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], -2^{19}), 2^{19}-1)
QACC_L[ 59: 40] = min(max(QACC_L[ 59: 40] + qx[ 23: 16] * qy[ 23: 16], -2^{19}), 2^{19}-1)
QACC_L[ 79: 60] = min(max(QACC_L[ 79: 60] + qx[ 31: 24] * qy[ 31: 24], -2^{19}), 2^{19}-1)
QACC_L[ 99: 80] = min(max(QACC_L[ 99: 80] + qx[ 39: 32] * qy[ 39: 32], -2^{19}), 2^{19}-1)
QACC_L[119:100] = min(max(QACC_L[119:100] + qx[ 47: 40] * qy[ 47: 40], -2^{19}), 2^{19}-1)
QACC_L[139:120] = min(max(QACC_L[139:120] + qx[ 55: 48] * qy[ 55: 48], -2^{19}), 2^{19}-1)
QACC_L[159:140] = min(max(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], -2^{19}), 2^{19}-1)
QACC_H[ 19:  0] = min(max(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], -2^{19}), 2^{19}-1)
QACC_H[ 39: 20] = min(max(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], -2^{19}), 2^{19}-1)
QACC_H[ 59: 40] = min(max(QACC_H[ 59: 40] + qx[ 87: 80] * qy[ 87: 80], -2^{19}), 2^{19}-1)
QACC_H[ 79: 60] = min(max(QACC_H[ 79: 60] + qx[ 95: 88] * qy[ 95: 88], -2^{19}), 2^{19}-1)
QACC_H[ 99: 80] = min(max(QACC_H[ 99: 80] + qx[103: 96] * qy[103: 96], -2^{19}), 2^{19}-1)
QACC_H[119:100] = min(max(QACC_H[119:100] + qx[111:104] * qy[111:104], -2^{19}), 2^{19}-1)
QACC_H[139:120] = min(max(QACC_H[139:120] + qx[119:112] * qy[119:112], -2^{19}), 2^{19}-1)
QACC_H[159:140] = min(max(QACC_H[159:140] + qx[127:120] * qy[127:120], -2^{19}), 2^{19}-1)

```

```
18  qu[127:0] = {16{load8(as[31:0])}}
19  as[31:0] = as[31:0] + 1
20  qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.158 EE.VMULAS.U16.ACCX

Instruction Word

000010100	qy[2]	0	qy[1:0]	qx[2:0]	10000100

Assembler Syntax

EE.VMULAS.U16.ACCX qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后分别计算出 8 组数据的无符号数乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

Operation

```
add0[31:0] = qx[ 15:  0] * qy[ 15:  0]
add1[31:0] = qx[ 31: 16] * qy[ 31: 16]
...
add7[31:0] = qx[127:112] * qy[127:112]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[31:0] + ad[31:0]d1[31:0] + ... + ad[31:0]d
    7[31:0]

ACCX[39:0] = min(max(sum[40:0], 0), 2^{40}-1)
```

1.8.159 EE.VMULAS.U16.ACCX.LD.IP

Instruction Word

1111	imm16[5:4]	qu[2:1]	qy[0]	000	qu[0]	100	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U16.ACCX.LD.IP qu, as, -512..496, qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后分别计算出 8 组数据的无符号数乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

Operation

```
add0[31:0] = qx[ 15:  0] * qy[ 15:  0]
add1[31:0] = qx[ 31: 16] * qy[ 31: 16]
...
add7[31:0] = qx[127:112] * qy[127:112]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[31:0] + ad[31:0]d1[31:0] + ... + ad[31:0]d
    7[31:0]
ACCX[40:0] = min(max(sum[40:0], 0), 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.160 EE.VMULAS.U16.ACCX.LD.IP.QUP

Instruction Word

0100	imm16[5:4]	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U16.ACCX.LD.IP.QUP qu, as, imm16, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后分别计算出 8 组数据的有符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
add0[31:0] = qx[ 15:  0] * qy[ 15:  0]
add1[31:0] = qx[ 31: 16] * qy[ 31: 16]
...
add7[31:0] = qx[127:112] * qy[127:112]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[31:0] + ad[31:0]d1[31:0] + ... + ad[31:0]d
    7[31:0]
ACCX[40:0] = min(max(sum[40:0], 0), 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.161 EE.VMULAS.U16.ACCX.LD.XP

Instruction Word

111100	qu[2:1]	qy[0]	001	qu[0]	100	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U16.ACCX.LD.XP qu, as, ad, qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后分别计算出 8 组数据的无符号数乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
add0[31:0] = qx[ 15:  0] * qy[ 15:  0]
add1[31:0] = qx[ 31: 16] * qy[ 31: 16]
...
add7[31:0] = qx[127:112] * qy[127:112]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[31:0] + ad[31:0]d1[31:0] + ... + ad[31:0]d
    7[31:0]
ACCX[40:0] = min(max(sum[40:0], 0), 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.162 EE.VMULAS.U16.ACCX.LD.XP.QUP

Instruction Word

110000	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U16.ACCX.LD.XP.QUP qu, as, ad, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后分别计算出 8 组数据的无符号数乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
add0[31:0] = qx[ 15:  0] * qy[ 15:  0]
add1[31:0] = qx[ 31: 16] * qy[ 31: 16]
...
add7[31:0] = qx[127:112] * qy[127:112]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[31:0] + ad[31:0]d1[31:0] + ... + ad[31:0]d
    7[31:0]
ACCX[40:0] = min(max(sum[40:0], 0), 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
qs0[127:0] = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.163 EE.VMULAS.U16.QACC

Instruction Word

000010100	qy[2]	1	qy[1:0]	qx[2:0]	10000100

Assembler Syntax

EE.VMULAS.U16.QACC qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

Operation

```
QACC_L[ 39:  0] = min(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], 2^{40}-1)
QACC_L[ 79: 40] = min(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], 2^{40}-1)
...
QACC_L[159:120] = min(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], 2^{40}-1)
QACC_H[ 39:  0] = min(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], 2^{40}-1)
QACC_H[ 79: 40] = min(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], 2^{40}-1)
...
QACC_H[159:120] = min(QACC_H[159:120] + qx[127:112] * qy[127:112], 2^{40}-1)
```

1.8.164 EE.VMULAS.U16.QACC.LD.IP

Instruction Word

1111	imm16[5:4]	qu[2:1]	qy[0]	000	qu[0]	101	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U16.QACC.LD.IP qu, as, imm16, qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

Operation

```
QACC_L[ 39:  0] = min(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], 2^{40}-1)
QACC_L[ 79: 40] = min(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], 2^{40}-1)
...
QACC_L[159:120] = min(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], 2^{40}-1)
QACC_H[ 39:  0] = min(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], 2^{40}-1)
QACC_H[ 79: 40] = min(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], 2^{40}-1)
...
QACC_H[159:120] = min(QACC_H[159:120] + qx[127:112] * qy[127:112], 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {22{imm16[5]},imm16[5:0],4{0}}
```

1.8.165 EE.VMULAS.U16.QACC.LD.IP.QUP

Instruction Word

0101	imm16[5:4]	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U16.QACC.LD.IP.QUP qu, as, imm16, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 39:  0] = min(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], 2^{40}-1)
QACC_L[ 79: 40] = min(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], 2^{40}-1)
...
QACC_L[159:120] = min(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], 2^{40}-1)
QACC_H[ 39:  0] = min(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], 2^{40}-1)
QACC_H[ 79: 40] = min(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], 2^{40}-1)
...
QACC_H[159:120] = min(QACC_H[159:120] + qx[127:112] * qy[127:112], 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {22{imm16[5]},imm16[5:0],4{0}}
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.166 EE.VMULAS.U16.QACC.LD.XP

Instruction Word

111100	qu[2:1]	qy[0]	001	qu[0]	101	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U16.QACC.LD.XP qu, as, ad, qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
QACC_L[ 39:  0] = min(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], 2^{40}-1)
QACC_L[ 79: 40] = min(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], 2^{40}-1)
...
QACC_L[159:120] = min(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], 2^{40}-1)
QACC_H[ 39:  0] = min(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], 2^{40}-1)
QACC_H[ 79: 40] = min(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], 2^{40}-1)
...
QACC_H[159:120] = min(QACC_H[159:120] + qx[127:112] * qy[127:112], 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.167 EE.VMULAS.U16.QACC.LD.XP.QUP

Instruction Word

110001	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U16.QACC.LD.XP.QUP qu, as, ad, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 39:  0] = min(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], 2^{40}-1)
QACC_L[ 79: 40] = min(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], 2^{40}-1)
...
QACC_L[159:120] = min(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], 2^{40}-1)
QACC_H[ 39:  0] = min(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], 2^{40}-1)
QACC_H[ 79: 40] = min(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], 2^{40}-1)
...
QACC_H[159:120] = min(QACC_H[159:120] + qx[127:112] * qy[127:112], 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.168 EE.VMULAS.U16.QACC.LDBC.INCP

Instruction Word

110	qu[2]	0111	qu[1]	qy[2]	qu[0]	qy[1:0]	qx[2:0]	as[3:0]	0100

Assembler Syntax

EE.VMULAS.U16.QACC.LDBC.INCP qu, as, qx, qy

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 1 比特强制为 0，随后从内存中载入 16-bit 数据，并广播到 qu 寄存器中 8 个 16-bit 数据段内。访存结束后，as 寄存器内数值增加 2。

Operation

```
QACC_L[ 39:  0] = min(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], 2^{40}-1)
QACC_L[ 79: 40] = min(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], 2^{40}-1)
...
QACC_L[159:120] = min(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], 2^{40}-1)
QACC_H[ 39:  0] = min(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], 2^{40}-1)
QACC_H[ 79: 40] = min(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], 2^{40}-1)
...
QACC_H[159:120] = min(QACC_H[159:120] + qx[127:112] * qy[127:112], 2^{40}-1)

qu[127:0] = {8{load16({as[31:1],1{0}})}}
as[31:0] = as[31:0] + 2
```

1.8.169 EE.VMULAS.U16.QACC.LDBC.INCP.QUP

Instruction Word

111000	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	1010	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U16.QACC.LDBC.INCP.QUP qu, as, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 16-bit 分成 8 个数据段。随后计算得到 8 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 1 比特强制为 0，随后从内存中载入 16-bit 数据，并广播到 qu 寄存器中 8 个 16-bit 数据段内。访存结束后，as 寄存器内数值增加 2。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 39:  0] = min(QACC_L[ 39:  0] + qx[ 15:  0] * qy[ 15:  0], 2^{40}-1)
QACC_L[ 79: 40] = min(QACC_L[ 79: 40] + qx[ 31: 16] * qy[ 31: 16], 2^{40}-1)
...
QACC_L[159:120] = min(QACC_L[159:120] + qx[ 63: 48] * qy[ 63: 48], 2^{40}-1)
QACC_H[ 39:  0] = min(QACC_H[ 39:  0] + qx[ 79: 64] * qy[ 79: 64], 2^{40}-1)
QACC_H[ 79: 40] = min(QACC_H[ 79: 40] + qx[ 95: 80] * qy[ 95: 80], 2^{40}-1)
...
QACC_H[159:120] = min(QACC_H[159:120] + qx[127:112] * qy[127:112], 2^{40}-1)

qu[127:0] = {8{load16({as[31:1],1{0}})}}
as[31:0] = as[31:0] + 2
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.170 EE.VMULAS.U8.ACCX

Instruction Word

000010100	qy[2]	0	qy[1:0]	qx[2:0]	11000100

Assembler Syntax

EE.VMULAS.U8.ACCX qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后分别计算出 16 组数据的无符号乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

Operation

```
add0[15:0] = qx[  7:  0] * qy[  7:  0]
add1[15:0] = qx[ 15:  8] * qy[ 15:  8]
...
add15[15:0] = qx[127:120] * qy[127:120]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[15:0] + ad[31:0]d1[15:0] + ... + ad[31:0]d
    15[15:0]

ACCX[40:0] = min(max(sum[40:0], 0), 2^{40}-1)
```

1.8.171 EE.VMULAS.U8.ACCX.LD.IP

Instruction Word

1111	imm16[5:4]	qu[2:1]	qy[0]	000	qu[0]	110	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U8.ACCX.LD.IP qu, as, -512..496, qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后分别计算出 16 组数据的无符号数乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

Operation

```
add0[15:0] = qx[  7:  0] * qy[  7:  0]
add1[15:0] = qx[ 15:  8] * qy[ 15:  8]
...
add15[15:0] = qx[127:120] * qy[127:120]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[15:0] + ad[31:0]d1[15:0] + ... + ad[31:0]d
    15[15:0]
ACCX[40:0] = min(max(sum[40:0], 0), 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.172 EE.VMULAS.U8.ACCX.LD.IP.QUP

Instruction Word

0110	imm16[5:4]	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U8.ACCX.LD.IP.QUP qu, as, imm16, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后分别计算出 16 组数据的无符号数乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
add0[15:0] = qx[  7:  0] * qy[  7:  0]
add1[15:0] = qx[ 15:  8] * qy[ 15:  8]
...
add15[15:0] = qx[127:120] * qy[127:120]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[15:0] + ad[31:0]d1[15:0] + ... + ad[31:0]d
    15[15:0]
ACCX[40:0] = min(max(sum[40:0], 0), 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.173 EE.VMULAS.U8.ACCX.LD.XP

Instruction Word

111100	qu[2:1]	qy[0]	001	qu[0]	110	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U8.ACCX.LD.XP qu, as, ad, qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后分别计算出 16 组数据的无符号数乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
add0[15:0] = qx[  7:  0] * qy[  7:  0]
add1[15:0] = qx[ 15:  8] * qy[ 15:  8]
...
add15[15:0] = qx[127:120] * qy[127:120]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[15:0] + ad[31:0]d1[15:0] + ... + ad[31:0]d
    15[15:0]
ACCX[40:0] = min(max(sum[40:0], 0), 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.174 EE.VMULAS.U8.ACCX.LD.XP.QUP

Instruction Word

110010	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U8.ACCX.LD.XP.QUP qu, as, ad, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后分别计算出 16 组数据的无符号数乘法结果并进行累加操作。累加得到的结果数值再与特殊寄存器 ACCX 中的数值相加，该结果取饱和后存入 ACCX 中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
add0[15:0] = qx[  7:  0] * qy[  7:  0]
add1[15:0] = qx[ 15:  8] * qy[ 15:  8]
...
add15[15:0] = qx[127:120] * qy[127:120]
sum[40:0] = ACCX[39:0] + ad[31:0]d0[15:0] + ad[31:0]d1[15:0] + ... + ad[31:0]d
    15[15:0]
ACCX[40:0] = min(max(sum[40:0], 0), 2^{40}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
qs0[127:0] = {qs1[127: 0], qs0[127: 0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.175 EE.VMULAS.U8.QACC

Instruction Word

000010100	qy[2]	1	qy[1:0]	qx[2:0]	11000100

Assembler Syntax

EE.VMULAS.U8.QACC qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

Operation

```
QACC_L[ 19:  0] = min(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], 2^{20}-1)
QACC_L[ 39: 20] = min(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], 2^{w0}-1)
...
QACC_L[159:140] = min(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], 2^{20}-1)
QACC_H[ 19:  0] = min(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], 2^{20}-1)
QACC_H[ 39: 20] = min(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], 2^{20}-1)
...
QACC_H[159:140] = min(QACC_H[159:140] + qx[127:120] * qy[127:120], 2^{20}-1)
```

1.8.176 EE.VMULAS.U8.QACC.LD.IP

Instruction Word

1111	imm16[5:4]	qu[2:1]	qy[0]	000	qu[0]	111	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U8.QACC.LD.IP qu, as, imm16, qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

Operation

```
QACC_L[ 19:  0] = min(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], 2^{20}-1)
QACC_L[ 39: 20] = min(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], 2^{w0}-1)
...
QACC_L[159:140] = min(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], 2^{20}-1)
QACC_H[ 19:  0] = min(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], 2^{20}-1)
QACC_H[ 39: 20] = min(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], 2^{20}-1)
...
QACC_H[159:140] = min(QACC_H[159:140] + qx[127:120] * qy[127:120], 2^{20}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {22{imm16[5]},imm16[5:0],4{0}}
```

1.8.177 EE.VMULAS.U8.QACC.LD.IP.QUP

Instruction Word

0111	imm16[5:4]	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	imm16[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U8.QACC.LD.IP.QUP qu, as, imm16, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加指令编码段中 6-bit 符号位扩展常数左移 4 位的值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 19:  0] = min(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], 2^{20}-1)
QACC_L[ 39: 20] = min(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], 2^{w0}-1)
...
QACC_L[159:140] = min(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], 2^{20}-1)
QACC_H[ 19:  0] = min(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], 2^{20}-1)
QACC_H[ 39: 20] = min(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], 2^{20}-1)
...
QACC_H[159:140] = min(QACC_H[159:140] + qx[127:120] * qy[127:120], 2^{20}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + {22{imm16[5]},imm16[5:0],4{0}}
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.178 EE.VMULAS.U8.QACC.LD.XP

Instruction Word

111100	qu[2:1]	qy[0]	001	qu[0]	111	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U8.QACC.LD.XP qu, as, ad, qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
QACC_L[ 19:  0] = min(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], 2^{20}-1)
QACC_L[ 39: 20] = min(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], 2^{w0}-1)
...
QACC_L[159:140] = min(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], 2^{20}-1)
QACC_H[ 19:  0] = min(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], 2^{20}-1)
QACC_H[ 39: 20] = min(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], 2^{20}-1)
...
QACC_H[159:140] = min(QACC_H[159:140] + qx[127:120] * qy[127:120], 2^{20}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.179 EE.VMULAS.U8.QACC.LD.XP.QUP

Instruction Word

110011	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	ad[3:0]	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U8.QACC.LD.XP.QUP qu, as, ad, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 19:  0] = min(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], 2^{20}-1)
QACC_L[ 39: 20] = min(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], 2^{w0}-1)
...
QACC_L[159:140] = min(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], 2^{20}-1)
QACC_H[ 19:  0] = min(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], 2^{20}-1)
QACC_H[ 39: 20] = min(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], 2^{20}-1)
...
QACC_H[159:140] = min(QACC_H[159:140] + qx[127:120] * qy[127:120], 2^{20}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.180 EE.VMULAS.U8.QACC.LDBC.INCP

Instruction Word

111	qu[2]	0111	qu[1]	qy[2]	qu[0]	qy[1:0]	qx[2:0]	as[3:0]	0100

Assembler Syntax

EE.VMULAS.U8.QACC.LDBC.INCP qu, as, qx, qy

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，根据访存寄存器 as 中指示的地址，从内存中载入 8-bit 数据，并广播到 qu 寄存器中 16 个 8-bit 数据段内。访存结束后，as 寄存器内数值增加 1。

Operation

```
QACC_L[ 19:  0] = min(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], 2^{20}-1)
QACC_L[ 39: 20] = min(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], 2^{w0}-1)
...
QACC_L[159:140] = min(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], 2^{20}-1)
QACC_H[ 19:  0] = min(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], 2^{20}-1)
QACC_H[ 39: 20] = min(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], 2^{20}-1)
...
QACC_H[159:140] = min(QACC_H[159:140] + qx[127:120] * qy[127:120], 2^{20}-1)

qu[127:0] = {16{load8(as[31:0])}}
as[31:0] = as[31:0] + 1
```

1.8.181 EE.VMULAS.U8.QACC.LDBC.INCP.QUP

Instruction Word

111000	qu[2:1]	qy[0]	qs0[2:0]	qu[0]	qs1[2:0]	qx[1:0]	qy[2:1]	1011	as[3:0]	111	qx[2]

Assembler Syntax

EE.VMULAS.U8.QACC.LDBC.INCP.QUP qu, as, qx, qy, qs0, qs1

Description

该指令将 qx、qy 寄存器按 8-bit 分成 16 个数据段。随后计算得到 16 组数据的无符号数乘法结果，分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 无符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，根据访存寄存器 as 中指示的地址，从内存中载入 8-bit 数据，并广播到 qu 寄存器中 16 个 8-bit 数据段内。访存结束后，as 寄存器内数值增加 1。

该指令还同时从 2 个存放着连续对齐数据的寄存器 qs0、qs1 中通过拼接移位的方式得到 16-byte 非对齐数据存放到 qs0 寄存器中，特殊寄存器 SAR_BYTE 内存储着移位的字节数。

Operation

```
QACC_L[ 19:  0] = min(QACC_L[ 19:  0] + qx[  7:  0] * qy[  7:  0], 2^{20}-1)
QACC_L[ 39: 20] = min(QACC_L[ 39: 20] + qx[ 15:  8] * qy[ 15:  8], 2^{w0}-1)
...
QACC_L[159:140] = min(QACC_L[159:140] + qx[ 63: 56] * qy[ 63: 56], 2^{20}-1)
QACC_H[ 19:  0] = min(QACC_H[ 19:  0] + qx[ 71: 64] * qy[ 71: 64], 2^{20}-1)
QACC_H[ 39: 20] = min(QACC_H[ 39: 20] + qx[ 79: 72] * qy[ 79: 72], 2^{20}-1)
...
QACC_H[159:140] = min(QACC_H[159:140] + qx[127:120] * qy[127:120], 2^{20}-1)

qu[127:0] = {16{load8(as[31:0])}}
as[31:0] = as[31:0] + 1
qs0[127:0]  = {qs1[127:0], qs0[127:0]} >> {SAR_BYTE[3:0] << 3}
```

1.8.182 EE.VPRELU.S16

Instruction Word

10	qz[2:1]	1100	qz[0]	qy[2]	0	qy[1:0]	qx[2:0]	ay[3:0]	0100

Assembler Syntax

EE.VPRELU.S16 qz, qx, qy, ay

Description

该指令将 qx 寄存器按 16-bit 分成 8 个数据段。若数据段内数值不大于 0 时，将数据段数值乘以 qy 寄存器对应的 16-bit 数据段再右移 ay 寄存器的低 6-bit 数值，并将计算结果赋值给 qz 寄存器对应的 16-bit；否则直接将 qx 的数据段内数值赋值给 qz。

Operation

```
qz[ 15:  0] = (qx[ 15:  0]<=0) ? (qx[ 15:  0] * qy[ 15:  0]) >> ay[5:0] : qx[ 15:
    0]
qz[ 31: 16] = (qx[ 31: 16]<=0) ? (qx[ 31: 16] * qy[ 31: 16]) >> ay[5:0] : qx[ 31:
    16]
...
qz[127:112] = (qx[127:112]<=0) ? (qx[127:112] * qy[127:112]) >> ay[5:0] : qx
    [127:112]
```

1.8.183 EE.VPRELU.S8

Instruction Word

10	qz[2:1]	1100	qz[0]	qy[2]	1	qy[1:0]	qx[2:0]	ay[3:0]	0100

Assembler Syntax

EE.VPRELU.S8 qz, qx, qy, ay

Description

该指令将 qx 寄存器按 8-bit 分成 16 个数据段。若数据段内数值不大于 0 时，将数据段数值乘以 qy 寄存器对应的 8-bit 数据段再右移 ay 寄存器的低 5-bit 数值，并将计算结果赋值给 qz 寄存器对应的 8-bit；否则直接将 qx 的数据段内数值赋值给 qz。

Operation

```
qz[  7:  0] = (qx[  7:  0]<=0) ? (qx[  7:  0] * qy[  7:  0]) >> ay[4:0] : qx[  7:  0]
qz[ 15:  8] = (qx[ 15:  8]<=0) ? (qx[ 15:  8] * qy[ 15:  8]) >> ay[4:0] : qx[ 15:  8]
...
qz[127:120] = (qx[127:120]<=0) ? (qx[127:120] * qy[127:120]) >> ay[4:0] : qx[127:120]
```

1.8.184 EE.VRELU.S16

Instruction Word

11	qs[2:1]	1101	qs[0]	001	ax[3:0]	ay[3:0]	0100

Assembler Syntax

EE.VRELU.S16 qs, ax, ay

Description

该指令将 qs 寄存器按 16-bit 分成 8 个数据段。若数据段内数值不大于 0 时，将数据段内数值乘以 ax 寄存器低 16-bit 数值再右移 ay 寄存器的低 6-bit 数值，并用计算结果替换该区域内数值；否则数据段内数值保持不变。

Operation

```
qs[ 15:  0] = (qs[ 15:  0]<=0) ? (qs[ 15:  0] * ax[15:0]) >> ay[5:0] : qs[ 15:  0]
qs[ 31: 16] = (qs[ 31: 16]<=0) ? (qs[ 31: 16] * ax[15:0]) >> ay[5:0] : qs[ 31: 16]
...
qs[127:112] = (qs[127:112]<=0) ? (qs[127:112] * ax[15:0]) >> ay[5:0] : qs[127:112]
```

1.8.185 EE.VRELU.S8

Instruction Word

11	qs[2:1]	1101	qs[0]	101	ax[3:0]	ay[3:0]	0100

Assembler Syntax

EE.VRELU.S8 qs, ax, ay

Description

该指令将 qs 寄存器按 8-bit 分成 16 个数据段。若数据段内数值不大于 0 时，将数据段内数值乘以 ax 寄存器低 8-bit 数值再右移 ay 寄存器的低 5-bit 数值，并用计算结果替换该区域内数值；否则数据段内数值保持不变。

Operation

```
qs[  7:  0] = (qs[  7:  0]<=0) ? (qs[  7:  0] * ax[7:0]) >> ay[4:0] : qs[  7:  0]
qs[ 15:  8] = (qs[ 15:  8]<=0) ? (qs[ 15:  8] * ax[7:0]) >> ay[4:0] : qs[ 15:  8]
...
qs[127:120] = (qs[127:120]<=0) ? (qs[127:120] * ax[7:0]) >> ay[4:0] : qs[127:120]
```

1.8.186 EE.VSL.32

Instruction Word

11	qs[2:1]	1101	qs[0]	01111110	qa[2:0]	0100

Assembler Syntax

EE.VSL.32 qa, qs

Description

分别对 qs 寄存器内的 4 个 32-bit 数据段执行左移操作，左移数值大小为 6-bit 特殊寄存器 SAR 内数据内容。左移过程中，低位补 0，截取左移结果的低 32-bit 存储到 qa 寄存器对应的数据段中。

Operation

```
qa[ 31:  0] = (qs[ 31:  0] << SAR[5:0])
qa[ 63: 32] = (qs[ 63: 32] << SAR[5:0])
qa[ 95: 64] = (qs[ 95: 64] << SAR[5:0])
qa[127: 96] = (qs[127: 96] << SAR[5:0])
```

1.8.187 EE.VSMULAS.S16.QACC

Instruction Word

10	sel8[2:1]	1110	sel8[0]	qy[2]	1	qy[1:0]	qx[2:0]	11000100

Assembler Syntax

EE.VSMULAS.S16.QACC qx, qy, sel8

Description

该指令根据立即数 sel8，从 qy 寄存器的 8 个 16-bit 数据段中选择一个数据段，与 qx 寄存器按 16-bit 分成 8 个数据段分别进行有符号数乘法。将计算得到 8 组结果分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

Operation

```
temp[15:0] = qy[sel8*16+15:sel8*16]
QACC_L[ 39:  0] = min(max(QACC_L[ 39:  0] + qx[ 15:  0] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_L[ 79: 40] = min(max(QACC_L[ 79: 40] + qx[ 31: 16] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_L[119: 80] = min(max(QACC_L[119: 80] + qx[ 47: 32] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_L[159:120] = min(max(QACC_L[159:120] + qx[ 63: 48] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_H[ 39:  0] = min(max(QACC_H[ 39:  0] + qx[ 79: 64] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_H[ 79: 40] = min(max(QACC_H[ 79: 40] + qx[ 95: 80] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_H[119: 80] = min(max(QACC_H[119: 80] + qx[111: 96] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_H[159:120] = min(max(QACC_H[159:120] + qx[127:112] * temp[15:0], -2^{39}),
    2^{39}-1)
```

1.8.188 EE.VSMULAS.S16.QACC.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	111	qu[0]	sel8[2:0]	qx[1:0]	qy[2:1]	1100	as[3:0]	111	qx[2]

Assembler Syntax

EE.VSMULAS.S16.QACC.LD.INCP qu, as, qx, qy, sel8

Description

该指令根据立即数 sel8，从 qy 寄存器的 8 个 16-bit 数据段中选择一个数据段，与 qx 寄存器按 16-bit 分成 8 个数据段分别进行有符号数乘法。将计算得到 8 组结果分别与特殊寄存器 QACC_H 及 QACC_L 对应的 40-bit 数据段进行加法运算。对运算结果按照 40-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 40-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
temp[15:0] = qy[sel8*16+15:sel8*16]
QACC_L[ 39:  0] = min(max(QACC_L[ 39:  0] + qx[ 15:  0] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_L[ 79: 40] = min(max(QACC_L[ 79: 40] + qx[ 31: 16] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_L[119: 80] = min(max(QACC_L[119: 80] + qx[ 47: 32] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_L[159:120] = min(max(QACC_L[159:120] + qx[ 63: 48] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_H[ 39:  0] = min(max(QACC_H[ 39:  0] + qx[ 79: 64] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_H[ 79: 40] = min(max(QACC_H[ 79: 40] + qx[ 95: 80] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_H[119: 80] = min(max(QACC_H[119: 80] + qx[111: 96] * temp[15:0], -2^{39}),
    2^{39}-1)
QACC_H[159:120] = min(max(QACC_H[159:120] + qx[127:112] * temp[15:0], -2^{39}),
    2^{39}-1)

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.189 EE.VSMULAS.S8.QACC

Instruction Word

10	sel16[3:2]	1110	sel16[1]	qy[2]	0	qy[1:0]	qx[2:0]	010	sel16[0]	0100

Assembler Syntax

EE.VSMULAS.S8.QACC qx, qy, sel16

Description

该指令根据立即数 sel16，从 qy 寄存器的 16 个 8-bit 数据段中选择一个数据段，与 qx 寄存器按 8-bit 分成 16 个数据段分别进行有符号数乘法。将计算得到 16 组结果分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

Operation

```
temp[7:0] = qy[sel16*8+7:sel16*8]
QACC_L[ 19:  0] = min(max(QACC_L[ 19:  0] + qx[  7:  0] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_L[ 39: 20] = min(max(QACC_L[ 39: 20] + qx[ 15:  8] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_L[ 59: 40] = min(max(QACC_L[ 59: 40] + qx[ 23: 16] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_L[ 79: 60] = min(max(QACC_L[ 79: 60] + qx[ 31: 24] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_L[ 99: 80] = min(max(QACC_L[ 99: 80] + qx[ 39: 32] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_L[119:100] = min(max(QACC_L[119:100] + qx[ 47: 40] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_L[139:120] = min(max(QACC_L[139:120] + qx[ 55: 48] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_L[159:140] = min(max(QACC_L[159:140] + qx[ 63: 56] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_H[ 19:  0] = min(max(QACC_H[ 19:  0] + qx[ 71: 64] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_H[ 39: 20] = min(max(QACC_H[ 39: 20] + qx[ 79: 72] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_H[ 59: 40] = min(max(QACC_H[ 59: 40] + qx[ 87: 80] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_H[ 79: 60] = min(max(QACC_H[ 79: 60] + qx[ 95: 88] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_H[ 99: 80] = min(max(QACC_H[ 99: 80] + qx[103: 96] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_H[119:100] = min(max(QACC_H[119:100] + qx[111:104] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_H[139:120] = min(max(QACC_H[139:120] + qx[119:112] * temp[7:0], -2^{19}), 2^{19}-1)
QACC_H[159:140] = min(max(QACC_H[159:140] + qx[127:120] * temp[7:0], -2^{19}), 2^{19}-1)
```

1.8.190 EE.VSMULAS.S8.QACC.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	01	sel16[0]	qu[0]	sel16[3:1]	qx[1:0]	qy[2:1]	1100	as[3:0]	111	qx[2]

Assembler Syntax

EE.VSMULAS.S8.QACC.LD.INCP qu, as, qx, qy, sel16

Description

该指令根据立即数 sel16，从 qy 寄存器的 16 个 8-bit 数据段中选择一个数据段，与 qx 寄存器按 8-bit 分成 16 个数据段分别进行有符号数乘法。将计算得到 16 组结果分别与特殊寄存器 QACC_H 及 QACC_L 对应的 20-bit 数据段进行加法运算。对运算结果按照 20-bit 有符号数取饱和后存入 QACC_H 及 QACC_L 对应的 20-bit 数据段中。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
temp[7:0] = qy[sel16*8+7:sel16*8]
QACC_L[ 19:  0] = min(max(QACC_L[ 19:  0] + qx[  7:  0] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_L[ 39: 20] = min(max(QACC_L[ 39: 20] + qx[ 15:  8] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_L[ 59: 40] = min(max(QACC_L[ 59: 40] + qx[ 23: 16] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_L[ 79: 60] = min(max(QACC_L[ 79: 60] + qx[ 31: 24] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_L[ 99: 80] = min(max(QACC_L[ 99: 80] + qx[ 39: 32] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_L[119:100] = min(max(QACC_L[119:100] + qx[ 47: 40] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_L[139:120] = min(max(QACC_L[139:120] + qx[ 55: 48] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_L[159:140] = min(max(QACC_L[159:140] + qx[ 63: 56] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_H[ 19:  0] = min(max(QACC_H[ 19:  0] + qx[ 71: 64] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_H[ 39: 20] = min(max(QACC_H[ 39: 20] + qx[ 79: 72] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_H[ 59: 40] = min(max(QACC_H[ 59: 40] + qx[ 87: 80] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_H[ 79: 60] = min(max(QACC_H[ 79: 60] + qx[ 95: 88] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_H[ 99: 80] = min(max(QACC_H[ 99: 80] + qx[103: 96] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_H[119:100] = min(max(QACC_H[119:100] + qx[111:104] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_H[139:120] = min(max(QACC_H[139:120] + qx[119:112] * temp[7:0], -2^{19}),
    2^{19}-1)
QACC_H[159:140] = min(max(QACC_H[159:140] + qx[127:120] * temp[7:0], -2^{19}),
    2^{19}-1)

```

```
19  qu[127:0] = load128({as[31:4],4{0}})
20  as[31:0] = as[31:0] + 16
```

1.8.191 EE.VSR.32

Instruction Word

11	qs[2:1]	1101	qs[0]	01111111	qa[2:0]	0100

Assembler Syntax

EE.VSR.32 qa, qs

Description

分别对 qs 寄存器内的 4 个 32-bit 数据段执行算术右移操作，右移数值大小为 6-bit 特殊寄存器 SAR 内数据内容。右移过程中，高位填补符号位，截取右移结果的低 32-bit 存储到 qa 寄存器对应的数据段中。

Operation

```
qa[ 31:  0] = (qs[ 31:  0] >> SAR[5:0])
qa[ 63: 32] = (qs[ 63: 32] >> SAR[5:0])
qa[ 95: 64] = (qs[ 95: 64] >> SAR[5:0])
qa[127: 96] = (qs[127: 96] >> SAR[5:0])
```

1.8.192 EE.VST.128.IP

Instruction Word

1	imm16[7]	qv[2:1]	1010	qv[0]	imm16[6:0]	as[3:0]	0100

Assembler Syntax

EE.VST.128.IP qv, as, -2048..2032

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 内 16-byte 数据存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 4 位的值。

Operation

```
qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + {20{imm16[7]},imm16[7:0],4{0}}
```

1.8.193 EE.VST.128.XP

Instruction Word

10	qv[2:1]	1101	qv[0]	111	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.VST.128.XP qv, as, ad

Description

将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 内 16-byte 数据存储到内存。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + ad[31:0]
```

1.8.194 EE.VST.H.64.IP

Instruction Word

1	imm8[7]	qv[2:1]	1011	qv[0]	imm8[6:0]	as[3:0]	0100

Assembler Syntax

EE.VST.H.64.IP qv, as, -1024..1016

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后将寄存器 qv 的高 64-bit 存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 3 位的值。

Operation

```
qv[127: 64] => store64({as[31:3],3{0}})
as[31:0] = as[31:0] + {21{imm8[7]},imm8[7:0],3{0}}
```

1.8.195 EE.VST.H.64.XP

Instruction Word

11	qv[2:1]	1101	qv[0]	000	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.VST.H.64.XP qv, as, ad

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后将寄存器 qv 的高 64-bit 存储到内存。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
qv[127: 64] => store64({as[31:3],3{0}})
as  = as + ad[31:0]
```

1.8.196 EE.VST.L.64.IP

Instruction Word

1	imm8[7]	qv[2:1]	0100	qv[0]	imm8[6:0]	as[3:0]	0100

Assembler Syntax

EE.VST.L.64.IP qv, as, -1024..1016

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后将寄存器 qv 的低 64-bit 存储到内存。访存结束后，as 寄存器内数值增加指令编码段中 8-bit 符号位扩展常数左移 3 位的值。

Operation

```
qv[ 63:  0] => store64({as[31:3],3{0}})
as[31:0] = as[31:0] + {21{imm8[7]},imm8[7:0],3{0}}
```

1.8.197 EE.VST.L.64.XP

Instruction Word

11	qv[2:1]	1101	qv[0]	100	ad[3:0]	as[3:0]	0100

Assembler Syntax

EE.VST.L.64.XP qv, as, ad

Description

将寄存器 as 内访存地址低 3 比特强制为 0，然后将寄存器 qv 的低 64-bit 存储到内存。访存结束后，as 寄存器内数值增加 ad 寄存器内数值。

Operation

```
qv[63:0] => store64({as[31:3],3{0}})
as  = as + ad[31:0]
```

1.8.198 EE.VSUBS.S16

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	11010100

Assembler Syntax

EE.VSUBS.S16 qa, qx, qy

Description

该指令执行数据位宽为 16-bit 的向量减法。qx、qy 寄存器分别是减数和被减数，对计算得到 8 个数据结果取饱和后写入 qa 寄存器。

Operation

```
qa[ 15:  0] = min(max(qx[ 15:  0] - qy[ 15:  0], -2^{15}), 2^{15}-1)
qa[ 31: 16] = min(max(qx[ 31: 16] - qy[ 31: 16], -2^{15}), 2^{15}-1)
...
```

1.8.199 EE.VSUBS.S16.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	100	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1101	as[3:0]	111	qx[2]

Assembler Syntax

EE.VSUBS.S16.LD.INCP qu, as, qa, qx, qy

Description

该指令执行数据位宽为 16-bit 的向量减法。qx、qy 寄存器分别是减数和被减数，对计算得到 8 个数据结果取饱和后写入 qa 寄存器。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 15:  0] = min(max(qx[ 15:  0] - qy[ 15:  0], -2^{15}), 2^{15}-1)
qa[ 31: 16] = min(max(qx[ 31: 16] - qy[ 31: 16], -2^{15}), 2^{15}-1)
...

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.200 EE.VSUBS.S16.ST.INCP

Instruction Word

11101000	qy[0]	qv[2:0]	1	qa[2:0]	qx[1:0]	qy[2:1]	0001	as[3:0]	111	qx[2]

Assembler Syntax

EE.VSUBS.S16.ST.INCP qv, as, qa, qx, qy

Description

该指令执行数据位宽为 16-bit 的向量减法。qx、qy 寄存器分别是减数和被减数，对计算得到 8 个数据结果取饱和后写入 qa 寄存器。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 15:  0] = min(max(qx[ 15:  0] - qy[ 15:  0], -2^{15}), 2^{15}-1)
qa[ 31: 16] = min(max(qx[ 31: 16] - qy[ 31: 16], -2^{15}), 2^{15}-1)
...

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.201 EE.VSUBS.S32

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	11100100

Assembler Syntax

EE.VSUBS.S32 qa, qx, qy

Description

该指令执行数据位宽为 32-bit 的向量减法。qx、qy 寄存器分别是减数和被减数，对计算得到 4 个数据结果取饱和后写入 qa 寄存器。

Operation

```
qa[ 31:  0] = min(max(qx[ 31:  0] - qy[ 31:  0], -2^{31}), 2^{31}-1)
qa[ 63: 32] = min(max(qx[ 63: 32] - qy[ 63: 32], -2^{31}), 2^{31}-1)
...
```

1.8.202 EE.VSUBS.S32.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	101	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1101	as[3:0]	111	qx[2]

Assembler Syntax

EE.VSUBS.S32.LD.INCP qu, as, qa, qx, qy

Description

该指令执行数据位宽为 32-bit 的向量减法。qx、qy 寄存器分别是减数和被减数，对计算得到 4 个数据结果取饱和后写入 qa 寄存器。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[31: 0] = min(max(qx[31: 0] - qy[31: 0], -2^{31}), 2^{31}-1)
qa[63:32] = min(max(qx[63:32] - qy[63:32], -2^{31}), 2^{31}-1)
...

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.203 EE.VSUBS.S32.ST.INCP

Instruction Word

11101000	qy[0]	qv[2:0]	1	qa[2:0]	qx[1:0]	qy[2:1]	0010	as[3:0]	111	qx[2]

Assembler Syntax

EE.VSUBS.S32.ST.INCP qv, as, qa, qx, qy

Description

该指令执行数据位宽为 32-bit 的向量减法。qx、qy 寄存器分别是减数和被减数，对计算得到 4 个数据结果取饱和后写入 qa 寄存器。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[ 31:  0] = min(max(qx[ 31:  0] - qy[ 31:  0], -2^{31}), 2^{31}-1)
qa[ 63: 32] = min(max(qx[ 63: 32] - qy[ 63: 32], -2^{31}), 2^{31}-1)
...

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.204 EE.VSUBS.S8

Instruction Word

10	qa[2:1]	1110	qa[0]	qy[2]	1	qy[1:0]	qx[2:0]	11110100

Assembler Syntax

EE.VSUBS.S8 qa, qx, qy

Description

该指令执行数据位宽为 8-bit 的向量减法。qx、qy 寄存器分别是减数和被减数，对计算得到 16 个数据结果取饱和后写入 qa 寄存器。

Operation

```
qa[  7:  0] = min(max(qx[  7:  0] - qy[  7:  0], -2^{7}), 2^{7}-1)
qa[ 15:  8] = min(max(qx[ 15:  8] - qy[ 15:  8], -2^{7}), 2^{7}-1)
...
```

1.8.205 EE.VSUBS.S8.LD.INCP

Instruction Word

111000	qu[2:1]	qy[0]	110	qu[0]	qa[2:0]	qx[1:0]	qy[2:1]	1101	as[3:0]	111	qx[2]

Assembler Syntax

EE.VSUBS.S8.LD.INCP qu, as, qa, qx, qy

Description

该指令执行数据位宽为 8-bit 的向量减法。qx、qy 寄存器分别是减数和被减数，对计算得到 16 个数据结果取饱和后写入 qa 寄存器。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后从内存中载入 16-byte 数据，并存入 qu 寄存器。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[  7:  0] = min(max(qx[  7:  0] - qy[  7:  0], -2^{7}), 2^{7}-1)
qa[ 15:  8] = min(max(qx[ 15:  8] - qy[ 15:  8], -2^{7}), 2^{7}-1)
...

qu[127:0] = load128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.206 EE.VSUBS.S8.ST.INCP

Instruction Word

11101000	qy[0]	qv[2:0]	1	qa[2:0]	qx[1:0]	qy[2:1]	0011	as[3:0]	111	qx[2]

Assembler Syntax

EE.VSUBS.S8.ST.INCP qv, as, qa, qx, qy

Description

该指令执行数据位宽为 8-bit 的向量减法。qx、qy 寄存器分别是减数和被减数，对计算得到 16 个数据结果取饱和后写入 qa 寄存器。

执行运算操作的同时，将寄存器 as 内访存地址低 4 比特强制为 0，然后将寄存器 qv 的数值存储到内存。访存结束后，as 寄存器内数值增加 16。

Operation

```
qa[  7:  0] = min(max(qx[  7:  0] - qy[  7:  0], -2^{7}), 2^{7}-1)
qa[ 15:  8] = min(max(qx[ 15:  8] - qy[ 15:  8], -2^{7}), 2^{7}-1)
...

qv[127:0] => store128({as[31:4],4{0}})
as[31:0] = as[31:0] + 16
```

1.8.207 EE.VUNZIP.16

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	001110000100

Assembler Syntax

EE.VUNZIP.16 qs0, qs1

Description

该指令用于实现 16-bit 位宽的向量数据的 unzip 算法。

Operation

```
qs0[ 15:  0] = qs0[ 15:  0]
qs0[ 31: 16] = qs0[ 47: 32]
qs0[ 47: 32] = qs0[ 79: 64]
qs0[ 63: 48] = qs0[111: 96]
qs0[ 79: 64] = qs1[ 15:  0]
qs0[ 95: 80] = qs1[ 47: 32]
qs0[111: 96] = qs1[ 79: 64]
qs0[127:112] = qs1[111: 96]
qs1[ 15:  0] = qs0[ 31: 16]
qs1[ 31: 16] = qs0[ 63: 48]
qs1[ 47: 32] = qs0[ 95: 80]
qs1[ 63: 48] = qs0[127:112]
qs1[ 79: 64] = qs1[ 31: 16]
qs1[ 95: 80] = qs1[ 63: 48]
qs1[111: 96] = qs1[ 95: 80]
qs1[127:112] = qs1[127:112]
```

1.8.208 EE.VUNZIP.32

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	001110010100

Assembler Syntax

EE.VUNZIP.32 qs0, qs1

Description

该指令用于实现 32-bit 位宽的向量数据的 unzip 算法。

Operation

```
qs0[ 31:  0] = qs0[ 31:  0]
qs0[ 63: 32] = qs0[ 95: 64]
qs0[ 95: 64] = qs1[ 31:  0]
qs0[127: 96] = qs1[ 95: 64]
qs1[ 31:  0] = qs0[ 63: 32]
qs1[ 63: 32] = qs0[127: 96]
qs1[ 95: 64] = qs1[ 63: 32]
qs1[127: 96] = qs1[127: 96]
```

1.8.209 EE.VUNZIP.8

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	001110100100

Assembler Syntax

EE.VUNZIP.8 qs0, qs1

Description

该指令用于实现 8-bit 位宽的向量数据的 unzip 算法。

Operation

```
qs0[  7:  0] = qs0[  7:  0]
qs0[ 15:  8] = qs0[ 23: 16]
qs0[ 23: 16] = qs0[ 39: 32]
qs0[ 31: 24] = qs0[ 55: 48]
qs0[ 39: 32] = qs0[ 71: 64]
qs0[ 47: 40] = qs0[ 87: 80]
qs0[ 55: 48] = qs0[103: 96]
qs0[ 63: 56] = qs0[119:112]
qs0[ 71: 64] = qs1[  7:  0]
qs0[ 79: 72] = qs1[ 23: 16]
qs0[ 87: 80] = qs1[ 39: 32]
qs0[ 95: 88] = qs1[ 55: 48]
qs0[103: 96] = qs1[ 71: 64]
qs0[111:104] = qs1[ 87: 80]
qs0[119:112] = qs1[103: 96]
qs0[127:120] = qs1[119:112]
qs1[  7:  0] = qs0[ 15:  8]
qs1[ 15:  8] = qs0[ 31: 24]
qs1[ 23: 16] = qs0[ 47: 40]
qs1[ 31: 24] = qs0[ 63: 56]
qs1[ 39: 32] = qs0[ 79: 72]
qs1[ 47: 40] = qs0[ 95: 88]
qs1[ 55: 48] = qs0[111:104]
qs1[ 63: 56] = qs0[127:120]
qs1[ 71: 64] = qs1[ 15:  8]
qs1[ 79: 72] = qs1[ 31: 24]
qs1[ 87: 80] = qs1[ 47: 40]
qs1[ 95: 88] = qs1[ 63: 56]
qs1[103: 96] = qs1[ 79: 72]
qs1[111:104] = qs1[ 95: 88]
qs1[119:112] = qs1[111:104]
qs1[127:120] = qs1[127:120]
```

1.8.210 EE.VZIP.16

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	001110110100

Assembler Syntax

EE.VZIP.16 qs0, qs1

Description

该指令用于实现 16-bit 位宽的向量数据的 zip 算法。

Operation

```
qs0[ 15:  0] = qs0[ 15:  0]
qs0[ 31: 16] = qs1[ 15:  0]
qs0[ 47: 32] = qs0[ 31: 16]
qs0[ 63: 48] = qs1[ 31: 16]
qs0[ 79: 64] = qs0[ 47: 32]
qs0[ 95: 80] = qs1[ 47: 32]
qs0[111: 96] = qs0[ 63: 48]
qs0[127:112] = qs1[ 63: 48]
qs1[ 15:  0] = qs0[ 79: 64]
qs1[ 31: 16] = qs1[ 79: 64]
qs1[ 47: 32] = qs0[ 95: 80]
qs1[ 63: 48] = qs1[ 95: 80]
qs1[ 79: 64] = qs0[111: 96]
qs1[ 95: 80] = qs1[111: 96]
qs1[111: 96] = qs0[127:112]
qs1[127:112] = qs1[127:112]
```

1.8.211 EE.VZIP.32

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	001111000100

Assembler Syntax

EE.VZIP.32 qs0, qs1

Description

该指令用于实现 32-bit 位宽的向量数据的 zip 算法。

Operation

```
qs0[ 31:  0] = qs0[ 31:  0]
qs0[ 63: 32] = qs1[ 31:  0]
qs0[ 95: 64] = qs0[ 63: 32]
qs0[127: 96] = qs1[ 63: 32]
qs1[ 31:  0] = qs0[ 95: 64]
qs1[ 63: 32] = qs1[ 95: 64]
qs1[ 95: 64] = qs0[127: 96]
qs1[127: 96] = qs1[127: 96]
```

1.8.212 EE.VZIP.8

Instruction Word

11	qs1[2:1]	1100	qs1[0]	qs0[2:0]	001111010100

Assembler Syntax

EE.VZIP.8 qs0, qs1

Description

该指令用于实现 8-bit 位宽的向量数据的 zip 算法。

Operation

```
qs0[  7:  0] = qs0[  7:  0]
qs0[ 15:  8] = qs1[  7:  0]
qs0[ 23: 16] = qs0[ 15:  8]
qs0[ 31: 24] = qs1[ 15:  8]
qs0[ 39: 32] = qs0[ 23: 16]
qs0[ 47: 40] = qs1[ 23: 16]
qs0[ 55: 48] = qs0[ 31: 24]
qs0[ 63: 56] = qs1[ 31: 24]
qs0[ 71: 64] = qs0[ 39: 32]
qs0[ 79: 72] = qs1[ 39: 32]
qs0[ 87: 80] = qs0[ 47: 40]
qs0[ 95: 88] = qs1[ 47: 40]
qs0[103: 96] = qs0[ 55: 48]
qs0[111:104] = qs1[ 55: 48]
qs0[119:112] = qs0[ 63: 56]
qs0[127:120] = qs1[ 63: 56]
qs1[  7:  0] = qs0[ 71: 64]
qs1[ 15:  8] = qs1[ 71: 64]
qs1[ 23: 16] = qs0[ 79: 72]
qs1[ 31: 24] = qs1[ 79: 72]
qs1[ 39: 32] = qs0[ 87: 80]
qs1[ 47: 40] = qs1[ 87: 80]
qs1[ 55: 48] = qs0[ 95: 88]
qs1[ 63: 56] = qs1[ 95: 88]
qs1[ 71: 64] = qs0[103: 96]
qs1[ 79: 72] = qs1[103: 96]
qs1[ 87: 80] = qs0[111:104]
qs1[ 95: 88] = qs1[111:104]
qs1[103: 96] = qs0[119:112]
qs1[111:104] = qs1[119:112]
qs1[119:112] = qs0[127:120]
qs1[127:120] = qs1[127:120]
```

1.8.213 EE.WR_MASK_GPIO_OUT

Instruction Word

011100100100	ax[3:0]	as[3:0]	0100

Assembler Syntax

EE.WR_MASK_GPIO_OUT as, ax

Description

该指令为 CPU GPIO 专用指令。功能为置位 GPIO_OUT 某些特定比特位置数据。其中 ax 低 8-bit 存储掩码，as 低 8-bit 存储赋值内容。

Operation

```
GPIO_OUT[7:0] =  (GPIO_OUT[7:0] & ~ax[7:0]) | (as[7:0] & ax[7:0])
```

1.8.214 EE.XORQ

Instruction Word

11	qa[2:1]	1101	qa[0]	011	qy[2:1]	01	qx[2:1]	qy[0]	qx[0]	0100

Assembler Syntax

EE.XORQ qa, qx, qy

Description

该指令对 qx 和 qy 寄存器进行按位异或操作，并将逻辑运算的结果写入 qa 寄存器。

Operation

```
qa = qx ^ qy
```

1.8.215 EE.ZERO.ACCX

Instruction Word

00100101000010000000000100

Assembler Syntax

EE.ZERO.ACCX

Description

将特殊寄存器 ACCX 内数值清零。

Operation

```
ACCX = 0
```

1.8.216 EE.ZERO.Q

Instruction Word

11	qa[2:1]	1101	qa[0]	111111110100100

Assembler Syntax

EE.ZERO.Q qa

Description

将 qa 寄存器内数值清零。

Operation

```
qa = 0
```

1.8.217 EE.ZERO.QACC

Instruction Word

0010010100001000010000100

Assembler Syntax

EE.ZERO.QACC

Description

将特殊寄存器 QACC_L 和 QACC_H 内数值清零。

Operation

```
QACC_L = 0
QACC_H = 0
```

1.8.218 LD.QR

Instruction Word

11	qu[2:1]	1101	qu[0]	010	imm[3:0]	as[3:0]	0100

Assembler Syntax

LD.QR qu, as, imm, -128..112

Description

从内存加载 128 位数据到 QR 寄存器 qu 中。

Operation

```
qu = load128(as + imm)
```

1.8.219 ST.QR

Instruction Word

11	qs[2:1]	1101	qs[0]	110	imm[3:0]	as[3:0]	0100

Assembler Syntax

LD.QR qs, as, imm, -128..112

Description

将 QR 寄存器 qs 中的 128 位数据存储到内存中。

Operation

```
qs => store128(as + imm)
```

1.8.220 MV.QR

Instruction Word

10	qu[2:1]	1111	qu[0]	000	qs[2:1]	000	qs[0]	00100

Assembler Syntax

MV.QR qu, qs

Description

将 QR 寄存器 qs 中的值移动到 QR 寄存器 qu 中。

Operation

```
qu = qs
```

第 2 章

超低功耗协处理器 (ULP-FSM, ULP-RISC-V)

2.1 概述

超低功耗协处理器 (ULP, Ultra-Low-Power coprocessor) 是一种功耗极低的处理器设备，可在芯片进入 Deep-sleep 时保持上电（详见章节 10 低功耗管理 (RTC_CNTL)），允许开发者通过存储在 RTC 存储器中的专用程序，访问 RTC 外设、内部传感器及 RTC 寄存器。在对功耗敏感的场景下，主 CPU 处于睡眠状态以降低功耗，协处理器可以由协处理器定时器唤醒，通过控制 RTC GPIO、RTC I2C、SAR ADC、温度传感器 (TSENS) 等外设监测外部环境或与外部电路进行交互，并在达到唤醒条件时主动唤醒主 CPU。

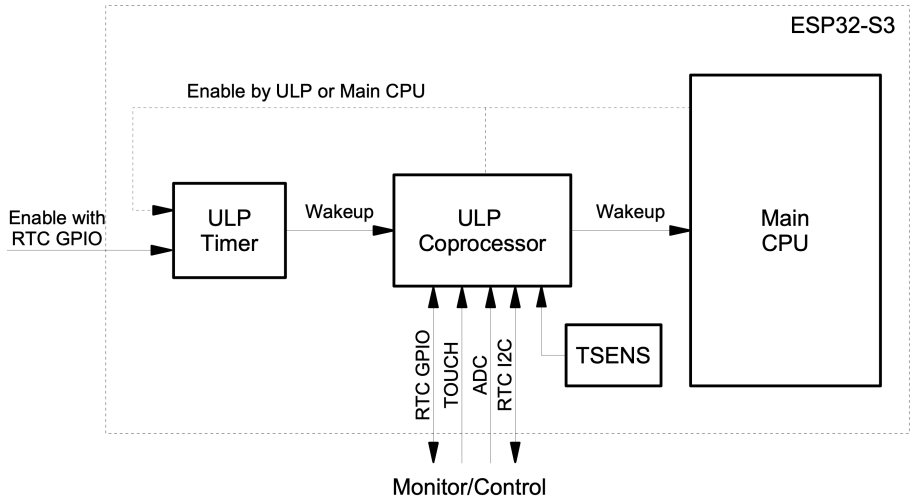


图 2.1-1. 超低功耗协处理器概图

ESP32-S3 搭载了基于有限状态机 (FSM) 实现的 ULP 协处理器（以下简称 ULP-FSM）和基于 RISC-V 指令集的 ULP 协处理器（以下简称 ULP-RISC-V），用户可根据需求灵活选择。

2.2 特性

- 可访问最多 8 KB SRAM RTC 慢速内存，用于储存指令和数据
- 时钟采用 17.5 MHz RTC_FAST_CLK
- 支持正常模式和 Monitor 模式
- 可唤醒 CPU 或向 CPU 发送中断

- 可访问外设、内部传感器及 RTC 寄存器

ULP-FSM 和 ULP-RISC-V 不能同时工作，用户只能选择其中一个作为 ESP32-S3 的超低功耗协处理器。
ULP-FSM 与 ULP-RISC-V 特性比较如下：

表 2.2-1. 超低功耗协处理器特性比较

特性		超低功耗协处理器	
		ULP-FSM	ULP-RISC-V
内存（RTC 慢速内存）		8 KB	
工作时钟频率		17.5 MHz	
唤醒源		ULP 定时器	
工作模式	正常模式	当芯片唤醒时，协助主 CPU 完成部分任务	
	Monitor 模式	当芯片休眠时，可以通过控制传感器完成监控外部环境等任务	
可控制的低功耗外设		ADC1/ADC2	
		RTC I2C	
		RTC GPIO	
		触摸传感器	
		温度传感器	
架构		可编程有限状态机	RISC-V
开发		专用指令集	标准 C 编译器

ESP32-S3 协处理器的功能灵活，可以通过 RTC 寄存器控制 RTC 域中的模块。协处理器可独立于 CPU 运行，是 CPU 的有力补充，甚至可以在一些功耗敏感的设计中取代 CPU。ESP32-S3 协处理器的基本架构可见图 2.2-1。

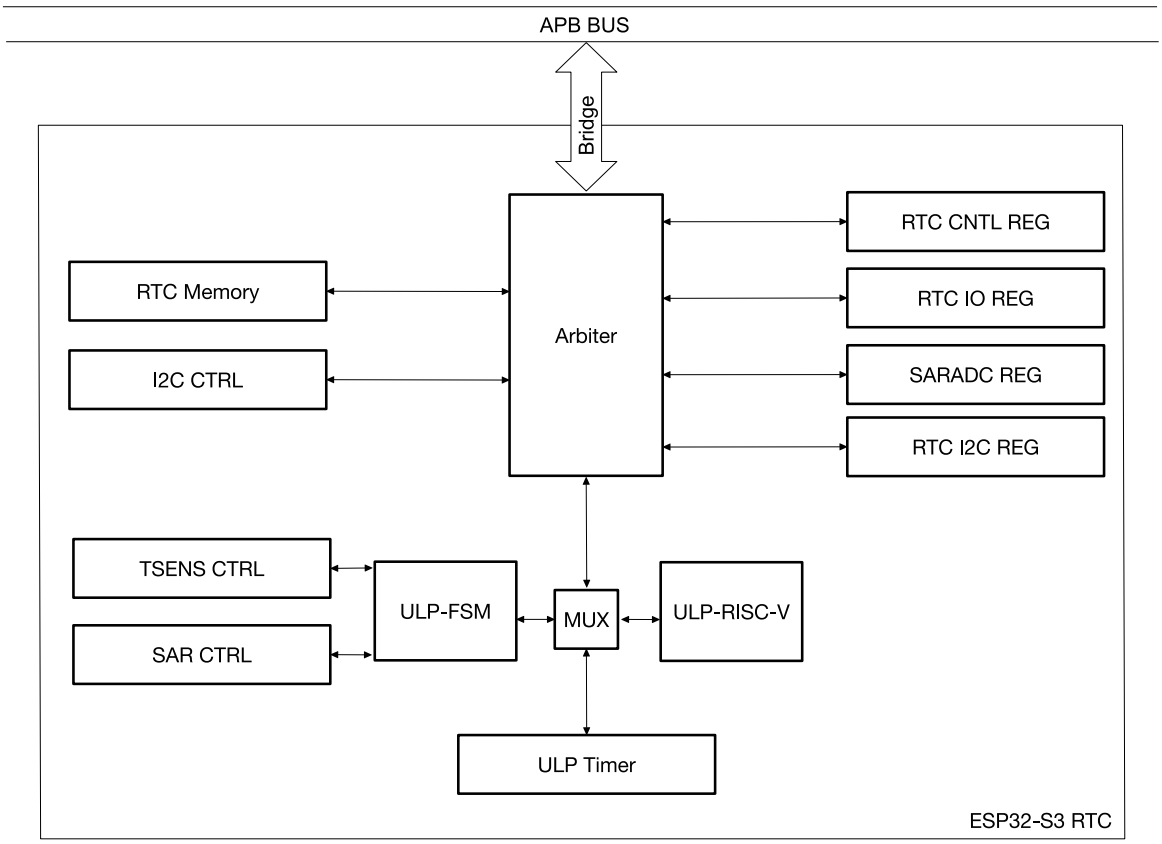


图 2.2-1. 超低功耗协处理器基本架构

2.3 编程流程

ULP-RISC-V 支持用户使用 C 语言编写程序，然后使用编译器将程序编译成 RV32IMC 标准指令码。ULP-RISC-V 支持 RV32IMC 指令集。ULP-FSM 不支持高级语言，用户需使用 ULP-FSM 专门指令集进行编程，见章节 2.5.2。

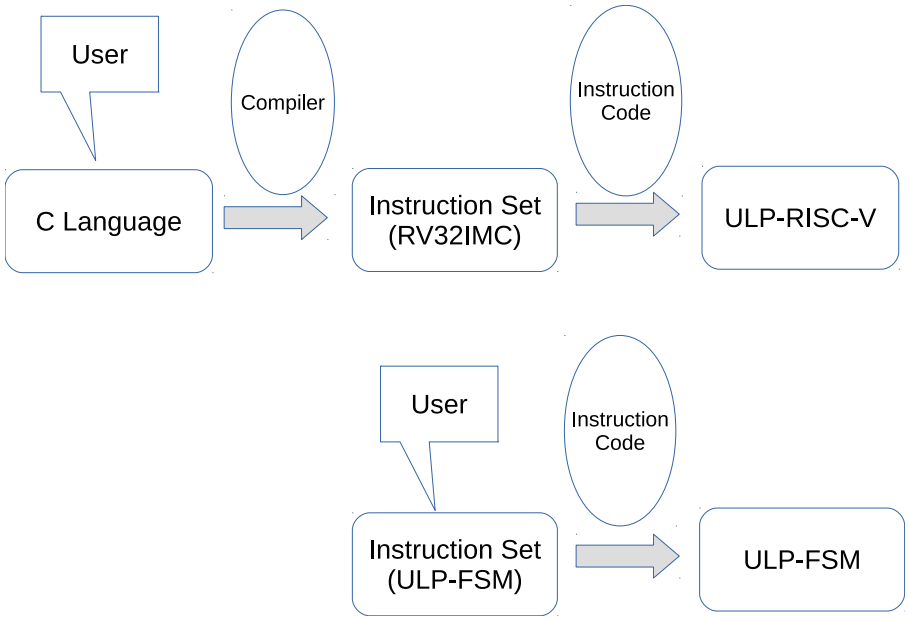


图 2.3-1. 编程流程图

2.4 协处理器的睡眠和唤醒流程

ESP32-S3 的协处理器经过专门设计，无论 CPU 是否处于休眠状态，均可独立于 CPU 运行。

在典型场景中，为了降低功耗，系统可进入 Deep-sleep 模式。系统进入睡眠模式前需完成以下操作：

1. 将协处理器需要执行的程序载入到 RTC 慢速内存；
2. 配置 RTC_CNTL_COCPU_SEL 寄存器，选择协处理器；
   - 0：选择使用 ULP-RISC-V
   - 1：选择使用 ULP-FSM
3. 如果选择使用 ULP-RISC-V，则还需要：
   - 置位再复位 RTC_CNTL_COCPU_CLK_FO；
   - 置位 RTC_CNTL_COCPU_CLKGATE_EN。
4. 配置 RTC_CNTL_ULP_CP_TIMER_1_REG 寄存器来设置硬件定时器的唤醒间隔时间；
5. 选择定时器使能方式：
   - 软件使能：软件置位 RTC_CNTL_ULP_CP_SLP_TIMER_EN；
   - RTC GPIO 使能：软件配置 RTC_CNTL_ULP_CP_GPIO_WAKEUP_ENA 开启通过 RTC GPIO 使能硬件定时器选项。更多信息见章节 10 *低功耗管理 (RTC_CNTL)*。
6. 配置系统进入睡眠状态，主 CPU 休眠。

在 Deep-sleep 模式下：

1. 硬件定时器周期性地将低功耗控制器置于 Monitor 状态，然后唤醒协处理器。更多信息见章节 10 低功耗管理 (RTC_CNTL)；
2. 协处理器唤醒后执行一些必要操作，例如通过低功耗传感器监控芯片外部环境等操作；
3. 操作完成后，系统返回 Deep-sleep 模式；
4. 协处理器进入休眠，等待下一次唤醒。

进入 Monitor 模式后，协处理器的唤醒和睡眠流程见图 2.4-1，包括：

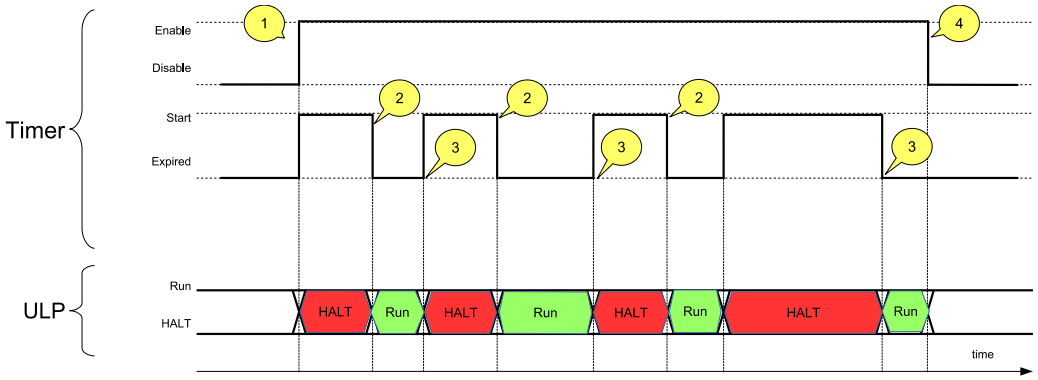


图 2.4-1. 协处理器睡眠和唤醒流程

1. 使能硬件定时器，定时器开始计数；
2. 硬件定时器过期，唤醒协处理器。协处理器进入 Run 状态，执行预先烧写的程序；
3. 协处理器执行 HALT 相关操作进入 HALT 状态；协处理器程序停止运行开始休眠。定时器再次启动，重复以上流程；
   - ULP-RISC-V 的 HALT 操作：置位寄存器 RTC_CNTL_COCPU_DONE；
   - ULP-FSM 的 HALT 操作：执行 HALT 指令。
4. 通过协处理器程序或软件关闭硬件定时器，协处理器将不再进入 Monitor 状态。
   - 软件关闭：软件清零 RTC_CNTL_ULP_CP_SLP_TIMER_EN；
   - RTC GPIO 关闭：软件清零 RTC_CNTL_ULP_CP_GPIO_WAKEUP_ENA，并置位 RTC_CNTL_ULP_CP_GPIO_WAKEUP_CLR。

     注：硬件定时器的关闭方法需要与使能方法一致。

注意：

由于 ULP-RISC-V 进入 HALT 状态需要配置 RTC_CNTL_COCPU_DONE，所以建议预先烧写的程序用下列代码结尾：

- 置位 RTC_CNTL_COCPU_DONE，结束本次 ULP-RISC-V 的运行，并使 ULP-RISC-V 进入休眠模式；
- 置位 RTC_CNTL_COCPU_SHUT_RESET_EN，复位 ULP-RISC-V。注：硬件已经预留了足够的时间支持 ULP-RISC-V 在进入休眠之前完成该操作。

上述信号与寄存器之间的关系可见图 2.4-2。

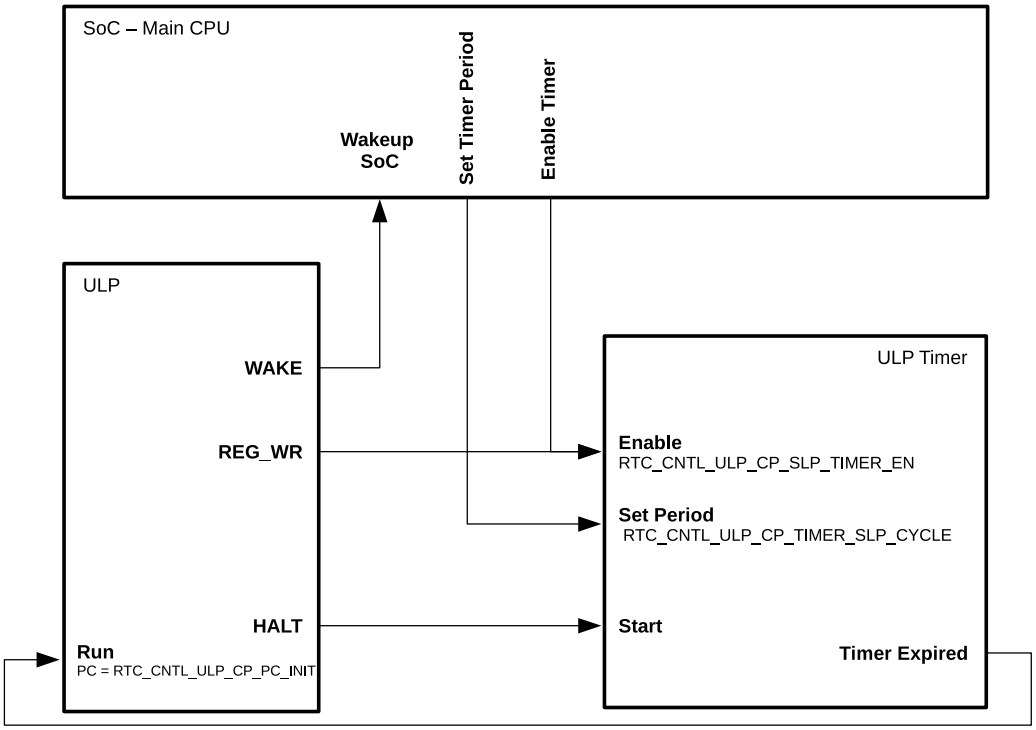


图 2.4-2. ULP 程序框图

2.5 ULP-FSM

2.5.1 特性

ULP-FSM 协处理器是一种可编程有限状态机，可在 CPU 进入 Deep-sleep 状态时工作。协处理器支持部分通用 CPU 指令，可进行一些复杂的逻辑与算术运算。此外，ULP-FSM 还支持一些特殊的 RTC 控制与外设控制指令。ULP-FSM 的运行代码和数据存储在 8 KB SRAM RTC 慢速内存中（CPU 也可访问该区域）。因此，这块内存经常用于存储一些协处理器和 CPU 的通用指令。ULP-FSM 可通过执行 HALT 指令停止运行。

ULP-FSM 具有以下特性：

- 采用 4 个 16 位通用寄存器 (R0 ~ R3)，进行数据操作和内存访问；
- 采用 1 个 8 位阶段计数器寄存器 Stage_cnt，可通过 ALU 指令进行操作并用于 JUMP 指令；
- 内置专用指令，可直接控制低功耗外设，如 SAR ADC，温度传感器等。

2.5.2 指令集

ULP-FSM 可支持下列指令：

- ALU - 算术与逻辑
- LD、ST、REG_RD 及 REG_WR - 加载与数据存储
- JUMP - 跳转至某地址
- WAIT 和 HALT - 管理程序执行
- WAKE - 唤醒 CPU 及与 CPU 通信

- TSENS 和 ADC - 测量

ULP-FSM 指令的格式见图 2.5-1。

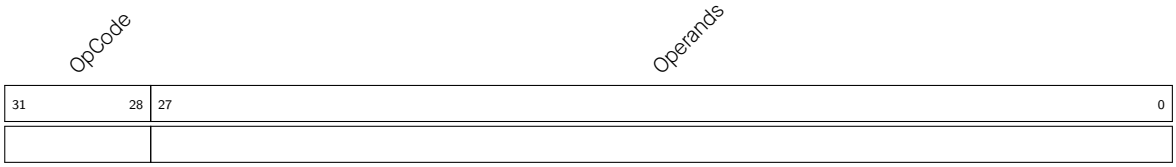


图 2.5-1. ULP-FSM 协处理器的指令格式

根据 *Operands* 的设置不同，同一个 *OpCode* 可对应多种不同操作。例如，ALU 能够执行 10 种不同的算术和逻辑运算，JUMP 也可执行有条件跳转、无条件跳转、绝对跳转及相对跳转等多种形式的跳转。

ULP-FSM 协处理器的所有指令均固定为 32 位。通过这一系列指令，协处理器程序即可得到执行。程序内部的执行均采用 32 位寻址。该程序具体存储在 1 块专用的慢速内存区，地址范围为 0x5000_0000 到 0x5000_1FFF (8 KB)，对主 CPU 可见。

2.5.2.1 ALU - 算术与逻辑运算

算术逻辑单元 (ALU) 可以进行算术和逻辑运算，操作对象为协处理器寄存器中存储的数值或指令中存储的立即值。具体可以支持的运算类型如下：

- 算术 - 加 (ADD) 和减 (SUB)
- 逻辑 - 按位与 (AND) 和按位或 (OR)
- 移位 - 左移 (LSH) 和右移 (RSH)
- 寄存器赋值 - 移动 (MOVE)
- 计数器寄存器操作 - STAGE_RST、STAGE_INC 和 STAGE_DEC

尽管此处 *OpCode* 相同，均为 7，但可通过设置协处理器指令 [27:21] 位，选择特定的算术和逻辑运算。

对寄存器数值的运算

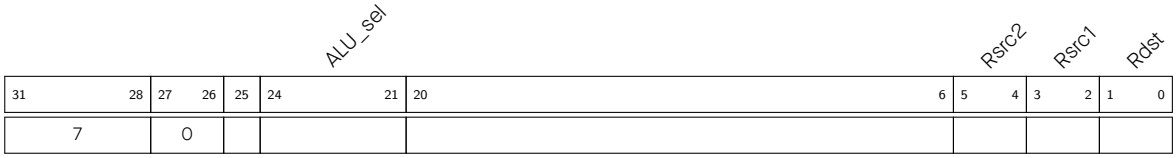


图 2.5-2. 指令类型 - 对寄存器数值的 ALU 运算

如图 2.5-2 所示，当指令 [27:26] 位设置为 0 时，ALU 将对寄存器 R[0-3] 中存储的内容进行运算，运算类型则取决于指令的 ALU_sel[24:21] 位，具体设置方式见表 2.5-1。

Operand	**描述** - 见图 2.5-2
Rdst	寄存器 R[0-3]，目标寄存器，存储计算结果
Rsrc1	寄存器 R[0-3]，源寄存器，存储用于计算的数据
Rsrc2	寄存器 R[0-3]，源寄存器，存储用于计算的数据
ALU_sel	ALU 运算类型选择，具体见表 2.5-1

ALU_sel	指令	运算	描述
0	ADD	*Rdst = Rsrc1 + Rsrc2*	加
1	SUB	*Rdst = Rsrc1 - Rsrc2*	减
2	AND	*Rdst = Rsrc1 & Rsrc2*	按位与
3	OR	*Rdst = Rsrc1 | Rsrc2*	按位或
4	MOVE	*Rdst = Rsrc1*	寄存器赋值
5	LSH	*Rdst = Rsrc1 « Rsrc2*	左移
6	RSH	*Rdst = Rsrc1 » Rsrc2*	右移

表 2.5-1. 对寄存器数值的 ALU 运算

注意：

- ADD 和 SUB 运算可用于设置或清除 ALU 溢出标志位。
- 所有 ALU 运算均可用于设置或清除 ALU 零标志位。

对指令立即值的运算

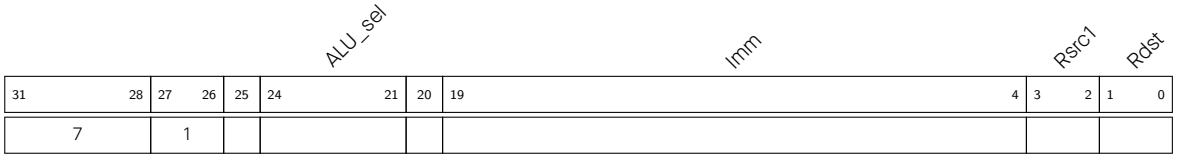


图 2.5-3. 指令类型 - 对指令立即值的 ALU 运算

如图 2.5-3 所示，当指令 [27:26] 位设置为 1 时，ALU 将对寄存器 R[0-3] 和指令 [19:4] 位存储的立即值进行运算。运算类型取决于指令的 ALU_sel [24:21] 位，具体设置方式见表 2.5-2。

Operand	**描述** - 见图 2.5-3
Rdst	寄存器 R[0-3]，目标寄存器，存储计算结果
Rsrc1	寄存器 R[0-3]，源寄存器，存储用于计算的数据
Imm	指令立即值，16 位有符号数，参与运算
ALU_sel	ALU 运算类型选择，具体见表 2.5-2

ALU_sel	指令	运算	描述
0	ADD	*Rdst = Rsrc1 + Imm*	加
1	SUB	*Rdst = Rsrc1 - Imm*	减
2	AND	*Rdst = Rsrc1 & Imm*	按位与
3	OR	*Rdst = Rsrc1 | Imm*	按位或
4	MOVE	*Rdst = Imm*	寄存器赋值
5	LSH	*Rdst = Rsrc1 « Imm*	左移
6	RSH	*Rdst = Rsrc1 » Imm*	右移

表 2.5-2. 对指令立即值的 ALU 运算

注意：

- ADD 和 SUB 运算可用于设置或清除 ALU 溢出标志位。
- 所有 ALU 运算均可用于设置或清除 ALU 零标志位。

对阶段计数器寄存器数值的运算

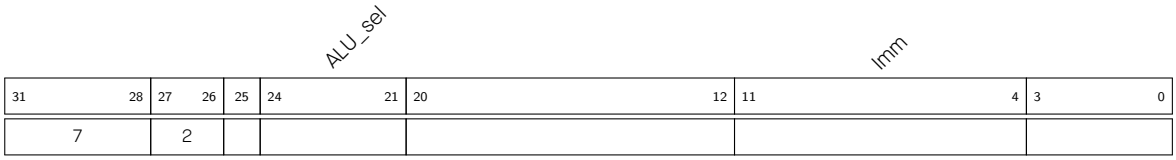


图 2.5-4. 指令类型 - 对阶段计数器寄存器的 ALU 运算

如图 2.5-4 所示，当指令 [27:26] 位设置为 2 时，ALU 将对 8 位寄存器 Stage_cnt 进行递增、递减或重置操作，运算类型取决于指令的 ALU_sel[24:21] 位，具体设置方式见表 2.5-4。Stage_cnt 是一个独立的寄存器，并不是图 2.5-4 所示指令的一部分。

Operand	**描述** - 见图 2.5-4
Imm	指令立即值，8 位数
ALU_sel	ALU 运算类型，具体见表 2.5-3
Stage_cnt	专用 8 位阶段计数器寄存器，可存储循环下标等变量

ALU_sel	指令	运算	描述
0	STAGE_INC	*Stage_cnt* = *Stage_cnt* + *Imm*	阶段计数器寄存器递增
1	STAGE_DEC	*Stage_cnt* = *Stage_cnt* - *Imm*	阶段计数器寄存器递减
2	STAGE_RST	*Stage_cnt* = 0	阶段计数器寄存器复位

表 2.5-3. 对阶段计数器寄存器的 ALU 运算

注意：

该指令主要是与基于阶段计数器的 JUMPS 指令配合使用，构成基于阶段计数器的 for 循环。用法可参考以下伪代码：

```
STAGE_RST                                  // 清空阶段计数器
STAGE_INC                                  // 阶段计数器 ++
{...}                                      // 循环主体，包含 n 条指令
JUMPS (step = n, cond = 0, threshold = m)  // 如果阶段计数值小于 m，则跳转至 STAGE_INC，否则跳出
```

该循环，以此来实现一个阈值为 m 的累加 for 循环。

2.5.2.2 ST - 存储数据至内存

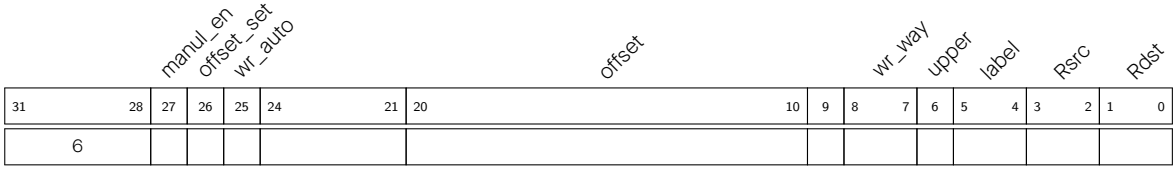


图 2.5-5. 指令类型 - ST

Operand	**描述** - 见图 2.5-5
Rdst	寄存器 R[0-3]，存储目标地址，地址单位为 32 位字
Rsrc	寄存器 R[0-3]，保留需存储的 16 位数
label	数据标志，用户自定义的 2 位无符号数
upper	0：写低半字；1：写高半字
wr_way	0：写全字；1：带 label；3：不带 label
offset	地址偏移，11 位有符号数，单位为 32 位字
wr_auto	使能地址自增模式
offset_set	offset 使能位。0：不设置地址自增模式的基地址偏移；1：设置地址自增模式下的基地址偏移。
manul_en	使能地址指定模式

地址自增模式

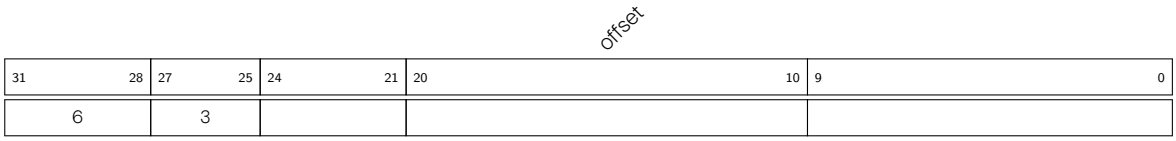


图 2.5-6. 指令类型 - 地址自增模式的基地址偏移 (ST-OFFSET)

Operand	**描述** - 见图 2.5-6
offset	初始地址偏移量，11 位有符号数，单位为 32 位字。

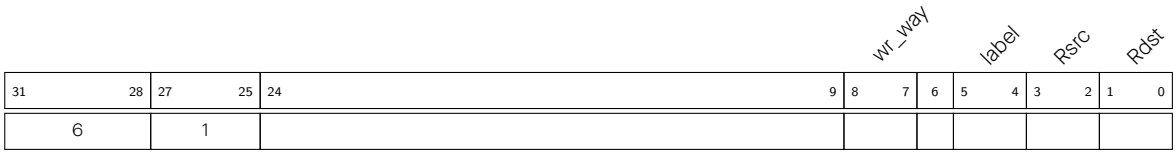


图 2.5-7. 指令类型 - 地址自增模式的数据存储 (ST-AUTO-DATA)

Operand	**描述** - 见图 2.5-7
Rdst	寄存器 R[0-3]，存储目标地址，地址单位为 32 位字
Rsrc	寄存器 R[0-3]，保留需存储的 16 位数
label	数据标志，用户自定义的 2 位无符号数
wr_way	0：写全字；1：带 label；3：不带 label

描述

地址自增模式适用于连续地址的访问，首次使用需设置 ST-OFFSET 指令来配置起始地址偏移，然后通过 ST-AUTO-DATA 指令将 *Rsrc* 中保留的 16 位数存储至内存地址 *Rdst* + *Offset* 中，存储方式见表 2.5-4。其中 ST-AUTO-DATA 的执行次数用 write_cnt 表示。

wr_way	write_cnt	存储数据	运算
0	*	Mem[Rdst + Offset]{31:0} ={PC[10:0], 3'b0, Label[1:0], Rsrc[15:0]}	写全字，包含指针和数据
1	奇数	Mem[Rdst + Offset]{15:0} = {Label[1:0],Rsrc[13:0]}	低半字存储带 label 的数据
1	偶数	Mem[Rdst + Offset]{31:16} = {Label[1:0],Rsrc[13:0]}	高半字存储带 label 的数据
3	奇数	Mem[Rdst + Offset]{15:0} = Rsrc[15:0]	低半字存储不带 label 的数据
3	偶数	Mem[Rdst + Offset]{31:16} = Rsrc[15:0]	高半字存储不带 label 的数据

表 2.5-4. 数据存储格式-地址自增模式

写全字的方式如下：

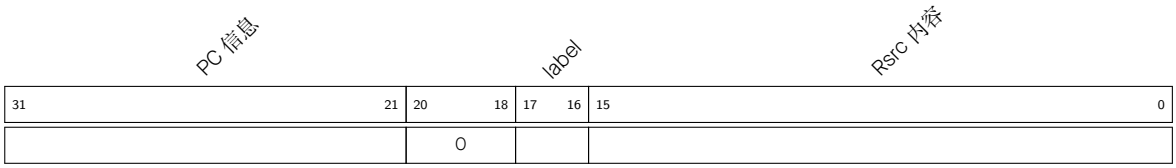


图 2.5-8. MEM[Rdst + Offset] 写全字

位	**描述** - 见图 2.5-8
[15:0]	存储 Rsrc 的内容
[17:16]	存储数据标志，用户定义的 2 位无符号数
[20:18]	默认 3'b0
[31:21]	存储当前指令的 PC 信息，单位为 32 位字

注意：

- 当存储操作为全字时，每执行完一次 ST-AUTO-DATA，Offset 就会自动加 1。
- 当存储操作为半字时，每执行完两次 ST-AUTO-DATA，Offset 就会自动加 1，并且存储顺序为先写低半字，再写高半字。
- 该指令仅能以 32 位字为单位进行访问。
- Mem 写入的是 RTC_SLOW_MEM 慢速内存，ULP 协处理器的地址 0 即相当于主 CPU 的地址 0x50000000。

地址指定模式

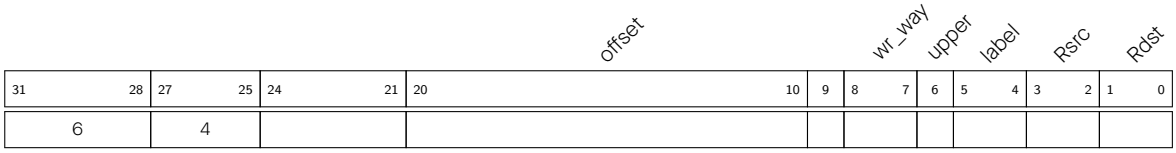


图 2.5-9. 指令类型 - 指定地址模式的数据存储

Operand	**描述** - 见图 2.5-9
Rdst	寄存器 R[0-3]，存储目标地址，地址单位为 32 位字
Rsrc	寄存器 R[0-3]，保留需存储的 16 位数
label	数据标志，用户自定义的 2 位无符号数
upper	0：写低半字；1：写高半字
wr_way	0：写全字；1：带 label；3：不带 label
offset	11 位有符号数，单位为 32 位字

描述

指定地址模式主要运用于地址不连续的存储需求，每条指令均需要提供存储地址及偏移量。存储方式见表 2.5-5。

wr_way	upper	存储数据	运算
0	*	Mem[Rdst + Offset]{31:0} ={PC[10:0], 3'b0, Label[1:0], Rsrc[15:0]}	写全字，包含指针和数据
1	0	Mem[Rdst + Offset]{15:0} = {Label[1:0],Rsrc[13:0]}	低半字存储带 label 的数据
1	1	Mem[Rdst + Offset]{31:16} = {Label[1:0], Rsrc[13:0]}	高半字存储带 label 的数据
3	0	Mem[Rdst + Offset]{15:0} = Rsrc[15:0]	低半字存储不带 label 的数据
3	1	Mem[Rdst + Offset]{31:16} = Rsrc[15:0]	高半字存储不带 label 的数据

表 2.5-5. 数据存储格式-地址指定模式

2.5.2.3 LD - 从内存加载数据

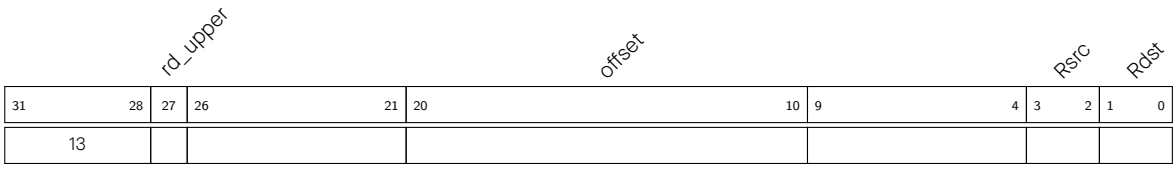


图 2.5-10. 指令类型 - LD

Operand	**描述** - 见图 2.5-10
Rdst	寄存器 R[0-3]，目标寄存器，用于保存从内存加载的数据
Rsrc	寄存器 R[0-3]，存储目标内存的地址，单位为 32 位字
Offset	11 位有符号数，单位为 32 位字
rd_upper	读取数据位置选择： 1 - 读取高半字 0 - 读取低半字

描述

该指令可根据 *rd_upper* 的配置将内存地址 *Rsrc* + *Offset* 中的高或低半字加载至目标寄存器 *Rdst*：

Rdst[15:0] = Mem[Rsrc + Offset]

注意：

- 该指令仅能以 32 位字为单位进行访问。
- 这里的 Mem 是指 RTC_SLOW_MEM 慢速内存，ULP 协处理器的地址 0 即相当于主 CPU 的地址 0x50000000。

2.5.2.4 JUMP - 跳转至绝对地址

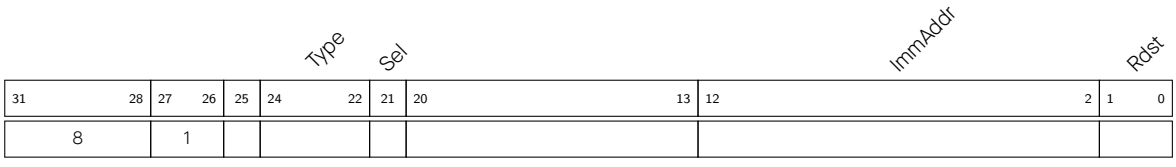


图 2.5-11. 指令类型 - JUMP

Operand	**描述** - 见图 2.5-11
Rdst	寄存器 R[0-3]，存储需跳转至的目标地址
ImmAddr	11 位地址，单位为 32 位字
Sel	跳转目标地址来源： 0 - *ImmAddr* 存储的地址 1 - *Rdst* 存储的地址
Type	跳转类型： 0 - 无条件跳转 1 - 有条件跳转，仅当最后一次 ALU 运算设置了零标志位时跳转 2 - 有条件跳转，仅当最后一次 ALU 运算设置了溢出标志位时跳转

注意：

所有跳转地址均以 32 位字为单位。

描述

该指令可以让 ULP-FSM 跳转至特定地址，跳转可以为无条件跳转或有条件跳转。

2.5.2.5 JUMPR - 跳转至相对地址（基于 R0 寄存器判断）

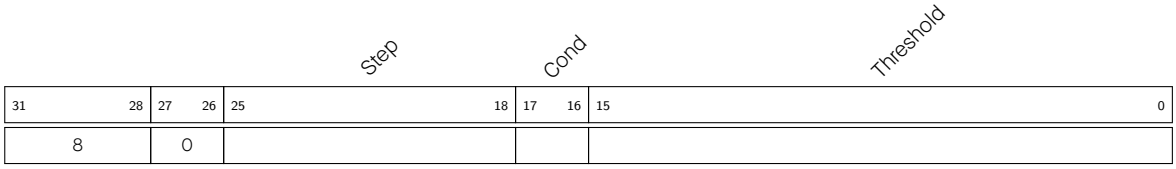


图 2.5-12. 指令类型 - JUMPR

Operand	**描述** - 见图 2.5-12
Threshold	跳转条件阈值，跳转条件见下方 *Cond*
Cond	跳转条件： 0 - 如果 R0 < *Threshold*，即跳转 1 - 如果 R0 > *Threshold*，即跳转 2 - 如果 R0 = *Threshold*，即跳转
Step	相对位移量，单位为 32 位字： 如果 Step[7] = 0，则 PC = PC + Step[6:0] 如果 Step[7] = 1，则 PC = PC - Step[6:0]

注意：

所有跳转地址均以 32 位字为单位。

描述

如果跳转条件（即比较 R0 寄存器的值与 *Threshold* 阈值）为真，该指令可以让协处理器跳转至 1 个相对地址。

2.5.2.6 JUMPS - 跳转至相对地址（基于阶段计数器寄存器判断）

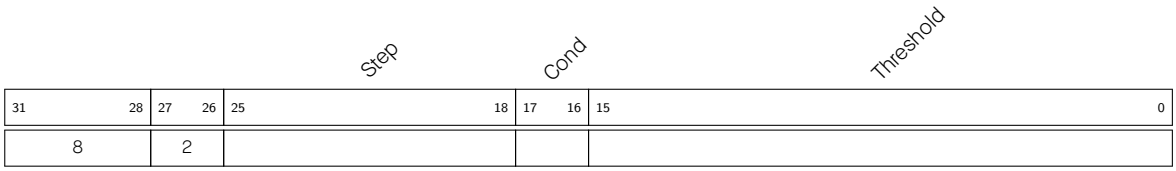


图 2.5-13. 指令类型 - JUMPS

Operand	**描述** - 见图 2.5-13
Threshold	跳转条件阈值，跳转条件见下方 *Cond*
Cond	跳转条件： 1X - 如果 *Stage_cnt* <= *Threshold*，即跳转 00 - 如果 *Stage_cnt* < *Threshold*，即跳转 01 - 如果 *Stage_cnt* >= *Threshold*，即跳转
Step	相对位移量，单位为 32 位字： 如果 Step[7] = 0，则 PC = PC + Step[6:0] 如果 Step[7] = 1，则 PC = PC - Step[6:0]

注意：

- 有关阶段计数器的相关设置，请见章节 2.5.2.1 ALU 阶段计数器。
- 所有跳转地址均以 32 位字为单位。

描述

如果跳转条件（即比较 *Stage_cnt* 阶段计数器寄存器的值与 *Threshold* 阈值）为真，该指令可以让协处理器跳转至 1 个相对地址。

2.5.2.7 HALT - 结束程序

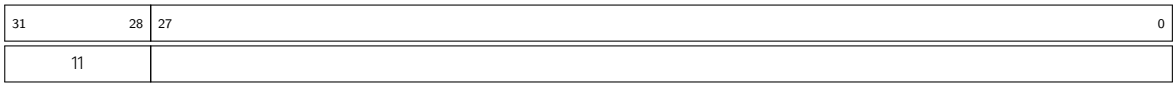


图 2.5-14. 指令类型 - HALT

描述

该指令可以让 ULP-FSM 进入断电模式。

注意：

执行该指令后，ULP 协处理器的硬件定时器将开始计时。

2.5.2.8 WAKE - 唤醒芯片

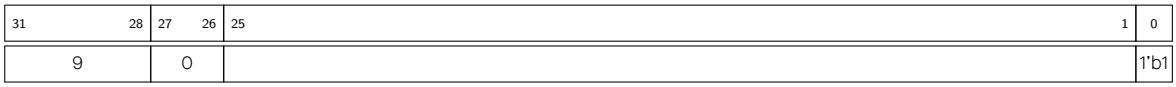


图 2.5-15. 指令类型 - WAKE

描述

该指令可以让 ULP-FSM 向 RTC 控制器发送中断。

- 当芯片处于 Deep-sleep 模式时，该指令可唤醒芯片。
- 当芯片处于 Deep-sleep 之外的模式时，如果 RTC_CNTL_INT_ENA_REG 寄存器设置了 RTC_CNTL_ULP_CP
  _INT_ENA 中断位，该指令即触发 RTC 中断。

2.5.2.9 WAIT - 等待若干个周期

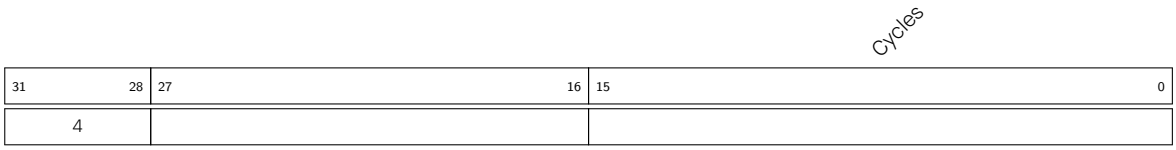


图 2.5-16. 指令类型 - WAIT

Operand	**描述** - 见图 2.5-16
Cycles	等待周期

描述

该指令可以设定 ULP-FSM 暂停工作的等待周期。

2.5.2.10 TSENS - 对温度传感器进行测量

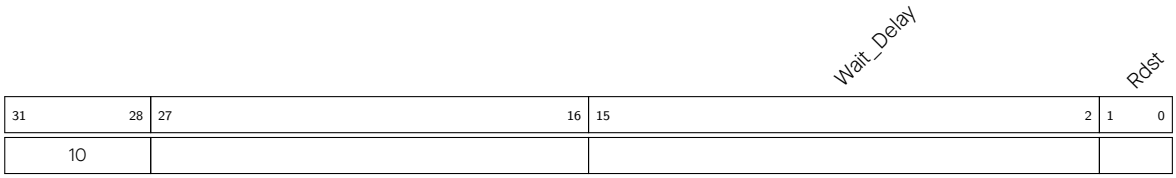


图 2.5-17. 指令类型 - TSENS

Operand	**描述** - 见图 2.5-17
Rdst	目标寄存器 R[0-3]，存储测量结果
Wait_Delay	测量进行的周期数

描述

增加测量周期数 *Wait_Delay* 有助于提高测量精确度或优化测量结果。该指令可对片上温度传感器的数据进行测量，并将测量结果存入 1 个通用寄存器中。

2.5.2.11 ADC - 对 ADC 进行测量

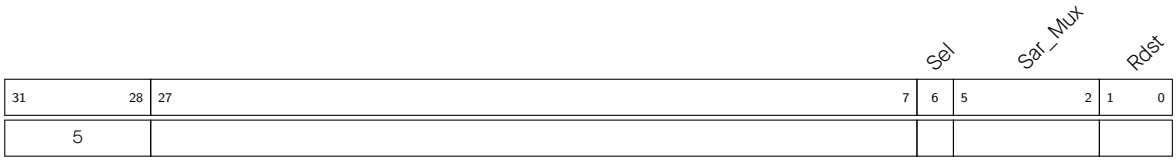


图 2.5-18. 指令类型 - ADC

Operand	**描述** - 见图 2.5-18
Rdst	目标寄存器 R[0-3]，存储测量结果。
Sar_Mux	使能 SAR ADC 通道，通道编号为 [Sar_Mux - 1]。更多通道信息见章节 39 片上传感器与模拟信号处理。
Sel	选择 ADC。0：选择 SAR ADC1；1：选择 SAR ADC2，具体可见表 2.5-6。

表 2.5-6. ADC 指令的输入信号

管脚名/信号名/GPIO	*Sar_Mux*	ADC 选择（*Sel*）
GPIO1	1	*Sel* = 0，选择 SAR ADC1
GPIO2	2	
GPIO3	3	
GPIO4	4	
GPIO5	5	
GPIO6	6	
GPIO7	7	
GPIO8	8	
GPIO9	9	
GPIO10	10	
GPIO11	1	*Sel* = 1，选择 SAR ADC2
GPIO12	2	
GPIO13	3	
GPIO14	4	
XTAL_32K_P	5	
XTAL_32K_N	6	
GPIO17	7	
GPIO18	8	
GPIO19	9	
GPIO20	10	

2.5.2.12 REG_RD - 从外设寄存器读取

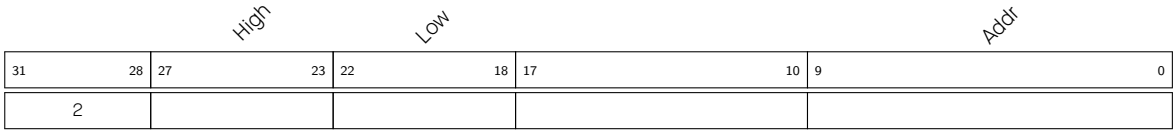


图 2.5-19. 指令类型 - REG_RD

Operand	**描述** - 见图 2.5-19
Addr	外设寄存器地址，单位为 32 位字
Low	寄存器开始位
High	寄存器结束位

描述

该指令可以从外设寄存器中读取最高 16 位的内容，并存入通用寄存器。

R0 = REG[Addr][High:Low]

如需读取的内容超过 16 位，即 *High* - *Low* + 1 > 16，则该指令将返回 [Low+15:Low] 的内容。

注意：

- 该指令可访问 RTC_CNTL、RTC_IO、SENS 及 RTC_I2C 外设中的寄存器。ULP 协处理器可通过相同寄存器在外设总线上的地址 (addr_bus)，计算外设寄存器的地址，具体方式见下：

addr_ulp = (addr_bus - DR_REG_RTCCNTL_BASE)/4

- *addr_ulp* 以 32 位字（而非字节）为单位，0 可投射至 DR_REG_RTCCNTL_BASE（从主 CPU 的角度）。因此，10 位 ULP 协处理器的地址可覆盖外设寄存器空间的 4096 字节，包括 DR_REG_RTCCNTL_BASE（0x60008000）、DR_REG_RTCIO_BASE（0x60008400）、DR_REG_SENS_BASE（0x60008800）及 DR_REG_
  RTC_I2C_BASE（0x60008800）区域。更多地址映射信息，见章节 2.8。

2.5.2.13 REG_WR - 写入外设寄存器

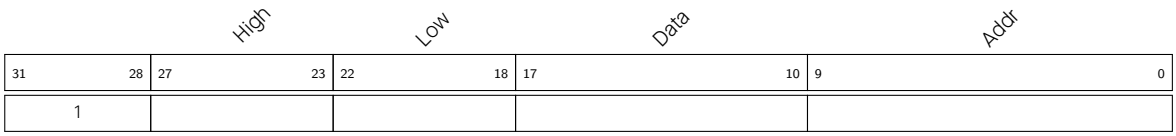


图 2.5-20. 指令类型 - REG_WR

Operand	**描述** - 见图 2.5-20
Addr	目标寄存器地址，单位为 32 位字
Data	需写入的值，8 位数
Low	寄存器开始位
High	寄存器结束位

描述

该指令最高可以向外设寄存器写入一个 8 位立即值 (Data)。

REG[Addr][High:Low] = Data

如需写入的内容超过 8 位，即 *High* - *Low* + 1 > 8，则该指令会给 8 位以上的内容填充 0。

注意：

有关 *addr_ulp* 的内容，请见第 2.5.2.12 节。

2.6 ULP-RISC-V

2.6.1 特性

- 支持 RV32IMC 指令集
- 32 个 32 位通用寄存器
- 32 位乘除法器
- 支持中断

2.6.2 乘除法器

ULP-RISC-V 带有独立的乘除法单元，乘除法相关指令的效率如下表所示：

表 2.6-1. 乘除法指令效率

算法	指令	执行周期	指令描述
乘	MUL	34	两个 32 位整数相乘，返回结果低 32 位
	MULH	66	两个 32 位有符号整数相乘，返回结果高 32 位
	MULHU	66	两个 32 位无符号整数相乘，返回结果高 32 位
	MULHSU	66	32 位有符号整数与无符号整数相乘，返回结果高 32 位
除	DIV	34	两个 32 位整数相除，返回商
	DIVU	34	两个 32 位无符号整数相除，返回商
	REM	34	两个 32 位有符号整数相除，返回余数
	REMU	34	两个 32 位无符号整数相除，返回余数

2.6.3 ULP-RISC-V 中断

2.6.3.1 概述

为了减小 ULP-RISC-V 的面积，ULP-RISC-V 的中断控制器设计没有遵循 RISC-V Privileged ISA 规范，而是使用自定义指令集实现了一个简单的中断控制器。

2.6.3.2 中断控制器

ULP-RISC-V 的中断控制支持 32 个中断输入，但是实际的系统设计中只接入了四个中断，如下表所示。0 ~ 2 号中断为内建中断，由内部中断事件触发；31 号中断接入了 ESP32-S3 的外设中断。

类型	IRQ	中断源
内部中断	0	内部定时器中断
内部中断	1	断点指令 (EBREAK)、环境调用 (ECALL)、或非法指令 (Illegal Instruction)
内部中断	2	总线错误 (BUS Error)，例如访问存储器地址非对齐 (Unaligned Memory Access)
外部中断	31	RTC 外设中断

表 2.6-2. ULP-RISC-V 中断列表

注意：

如果非法指令或者总线错误的中断被禁止，当这两种错误发生时，ULP-RISC-V 将进入 HALT 状态。

ULP-RISC-V 增加了 4 个 32-bit 的中断寄存器 Q0、Q1、Q2、Q3 用于处理中断服务程序。四个寄存器的作用如下：

寄存器	功能
Q0	存储返回地址，如果中断指令是压缩指令，则最低位将被置 1
Q1	包含一个位掩码，其中记录了需要处理的所有中断请求 (IRQ)。这意味着当 Q1 中有多个位被设置时，则调用的中断处理程序需要一次处理多个 IRQ。
Q2	保留寄存器，可供中断服务程序存取数据
Q3	保留寄存器，可供中断服务程序存取数据

表 2.6-3. ULP-RISC-V 的中断寄存器

注意：

- 当 Q1 中有多个中断号的使能为 1，所有的中断都会调用同一个中断服务程序，所以需要在中断服务程序中自行判断中断号并执行响应的程序。
- ULP-RISC-V 复位后，所有的中断为禁止状态。
- 所有中断服务程序入口为 0x10，复位入口为 0x0。

2.6.3.3 中断处理

当中断发生时，ULP-RISC-V 将执行以下操作：

1. 将当前 PC 保存到 Q0
2. 将被响应的中断保存到 Q1
3. 跳转到中断入口 (0x10)

进入中断入口后，需要保存上下文，可使用 Q2 和 Q3 作为暂存备份寄存器。当需要退出中断时，恢复上下文，执行 retirq 指令即可从中断返回，跳转到 Q0。ULP 不支持中断嵌套，在执行 retirq 指令之前不会被任何中断打断。

2.6.3.4 中断相关指令

所有的中断指令均为标准的 R-type 指令，操作码均为 custom0 (0001011)。f3 (funct3) 和 rs2 域在这些指令中会被忽略。标准的 R-type 指令格式见下图。

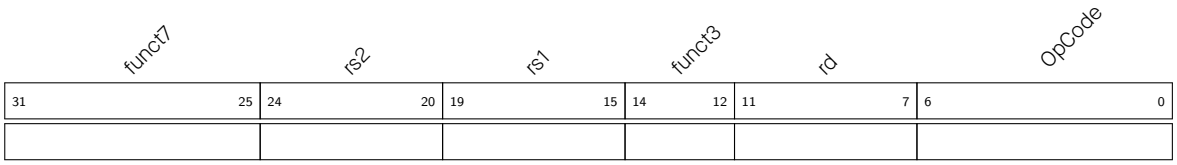


图 2.6-1. 标准 R-type 指令格式

getq rd,qs 指令

getq rd,qs 将 Q*x* 中的寄存器值复制到通用寄存器 rd 中。

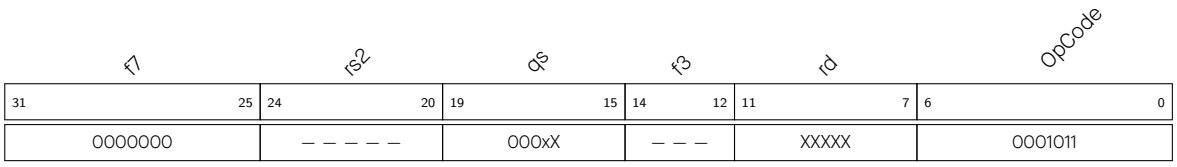


图 2.6-2. 中断指令 - getq rd, qs

Operand	**描述** - 见图 2.6-2
rd	通用目标寄存器地址，该寄存器用于暂存 qs 指定中断寄存器的值
qs	中断寄存器 Q*x* 地址
f7	中断指令编号

setq qd,rs 指令

setq qd,rs 将通用寄存器 rs 的值复制到 Q*x* 寄存器中。

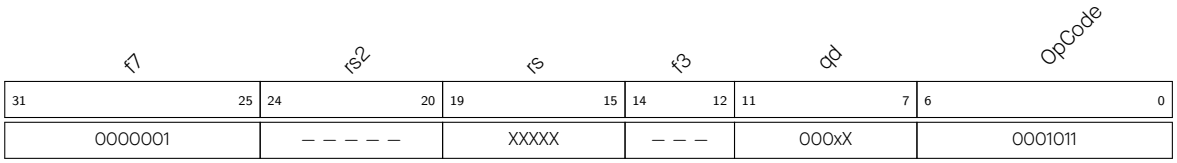


图 2.6-3. 中断指令 - setq qd, rs

Operand	**描述** - 见图 2.6-3
qd	目标中断寄存器地址
rs	通用源寄存器的地址，该寄存器中存储了将要写入中断寄存器的值
f7	中断指令编号

retirq 指令

retirq 中断返回指令，该指令将 Q0 的值复制到 CPU 的 PC 中并且重新使能中断。

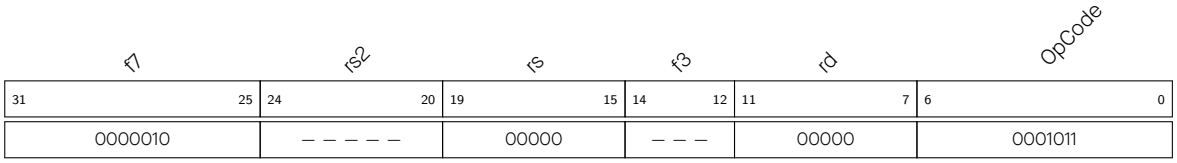


图 2.6-4. 中断指令 - retirq

Operand	**描述** - 见图 2.6-4
f7	中断指令编号

maskirq rd, rs 指令

maskirq rd, rs 指令将 IRQ Mask 寄存器中的值复制到 rd 中，并将 rs 的值复制到 IRQ Mask 中。

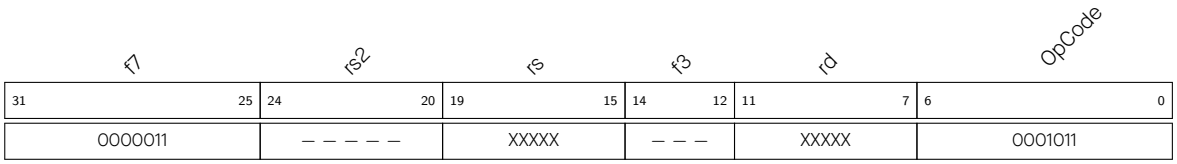


图 2.6-5. 中断指令 - Maskirq rd rs

Operand	**描述** - 见图 2.6-5
rd	通用目标寄存器地址，用于暂存当前 Q1 寄存器中的值
rs	通用源寄存器地址，该寄存器暂存即将写入 Q1 的值
f7	中断指令编号

2.6.3.5 RTC 外设中断

ESP32-S3 的部分传感器、软件以及 RTC I2C 的中断可接入 ULP-RISC-V。用户可配置寄存器 SENS_SAR_COCPU_
INT_ENA_REG 使能中断，如表 2.6-4 所示：

中断使能位	中断名称	中断描述
0	TOUCH_DONE_INT	触摸传感器完成一个通道扫描则触发此中断
1	TOUCH_INACTIVE_INT	触摸传感器的触碰被释放则触发此中断
2	TOUCH_ACTIVE_INT	触摸传感器感受到触碰则触发此中断
3	SARADC1_DONE_INT	SAR ADC1 完成一次转换则触发此中断
4	SARADC2_DONE_INT	SAR ADC2 完成一次转换则触发此中断
5	TSENS_DONE_INT	温度传感器数据转存完成则触发此中断
6	RISCV_START_INT	ULP-RISC-V 上电开始工作则触发此中断
7	SW_INT	软件中断
8	SWD_INT	超级看门狗超时则触发此中断
9	TOUCH_TIME_OUT_INT	TOUCH 采样超时则触发此中断
10	TOUCH_APPROACH_LOOP_DONE_INT	TOUCH 完成一个 APPROACH 采样周期触发此中断
11	TOUCH_SCAN_DONE_INT	TOUCH 每扫描完最后一个 TOUCH 通道触发此中断

表 2.6-4. ULP-RISC-V 的外设中断列表

注意：

- 除了上述中断以外，ULP-RISC-V 还可以响应来自 RTC_IO 的中断，只需要将 RTC_IO 配置为输入模式即可。具体触发方式可通过 RTCIO_GPIO_PIN*n*_INT_TYPE 进行配置。关于 RTC_IO 的配置，见 IO MUX 和 GPIO 交换矩阵章节。
- RTC_IO 的中断需要通过查询寄存器 RTCIO_RTC_GPIO_STATUS_INT 识别中断来源，停用 RTC_IO 可清除此中断。
- 软件中断通过配置寄存器 RTC_CNTL_COCPU_SW_INT_TRIGGER 产生。
- RTC I2C 的中断描述见章节 2.7.4。

2.7 RTC I2C 控制器

ULP 协处理器可通过 RTC I2C 控制器与外部 I2C 从机进行基本的读写操作。

2.7.1 连接 RTC I2C 信号

SDA 和 SCL 时钟信号可通过 RTCIO_SAR_I2C_IO_REG 寄存器，连接至 2 个 GPIO 管脚（4 个可选），详细定义请见章节 IO MUX 和 GPIO 矩阵中的 RTC_MUX 管脚清单。

2.7.2 配置 RTC I2C 控制器

ULP 协处理器在正常使用 I2C 指令之前必须配置 RTC I2C 控制器中的特定参数，具体即向 RTC I2C 寄存器写入特定计时参数。这一步可通过主 CPU 或 ULP 自身运行程序完成。

> **说明：**
> 计时参数均以 RTC_FAST_CLK (17.5 MHz) 为单位。

1. 通过 RTC_I2C_SCL_LOW_PERIOD_REG 和 RTC_I2C_SCL_HIGH_PERIOD_REG 设置 RTC_FAST_CLK 周期中 SCL 时钟的高低电平宽度和周期（例，频率为 100 kHz 时，设置 RTC_I2C_SCL_LOW_PERIOD_REG

= 40、RTC_I2C_SCL_HIGH_PERIOD_REG = 40）。

2. 通过 RTC_FAST_CLK 中的 RTC_I2C_SDA_DUTY_REG 设置 SDA 切换前等待的周期数（例，RTC_I2C_SDA_
DUTY_REG = 16）。
3. 通过 RTC_I2C_SCL_START_PERIOD_REG 设置启动信号后的等待时间（例，RTC_I2C_SCL_START_PERIOD_REG = 30）。
4. 通过 RTC_I2C_SCL_STOP_PERIOD_REG 设置停止信号前的等待时间（例，RTC_I2C_SCL_STOP_PERIOD_REG = 44）。
5. 通过 RTC_I2C_TIME_OUT_REG 设置通信超时参数（例，RTC_I2C_TIME_OUT_REG = 200）。
6. 通过 RTC_I2C_CTRL_REG 中的 RTC_I2C_MS_MODE 位启动主机模式。
7. 配置外部从机地址：
   - 如果选择使用主 CPU 或 ULP-RISC-V，则需要将外部从机的地址写入 SENS_SAR_I2C_SLAVE_ADDR。
   - 如果选择使用 ULP-FSM，则需要将外部从机的地址写入 SENS_I2C_SLAVE_ADDR*n* (*n*: 0-7)。

   最多可通过这种方式预编程 8 个从机地址。此后，可为每次通信选择 1 个上述地址，共同组成协处理器 I2C 指令。

完成上述 RTC I2C 初始化配置后，即可由主 CPU 或者协处理器开始与外部 I2C 发起通信。

2.7.3 使用 RTC I2C

2.7.3.1 I2C 指令编码格式

RTC I2C 的指令编码与 I2C0/I2C1 的编码方式基本一致（见 I2C 控制器章节的 CMD_Controller）。不同的地方在于：

- RTC I2C 的 op_code 对应的命令不同：
  - RSTART: op_code 等于 0 时为 RSTART 命令
  - WRITE: op_code 等于 1 时为 WRITE 命令
  - READ: op_code 等于 2 时为 READ 命令
  - STOP: op_code 等于 3 时为 STOP 命令
  - END: op_code 等于 4 时为 END 命令
- RTC I2C 为不同的操作设置了固定的指令段，具体如下
  - 命令 0 ~ 命令 1：I2C 写操作
  - 命令 2 ~ 命令 6：I2C 读操作

注意：所有从机地址均为 7 位。

2.7.3.2 I2C_RD - I2C 读流程

执行 I2C 读取操作之前，需配置以下信息：

- 根据需求配置 I2C 的指令列表，包括指令顺序、指令码和读取数据个数 (byte_num) 等信息。配置方法见 I2C 控制器章节中 I2C0/I2C1 的配置方法。

- 使用寄存器 SENS_SAR_I2C_REG_ADDR 配置从机寄存器地址。
- 置位 SENS_SAR_I2C_START_FORCE，以及 SENS_SAR_I2C_START 开始 I2C 的传输。
- 每接收到 RTC_I2C_RX_DATA_INT 中断，将读取的数据 (RTC_I2C_RDATA) 转存至 SRAM RTC 慢速内存中或直接使用。

I2C_RD 指令的执行步骤如下，见图 2.7-1:

1. 主机发送启动 (START) 信号；
2. 主机发送命令字节，包括从机地址和读/写控制位（此时，读/写控制位置为 0，代表“写”）。从机地址可从 SENS_I2C_SLAVE_ADDR*n* 中获取；
3. 从机发送应答 (ACK) 信号；
4. 主机发送从机寄存器地址；
5. 从机发送应答信号；
6. 主机发送重复启动 (RSTART) 信号；
7. 主机发送从机地址，其中读/写控制位置为 1，代表“读”；
8. 从机发送 1 个字节的数据；
9. 主机判断传输字节数是否达到当前指令设定的传输字节数。如果达到规定字数，则从读指令中跳出，主机发送非应答信号。否则重复步骤 8，继续等待从机发送下一个字节。
10. 进入停止指令，主机发送停止 (STOP) 信号，结束读取。

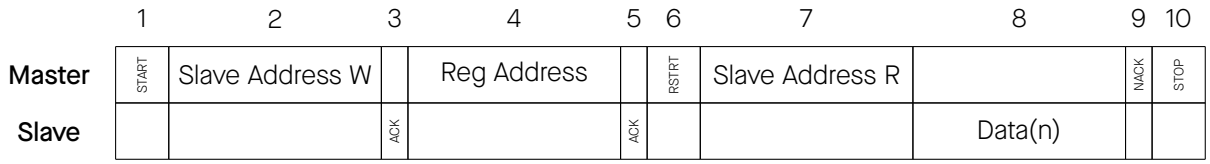


图 2.7-1. I2C 读操作

注意：
RTC I2C 控制器外设会对 SCL 时钟下降沿上的 SDA 信号进行采样。如果从机的 SDA 信号在约 0.38 ms 内发生改变，主机则将接收到不正确的数据。

2.7.3.3 I2C_WR - I2C 写流程

执行 I2C 写操作之前，需配置以下信息：

- 根据需求配置 I2C 的指令列表，包括指令顺序、指令码和待写的数据个数 (byte_num) 等信息。配置方法见 I2C 控制器章节中 I2C0/I2C1 的配置方法；
- 使用寄存器 SENS_SAR_I2C_REG_ADDR 配置从机寄存器地址；
- 使用寄存器 hyperref[fielddesc:SENSSARI2CWDATA]SENS_SAR_I2C_WDATA 配置传输数据；
- 置位 SENS_SAR_I2C_START_FORCE，以及 SENS_SAR_I2C_START 开始 I2C 的传输；
- 每次当接收到 RTC_I2C_TX_DATA_INT 中断，更新下一个需要传输的数据 (hyperref[fielddesc:SENSSARI2CWDATA]SENS_SAR_I2C_WDATA)。

I2C_WR 指令的执行步骤如下，见图 2.7-2：

1. 主机发送开始信号；
2. 主机发送命令字节，包括从机地址和读/写控制位（此时，读/写控制位置为 0，代表“写”）。从机地址可从 SENS_I2C_SLAVE_ADDR*n* 中获取；
3. 从机发送应答信号；
4. 进入下一条指令，主机发送从机寄存器地址；
5. 从机发送应答信号；
6. 主机发送重复启动信号；
7. 主机发送从机地址，其中读/写位置为 0，代表“写”；
8. 主机发送 1 个字节的数据；
9. 从机发送应答信号；主机判断传输字节数是否达到当前指令设定的传输字节数，如果达到规定字数，则结束写指令跳转至下一条指令。否则重复步骤 8，继续发送下一个字节；
10. 进入停止指令，主机发送停止指令，结束本次传输。

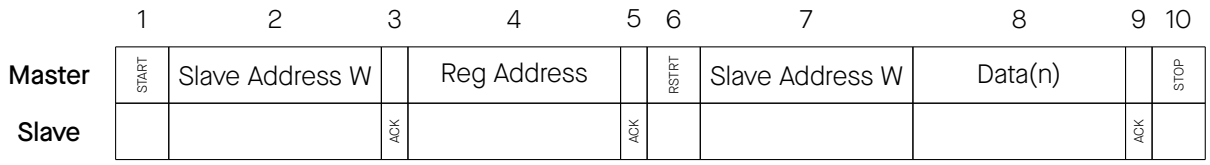


图 2.7-2. I2C 写操作

2.7.3.4 检测错误条件

应用程序可以通过查询 RTC_I2C_INT_ST_REG 寄存器中的特定位，判断指令是否成功执行。为了检查特定的通信活动，应首先设置 RTC_I2C_INT_ENA_REG 寄存器中的相应位。注意，系统位图将移 1。如果检测到特定通信活动，且设置了 RTC_I2C_INT_ST_REG 寄存器，则可通过 RTC_I2C_INT_CLR_REG 寄存器清零。

2.7.4 RTC I2C 中断

- RTC_I2C_SLAVE_TRAN_COMP_INT：从机完成传输后则触发此中断。
- RTC_I2C_ARBITRATION_LOST_INT：主机失去总线控制权后则触发此中断。
- RTC_I2C_MASTER_TRAN_COMP_INT：主机传输完成后则触发此中断。
- RTC_I2C_TRANS_COMPLETE_INT：检测到 STOP 位时则触发此中断。
- RTC_I2C_TIME_OUT_INT：出现超时事件则触发此中断。
- RTC_I2C_ACK_ERR_INT：出现 ACK 错误则触发此中断。
- RTC_I2C_RX_DATA_INT：接收数据则触发此中断。
- RTC_I2C_TX_DATA_INT：发送数据则触发此中断。
- RTC_I2C_DETECT_START_INT：检测到开始信号则触发此中断。

2.8 地址映射

表 2.8-1 列出了 ULP 协处理器访问外设寄存器所用到的基地址寄存器以及相关的地址映射。

表 2.8-1. 地址映射

外设	基地址寄存器	总线地址	ULP-FSM 基地址	ULP-RISC-V 基地址
RTC Control	DR_REG_RTCCNTL_BASE	0x60008000	0x8000	0x8000
RTC GPIO	DR_REG_RTC_IO_BASE	0x60008400	0x8400	0xA400
ADC, Touch, TSENS	DR_REG_SENS_BASE	0x60008800	0x8800	0xC800
RTC I2C	DR_REG_RTC_I2C_BASE	0x60008C00	0x8C00	0xEC00

表 2.8-2 列出了 ULP 协处理器可访问的外设寄存器。

表 2.8-2. ULP 协处理器可访问的外设寄存器

外设寄存器	寄存器描述
RTC CNTL 寄存器	描述见章节 10 低功耗管理 (RTC_CNTL)
RTC GPIO 寄存器	描述见章节 6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)
ARC、Touch、TSENS 寄存器	描述见章节 39 片上传感器与模拟信号处理
RTC I2C 寄存器	见本章节第 2.9.4 小节 RTC I2C (I2C) 寄存器列表

2.9 寄存器列表

本章节涉及到以下寄存器：

- ULP (ALWAYS_ON) 为非掉电寄存器，寄存器不会随着 RTC_PERI 的电源域（见章节 10 低功耗管理 (RTC_CNTL)）掉电而复位。
- ULP (RTC_PERI) 处于 RTC_PERI 电源域下，如果 RTC_PERI 的电源域（见章节 10 低功耗管理 (RTC_CNTL)）掉电，寄存器值将会被复位。
- RTC I2C 寄存器：包括 RTC_PERI 寄存器与 I2C 寄存器，并且都处于 RTC_PERI 的电源域下。

2.9.1 ULP (ALWAYS_ON) 寄存器列表

本小节的所有地址均为相对于低功耗管理模块基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

名称	描述	地址	访问
ULP 定时器控制器			
RTC_CNTL_ULP_CP_TIMER_REG	配置协处理器定时器	0x00FC	不定
RTC_CNTL_ULP_CP_TIMER_1_REG	配置定时器睡眠周期	0x0134	读/写
ULP-FSM 寄存器			
RTC_CNTL_ULP_CP_CTRL_REG	ULP-FSM 配置寄存器	0x0100	读/写
ULP-RISC-V 寄存器			
RTC_CNTL_COCPU_CTRL_REG	ULP-RISC-V 配置寄存器	0x0104	不定

2.9.2 ULP (RTC_PERI) 寄存器列表

本小节的所有地址均为相对于低功耗管理模块基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

名称	描述	地址	访问
ULP-RISC-V 寄存器			
SENS_ SAR_COCPU_INT_RAW_REG	ULP-RISC-V 的原始中断位	0x00E8	只读
SENS_SAR_COCPU_INT_ENA_REG	ULP-RISC-V 的中断使能位	0x00EC	读/写
SENS_SAR_COCPU_INT_ST_REG	ULP-RISC-V 的中断状态位	0x00F0	只读
SENS_SAR_COCPU_INT_CLR_REG	ULP-RISC-V 的中断清零位	0x00F4	只写

2.9.3 RTC I2C (RTC_PERI) 寄存器列表

本小节的所有地址均为相对于低功耗管理模块基地址 + 0x0800 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

名称	描述	地址	访问
RTC I2C 控制器寄存器			
SENS_SAR_I2C_CTRL_REG	RTC I2C 传输配置	0x0058	读/写
RTC I2C 从机地址配置寄存器			
SENS_SAR_SLAVE_ADDR1_REG	配置 RTC I2C 从机地址 0-1	0x0040	读/写
SENS_SAR_SLAVE_ADDR2_REG	配置 RTC I2C 从机地址 2-3	0x0044	读/写
SENS_SAR_SLAVE_ADDR3_REG	配置 RTC I2C 从机地址 4-5	0x0048	读/写
SENS_SAR_SLAVE_ADDR4_REG	配置 RTC I2C 从机地址 6-7	0x004C	读/写

2.9.4 RTC I2C (I2C) 寄存器列表

本小节的所有地址均为相对于低功耗管理模块基地址 + 0x0C00 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

名称	描述	地址	访问
RTC I2C 信号设置寄存器			
RTC_I2C_SCL_LOW_REG	配置 SCL 时钟的低电平宽度	0x0000	读/写
RTC_I2C_SCL_HIGH_REG	配置 SCL 时钟的高电平宽度	0x0014	读/写
RTC_I2C_SDA_DUTY_REG	配置 SCL 下降沿后的 SDA 保持时间	0x0018	读/写
RTC_I2C_SCL_START_PERIOD_REG	配置开始条件下，SDA 与 SCL 下降沿之间的延迟	0x001C	读/写
RTC_I2C_SCL_STOP_PERIOD_REG	配置停止条件下，SDA 与 SCL 下降沿之间的延迟	0x0020	读/写
RTC I2C 控制寄存器			
RTC_I2C_CTRL_REG	传输设置	0x0004	读/写
RTC_I2C_STATUS_REG	RTC I2C 状态	0x0008	只读
RTC_I2C_TO_REG	RTC I2C 超时设置	0x000C	读/写
RTC_I2C_SLAVE_ADDR_REG	配置从机地址	0x0010	读/写

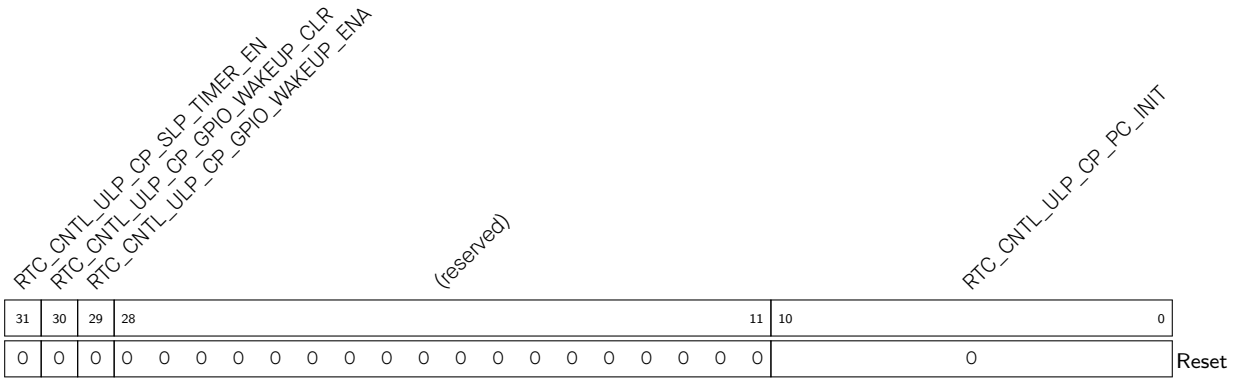
名称	描述	地址	访问
RTC I2C 中断			
RTC_I2C_INT_CLR_REG	清除 RTC I2C 中断	0x0024	只写
RTC_I2C_INT_RAW_REG	RTC I2C 原始中断位	0x0028	只读
RTC_I2C_INT_ST_REG	RTC I2C 中断状态位	0x002C	只读
RTC_I2C_INT_ENA_REG	使能 RTC I2C 中断	0x0030	读/写
RTC I2C 状态寄存器）			
RTC_I2C_DATA_REG	RTC I2C 读数据 (RDDATA)	0x0034	不定
RTC I2C 命令			
RTC_I2C_CMD0_REG	RTC I2C 命令 0	0x0038	不定
RTC_I2C_CMD1_REG	RTC I2C 命令 1	0x003C	不定
RTC_I2C_CMD2_REG	RTC I2C 命令 2	0x0040	不定
RTC_I2C_CMD3_REG	RTC I2C 命令 3	0x0044	不定
RTC_I2C_CMD4_REG	RTC I2C 命令 4	0x0048	不定
RTC_I2C_CMD5_REG	RTC I2C 命令 5	0x004C	不定
RTC_I2C_CMD6_REG	RTC I2C 命令 6	0x0050	不定
RTC_I2C_CMD7_REG	RTC I2C 命令 7	0x0054	不定
RTC_I2C_CMD8_REG	RTC I2C 命令 8	0x0058	不定
RTC_I2C_CMD9_REG	RTC I2C 命令 9	0x005C	不定
RTC_I2C_CMD10_REG	RTC I2C 命令 10	0x0060	不定
RTC_I2C_CMD11_REG	RTC I2C 命令 11	0x0064	不定
RTC_I2C_CMD12_REG	RTC I2C 命令 12	0x0068	不定
RTC_I2C_CMD13_REG	RTC I2C 命令 13	0x006C	不定
RTC_I2C_CMD14_REG	RTC I2C 命令 14	0x0070	不定
RTC_I2C_CMD15_REG	RTC I2C 命令 15	0x0074	不定
版本寄存器			
RTC_I2C_DATE_REG	版本控制寄存器	0x00FC	读/写

2.10 寄存器

2.10.1 ULP (ALWAYS_ON) 寄存器

本小节的所有地址均为相对于低功耗管理模块基地址的地址偏移量（相对地址），具体基地址请见章节 *4 系统和存储器* 中的表 4.3-3。

Register 2.1. RTC_CNTL_ULP_CP_TIMER_REG (0x00FC)



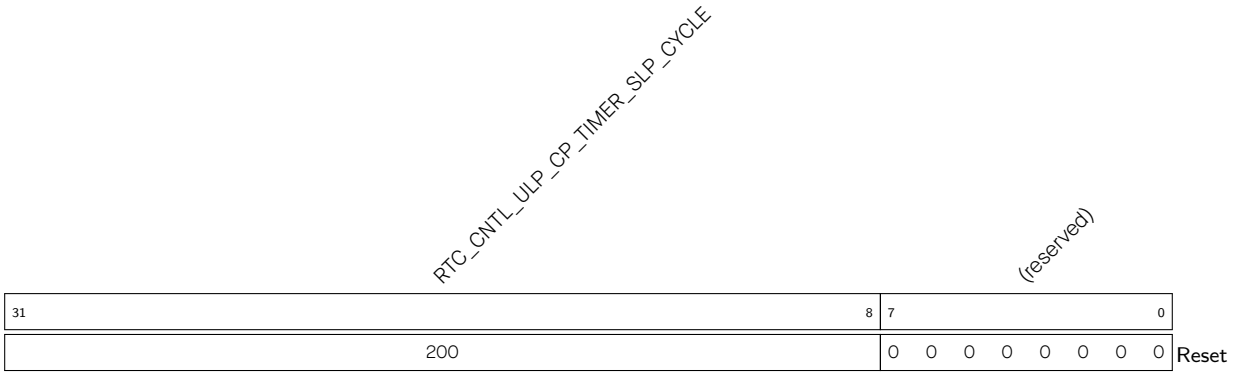
RTC_CNTL_ULP_CP_PC_INIT ULP 协处理器 PC 初始地址。(R/W)

RTC_CNTL_ULP_CP_GPIO_WAKEUP_ENA 开启使用 RTC GPIO 唤醒 ULP 协定时器的选项。(R/W)

RTC_CNTL_ULP_CP_GPIO_WAKEUP_CLR 关闭使用 RTC GPIO 唤醒 ULP 协定时器的选项。(WO)

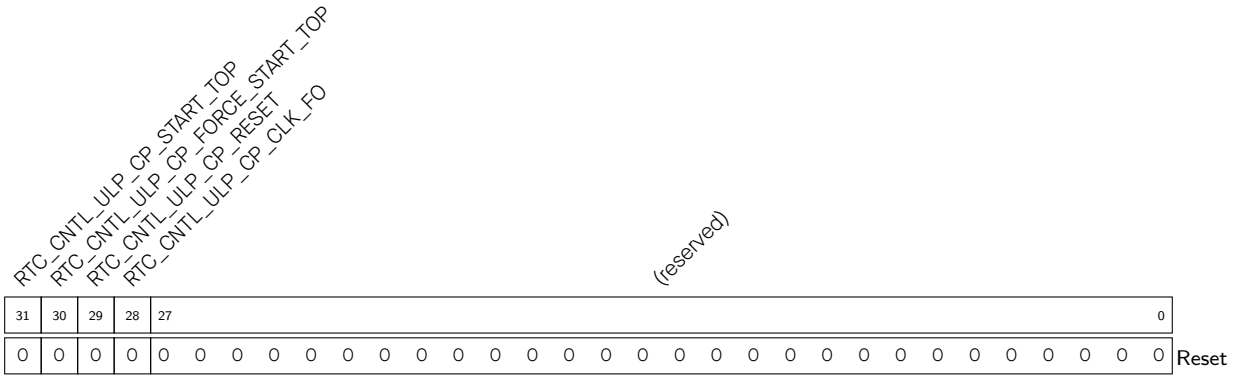
RTC_CNTL_ULP_CP_SLP_TIMER_EN ULP 协处理器定时器使能位。0：关闭硬件定时器；1：使能硬件定时器。(R/W)

Register 2.2. RTC_CNTL_ULP_CP_TIMER_1_REG (0x0134)



RTC_CNTL_ULP_CP_TIMER_SLP_CYCLE 设置 ULP 协处理器定时器的睡眠周期。(R/W)

Register 2.3. RTC_CNTL_ULP_CP_CTRL_REG (0x0100)



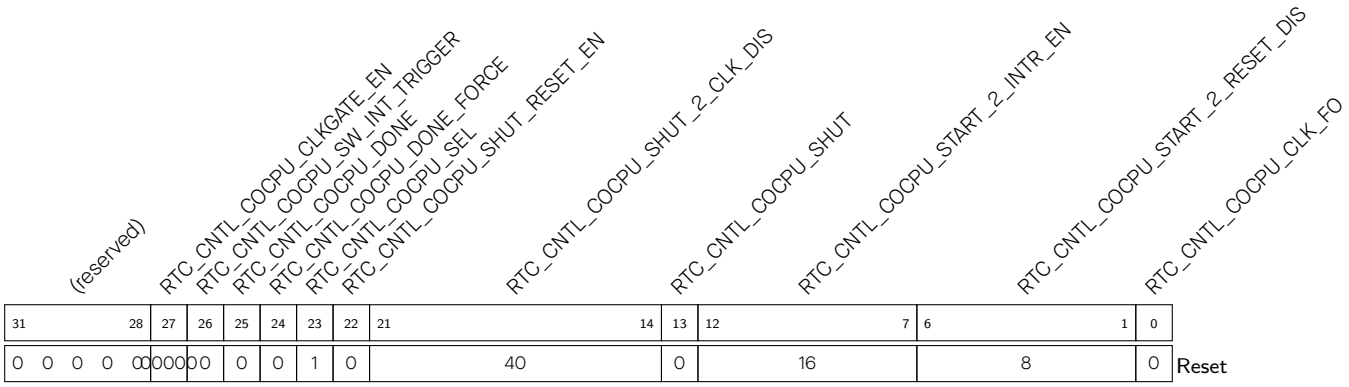
RTC_CNTL_ULP_CP_CLK_FO 强制使能 ULP-FSM 时钟。(R/W)

RTC_CNTL_ULP_CP_RESET ULP-FSM 时钟软件重置。(R/W)

RTC_CNTL_ULP_CP_FORCE_START_TOP 写入 1 时，ULP-FSM 由软件启动。(R/W)

RTC_CNTL_ULP_CP_START_TOP 写入 1 启动 ULP-FSM。(R/W)

Register 2.4. RTC_CNTL_COCPU_CTRL_REG (0x0104)



RTC_CNTL_COCPU_CLK_FO 强制使能 ULP-RISC-V 时钟。(R/W)

RTC_CNTL_COCPU_START_2_RESET_DIS 从 ULP-RISC-V 启动到下拉复位的时间。(R/W)

RTC_CNTL_COCPU_START_2_INTR_EN 从 ULP-RISC-V 启动到发出 RISCV_START_INT 中断的时间。(R/W)

RTC_CNTL_COCPU_SHUT 关闭 ULP-RISC-V。(R/W)

RTC_CNTL_COCPU_SHUT_2_CLK_DIS 关闭 ULP-RISC-V 至关闭时钟之间的延迟时间。(R/W)

RTC_CNTL_COCPU_SHUT_RESET_EN 复位 ULP-RISC-V。(R/W)

RTC_CNTL_COCPU_SEL 选择要使用的协处理器。0：选择使用 ULP-RISC-V；1：选择使用 ULP-FSM。(R/W)

RTC_CNTL_COCPU_DONE_FORCE 0：选择 ULP-FSM 的完成信号；1：选择 ULP-RISC-V 的完成信号。(R/W)

RTC_CNTL_COCPU_DONE DONE 信号，置 1 则 ULP-RISC-V 进入 HALT，同时定时器开始计数。(R/W)

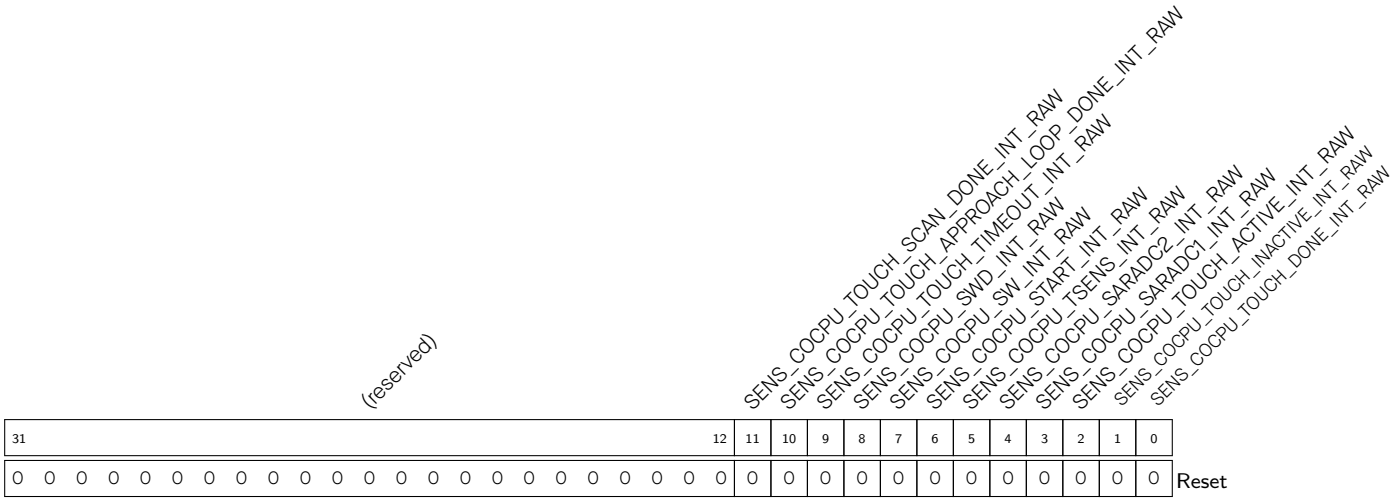
RTC_CNTL_COCPU_SW_INT_TRIGGER 触发 ULP-RISC-V 寄存器中断。(WO)

RTC_CNTL_COCPU_CLKGATE_EN 使能 ULP-RICS-V 时钟门控。(WO)

2.10.2 ULP (RTC_PERI) 寄存器

本小节的所有地址均为相对于低功耗管理模块基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 2.5. SENS_ SAR_COCPU_INT_RAW_REG (0x00E8)



SENS_COCPU_TOUCH_DONE_INT_RAW TOUCH_DONE_INT 的原始中断位。(RO)

SENS_COCPU_TOUCH_INACTIVE_INT_RAW TOUCH_INACTIVE_INT 的原始中断位。(RO)

SENS_COCPU_TOUCH_ACTIVE_INT_RAW TOUCH_ACTIVE_INT 的原始中断位。(RO)

SENS_COCPU_SARADC1_INT_RAW SARADC1_DONE_INT 的原始中断位。(RO)

SENS_COCPU_SARADC2_INT_RAW SARADC2_DONE_INT 的原始中断位。(RO)

SENS_COCPU_TSENS_INT_RAW TSENS_DONE_INT 的原始中断位。(RO)

SENS_COCPU_START_INT_RAW RISCV_START_INT 的原始中断位。(RO)

SENS_COCPU_SW_INT_RAW SW_INT 的原始中断位。(RO)

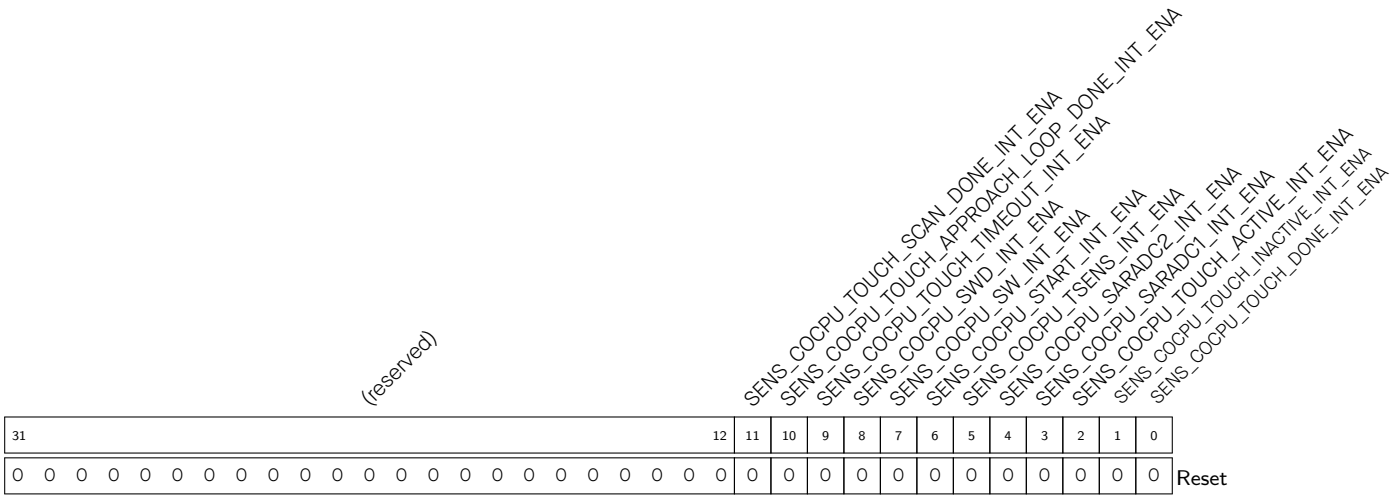
SENS_COCPU_SWD_INT_RAW SWD_INT 的原始中断位。(RO)

SENS_COCPU_TOUCH_TIMEOUT_INT_RAW TOUCH_TIME_OUT 的原始中断位。(RO)

SENS_COCPU_TOUCH_APPROACH_LOOP_DONE_INT_RAW TOUCH_APPROACH_LOOP_DONE_INT 的原始中断位。(RO)

SENS_COCPU_TOUCH_SCAN_DONE_INT_RAW TOUCH_SCAN_DONE_INT 的原始中断位。(RO)

Register 2.6. SENS_SAR_COCPU_INT_ENA_REG (0x00EC)



SENS_COCPU_TOUCH_DONE_INT_ENA TOUCH_DONE_INT 的中断使能位。(R/W)

SENS_COCPU_TOUCH_INACTIVE_INT_ENA TOUCH_INACTIVE_INT 的中断使能位。(R/W)

SENS_COCPU_TOUCH_ACTIVE_INT_ENA TOUCH_ACTIVE_INT 的中断使能位。(R/W)

SENS_COCPU_SARADC1_INT_ENA SARADC1_DONE_INT 的中断使能位。(R/W)

SENS_COCPU_SARADC2_INT_ENA SARADC2_DONE_INT 的中断使能位。(R/W)

SENS_COCPU_TSENS_INT_ENA TSENS_DONE_INT 的中断使能位。(R/W)

SENS_COCPU_START_INT_ENA RISCV_START_INT 的中断使能位。(R/W)

SENS_COCPU_SW_INT_ENA SW_INT 的中断使能位。(R/W)

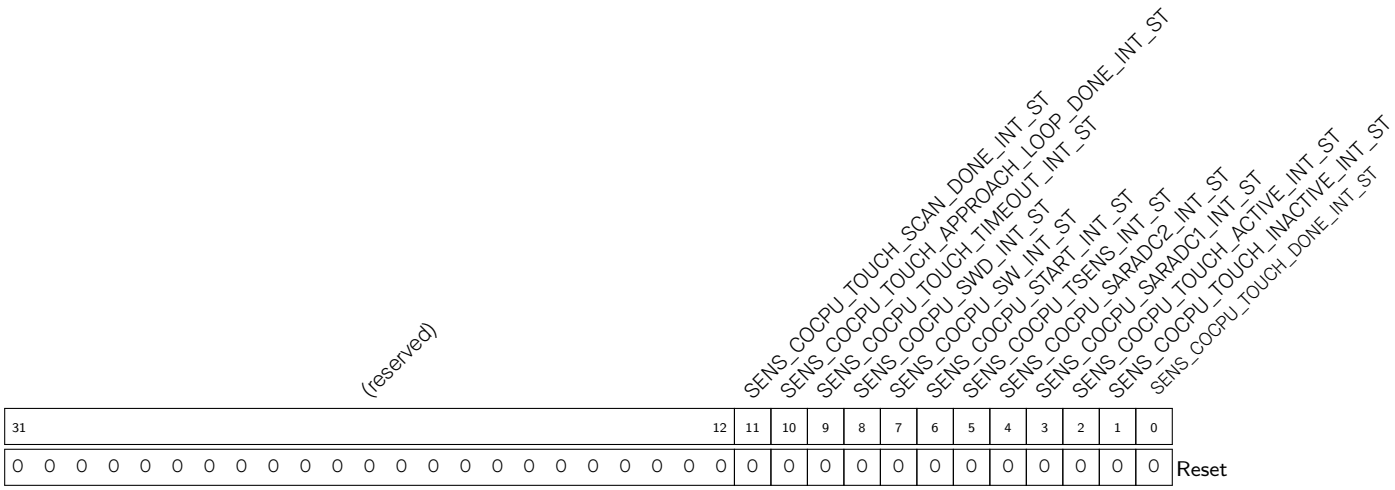
SENS_COCPU_SWD_INT_ENA SWD_INT 的中断使能位。(R/W)

SENS_COCPU_TOUCH_TIMEOUT_INT_ENA TOUCH_TIME_OUT 的中断使能位。(R/W)

SENS_COCPU_TOUCH_APPROACH_LOOP_DONE_INT_ENA TOUCH_APPROACH_LOOP_DONE_INT 的中断使能位。(R/W)

SENS_COCPU_TOUCH_SCAN_DONE_INT_ENA TOUCH_SCAN_DONE_INT 的中断使能位。(R/W)

Register 2.7. SENS_SAR_COCPU_INT_ST_REG (0x00F0)



SENS_COCPU_TOUCH_DONE_INT_ST TOUCH_DONE_INT 的中断状态位。(RO)

SENS_COCPU_TOUCH_INACTIVE_INT_ST TOUCH_INACTIVE_INT 的中断状态位。(RO)

SENS_COCPU_TOUCH_ACTIVE_INT_ST TOUCH_ACTIVE_INT 的中断状态位。(RO)

SENS_COCPU_SARADC1_INT_ST SARADC1_DONE_INT 的中断状态位。(RO)

SENS_COCPU_SARADC2_INT_ST SARADC2_DONE_INT 的中断状态位。(RO)

SENS_COCPU_TSENS_INT_ST TSENS_DONE_INT 的中断状态位。(RO)

SENS_COCPU_START_INT_ST RISCV_START_INT 的中断状态位。(RO)

SENS_COCPU_SW_INT_ST SW_INT 的中断状态位。(RO)

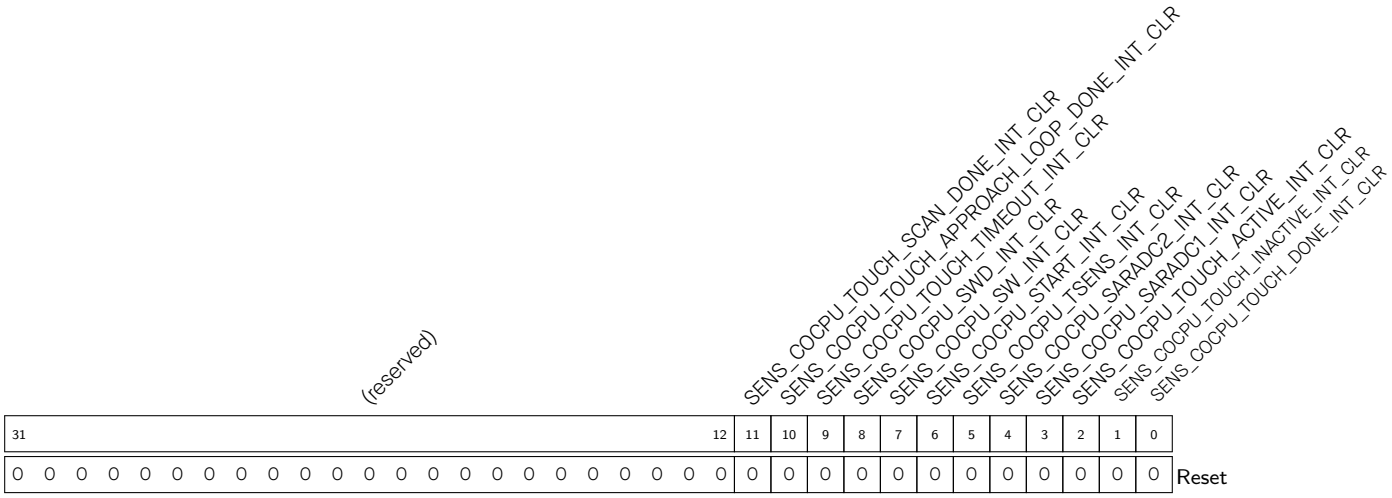
SENS_COCPU_SWD_INT_ST SWD_INT 的中断状态位。(RO)

SENS_COCPU_TOUCH_TIMEOUT_INT_ST TOUCH_TIME_OUT 的中断状态位。(RO)

SENS_COCPU_TOUCH_APPROACH_LOOP_DONE_INT_ST TOUCH_APPROACH_LOOP_DONE_INT 的中断状态位。(RO)

SENS_COCPU_TOUCH_SCAN_DONE_INT_ST TOUCH_SCAN_DONE_INT 的中断状态位。(RO)

Register 2.8. SENS_SAR_COCPU_INT_CLR_REG (0x00F4)



SENS_COCPU_TOUCH_DONE_INT_CLR TOUCH_DONE_INT 的中断清零位。(WO)

SENS_COCPU_TOUCH_INACTIVE_INT_CLR TOUCH_INACTIVE_INT 的中断清零位。(WO)

SENS_COCPU_TOUCH_ACTIVE_INT_CLR TOUCH_ACTIVE_INT 的中断清零位。(WO)

SENS_COCPU_SARADC1_INT_CLR SARADC1_DONE_INT 的中断清零位。(WO)

SENS_COCPU_SARADC2_INT_CLR SARADC2_DONE_INT 的中断清零位。(WO)

SENS_COCPU_TSENS_INT_CLR TSENS_DONE_INT 的中断清零位。(WO)

SENS_COCPU_START_INT_CLR RISCV_START_INT 的中断清零位。(WO)

SENS_COCPU_SW_INT_CLR SW_INT 的中断清零位。(WO)

SENS_COCPU_SWD_INT_CLR SWD_INT 的中断清零位。(WO)

SENS_COCPU_TOUCH_TIMEOUT_INT_CLR TOUCH_TIME_OUT 的中断清零位。(WO)

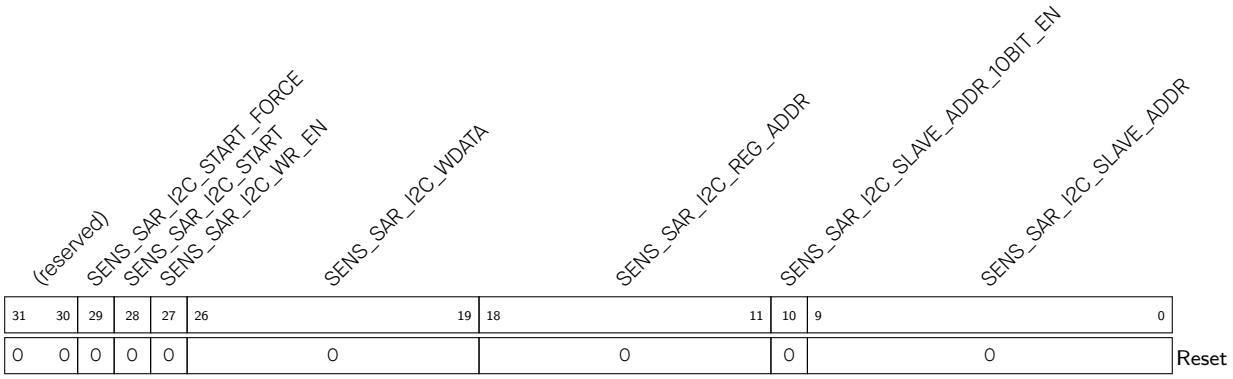
SENS_COCPU_TOUCH_APPROACH_LOOP_DONE_INT_CLR TOUCH_APPROACH_LOOP_DONE_INT 的中断清零位。(WO)

SENS_COCPU_TOUCH_SCAN_DONE_INT_CLR TOUCH_SCAN_DONE_INT 的中断清零位。(WO)

2.10.3 RTC I2C (RTC_PERI) 寄存器

本小节的所有地址均为相对于低功耗管理模块基地址 + 0x0800 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 2.9. SENS_SAR_I2C_CTRL_REG (0x0058)



SENS_SAR_I2C_SLAVE_ADDR 配置从机地址。(R/W)

SENS_SAR_I2C_SLAVE_ADDR_10BIT_EN 配置是否将从机地址扩展为 10 位。

0：不扩展

1：扩展

(R/W)

SENS_SAR_I2C_REG_ADDR 配置寄存器地址。(R/W)

SENS_SAR_I2C_WDATA 配置写入的数据。(R/W)

SENS_SAR_I2C_WR_EN 配置是否使用写命令。

0：使用读命令

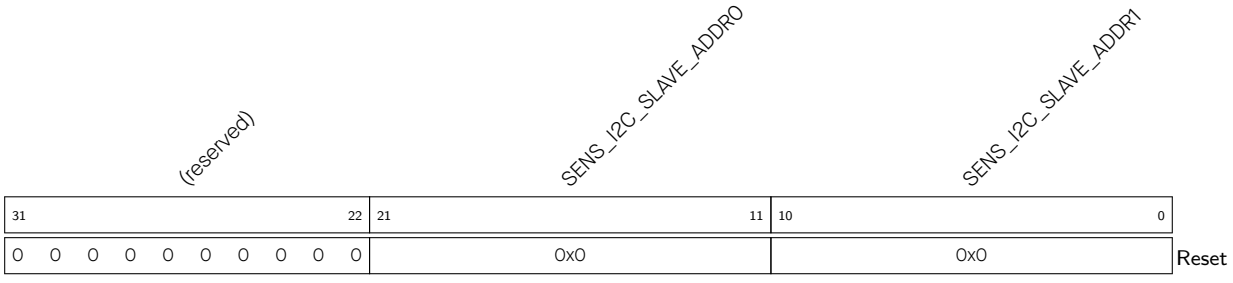
1：使用写命令

(R/W)

SENS_SAR_I2C_START 启动 RTC I2C，仅当 SENS_SAR_I2C_START_FORCE = 1 时有效。(R/W)

SENS_SAR_I2C_START_FORCE RTC I2C 启动模式。0：由 FSM 启动；1：由软件启动。(R/W)

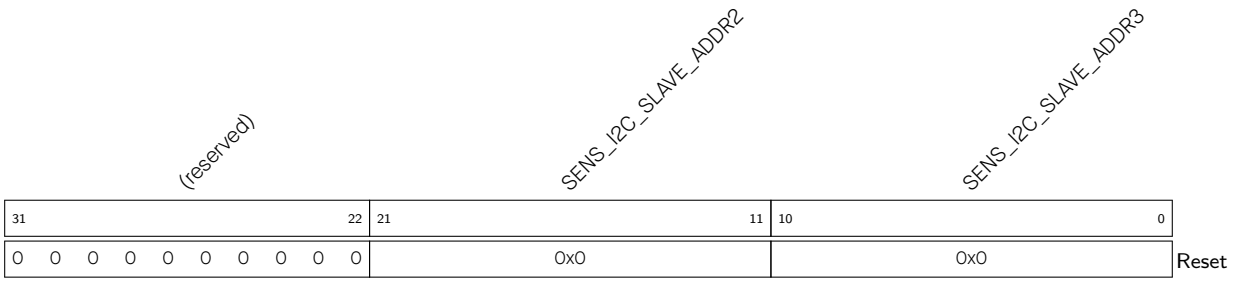
Register 2.10. SENS_SAR_SLAVE_ADDR1_REG (0x0040)



SENS_I2C_SLAVE_ADDR1 RTC I2C 从机地址 1。(R/W)

SENS_I2C_SLAVE_ADDR0 RTC I2C 从机地址 0。(R/W)

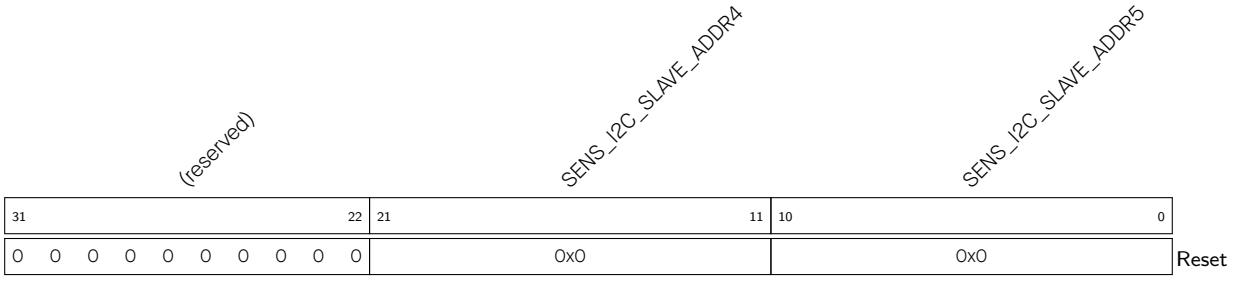
Register 2.11. SENS_SAR_SLAVE_ADDR2_REG (0x0044)



SENS_I2C_SLAVE_ADDR3 RTC I2C 从机地址 3。(R/W)

SENS_I2C_SLAVE_ADDR2 RTC I2C 从机地址 2。(R/W)

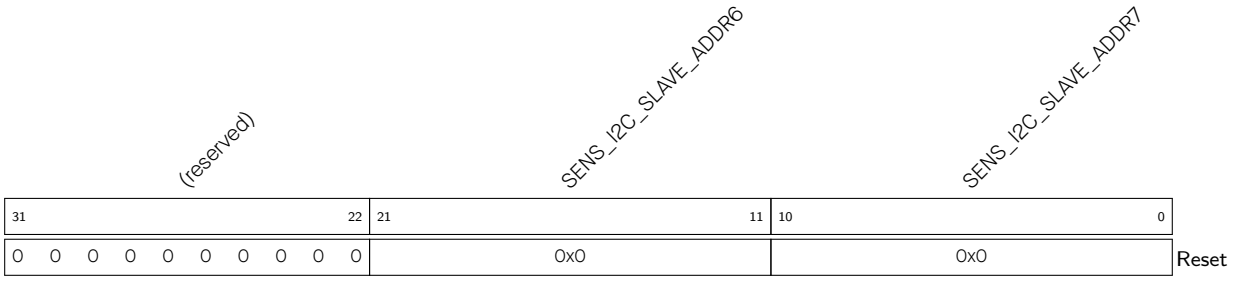
Register 2.12. SENS_SAR_SLAVE_ADDR3_REG (0x0048)



SENS_I2C_SLAVE_ADDR5 RTC I2C 从机地址 5。(R/W)

SENS_I2C_SLAVE_ADDR4 RTC I2C 从机地址 4。(R/W)

Register 2.13. SENS_SAR_SLAVE_ADDR4_REG (0x004C)



SENS_I2C_SLAVE_ADDR7 RTC I2C 从机地址 7。(R/W)

SENS_I2C_SLAVE_ADDR6 RTC I2C 从机地址 6。(R/W)

2.10.4 RTC I2C (I2C) 寄存器

本小节的所有地址均为相对于低功耗管理模块基地址 + 0x0C00 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 2.14. RTC_I2C_SCL_LOW_REG (0x0000)



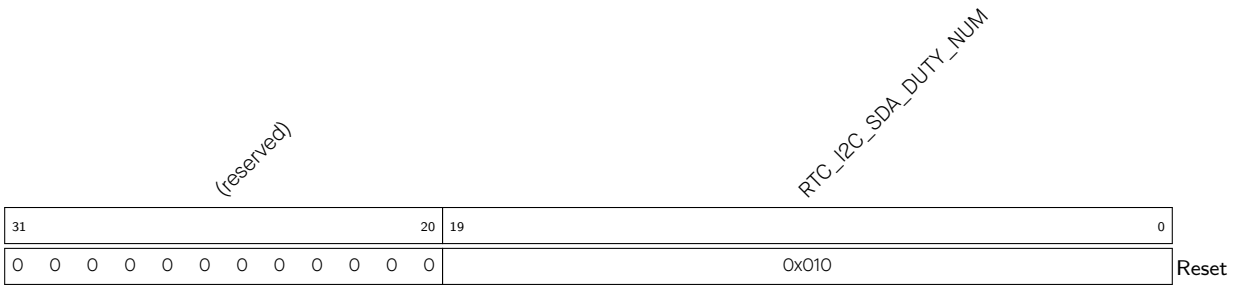
RTC_I2C_SCL_LOW_PERIOD 配置 SCL 低电平周期。(R/W)

Register 2.15. RTC_I2C_SCL_HIGH_REG (0x0014)



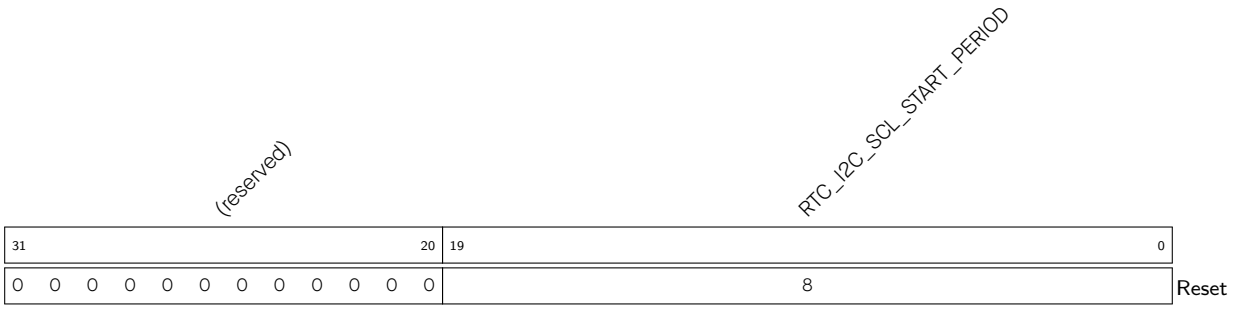
RTC_I2C_SCL_HIGH_PERIOD 配置 SCL 高电平周期。(R/W)

Register 2.16. RTC_I2C_SDA_DUTY_REG (0x0018)



RTC_I2C_SDA_DUTY_NUM SCL 下降沿与 SDA 切换之间的时钟周期数。(R/W)

Register 2.17. RTC_I2C_SCL_START_PERIOD_REG (0x001C)



RTC_I2C_SCL_START_PERIOD RTC I2C START 信号发出后，SDA 信号拉低到 SCL 信号拉低需等待的时间间隔。(R/W)

Register 2.18. RTC_I2C_SCL_STOP_PERIOD_REG (0x0020)



RTC_I2C_SCL_STOP_PERIOD RTC I2C STOP 信号发出前，SDA 信号拉高到 SCL 信号拉高需等待的时间间隔。(R/W)

Register 2.19. RTC_I2C_CTRL_REG (0x0004)

	(reserved)	RTC_I2C_RESET	RTC_I2C_CTRL_CLK_GATE_EN	(reserved)	RTC_I2C_RX_LSB_FIRST	RTC_I2C_TX_LSB_FIRST	RTC_I2C_TRANS_START	RTC_I2C_MS_MODE	RTC_I2C_SCL_FORCE_OUT	RTC_I2C_SDA_FORCE_OUT
Bit	31	30	29	28 … 6	5	4	3	2	1	0
Reset	0	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0	0	0	0	0	0

RTC_I2C_SDA_FORCE_OUT SDA 输出模式配置。0：开漏输出；1：推挽输出。(R/W)

RTC_I2C_SCL_FORCE_OUT SCL 输出模式配置。0：开漏输出；1：推挽输出。(R/W)

RTC_I2C_MS_MODE 置位此位，将 RTC I2C 配置为主机。

RTC_I2C_TRANS_START 置位此位，RTC I2C 开始发送数据。(R/W)

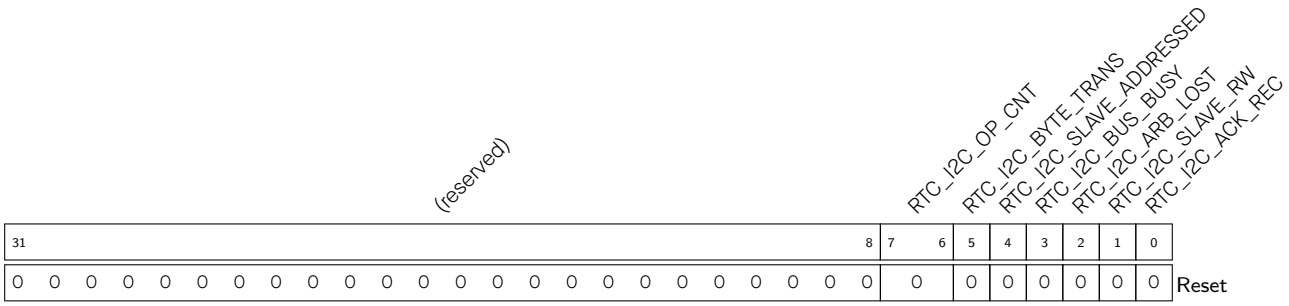
RTC_I2C_TX_LSB_FIRST 用于控制待发送数据的发送模式。0：从最高有效位发送数据；1：从最低有效位开始发送数据。(R/W)

RTC_I2C_RX_LSB_FIRST 用于控制接收数据的存储模式。0：从最高有效位开始接收数据；1：从最低有效位开始接收数据。(R/W)

RTC_I2C_CTRL_CLK_GATE_EN RTC I2C 控制器时钟门控。(R/W)

RTC_I2C_RESET 置位此位，RTC I2C 软件重置。(R/W)

Register 2.20. RTC_I2C_STATUS_REG (0x0008)



RTC_I2C_ACK_REC ACK 电平值。0：ACK；1：NACK。(RO)

RTC_I2C_SLAVE_RW 0：主机向从机写入数据；1：主机读取从机数据。(RO)

RTC_I2C_ARB_LOST RTC I2C 不控制 SCL 线时，该寄存器变为 1。(RO)

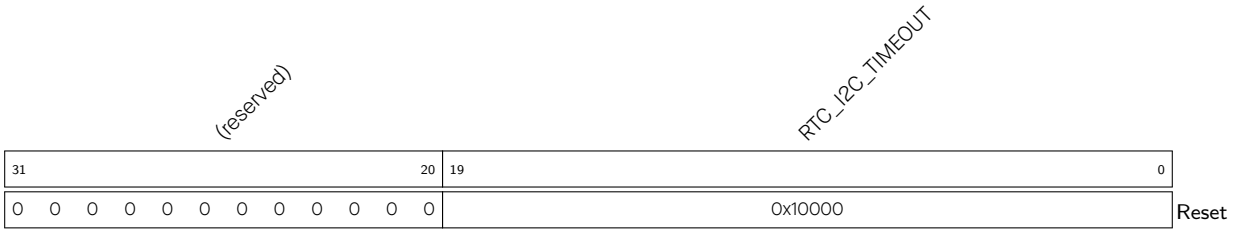
RTC_I2C_BUS_BUSY 0：RTC I2C 总线处于空闲状态；1：RTC I2C 总线正在传输数据。(RO)

RTC_I2C_SLAVE_ADDRESSED 主机发送地址与从机地址匹配时，该位翻转为高电平。(RO)

RTC_I2C_BYTE_TRANS 传输一个字节后，该位变为 1。(RO)

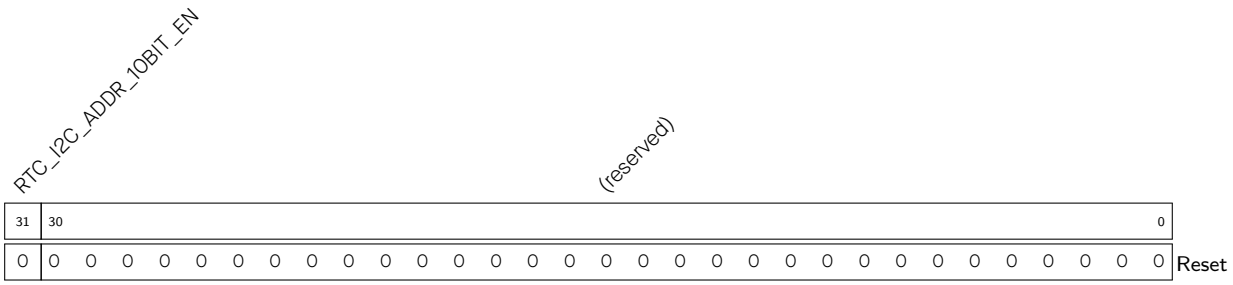
RTC_I2C_OP_CNT 表示正在执行的操作。(RO)

Register 2.21. RTC_I2C_TIMEOUT_REG (0x000C)



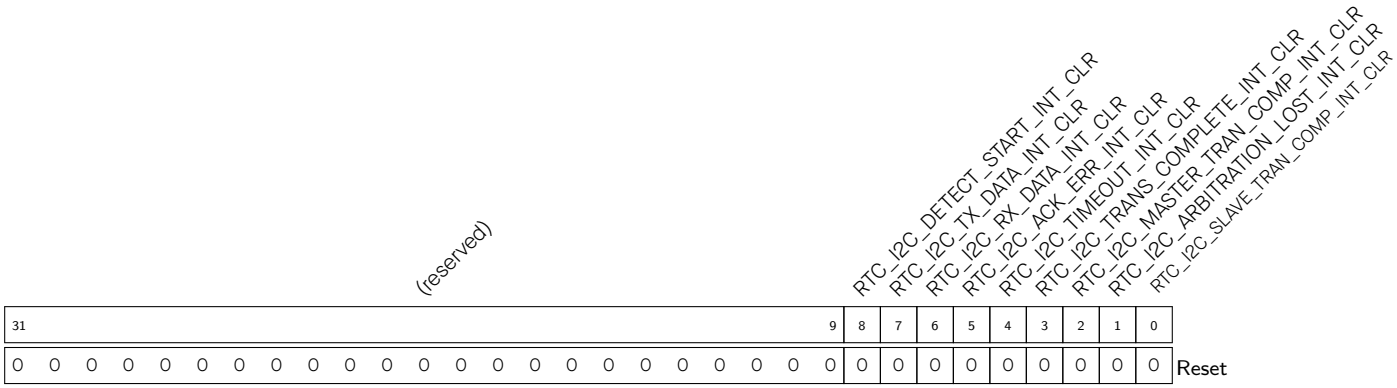
RTC_I2C_TIMEOUT 超时阈值。(R/W)

Register 2.22. RTC_I2C_SLAVE_ADDR_REG (0x0010)



RTC_I2C_ADDR_10BIT_EN 用于在主机模式下使能从机的 10 位寻址模式。(R/W)

Register 2.23. RTC_I2C_INT_CLR_REG (0x0024)



RTC_I2C_SLAVE_TRAN_COMP_INT_CLR RTC_I2C_SLAVE_TRAN_COMP_INT 中断清零位。(WO)

RTC_I2C_ARBITRATION_LOST_INT_CLR RTC_I2C_ARBITRATION_LOST_INT 中断清零位。(WO)

RTC_I2C_MASTER_TRAN_COMP_INT_CLR RTC_I2C_MASTER_TRAN_COMP_INT 中断清零位。(WO)

RTC_I2C_TRANS_COMPLETE_INT_CLR RTC_I2C_TRANS_COMPLETE_INT 中断清零位。(WO)

RTC_I2C_TIMEOUT_INT_CLR RTC_I2C_TIME_OUT_INT 中断清零位。(WO)

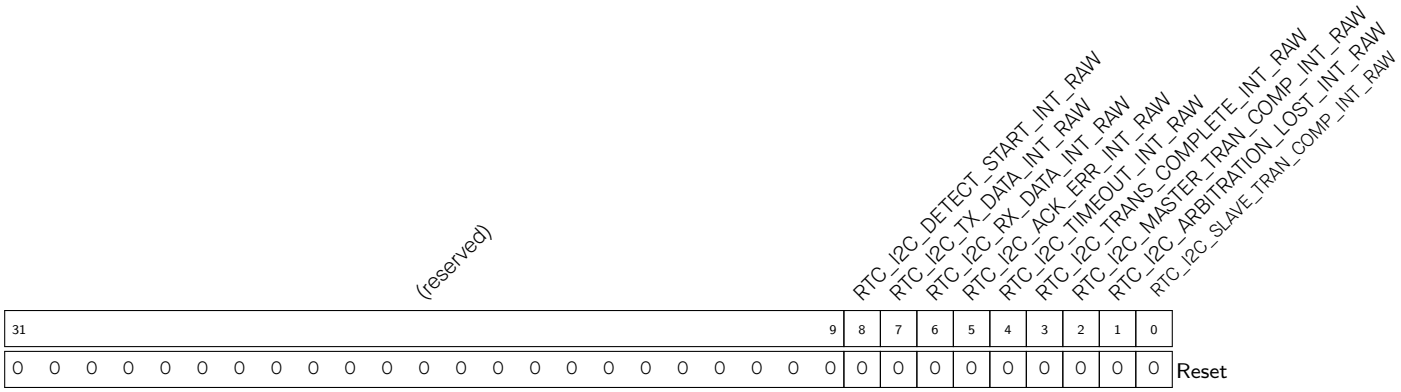
RTC_I2C_ACK_ERR_INT_CLR RTC_I2C_ACK_ERR_INT 中断清零位。(WO)

RTC_I2C_RX_DATA_INT_CLR RTC_I2C_RX_DATA_INT 中断清零位。(WO)

RTC_I2C_TX_DATA_INT_CLR RTC_I2C_TX_DATA_INT 中断清零位。(WO)

RTC_I2C_DETECT_START_INT_CLR RTC_I2C_DETECT_START_INT 中断清零位。(WO)

Register 2.24. RTC_I2C_INT_RAW_REG (0x0028)



RTC_I2C_SLAVE_TRAN_COMP_INT_RAW RTC_I2C_SLAVE_TRAN_COMP_INT 原始中断位。(RO)

RTC_I2C_ARBITRATION_LOST_INT_RAW RTC_I2C_ARBITRATION_LOST_INT 原始中断位。(RO)

RTC_I2C_MASTER_TRAN_COMP_INT_RAW RTC_I2C_MASTER_TRAN_COMP_INT 原始中断位。(RO)

RTC_I2C_TRANS_COMPLETE_INT_RAW RTC_I2C_TRANS_COMPLETE_INT 原始中断位。(RO)

RTC_I2C_TIMEOUT_INT_RAW RTC_I2C_TIME_OUT_INT 原始中断位。(RO)

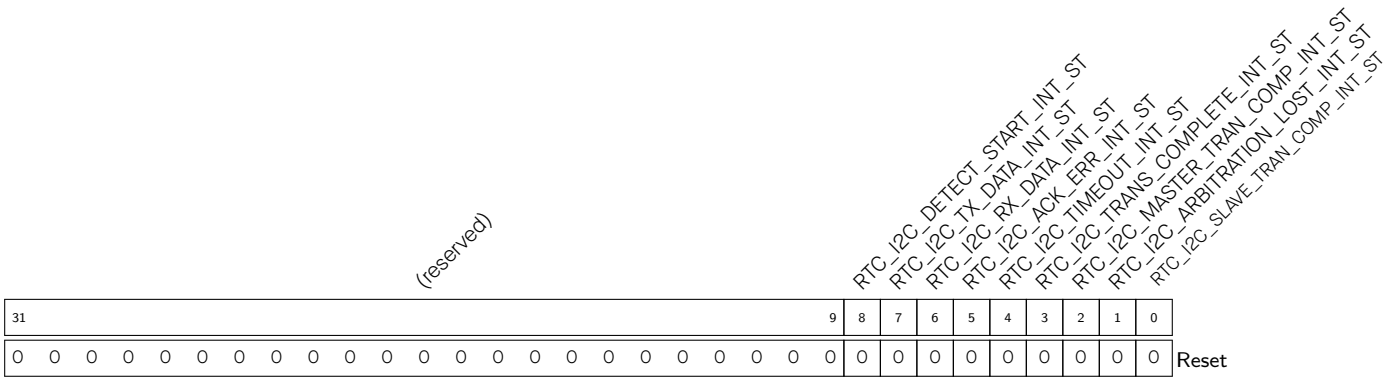
RTC_I2C_ACK_ERR_INT_RAW RTC_I2C_ACK_ERR_INT 原始中断位。(RO)

RTC_I2C_RX_DATA_INT_RAW RTC_I2C_RX_DATA_INT 原始中断位。(RO)

RTC_I2C_TX_DATA_INT_RAW RTC_I2C_TX_DATA_INT 原始中断位。(RO)

RTC_I2C_DETECT_START_INT_RAW RTC_I2C_DETECT_START_INT 原始中断位。(RO)

Register 2.25. RTC_I2C_INT_ST_REG (0x002C)



RTC_I2C_SLAVE_TRAN_COMP_INT_ST RTC_I2C_SLAVE_TRAN_COMP_INT 中断状态位。(RO)

RTC_I2C_ARBITRATION_LOST_INT_ST RTC_I2C_ARBITRATION_LOST_INT 中断状态位。(RO)

RTC_I2C_MASTER_TRAN_COMP_INT_ST RTC_I2C_MASTER_TRAN_COMP_INT 中断状态位。(RO)

RTC_I2C_TRANS_COMPLETE_INT_ST RTC_I2C_TRANS_COMPLETE_INT 中断状态位。(RO)

RTC_I2C_TIMEOUT_INT_ST RTC_I2C_TIME_OUT_INT 中断状态位。(RO)

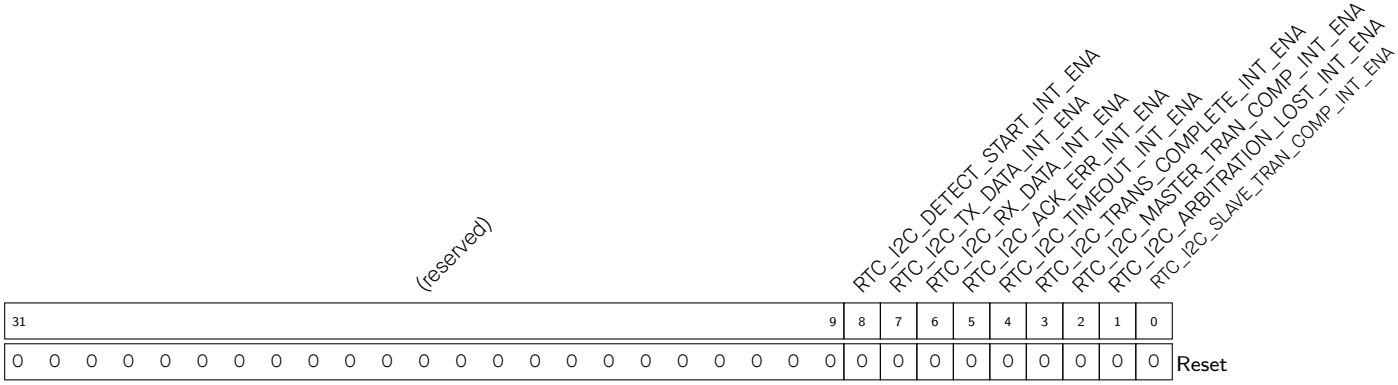
RTC_I2C_ACK_ERR_INT_ST RTC_I2C_ACK_ERR_INT 中断状态位。(RO)

RTC_I2C_RX_DATA_INT_ST RTC_I2C_RX_DATA_INT 中断状态位。(RO)

RTC_I2C_TX_DATA_INT_ST RTC_I2C_TX_DATA_INT 中断状态位。(RO)

RTC_I2C_DETECT_START_INT_ST RTC_I2C_DETECT_START_INT 中断状态位。(RO)

Register 2.26. RTC_I2C_INT_ENA_REG (0x0030)



RTC_I2C_SLAVE_TRAN_COMP_INT_ENA RTC_I2C_SLAVE_TRAN_COMP_INT 中断使能位。(R/W)

RTC_I2C_ARBITRATION_LOST_INT_ENA RTC_I2C_ARBITRATION_LOST_INT 中断使能位。(R/W)

RTC_I2C_MASTER_TRAN_COMP_INT_ENA RTC_I2C_MASTER_TRAN_COMP_INT 中断使能位。(R/W)

RTC_I2C_TRANS_COMPLETE_INT_ENA RTC_I2C_TRANS_COMPLETE_INT 中断使能位。(R/W)

RTC_I2C_TIMEOUT_INT_ENA RTC_I2C_TIME_OUT_INT 中断使能位。(R/W)

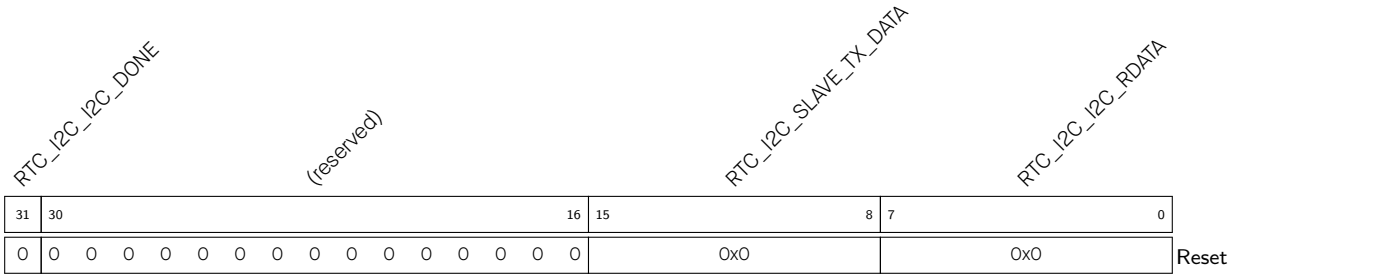
RTC_I2C_ACK_ERR_INT_ENA RTC_I2C_ACK_ERR_INT 中断使能位。(R/W)

RTC_I2C_RX_DATA_INT_ENA RTC_I2C_RX_DATA_INT 中断使能位。(R/W)

RTC_I2C_TX_DATA_INT_ENA RTC_I2C_TX_DATA_INT 中断使能位。(R/W)

RTC_I2C_DETECT_START_INT_ENA RTC_I2C_DETECT_START_INT 中断使能位。(R/W)

Register 2.27. RTC_I2C_DATA_REG (0x0034)

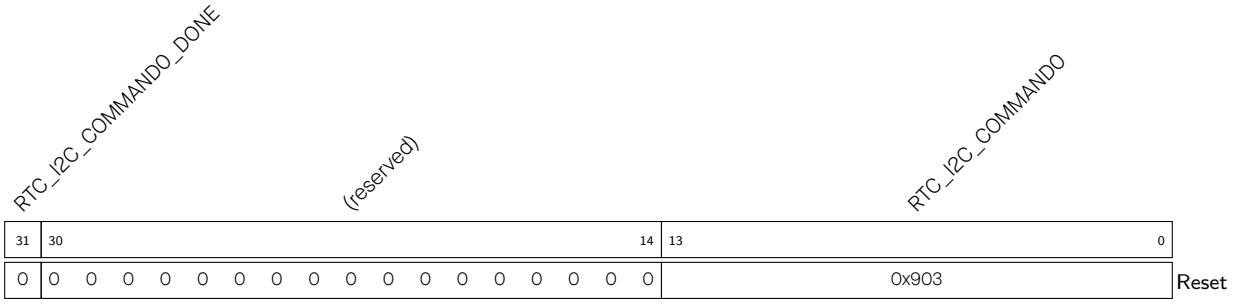


RTC_I2C_RDATA RTC I2C 主机接收的数据。(RO)

RTC_I2C_SLAVE_TX_DATA RTC I2C 从机发送的数据。(R/W)

RTC_I2C_DONE RTC I2C 数据传输结束。(RO)

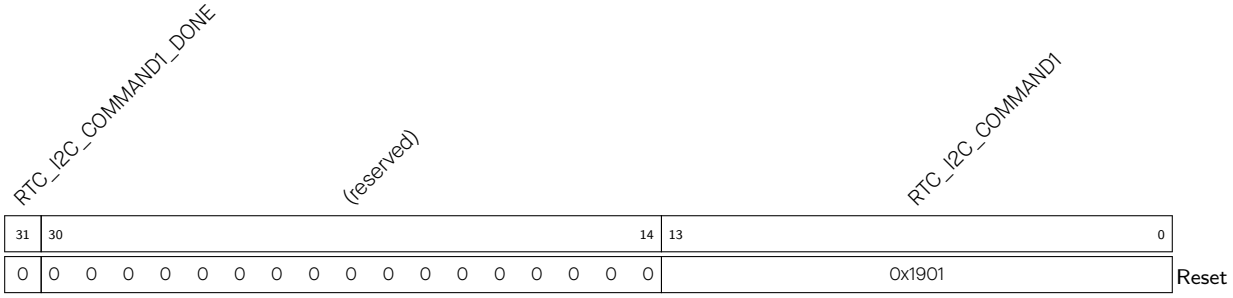
Register 2.28. RTC_I2C_CMD0_REG (0x0038)



RTC_I2C_COMMAND0 命令 0，详细信息可参考 I2C 控制器章节中的 I2C_COMD0_REG 寄存器。(R/W)

RTC_I2C_COMMAND0_DONE 命令 0 完成时，该位翻转为高电平。(RO)

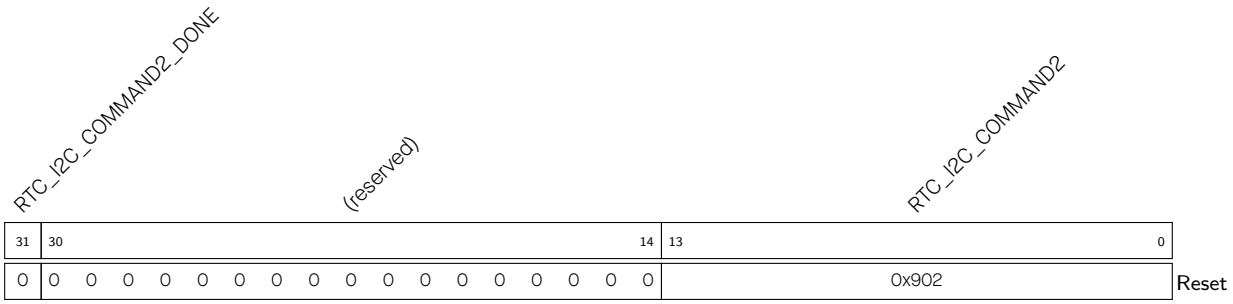
Register 2.29. RTC_I2C_CMD1_REG (0x003C)



RTC_I2C_COMMAND1 命令 1，详细信息可参考 I2C 控制器章节中的 I2C_COMD1_REG 寄存器。(R/W)

RTC_I2C_COMMAND1_DONE 命令 1 完成时，该位翻转为高电平。(RO)

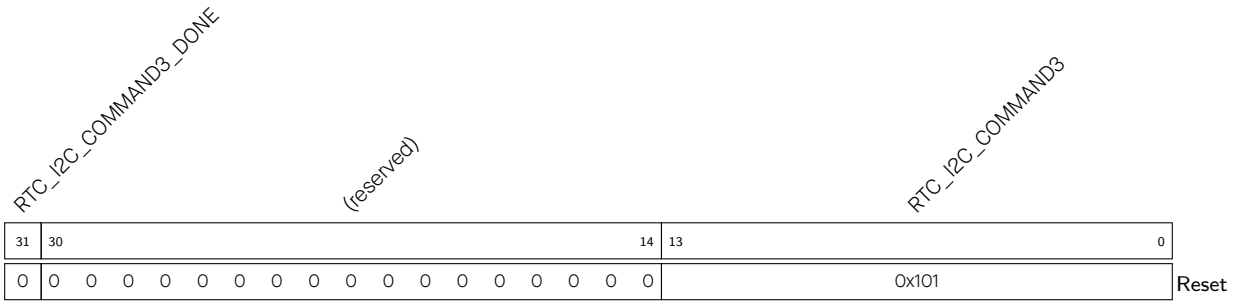
Register 2.30. RTC_I2C_CMD2_REG (0x0040)



RTC_I2C_COMMAND2 命令 2，详细信息可参考 I2C 控制器章节中的 I2C_COMD2_REG 寄存器。(R/W)

RTC_I2C_COMMAND2_DONE 命令 2 完成时，该位翻转为高电平。(RO)

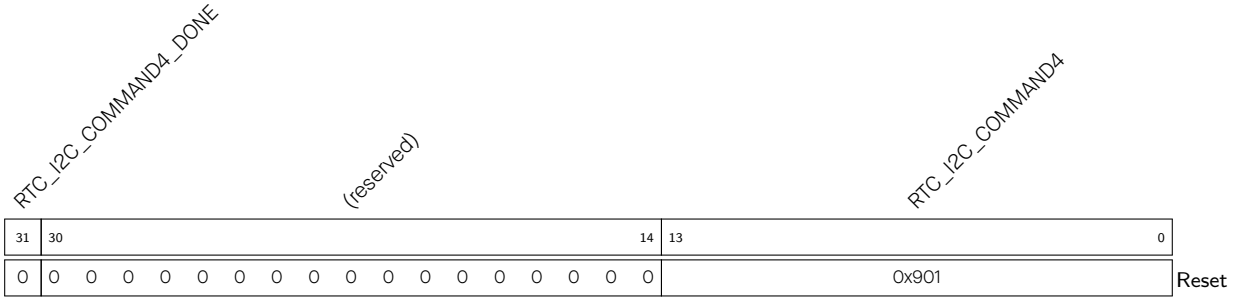
Register 2.31. RTC_I2C_CMD3_REG (0x0044)



RTC_I2C_COMMAND3 命令 3，详细信息可参考 I2C 控制器章节中的 I2C_COMD3_REG 寄存器。(R/W)

RTC_I2C_COMMAND3_DONE 命令 3 完成时，该位翻转为高电平。(RO)

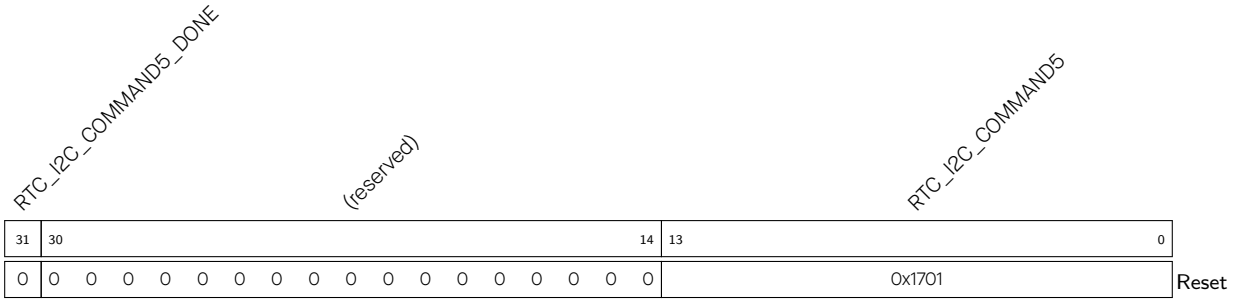
Register 2.32. RTC_I2C_CMD4_REG (0x0048)



RTC_I2C_COMMAND4 命令 4，详细信息可参考 I2C 控制器章节中的 I2C_COMD4_REG 寄存器。(R/W)

RTC_I2C_COMMAND4_DONE 命令 4 完成时，该位翻转为高电平。(RO)

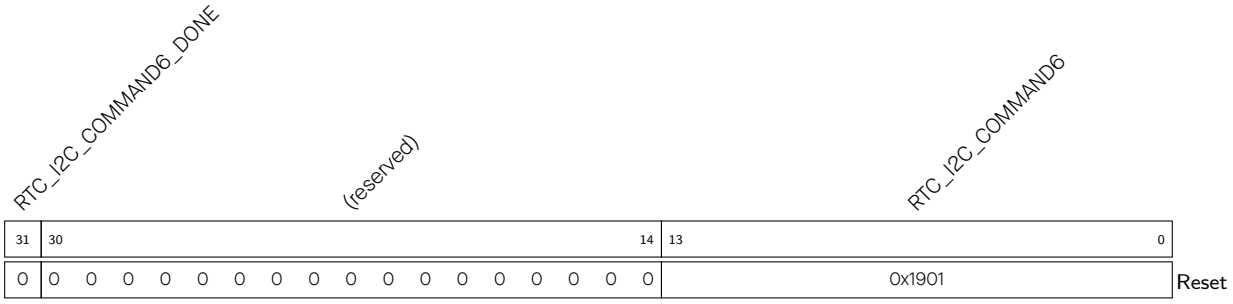
Register 2.33. RTC_I2C_CMD5_REG (0x004C)



RTC_I2C_COMMAND5 命令 5，详细信息可参考 I2C 控制器章节中的 I2C_COMD5_REG 寄存器。(R/W)

RTC_I2C_COMMAND5_DONE 命令 5 完成时，该位翻转为高电平。(RO)

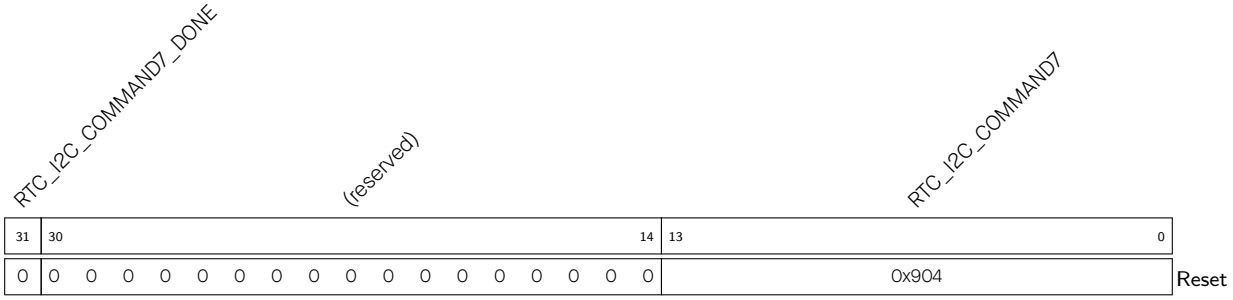
Register 2.34. RTC_I2C_CMD6_REG (0x0050)



RTC_I2C_COMMAND6 命令 6，详细信息可参考 I2C 控制器章节中的 I2C_COMD6_REG 寄存器。(R/W)

RTC_I2C_COMMAND6_DONE 命令 6 完成时，该位翻转为高电平。(RO)

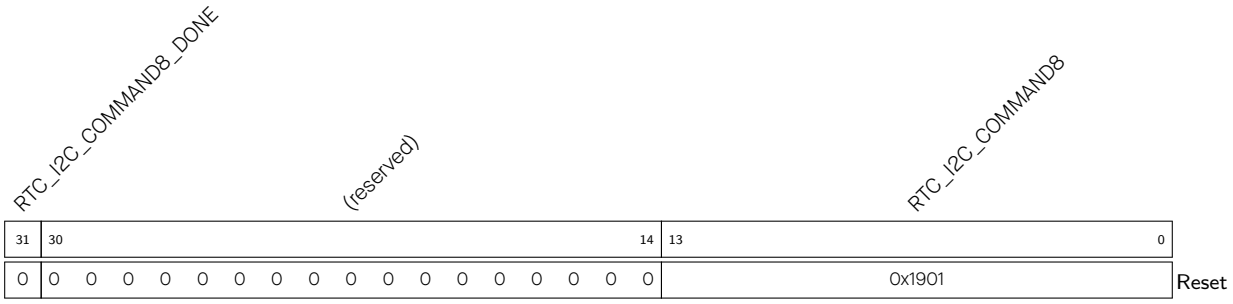
Register 2.35. RTC_I2C_CMD7_REG (0x0054)



RTC_I2C_COMMAND7 命令 7，详细信息可参考 I2C 控制器章节中的 I2C_COMD7_REG 寄存器。(R/W)

RTC_I2C_COMMAND7_DONE 命令 7 完成时，该位翻转为高电平。(RO)

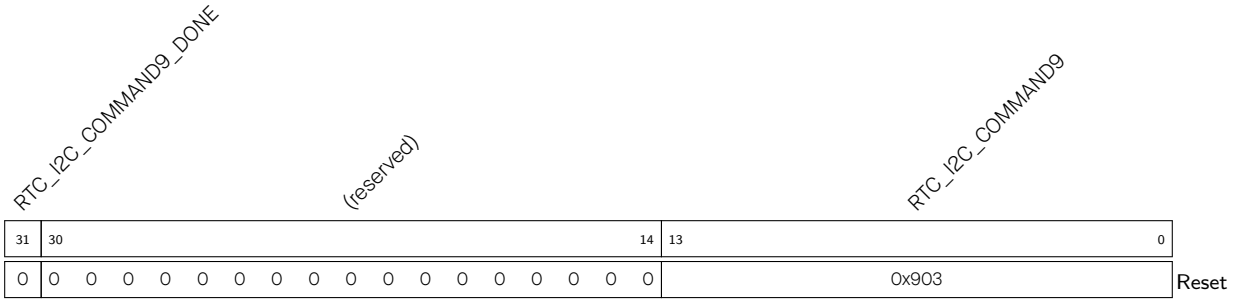
Register 2.36. RTC_I2C_CMD8_REG (0x0058)



RTC_I2C_COMMAND8 命令 8，详细信息可参考 I2C 控制器章节中的 I2C_COMD8_REG 寄存器。(R/W)

RTC_I2C_COMMAND8_DONE 命令 8 完成时，该位翻转为高电平。(RO)

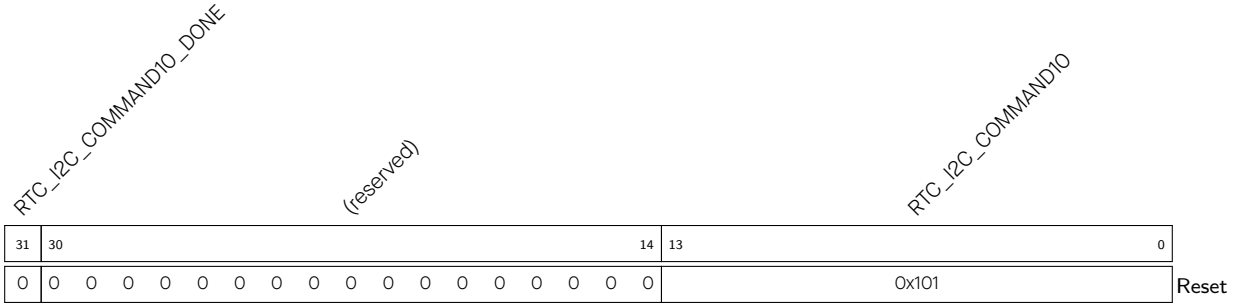
Register 2.37. RTC_I2C_CMD9_REG (0x005C)



RTC_I2C_COMMAND9 命令 9，详细信息可参考 I2C 控制器章节中的 I2C_COMD9_REG 寄存器。(R/W)

RTC_I2C_COMMAND9_DONE 命令 9 完成时，该位翻转为高电平。(RO)

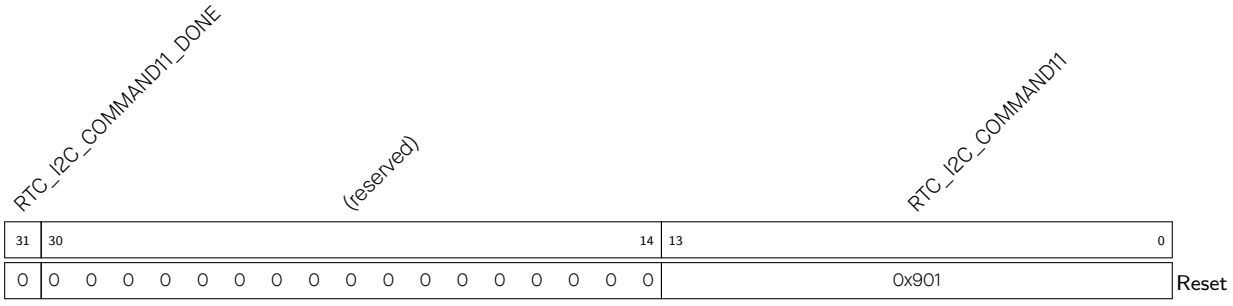
Register 2.38. RTC_I2C_CMD10_REG (0x0060)



RTC_I2C_COMMAND10 命令 10，详细信息可参考 I2C 控制器章节中的 I2C_COMD10_REG 寄存器。(R/W)

RTC_I2C_COMMAND10_DONE 命令 10 完成时，该位翻转为高电平。(RO)

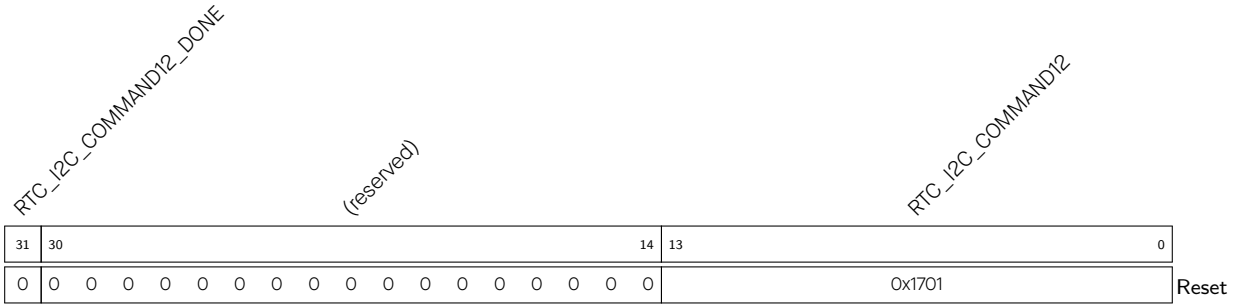
Register 2.39. RTC_I2C_CMD11_REG (0x0064)



RTC_I2C_COMMAND11 命令 11，详细信息可参考 I2C 控制器章节中的 I2C_COMD11_REG 寄存器。(R/W)

RTC_I2C_COMMAND11_DONE 命令 11 完成时，该位翻转为高电平。(RO)

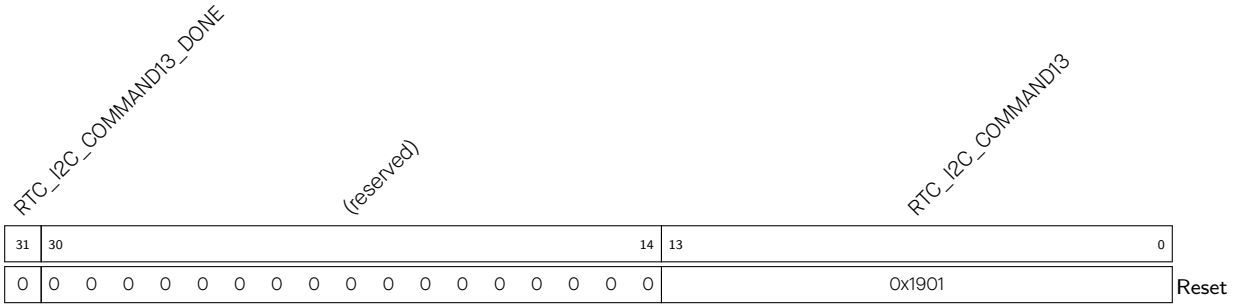
Register 2.40. RTC_I2C_CMD12_REG (0x0068)



RTC_I2C_COMMAND12 命令 12，详细信息可参考 I2C 控制器章节中的 I2C_COMD12_REG 寄存器。(R/W)

RTC_I2C_COMMAND12_DONE 命令 12 完成时，该位翻转为高电平。(RO)

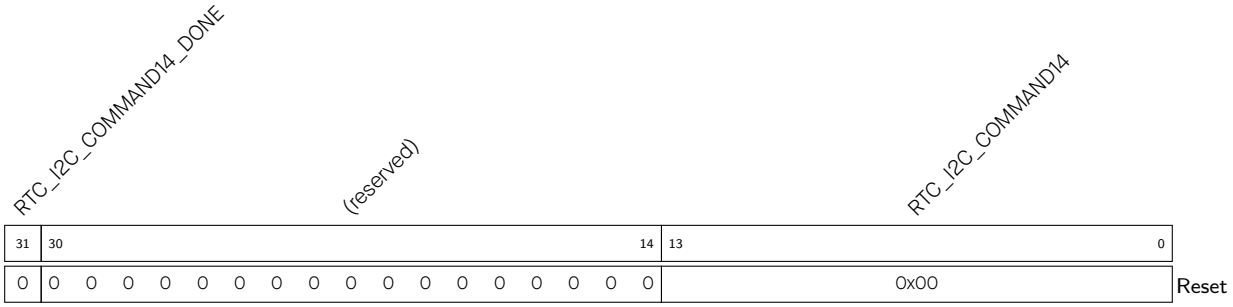
Register 2.41. RTC_I2C_CMD13_REG (0x006C)



RTC_I2C_COMMAND13 命令 13，详细信息可参考 I2C 控制器章节中的 I2C_COMD13_REG 寄存器。(R/W)

RTC_I2C_COMMAND13_DONE 命令 13 完成时，该位翻转为高电平。(RO)

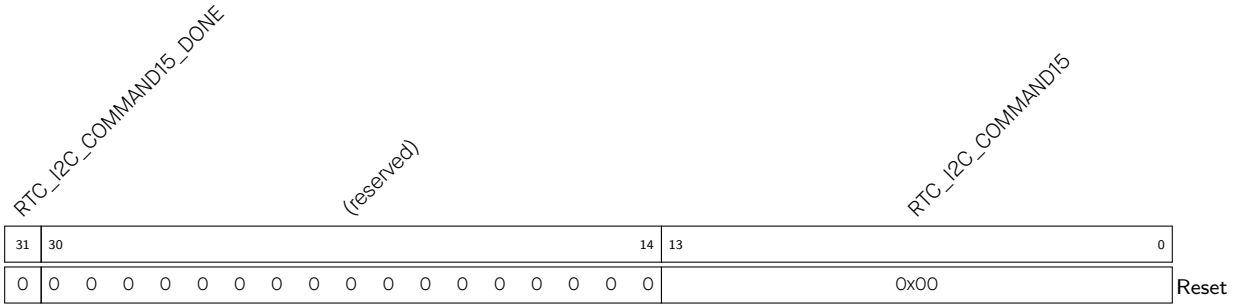
Register 2.42. RTC_I2C_CMD14_REG (0x0070)



RTC_I2C_COMMAND14 命令 14，详细信息可参考 I2C 控制器章节中的 I2C_COMD14_REG 寄存器。(R/W)

RTC_I2C_COMMAND14_DONE 命令 14 完成时，该位翻转为高电平。(RO)

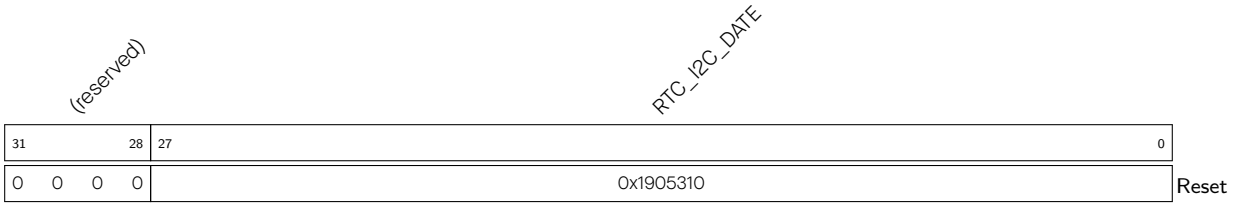
Register 2.43. RTC_I2C_CMD15_REG (0x0074)



RTC_I2C_COMMAND15 命令 15，详细信息可参考 I2C 控制器章节中的 I2C_COMD15_REG 寄存器。(R/W)

RTC_I2C_COMMAND15_DONE 命令 15 完成时，该位翻转为高电平。(RO)

Register 2.44. RTC_I2C_DATE_REG (0x00FC)



RTC_I2C_DATE 版本控制寄存器。(R/W)

第 3 章

通用 DMA 控制器 (GDMA)

3.1 概述

通用直接存储访问 (General Direct Memory Access, GDMA) 用于在外设与存储器之间以及存储器与存储器之间提供高速数据传输。软件可以在无需任何 CPU 操作的情况下通过 GDMA 快速搬移数据，从而降低了 CPU 的工作负载，提高了效率。

ESP32-S3 GDMA 共有 10 个独立的通道，其中包括 5 个接收通道和 5 个发送通道。这 10 个通道被支持 GDMA 功能的外设所共享，也就是说用户可以将通道分配给任何支持 GDMA 功能的外设。这些外设包括：SPI2、SPI3、UHCI0、I2S0、I2S1、LCD/CAM、AES、SHA、ADC 和 RMT。并且，每一个通道均支持访问内部 RAM 或者外部 RAM。

GDMA 支持通道间固定优先级及轮询仲裁以管理外设不同的带宽需求。

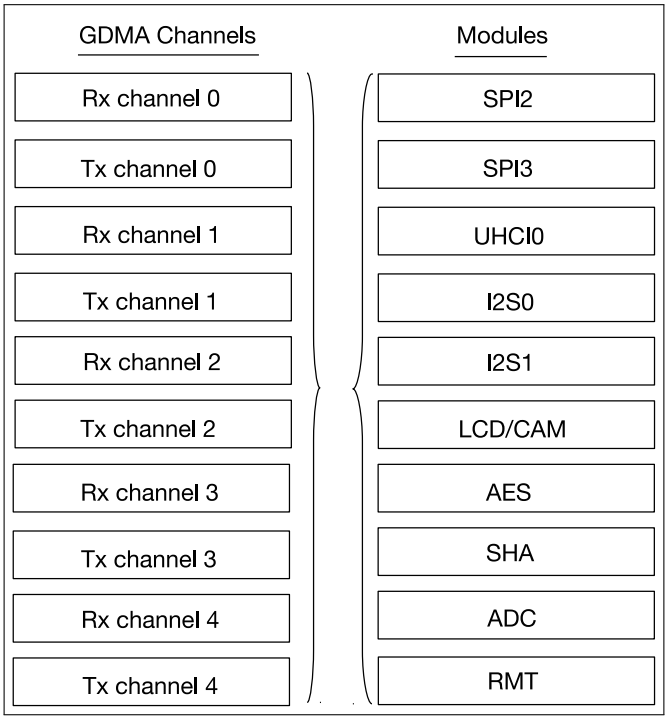


图 3.1-1. 具有 GDMA 功能的模块和 GDMA 通道

3.2 特性

GDMA 控制器具有以下几个特点：

- AHB 总线架构

- 数据传输以字节为单位，传输数据量可软件编程
- 支持链表
- 访问内部 RAM 时，支持 INCR burst 传输
- GDMA 能够访问的内部 RAM 最大地址空间为 480 KB
- GDMA 能够访问的最大外部 RAM 地址空间为 32 MB
- 包含 5 个接收、5 个发送通道
- 任一通道可访问内部及外部 RAM
- 任一通道支持可配置的外设选择
- 通道间固定优先级及轮询仲裁

3.3 架构

ESP32-S3 中所有需要进行高速数据传输的模块都具有 GDMA 功能。GDMA 控制器与 CPU 的数据总线使用相同的地址空间访问内部和外部 RAM。图 3.3-1 为 GDMA 引擎基本架构图。

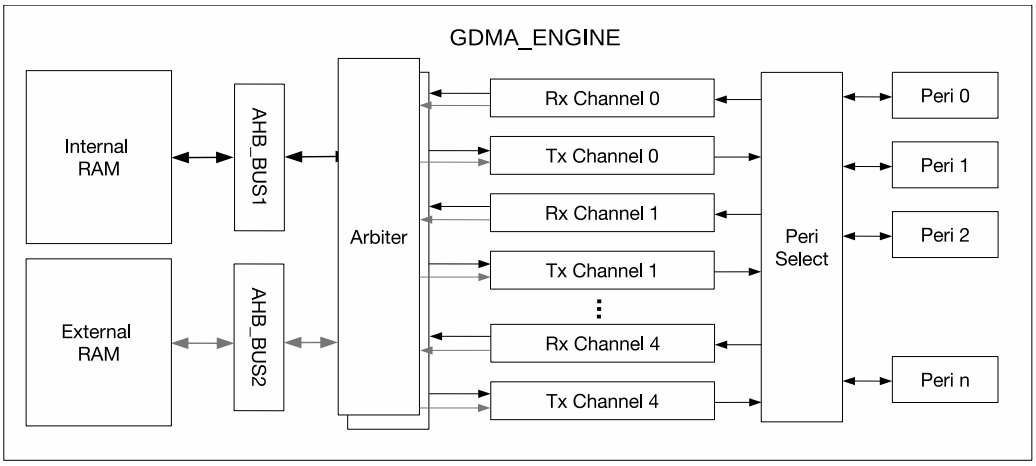


图 3.3-1. GDMA 引擎的架构

GDMA 引擎共有 10 个独立的通道，其中包括 5 个接收通道和 5 个发送通道。每个通道通道可选择与不同的外设相连，从而实现通道资源被外设共享。

GDMA 引擎包括两条独立的 AHB_BUS，记为 AHB_BUS1 和 AHB_BUS2。GDMA 引擎通过 AHB_BUS1 将数据存入内部 RAM 或者将数据从内部 RAM 取出，通过 AHB_BUS2 将数据存入外部 RAM 或者将数据从外部 RAM 取出。在通过 AHB_BUS 传输数据之前，GDMA 采用固定优先级的仲裁机制对每个通道的读写请求进行仲裁。RAM 的具体使用范围详见章节 *4 系统和存储器*。

软件可以通过挂载链表的方式来使用 GDMA 引擎。链表本身须存储在内部 RAM 中，包括 outlink*n* 与 inlink*n*，本文以 *n* 来表示通道号，*n* 为 0 ~ 4。GDMA 从内部 RAM 中取得链表，然后根据 outlink*n* 中的内容将相应 RAM 中的数据发送出去，也可根据 inlink*n* 中的内容将接收的数据存入指定 RAM 地址空间。

3.4 功能描述

3.4.1 链表

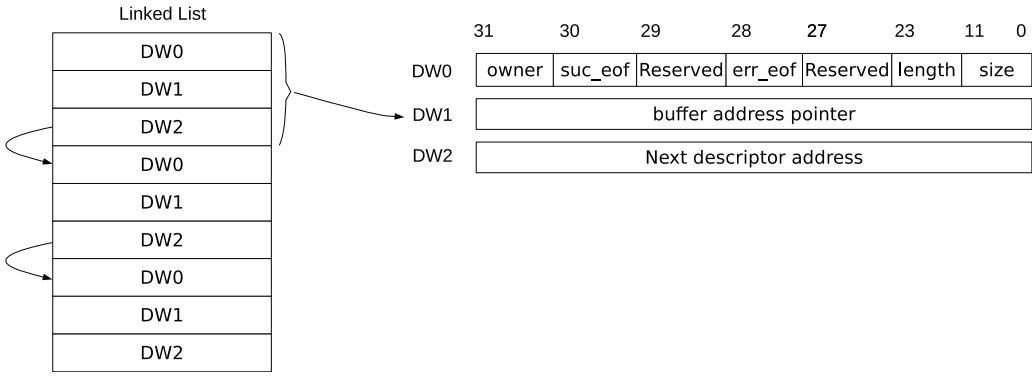


图 3.4-1. 链表结构图

图 3.4-1 所示为链表的结构图。发送链表与接收链表结构相同。每个链表由一个或者若干个描述符构成，一个描述符由三个字组成。链表应存放在内部 RAM 中，供 GDMA 引擎使用。描述符每一字段的意义如下：

- owner (DW0) [31]：表示当前描述符对应的 buffer 允许的操作者。
  1'b0：允许的操作者为 CPU；
  1'b1：允许的操作者为 GDMA 控制器。
  在 GDMA 使用完该描述符对应的 buffer 后，对于接收描述符，硬件默认会自动将该 bit 清零；对于发送描述符，需要将 GDMA_OUT_AUTO_WRBACK_CH*n* 置 1，硬件才会自动将该 bit 清零。软件在挂载链表时需要将该 bit 置 1。

  注意：本文以 GDMA_OUT 开头的寄存器对应 TX 通道寄存器，以 GDMA_IN 开头的寄存器对应 RX 通道寄存器。

- suc_eof (DW0) [30]：表示一个描述符对应的数据成功传输后是否触发 GDMA_IN_SUC_EOF_CH*n*_INT 或 GDMA_OUT_EOF_CH*n*_INT 中断。
  1'b0：当前描述符成功传输后不会触发中断；
  1'b1：当前描述符成功传输后触发中断。
  对于接收描述符，需要软件将该位写 0，硬件会在接收到包含 EOF 标志的数据后将该位置 1。
  对于发送描述符，需要软件按需要将描述符中的该位置 1。如果软件将在某个描述符中将该位配置为 1，则 GDMA 在处理完该描述符时，会在发送给外设的数据中加入 EOF 标志，告知外设该段数据是一个阶段性结束。

- Reserved (DW0) [29]：保留。此位为无关项。

- err_eof (DW0) [28]：表示接收结束错误标志。
  该位只用于 UHCI0 利用 GDMA 接收数据。对于接收描述符，硬件在收完描述符对应的数据段并检测到接收数据错误会将该位置 1。

- Reserved (DW0) [27:24]：保留。

- length (DW0) [23:12]：表示当前描述符对应的 buffer 中的有效字节数。对于发送描述符，该段由软件填写，表示从 buffer 中读取数据时需要读取的字节数；对于接收描述符，该段由硬件使用完该 buffer 后或者接收到最后一个数据时自动填写，表示 buffer 中存储的有效字节数。

- size (DW0) [11:0]：表示当前描述符对应的 buffer 容量的字节数。
- buffer address pointer (DW1)：buffer 的地址。
- next descriptor address (DW2)：下一个描述符的地址。如果当前描述符为链表中最后一个描述符时，该值可以为 0。该地址必须指向内部 RAM 的地址空间。

用 GDMA 接收数据时，如果接收数据的长度小于当前描述符指定的 buffer 长度，那么下一个描述符对应的接收数据不会占用该 buffer 的剩余空间。

3.4.2 外设到存储及存储到外设的数据传输

GDMA 支持存储到外设及外设到存储的数据传输，分别对应 TX 及 RX 功能。TX 通道通过 outlink*n* 实现将指定存储区域中的数据搬运到外设的发送端；RX 通道通过 inlink*n* 实现将外设接收到的数据搬运到指定的存储区域。

每个 RX/TX 通道均可以被配置连接到任意一个支持 GDMA 功能的外设，表 3.4-1 所示为配置寄存器与其对应外设的关系。当其中一个通道已经与某一个外设连接时，其他通道将不能配置为与该外设连接。同时，任一 RX/TX 通道均支持对内部及外部 RAM 的访问，请参见章节 3.4.8 及章节 3.4.9。

表 3.4-1. 配置寄存器与外设选择关系表

GDMA_PERI_IN_SEL_CH*n* GDMA_PERI_OUT_SEL_CH*n*	外设
0	SPI2
1	SPI3
2	UHCI0
3	I2S0
4	I2S1
5	LCD/CAM
6	AES
7	SHA
8	ADC
9	RMT
10 ~ 63	无效值

3.4.3 存储到存储数据传输

GDMA 支持存储到存储的数据传输。置位 GDMA_MEM_TRANS_EN_CH*n*，TX 通道 *n* 的输出将与 RX 通道 *n* 的输入相连，从而使能存储到存储的数据传输功能。需要注意的是，一个 TX 通道只与其编号对应的 RX 通道相连而实现存储到存储的数据传输。同样的，每一个 RX/TX 通道均可访问内部及外部 RAM，由此可产生 4 种数据传输组合模式：

- 内部 RAM 到内部 RAM
- 内部 RAM 到外部 RAM
- 外部 RAM 到内部 RAM
- 外部 RAM 到外部 RAM

3.4.4 通道 Buffer

GDMA 的每一个 Rx/Tx 通道均包括 3 级 FIFO，即 L1FIFO、L2FIFO、L3FIFO。如图 3.4-2，靠近存储器的为 L1FIFO，靠近外设的为 L3FIFO，中间一级为 L2FIFO。通道的 L1FIFO、L2FIFO 和 L3FIFO 具有固定深度，分别为 24、128 和 16 字节。

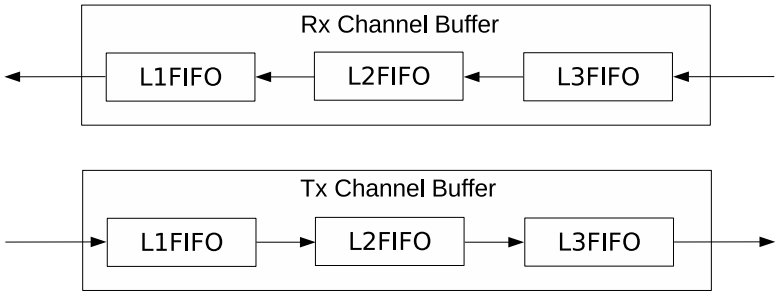


图 3.4-2. 通道 Buffer 示意图

3.4.5 启动 GDMA

软件通过挂载链表的方式来使用 GDMA。对于接收数据，软件挂载好接收链表并准备好接收数据，配置 GDMA_INLINK_ADDR_CH*n* 字段指向第一个接收链表描述符。置位 GDMA_INLINK_START_CH*n* 位启动 GDMA。对于发送数据，软件挂载好发送链表并准备好发送数据，配置 GDMA_OUTLINK_ADDR_CH*n* 字段指向第一个发送链表描述符。置位 GDMA_OUTLINK_START_CH*n* 位启动 GDMA。GDMA_INLINK_START_CH*n* 与 GDMA_OUTLINK_START_CH*n* 位由硬件自动清零。

有时您可能想要在 DMA 数据传输已经开始后追加更多描述符。要挂载更多描述符，原本看似只需清空已挂载链表最后一个描述符的 EOF 位，并将该描述符的 next descriptor address (DW2) 字段配置为新链表第一个描述符。但如果 DMA 数据传输已经或马上就要结束，这个方法便行不通了。GDMA 引擎有专门的逻辑来确保数据传输继续或重启：如果数据传输仍在进行，GDMA 引擎会确保顾及到新追加的描述符；如果数据传输已经结束，GDMA 引擎会重启数据传输，传输新追加的描述符。这个逻辑由 Restart 功能实现。

软件使用 Restart 功能时，需要重写已挂载链表的最后一个描述符，使其第三个字中的内容（即 DW2）指向新链表的首地址；然后置位 GDMA_INLINK_RESTART_CH*n* 或者 GDMA_OUTLINK_RESTART_CH*n*（这两个位由硬件自动清零），如图 3.4-3 所示，硬件会在读取已挂载链表的最后一个描述符时，获取新挂载链表的地址，从而继续处理新挂载的链表。

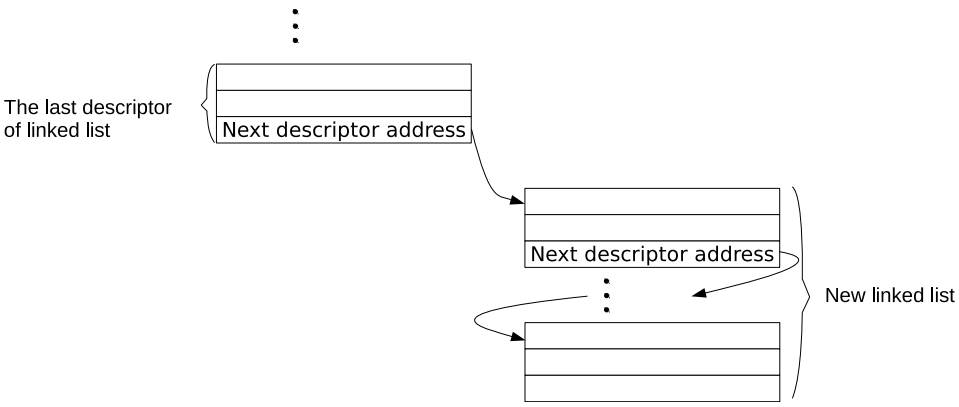


图 3.4-3. 链表关系图

3.4.6 读链表

软件在配置并启动 GDMA 后，GDMA 会从内部 RAM 读取链表。GDMA 会检查读入的链表描述符是否正确。只有当链表描述符通过检查时，GDMA 对应的通道才会开始搬运数据。当链表描述符没有通过检查，硬件将触发描述符错误中断（GDMA_IN_DSCR_ERR_CH*n*_INT 或者 GDMA_OUT_DSCR_ERR_CH*n*_INT），同时该通道将会处于阻塞状态，停止工作。

描述符检查项包括：

- GDMA_IN_CHECK_OWNER_CH*n* 或者 GDMA_OUT_CHECK_OWNER_CH*n* 置 1 时，检查描述符的 owner 位。如果该位为 0，表示当前操作者为 CPU，则不通过检查。GDMA_IN_CHECK_OWNER_CH*n* 或者 GDMA_OUT_CHECK_OWNER_CH*n* 置 0 时，不检查描述符的 owner 位；
- 检查描述符中第二个字指示的地址是否在 0x3FC88000 ~ 0x3FCFFFFF 或者 0x3C000000 ~ 0x3DFFFFFF（请参见本节 3.4.8）。如果在该范围内，则通过检查。否则，不通过检查。

软件在检查到通道描述符错误中断后，需要复位对应的通道，并置位 GDMA_OUTLINK_START_CH*n* 或者 GDMA_INLINK_START_CH*n* 位启动 GDMA。

注意：描述符的第三个字指示的地址只能在内部 RAM，指向下一个可用描述符；所有描述符都需存在内存中。

3.4.7 数据传输结束标志

GDMA 通过 EOF 来指示对应描述符所需传输的数据段传输结束。

发送数据时，置位 GDMA_OUT_TOTAL_EOF_CH*n*_INT_ENA 位使能 GDMA_OUT_TOTAL_EOF_CH*n*_INT 中断，当带有 EOF 标志的描述符对应 buffer 的数据传输完成后，GDMA 会产生该中断。

接收数据时，置位 GDMA_IN_SUC_EOF_CH*n*_INT_ENA 位使能 GDMA_IN_SUC_EOF_CH*n*_INT 中断，表示带有 EOF 标志的数据段接收完成。GDMA 还支持中断 GDMA_IN_ERR_CH*n*_EOF_INT，置位 GDMA_IN_ERR_EOF_CH*n*_INT_ENA 使能该中断，表示对应描述符所需传输的数据段数据接收完成但接收数据有错误。需要注意的是，只有当通道连接的外设为 UHCI0 时，才支持该中断。

软件在检测到 GDMA_OUT_TOTAL_EOF_CH*n*_INT 或 GDMA_IN_SUC_EOF_CH*n*_INT 中断时，可以记录 GDMA_OUT_EOF_DES_ADDR_CH*n* 或 GDMA_IN_SUC_EOF_DES_ADDR_CH*n* 字段的值，即最后一个描述符的地址。这样，软件可以知道哪些描述符已经被使用并根据需要回收描述符。

注意：本章中提到发送链表描述符的 EOF 为 suc_eof，接收链表描述符的 EOF 可以为 suc_eof 和 err_eof。

3.4.8 访问内部 RAM

GDMA 任意 RX/TX 通道均可以访问内部 RAM，其可访问的内部 RAM 地址空间为 0x3FC88000 ~ 0x3FCFFFFF。为加速数据传输速率，支持突发传输模式。置位 GDMA_IN_DATA_BURST_EN_CH*n* 使能 RX 通道突发传输模式；置位 GDMA_OUT_DATA_BURST_EN_CH*n* 使能 TX 通道突发传输模式。默认情况下，突发传输没有使能。

表 3.4-2. 访问内部 RAM 的链表描述符参数对齐要求

inlink/outlink	burst mode	size	length	buffer address pointer
inlink	0	无对齐要求	无对齐要求	无对齐要求
	1	4 字节对齐	无对齐要求	4 字节对齐

outlink	0	无对齐要求	无对齐要求	无对齐要求
	1	无对齐要求	无对齐要求	无对齐要求

如表 3.4-2 所示为访问内部 RAM 时，链表描述符参数配置对齐要求。

当突发模式没有被使能时，无论是发送链表描述符还是接收链表描述符，其参数 size，length 及 buffer address pointer 均没有字对齐的要求。也就是说，对于一个描述符，在可访问的内部 RAM 地址空间，GDMA 可以从任意起始地址，读出配置长度的数据，长度取值范围为 1 ~ 4095；或者，将接收到的数据长度（1 ~ 4095）写入任意起始地址开始的连续地址。

当突发模式使能时，对于发送链表描述符，参数 size，length 及 buffer address pointer 均没有字对齐的要求。而对于接收链表描述符，除了参数 length，参数 size 和 buffer address pointer 均需要保持字对齐。

3.4.9 访问外部 RAM

GDMA 任一 RX/TX 通道均可以访问外部 RAM，并且其数据传输只能基于突发模式，其可访问的外部地址空间为：0x3C000000 ~ 0x3DFFFFFF。本文中定义，一次突发传输的数据量为块大小 (Block Size)。GDMA 支持的块大小为 16/32/64 字节。GDMA_IN_EXT_MEM_BK_SIZE_CH*n* 和 GDMA_OUT_EXT_MEM_BK_SIZE_CH*n* 分别用来配置 Rx/Tx 通道的块大小。

表 3.4-3. 访问外部 RAM 的链表描述符参数对齐要求

inlink/outlink	size	length	buffer address pointer
inlink	块大小对齐	无对齐要求	块大小对齐
outlink	无对齐要求	无对齐要求	无对齐要求

如表 3.4-3 所示为访问外部 RAM 时，链表描述符参数配置对齐要求。对于发送链表描述符，参数 size，length 及 buffer address pointer 均没有对齐的要求。而对于接收链表描述符，除了参数 length，参数 size 和 buffer address pointer 均需要保持块大小对齐。表 3.4-4 显示了 GDMA_OUT_EXT_MEM_BK_SIZE_CH*n* 或 GDMA_OUT_EXT_MEM_BK_SIZE_CH*n* 与对齐方式的关系。

表 3.4-4. 配置寄存器、块大小和对齐方式的关系表

GDMA_IN_EXT_MEM_BK_SIZE_CH*n* 或 GDMA_OUT_EXT_MEM_BK_SIZE_CH*n*	块大小	对齐方式
0	16 字节	16 字节对齐
1	32 字节	32 字节对齐
2	64 字节	64 字节对齐

注意：对于接收链表描述符，当接收的数据长度不是块大小对齐，由于 GDMA 此时的数据传输基于突发传输，GDMA 会在最后不满块大小的有效数据之后补若干个 0 使 GDMA 可以发起突发传输。用户可以通过读回写的接收链表描述符中的参数 length 来得到真实接收的数据长度。

3.4.10 访问外部 RAM 的权限管理

在 ESP32-S3 中，GDMA 包含一个针对外部 RAM 访问的权限管理模块。如图 3.4-4 所示，权限管理模块通过三条可配置的分割线，即分割线 0，1，2，将 32 MB 的外部 RAM 空间分割为四块区域。

- 区域 0：0x3C000000 ~ 分割线 0 地址（包括 0x3C000000，但不包括分割线 0 地址）
- 区域 1：分割线 0 地址 ~ 分割线 1 地址（包括分割线 0 地址，但不包括分割线 1 地址）
- 区域 2：分割线 1 地址 ~ 分割线 2 地址（包括分割线 1 地址，但不包括分割线 2 地址）
- 区域 3：分割线 2 地址 ~ 0x3DFFFFFF（包括分割线 2 地址）

分割线 0，1，2 分别由寄存器 PMS_EDMA_BOUNDARY_0、PMS_EDMA_BOUNDARY_1、PMS_EDMA_BOUNDARY_2 配置，这些寄存器位于权限管理模块中，请参考章节 15 权限控制 *(PMS)*。分割线以 4 KB 为单位进行地址划分。例如，配置 PMS_EDMA_BOUNDARY_0 为 0x80，则分割线 0 对应的地址计算方式为 0x3C000000 + 0x80 * 4 KB = 3c080000，其中 0x3C000000 为 GDMA 可访问的外部 RAM 的起始地址。

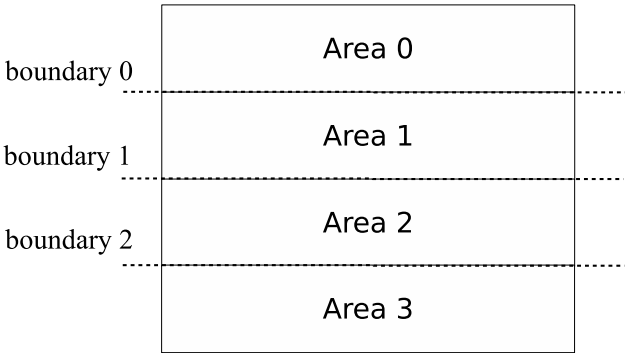


图 3.4-4. 外部 RAM 的权限区域划分

对于区域 0 和区域 3，支持 GDMA 功能的外设均没有访问权限。而对于区域 1 和区域 2，用户可以独立管理支持 GDMA 功能的外设对这两块区域的访问权限。在权限管理模块中，包含外设（SPI2，SPI3，UHCI0，I2S0，I2S1，LCD/CAM，AES，SHA，ADC 和 RMT）访问区域 1 和区域 2 的权限管理寄存器。例如，对于外设 SPI2，PMS_EDMA_PMS_SPI2_ATTR1 用于配置 SPI2 对区域 1 的读写权限。其中该寄存器的 bit 0 写 1 用于使能读权限，bit 1 写 1 用于使能写权限。同样，PMS_EDMA_PMS_SPI2_ATTR2 用于配置 SPI2 对区域 2 的读写权限。

当外设通过 GDMA 访问非法的地址空间时，会触发 GDMA_ETXMEN_REJECT_INT 中断。用户可通过 GDMA_ETXMEM_REJECT_ADDR、GDMA_ETXMEN_REJECT_PERI_NUM、GDMA_ETXMEN_REJECT_CHANNEL_NUM、GDMA_ETXMEM_REJECT_ATTR 查看记录下的非法访问对应的地址，外设，通道号及读写属性。

3.4.11 内部及外部 RAM 数据无缝访问

在一些应用场景中，希望一帧或一个包中的数据，某些数据通过访问内部 RAM 实现，某些数据通过访问外部 RAM 实现。为保证数据处理的实时性，GDMA 支持在由多个描述符构成的链表中，某些描述符控制访问内部 RAM 中的数据，而某些描述符控制访问外部 RAM 中的数据，从而实现对内部及外部 RAM 数据访问的无缝切换。

3.4.12 仲裁

为了确保及时响应高速低延迟的外设请求，比如 SPI，LCD/CAM 等，GDMA 在通道仲裁机制中引入固定优先级，即每个通道的优先级可配置。GDMA 支持 10（0 ~ 9）个等级的优先级。其数值越大，对应的优先级越高，请求响应越及时。当若干个通道配置为相同的优先级时，这几个通道间对请求的响应将采用轮询仲裁机制。

需要注意的是，所有外设总的吞吐率之和不能超过 GDMA 能支持的最大有效带宽，从而保证低优先级的外设请求也能获得响应。

3.5 GDMA 中断

- GDMA_OUT_TOTAL_EOF_CH*n*_INT：对于发送通道*n*，当一个链表（可包含多个链表描述符）对应的所有数据都已发送完成时触发此中断。
- GDMA_IN_DSCR_EMPTY_CH*n*_INT：对于接收通道*n*，当接收链表描述符指向的 buffer 大小小于待接收数据长度时触发此中断。
- GDMA_OUT_DSCR_ERR_CH*n*_INT：对于发送通道*n*，当发送链表描述符里有错误时触发此中断。
- GDMA_IN_DSCR_ERR_CH*n*_INT：对于接收通道*n*，当接收链表描述符里有错误时触发此中断。
- GDMA_OUT_EOF_CH*n*_INT：对于发送通道 *n*，当发送描述符的 EOF 位为 1，并且该描述符对应的数据发送完成时触发此中断。当 GDMA_OUT_EOF_MODE_CH*n* 为 0 时，该描述符对应的最后一个数据进入到 GDMA TX 通道时，该中断触发；当 GDMA_OUT_EOF_MODE_CH*n* 为 1 时，该描述符对应的最后一个数据从 GDMA TX 通道取出时，该中断触发。
- GDMA_OUT_DONE_CH*n*_INT：对于发送通道*n*，当一个发送链表描述符对应的数据发送完成时触发此中断。
- GDMA_IN_ERR_EOF_CH*n*_INT：对于接收通道 *n*，当接收的描述符对应数据段中有错误发生时触发此中断。（该中断只用于外设选择 UHCI0 (UART0/UART1) 时）。
- GDMA_IN_SUC_EOF_CH*n*_INT：对于接收通道*n*，当一个接收链表描述符对应的数据接收完成，并且描述符的 suc_eof 为 1 时触发此中断。
- GDMA_IN_DONE_CH*n*_INT：对于接收通道*n*，当一个接收链表描述符对应的数据接收完成时触发此中断。

3.6 编程流程

3.6.1 GDMA 时钟与复位配置流程

GDMA 的时钟与复位配置流程如下：

1. 置位系统寄存器 SYSTEM_DMA_CLK_EN，使能 GDMA 模块时钟；
2. 对系统寄存器 SYSTEM_DMA_RST 置 0，释放 GDMA 的复位信号。

3.6.2 GDMA TX 通道配置流程

利用 GDMA 发送数据时，GDMA TX 通道的软件配置流程如下：

1. 对寄存器 GDMA_OUT_RST_CH*n* 置 1 然后置 0，复位 GDMA TX 通道状态机和 FIFO 指针；
2. 挂载好发送链表，配置寄存器 GDMA_OUTLINK_ADDR_CH*n* 指向第一个发送链表描述符；
3. 配置 GDMA_PERI_OUT_SEL_CH*n* 为对应的外设号，见表3.4-1；
4. 置位 GDMA_OUTLINK_START_CH*n* 启动 GDMA TX 通道发送数据；

5. 配置对应的外设（SPI2、SPI3、UHCI0 (UART0/UART1/UART2)、I2S0、I2S1、AES、SHA、ADC），并启动该外设，具体配置请参考对应的外设章节；
6. 等待 GDMA_OUT_TOTAL_EOF_CH*n*_INT 中断，即数据传输完成。

3.6.3 GDMA RX 通道配置流程

利用 GDMA 接收数据时，GDMA RX 通道的软件配置流程如下：

1. 对寄存器 GDMA_IN_RST_CH*n* 置 1 然后置 0，复位 GDMA RX 通道状态机和 FIFO 指针；
2. 挂载好接收链表，配置寄存器 GDMA_INLINK_ADDR_CH*n* 指向第一个接收链表描述符；
3. 配置 GDMA_PERI_IN_SEL_CH*n* 为对应的外设号，见表3.4-1；
4. 置位 GDMA_INLINK_START_CH*n* 启动 GDMA RX 通道发送数据；
5. 配置对应的外设（SPI2、SPI3、UHCI0 (UART0/UART1/UART2)、I2S0、I2S1、AES、SHA、ADC），并启动该外设，具体配置请参考对应的外设章节；

3.6.4 GDMA 存储器到存储器配置流程

利用 GDMA 从存储到存储搬运数据时配置流程如下：

1. 对寄存器 GDMA_OUT_RST_CH*n* 置 1 然后置 0，复位 GDMA TX 通道状态机和 FIFO 指针；
2. 对寄存器 GDMA_IN_RST_CH*n* 置 1 然后置 0，复位 GDMA RX 通道状态机和 FIFO 指针；
3. 挂载好发送链表，配置寄存器 GDMA_OUTLINK_ADDR_CH*n* 指向第一个发送链表描述符；
4. 挂载好接收链表，配置寄存器 GDMA_INLINK_ADDR_CH*n* 指向第一个接收链表描述符；
5. 置位 GDMA_MEM_TRANS_EN_CH*n* 使能 memory-to-memory 传输功能；
6. 置位 GDMA_OUTLINK_START_CH*n* 启动 GDMA TX 通道发送数据；
7. 置位 GDMA_INLINK_START_CH*n* 启动 GDMA RX 通道发送数据；
8. 如果某个发送链表描述符的 suc_eof 位配置为 1，则该描述符对应的数据段传输结束后会触发 GDMA_IN_SUC_EOF_CH*n*_INT 中断。

3.7 寄存器列表

本小节的所有地址均为相对于 GDMA 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
配置寄存器			
GDMA_IN_CONF0_CH0_REG	接收通道 0 的配置寄存器 0	0x0000	R/W
GDMA_IN_CONF1_CH0_REG	接收通道 0 的配置寄存器 1	0x0004	R/W
GDMA_IN_POP_CH0_REG	接送通道 0 的数据弹出控制寄存器	0x001C	varies
GDMA_IN_LINK_CH0_REG	接送通道 0 的链表配置和控制寄存器	0x0020	varies
GDMA_OUT_CONF0_CH0_REG	发送通道 0 的配置寄存器 0	0x0060	R/W
GDMA_OUT_CONF1_CH0_REG	发送通道 0 的配置寄存器 1	0x0064	R/W
GDMA_OUT_PUSH_CH0_REG	接收通道 0 的数据推送控制寄存器	0x007C	varies
GDMA_OUT_LINK_CH0_REG	发送通道 0 的链表配置和控制寄存器	0x0080	varies
GDMA_IN_CONF0_CH1_REG	接收通道 1 的配置寄存器 0	0x00C0	R/W
GDMA_IN_CONF1_CH1_REG	接收通道 1 的配置寄存器 1	0x00C4	R/W
GDMA_IN_POP_CH1_REG	接送通道 1 的数据弹出控制寄存器	0x00DC	varies
GDMA_IN_LINK_CH1_REG	接送通道 1 的链表配置和控制寄存器	0x00E0	varies
GDMA_OUT_CONF0_CH1_REG	发送通道 1 的配置寄存器 0	0x0120	R/W
GDMA_OUT_CONF1_CH1_REG	发送通道 1 的配置寄存器 1	0x0124	R/W
GDMA_OUT_PUSH_CH1_REG	接收通道 1 的数据推送控制寄存器	0x013C	varies
GDMA_OUT_LINK_CH1_REG	发送通道 1 的链表配置和控制寄存器	0x0140	varies
GDMA_IN_CONF0_CH2_REG	接收通道 2 的配置寄存器 0	0x0180	R/W
GDMA_IN_CONF1_CH2_REG	接收通道 2 的配置寄存器 1	0x0184	R/W
GDMA_IN_POP_CH2_REG	接送通道 2 的数据弹出控制寄存器	0x019C	varies
GDMA_IN_LINK_CH2_REG	接送通道 2 的链表配置和控制寄存器	0x01A0	varies
GDMA_OUT_CONF0_CH2_REG	发送通道 2 的配置寄存器 0	0x01E0	R/W
GDMA_OUT_CONF1_CH2_REG	发送通道 2 的配置寄存器 1	0x01E4	R/W
GDMA_OUT_PUSH_CH2_REG	接收通道 2 的数据推送控制寄存器	0x01FC	varies
GDMA_OUT_LINK_CH2_REG	发送通道 2 的链表配置和控制寄存器	0x0200	varies
GDMA_IN_CONF0_CH3_REG	接收通道 3 的配置寄存器 0	0x0240	R/W
GDMA_IN_CONF1_CH3_REG	接收通道 3 的配置寄存器 1	0x0244	R/W
GDMA_IN_POP_CH3_REG	接送通道 3 的数据弹出控制寄存器	0x025C	varies
GDMA_IN_LINK_CH3_REG	接送通道 3 的链表配置和控制寄存器	0x0260	varies
GDMA_OUT_CONF0_CH3_REG	发送通道 3 的配置寄存器 0	0x02A0	R/W
GDMA_OUT_CONF1_CH3_REG	发送通道 3 的配置寄存器 1	0x02A4	R/W
GDMA_OUT_PUSH_CH3_REG	接收通道 3 的数据推送控制寄存器	0x02BC	varies
GDMA_OUT_LINK_CH3_REG	发送通道 3 的链表配置和控制寄存器	0x02C0	varies
GDMA_IN_CONF0_CH4_REG	接收通道 4 的配置寄存器 0	0x0300	R/W
GDMA_IN_CONF1_CH4_REG	接收通道 4 的配置寄存器 1	0x0304	R/W
GDMA_IN_POP_CH4_REG	接送通道 4 的数据弹出控制寄存器	0x031C	varies
GDMA_IN_LINK_CH4_REG	接送通道 4 的链表配置和控制寄存器	0x0320	varies

名称	描述	地址	访问
GDMA_OUT_CONF0_CH4_REG	发送通道 4 的配置寄存器 0	0x0360	R/W
GDMA_OUT_CONF1_CH4_REG	发送通道 4 的配置寄存器 1	0x0364	R/W
GDMA_OUT_PUSH_CH4_REG	接收通道 4 的数据推送控制寄存器	0x037C	varies
GDMA_OUT_LINK_CH4_REG	发送通道 4 的链表配置和控制寄存器	0x0380	varies
GDMA_PD_CONF_REG	保留	0x03C4	R/W
GDMA_MISC_CONF_REG	杂项控制寄存器	0x03C8	R/W
中断寄存器			
GDMA_IN_INT_RAW_CH0_REG	接收通道 0 的原始中断状态	0x0008	R/WTC/SS
GDMA_IN_INT_ST_CH0_REG	接收通道 0 的屏蔽中断	0x000C	RO
GDMA_IN_INT_ENA_CH0_REG	接收通道 0 的中断使能位	0x0010	R/W
GDMA_IN_INT_CLR_CH0_REG	接收通道 0 的中断清除位	0x0014	WT
GDMA_OUT_INT_RAW_CH0_REG	发送通道 0 的原始中断状态	0x0068	R/WTC/SS
GDMA_OUT_INT_ST_CH0_REG	发送通道 0 的屏蔽中断	0x006C	RO
GDMA_OUT_INT_ENA_CH0_REG	发送通道 0 的中断使能位	0x0070	R/W
GDMA_OUT_INT_CLR_CH0_REG	发送通道 0 的中断清除位	0x0074	WT
GDMA_IN_INT_RAW_CH1_REG	接收通道 1 的原始中断状态	0x00C8	R/WTC/SS
GDMA_IN_INT_ST_CH1_REG	接收通道 1 的屏蔽中断	0x00CC	RO
GDMA_IN_INT_ENA_CH1_REG	接收通道 1 的中断使能位	0x00D0	R/W
GDMA_IN_INT_CLR_CH1_REG	接收通道 1 的中断清除位	0x00D4	WT
GDMA_OUT_INT_RAW_CH1_REG	发送通道 1 的原始中断状态	0x0128	R/WTC/SS
GDMA_OUT_INT_ST_CH1_REG	发送通道 1 的屏蔽中断	0x012C	RO
GDMA_OUT_INT_ENA_CH1_REG	发送通道 1 的中断使能位	0x0130	R/W
GDMA_OUT_INT_CLR_CH1_REG	发送通道 1 的中断清除位	0x0134	WT
GDMA_IN_INT_RAW_CH2_REG	接收通道 2 的原始中断状态	0x0188	R/WTC/SS
GDMA_IN_INT_ST_CH2_REG	接收通道 2 的屏蔽中断	0x018C	RO
GDMA_IN_INT_ENA_CH2_REG	接收通道 2 的中断使能位	0x0190	R/W
GDMA_IN_INT_CLR_CH2_REG	接收通道 2 的中断清除位	0x0194	WT
GDMA_OUT_INT_RAW_CH2_REG	发送通道 2 的原始中断状态	0x01E8	R/WTC/SS
GDMA_OUT_INT_ST_CH2_REG	发送通道 2 的屏蔽中断	0x01EC	RO
GDMA_OUT_INT_ENA_CH2_REG	发送通道 2 的中断使能位	0x01F0	R/W
GDMA_OUT_INT_CLR_CH2_REG	发送通道 2 的中断清除位	0x01F4	WT
GDMA_IN_INT_RAW_CH3_REG	接收通道 3 的原始中断状态	0x0248	R/WTC/SS
GDMA_IN_INT_ST_CH3_REG	接收通道 3 的屏蔽中断	0x024C	RO
GDMA_IN_INT_ENA_CH3_REG	接收通道 3 的中断使能位	0x0250	R/W
GDMA_IN_INT_CLR_CH3_REG	接收通道 3 的中断清除位	0x0254	WT

名称	描述	地址	访问
GDMA_OUT_INT_RAW_CH3_REG	发送通道 3 的原始中断状态	0x02A8	R/WTC/SS
GDMA_OUT_INT_ST_CH3_REG	发送通道 3 的屏蔽中断	0x02AC	RO
GDMA_OUT_INT_ENA_CH3_REG	发送通道 3 的中断使能位	0x02B0	R/W
GDMA_OUT_INT_CLR_CH3_REG	发送通道 3 的中断清除位	0x02B4	WT
GDMA_IN_INT_RAW_CH4_REG	接收通道 4 的原始中断状态	0x0308	R/WTC/SS
GDMA_IN_INT_ST_CH4_REG	接收通道 4 的屏蔽中断	0x030C	RO
GDMA_IN_INT_ENA_CH4_REG	接收通道 4 的中断使能位	0x0310	R/W
GDMA_IN_INT_CLR_CH4_REG	接收通道 4 的中断清除位	0x0314	WT
GDMA_OUT_INT_RAW_CH4_REG	发送通道 4 的原始中断状态	0x0368	R/WTC/SS
GDMA_OUT_INT_ST_CH4_REG	发送通道 4 的屏蔽中断	0x036C	RO
GDMA_OUT_INT_ENA_CH4_REG	发送通道 4 的中断使能位	0x0370	R/W
GDMA_OUT_INT_CLR_CH4_REG	发送通道 4 的中断清除位	0x0374	WT
GDMA_EXTMEM_REJECT_INT_RAW_REG	外部 RAM 权限的原始中断状态	0x03FC	R/WTC/SS
GDMA_EXTMEM_REJECT_INT_ST_REG	外部 RAM 权限的屏蔽中断状态	0x0400	RO
GDMA_EXTMEM_REJECT_INT_ENA_REG	外部 RAM 权限的中断使能位	0x0404	R/W
GDMA_EXTMEM_REJECT_INT_CLR_REG	外部 RAM 权限的中断清除位	0x0408	WT
状态寄存器			
GDMA_INFIFO_STATUS_CH0_REG	接收通道 0 的 RX FIFO 状态 0	0x0018	RO
GDMA_IN_STATE_CH0_REG	接收通道 0 的接收状态	0x0024	RO
GDMA_IN_SUC_EOF_DES_ADDR_CH0_REG	接收通道 0 传输完成时的接收链表描述符地址	0x0028	RO
GDMA_IN_ERR_EOF_DES_ADDR_CH0_REG	接收通道 0 发生错误时的接收链表描述符地址	0x002C	RO
GDMA_IN_DSCR_CH0_REG	接收通道 0 已预读取的接收链表描述符指向的下一个接收链表描述符地址	0x0030	RO
GDMA_IN_DSCR_BF0_CH0_REG	接收通道 0 当前已预读取的接收链表描述符所在地址	0x0034	RO
GDMA_IN_DSCR_BF1_CH0_REG	接收通道 0 前一个已预读取的接收链表描述符所在地址	0x0038	RO
GDMA_OUTFIFO_STATUS_CH0_REG	发送通道 0 的 TX FIFO 状态	0x0078	RO
GDMA_OUT_STATE_CH0_REG	发送通道 0 的发送状态	0x0084	RO
GDMA_OUT_EOF_DES_ADDR_CH0_REG	发送通道 0 传输完成时的发送链表描述符地址	0x0088	RO
GDMA_OUT_EOF_BFR_DES_ADDR_CH0_REG	发送通道 0 传输完成时的最后一个发送链表描述符地址	0x008C	RO
GDMA_OUT_DSCR_CH0_REG	发送通道 0 当前已预读取的发送链表描述符指向的下一个发送链表描述符地址	0x0090	RO
GDMA_OUT_DSCR_BF0_CH0_REG	发送通道 0 当前已预读取的发送链表描述符所在地址	0x0094	RO

名称	描述	地址	访问
GDMA_OUT_DSCR_BF1_CHO_REG	发送通道 0 前一个已预读取的发送链表描述符所在地址	0x0098	RO
GDMA_INFIFO_STATUS_CH1_REG	接收通道 1 的 RX FIFO 状态 0	0x00D8	RO
GDMA_IN_STATE_CH1_REG	接收通道 1 的接收状态	0x00E4	RO
GDMA_IN_SUC_EOF_DES_ADDR_CH1_REG	接收通道 1 传输完成时的接收链表描述符地址	0x00E8	RO
GDMA_IN_ERR_EOF_DES_ADDR_CH1_REG	接收通道 1 发生错误时的接收链表描述符地址	0x00EC	RO
GDMA_IN_DSCR_CH1_REG	接收通道 1 已预读取的接收链表描述符指向的下一个接收链表描述符地址	0x00F0	RO
GDMA_IN_DSCR_BFO_CH1_REG	接收通道 1 当前已预读取的接收链表描述符所在地址	0x00F4	RO
GDMA_IN_DSCR_BF1_CH1_REG	接收通道 1 前一个已预读取的接收链表描述符所在地址	0x00F8	RO
GDMA_OUTFIFO_STATUS_CH1_REG	发送通道 1 的 TX FIFO 状态	0x0138	RO
GDMA_OUT_STATE_CH1_REG	发送通道 1 的发送状态	0x0144	RO
GDMA_OUT_EOF_DES_ADDR_CH1_REG	发送通道 1 传输完成时的发送链表描述符地址	0x0148	RO
GDMA_OUT_EOF_BFR_DES_ADDR_CH1_REG	发送通道 0 传输完成时的最后一个发送链表描述符地址	0x014C	RO
GDMA_OUT_DSCR_CH1_REG	发送通道 1 当前已预读取的发送链表描述符指向的下一个发送链表描述符地址	0x0150	RO
GDMA_OUT_DSCR_BFO_CH1_REG	发送通道 1 当前已预读取的发送链表描述符所在地址	0x0154	RO
GDMA_OUT_DSCR_BF1_CH1_REG	发送通道 1 前一个已预读取的发送链表描述符所在地址	0x0158	RO
GDMA_INFIFO_STATUS_CH2_REG	接收通道 2 的 RX FIFO 状态 0	0x0198	RO
GDMA_IN_STATE_CH2_REG	接收通道 2 的接收状态	0x01A4	RO
GDMA_IN_SUC_EOF_DES_ADDR_CH2_REG	接收通道 2 传输完成时的接收链表描述符地址	0x01A8	RO
GDMA_IN_ERR_EOF_DES_ADDR_CH2_REG	接收通道 2 发生错误时的接收链表描述符地址	0x01AC	RO
GDMA_IN_DSCR_CH2_REG	接收通道 2 已预读取的接收链表描述符指向的下一个接收链表描述符地址	0x01B0	RO
GDMA_IN_DSCR_BFO_CH2_REG	接收通道 2 当前已预读取的接收链表描述符所在地址	0x01B4	RO
GDMA_IN_DSCR_BF1_CH2_REG	接收通道 2 前一个已预读取的接收链表描述符所在地址	0x01B8	RO
GDMA_OUTFIFO_STATUS_CH2_REG	发送通道 2 的 TX FIFO 状态	0x01F8	RO
GDMA_OUT_STATE_CH2_REG	发送通道 2 的发送状态	0x0204	RO
GDMA_OUT_EOF_DES_ADDR_CH2_REG	发送通道 2 传输完成时的发送链表描述符地址	0x0208	RO
GDMA_OUT_EOF_BFR_DES_ADDR_CH2_REG	发送通道 0 传输完成时的最后一个发送链表描述符地址	0x020C	RO

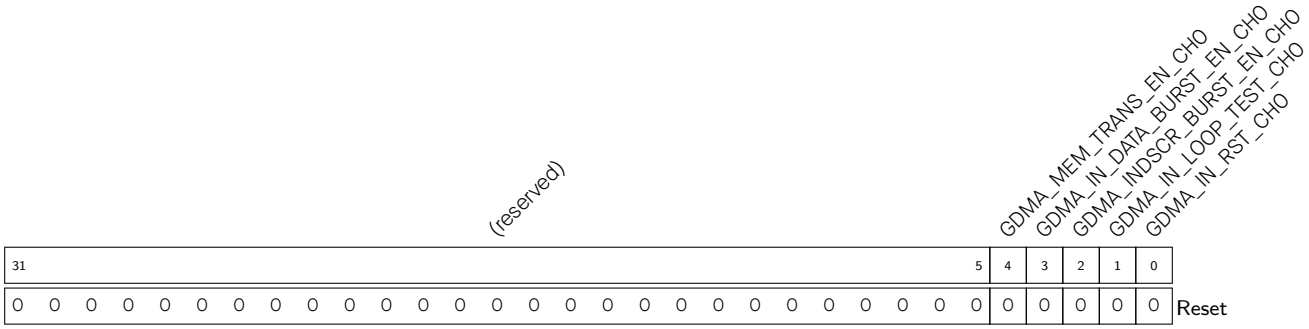
名称	描述	地址	访问
GDMA_OUT_DSCR_CH2_REG	发送通道 2 当前已预读取的发送链表描述符指向的下一个发送链表描述符地址	0x0210	RO
GDMA_OUT_DSCR_BF0_CH2_REG	发送通道 2 当前已预读取的发送链表描述符所在地址	0x0214	RO
GDMA_OUT_DSCR_BF1_CH2_REG	发送通道 2 前一个已预读取的发送链表描述符所在地址	0x0218	RO
GDMA_INFIFO_STATUS_CH3_REG	接收通道 3 的 RX FIFO 状态 0	0x0258	RO
GDMA_IN_STATE_CH3_REG	接收通道 3 的接收状态	0x0264	RO
GDMA_IN_SUC_EOF_DES_ADDR_CH3_REG	接收通道 0 传输完成时的接收链表描述符地址	0x0268	RO
GDMA_IN_ERR_EOF_DES_ADDR_CH3_REG	接收通道 0 发生错误时的接收链表描述符地址	0x026C	RO
GDMA_IN_DSCR_CH3_REG	接收通道 3 已预读取的接收链表描述符指向的下一个接收链表描述符地址	0x0270	RO
GDMA_IN_DSCR_BF0_CH3_REG	接收通道 3 当前已预读取的接收链表描述符所在地址	0x0274	RO
GDMA_IN_DSCR_BF1_CH3_REG	接收通道 3 前一个已预读取的接收链表描述符所在地址	0x0278	RO
GDMA_OUTFIFO_STATUS_CH3_REG	发送通道 3 的 TX FIFO 状态	0x02B8	RO
GDMA_OUT_STATE_CH3_REG	发送通道 3 的发送状态	0x02C4	RO
GDMA_OUT_EOF_DES_ADDR_CH3_REG	发送通道 3 传输完成时的发送链表描述符地址	0x02C8	RO
GDMA_OUT_EOF_BFR_DES_ADDR_CH3_REG	发送通道 0 传输完成时的最后一个发送链表描述符地址	0x02CC	RO
GDMA_OUT_DSCR_CH3_REG	发送通道 3 当前已预读取的发送链表描述符指向的下一个发送链表描述符地址	0x02D0	RO
GDMA_OUT_DSCR_BF0_CH3_REG	发送通道 3 当前已预读取的发送链表描述符所在地址	0x02D4	RO
GDMA_OUT_DSCR_BF1_CH3_REG	发送通道 3 前一个已预读取的发送链表描述符所在地址	0x02D8	RO
GDMA_INFIFO_STATUS_CH4_REG	接收通道 4 的 RX FIFO 状态 0	0x0318	RO
GDMA_IN_STATE_CH4_REG	接收通道 4 的接收状态	0x0324	RO
GDMA_IN_SUC_EOF_DES_ADDR_CH4_REG	接收通道 4 传输完成时的接收链表描述符地址	0x0328	RO
GDMA_IN_ERR_EOF_DES_ADDR_CH4_REG	接收通道 4 发生错误时的接收链表描述符地址	0x032C	RO
GDMA_IN_DSCR_CH4_REG	接收通道 4 已预读取的接收链表描述符指向的下一个接收链表描述符地址	0x0330	RO
GDMA_IN_DSCR_BF0_CH4_REG	接收通道 4 当前已预读取的接收链表描述符所在地址	0x0334	RO
GDMA_IN_DSCR_BF1_CH4_REG	接收通道 4 前一个已预读取的接收链表描述符所在地址	0x0338	RO
GDMA_OUTFIFO_STATUS_CH4_REG	发送通道 4 的 TX FIFO 状态	0x0378	RO
GDMA_OUT_STATE_CH4_REG	发送通道 4 的发送状态	0x0384	RO

名称	描述	地址	访问
GDMA_OUT_EOF_DES_ADDR_CH4_REG	发送通道 4 传输完成时的发送链表描述符地址	0x0388	RO
GDMA_OUT_EOF_BFR_DES_ADDR_CH4_REG	发送通道 4 传输完成时的最后一个发送链表描述符地址	0x038C	RO
GDMA_OUT_DSCR_CH4_REG	发送通道 4 当前已预读取的发送链表描述符指向的下一个发送链表描述符地址	0x0390	RO
GDMA_OUT_DSCR_BF0_CH4_REG	发送通道 4 当前已预读取的发送链表描述符所在地址	0x0394	RO
GDMA_OUT_DSCR_BF1_CH4_REG	发送通道 4 前一个已预读取的发送链表描述符所在地址	0x0398	RO
优先级寄存器			
GDMA_IN_PRI_CH0_REG	接收通道 0 的优先级寄存器	0x0044	R/W
GDMA_OUT_PRI_CH0_REG	发送通道 0 的优先级寄存器	0x00A4	R/W
GDMA_IN_PRI_CH1_REG	接收通道 1 的优先级寄存器	0x0104	R/W
GDMA_OUT_PRI_CH1_REG	发送通道 1 的优先级寄存器	0x0164	R/W
GDMA_IN_PRI_CH2_REG	接收通道 2 的优先级寄存器	0x01C4	R/W
GDMA_OUT_PRI_CH2_REG	发送通道 2 的优先级寄存器	0x0224	R/W
GDMA_IN_PRI_CH3_REG	接收通道 3 的优先级寄存器	0x0284	R/W
GDMA_OUT_PRI_CH3_REG	发送通道 3 的优先级寄存器	0x02E4	R/W
GDMA_IN_PRI_CH4_REG	接收通道 4 的优先级寄存器	0x0344	R/W
GDMA_OUT_PRI_CH4_REG	发送通道 4 的优先级寄存器	0x03A4	R/W
外设选择寄存器			
GDMA_IN_PERI_SEL_CH0_REG	接收通道 0 的外设选择	0x0048	R/W
GDMA_OUT_PERI_SEL_CH0_REG	发送通道 0 的外设选择	0x00A8	R/W
GDMA_IN_PERI_SEL_CH1_REG	接收通道 1 的外设选择	0x0108	R/W
GDMA_OUT_PERI_SEL_CH1_REG	发送通道 1 的外设选择	0x0168	R/W
GDMA_IN_PERI_SEL_CH2_REG	接收通道 2 的外设选择	0x01C8	R/W
GDMA_OUT_PERI_SEL_CH2_REG	发送通道 2 的外设选择	0x0228	R/W
GDMA_IN_PERI_SEL_CH3_REG	接收通道 3 的外设选择	0x0288	R/W
GDMA_OUT_PERI_SEL_CH3_REG	发送通道 3 的外设选择	0x02E8	R/W
GDMA_IN_PERI_SEL_CH4_REG	接收通道 4 的外设选择	0x0348	R/W
GDMA_OUT_PERI_SEL_CH4_REG	发送通道 4 的外设选择	0x03A8	R/W
权限管理状态寄存器			
GDMA_EXTMEM_REJECT_ADDR_REG	被非法访问的外部 RAM 地址	0x03F4	RO
GDMA_EXTMEM_REJECT_ST_REG	被非法访问的外部 RAM 状态	0x03F8	RO
版本寄存器			
GDMA_DATE_REG	版本控制寄存器	0x040C	R/W

3.8 寄存器

本小节的所有地址均为相对于 GDMA 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 3.1. GDMA_IN_CONF0_CH*n*_REG (*n*: 0-4) (0x0000+192**n*)



GDMA_IN_RST_CH*n* 用于复位 GDMA 通道 0 的 RX 状态机和 RX FIFO 指针。(R/W)

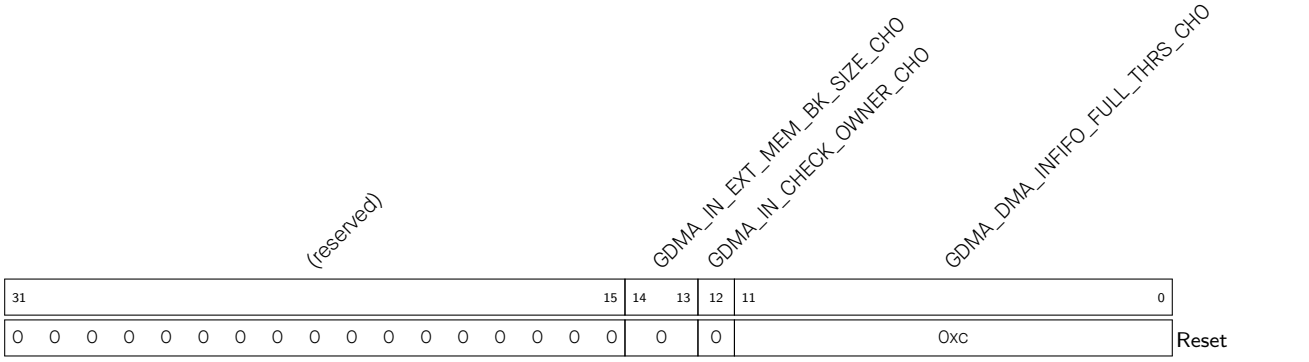
GDMA_IN_LOOP_TEST_CH*n* 保留。(R/W)

GDMA_INDSCR_BURST_EN_CH*n* 将此位置 1，在接收通道 0 访问内部 RAM 读取接收链表描述符时使能 INCR 突发传输。(R/W)

GDMA_IN_DATA_BURST_EN_CH*n* 将此位置 1，在接收通道 0 访问内部 RAM 接收数据时使能 INCR 突发传输。(R/W)

GDMA_MEM_TRANS_EN_CH*n* 将此位置 1，使能 GDMA 存储器到存储器自动传输。(R/W)

Register 3.2. GDMA_IN_CONF1_CH*n*_REG (*n*: 0-4) (0x0004+192**n*)

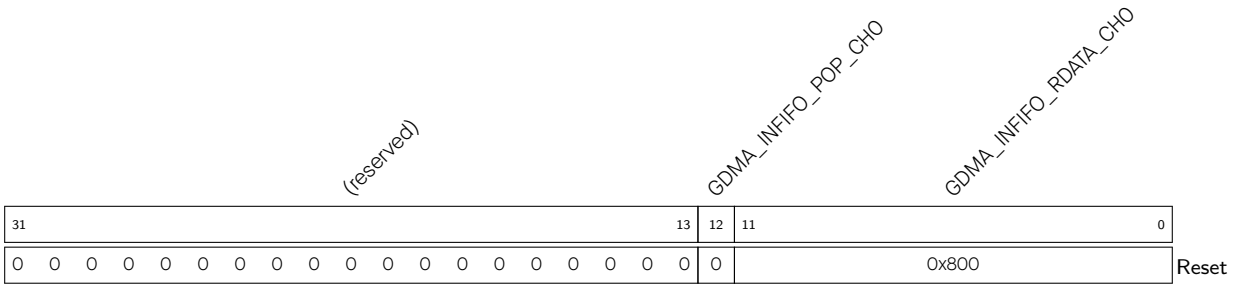


GDMA_DMA_INFIFO_FULL_THRS_CH*n* 接收通道 0 RX FIFO 中接收到的字节数达到该字段的值时，生成 GDMA_INFIFO_FULL_WM_INT 中断。(R/W)

GDMA_IN_CHECK_OWNER_CH*n* 置位此位，使能链表描述符 owner 位检查。(R/W)

GDMA_IN_EXT_MEM_BK_SIZE_CH*n* GDMA 访问外部 RAM 时接收通道 0 的块大小。0：16 字节；1：32 字节；2：64 字节；3：保留。(R/W)

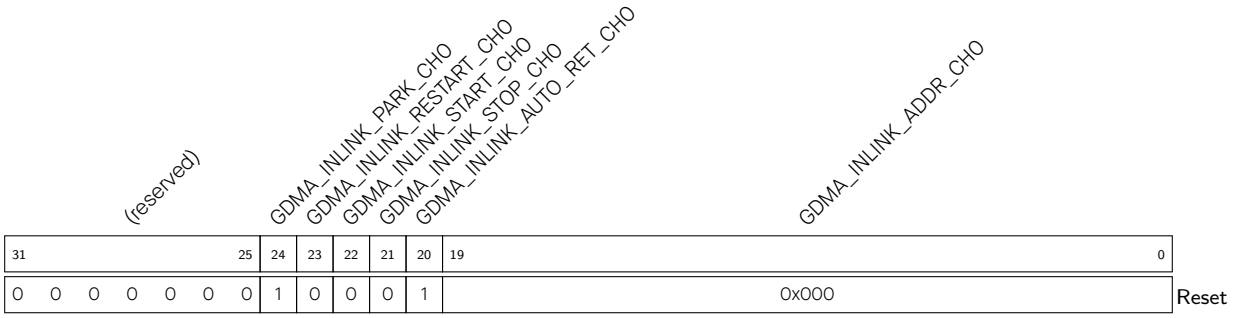
Register 3.3. GDMA_IN_POP_CH*n*_REG (*n*: 0-4) (0x001C+192**n*)



GDMA_INFIFO_RDATA_CH*n* 存储从 GDMA FIFO 中弹出的数据（用于排错）。(RO)

GDMA_INFIFO_POP_CH*n* 置位此位，从 GDMA FIFO 中弹出数据（用于排错）。(R/W/SC)

Register 3.4. GDMA_IN_LINK_CH*n*_REG (*n*: 0-4) (0x0020+192**n*)



GDMA_INLINK_ADDR_CH*n* 存储第一个接收链表描述符地址的低 20 位。(R/W)

GDMA_INLINK_AUTO_RET_CH*n* 当前接收数据有错误时，置位此位返回当前接收链表描述符的地址。(R/W)

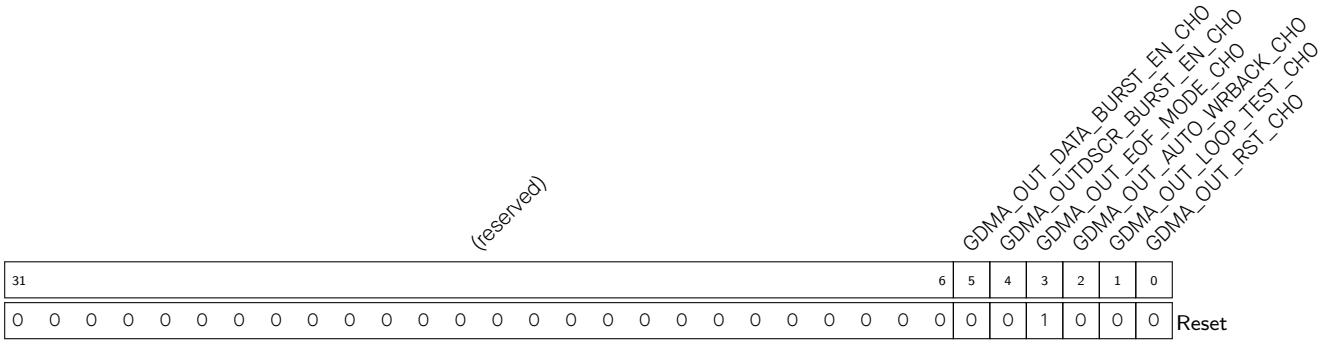
GDMA_INLINK_STOP_CH*n* 置位此位，停止处理接收链表描述符。(R/W/SC)

GDMA_INLINK_START_CH*n* 置位此位，开始处理接收链表描述符。(R/W/SC)

GDMA_INLINK_RESTART_CH*n* 置位此位，挂载新的接收链表描述符。(R/W/SC)

GDMA_INLINK_PARK_CH*n* 1：接收链表描述符的状态机空闲；0：接收链表描述符的状态机工作中。(RO)

Register 3.5. GDMA_OUT_CONF0_CH*n*_REG (*n*: 0-4) (0x0060+192**n*)



GDMA_OUT_RST_CH*n* 用于复位 GDMA 通道 0 的 TX 状态机和 TX FIFO 指针。(R/W)

GDMA_OUT_LOOP_TEST_CH*n* 保留。(R/W)

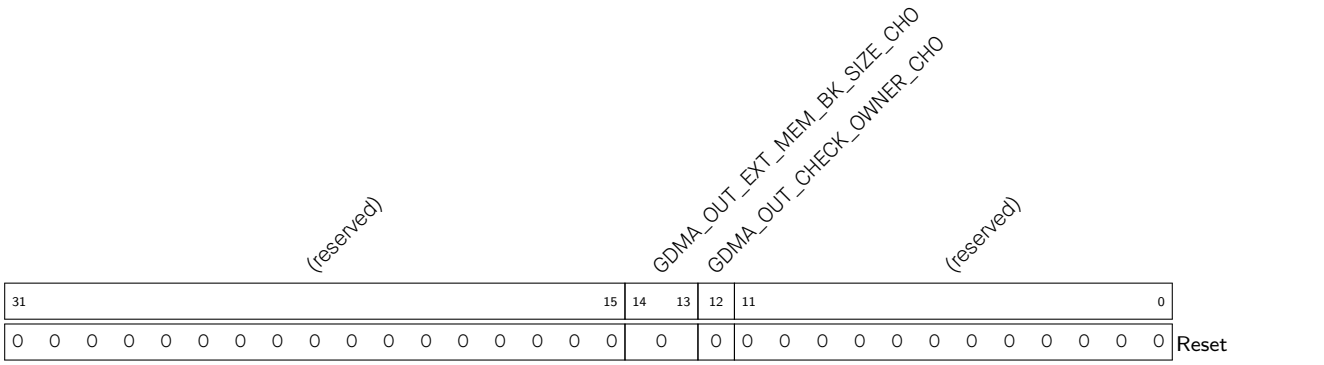
GDMA_OUT_AUTO_WRBACK_CH*n* 置位此位，在 TX FIFO 中所有数据发送出去后自动回写发送链表。(R/W)

GDMA_OUT_EOF_MODE_CH*n* 发送数据时生成 EOF 标志位。1：需要发送的数据已从 GDMA FIFO 中弹出时，发送通道 0 的 EOF 标志生成。(R/W)

GDMA_OUTDSCR_BURST_EN_CH*n* 将此位置 1，在发送通道 0 访问内部 RAM 读取发送链表描述符时使能 INCR 突发传输。(R/W)

GDMA_OUT_DATA_BURST_EN_CH*n* 将此位置 1，在发送通道 0 访问内部 RAM 发送数据时使能 INCR 突发传输。(R/W)

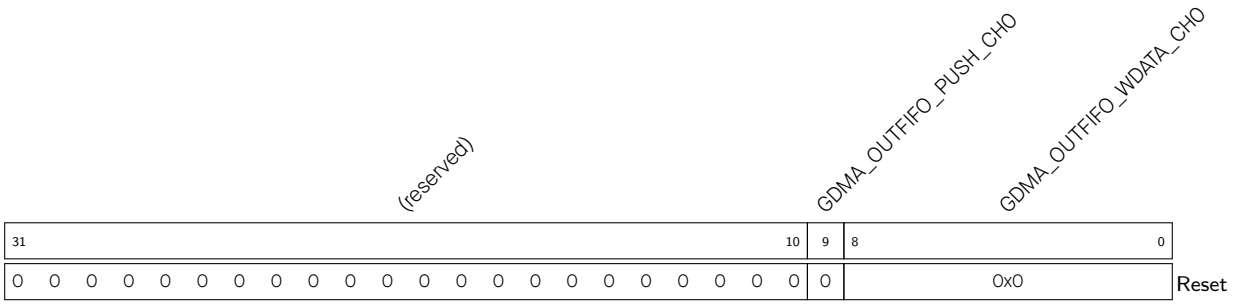
Register 3.6. GDMA_OUT_CONF1_CH*n*_REG (*n*: 0-4) (0x0064+192**n*)



GDMA_OUT_CHECK_OWNER_CH*n* 置位此位，使能链表描述符的 owner 位检查。(R/W)

GDMA_OUT_EXT_MEM_BK_SIZE_CH*n* GDMA 访问外部 RAM 时接收通道 0 的块大小。0：16 字节；1：32 字节；2：64 字节；3：保留。(R/W)

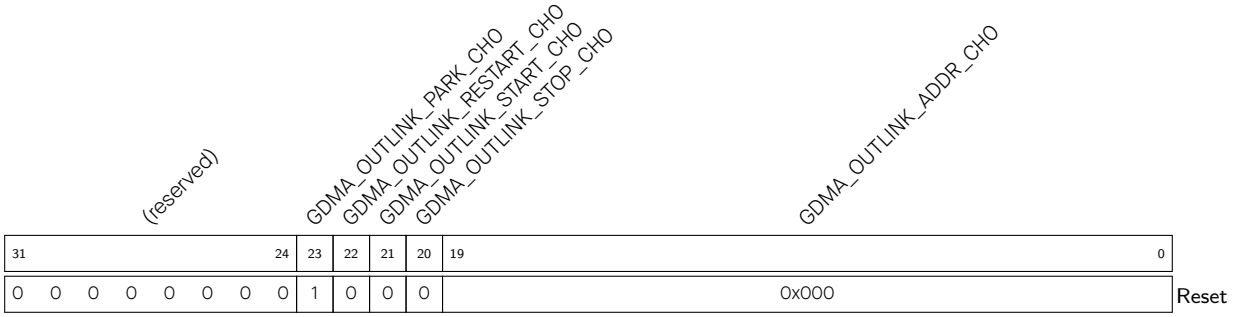
Register 3.7. GDMA_OUT_PUSH_CH*n*_REG (*n*: 0-4) (0x007C+192**n*)



GDMA_OUTFIFO_WDATA_CH*n* 存储需推送至 GDMA FIFO 的数据。(R/W)

GDMA_OUTFIFO_PUSH_CH*n* 置位此位，将数据推送至 GDMA FIFO 中。(R/W/SC)

Register 3.8. GDMA_OUT_LINK_CH*n*_REG (*n*: 0-4) (0x0080+192**n*)



GDMA_OUTLINK_ADDR_CH*n* 存储第一个发送链表描述符地址的低 20 位。(R/W)

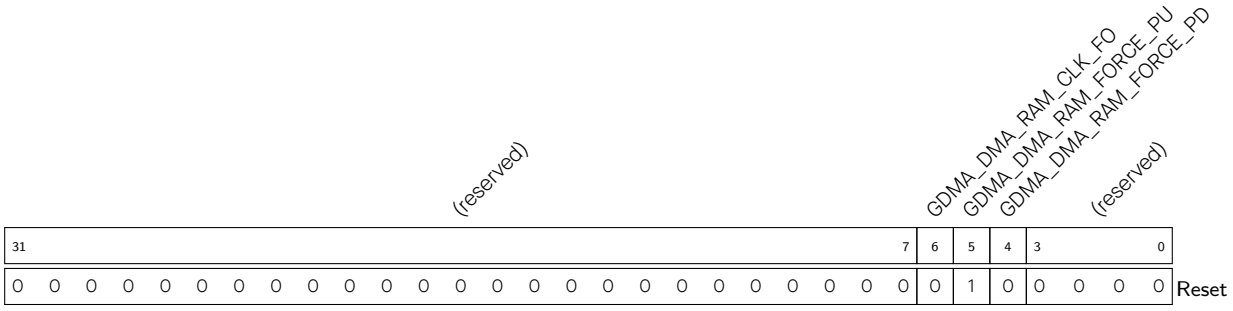
GDMA_OUTLINK_STOP_CH*n* 置位此位，停止处理发送链表描述符。(R/W/SC)

GDMA_OUTLINK_START_CH*n* 置位此位，开始处理发送链表描述符。(R/W/SC)

GDMA_OUTLINK_RESTART_CH*n* 置位此位，在最后一个地址挂载新的发送链表。(R/W/SC)

GDMA_OUTLINK_PARK_CH*n* 1：发送链表描述符的状态机空闲；0：发送链表描述符的状态机工作中。(RO)

Register 3.9. GDMA_PD_CONF_REG (0x03C4)

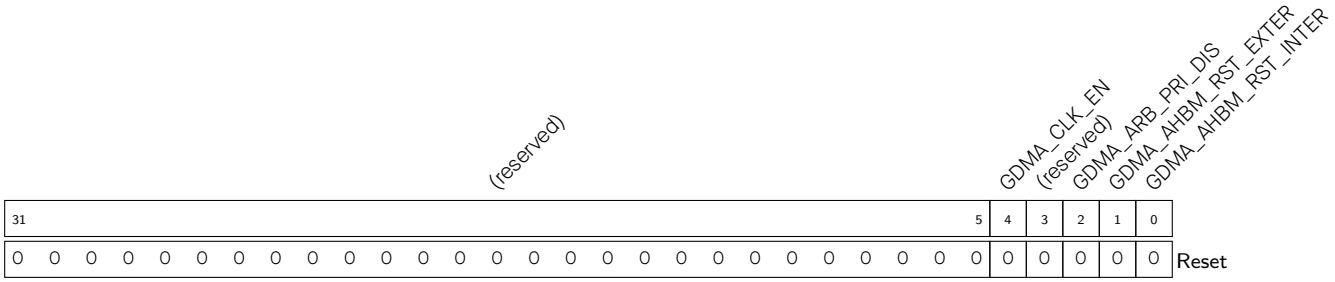


GDMA_DMA_RAM_FORCE_PD 置位此位，强制关闭 GDMA 内部存储器。(R/W)

GDMA_DMA_RAM_FORCE_PU 置位此位，强制开启 GDMA 内部存储器。(R/W)

GDMA_DMA_RAM_CLK_FO 1: 在访问 GDMA RAM 时强制开启时钟，绕过门控时钟；0: 在访问 GDMA RAM 时使用门控时钟。(R/W)

Register 3.10. GDMA_MISC_CONF_REG (0x03C8)



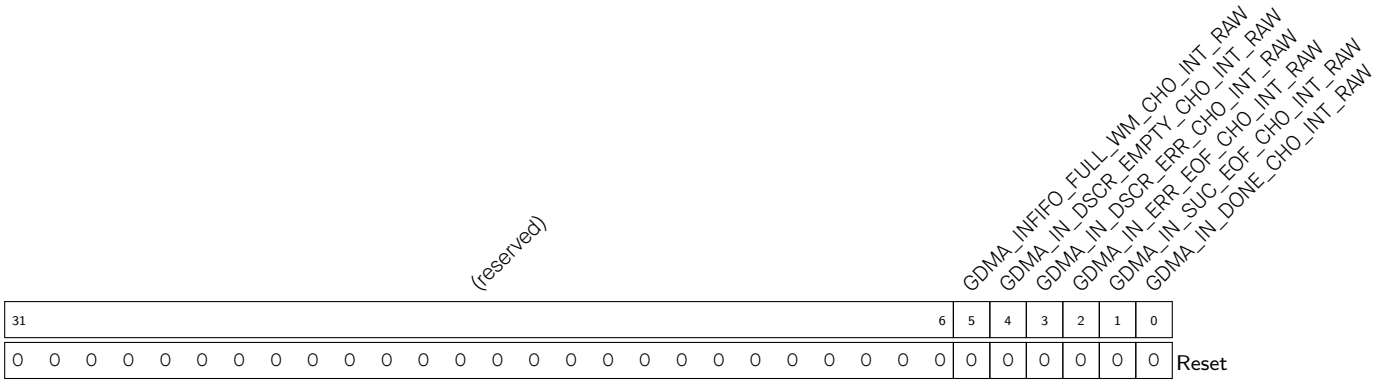
GDMA_AHBM_RST_INTER 置位此位，然后清零此位，重置内部 AHB 状态机。(R/W)

GDMA_AHBM_RST_EXTER 置位此位，然后清零此位，重置外部 AHB 状态机。(R/W)

GDMA_ARB_PRI_DIS 置位此位，关闭优先级仲裁功能。(R/W)

GDMA_CLK_EN 1：强制开启寄存器的时钟；0：仅在应用写寄存器时开启时钟。(R/W)

Register 3.11. GDMA_IN_INT_RAW_CH*n*_REG (*n*: 0-4) (0x0008+192**n*)



GDMA_IN_DONE_CH*n*_INT_RAW 接收通道 0 接收到接收链表描述符指向的最后一个字节数据时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_IN_SUC_EOF_CH*n*_INT_RAW 接收通道 0 接收到接收链表描述符指向的最后一个字节数据且描述符的 suc_eof 位为 1 时，该原始中断位翻转至高电平。对于 UHCI0，接收通道 0 接收到接收链表描述符指向的最后一个字节数据且没有检测到数据错误时，该原始中断位翻转至高电平。(R/WTC/SS)

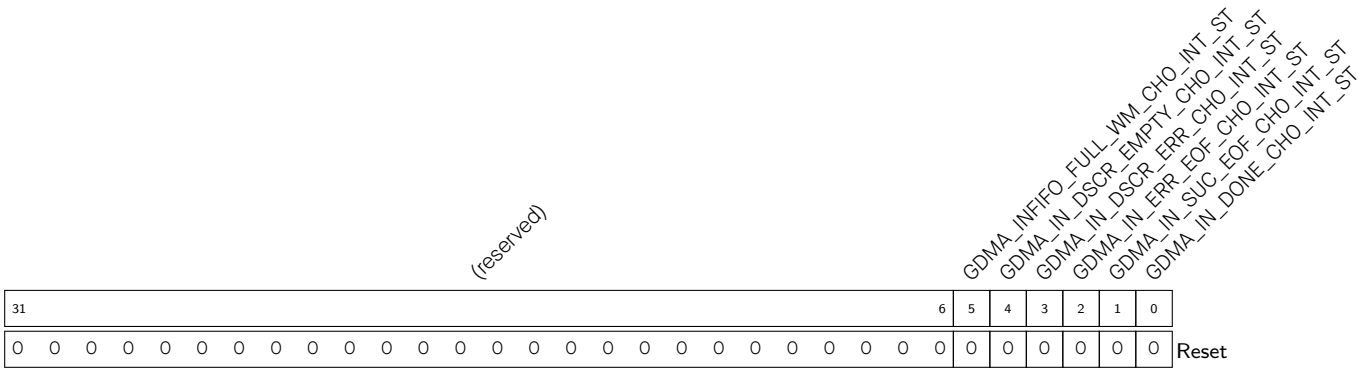
GDMA_IN_ERR_EOF_CH*n*_INT_RAW 仅在接收通道 0 连接 UHCI0，且检测到数据错误时，该原始中断位翻转至高电平。如通道连接其他外设，该原始中断保留。(R/WTC/SS)

GDMA_IN_DSCR_ERR_CH*n*_INT_RAW 接收通道 0 检测到接收链表描述符错误时，包括 owner 位错误、接收链表描述符的第二个字和第三个字错误，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_IN_DSCR_EMPTY_CH*n*_INT_RAW 接收通道 0 接收链表指向的 RX FIFO 已满，数据接收未完成，但没有更多接收链表时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_INFIFO_FULL_WM_CH*n*_INT_RAW 接收通道 0 RX FIFO 中接收的字节数达到 GDMA_DMA_INFIFO_FULL_THRS_CH0 的值时，该原始中断位翻转至高电平。(R/ WTC/SS)

Register 3.12. GDMA_IN_INT_ST_CH*n*_REG (*n*: 0-4) (0x000C+192**n*)



GDMA_IN_DONE_CH*n*_INT_ST GDMA_IN_DONE_CH_INT 中断的原始状态位。(RO)

GDMA_IN_SUC_EOF_CH*n*_INT_ST GDMA_IN_SUC_EOF_CH_INT 中断的原始状态位。(RO)

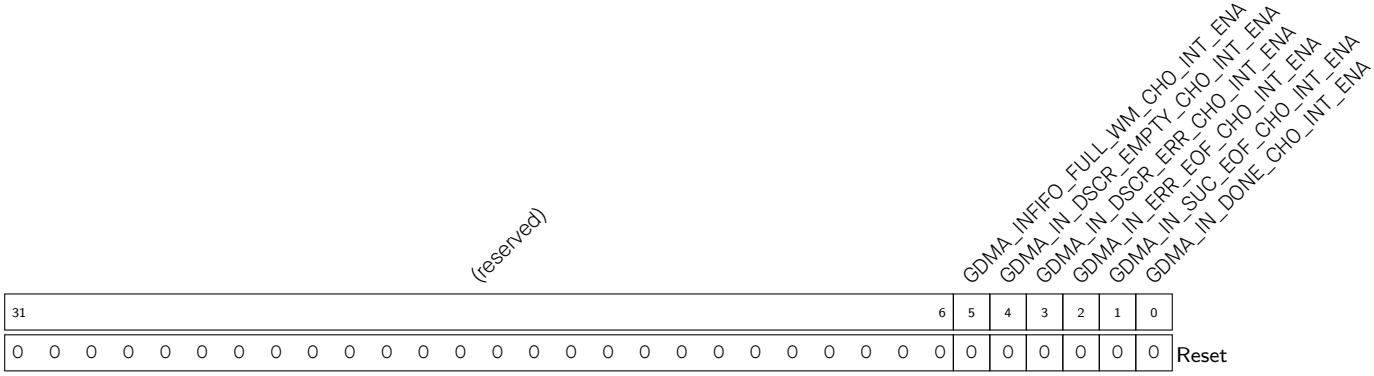
GDMA_IN_ERR_EOF_CH*n*_INT_ST GDMA_IN_ERR_EOF_CH_INT 中断的原始状态位。(RO)

GDMA_IN_DSCR_ERR_CH*n*_INT_ST GDMA_IN_DSCR_ERR_CH_INT 中断的原始状态位。(RO)

GDMA_IN_DSCR_EMPTY_CH*n*_INT_ST GDMA_IN_DSCR_EMPTY_CH_INT 中断的原始状态位。(RO)

GDMA_INFIFO_FULL_WM_CH*n*_INT_ST GDMA_INFIFO_FULL_WM_CH_INT 中断的原始状态位。(RO)

Register 3.13. GDMA_IN_INT_ENA_CH*n*_REG (*n*: 0-4) (0x0010+192**n*)



GDMA_IN_DONE_CH*n*_INT_ENA GDMA_IN_DONE_CH_INT 中断的使能位。(R/W)

GDMA_IN_SUC_EOF_CH*n*_INT_ENA GDMA_IN_SUC_EOF_CH_INT 中断的使能位。(R/W)

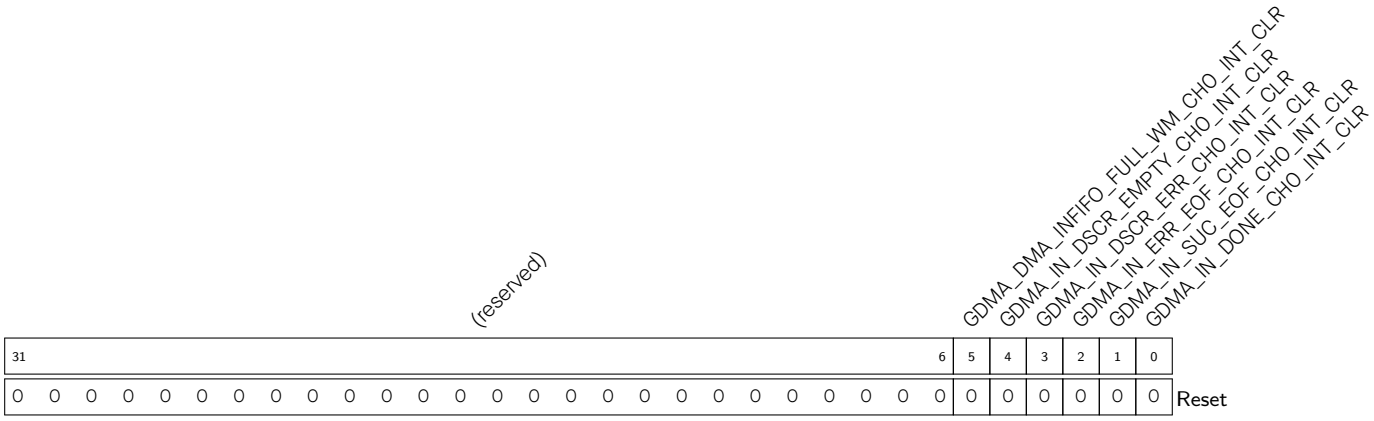
GDMA_IN_ERR_EOF_CH*n*_INT_ENA GDMA_IN_ERR_EOF_CH_INT 中断的使能位。(R/W)

GDMA_IN_DSCR_ERR_CH*n*_INT_ENA GDMA_IN_DSCR_ERR_CH_INT 中断的使能位。(R/W)

GDMA_IN_DSCR_EMPTY_CH*n*_INT_ENA GDMA_IN_DSCR_EMPTY_CH_INT 中断的使能位。(R/W)

GDMA_INFIFO_FULL_WM_CH*n*_INT_ENA GDMA_INFIFO_FULL_WM_CH_INT 中断的使能位。(R/W)

Register 3.14. GDMA_IN_INT_CLR_CH*n*_REG (*n*: 0-4) (0x0014+192**n*)



GDMA_IN_DONE_CH*n*_INT_CLR 置位此位，清除 GDMA_IN_DONE_CH_INT 中断。(WT)

GDMA_IN_SUC_EOF_CH*n*_INT_CLR 置位此位，清除 GDMA_IN_SUC_EOF_CH_INT 中断。(WT)

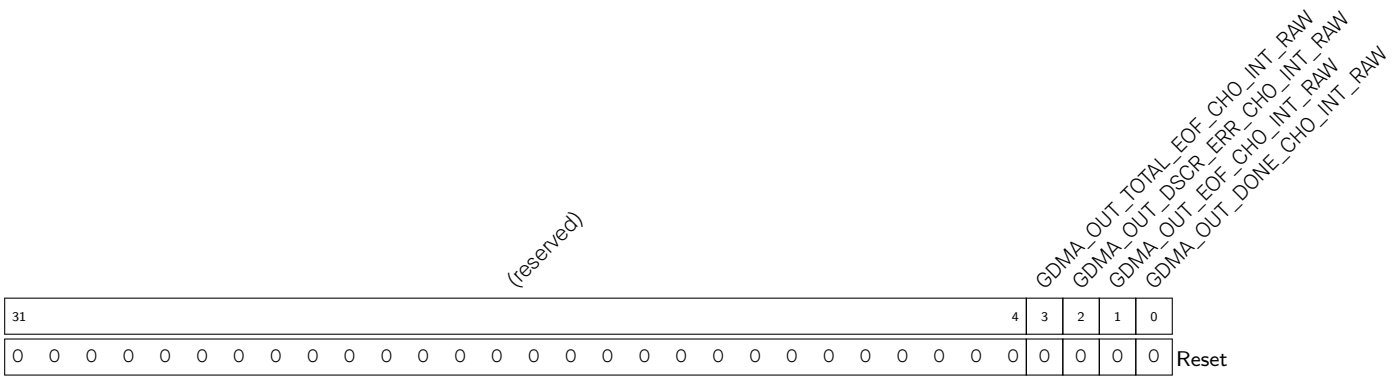
GDMA_IN_ERR_EOF_CH*n*_INT_CLR 置位此位，清除 GDMA_IN_ERR_EOF_CH_INT 中断。(WT)

GDMA_IN_DSCR_ERR_CH*n*_INT_CLR 置位此位，清除 GDMA_IN_DSCR_ERR_CH_INT 中断。(WT)

GDMA_IN_DSCR_EMPTY_CH*n*_INT_CLR 置位此位，清除 GDMA_IN_DSCR_EMPTY_CH_INT 中断。(WT)

GDMA_DMA_INFIFO_FULL_WM_CH*n*_INT_CLR 置位此位，清除 INFIFO_FULL_WM_CH_INT 中断。(WT)

Register 3.15. GDMA_OUT_INT_RAW_CH*n*_REG (*n*: 0-4) (0x0068+192**n*)



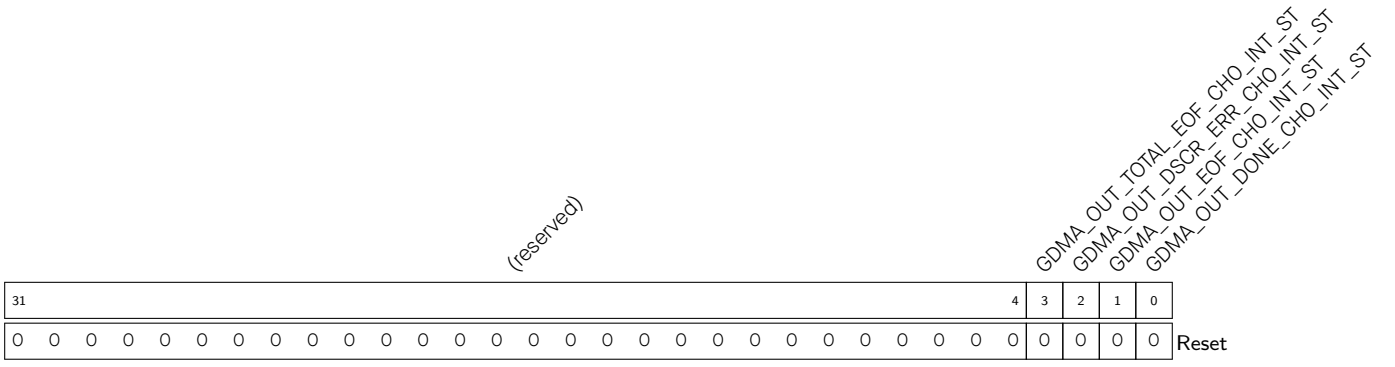
GDMA_OUT_DONE_CH*n*_INT_RAW 发送通道 0 向外设发送了发送链表描述符指向的最后一个字节数据时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_OUT_EOF_CH*n*_INT_RAW 发送通道 0 从存储器读取了发送链表描述符指向的最后一个字节数据时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_OUT_DSCR_ERR_CH*n*_INT_RAW 发送通道 0 检测到发送链表描述符错误时，包括 owner 位错误、发送链表描述符的第二个字和第三个字错误，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_OUT_TOTAL_EOF_CH*n*_INT_RAW 发送通道 0 发送了发送链表（包括一个或几个发送链表描述符）对应的数据时，该原始中断位翻转至高电平。(R/WTC/SS)

Register 3.16. GDMA_OUT_INT_ST_CH*n*_REG (*n*: 0-4) (0x006C+192**n*)



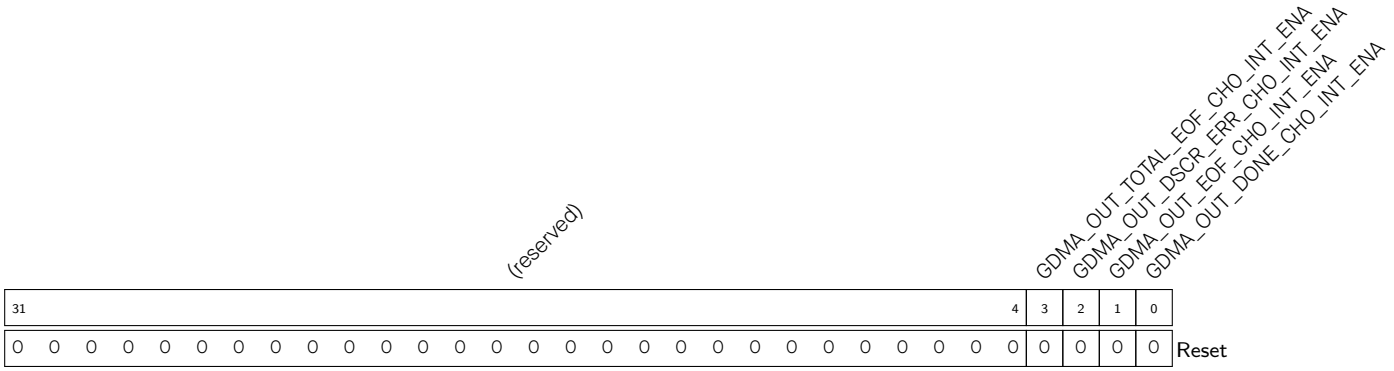
GDMA_OUT_DONE_CH*n*_INT_ST GDMA_OUT_DONE_CH_INT 中断的原始状态位。(RO)

GDMA_OUT_EOF_CH*n*_INT_ST GDMA_OUT_EOF_CH_INT 中断的原始状态位。(RO)

GDMA_OUT_DSCR_ERR_CH*n*_INT_ST GDMA_OUT_DSCR_ERR_CH_INT 中断的原始状态位。(RO)

GDMA_OUT_TOTAL_EOF_CH*n*_INT_ST GDMA_OUT_TOTAL_EOF_CH_INT 中断的原始状态位。(RO)

Register 3.17. GDMA_OUT_INT_ENA_CH*n*_REG (*n*: 0-4) (0x0070+192**n*)



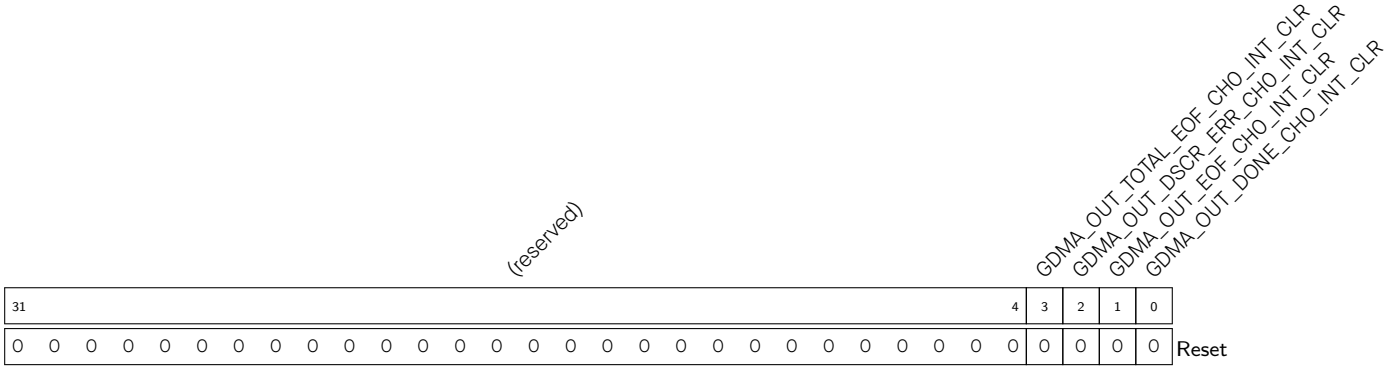
GDMA_OUT_DONE_CH*n*_INT_ENA GDMA_OUT_DONE_CH_INT 中断的使能位。(R/W)

GDMA_OUT_EOF_CH*n*_INT_ENA GDMA_OUT_EOF_CH_INT 中断的使能位。(R/W)

GDMA_OUT_DSCR_ERR_CH*n*_INT_ENA GDMA_OUT_DSCR_ERR_CH_INT 中断的使能位。(R/W)

GDMA_OUT_TOTAL_EOF_CH*n*_INT_ENA GDMA_OUT_TOTAL_EOF_CH_INT 中断的使能位。(R/W)

Register 3.18. GDMA_OUT_INT_CLR_CH*n*_REG (*n*: 0-4) (0x0074+192**n*)



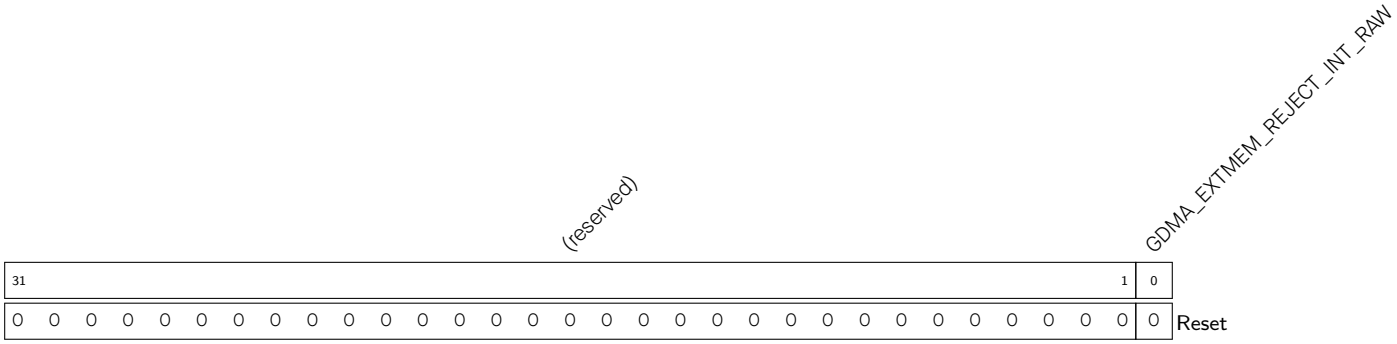
GDMA_OUT_DONE_CH*n*_INT_CLR 置位此位，清除 GDMA_OUT_DONE_CH_INT 中断。(WT)

GDMA_OUT_EOF_CH*n*_INT_CLR 置位此位，清除 GDMA_OUT_EOF_CH_INT 中断。(WT)

GDMA_OUT_DSCR_ERR_CH*n*_INT_CLR 置位此位，清除 GDMA_OUT_DSCR_ERR_CH_INT 中断。(WT)

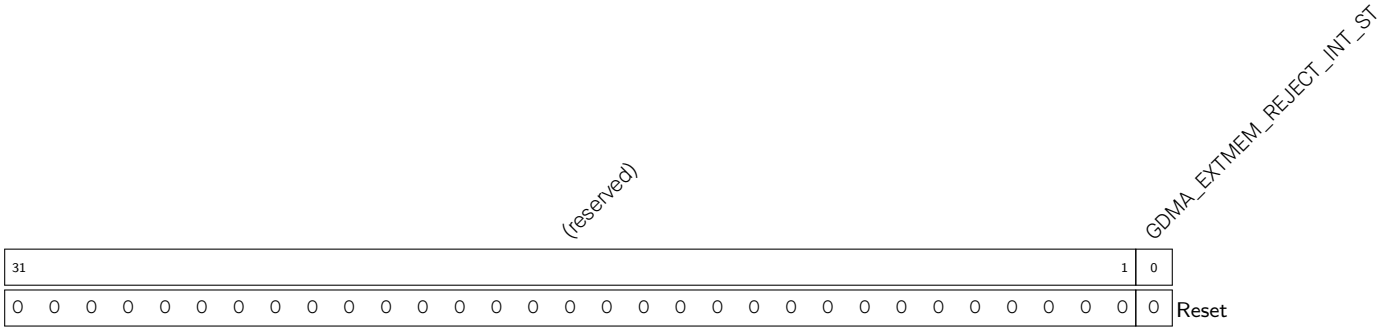
GDMA_OUT_TOTAL_EOF_CH*n*_INT_CLR 置位此位，清除 GDMA_OUT_TOTAL_EOF_CH_INT 中断。(WT)

Register 3.19. GDMA_EXTMEM_REJECT_INT_RAW_REG (0x03FC)



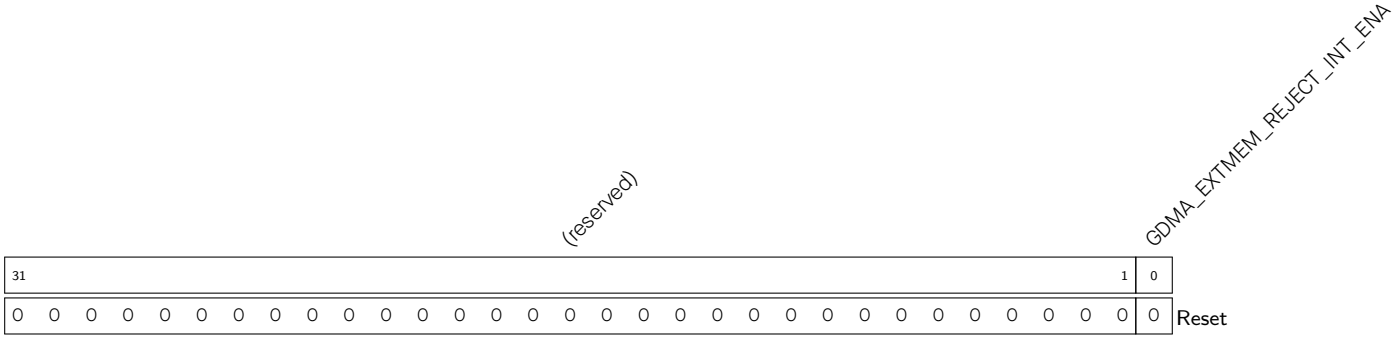
GDMA_EXTMEM_REJECT_INT_RAW 权限管理模块拒绝了对外部 RAM 的非法访问时，该原始中断位翻转至高电平。(R/WTC/SS)

Register 3.20. GDMA_EXTMEM_REJECT_INT_ST_REG (0x0400)



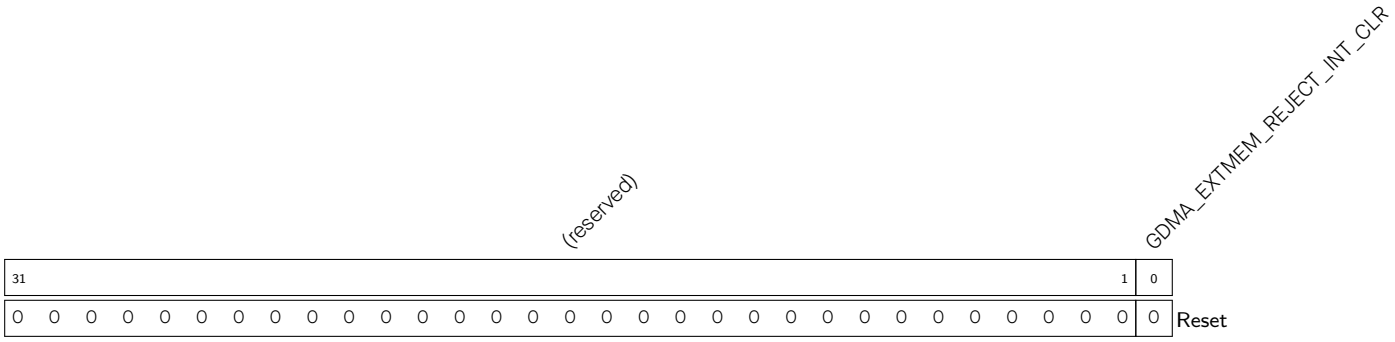
GDMA_EXTMEM_REJECT_INT_ST GDMA_EXTMEM_REJECT_INT 中断的原始状态位。(RO)

Register 3.21. GDMA_EXTMEM_REJECT_INT_ENA_REG (0x0404)



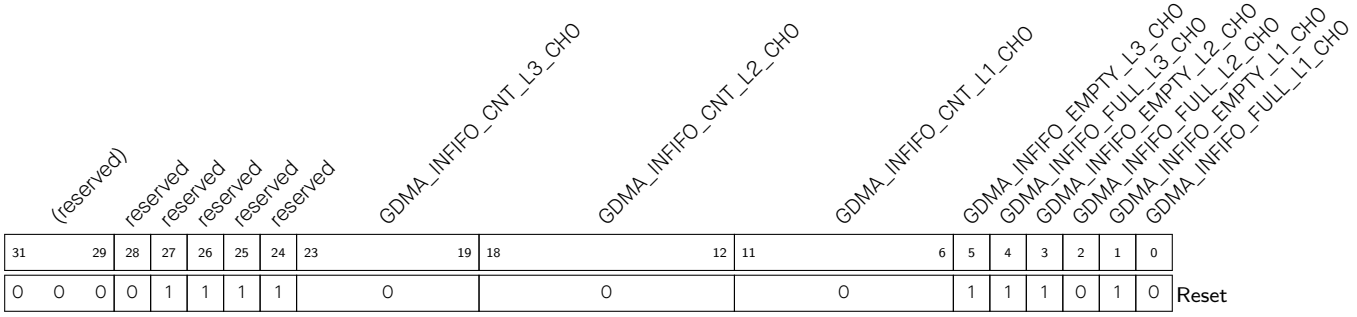
GDMA_EXTMEM_REJECT_INT_ENA GDMA_EXTMEM_REJECT_INT 中断的使能位。(R/W)

Register 3.22. GDMA_EXTMEM_REJECT_INT_CLR_REG (0x0408)



GDMA_EXTMEM_REJECT_INT_CLR 置位此位，清除 GDMA_EXTMEM_REJECT_INT 中断。(WT)

Register 3.23. GDMA_INFIFO_STATUS_CH*n*_REG (*n*: 0-4) (0x0018+192**n*)



GDMA_INFIFO_FULL_L1_CH*n* 接收通道 0 的 L1 RX FIFO 已满。(RO)

GDMA_INFIFO_EMPTY_L1_CH*n* 接收通道 0 的 L1 RX FIFO 为空。(RO)

GDMA_INFIFO_FULL_L2_CH*n* 接收通道 0 的 L2 RX FIFO 已满。(RO)

GDMA_INFIFO_EMPTY_L2_CH*n* 接收通道 0 的 L2 RX FIFO 为空。(RO)

GDMA_INFIFO_FULL_L3_CH*n* 接收通道 0 的 L3 RX FIFO 已满。(RO)

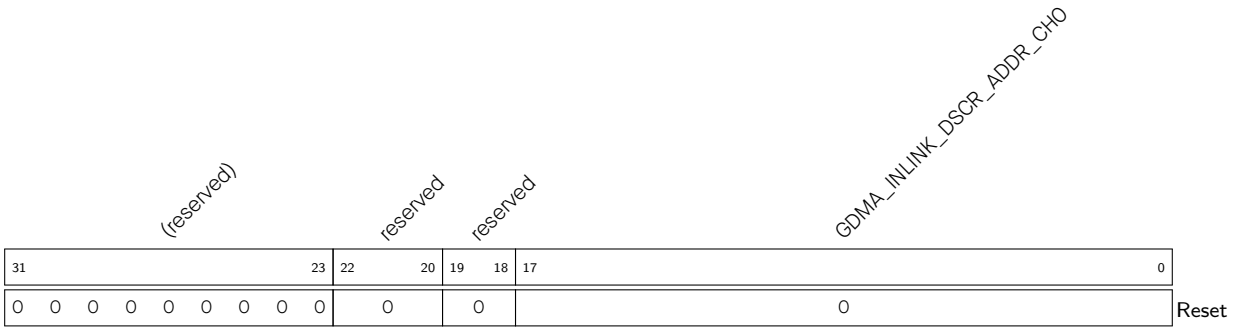
GDMA_INFIFO_EMPTY_L3_CH*n* 接收通道 0 的 L3 RX FIFO 为空。(RO)

GDMA_INFIFO_CNT_L1_CH*n* 接收通道 0 的 L1 RX FIFO 存储的字节数。(RO)

GDMA_INFIFO_CNT_L2_CH*n* 接收通道 0 的 L2 RX FIFO 存储的字节数。(RO)

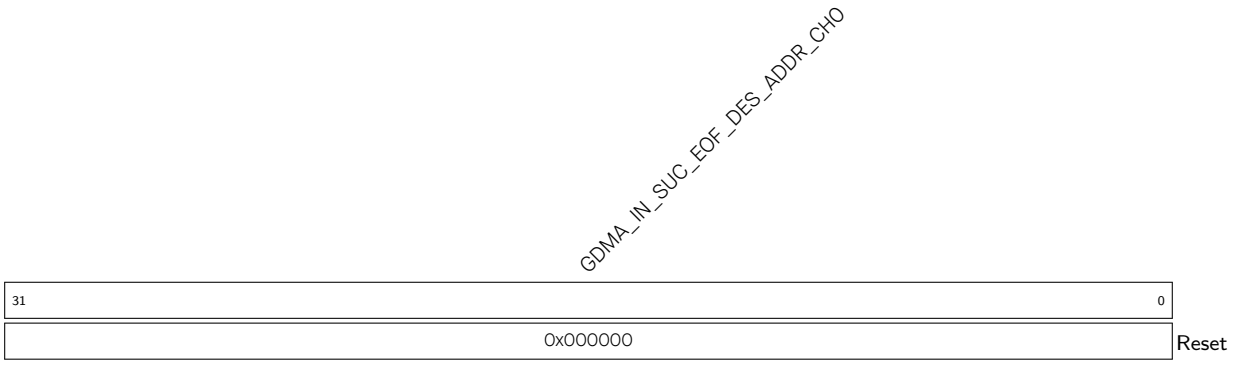
GDMA_INFIFO_CNT_L3_CH*n* 接收通道 0 的 L3 RX FIFO 存储的字节数。(RO)

Register 3.24. GDMA_IN_STATE_CH*n*_REG (*n*: 0-4) (0x0024+192**n*)



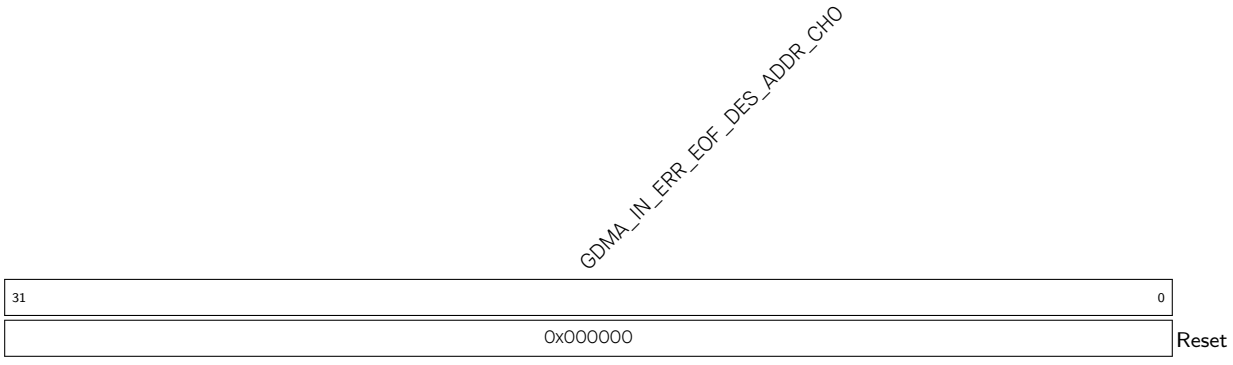
GDMA_INLINK_DSCR_ADDR_CH*n* 表示下一个预读取（但未处理）的接收链表描述符所在地址的低 18 位。如果当前处理的接收链表描述符是最后一个描述符，则该字段表示当前处理的接收链表描述符地址。(RO)

Register 3.25. GDMA_IN_SUC_EOF_DES_ADDR_CH*n*_REG (*n*: 0-4) (0x0028+192**n*)



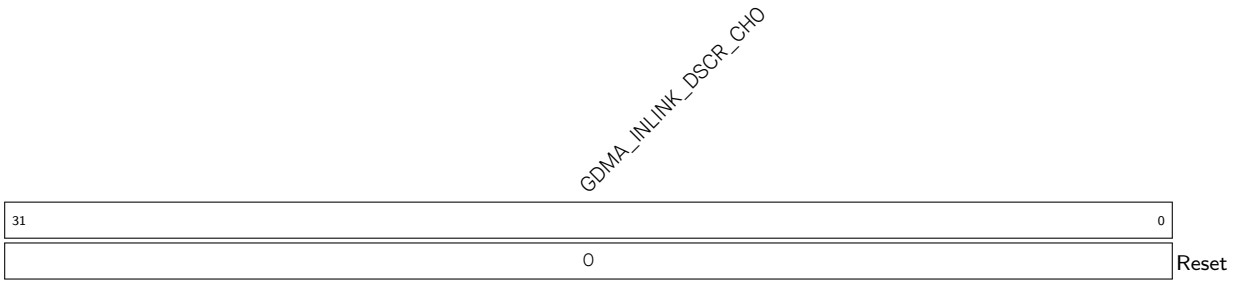
GDMA_IN_SUC_EOF_DES_ADDR_CH*n* 接收链表描述符的 EOF 为 1 时，该描述符的地址。(RO)

Register 3.26. GDMA_IN_ERR_EOF_DES_ADDR_CH*n*_REG (*n*: 0-4) (0x002C+192**n*)



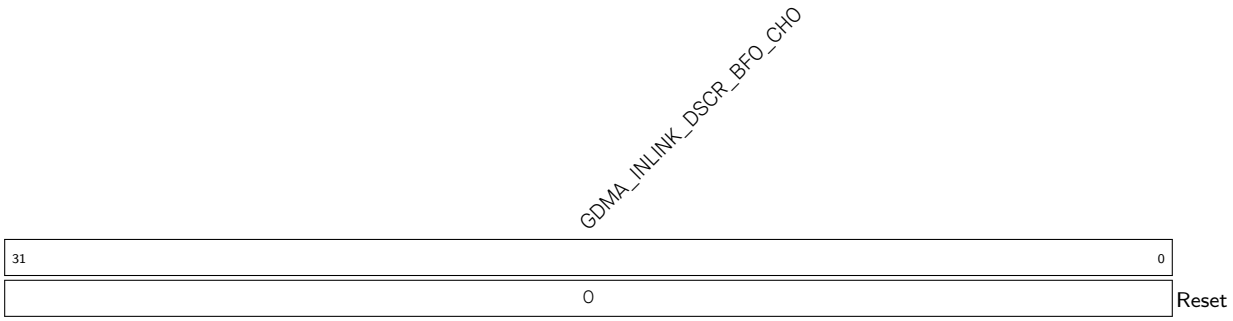
GDMA_IN_ERR_EOF_DES_ADDR_CH*n* 当前接收数据有错误时，接收链表描述符的地址。仅在连接外设为 UHCI 0 时使用。(RO)

Register 3.27. GDMA_IN_DSCR_CH*n*_REG (*n*: 0-4) (0x0030+192**n*)



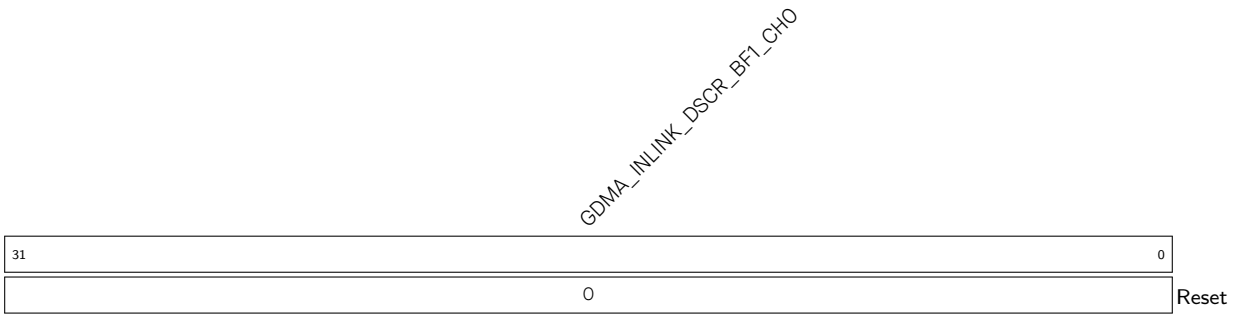
GDMA_INLINK_DSCR_CH*n* 表示当前已预读取的接收链表描述符指向的下一个接收链表描述符地址 x+1。(RO)

Register 3.28. GDMA_IN_DSCR_BF0_CH*n*_REG (*n*: 0-4) (0x0034+192**n*)



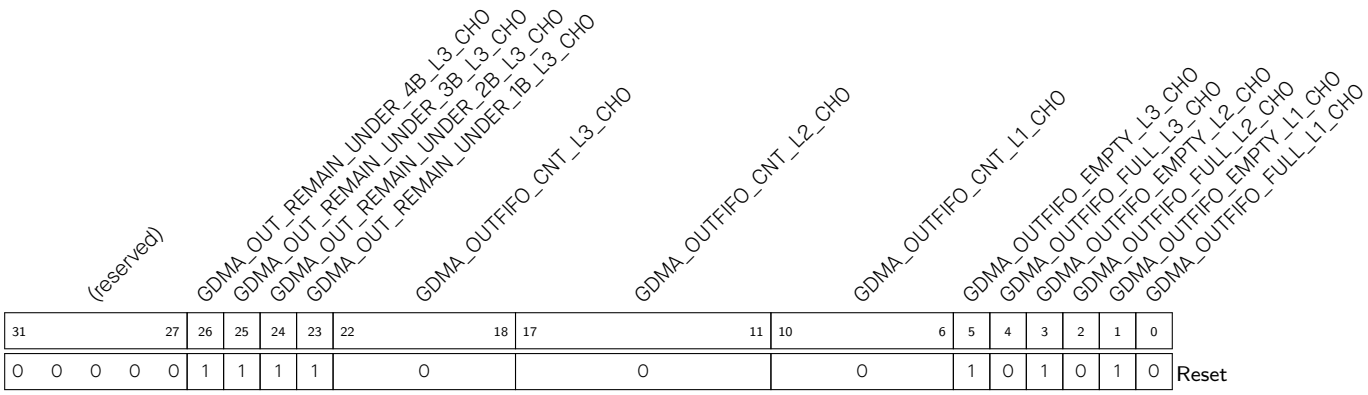
GDMA_INLINK_DSCR_BF0_CH*n* 表示当前已预读取的接收链表描述符所在地址 x。(RO)

Register 3.29. GDMA_IN_DSCR_BF1_CH*n*_REG (*n*: 0-4) (0x0038+192**n*)



GDMA_INLINK_DSCR_BF1_CH*n* 表示前一个已预读取的接收链表描述符所在地址 x-1。(RO)

Register 3.30. GDMA_OUTFIFO_STATUS_CH*n*_REG (*n*: 0-4) (0x0078+192**n*)



GDMA_OUTFIFO_FULL_L1_CH*n* 发送通道 0 的 L1 TX FIFO 已满。(RO)

GDMA_OUTFIFO_EMPTY_L1_CH*n* 发送通道 0 的 L1 TX FIFO 为空。(RO)

GDMA_OUTFIFO_FULL_L2_CH*n* 发送通道 0 的 L2 TX FIFO 已满。(RO)

GDMA_OUTFIFO_EMPTY_L2_CH*n* 发送通道 0 的 L2 TX FIFO 为空。(RO)

GDMA_OUTFIFO_FULL_L3_CH*n* 发送通道 0 的 L3 TX FIFO 已满。(RO)

GDMA_OUTFIFO_EMPTY_L3_CH*n* 发送通道 0 的 L3 TX FIFO 为空。(RO)

GDMA_OUTFIFO_CNT_L1_CH*n* 发送通道 0 的 L1 TX FIFO 存储的字节数。(RO)

GDMA_OUTFIFO_CNT_L2_CH*n* 发送通道 0 的 L2 TX FIFO 存储的字节数。(RO)

GDMA_OUTFIFO_CNT_L3_CH*n* 发送通道 0 的 L3 TX FIFO 存储的字节数。(RO)

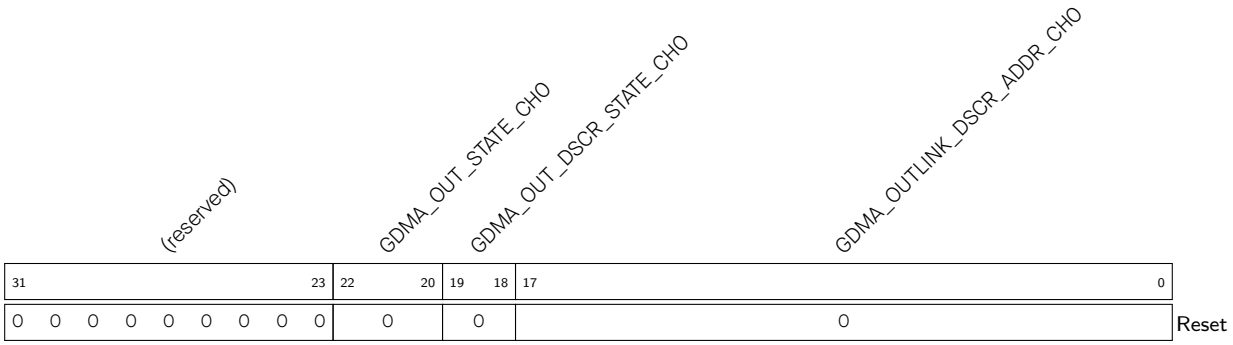
GDMA_OUT_REMAIN_UNDER_1B_L3_CH*n* 保留。(RO)

GDMA_OUT_REMAIN_UNDER_2B_L3_CH*n* 保留。(RO)

GDMA_OUT_REMAIN_UNDER_3B_L3_CH*n* 保留。(RO)

GDMA_OUT_REMAIN_UNDER_4B_L3_CH*n* 保留。(RO)

Register 3.31. GDMA_OUT_STATE_CH*n*_REG (*n*: 0-4) (0x0084+192**n*)

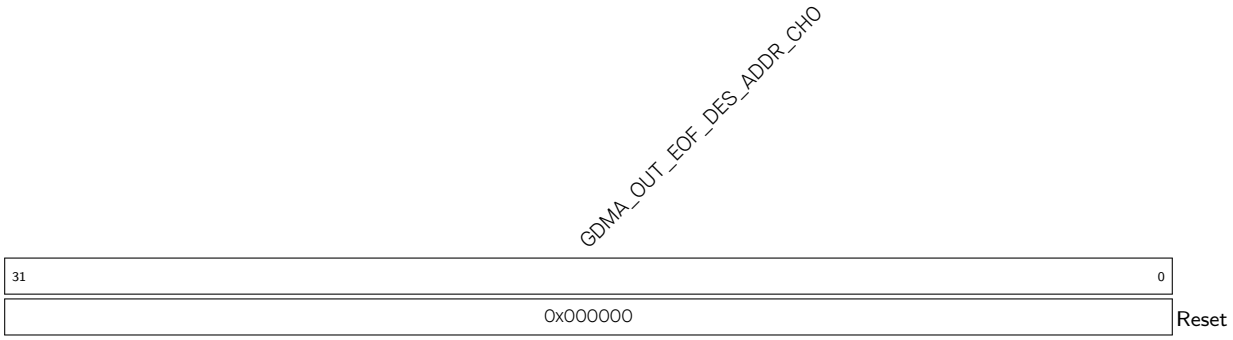


GDMA_OUTLINK_DSCR_ADDR_CH*n* 表示下一个预读取（但未处理）的发送接收链表描述符所在地址的低 18 位。如果当前处理的发送链表描述符是最后一个描述符，则该字段表示当前处理的发送链表描述符地址。(RO)

GDMA_OUT_DSCR_STATE_CH*n* 保留。(RO)

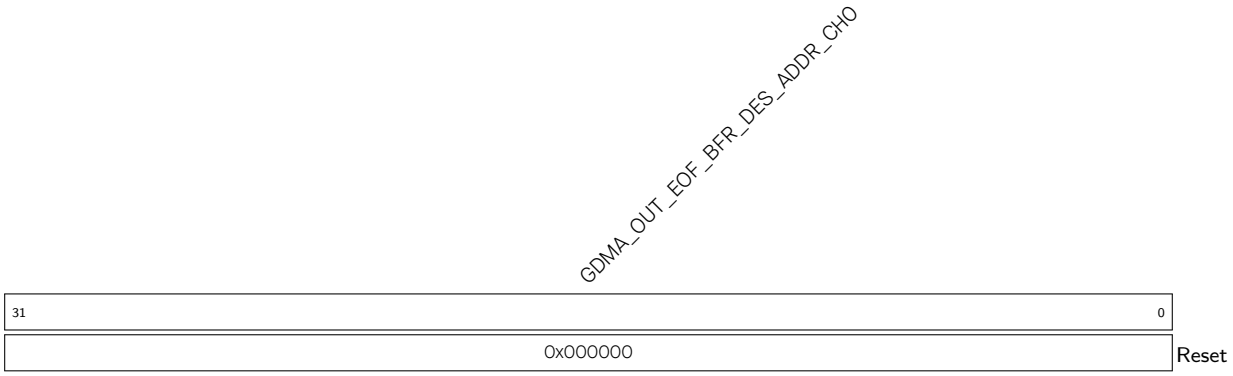
GDMA_OUT_STATE_CH*n* 保留。(RO)

Register 3.32. GDMA_OUT_EOF_DES_ADDR_CH*n*_REG (*n*: 0-4) (0x0088+192**n*)



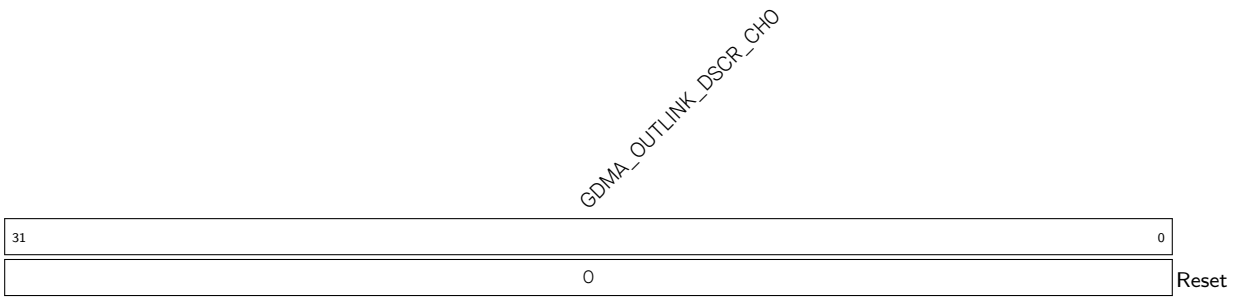
GDMA_OUT_EOF_DES_ADDR_CH*n* 发送链表描述符的 EOF 为 1 时，该描述符的地址。(RO)

Register 3.33. GDMA_OUT_EOF_BFR_DES_ADDR_CH*n*_REG (*n*: 0-4) (0x008C+192**n*)



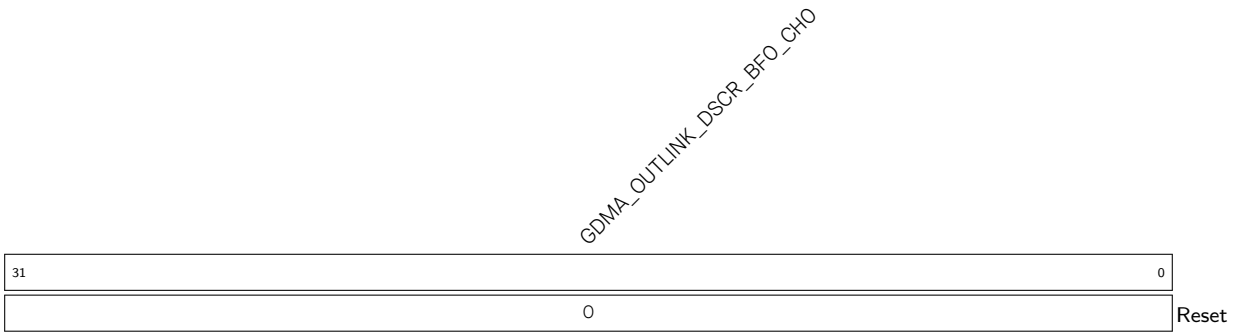
GDMA_OUT_EOF_BFR_DES_ADDR_CH*n* 倒数第二个发送链表描述符的地址。(RO)

Register 3.34. GDMA_OUT_DSCR_CH*n*_REG (*n*: 0-4) (0x0090+192**n*)



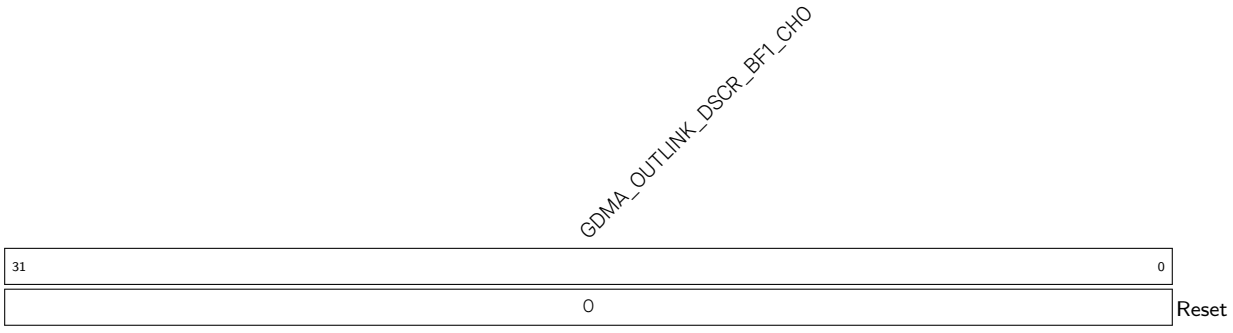
GDMA_OUTLINK_DSCR_CH*n* 表示当前已预读取的发送链表描述符指向的下一个发送链表描述符地址 y+1。(RO)

Register 3.35. GDMA_OUT_DSCR_BF0_CH*n*_REG (*n*: 0-4) (0x0094+192**n*)



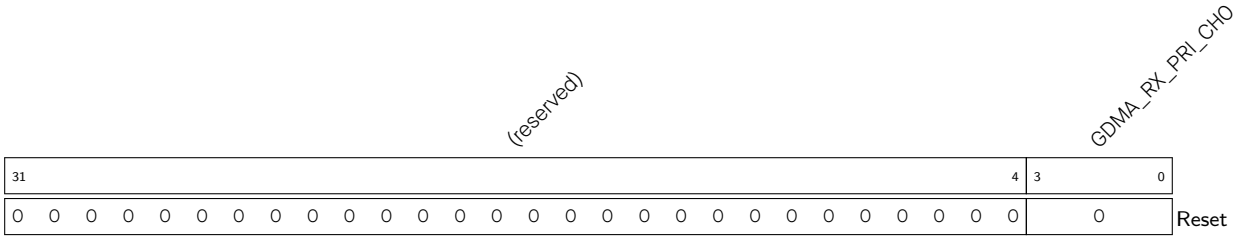
GDMA_OUTLINK_DSCR_BF0_CH*n* 表示当前已预读取的发送链表描述符所在地址 y。(RO)

Register 3.36. GDMA_OUT_DSCR_BF1_CH*n*_REG (*n*: 0-4) (0x0098+192**n*)



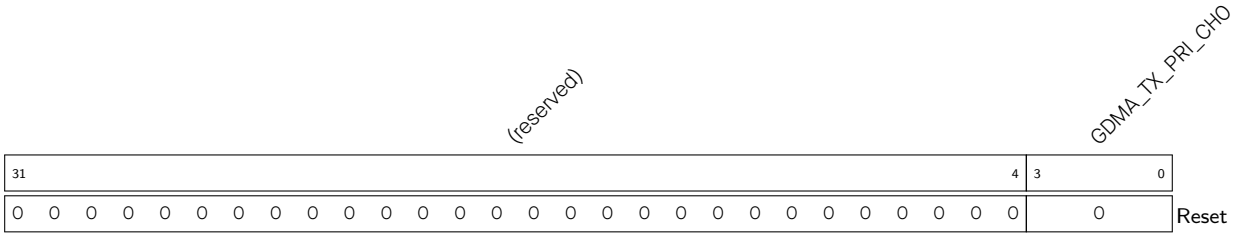
GDMA_OUTLINK_DSCR_BF1_CH*n* 表示前一个已预读取的发送链表描述符所在地址 y-1。(RO)

Register 3.37. GDMA_IN_PRI_CH*n*_REG (*n*: 0-4) (0x0044+192**n*)



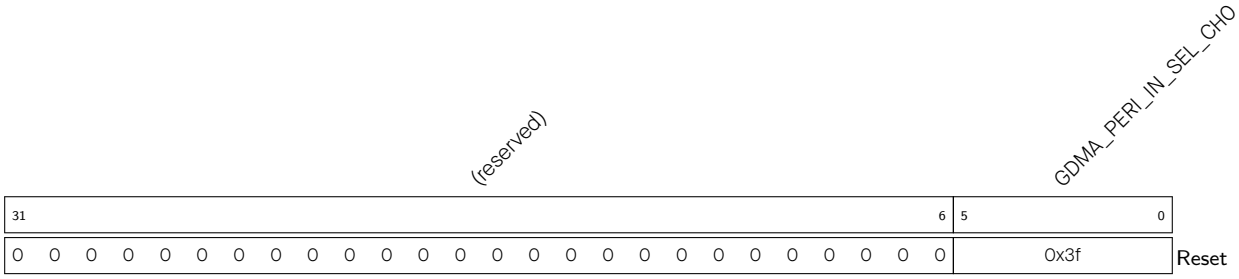
GDMA_RX_PRI_CH*n* 接收通道 0 的优先级。该值越大，优先级越高。(R/W)

Register 3.38. GDMA_OUT_PRI_CH*n*_REG (*n*: 0-4) (0x00A4+192**n*)



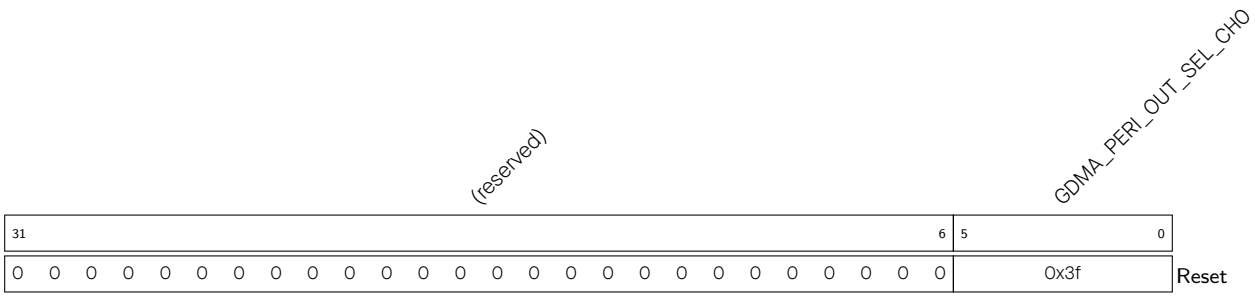
GDMA_TX_PRI_CH*n* 发送通道 0 的优先级。该值越大，优先级越高。(R/W)

Register 3.39. GDMA_IN_PERI_SEL_CH*n*_REG (*n*: 0-4) (0x0048+192**n*)



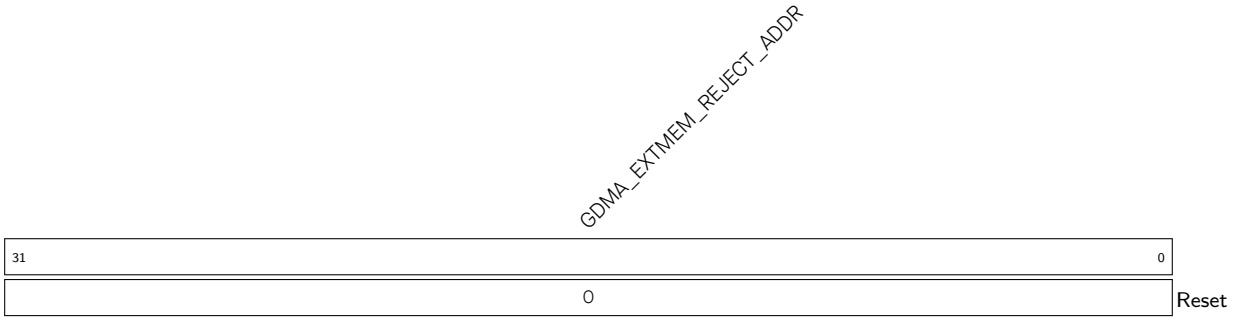
GDMA_PERI_IN_SEL_CH*n* 用于选择接收通道 0 连接的外设。0：SPI2；1：SPI3；2：UHCI0；3：I2S0；4：I2S1；5：LCD_CAM；6：AES；7：SHA；8：ADC_DAC；9：RMT；10 ~ 63：无效值。(R/W)

Register 3.40. GDMA_OUT_PERI_SEL_CH*n*_REG (*n*: 0-4) (0x00A8+192**n*)



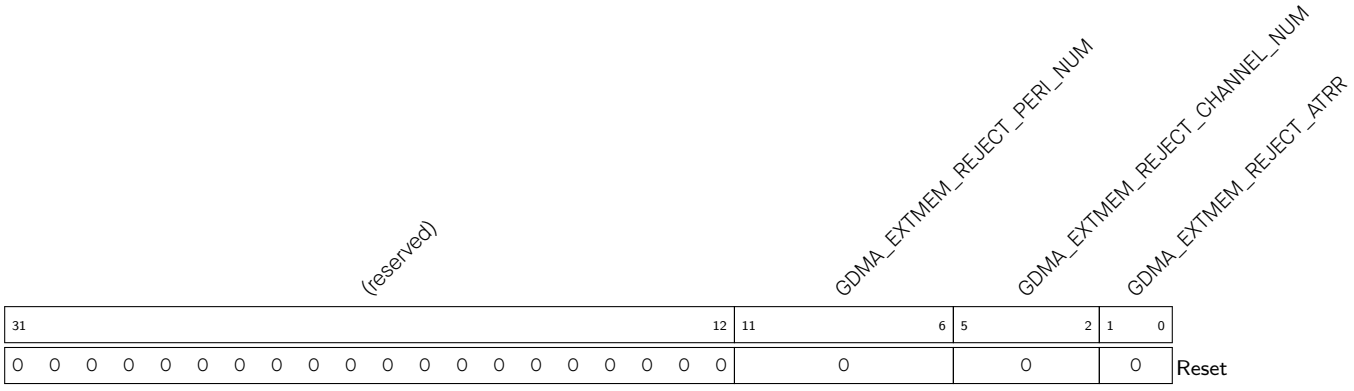
GDMA_PERI_OUT_SEL_CH*n* 用于选择发送通道 0 连接的外设。0：SPI2；1：SPI3；2：UHCI0；3：I2S0；4：I2S1；5：LCD_CAM；6：AES；7：SHA；8：ADC_DAC；9：RMT；10 ~ 63：无效值。(R/W)

Register 3.41. GDMA_EXTMEM_REJECT_ADDR_REG (0x03F4)



GDMA_EXTMEM_REJECT_ADDR 存储非法访问外部 RAM 的第一个字节。(RO)

Register 3.42. GDMA_EXTMEM_REJECT_ST_REG (0x03F8)

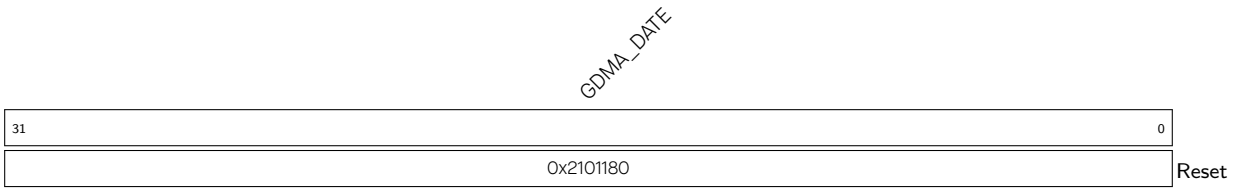


GDMA_EXTMEM_REJECT_ATRR 非法访问的读写属性。0 位是 1 是读，1 位为 1 是写。(RO)

GDMA_EXTMEM_REJECT_CHANNEL_NUM 非法访问使用的通道。(RO)

GDMA_EXTMEM_REJECT_PERI_NUM 非法访问请求来自于哪个外设。(RO)

Register 3.43. GDMA_DATE_REG (0x040C)



GDMA_DATE 版本控制寄存器。(R/W)

第 II 卷

存储器组织结构

该部分详细描述系统的存储器组织结构，例如 RAM、ROM、eFuse 和外部存储器的组织和映射，为理解与存储器相关的子系统提供了框架。

第 4 章

系统和存储器

4.1 概述

ESP32-S3 采用哈佛结构 Xtensa® LX7 CPU 构成双核系统。所有的内部存储器、外部存储器以及外设都分布在 CPU 的总线上。

4.2 主要特性

- **地址空间**
  - 848 KB 内部存储器指令地址空间
  - 560 KB 内部存储器数据地址空间
  - 836 KB 外设地址空间
  - 32 MB 外部存储器指令虚地址空间
  - 32 MB 外部存储器数据虚地址空间
  - 480 KB 内部 DMA 地址空间
  - 32 MB 外部 DMA 地址空间
- **内部存储器**
  - 384 KB 内部 ROM
  - 512 KB 内部 SRAM
  - 8 KB RTC 快速存储器
  - 8 KB RTC 慢速存储器
- **外部存储器**
  - 最大支持 1 GB 片外 flash
  - 最大支持 1 GB 片外 RAM
- **外设空间**
  - 总计 45 个模块/外设
- **GDMA**
  - 10 个具有 GDMA 功能的模块/外设

图 4.2-1 描述了系统结构与地址映射结构。

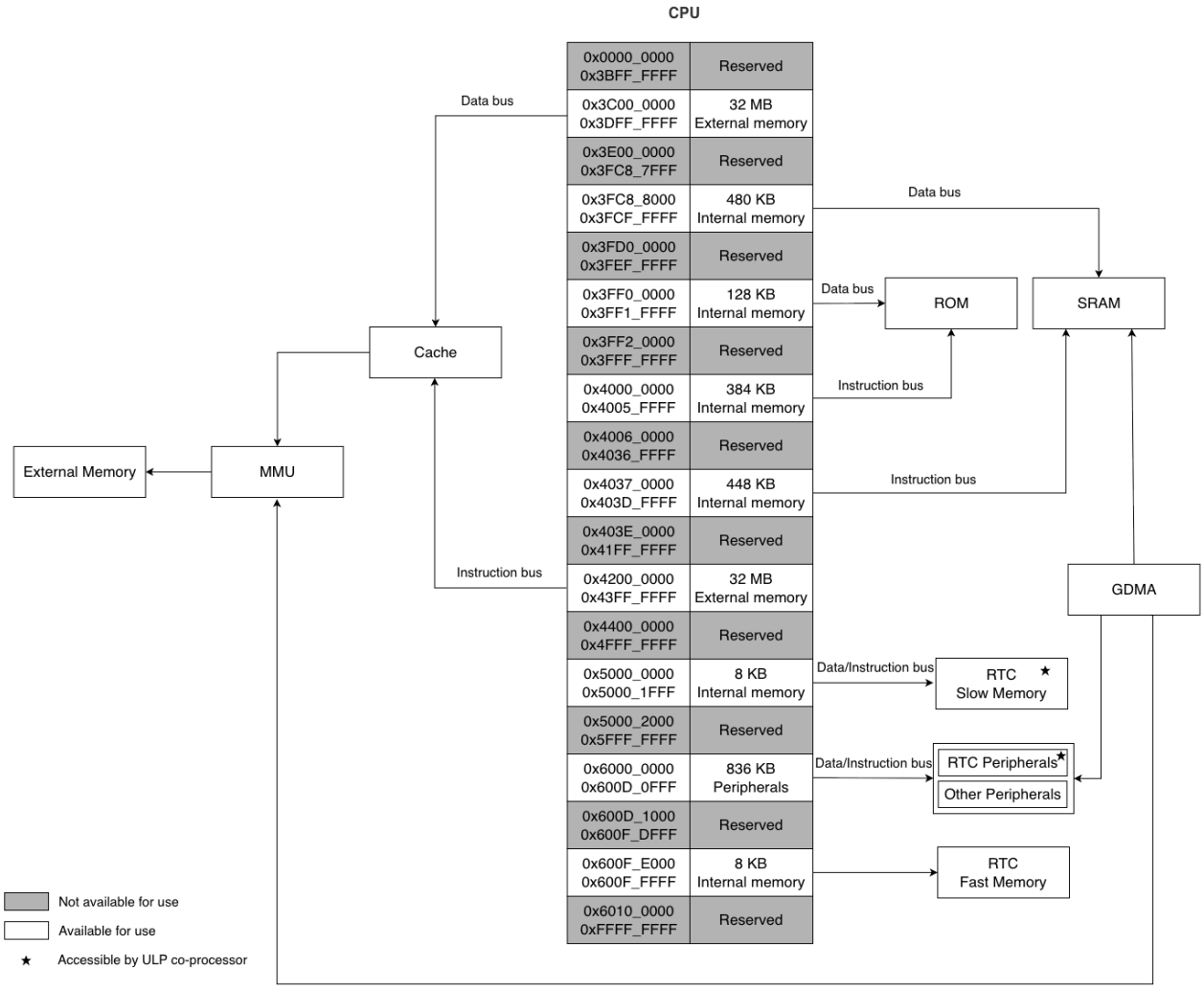


图 4.2-1. 系统结构与地址映射结构

说明：

- 图中灰色背景标注的地址空间不可用。
- 图中五角星表示对应存储器或外设可以被协处理器访问。
- 地址空间中可用的地址范围可能大于实际可用的内存。

4.3 功能描述

4.3.1 地址映射

系统由两个哈佛结构 Xtensa® LX7 CPU 构成，这两个 CPU 能够访问的地址空间范围完全一致。

地址 0x4000_0000 以下的部分属于数据总线的地址范围，地址 0x4000_0000 ~ 0x4FFF_FFFF 部分为指令总线的地址范围，地址 0x5000_0000 及以上的部分是数据总线与指令总线共用的地址范围。

CPU 的数据总线与指令总线都为小端序。CPU 可以通过数据总线进行单字节、双字节、4 字节、16 字节的数据访问。CPU 也可以通过指令总线进行数据访问，但必须是 4 字节对齐方式；非对齐数据访问会导致 CPU 工作异常。

CPU 能够：

- 通过数据总线与指令总线直接访问内部存储器；
- 通过 Cache 直接访问映射到地址空间的外部存储器；
- 通过数据总线直接访问模块/外设。

图 4.2-1 描述了 CPU 的数据总线与指令总线中的各段地址所能访问的目标。

系统中部分内部存储器与部分外部存储器既可以被数据总线访问也可以被指令总线访问，这种情况下，CPU 可以通过多个地址访问到同一目标。

4.3.2 内部存储器

ESP32-S3 的内部存储器包含如下三种类型：

- Internal ROM (384 KB)：Internal ROM 是只读存储器，不可编程。Internal ROM 中存放有一些系统底层软件的 ROM 代码（程序指令和一些只读数据）。
- Internal SRAM (512 KB)：内部静态存储器 (SRAM) 是易失性 (volatile) 存储器，可以快速响应 CPU 的访问请求（通常一个 CPU 时钟周期）。
  - SRAM 中的一部分可以被配置成外部存储器访问的缓存 (Cache)，在这种情况下无法被 CPU 访问。
  - SRAM 中的某些部分只可以被 CPU 的指令总线访问。
  - SRAM 中的某些部分只可以被 CPU 的数据总线访问。
  - SRAM 中的某些部分既可以被 CPU 的指令总线访问，又可以被 CPU 的数据总线访问。
- RTC Memory (16 KB)：RTC 存储器以静态 RAM (SRAM) 方式实现，因此也是易失性存储器。但是，在 deep sleep 模式下，存放在 RTC 存储器中的数据不会丢失。
  - RTC FAST Memory (8 KB)：RTC FAST memory 只可以被 CPU 访问，不可以被协处理器访问，通常用来存放一些在 Deep Sleep 模式下仍需保持的程序指令和数据。
  - RTC SLOW Memory (8 KB)：RTC SLOW memory 既可以被 CPU 访问，又可以被协处理器访问，因此通常用来存放一些 CPU 和协处理器需要共享的程序指令和数据。

基于上述对几种类型的内部存储器的描述，ESP32-S3 的内部存储器可以被分为四个部分：Internal ROM (384 KB)、Internal SRAM (512 KB)、RTC FAST Memory (8 KB)、RTC SLOW Memory (8 KB)。CPU 通过不同的总线访问这几部分内部存储器时会有些许限制（如某些部分只允许 CPU 通过指令总线访问），据此内部存储器可以被区分的更加细致。表 4.3-1 列出了所有内部存储器以及可以访问内部存储器的数据总线与指令总线地址段。

表 4.3-1. 内部存储器地址映射

总线类型	边界地址		容量 (KB)	目标
	低位地址	高位地址		
数据	0x3FF0_0000	0x3FF1_FFFF	128	Internal ROM 1
	0x3FC8_8000	0x3FCE_FFFF	416	Internal SRAM 1

见下页

表 4.3-1 – 接上页

总线类型	边界地址		容量 (KB)	目标
	低位地址	高位地址		
	0x3FCF_0000	0x3FCF_FFFF	64	Internal SRAM 2
指令	0x4000_0000	0x4003_FFFF	256	Internal ROM 0
	0x4004_0000	0x4005_FFFF	128	Internal ROM 1
	0x4037_0000	0x4037_7FFF	32	Internal SRAM 0
	0x4037_8000	0x403D_FFFF	416	Internal SRAM 1
数据/指令	0x5000_0000	0x5000_1FFF	8	RTC SLOW Memory
	0x600F_E000	0x600F_FFFF	8	RTC FAST Memory

说明：

所有的内部存储器都接受权限管理。只有获取到访问内部存储器的访问权限，才可以执行正常的访问操作，CPU 访问内部存储器时才可以被响应。关于权限管理的更多信息，请参考章节 15 权限控制 *(PMS)*。

1. Internal ROM 0

Internal ROM 0 的容量为 256 KB，只读。如表 4.3-1 所示，CPU 只可以通过指令总线访问这部分存储器。

2. Internal ROM 1

Internal ROM 1 的容量为 128 KB，只读。如表 4.3-1 所示，CPU 可以通过指令总线地址段 0x4004_0000 ~ 0x4005_FFFF 或数据总线地址段 0x3FF0_0000 ~ 0x3FF1_FFFF 同序访问这部分存储器。

这两段地址同序访问 Internal ROM 1 是指：地址 0x4004_0000 与 0x3FF0_0000 访问到相同的字，0x4004_0004 与 0x3FF0_0004 访问到相同的字，0x4004_0008 与 0x3FF0_0008 访问到相同的字，以此类推（下同）。

3. Internal SRAM 0

Internal SRAM 0 的容量为 32 KB，可读可写。如表 4.3-1 所示，CPU 只可以通过指令总线访问这部分存储器。

通过配置，这部分存储器中的 16 KB、或全部 32 KB 可以被 instruction Cache (ICache) 占用，用来缓存外部存储器的指令或只读数据。被 ICache 占用的部分不可以被 CPU 访问，未被 ICache 占用的部分仍然可以被 CPU 访问。

4. Internal SRAM 1

Internal SRAM 1 容量为 416 KB，可读可写。如表 4.3-1 所示，CPU 可以通过数据或指令总线同序访问。

Internal SRAM 1 存储器的 416 KB 地址空间由多个容量为 8 KB 和 16 KB 的小存储器（子存储器）组成。可以从中选取至多 16 KB 的存储空间作为 Trace Memory 用于 CPU 的 Trace 功能，并且 Trace Memory 仍可被 CPU 访问。

5. Internal SRAM 2

Internal SRAM 2 的容量为 64 KB，可读可写。如表 4.3-1 所示，CPU 只可以通过数据总线访问这部分存储器。

通过配置，这部分存储器中的 32 KB、或全部 64 KB 可以被 data Cache (DCache) 占用，用来缓存外部存储器的数据。被 DCache 占用的部分不可以被 CPU 访问，未被 DCache 占用的部分仍然可以被 CPU 访问。

6. RTC FAST Memory

RTC FAST Memory 容量为 8 KB，可读可写。如表 4.3-1 所示，CPU 可以通过数据/指令总线的共用地址段 0x600F_E000 ~ 0x600F_FFFF 访问这部分存储器。

7. RTC SLOW Memory

RTC SLOW Memory 容量为 8 KB，可读可写。如表 4.3-1 所示，CPU 可以通过数据/指令总线的共用地址段 0x5000_E000 ~ 0x5001_FFFF 访问这部分存储器。

RTC SLOW Memory 也可以被当作一个外设来使用，此时 CPU 可以通过地址段 0x6002_1000 ~ 0x6002_2FFF 来访问它。

4.3.3 外部存储器

ESP32-S3 支持以 SPI、Dual SPI、Quad SPI、Octal SPI、QPI、OPI 等接口形式连接 flash 和片外 RAM。ESP32-S3 还支持基于 XTS-AES 算法的硬件加解密功能，从而保护开发者片外 flash 和片外 RAM 中的程序和数据。

4.3.3.1 外部存储器地址映射

CPU 借助高速缓存 (Cache) 来访问外部存储器。Cache 将根据内存管理单元 (MMU) 中的信息把 CPU 指令总线或数据总线的地址变换为访问片外 flash 与片外 RAM 的实地址。经过变换的实地址所组成的实地址空间最大支持 1 GB 的片外 flash 与 1 GB 的片外 RAM。

通过高速缓存，ESP32-S3 一次最多可以同时有：

- 32 MB 的指令总线地址空间，通过指令缓存 (ICache) 以 64 KB 为单位映射到片外 flash 或片外 RAM，支持 4 字节对齐的读访问或取指访问。
- 32 MB 的数据总线地址空间，通过数据缓存 (DCache) 以 64 KB 为单位映射到片外 RAM，支持单字节、双字节、4 字节、16 字节的读写访问。这部分地址空间也可以用作只读数据空间，映射到片外 flash 或片外 RAM。

表 4.3-2 列出了在访问外部存储器时 CPU 的数据总线与指令总线与 Cache 的对应关系。

表 4.3-2. 外部存储器地址映射

总线类型	边界地址		容量 (MB)	目标
	低位地址	高位地址		
数据	0x3C00_0000	0x3DFF_FFFF	32	DCache
指令	0x4200_0000	0x43FF_FFFF	32	ICache

说明：

只有获取到外部存储器的访问权限，CPU 访问外部存储器时才可以被响应。关于权限管理的更多信息，请参考章节 15 权限控制 *(PMS)*。

4.3.3.2 高速缓存

如图 4.3-1 所示，ESP32-S3 采用双核共享 ICache 和 DCache 结构，以便当 CPU 的指令总线和数据总线同时发起请求时，也可以迅速响应。Cache 的存储空间与内部存储空间可以复用（详见章节 4.3.2 中内部 SRAM 0 与

内部 SRAM 2）。

当两个核的指令总线同时访问 ICache 时，由仲裁器决定谁先获得访问 ICache 的权限；当两个核的数据总线同时访问 DCache 时，由仲裁器决定谁先获得访问 DCache 的权限。当 Cache 缺失时，Cache 控制器会向外部存储器发起请求，当 ICache 和 DCache 同时发起外部存储器请求时，由仲裁器决定谁先获得外部存储器的使用权。ICache 的缓存大小可配置为 16 KB 或 32 KB，块大小可以配置为 16 B 或 32 B，当 ICache 缓存大小配置为 32 KB 时，禁用 16 B 块大小模式。DCache 的缓存大小可配置为 32 KB 或 64 KB，块大小可以配置为 16 B、32 B 或 64 B，当 DCache 缓存大小配置为 64 KB 时，禁用 16 B 块大小模式。

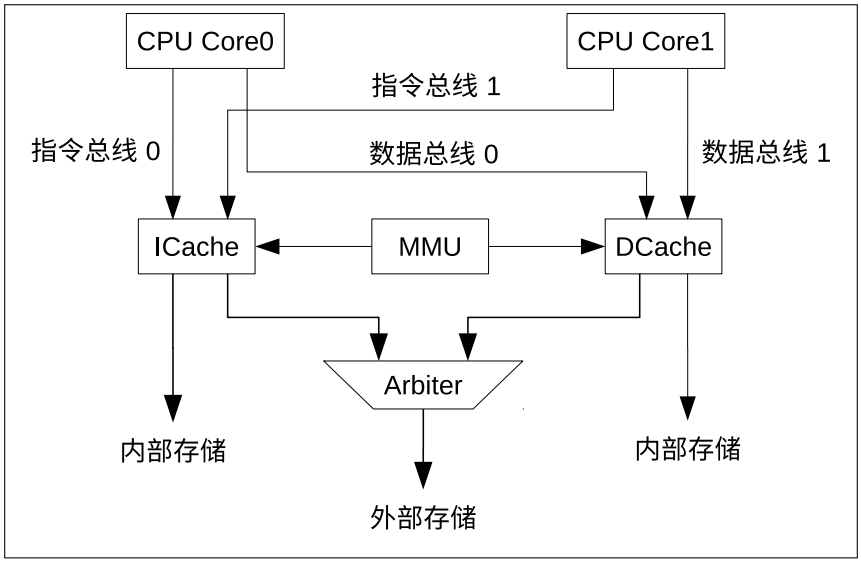


图 4.3-1. Cache 系统框图

4.3.3.3 Cache 操作

ESP32-S3 Cache 共有如下几种操作：

1. **Write-Back**：Write-Back 操作用于将 Cache 中“脏块”的“脏”标记 (Dirty bit) 抹除，并将数据同步到外部存储器。Write-Back 操作结束后，外部存储器和 Cache 中存放的都是新数据。如果 CPU 接着去访问该数据，那么可以直接从 Cache 中访问到。只有 DCache 具有此功能。

   所谓“脏块”是指，如果 Cache 中的数据比外部存储器中的数据要新，则 Cache 中的新数据被称为“脏块”，Cache 通过“脏”标记来追踪这些“脏块”数据。如果抹除掉某个新数据的“脏”标记，Cache 将认为这个数据不是新的。

2. **Clean**：Clean 操作用于将 Cache 中的“脏块”的“脏”标记 (Dirty bit) 抹除，但不将数据同步到外部存储器。Clean 操作结束后，外部存储器中仍是旧数据，Cache 中存放的是新数据，但 Cache 以为不是新的。如果 CPU 接着去访问该数据，那么直接可以从 Cache 中访问到。只有 DCache 具有此功能。

3. **Invalidate**：Invalidate 操作用于删除 Cache 中的有效数据，即使该数据是“脏块”，也不会将脏块同步到外部存储器。对于非脏块数据，Invalidate 操作结束后，该数据仅存在于外部存储器中。如果 CPU 接着去访问该数据，那么需要访问外部存储器。对于脏块数据，invalidate 结束后，外部存储器中存在的是旧数据，新数据将彻底丢失。Invalidate 分为自动失效 (Auto-Invalidate) 和手动失效 (Manual-Invalidate)。Manual-Invalidate 仅对 Cache 中落入指定区域的地址对应的数据做失效处理，而 Auto-Invalidate 会对 Cache 中的所有数据做失效处理。ICache 和 DCache 均具有此功能。

4. **预取** (Preload)：Preload 功能用于将指令和数据提前加载到 Cache 中。预取操作的最小单位为 1 个块。预取分为手动预取 (Manual-Preload) 和自动预取 (Auto-Preload)，Manual-Preload 是指硬件按软件指定的

虚地址预取一段连续的数据；Auto-Preload 是指硬件根据当前命中/缺失（取决于配置）的地址，自动地预取一段连续的数据。ICache 和 DCache 均具有此功能。

5. **锁定/解锁** (Lock/Unlock)：Lock 操作用于保护 Cache 中的数据不被替换掉。锁定分为预锁定和手动锁定。预锁定开启时，Cache 在填充缺失数据到 Cache 时，如果该数据落在指定区域，则将该数据锁定，未落入指定区域的数据不会被锁定。手动锁定开启时，Cache 检查 Cache 中的数据，并将落在指定区域的数据锁定，未落入指定区域的数据不会被锁定。当缺失发生时，Cache 会优先替换掉未被锁定的那一路 (way) 的数据，因此锁定区域的数据会一直保存在 Cache 中。但当所有路都被锁定时，Cache 将进行正常替换，就像所有路都没有被锁定一样。解锁是锁定的逆操作，但解锁只有手动解锁。ICache 和 DCache 均具有此功能。

   需要注意的是，Cache 的 Clean、Write-Back 和 Manual-Invalidate 操作均只对未被锁定的数据起作用。如果想对被锁定的数据执行这些操作，请先解锁这些数据。

4.3.4 GDMA 地址空间

ESP32-S3 中的通用直接存储访问 (General Direct Memory Access, GDMA) 外设可以提供直接存储访问 (Direct Memory Access, DMA) 服务，包括：

- 内部存储器与内部存储器之间的数据搬运；
- 内部存储器与外部存储器之间的数据搬运；
- 外部存储器与外部存储器之间的数据搬运。

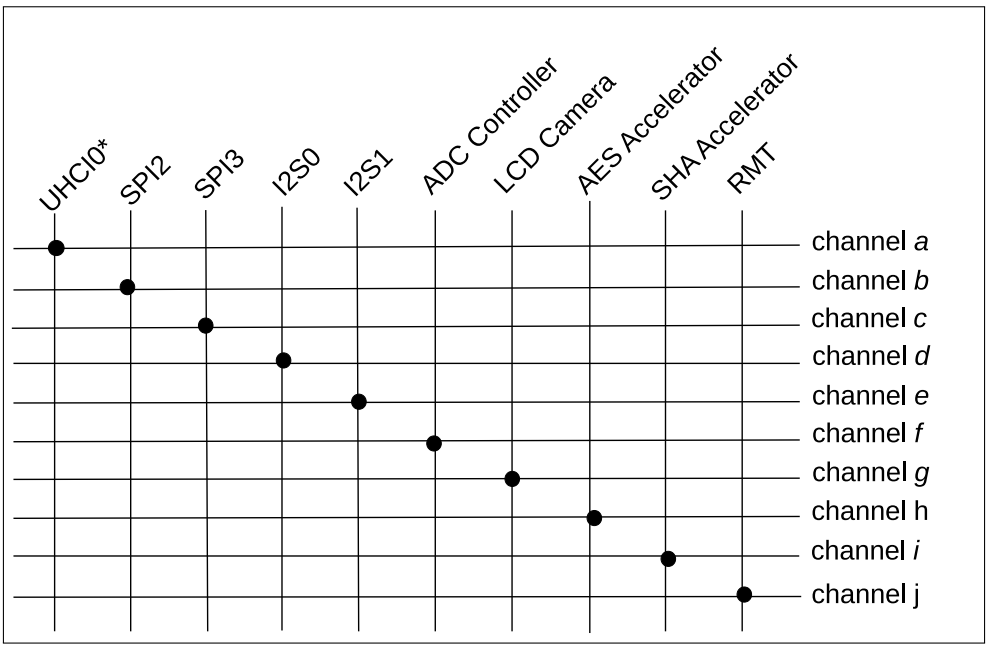
GDMA 可以通过与 CPU 数据总线完全相同的地址访问 Internal SRAM 1 与 Internal SRAM 2，即通过地址 0x3FC8_8000 ~ 0x3FCE_FFFF 访问 Internal SRAM 1，通过地址 0x3FCF_0000 ~ 0x3FCF_FFFF 访问 Internal SRAM 2。但 GDMA 无法访问被 Cache 占用的内部存储器。

GDMA 可以通过与 CPU 访问 DCache 相同的地址 (0x3C00_0000 ~ 0x3DFF_FFFF) 来访问外部存储器，但只可以访问片外 RAM。当 GDMA 与 DCache 同时访问外部存储器时，数据一致性问题需要由软件来保证。

另外，ESP32-S3 中的某些外设/模块可以和 GDMA 联合工作，此时 GDMA 可以为这些外设提供如下服务：

- 模块/外设与内部存储器之间的数据搬运；
- 模块/外设与外部存储器之间的数据搬运。

ESP32-S3 中有 10 个外设/模块可以和 GDMA 联合工作，如图 4.3-2 所示。其中的 10 根竖线依次对应这 10 个具有 GDMA 功能的外设/模块，横线表示 GDMA 的某一个通道（可以是任意一个通道），竖线与横线的交点表示对应外设/模块可以访问 GDMA 的某一个通道。同一行上有多个交点则表示这几个外设/模块不可以同时开启 GDMA 功能。



*** 说明：** UART0、UART1 和 UART2 可以通过 UHCI0 实现 DMA 功能。

图 4.3-2. 具有 GDMA 功能的外设

具有 GDMA 功能的模块/外设通过 GDMA 可以访问任何 GDMA 可以访问到的存储器。更多关于 GDMA 的信息，请参考章节 3 通用 DMA 控制器 (GDMA)。

说明：

当使用 GDMA 访问任何存储器时，都需要获取对应的访问权限，否则访问将会失败。关于权限管理的更多信息，请参考章节 15 权限控制 (PMS)。

4.3.5 模块/外设地址空间

CPU 可以通过数据/指令总线的共用地址段 0x6000_0000 ~ 0x600D_0FFF 来访问模块/外设。

4.3.5.1 模块/外设地址空间列表

表 4.3-3 详细列出了模块/外设地址空间的各段地址与其能访问到的模块/外设的映射关系。其中，“边界地址”栏中的两列数值共同决定了对应模块/外设的地址空间。

表 4.3-3. 模块/外设地址空间映射表

目标	边界地址		容量 (KB)	说明
	低位地址	高位地址		
UART 控制器 0	0x6000_0000	0x6000_0FFF	4	
保留	0x6000_1000	0x6000_1FFF		
SPI 控制器 1	0x6000_2000	0x6000_2FFF	4	
SPI 控制器 0	0x6000_3000	0x6000_3FFF	4	
GPIO	0x6000_4000	0x6000_4FFF	4	
保留	0x6000_5000	0x6000_6FFF		

见下页

表 4.3-3 – 接上页

目标	边界地址		容量 (KB)	说明
	低位地址	高位地址		
eFuse 控制器	0x6000_7000	0x6000_7FFF	4	
低功耗管理	0x6000_8000	0x6000_8FFF	4	
IO MUX	0x6000_9000	0x6000_9FFF	4	
保留	0x6000_A000	0x6000_EFFF		
I2S 控制器 0	0x6000_F000	0x6000_FFFF	4	
UART 控制器 1	0x6001_0000	0x6001_0FFF	4	
保留	0x6001_1000	0x6001_2FFF		
I2C 控制器 0	0x6001_3000	0x6001_3FFF	4	
UHCI0	0x6001_4000	0x6001_4FFF	4	
保留	0x6001_5000	0x6001_5FFF		
红外遥控	0x6001_6000	0x6001_6FFF	4	
脉冲计数控制器	0x6001_7000	0x6001_7FFF	4	
保留	0x6001_8000	0x6001_8FFF		
LED PWM 控制器	0x6001_9000	0x6001_9FFF	4	
保留	0x6001_A000	0x6001_DFFF		
电机控制器 0	0x6001_E000	0x6001_EFFF	4	
定时器组 0	0x6001_F000	0x6001_FFFF	4	
定时器组 1	0x6002_0000	0x6002_0FFF	4	
RTC SLOW Memory	0x6002_1000	0x6002_2FFF	8	
系统定时器	0x6002_3000	0x6002_3FFF	4	
SPI 控制器 2	0x6002_4000	0x6002_4FFF	4	
SPI 控制器 3	0x6002_5000	0x6002_5FFF	4	
SYSCON	0x6002_6000	0x6002_6FFF	4	
I2C 控制器 1	0x6002_7000	0x6002_7FFF	4	
SD/MMC 主机控制器	0x6002_8000	0x6002_8FFF	4	
保留	0x6002_9000	0x6002_AFFF		
双线汽车接口	0x6002_B000	0x6002_BFFF	4	
电机控制器 1	0x6002_C000	0x6002_CFFF	4	
I2S 控制器 1	0x6002_D000	0x6002_DFFF	4	
UART 控制器 2	0x6002_E000	0x6002_EFFF	4	
保留	0x6002_F000	0x6003_7FFF		
USB Serial/JTAG 控制器	0x6003_8000	0x6003_8FFF	4	
USB 外部控制寄存器	0x6003_9000	0x6003_9FFF	4	1
AES 加速器	0x6003_A000	0x6003_AFFF	4	
SHA 加速器	0x6003_B000	0x6003_BFFF	4	
RSA 加速器	0x6003_C000	0x6003_CFFF	4	
数字签名	0x6003_D000	0x6003_DFFF	4	
HMAC 加速器	0x6003_E000	0x6003_EFFF	4	
通用 DMA 控制器	0x6003_F000	0x6003_FFFF	4	
ADC 控制器	0x6004_0000	0x6004_0FFF	4	
Camera 与 LCD 控制器	0x6004_1000	0x6004_1FFF	4	

见下页

表 4.3-3 – 接上页

目标	边界地址		容量 (KB)	说明
	低位地址	高位地址		
保留	0x6004_2000	0x6007_FFFF		
USB 内核寄存器	0x6008_0000	0x600B_FFFF	256	1
系统寄存器	0x600C_0000	0x600C_0FFF	4	
PMS 寄存器	0x600C_1000	0x600C_1FFF	4	
中断矩阵	0x600C_2000	0x600C_2FFF	4	
保留	0x600C_3000	0x600C_3FFF		
保留	0x600C_4000	0x600C_BFFF		
片外存储器加密与解密	0x600C_C000	0x600C_CFFF	4	
保留	0x600C_D000	0x600C_DFFF		
保留	0x600C_E000	0x600C_EFFF		
保留	0x600C_F000	0x600C_FFFF		
World 控制器	0x600D_0000	0x600D_0FFF	4	

说明：

1. 该模块/外设的地址空间不连续。
2. CPU 要想访问某一个模块/外设，需要先获取该模块/外设的访问权限，否则访问将不会被响应。关于权限管理的更多信息，请参考章节 15 权限控制 *(PMS)*。

第 5 章

eFuse 控制器 (EFUSE)

5.1 概述

ESP32-S3 系统中有一块 4-Kbit 的 eFuse，其中存储着参数内容。eFuse 的各个位一旦被烧写为 1，则不能再恢复为 0。eFuse 控制器按照用户配置完成对 eFuse 中各参数中的各个位的烧写。从芯片外部，eFuse 数据只能通过 eFuse 控制器读取。对于某些数据，如果未启用读保护，则可以从芯片外部读取该数据；如果启用了读保护，则无法从芯片外部读取该数据。不过，存储在 eFuse 中的某些密钥始终可以供硬件加密模块（例如数字签名、HMAC 等）在内部使用，芯片外部无法获得这些数据。

5.2 主要特性

- 总存储空间为 4-Kbit，其中 1792 位可供用户使用
- 一次性可编程存储
- 烧写保护可配置
- 读取保护可配置
- 使用多种硬件编码方式保护参数内容

5.3 功能描述

5.3.1 结构

eFuse 从结构上分成 11 个块 (BLOCK0 ~ BLOCK10)。BLOCK0 为 640 位，BLOCK1 为 288 位，其余每个块为 352 位。

BLOCK0 存储大部分参数，其中 25 位供硬件使用，用户不可见（详细信息可参见第 5.3.2 节）；还有 29 位处于保留状态，留作未来使用。

表 5.3-1 列出了 BLOCK0 中的参数名称、偏移地址、位宽、是否可供硬件使用、烧写保护，以及描述信息。

在这些参数中，**EFUSE_WR_DIS** 用于控制其他参数的烧写，**EFUSE_RD_DIS** 用于控制用户读取 BLOCK4 ~ BLOCK10。更多关于这两个参数的信息请见章节 5.3.1.1、5.3.1.2。

表 5.3-1. BLOCK0 参数

参数	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	描述
EFUSE_WR_DIS	32	Y	N/A	表示是否禁止 eFuse 烧写
EFUSE_RD_DIS	7	Y	0	表示是否禁止用户读取 eFuse BLOCK4 ~ 10 的内容
EFUSE_DIS_ICACHE	1	Y	2	表示是否禁用 ICache
EFUSE_DIS_DCACHE	1	Y	2	表示是否禁用 DCache
EFUSE_DIS_DOWNLOAD_ICACHE	1	Y	2	表示是否在下载模式下关闭 ICache
EFUSE_DIS_DOWNLOAD_DCACHE	1	Y	2	表示是否在下载模式下关闭 DCache
EFUSE_DIS_FORCE_DOWNLOAD	1	Y	2	表示是否禁止强制芯片进入下载模式
EFUSE_DIS_USB_OTG	1	Y	2	表示是否禁用 USB OTG 功能
EFUSE_DIS_TWAI	1	Y	2	表示是否禁用 TWAI 控制器功能
EFUSE_DIS_APP_CPU	1	Y	2	表示是否禁用 app CPU
EFUSE_SOFT_DIS_JTAG	3	Y	31	表示是否软关断 JTAG 功能
EFUSE_DIS_PAD_JTAG	1	Y	2	表示是否永久禁用 JTAG 功能
EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT	1	Y	2	表示是否在 download boot 模式下禁用 flash 加密功能
EFUSE_USB_EXCHG_PINS	1	Y	30	表示是否交换 USB D+/D- 管脚
EFUSE_EXT_PHY_ENABLE	1	N	30	表示是否使能外部 USB PHY
EFUSE_VDD_SPI_XPD	1	Y	3	表示 flash 电压调节器是否上电
EFUSE_VDD_SPI_TIEH	1	Y	3	表示 flash 电压调节器是否短接至 VDD_RTC_IO
EFUSE_VDD_SPI_FORCE	1	Y	3	表示是否强制使用 EFUSE_VDD_SPI_XPD 和 EFUSE_VDD_SPI_TIEH 配置 flash 电压 LDO
EFUSE_WDT_DELAY_SEL	2	Y	3	表示 RTC 看门狗超时阈值
EFUSE_SPI_BOOT_CRYPT_CNT	3	Y	4	表示是否使能 SPI boot 加解密
EFUSE_SECURE_BOOT_KEY_REVOKE0	1	N	5	表示是否使能撤销第一个安全启动密钥
EFUSE_SECURE_BOOT_KEY_REVOKE1	1	N	6	表示是否使能撤销第二个安全启动密钥
EFUSE_SECURE_BOOT_KEY_REVOKE2	1	N	7	表示是否使能撤销第三个安全启动密钥
EFUSE_KEY_PURPOSE_0	4	Y	8	表示 Key0 用途，见表 5.3-2
EFUSE_KEY_PURPOSE_1	4	Y	9	表示 Key1 用途，见表 5.3-2

见下页

GoBack

表 5.3-1 – 接上页

参数	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	描述
EFUSE_KEY_PURPOSE_2	4	Y	10	表示 Key2 用途，见表 5.3-2
EFUSE_KEY_PURPOSE_3	4	Y	11	表示 Key3 用途，见表 5.3-2
EFUSE_KEY_PURPOSE_4	4	Y	12	表示 Key4 用途，见表 5.3-2
EFUSE_KEY_PURPOSE_5	4	Y	13	表示 Key5 用途，见表 5.3-2
EFUSE_SECURE_BOOT_EN	1	N	15	表示是否使能安全启动
EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE	1	N	16	表示是否使 Secure Boot 的撤销采用激进策略
EFUSE_DIS_USB_JTAG	1	Y	2	表示是否禁用 usb_serial_jtag 模块的 usb 转 jtag 功能
EFUSE_DIS_USB_SERIAL_JTAG	1	Y	2	表示是否禁用 usb_serial_jtag 模块
EFUSE_STRAP_JTAG_SEL	1	Y	2	表示是否使能使用 strapping GPIO3 选择 usb_to_jtag 或 pad_to_jtag 的功能
EFUSE_USB_PHY_SEL	1	Y	2	表示内部 PHY、外部 PHY 和 USB OTG、USB Serial/JTAG 之间的连接关系
EFUSE_FLASH_TPUW	4	N	18	表示配置上电后 flash 等待时间
EFUSE_DIS_DOWNLOAD_MODE	1	N	18	表示是否关闭所有 Download boot 模式
EFUSE_DIS_LEGACY_SPI_BOOT	1	N	18	表示是否关闭 Legacy SPI boot 模式
EFUSE_DIS_USB_PRINT	1	N	18	表示是否关闭 USB 打印
EFUSE_FLASH_ECC_MODE	1	N	18	表示 ROM 中的 flash ECC 模式
EFUSE_DIS_USB_SERIAL_JTAG_DOWNLOAD_MODE	1	N	18	表示是否禁用 USB-Serial-JTAG 下载功能
EFUSE_ENABLE_SECURITY_DOWNLOAD	1	N	18	表示是否使能 UART 安全下载模式
EFUSE_UART_PRINT_CONTROL	2	N	18	表示 UART boot 信息的默认打印方式
EFUSE_PIN_POWER_SELECTION	1	N	18	表示 GPIO33 ~ GPIO37、GPIO47、GPIO48 的供电来源
EFUSE_FLASH_TYPE	1	N	18	表示 SPI flash 的最大行数
EFUSE_FLASH_PAGE_SIZE	2	N	18	表示 flash 的页大小
EFUSE_FLASH_ECC_EN	1	N	18	表示是否在 flash boot 模式下使能 ECC 功能
EFUSE_FORCE_SEND_RESUME	1	N	18	表示是否强制 ROM 代码在 SPI 启动过程中向 SPI flash 发送恢复命令

见下页

GoBack

GoBack

表 5.3-1 – 接上页

参数	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	描述
EFUSE_SECURE_VERSION	16	N	18	表示 IDF 安全版本
EFUSE_DIS_USB_OTG_DOWNLOAD_MODE	1	N	19	表示是否禁用 USB-OTG 下载功能

表 5.3-2 为密钥用途各个数值对应的含义。通过配置参数 EFUSE_KEY_PURPOSE_*n* 来声明 KEY*n* 用途 (*n*: 0 ~ 5)。

表 5.3-2. 密钥用途数值对应的含义

密钥用途数值	含义
0	指定为用户使用
1	保留
2	指定为 XTS_AES_256_KEY_1 使用（用于 flash/SRAM 加解密）
3	指定为 XTS_AES_256_KEY_2 使用（用于 flash/SRAM 加解密）
4	指定为 XTS_AES_128_KEY 使用（用于 flash/SRAM 加解密）
5	指定为 HMAC 下行模式使用
6	指定为 HMAC 下行模式下的 JTAG 使用
7	指定为 HMAC 下行模式下的数字签名使用
8	指定为 HMAC 上行模式使用
9	指定为 SECURE_BOOT_DIGEST0 使用（secure boot 密钥摘要）
10	指定为 SECURE_BOOT_DIGEST1 使用（secure boot 密钥摘要）
11	指定为 SECURE_BOOT_DIGEST2 使用（secure boot 密钥摘要）

表 5.3-3 列出了 BLOCK1 ~ BLOCK10 中存储的参数的信息。

表 5.3-3. BLOCK1-10 参数

块	参数	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	EFUSE_RD_DIS 读取保护位	描述
BLOCK1	EFUSE_MAC	48	N	20	N/A	MAC 地址
	EFUSE_SPI_PAD_CONFIGURE	6	N	20	N/A	CLK
		6	N	20	N/A	Q (D1)
		6	N	20	N/A	D (D0)
		6	N	20	N/A	CS
		6	N	20	N/A	HD (D3)
		6	N	20	N/A	WP (D2)
		6	N	20	N/A	DQS
		6	N	20	N/A	D4
		6	N	20	N/A	D5
		6	N	20	N/A	D6
		6	N	20	N/A	D7
	EFUSE_WAFER_VERSION	3	N	20	N/A	系统数据
	EFUSE_PKG_VERSION	3	N	20	N/A	系统数据
	EFUSE_SYS_DATA_PART0	72	N	20	N/A	系统数据
BLOCK2	EFUSE_OPTIONAL_UNIQUE_ID	128	N	20	N/A	系统数据
	EFUSE_SYS_DATA_PART1	128	N	21	N/A	系统数据
BLOCK3	EFUSE_USR_DATA	256	N	22	N/A	用户数据
BLOCK4	EFUSE_KEY0_DATA	256	Y	23	0	KEY0 或用户数据
BLOCK5	EFUSE_KEY1_DATA	256	Y	24	1	KEY1 或用户数据

见下页

表 5.3-3 – 接上页

块	参数	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	EFUSE_RD_DIS 读取保护位	描述
BLOCK6	EFUSE_KEY2_DATA	256	Y	25	2	KEY2 或用户数据
BLOCK7	EFUSE_KEY3_DATA	256	Y	26	3	KEY3 或用户数据
BLOCK8	EFUSE_KEY4_DATA	256	Y	27	4	KEY4 或用户数据
BLOCK9	EFUSE_KEY5_DATA	256	Y	28	5	KEY5 或用户数据
BLOCK10	EFUSE_SYS_DATA_PART2	256	N	29	6	系统数据

其中，BLOCK4 ~ 9 分别存储 KEY0 ~ 5，表示 eFuse 中至多可以烧写 6 个 256 位的密钥。每烧写一个密钥，还需要烧写该密钥用途的数值（见表 5.3-2）。例如，用户将用于 HMAC Downstream 模式下的 JTAG 功能的密钥烧写到 KEY3（即 BLOCK7），还需要将密钥用途的数值 6 烧写到 EFUSE_KEY_PURPOSE_3。

BLOCK1 ~ BLOCK10 均采用 RS 编码方式，因此参数烧写受到一定的限制，具体请参考章节 5.3.1.3 和章节 5.3.2。

5.3.1.1 EFUSE_WR_DIS

参数 EFUSE_WR_DIS 决定了 eFuse 中所有的参数是否处于烧写保护状态。烧写完 EFUSE_WR_DIS 参数后，需要更新 eFuse 读寄存器才能生效（参考章节 5.3.3）。

表 5.3-1 以及表 5.3-3 中的“EFUSE_WR_DIS 烧写保护位”列描述了各参数的烧写保护状态具体由 EFUSE_WR_DIS 的哪个位决定。

当某个参数对应的烧写保护位为 0 时，表示此参数未处于烧写保护状态，可以烧写该参数。

当某个参数对应的烧写保护位为 1 时，表示此参数处于烧写保护状态，此参数的每一个位都无法被更改，未被烧写的位永远为 0，已经被烧写的位永远为 1。所以如果某个参数已经处于烧写保护状态了，则会一直处在该状态，无法再更改。

5.3.1.2 EFUSE_RD_DIS

所有参数中，只有 BLCOK4 ~ BLOCK10 的参数受用户读取保护状态的约束，即表 5.3-3 中“EFUSE_RD_DIS 读取保护位”列非“N/A”的参数。烧写完 EFUSE_RD_DIS 参数后，需要更新 eFuse 读寄存器才能生效（参考章节 5.3.3）。

参数 EFUSE_RD_DIS 中的某个位为 0，表示此位管理的参数未处于用户读取保护状态；某个位为 1，表示此位管理的参数处于用户读取保护状态，用户无法以任何方式读取该参数。

除 BLOCK4 ~ BLOCK10 之外，其他参数不受用户读取保护状态的约束，均可被用户读取。

BLOCK4 ~ BLOCK10 即使被配置处于读取保护状态，仍然可以通过设置 EFUSE_KEY_PURPOSE_*n* 供硬件加密模块在内部使用。

5.3.1.3 数据存储方式

eFuse 使用硬件编码机制保护数据，对用户不可见。

BLOCK0 使用 4 备份方式存储参数，即 BLOCK0 中的所有参数（除了 EFUSE_WR_DIS ）均在 eFuse 中存储了 4 份。4 备份机制对用户不可见。

BLOCK1 ~ BLOCK10 用于存储重要数据和参数，使用 RS (44, 32) 编码方式，最多支持自动校正 6 个字节。本文 RS (44, 32) 使用的本源多项式为 $p(x) = x^8 + x^4 + x^3 + x^2 + 1$。

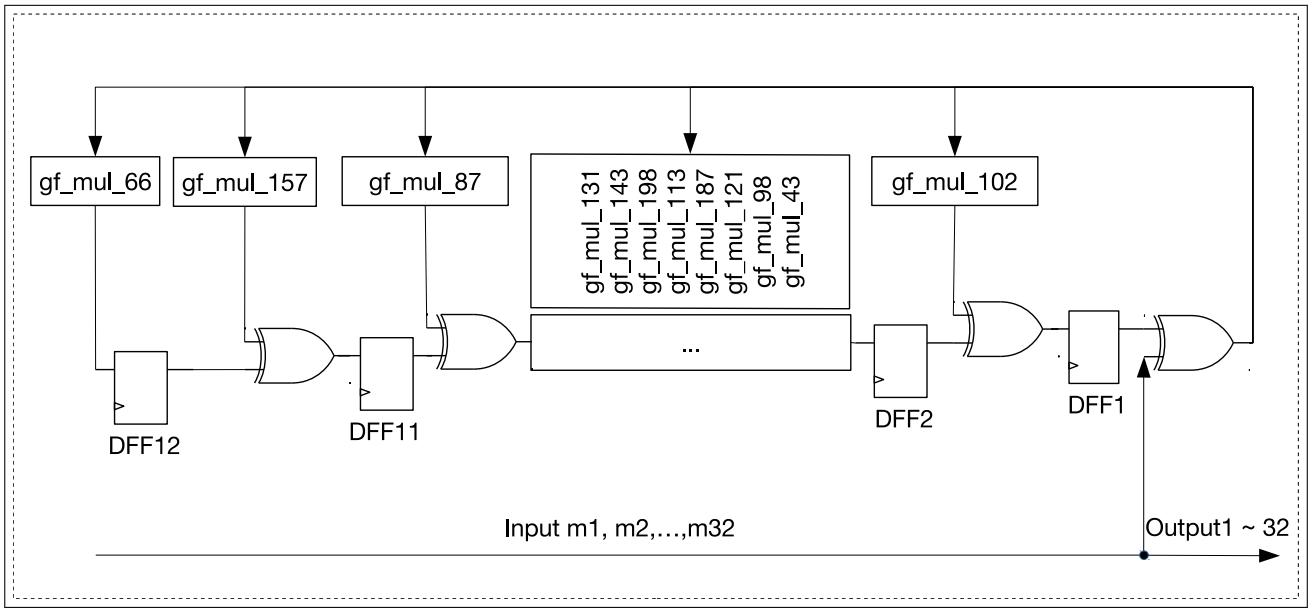


图 5.3-1. 移位寄存器电路图（前 32 字节）

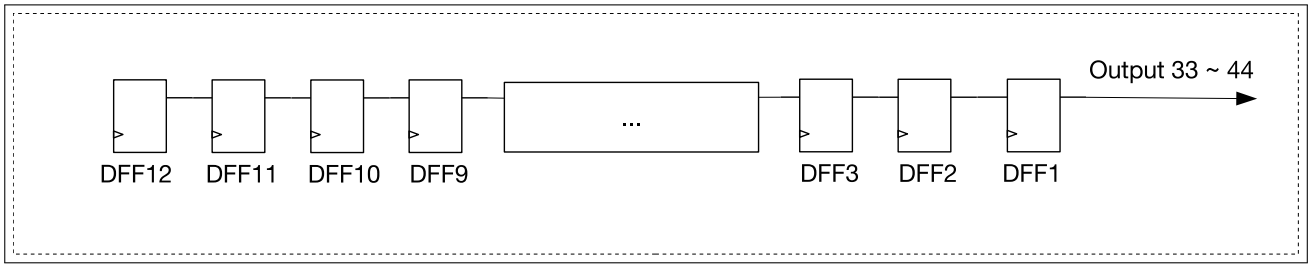


图 5.3-2. 移位寄存器电路图（后 12 字节）

如图 5.3-1 和 5.3-2 所示，软件对 32 字节参数进行 RS (44, 32) 编码处理，将 32 字节数据处理为 44 字节，其中：

- 字节 [0:31] 为数据本身
- 字节 [32:43] 为储存在 8 位触发器 DFF1, DFF2, ..., DFF12 中的奇偶校验字节（gf_mul_n 为 $GF(2^8)$ 域中某一字节数据与元素 α^n 相乘的结果，n 为整数）

然后，硬件将这 44 字节数据一起烧入 eFuse。eFuse 控制器会在读 eFuse 的过程中自动完成解码和自动校正。

由于 RS 校验码是在整个 256 位的 eFuse block 上生成的，因此每个 block 只能写入一次。

5.3.2 烧写参数

烧写 eFuse 参数时，需要按块烧写。BLOCK0 ~ BLOCK10 共用同一段地址来存储即将烧写的参数。通过配置 EFUSE_BLK_NUM 参数表明当前需要烧写的是哪一个块。

用户烧写参数之前，请确保 eFuse 烧写电压 VDDQ 的配置正确，具体请参考章节 5.3.4。

烧写 BLOCK0

当 EFUSE_BLK_NUM = 0 时，烧写 BLOCK0。EFUSE_PGM_DATA0_REG 寄存器存储着 EFUSE_WR_DIS。EFUSE_PGM_DATA1_REG ~ EFUSE_PGM_DATA5_REG 用来存储即将烧写的参数的有效信息，其中 25 位为用户可读但对用户没有意义的有效信息，必须写入 0，对应位置为：

- EFUSE_PGM_DATA1_REG[27:31]
- EFUSE_PGM_DATA1_REG[21:24]
- EFUSE_PGM_DATA2_REG[7:15]
- EFUSE_PGM_DATA2_REG[0:3]
- EFUSE_PGM_DATA3_REG[26:27]
- EFUSE_PGM_DATA4_REG[30]

EFUSE_PGM_DATA6_REG ~ EFUSE_PGM_DATA7_REG 以及 EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG 中的数据不影响 BLOCK0 的烧写。

烧写 BLOCK1

当 EFUSE_BLK_NUM = 1 时，EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA5_REG 存储着 BLOCK1 即将烧写的参数，EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG 中存储着对应的 RS 校验码。EFUSE_PGM_DATA6_REG ~ EFUSE_PGM_DATA7_REG 中的数据不影响 BLOCK1 的烧写。RS 校验码的计算视这些位为 0。

烧写 BLOCK2 ~ 10

当 EFUSE_BLK_NUM = 2 ~ 10 时，EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG 存储着即将烧写的参数，EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG 中存储着对应的 RS 校验码。

烧写流程

烧写参数的流程如下：

1. 根据以上描述配置 EFUSE_BLK_NUM 参数，决定烧写哪一个块。

2. 将需要烧写的参数填写到寄存器 EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG 和 EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG 中。
3. 配置寄存器 EFUSE_CONF_REG 的 EFUSE_OP_CODE 位段为 0x5A5A。
4. 配置寄存器 EFUSE_CMD_REG 的 EFUSE_PGM_CMD 位段为 1。
5. 轮询寄存器 EFUSE_CMD_REG 直到其为 0x0，或者等待烧写完成中断 (PGM_DONE) 产生。识别烧写完成中断产生的方法详见章节 5.3.3 最后的说明。
6. 将 EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG 和 EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG 中写入的参数清零，防止烧写内容泄露。
7. 执行更新 eFuse 读寄存器操作使写入的新值生效，具体请参考章节 5.3.3。
8. 检查错误寄存器内容。若读取错误寄存器内数值不为 0，需要再次执行上述步骤 1 ~ 7 重新烧写一次。解决烧写不充分导致错误寄存器内数值不为 0 的问题，对于不同的 eFuse 块，需要检查的错误寄存器如下。
    - 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]

限制

BLOCK0 中不同的参数，甚至对于同一个参数中的不同位可以在多次烧写中分别完成。但是并不推荐这样做，而是建议尽量减少烧写次数。我们建议对于某个参数中的所有需要烧写的位都在一次烧写中完成。并且当 EFUSE_WR_DIS 的某个位管理的所有参数都烧写之后，就立即烧写 EFUSE_WR_DIS 的这个位。甚至可以在同一次烧写中既烧写 EFUSE_WR_DIS 的某个位管理的所有参数，同时也烧写 EFUSE_WR_DIS 的这个位。另外严禁对已经烧写了的位重复烧写，否则将发生烧写错误。

BLOCK1 中数据信息在出厂时已经烧写完毕，不允许再次烧写。

BLOCK2 ~ 10 中每一个 BLOCK 都只能烧写一次，不允许重复烧写。

5.3.3 用户读取参数

用户不能直接读取 eFuse 中烧写的信息内容。eFuse 控制器能够将烧写的数据信息读取到对应的地址段的寄存器内，用户再通过读取以 EFUSE_RD_ 开始的寄存器来获取 eFuse 信息。详细信息见表 5.3-4。

表 5.3-4. 寄存器信息

BLOCK	读寄存器	烧写寄存器
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_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_PART2_0 ~ 7_REG	EFUSE_PGM_DATA0 ~ 7_REG

更新 eFuse 读寄存器

eFuse 控制器读取内部 eFuse 来更新相应寄存器的数据。读取操作在系统复位时进行，也可以根据需要由用户手动触发（例如在需要读取新烧写 eFuse 中的数据内容时）。用户触发 eFuse 读取操作的流程如下：

1. 配置寄存器 EFUSE_CONF_REG 的 EFUSE_OP_CODE 位段为 0x5AA5。
2. 配置寄存器 EFUSE_CMD_REG 的 EFUSE_READ_CMD 位段为 1。
3. 轮询寄存器 EFUSE_CMD_REG 直到其为 0x0，或者等待读取完成 (READ_DONE) 产生，识别读取完成中断产生的方法详见下方说明。
4. 用户从 eFuse 存储器中读取参数的值。

eFuse 读寄存器中的数值将一直保持到下一次执行更新 eFuse 读操作。

烧写错误检测

烧写错误记录寄存器允许用户检测 eFuse 参数的备份是否有不一致的错误。

EFUSE_RD_REPEAT_ERR0 ~ 3_REG 寄存器用于指示 BLOCK0 中除了 EFUSE_WR_DIS 外的其他参数的烧写是否出错（对应位为 1 代表烧写出错，此位作废；为 0 代表烧写正确）。

EFUSE_RD_RS_ERR0 ~ 1_REG 寄存器记录 eFuse 读 BLOCK1 ~ BLOCK10 过程中，纠错的字节数目以及 RS 解码是否失败的信息。

用户只可以在更新 eFuse 读寄存器操作完成之后才可以读取上面几个寄存器的值。

识别烧写/读取操作完成

识别烧写/读取操作完成的方法如下。位 1 对应烧写操作，位 0 对应读取操作。

- 方法 1:
  1. 轮询寄存器 EFUSE_INT_RAW_REG 的位 1/0，直到位 1/0 为 1，表示烧写/读取操作完成。
- 方法 2:
  1. 对寄存器 EFUSE_INT_ENA_REG 的位 1/0 置 1，使 eFuse 控制器能够产生烧写或读取完成中断。
  2. 配置中断矩阵使 CPU 能够响应 eFuse 的中断信号，可参见章节 9 中断矩阵 *(INTERRUPT)*。
  3. 等待烧写或读取完成中断产生。
  4. 对寄存器 EFUSE_INT_CLR_REG 的位 1/0 置 1 以分别清除烧写或读取完成中断。

注意事项

在 eFuse 控制器执行寄存器更新操作过程中，会复用 EFUSE_PGM_DATA*n*_REG (n=0，1，..,7) 寄存器的存储空间，所以在启动 eFuse 控制器更新寄存器之前，不要将有意义的数据写入上述寄存器中。
芯片启动过程中，eFuse 控制器会自动更新 eFuse 数据到用户可访问的寄存器。用户可以通过读取相应的寄存器获取 eFuse 内烧写的数据。因此，用户无需再驱动 eFuse 控制器执行读更新操作。

5.3.4 eFuse VDDQ 时序

eFuse 控制器工作在 20 MHz 时钟频率下，其烧写电压 VDDQ 的配置参数需要满足以下条件：

- EFUSE_DAC_NUM（烧写电压上升周期数）：默认烧写电压为 2.5 V，每个上升周期增加 0.01 V，该参数对应的默认值为 255；
- EFUSE_DAC_CLK_DIV（烧写电压时钟分频系数）：要求烧写电压时钟周期大于 1 μs；
- EFUSE_PWR_ON_NUM（eFuse 烧写电压上电等待时间）：要求该等待时间结束后烧写电压已稳定，即要求配置数值大于 EFUSE_DAC_CLK_DIV 乘 EFUSE_DAC_NUM 的值；
- EFUSE_PWR_OFF_NUM（烧写电压掉电等待时间）：要求该时间大于 10 μs；

表 5.3-5. VDDQ 默认时序参数配置

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

5.3.5 硬件模块使用参数

硬件模块使用参数是通过电路连接实现的，为表 5.3-1 和 5.3-3 “硬件使用”一栏中标记为 “Y” 的参数。用户无法干预这个过程。

5.3.6 中断

- 烧写完成 (PGM_DONE) 中断：当 eFuse 烧写完成后，此中断被触发。如果要启动该中断信号，需将 EFUSE_PGM_DONE_INT_ENA 置 1。
- 读取完成 (READ_DONE) 中断：当 eFuse 读取完成后，此中断被触发。如果要启动该中断信号，需将 EFUSE_READ_DONE_INT_ENA 置 1。

5.4 寄存器列表

本小节的所有地址均为相对于 eFuse 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

07

名称	描述	地址	访问
烧写数据寄存器			
EFUSE_PGM_DATA0_REG	存放待烧写数据的第 0 个寄存器内容	0x0000	R/W
EFUSE_PGM_DATA1_REG	存放待烧写数据的第 1 个寄存器内容	0x0004	R/W
EFUSE_PGM_DATA2_REG	存放待烧写数据的第 2 个寄存器内容	0x0008	R/W
EFUSE_PGM_DATA3_REG	存放待烧写数据的第 3 个寄存器内容	0x000C	R/W
EFUSE_PGM_DATA4_REG	存放待烧写数据的第 4 个寄存器内容	0x0010	R/W
EFUSE_PGM_DATA5_REG	存放待烧写数据的第 5 个寄存器内容	0x0014	R/W
EFUSE_PGM_DATA6_REG	存放待烧写数据的第 6 个寄存器内容	0x0018	R/W
EFUSE_PGM_DATA7_REG	存放待烧写数据的第 7 个寄存器内容	0x001C	R/W
EFUSE_PGM_CHECK_VALUE0_REG	存放待烧写 RS 代码的第 0 个寄存器数据	0x0020	R/W
EFUSE_PGM_CHECK_VALUE1_REG	存放待烧写 RS 代码的第 1 个寄存器数据	0x0024	R/W
EFUSE_PGM_CHECK_VALUE2_REG	存放待烧写 RS 代码的第 2 个寄存器数据	0x0028	R/W
读取数据寄存器			
EFUSE_RD_WR_DIS_REG	BLOCK0 的第 0 个寄存器内容	0x002C	RO
EFUSE_RD_REPEAT_DATA0_REG	BLOCK0 的第 1 个寄存器内容	0x0030	RO
EFUSE_RD_REPEAT_DATA1_REG	BLOCK0 的第 2 个寄存器内容	0x0034	RO
EFUSE_RD_REPEAT_DATA2_REG	BLOCK0 的第 3 个寄存器内容	0x0038	RO
EFUSE_RD_REPEAT_DATA3_REG	BLOCK0 的第 4 个寄存器内容	0x003C	RO
EFUSE_RD_REPEAT_DATA4_REG	BLOCK0 的第 5 个寄存器内容	0x0040	RO
EFUSE_RD_MAC_SPI_SYS_0_REG	BLOCK1 的第 0 个寄存器内容	0x0044	RO
EFUSE_RD_MAC_SPI_SYS_1_REG	BLOCK1 的第 1 个寄存器内容	0x0048	RO
EFUSE_RD_MAC_SPI_SYS_2_REG	BLOCK1 的第 2 个寄存器内容	0x004C	RO
EFUSE_RD_MAC_SPI_SYS_3_REG	BLOCK1 的第 3 个寄存器内容	0x0050	RO
EFUSE_RD_MAC_SPI_SYS_4_REG	BLOCK1 的第 4 个寄存器内容	0x0054	RO
EFUSE_RD_MAC_SPI_SYS_5_REG	BLOCK1 的第 5 个寄存器内容	0x0058	RO
EFUSE_RD_SYS_PART1_DATA0_REG	BLOCK2 (system) 的第 0 个寄存器内容	0x005C	RO
EFUSE_RD_SYS_PART1_DATA1_REG	BLOCK2 (system) 的第 1 个寄存器内容	0x0060	RO
EFUSE_RD_SYS_PART1_DATA2_REG	BLOCK2 (system) 的第 2 个寄存器内容	0x0064	RO
EFUSE_RD_SYS_PART1_DATA3_REG	BLOCK2 (system) 的第 3 个寄存器内容	0x0068	RO
EFUSE_RD_SYS_PART1_DATA4_REG	BLOCK2 (system) 的第 4 个寄存器内容	0x006C	RO
EFUSE_RD_SYS_PART1_DATA5_REG	BLOCK2 (system) 的第 5 个寄存器内容	0x0070	RO
EFUSE_RD_SYS_PART1_DATA6_REG	BLOCK2 (system) 的第 6 个寄存器内容	0x0074	RO
EFUSE_RD_SYS_PART1_DATA7_REG	BLOCK2 (system) 的第 7 个寄存器内容	0x0078	RO
EFUSE_RD_USR_DATA0_REG	BLOCK3 (user) 的第 0 个寄存器内容	0x007C	RO
EFUSE_RD_USR_DATA1_REG	BLOCK3 (user) 的第 1 个寄存器内容	0x0080	RO
EFUSE_RD_USR_DATA2_REG	BLOCK3 (user) 的第 2 个寄存器内容	0x0084	RO

名称	描述	地址	访问
EFUSE_RD_USR_DATA3_REG	BLOCK3 (user) 的第 3 个寄存器内容	0x0088	RO
EFUSE_RD_USR_DATA4_REG	BLOCK3 (user) 的第 4 个寄存器内容	0x008C	RO
EFUSE_RD_USR_DATA5_REG	BLOCK3 (user) 的第 5 个寄存器内容	0x0090	RO
EFUSE_RD_USR_DATA6_REG	BLOCK3 (user) 的第 6 个寄存器内容	0x0094	RO
EFUSE_RD_USR_DATA7_REG	BLOCK3 (user) 的第 7 个寄存器内容	0x0098	RO
EFUSE_RD_KEY0_DATA0_REG	BLOCK4 (KEY0) 的第 0 个寄存器内容	0x009C	RO
EFUSE_RD_KEY0_DATA1_REG	BLOCK4 (KEY0) 的第 1 个寄存器内容	0x00A0	RO
EFUSE_RD_KEY0_DATA2_REG	BLOCK4 (KEY0) 的第 2 个寄存器内容	0x00A4	RO
EFUSE_RD_KEY0_DATA3_REG	BLOCK4 (KEY0) 的第 3 个寄存器内容	0x00A8	RO
EFUSE_RD_KEY0_DATA4_REG	BLOCK4 (KEY0) 的第 4 个寄存器内容	0x00AC	RO
EFUSE_RD_KEY0_DATA5_REG	BLOCK4 (KEY0) 的第 5 个寄存器内容	0x00B0	RO
EFUSE_RD_KEY0_DATA6_REG	BLOCK4 (KEY0) 的第 6 个寄存器内容	0x00B4	RO
EFUSE_RD_KEY0_DATA7_REG	BLOCK4 (KEY0) 的第 7 个寄存器内容	0x00B8	RO
EFUSE_RD_KEY1_DATA0_REG	BLOCK5 (KEY1) 的第 0 个寄存器内容	0x00BC	RO
EFUSE_RD_KEY1_DATA1_REG	BLOCK5 (KEY1) 的第 1 个寄存器内容	0x00C0	RO
EFUSE_RD_KEY1_DATA2_REG	BLOCK5 (KEY1) 的第 2 个寄存器内容	0x00C4	RO
EFUSE_RD_KEY1_DATA3_REG	BLOCK5 (KEY1) 的第 3 个寄存器内容	0x00C8	RO
EFUSE_RD_KEY1_DATA4_REG	BLOCK5 (KEY1) 的第 4 个寄存器内容	0x00CC	RO
EFUSE_RD_KEY1_DATA5_REG	BLOCK5 (KEY1) 的第 5 个寄存器内容	0x00D0	RO
EFUSE_RD_KEY1_DATA6_REG	BLOCK5 (KEY1) 的第 6 个寄存器内容	0x00D4	RO
EFUSE_RD_KEY1_DATA7_REG	BLOCK5 (KEY1) 的第 7 个寄存器内容	0x00D8	RO
EFUSE_RD_KEY2_DATA0_REG	BLOCK6 (KEY2) 的第 0 个寄存器内容	0x00DC	RO
EFUSE_RD_KEY2_DATA1_REG	BLOCK6 (KEY2) 的第 1 个寄存器内容	0x00E0	RO
EFUSE_RD_KEY2_DATA2_REG	BLOCK6 (KEY2) 的第 2 个寄存器内容	0x00E4	RO
EFUSE_RD_KEY2_DATA3_REG	BLOCK6 (KEY2) 的第 3 个寄存器内容	0x00E8	RO
EFUSE_RD_KEY2_DATA4_REG	BLOCK6 (KEY2) 的第 4 个寄存器内容	0x00EC	RO
EFUSE_RD_KEY2_DATA5_REG	BLOCK6 (KEY2) 的第 5 个寄存器内容	0x00F0	RO
EFUSE_RD_KEY2_DATA6_REG	BLOCK6 (KEY2) 的第 6 个寄存器内容	0x00F4	RO
EFUSE_RD_KEY2_DATA7_REG	BLOCK6 (KEY2) 的第 7 个寄存器内容	0x00F8	RO
EFUSE_RD_KEY3_DATA0_REG	BLOCK7 (KEY3) 的第 0 个寄存器内容	0x00FC	RO
EFUSE_RD_KEY3_DATA1_REG	BLOCK7 (KEY3) 的第 1 个寄存器内容	0x0100	RO
EFUSE_RD_KEY3_DATA2_REG	BLOCK7 (KEY3) 的第 2 个寄存器内容	0x0104	RO
EFUSE_RD_KEY3_DATA3_REG	BLOCK7 (KEY3) 的第 3 个寄存器内容	0x0108	RO
EFUSE_RD_KEY3_DATA4_REG	BLOCK7 (KEY3) 的第 4 个寄存器内容	0x010C	RO
EFUSE_RD_KEY3_DATA5_REG	BLOCK7 (KEY3) 的第 5 个寄存器内容	0x0110	RO
EFUSE_RD_KEY3_DATA6_REG	BLOCK7 (KEY3) 的第 6 个寄存器内容	0x0114	RO
EFUSE_RD_KEY3_DATA7_REG	BLOCK7 (KEY3) 的第 7 个寄存器内容	0x0118	RO
EFUSE_RD_KEY4_DATA0_REG	BLOCK8 (KEY4) 的第 0 个寄存器内容	0x011C	RO
EFUSE_RD_KEY4_DATA1_REG	BLOCK8 (KEY4) 的第 1 个寄存器内容	0x0120	RO
EFUSE_RD_KEY4_DATA2_REG	BLOCK8 (KEY4) 的第 2 个寄存器内容	0x0124	RO
EFUSE_RD_KEY4_DATA3_REG	BLOCK8 (KEY4) 的第 3 个寄存器内容	0x0128	RO
EFUSE_RD_KEY4_DATA4_REG	BLOCK8 (KEY4) 的第 4 个寄存器内容	0x012C	RO
EFUSE_RD_KEY4_DATA5_REG	BLOCK8 (KEY4) 的第 5 个寄存器内容	0x0130	RO

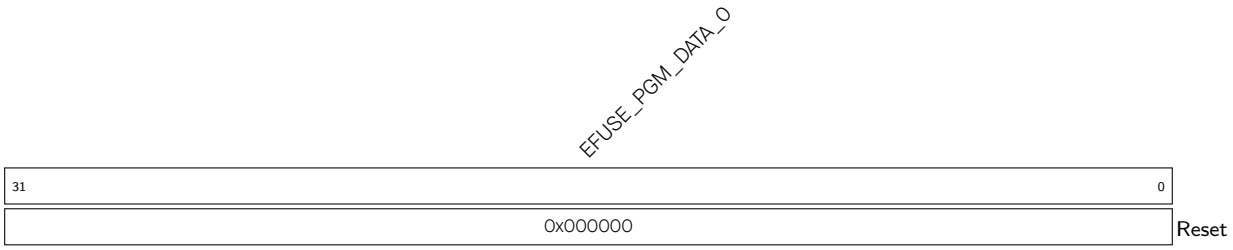
名称	描述	地址	访问
EFUSE_RD_KEY4_DATA6_REG	BLOCK8 (KEY4) 的第 6 个寄存器内容	0x0134	RO
EFUSE_RD_KEY4_DATA7_REG	BLOCK8 (KEY4) 的第 7 个寄存器内容	0x0138	RO
EFUSE_RD_KEY5_DATA0_REG	BLOCK9 (KEY5) 的第 0 个寄存器内容	0x013C	RO
EFUSE_RD_KEY5_DATA1_REG	BLOCK9 (KEY5) 的第 1 个寄存器内容	0x0140	RO
EFUSE_RD_KEY5_DATA2_REG	BLOCK9 (KEY5) 的第 2 个寄存器内容	0x0144	RO
EFUSE_RD_KEY5_DATA3_REG	BLOCK9 (KEY5) 的第 3 个寄存器内容	0x0148	RO
EFUSE_RD_KEY5_DATA4_REG	BLOCK9 (KEY5) 的第 4 个寄存器内容	0x014C	RO
EFUSE_RD_KEY5_DATA5_REG	BLOCK9 (KEY5) 的第 5 个寄存器内容	0x0150	RO
EFUSE_RD_KEY5_DATA6_REG	BLOCK9 (KEY5) 的第 6 个寄存器内容	0x0154	RO
EFUSE_RD_KEY5_DATA7_REG	BLOCK9 (KEY5) 的第 7 个寄存器内容	0x0158	RO
EFUSE_RD_SYS_PART2_DATA0_REG	BLOCK10 (system) 的第 0 个寄存器内容	0x015C	RO
EFUSE_RD_SYS_PART2_DATA1_REG	BLOCK10 (system) 的第 1 个寄存器内容	0x0160	RO
EFUSE_RD_SYS_PART2_DATA2_REG	BLOCK10 (system) 的第 2 个寄存器内容	0x0164	RO
EFUSE_RD_SYS_PART2_DATA3_REG	BLOCK10 (system) 的第 3 个寄存器内容	0x0168	RO
EFUSE_RD_SYS_PART2_DATA4_REG	BLOCK10 (system) 的第 4 个寄存器内容	0x016C	RO
EFUSE_RD_SYS_PART2_DATA5_REG	BLOCK10 (system) 的第 5 个寄存器内容	0x0170	RO
EFUSE_RD_SYS_PART2_DATA6_REG	BLOCK10 (system) 的第 6 个寄存器内容	0x0174	RO
EFUSE_RD_SYS_PART2_DATA7_REG	BLOCK10 (system) 的第 7 个寄存器内容	0x0178	RO
报告寄存器			
EFUSE_RD_REPEAT_ERR0_REG	BLOCK0 参数烧写错误记录第 0 个寄存器	0x017C	RO
EFUSE_RD_REPEAT_ERR1_REG	BLOCK0 参数烧写错误记录第 1 个寄存器	0x0180	RO
EFUSE_RD_REPEAT_ERR2_REG	BLOCK0 参数烧写错误记录第 2 个寄存器	0x0184	RO
EFUSE_RD_REPEAT_ERR3_REG	BLOCK0 参数烧写错误记录第 3 个寄存器	0x0188	RO
EFUSE_RD_REPEAT_ERR4_REG	BLOCK0 参数烧写错误记录第 4 个寄存器	0x0190	RO
EFUSE_RD_RS_ERR0_REG	记录 BLOCK1 ~ 10 参数烧写错误信息的第 0 个寄存器	0x01C0	RO
EFUSE_RD_RS_ERR1_REG	记录 BLOCK1 ~ 10 参数烧写错误信息的第 1 个寄存器	0x01C4	RO
配置寄存器			
EFUSE_CLK_REG	eFuse 时钟配置寄存器	0x01C8	R/W
EFUSE_CONF_REG	eFuse 运行模式配置寄存器	0x01CC	R/W
EFUSE_CMD_REG	eFuse 指令寄存器	0x01D4	varies
EFUSE_DAC_CONF_REG	eFuse 烧写电压控制寄存器	0x01E8	R/W
EFUSE_RD_TIM_CONF_REG	eFuse 读取时序参数配置寄存器	0x01EC	R/W
EFUSE_WR_TIM_CONF1_REG	eFuse 烧写时序参数第 1 个配置寄存器	0x01F4	R/W
EFUSE_WR_TIM_CONF2_REG	eFuse 烧写时序参数第 2 个配置寄存器	0x01F8	R/W
状态寄存器			
EFUSE_STATUS_REG	eFuse 状态寄存器	0x01D0	RO
中断寄存器			
EFUSE_INT_RAW_REG	eFuse 原始中断寄存器	0x01D8	R/WC/SS
EFUSE_INT_ST_REG	eFuse 中断状态寄存器	0x01DC	RO

名称	描述	地址	访问
EFUSE_INT_ENA_REG	eFuse 中断使能寄存器	0x01E0	R/W
EFUSE_INT_CLR_REG	eFuse 中断清除寄存器	0x01E4	WO
版本寄存器			
EFUSE_DATE_REG	版本控制寄存器	0x01FC	R/W

5.5 寄存器

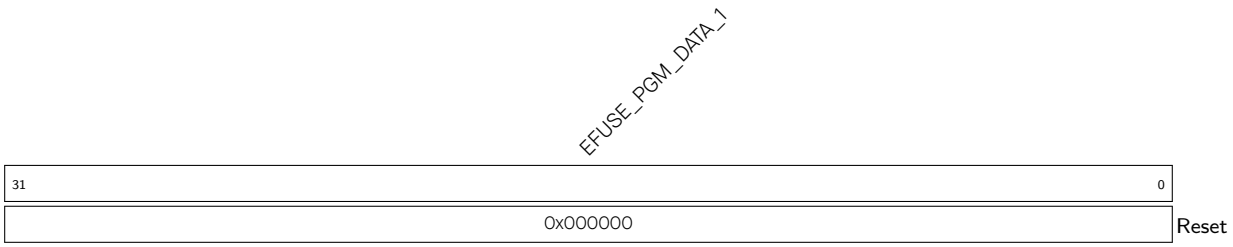
本小节的所有地址均为相对于 eFuse 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 5.1. EFUSE_PGM_DATA0_REG (0x0000)



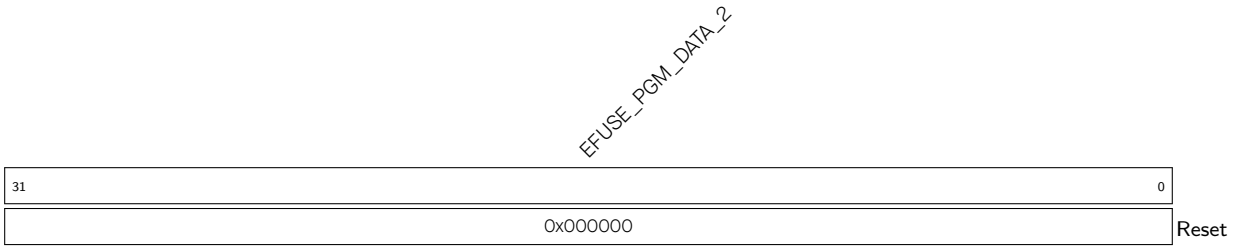
EFUSE_PGM_DATA_0 配置待烧写数据的第 0 个 32 位数据内容。(R/W)

Register 5.2. EFUSE_PGM_DATA1_REG (0x0004)



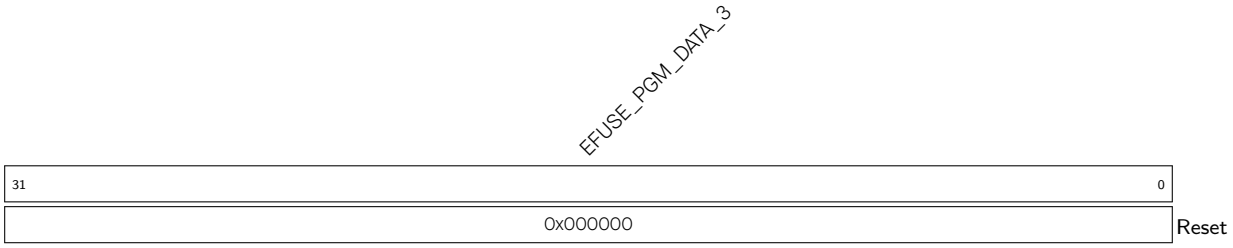
EFUSE_PGM_DATA_1 配置待烧写数据的第 1 个 32 位数据内容。(R/W)

Register 5.3. EFUSE_PGM_DATA2_REG (0x0008)



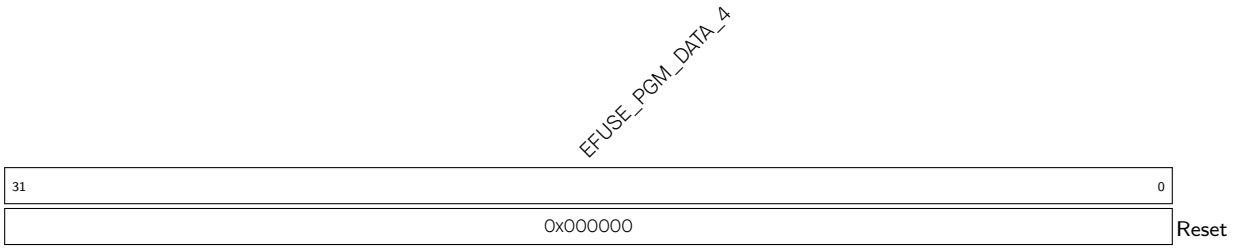
EFUSE_PGM_DATA_2 配置待烧写数据的第 2 个 32 位数据内容。(R/W)

Register 5.4. EFUSE_PGM_DATA3_REG (0x000C)



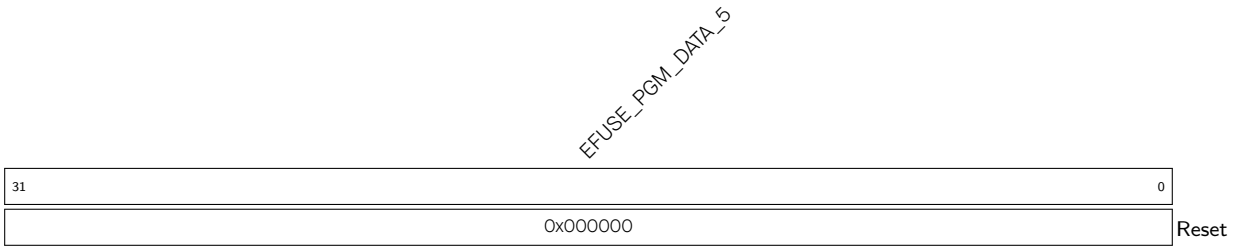
EFUSE_PGM_DATA_3 配置待烧写数据的第 3 个 32 位数据内容。(R/W)

Register 5.5. EFUSE_PGM_DATA4_REG (0x0010)



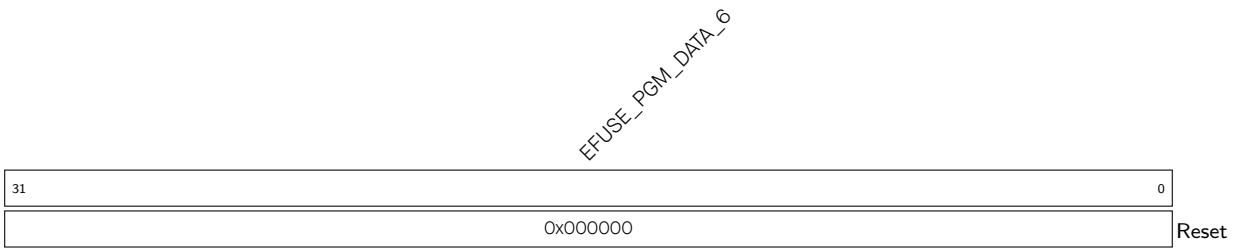
EFUSE_PGM_DATA_4 配置待烧写数据的第 4 个 32 位数据内容。(R/W)

Register 5.6. EFUSE_PGM_DATA5_REG (0x0014)



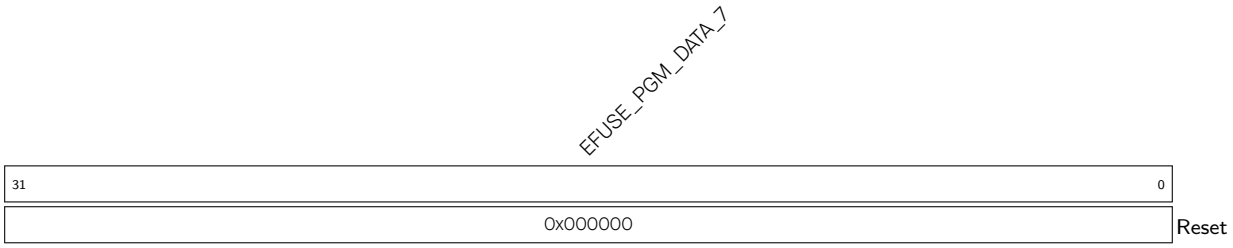
EFUSE_PGM_DATA_5 配置待烧写数据的第 5 个 32 位数据内容。(R/W)

Register 5.7. EFUSE_PGM_DATA6_REG (0x0018)



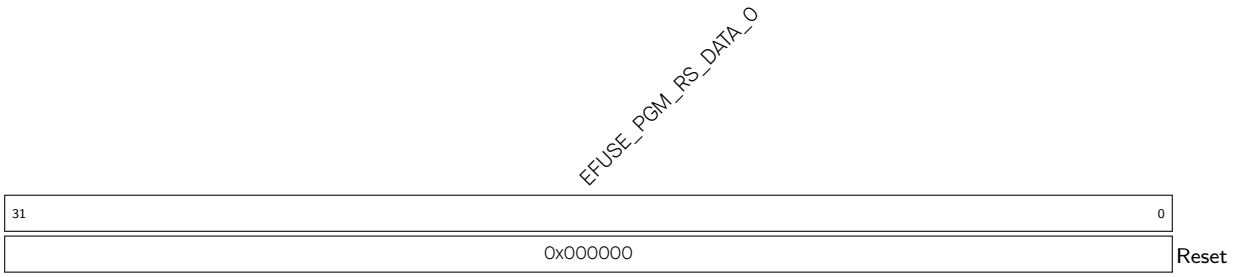
EFUSE_PGM_DATA_6 配置待烧写数据的第 6 个 32 位数据内容。(R/W)

Register 5.8. EFUSE_PGM_DATA7_REG (0x001C)



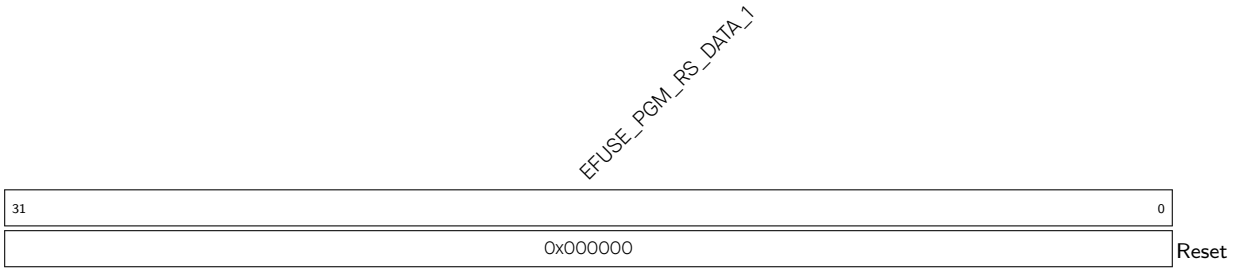
EFUSE_PGM_DATA_7 配置待烧写数据的第 7 个 32 位数据内容。(R/W)

Register 5.9. EFUSE_PGM_CHECK_VALUE0_REG (0x0020)



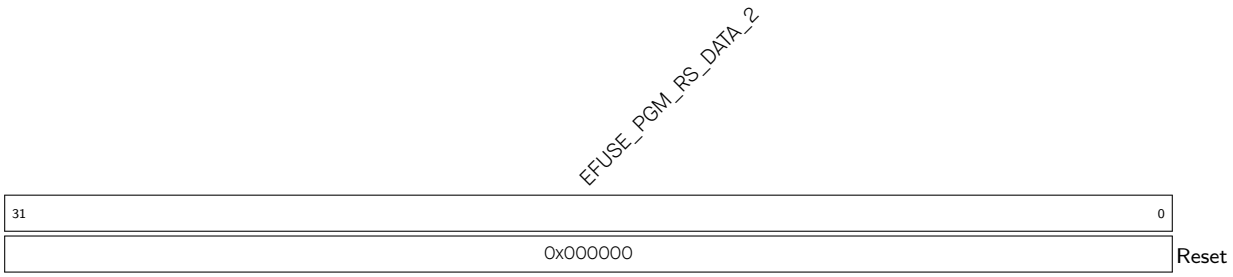
EFUSE_PGM_RS_DATA_0 配置待烧写 RS 代码的第 0 个 32 位数据内容。(R/W)

Register 5.10. EFUSE_PGM_CHECK_VALUE1_REG (0x0024)



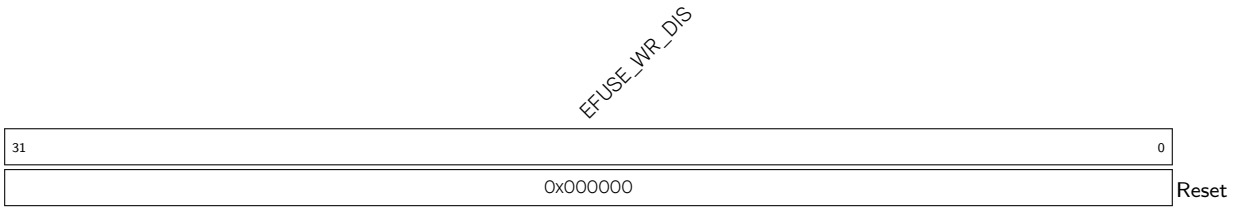
EFUSE_PGM_RS_DATA_1 配置待烧写 RS 代码的第 1 个 32 位数据内容。(R/W)

Register 5.11. EFUSE_PGM_CHECK_VALUE2_REG (0x0028)



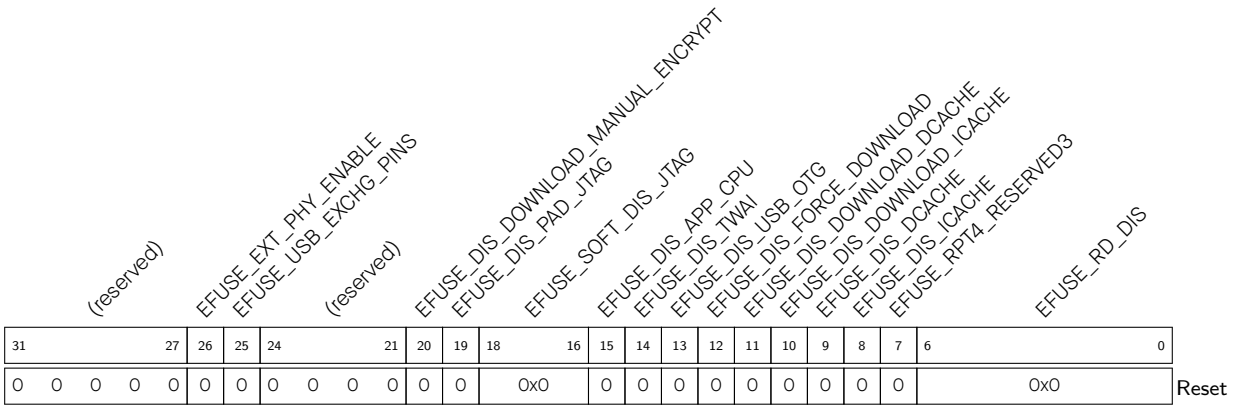
EFUSE_PGM_RS_DATA_2 配置待烧写 RS 代码的第 2 个 32 位数据内容。(R/W)

Register 5.12. EFUSE_RD_WR_DIS_REG (0x002C)



EFUSE_WR_DIS 表示是否禁用相应 eFuse 数据段的烧写。1: 禁用。0: 启用。(RO)

Register 5.13. EFUSE_RD_REPEAT_DATA0_REG (0x0030)



EFUSE_RD_DIS 表示是否禁用读取 eFuse Block4 ~ 10 的内容。1：禁用。0：启用。(RO)

EFUSE_RPT4_RESERVED3 保留（采用 4 备份编码）。(RO)

EFUSE_DIS_ICACHE 表示是否禁用 iCache。1：禁用。0：启用。(RO)

EFUSE_DIS_DCACHE 表示是否禁用 dCache。1：禁用。0：启用。(RO)

EFUSE_DIS_DOWNLOAD_ICACHE 表示是否在下载模式下关闭 iCache（boot_mode[3:0] 为 0, 1, 2, 3, 6, 7）。1：禁用。0：启用。(RO)

EFUSE_DIS_DOWNLOAD_DCACHE 表示是否在下载模式下关闭 dCache（boot_mode[3:0] 为 0, 1, 2, 3, 6, 7）。1：禁用。0：启用。(RO)

EFUSE_DIS_FORCE_DOWNLOAD 表示是否禁止强制芯片进入下载模式。1：禁用。0：启用。(RO)

EFUSE_DIS_USB_OTG 表示是否禁用 USB OTG 功能。1：禁用。0：启用。(RO)

EFUSE_DIS_TWAI 表示是否禁用 TWAI 功能。1：禁用。0：启用。(RO)

EFUSE_DIS_APP_CPU 表示是否禁用 app CPU。1：禁用。0：启用。(RO)

EFUSE_SOFT_DIS_JTAG 表示是否软关断 JTAG 功能。奇数个 1：关断。用户还可以通过 HAMC 模块再次打开 JTAG。偶数个 1：不关断。(RO)

EFUSE_DIS_PAD_JTAG 表示是否永久禁用 JTAG 功能。1：禁用。0：启用。(RO)

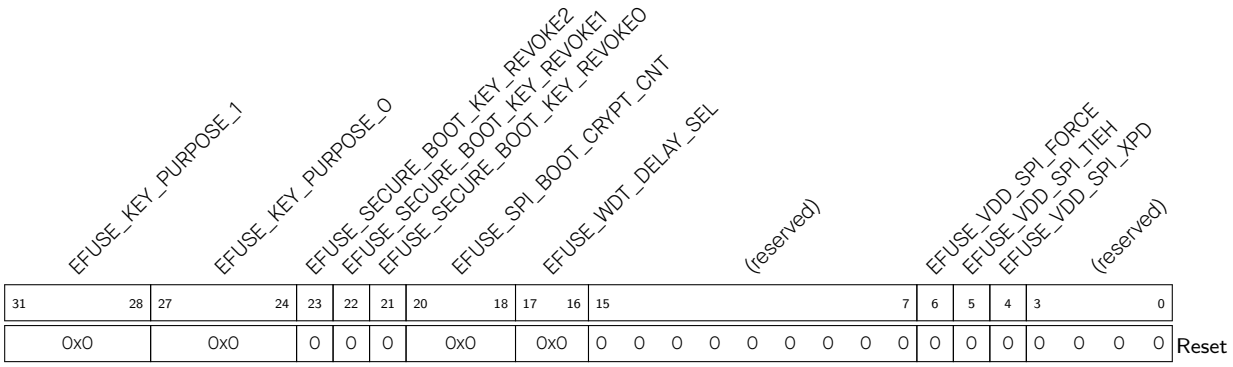
EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT 表示是否在 download boot 模式下禁用 flash 加密功能。1：禁用。0：启用。(RO)

EFUSE_USB_EXCHG_PINS 表示 USB D+ 和 D- 管脚是否交换。1：交换。0：不交换。(RO)

请注意：由于设计缺陷，该参数**不会**移除上拉电平（用于检测 USB 速度），因而 PC 会将芯片视为低速设备，从而停止通信。详细信息请参阅章节 *33 USB 串口/JTAG 控制器 (USB_SERIAL_JTAG)*。

EFUSE_EXT_PHY_ENABLE 表示是否使能外部 USB PHY。1：使能。0：不使能。(RO)

Register 5.14. EFUSE_RD_REPEAT_DATA1_REG (0x0034)



EFUSE_VDD_SPI_XPD 表示 flash 电压调节器是否上电。1：上电。0：不上电。(RO)

EFUSE_VDD_SPI_TIEH 表示 flash 电压调节器是否短接至 VDD3P3_RTC_IO。1：短接至 VDD3P3_RTC_IO。0：连接至 1.8 V flash 电压调节器。(RO)

EFUSE_VDD_SPI_FORCE 表示是否强制使用 EFUSE_VDD_SPI_XPD、EFUSE_VDD_SPI_TIEH 等 eFuse 参数配置 flash 电压 LDO。1：使用。0：不使用。(RO)

EFUSE_WDT_DELAY_SEL 表示 RTC 看门狗超时阈值。单位：慢速时钟周期。00: 40,000 个慢速时钟周期；01: 80,000 个慢速时钟周期；10: 160,000 个慢速时钟周期；11: 320,000 个慢速时钟周期。(RO)

EFUSE_SPI_BOOT_CRYPT_CNT 表示是否使能 SPI boot 加解密。奇数个 1：使能；偶数个 1：禁用。(RO)

EFUSE_SECURE_BOOT_KEY_REVOKE0 表示是否使能撤销第一个安全启动密钥。1：使能。0：不使能。(RO)

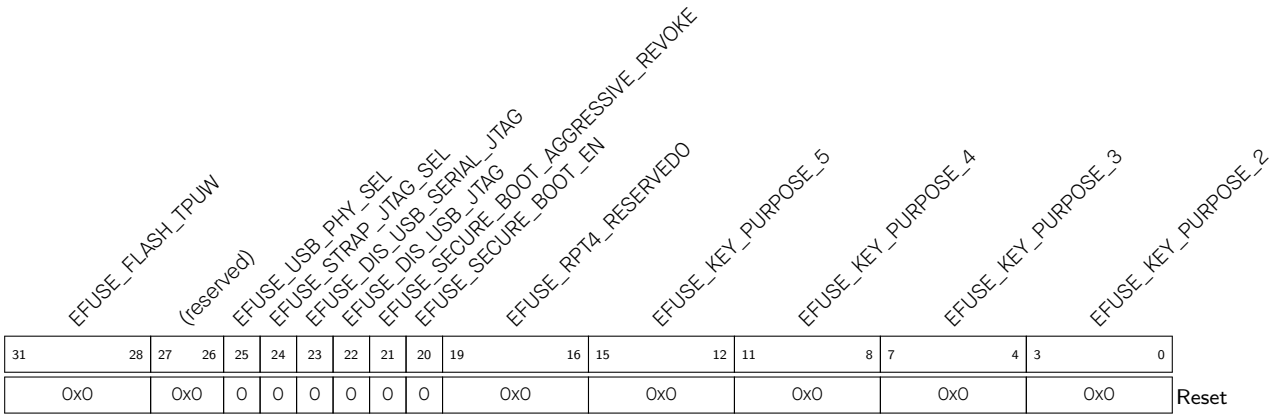
EFUSE_SECURE_BOOT_KEY_REVOKE1 表示是否使能撤销第二个安全启动密钥。1：使能。0：不使能。(RO)

EFUSE_SECURE_BOOT_KEY_REVOKE2 表示是否使能撤销第三个安全启动密钥。1：使能。0：不使能。(RO)

EFUSE_KEY_PURPOSE_0 表示 Key0 用途。(RO)

EFUSE_KEY_PURPOSE_1 表示 Key1 用途。(RO)

Register 5.15. EFUSE_RD_REPEAT_DATA2_REG (0x0038)



EFUSE_KEY_PURPOSE_2 表示 Key2 用途。(RO)

EFUSE_KEY_PURPOSE_3 表示 Key3 用途。(RO)

EFUSE_KEY_PURPOSE_4 表示 Key4 用途。(RO)

EFUSE_KEY_PURPOSE_5 表示 Key5 用途。(RO)

EFUSE_RPT4_RESERVED0 保留（采用 4 备份编码）。(RO)

EFUSE_SECURE_BOOT_EN 表示是否使能安全启动。1：使能。0：不使能。(RO)

EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE 表示是否使能密钥失效的激进策略。1：使能。0：不使能。(RO)

EFUSE_DIS_USB_JTAG 表示是否禁用 usb_serial_jtag 模块的 usb 转 jtag 功能。1：禁用。0：启用。(RO)

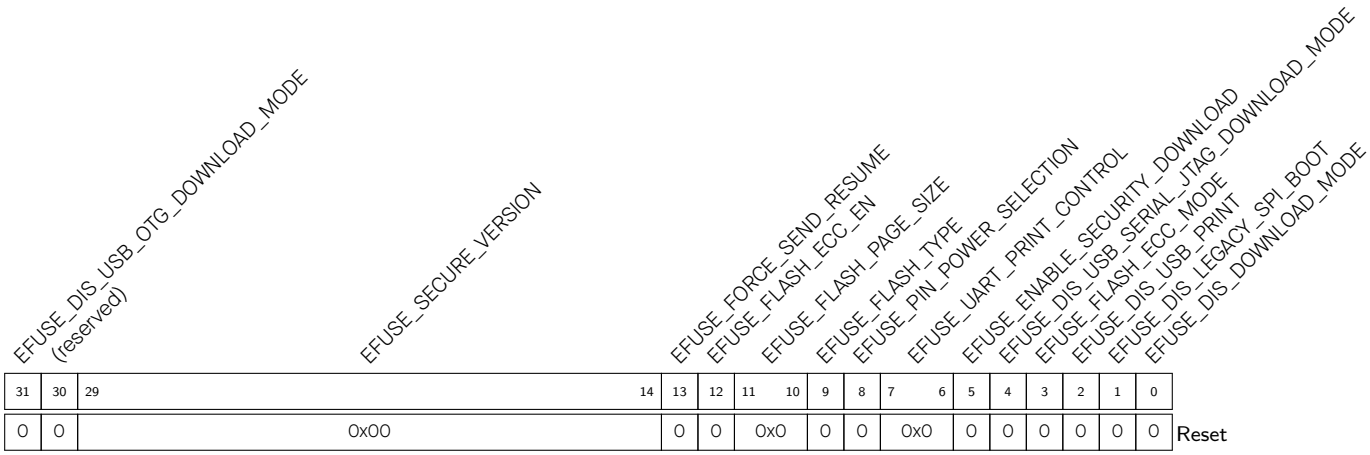
EFUSE_DIS_USB_SETIAL_JTAG 表示是否禁用 usb_serial_jtag 模块。1：禁用。0：启用。(RO)

EFUSE_STRAP_JTAG_SEL 表示当 reg_dis_usb_jtag 和 reg_dis_pad_jtag 都为 0 时，是否使能使用 strapping GPIO3 选择 usb_to_jtag 或 pad_to_jtag 的功能。1：使能。0：不使能。(RO)

EFUSE_USB_PHY_SEL 表示内部 PHY、外部 PHY 和 USB OTG、USB Serial/JTAG 之间的连接关系。0: USB Serial/JTAG 使用内部 PHY，USB OTG 使用外部 PHY。1: USB OTG 使用内部 PHY，USB Serial/JTAG 使用外部 PHY。(RO)

EFUSE_FLASH_TPUW 表示配置上电后 flash 等待时间。单位：ms。当值小于 15 时，等待时间为配置值。当值大于等于 15 时，等待时间固定为 30 ms。(RO)

Register 5.16. EFUSE_RD_REPEAT_DATA3_REG (0x003C)



EFUSE_DIS_DOWNLOAD_MODE 表示是否禁用下载模式 (boot_mode[3:0] = 0, 1, 2, 3, 6, 7)。1：禁用。0：启用。(RO)

EFUSE_DIS_LEGACY_SPI_BOOT 表示是否禁用 Legacy SPI boot 模式 (boot_mode[3:0] = 4)。1：禁用。0：启用。(RO)

EFUSE_DIS_USB_PRINT 表示是否禁用 USB 打印。1：禁用。0：启用。(RO)

EFUSE_FLASH_ECC_MODE 表示 ROM 中的 flash ECC 模式。0: 使能 16-to-18 字节模式；1：使能 16-to-17 字节模式。(RO)

EFUSE_DIS_USB_SERIAL_JTAG_DOWNLOAD_MODE 表示是否禁用 USB-Serial-JTAG 下载功能。1：禁用。0：启用。(RO)

EFUSE_ENABLE_SECURITY_DOWNLOAD 表示是否使能安全 UART 下载模式（仅支持读写 flash）。1：使能。0：不使能。(RO)

EFUSE_UART_PRINT_CONTROL 表示 UART boot 信息的默认打印方式。00: 使能打印；01: GPIO46 低电平复位时，使能打印；10: GPIO46 高电平复位时，使能打印；11: 关闭打印。(RO)

EFUSE_PIN_POWER_SELECTION 表示执行 ROM 代码时 GPIO33 ~ GPIO37、GPIO47、GPIO48 的供电来源。0：VDD3P3_CPU；1：VDD_SPI。(RO)

EFUSE_FLASH_TYPE 表示 SPI flash 的最大行数。0: 4 行；1: 8 行。(RO)

EFUSE_FLASH_PAGE_SIZE 表示 flash 的页大小。0：256 字节；1：512 字节；2：1 KB；3：2 KB。(RO)

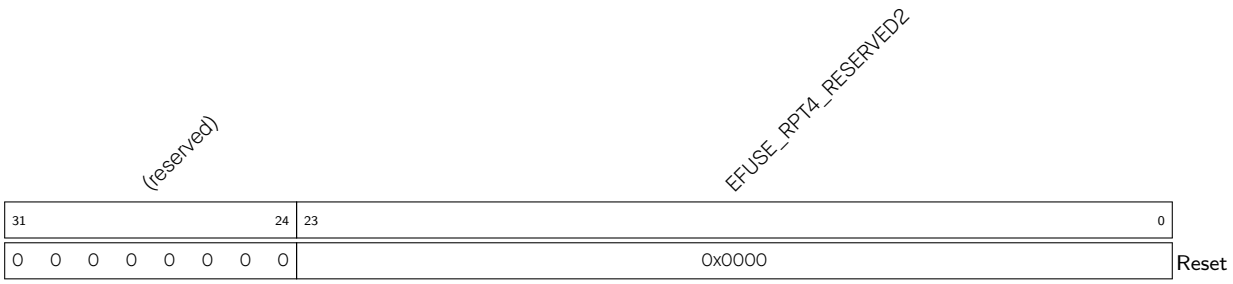
EFUSE_FLASH_ECC_EN 表示是否在 flash boot 中启用 ECC 功能。1：启用。0：禁用。(RO)

EFUSE_FORCE_SEND_RESUME 表示是否强制 ROM 代码在 SPI 启动过程中发送恢复指令。1：发送。0：不发送。(RO)

EFUSE_SECURE_VERSION 表示 IDF 安全版本（用于 ESP-IDF 的防回滚功能）。(RO)

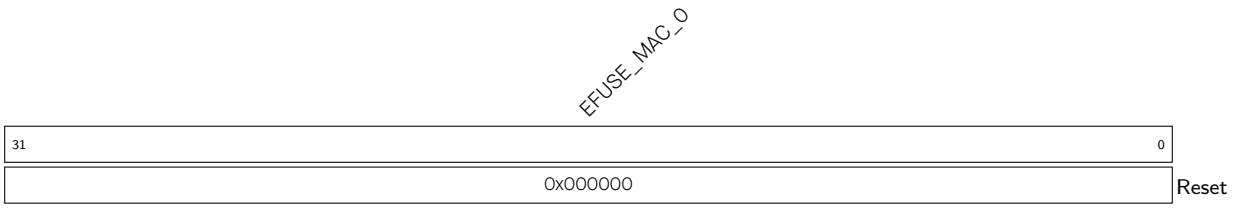
EFUSE_DIS_USB_OTG_DOWNLOAD_MODE 表示是否禁用 USB-OTG 下载功能。1：禁用。0：启用。(RO)

Register 5.17. EFUSE_RD_REPEAT_DATA4_REG (0x0040)



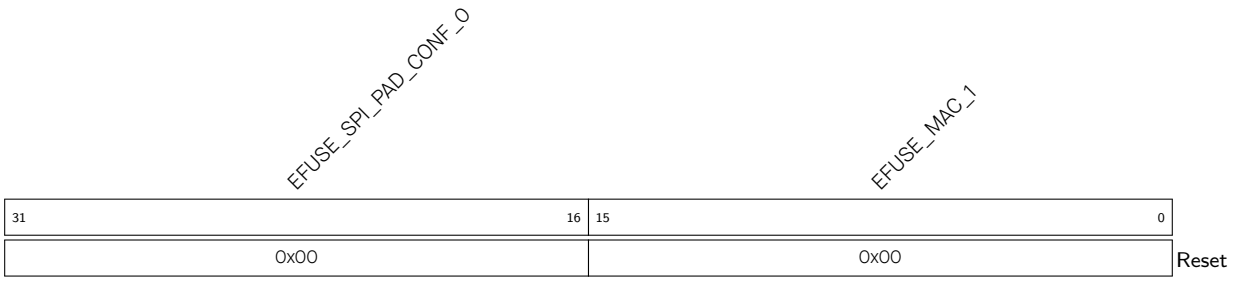
EFUSE_RPT4_RESERVED2 保留（采用 4 备份编码）。(RO)

Register 5.18. EFUSE_RD_MAC_SPI_SYS_0_REG (0x0044)



EFUSE_MAC_0 表示 MAC 地址低 32 位的内容。(RO)

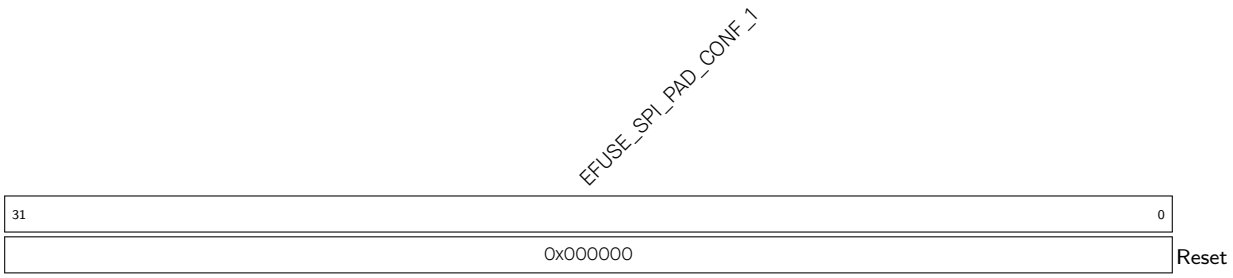
Register 5.19. EFUSE_RD_MAC_SPI_SYS_1_REG (0x0048)



EFUSE_MAC_1 表示 MAC 地址高 16 位的内容。(RO)

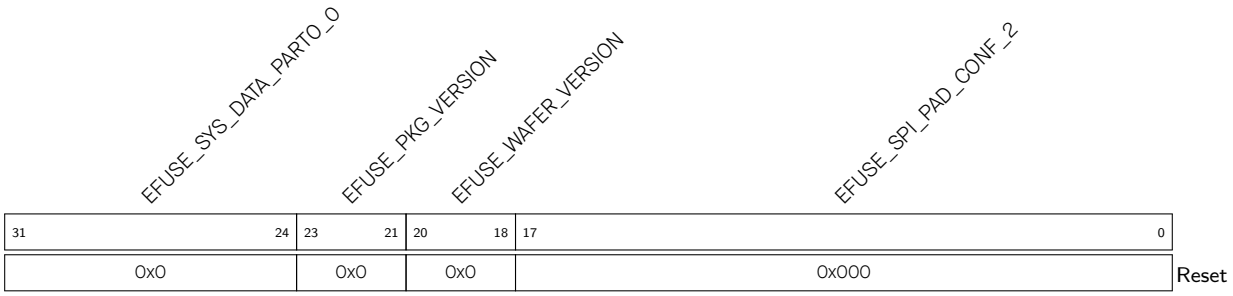
EFUSE_SPI_PAD_CONF_0 表示 SPI_PAD_CONF 第 0 部分的内容。(RO)

Register 5.20. EFUSE_RD_MAC_SPI_SYS_2_REG (0x004C)



EFUSE_SPI_PAD_CONF_1 表示 SPI_PAD_CONF 第 1 部分的内容。(RO)

Register 5.21. EFUSE_RD_MAC_SPI_SYS_3_REG (0x0050)



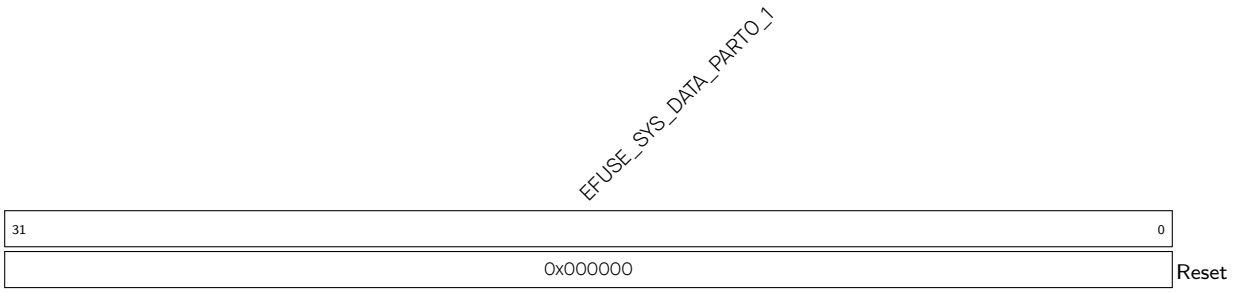
EFUSE_SPI_PAD_CONF_2 表示 SPI_PAD_CONF 第 2 部分的内容。(RO)

EFUSE_WAFER_VERSION 表示 wafer 版本信息。(RO)

EFUSE_PKG_VERSION 表示封装版本信息。(RO)

EFUSE_SYS_DATA_PART0_0 表示系统数据第 0 部分的第 0 个 8 位内容。(RO)

Register 5.22. EFUSE_RD_MAC_SPI_SYS_4_REG (0x0054)



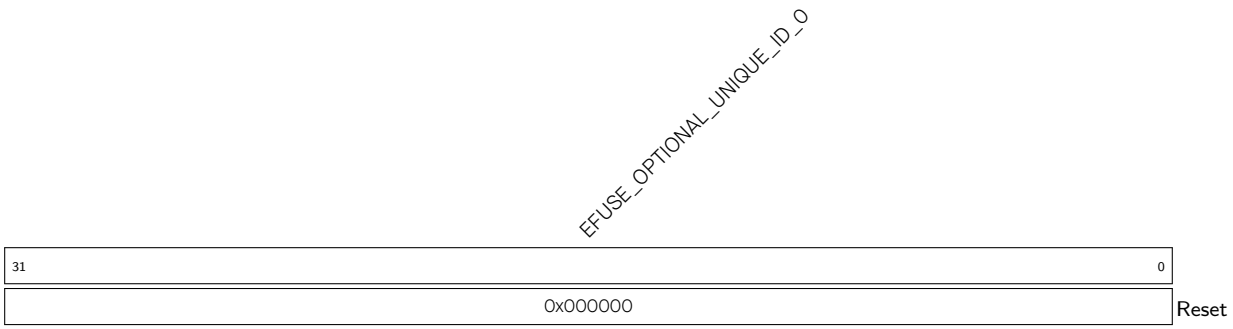
EFUSE_SYS_DATA_PART0_1 表示系统数据第 0 部分的第 1 个 32 位内容。(RO)

Register 5.23. EFUSE_RD_MAC_SPI_SYS_5_REG (0x0058)



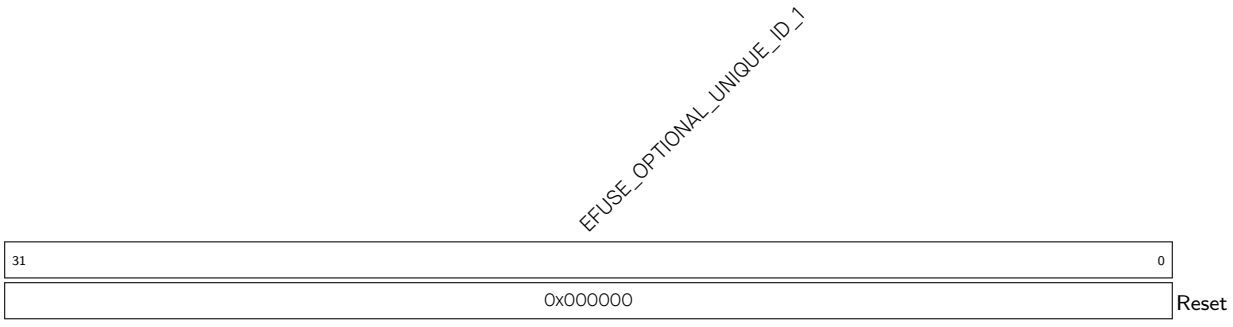
EFUSE_SYS_DATA_PART0_2 表示系统数据第 0 部分的第 2 个 32 位内容。(RO)

Register 5.24. EFUSE_RD_SYS_PART1_DATA0_REG (0x005C)



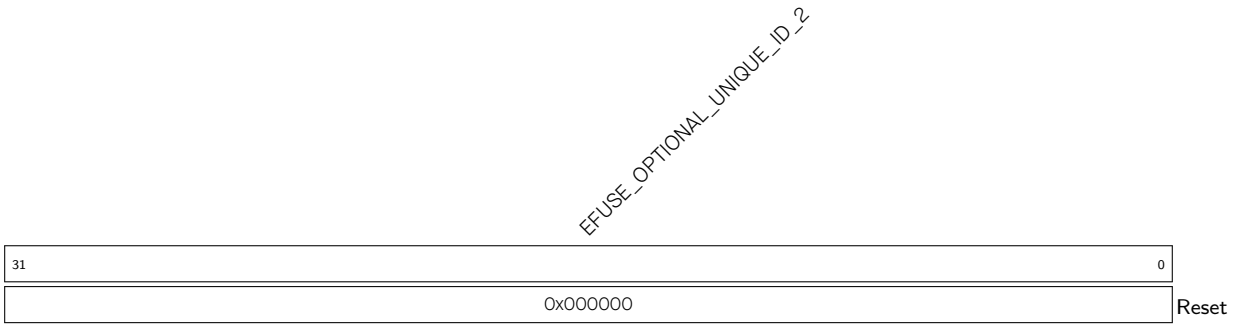
EFUSE_OPTIONAL_UNIQUE_ID_0 表示 OPTIONAL UNIQUE ID 信息 0 至 31 比特数据。(RO)

Register 5.25. EFUSE_RD_SYS_PART1_DATA1_REG (0x0060)



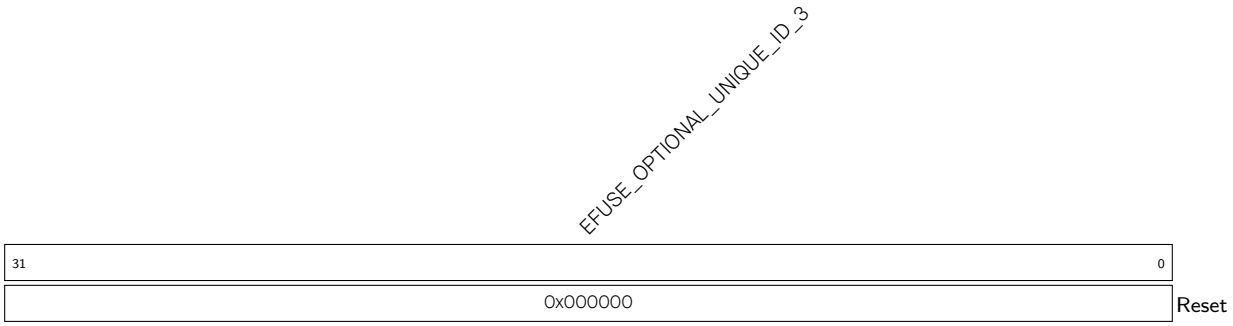
EFUSE_OPTIONAL_UNIQUE_ID_1 表示 OPTIONAL UNIQUE ID 信息 32 至 63 比特数据。(RO)

Register 5.26. EFUSE_RD_SYS_PART1_DATA2_REG (0x0064)



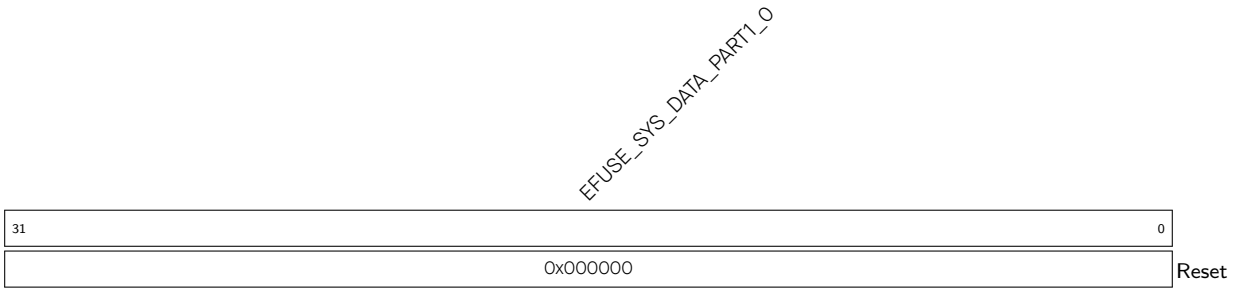
EFUSE_OPTIONAL_UNIQUE_ID_2 表示 OPTIONAL UNIQUE ID 信息 64 至 95 比特数据。(RO)

Register 5.27. EFUSE_RD_SYS_PART1_DATA3_REG (0x0068)



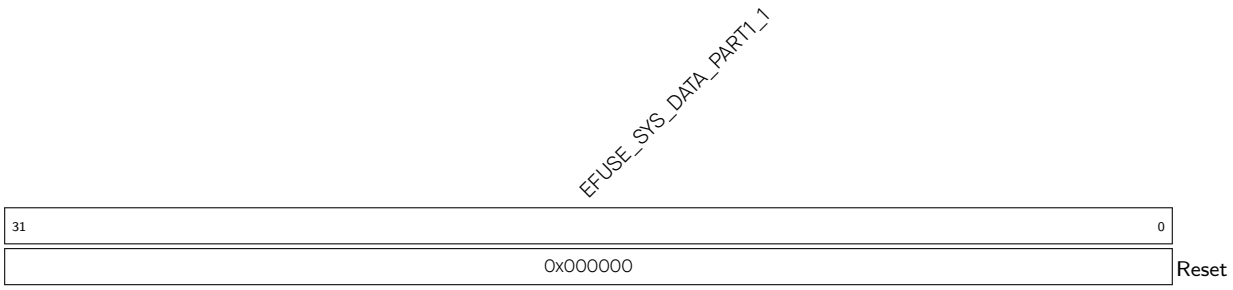
EFUSE_OPTIONAL_UNIQUE_ID_3 表示 OPTIONAL UNIQUE ID 信息 96 至 127 比特数据。(RO)

Register 5.28. EFUSE_RD_SYS_PART1_DATA4_REG (0x006C)



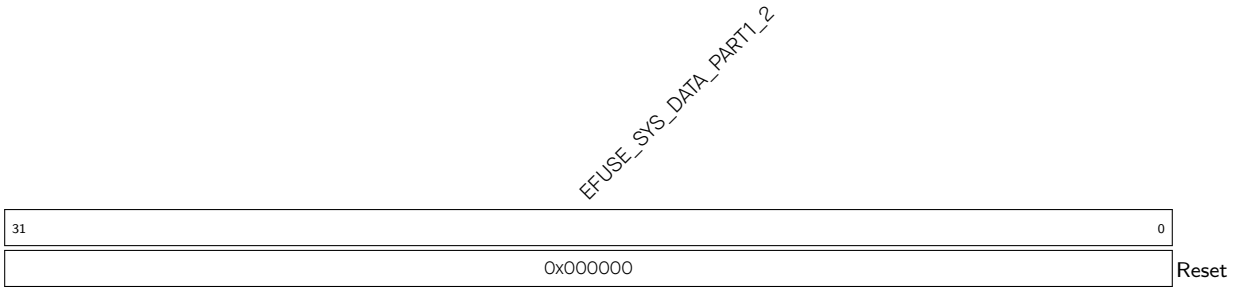
EFUSE_SYS_DATA_PART1_0 表示系统数据第 1 部分的第 0 个 32 位内容。(RO)

Register 5.29. EFUSE_RD_SYS_PART1_DATA5_REG (0x0070)



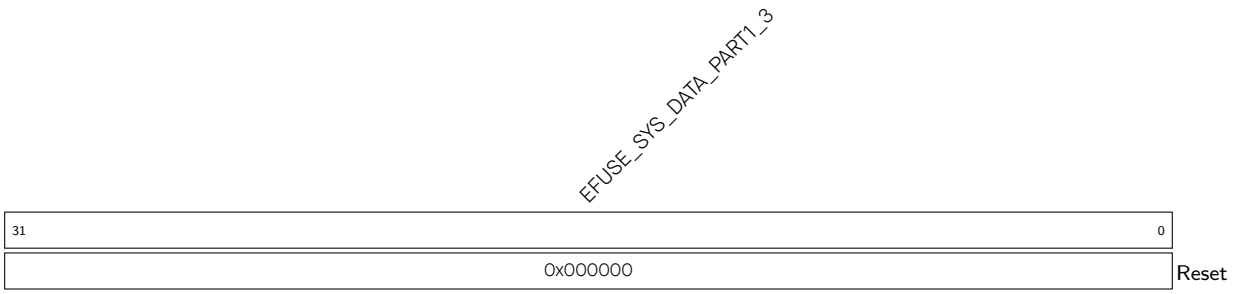
EFUSE_SYS_DATA_PART1_1 表示系统数据第 1 部分的第 1 个 32 位内容。(RO)

Register 5.30. EFUSE_RD_SYS_PART1_DATA6_REG (0x0074)



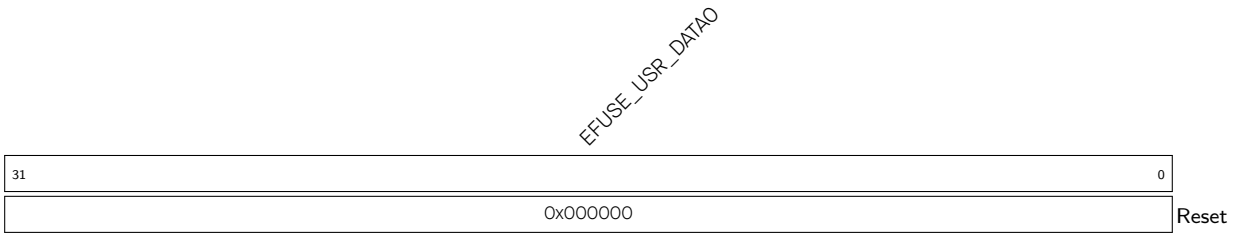
EFUSE_SYS_DATA_PART1_2 表示系统数据第 1 部分的第 2 个 32 位内容。(RO)

Register 5.31. EFUSE_RD_SYS_PART1_DATA7_REG (0x0078)



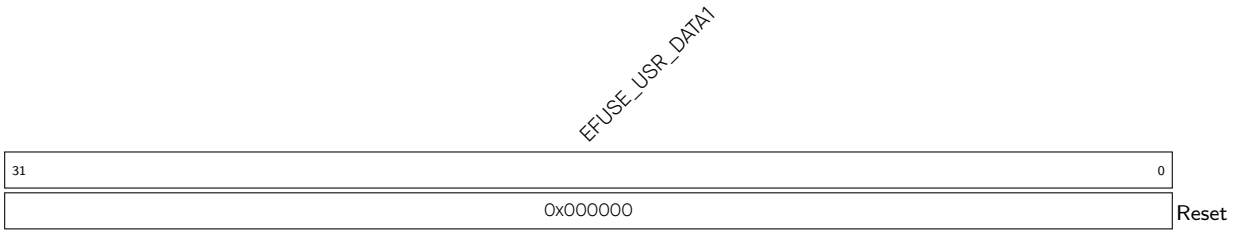
EFUSE_SYS_DATA_PART1_3 表示系统数据第 1 部分的第 3 个 32 位内容。(RO)

Register 5.32. EFUSE_RD_USR_DATA0_REG (0x007C)



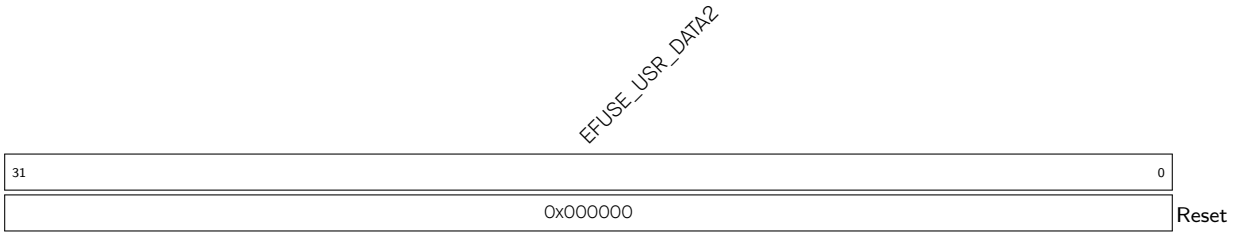
EFUSE_USR_DATA0 表示 BLOCK3 (user) 第 0 个 32 位内容。(RO)

Register 5.33. EFUSE_RD_USR_DATA1_REG (0x0080)



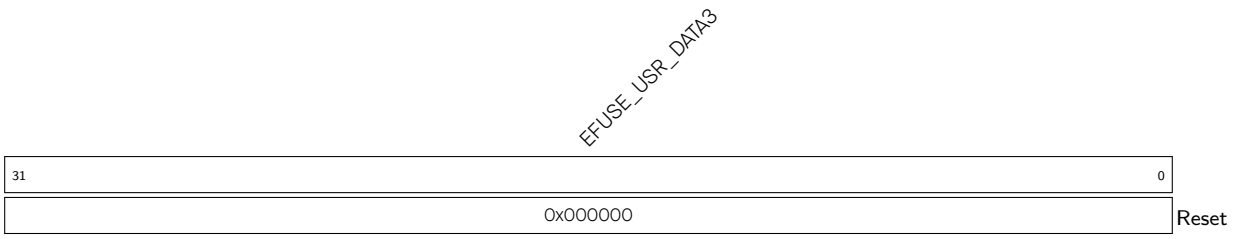
EFUSE_USR_DATA1 表示 BLOCK3 (user) 第 1 个 32 位内容。(RO)

Register 5.34. EFUSE_RD_USR_DATA2_REG (0x0084)



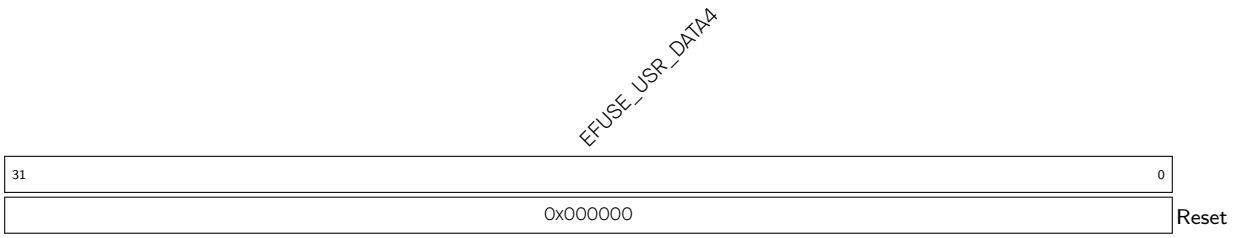
EFUSE_USR_DATA2 表示 BLOCK3 (user) 第 2 个 32 位内容。(RO)

Register 5.35. EFUSE_RD_USR_DATA3_REG (0x0088)



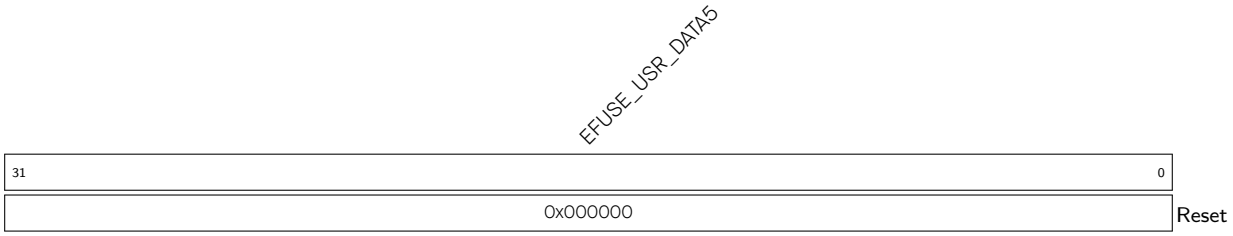
EFUSE_USR_DATA3 表示 BLOCK3 (user) 第 3 个 32 位内容。(RO)

Register 5.36. EFUSE_RD_USR_DATA4_REG (0x008C)



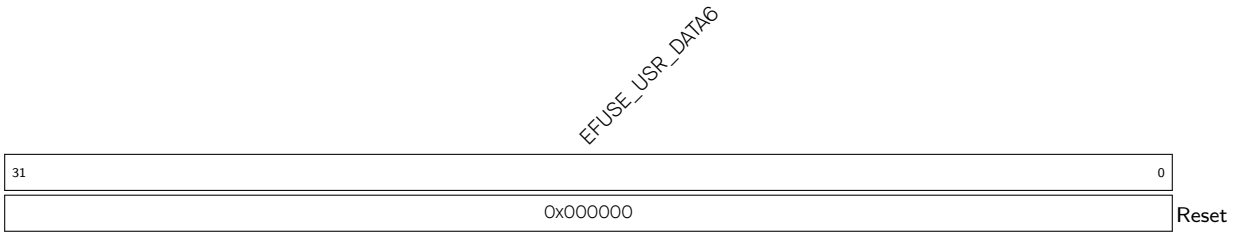
EFUSE_USR_DATA4 表示 BLOCK3 (user) 第 4 个 32 位内容。(RO)

Register 5.37. EFUSE_RD_USR_DATA5_REG (0x0090)



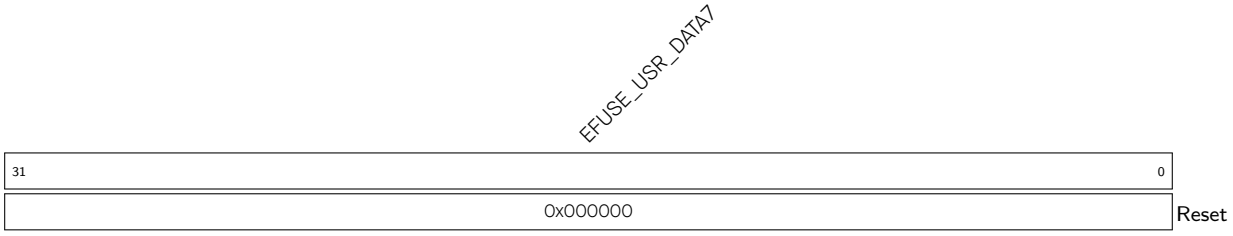
EFUSE_USR_DATA5 表示 BLOCK3 (user) 第 5 个 32 位内容。(RO)

Register 5.38. EFUSE_RD_USR_DATA6_REG (0x0094)



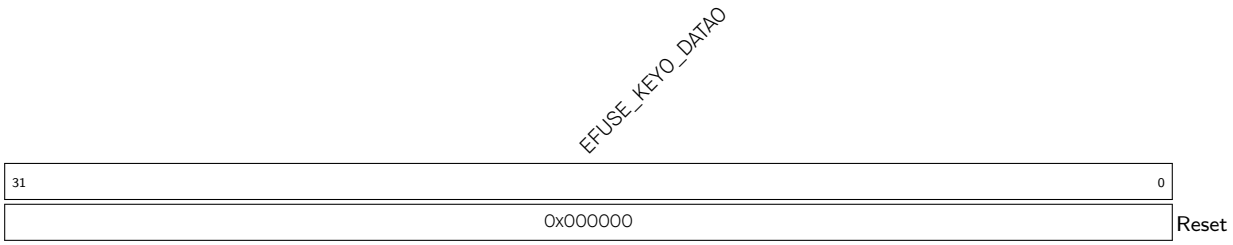
EFUSE_USR_DATA6 表示 BLOCK3 (user) 第 6 个 32 位内容。(RO)

Register 5.39. EFUSE_RD_USR_DATA7_REG (0x0098)



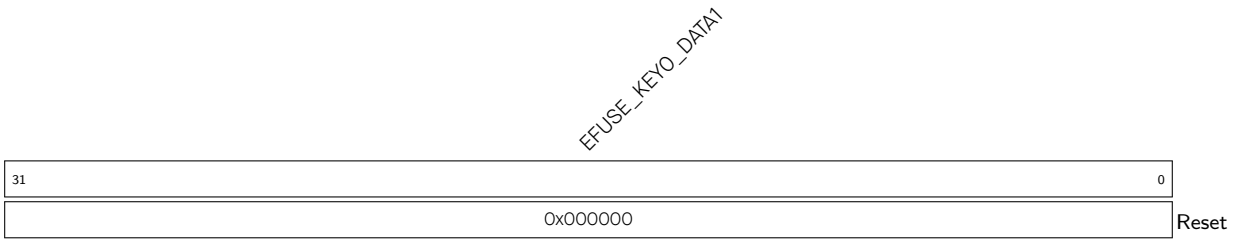
EFUSE_USR_DATA7 表示 BLOCK3 (user) 第 7 个 32 位内容。(RO)

Register 5.40. EFUSE_RD_KEY0_DATA0_REG (0x009C)



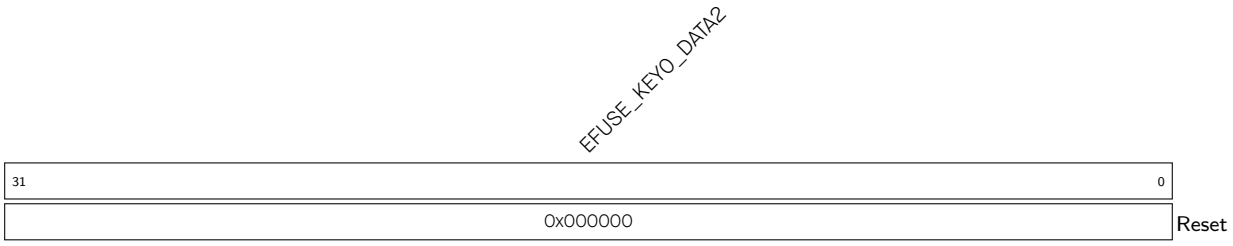
EFUSE_KEY0_DATA0 表示 KEY0 第 0 个 32 位内容。(RO)

Register 5.41. EFUSE_RD_KEY0_DATA1_REG (0x00A0)



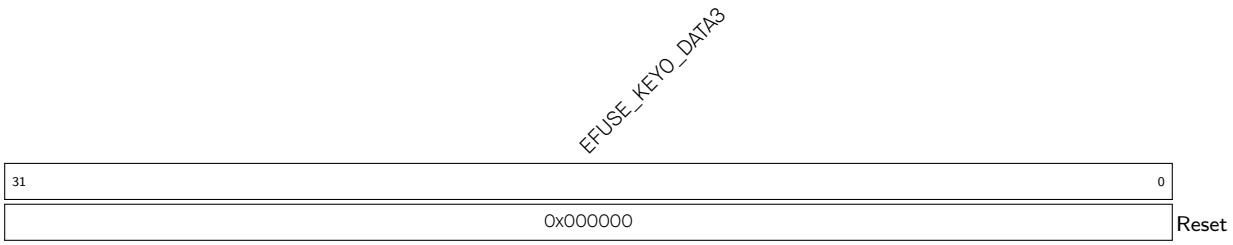
EFUSE_KEY0_DATA1 表示 KEY0 第 1 个 32 位内容。(RO)

Register 5.42. EFUSE_RD_KEY0_DATA2_REG (0x00A4)



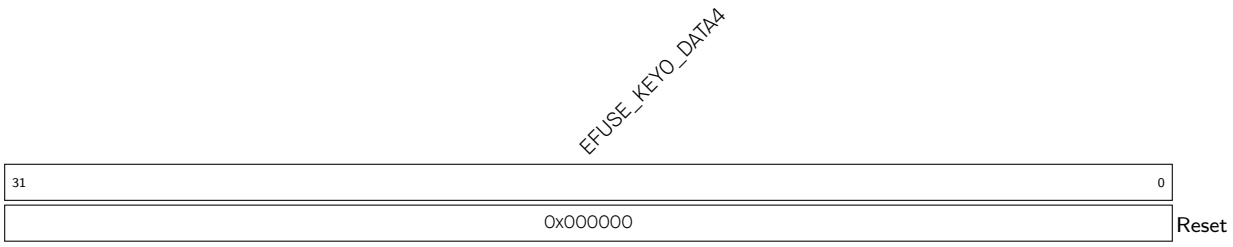
EFUSE_KEY0_DATA2 表示 KEY0 第 2 个 32 位内容。(RO)

Register 5.43. EFUSE_RD_KEY0_DATA3_REG (0x00A8)



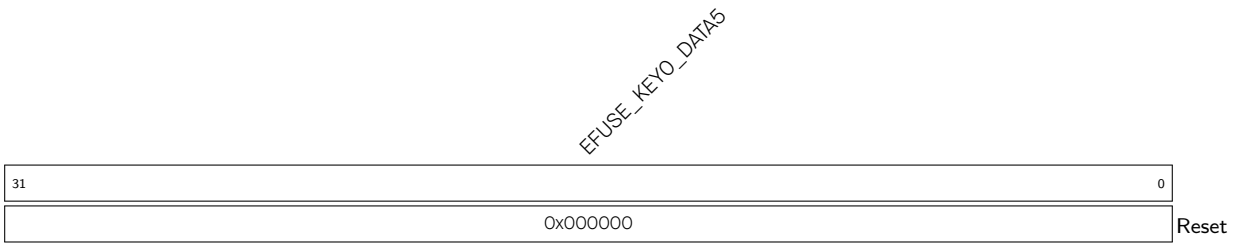
EFUSE_KEY0_DATA3 表示 KEY0 第 3 个 32 位内容。(RO)

Register 5.44. EFUSE_RD_KEY0_DATA4_REG (0x00AC)



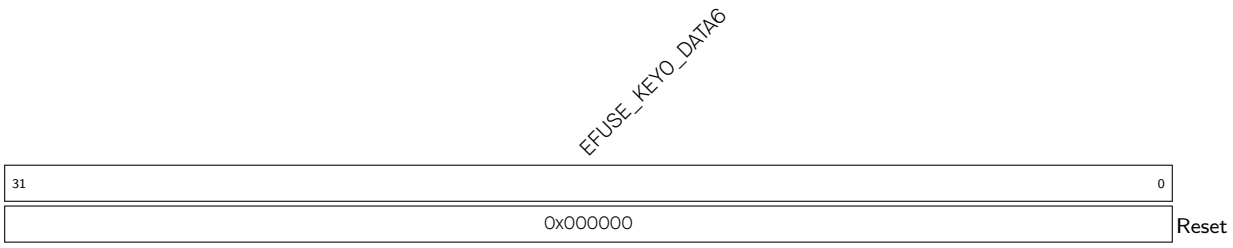
EFUSE_KEY0_DATA4 表示 KEY0 第 4 个 32 位内容。(RO)

Register 5.45. EFUSE_RD_KEY0_DATA5_REG (0x00B0)



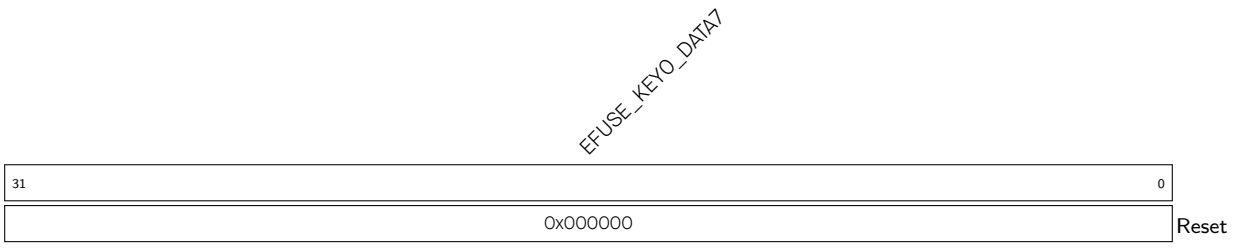
EFUSE_KEY0_DATA5 表示 KEY0 第 5 个 32 位内容。(RO)

Register 5.46. EFUSE_RD_KEY0_DATA6_REG (0x00B4)



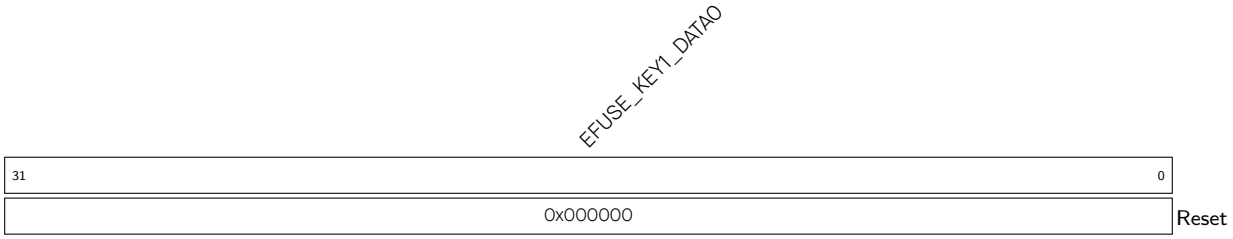
EFUSE_KEY0_DATA6 表示 KEY0 第 6 个 32 位内容。(RO)

Register 5.47. EFUSE_RD_KEY0_DATA7_REG (0x00B8)



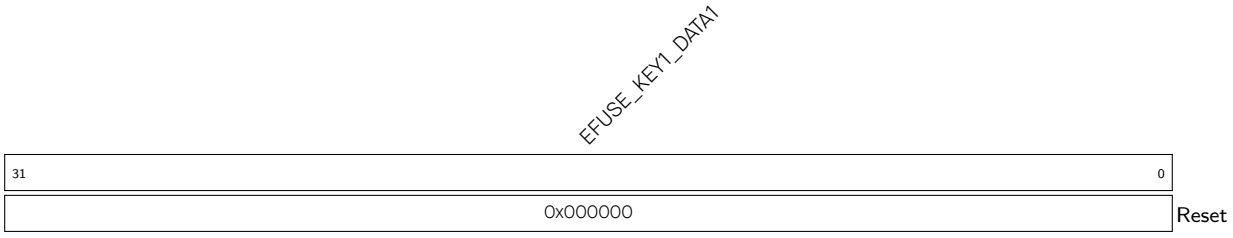
EFUSE_KEY0_DATA7 表示 KEY0 第 7 个 32 位内容。(RO)

Register 5.48. EFUSE_RD_KEY1_DATA0_REG (0x00BC)



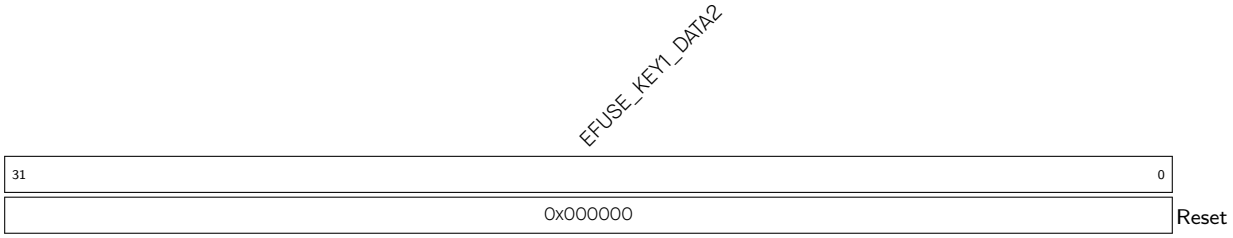
EFUSE_KEY1_DATA0 表示 KEY1 第 0 个 32 位内容。(RO)

Register 5.49. EFUSE_RD_KEY1_DATA1_REG (0x00C0)



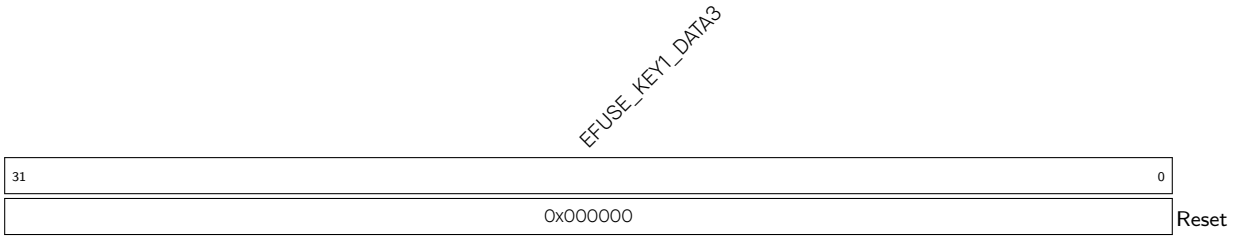
EFUSE_KEY1_DATA1 表示 KEY1 第 1 个 32 位内容。(RO)

Register 5.50. EFUSE_RD_KEY1_DATA2_REG (0x00C4)



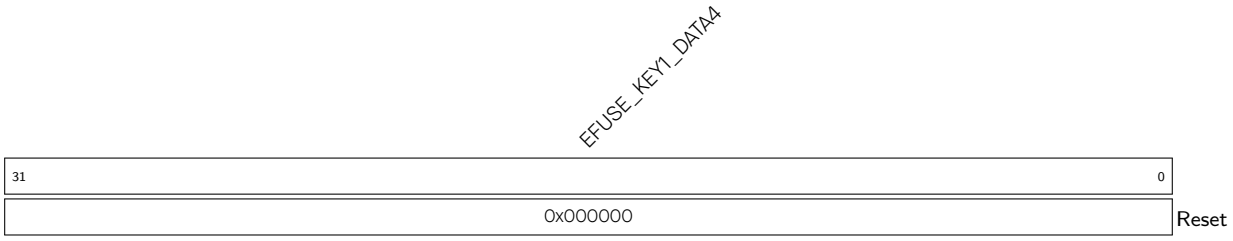
EFUSE_KEY1_DATA2 表示 KEY1 第 2 个 32 位内容。(RO)

Register 5.51. EFUSE_RD_KEY1_DATA3_REG (0x00C8)



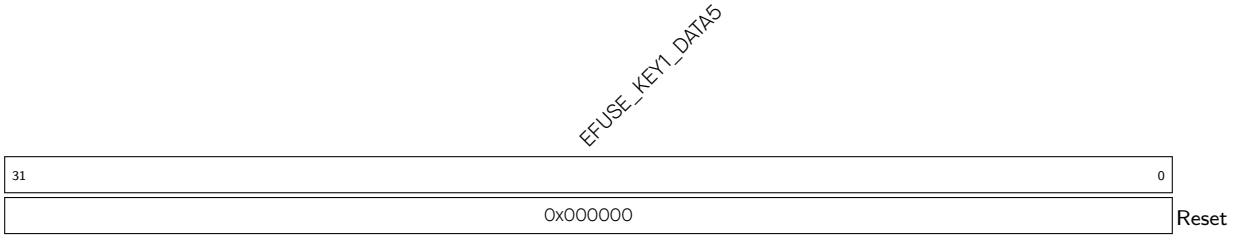
EFUSE_KEY1_DATA3 表示 KEY1 第 3 个 32 位内容。(RO)

Register 5.52. EFUSE_RD_KEY1_DATA4_REG (0x00CC)



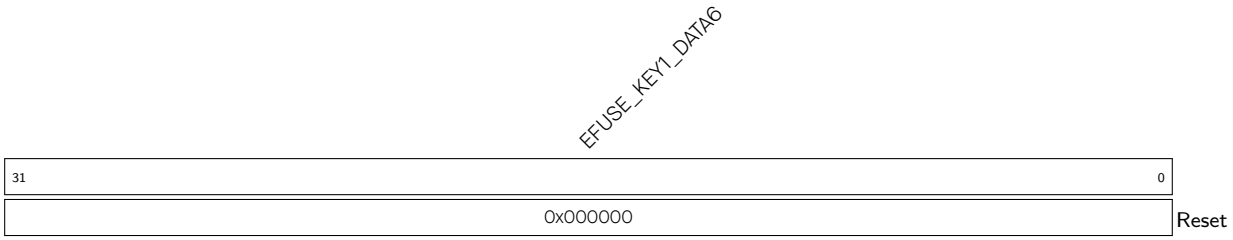
EFUSE_KEY1_DATA4 表示 KEY1 第 4 个 32 位内容。(RO)

Register 5.53. EFUSE_RD_KEY1_DATA5_REG (0x00D0)



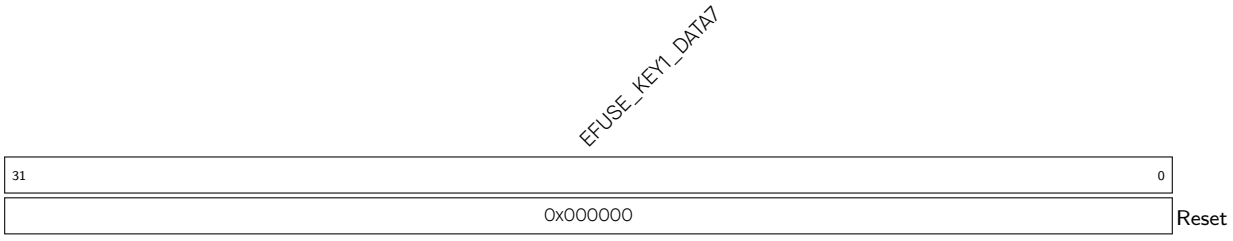
EFUSE_KEY1_DATA5 表示 KEY1 第 5 个 32 位内容。(RO)

Register 5.54. EFUSE_RD_KEY1_DATA6_REG (0x00D4)



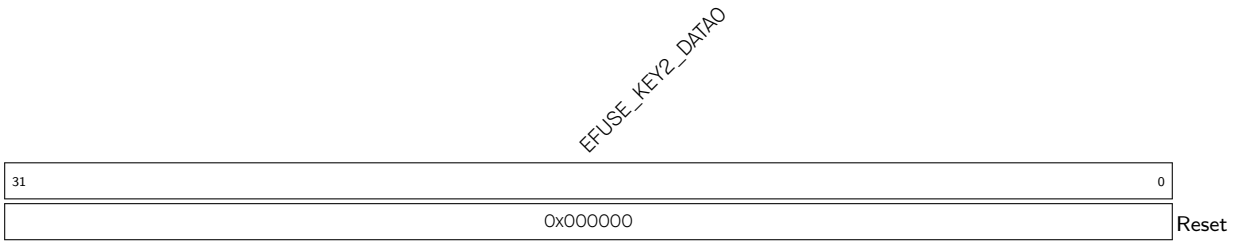
EFUSE_KEY1_DATA6 表示 KEY1 第 6 个 32 位内容。(RO)

Register 5.55. EFUSE_RD_KEY1_DATA7_REG (0x00D8)



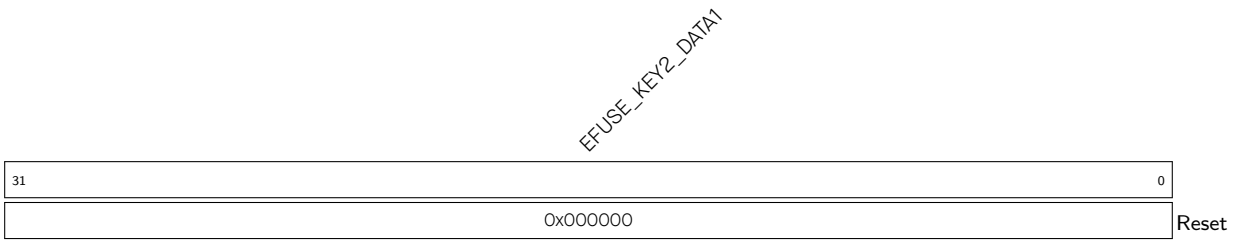
EFUSE_KEY1_DATA7 表示 KEY1 第 7 个 32 位内容。(RO)

Register 5.56. EFUSE_RD_KEY2_DATA0_REG (0x00DC)



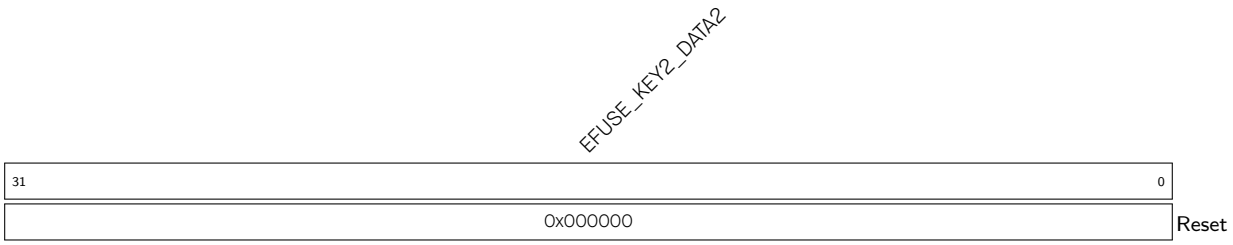
EFUSE_KEY2_DATA0 表示 KEY2 第 0 个 32 位内容。(RO)

Register 5.57. EFUSE_RD_KEY2_DATA1_REG (0x00E0)



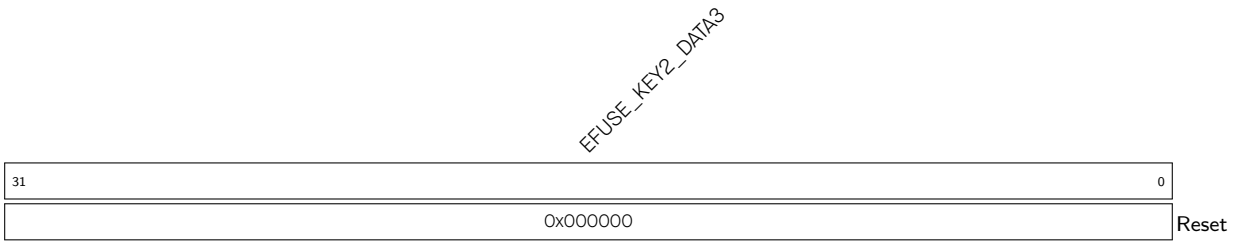
EFUSE_KEY2_DATA1 表示 KEY2 第 1 个 32 位内容。(RO)

Register 5.58. EFUSE_RD_KEY2_DATA2_REG (0x00E4)



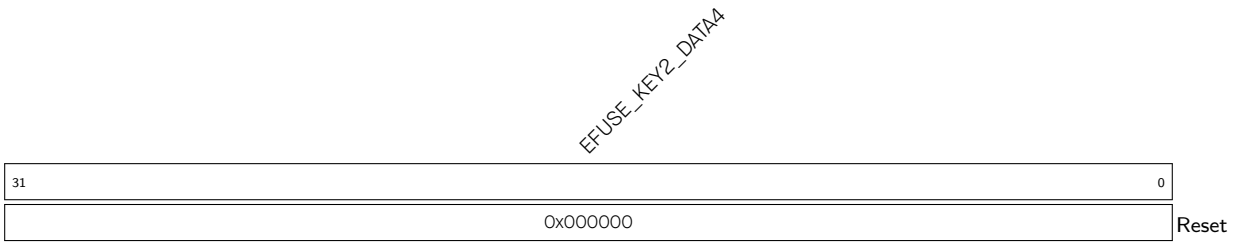
EFUSE_KEY2_DATA2 表示 KEY2 第 2 个 32 位内容。(RO)

Register 5.59. EFUSE_RD_KEY2_DATA3_REG (0x00E8)



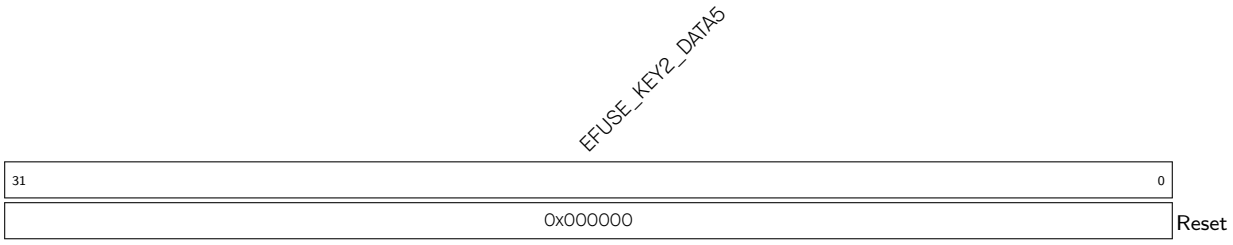
EFUSE_KEY2_DATA3 表示 KEY2 第 3 个 32 位内容。(RO)

Register 5.60. EFUSE_RD_KEY2_DATA4_REG (0x00EC)



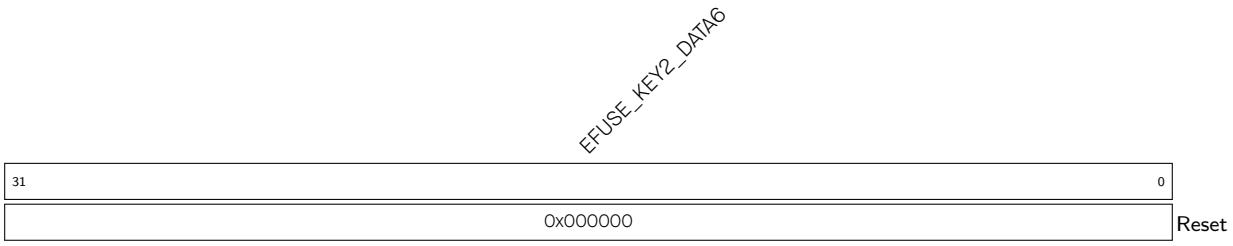
EFUSE_KEY2_DATA4 表示 KEY2 第 4 个 32 位内容。(RO)

Register 5.61. EFUSE_RD_KEY2_DATA5_REG (0x00F0)



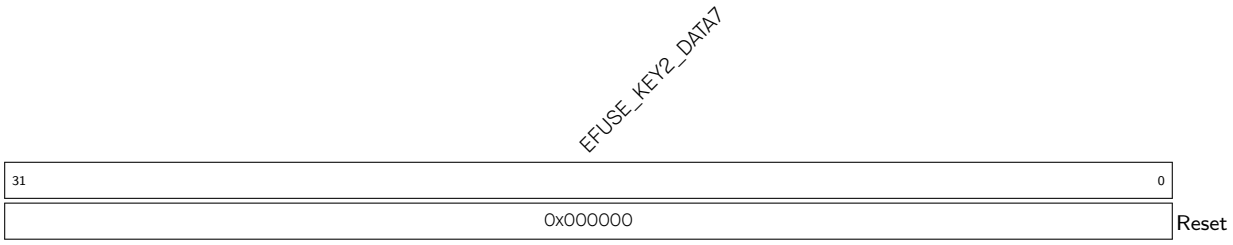
EFUSE_KEY2_DATA5 表示 KEY2 第 5 个 32 位内容。(RO)

Register 5.62. EFUSE_RD_KEY2_DATA6_REG (0x00F4)



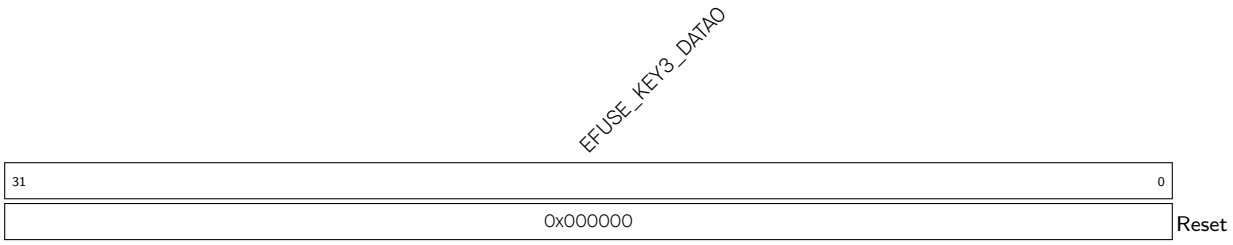
EFUSE_KEY2_DATA6 表示 KEY2 第 6 个 32 位内容。(RO)

Register 5.63. EFUSE_RD_KEY2_DATA7_REG (0x00F8)



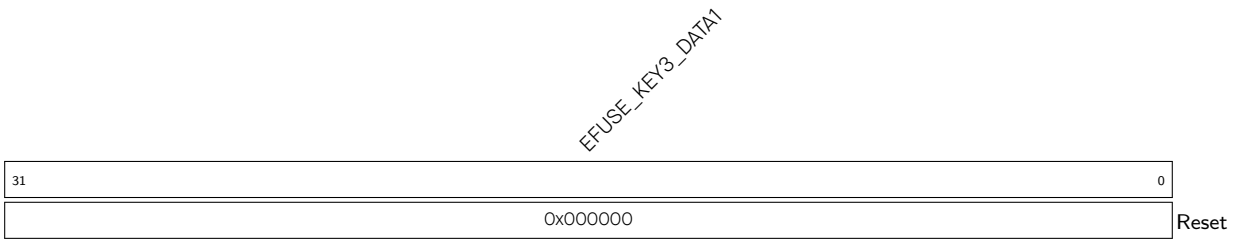
EFUSE_KEY2_DATA7 表示 KEY2 第 7 个 32 位内容。(RO)

Register 5.64. EFUSE_RD_KEY3_DATA0_REG (0x00FC)



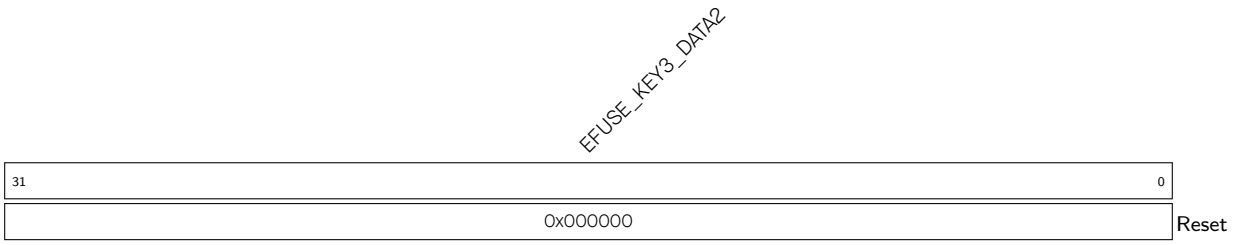
EFUSE_KEY3_DATA0 表示 KEY3 第 0 个 32 位内容。(RO)

Register 5.65. EFUSE_RD_KEY3_DATA1_REG (0x0100)



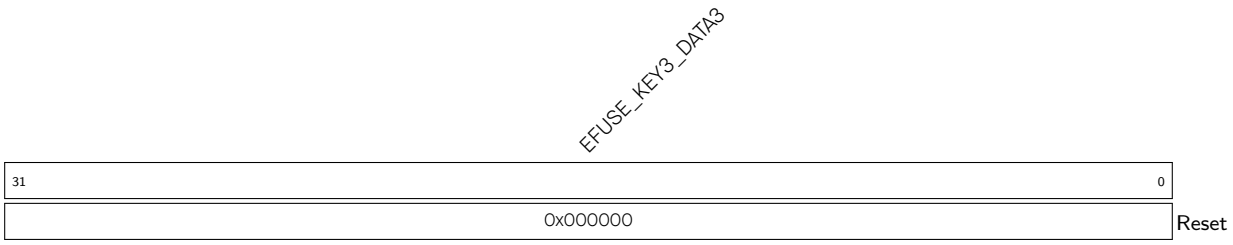
EFUSE_KEY3_DATA1 表示 KEY3 第 1 个 32 位内容。(RO)

Register 5.66. EFUSE_RD_KEY3_DATA2_REG (0x0104)



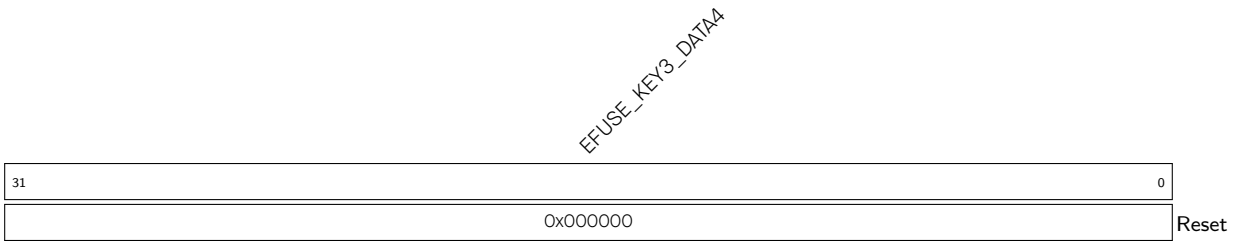
EFUSE_KEY3_DATA2 表示 KEY3 第 2 个 32 位内容。(RO)

Register 5.67. EFUSE_RD_KEY3_DATA3_REG (0x0108)



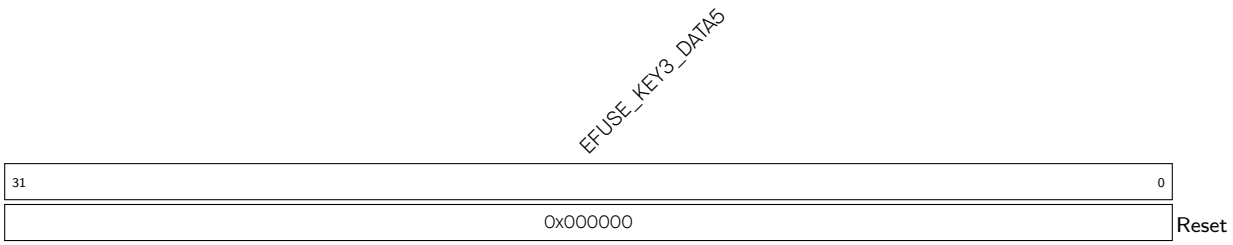
EFUSE_KEY3_DATA3 表示 KEY3 第 3 个 32 位内容。(RO)

Register 5.68. EFUSE_RD_KEY3_DATA4_REG (0x010C)



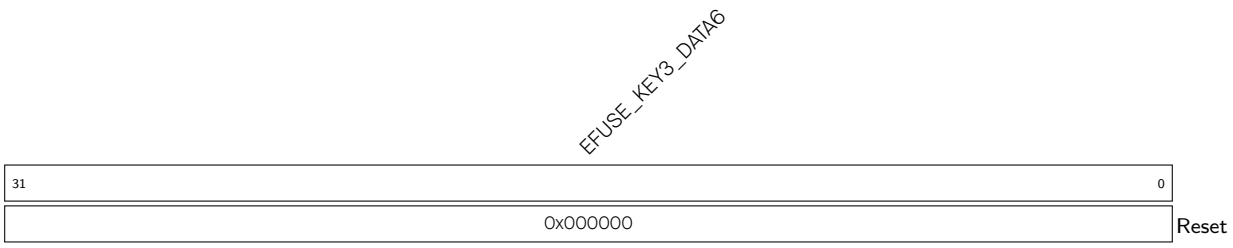
EFUSE_KEY3_DATA4 表示 KEY3 第 4 个 32 位内容。(RO)

Register 5.69. EFUSE_RD_KEY3_DATA5_REG (0x0110)



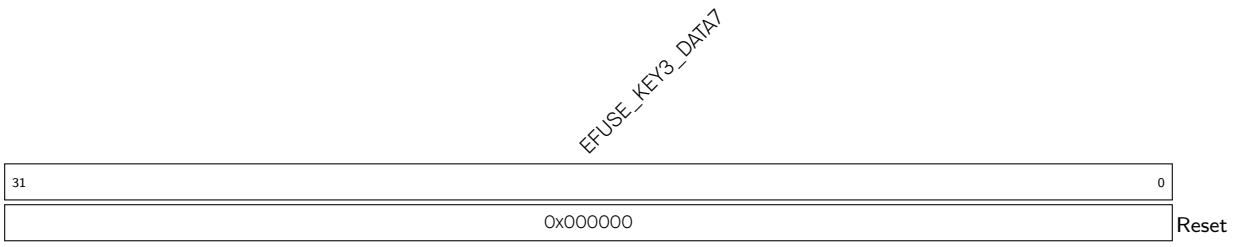
EFUSE_KEY3_DATA5 表示 KEY3 第 5 个 32 位内容。(RO)

Register 5.70. EFUSE_RD_KEY3_DATA6_REG (0x0114)



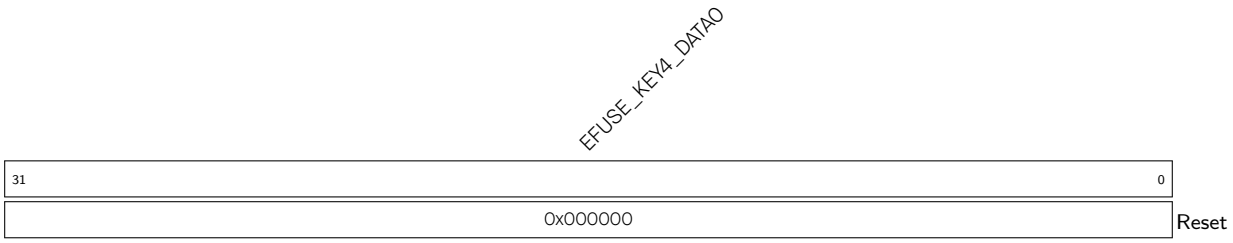
EFUSE_KEY3_DATA6 表示 KEY3 第 6 个 32 位内容。(RO)

Register 5.71. EFUSE_RD_KEY3_DATA7_REG (0x0118)



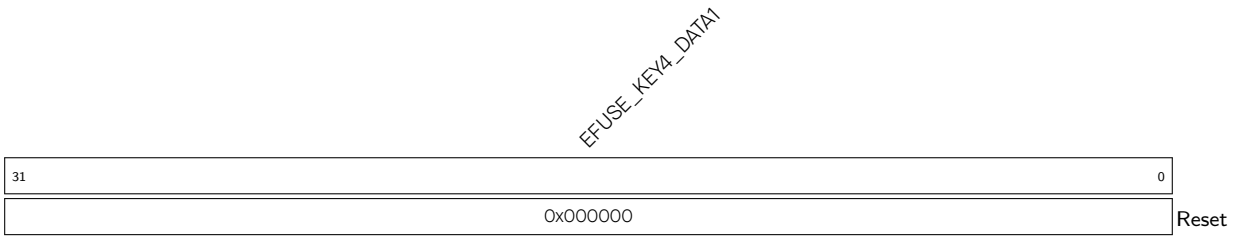
EFUSE_KEY3_DATA7 表示 KEY3 第 7 个 32 位内容。(RO)

Register 5.72. EFUSE_RD_KEY4_DATA0_REG (0x011C)



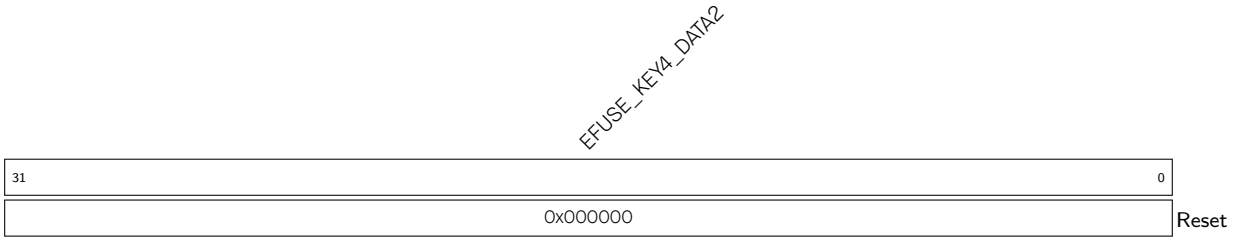
EFUSE_KEY4_DATA0 表示 KEY4 第 0 个 32 位内容。(RO)

Register 5.73. EFUSE_RD_KEY4_DATA1_REG (0x0120)



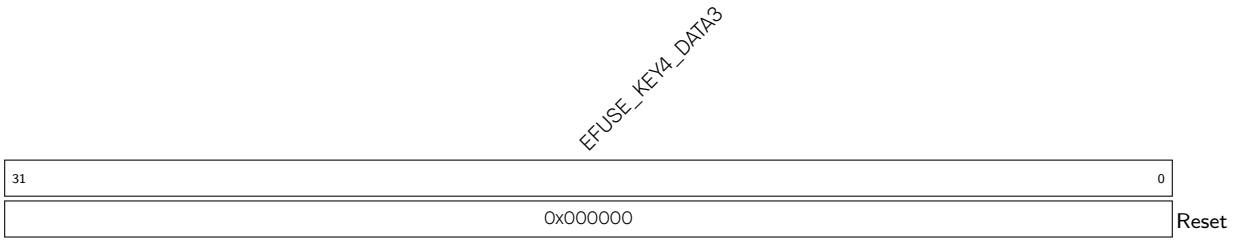
EFUSE_KEY4_DATA1 表示 KEY4 第 1 个 32 位内容。(RO)

Register 5.74. EFUSE_RD_KEY4_DATA2_REG (0x0124)



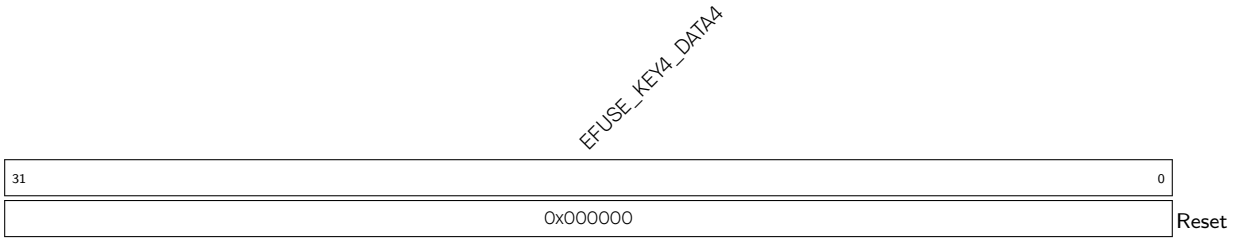
EFUSE_KEY4_DATA2 表示 KEY4 第 2 个 32 位内容。(RO)

Register 5.75. EFUSE_RD_KEY4_DATA3_REG (0x0128)



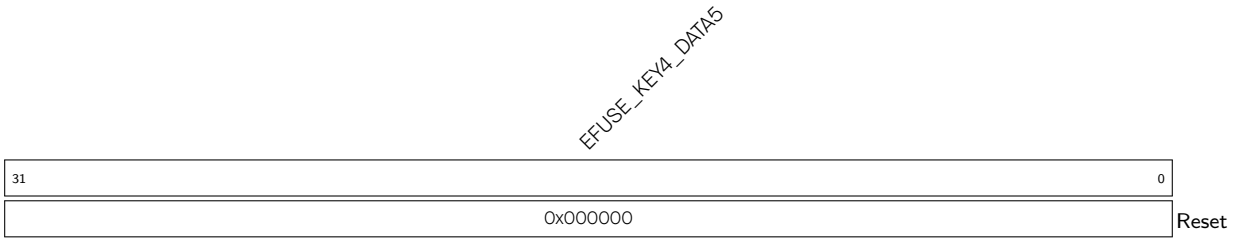
EFUSE_KEY4_DATA3 表示 KEY4 第 3 个 32 位内容。(RO)

Register 5.76. EFUSE_RD_KEY4_DATA4_REG (0x012C)



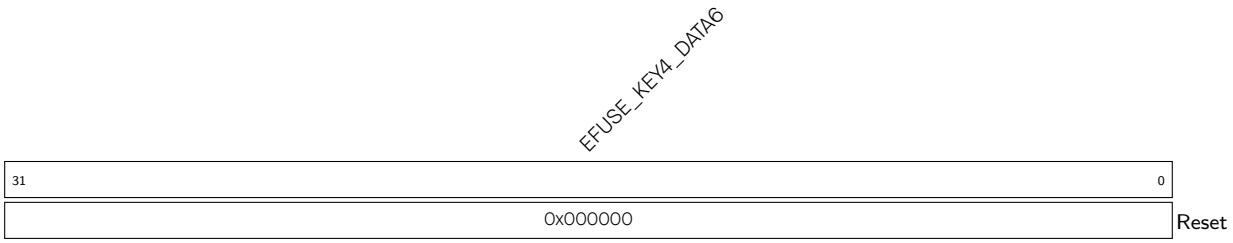
EFUSE_KEY4_DATA4 表示 KEY4 第 4 个 32 位内容。(RO)

Register 5.77. EFUSE_RD_KEY4_DATA5_REG (0x0130)



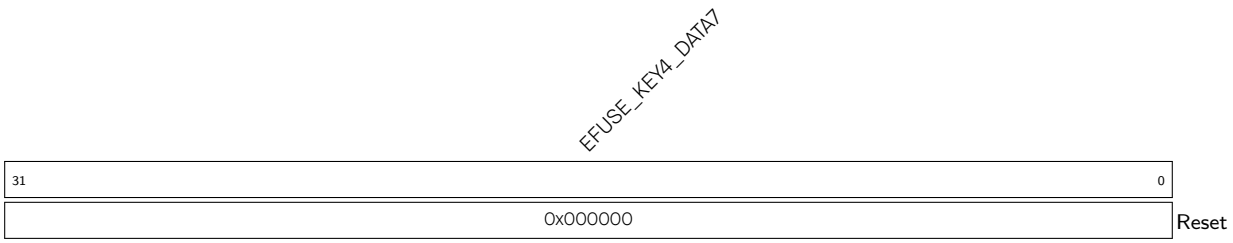
EFUSE_KEY4_DATA5 表示 KEY4 第 5 个 32 位内容。(RO)

Register 5.78. EFUSE_RD_KEY4_DATA6_REG (0x0134)



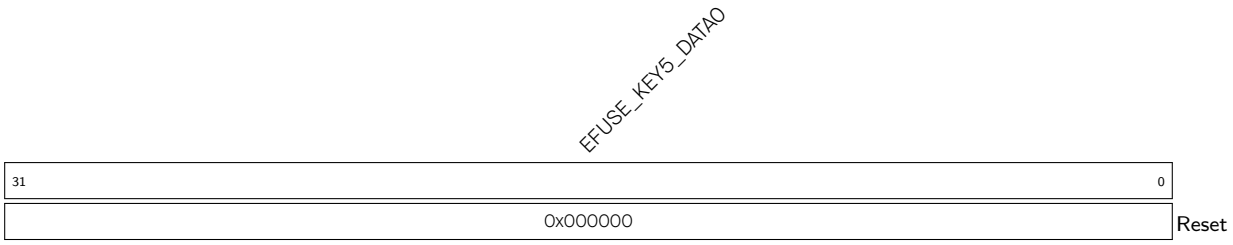
EFUSE_KEY4_DATA6 表示 KEY4 第 6 个 32 位内容。(RO)

Register 5.79. EFUSE_RD_KEY4_DATA7_REG (0x0138)



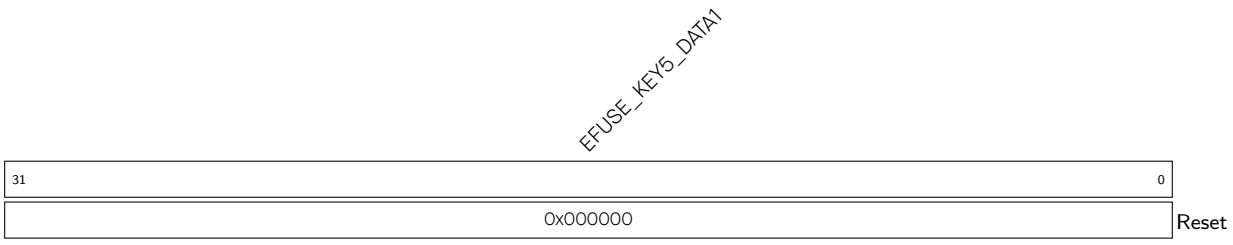
EFUSE_KEY4_DATA7 表示 KEY4 第 7 个 32 位内容。(RO)

Register 5.80. EFUSE_RD_KEY5_DATA0_REG (0x013C)



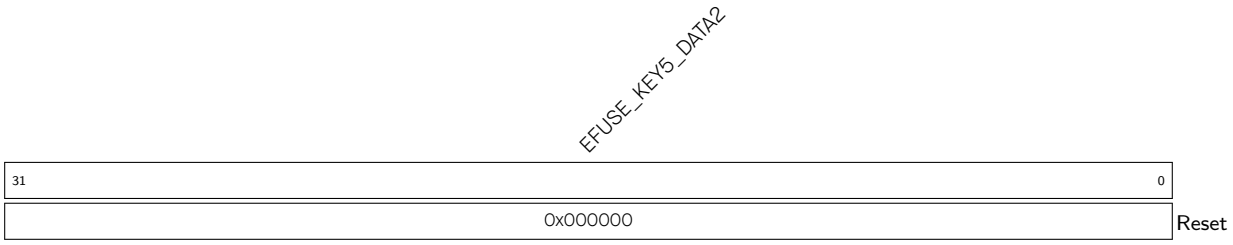
EFUSE_KEY5_DATA0 表示 KEY5 第 0 个 32 位内容。(RO)

Register 5.81. EFUSE_RD_KEY5_DATA1_REG (0x0140)



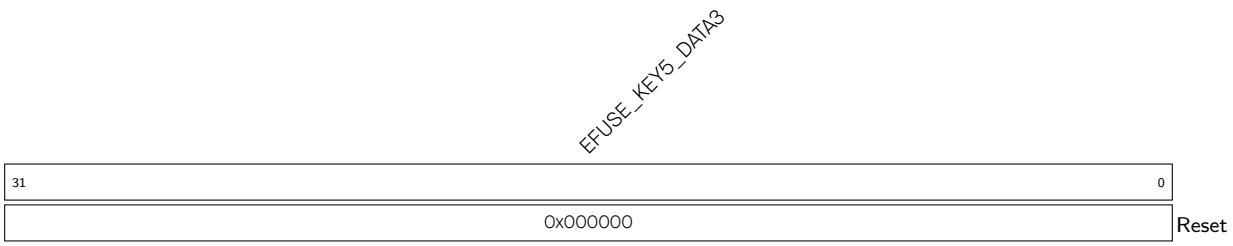
EFUSE_KEY5_DATA1 表示 KEY5 第 1 个 32 位内容。(RO)

Register 5.82. EFUSE_RD_KEY5_DATA2_REG (0x0144)



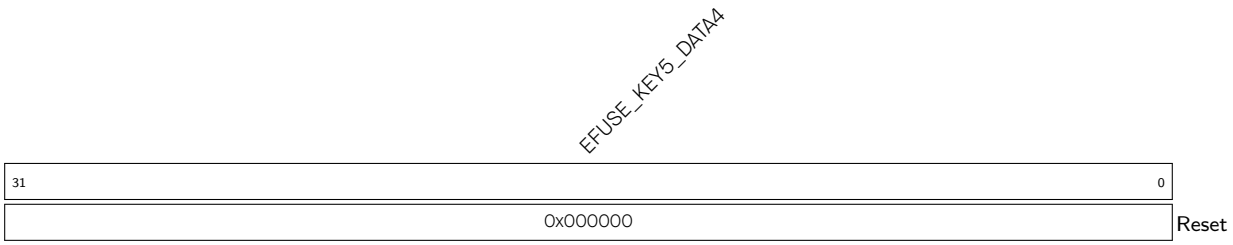
EFUSE_KEY5_DATA2 表示 KEY5 第 2 个 32 位内容。(RO)

Register 5.83. EFUSE_RD_KEY5_DATA3_REG (0x0148)



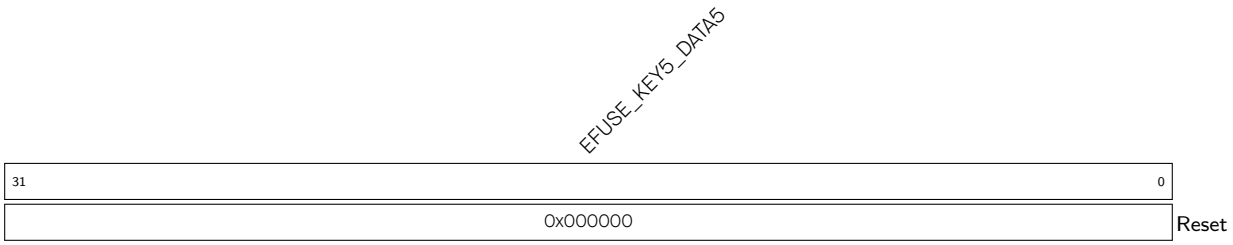
EFUSE_KEY5_DATA3 表示 KEY5 第 3 个 32 位内容。(RO)

Register 5.84. EFUSE_RD_KEY5_DATA4_REG (0x014C)



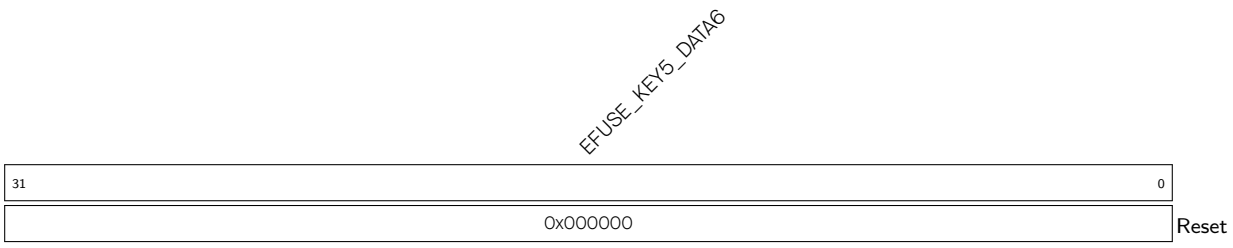
EFUSE_KEY5_DATA4 表示 KEY5 第 4 个 32 位内容。(RO)

Register 5.85. EFUSE_RD_KEY5_DATA5_REG (0x0150)



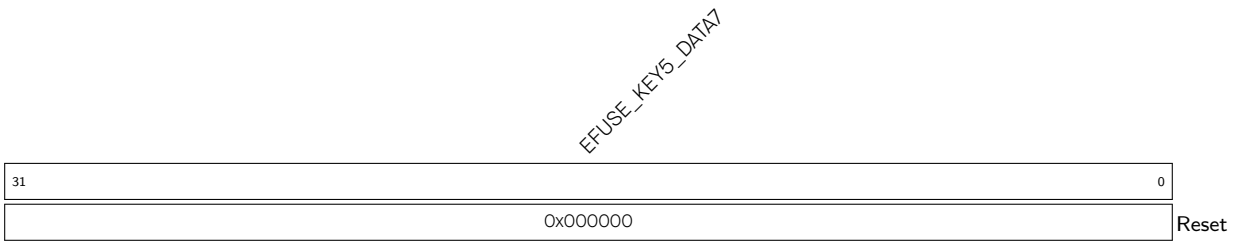
EFUSE_KEY5_DATA5 表示 KEY5 第 5 个 32 位内容。(RO)

Register 5.86. EFUSE_RD_KEY5_DATA6_REG (0x0154)



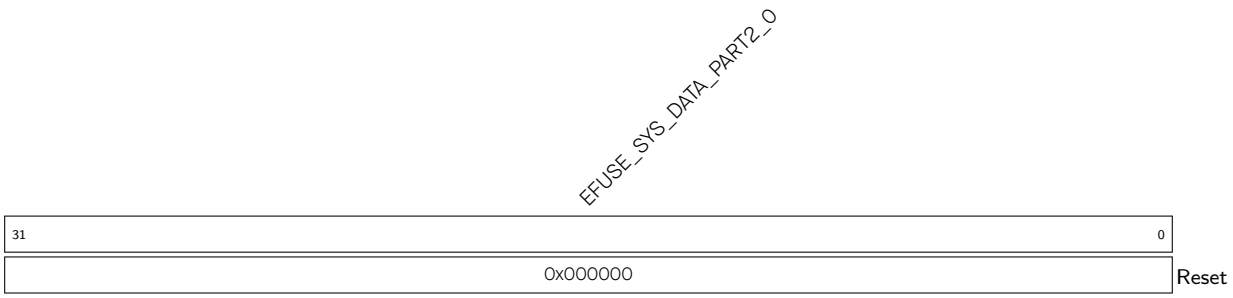
EFUSE_KEY5_DATA6 表示 KEY5 第 6 个 32 位内容。(RO)

Register 5.87. EFUSE_RD_KEY5_DATA7_REG (0x0158)



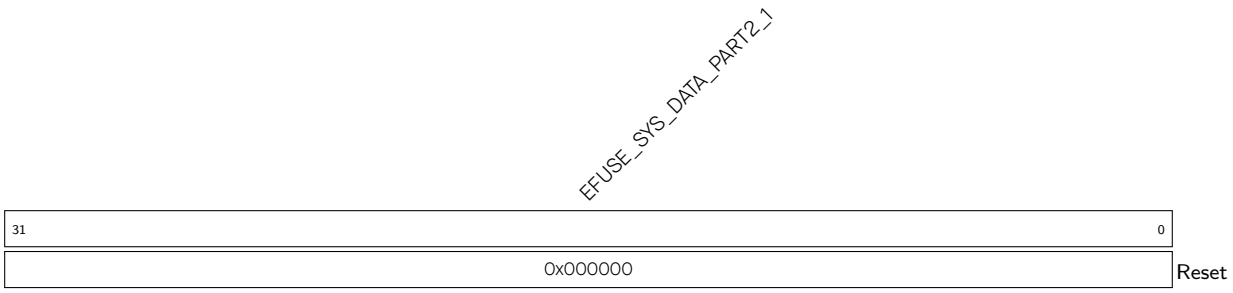
EFUSE_KEY5_DATA7 表示 KEY5 第 7 个 32 位内容。(RO)

Register 5.88. EFUSE_RD_SYS_PART2_DATA0_REG (0x015C)



EFUSE_SYS_DATA_PART2_0 表示系统数据第 2 部分的第 0 个 32 位内容。(RO)

Register 5.89. EFUSE_RD_SYS_PART2_DATA1_REG (0x0160)



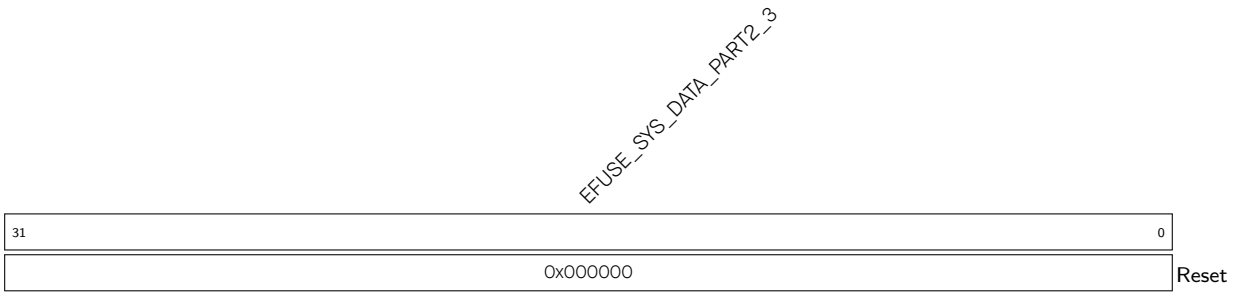
EFUSE_SYS_DATA_PART2_1 表示系统数据第 2 部分的第 1 个 32 位内容。(RO)

Register 5.90. EFUSE_RD_SYS_PART2_DATA2_REG (0x0164)



EFUSE_SYS_DATA_PART2_2 表示系统数据第 2 部分的第 2 个 32 位内容。(RO)

Register 5.91. EFUSE_RD_SYS_PART2_DATA3_REG (0x0168)



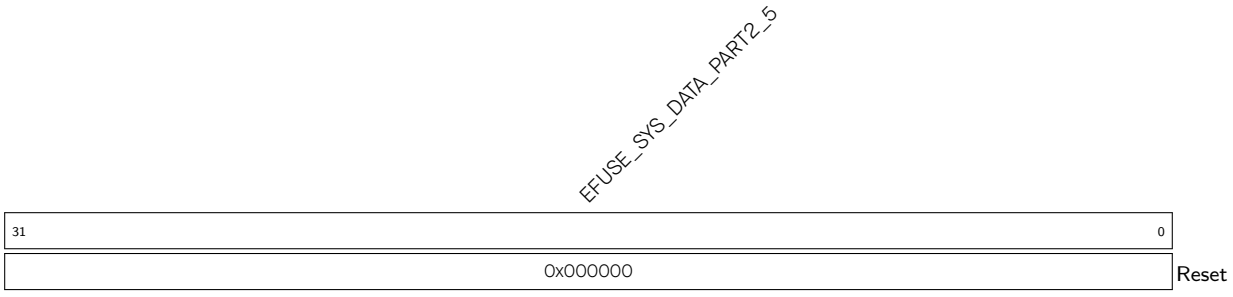
EFUSE_SYS_DATA_PART2_3 表示系统数据第 2 部分的第 3 个 32 位内容。(RO)

Register 5.92. EFUSE_RD_SYS_PART2_DATA4_REG (0x016C)



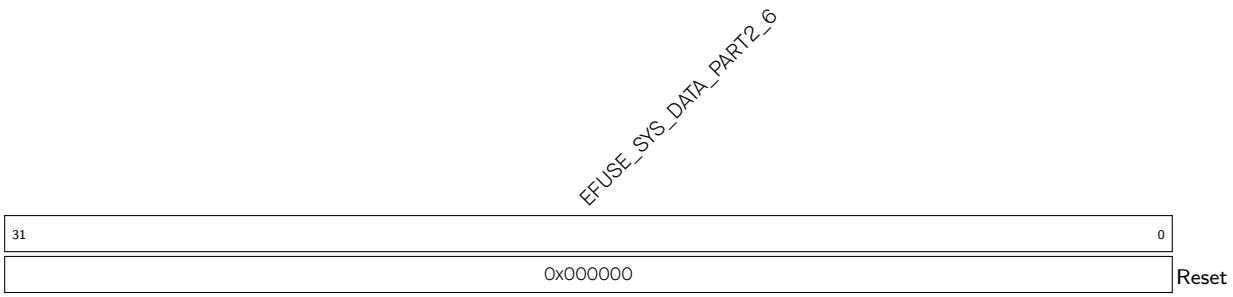
EFUSE_SYS_DATA_PART2_4 表示系统数据第 2 部分的第 4 个 32 位内容。(RO)

Register 5.93. EFUSE_RD_SYS_PART2_DATA5_REG (0x0170)



EFUSE_SYS_DATA_PART2_5 表示系统数据第 2 部分的第 5 个 32 位内容。(RO)

Register 5.94. EFUSE_RD_SYS_PART2_DATA6_REG (0x0174)



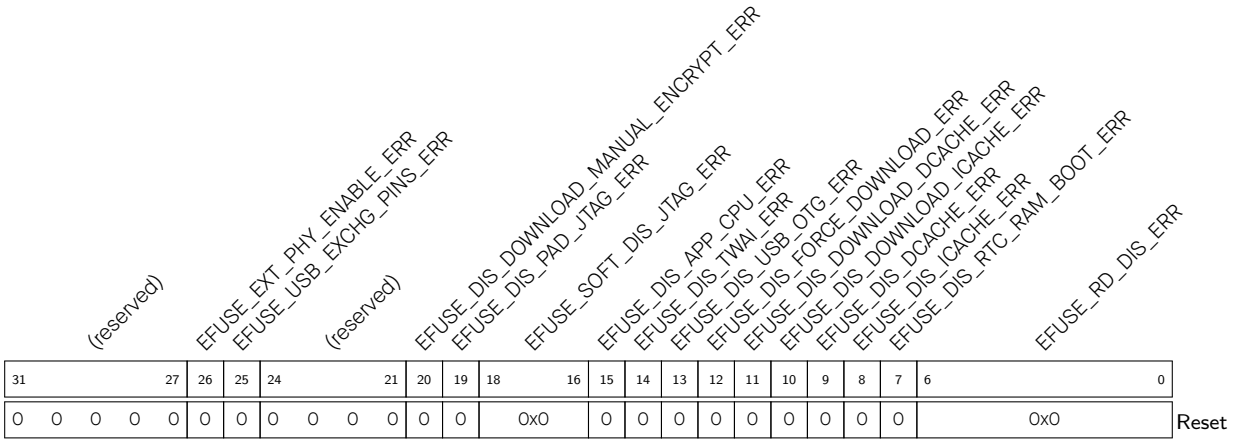
EFUSE_SYS_DATA_PART2_6 表示系统数据第 2 部分的第 6 个 32 位内容。(RO)

Register 5.95. EFUSE_RD_SYS_PART2_DATA7_REG (0x0178)



EFUSE_SYS_DATA_PART2_7 表示系统数据第 2 部分的第 7 个 32 位内容。(RO)

Register 5.96. EFUSE_RD_REPEAT_ERR0_REG (0x017C)



EFUSE_RD_DIS_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_RTC_RAM_BOOT_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_ICACHE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_DCACHE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_DOWNLOAD_ICACHE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_DOWNLOAD_DCACHE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_FORCE_DOWNLOAD_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_USB_OTG_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_TWAI_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_APP_CPU_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_SOFT_DIS_JTAG_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

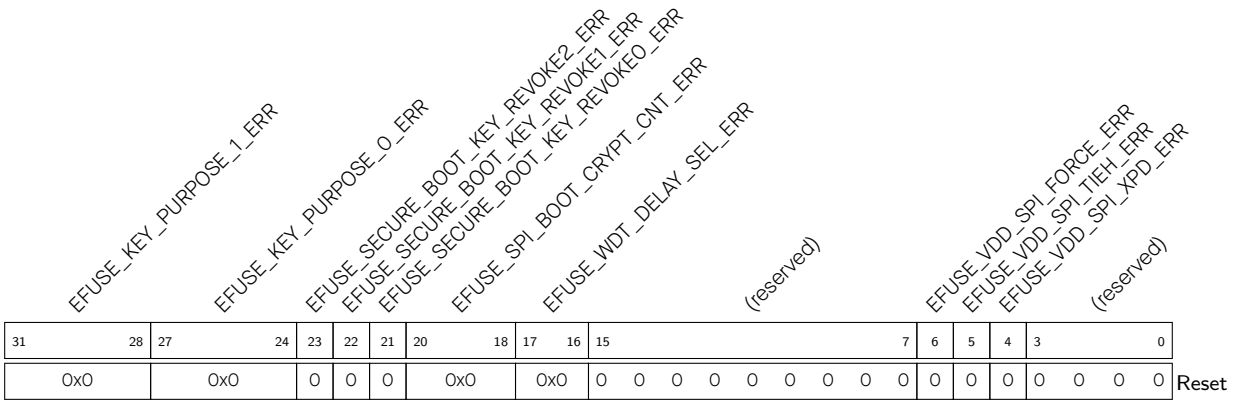
EFUSE_DIS_PAD_JTAG_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_USB_EXCHG_PINS_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_EXT_PHY_ENABLE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

Register 5.97. EFUSE_RD_REPEAT_ERR1_REG (0x0180)



EFUSE_VDD_SPI_XPD_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_VDD_SPI_TIEH_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_VDD_SPI_FORCE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_WDT_DELAY_SEL_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_SPI_BOOT_CRYPT_CNT_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_SECURE_BOOT_KEY_REVOKE0_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

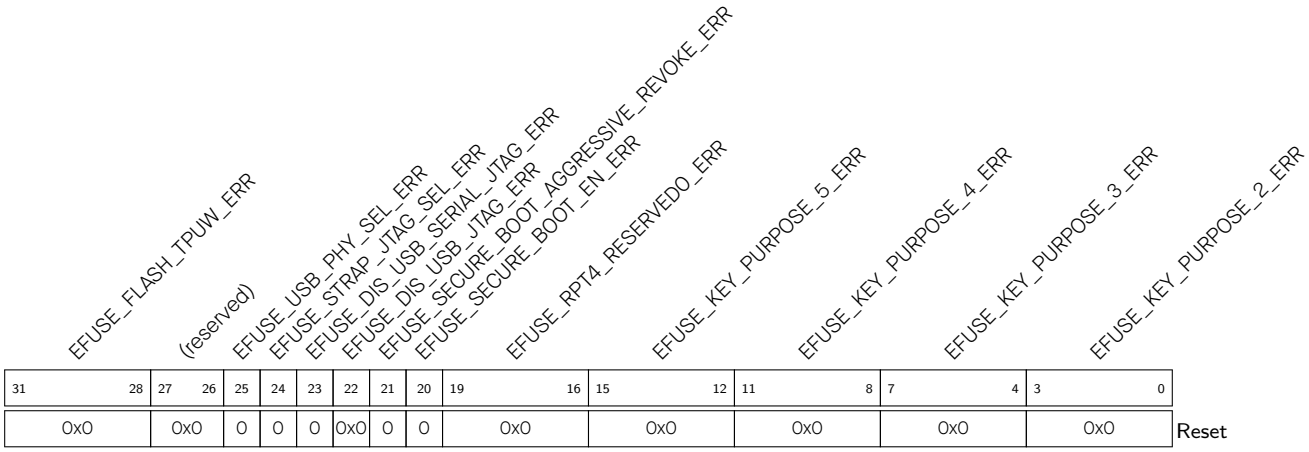
EFUSE_SECURE_BOOT_KEY_REVOKE1_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_SECURE_BOOT_KEY_REVOKE2_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_KEY_PURPOSE_0_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_KEY_PURPOSE_1_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

Register 5.98. EFUSE_RD_REPEAT_ERR2_REG (0x0184)



EFUSE_KEY_PURPOSE_2_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_KEY_PURPOSE_3_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_KEY_PURPOSE_4_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_KEY_PURPOSE_5_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_RPT4_RESERVED0_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_SECURE_BOOT_EN_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE_ERR 若该参数中任意比特为 1, 表示出现烧写错误。(RO)

EFUSE_DIS_USB_JTAG_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

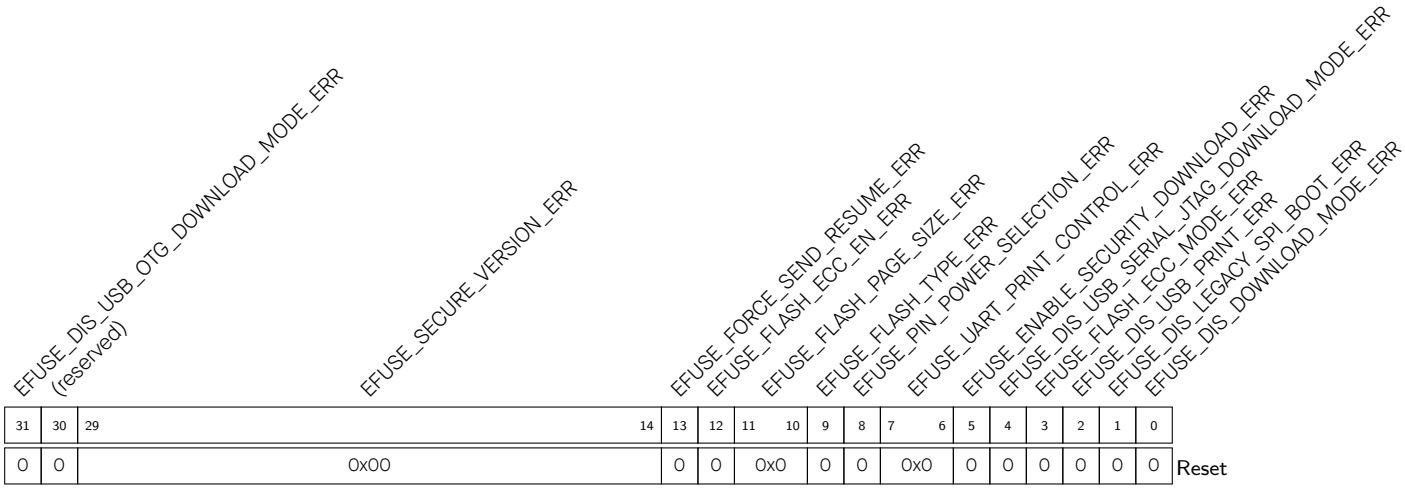
EFUSE_DIS_USB_SERIAL_JTAG_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_STRAP_JTAG_SEL_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_USB_PHY_SEL_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_FLASH_TPUW_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

Register 5.99. EFUSE_RD_REPEAT_ERR3_REG (0x0188)



EFUSE_DIS_DOWNLOAD_MODE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_LEGACY_SPI_BOOT_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_USB_PRINT_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_FLASH_ECC_MODE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_USB_SERIAL_JTAG_DOWNLOAD_MODE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_ENABLE_SECURITY_DOWNLOAD_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_UART_PRINT_CONTROL_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_PIN_POWER_SELECTION_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_FLASH_TYPE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_FLASH_PAGE_SIZE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

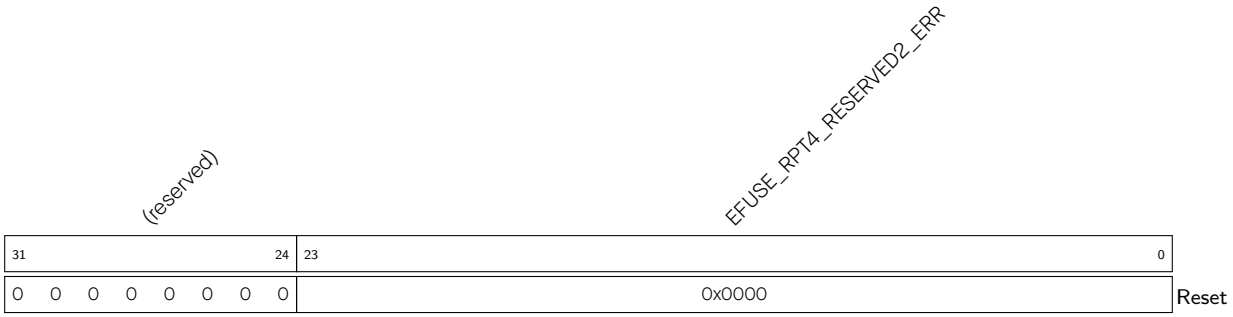
EFUSE_FLASH_ECC_EN_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_FORCE_SEND_RESUME_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_SECURE_VERSION_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

EFUSE_DIS_USB_OTG_DOWNLOAD_MODE_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

Register 5.100. EFUSE_RD_REPEAT_ERR4_REG (0x018C)



EFUSE_RPT4_RESERVED2_ERR 若该参数中任意比特为 1，表示出现烧写错误。(RO)

Register 5.101. EFUSE_RD_RS_ERR0_REG (0x01C0)



EFUSE_MAC_SPI_8M_ERR_NUM 表示烧写过程中 MAC_SPI_8M 数据块的错误字节个数。(RO)

EFUSE_MAC_SPI_8M_FAIL 表示数据是否可靠。0: 无烧写错误，MAC_SPI_8M 的数据是可靠的；1: 烧写 MAC_SPI_8M 数据失败，错误字节数超过 6。(RO)

EFUSE_SYS_PART1_NUM 表示烧写过程中第 1 部分系统数据的错误字节个数。(RO)

EFUSE_SYS_PART1_FAIL 表示数据是否可靠。0: 无烧写错误，第 1 部分的系统数据是可靠的；1: 烧写第 1 部分的系统数据失败，错误字节数超过 6。(RO)

EFUSE_USR_DATA_ERR_NUM 表示烧写过程中用户数据的错误字节个数。(RO)

EFUSE_USR_DATA_FAIL 表示数据是否可靠。0: 无烧写错误，用户数据是可靠的；1: 烧写用户数据失败，错误字节数超过 6。(RO)

EFUSE_KEY0_ERR_NUM 表示烧写过程中 KEY0 的错误字节个数。(RO)

EFUSE_KEY0_FAIL 表示数据是否可靠。0: 无烧写错误，KEY0 数据是可靠的；1: KEY0 烧写失败，错误字节数超过 6。(RO)

EFUSE_KEY1_ERR_NUM 表示烧写过程中 KEY1 的错误字节个数。(RO)

EFUSE_KEY1_FAIL 表示数据是否可靠。0: 无烧写错误，KEY1 数据是可靠的；1: KEY1 烧写失败，错误字节数超过 6。(RO)

EFUSE_KEY2_ERR_NUM 表示烧写过程中 KEY2 的错误字节个数。(RO)

EFUSE_KEY2_FAIL 表示数据是否可靠。0: 无烧写错误，KEY2 数据是可靠的；1: KEY2 烧写失败，错误字节数超过 6。(RO)

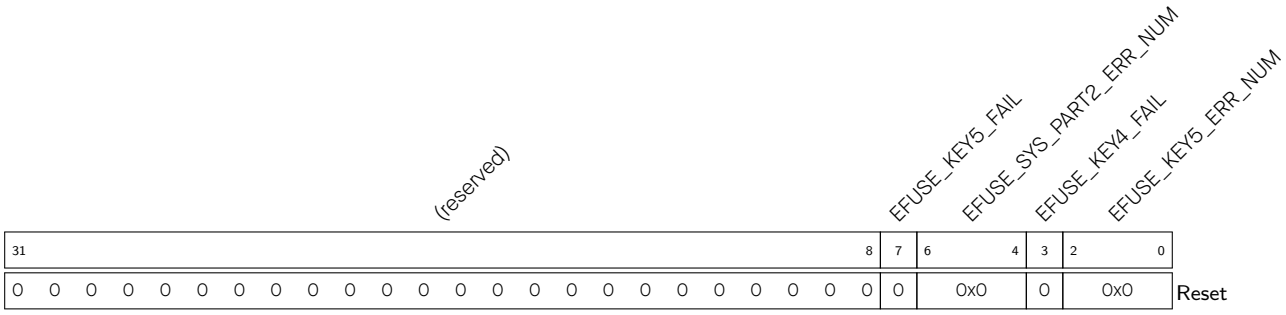
EFUSE_KEY3_ERR_NUM 表示烧写过程中 KEY3 的错误字节个数。(RO)

EFUSE_KEY3_FAIL 表示数据是否可靠。0: 无烧写错误，KEY3 数据是可靠的；1: KEY3 烧写失败，错误字节数超过 6。(RO)

EFUSE_KEY4_ERR_NUM 表示烧写过程中 KEY4 的错误字节个数。(RO)

EFUSE_KEY4_FAIL 表示数据是否可靠。0: 无烧写错误，KEY4 数据是可靠的；1: KEY4 数据烧写失败，错误字节数超过 6。(RO)

Register 5.102. EFUSE_RD_RS_ERR1_REG (0x01C4)



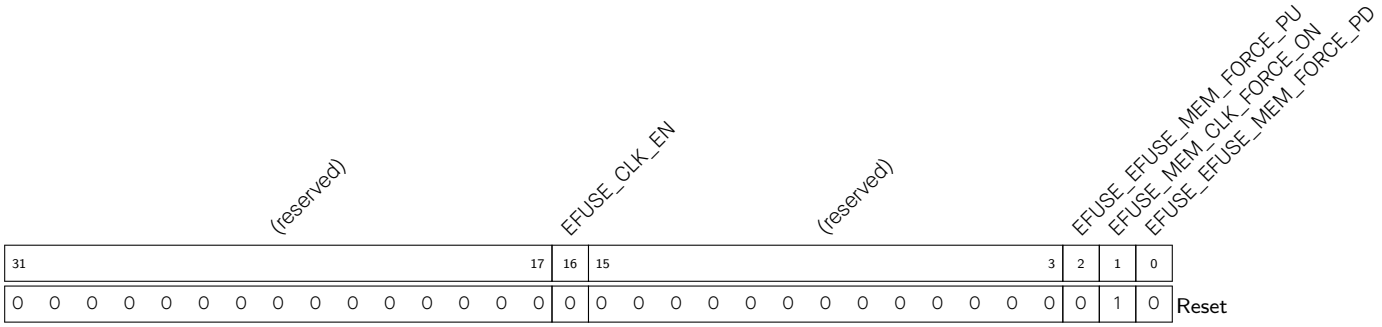
EFUSE_KEY5_ERR_NUM 表示烧写过程中 KEY5 的错误字节个数。(RO)

EFUSE_KEY5_FAIL 表示数据是否可靠。0: 无烧写错误，KEY5 数据是可靠的；1: KEY5 数据烧写失败，错误字节数超过 6。(RO)

EFUSE_SYS_PART2_ERR_NUM 表示烧写过程中第 2 部分系统数据的错误字节个数。(RO)

EFUSE_SYS_PART2_FAIL 表示数据是否可靠。0: 无烧写错误，part2 数据是可靠的；1: part2 数据烧写失败，错误字节数超过 6。(RO)

Register 5.103. EFUSE_CLK_REG (0x01C8)



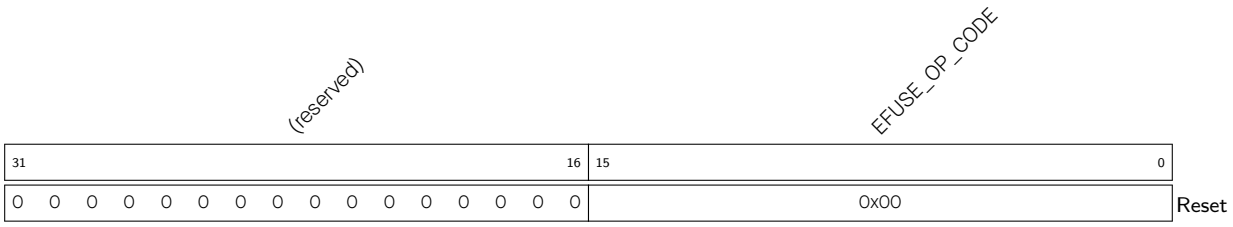
EFUSE_EFUSE_MEM_FORCE_PD 置位强制使 eFuse SRAM 进入低功耗模式。(R/W)

EFUSE_MEM_CLK_FORCE_ON 置位强制激活 eFuse SRAM 的时钟信号。(R/W)

EFUSE_EFUSE_MEM_FORCE_PU 置位强制使 eFuse SRAM 进入工作模式。(R/W)

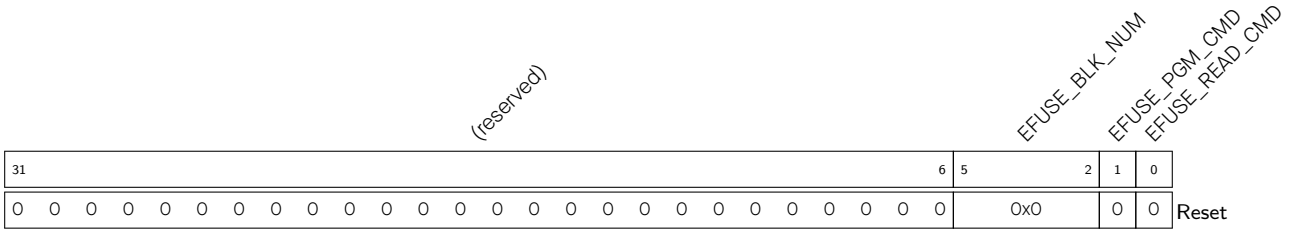
EFUSE_CLK_EN 置位强制使能 eFuse 存储器的时钟信号。(R/W)

Register 5.104. EFUSE_CONF_REG (0x01CC)



EFUSE_OP_CODE 配置运行烧写指令或运行读取指令。0x5A5A：运行烧写指令；0x5AA5：运行读取指令。(R/W)

Register 5.105. EFUSE_CMD_REG (0x01D4)

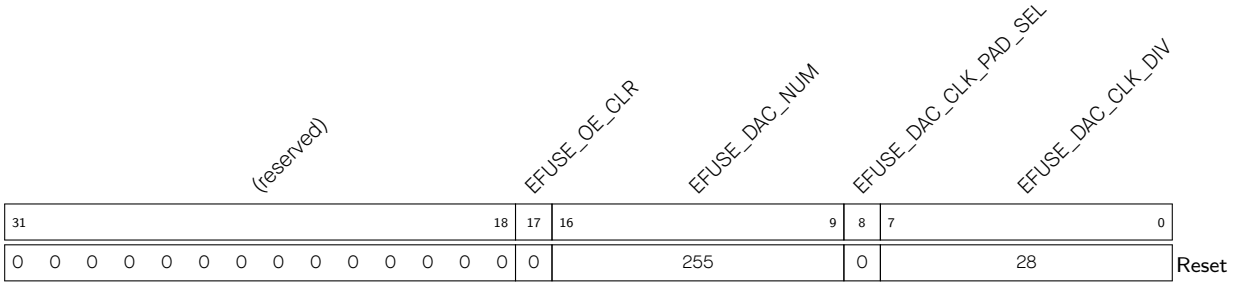


EFUSE_READ_CMD 置位发送读取指令。(R/WS/SC)

EFUSE_PGM_CMD 置位发送烧写指令。(R/WS/SC)

EFUSE_BLK_NUM 表示烧写哪个块，值 0 ~ 10 分别对应 BLOCK0 ~ 10。(R/W)

Register 5.106. EFUSE_DAC_CONF_REG (0x01E8)



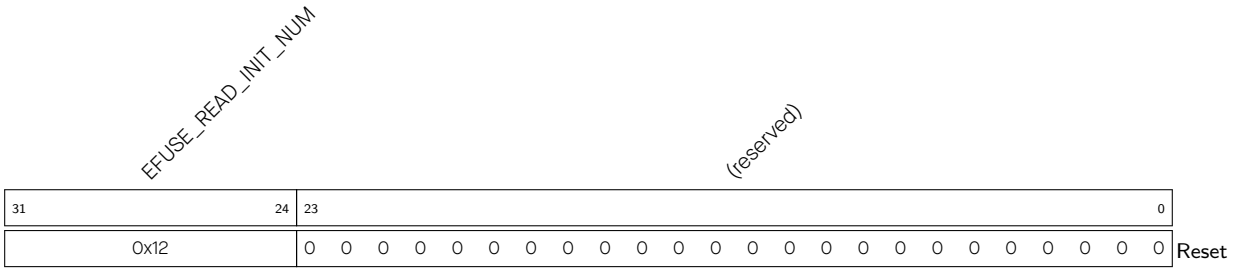
EFUSE_DAC_CLK_DIV 控制烧写电压的爬升时钟分频系数。(R/W)

EFUSE_DAC_CLK_PAD_SEL 无关项。(R/W)

EFUSE_DAC_NUM 烧写供电的上升周期。(R/W)

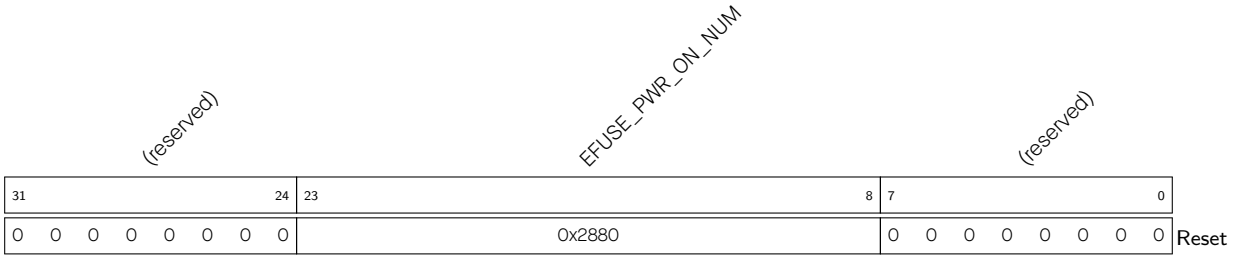
EFUSE_OE_CLR 降低烧写电压的供电能力。(R/W)

Register 5.107. EFUSE_RD_TIM_CONF_REG (0x01EC)



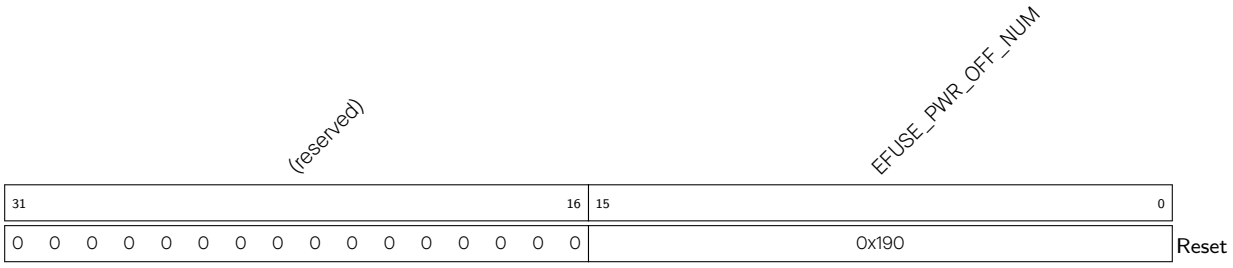
EFUSE_READ_INIT_NUM 配置 eFuse 的首次读取时间。(R/W)

Register 5.108. EFUSE_WR_TIM_CONF1_REG (0x01F4)



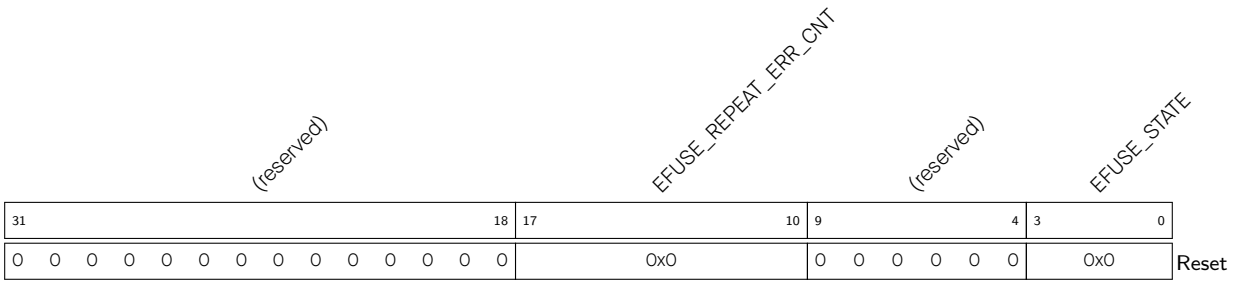
EFUSE_PWR_ON_NUM 配置 VDDQ 的上电时间。(R/W)

Register 5.109. EFUSE_WR_TIM_CONF2_REG (0x01F8)



EFUSE_PWR_OFF_NUM 配置 VDDQ 的掉电时间。(R/W)

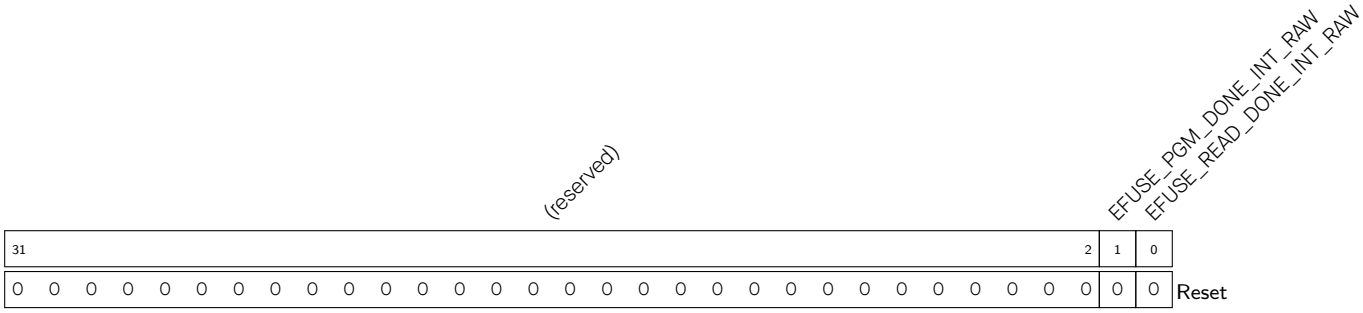
Register 5.110. EFUSE_STATUS_REG (0x01D0)



EFUSE_STATE 表示 eFuse 控制器所处的状态。(RO)

EFUSE_REPEAT_ERR_CNT 表示烧写 BLOCK0 时的错误位的个数。(RO)

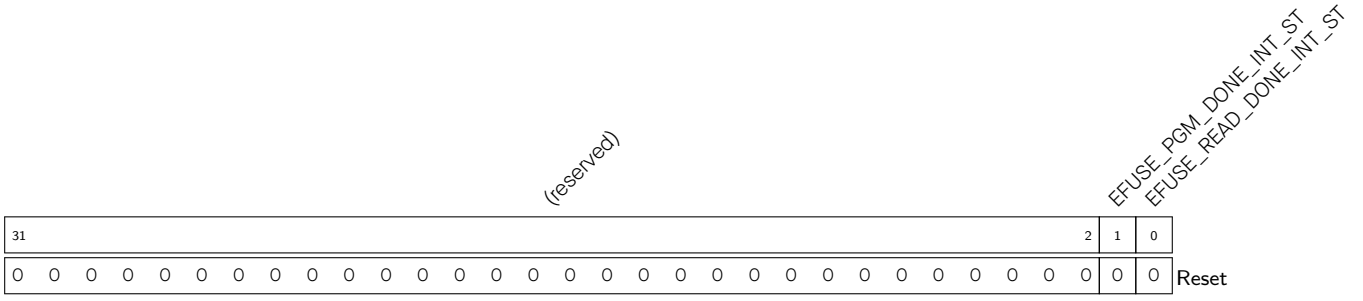
Register 5.111. EFUSE_INT_RAW_REG (0x01D8)



EFUSE_READ_DONE_INT_RAW 读取完成中断的原始中断状态位。(R/WC/SS)

EFUSE_PGM_DONE_INT_RAW 烧写完成中断的原始中断状态位。(R/WC/SS)

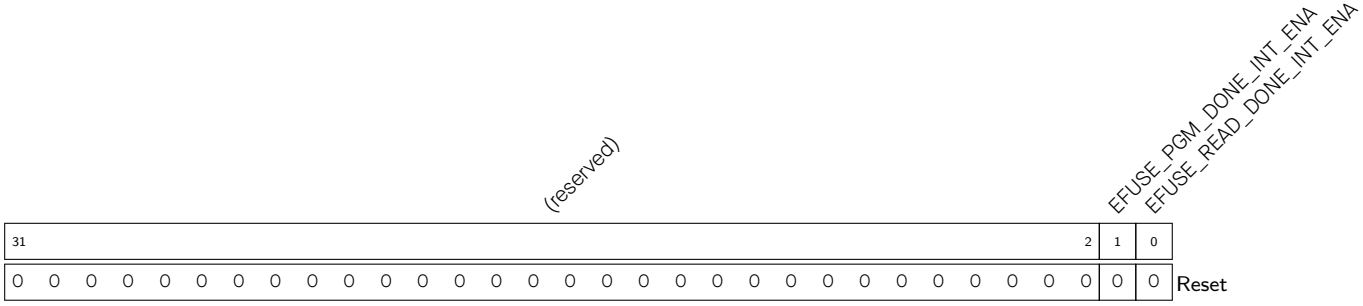
Register 5.112. EFUSE_INT_ST_REG (0x01DC)



EFUSE_READ_DONE_INT_ST 读取完成中断的状态位。(RO)

EFUSE_PGM_DONE_INT_ST 烧写完成中断的状态位。(RO)

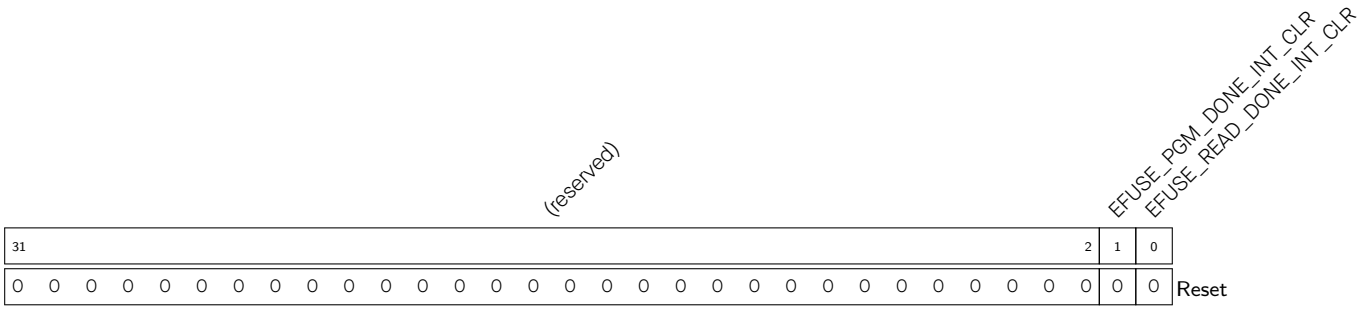
Register 5.113. EFUSE_INT_ENA_REG (0x01E0)



EFUSE_READ_DONE_INT_ENA 读取完成中断的使能位。(R/W)

EFUSE_PGM_DONE_INT_ENA 烧写完成中断的使能位。(R/W)

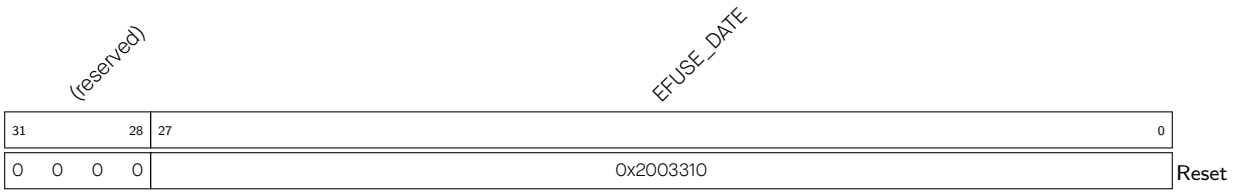
Register 5.114. EFUSE_INT_CLR_REG (0x01E4)



EFUSE_READ_DONE_INT_CLR 读取完成中断的清除位。(WO)

EFUSE_PGM_DONE_INT_CLR 烧写完成中断的清除位。(WO)

Register 5.115. EFUSE_DATE_REG (0x01FC)



EFUSE_DATE 版本控制寄存器。(R/W)

第 III 卷

系统组件

该部分介绍芯片的系统级功能，包括系统启动、时钟、GPIO、定时器、看门狗、中断处理、低功耗管理以及各种系统寄存器。

第 6 章

IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)

6.1 概述

ESP32-S3 芯片有 45 个物理通用输入输出管脚 (GPIO Pin)。每个管脚都可用作一个通用输入输出，或连接一个内部外设信号。利用 GPIO 交换矩阵、IO MUX 和 RTC IO MUX，可配置外设模块的输入信号来源于任何的 GPIO 管脚，并且外设模块的输出信号也可连接到任意 GPIO 管脚。这些模块共同组成了芯片的输入输出控制。

注意：这 45 个物理 GPIO 管脚的编号为：0 ~ 21、26 ~ 48。这些管脚既可作为输入又可作为输出管脚。

6.2 特性

GPIO 交换矩阵特性

- GPIO 交换矩阵是外设输入输出信号和 GPIO 管脚之间的全交换矩阵；
- 175 个数字外设输入信号可以选择任意一个 GPIO 管脚的输入信号；
- 每个 GPIO 管脚的输出信号可以来自 184 个数字外设输出信号的任意一个；
- 支持输入信号经 GPIO SYNC 模块同步至 APB 时钟总线；
- 支持输入信号滤波；
- 支持 Sigma Delta 调制输出 (SDM)；
- 支持 GPIO 简单输入输出。

IO MUX 特性

- 为每个 GPIO 管脚提供一个寄存器 IO_MUX_GPIO*n*_REG，每个管脚可配置成：
  - GPIO 功能，连接 GPIO 交换矩阵；
  - 直连功能，旁路 GPIO 交换矩阵。
- 支持快速信号如 SPI、JTAG、UART 等可以旁路 GPIO 交换矩阵以实现更好的高频数字特性。所以高速信号会直接通过 IO MUX 输入和输出。

RTC IO MUX 特性

- 控制 22 个 RTC GPIO 管脚的低功耗特性；
- 控制 22 个 RTC GPIO 管脚的模拟功能；
- 将 22 个 RTC 输入输出信号引入 RTC 系统。

6.3 结构概览

图 6.3-1 所示为 GPIO 交换矩阵、IO MUX 和 RTC IO MUX 将信号引入外设和引出至管脚的具体过程。

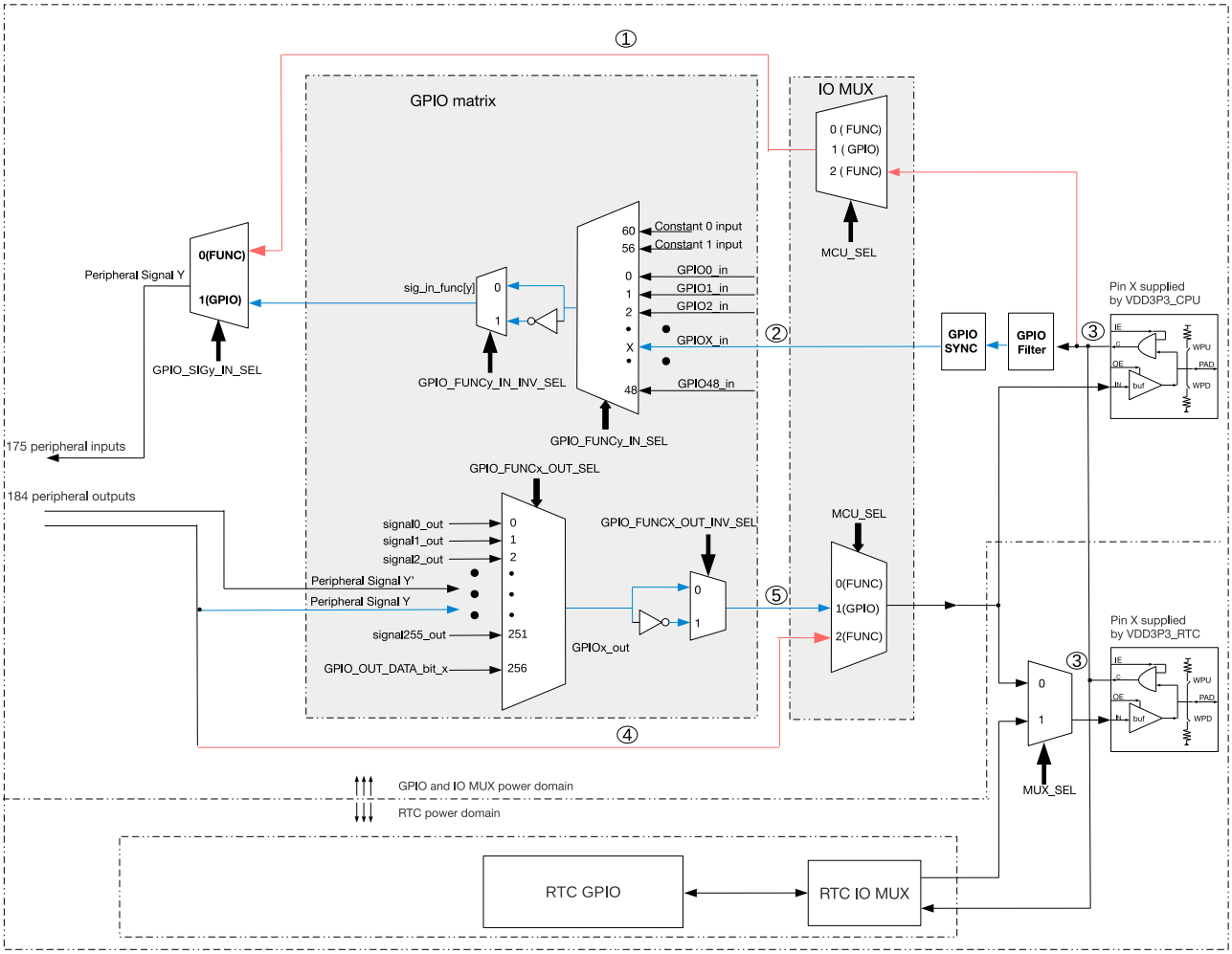


图 6.3-1. IO MUX、RTC IO MUX 和 GPIO 交换矩阵结构框图

① 仅有部分输入信号可以直接通过 IO MUX 直连外设，这些输入信号在表 6.11-1 “信号可经由 IO MUX 直接输入”一栏中被标为“yes”。剩余其它信号只能通过 GPIO 交换矩阵连接至外设；

② ESP32-S3 共有 45 个 GPIO 管脚，因此从 GPIO SYNC 进入到 GPIO 交换矩阵的输入共有 45 个；

③ 位于 VDD3P3_CPU 电源域和 VDD3P3_RTC 电源域的管脚由 IE、OE、WPU 和 WPD 信号控制；

④ 仅有部分输出信号可通过 IO MUX 直连管脚，这些输出信号在表 6.11-1 “信号可经由 IO MUX 直接输出”一栏中被标为“yes”。剩余其它信号只能通过 GPIO 交换矩阵连接至外设；

⑤ 从 GPIO 交换矩阵到 IO MUX 的输出共有 45 个，对应 GPIO X：0 ~ 21、26 ~ 48。

图 6.3-2 展示了芯片焊盘 (PAD) 的内部结构，即芯片逻辑与 GPIO 管脚之间的电气接口。45 个 GPIO 管脚均采用这一结构，且由 IE、OE、WPU 和 WPD 信号控制。

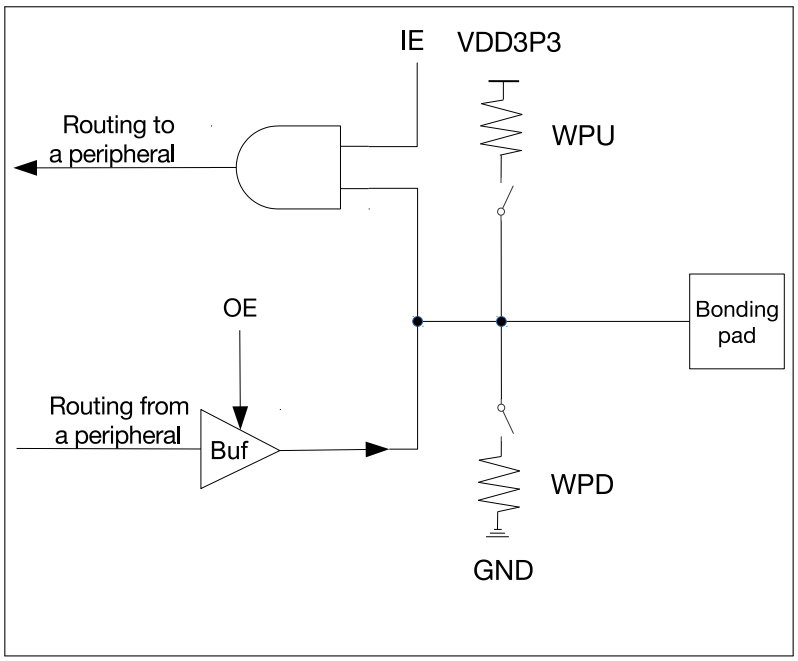


图 6.3-2. 焊盘内部结构

说明：

- IE：输入使能
- OE：输出使能
- WPU：内部弱上拉
- WPD：内部弱下拉
- Bonding pad：接合焊盘，芯片逻辑的结点，实现芯片封装内晶片与 GPIO 管脚之间的物理连接。

6.4 通过 GPIO 交换矩阵的外设输入

6.4.1 概述

为实现通过 GPIO 交换矩阵接收外设输入信号，需要配置 GPIO 交换矩阵从 45 个 GPIO（0 ~ 21、26 ~ 48）中获取外设输入信号，见交换矩阵表格 6.11-1。并需要配置外设输入选择通过 GPIO 交换矩阵接收输入信号。

6.4.2 信号同步

如图 6.3-1 所示，对于信号输入，外部输入信号从 GPIO 管脚输入，经 GPIO SYNC 模块同步至 APB 总线时钟后进入 GPIO 交换矩阵。外部输入信号也可以通过 IO MUX 直接进入外设，但信号无法经由 GPIO SYNC 模块同步。

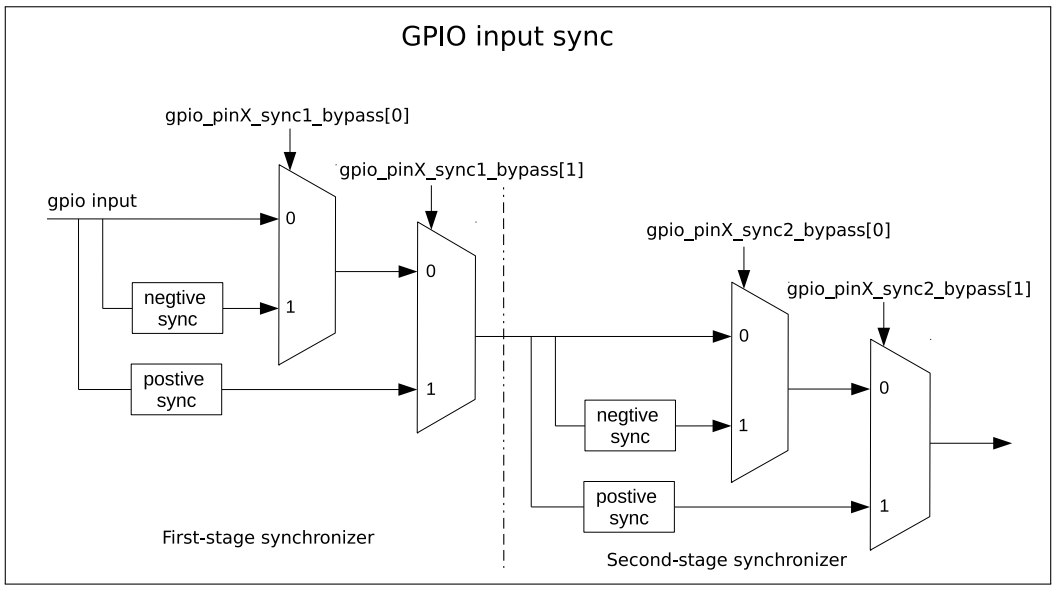


图 6.4-1. GPIO 输入经 APB 时钟上升沿或下降沿同步

GPIO SYNC 模块的功能如图 6.4-1 所示。其中，negative sync 表示 GPIO 输入信号经 APB 时钟的下降沿同步，positive sync 表示 GPIO 输入信号经 APB 时钟上升沿同步。

6.4.3 功能描述

把某个外设输入信号 *Y* 绑定到某个 GPIO 管脚 *X* [1] 的配置过程如下：

1. 在 GPIO 交换矩阵中配置外设信号 *Y* 的 GPIO_FUNC*y*_IN_SEL_CFG_REG 寄存器：
   - 置位 GPIO_SIG*y*_IN_SEL 选择通过 GPIO 交换矩阵接收外部输入信号；
   - 设置 GPIO_FUNC*y*_IN_SEL 为需要的 GPIO 管脚编号，此处应为 *X*。

   注意： 并不是所有外设信号都有有效的 GPIO_SIG*y*_IN_SEL，即有些外设信号只能通过 GPIO 交换矩阵接收外部输入信号。
2. 可选：置位 IO_MUX_FILTER_EN 使能 GPIO 管脚的输入信号滤波功能，如图 6.4-2 所示。只有当输入信号的有效宽度大于两个 APB 时钟周期时，输入信号才会被采样。否则，输入信号将会被滤掉。

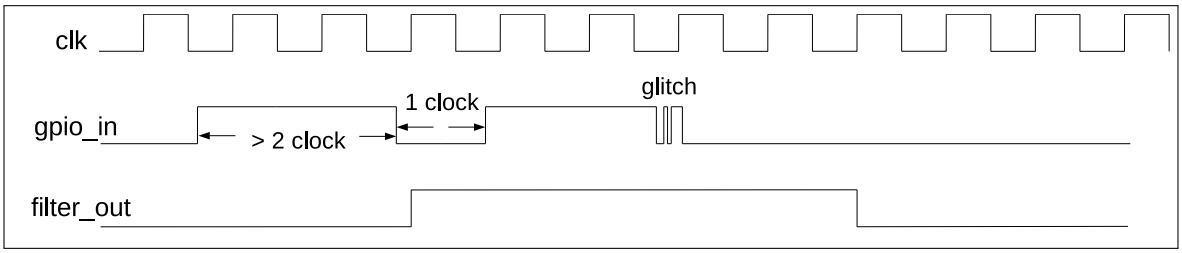


图 6.4-2. GPIO 输入信号滤波时序图

3. 同步 GPIO 输入信号。配置 GPIO 管脚 *X* 的 GPIO_PIN*x*_REG 来同步 GPIO 输入信号，过程如下：
   - 如图 6.4-1 所示，配置 GPIO_PIN*x*_SYNC1_BYPASS 使能输入信号第一拍为上升沿或下降沿同步；
   - 如图 6.4-1 所示，配置 GPIO_PIN*x*_SYNC2_BYPASS 使能输入信号第二拍为上升沿或下降沿同步。

4. 配置 IO MUX 寄存器使能 GPIO 管脚的输入功能。配置 GPIO 管脚 *X* 的 IO_MUX_*x*_REG，过程如下：
    - 置位 IO_MUX_FUN_IE 使能输入 [2]；
    - 置位或清零 IO_MUX_FUN_WPU 和 IO_MUX_FUN_WPD，使能或关闭内部上拉/下拉电阻。

例如，要把 RMT 外设通道 0 的输入信号 rmt_sig_in0[3]（信号索引号 81）绑定到 GPIO40，请按照以下步骤操作。注意，GPIO40 也叫做 MTDO 管脚：

1. 置位 GPIO_FUNC81_IN_SEL_CFG_REG 寄存器中 GPIO_SIG81_IN_SEL 位，使能通过 GPIO 交换矩阵接收外部输入信号；
2. 将 GPIO_FUNC81_IN_SEL_CFG_REG 寄存器中 GPIO_FUNC81_IN_SEL 字段设置为 40，即选择管脚 GPIO40；
3. 置位 IO_MUX_GPIO40_REG 寄存器中 IO_MUX_FUN_IE 位使能管脚输入。

说明：

1. 同一个输入管脚可以同时绑定多个内部输入信号；
2. 置位 GPIO_FUNC*y*_IN_INV_SEL 可以把输入的信号取反；
3. 无需将输入信号绑定到一个 GPIO 管脚也可以使外设读取恒低或恒高电平的输入值。实现方式为选择特定的 GPIO_FUNC*y*_IN_SEL 输入值而不是一个 GPIO 序号：
    - 当 GPIO_FUNC*y*_IN_SEL 是 0x3C 时，输入信号始终为 0；
    - 当 GPIO_FUNC*y*_IN_SEL 是 0x38 时，输入信号始终为 1。

6.4.4 简单 GPIO 输入

GPIO_IN_REG/GPIO_IN1_REG 寄存器存储着每一个 GPIO 管脚的输入值。

任意 GPIO 管脚的输入值都可以随时读取而无需为某一个外设信号配置 GPIO 交换矩阵。但是需要配置 GPIO 管脚 *X* 对应的 IO_MUX_*x*_REG 寄存器中 IO_MUX_FUN_IE 位以使能输入，如章节 6.4.2 所述。

6.5 通过 GPIO 交换矩阵的外设输出

6.5.1 概述

为实现通过 GPIO 交换矩阵输出外设信号，需要配置 GPIO 交换矩阵将外设信号（即在表 6.11-1 中“输出信号”一栏所列出的信号）输出到 45 个 GPIO 管脚（0 ~ 21、26 ~ 48）。

输出信号从外设输出到 GPIO 交换矩阵，然后到达 IO MUX。IO MUX 必须设置相应管脚为 GPIO 功能，这样输出信号就能连接到相应管脚。

说明：

表 6.11-1 中输出索引号为 208 ~ 212 的输出信号，与输入索引号为 208 ~ 212 的输入信号对应相连。可配置从一个 GPIO 管脚输入，直接由另一个 GPIO 管脚输出。

6.5.2 功能描述

如图 6.3-1 所示，对于信号输出，256 个输出信号（即在表 6.11-1 中“输出信号”列的所有信号）中的某一个信号通过 GPIO 交换矩阵到达 IO MUX，然后连接到某个 GPIO 管脚。

输出外设信号 *Y* 到某一 GPIO 管脚 *X*[1，2] 的步骤为：

1. 在 GPIO 交换矩阵里配置 GPIO 管脚 *X* 的 GPIO_FUNC*x*_OUT_SEL_CFG_REG 寄存器和 GPIO_ENABLE_REG[*x*] 字段。推荐使用相应 GPIO_ENABLE_W1TS_REG (write 1 to set) 或 GPIO_ENABLE_W1TC_REG (write 1 to clear) 寄存器来更新 GPIO_ENABLE_REG 中的值：
   - 设置 GPIO_FUNC*x*_OUT_SEL_CFG_REG 寄存器的 GPIO_FUNC*x*_OUT_SEL 字段为外设输出信号 *Y* 的索引号 (*Y*)。
   - 要将信号强制使能为输出模式，需要将 GPIO 管脚 *X* 对应的 GPIO_FUNC*x*_OUT_SEL_CFG_REG 寄存器中 GPIO_FUNC*x*_OEN_SEL 字段置位；同时需要将 GPIO_ENABLE_W1TS_REG 或 GPIO_ENABLE1_W1TS_REG 中相应字段置位。或者，将 GPIO_FUNC*x*_OEN_SEL 清零，即选择采用外设的输出使能信号，此时输出使能信号由内部逻辑功能决定。比如，表 6.11-1 中 “GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号” 一栏的 SPIQ_oe 信号。
   - 置位 GPIO_ENABLE_W1TC_REG 或 GPIO_ENABLE1_W1TC_REG 中相应位可以关闭 GPIO 管脚的输出。
2. 要选择以开漏方式输出，可以设置 GPIO 管脚 *X* 的 GPIO_PIN*x*_REG 寄存器中 GPIO_PIN*x*_PAD_DRIVER 位。
3. 配置 IO MUX 寄存器来选择经由 GPIO 交换矩阵输出信号。配置 GPIO 管脚 *X* 相应寄存器 IO_MUX_*x*_REG 的过程如下：
   - 配置 GPIO 管脚 *X* 的 IO_MUX_MCU_SEL 为所需的管脚功能。此处选择数值 1，即 Function 1（GPIO 功能），适用于所有管脚。
   - 设置 IO_MUX_FUN_DRV 字段为特定的输出强度值 (0 ~ 3)。
   - 在开漏模式下，通过置位/清零 IO_MUX_FUN_WPU 和 IO_MUX_FUN_WPD 使能或关闭上拉/下拉电阻。

说明：

1. 某一个外设的输出信号可以同时从多个管脚输出；
2. 置位 GPIO_FUNC*x*_OUT_INV_SEL 可以把输出的信号取反。

6.5.3 简单 GPIO 输出

GPIO 交换矩阵也可用于简单 GPIO 输出，具体配置如下：

- 设置 GPIO 交换矩阵 GPIO_FUNC*n*_OUT_SEL 为特定的外设索引值 256 (0x100)；
- 设置 GPIO_OUT_REG[31:0] 或者 GPIO_OUT1_REG[21:0] 寄存器中相应位的值为期望 GPIO 输出的值。

说明：

- GPIO_OUT_REG[0] ~ GPIO_OUT_REG[21] 对应 GPIO0 ~ GPIO21；GPIO_OUT_REG[26] ~ GPIO_OUT_REG[31] 对应 GPIO26 ~ GPIO31。GPIO_OUT_REG[25:22] 无效。

- GPIO_OUT1_REG[0] ~ GPIO_OUT1_REG[16] 对应 GPIO32 ~ GPIO48，GPIO_OUT1_REG[21:17] 无效。
- 推荐使用相应的 W1TS (write 1 to set) 和 W1TC (write 1 to clear) 寄存器，例如：GPIO_OUT_W1TS/GPIO_OUT_W1TC 来置位/清零 GPIO_OUT_REG 或 GPIO_OUT1_REG。

6.5.4 Sigma Delta 调制输出 (SDM)

6.5.4.1 功能描述

256 个数字外设输出中有八个信号（在表 6.11-1 中索引为：93 ~ 100）支持 1-bit 二阶 Sigma Delta 调制输出。上述 8 个信号通道默认输出使能。Sigma Delta 调制器可实现输出可配占空比的 PDM（脉冲密度调制）信号。二阶 Sigma Delta 调制器传输函数为：

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

E(z) 为量化误差，X(z) 为输入。
Sigma Delta 调制器内部支持对 APB_CLK 的 1 ~ 256 倍分频：

- 置位 GPIO_FUNCTION_CLK_EN 使能调制器时钟；
- 配置寄存器 GPIO_SD*n*_PRESCALE 实现分频。*n* 为 0 ~ 7，对应 8 个通道。

分频后的时钟周期为调制器输出单位脉冲的周期。

GPIO_SD*n*_IN 为有符号数，范围为 [-128, 127]，配置此寄存器控制输出 PDM 信号的占空比 [1]。

- GPIO_SD*n*_IN = -128，调制器输出信号占空比为 0%；
- GPIO_SD*n*_IN = 0，调制器输出信号占空比接近 50%；
- GPIO_SD*n*_IN = 127，调制器输出信号占空比接近 100%。

PDM 信号占空比计算公式为：

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

说明：
对 PDM 信号来说，占空比是指在若干脉冲周期内（比如 256 个脉冲周期），高电平占整个统计周期的比值。

6.5.4.2 配置方法

SDM 的配置方法如下：

- 将 SDM 输出经 GPIO 交换矩阵连接至管脚，见章节 6.5.2；
- 置位 GPIO_FUNCTION_CLK_EN，使能 SDM 时钟；
- 配置 GPIO_SD*n*_PRESCALE 设置时钟分频系数；
- 配置 GPIO_SD*n*_IN 设置 SDM 输出信号的占空比。

6.6 IO MUX 的直接输入输出功能

6.6.1 概述

快速信号如 SPI、JTAG 等会旁路 GPIO 交换矩阵以实现更好的高频数字特性。所以高速信号会直接通过 IO MUX 输入和输出。

这样比使用 GPIO 交换矩阵的灵活度要低，即每个 GPIO 管脚的 IO MUX 寄存器只有较少的功能选择，但可以实现更好的高频数字特性。

6.6.2 功能描述

对于外设输入信号，旁路 GPIO 交换矩阵必须配置两个寄存器：

1. GPIO 管脚的 IO_MUX_MCU_SEL 必须设置为相应的管脚功能，章节 6.12 列出了管脚功能。
2. 清零 GPIO_SIG*n*_IN_SEL，直接将输入信号连接到外设。

对于外设输出信号，旁路 GPIO 交换矩阵只需将 GPIO 管脚的 IO_MUX_MCU_SEL 配置为相应的管脚功能即可。

说明：
并非所有外设输入/输出信号均可直接通过 IO MUX 连接到外设，某些输入/输出信号只能通过 GPIO 交换矩阵连接到外设。

6.7 RTC IO MUX 的低功耗性能和模拟输入输出功能

6.7.1 概述

ESP32-S3 中有 22 个 GPIO 管脚具有低功耗 (RTC) 性能和模拟功能，由 RTC 子系统控制。这些功能不使用 IO MUX 和 GPIO 交换矩阵，而是使用 RTC IO MUX 将 22 个 RTC 输入输出信号引入 RTC 子系统。

当这些管脚被配置为 RTC GPIO 管脚，作为输出管脚时仍然能够在芯片处于 Deep-sleep 模式下保持输出电平值或者作为输入管脚使用时可以将芯片从 Deep-sleep 中唤醒。

6.7.2 低功耗性能描述

每个管脚的 RTC 功能是由 RTC_IO_TOUCH/RTC_PAD*n*_REG 寄存器中的 RTC_IO_TOUCH/RTC_PAD*n*_MUX_SEL 位控制的。此位默认置为 0，通过 IO MUX 子系统输入输出信号，如前文所述。

如果置位 RTC_IO_TOUCH/RTC_PAD*n*_MUX_SEL，则输入输出信号会经过 RTC 子系统。在这种模式下，RTC_IO_TOUCH/RTC_PAD*n*_REG 寄存器用于控制 RTC 低功耗 GPIO 管脚。表 6.13-1 列出了 GPIO 管脚的 RTC 功能。请注意 RTC_IO_TOUCH/RTC_PAD*n*_REG 寄存器使用的是 RTC GPIO 管脚的序号，不是 GPIO 管脚的序号。

6.7.3 模拟功能描述

当使用管脚的模拟功能时，需要确保该管脚处于悬空状态，此时外部模拟信号通过 GPIO 管脚直接与内部的模拟信号相连。通常利用 RTC_IO_TOUCH/RTC_PAD*n*_REG 寄存器控制使管脚处于浮空状态。相关配置如下：

- 置位 RTC_IO_TOUCH/RTC_PAD*n*_MUX_SEL 选择通过 RTC IO MUX 输入输出信号；
- 同时清零 RTC_IO_TOUCH/RTC_PAD*n*_FUN_IE、RTC_IO_TOUCH/RTC_PAD*n*_FUN_RUE、RTC_IO_TOUCH/RTC_PAD*n*_FUN_RDE 将管脚设置为浮空状态；
- 配置 RTC_IO_TOUCH/RTC_PAD*n*_FUN_SEL 为 0，即选择模拟功能 0；
- 向 RTC_IO_GPIO_ENABLE_W1TC 中对应位写 1，清零输出使能。

表 6.13-2 列出了 GPIO 管脚的模拟功能。

6.8 Light-sleep 模式管脚功能

当 ESP32-S3 处于 Light-sleep 模式时管脚可以有不同的功能。如果某一 GPIO 管脚的 IO_MUX_*n*_REG 寄存器中 IO_MUX_SLP_SEL 位置为 1，芯片处于 Light-sleep 模式下将由另一组不同的寄存器控制管脚。

表 6.8-1. IO MUX Light-sleep 管脚功能控制寄存器

IO MUX 功能	正常工作模式 OR IO_MUX_SLP_SEL = 0	Light-sleep 模式 AND IO_MUX_SLP_SEL = 1
输出驱动强度	IO_MUX_FUN_DRV	IO_MUX_MCU_DRV
上拉电阻	IO_MUX_FUN_WPU	IO_MUX_MCU_WPU
下拉电阻	IO_MUX_FUN_WPD	IO_MUX_MCU_WPD
输出使能	由 GPIO 交换矩阵的 OEN_SEL 位控制 *	IO_MUX_MCU_OE

说明：
如果 IO_MUX_SLP_SEL 置为 0，则芯片在正常工作和 Light-sleep 模式下，管脚的功能一样。此时，具体的输出使能配置请参考 6.5.2 章节。

6.9 管脚 Hold 特性

每个 GPIO 管脚（包括 RTC 管脚）都有单独的 Hold 功能，由 RTC 寄存器控制。管脚的 Hold 功能被置上后，管脚在置上 Hold 那一刻的状态被强制保持，无论内部信号如何变化，修改 IO MUX 配置或者 GPIO 配置，都不会改变管脚的状态。应用如果希望在看门狗超时触发内核复位时或 Deep-sleep 触发内核复位时管脚的状态不被改变，就需要提前把 Hold 置上。

- **数字管脚** (GPIO26 ~ GPIO48)
  每个数字管脚的 Hold 状态由各管脚的 Hold 使能信号和全局的 Hold 使能信号经过或运算后共同控制。
  - RTC_CNTL_DIG_PAD_HOLD_REG[*n*]，控制 GPIO26 ~ GPIO48 各个管脚的 Hold 使能信号。
  - RTC_CNTL_DG_PAD_FORCE_HOLD，控制数字管脚全局的 Hold 使能信号。

  要使用此功能，请按照以下步骤操作：
  - 若要在 Deep-sleep 中保持管脚输入输出的状态值，需要在掉电之前将寄存器 RTC_CNTL_DIG_PAD_HOLD_REG[*n*] 置位，*n* = 5 ~ 27，对应管脚 GPIO26 ~ GPIO48。在芯片被唤醒后，若要关闭 Hold 功能，可将上述寄存器清除。

  - 或者，将 RTC_CNTL_DG_PAD_FORCE_HOLD 置位，用以保持所有数字管脚的状态值。在芯片被唤醒后，也可将 RTC_CNTL_DG_PAD_FORCE_UNHOLD 置位，用以关闭所有数字管脚的 Hold 功能。

- **RTC 管脚** (GPIO0 ~ GPIO21)
  每个 RTC 管脚的 Hold 状态由各管脚的 Hold 使能信号和全局的 Hold 使能信号经过或运算后共同控制。
  - RTC_CNTL_RTC_PAD_HOLD_REG[*n*](*n* = 0 ~ 21)，控制 GPIO0 ~ GPIO21 各个管脚的 Hold 使能信号。
  - RTC_CNTL_RTC_PAD_FORCE_HOLD 控制 RTC 管脚全局的 Hold 使能信号。

  要使用此功能，请按照以下步骤操作：
  - 若要在 Deep-sleep 中保持管脚输入输出的状态值，需要在掉电之前将寄存器 RTC_CNTL_RTC_PAD_HOLD_REG[*n*] 置位，*n* = 0 ~ 21，对应管脚 GPIO0 ~ GPIO21。在芯片被唤醒后，若要关闭 Hold 功能，可将上述寄存器清除。
  - 或者，将 RTC_CNTL_RTC_PAD_FORCE_HOLD 置位，用以保持所有 RTC 管脚的状态值。在芯片被唤醒后，也可将 RTC_CNTL_RTC_PAD_FORCE_HOLD 清零，用以关闭所有 RTC 管脚的 Hold 功能。

6.10 GPIO 管脚供电和电源管理

6.10.1 GPIO 管脚供电

GPIO 管脚供电请参考《ESP32-S3 技术规格书》中管脚定义章节。

6.10.2 电源管理

ESP32-S3 的管脚可分为如下三种不同的电源域。

- VDD3P3_RTC：RTC 和 CPU 的输入电源
- VDD3P3_CPU：CPU 的输入电源
- VDD_SPI：可配置为输入电源或输出电源

VDD_SPI 可配置使用一个内置 LDO，该内置 LDO 的输入和输出均为 1.8 V。如未使能 LDO，VDD_SPI 可以与 VDD3P3_RTC 连接在相同的电源上。

VDD_SPI 的具体配置由 GPIO45 的 Strapping 值决定，用户可通过 eFuse 或寄存器修改 VDD_SPI 的配置。请参考《ESP32-S3 技术规格书》中的电源管理章节和 Strapping 管脚章节查看更多信息。

其中，GPIO33 ~ GPIO37 和 GPIO47 ~ GPIO48 管脚既可以由 VDD_SPI 供电，也可以由 VDD3P3_CPU 供电。

6.11 GPIO 交换矩阵外设信号列表

表 6.11-1 列出了所有经由 GPIO 交换矩阵的外设输入输出信号。

请注意 GPIO_FUNC*n*_OEN_SEL 位的配置：

- GPIO_FUNC*n*_OEN_SEL = 1，则寄存器 GPIO_ENABLE_REG 中的相应位 *n* 将用于控制信号输出使能。
  - GPIO_ENABLE_REG = 0：输出关闭；

  - GPIO_ENABLE_REG = 1：输出使能；

- GPIO_FUNC*n*_OEN_SEL = 0，则输出信号的使能由外设控制，例如表 6.11-1 中 “GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号” 一栏的 SPIQ_oe。注意，使能信号 SPIQ_oe 可设置为 1 (1'd1) 或 0 (1'd0)，具体由外设的配置决定。如果 “GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号” 一栏中为 1'd1，则表示寄存器 GPIO_FUNC*n*_OEN_SEL 已清零，输出信号默认始终使能。

说明：

信号连续编号，但并非所有信号均有效。

- 表 6.11-1 “输入信号” 一栏中有名字的信号均为有效输入信号；
- 表 6.11-1 “输出信号” 一栏中有名字的信号均为有效输出信号。

GoBack

表 6.11-1. GPIO 交换矩阵外设信号

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
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	-	-	-	SPICS1_out	SPICS1_oe	yes
7	SPID4_in	0	yes	SPID4_out	SPID4_oe	yes
8	SPID5_in	0	yes	SPID5_out	SPID5_oe	yes
9	SPID6_in	0	yes	SPID6_out	SPID6_oe	yes
10	SPID7_in	0	yes	SPID7_out	SPID7_oe	yes
11	SPIDQS_in	0	yes	SPIDQS_out	SPIDQS_oe	yes
12	U0RXD_in	0	yes	U0TXD_out	1'd1	yes
13	U0CTS_in	0	yes	U0RTS_out	1'd1	yes
14	U0DSR_in	0	no	U0DTR_out	1'd1	no
15	U1RXD_in	0	yes	U1TXD_out	1'd1	yes
16	U1CTS_in	0	yes	U1RTS_out	1'd1	yes
17	U1DSR_in	0	no	U1DTR_out	1'd1	no
18	U2RXD_in	0	no	U2TXD_out	1'd1	no
19	U2CTS_in	0	no	U2RTS_out	1'd1	no
20	U2DSR_in	0	no	U2DTR_out	1'd1	no
21	I2S1_MCLK_in	0	no	I2S1_MCLK_out	1'd1	no
22	I2S0O_BCK_in	0	no	I2S0O_BCK_out	1'd1	no
23	I2S0_MCLK_in	0	no	I2S0_MCLK_out	1'd1	no
24	I2S0O_WS_in	0	no	I2S0O_WS_out	1'd1	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
25	I2S0I_SD_in	0	no	I2S0O_SD_out	1'd1	no
26	I2S0I_BCK_in	0	no	I2S0I_BCK_out	1'd1	no
27	I2S0I_WS_in	0	no	I2S0I_WS_out	1'd1	no
28	I2S1O_BCK_in	0	no	I2S1O_BCK_out	1'd1	no
29	I2S1O_WS_in	0	no	I2S1O_WS_out	1'd1	no
30	I2S1I_SD_in	0	no	I2S1O_SD_out	1'd1	no
31	I2S1I_BCK_in	0	no	I2S1I_BCK_out	1'd1	no
32	I2S1I_WS_in	0	no	I2S1I_WS_out	1'd1	no
33	pcnt_sig_ch0_in0	0	no	-	1'd1	no
34	pcnt_sig_ch1_in0	0	no	-	1'd1	no
35	pcnt_ctrl_ch0_in0	0	no	-	1'd1	-
36	pcnt_ctrl_ch1_in0	0	no	-	1'd1	-
37	pcnt_sig_ch0_in1	0	no	-	1'd1	-
38	pcnt_sig_ch1_in1	0	no	-	1'd1	-
39	pcnt_ctrl_ch0_in1	0	no	-	1'd1	-
40	pcnt_ctrl_ch1_in1	0	no	-	1'd1	-
41	pcnt_sig_ch0_in2	0	no	-	1'd1	-
42	pcnt_sig_ch1_in2	0	no	-	1'd1	-
43	pcnt_ctrl_ch0_in2	0	no	-	1'd1	-
44	pcnt_ctrl_ch1_in2	0	no	-	1'd1	-
45	pcnt_sig_ch0_in3	0	no	-	1'd1	-
46	pcnt_sig_ch1_in3	0	no	-	1'd1	-
47	pcnt_ctrl_ch0_in3	0	no	-	1'd1	-
48	pcnt_ctrl_ch1_in3	0	no	-	1'd1	-
49	-	-	-	-	1'd1	-
50	-	-	-	-	1'd1	-
51	I2S0I_SD1_in	0	no	-	1'd1	-

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
52	I2S0I_SD2_in	0	no	-	1'd1	-
53	I2S0I_SD3_in	0	no	-	1'd1	-
54	Core1_gpio_in7	0	no	Core1_gpio_out7	1'd1	no
55	-	-	-	-	1'd1	-
56	-	-	-	-	1'd1	-
57	-	-	-	-	1'd1	-
58	usb_otg_iddig_in	0	no	-	1'd1	-
59	usb_otg_avalid_in	0	no	-	1'd1	-
60	usb_srp_bvalid_in	0	no	usb_otg_idpullup	1'd1	no
61	usb_otg_vbusvalid_in	0	no	usb_otg_dppulldown	1'd1	no
62	usb_srp_sessend_in	0	no	usb_otg_dmpulldown	1'd1	no
63	-	-	-	usb_otg_drvvbus	1'd1	no
64	-	-	-	usb_srp_chrgvbus	1'd1	no
65	-	-	-	usb_srp_dischrgvbus	1'd1	no
66	SPI3_CLK_in	0	no	SPI3_CLK_out_mux	SPI3_CLK_oe	no
67	SPI3_Q_in	0	no	SPI3_Q_out	SPI3_Q_oe	no
68	SPI3_D_in	0	no	SPI3_D_out	SPI3_D_oe	no
69	SPI3_HD_in	0	no	SPI3_HD_out	SPI3_HD_oe	no
70	SPI3_WP_in	0	no	SPI3_WP_out	SPI3_WP_oe	no
71	SPI3_CS0_in	0	no	SPI3_CS0_out	SPI3_CS0_oe	no
72	-	-	-	SPI3_CS1_out	SPI3_CS1_oe	no
73	ext_adc_start	0	no	ledc_ls_sig_out0	1'd1	no
74	-	-	-	ledc_ls_sig_out1	1'd1	no
75	-	-	-	ledc_ls_sig_out2	1'd1	no
76	-	-	-	ledc_ls_sig_out3	1'd1	no
77	-	-	-	ledc_ls_sig_out4	1'd1	no
78	-	-	-	ledc_ls_sig_out5	1'd1	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
79	-	-	-	ledc_ls_sig_out6	1'd1	no
80	-	-	-	ledc_ls_sig_out7	1'd1	no
81	rmt_sig_in0	0	no	rmt_sig_out0	1'd1	no
82	rmt_sig_in1	0	no	rmt_sig_out1	1'd1	no
83	rmt_sig_in2	0	no	rmt_sig_out2	1'd1	no
84	rmt_sig_in3	0	no	rmt_sig_out3	1'd1	no
85	-	-	-	-	1'd1	-
86	-	-	-	-	1'd1	-
87	-	-	-	-	1'd1	-
88	-	-	-	-	1'd1	-
89	I2CEXT0_SCL_in	1	no	I2CEXT0_SCL_out	I2CEXT0_SCL_oe	no
90	I2CEXT0_SDA_in	1	no	I2CEXT0_SDA_out	I2CEXT0_SDA_oe	no
91	I2CEXT1_SCL_in	1	no	I2CEXT1_SCL_out	I2CEXT1_SCL_oe	no
92	I2CEXT1_SDA_in	1	no	I2CEXT1_SDA_out	I2CEXT1_SDA_oe	no
93	-	-	-	gpio_sd0_out	1'd1	no
94	-	-	-	gpio_sd1_out	1'd1	no
95	-	-	-	gpio_sd2_out	1'd1	no
96	-	-	-	gpio_sd3_out	1'd1	no
97	-	-	-	gpio_sd4_out	1'd1	no
98	-	-	-	gpio_sd5_out	1'd1	no
99	-	-	-	gpio_sd6_out	1'd1	no
100	-	-	-	gpio_sd7_out	1'd1	no
101	FSPICLK_in	0	yes	FSPICLK_out_mux	FSPICLK_oe	yes
102	FSPIQ_in	0	yes	FSPIQ_out	FSPIQ_oe	yes
103	FSPID_in	0	yes	FSPID_out	FSPID_oe	yes
104	FSPIHD_in	0	yes	FSPIHD_out	FSPIHD_oe	yes
105	FSPIWP_in	0	yes	FSPIWP_out	FSPIWP_oe	yes

GoBack

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
106	FSPII04_in	0	yes	FSPII04_out	FSPII04_oe	yes
107	FSPII05_in	0	yes	FSPII05_out	FSPII05_oe	yes
108	FSPII06_in	0	yes	FSPII06_out	FSPII06_oe	yes
109	FSPII07_in	0	yes	FSPII07_out	FSPII07_oe	yes
110	FSPICS0_in	0	yes	FSPICS0_out	FSPICS0_oe	yes
111	-	-	-	FSPICS1_out	FSPICS1_oe	no
112	-	-	-	FSPICS2_out	FSPICS2_oe	no
113	-	-	-	FSPICS3_out	FSPICS3_oe	no
114	-	-	-	FSPICS4_out	FSPICS4_oe	no
115	-	-	-	FSPICS5_out	FSPICS5_oe	no
116	twai_rx	1	no	twai_tx	1'd1	no
117	-	-	-	twai_bus_off_on	1'd1	no
118	-	-	-	twai_clkout	1'd1	no
119	-	-	-	SUBSPICLK_out_mux	SUBSPICLK_oe	no
120	SUBSPIQ_in	0	yes	SUBSPIQ_out	SUBSPIQ_oe	yes
121	SUBSPID_in	0	yes	SUBSPID_out	SUBSPID_oe	yes
122	SUBSPIHD_in	0	yes	SUBSPIHD_out	SUBSPIHD_oe	yes
123	SUBSPIWP_in	0	yes	SUBSPIWP_out	SUBSPIWP_oe	yes
124	-	-	-	SUBSPICS0_out	SUBSPICS0_oe	yes
125	-	-	-	SUBSPICS1_out	SUBSPICS1_oe	yes
126	-	-	-	FSPIDQS_out	FSPIDQS_oe	yes
127	-	-	-	SPI3_CS2_out	SPI3_CS2_oe	no
128	-	-	-	I2S0O_SD1_out	1'd1	no
129	Core1_gpio_in0	0	no	Core1_gpio_out0	1'd1	no
130	Core1_gpio_in1	0	no	Core1_gpio_out1	1'd1	no
131	Core1_gpio_in2	0	no	Core1_gpio_out2	1'd1	no
132	-	-	-	LCD_CS	1'd1	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
133	CAM_DATA_in0	0	no	LCD_DATA_out0	1'd1	no
134	CAM_DATA_in1	0	no	LCD_DATA_out1	1'd1	no
135	CAM_DATA_in2	0	no	LCD_DATA_out2	1'd1	no
136	CAM_DATA_in3	0	no	LCD_DATA_out3	1'd1	no
137	CAM_DATA_in4	0	no	LCD_DATA_out4	1'd1	no
138	CAM_DATA_in5	0	no	LCD_DATA_out5	1'd1	no
139	CAM_DATA_in6	0	no	LCD_DATA_out6	1'd1	no
140	CAM_DATA_in7	0	no	LCD_DATA_out7	1'd1	no
141	CAM_DATA_in8	0	no	LCD_DATA_out8	1'd1	no
142	CAM_DATA_in9	0	no	LCD_DATA_out9	1'd1	no
143	CAM_DATA_in10	0	no	LCD_DATA_out10	1'd1	no
144	CAM_DATA_in11	0	no	LCD_DATA_out11	1'd1	no
145	CAM_DATA_in12	0	no	LCD_DATA_out12	1'd1	no
146	CAM_DATA_in13	0	no	LCD_DATA_out13	1'd1	no
147	CAM_DATA_in14	0	no	LCD_DATA_out14	1'd1	no
148	CAM_DATA_in15	0	no	LCD_DATA_out15	1'd1	no
149	CAM_PCLK	0	no	CAM_CLK	1'd1	no
150	CAM_H_ENABLE	0	no	LCD_H_ENABLE	1'd1	no
151	CAM_H_SYNC	0	no	LCD_H_SYNC	1'd1	no
152	CAM_V_SYNC	0	no	LCD_V_SYNC	1'd1	no
153	-	-	-	LCD_DC	1'd1	no
154	-	-	-	LCD_PCLK	1'd1	no
155	SUBSPID4_in	0	yes	SUBSPID4_out	SUBSPID4_oe	no
156	SUBSPID5_in	0	yes	SUBSPID5_out	SUBSPID5_oe	no
157	SUBSPID6_in	0	yes	SUBSPID6_out	SUBSPID6_oe	no
158	SUBSPID7_in	0	yes	SUBSPID7_out	SUBSPID7_oe	no
159	SUBSPIDQS_in	0	yes	SUBSPIDQS_out	SUBSPIDQS_oe	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
160	pwm0_sync0_in	0	no	pwm0_out0a	1'd1	no
161	pwm0_sync1_in	0	no	pwm0_out0b	1'd1	no
162	pwm0_sync2_in	0	no	pwm0_out1a	1'd1	no
163	pwm0_f0_in	0	no	pwm0_out1b	1'd1	no
164	pwm0_f1_in	0	no	pwm0_out2a	1'd1	no
165	pwm0_f2_in	0	no	pwm0_out2b	1'd1	no
166	pwm0_cap0_in	0	no	pwm1_out0a	1'd1	no
167	pwm0_cap1_in	0	no	pwm1_out0b	1'd1	no
168	pwm0_cap2_in	0	no	pwm1_out1a	1'd1	no
169	pwm1_sync0_in	0	no	pwm1_out1b	1'd1	no
170	pwm1_sync1_in	0	no	pwm1_out2a	1'd1	no
171	pwm1_sync2_in	0	no	pwm1_out2b	1'd1	no
172	pwm1_f0_in	0	no	sdhost_cclk_out_1	1'd1	no
173	pwm1_f1_in	0	no	sdhost_cclk_out_2	1'd1	no
174	pwm1_f2_in	0	no	sdhost_rst_n_1	1'd1	no
175	pwm1_cap0_in	0	no	sdhost_rst_n_2	1'd1	no
176	pwm1_cap1_in	0	no	sd-host_ccmd_od_pullup_en_n	1'd1	no
177	pwm1_cap2_in	0	no	sdio_tohost_int_out	1'd1	no
178	sdhost_ccmd_in_1	1	no	sdhost_ccmd_out_1	sdhost_ccmd_out_en_1	no
179	sdhost_ccmd_in_2	1	no	sdhost_ccmd_out_2	sdhost_ccmd_out_en_2	no
180	sdhost_cdata_in_10	1	no	sdhost_cdata_out_10	sdhost_cdata_out_en_10	no
181	sdhost_cdata_in_11	1	no	sdhost_cdata_out_11	sdhost_cdata_out_en_11	no
182	sdhost_cdata_in_12	1	no	sdhost_cdata_out_12	sdhost_cdata_out_en_12	no
183	sdhost_cdata_in_13	1	no	sdhost_cdata_out_13	sdhost_cdata_out_en_13	no
184	sdhost_cdata_in_14	1	no	sdhost_cdata_out_14	sdhost_cdata_out_en_14	no
185	sdhost_cdata_in_15	1	no	sdhost_cdata_out_15	sdhost_cdata_out_en_15	no

GoBack

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
186	sdhost_cdata_in_16	1	no	sdhost_cdata_out_16	sdhost_cdata_out_en_16	no
187	sdhost_cdata_in_17	1	no	sdhost_cdata_out_17	sdhost_cdata_out_en_17	no
188	-	-	-	-	1'd1	-
189	-	-	-	-	1'd1	-
190	-	-	-	-	1'd1	-
191	-	-	-	-	1'd1	-
192	sdhost_data_strobe_1	0	no	-	1'd1	-
193	sdhost_data_strobe_2	0	no	-	1'd1	-
194	sdhost_card_detect_n_1	0	no	-	1'd1	-
195	sdhost_card_detect_n_2	0	no	-	1'd1	-
196	sdhost_card_write_prt_1	0	no	-	1'd1	-
197	sdhost_card_write_prt_2	0	no	-	1'd1	-
198	sdhost_card_int_n_1	0	no	-	1'd1	-
199	sdhost_card_int_n_2	0	no	-	1'd1	-
200	-	-	-	-	1'd1	no
201	-	-	-	-	1'd1	no
202	-	-	-	-	1'd1	no
203	-	-	-	-	1'd1	no
204	-	-	-	-	1'd1	no
205	-	-	-	-	1'd1	no
206	-	-	-	-	1'd1	no
207	-	-	-	-	1'd1	no
208	sig_in_func_208	0	no	sig_in_func208	1'd1	no
209	sig_in_func_209	0	no	sig_in_func209	1'd1	no
210	sig_in_func_210	0	no	sig_in_func210	1'd1	no
211	sig_in_func_211	0	no	sig_in_func211	1'd1	no
212	sig_in_func_212	0	no	sig_in_func212	1'd1	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
213	sdhost_cdata_in_20	1	no	sdhost_cdata_out_20	sdhost_cdata_out_en_20	no
214	sdhost_cdata_in_21	1	no	sdhost_cdata_out_21	sdhost_cdata_out_en_21	no
215	sdhost_cdata_in_22	1	no	sdhost_cdata_out_22	sdhost_cdata_out_en_22	no
216	sdhost_cdata_in_23	1	no	sdhost_cdata_out_23	sdhost_cdata_out_en_23	no
217	sdhost_cdata_in_24	1	no	sdhost_cdata_out_24	sdhost_cdata_out_en_24	no
218	sdhost_cdata_in_25	1	no	sdhost_cdata_out_25	sdhost_cdata_out_en_25	no
219	sdhost_cdata_in_26	1	no	sdhost_cdata_out_26	sdhost_cdata_out_en_26	no
220	sdhost_cdata_in_27	1	no	sdhost_cdata_out_27	sdhost_cdata_out_en_27	no
221	pro_alonegpio_in0	0	no	pro_alonegpio_out0	1'd1	no
222	pro_alonegpio_in1	0	no	pro_alonegpio_out1	1'd1	no
223	pro_alonegpio_in2	0	no	pro_alonegpio_out2	1'd1	no
224	pro_alonegpio_in3	0	no	pro_alonegpio_out3	1'd1	no
225	pro_alonegpio_in4	0	no	pro_alonegpio_out4	1'd1	no
226	pro_alonegpio_in5	0	no	pro_alonegpio_out5	1'd1	no
227	pro_alonegpio_in6	0	no	pro_alonegpio_out6	1'd1	no
228	pro_alonegpio_in7	0	no	pro_alonegpio_out7	1'd1	no
229	-	-	-	-	1'd1	-
230	-	-	-	-	1'd1	-
231	-	-	-	-	1'd1	-
232	-	-	-	-	1'd1	-
233	-	-	-	-	1'd1	-
234	-	-	-	-	1'd1	-
235	-	-	-	-	1'd1	-
236	-	-	-	-	1'd1	-
237	-	-	-	-	1'd1	-
238	-	-	-	-	1'd1	-
239	-	-	-	-	1'd1	-

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
240	-	-	-	-	1'd1	-
241	-	-	-	-	1'd1	-
242	-	-	-	-	1'd1	-
243	-	-	-	-	1'd1	-
244	-	-	-	-	1'd1	-
245	-	-	-	-	1'd1	-
246	-	-	-	-	1'd1	-
247	-	-	-	-	1'd1	-
248	-	-	-	-	1'd1	-
249	-	-	-	-	1'd1	-
250	-	-	-	-	1'd1	-
251	usb_jtag_tdo_bridge	0	no	usb_jtag_trst	1'd1	no
252	Core1_gpio_in3	0	no	Core1_gpio_out3	1'd1	no
253	Core1_gpio_in4	0	no	Core1_gpio_out4	1'd1	no
254	Core1_gpio_in5	0	no	Core1_gpio_out5	1'd1	no
255	Core1_gpio_in6	0	no	Core1_gpio_out6	1'd1	no

6.12 IO MUX 管脚功能列表

表 6.12-1 列出了所有 GPIO 管脚的 IO MUX 功能。

表 6.12-1. IO MUX 管脚功能

GPIO	管脚	功能 0	功能 1	功能 2	功能 3	功能 4	DRV	RST	说明
0	GPIO0	GPIO0	GPIO0	-	-	-	2	3	R
1	GPIO1	GPIO1	GPIO1	-	-	-	2	1	R
2	GPIO2	GPIO2	GPIO2	-	-	-	2	1	R
3	GPIO3	GPIO3	GPIO3	-	-	-	2	1	R
4	GPIO4	GPIO4	GPIO4	-	-	-	2	0	R
5	GPIO5	GPIO5	GPIO5	-	-	-	2	0	R
6	GPIO6	GPIO6	GPIO6	-	-	-	2	0	R
7	GPIO7	GPIO7	GPIO7	-	-	-	2	0	R
8	GPIO8	GPIO8	GPIO8	-	SUBSPICS1	-	2	0	R
9	GPIO9	GPIO9	GPIO9	-	SUBSPIHD	FSPIHD	2	1	R
10	GPIO10	GPIO10	GPIO10	FSPIIO4	SUBSPICS0	FSPICS0	2	1	R
11	GPIO11	GPIO11	GPIO11	FSPIIO5	SUBSPID	FSPID	2	1	R
12	GPIO12	GPIO12	GPIO12	FSPIIO6	SUBSPICLK	FSPICLK	2	1	R
13	GPIO13	GPIO13	GPIO13	FSPIIO7	SUBSPIQ	FSPIQ	2	1	R
14	GPIO14	GPIO14	GPIO14	FSPIDQS	SUBSPIWP	FSPIWP	2	1	R
15	XTAL_32K_P	GPIO15	GPIO15	U0RTS	-	-	2	0	R
16	XTAL_32K_N	GPIO16	GPIO16	U0CTS	-	-	2	0	R
17	GPIO17	GPIO17	GPIO17	U1TXD	-	-	2	1	R
18	GPIO18	GPIO18	GPIO18	U1RXD	CLK_OUT3	-	2	1	R
19	GPIO19	GPIO19	GPIO19	U1RTS	CLK_OUT2	-	3	0	R
20	GPIO20	GPIO20	GPIO20	U1CTS	CLK_OUT1	-	3	0	R
21	GPIO21	GPIO21	GPIO21	-	-	-	2	0	R
26	SPICS1	SPICS1	GPIO26	-	-	-	2	3	-
27	SPIHD	SPIHD	GPIO27	-	-	-	2	3	-
28	SPIWP	SPIWP	GPIO28	-	-	-	2	3	-
29	SPICS0	SPICS0	GPIO29	-	-	-	2	3	-
30	SPICLK	SPICLK	GPIO30	-	-	-	2	3	-
31	SPIQ	SPIQ	GPIO31	-	-	-	2	3	-
32	SPID	SPID	GPIO32	-	-	-	2	3	-
33	GPIO33	GPIO33	GPIO33	FSPIHD	SUBSPIHD	SPIIO4	2	1	-
34	GPIO34	GPIO34	GPIO34	FSPICS0	SUBSPICS0	SPIIO5	2	1	-
35	GPIO35	GPIO35	GPIO35	FSPID	SUBSPID	SPIIO6	2	1	-
36	GPIO36	GPIO36	GPIO36	FSPICLK	SUBSPICLK	SPIIO7	2	1	-
37	GPIO37	GPIO37	GPIO37	FSPIQ	SUBSPIQ	SPIDQS	2	1	-
38	GPIO38	GPIO38	GPIO38	FSPIWP	SUBSPIWP	-	2	1	-
39	MTCK	MTCK	GPIO39	CLK_OUT3	SUBSPICS1	-	2	1*	-
40	MTDO	MTDO	GPIO40	CLK_OUT2	-	-	2	1	-
41	MTDI	MTDI	GPIO41	CLK_OUT1	-	-	2	1	-

GPIO	管脚	功能 0	功能 1	功能 2	功能 3	功能 4	DRV	RST	说明
42	MTMS	MTMS	GPIO42	-	-	-	2	1	-
43	U0TXD	U0TXD	GPIO43	CLK_OUT1	-	-	2	4	-
44	U0RXD	U0RXD	GPIO44	CLK_OUT2	-	-	2	3	-
45	GPIO45	GPIO45	GPIO45	-	-	-	2	2	-
46	GPIO46	GPIO46	GPIO46	-	-	-	2	2	-
47	SPICLK_P	SPICLK_P_DIFF	GPIO47	SUBSPICLK_P_DIFF	-	-	2	1	-
48	SPICLK_N	SPICLK_N_DIFF	GPIO48	SUBSPICLK_N_DIFF	-	-	2	1	-

驱动强度

“DRV” 一栏所示为每个管脚复位后的默认驱动强度。

- GPIO17 和 GPIO18
  - **0** - 驱动电流 = ~5 mA
  - **1** - 驱动电流 = ~20 mA
  - **2** - 驱动电流 = ~10 mA
  - **3** - 驱动电流 = ~40 mA
- 其他 GPIO
  - **0** - 驱动电流 = ~5 mA
  - **1** - 驱动电流 = ~10 mA
  - **2** - 驱动电流 = ~20 mA
  - **3** - 驱动电流 = ~40 mA

复位配置

“RST” 一栏是每个管脚复位后的默认配置。

- **0** - IE = 0（输入关闭）
- **1** - IE = 1（输入使能）
- **2** - IE = 1，WPD = 1（输入使能，下拉电阻使能）
- **3** - IE = 1，WPU = 1（输入使能，上拉电阻使能）
- **4** - OE = 1，WPU = 1（输出使能，上拉电阻使能）
- **1*** - 如果 EFUSE_DIS_PAD_JTAG = 1，则 MTCK 管脚复位后浮空，即 IE = 1。如果 EFUSE_DIS_PAD_JTAG = 0，则 MTCK 复位之后连接内部上拉电阻，即 IE = 1，WPU = 1。

说明

- **R** - 管脚通过 RTC IO MUX 具有 RTC/模拟功能。

6.13 RTC IO MUX 管脚功能列表

表 6.13-1 列出了 RTC 管脚和对应 GPIO 管脚及 RTC 功能。

表 6.13-1. RTC IO MUX 管脚的 RTC 功能

RTC GPIO No.	GPIO No.	管脚	RTC 功能			
			0	1	2	3
0	0	GPIO0	RTC_GPIO0	-	-	sar_i2c_scl_0[a]
1	1	GPIO1	RTC_GPIO1	-	-	sar_i2c_sda_0[a]
2	2	GPIO2	RTC_GPIO2	-	-	sar_i2c_scl_1[a]
3	3	GPIO3	RTC_GPIO3	-	-	sar_i2c_sda_1[a]
4	4	GPIO4	RTC_GPIO4	-	-	-
5	5	GPIO5	RTC_GPIO5	-	-	-
6	6	GPIO6	RTC_GPIO6	-	-	-
7	7	GPIO7	RTC_GPIO7	-	-	-
8	8	GPIO8	RTC_GPIO8	-	-	-
9	9	GPIO9	RTC_GPIO9	-	-	-
10	10	GPIO10	RTC_GPIO10	-	-	-
11	11	GPIO11	RTC_GPIO11	-	-	-
12	12	GPIO12	RTC_GPIO12	-	-	-
13	13	GPIO13	RTC_GPIO13	-	-	-
14	14	GPIO14	RTC_GPIO14	-	-	-
15	15	XTAL_32K_P	RTC_GPIO15	-	-	-
16	16	XTAL_32K_N	RTC_GPIO16	-	-	-
17	17	GPIO17	RTC_GPIO17	-	-	-
18	18	GPIO18	RTC_GPIO18	-	-	-
19	19	GPIO19	RTC_GPIO19	-	-	-
20	20	GPIO20	RTC_GPIO20	-	-	-
21	21	GPIO21	RTC_GPIO21	-	-	-

[a] 有关 sar_i2c_xx 的配置信息，请参考章节 *2 超低功耗协处理器 (ULP-FSM, ULP-RISC-V)*: RTC I2C 控制器。

表 6.13-2 列出了 RTC 管脚和对应 GPIO 管脚及模拟功能。

表 6.13-2. RTC IO MUX 管脚模拟功能

RTC GPIO No.	GPIO No.	管脚	模拟功能	
			0	1
0	0	GPIO0	-	-
1	1	GPIO1	TOUCH1	ADC1_CH0
2	2	GPIO2	TOUCH2	ADC1_CH1
3	3	GPIO3	TOUCH3	ADC1_CH2
4	4	GPIO4	TOUCH4	ADC1_CH3
5	5	GPIO5	TOUCH5	ADC1_CH4
6	6	GPIO6	TOUCH6	ADC1_CH5
7	7	GPIO7	TOUCH7	ADC1_CH6
8	8	GPIO8	TOUCH8	ADC1_CH7
9	9	GPIO9	TOUCH9	ADC1_CH8

RTC GPIO No.	GPIO No.	管脚	模拟功能	
			0	1
10	10	GPIO10	TOUCH10	ADC1_CH9
11	11	GPIO11	TOUCH11	ADC2_CH0
12	12	GPIO12	TOUCH12	ADC2_CH1
13	13	GPIO13	TOUCH13	ADC2_CH2
14	14	GPIO14	TOUCH14	ADC2_CH3
15	15	XTAL_32K_P	XTAL_32K_P	ADC2_CH4
16	16	XTAL_32K_N	XTAL_32K_N	ADC2_CH5
17	17	GPIO17	-	ADC2_CH6
18	18	GPIO18	-	ADC2_CH7
19	19	GPIO19	USB_D-	ADC2_CH8
20	20	GPIO20	USB_D+	ADC2_CH9
21	21	GPIO21	-	-

6.14 寄存器列表

6.14.1 GPIO 交换矩阵寄存器列表

本小节的所有地址均为相对于 GPIO 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
GPIO 配置寄存器			
GPIO_BT_SELECT_REG	GPIO 位选择寄存器	0x0000	读/写
GPIO_OUT_REG	GPIO0 ~ 31 输出寄存器	0x0004	读/写
GPIO_OUT_W1TS_REG	GPIO0 ~ 31 输出置位寄存器	0x0008	只写
GPIO_OUT_W1TC_REG	GPIO0 ~ 31 输出清零寄存器	0x000C	只写
GPIO_OUT1_REG	GPIO32 ~ 48 输出寄存器	0x0010	读/写
GPIO_OUT1_W1TS_REG	GPIO32 ~ 48 输出置位寄存器	0x0014	只写
GPIO_OUT1_W1TC_REG	GPIO32 ~ 48 输出清零寄存器	0x0018	只写
GPIO_SDIO_SELECT_REG	GPIO SDIO 选择寄存器	0x001C	读/写
GPIO_ENABLE_REG	GPIO0 ~ 31 输出使能寄存器	0x0020	读/写
GPIO_ENABLE_W1TS_REG	GPIO0 ~ 31 输出使能置位寄存器	0x0024	只写
GPIO_ENABLE_W1TC_REG	GPIO0 ~ 31 输出使能清零寄存器	0x0028	只写
GPIO_ENABLE1_REG	GPIO32 ~ 48 输出使能寄存器	0x002C	读/写
GPIO_ENABLE1_W1TS_REG	GPIO32 ~ 48 输出使能置位寄存器	0x0030	只写
GPIO_ENABLE1_W1TC_REG	GPIO32 ~ 48 输出使能清零寄存器	0x0034	只写
GPIO_STRAP_REG	Strapping 管脚寄存器	0x0038	只读
GPIO_IN_REG	GPIO0 ~ 31 输入寄存器	0x003C	只读
GPIO_IN1_REG	GPIO32 ~ 48 输入寄存器	0x0040	只读
GPIO_PIN0_REG	配置 GPIO pin 0	0x0074	读/写
GPIO_PIN1_REG	配置 GPIO pin 1	0x0078	读/写

名称	描述	地址	访问
GPIO_PIN2_REG	配置 GPIO pin 2	0x007C	读/写
...	...	...	...
GPIO_PIN46_REG	配置 GPIO pin 46	0x012C	读/写
GPIO_PIN47_REG	配置 GPIO pin 47	0x0130	读/写
GPIO_PIN48_REG	配置 GPIO pin 48	0x0134	读/写
GPIO_FUNC0_IN_SEL_CFG_REG	外设信号 0 的输入选择寄存器	0x0154	读/写
GPIO_FUNC1_IN_SEL_CFG_REG	外设信号 1 的输入选择寄存器	0x0158	读/写
GPIO_FUNC2_IN_SEL_CFG_REG	外设信号 2 的输入选择寄存器	0x015C	读/写
...	...	...	...
GPIO_FUNC253_IN_SEL_CFG_REG	外设信号 253 的输入选择寄存器	0x0548	读/写
GPIO_FUNC254_IN_SEL_CFG_REG	外设信号 254 的输入选择寄存器	0x054C	读/写
GPIO_FUNC255_IN_SEL_CFG_REG	外设信号 255 的输入选择寄存器	0x0550	读/写
GPIO_FUNC0_OUT_SEL_CFG_REG	GPIO0 的外设输出信号选择寄存器	0x0554	读/写
GPIO_FUNC1_OUT_SEL_CFG_REG	GPIO1 的外设输出信号选择寄存器	0x0558	读/写
GPIO_FUNC2_OUT_SEL_CFG_REG	GPIO2 的外设输出信号选择寄存器	0x055C	读/写
...	...	...	...
GPIO_FUNC47_OUT_SEL_CFG_REG	GPIO47 的外设输出信号选择寄存器	0x0610	读/写
GPIO_FUNC48_OUT_SEL_CFG_REG	GPIO48 的外设输出信号选择寄存器	0x0614	读/写
GPIO_CLOCK_GATE_REG	GPIO 时钟门控寄存器	0x062C	读/写
中断状态寄存器			
GPIO_STATUS_REG	GPIO0 ~ 31 中断状态寄存器	0x0044	读/写
GPIO_STATUS1_REG	GPIO32 ~ 48 中断状态寄存器	0x0050	读/写
GPIO_CPU_INT_REG	GPIO0 ~ 31 CPU 中断状态寄存器	0x005C	只读
GPIO_CPU_NMI_INT_REG	GPIO0 ~ 31 CPU 非屏蔽中断状态寄存器	0x0060	只读
GPIO_CPU_INT1_REG	GPIO32 ~ 48 CPU 中断状态寄存器	0x0068	只读
GPIO_CPU_NMI_INT1_REG	GPIO32 ~ 48 CPU 非屏蔽状态寄存器	0x006C	只读
中断配置寄存器			
GPIO_STATUS_W1TS_REG	GPIO0 ~ 31 中断状态置位寄存器	0x0048	只写
GPIO_STATUS_W1TC_REG	GPIO0 ~ 31 中断状态清零寄存器	0x004C	只写
GPIO_STATUS1_W1TS_REG	GPIO32 ~ 48 中断状态置位寄存器	0x0054	只写
GPIO_STATUS1_W1TC_REG	GPIO32 ~ 48 中断状态清零寄存器	0x0058	只写
GPIO 中断源寄存器			
GPIO_STATUS_NEXT_REG	GPIO0 ~ 31 中断源寄存器	0x014C	只读
GPIO_STATUS_NEXT1_REG	GPIO32 ~ 48 中断源寄存器	0x0150	只读
版本寄存器			
GPIO_DATE_REG	版本控制寄存器	0x06FC	读/写

6.14.2 IO MUX 寄存器列表

本小节的所有地址均为相对于 IO MUX 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
IO_MUX_PIN_CTRL_REG	时钟输出配置寄存器	0x0000	读/写
IO_MUX_GPIO0_REG	GPIO0 配置寄存器	0x0004	读/写
IO_MUX_GPIO1_REG	GPIO1 配置寄存器	0x0008	读/写
IO_MUX_GPIO2_REG	GPIO2 配置寄存器	0x000C	读/写
IO_MUX_GPIO3_REG	GPIO3 配置寄存器	0x0010	读/写
IO_MUX_GPIO4_REG	GPIO4 配置寄存器	0x0014	读/写
IO_MUX_GPIO5_REG	GPIO5 配置寄存器	0x0018	读/写
IO_MUX_GPIO6_REG	GPIO6 配置寄存器	0x001C	读/写
IO_MUX_GPIO7_REG	GPIO7 配置寄存器	0x0020	读/写
IO_MUX_GPIO8_REG	GPIO8 配置寄存器	0x0024	读/写
IO_MUX_GPIO9_REG	GPIO9 配置寄存器	0x0028	读/写
IO_MUX_GPIO10_REG	GPIO10 配置寄存器	0x002C	读/写
IO_MUX_GPIO11_REG	GPIO11 配置寄存器	0x0030	读/写
IO_MUX_GPIO12_REG	GPIO12 配置寄存器	0x0034	读/写
IO_MUX_GPIO13_REG	GPIO13 配置寄存器	0x0038	读/写
IO_MUX_GPIO14_REG	GPIO14 配置寄存器	0x003C	读/写
IO_MUX_GPIO15_REG	GPIO15 配置寄存器	0x0040	读/写
IO_MUX_GPIO16_REG	GPIO16 配置寄存器	0x0044	读/写
IO_MUX_GPIO17_REG	GPIO17 配置寄存器	0x0048	读/写
IO_MUX_GPIO18_REG	GPIO18 配置寄存器	0x004C	读/写
IO_MUX_GPIO19_REG	GPIO19 配置寄存器	0x0050	读/写
IO_MUX_GPIO20_REG	GPIO20 配置寄存器	0x0054	读/写
IO_MUX_GPIO21_REG	GPIO21 配置寄存器	0x0058	读/写
IO_MUX_GPIO26_REG	GPIO26 配置寄存器	0x006C	读/写
IO_MUX_GPIO27_REG	GPIO27 配置寄存器	0x0070	读/写
IO_MUX_GPIO28_REG	GPIO28 配置寄存器	0x0074	读/写
IO_MUX_GPIO29_REG	GPIO29 配置寄存器	0x0078	读/写
IO_MUX_GPIO30_REG	GPIO30 配置寄存器	0x007C	读/写
IO_MUX_GPIO31_REG	GPIO31 配置寄存器	0x0080	读/写
IO_MUX_GPIO32_REG	GPIO32 配置寄存器	0x0084	读/写
IO_MUX_GPIO33_REG	GPIO33 配置寄存器	0x0088	读/写
IO_MUX_GPIO34_REG	GPIO34 配置寄存器	0x008C	读/写
IO_MUX_GPIO35_REG	GPIO35 配置寄存器	0x0090	读/写
IO_MUX_GPIO36_REG	GPIO36 配置寄存器	0x0094	读/写
IO_MUX_GPIO37_REG	GPIO37 配置寄存器	0x0098	读/写
IO_MUX_GPIO38_REG	GPIO38 配置寄存器	0x009C	读/写
IO_MUX_GPIO39_REG	GPIO39 配置寄存器	0x00A0	读/写
IO_MUX_GPIO40_REG	GPIO40 配置寄存器	0x00A4	读/写
IO_MUX_GPIO41_REG	GPIO41 配置寄存器	0x00A8	读/写
IO_MUX_GPIO42_REG	GPIO42 配置寄存器	0x00AC	读/写
IO_MUX_GPIO43_REG	GPIO43 配置寄存器	0x00B0	读/写
IO_MUX_GPIO44_REG	GPIO44 配置寄存器	0x00B4	读/写
IO_MUX_GPIO45_REG	GPIO45 配置寄存器	0x00B8	读/写

名称	描述	地址	访问
IO_MUX_GPIO46_REG	GPIO46 配置寄存器	0x00BC	读/写
IO_MUX_GPIO47_REG	GPIO47 配置寄存器	0x00C0	读/写
IO_MUX_GPIO48_REG	GPIO48 配置寄存器	0x00C4	读/写

6.14.3 SDM 寄存器列表

本小节的所有地址均为相对于 GPIO 基地址 + 0x0F00 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	权限
配置寄存器			
GPIO_SIGMADELTA0_REG	SDM0 占空比配置寄存器	0x0000	读/写
GPIO_SIGMADELTA1_REG	SDM1 占空比配置寄存器	0x0004	读/写
GPIO_SIGMADELTA2_REG	SDM2 占空比配置寄存器	0x0008	读/写
GPIO_SIGMADELTA3_REG	SDM3 占空比配置寄存器	0x000C	读/写
GPIO_SIGMADELTA4_REG	SDM4 占空比配置寄存器	0x0010	读/写
GPIO_SIGMADELTA5_REG	SDM5 占空比配置寄存器	0x0014	读/写
GPIO_SIGMADELTA6_REG	SDM6 占空比配置寄存器	0x0018	读/写
GPIO_SIGMADELTA7_REG	SDM7 占空比配置寄存器	0x001C	读/写
GPIO_SIGMADELTA_CG_REG	时钟门控配置寄存器	0x0020	读/写
GPIO_SIGMADELTA_MISC_REG	MISC 寄存器	0x0024	读/写
GPIO_SIGMADELTA_VERSION_REG	版本控制寄存器	0x0028	读/写

6.14.4 RTC IO MUX 寄存器列表

本小节的所有地址均为相对于低功耗管理模块基地址 + 0x0400 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	权限
GPIO 配置/数据寄存器			
RTC_GPIO_OUT_REG	RTC GPIO 输出寄存器	0x0000	读/写
RTC_GPIO_OUT_W1TS_REG	RTC GPIO 输出置位寄存器	0x0004	只写
RTC_GPIO_OUT_W1TC_REG	RTC GPIO 输出置位清零寄存器	0x0008	只写
RTC_GPIO_ENABLE_REG	RTC GPIO 输出使能寄存器	0x000C	读/写
RTC_GPIO_ENABLE_W1TS_REG	RTC GPIO 输出使能置位寄存器	0x0010	只写
RTC_GPIO_ENABLE_W1TC_REG	RTC GPIO 输出使能清零寄存器	0x0014	只写
RTC_GPIO_STATUS_REG	RTC GPIO 中断状态寄存器	0x0018	读/写
RTC_GPIO_STATUS_W1TS_REG	RTC GPIO 中断状态置位寄存器	0x001C	只写
RTC_GPIO_STATUS_W1TC_REG	RTC GPIO 中断状态清零寄存器	0x0020	只写
RTC_GPIO_IN_REG	RTC GPIO 输入寄存器	0x0024	只读
RTC_GPIO_PIN0_REG	Pin0 RTC 配置	0x0028	读/写

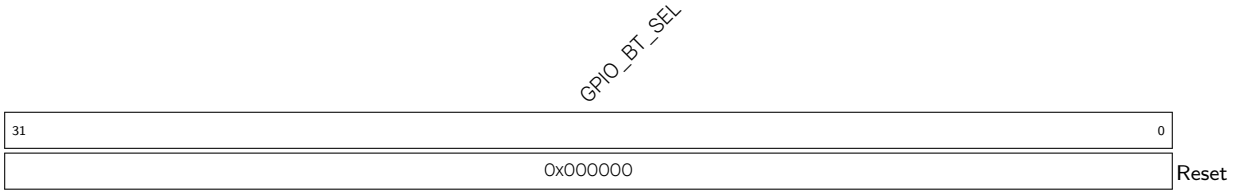
名称	描述	地址	权限
RTC_GPIO_PIN1_REG	Pin1 RTC 配置	0x002C	读/写
RTC_GPIO_PIN2_REG	Pin2 RTC 配置	0x0030	读/写
RTC_GPIO_PIN3_REG	Pin3 RTC 配置	0x0034	读/写
...	...	...	...
RTC_GPIO_PIN19_REG	Pin19 RTC 配置	0x0074	读/写
RTC_GPIO_PIN20_REG	Pin20 RTC 配置	0x0078	读/写
RTC_GPIO_PIN21_REG	Pin21 RTC 配置	0x007C	读/写
GPIO RTC 功能配置寄存器			
RTC_IO_TOUCH_PAD0_REG	Touch pad 0 配置寄存器	0x0084	读/写
RTC_IO_TOUCH_PAD1_REG	Touch pad 1 配置寄存器	0x0088	读/写
RTC_IO_TOUCH_PAD2_REG	Touch pad 2 配置寄存器	0x008C	读/写
RTC_IO_TOUCH_PAD3_REG	Touch pad 3 配置寄存器	0x0090	读/写
RTC_IO_TOUCH_PAD4_REG	Touch pad 4 配置寄存器	0x0094	读/写
RTC_IO_TOUCH_PAD5_REG	Touch pad 5 配置寄存器	0x0098	读/写
RTC_IO_TOUCH_PAD6_REG	Touch pad 6 配置寄存器	0x009C	读/写
RTC_IO_TOUCH_PAD7_REG	Touch pad 7 配置寄存器	0x00A0	读/写
RTC_IO_TOUCH_PAD8_REG	Touch pad 8 配置寄存器	0x00A4	读/写
RTC_IO_TOUCH_PAD9_REG	Touch pad 9 配置寄存器	0x00A8	读/写
RTC_IO_TOUCH_PAD10_REG	Touch pad 10 配置寄存器	0x00AC	读/写
RTC_IO_TOUCH_PAD11_REG	Touch pad 11 配置寄存器	0x00B0	读/写
RTC_IO_TOUCH_PAD12_REG	Touch pad 12 配置寄存器	0x00B4	读/写
RTC_IO_TOUCH_PAD13_REG	Touch pad 13 配置寄存器	0x00B8	读/写
RTC_IO_TOUCH_PAD14_REG	Touch pad 14 配置寄存器	0x00BC	读/写
RTC_IO_XTAL_32P_PAD_REG	32 kHz crystal P-pad 配置寄存器	0x00C0	读/写
RTC_IO_XTAL_32N_PAD_REG	32 kHz crystal N-pad 配置寄存器	0x00C4	读/写
RTC_IO_RTC_PAD17_REG	管脚 17 的配置寄存器	0x00C8	读/写
RTC_IO_RTC_PAD18_REG	管脚 17 的配置寄存器	0x00CC	读/写
RTC_IO_RTC_PAD19_REG	管脚 19 的配置寄存器	0x00D0	读/写
RTC_IO_RTC_PAD20_REG	管脚 20 的配置寄存器	0x00D4	读/写
RTC_IO_RTC_PAD21_REG	管脚 21 的配置寄存器	0x00D8	读/写
RTC_IO_XTL_EXT_CTR_REG	晶振断电 GPIO 使能源	0x00E0	读/写
RTC_IO_SAR_I2C_IO_REG	RTC I2C Pad 选择寄存器	0x00E4	读/写
版本寄存器			
RTC_IO_DATE_REG	版本控制寄存器	0x01FC	读/写

6.15 寄存器

6.15.1 GPIO 交换矩阵寄存器

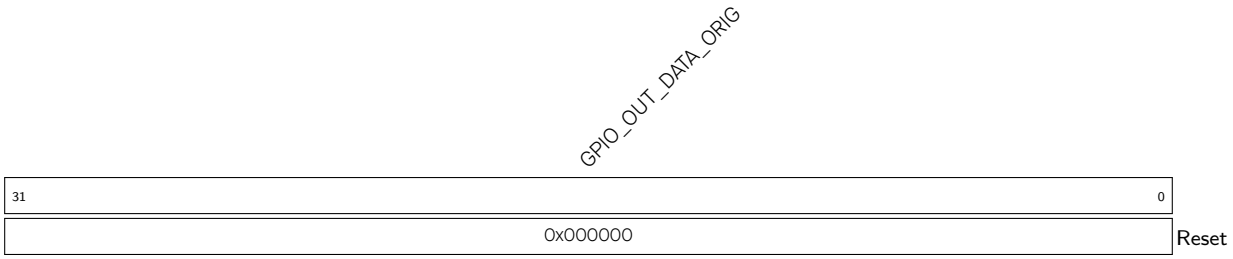
本小节的所有地址均为相对于 GPIO 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 6.1. GPIO_BT_SELECT_REG (0x0000)



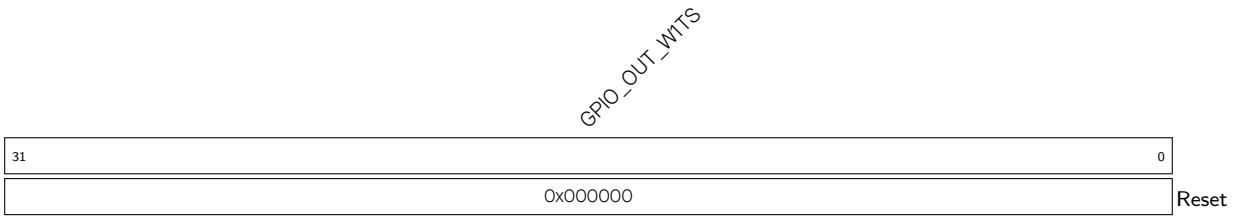
GPIO_BT_SEL 保留。(R/W)

Register 6.2. GPIO_OUT_REG (0x0004)



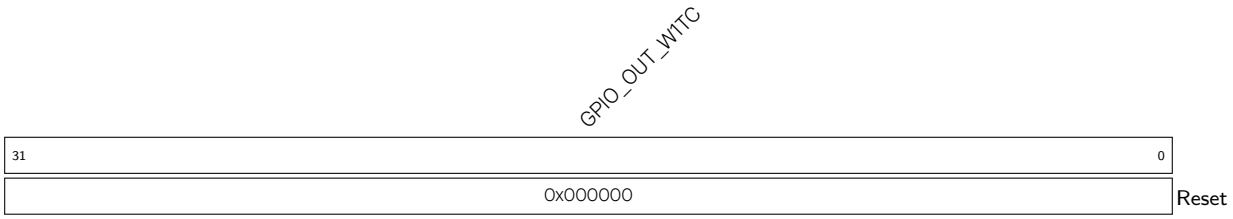
GPIO_OUT_DATA_ORIG 简单 GPIO 输出模式下，GPIO0 ~ 21 和 GPIO26 ~31 的输出值。bit0 ~ bit21 的值分别对应 GPIO0 ~ GPIO21 的输出值；bit26 ~ bit31 的值分别对应 GPIO26 ~ GPIO31 的输出值。bit22 ~ bit25 无效。(R/W)

Register 6.3. GPIO_OUT_W1TS_REG (0x0008)



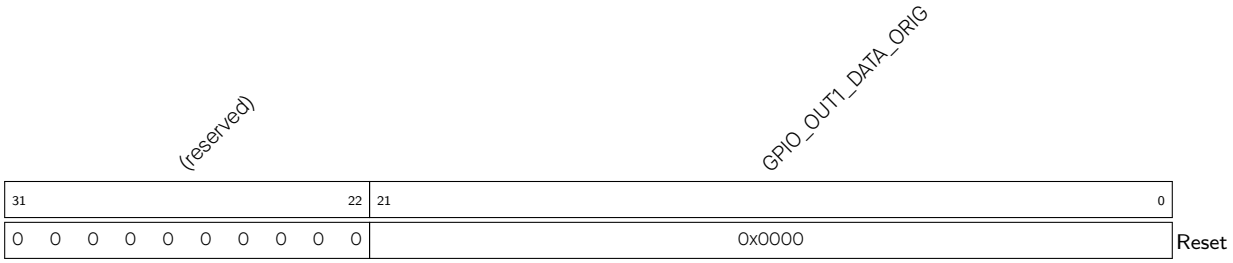
GPIO_OUT_W1TS GPIO0 ~ 31 输出置位寄存器。每一位置 1，GPIO_OUT_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_OUT_REG。(WO)

Register 6.4. GPIO_OUT_W1TC_REG (0x000C)



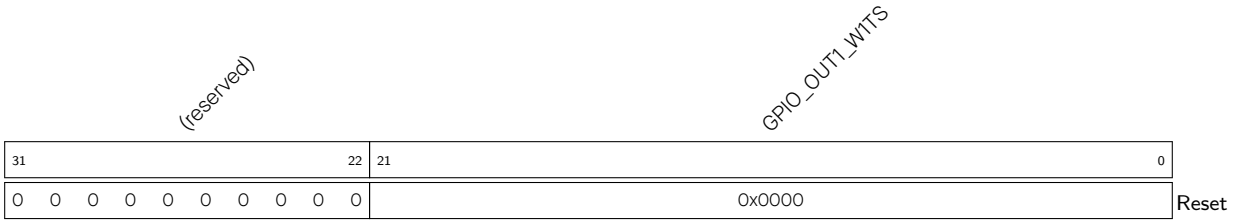
GPIO_OUT_W1TC GPIO0 ~ 31 输出清零寄存器。每一位置 1，则 GPIO_OUT_REG 中的相应位会清零。注：推荐使用此寄存器来清零 GPIO_OUT_REG。(WO)

Register 6.5. GPIO_OUT1_REG (0x0010)



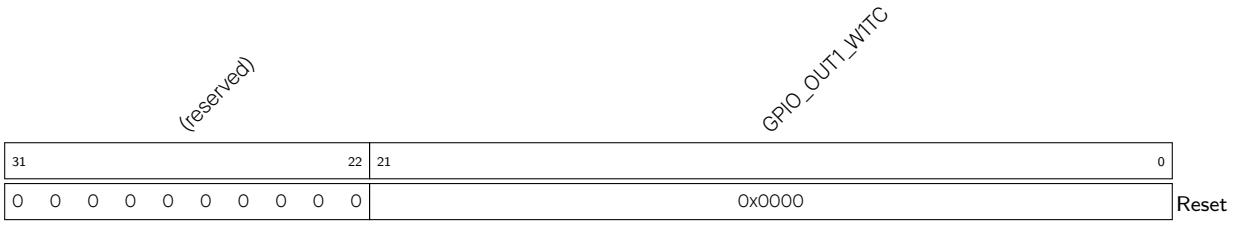
GPIO_OUT1_DATA_ORIG 简单 GPIO 输出模式下，GPIO32 ~ 48 的输出值。bit0 ~ bit16 的值分别对应 GPIO32 ~ GPIO48 的输出值。bit17 ~ bit21 无效。(R/W)

Register 6.6. GPIO_OUT1_W1TS_REG (0x0014)



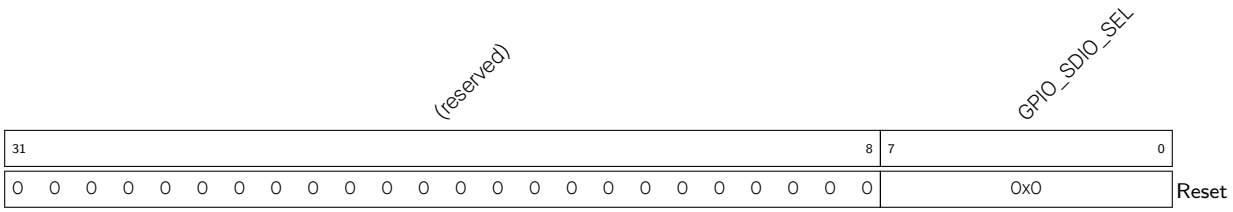
GPIO_OUT1_W1TS GPIO32 ~ 48 输出置位寄存器。每一位置 1，则 GPIO_OUT1_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_OUT1_REG。(WO)

Register 6.7. GPIO_OUT1_W1TC_REG (0x0018)



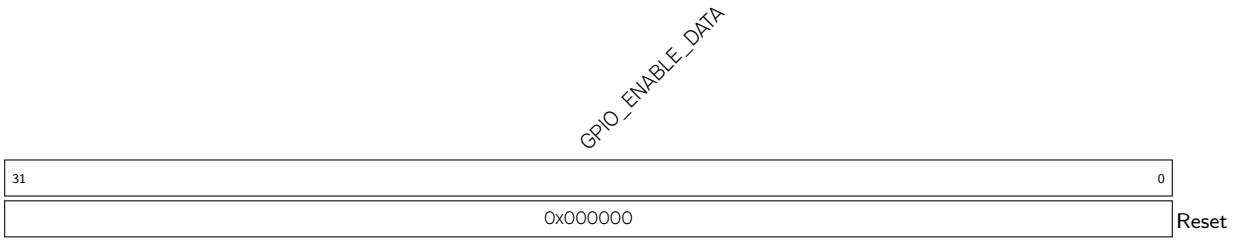
GPIO_OUT1_W1TC GPIO32 ~ 48 输出清零寄存器。每一位置 1，则 GPIO_OUT1_REG 中的相应位会清零。注：推荐使用此寄存器来清零 GPIO_OUT1_REG。(WO)

Register 6.8. GPIO_SDIO_SELECT_REG (0x001C)



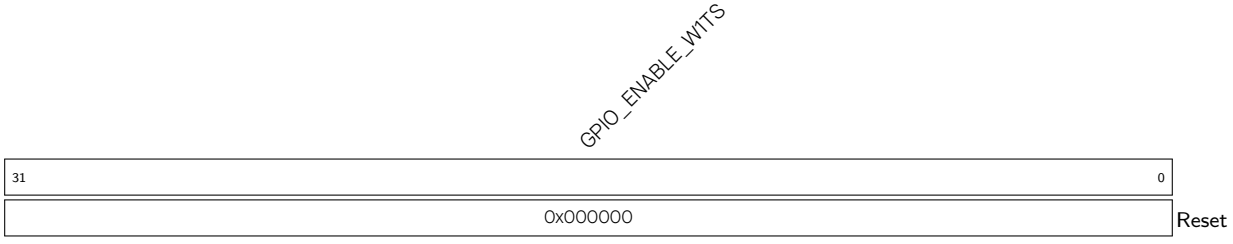
GPIO_SDIO_SEL 保留。(R/W)

Register 6.9. GPIO_ENABLE_REG (0x0020)



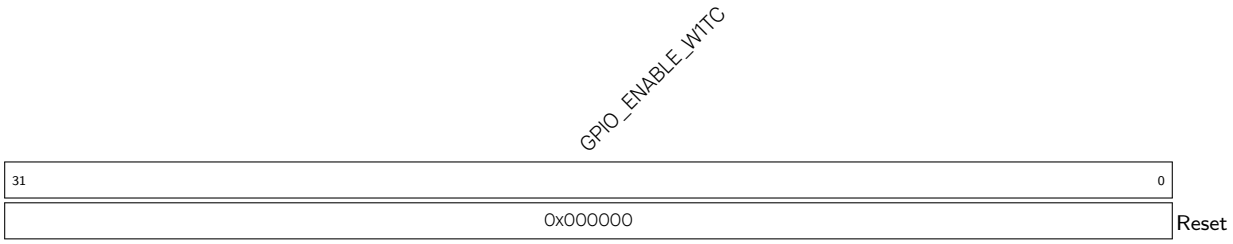
GPIO_ENABLE_DATA GPIO0 ~ 31 输出使能寄存器。(R/W)

Register 6.10. GPIO_ENABLE_W1TS_REG (0x0024)



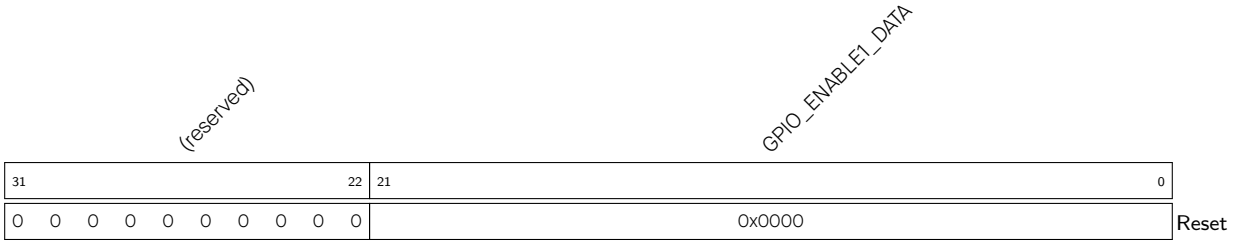
GPIO_ENABLE_W1TS GPIO0 ~ 31 输出使能置位寄存器。每一位置 1，则 GPIO_ENABLE_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_ENABLE_REG。(WO)

Register 6.11. GPIO_ENABLE_W1TC_REG (0x0028)



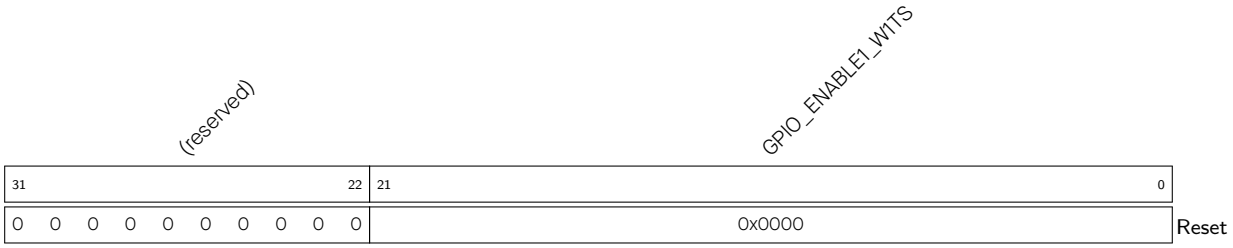
GPIO_ENABLE_W1TC GPIO0 ~ 31 输出使能清零寄存器。每一位置 1，则 GPIO_ENABLE_REG 中的相应位会清零。注：推荐使用此寄存器清零 GPIO_ENABLE_REG。(WO)

Register 6.12. GPIO_ENABLE1_REG (0x002C)



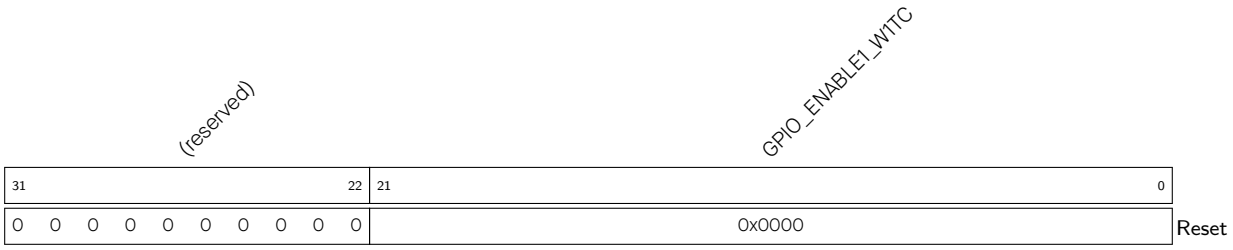
GPIO_ENABLE1_DATA GPIO32 ~ 48 输出使能寄存器。(R/W)

Register 6.13. GPIO_ENABLE1_W1TS_REG (0x0030)



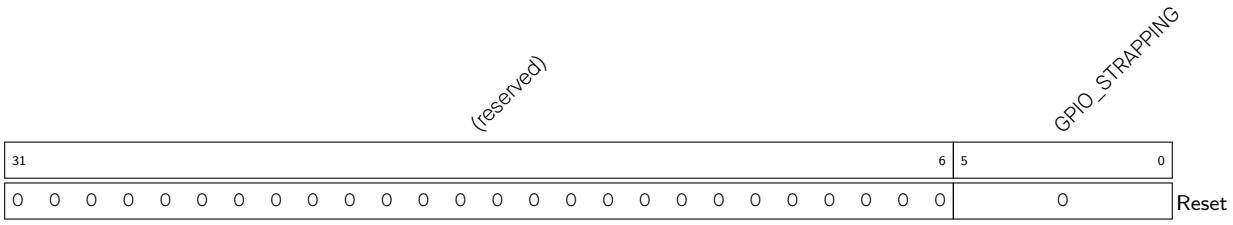
GPIO_ENABLE1_W1TS GPIO32 ~ 48 输出使能置位寄存器。每一位置 1，则 GPIO_ENABLE1_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_ENABLE1_REG。(WO)

Register 6.14. GPIO_ENABLE1_W1TC_REG (0x0034)



GPIO_ENABLE1_W1TC GPIO32 ~ 48 输出使能清零寄存器。每一位置 1，则 GPIO_ENABLE1_REG 中的相应位会清零。注：推荐使用此寄存器清零 GPIO_ENABLE1_REG。(WO)

Register 6.15. GPIO_STRAP_REG (0x0038)



GPIO_STRAPPING 表示对应 GPIO 管脚的 Strapping 值。

bit0 ~ bit1：保留

bit2：GPIO46

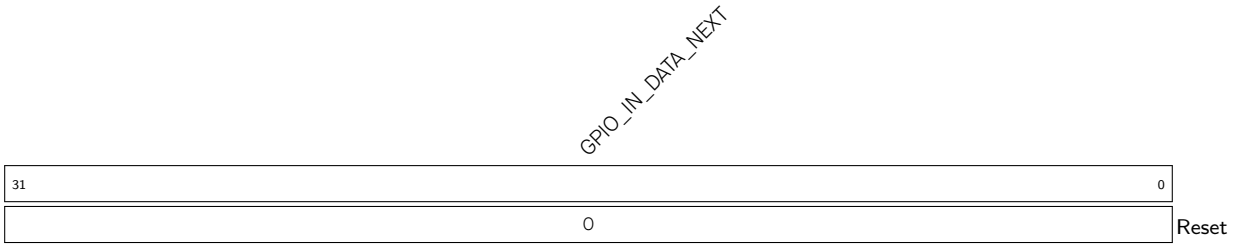
bit3：GPIO0

bit4：GPIO45

bit5：GPIO3

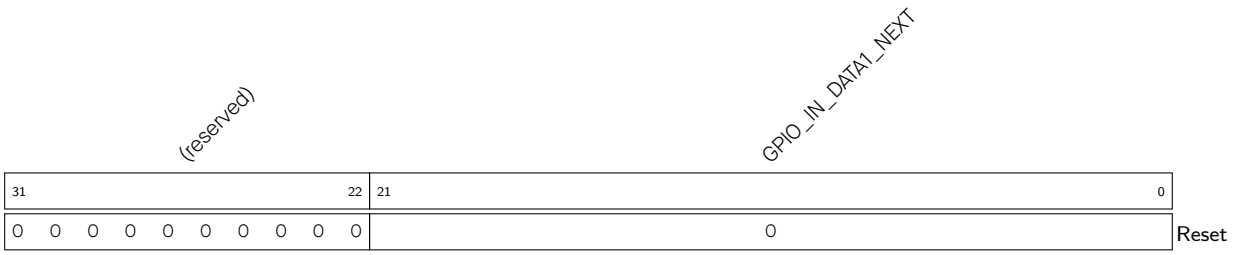
(RO)

Register 6.16. GPIO_IN_REG (0x003C)



GPIO_IN_DATA_NEXT GPIO0 ~ 31 输入值。每一位代表一个管脚的片外输入值。比如片外引脚为高电平，则此位应为 1；片外引脚为低电平，此位应为 0。(RO)

Register 6.17. GPIO_IN1_REG (0x0040)



GPIO_IN_DATA1_NEXT GPIO32 ~ 48 输入值。每一位代表一个管脚的片外输入值。(RO)

Register 6.18. GPIO_PIN*n*_REG (*n*: 0-48) (0x0074+0x4**n*)



GPIO_PIN*n*_SYNC2_BYPASS 使能 GPIO 输入信号第二拍为 APB 时钟上升沿或下降沿同步。0：关闭同步；1：下降沿同步；2 或 3：上升沿同步。(R/W)

GPIO_PIN*n*_PAD_DRIVER 管脚驱动选择。0：正常输出；1：开漏输出。(R/W)

GPIO_PIN*n*_SYNC1_BYPASS 使能 GPIO 输入信号第一拍为 APB 时钟上升沿或下降沿同步。0：关闭同步；1：下降沿同步；2 或 3：上升沿同步。(R/W)

GPIO_PIN*n*_INT_TYPE 中断类型选择。(R/W)

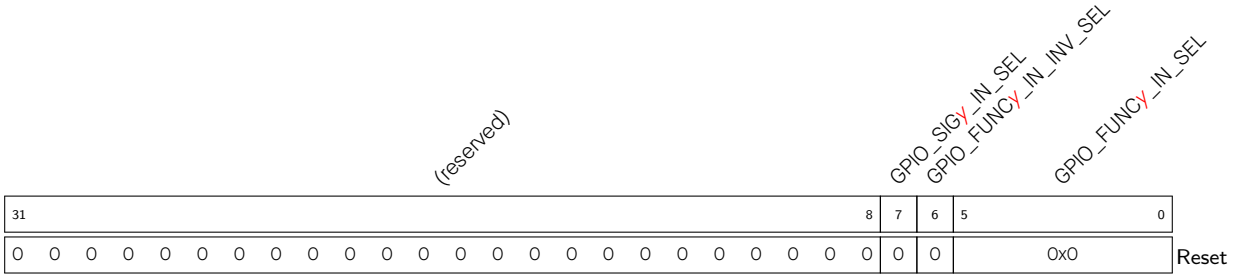
0：禁用 GPIO 中断
1：上升沿触发
2：下降沿触发
3：任一沿触发
4：低电平触发
5：高电平触发

GPIO_PIN*n*_WAKEUP_ENABLE 使能 GPIO 唤醒，仅能将 CPU 从 Light-sleep 模式唤醒。(R/W)

GPIO_PIN*n*_CONFIG 保留。(R/W)

GPIO_PIN*n*_INT_ENA 中断使能位。bit13：使能 CPU 中断；bit14：使能 CPU 非屏蔽中断。(R/W)

Register 6.19. GPIO_FUNC*y*_IN_SEL_CFG_REG (*y*: 0-255) (0x0154+0x4**y*)

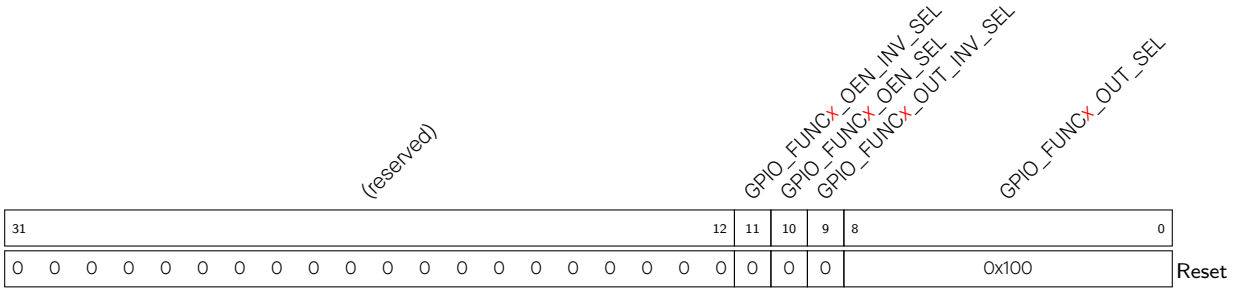


GPIO_FUNC*y*_IN_SEL 外设输入信号 *Y* 的选择控制位。此位选择 1 个 GPIO 交换矩阵输入管脚与信号连接；或者选择 0x38，则输入信号恒为高电平；或者选择 0x3C，则输入信号恒为低电平。(R/W)

GPIO_FUNC*y*_IN_INV_SEL 反转输入值。1：反转；0：不反转。(R/W)

GPIO_SIG*y*_IN_SEL 旁路 GPIO 交换矩阵。1：通过 GPIO 交换矩阵；0：直接通过 IO MUX 连接信号与外设。(R/W)

Register 6.20. GPIO_FUNC*x*_OUT_SEL_CFG_REG (*x*: 0-48) (0x0554+0x4**x*)



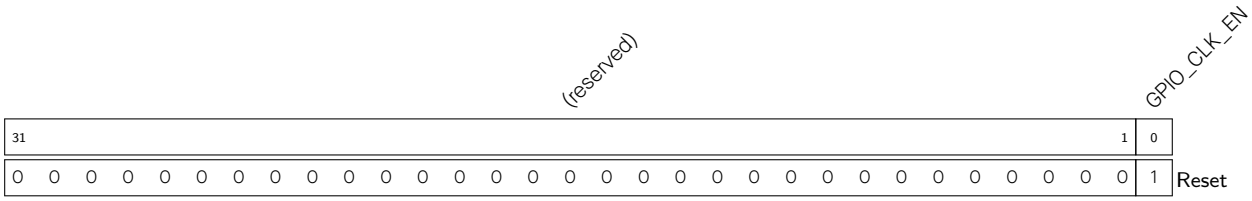
GPIO_FUNC*x*_OUT_SEL GPIO 管脚 *X* 的输出信号选择控制位。值为 *y* (0<=*y*<256) 连接外设输出 *y* 与 GPIO 输出 *x*。值为 256 选择 GPIO_OUT_REG/GPIO_OUT1_REG[*x*] 和 GPIO_ENABLE_REG/GPIO_ENABLE1_REG [*x*] 作为输出值和输出使能。(R/W)

GPIO_FUNC*x*_OUT_INV_SEL 0：不反转输出值；1：反转输出值。(R/W)

GPIO_FUNC*x*_OEN_SEL 0：采用外设的输出使能信号；1：强制使用 GPIO_ENABLE_REG[*x*] 用作输出使能信号。(R/W)

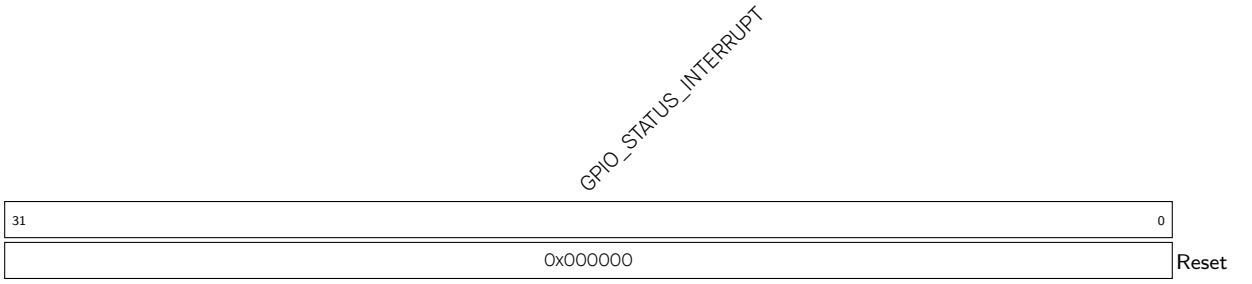
GPIO_FUNC*x*_OEN_INV_SEL 0：不反转输出使能信号；1：反转输出使能信号。(R/W)

Register 6.21. GPIO_CLOCK_GATE_REG (0x062C)



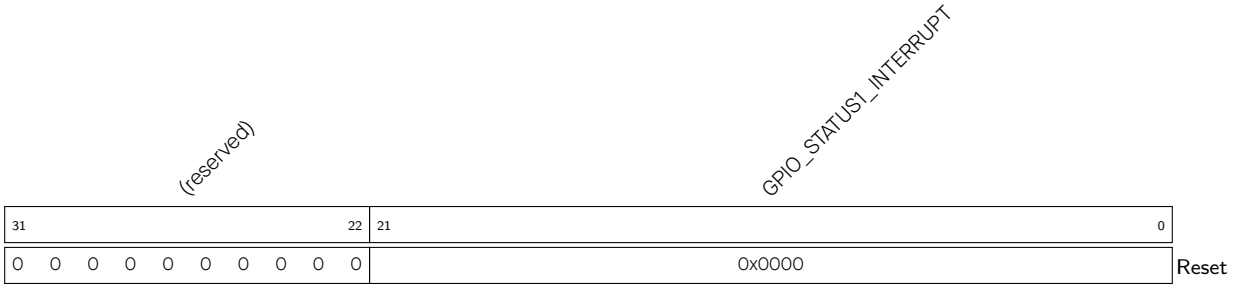
GPIO_CLK_EN 时钟门控使能。此位置 1，则时钟自由运转。(R/W)

Register 6.22. GPIO_STATUS_REG (0x0044)



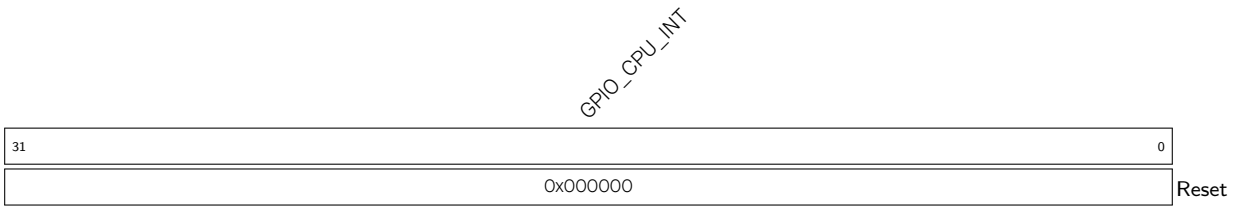
GPIO_STATUS_INTERRUPT GPIO0 ~ 31 中断状态寄存器。(R/W)

Register 6.23. GPIO_STATUS1_REG (0x0050)



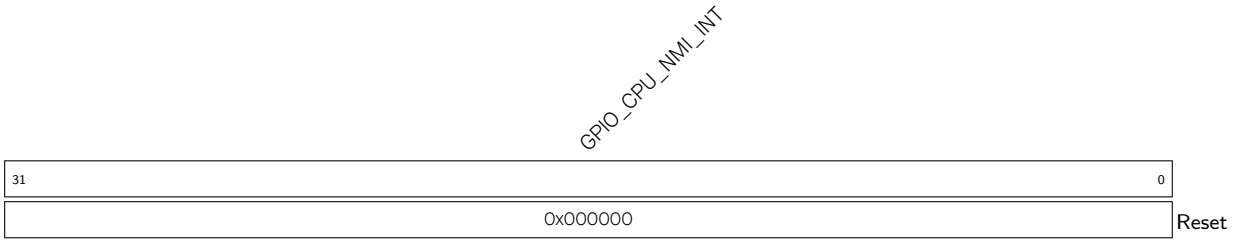
GPIO_STATUS1_INTERRUPT GPIO32 ~ 48 中断状态寄存器。(R/W)

Register 6.24. GPIO_CPU_INT_REG (0x005C)



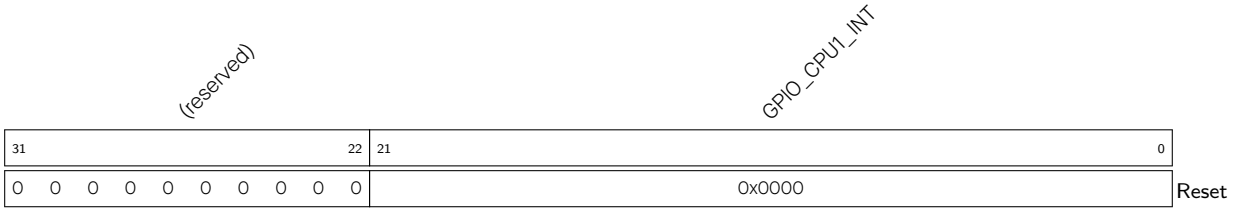
GPIO_CPU_INT GPIO0 ~ 31 CPU 中断状态。如果 GPIO_PIN*n*_REG 中 bit13 高电平有效，即使能 CPU 中断，则此寄存器所示的中断状态应与 GPIO_STATUS_REG 中相应 bit 的中断状态一致。(RO)

Register 6.25. GPIO_CPU_NMI_INT_REG (0x0060)



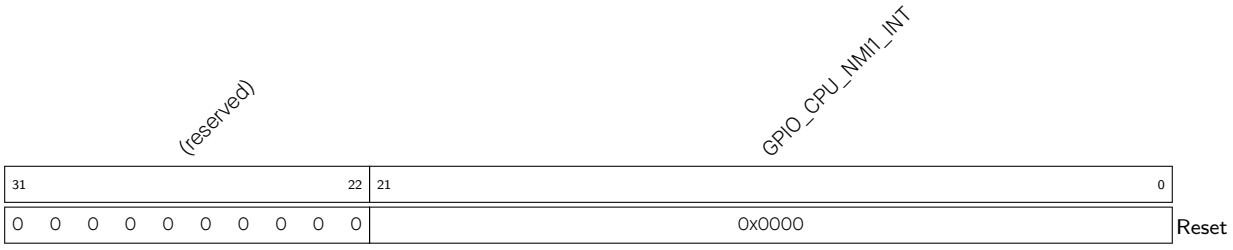
GPIO_CPU_NMI_INT GPIO0 ~ 31 CPU 非屏蔽中断状态寄存器。如果 GPIO_PIN*n*_REG 中 bit14 高电平有效，即使能 CPU 非屏蔽中断，则此寄存器所示的中断状态应与 GPIO_STATUS_REG 中相应 bit 的中断状态一致。(RO)

Register 6.26. GPIO_CPU_INT1_REG (0x0068)



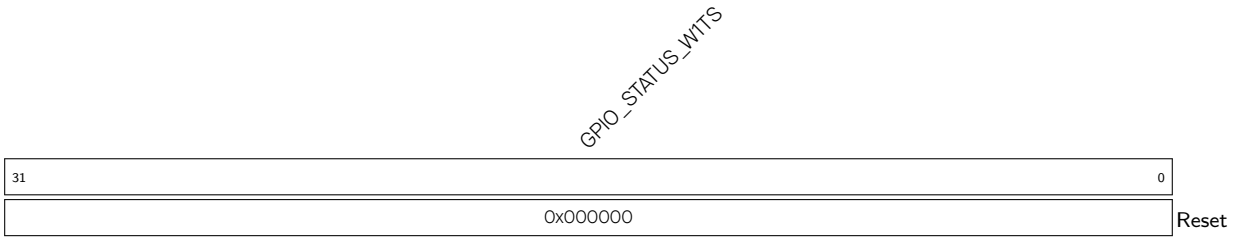
GPIO_CPU1_INT GPIO32 ~ 48 CPU 中断状态寄存器。如果 GPIO_PIN*n*_REG 中 bit13 高电平有效，即使能 CPU 中断，则此寄存器所示的中断状态应与 GPIO_STATUS1_REG 中相应 bit 的中断状态一致。(RO)

Register 6.27. GPIO_CPU_NMI_INT1_REG (0x006C)



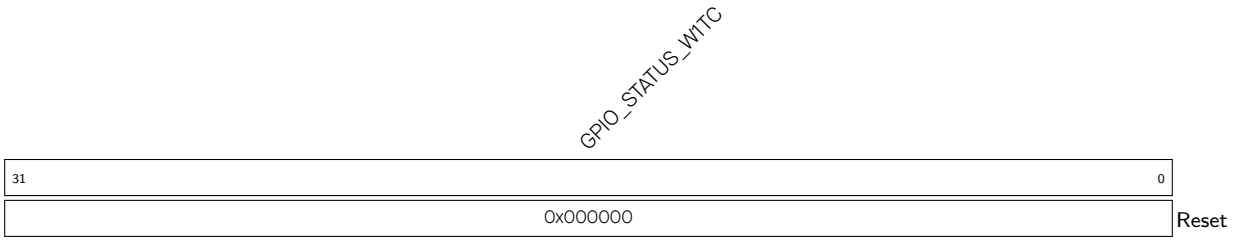
GPIO_CPU_NMI1_INT GPIO32 ~ 48 CPU 非屏蔽中断状态寄存器。如果 GPIO_PIN*n*_REG 中 bit14 高电平有效，即使能 CPU 非屏蔽中断，则此寄存器所示的中断状态应与 GPIO_STATUS1_REG 中相应 bit 的中断状态一致。(RO)

Register 6.28. GPIO_STATUS_W1TS_REG (0x0048)



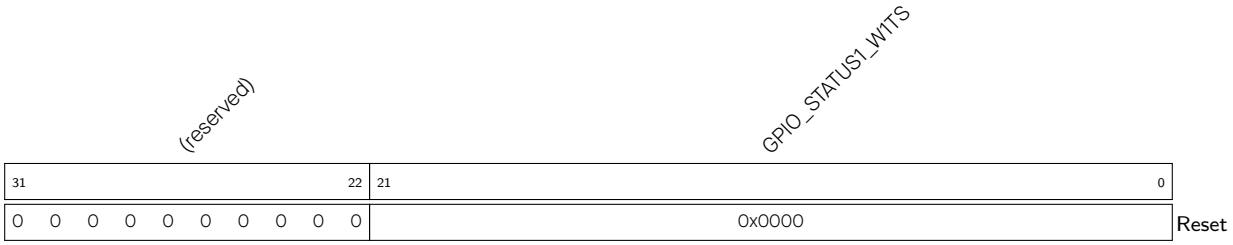
GPIO_STATUS_W1TS GPIO0 ~ 31 中断状态置位寄存器。每位置 1，则 GPIO_STATUS_INTERRUPT 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_STATUS_INTERRUPT。(WO)

Register 6.29. GPIO_STATUS_W1TC_REG (0x004C)



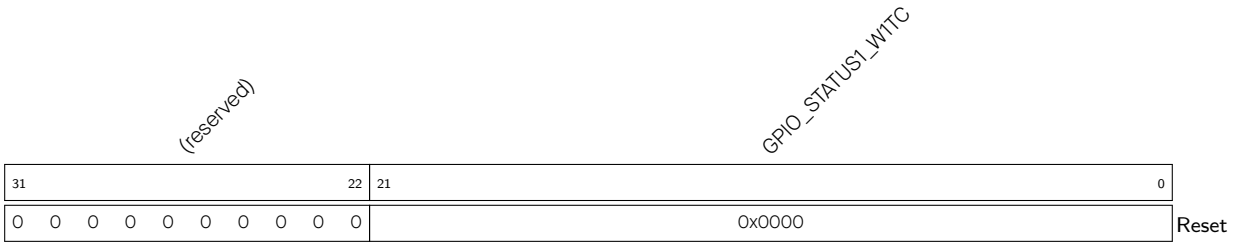
GPIO_STATUS_W1TC GPIO0 ~ 31 中断状态清除寄存器。每一位置 1，则 GPIO_STATUS_INTERRUPT 中的相应位也会清零。注：推荐使用此寄存器来清零 GPIO_STATUS_INTERRUPT。(WO)

Register 6.30. GPIO_STATUS1_W1TS_REG (0x0054)



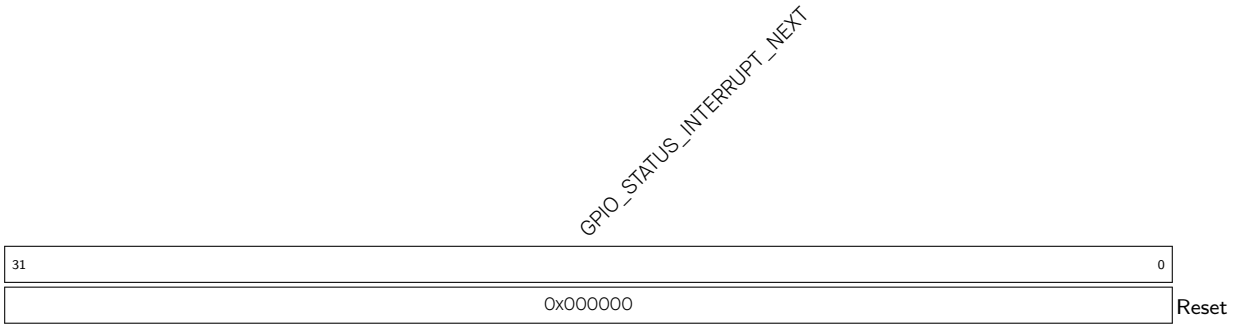
GPIO_STATUS1_W1TS GPIO32 ~ 48 中断状态置位寄存器。每一位置 1，GPIO_STATUS_INTERRUPT1 中相应位也置 1。注：推荐使用此寄存器来置位 GPIO_STATUS_INTERRUPT1。(WO)

Register 6.31. GPIO_STATUS1_W1TC_REG (0x0058)



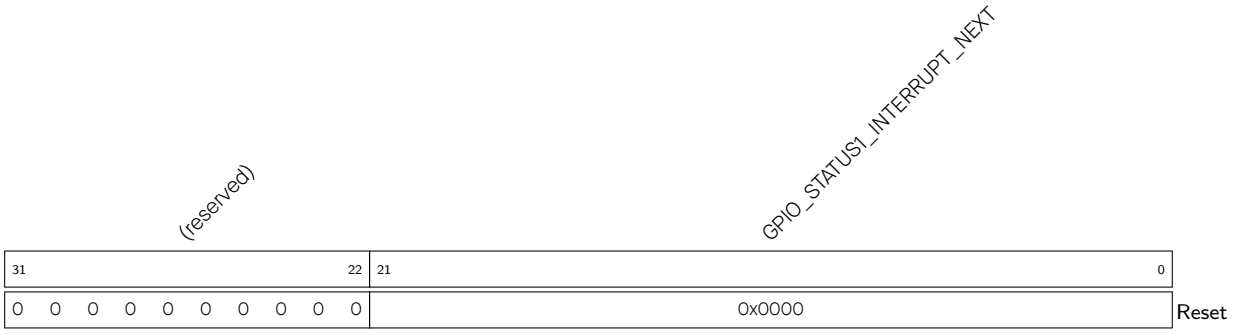
GPIO_STATUS1_W1TC GPIO32 ~ 48 中断状态清除寄存器。每一位置 1，则 GPIO_STATUS_INTERRUPT1 中的相应位也将清零。注：推荐使用此寄存器来清零 GPIO_STATUS_INTERRUPT1。(WO)

Register 6.32. GPIO_STATUS_NEXT_REG (0x014C)



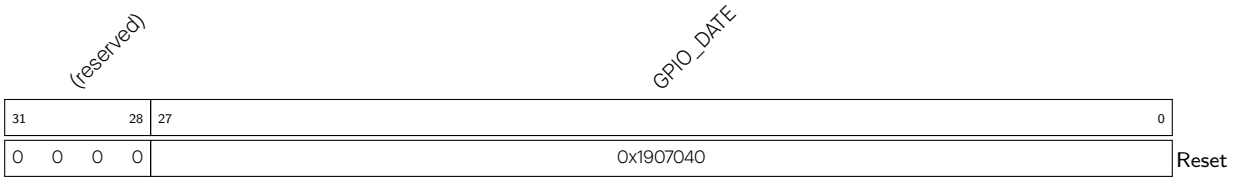
GPIO_STATUS_INTERRUPT_NEXT GPIO0 ~ 31 中断源信号，可以设置为上升沿中断、下降沿中断、电平敏感中断或任一沿中断。(RO)

Register 6.33. GPIO_STATUS_NEXT1_REG (0x0150)



GPIO_STATUS1_INTERRUPT_NEXT GPIO32 ~ 48 的中断源信号。(RO)

Register 6.34. GPIO_REG_DATE_REG (0x06FC)

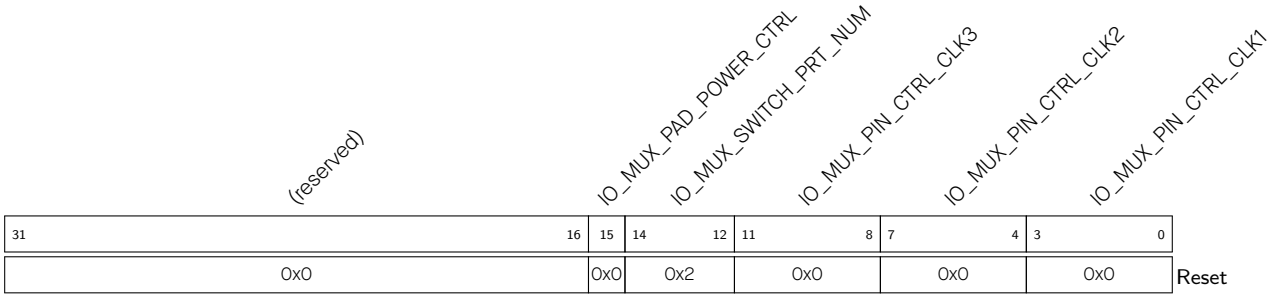


GPIO_DATE 版本控制寄存器。(R/W)

6.15.2 IO MUX 寄存器

本小节的所有地址均为相对于 IO MUX 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 6.35. IO_MUX_PIN_CTRL_REG (0x0000)



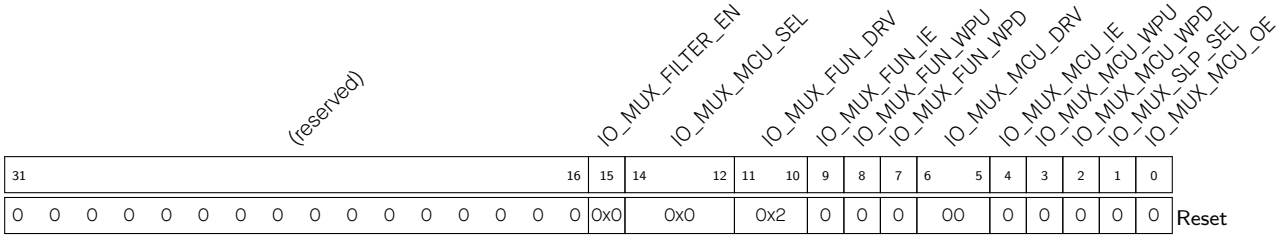
IO_MUX_PIN_CTRL_CLK*x* 配置 I2S0 外设时钟输出到：
CLK_OUT1，配置 IO_MUX_PIN_CTRL_CLK1 = 0x0；
CLK_OUT2，配置 IO_MUX_PIN_CTRL_CLK2 = 0x0；
CLK_OUT3，配置 IO_MUX_PIN_CTRL_CLK3 = 0x0。
(R/W)

配置 I2S1 外设时钟输出到：
CLK_OUT1，配置 IO_MUX_PIN_CTRL_CLK1 = 0xF；
CLK_OUT2，配置 IO_MUX_PIN_CTRL_CLK2 = 0xF；
CLK_OUT3，配置 IO_MUX_PIN_CTRL_CLK3 = 0xF。
(R/W) **说明：**
只能有上述配置组合。
CLK_OUT1 ~ 3 可在 IO MUX 管脚功能列表中查询。

IO_MUX_SWITCH_PRT_NUM GPIO 管脚电源切换延时，延时单位为一个 APB 时钟周期。(R/W)

IO_MUX_PAD_POWER_CTRL 选择 GPIO33 ~ 37 和 GPIO47 ~ 48 的电源电压。1：选择 VDD_SPI 1.8 V 供电；0：选择 VDD3P3_CPU 3.3 V 供电。(R/W)

Register 6.36. IO_MUX_GPIO*n*_REG (*n*: 0-21, 26-48) (0x0004+4**n*)



IO_MUX_MCU_OE 睡眠模式下管脚的输出使能。1：输出使能；0：输出关闭。(R/W)

IO_MUX_SLP_SEL 管脚的睡眠模式选择。置 1 将使能睡眠模式。(R/W)

IO_MUX_MCU_WPD 睡眠模式下管脚的下拉使能。1：内部下拉使能；0：内部下拉关闭。(R/W)

IO_MUX_MCU_WPU 睡眠模式下管脚的上拉使能。1：内部上拉使能；0：内部上拉关闭。(R/W)

IO_MUX_MCU_IE 睡眠模式下管脚的输入使能。1：输入使能；0：输入关闭。(R/W)

IO_MUX_MCU_DRV 配置睡眠模式下 GPIO*n* 的驱动强度。

- GPIO17 和 GPIO18
  0: ~5 mA
  1: **~20 mA**
  2: **~10 mA**
  3: ~40 mA
- 其他 GPIO
  0: ~5 mA
  1: **~10 mA**
  2: **~20 mA**
  3: ~40 mA

(R/W)

IO_MUX_FUN_WPD 管脚的下拉使能。1：内部下拉使能；0：内部下拉关闭。(R/W)

IO_MUX_FUN_WPU 管脚的上拉使能。1：内部上拉使能；0：内部上拉关闭。(R/W)

IO_MUX_FUN_IE 管脚的输入使能。1：输入使能；0：输入关闭。(R/W)

见下页

Register 6.36. IO_MUX_*n*_REG (*n*: GPIO0-GPIO21, GPIO26-GPIO48) (0x0010+4**n*)

接上页

IO_MUX_FUN_DRV 选择管脚驱动强度。

- GPIO17 和 GPIO18
  0: ~5 mA
  1: **~20 mA**
  2: **~10 mA**
  3: ~40 mA
- 其他 GPIO
  0: ~5 mA
  1: **~10 mA**
  2: **~20 mA**
  3: ~40 mA

(R/W)

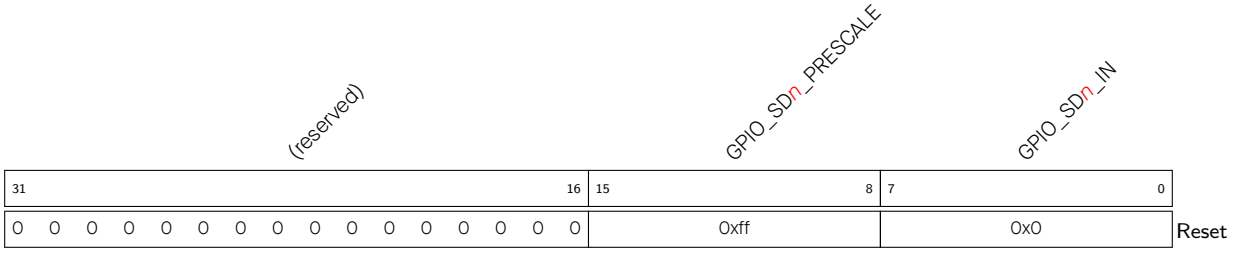
IO_MUX_MCU_SEL 为信号选择 IO MUX 功能。0：选择 Function 0；1：选择 Function 1；以此类推。(R/W)

IO_MUX_FILTER_EN 管脚输入信号滤波使能。1：滤波使能；0：滤波关闭。(R/W)

6.15.3 SDM 寄存器

本小节的所有地址均为相对于 GPIO 基地址 + 0x0F00 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

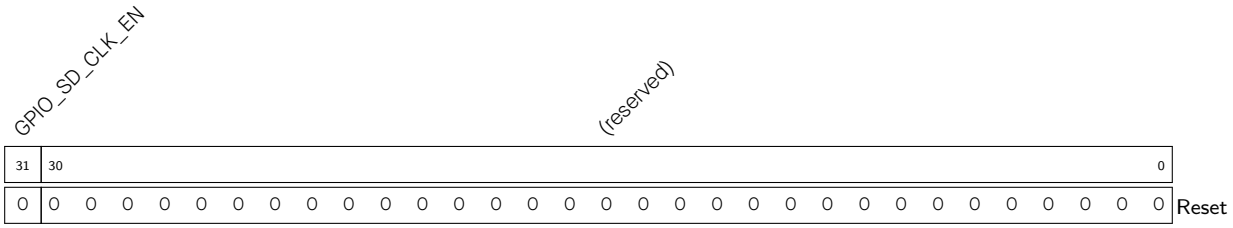
Register 6.37. GPIO_SIGMADELTA*n*_REG (*n*: 0-7) (0x0000+4**n*)



GPIO_SD*n*_IN 配置 SDM 输出信号的占空比。(R/W)

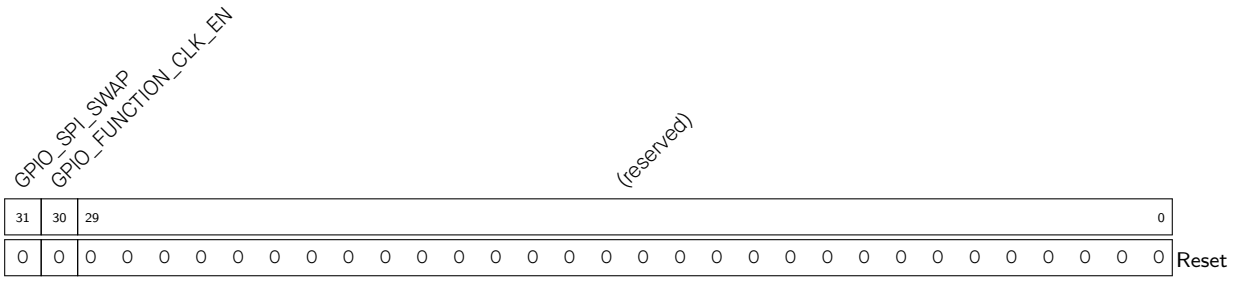
GPIO_SD*n*_PRESCALE 配置 APB_CLK 分频系数。(R/W)

Register 6.38. GPIO_SIGMADELTA_CG_REG (0x0020)



GPIO_SD_CLK_EN 使能 SDM 配置寄存器的时钟。(R/W)

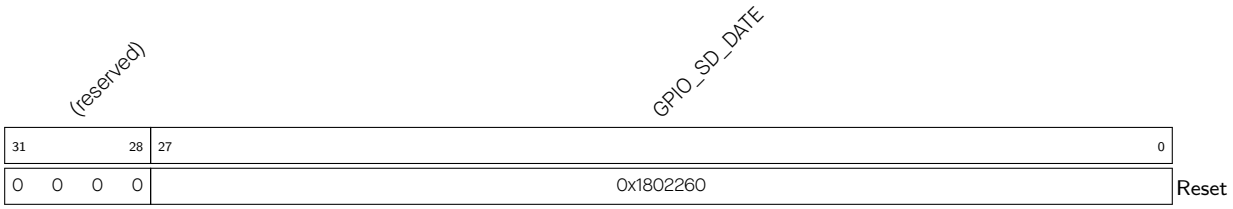
Register 6.39. GPIO_SIGMADELTA_MISC_REG (0x0024)



GPIO_FUNCTION_CLK_EN 使能 SDM 的时钟。(R/W)

GPIO_SPI_SWAP 保留。(R/W)

Register 6.40. GPIO_SIGMADELTA_VERSION_REG (0x0028)

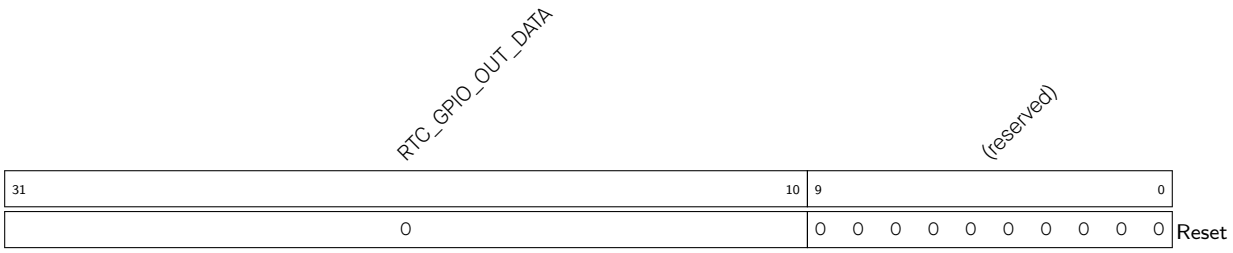


GPIO_SD_DATE 版本控制寄存器。(R/W)

6.15.4 RTC IO MUX 寄存器

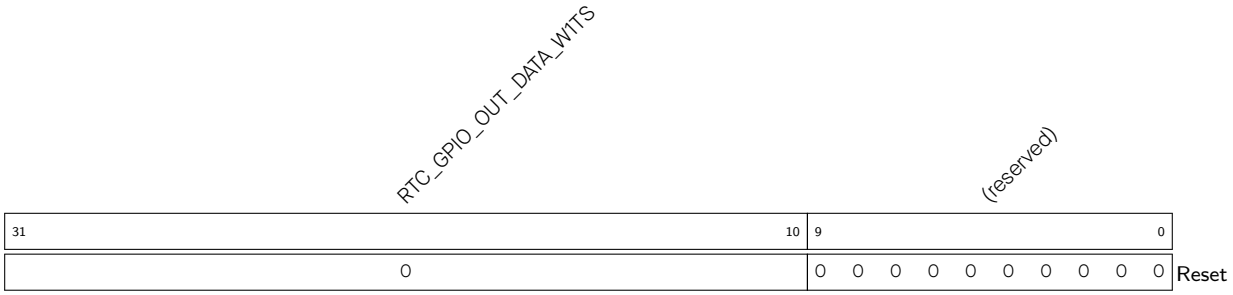
本小节的所有地址均为相对于低功耗管理模块基地址 + 0x0400 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 6.41. RTC_GPIO_OUT_REG (0x0000)



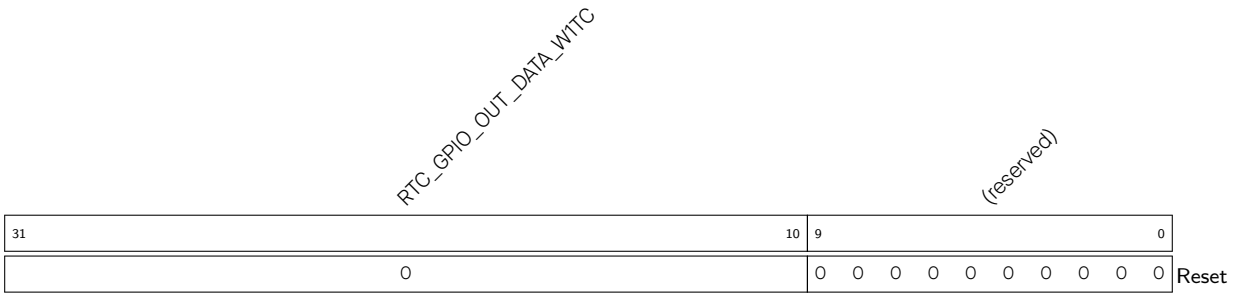
RTC_GPIO_OUT_DATA GPIO0 ~ 21 输出寄存器。bit10 对应 GPIO0，bit11 对应 GPIO1，以此类推。(R/W)

Register 6.42. RTC_GPIO_OUT_W1TS_REG (0x0004)



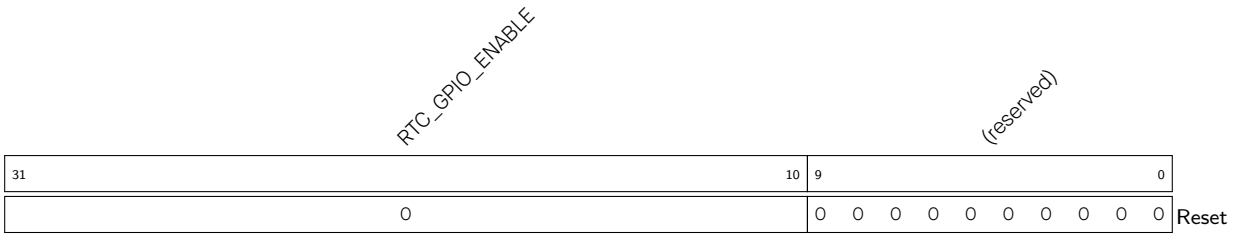
RTC_GPIO_OUT_DATA_W1TS GPIO0 ~ 21 输出置位寄存器。每一位置 1，RTC_GPIO_OUT_REG 中相应位也置 1。注：推荐使用此寄存器来置位 RTC_GPIO_OUT_REG。(WO)

Register 6.43. RTC_GPIO_OUT_W1TC_REG (0x0008)



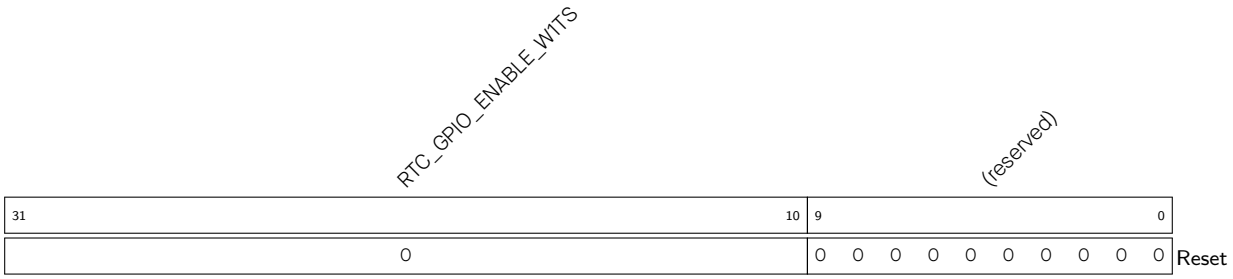
RTC_GPIO_OUT_DATA_W1TC GPIO0 ~ 21 输出清零寄存器。每一位置 1，则 RTC_GPIO_OUT_REG 中相应位将被清零。注：推荐使用此寄存器来清零 RTC_GPIO_OUT_REG。(WO)

Register 6.44. RTC_GPIO_ENABLE_REG (0x000C)



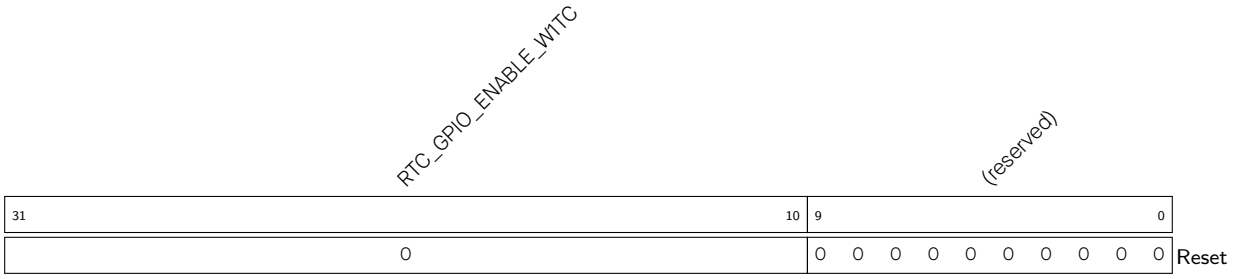
RTC_GPIO_ENABLE GPIO0 ~ 21 输出使能。bit10 对应 GPIO0, bit11 对应 GPIO1，以此类推。此位置 1，即该 GPIO 管脚为输出。(R/W)

Register 6.45. RTC_GPIO_ENABLE_W1TS_REG (0x0010)



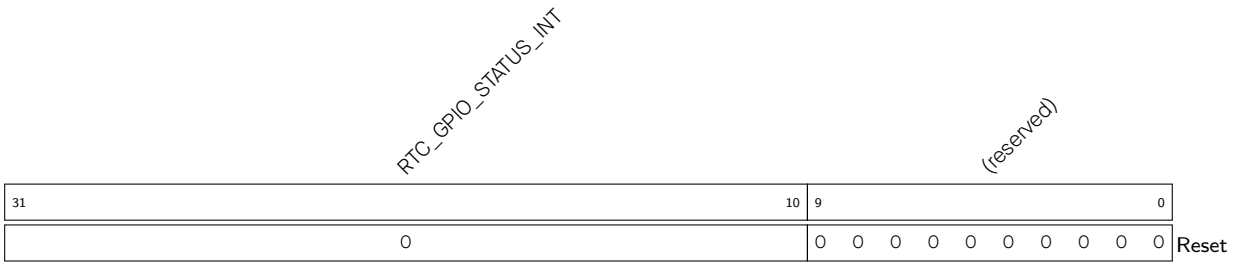
RTC_GPIO_ENABLE_W1TS GPIO0 ~ 21 输出使能置位寄存器。每一位置 1，则 RTC_GPIO_ENABLE_REG 中相应位也将置 1。注：推荐使用此寄存器来置位 RTC_GPIO_ENABLE_REG。(WO)

Register 6.46. RTC_GPIO_ENABLE_W1TC_REG (0x0014)



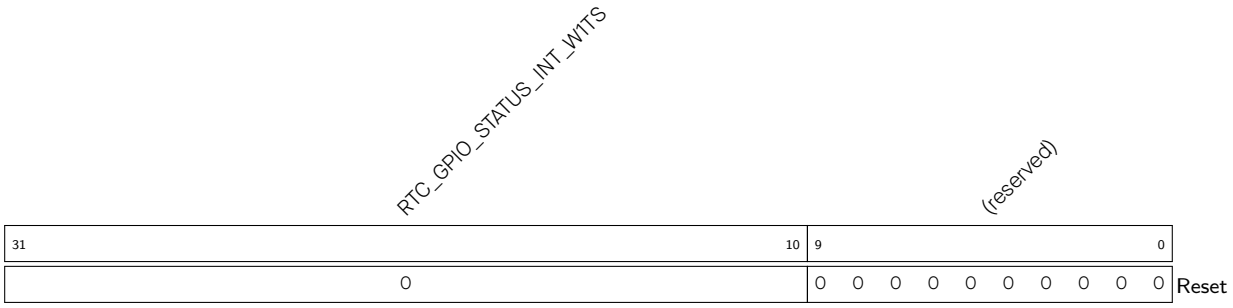
RTC_GPIO_ENABLE_W1TC GPIO0 ~ 21 输出使能清零寄存器。每一位置 1，则 RTC_GPIO_ENABLE_REG 中相应位将被清零。注：推荐使用此寄存器来清零 RTC_GPIO_ENABLE_REG。(WO)

Register 6.47. RTC_GPIO_STATUS_REG (0x0018)



RTC_GPIO_STATUS_INT GPIO0 ~ 21 中断状态寄存器。bit10 对应 GPIO0，bit11 对应 GPIO1，以此类推。此寄存器应同时与 RTC_GPIO_PIN*n*_REG 寄存器中的 RTC_GPIO_PIN*n*_INT_TYPE 中断类型配合使用。0：代表没有中断；1：代表有相应中断。(R/W)

Register 6.48. RTC_GPIO_STATUS_W1TS_REG (0x001C)



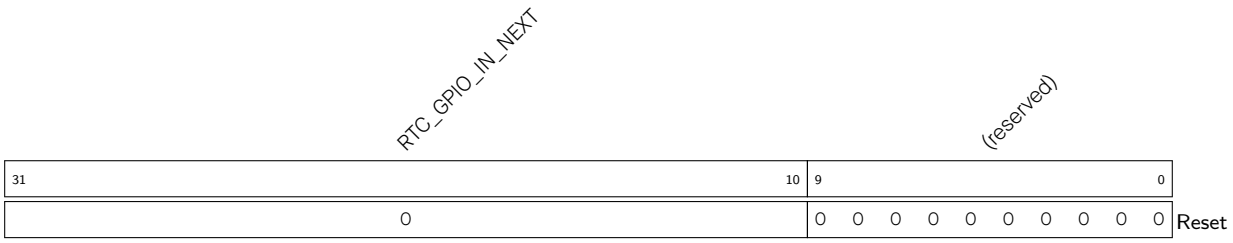
RTC_GPIO_STATUS_INT_W1TS GPIO0 ~ 21 中断状态置位寄存器。每一位置 1，则 RTC_GPIO_STATUS_INT 中相应位也将置 1。注：推荐使用此寄存器来置位 RTC_GPIO_STATUS_INT。(WO)

Register 6.49. RTC_GPIO_STATUS_W1TC_REG (0x0020)



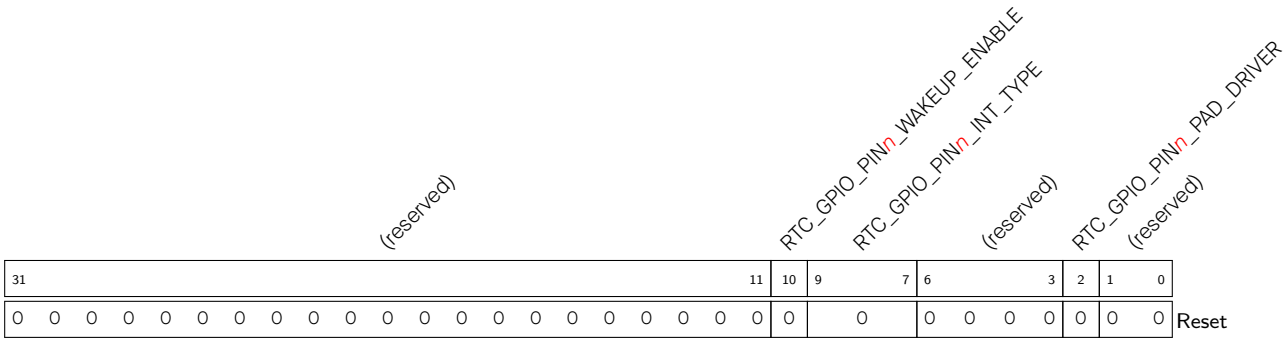
RTC_GPIO_STATUS_INT_W1TC GPIO0 ~ 21 中断状态清零寄存器。每一位置 1，则 RTC_GPIO_STATUS_INT 中的相应位也将清零。注：推荐使用此寄存器来清零 RTC_GPIO_STATUS_INT。(WO)

Register 6.50. RTC_GPIO_IN_REG (0x0024)



RTC_GPIO_IN_NEXT GPIO0 ~ 21 输入值。bit10 对应 GPIO0，bit11 对应 GPIO1，以此类推。每个 bit 代表 pad 的片外输入值，比如片外引脚为高电平，此 bit 值应为 1，片外引脚为低电平，此 bit 值应为 0。(RO)

Register 6.51. RTC_GPIO_PIN*n*_REG (*n*: 0-21) (0x0028+0x4**n*)



RTC_GPIO_PIN*n*_PAD_DRIVER 管脚驱动选择寄存器。0：正常输出；1：开漏输出。(R/W)

RTC_GPIO_PIN*n*_INT_TYPE GPIO 中断类型选择寄存器。(R/W)

0：禁用 GPIO 中断
1：上升沿触发
2：下降沿触发
3：任一沿触发
4：低电平触发
5：高电平触发

RTC_GPIO_PIN*n*_WAKEUP_ENABLE GPIO 唤醒使能。只能将芯片从 Light-sleep 中唤醒。(R/W)

Register 6.52. RTC_IO_TOUCH_PADn_REG (n: 0-14) (0x0084+0x4*n)

Bits	31	30–29	28	27	26–23	22	21	20	19	18–17	16	15	14	13	12–0	
Field	(reserved)	RTC_IO_TOUCH_PADn_DRV	RTC_IO_TOUCH_PADn_RDE	RTC_IO_TOUCH_PADn_RUE	(reserved)	RTC_IO_TOUCH_PADn_START	RTC_IO_TOUCH_PADn_TIE_OPT	RTC_IO_TOUCH_PADn_XPD	RTC_IO_TOUCH_PADn_MUX_SEL	RTC_IO_TOUCH_PADn_FUN_SEL	RTC_IO_TOUCH_PADn_SLP_SEL	RTC_IO_TOUCH_PADn_SLP_IE	RTC_IO_TOUCH_PADn_SLP_OE	RTC_IO_TOUCH_PADn_FUN_IE	(reserved)	
Value	0	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	Reset

RTC_IO_TOUCH_PADn_FUN_IE 工作模式下输入使能。(R/W)

RTC_IO_TOUCH_PADn_SLP_OE 睡眠模式下输出使能。(R/W)

RTC_IO_TOUCH_PADn_SLP_IE 睡眠模式下输入使能。(R/W)

RTC_IO_TOUCH_PADn_SLP_SEL 0: 无睡眠模式; 1: 使能睡眠模式。(R/W)

RTC_IO_TOUCH_PADn_FUN_SEL Function 选择。(R/W)

RTC_IO_TOUCH_PADn_MUX_SEL 选择连接 RTC 管脚输入或数字管脚输入，可以置 0，即选择数字管脚输入。(R/W)

RTC_IO_TOUCH_PADn_XPD 触摸传感器上电。(R/W)

RTC_IO_TOUCH_PADn_TIE_OPT 默认触摸传感器初始电压位。0: 拉高; 1: 拉低。(R/W)

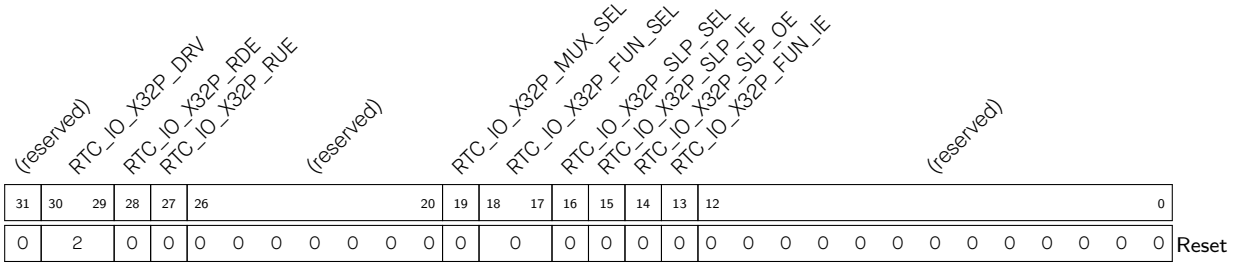
RTC_IO_TOUCH_PADn_START 启动触摸传感器。(R/W)

RTC_IO_TOUCH_PADn_RUE 使能管脚上拉。1: 内部上拉使能; 0: 内部上拉关闭。(R/W)

RTC_IO_TOUCH_PADn_RDE 使能管脚下拉。1: 内部下拉使能; 0: 内部下拉关闭。(R/W)

RTC_IO_TOUCH_PADn_DRV 选择管脚的驱动强度。0: ~5 mA; 1: ~10 mA; 2: ~20 mA; 3: ~40 mA。(R/W)

Register 6.53. RTC_IO_XTAL_32P_PAD_REG (0x00C0)



RTC_IO_X32P_FUN_IE 使能工作模式下管脚输入。(R/W)

RTC_IO_X32P_SLP_OE 使能睡眠模式下管脚输出。(R/W)

RTC_IO_X32P_SLP_IE 使能睡眠模式下管脚输入。(R/W)

RTC_IO_X32P_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(R/W)

RTC_IO_X32P_FUN_SEL Function 选择。(R/W)

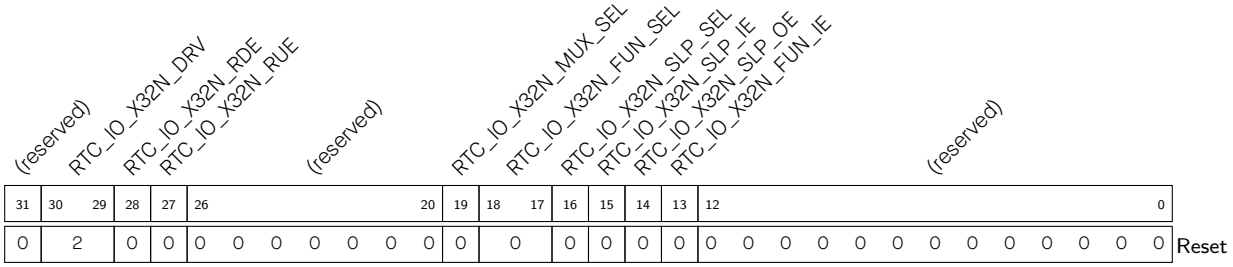
RTC_IO_X32P_MUX_SEL 1：选择使用 RTC GPIO；0：选择使用数字 GPIO。(R/W)

RTC_IO_X32P_RUE 管脚上拉使能。1：内部上拉使能；0：内部上拉关闭。(R/W)

RTC_IO_X32P_RDE 管脚下拉使能。1：内部下拉使能；0：内部下拉关闭。(R/W)

RTC_IO_X32P_DRV 选择管脚的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(R/W)

Register 6.54. RTC_IO_XTAL_32N_PAD_REG (0x00C4)



RTC_IO_X32N_FUN_IE 工作模式下输入使能。(R/W)

RTC_IO_X32N_SLP_OE 睡眠模式下输出使能。(R/W)

RTC_IO_X32N_SLP_IE 睡眠模式下输入使能。(R/W)

RTC_IO_X32N_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(R/W)

RTC_IO_X32N_FUN_SEL Function 选择。(R/W)

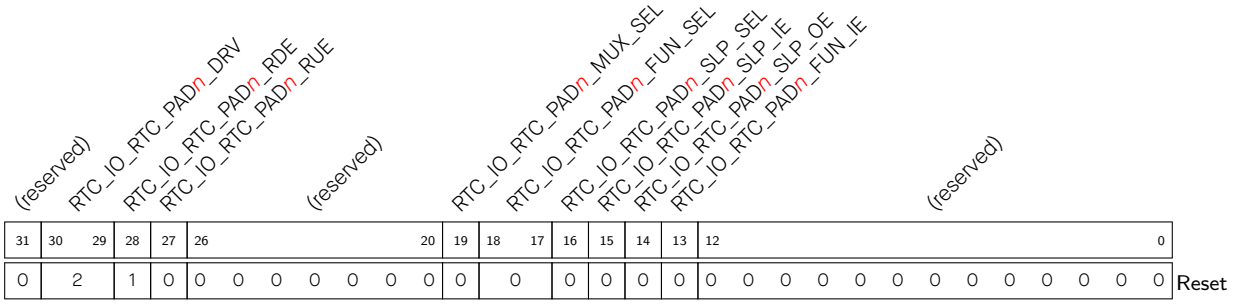
RTC_IO_X32N_MUX_SEL 1：选择使用 RTC GPIO；0：选择使用数字 GPIO。(R/W)

RTC_IO_X32N_RUE Pad 上拉使能。1：内部上拉使能；0：内部上拉关闭。(R/W)

RTC_IO_X32N_RDE Pad 下拉使能。1：内部下拉使能；0：内部下拉关闭。(R/W)

RTC_IO_X32N_DRV 选择 pad 的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(R/W)

Register 6.55. RTC_IO_RTC_PAD*n*_REG (*n*: 17-21) (0x00C8, 0x00CC, 0x00D0, 0x00D4, 0x00D8)



RTC_IO_RTC_PAD*n*_FUN_IE 工作模式下输入使能。(R/W)

RTC_IO_RTC_PAD*n*_SLP_OE 睡眠模式下输出使能。(R/W)

RTC_IO_RTC_PAD*n*_SLP_IE 睡眠模式下输入使能。(R/W)

RTC_IO_RTC_PAD*n*_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(R/W)

RTC_IO_RTC_PAD*n*_FUN_SEL Function 选择。(R/W)

RTC_IO_RTC_PAD*n*_MUX_SEL 1：选择使用 RTC GPIO；2：选择使用数字 GPIO。(R/W)

RTC_IO_RTC_PAD*n*_RUE 管脚上拉使能。1：内部上拉使能；0：内部上拉关闭。(R/W)

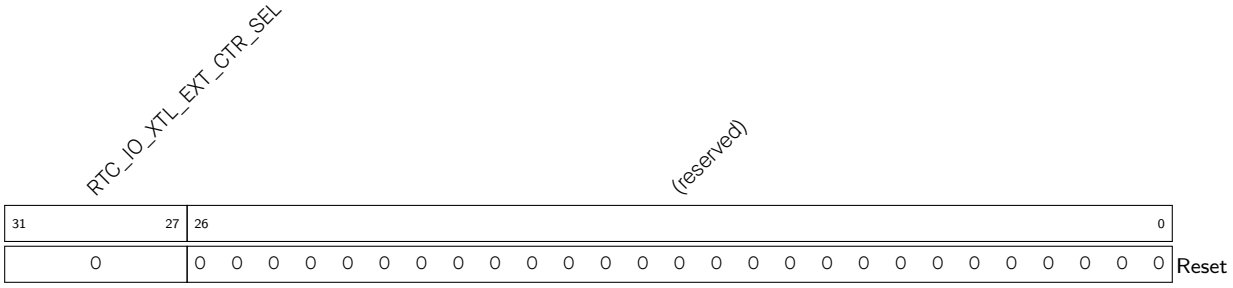
RTC_IO_RTC_PAD*n*_RDE 管脚下拉使能。1：内部下拉使能；0：内部下拉关闭。(R/W)

RTC_IO_RTC_PAD*n*_DRV 选择管脚的驱动强度。

- RTC GPIO17 和 RTC GPIO18
  0: ~5 mA
  1: **~20 mA**
  2: **~10 mA**
  3: ~40 mA
- 其他 RTC GPIO
  0: ~5 mA
  1: **~10 mA**
  2: **~20 mA**
  3: ~40 mA

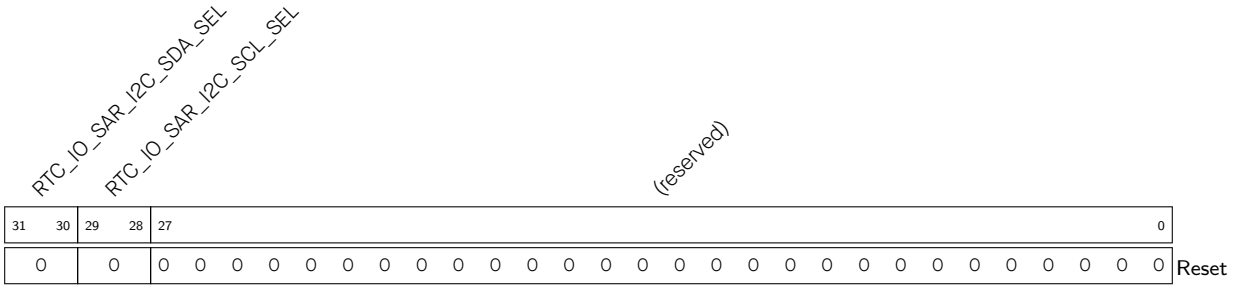
(R/W)

Register 6.56. RTC_IO_XTL_EXT_CTR_REG (0x00E0)



RTC_IO_XTL_EXT_CTR_SEL 选择睡眠模式下外部晶振断电使能源。0：选择 GPIO0；1：选择 GPIO1，以此类推。被选择管脚的值异或 RTC_CNTL_EXT_XTL_CONF_REG[30] 上的逻辑值时晶振断电使能信号。(R/W)

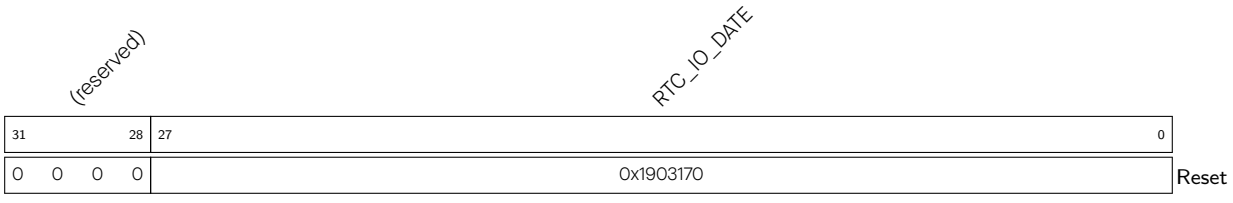
Register 6.57. RTC_IO_SAR_I2C_IO_REG (0x00E4)



RTC_IO_SAR_I2C_SCL_SEL 选择 RTC I2C SCL 信号连接的管脚。0：选择 RTC GPIO0；1：选择 RTC GPIO2。(R/W)

RTC_IO_SAR_I2C_SDA_SEL 选择 RTC I2C SDA 信号连接的管脚。0：选择 RTC GPIO1；1：选择 RTC GPIO3。(R/W)

Register 6.58. RTC_IO_DATE_REG (0x01FC)



RTC_IO_DATE 版本控制寄存器。(R/W)

第 7 章

复位和时钟

7.1 复位

7.1.1 概述

ESP32-S3 提供四种级别的复位方式，分别是 CPU 复位、内核复位、系统复位和芯片复位。除芯片复位外其它复位方式不影响片上内存存储的数据。图 7.1-1 展示了整个芯片系统的结构以及四种复位等级。

7.1.2 结构图

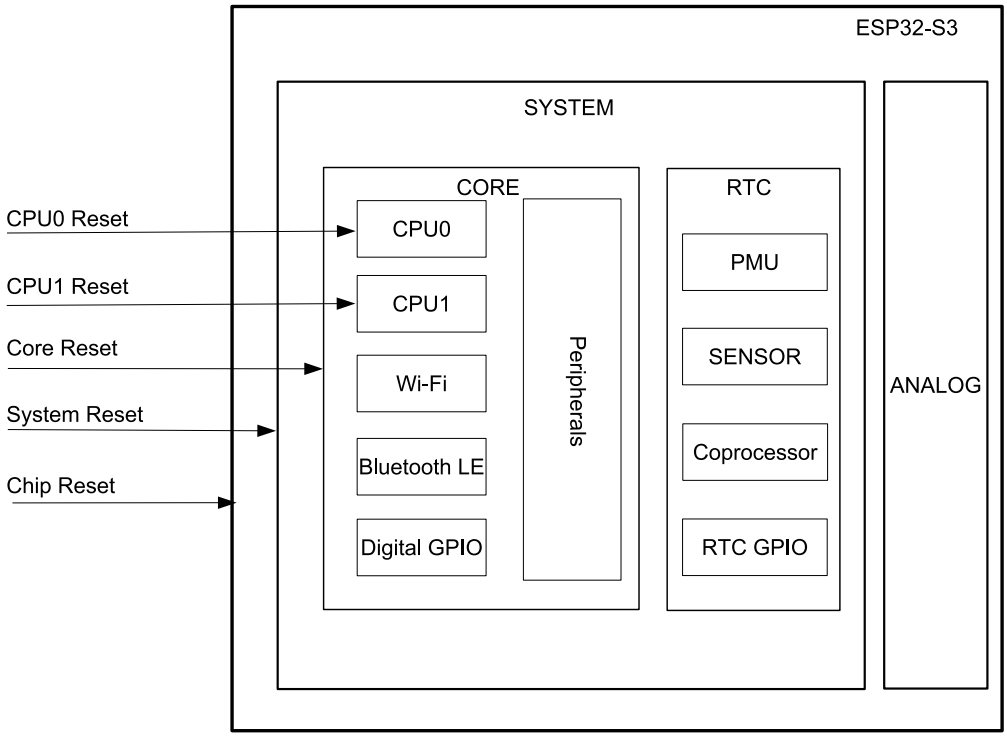


图 7.1-1. 四种复位等级

7.1.3 特性

- 支持四种复位等级：
  - CPU 复位：只复位 CPU*x* 核。这里的 CPU*x* 代表 CPU0 或 CPU1。复位释放后，程序将从 CPU*x* Reset Vector 开始执行。每个 CPU 核拥有独立的复位逻辑。
  - 内核复位：复位除 RTC 以外的其它数字系统，包括 CPU0、CPU1、外设、Wi-Fi、Bluetooth® LE 及数字 GPIO；

  - 系统复位：复位包括 RTC 在内的整个数字系统；
  - 芯片复位：复位整个芯片。
- 支持软件复位和硬件复位：
  - 软件复位：CPU*x* 配置相关寄存器可触发软件复位，见章节 10 低功耗管理 *(RTC_CNTL)*；
  - 硬件复位：硬件复位直接由硬件电路触发。

说明：
如果 CPU 复位来自 CPU0，则 PMS 寄存器也将复位。

7.1.4 功能描述

上述任一复位源产生时，CPU0 和 CPU1 均将立刻复位。复位释放后，CPU0 和 CPU1 可分别通过读取寄存器 RTC_CNTL_RESET_CAUSE_PROCPU 和 RTC_CNTL_RESET_CAUSE_APPCPU 获取复位源。这两个寄存器记录的复位源除了复位级别为 CPU 复位的复位源分别对应自身的 CPU*x* 以外，其余的复位源保持一致。

表 7.1-1 列出了从上述两个寄存器中可能读出的复位源。

表 7.1-1. 复位源

编码	复位源	复位等级	说明
0x01	芯片复位[1]	芯片复位	—
0x0F	欠压系统复位	系统复位或芯片复位	欠压检测器触发的系统复位[2]
0x10	RWDT 系统复位	系统复位	详见章节 13 看门狗定时器 (WDT)
0x12	Super Watchdog 复位	系统复位	详见章节 13 看门狗定时器 (WDT)
0x13	GLITCH 复位	系统复位	详见章节 24 时钟毛刺检测
0x03	软件系统复位	内核复位	配置 RTC_CNTL_SW_SYS_RST 寄存器触发
0x05	Deep-sleep 复位	内核复位	详见章节 10 低功耗管理 (RTC_CNTL)
0x07	MWDT0 内核复位	内核复位	详见章节 13 看门狗定时器 (WDT)
0x08	MWDT1 内核复位	内核复位	详见章节 13 看门狗定时器 (WDT)
0x09	RWDT 内核复位	内核复位	详见章节 13 看门狗定时器 (WDT)
0x14	eFuse 复位	内核复位	eFuse CRC 校验错误触发复位
0x15	USB (UART) 复位	内核复位	外部 USB 主机发送特定命令给 USB-Serial-JTAG 的 Serial 接口将触发此复位，详见章节 33 USB 串口/JTAG 控制器 (USB_SERIAL_JTAG)
0x16	USB (JTAG) 复位	内核复位	外部 USB 主机发送特定命令给 USB-Serial-JTAG 的 JTAG 接口将触发此复位，详见章节 33 USB 串口/JTAG 控制器 (USB_SERIAL_JTAG)
0x0B	MWDT0 CPU*x* 复位	CPU 复位	详见章节 13 看门狗定时器 (WDT)
0x0C	软件 CPU*x* 复位	CPU 复位	配置 RTC_CNTL_SW_PROCPU_RST 寄存器触发
0x0D	RWDT CPU*x* 复位	CPU 复位	详见章节 13 看门狗定时器 (WDT)
0x11	MWDT1 CPU*x* 复位	CPU 复位	详见章节 13 看门狗定时器 (WDT)

[1] 芯片复位的触发源包括以下三项：

- 芯片上电触发芯片复位
- 欠压检测器触发芯片复位
- 超级看门狗 (SWD) 触发芯片复位

[2] 欠压检测器在检测到欠压状态时，将根据寄存器配置，选择触发系统复位或者芯片复位。详见章节 10 低功耗管理 (RTC_CNTL)。

7.2 时钟

7.2.1 概述

ESP32-S3 的时钟主要来源于振荡器 (oscillator, OSC)、RC 振荡电路和 PLL 时钟生成电路。上述时钟源产生的时钟经时钟分频器或时钟选择器等时钟模块的处理，使得大部分功能模块可以根据不同功耗和性能需求来获取及选择对应频率的工作时钟。图 7.2-1 为系统时钟结构。

7.2.2 结构图

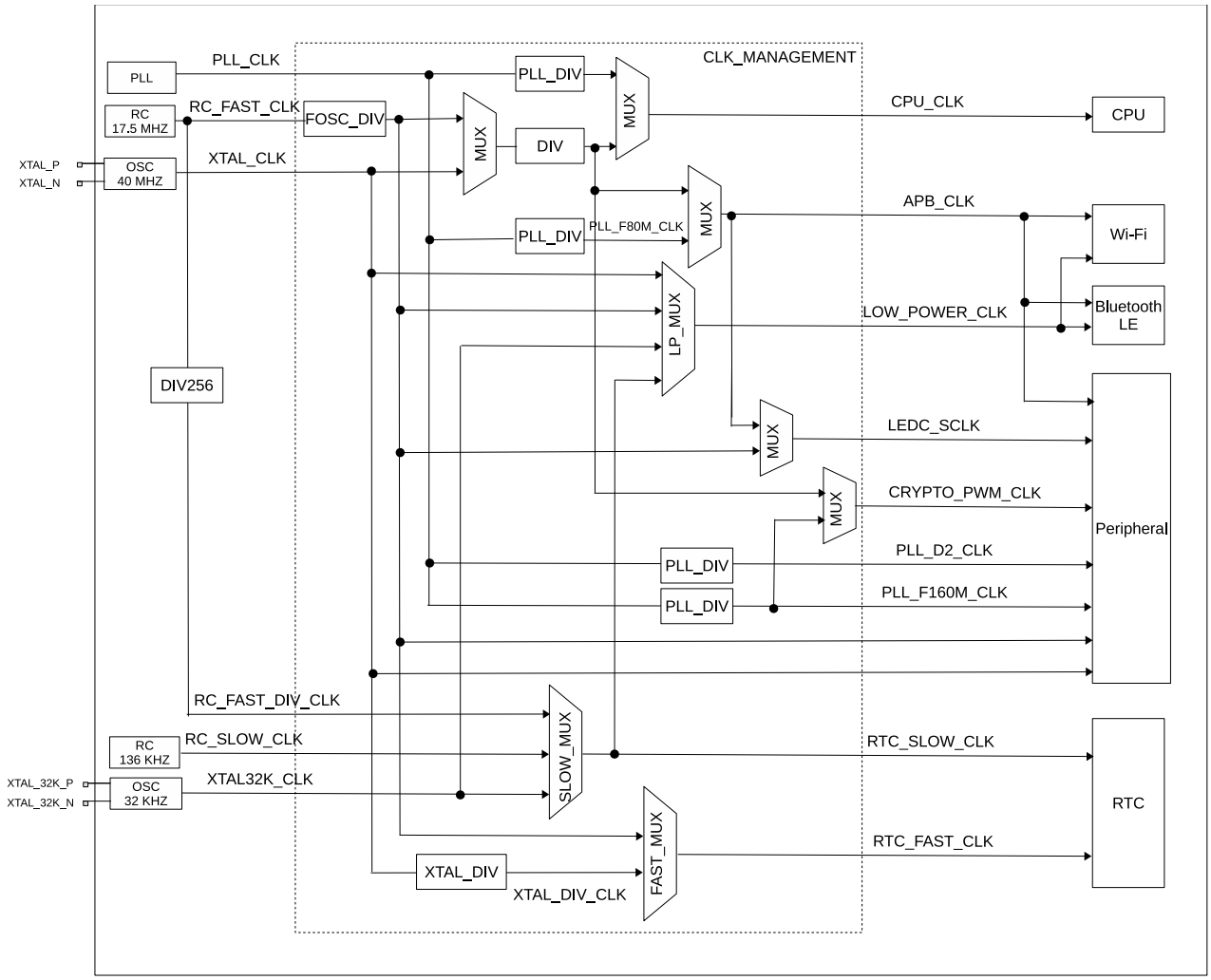


图 7.2-1. 系统时钟

7.2.3 特性

ESP32-S3 的时钟根据频率不同，可分为：

- 高性能时钟，主要为 CPU 和数字外设提供工作时钟
  - PLL_CLK：320 MHz 或 480 MHz 内部 PLL 时钟
  - XTAL_CLK：40 MHz 外部晶振时钟
- 低功耗时钟，主要为 RTC 模块以及部分处于低功耗模式的外设提供工作时钟
  - XTAL32K_CLK：32 kHz 外部晶振时钟
  - RC_FAST_CLK：内置快速 RC 振荡器时钟，频率可调节（通常为 17.5 MHz）
  - RC_FAST_DIV_CLK：内置快速 RC 振荡器分频时钟，由内置快速 RC 振荡器时钟经 256 分频生成
  - RC_SLOW_CLK：内置慢速 RC 振荡器，频率可调节（通常为 136 kHz）

7.2.4 功能描述

7.2.4.1 CPU 时钟

如图 7.2-1 所示，CPU_CLK 为 CPU 主时钟。CPU 在最高效工作模式下，主频可以达到 240 MHz。同时，CPU 能够在超低频下工作（通常为 2 MHz），以减少功耗。

CPU_CLK 由 SYSTEM_SOC_CLK_SEL 来选择时钟源，允许选择 PLL_CLK、RC_FAST_CLK 或 XTAL_CLK 作为 CPU_CLK 的时钟源。具体请参考表 7.2-1 和表 7.2-2。默认状态下，CPU 的时钟为 XTAL_CLK，且分频系数为 2 分频，即 20 MHz。

表 7.2-1. CPU_CLK 时钟源选择

SYSTEM_SOC_CLK_SEL 值	时钟源
0	XTAL_CLK
1	PLL_CLK
2	RC_FAST_CLK

表 7.2-2. CPU_CLK 时钟频率

时钟源	SEL_0[a]	SEL_2[b]	SEL_3[c]	CPU 时钟频率
XTAL_CLK	0	-	-	CPU_CLK = XTAL_CLK/(SYSTEM_PRE_DIV_CNT + 1) SYSTEM_PRE_DIV_CNT 默认值为 1，范围 0 ~ 1023。
PLL_CLK (480 MHz)	1	1	0	CPU_CLK = PLL_CLK/6 CPU_CLK 频率为 80 MHz。
PLL_CLK (480 MHz)	1	1	1	CPU_CLK = PLL_CLK/3 CPU_CLK 频率为 160 MHz。
PLL_CLK (480 MHz)	1	1	2	CPU_CLK = PLL_CLK/2 CPU_CLK 频率为 240 MHz。
PLL_CLK (320 MHz)	1	0	0	CPU_CLK = PLL_CLK/4 CPU_CLK 频率为 80 MHz。
PLL_CLK (320 MHz)	1	0	1	CPU_CLK = PLL_CLK/2 CPU_CLK 频率为 160 MHz。
RC_FAST_CLK	2	-	-	CPU_CLK = RC_FAST_CLK/(SYSTEM_PRE_DIV_CNT + 1) SYSTEM_PRE_DIV_CNT 默认值为 1，范围 0 ~ 1023。

[a] 寄存器 SYSTEM_SOC_CLK_SEL 的值
[b] 寄存器 SYSTEM_PLL_FREQ_SEL 的值
[c] 寄存器 SYSTEM_CPUPERIOD_SEL 的值

7.2.4.2 外设时钟

外设所需要的时钟包括 APB_CLK、CRYPTO_PWM_CLK、PLL_F160M_CLK、PLL_D2_CLK、LEDC_CLK、XTAL_CLK 和 RC_FAST_CLK。表 7.2-3 列出了接入各个外设的时钟。

表 7.2-3. 外设时钟

外设	XTAL_CLK	APB_CLK	PLL_F160M_CLK	PLL_D2_CLK	RC_FAST_CLK	CRYPTO_PWM_CLK	LEDC_CLK
TIMG	Y	Y					
I2S	Y		Y	Y			
UHCI		Y					
UART	Y	Y			Y		
RMT	Y	Y			Y		
PWM						Y	
I2C	Y				Y		
SPI	Y	Y					
PCNT		Y					
eFuse Controller	Y	Y					
SARADC		Y		Y			
USB		Y					
CRYPTO						Y	
TWAI Controller		Y					
SDIO HOST	Y		Y				
LEDC	Y	Y			Y		Y
LCD_CAM	Y		Y	Y			
SYS_TIMER	Y	Y					

APB_CLK 时钟

如表 7.2-4 所示，APB_CLK 的频率由 CPU_CLK 的时钟源决定。

表 7.2-4. APB_CLK 时钟

CPU_CLK 时钟源	APB_CLK 频率
PLL_CLK	80 MHz
XTAL_CLK	CPU_CLK
RC_FAST_CLK	CPU_CLK

CRYPTO_PWM_CLK 时钟

如表 7.2-5 所示，CRYPTO_PWM_CLK 的频率由 CPU_CLK 的时钟源决定。

表 7.2-5. CRYPTO_PWM_CLK 时钟

CPU_CLK 时钟源	CRYPTO_PWM_CLK 频率
PLL_CLK	160 MHz
XTAL_CLK	CPU_CLK
RC_FAST_CLK	CPU_CLK

PLL_F160M_CLK 时钟

PLL_F160M_CLK 是 PLL_CLK 根据当前 PLL 的频率分频所得，因此 PLL_F160M_CLK 的频率始终为 160 Mhz。

PLL_D2_CLK 时钟

PLL_D2_CLK 是 PLL_CLK 根据当前 PLL 的频率分频所得。

LEDC_CLK 时钟

LEDC 模块能将 RC_FAST_CLK 作为时钟源使用，即在 APB_CLK 关闭的时候，LEDC 也可工作。换而言之，当系统处于低功耗模式时，其他外设都将停止工作（APB_CLK 关闭），但是 LEDC 仍然可以通过 RC_FAST_CLK 来正常工作。

7.2.4.3 Wi-Fi 和 Bluetooth LE 时钟

Wi-Fi 和 Bluetooth LE 必须在 CPU_CLK 时钟源选择 PLL_CLK 下才能工作。只有当 Wi-Fi 和 Bluetooth LE 进入低功耗模式时，才能暂时关闭 PLL_CLK。

LOW_POWER_CLK 允许选择 XTAL32K_CLK、XTAL_CLK、RC_FAST_CLK、RTC_SLOW_CLK（RTC 当前所选的慢速时钟）用于 Wi-Fi 和 Bluetooth LE 的低功耗模式。

7.2.4.4 RTC 时钟

RTC_SLOW_CLK 和 RTC_FAST_CLK 的时钟源为低频时钟。RTC 模块能够在大多数时钟源关闭的状态下工作。RTC_SLOW_CLK 允许选择 RC_SLOW_CLK、XTAL32K_CLK 或 RC_FAST_DIV_CLK，用于驱动功耗管理模块。RTC_FAST_CLK 允许选择 XTAL_CLK 或 RC_FAST_CLK 的分频时钟，用于驱动片上传感器模块。

第 8 章

芯片 Boot 控制

8.1 概述

ESP32-S3 共有四个 Strapping 管脚：

- GPIO0
- GPIO3
- GPIO45
- GPIO46

Strapping 管脚用于控制 ESP32-S3 芯片上电或硬件复位时的一些功能：

- 控制 Boot 模式
- 控制 ROM 日志打印
- 设置 VDD_SPI 电压
- 控制 JTAG 信号源

在芯片复位过程中（请参考章节 7 复位和时钟），硬件将采样 Strapping 管脚电平存储到锁存器中，并一直保持到芯片掉电或关闭。GPIO0、GPIO3、GPIO45 和 GPIO46 锁存的状态可以通过软件从寄存器 GPIO_STRAPPING 中读取。

GPIO0、GPIO45 和 GPIO46 默认连接内部上拉/下拉。如果这些管脚没有外部连接或者连接的外部线路处于高阻抗状态，内部弱上拉/下拉将决定这几个管脚输入电平的默认值，如表 8.1-1 所示。

表 8.1-1. Strapping 管脚默认上拉/下拉

管脚	默认值
GPIO0	上拉
GPIO3	N/A
GPIO45	下拉
GPIO46	下拉

如需改变 Strapping 管脚的默认值，用户可以应用外部下拉/上拉电阻，或者应用主机 MCU 的 GPIO 来控制 ESP32-S3 上电复位时的 Strapping 管脚电平。复位释放后，Strapping 管脚和普通管脚功能相同。

说明：
以下小节介绍了芯片复位时的功能以及控制该功能使用到的 Strapping 组合模式。请使用本章节所介绍的组合，其它组合可能会导致不可控结果。

8.2 Boot 模式控制

复位释放后，GPIO0、GPIO1、GPIO2 和 GPIO46 在复位时的值将共同决定 Boot 模式。表 8.2-1 列出了 GPIO0、GPIO1、GPIO2 和 GPIO46 的 Strapping 值及其对应的系统启动模式。

表 8.2-1. 系统启动模式

启动模式	GPIO0	GPIO46	GPIO1	GPIO2
SPI Boot 模式	1	忽略	忽略	忽略
Joint Download Boot 模式[1]	0	0	忽略	忽略
SPI Download Boot 模式[2]	0	1	1	0

[1] Joint Download Boot：该模式下支持以下下载方式：

- USB Download Boot
  - USB-Serial-JTAG Download Boot
  - USB-OTG Download Boot
- UART Download Boot

[2] SPI Download Boot 模式：GPIO1 和 GPIO2 不是 strapping 管脚，但使用 SPI Download Boot 模式时需要预留 GPIO1 和 GPIO2。GPIO1 和 GPIO2 默认浮空，在复位时处于高阻抗状态。

在 SPI Boot 模式下，ROM 引导加载程序通过从 SPI flash 中读取程序来启动系统。SPI Boot 模式可进一步细分为以下两种启动方式：

- 常规 flash 启动方式：支持安全启动。ROM 引导加载程序将程序从 flash 加载到 SRAM，并执行。在大多数实际应用场景中，上述执行的程序多为二级引导程序，该二级引导程序将启动最终的应用程序。
- 直接启动方式：不支持安全启动，程序直接从 flash 中运行。如需使能这一启动方式，请确保下载至 flash 的 bin 文件其前两个字（地址：0x42000000）为 0xaedb041d。

在 Joint Download Boot 模式下，用户可通过 UART 或 USB 接口将二进制文件下载至 flash，或将二进制文件下载至 SRAM 并在 SRAM 中运行程序。

在 SPI Download Boot 模式下，用户可通过 SPI 接口将二进制文件下载至 flash，或将二进制文件下载至 SRAM 并运行 SRAM 中的程序。

下面几个 eFuse 可用于控制启动模式的具体行为：

- EFUSE_DIS_FORCE_DOWNLOAD
  - 如果此 eFuse 设置为 0（默认），软件可通过设置 RTC_CNTL_FORCE_DOWNLOAD_BOOT，触发 CPU 复位，将芯片启动模式强制从 SPI Boot 模式切换至 Joint Download Boot 模式。这种情况下，硬件会将 GPIO_STRAPPING[3:2] 的值从 “1x” 覆盖为 “00”；
  - 如果此 eFuse 设置为 1，则禁用 RTC_CNTL_FORCE_DOWNLOAD_BOOT。GPIO_STRAPPING 的值不受影响。
- EFUSE_DIS_DOWNLOAD_MODE

  如果此 eFuse 设置为 1，则禁用 Joint Download Boot 模式。GPIO_STRAPPING 的值则不受 RTC_CNTL_FORCE _DOWNLOAD_BOOT 的影响。

- EFUSE_ENABLE_SECURITY_DOWNLOAD

  如果此 eFuse 设置为 1，则启用安全下载模式，允许读取、写入和擦除 **明文** flash，不允许通过 UART、USB 或 SPI 接口下载代码到 flash 直接执行，并且不支持寄存器的操作。注意：在该模式下，支持的 esptool 命令受限，例如只能写入 flash 而无法读取。如需读取 flash，请切换至 SPI Boot 模式，启用引导加载程序。

  如果 Joint Download Boot 模式已禁用，则此 eFuse 设置将被忽略。

- EFUSE_DIS_DIRECT_BOOT

  如果此 eFuse 设置为 1，则禁用 Direct Boot 模式。

USB Serial/JTAG 控制器可将芯片从 SPI Boot 模式强制切换到 Joint Download Boot 模式，或从 Joint Download Boot 模式强制切换到 SPI Boot 模式。更多信息，请参考章节 *33 USB 串口/JTAG 控制器 (USB_SERIAL_JTAG)*。

8.3 ROM 日志打印控制

在系统启动过程中，默认配置下，系统将同时把 ROM 日志打印至 UART0 和 USB Serial/JTAG 控制器。在不同启动模式以及配置下，可以分别关闭 UART0 和 USB Serial/JTAG 控制器的 ROM 日志打印功能。

UART0 的 ROM 日志打印控制如下表：

表 8.3-1. UART0 的 ROM 日志打印控制

启动模式	Register[1]	eFuse[2]	GPIO46	ROM 代码打印至 UART0
Download Boot 模式	忽略	忽略	忽略	使能打印
SPI Boot 模式	0	0	忽略	使能打印
		1	0	使能打印
			1	关闭打印
		2	0	关闭打印
			1	使能打印
		3	忽略	关闭打印
	1	忽略	忽略	关闭打印

[1] Register：RTC_CNTL_RTC_STORE4_REG[0]
[2] eFuse：EFUSE_UART_PRINT_CONTROL

USB Serial/JTAG 控制器的 ROM 日志打印控制如下表：

表 8.3-2. USB Serial/JTAG 控制器的 ROM 日志打印控制

启动模式	Register[a]	eFuse_1[b]	eFuse_2[c]	eFuse_3[d]	ROM 日志打印至 USB Serial/JTAG 控制器
Download Boot 模式	忽略	0	0	0	使能打印
		其他			关闭打印
SPI Boot 模式	0	1	1	忽略	使能打印
		00/10/01		0	使能打印
		00/10/01		1	关闭打印
	1	忽略	忽略	忽略	关闭打印

[a] Register：RTC_CNTL_RTC_STORE4_REG[0]

[b] eFuse_1: EFUSE_DIS_USB_SERIAL_JTAG

[c] eFuse_2：在 Download Boot 模式下为 EFUSE_DIS_USB_SERIAL_JTAG_DOWNLOAD_MODE，在 SPI Boot 模式下为 EFUSE_DIS_USB_OTG

[d] eFuse_3：在 Download Boot 模式下为 EFUSE_DIS_DOWNLOAD_MODE，在 SPI Boot 模式下为 EFUSE_DIS_USB_PRINT

说明：

RTC_CNTL_RTC_STORE4_REG[0] 的值可以任意多次读写，而 eFuse 仅可烧写一次。因此，RTC_CNTL_RTC_STORE4_REG[0] 可用于临时关闭 ROM 日志打印，而 eFuse 可用于永久关闭 ROM 日志打印。

8.4 VDD_SPI 电压控制

芯片复位时，GPIO45 可用于选择 VDD_SPI 电压：

- GPIO45 = 0 时，VDD_SPI 由 VDD3P3_RTC 通过电阻 R_{SPI} 后供电（电压典型值为 3.3 V）；更多信息见《ESP32-S3 技术规格书》中图：ESP32-S3 电源管理。
- GPIO45 = 1 时，VDD_SPI 可选择由内置 LDO 供电（电压为 1.8 V）。

EFUSE_VDD_SPI_FORCE 设置为 1 时，可关闭上述功能。此时 VDD_SPI 电压由 EFUSE_VDD_SPI_TIEH 的值决定：

- EFUSE_VDD_SPI_TIEH = 0 时，VDD_SPI 连接 1.8 V LDO；
- EFUSE_VDD_SPI_TIEH = 1 时，VDD_SPI 连接 VDD3P3_RTC。

8.5 JTAG 信号源控制

在系统启动早期阶段，GPIO3 与 EFUSE_DIS_PAD_JTAG、EFUSE_DIS_USB_JTAG 和 EFUSE_STRAP_JTAG_SEL 一起控制 JTAG 信号源，见表 8.5-1。

表 8.5-1. JTAG 信号源控制

JTAG 信号源	EFUSE_DIS_PAD_JTAG	EFUSE_DIS_USB_JTAG	EFUSE_STRAP_JTAG_SEL	GPIO3
USB 串口/JTAG 控制器	**0**[a]	**0**	**0**	忽略
	0	0	1	1
	1	0	忽略	忽略
JTAG 管脚 MTDI、MTCK、MTMS 和 MTDO	0	0	1	0
	0	1	忽略	忽略
JTAG 关闭	1	1	忽略	忽略

[a] **加粗**表示默认值和默认配置

第 9 章

中断矩阵 (INTERRUPT)

9.1 概述

ESP32-S3 中断矩阵将任一外部中断源单独分配到双核 CPU 的任一外部中断上，以便在外设中断信号产生后，及时通知 CPU0 或 CPU1 进行处理。

外部中断源必须经中断矩阵分配至 CPU0/CPU1 外部中断，主要是因为：

- ESP32-S3 有 99 个外部中断源，但每个 CPU 只有 32 个中断。将这些外部中断源映射至 CPU0 中断或 CPU1 中断需要使用中断矩阵。
- 通过中断矩阵，可以根据应用需要，将一个外部中断源映射至多个 CPU0 中断或 CPU1 中断。

9.2 主要特性

- 接收 99 个外部中断源作为输入
- 生成 26 个 CPU0 的外部中断和 26 个 CPU1 的外部中断作为输出。

  注意，CPU0 剩余的 6 个中断和 CPU1 剩余的 6 个中断均为内部中断
- 支持屏蔽 CPU 的 NMI 类型中断
- 支持查询外部中断源当前的中断状态

中断矩阵的结构如图 9.2-1 所示。

所有外部中断源产生的中断信号均可由 CPU0 或 CPU1 进行处理。配置 CPU0 中断寄存器（图 9.2-1 Core0 Interrupt Reg 模块）可将外部中断源分配给 CPU0 的外部中断，此时外部中断源产生的中断信号会由 CPU0 响应处理。同理配置 CPU1 中断寄存器（图 9.2-1 Core1 Interrupt Reg 模块）可将中断信号交由 CPU1 响应处理。也可将外部中断源同时分配给 CPU0 和 CPU1, 此时 CPU0 和 CPU1 都会接收到中断信号。

9.3 功能描述

9.3.1 外部中断源

ESP32-S3 共有 99 个外部中断源。表 9.3-1 列出了所有外部中断源，以及对应的中断配置寄存器与中断状态寄存器。

- “No.”：表示外部中断源序号，范围：0 ~ 98
- “中断源”：表示所有外部中断源
- “配置寄存器”：用于将外部中断源分配至 CPU0/CPU1 外部中断

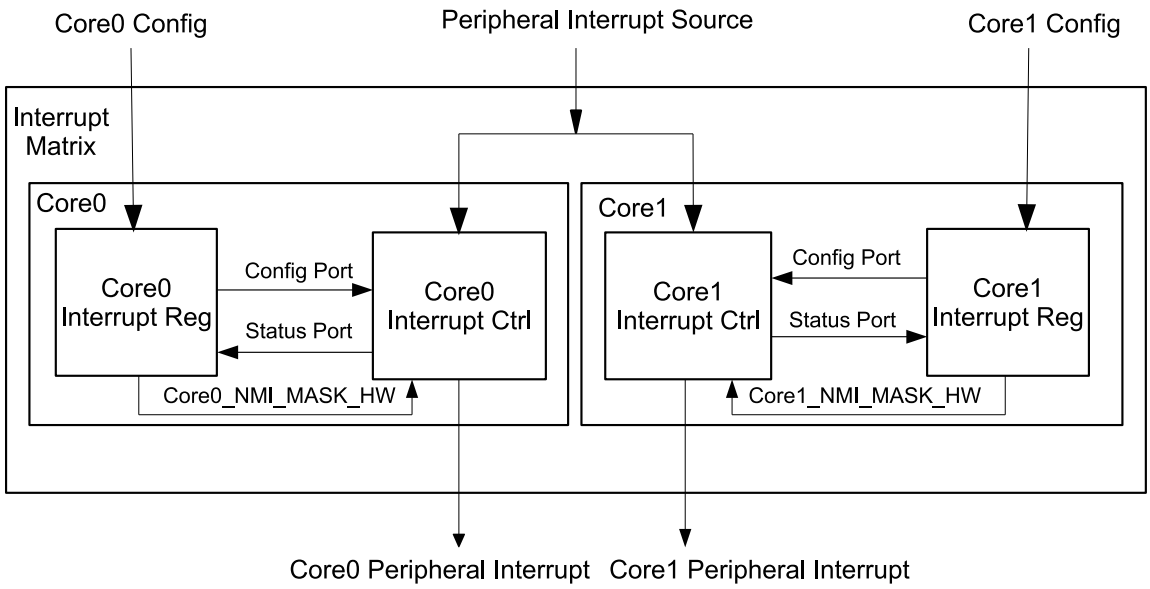


图 9.2-1. 中断矩阵结构图

- “状态寄存器”：用于读取中断源的中断状态
  - “状态寄存器 - 位”：表示在状态寄存器中的比特位置
  - “状态寄存器 - 名称”：表示状态寄存器的名称

中断源与同一行中断配置寄存器和中断状态寄存器位一一对应，例如：中断源 MAC_INTR 对应的中断配置寄存器为 INTERRUPT_CORE*x*_MAC_INTR_MAP_REG，对应的中断状态寄存器位为 INTERRUPT_CORE*x*_INTR_STATUS_0_REG 的 0 比特。

注意，表格中 CORE*x* 可以是 CORE0 (CPU0) 或 CORE1 (CPU1)。

表 9.3-1. CPU 外部中断配置寄存器、外部中断状态寄存器、外部中断源

No.	中断源	配置寄存器	状态寄存器	
			位	名称
0	MAC_INTR	INTERRUPT_COREx_MAC_INTR_MAP_REG	0	INTERRUPT_COREx_INTR_STATUS_0_REG
1	MAC_NMI	INTERRUPT_COREx_MAC_NMI_MAP_REG	1	
2	PWR_INTR	INTERRUPT_COREx_PWR_INTR_MAP_REG	2	
3	BB_INT	INTERRUPT_COREx_BB_INT_MAP_REG	3	
4	BT_MAC_INT	INTERRUPT_COREx_BT_MAC_INT_MAP_REG	4	
5	BT_BB_INT	INTERRUPT_COREx_BT_BB_INT_MAP_REG	5	
6	BT_BB_NMI	INTERRUPT_COREx_BT_BB_NMI_MAP_REG	6	
7	RWBT_IRQ	INTERRUPT_COREx_RWBT_IRQ_MAP_REG	7	
8	RWBLE_IRQ	INTERRUPT_COREx_RWBLE_IRQ_MAP_REG	8	
9	RWBT_NMI	INTERRUPT_COREx_RWBT_NMI_MAP_REG	9	
10	RWBLE_NMI	INTERRUPT_COREx_RWBLE_NMI_MAP_REG	10	
11	I2C_MST_INT	INTERRUPT_COREx_I2C_MST_INT_MAP_REG	11	
12	保留	保留	12	
13	保留	保留	13	
14	UHCI0_INTR	INTERRUPT_COREx_UHCI0_INTR_MAP_REG	14	
15	保留	保留	15	
16	GPIO_INTERRUPT_CPU	INTERRUPT_COREx_GPIO_INTERRUPT_CPU_MAP_REG	16	
17	GPIO_INTERRUPT_CPU_NMI	INTERRUPT_COREx_GPIO_INTERRUPT_CPU_NMI_MAP_REG	17	
18	保留	保留	18	
19	保留	保留	19	
20	SPI_INTR_1	INTERRUPT_COREx_SPI_INTR_1_MAP_REG	20	
21	SPI_INTR_2	INTERRUPT_COREx_SPI_INTR_2_MAP_REG	21	
22	SPI_INTR_3	INTERRUPT_COREx_SPI_INTR_3_MAP_REG	22	
23	保留	保留	23	
24	LCD_CAM_INT	INTERRUPT_COREx_LCD_CAM_INT_MAP_REG	24	
25	I2S0_INT	INTERRUPT_COREx_I2S0_INT_MAP_REG	25	
26	I2S1_INT	INTERRUPT_COREx_I2S1_INT_MAP_REG	26	
27	UART_INTR	INTERRUPT_COREx_UART_INTR_MAP_REG	27	
28	UART1_INTR	INTERRUPT_COREx_UART1_INTR_MAP_REG	28	
29	UART2_INTR	INTERRUPT_COREx_UART2_INTR_MAP_REG	29	
30	SDIO_HOST_INTERRUPT	INTERRUPT_COREx_SDIO_HOST_INTERRUPT_MAP_REG	30	
31	PWM0_INTR	INTERRUPT_COREx_PWM0_INTR_MAP_REG	31	
32	PWM1_INTR	INTERRUPT_COREx_PWM1_INTR_MAP_REG	0	INTERRUPT_COREx_INTR_STATUS_1_REG
33	保留	保留	1	
34	保留	保留	2	

No.	中断源	配置寄存器	状态寄存器	
			位	名称
35	LEDC_INT	INTERRUPT_COREx_LEDC_INT_MAP_REG	3	INTERRUPT_COREx_INTR_STATUS_1_REG
36	EFUSE_INT	INTERRUPT_COREx_EFUSE_INT_MAP_REG	4	
37	TWAI_INT	INTERRUPT_COREx_TWAI_INT_MAP_REG	5	
38	USB_INTR	INTERRUPT_COREx_USB_INTR_MAP_REG	6	
39	RTC_CORE_INTR	INTERRUPT_COREx_RTC_CORE_INTR_MAP_REG	7	
40	RMT_INTR	INTERRUPT_COREx_RMT_INTR_MAP_REG	8	
41	PCNT_INTR	INTERRUPT_COREx_PCNT_INTR_MAP_REG	9	
42	I2C_EXT0_INTR	INTERRUPT_COREx_I2C_EXT0_INTR_MAP_REG	10	
43	I2C_EXT1_INTR	INTERRUPT_COREx_I2C_EXT1_INTR_MAP_REG	11	
44	保留	保留	12	
45	保留	保留	13	
46	保留	保留	14	
47	保留	保留	15	
48	保留	保留	16	
49	保留	保留	17	
50	TG_T0_INT	INTERRUPT_COREx_TG_T0_INT_MAP_REG	18	
51	TG_T1_INT	INTERRUPT_COREx_TG_T1_INT_MAP_REG	19	
52	TG_WDT_INT	INTERRUPT_COREx_TG_WDT_INT_MAP_REG	20	
53	TG1_T0_INT	INTERRUPT_COREx_TG1_T0_INT_MAP_REG	21	
54	TG1_T1_INT	INTERRUPT_COREx_TG1_T1_INT_MAP_REG	22	
55	TG1_WDT_INT	INTERRUPT_COREx_TG1_WDT_INT_MAP_REG	23	
56	CACHE_IA_INT	INTERRUPT_COREx_CACHE_IA_INT_MAP_REG	24	
57	SYSTIMER_TARGET0_INT	INTERRUPT_COREx_SYSTIMER_TARGET0_INT_MAP_REG	25	
58	SYSTIMER_TARGET1_INT	INTERRUPT_COREx_SYSTIMER_TARGET1_INT_MAP_REG	26	
59	SYSTIMER_TARGET2_INT	INTERRUPT_COREx_SYSTIMER_TARGET2_INT_MAP_REG	27	
60	SPI_MEM_REJECT_INTR	INTERRUPT_COREx_SPI_MEM_REJECT_INTR_MAP_REG	28	
61	DCACHE_PRELOAD_INT	INTERRUPT_COREx_DCACHE_PRELOAD_INT_MAP_REG	29	
62	ICACHE_PRELOAD_INT	INTERRUPT_COREx_ICACHE_PRELOAD_INT_MAP_REG	30	
63	DCACHE_SYNC_INT	INTERRUPT_COREx_DCACHE_SYNC_INT_MAP_REG	31	
64	ICACHE_SYNC_INT	INTERRUPT_COREx_ICACHE_SYNC_INT_MAP_REG	0	INTERRUPT_COREx_INTR_STATUS_2_REG
65	APB_ADC_INT	INTERRUPT_COREX_APB_ADC_INT_MAP_REG	1	
66	DMA_IN_CH0_INT	INTERRUPT_COREX_DMA_IN_CH0_INT_MAP_REG	2	
67	DMA_IN_CH1_INT	INTERRUPT_COREx_DMA_IN_CH1_INT_MAP_REG	3	
68	DMA_IN_CH2_INT	INTERRUPT_COREx_DMA_IN_CH2_INT_MAP_REG	4	
69	DMA_IN_CH3_INT	INTERRUPT_COREx_DMA_IN_CH3_INT_MAP_REG	5	
70	DMA_IN_CH4_INT	INTERRUPT_COREx_DMA_IN_CH4_INT_MAP_REG	6	
71	DMA_OUT_CH0_INT	INTERRUPT_COREX_DMA_OUT_CH0_INT_MAP_REG	7	

GoBack

No.	中断源	配置寄存器	状态寄存器	
			位	名称
72	DMA_OUT_CH1_INT	INTERRUPT_COREx_DMA_OUT_CH1_INT_MAP_REG	8	INTERRUPT_COREx_INTR_STATUS_2_REG
73	DMA_OUT_CH2_INT	INTERRUPT_COREx_DMA_OUT_CH2_INT_MAP_REG	9	
74	DMA_OUT_CH3_INT	INTERRUPT_COREx_DMA_OUT_CH3_INT_MAP_REG	10	
75	DMA_OUT_CH4_INT	INTERRUPT_COREx_DMA_OUT_CH4_INT_MAP_REG	11	
76	RSA_INTR	INTERRUPT_COREx_RSA_INTR_MAP_REG	12	
77	AES_INTR	INTERRUPT_COREx_AES_INTR_MAP_REG	13	
78	SHA_INTR	INTERRUPT_COREx_SHA_INTR_MAP_REG	14	
79	CPU_INTR_FROM_CPU_0	INTERRUPT_COREx_CPU_INTR_FROM_CPU_0_MAP_REG	15	
80	CPU_INTR_FROM_CPU_1	INTERRUPT_COREx_CPU_INTR_FROM_CPU_1_MAP_REG	16	
81	CPU_INTR_FROM_CPU_2	INTERRUPT_COREx_CPU_INTR_FROM_CPU_2_MAP_REG	17	
82	CPU_INTR_FROM_CPU_3	INTERRUPT_COREx_CPU_INTR_FROM_CPU_3_MAP_REG	18	
83	保留	保留	19	
84	DMA_APB_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_DMA_APB_PMS_MONITOR_VIOLATE_INTR_MAP_REG	20	
85	CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	21	
86	CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	22	
87	CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	23	
88	CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR	INTERRUPT_COREx_CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	24	
89	CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	25	
90	CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	26	
91	CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	27	
92	CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR	INTERRUPT_COREx_CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	28	
93	BACKUP_PMS_VIOLATE_INT	INTERRUPT_COREx_BACKUP_PMS_VIOLATE_INTR_MAP_REG	29	
94	CACHE_COREO_ACS_INT	INTERRUPT_COREx_CACHE_COREO_ACS_INT_MAP_REG	30	
95	CACHE_CORE1_ACS_INT	INTERRUPT_COREx_CACHE_CORE1_ACS_INT_MAP_REG	31	
96	USB_DEVICE_INT	INTERRUPT_COREx_USB_DEVICE_INT_MAP_REG	0	INTERRUPT_COREx_INTR_STATUS_3_REG
97	PERI_BACKUP_INT	INTERRUPT_COREx_PERI_BACKUP_INT_MAP_REG	1	
98	DMA_EXTMEM_REJECT_INT	INTERRUPT_COREx_DMA_EXTMEM_REJECT_INT_MAP_REG	2	

GoBack

9.3.2 CPU 中断

每个 CPU 都有 32 个中断号 (0 ~ 31)，其中包括 26 个外部中断，6 个内部中断。

- 外部中断为外部中断源引发的中断，包括下面三种类型：
  - 电平触发类型中断：高电平触发，要求保持中断的电平状态直到 CPU*x* 响应；
  - 边沿触发类型中断：上升沿触发，此中断一旦产生，CPU*x* 即可响应；
  - NMI 中断：软件不可使用 CPU*x* 内部特殊寄存器屏蔽此类中断，World Controller 模块提供了屏蔽此中断的机制。更多信息，见章节 World Controller。
- 内部中断为 CPU*x* 内部自己产生的中断，包括下面三种类型：
  - 定时器中断：由内部定时器触发，可用于产生周期性的中断；
  - 软件中断：软件写特殊寄存器时将触发此中断；
  - 解析中断：用于性能监测和分析。

上述电平类型和边沿类型中断指 CPU 接收中断信号的方式。对于电平类型中断，在 CPU 响应此中断之前该中断信号的电平需要一直保持，如果电平提前掉下去 CPU 会丢失此次中断。对于边沿类型中断，CPU 会检测中断信号的边沿，当检测到边沿之后 CPU 会记录此次中断，此时中断信号可以提前拉低。

通过中断矩阵将外部中断源映射到任意一个外部中断，即可让 CPU 接收到中断源的中断信号。表 9.3-2 列出了所有的中断号对应的类型和优先级。

ESP32-S3 支持六级中断，同时支持中断嵌套，即低优先级中断可以被高优先级中断打断。在下表优先级一栏中，数字越大代表优先级越高。其中，NMI 中断拥有最高优先级，此类中断一经触发，CPU 必须处理。

表 9.3-2. CPU 中断

中断号	类别	种类	优先级
0	外部中断	电平触发	1
1	外部中断	电平触发	1
2	外部中断	电平触发	1
3	外部中断	电平触发	1
4	外部中断	电平触发	1
5	外部中断	电平触发	1
6	内部中断	定时器 0	1
7	内部中断	软件	1
8	外部中断	电平触发	1
9	外部中断	电平触发	1
10	外部中断	边沿触发	1
11	内部中断	解析	3
12	外部中断	电平触发	1
13	外部中断	电平触发	1
14	外部中断	NMI	NMI
15	内部中断	定时器 1	3
16	内部中断	定时器 2	5
17	外部中断	电平触发	1
18	外部中断	电平触发	1

中断号	类别	种类	优先级
19	外部中断	电平触发	2
20	外部中断	电平触发	2
21	外部中断	电平触发	2
22	外部中断	边沿触发	3
23	外部中断	电平触发	3
24	外部中断	电平触发	4
25	外部中断	电平触发	4
26	外部中断	电平触发	5
27	外部中断	电平触发	3
28	外部中断	边沿触发	4
29	内部中断	软件	3
30	外部中断	边沿触发	4
31	外部中断	电平触发	5

9.3.3 分配外部中断源至 CPU*x* 外部中断

在本小节中，我们将使用以下术语描述中断矩阵相关操作：

- Source_*Y*：代表某个外部中断源，其中 Y 为中断源编号，详见表 9.3-1。
- INTERRUPT_CORE*x*_SOURCE_*Y*_MAP_REG：CPU*x* 外部中断源 (Source_*Y*) 的中断配置寄存器。
- Interrupt_P：表示中断号为 Num_P 的外部中断，Num_P 的取值范围为 0 ~ 5、8 ~ 10、12 ~ 14、17 ~ 28、30 ~ 31，详见表 9.3-2。
- Interrupt_I：表示中断号为 Num_I 的内部中断，Num_I 的取值范围为 6、7、11、15、16、29，详见表 9.3-2。

9.3.3.1 分配一个外部中断源 Source_Y 至 CPU*x* 外部中断

将外部中断源 Source_Y 对应的寄存器 INTERRUPT_CORE*x*_SOURCE_*Y*_MAP_REG 配成 Num_P，即可将该中断源分配至序号为 Num_P 的外部中断 (Interrupt_P)。Num_P 可以取 CPU*x* 的任一外部中断号，包括 0 ~ 5、8 ~ 10、12 ~ 14、17 ~ 28、30 ~ 31。每个 CPU 中断可被多个外设共享。

9.3.3.2 分配多个外部中断源 Source_Y*n* 至 CPU*x* 外部中断

将各个中断源对应的寄存器 INTERRUPT_CORE*x*_SOURCE_Y*n*_MAP_REG 均配置成相同的 Num_P，即可将多个中断源 Source_Y*n* 分配至同一 CPU*x* 外部中断 Interrupt_P。上述任一外设中断均会触发 CPU*x* 外部中断 Interrupt_P。待中断触发后，CPU*x* 需查询中断状态寄存器，判断产生中断的外设。

9.3.3.3 关闭 CPU*x* 外部中断源 Source_Y

将中断源对应的寄存器 INTERRUPT_CORE*x*_SOURCE_*Y*_MAP_REG 配置成任意 Num_I，即可关闭外部中断源。这是因为任何被配置成 Num_I 的外部中断均无法连接至 CPU*x*，而且选择任一内部中断号 (6、7、11、15、16、29) 不会造成其他影响，可用于关闭外部中断。

9.3.4 关闭 CPU*x* 的 NMI 类型中断

表 9.3-2 中的 32 个中断，除 NMI 类型中断，其余中断均可通过软件配置 CPU 特殊寄存器 (INTENABLE) 进行屏蔽和使能。ESP32-S3 另外提供两种屏蔽 NMI 的机制：

- 断开外部中断源与 NMI 中断的连接，即将所有分配给 NMI 中断的外部中断源分配给其它中断；
- 不断开外部中断源与 NMI 中断的连接，但使用 World Controller 模块的 NMI 屏蔽功能进行屏蔽。更多信息，请参考 World Controller 章节。

9.3.5 查询外部中断源当前的中断状态

读取寄存器 INTERRUPT_CORE*x*_INTR_STATUS_*n*_REG（只读）中特定位的值可以获取 CPU*x* 外部中断源当前的中断状态。寄存器 INTERRUPT_CORE*x*_INTR_STATUS_*n*_REG 与外部中断源的对应关系如表 9.3-1 所示。

9.4 寄存器列表

本小节的所有地址均为相对于中断矩阵基地址的地址偏移量（相对地址），具体基地址请见章节 4 *系统和存储器* 中的表 4.3-3。

请查看章节 *寄存器的访问类型*，了解“访问”列缩写的含义。

9.4.1 CPU0 中断寄存器列表

名称	描述	地址	访问
配置寄存器			
INTERRUPT_CORE0_MAC_INTR_MAP_REG	MAC 中断配置寄存器	0x0000	R/W
INTERRUPT_CORE0_MAC_NMI_MAP_REG	MAC_NMI 中断配置寄存器	0x0004	R/W
INTERRUPT_CORE0_PWR_INTR_MAP_REG	PWR 中断配置寄存器	0x0008	R/W
INTERRUPT_CORE0_BB_INT_MAP_REG	BB 中断配置寄存器	0x000C	R/W
INTERRUPT_CORE0_BT_MAC_INT_MAP_REG	BB_MAC 中断配置寄存器	0x0010	R/W
INTERRUPT_CORE0_BT_BB_INT_MAP_REG	BT_BB 中断配置寄存器	0x0014	R/W
INTERRUPT_CORE0_BT_BB_NMI_MAP_REG	BT_BB_NMI 中断配置寄存器	0x0018	R/W
INTERRUPT_CORE0_RWBT_IRQ_MAP_REG	RWBT_IRQ 中断配置寄存器	0x001C	R/W
INTERRUPT_CORE0_RWBLE_IRQ_MAP_REG	RWBLE_IRQ 中断配置寄存器	0x0020	R/W
INTERRUPT_CORE0_RWBT_NMI_MAP_REG	RWBT_NMI 中断配置寄存器	0x0024	R/W
INTERRUPT_CORE0_RWBLE_NMI_MAP_REG	RWBLE_NMI 中断配置寄存器	0x0028	R/W
INTERRUPT_CORE0_I2C_MST_INT_MAP_REG	I2C_MST 中断配置寄存器	0x002C	R/W
INTERRUPT_CORE0_UHCI0_INTR_MAP_REG	UHCI0 中断配置寄存器	0x0038	R/W
INTERRUPT_CORE0_GPIO_INTERRUPT_CPU_MAP_REG	GPIO_INTERRUPT_CPU 中断配置寄存器	0x0040	R/W
INTERRUPT_CORE0_GPIO_INTERRUPT_CPU_NMI_MAP_REG	GPIO_INTERRUPT_CPU_NMI 中断配置寄存器	0x0044	R/W
INTERRUPT_CORE0_SPI_INTR_1_MAP_REG	SPI_INTR_1 中断配置寄存器	0x0050	R/W
INTERRUPT_CORE0_SPI_INTR_2_MAP_REG	SPI_INTR_2 中断配置寄存器	0x0054	R/W
INTERRUPT_CORE0_SPI_INTR_3_MAP_REG	SPI_INTR_3 中断配置寄存器	0x0058	R/W
INTERRUPT_CORE0_LCD_CAM_INT_MAP_REG	LCD_CAM 中断配置寄存器	0x0060	R/W
INTERRUPT_CORE0_I2S0_INT_MAP_REG	I2S0 中断配置寄存器	0x0064	R/W
INTERRUPT_CORE0_I2S1_INT_MAP_REG	I2S1 中断配置寄存器	0x0068	R/W
INTERRUPT_CORE0_UART_INTR_MAP_REG	UART 中断配置寄存器	0x006C	R/W
INTERRUPT_CORE0_UART1_INTR_MAP_REG	UART1 中断配置寄存器	0x0070	R/W
INTERRUPT_CORE0_UART2_INTR_MAP_REG	UART2 中断配置寄存器	0x0074	R/W
INTERRUPT_CORE0_SDIO_HOST_INTERRUPT_MAP_REG	SDIO_HOST 中断配置寄存器	0x0078	R/W

GoBack

名称	描述	地址	访问
INTERRUPT_COREO_PWMO_INTR_MAP_REG	PWMO 中断配置寄存器	0x007C	R/W
INTERRUPT_COREO_PWM1_INTR_MAP_REG	PWM1 中断配置寄存器	0x0080	R/W
INTERRUPT_COREO_LEDC_INT_MAP_REG	LEDC 中断配置寄存器	0x008C	R/W
INTERRUPT_COREO_EFUSE_INT_MAP_REG	EFUSE 中断配置寄存器	0x0090	R/W
INTERRUPT_COREO_TWAI_INT_MAP_REG	TWAI 中断配置寄存器	0x0094	R/W
INTERRUPT_COREO_USB_INTR_MAP_REG	USB 中断配置寄存器	0x0098	R/W
INTERRUPT_COREO_RTC_CORE_INTR_MAP_REG	RTC_CORE 中断配置寄存器	0x009C	R/W
INTERRUPT_COREO_RMT_INTR_MAP_REG	RMT 中断配置寄存器	0x00A0	R/W
INTERRUPT_COREO_PCNT_INTR_MAP_REG	PCNT 中断配置寄存器	0x00A4	R/W
INTERRUPT_COREO_I2C_EXTO_INTR_MAP_REG	I2C_EXTO 中断配置寄存器	0x00A8	R/W
INTERRUPT_COREO_I2C_EXT1_INTR_MAP_REG	I2C_EXT1 中断配置寄存器	0x00AC	R/W
INTERRUPT_COREO_TG_TO_INT_MAP_REG	TG_TO 中断配置寄存器	0x00C8	R/W
INTERRUPT_COREO_TG_T1_INT_MAP_REG	TG_T1 中断配置寄存器	0x00CC	R/W
INTERRUPT_COREO_TG_WDT_INT_MAP_REG	TG_WDT 中断配置寄存器	0x00D0	R/W
INTERRUPT_COREO_TG1_TO_INT_MAP_REG	TG1_TO 中断配置寄存器	0x00D4	R/W
INTERRUPT_COREO_TG1_T1_INT_MAP_REG	TG1_T1 中断配置寄存器	0x00D8	R/W
INTERRUPT_COREO_TG1_WDT_INT_MAP_REG	TG1_WDT 中断配置寄存器	0x00DC	R/W
INTERRUPT_COREO_CACHE_IA_INT_MAP_REG	CACHE_IA 中断配置寄存器	0x00E0	R/W
INTERRUPT_COREO_SYSTIMER_TARGETO_INT_MAP_REG	SYSTIMER_TARGETO 中断配置寄存器	0x00E4	R/W
INTERRUPT_COREO_SYSTIMER_TARGET1_INT_MAP_REG	SYSTIMER_TARGET1 中断配置寄存器	0x00E8	R/W
INTERRUPT_COREO_SYSTIMER_TARGET2_INT_MAP_REG	SYSTIMER_TARGET2 中断配置寄存器	0x00EC	R/W
INTERRUPT_COREO_SPI_MEM_REJECT_INTR_MAP_REG	SPI_MEM_REJECT 中断配置寄存器	0x00F0	R/W
INTERRUPT_COREO_DCACHE_PRELOAD_INT_MAP_REG	DCACHE_PRELAOD 中断配置寄存器	0x00F4	R/W
INTERRUPT_COREO_ICACHE_PRELOAD_INT_MAP_REG	ICACHE_PRELOAD 中断配置寄存器	0x00F8	R/W
INTERRUPT_COREO_DCACHE_SYNC_INT_MAP_REG	DCACHE_SYNC 中断配置寄存器	0x00FC	R/W
INTERRUPT_COREO_ICACHE_SYNC_INT_MAP_REG	ICACHE_SYNC 中断配置寄存器	0x0100	R/W
INTERRUPT_COREO_APB_ADC_INT_MAP_REG	APB_ADC 中断配置寄存器	0x0104	R/W
INTERRUPT_COREO_DMA_IN_CHO_INT_MAP_REG	DMA_IN_CHO 中断配置寄存器	0x0108	R/W
INTERRUPT_COREO_DMA_IN_CH1_INT_MAP_REG	DMA_IN_CH1 中断配置寄存器	0x010C	R/W

GoBack

名称	描述	地址	访问
INTERRUPT_COREO_DMA_IN_CH2_INT_MAP_REG	DMA_IN_CH2 中断配置寄存器	0x0110	R/W
INTERRUPT_COREO_DMA_IN_CH3_INT_MAP_REG	DMA_IN_CH3 中断配置寄存器	0x0114	R/W
INTERRUPT_COREO_DMA_IN_CH4_INT_MAP_REG	DMA_IN_CH4 中断配置寄存器	0x0118	R/W
INTERRUPT_COREO_DMA_OUT_CHO_INT_MAP_REG	DMA_OUT_CHO 中断配置寄存器	0x011C	R/W
INTERRUPT_COREO_DMA_OUT_CH1_INT_MAP_REG	DMA_OUT_CH1 中断配置寄存器	0x0120	R/W
INTERRUPT_COREO_DMA_OUT_CH2_INT_MAP_REG	DMA_OUT_CH2 中断配置寄存器	0x0124	R/W
INTERRUPT_COREO_DMA_OUT_CH3_INT_MAP_REG	DMA_OUT_CH3 中断配置寄存器	0x0128	R/W
INTERRUPT_COREO_DMA_OUT_CH4_INT_MAP_REG	DMA_OUT_CH4 中断配置寄存器	0x012C	R/W
INTERRUPT_COREO_RSA_INT_MAP_REG	RSA 中断配置寄存器	0x0130	R/W
INTERRUPT_COREO_AES_INT_MAP_REG	AES 中断配置寄存器	0x0134	R/W
INTERRUPT_COREO_SHA_INT_MAP_REG	SHA 中断配置寄存器	0x0138	R/W
INTERRUPT_COREO_CPU_INTR_FROM_CPU_0_MAP_REG	CPU_INTR_FROM_CPU_0 中断配置寄存器	0x013C	R/W
INTERRUPT_COREO_CPU_INTR_FROM_CPU_1_MAP_REG	CPU_INTR_FROM_CPU_1 中断配置寄存器	0x0140	R/W
INTERRUPT_COREO_CPU_INTR_FROM_CPU_2_MAP_REG	CPU_INTR_FROM_CPU_2 中断配置寄存器	0x0144	R/W
INTERRUPT_COREO_CPU_INTR_FROM_CPU_3_MAP_REG	CPU_INTR_FROM_CPU_3 中断配置寄存器	0x0148	R/W
INTERRUPT_COREO_DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR_MAP_REG	dma_pms_monitor_violatile 中断配置寄存器	0x0150	R/W
INTERRUPT_COREO_CORE_0_IRAMO_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_IRam0_pms_monitor_violatile 中断配置寄存器	0x0154	R/W
INTERRUPT_COREO_CORE_0_DRAMO_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_DRam0_pms_monitor_violatile 中断配置寄存器	0x0158	R/W
INTERRUPT_COREO_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_PIF_pms_monitor_violatile 中断配置寄存器	0x015C	R/W
INTERRUPT_COREO_CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	core0_PIF_pms_monitor_violatile_size 中断配置寄存器	0x0160	R/W
INTERRUPT_COREO_CORE_1_IRAMO_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_IRam0_pms_monitor_violatile 中断配置寄存器	0x0164	R/W
INTERRUPT_COREO_CORE_1_DRAMO_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_DRam0_pms_monitor_violatile 中断配置寄存器	0x0168	R/W

GoBack

名称	描述	地址	访问
INTERRUPT_COREO_CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_PIF_pms_monitor_violatile 中断配置寄存器	0x016C	R/W
INTERRUPT_COREO_CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	core1_PIF_pms_monitor_violatile_size 中断配置寄存器	0x0170	R/W
INTERRUPT_COREO_BACKUP_PMS_VIOLATE_INTR_MAP_REG	BACKUP_PMS_MONITOR_VIOLATILE 中断配置寄存器	0x0174	R/W
INTERRUPT_COREO_CACHE_COREO_ACS_INT_MAP_REG	CACHE_COREO_ACS 中断配置寄存器	0x0178	R/W
INTERRUPT_COREO_CACHE_CORE1_ACS_INT_MAP_REG	CACHE_CORE1_ACS 中断配置寄存器	0x017C	R/W
INTERRUPT_COREO_USB_DEVICE_INT_MAP_REG	USB_DEVICE 中断配置寄存器	0x0180	R/W
INTERRUPT_COREO_PERI_BACKUP_INT_MAP_REG	PERI_BACKUP 中断配置寄存器	0x0184	R/W
INTERRUPT_COREO_DMA_EXTMEM_REJECT_INT_MAP_REG	DMA_EXTMEM_REJECT 中断配置寄存器	0x0188	R/W
状态寄存器			
INTERRUPT_COREO_INTR_STATUS_0_REG	中断状态寄存器 0	0x018C	RO
INTERRUPT_COREO_INTR_STATUS_1_REG	中断状态寄存器 1	0x0190	RO
INTERRUPT_COREO_INTR_STATUS_2_REG	中断状态寄存器 2	0x0194	RO
INTERRUPT_COREO_INTR_STATUS_3_REG	中断状态寄存器 3	0x0198	RO
时钟寄存器			
INTERRUPT_COREO_CLOCK_GATE_REG	时钟门控寄存器	0x019C	R/W
版本寄存器			
INTERRUPT_COREO_DATE_REG	版本控制寄存器	0x07FC	R/W

9.4.2 CPU1 中断寄存器列表

名称	描述	地址	访问
配置寄存器			
INTERRUPT_CORE1_MAC_INTR_MAP_REG	MAC 中断配置寄存器	0x0800	R/W
INTERRUPT_CORE1_MAC_NMI_MAP_REG	MAC_NMI 中断配置寄存器	0x0804	R/W
INTERRUPT_CORE1_PWR_INTR_MAP_REG	PWR 中断配置寄存器	0x0808	R/W
INTERRUPT_CORE1_BB_INT_MAP_REG	BB 中断配置寄存器	0x080C	R/W

名称	描述	地址	访问
INTERRUPT_CORE1_BT_MAC_INT_MAP_REG	BB_MAC 中断配置寄存器	0x0810	R/W
INTERRUPT_CORE1_BT_BB_INT_MAP_REG	BT_BB 中断配置寄存器	0x0814	R/W
INTERRUPT_CORE1_BT_BB_NMI_MAP_REG	BT_BB_NMI 中断配置寄存器	0x0818	R/W
INTERRUPT_CORE1_RWBT_IRQ_MAP_REG	RWBT_IRQ 中断配置寄存器	0x081C	R/W
INTERRUPT_CORE1_RWBLE_IRQ_MAP_REG	RWBLE_IRQ 中断配置寄存器	0x0820	R/W
INTERRUPT_CORE1_RWBT_NMI_MAP_REG	RWBT_NMI 中断配置寄存器	0x0824	R/W
INTERRUPT_CORE1_RWBLE_NMI_MAP_REG	RWBLE_NMI 中断配置寄存器	0x0828	R/W
INTERRUPT_CORE1_I2C_MST_INT_MAP_REG	I2C_MST 中断配置寄存器	0x082C	R/W
INTERRUPT_CORE1_UHCI0_INTR_MAP_REG	UHCI0 中断配置寄存器	0x0838	R/W
INTERRUPT_CORE1_GPIO_INTERRUPT_CPU_MAP_REG	GPIO_INTERRUPT_CPU 中断配置寄存器	0x0840	R/W
INTERRUPT_CORE1_GPIO_INTERRUPT_CPU_NMI_MAP_REG	GPIO_INTERRUPT_CPU_NMI 中断配置寄存器	0x0844	R/W
INTERRUPT_CORE1_SPI_INTR_1_MAP_REG	SPI_INTR_1 中断配置寄存器	0x0850	R/W
INTERRUPT_CORE1_SPI_INTR_2_MAP_REG	SPI_INTR_2 中断配置寄存器	0x0854	R/W
INTERRUPT_CORE1_SPI_INTR_3_MAP_REG	SPI_INTR_3 中断配置寄存器	0x0858	R/W
INTERRUPT_CORE1_LCD_CAM_INT_MAP_REG	LCD_CAM 中断配置寄存器	0x0860	R/W
INTERRUPT_CORE1_I2S0_INT_MAP_REG	I2S0 中断配置寄存器	0x0864	R/W
INTERRUPT_CORE1_I2S1_INT_MAP_REG	I2S1 中断配置寄存器	0x0868	R/W
INTERRUPT_CORE1_UART_INTR_MAP_REG	UART 中断配置寄存器	0x086C	R/W
INTERRUPT_CORE1_UART1_INTR_MAP_REG	UART1 中断配置寄存器	0x0870	R/W
INTERRUPT_CORE1_UART2_INTR_MAP_REG	UART2 中断配置寄存器	0x0874	R/W
INTERRUPT_CORE1_SDIO_HOST_INTERRUPT_MAP_REG	SDIO_HOST 中断配置寄存器	0x0878	R/W
INTERRUPT_CORE1_PWM0_INTR_MAP_REG	PWM0 中断配置寄存器	0x087C	R/W
INTERRUPT_CORE1_PWM1_INTR_MAP_REG	PWM1 中断配置寄存器	0x0880	R/W
INTERRUPT_CORE1_LEDC_INT_MAP_REG	LEDC 中断配置寄存器	0x088C	R/W
INTERRUPT_CORE1_EFUSE_INT_MAP_REG	EFUSE 中断配置寄存器	0x0890	R/W
INTERRUPT_CORE1_TWAI_INT_MAP_REG	TWAI 中断配置寄存器	0x0894	R/W
INTERRUPT_CORE1_USB_INTR_MAP_REG	USB 中断配置寄存器	0x0898	R/W
INTERRUPT_CORE1_RTC_CORE_INTR_MAP_REG	RTC_CORE 中断配置寄存器	0x089C	R/W
INTERRUPT_CORE1_RMT_INTR_MAP_REG	RMT 中断配置寄存器	0x08A0	R/W

GoBack

名称	描述	地址	访问
INTERRUPT_CORE1_PCNT_INTR_MAP_REG	PCNT 中断配置寄存器	0x08A4	R/W
INTERRUPT_CORE1_I2C_EXT0_INTR_MAP_REG	I2C_EXT0 中断配置寄存器	0x08A8	R/W
INTERRUPT_CORE1_I2C_EXT1_INTR_MAP_REG	I2C_EXT1 中断配置寄存器	0x08AC	R/W
INTERRUPT_CORE1_TG_T0_INT_MAP_REG	TG_T0 中断配置寄存器	0x08C8	R/W
INTERRUPT_CORE1_TG_T1_INT_MAP_REG	TG_T1 中断配置寄存器	0x08CC	R/W
INTERRUPT_CORE1_TG_WDT_INT_MAP_REG	TG_WDT 中断配置寄存器	0x08D0	R/W
INTERRUPT_CORE1_TG1_T0_INT_MAP_REG	TG1_T0 中断配置寄存器	0x08D4	R/W
INTERRUPT_CORE1_TG1_T1_INT_MAP_REG	TG1_T1 中断配置寄存器	0x08D8	R/W
INTERRUPT_CORE1_TG1_WDT_INT_MAP_REG	TG1_WDT 中断配置寄存器	0x08DC	R/W
INTERRUPT_CORE1_CACHE_IA_INT_MAP_REG	CACHE_IA 中断配置寄存器	0x08E0	R/W
INTERRUPT_CORE1_SYSTIMER_TARGET0_INT_MAP_REG	SYSTIMER_TARGET0 中断配置寄存器	0x08E4	R/W
INTERRUPT_CORE1_SYSTIMER_TARGET1_INT_MAP_REG	SYSTIMER_TARGET1 中断配置寄存器	0x08E8	R/W
INTERRUPT_CORE1_SYSTIMER_TARGET2_INT_MAP_REG	SYSTIMER_TARGET2 中断配置寄存器	0x08EC	R/W
INTERRUPT_CORE1_SPI_MEM_REJECT_INTR_MAP_REG	SPI_MEM_REJECT 中断配置寄存器	0x08F0	R/W
INTERRUPT_CORE1_DCACHE_PRELOAD_INT_MAP_REG	DCACHE_PRELAOD 中断配置寄存器	0x08F4	R/W
INTERRUPT_CORE1_ICACHE_PRELOAD_INT_MAP_REG	ICACHE_PRELOAD 中断配置寄存器	0x08F8	R/W
INTERRUPT_CORE1_DCACHE_SYNC_INT_MAP_REG	DCACHE_SYNC 中断配置寄存器	0x08FC	R/W
INTERRUPT_CORE1_ICACHE_SYNC_INT_MAP_REG	ICACHE_SYNC 中断配置寄存器	0x0900	R/W
INTERRUPT_CORE1_APB_ADC_INT_MAP_REG	APB_ADC 中断配置寄存器	0x0904	R/W
INTERRUPT_CORE1_DMA_IN_CH0_INT_MAP_REG	DMA_IN_CH0 中断配置寄存器	0x0908	R/W
INTERRUPT_CORE1_DMA_IN_CH1_INT_MAP_REG	DMA_IN_CH1 中断配置寄存器	0x090C	R/W
INTERRUPT_CORE1_DMA_IN_CH2_INT_MAP_REG	DMA_IN_CH2 中断配置寄存器	0x0910	R/W
INTERRUPT_CORE1_DMA_IN_CH3_INT_MAP_REG	DMA_IN_CH3 中断配置寄存器	0x0914	R/W
INTERRUPT_CORE1_DMA_IN_CH4_INT_MAP_REG	DMA_IN_CH4 中断配置寄存器	0x0918	R/W
INTERRUPT_CORE1_DMA_OUT_CH0_INT_MAP_REG	DMA_OUT_CH0 中断配置寄存器	0x091C	R/W
INTERRUPT_CORE1_DMA_OUT_CH1_INT_MAP_REG	DMA_OUT_CH1 中断配置寄存器	0x0920	R/W
INTERRUPT_CORE1_DMA_OUT_CH2_INT_MAP_REG	DMA_OUT_CH2 中断配置寄存器	0x0924	R/W
INTERRUPT_CORE1_DMA_OUT_CH3_INT_MAP_REG	DMA_OUT_CH3 中断配置寄存器	0x0928	R/W
INTERRUPT_CORE1_DMA_OUT_CH4_INT_MAP_REG	DMA_OUT_CH4 中断配置寄存器	0x092C	R/W

GoBack

名称	描述	地址	访问
INTERRUPT_CORE1_RSA_INT_MAP_REG	RSA 中断配置寄存器	0x0930	R/W
INTERRUPT_CORE1_AES_INT_MAP_REG	AES 中断配置寄存器	0x0934	R/W
INTERRUPT_CORE1_SHA_INT_MAP_REG	SHA 中断配置寄存器	0x0938	R/W
INTERRUPT_CORE1_CPU_INTR_FROM_CPU_0_MAP_REG	CPU_INTR_FROM_CPU_0 中断配置寄存器	0x093C	R/W
INTERRUPT_CORE1_CPU_INTR_FROM_CPU_1_MAP_REG	CPU_INTR_FROM_CPU_1 中断配置寄存器	0x0940	R/W
INTERRUPT_CORE1_CPU_INTR_FROM_CPU_2_MAP_REG	CPU_INTR_FROM_CPU_2 中断配置寄存器	0x0944	R/W
INTERRUPT_CORE1_CPU_INTR_FROM_CPU_3_MAP_REG	CPU_INTR_FROM_CPU_3 中断配置寄存器	0x0948	R/W
INTERRUPT_CORE1_DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR_MAP_REG	dma_pms_monitor_violatile 中断配置寄存器	0x0950	R/W
INTERRUPT_CORE1_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_IRam0_pms_monitor_violatile 中断配置寄存器	0x0954	R/W
INTERRUPT_CORE1_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_DRam0_pms_monitor_violatile 中断配置寄存器	0x0958	R/W
INTERRUPT_CORE1_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_PIF_pms_monitor_violatile 中断配置寄存器	0x095C	R/W
INTERRUPT_CORE1_CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	core0_PIF_pms_monitor_violatile_size 中断配置寄存器	0x0960	R/W
INTERRUPT_CORE1_CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_IRam0_pms_monitor_violatile 中断配置寄存器	0x0964	R/W
INTERRUPT_CORE1_CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_DRam0_pms_monitor_violatile 中断配置寄存器	0x0968	R/W
INTERRUPT_CORE1_CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_PIF_pms_monitor_violatile 中断配置寄存器	0x096C	R/W
INTERRUPT_CORE1_CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	core1_PIF_pms_monitor_violatile_size 中断配置寄存器	0x0970	R/W
INTERRUPT_CORE1_BACKUP_PMS_VIOLATE_INTR_MAP_REG	BACKUP_PMS_MONITOR_VIOLATILE 中断配置寄存器	0x0974	R/W
INTERRUPT_CORE1_CACHE_COREO_ACS_INT_MAP_REG	CACHE_COREO_ACS 中断配置寄存器 REG	0x0978	R/W
INTERRUPT_CORE1_CACHE_CORE1_ACS_INT_MAP_REG	CACHE_CORE1_ACS 中断配置寄存器 REG	0x097C	R/W
INTERRUPT_CORE1_USB_DEVICE_INT_MAP_REG	USB_DEVICE 中断配置寄存器	0x0980	R/W

GoBack

名称	描述	地址	访问
INTERRUPT_CORE1_PERI_BACKUP_INT_MAP_REG	PERI_BACKUP 中断配置寄存器	0x0984	R/W
INTERRUPT_CORE1_DMA_EXTMEM_REJECT_INT_MAP_REG	DMA_EXTMEM_REJECT 中断配置寄存器	0x0988	R/W
状态寄存器			
INTERRUPT_CORE1_INTR_STATUS_0_REG	中断状态寄存器 0	0x098C	RO
INTERRUPT_CORE1_INTR_STATUS_1_REG	中断状态寄存器 1	0x0990	RO
INTERRUPT_CORE1_INTR_STATUS_2_REG	中断状态寄存器 2	0x0994	RO
INTERRUPT_CORE1_INTR_STATUS_3_REG	中断状态寄存器 3	0x0998	RO
时钟寄存器			
INTERRUPT_CORE1_CLOCK_GATE_REG	时钟门控寄存器	0x099C	R/W
版本寄存器			
INTERRUPT_CORE1_DATE_REG	版本控制寄存器	0x0FFC	R/W

9.5 寄存器

本小节的所有地址均为相对于中断矩阵基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

9.5.1 CPU0 中断寄存器

Register 9.1. INTERRUPT_CORE0_*MAC_INTR*_MAP_REG (0x0000)

Register 9.2. INTERRUPT_CORE0_*MAC_NMI*_MAP_REG (0x0004)

Register 9.3. INTERRUPT_CORE0_*PWR_INTR*_MAP_REG (0x0008)

Register 9.4. INTERRUPT_CORE0_*BB_INT*_MAP_REG (0x000C)

Register 9.5. INTERRUPT_CORE0_*BT_MAC_INT*_MAP_REG (0x0010)

Register 9.6. INTERRUPT_CORE0_*BT_BB_INT*_MAP_REG (0x0014)

Register 9.7. INTERRUPT_CORE0_*BT_BB_NMI*_MAP_REG (0x0018)

Register 9.8. INTERRUPT_CORE0_*RWBT_IRQ*_MAP_REG (0x001C)

Register 9.9. INTERRUPT_CORE0_*RWBLE_IRQ*_MAP_REG (0x0020)

Register 9.10. INTERRUPT_CORE0_*RWBT_NMI*_MAP_REG (0x0024)

Register 9.11. INTERRUPT_CORE0_*RWBLE_NMI*_MAP_REG (0x0028)

Register 9.12. INTERRUPT_CORE0_*I2C_MST_INT*_MAP_REG (0x002C)

Register 9.13. INTERRUPT_CORE0_*UHCI0_INTR*_MAP_REG (0x0038)

Register 9.14. INTERRUPT_CORE0_*GPIO_INTERRUPT_CPU*_MAP_REG (0x0040)

Register 9.15. INTERRUPT_CORE0_*GPIO_INTERRUPT_CPU_NMI*_MAP_REG (0x0044)

Register 9.16. INTERRUPT_CORE0_*SPI_INTR_1*_MAP_REG (0x0050)

Register 9.17. INTERRUPT_CORE0_*SPI_INTR_2*_MAP_REG (0x0054)

Register 9.18. INTERRUPT_CORE0_*SPI_INTR_3*_MAP_REG (0x0058)

Register 9.19. INTERRUPT_CORE0_*LCD_CAM_INT*_MAP_REG (0x0060)

Register 9.20. INTERRUPT_CORE0_*I2S0_INT*_MAP_REG (0x0064)

Register 9.21. INTERRUPT_CORE0_*I2S1_INT*_MAP_REG (0x0068)

Register 9.22. INTERRUPT_CORE0_*UART_INTR*_MAP_REG (0x006C)

Register 9.23. INTERRUPT_CORE0_*UART1_INTR*_MAP_REG (0x0070)

Register 9.24. INTERRUPT_CORE0_*UART2_INTR*_MAP_REG (0x0074)

Register 9.25. INTERRUPT_CORE0_*SDIO_HOST_INTERRUPT*_MAP_REG (0x0078)

Register 9.26. INTERRUPT_CORE0_*PWM0_INTR*_MAP_REG (0x007C)

Register 9.27. INTERRUPT_CORE0_*PWM1_INTR*_MAP_REG (0x0080)

Register 9.28. INTERRUPT_CORE0_*LEDC_INT*_MAP_REG (0x008C)

Register 9.29. INTERRUPT_CORE0_*EFUSE_INT*_MAP_REG (0x0090)

Register 9.30. INTERRUPT_CORE0_*TWAI_INT*_MAP_REG (0x0094)
Register 9.31. INTERRUPT_CORE0_*USB_INTR*_MAP_REG (0x0098)
Register 9.32. INTERRUPT_CORE0_*RTC_CORE_INTR*_MAP_REG (0x009C)
Register 9.33. INTERRUPT_CORE0_*RMT_INTR*_MAP_REG (0x00A0)
Register 9.34. INTERRUPT_CORE0_*PCNT_INTR*_MAP_REG (0x00A4)
Register 9.35. INTERRUPT_CORE0_*I2C_EXT0_INTR*_MAP_REG (0x00A8)
Register 9.36. INTERRUPT_CORE0_*I2C_EXT1_INTR*_MAP_REG (0x00AC)
Register 9.37. INTERRUPT_CORE0_*TG_T0_INT*_MAP_REG (0x00C8)
Register 9.38. INTERRUPT_CORE0_*TG_T1_INT*_MAP_REG (0x00CC)
Register 9.39. INTERRUPT_CORE0_*TG_WDT_INT*_MAP_REG (0x00D0)
Register 9.40. INTERRUPT_CORE0_*TG1_T0_INT*_MAP_REG (0x00D4)
Register 9.41. INTERRUPT_CORE0_*TG1_T1_INT*_MAP_REG (0x00D8)
Register 9.42. INTERRUPT_CORE0_*TG1_WDT_INT*_MAP_REG (0x00DC)
Register 9.43. INTERRUPT_CORE0_*CACHE_IA_INT*_MAP_REG (0x00E0)
Register 9.44. INTERRUPT_CORE0_*SYSTIMER_TARGET0_INT*_MAP_REG (0x00E4)
Register 9.45. INTERRUPT_CORE0_*SYSTIMER_TARGET1_INT*_MAP_REG (0x00E8)
Register 9.46. INTERRUPT_CORE0_*SYSTIMER_TARGET2_INT*_MAP_REG (0x00EC)
Register 9.47. INTERRUPT_CORE0_*SPI_MEM_REJECT_INTR*_MAP_REG (0x00F0)
Register 9.48. INTERRUPT_CORE0_*DCACHE_PRELOAD_INT*_MAP_REG (0x00F4)
Register 9.49. INTERRUPT_CORE0_*ICACHE_PRELOAD_INT*_MAP_REG (0x00F8)
Register 9.50. INTERRUPT_CORE0_*DCACHE_SYNC_INT*_MAP_REG (0x00FC)
Register 9.51. INTERRUPT_CORE0_*ICACHE_SYNC_INT*_MAP_REG (0x0100)
Register 9.52. INTERRUPT_CORE0_*APB_ADC_INT*_MAP_REG (0x0104)
Register 9.53. INTERRUPT_CORE0_*DMA_IN_CH0_INT*_MAP_REG (0x0108)
Register 9.54. INTERRUPT_CORE0_*DMA_IN_CH1_INT*_MAP_REG (0x010C)
Register 9.55. INTERRUPT_CORE0_*DMA_IN_CH2_INT*_MAP_REG (0x0110)
Register 9.56. INTERRUPT_CORE0_*DMA_IN_CH3_INT*_MAP_REG (0x0114)
Register 9.57. INTERRUPT_CORE0_*DMA_IN_CH4_INT*_MAP_REG (0x0118)
Register 9.58. INTERRUPT_CORE0_*DMA_OUT_CH0_INT*_MAP_REG (0x011C)
Register 9.59. INTERRUPT_CORE0_*DMA_OUT_CH1_INT*_MAP_REG (0x0120)
Register 9.60. INTERRUPT_CORE0_*DMA_OUT_CH2_INT*_MAP_REG (0x0124)
Register 9.61. INTERRUPT_CORE0_*DMA_OUT_CH3_INT*_MAP_REG (0x0128)
Register 9.62. INTERRUPT_CORE0_*DMA_OUT_CH4_INT*_MAP_REG (0x012C)
Register 9.63. INTERRUPT_CORE0_*RSA_INT*_MAP_REG (0x0130)
Register 9.64. INTERRUPT_CORE0_*AES_INT*_MAP_REG (0x0134)

Register 9.65. INTERRUPT_COREO_*SHA_INT*_MAP_REG (0x0138)

Register 9.66. INTERRUPT_COREO_*CPU_INTR_FROM_CPU_0*_MAP_REG (0x013C)

Register 9.67. INTERRUPT_COREO_*CPU_INTR_FROM_CPU_1*_MAP_REG (0x0140)

Register 9.68. INTERRUPT_COREO_*CPU_INTR_FROM_CPU_2*_MAP_REG (0x0144)

Register 9.69. INTERRUPT_COREO_*CPU_INTR_FROM_CPU_3*_MAP_REG (0x0148)

Register 9.70. INTERRUPT_COREO_*DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0150)

Register 9.71. INTERRUPT_COREO_*CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0154)

Register 9.72. INTERRUPT_COREO_*CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0158)

Register 9.73. INTERRUPT_COREO_*CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x015C)

Register 9.74. INTERRUPT_COREO_*CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR*_MAP_REG (0x0160)

Register 9.75. INTERRUPT_COREO_*CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0164)

Register 9.76. INTERRUPT_COREO_*CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0168)

Register 9.77. INTERRUPT_COREO_*CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x016C)

Register 9.78. INTERRUPT_COREO_*CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR*_MAP_REG (0x0170)

Register 9.79. INTERRUPT_COREO_*BACKUP_PMS_VIOLATE_INTR*_MAP_REG (0x0174)

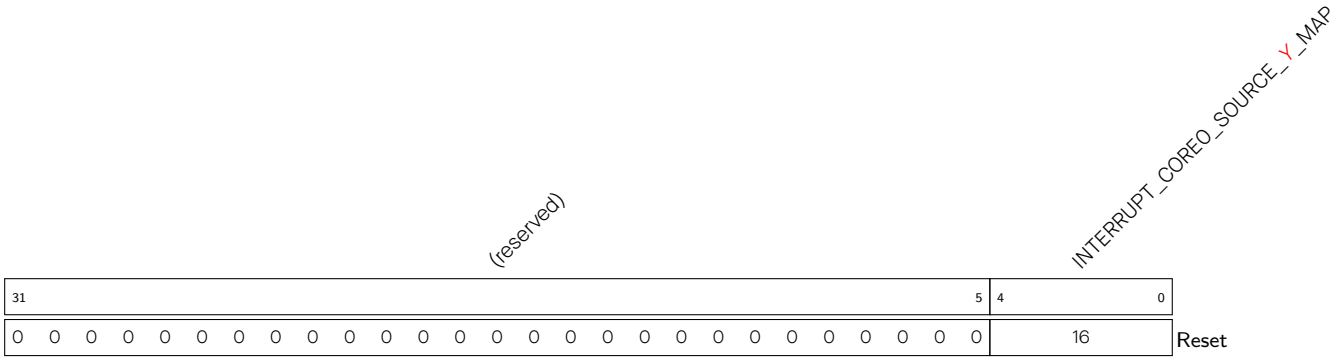
Register 9.80. INTERRUPT_COREO_*CACHE_COREO_ACS_INT*_MAP_REG (0x0178)

Register 9.81. INTERRUPT_COREO_*CACHE_CORE1_ACS_INT*_MAP_REG (0x017C)

Register 9.82. INTERRUPT_COREO_*USB_DEVICE_INT*_MAP_REG (0x0180)

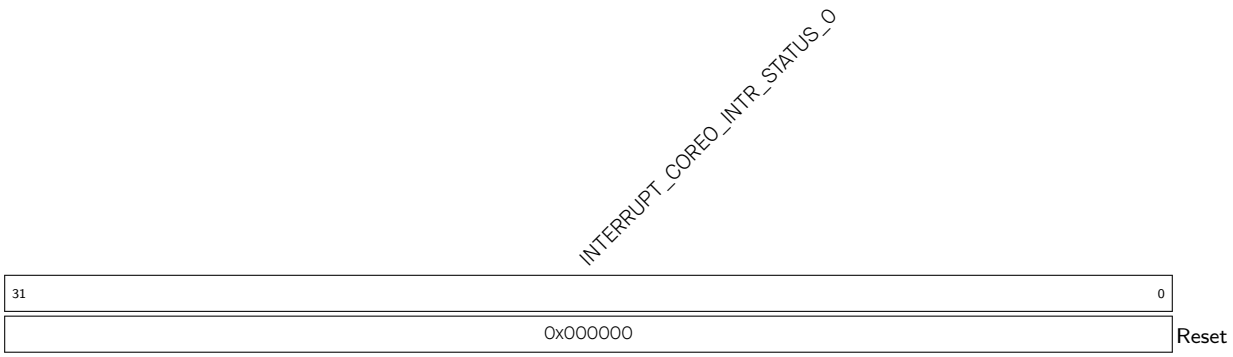
Register 9.83. INTERRUPT_COREO_*PERI_BACKUP_INT*_MAP_REG (0x0184)

Register 9.84. INTERRUPT_COREO_*DMA_EXTMEM_REJECT_INT*_MAP_REG (0x0188)



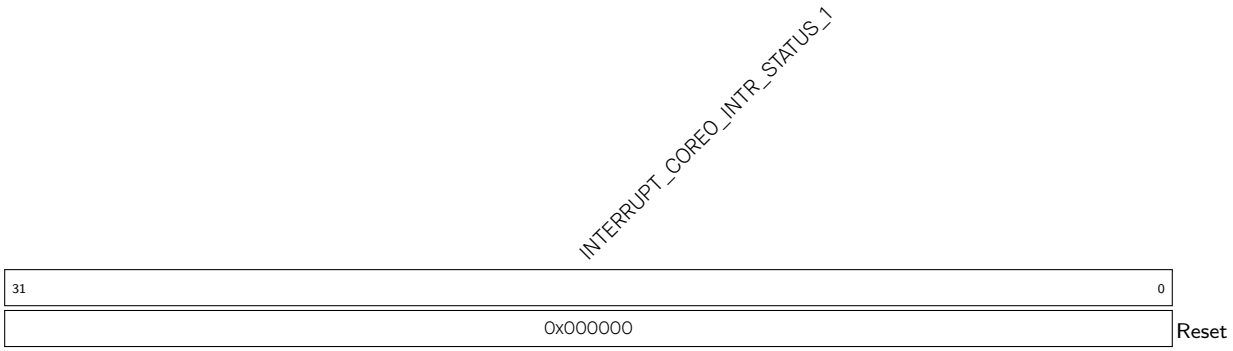
INTERRUPT_COREO_SOURCE_*Y*_MAP 将中断源 Source_*Y* 的中断信号映射至 CPU0 外部中断，可配置为 0 ~ 5、8 ~ 10、12 ~ 14、17 ~ 28 和 30 ~ 31，其它值无效。中断源 Source_*Y* 见表 9.3-1。(R/W)

Register 9.85. INTERRUPT_COREO_INTR_STATUS_0_REG (0x018C)



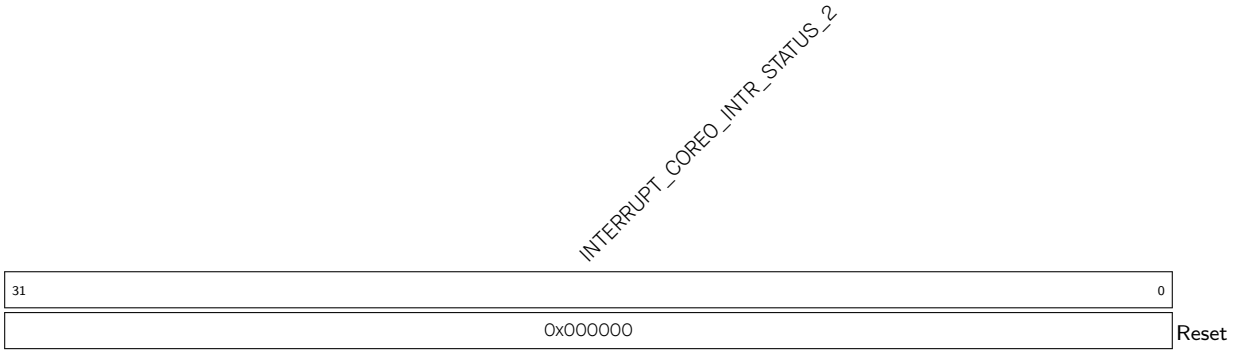
INTERRUPT_COREO_INTR_STATUS_0 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：0 ~ 31。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 9.86. INTERRUPT_COREO_INTR_STATUS_1_REG (0x0190)



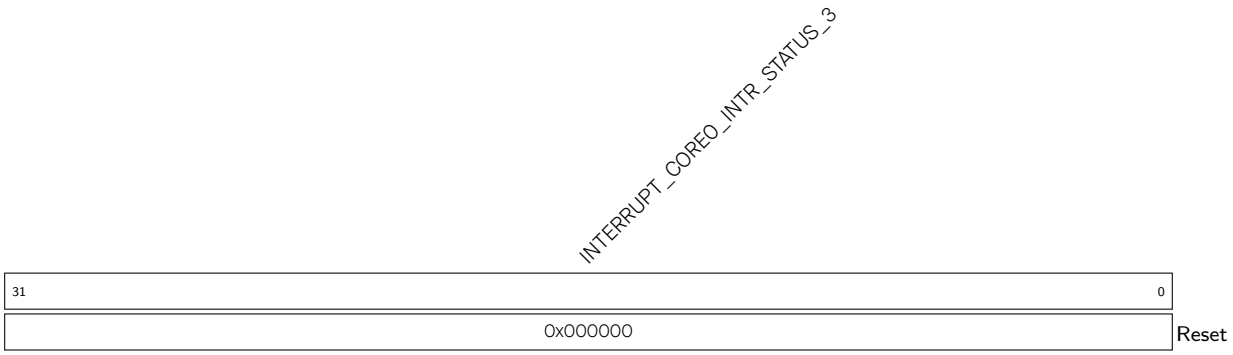
INTERRUPT_COREO_INTR_STATUS_1 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：32 ~ 63。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 9.87. INTERRUPT_COREO_INTR_STATUS_2_REG (0x0194)



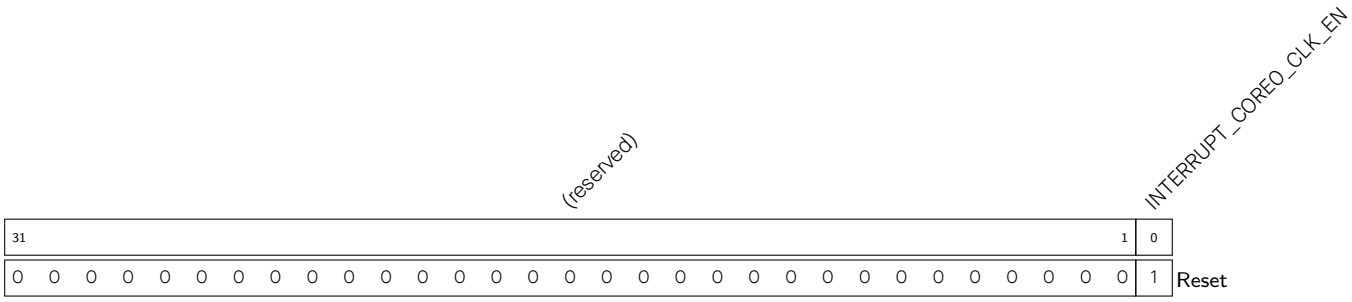
INTERRUPT_COREO_INTR_STATUS_2 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：64 ~ 95。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 9.88. INTERRUPT_COREO_INTR_STATUS_3_REG (0x0198)



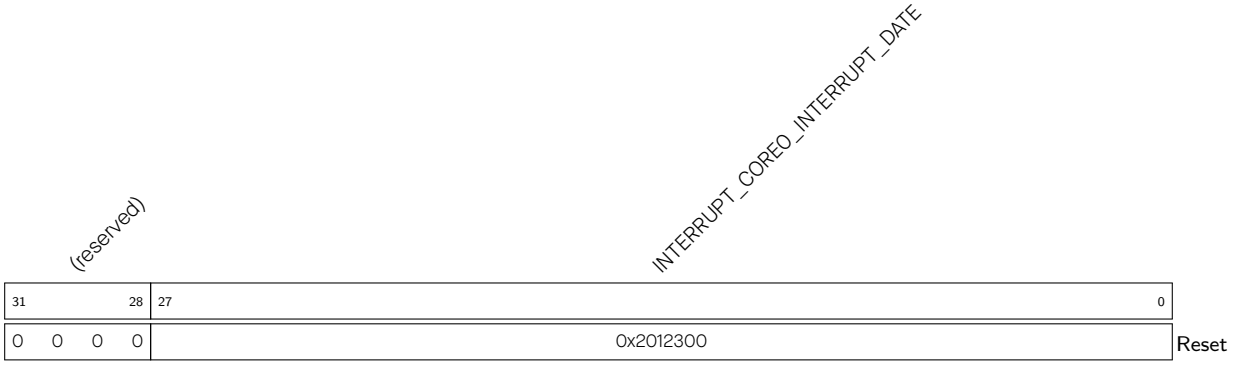
INTERRUPT_COREO_INTR_STATUS_3 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：96 ~ 98。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 9.89. INTERRUPT_COREO_CLOCK_GATE_REG (0x019C)



INTERRUPT_COREO_CLK_EN 中断矩阵的时钟门控控制位。(R/W)

Register 9.90. INTERRUPT_COREO_DATE_REG (0x07FC)



INTERRUPT_COREO_INTERRUPT_DATE 版本控制寄存器。(R/W)

9.5.2 CPU1 中断寄存器

Register 9.91. INTERRUPT_CORE1_*MAC_INTR*_MAP_REG (0x0800)

Register 9.92. INTERRUPT_CORE1_*MAC_NMI*_MAP_REG (0x0804)

Register 9.93. INTERRUPT_CORE1_*PWR_INTR*_MAP_REG (0x0808)

Register 9.94. INTERRUPT_CORE1_*BB_INT*_MAP_REG (0x080C)

Register 9.95. INTERRUPT_CORE1_*BT_MAC_INT*_MAP_REG (0x0810)

Register 9.96. INTERRUPT_CORE1_*BT_BB_INT*_MAP_REG (0x0814)

Register 9.97. INTERRUPT_CORE1_*BT_BB_NMI*_MAP_REG (0x0818)

Register 9.98. INTERRUPT_CORE1_*RWBT_IRQ*_MAP_REG (0x081C)

Register 9.99. INTERRUPT_CORE1_*RWBLE_IRQ*_MAP_REG (0x0820)

Register 9.100. INTERRUPT_CORE1_*RWBT_NMI*_MAP_REG (0x0824)

Register 9.101. INTERRUPT_CORE1_*RWBLE_NMI*_MAP_REG (0x0828)

Register 9.102. INTERRUPT_CORE1_*I2C_MST_INT*_MAP_REG (0x082C)

Register 9.103. INTERRUPT_CORE1_*UHCI0_INTR*_MAP_REG (0x0838)

Register 9.104. INTERRUPT_CORE1_*GPIO_INTERRUPT_CPU*_MAP_REG (0x0840)

Register 9.105. INTERRUPT_CORE1_*GPIO_INTERRUPT_CPU_NMI*_MAP_REG (0x0844)

Register 9.106. INTERRUPT_CORE1_*SPI_INTR_1*_MAP_REG (0x0850)

Register 9.107. INTERRUPT_CORE1_*SPI_INTR_2*_MAP_REG (0x0854)

Register 9.108. INTERRUPT_CORE1_*SPI_INTR_3*_MAP_REG (0x0858)

Register 9.109. INTERRUPT_CORE1_*LCD_CAM_INT*_MAP_REG (0x0860)

Register 9.110. INTERRUPT_CORE1_*I2S0_INT*_MAP_REG (0x0864)

Register 9.111. INTERRUPT_CORE1_*I2S1_INT*_MAP_REG (0x0868)

Register 9.112. INTERRUPT_CORE1_*UART_INTR*_MAP_REG (0x086C)

Register 9.113. INTERRUPT_CORE1_*UART1_INTR*_MAP_REG (0x0870)

Register 9.114. INTERRUPT_CORE1_*UART2_INTR*_MAP_REG (0x0874)

Register 9.115. INTERRUPT_CORE1_*SDIO_HOST_INTERRUPT*_MAP_REG (0x0878)

Register 9.116. INTERRUPT_CORE1_*PWM0_INTR*_MAP_REG (0x087C)

Register 9.117. INTERRUPT_CORE1_*PWM1_INTR*_MAP_REG (0x0880)

Register 9.118. INTERRUPT_CORE1_*LEDC_INT*_MAP_REG (0x088C)

Register 9.119. INTERRUPT_CORE1_*EFUSE_INT*_MAP_REG (0x0890)

Register 9.120. INTERRUPT_CORE1_*TWAI_INT*_MAP_REG (0x0894)

Register 9.121. INTERRUPT_CORE1_*USB_INTR*_MAP_REG (0x0898)

Register 9.122. INTERRUPT_CORE1_*RTC_CORE_INTR*_MAP_REG (0x089C)

Register 9.123. INTERRUPT_CORE1_*RMT_INTR*_MAP_REG (0x08A0)

Register 9.124. INTERRUPT_CORE1_*PCNT_INTR*_MAP_REG (0x08A4)

Register 9.125. INTERRUPT_CORE1_*I2C_EXT0_INTR*_MAP_REG (0x08A8)

Register 9.126. INTERRUPT_CORE1_*I2C_EXT1_INTR*_MAP_REG (0x08AC)

Register 9.127. INTERRUPT_CORE1_*TG_T0_INT*_MAP_REG (0x08C8)

Register 9.128. INTERRUPT_CORE1_*TG_T1_INT*_MAP_REG (0x08CC)

Register 9.129. INTERRUPT_CORE1_*TG_WDT_INT*_MAP_REG (0x08D0)

Register 9.130. INTERRUPT_CORE1_*TG1_T0_INT*_MAP_REG (0x08D4)

Register 9.131. INTERRUPT_CORE1_*TG1_T1_INT*_MAP_REG (0x08D8)

Register 9.132. INTERRUPT_CORE1_*TG1_WDT_INT*_MAP_REG (0x08DC)

Register 9.133. INTERRUPT_CORE1_*CACHE_IA_INT*_MAP_REG (0x08E0)

Register 9.134. INTERRUPT_CORE1_*SYSTIMER_TARGET0_INT*_MAP_REG (0x08E4)

Register 9.135. INTERRUPT_CORE1_*SYSTIMER_TARGET1_INT*_MAP_REG (0x08E8)

Register 9.136. INTERRUPT_CORE1_*SYSTIMER_TARGET2_INT*_MAP_REG (0x08EC)

Register 9.137. INTERRUPT_CORE1_*SPI_MEM_REJECT_INTR*_MAP_REG (0x08F0)

Register 9.138. INTERRUPT_CORE1_*DCACHE_PRELOAD_INT*_MAP_REG (0x08F4)

Register 9.139. INTERRUPT_CORE1_*ICACHE_PRELOAD_INT*_MAP_REG (0x08F8)

Register 9.140. INTERRUPT_CORE1_*DCACHE_SYNC_INT*_MAP_REG (0x08FC)

Register 9.141. INTERRUPT_CORE1_*ICACHE_SYNC_INT*_MAP_REG (0x0900)

Register 9.142. INTERRUPT_CORE1_*APB_ADC_INT*_MAP_REG (0x0904)

Register 9.143. INTERRUPT_CORE1_*DMA_IN_CH0_INT*_MAP_REG (0x0908)

Register 9.144. INTERRUPT_CORE1_*DMA_IN_CH1_INT*_MAP_REG (0x090C)

Register 9.145. INTERRUPT_CORE1_*DMA_IN_CH2_INT*_MAP_REG (0x0910)

Register 9.146. INTERRUPT_CORE1_*DMA_IN_CH3_INT*_MAP_REG (0x0914)

Register 9.147. INTERRUPT_CORE1_*DMA_IN_CH4_INT*_MAP_REG (0x0918)

Register 9.148. INTERRUPT_CORE1_*DMA_OUT_CH0_INT*_MAP_REG (0x091C)

Register 9.149. INTERRUPT_CORE1_*DMA_OUT_CH1_INT*_MAP_REG (0x0920)

Register 9.150. INTERRUPT_CORE1_*DMA_OUT_CH2_INT*_MAP_REG (0x0924)

Register 9.151. INTERRUPT_CORE1_*DMA_OUT_CH3_INT*_MAP_REG (0x0928)

Register 9.152. INTERRUPT_CORE1_*DMA_OUT_CH4_INT*_MAP_REG (0x092C)

Register 9.153. INTERRUPT_CORE1_*RSA_INT*_MAP_REG (0x0930)

Register 9.154. INTERRUPT_CORE1_*AES_INT*_MAP_REG (0x0934)

Register 9.155. INTERRUPT_CORE1_*SHA_INT*_MAP_REG (0x0938)

Register 9.156. INTERRUPT_CORE1_*CPU_INTR_FROM_CPU_0*_MAP_REG (0x093C)

Register 9.157. INTERRUPT_CORE1_*CPU_INTR_FROM_CPU_1*_MAP_REG (0x0940)

Register 9.158. INTERRUPT_CORE1_*CPU_INTR_FROM_CPU_2*_MAP_REG (0x0944)

Register 9.159. INTERRUPT_CORE1_*CPU_INTR_FROM_CPU_3*_MAP_REG (0x0948)

Register 9.160. INTERRUPT_CORE1_*DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0950)

Register 9.161. INTERRUPT_CORE1_*CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0954)

Register 9.162. INTERRUPT_CORE1_*CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0958)

Register 9.163. INTERRUPT_CORE1_*CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x095C)

Register 9.164. INTERRUPT_CORE1_*CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR*_MAP_REG (0x0960)

Register 9.165. INTERRUPT_CORE1_*CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0964)

Register 9.166. INTERRUPT_CORE1_*CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0968)

Register 9.167. INTERRUPT_CORE1_*CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x096C)

Register 9.168. INTERRUPT_CORE1_*CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR*_MAP_REG (0x0970)

Register 9.169. INTERRUPT_CORE1_*BACKUP_PMS_VIOLATE_INTR*_MAP_REG (0x0974)

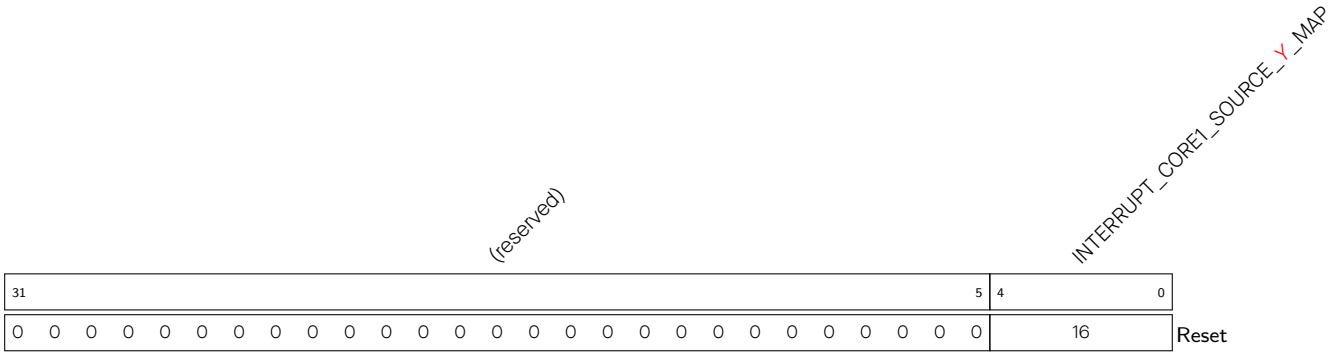
Register 9.170. INTERRUPT_CORE1_*CACHE_CORE0_ACS_INT*_MAP_REG (0x0978)

Register 9.171. INTERRUPT_CORE1_*CACHE_CORE1_ACS_INT*_MAP_REG (0x097C)

Register 9.172. INTERRUPT_CORE1_*USB_DEVICE_INT*_MAP_REG (0x0980)

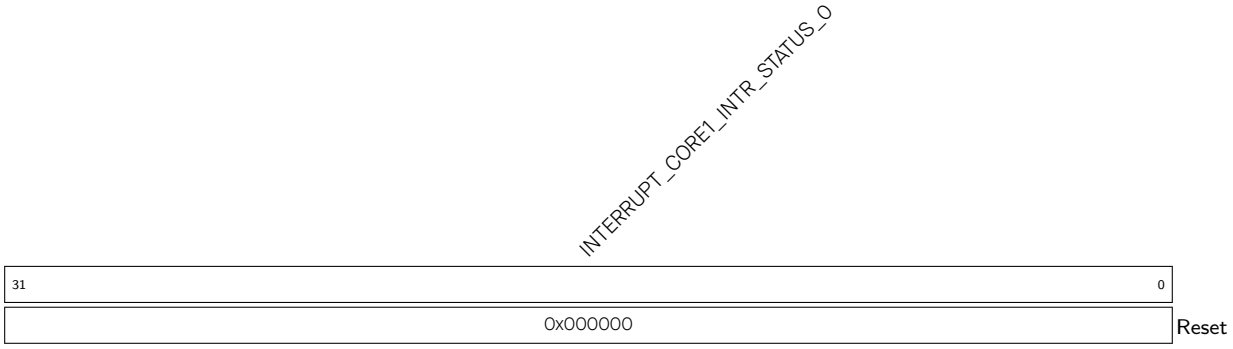
Register 9.173. INTERRUPT_CORE1_*PERI_BACKUP_INT*_MAP_REG (0x0984)

Register 9.174. INTERRUPT_CORE1_*DMA_EXTMEM_REJECT_INT*_MAP_REG (0x0988)



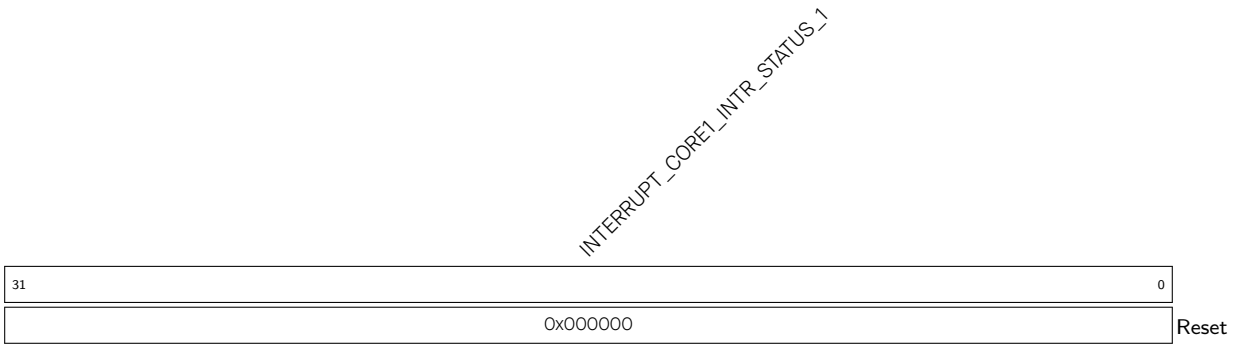
INTERRUPT_CORE1_SOURCE_*Y*_MAP 将中断源 Source_*Y* 的中断信号映射至 CPU1 外部中断，可配置为 0 ~ 5、8 ~ 10、12 ~ 14、17 ~ 28 和 30 ~ 31，其它值无效。中断源 Source_*Y* 见表 9.3-1。(R/W)

Register 9.175. INTERRUPT_CORE1_INTR_STATUS_0_REG (0x098C)



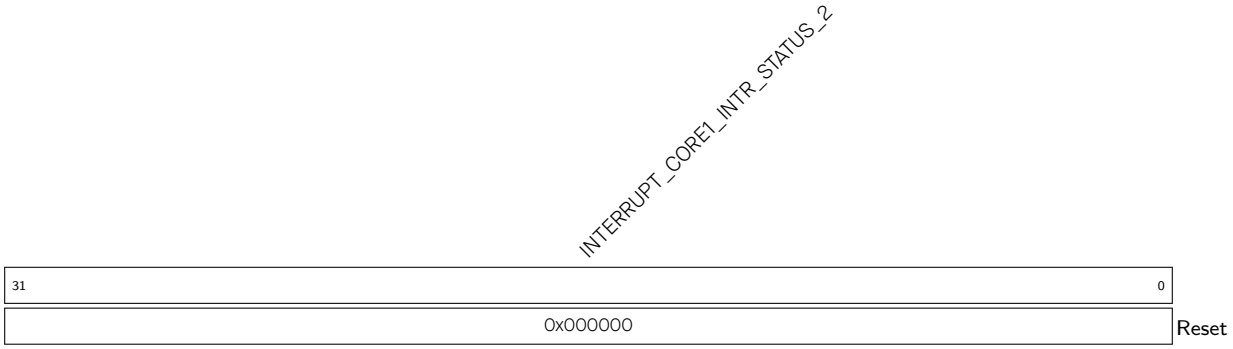
INTERRUPT_CORE1_INTR_STATUS_0 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：0 ~ 31。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 9.176. INTERRUPT_CORE1_INTR_STATUS_1_REG (0x0990)



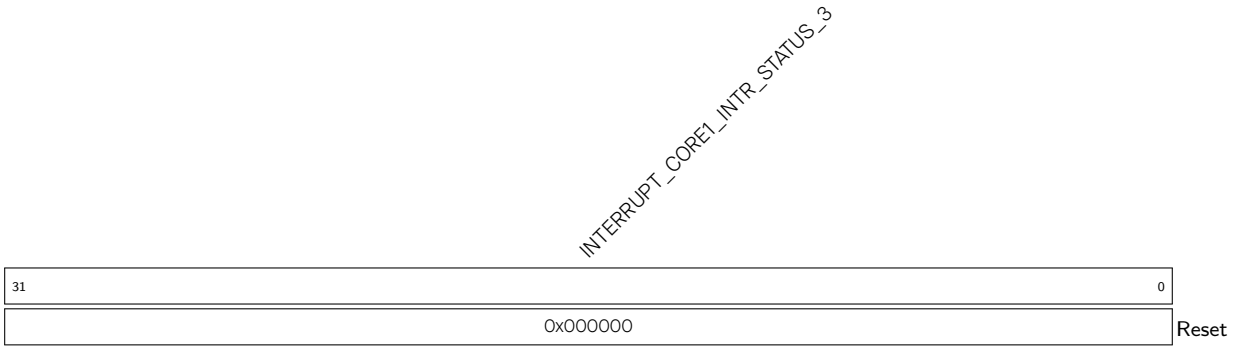
INTERRUPT_CORE1_INTR_STATUS_1 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：32 ~ 63。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 9.177. INTERRUPT_CORE1_INTR_STATUS_2_REG (0x0994)



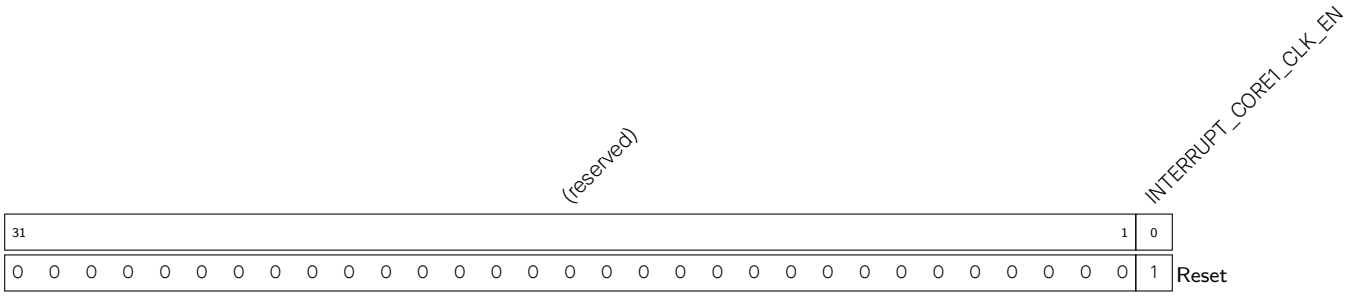
INTERRUPT_CORE1_INTR_STATUS_2 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：64 ~ 95。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 9.178. INTERRUPT_CORE1_INTR_STATUS_3_REG (0x0998)



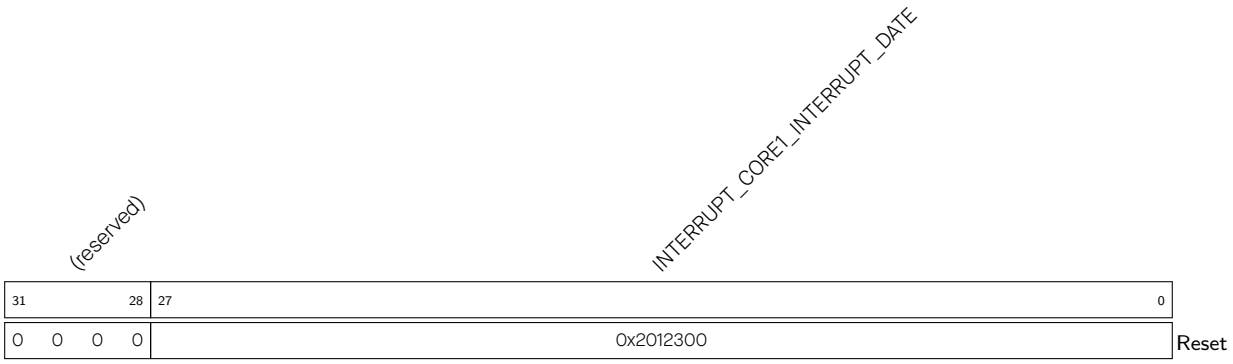
INTERRUPT_CORE1_INTR_STATUS_3 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：96 ~ 98。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 9.179. INTERRUPT_CORE1_CLOCK_GATE_REG (0x099C)



INTERRUPT_CORE1_CLK_EN 中断矩阵的时钟门控控制位。(R/W)

Register 9.180. INTERRUPT_CORE1_DATE_REG (0x0FFC)



INTERRUPT_CORE1_INTERRUPT_DATE 版本控制寄存器。(R/W)

第 10 章

低功耗管理 (RTC_CNTL)

10.1 概述

ESP32-S3 拥有一个先进的电源管理单元，可在低功耗协处理器的配合下，灵活打开或关闭芯片的不同电源域，协助客户在芯片工作性能、功耗控制和唤醒延迟之间实现最佳平衡。为了便利用户的使用，ESP32-S3 定义了四种最常见的电源域设置组合，对应四种预设功耗模式，可满足用户的常见场景需求，但也同时支持用户对某个电源域的独立控制，以满足一些复杂场景的功耗需求。ESP32-S3 内置 2 个超低功耗协处理器，允许芯片在绝大多数电源域均关闭的情况下正常工作，因此可以实现超低功耗。

10.2 主要特性

ESP32-S3 的低功耗管理有如下特性：

- 4 种预设功耗模式，可满足多种典型应用场景需求
- 高达 16 KB 保留内存 (retention memory)，含快速内存和慢速内存
- 8 个 32 位保留寄存器 (retention register)
- 支持 RTC Boot 功能，用于快速唤醒
- 各功耗模式下均支持超低功耗协处理器

在本章节中，我们将首先介绍 ESP32-S3 低功耗管理的工作过程，其次介绍芯片的预设低功耗工作模式，最后介绍芯片的 RTC Boot 过程。

10.3 功能描述

ESP32-S3 的低功耗管理主要由以下模块实现：

- 功耗管理单元 (PMU)：控制向模拟、RTC 和数字类电源域的供电。具体细分电源域列表，请见第 10.4.1 小节；
- 电源隔离单元：保证各电源域的独立工作，防止掉电电源域影响其他电源域的工作；
- 低功耗时钟：为低功耗模式下工作的电源域提供时钟信号；
- 定时器：
  - RTC 定时器：在专用寄存器中记录 RTC 主状态机的状态；
  - ULP 定时器：在预设时间唤醒超低功耗协处理器。更多详情，请见章节 2 *超低功耗协处理器 (ULP-FSM, ULP-RISC-V)*。

  - 触摸传感器定时器：在预设时间唤醒触摸传感器。更多详情，请见章节 39片上传感器与模拟信号处理。

- 8 个 32 位 “always-on” 保留寄存器：即这 8 个寄存器永远处于工作状态，不受 deep-sleep 等低功耗模式的影响，可用于存储一些不能丢失的数据。
- 22 个 “always-on” 管脚：即这 22 个管脚永远处于工作状态，不受 deep-sleep 等低功耗模式的影响，可用作低功耗模式下的唤醒源（详见第 10.4.4 节），也作为正常 GPIO 使用（详见 6 *IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)* 章节）。
- RTC 慢速内存：8 KB SRAM，工作时钟为 RTC 快速时钟 (rtc_fast_clk)，可用作扩展内存或存储 ULP 指令和数据内存。
- RTC 快速内存：8 KB SRAM，工作时钟与 CPU 时钟 (CPU_CLK) 同频，可用作扩展内存。
- 调压器：调节向不同电源域的供电电压。

ESP32-S3 低功耗管理的原理图可见图 10.3-1。

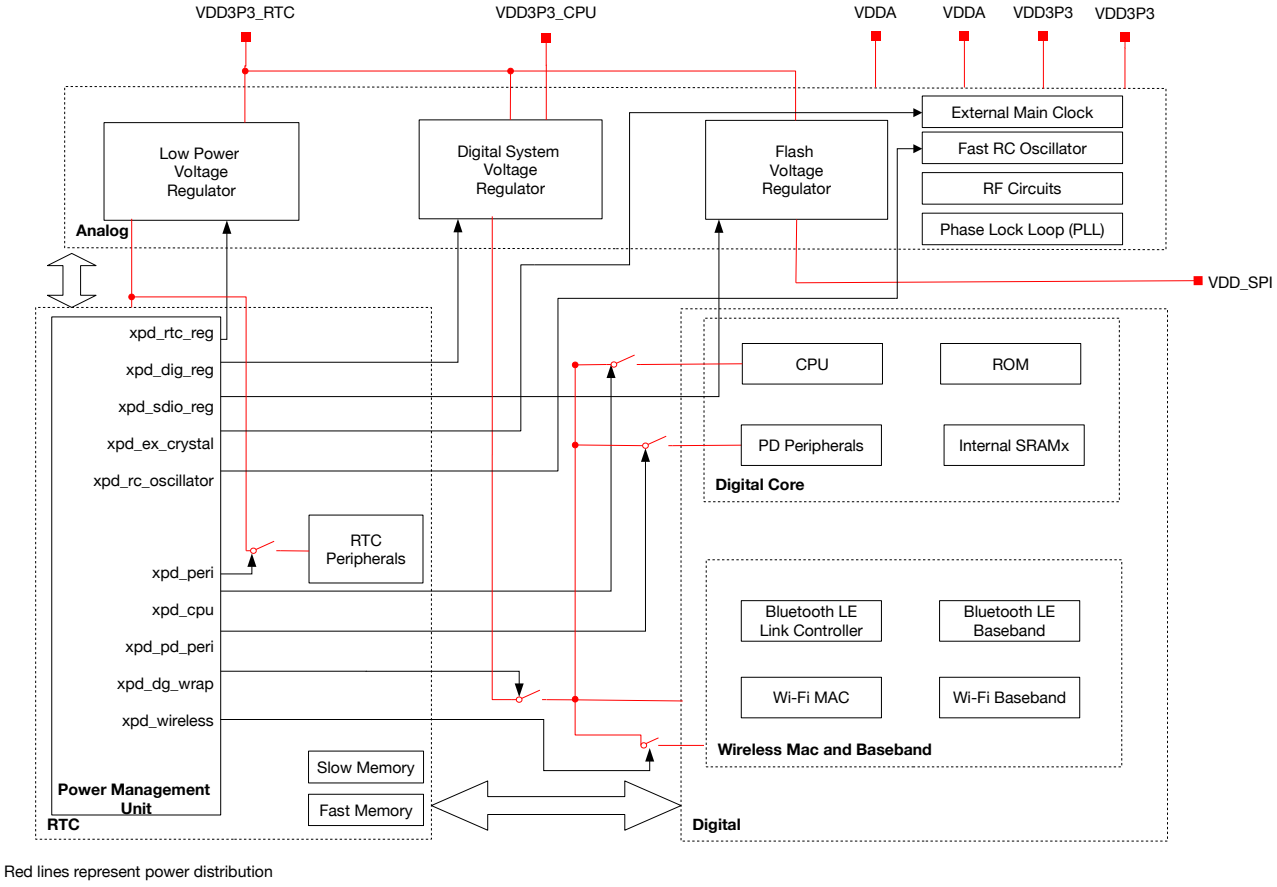


图 10.3-1. 低功耗管理原理图

说明：

- 有关各电源域的具体描述，请见第 10.4.1 节。
- 上图中的信号描述如下：
  - xpd_rtc_reg：
    * 当 RTC_CNTL_RTC_REGULATOR_FORCE_PU 置 1 时，低功耗调压器常开；

    * 否则，低功耗调压器在芯片进入 Light-sleep 和 Deep-sleep 时关闭。此时，RTC 电路由一个极低功耗的内置电源供电。
  - xpd_dig_reg：
    * 当 RTC_CNTL_DG_WRAP_PD_EN 使能时，数字调压器在芯片进入 Light-sleep 和 Deep-sleep 时关闭；
    * 否则，数字调压器常开。
  - xpd_peri：
    * 当 RTC_CNTL_RTC_PD_EN 使能时，RTC 外设在芯片进入 Light-sleep 和 Deep-sleep 时关闭；
    * 否则，RTC 外设常开。
  - xpd_cpu：
    * 当 RTC_CNTL_CPU_PD_EN 使能时，CPU 在芯片进入 Light-sleep 和 Deep-sleep 时关闭；
    * 否则 CPU 常开。
  - xpd_pd_peri：
    * 当 RTC_CNTL_DG_PERI_PD_EN 使能时，PD 外设在芯片进入 Light-sleep 和 Deep-sleep 时关闭；
    * 否则，PD 外设常开。
  - xpd_dg_wrap：该开关的控制与 xpd_dig_reg 保持一致
  - xpd_wireless：
    * 当 RTC_CNTL_WIFI_PD_EN 使能时，Wireless circuit 在芯片进入 Light-sleep 和 Deep-sleep 时关闭；
    * 否则 Wireless circuit 常开。
  - xpd_sdio_reg：见下方第 10.3.4.3 小节。
  - xpd_ex_crystal:
    * 当 RTC_CNTL_XTL_FORCE_PU 置 1 时，外部主晶振常开；
    * 否则，外部主晶振在芯片进入 Light-sleep 和 Deep-sleep 时关闭。
  - xpd_rc_oscilator:
    * 当 RTC_CNTL_CK8M_FORCE_PU 置 1 时，快速 RC 振荡器常开；
    * 否则，快速 RC 振荡器在芯片进入 Light-sleep 和 Deep-sleep 时关闭。
  - RF 电路和 PLL 由内部信号控制，不对用户开放。

10.3.1 功耗管理单元

ESP32-S3 功耗管理单元可以控制向不同电源域的供电，其主要组成部分包括：

- RTC 主状态机 (RTC Main State Machine)：产生电源门控、时钟门控和复位信号。
- 功耗控制器 (Power Controller)：根据 RTC 主状态机产生的电源门控信号，打开或关闭各电源域。
- 睡眠和唤醒控制器 (Sleep Controller, Wakeup Controller)：向 RTC 主状态机发送睡眠或唤醒请求。

- 时钟控制器 (Clock Controller)：选择并打开或关闭时钟源。
- 保护定时器 (Protection Timer)：控制主状态机切换状态的等待时间。

在 ESP32-S3 的电源管理单元中，睡眠和唤醒控制器向 RTC 主状态机发送睡眠或唤醒请求，RTC 主状态机接着产生电源门控、时钟门控和复位信号。此后，电源控制器和时钟控制器会根据 RTC 主状态机产生的信号，打开或关闭不同的电源域和时钟信号，从而让芯片进入或退出低功耗模式。电源管理单元的主要工作流程可见图 10.3-2。

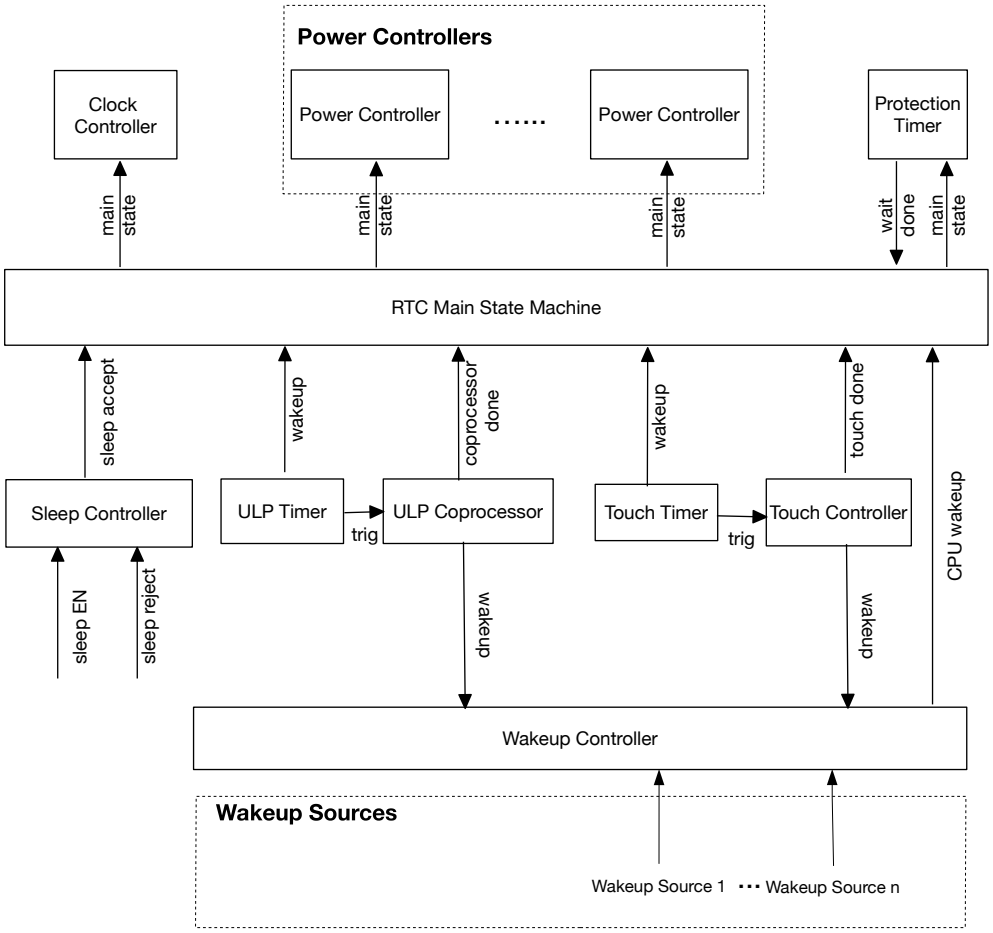


图 10.3-2. 电源管理单元的主要工作流程

说明：

1. 完整电源域列表，请见第 10.4.1 节。
2. 有关各唤醒源的具体描述，请见表 10.4-3。

10.3.2 低功耗时钟

通常情况下，当 ESP32-S3 处于低功耗模式下，芯片的外部晶振 XTAL_CLK 和 PLL 将被断电以降低功耗，但低功耗时钟仍保持开启，为芯片的不同电源域提供时钟，比如电源管理单元、RTC 外设等，以确保芯片在低功耗模式下的正常工作。

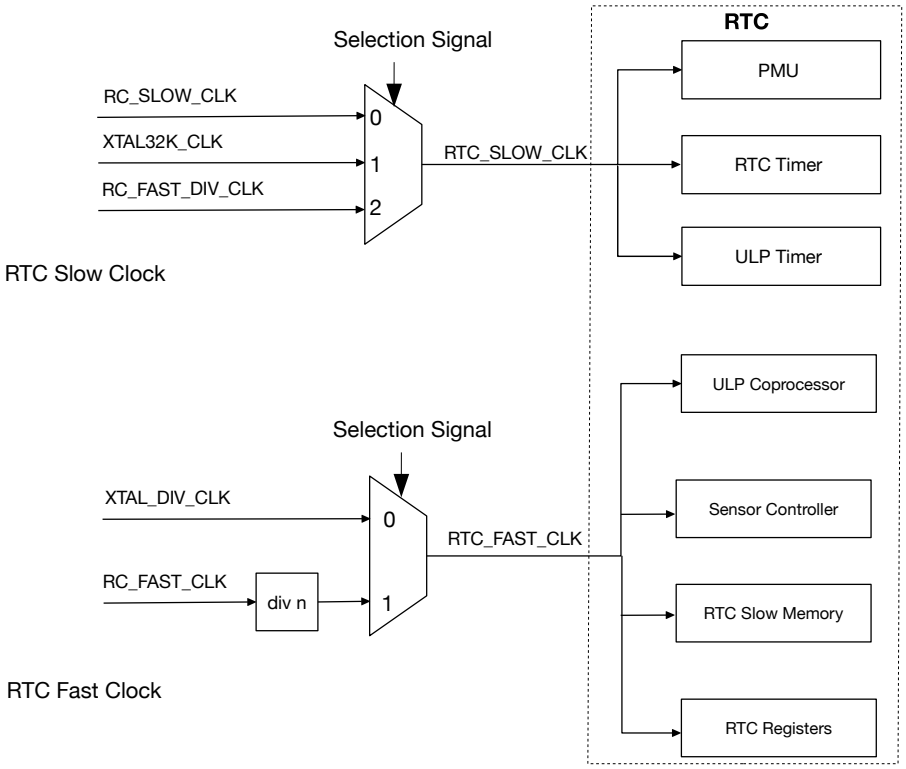


图 10.3-3. RTC 时钟

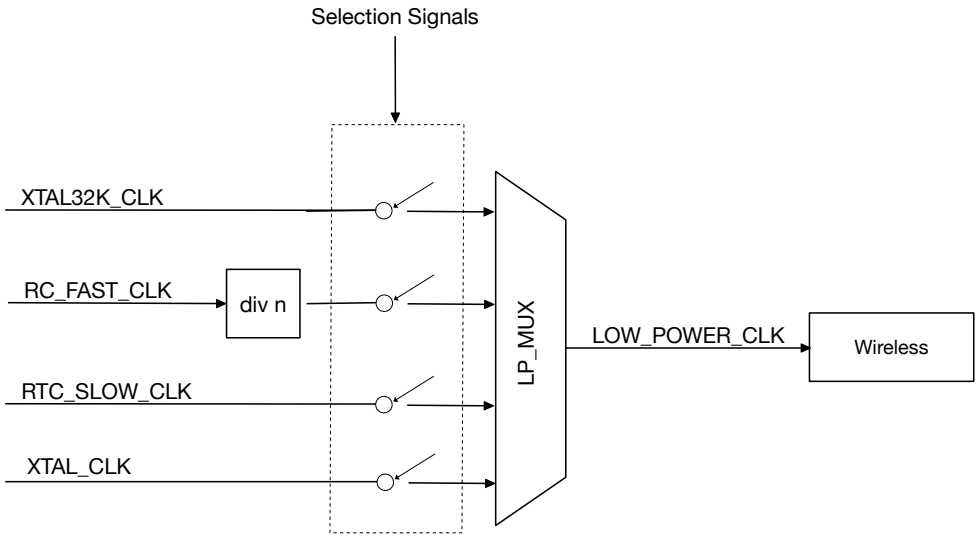


图 10.3-4. Wireless 时钟

表 10.3-1. 低功耗时钟

时钟类型	可选时钟源	时钟选择寄存器	作用范围
RTC 慢速时钟	XTAL32K_CLK	RTC_CNTL_ANA_CLK_RTC_SEL	功耗管理单元
	RC_FAST_DIV_CLK		RTC 定时器
	RC_SLOW_CLK（默认）		ULP 定时器
RTC 快速时钟	RC_FAST_CLK 的 n 分频（默认）	RTC_CNTL_FAST_CLK_RTC_SEL	ULP 协处理器
	XTAL_DIV_CLK		传感器控制器
			RTC 寄存器
			RTC 慢速内存
低功耗时钟	XTAL32K_CLK	SYSTEM_LPCLK_SEL_XTAL32K	低功耗模式下的数字中的无线通信模块 (Wi-Fi/BT)
	RC_FAST_CLK 的 n 分频	SYSTEM_LPCLK_SEL_8M	
	RTC_SLOW_CLK	SYSTEM_LPCLK_SEL_RTC_SLOW	
	XTAL_CLK	SYSTEM_LPCLK_SEL_XTAL	

更多有关时钟的描述，请见章节 7 复位和时钟。

10.3.3 定时器

ESP32-S3 的低功耗管理使用三个定时器：

- RTC 定时器
- ULP 定时器
- 触摸定时器

本章节主要介绍 RTC 定时器。有关 ULP 定时器的内容，请见章节 2 超低功耗协处理器 (ULP-FSM, ULP-RISC-V)。有关触摸定时器的内容，请见章节 39 片上传感器与模拟信号处理。

RTC 定时器是一个 48 位的可读计数器，可通过配置，使用 RTC 慢速时钟记录以下任一事件发生的时刻。更多详情，请见表 10.3-2。

表 10.3-2. RTC 定时器的触发条件

使能条件	触发条件
RTC_CNTL_TIMER_XTL_OFF	RTC 主状态机关闭或打开 XTAL_CLK 晶振时触发。
RTC_CNTL_TIMER_SYS_STALL	CPU 进入或退出 stall 状态时触发。该设置可保证 SYS_TIMER 的时间连续性。
RTC_CNTL_TIMER_SYS_RST	系统复位时触发。
RTC_REG_TIME_UPDATE	配置寄存器 RTC_CNTL_RTC_TIME_UPDATE 时触发。该触发由 CPU 产生（比如用户）。

RTC 定时器会在每次触发时更新两组寄存器。其中第一组寄存器记录本次触发的信息，第二组寄存器记录之前触发的信息。这两组寄存器的具体情况见下：

- 寄存器组 0 用于记录 RTC 定时器在当前触发下的计数值。
  - RTC_CNTL_RTC_TIME_HIGH0_REG
  - RTC_CNTL_RTC_TIME_LOW0_REG

- 寄存器组 1 用于记录 RTC 定时器在上一次触发下的计数值。
  - RTC_CNTL_RTC_TIME_HIGH1_REG
  - RTC_CNTL_RTC_TIME_LOW1_REG

每次有新的触发，上一次触发时的记录将从寄存器组 0 移至寄存器组 1（寄存器组 1 中之前的记录将被覆盖），而本次触发的记录将存储在寄存器组 0。因此，RTC 定时器最多可同时记录两次触发的值。

值得注意的是，除上电复位外的其余任何复位 / 睡眠均不会使 RTC 定时器停止或复位。

此外，RTC 定时器还能用作唤醒源。更多详情，请见第 10.4.4 节。

10.3.4 调压器

ESP32-S3 共有三个调压器，负责调节向不同电源域的供电：

- 数字调压器：负责数字类电源域；
- 低功耗调压器：负责 RTC 类电源域；
- flash 调压器：负责数字类和 RTC 类之外的电源域。

说明：
更多有关不同电源域的描述，请见第 10.4.1 节。

10.3.4.1 数字调压器

ESP32-S3 的内置数字调压器可以将外部电源电压（通常为 3.3 V）转换为 1.1 V，支持数字类电源域的正常工作，该寄存器主要 xpd_dig_reg 信号控制，详见 10.3-1。具体结构示意图可见下方图 10.3-5。

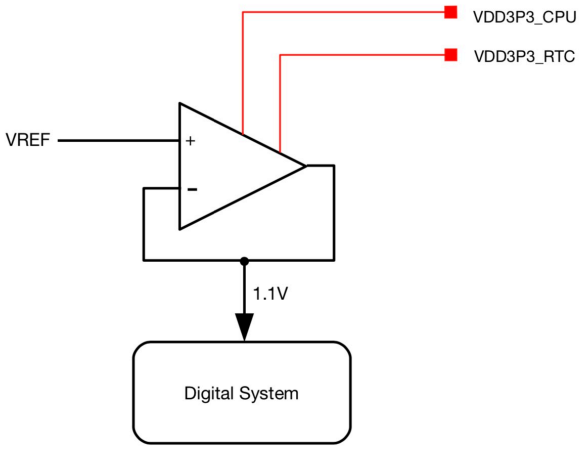


图 10.3-5. 数字调压器

10.3.4.2 低功耗调压器

ESP32-S3 的内置低功耗调压器可以将外部电源电压（通常为 3.3 V）转换为 1.1 V，支持 RTC 类电源域的正常工作。当管脚 CHIP_PU 为高电平时，RTC 电路不会掉电。低功耗调压器在芯片进入 Light-sleep 和 Deep-sleep

时可以选择关闭。此时，RTC 电路由一个极低功耗的内置电源供电（该电源无法关闭）。具体结构示意图可见下方图 10.3-1。

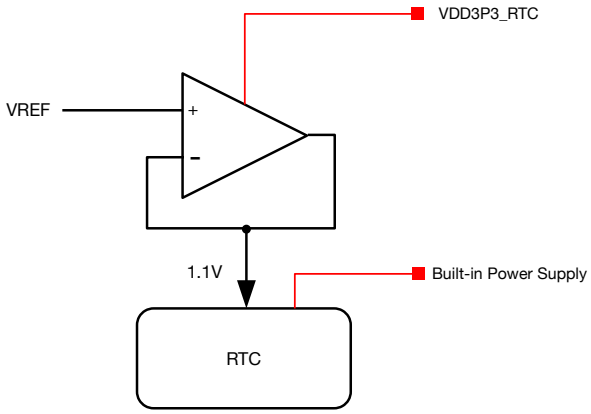


图 10.3-6. 低功耗调压器

10.3.4.3 flash 调压器

ESP32-S3 的内置 flash 调压器可以向数字类和 RTC 类之外的电源域（比如 flash）输出 3.3 V 或 1.8 V 电压，比如 flash，具体结构示意图可见下方图 10.3-7。

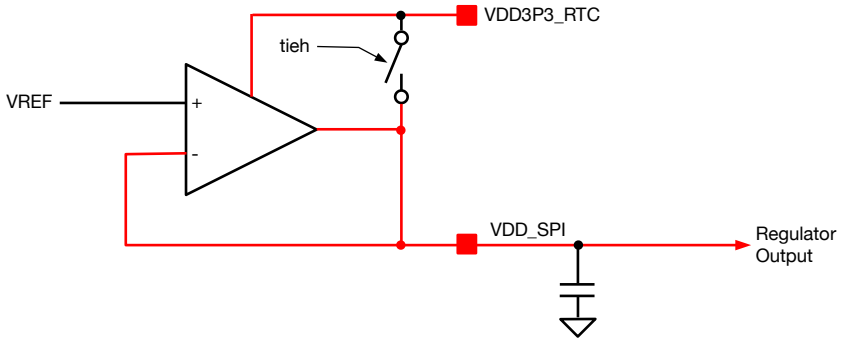


图 10.3-7. flash 调压器

1. 配置 XPD_SDIO_REG 选择调压器供电或外部电源供电：
   - 1：由调压器的输出电压供电
   - 0：由外部电源供电
2. 配置 SDIO_TIEH 选择 3.3 V 或 1.8 V 电压
   - 1：调压器将管脚 VDD_SPI 和管脚 VDD3P3_RTC 短路，电压输出为 3.3 V，即管脚 VDD3P3_RTC 的电压。
   - 0：调压器的电压输出为参考电压 VREF，通常为 1.8 V。

上述信号的具体配置如下：

- XPD_SDIO_REG 的配置：

  - 当芯片处于 Active 状态，且RTC_CNTL_SDIO_FORCE == 0，EFUSE_VDD_SPI_FORCE == 1 时，XPD_SDIO_REG 由 EFUSE_VDD_SPI_XPD 决定；
  - 当芯片处于 sleep 状态，且RTC_CNTL_SDIO_REG_PD_EN == 1 时，XPD_SDIO_REG 为 0；
  - 当 RTC_CNTL_SDIO_FORCE == 1 时，XPD_SDIO_REG 由 RTC_CNTL_XPD_SDIO_REG 决定。

- SDIO_TIEH 的配置：
  - 当 RTC_CNTL_SDIO_FORCE == 0 且 EFUSE_VDD_SPI_FORCE == 1 时，SDIO_TIEH = EFUSE_VDD_SPI_TIEH ；
  - 否则，SDIO_TIEH = RTC_CNTL_SDIO_TIEH。

10.3.4.4 欠压检测器

ESP32-S3 的欠压检测器可以检查管脚 VDD3P3，VDD3P3_RTC 和 VDD3P3_CPU 的电压，在电压快速下落至预设阈值（默认为 2.7 V）以下时发出触发信号，并进行相应处理，从而关闭部分耗电模块（比如 LNA 和 PA 等），为数字模块争取更多时间，用以保存、转移重要数据。

欠压检测器的功耗非常低，在芯片开启时将永远保持开启。ESP32-S3 欠压检测器的具体结构示意图可见下方图 10.3-8。

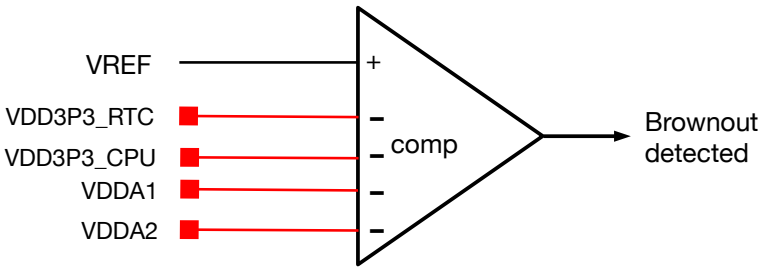


图 10.3-8. 欠压检测器

RTC_CNTL_RTC_BROWN_OUT_DET 可用于指示欠压检测器的输出电平，默认为低电平，可在检测管脚电压下降至阈值以下时跳至高电平。

欠压检测器检测到欠压信号后，有两种处理方法：

- mode0：当欠压计数器达到 int comparer 和 rst comparer 中设定的阈值后触发中断，并根据 rst_sel 的配置选择复位方式。此模式需通过 bod_mode0_en 使能。
- mode1：直接触发系统复位。

详情如下图所示。

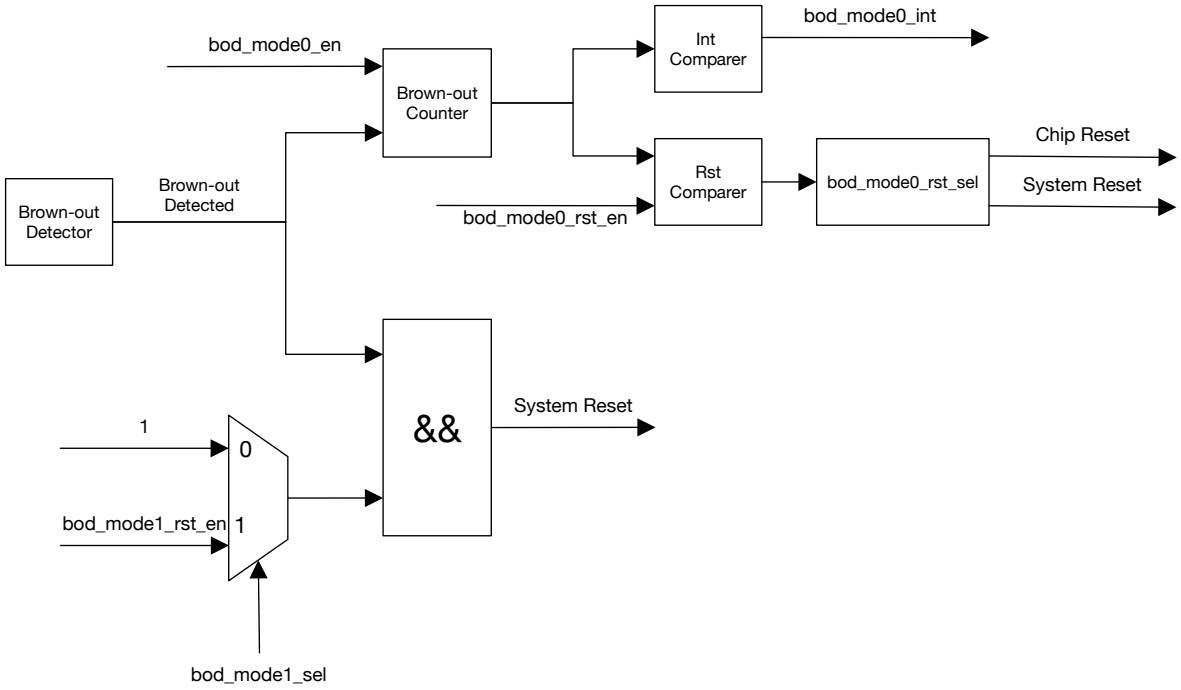


图 10.3-9. 欠压检测器处理方式

图中的选择信号均由寄存器控制，对应关系如下：

- bod_mode0_en：RTC_CNTL_BROWN_OUT_ENA
- bod_mode0_rst_en：RTC_CNTL_BROWN_OUT_RST_ENA
- bod_mode0_en: RTC_CNTL_BROWN_OUT_ENA
- bod_mode0_rst_sel：RTC_CNTL_BROWN_OUT_RST_SEL 可用于选择欠压检测器被触发后的复位方式。
  - 0：芯片复位
  - 1：系统复位
- bod_mode1_sel：RTC_CNTL_RTC_FIB_SEL 的第一位。
- bod_mode1_rst_en：RTC_CNTL_BROWN_OUT_ANA_RST_EN

10.4 功耗模式管理

10.4.1 电源域

ESP32-S3 共有三大类共 10 个电源域：

- RTC 类
  - 功耗管理单元
  - RTC 外设，包括 RTC GPIO、RTC I2C、温度传感器、触摸传感器、RTC ADC 控制器、ULP 协处理器
- 数字类
  - 数字内核

  - 无线通信 MAC 和 BB
  - CPU
  - 部分数字外设，包括 SPI2、GDMA、SHA、RSA、AES、HMAC、DS、Secure Boot、SDIO HOST、USB OTG 等

- 模拟类
  - 快速 RC 振荡器 (RC_FAST_CLK)
  - 外部晶振 (XTAL_CLK)
  - 锁相环 (PLL)
  - RF 电路

10.4.2 RTC 状态

ESP32-S3 共有活跃 (Active)、监测 (Monitor) 和睡眠 (Sleep) 三个主要 RTC 状态，其相互转换过程可见图 10.4-1。

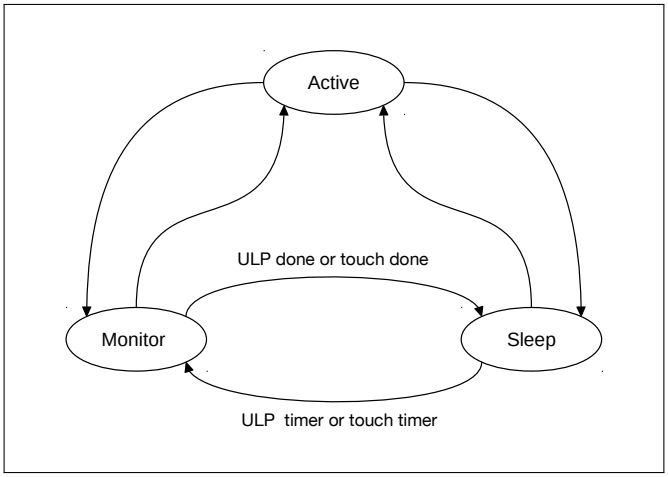


图 10.4-1. RTC 状态

在不同 RTC 状态下，各电源域的默认开关状态不同，但值得注意的是，用户还可以根据具体需求，强制打开 (FPU) 或关闭 (FPD) 特定电源域。具体可见表 10.4-1。

表 10.4-1. RTC 状态转换

电源域		RTC 状态		
类型	子类型	Active	Monitor	Sleep
RTC[1]， 2	电源管理单元[3]	ON	ON	ON
	RTC 外设[4]	ON	ON	OFF
数字类	CPU[5]	ON	OFF*	OFF*
	无线数字电路[6]	ON	OFF*	OFF*
	数字内核 [7]	ON	OFF*	OFF*
	部分数字外设	ON	OFF*	OFF*
模拟类	RC_FAST_CLK	ON	ON	OFF
	XTAL_CLK	ON	OFF	OFF
	PLL	ON	OFF	OFF
	RF 电路	-	-	-

* 可配置。

[1] RTC 慢速内存支持 8 KB SRAM，可用作保留内存或存储 ULP 协处理器指令和数据内存，因此应强制打开。CPU 可通过 PIF 总线访问慢速内存，起始地址为 0x5000_0000。RTC 慢速内存在 Monitor 状态下通常处于 OFF 状态，但这里有一个例外情况，即当 ULP 协处理器工作时，RTC 慢速内存还是处于 ON 的状态。

[2] RTC 快速内存支持 8 KB SRAM，可用作保留内存，因此应强制打开。CPU 可通过 IRAM0/DRAM0 访问快速内存。

[3] ESP32-S3 的电源管理单元经过专门设计，一旦芯片上电即处于 “always-on”（常开）状态，因此无法 FPU 和 FPD。

[4] RTC 外设包括 *2 超低功耗协处理器 (ULP-FSM, ULP-RISC-V)* 和 *片上传感器与模拟信号处理*（比如温度传感器控制器、触摸传感器和 SAR ADC 控制器）。

[5] CPU 在 light-sleep 下也可以单独掉电，但需要 retention DMA 保持恢复 CPU 状态。

[6] 无线通信模块电源域包括 Wi-Fi/BT 各自的 MAC 和 BB（基带）。

[7] 当数字内核关闭时，所有数字类电源域均关闭。值得注意的是，ESP32-S3 的 ROM 与 SRAM 也属于数字内核电源域。因此，当数字内核关闭后，用户不可以单独强制打开 ROM 或 SRAM。

10.4.3 预设功耗模式

如上文所示，ESP32-S3 定义了四种最常见的电源域设置组合，对应 4 种预设功耗模式，可满足用户的常见场景需求，详见表 10.4-2。

表 10.4-2. 预设功耗模式

功耗模式	电源域									
	PMU	RTC 外设	数字内核	CPU	部分数字外设	无线数字电路	RC_FAST_CLK	XTAL_CLK	PLL	RF 电路
Active	ON	ON	ON	ON	ON	ON	ON	ON	ON	ON
Modem-sleep	ON	ON	ON	ON	ON	ON*	ON	ON	ON	OFF
Light-sleep	ON	ON	ON	OFF*	ON*	OFF*	OFF	OFF	OFF	OFF
Deep-sleep	ON	ON*	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF

* 可配置

默认情况下，ESP32-S3 系统复位后将进入 Modem-sleep 模式，当有收发包任务时，可配置为 Active 模式。当 CPU 停止工作一段时间后，可以进入 Modem-sleep、Light-sleep 和 Deep-sleep 等低功耗模式。从 Active 到 Deep-sleep 模式，性能递减 [1]、功耗递减[2]、唤醒延迟时间递增。此外，这些模式可支持的唤醒源 [3] 不同。用户根据具体需求，从性能要求、功耗高低、唤醒延迟及可用唤醒源等方面考虑，选择合适的功耗模式。

说明：

1. 更多详情，请见表 10.4-2。
2. 具体功耗数据可见《ESP32-S3 技术规格书》中的功耗特性章节。
3. 具体可支持的唤醒源，请见第 10.4.4 节。

10.4.4 唤醒源

ESP32-S3 可支持多种唤醒源将 CPU 从不同睡眠模式中唤醒。唤醒源的选择由 RTC_CNTL_RTC_WAKEUP_ENA 决定，见表 10.4-3。

表 10.4-3. 唤醒源

WAKEUP_ENA	唤醒源[10]	Light-sleep	Deep-sleep	说明
0x1	EXT0	Y	Y	1
0x2	EXT1	Y	Y	2
0x4	GPIO	Y	Y	3
0x8	RTC 定时器	Y	Y	-
0x20	Wi-Fi	Y	-	4
0x40	UART0	Y	-	5
0x80	UART1	Y	-	5
0x100	TOUCH Active	Y	Y	6
0x200	ULP-FSM	Y	Y	7
0x400	BT	Y	-	4
0x800	ULP-RISC-V	Y	Y	
0x1000	XTAL_32K	Y	Y	8
0x2000	ULP-RISC-V Trap	Y	Y	9
0x8000	TOUCH Timeout	Y	Y	-
0xc000	BROWNOUT	Y	Y	-

说明：

1. EXT0 仅可将芯片从 Light-sleep/Deep-sleep 模式中唤醒。当 RTC_CNTL_EXT_WAKEUP0_LV 为 1 时将触发管脚高电平，否则触发管脚低电平。用户可通过设置 RTCIO_EXT_WAKEUP0_SEL 选择作为唤醒源的 RTC 管脚。
2. EXT1 经过专门设计，可将芯片从任何睡眠模式中唤醒，而且还支持多个管脚的组合。首先，应按照选定作为唤醒源的管脚位图，配置 RTC_CNTL_EXT_WAKEUP1_SEL[17:0]。当 RTC_CNTL_EXT_WAKEUP1_LV == 1 时，任何选中管脚之一为高电压则将触发信号唤醒芯片；当 RTC_CNTL_REG_EXT_WAKEUP1_LV == 0 时，任何选中管脚为低电压才能触发信号唤醒芯片。**注意：**用户应确保 EXT1 信号的 hold 时长至少三个 RTC_SLOW_CLK 周期，否则信号状态无法被 RTC_CNTL_EXT_WAKEUP1_STATUS 记录。
3. 在 Deep-sleep 模式下，该唤醒源仅有 RTC GPIO 可以作为唤醒源。

4. 为了通过 Wi-Fi 或 BT 唤醒芯片，芯片将在 Active、Modem-sleep 和 Light-sleep 之间进行切换，CPU 和 RF 模块均将在预设间隔中唤醒，保证 Wi-Fi 和 BT 的正常连接和数据通信。
5. 当接收到的 RX 脉冲数量超过阈值寄存器中的设置时，即触发唤醒。
6. 当触摸传感器检测到触摸时，即触发唤醒。
7. 当超低功耗协处理器配置寄存器 RTC_CNTL_RTC_SW_CPU_INT，即触发唤醒。
8. 32 kHz 晶振作为 RTC 慢速时钟时，当 32 kHz 看门狗定时器检测到任何时钟停振，即触发唤醒。
9. 当超低功耗协处理器进入捕获异常事件时（比如堆栈溢出），即触发唤醒。
10. 表格中的所有唤醒源均可配置为拒绝睡眠原因，但 UART 除外。

10.4.5 拒绝睡眠

ESP32-S3 提供了硬件拒绝睡眠的机制，防止系统在某些外设仍在工作但是未被 CPU 检测到时进入睡眠，最终导致该外设不能正常工作。

ESP32-S3 的所有唤醒源也同时可以作为芯片拒绝睡眠的原因，因此具体的拒绝睡眠源请见表 10.4-3（UART 除外）。

用户可以配置以下寄存器使能或禁用拒绝睡眠功能。

- 配置 RTC_CNTL_RTC_SLEEP_REJECT_ENA 整体使能或关闭拒绝休眠功能：
  - 进一步配置 RTC_CNTL_LIGHT_SLP_REJECT_EN，具体使能拒绝进入 light-sleep；
  - 进一步配置 RTC_CNTL_DEEP_SLP_REJECT_EN，具体使能拒绝进入 deep-sleep；
- 读取 RTC_CNTL_SLP_REJECT_CAUSE_REG 了解拒绝睡眠的原因。

10.5 Retention DMA

ESP32-S3 可以选择将 CPU 掉电，进一步降低 light-sleep 模式下的功耗。为了解决 CPU 掉电之后从 light-sleep 唤醒时，程序无法接着从进入睡眠之前的断点运行，ESP32-S3 专门新增了 retention 模块。ESP32-S3 的 retention 模块可以在芯片进入 light-sleep 前通过 retention DMA 将 CPU 中的信息备份到 Internal SRAM 的第 2~8 块的任意地址中，然后在退出 light-sleep 后将这些信息从 Internal SRAM 中恢复至 CPU，从而保证 CPU 可以接着睡眠之前的断点运行。ESP32-S3 中 retention DMA 的具体特性如下：

- Retention DMA 的传输位宽为 128 位，并且只支持 4 字 (Word) 对齐的地址访问。
- Retention DMA 的链表经过专门设计，读写操作均共用同一个链表，与通用 DMA 链表的配置方法保持一致：
  1. 用户需要在进入睡眠前申请一定的 SRAM 内存空间 *，用于存放 CPU 的寄存器信息（428 个字）和配置信息（4 个字），共 432 个字；
  2. 根据内存的申请情况，填写链表的地址和长度信息。具体可参考章节 3 *通用 DMA 控制器 (GDMA)*。

说明：

* 如果申请的 SRAM 空间不够 432 个字，芯片仅能进入普通的 light-sleep，无法进一步关闭 CPU。

此后，用户即可使能寄存器 RTC_CNTL_RETENTION_CTRL_REG 中的 RTC_CNTL_RETENTION_EN 域，开启 retention 功能，即：

- 在系统进入睡眠前，自动启动保留 DMA 进行数据备份；
- 在系统唤醒后但未恢复前，自动启动保留 DMA 进行数据还原。

10.6 RTC Boot

在 Deep-sleep 下，芯片的 ROM 和 RAM 均将断电，因此在唤醒时 SPI 启动（从 flash 复制数据）所需时间更长。因此，相较于 Light-sleep 和 Modem-sleep 模式，Deep-sleep 模式的唤醒时间要长的多。不过，值得注意的是，在 Deep-sleep 模式下，RTC 快速内存和慢速内存均可以处于上电状态。因此，用户可以将一些代码规模不大（即小于 8 KB 的 “deep sleep wake stub”）写入 RTC 快速内存或慢速内存，避免 SPI 启动带来的延迟，从而加速芯片唤醒过程。

第一种方法：RTC 慢速内存

1. 设置寄存器 RTC_CNTL_PROCPU_STAT_VECTOR_SEL 为 0。
2. 芯片进入睡眠模式。
3. 当 CPU 开启时，复位向量将从地址 0x50000000，而非 0x40000400 开始复位，整个过程并不需要进行 SPI 启动。RTC 内存中的代码仅需在 C 语言环境中进行部分初始化操作即可。

第二种方法：RTC 快速内存

1. 设置寄存器 RTC_CNTL_PROCPU_STAT_VECTOR_SEL 为 1。
2. 计算 RTC 快速内存的 CRC 码，并将结果保存在寄存器 RTC_CNTL_RTC_STORE7_REG[31:0] 中。
3. 设置寄存器 RTC_CNTL_RTC_STORE6_REG[31:0] 为 RTC 快速内存的入口地址。
4. 芯片进入睡眠模式。
5. 当 CPU 开启时，开始进行 ROM 解包和部分初始化工作。此后，再次计算 RTC 快速内存的 CRC 码。如果与寄存器 RTC_CNTL_RTC_STORE7_REG[31:0] 中保存的结果一致，则 CPU 跳转至 RTC 快速内存的入口地址。

ESP32-S3 的 RTC 启动流程见下方图 10.6-1：

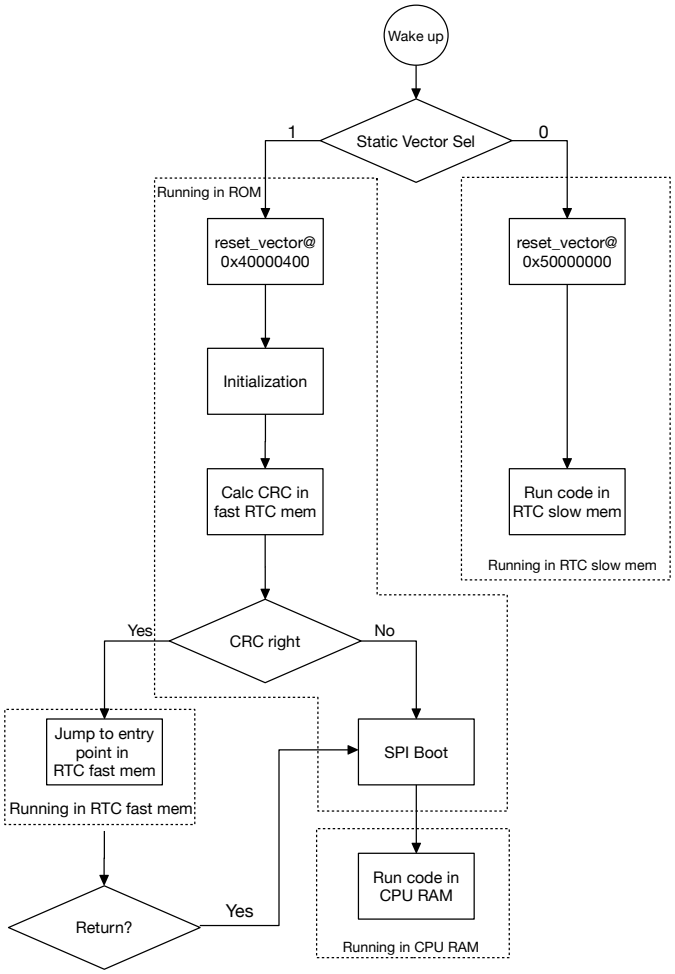


图 10.6-1. ESP32-S3 启动流程图

10.7 寄存器列表

本小节的所有地址均为相对于低功耗管理基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

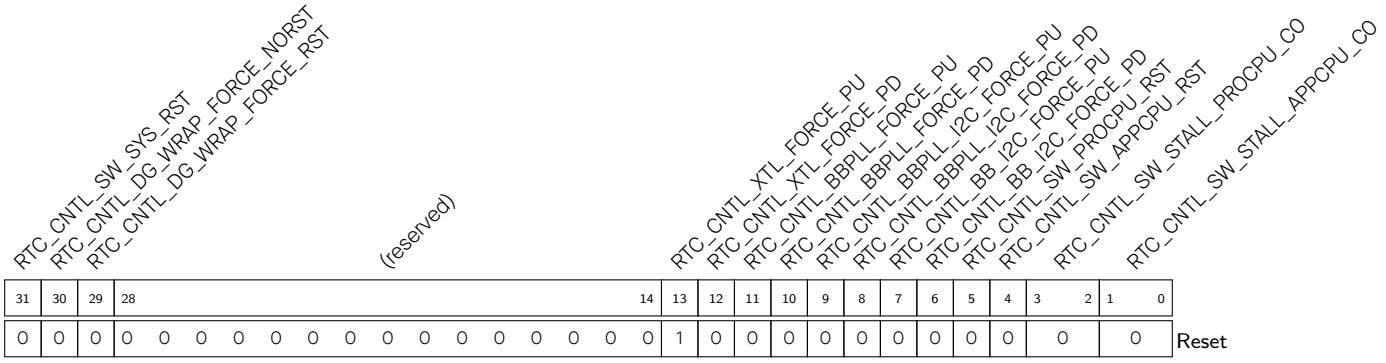
名称	描述	地址	访问
控制/配置寄存器			
RTC_CNTL_RTC_OPTIONS0_REG	设置晶振和 PLL 时钟的电源选项，并启动软件复位	0x0000	varies
RTC_CNTL_RTC_SLP_TIMER0_REG	RTC 定时器阈值寄存器 0	0x0004	R/W
RTC_CNTL_RTC_SLP_TIMER1_REG	RTC 定时器阈值寄存器 1	0x0008	varies
RTC_CNTL_RTC_TIME_UPDATE_REG	RTC 定时器更新控制寄存器	0x000C	varies
RTC_CNTL_RTC_STATE0_REG	配置 sleep/reject/wakeup 状态	0x0018	varies
RTC_CNTL_RTC_TIMER1_REG	配置 CPU stall 选项	0x001C	R/W
RTC_CNTL_RTC_TIMER2_REG	配置 RTC 慢速时钟	0x0020	R/W
RTC_CNTL_RTC_TIMER5_REG	配置最小睡眠周期	0x002C	R/W
RTC_CNTL_RTC_ANA_CONF_REG	配置 I2C 和 PLLA 的供电	0x0034	R/W
RTC_CNTL_RTC_WAKEUP_STATE_REG	唤醒位图使能寄存器	0x003C	R/W
RTC_CNTL_RTC_EXT_XTL_CONF_REG	32 kHz 晶振配置寄存器	0x0060	varies
RTC_CNTL_RTC_EXT_WAKEUP_CONF_REG	GPIO 唤醒配置寄存器	0x0064	R/W
RTC_CNTL_RTC_SLP_REJECT_CONF_REG	睡眠/拒绝选项配置寄存器	0x0068	R/W
RTC_CNTL_RTC_CLK_CONF_REG	RTC 时钟配置寄存器	0x0074	R/W
RTC_CNTL_RTC_SLOW_CLK_CONF_REG	RTC 慢速时钟配置寄存器	0x0078	R/W
RTC_CNTL_RTC_SDIO_CONF_REG	配置 flash 上电	0x007C	varies
RTC_CNTL_RTC_REG	RTC/DIG 调压器配置寄存器	0x0084	R/W
RTC_CNTL_RTC_PWC_REG	RTC 电源配置寄存器	0x0088	R/W
RTC_CNTL_DIG_PWC_REG	数字系统电源配置寄存器	0x0090	R/W
RTC_CNTL_DIG_ISO_REG	数字系统 ISO 配置寄存器	0x0094	varies
RTC_CNTL_RTC_WDTCONFIG0_REG	RTC 看门狗配置寄存器	0x0098	R/W
RTC_CNTL_RTC_WDTCONFIG1_REG	配置 1 级 RTC 看门狗的保持时间	0x009C	R/W
RTC_CNTL_RTC_WDTCONFIG2_REG	配置 2 级 RTC 看门狗的保持时间	0x00A0	R/W
RTC_CNTL_RTC_WDTCONFIG3_REG	配置 3 级 RTC 看门狗的保持时间	0x00A4	R/W
RTC_CNTL_RTC_WDTCONFIG4_REG	配置 4 级 RTC 看门狗的保持时间	0x00A8	R/W
RTC_CNTL_RTC_WDTFEED_REG	RTC 看门狗软件喂狗配置寄存器	0x00AC	WO
RTC_CNTL_RTC_WDTWPROTECT_REG	RTC 看门狗写保护配置寄存器	0x00B0	R/W
RTC_CNTL_RTC_SWD_CONF_REG	超级看门狗配置寄存器	0x00B4	varies
RTC_CNTL_RTC_SWD_WPROTECT_REG	超级看门狗写保护配置寄存器	0x00B8	R/W
RTC_CNTL_RTC_SW_CPU_STALL_REG	CPU stall 配置寄存器	0x00BC	R/W
RTC_CNTL_RTC_PAD_HOLD_REG	配置 RTC GPIO 的保持时间	0x00D8	R/W
RTC_CNTL_DIG_PAD_HOLD_REG	配置数字 GPIO 的保持时间	0x00DC	R/W
RTC_CNTL_RTC_EXT_WAKEUP1_REG	EXT1 唤醒配置寄存器	0x00E0	varies
RTC_CNTL_RTC_EXT_WAKEUP1_STATUS_REG	EXT1 唤醒配置寄存器	0x00E4	RO
RTC_CNTL_RTC_BROWN_OUT_REG	欠压检测配置寄存器	0x00E8	varies

名称	描述	地址	访问
RTC_CNTL_RTC_XTAL32K_CLK_FACTOR_REG	配置 32 kHz 晶振备用时钟的分频数	0x00F4	R/W
RTC_CNTL_RTC_XTAL32K_CONF_REG	32 kHz 晶振配置寄存器	0x00F8	R/W
RTC_CNTL_RTC_USB_CONF_REG	USB 配置寄存器	0x0120	R/W
RTC_CNTL_RTC_OPTION1_REG	RTC 选项寄存器	0x012C	R/W
RTC_CNTL_RETENTION_CTRL_REG	保留配置寄存器	0x0140	R/W
RTC_CNTL_RTC_FIB_SEL_REG	欠压检测配置寄存器	0x0148	R/W
状态寄存器			
RTC_CNTL_RTC_TIME_LOW0_REG	存储 RTC 定时器 0 的低 32 位	0x0010	RO
RTC_CNTL_RTC_TIME_HIGH0_REG	存储 RTC 定时器 0 的高 16 位	0x0014	RO
RTC_CNTL_RTC_RESET_STATE_REG	指示 CPU 复位的来源	0x0038	varies
RTC_CNTL_RTC_STORE0_REG	Retention 寄存器 1	0x0050	R/W
RTC_CNTL_RTC_STORE1_REG	Retention 寄存器 2	0x0054	R/W
RTC_CNTL_RTC_STORE2_REG	Retention 寄存器 3	0x0058	R/W
RTC_CNTL_RTC_STORE3_REG	Retention 寄存器 4	0x005C	R/W
RTC_CNTL_RTC_STORE4_REG	Retention 寄存器 4	0x00C0	R/W
RTC_CNTL_RTC_STORE5_REG	Retention 寄存器 5	0x00C4	R/W
RTC_CNTL_RTC_STORE6_REG	Retention 寄存器 6	0x00C8	R/W
RTC_CNTL_RTC_STORE7_REG	Retention 寄存器 7	0x00CC	R/W
RTC_CNTL_RTC_LOW_POWER_ST_REG	指示 RTC 可随时被任何唤醒源唤醒	0x00D0	RO
RTC_CNTL_RTC_TIME_LOW1_REG	存储 RTC 定时器 1 的低 32 位	0x00EC	RO
RTC_CNTL_RTC_TIME_HIGH1_REG	存储 RTC 定时器 1 的高 16 位	0x00F0	RO
RTC_CNTL_RTC_SLP_WAKEUP_CAUSE_REG	存储唤醒原因寄存器	0x0130	RO
RTC_CNTL_RTC_SLP_REJECT_CAUSE_REG	存储拒绝入睡原因寄存器	0x0128	RO
中断寄存器			
RTC_CNTL_INT_ENA_RTC_REG	RTC 中断使能寄存器	0x0040	R/W
RTC_CNTL_INT_RAW_RTC_REG	RTC 原始中断寄存器	0x0044	varies
RTC_CNTL_INT_ST_RTC_REG	RTC 中断状态寄存器	0x0048	RO
RTC_CNTL_INT_CLR_RTC_REG	RTC 中断清除寄存器	0x004C	WO
RTC_CNTL_INT_ENA_RTC_W1TS_REG	RTC 中断使能寄存器 (W1TS)	0x0138	WO
RTC_CNTL_INT_ENA_RTC_W1TC_REG	RTC 中断清除寄存器 (W1TC)	0x013C	WO

10.8 寄存器

本小节的所有地址均为相对于低功耗管理基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 10.1. RTC_CNTL_RTC_OPTIONS0_REG (0x0000)



RTC_CNTL_SW_STALL_APPCPU_C0 当 RTC_CNTL_SW_STALL_APPCPU_C1 配置为 0x21 时，设置该位为 0x2 通过软件使 CPU1 进入 stall 状态。(R/W)

RTC_CNTL_SW_STALL_PROCPU_C0 当 RTC_CNTL_SW_STALL_PROCPU_C1 配置为 0x21 时，设置该位为 0x2 通过软件使 CPU0 进入 stall 状态。(R/W)

RTC_CNTL_SW_APPCPU_RST 置 1 软件复位 CPU1。(WO)

RTC_CNTL_SW_PROCPU_RST 置 1 软件复位 CPU0。(WO)

RTC_CNTL_BB_I2C_FORCE_PD 置 1 强制关闭 BB_I2C。(R/W)

RTC_CNTL_BB_I2C_FORCE_PU 置 1 强制打开 BB_I2C。(R/W)

RTC_CNTL_BBPLL_I2C_FORCE_PD 置 1 强制关闭 BB_PLL_I2C。(R/W)

RTC_CNTL_BBPLL_I2C_FORCE_PU 置 1 强制打开 BB_PLL_I2C。(R/W)

RTC_CNTL_BBPLL_FORCE_PD 置 1 强制关闭 BB_PLL。(R/W)

RTC_CNTL_BBPLL_FORCE_PU 置 1 强制打开 BB_PLL。(R/W)

接下页...

Register 10.1. RTC_CNTL_RTC_OPTIONS0_REG (0x0000)

接上页...

RTC_CNTL_XTL_FORCE_PD 置 1 强制关闭晶振。(R/W)

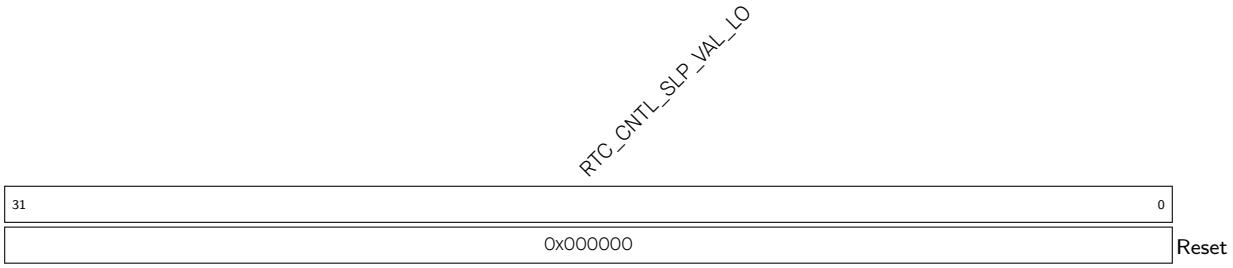
RTC_CNTL_XTL_FORCE_PU 置 1 强制打开晶振。(R/W)

RTC_CNTL_DG_WRAP_FORCE_RST 置 1 强制 deep-sleep 中的数字系统复位。(R/W)

RTC_CNTL_DG_WRAP_FORCE_NORST 置 1 强制 deep-sleep 中的数字系统不复位。(R/W)

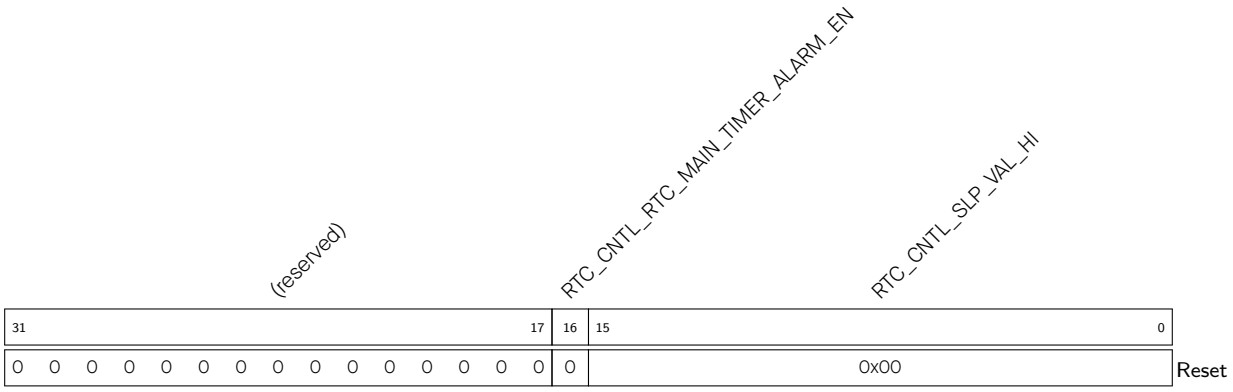
RTC_CNTL_SW_SYS_RST 置 1 通过软件复位系统。(WO)

Register 10.2. RTC_CNTL_RTC_SLP_TIMER0_REG (0x0004)



RTC_CNTL_SLP_VAL_LO 配置 RTC 计时器触发阈值的低 32 位。(R/W)

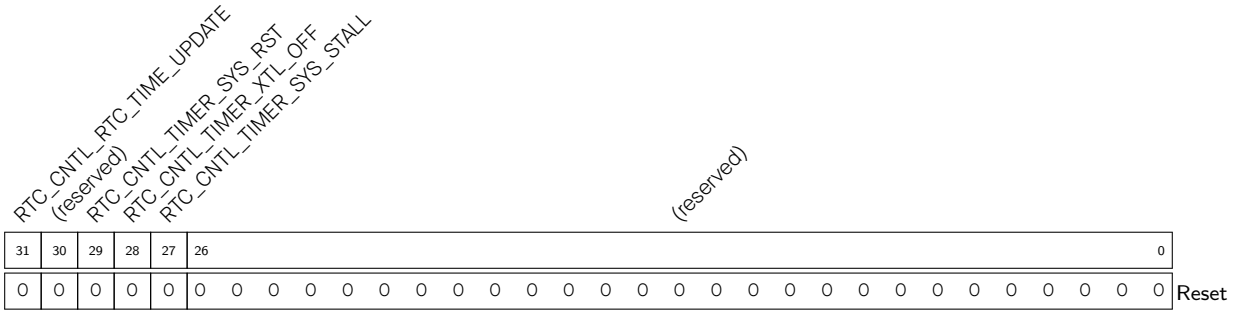
Register 10.3. RTC_CNTL_RTC_SLP_TIMER1_REG (0x0008)



RTC_CNTL_SLP_VAL_HI 配置 RTC 计时器触发阈值的高 16 位。(R/W)

RTC_CNTL_RTC_MAIN_TIMER_ALARM_EN 置 1 使能定时器警报。(WO)

Register 10.4. RTC_CNTL_RTC_TIME_UPDATE_REG (0x000C)



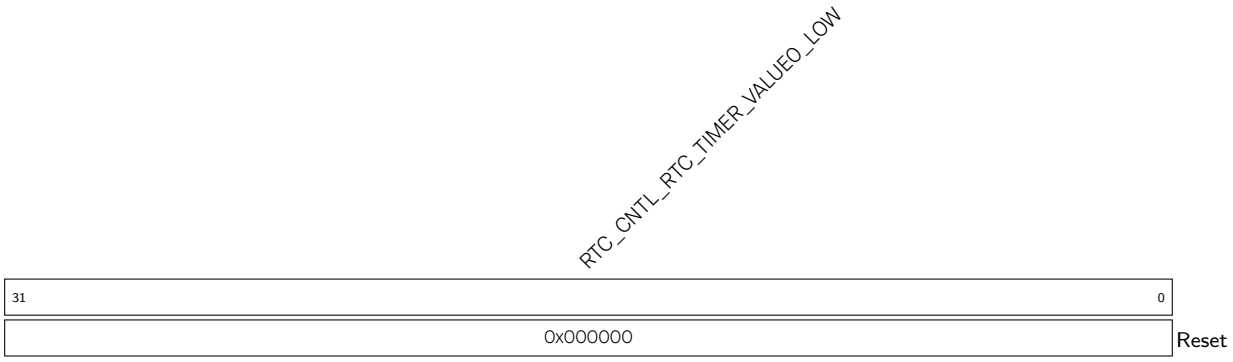
RTC_CNTL_TIMER_SYS_STALL 选择 RTC 计时器的触发条件。详见表 10.3-2。(R/W)

RTC_CNTL_TIMER_XTL_OFF 选择 RTC 计时器的触发条件。详见表 10.3-2。(R/W)

RTC_CNTL_TIMER_SYS_RST 选择 RTC 计时器的触发条件。详见表 10.3-2。(R/W)

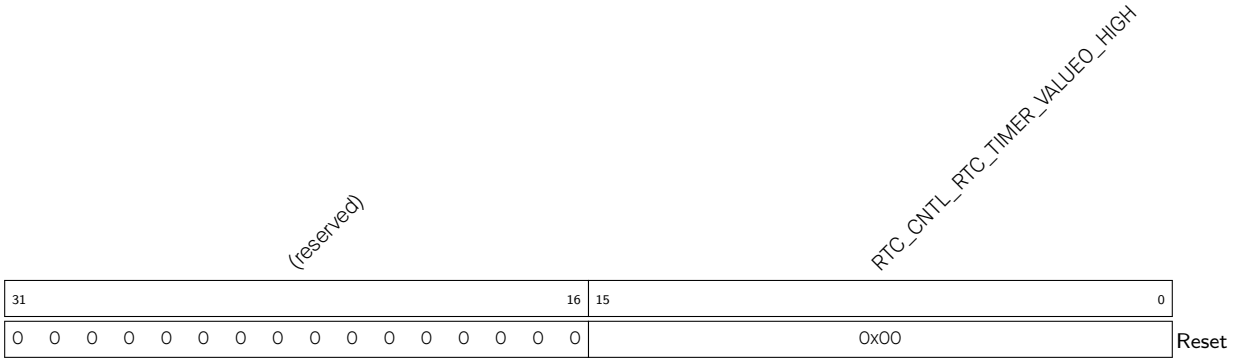
RTC_CNTL_RTC_TIME_UPDATE 选择 RTC 计时器的触发条件。详见表 10.3-2。(WO)

Register 10.5. RTC_CNTL_RTC_TIME_LOW0_REG (0x0010)



RTC_CNTL_RTC_TIMER_VALUE0_LOW 存储 RTC 计时器 0 的低 32 位。(RO)

Register 10.6. RTC_CNTL_RTC_TIME_HIGH0_REG (0x0014)



RTC_CNTL_RTC_TIMER_VALUE0_HIGH 存储 RTC 计时器 0 的高 16 位。(RO)

Register 10.7. RTC_CNTL_RTC_STATE0_REG (0x0018)

RTC_CNTL_SLEEP_EN	RTC_CNTL_SLP_REJECT	RTC_CNTL_SLP_WAKEUP	RTC_CNTL_SDIO_ACTIVE_IND	(reserved)	RTC_CNTL_APB2RTC_BRIDGE_SEL	(reserved)	RTC_CNTL_RTC_SLP_REJECT_CAUSE_CLR	RTC_CNTL_RTC_SW_CPU_INT	
31	30	29	28	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	0	Reset

RTC_CNTL_RTC_SW_CPU_INT 发送软件 RTC 中断给 CPU。(WO)

RTC_CNTL_RTC_SLP_REJECT_CAUSE_CLR 清除 RTC 拒绝入睡原因。(WO)

RTC_CNTL_APB2RTC_BRIDGE_SEL 置 1 选择桥接 APB 至 RTC。清 0 适用 sync 链接 APB 至 RTC。(R/W)

RTC_CNTL_SDIO_ACTIVE_IND 代表 SDIO 处于 active 状态。(RO)

RTC_CNTL_SLP_WAKEUP 睡眠唤醒位。(R/W)

RTC_CNTL_SLP_REJECT 拒绝入睡位。(R/W)

RTC_CNTL_SLEEP_EN 使芯片进入睡眠。(R/W)

Register 10.8. RTC_CNTL_RTC_TIMER1_REG (0x001C)

RTC_CNTL_PLL_BUF_WAIT	RTC_CNTL_XTL_BUF_WAIT	RTC_CNTL_CK8M_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 使能 CPU stall。(R/W)

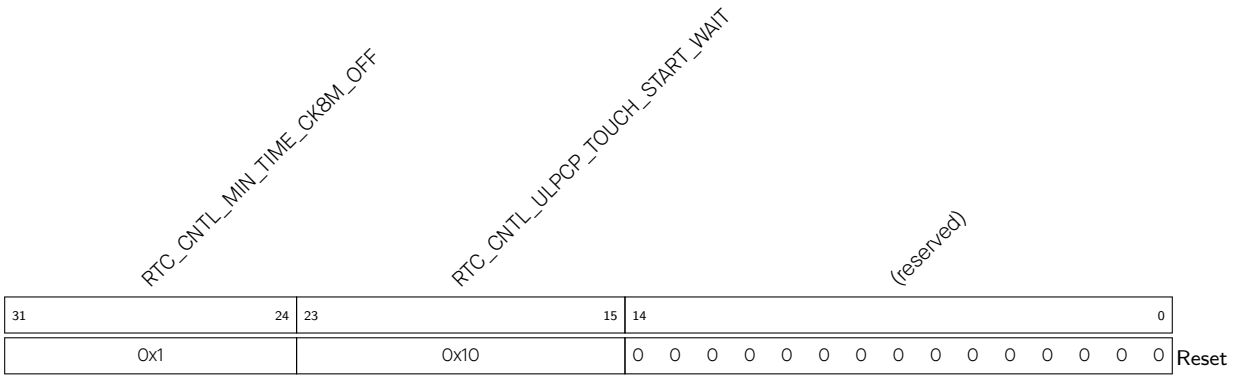
RTC_CNTL_CPU_STALL_WAIT 设置 CPU stall 的等待周期（使用 RTC 快速时钟）。(R/W)

RTC_CNTL_CK8M_WAIT 设置 FOSC 时钟的等待周期（使用 RTC 慢速时钟）。(R/W)

RTC_CNTL_XTL_BUF_WAIT 设置 XTAL 的等待周期（使用 RTC 慢速时钟）。(R/W)

RTC_CNTL_PLL_BUF_WAIT 设置 PLL 的等待周期（使用 RTC 慢速时钟）。(R/W)

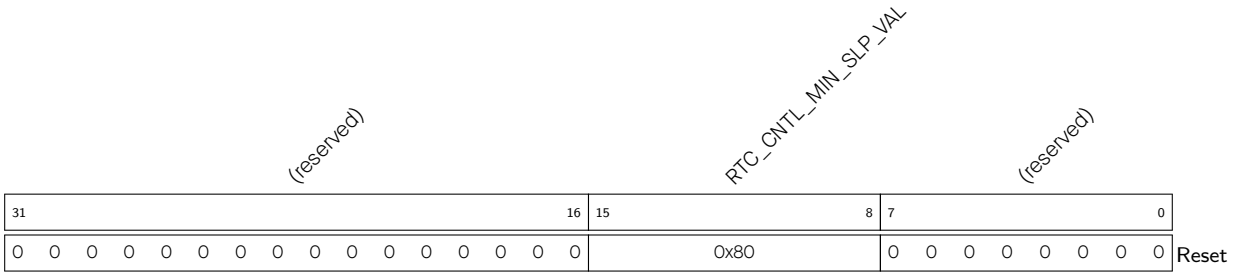
Register 10.9. RTC_CNTL_RTC_TIMER2_REG (0x0020)



RTC_CNTL_ULPCP_TOUCH_START_WAIT 设置超低功耗协处理器开始工作之前的等待周期（使用 RTC 慢速时钟）。(R/W)

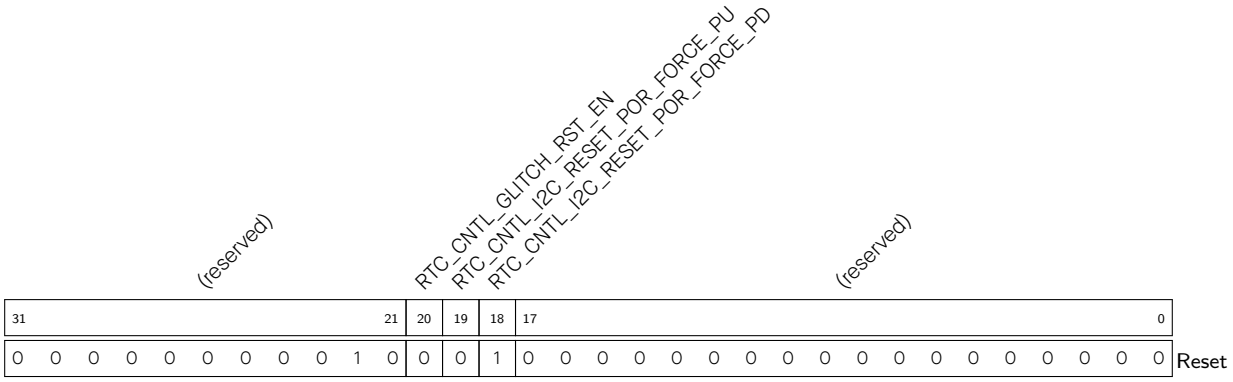
RTC_CNTL_MIN_TIME_CK8M_OFF 设置 FOSC 时钟断电时的最小等待周期（使用 RTC 慢速时钟）。(R/W)

Register 10.10. RTC_CNTL_RTC_TIMER5_REG (0x002C)



RTC_CNTL_MIN_SLP_VAL 设置最小睡眠周期（使用 RTC 慢速时钟）。(R/W)

Register 10.11. RTC_CNTL_RTC_ANA_CONF_REG (0x0034)

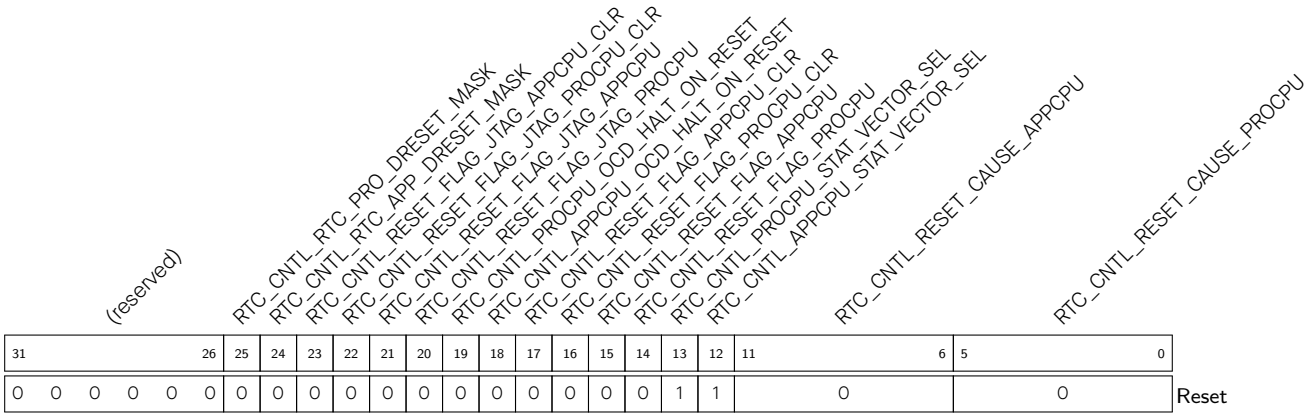


RTC_CNTL_I2C_RESET_POR_FORCE_PD 置 1 强制关闭 SLEEP_I2CPOR。(R/W)

RTC_CNTL_I2C_RESET_POR_FORCE_PU 置 1 强制打开 SLEEP_I2CPOR。(R/W)

RTC_CNTL_GLITCH_RST_EN 置 1 使能在系统监测到脉冲毛刺时进行复位。(R/W)

Register 10.12. RTC_CNTL_RTC_RESET_STATE_REG (0x0038)



RTC_CNTL_RESET_CAUSE_PROCPU 存储 CPU0 复位原因。(RO)

RTC_CNTL_RESET_CAUSE_APPCPU 存储 CPU1 复位原因。(RO)

RTC_CNTL_APPCPU_STAT_VECTOR_SEL 选择 CPU1 静态矢量 (static vector)。(R/W)

RTC_CNTL_PROCPU_STAT_VECTOR_SEL 选择 CPU0 静态矢量 (static vector)。(R/W)

RTC_CNTL_RESET_FLAG_PROCPU 设置 CPU0 复位标志。(RO)

RTC_CNTL_RESET_FLAG_APPCPU 设置 CPU1 复位标志。(RO)

RTC_CNTL_RESET_FLAG_PROCPU_CLR 置 1 清除 CPU0 复位标志。(WO)

RTC_CNTL_RESET_FLAG_APPCPU_CLR 置 1 清除 CPU1 复位标志。(WO)

RTC_CNTL_APPCPU_OCD_HALT_ON_RESET 置 1 配置 CPU1 在复位时进入 halt 状态。(R/W)

RTC_CNTL_PROCPU_OCD_HALT_ON_RESET 置 1 配置 CPU0 在复位时进入 halt 状态。(R/W)

RTC_CNTL_RESET_FLAG_JTAG_PROCPU 设置 CPU0 的 JTAG 复位标志。(RO)

RTC_CNTL_RESET_FLAG_JTAG_APPCPU 设置 CPU1 的 JTAG 复位标志。(RO)

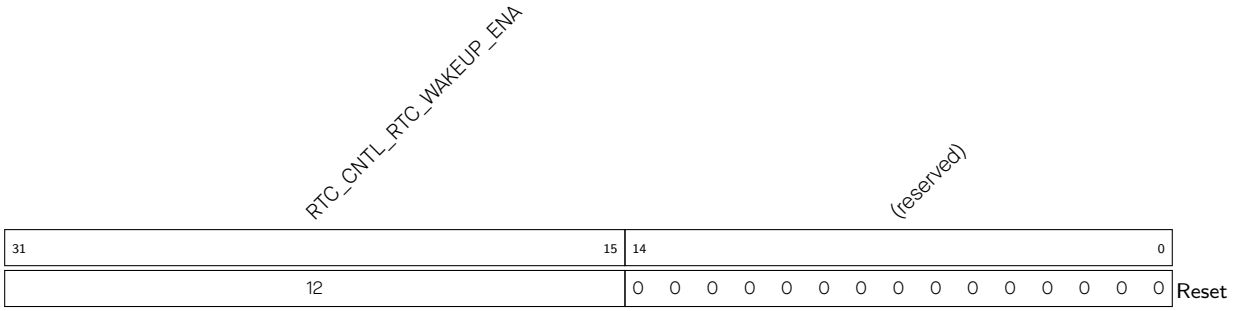
RTC_CNTL_RESET_FLAG_JTAG_PROCPU_CLR 置 1 清除 CPU0 的 JTAG 复位标志。(WO)

RTC_CNTL_RESET_FLAG_JTAG_APPCPU_CLR 置 1 清除 CPU1 的 JTAG 复位标志。(WO)

RTC_CNTL_RTC_APP_DRESET_MASK 置 1 绕过 CPU1 dreset。(R/W)

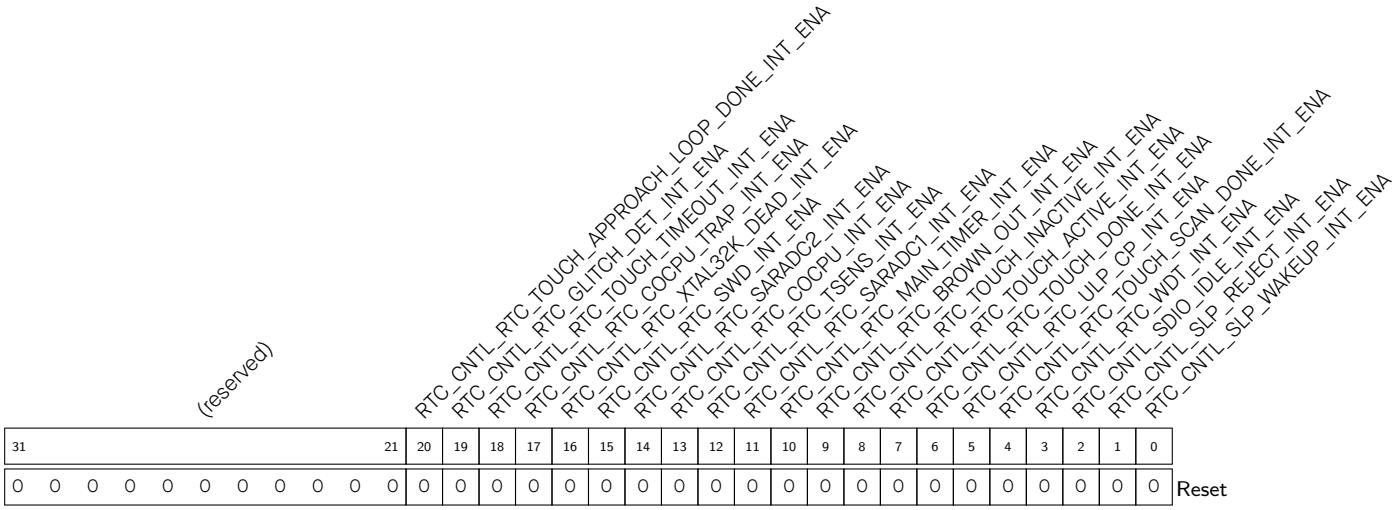
RTC_CNTL_RTC_PRO_DRESET_MASK 置 1 绕过 CPU0 dreset。(R/W)

Register 10.13. RTC_CNTL_RTC_WAKEUP_STATE_REG (0x003C)



RTC_CNTL_RTC_WAKEUP_ENA 选择唤醒源。详见表 10.4-3。(R/W)

Register 10.14. RTC_CNTL_INT_ENA_RTC_REG (0x0040)



RTC_CNTL_SLP_WAKEUP_INT_ENA 使能在芯片“从睡眠中唤醒”时发送中断。(R/W)

RTC_CNTL_SLP_REJECT_INT_ENA 使能在芯片“拒绝入睡”时发送中断。(R/W)

RTC_CNTL_SDIO_IDLE_INT_ENA 使能在 SDIO idle 时发送中断。(R/W)

RTC_CNTL_RTC_WDT_INT_ENA 使能 RTC 看门狗中断。(R/W)

RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_ENA 使能在触摸扫描完成时发送中断。(R/W)

RTC_CNTL_RTC_ULP_CP_INT_ENA 使能超低功耗协处理器中断。(R/W)

RTC_CNTL_RTC_TOUCH_DONE_INT_ENA 使能在单次触摸完成时发送中断。(R/W)

RTC_CNTL_RTC_TOUCH_ACTIVE_INT_ENA 使能在检测到触摸时发送中断。(R/W)

RTC_CNTL_RTC_TOUCH_INACTIVE_INT_ENA 使能在触摸释放时发送中断。(R/W)

RTC_CNTL_RTC_BROWN_OUT_INT_ENA 使能欠压检测中断。(R/W)

接下页...

Register 10.14. RTC_CNTL_INT_ENA_RTC_REG (0x0040)

接上页...

RTC_CNTL_RTC_MAIN_TIMER_INT_ENA 使能 RTC 主计时器中断。(R/W)

RTC_CNTL_RTC_SARADC1_INT_ENA 使能 SAR ADC1 中断。(R/W)

RTC_CNTL_RTC_TSENS_INT_ENA 使能温度传感器中断。(R/W)

RTC_CNTL_RTC_COCPU_INT_ENA 使能 ULP-RISCV 中断。(R/W)

RTC_CNTL_RTC_SARADC2_INT_ENA 使能 SAR ADC2 中断。(R/W)

RTC_CNTL_RTC_SWD_INT_ENA 使能超级看门狗中断。(R/W)

RTC_CNTL_RTC_XTAL32K_DEAD_INT_ENA 使能在 32 kHz 晶振掉电时发送中断。(R/W)

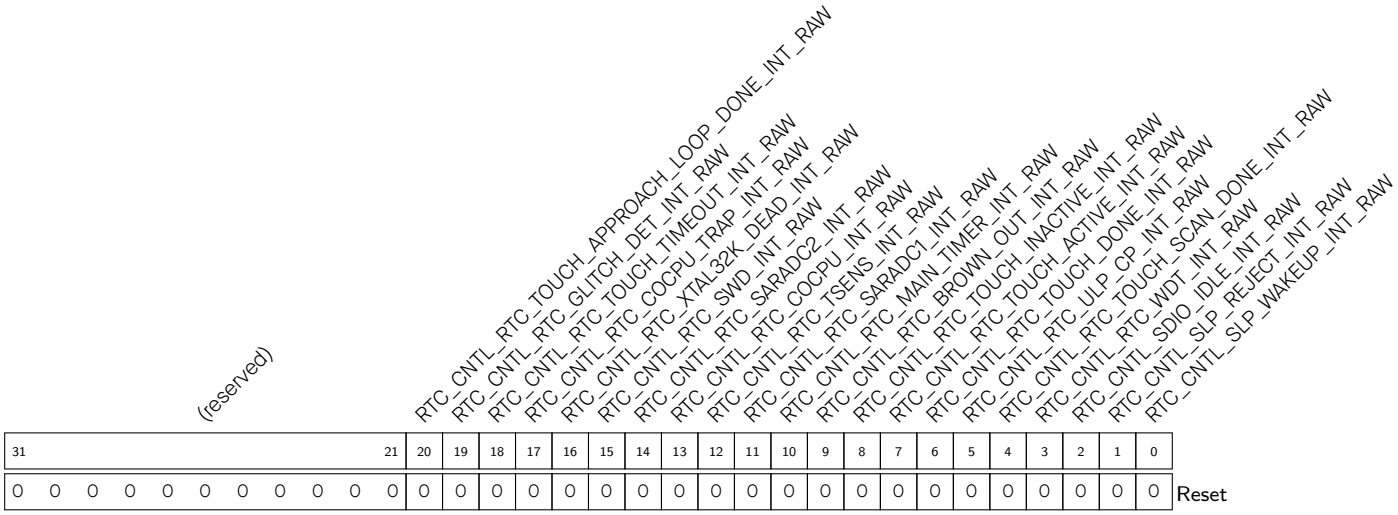
RTC_CNTL_RTC_COCPU_TRAP_INT_ENA 使能在 ULP-RISCV 被困时发送中断。(R/W)

RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_ENA 使能在触摸传感器超时时发送中断。(R/W)

RTC_CNTL_RTC_GLITCH_DET_INT_ENA 使能在检测到脉冲毛刺时发送中断。(R/W)

RTC_CNTL_RTC_TOUCH_APPROACH_LOOP_DONE_INT_ENA 使能在触摸循环完成后发送中断。(R/W)

Register 10.15. RTC_CNTL_INT_RAW_RTC_REG (0x0044)



RTC_CNTL_SLP_WAKEUP_INT_RAW 存储芯片“从睡眠中唤醒”时中断的原始中断位。(RO)

RTC_CNTL_SLP_REJECT_INT_RAW 存储芯片“拒绝入睡”时中断的原始中断位。(RO)

RTC_CNTL_SDIO_IDLE_INT_RAW 使能在 SDIO idle 时发送中断的原始中断位。(RO)

RTC_CNTL_RTC_WDT_INT_RAW 存储 RTC 看门狗中断的原始中断位。(RO)

RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_RAW 存储触摸完成时中断的原始中断位。(RO)

RTC_CNTL_RTC_ULP_CP_INT_RAW 存储超低功耗协处理器中断的原始中断位。(RO)

RTC_CNTL_RTC_TOUCH_DONE_INT_RAW 存储单次触摸完成时中断的原始中断位。(RO)

RTC_CNTL_RTC_TOUCH_ACTIVE_INT_RAW 存储检测到触摸时中断的原始中断位。(RO)

RTC_CNTL_RTC_TOUCH_INACTIVE_INT_RAW 存储触摸释放中断的原始中断位。(RO)

RTC_CNTL_RTC_BROWN_OUT_INT_RAW 存储欠压检测中断的原始中断位。(RO)

接下页...

Register 10.15. RTC_CNTL_INT_RAW_RTC_REG (0x0044)

接上页...

RTC_CNTL_RTC_MAIN_TIMER_INT_RAW 存储 RTC 主定时器中断的原始中断位。(RO)

RTC_CNTL_RTC_SARADC1_INT_RAW 存储 SAR ADC1 中断的原始中断位。(RO)

RTC_CNTL_RTC_TSENS_INT_RAW 存储温度传感器中断的原始中断位。(RO)

RTC_CNTL_RTC_COCPU_INT_RAW 存储 ULP-RISCV 中断的原始中断位。(RO)

RTC_CNTL_RTC_SARADC2_INT_RAW 存储 SAR ADC2 中断的原始中断位。(RO)

RTC_CNTL_RTC_SWD_INT_RAW 存储超级看门狗中断的原始中断位。(RO)

RTC_CNTL_RTC_XTAL32K_DEAD_INT_RAW 存储 32 kHz 晶振掉电中断的原始中断位。(RO)

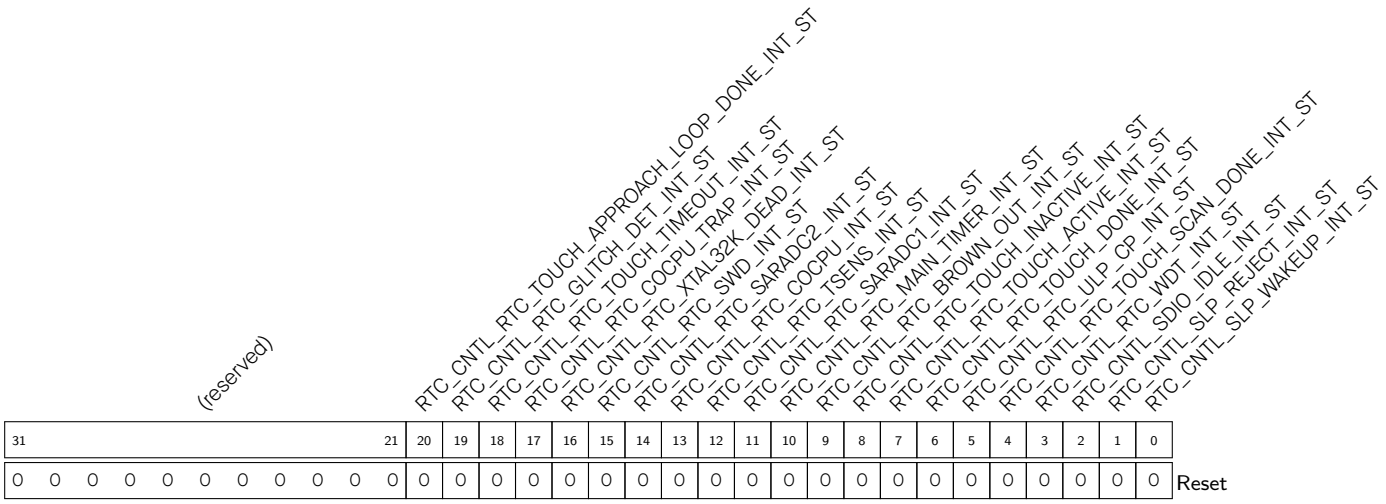
RTC_CNTL_RTC_COCPU_TRAP_INT_RAW 存储 ULP-RISCV 受困时中断的原始中断位。(RO)

RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_RAW 存储触摸传感器超时中断的原始中断位。(RO)

RTC_CNTL_RTC_GLITCH_DET_INT_RAW 存储在检测到脉冲毛刺时中断的原始中断位。(RO)

RTC_CNTL_RTC_TOUCH_APPROACH_LOOP_DONE_INT_RAW 存储在触摸循环完成后发送中断的原始中断位。(R/W)

Register 10.16. RTC_CNTL_INT_ST_RTC_REG (0x0048)



RTC_CNTL_SLP_WAKEUP_INT_ST 存储芯片“从睡眠中唤醒”时中断的中断状态。(RO)

RTC_CNTL_SLP_REJECT_INT_ST 存储芯片“拒绝进入睡眠”时中断的中断状态。(RO)

RTC_CNTL_SDIO_IDLE_INT_ST 存储在 SDIO idle 时发送中断的中断状态。(RO)

RTC_CNTL_RTC_WDT_INT_ST 存储 RTC 看门狗中断的中断状态。(RO)

RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_ST 存储触摸扫描完成时中断的中断状态。(RO)

RTC_CNTL_RTC_ULP_CP_INT_ST 存储超低功耗协处理器中断的中断状态。(RO)

RTC_CNTL_RTC_TOUCH_DONE_INT_ST 存储单次触摸完成时中断的中断状态。(RO)

RTC_CNTL_RTC_TOUCH_ACTIVE_INT_ST 存储检测到触摸时中断的中断状态。(RO)

RTC_CNTL_RTC_TOUCH_INACTIVE_INT_ST 存储触摸释放时中断的中断状态。(RO)

RTC_CNTL_RTC_BROWN_OUT_INT_ST 存储欠压检测中断的中断状态。(RO)

接下页...

Register 10.16. RTC_CNTL_INT_ST_RTC_REG (0x0048)

接上页...

RTC_CNTL_RTC_MAIN_TIMER_INT_ST 存储 RTC 主定时器中断的中断状态。(RO)

RTC_CNTL_RTC_SARADC1_INT_ST 存储 SAR ADC1 中断的中断状态。(RO)

RTC_CNTL_RTC_TSENS_INT_ST 存储温度传感器中断的中断状态。(RO)

RTC_CNTL_RTC_COCPU_INT_ST 存储 ULP-RISCV 中断的中断状态。(RO)

RTC_CNTL_RTC_SARADC2_INT_ST 存储 SAR ADC2 中断的中断状态。(RO)

RTC_CNTL_RTC_SWD_INT_ST 存储超级看门狗中断的中断状态。(RO)

RTC_CNTL_RTC_XTAL32K_DEAD_INT_ST 存储 32 kHz 掉电中断的中断状态。(RO)

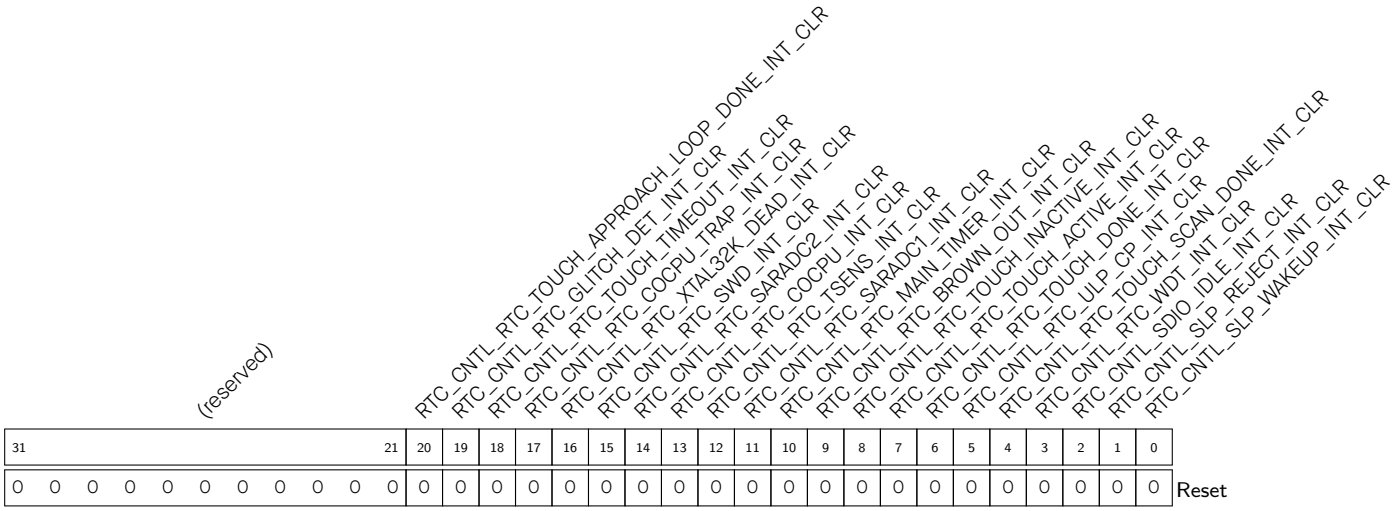
RTC_CNTL_RTC_COCPU_TRAP_INT_ST 存储 ULP-RISCV 受困时中断的中断状态。(RO)

RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_ST 存储触摸传感器超时时中断的中断状态。(RO)

RTC_CNTL_RTC_GLITCH_DET_INT_ST 存储检测到脉冲毛刺时中断的中断状态。(RO)

RTC_CNTL_RTC_TOUCH_APPROACH_LOOP_DONE_INT_ST 存储在触摸循环完成后发送中断的中断状态。(RO)

Register 10.17. RTC_CNTL_INT_CLR_RTC_REG (0x004C)



RTC_CNTL_SLP_WAKEUP_INT_CLR 清除芯片“从睡眠中唤醒”时的中断。(WO)

RTC_CNTL_SLP_REJECT_INT_CLR 清除芯片“拒绝入睡”时的中断。(WO)

RTC_CNTL_SDIO_IDLE_INT_CLR 清除 SDIO idle 时发送的中断。(WO)

RTC_CNTL_RTC_WDT_INT_CLR 清除 RTC 看门狗中断。(WO)

RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_CLR 清除触摸完成时中断。(WO)

RTC_CNTL_RTC_ULP_CP_INT_CLR 清除超低功耗斜处理器中断。(WO)

RTC_CNTL_RTC_TOUCH_DONE_INT_CLR 清除单次触摸完成时中断。(WO)

RTC_CNTL_RTC_TOUCH_ACTIVE_INT_CLR 清除在发生触摸时出发的中断。(WO)

RTC_CNTL_RTC_TOUCH_INACTIVE_INT_CLR 清除在触摸释放时发生的中断。(WO)

RTC_CNTL_RTC_BROWN_OUT_INT_CLR 清除欠压监测中断。(WO)

接下页...

Register 10.17. RTC_CNTL_INT_CLR_RTC_REG (0x004C)

接上页...

RTC_CNTL_RTC_MAIN_TIMER_INT_CLR 清除 RTC 主定时器中断。(WO)

RTC_CNTL_RTC_SARADC1_INT_CLR 清除 SAR ADC1 中断。(WO)

RTC_CNTL_RTC_TSENS_INT_CLR 清除温度传感器中断。(WO)

RTC_CNTL_RTC_COCPU_INT_CLR 清除 ULP-RISCV 中断。(WO)

RTC_CNTL_RTC_SARADC2_INT_CLR 清除 SAR ADC2 中断。(WO)

RTC_CNTL_RTC_SWD_INT_CLR 清除超级看门狗中断。(WO)

RTC_CNTL_RTC_XTAL32K_DEAD_INT_CLR 清除 32 kHz 晶振掉电中断。(WO)

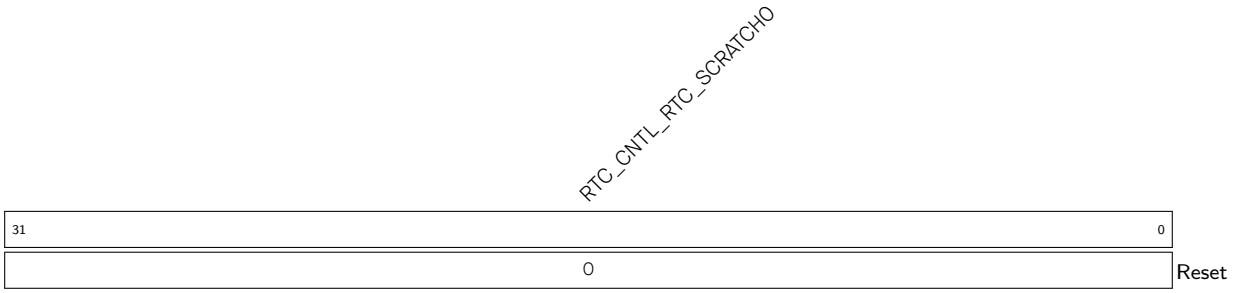
RTC_CNTL_RTC_COCPU_TRAP_INT_CLR 清除 ULP-RISCV 受困时中断。(WO)

RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_CLR 清除触摸传感器超时中断。(WO)

RTC_CNTL_RTC_GLITCH_DET_INT_CLR 清除脉冲毛刺中断。(WO)

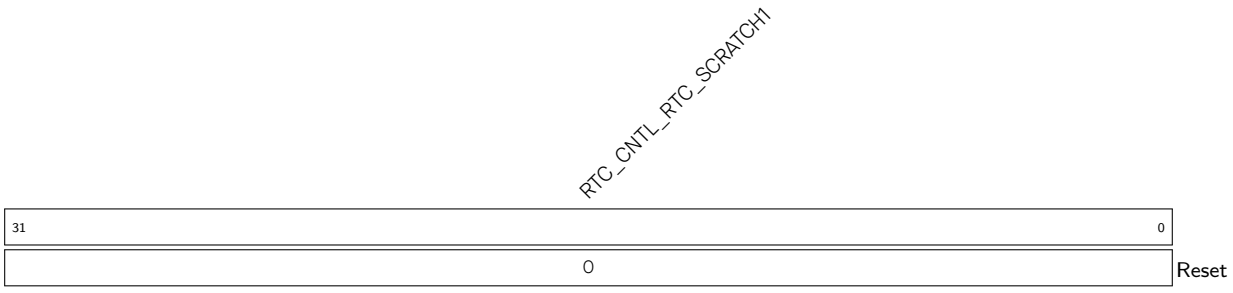
RTC_CNTL_RTC_TOUCH_APPROACH_LOOP_DONE_INT_CLR 清除触摸循环完成后中断。(WO)

Register 10.18. RTC_CNTL_RTC_STORE0_REG (0x0050)



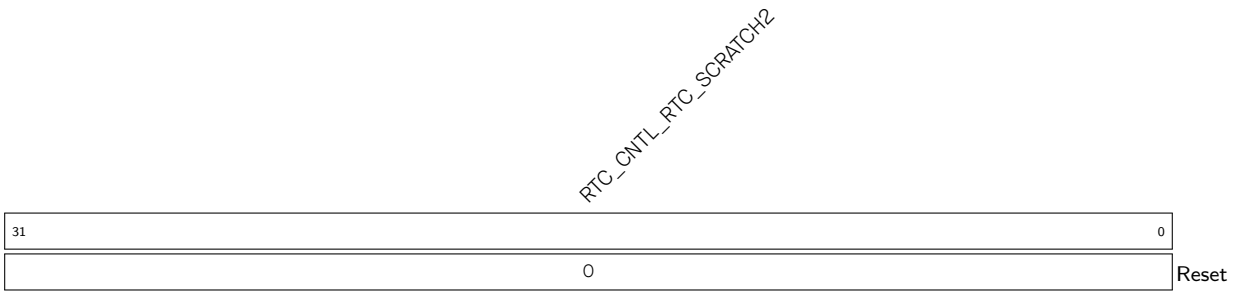
RTC_CNTL_RTC_SCRATCH0 Retention 寄存器。(R/W)

Register 10.19. RTC_CNTL_RTC_STORE1_REG (0x0054)



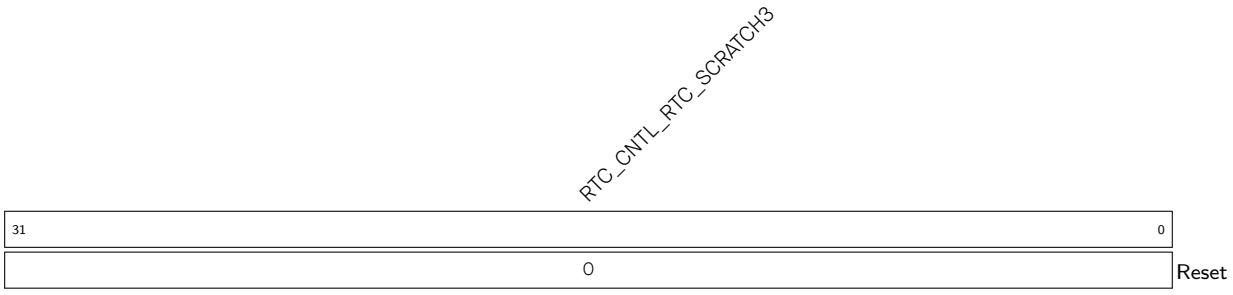
RTC_CNTL_RTC_SCRATCH1 Retention 寄存器。(R/W)

Register 10.20. RTC_CNTL_RTC_STORE2_REG (0x0058)



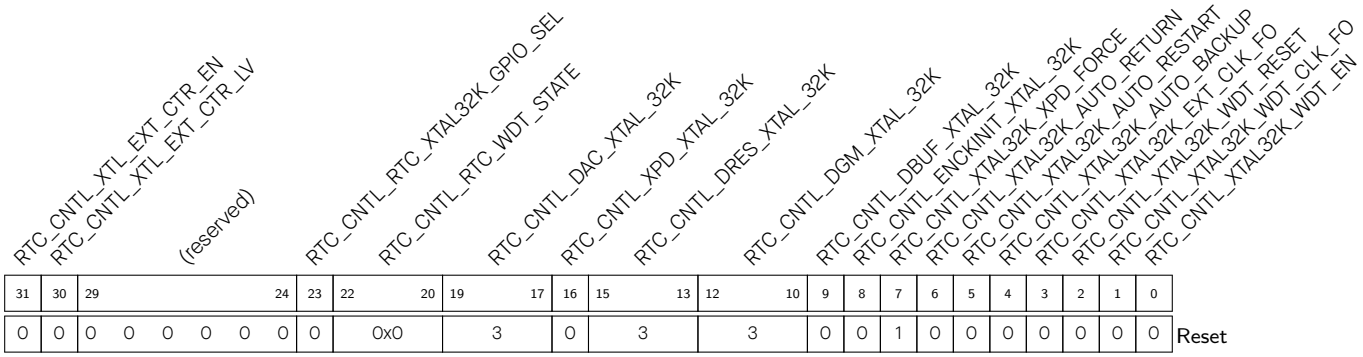
RTC_CNTL_RTC_SCRATCH2 Retention 寄存器。(R/W)

Register 10.21. RTC_CNTL_RTC_STORE3_REG (0x005C)



RTC_CNTL_RTC_SCRATCH3 Retention 寄存器。(R/W)

Register 10.22. RTC_CNTL_RTC_EXT_XTL_CONF_REG (0x0060)



RTC_CNTL_XTAL32K_WDT_EN 置 1 使能 32 kHz 晶振看门狗。(R/W)

RTC_CNTL_XTAL32K_WDT_CLK_FO 置 1 强制打开 32 kHz 晶振看门狗时钟。(R/W)

RTC_CNTL_XTAL32K_WDT_RESET 置 1 软件复位 32 kHz 晶振软件看门狗。(R/W)

RTC_CNTL_XTAL32K_EXT_CLK_FO 置 1 强制打开外部 32 kHz 晶振时钟。(R/W)

RTC_CNTL_XTAL32K_AUTO_BACKUP 置 1 在 32 kHz 晶振掉电时切换至后备时钟。(R/W)

RTC_CNTL_XTAL32K_AUTO_RESTART 置 1 在 32 kHz 晶振掉电时自动重启 32 kHz 晶振。(R/W)

RTC_CNTL_XTAL32K_AUTO_RETURN 置 1 在 32 kHz 晶振重启时自动切换回 32 kHz 晶振。(R/W)

RTC_CNTL_XTAL32K_XPD_FORCE 置 1 允许软件强制关闭 32 kHz 晶振，置 0 允许 FSM 强制关闭 32 kHz 晶振。(R/W)

RTC_CNTL_ENCKINIT_XTAL_32K 置 1 使用内部时钟协助 32 kHz 晶振重启。(R/W)

RTC_CNTL_DBUF_XTAL_32K 0: single-end buffer 1: differential buffer (R/W)

RTC_CNTL_DGM_XTAL_32K 配置 32 kHz 晶振 gm 控制。(R/W)

接下页...

Register 10.22. RTC_CNTL_RTC_EXT_XTL_CONF_REG (0x0060)

接上页...

RTC_CNTL_DRES_XTAL_32K DRES_XTAL_32K (R/W)

RTC_CNTL_XPD_XTAL_32K XPD_XTAL_32K (R/W)

RTC_CNTL_DAC_XTAL_32K DAC_XTAL_32K (R/W)

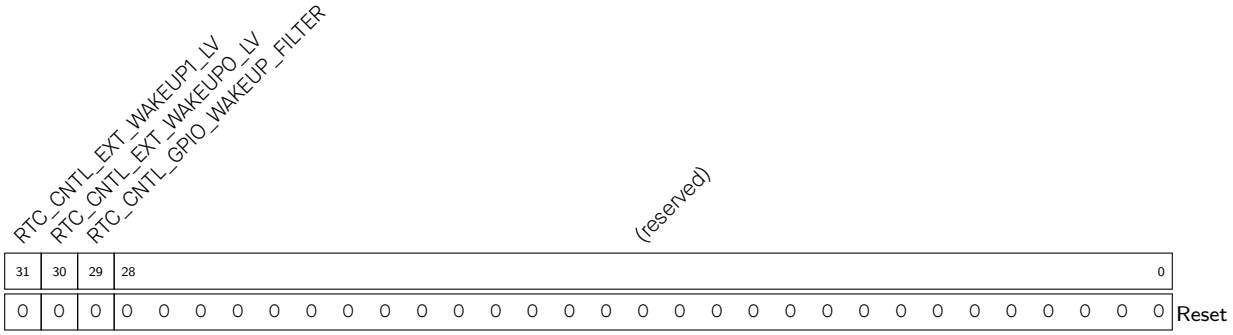
RTC_CNTL_RTC_WDT_STATE 存储 32 kHz 看门狗状态。(RO)

RTC_CNTL_RTC_XTAL32K_GPIO_SEL 选择 32 kHz 晶振时钟。0：外部 32 kHz 时钟；1：选择来自 RTC GPIO X32P_C 的时钟。(R/W)

RTC_CNTL_XTL_EXT_CTR_LV 0：高电平 XTAL 掉电；1：低电平 XTAL 掉电。(R/W)

RTC_CNTL_XTL_EXT_CTR_EN 置 1 使能 GPIO 控制晶振掉电。(R/W)

Register 10.23. RTC_CNTL_RTC_EXT_WAKEUP_CONF_REG (0x0064)

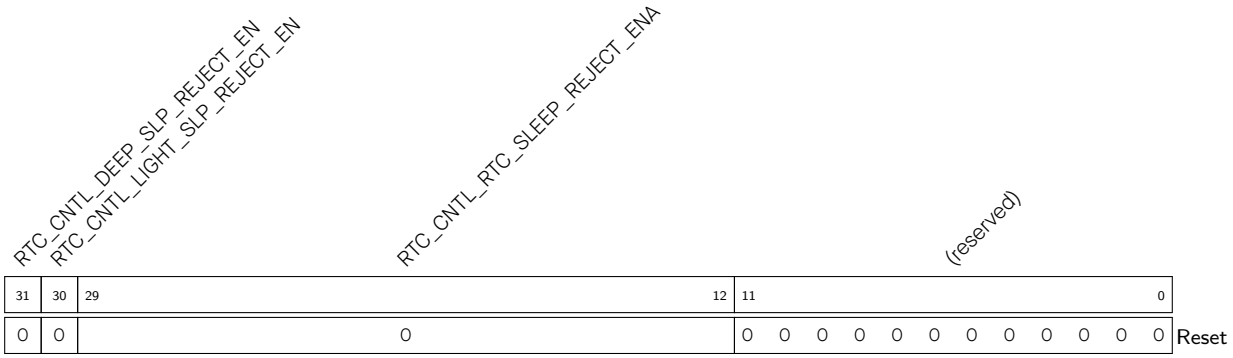


RTC_CNTL_GPIO_WAKEUP_FILTER 置 1 使能 GPIO 唤醒事件过滤器。(R/W)

RTC_CNTL_EXT_WAKEUP0_LV 0：低电平外部唤醒 0；1：高电平外部唤醒 0。(R/W)

RTC_CNTL_EXT_WAKEUP1_LV 0：低电平外部唤醒 1；1：高电平外部唤醒 1。(R/W)

Register 10.24. RTC_CNTL_RTC_SLP_REJECT_CONF_REG (0x0068)

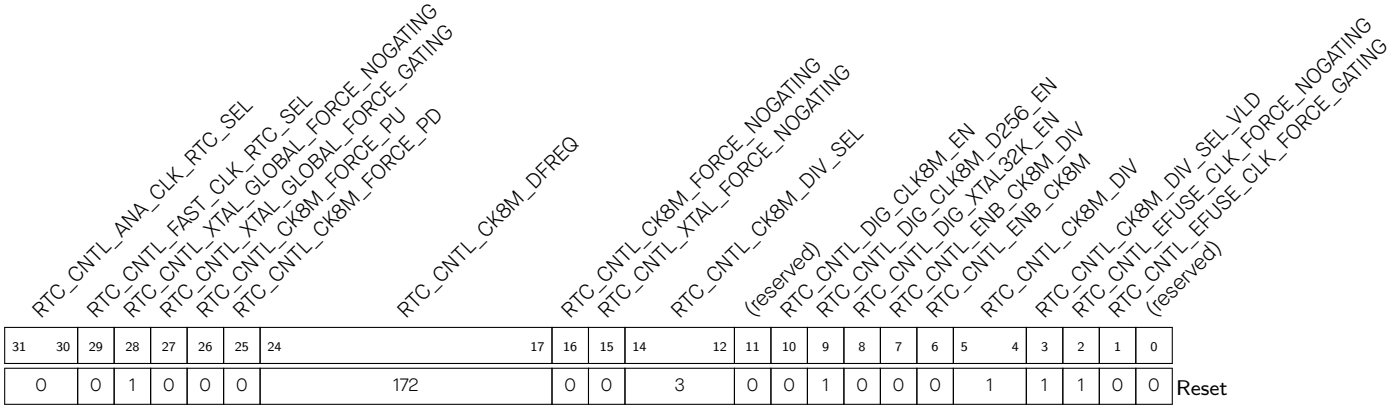


RTC_CNTL_RTC_SLEEP_REJECT_ENA 置 1 使能“拒绝入睡”。(R/W)

RTC_CNTL_LIGHT_SLP_REJECT_EN 置 1 使能“拒绝进入 Light-sleep”。(R/W)

RTC_CNTL_DEEP_SLP_REJECT_EN 置 1 使能“拒绝进入 Deep-sleep”。(R/W)

Register 10.25. RTC_CNTL_RTC_CLK_CONF_REG (0x0074)



RTC_CNTL_EFUSE_CLK_FORCE_GATING 置 1 强制打开 eFuse 时钟门控。(R/W)

RTC_CNTL_EFUSE_CLK_FORCE_NOGATING 置 1 强制关闭 eFuse 时钟门控。(R/W)

RTC_CNTL_CK8M_DIV_SEL_VLD 同步 reg_fosc_div_sel。注意在修改分频器前必须先使总线无效，然后重新使分频器时钟生效。(R/W)

RTC_CNTL_CK8M_DIV 配置 FOSC_D256_OUT 的分频器 00: 128 分频，01: 256 分频，10: 512 分频，11: 1024 分频。(R/W)

RTC_CNTL_ENB_CK8M 置 1 禁用 FOSC 和 FOSC_D256_OUT。(R/W)

RTC_CNTL_ENB_CK8M_DIV 选择 FOSC_D256_OUT。1: FOSC，0: FOSC 的 256 分频。(R/W)

RTC_CNTL_DIG_XTAL32K_EN 置 1 使能数字系统 CK_XTAL_32K 时钟。R/W)

RTC_CNTL_DIG_CLK8M_D256_EN 置 1 使能数字系统 FOSC_D256_OUT 时钟。(R/W)

RTC_CNTL_DIG_CLK8M_EN 置 1 使能数字系统 FOSC 时钟。(R/W)

RTC_CNTL_CK8M_DIV_SEL 存储 FOSC 分频数，即 reg_FOSC_div_sel + 1。(R/W)

接下页...

Register 10.25. RTC_CNTL_RTC_CLK_CONF_REG (0x0074)

接上页...

RTC_CNTL_XTAL_FORCE_NOGATING 置 1 强制关闭 Sleep 状态下的晶振门控。(R/W)

RTC_CNTL_CK8M_FORCE_NOGATING 置 1 强制关闭 Sleep 状态下的 FOSC 门控。(R/W)

RTC_CNTL_CK8M_DFREQ CK8M_DFREQ (R/W)

RTC_CNTL_CK8M_FORCE_PD 置 1 强制关闭 RC_FAST_CLK 时钟。(R/W)

RTC_CNTL_CK8M_FORCE_PU 置 1 强制打开 RC_FAST_CLK 时钟。(R/W)

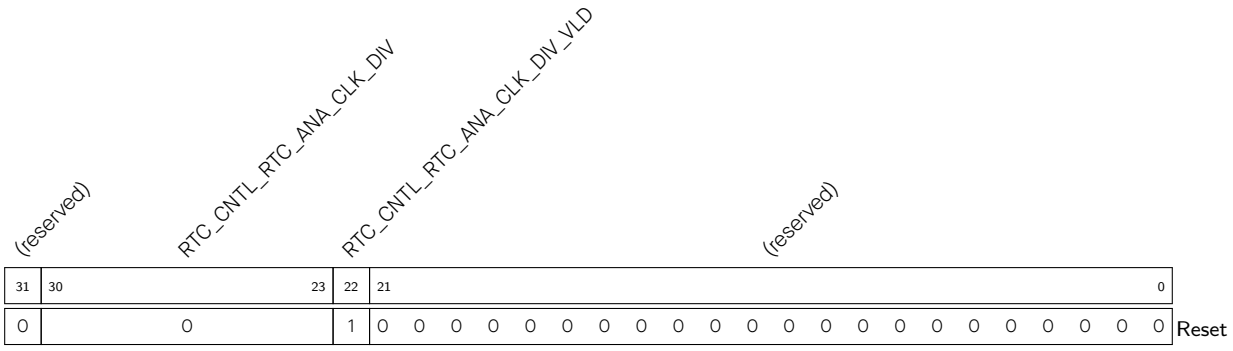
RTC_CNTL_XTAL_GLOBAL_FORCE_GATING 置 1 强制打开 XTAL 时钟门控。(R/W)

RTC_CNTL_XTAL_GLOBAL_FORCE_NOGATING 置 1 强制绕过 XTAL 时钟门控。(R/W)

RTC_CNTL_FAST_CLK_RTC_SEL 选择 RTC 快速时钟。0: XTAL_DIV_CLK，1: RC_FAST_CLK div n。(R/W)

RTC_CNTL_ANA_CLK_RTC_SEL 选择 RTC 慢速时钟。0: RC_SLOW_CLK，1: XTAL32K_CLK，2: RC_FAST_DIV_CLK。(R/W)

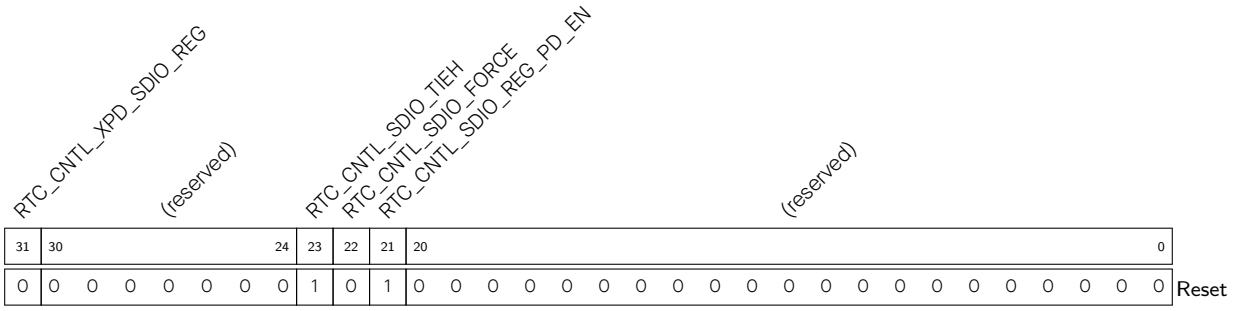
Register 10.26. RTC_CNTL_RTC_SLOW_CLK_CONF_REG (0x0078)



RTC_CNTL_RTC_ANA_CLK_DIV_VLD 同步 reg_fosc_div_sel 信号。注意在修改分频器前必须先使总线无效，然后重新使分频器时钟生效。(R/W)

RTC_CNTL_RTC_ANA_CLK_DIV 配置 RTC 时钟的分频器。(R/W)

Register 10.27. RTC_CNTL_RTC_SDIO_CONF_REG (0x007C)



RTC_CNTL_SDIO_REG_PD_EN 置 1 在芯片处于 deep sleep 和 light sleep 时使 SDIO_REG 掉电。本字段仅在 RTC_CNTL_SDIO_FORCE = 0 时有效。(R/W)

RTC_CNTL_SDIO_FORCE 置 1 使用软件控制 SDIO_REG，清 0 使用状态机控制 SDIO_REG。(R/W)

RTC_CNTL_SDIO_TIEH 软件配置 SDIO_TIEH。本字段仅在 RTC_CNTL_SDIO_FORCE = 1 时有效。(R/W)

RTC_CNTL_XPD_SDIO_REG 置 1 使 flash 调压器上电。(R/W)

Register 10.28. RTC_CNTL_RTC_REG (0x0084)



RTC_CNTL_DIG_REG_CAL_EN 置 1 使能数字调压器的软件校准。(R/W)

RTC_CNTL_SCK_DCAP 配置 RC_SLOW_CLK 频率。(R/W)

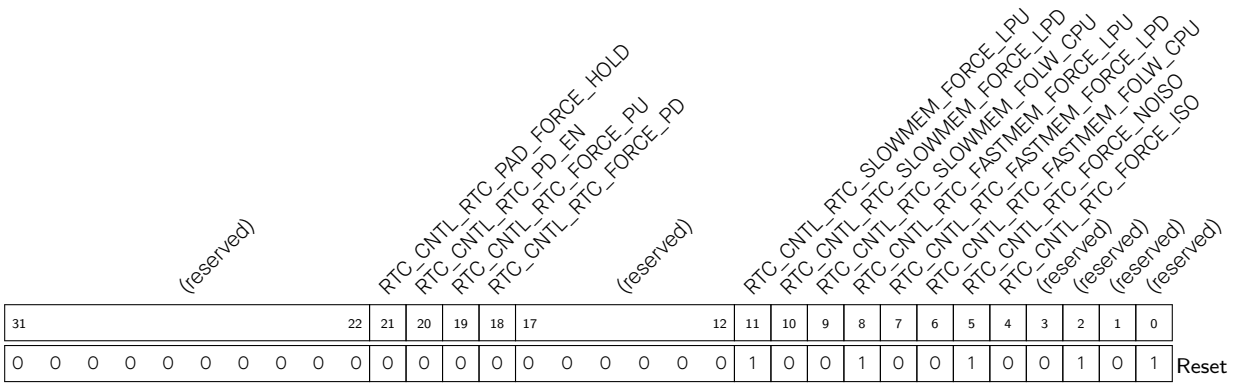
RTC_CNTL_RTC_DBOOST_FORCE_PD 置 1 强制关闭 RTC_DBOOST。(R/W)

RTC_CNTL_RTC_DBOOST_FORCE_PU 置 1 强制打开 RTC_DBOOST。(R/W)

RTC_CNTL_RTC_REGULATOR_FORCE_PD 置 1 强制关闭低功耗调压器（即将电压降低至 0.8 V 及以下）。(R/W)

RTC_CNTL_RTC_REGULATOR_FORCE_PU 置 1 强制打开低功耗调压器（即将电压升高至 0.8 V 以上）。(R/W)

Register 10.29. RTC_CNTL_RTC_PWC_REG (0x0088)



RTC_CNTL_RTC_FORCE_ISO 置 1 强制隔离 RTC 外设。(R/W)

RTC_CNTL_RTC_FORCE_NOISO 置 1 强制不隔离 RTC 外设。(R/W)

RTC_CNTL_RTC_FASTMEM_FOLW_CPU 置 1 强制在 CPU 掉电时关闭 RTC 慢速内存；清 0 强制在主状态机掉电时关闭 RTC 慢速内存。(R/W)

RTC_CNTL_RTC_FASTMEM_FORCE_LPD 置 1 禁止快速内存在睡眠中进入 retention 模式。(R/W)

RTC_CNTL_RTC_FASTMEM_FORCE_LPU 置 1 强制快速内存在睡眠中进入 retention 模式。(R/W)

RTC_CNTL_RTC_SLOWMEM_FOLW_CPU 置 1 强制 RTC 内存跟随 CPU 掉电；清 0 强制 RTC 内存跟随主状态机掉电。(R/W)

接下页...

Register 10.29. RTC_CNTL_RTC_PWC_REG (0x0088)

接上页...

RTC_CNTL_RTC_SLOWMEM_FORCE_LPD 置 1 禁止慢速内存在睡眠中进入 retention 模式。(R/W)

RTC_CNTL_RTC_SLOWMEM_FORCE_LPU 置 1 强制慢速内存在睡眠中进入 retention 模式。(R/W)

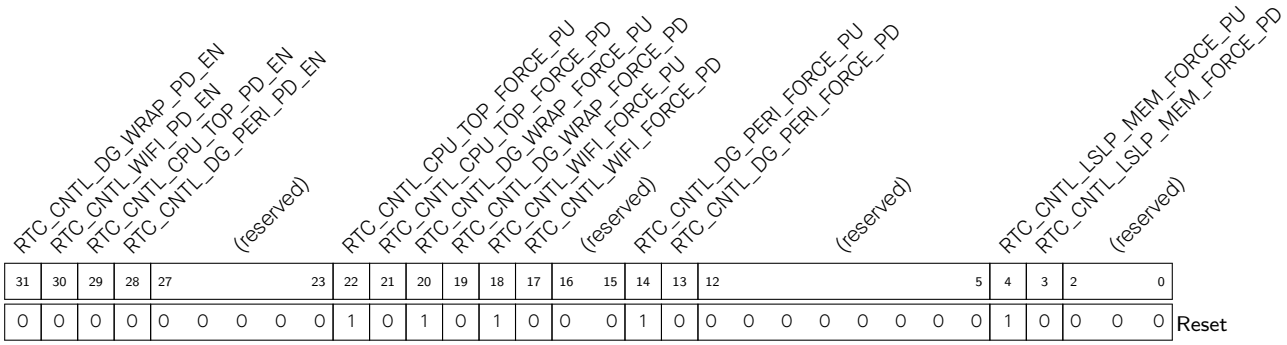
RTC_CNTL_RTC_FORCE_PD 置 1 强制关闭 RTC 外设。(R/W)

RTC_CNTL_RTC_FORCE_PU 置 1 强制打开 RTC 外设。(R/W)

RTC_CNTL_RTC_PD_EN 置 1 允许 RTC 外设在睡眠中掉电。(R/W)

RTC_CNTL_RTC_PAD_FORCE_HOLD 置 1 强制数字 GPIO 进入 hold 状态。(R/W)

Register 10.30. RTC_CNTL_DIG_PWC_REG (0x0090)



RTC_CNTL_LSLP_MEM_FORCE_PD 置 1 强制数字系统中的内存在睡眠中不进入 retention 模式。(R/W)

RTC_CNTL_LSLP_MEM_FORCE_PU 置 1 强制数字系统中的内存在睡眠时进入 retention 模式。(R/W)

RTC_CNTL_DG_PERI_FORCE_PD 置 1 强制关闭 PD 外设。(R/W)

RTC_CNTL_DG_PERI_FORCE_PU 置 1 强制打开 PD 外设。(R/W)

RTC_CNTL_WIFI_FORCE_PD 置 1 强制关闭 Wi-Fi. (R/W)

RTC_CNTL_WIFI_FORCE_PU 置 1 强制打开 Wi-Fi. (R/W)

RTC_CNTL_DG_WRAP_FORCE_PD 置 1 强制关闭数字内核。(R/W)

RTC_CNTL_DG_WRAP_FORCE_PU 置 1 强制打开数字内核。(R/W)

RTC_CNTL_CPU_TOP_FORCE_PD 置 1 强制关闭 CPU。(R/W)

RTC_CNTL_CPU_TOP_FORCE_PU 置 1 强制打开 CPU。(R/W)

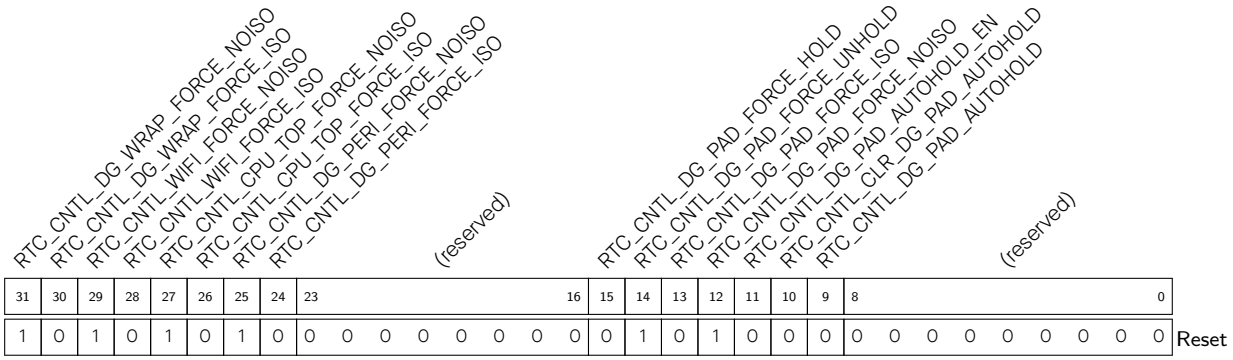
RTC_CNTL_DG_PERI_PD_EN 置 1 使能在睡眠中强制关闭数字外设。(R/W)

RTC_CNTL_CPU_TOP_PD_EN 置 1 使能在睡眠中强制关闭 CPU。(R/W)

RTC_CNTL_WIFI_PD_EN 置 1 使能在睡眠中强制关闭 wireless 模块。(R/W)

RTC_CNTL_DG_WRAP_PD_EN 置 1 使能在睡眠中强制关闭数字系统。(R/W)

Register 10.31. RTC_CNTL_DIG_ISO_REG (0x0094)



RTC_CNTL_DG_PAD_AUTOHOLD 指示数字 GPIO 的 auto-hold 状态。(RO)

RTC_CNTL_CLR_DG_PAD_AUTOHOLD 置 1 取消数字 GPIO 的 auto-hold 状态。(WO)

RTC_CNTL_DG_PAD_AUTOHOLD_EN 置 1 允许数字 GPIO 进入 auto-hold 状态。(R/W)

RTC_CNTL_DG_PAD_FORCE_NOISO 置 1 强制不隔离数字 GPIO。(R/W)

RTC_CNTL_DG_PAD_FORCE_ISO 置 1 强制隔离数字 GPIO。(R/W)

RTC_CNTL_DG_PAD_FORCE_UNHOLD 置 1 强制数字 GPIO 进入 unhold 状态。(R/W)

RTC_CNTL_DG_PAD_FORCE_HOLD 置 1 强制数字 GPIO 进入 hold 状态。(R/W)

RTC_CNTL_DG_PERI_FORCE_ISO 置 1 强制隔离数字外设。(R/W)

RTC_CNTL_DG_PERI_FORCE_NOISO 置 1 强制不隔离数字外设。(R/W)

RTC_CNTL_CPU_TOP_FORCE_ISO 置 1 强制隔离 CPU。(R/W)

RTC_CNTL_CPU_TOP_FORCE_NOISO 置 1 强制不隔离 CPU。(R/W)

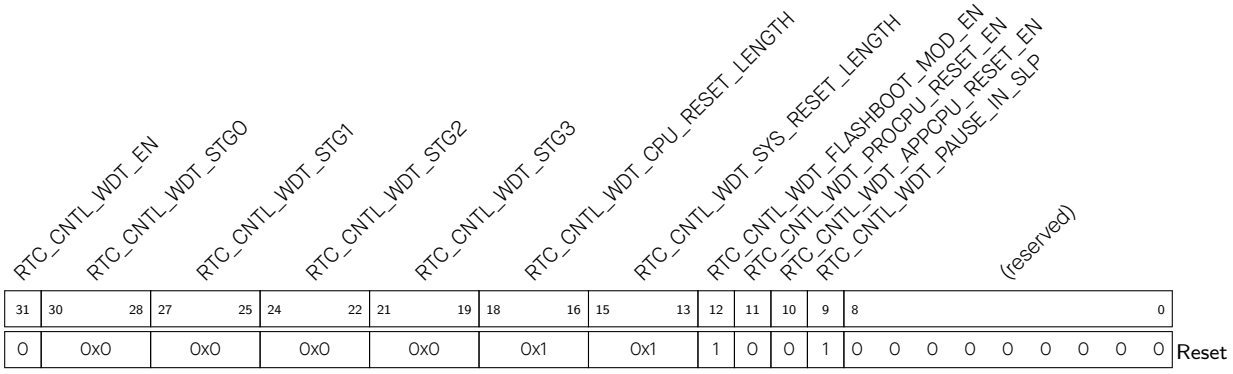
RTC_CNTL_WIFI_FORCE_ISO 置 1 强制隔离 wireless 电路。(R/W)

RTC_CNTL_WIFI_FORCE_NOISO 置 1 强制不隔离 wireless 电路。(R/W)

RTC_CNTL_DG_WRAP_FORCE_ISO 置 1 强制隔离数字系统。(R/W)

RTC_CNTL_DG_WRAP_FORCE_NOISO 置 1 强制不隔离数字系统。(R/W)

Register 10.32. RTC_CNTL_RTC_WDTCONFIG0_REG (0x0098)



RTC_CNTL_WDT_PAUSE_IN_SLP 置 1 设置“睡眠中看门狗暂停使用”。(R/W)

RTC_CNTL_WDT_APPCPU_RESET_EN 置 1 允许看门狗重启 CPU1。(R/W)

RTC_CNTL_WDT_PROCPU_RESET_EN 置 1 允许看门狗重启 CPU0。(R/W)

RTC_CNTL_WDT_FLASHBOOT_MOD_EN 置 1 在芯片从 flash 重启时使能看门狗。(R/W)

RTC_CNTL_WDT_SYS_RESET_LENGTH 设置系统复位计数器的长度。(R/W)

RTC_CNTL_WDT_CPU_RESET_LENGTH 设置 CPU 复位计数器的长度。(R/W)

RTC_CNTL_WDT_STG3 1：在中断阶段使能，2：在 CPU 阶段使能，3：在系统阶段使能，4：在系统和 RTC 阶段使能。(R/W)

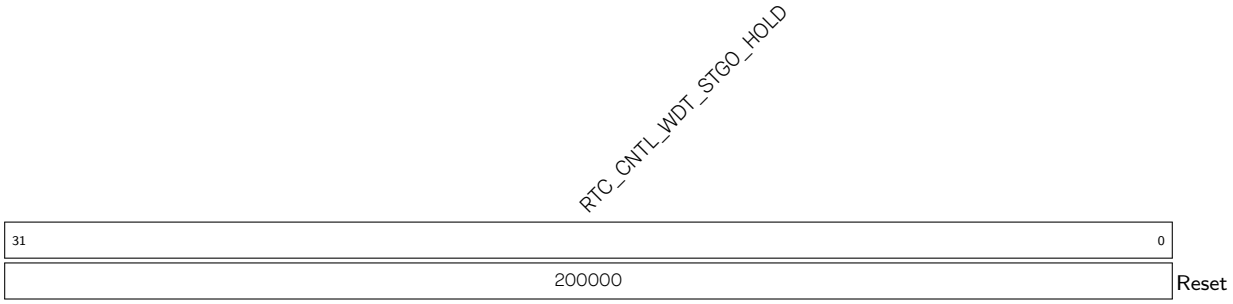
RTC_CNTL_WDT_STG2 1：在中断阶段使能，2：在 CPU 阶段使能，3：在系统阶段使能，4：在系统和 RTC 阶段使能。(R/W)

RTC_CNTL_WDT_STG1 1：在中断阶段使能，2：在 CPU 阶段使能，3：在系统阶段使能，4：在系统和 RTC 阶段使能。(R/W)

RTC_CNTL_WDT_STG0 1：在中断阶段使能，2：在 CPU 阶段使能，3：在系统阶段使能，4：在系统和 RTC 阶段使能。(R/W)

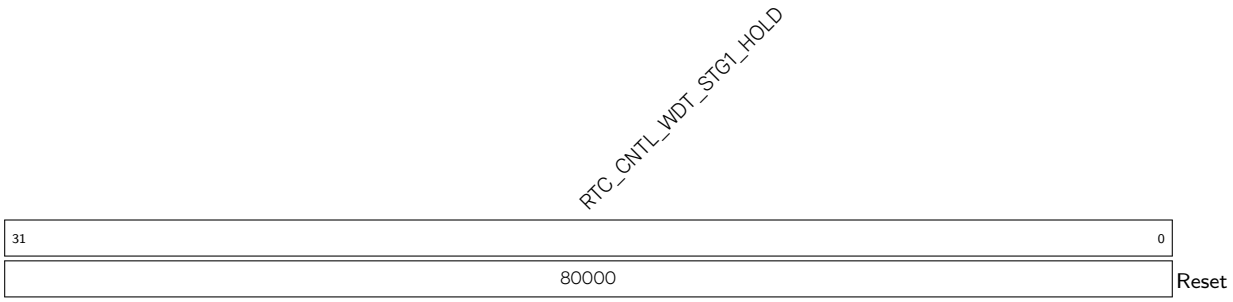
RTC_CNTL_WDT_EN 置 1 使能 RTC 看门狗。(R/W)

Register 10.33. RTC_CNTL_RTC_WDTCONFIG1_REG (0x009C)



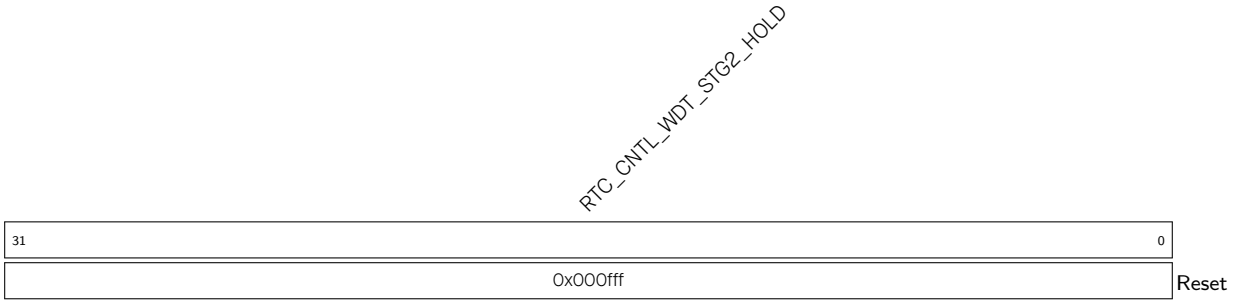
RTC_CNTL_WDT_STG0_HOLD 配置阶段 1 的 RTC 看门狗保持时间。(R/W)

Register 10.34. RTC_CNTL_RTC_WDTCONFIG2_REG (0x00A0)



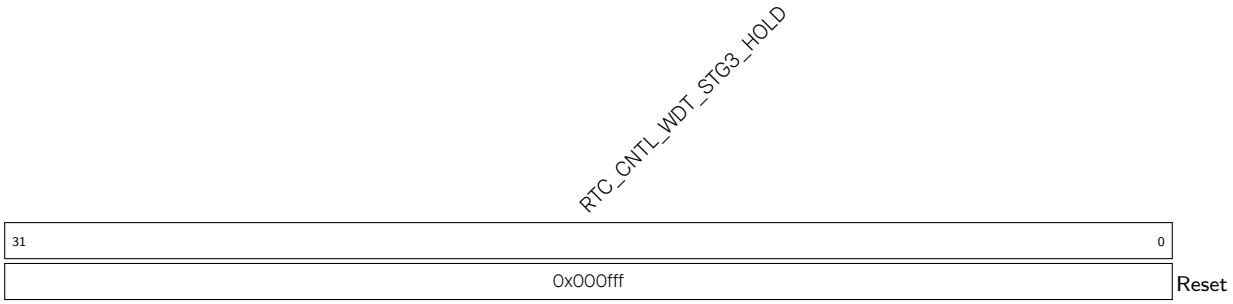
RTC_CNTL_WDT_STG1_HOLD 配置阶段 2 的 RTC 看门狗保持时间。(R/W)

Register 10.35. RTC_CNTL_RTC_WDTCONFIG3_REG (0x00A4)



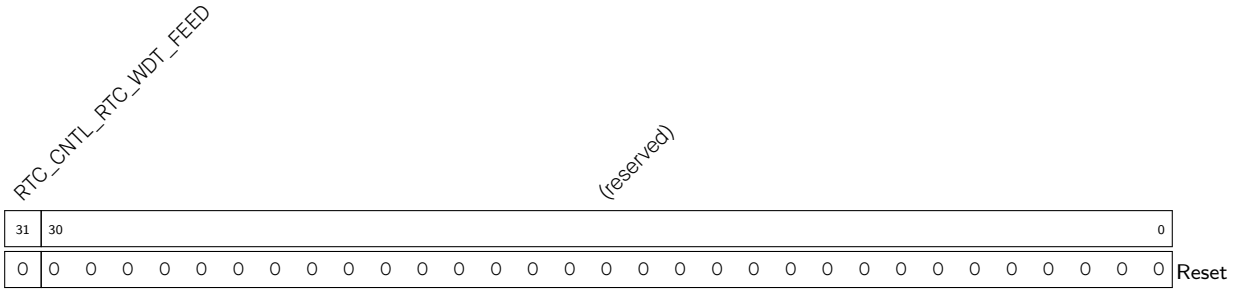
RTC_CNTL_WDT_STG2_HOLD 配置阶段 3 的 RTC 看门狗保持时间。(R/W)

Register 10.36. RTC_CNTL_RTC_WDTCONFIG4_REG (0x00A8)



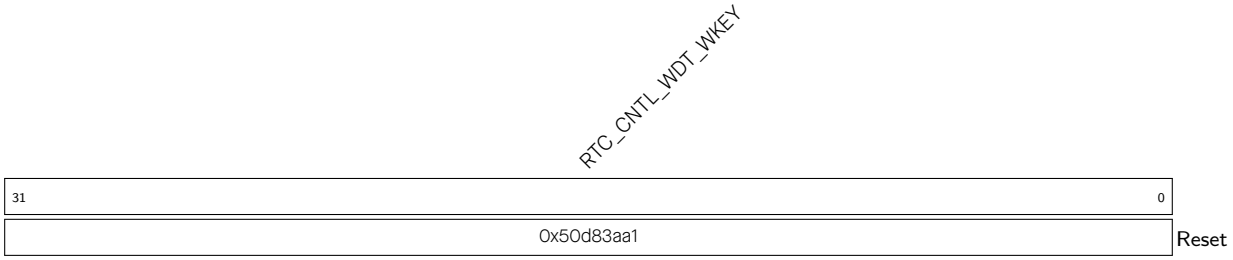
RTC_CNTL_WDT_STG3_HOLD 配置阶段 4 的 RTC 看门狗保持时间。(R/W)

Register 10.37. RTC_CNTL_RTC_WDTFEED_REG (0x00AC)



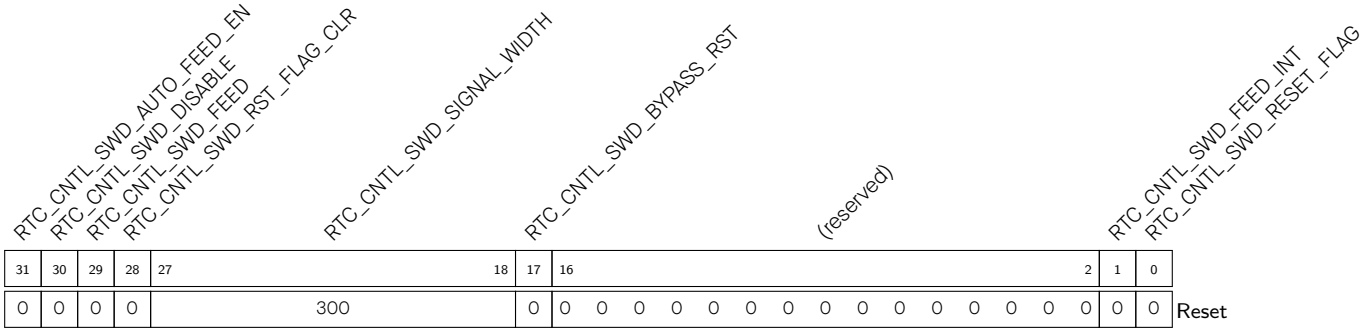
RTC_CNTL_RTC_WDT_FEED 置 1 开始喂 RTC 看门狗。(WO)

Register 10.38. RTC_CNTL_RTC_WDTWPROTECT_REG (0x00B0)



RTC_CNTL_WDT_WKEY 当该寄存器中的值不为 0x50d83aa1 时，使能 RTC 看门狗 (RWDT) 的写保护。(R/W)

Register 10.39. RTC_CNTL_RTC_SWD_CONF_REG (0x00B4)



RTC_CNTL_SWD_RESET_FLAG 代表超级看门狗的复位旗帜。(RO)

RTC_CNTL_SWD_FEED_INT 该中断产生则通过软件进行超级看门狗喂狗。(RO)

RTC_CNTL_SWD_BYPASS_RST 置 1 绕过超级看门狗复位。(R/W)

RTC_CNTL_SWD_SIGNAL_WIDTH 调节传递给超级看门狗的信号宽度。(R/W)

RTC_CNTL_SWD_RST_FLAG_CLR 置 1 清除超级看门狗复位旗帜。(WO)

RTC_CNTL_SWD_FEED 置 1 开始软件喂超级看门狗。(WO)

RTC_CNTL_SWD_DISABLE 置 1 禁用超级看门狗。(R/W)

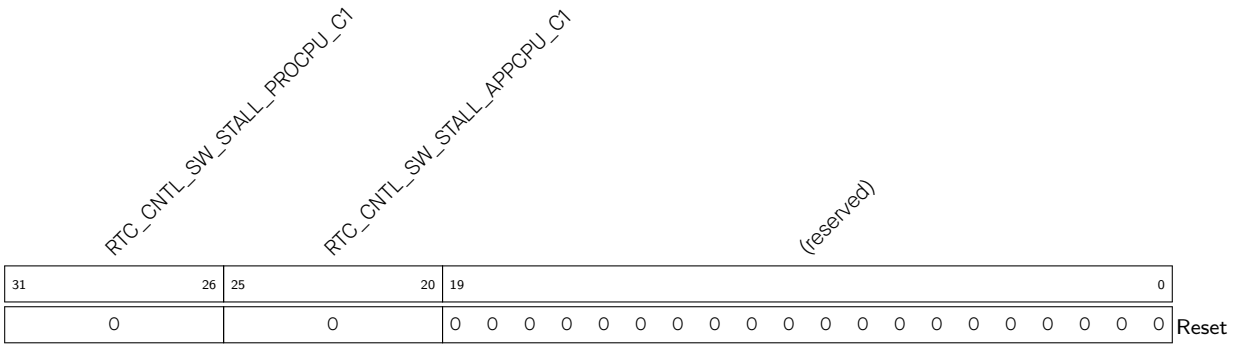
RTC_CNTL_SWD_AUTO_FEED_EN 置 1 使能在发生中断时自动喂狗。(R/W)

Register 10.40. RTC_CNTL_RTC_SWD_WPROTECT_REG (0x00B8)



RTC_CNTL_SWD_WKEY 设置超级看门狗的写保护密钥。(R/W)

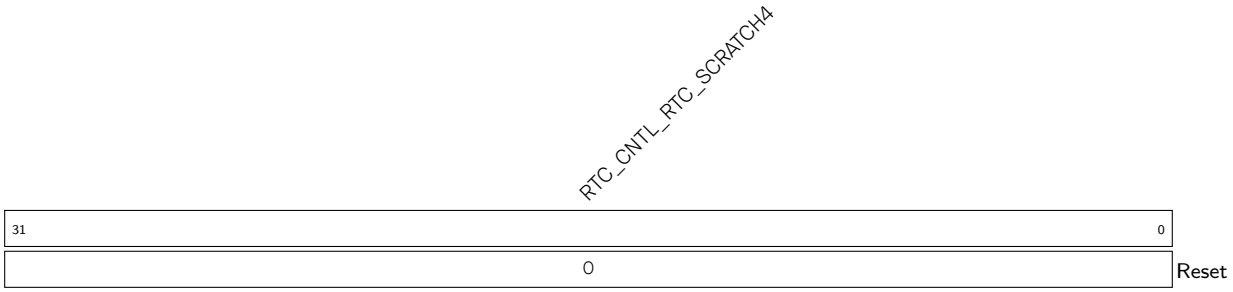
Register 10.41. RTC_CNTL_RTC_SW_CPU_STALL_REG (0x00BC)



RTC_CNTL_SW_STALL_APPCPU_C1 当 RTC_CNTL_SW_STALL_APPCPU_C0 配置为 0x2 时，设置该位为 0x21 将软件使 CPU1 进入 stall 状态。(R/W)

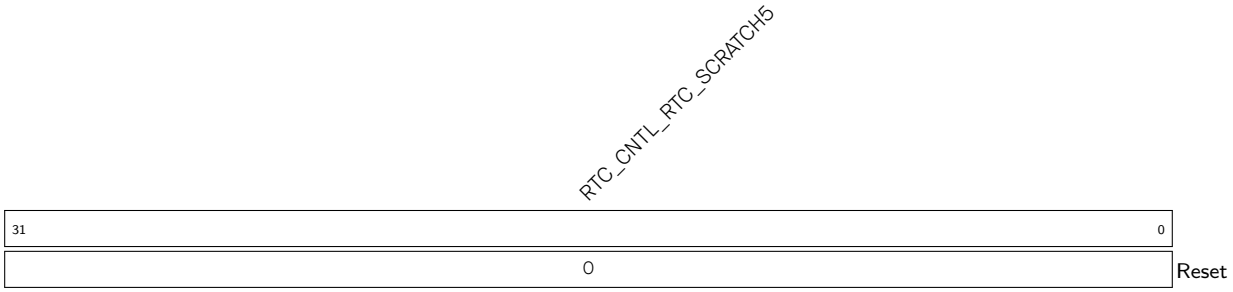
RTC_CNTL_SW_STALL_PROCPU_C1 当 RTC_CNTL_SW_STALL_PROCPU_C0 配置为 0x2 时，设置该位为 0x21 将软件使 CPU0 进入 stall 状态。(R/W)

Register 10.42. RTC_CNTL_RTC_STORE4_REG (0x00C0)



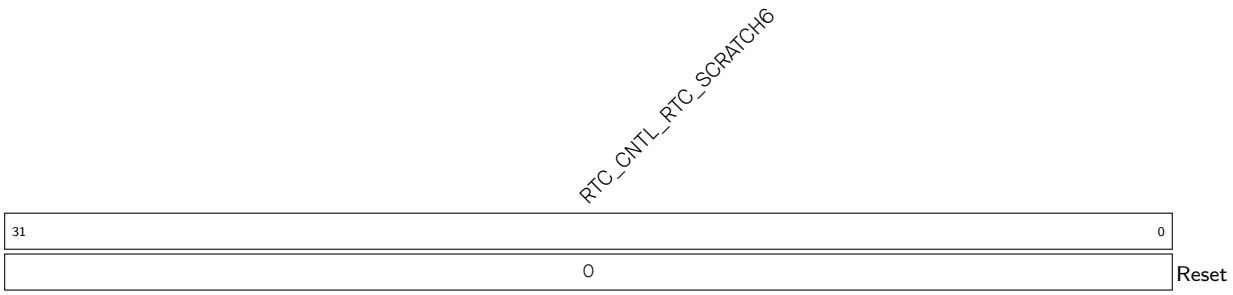
RTC_CNTL_RTC_SCRATCH4 Retention 寄存器 4。(R/W)

Register 10.43. RTC_CNTL_RTC_STORE5_REG (0x00C4)



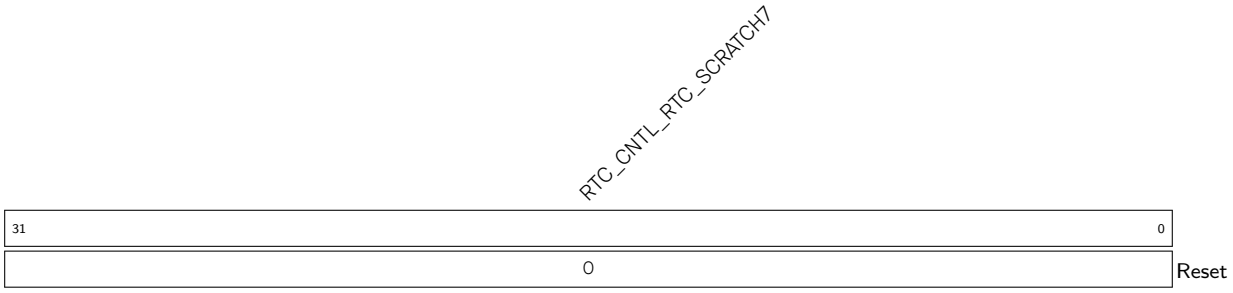
RTC_CNTL_RTC_SCRATCH5 Retention 寄存器 5。(R/W)

Register 10.44. RTC_CNTL_RTC_STORE6_REG (0x00C8)



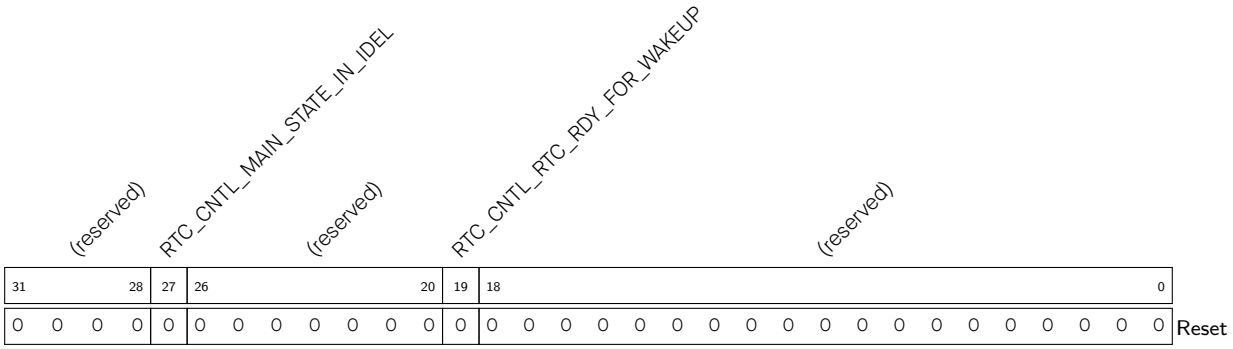
RTC_CNTL_RTC_SCRATCH6 Retention 寄存器 6。(R/W)

Register 10.45. RTC_CNTL_RTC_STORE7_REG (0x00CC)



RTC_CNTL_RTC_SCRATCH7 Retention 寄存器 7。(R/W)

Register 10.46. RTC_CNTL_RTC_LOW_POWER_ST_REG (0x00D0)

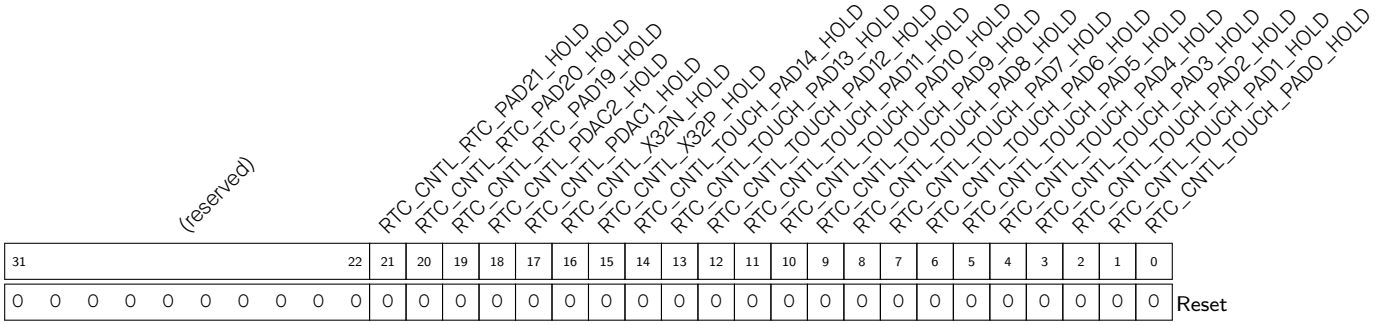


RTC_CNTL_RTC_RDY_FOR_WAKEUP 代表 RTC 已经准备好被任何唤醒源唤醒。(RO)

RTC_CNTL_MAIN_STATE_IN_IDLE 代表 RTC 状态。

- 0：芯片可能处于以下任一情况
  - 已经进入睡眠模式
  - 正在进入睡眠模式。此时，需等待 RTC_CNTL_RTC_RDY_FOR_WAKEUP 变为 1，然后可以唤醒芯片。
  - 正在退出睡眠模式。此时，RTC_CNTL_MAIN_STATE_IN_IDLE 终将变为 1。
- 1：芯片不处于睡眠模式（比如正常运行中）。

Register 10.47. RTC_CNTL_RTC_PAD_HOLD_REG (0x00D8)



RTC_CNTL_TOUCH_PAD0_HOLD 置 1 使触摸 GPIO 0 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD1_HOLD 置 1 使触摸 GPIO 1 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD2_HOLD 置 1 使触摸 GPIO 2 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD3_HOLD 置 1 使触摸 GPIO 3 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD4_HOLD 置 1 使触摸 GPIO 4 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD5_HOLD 置 1 使触摸 GPIO 5 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD6_HOLD 置 1 使触摸 GPIO 6 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD7_HOLD 置 1 使触摸 GPIO 7 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD8_HOLD 置 1 使触摸 GPIO 8 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD9_HOLD 置 1 使触摸 GPIO 9 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD10_HOLD 置 1 使触摸 GPIO 10 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD11_HOLD 置 1 使触摸 GPIO 11 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD12_HOLD 置 1 使触摸 GPIO 12 进入 hold 状态。(R/W)

接下页...

Register 10.47. RTC_CNTL_RTC_PAD_HOLD_REG (0x00D8)

接上页...

RTC_CNTL_TOUCH_PAD13_HOLD 置 1 使触摸 GPIO 13 进入 hold 状态。(R/W)

RTC_CNTL_TOUCH_PAD14_HOLD 置 1 使触摸 GPIO 14 进入 hold 状态。(R/W)

RTC_CNTL_X32P_HOLD 置 1 使 x32p 进入 hold 状态。(R/W)

RTC_CNTL_X32N_HOLD 置 1 使 x32n 进入 hold 状态。(R/W)

RTC_CNTL_PDAC1_HOLD 置 1 使 pdac1 进入 hold 状态。(R/W)

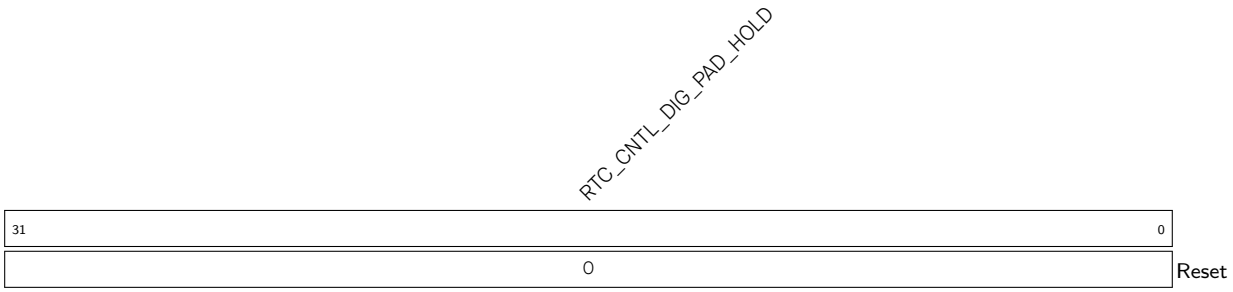
RTC_CNTL_PDAC2_HOLD 置 1 使 pdac2 进入 hold 状态。(R/W)

RTC_CNTL_RTC_PAD19_HOLD 置 1 使 RTG GPIO 19 进入 hold 状态。(R/W)

RTC_CNTL_RTC_PAD20_HOLD 置 1 使 RTG GPIO 20 进入 hold 状态。(R/W)

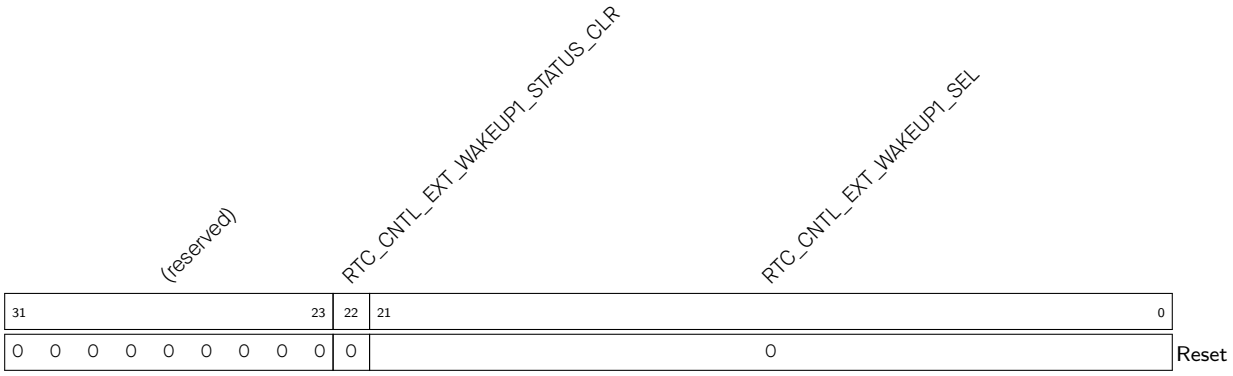
RTC_CNTL_RTC_PAD21_HOLD 置 1 使 RTG GPIO 21 进入 hold 状态。(R/W)

Register 10.48. RTC_CNTL_DIG_PAD_HOLD_REG (0x00DC)



RTC_CNTL_DIG_PAD_HOLD 置 1 使 GPIO 21 至 GPIO 45 进入 hold 状态。其中，GPIO 的位置可见芯片位图。(R/W)

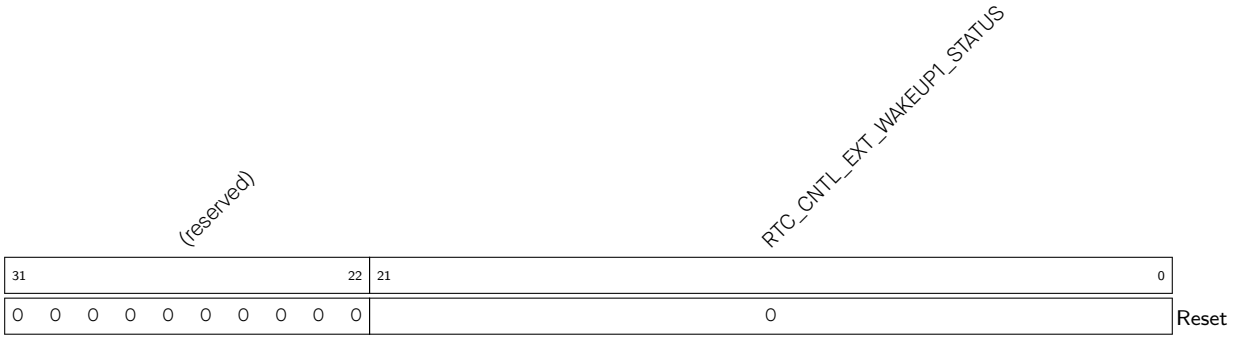
Register 10.49. RTC_CNTL_RTC_EXT_WAKEUP1_REG (0x00E0)



RTC_CNTL_EXT_WAKEUP1_SEL 置 1 使 RTC GPIO 作为 EXT1 唤醒源。(R/W)

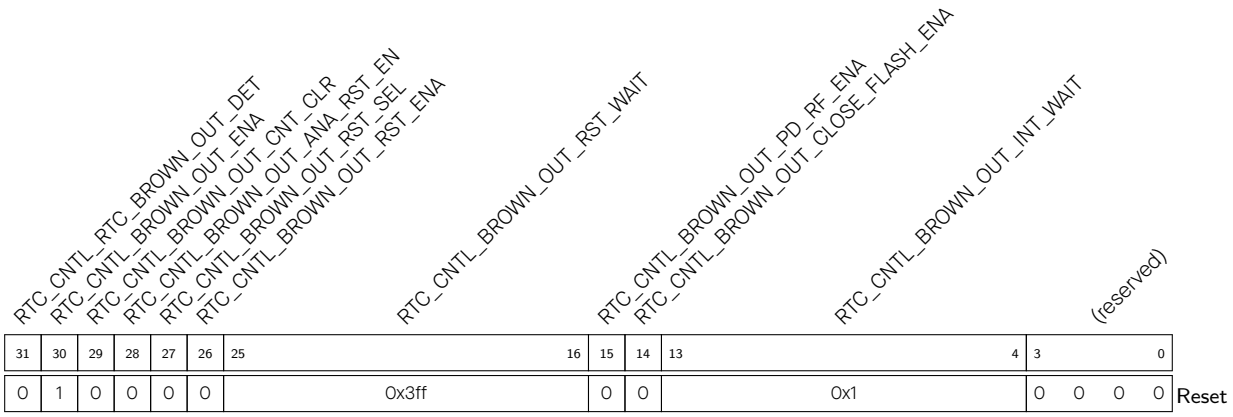
RTC_CNTL_EXT_WAKEUP1_STATUS_CLR 清除 EXT1 唤醒状态。(WO)

Register 10.50. RTC_CNTL_RTC_EXT_WAKEUP1_STATUS_REG (0x00E4)



RTC_CNTL_EXT_WAKEUP1_STATUS 指示 EXT1 唤醒源状态。(RO)

Register 10.51. RTC_CNTL_RTC_BROWN_OUT_REG (0x00E8)



RTC_CNTL_BROWN_OUT_INT_WAIT 配置发送欠压掉电置 1 使能在发生欠压掉电时强制关闭 flash。(R/W)

RTC_CNTL_BROWN_OUT_PD_RF_ENA 置 1 使能在发生欠压掉电前强制关闭 RF 电路。(R/W)

RTC_CNTL_BROWN_OUT_RST_WAIT 配置欠压掉电后芯片重启之前的等待周期。(R/W)

RTC_CNTL_BROWN_OUT_RST_ENA 置 1 使能欠压掉电复位。(R/W)

RTC_CNTL_BROWN_OUT_RST_SEL 选择欠压掉电复位方式。置 1 选择芯片复位，置 1 选择系统复位。(R/W)

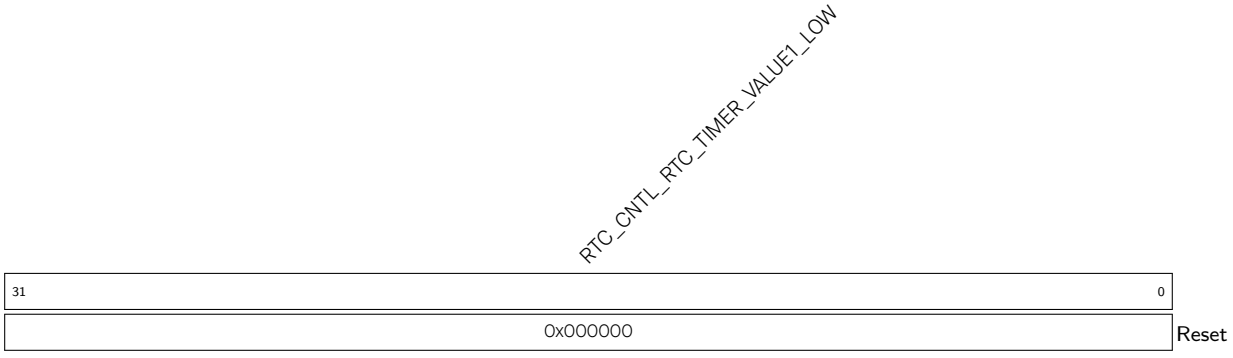
RTC_CNTL_BROWN_OUT_ANA_RST_EN 置 1 复位欠压掉电。(R/W)

RTC_CNTL_BROWN_OUT_CNT_CLR 清除欠压检测计数器。(WO)

RTC_CNTL_BROWN_OUT_ENA 置 1 使能欠压检测。(R/W)

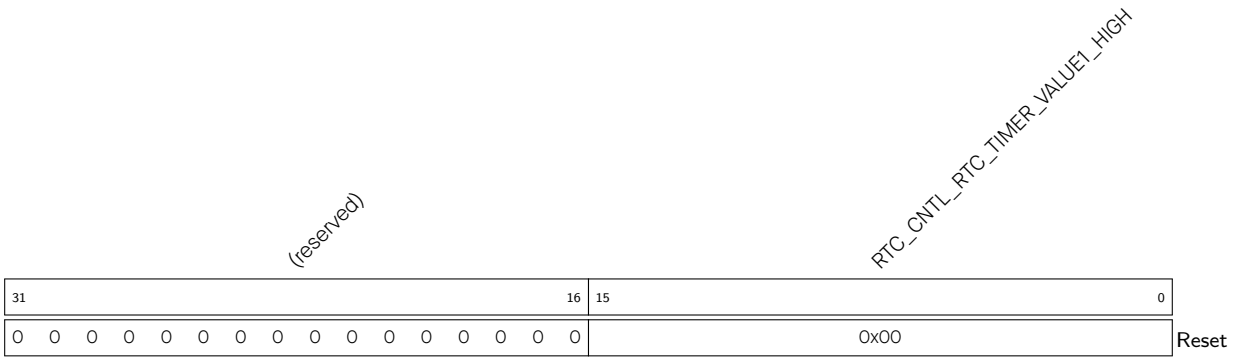
RTC_CNTL_RTC_BROWN_OUT_DET 指示欠压掉电信号状态。(RO)

Register 10.52. RTC_CNTL_RTC_TIME_LOW1_REG (0x00EC)



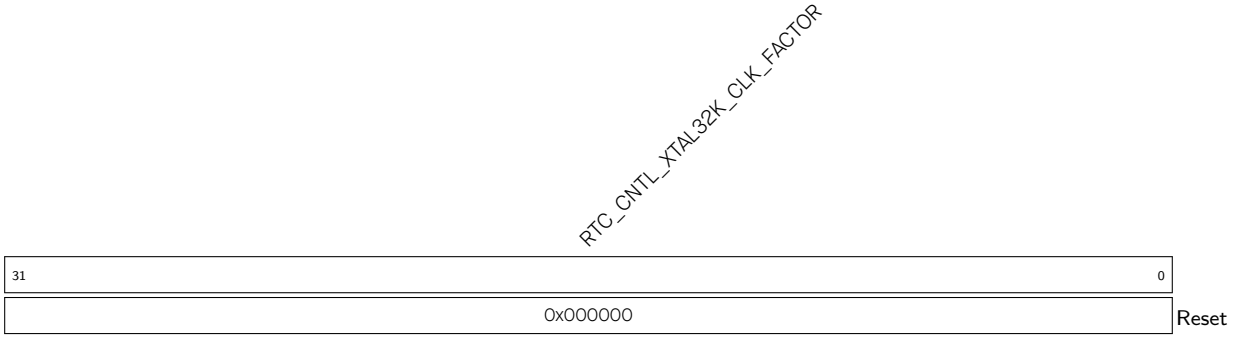
RTC_CNTL_RTC_TIMER_VALUE1_LOW 存储 RTC 计时器 1 的低 32 位。(RO)

Register 10.53. RTC_CNTL_RTC_TIME_HIGH1_REG (0x00F0)



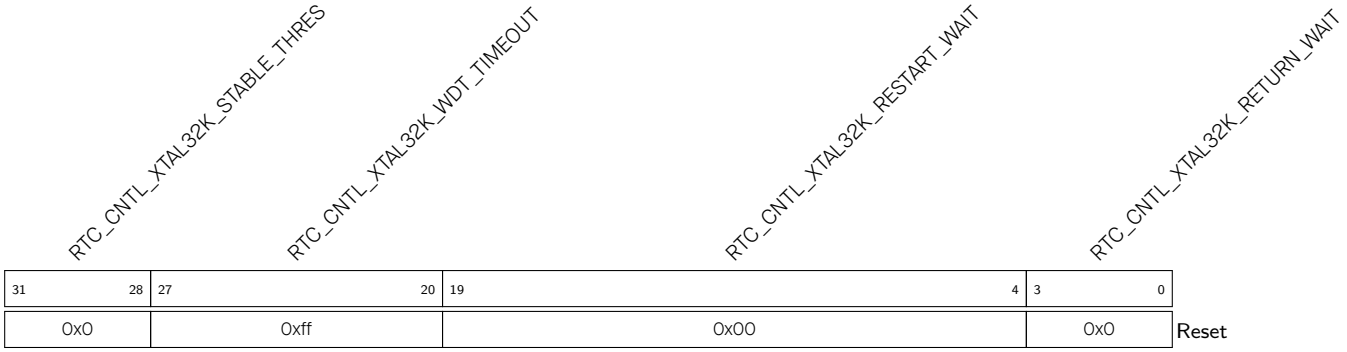
RTC_CNTL_RTC_TIMER_VALUE1_HIGH 存储 RTC 计时器 1 的高 16 位。(RO)

Register 10.54. RTC_CNTL_RTC_XTAL32K_CLK_FACTOR_REG (0x00F4)



RTC_CNTL_XTAL32K_CLK_FACTOR 设置 32 kHz 晶振的分频器系数。(R/W)

Register 10.55. RTC_CNTL_RTC_XTAL32K_CONF_REG (0x00F8)



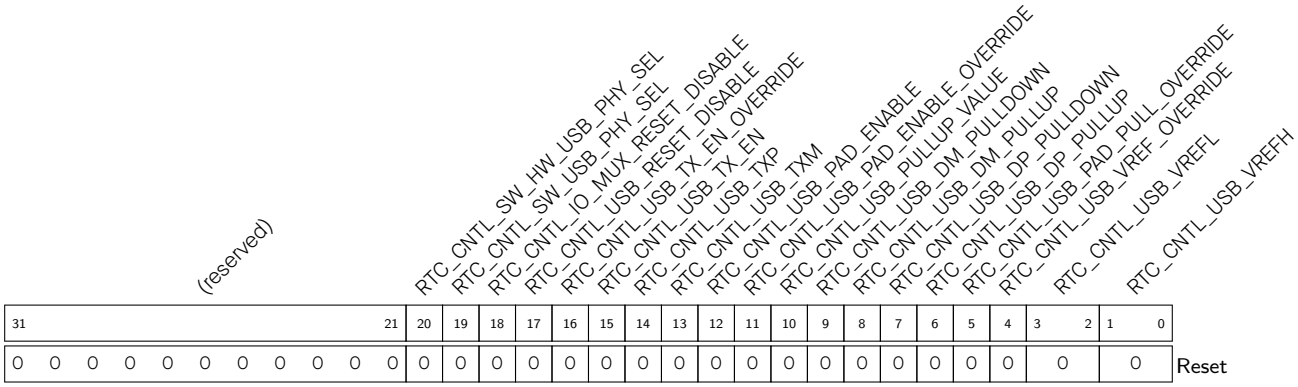
RTC_CNTL_XTAL32K_RETURN_WAIT 设置切换回 32 kHz 晶振之前的等待周期。(R/W)

RTC_CNTL_XTAL32K_RESTART_WAIT 设置重启 32 kHz 晶振之前的等待周期。(R/W)

RTC_CNTL_XTAL32K_WDT_TIMEOUT 设置时钟检测的等待周期。如果超过该周期后仍未检测到时钟，则视为 32 kHz 晶振掉电。(R/W)

RTC_CNTL_XTAL32K_STABLE_THRES 设置最大重启周期。如果 32 kHz 晶振可以在该周期内完成掉电重启，则视为 32 kHz 晶振稳定。(R/W)

Register 10.56. RTC_CNTL_RTC_USB_CONF_REG (0x0120)



RTC_CNTL_USB_VREFH 控制内部 USB 接收器单端输入高阈值（1.76 V 到 2 V，步进为 80 mV。本字段仅在 RTC_CNTL_USB_VREF_OVERRIDE 配置时生效。(R/W)

RTC_CNTL_USB_VREFL 控制内部 USB 接收器单端输入低阈值（0.8 V 到 1.04 V，步进为 80 mV）。本字段仅在 RTC_CNTL_USB_VREF_OVERRIDE 配置时生效。(R/W)

RTC_CNTL_USB_VREF_OVERRIDE 置 1 允许软件控制内部 USB 接收器输入电压阈值。(R/W)

RTC_CNTL_USB_PAD_PULL_OVERRIDE 置 1 允许软件控制内部 USB 接收器的 USB D+/D- 上拉和下拉电阻。(R/W)

RTC_CNTL_USB_DP_PULLUP 置 1 使能 USB+ 上拉电阻。本字段仅在 RTC_CNTL_USB_PAD_PULL_OVERRIDE 配置时生效。(R/W)

RTC_CNTL_USB_DP_PULLDOWN 置 1 使能 USB+ 下拉电阻。本字段仅在 RTC_CNTL_USB_PAD_PULL_OVERRIDE 配置时生效。(R/W)

RTC_CNTL_USB_DM_PULLUP 置 1 使能 USB- 下拉电阻。本字段仅在 RTC_CNTL_USB_PAD_PULL_OVERRIDE 配置时生效。(R/W)

RTC_CNTL_USB_DM_PULLDOWN 置 1 使能 USB- 下拉电阻。本字段仅在 RTC_CNTL_USB_PAD_PULL_OVERRIDE 配置时生效。(R/W)

RTC_CNTL_USB_PULLUP_VALUE 控制上拉电阻值。0：典型值 2.4 kΩ；1：典型值 1.2 kΩ。本字段仅在 RTC_CNTL_USB_PAD_PULL_OVERRIDE 配置时生效。(R/W)

接下页...

Register 10.56. RTC_CNTL_RTC_USB_CONF_REG (0x0120)

接上页...

RTC_CNTL_USB_PAD_ENABLE_OVERRIDE 置 1 允许软件控制内部 USB 接收器功能。(R/W)

RTC_CNTL_USB_PAD_ENABLE 置 1 使能 USB 接收器功能。本字段仅在 RTC_CNTL_USB_PAD_ENABLE_OVERRIDE 配置时生效。(R/W)

RTC_CNTL_USB_TXM 配置测试模式下 USB D- 的 TX 值。本字段仅在 RTC_CNTL_USB_TX_EN_OVERRIDE 时生效。(R/W)

RTC_CNTL_USB_TXP 配置测试模式下 USB D+ 的 TX 值。本字段仅在 RTC_CNTL_USB_TX_EN_OVERRIDE 时生效。(R/W)

RTC_CNTL_USB_TX_EN 配置测试模式下 USB pad oen。本字段仅在 RTC_CNTL_USB_TX_EN_OVERRIDE 时生效。(R/W)

RTC_CNTL_USB_TX_EN_OVERRIDE 置 1 允许软件控制测试模式下的内部 USB 接收器 TX 值。(R/W)

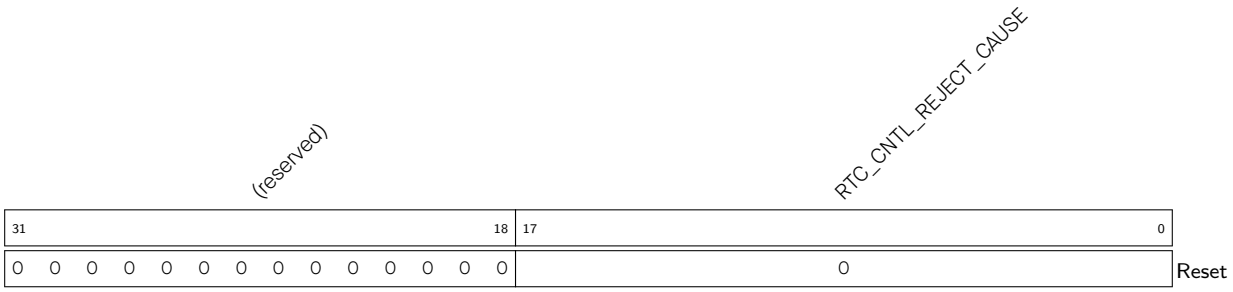
RTC_CNTL_USB_RESET_DISABLE 置 1 禁止复位 USB OTG。(R/W)

RTC_CNTL_IO_MUX_RESET_DISABLE 置 1 禁止复位 IO MUX 和 GPIO 矩阵。(R/W)

RTC_CNTL_SW_USB_PHY_SEL 置 1 允许 USB OTG 使用内部 USB 接收器。清 0 允许 USB-Serial-JTAG 使用内部 USB 接收器。本字段仅在 RTC_CNTL_SW_HW_USB_PHY_SEL 时生效。(R/W)

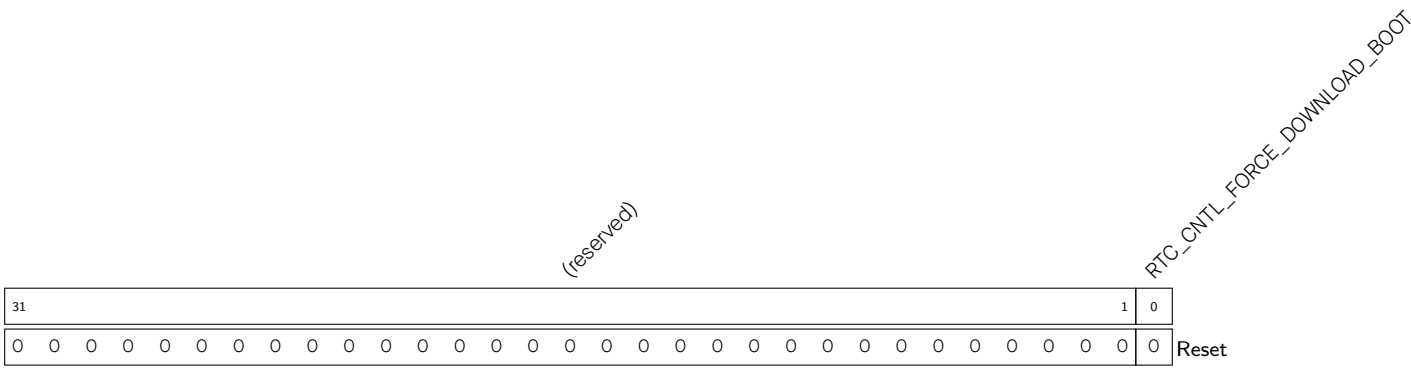
RTC_CNTL_SW_HW_USB_PHY_SEL 置 1 软件分配内部 USB 接收器的使用 (RTC_CNTL_SW_USB_PHY_SEL)；清 0 硬件分配内部 USB 接收器的使用。(R/W)

Register 10.57. RTC_CNTL_RTC_SLP_REJECT_CAUSE_REG (0x0128)



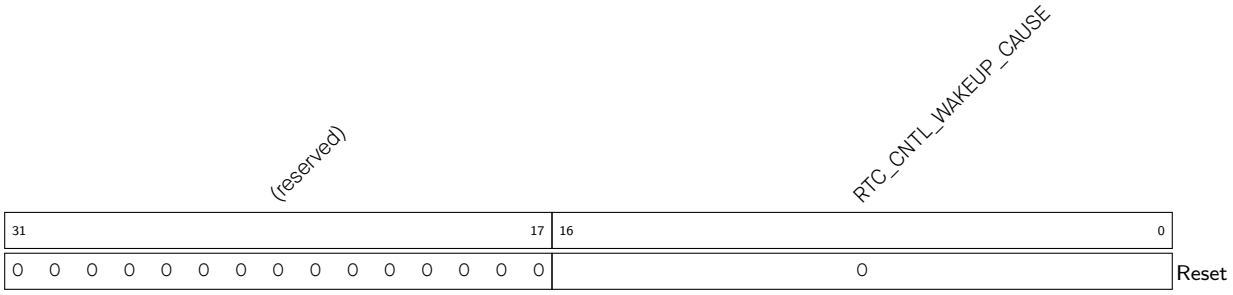
RTC_CNTL_REJECT_CAUSE 存储“拒绝入睡”的原因。(RO)

Register 10.58. RTC_CNTL_RTC_OPTION1_REG (0x012C)



RTC_CNTL_FORCE_DOWNLOAD_BOOT 置 1 强制芯片从下载模式启动。(R/W)

Register 10.59. RTC_CNTL_RTC_SLP_WAKEUP_CAUSE_REG (0x0130)



RTC_CNTL_WAKEUP_CAUSE 存储唤醒原因。(RO)

Register 10.60. RTC_CNTL_INT_ENA_RTC_W1TS_REG (0x0138)

Bit	Field	Reset
31:21	(reserved)	0 0 0 0 0 0 0 0 0 0 0
20	RTC_CNTL_RTC_TOUCH_APPROACH_LOOP_DONE_INT_ENA_W1TS	0
19	RTC_CNTL_RTC_GLITCH_DET_INT_ENA_W1TS	0
18	RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_ENA_W1TS	0
17	RTC_CNTL_RTC_COCPU_TRAP_INT_ENA_W1TS	0
16	RTC_CNTL_RTC_XTAL32K_DEAD_INT_ENA_W1TS	0
15	RTC_CNTL_RTC_SWD_INT_ENA_W1TS	0
14	RTC_CNTL_RTC_SARADC2_INT_ENA_W1TS	0
13	RTC_CNTL_RTC_COCPU_INT_ENA_W1TS	0
12	RTC_CNTL_RTC_TSENS_INT_ENA_W1TS	0
11	RTC_CNTL_RTC_SARADC1_INT_ENA_W1TS	0
10	RTC_CNTL_RTC_MAIN_TIMER_INT_ENA_W1TS	0
9	RTC_CNTL_RTC_BROWN_OUT_INT_ENA_W1TS	0
8	RTC_CNTL_RTC_TOUCH_INACTIVE_INT_ENA_W1TS	0
7	RTC_CNTL_RTC_TOUCH_ACTIVE_INT_ENA_W1TS	0
6	RTC_CNTL_RTC_TOUCH_DONE_INT_ENA_W1TS	0
5	RTC_CNTL_RTC_ULP_CP_INT_ENA_W1TS	0
4	RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_ENA_W1TS	0
3	RTC_CNTL_RTC_WDT_INT_ENA_W1TS	0
2	RTC_CNTL_SDIO_IDLE_INT_ENA_W1TS	0
1	RTC_CNTL_SLP_REJECT_INT_ENA_W1TS	0
0	RTC_CNTL_SLP_WAKEUP_INT_ENA_W1TS	0

RTC_CNTL_SLP_WAKEUP_INT_ENA_W1TS 使能在芯片“从睡眠中唤醒”时发送中断。此位一旦置 1，则 RTC_CNTL_SLP_WAKEUP_INT_ENA 也将置 1。(WO)

RTC_CNTL_SLP_REJECT_INT_ENA_W1TS 使能在芯片“拒绝从睡眠中唤醒”时发送中断。此位一旦置 1，则 RTC_CNTL_SLP_REJECT_INT_ENA 也将置 1。(WO)

RTC_CNTL_SDIO_IDLE_INT_ENA_W1TS 使能在 SDIO idle 时发送中断。此位一旦置 1，则 RTC_CNTL_SDIO_IDLE_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_WDT_INT_ENA_W1TS 使能 RTC 看门狗中断。此位一旦置 1，则 RTC_CNTL_RTC_WDT_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_ENA_W1TS 使能在触摸扫描完成时发送中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_ULP_CP_INT_ENA_W1TS 使能超低功耗协处理器中断。此位一旦置 1，则 RTC_CNTL_RTC_ULP_CP_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_TOUCH_DONE_INT_ENA_W1TS 使能在单次触摸完成时发送中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_DONE_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_TOUCH_ACTIVE_INT_ENA_W1TS 使能在检测到触摸时发送中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_ACTIVE_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_TOUCH_INACTIVE_INT_ENA_W1TS 使能在触摸释放时发送中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_ACTIVE_INT_ENA 也将置 1。(WO)

接下页...

Register 10.60. RTC_CNTL_INT_ENA_RTC_W1TS_REG (0x0138)

接上页...

RTC_CNTL_RTC_BROWN_OUT_INT_ENA_W1TS 使能欠压检测中断。此位一旦置 1，则 RTC_CNTL_RTC_BROWN_OUT_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_MAIN_TIMER_INT_ENA_W1TS 使能 RTC 主计时器中断。此位一旦置 1，则 RTC_CNTL_RTC_MAIN_TIMER_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_SARADC1_INT_ENA_W1TS 使能 SAR ADC1 中断。此位一旦置 1，则 RTC_CNTL_RTC_SARADC1_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_TSENS_INT_ENA_W1TS 使能温度传感器中断。此位一旦置 1，则 RTC_CNTL_RTC_TSENS_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_COCPU_INT_ENA_W1TS 使能 ULP-RISCV 中断。此位一旦置 1，则 RTC_CNTL_RTC_COCPU_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_SARADC2_INT_ENA_W1TS 使能 SAR ADC2 中断。此位一旦置 1，则 RTC_CNTL_RTC_SARADC2_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_SWD_INT_ENA_W1TS 使能超级看门狗中断。此位一旦置 1，则 RTC_CNTL_RTC_SWD_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_XTAL32K_DEAD_INT_ENA_W1TS 使能在 32 kHz 晶振掉电时发送中断。此位一旦置 1，则 RTC_CNTL_RTC_XTAL32K_DEAD_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_COCPU_TRAP_INT_ENA_W1TS 使能在 ULP-RISCV 被困时发送中断。此位一旦置 1，则 RTC_CNTL_RTC_COCPU_TRAP_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_ENA_W1TS 使能在触摸传感器超时时发送中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_GLITCH_DET_INT_ENA_W1TS 使能在触摸传感器超时时发送中断。此位一旦置 1，则 RTC_CNTL_RTC_GLITCH_DET_INT_ENA 也将置 1。(WO)

RTC_CNTL_RTC_TOUCH_APPROACH_LOOP_DONE_INT_ENA_W1TS 使能在触摸循环完成后发送中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_APPROACH_LOOP_DONE_INT_ENA 也将置 1。(WO)

Register 10.61. RTC_CNTL_INT_ENA_RTC_W1TC_REG (0x013C)

Bits	Field	Reset
31:21	(reserved)	0 0 0 0 0 0 0 0 0 0 0
20	RTC_CNTL_RTC_TOUCH_APPROACH_LOOP_DONE_INT_ENA_W1TC	0
19	RTC_CNTL_RTC_GLITCH_DET_INT_ENA_W1TC	0
18	RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_ENA_W1TC	0
17	RTC_CNTL_RTC_COCPU_TRAP_INT_ENA_W1TC	0
16	RTC_CNTL_RTC_XTAL32K_DEAD_INT_ENA_W1TC	0
15	RTC_CNTL_RTC_SWD_INT_ENA_W1TC	0
14	RTC_CNTL_RTC_SARADC2_INT_ENA_W1TC	0
13	RTC_CNTL_RTC_COCPU_INT_ENA_W1TC	0
12	RTC_CNTL_RTC_TSENS_INT_ENA_W1TC	0
11	RTC_CNTL_RTC_SARADC1_INT_ENA_W1TC	0
10	RTC_CNTL_RTC_MAIN_TIMER_INT_ENA_W1TC	0
9	RTC_CNTL_RTC_BROWN_OUT_INT_ENA_W1TC	0
8	RTC_CNTL_RTC_TOUCH_INACTIVE_INT_ENA_W1TC	0
7	RTC_CNTL_RTC_TOUCH_ACTIVE_INT_ENA_W1TC	0
6	RTC_CNTL_RTC_TOUCH_DONE_INT_ENA_W1TC	0
5	RTC_CNTL_RTC_ULP_CP_INT_ENA_W1TC	0
4	RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_ENA_W1TC	0
3	RTC_CNTL_RTC_WDT_INT_ENA_W1TC	0
2	RTC_CNTL_SDIO_IDLE_INT_ENA_W1TC	0
1	RTC_CNTL_SLP_REJECT_INT_ENA_W1TC	0
0	RTC_CNTL_SLP_WAKEUP_INT_ENA_W1TC	0

RTC_CNTL_SLP_WAKEUP_INT_ENA_W1TC 清除芯片"从睡眠中唤醒"时中断。此位一旦置 1，则 RTC_CNTL_SLP_WAKEUP_INT_CLR 将清零。(WO)

RTC_CNTL_SLP_REJECT_INT_ENA_W1TC 清除芯片"拒绝入睡"时中断。此位一旦置 1，则 RTC_CNTL_SLP_REJECT_INT_CLR 将清零。(WO)

RTC_CNTL_SDIO_IDLE_INT_ENA_W1TC 清除 SDIO idle 中断。此位一旦置 1，则 RTC_CNTL_SDIO_IDLE_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_WDT_INT_ENA_W1TC 清除 RTC 看门狗中断。此位一旦置 1，则 RTC_CNTL_RTC_WDT_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_ENA_W1TC 清除存储触摸完成时中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_ULP_CP_INT_ENA_W1TC 清除超低功耗协处理器中断。此位一旦置 1，则 RTC_CNTL_RTC_ULP_CP_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_TOUCH_DONE_INT_ENA_W1TC 清除单次触摸完成时中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_DONE_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_TOUCH_ACTIVE_INT_ENA_W1TC 清除检测到触摸时中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_ACTIVE_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_TOUCH_INACTIVE_INT_ENA_W1TC 清除触摸释放中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_INACTIVE_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_BROWN_OUT_INT_ENA_W1TC 清除欠压检测中断。此位一旦置 1，则 RTC_CNTL_RTC_BROWN_OUT_INT_CLR 将清零。(WO)

接下页...

Register 10.61. RTC_CNTL_INT_ENA_RTC_W1TC_REG (0x013C)

接上页...

RTC_CNTL_RTC_MAIN_TIMER_INT_ENA_W1TC 清除 RTC 主定时器中断。此位一旦置 1，则 RTC_CNTL_RTC_MAIN_TIMER_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_SARADC1_INT_ENA_W1TC 清除 SAR ADC1 中断。此位一旦置 1，则 RTC_CNTL_RTC_SARADC1_INT_ENA_CLR 将清零。(WO)

RTC_CNTL_RTC_TSENS_INT_ENA_W1TC 清除温度传感器中断。此位一旦置 1，则 RTC_CNTL_RTC_TSENS_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_COCPU_INT_ENA_W1TC 清除 ULP-RISCV 中断。此位一旦置 1，则 RTC_CNTL_RTC_COCPU_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_SARADC2_INT_ENA_W1TC 清除 SAR ADC2 中断。此位一旦置 1，则 RTC_CNTL_RTC_SARADC2_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_SWD_INT_ENA_W1TC 清除超级看门狗中断。此位一旦置 1，则 RTC_CNTL_RTC_SWD_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_XTAL32K_DEAD_INT_ENA_W1TC 清除在 32 kHz 晶振掉电时发送中断。此位一旦置 1，则 RTC_CNTL_RTC_XTAL32K_DEAD_INT_CLR 将清零。(WO)

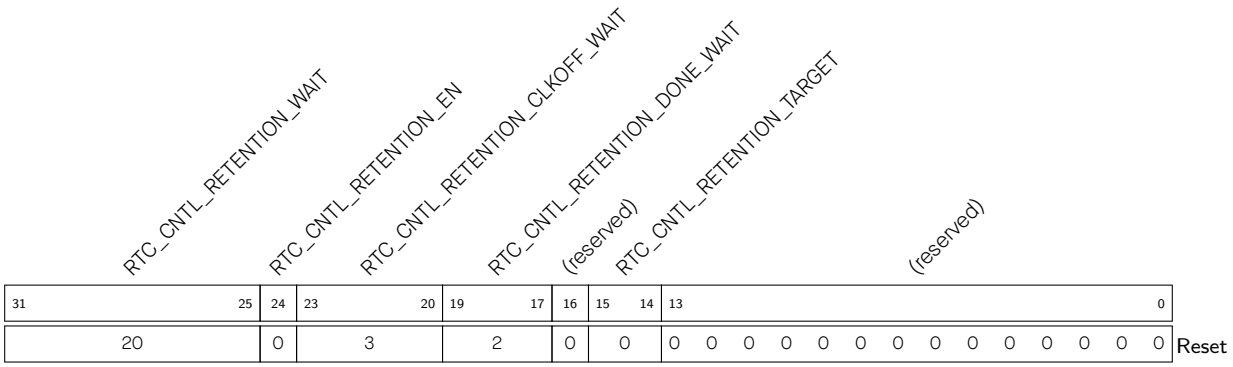
RTC_CNTL_RTC_COCPU_TRAP_INT_ENA_W1TC 清除 ULP-RISCV 受困时中断。此位一旦置 1，则 RTC_CNTL_RTC_COCPU_TRAP_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_ENA_W1TC 清除触摸传感器超时中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_GLITCH_DET_INT_ENA_W1TC 清除检测到脉冲毛刺时中断。此位一旦置 1，则 RTC_CNTL_RTC_GLITCH_DET_INT_CLR 将清零。(WO)

RTC_CNTL_RTC_TOUCH_APPROACH_LOOP_DONE_INT_ENA_W1TC 清除触摸循环完成后中断。此位一旦置 1，则 RTC_CNTL_RTC_TOUCH_APPROACH_LOOP_DONE_INT_CLR 将清零。(WO)

Register 10.62. RTC_CNTL_RETENTION_CTRL_REG (0x0140)



RTC_CNTL_RETENTION_TARGET 配置 retention 目标：CPU 和/或 TAG。(R/W)

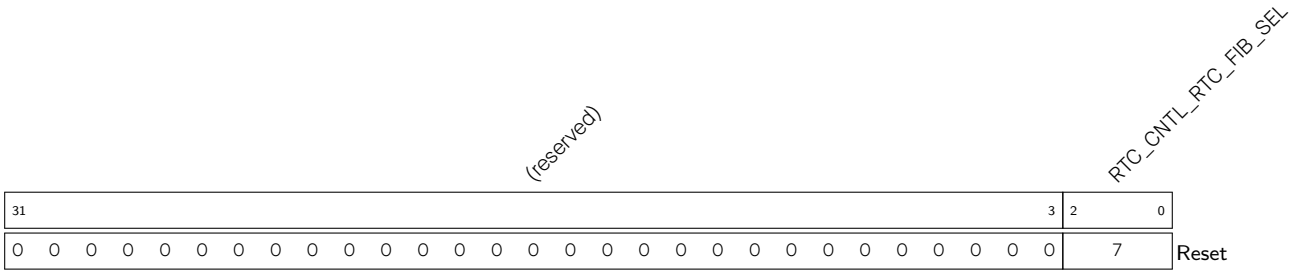
RTC_CNTL_RETENTION_DONE_WAIT 配置 retention 完成前的等待时间。(R/W)

RTC_CNTL_RETENTION_CLKOFF_WAIT 配置 clk_off 前的等待时间。(R/W)

RTC_CNTL_RETENTION_EN 置 1 使能 retention 功能。(R/W)

RTC_CNTL_RETENTION_WAIT 配置 retention 操作的等待周期。(R/W)

Register 10.63. RTC_CNTL_RTC_FIB_SEL_REG (0x0148)



RTC_CNTL_RTC_FIB_SEL 配置欠压检测器。(R/W)

第 11 章

系统定时器 (SYSTIMER)

11.1 概述

ESP32-S3 芯片内置一组 52 位系统定时器。该定时器可用于生成操作系统所需的滴答定时中断，也可用作普通定时器生成周期或单次延时中断。

系统定时器内置两个计数器（UNIT0 和 UNIT1）以及三个比较器（COMP 0、COMP1 和 COMP2）。比较器用于监控计数器的计数值是否达到报警值。定时器的功能块图见图 11.1-1。

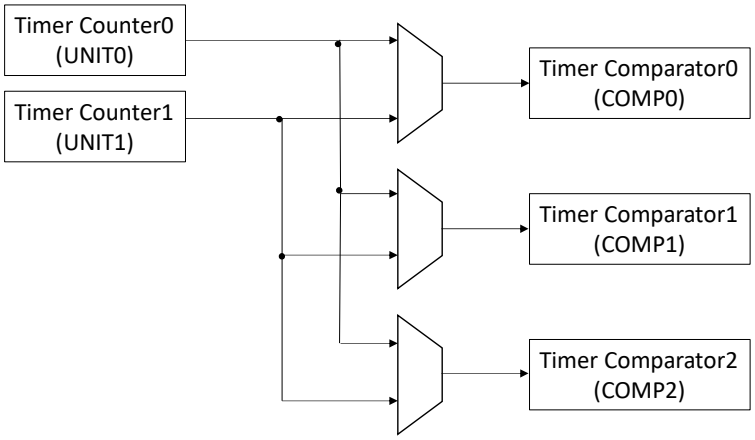


图 11.1-1. 系统定时器结构图

11.2 特性

- 由两个 52 位计数器和三个 52 位比较器组成
- 软件操作使用 APB_CLK 时钟
- 计数采用 CNT_CLK 时钟，两次计数周期的平均频率为 16 MHz
- CNT_CLK 的时钟源为 XTAL_CLK (40 MHz)
- 支持 52 位报警值 (t) 和 26 位报警周期 (δt)
- 支持两种报警模式：
  - 单次报警模式：根据设定的目标报警值 (t)，生成一次性报警
  - 周期报警模式：根据设定的报警周期 (δt)，生成周期性报警
- 三个比较器可根据设置的报警值 (t) 或报警周期 (δt) 生成三个独立中断

- 支持软件配置基准计数值。例如，支持从 Light-sleep 唤醒之后，系统定时器通过软件加载 RTC 定时器记录的睡眠时间，并进行补偿
- CPU 处于停止状态或处于在线调试状态时，系统定时器可选择停止运行或继续运行

11.3 时钟源选择

计数器和比较器使用 XTAL_CLK 用作时钟源。XTAL_CLK 经分数分频后，在一个计数周期生成频率为 $f_{XTAL\_CLK}/3$ 的时钟信号，然后在另一个计数周期生成频率为 $f_{XTAL\_CLK}/2$ 的时钟信号。因此，计数器使用的时钟 CNT_CLK，其实际平均频率为 $f_{XTAL\_CLK}/2.5$，即 16 MHz，见图 11.4-1。每个 CNT_CLK 时钟周期，计数递增 1/16 μs，即 16 个周期递增 1 μs。

配置寄存器等软件操作则是由 APB_CLK 提供时钟信号。更多有关 APB_CLK 的信息，见章节 7 复位和时钟。

用户可使用以下系统寄存器的相关位来控制系统定时器：

- 置位寄存器 SYSTEM_PERIP_CLK_ENO_REG 中 SYSTEM_SYSTIMER_CLK_EN 位使能系统定时器的 APB_CLK 信号；
- 置位寄存器 SYSTEM_PERIP_CLK_ENO_REG 中 SYSTEM_SYSTIMER_RST 位，复位系统定时器。

注意，复位后，系统定时器的寄存器将恢复到默认值。更多信息可参考章节 17 系统寄存器 (SYSTEM) 中表：外设时钟门控与复位控制位。

11.4 功能描述

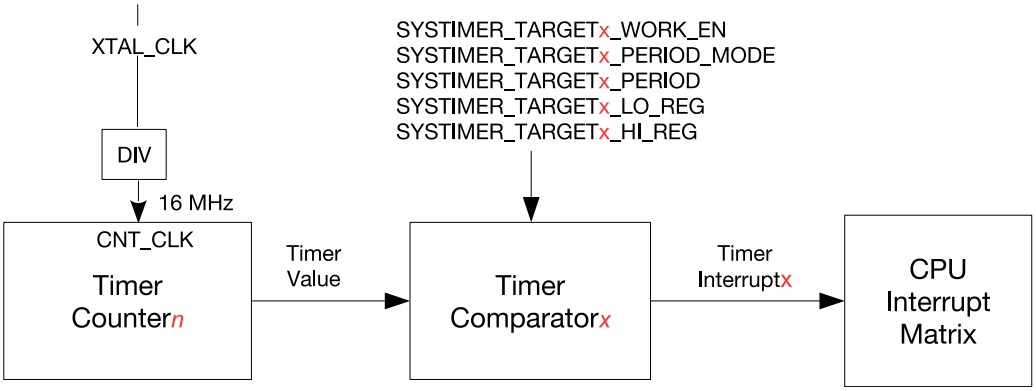


图 11.4-1. 系统定时器生成报警

图 11.4-1 展示了系统定时器生成报警的过程。在上述过程中用到一个计数器和一个比较器，比较器将根据比较结果，生成报警中断。

11.4.1 计数器

系统定时器提供两个 52 位计数器，下文用 UNIT*n* 表示，*n* 可以取 0 或 1。计数器使用 16 MHz CNT_CLK 作为计数时钟。用户可通过配置寄存器 SYSTIMER_CONF_REG 中下面三个位来控制计数器 UNIT*n*：

- SYSTIMER_TIMER_UNIT*n*_WORK_EN：置位此位，使能计数器 UNIT*n*；

- SYSTIMER_TIMER_UNIT*n*_CORE0_STALL_EN：置位此位，CPU0 停止运行后，计数器 UNIT*n* 将暂停计数，但 CPU0 苏醒后，UNIT*n* 将继续计数；
- SYSTIMER_TIMER_UNIT*n*_CORE1_STALL_EN：置位此位，CPU1 停止运行后，计数器 UNIT*n* 将暂停计数，但 CPU1 苏醒后，UNIT*n* 将继续计数。

UNIT*n* 的具体配置见下表，其中假设 CPU0 和 CPU1 当前状态为停止工作。

表 11.4-1. UNIT*n* 配置控制位

SYSTIMER_TIMER_UNIT*n*_WORK_EN	SYSTIMER_TIMER_UNIT*n*_CORE0_STALL_EN	SYSTIMER_TIMER_UNIT*n*_CORE1_STALL_EN	计数器 UNIT*n*
0	x *	x	未处于工作状态
1	x	1	暂停计数，但 CPU1 苏醒后，会继续计数。
1	1	x	暂停计数，但 CPU0 苏醒后，会继续计数。
1	0	0	不受影响，照常计数。

* x：无关项

计数器 UNIT*n* 处于工作状态时，计数值按计数周期递增。UNIT*n* 停止工作或暂停工作，则计数值将保持不变，不再递增。

计数起始值的低 32 位和高 20 位分别从 SYSTIMER_TIMER_UNIT*n*_LOAD_LO 和 SYSTIMER_TIMER_UNIT*n*_LOAD_HI 装载。置位 SYSTIMER_TIMER_UNIT*n*_LOAD 将触发重装载事件，当前计数起始值立即更新。如果计数器 UNIT*n* 处于工作状态，则将从新装载的计数值开始计数。

置位 SYSTIMER_TIMER_UNIT*n*_UPDATE 将触发更新事件，当前计数值的低 32 位和高 20 位被锁存至寄存器 SYSTIMER_TIMER_UNIT*n*_VALUE_LO 和 SYSTIMER_TIMER_UNIT*n*_VALUE_HI 后，SYSTIMER_TIMER_UNIT*n*_VALUE_VALID 会被置起。SYSTIMER_TIMER_UNIT*n*_VALUE_LO 和 SYSTIMER_TIMER_UNIT*n*_VALUE_HI 寄存器中的值保持不变，直至下次更新事件发生。

11.4.2 比较器和报警

系统定时器有三个 52 位比较器，用 COMP*x* 表示，其中 *x* 可以取 0、1、2。比较器可根据设置的不同报警值 (t) 或报警周期 (δt)，触发不同的中断。

用户可配置寄存器 SYSTIMER_TARGET*x*_PERIOD_MODE 选择比较器 COMP*x* 生成报警的模式：

- 1：选择周期报警模式
- 0：选择单次报警模式

选择周期报警模式时，寄存器 SYSTIMER_TARGET*x*_PERIOD 中的值为报警周期 (δt)。假设当前计数值为 t1，经过一段时间，当计数值达到 t1 + δt 时，将触发一次报警中断。再经过一段时间，当计数值达到 t1 + 2*δt 时，将再次触发一次报警中断，以此类推。通过上述方式即可实现周期性报警。

选择单次报警模式时，SYSTIMER_TIMER_TARGET*x*_LO 和 SYSTIMER_TIMER_TARGET*x*_HI 分别提供报警值 (t) 的低 32 位和高 20 位。假设当前计数值为 t2 (t2 <= t)，经过一段时间，当计数到报警值 (t) 时，则触发一次报警。与周期报警模式不同，单次报警模式仅生成一次报警中断。

用户可配置寄存器 SYSTIMER_TARGET*x*_TIMER_UNIT_SEL 选择用于与 COMP*x* 进行比较的计数器值，然后生成报警：

- 1：选择与计数器 UNIT*1* 的计数值进行比较
- 0：选择与计数器 UNIT*0* 的计数值进行比较

置位 SYSTIMER_TARGET*x*_WORK_EN，COMP*x* 开始进行比较：

- 在单次报警模式下，COMP*x* 将比较计数器中的实际计数值与寄存器中设置的报警值 (t)；
- 在周期报警模式下，COMP*x* 将比较计数器中的实际计数值与 t1 + n*δt (n = 1,2,3...)。

实际计数值等于报警值 (t)，或等于 t1 + n*δt (n=1,2,3...)，则触发一次报警中断。但如果设定的报警值 (t) 小于当前计数值，即报警值 (t) 已成为过去，或当前计数值超过设定的报警值 (t) 一定范围 (0 ~ 2^{51} -1)，则也将立即触发中断。当前计数值 t_c、报警值 t_t 和触发报警的关系如下表所示。注意，无论选择单次报警模式还是选择周期报警模式，真正的报警值（即预设的报警值）均可从 SYSTIMER_TARGET*x*_LO_RO（低 32 位）和 SYSTIMER_TARGET*x*_HI_RO（高 20 位）中读取。

表 11.4-2. 报警触发条件

t_c 与 t_t 的关系	触发条件
t_c- t_t <= 0	当 t_c = t_t 时，触发报警
0 <= t_c - t_t < 2^{51} - 1 (t_c < 2^{51} 且 t_t < 2^{51}； 或 t_c >= 2^{51} 且 t_t >= 2^{51})	立即触发报警
t_c - t_t >= 2^{51} - 1	t_c 达到最大值 52'hfffffffffffff 后溢出，然后从 0 开始计数，计数达到 t_t 时触发报警

11.4.3 同步操作

软件操作与计数器和比较器工作在不同时钟频率下，因此需要对部分配置寄存器进行同步。完整的同步过程包括下面两个步骤：

1. 通过软件向配置寄存器写入合适的值，见表 11.4-3 第一列；
2. 通过软件置位相应的同步使能位，开始同步操作，见表 11.4-3 第二列。

表 11.4-3. 同步操作

需要同步的字段	同步使能位
SYSTIMER_TIMER_UNIT*n*_LOAD_LO SYSTIMER_TIMER_UNIT*n*_LOAD_HI	SYSTIMER_TIMER_UNIT*n*_LOAD
SYSTIMER_TARGET*x*_PERIOD SYSTIMER_TIMER_TARGET*x*_HI SYSTIMER_TIMER_TARGET*x*_LO	SYSTIMER_TIMER_COMP*x*_LOAD

11.4.4 中断

上述三个比较器均有一个对应的报警中断，即 SYSTIMER_TARGET*x*_INT 中断，该中断为电平类型中断。比较器开始触发报警，即拉高中断信号。中断信号将一直保持高电平，直至软件清除中断。用户可置位

SYSTIMER_TARGET
*x*_INT_ENA 使能中断。

11.5 编程示例

注意，在配置 COMP*x* 和 UNIT*n* 过程中，需保证对应的 COMP 和 UNIT 处于工作状态。

11.5.1 读取当前计数器的值

1. 置位 SYSTIMER_TIMER_UNIT*n*_UPDATE，将计数器 UNIT*n* 的值更新至寄存器 SYSTIMER_TIMER_UNIT*n*_VALUE_LO 和 SYSTIMER_TIMER_UNIT*n*_VALUE_HI；
2. 轮询 SYSTIMER_TIMER_UNIT*n*_VALUE_VALID 直至其为 1。之后，用户可从寄存器 SYSTIMER_TIMER_UNIT*n*_VALUE_HI 和 SYSTIMER_TIMER_UNIT*n*_VALUE_LO 中读取计数器的值；
3. 读取寄存器 SYSTIMER_TIMER_UNIT*n*_VALUE_LO（低 32 位）和 SYSTIMER_TIMER_UNIT*n*_VALUE_HI（高 20 位）。

11.5.2 在单次报警模式下配置一次性报警

1. 设置 SYSTIMER_TARGET*x*_TIMER_UNIT_SEL 选择与 COMP*x* 进行比较的计数器（UNIT*0* 或 UNIT*1*）；
2. 读取当前计数器的值，步骤见章节 11.5.1。读取的当前值可用于计算步骤 4 中的报警值 (t)；
3. 清除 SYSTIMER_TARGET*x*_PERIOD_MODE，使能单次报警模式；
4. 设置报警值 (t)，并将报警值 (t) 的低 32 位和高 20 位分别写入 SYSTIMER_TIMER_TARGET*x*_LO 和 SYSTIMER_TIMER_TARGET*x*_HI；
5. 置位 SYSTIMER_TIMER_COMP*x*_LOAD，同步报警值 (t)，即将报警值 (t) 装载至比较器 COMP*x*；
6. 置位 SYSTIMER_TARGET*x*_WORK_EN 使能选择的比较器 COMP*x*；比较器 COMP 开始比较计数值与报警值 (t)；
7. 置位 SYSTIMER_TARGET*x*_INT_ENA，使能中断。UNIT*n* 达到报警值 (t) 则触发一次报警中断 SYSTIMER_TARGET*x*_INT。

11.5.3 在周期报警模式下配置周期性报警

1. 设置 SYSTIMER_TARGET*x*_TIMER_UNIT_SEL 选择与 COMP*x* 进行比较的计数器；
2. 将报警周期 (δt) 写入 SYSTIMER_TARGET*x*_PERIOD；
3. 置位 SYSTIMER_TIMER_COMP*x*_LOAD 同步报警周期值，即将 (δt) 装载至比较器 COMP*x*；
4. 先清除再置位 SYSTIMER_TARGET*x*_PERIOD_MODE 将 COMP*x* 配置为周期报警模式；
5. 置位 SYSTIMER_TARGET*x*_WORK_EN 使能选择的比较器 COMP*x*；比较器 COMP*x* 开始将计数值与（计数初始值 + n*δt (n = 1, 2, 3...)）进行比较；
6. 置位 SYSTIMER_TARGET*x*_INT_ENA，使能中断。UNIT*n* 计数达到计数初始值 + n*δt (n = 1, 2, 3...)，则触发一次 SYSTIMER_TARGET*x*_INT 中断。

11.5.4 唤醒后时间补偿

1. 在芯片进入 Light-sleep 模式之前，用户需配置 RTC 定时器用于精确记录睡眠时间，见章节 10 低功耗管理 (RTC_CNTL)；
2. 系统从 Light-sleep 模式唤醒后，读取 RTC 定时器记录的睡眠时间；
3. 读取当前系统定时器的计数值，见章节 11.5.1；
4. 将 RTC 记录的睡眠时间，单位：RTC_SLOW_CLK 周期，转换成以 CNT_CLK (16 MHz) 周期为单位的睡眠时间。例如，如果 RTC_SLOW_CLK 频率为 32 kHz，则 RTC 定时器记录的时间乘以 500 即可。
5. 将 RTC 定时器转换后的值加到系统定时器当前计数值：
   - 将计算所得值，低 32 位写入 SYSTIMER_TIMER_UNIT*n*_LOAD_LO，高 20 位写入 SYSTIMER_TIMER_UNIT*n*_LOAD_HI；
   - 置位 SYSTIMER_TIMER_UNIT*n*_LOAD，将新的定时器值装载到系统定时器。这样即可完成系统定时器更新。

11.6 寄存器列表

本小节的所有地址均为相对于系统定时器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

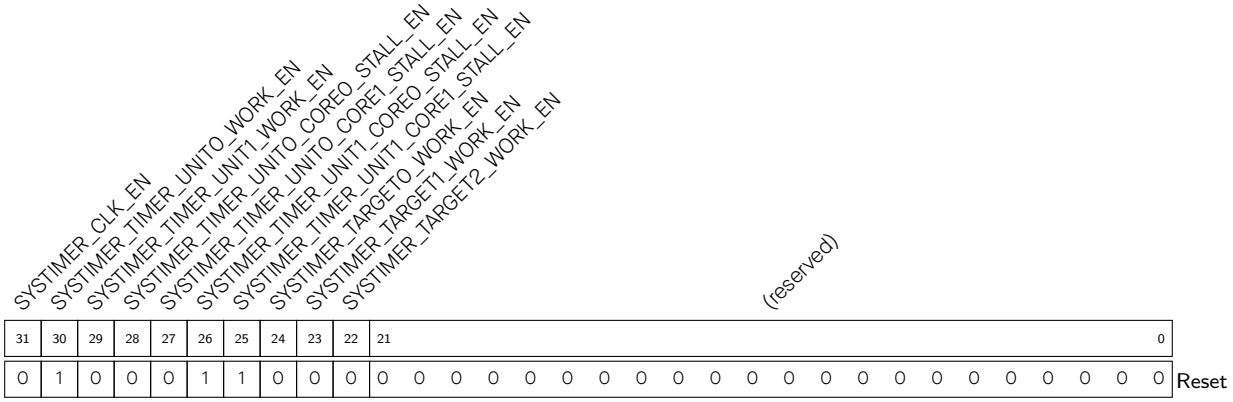
名称	描述	地址	访问
时钟控制寄存器			
SYSTIMER_CONF_REG	配置系统定时器时钟	0x0000	R/W
UNIT*0* 控制和配置寄存器			
SYSTIMER_UNIT0_OP_REG	读取 UNT0 的值到相应寄存器	0x0004	varies
SYSTIMER_UNIT0_LOAD_HI_REG	待装载至 UNIT0 的值，高 20 位	0x000C	R/W
SYSTIMER_UNIT0_LOAD_LO_REG	待装载至 UNIT0 的值，低 32 位	0x0010	R/W
SYSTIMER_UNIT0_VALUE_HI_REG	UNIT0 的读值，高 20 位	0x0040	RO
SYSTIMER_UNIT0_VALUE_LO_REG	UNIT0 的读值，低 32 位	0x0044	RO
SYSTIMER_UNIT0_LOAD_REG	计数器 UNIT0 的装载同步寄存器	0x005C	WT
UNIT*1* 控制和配置寄存器			
SYSTIMER_UNIT1_OP_REG	读取计数器 UNIT1 的值	0x0008	varies
SYSTIMER_UNIT1_LOAD_HI_REG	待装载至计数器 UNIT1 的值，高 20 位	0x0014	R/W
SYSTIMER_UNIT1_LOAD_LO_REG	待装载至计数器 UNIT1 的值，低 32 位	0x0018	R/W
SYSTIMER_UNIT1_VALUE_HI_REG	计数器 UNIT1 的读值，高 20 位	0x0048	RO
SYSTIMER_UNIT1_VALUE_LO_REG	计数器 UNIT1 的读值，低 32 位	0x004C	RO
SYSTIMER_UNIT1_LOAD_REG	计数器 UNIT1 的装载同步寄存器	0x0060	WT
比较器 COMP*0* 的控制和配置寄存器			
SYSTIMER_TARGET0_HI_REG	待装载至比较器 COMP0 的报警值，高 20 位	0x001C	R/W
SYSTIMER_TARGET0_LO_REG	待装载至比较器 COMP0 的报警值，低 32 位	0x0020	R/W
SYSTIMER_TARGET0_CONF_REG	配置比较器 COMP0 的报警模式	0x0034	R/W
SYSTIMER_COMP0_LOAD_REG	比较器 COMP0 的装载同步寄存器	0x0050	WT

名称	描述	地址	访问
比较器 COMP*1* 的控制和配置寄存器			
SYSTIMER_TARGET1_HI_REG	待装载至比较器 COMP1 的报警值，高 20 位	0x0024	R/W
SYSTIMER_TARGET1_LO_REG	待装载至比较器 COMP1 的报警值，低 32 位	0x0028	R/W
SYSTIMER_TARGET1_CONF_REG	配置比较器 COMP1 的报警模式	0x0038	R/W
SYSTIMER_COMP1_LOAD_REG	比较器 COMP1 的装载同步寄存器	0x0054	WT
比较器 COMP*2* 的控制和配置寄存器			
SYSTIMER_TARGET2_HI_REG	待装载至比较器 COMP2 的报警值，高 20 位	0x002C	R/W
SYSTIMER_TARGET2_LO_REG	待装载至比较器 COMP2 的报警值，低 32 位	0x0030	R/W
SYSTIMER_TARGET2_CONF_REG	配置比较器 COMP2 的报警模式	0x003C	R/W
SYSTIMER_COMP2_LOAD_REG	比较器 COMP2 的装载同步寄存器	0x0058	WT
中断寄存器			
SYSTIMER_INT_ENA_REG	系统定时器的中断使能寄存器	0x0064	R/W
SYSTIMER_INT_RAW_REG	系统定时器的原始中断寄存器	0x0068	R/WTC/SS
SYSTIMER_INT_CLR_REG	系统定时器的中断清除寄存器	0x006C	WT
SYSTIMER_INT_ST_REG	系统定时器的中断状态寄存器	0x0070	RO
COMP*0* 状态寄存器			
SYSTIMER_REAL_TARGET0_LO_REG	COMP0 的实际报警值，低 32 位	0x0074	RO
SYSTIMER_REAL_TARGET0_HI_REG	COMP0 的实际报警值，高 20 位	0x0078	RO
COMP*1* 状态寄存器			
SYSTIMER_REAL_TARGET1_LO_REG	COMP1 的实际报警值，低 32 位	0x007C	RO
SYSTIMER_REAL_TARGET1_HI_REG	COMP1 的实际报警值，高 20 位	0x0080	RO
COMP*2* 状态寄存器			
SYSTIMER_REAL_TARGET2_LO_REG	COMP2 的实际报警值，低 32 位	0x0084	RO
SYSTIMER_REAL_TARGET2_HI_REG	COMP2 的实际报警值，高 20 位	0x0088	RO
版本寄存器			
SYSTIMER_DATE_REG	版本控制寄存器	0x00FC	R/W

11.7 寄存器

本小节的所有地址均为相对于系统定时器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 11.1. SYSTIMER_CONF_REG (0x0000)



SYSTIMER_TARGET2_WORK_EN 置位此位，使能比较器 COMP2。(R/W)

SYSTIMER_TARGET1_WORK_EN 置位此位，使能比较器 COMP1。(R/W)

SYSTIMER_TARGET0_WORK_EN 置位此位，使能比较器 COMP0。(R/W)

SYSTIMER_TIMER_UNIT1_CORE1_STALL_EN 置位此位，则如果 CPU1 停止工作，计数器 UNIT1 也将停止工作。(R/W)

SYSTIMER_TIMER_UNIT1_CORE0_STALL_EN 置位此位，则如果 CPU0 停止工作，计数器 UNIT1 也将停止工作。(R/W)

SYSTIMER_TIMER_UNIT0_CORE1_STALL_EN 置位此位，则如果 CPU1 停止工作，计数器 UNIT0 也将停止工作。(R/W)

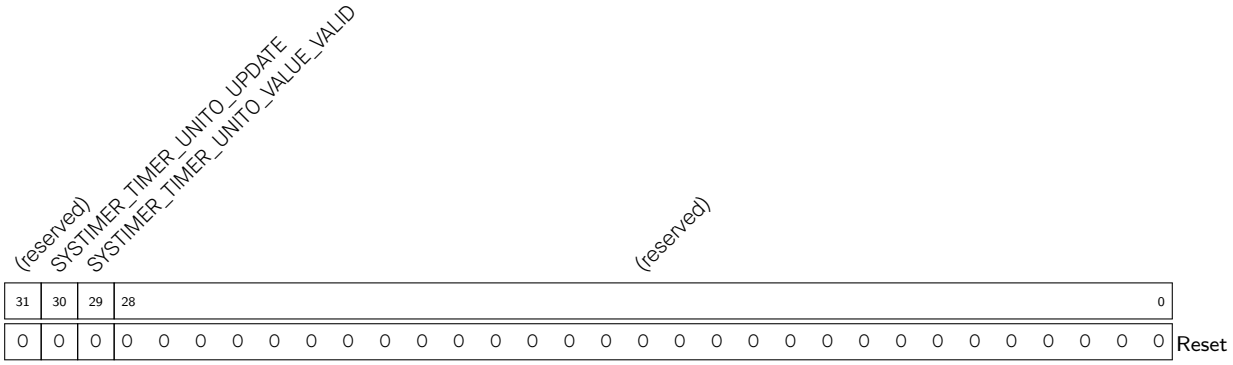
SYSTIMER_TIMER_UNIT0_CORE0_STALL_EN 置位此位，则如果 CPU0 停止工作，计数器 UNIT0 也将停止工作。(R/W)

SYSTIMER_TIMER_UNIT1_WORK_EN 使能计数器 UNIT1。(R/W)

SYSTIMER_TIMER_UNIT0_WORK_EN 使能计数器 UNIT0。(R/W)

SYSTIMER_CLK_EN 寄存器时钟门控。1：持续开启读写寄存器的时钟；0：只在读写寄存器时打开所需时钟。(R/W)

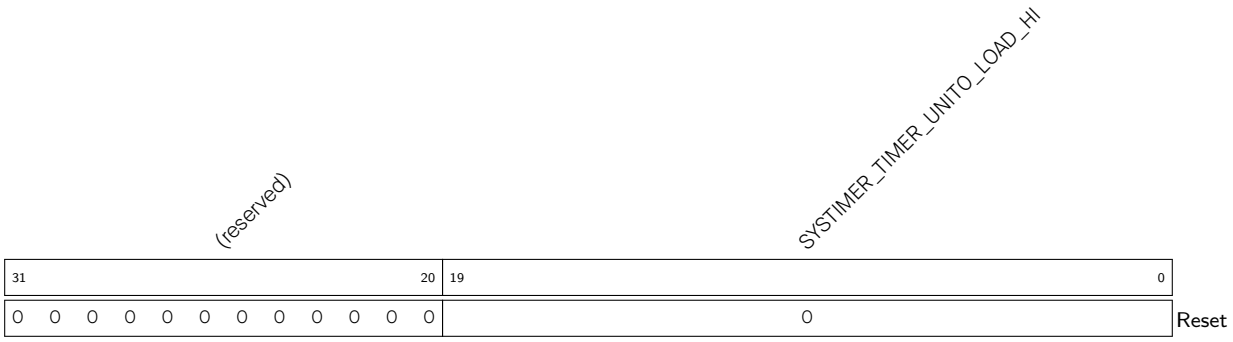
Register 11.2. SYSTIMER_UNIT0_OP_REG (0x0004)



SYSTIMER_TIMER_UNIT0_VALUE_VALID 计数器 UNIT0 的值已同步读取至寄存器，且读值有效。(R/SS/WTC)

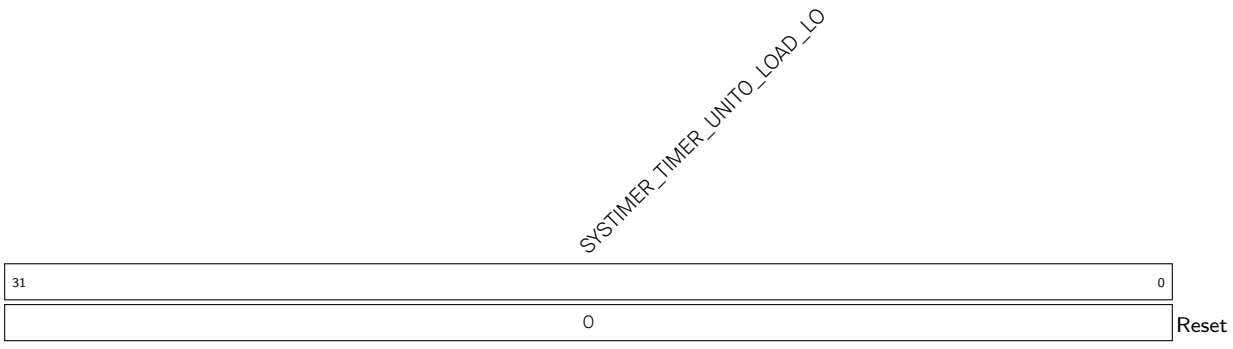
SYSTIMER_TIMER_UNIT0_UPDATE 读取计数器 UNIT0 的值到寄存器 SYSTIMER_TIMER_UNIT0_VALUE_HI 和 SYSTIMER_TIMER_UNIT0_VALUE_LO。(WT)

Register 11.3. SYSTIMER_UNIT0_LOAD_HI_REG (0x000C)



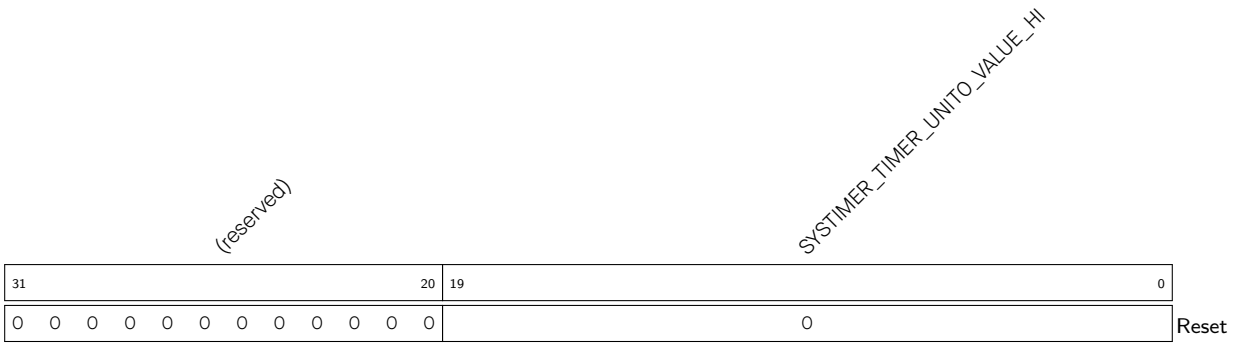
SYSTIMER_TIMER_UNIT0_LOAD_HI 待装载至计数器 UNIT0 的值，高 20 位。(R/W)

Register 11.4. SYSTIMER_UNIT0_LOAD_LO_REG (0x0010)



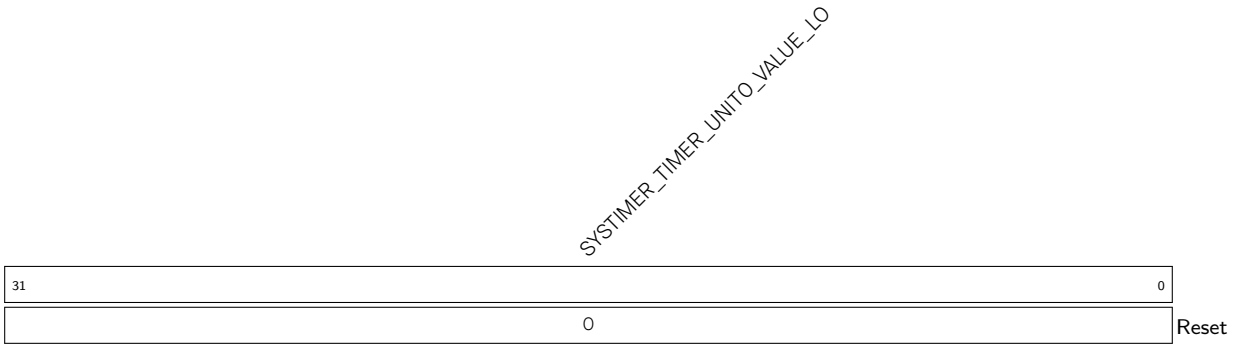
SYSTIMER_TIMER_UNIT0_LOAD_LO 待装载至计数器 UNIT0 的值，低 32 位。(R/W)

Register 11.5. SYSTIMER_UNIT0_VALUE_HI_REG (0x0040)



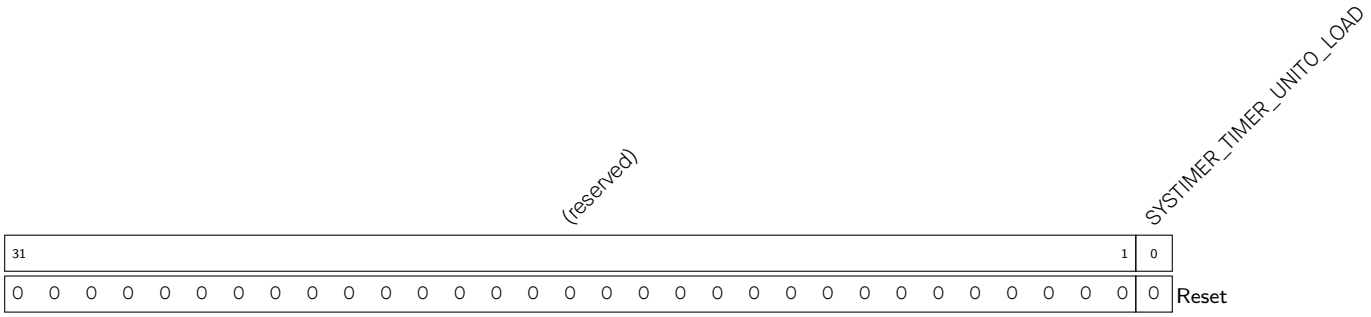
SYSTIMER_TIMER_UNIT0_VALUE_HI 计数器 UNIT0 的读数，高 20 位。(RO)

Register 11.6. SYSTIMER_UNIT0_VALUE_LO_REG (0x0044)



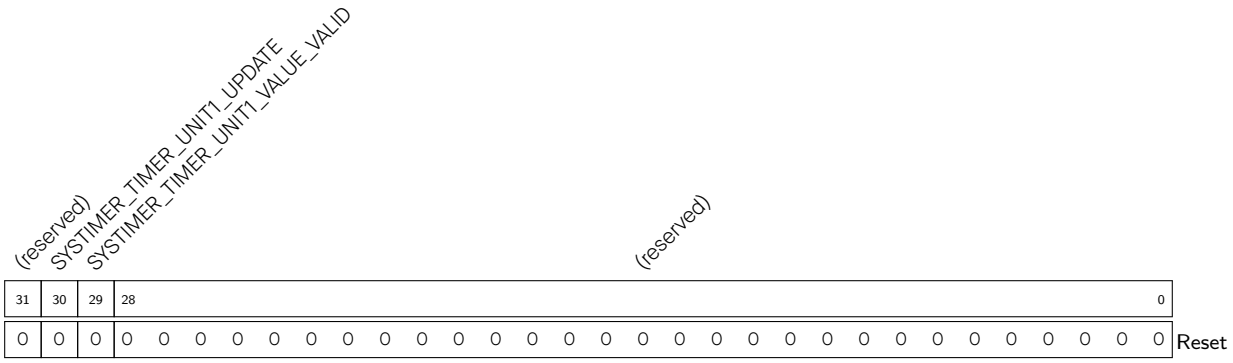
SYSTIMER_TIMER_UNIT0_VALUE_LO 计数器 UNIT0 的读数，低 32 位。(RO)

Register 11.7. SYSTIMER_UNIT0_LOAD_REG (0x005C)



SYSTIMER_TIMER_UNIT0_LOAD 计数器 UNIT0 的同步使能信号。置位此位，将重新装载寄存器 SYSTIMER_TIMER_UNIT0_LOAD_HI 和 SYSTIMER_TIMER_UNIT0_LOAD_LO 的值到计数器 UNIT0。(WT)

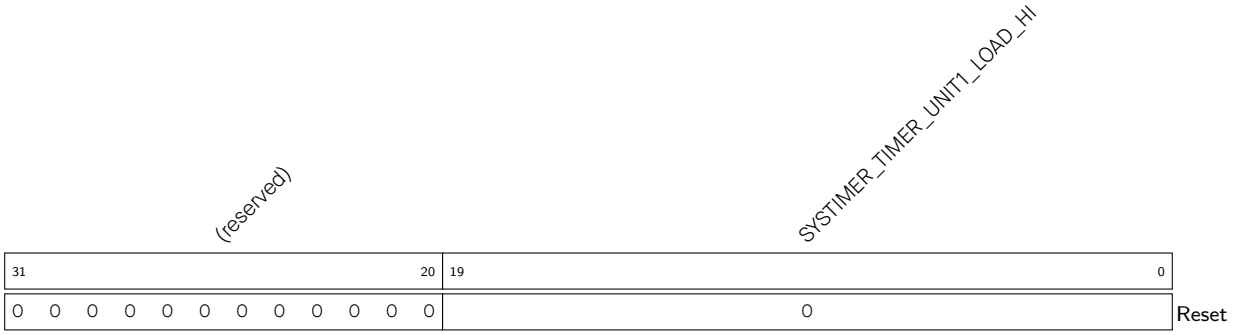
Register 11.8. SYSTIMER_UNIT1_OP_REG (0x0008)



SYSTIMER_TIMER_UNIT1_VALUE_VALID 计数器 UNIT1 的值已同步读取至寄存器，且读值有效。(R/SS/WTC)

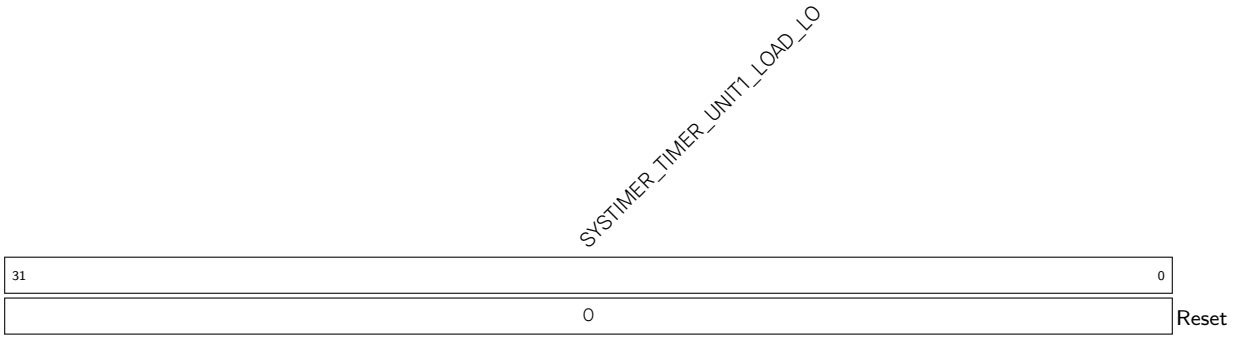
SYSTIMER_TIMER_UNIT1_UPDATE 将计数器 UNIT1 的值读取到寄存器 SYSTIMER_TIMER_UNIT1_VALUE_HI 和 SYSTIMER_TIMER_UNIT1_VALUE_LO。(WT)

Register 11.9. SYSTIMER_UNIT1_LOAD_HI_REG (0x0014)



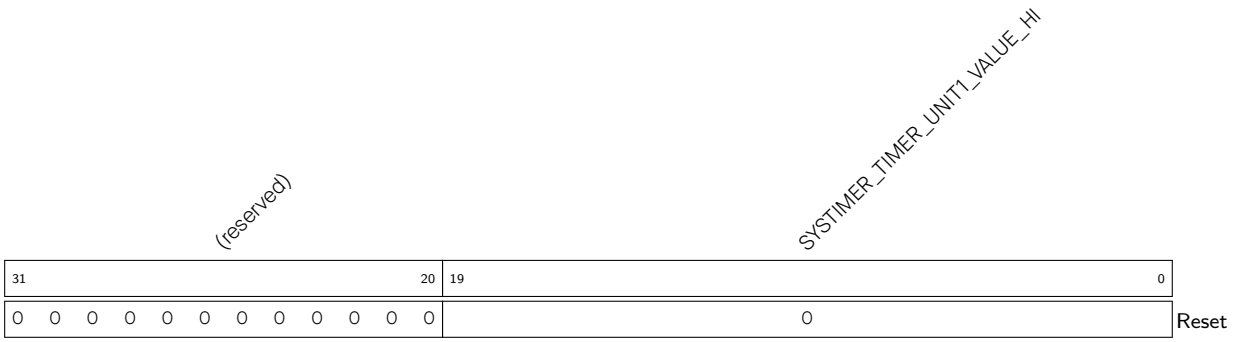
SYSTIMER_TIMER_UNIT1_LOAD_HI 待装载至计数器 UNIT1 的值，高 20 位。(R/W)

Register 11.10. SYSTIMER_UNIT1_LOAD_LO_REG (0x0018)



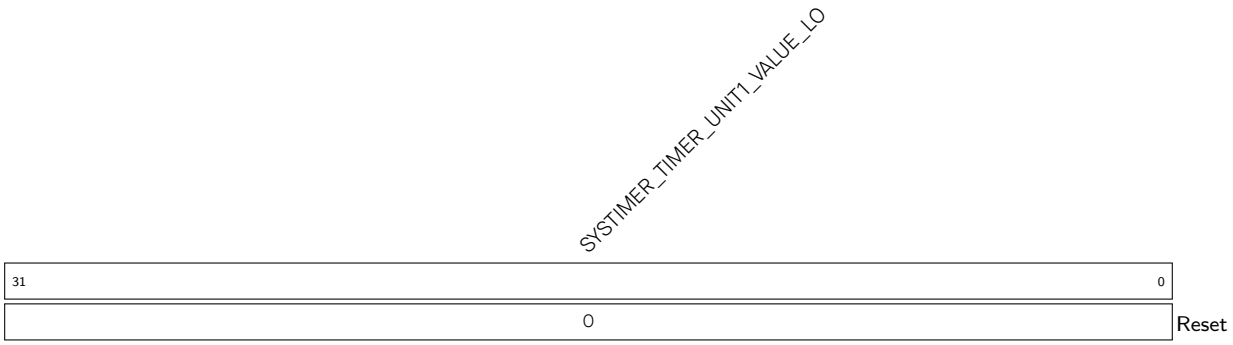
SYSTIMER_TIMER_UNIT1_LOAD_LO 待装载至计数器 UNIT1 的值，低 32 位。(R/W)

Register 11.11. SYSTIMER_UNIT1_VALUE_HI_REG (0x0048)



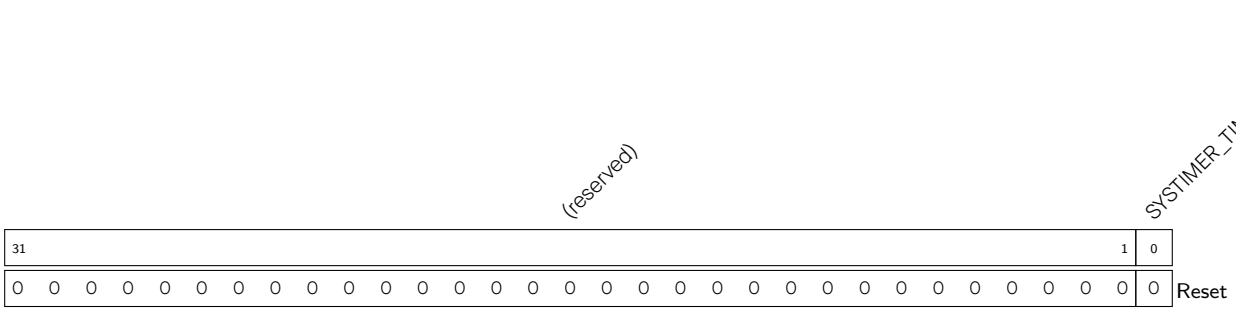
SYSTIMER_TIMER_UNIT1_VALUE_HI 计数器 UNIT1 的读数，高 20 位。(RO)

Register 11.12. SYSTIMER_UNIT1_VALUE_LO_REG (0x004C)



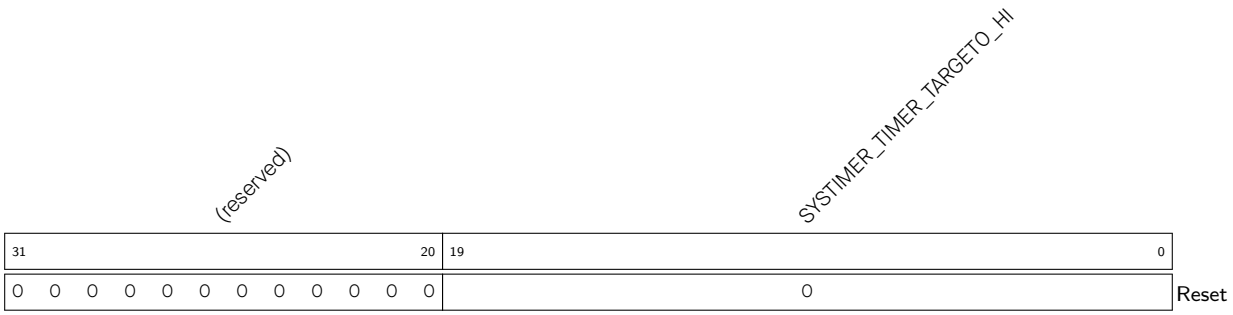
SYSTIMER_TIMER_UNIT1_VALUE_LO 计数器 UNIT1 的读数，低 32 位。(RO)

Register 11.13. SYSTIMER_UNIT1_LOAD_REG (0x0060)



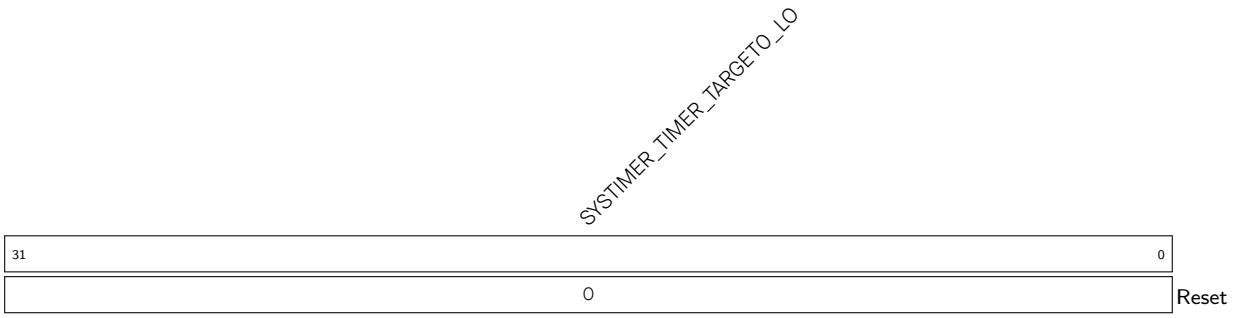
SYSTIMER_TIMER_UNIT1_LOAD 计数器 UNIT1 的同步使能信号。置位此位，将重新装载寄存器 SYSTIMER_TIMER_UNIT1_LOAD_HI 和 SYSTIMER_TIMER_UNIT1_LOAD_LO 的值到计数器 UNIT1。(WT)

Register 11.14. SYSTIMER_TARGET0_HI_REG (0x001C)



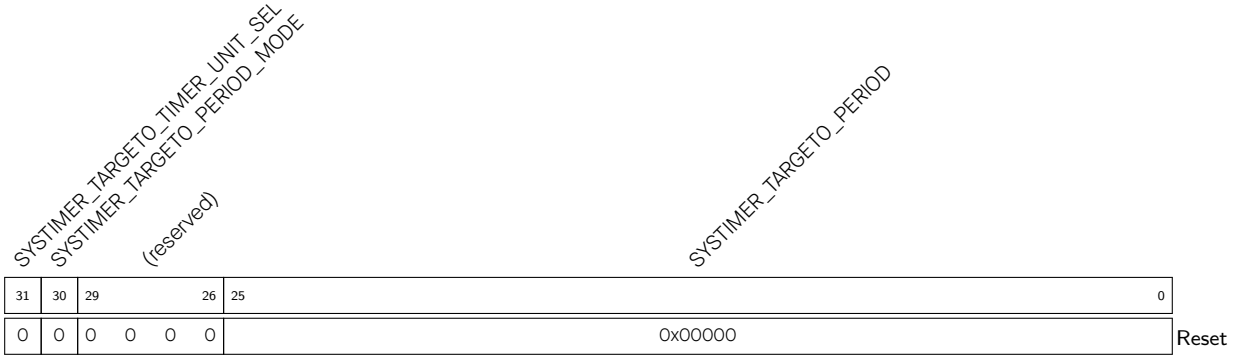
SYSTIMER_TIMER_TARGET0_HI 待装载至 COMP0 的报警值，高 20 位。(R/W)

Register 11.15. SYSTIMER_TARGET0_LO_REG (0x0020)



SYSTIMER_TIMER_TARGET0_LO 待装载至 COMP0 的报警值，低 32 位。(R/W)

Register 11.16. SYSTIMER_TARGET0_CONF_REG (0x0034)

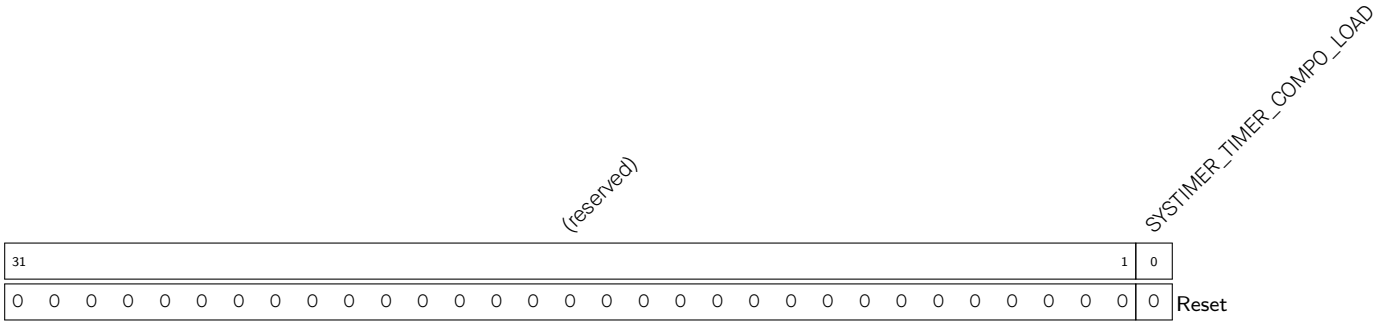


SYSTIMER_TARGET0_PERIOD 待装载至 COMP0 的报警周期。(R/W)

SYSTIMER_TARGET0_PERIOD_MODE 设置 COMP0 为周期报警模式。(R/W)

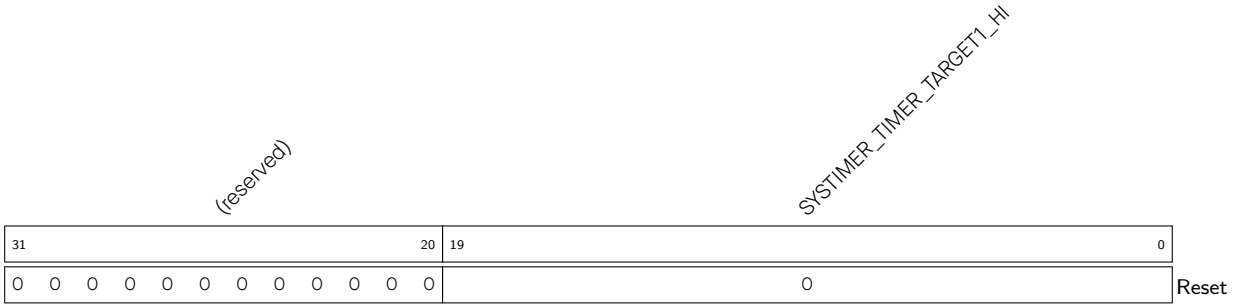
SYSTIMER_TARGET0_TIMER_UNIT_SEL 选择要与 COMP0 比较的计数器。(R/W)

Register 11.17. SYSTIMER_COMP0_LOAD_REG (0x0050)



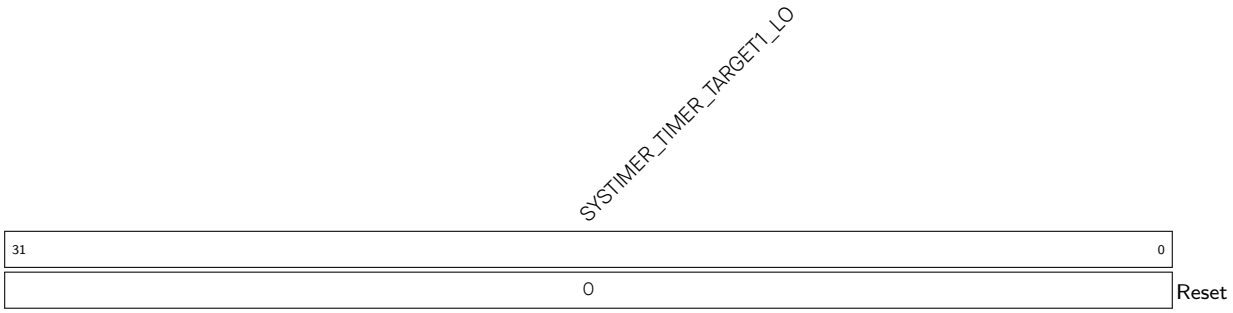
SYSTIMER_TIMER_COMP0_LOAD 比较器 COMP0 的同步使能信号。置位此位，将重新装载报警值或报警周期到 COMP0。(WT)

Register 11.18. SYSTIMER_TARGET1_HI_REG (0x0024)



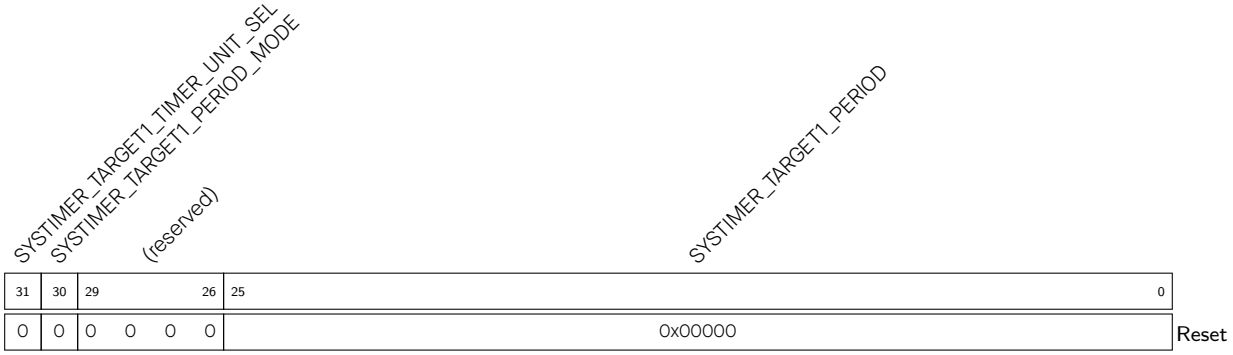
SYSTIMER_TIMER_TARGET1_HI 待装载至 COMP1 的报警值，高 20 位。(R/W)

Register 11.19. SYSTIMER_TARGET1_LO_REG (0x0028)



SYSTIMER_TIMER_TARGET1_LO 待装载至 COMP1 的报警值，低 32 位。(R/W)

Register 11.20. SYSTIMER_TARGET1_CONF_REG (0x0038)

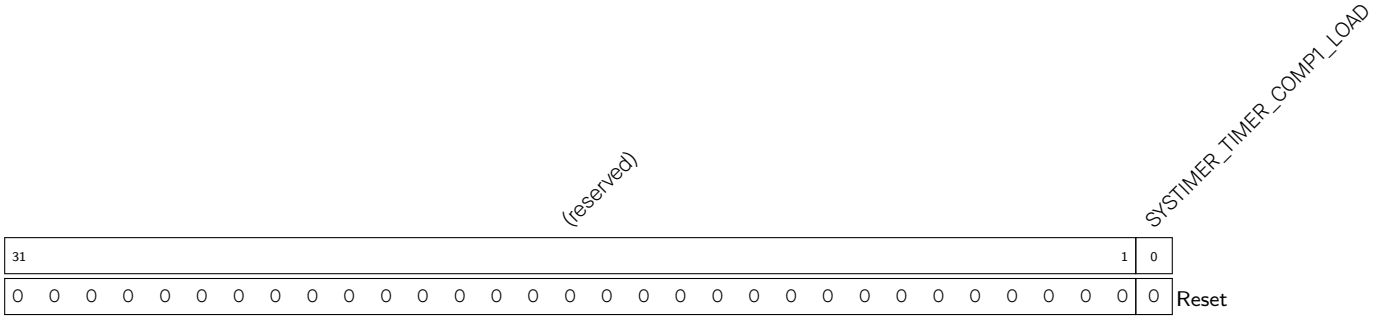


SYSTIMER_TARGET1_PERIOD 待装载至 COMP0 的报警周期。(R/W)

SYSTIMER_TARGET1_PERIOD_MODE 待装载至 COMP1 的报警周期。(R/W)

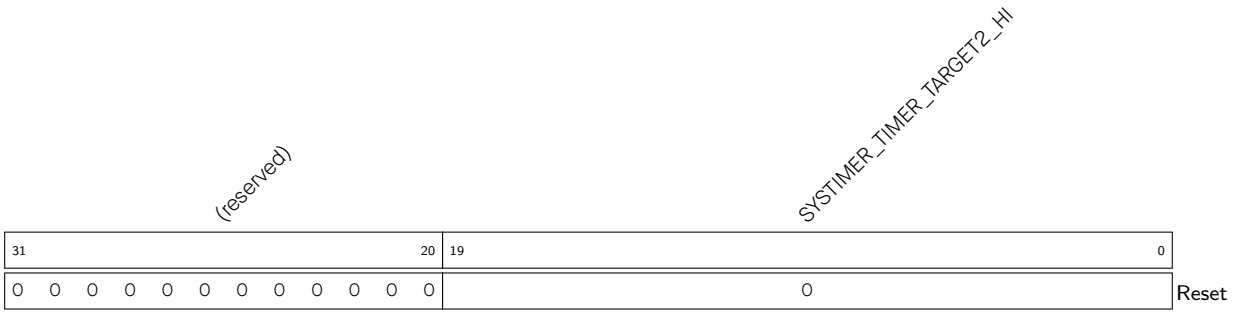
SYSTIMER_TARGET1_TIMER_UNIT_SEL 选择要与 COMP1 比较的计数器。(R/W)

Register 11.21. SYSTIMER_COMP1_LOAD_REG (0x0054)



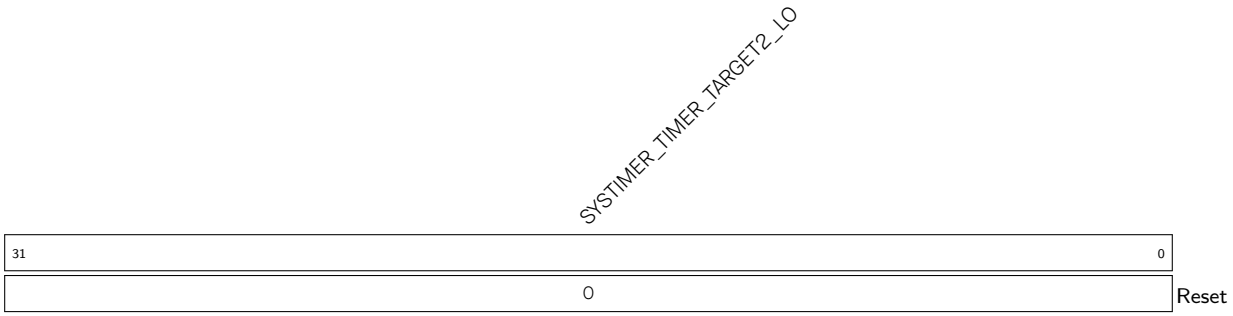
SYSTIMER_TIMER_COMP1_LOAD 比较器 COMP1 的同步使能信号。置位此位，将重新装载报警值或报警周期到 COMP1。(WT)

Register 11.22. SYSTIMER_TARGET2_HI_REG (0x002C)



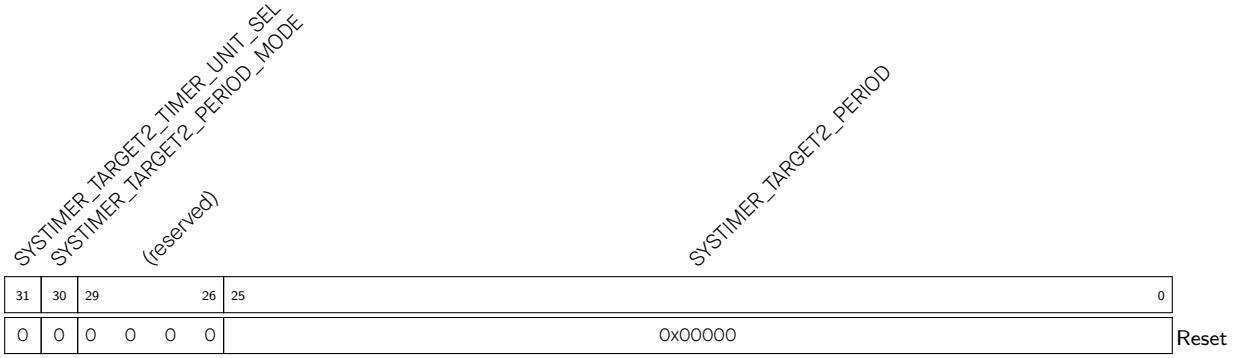
SYSTIMER_TIMER_TARGET2_HI 待装载至比较器 COMP2 的报警值，高 20 位。(R/W)

Register 11.23. SYSTIMER_TARGET2_LO_REG (0x0030)



SYSTIMER_TIMER_TARGET2_LO 待装载至比较器 COMP2 的报警值，低 32 位。(R/W)

Register 11.24. SYSTIMER_TARGET2_CONF_REG (0x003C)

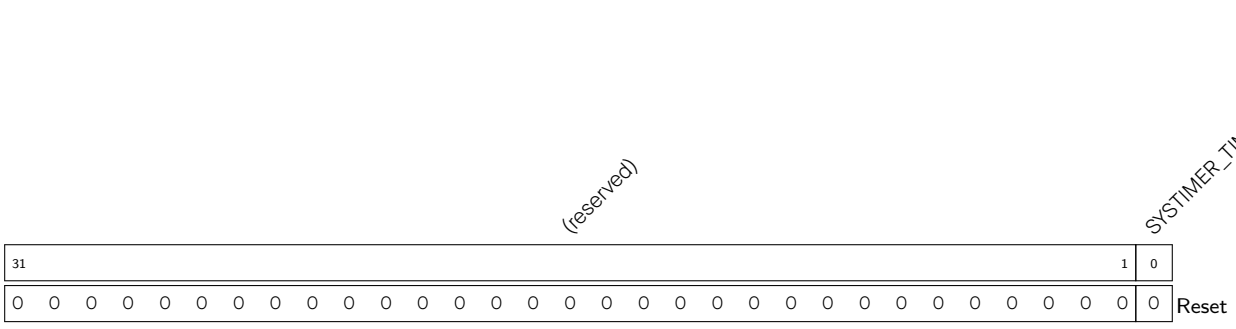


SYSTIMER_TARGET2_PERIOD 待装载至 COMP2 的报警周期。(R/W)

SYSTIMER_TARGET2_PERIOD_MODE 设置 COMP2 为周期报警模式。(R/W)

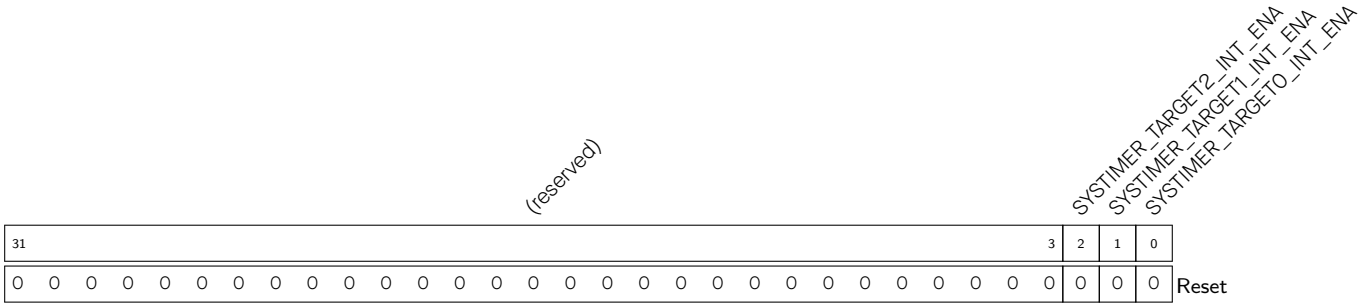
SYSTIMER_TARGET2_TIMER_UNIT_SEL 选择要与 COMP2 比较的计数器。(R/W)

Register 11.25. SYSTIMER_COMP2_LOAD_REG (0x0058)



SYSTIMER_TIMER_COMP2_LOAD 比较器 COMP2 的同步使能信号。置位此位，将重新装载报警值或报警周期至 COMP2。(WT)

Register 11.26. SYSTIMER_INT_ENA_REG (0x0064)

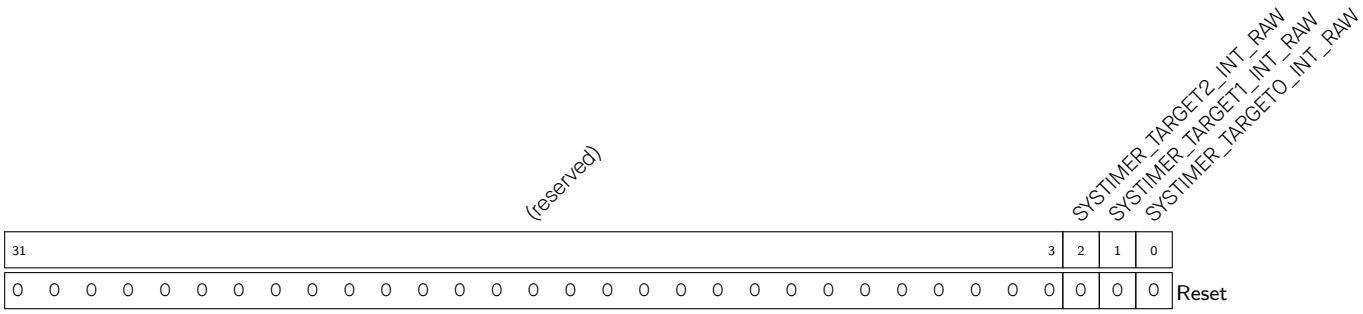


SYSTIMER_TARGET0_INT_ENA SYSTIMER_TARGET0_INT 中断使能位。(R/W)

SYSTIMER_TARGET1_INT_ENA SYSTIMER_TARGET1_INT 中断使能位。(R/W)

SYSTIMER_TARGET2_INT_ENA SYSTIMER_TARGET2_INT 中断使能位。(R/W)

Register 11.27. SYSTIMER_INT_RAW_REG (0x0068)

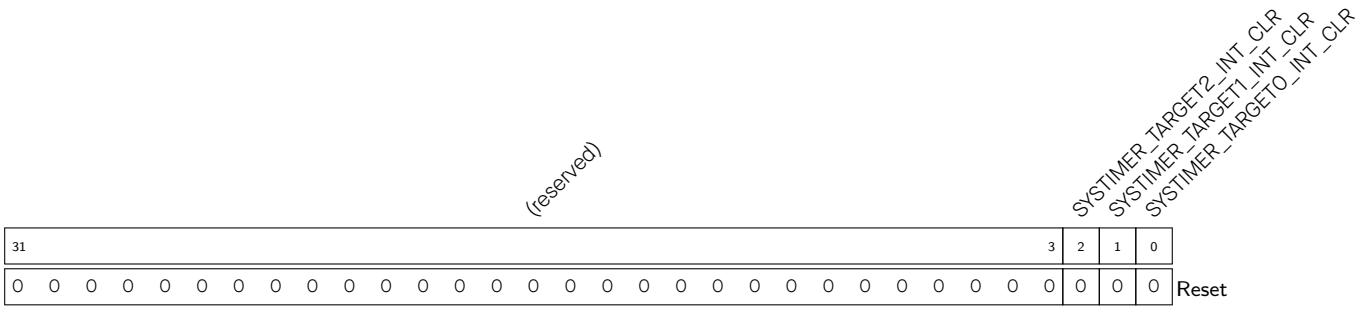


SYSTIMER_TARGET0_INT_RAW SYSTIMER_TARGET0_INT 中断原始位。(R/WTC/SS)

SYSTIMER_TARGET1_INT_RAW SYSTIMER_TARGET1_INT 中断原始位。(R/WTC/SS)

SYSTIMER_TARGET2_INT_RAW SYSTIMER_TARGET2_INT 中断原始位。(R/WTC/SS)

Register 11.28. SYSTIMER_INT_CLR_REG (0x006C)

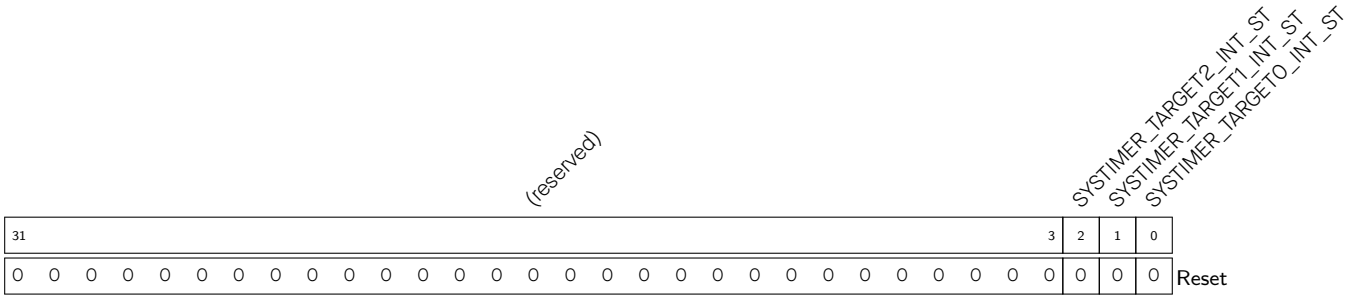


SYSTIMER_TARGET0_INT_CLR SYSTIMER_TARGET0_INT 中断清除位。(WT)

SYSTIMER_TARGET1_INT_CLR SYSTIMER_TARGET1_INT 中断清除位。(WT)

SYSTIMER_TARGET2_INT_CLR SYSTIMER_TARGET2_INT 中断清除位。(WT)

Register 11.29. SYSTIMER_INT_ST_REG (0x0070)

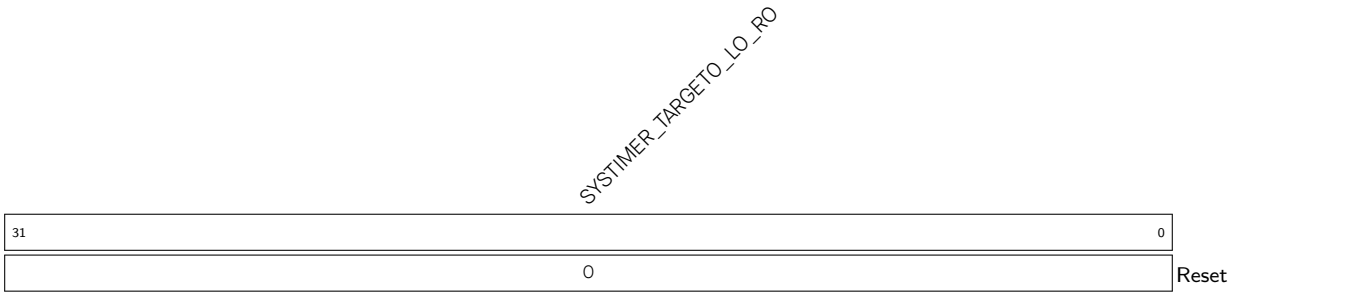


SYSTIMER_TARGET0_INT_ST SYSTIMER_TARGET0_INT 中断状态位。(RO)

SYSTIMER_TARGET1_INT_ST SYSTIMER_TARGET1_INT 中断状态位。(RO)

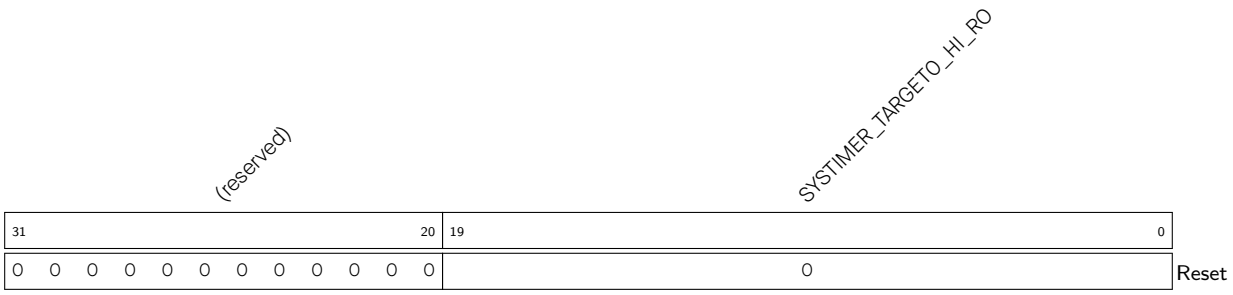
SYSTIMER_TARGET2_INT_ST SYSTIMER_TARGET2_INT 中断状态位。(RO)

Register 11.30. SYSTIMER_REAL_TARGET0_LO_REG (0x0074)



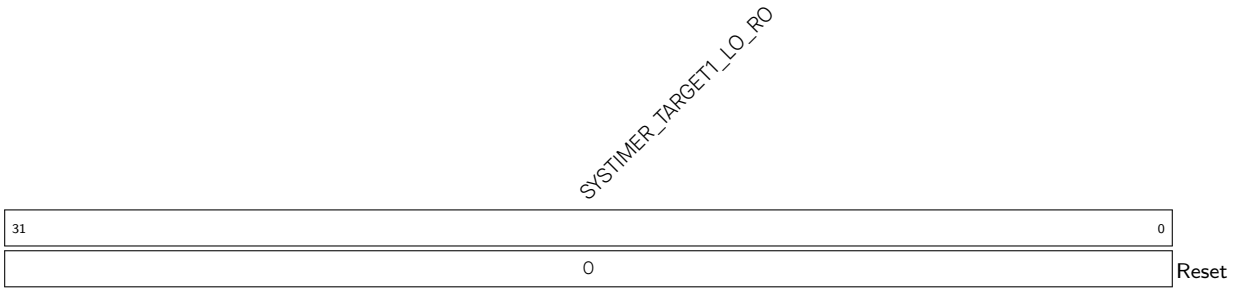
SYSTIMER_TARGET0_LO_RO COMP0 的实际报警值，低 32 位。(RO)

Register 11.31. SYSTIMER_REAL_TARGET0_HI_REG (0x0078)



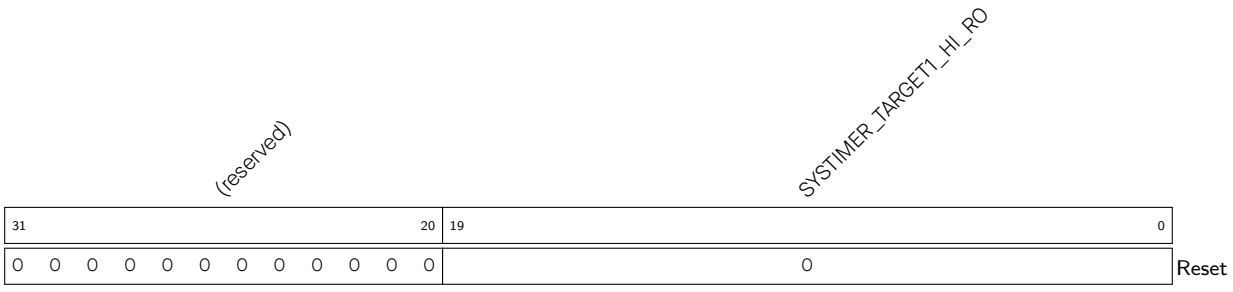
SYSTIMER_TARGET0_HI_RO COMP0 的实际报警值，高 20 位。(RO)

Register 11.32. SYSTIMER_REAL_TARGET1_LO_REG (0x007C)



SYSTIMER_TARGET1_LO_RO COMP1 的实际报警值，低 32 位。(RO)

Register 11.33. SYSTIMER_REAL_TARGET1_HI_REG (0x0080)



SYSTIMER_TARGET1_HI_RO COMP1 的实际报警值，高 20 位。(RO)

Register 11.34. SYSTIMER_REAL_TARGET2_LO_REG (0x0084)



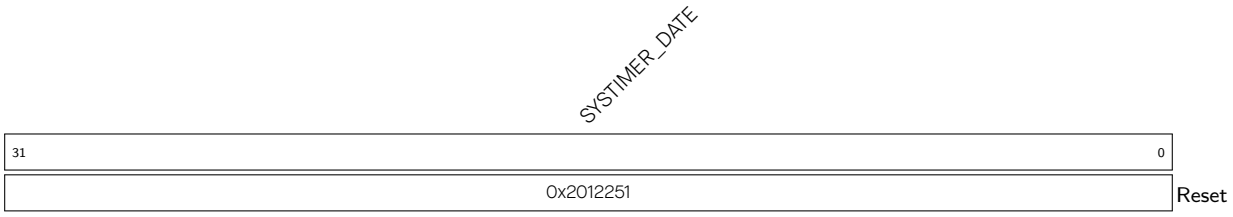
SYSTIMER_TARGET2_LO_RO COMP2 的实际报警值，低 32 位。(RO)

Register 11.35. SYSTIMER_REAL_TARGET2_HI_REG (0x0088)



SYSTIMER_TARGET2_HI_RO COMP2 的实际报警值，高 20 位。(RO)

Register 11.36. SYSTIMER_DATE_REG (0x00FC)



SYSTIMER_DATE 版本控制寄存器 (R/W)

第 12 章

定时器组 (TIMG)

12.1 概述

通用定时器可用于准确设定时间间隔、在一定间隔后触发（周期或非周期的）中断或充当硬件时钟。如图 12.1-1 所示，ESP32-S3 包含两个定时器组，即定时器组 0 和定时器组 1。每个定时器组有两个通用定时器（下文用 T*x* 表示，*x* 为 0 或 1）和一个主系统看门狗定时器。所有通用定时器均基于 16 位预分频器和 54 位可自动重新加载向上 / 向下计数器。

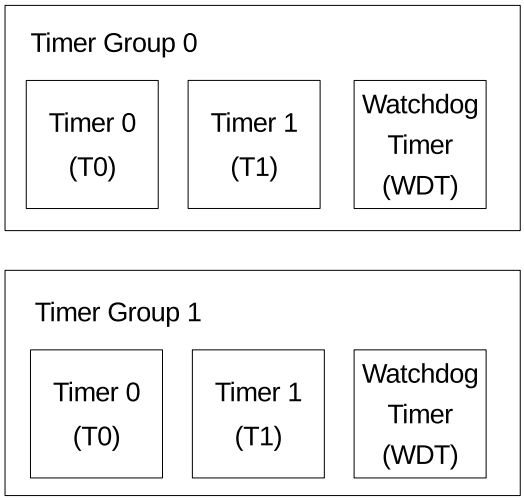


图 12.1-1. 定时器组

本章包含主系统看门狗定时器的寄存器描述，其功能描述请参阅章节 13 *看门狗定时器 (WDT)*。本章中“定时器”指代通用定时器。

定时器具有如下功能：

- 16 位时钟预分频器，分频系数为 2 到 65536
- 54 位时基计数器可配置成递增或递减
- 可读取时基计数器的实时值
- 暂停和恢复时基计数器
- 可配置的报警产生机制
- 计数器值重新加载（报警时自动重新加载或软件控制的即时重新加载）
- 电平触发中断

12.2 功能描述

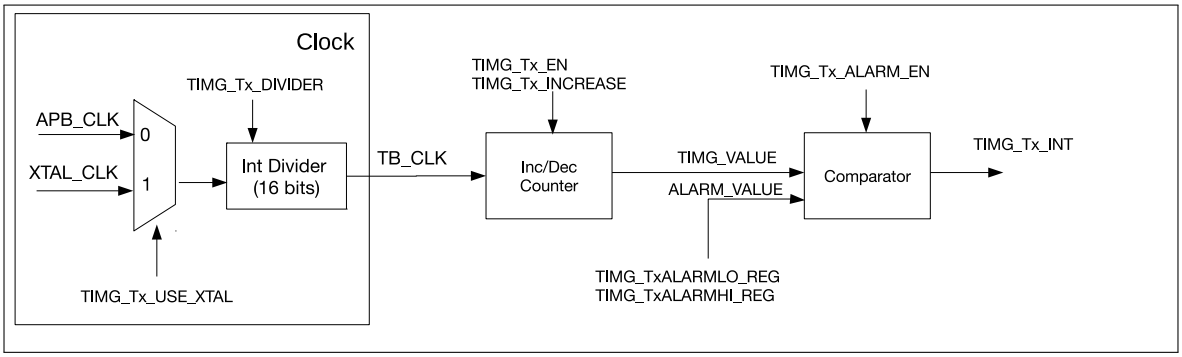


图 12.2-1. 定时器组架构

图 12.2-1 为定时器组的 T*x*。T*x* 包含时钟选择器、一个 16 位整数预分频器、一个时基计数器和一个用于产生警报的比较器。

12.2.1 16 位预分频器与时钟选择器

每个定时器可通过配置寄存器 TIMG_T*x*CONFIG_REG 的 TIMG_T*x*_USE_XTAL 字段，选择 APB 时钟 (APB_CLK) 或外部时钟 (XTAL_CLK) 作为时钟源。时钟源经 16 位预分频器分频，产生时基计数器使用的时基计数器时钟 (TB_CLK)。16 位预分频器的分频系数可通过 TIMG_T*x*_DIVIDER 字段配置，选取从 2 到 65536 之间的任意值。注意，将 TIMG_T*x*_DIVIDER 置 0 后，分频系数会变为 65536。TIMG_T*x*_DIVIDER 置 1 时，实际分频系数为 2，计数器的值为实际时间的一半。

定时器必须关闭（即 TIMG_T*x*_EN 必须清零），才能更改 16 位预分频器。在定时器使能时更改 16 位预分频器会造成不可预知的结果。

12.2.2 54 位时基计数器

54 位时基计数器基于 TB_CLK，可通过 TIMG_T*x*_INCREASE 字段配置为递增或递减。时基计数器可通过置位或清零 TIMG_T*x*_EN 字段使能或关闭。使能时，时基计数器的值会在每个 TB_CLK 周期递增或递减。关闭时，时基计数器暂停计数。注意，TIMG_T*x*_EN 置位后，TIMG_T*x*_INCREASE 字段还可以更改，时基计数器可立即改变计数方向。

时基计数器 54 位定时器的当前值必须被锁入两个寄存器，才能被 CPU 读取（因为 CPU 为 32 位）。向 TIMG_T*x*UPDATE_REG 写入任意值时，54 位定时器的值开始被锁入寄存器 TIMG_T*x*LO_REG 和 TIMG_T*x*HI_REG，两个寄存器分别锁存低 32 位和高 22 位。当 TIMG_T*x*UPDATE_REG 被硬件清零，表明锁存操作已经完成，可以从这两个寄存器中读取当前计数值。在 TIMG_T*x*UPDATE_REG 被写入新值之前，保持寄存器 TIMG_T*x*LO_REG 和 TIMG_T*x*HI_REG 的值不变，以供 32 位的 CPU 读值。

12.2.3 报警产生

定时器可配置为在当前值与报警值相同时触发报警。报警会产生中断，（可选择）让定时器的当前值自动重新加载（详见第 12.2.4 节）。

54 位报警值可在 TIMG_T*x*ALARMLO_REG 和 TIMG_T*x*ALARMHI_REG 配置，两者分别代表报警值的低 32 位和高 22 位。但是，只有置位 TIMG_T*x*_ALARM_EN 字段使能报警功能后，配置的报警值才会生效。为解决报警使能“过晚”（即报警使能时，定时器的值已过报警值），可逆计数器向上计数时，若定时器的当前值高于报警值

(在一定范围内)，或可逆计数器向下计数时，定时器的当前值低于报警值（在一定范围内），硬件都会立即触发报警。表 12.2-1 和表 12.2-2 说明了定时器当前值、报警值与报警触发的关系。假设定时器当前值和报警值如下：

- TIMG_VALUE = {TIMG_TxHI_REG, TIMG_TxLO_REG}
- ALARM_VALUE = {TIMG_TxALARMHI_REG, TIMG_TxALARMLO_REG}

表 12.2-1. 可逆计数器向上计数时的报警触发场景

场景	范围	报警
1	ALARM_VALUE − TIMG_VALUE $> 2^{53}$	触发
2	$0 <$ ALARM_VALUE − TIMG_VALUE $\leq 2^{53}$	定时器计数器向上计数，TIMG_VALUE 达到 ALARM_VALUE 时报警
3	$0 \leq$ TIMG_VALUE − ALARM_VALUE $< 2^{53}$	触发
4	TIMG_VALUE − ALARM_VALUE $\geq 2^{53}$	定时器计数器向上计数达到最大值时，重新开始从 0 向上计数，TIMG_VALUE 达到 ALARM_VALUE 时触发报警

表 12.2-2. 可逆计数器向下计数时的报警触发场景

场景	范围	报警
5	TIMG_VALUE − ALARM_VALUE $> 2^{53}$	触发
6	$0 <$ TIMG_VALUE − ALARM_VALUE $\leq 2^{53}$	定时器计数器向下计数，TIMG_VALUE 达到 ALARM_VALUE 时报警
7	$0 \leq$ ALARM_VALUE − TIMG_VALUE $< 2^{53}$	触发
8	ALARM_VALUE − TIMG_VALUE $\geq 2^{53}$	定时器计数器向下计数达到最小值时，重新开始从最大值向下计数，TIMG_VALUE 达到 ALARM_VALUE 时触发报警

报警时，TIMG_Tx_ALARM_EN 字段自动清零，在下次置位 TIMG_Tx_ALARM_EN 前不会再次报警。

12.2.4 定时器重新加载

定时器重新加载指将定时器的低 32 位和高 22 位分别更新为寄存器 TIMG_Tx_LOAD_LO 和 TIMG_Tx_LOAD_HI 存储的重新加载值。但是，把重新加载值写入 TIMG_Tx_LOAD_LO 和 TIMG_Tx_LOAD_HI 寄存器不会改变定时器的当前值。写入的重新加载值会被定时器忽视，直到重新加载事件被触发。重新加载事件可由软件即时重新加载或报警时自动重新加载触发。

CPU 在寄存器 TIMG_TxLOAD_REG 写任意值会触发软件即时重新加载，定时器的当前值会立即改变。如置位 TIMG_Tx_EN，定时器会继续从新数值开始递增或递减计数。如清零 TIMG_Tx_EN，定时器将保持当前值，直至计数重新使能。

报警时自动重新加载功能可让定时器在报警时重新加载，从重新加载值开始继续递增或递减计数。该功能通常用于周期性报警时重置定时器的值。TIMG_Tx_AUTORELOAD 字段置 1 可以使能报警时自动重新加载。如未使能该功能，报警后定时器的值会在过报警值后继续递增或递减。

12.2.5 RTC 慢速时钟 (RTC_SLOW_CLK) 频率计算

定时器组可以通过使用 XTAL_CLK 计算低功耗时钟的三个慢速时钟源 RC_SLOW_CLK、RC_FAST_DIV_CLK 和 XTAL32K_CLK 的实际频率。计算方式如下：

1. 通过周期性或单次计算的方式启动频率计算模块；
2. 在接收到计算开始的信号后，两个分别工作在 XTAL_CLK 以及 RTC_SLOW_CLK 的计数器同时开始计数，当 RTC_SLOW_CLK 的计数器达到设定的计算周期 C0 时，同时停止两个计数器；
3. 通过 XTAL_CLK 的计数器值 C1 即可计算 RTC_SLOW_CLK 的时钟频率：$f\_rtc = \frac{C0 \times f\_XTAL\_CLK}{C1}$

12.2.6 中断

每个定时器都有一根连接至 CPU 的中断线。因此，每个定时器组有三根中断线。定时器每次产生的电平中断必须由 CPU 清除。

电平中断在报警后（或看门狗定时器阶段超时）触发。报警（或阶段超时）后，电平中断会一直被拉高，直至手动清除中断。要使能定时器的中断，TIMG_Tx_INT_ENA 需置 1。

每个定时器组的中断受一组寄存器控制。每个定时器在下列寄存器中都有对应的位：

- TIMG_Tx_INT_RAW：报警时置 1。该位在写值到对应的 TIMG_Tx_INT_CLR 位后才会被清零。
- TIMG_WDT_INT_RAW：阶段超时时置 1。该位在写值到对应的 TIMG_WDT_INT_CLR 位后才会被清零。
- TIMG_Tx_INT_ST：反映每个定时器中断的状态，通过用 TIMG_Tx_INT_ENA 屏蔽 TIMG_Tx_INT_RAW 位来生成。
- TIMG_WDT_INT_ST：反映每个看门狗定时器中断的状态，通过用 TIMG_WDT_INT_ENA 屏蔽 TIMG_WDT_INT_RAW 位来生成。
- TIMG_Tx_INT_ENA：用于使能或屏蔽组内定时器的中断状态位。
- TIMG_WDT_INT_ENA：用于使能或屏蔽组内看门狗定时器的中断状态位。
- TIMG_Tx_INT_CLR：置 1 此位清除定时器中断，定时器在 TIMG_Tx_INT_RAW 和 TIMG_Tx_INT_ST 的对应位会清零。注意，下一个中断产生前，必须清除定时器中断。
- TIMG_WDT_INT_CLR：置 1 此位清除定时器中断，看门狗定时器在 TIMG_WDT_INT_RAW 和 TIMG_WDT_INT_ST 的对应位会清零。注意，下一个中断产生前，必须清除看门狗定时器中断。

12.3 配置与使用

12.3.1 定时器用作简单时钟

1. 配置时基计数器。
   - 置位或清除 TIMG_Tx_USE_XTAL 字段选择时钟源。
   - 置位 TIMG_Tx_DIVIDER 配置 16 位预分频器。
   - 置位或清除 TIMG_Tx_INCREASE 配置定时器方向。
   - 在 TIMG_Tx_LOAD_LO 和 TIMG_Tx_LOAD_HI 上写初始值设置定时器的初始值，然后在 TIMG_TxLOAD_REG 上写任意值将初始值重新加载进定时器。

2. 置位 TIMG_Tx_EN 开启定时器。
3. 获得定时器的当前值。
   - 在 TIMG_TxUPDATE_REG 上写任意值锁存定时器的当前值。
   - 等待硬件将 TIMG_TxUPDATE_REG 清 0。
   - 从 TIMG_TxLO_REG 和 TIMG_TxHI_REG 读取锁存的定时器值。

12.3.2 定时器用于一次性报警

1. 按照第 12.3.1 节的第 1 步配置时基计数器。
2. 配置报警。
   - 置位 TIMG_TxALARMLO_REG 和 TIMG_TxALARMHI_REG 配置报警值。
   - 置位 TIMG_Tx_INT_ENA 使能中断。
3. 清零 TIMG_Tx_AUTORELOAD 关闭自动重新加载。
4. 置位 TIMG_Tx_ALARM_EN 开启报警。
5. 处理报警中断。
   - 置位定时器在 TIMG_Tx_INT_CLR 的对应位清除中断。
   - 清零 TIMG_Tx_EN 关闭定时器。

12.3.3 定时器用于周期性报警

1. 按照第 12.3.1 节的第 1 步配置时基计数器。
2. 按照第 12.3.2 节的第 2 步配置报警。
3. 置位 TIMG_Tx_AUTORELOAD 使能自动重新加载，将重新加载值写入 TIMG_Tx_LOAD_LO 和 TIMG_Tx_LOAD_HI。
4. 置位 TIMG_Tx_ALARM_EN 开启报警。
5. 处理报警中断（每次报警时重复）。
   - 置位定时器在 TIMG_Tx_INT_CLR 的对应位清除中断。
   - 如下一次报警需要新的报警值和重新加载值（即每次都有不同的报警间隔），则应根据需要重新配置 TIMG_TxALARMLO_REG、TIMG_TxALARMHI_REG、TIMG_Tx_LOAD_LO 和 TIMG_Tx_LOAD_HI 。否则，上述寄存器应保持不变。
   - 置位 TIMG_Tx_ALARM_EN 重新使能报警。
6. （最后一次报警时）关闭定时器。
   - 置位定时器在 TIMG_Tx_INT_CLR 的对应位清除中断。
   - 清零 TIMG_Tx_EN 关闭定时器。

12.3.4 RTC_SLOW_CLK 频率计算

1. 单次计算
   - 设置 TIMG_RTC_CALI_CLK_SEL 选择需要计算频率的时钟（RTC_SLOW_CLK 的时钟源），设置 TIMG_RTC_CALI_MAX 配置频率计算时间。
   - 清空 TIMG_RTC_CALI_START_CYCLING 选择单次校准模式，然后配置 TIMG_RTC_CALI_START 开启两个计数器。
   - 等待 TIMG_RTC_CALI_RDY 的值变为 1，读取 TIMG_RTC_CALI_VALUE 获取 XTAL_CLK 计数器值，计算 RTC_SLOW_CLK 频率。
2. 周期性计算
   - 设置 TIMG_RTC_CALI_CLK_SEL 选择需要计算频率的时钟（RTC_SLOW_CLK 的时钟源），设置 TIMG_RTC_CALI_MAX 配置频率计算时间。
   - 使能 TIMG_RTC_CALI_START_CYCLING，硬件将不间断进行频率计算过程。
   - 只要 TIMG_RTC_CALI_CYCLING_DATA_VLD 为 1，即表示 TIMG_RTC_CALI_VALUE 有效。
3. 超时
   如果 RTC_SLOW_CLK 的计数器没有在 TIMG_RTC_CALI_TIMEOUT_RST_CNT 的 XTAL_CLK 计数器内完成计数，将置位 TIMG_RTC_CALI_TIMEOUT 标记计算超时。

12.4 寄存器列表

本小节的所有地址均为相对于 定时器组 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
定时器 0 配置和控制寄存器			
TIMG_T0CONFIG_REG	定时器 0 配置寄存器	0x0000	varies
TIMG_T0LO_REG	定时器 0 的当前值，低 32 位	0x0004	RO
TIMG_T0HI_REG	定时器 0 的当前值，高 22 位	0x0008	RO
TIMG_T0UPDATE_REG	写值将当前定时器的值复制到 TIMG_T0LO_REG 或 TIMG_T0HI_REG	0x000C	R/ W/ SC
TIMG_T0ALARMLO_REG	定时器 0 的报警值，低 32 位	0x0010	R/W
TIMG_T0ALARMHI_REG	定时器 0 的报警值，高位	0x0014	R/W
TIMG_T0LOADLO_REG	定时器 0 的重新加载值，低 32 位	0x0018	R/W
TIMG_T0LOADHI_REG	定时器 0 的重新加载值，高 22 位	0x001C	R/W
TIMG_T0LOAD_REG	写值从 TIMG_T0LOADLO_REG 或 TIMG_T0LOADHI_REG 上加载定时器	0x0020	WT
定时器 1 配置和控制寄存器			
TIMG_T1CONFIG_REG	定时器 1 配置寄存器	0x0024	varies
TIMG_T1LO_REG	定时器 1 的当前值，低 32 位	0x0028	RO
TIMG_T1HI_REG	定时器 1 的当前值，高 22 位	0x002C	RO
TIMG_T1UPDATE_REG	写值将当前定时器的值复制到 TIMG_T1LO_REG 或 TIMG_T1HI_REG	0x0030	R/ W/ SC
TIMG_T1ALARMLO_REG	定时器 1 的报警值，低 32 位	0x0034	R/W
TIMG_T1ALARMHI_REG	定时器 1 的报警值，高位	0x0038	R/W
TIMG_T1LOADLO_REG	定时器 1 的重新加载值，低 32 位	0x003C	R/W
TIMG_T1LOADHI_REG	定时器 1 的重新加载值，高 22 位	0x0040	R/W
TIMG_T1LOAD_REG	写值从 TIMG_T1LOADLO_REG 或 TIMG_T1LOADHI_REG 上加载定时器	0x0044	WT
看门狗定时器配置和控制寄存器			
TIMG_WDTCONFIG0_REG	看门狗定时器配置寄存器	0x0048	R/W
TIMG_WDTCONFIG1_REG	看门狗定时器预分频器寄存器	0x004C	R/W
TIMG_WDTCONFIG2_REG	看门狗定时器阶段 0 超时值	0x0050	R/W
TIMG_WDTCONFIG3_REG	看门狗定时器阶段 1 超时值	0x0054	R/W
TIMG_WDTCONFIG4_REG	看门狗定时器阶段 2 超时值	0x0058	R/W
TIMG_WDTCONFIG5_REG	看门狗定时器阶段 3 超时值	0x005C	R/W
TIMG_WDTFEED_REG	写值喂看门狗定时器	0x0060	WT
TIMG_WDTWPROTECT_REG	看门狗写保护寄存器	0x0064	R/W
RTC 频率计算配置和控制寄存器			
TIMG_RTCCALICFG_REG	RTC 频率计算配置寄存器 0	0x0068	varies
TIMG_RTCCALICFG1_REG	RTC 频率计算配置寄存器 1	0x006C	RO
TIMG_RTCCALICFG2_REG	RTC 频率计算配置寄存器 2	0x0080	varies

名称	描述	地址	访问
中断寄存器			
TIMG_INT_ENA_TIMERS_REG	中断使能位	0x0070	R/W
TIMG_INT_RAW_TIMERS_REG	原始中断状态	0x0074	R/WTC/SS
TIMG_INT_ST_TIMERS_REG	屏蔽中断状态	0x0078	RO
TIMG_INT_CLR_TIMERS_REG	中断清除位	0x007C	WT
版本寄存器			
TIMG_NTIMERS_DATE_REG	版本控制寄存器	0x00F8	R/W
定时器组配置寄存器			
TIMG_REGCLK_REG	定时器组时钟门控寄存器	0x00FC	R/W

12.5 寄存器

本小节的所有地址均为相对于 定时器组 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 12.1. TIMG_TxCONFIG_REG (x: 0-1) (0x0000+0x24*x)



TIMG_Tx_USE_XTAL 0：使用 APB_CLK 作为定时器组的源时钟；1：使用 XTAL_CLK 作为定时器组的源时钟。(R/W)

TIMG_Tx_ALARM_EN 置 1 后，报警使能。报警时，此位自动清零。(R/W/SC)

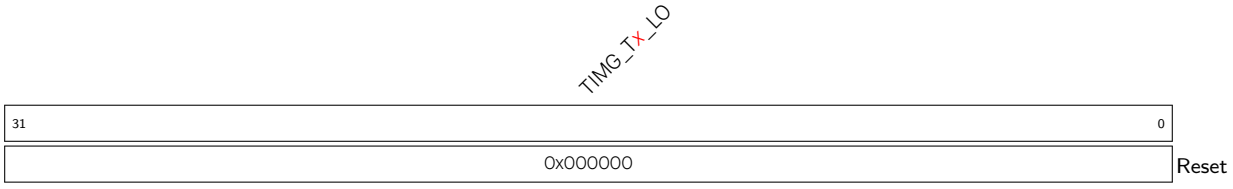
TIMG_Tx_DIVIDER 定时器 x 时钟 (Tx_clk) 的预分频器值。(R/W)

TIMG_Tx_AUTORELOAD 置 1 后，定时器 x 报警时自动重新加载使能。(R/W)

TIMG_Tx_INCREASE 置 1 后，定时器 x 的时基计数器会在每个时钟周期后递增。清零后，定时器 x 的时基计数器会递减。(R/W)

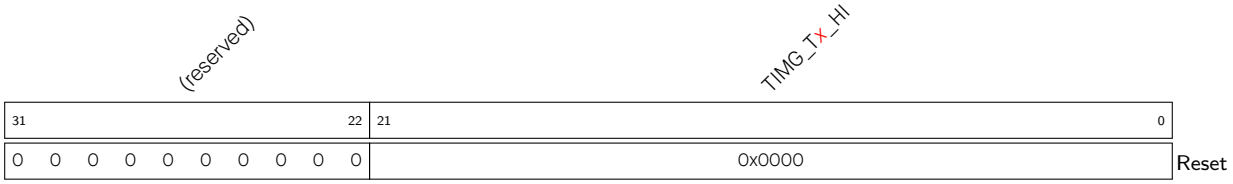
TIMG_Tx_EN 置 1 后，定时器 x 时基计数器使能。(R/W)

Register 12.2. TIMG_TxLO_REG (x: 0-1) (0x0004+0x24*x)



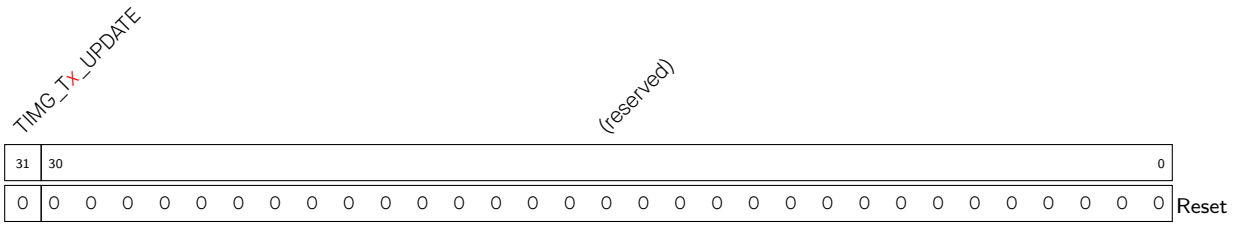
TIMG_Tx_LO 在 TIMG_TxUPDATE_REG 上写值后，可读取定时器 x 时基计数器的低 32 位。(RO)

Register 12.3. TIMG_TxHI_REG (x: 0-1) (0x0008+0x24*x)



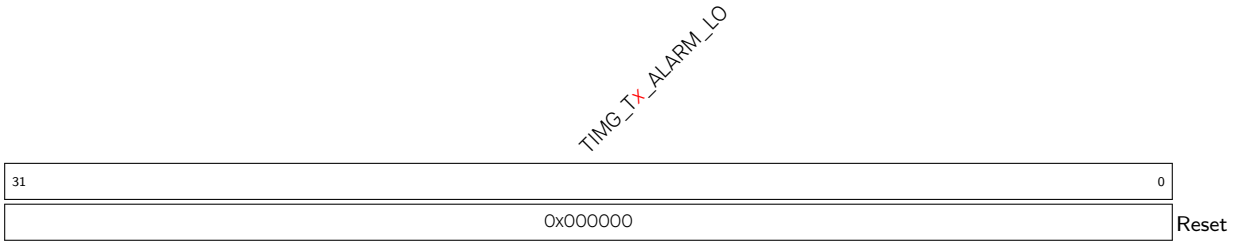
TIMG_Tx_HI 在 TIMG_TxUPDATE_REG 上写值后，可读取定时器 x 时基计数器的高 22 位。(RO)

Register 12.4. TIMG_TxUPDATE_REG (x: 0-1) (0x000C+0x24*x)



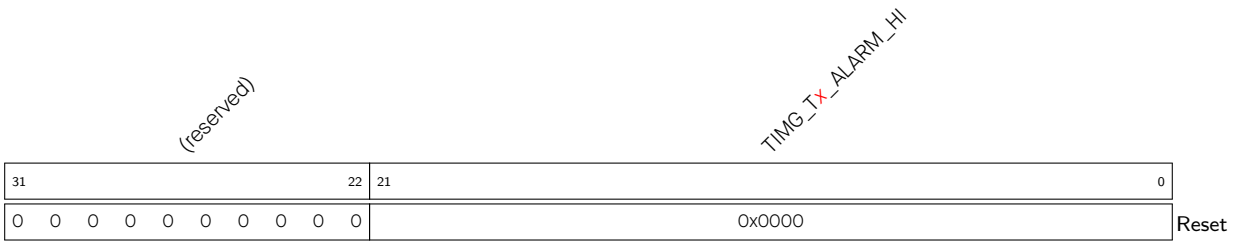
TIMG_Tx_UPDATE 在 TIMG_TxUPDATE_REG 上写 0 或 1，计数器的值被锁住。(R/W/SC)

Register 12.5. TIMG_TxALARMLO_REG (x: 0-1) (0x0010+0x24*x)



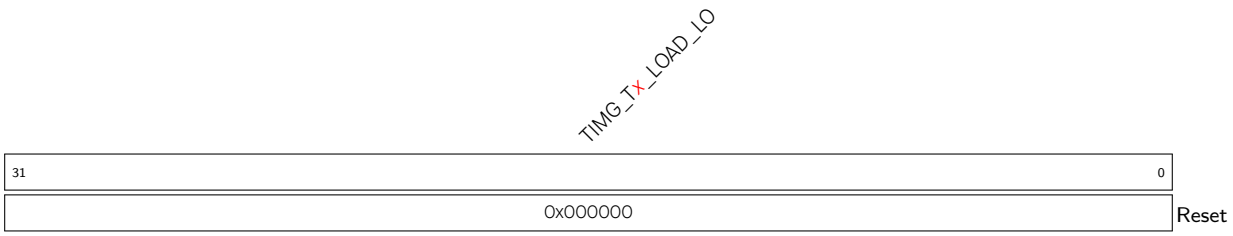
TIMG_Tx_ALARM_LO 定时器 x 时基计数器触发警报值的低 32 位。(R/W)

Register 12.6. TIMG_TxALARMHI_REG (x: 0-1) (0x0014+0x24*x)



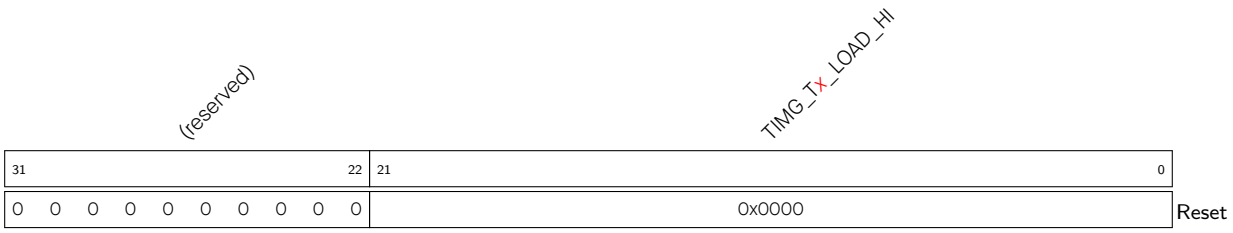
TIMG_Tx_ALARM_HI 定时器 x 时基计数器触发警报值的高 22 位。(R/W)

Register 12.7. TIMG_TxLOADLO_REG (x: 0-1) (0x0018+0x24*x)



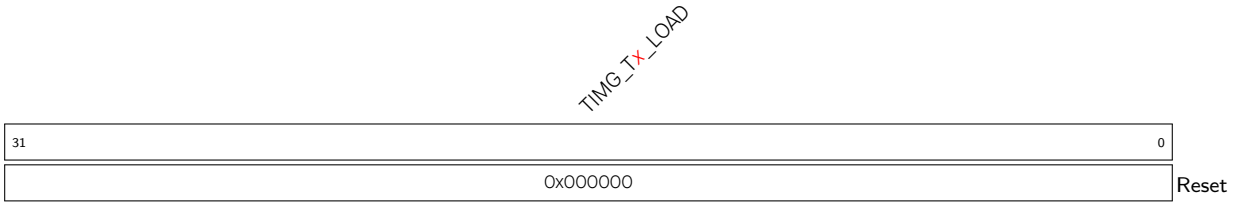
TIMG_Tx_LOAD_LO 定时器 x 时基计数器重新加载的低 32 位值。(R/W)

Register 12.8. TIMG_TxLOADHI_REG (x: 0-1) (0x001C+0x24*x)



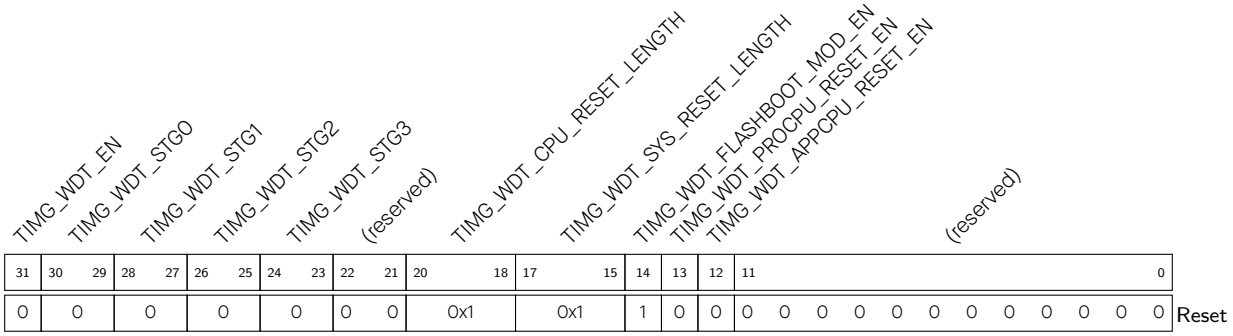
TIMG_Tx_LOAD_HI 定时器 x 时基计数器重新加载的高 22 位值。(R/W)

Register 12.9. TIMG_TxLOAD_REG (x: 0-1) (0x0020+0x24*x)



TIMG_Tx_LOAD 写任意值触发定时器 x 时基计数器重新加载。(WT)

Register 12.10. TIMG_WDTCONFIG0_REG (0x0048)



TIMG_WDT_APPCPU_RESET_EN 保留。(R/W)

TIMG_WDT_PROCPU_RESET_EN WDT 复位 CPU 使能。(R/W)

TIMG_WDT_FLASHBOOT_MOD_EN 置 1 后, flash 启动保护使能。(R/W)

TIMG_WDT_SYS_RESET_LENGTH 系统复位信号长度选择。0：100 ns；1：200 ns；2：300 ns；3：400 ns；4：500 ns；5：800 ns；6：1.6 μs；7：3.2 μs。(R/W)

TIMG_WDT_CPU_RESET_LENGTH CPU 复位信号长度选择。0：100 ns；1：200 ns；2：300 ns；3：400 ns；4：500 ns；5：800 ns；6：1.6 μs；7：3.2 μs。(R/W)

TIMG_WDT_STG3 阶段 3 配置。0：关闭；1：中断；2：复位 CPU；3：复位系统。(R/W)

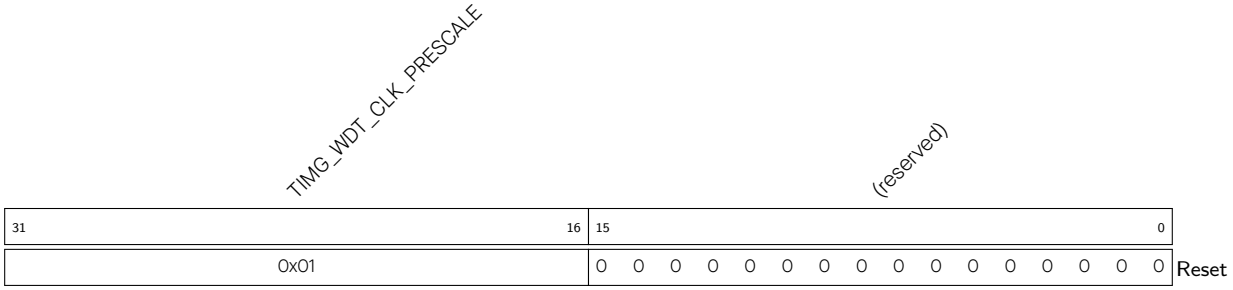
TIMG_WDT_STG2 阶段 2 配置。0：关闭；1：中断；2：复位 CPU；3：复位系统。(R/W)

TIMG_WDT_STG1 阶段 1 配置。0：关闭；1：中断；2：复位 CPU；3：复位系统。(R/W)

TIMG_WDT_STG0 阶段 0 配置。0：关闭；1：中断；2：复位 CPU；3：复位系统。(R/W)

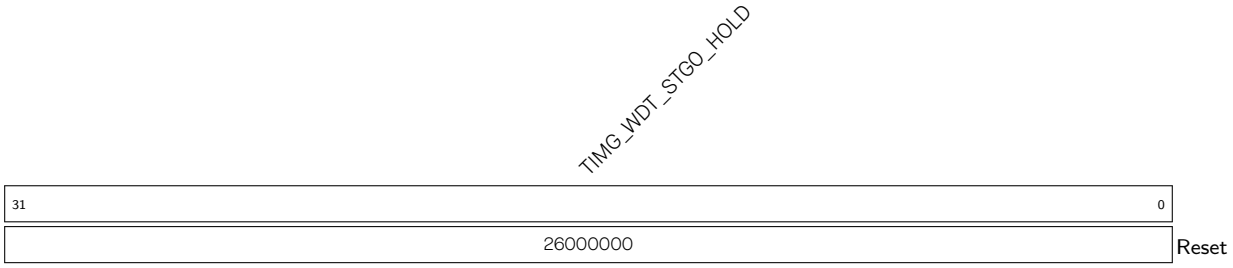
TIMG_WDT_EN 置 1 后，MWDT 使能。(R/W)

Register 12.11. TIMG_WDTCONFIG1_REG (0x004C)



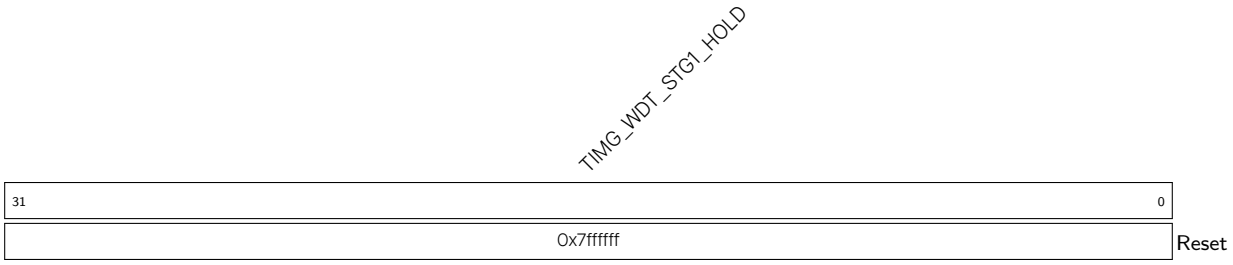
TIMG_WDT_CLK_PRESCALE MWDT 时钟预分频器值。MWDT 时钟周期 = MWDT 时钟源周期 * TIMG_WDT_CLK_PRESCALE。(R/W)

Register 12.12. TIMG_WDTCONFIG2_REG (0x0050)



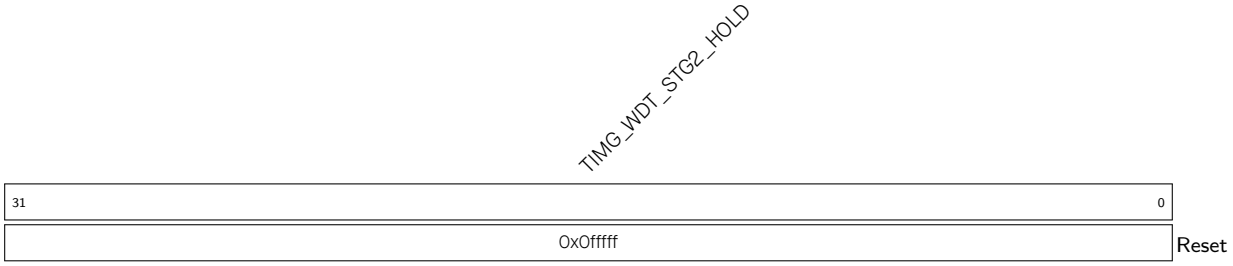
TIMG_WDT_STG0_HOLD 阶段 0 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 12.13. TIMG_WDTCONFIG3_REG (0x0054)



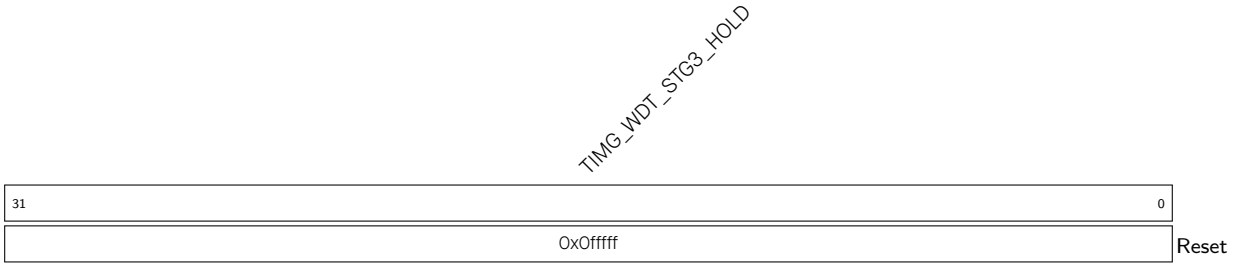
TIMG_WDT_STG1_HOLD 阶段 1 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 12.14. TIMG_WDTCONFIG4_REG (0x0058)



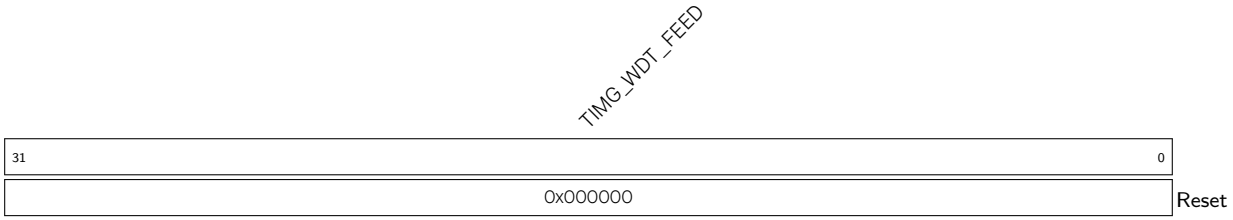
TIMG_WDT_STG2_HOLD 阶段 2 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 12.15. TIMG_WDTCONFIG5_REG (0x005C)



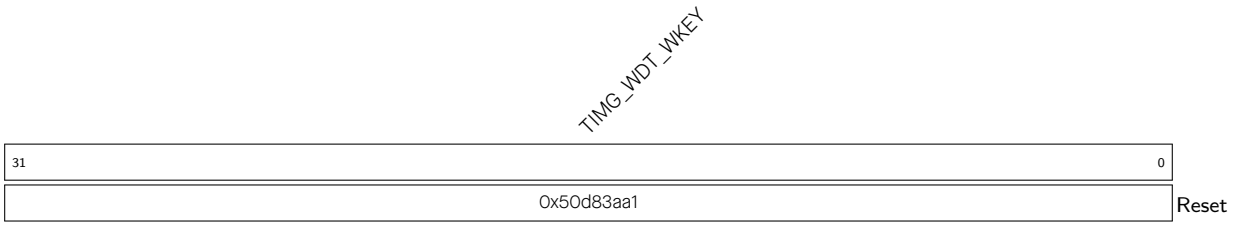
TIMG_WDT_STG3_HOLD 阶段 3 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 12.16. TIMG_WDTFEED_REG (0x0060)



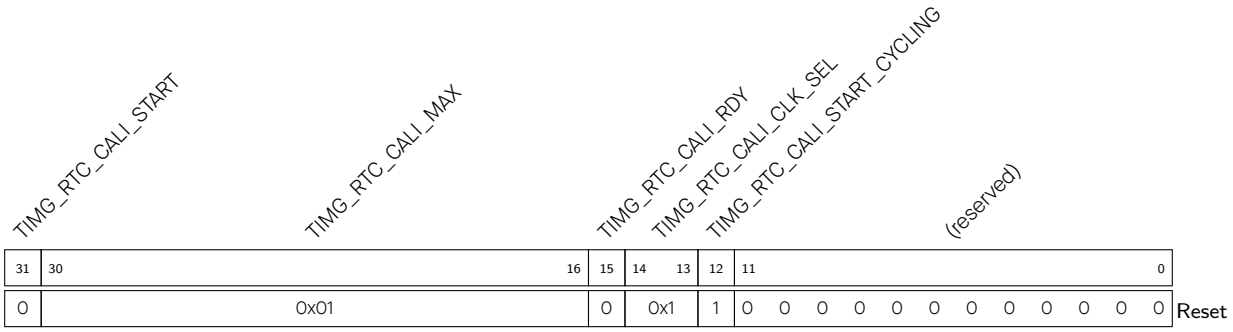
TIMG_WDT_FEED 写任意值喂 MWDT。(WT)

Register 12.17. TIMG_WDTWPROTECT_REG (0x0064)



TIMG_WDT_WKEY 如果寄存器的值与复位值不同，写保护使能。(R/W)

Register 12.18. TIMG_RTCCALICFG_REG (0x0068)



TIMG_RTC_CALI_START_CYCLING 使能周期性频率计算。(R/W)

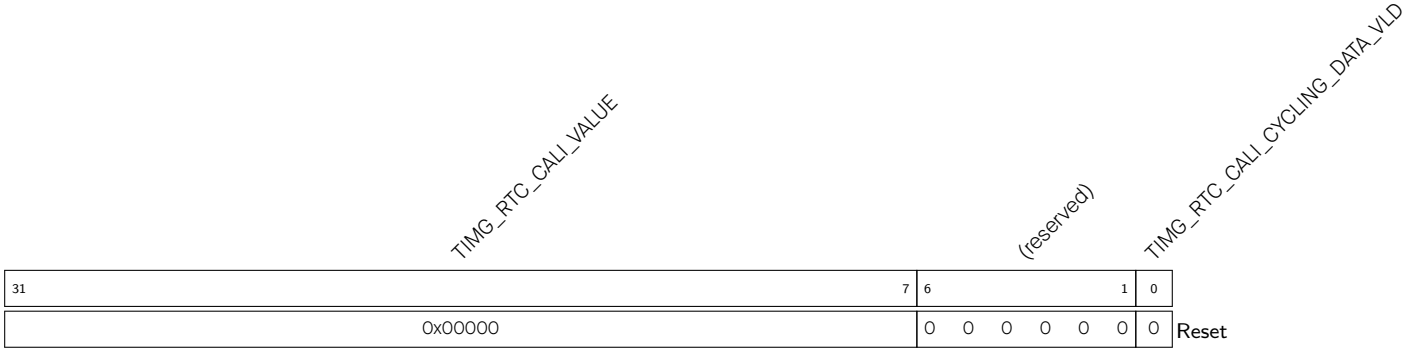
TIMG_RTC_CALI_CLK_SEL 选择待校准时钟。0：RC_SLOW_CLK；1：RC_FAST_DIV_CLK；2：XTAL32K_CLK。(R/W)

TIMG_RTC_CALI_RDY 标记单次频率计算完成。(RO)

TIMG_RTC_CALI_MAX 配置计算 RTC 慢速时钟 RTC_SLOW_CLK 频率的时间。单位：RTC_SLOW_CLK 时钟周期。(R/W)

TIMG_RTC_CALI_START 置位此位，使能单次频率计算。(R/W)

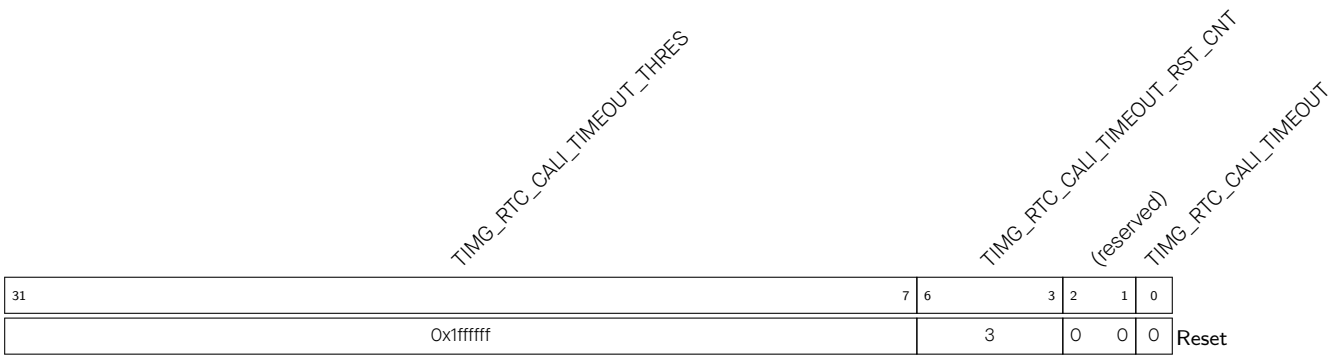
Register 12.19. TIMG_RTCCALICFG1_REG (0x006C)



TIMG_RTC_CALI_CYCLING_DATA_VLD 周期性频率计算结束标志。(RO)

TIMG_RTC_CALI_VALUE 单次或周期性频率计算完成时，读取此位计算 RTC 慢速时钟 RTC_SLOW_CLK 的频率。单位：XTAL_CLK 时钟周期。(RO)

Register 12.20. TIMG_RTCCALICFG2_REG (0x0080)

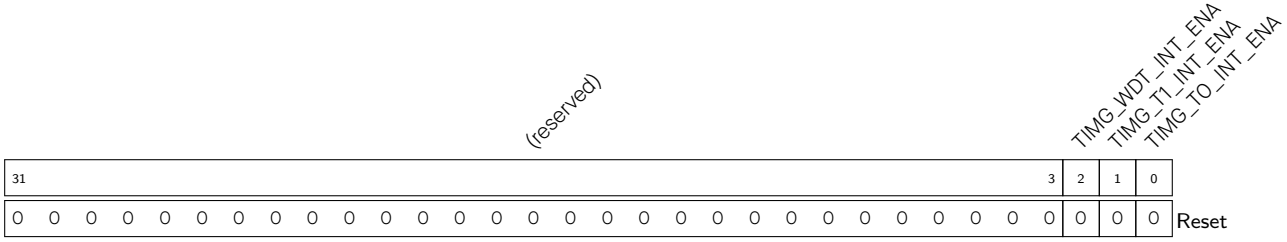


TIMG_RTC_CALI_TIMEOUT 提示时钟频率计算超时。(RO)

TIMG_RTC_CALI_TIMEOUT_RST_CNT 频率计算超时复位周期。(R/W)

TIMG_RTC_CALI_TIMEOUT_THRES RTC 频率计算定时器的阈值。频率计算定时器的值超过此值时触发超时。(R/W)

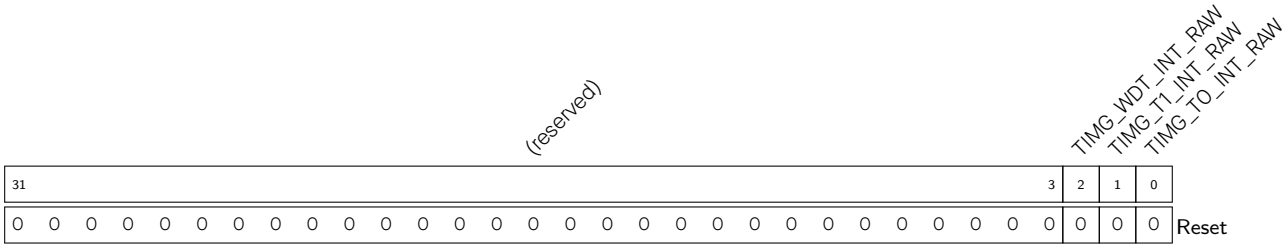
Register 12.21. TIMG_INT_ENA_TIMERS_REG (0x0070)



TIMG_T*x*_INT_ENA TIMG_T*x*_INT 中断的使能位。(R/W)

TIMG_WDT_INT_ENA TIMG_WDT_INT 中断的使能位。(R/W)

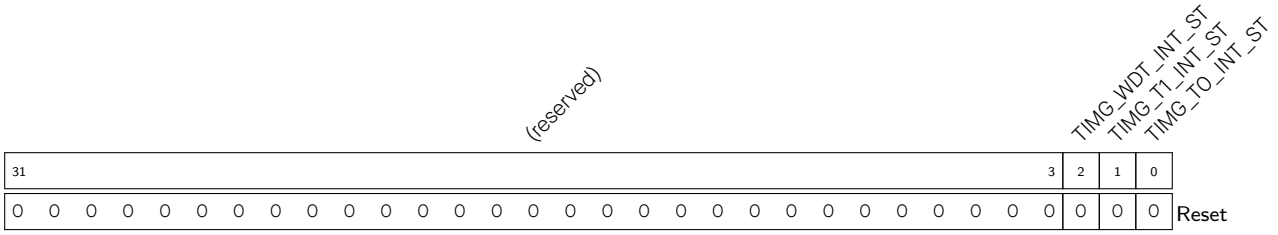
Register 12.22. TIMG_INT_RAW_TIMERS_REG (0x0074)



TIMG_T*x*_INT_RAW TIMG_T*x*_INT 中断的原始中断状态位。(R/WTC/SS)

TIMG_WDT_INT_RAW TIMG_WDT_INT 中断的原始中断状态位。(R/WTC/SS)

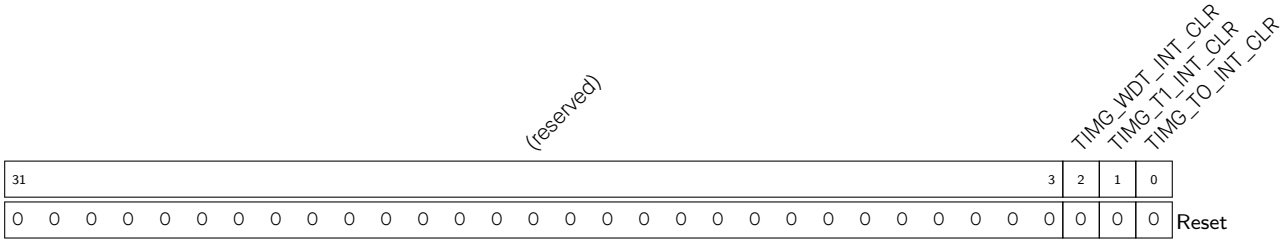
Register 12.23. TIMG_INT_ST_TIMERS_REG (0x0078)



TIMG_T*x*_INT_ST TIMG_T*x*_INT 中断的屏蔽中断状态位。(RO)

TIMG_WDT_INT_ST TIMG_WDT_INT 中断的屏蔽中断状态位。(RO)

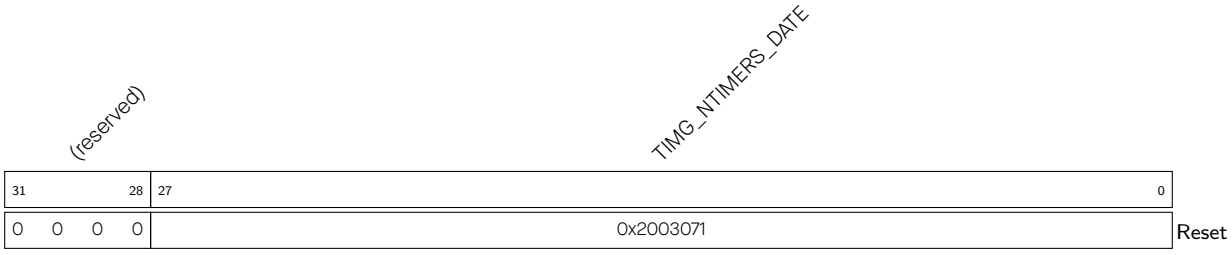
Register 12.24. TIMG_INT_CLR_TIMERS_REG (0x007C)



TIMG_T*x*_INT_CLR 置位此位，清除 TIMG_T*x*_INT 中断。(WT)

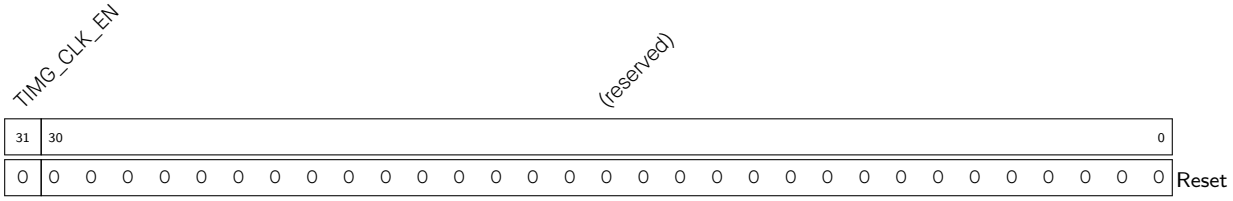
TIMG_WDT_INT_CLR 置位此位，清除 TIMG_WDT_INT 中断。(WT)

Register 12.25. TIMG_NTIMERS_DATE_REG (0x00F8)



TIMG_NTIMERS_DATE 版本控制寄存器。(R/W)

Register 12.26. TIMG_REGCLK_REG (0x00FC)



TIMG_CLK_EN 寄存器时钟门控信号。0：仅在软件运行时打开读写寄存器所需的时钟；1：一直开启软件读写寄存器所需时钟。(R/W)

第 13 章

看门狗定时器 (WDT)

13.1 概述

看门狗定时器是一种硬件定时器，用于检测和修复故障。软件必须定期喂狗（复位），以防超时。系统或软件若出现不可预知的问题（比如软件卡在某个循环或逾期事件中）将无法按时喂狗，造成看门狗超时。因此，看门狗定时器有助于检测、处理系统或软件的错误行为。

如图 13.1-1 所示，ESP32-S3 中有三个数字看门狗定时器：章节 12 定时器组 *(TIMG)* 描述的两个定时器组中各有一个（称作主系统看门狗定时器，缩写为 MWDT），RTC 模块中有一个（称作 RTC 看门狗定时器，缩写为 RWDT）。数字看门狗在运行期间会经历四个阶段（除非看门狗按时喂狗或者处于关闭状态），每个阶段均可配置单独的超时时间和超时动作，其中 MWDT 支持中断、CPU 复位和内核复位三种超时动作，RWDT 支持中断、CPU 复位、内核复位和系统复位四种超时动作（详见章节 13.2.2.2 阶段与超时动作）。每个阶段的超时时间都可单独设置。

在 flash 引导模式下，RWDT 和定时器组 0 的 MWDT 会默认使能，以检测引导过程中发生的错误，并恢复运行。

ESP32-S3 中还有一个模拟看门狗定时器——超级看门狗 (SWD)。超级看门狗是模拟域的超低功耗电路，可以防止系统在数字电路异常状态下运行，并在必要时复位系统。

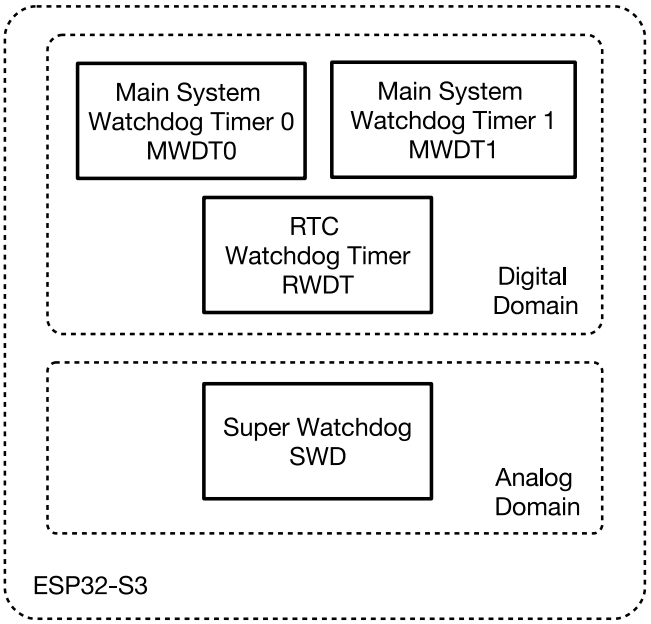


图 13.1-1. 看门狗定时器概览

请注意，本章节仅包含看门狗定时器的功能描述，其寄存器部分详见章节 12 定时器组 *(TIMG)* 和章节 10 低功耗管理 *(RTC_CNTL)*。

13.2 数字看门狗定时器

13.2.1 主要特性

看门狗定时器具有如下特性：

- 四个阶段，每个阶段都可配置超时时间。每阶段都可单独配置、使能和关闭
- 如在某个阶段发生超时，MWDT 会采取中断、CPU 复位和内核复位中的一种超时动作，RWDT 则会采取中断、CPU 复位、内核复位和系统复位中的一种超时动作
- 32 位超时计数器
- 写保护，防止 RWDT 和 MWDT 配置误改动
- flash 启动保护
  如果在预定时间内 SPI flash 的引导过程没有完成，看门狗会重启整个主系统

13.2.2 功能描述

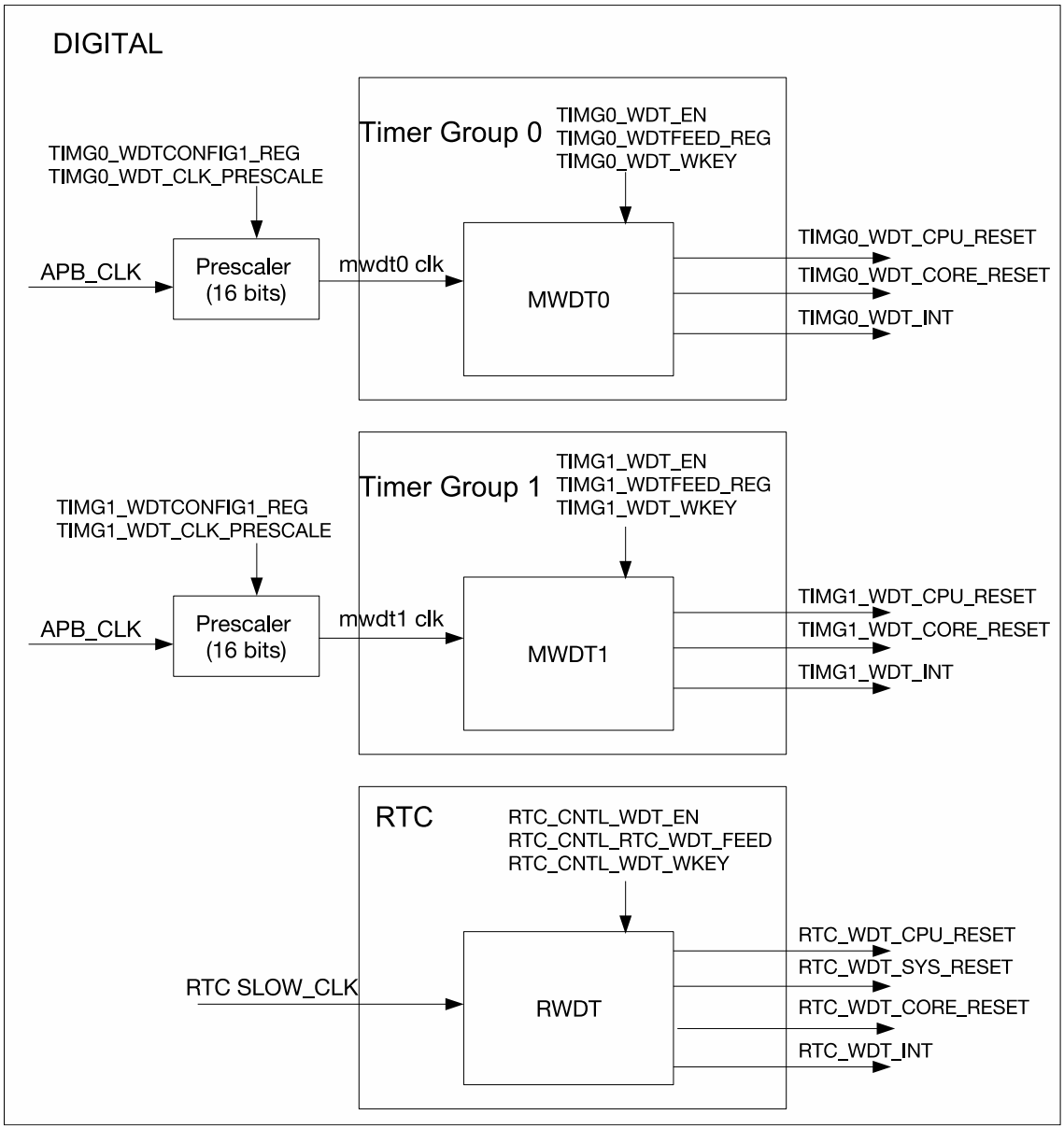


图 13.2-1. ESP32-S3 的看门狗定时器

图 13.2-1 为 ESP32-S3 数字系统中的三个看门狗定时器。

13.2.2.1 时钟源与 32 位计数器

每个看门狗定时器的核心是一个 32 位计数器。APB 时钟经过可配置的 16 位预分频器后会得到 MWDT 的时钟源。RWDT 的时钟源则直接取自于 RTC 慢速时钟（RTC 慢速时钟源详见章节 7 复位和时钟）。MWDT 的 16 位预分频器可通过 TIMG_WDTCONFIG1_REG 寄存器的 TIMG_WDT_CLK_PRESCALE 字段配置。

MWDT 和 RWDT 看门狗可分别通过设置 TIMG_WDT_EN 和 RTC_CNTL_WDT_EN 字段使能。看门狗使能后，其内部 32 位计数器的值会在每个时钟源周期内累加 1，直到达到该阶段的超时时间（即在该阶段发生超时）。如发生超时，计数器的值会重置为 0，同时看门狗进入下一阶段。如果软件在规定的时间内成功喂狗，看门狗定时器会回到阶段 0，并将计数器的值重置为 0。软件向 TIMG_WDTFEED_REG 和 RTC_CNTL_RTC_WDT_FEED 寄存器内写入任意值，便可分别为 MDWT 和 RWDT 喂狗。

13.2.2.2 阶段与超时动作

定时器在各阶段可以配置不同的超时时间和对应的超时动作。某一阶段超时会触发对应的超时动作，同时计数器的值被重置为 0，看门狗进入下一阶段。MWDT 和 RWDT 有四个阶段（称为阶段 0 至阶段 3）。看门狗定时器会循环工作（即从阶段 0 至阶段 3，再回到阶段 0）。

MWDT 每个阶段的超时时间可用 TIMG_WDTCONFIG*i*_REG（i 的范围是 2 到 5）寄存器配置，RWDT 的超时时间可用 RTC_CNTL_WDT_STG*j*_HOLD（j 的范围是 0 到 3）字段配置。

值得注意的是，RWDT 在阶段 0 的超时时间 ($Thold_0$) 受 eFuse 寄存器 EFUSE_RD_REPEAT_DATA1_REG 的
EFUSE_WDT_DELAY_SEL 字段和 RTC_CNTL_WDT_STG0_HOLD 字段共同影响，关系如下：

$$T_{hold0} = RTC\_CNTL\_WDT\_STG0\_HOLD << (EFUSE\_WDT\_DELAY\_SEL + 1)$$

其中，<< 为左移运算符。

如某个阶段超时，下列超时动作之一将会执行：

- 触发中断
  如阶段超时，中断被触发。
- CPU 复位 – 复位 CPU 核心
  如阶段超时，复位 CPU 核心。
- 内核复位 – 复位主系统
  如阶段超时，主系统（包括 MWDT、CPU 和所有外设）复位。功耗管理单元和 RTC 外设不会复位。
- 系统复位 – 复位主系统、功耗管理单元和 RTC 外设
  如阶段超时，主系统、功耗管理单元和 RTC 外设（详见章节 10 *低功耗管理 (RTC_CNTL)*）同时复位。此动作仅可在 RWDT 中实现。
- 关闭
  该阶段对系统不产生影响。

MWDT 所有阶段的超时动作均在 TIMG_WDTCONFIG0_REG 寄存器中配置。RWDT 的超时动作可在 RTC_CNTL_WDTCONFIG0_REG 寄存器配置。

13.2.2.3 写保护

看门狗定时器对于检测和处理系统或软件错误而言至关重要，不应轻易关闭（例如，因写寄存器位置错误而误将看门狗关闭）。因此，MWDT 和 RWDT 引入写保护机制，防止看门狗因无意的写操作而被关闭或篡改。

写保护机制通过每个看门狗定时器的写密钥字段运行（MWDT 看门狗使用 TIMG_WDT_WKEY，RWDT 看门狗使用 RTC_CNTL_WDT_WKEY）。必须向看门狗定时器的写密钥字段写入 0x50D83AA1，才能修改其它看门狗寄存器。如果写密钥字段的值不是 0x50D83AA1，任何试图向看门狗定时器寄存器（除了向写密钥字段本身）写值的操作都会被忽略。推荐按以下步骤访问看门狗定时器：

1. 将 0x50D83AA1 写入看门狗定时器的写密钥字段，关闭写保护。
2. 根据需要修改看门狗，如喂狗或改变配置。
3. 向看门狗定时器的写密钥字段上写入除 0x50D83AA1 以外的任意值，重新使能写保护。

13.2.2.4 flash 引导保护

在 flash 引导模式下，定时器组 0（见图 12.1-1 定时器组）的 MWDT 和 RWDT 会默认使能。MWDT 的阶段 0 的默认超时动作为内核复位（复位主系统）。RWDT 的阶段 0 超时动作为系统复位（复位主系统和 RTC）。引导后，应将 TIMG_WDT_FLASHBOOT_MOD_EN 和 RTC_CNTL_WDT_FLASHBOOT_MOD_EN 位清零，分别关闭 MWDT 和 RWDT 的 flash 引导保护。然后，软件可以配置 MWDT 和 RWDT。

13.3 模拟看门狗定时器

超级看门狗 (SWD) 是模拟域的超低功耗电路，可以防止系统在数字电路异常状态下运行，并在必要时复位系统。SWD 包含一个看门狗电路，需在每个超时阶段（约不足一秒）至少喂狗一次。该电路会在看门狗超时时间约 100 ms 之前发送 WD_INTR 信号提醒系统喂狗。

如果系统不回应 SWD 的喂狗请求，看门狗超时，SWD 会产生系统电平信号 SWD_RSTB，复位芯片上的整个数字电路。

13.3.1 主要特性

SWD 具有如下特性：

- 超低功耗
- 用中断提醒 SWD 即将超时
- 软件有多种专用的方法喂 SWD，让 SWD 监控整个操作系统的工作状态

13.3.2 SWD 控制器

13.3.2.1 结构

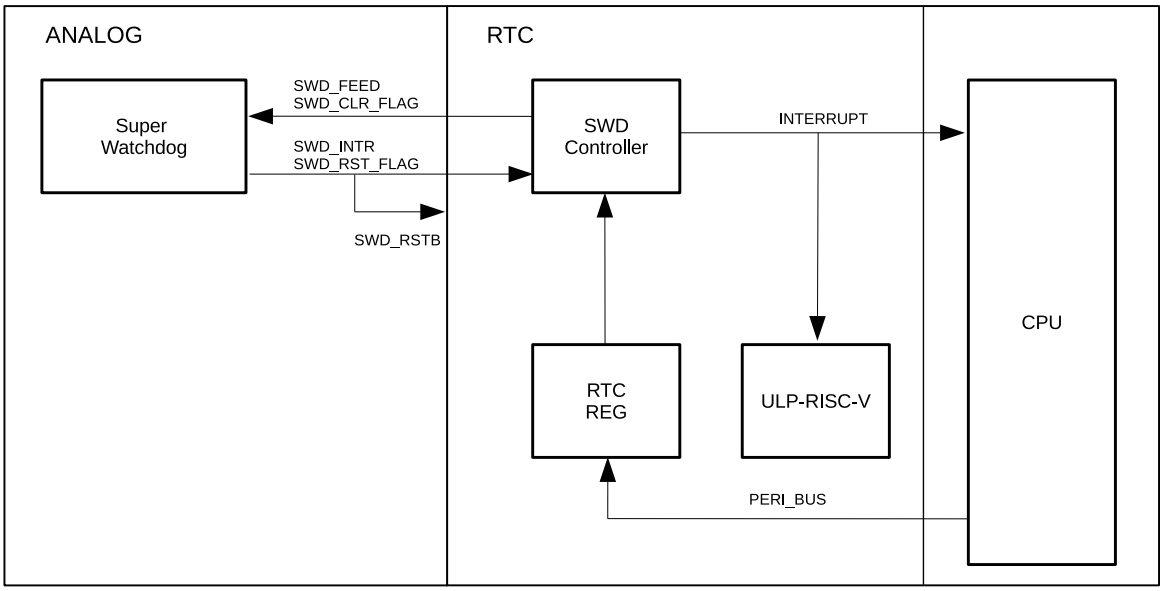


图 13.3-1. SWD 控制器结构

13.3.2.2 工作流程

正常状态下：

- SWD 控制器收到 SWD 的喂狗请求。
- SWD 控制器可以向主 CPU 或 ULP-RISC-V 发送中断。
- 主 CPU 可以决定是通过置位 RTC_CNTL_SWD_FEED 直接喂狗，还是发送中断让 ULP-RISC-V 置位该字段喂狗。
- CPU 或 ULP-RISC-V 喂狗时，需要先向 RTC_CNTL_SWD_WKEY 写 0x8F1D312A 关闭 SWD 控制器的写保护。这样做可以防止系统在数字电路异常状态下运行时误喂 SWD。
- 如将 RTC_CNTL_SWD_AUTO_FEED_EN 置 1，SWD 控制器也可配置为在不需要 CPU 或 ULP-RISC-V 干预的情况下喂 SWD。

复位后：

- 可查看 RTC_CNTL_RESET_CAUSE_PROCPU[5:0] 获知 CPU 复位原因。
  如 RTC_CNTL_RESET_CAUSE_PROCPU[5:0] == 0x12，则表示上一次复位的原因是 SWD 复位。
- 置位 RTC_CNTL_SWD_RST_FLAG_CLR 清除 SWD 复位标志。

13.4 中断

看门狗定时器中断，请前往章节 12 定时器组 *(TIMG)* 的第 12.2.6 节 中断 查看。

13.5 寄存器

MWDT 寄存器是定时器组模块的一部分，在章节 12 定时器组 *(TIMG)* 的第 12.4 节 寄存器列表 中有详细描述。RWDT 和 SWD 寄存器是 RTC 模块的一部分，在章节 10 低功耗管理 *(RTC_CNTL)* 的第 10.7 节 寄存器列表 中有详细描述。

第 14 章

XTAL32K 看门狗定时器 (XTWDT)

ESP32-S3 的 XTAL32K 看门狗定时器是用于检测 XTAL32K_CLK 时钟的工作状态，有 XTAL32K_CLK 停振监测，切换 RTC 时钟源等功能。当外部晶振 XTAL32K_CLK 作为 RTC_SLOW_CLK 源（时钟描述详见章节 7 复位和时钟），若 XTAL32K_CLK 时钟停振，XTAL32K 看门狗定时器会将 XTAL32K_CLK 替换为 RC_SLOW_CLK 的分频时钟 BACKUP32K_CLK 并发送中断（若芯片处于 Light-sleep 和 Deep-sleep 状态则唤醒 CPU），由软件重启 XTAL32K_CLK，并切回。

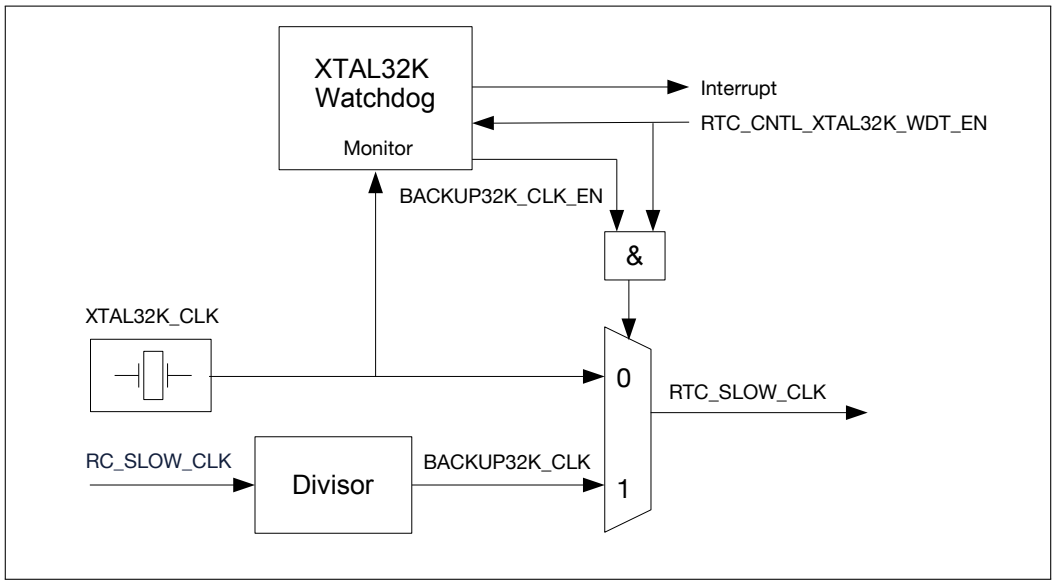


图 14.0-1. XTAL32K 看门狗定时器

14.1 主要特性

14.1.1 XTAL32K 看门狗定时器的中断及唤醒

XTAL32K 看门狗定时器监控到 XTAL32K_CLK 停振时，将发起停振中断 RTC_XTAL32K_DEAD_INT（中断描述详见章节 10 低功耗管理 *(RTC_CNTL)*），，如果 CPU 处于 Light-sleep 和 Deep-sleep 状态，将唤醒 CPU。

14.1.2 BACKUP32K_CLK

XTAL32K 看门狗定时器监控到 XTAL32K_CLK 停振后，将使用 RC_SLOW_CLK 的分频时钟 BACKUP32K_CLK（频率约为 32 kHz）替代 XTAL32K_CLK 作为 RTC_SLOW_CLK 维持系统继续正常工作。

14.2 功能描述

14.2.1 工作流程

1. 使能 RTC_CNTL_XTAL32K_WDT_EN，XTAL32K 看门狗定时器将由空闲状态转入计数状态，看门狗的计数器（工作时钟为 RC_SLOW_CLK）在检测到 XTAL_32K 的时钟上升沿时将被清零，否则将持续计数；当计数器的值达到 RTC_CNTL_XTAL32K_WDT_TIMEOUT 时，发出中断/唤醒信号，随后计数器复位。
2. 如果 RTC_CNTL_XTAL32K_AUTO_BACKUP 已置 1 且步骤 1 已完成，XTAL32K 看门狗定时器会自动开启 BACKUP32K_CLK，替换 RTC 的慢速时钟源 RTC_SLOW_CLK，保证系统能够正常运行，以及工作在 RTC 慢速时钟 RTC_SLOW_CLK 的定时器（如 RTC_TIMER 等）能够保持计时准确性。时钟频率的配置参见 14.2.2。
3. 软件通过 RTC_CNTL_XPD_XTAL_32K 位开关 XTAL32K_CLK 的 XPD（no power-down 的缩写，意为不关闭电源）信号来重启 XTAL32K_CLK，然后通过将 RTC_CNTL_XTAL32K_WDT_EN 位设置 0（BACKUP32K_CLK_EN 会随之自动清零），RTC_SLOW_CLK 时钟源将从 BACKUP32K_CLK 切回到 XTAL32K_CLK。若是芯片处于 Light-sleep 和 Deep-sleep 状态，则 XTAL32K 看门狗定时器将唤醒 CPU，完成上述操作。

14.2.2 BACKUP32K_CLK 实现原理

由于 RC_SLOW_CLK 的时钟频率存在芯片差异，所以为保证 BACKUP32K_CLK 生效期间，RTC_TIMER 等使用 RTC_SLOW_CLK 工作的定时器依然能够准确计时，需要根据 RC_SLOW_CLK（详见章节 10 低功耗管理 (RTC_CNTL)）的实际频率，可通过配置 RTC_CNTL_XTAL32K_CLK_FACTOR_REG 调整 BACKUP32K_CLK 的分频系数。该寄存器的每个字节对应一个分频因子 (x_0 ~x_7)。BACKUP32K_CLK 的分频器是一个分母恒为 4 的小数分频器，算法如下：

$$f\_back\_clk/4 = f\_rc\_slow\_clk/S$$
$$S = x_0 + x_1 + ... + x_7$$

其中 f_back_clk 为分频后的 BACKUP32K_CLK 目标频率为 32.768 kHz；f_rc_slow_clk 为当前 RC_SLOW_CLK 的实际频率；x_0 ~x_7 分别对应四个 BACKUP32K 时钟信号的高低电平的脉宽，单位为 RC_SLOW_CLK 的周期.

14.2.3 BACKUP32K_CLK 分频因子配置方法

根据 14.2.2 小节的分频原理描述，可以通过以下步骤计算并完成分频因子的配置：

- 根据 RC_SLOW_CLK 的频率以及 BACKUP32K 的目标分频频率计算出分频因子的总和 S；
- 计算出分频器的整数部分，$N = f\_rc\_slow\_clk/f\_back\_clk$；
- 因为 BACKUP32K 的分频因子是单个脉宽（高或低电平），所以需要将分频系数的整数部分分成两份，计算分频因子的整数部分，$M = N/2$；
- 根据 M 和 S 确定 xn = M 以及 xn = M + 1 的个数，M + 1 即为分频因子的小数部分。

例如，RC_SLOW_CLK 的时钟频率为 163 kHz，则 $f\_rc\_slow\_clk = 163000$，$f\_back\_clk = 32768$，$S = 20$，$M = 2$，所以满足条件的 $\{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\} = \{2, 3, 2, 3, 2, 3, 2, 3\}$，BACKUP32K_CLK 的时钟频率为 32.6 kHz。

第 15 章

权限控制 (PMS)

15.1 概述

ESP32-S3 中所有的片内存储器、片外存储器、及外设均支持访问权限管理，CPU 必须拥有相应访问权限才能访问相应的从设备，从而保护数据和指令不被非法读取、改写和取指。

特别地，ESP32-S3 允许用户使能 World 控制，将芯片的资源分为安全世界资源和非安全世界资源，CPU 处于不同世界时可以访问的权限不同，这部分内容请见章节 16 *World 控制器 (WCL)*。本章节将主要介绍 ESP32-S3 的权限管理机制及相关保护措施。

15.2 主要特性

ESP32-S3 的权限控制具有以下特性：

- 安全世界和非安全世界可独立配置
- 支持片内存储器的权限管理，包括：
  - CPU 对片内存储器的访问权限控制
  - CPU Trace 对片内存储器的访问权限控制
  - GDMA 对片内存储器的访问权限控制
- 支持片外存储器的权限管理
  - SPI1 访问外部存储器的权限控制
  - GDMA 访问外部存储器的权限控制
  - CPU 通过 CACHE 访问外部存储器的权限控制
- 支持外设的权限管理
  - 绝大部份外设空间均支持独立的权限控制
  - 支持非对齐访问的监测
  - 支持自定义地址段权限管理
- 内置权限寄存器锁保护机制
  - 所有的权限寄存器都能够通过 lock 寄存器进行锁定，一旦权限寄存器被 lock 寄存器锁住，该权限寄存器以及 lock 寄存器都无法再次被修改，直到 CPU 复位才能解除锁定
- 内置权限监测中断机制
- 内置 CPU 内部 VECBASE 寄存器覆盖机制

15.3 片内存储器的权限管理

ESP32-S3 片内存储器的主要包括：

- ROM：共 384 KB，分为 256 KB Internal ROM0 和 128 KB Internal ROM1。
- SRAM：共 512 KB，分为 32 KB Internal SRAM0、416 KB Internal SRAM1 和 64 KB Internal SRAM2。
- RTC 快速内存：共 8 KB，可由分割线分为高、低两个片区，各片区可独立配置权限。
- RTC 慢速内存：共 8 KB，可由分割线分为高、低两个片区，各片区可独立配置权限。

本节将依次介绍 ESP32-S3 中有关片内存储资源访问权限管理。

15.3.1 ROM 的访问权限管理

15.3.1.1 地址范围

ESP32-S3 的 ROM 允许 CPU 通过指令总线 IBUS 和数据总线 DBUS 进行访问，安全世界和非安全世界的权限可单独配置，但一旦配置对 CPU0 和 CPU1 同时生效，具体允许访问的地址空间范围请见下方表 15.3-1。

表 15.3-1. ROM 地址范围

ROM	IBUS 地址		DBUS 地址	
	起始地址	结束地址	起始地址	结束地址
Internal ROM0	0x4000_0000	0x4003_FFFF	-	-
Internal ROM1	0x4004_0000	0x4005_FFFF	0x3FF0_0000	0x3FF1_FFFF

15.3.1.2 权限配置

ESP32-S3 使用下方表 15.3-2 中的寄存器，控制 CPU 的指令总线 (IBUS) 和数据总线 (DBUS) 从安全世界和非安全世界中对 ROM 的取指 (X)、写入 (W) 和读取 (R) 权限：

表 15.3-2. ROM 的权限配置

总线	所处的世界	ROM 权限配置寄存器[A]	权限设置顺序
IBUS	安全世界	PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG [20:18] [B]	X/W/R
	非安全世界	PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG [20:18]	X/W/R
DBUS	安全世界	PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG [25:24] [C]	W/R
	非安全世界	PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG [27:26]	W/R

[A] 配置为 1 表示具有对应权限，配置为 0 表示不具有对应权限。

[B] 举例而言，配置该寄存器为 101 代表 CPU 的 IBUS 总线在安全世界中对 ROM 具有取指和读取权限，但没有写入权限。

[C] 举例而言，配置该寄存器为 01 代表 CPU 的 DBUS 总线在安全世界中对 ROM 具有读取权限，但没有写入权限。

15.3.2 SRAM 的权限管理

15.3.2.1 地址范围

ESP32-S3 的 SRAM 允许 CPU 通过指令总线 IBUS 或数据总线 DBUS 进行访问，安全世界和非安全世界的权限可单独配置，但一旦配置对 CPU0 和 CPU1 同时生效，具体允许访问的地址空间范围请见下方表 15.3-3。

表 15.3-3. SRAM Block 地址范围

SRAM	Block	IBUS 地址		DBUS 地址	
		起始地址	结束地址	起始地址	结束地址
Internal SRAM0	Block0	0x4037_0000	0x4037_3FFF	-	-
	Block1	0x4037_4000	0x4037_7FFF	-	-
Internal SRAM1	Block2	0x4037_8000	0x4037_FFFF	0x3FC8_8000	0x3FC8_FFFF
	Block3	0x4038_0000	0x4038_FFFF	0x3FC9_0000	0x3FC9_FFFF
	Block4	0x4039_0000	0x4039_FFFF	0x3FCA_0000	0x3FCA_FFFF
	Block5	0x403A_0000	0x403A_FFFF	0x3FCB_0000	0x3FCB_FFFF
	Block6	0x403B_0000	0x403B_FFFF	0x3FCC_0000	0x3FCC_FFFF
	Block7	0x403C_0000	0x403C_FFFF	0x3FCD_0000	0x3FCD_FFFF
	Block8	0x403D_0000	0x403D_FFFF	0x3FCE_0000	0x3FCE_FFFF
Internal SRAM 2	Block9	-	-	0x3FCF_0000	0x3FCF_7FFF
	Block10	-	-	0x3FCF_8000	0x3FCF_FFFF

本小节依次介绍了 Internal SRAM0、Internal SRAM1 和 Internal SRAM2 的权限配置。此外，还介绍了如何配置 Internal SRAM1 用作 Trace 内存。

15.3.2.2 Internal SRAM0 的权限配置

ESP32-S3 的 Internal SRAM0 分为 Block0 和 Block1（详见表 15.3-3），可分配给 CPU 或者 ICACHE 单独使用。一旦配置，对 CPU0 和 CPU1 同时生效。具体方式为配置如下表 15.3-4 寄存器的对应位：

表 15.3-4. Internal SRAM0 的使用权限配置

	Block	PMS_INTERNAL_SRAM_USAGE_1_REG[A]
SRAM	Block0	[0] [B]
	Block1	[1]

[A] 配置为 1 表示给 CPU 使用，配置为 0 表示给 ICACHE 使用。
[B] 举例而言，配置该位为 1，表示将 SRAM 的 Block0 给 CPU 使用。

当 CPU 获得使用权限时，ESP32-S3 使用下方表 15.3-5 中的寄存器，控制 CPU 的指令总线 (IBUS) 从安全世界和非安全世界中对 SRAM0 的取指 (X)、写入 (W) 和读取 (R) 权限：

表 15.3-5. Internal SRAM0 的访问权限配置

总线[A]	所处的世界	配置寄存器[B]	SRAM0		权限设置顺序
			Block0	Block1	
IBUS	安全世界	PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG	[14:12][C]	[17:15]	X/W/R
	非安全世界	PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG	[14:12]	[17:15]	X/W/R

[A] CPU 必须同时拥有使用权限和访问权限才能访问 Internal SRAM0。

[B] 配置为 1 表示具有对应权限，配置为 0 表示不具有对应权限。

[C] 举例而言，配置该寄存器为 0b101 代表 CPU 的 IBUS 总线在安全世界中对 SRAM 的 Block0 具有取指和读取权限，没有写入权限。

15.3.2.3 Internal SRAM1 权限配置

ESP32-S3 的 Internal SRAM1 分为 Block2 ~ Block8（详见表 15.3-3）：

- CPU 的数据总线 DBUS、指令总线 IBUS 和 GDMA 均可访问，且允许同时进行访问。
- 可用作 Trace 内存。
- 支持自定义地址段，可实现灵活的权限控制。

ESP32-S3 允许用户通过 5 条分割线将 SRAM1 分割为 6 个自定义地址段，每个区域可以分别配置权限。

具体来说，Internal SRAM1 可由 IRam0_DRam0_split_line 划分为指令空间和数据空间。

- 指令空间 (Instruction Region)：
  - 指令空间应配置为仅可由 IBUS 访问
  - 且可以由 IRam0_split_line_0 和 IRam0_split_line_1 进一步划分为 3 个分割区域
- 数据空间 (Data Region)：
  - 数据空间应配置为仅可由 DBUS 访问
  - 且可由 DRam0_split_line_0 和 DRam0_split_line_1 进一步划分为 3 个分割区域

请见下方图 15.3-1 和表 15.3-6 所示。

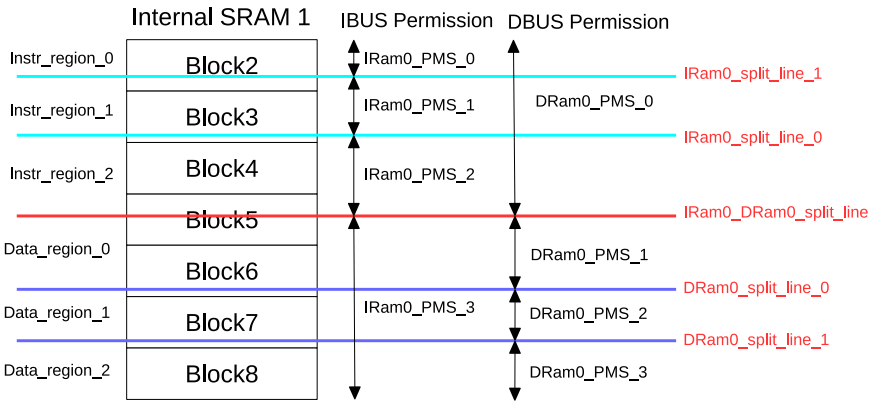


图 15.3-1. Internal SRAM1 的区域分割示意图

具体请见下表：

表 15.3-6. Internal SRAM1 的区域分割

内存[A]	空间	区域[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] 各细分区域的权限可分别配置，请见下方表 15.3-7 和 15.3-8。
[B] 有关分割线的具体配置，请见下方的描述。

划分区域

ESP32-S3 允许用户自行配置分割线的位置，每条分割线均有自己对应的配置寄存器。

- 指令和数据空间分割线 IRam0_DRam0_split_line 配置寄存器：
  - PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_1_REG
- 指令空间内部分割线 IRam0_split_line_0 配置寄存器：
  - PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_2_REG
- 指令空间内部分割线 IRam0_split_line_1 配置寄存器：
  - PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_3_REG
- 数据空间内部分割线 DRam0_split_line_0 配置寄存器：
  - PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_4_REG
- 数据空间内部分割线 DRam0_split_line_1 配置寄存器：
  - PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_5_REG

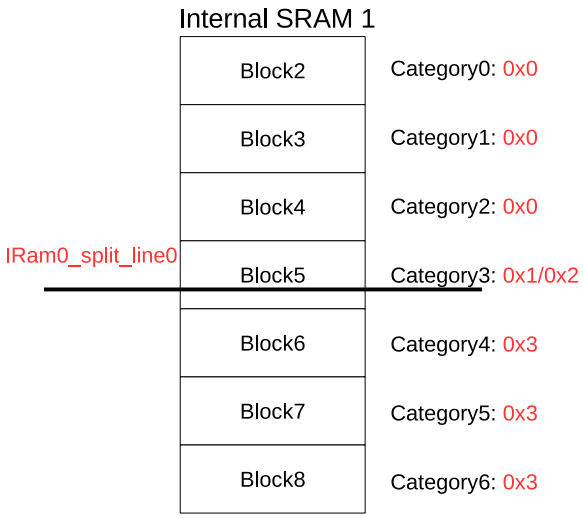


图 15.3-2. 分割线 Category 分配示意图

每条分割线在配置时应：

1. 首先配置分割线所处的 Block：
   - 将分割线所处 Block 对应的 Category_*x* 域配置为 0x1 或者 0x2（没有区别）；
   - 之前所有 Block 对应的 Category_0 ~ Category_*x-1* 域全被配置为 0x0；
   - 之后所有 Block 对应的 Category_*x+1* ~ Category_6 域全部配置为 0x3。

   举例而言，假设分割线位于 Block5，则应将 Block5 对应的 Category_3 域配置为 0x1 或者 0x2；将 Block2 ~ Block4 对应的 Category_0 ~ Category_2 域全部配置为 0x0；将 Block6 ~ Block8 对应的 Category_4 ~ Category_6 域全部配置为 0x3。可参考图 15.3-2。反过来，查询到 Category_3 域为 0x1 或者 0x2 也可以了解分割线处于 Block5。
2. 配置分割线在所处 Block 中的位置：
   - 将需要分割的 IBUS 或 DBUS 访问地址的 [15:8] 位写入分割线对应配置寄存器的 SPLITADDR 域中。
   - 注意分割地址必须配置为与 256 Byte 对齐，也就是说必须是 0x100 的整数倍。

   举例而言，假设需要分割的 IBUS 地址为 0x3fc88000，则应将该地址的 [15:8] 位，即 0b10000000 写入该分割线对应配置寄存器的 SPLITADDR 域。

值得注意的是，用户在配置 5 条分割线时还应遵循以下原则：

- 位置要求
  - 指令和数据空间分割线可以配置为 SRAM1 的任何位置
  - 指令空间中的 2 条内部分割线不能超出指令空间，相对位置无要求。
  - 数据空间中的 2 条内部分割线不能超出数据空间，相对位置无要求。
- 各分割线可以相互重合，相当于不存在。举例而言：
  - 数据空间中的 2 条内部分隔线如果不重合，则相当于将数据空间分为 3 个细分区域；
  - 数据空间中的 2 条内部分隔线如果重合，则相当于仅将数据空间分为 2 个细分区域；
  - 数据空间中的 2 条内部分隔线如果本身两两重合，且与指令和数据空间分割线重合，则相当于对数据空间内部不做细分，仅有一个数据空间区域.

配置权限

完成对分割线的配置后，用户可通过下表中的寄存器，独立配置 CPU 的 IBUS、DBUS 和 GDMA 外设从不同世界中，对细分指令空间和数据空间的访问权限。

对细分指令空间的配置如表 15.3-7 所示：

表 15.3-7. Internal SRAM1 的访问权限控制 – 指令空间

总线	所处的世界	配置寄存器	指令空间			权限设置顺序
			instr_region_0	instr_region_1	instr_region_2	
IBUS	安全世界	PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG	[2:0]	[5:3]	[8:6]	X/W/R
	非安全世界	PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG	[2:0]	[5:3]	[8:6]	X/W/R
DBUS	安全世界	PMS_Core_X_DRAM0_PMS_CONSTRAIN_1_REG	[1:0][A]			W/R
	非安全世界		[13:11][A]			W/R
GDMA	*XX* 外设[C]	PMS_DMA_APBPERI_*XX*_PMS_CONSTRAIN_1_REG	[1:0][B]			W/R

[A] 可配置 DBUS 数据总线在指令空间中的访问权限，但建议将 DBUS 对指令空间的权限配置为 0。

[B] 可配置 GDMA 在指令空间中的访问权限，但建议将 GDMA 对指令空间的权限配置为 0。

[C] ESP32-S3 共有 9 个外设（分别为 SPI2、SPI3、UCHI0、I2S0、I2S1、AES、SHA、ADC、LCD_CAM、USB、SDIO_HOST、RMT）可通过 GDMA 访问 Internal SRAM 1，所有的外设都有独立的权限寄存器控制其访问权限。

对细分数据空间的配置如表 15.3-8 所示：

表 15.3-8. Internal SRAM1 的访问权限控制 – 数据空间

总线	所处的世界	配置寄存器	数据空间			权限设置顺序
			data_region_0	data_region_1	data_region_2	
IBUS	安全世界	PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG	[11:9][A]			X/W/R
	非安全世界	PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG	[11:9][A]			X/W/R
DBUS	安全世界	PMS_Core_X_DRAM0_PMS_CONSTRAIN_1_REG	[3:2]	[5:4]	[7:6]	W/R
	非安全世界		[15:14]	[17:16]	[19:18]	W/R
GDMA	*XX* 外设[B]	PMS_DMA_APBPERI_*XX*_PMS_CONSTRAIN_1_REG	[3:2]	[5:4]	[7:6]	W/R

[A] 可配置 IBUS 指令总线在数据空间中的访问权限，但建议将 IBUS 对数据空间的权限配置为 0。

[B] ESP32-S3 共有 9 个外设（分别为 SPI2、SPI3、UCHI0、I2S0、I2S1、AES、SHA、ADC、LCD_CAM、USB、SDIO_HOST、RMT）可通过 GDMA 访问 Internal SRAM 1，所有的外设都有独立的权限寄存器控制其访问权限。

有关分割线的具体配置，请见第 15.3.2.3。

Trace 内存

ESP32-S3 搭载低功耗 Xtensa LX7 32 位双核处理器，内置 TRAX (Real-time Trace) 压缩模块，可便利开发人员的调试。该模块的实现需要 16 KB 存储空间，用户可从 Internal SRAM1 中分配 16 KB 用作 Trace 记录存储器，CPU0 和 CPU1 可分别配置。

用户可通过配置 PMS_INTERNAL_SRAM_USAGE_2_REG 寄存器从 Internal SRAM1 中选择 16 KB 作为 Trace Memory，详细步骤如下所示。

1. 首先，配置 PMS_INTERNAL_SRAM_CORE*m*_TRACE_USAGE 域中各 Block 对应的位为 1 选择 Block：
   - Block2: 0b00000001
   - Block3: 0b00000010
   - Block4: 0b00000100
   - Block5: 0b00001000

- Block6: 0b00010000
- Block7: 0b00100000
- Block8: 0b01000000

2. 其次，配置 PMS_INTERNAL_SRAM_CORE*m*_TRACE_ALLOC 域，从选中的 Block 中选择 16 KB
   - 2'b00：第一个 16 KB
   - 2'b01：第二个 16 KB
   - 2'b10：第三个 16 KB
   - 2'b11：第四个 16 KB

   注意，Block2 仅有 32 KB，因此如在第一步选择 Block2 则这里仅可配置为 2'b00 或 2'b0。

举例而言，假设你希望配置 Internal SRAM1 中 Block3 的第一个 16 KB 作为 CPU0 的 Trace memory, 则需要

- 配置 PMS_INTERNAL_SRAM_COREO_TRACE_USAGE 域为 0b0000010，选择 Block3
- 配置 PMS_INTERNAL_SRAM_COREO_TRACE_ALLOC 域为 2'b00，选择第一个 16 KB。

15.3.2.4 Internal SRAM2 权限配置

ESP32-S3 的 Internal SRAM2 分为 Block9 和 Block10（详见表 15.3-3），可分配给 CPU/GDMA 或者 DCACHE 单独使用。一旦配置，对 CPU0 和 CPU1 同时生效。具体方式为配置如下表 15.3-9 寄存器的对应位：

表 15.3-9. Internal SRAM2 的使用权限配置

	Block	PMS_INTERNAL_SRAM_USAGE_1_REG[A]
SRAM	Block9	[3][B]
	Block10	[2]

[A] 配置为 1 表示给 CPU 和 GDMA 使用，配置为 0 表示给 DCACHE 使用。

[B] 举例而言，配置该位为 1，表示将 SRAM 的 Block9 给 CPU 和 GDMA 使用。

当 CPU 和 GDMA 获得使用权限时，ESP32-S3 使用下方表 15.3-10 中的寄存器，控制 CPU 的数据总线 (DBUS) 在不同世界中对 SRAM2 的写入 (W) 和读取 (R) 权限：

表 15.3-10. Internal SRAM2 的访问权限配置

总线[A]	所处的世界	配置寄存器	SRAM2		权限设置顺序[B]
			Block9	Block10	
DBUS	安全世界	PMS_CORE_X_DRAMO_PMS_CONSTRAIN_1_REG	[9:8][C]	[11:10]	W/R
	非安全世界	PMS_CORE_X_DRAMO_PMS_CONSTRAIN_1_REG	[21:20]	[23:22]	W/R
GDMA	*XX* 外设[B]	PMS_DMA_APBPERI_*XX*_PMS_CONSTRAIN_1_REG	[9:8]	[11:10]	W/R

[A] CPU 和 GDMA 必须同时拥有使用权限和访问权限才能访问 Internal SRAM2。

[B] 配置为 1 表示具有对应权限，配置为 0 表示不具有对应权限。

[C] 举例而言，配置该寄存器为 0b10 代表 CPU 和 GDMA 的 DBUS 总线在安全世界中对 SRAM 的 Block9 具有写入权限，但没有读取权限。

15.3.3 RTC 快速内存 (FAST Memory) 的权限管理

15.3.3.1 地址范围

ESP32-S3 的 RTC FAST Memory 的存储空间大小为 8 KB，访问地址范围如下：

表 15.3-11. 地址范围

内存	起始地址	结束地址
RTC 快速内存	0x600F_E000	0x600F_FFFF

15.3.3.2 权限配置

为了实现更细化的权限管理，ESP32-S3 允许用户使用 1 条分割线将 RTC 快速存储空间分割为 2 个区域。用户可通过配置相应寄存器（PMS_CORE_*m*_PIF_PMS_CONSTRAN_*n*_REG），独立配置 CPU0 和 CPU1 在安全世界和非安全世界中的分割线及分割区域权限。

分割线的配置如下表所示：

表 15.3-12. RTC 快速内存的分割

RTC 快速内存	分割地址配置寄存器[1]	
	安全世界	非安全世界
高地址段	PIF_PMS_CONSTRAN_9_REG [10:0]	PIF_PMS_CONSTRAN_9_REG [21:11]
低地址段		

[1] 分割地址应采用相对于 RTC 快速内存基地址的偏移量。举例而言，用户如需设置 0x600F_F000 作为分割线地址，则应配置此寄存器为 0x1000 的字地址。

各地址段的权限配置如下表所示：

表 15.3-13. RTC 快速内存的权限配置

总线	RTC 快速内存	权限配置寄存器		配置顺序[A]
		安全世界	非安全世界	
外设总线 (PIF)	高地址段	PIF_PMS_CONSTRAN_10_REG [5:3] [B]	PIF_PMS_CONSTRAN_10_REG [11:9]	X/W/R
	低地址段	PIF_PMS_CONSTRAN_10_REG [2:0]	PIF_PMS_CONSTRAN_10_REG [8:6]	

[A] 配置为 1 表示具有对应权限，配置为 0 表示不具有对应权限。

[B] 举例而言，配置该寄存器为 101 代表 CPU 的 PIF 外设总线在安全世界中对 RTC 快速内存的高地址段有取指和读取权限，但没有写入权限。

15.3.4 RTC 慢速内存 (SLOW Memory) 的权限管理

15.3.4.1 地址范围

ESP32-S3 的 RTC 慢速内存总大小为 8 KB。这块内存允许用户使用两套地址访问，详情见下方表 15.3-14：

表 15.3-14. 地址范围

RTC SLOW Memory	起始地址	结束地址
RTCSlow_0	0x5000_0000	0x5000_1FFF
RTCSlow_1	0x6002_1000	0x6002_2FFF

15.3.4.2 权限配置

为了实现更细化的权限管理，ESP32-S3 允许用户使用 2 条分割线分别将上述 2 个存储空间分别分割为 2 个区域。用户可通过配置相应寄存器（PMS_CORE_*m*_PIF_PMS_CONSTRAN_*n*_REG），独立配置 CPU0 和 CPU1 在安全世界和非安全世界中的分割线及分割区域权限。

分割线的配置如下表所示：

表 15.3-15. RTC 慢速内存的分割

RTC 慢速内存	分割地址配置寄存器[1]	
	安全世界	非安全世界
RTCSlow_0 高地址段	PIF_PMS_CONSTRAN_11_REG [10:0]	PIF_PMS_CONSTRAN_9_REG [21:11]
RTCSlow_0 低地址段		
RTCSlow_1 高地址段	PIF_PMS_CONSTRAN_13_REG [10:0]	PIF_PMS_CONSTRAN_13_REG [21:11]
RTCSlow_1 低地址段		

[1] 分割地址应采用相对于 RTC 慢速内存基地址的偏移量。举例而言，用户如需设置 0x6002_2000 作为分割线地址，则应配置此寄存器为 0x1000 的地址。

表 15.3-16. RTC 慢速内存的权限管理

总线	RTC 慢速内存	权限配置		配置[A] 顺序
		安全世界	安全世界	
外设总线 (PIF)	RTCSlow_0 高地址段	PIF_PMS_CONSTRAN_12_REG [5:3][B]	PIF_PMS_CONSTRAN_12_REG [11:9]	X/W/R
	RTCSlow_0 低地址段	PIF_PMS_CONSTRAN_12_REG [2:0]	PIF_PMS_CONSTRAN_12_REG [8:6]	
	RTCSlow_1 高地址段	PIF_PMS_CONSTRAN_14_REG [5:3]	PIF_PMS_CONSTRAN_14_REG [11:9]	
	RTCSlow_1 低地址段	PIF_PMS_CONSTRAN_14_REG [2:0]	PIF_PMS_CONSTRAN_14_REG [8:6]	

[A] 权限设置顺序为 X/W/R。如访问权限与配置权限不一致时，则访问将被拒绝，具体表现为取指和读取操作仅能得到 0，写入操作失败。

[B] 举例而言，配置该寄存器为 0b100 代表 CPU 的 PIF 外设总线从安全世界中对 RTC Slow_0 的高地址段仅有取指权限，没有写入或读取权限。

15.4 外设权限管理

15.4.1 外设空间权限控制

ESP32-S3 的模块和外设的权限大多均可独立控制，仅有部分外设共用一套权限管理寄存器，详见下表 15.4-1。用户可通过配置相应寄存器（PMS_CORE_*m*_PIF_PMS_CONSTRAN_*n*_REG），独立配置 CPU0 和 CPU1 在安全世界和非安全世界中对相应模块或外设的读写权限（从高到低为 R/W）。当 CPU 对外设发起的访问类型与所

配置类型不一致时，该访问将被拒绝，具体表现为读操作仅能得到 0，写操作直接失败。关于 PMS_CORE_*m*_PIF_PMS_CONSTRAN_*n*_REG 的说明：

- *m* 可为 0 或 1，分别针对 CPU0 和 CPU1 的配置；
- *n* 可为 1 ~ 8，其中 1~4 用于配置安全世界的权限，5 ~ 8 用于配置非安全世界的权限。

举例而言，用户可配置 PMS_CORE_0_PIF_PMS_CONSTRAIN_1_REG [1:0] 数值为 0x2，代表 CPU0 在安全世界下对 UART0 只有读权限，但无写权限。此时，CPU0 在安全世界下如在执行代码过程中尝试修改 UART0 的内部寄存器数值，将不会成功。

表 15.4-1. 外设权限管理寄存器

外设	安全世界权限配置寄存器	非安全世界权限配置寄存器	位
GDMA	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[7:6]
eFuse Controller & PMU[2]	PIF_PMS_CONSTRAN_1_REG	PIF_PMS_CONSTRAN_5_REG	[15:14]
IO_MUX	PIF_PMS_CONSTRAN_1_REG	PIF_PMS_CONSTRAN_5_REG	[17:16]
GPIO	PIF_PMS_CONSTRAN_1_REG	PIF_PMS_CONSTRAN_5_REG	[7:6]
Interrupt Matrix	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[21:20]
System Timer	PIF_PMS_CONSTRAN_2_REG	PIF_PMS_CONSTRAN_6_REG	[31:30]
Timer Group 0	PIF_PMS_CONSTRAN_2_REG	PIF_PMS_CONSTRAN_6_REG	[27:26]
Timer Group 1	PIF_PMS_CONSTRAN_2_REG	PIF_PMS_CONSTRAN_6_REG	[29:28]
World Controller	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[31:30]
System Registers	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[17:16]
Sensitive Registers	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[19:18]
Accelerators[1]	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[5:4]
CACHE & XTS_AES[2]	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[25:25]
UART 0	PIF_PMS_CONSTRAN_1_REG	PIF_PMS_CONSTRAN_5_REG	[1:0]
UART 1	PIF_PMS_CONSTRAN_1_REG	PIF_PMS_CONSTRAN_5_REG	[31:30]
UART 2	PIF_PMS_CONSTRAN_3_REG	PIF_PMS_CONSTRAN_7_REG	[17:16]
SPI 0	PIF_PMS_CONSTRAN_1_REG	PIF_PMS_CONSTRAN_5_REG	[5:4]
SPI 1	PIF_PMS_CONSTRAN_1_REG	PIF_PMS_CONSTRAN_5_REG	[3:2]
SPI 2	PIF_PMS_CONSTRAN_3_REG	PIF_PMS_CONSTRAN_7_REG	[1:0]
SPI 3	PIF_PMS_CONSTRAN_3_REG	PIF_PMS_CONSTRAN_7_REG	[3:2]
I2C 0	PIF_PMS_CONSTRAN_2_REG	PIF_PMS_CONSTRAN_6_REG	[5:4]
I2C 1	PIF_PMS_CONSTRAN_3_REG	PIF_PMS_CONSTRAN_7_REG	[7:6]
I2S 0	PIF_PMS_CONSTRAN_1_REG	PIF_PMS_CONSTRAN_5_REG	[29:28]
I2S 1	PIF_PMS_CONSTRAN_3_REG	PIF_PMS_CONSTRAN_7_REG	[15:14]
Pulse Count Controller	PIF_PMS_CONSTRAN_2_REG	PIF_PMS_CONSTRAN_6_REG	[13:12]
USB Serial/JTAG Controller	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[1:0]
USB OTG Core	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[15:14]
USB OTG External	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[3:2]
Two-wire Automotive Interface	PIF_PMS_CONSTRAN_3_REG	PIF_PMS_CONSTRAN_7_REG	[11:10]
UHCI 0	PIF_PMS_CONSTRAN_2_REG	PIF_PMS_CONSTRAN_6_REG	[7:6]
SD/MMC Host Controller	PIF_PMS_CONSTRAN_3_REG	PIF_PMS_CONSTRAN_7_REG	[9:8]
LED PWM Controller	PIF_PMS_CONSTRAN_2_REG	PIF_PMS_CONSTRAN_6_REG	[17:16]
Motor Control PWM 0	PIF_PMS_CONSTRAN_2_REG	PIF_PMS_CONSTRAN_6_REG	[25:24]

Cont'd on next page

表 15.4-1 – 接上页

外设	安全世界权限配置寄存器	非安全世界权限配置寄存器	位[3]
Motor Control PWM 1	PIF_PMS_CONSTRAN_3_REG	PIF_PMS_CONSTRAN_7_REG	[13:12]
Remote Control Peripheral	PIF_PMS_CONSTRAN_2_REG	PIF_PMS_CONSTRAN_6_REG	[11:10]
Camera-LCD Controller	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[11:10]
APB Controller	PIF_PMS_CONSTRAN_3_REG	PIF_PMS_CONSTRAN_7_REG	[5:4]
ADC Controller	PIF_PMS_CONSTRAN_4_REG	PIF_PMS_CONSTRAN_8_REG	[9:8]

[1]：加速器包括 AES、SHA、RSA、数字签名、HMAC

[2]：多个外设共用

[3]：配置顺序为 R/W

15.4.2 自定义地址段权限管理

ESP32-S3 中除了支持以上述外设地址范围为单位的权限检查外，还允许用户进一步将具体某个外设地址范围，分割为 11 个地址段（即 Peri Region0 ~ Peri Region10），实现更细化的权限管理。

举例而言，GDMA 控制器寄存器包括：

- 针对每个接收通道，共 5 套；
- 针对每个发送通道，共 5 套；
- 共用配置寄存器，共 1 套。

可以看到，GDMA 控制器从硬件设计上已经将地址划分为 11 个地址段，因此可以实现对不同 GDMA 通道的权限管理。

用户可通过 PMS_CORE_*m*_Region_PMS_CONSTRAN_*n*_REG 寄存器，独立配置 CPU0 和 CPU1 从安全世界和非安全世界中对具体地址段的读写权限（从高到低为 R/W）。

关于 PMS_CORE_*m*_Region_PMS_CONSTRAN_*n*_REG 的说明：

- *m* 可为 0 或 1，分别针对对 CPU0 和 CPU1 的配置。
- *n* 可为 1 ~ 14，其中
  - Region_PMS_CONSTRAN_1_REG 用于配置从安全世界对相应地址段的权限。
  - Region_PMS_CONSTRAN_2_REG 用于配置从非安全世界对相应地址段的权限。
  - Region_PMS_CONSTRAN_*n*_REG（*n* = 3~14），用于配置每个地址段的起始地址。注意，每个地址段对起始地址，也是上一个地址段的结束地址。

表 15.4-2. 针对地址段的权限管理

地址段	起始地址配置	权限配置	
		安全世界	非安全世界
Peri Region0	Region_PMS_CONSTRAN_3_REG	Region_PMS_CONSTRAN_1_REG [1:0]	Region_PMS_CONSTRAN_2_REG [1:0]
Peri Region1	Region_PMS_CONSTRAN_4_REG	Region_PMS_CONSTRAN_1_REG [3:2]	Region_PMS_CONSTRAN_2_REG [3:2]
Peri Region2	Region_PMS_CONSTRAN_5_REG	Region_PMS_CONSTRAN_1_REG [5:4]	Region_PMS_CONSTRAN_2_REG [5:4]
Peri Region3	Region_PMS_CONSTRAN_6_REG	Region_PMS_CONSTRAN_1_REG [7:6]	Region_PMS_CONSTRAN_2_REG [7:6]
Peri Region4	Region_PMS_CONSTRAN_7_REG	Region_PMS_CONSTRAN_1_REG [9:8]	Region_PMS_CONSTRAN_2_REG [9:8]
Peri Region5	Region_PMS_CONSTRAN_8_REG	Region_PMS_CONSTRAN_1_REG [11:10]	Region_PMS_CONSTRAN_2_REG [11:10]
Peri Region6	Region_PMS_CONSTRAN_9_REG	Region_PMS_CONSTRAN_1_REG [13:12]	Region_PMS_CONSTRAN_2_REG [13:12]
Peri Region7	Region_PMS_CONSTRAN_10_REG	Region_PMS_CONSTRAN_1_REG [15:14]	Region_PMS_CONSTRAN_2_REG [15:14]
Peri Region8	Region_PMS_CONSTRAN_11_REG	Region_PMS_CONSTRAN_1_REG [17:16]	Region_PMS_CONSTRAN_2_REG [17:16]
Peri Region9	Region_PMS_CONSTRAN_12_REG	Region_PMS_CONSTRAN_1_REG [19:18]	Region_PMS_CONSTRAN_2_REG [19:18]
Peri Region10	Region_PMS_CONSTRAN_13_REG	Region_PMS_CONSTRAN_1_REG [21:20]	Region_PMS_CONSTRAN_2_REG [21:20]

15.5 片外存储器权限管理

在 ESP32-S3 中，存在三条访问外部存储器的数据路径，如图 15.5-1 所示：

- CPU 通过 SPI1 访问
- CPU 通过 CACHE 访问
- GDMA 访问

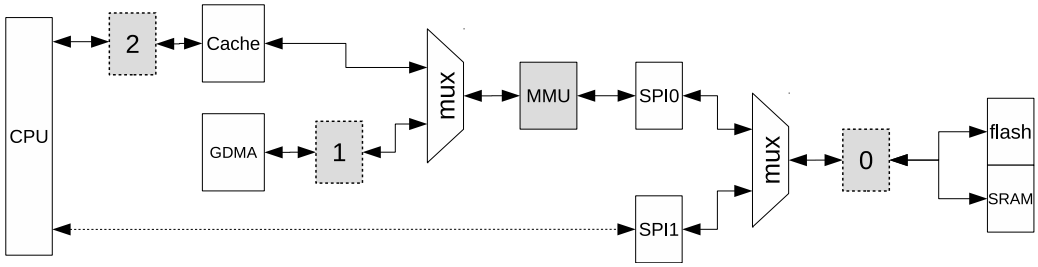


图 15.5-1. 外部存储器访问路径示意图

SPI1、GDMA 或 CACHE 如需访问外部存储器，必须首先获得相应的访问权限。在图 15.5-1 所示：

- 虚线框 0 对实地址做权限检查
- 虚线框 1 对 GDMA 访问的虚地址做权限检查。
- 虚线框 2 对 CPU 访问 CACHE 的请求做虚地址检查。

15.5.1 外部存储器实地址空间划分

ESP32-S3 可以将外部存储器 (flash + SRAM) 的实地址划分为共 8 个区域，从而实现更细化的权限控制：

- flash 划分为 4 个区域，每个区域的大小均为 64 KB 的倍数。
- SRAM 划分为 4 个区域，每个区域的大小均为 64 KB 的倍数。

- 此外，各区域的起始地址也均需以 64 KB 对齐。

用户可以通过下列寄存器配置这 8 个区域的地址范围：

表 15.5-1. 外部存储器的区域范围配置

地址段	区域范围的配置[3]	
	起始地址[1]	长度[2]
flash 区域*n* (*n*: 0~3)	SYSCON_FLASH_ACE_*n*_ADDR_REG	SYSCON_FLASH_ACE_*n*_SIZE_REG
SRAM 区域 *n* (*n*: 0~3)	SYSCON_SRAM_ACE_*n*_ADDR_REG	SYSCON_SRAM_ACE_*n*_SIZE_REG

[1] 配置区域 *n* 的起始地址，为实地址，需要与 64 KB 对齐。
[2] 配置区域 *n* 的长度。注意，flash 或 SRAM 的 4 个细分区域的总长度不得超过 1 GB。
[3] 各区域不允许相互重合。

15.5.2 外部存储器的权限配置

flash 的 4 个区域和外部 SRAM 的 4 个区域均可单独配置权限。用户可通过**访问属性列表**对每个区域的具体权限进行配置。

软件可以通过寄存器 SYSCON_SRAM_ACE*n*_ATTR_REG 和 SYSCON_FLASH_ACE*n*_ATTR 配置 SRAM 和 flash 对应区域的权限。

表 15.5-2. 外部存储器的权限配置

地址段	权限配置寄存器	权限配置		
		CACHE		SPI1
		安全世界权限 [A]	非安全世界权限 [A]	权限 [B]
flash 区域*n* (*n*: 0 ~ 3)	SYSCON_FLASH_ACE*n*_ATTR	[2:0] [C]	[5:3]	[7:6] [D]
SRAM 区域 *n* (*n*: 0 ~ 3)	SYSCON_SRAM_ACE*n*_ATTR_REG	[2:0]	[5:3]	[7:6]

[A] 配置顺序为 W/R/X。
[B] 配置顺序为 W/R。
[C] 举例而言，配置这些比特位为 0b010 代表 CACHE 在安全世界中对 flash 的区域*n* 有读取权限，但没有写入和取指权限。
[D] 举例而言，配置这些比特位为 0b01 代表 SPI 对 flash 的区域*n* 有读取权限，但没有写入权限。

15.5.3 GDMA 权限管理

ESP32-S3 还允许细化 GDMA 对外部 SRAM 的权限管理，即将 32 MB 的外部 SRAM 空间分割为四块区域，并针对其中的第二块和第三块区域，具体配置 GDMA 的访问权限。

表 15.5-3. 外部 SRAM 的区域范围配置（DMA 访问）

地址段	起始地址（含）	结束地址（不含）[2]
Region0 3	0x3C000000	PMS_EDMA_BOUNDARY_0_REG [1,2]
Region1 [4]	PMS_EDMA_BOUNDARY_0_REG [1]	PMS_EDMA_BOUNDARY_1_REG [1,2]
Region2 4	PMS_EDMA_BOUNDARY_1_REG [1]	PMS_EDMA_BOUNDARY_2_REG [1,2]
Region3 3	PMS_EDMA_BOUNDARY_2_REG [1]	0x3E000000 [2]

[1] 该地址在配置时以 4 KB 为单位，且为相对于 0x3C000000 的偏移量。举例而言，配置该寄存器为 0x80，则意味着地址为 0x3C000000 + 0x80 * 4 KB = 0x3C080000。

[2] 该地址值是结束地址加一。举例而言，0x3E000000 意味着结束地址为 0x3DFFFFFF。

[3] 此区域不允许支持 GDMA 功能的外设访问。

[4] 此区域允许支持 GDMA 功能的外设访问，包括 SPI2，SPI3，UHCI0，I2S0，I2S1，Camera-LCD Controller，AES，SHA，ADC 控制器和 Remote Control Peripheral，详见下方介绍。

表 15.5-4. 外部 SRAM 区域的权限配置

外设	权限配置		
	Region1	Region2	配置顺序
SPI2	PMS_EDMA_PMS_SPI2_ATTR1	PMS_EDMA_PMS_SPI2_ATTR2	W/R
SPI3	PMS_EDMA_PMS_SPI3_ATTR1	PMS_EDMA_PMS_SPI3_ATTR2	
UHCI0	PMS_EDMA_PMS_UHCI0_ATTR1	PMS_EDMA_PMS_UHCI0_ATTR2	
I2S0	PMS_EDMA_PMS_I2S0_ATTR1	PMS_EDMA_PMS_I2S0_ATTR2	
I2S1	PMS_EDMA_PMS_I2S1_ATTR1	PMS_EDMA_PMS_I2S1_ATTR2	
Camera-LCD Controller	PMS_EDMA_PMS_LCD_CAM_ATTR1	PMS_EDMA_PMS_LCD_CAM_ATTR2	
AES	PMS_EDMA_PMS_AES_ATTR1	PMS_EDMA_PMS_AES_ATTR2	
SHA	PMS_EDMA_PMS_SHA_ATTR1[A]	PMS_EDMA_PMS_SHA_ATTR2	
ADC Controller	PMS_EDMA_PMS_ADC_DAC_ATTR1	PMS_EDMA_PMS_ADC_DAC_ATTR2	
Remote Control Peripheral	PMS_EDMA_PMS_RMT_ATTR1	PMS_EDMA_PMS_RMT_ATTR2	

[A] 举例而言，配置该寄存器为 0b10 代表 SHA 对外部 SRAM Area0 有写入权限，没有读取权限。

15.6 非法访问与中断

当访问类型与配置允许访问的类型不一致时，ESP32-S3 将视其为**非法访问**，并进行如下处理：

- 过滤该访问请求，并返回默认值，具体表现为：
  - 取指和读取操作返回 0（非法访问片内存储器）或 0xdeadbeaf（非法访问片外存储器）的默认值
  - 写入操作无效
- 触发相应中断（如相应中断已使能），详见下文介绍。

本章节介绍的所有权限管理中断均可针对 CPU0 和 CPU1 独立配置，但均仅记录第一次发生中断的信息，因此建议用户检查到该中断信号后及时处理，并在处理后及时置位中断清除信号，从而保证下一次中断的顺利记录。

15.6.1 IBUS 总线非法访问中断

ESP32-S3 可经过配置，在 IBUS 总线非法访问 ROM 和 SRAM 时触发相应中断，并记录非法访问的信息。注意，此中断一经使能，对所有区域生效，不能单独使能针对某细分区域的非法访问中断。此中断对应章节 9 中断矩阵 *(INTERRUPT)* 表 9.3-1 中的 CORE_*m*_IRAM0_PMS_MONITOR_VIOLATE_INTR 中断源。

表 15.6-1. iBUS 总线的非法访问中断寄存器

寄存器	位	描述
PMS_CORE_*m*_IRAM0_PMS_MONITOR_1_REG	[0]	清除中断信号
	[1]	使能中断
PMS_CORE_*m*_IRAM0_PMS_MONITOR_2_REG	[0]	存储中断信号
	[1]	存储读写类型。1：写；0：读
	[2]	存储访问类型。1：加载/存储；0：取指
	[4:3]	存储中断发生时的世界信息。0b01：安全世界；0b10：非安全世界
	[28:5]	存储非法访问的地址

15.6.2 DBUS 总线非法访问中断

ESP32-S3 可经过配置，在 DBUS 总线非法访问 ROM 和 SRAM 时触发相应中断，并记录非法访问的信息。注意，此中断一经使能，对所有区域生效，不能单独使能针对某细分区域的非法访问中断。此中断对应章节 9 中断矩阵 *(INTERRUPT)* 表 9.3-1 中的 CORE_*m*_DRAM0_PMS_MONITOR_VIOLATE_INTR 中断源。

表 15.6-2. dBUS 总线的非法访问中断寄存器

寄存器	位	描述
PMS_CORE_*m*_DRAM0_PMS_MONITOR_1_REG	[0]	清除中断信号
	[1]	使能中断
PMS_CORE_*m*_DRAM0_PMS_MONITOR_2_REG	[0]	存储中断信号
	[1]	原子访问标志
	[3:2]	存储中断发生时的世界信息。0b01：安全世界；0b10：非安全世界
	[25:4]	存储非法访问的地址
PMS_CORE_*m*_DRAM0_PMS_MONITOR_3_REG	[0]	存储读写类型
	[25:4]	存储非法访问的字节信息

15.6.3 外部存储器中断

ESP32-S3 可在非法访问外部存储器时触发中断，并记录异常的类型，具体使用 SYSCON_SPI_MEM_PMS_CTRL_REG 寄存器。此中断对应章节 9 中断矩阵 *(INTERRUPT)* 表 9.3-1 中的 SPI_MEM_REJECT_INTR 中断源。

表 15.6-3. 外部存储器的非法访问中断寄存器

寄存器	位	描述
SYSCON_SPI_MEM_PMS_CTRL_REG	[0]	存储异常信号
	[1]	清除异常信号和记录
	[2]	非法取指
	[3]	非法读
	[4]	非法写
	[5]	访问落入多个有效区域（区域重叠）
	[6]	访问没有落入有效区域

15.6.4 GDMA 中断

ESP32-S3 可经过配置，在 GDMA 非法访问内部存储器时触发相应中断，并记录非法访问信息。此中断对应章节 9 中断矩阵 *(INTERRUPT)* 表 9.3-1 中的 DMA_APB_PMS_MONITOR_VIOLATE_INTR 中断源。

表 15.6-4. DMA 非法访问中断寄存器

寄存器	位	描述
PMS_DMA_APBPERI_PMS_MONITOR_1_REG	[0]	清除中断信号
	[1]	使能中断
PMS_DMA_APBPERI_PMS_MONITOR_2_REG	[0]	存储中断信号
	[2:1]	存储中断发生时的世界信息。0b01：安全世界；0b10：非安全世界
	[24:3]	存储非法访问的地址
PMS_DMA_APBPERI_PMS_MONITOR_3_REG	[0]	存储访问方向
	[16:1]	存储非法访问地址

GDMA 访问外部存储器的中断请参考章节 3。

15.6.5 PIF 外设总线中断

ESP32-S3 可经过配置，在 PIF 外设总线非法访问 RTC 快速内存、RTC 慢速内存和外设空间时触发中断，并记录非法访问信息。注意，此中断一经使能，对所有区域生效，不能单独设置仅对 RTC 快速内存、RTC 慢速内存或外设空间生效。此中断对应此中断对应章节 9 中断矩阵 *(INTERRUPT)* 表 9.3-1 中的 CORE_*m*_PIF_PMS_MONITOR_VIOLATE_INTR 中断源。

表 15.6-5. PIF 总线的非法访问中断寄存器

寄存器	位	描述
PMS_CORE_*m*_PIF_PMS_MONITOR_1_REG	[1]	中断使能信号，通过将该比特置位使中断处于使能状态
	[0]	中断清除信号，通过将该比特置位清除当前的中断信号及中断记录信息
PMS_CORE_*m*_PIF_PMS_MONITOR_2_REG	[7:6]	记录中断发生时所处的世界信息。0b01：安全世界；0b10：非安全世界
	[5]	记录中断发生时的访问方向。0 表示读；1 表示写
	[4:2]	记录中断发生时的访问数据类型。0：字节访问；1：半字访问；2：字访问
	[1]	记录中断发生时的访问类型。0：指令访问；1：数据访问
	[0]	记录中断源信息。0：没有中断发生；1：有中断发生
PMS_CORE_*m*_PIF_PMS_MONITOR_3_REG	[31:0]	记录中断发生时的访问地址信息

特别地，PIF 外设总线还可以针对错误的访问对齐方式进行检查，并触发中断，请见下节介绍。

15.6.6 非字对齐访问检查

ESP32-S3 的所有模块/外设（除了 RTC 慢速内存和快速内存）均仅支持**字对齐**访问，详见表 4.3-3。对此，ESP32-S3 可以监控对模块/外设的**非字对齐访问**，并在发现此类访问时（详见表 15.6-6）触发中断信号。此中断对应章节 9 中断矩阵 *(INTERRUPT)* 表 9.3-1 中的 CORE_*m*_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR 中断源。

由于处理器会将部分软件代码中的非对齐访问处理成对齐的访问发出，此时硬件监控系统便不会产生中断信号。下表列出了软件代码中所有可能发出的访问以及产生的结果（中断使能情况下），其中 **INTR** 表示会产生中断，√ 表示访问可以正常通过。

表 15.6-6. 非字对齐访问外设情况汇总

访问地址	访问数据类型	读访问	写访问
0x0	字节	INTR	INTR
	半字	INTR	INTR
	字	√	√
0x1	字节	INTR	INTR
	半字	√	INTR
	字	√	INTR
0x2	字节	INTR	INTR
	半字	INTR	INTR
	字	√	INTR
0x3	字节	INTR	INTR
	半字	√	INTR
	字	√	INTR

具体中断配置信息如下：

表 15.6-7. 非字对齐中断寄存器

寄存器	位	描述
PMS_CORE_*m*_PIF_PMS_MONITOR_4_REG	[1]	中断使能信号，通过将该比特置位使中断处于使能状态
	[0]	中断清除信号，通过将该比特置位清除当前的中断信号及中断记录信息
PMS_CORE_*m*_PIF_PMS_MONITOR_5_REG	[4:3]	记录中断发生时所处的世界信息。0b01：安全世界；0b10：非安全世界
	[2:1]	记录中断发生时的访问数据类型。0：字节访问；1：半字访问；2：字访问
	[0]	记录中断源信息。0：没有中断发生；1：有中断发生
PMS_CORE_*m*_PIF_PMS_MONITOR_6_REG	[31:0]	记录中断发生时的访问地址信息

15.7 CPU VECBASE 寄存器保护

CPU 内部特殊寄存器 VECBASE 决定中断和异常入口的基地址。为防止该寄存器被恶意篡改，ESP32-S3 允许将 VECBASE 寄存器中的值配置至 PMS_CORE_*m*_VECBASE_OVERRIDE_*n*_REG 寄存器中的 PMS_CORE_*m*_VECBASE_OVERRIDE_WORLD*n*_VALUE 域，并此后仅使用经过配置的 PMS_CORE_*m*_VECBASE_OVERRIDE_WORLD*n*_VALUE 值，而非 VECBASE 寄存器中的值。

接着，通过配置 PMS_CORE_*m*_VECBASE_OVERRIDE_*n*_REG 寄存器仅可配置为仅由安全世界访问，从而保护 VECBASE 寄存器的安全。

配置流程为：

- 配置 PMS_CORE_*m*_VECBASE_OVERRIDE_WORLD0_VALUE 为 CORE_*m* 在安全世界中 VECBASE 的值
- 配置 PMS_CORE_*m*_VECBASE_OVERRIDE_WORLD1_VALUE 为 CORE_*m* 在非安全世界中 VECBASE 的值
- 配置寄存器 PMS_CORE_*m*_VECBASE_OVERRIDE_1_REG 中的 PMS_CORE_*m*_VECBASE_OVERRIDE_SEL 域进行选择：
  - 2'b00: 不覆盖，使用 CORE_*m* 本身的 VECBASE 寄存器
  - 2'b11: 覆盖，使用 PMS_CORE_*m*_VECBASE_OVERRIDE_WORLD*n*_VALUE 覆盖 CPU 内部 VECBASE 寄存器
  - 此寄存器不得配置为其他值
- 配置寄存器 PMS_CORE_*m*_VECBASE_OVERRIDE_0_REG 中的 PMS_CORE_*m*_VECBASE_WORLD_MASK 域，决定 CPU 处于不同世界时，使用 World0_Value 还是 World1_Value。
  - 1: CORE*m* 处于非安全世界和安全世界都使用 WORLD0_VALUE
  - 0:
    * CPU*m* 处于安全世界时，使用 WORLD0_VALUE
    * CPU*m* 处于非安全世界时，使用 WORLD1_VALUE

15.8 寄存器锁

ESP32-S3 所有的权限寄存器都支持锁保护，即每一个权限寄存器都有对应的 lock 寄存器，当 lock 寄存器配置为 1 时，则相应的权限寄存器和 lock 寄存器本身都会被锁定，无法再次更改。当且仅当 CPU 复位时，lock 寄存器才会重新恢复成 0，允许更改权限。

权限寄存器和 lock 寄存器并非一一对应关系，具体见表 15.8-1。

表 15.8-1. 寄存器锁保护配置

寄存器锁寄存器	适用范围
VECBASE 配置寄存器锁	
PMS_CORE_*m*_VECBASE_OVERRIDE_LOCK_REG	PMS_CORE_*m*_VECBASE_OVERRIDE_LOCK_REG
	PMS_CORE_*m*_VECBASE_OVERRIDE_0_REG
	PMS_CORE_*m*_VECBASE_OVERRIDE_1_REG
	PMS_CORE_*m*_VECBASE_OVERRIDE_2_REG
内部 SRAM 使用与权限配置寄存器锁	
PMS_INTERNAL_SRAM_USAGE_0_REG	PMS_INTERNAL_SRAM_USAGE_0_REG
	PMS_INTERNAL_SRAM_USAGE_1_REG
	PMS_INTERNAL_SRAM_USAGE_2_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
	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
内部 SRAM 分割线配置寄存器锁	
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)
外设权限配置寄存器锁	
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_CORE_*m*_REGION_PMS_CONSTRAIN_0_REG	PMS_CORE_*m*_REGION_PMS_CONSTRAIN_0_REG
	PMS_CORE_*m*_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)
GDMA 外设访问 Internal SRAM 权限配置寄存器	
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_SPI3_PMS_CONSTRAIN_0_REG	PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_0_REG

寄存器锁寄存器	适用范围
	PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_1_REG
PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_0_REG	PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_0_REG
	PMS_DMA_APBPERI_UHCI0_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_I2S0_PMS_CONSTRAIN_1_REG	PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_0_REG
	PMS_DMA_APBPERI_I2S1_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_RMT_PMS_CONSTRAIN_0_REG	PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_0_REG
	PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_1_REG
PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_0_REG	PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_0_REG
	PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_1_REG
PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_0_REG	PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_0_REG
	PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_1_REG
PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_0_REG	PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_0_REG
	PMS_DMA_APBPERI_SDIO_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
外设访问外部 SRAM 寄存器锁	
PMS_EDMA_BOUNDARY_LOCK_REG	PMS_EDMA_BOUNDARY_LOCK_REG
	PMS_EDMA_BOUNDARY_0_REG
	PMS_EDMA_BOUNDARY_1_REG
	PMS_EDMA_BOUNDARY_2_REG
PMS_EDMA_PMS_SPI2_LOCK_REG	PMS_EDMA_PMS_SPI2_LOCK_REG
	PMS_EDMA_PMS_SPI2_REG
PMS_EDMA_PMS_SPI3_LOCK_REG	PMS_EDMA_PMS_SPI3_LOCK_REG
	PMS_EDMA_PMS_SPI3_REG
PMS_EDMA_PMS_UHCI0_LOCK_REG	PMS_EDMA_PMS_UHCI0_LOCK_REG
	PMS_EDMA_PMS_UHCI0_REG
PMS_EDMA_PMS_I2S0_LOCK_REG	PMS_EDMA_PMS_I2S0_LOCK_REG
	PMS_EDMA_PMS_I2S0_REG
PMS_EDMA_PMS_I2S1_LOCK_REG	PMS_EDMA_PMS_I2S1_LOCK_REG
	PMS_EDMA_PMS_I2S1_REG

寄存器锁寄存器	适用范围
PMS_EDMA_PMS_LCD_CAM_LOCK_REG	PMS_EDMA_PMS_LCD_CAM_LOCK_REG
	PMS_EDMA_PMS_LCD_CAM_REG
PMS_EDMA_PMS_AES_LOCK_REG	PMS_EDMA_PMS_AES_LOCK_REG
	PMS_EDMA_PMS_AES_REG
PMS_EDMA_PMS_AES_LOCK_REG	PMS_EDMA_PMS_AES_LOCK_REG
	PMS_EDMA_PMS_AES_REG
PMS_EDMA_PMS_SHA_LOCK_REG	PMS_EDMA_PMS_SHA_LOCK_REG
	PMS_EDMA_PMS_SHA_REG
PMS_EDMA_PMS_ADC_DAC_LOCK_REG	PMS_EDMA_PMS_ADC_DAC_LOCK_REG
	PMS_EDMA_PMS_ADC_DAC_REG
PMS_EDMA_PMS_RMT_LOCK_REG	PMS_EDMA_PMS_RMT_LOCK_REG
	PMS_EDMA_PMS_RMT_REG

15.9 寄存器列表

本小节的所有以 PMS 开头的寄存器地址均为相对于权限管理基地址的地址偏移量（相对地址），所有以 APB 开头的寄存器地址均为相对于 APB 控制寄存器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

注意，本章节中所有含有 CORE_X 的寄存器均适用于 CPU0 和 CPU1。其余寄存器 CPU0 与 CPU1 完全相同。下方表格仅列出 CPU0 的相关寄存器，CPU1 的寄存器仅需在 CPU0 相关寄存器地址偏移量的基础上再加 0x0400 即可。

比如 CPU0 寄存器 PMS_CORE_0_IRAM0_PMS_MONITOR_0_REG 的地址偏移量为 0x00E4，则对应的 CPU1 寄存器 PMS_CORE_1_IRAM0_PMS_MONITOR_0_REG 的地址为 0x00E4 + 0x0400，即 0x04E4。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问权限
配置寄存器			
PMS_APB_PERIPHERAL_ACCESS_0_REG	APB 外设配置寄存器 0	0x0008	R/W
PMS_APB_PERIPHERAL_ACCESS_1_REG	APB 外设配置寄存器 1	0x000C	R/W
PMS_INTERNAL_SRAM_USAGE_0_REG	内部 SRAM 配置寄存器 0	0x0010	R/W
PMS_INTERNAL_SRAM_USAGE_1_REG	内部 SRAM 配置寄存器 1	0x0014	R/W
PMS_INTERNAL_SRAM_USAGE_2_REG	内部 SRAM 配置寄存器 2	0x0018	R/W
PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_0_REG	SPI2 GDMA 权限配置寄存器 0	0x0038	R/W
PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_1_REG	SPI2 GDMA 权限配置寄存器 1	0x003C	R/W
PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_0_REG	SPI3 GDMA 权限配置寄存器 0	0x0040	R/W
PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_1_REG	SPI3 GDMA 权限配置寄存器 1	0x0044	R/W
PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_0_REG	UHCI0 GDMA 权限配置寄存器 0	0x0048	R/W
PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_1_REG	UHCI0 GDMA 权限配置寄存器 1	0x004C	R/W
PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_0_REG	I2S0 GDMA 权限配置寄存器 0	0x0050	R/W
PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_1_REG	I2S0 GDMA 权限配置寄存器 1	0x0054	R/W
PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_0_REG	I2S1 GDMA 权限配置寄存器 0	0x0058	R/W
PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_1_REG	I2S1 GDMA 权限配置寄存器 1	0x005C	R/W
PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_0_REG	AES GDMA 权限配置寄存器 0	0x0070	R/W
PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_1_REG	AES GDMA 权限配置寄存器 1	0x0074	R/W

GoBack

名称	描述	地址	访问权限
PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_0_REG	SHA GDMA 权限配置寄存器 0	0x0078	R/W
PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_1_REG	SHA GDMA 权限配置寄存器 1	0x007C	R/W
PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_0_REG	ADC_DAC GDMA 权限配置寄存器 0	0x0080	R/W
PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_1_REG	ADC_DAC GDMA 权限配置寄存器 1	0x0084	R/W
PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_0_REG	RMT GDMA 权限配置寄存器 0	0x0088	R/W
PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_1_REG	RMT GDMA 权限配置寄存器 1	0x008C	R/W
PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_0_REG	LCD_CAM GDMA 权限配置寄存器 0	0x0090	R/W
PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_1_REG	LCD_CAM GDMA 权限配置寄存器 1	0x0094	R/W
PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_0_REG	USB GDMA 权限配置寄存器 0	0x0098	R/W
PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_1_REG	USB GDMA 权限配置寄存器 0	0x009C	R/W
PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_0_REG	SDIO GDMA 权限配置寄存器 0	0x00A8	R/W
PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_1_REG	SDIO GDMA 权限配置寄存器 1	0x00AC	R/W
PMS_DMA_APBPERI_PMS_MONITOR_0_REG	GDMA 权限中断寄存器 0	0x00B0	R/W
PMS_DMA_APBPERI_PMS_MONITOR_1_REG	GDMA 权限中断寄存器 1	0x00B4	R/W
PMS_DMA_APBPERI_PMS_MONITOR_2_REG	GDMA 权限中断寄存器 2	0x00B8	RO
PMS_DMA_APBPERI_PMS_MONITOR_3_REG	GDMA 权限中断寄存器 3	0x00BC	RO
PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_0_REG	SRAM 分割线配置寄存器 0	0x00C0	R/W
PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_1_REG	SRAM 分割线配置寄存器 1	0x00C4	R/W
PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_2_REG	SRAM 分割线配置寄存器 2	0x00C8	R/W
PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_3_REG	SRAM 分割线配置寄存器 3	0x00CC	R/W
PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_4_REG	SRAM 分割线配置寄存器 4	0x00D0	R/W
PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_5_REG	SRAM 分割线配置寄存器 5	0x00D4	R/W
PMS_CORE_X_IRAM0_PMS_CONSTRAIN_0_REG	IBUS 权限配置寄存器 0	0x00D8	R/W

GoBack

名称	描述	地址	访问权限
PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG	IBUS 权限配置寄存器 1	0x00DC	R/W
PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG	IBUS 权限配置寄存器 2	0x00E0	R/W
PMS_CORE_0_IRAM0_PMS_MONITOR_0_REG	CPU0 IBUS 权限中断寄存器 0	0x00E4	R/W
PMS_CORE_0_IRAM0_PMS_MONITOR_1_REG	CPU0 IBUS 权限中断寄存器 1	0x00E8	R/W
PMS_CORE_X_DRAM0_PMS_CONSTRAIN_0_REG	DBUS 权限配置寄存器 0	0x00FC	R/W
PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG	DBUS 权限配置寄存器 1	0x0100	R/W
PMS_CORE_0_DRAM0_PMS_MONITOR_0_REG	CPU0 dBUS 权限中断寄存器 0	0x0104	R/W
PMS_CORE_0_DRAM0_PMS_MONITOR_1_REG	CPU0 dBUS 权限中断寄存器 1	0x0108	R/W
PMS_CORE_0_PIF_PMS_CONSTRAIN_*n*_REG (*n*: 0 -14)	外设权限配置寄存器 s	0x0124 + 4**n*	R/W
PMS_CORE_0_REGION_PMS_CONSTRAIN_0_REG	CPU0 Split_Region 权限寄存器 0	0x0160	R/W
PMS_CORE_0_REGION_PMS_CONSTRAIN_1_REG	CPU0 Split_Region 权限寄存器 1	0x0164	R/W
PMS_CORE_0_REGION_PMS_CONSTRAIN_2_REG	CPU0 Split_Region 权限寄存器 2	0x0168	R/W
PMS_CORE_0_REGION_PMS_CONSTRAIN_3_REG	CPU0 Split_Region 权限寄存器 3	0x016C	R/W
PMS_CORE_0_PIF_PMS_MONITOR_0_REG	CPU0 PIF 权限中断寄存器 0	0x019C	R/W
PMS_CORE_0_PIF_PMS_MONITOR_1_REG	CPU0 PIF 权限中断寄存器 1	0x01A0	R/W
PMS_CORE_0_PIF_PMS_MONITOR_4_REG	CPU0 PIF 权限中断寄存器 4	0x01AC	R/W
PMS_CORE_0_VECBASE_OVERRIDE_LOCK_REG	CPU0 vecbase 选项配置寄存器 0	0x01B8	R/W
PMS_CORE_0_VECBASE_OVERRIDE_0_REG	CPU0 vecbase 选项配置寄存器 0	0x01BC	R/W
PMS_CORE_0_VECBASE_OVERRIDE_1_REG	CPU0 vecbase 选项配置寄存器 1	0x01C0	R/W
PMS_CORE_0_VECBASE_OVERRIDE_2_REG	CPU0 vecbase 选项配置寄存器 1	0x01C4	R/W
PMS_EDMA_BOUNDARY_LOCK_REG	EDMA 边界寄存器锁	0x02A8	R/W
PMS_EDMA_BOUNDARY_0_REG	EDMA 边界 0 配置寄存器	0x02AC	R/W
PMS_EDMA_BOUNDARY_1_REG	EDMA 边界 1 配置寄存器	0x02B0	R/W
PMS_EDMA_BOUNDARY_2_REG	EDMA 边界 2 配置寄存器 0	0x02B4	R/W
PMS_EDMA_PMS_SPI2_LOCK_REG	SPI2 外部存储器权限寄存器锁	0x02B8	R/W
PMS_EDMA_PMS_SPI2_REG	SPI2 外部存储器权限配置寄存器	0x02BC	R/W
PMS_EDMA_PMS_SPI3_LOCK_REG	SPI3 外部存储器权限寄存器锁	0x02C0	R/W
PMS_EDMA_PMS_SPI3_REG	SPI3 外部存储器权限配置寄存器	0x02C4	R/W
PMS_EDMA_PMS_UHCI0_LOCK_REG	UHCI0 外部存储器权限寄存器锁	0x02C8	R/W

GoBack

名称	描述	地址	访问权限
PMS_EDMA_PMS_UHCI0_REG	UHCI0 外部存储器权限配置寄存器	0x02CC	R/W
PMS_EDMA_PMS_I2S0_LOCK_REG	I2S0 外部存储器权限寄存器锁	0x02D0	R/W
PMS_EDMA_PMS_I2S0_REG	I2S0 外部存储器权限配置寄存器	0x02D4	R/W
PMS_EDMA_PMS_I2S1_LOCK_REG	I2S1 外部存储器权限寄存器锁	0x02D8	R/W
PMS_EDMA_PMS_I2S1_REG	I2S1 外部存储器权限配置寄存器	0x02DC	R/W
PMS_EDMA_PMS_LCD_CAM_LOCK_REG	LCD/CAM 外部存储器权限寄存器锁	0x02E0	R/W
PMS_EDMA_PMS_LCD_CAM_REG	LCD/CAM 外部存储器权限配置寄存器	0x02E4	R/W
PMS_EDMA_PMS_AES_LOCK_REG	AES 外部存储器权限寄存器锁	0x02E8	R/W
PMS_EDMA_PMS_AES_REG	AES 外部存储器权限配置寄存器	0x02EC	R/W
PMS_EDMA_PMS_SHA_LOCK_REG	SHA 外部存储器权限寄存器锁	0x02F0	R/W
PMS_EDMA_PMS_SHA_REG	SHA 外部存储器权限配置寄存器	0x02F4	R/W
PMS_EDMA_PMS_ADC_DAC_LOCK_REG	ADC/DAC 外部存储器权限寄存器锁	0x02F8	R/W
PMS_EDMA_PMS_ADC_DAC_REG	ADC/DAC 外部存储器权限配置寄存器	0x02FC	R/W
PMS_EDMA_PMS_RMT_LOCK_REG	RMT 外部存储器权限寄存器锁	0x0300	R/W
PMS_EDMA_PMS_RMT_REG	RMT 权限配置寄存器	0x0304	R/W
PMS_CLOCK_GATE_REG_REG	门控配置寄存器	0x0308	R/W
状态寄存器			
PMS_CORE_0_IRAM0_PMS_MONITOR_2_REG	CPU0 IBUS 权限中断寄存器 2	0x00EC	RO
PMS_CORE_0_DRAM0_PMS_MONITOR_2_REG	CPU0 dBUS 权限中断寄存器 2	0x010C	RO
PMS_CORE_0_DRAM0_PMS_MONITOR_3_REG	CPU0 dBUS 权限中断寄存器 3	0x0110	RO
PMS_CORE_0_PIF_PMS_MONITOR_2_REG	CPU0 PIF 权限中断寄存器 2	0x01A4	RO
PMS_CORE_0_PIF_PMS_MONITOR_3_REG	CPU0 PIF 权限中断寄存器 3	0x01A8	RO
PMS_CORE_0_PIF_PMS_MONITOR_5_REG	CPU0 PIF 权限中断寄存器 5	0x01B0	RO
PMS_CORE_0_PIF_PMS_MONITOR_6_REG	CPU0 PIF 权限中断寄存器 6	0x01B4	RO
版本寄存器			
PMS_DATE_REG	日期与版本寄存器	0x0FFC	R/W

GoBack

名称	描述	地址	权限
配置寄存器			
SYSCON_EXT_MEM_PMS_LOCK_REG	外部存储器权限锁定寄存器	0x0020	R/W
SYSCON_FLASH_ACE*n*_ATTR_REG (*n*: 0 - 3)	flash 区域*n* 权限配置寄存器	0x0028 + 4**n*	R/W
SYSCON_SRAM_ACE*n*_ADDR_S (*n*: 0 - 3)	flash 区域*n* 起始地址配置寄存器	0x0038 + 4**n*	R/W
SYSCON_FLASH_ACE*n*_SIZE_REG (*n*: 0 - 3)	flash 区域*n* 长度配置寄存器	0x0048 + 4**n*	R/W
SYSCON_SRAM_ACE*n*_ATTR_REG (*n*: 0 - 3)	External SRAM Area*n* 权限配置寄存器	0x0058 + 4**n*	R/W
SYSCON_SRAM_ACE*n*_ADDR_REG (*n*: 0 - 3)	External SRAM Area*n* 起始地址配置寄存器	0x0068 + 4**n*	R/W
SYSCON_SRAM_ACE*n*_SIZE_REG (*n*: 0 - 3)	External SRAM Area*n* 长度配置寄存器	0x0078 + 4**n*	R/W
SYSCON_SPI_MEM_PMS_CTRL_REG	外部存储器访问异常控制寄存器	0x0088	varies
SYSCON_SPI_MEM_REJECT_ADDR_REG	外部存储器访问异常实地址记录寄存器	0x008C	RO

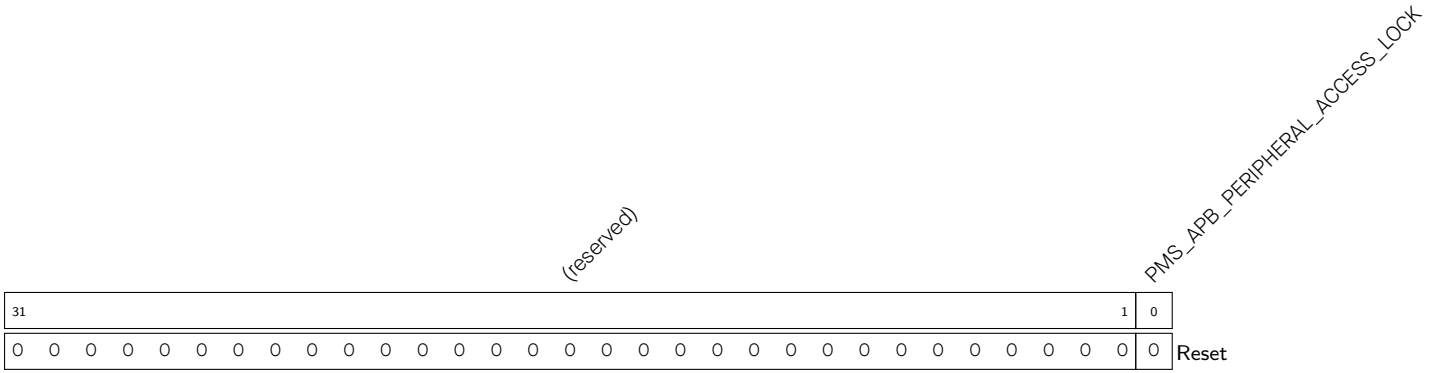
15.10 寄存器

本小节的所有以 PMS 开头的寄存器地址均为相对于权限管理基地址的地址偏移量（相对地址），所有以 APB 开头的寄存器地址均为相对于 APB 控制寄存器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

注意，本章节中所有含有 CORE_X 的寄存器均适用于 CPU0 和 CPU1。其余寄存器 CPU0 与 CPU1 完全相同。下方表格仅列出 CPU0 的相关寄存器，CPU1 的寄存器仅需在 CPU0 相关寄存器地址偏移量的基础上再加 0x0400 即可。

比如 CPU0 寄存器 PMS_CORE_0_IRAM0_PMS_MONITOR_0_REG 的地址偏移量为 0x00E4，则对应的 CPU1 寄存器 PMS_CORE_1_IRAM0_PMS_MONITOR_0_REG 的地址为 0x00E4 + 0x0400，即 0x04E4。

Register 15.1. PMS_APB_PERIPHERAL_ACCESS_0_REG (0x0008)



PMS_APB_PERIPHERAL_ACCESS_LOCK 置 1 锁住 APB 外设配置寄存器。(R/W)

GoBack

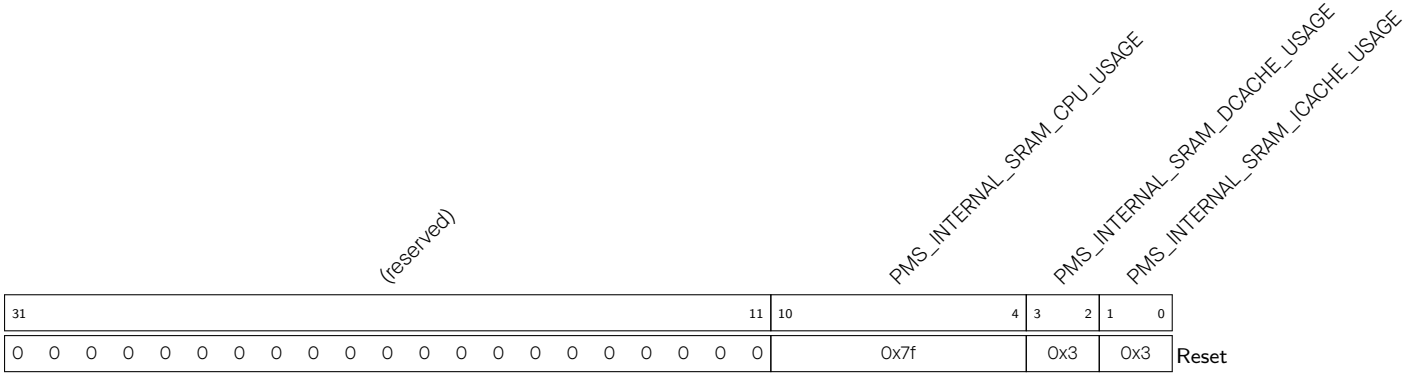
Register 15.2. PMS_APB_PERIPHERAL_ACCESS_1_REG (0x000C)

PMS_APB_PERIPHERAL_ACCESS_SPLIT_BURST 置 1 表示支持 PIF 总线对外设地址空间发起连续背靠背访问。(R/W)

Register 15.3. PMS_INTERNAL_SRAM_USAGE_0_REG (0x0010)

PMS_INTERNAL_SRAM_USAGE_LOCK 置 1 锁住内部 SRAM 配置寄存器。(R/W)

Register 15.4. PMS_INTERNAL_SRAM_USAGE_1_REG (0x0014)

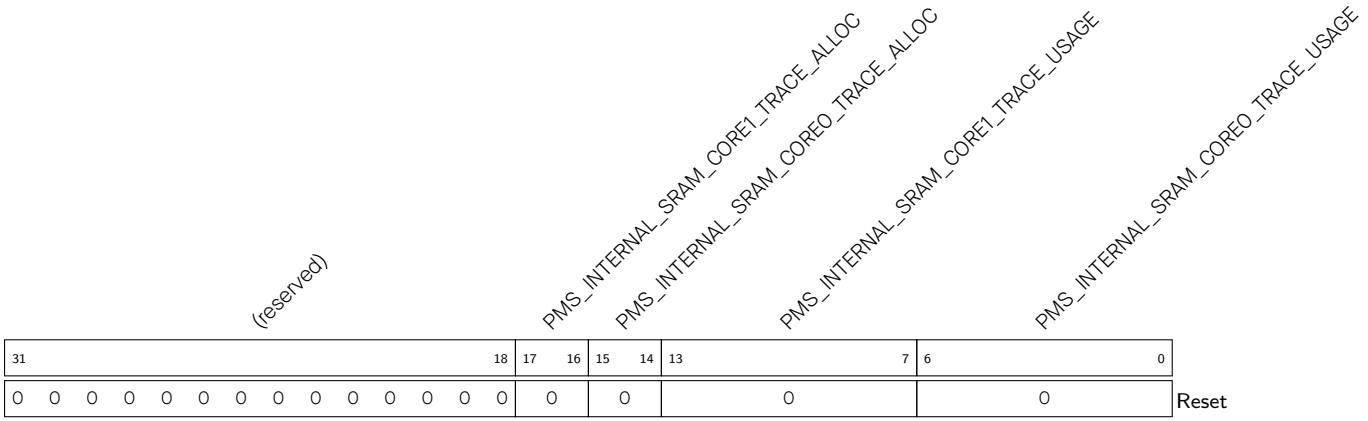


PMS_INTERNAL_SRAM_ICACHE_USAGE 配置 SRAM0 的特定 block 给 CPU 或 ICACHE 使用。(R/W)

PMS_INTERNAL_SRAM_DCACHE_USAGE 配置 SRAM2 的特定 block 给 CPU 或 DCACHE 使用。(R/W)

PMS_INTERNAL_SRAM_CPU_USAGE 配置允许 CPU 使用 SRAM1 的特定 block。(R/W)

Register 15.5. PMS_INTERNAL_SRAM_USAGE_2_REG (0x0018)



PMS_INTERNAL_SRAM_COREO_TRACE_USAGE 选择 SRAM1 的特定 block 作为 CPU0 的 trace memory block。(R/W)

PMS_INTERNAL_SRAM_CORE1_TRACE_USAGE 选择 SRAM1 的特定 block 作为 CPU1 的 trace memory block。(R/W)

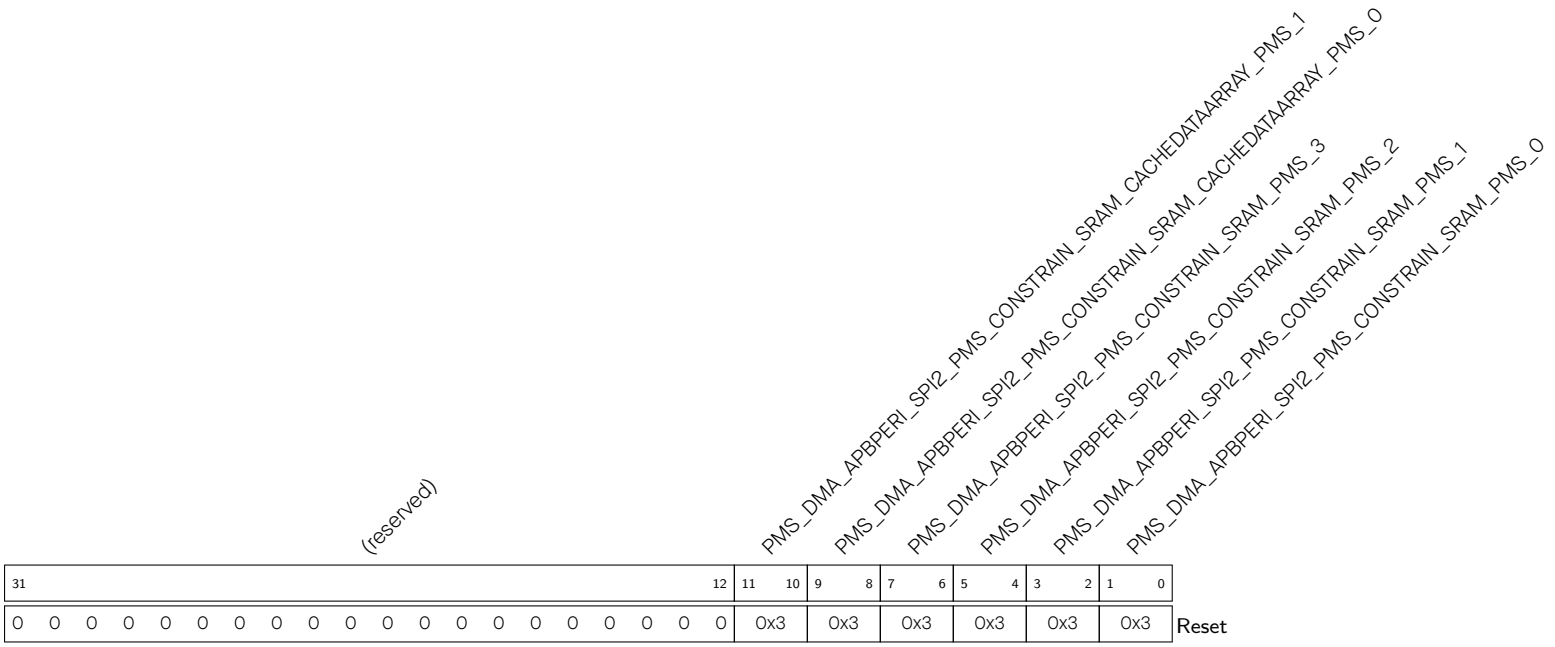
PMS_INTERNAL_SRAM_COREO_TRACE_ALLOC 在选中的 trace memory block 中选择特定 16 KB 作为 CPU0 的 trace memory。(R/W)

PMS_INTERNAL_SRAM_CORE1_TRACE_ALLOC 在选中的 trace memory block 中选择特定 16 KB 作为 CPU1 的 trace memory。(R/W)

Register 15.6. PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_0_REG (0x0038)

PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_LOCK 置 1 锁住 SPI2 的 GDMA 权限配置寄存器。(R/W)

Register 15.7. PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_1_REG (0x003C)



PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_PMS_0 配置 SPI2 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_PMS_1 配置 SPI2 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_PMS_2 配置 SPI2 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_PMS_3 配置 SPI2 对 SRAM 数据 region2 的权限。(R/W)

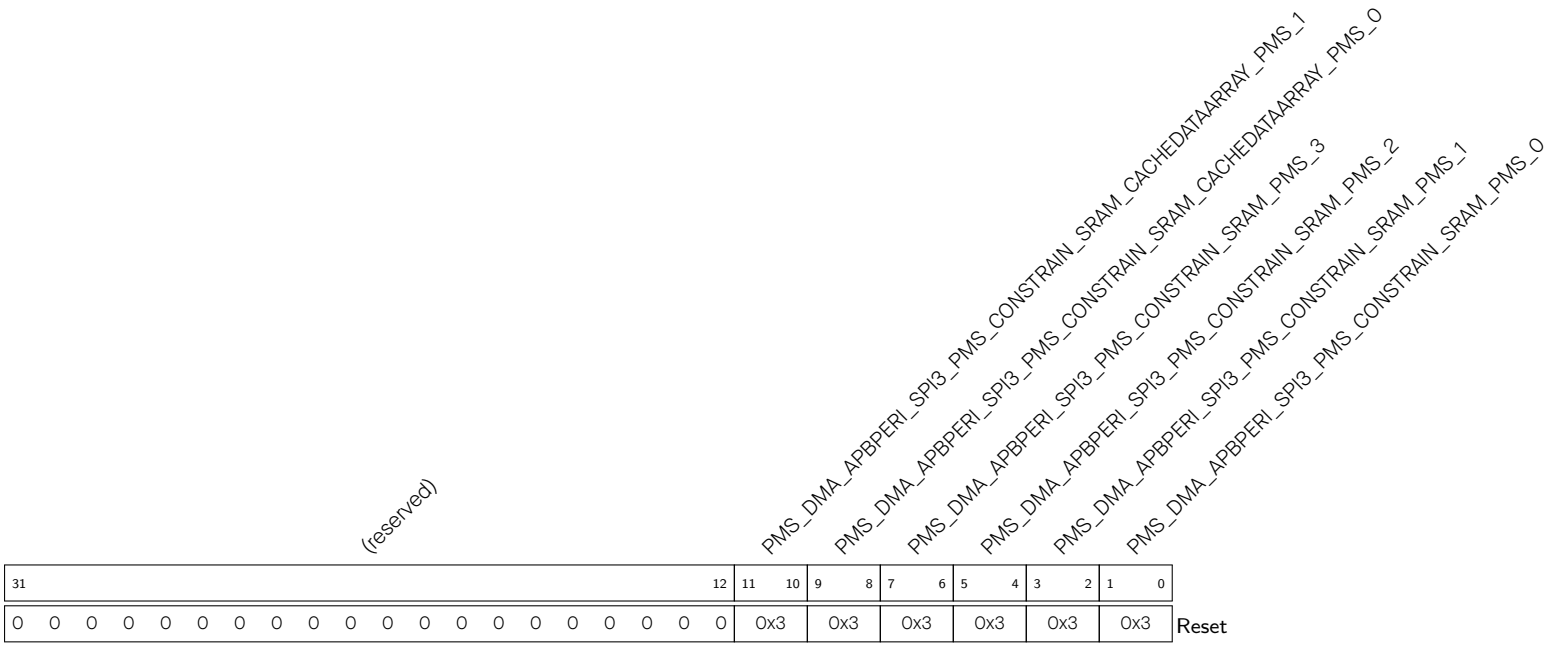
PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 SPI2 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 SPI2 对 SRAM block10 的权限。(R/W)

Register 15.8. PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_0_REG (0x0040)

PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_LOCK 置 1 锁住 SPI3 的 GDMA 权限配置寄存器。(R/W)

Register 15.9. PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_1_REG (0x0044)



PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_SRAM_PMS_0 配置 SPI3 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_SRAM_PMS_1 配置 SPI3 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_SRAM_PMS_2 配置 SPI3 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_SRAM_PMS_3 配置 SPI3 对 SRAM 数据 region2 的权限。(R/W)

PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 SPI3 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_SPI3_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 SPI3 对 SRAM block10 的权限。(R/W)

Register 15.10. PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_0_REG (0x0048)

PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_LOCK 置 1 锁住 UHCI0 的 GDMA 权限配置寄存器。(R/W)

Register 15.11. PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_1_REG (0x004C)

PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_SRAM_PMS_0 配置 UHCI0 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_SRAM_PMS_1 配置 UHCI0 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_SRAM_PMS_2 配置 UHCI0 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_SRAM_PMS_3 配置 UHCI0 对 SRAM 数据 region2 的权限。(R/W)

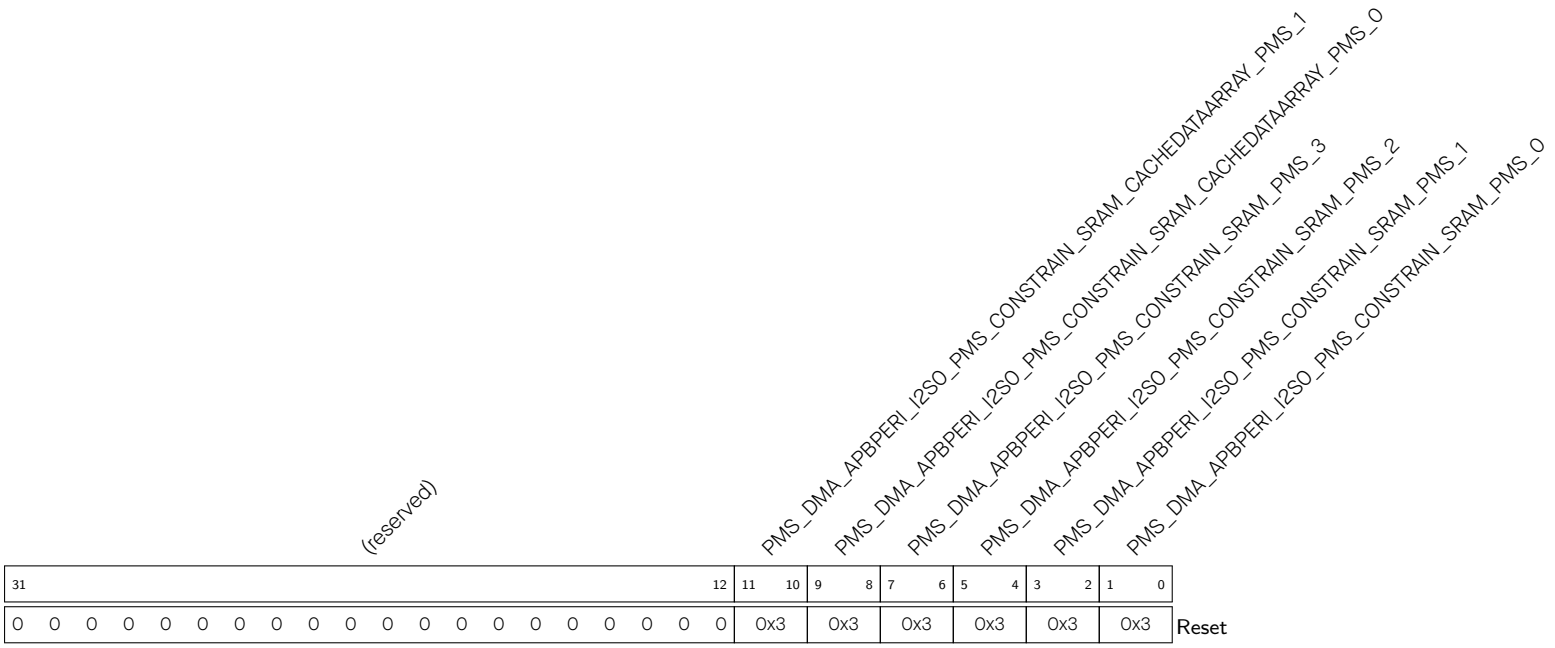
PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 UHCI0 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_UHCI0_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 UHCI0 对 SRAM block10 的权限。(R/W)

Register 15.12. PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_0_REG (0x0050)

PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_LOCK 置 1 锁住 I2S0 的 GDMA 权限配置寄存器。(R/W)

Register 15.13. PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_1_REG (0x0054)



PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_PMS_0 配置 I2S0 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_PMS_1 配置 I2S0 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_PMS_2 配置 I2S0 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_PMS_3 配置 I2S0 对 SRAM 数据 region2 的权限。(R/W)

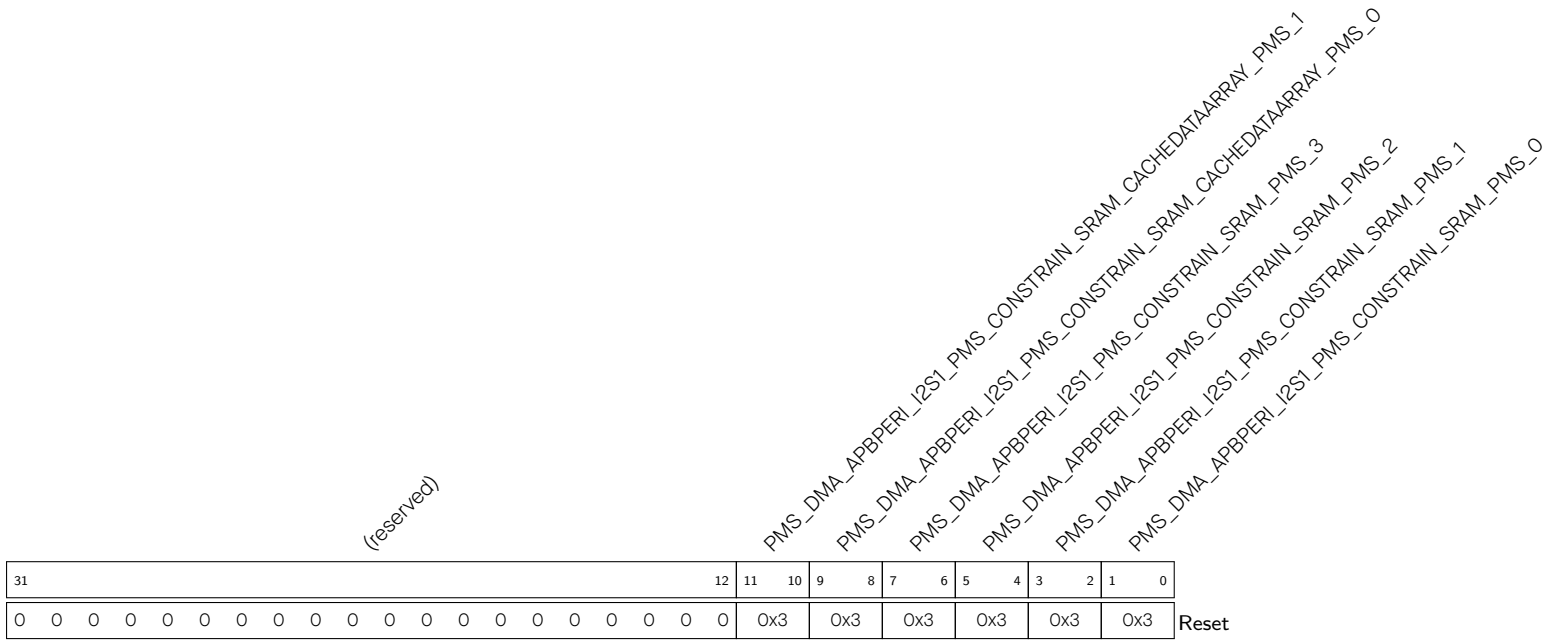
PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 I2S0 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 I2S0 对 SRAM block10 的权限。(R/W)

Register 15.14. PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_0_REG (0x0058)

PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_LOCK 置 1 锁住 I2S1 的 GDMA 权限配置寄存器.(R/W)

Register 15.15. PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_1_REG (0x005C)



PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_SRAM_PMS_0 配置 I2S1 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_SRAM_PMS_1 配置 I2S1 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_SRAM_PMS_2 配置 I2S1 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_SRAM_PMS_3 配置 I2S1 对 SRAM 数据 region2 权限。(R/W)

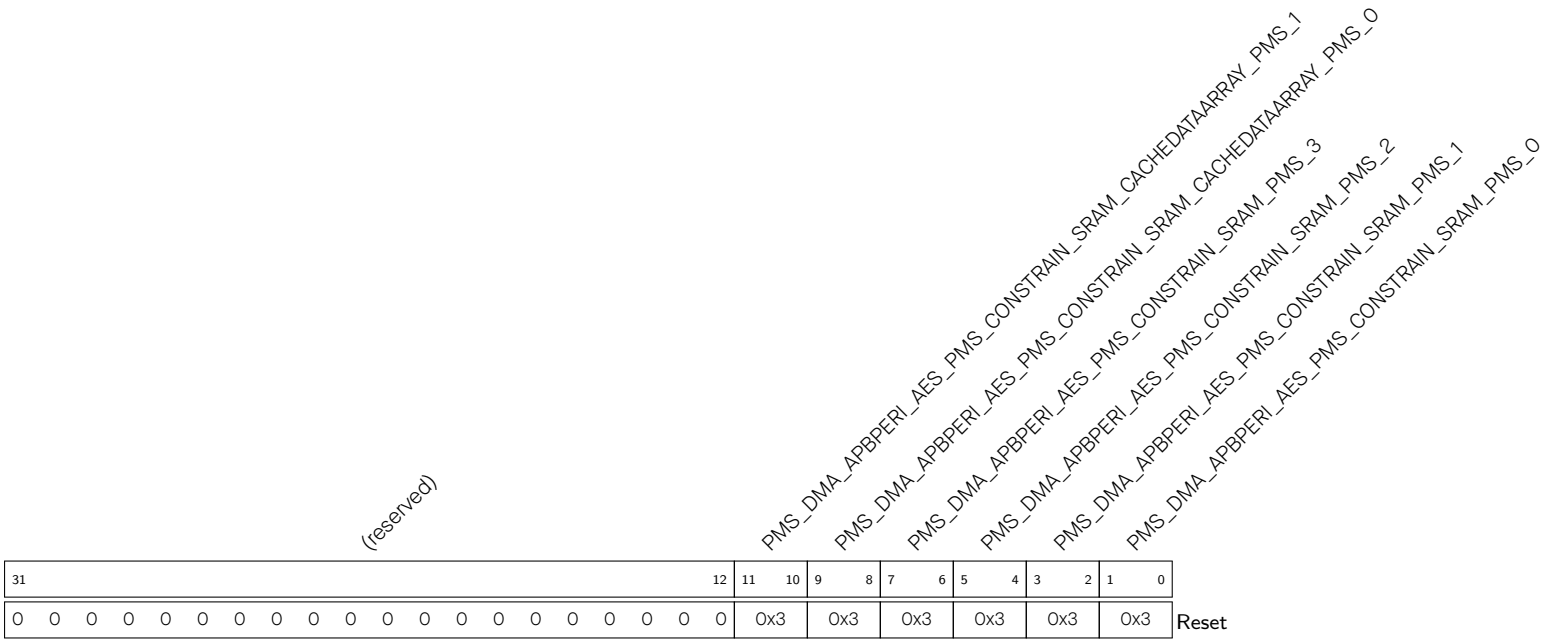
PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 I2S1 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_I2S1_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 I2S1 对 SRAM block10 的权限。(R/W)

Register 15.16. PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_0_REG (0x0070)

PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_LOCK 置 1 锁住 AES 的 GDMA 权限配置寄存器。(R/W)

Register 15.17. PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_1_REG (0x0074)



PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_PMS_0 配置 AES 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_PMS_1 配置 AES 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_PMS_2 配置 AES 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_PMS_3 配置 AES 对 SRAM 数据 region2 权限。(R/W)

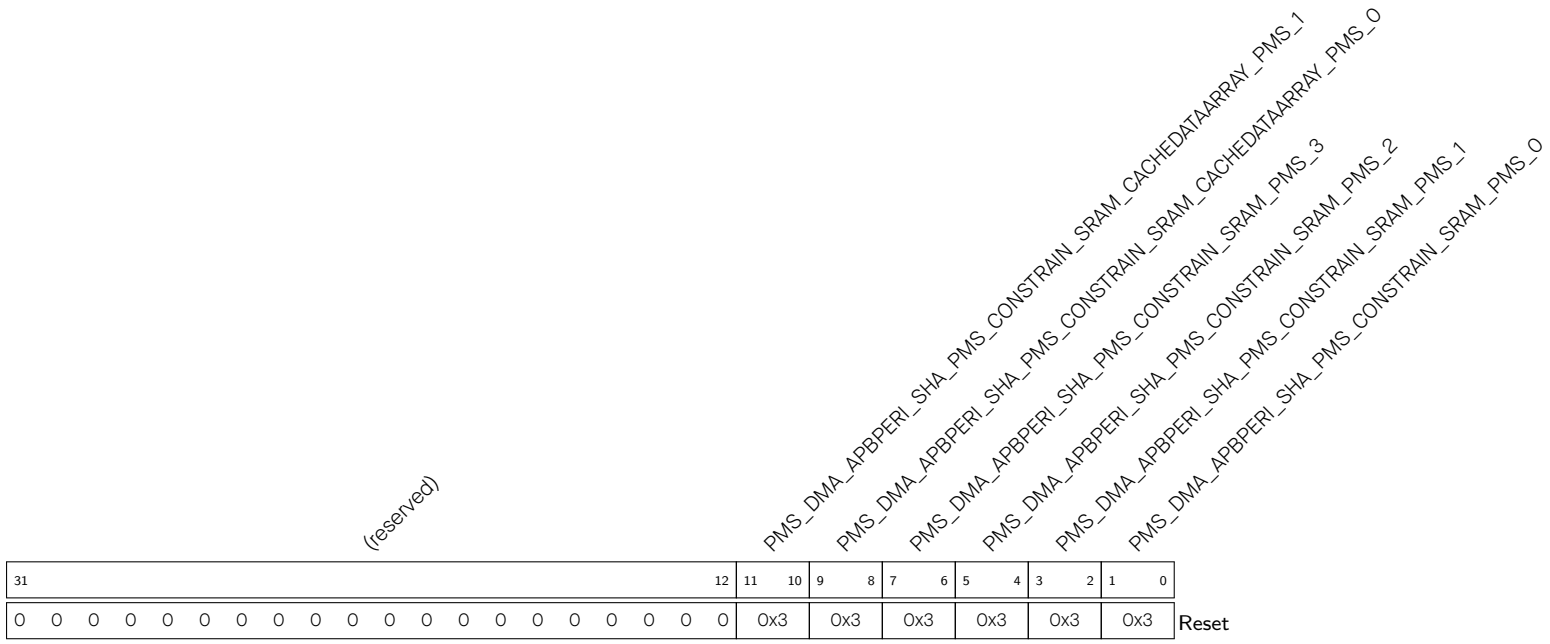
PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 AES 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 AES 对 SRAM block10 的权限。(R/W)

Register 15.18. PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_0_REG (0x0078)

PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_LOCK 置 1 锁住 SHA 的 GDMA 权限配置寄存器。(R/W)

Register 15.19. PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_1_REG (0x007C)



PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_PMS_0 配置 SHA 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_PMS_1 配置 SHA 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_PMS_2 配置 SHA 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_PMS_3 配置 SHA 对 SRAM 数据 region2 的权限。(R/W)

PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 SHA 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 SHA 对 SRAM block10 的权限。(R/W)

Register 15.20. PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_0_REG (0x0080)

PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_LOCK 置 1 锁住 ADC_DAC 的 GDMA 权限配置寄存器。(R/W)

Register 15.21. PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_1_REG (0x0084)

PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_PMS_0 配置 ADC_DAC 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_PMS_1 配置 ADC_DAC 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_PMS_2 配置 ADC_DAC 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_PMS_3 配置 ADC_DAC 对 SRAM 数据 region2 的权限。(R/W)

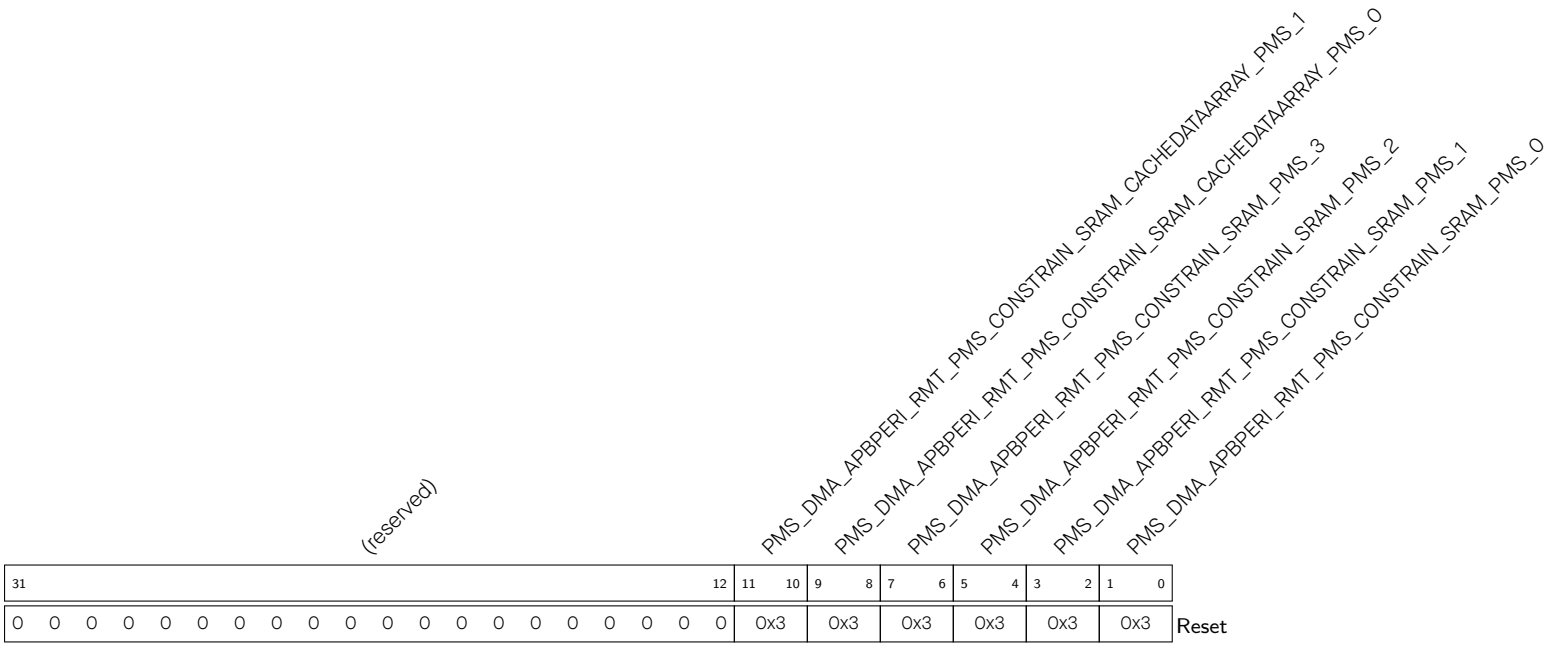
PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 ADC_DAC 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 ADC_DAC 对 SRAM block10 的权限。(R/W)

Register 15.22. PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_0_REG (0x0088)

PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_LOCK 置 1 锁住 RMT 的 GDMA 权限配置寄存器。(R/W)

Register 15.23. PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_1_REG (0x008C)



PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_SRAM_PMS_0 配置 RMT 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_SRAM_PMS_1 配置 RMT 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_SRAM_PMS_2 配置 RMT 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_SRAM_PMS_3 配置 RMT 对 SRAM 数据 region2 的权限。(R/W)

PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 RMT 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_RMT_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 RMT 对 SRAM block10 的权限。(R/W)

Register 15.24. PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_0_REG (0x0090)

PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_LOCK 置 1 锁住 LCD_CAM 的 GDMA 权限配置寄存器。(R/W)

Register 15.25. PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_1_REG (0x0094)

PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_SRAM_PMS_0 配置 LCD_CAM 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_SRAM_PMS_1 配置 LCD_CAM 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_SRAM_PMS_2 配置 LCD_CAM 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_SRAM_PMS_3 配置 LCD_CAM 对 SRAM 数据 region2 的权限。(R/W)

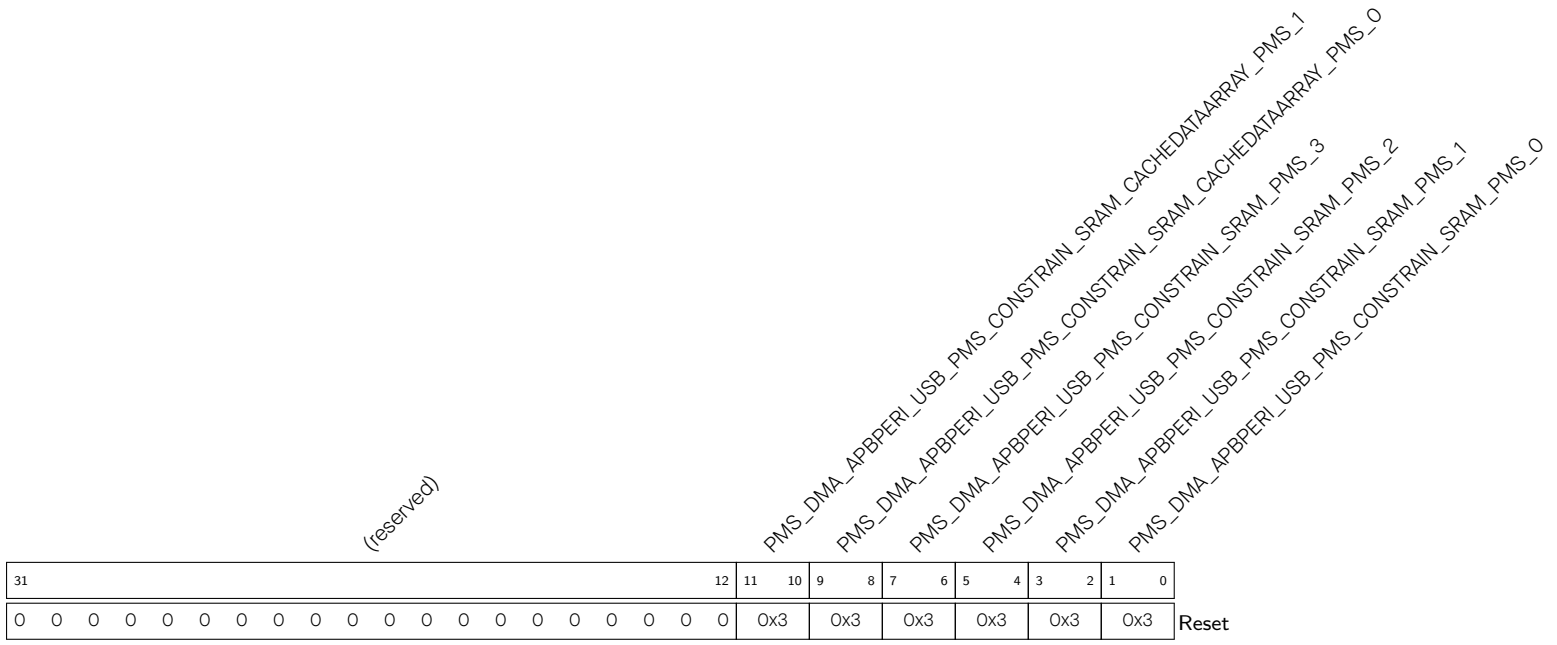
PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 LCD_CAM 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_LCD_CAM_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 LCD_CAM 对 SRAM block10 的权限。(R/W)

Register 15.26. PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_0_REG (0x0098)

PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_LOCK 置 1 锁住 USB 的 GDMA 权限配置寄存器。(R/W)

Register 15.27. PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_1_REG (0x009C)



PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_SRAM_PMS_0 配置 USB 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_SRAM_PMS_1 配置 USB 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_SRAM_PMS_2 配置 USB 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_SRAM_PMS_3 配置 USB 对 SRAM 数据 region2 的权限。(R/W)

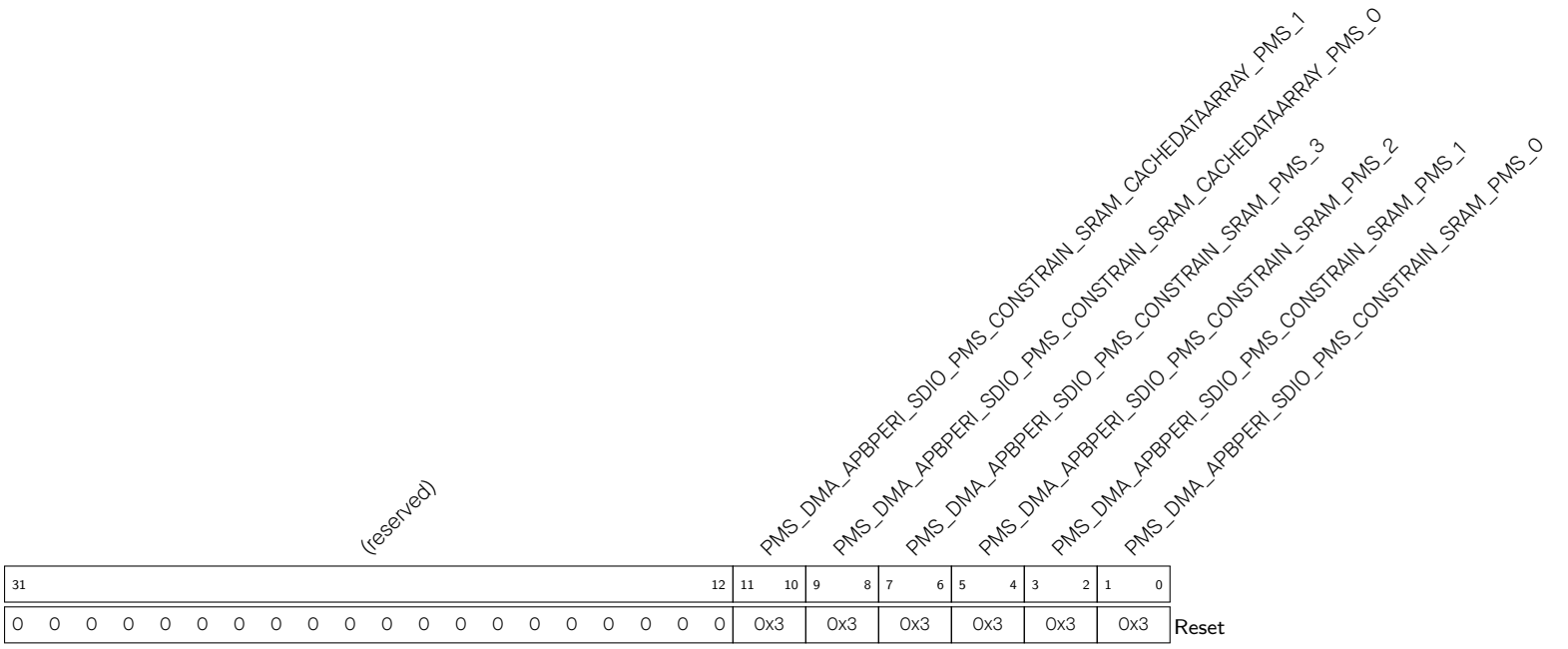
PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 USB 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_USB_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 USB 对 SRAM block10 的权限。(R/W)

Register 15.28. PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_0_REG (0x00A8)

PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_LOCK 置 1 锁住 SDIO 的 GDMA 权限配置寄存器。(R/W)

Register 15.29. PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_1_REG (0x00AC)



PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_SRAM_PMS_0 配置 SDIO 对 SRAM 指令空间的权限。(R/W)

PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_SRAM_PMS_1 配置 SDIO 对 SRAM 数据 region0 的权限。(R/W)

PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_SRAM_PMS_2 配置 SDIO 对 SRAM 数据 region1 的权限。(R/W)

PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_SRAM_PMS_3 配置 SDIO 对 SRAM 数据 region2 的权限。(R/W)

PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_0 配置 SDIO 对 SRAM block9 的权限。(R/W)

PMS_DMA_APBPERI_SDIO_PMS_CONSTRAIN_SRAM_CACHEDATAARRAY_PMS_1 配置 SDIO 对 SRAM block10 的权限。(R/W)

Register 15.30. PMS_DMA_APBPERI_PMS_MONITOR_0_REG (0x00B0)

PMS_DMA_APBPERI_PMS_MONITOR_LOCK 置 1 锁住 GDMA 权限中断配置寄存器。(R/W)

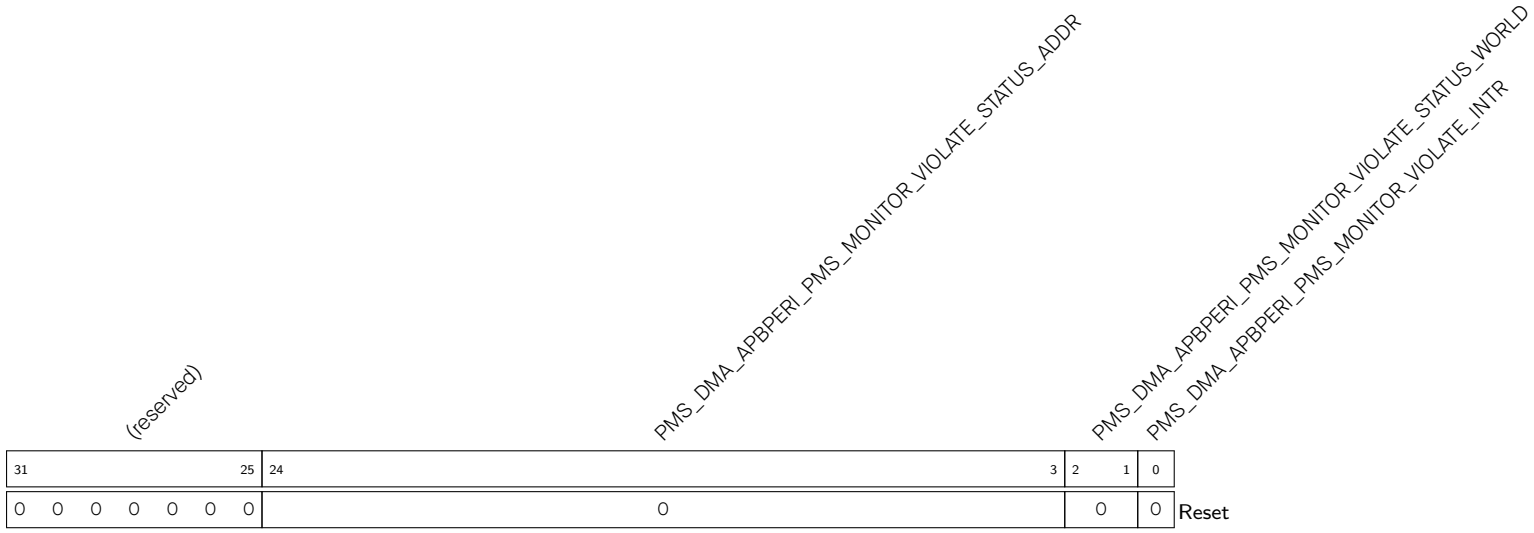
Register 15.31. PMS_DMA_APBPERI_PMS_MONITOR_1_REG (0x00B4)

PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_CLR 置 1 清除 GDMA 权限中断状态。(R/W)

PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_EN 置 1 使能非法 GDMA 访问中断。(R/W)

GoBack

Register 15.32. PMS_DMA_APBPERI_PMS_MONITOR_2_REG (0x00B8)



PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR 存储非法 GDMA 访问权限中断状态。(RO)

PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_STATUS_WORLD 存储发生非法 GDMA 访问时 CPU 所处的世界。0b01：安全世界；0b10：非安全世界。(RO)

PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_STATUS_ADDR 存储发生非法 GDMA 访问地址。注意，此地址为相对于 0x3c000000 的偏移地址，且单位为 16，即实际地址应为 0x3c000000 + PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_STATUS_ADDR * 16。(RO)

Register 15.33. PMS_DMA_APBPERI_PMS_MONITOR_3_REG (0x00BC)

PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_STATUS_WR 存储非法 GDMA 访问的方向。1：写；0：读。(RO)

PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_STATUS_BYTEEN 存储非法 GDMA 访问的字节信息。(RO)

Register 15.34. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_0_REG (0x00C0)

PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_LOCK 置 1 锁住内部 SRAM 分割线的配置寄存器。(R/W)

Register 15.35. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_1_REG (0x00C4)

PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_0 配置指令和数据分割线 IRAM0_DRAM0_Split_Line block2 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_1 配置指令和数据分割线 IRAM0_DRAM0_Split_Line block3 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_2 配置指令和数据分割线 IRAM0_DRAM0_Split_Line block4 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_3 配置指令和数据分割线 IRAM0_DRAM0_Split_Line block5 的 category 字段。(R/W)

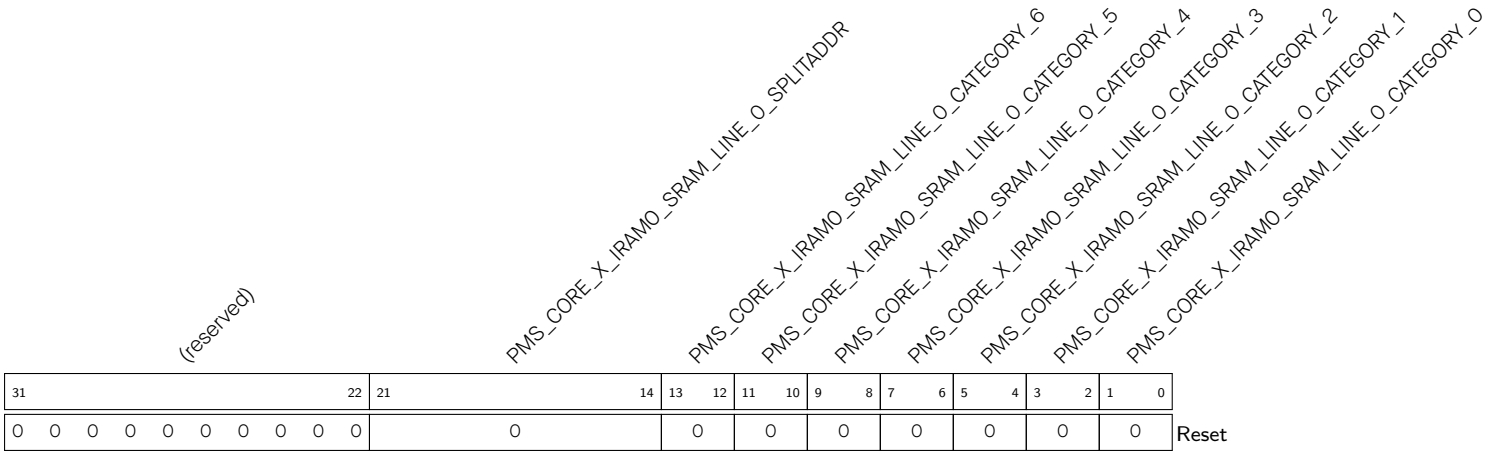
PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_4 配置指令和数据分割线 IRAM0_DRAM0_Split_Line block6 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_5 配置指令和数据分割线 IRAM0_DRAM0_Split_Line block7 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_6 配置指令和数据分割线 IRAM0_DRAM0_Split_Line block8 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_SPLITADDR 配置指令和数据分割线 IRAM0_DRAM0_Split_Line 的分割地址。(R/W)

Register 15.36. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_2_REG (0x00C8)



PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_0 配置指令内部分割线 IRAM0_Split_Line_0 block2 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_1 配置指令内部分割线 IRAM0_Split_Line_0 block3 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_2 配置指令内部分割线 IRAM0_Split_Line_0 block4 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_3 配置指令内部分割线 IRAM0_Split_Line_0 block5 的 category 字段。(R/W)

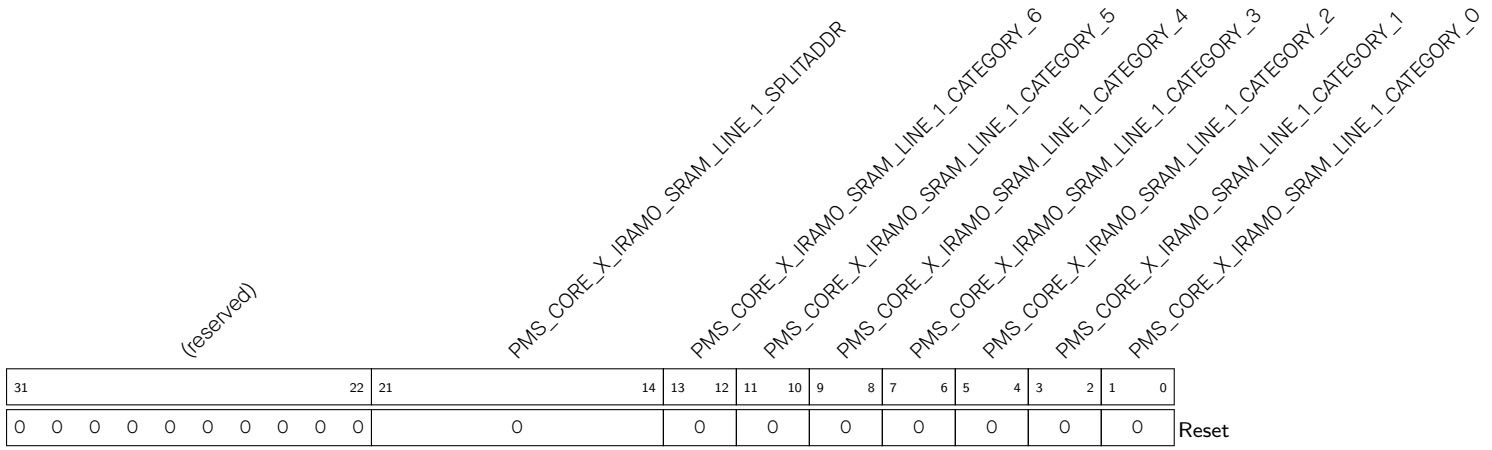
PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_4 配置指令内部分割线 IRAM0_Split_Line_0 block6 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_5 配置指令内部分割线 IRAM0_Split_Line_0 block7 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_6 配置指令内部分割线 IRAM0_Split_Line_0 block8 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_0_SPLITADDR 配置指令内部分割线 IRAM0_Split_Line_0 的分割地址。(R/W)

Register 15.37. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_3_REG (0x00CC)



PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_0 配置指令内部分割线 IRAM0_Split_Line_1 block2 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_1 配置指令内部分割线 IRAM0_Split_Line_1 block3 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_2 配置指令内部分割线 IRAM0_Split_Line_1 block4 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_3 配置指令内部分割线 IRAM0_Split_Line_1 block5 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_4 配置指令内部分割线 IRAM0_Split_Line_1 block6 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_5 配置指令内部分割线 IRAM0_Split_Line_1 block7 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_6 配置指令内部分割线 IRAM0_Split_Line_1 block8 的 category 字段。(R/W)

PMS_CORE_X_IRAM0_SRAM_LINE_1_SPLITADDR 配置指令内部分割线 IRAM0_Split_Line_1 的分割地址。(R/W)

Register 15.38. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_4_REG (0x00D0)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_0 配置数据内部分割线 DRAM0_Split_Line_0 block2 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_1 配置数据内部分割线 DRAM0_Split_Line_0 block3 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_2 配置数据内部分割线 DRAM0_Split_Line_0 block4 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_3 配置数据内部分割线 DRAM0_Split_Line_0 block5 的 category 字段。(R/W)

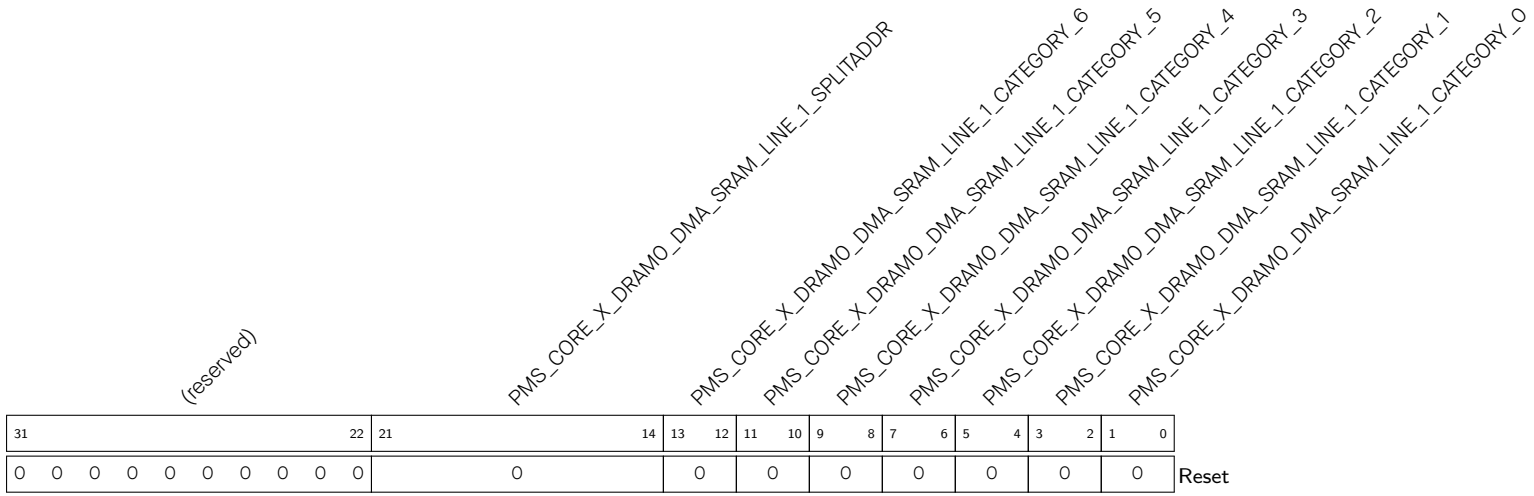
PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_4 配置数据内部分割线 DRAM0_Split_Line_0 block6 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_5 配置数据内部分割线 DRAM0_Split_Line_0 block7 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_6 配置数据内部分割线 DRAM0_Split_Line_0 block8 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_SPLITADDR 配置数据内部分割线 DRAM0_Split_Line_0 的分割地址。(R/W)

Register 15.39. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_5_REG (0x00D4)



PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_0 配置数据内部分割线 DRAM0_Split_Line_1 block2 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_1 配置数据内部分割线 DRAM0_Split_Line_1 block3 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_2 配置数据内部分割线 DRAM0_Split_Line_1 block4 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_3 配置数据内部分割线 DRAM0_Split_Line_1 block5 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_4 配置数据内部分割线 DRAM0_Split_Line_1 block6 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_5 配置数据内部分割线 DRAM0_Split_Line_1 block7 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_6 配置数据内部分割线 DRAM0_Split_Line_1 block8 的 category 字段。(R/W)

PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_SPLITADDR 配置数据内部分割线 DRAM0_Split_Line_1 的分割地址。(R/W)

Register 15.40. PMS_CORE_X_IRAM0_PMS_CONSTRAIN_0_REG (0x00D8)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_LOCK 置 1 锁住 CPU 的 IBUS 对内部 SRAM 的权限配置。(R/W)

Register 15.41. PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG (0x00DC)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_PMS_0 配置 CPU 的 IBUS 从非安全世界对 SRAM 指令 region0 的权限。(R/W)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_PMS_1 配置 CPU 的 IBUS 从非安全世界对 SRAM 指令 region1 的权限。(R/W)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_PMS_2 配置 CPU 的 IBUS 从非安全世界对 SRAM 指令 region2 的权限。(R/W)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_PMS_3 配置 CPU 的 IBUS 从非安全世界对 SRAM 数据空间的权限，建议配置为 0。(R/W)

Continued on the next page...

Register 15.41. PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG (0x00DC)

Continued from the previous page...

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_CACHEDATAARRAY_PMS_0 配置 CPU 的 IBUS 从安全世界对 SRAM0 block0 的权限。(R/W)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_CACHEDATAARRAY_PMS_1 配置 CPU 的 IBUS 从安全世界对 SRAM0 block1 的权限。(R/W)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_ROM_WORLD_1_PMS 配置 CPU 的 IBUS 从非安全世界对 ROM 的权限。(R/W)

Register 15.42. PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG (0x00E0)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_PMS_0 配置 CPU 的 IBUS 从安全世界对 SRAM 指令 region0 的权限。(R/W)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_PMS_1 配置 CPU 的 IBUS 从安全世界对 SRAM 指令 region1 的权限。(R/W)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_PMS_2 配置 CPU 的 IBUS 从安全世界对 SRAM 指令 region2 的权限。(R/W)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_PMS_3 配置 CPU 的 IBUS 从安全世界对 SRAM 数据空间的权限，建议配置为 0。(R/W)

Continued on the next page...

Register 15.42. PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG (0x00E0)

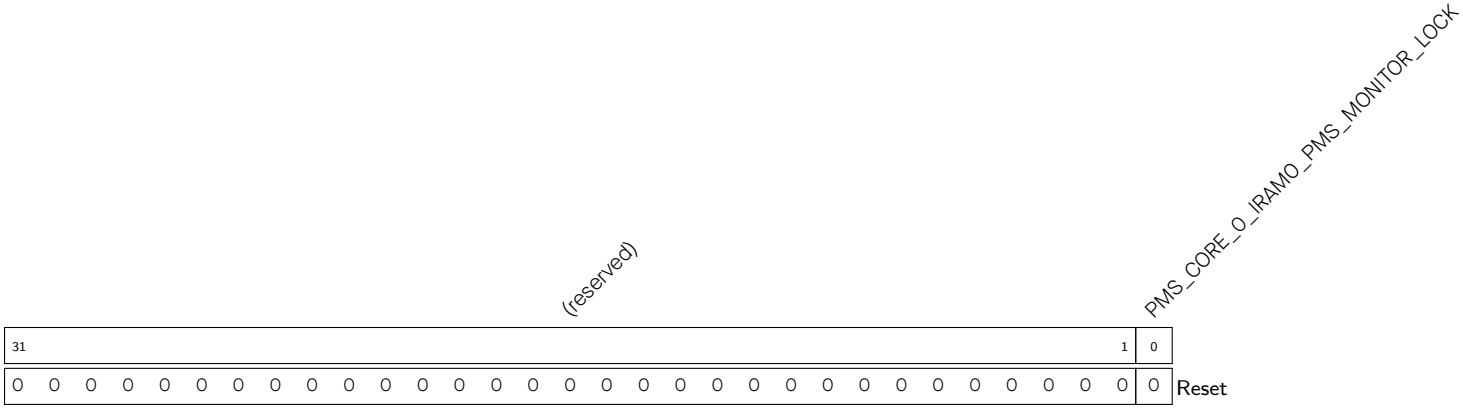
Continued from the previous page...

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_CACHEDATAARRAY_PMS_0 配置 CPU 的 IBUS 从安全世界对 SRAM0 block0 的权限。(R/W)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_CACHEDATAARRAY_PMS_1 配置 CPU 的 IBUS 从安全世界对 SRAM0 block1 的权限。(R/W)

PMS_CORE_X_IRAM0_PMS_CONSTRAIN_ROM_WORLD_0_PMS 配置 CPU 的 IBUS 从安全世界对 ROM 的权限。(R/W)

Register 15.43. PMS_CORE_0_IRAM0_PMS_MONITOR_0_REG (0x00E4)



PMS_CORE_0_IRAM0_PMS_MONITOR_LOCK 置 1 锁住 CPU0 IBUS 的中断配置。(R/W)

Register 15.44. PMS_CORE_0_IRAM0_PMS_MONITOR_1_REG (0x00E8)

PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_CLR 置 1 清除 CPU0 的 iBUS 非法访问 SRAM 或 ROM 发生的中断。(R/W)

PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_EN 置 1 使能在 CPU0 的 iBUS 非法访问 SRAM 或 ROM 时触发中断。(R/W)

Register 15.45. PMS_CORE_X_DRAM0_PMS_CONSTRAIN_0_REG (0x00FC)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_LOCK 置 1 锁住 CPU 的 DBUS 对内部 SRAM 的权限配置。(R/W)

Register 15.46. PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG (0x0100)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_PMS_0 配置 CPU 的 DBUS 从安全世界对 SRAM 指令空间的权限，建议配置为 0。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_PMS_1 配置 CPU 的 DBUS 从安全世界对 SRAM 数据 region0 的权限。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_PMS_2 配置 CPU 的 DBUS 从安全世界对 SRAM 数据 region1 的权限。(R/W)

Continued on the next page...

Register 15.46. PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG (0x0100)

Continued from the previous page...

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_PMS_3 配置 CPU 的 DBUS 从安全世界对 SRAM 数据 region2 的权限。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_CACHEDATAARRAY_PMS_0 配置 CPU 的 DBUS 从安全世界对 SRAM2 block9 的权限。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_0_CACHEDATAARRAY_PMS_1 配置 CPU 的 DBUS 从安全世界对 SRAM2 block10 的权限。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_PMS_0 配置 CPU 的 DBUS 从非安全世界对 SRAM 指令空间的权限。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_PMS_1 配置 CPU 的 DBUS 从非安全世界对 SRAM 数据 region0 的权限。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_PMS_2 配置 CPU 的 DBUS 从非安全世界对 SRAM 数据 region1 的权限。(R/W)(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_PMS_3 配置 CPU 的 DBUS 从非安全世界对 SRAM 数据 region2 的权限。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_CACHEDATAARRAY_PMS_0 配置 CPU 的 DBUS 从非安全世界对 SRAM2 block9 的权限。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_WORLD_1_CACHEDATAARRAY_PMS_1 配置 CPU 的 DBUS 从非安全世界对 SRAM2 block10 的权限。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_ROM_WORLD_0_PMS 配置 CPU 的 DBUS 从非安全世界对 ROM 的权限。(R/W)

PMS_CORE_X_DRAM0_PMS_CONSTRAIN_ROM_WORLD_1_PMS 配置 CPU 的 DBUS 从非安全世界对 ROM 的权限。(R/W)

GoBack

Register 15.47. PMS_CORE_0_DRAM0_PMS_MONITOR_0_REG (0x0104)

PMS_CORE_0_DRAM0_PMS_MONITOR_LOCK 置 1 锁住 CPU0 的 DBUS 非法访问中断配置。(R/W)

Register 15.48. PMS_CORE_0_DRAM0_PMS_MONITOR_1_REG (0x0108)

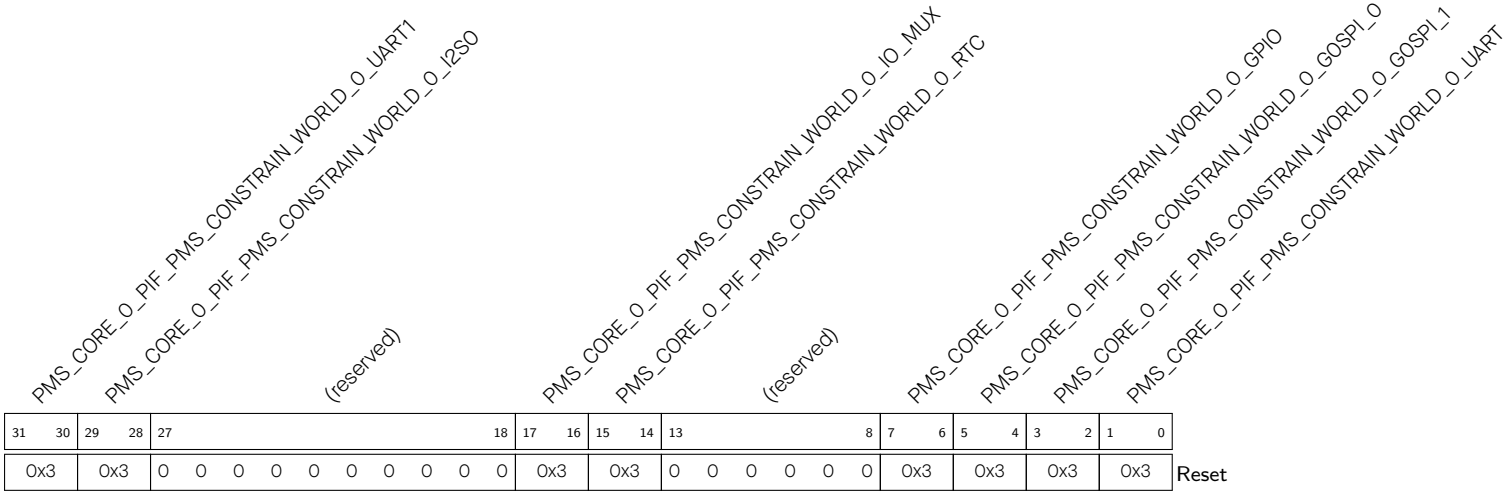
PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_CLR 置 1 清除 CPU0 的 DBUS 非法访问 SRAM 或 ROM 时触发的中断。(R/W)

PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_EN 置 1 使能在 CPU0 的 DBUS 非法访问 SRAM 或 ROM 时触发中断。(R/W)

Register 15.49. PMS_CORE_0_PIF_PMS_CONSTRAIN_0_REG (0x0124)

PMS_CORE_0_PIF_PMS_CONSTRAIN_LOCK 置 1 锁住 CPU0 访问不同外设的权限配置。(R/W)

Register 15.50. PMS_CORE_0_PIF_PMS_CONSTRAIN_*n*_REG (*n*: 1 - 8) (0x0128 + 4**n*)



PMS_CORE_0_PIF_PMS_CONSTRAIN_WORLD_0_UART 配置 CPU0 从安全世界访问 UART0 的权限。

PMS_CORE_0_PIF_PMS_CONSTRAIN_WORLD_0_GOSPI_1 配置 CPU0 从安全世界访问 SPI1 的权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_WORLD_0_GOSPI_0 配置 CPU0 从安全世界访问 SPI0 的权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_WORLD_0_GPIO 配置 CPU0 从安全世界访问 GPIO 的权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_WORLD_0_RTC 配置 CPU0 从安全世界访问 eFuse 控制器 & PMU 的权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_WORLD_0_IO_MUX 配置 CPU0 从安全世界访问 IO_MUX 的权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_WORLD_0_I2S0 配置 CPU0 从安全世界访问 I2S0 的权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_WORLD_0_UART1 配置 CPU0 从安全世界访问 UART1 的权限。(R/W)

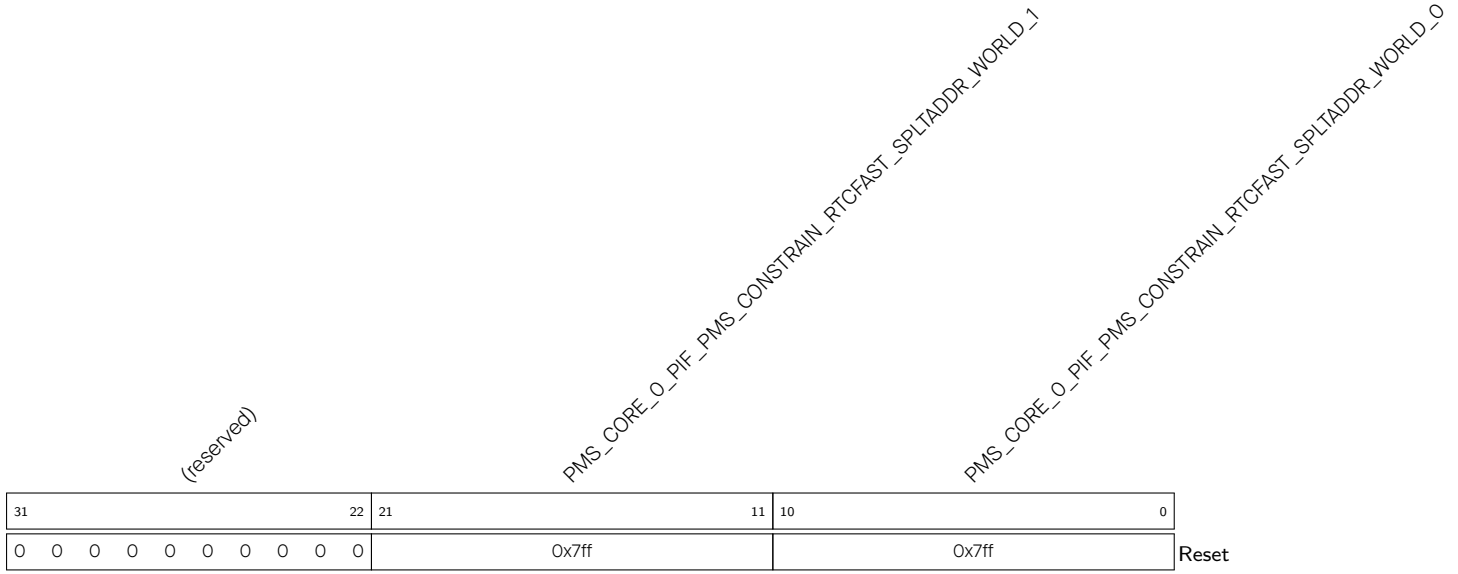
说明：

- 寄存器 PMS_CORE_0_PIF_PMS_CONSTRAIN_*n*_REG (*n*: 1 - 4) 用于配置 CPU0 从安全世界访问不同外设的权限。

GoBack

- 寄存器 PMS_CORE_0_PIF_PMS_CONSTRAIN_*n*_REG (*n*: 5 - 8) 用于配置 CPU0 从非安全世界访问不同外设的权限。
- 详细信息，请见表 17.3-2，这里不再赘述。

Register 15.51. PMS_CORE_0_PIF_PMS_CONSTRAIN_9_REG (0x0148)



PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_SPLTADDR_WORLD_0 配置针对 CPU0 从非安全世界访问权限的 RTC 快速内存分割地址。注意，配置时请采用地址偏移量，而非绝对地址。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_SPLTADDR_WORLD_1 配置针对 CPU0 从安全世界访问权限的 RTC 快速内存分割地址。注意，配置时请采用地址偏移量，而非绝对地址。(R/W)

Register 15.52. PMS_CORE_0_PIF_PMS_CONSTRAIN_10_REG (0x014C)

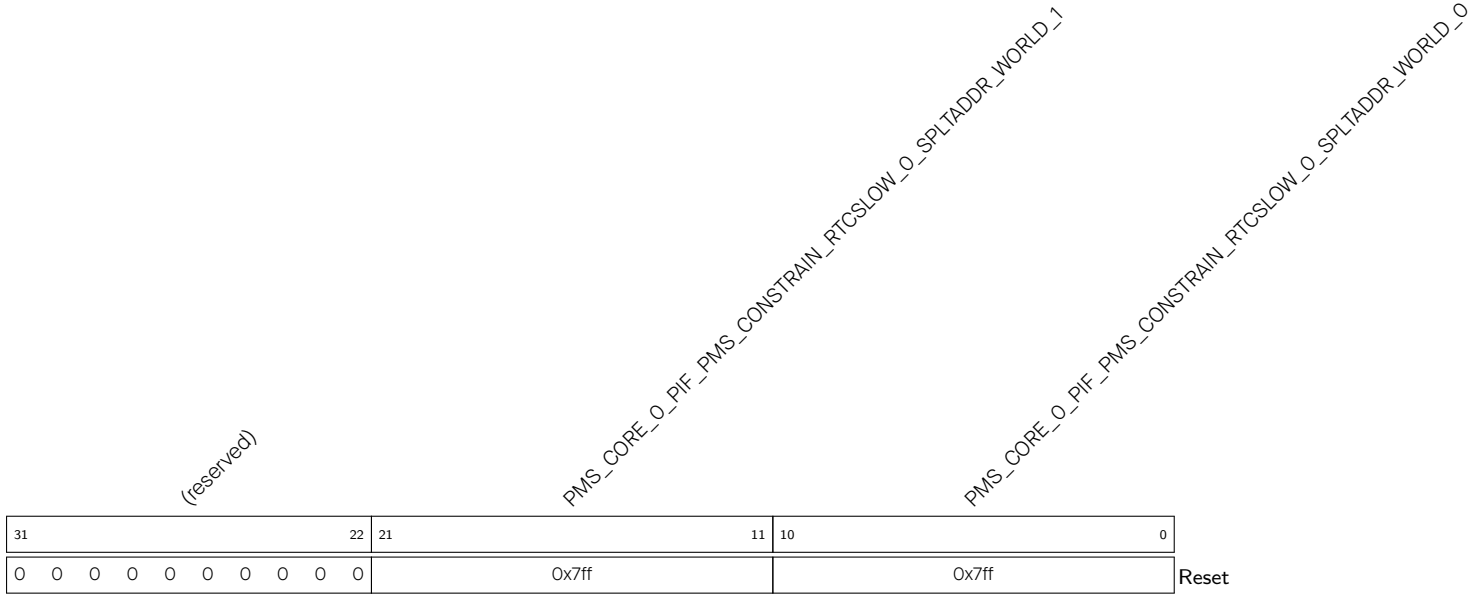
PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_WORLD_0_L 配置 CPU0 从非安全世界对 RTC 快速内存低区的访问权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_WORLD_0_H 配置 CPU0 从非安全世界对 RTC 快速内存高区的访问权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_WORLD_1_L 配置 CPU0 从安全世界对 RTC 快速内存低区的访问权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_WORLD_1_H 配置 CPU0 从安全世界对 RTC 快速内存高区的访问权限。(R/W)

Register 15.53. PMS_CORE_0_PIF_PMS_CONSTRAIN_11_REG (0x0150)



PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_0_SPLTADDR_WORLD_0 配置针对 CPU0 从安全世界访问权限的 RTC 慢速内存 0 分割地址。注意，配置时请采用地址偏移量，而非绝对地址。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_0_SPLTADDR_WORLD_1 配置针对 CPU0 从非安全世界访问权限的 RTC 慢速内存 0 分割地址。注意，配置时请采用地址偏移量，而非绝对地址。(R/W)

Register 15.54. PMS_CORE_0_PIF_PMS_CONSTRAIN_12_REG (0x0154)

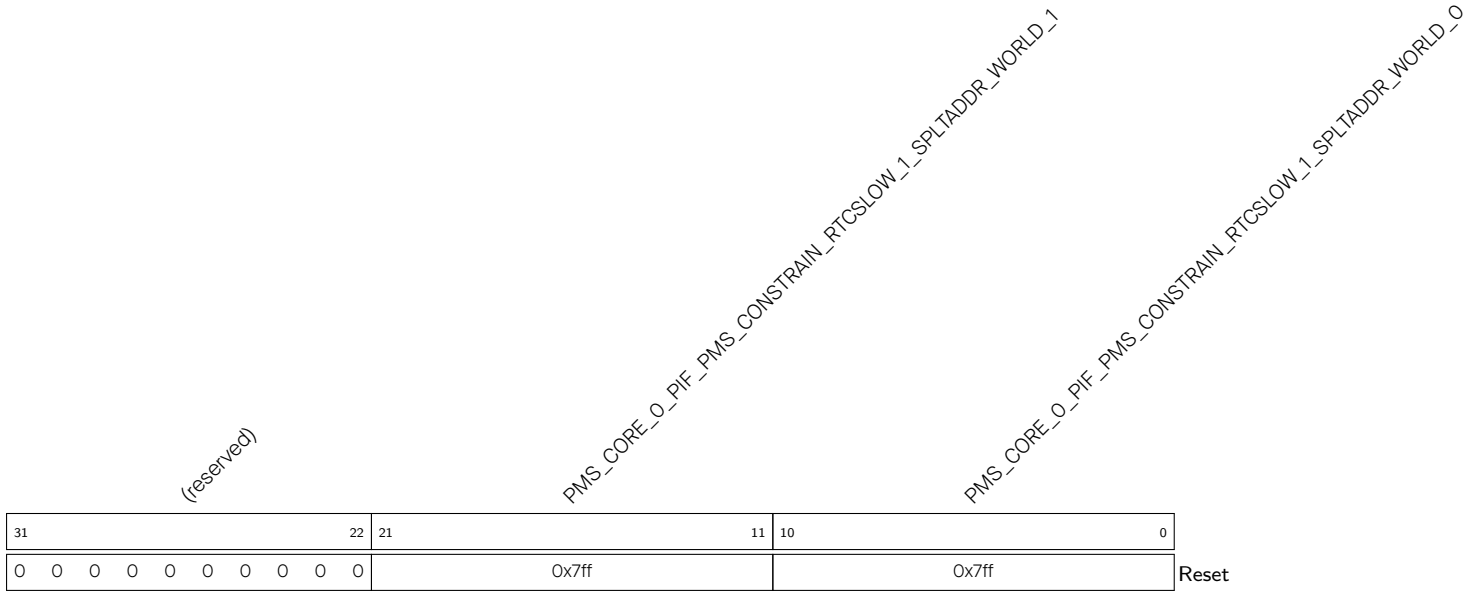
PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_0_WORLD_0_L 配置 CPU0 从安全世界对 RTC 慢速内存 0 低区的访问权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_0_WORLD_0_H 配置 CPU0 从安全世界对 RTC 慢速内存 0 高区的访问权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_0_WORLD_1_L 配置 CPU0 从非安全世界对 RTC 慢速内存 0 低区的访问权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_0_WORLD_1_H 配置 CPU0 从非安全世界对 RTC 慢速内存 0 低区的访问权限。(R/W)

Register 15.55. PMS_CORE_0_PIF_PMS_CONSTRAIN_13_REG (0x0158)



PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_1_SPLTADDR_WORLD_0 配置针对 CPU0 从安全世界访问权限的 RTC 慢速内存 1 分割地址。注意，配置时请采用地址偏移量，而非绝对地址。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_1_SPLTADDR_WORLD_1 配置针对 CPU0 从非安全世界访问权限的 RTC 慢速内存 1 分割地址。注意，配置时请采用地址偏移量，而非绝对地址。(R/W)

Register 15.56. PMS_CORE_0_PIF_PMS_CONSTRAIN_14_REG (0x015C)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_1_WORLD_0_L 配置 CPU0 从非安全世界对 RTC 慢速内存 1 低区的访问权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_1_WORLD_0_H 配置 CPU0 从非安全世界对 RTC 慢速内存 1 高区的访问权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_1_WORLD_1_L 配置 CPU0 从安全世界对 RTC 慢速内存 1 低区的访问权限。(R/W)

PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCSLOW_1_WORLD_1_H 配置 CPU0 从安全世界对 RTC 慢速内存 1 高区的访问权限。(R/W)

Register 15.57. PMS_CORE_0_REGION_PMS_CONSTRAIN_0_REG (0x0160)

PMS_CORE_0_REGION_PMS_CONSTRAIN_LOCK 置 1 锁住 CPU0 对外设区域的访问权限配置。(R/W)

Register 15.58. PMS_CORE_0_REGION_PMS_CONSTRAIN_1_REG (0x0164)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_0 配置 CPU0 从安全世界对 Peri Region0 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_1 配置 CPU0 从安全世界对 Peri Region1 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_2 配置 CPU0 从安全世界对 Peri Region2 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_3 配置 CPU0 从安全世界对 Peri Region3 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_4 配置 CPU0 从安全世界对 Peri Region4 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_5 配置 CPU0 从安全世界对 Peri Region5 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_6 配置 CPU0 从安全世界对 Peri Region6 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_7 配置 CPU0 从安全世界对 Peri Region7 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_8 配置 CPU0 从安全世界对 Peri Region8 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_9 配置 CPU0 从安全世界对 Peri Region9 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_0_AREA_10 配置 CPU0 从安全世界对 Peri Region10 的访问权限。(R/W)

Register 15.59. PMS_CORE_0_REGION_PMS_CONSTRAIN_2_REG (0x0168)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_0 配置 CPU0 从非安全世界对 Peri Region0 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_1 配置 CPU0 从非安全世界对 Peri Region1 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_2 配置 CPU0 从非安全世界对 Peri Region2 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_3 配置 CPU0 从非安全世界对 Peri Region3 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_4 配置 CPU0 从非安全世界对 Peri Region4 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_5 配置 CPU0 从非安全世界对 Peri Region5 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_6 配置 CPU0 从非安全世界对 Peri Region6 的访问权限。(R/W)

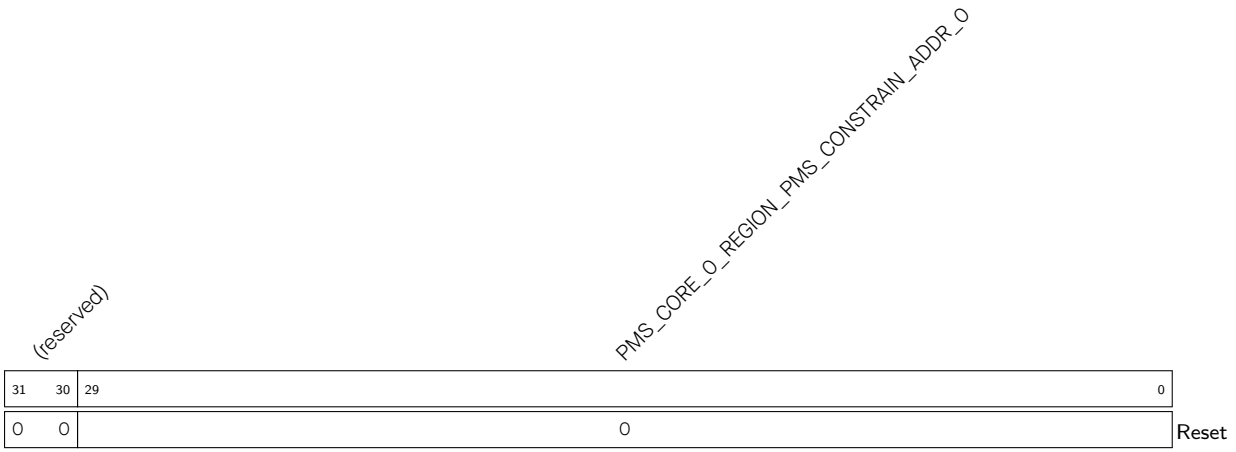
PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_7 配置 CPU0 从非安全世界对 Peri Region7 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_8 配置 CPU0 从非安全世界对 Peri Region8 的访问权限。(R/W)

PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_9 配置 CPU0 从非安全世界对 Peri Region9 的访问权限。(R/W)

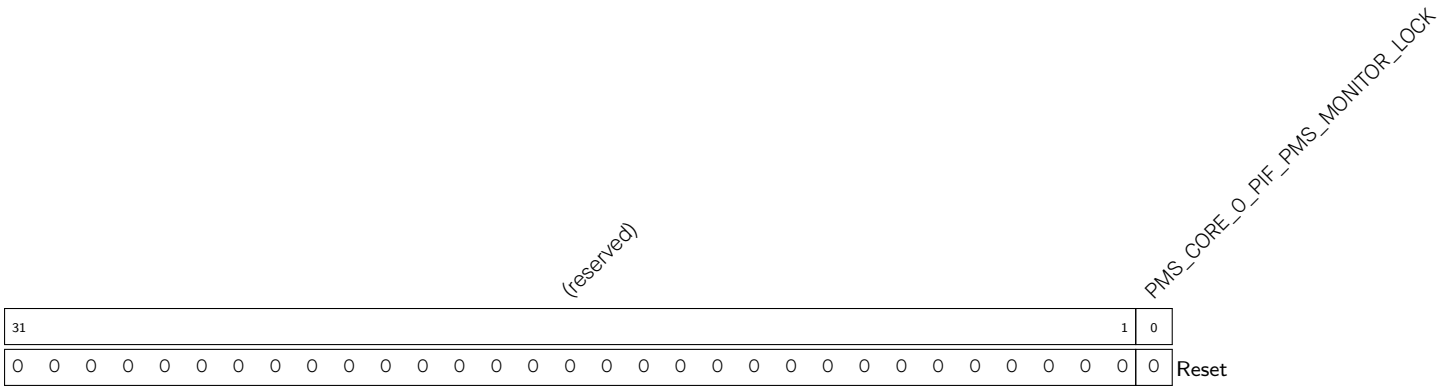
PMS_CORE_0_REGION_PMS_CONSTRAIN_WORLD_1_AREA_10 配置 CPU0 从非安全世界对 Peri Region10 的访问权限。(R/W)

Register 15.60. PMS_CORE_0_REGION_PMS_CONSTRAIN_*n*_REG (*n*: 3 - 14) (0x016C + 4*n*)



PMS_CORE_0_REGION_PMS_CONSTRAIN_ADDR_0 配置针对 CPU0 的 Region 0 的起始地址。(R/W)

Register 15.61. PMS_CORE_0_PIF_PMS_MONITOR_0_REG (0x019C)



PMS_CORE_0_PIF_PMS_MONITOR_LOCK 置 1 锁住 CPU0 的 PIF 非法访问中断配置。(R/W)

Register 15.62. PMS_CORE_0_PIF_PMS_MONITOR_1_REG (0x01A0)

PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_CLR 置 1 清除 CPU0 的 PIF 总线非法访问 RTC 内存或外设时触发的中断。(R/W)

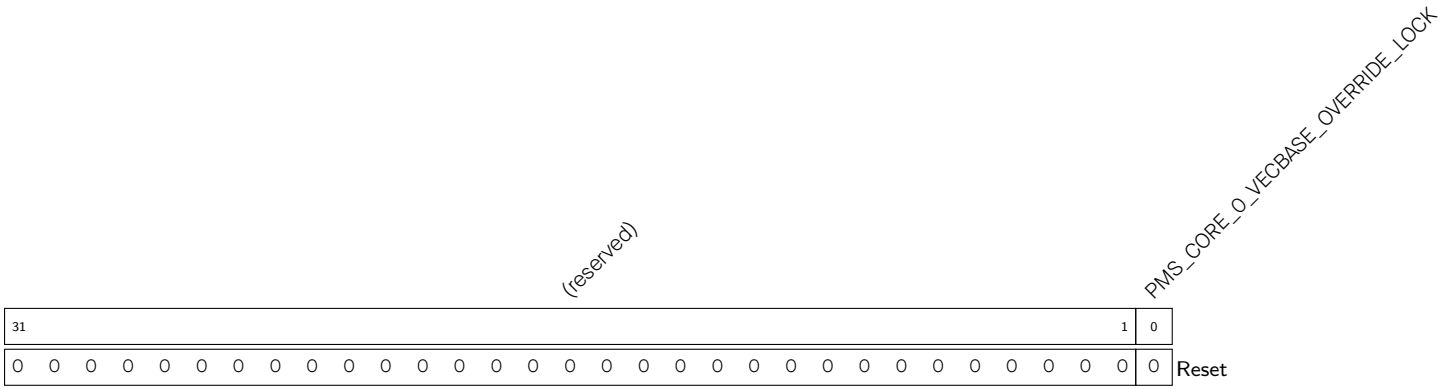
PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_EN 置 1 使能在 CPU0 的 PIF 总线非法访问 RTC 内存或外设时触发中断。(R/W)

Register 15.63. PMS_CORE_0_PIF_PMS_MONITOR_4_REG (0x01AC)

PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_CLR 置 1 清除 CPU0 的 PIF 总线使用非法数据类型访问 RTC 内存或外设时触发的中断。(R/W)

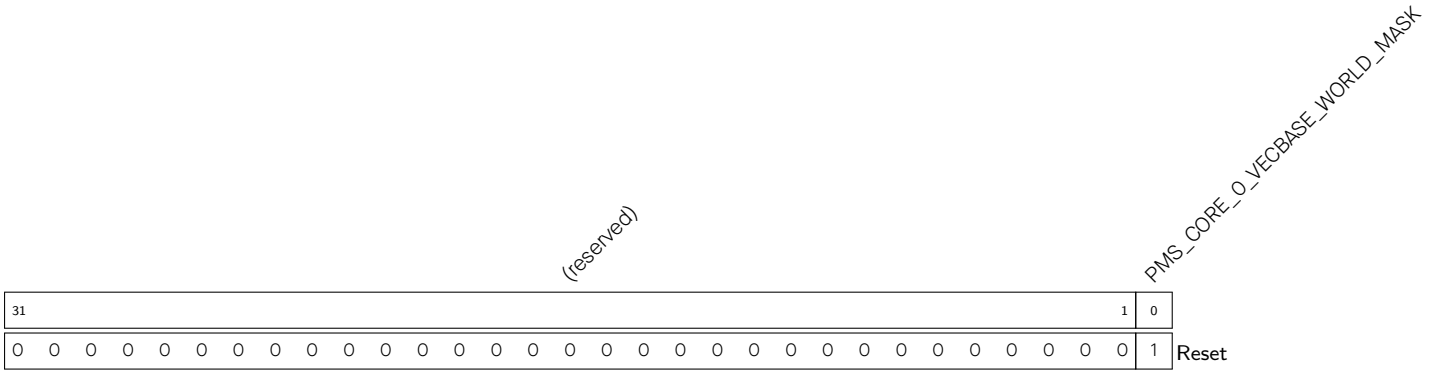
PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_EN 置 1 使能在 CPU0 的 PIF 总线使用非法数据类型访问 RTC 内存或外设时触发中断。(R/W)

Register 15.64. PMS_CORE_0_VECBASE_OVERRIDE_LOCK_REG (0x01B8)



PMS_CORE_0_VECBASE_OVERRIDE_LOCK 置 1 锁住 CPU0 的 VECBASE 配置寄存器。(R/W)

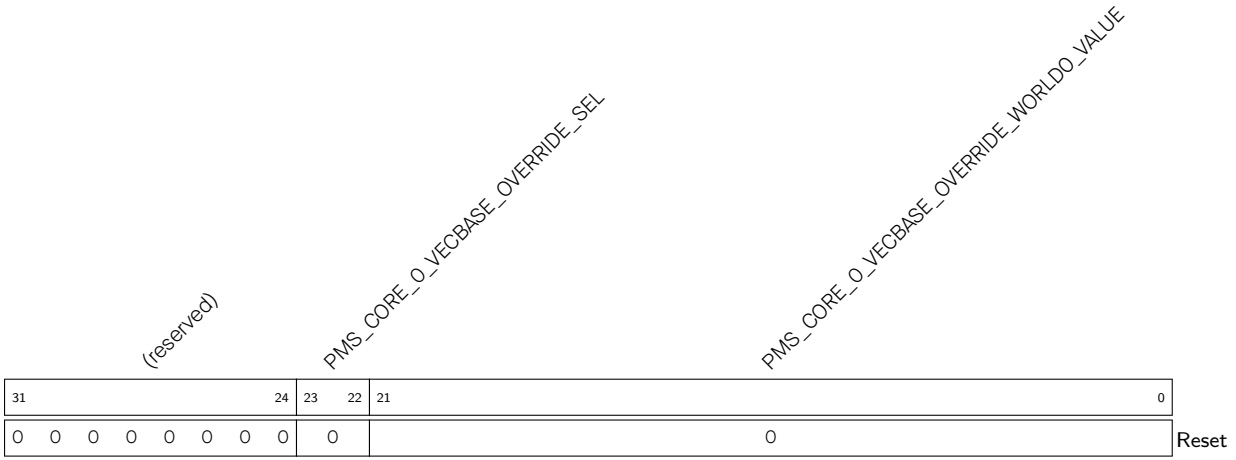
Register 15.65. PMS_CORE_0_VECBASE_OVERRIDE_0_REG (0x01BC)



PMS_CORE_0_VECBASE_WORLD_MASK 置 1 使安全世界和非安全世界均使用 WORLD0_VALUE；清 0 使安全世界使用 WORLD0_VALUE 值、非安全世界使用 WORLD1_VALUE 值。(R/W)

GoBack

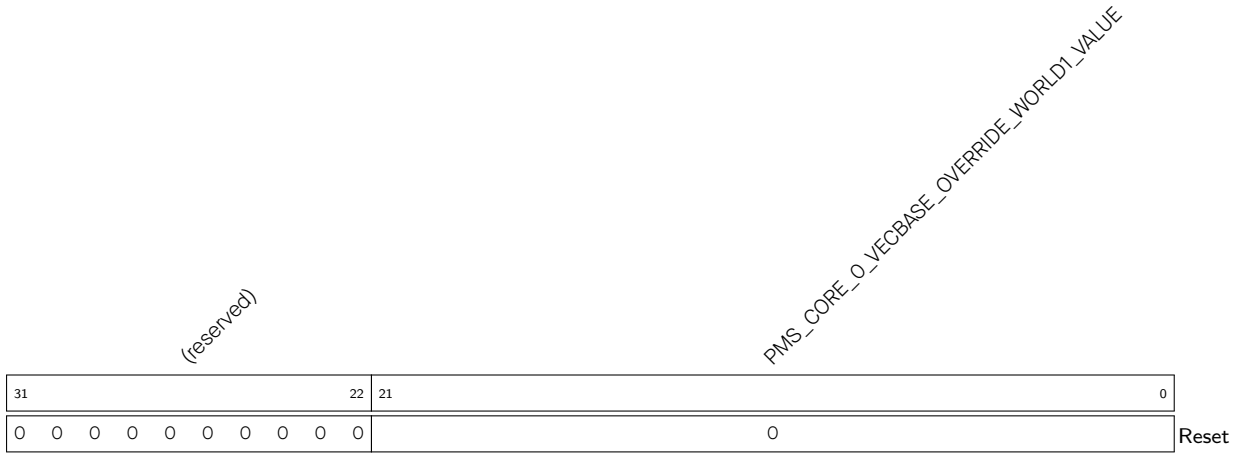
Register 15.66. PMS_CORE_0_VECBASE_OVERRIDE_1_REG (0x01C0)



PMS_CORE_0_VECBASE_OVERRIDE_WORLD0_VALUE 配置安全世界的 VECBASE 值。(R/W)

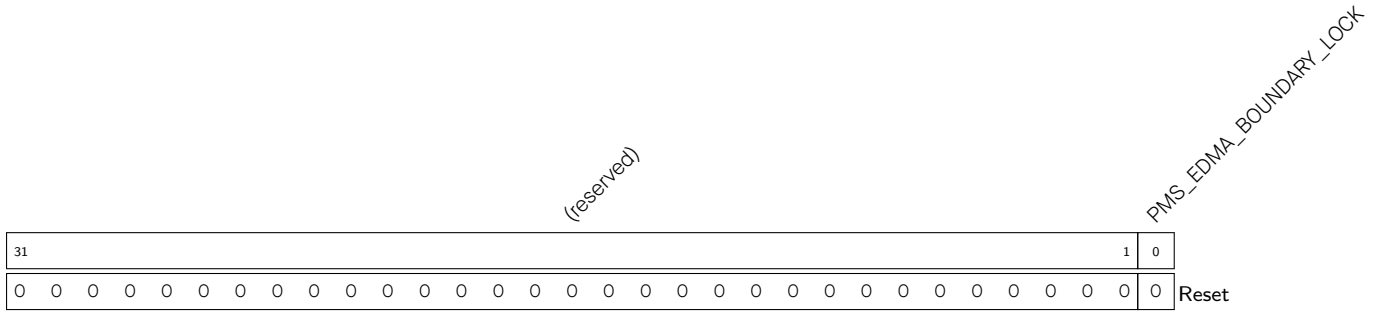
PMS_CORE_0_VECBASE_OVERRIDE_SEL 配置 VECBASE 选项。配置为 00 选择使用 VECBASE；配置为 11 选择使用 PMS_CORE_0_VECBASE_OVERRIDE_WORLD*n*_VALUE。(R/W)

Register 15.67. PMS_CORE_0_VECBASE_OVERRIDE_2_REG (0x01C4)



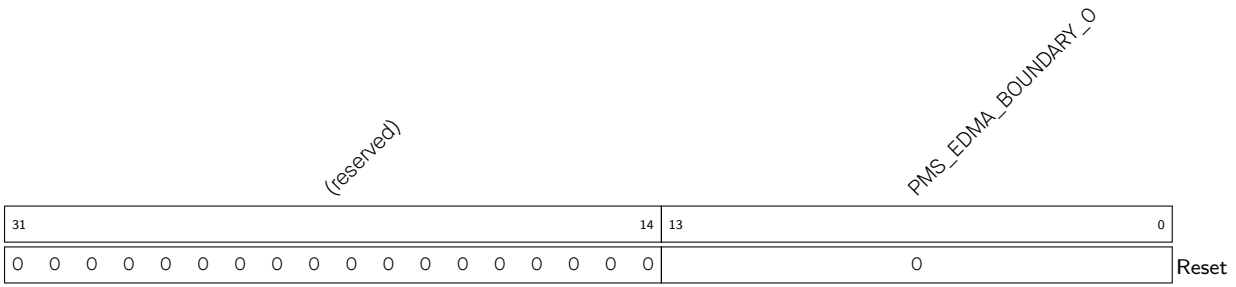
PMS_CORE_0_VECBASE_OVERRIDE_WORLD1_VALUE 配置非安全世界的 VECBASE 值。(R/W)

Register 15.68. PMS_EDMA_BOUNDARY_LOCK_REG (0x02A8)



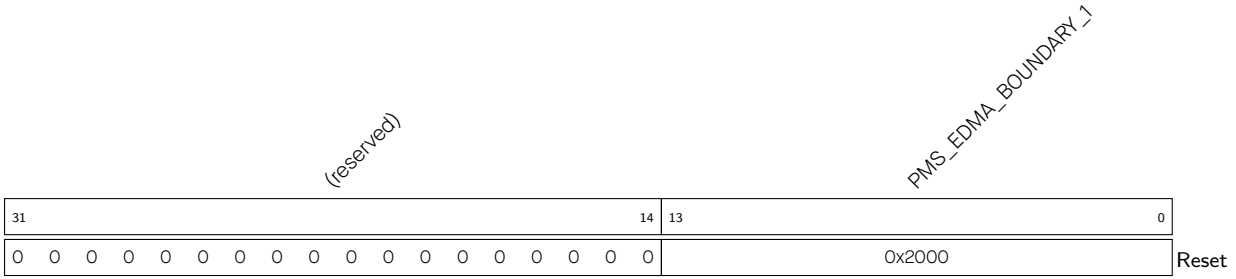
PMS_EDMA_BOUNDARY_LOCK 置 1 锁住 EDMA 边界寄存器。(R/W)

Register 15.69. PMS_EDMA_BOUNDARY_0_REG (0x02AC)



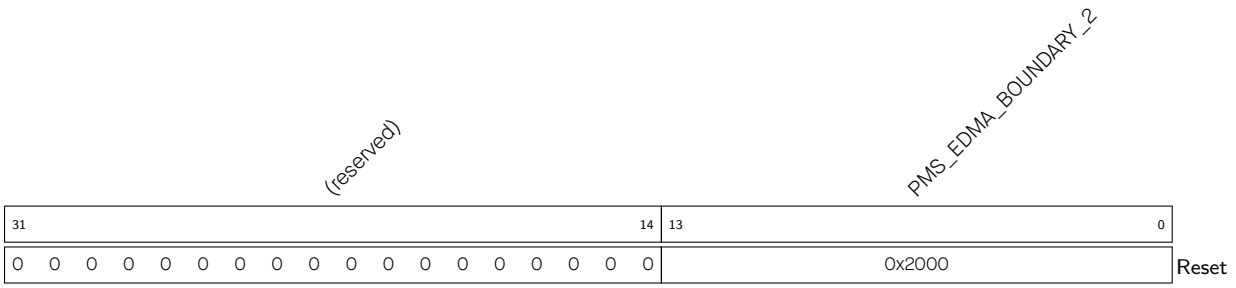
PMS_EDMA_BOUNDARY_0 配置外部 SRAM area0 的终止地址。详见表 15.5-3。(R/W)

Register 15.70. PMS_EDMA_BOUNDARY_1_REG (0x02B0)



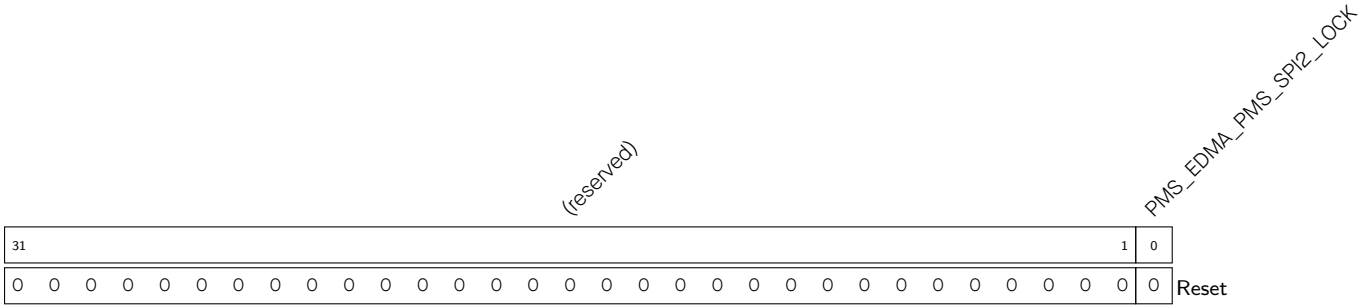
PMS_EDMA_BOUNDARY_1 配置外部 SRAM area1 的终止地址。详见表 15.5-3。(R/W)

Register 15.71. PMS_EDMA_BOUNDARY_2_REG (0x02B4)



PMS_EDMA_BOUNDARY_2 配置外部 SRAM area2 的终止地址。详见表 15.5-3。(R/W)

Register 15.72. PMS_EDMA_PMS_SPI2_LOCK_REG (0x02B8)



PMS_EDMA_PMS_SPI2_LOCK 置 1 锁住 SPI2 访问外部 SRAM 的权限配置。(R/W)

Register 15.73. PMS_EDMA_PMS_SPI2_REG (0x02BC)

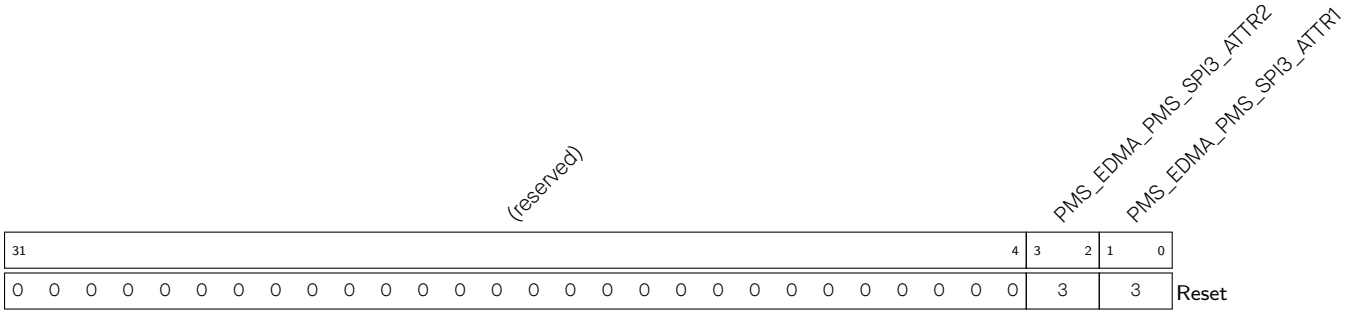
PMS_EDMA_PMS_SPI2_ATTR1 配置 SPI2 对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

PMS_EDMA_PMS_SPI2_ATTR2 配置 SPI2 对外部 SRAM Area1 的访问权限。详见表 15.5-4。(R/W)

Register 15.74. PMS_EDMA_PMS_SPI3_LOCK_REG (0x02C0)

PMS_EDMA_PMS_SPI3_LOCK 置 1 锁住 SPI3 访问外部 SRAM 的权限配置。(R/W)

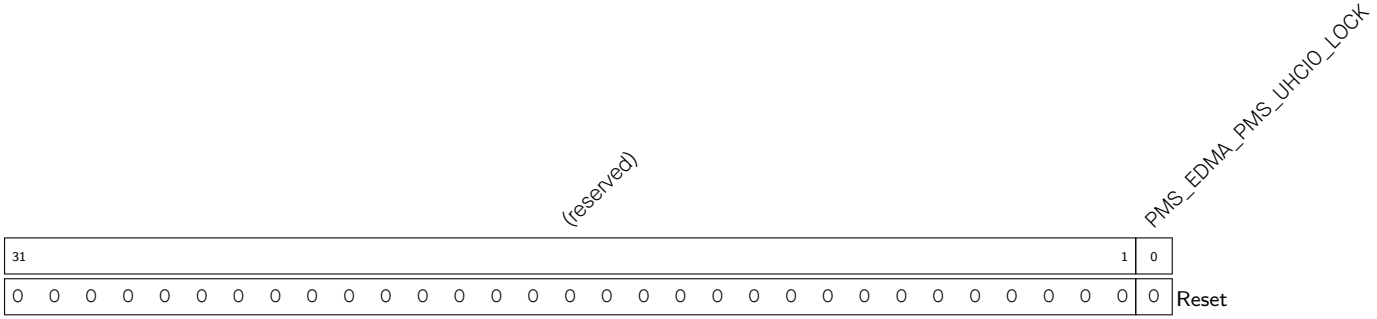
Register 15.75. PMS_EDMA_PMS_SPI3_REG (0x02C4)



PMS_EDMA_PMS_SPI3_ATTR1 配置 SPI3 对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

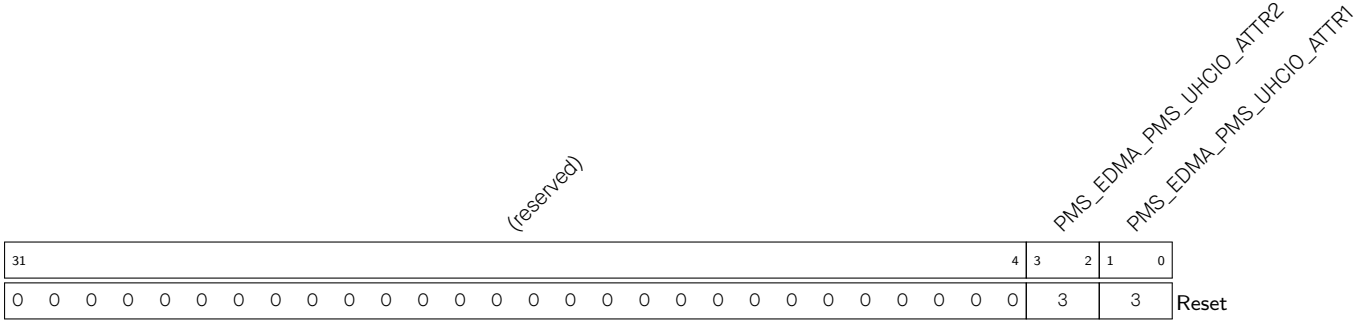
PMS_EDMA_PMS_SPI3_ATTR2 配置 SPI3 对外部 SRAM Area1 的访问权限。详见表 15.5-4。(R/W)

Register 15.76. PMS_EDMA_PMS_UHCI0_LOCK_REG (0x02C8)



PMS_EDMA_PMS_UHCI0_LOCK 置 1 锁住 UHCI0 访问外部 SRAM 的权限配置。(R/W)

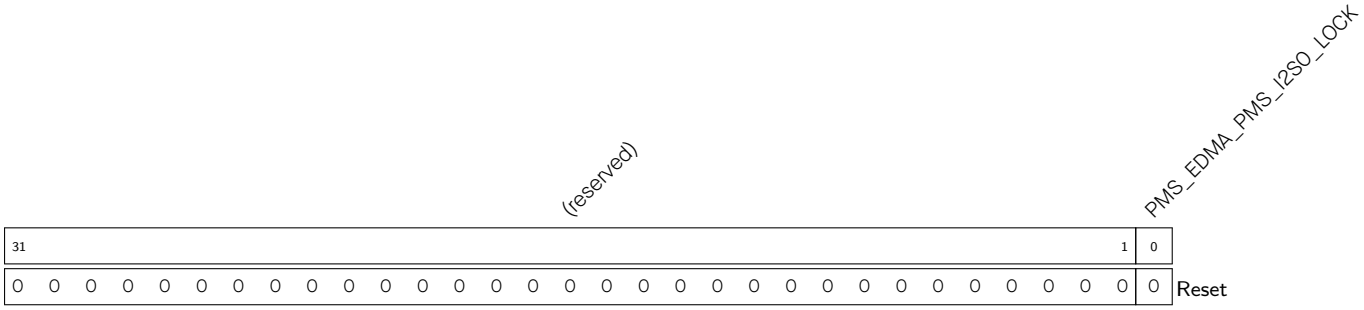
Register 15.77. PMS_EDMA_PMS_UHCI0_REG (0x02CC)



PMS_EDMA_PMS_UHCI0_ATTR1 配置 UHCI0 对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

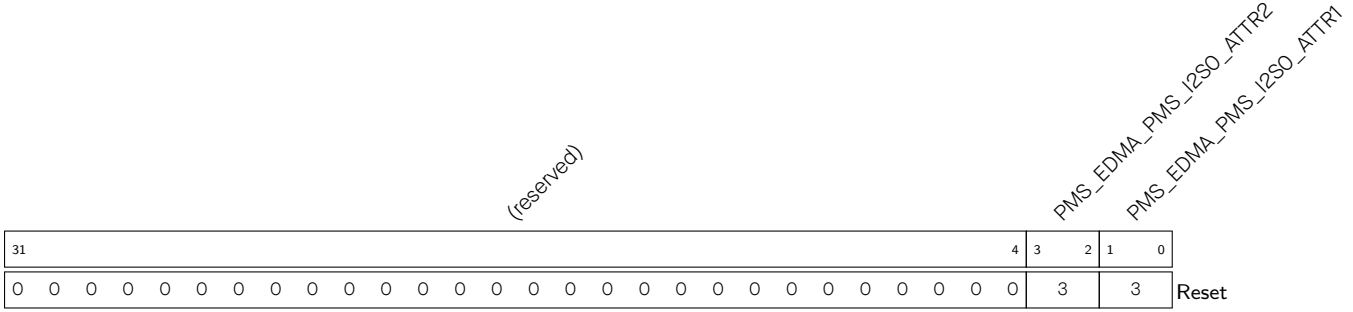
PMS_EDMA_PMS_UHCI0_ATTR2 配置 UHCI0 对外部 SRAM Area1 的访问权限。详见表 15.5-4。(R/W)

Register 15.78. PMS_EDMA_PMS_I2S0_LOCK_REG (0x02D0)



PMS_EDMA_PMS_I2S0_LOCK 置 1 锁住 I2S0 访问外部 SRAM 的权限配置。(R/W)

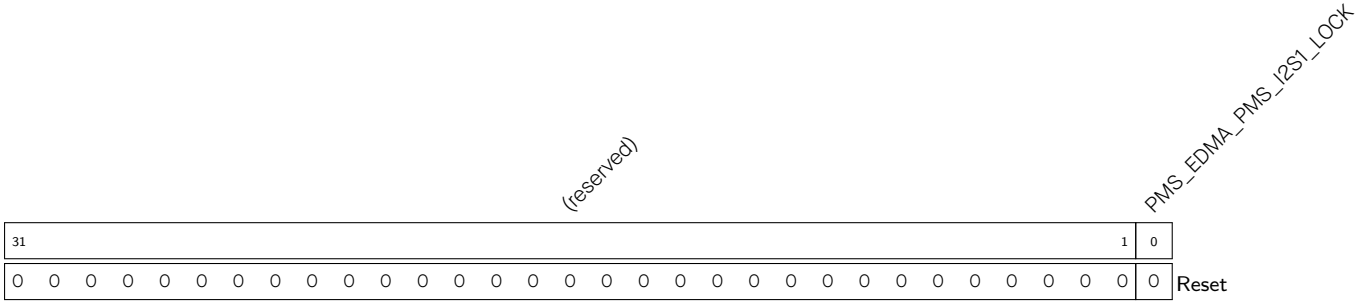
Register 15.79. PMS_EDMA_PMS_I2S0_REG (0x02D4)



PMS_EDMA_PMS_I2S0_ATTR1 配置 I2S0 对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

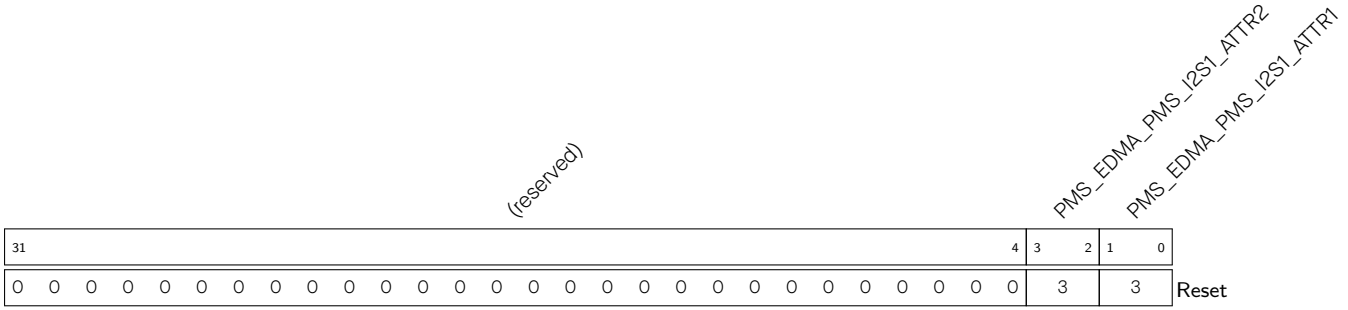
PMS_EDMA_PMS_I2S0_ATTR2 配置 I2S0 对外部 SRAM Area1 的访问权限。详见表 15.5-4。(R/W)

Register 15.80. PMS_EDMA_PMS_I2S1_LOCK_REG (0x02D8)



PMS_EDMA_PMS_I2S1_LOCK 置 1 锁住 I2S1 访问外部 SRAM 的权限配置。(R/W)

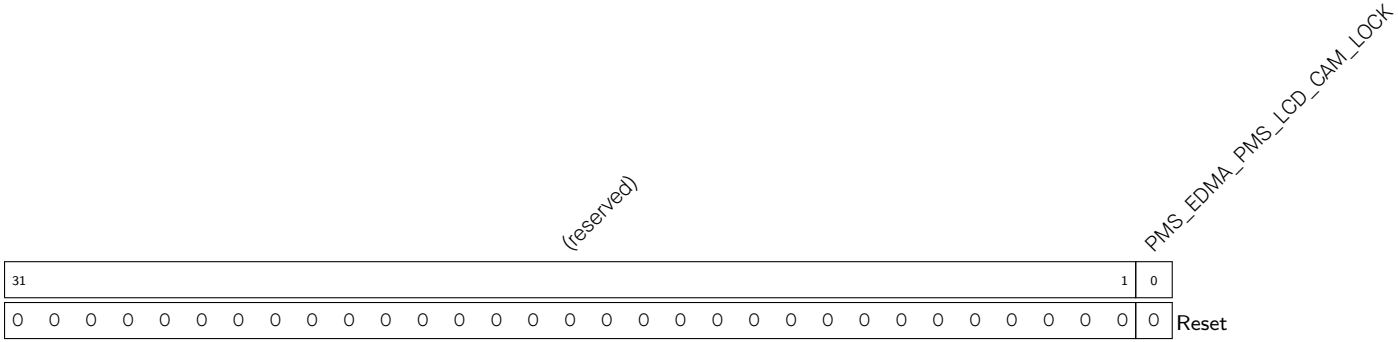
Register 15.81. PMS_EDMA_PMS_I2S1_REG (0x02DC)



PMS_EDMA_PMS_I2S1_ATTR1 配置 I2S1 对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

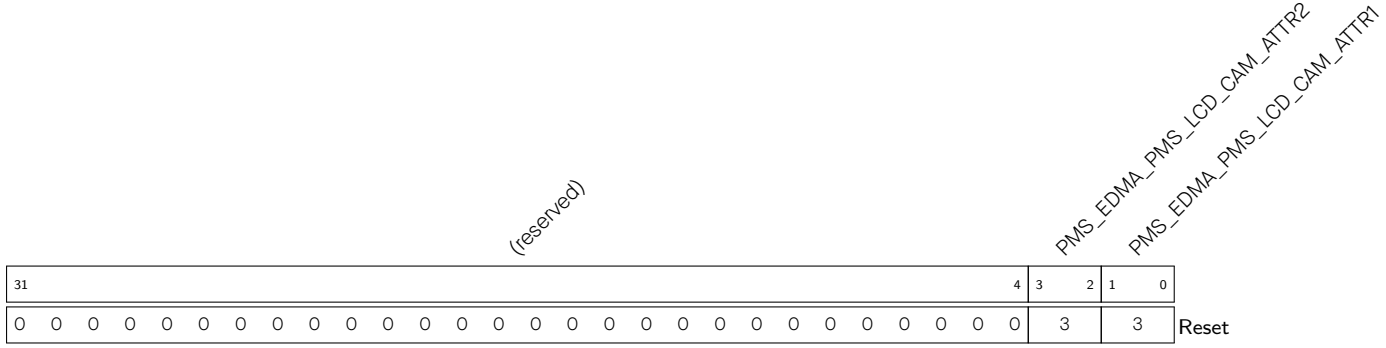
PMS_EDMA_PMS_I2S1_ATTR2 配置 I2S1 对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

Register 15.82. PMS_EDMA_PMS_LCD_CAM_LOCK_REG (0x02E0)



PMS_EDMA_PMS_LCD_CAM_LOCK 置 1 锁住 Camera-LCD 控制器访问外部 SRAM 的权限配置。(R/W)

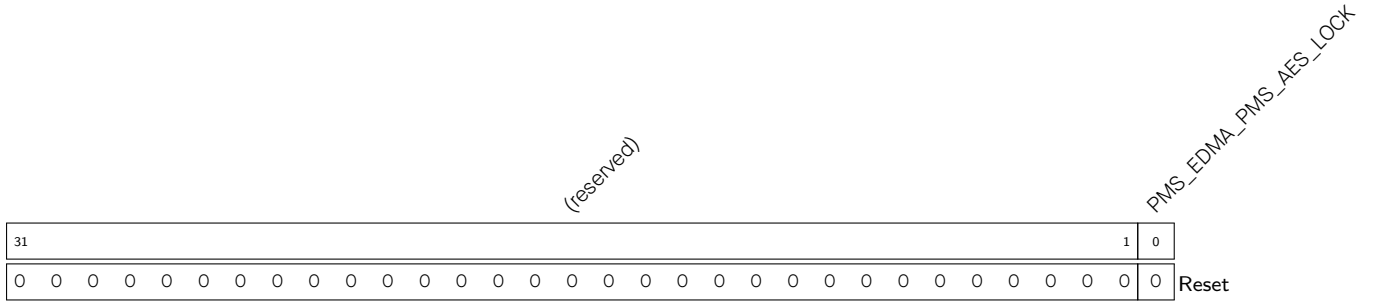
Register 15.83. PMS_EDMA_PMS_LCD_CAM_REG (0x02E4)



PMS_EDMA_PMS_LCD_CAM_ATTR1 配置 Camera-LCD 控制器对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

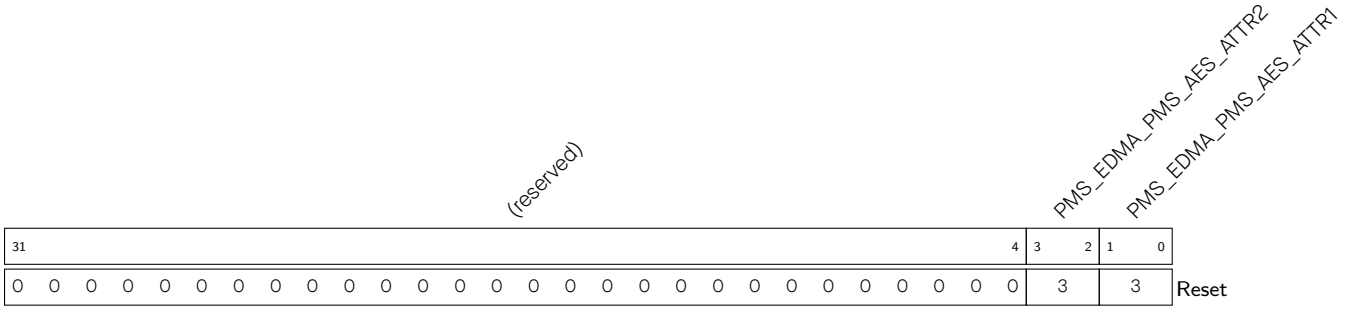
PMS_EDMA_PMS_LCD_CAM_ATTR2 配置 Camera-LCD 控制器对外部 SRAM Area1 的访问权限。详见表 15.5-4。(R/W)

Register 15.84. PMS_EDMA_PMS_AES_LOCK_REG (0x02E8)



PMS_EDMA_PMS_AES_LOCK 置 1 锁住 AES 访问外部 SRAM 的权限配置。(R/W)

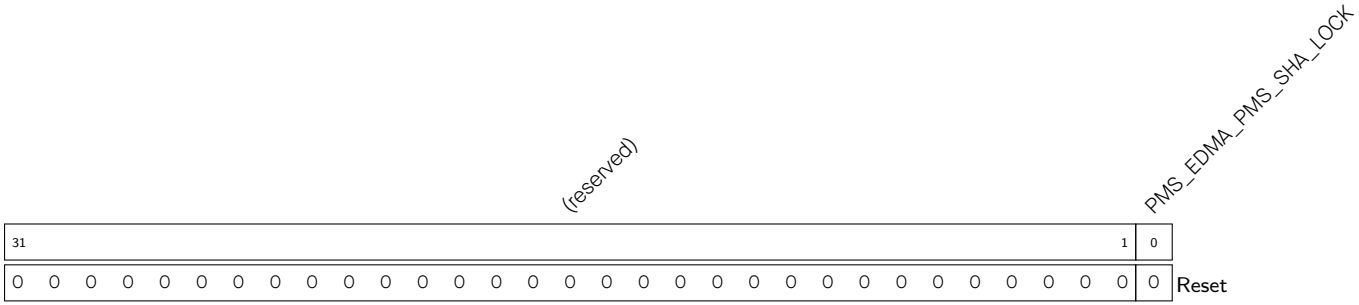
Register 15.85. PMS_EDMA_PMS_AES_REG (0x02EC)



PMS_EDMA_PMS_AES_ATTR1 配置 AES 对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

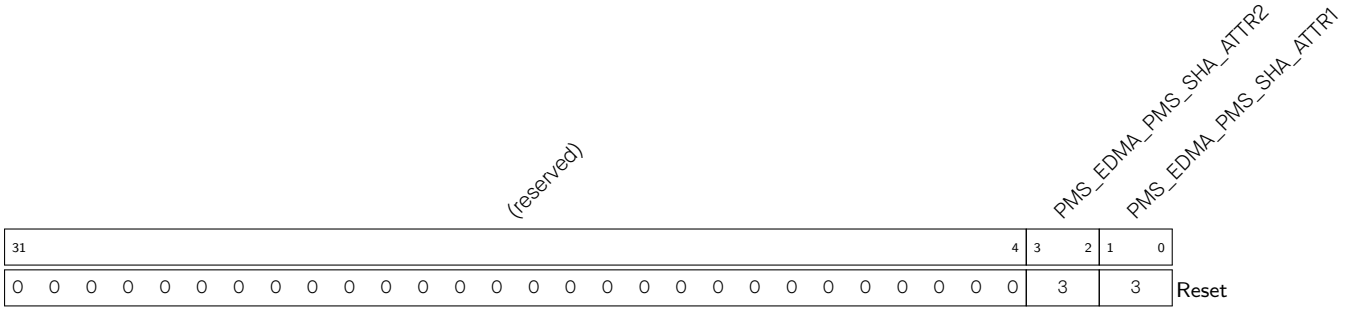
PMS_EDMA_PMS_AES_ATTR2 配置 AES 对外部 SRAM Area1 的访问权限。详见表 15.5-4。(R/W)

Register 15.86. PMS_EDMA_PMS_SHA_LOCK_REG (0x02F0)



PMS_EDMA_PMS_SHA_LOCK 置 1 锁住 SHA 访问外部 SRAM 的权限配置。(R/W)

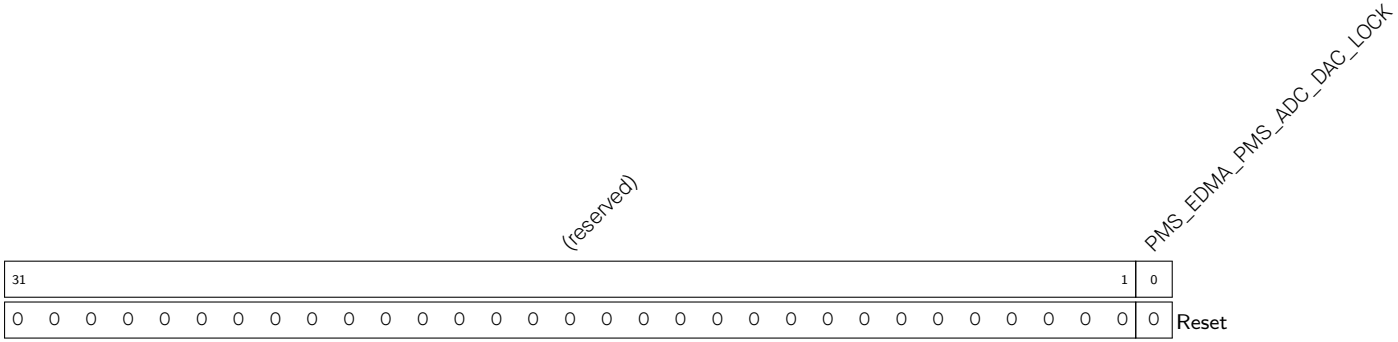
Register 15.87. PMS_EDMA_PMS_SHA_REG (0x02F4)



PMS_EDMA_PMS_SHA_ATTR1 配置 SHA 对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

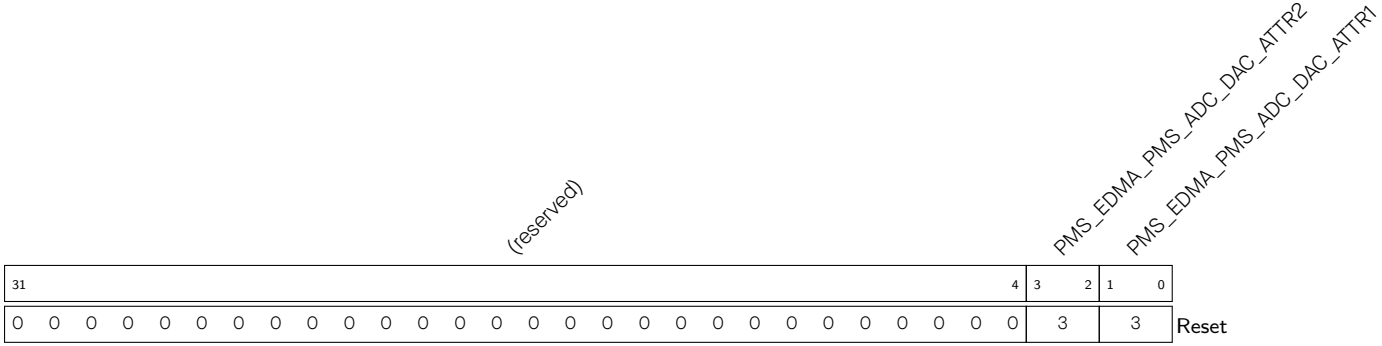
PMS_EDMA_PMS_SHA_ATTR2 配置 SHA 访问外部 SRAM 的权限配置 Area1. For details, see Table 15.5-4. (R/W)

Register 15.88. PMS_EDMA_PMS_ADC_DAC_LOCK_REG (0x02F8)



PMS_EDMA_PMS_ADC_DAC_LOCK 置 1 锁住 ADC Controller 访问外部 SRAM 的权限配置。(R/W)

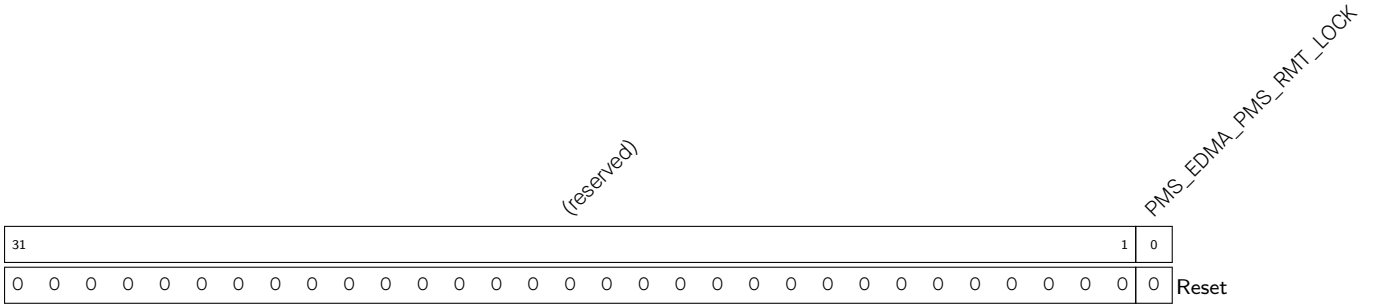
Register 15.89. PMS_EDMA_PMS_ADC_DAC_REG (0x02FC)



PMS_EDMA_PMS_ADC_DAC_ATTR1 配置 ADC Controller 对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

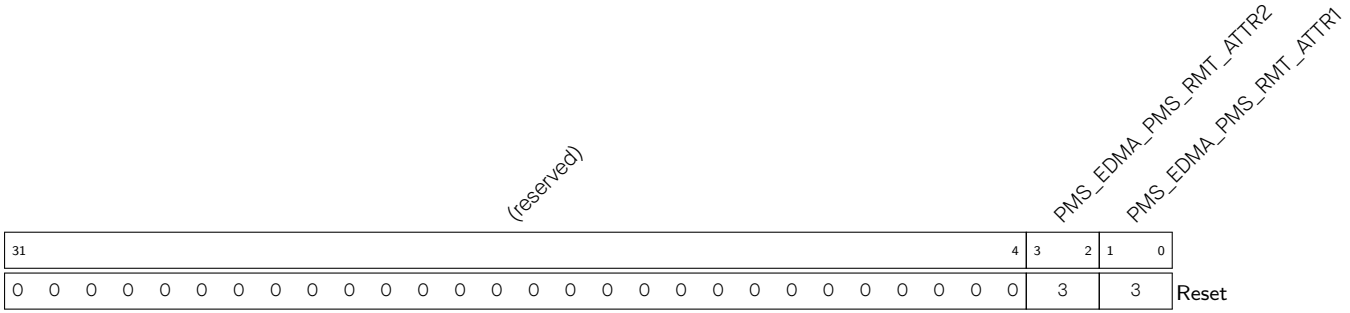
PMS_EDMA_PMS_ADC_DAC_ATTR2 配置 ADC Controller 对外部 SRAM Area1 的访问权限。详见表 15.5-4。(R/W)

Register 15.90. PMS_EDMA_PMS_RMT_LOCK_REG (0x0300)



PMS_EDMA_PMS_RMT_LOCK 置 1 锁住 Remote Control Peripheral 访问外部 SRAM 的权限配置。(R/W)

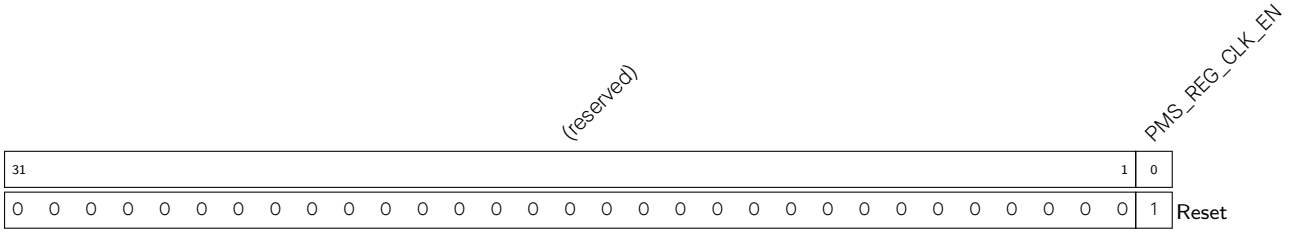
Register 15.91. PMS_EDMA_PMS_RMT_REG (0x0304)



PMS_EDMA_PMS_RMT_ATTR1 配置 Remote Control Peripheral 对外部 SRAM Area0 的访问权限。详见表 15.5-4。(R/W)

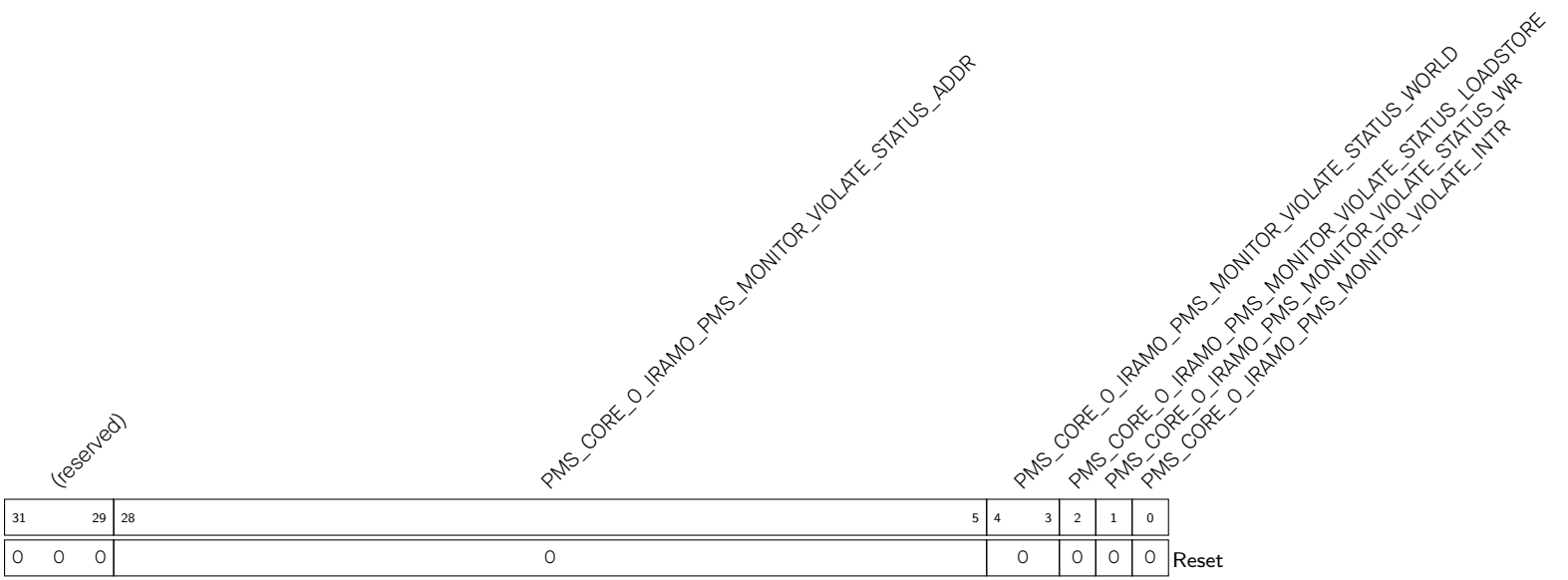
PMS_EDMA_PMS_RMT_ATTR2 配置 Remote Control Peripheral 对外部 SRAM Area1 的访问权限。详见表 15.5-4。(R/W)

Register 15.92. PMS_CLOCK_GATE_REG_REG (0x0308)



PMS_REG_CLK_EN 置 1 使能时钟门控功能。(R/W)

Register 15.93. PMS_CORE_0_IRAM0_PMS_MONITOR_2_REG (0x00EC)



PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR 存储 IBUS 非法访问的中断状态。(RO)

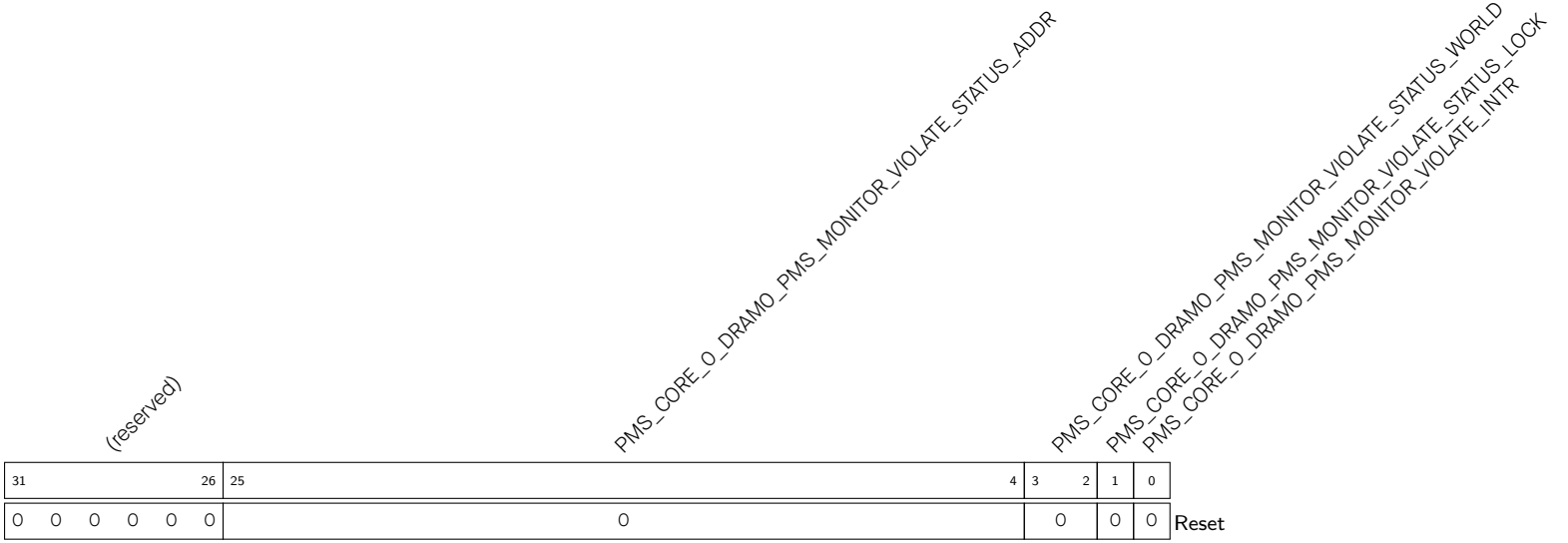
PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_WR 存储非法访问的访问方向。1：写；0：读。注意，本字段仅在 PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_LOADSTORE 为 1 时生效。(RO)

PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_LOADSTORE 存储非法访问的指令访问方向。1：加载/存储；0：取指。(RO)

PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_WORLD 存储非法访问发生时 CPU 所处的世界。0b01：安全世界；0b10：非安全世界。(RO)

PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_ADDR 存储 CPU0 的 IBUS 的非法访问地址。注意，此地址为相对于 0x40000000 的偏移地址，且单位为 4，即实际地址应为 0x40000000 + PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_ADDR * 4。(RO)

Register 15.94. PMS_CORE_0_DRAM0_PMS_MONITOR_2_REG (0x010C)



PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR 存储 DBUS 非法访问的中断状态。(RO)

PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_LOCK 原子访问标志。1：原子访问；0：非原子访问。(RO)

PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_WORLD 存储非法访问发生时 CPU 所处的世界。0b01：安全世界；0b10：非安全世界。(RO)

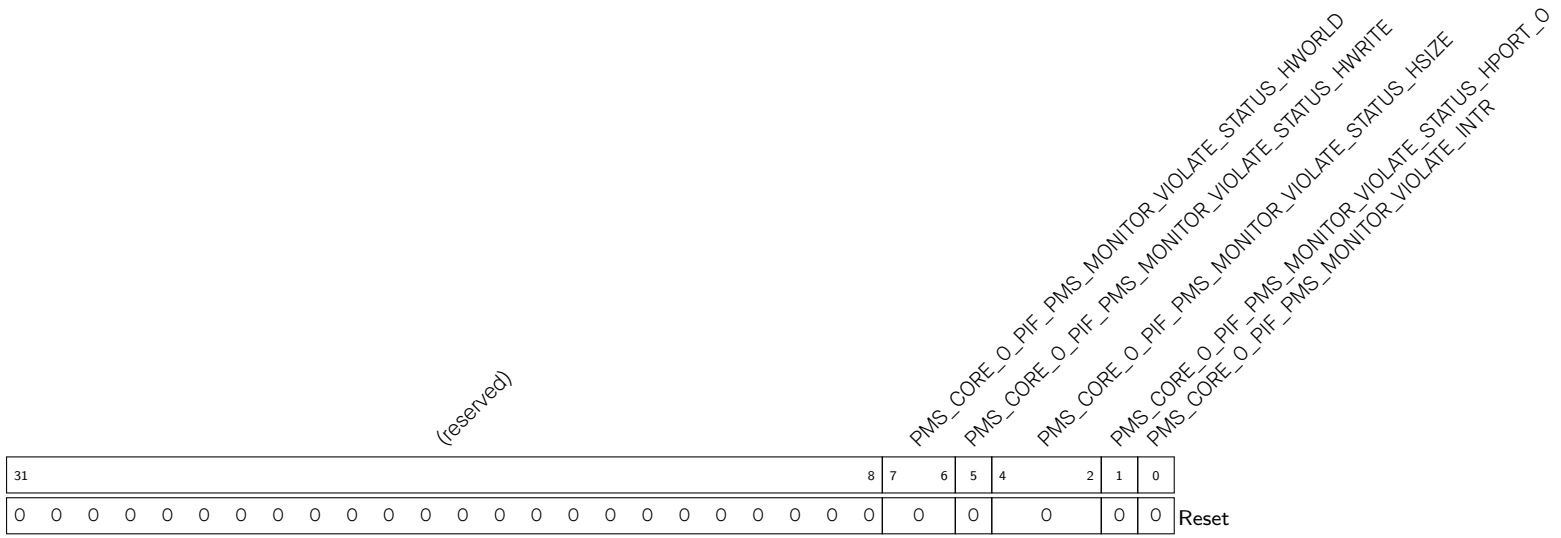
PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_ADDR 存储 CPU0 的 dBUS 的非法访问地址。注意，此地址为相对于 0x3c000000 的偏移地址，且单位为 16，即实际地址应为 0x3c000000 + PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_ADDR * 16。(RO)

Register 15.95. PMS_CORE_0_DRAM0_PMS_MONITOR_3_REG (0x0110)

PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_WR 存储非法访问方向。1：写；0：读。(RO)

PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_BYTEEN 存储非法访问的字节信息。(RO)

Register 15.96. PMS_CORE_0_PIF_PMS_MONITOR_2_REG (0x01A4)



PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR 存储 PIF 非法访问的中断状态。(RO)

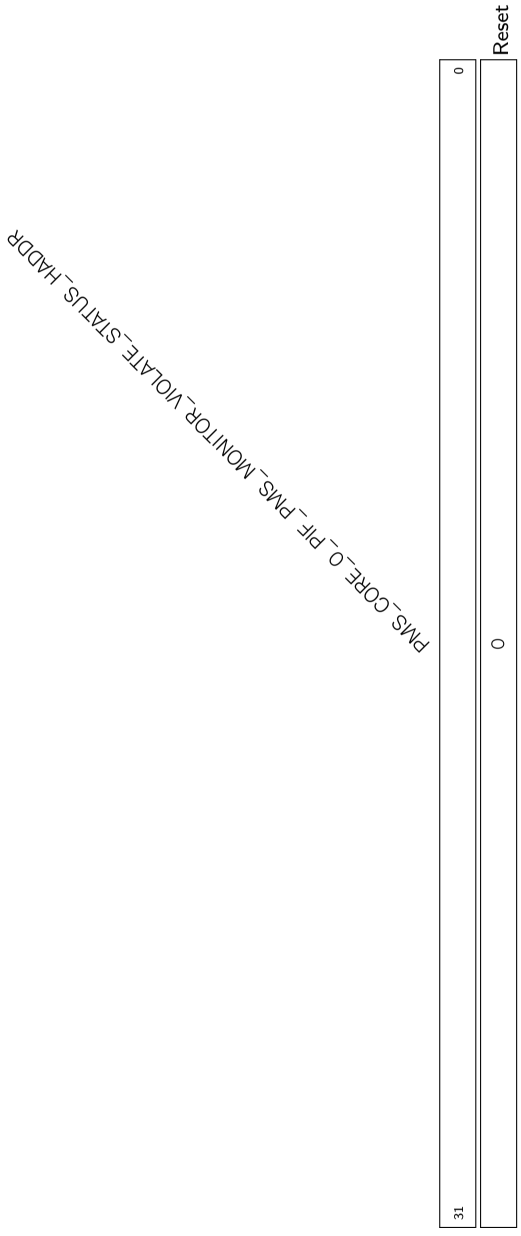
PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HPORT_0 存储非法访问类型。0：指令；1：数据。(RO)

PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HSIZE 存储非法访问的数据类型。0：字节；1：半字；2：字。(RO)

PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HWRITE 存储非法访问的方向。0：读；1：写。(RO)

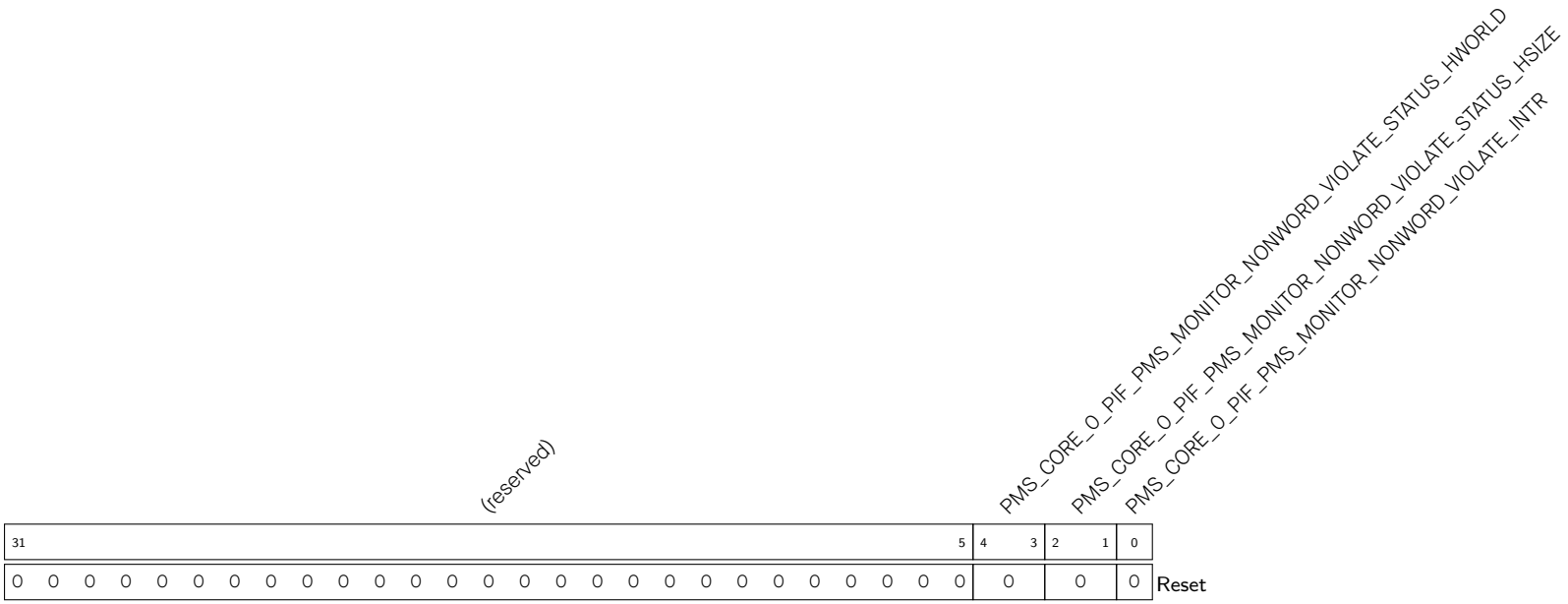
PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HWORLD 存储非法访问发生时 CPU 所处的世界。0b01：安全世界；0b10：非安全世界。(RO)

Register 15.97. PMS_CORE_0_PIF_PMS_MONITOR_3_REG (0x01A8)



PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HADDR 存储 CPU0 的 PIF 的非法访问地址。(RO)

Register 15.98. PMS_CORE_0_PIF_PMS_MONITOR_5_REG (0x01B0)

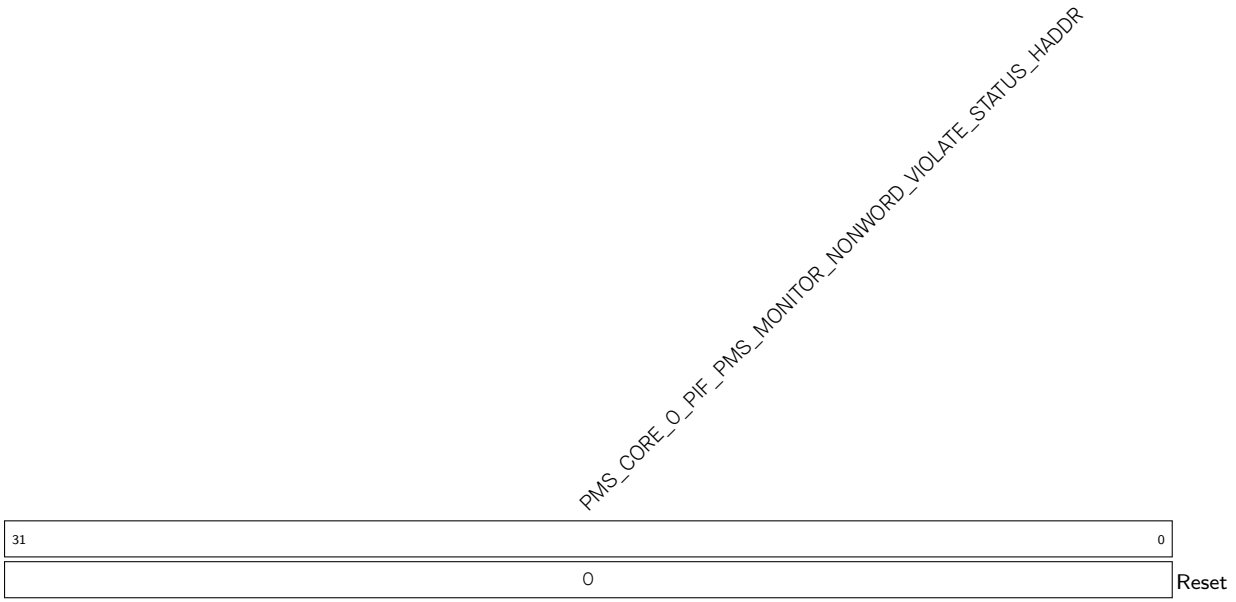


PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_INTR 存储 PIF 非法数据类型的中断状态。(RO)

PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_STATUS_HSIZE 存储非法访问的数据类型。(RO)

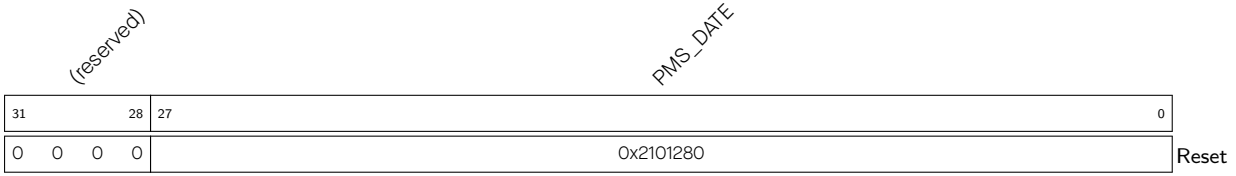
PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_STATUS_HWORLD 存储非法访问发生时 CPU 所处的世界。0b01：安全世界；0b10：非安全世界。(RO)

Register 15.99. PMS_CORE_0_PIF_PMS_MONITOR_6_REG (0x01B4)



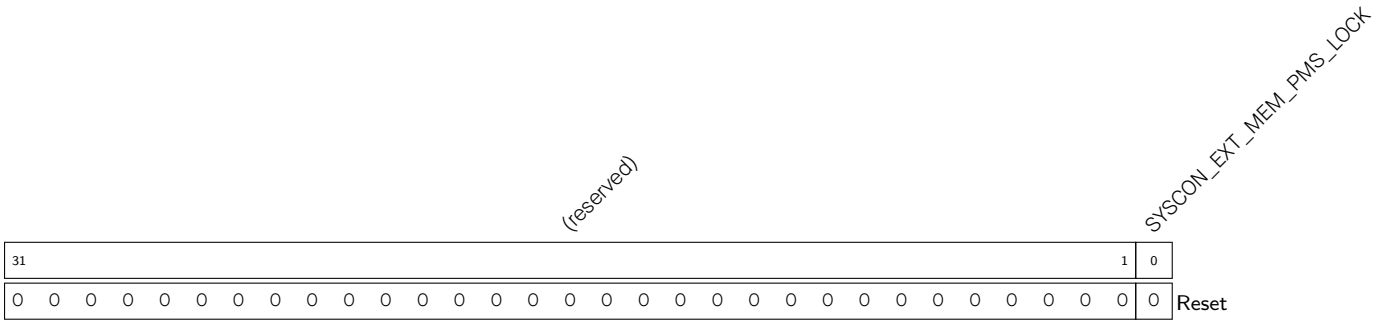
PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_STATUS_HADDR 存储 CPU0 的 PIF 的非法数据类型访问的地址。(RO)

Register 15.100. PMS_DATE_REG (0x0FFC)



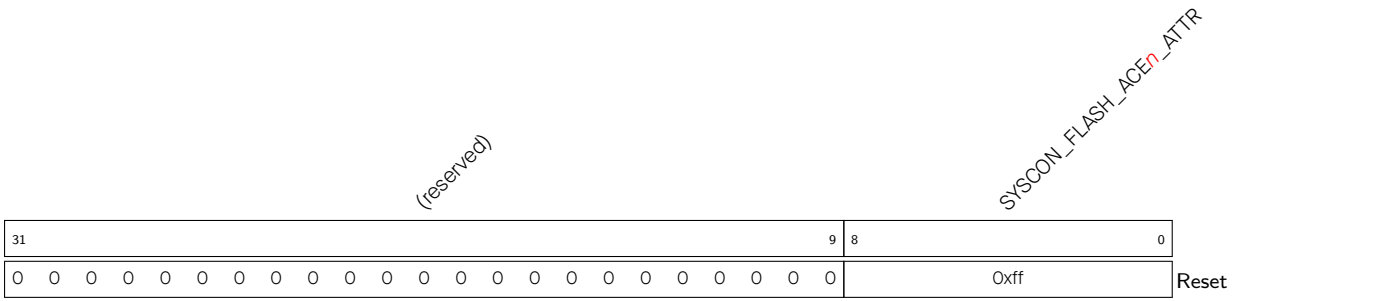
PMS_DATE 日期及版本寄存器。(R/W)

Register 15.101. SYSCON_EXT_MEM_PMS_LOCK_REG (0x0020)



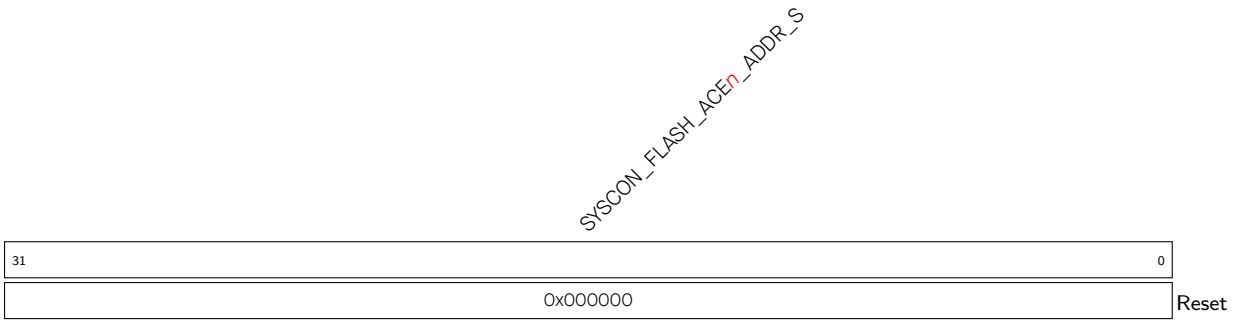
SYSCON_EXT_MEM_PMS_LOCK　如果为 1，则外部存储器的相关配置被锁定，无法修改。(R/W)

Register 15.102. SYSCON_FLASH_ACE*n*_ATTR_REG (*n*: 0 - 3) (0x0028 + 4**n*)



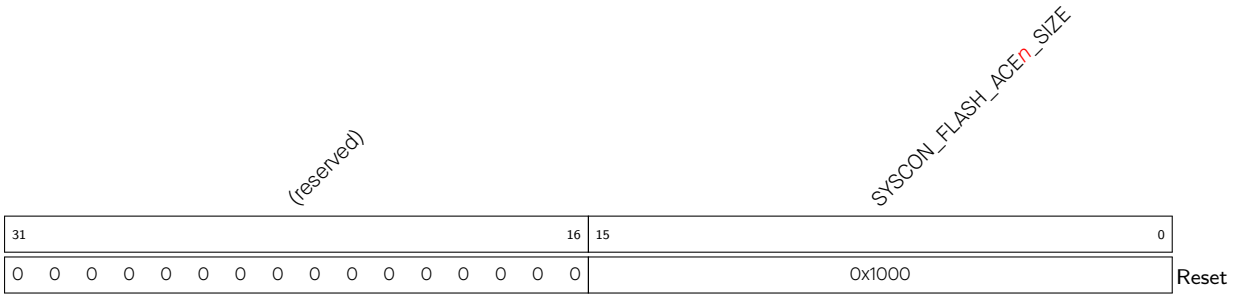
SYSCON_FLASH_ACE*n*_ATTR　配置 flash 区域*n* 的权限。(R/W)

Register 15.103. SYSCON_FLASH_ACE*n*_ADDR_REG (*n*: 0-3) (0x0038 + 4**n*)



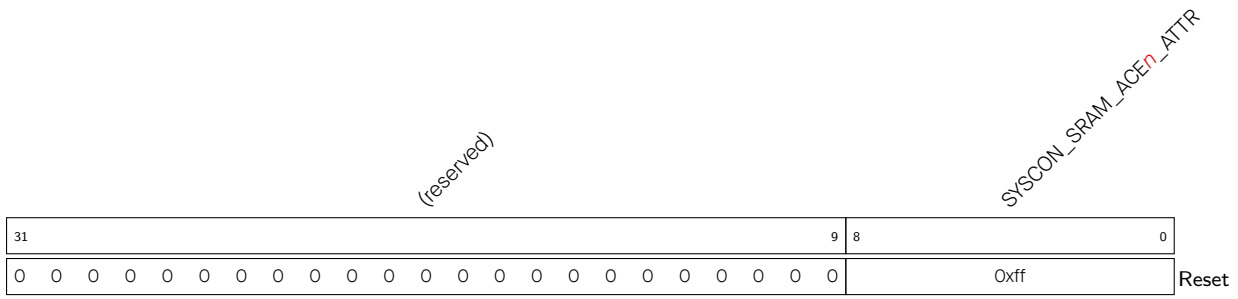
SYSCON_FLASH_ACE0_ADDR_S 配置 flash 区域*n* 的起始地址，要求以 64 KB 对齐。(R/W)

Register 15.104. SYSCON_FLASH_ACE*n*_SIZE_REG (*n*: 0-3) (0x0048 + 4**n*)



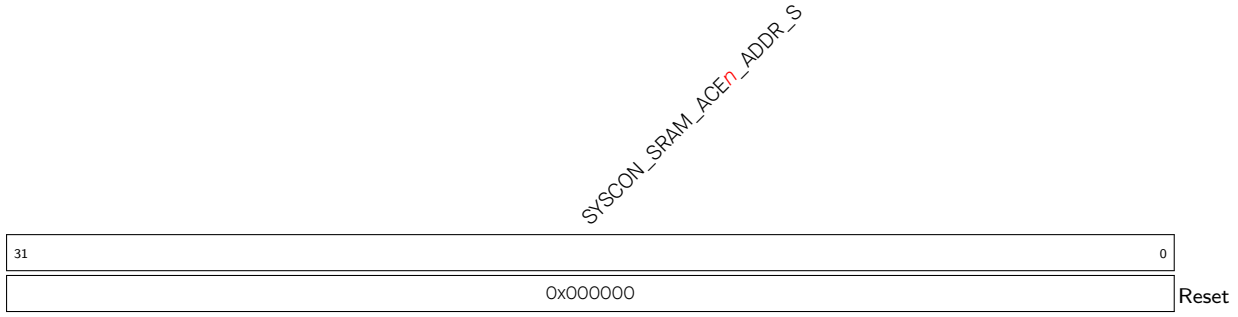
SYSCON_FLASH_ACE*n*_SIZE 配置 flash 区域*n* 的长度，单位为 64 KB。(R/W)

Register 15.105. SYSCON_SRAM_ACE*n*_ATTR_REG (0x0058)



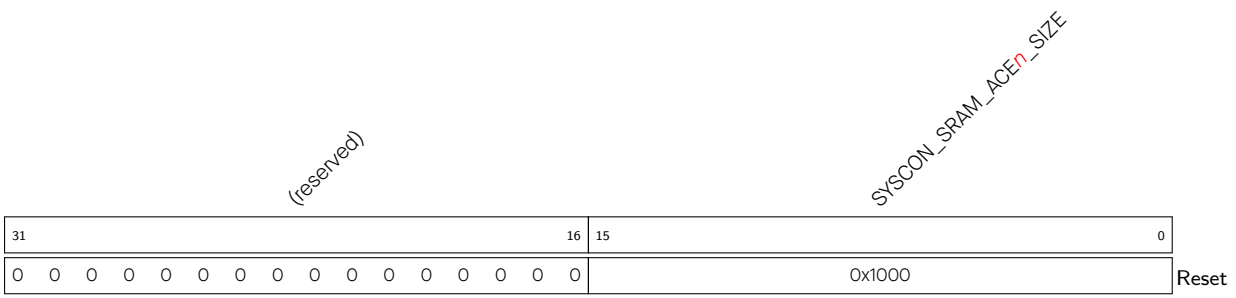
SYSCON_SRAM_ACE*n*_ATTR 配置对 SRAM Region*n* 的权限。(R/W)

Register 15.106. SYSCON_SRAM_ACE*n*_ADDR_REG (*n*: 0-3) (0x0068 + 4**n*)



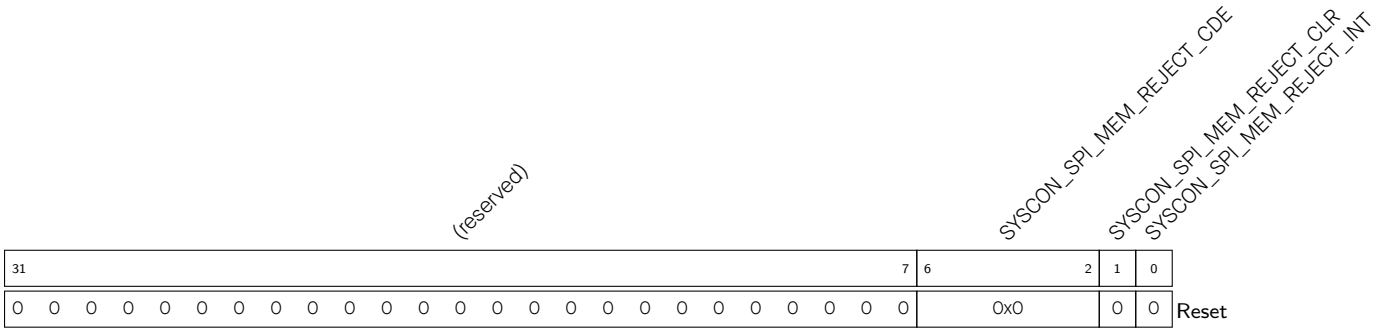
SYSCON_SRAM_ACE*n*_ADDR_S 配置 SRAM 区域 *n* 的起始地址，要求以 64 KB 对齐。(R/W)

Register 15.107. SYSCON_SRAM_ACE*n*_SIZE_REG (*n*: 0-3) (0x0078 + 4**n*)



SYSCON_SRAM_ACE*n*_SIZE 配置 SRAM 区域 *n* 的长度，单位为 64 KB。(R/W)

Register 15.108. SYSCON_SPI_MEM_PMS_CTRL_REG (0x0088)



SYSCON_SPI_MEM_REJECT_INT 当发生外部存储器访问异常时，该寄存器会置起，同时会触发中断。(RO)

SYSCON_SPI_MEM_REJECT_CLR 当发生外部存储器访问异常后，往该位写 1 清除异常记录。(WOD)

SYSCON_SPI_MEM_REJECT_CDE 当发生外部存储器访问异常时，该寄存器记录异常类型，独热码，从高位到低位依次表示：访问没有落入有效区域、访问落入多个有效区域（区域重叠）、写拒绝、读拒绝、取指拒绝。(RO)

Register 15.109. SYSCON_SPI_MEM_REJECT_ADDR_REG (0x008C)

SYSCON_SPI_MEM_REJECT_ADDR 当发生外部存储器访问异常时，该寄存器记录的是异常实地址。(RO)

第 16 章

World 控制器 (WCL)

16.1 概述

ESP32-S3 允许用户将芯片的硬件和软件资源分为安全世界 (World0) 和非安全世界 (World1)，从而有效防止非法数据拷贝、破坏或篡改，或通过恶意软件、硬件监视、硬件干预等方式破坏或获取设备信息。两个世界之间可以由 World 控制器进行切换。

默认情况，ESP32-S3 不划分安全世界和非安全世界，所有资源全部共享。如有需求，用户可通过权限配置将资源分为安全世界和非安全世界，具体配置请见章节 15 权限控制 *(PMS)*，本章节仅介绍 World 控制器及 CPU 在两个世界之间的切换。

16.2 主要特性

ESP32-S3 的 World 控制器主要用于：

- 控制 CPU 在安全世界与非安全世界中的相互切换
- 记录 CPU 的世界切换信息
- 屏蔽 CPU 的 NMI 中断
- 两个 CPU（CORE_*m*：CPU0 和 CPU1）可独立切换

16.3 功能描述

在 World 控制器的帮助下，ESP32-S3 的安全世界和非安全世界可以访问的资源可以不同：

- 安全世界 (World0)：
  - 可以访问所有的外设和存储空间。
  - 主要执行所有需要保密的操作，如指纹识别、密码处理、数据加解密、安全认证等。
- 非安全世界 (World1)：
  - 只能访问部分的外设和存储空间。
  - 执行其他操作，如用户操作系统、各种应用程序等。

ESP32-S3 的 CPU 和从设备均有相应的安全世界和非安全世界权限：

- CPU 可以在两个世界中运行：
  - 在安全世界：运行安全世界的程序
  - 在非安全世界：运行非安全世界的程序

  - CPU 上电时均默认处于安全世界，此后可配置通过 World 控制器在两个世界间进行切换。

- 所有的从设备（外设*、存储器等）均可分别配置：
  - 安全世界权限：表示可以从安全世界访问，即允许 CPU 从安全世界中访问该从设备；
  - 非安全世界权限：表示可以从非安全世界访问，即允许 CPU 从非安全世界中访问该从设备。
  - 注意，从设备可配置为同时具有安全世界和非安全世界权限。

  具体请见章节 15 权限控制 *(PMS)*。

说明：
World 控制器本身也属于一种外设。因此，与所有从设备一样，World 控制器理论上也可以配备安全世界和非安全世界的访问权限，即允许从安全世界和非安全世界访问 World 控制器。不过，为了保证世界切换的安全性，应**禁止** World 控制器的非安全世界访问权限，确保 CPU 无法从非安全世界中更改 World 控制器的配置。

CPU 在访问从设备时：

1. 首先，CPU 将告知从设备自己所处的世界；
2. 接着，从设备会根据自身的权限配置，判断是否允许 CPU 的访问。
   - 如果允许访问，则访问继续；
   - 如果不允许访问，则不响应此次访问并且触发中断。

这样一来，可以保证安全世界的资源不会被非安全世界非法访问。

16.4 CPU 的世界切换

CPU 可以从安全世界切换至非安全世界，也可以从非安全世界切换至安全世界。

16.4.1 安全世界切换到非安全世界

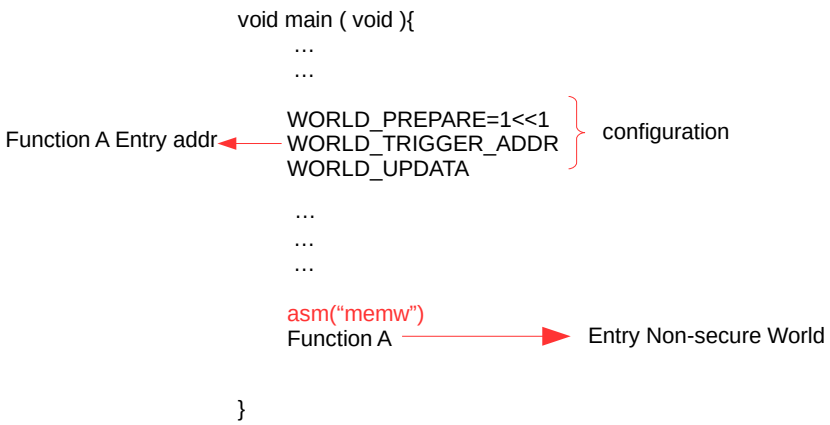


图 16.4-1. 安全世界切换到非安全世界

当 ESP32-S3 的 CPU 准备从安全世界执行非安全世界程序时，只需完成如下步骤：

1. 配置 World 控制器，详见下方描述。
2. 冲刷 write_buffer 中残留的数据，详见第 16.4.3 小节。

完成以上操作后，即可调用非安全世界的程序。

但值得注意的是，由于存在预取指和流水线等方面原因，如果在 World 控制器配置完成后立即调用非安全世界程序，则可能存在配置程序尚未运行完成，但 CPU 已经执行了非安全世界应用程序的情况。这样一来，World 控制器将一直无法检测到 CPU 读取入口地址的行为，从而无法切换至非安全世界，导致非安全世界的程序在安全世界中运行，带来安全风险。

因此，一定要确保配置程序运行完成后，再调用非安全世界应用程序。具体来说，可以**将非安全世界应用程序申明为 noinline 属性**。

配置 World 控制器

World 控制器的具体配置过程如下：

1. 配置 WCL_CORE_*m*_WORLD_PERPARE_REG 寄存器为 0x2，表示要切换至进入非安全世界。
2. 配置 WCL_CORE_*m*_World_TRIGGER_ADDR_REG 寄存器为非安全世界的入口地址，即需要执行的非安全世界应用程序的入口地址。
3. 配置 WCL_CORE_*m*_World_UPDATE_REG 寄存器（可配置任意值），代表已完成配置。

说明：

- 注意 WCL_CORE*m*_WORLD_PERPARE_REG 和 WCL_CORE_*m*_World_TRIGGER_ADDR_REG 的配置前后顺序不作要求，WCL_CORE_*m*_World_UPDATE_REG 寄存器必须最后配置。
- 配置完成后，需要使用 memw (memory wait) 汇编指令清除 write_buffer，详情请见第 16.4.3。

此后，World 控制器会持续检测 CPU 是否执行到配置的应用程序入口：一旦检测到 CPU 开始从配置的入口地址取指，则会将 CPU 切换至非安全世界，并开始从非安全世界中执行应用程序。

需要注意的是，World 控制器经过配置后：

- 将持续进行检测，直至检测到 CPU 开始从配置的入口地址取指，则从安全世界切换至非安全世界。
  - 如需取消配置，可通过配置 WCL_CORE_*m*_World_Cancel_REG 寄存器取消，此时即使 CPU 执行到了配置的入口地址处，也不会触发世界切换。注意配置完成后，同样需要使用 memw (memory wait) 汇编指令清除 write_buffer，详情请见第 16.4.3。
- 每次生效后将自动停止检测。因此，每次从安全世界切换到非安全世界都需要重新配置。

16.4.2 非安全世界切换到安全世界

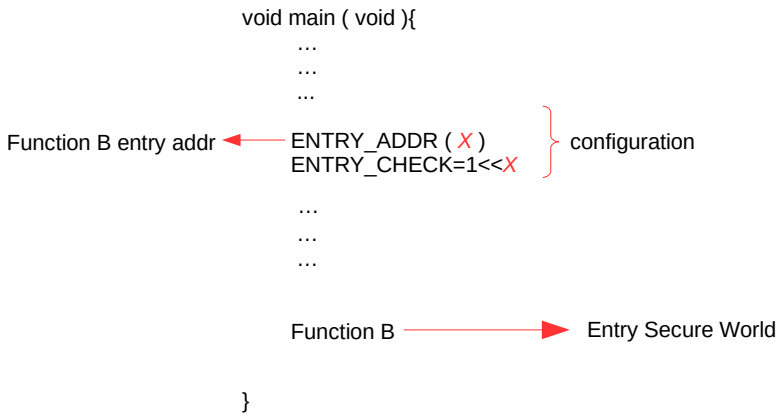


图 16.4-2. 非安全世界切换到安全世界

当 CPU 结束非安全世界应用程序，准备切换执行安全世界程序时：需要通过触发**中断**和**异常**的方式。经过配置后，当 World 控制器检测到触发的中断或异常后，会自动切换到安全世界中。

具体流程如下：

1. 配置 WCL_CORE_*m*_MESSAGE_ADDR_REG 和 WCL_CORE_*m*_MESSAGE_MAX_REG 寄存器清除 write_buffer，详见 16.4.3。
2. 配置 WCL_CORE_*m*_ENTRY_*n*_ADDR_REG (*n*: 1-13) 为各级中断或异常的入口地址。
   - 由于 ESP32-S3 所有的中断和异常的入口地址都是基于 VecBase 特殊寄存器（基地址）+ 固定偏移，因此每次修改 VecBase 特殊寄存器时都需要重新配置入口地址。
3. 配置 WCL_CORE_*m*_ENTRY_CHECK_REG 使能对任意入口地址的监测，一旦 CPU 执行到监控的入口地址处即立刻将 CPU 切换到安全世界。
   - 第 *x* 比特控制对 *x* 号入口 WCL_CORE_*m*_ENTRY_*x*_ADDR_REG 的监测，可配置为同时监测多个入口
     - 0: 禁用监测
     - 1: 开始监测
   - WCL_CORE_*m*_ENTRY_CHECK_REG 寄存器一次使能，长期有效。之后每次监测到入口地址的取指行为都会自动切换到安全世界，直至被禁用。

16.4.3 清除 write_buffer

由于 ESP32-S3 的 write_buffer 机制，因此在 CPU 切换世界时，总线上可能还残留另一个世界的信息并执行另一个世界的操作。为保证安全，必须在世界切换时，对 writer_buffer 进行冲刷，具体方法有两种：

- **安全世界切换至非安全世界时**：
  - 使用 memw 汇编命令。

- **从非安全世界进入安全世界时**：
  - 切换 CPU 数据总线：
    1. 预配置以下寄存器：
       * WCL_CORE_*m*_MESSAGE_ADDR_REG 为一段非安全世界的地址；
       * WCL_CORE_*m*_MESSAGE_MAX_REG 为 *Z* ($Z \in \{3, 4, \ldots, 16\}$)
    2. 每次切换世界时均向 WCL_CORE_*m*_MESSAGE_ADDR_REG 配置的地址中连续写入 0、1、…、*Z-1*、*Z* 这个序列。比如，*Z* 若配置为 3，则连续写入 0、1、2、3 这四个数字；如 *Z* 配置为 5，则连续写入 0、1、2、3、4、5。
    3. 此后，当硬件检测到 WCL_CORE_*m*_MESSAGE_ADDR_REG 配置的地址中开始连续写入 WCL_CORE_*m*_MESSAGE_MAX_REG 中约定的序列时，则将立即切换 CPU 数据总线至另一个世界。

16.5 世界切换记录表

通常情况下，CPU 会频繁进行世界切换且存在中断嵌套。为了能够回溯到切换至安全世界之前的世界属性，World 控制器还会自动记录世界切换日志，并将其保存在一系列寄存器中，即“世界切换记录表”。

16.5.1 世界切换记录表寄存器的组成

世界切换记录表由 13 个 WCL_CORE_*m*_STATUSTABLE*n*_REG(*n*: 1-13) 寄存器组成（结构如图 16.5-1 所示），其中 WCL_CORE_*m*_STATUSTABLE*x*_REG 寄存器记录 *x* 号入口 WCL_CORE_*m*_ENTRY_*x*_ADDR_REG 的世界切换情况。

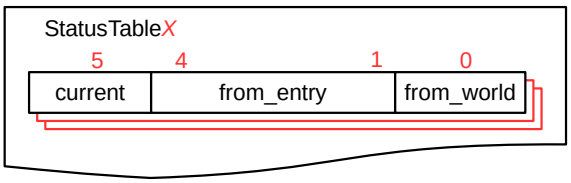


图 16.5-1. 世界切换记录表

具体描述如下：

- WCL_CORE_*m*_FROM_WORLD_*n*：记录世界切换之前的世界信息。
  - 0：安全世界
  - 1：非安全世界
- WCL_CORE_*m*_FROM_ENTRY_*n*：记录发生跳转的入口。
  - 0：表示切换之前不处于任何入口监测的中断。
  - 1 - 13：表示上一个发生中断的入口。
- WCL_CORE_*m*_CURRENT_*n*：表示该入口当前是否处于监测的中断中。当某个入口 *x* 发生中断时，
  - 该入口对应的 WCL_CORE_*m*_CURRENT_*x* 域均将被更新为 1，

- 其余入口的 current 域均将更新为 0。

16.5.2 世界切换记录表寄存器的更新

假设初始状态下，CPU 处于非安全世界，所有 WCL_CORE_*m*_STATUSTABLE*n*_REG(*n*: 1-13) 寄存器为 0。接着，入口 9、入口 1 和入口 4 依次发生优先级更高的中断。

此时，世界切换记录表更新过程如下：

1. 首先，9 号入口监测到中断发生。CPU 开始执行 9 号中断的入口地址时，世界切换记录表更新如图 16.5-2 所示：

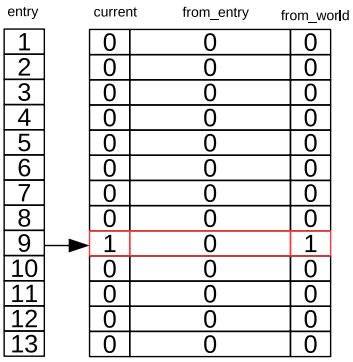


图 16.5-2. 中断嵌套流程示意图 - 9 号入口

其中，

- WCL_CORE_*m*_STATUSTABLE9_REG
  - WCL_CORE_*m*_FROM_WORLD_9 域为 1，表示 CPU 在切换之前处于非安全世界。
  - WCL_CORE_*m*_FROM_ENTRY_9 域为 0，表示切换之前不处于任何入口监测的中断。
  - WCL_CORE_*m*_CURRENT_9 域为 1，表示此时 CPU 正在 9 号入口监测的中断中。
- 其他 WCL_CORE_*m*_STATUSTABLE*n*_REG 寄存器均保持不变。

2. 接着，1 号入口监测到一个更高优先级的中断。CPU 开始执行 1 号中断的入口地址时，世界切换记录表再次发生更新，如图 16.5-3 所示：

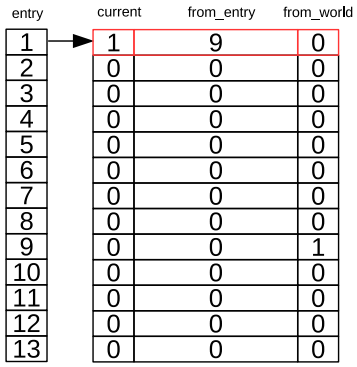


图 16.5-3. 中断嵌套流程示意图 - 1 号入口

其中，

- WCL_CORE_*m*_STATUSTABLE1_REG
  - WCL_CORE_*m*_FROM_WORLD_1 域为 0，表示 CPU 在切换之前处于安全世界。
  - WCL_CORE_*m*_FROM_ENTRY_1 域为 9，表示切换之前处于 9 号入口监测的中断。
  - WCL_CORE_*m*_CURRENT_1 域更新为 1，表示此时 CPU 正在 1 号入口监测的中断中。
- WCL_CORE_*m*_STATUSTABLE9_REG
  - WCL_CORE_*m*_CURRENT_9 域更新为 0，表示此时 CPU 已经不在 9 号入口监测的中断中（进入 1 号入口中断的监控中）。
  - WCL_CORE_*m*_FROM_WORLD_9 域和 WCL_CORE_*m*_FROM_ENTRY_9 域保持不变。
- 其他 WCL_CORE_*m*_STATUSTABLE*n*_REG 寄存器均保持不变。

3. 接着，4 号入口又监测到一个更高优先级的中断。CPU 开始执行 4 号中断的入口地址时，世界切换记录表再次发生更新，如图 16.5-4 所示：

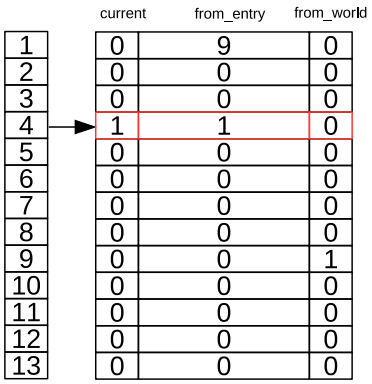


图 16.5-4. 中断嵌套流程示意图 - 4 号入口

其中，

- WCL_CORE_*m*_STATUSTABLE4_REG
  - WCL_CORE_*m*_FROM_WORLD_4 域为 0，表示 CPU 在切换之前处于安全世界。
  - WCL_CORE_*m*_FROM_ENTRY_4 域为 1，表示切换之前处于 1 号入口监测的中断。
  - WCL_CORE_*m*_CURRENT_4 域更新为 1，表示此时 CPU 正在 4 号入口监测的中断中。
- WCL_CORE_*m*_STATUSTABLE1_REG
  - WCL_CORE_*m*_CURRENT_1 域更新为 0，表示此时 CPU 已经不在 1 号入口监测的中断中（进入 4 号入口中断的监控中）。
  - WCL_CORE_*m*_FROM_WORLD_1 域和 WCL_CORE_*m*_FROM_ENTRY_1 域保持不变。
- 其他 WCL_CORE_*m*_STATUSTABLE*n*_REG 寄存器均保持不变。

16.5.3 世界切换记录表寄存器的读取

通过查看世界切换记录表，我们也可以了解世界切换前的世界以及是否发生中断嵌套等信息，从而能够正确恢复到世界切换前的状态。

具体方式如下（以图 16.5-4 为例）：

1. 查询 WCL_CORE_*m*_STATUSTABLE_CURRENT_REG 寄存器，得知目前 CPU 正处于 4 号入口监测的中断中。
2. 查询 WCL_CORE_*m*_FROM_ENTRY_4 域为 1，得知之前存在 1 号入口监测的中断。
3. 查询 WCL_CORE_*m*_FROM_ENTRY_1 域为 9，得知之前存在 9 号入口监测的中断。
4. 查询 WCL_CORE_*m*_FROM_ENTRY_9 域为 0，得知之前已经不存在其他入口监测的中断；此时，接着查询 WCL_CORE_*m*_FROM_WORLD_9 域为 1，得知世界切换前处于非安全世界。

16.5.4 中断嵌套

ESP32-S3 支持中断嵌套，举例而言：

1. 假设首先发生中断 A，CPU 跳转至中断 A 的入口地址并触发世界切换。
2. 接着，在 CPU 正在处理 A 中断时，又发生优先级更高的中断 B。
3. 此后，CPU 将停止处理中断 A，转而处理优先级更高的中断 B。
4. 处理完中断 B 后，CPU 将返回中断 A 的入口地址，这会再次触发世界切换。

这种情况会导致 世界切换记录表 记录出错，无法恢复正确的世界。

16.5.4.1 处理方法

为避免中断返回误触发的世界切换，用户需进行如下操作：

- 在设计阶段，在所有中断和异常的 vector 入口处插入两条 NOP.N (No Operation) 汇编指令。
- 在执行阶段，对当前中断的返回地址进行特别处理。

上述操作可以保证 *A* 号入口监测的中断在返回时不会返回到 *B* 号入口的入口地址，而是 NOP 指令之后的地址，从而确保世界切换记录表不会触发。此外，由于所有中断前两条指令均为 NOP 指令，因此即使未返回至起始地址也不会影响程序的正常执行。

16.5.4.2 编程指南

处理 *A* 处的中断：

1. 执行清除 write_buffer 的操作，向 WCL_CORE_*m*_MESSAGE_ADDR_REG 配置的地址中连续写入 0、1、...、*Z-1*、*Z* 序列。
2. 正常执行中断服务程序。
3. 禁止所有中断使能。
4. 处理 *A* 处中断的返回地址：
    - 查询 *A* 号入口的世界切换记录寄存器中的 WCL_CORE_*m*_FROM_ENTRY_*A*：
        - 0：表示所有中断均已全部处理，需返回普通应用程序：
            (a) 更新 *A* 的世界切换寄存器：WCL_CORE_*m*_CURRENT_*A* 为 0，表示中断已不再 *A* 处。
            (b) 跳转至第 5 步。
        - 1 - 13：表示需返回其他 *B* 号入口监测的中断，
            (a) 首先检查并处理当前 *A* 号入口的中断返回地址，如果：

* 指向 *B* 号入口中断的第一条 NOP.N 指令，则将 *A* 号中断的返回地址加 4
* 指向 *B* 号入口中断的第二条 NOP.N 指令，则将 *A* 号中断的返回地址加 2
* 指向其他地址，返回地址不变

(b) 手动更新世界切换记录表：

* 更新 *A* 的世界切换寄存器：
  - WCL_CORE_*m*_CURRENT_*A* 为 0，表示中断已不在 *A* 处。
  - WCL_CORE_*m*_FROM_WORLD_*A* 和 WCL_CORE_*m*_FROM_ENTRY_*A* 的更新不作要求。
* 更新 *B* 的世界切换寄存器：
  - WCL_CORE_*m*_CURRENT_*B* 为 1，表示将返回至 *B* 入口。

5. 准备退出中断。

(a) 查询需前往的世界：

- 如仍处于相同世界，则跳转到第 6 步。
- 如需前往其他世界，则切换至需要的世界，参考16.4。然后跳转到第 6 步。

6. 恢复中断使能，退出中断。

说明：
上述处理过程中的 4 和 5 不能被高级中断打断，因此在执行这两个步骤之前应该将所有的中断关闭，处理完之后再将所有的中断打开。

16.6 NMI 中断屏蔽

ESP32-S3 的某些软件过程（比如 World 控制器的配置过程等）可能希望不被任何中断打断，包括 NMI 中断。

通常来说，我们可以通过 CPU 特殊寄存器，使用软件过程来屏蔽中断，但不包括 NMI 中断。NMI 中断必须通过修改中断源来配置，相对麻烦。（详见 9 中断矩阵 *(INTERRUPT)* ）

为了快速使能或屏蔽 NMI 中断，World 控制器设计了一种硬件机制：

- **所有**应用程序完全屏蔽 NMI 中断：
  - 配置 WCL_CORE_*m*_NMI_MASK
    * 1：可直接完全屏蔽所有 NMI 中断
    * 0：可取消屏蔽
- **部分**应用程序屏蔽 NMI 中断：
  1. 配置 WCL_CORE_*m*_NMI_MASK_TRIGGER_ADDR 为需要结束 NMI 中断的应用程序地址，即 CPU 执行到这个地址开始不再屏蔽 NMI 中断。
  2. 配置 WCL_CORE_*m*_NMI_MASK_DISABLE 为任意值，表明 TRIGGER_ADDR 配置完成

3. 配置 WCL_CORE_*m*_NMI_MASK_ENABLE 为任意值，开始屏蔽 NMI 中断，直到触发 WCL_CORE_*m*_NMI_MASK_TRIGGER_ADDR 后不再屏蔽 NMI 中断。

说明：

- 上述两种屏蔽方式不建议混用，需选择其中一种进行使用。
- NMI 中断屏蔽配置后长期有效，直至触发。触发之后需要重新配置。

16.7 寄存器列表

本小节的所有地址均为相对于 World 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

注意，CPU0 与 CPU1 的寄存器完全相同。下方表格仅列出 CPU0 的相关寄存器，CPU1 的寄存器仅需在 CPU0 相关寄存器地址偏移量的基础上再加 0x0400 即可。

比如 WCL_CORE_0_ENTRY_CHECK_REG 的地址偏移量为 0x007C，则 WCL_CORE_1_ENTRY_CHECK_REG 的地址偏移量为 0x047C。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问权限
非安全世界到安全世界配置寄存器			
WCL_CORE_0_ENTRY_*n*_ADDR_REG (*n*: 1-13)	CPU0 *n* 号入口的地址配置寄存器	0x0000 + 4*(*n*-1)	R/W
WCL_CORE_0_ENTRY_CHECK_REG	CPU0 的入口地址监测使能寄存器	0x007C	R/W
WCL_CORE_0_MESSAGE_ADDR_REG	CPU0 清除 Writer_buffer 配置寄存器 - 配置地址	0x0100	R/W
WCL_CORE_0_MESSAGE_MAX_REG	CPU0 清除 Writer_buffer 配置寄存器 - 配置序列	0x0104	R/W
WCL_CORE_0_MESSAGE_PHASE_REG	CPU0 清除 Writer_buffer 配置寄存器 - 检查状态	0x0108	RO
世界切换记录表寄存器			
WCL_CORE_0_STATUSTABLE*n*_REG(*n*: 1-13)	CPU0 *n* 号入口的世界切换状态寄存器	0x0080 + 4*(*n*-1)	R/W
WCL_CORE_0_STATUSTABLE_CURRENT_REG	CPU0 当前入口快速检查寄存器	0x00FC	R/W
安全世界到非安全世界配置寄存器			
WCL_CORE_0_World_TRIGGER_ADDR_REG	CPU0 触发地址配置寄存器	0x0140	RW
WCL_CORE_0_World_PREPARE_REG	CPU0 世界配置寄存器 - 配置入口地址	0x0144	R/W
WCL_CORE_0_World_UPDATE_REG	CPU0 世界配置寄存器 - 确认配置完成	0x0148	WO
WCL_CORE_0_World_Cancel_REG	CPU0 世界配置寄存器 - 取消配置	0x014C	WO
WCL_CORE_0_World_IRam0_REG	CPU0 IRAM0 总线世界状态寄存器	0x0150	R/W
WCL_CORE_0_World_DRam0_PIF_REG	CPU0 Dram0 和 PIF 总线世界状态寄存器	0x0154	R/W
WCL_CORE_0_World_Phase_REG	CPU0 世界状态寄存器	0x0158	RO
NMI 中断屏蔽配置寄存器			
WCL_CORE_0_NMI_MASK_ENABLE_REG	CPU0 NMI 中断屏蔽使能寄存器	0x0180	WO
WCL_CORE_0_NMI_MASK_TRIGGER_ADDR_REG	CPU0 NMI 中断屏蔽触发地址寄存器	0x0184	R/W
WCL_CORE_0_NMI_MASK_DISABLE_REG	CPU0 NMI 中断屏蔽禁用寄存器	0x0188	WO
WCL_CORE_0_NMI_MASK_CANCLE_REG	CPU0 NMI 中断屏蔽取消配置寄存器	0x018C	WO
WCL_CORE_0_NMI_MASK_REG	CPU0 NMI 中断屏蔽寄存器	0x0190	R/W
WCL_CORE_0_NMI_MASK_PHASE_REG	CPU0 NMI 中断屏蔽状态寄存器	0x0194	RO

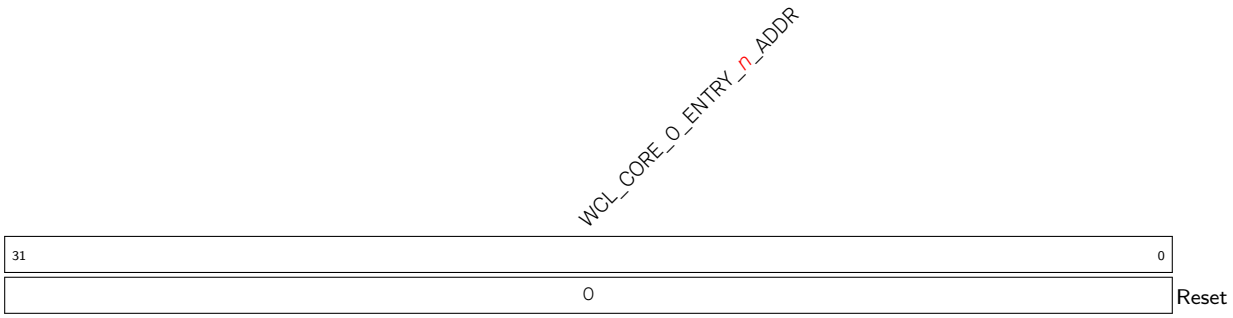
16.8 寄存器

本小节的所有地址均为相对于 World 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

注意，CPU0 与 CPU1 的寄存器完全相同。下方仅列出 CPU0 的相关寄存器，CPU1 的寄存器仅需在 CPU0 相关寄存器地址偏移量的基础上再加 0x0400 即可。

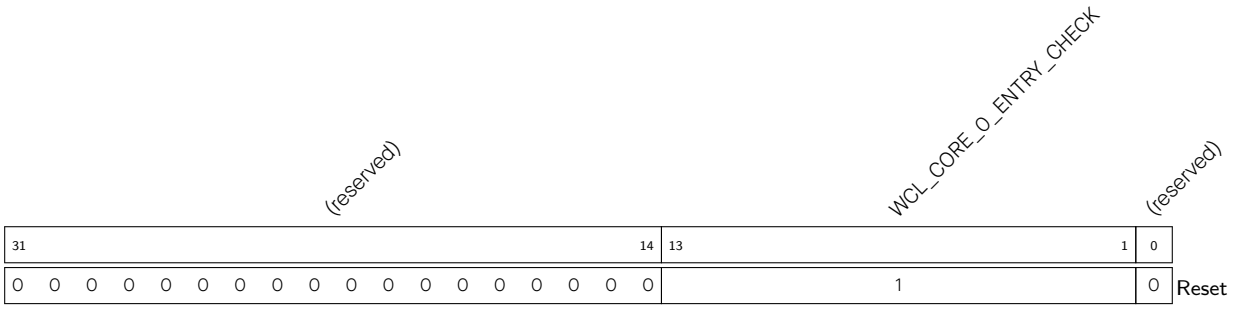
比如 WCL_CORE_0_ENTRY_CHECK_REG 的地址偏移量为 0x007C，则 WCL_CORE_1_ENTRY_CHECK_REG 的地址偏移量为 0x007C + 0x0400，即 0x047C。

Register 16.1. WCL_CORE_0_ENTRY_*n*_ADDR_REG (*n*: 1-13) (0x0000+4*(*n*-1))



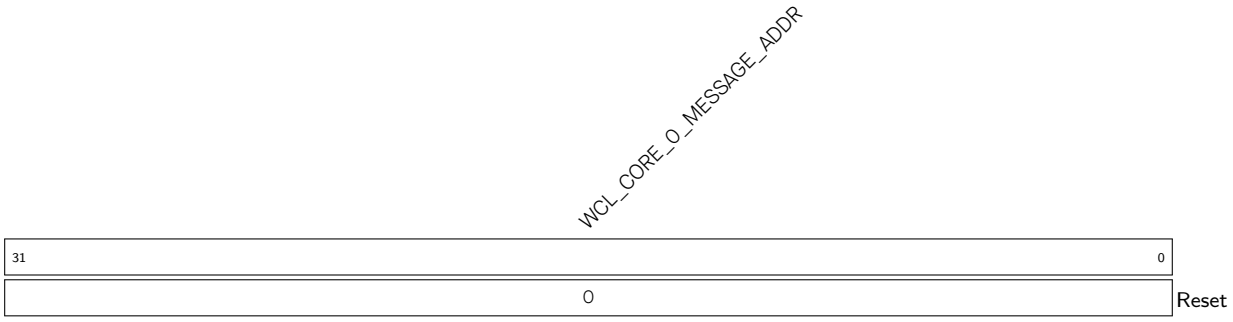
WCL_CORE_0_ENTRY_*n*_ADDR 配置 CPU0 从非安全世界进入安全世界的 *n* 号入口监测地址。(R/W)

Register 16.2. WCL_CORE_0_ENTRY_CHECK_REG (0x007C)



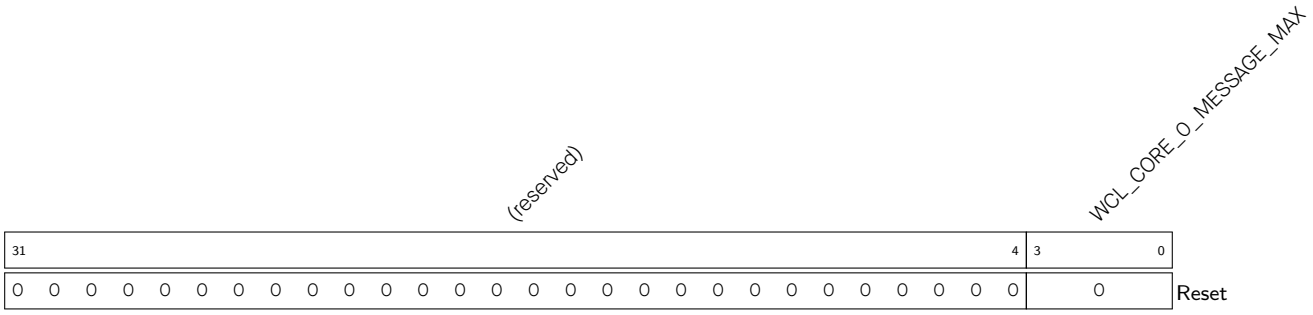
WCL_CORE_0_ENTRY_CHECK 置 1 使能 CPU0 在执行到监测地址时从非安全世界切换至安全世界。(R/W)

Register 16.3. WCL_CORE_0_MESSAGE_ADDR_REG (0x0100)



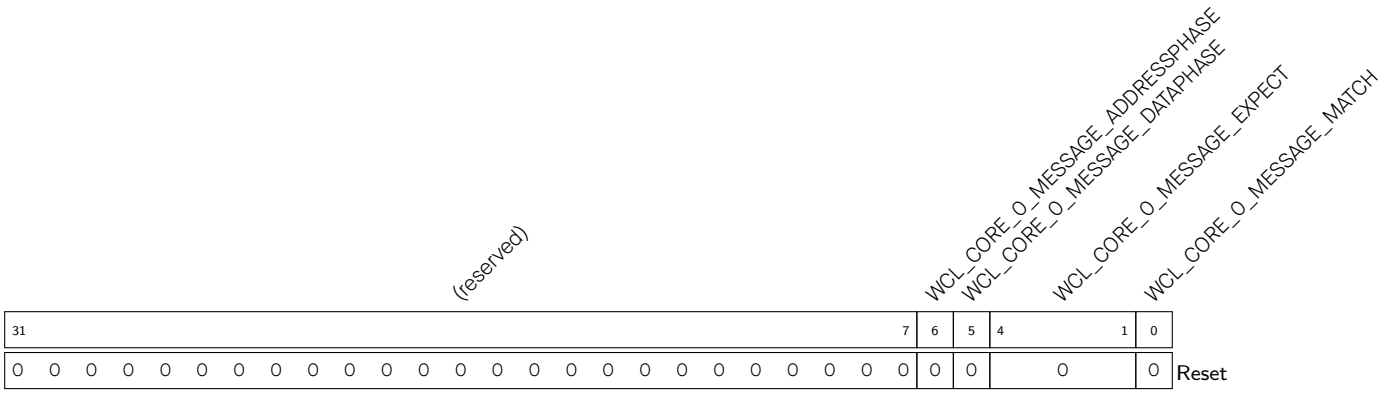
WCL_CORE_0_MESSAGE_ADDR 配置 CPU0 清除 write_buffer 时需写入规定序列的地址。(R/W)

Register 16.4. WCL_CORE_0_MESSAGE_MAX_REG (0x0104)



WCL_CORE_0_MESSAGE_MAX 配置 CPU0 清除 write_buffer 时需向规定地址写入的序列，建议配置不小于 3。(R/W)

Register 16.5. WCL_CORE_0_MESSAGE_PHASE_REG (0x0108)



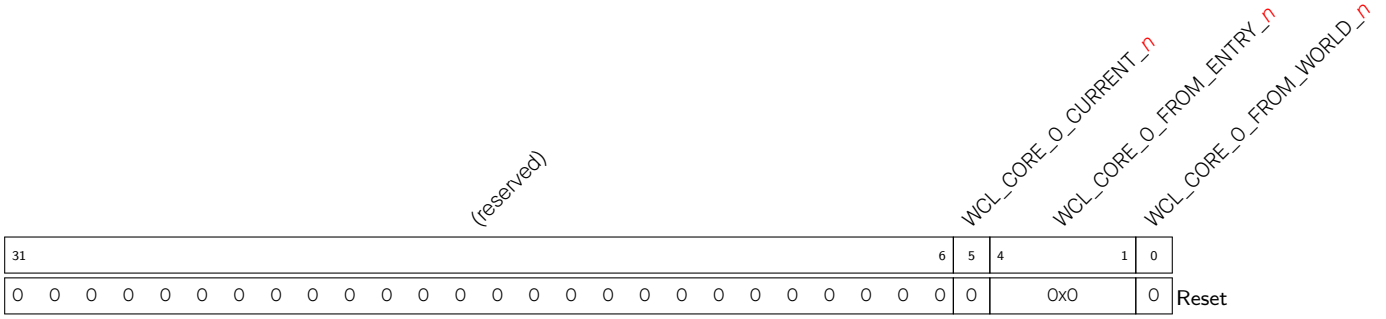
WCL_CORE_0_MESSAGE_MATCH 指示 CPU0 的世界切换是否成功。此域仅调试使用。(RO)

WCL_CORE_0_MESSAGE_EXPECT 存储下一个需写入的数字。此域仅调试使用。(RO)

WCL_CORE_0_MESSAGE_DATAPHASE 1 代表 CPU0 正在检查写入的序列是否匹配。此域仅调试使用。(RO)

WCL_CORE_0_MESSAGE_ADDRESSPHASE 1 代表 CPU0 正在检查是否有序列写入规定的地址。此域仅调试使用。(RO)

Register 16.6. WCL_CORE_0_STATUSTABLE*n*_REG (*n*: 1-13) (0x0080+4*(*n*-1))

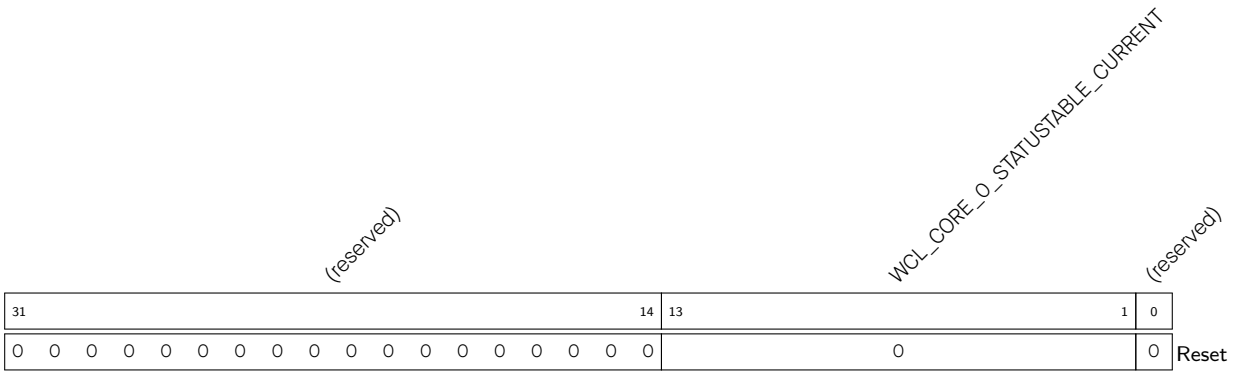


WCL_CORE_0_FROM_WORLD_*n* 存储 CPU0 进入 *n* 号入口中断之前的世界信息。(R/W)

WCL_CORE_0_FROM_ENTRY_*n* 存储 CPU0 进入 *n* 号入口中断之前的上一个入口信息。(R/W)

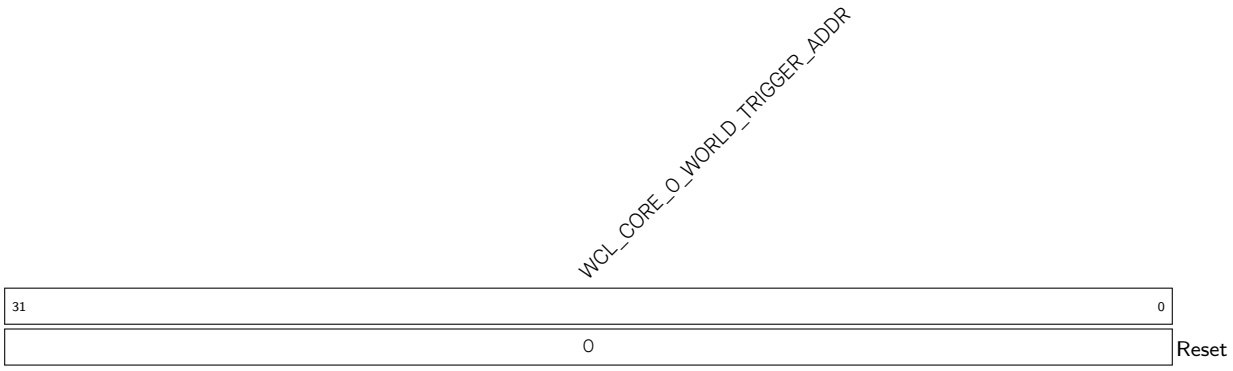
WCL_CORE_0_CURRENT_*n* 指示 CPU0 目前是否处于 *n* 号入口的中断。(R/W)

Register 16.7. WCL_CORE_0_STATUSTABLE_CURRENT_REG (0x00FC)



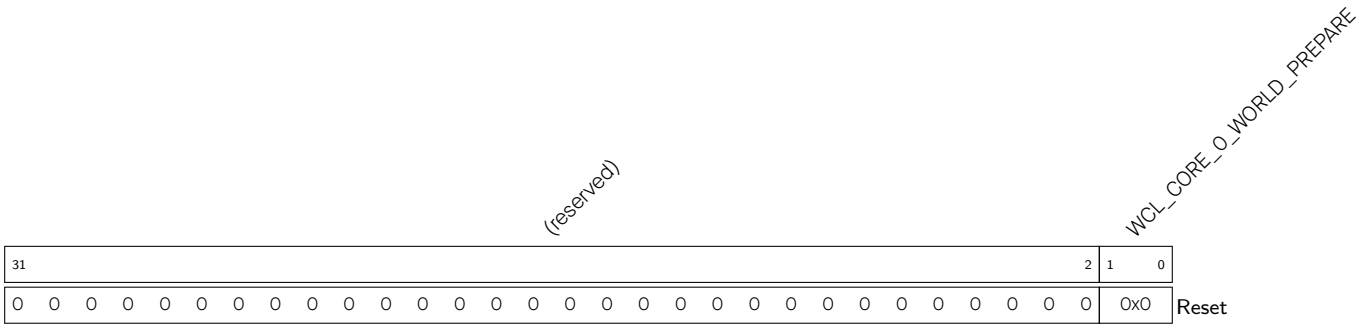
WCL_CORE_0_STATUSTABLE_CURRENT 指示 CPU0 目前正处于 *n* 号入口的中断。(R/W)

Register 16.8. WCL_CORE_0_World_TRIGGER_ADDR_REG (0x0140)



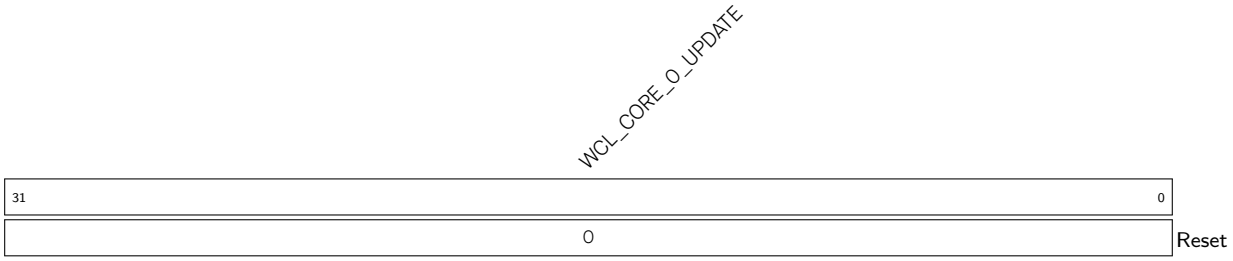
WCL_CORE_0_WORLD_TRIGGER_ADDR 配置 CPU0 从安全世界切换至非安全世界的入口地址。(RW)

Register 16.9. WCL_CORE_0_World_PREPARE_REG (0x0144)



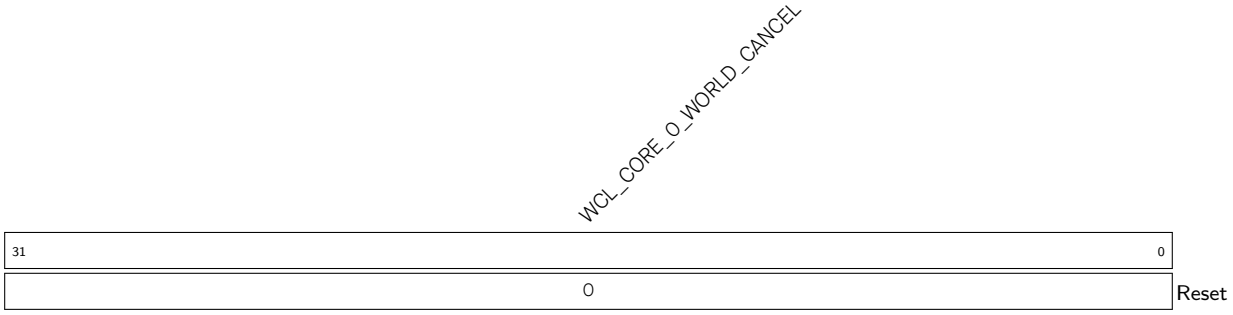
WCL_CORE_0_WORLD_PREPARE 写 0x2 可以使 CPU0 准备好从安全世界切换至非安全世界。此域仅调试使用。(R/W)

Register 16.10. WCL_CORE_0_World_UPDATE_REG (0x0148)



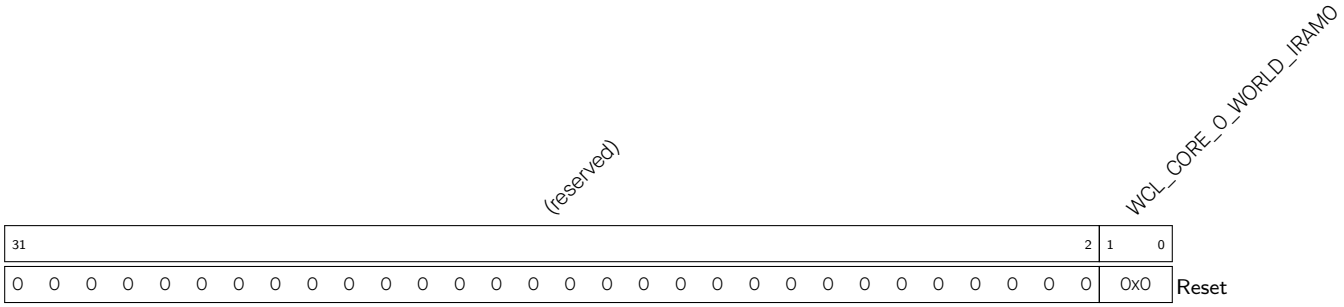
WCL_CORE_0_UPDATE 写入任意值代表 CPU0 已完成从安全世界切换至非安全世界的配置。(WO)

Register 16.11. WCL_CORE_0_World_Cancel_REG (0x014C)



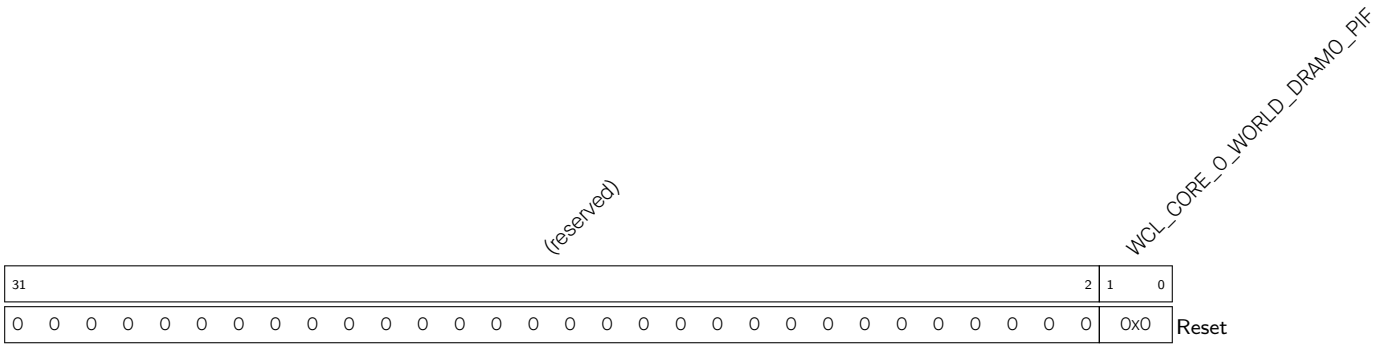
WCL_CORE_0_WORLD_CANCEL 写入任意值代表 CPU0 已取消从安全世界切换至非安全世界的配置。(WO)

Register 16.12. WCL_CORE_0_World_IRam0_REG (0x0150)



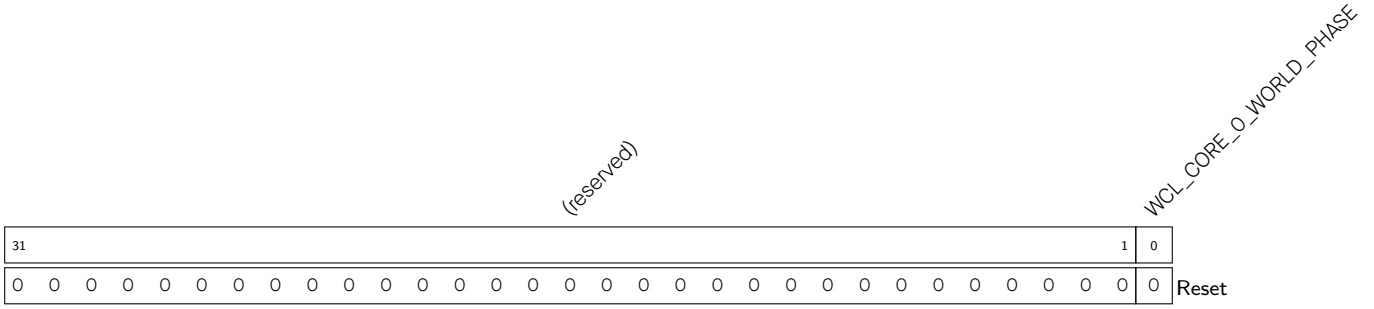
WCL_CORE_0_WORLD_IRAM0 存储 CPU0 指令总线的世界信息。此域仅调试使用。(R/W)

Register 16.13. WCL_CORE_0_World_DRam0_PIF_REG (0x0154)



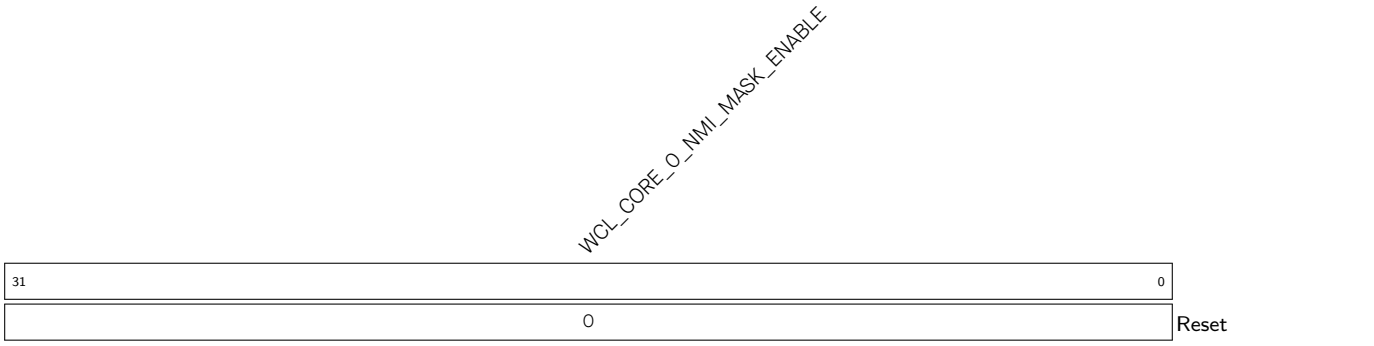
WCL_CORE_0_WORLD_DRAM0_PIF 存储 CPU0 数据总线和外设总线的世界信息。此域仅调试使用。(R/W)

Register 16.14. WCL_CORE_0_World_Phase_REG (0x0158)



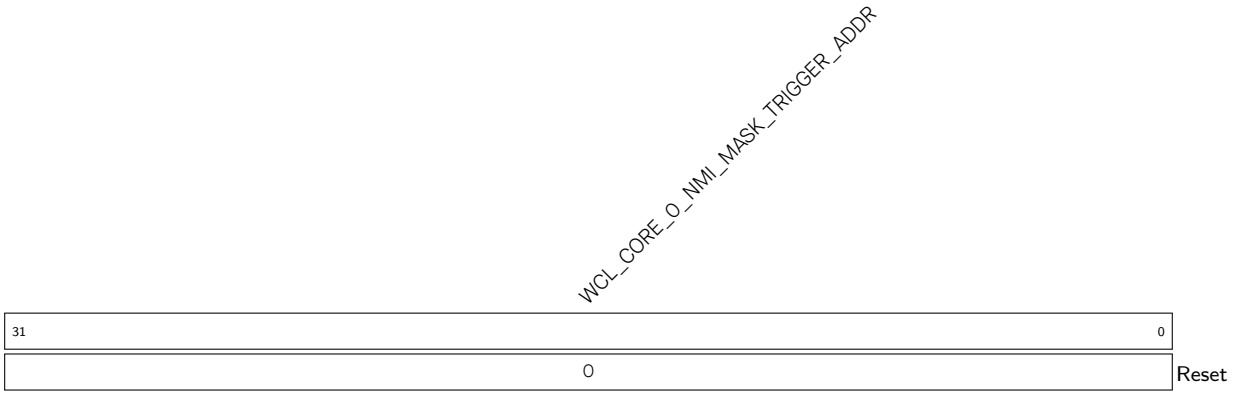
WCL_CORE_0_WORLD_PHASE 指示 CPU0 是否准备好从非安全世界切换至安全世界。1：准备好；2：未准备好。(RO)

Register 16.15. WCL_CORE_0_NMI_MASK_ENABLE_REG (0x0180)



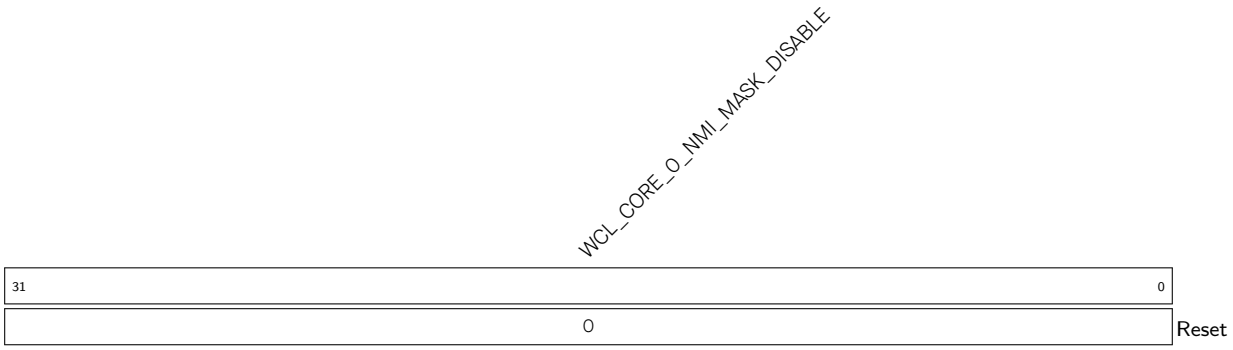
WCL_CORE_0_NMI_MASK_ENABLE 写入任意值开始屏蔽 CPU0 的所有 NMI 中断。(WO)

Register 16.16. WCL_CORE_0_NMI_MASK_TRIGGER_ADDR_REG (0x0184)



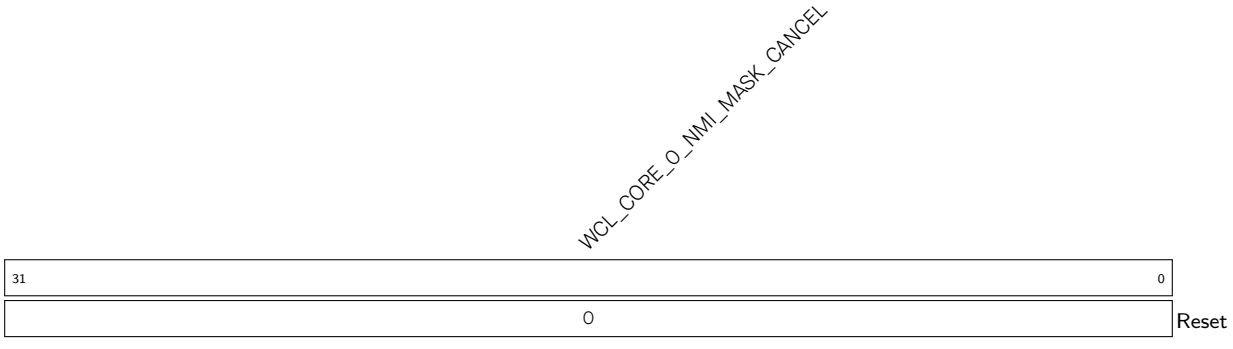
WCL_CORE_0_NMI_MASK_TRIGGER_ADDR 配置 CPU0 开始停止 NMI 中断屏蔽的地址。(R/W)

Register 16.17. WCL_CORE_0_NMI_MASK_DISABLE_REG (0x0188)



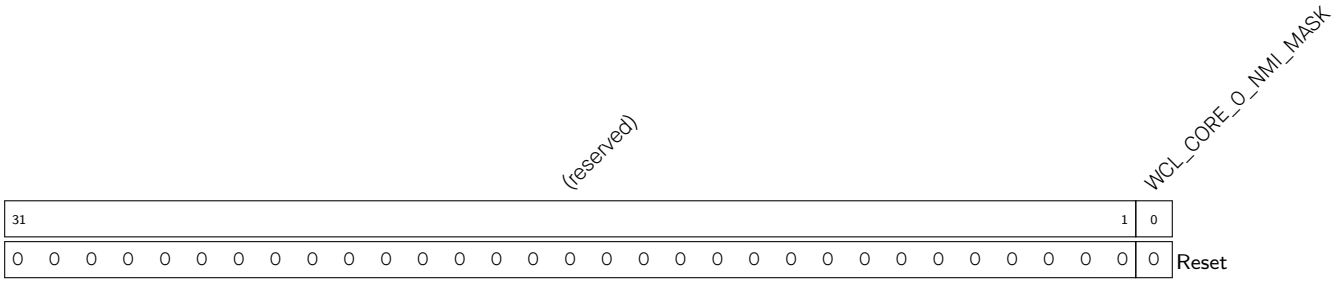
WCL_CORE_0_NMI_MASK_DISABLE 写入任意值使 CPU0 开始检查 NMI 中断屏蔽。(WO)

Register 16.18. WCL_CORE_0_NMI_MASK_CANCLE_REG (0x018C)



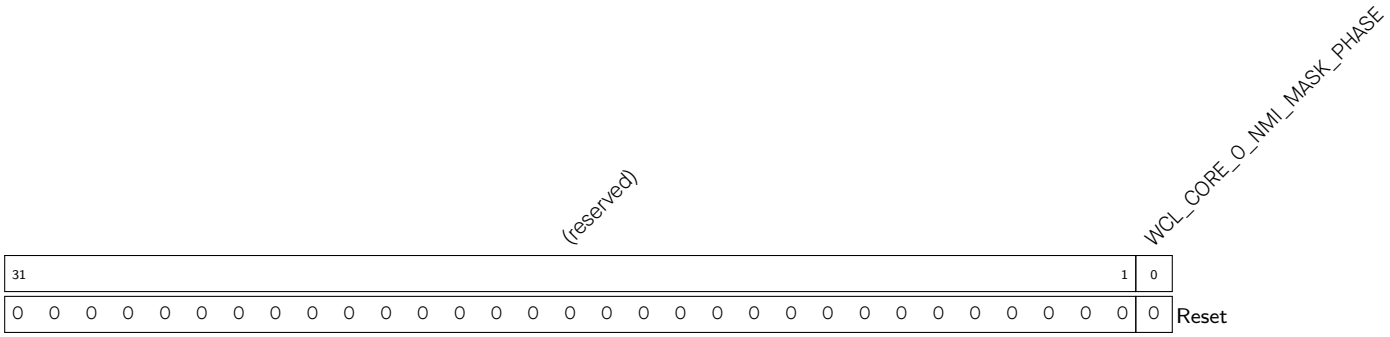
WCL_CORE_0_NMI_MASK_CANCEL 写入任意值取消 CPU0 的 NMI 中断屏蔽。(WO)

Register 16.19. WCL_CORE_0_NMI_MASK_REG (0x0190)



WCL_CORE_0_NMI_MASK 置 1 开始屏蔽 CPU0 的所有 NMI 中断。(R/W)

Register 16.20. WCL_CORE_0_NMI_MASK_PHASE_REG (0x0194)



WCL_CORE_0_NMI_MASK_PHASE 代表 CPU0 是否正在屏蔽 NMI 中断。1：正在屏蔽；0：未在屏蔽。(RO)

第 17 章

系统寄存器 (SYSTEM)

17.1 概述

ESP32-S3 集成了丰富的外设，且允许对不同外设模块进行独立控制，从而在保持最佳性能的同时将功耗降至最低。具体来说，ESP32-S3 设计了一系列系统配置寄存器，用于芯片的时钟管理（时钟门控）、功耗管理、外设模块及核心模块配置。本章将简要例举这些系统配置寄存器及其功能。

17.2 主要特性

ESP32-S3 的系统寄存器可用于控制以下外设和模块：

- 系统和存储器
- 时钟
- 软件中断
- 低功耗管理寄存器
- 外设时钟门控和复位
- CPU 控制

17.3 功能描述

17.3.1 系统和存储器寄存器

17.3.1.1 内部存储器

以下寄存器用以控制 ESP32-S3 内部 ROM 和 SRAM 存储器的功耗，具体来说：

- 在寄存器 SYSCON_CLKGATE_FORCE_ON_REG 中：
  - 设置 SYSCON_ROM_CLKGATE_FORCE_ON 的相应位可分别控制 Internal ROM 0、Internal ROM 1 和 Internal ROM 2 的时钟门控。
  - 设置 SYSCON_SRAM_CLKGATE_FORCE_ON 可以控制 Internal SRAM 不同 block 的时钟门控。
  - 配置为 1 时，ROM 或 SRAM 内存的时钟门控始终开启；配置为 0 时，则 ROM 或 SRAM 内存的时钟门控在被访问时自动打开，没有访问时自动关闭。因此，建议将本寄存器配置为 0，以降低功耗。
- 在寄存器 SYSCON_MEM_POWER_DOWN_REG 中：
  - 设置 SYSCON_ROM_POWER_DOWN 的相应位分别控制 Internal ROM 0 、Internal ROM 1 和 Internal ROM 2 进入 Retention 状态。

  - 设置 SYSCON_SRAM_POWER_DOWN 控制 Internal SRAM 对应 block 进入 Retention 状态。
  - Retention 状态是存储器的一种低功耗模式。在此状态下，存储器中的数据不会丢失，但是不允许访问，因此可降低功耗。所以，如果用户在一段时间内不会访问某些存储器，也可以配置 PD 寄存器让这些存储器进入 Retention 状态，以降低功耗。

- 在寄存器 SYSCON_MEM_POWER_UP_REG 中：
  - 默认情况下，芯片进入 Light-sleep 时会让所有的存储器进入 Retention 状态。
  - 设置 SYSCON_ROM_POWER_UP 的相应位可以强制 Internal ROM 0 、Internal ROM 1 和 Internal ROM 2 对应 block 在芯片进入 Light-sleep 时不会进入 Retention 状态。
  - 设置 SYSCON_SRAM_POWER_UP 可以强制 Internal SRAM 对应 block 在芯片进入 Light-sleep 时不会进入 Retention 状态。

更多有关内部存储器控制位的对应关系，请见下方表 17.3-1。

表 17.3-1. 内部存储器控制位

存储器	低地址 1	高地址 1	低地址 2	高地址 2	控制域
Internal ROM 0	0x4000_0000	0x4003_FFFF	-	-	Bit0
Internal ROM 1	0x4004_0000	0x4004_FFFF	-	-	Bit1
Internal ROM 2	0x4005_0000	0x4005_FFFF	0x3FF0_0000	0x3FF0_FFFF	Bit2
SRAM Block0	0x4037_0000	0x4037_3FFF	-	-	Bit0
SRAM Block1	0x4037_4000	0x4037_7FFF	-	-	Bit1
SRAM Block2	0x4037_8000	0x4037_FFFF	0x3FC8_8000	0x3FC8_FFFF	Bit2
SRAM Block3	0x4038_0000	0x4038_FFFF	0x3FC9_0000	0x3FC9_FFFF	Bit3
SRAM Block4	0x4039_8000	0x4039_FFFF	0x3FCA_0000	0x3FCA_FFFF	Bit4
SRAM Block5	0x403A_C000	0x403A_FFFF	0x3FCB_C000	0x3FCB_FFFF	Bit5
SRAM Block6	0x403B_0000	0x403B_FFFF	0x3FCC_0000	0x3FCC_FFFF	Bit6
SRAM Block7	0x403C_0000	0x403C_FFFF	0x3FCD_4000	0x3FCD_FFFF	Bit7
SRAM Block8	0x403D_0000	0x403D_BFFF	0x3FCE_8000	0x3FCE_FFFF	Bit8
SRAM Block9	-	-	0x3FCF_0000	0x3FCF_7FFF	Bit9
SRAM Block10	-	-	0x3FCF_8000	0x3FCF_FFFF	Bit10

更多信息，请见章节 4 *系统和存储器*。

17.3.1.2 片外存储器

SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 可用于控制外部存储的加解密配置，详情请见章节 23 *片外存储器加密与解密 (XTS_AES)*。

17.3.1.3 RSA 存储器

SYSTEM_RSA_PD_CTRL_REG 可控制 RSA 加速器中的 SRAM 存储器。

- SYSTEM_RSA_MEM_PD 置 1 控制 RSA 存储器进入 Retention 状态。此位的优先级最低，其设置可被 SYSTEM_RSA_MEM_FORCE_PU 覆盖。当 *数字签名 (DS)* 使用 RSA 加速器时，此位无效。
- SYSTEM_RSA_MEM_FORCE_PU 置 1 控制 RSA 存储器在芯片进入 Light sleep 时不会进入 Retention 状态。此位的优先级第二高，可覆盖 SYSTEM_RSA_MEM_PD 的设置。

- SYSTEM_RSA_MEM_FORCE_PD 置 1 控制 RSA 存储器进入 Retention 状态。此位的优先级最高，可以覆盖 SYSTEM_RSA_MEM_FORCE_PU 的设置。

17.3.2 时钟配置寄存器

以下系统寄存器用于系统和外设时钟源和时钟频率的配置。更多信息，请见章节 7 复位和时钟。

- SYSTEM_CPU_PER_CONF_REG
- SYSTEM_SYSCLK_CONF_REG
- SYSTEM_BT_LPCK_DIV_FRAC_REG

17.3.3 中断信号寄存器

以下系统寄存器用于产生中断信号，经过配置可通过中断矩阵，产生不同的 CPU 外设中断。当以下寄存器写为 1 时，会产生中断信号，可用于软件自己控制中断的产生。更多信息，请见章节 9 中断矩阵 *(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

17.3.4 低功耗管理寄存器

以下系统寄存器用于低功耗管理。更多信息，请见章节 10 低功耗管理 *(RTC_CNTL)*。

- SYSTEM_RTC_FASTMEM_CONFIG_REG：用于 RTC 快速内存的 CRC 配置
- SYSTEM_RTC_FASTMEM_CRC_REG：配置 CRC 校验值

17.3.5 外设时钟门控和复位寄存器

以下系统寄存器用于控制外设时钟门控和复位，相应位分别为不同外设的门控使能和复位使能，详见下方表 17.3-2。

- SYSTEM_CACHE_CONTROL_REG
- SYSTEM_EDMA_CTRL_REG
- SYSTEM_PERIP_CLK_EN0_REG
- SYSTEM_PERIP_RST_EN0_REG
- SYSTEM_PERIP_CLK_EN1_REG
- SYSTEM_PERIP_RST_EN1_REG

ESP32-S3 具有低功耗特性，因此有些外设时钟默认为关闭状态。在启用这些外设之前，必须将外设的时钟打开，并且解除外设的复位状态，具体见下表：

表 17.3-2. 外设时钟门控与复位控制位

外设	时钟使能位[1]	复位使能位[2][3]

EDMA Ctrl	SYSTEM_EDMA_CTRL_REG	
EDMA	SYSTEM_EDMA_CLK_ON	SYSTEM_EDMA_RESET
CACHE Ctrl	**SYSTEM_CACHE_CONTROL_REG**	
DCACHE	SYSTEM_DCACHE_CLK_ON	SYSTEM_DCACHE_RESET
ICACHE	SYSTEM_ICACHE_CLK_ON	SYSTEM_ICACHE_RESET
外设	**SYSTEM_PERIP_CLK_EN0_REG**	**SYSTEM_PERIP_RST_EN0_REG**
Timer Group0	SYSTEM_TIMERGROUP_CLK_EN	SYSTEM_TIMERGROUP_RST
Timer Group1	SYSTEM_TIMERGROUP1_CLK_EN	SYSTEM_TIMERGROUP1_RST
System Timer	SYSTEM_SYSTIMER_CLK_EN	SYSTEM_SYSTIMER_RST
UART0	SYSTEM_UART_CLK_EN	SYSTEM_UART_RST
UART1	SYSTEM_UART1_CLK_EN	SYSTEM_UART1_RST
UART MEM	SYSTEM_UART_MEM_CLK_EN [4]	SYSTEM_UART_MEM_RST
SPI0，SPI1	SYSTEM_SPI01_CLK_EN	SYSTEM_SPI01_RST
SPI2	SYSTEM_SPI2_CLK_EN	SYSTEM_SPI2_RST
SPI3	SYSTEM_SPI3_CLK_EN	SYSTEM_SPI3_RST
I2C0	SYSTEM_I2C_EXT0_CLK_EN	SYSTEM_I2C_EXT0_RST
I2C1	SYSTEM_I2C_EXT1_CLK_EN	SYSTEM_I2C_EXT1_RST
I2S0	SYSTEM_I2S0_CLK_EN	SYSTEM_I2S0_RST
I2S1	SYSTEM_I2S1_CLK_EN	SYSTEM_I2S1_RST
TWAI 控制器	SYSTEM_CAN_CLK_EN	SYSTEM_CAN_RST
UHCI0	SYSTEM_UHCI0_CLK_EN	SYSTEM_UHCI0_RST
USB	SYSTEM_USB_CLK_EN	SYSTEM_USB_RST
RMT	SYSTEM_RMT_CLK_EN	SYSTEM_RMT_RST
PCNT	SYSTEM_PCNT_CLK_EN	SYSTEM_PCNT_RST
PWM0	SYSTEM_PWM0_CLK_EN	SYSTEM_PWM0_RST
PWM1	SYSTEM_PWM1_CLK_EN	SYSTEM_PWM1_RST
LED_PWM 控制器	SYSTEM_LEDC_CLK_EN	SYSTEM_LEDC_RST
ADC 仲裁器	SYSTEM_ADC2_ARB_CLK_EN	SYSTEM_ADC2_ARB_RST
ADC 控制器	SYSTEM_APB_SARADC_CLK_EN	SYSTEM_APB_SARADC_RST
加速器	**SYSTEM_PERIP_CLK_EN1_REG**	**SYSTEM_PERIP_RST_EN1_REG**
USB_DEVICE	SYSTEM_USB_DEVICE_CLK_EN	SYSTEM_USB_DEVICE_RST
UART2	SYSTEM_UART2_CLK_EN	SYSTEM_UART2_RST
LCD_CAM	SYSTEM_LCD_CAM_CLK_EN	SYSTEM_LCD_CAM_RST
SDIO_HOST	SYSTEM_SDIO_HOST_CLK_EN	SYSTEM_SDIO_HOST_RST
DMA	SYSTEM_DMA_CLK_EN	SYSTEM_DMA_RST [5]
HMAC	SYSTEM_CRYPTO_HMAC_CLK_EN	SYSTEM_CRYPTO_HMAC_RST [6]
数字签名	SYSTEM_CRYPTO_DS_CLK_EN	SYSTEM_CRYPTO_DS_RST [7]
RSA 加速器	SYSTEM_CRYPTO_RSA_CLK_EN	SYSTEM_CRYPTO_RSA_RST
SHA 加速器	SYSTEM_CRYPTO_SHA_CLK_EN	SYSTEM_CRYPTO_SHA_RST
AES 加速器	SYSTEM_CRYPTO_AES_CLK_EN	SYSTEM_CRYPTO_AES_RST
peri backup	SYSTEM_PERI_BACKUP_CLK_EN	SYSTEM_PERI_BACKUP_RST

说明：

1. 时钟控制寄存器相应比特置 1 表示打开对应时钟，置 0 表示关闭对应时钟。
2. 复位寄存器相应比特置 1 表示使能复位状态，对应外设进行复位，置 0 表示关闭复位状态，对应外设正常工作。
3. 复位寄存器无法通过硬件清除，因此软件将外设复位后需要清除复位寄存器。
4. UART 存储器为所有 UART 外设所共用，因此只要有一个 UART 在工作，UART 存储器就不能处于门控状态。
5. 当外设需要通过 DMA 进行数据传输时，比如 UCHIO、SPI、I2S、LCD_CAM、AES、SHA、ADC 等，需要同时将 DMA 的时钟打开。
6. 该位复位后，SHA 加速器也会同时被复位。
7. 该位复位后，AES 加速器、SHA 加速器和 RSA 加速器也会同时被复位。

17.3.6 CPU 控制寄存器

系统上电默认仅启动 CPU0，此时 CPU1 的时钟处于关闭状态。因此，若需要使用双核 CPU，需要打开 CPU1 的时钟。以下寄存器用于控制 CPU1 的时钟使能、复位和运行状态等。

- SYSTEM_CORE_1_CONTROL_0_REG
  - SYSTEM_CONTROL_CORE_1_RESETING 位用于控制 CPU1 的复位。
  - SYSTEM_CONTROL_CORE_1_CLKGATE_EN 位用于打开和关闭 CPU1 的时钟。
  - SYSTEM_CONTROL_CORE_1_RUNSTALL 位用于暂停 CPU1，当此位置 1 时，CPU1 会将当前正在执行的操作完成然后停止运行。
- SYSTEM_CORE_1_CONTROL_1_REG 用于协调 CPU0 和 CPU1 之间的通信。具体来说，这是一个可读可写的寄存器，其值不会影响硬件功能。因此，软件可以使用此寄存器来进行双核通信，由一个 CPU 按照约定好的格式进行写操作，然后由另一个 CPU 进行读操作，从而实现两个 CPU 之间的通信。

17.4 寄存器列表

本小节的所有以 SYSTEM 开头的寄存器地址均为相对于系统寄存器基地址的地址偏移量（相对地址），所有以 APB 开头的寄存器地址均为相对于 APB 控制寄存器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

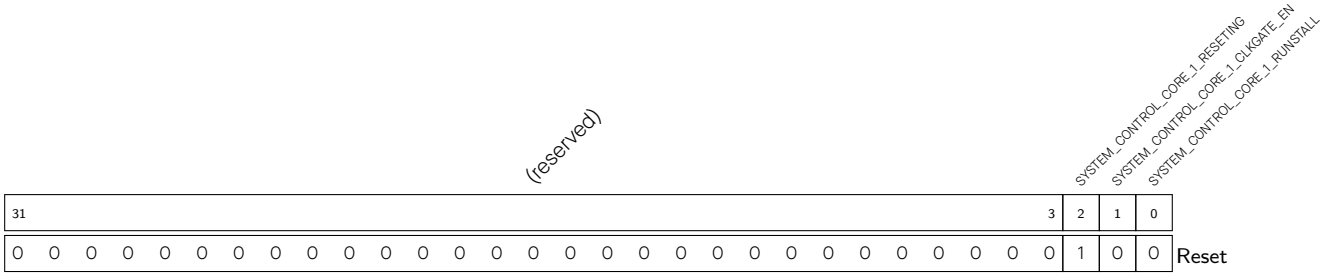
名称	描述	地址	访问权限
SYSTEM_CORE_1_CONTROL_0_REG	Core1 控制寄存器 0	0x0000	读写
SYSTEM_CORE_1_CONTROL_1_REG	Core1 控制寄存器 1	0x0004	读写
SYSTEM_CPU_PER_CONF_REG	CPU 外设时钟配置寄存器	0x0010	读写
SYSTEM_PERIP_CLK_EN0_REG	系统外设时钟使能寄存器 0	0x0018	读写
SYSTEM_PERIP_CLK_EN1_REG	系统外设时钟使能寄存器 1	0x001C	读写
SYSTEM_PERIP_RST_EN0_REG	系统外设复位寄存器 0	0x0020	读写
SYSTEM_PERIP_RST_EN1_REG	系统外设复位寄存器 1	0x0024	读写
SYSTEM_BT_LPCK_DIV_FRAC_REG	低功耗时钟配置寄存器 1	0x002C	读写
SYSTEM_CPU_INTR_FROM_CPU_0_REG	软件中断源寄存器 0	0x0030	读写
SYSTEM_CPU_INTR_FROM_CPU_1_REG	软件中断源寄存器 1	0x0034	读写
SYSTEM_CPU_INTR_FROM_CPU_2_REG	软件中断源寄存器 2	0x0038	读写
SYSTEM_CPU_INTR_FROM_CPU_3_REG	软件中断源寄存器 3	0x003C	读写
SYSTEM_RSA_PD_CTRL_REG	RSA 内存掉电寄存器	0x0040	读写
SYSTEM_EDMA_CTRL_REG	EDMA 控制寄存器	0x0044	读写
SYSTEM_CACHE_CONTROL_REG	Cache 控制寄存器	0x0048	读写
SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG	外部内存加解密控制寄存	0x004C	读写
SYSTEM_RTC_FASTMEM_CONFIG_REG	快速内存 CRC 配置寄存器	0x0050	可变
SYSTEM_RTC_FASTMEM_CRC_REG	快速内存 CRC 结果寄存器	0x0054	只读
SYSTEM_CLOCK_GATE_REG	系统时钟控制寄存器	0x005C	读写
SYSTEM_SYSCLK_CONF_REG	系统时钟配置寄存器	0x0060	可变
SYSTEM_DATE_REG	版本控制寄存器	0x0FFC	读写

名称	描述	地址	访问权限
SYSCON_CLKGATE_FORCE_ON_REG	内存时钟门控使能寄存器	0x00A8	读/写
SYSCON_MEM_POWER_DOWN_REG	内存控制寄存器	0x00AC	读/写
SYSCON_MEM_POWER_UP_REG	内存控制寄存器	0x00B0	读/写

17.5 寄存器

本小节的所有以 SYSTEM 开头的寄存器地址均为相对于系统寄存器基地址的地址偏移量（相对地址），所有以 APB 开头的寄存器地址均为相对于 APB 控制寄存器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 17.1. SYSTEM_CORE_1_CONTROL_0_REG (0x0000)

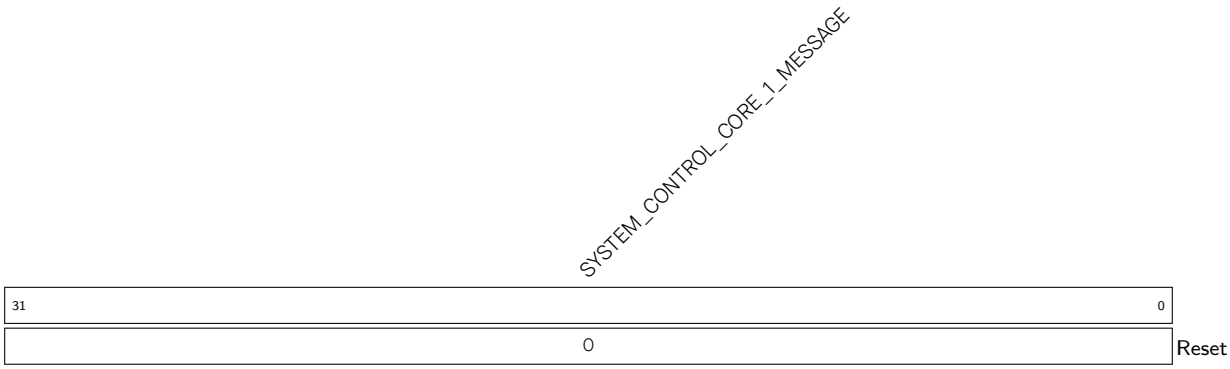


SYSTEM_CONTROL_CORE_1_RUNSTALL 置 1 停止 Core 1。(R/W)

SYSTEM_CONTROL_CORE_1_CLKGATE_EN 置 1 使能 Core 1 时钟。(R/W)

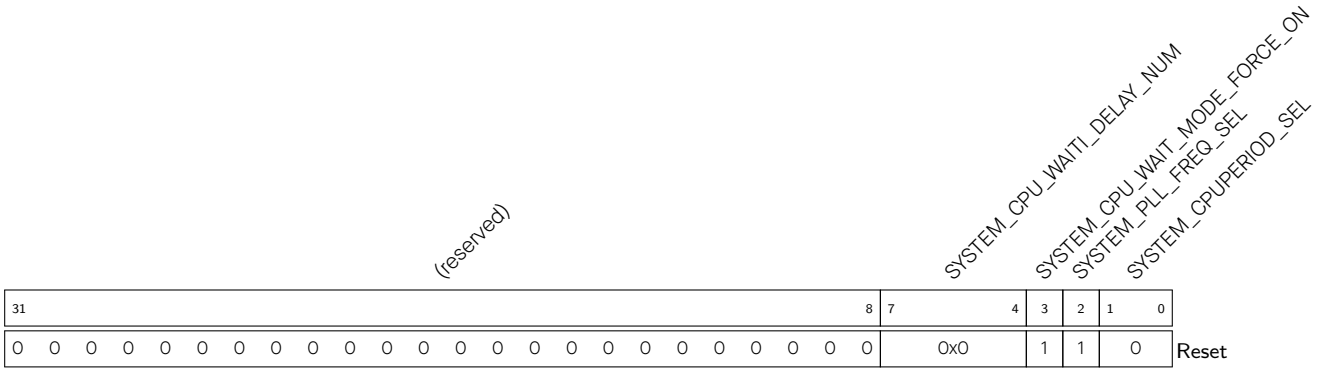
SYSTEM_CONTROL_CORE_1_RESETING 置 1 复位 Core 1。(R/W)

Register 17.2. SYSTEM_CORE_1_CONTROL_1_REG (0x0004)



SYSTEM_CONTROL_CORE_1_MESSAGE 用于设置 CPU1 的启动地址。注意，此字段仅首次启动 CPU1 时有效，此后用户可根据自身需求配作他用。(R/W)

Register 17.3. SYSTEM_CPU_PER_CONF_REG (0x0010)



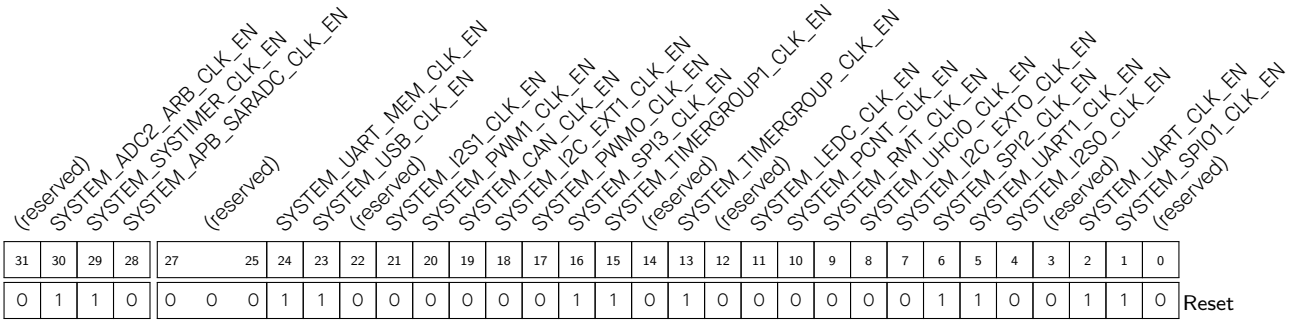
SYSTEM_CPUPERIOD_SEL 选择 CPU 时钟频率。(R/W)

SYSTEM_PLL_FREQ_SEL 选择 PLL 时钟频率。(R/W)

SYSTEM_CPU_WAIT_MODE_FORCE_ON 置 1 强制打开 CPU 等待中断模式下的门控时钟。通常情况下，CPU 执行 WAITI 指令后会进入等待中断模式。在此模式下 CPU 的时钟门控一直处于关闭状态，直到中断产生，因此可降低功耗。若此位置 1，CPU 的门控时钟会被强制打开，不受 WAITI 指令的影响。(R/W)

SYSTEM_CPU_WAITI_DELAY_NUM 设置 CPU 在收到 WAITI 指令后进入 CPU 等待中断模式后，关闭 CPU 的门控时钟需要的等待周期。(R/W)

Register 17.4. SYSTEM_PERIP_CLK_EN0_REG (0x0018)



SYSTEM_SPI01_CLK_EN 置 1 使能 SPI01 时钟。(R/W)

SYSTEM_UART_CLK_EN 置 1 使能 UART 时钟。(R/W)

SYSTEM_I2S0_CLK_EN 置 1 使能 I2S0 时钟。(R/W)

SYSTEM_UART1_CLK_EN 置 1 使能 UART1 时钟。(R/W)

SYSTEM_SPI2_CLK_EN 置 1 使能 SPI2 时钟。(R/W)

SYSTEM_I2C_EXT0_CLK_EN 置 1 使能 I2C_EXT0 时钟。(R/W)

SYSTEM_UHCI0_CLK_EN 置 1 使能 UHCI0 时钟。(R/W)

SYSTEM_RMT_CLK_EN 置 1 使能 RMT 时钟。(R/W)

SYSTEM_PCNT_CLK_EN 置 1 使能 PCNT 时钟。(R/W)

SYSTEM_LEDC_CLK_EN 置 1 使能 LEDC 时钟。(R/W)

SYSTEM_TIMERGROUP_CLK_EN 置 1 使能 TIMERGROUP0 时钟。(R/W)

SYSTEM_TIMERGROUP1_CLK_EN 置 1 使能 TIMERGROUP1 时钟。(R/W)

SYSTEM_SPI3_CLK_EN 置 1 使能 SPI3 时钟。(R/W)

SYSTEM_PWM0_CLK_EN 置 1 使能 PWM0 时钟。(R/W)

SYSTEM_I2C_EXT1_CLK_EN 置 1 使能 I2C_EXT1 时钟。(R/W)

SYSTEM_CAN_CLK_EN 置 1 使能 CAN 时钟。(R/W)

SYSTEM_PWM1_CLK_EN 置 1 使能 PWM1 时钟。(R/W)

SYSTEM_I2S1_CLK_EN 置 1 使能 I2S1 时钟。(R/W)

SYSTEM_USB_CLK_EN 置 1 使能 USB 时钟。(R/W)

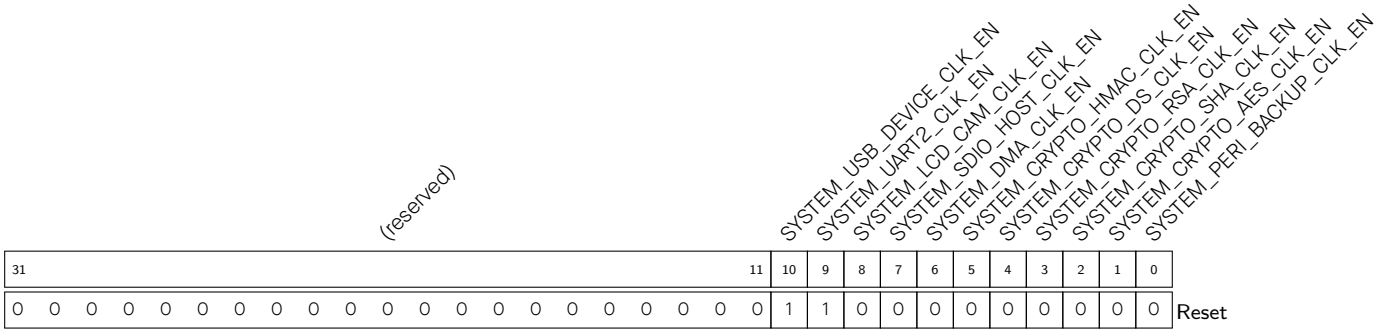
SYSTEM_UART_MEM_CLK_EN 置 1 使能 UART_MEM 时钟。(R/W)

SYSTEM_APB_SARADC_CLK_EN 置 1 使能 ADC 控制器时钟。(R/W)

SYSTEM_SYSTIMER_CLK_EN 置 1 使能 SYSTEM TIMER 时钟。(R/W)

SYSTEM_ADC2_ARB_CLK_EN 置 1 使能 ADC2_ARB 时钟。(R/W)

Register 17.5. SYSTEM_PERIP_CLK_EN1_REG (0x001C)



SYSTEM_PERI_BACKUP_CLK_EN 置 1 使能 peri backup 时钟。(R/W)

SYSTEM_CRYPTO_AES_CLK_EN 置 1 使能 AES 时钟。(R/W)

SYSTEM_CRYPTO_SHA_CLK_EN 置 1 使能 SHA 时钟。(R/W)

SYSTEM_CRYPTO_RSA_CLK_EN 置 1 使能 RSA 时钟。(R/W)

SYSTEM_CRYPTO_DS_CLK_EN 置 1 使能 DS 时钟。(R/W)

SYSTEM_CRYPTO_HMAC_CLK_EN 置 1 使能 HMAC 时钟。(R/W)

SYSTEM_DMA_CLK_EN 置 1 使能 DMA 时钟。(R/W)

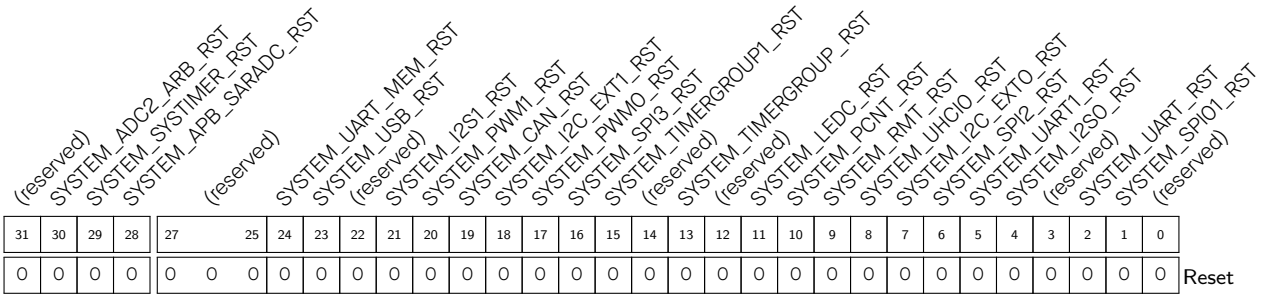
SYSTEM_SDIO_HOST_CLK_EN 置 1 使能 SDIO_HOST 时钟。(R/W)

SYSTEM_LCD_CAM_CLK_EN 置 1 使能 LCD_CAM 时钟。(R/W)

SYSTEM_UART2_CLK_EN 置 1 使能 UART2 时钟。(R/W)

SYSTEM_USB_DEVICE_CLK_EN 置 1 使能 USB_DEVICE 时钟。(R/W)

Register 17.6. SYSTEM_PERIP_RST_EN0_REG (0x0020)



SYSTEM_SPI01_RST 置 1 复位 SPI01。(R/W)

SYSTEM_UART_RST 置 1 复位 UART。(R/W)

SYSTEM_I2S0_RST 置 1 复位 I2S0。(R/W)

SYSTEM_UART1_RST 置 1 复位 UART1。(R/W)

SYSTEM_SPI2_RST 置 1 复位 SPI2。(R/W)

SYSTEM_I2C_EXT0_RST 置 1 复位 I2C_EXT0。(R/W)

SYSTEM_UHCI0_RST 置 1 复位 UHCI0。(R/W)

SYSTEM_RMT_RST 置 1 复位 RMT。(R/W)

SYSTEM_PCNT_RST 置 1 复位 PCNT。(R/W)

SYSTEM_LEDC_RST 置 1 复位 LEDC。(R/W)

SYSTEM_TIMERGROUP_RST 置 1 复位 TIMERGROUP0。(R/W)

SYSTEM_TIMERGROUP1_RST 置 1 复位 TIMERGROUP1。(R/W)

SYSTEM_SPI3_RST 置 1 复位 SPI3。(R/W)

SYSTEM_PWM0_RST 置 1 复位 PWM0。(R/W)

SYSTEM_I2C_EXT1_RST 置 1 复位 I2C_EXT1。(R/W)

SYSTEM_CAN_RST 置 1 复位 CAN。(R/W)

SYSTEM_PWM1_RST 置 1 复位 PWM1。(R/W)

SYSTEM_I2S1_RST 置 1 复位 I2S1。(R/W)

SYSTEM_USB_RST 置 1 复位 USB。(R/W)

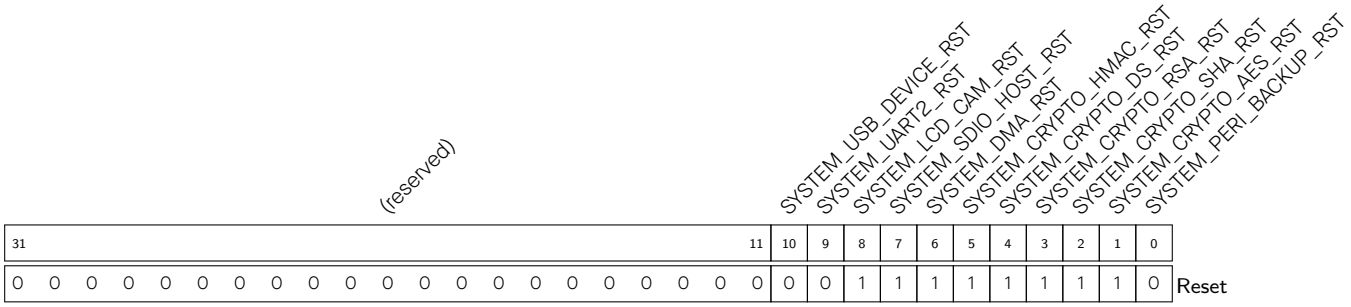
SYSTEM_UART_MEM_RST 置 1 复位 UART_MEM。(R/W)

SYSTEM_APB_SARADC_RST 置 1 复位 ADC 控制器。(R/W)

SYSTEM_SYSTIMER_RST 置 1 复位 SYSTIMER。(R/W)

SYSTEM_ADC2_ARB_RST 置 1 复位 ADC2_ARB。(R/W)

Register 17.7. SYSTEM_PERIP_RST_EN1_REG (0x0024)



SYSTEM_PERI_BACKUP_RST 置 1 复位 BACKUP。(R/W)

SYSTEM_CRYPTO_AES_RST 置 1 复位 CRYPTO_AES。(R/W)

SYSTEM_CRYPTO_SHA_RST 置 1 复位 CRYPTO_SHA。(R/W)

SYSTEM_CRYPTO_RSA_RST 置 1 复位 CRYPTO_RSA。(R/W)

SYSTEM_CRYPTO_DS_RST 置 1 复位 CRYPTO_DS。(R/W)

SYSTEM_CRYPTO_HMAC_RST 置 1 复位 CRYPTO_HMAC。(R/W)

SYSTEM_DMA_RST 置 1 复位 DMA。(R/W)

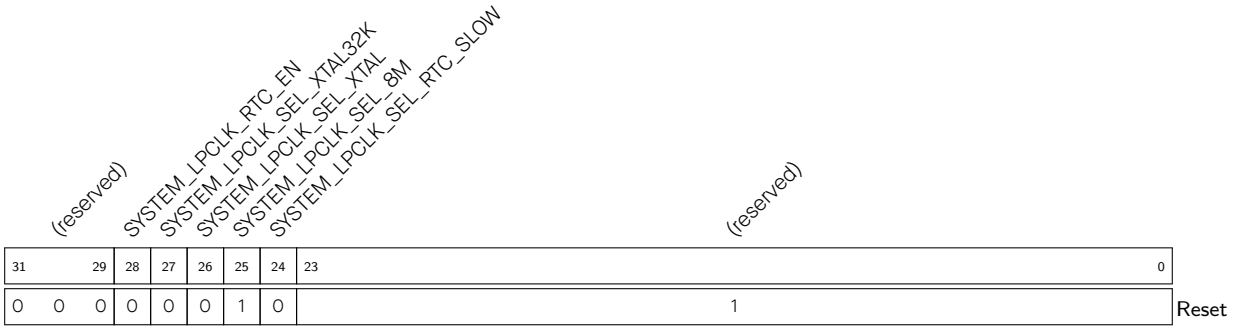
SYSTEM_SDIO_HOST_RST 置 1 复位 SDIO_HOST。(R/W)

SYSTEM_LCD_CAM_RST 置 1 复位 LCD_CAM。(R/W)

SYSTEM_UART2_RST 置 1 复位 UART2。(R/W)

SYSTEM_USB_DEVICE_RST 置 1 复位 USB_DEVICE。(R/W)

Register 17.8. SYSTEM_BT_LPCK_DIV_FRAC_REG (0x002C)



SYSTEM_LPCLK_SEL_RTC_SLOW 选择 RTC_SLOW_CLK 为低功耗时钟。(R/W)

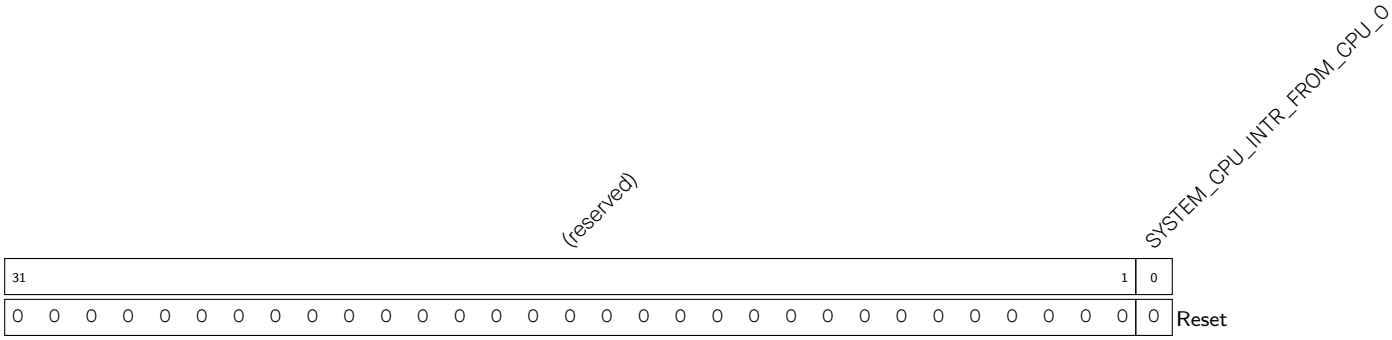
SYSTEM_LPCLK_SEL_8M 选择 RC_FAST_CLK 的 n 分频为低功耗时钟。(R/W)

SYSTEM_LPCLK_SEL_XTAL 选择 XTAL_CLK 时钟为低功耗时钟。(R/W)

SYSTEM_LPCLK_SEL_XTAL32K 选择 XTAL32K_CLK 时钟为低功耗时钟。(R/W)

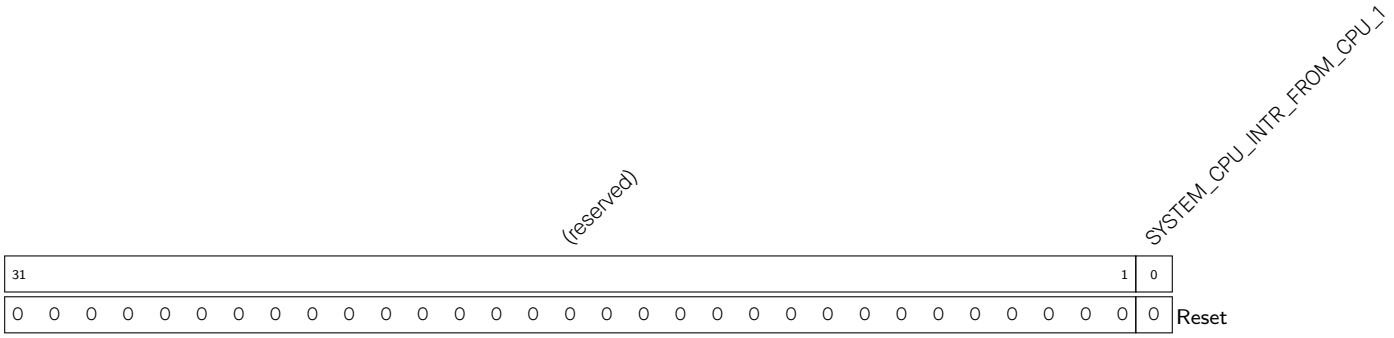
SYSTEM_LPCLK_RTC_EN 置 1 使能 LOW_POWER_CLK 低功耗时钟。(R/W)

Register 17.9. SYSTEM_CPU_INTR_FROM_CPU_0_REG (0x0030)



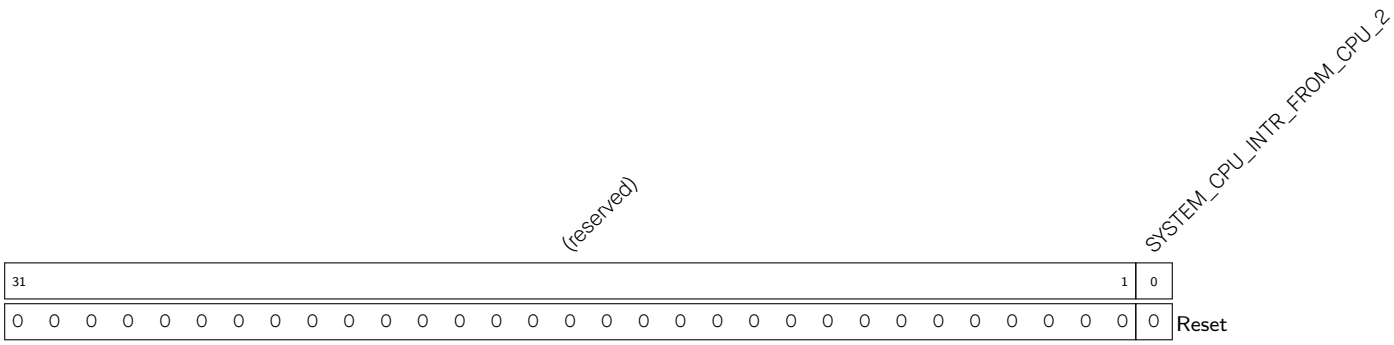
SYSTEM_CPU_INTR_FROM_CPU_0 置 1 生成 CPU 中断 0。该位需在 ISR 过程中由软件清 0。(R/W)

Register 17.10. SYSTEM_CPU_INTR_FROM_CPU_1_REG (0x0034)



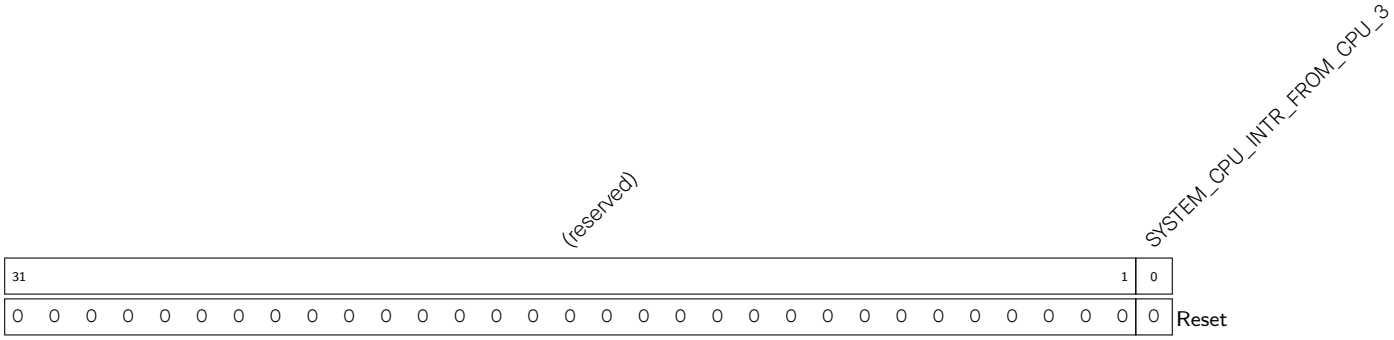
SYSTEM_CPU_INTR_FROM_CPU_1 置 1 生成 CPU 中断 1。该位需在 ISR 过程中由软件清 0。(R/W)

Register 17.11. SYSTEM_CPU_INTR_FROM_CPU_2_REG (0x0038)



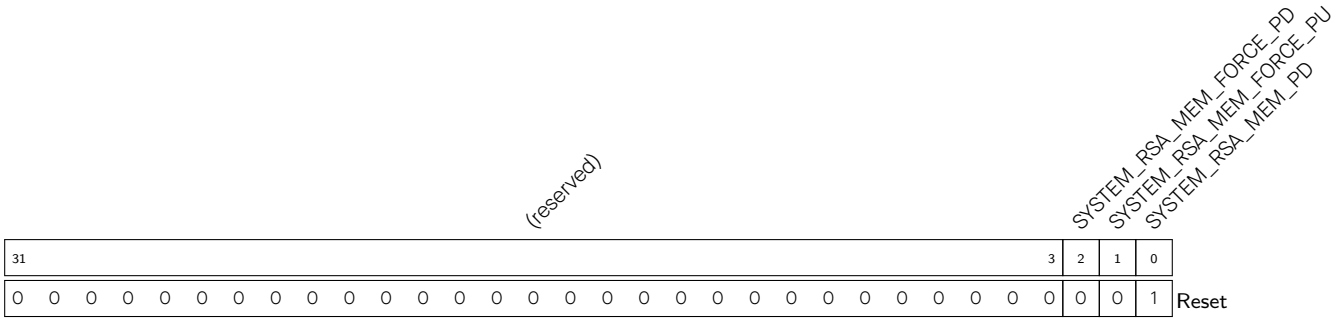
SYSTEM_CPU_INTR_FROM_CPU_2 置 1 生成 CPU 中断 2。该位需在 ISR 过程中由软件清 0。(R/W)

Register 17.12. SYSTEM_CPU_INTR_FROM_CPU_3_REG (0x003C)



SYSTEM_CPU_INTR_FROM_CPU_3 置 1 生成 CPU 中断 3。该位需在 ISR 过程中由软件清 0。(R/W)

Register 17.13. SYSTEM_RSA_PD_CTRL_REG (0x0040)

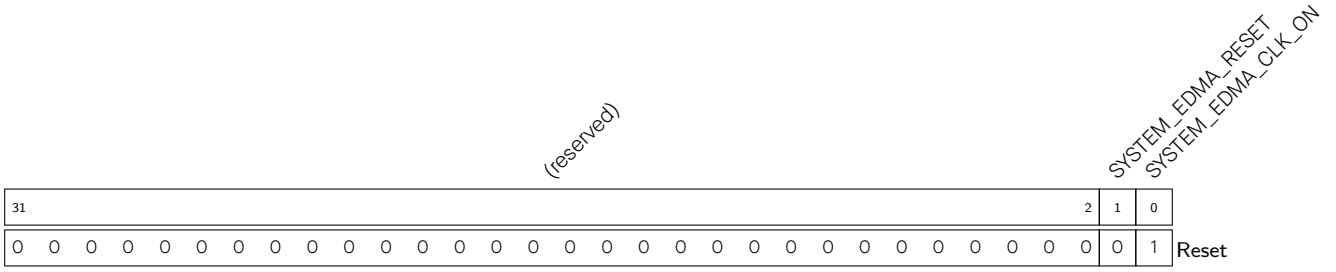


SYSTEM_RSA_MEM_PD 置 1 控制 RSA 存储器进入 Retention 状态。此位的优先级最低，其设置可被 SYSTEM_RSA_MEM_FORCE_PU 覆盖。当数字签名占用 RSA 加速器时，该位无效。(R/W)

SYSTEM_RSA_MEM_FORCE_PU 置 1 控制 RSA 存储器再芯片进入 Light sleep 时不会进入 Retention 状态。此位的优先级第二高，可覆盖 SYSTEM_RSA_MEM_PD 的设置。(R/W)

SYSTEM_RSA_MEM_FORCE_PD 置 1 控制 RSA 存储器进入 Retention 状态。此位的优先级最高，可以覆盖 SYSTEM_RSA_MEM_FORCE_PU 的设置。(R/W)

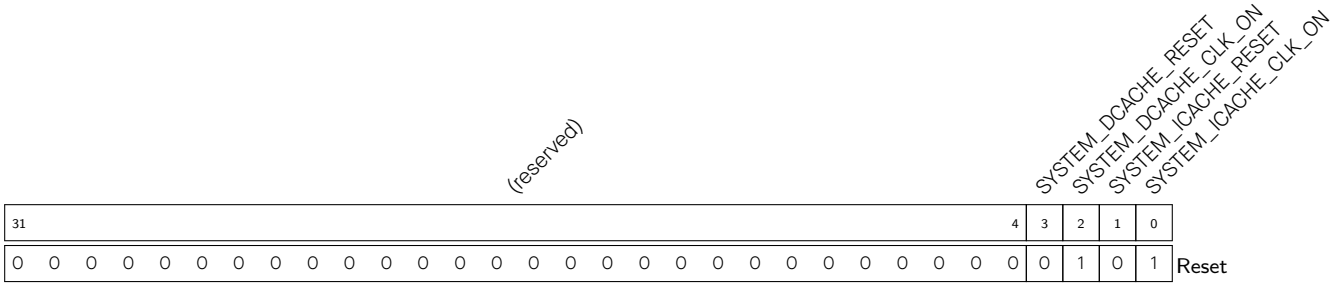
Register 17.14. SYSTEM_EDMA_CTRL_REG (0x0044)



SYSTEM_EDMA_CLK_ON 置 1 使能 EDMA 时钟。(R/W)

SYSTEM_EDMA_RESET 置 1 复位 EDMA。(R/W)

Register 17.15. SYSTEM_CACHE_CONTROL_REG (0x0048)



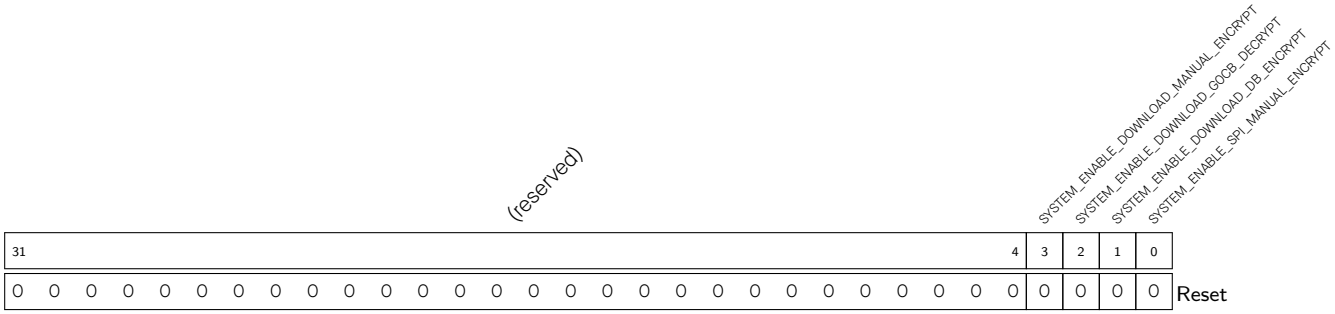
SYSTEM_ICACHE_CLK_ON 置 1 使能 i-cache 时钟。(R/W)

SYSTEM_ICACHE_RESET 置 1 复位 i-cache。(R/W)

SYSTEM_DCACHE_CLK_ON 置 1 使能 d-cache 时钟。(R/W)

SYSTEM_DCACHE_RESET 置 1 复位 d-cache。(R/W)

Register 17.16. SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG (0x004C)



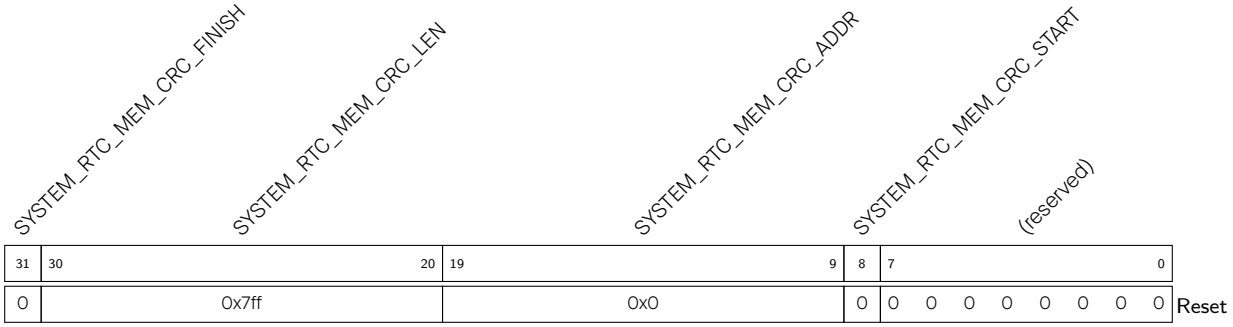
SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT 置 1 在 SPI Boot 模式下使能手动加密 (Manual Encryption)。(R/W)

SYSTEM_ENABLE_DOWNLOAD_DB_ENCRYPT 置 1 在 Download Boot 模式下使能自动加密 (Auto Encryption)。(R/W)

SYSTEM_ENABLE_DOWNLOAD_GOCB_DECRYPT 置 1 在 Download Boot 模式下使能自动解密 (Auto Decryption)。(R/W)

SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT 置 1 在 Download Boot 模式下使能手动加密 (Manual Encryption)。(R/W)

Register 17.17. SYSTEM_RTC_FASTMEM_CONFIG_REG (0x0050)



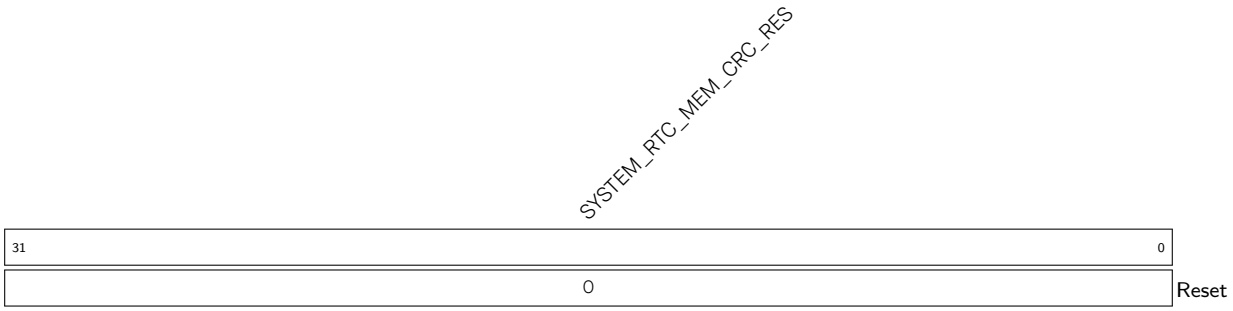
SYSTEM_RTC_MEM_CRC_START 置 1 启动 RTC 内存的 CRC 校验。(R/W)

SYSTEM_RTC_MEM_CRC_ADDR 设置 CRC 校验的 RTC 存储地址。(R/W)

SYSTEM_RTC_MEM_CRC_LEN 设置用于 CRC 校验的 RTC 存储长度（基于起始地址）。(R/W)

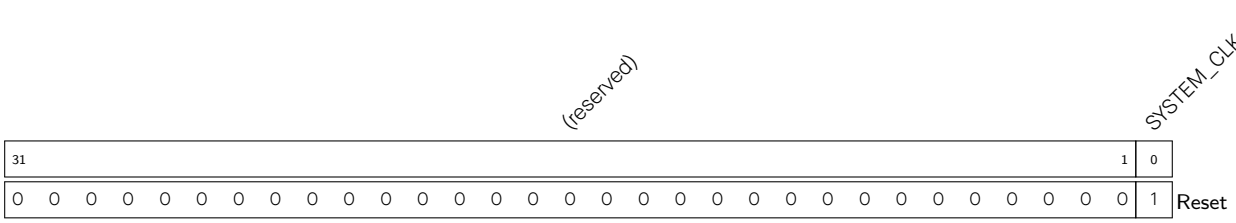
SYSTEM_RTC_MEM_CRC_FINISH 储存 RTC 存储 CRC 校验状态。高电平表示校验完成，低电平表示校验未完成。(RO)

Register 17.18. SYSTEM_RTC_FASTMEM_CRC_REG (0x0054)



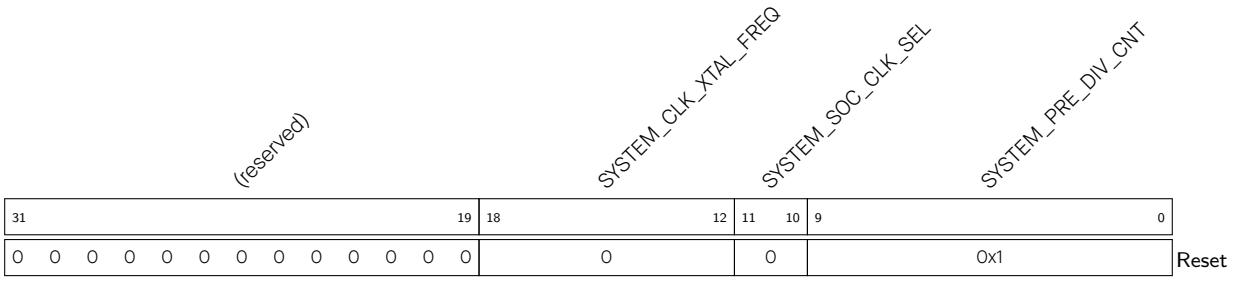
SYSTEM_RTC_MEM_CRC_RES 储存 RTC 存储的 CRC 校验结果。(RO)

Register 17.19. SYSTEM_CLOCK_GATE_REG (0x005C)



SYSTEM_CLK_EN 置 1 使能系统寄存器模块时钟。(R/W)

Register 17.20. SYSTEM_SYSCLK_CONF_REG (0x0060)

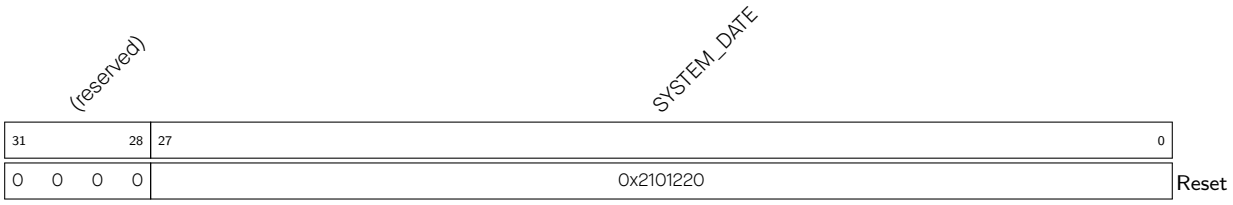


SYSTEM_PRE_DIV_CNT 设置预分频器计数器。具体配置，请见章节 7 复位和时钟 中的表 7.2-3。(R/W)

SYSTEM_SOC_CLK_SEL 选择 SoC 时钟。具体配置，请见章节 7 复位和时钟 中的表 7.2-1。(R/W)

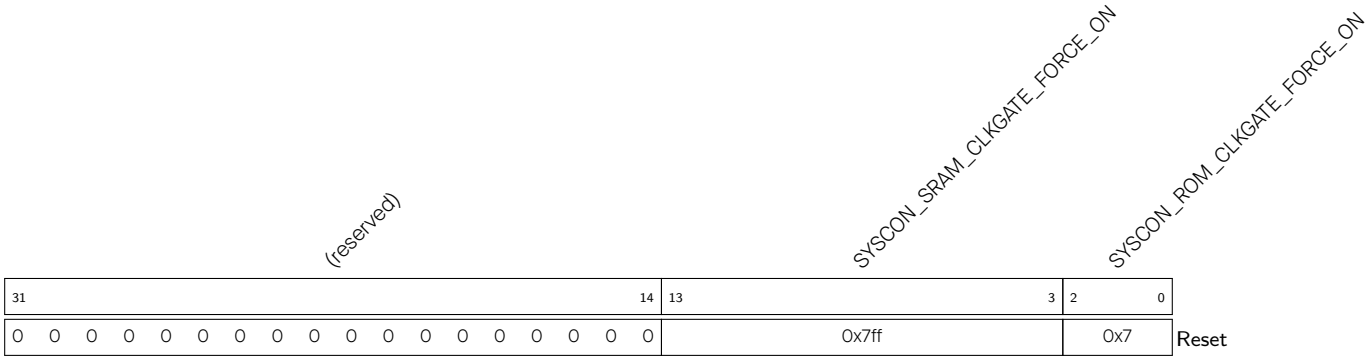
SYSTEM_CLK_XTAL_FREQ 读取晶振频率（单位：MHz）。(RO)

Register 17.21. SYSTEM_DATE_REG (0x0FFC)



SYSTEM_DATE 版本控制寄存器。(R/W)

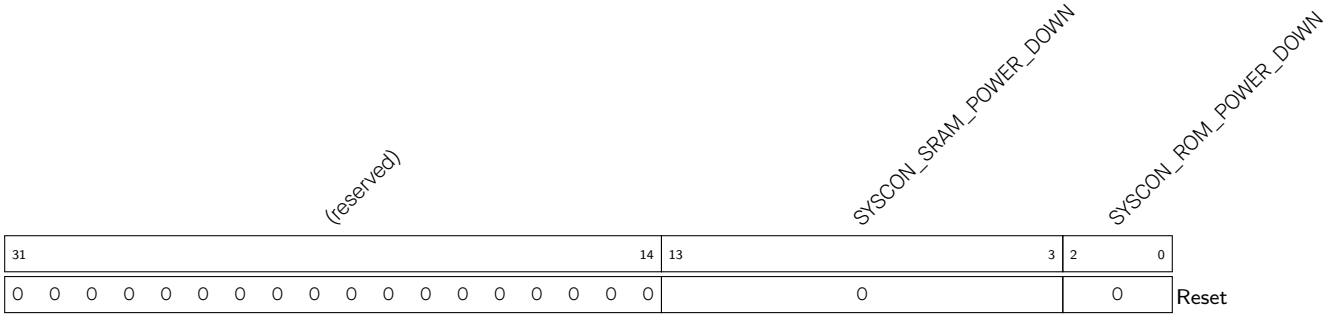
Register 17.22. SYSCON_CLKGATE_FORCE_ON_REG (0x00A8)



SYSCON_ROM_CLKGATE_FORCE_ON 置 1 配置 ROM 内存的时钟门控始终打开；置 0 则配置 ROM 内存的时钟门控在被访问时自动打开，没有访问时自动关闭。(R/W)

SYSCON_SRAM_CLKGATE_FORCE_ON 置 1 配置 SRAM 内存的时钟门控始终打开；置 0 则配置 SRAM 内存的时钟门控在被访问时自动打开，没有访问时自动关闭。(R/W)

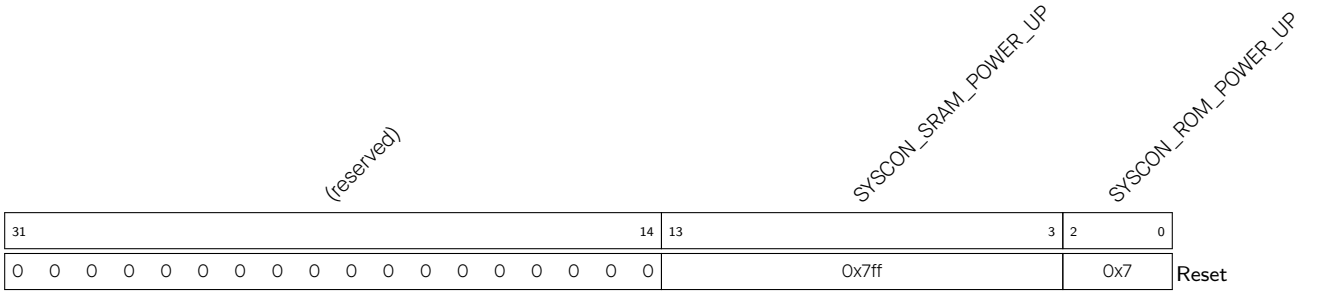
Register 17.23. SYSCON_MEM_POWER_DOWN_REG (0x00AC)



SYSCON_ROM_POWER_DOWN 控制 Internal ROM 进入 Retention 状态。(R/W)

SYSCON_SRAM_POWER_DOWN 控制 Internal SRAM 进入 Retention 状态。(R/W)

Register 17.24. SYSCON_MEM_POWER_UP_REG (0x00B0)



SYSCON_ROM_POWER_UP 控制 Internal ROM 在芯片进入 Light-sleep 时不会进入 Retention 状态。(R/W)

SYSCON_SRAM_POWER_UP 控制 Internal SRAM 在芯片进入 Light-sleep 时不会进入 Retention 状态。(R/W)

第 IV 卷

加密和安全组件

该部分专门介绍芯片的安全功能，包括 SHA、AES 等加密加速器，数字签名、随机数生成、加密/解密算法等安全特性，展示了芯片在加密和安全数据处理方面的性能。

第 18 章

SHA 加速器 (SHA)

18.1 概述

ESP32-S3 内置 SHA（安全哈希算法）硬件加速器可完成 SHA 运算，具有 Typical SHA 和 DMA-SHA 两种工作模式。整体而言，相比基于纯软件的 SHA 运算，SHA 硬件加速器能够极大地提高运算速度。

18.2 主要特性

ESP32-S3 的 SHA 硬件加速器：

- 支持 FIPS PUB 180-4 规范的全部运算标准
  - SHA-1 运算
  - SHA-224 运算
  - SHA-256 运算
  - SHA-384 运算
  - SHA-512 运算
  - SHA-512/224 运算
  - SHA-512/256 运算
  - SHA-512/*t* 运算
- 提供两种工作模式
  - Typical SHA 工作模式
  - DMA-SHA 工作模式
- 允许插入 (interleaved) 功能（仅限 Typical SHA 工作模式）
- 允许中断功能（仅限 DMA-SHA 工作模式）

18.3 工作模式简介

ESP32-S3 内置的 SHA 加速器支持两种工作模式。

- Typical SHA 工作模式：所有数据读写统一通过 CPU 访问完成。
- DMA-SHA 工作模式：所有读数据通过硬件上的 DMA 完成。具体来说，用户可配置 DMA 控制器，由 DMA 控制器提供 SHA 运算过程中所需的数据信息。因此，可以释放 CPU 执行其他任务。

用户可通过配置 SHA_START_REG 或 SHA_DMA_START_REG 选择 SHA 加速器的工作模式，先配置的工作模式生效，具体请见表 18.3-1。

表 18.3-1. 工作模式选择

工作模式	选择方式
Typical SHA	SHA_START_REG 置 1
DMA-SHA	SHA_DMA_START_REG 置 1

用户可通过配置 SHA_MODE_REG 寄存器选择 SHA 加速器的运算标准，具体请见表 18.3-2。

表 18.3-2. 运算标准选择

哈希运算标准	SHA_MODE_REG 的配置
SHA-1	0
SHA-224	1
SHA-256	2
SHA-384	3
SHA-512	4
SHA-512/224	5
SHA-512/256	6
SHA-512/*t*	7

注意：

ESP32-S3 的数字签名 (DS) 和 HMAC 模块也会调用 SHA 加速器。此时，用户无法正常访问 SHA 加速器。

18.4 功能描述

SHA 加速器可以提取信息摘要 (message digest)，其主要工作流程分为两步：信息预处理和哈希运算。

18.4.1 信息预处理

信息预处理分为三个主要步骤：附加填充比特、信息解析和设置初始哈希值。

18.4.1.1 附加填充比特

SHA 加速器仅能处理长度为 512 位及其整倍数或 1024 位及其整倍数的信息。因此，在将信息送至 SHA 加速器进行运算前，应先通过软件操作将信息填充为符合要求的长度。

假设待处理信息 M 的长度为 m 位，则针对不同运算标准的填充步骤见下：

- **SHA-1、SHA-224 和 SHA-256**
  1. 首先，在待处理信息后填充 1 个 “1”；
  2. 随后，再填充 k 个 “0”。其中，k 为满足 $m + 1 + k \equiv 448 \ mod \ 512$ 的最小非负数解；
  3. 最后，在末尾填充一个 64 位的信息块。该信息块的内容为用二进制表示的待处理信息的长度，即 m 的值。

- **SHA-384、SHA-512、SHA-512/224、SHA-512/256** 和 **SHA-512/t**

  1. 首先，在待处理信息后填充 1 个 “1”；
  2. 随后，再填充 k 个 “0”。其中，k 为满足 $m + 1 + k \equiv 896 \ mod \ 1024$ 的最小非负数解；
  3. 最后，在末尾填充一个 128 位的信息块。该信息块的内容为用二进制表示的待处理信息的长度，即 m 的值。

更多详情，请参考 FIPS PUB 180-4 规范 中的 “5.1 Padding the Message” 章节。

18.4.1.2 信息解析

在完成信息填充后，我们还需将待处理信息（及其填充）解析为 N 个 512 位或 1024 位的信息块。

- 对于 **SHA-1、SHA-224** 和 **SHA-256**：待处理信息（及其填充）应解析为 N 个 512 位的信息块，即 $M^{(1)}$、$M^{(2)}$、...、$M^{(N)}$。一个 512 位信息块包括 16 个 32 位的字 (word)，则第 i 个信息块的第一个 32 位字表示为 $M_0^{(i)}$，第二个 32 位字表示为 $M_1^{(i)}$，...，第 16 个 32 位字表示为 $M_{15}^{(i)}$。
- 对于 **SHA-384、SHA-512、SHA-512/224、SHA-512/256** 和 **SHA-512/t**：待处理信息（及其填充）应解析为 N 个 1024 位的信息块，即 $M^{(1)}$、$M^{(2)}$、...、$M^{(N)}$。一个 1024 位信息块包括 16 个 64 位的字 (word)，则第 i 个信息块的第一个 64 位字表示为 $M_0^{(i)}$，第二个 64 位字表示为 $M_1^{(i)}$，...，第 16 个 64 位字表示为 $M_{15}^{(i)}$。

SHA 加速器在工作时，每次处理的信息块数据均将按照如下规则写入相应的寄存器中：

- **SHA-1、SHA-224、SHA-256** 将 $M_0^{(i)}$ 存放在 SHA_M_0_REG 中，$M_1^{(i)}$ 存放在 SHA_M_1_REG，...，$M_{15}^{(i)}$ 存放在 SHA_M_15_REG 中。
- **SHA-384、SHA-512、SHA-512/224、SHA-512/256** 将 $M_0^{(i)}$ 的高 32 位存放在 SHA_M_0_REG 中，低 32 位存放在 SHA_M_1_REG，$M_1^{(i)}$ 的高 32 位存放在 SHA_M_2_REG 中，低 32 位存放在 SHA_M_3_REG，...，$M_{15}^{(i)}$ 的高 32 位存放在 SHA_M_30_REG 中，低 32 位存放在 SHA_M_31_REG。

> **说明：**
> 有关 “信息块” 及相关概念的描述，请参考 FIPS PUB 180-4 规范 中 “2.1 Glossary of Terms and Acronyms” 章节。

18.4.1.3 哈希初始值 (Initial Hash Value)

在进行哈希运算前，首先必须设置哈希初始值 $H^{(0)}$。不同运算标准的哈希初始值设置要求不同，其中 SHA-1、SHA-224、SHA-256、SHA-384、SHA-512、SHA-512/224、SHA-512/256 等运算的哈希初始值为常量 C，且已经固定在硬件中，无需专门计算。

然而，SHA-512/t 对于不同的 t 均需要一个不同的哈希初始值。简单来说，SHA-512/t 是一种基于 SHA-512 的 t 位运算标准，其运算结果将截断至 t 位。其中，运算标准对 t 值的要求为 “大于 0 小于 512 且不等于 384 的正整数”。对于不同 t 值的 SHA-512/t 运算，其哈希初始值可通过对 “SHA-512/t” 字符串的十六进制表示进行 SHA-512 运算获得。不难看出，对于 t 取值不同的 SHA-512/t 运算标准，其不同之处仅在于 t 值不同。

因此，为了简化 SHA-512/t 的哈希初始值计算，我们特别提出了以下方法：

1. **计算 t_string 和 t_length**：其中，t_string 为 t 的字符串信息，长度为 32-bit。t_length 指明字符串长度信息，长度为 7-bit。根据 t 的取值范围不同，t_string 和 t_length 的计算过程如下：

   - 如果 $1 <= t <= 9$，则 $t\_length = 7'h48$，t_string 需要按照如下格式封装：

$8'h30 + 8'ht_0$	$1'b1$	$23'b0$

其中，$t_0 = t$。

举例，如果 t = 8，则 t_0 = 8，t_string = $32'h38800000$。

- 如果 $10 <= t <= 99$，则 $t\_length = 7'h50$，t_string 需要按照如下格式封装:

$8'h30 + 8'ht_1$	$8'h30 + 8'ht_0$	$1'b1$	$15'b0$

其中，$t_0 = t\%10$，$t_1 = t/10$。

举例，如果 t = 56，则 t_0 = 6，t_1 = 5，t_string = $32'h35368000$。

- 如果 $100 <= t < 512$，则 $t\_length = 7'h58$，t_string 需要按照如下格式封装:

$8'h30 + 8'ht_2$	$8'h30 + 8'ht_1$	$8'h30 + 8'ht_0$	$1'b1$	$7'b0$

其中，$t_0 = t\%10$，$t_1 = (t/10)\%10$，$t_2 = t/100$。

举例，如果 t = 231，则 t_0 = 1，t_1 = 3，t_2 = 2，t_string = $32'h32333180$。

2. **配置计算哈希初始值所需的寄存器**：用 t_string 和 t_length 初始化文本寄存器 SHA_T_STRING_REG 和 SHA_T_LENGTH_REG。
3. **计算得到哈希初始值**：对 SHA_MODE_REG 寄存器置 7 选择 SHA-512/t 运算，并对 SHA_START_REG 寄存器置 1，启动 SHA 加速器的运算即可。最后，轮询寄存器 SHA_BUSY_REG 结果为 0，则哈希初始值已计算完毕。

此外，您也可以按照 FIPS PUB 180-4 Spec 中 “5.3.6 SHA-512/t” 章节的描述计算 SHA-512/t 的哈希初始值，也就是对 “SHA-512/t” 字符串的十六进制表示进行一次 “特殊” 的 SHA-512 运算，其运算得到的信息摘要即为所需的哈希初始值。这里的 “特殊” 指本次 SHA-512 运算的哈希初始值为 “SHA-512 运算标准的初始值常量 C 与 0xa5 每 8 位进行一次异或位运算后得到的结果”。

18.4.2 哈希运算流程

在完成信息预处理后，ESP32-S3 SHA 加速器将正式开始哈希运算，最终根据不同运算标准得到不同长度的信息摘要。正如上文所述，ESP32-S3 SHA 加速器支持 Typical SHA 和 DMA-SHA 两种工作模式，下面将对这两种工作模式的具体流程进行介绍。

18.4.2.1 Typical SHA 模式下的运算流程

通常情况下，ESP32-S3 的 SHA 会处理完当前信息的所有信息块并生成该信息的信息摘要，之后再开始计算新的信息摘要。不过，ESP32-S3 SHA 加速器在 Typical SHA 工作模式下还支持 “interleaved” 运算，即在每次运算全部完成前，允许插入其他运算任务。具体来说，在计算完每一个信息块后，用户都可以将存储在 SHA_H_*n*_REG 寄存器中的信息摘要暂存起来，然后插入优先级更高的运算任务，包括 Typical SHA 运算和 DMA-SHA 运算。当临时任务结束后，再将之前暂存的信息摘要重新写入 SHA_H_*n*_REG 中，并继续完成之前中断的计算。

Typical SHA 的具体运算流程（SHA-512/t 除外）

1. 选择运算标准。
   - 配置 SHA_MODE_REG 寄存器，设置运算标准。具体配置，请参考表 18.3-2。

2. 处理当前信息块。
   - 将当前信息块写入 SHA_M_*n*_REG 寄存器。
3. 启动 SHA 加速器 [1]。
   - 如果为首次运算，则对 SHA_START_REG 寄存器置 1，启动 SHA 加速器的运算。此时，SHA 加速器按照步骤 1 中选定的运算标准，使用硬件中固定的哈希初始值进行运算；
   - 如果非首次运算[2]，则对 SHA_CONTINUE_REG 寄存器置 1，启动 SHA 加速器的运算。此时，SHA 加速器使用 SHA_H_*n*_REG 寄存器中的值作为哈希初始值进行运算。
4. 查询当前信息块的处理进度。
   - 轮询寄存器 SHA_BUSY_REG 一直到读回的值为 0，代表 SHA 硬件加速器已完成对当前信息块的计算，进入“空闲”状态[3]。
5. 选择是否有后续的待处理信息块。
   - 如果存在后续待处理信息块，则跳回执行步骤 2。
   - 否则，继续执行。
6. 获取信息摘要：
   - 从寄存器堆 SHA_H_*n*_REG 取出信息摘要。

Typical SHA 的具体运算流程（SHA-512/*t*）

1. 选择运算标准。
   - 配置 SHA_MODE_REG 寄存器为 7 选择 SHA-512/*t* 运算标准。
2. 计算哈希初始值。
   (a) 配置 t_string 和 t_length，并将其写入 SHA_T_STRING_REG 和 SHA_T_LENGTH_REG 寄存器。具体请见 18.4.1.3 章节。
   (b) 对 SHA_START_REG 寄存器置 1，启动 SHA 加速器的运算。
   (c) 轮询寄存器 SHA_BUSY_REG 一直到读回的值为 0，代表 SHA 硬件加速器已完成哈希初始值的计算。
3. 处理当前信息块[1]。
   - 将当前信息块写入 SHA_M_*n*_REG 寄存器。
4. 启动 SHA 加速器。
   - 对 SHA_CONTINUE_REG 寄存器置 1，启动 SHA 加速器的运算。此时，SHA 加速器使用 SHA_H_*n*_REG 寄存器中的值作为哈希初始值进行运算。
5. 查询当前信息块的处理进度。
   - 轮询寄存器 SHA_BUSY_REG 一直到读回的值为 0，代表 SHA 硬件加速器已完成对当前信息块的计算，进入“空闲”状态[3]。
6. 选择是否有后续的待处理信息块。
   - 如果存在后续待处理信息块，则跳回执行步骤 3。
   - 否则，继续执行。

7. 获取信息摘要：
    - 从寄存器堆 SHA_H_*n*_REG 取出信息摘要。

说明：

1. 这里，在 SHA 加速器进行硬件运算时，如果存在后续待处理信息块，软件还可以同时将后续信息块写入 SHA_M_*n*_REG 寄存器，以节省时间。
2. 比如重新启动 SHA 加速器完成之前暂停任务的情况。
3. 这里，你可以选择是否需要插入其他任务。如需插入，请前往 插入任务工作流程 具体查看。

如上文所述，ESP32-S3 SHA 加速器**支持在 Typical SHA 模式下“插入”任务**。

具体工作流程如下。

1. 保存插入前任务的以下数据，准备将 SHA 加速器的使用权移交给插入的任务。
    - 读取并保存寄存器 SHA_MODE_REG 中的运算标准类型。
    - 读取并保存寄存器堆 SHA_H_*n*_REG 中的信息摘要。
2. 执行插入的任务。具体按照插入运行类型的不同，请见 Typical SHA 或 DMA-SHA 工作流程。
3. 恢复插入前任务的以下数据，准备将 SHA 加速器的使用权交还给插入前的任务。
    - 将获得使用权前保存的运算标准类型重新写入寄存器 SHA_MODE_REG;
    - 将获得使用权前保存的信息摘要写入寄存器堆 SHA_H_*n*_REG。
4. 将之前任务的下一个待处理信息块写入 SHA_M_*n*_REG 寄存器，并对 SHA_CONTINUE_REG 寄存器置 1，重新启动 SHA 加速器，完成之前暂停的任务。

18.4.2.2 DMA-SHA 模式下的运算流程

ESP32-S3 SHA 加速器在 DMA-SHA 工作模式下不支持“interleaved”运算方法，即在每次运算全部完成前，不允许插入其他运算任务。这种情况下，用户如有插入运算需求，可将较大信息块进行拆分，并进行多次 DMA-SHA 运算。每次 DMA-SHA 运算之间允许插入其他运算标准的计算任务。

与 Typical SHA 不同，SHA 在 DMA-SHA 工作模式下，运算过程中的数据搬运过程均由硬件完成，具体配置可见章节 3 *通用 DMA 控制器 (GDMA)*。

DMA-SHA 的具体工作流程（SHA-512/*t* 除外）

1. 选择运算标准。
    - 配置 SHA_MODE_REG 寄存器，设置运算标准。具体配置，请参考表 18.3-2。
2. 选择是否启用中断。请将 SHA_INT_ENA_REG 寄存器配置为 1 以启动中断。
3. 配置块个数。
    - 将待加密数据的总块数 *M* 写入 SHA_DMA_BLOCK_NUM_REG 寄存器。
4. 开始 DMA-SHA 运算。
    - 如果当前 DMA-SHA 运算为接着另一次 DMA-SHA 的运算，需要提前将另一次计算得到的信息摘要写入寄存器堆 SHA_H_*n*_REG 中，随后将 1 写入寄存器 SHA_DMA_CONTINUE_REG;
    - 否则，只需要将 1 写入寄存器 SHA_DMA_START_REG。

5. 等待 DMA-SHA 运算结束。判断 DMA-SHA 运算结束有以下两种方法：
   - 轮询寄存器 SHA_BUSY_REG 结果为 0。
   - 等待中断信号产生。此时，应及时通过软件将 SHA_INT_CLEAR_REG 寄存器置为 1 以清除中断。
6. 获取信息摘要
   - 从寄存器堆 SHA_H_*n*_REG 取出信息摘要。

DMA-SHA 的具体工作流程（SHA-512/*t*）

1. 选择运算标准。
   - 配置 SHA_MODE_REG 寄存器为 7 选择 SHA-512/*t* 运算标准。
2. 选择是否启用中断。请将 SHA_INT_ENA_REG 寄存器配置为 1 以启动中断。
3. 计算哈希初始值。
   (a) 配置 t_string 和 t_length，并将其写入 SHA_T_STRING_REG 和 SHA_T_LENGTH_REG 寄存器。具体请见 18.4.1.3 章节。
   (b) 对 SHA_START_REG 寄存器置 1，启动 SHA 加速器的运算。
   (c) 轮询寄存器 SHA_BUSY_REG 一直到读回的值为 0，代表 SHA 硬件加速器已完成哈希初始值的计算。
4. 配置块个数。
   - 将待加密数据的总块数 M 写入 SHA_DMA_BLOCK_NUM_REG 寄存器。
5. 开始 DMA-SHA 运算。
   - 对 SHA_DMA_CONTINUE_REG 置 1 启动 SHA 加速器。
6. 等待 DMA-SHA 运算结束。判断 DMA-SHA 运算结束有以下两种方法：
   - 轮询寄存器 SHA_BUSY_REG 结果为 0。
   - 等待中断信号产生。此时，应及时通过软件将 SHA_INT_CLEAR_REG 寄存器置为 1 以清除中断。
7. 获取信息摘要
   - 从寄存器堆 SHA_H_*n*_REG 取出信息摘要。

18.4.3 信息摘要存储

哈希运算完成之后，计算得到的信息摘要被 SHA 加速器更新至对应的 SHA_H_*n*_REG (*n*: 0~15) 寄存器中。不同运算标准得到的信息摘要长度也不同，详情见表 18.4-4：

表 18.4-4. 不同运算标准信息摘要的寄存器占用情况

哈希运算标准	信息摘要长度（位）	寄存器占用情况[1]
SHA-1	160	SHA_H_0_REG ~ SHA_H_4_REG
SHA-224	224	SHA_H_0_REG ~ SHA_H_6_REG
SHA-256	256	SHA_H_0_REG ~ SHA_H_7_REG
SHA-384	384	SHA_H_0_REG ~ SHA_H_11_REG
SHA-512	512	SHA_H_0_REG ~ SHA_H_15_REG
SHA-512/224	224	SHA_H_0_REG ~ SHA_H_6_REG
SHA-512/256	256	SHA_H_0_REG ~ SHA_H_7_REG
SHA-512/*t*[2]	*t*	SHA_H_0_REG ~ SHA_H_*x*_REG

[1] 信息摘要从左至右存放，第一个 word 存放在寄存器 SHA_H_0_REG 中，第二个 word 存放在寄存器 SHA_H_1_REG 中，以此类推。

[2] SHA-512/*t* 运算标准使用的寄存器与 *t* 的取值有关。*x*+1 代表用于存储 *t* 位信息摘要的 32 位寄存器个数，因此 *x* = $roundup(t/32)$-1。举例：

- 当 *t* = 8 时，则 *x* = 0，代表最终的信息摘要长度为 8 位，存放在寄存器 SHA_H_0_REG 的高 8 位中；
- 当 *t* = 32 时，则 *x* = 0，代表最终的信息摘要长度为 32 位，存放在寄存器 SHA_H_0_REG 中；
- 当 *t* = 132 时，则 *x* = 4，代表最终的信息摘要长度为 132 位，存放在寄存器 SHA_H_0_REG、SHA_H_1_REG、SHA_H_2_REG、SHA_H_3_REG，及 SHA_H_4_REG 中。

18.4.4 中断

SHA 加速器在 DMA-SHA 工作模式下允许中断发生。用户可通过将 SHA_INT_ENA_REG 寄存器配置为 1 开启中断。如开启中断功能，SHA 加速器在完成运算时，中断发生。注意，该中断必须由软件将 SHA_INT_CLEAR_REG 寄存器置为 1 进行清除。由于 SHA 加速器在 Typical SHA 工作模式下的时间开销较小，因此不支持中断功能。

18.5 寄存器列表

本小节的所有地址均为相对于 SHA 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	权限
控制与状态寄存器			
SHA_CONTINUE_REG	继续 SHA 运算（仅用于 Typical SHA 模式）	0x0014	WO
SHA_BUSY_REG	指示 SHA 加速器是否处于“忙碌”状态	0x0018	RO
SHA_DMA_START_REG	启动 SHA 加速器的 DMA-SHA 模式	0x001C	WO
SHA_START_REG	启动 SHA 加速器的 Typical SHA 模式	0x0010	WO
SHA_DMA_CONTINUE_REG	继续 SHA 运算（仅用于 DMA-SHA 模式）	0x0020	WO
SHA_INT_CLEAR_REG	DMA-SHA 中断清除寄存器	0x0024	WO

名称	描述	地址	权限
SHA_INT_ENA_REG	DMA-SHA 中断使能寄存器	0x0028	R/W
版本寄存器			
SHA_DATE_REG	版本控制寄存器	0x002C	R/W
配置寄存器			
SHA_MODE_REG	配置 SHA 加速器的运算标准	0x0000	R/W
SHA_T_STRING_REG	哈希字符串内容寄存器（仅用于计算 SHA-512/t 的哈希初始值）	0x0004	R/W
SHA_T_LENGTH_REG	哈希字符串长度寄存器（仅用于计算 SHA-512/t 的哈希初始值）	0x0008	R/W
存储器			
SHA_DMA_BLOCK_NUM_REG	信息块个数寄存器(仅用于 DMA-SHA 工作模式)	0x000C	R/W
SHA_H_0_REG	哈希值	0x0040	R/W
SHA_H_1_REG	哈希值	0x0044	R/W
SHA_H_2_REG	哈希值	0x0048	R/W
SHA_H_3_REG	哈希值	0x004C	R/W
SHA_H_4_REG	哈希值	0x0050	R/W
SHA_H_5_REG	哈希值	0x0054	R/W
SHA_H_6_REG	哈希值	0x0058	R/W
SHA_H_7_REG	哈希值	0x005C	R/W
SHA_H_8_REG	哈希值	0x0060	R/W
SHA_H_9_REG	哈希值	0x0064	R/W
SHA_H_10_REG	哈希值	0x0068	R/W
SHA_H_11_REG	哈希值	0x006C	R/W
SHA_H_12_REG	哈希值	0x0070	R/W
SHA_H_13_REG	哈希值	0x0074	R/W
SHA_H_14_REG	哈希值	0x0078	R/W
SHA_H_15_REG	哈希值	0x007C	R/W
SHA_M_0_REG	输入信息	0x0080	R/W
SHA_M_1_REG	输入信息	0x0084	R/W
SHA_M_2_REG	输入信息	0x0088	R/W
SHA_M_3_REG	输入信息	0x008C	R/W
SHA_M_4_REG	输入信息	0x0090	R/W
SHA_M_5_REG	输入信息	0x0094	R/W
SHA_M_6_REG	输入信息	0x0098	R/W
SHA_M_7_REG	输入信息	0x009C	R/W
SHA_M_8_REG	输入信息	0x00A0	R/W
SHA_M_9_REG	输入信息	0x00A4	R/W
SHA_M_10_REG	输入信息	0x00A8	R/W
SHA_M_11_REG	输入信息	0x00AC	R/W
SHA_M_12_REG	输入信息	0x00B0	R/W
SHA_M_13_REG	输入信息	0x00B4	R/W
SHA_M_14_REG	输入信息	0x00B8	R/W
SHA_M_15_REG	输入信息	0x00BC	R/W

名称	描述	地址	权限
SHA_M_16_REG	输入信息	0x00C0	R/W
SHA_M_17_REG	输入信息	0x00C4	R/W
SHA_M_18_REG	输入信息	0x00C8	R/W
SHA_M_19_REG	输入信息	0x00CC	R/W
SHA_M_20_REG	输入信息	0x00D0	R/W
SHA_M_21_REG	输入信息	0x00D4	R/W
SHA_M_22_REG	输入信息	0x00D8	R/W
SHA_M_23_REG	输入信息	0x00DC	R/W
SHA_M_24_REG	输入信息	0x00E0	R/W
SHA_M_25_REG	输入信息	0x00E4	R/W
SHA_M_26_REG	输入信息	0x00E8	R/W
SHA_M_27_REG	输入信息	0x00EC	R/W
SHA_M_28_REG	输入信息	0x00F0	R/W
SHA_M_29_REG	输入信息	0x00F4	R/W
SHA_M_30_REG	输入信息	0x00F8	R/W
SHA_M_31_REG	输入信息	0x00FC	R/W

18.6 寄存器

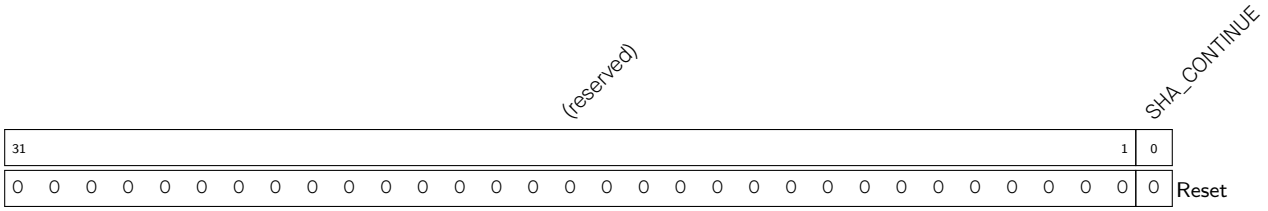
本小节的所有地址均为相对于 SHA 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 18.1. SHA_START_REG (0x0010)



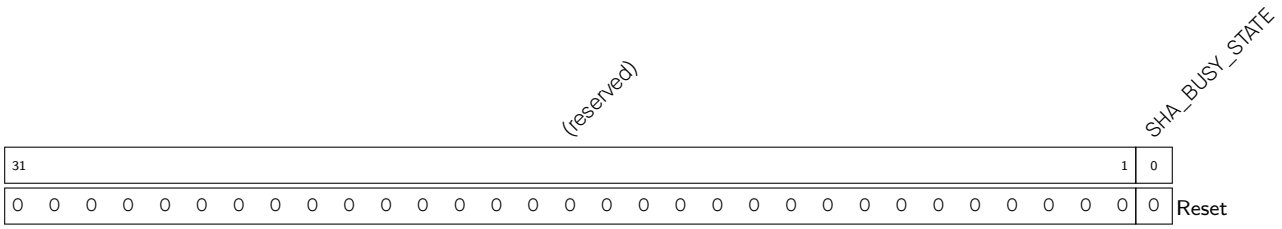
SHA_START 置 1 启动 SHA 加速器的 Typical SHA 模式。(WO)

Register 18.2. SHA_CONTINUE_REG (0x0014)



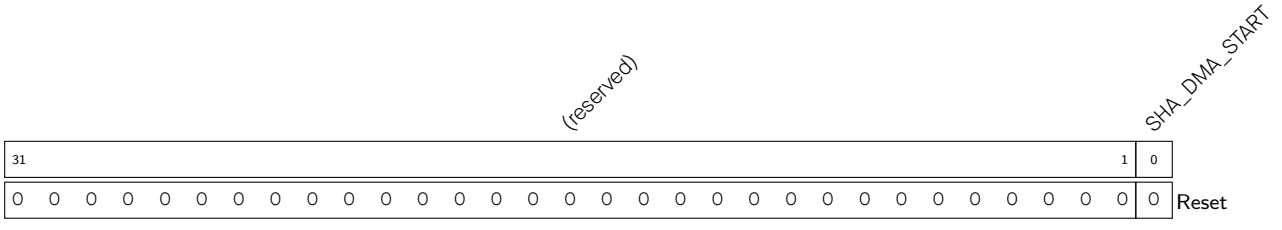
SHA_CONTINUE 置 1 继续 SHA 加速器的 Typical SHA 运算。(WO)

Register 18.3. SHA_BUSY_REG (0x0018)



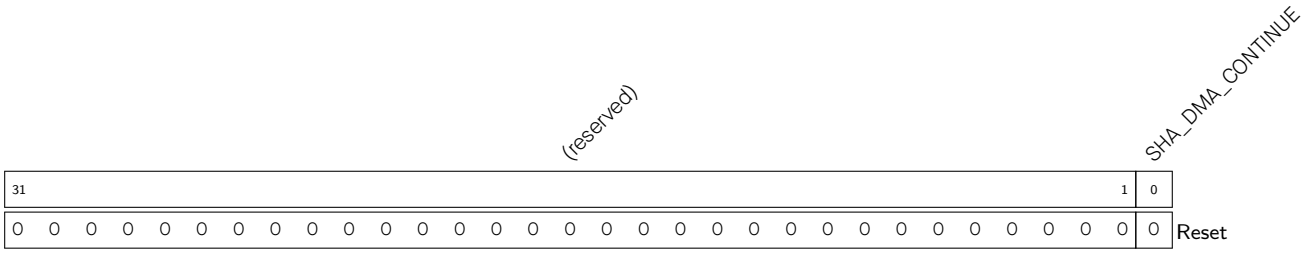
SHA_BUSY_STATE 指示 SHA 是否处于“忙碌”状态。(RO) 1'h0: 空闲 1'h1: 忙碌

Register 18.4. SHA_DMA_START_REG (0x001C)



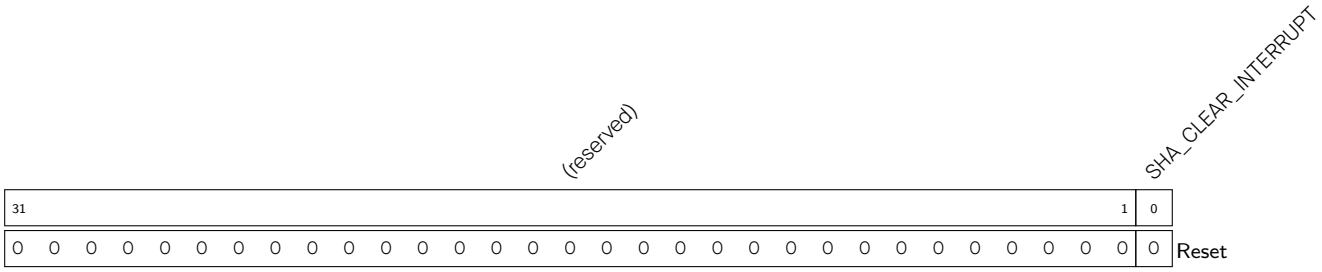
SHA_DMA_START 置 1 启动 SHA 加速器的 DMA-SHA 模式。(WO)

Register 18.5. SHA_DMA_CONTINUE_REG (0x0020)



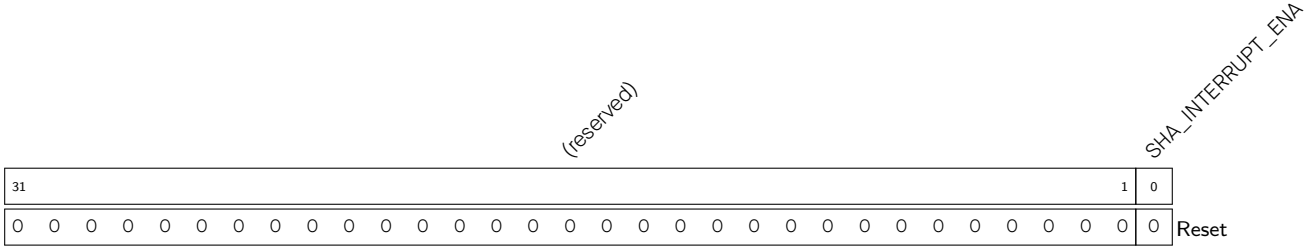
SHA_DMA_CONTINUE 置 1 继续 SHA 加速器的 DMA-SHA 运算。(WO)

Register 18.6. SHA_INT_CLEAR_REG (0x0024)



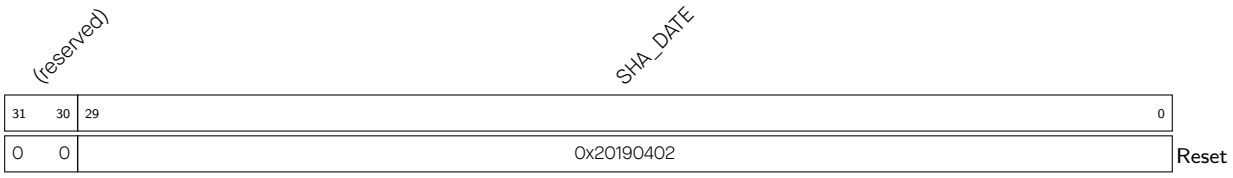
SHA_CLEAR_INTERRUPT 清除 DMA-SHA 中断。(WO)

Register 18.7. SHA_INT_ENA_REG (0x0028)



SHA_INTERRUPT_ENA 使能 DMA-SHA 中断。(R/W)

Register 18.8. SHA_DATE_REG (0x002C)



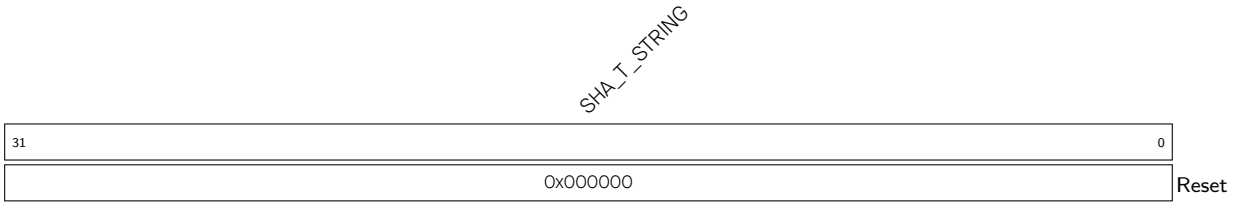
SHA_DATE 版本控制寄存器。(R/W)

Register 18.9. SHA_MODE_REG (0x0000)



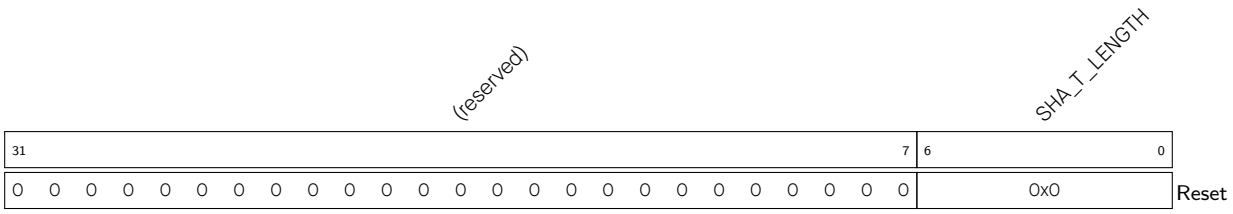
SHA_MODE 选择 SHA 加速器的运算标准，详见表 18.3-2。(R/W)

Register 18.10. SHA_T_STRING_REG (0x0004)



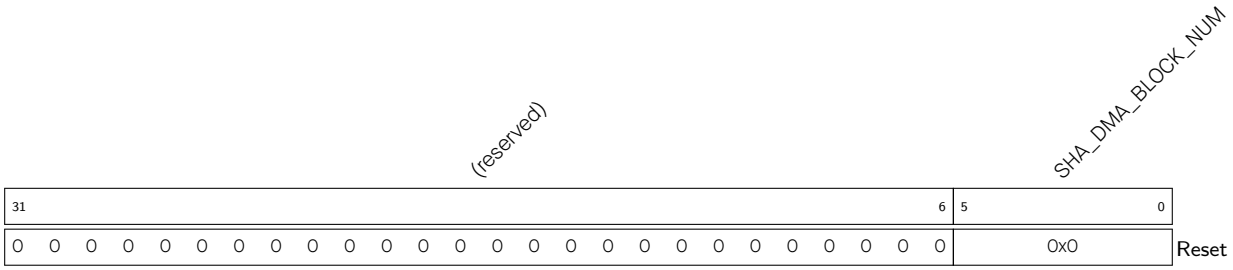
SHA_T_STRING 存储哈希字符串内容（仅用于计算 SHA-512/t 的哈希初始值）。(R/W)

Register 18.11. SHA_T_LENGTH_REG (0x0008)



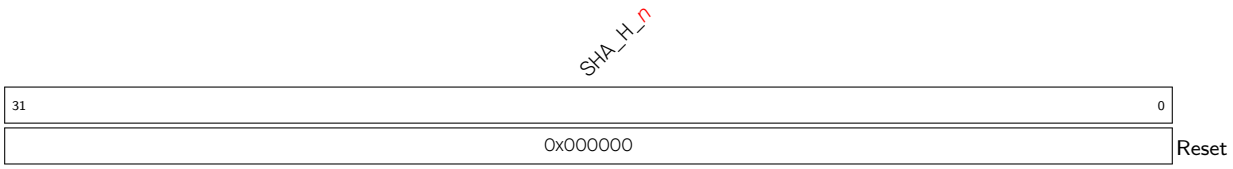
SHA_T_LENGTH 存储哈希字符串长度（仅用于计算 SHA-512/t 的哈希初始值）。(R/W)

Register 18.12. SHA_DMA_BLOCK_NUM_REG (0x000C)



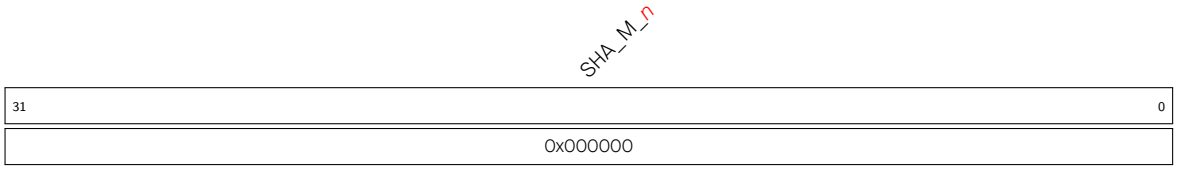
SHA_DMA_BLOCK_NUM 定义 DMA-SHA 工作模式下的信息块个数。(R/W)

Register 18.13. SHA_H_*n*_REG (*n*: 0-15) (0x0040+4**n*)



SHA_H_*n* 存储第 *n* 个 32 位哈希值。(R/W)

Register 18.14. SHA_M_*n*_REG (*n*: 0-31) (0x0080+4**n*)



SHA_M_*n* 存储第 *n* 个 32 位输入信息。(R/W)

第 19 章

AES 加速器 (AES)

19.1 概述

ESP32-S3 内置 AES（高级加密标准）硬件加速器可使用 AES 算法，完成数据的加解密运算，具有 Typical AES 和 DMA-AES 两种工作模式。整体而言，相比基于纯软件的 AES 运算，AES 硬件加速器能够极大地提高运算速度。

19.2 主要特性

ESP32-S3 支持以下特性：

- Typical AES 工作模式
  - AES-128/AES-256 加解密运算
- DMA-AES 工作模式
  - AES-128/AES-256 加解密运算
  - 块（加密）模式
    - ECB (Electronic Codebook)
    - CBC (Cipher Block Chaining)
    - OFB (Output Feedback)
    - CTR (Counter)
    - CFB8 (8-bit Cipher Feedback)
    - CFB128 (128-bit Cipher Feedback)
  - 中断发生

19.3 工作模式简介

ESP32-S3 内置的 AES 加速器支持 Typical AES 和 DMA-AES 两种工作模式。

- Typical AES 工作模式：
  - 支持使用 128 位或 256 位密钥进行加密与解密运算，即 NIST FIPS 197 标准中的 AES-128 和 AES-256 加解密运算。

  这种情况下，明文/密文的读/写操作统一通过 CPU 访问完成。
- DMA-AES 工作模式：

- 支持使用 128 位或 256 位密钥进行加密与解密运算，即 NIST FIPS 197 标准中的 AES-128 和 AES-256 加解密运算；
- 还支持 NIST SP 800-38A 标准中的 ECB/CBC/OFB/CTR/CFB8/CFB128 等块加密模式运算。

在这种情况下，明文/密文的传输通过硬件上的 DMA 完成，计算完成时会有中断发生。

用户可通过配置 AES_DMA_ENABLE_REG 选择 AES 加速器的工作模式，具体参考表 19.3-1。

表 19.3-1. 工作模式

AES_DMA_ENABLE_REG	工作模式
0	Typical AES
1	DMA-AES

用户可通过配置 AES_MODE_REG 寄存器选择密钥长度和加解密方向，具体可参考表 19.3-2。

表 19.3-2. 密钥长度和加解密方向

AES_MODE_REG[2:0]	密钥长度和加解密方向
0	AES-128 加密
1	保留
2	AES-256 加密
3	保留
4	AES-128 解密
5	保留
6	AES-256 解密
7	保留

有关 Typical AES 和 DMA-AES 两种工作模式的具体介绍，请见下方 19.4 章节和 19.5 章节。

注意：

ESP32-S3 的数字签名 (DS)模块也会调用 AES 加速器。此时，用户无法正常访问 AES 加速器。

19.4 Typical AES 工作模式

在 Typical AES 工作模式下，AES 加速器的状态值可查看寄存器 AES_STATE_REG，具体见表 19.4-1 所示：

表 19.4-1. 状态返回值

返回值	描述	状态说明
0	IDLE	加速器空闲或计算完成
1	WORK	加速器忙于计算

19.4.1 密钥、明文、密文

寄存器 AES_KEY_*n*_REG 用于存放密钥，由 8 个 32 位寄存器组成。

- 如果为 AES-128 加解密运算，则 128 位密钥在寄存器 AES_KEY_0_REG ~ AES_KEY_3_REG 中。
- 如果为 AES-256 加解密运算，则 256 位密钥在寄存器 AES_KEY_0_REG ~ AES_KEY_7_REG 中。

寄存器 AES_TEXT_IN_*m*_REG 和 AES_TEXT_OUT_*m*_REG 用于存放明文和密文，各由 4 个 32 位寄存器组成。

- 如果为 AES-128/265 加密运算，则运算开始之前用明文初始化寄存器 AES_TEXT_IN_*m*_REG。运算完成之后，AES 加速器将把密文更新入寄存器 AES_TEXT_OUT_*m*_REG。
- 如果为 AES-128/256 解密运算，则运算开始之前用密文初始化寄存器 AES_TEXT_IN_*m*_REG。运算完成之后，AES 加速器将把明文更新入寄存器 AES_TEXT_OUT_*m*_REG。

19.4.2 字节序

文本字节序

在 Typical AES 工作模式下，AES 加速器可以使用密钥对 128 位的 block 进行加解密。在操作寄存器 AES_TEXT_IN_*m*_REG 和 AES_TEXT_OUT_*m*_REG 中的数据时，用户应遵循表 19.4-2 中定义的文本字节序。

表 19.4-2. Typical AES 文本字节序

明文/密文					
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] 有关 “State（以及 c 和 r）” 的详细定义，请参考 NIST FIPS 197 中 “3.4 The State” 章节。
[2] 其中，*x* = IN 或 OUT。

密钥字节序

在 Typical AES 工作模式下，在向寄存器 AES_KEY_*n*_REG 中填入数据时，用户应遵循表 19.4-3 和表 19.4-4 中定义的文本字节序。

表 19.4-3. AES-128 密钥字节序

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] Bit 列代表 w[0] ~ w[3] 每个 word 中的各个字节。
[2] w[0] ~ w[3] 符合标准 NIST FIPS 197 中 “5.2 Key Expansion” 章节中对 “the first Nk words of the expanded key” 的描述。

表 19.4-4. AES-256 密钥字节序

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] Bit 列代表 w[0] ~ w[7] 每个 word 中的各个字节。

[2] w[0] ~ w[7] 符合标准 NIST FIPS 197 中 “5.2 Key Expansion” 章节中对 “the first Nk words of the expanded key” 的描述。

19.4.3 Typical AES 工作模式的流程

单次运算

1. 对寄存器 AES_DMA_ENABLE_REG 写入 0。
2. 初始化寄存器 AES_MODE_REG、AES_KEY_*n*_REG、AES_TEXT_IN_*m*_REG。
3. 启动运算。对寄存器 AES_TRIGGER_REG 写入 1。
4. 等待运算完成。轮询寄存器 AES_STATE_REG，直到读到 0。
5. 从寄存器 AES_TEXT_OUT_*m*_REG 读取结果。

连续运算

在连续运算过程中，每次运算完成之后，只有寄存器 AES_TEXT_IN_*m*_REG 和 AES_TEXT_OUT_*m*_REG (*m*: 0-3) 会被 AES 加速器更新，而 AES_DMA_ENABLE_REG、AES_MODE_REG、AES_KEY_*n*_REG 等寄存器中的内容不会变化。所以进行连续运算时可以简化初始化操作。

1. 第一次运算之前对寄存器 AES_DMA_ENABLE_REG 写入 0。
2. 第一次运算之前初始化寄存器 AES_MODE_REG 和 AES_KEY_*n*_REG。
3. 更新寄存器 AES_TEXT_IN_*m*_REG。
4. 启动运算。对寄存器 AES_TRIGGER_REG 写入 1。
5. 等待运算完成。轮询寄存器 AES_STATE_REG，直到读到 0。
6. 从寄存器 AES_TEXT_OUT_*m*_REG 读取结果 。返回步骤 3，进行下一轮运算。

19.5 DMA-AES 工作模式

在 DMA-AES 工作模式下，AES 加速器可支持 ECB/CBC/OFB/CTR/CFB8/CFB128 等 6 种块模式运算。用户可通过配置 AES_BLOCK_MODE_REG 寄存器选择具体运算类型，具体可参考表 19.5-1。

表 19.5-1. 块模式选择

AES_BLOCK_MODE_REG[2:0]	块模式
0	ECB (Electronic Code Book)
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	保留
7	保留

AES 加速器的状态值可查看寄存器 AES_STATE_REG，具体见表 19.5-2 所示：

表 19.5-2. 状态返回值

返回值	描述	状态说明
0	IDLE	加速器空闲
1	WORK	加速器忙于计算
2	DONE	加速器计算完成

AES 加速器在 DMA-AES 工作模式下允许中断发生，软件清零。中断功能默认关闭，用户可通过将 AES_INT_ENA_REG 寄存器配置为 1 开启中断。如开启中断功能，AES 加速器在完成计算时，中断发生。

19.5.1 密钥、明文、密文

块运算模式

在块运算模式下，AES 加速器的源数据来自 DMA，结果数据也将被写入 DMA。

- 如果为加密运算，则 DMA 从 memory 中读取明文数据流并将其传给 AES。AES 计算出密文后将密文写入 DMA。DMA 再将密文写入 memory。
- 如果为解密运算，则 DMA 从 memory 中读取密文数据流并将其传给 AES。AES 计算出明文后将明文写入 DMA。DMA 再将明文写入 memory。

AES 加速器在进行块运算时，结果数据与源数据的大小保持一致。此时，DMA 的数据搬运过程和 AES 的计算过程有所交叠，因此总工作时间有所减少。

值得注意的是，AES 加速器在 DMA-AES 工作模式下要求源数据的大小必须是 128 位的整数倍，否则需要将原始明文封装为 128 位的整数倍，即在原比特串 (bit string) 尾部尽可能少的补 “0”，具体过程见表 19.5-3 所示。

表 19.5-3. TEXT-PADDING

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

19.5.2 字节序

在 DMA-AES 工作模式下，源数据和结果数据的传输完全由 DMA 完成，因此不支持字节序的控制调节，但要求它们在 memory 中以一定的方式来存放，且要求数据量必须是 block 的整数倍。

举例说明，假设 DMA 需要搬运 2 个 block 大小的源数据：

- 十六进制：0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F20

假设起始地址为 0x0280，则源数据在 memory 中的存放位置如表 19.5-4 所示。结果数据也遵从相同的存放规则，在此不多做介绍。

表 19.5-4. DMA AES 存储字节序

地址	字节	地址	字节	地址	字节	地址	字节
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

另外，值得注意的是，DMA 既可以访问片内存储空间，又可以访问片外 PSRAM。当访问片外 PSRAM 时，基地址必须满足 DMA 对地址的相关要求，但当访问片内存储器空间时则没有限制。详情请见章节 3 *通用 DMA 控制器 (GDMA)*。

19.5.3 标准增量函数

AES 加速器在进行 CTR 块运算时，还可提供两种标准增量函数供用户选择：INC_{32} 和 INC_{128} 。用户可通过将寄存器 AES_INC_SEL_REG 置为 0 或 1 选择 INC_{32} 或 INC_{128} 标准增量函数。更多有关标准增量函数的内容，请见 NIST SP 800-38A 标准中的“B.1 The Standard Incrementing Function”章节。

19.5.4 块个数

寄存器 AES_BLOCK_NUM_REG 存放明文或密文的块个数 (Block Number)，其值等于 length(**TEXT-PADDING**(P))/128，也等于 length(**TEXT-PADDING**(C))/128。这里的 P 指明文 (plaintext)，C 指密文 (ciphertext)。该寄存器仅在 DMA-AES 工作模式下有意义。

19.5.5 初始向量

存储器 AES_IV_MEM 的空间大小为 16 字节，仅在块运算模式下有效。对于 CBC/OFB/CFB8/CFB128 等操作，AES_IV_MEM 用于存放初始向量 (Initialization Vector, IV) 的值。对于 CTR 操作，AES_IV_MEM 存放初始计数器 (Initial Counter Block, ICB) 的值。

IV 和 ICB 都是 128-bit 长的比特串，从左向右被分割成 16 个字节 (Byte0, Byte1, Byte2, ⋯, Byte15)，构成一个字节序列，在 AES_IV_MEM 中存放时需要遵循表 19.5-4 中的字节序规则，即 Byte0 存放在 AES_IV_MEM 中的最低地址中，Byte15 存放在 AES_IV_MEM 中的最高地址中。

更多有关 IV 和 ICB 的信息，请参考 NIST SP 800-38A 标准。

19.5.6 DMA-AES 工作模式的流程

1. 选择一条 DMA 通道与 AES 加速器连接，配置 DMA 链表，而后启动 DMA。详情请见章节 3 *通用 DMA 控制器 (GDMA)*。
2. 配置 AES：
   - 对寄存器 AES_DMA_ENABLE_REG 写入 1。
   - 选择是否开启中断。根据需要设置寄存器 AES_INT_ENA_REG 的值。
   - 初始化 AES_MODE_REG 和 AES_KEY_*n*_REG 寄存器。
   - 配置 AES_BLOCK_MODE_REG 寄存器，选择具体块加密模式。详见表 19.5-1。
   - 初始化寄存器 AES_BLOCK_NUM_REG，请参照章节 19.5.4。
   - 初始化寄存器 AES_INC_SEL_REG（仅在 CTR 块模式下使用）。
   - 初始化存储器 AES_IV_MEM（在 ECB 块模式下不使用）。
3. 启动运算。对寄存器 AES_TRIGGER_REG 写入 1。
4. 等待运算完成。轮询寄存器 AES_STATE_REG，直到读到 2。如果开启了中断功能，也可以等待 AES_INT 中断产生。
5. 确认 DMA 完成从 AES 到内存的数据传输。此时，结果数据已经被 DMA 写入 memory，可以直接从中读取。详情请参考章节 3 *通用 DMA 控制器 (GDMA)*。
6. 如果开启了中断，当处理中断程序完成后，请及时对寄存器 AES_INT_CLR_REG 写 1 以清除中断。
7. 对寄存器 AES_DMA_EXIT_REG 写入 1 释放 AES 加速器。之后如果再读取寄存器 AES_STATE_REG 将读到 0。该步操作可以提前完成，但必须在步骤 4 之后。

19.6 存储器列表

本小节的所有地址均为相对于 AES 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

名称	描述	大小（比特）	起始地址	结束地址	访问权限
AES_IV_MEM	存储器 IV	16 字节	0x0050	0x005F	读 / 写

19.7 寄存器列表

本小节的所有地址均为相对于 AES 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

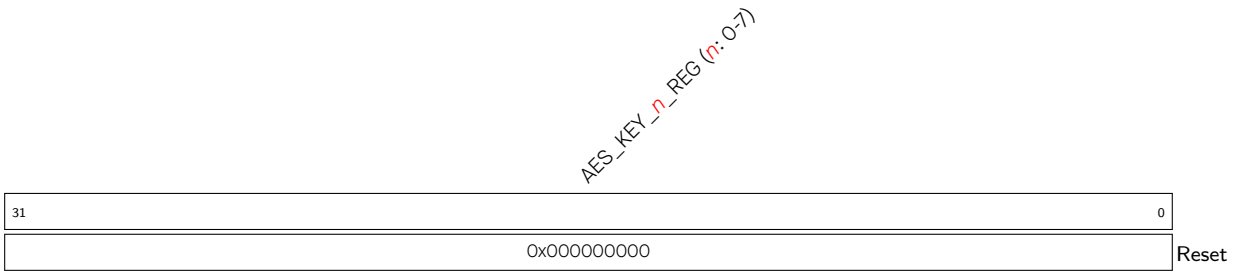
请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
密钥寄存器			
AES_KEY_0_REG	AES 密钥寄存器 0	0x0000	读 / 写
AES_KEY_1_REG	AES 密钥寄存器 1	0x0004	读 / 写
AES_KEY_2_REG	AES 密钥寄存器 2	0x0008	读 / 写
AES_KEY_3_REG	AES 密钥寄存器 3	0x000C	读 / 写
AES_KEY_4_REG	AES 密钥寄存器 4	0x0010	读 / 写
AES_KEY_5_REG	AES 密钥寄存器 5	0x0014	读 / 写
AES_KEY_6_REG	AES 密钥寄存器 6	0x0018	读 / 写
AES_KEY_7_REG	AES 密钥寄存器 7	0x001C	读 / 写
TEXT_IN 寄存器			
AES_TEXT_IN_0_REG	源数据寄存器 0	0x0020	读 / 写
AES_TEXT_IN_1_REG	源数据寄存器 1	0x0024	读 / 写
AES_TEXT_IN_2_REG	源数据寄存器 2	0x0028	读 / 写
AES_TEXT_IN_3_REG	源数据寄存器 3	0x002C	读 / 写
TEXT_OUT 寄存器			
AES_TEXT_OUT_0_REG	结果数据寄存器 0	0x0030	只读
AES_TEXT_OUT_1_REG	结果数据寄存器 1	0x0034	只读
AES_TEXT_OUT_2_REG	结果数据寄存器 2	0x0038	只读
AES_TEXT_OUT_3_REG	结果数据寄存器 3	0x003C	只读
配置寄存器			
AES_MODE_REG	选择密钥长度和加解密方向	0x0040	读 / 写
AES_DMA_ENABLE_REG	选择 AES 加速器工作模式	0x0090	读 / 写
AES_BLOCK_MODE_REG	选择 DMA-AES 下的块运算模式	0x0094	读 / 写
AES_BLOCK_NUM_REG	块数量配置寄存器	0x0098	读 / 写
AES_INC_SEL_REG	标准增量函数选择寄存器	0x009C	读 / 写
控制 / 状态寄存器			
AES_TRIGGER_REG	开始运算寄存器	0x0048	只写
AES_STATE_REG	运算状态寄存器	0x004C	只读
AES_DMA_EXIT_REG	退出运算寄存器	0x00B8	只写
中断寄存器			
AES_INT_CLR_REG	DMA-AES 中断清除	0x00AC	只写
AES_INT_ENA_REG	DMA-AES 中断使能寄存器	0x00B0	读 / 写

19.8 寄存器

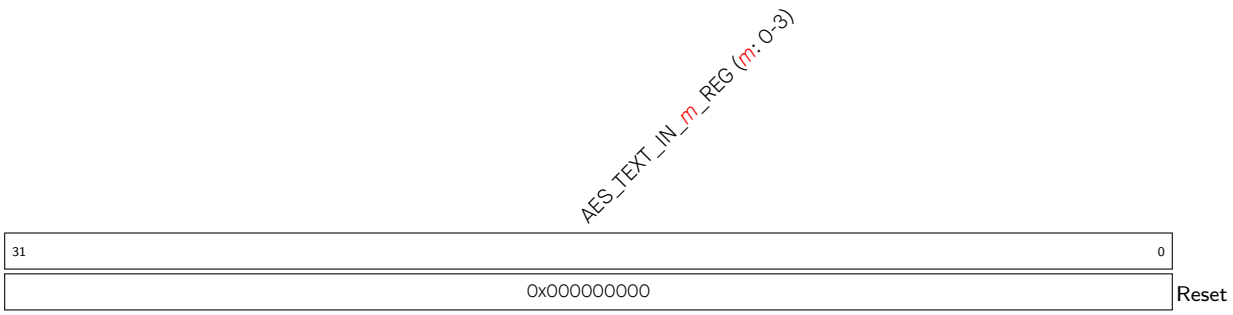
本小节的所有地址均为相对于 AES 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 19.1. AES_KEY_*n*_REG (*n*: 0-7) (0x0000+4**n*)



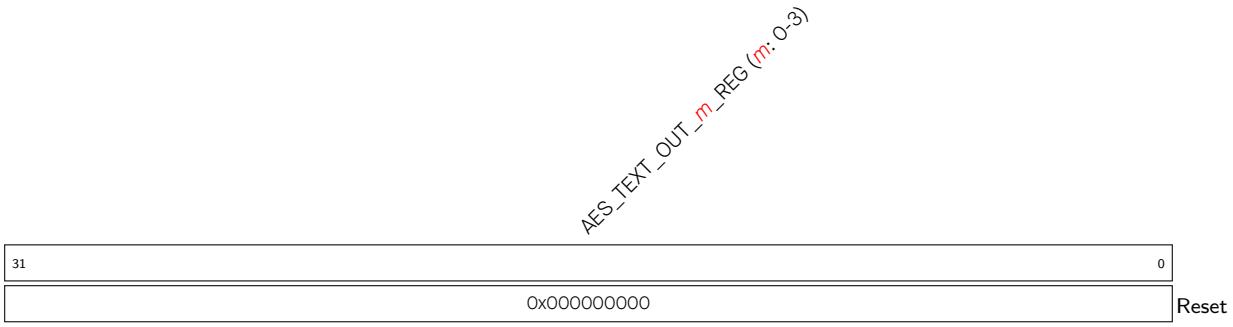
AES_KEY_*n*_REG (*n*: 0-7) AES 密钥寄存器。(R/W)

Register 19.2. AES_TEXT_IN_*m*_REG (*m*: 0-3) (0x0020+4**m*)



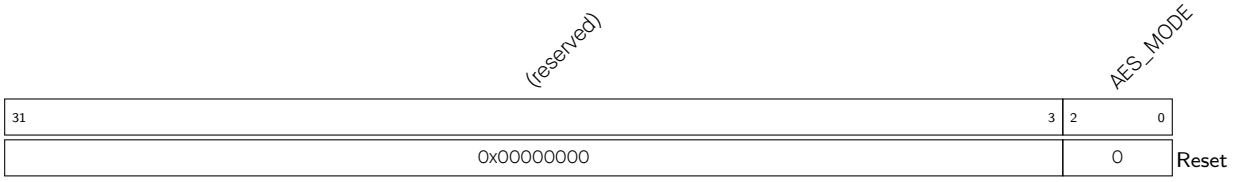
AES_TEXT_IN_*m*_REG (*m*: 0-3) Typical AES 文本输入寄存器。(R/W)

Register 19.3. AES_TEXT_OUT_*m*_REG (*m*: 0-3) (0x0030+4**m*)



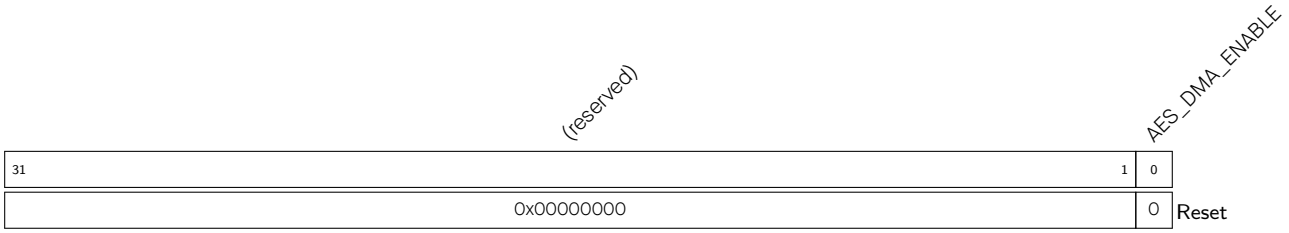
AES_TEXT_OUT_*m*_REG (*m*: 0-3) Typical AES 文本输出寄存器。(RO)

Register 19.4. AES_MODE_REG (0x0040)



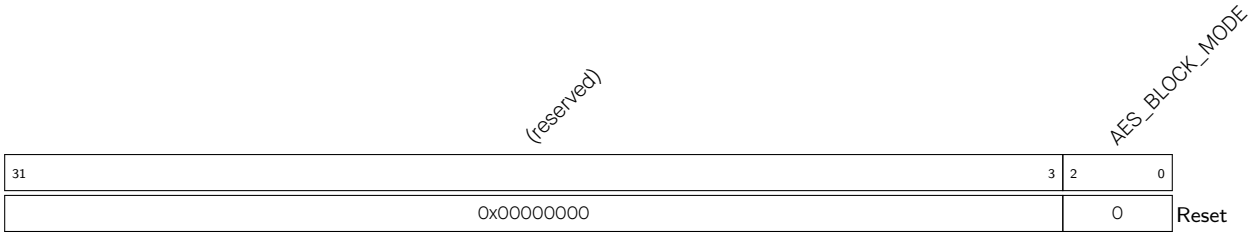
AES_MODE 选择 AES 加速器的密钥长度和加解密方向，详情请见表 19.3-2。(R/W)

Register 19.5. AES_DMA_ENABLE_REG (0x0090)



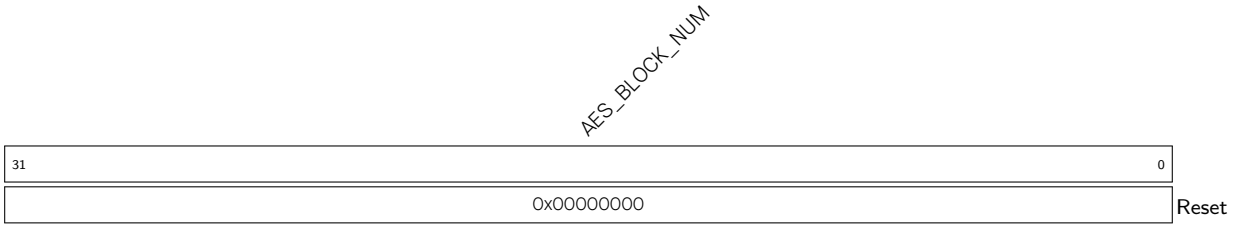
AES_DMA_ENABLE 选择 AES 加速器的工作模式。0：Typical AES，1：DMA-AES。详情请见表 19.3-1。(R/W)

Register 19.6. AES_BLOCK_MODE_REG (0x0094)



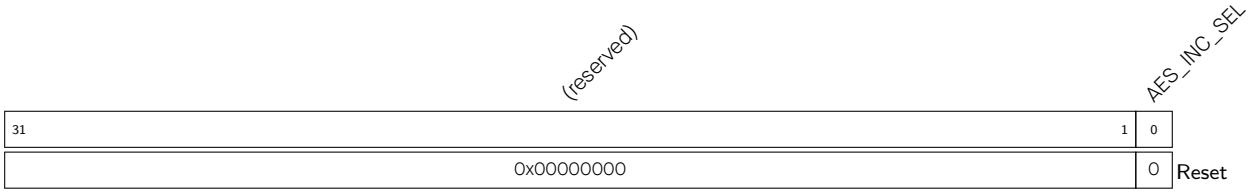
AES_BLOCK_MODE 选择 AES 加速器在 DMA-AES 工作模式下的块模式，详情请见表 19.5-1。(R/W)

Register 19.7. AES_BLOCK_NUM_REG (0x0098)



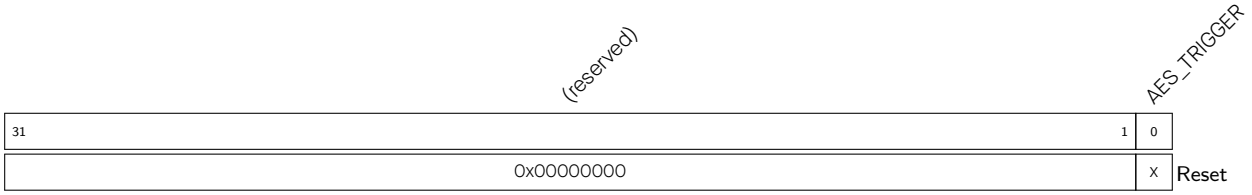
AES_BLOCK_NUM 在 DMA-AES 运算中待加解密的文本块数。详情请见章节 19.5.4。(R/W)

Register 19.8. AES_INC_SEL_REG (0x009C)



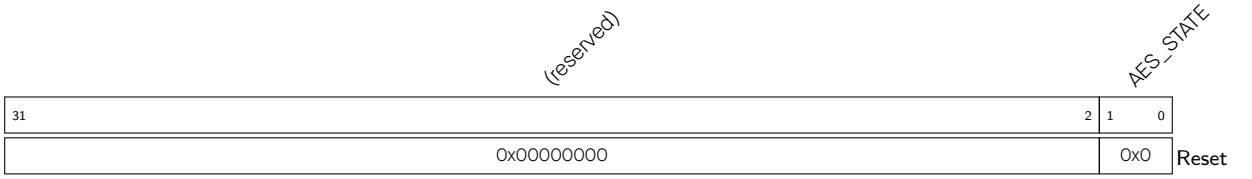
AES_INC_SEL 选择 CTR 块模式使用的标准增量函数。置 0 选择 INC_{32} 标准增量函数，置 1 选择 INC_{128} 标准增量函数。(R/W)

Register 19.9. AES_TRIGGER_REG (0x0048)



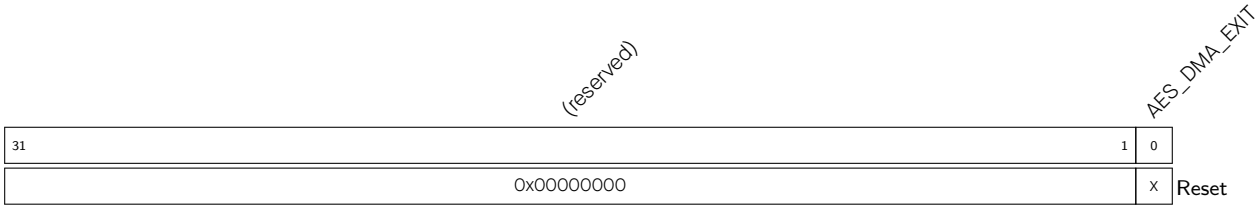
AES_TRIGGER 写入 1 使能 AES 运算。(WO)

Register 19.10. AES_STATE_REG (0x004C)



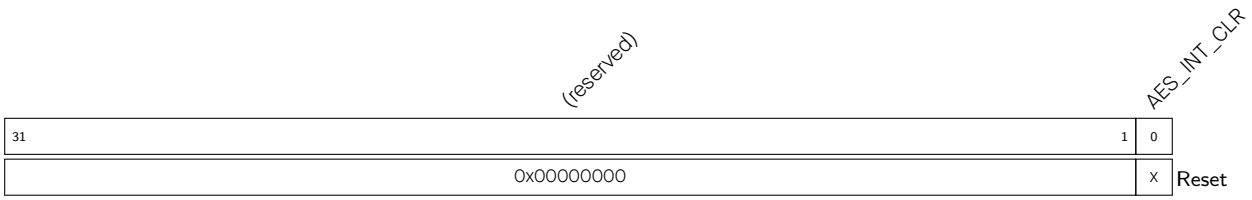
AES_STATE AES 状态寄存器。详见表 19.4-1（Typical AES 工作模式）和表 19.5-2（DMA-AES 工作模式）。(RO)

Register 19.11. AES_DMA_EXIT_REG (0x00B8)



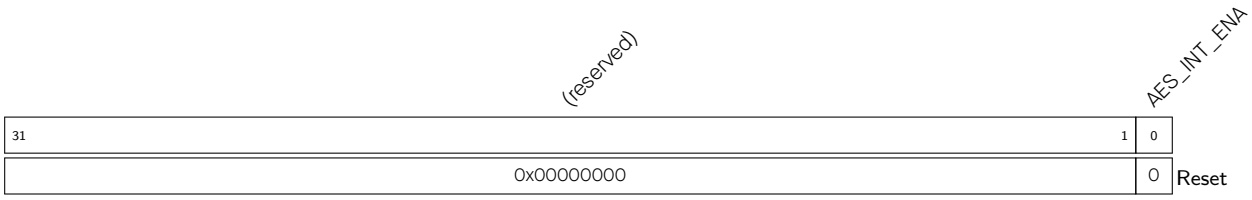
AES_DMA_EXIT 在 DMA-AES 运算完成后，在下一次配置 AES 任何寄存器之前，写入 1 使 AES 回到空闲状态。(WO)

Register 19.12. AES_INT_CLR_REG (0x00AC)



AES_INT_CLR 写入 1 清除 AES 中断。(WO)

Register 19.13. AES_INT_ENA_REG (0x00B0)



AES_INT_ENA 写入 1 使能 AES 中断功能，写入 0 关闭 AES 中断功能。(R/W)

第 20 章

RSA 加速器 (RSA)

20.1 概述

RSA 加速器可为多种运用于“RSA 非对称式加密演算法”的高精度计算提供硬件支持，能够极大地降低此类运算的软件复杂度，且支持多种“运算子长度”，具有很高的运算效率。

20.2 主要特性

RSA 加速器支持以下功能：

- 大数模幂运算（支持两个加速选项）
- 大数模乘运算
- 大数乘法运算
- 多种运算子长度
- 中断功能

20.3 功能描述

RSA 加速器的激活仅需使能 SYSTEM_PERIP_CLK_EN1_REG 外围时钟的 SYSTEM_CRYPTO_RSA_CLK_EN 位，并同时清零 SYSTEM_RSA_PD_CTRL_REG 寄存器中的SYSTEM_RSA_MEM_PD 位。

不过，RSA 加速器激活后还须等待 RSA 相关存储器初始化完成后才能开始工作。具体来说，寄存器 RSA_CLEAN_REG 读 0 时初始化开始，读 1 时初始化完成。因此，在复位后首次使用 RSA 加速器时，软件需要先查询寄存器 RSA_CLEAN_REG 的值是否为 1，以确保 RSA 加速器可正常工作。

此外，RSA 加速器支持中断功能，可对寄存器 RSA_INTERRUPT_ENA_REG 写 1/0 以开启/关闭中断。RSA 加速器的中断功能默认开启。

注意：

ESP32-S3 的数字签名 (DS)模块也会调用 RSA 加速器。此时，用户无法正常访问 RSA 加速器。

20.3.1 大数模幂运算

大数模幂运算的算法是 $Z = X^Y \bmod M$，它是基于 Montgomery Multiplication（蒙哥马利乘法）实现的。因此，对于大数模幂运算，除了需要运算子 X、Y、M 外，还需要额外两个运算子，即参数 $\overline{r}$ 和 M'。这两个参数需要通过软件提前运算得到。

RSA 加速器支持运算子长度为 $N = 32 \times x$ ($x \in \{1, 2, 3, \ldots, 128\}$) 的大数模幂运算。$Z$、$X$、$Y$、$M$ 和 $\overline{r}$ 的位宽为这 128 种中的任意一种，要求它们的位宽必须相同，而 M' 的位宽始终是 32。

设进制数

$$b = 2^{32}$$

则运算子可以由若干个 b 进制数来表示：

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

其中 $Z_{n-1}\cdots Z_0$、$X_{n-1}\cdots X_0$、$Y_{n-1}\cdots Y_0$、$M_{n-1}\cdots M_0$、$\overline{r}_{n-1}\cdots \overline{r}_0$ 分别表示一个 b 进制数，位宽皆为 32。且 Z_{n-1}、X_{n-1}、Y_{n-1}、M_{n-1}、$\overline{r}_{n-1}$ 分别为 Z、X、Y、M、$\overline{r}$ 最高位的 b 进制数，而 Z_0、X_0、Y_0、M_0、$\overline{r}_0$ 分别为 Z、X、Y、M、$\overline{r}$ 最低位的 b 进制数。

另设 $R = b^n$，则计算得参数 $\overline{r} = R^2 \bmod M$。

M' 可使用下方公式计算：

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

注意，上方公式适用于使用扩展二进制 GCD 算法的运算。

大数模幂运算的软件流程为：

1. 对寄存器 RSA_INTERRUPT_ENA_REG 写 1/0 以开启/关闭中断。
2. 配置相关寄存器。
   (a) 对寄存器 RSA_MODE_REG 写入 ($\frac{N}{32} - 1$)。
   (b) 对寄存器 RSA_M_PRIME_REG 写入 M'。
   (c) 根据需要配置加速选项相关寄存器。请参照章节 20.3.4 获取详细信息。
3. 将 X_i、Y_i、M_i、$\overline{r}_i (i \in \{0, 1, \ldots, n-1\})$ 分别写入存储器 RSA_X_MEM、RSA_Y_MEM、RSA_M_MEM、RSA_Z_MEM。每块存储器的容量都是 128 字 (word)。每块存储器的每一个字刚好存放一个 b 进制数。这些存储器都是低地址存放运算子的低位进制数，高地址存放运算子的高位进制数。

   只需要根据运算子长度，将各个运算子中有效的数据写入存储器，没有使用到的存储器可以是任意值。
4. 对寄存器 RSA_MODEXP_START_REG 写入 1 启动计算。
5. 等待运算结束。轮询寄存器 RSA_IDLE_REG 直到读到 1，或者等待 RSA 中断产生。
6. 从存储器 RSA_Z_MEM 读出运算结果 $Z_i (i \in \{0, 1, \ldots, n-1\})$。

7. 若中断功能已开启，对寄存器 RSA_CLEAR_INTERRUPT_REG 写入 1 以清除中断。

运算结束后，寄存器 RSA_MODE_REG 中存储的运算子长度信息以及存储器 RSA_Y_MEM 中的 Y_i、存储器 RSA_M_MEM 中的 M_i、寄存器 RSA_M_PRIME_REG 中的 M' 都不会变化。但是，存储器 RSA_X_MEM 中的 X_i 与存储器 RSA_Z_MEM 中的 $\overline{r}_i$ 都已经被覆盖。所以当需要连续运算时，只需要更新必需的寄存器与存储器即可。

20.3.2 大数模乘运算

大数模乘运算 $Z = X \times Y \bmod M$ 也是基于 Montgomery Multiplication 实现的。因此，与大数模幂运算类似，也需要预先通过软件计算额外的两个运算子 $\overline{r}$ 和 M'。

RSA 加速器也支持 128 种运算子长度的大数模乘运算。

大数模乘运算的软件流程为：

1. 对寄存器 RSA_INTERRUPT_ENA_REG 写 1/0 以开启/关闭中断。
2. 配置相关寄存器。
   (a) 对寄存器 RSA_MODE_REG 写入 $(\frac{N}{32} - 1)$。
   (b) 对寄存器 RSA_M_PRIME_REG 写入 M'。
3. 将 X_i、Y_i、M_i、$\overline{r}_i$ $(i \in \{0, 1, \ldots, n-1\})$ 分别写入存储器 RSA_X_MEM、RSA_Y_MEM、RSA_M_MEM、RSA_Z_MEM。每块存储器的容量都是 128 字 (word)。

   每块存储器的每一个字刚好存放一个 b 进制数。这些存储器都是低地址存放运算子的低位进制数，高地址存放运算子的高位进制数。

   只需要根据运算子长度，将各个运算子中有效的数据写入存储器，没有使用到的存储器可以是任意值。
4. 对寄存器 RSA_MODMULT_START_REG 写入 1。
5. 等待运算结束。轮询寄存器 RSA_IDLE_REG 直到读到 1，或者等待 RSA 中断产生。
6. 从存储器 RSA_Z_MEM 读出运算结果 Z_i $(i \in \{0, 1, \ldots, n-1\})$。
7. 若中断功能已开启，对寄存器 RSA_CLEAR_INTERRUPT_REG 写入 1 以清除中断。

运算结束后，寄存器 RSA_MODE_REG 中存储的运算子长度信息以及存储器 RSA_X_MEM 中的 X_i、存储器 RSA_Y_MEM 中的 Y_i、存储器 RSA_M_MEM 中的 M_i、寄存器 RSA_M_PRIME_REG 中的 M' 都不会变化。但是，存储器 RSA_Z_MEM 中的 $\overline{r}_i$ 已经被覆盖。所以当需要连续运算时，只需要更新必需的寄存器与存储器即可。

20.3.3 大数乘法运算

大数乘法运算实现了 $Z = X \times Y$。其中 Z 的长度是运算子 X、Y 长度的两倍。所以 RSA 加速器只支持运算子 X、Y 长度为 $N = 32 \times x$ $(x \in \{1, 2, 3, \ldots, 64\})$ 的大数乘法运算。运算子 Z 的长度 $\hat{N}$ 为 $2 \times N$。

大数乘法运算的软件流程为：

1. 对寄存器 RSA_INTERRUPT_ENA_REG 写 1/0 以开启/关闭中断。
2. 配置相关寄存器。对寄存器 RSA_MODE_REG 写入 $(\frac{\hat{N}}{32} - 1)$，即 $(\frac{N}{16} - 1)$。

3. 将 X_i、Y_i ($i \in \{0, 1, \ldots, n-1\}$) 分别写入存储器 RSA_X_MEM、RSA_Z_MEM。每块存储器的容量都是 64 字。每块存储器的每一个字刚好存放一个 b 进制数。这些存储器都是低地址存放运算子的低位进制数，高地址存放运算子的高位进制数。n 为 $\frac{N}{32}$ 。

   X_i ($i \in \{0, 1, \ldots, n-1\}$) 要填充到存储器 RSA_X_MEM 中的第 i 个字对应的地址中，但需要注意的是，Y_i ($i \in \{0, 1, \ldots, n-1\}$) 并不是要填充到存储器 RSA_Z_MEM 中的第 i 个字对应的地址中，而是需要填充到存储器 RSA_Z_MEM 中的第 $n+i$ 个字对应的地址中，即存储器 RSA_Z_MEM 的基地址加上偏移量 $4 \times (n+i)$。

   只需要根据运算子长度，将这两个运算子中有效的数据写入存储器，没有使用到的存储器可以是任意值。

4. 对寄存器 RSA_MULT_START_REG 写入 1。
5. 等待运算结束。轮询寄存器 RSA_IDLE_REG 直到读到 1，或者等待 RSA 中断产生。
6. 从存储器 RSA_Z_MEM 读出运算结果 Z_i ($i \in \{0, 1, \ldots, \hat{n}-1\}$)。$\hat{n}$ 为 $2 \times n$ 。
7. 若中断功能已开启，对寄存器 RSA_CLEAR_INTERRUPT_REG 写入 1 以清除中断。

运算结束后，寄存器 RSA_MODE_REG 中存储的运算子长度信息以及存储器 RSA_X_MEM 中的 X_i 都不会变化。但是，存储器 RSA_Z_MEM 中的 Y_i 已经被覆盖。所以当需要连续运算时，只需要更新必需的寄存器与存储器即可。

20.3.4 控制加速

对于大数模幂运算，ESP32-S3 的 RSA 加速器还特别提供 SEARCH 和 CONSTANT_TIME 两个选项，可提高运算速度。默认情况下，这两个选项均处于不加速状态，可以单独使用，也可以同时使用。

具体来说，当这两个选项均处于不加速状态时，求解 $Z = X^Y \bmod M$ 的时间开销完全由运算子长度决定。否则，只要有某个选项携带有加速效果，那么运算的时间开销还与 Y 的 0/1 分布有关。

为了更清楚地说明问题，首先假设 Y 的二进制表示为：

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

其中，

- N 代表 Y 的长度，
- $\widetilde{Y}_t$ 的值为 1，
- $\widetilde{Y}_{N-1}$, $\widetilde{Y}_{N-2}$, ..., $\widetilde{Y}_{t+1}$ 的值均为 0，
- 且 $\widetilde{Y}_{t-1}$, $\widetilde{Y}_{t-2}$, ..., $\widetilde{Y}_0$ 中包括 m 个 0，其余 t-m 全部为 1，即 $\widetilde{Y}_{t-1}\widetilde{Y}_{t-2}, \cdots, \widetilde{Y}_0$ 的汉明重量 (Hamming weight) 为 $t-m$。

此时，当启动任一选项时：

- SEARCH 选项（RSA_SEARCH_ENABLE 置 1 开始加速）
  - RSA 加速器将忽略所有 $\widetilde{Y}_i$ （$i > \alpha$）位。其中，加速位置 α 可通过 RSA_SEARCH_POS_REG 寄存器配置。α 的最大值不能超过 N-1，否则相当于没有加速；且不建议小于 t，否则无法正确求解 $Z = X^Y \bmod M$。当设置 α 为 t 时，加速效果最佳。此时，$\widetilde{Y}_{N-1}$, $\widetilde{Y}_{N-2}$, ..., $\widetilde{Y}_{t+1}$ 中的 0 位将在运算中全部被忽略。

- CONSTANT_TIME 选项（RSA_CONSTANT_TIME_REG 置 0 开始加速）
  - RSA 加速器在运算过程中将简化对 Y 中 0 位的处理。因此不难想象，Y 中的 0 越多，加速效果越明显。

为了直观地展示这两个选项带来的加速效果，下面通过一个典型实例加以说明。在 $Z = X^Y \bmod M$ 中，N 等于 3072，Y 等于 65537。表 20.3-1 展示了 4 种选项组合对应的时间开销。注意，这里 SEARCH 选项开启时设定 α 为 16。

表 20.3-1. 加速效果

SEARCH 选项	CONSTANT_TIME 选项	时间开销 (ms)
不加速	不加速	752.81
加速	不加速	4.52
不加速	加速	2.406
加速	加速	2.33

可以看到：

- 当两个选项均处于不加速状态时，时间开销最大。
- 当两个选项均处于加速状态时，时间开销最小。
- 相比于不加速状态，任一选项处于加速状态时的时间开销明显大幅度降低。

20.4 存储器列表

请注意，这里的地址都是相对于 RSA 加速器基地址的地址偏移量（相对地址），详见章节 4 系统和存储器 中的表 4.3-3。

名称	描述	大小（字节）	起始地址	结束地址	访问
RSA_M_MEM	存储器 M	512	0x0000	0x01FF	只写
RSA_Z_MEM	存储器 Z	512	0x0200	0x03FF	读 / 写
RSA_Y_MEM	存储器 Y	512	0x0400	0x05FF	只写
RSA_X_MEM	存储器 X	512	0x0600	0x07FF	只写

20.5 寄存器列表

请注意，这里的地址都是相对于 RSA 加速器基地址的地址偏移量（相对地址），详见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

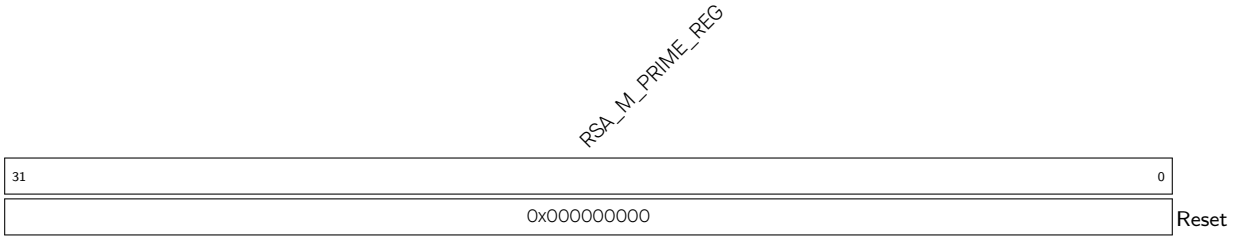
名称	描述	地址	访问
配置寄存器			
RSA_M_PRIME_REG	M' 存储器	0x0800	读 / 写
RSA_MODE_REG	RSA 长度模式	0x0804	读 / 写
RSA_CONSTANT_TIME_REG	固定时间选项	0x0820	读 / 写

RSA_SEARCH_ENABLE_REG	使能 search 加速选项	0x0824	读 / 写
RSA_SEARCH_POS_REG	search 起始位置	0x0828	读 / 写
状态/控制寄存器			
RSA_CLEAN_REG	RSA 清除寄存器	0x0808	只读
RSA_MODEXP_START_REG	模幂运算起始位	0x080C	只写
RSA_MODMULT_START_REG	模乘运算起始位	0x0810	只写
RSA_MULT_START_REG	乘法运算起始位	0x0814	只写
RSA_IDLE_REG	RSA 闲置寄存器	0x0818	只读
中断寄存器			
RSA_CLEAR_INTERRUPT_REG	RSA 中断清除寄存器	0x081C	只写
RSA_INTERRUPT_ENA_REG	RSA 中断使能寄存器	0x082C	读 / 写
版本寄存器			
RSA_DATE_REG	RSA 日期与版本寄存器	0x0830	读 / 写

20.6 寄存器

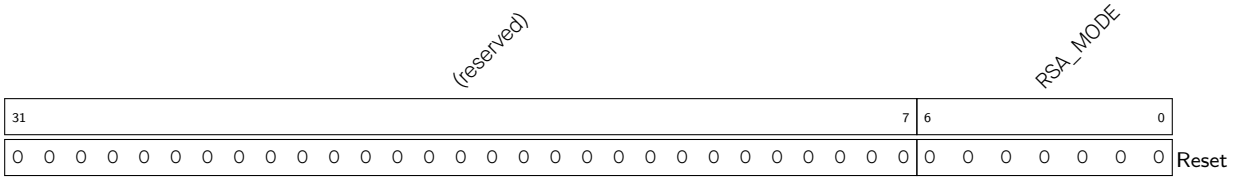
请注意，这里的地址都是相对于 RSA 加速器基地址的地址偏移量（相对地址），详见章节 4 系统和存储器 中的表 4.3-3。

Register 20.1. RSA_M_PRIME_REG (0x0800)



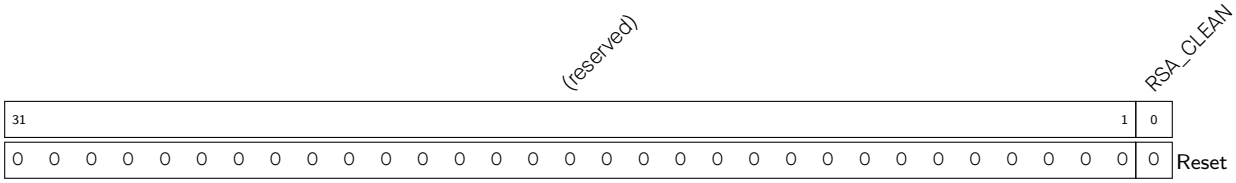
RSA_M_PRIME_REG 此寄存器存储 M'。(R/W)

Register 20.2. RSA_MODE_REG (0x0804)



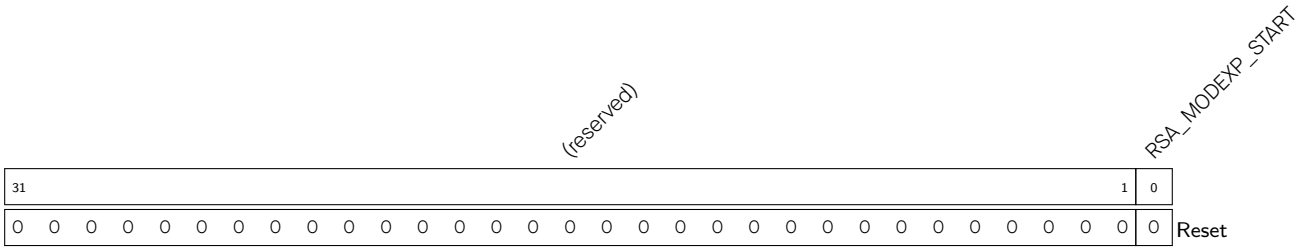
RSA_MODE 此寄存器存储模幂运算的模式。(R/W)

Register 20.3. RSA_CLEAN_REG (0x0808)



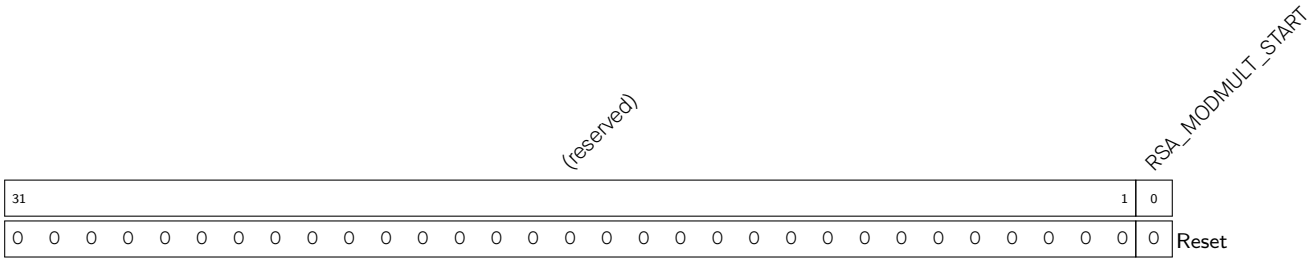
RSA_CLEAN 一旦存储器初始化结束，此位为 1。(RO)

Register 20.4. RSA_MODEXP_START_REG (0x080C)



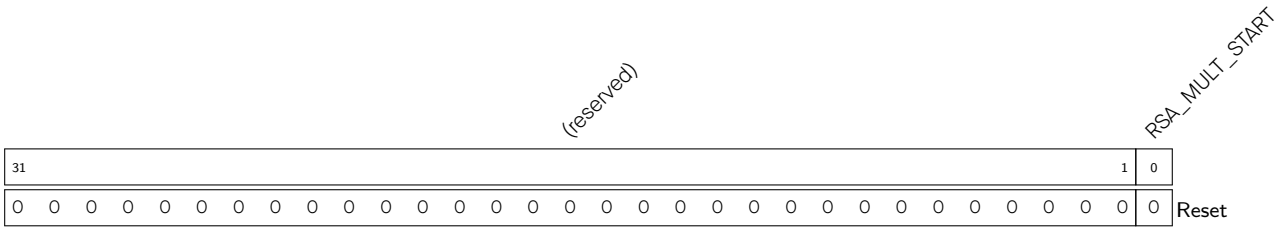
RSA_MODEXP_START 写入 1 以开始模幂运算。(WO)

Register 20.5. RSA_MODMULT_START_REG (0x0810)



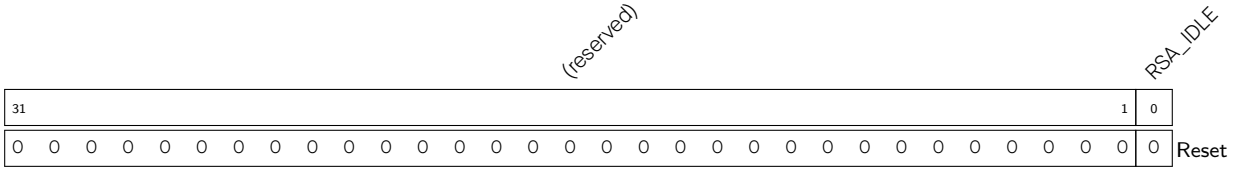
RSA_MODMULT_START 写入 1 以开始模乘运算。(WO)

Register 20.6. RSA_MULT_START_REG (0x0814)



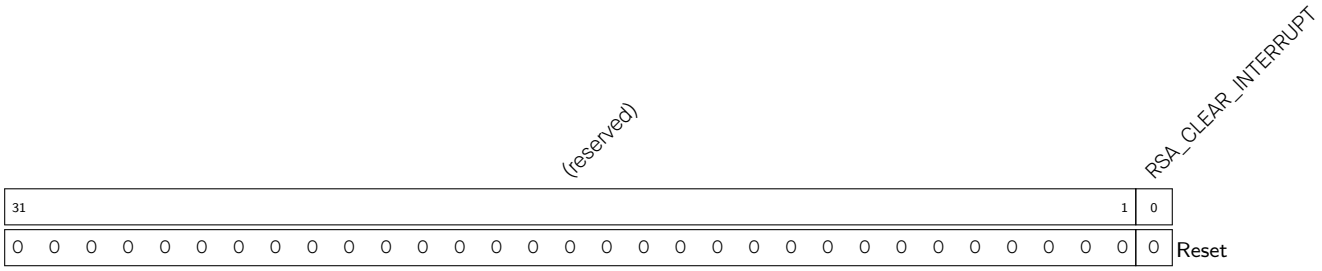
RSA_MULT_START 写入 1 以开始乘法运算。(WO)

Register 20.7. RSA_IDLE_REG (0x0818)



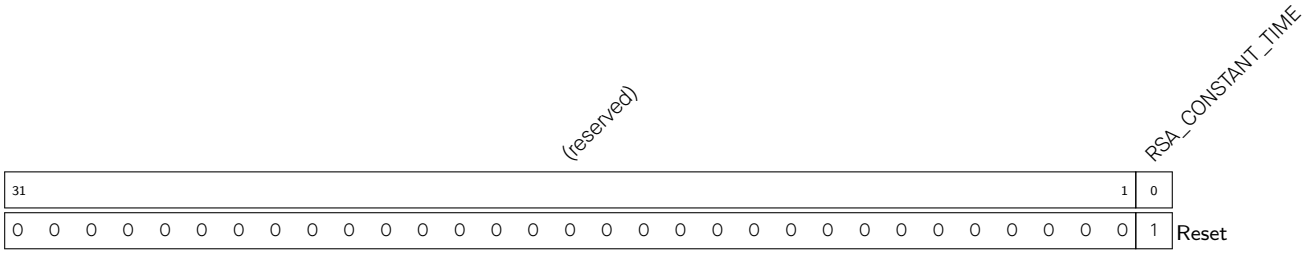
RSA_IDLE 当 RSA 空闲时，此位为 1。(RO)

Register 20.8. RSA_CLEAR_INTERRUPT_REG (0x081C)



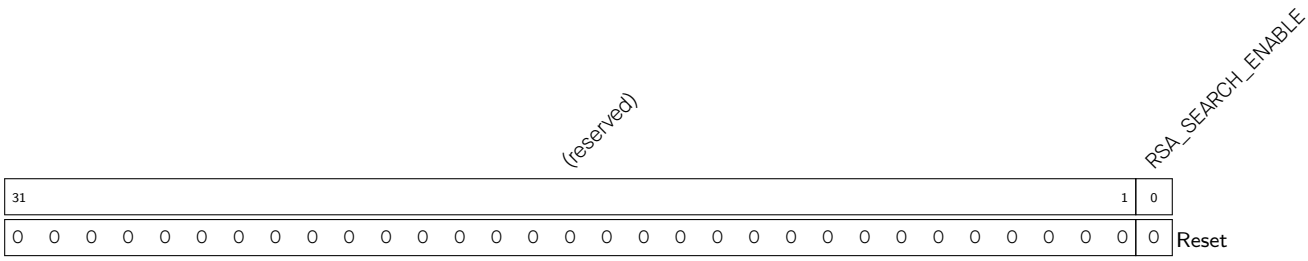
RSA_CLEAR_INTERRUPT RSA 中断清除寄存器。写入 1 清除中断。(WO)

Register 20.9. RSA_CONSTANT_TIME_REG (0x0820)



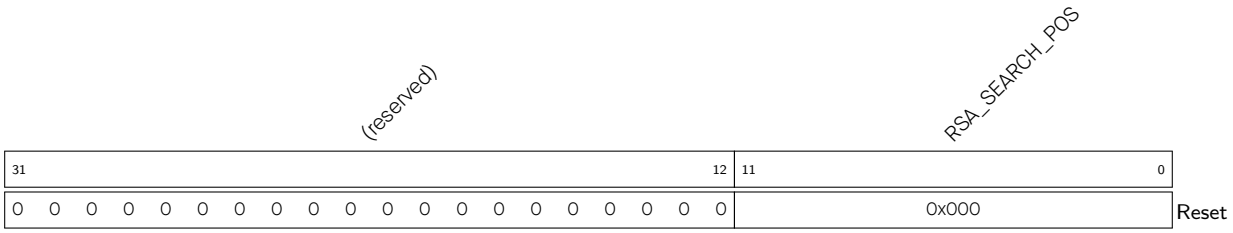
RSA_CONSTANT_TIME_REG 控制模幂运算中的 constant_time 选项。0：加速；1：不加速（默认）。(R/W)

Register 20.10. RSA_SEARCH_ENABLE_REG (0x0824)



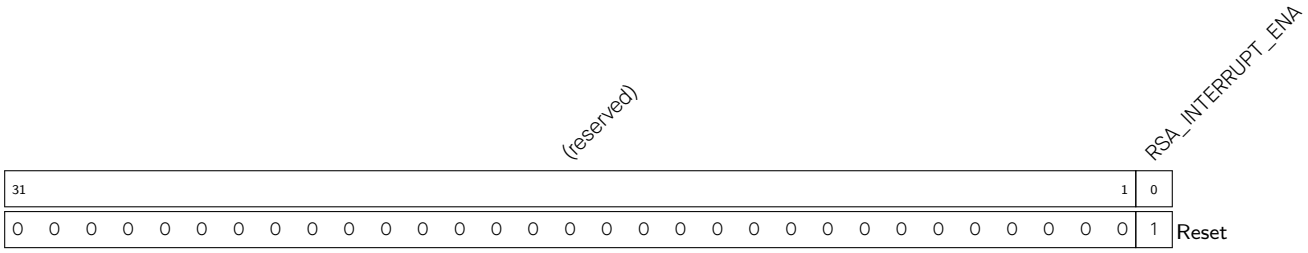
RSA_SEARCH_ENABLE 控制模幂运算中的 search 选项。1：加速；0：不加速（默认）。(R/W)

Register 20.11. RSA_SEARCH_POS_REG (0x0828)



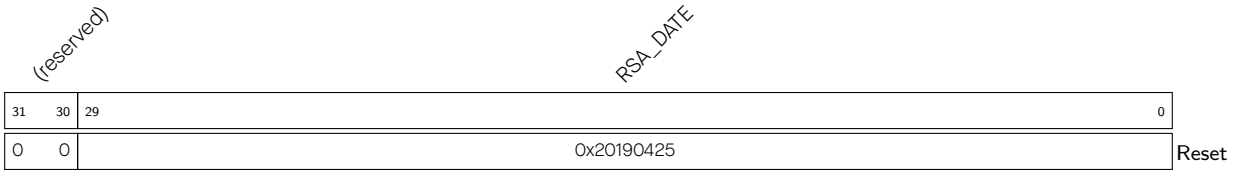
RSA_SEARCH_POS 模幂运算中的 search 加速选项。用于配置 search 的起始位置。(R/W)。

Register 20.12. RSA_INTERRUPT_ENA_REG (0x082C)



RSA_INTERRUPT_ENA RSA 中断使能寄存器。写入 1 开启中断，默认开启。(R/W)

Register 20.13. RSA_DATE_REG (0x0830)



RSA_DATE 版本控制寄存器。(R/W)

第 21 章

HMAC 加速器 (HMAC)

如 RFC 2104 中所述，HMAC 模块通过 hash 算法和密钥计算得到数据信息的信息认证码 (MAC)。其中的 hash 算法是 SHA-256，长度为 256-bit 的 HMAC 密钥存储在 eFuse 的密钥块中，可配置成不能被用户读取。

21.1 主要特性

- 标准 HMAC-SHA-256 算法
- HMAC 计算的 hash 结果仅支持特定的硬件外设访问（下行模式）
- 兼容挑战-应答身份验证算法
- 生成数字签名外设所需的密钥（下行模式）
- 重启软禁用的 JTAG（下行模式）

21.2 功能描述

HMAC 模块可工作于两种模式，即上行模式和下行模式。上行模式中，由用户提供 HMAC 信息，且用户回读其计算结果；下行模式中，HMAC 模块作为其他内部硬件的密钥导出函数 (KDF)。例如，通过将 eFuse 中 EFUSE_SOFT_DIS_JTAG 参数的奇数个比特烧写成 1 可以禁用 JTAG 功能。在该情况下，用户可以通过 HMAC 的下行模式重新开启 JTAG 功能。

此外，HMAC 的复位信号被释放后，将自动检查 eFuse 中是否有用于计算数字签名模块密钥导出函数的 Key。如果 Key 存在，HMAC 将主动完成下行模式的数字签名导出函数计算任务。

21.2.1 上行模式

上行模式通常用于支持 HMAC-SHA-256 的挑战 - 应答协议的应用场景。在上行模式中，由用户提供 HMAC 信息，且用户回读其计算结果。

在挑战-应答协议中，通讯双方称为 A、B，且 A、B 使用同个密钥。想要交换的完整的数据信息为 M，协议的一般验证流程为：

- A 计算出一个特殊的随机数信息 M
- A 将 M 发送给 B
- B 计算 HMAC 结果（通过 M 和密钥）并将其发送给 A
- A 内部计算 HMAC 结果（通过 M 和密钥）
- A 比较两次计算结果。如比较结果相同，则验证通过了 B 的身份

用户操作流程如下：

1. 用户初始化 HMAC 模块，进入上行模式。
2. 用户将正确填充的信息写入外设中，一次写入 512 比特数据。
3. 用户从外设寄存器中回读 HMAC 值。

有关此过程的详细步骤，可参见章节 21.2.6。

21.2.2 下行 JTAG 启动模式

eFuse memory 中有两个参数可以关闭 JTAG 调试：EFUSE_DIS_PAD_JTAG 和 EFUSE_SOFT_DIS_JTAG。前者烧写为 1，JTAG 将被永久关闭；后者烧写为奇数个 1，JTAG 将被暂时关闭。详细信息可参见章节 *5 eFuse 控制器 (EFUSE)*。

对于暂时关闭的 JTAG，用户可以按照以下步骤重新启动 JTAG：

1. 用户启动 HMAC 模块，配置其进入下行 JTAG 启动模式。
2. 用户将 1 写入 HMAC_SOFT_JTAG_CTRL_REG 寄存器进入 JTAG 重启比较模式。
3. 用户将预先在本地使用 SHA-256 和已知的随机密钥对 32 字节的 0x00 进行 HMAC 计算得到的 256-bit 数值结果按照 word 的大端序依次写入 HMAC_WR_JTAG_REG。
4. 如果 HMAC 计算的结果与用户本地计算的数值结果匹配，则 JTAG 重启。否则，JTAG 仍保持关闭状态。
5. 在用户将 1 烧写入寄存器 HMAC_SET_INVALIDATE_JTAG_REG 或上电重启之前，JTAG 将保持步骤 4 中的状态。

有关此过程的详细步骤，可参见章节 21.2.6。

21.2.3 下行数字签名模式

数字签名 (DS) 模块使用 AES-CBC 加密其参数。HMAC 模块作为密钥导出函数 (KDF) 导出解密上述参数的 AES 密钥。

在启用 DS 模块之前，用户必须先通过 HMAC 模块计算得到 DS 模块工作时所需的密钥。详细信息可参见章节 *22 数字签名 (DS)*。当 HMAC 复位信号释放且时钟信号有效的情况下，HMAC 模块将自动检查 eFuse 中是否烧写有 DS 模块所需要的功能密钥，如果存在该密钥，将自动进入下行数字签名模式，完成相应密钥计算。

21.2.4 烧写 HMAC 密钥

当前 HMAC 模块共支持 3 种功能：下行模式下的 JTAG 重启功能和 DS 密钥导出功能以及上行模式下的 HMAC 计算功能。表 21.2-1 列出了各功能对应的配置寄存器时的数值。

表 21.2-1. HMAC 功能及配置数值

功能	模式	数值	描述
JTAG 重启	下行模式	6	EFUSE_KEY_PURPOSE_HMAC_DOWN_JTAG
DS 密钥导出	下行模式	7	EFUSE_KEY_PURPOSE_HMAC_DOWN_DIGITAL_SIGNATURE
HMAC 计算	上行模式	8	EFUSE_KEY_PURPOSE_HMAC_UP
JTAG 重启和 DS 密钥导出	下行模式	5	EFUSE_KEY_PURPOSE_HMAC_DOWN_ALL

启动 HMAC 计算前，必须确保 eFuse 中计算过程中使用到的密钥数据，否则计算将失败。烧写密钥到 eFuse 的大致流程如下：

1. 准备一个 256-bit HMAC 密钥，将其烧写到空的 eFuse 密钥块 y 中（eFuse 中共有 6 个密钥块，分别为 4 ~ 9，因此 y 的值为 4 ~ 9。因此，若密钥为 key0，则对应的密钥块为 block4），并声明该密钥块的功能为 EFUSE_KEY_PURPOSE_$(y-4)$。例如在上行模式下，烧写密钥后，应将 EFUSE_KEY_PURPOSE_HMAC_UP（对应值为 8）写入 EFUSE_KEY_PURPOSE_$(y-4)$。更多烧写 eFuse 密钥的详细信息，可参见章节 5 *eFuse 控制器 (EFUSE)*。
2. 配置 eFuse 密钥块读保护功能，使用户无法读取密钥值。用户将步骤中生成的随机密钥安全地存储在其他位置。

21.2.5 HMAC 功能初始化

HMAC 模块的正确配置取决于选取的 eFuse 密钥块（烧写进 eFuse 时已经声明了用途）是否与用户配置的工作模式一致。错误的配置将结束本次计算任务。

配置 HMAC 工作模式

用户将表示工作模式的有效数值（见表 21.2-1）写入寄存器 HMAC_SET_PARA_PURPOSE_REG。

选取 eFuse 的密钥块

eFuse 控制器共提供 6 个密钥块，KEY0 ~ 5 。用户将编号 n 写入寄存器 HMAC_SET_PARA_KEY_REG，表示选择 KEYn 作为本次 HMAC 模块运行时使用的密钥。

需要注意的是，eFuse memory 中的密钥在烧写时都定义了功能用途，只有当 HMAC 的配置功能与 KEYn 定义的功能用途相匹配时，HMAC 模块才会执行配置好的计算任务。否则，返回匹配错误结果并结束当前计算任务。

比如，如果用户选择了 KEY3 作为本次计算的密钥，且烧写入 KEY_PURPOSE_3 中的数值为 6 (EFUSE_KEY_PURPOSE_HMAC_DOWN_JTAG)，则参照表 21.2-1 可知，KEY3 是用于 JTAG 重启的密钥。如果此时寄存器 HMAC_SET_PARA_PURPOSE_REG 中配置的数值也是 6，HMAC 外设才会启动 JTAG 重启功能的计算。

21.2.6 调用 HMAC 流程（详细说明）

用户调用 ESP32-S3 中 HMAC 流程如下：

1. 启动 HMAC 模块
   (a) 启动寄存器 SYSTEM_PEIRP_CLK_EN1_REG 中的 HMAC 和 SHA 的外设时钟位，清除寄存器 SYSTEM_PEIRP_RST_EN1_REG 中相应的外设重启位。相应的寄存器信息参见章节 17 *系统寄存器 (SYSTEM)*。
   (b) 将数值 1 写入寄存器 HMAC_SET_START_REG。
2. 配置 HMAC 密钥和密钥功能
   (a) 将表示密钥功能的 m 写入寄存器 HMAC_SET_PARA_PURPOSE_REG。表 21.2-1 描述了数值 m 对应的密钥功能，可参见章节 21.2.4。

(b) 通过将数值 n 写入寄存器 HMAC_SET_PARA_KEY_REG，选择 eFuse memory 中的 KEYn 作为本次计算的密钥（n 的取值范围为 0 ~ 5），可参见章节 21.2.5。

(c) 将数值 1 写入寄存器 HMAC_SET_PARA_FINISH_REG，完成配置工作。

(d) 读取寄存器 HMAC_QUERY_ERROR_REG。如果返回值为 1，表明选取的密钥块与配置的密钥功能不匹配，结束本次计算任务；如果返回值为 0，表明选取的密钥块与配置的密钥功能匹配，可以执行计算流程。

(e) 如果设置 HMAC_SET_PARA_PURPOSE_REG 的数值不为 8，表明 HMAC 模块将工作在下行模式下，跳转到步骤 3；如果设置其数值为 8，表明 HMAC 模块将工作在上行模式下，跳转到步骤 4。

3. 下行模式

(a) 轮询状态寄存器 HMAC_QUERY_BUSY_REG，当读取到该寄存器的值为 0 时，表明下行模式下的 HMAC 运算完成。

(b) 下行模式下，计算结果供硬件内部的 JTAG 模块或 DS 模块使用。如需清除计算结果以更好地使用硬件空间（JTAG 或 DS 模块）用户可将数值 1 写入寄存器 HMAC_SET_INVALIDATE_JTAG_REG 清除 JTAG 密钥生成的结果；也可将数值 1 写入寄存器 HMAC_SET_INVALIDATE_DS_REG 清除数字签名密钥生成的结果。

(c) 下行模式下的操作完成。

4. 上行模式下传输数据块 Block_n（$n >= 1$）

(a) 轮询状态寄存器 HMAC_QUERY_BUSY_REG，当读取到该寄存器的值为 0 时，继续步骤 4(b)。

(b) 将 512-bit 的数据块 Block_n 写入寄存器 HMAC_WDATA0~15_REG 中，随后将数值 1 写入寄存器 HMAC_SET_MESSAGE_ONE_REG，HMAC 模块将计算该数据块。

(c) 轮询状态寄存器 HMAC_QUERY_BUSY_REG，当读取到该寄存器的值为 0 时，继续步骤 4(d)。

(d) 根据待处理数据总比特数是否是 512 的整数倍，后续将产生不同数据块。

- 如果待处理数据总比特数是 512 的整数倍，有以下 3 种选项：
  i. 如果 Block_n+1 存在，将数值 1 写入寄存器 HMAC_SET_MESSAGE_ING_REG，令 $n = n + 1$，随后跳转到步骤 4.(b)。
  ii. 如果 Block_n 是最后一个待处理数据块，用户希望由硬件进行 SHA 附加填充，将数值 1 写入寄存器 HMAC_SET_MESSAGE_END_REG，随后跳转到步骤 6。
  iii. 如果 Block_n 是最后一个填充的数据块，且用户已在软件中进行 SHA 附加填充时，将数值 1 写入寄存器 HMAC_SET_MESSAGE_PAD_REG，随后跳转到步骤 5。
- 如果待处理数据总比特数不是 512 的倍数，有以下 3 种选项。注意，这种情况下用户应对数据进行 SHA 附加填充，且填充后待输入数据总比特数应为 512 的整数倍。
  i. 如果 Block_n 是唯一一个数据块，且 $n = 1$，同时 Block_1 已经包含了所有的填充位，则将数值 1 写入寄存器 HMAC_ONE_BLOCK_REG，随后跳转到步骤 6。
  ii. 如果 Block_n 是倒数第二个填充数据块，将数值 1 写入寄存器 HMAC_SET_MESSAGE_PAD_REG，执行附加填充比特操作，随后跳转到步骤 5。
  iii. 如果 Block_n 既不是最后一个也不是倒数第二个数据块，将数值 1 写入寄存器 HMAC_SET_MESSAGE_ING_REG，令 $n = n + 1$，随后跳转到步骤 4.(b)。

5. 进行 SHA 附加填充

(a) 用户根据 21.3.1 章节描述对最后一个数据块进行 SHA 附加填充，并将该数据块写入寄存器 HMAC_WDATA0~15_REG，随后将数值 1 写入寄存器 HMAC_SET_MESSAGE_ONE_REG，HMAC 模块开始计算该数据块。

(b) 跳转到步骤 6。

6. 读取上行模式下的结果 hash 值

(a) 轮询状态寄存器 HMAC_QUERY_BUSY_REG，当读取到该寄存器的值为 0 时，继续下一步。

(b) 从寄存器 HMAC_RDATA0~7_REG 中读取出 hash 结果数值。

(c) 将数值 1 写入寄存器 HMAC_SET_RESULT_FINISH_REG，结束当次计算。

(d) 上行模式下的操作完成。

说明：
DS 模块和 HMAC 模块可直接调用或在内部使用 SHA 加速器，但不能同时与其共享硬件资源。因此在 HMAC 模块运行过程中，SHA 模块无法被 CPU 和 DS 模块调用。

21.3 HMAC 算法细节

21.3.1 附加填充比特

HMAC 模块中采用 SHA-256 作为加密 HASH 算法。该算法中，若待输入数据的总比特数不是 512 的倍数，用户须在软件中应用 SHA-256 附加填充算法。SHA-256 附加填充算法与 *FIPS PUB 180-4* 中 *Padding the Message* 相同，并在其章节中有详细描述。

如图 21.3-1 所示，假设待处理数据长度为 m 个比特，填充步骤如下：

1. 在待处理数据末尾附加 1 个比特长度的数值 “1”。
2. 附加 k 个比特的数值 “0”。其中，k 为满足 $m+1+k \equiv 448 (mod 512)$ 的最小非负数。
3. 附加一个 64 位的整数值作为二进制块。该二进制块的内容为待填充数据作为一个大端二进制整数值 m 的长度。

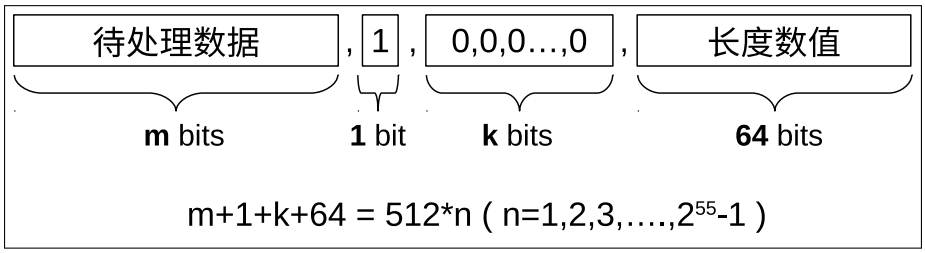


图 21.3-1. HMAC 附加填充比特示意图

下行模式下，用户无需输入数据或进行附加填充。上行模式下，若待填充数据总比特数是 512 的整数倍，则用户可配置由硬件完成 SHA 附加填充操作；若待填充数据总比特数不是 512 的整数倍，则用户只能自行完成 SHA 附加填充操作。详细步骤可参见章节 21.2.6。

21.3.2 HMAC 算法结构

HMAC 模块中应用的算法结构示意图如 21.3-2 所示。这是 RFC 2104 中描述的标准 HMAC 算法。

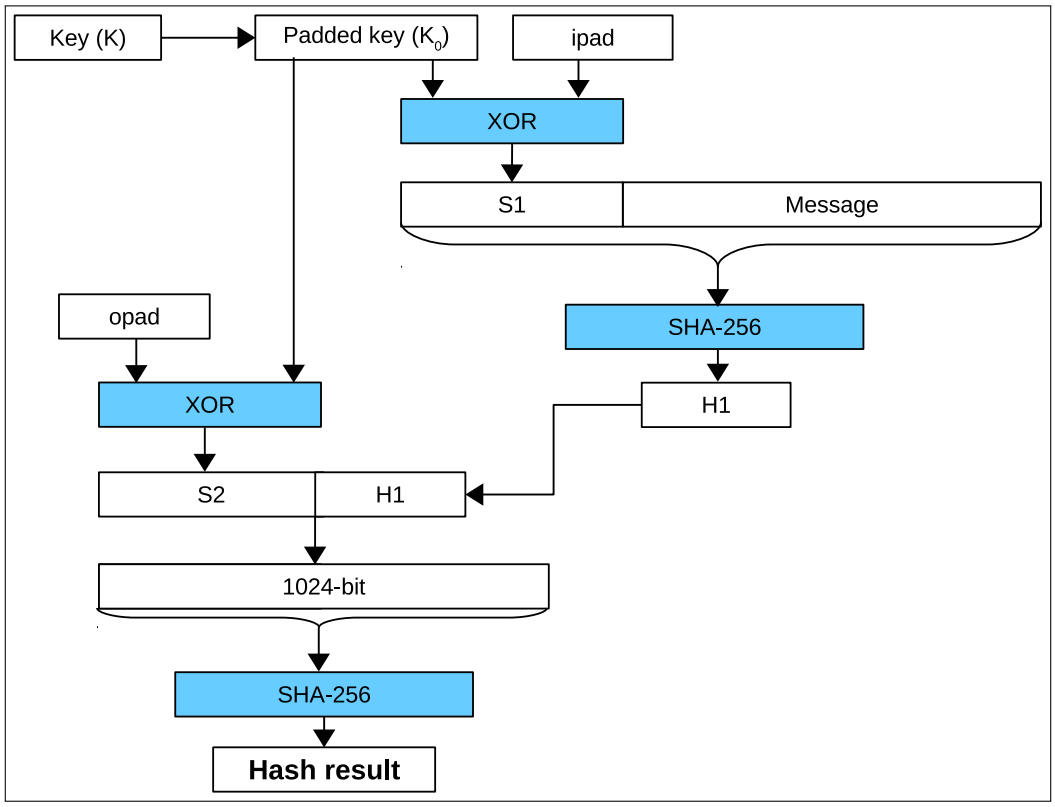


图 21.3-2. HMAC 结构示意图

图 21.3-2 中，

1. ipad 是由 64 个 0x36 字节组成的 512-bit 数据块。
2. opad 是由 64 个 0x5c 字节组成的 512-bit 数据块。

首先，HMAC 模块在 256-bit 的密钥 K 的比特序列后附加上 256-bit 的 0 序列，得到 512-bit 的 K_0。再对 K_0 和 ipad 进行异或运算，得到 512-bit 的 S1。将总比特数为 512 倍数的待输入数据附加到 512-bit 的 S1 数值后，使用 SHA-256 加密算法计算得到 256-bit 的 H1。

HMAC 模块通过对 K_0 和 opad 进行异或运算得到 S2，将 256-bit 的 hash 计算结果附加到 512-bit 的 S2 数值后，得到 768-bit 长度的序列，使用 21.3.1 章节中描述的 SHA 附加填充算法将该序列填充成 1024-bit 的序列，最后使用 SHA-256 加密算法计算得到的最终 hash 结果 (256-bit)。

21.4 寄存器列表

本小节的所有地址均为相对于 HMAC 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

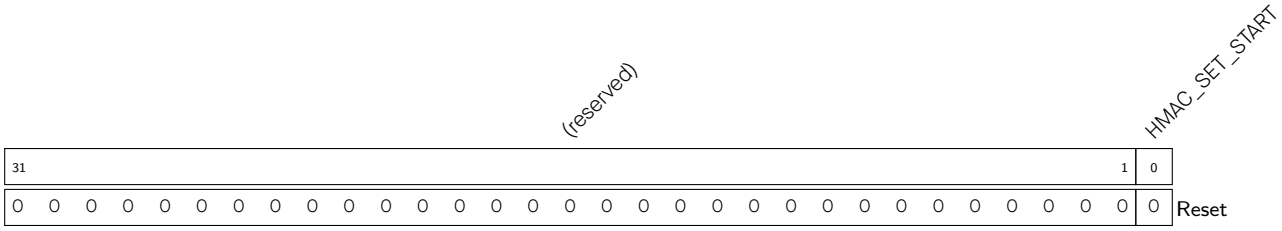
名称	描述	地址	访问
控制/状态寄存器			
HMAC_SET_START_REG	开始控制寄存器 0	0x040	WO
HMAC_SET_PARA_PURPOSE_REG	HMAC 参数配置寄存器	0x044	WO
HMAC_SET_PARA_KEY_REG	HMAC 密钥配置寄存器	0x048	WO
HMAC_SET_PARA_FINISH_REG	配置完成寄存器	0x04C	WO
HMAC_SET_MESSAGE_ONE_REG	信息控制寄存器	0x050	WO
HMAC_SET_MESSAGE_ING_REG	信息继续寄存器	0x054	WO
HMAC_SET_MESSAGE_END_REG	信息终止寄存器	0x058	WO
HMAC_SET_RESULT_FINISH_REG	HMAC 读取结果完成寄存器	0x05C	WO
HMAC_SET_INVALIDATE_JTAG_REG	注销 JTAG 结果寄存器	0x060	WO
HMAC_SET_INVALIDATE_DS_REG	注销数字签名结果寄存器	0x064	WO
HMAC_QUERY_ERROR_REG	存储用户配置的密钥和功能的配对结果	0x068	RO
HMAC_QUERY_BUSY_REG	存储 HMAC 模块的忙碌状态	0x06C	RO
HMAC 信息块寄存器			
HMAC_WR_MESSAGE_0_REG	信息寄存器 0	0x080	WO
HMAC_WR_MESSAGE_1_REG	信息寄存器 1	0x084	WO
HMAC_WR_MESSAGE_2_REG	信息寄存器 2	0x088	WO
HMAC_WR_MESSAGE_3_REG	信息寄存器 3	0x08C	WO
HMAC_WR_MESSAGE_4_REG	信息寄存器 4	0x090	WO
HMAC_WR_MESSAGE_5_REG	信息寄存器 5	0x094	WO
HMAC_WR_MESSAGE_6_REG	信息寄存器 6	0x098	WO
HMAC_WR_MESSAGE_7_REG	信息寄存器 7	0x09C	WO
HMAC_WR_MESSAGE_8_REG	信息寄存器 8	0x0A0	WO
HMAC_WR_MESSAGE_9_REG	信息寄存器 9	0x0A4	WO
HMAC_WR_MESSAGE_10_REG	信息寄存器 10	0x0A8	WO
HMAC_WR_MESSAGE_11_REG	信息寄存器 11	0x0AC	WO
HMAC_WR_MESSAGE_12_REG	信息寄存器 12	0x0B0	WO
HMAC_WR_MESSAGE_13_REG	信息寄存器 13	0x0B4	WO
HMAC_WR_MESSAGE_14_REG	信息寄存器 14	0x0B8	WO
HMAC_WR_MESSAGE_15_REG	信息寄存器 15	0x0BC	WO
HMAC 上行结果寄存器			
HMAC_RD_RESULT_0_REG	Hash 结果寄存器 0	0x0C0	RO
HMAC_RD_RESULT_1_REG	Hash 结果寄存器 1	0x0C4	RO
HMAC_RD_RESULT_2_REG	Hash 结果寄存器 2	0x0C8	RO
HMAC_RD_RESULT_3_REG	Hash 结果寄存器 3	0x0CC	RO
HMAC_RD_RESULT_4_REG	Hash 结果寄存器 4	0x0D0	RO
HMAC_RD_RESULT_5_REG	Hash 结果寄存器 5	0x0D4	RO

名称	描述	地址	访问
HMAC_RD_RESULT_6_REG	Hash 结果寄存器 6	0x0D8	RO
HMAC_RD_RESULT_7_REG	Hash 结果寄存器 7	0x0DC	RO
配置寄存器			
HMAC_SET_MESSAGE_PAD_REG	软件填充寄存器	0x0F0	WO
HMAC_ONE_BLOCK_REG	One block 信息寄存器	0x0F4	WO
HMAC_SOFT_JTAG_CTRL_REG	重启 JTAG 寄存器 0	0x0F8	WO
HMAC_WR_JTAG_REG	重启 JTAG 寄存器 1	0x0FC	WO
版本寄存器			
HMAC_DATE_REG	版本控制寄存器	0x1FC	R/W

21.5 寄存器

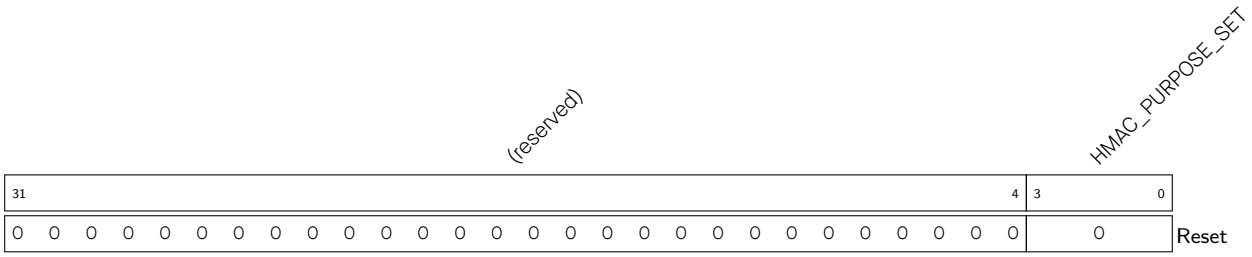
本小节的所有地址均为相对于 HMAC 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 21.1. HMAC_SET_START_REG (0x040)



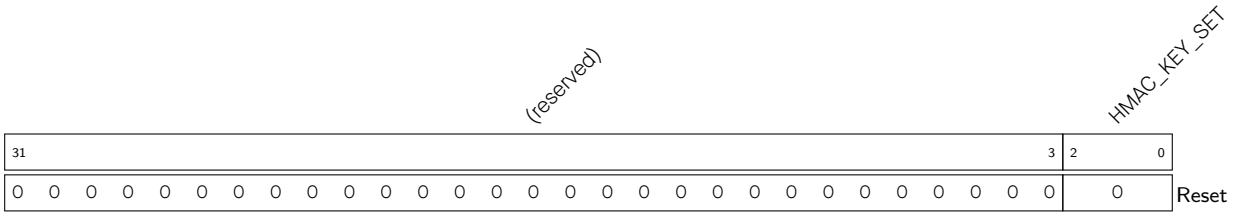
HMAC_SET_START 置位启动 HMAC。(WO)

Register 21.2. HMAC_SET_PARA_PURPOSE_REG (0x044)



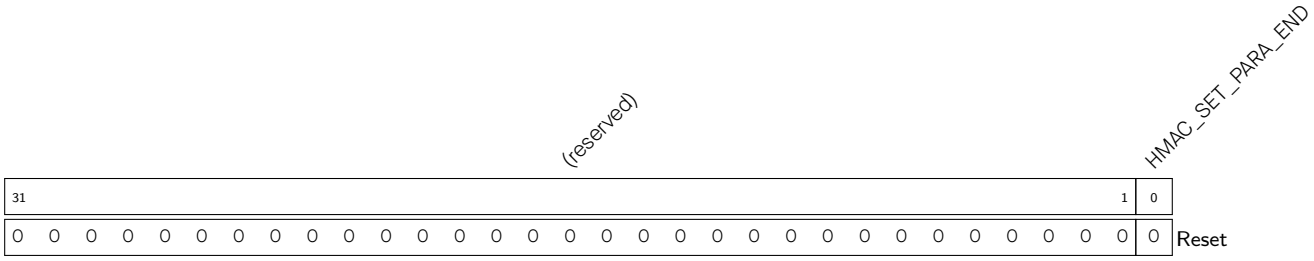
HMAC_PURPOSE_SET 置位 HMAC 功能，请参阅表 21.2-1。(WO)

Register 21.3. HMAC_SET_PARA_KEY_REG (0x048)



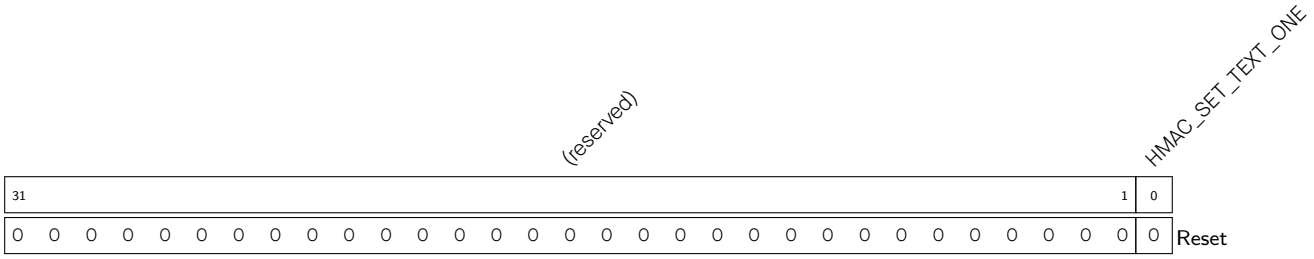
HMAC_KEY_SET 选择 HMAC 密钥。共有 6 个密钥，编号 0 至 5，将选择的密钥编号写入该字段即可。(WO)

Register 21.4. HMAC_SET_PARA_FINISH_REG (0x04C)



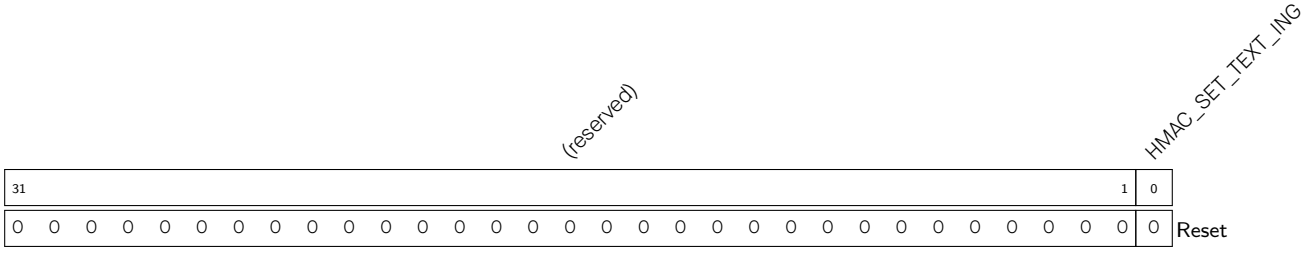
HMAC_SET_PARA_END 置位完成 HMAC 配置。(WO)

Register 21.5. HMAC_SET_MESSAGE_ONE_REG (0x050)



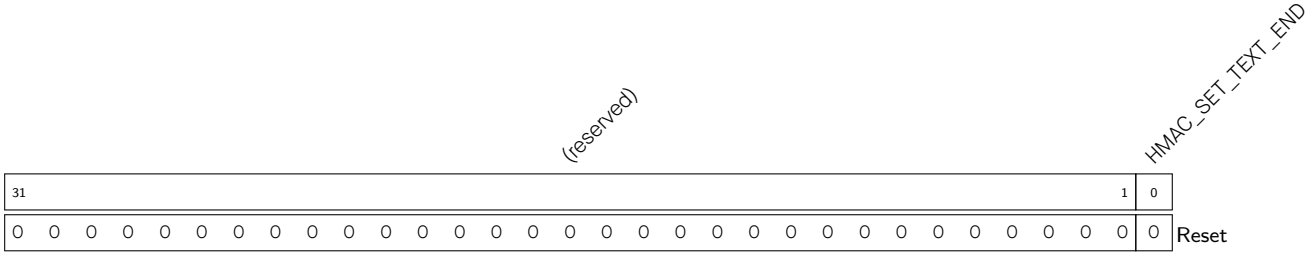
HMAC_SET_TEXT_ONE 调用 SHA 计算信息块。(WO)

Register 21.6. HMAC_SET_MESSAGE_ING_REG (0x054)



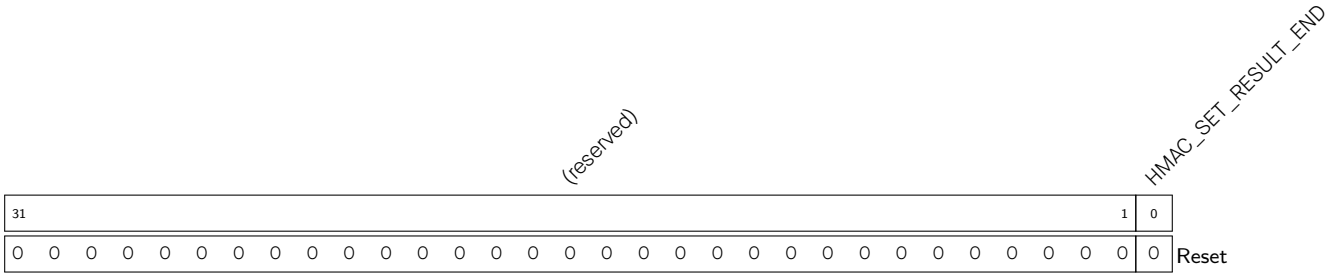
HMAC_SET_TEXT_ING 置位表明仍存在未处理信息块。(WO)

Register 21.7. HMAC_SET_MESSAGE_END_REG (0x058)



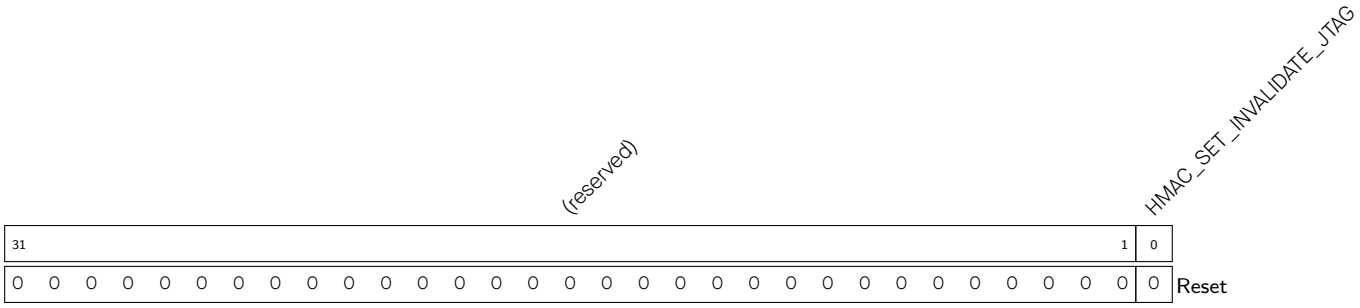
HMAC_SET_TEXT_END 置位开始硬件填充。(WO)

Register 21.8. HMAC_SET_RESULT_FINISH_REG (0x05C)



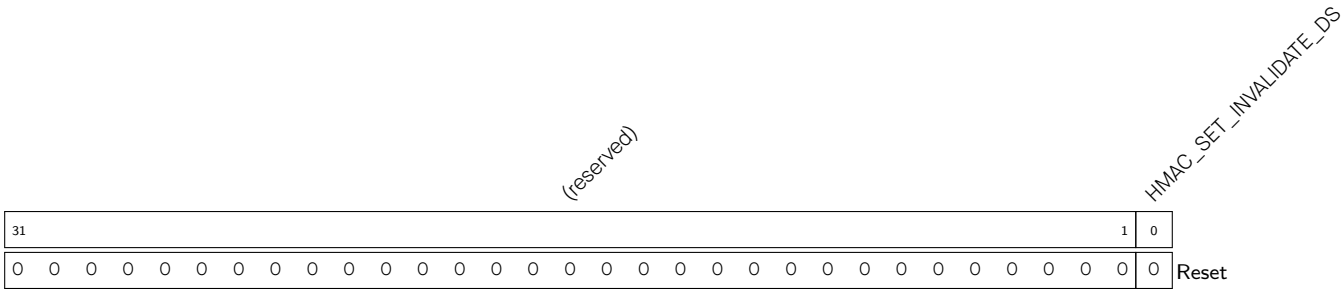
HMAC_SET_RESULT_END 置位结束上行模式，清空计算结果。(WO)

Register 21.9. HMAC_SET_INVALIDATE_JTAG_REG (0x060)



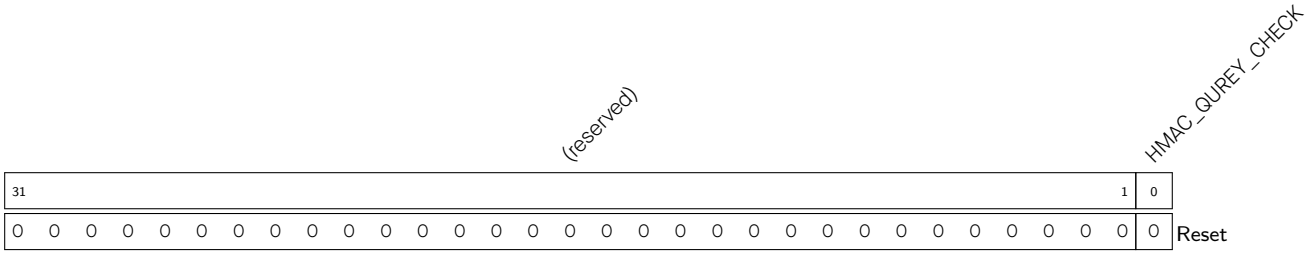
HMAC_SET_INVALIDATE_JTAG 置位清空下行模式下 JTAG 重启功能的计算结果。(WO)

Register 21.10. HMAC_SET_INVALIDATE_DS_REG (0x064)



HMAC_SET_INVALIDATE_DS 置位清空下行模式下电子签名的计算结果。(WO)

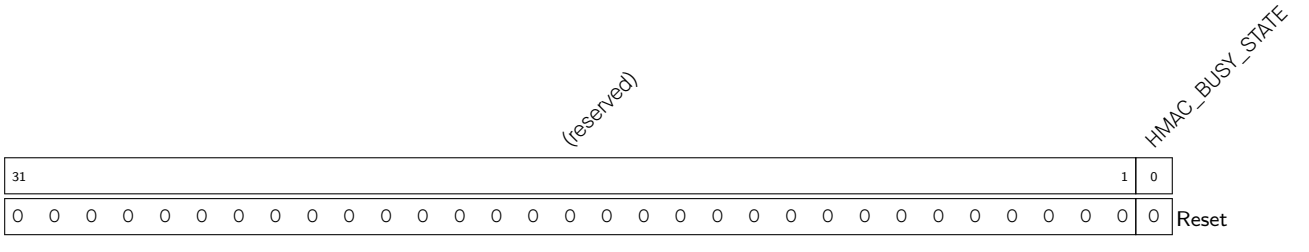
Register 21.11. HMAC_QUERY_ERROR_REG (0x068)



HMAC_QUREY_CHECK 指示 HMAC 密钥是否与功能匹配。

- 0: HMAC 密钥和功能匹配。
- 1: 错误。(RO)

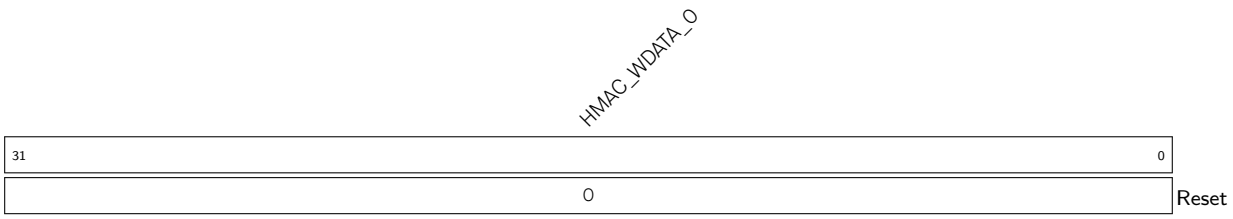
Register 21.12. HMAC_QUERY_BUSY_REG (0x06C)



HMAC_BUSY_STATE 指示 HMAC 是否处于忙碌状态。(RO)

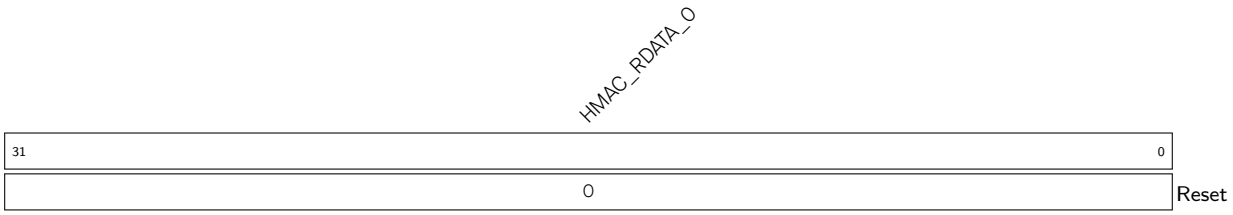
- 1'b0：空闲。
- 1'b1：HMAC 仍处于工作状态。

Register 21.13. HMAC_WR_MESSAGE_*n*_REG (*n*: 0-15) (0x080+4**n*)



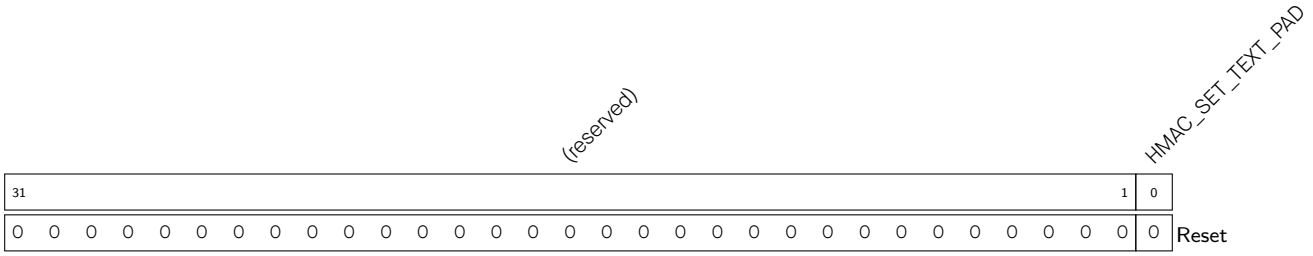
HMAC_WDATA_*n* 存储信息的第 *n* 个 32 位数据信息。(WO)

Register 21.14. HMAC_RD_RESULT_*n*_REG (*n*: 0-7) (0x0C0+4**n*)



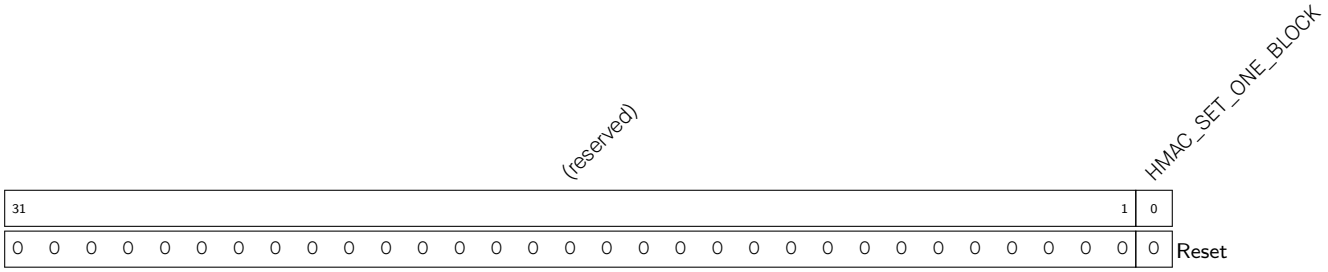
HMAC_RDATA_*n* 读取 hash 结果的第 *n* 个 32 位。(RO)

Register 21.15. HMAC_SET_MESSAGE_PAD_REG (0x0F0)



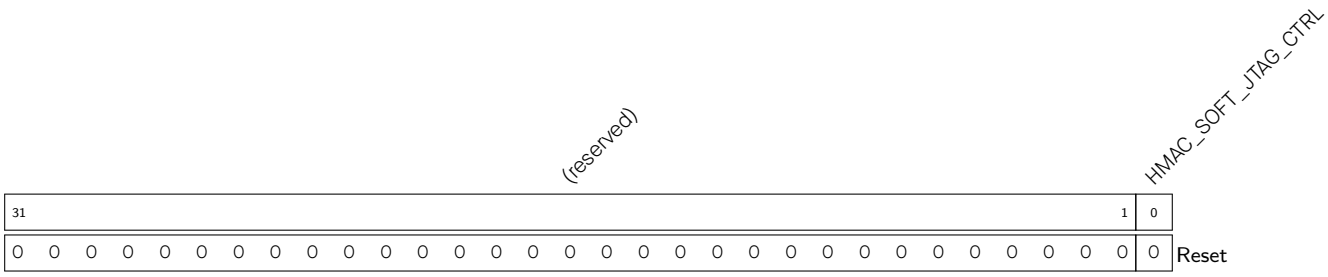
HMAC_SET_TEXT_PAD 置位由软件执行填充操作。(WO)

Register 21.16. HMAC_ONE_BLOCK_REG (0x0F4)



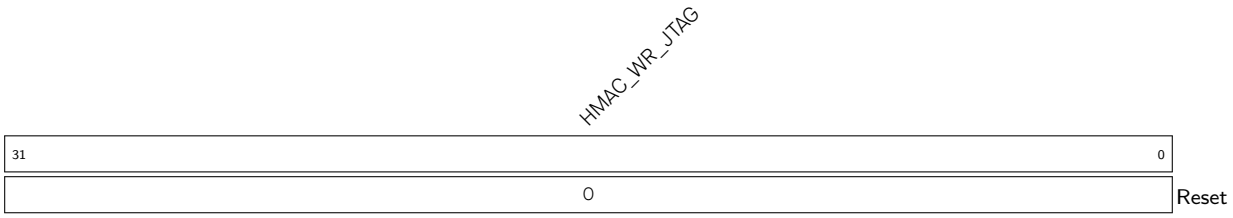
HMAC_SET_ONE_BLOCK 置位表明无需填充。(WO)

Register 21.17. HMAC_SOFT_JTAG_CTRL_REG (0x0F8)



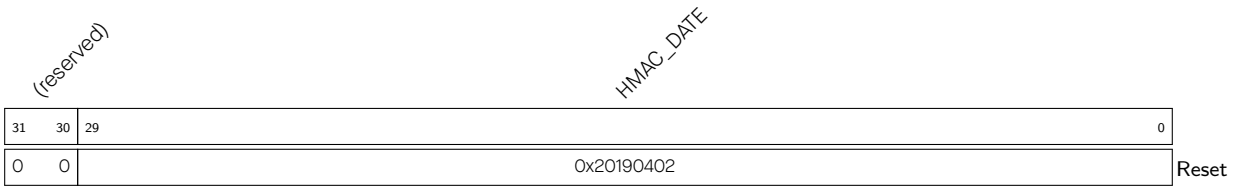
HMAC_SOFT_JTAG_CTRL 置位开启 JTAG 验证。(WO)

Register 21.18. HMAC_WR_JTAG_REG (0x0FC)



HMAC_WR_JTAG 置位对比 32 位密钥。(WO)

Register 21.19. HMAC_DATE_REG (0x1FC)



HMAC_DATE 版本控制寄存器。(R/W)

第 22 章

数字签名 (DS)

22.1 概述

数字签名技术在密码学算法层面上用于验证消息的真实性和完整性。这可用于向服务器验证设备自身身份，或检查消息是否未被篡改。

ESP32-S3 包含一个数字签名 (Digital Signature, DS) 模块，可基于 RSA 高效生成数字签名。数字签名模块使用预先加密的参数计算出签名，HMAC 作为密钥导出函数加密这些参数。反过来，HMAC 则使用 eFuse 作为输入密钥。上述整个过程都发生在硬件层面，因此在计算过程中，不论是解密 RSA 参数的密钥还是用于 HMAC 密钥导出函数的输入密钥都对用户不可见。

22.2 主要特性

- RSA 数字签名支持密钥长度最大为 4096 位
- 私钥数据已加密，并且只能由 DS 读取
- SHA-256 摘要用于保护私钥数据免遭攻击者篡改

22.3 功能描述

22.3.1 概述

DS 模块计算 RSA 签名操作 $Z = X^Y \bmod M$，其中 Z 是签名，X 是输入消息，Y 和 M 是 RSA 私钥参数。

私钥参数以密文形式存储在 flash 或其他存储器中。对其解密而使用的密钥（DS_KEY）只能由 DS 模块通过 HMAC 模块获取，并且 HMAC 模块求解该密钥所需的一切输入信息（$HMAC\_KEY$）只存放在 eFuse 中且只允许被 HMAC 模块访问。这意味着只有 DS 硬件才能解密私钥密文，软件绝对不会获取私钥明文。关于 eFuse 和 HMAC 的相关细节请参照章节 5 *eFuse 控制器 (EFUSE)* 和章节 21 *HMAC 加速器 (HMAC)*。

需要签名时，软件直接将输入消息 X 发送到 DS 外设。RSA 签名操作完成后，软件将读取签名结果 Z。

为方便描述，这里约定几个符号，它们的作用域局限于本章。

- 1^s 表示一个完全由 “1” 组成的长度为 s 位的位串。
- $[x]_s$ 一个长度为 s 位的位串，要求 s 为 8 的整数倍。如果 x 是一个数 ($x < 2^s$)，那么其在位串中遵循小端字节序。x 可以是一个变量，例如 $[Y]_{4096}$，或一个十六进制的常数，比如 $[0xOC]_8$。根据需要，$[x]_t$ 右边可以加上 $(s-t)$ 个 0，使字串长度扩展成 s 位，得到 $[x]_s$。例如：$[0x05]_8 = 00000101$，$[0x05]_{16} = 0000010100000000$，$[0x0005]_{16} = 0000000000000101$，$[0x13]_8 = 00010011$，$[0x13]_{16} = 0001001100000000$，$[0x0013]_{16} = 0000000000010011$。
- $||$ 表示位串粘接操作符，用于将两个位串前后粘成一个较长的位串。

22.3.2 私钥运算子

私钥运算子 Y（私钥指数）和 M（密钥模数）由用户生成。它们具有特定的 RSA 密钥长度（最大为 4096 位）。RSA 签名操作还额外需要两个运算子，即参数 $\overline{r}$ 和 M'，这两个参数需要软件由 Y 和 M 运算得到。

运算子 Y、M、$\overline{r}$ 和 M' 与验证摘要一起由用户加密并以密文 C 的形式存储。密文 C 输入到 DS 模块之后先由硬件解密，然后参与 RSA 签名运算。如何生成 C 请参考第 22.3.3 节。

DS 模块支持运算子长度为 $N = 32 \times x$ ($x \in \{1, 2, 3, \ldots, 128\}$) 的 RSA 签名运算 $Z = X^Y \bmod M$，需要满足 RSA 计算的运算子长度要求，即 Z、X、Y、M 和 $\overline{r}$ 的位宽必须相同，但必须为这 128 种中的其中一种，而 M' 的位宽始终是 32。更多 RSA 计算相关信息请参考章节 20 *RSA 加速器 (RSA)* 中的 20.3.1 大数模幂运算 部分。

22.3.3 软件需要做的准备工作

如果用户想使用 DS 模块进行数字签名，软件和硬件必须紧密配合才可以顺利完成，并且软件需要做一系列准备工作，如图 22.3-1 所示。图中左半边给出了在硬件开始 RSA 签名计算之前，软件需要做哪些准备工作。图中右半边展示了硬件在整个签名计算的过程中具体会做些什么。

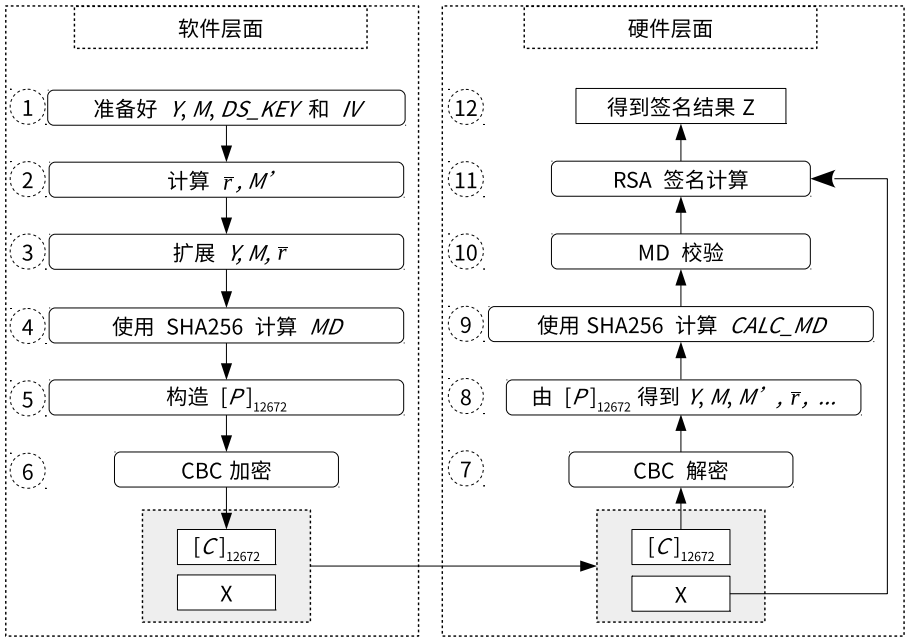


图 22.3-1. 软件准备工作与硬件工作流程

说明：

1. 对于多次签名计算，软件准备工作（图 22.3-1 左侧）只需要进行一次，但对于每一次签名计算，硬件运算（图 1-1 右侧）都需要重新进行。

用户需要依照图 22.3-1 指定的步骤来计算 C。详细的过程描述如下：

- **步骤 1**：按照第 22.3.2 节所述准备好 RSA 私钥运算子 Y 和 M，它们应符合运算子的长度要求。记 $[L]_{32} = \frac{N}{32} - 1$（比如，对于 RSA 4096，$[L]_{32}$ == [0x80-1]$_{32}$）。还需要准备好 $[HMAC\_KEY]_{256}$，并计算出 $[DS\_KEY]_{256}$，即 DS_KEY = HMAC-SHA256($[HMAC\_KEY]_{256}$, 1^{256})。另外还需要引入一个随机的 $[IV]_{128}$，它需要符合 AES-CBC 块加密算法的要求。有关 AES 更多信息，请参考章节 19 *AES 加速器*

(AES)。

- **步骤 2**：根据 M 求解 $\overline{r}$ 和 M'。
- **步骤 3**：扩展 Y、M 和 $\overline{r}$，得到 $[Y]_{4096}$、$[M]_{4096}$ 和 $[\overline{r}]_{4096}$。由于 Y、M 和 $\overline{r}$ 的最大位宽为 4096，运算子位宽小于 4096 需要扩展为 4096，位宽等于 4096 则不需要扩展。
- **步骤 4**：使用 SHA-256 计算 MD 校验码：$[MD]_{256}$ = SHA256 $([Y]_{4096}||[M]_{4096}||[\overline{r}]_{4096}||[M']_{32}||[L]_{32}||[IV]_{128})$
- **步骤 5**：构造 $[P]_{12672}$ = ($[Y]_{4096}||[M]_{4096}||[\overline{r}]_{4096}||[MD]_{256}||[M']_{32}||[L]_{32}||[\beta]_{64}$)，其中 $[\beta]_{64}$ 是符合 PKCS#7 封装方式的追加码，由 8 个字节码 0x80 组成的一个 64 位的位串 $[0x0808080808080808]_{64}$，目的在于使 P 的长度为 128 位的整数倍。
- **步骤 6**：计算密文形式的私钥参数 C = $[C]_{12672}$ = AES-CBC-ENC ($[P]_{12672}$, $[DS\_KEY]_{256}$, $[IV]_{128}$)，长度为 1584 字节。

22.3.4 硬件工作流程

每次需要计算数字签名时，都会触发硬件操作。硬件需要三个输入信息：预先生成的私钥密文 C、唯一的消息 X、IV。

DS 模块的工作流程可以分为如下三个阶段：

1. **解析阶段，即图 22.3-1 中的步骤 7 和步骤 8**

   解析过程是图 22.3-1 中步骤 6 的逆过程。DS 模块将调用 AES 硬件加速器以 CBC 块模式对输入的密文信息 C 进行解密，获取明文信息。该过程可以表示为 P = AES-CBC-DEC (C, DS_KEY, IV)，其中 IV 就是 $[IV]_{128}$，由用户直接指定；$[DS\_KEY]_{256}$ 由硬件 HMAC 提供，由存储在 eFuse 中的 $HMAC\_KEY$ 得到，软件无法获取。请参考章节 *21 HMAC 加速器 (HMAC)* 获取更多信息。

   显然，DS 模块能够通过 P 解析出 $[Y]_{4096}$、$[M]_{4096}$、$[\overline{r}]_{4096}$、$[M']_{32}$、$[L]_{32}$、MD 校验码和追加码 $[\beta]_{64}$，这相当于步骤 5 的逆过程。

2. **校验阶段，即图 22.3-1 中的步骤 9 和步骤 10**

   DS 模块会执行两种校验操作：MD 校验和填充 (padding) 校验。由于填充校验和 MD 校验同步进行，因此填充校验不在图 22.3-1 中体现。

   - MD 校验——DS 模块调用 SHA-256 进行哈希计算获取哈希结果值 $[CALC\_MD]_{256}$（即步骤 4），然后将 $[CALC\_MD]_{256}$ 与 MD 校验码 $[MD]_{256}$ 作比较，当且仅当二者相同时，MD 校验通过。
   - 填充校验——DS 模块将检查解析阶段解析出的追加码 $[\beta]_{64}$ 是否符合 PKCS#7 标准，当且仅当符合标准时，填充校验通过。

   如果 MD 校验通过，DS 模块将执行后续计算；否则 DS 模块拒绝执行。如果填充校验失败，将生成警告信息，但不会影响 DS 模块的后续操作。

3. **计算阶段，即图 22.3-1 中的步骤 11 和步骤 12**

   DS 模块将把用户输入的 X，以及解析得到的 Y、M 和 $\overline{r}$ 都视为大数，结合解析得到的 M'，构成了大数模幂运算 X^Y mod M 的所有必要输入参数。大数模幂运算的运算长度由 L 的值唯一指定。DS 模块调用 RSA 加速器完成大数模幂运算 $Z = X^Y \bmod M$，Z 为签名结果。

22.3.5 软件工作流程

每次需要计算数字签名时，都应执行以下软件操作。输入消息是预先生成的私钥密文 C、唯一的消息 X、IV。这些软件步骤触发章节 22.3.4 中描述的硬件工作流程。下述流程基于一个假设：软件已经调用了 HMAC 外设，硬件上 HMAC 已经根据 $HMAC\_KEY$ 计算出了 DS_KEY。

1. **准备工作**：准备好 C、X、IV。具体方法请参考章节 22.3.3 。
2. **启动 DS**：对寄存器 DS_SET_START_REG 写 1。
3. **检查 DS_KEY 是否已经准备好**：轮询寄存器 DS_QUERY_BUSY_REG 直到读到 0。

   如果 DS_QUERY_BUSY_REG 超过 1 ms 还没读到 0，则说明 HMAC 未被调用。此时，软件应当读寄存器 DS_QUERY_KEY_WRONG_REG，根据返回值判断具体是哪一种情况。

   - 如果读到零值，说明 HMAC 未被调用。
   - 如果读到非零值 (1 ~ 15)，则说明 HMAC 被调用过，但是 DS 模块没有拿到 DS_KEY，原因可能是有其他程序的干扰。
4. **配置寄存器**：将 IV block 写入寄存器 DS_IV_*m*_REG (*m*: 0-3)。有关 IV block 的更多信息，请参考章节 19 *AES 加速器 (AES)*。
5. **将 X 写入存储器 DS_X_MEM**：将 X_i ($i \in \{0, 1, \ldots, n-1\}$) 写入存储器 DS_X_MEM，容量为 128 个字 (word)，其中 $n = \frac{N}{32}$ 。每一个字刚好存放一个 b 进制数。存储器的低地址存放运算子的低位进制数，高地址存放运算子的高位进制数。当 X 的长度小于 128 个字时，存储器 DS_X_MEM 中有一部分空间未使用，该部分空间中的数据可以是任意值。
6. **将 C 写入存储器 DS_C_MEM**：将 C_i ($i \in \{0, 1, \ldots, 395\}$) 写入存储器 DS_C_MEM，容量为 396 个字。每一个字刚好存放一个 b 进制数。
7. **启动计算**：对寄存器 DS_SET_ME_REG 写入 1。
8. **等待运算结束**：轮询寄存器 DS_QUERY_BUSY_REG 直到读到 0。
9. **检查校验结果**：读寄存器 DS_QUERY_CHECK_REG，根据返回值决定后续操作。
   - 如果返回值为 0，则说明填充校验通过，MD 校验通过，可以继续读取 Z 结果值。
   - 如果返回值为 1，则说明填充校验通过，但 MD 校验失败。Z 结果值全零无效，跳至步骤 11。
   - 如果返回值为 2，则说明填充校验通过失败，但 MD 校验通过，用户可以继续读取 Z 结果值。但仍需注意的是，数据填充不符合 PKCS#7 封装方式，这可能不是你想要的。
   - 如果返回值为 3，则说明填充填充校验失败，且 MD 校验失败。这种情况说明有致命错误发生。Z 结果值全零无效。跳至步骤 11。
10. **读出运算结果**：从存储器 DS_Z_MEM 读出运算结果 Z_i ($i \in \{0, 1, \ldots, n-1\}$)，其中 $n = \frac{N}{32}$ 。Z 以小端字节序存储在存储器中。
11. **退出计算环境**：对寄存器 DS_SET_FINISH_REG 写入 1，然后轮询寄存器 DS_QUERY_BUSY_REG 直到读到 0。

DS 退出计算环境后，所有输入/输出寄存器和存储器中的数据都已经被抹除（清零）。

22.4 存储器列表

请注意，这里的起始地址和结束地址都是相对于 Digital Signature 基地址的地址偏移量（相对地址）。请参阅章节 4 系统和存储器 中的表 4.3-3 获取 DS 模块的基地址。

名称	描述	大小（字节）	起始地址	结束地址	访问
DS_C_MEM	存储器 C	1584	0x0000	0x062F	WO
DS_X_MEM	存储器 X	512	0x0800	0x09FF	WO
DS_Z_MEM	存储器 Z	512	0x0A00	0x0BFF	RO

22.5 寄存器列表

本小节的所有地址均为相对于 Digital Signature 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

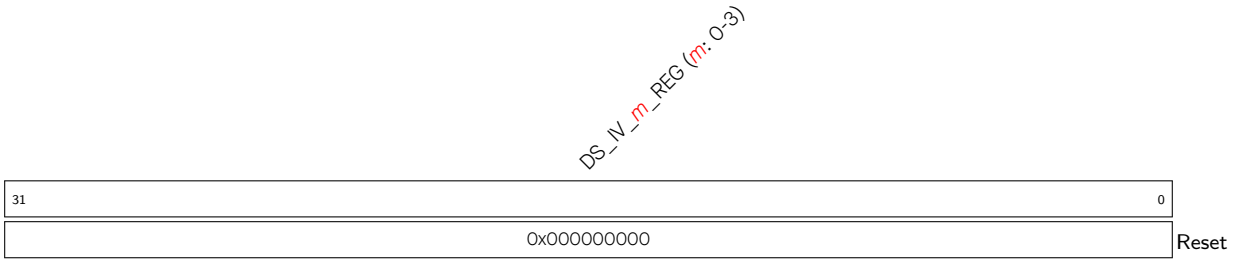
请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
配置寄存器			
DS_IV_0_REG	IV block 数据	0x0630	WO
DS_IV_1_REG	IV block 数据	0x0634	WO
DS_IV_2_REG	IV block 数据	0x0638	WO
DS_IV_3_REG	IV block 数据	0x063C	WO
状态/控制寄存器			
DS_SET_START_REG	启动 DS 模块	0x0E00	WO
DS_SET_ME_REG	开始计算	0x0E04	WO
DS_SET_FINISH_REG	结束计算	0x0E08	WO
DS_QUERY_BUSY_REG	DS 模块状态	0x0E0C	RO
DS_QUERY_KEY_WRONG_REG	查询 DS_KEY 未准备好的原因	0x0E10	RO
DS_QUERY_CHECK_REG	查询校验结果	0x0814	RO
版本寄存器			
DS_DATE_REG	版本控制寄存器	0x0820	R/W

22.6 寄存器

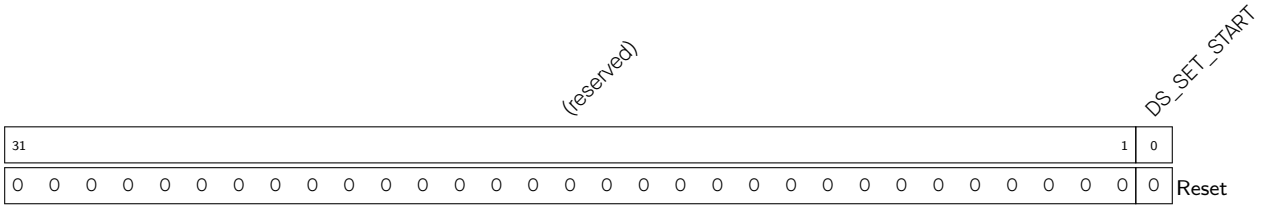
本小节的所有地址均为相对于 Digital Signature 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 22.1. DS_IV_*m*_REG (*m*: 0-3) (0x0630+4**m*)



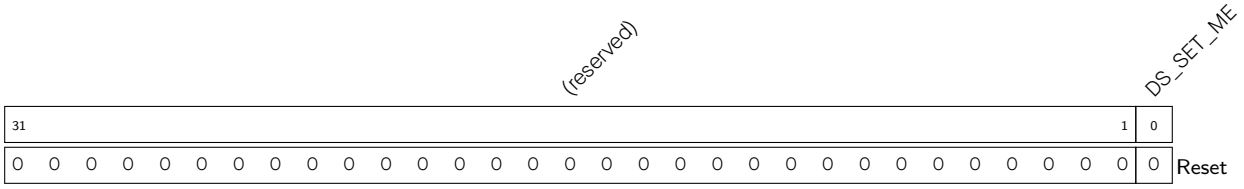
DS_IV_*m*_REG (*m*: 0-3) *IV* block 数据。(WO)

Register 22.2. DS_SET_START_REG (0x0E00)



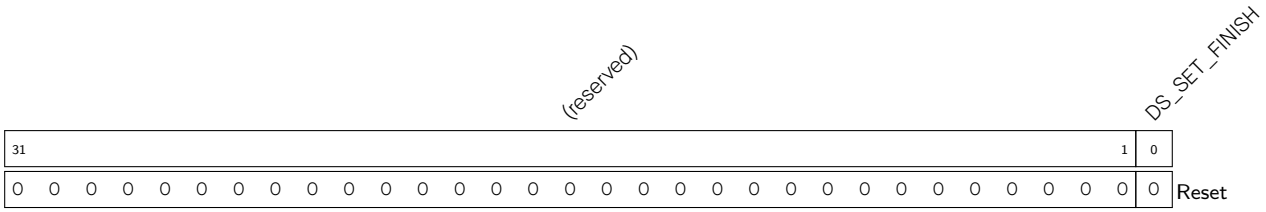
DS_SET_START 写入 1 启动 DS 模块。(WO)

Register 22.3. DS_SET_ME_REG (0x0E04)



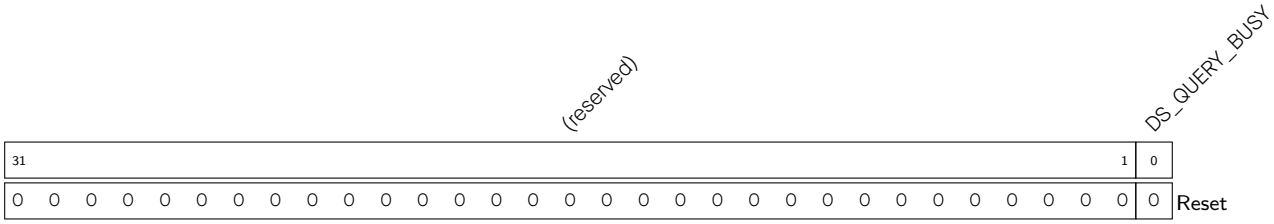
DS_SET_ME 写入 1 以开始计算。(WO)

Register 22.4. DS_SET_FINISH_REG (0x0E08)



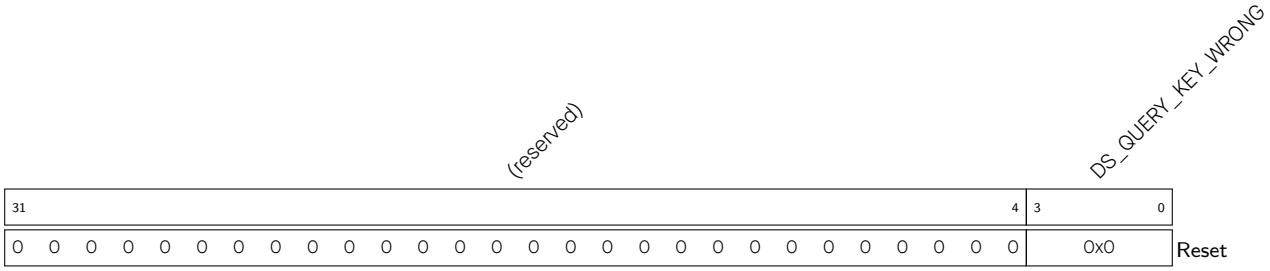
DS_SET_FINISH 写入 1 以结束运算。(WO)

Register 22.5. DS_QUERY_BUSY_REG (0x0E0C)



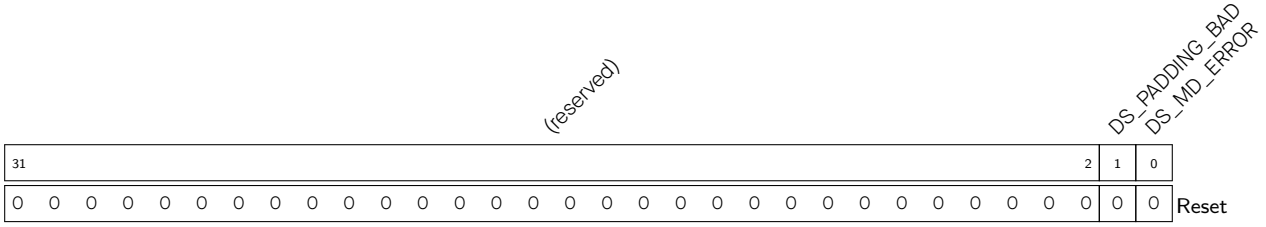
DS_QUERY_BUSY 1：DS 模块正在忙；0：DS 模块空闲。(RO)

Register 22.6. DS_QUERY_KEY_WRONG_REG (0x0E10)



DS_QUERY_KEY_WRONG 1 ~ 15：HMAC 被调用，但 DS 模块未拿到 DS_KEY（最大值为 15）；0：HMAC 未被调用。(RO)

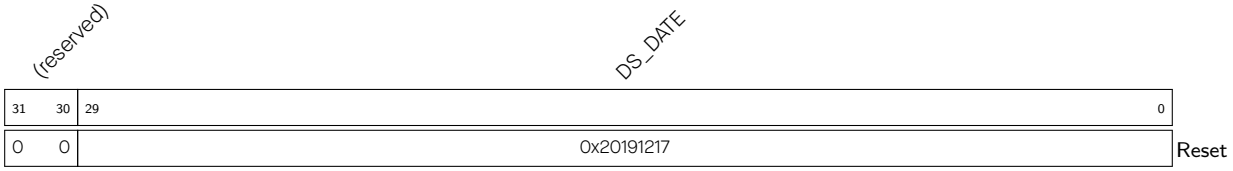
Register 22.7. DS_QUERY_CHECK_REG (0x0E14)



DS_PADDING_BAD 1：填充校验失败；0：填充校验通过。(RO)

DS_MD_ERROR 1：MD 校验失败；0：MD 校验通过。(RO)

Register 22.8. DS_DATE_REG (0x0E20)



DS_DATE 版本控制寄存器。(R/W)

第 23 章

片外存储器加密与解密 (XTS_AES)

23.1 概述

ESP32-S3 芯片集成了片外存储器加密与解密模块，采用符合 IEEE Std 1619-2007 指定的 XTS-AES 标准的算法，为用户存放在片外存储器（片外 flash 与片外 RAM）的应用代码和数据提供了安全保障。用户可以将专有软件、敏感的用户数据（如用来访问私有网络的凭据）存放在片外 flash 中，或将通用数据存放在片外 RAM 中。

23.2 主要特性

- 通用 XTS-AES 算法，符合 IEEE Std 1619-2007
- 手动加密过程需要软件参与
- 高速的自动加密过程，无需软件参与
- 高速的自动解密过程，无需软件参与
- 寄存器配置、eFuse 参数、启动 (boot) 模式共同决定加解密功能

23.3 模块结构

片外存储器加解密模块包含三个部分：手动加密 (Manual Encryption) 模块、自动加密 (Auto Encryption) 模块、自动解密 (Auto Decryption) 模块。结构图如图 23.3-1 所示。

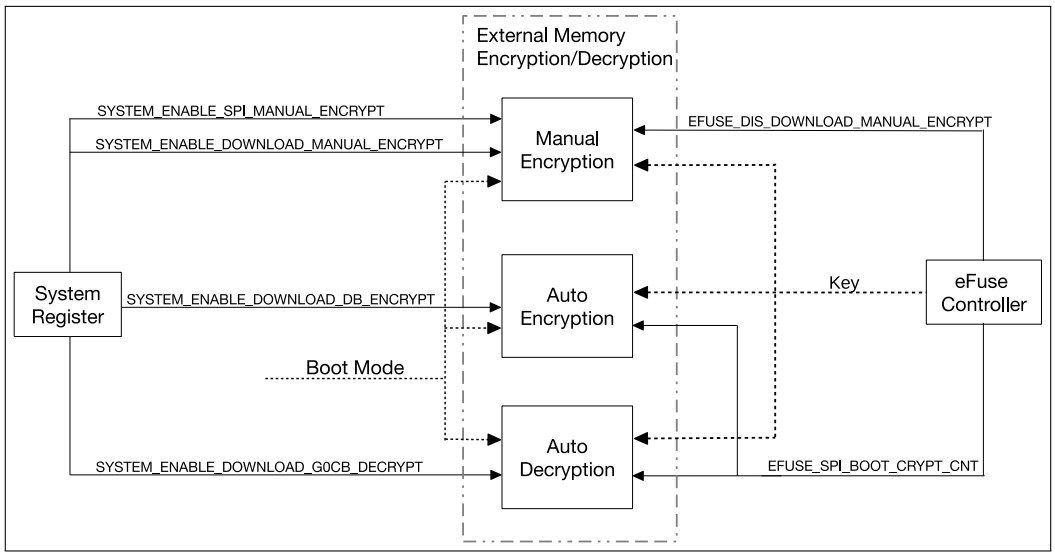


图 23.3-1. 片外存储器加解密工作配置

手动加密模块能够对指令/数据进行加密，指令/数据将以密文状态通过 SPI1 被写入片外 flash。

当 CPU 通过 cache 写片外 RAM 时，自动加密模块会先对数据自动进行加密，数据将以密文状态被写入片外 RAM。

当 CPU 通过 cache 读片外 flash 或片外 RAM 时，自动解密模块将对读取到的密文自动进行解密以恢复指令和数据。

系统寄存器 (SYSREG) 外设中 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 内的以下 4 个位与片外存储器加解密相关：

- SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT
- SYSTEM_ENABLE_DOWNLOAD_GOCB_DECRYPT
- SYSTEM_ENABLE_DOWNLOAD_DB_ENCRYPT
- SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT

片外存储器加解密模块还会从外设 eFuse 控制器中获取 2 个参数：EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT 和 EFUSE_SPI_BOOT_CRYPT_CNT。

23.4 功能描述

23.4.1 XTS 算法

不论是手动加密，还是自动加/解密，使用的是同一种算法——XTS 算法。根据算法特征，在具体实现中，使用 1024 位为一个数据单元 (data unit)，此处"data unit" 由 *XTS-AES Tweakable Block Cipher* 标准中的章节 *XTS-AES encryption procedure* 定义。更多关于 XTS-AES 算法的信息，请参考 IEEE Std 1619-2007。

23.4.2 密钥 Key

在执行 XTS 运算时，手动加密模块、自动加密模块和自动解密模块使用完全相同的密钥 Key 。密钥 Key 来自硬件 eFuse，且无法被用户访问获取。

密钥 Key 支持两种长度：256 位、512 位。Key 的值和长度完全由 eFuse 参数信息决定。为方便阐述如何通过 eFuse 参数信息推导出 Key 的值，现作如下约定：

- $Block_A$：指 BLOCK4 ~ BLOCK9 中 key purpose（密钥用途）的值等于 EFUSE_KEY_PURPOSE_XTS_AES_256_KEY_1 的 BLOCK。如果 $Block_A$ 存在，那么 $Block_A$ 中存放着 256 位的 Key_A。
- $Block_B$：指 BLOCK4 ~ BLOCK9 中密钥用途的值等于 EFUSE_KEY_PURPOSE_XTS_AES_256_KEY_2 的 BLOCK。如果 $Block_B$ 存在，那么 $Block_B$ 中存放 256 位的 Key_B。
- $Block_C$：指 BLOCK4 ~ BLOCK9 中密钥用途的值等于 EFUSE_KEY_PURPOSE_XTS_AES_128_KEY 的 BLOCK。如果 $Block_C$ 存在，那么 $Block_C$ 中存放 256 位的 Key_C。

根据 $Block_A$、$Block_B$ 和 $Block_C$ 是否存在，可以获得 5 种组合。在不同组合情况下，Key 值可以由 Key_A、Key_B、Key_C 的值惟一确定，如表 23.4-1 所示。

表 23.4-1. 根据 Key_A、Key_B、Key_C 生成 Key 值

$Block_A$	$Block_B$	$Block_C$	生成 Key 值	Key 长度（位）
Yes	Yes	无关项	$Key_A \| \| Key_B$	512
Yes	No	无关项	$Key_A \| \| 0^{256}$	512
No	Yes	无关项	$0^{256} \| \| Key_B$	512
No	No	Yes	Key_C	256
No	No	No	0^{256}	256

Notes:

表 23.4-1 中的 “Yes” 指存在，“No” 指不存在，“0^{256}” 表示 256 个位”0” 组成的位串，“||” 表示将两个比特串按照前后顺序合成一个更长的新位串。

更多有关密钥用途的信息，请参考章节 5 *eFuse 控制器 (EFUSE)* 中的表 5.3-2 密钥用途数值对应的含义。

23.4.3 目标空间

目标空间是指单次加密后的密文将要存放在片外存储器中的哪一段连续的地址空间中。目标空间可以由目标类型、目标尺寸、目标基地址这三个参数唯一确定。这三个参数的定义如下：

- 目标类型：目标空间的类型 ($type$)，片外 flash 或片外 RAM。目标类型的值为 0 时指片外 flash，值为 1 时指片外 RAM。
- 目标尺寸：目标空间的大小 ($size$)，以字节为单位，即单次对多少数据进行加密。只有 16、32 和 64 可选。
- 目标基地址：目标空间的基地址 ($base\_addr$)，这是一个 30-bit 的物理地址，取值范围为 0x0000_0000 ~ 0x3FFF_FFFF，但要求以目标尺寸为单位对齐，即 $base\_addr \% size == 0$。

如某一次加密操作，要将 16 字节的指令数据加密后存放在片外 flash 中的地址段 0x130 ~ 0x13F 中，则目标空间为 0x130 ~ 0x13F，目标类型为 0（片外 flash），目标尺寸为 16 字节，目标基地址为 0x130。

对于任意长度（必须是 16 字节的整数倍）的明文指令/数据的加密，可以将整个加密过程拆分成多次进行，每次都有各自的目标空间参数。

对于自动加/解密模块，目标空间等参数由硬件自动调节。对于手动加密模块，目标空间等参数需要用户主动配置。

说明：

IEEE Std 1619-2007 中的章节 *5.1 Data units and tweaks* 中定义的 “tweak” 是一个 128-bit 的非负整数 ($tweak$)，其值可以通过公式求出：$tweak$ = $type * 2^{30}$ + ($base\_addr$ & 0x3FFFFF80)。显然，$tweak$ 中最低的 7 个比特和最高的 97 个比特恒为零。

23.4.4 数据填充

对于自动加/解密模块，数据的填充由硬件自动完成。对于手动加密模块，数据的填充需要用户主动配置。手动加密模块中由 16 个寄存器 XTS_AES_PLAIN_*n*_REG (*n*: 0-15) 构成的寄存器块专用于数据填充，一次可以存放最多 512 位明文指令/数据。

实际上，手动加密模块不在乎明文来自什么地方，只注重密文将要存放在什么地方。考虑到明文和密文之间呈严格的对应关系，为了更好地描述明文如何封装在寄存器块中，现假设明文从一开始就放在目标空间中，并在

加密完成后被密文替换。因此，接下来的描述不再出现“明文”这个概念，而用“目标空间”来代替。但请注意，在真正使用时，明文可以来自任何地方，但用户必须清晰知道明文如何封装在寄存器块中。

目标空间映射到寄存器块的方法为：

假设目标空间中某一 word 的存放地址为 $address$，记 $offset = address\%64$，$n = \frac{offset}{4}$，那么该 word 将被存放在编号为 n 的寄存器 XTS_AES_PLAIN_n_REG 中。

例如，当目标尺寸为 64 时，寄存器块中的所有寄存器都将被使用，目标空间中的地址与寄存器块之间的填充映射关系如表 23.4-2 所示。

表 23.4-2. 目标空间与寄存器堆的映射关系

$offset$	寄存器	$offset$	寄存器
0x00	XTS_AES_PLAIN_0_REG	0x20	XTS_AES_PLAIN_8_REG
0x04	XTS_AES_PLAIN_1_REG	0x24	XTS_AES_PLAIN_9_REG
0x08	XTS_AES_PLAIN_2_REG	0x28	XTS_AES_PLAIN_10_REG
0x0C	XTS_AES_PLAIN_3_REG	0x2C	XTS_AES_PLAIN_11_REG
0x10	XTS_AES_PLAIN_4_REG	0x30	XTS_AES_PLAIN_12_REG
0x14	XTS_AES_PLAIN_5_REG	0x34	XTS_AES_PLAIN_13_REG
0x18	XTS_AES_PLAIN_6_REG	0x38	XTS_AES_PLAIN_14_REG
0x1C	XTS_AES_PLAIN_7_REG	0x3C	XTS_AES_PLAIN_15_REG

23.4.5 手动加密模块

手动加密模块是一个外设模块，自身带有寄存器，可以被 CPU 直接访问。模块内的寄存器、系统寄存器 (SYSREG) 外设、eFuse 参数、boot 模式共同配置并使用这一模块。请注意，手动加密模块只能加密片外 flash。

当且仅当手动加密模块拥有工作权限时，才允许手动加密。手动加密模块是否拥有工作权限取决于：

- SPI Boot 模式下

  当寄存器 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 的 SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT 位为 1 时，手动加密模块拥有工作权限，否则无法工作。

- Download Boot 模式下

  当寄存器 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 的 SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT 位为 1，且 eFuse 参数 EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT 为 0 时，手动加密模块拥有工作权限，否则无法工作。

说明：

- 虽然 CPU 可以不通过 cache 而直接读片外存储器从而得到加密指令/数据，但软件还是绝对无法获取到密钥 Key。

23.4.6 自动加密模块

自动加密并非外设模块，自身不带寄存器，不能被 CPU 直接访问。系统寄存器 (SYSREG) 外设、eFuse 参数、boot 模式共同配置并使用这一模块。

当且仅当自动加密模块拥有工作权限时，才允许自动加密。自动加密模块是否拥有工作权限取决于：

- SPI Boot 模式下

  当参数 SPI_BOOT_CRYPT_CNT（3 位宽）中有奇数个位为 1 时，自动加密模块拥有工作权限，否则无法工作。

- Download Boot 模式下

  当寄存器 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 的 SYSTEM_ENABLE_DOWNLOAD_DB_ENCRYPT 位为 1 时，自动加密模块拥有工作权限，否则无法工作。

说明：

- 当自动加密模块拥有工作权限时，如果 CPU 通过 cache 写访问片外 RAM，自动加密模块将自动对数据进行加密，然后将加密结果写到片外 RAM。加密的整个过程无需软件参与并且对 cache 是透明的。加密算法过程中密钥 Key 绝对无法被用户获取。
- 当自动加密模块没有工作权限时，自动加密模块将不理睬 CPU 对 cache 的访问请求，也不对数据做任何处理，因此数据将以明文状态被直接写到片外 RAM。

23.4.7 自动解密模块

自动解密并非外设模块，自身不带寄存器，不能被 CPU 直接访问。系统寄存器 (SYSREG) 外设、eFuse 参数、boot 模式共同配置并使用这一模块。

当且仅当自动解密模块拥有工作权限时，才允许自动解密。自动解密模块是否拥有工作权限取决于：

- SPI Boot 模式下

  当参数 SPI_BOOT_CRYPT_CNT（3 位宽）中有奇数个位为 1 时，自动解密模块拥有工作权限，否则无法工作。

- Download Boot 模式下

  当寄存器 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 的 SYSTEM_ENABLE_DOWNLOAD_GOCB_DECRYPT 位为 1 时，自动解密模块拥有工作权限，否则无法工作。

说明：

- 当自动解密模块拥有工作权限时，如果 CPU 通过 cache 读取片外存储器中的指令/数据，自动解密将自动对读取到的密文进行解密以恢复指令/数据。解密的整个过程无需软件参与并且对 cache 是透明的。解密算法过程中密钥 Key 绝对无法被用户获取。
- 当自动解密模块没有工作权限时，自动解密模块不对片外存储器中的内容产生作用，无论是加密内容还是未加密内容，因此 CPU 通过 cache 读取到的是片外存储器中的原始内容。

23.5 软件流程

手动加密模块工作时需要软件参与，软件流程为：

1. 配置 XTS_AES：
   - 将寄存器 XTS_AES_DESTINATION_REG 的值设置为 $type$ = 0。

- 将寄存器 XTS_AES_PHYSICAL_ADDRESS_REG 的值设置为 $base\_addr$。
- 将寄存器 XTS_AES_LINESIZE_REG 的值设置为 $\frac{size}{32}$。

关于 $type$、$base\_addr$、$size$ 的定义，请参考章节 23.4.3。

2. 用明文填充寄存器块 XTS_AES_PLAIN_*n*_REG (*n*: 0-15)。请参考章节 23.4.4 获取相关信息。
只需要根据需要填充寄存器，不需要使用的寄存器可以是任意值。
3. 轮询寄存器 XTS_AES_STATE_REG 直到读到 0，确保手动加密模块是空闲的。
4. 启动计算。对寄存器 XTS_AES_TRIGGER_REG 写入 1。
5. 等待加密完成。轮询寄存器 XTS_AES_STATE_REG，直到读到 2。
步骤 1 至 5 操作手动加密模块对明文指令进行加密。加密算法使用的密钥就是 Key 。
6. 下放密文访问权限给 SPI1。对寄存器 XTS_AES_RELEASE_REG 写入 1，使得加密结果允许被 SPI1 获取。
之后如果读取寄存器 XTS_AES_STATE_REG 将读到 3。
7. 调用 SPI1，将加密结果写入片外 flash（请参阅章节 30 *SPI 控制器 (SPI)*）。
8. 销毁加密结果。对寄存器 XTS_AES_DESTROY_REG 写入 1。之后如果读取寄存器 XTS_AES_STATE_REG 将读到 0。

重复上述步骤，即可满足明文指令/数据的加密需求。

23.6 寄存器列表

本小节的所有地址均为相对于 External Memory Encryption and Decryption 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

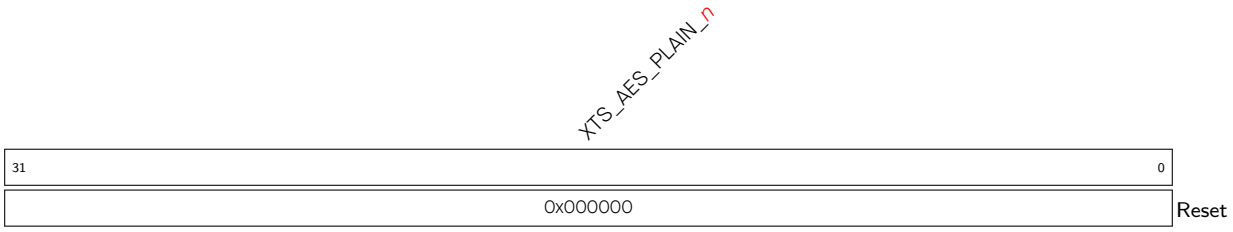
请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
明文寄存器堆			
XTS_AES_PLAIN_0_REG	明文寄存器 0	0x0000	R/W
XTS_AES_PLAIN_1_REG	明文寄存器 1	0x0004	R/W
XTS_AES_PLAIN_2_REG	明文寄存器 2	0x0008	R/W
XTS_AES_PLAIN_3_REG	明文寄存器 3	0x000C	R/W
XTS_AES_PLAIN_4_REG	明文寄存器 4	0x0010	R/W
XTS_AES_PLAIN_5_REG	明文寄存器 5	0x0014	R/W
XTS_AES_PLAIN_6_REG	明文寄存器 6	0x0018	R/W
XTS_AES_PLAIN_7_REG	明文寄存器 7	0x001C	R/W
XTS_AES_PLAIN_8_REG	明文寄存器 8	0x0020	R/W
XTS_AES_PLAIN_9_REG	明文寄存器 9	0x0024	R/W
XTS_AES_PLAIN_10_REG	明文寄存器 10	0x0028	R/W
XTS_AES_PLAIN_11_REG	明文寄存器 11	0x002C	R/W
XTS_AES_PLAIN_12_REG	明文寄存器 12	0x0030	R/W
XTS_AES_PLAIN_13_REG	明文寄存器 13	0x0034	R/W
XTS_AES_PLAIN_14_REG	明文寄存器 14	0x0038	R/W
XTS_AES_PLAIN_15_REG	明文寄存器 15	0x003C	R/W
配置寄存器			
XTS_AES_LINESIZE_REG	加密块大小寄存器	0x0040	R/W
XTS_AES_DESTINATION_REG	加密类型寄存器	0x0044	R/W
XTS_AES_PHYSICAL_ADDRESS_REG	物理地址寄存器	0x0048	R/W
控制/状态寄存器			
XTS_AES_TRIGGER_REG	启动运算	0x004C	WO
XTS_AES_RELEASE_REG	释放控制	0x0050	WO
XTS_AES_DESTROY_REG	销毁控制	0x0054	WO
XTS_AES_STATE_REG	状态寄存器	0x0058	RO
版本寄存器			
XTS_AES_DATE_REG	版本控制寄存器	0x005C	RO

23.7 寄存器

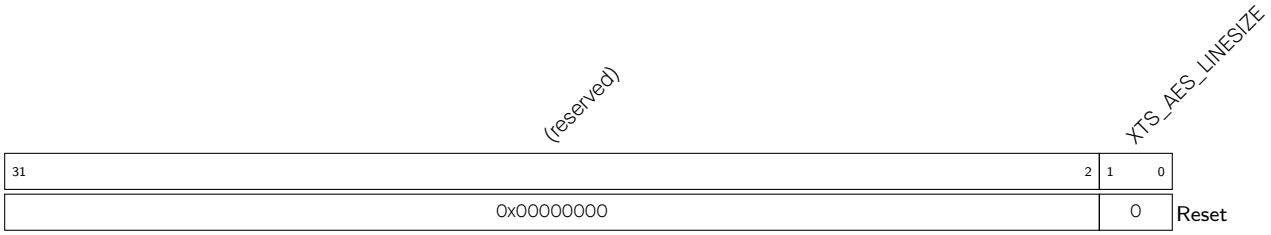
本小节的所有地址均为相对于 External Memory Encryption and Decryption 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 23.1. XTS_AES_PLAIN_*n*_REG (*n*: 0-15) (0x0000+4**n*)



XTS_AES_PLAIN_*n* 存储明文的第 *n* 个 32 位部分。(R/W)

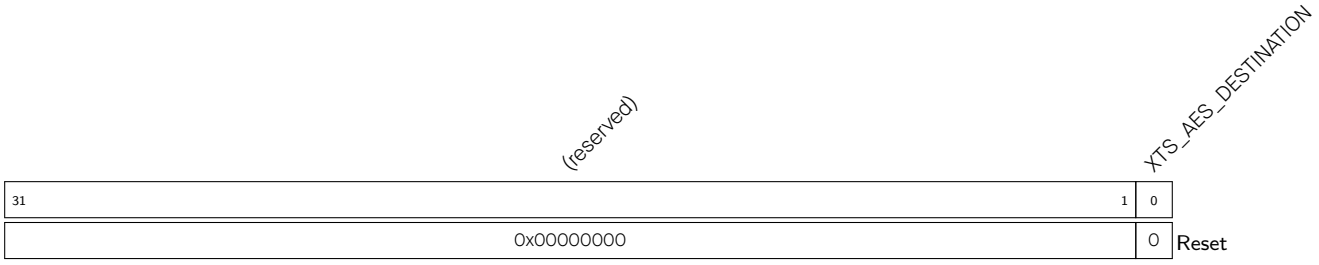
Register 23.2. XTS_AES_LINESIZE_REG (0x0040)



XTS_AES_LINESIZE 块大小寄存器，决定单次加密的数据量。

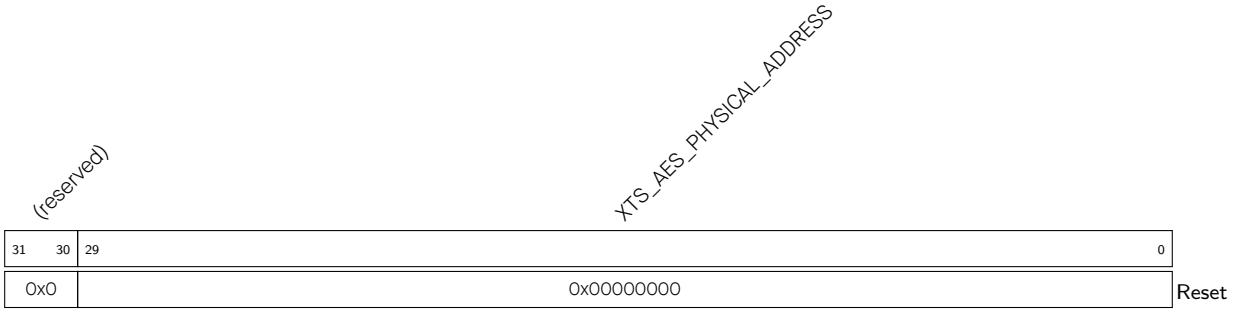
- 0：加密 16 bytes；
- 1：加密 32 bytes；
- 2：加密 64 bytes。(R/W)

Register 23.3. XTS_AES_DESTINATION_REG (0x0044)



XTS_AES_DESTINATION 决定手动加密类型，目前只能手动加密 flash，所以只能为 0。用户不能写入 1，否则将发生错误。0：加密 flash；1：加密片外 RAM。(R/W)

Register 23.4. XTS_AES_PHYSICAL_ADDRESS_REG (0x0048)



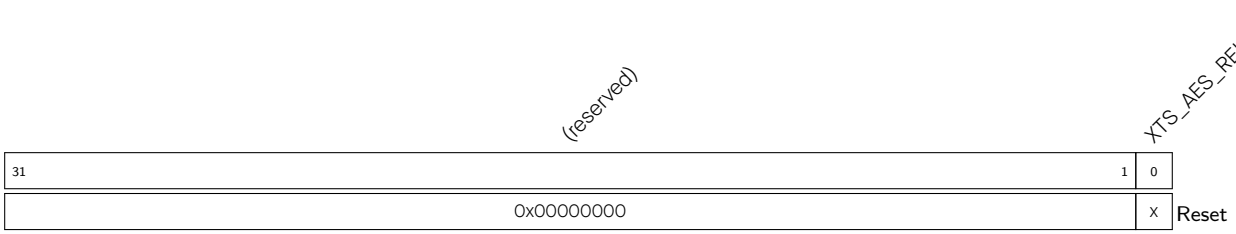
XTS_AES_PHYSICAL_ADDRESS 物理地址。(R/W)

Register 23.5. XTS_AES_TRIGGER_REG (0x004C)



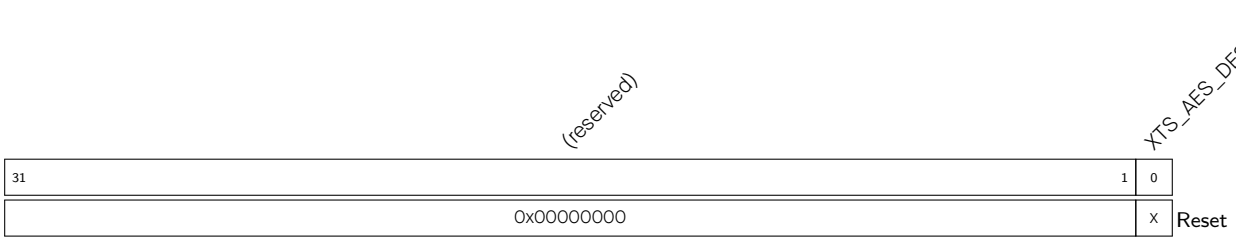
XTS_AES_TRIGGER 写入 1 使能手动加密运算。(WO)

Register 23.6. XTS_AES_RELEASE_REG (0x0050)



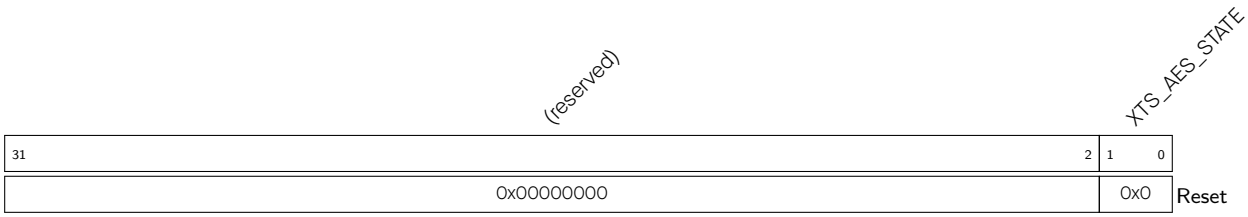
XTS_AES_RELEASE 写入 1 使加密结果对 SPI1 可见，因而 SPI1 可以获取到加密结果。(WO)

Register 23.7. XTS_AES_DESTROY_REG (0x0054)



XTS_AES_DESTROY 写入 1 销毁加密结果。(WO)

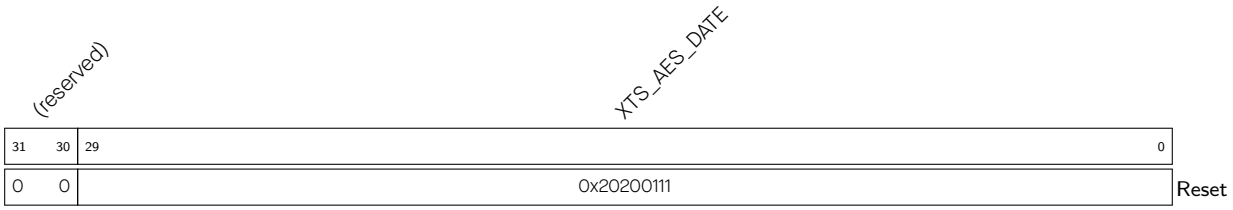
Register 23.8. XTS_AES_STATE_REG (0x0058)



XTS_AES_STATE 手动加密模块状态寄存器。

- 0x0 (XTS_AES_IDLE)：空闲；
- 0x1 (XTS_AES_BUSY)：忙于计算；
- 0x2 (XTS_AES_DONE)：计算完成，但手动加密结果数据对 SPI 不可见；
- 0x3 (XTS_AES_RELEASE)：手动加密结果对 SPI 可见。(RO)

Register 23.9. XTS_AES_DATE_REG (0x005C)



XTS_AES_DATE 版本控制寄存器。(R/W)

第 24 章

时钟毛刺检测

24.1 概述

为提升 ESP32-S3 的安全性能，防止攻击者通过给外部晶振 XTAL_CLK 附加毛刺，使芯片进入异常状态，从而实施对芯片的攻击，ESP32-S3 搭载了毛刺检测模块 (CLK Glitch_Detect)，用于检测从外部晶振输入的 XTAL_CLK 是否携带毛刺，并在检测到毛刺后，产生数字系统复位信号，复位包括 RTC 在内的整个数字电路（请参考章节 7 复位和时钟）。

24.2 功能描述

24.2.1 时钟毛刺检测

ESP32-S3 的毛刺检测模块将对输入芯片的 XTAL_CLK 时钟信号进行检测，当时钟的脉宽（a 或 b）小于 3 ns 时，将认为检测到毛刺，触发毛刺检测信号，屏蔽输入的 XTAL_CLK 时钟信号。

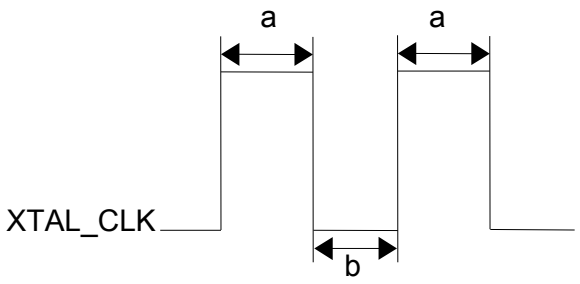


图 24.2-1. XTAL_CLK 脉宽

24.2.2 复位

当时钟毛刺检测电路检测到 XTAL_CLK 上有影响电路正常工作的毛刺之后，如果 RTC_CNTL_GLITCH_RST_EN 位为 1，将触发系统级复位。该位默认为开启复位状态。

第 25 章

随机数发生器 (RNG)

25.1 概述

ESP32-S3 内置一个真随机数发生器，其生成的 32 位随机数可作为加密等操作的基础。

25.2 主要特性

ESP32-S3 的随机数发生器可通过物理过程而非算法生成真随机数，所有生成的随机数在特定范围内出现的概率完全一致。

25.3 功能描述

系统可以从随机数发生器的寄存器 RNG_DATA_REG 中读取随机数，每个读到的 32 位随机数都是真随机数，噪声源为系统中的热噪声和异步时钟。

具体来说，这些热噪声可以来自 SAR ADC 或高速 ADC 或两者兼有。当芯片的 SAR ADC 或高速 ADC 工作时，就会产生比特流，并通过异或 (XOR) 逻辑运算作为随机数种子进入随机数生成器。

当为数字内核使能 RC_FAST_CLK 时钟时，随机数发生器也会对 RC_FAST_CLK (20 MHz) 进行采样，作为随机数种子。RC_FAST_CLK 是一种异步时钟源，由于存在电路亚稳态，因此可以提高随机数发生器的熵值。然而，为了保证随机数发生器可以获得最大熵值，仍建议在使用随机数发生器时至少使能一路 ADC 作为随机数种子。

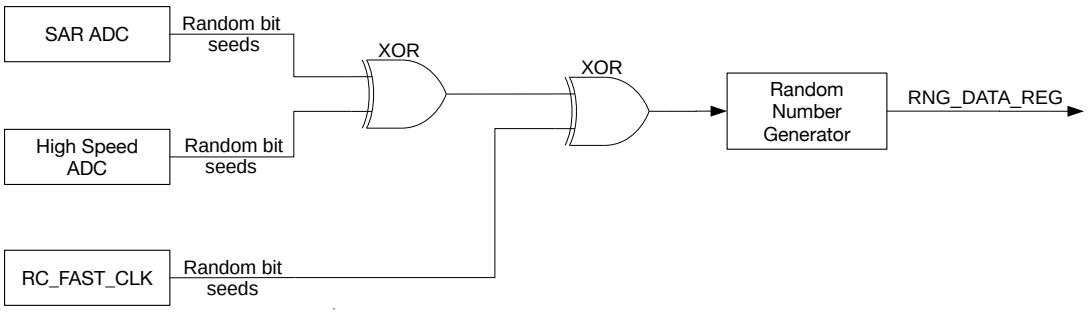


图 25.3-1. 噪声源

当 SAR ADC 打开时，每个 RC_FAST_CLK (20 MHz) 时钟周期内（来自内部 RC 振荡器，详见 7 复位和时钟 章节），随机数发生器将获得 2 位的熵。因此，为了获得最大的熵值，建议读取 RNG_DATA_REG 寄存器时的速率不超过 500 kHz。

当高速 ADC 打开时，每个 APB 时钟周期（通常为 80 MHz）内，随机数发生器将获得 2 位的熵。因此，为了获得最大的熵值，建议读取 RNG_DATA_REG 寄存器时的速率不超过 5 MHz。

25.4 编程指南

在使用 ESP32-S3 的随机数生成器时，应该至少打开 SAR ADC，高速 ADC，或 RC_FAST_CLK，否则可能会导致产生伪随机数，应注意避免。其中，

- SAR ADC 可通过 DIG ADC 控制器打开，详见 39 片上传感器与模拟信号处理 章节。
- 高速 ADC 在 Wi-Fi 或蓝牙开启时自动打开。
- RC_FAST_CLK 可通过设置 RTC_CNTL_CLK_CONF_REG 寄存器中的 RTC_CNTL_DIG_CLK8M_EN 位使能。

说明：
注意，在 Wi-Fi 开启时，极端情况下高速 ADC 有读值饱和的可能，这会降低熵值。因此，建议在 Wi-Fi 开启时，同时通过 DIG ADC1 控制器打开 SAR ADC 产生随机数。

在使用随机数生成器时，请多次读取 RNG_DATA_REG 寄存器的值，直至获得足够多的随机数。在读取寄存器时，注意控制速率不要超过上方第 25.3 小节的介绍。

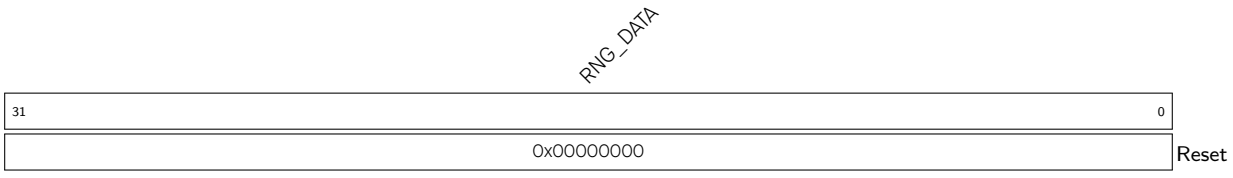
25.5 寄存器列表

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
RNG_DATA_REG	随机数数据	0x6003_507C	只读

25.6 寄存器

Register 25.1. RNG_DATA_REG (0x6003_507C)



RNG_DATA 随机数来源。(只读)

第 V 卷

通讯接口

该部分涉及系统的连接性，介绍各种带有通讯接口的模块，如 I2C、I2S、SPI、UART、USB 等，同时还包括其他用于遥控、电机控制、LED 控制等功能的信号生成接口。

第 26 章

UART 控制器 (UART)

26.1 概述

嵌入式应用通常要求一个简单的并且占用系统资源少的方法来传输数据。通用异步收发传输器 (UART) 即可以满足这些要求，它能够灵活地与外部设备进行全双工数据交换。ESP32-S3 芯片中有三个 UART 控制器可供使用，并且兼容不同的 UART 设备。另外，UART 还可以用作红外数据交换 (IrDA) 或 RS485 调制解调器。

三个 UART 控制器分别有一组功能相同的寄存器。本文以 UART*n* 指代三个 UART 控制器，*n* 为 0、1、2。

UART 是一种以字符为导向的通用数据链，可以实现设备间的通信。异步传输的意思是不需要在发送数据上添加时钟信息。这也要求发送端和接收端的速率、停止位、奇偶校验位等都要相同，通信才能成功。

一个典型的 UART 帧开始于一个起始位，紧接着是有效数据，然后是奇偶校验位（可有可无），最后是停止位。ESP32-S3 芯片上的 UART 控制器支持多种字符长度和停止位。另外，控制器还支持软硬件流控和 GDMA，可以实现无缝高速的数据传输。开发者可以使用多个 UART 端口，同时又能保证很少的软件开销。

26.2 主要特性

UART 控制器具有如下特性：

- 支持三个可预分频的时钟源
- 可编程收发波特率
- 三个 UART 的发送 FIFO 以及接收 FIFO 共享 1024 x 8-bit RAM
- 全双工异步通信
- 支持输入信号波特率自检功能
- 支持 5/6/7/8 位数据长度
- 支持 1/1.5/2 个停止位
- 支持奇偶校验位
- 支持 AT_CMD 特殊字符检测
- 支持 RS485 协议
- 支持 IrDA 协议
- 支持 GDMA 高速数据通信
- 支持 UART 唤醒模式
- 支持软件流控和硬件流控

26.3 UART 架构

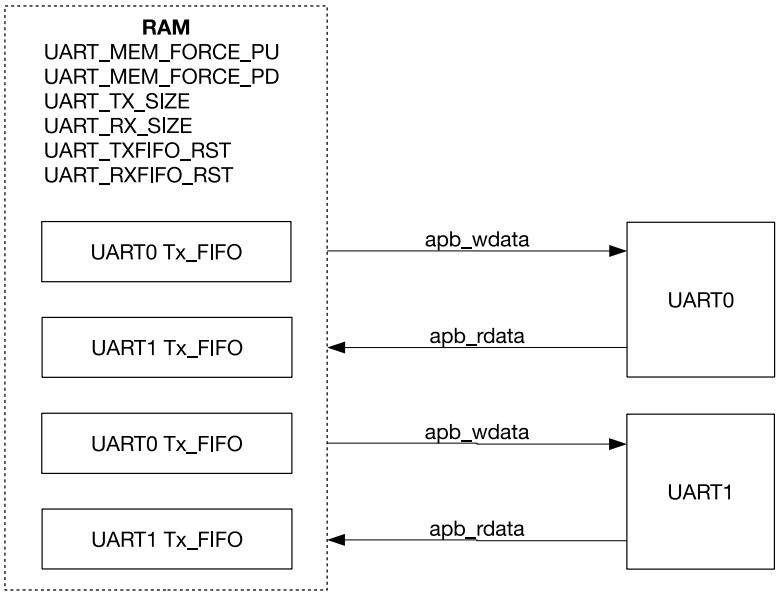


图 26.3-1. UART 架构概况

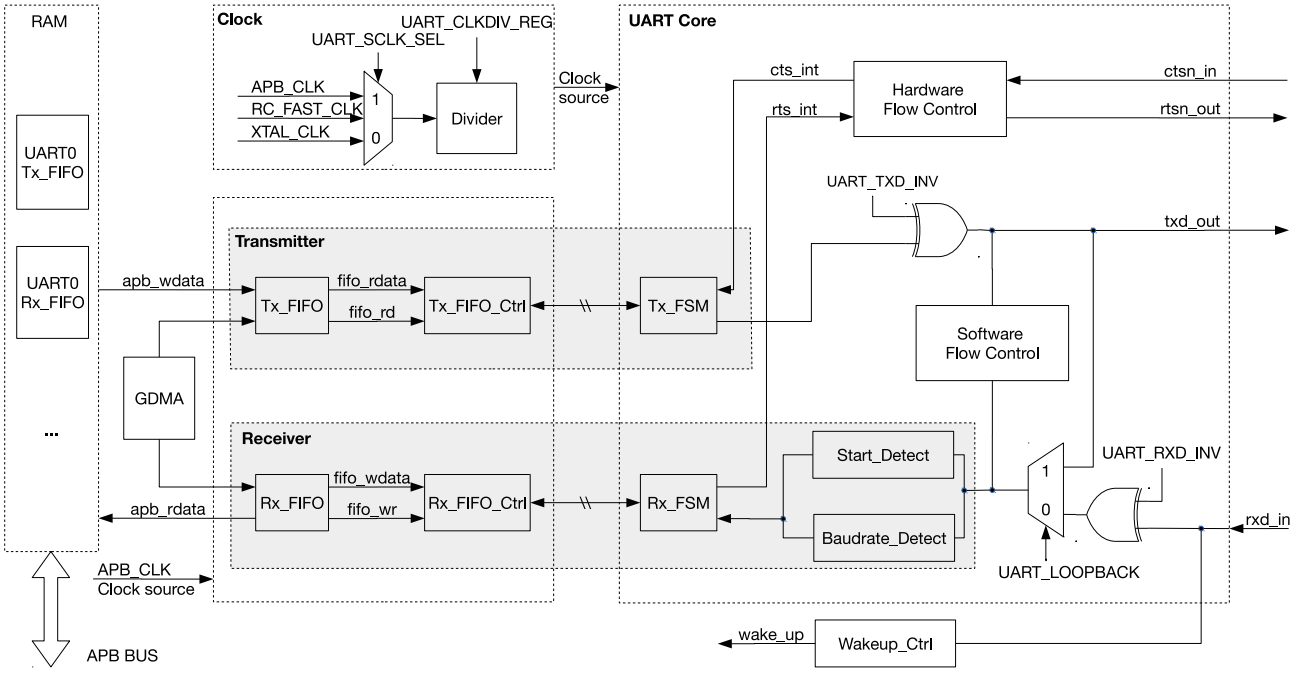


图 26.3-2. UART 基本架构图

图 26.3-2 为 UART 基本架构图。UART 模块工作在两个时钟域：APB_CLK 时钟域和 Core 时钟域。UART Core 有三个时钟源：80-MHz APB_CLK、RC_FAST_CLK 以及晶振时钟 XTAL_CLK（详情请参考章节 7 复位和时钟）。可以通过配置 UART_SCLK_SEL 来选择时钟源。分频器用于对时钟源进行分频，然后产生时钟信号来驱动 UART Core 模块。

UART 控制器可以分为两个功能块：发送块和接收块。

发送块包含一个发送 FIFO 用于缓存待发送的数据。软件可以通过 APB 总线向 Tx_FIFO 写数据，也可以通过

GDMA 将数据搬入 Tx_FIFO。Tx_FIFO_Ctrl 用于控制 Tx_FIFO 的读写过程，当 Tx_FIFO 非空时，Tx_FSM 通过 Tx_FIFO_Ctrl 读取数据，并将数据按照配置的帧格式转化成比特流。比特流输出信号 txd_out 可以通过配置 UART_TXD_INV 寄存器实现取反功能。

接收块包含一个接收 FIFO 用于缓存待处理的数据。输入比特流 rxd_in 可以输入到 UART 控制器。可以通过 UART_RXD_INV 寄存器实现取反。Baudrate_Detect 通过检测最小比特流输入信号的脉宽来测量输入信号的波特率。Start_Detect 用于检测数据的 START 位，当检测到 START 位之后，Rx_FSM 通过 Rx_FIFO_Ctrl 将帧解析后的数据存入 Rx_FIFO 中。软件可以通过 APB 总线读取 Rx_FIFO 中的数据也可以使用 GDMA 方式进行数据接收。

HW_Flow_Ctrl 通过标准 UART RTS 和 CTS（rtsn_out 和 ctsn_in）流控信号来控制 rxd_in 和 txd_out 的数据流。SW_Flow_Ctrl 通过在发送数据流中插入特殊字符以及在接收数据流中检测特殊字符来进行数据流的控制。

当 UART 处于 Light-sleep 状态（详情请参考章节 10 低功耗管理 (RTC_CNTL)）时，Wakeup_Ctrl 开始计算 rxd_in 的上升沿个数，当上升沿个数大于等于（UART_ACTIVE_THRESHOLD + 3）时产生 wake_up 信号给 RTC 模块，由 RTC 来唤醒 ESP32-S3 芯片。

26.4 功能描述

26.4.1 时钟与复位

UART 为异步外设。其寄存器配置模块与 TX/RX FIFO 工作在 APB_CLK 时钟域，而控制 UART 发送与接收的 Core 模块工作在 UART Core 时钟域。UART Core 有三个时钟源：APB_CLK，RC_FAST_CLK 以及晶振时钟 XTAL_CLK，可通过配置寄存器 UART_SCLK_SEL 来选择时钟源。选择后的时钟源通过预分频器分频后进入 UART Core 模块。该预分频器支持小数分频，分频系数为：

$$UART\_SCLK\_DIV\_NUM + \frac{UART\_SCLK\_DIV\_B}{UART\_SCLK\_DIV\_A}$$

支持的分频范围为：1 ~ 256。

若分频之后的 Core 时钟频率还能满足生成波特率的需求，可通过预分频使 UART Core 模块工作在较小的时钟频率，从而减小 UART 外设的功耗。通常情况下，UART Core 模块时钟小于 APB_CLK 时钟，并且在满足 UART 波特率的情况下，UART Core 时钟分频系数可以配置到最大值。UART 也支持 UART Core 模块时钟大于 APB_CLK 时钟，此时，UART Core 模块时钟最大为 APB_CLK 的 3 倍。另外，UART TX/RX 的 Core 时钟可以被单独控制。置位 UART_TX_SCLK_EN 使能 UART TX 的 Core 时钟；置位 UART_RX_SCLK_EN 使能 UART RX 的 Core 时钟。

为确保配置寄存器的值成功从 APB_CLK 时钟域同步到 UART Core 时钟域，寄存器配置需要遵循一定的流程，详情请参考章节26.5。

对整个 UART 的复位，需要遵循一定的配置流程，详情请参考章节26.5.2.1。
注：不推荐单独复位 UART APB 模块或者 UART Core 模块。

26.4.2 UART RAM

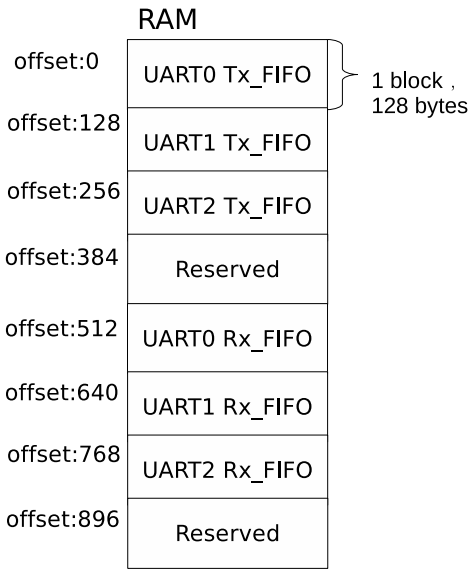


图 26.4-1. UART 共享 RAM 图

ESP32-S3 芯片中三个 UART 控制器共用 1024 x 8-bit RAM 空间。如图 26.4-1 所示，RAM 以 block 为单位进行分配，1 block 为 128x8 bits。图 26.4-1 所示为默认情况下三个 UART 控制器的 Tx_FIFO 和 Rx_FIFO 占用 RAM 的情况。通过配置 UART_TX_SIZE 可以对 UART*n* 的 Tx_FIFO 以 1 block 为单位进行扩展，通过配置 UART_RX_SIZE 可以对 UART*n* 的 Rx_FIFO 以 1 block 为单位进行扩展：

- UART0 Tx_FIFO 可以从地址 0 扩展到整个 RAM 空间；
- UART1 Tx_FIFO 可以从地址 128 扩展到 RAM 的尾地址；
- UART2 Tx_FIFO 可以从地址 256 扩展到 RAM 的尾地址；
- UART0 Rx_FIFO 可以从地址 512 扩展到 RAM 的尾地址；
- UART1 Rx_FIFO 可以从地址 640 扩展到 RAM 的尾地址；
- UART2 Rx_FIFO 可以从地址 768 扩展到 RAM 的尾地址。

需要注意的是所有 UART 的 FIFO 起始地址是固定的，因此前一个 UART 的 FIFO 空间向后扩展会占用后面 UART 的 FIFO 空间。比如，设置 UART0 的 UART_TX_SIZE 为 2，则 UART0 Tx_FIFO 的地址从 0 扩展到 255。这时，UART1 Tx_FIFO 的默认空间被占用，这时将不能使用 UART1 发送器功能。

当三个 UART 控制器都不工作时，可以通过置位 UART_MEM_FORCE_PD 来使 RAM 进入低功耗状态。

UART*n* 的 Tx_FIFO 可以通过置位 UART_TXFIFO_RST 来复位，UART*n* 的 Rx_FIFO 可以通过置位 UART_RXFIFO_RST 来复位。

配置 UART_TXFIFO_EMPTY_THRHD 可以设置 Tx_FIFO 空信号阈值，当存储在 Tx_FIFO 中的数据量等于或小于 UART_TXFIFO_EMPTY_THRHD 时会产生中断 UART_TXFIFO_EMPTY_INT；配置 UART_RXFIFO_FULL_THRHD 可以设置 Rx_FIFO 满信号阈值，当储存在 Rx_FIFO 中的数据量大于 UART_RXFIFO_FULL_THRHD 会产生中断 UART_RXFIFO_FULL_INT。另外，当 Rx_FIFO 中储存的数据量超过其能存储的最大值时，会产生 UART_RXFIFO_OVF_INT 中断。

对于 TX FIFO 和 RX FIFO 的访问，可以通过 APB 总线或者 GDMA 外设接口两种方式。通过 APB 总线访问，即通过寄存器 UART_FIFO_REG 访问 FIFO。您可以写 UART_RXFIFO_RD_BYTE 将数据存入 TX FIFO，也可以读取该字段获取 RX FIFO 中的数据。通过 GDMA 的外设接口访问 TX FIFO 和 RX FIFO 请参考章节 26.4.11。

26.4.3 波特率产生与检测

26.4.3.1 波特率产生

在 UART 发送或接收数据之前，需要配置寄存器来设置波特率。波特率发生器主要通过对输入时钟源的分频来实现，支持小数分频。UART_CLKDIV_REG 将分频系数分成两个部分：UART_CLKDIV 用于配置整数部分，UART_CLKDIV_FRAG 用于配置小数部分。在输入时钟为 80 MHz 的情况下，UART 能支持的最大波特率为 5 MBaud。

波特率发生器分频系数为：

$$UART\_CLKDIV + \frac{UART\_CLKDIV\_FRAG}{16}$$

也就是说，最终波特率为

$$\frac{INPUT\_FREQ}{UART\_CLKDIV + \frac{UART\_CLKDIV\_FRAG}{16}}$$

其中，INPUT_FREQ 为 UART Core 时钟。例如，若 UART_CLKDIV = 694，UART_CLKDIV_FRAG = 7，则分频系数为

$$694 + \frac{7}{16} = 694.4375$$

UART_CLKDIV_FRAG 为 0 时，分频器为整数分频，每 UART_CLKDIV 个输入脉冲都会产生一个输出脉冲。

UART_CLKDIV_FRAG 不为 0 时，分频器为小数分频，输出波特率脉冲不完全统一。如图 26.4-2 所示，每 16 个输出脉冲中，部分脉冲的频率是 INPUT_FREQ/(UART_CLKDIV + 1)，剩余脉冲的频率是 INPUT_FREQ/UART_CLKDIV。分频 (UART_CLKDIV + 1) 个输入脉冲产生 UART_CLKDIV_FRAG 个输出脉冲，分频 UART_CLKDIV 个输入脉冲产生剩余的 (16 - UART_CLKDIV_FRAG) 个输出脉冲。

如图 26.4-2 所示，输出脉冲相互交错，使得输出时序更加统一。

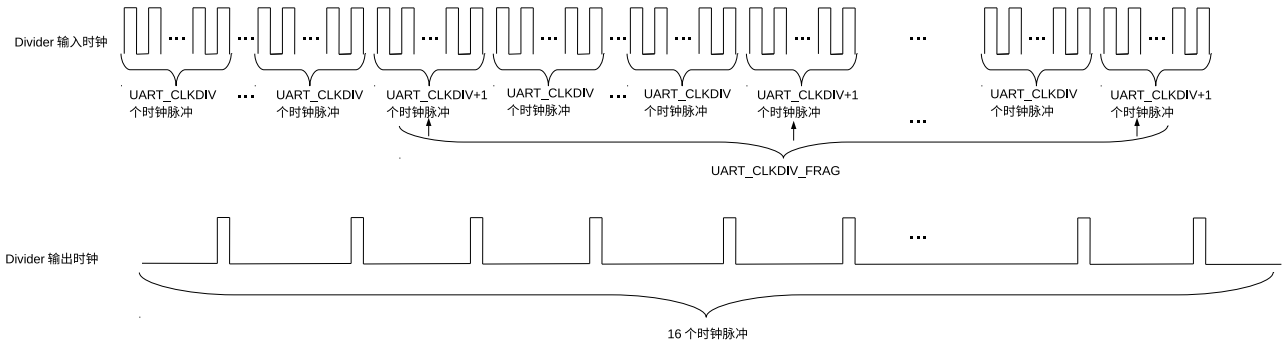


图 26.4-2. UART 控制器分频

为了支持 IrDA（详情见章节 26.4.7），IrDA 小数分频器会产生 16× UART_CLKDIV_REG 分频的时钟用于 IrDA 数据传输。产生 IrDA 数据传输时钟的小数分频器原理与上述小数分频器一样，取 UART_CLKDIV/16 作为分频值的整数部分，取 UART_CLKDIV 的低 4 比特作为小数部分。

26.4.3.2 波特率检测

置位 UART_AUTOBAUD_EN 可以开启 UART 波特率自检测功能。图 26.3-2 中的 Baudrate_Detect 可以滤除信号脉宽小于 UART_GLITCH_FILT 的噪声。

在 UART 双方进行通信之前可以通过发送几个随机数据让具有波特率检测功能的数据接收方进行波特率分析。UART_LOWPULSE_MIN_CNT 存储了最小低电平脉冲宽度，UART_HIGHPULSE_MIN_CNT 存储了最小高电平脉冲宽度，UART_POSEDGE_MIN_CNT 存储了两个上升沿之间的最小脉冲宽度，UART_NEGEDGE_MIN_CNT 存储了两个下降沿之间最小的脉冲宽度。软件可以通过读取这四个寄存器获取发送方的波特率。

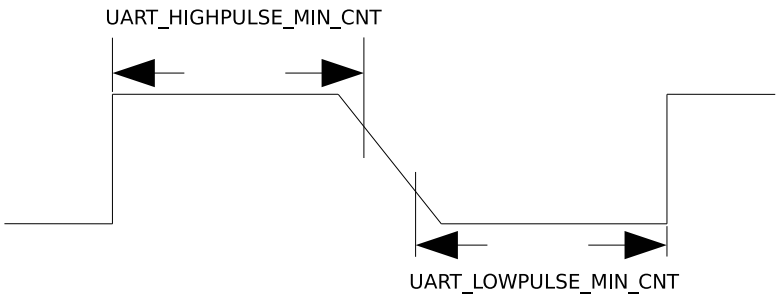


图 26.4-3. UART 信号下降沿较差时序图

波特率的计算分为三种情况：

1. 正常情况下，为防止因亚稳态在上升沿或下降沿附近采样数据错误而导致 UART_LOWPULSE_MIN_CNT 或者 UART_HIGHPULSE_MIN_CNT 不准确，单比特脉冲宽度可以通过将这两个值相加取平均消除误差。计算公式如下：

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

2. 对于 UART 信号的下降沿信号比较差的情况，如图 26.4-3 所示，这时通过取 UART_LOWPULSE_MIN_CNT 与 UART_HIGHPULSE_MIN_CNT 的和平均得到的值不准确，可以通过 UART_POSEDGE_MIN_CNT 获取发送方波特率。计算公式如下：

$$B_{\text{uart}} = \frac{f_{\text{clk}}}{(\text{UART\_POSEDGE\_MIN\_CNT} + 1)/2}$$

3. 对于 UART 信号的上升沿信号比较差的情况，可以通过 UART_NEGEDGE_MIN_CNT 获取发送方波特率。计算公式如下：

$$B_{\text{uart}} = \frac{f_{\text{clk}}}{(\text{UART\_NEGEDGE\_MIN\_CNT} + 1)/2}$$

26.4.4 UART 数据帧

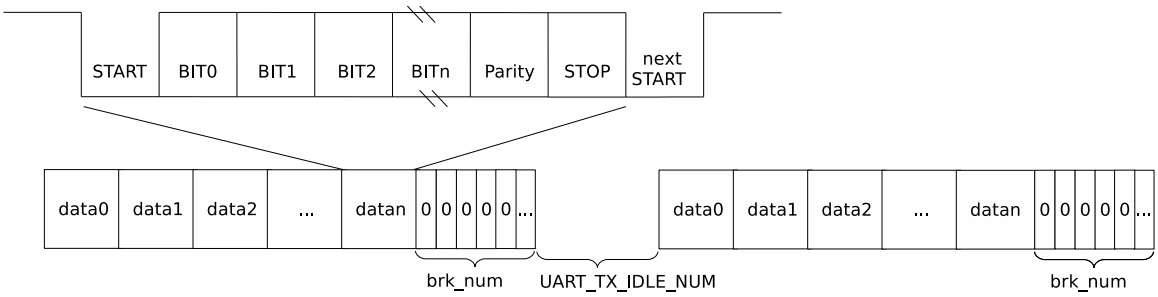


图 26.4-4. UART 数据帧结构

图 26.4-4 所示为基本数据帧格式，数据帧从 START 位开始以 STOP 位结束。START 占用 1 bit，STOP 位可以通过配置 UART_STOP_BIT_NUM 实现 1、1.5、2 位宽（RS485 模式下可增加转换延时，详见章节 26.4.6.2）。

START 为低电平，STOP 为高电平。

数据位宽 (BIT0 ~ BITn) 为 5 ~ 8 bit，可以通过 UART_BIT_NUM 进行配置。当置位 UART_PARITY_EN 时，数据帧会在数据之后添加一位奇偶校验位。UART_PARITY 用于选择奇校验或是偶校验。当接收器检测到输入数据的校验位错误时会产生 UART_PARITY_ERR_INT 中断，错误数据仍会存入 Rx_FIFO。当接收器检测到数据数据帧格式错误时会产生 UART_FRM_ERR_INT 中断，默认情况下，错误数据仍会存入 Rx_FIFO。

Tx_FIFO 中数据都发送完成后会产生 UART_TX_DONE_INT 中断。置位 UART_TXD_BRK 时，Tx_FIFO 中数据发送完成后，发送端会进入终止状态 (break condition)，继续发送几个连续的特殊数据帧 NULL。在 NULL 数据帧，TX 数据线输出为低电平。NULL 数据帧的数量可由可由 UART_TX_BRK_NUM 进行配置。发送器发送完所有的 NULL 数据帧之后会产生 UART_TX_BRK_DONE_INT 中断。数据帧之间可以通过配置 UART_TX_IDLE_NUM 保持最小间隔时间。当一帧数据之后的空闲时间大于等于 UART_TX_IDLE_NUM (单位为比特时间，即传输一个比特所需的时间) 寄存器的配置值时则产生 UART_TX_BRK
_IDLE_DONE_INT 中断。

在传输一个 NULL 数据帧所需的时间内，RX 数据线若一直输出低电平，接收端会检测为终止状态，并触发 UART_BRK_DET_INT 中断表示终止状态已结束。

接收端通过 UART_RXFIFO_TOUT_INT 中断检测总线状态。接收端接收到至少一个字节数据后，总线处于空闲状态超过 UART_RX_TOUT_THRHD 位时间时，触发 UART_RXFIFO_TOUT_INT 中断。您可用此中断检测发送端是否已经发送所有数据。

26.4.5 AT_CMD 字符格式

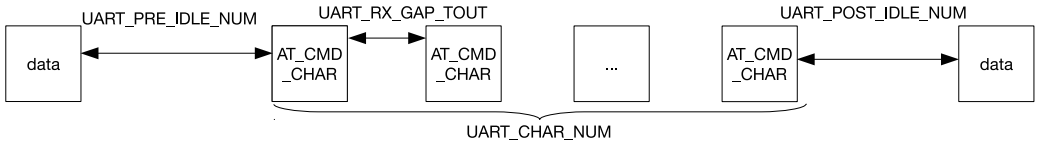


图 26.4-5. AT_CMD 字符格式

图 26.4-5 为一种特殊的 AT_CMD 字符格式。当接收器连续收到 AT_CMD_CHAR 字符且字符之间满足如下条件时将会产生 UART_AT_CMD_CHAR_DET_INT 中断。AT_CMD_CHAR 字符的具体值由寄存器 UART*n*_AT_CMD_CHAR 得到。

- 接收到的第一个 AT_CMD_CHAR 与上一个非 AT_CMD_CHAR 之间间隔至少 UART_PRE_IDLE_NUM 个波特率周期。
- AT_CMD_CHAR 字符之间间隔小于 UART_RX_GAP_TOUT 个波特率周期。
- 接收的 AT_CMD_CHAR 字符个数必须大于等于 UART_CHAR_NUM。
- 接收到的最后一个 AT_CMD_CHAR 字符与下一个非 AT_CMD_CHAR 之间间隔至少 UART_POST_IDLE_NUM 个波特率周期。

26.4.6 RS485

UART 支持 RS485 协议，RS485 因使用差分信号传输数据，相比于 RS232 具有更远的传输距离及更高的传输速率。RS485 有两线半双工及四线全双工模式，UART 模块采用两线半双工模式，并支持侦听总线的功能。RS485 两线 multidrop 模式，最大可支持 32 个 slave。

26.4.6.1 驱动控制

如图 26.4-6 所示，RS485 两线 multidrop 系统中，需要一个外部 RS485 传输器实现单端信号与差分信号的转换。RS485 传输器包括一个驱动器与一个接收器。当 UART 不作为发送器时，通过关闭驱动器来断开与差分传输线的连接。DE（驱动使能）信号为 1 时，使能驱动器；DE 为 0 关闭驱动器。

UART 接收端通过外部接收器将差分信号转为单端信号。RE 作为接收器的使能控制信号，RE 为 0，使能接收器；RE 为 1，关闭接收器。如果 RE 被配置为 0，从而允许 UART 保持侦听总线上的数据，包括 UART 发送的数据。

DE 信号的控制分为软件控制和硬件控制两种方法。为减少软件的开销，DE 信号采用硬件来控制。图 26.4-6 所示，DE 与 UART 的 dtrn_out 相连（详见 26.4.10.1 小节）。

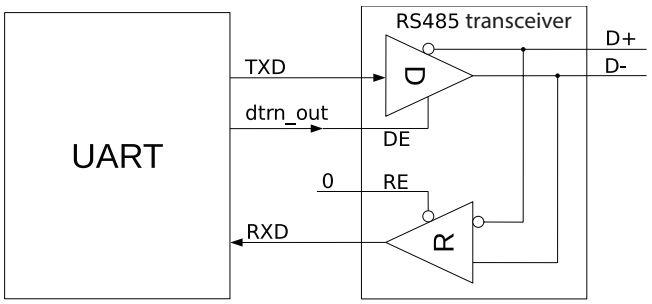


图 26.4-6. RS485 模式驱动控制结构图

26.4.6.2 转换延时

默认情况下，UART 处于接收状态。当从发送转为接收状态时，为保证发送数据被稳定接收，RS485 协议推荐在发送停止位之后增加一个波特率的转换延时。UART 发送模块支持在 Start 位之前和在停止位之后增加一个波特率的延时。置位 UART_DLO_EN，在 Start 位之前增加一个波特率周期延时；置位 UART_DL1_EN，在停止位之后增加一个波特率周期延时。

26.4.6.3 总线侦听

RS485 两线 multidrop 系统中，当外部 RS485 传输器的 RE 被配置为 0 时，UART 支持侦听总线。默认情况下，不允许 UART 在发送数据时接收数据。置位 UART_RS485TX_RX_EN，允许在发送数据时接收数据，配合外部 RS485 传输器的配置（如图 26.4-6 所示），UART 保持侦听传输总线。另外，默认情况下，不允许 UART 在接收数据时发送数据。置位 UART_RS485RXBY_TX_EN，允许在接收数据时发送数据。

UART 支持侦听 UART 发送的数据。UART 处于发送状态下，当侦听到 UART 发送的数据与 UART 接收的数据不同时，触发 UART_RS485_CLASH_INT 中断；侦听到发送的数据帧错误时，触发 UART_RS485_FRM_ERR_INT 中断；侦听到发送数据极性错误时，触发 UART_RS485_PARITY_ERR_INT 中断。

26.4.7 IrDA

IrDA 数据协议由物理层，链路接入层和链路管理层三个基本层协议组成。UART 实现了其物理层协议。在 IrDA 编码模式下，支持最大信号速率到 115.2 Kbit/s，即 SIR 模式。如图 26.4-7 所示，IrDA 编码器将来自 UART 的非归零编码 (NRZ) 信号采用反向归零编码 (RZI) 并输出给外部驱动和红外 LED，用 3/16 Bit Time 的脉宽调制信号表示逻辑 “0”，用低电平表示逻辑 “1”。IrDA 解码器接收来自红外接收器的信号并输出为 UART 的 NRZ 编码。一般情况下，接收端信号空闲时为高电平，编码器输出极性与解码器输入极性相反。当检测到低脉冲表示接收到开始信号。

IrDA 使能时，一个比特被划分为 16 个时钟周期，在其第 9、10、11 个时钟周期中，当需要发送的比特为 0 时，IrDA 输出为高。

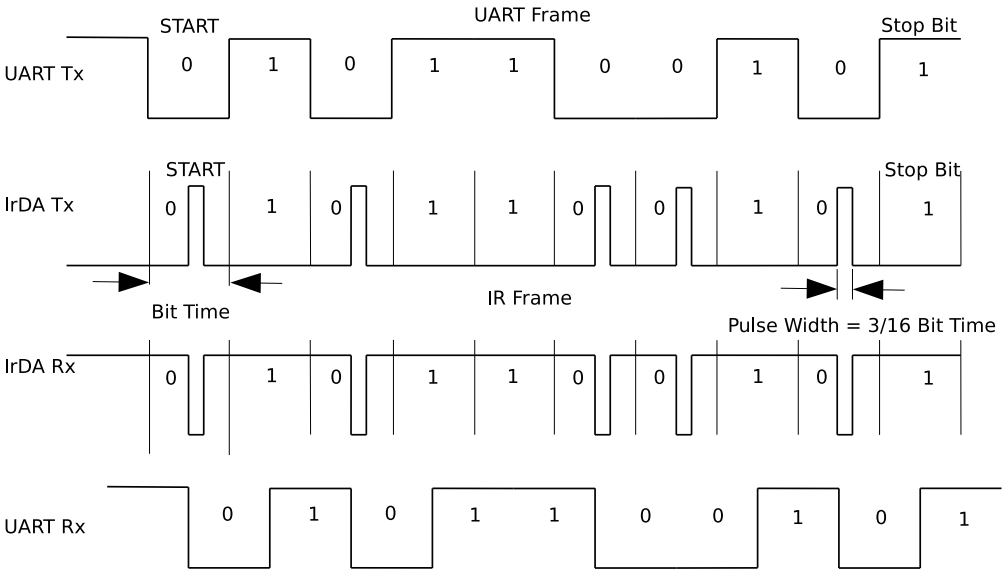


图 26.4-7. SIR 模式编解码时序图

IrDA 为半双工传输协议，不允许同时进行收发。如图 26.4-8 所示，置位 UART_IRDA_EN 使能 IrDA 功能。置位 UART_IRDA_TX_EN（拉高）使能 IrDA 发送数据，这时不允许 IrDA 接收数据；复位 UART_IRDA_TX_EN（拉低）使能 IrDA 接收数据，这时不允许 IrDA 发送数据。

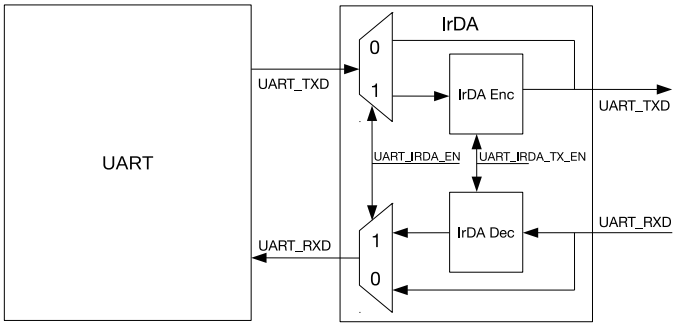


图 26.4-8. IrDA 编解码结构图

26.4.8 唤醒

UART0 和 UART1 支持唤醒功能。当 UART 处于 Light-sleep 状态时，Wakeup_Ctrl 开始统计 rxd_in 的上升沿个数，当上升沿个数大于等于（UART_ACTIVE_THRESHOLD + 3）时产生 wake_up 信号给 RTC 模块，由 RTC 来唤醒 ESP32-S3 芯片。

使用 UART 唤醒之后，需要通过在 Active 模式下向 UART 传输数据或是复位整个 UART 模块清除 wake_up 信号，否则下一次唤醒所需的上升沿个数将减少。

26.4.9 回环功能

UART*n* 支持回环功能。置位 UART_LOOPBACK 即开启 UART 的回环测试功能。此时 UART 的输出信号 txd_out 和其输入信号 rxd_in 相连，rtsn_out 和 ctsn_in 相连，dtrn_out 和 dsrn_out 相连。数据之后通过 txd_out 发

送。当接收的数据与发送的数据相同时表明 UART 能够正常发送和接收数据。

26.4.10 流控

UART 控制器有两种数据流控方式：硬件流控和软件流控。硬件流控主要通过输出信号 rtsn_out 以及输入信号 dsrn_in 进行数据流控制。软件流控主要通过在发送数据流中插入 XON/XOFF 字符以及在接收数据流中检测特殊字符来实现数据流控功能。

26.4.10.1 硬件流控

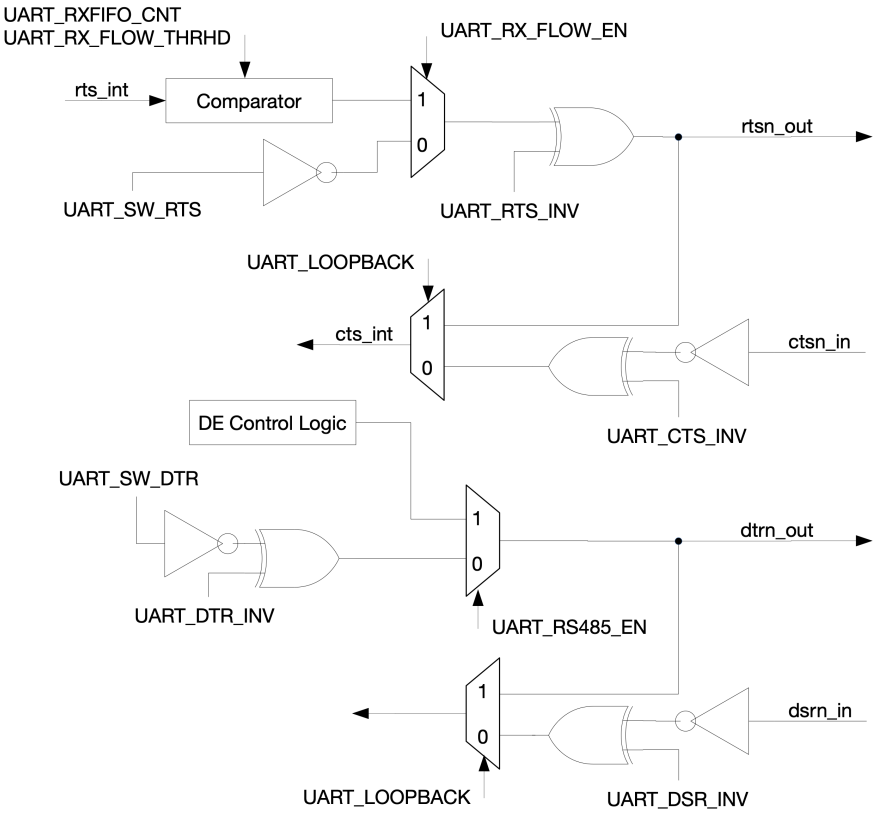


图 26.4-9. 硬件流控图

图 26.4-9 为 UART 硬件流控图。硬件流控的控制信号为输出信号 rtsn_out 及输入信号 ctsn_in。图 26.4-10 为两个 UART 之间硬件流控信号连接图。记 ESP32-S3 UART 为 IUO，External UART 为 EUO，下文将使用这两个标记来区分两个 UART。输出信号 rtsn_out (IUO) 为低电平表示允许对方 (EUO) 发送数据，rtsn_out (IUO) 为高电平表示通知对方 (EUO) 中止数据发送直到 rtsn_out (IUO) 恢复低电平。rtsn_out 输出信号的控制有两种方式。

- 软件控制：将 UART_RX_FLOW_EN 置 0 进入该模式。该模式下通过软件配置 UART_SW_RTS 改变 rtsn_out 的电平。
- 硬件控制：将 UART_RX_FLOW_EN 置 1 进入该模式。该模式下硬件会当 Rx_FIFO 中的数据大于 UART_RX_FLOW_THRHD 时拉高 rtsn_out 的电平。

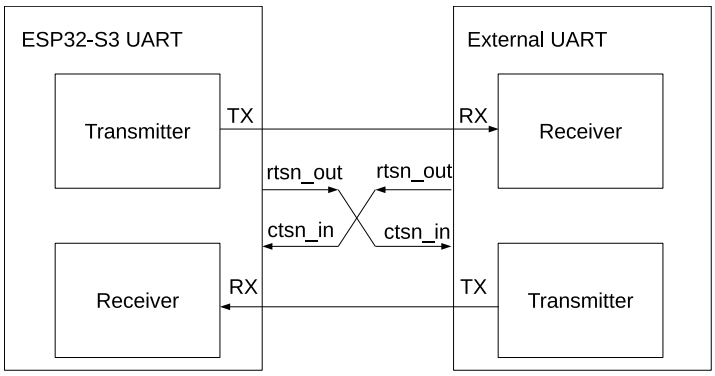


图 26.4-10. 硬件流控信号连接图

输入信号 ctsn_in (IUO) 为低电平表示允许发送端 (IUO) 发送数据；ctsn_in (IUO) 为高电平表示禁止发送端 (IUO) 发送数据。当 UART 检测到输入信号 ctsn_in (IUO) 的沿变化时会产生 UART_CTS_CHG_INT 中断。

UART 发送设备 (IUO) 输出信号 dtrn_out 为高电平表示发送数据已经准备完毕，处于可用状态。dtrn_out 通过配置寄存器 UART_SW_DTR 产生。UART 接收设备 (IUO) 在检测到输入信号 dsrn_in 的沿变化时会产生 UART_DSR_CHG_INT 中断。软件在检测到中断后，通过读取 UART_DSRN 可以获取 dsrn_in 的输入信号电平，UART_DSRN 为高电平时，表示对方设备 (EUO) 处于可用状态。

对于 RS485 两线 multidrop 系统，使用 dtrn_out 来收发转换。置位 UART_RS485_EN 使能 RS485 功能，dtrn_out 由硬件产生。数据开始发送时，dtrn_out 拉高，使能外部驱动器；数据最后一位发送完成后，dtrn_out 拉低，关闭外部驱动器。注意，当使能停止位之后增加一个波特率延时时，dtrn_out 会在延时结束后才拉低。

26.4.10.2 软件流控

软件流控不使用硬件的 CTS/RTS 控制线，而是在发送数据流中嵌入 XON/XOFF 字符来通知对方是否可以使用数据发送来实现流控。将 UART_SW_FLOW_CON_EN 置 1 使能软件流控。

在使用软件流控后，硬件会自动检测接收数据流中是否有 XON/XOFF 字符，在检测到相应的字符后会产生
UART_SW_XOFF_INT 或 UART_SW_XON_INT 中断。在检测到接收数据流中有 XOFF 字符后，发送器将会在发送完当前数据后停止发送；在检测到接收数据流中有 XON 字符后，将会使能发送器发送数据。另外，软件可以通过置位 UART_FORCE_XOFF 来强制发送器停止发送数据，发送器会在发送完当前字节后停止发送；也可以通过置位 UART_FORCE_XON 来使能发送器发送数据。

软件可以根据 Rx_FIFO 中剩余空间大小决定流控字符的发送。置位 UART_SEND_XOFF，发送器会在发送完当前数据之后插入一个 XOFF 字符，该字符通过寄存器 UART_XOFF_CHAR 配置；置位 UART_SEND_XON，发送器会在发送完当前数据之后插入一个 XON 字符，该字符通过寄存器 UART_XON_CHAR 配置。另外，当 UART 接收 FIFO 中的数据量超过 UART_XOFF_THRESHOLD 时，硬件会置位 UART_SEND_XOFF，UART 发送器会在发送完当前数据之后插入一个 XOFF 字符，该字符通过寄存器 UART_XOFF_CHAR 配置。当 UART 接收 FIFO 中的数据量小于 UART_XON_THRESHOLD 时，硬件会置位 UART_SEND_XON，UART 发送器会在发送完当前数据之后插入一个 XON 字符，该字符通过寄存器 UART_XON_CHAR 配置。

26.4.11 GDMA 模式

ESP32-S3 中的三个 UART 接口通过通用主机控制器接口 (UHCI) 共用 1 组 GDMA TX/RX 通道。在 GDMA 模式下，支持对 HCI 协议数据包的解 (decoder) 及数据包封装 (encoder)。UHCI_UART*n*_CE 寄存器用于选择哪个串

口占用 GDMA 通道。

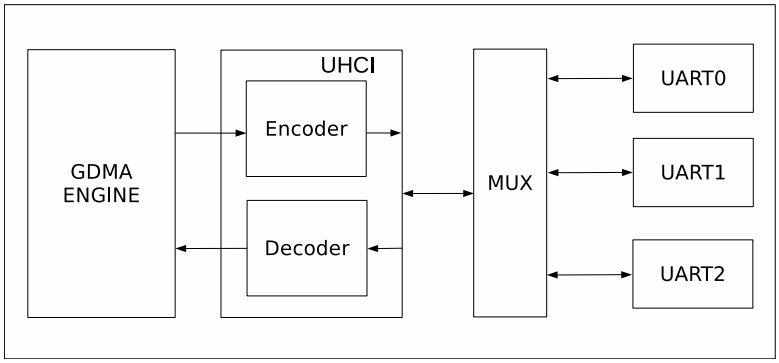


图 26.4-11. GDMA 模式数据传输

图 26.4-11 为 GDMA 方式数据传输图。在 GDMA RX 通道接收数据前，软件将接收链表准备好（有关接收链表的更多信息，请参考章节 3 通用 DMA 控制器 (GDMA)）。GDMA_INLINK_ADDR_CH*n* 用于指向第一个接收链表描述符。置位 GDMA_INLINK_START_CH*n* 之后，通用主机控制器接口 (UHCI) 会将 UART 接收到的数据传送给 Decoder。经过 Decoder 解析之后的数据在 GDMA 通道的控制下存入接收链表指定的 RAM 空间。

在 GDMA TX 通道发送数据前，软件需要将发送链表和发送数据准备好，GDMA_OUTLINK_ADDR_CH*n* 用于指向第一个发送链表描述符。置位 GDMA_OUTLINK_START_CH*n* 之后，GDMA 引擎即从链表中指定的 RAM 地址读取数据，并通过 Encoder 进行数据包封装，然后经 UART 的发送模块串行发送出去。

HCI 的数据包格式为（分隔符 + 数据 + 分隔符）。Encoder 用于在数据前后加上分隔符，并将数据中和分隔符一样的数据用特殊字符（即转义字符）替换。Decoder 用于去除数据包前后分隔符，并将数据中的转义字符替换为分隔符。数据前后的分隔符可以有连续多个。分隔符可由 UHCI_SEPER_CHAR 进行配置，默认值为 0xC0。数据中与分隔符一样的数据可以用 UHCI_ESC_SEQ0_CHAR0（默认为 0xDB）和 UHCI_ESC_SEQ0_CHAR1（默认为 0xDD）进行替换。当数据全部发送完成后，会产生 GDMA_OUT_TOTAL_EOF_CH*n*_INT 中断。当数据接收完成后，会产生 GDMA_IN_SUC_EOF_CH*n*_INT 中断。

26.4.12 UART 中断

- UART_AT_CMD_CHAR_DET_INT：当接收器检测到 AT_CMD 字符时触发此中断。
- UART_RS485_CLASH_INT：在 RS485 模式下检测到发送器和接收器之间的冲突时触发此中断。
- UART_RS485_FRM_ERR_INT：在 RS485 模式下检测到发送块发送的数据帧错误时触发此中断。
- UART_RS485_PARITY_ERR_INT：在 RS485 模式下检测到发送块发送的数据校验位错误时触发此中断。
- UART_TX_DONE_INT：当发送器发送完 FIFO 中的所有数据时触发此中断。
- UART_TX_BRK_IDLE_DONE_INT：发送器发送完最后一个数据后保持空闲状态时触发此中断。标记为空闲状态的最短时间由阈值决定（可配置）。
- UART_TX_BRK_DONE_INT：当发送 FIFO 中的数据发送完之后发送器完成了发送 NULL 则触发此中断。
- UART_GLITCH_DET_INT：当接收器在起始位的中点处检测到毛刺时触发此中断。
- UART_SW_XOFF_INT：UART_SW_FLOW_CON_EN 置位时，当接收器接收到 XOFF 字符时触发此中断。
- UART_SW_XON_INT：UART_SW_FLOW_CON_EN 置位时，当接收器接收到 XON 字符时触发此中断。
- UART_RXFIFO_TOUT_INT：当接收器接收一个字节的时间大于 UART_RX_TOUT_THRHD 时触发此中断。

- UART_BRK_DET_INT：当接收器在停止位之后检测到一个 NULL（即传输一个 NULL 的时间内保持逻辑低电平）时触发此中断。
- UART_CTS_CHG_INT：当接收器检测到 CTSn 信号的沿变化时触发此中断。
- UART_DSR_CHG_INT：当接收器检测到 DSRn 信号的沿变化时触发此中断。
- UART_RXFIFO_OVF_INT：当接收器接收到的数据量多于 FIFO 的存储量时触发此中断。
- UART_FRM_ERR_INT：当接收器检测到数据帧错误时触发此中断。
- UART_PARITY_ERR_INT：当接收器检测到校验位错误时触发此中断。
- UART_TXFIFO_EMPTY_INT：当发送 FIFO 中的数据量少于 UART_TXFIFO_EMPTY_THRHD 所指定的值时触发此中断。
- UART_RXFIFO_FULL_INT：当接收器接收到的数据多于 UART_RXFIFO_FULL_THRHD 所指定的值时触发此中断。
- UART_WAKEUP_INT：UART 被唤醒时产生此中断。

26.4.13 UCHI 中断

- UHCI_APP_CTRL1_INT：软件置位 UHCI_APP_CTRL1_INT_RAW 时触发此中断。
- UHCI_APP_CTRL0_INT：软件置位 UHCI_APP_CTRL0_INT_RAW 时触发此中断。
- UHCI_OUTLINK_EOF_ERR_INT：当检测到发送链表描述符中的 EOF 有错误时触发此中断。
- UHCI_SEND_A_REG_Q_INT：当使用 always_send 发送一串短包，UHCI 发送了短包后触发此中断。
- UHCI_SEND_S_REG_Q_INT：当使用 single_send 发送一串短包，UHCI 发送了短包后触发此中断。
- UHCI_TX_HUNG_INT：当 UHCI 利用 GDMA TX 通道从 RAM 中读取数据的时间过长时触发此中断。
- UHCI_RX_HUNG_INT：当 UHCI 利用 GDMA RX 通道接收数据的时间过长时触发此中断。
- UHCI_TX_START_INT：当检测到分隔符时触发此中断。
- UHCI_RX_START_INT：当分隔符已发送时触发此中断。

26.5 编程流程

26.5.1 寄存器类型

UART 的所有寄存器都处于 APB_CLK 时钟域。对于软件可配置的寄存器，根据其作用的时钟域及同步处理，将其分为三类：同步寄存器、静态寄存器及立即寄存器。同步寄存器作用于 Core 时钟域，这些寄存器需要经过同步之后才能生效。静态寄存器也作用于 Core 时钟域，但这些寄存器不会在 UART 工作过程中动态修改。静态寄存器没有同步处理，软件可以通过开关 UART TX/RX Core 时钟的方式保证 UART Core 时钟域采样到正确的配置信息。立即寄存器作用于 APB_CLK 时钟域，通过 APB 总线配置后立即生效。

26.5.1.1 同步寄存器

为了确保作用于 UART Core 时钟域的寄存器被正确采样，他们中大多数都做了跨时钟域处理，这部分即为同步寄存器。同步寄存器如表 26.5-1 所示。对这些寄存器的配置流程如下：

- 将 UART_UPDATE_CTRL 清 0 使能寄存器同步功能；

- 等待 UART_REG_UPDATE 为 0，确保上一次同步已经完成；
- 配置同步寄存器；
- 向 UART_REG_UPDATE 写 1，将配置的值同步到 Core 时钟域。

表 26.5-1. UART*n* 同步寄存器

寄存器	域名
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
	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_RS485_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_DL0_EN
	UART_RS485_EN

26.5.1.2 静态寄存器

在作用于 UART Core 时钟域的寄存器中，有一部分寄存器不会在 UART 工作过程中动态修改，被认为是静态的，称为静态寄存器。静态寄存器没有做跨时钟域处理。静态寄存器的配置一定是 UART TX/RX 停止工作阶段，因此可以通过关闭 UART TX/RX 时钟的方式，保证配置寄存器的亚稳态不会被采样到。当打开 UART TX/RX 时钟打开时，软件配置的值已经稳定，从而确保配置的值被正确采样。表 26.5-2 列出了这些寄存器。对这些寄存器的配置流程如下：

- 根据当前停止工作的模块为 UART TX 还是 RX，将 UART_TX_SCLK_EN 或 UART_RX_SCLK_EN 清 0 关闭 UART TX 或 RX 时钟；
- 配置静态寄存器；
- 向 UART_TX_SCLK_EN 或 UART_RX_SCLK_EN 写 1 打开 UART TX 或 RX 时钟。

表 26.5-2. UART*n* 静态寄存器

寄存器	域名
UART_RX_FILT_REG	UART_GLITCH_FILT_EN
	UART_GLITCH_FILT[7:0]
UART_SLEEP_CONF_REG	UART_ACTIVE_THRESHOLD[9:0]
UART_SWFC_CONF0_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]
UART_AT_CMD_CHAR_REG	UART_CHAR_NUM[7:0]
	UART_AT_CMD_CHAR[7:0]

26.5.1.3 立即寄存器

除表 26.5-1 与 26.5-2 外的所有软件可配置寄存器作用于 APB_CLK 时钟域，即为立即寄存器，例如，中断及 FIFO 配置寄存器等。

26.5.2 具体步骤

图26.5-1 显示了 UART 模块的编程流程。主要包括：初始化、寄存器配置、启动 UART TX/RX 和数据传输结束。

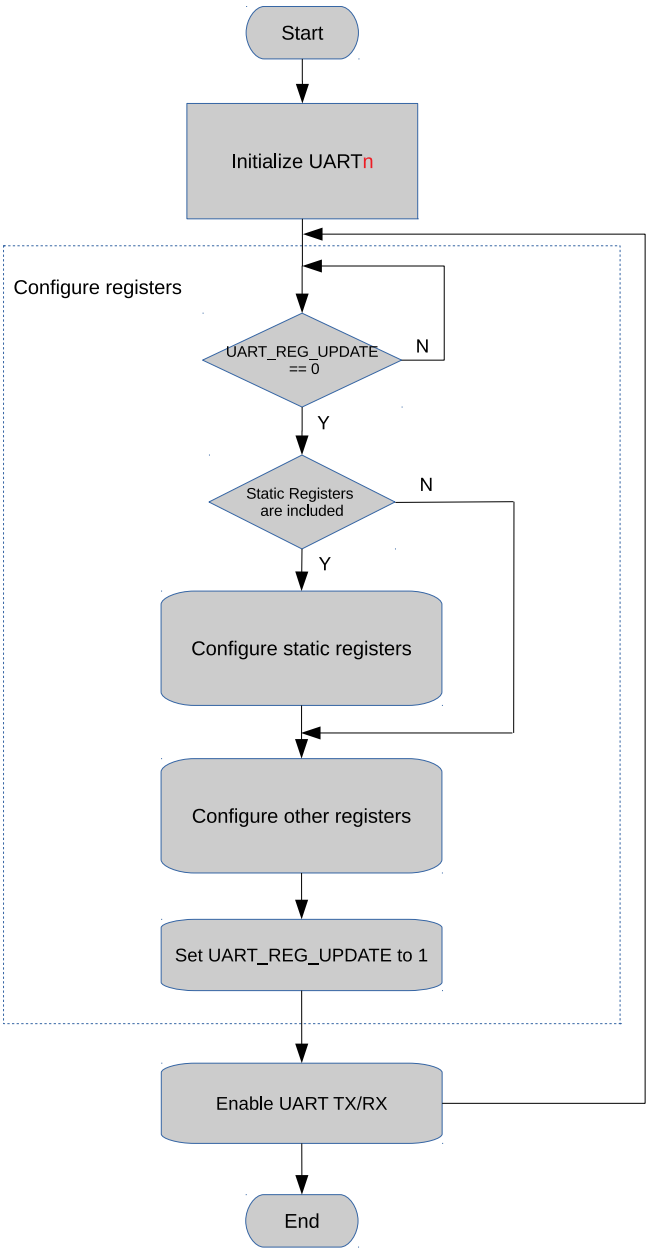


图 26.5-1. UART 编程流程

26.5.2.1 UART*n* 模块初始化

UART*n* 模块初始化流程包括：UART*n* 复位和使能寄存器同步功能。

UART*n* 复位流程如下：

- 将 SYSTEM_UART_MEM_CLK_EN 置 1 打开 UART RAM 时钟；
- 将 SYSTEM_UART*n*_CLK_EN 置 1 打开 UART*n* APB_CLK；
- 将寄存器 SYSTEM_UART*n*_RST 清 0；
- 向寄存器 UART_RST_CORE 写 1；
- 向寄存器 SYSTEM_UART*n*_RST 写 1；
- 将寄存器 SYSTEM_UART*n*_RST 清 0；
- 将寄存器 UART_RST_CORE 清 0。

使能寄存器同步功能，需将 UART_UPDATE_CTRL 清 0。

26.5.2.2 UART*n* 通信配置

UART*n* 通信配置流程如下：

- 等待 UART_REG_UPDATE 为 0，确保上一次同步已经完成；
- 如果配置寄存器中包含静态寄存器，配置流程参考26.5.1.2 完成配置；
- 配置 UART_SCLK_SEL 选择时钟源；
- 配置 UART_SCLK_DIV_NUM、UART_SCLK_DIV_A、UART_SCLK_DIV_B 设置预分频器系数；
- 配置 UART_CLKDIV、UART_CLKDIV_FRAG 设置发送波特率；
- 配置 UART_BIT_NUM 设置数据长度；
- 配置 UART_PARITY_EN、UART_PARITY 设置奇偶校验；
- 可选步骤，根据应用不同存在差异...
- 向 UART_REG_UPDATE 写 1，将配置的值同步到 Core 时钟域。

26.5.2.3 启动 UART*n*

启动 UART*n* TX 发送数据：

- 配置 UART_TXFIFO_EMPTY_THRHD，设置 TXFIFO 空阈值；
- 对 UART_TXFIFO_EMPTY_INT_ENA 置 0，关闭 UART_TXFIFO_EMPTY_INT 中断；
- 向 UART_RXFIFO_RD_BYTE 写入需要发送的数据；
- 置位 UART_TXFIFO_EMPTY_INT_CLR，清除 UART_TXFIFO_EMPTY_INT 中断；
- 置位 UART_TXFIFO_EMPTY_INT_ENA，使能 UART_TXFIFO_EMPTY_INT 中断；
- 检测 UART_TXFIFO_EMPTY_INT，等待发送数据结束。

启动 UART*n* RX 数据接收：

- 配置 UART_RXFIFO_FULL_THRHD，设置 RXFIFO 满阈值；
- 置位 UART_RXFIFO_FULL_INT_ENA，使能 UART_RXFIFO_FULL_INT 中断；
- 检测 UART_TXFIFO_FULL_INT，等待 RXFIFO 接收数据满；
- 通过读 UART_RXFIFO_RD_BYTE，从 RXFIFO 中读出数据，并可通过 UART_RXFIFO_CNT 获得当前 RXFIFO 中的接收数据量。

26.6 寄存器列表

26.6.1 UART 寄存器列表

本小节的所有地址均为相对于 UART 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
FIFO 配置			
UART_FIFO_REG	FIFO 数据寄存器	0x0000	RO
UART_MEM_CONF_REG	UART 阈值和分配配置	0x0060	R/W
UART 中断寄存器			
UART_INT_RAW_REG	原始中断状态	0x0004	R/WTC/SS
UART_INT_ST_REG	屏蔽中断状态	0x0008	RO
UART_INT_ENA_REG	中断使能位	0x000C	R/W
UART_INT_CLR_REG	中断清除位	0x0010	WT
配置寄存器			
UART_CLKDIV_REG	时钟分频配置	0x0014	R/W
UART_RX_FILT_REG	RX 滤波器配置	0x0018	R/W
UART_CONF0_REG	配置寄存器 0	0x0020	R/W
UART_CONF1_REG	配置寄存器 1	0x0024	R/W
UART_FLOW_CONF_REG	软件流控配置	0x0034	varies
UART_SLEEP_CONF_REG	睡眠模式配置	0x0038	R/W
UART_SWFC_CONF0_REG	软件流控字符配置	0x003C	R/W
UART_SWFC_CONF1_REG	软件流控字符配置	0x0040	R/W
UART_TXBRK_CONF_REG	TX 断开字符配置	0x0044	R/W
UART_IDLE_CONF_REG	帧结束空闲配置	0x0048	R/W
UART_RS485_CONF_REG	RS485 模式配置	0x004C	R/W
UART_CLK_CONF_REG	UART core 时钟配置	0x0078	R/W
状态寄存器			
UART_STATUS_REG	UART 状态寄存器	0x001C	RO
UART_MEM_TX_STATUS_REG	TX FIFO 写入、读取偏移地址	0x0064	RO
UART_MEM_RX_STATUS_REG	RX FIFO 写入、读取偏移地址	0x0068	RO
UART_FSM_STATUS_REG	UART 发送和接收状态	0x006C	RO
自动波特率检测寄存器			
UART_LOWPULSE_REG	自动波特率检测最短低电平脉冲持续时间寄存器	0x0028	RO
UART_HIGHPULSE_REG	自动波特率检测最短高电平脉冲持续时间寄存器	0x002C	RO
UART_RXD_CNT_REG	自动波特率检测沿变化计数寄存器	0x0030	RO
UART_POSPULSE_REG	自动波特率检测高电平脉冲寄存器	0x0070	RO
UART_NEGPULSE_REG	自动波特率检测低电平脉冲寄存器	0x0074	RO
AT 转义序列检测配置			

名称	描述	地址	访问
UART_AT_CMD_PRECNT_REG	序列发送前的时序配置	0x0050	R/W
UART_AT_CMD_POSTCNT_REG	序列发送后的时序配置	0x0054	R/W
UART_AT_CMD_GAPTOUT_REG	超时配置	0x0058	R/W
UART_AT_CMD_CHAR_REG	AT 转义序列检测配置	0x005C	R/W
版本寄存器			
UART_DATE_REG	UART 版本控制寄存器	0x007C	R/W
UART_ID_REG	UART ID 寄存器	0x0080	varies

26.6.2 UHCI 寄存器列表

本小节的所有地址均为相对于 UHCI 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
配置寄存器			
UHCI_CONF0_REG	UHCI 配置寄存器	0x0000	R/W
UHCI_CONF1_REG	UHCI 配置寄存器	0x0018	varies
UHCI_ESCAPE_CONF_REG	转义符配置	0x0024	R/W
UHCI_HUNG_CONF_REG	超时配置	0x0028	R/W
UHCI_ACK_NUM_REG	配置 UHCI ACK 值	0x002C	varies
UHCI_QUICK_SENT_REG	UHCI 快速发送配置寄存器	0x0034	varies
UHCI_REG_Q0_WORD0_REG	Q0_WORD0 快速发送寄存器	0x0038	R/W
UHCI_REG_Q0_WORD1_REG	Q0_WORD1 快速发送寄存器	0x003C	R/W
UHCI_REG_Q1_WORD0_REG	Q1_WORD0 快速发送寄存器	0x0040	R/W
UHCI_REG_Q1_WORD1_REG	Q1_WORD1 快速发送寄存器	0x0044	R/W
UHCI_REG_Q2_WORD0_REG	Q2_WORD0 快速发送寄存器	0x0048	R/W
UHCI_REG_Q2_WORD1_REG	Q2_WORD1 快速发送寄存器	0x004C	R/W
UHCI_REG_Q3_WORD0_REG	Q3_WORD0 快速发送寄存器	0x0050	R/W
UHCI_REG_Q3_WORD1_REG	Q3_WORD1 快速发送寄存器	0x0054	R/W
UHCI_REG_Q4_WORD0_REG	Q4_WORD0 快速发送寄存器	0x0058	R/W
UHCI_REG_Q4_WORD1_REG	Q4_WORD1 快速发送寄存器	0x005C	R/W
UHCI_REG_Q5_WORD0_REG	Q5_WORD0 快速发送寄存器	0x0060	R/W
UHCI_REG_Q5_WORD1_REG	Q5_WORD1 快速发送寄存器	0x0064	R/W
UHCI_REG_Q6_WORD0_REG	Q6_WORD0 快速发送寄存器	0x0068	R/W
UHCI_REG_Q6_WORD1_REG	Q6_WORD1 快速发送寄存器	0x006C	R/W
UHCI_ESC_CONF0_REG	转义序列配置寄存器 0	0x0070	R/W
UHCI_ESC_CONF1_REG	转义序列配置寄存器 1	0x0074	R/W
UHCI_ESC_CONF2_REG	转义序列配置寄存器 2	0x0078	R/W
UHCI_ESC_CONF3_REG	转义序列配置寄存器 3	0x007C	R/W
UHCI_PKT_THRES_REG	包长度配置寄存器	0x0080	R/W
UHCI 中断寄存器			
UHCI_INT_RAW_REG	原始中断状态	0x0004	varies

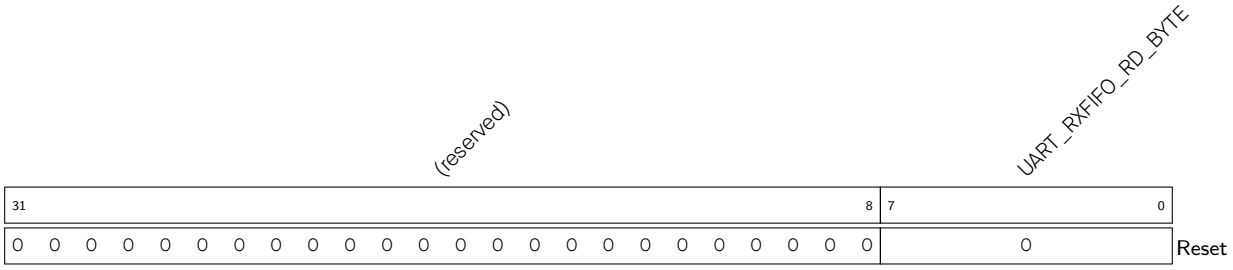
名称	描述	地址	访问
UHCI_INT_ST_REG	屏蔽中断状态	0x0008	RO
UHCI_INT_ENA_REG	中断使能位	0x000C	R/W
UHCI_INT_CLR_REG	中断清除位	0x0010	WT
UHCI_APP_INT_SET_REG	软件中断触发源	0x0014	WT
UHCI 状态寄存器			
UHCI_STATE0_REG	UHCI 接收状态	0x001C	RO
UHCI_STATE1_REG	UHCI 发送状态	0x0020	RO
UHCI_RX_HEAD_REG	UHCI 包报头寄存器	0x0030	RO
版本寄存器			
UHCI_DATE_REG	UHCI 版本控制寄存器	0x0084	R/W

26.7 寄存器

26.7.1 UART 寄存器

本小节的所有地址均为相对于 [UART 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 26.1. UART_FIFO_REG (0x0000)



UART_RXFIFO_RD_BYTE UART*n* 通过此字段访问 FIFO。(RO)

Register 26.2. UART_MEM_CONF_REG (0x0060)



UART_RX_SIZE 配置 RAM 分配给 RX FIFO 的空间大小。默认为 128 字节。(R/W)

UART_TX_SIZE 配置 RAM 分配给 TX FIFO 的空间大小。默认为 128 字节。(R/W)

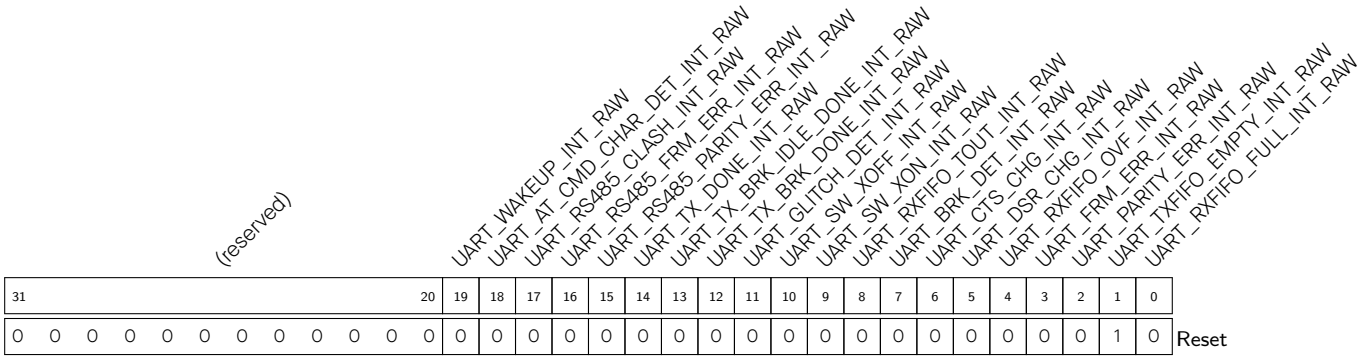
UART_RX_FLOW_THRHD 配置使用硬件流控时接收数据的最大值。(R/W)

UART_RX_TOUT_THRHD 配置接收器接收一个字节所需时间的阈值，单位是比特时间（即传输一个比特所需的时间）。接收器接收一个字节所需时间超过阈值且 UART_RX_TOUT_EN 置 1 时触发 UART_RXFIFO_TOUT_INT 中断。(R/W)

UART_MEM_FORCE_PD 置位此位强制关闭 UART RAM。(R/W)

UART_MEM_FORCE_PU 置位此位强制开启 UART RAM。(R/W)

Register 26.3. UART_INT_RAW_REG (0x0004)



UART_RXFIFO_FULL_INT_RAW 接收器接收数据多于 UART_RXFIFO_FULL_THRHD 的值时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_TXFIFO_EMPTY_INT_RAW TX FIFO 中的数据少于 UART_TXFIFO_EMPTY_THRHD 的值时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_PARITY_ERR_INT_RAW 接收器检测到数据奇偶检验位错误时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_FRM_ERR_INT_RAW 接收器检测到数据帧错误时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_RXFIFO_OVF_INT_RAW 接收器接收数据超过 RX FIFO 的存储容量时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_DSR_CHG_INT_RAW 接收器检测到 DSRn 信号的沿变化时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_CTS_CHG_INT_RAW 接收器检测到 CTSn 信号的沿变化时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_BRK_DET_INT_RAW 接收器在停止位后检测到 0 时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_RXFIFO_TOUT_INT_RAW 接收器接收一个字节所需时间超过 UART_RX_TOUT_THRHD 时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_SW_XON_INT_RAW 接收器接收到 XON 字符且 UART_SW_FLOW_CON_EN 置 1 时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_SW_XOFF_INT_RAW 接收器接收到 XOFF 字符且 UART_SW_FLOW_CON_EN 置 1 时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_GLITCH_DET_INT_RAW 接收器在起始位的中点处检测到毛刺时，该原始中断位翻转至高电平。(R/WTC/SS)

见下页...

Register 26.3. UART_INT_RAW_REG (0x0004)

接上页...

UART_TX_BRK_DONE_INT_RAW 发送器在发送完 TX FIFO 中所有数据后完成 NULL 字符的发送时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_TX_BRK_IDLE_DONE_INT_RAW 发送器发送完最后一个数据后的间隔时间达到阈值时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_TX_DONE_INT_RAW 发送器发完 FIFO 中的所有数据后，该原始中断位翻转至高电平。(R/WTC/SS)

UART_RS485_PARITY_ERR_INT_RAW RS485 模式下接收器检测到发送器回音的数据检验位错误时，该原始中断位翻转至高电平。(R/WTC/SS)

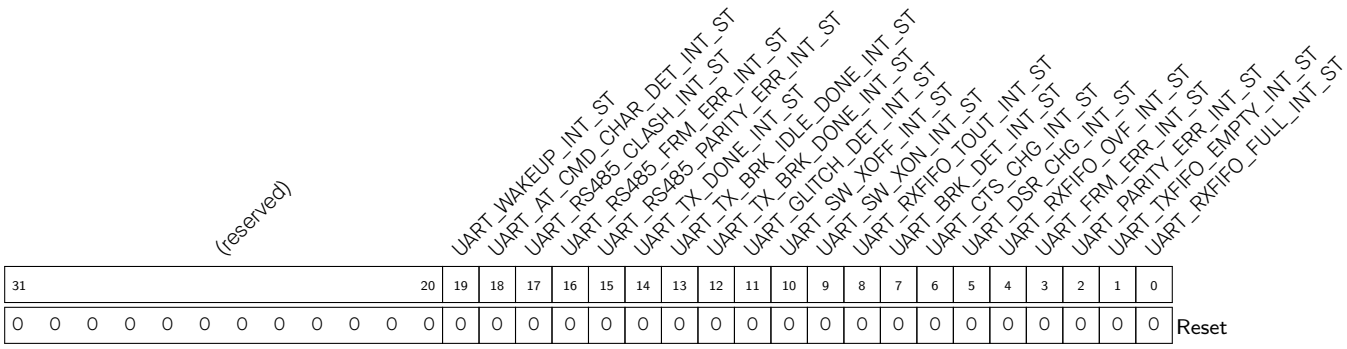
UART_RS485_FRM_ERR_INT_RAW RS485 模式下接收器检测到发送器回音的数据帧错误时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_RS485_CLASH_INT_RAW RS485 模式下检测到发送器与接收器冲突时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_AT_CMD_CHAR_DET_INT_RAW 接收器检测到配置的 UART_AT_CMD_CHAR 时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_WAKEUP_INT_RAW 输入 RXD 沿变化次数超过 Light-sleep 模式指定的 (UART_ACTIVE_THRESHOLD + 3) 值时，该原始中断位翻转至高电平。(R/WTC/SS)

Register 26.4. UART_INT_ST_REG (0x0008)



UART_RXFIFO_FULL_INT_ST UART_RXFIFO_FULL_INT_ENA 置 1 时 UART_RXFIFO_FULL_INT 的状态位。(RO)

UART_TXFIFO_EMPTY_INT_ST UART_TXFIFO_EMPTY_INT_ENA 置 1 时 UART_TXFIFO_EMPTY_INT 的状态位。(RO)

UART_PARITY_ERR_INT_ST UART_PARITY_ERR_INT_ENA 置 1 时 UART_PARITY_ERR_INT 的状态位。(RO)

UART_FRM_ERR_INT_ST UART_FRM_ERR_INT_ENA 置 1 时 UART_FRM_ERR_INT 的状态位。(RO)

UART_RXFIFO_OVF_INT_ST UART_RXFIFO_OVF_INT_ENA 置 1 时 UART_RXFIFO_OVF_INT 的状态位。(RO)

UART_DSR_CHG_INT_ST UART_DSR_CHG_INT_ENA 置 1 时 UART_DSR_CHG_INT 的状态位。(RO)

UART_CTS_CHG_INT_ST UART_CTS_CHG_INT_ENA 置 1 时 UART_CTS_CHG_INT 的状态位。(RO)

UART_BRK_DET_INT_ST UART_BRK_DET_INT_ENA 置 1 时 UART_BRK_DET_INT 的状态位。(RO)

UART_RXFIFO_TOUT_INT_ST UART_RXFIFO_TOUT_INT_ENA 置 1 时 UART_RXFIFO_TOUT_INT 的状态位。(RO)

UART_SW_XON_INT_ST UART_SW_XON_INT_ENA 置 1 时 UART_SW_XON_INT 的状态位。(RO)

UART_SW_XOFF_INT_ST UART_SW_XOFF_INT_ENA 置 1 时 UART_SW_XOFF_INT 的状态位。(RO)

UART_GLITCH_DET_INT_ST UART_GLITCH_DET_INT_ENA 置 1 时 UART_GLITCH_DET_INT 的状态位。(RO)

见下页...

Register 26.4. UART_INT_ST_REG (0x0008)

接上页...

UART_TX_BRK_DONE_INT_ST UART_TX_BRK_DONE_INT_ENA 置 1 时 UART_TX_BRK_DONE_INT 的状态位。(RO)

UART_TX_BRK_IDLE_DONE_INT_ST UART_TX_BRK_IDLE_DONE_INT_ENA 置 1 时 UART_TX_BRK_IDLE_DONE_INT 的状态位。(RO)

UART_TX_DONE_INT_ST UART_TX_DONE_INT_ENA 置 1 时 UART_TX_DONE_INT 的状态位。(RO)

UART_RS485_PARITY_ERR_INT_ST UART_RS485_PARITY_INT_ENA 置 1 时 UART_RS485_PARITY_ERR_INT 的状态位。(RO)

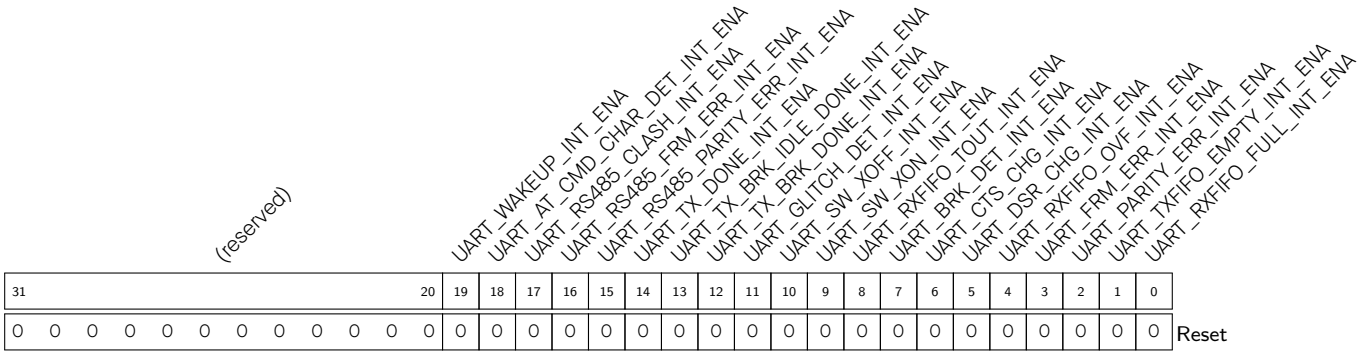
UART_RS485_FRM_ERR_INT_ST UART_RS485_FRM_ERR_INT_ENA 置 1 时 UART_RS485_FRM_ERR_INT 的状态位。(RO)

UART_RS485_CLASH_INT_ST UART_RS485_CLASH_INT_ENA 置 1 时 UART_RS485_CLASH_INT 的状态位。(RO)

UART_AT_CMD_CHAR_DET_INT_ST UART_AT_CMD_CHAR_DET_INT_ENA 置 1 时 UART_AT_CMD_CHAR_DET_INT 的状态位。(RO)

UART_WAKEUP_INT_ST UART_WAKEUP_INT_ENA 置 1 时 UART_WAKEUP_INT 的状态位。(RO)

Register 26.5. UART_INT_ENA_REG (0x000C)



UART_RXFIFO_FULL_INT_ENA UART_RXFIFO_FULL_INT 的使能位。(R/W)

UART_TXFIFO_EMPTY_INT_ENA UART_TXFIFO_EMPTY_INT 的使能位。(R/W)

UART_PARITY_ERR_INT_ENA UART_PARITY_ERR_INT 的使能位。(R/W)

UART_FRM_ERR_INT_ENA UART_FRM_ERR_INT 的使能位。(R/W)

UART_RXFIFO_OVF_INT_ENA UART_RXFIFO_OVF_INT 的使能位。(R/W)

UART_DSR_CHG_INT_ENA UART_DSR_CHG_INT 的使能位。(R/W)

UART_CTS_CHG_INT_ENA UART_CTS_CHG_INT 的使能位。(R/W)

UART_BRK_DET_INT_ENA UART_BRK_DET_INT 的使能位。(R/W)

UART_RXFIFO_TOUT_INT_ENA UART_RXFIFO_TOUT_INT 的使能位。(R/W)

UART_SW_XON_INT_ENA UART_SW_XON_INT 的使能位。(R/W)

UART_SW_XOFF_INT_ENA UART_SW_XOFF_INT 的使能位。(R/W)

UART_GLITCH_DET_INT_ENA UART_GLITCH_DET_INT 的使能位。(R/W)

UART_TX_BRK_DONE_INT_ENA UART_TX_BRK_DONE_INT 的使能位。(R/W)

UART_TX_BRK_IDLE_DONE_INT_ENA UART_TX_BRK_IDLE_DONE_INT 的使能位。(R/W)

UART_TX_DONE_INT_ENA UART_TX_DONE_INT 的使能位。(R/W)

UART_RS485_PARITY_ERR_INT_ENA UART_RS485_PARITY_ERR_INT 的使能位。(R/W)

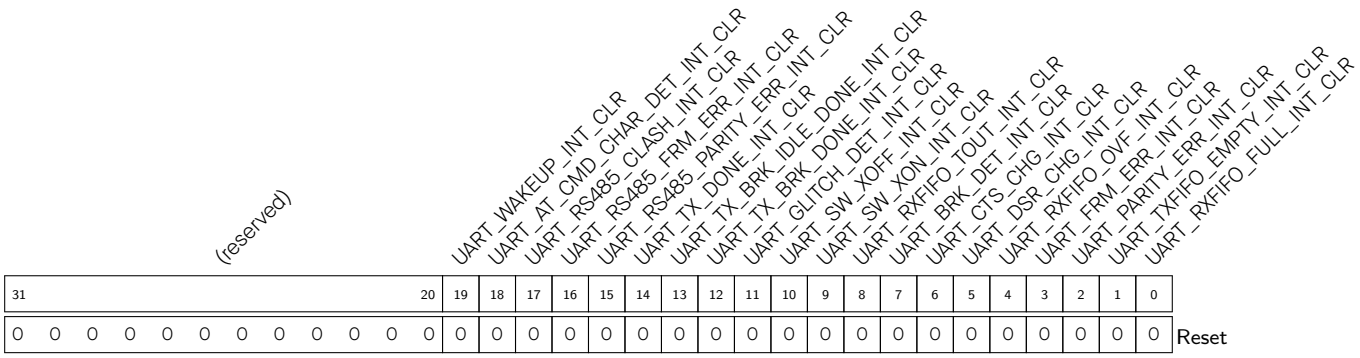
UART_RS485_FRM_ERR_INT_ENA UART_RS485_PARITY_ERR_INT 的使能位。(R/W)

UART_RS485_CLASH_INT_ENA UART_RS485_CLASH_INT 的使能位。(R/W)

UART_AT_CMD_CHAR_DET_INT_ENA UART_AT_CMD_CHAR_DET_INT 的使能位。(R/W)

UART_WAKEUP_INT_ENA UART_WAKEUP_INT 的使能位。(R/W)

Register 26.6. UART_INT_CLR_REG (0x0010)



UART_RXFIFO_FULL_INT_CLR 置位此位清除 UART_RXFIFO_FULL_INT 中断。(WT)

UART_TXFIFO_EMPTY_INT_CLR 置位此位清除 UART_TXFIFO_EMPTY_INT 中断。(WT)

UART_PARITY_ERR_INT_CLR 置位此位清除 UART_PARITY_ERR_INT 中断。(WT)

UART_FRM_ERR_INT_CLR 置位此位清除 UART_FRM_ERR_INT 中断。(WT)

UART_RXFIFO_OVF_INT_CLR 置位此位清除 UART_RXFIFO_OVF_INT 中断。(WT)

UART_DSR_CHG_INT_CLR 置位此位清除 UART_DSR_CHG_INT 中断。(WT)

UART_CTS_CHG_INT_CLR 置位此位清除 UART_CTS_CHG_INT 中断。(WT)

UART_BRK_DET_INT_CLR 置位此位清除 UART_BRK_DET_INT 中断。(WT)

UART_RXFIFO_TOUT_INT_CLR 置位此位清除 UART_RXFIFO_TOUT_INT 中断。(WT)

UART_SW_XON_INT_CLR 置位此位清除 UART_SW_XON_INT 中断。(WT)

UART_SW_XOFF_INT_CLR 置位此位清除 UART_SW_XOFF_INT 中断。(WT)

UART_GLITCH_DET_INT_CLR 置位此位清除 UART_GLITCH_DET_INT 中断。(WT)

UART_TX_BRK_DONE_INT_CLR 置位此位清除 UART_TX_BRK_DONE_INT 中断。(WT)

UART_TX_BRK_IDLE_DONE_INT_CLR 置位此位清除 UART_TX_BRK_IDLE_DONE_INT 中断。(WT)

UART_TX_DONE_INT_CLR 置位此位清除 UART_TX_DONE_INT 中断。(WT)

UART_RS485_PARITY_ERR_INT_CLR 置位此位清除 UART_RS485_PARITY_ERR_INT 中断。(WT)

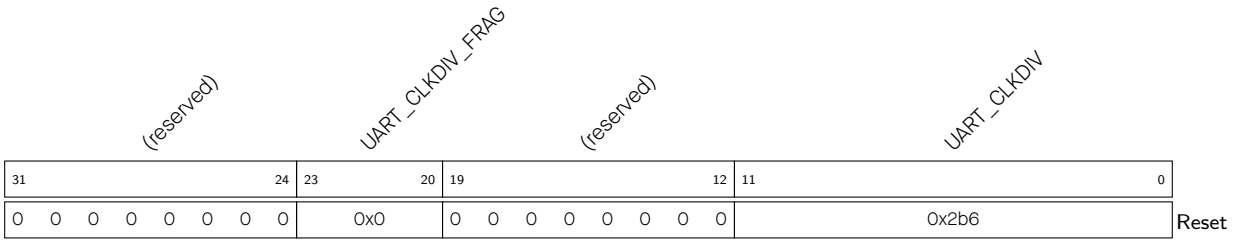
UART_RS485_FRM_ERR_INT_CLR 置位此位清除 UART_RS485_FRM_ERR_INT 中断。(WT)

UART_RS485_CLASH_INT_CLR 置位此位清除 UART_RS485_CLASH_INT 中断。(WT)

UART_AT_CMD_CHAR_DET_INT_CLR 置位此位清除 UART_AT_CMD_CHAR_DET_INT 中断。(WT)

UART_WAKEUP_INT_CLR 置位此位清除 UART_WAKEUP_INT 中断。(WT)

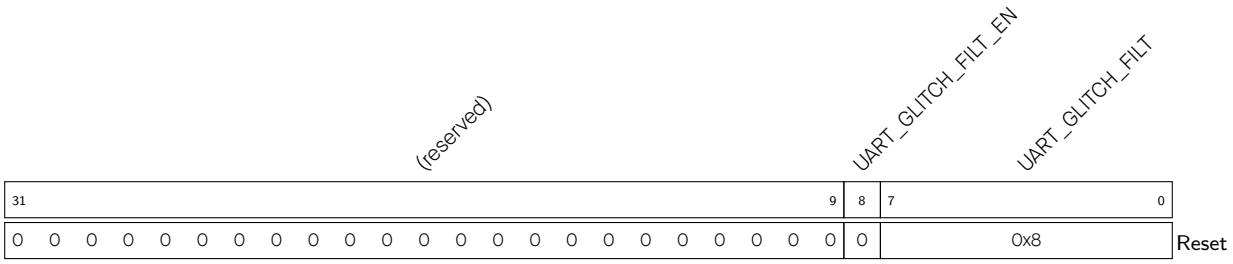
Register 26.7. UART_CLKDIV_REG (0x0014)



UART_CLKDIV 分频系数的整数部分。(R/W)

UART_CLKDIV_FRAG 分频系数的小数部分。(R/W)

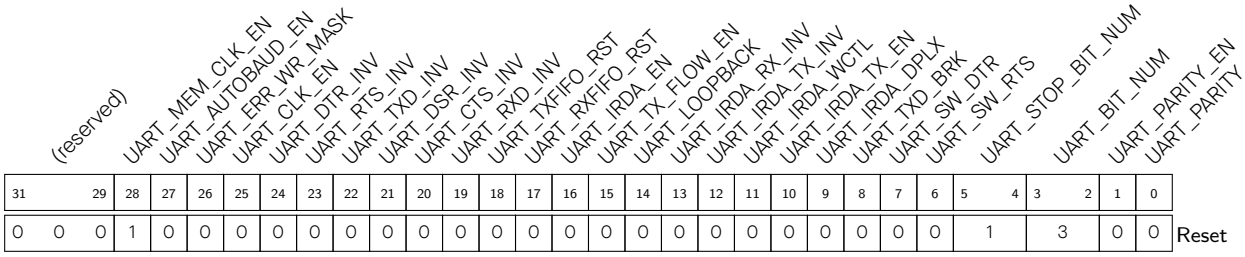
Register 26.8. UART_RX_FILT_REG (0x0018)



UART_GLITCH_FILT 宽度小于该字段值的输入脉冲会被忽略。(R/W)

UART_GLITCH_FILT_EN 置位此位，使能 RX 信号滤波器。(R/W)

Register 26.9. UART_CONF0_REG (0x0020)



UART_PARITY 配置奇偶检验方式。(R/W)

UART_PARITY_EN 置位此位使能 UART 奇偶检验。(R/W)

UART_BIT_NUM 设置数据长度。(R/W)

UART_STOP_BIT_NUM 设置停止位的长度。(R/W)

UART_SW_RTS 该位用于配置软件流控使用的软件 RTS 信号。(R/W)

UART_SW_DTR 该位用于配置软件流控使用的软件 DTR 信号。(R/W)

UART_TXD_BRK 置位此位，使能发送器在发完数据后发送 NULL。(R/W)

UART_IRDA_DPLX 置位此位开启 IrDA 回环测试模式。(R/W)

UART_IRDA_TX_EN IrDA 发送器的启动使能位。(R/W)

UART_IRDA_WCTL 0：将 IrDA 发送器的第 11 位置 0；1：IrDA 发送器的第 11 位与第 10 位相同。(R/W)

UART_IRDA_TX_INV 置位此位翻转 IrDA 发送器的电平。(R/W)

UART_IRDA_RX_INV 置位此位翻转 IrDA 接收器的电平。(R/W)

UART_LOOPBACK 置位此位开启 UART 回环测试模式。(R/W)

UART_TX_FLOW_EN 置位此位使能发送器的流控功能。(R/W)

UART_IRDA_EN 置位此位使能 IrDA 协议。(R/W)

UART_RXFIFO_RST 置位此位复位 UART RX FIFO。(R/W)

UART_TXFIFO_RST 置位此位复位 UART TX FIFO。(R/W)

UART_RXD_INV 置位此位翻转 UART RXD 信号电平。(R/W)

UART_CTS_INV 置位此位翻转 UART CTS 信号电平。(R/W)

UART_DSR_INV 置位此位翻转 UART DSR 信号电平。(R/W)

UART_TXD_INV 置位此位翻转 UART TXD 信号电平。(R/W)

UART_RTS_INV 置位此位翻转 UART RTS 信号电平。(R/W)

UART_DTR_INV 置位此位翻转 UART DTR 信号电平。(R/W)

见下页...

Register 26.9. UART_CONF0_REG (0x0020)

接上页...

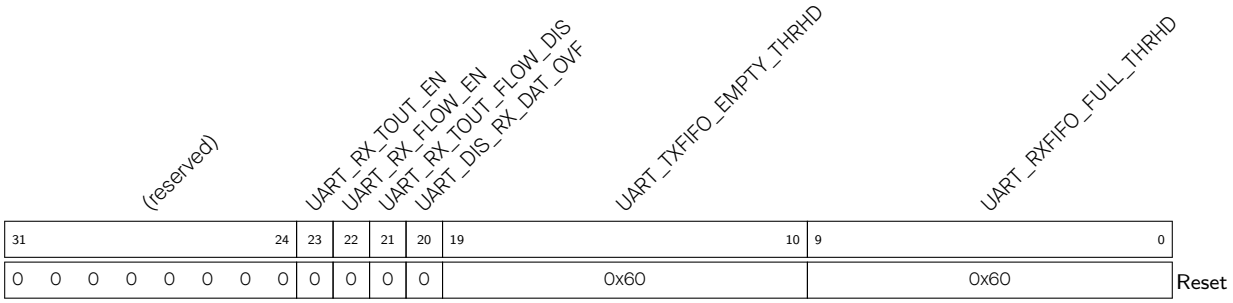
UART_CLK_EN 0：仅在应用写寄存器时支持时钟；1：强制为寄存器开启时钟。(R/W)

UART_ERR_WR_MASK 0：若数据错误，接收器仍存储；1：若数据错误，接收器不再将数据存入 FIFO。(R/W)

UART_AUTOBAUD_EN 波特率检测的使能信号。(R/W)

UART_MEM_CLK_EN UART RAM 门控使能信号。(R/W)

Register 26.10. UART_CONF1_REG (0x0024)



UART_RXFIFO_FULL_THRHD 接收器接收数据多于该字段的值时产生 UART_RXFIFO_FULL_INT 中断。(R/W)

UART_TXFIFO_EMPTY_THRHD TX FIFO 中的数据少于该字段的值时产生 UART_TXFIFO_EMPTY_INT 中断。(R/W)

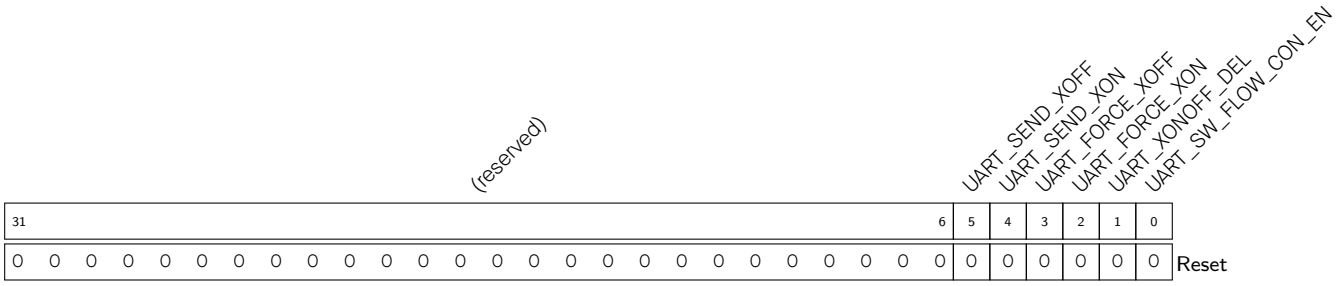
UART_DIS_RX_DAT_OVF 关闭 UART RX 数据溢出检测。(R/W)

UART_RX_TOUT_FLOW_DIS 使用硬件流控时置位此位停止堆积 idle_cnt。(R/W)

UART_RX_FLOW_EN UART 接收器流控功能的使能位。(R/W)

UART_RX_TOUT_EN UART 接收器超时功能的使能位。(R/W)

Register 26.11. UART_FLOW_CONF_REG (0x0034)



UART_SW_FLOW_CON_EN 置位此位使能软件流控。UART 接收到 UART_XON_CHAR 或 UART_XOFF_CHAR 配置的流控字符 XON 或 XOFF 时，UART_SW_XON_INT 或 UART_SW_XOFF_INT 中断可在使能时触发。(R/W)

UART_XONOFF_DEL 置位此位移除接收数据中的流控字符。(R/W)

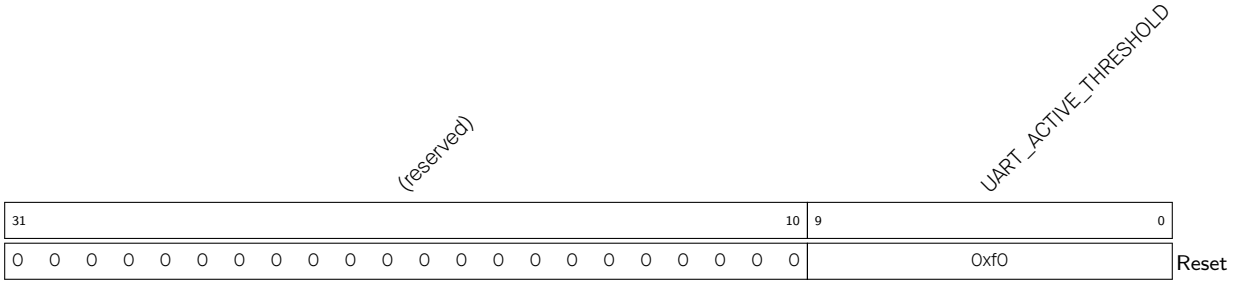
UART_FORCE_XON 置位此位让发送器继续发送数据。(R/W)

UART_FORCE_XOFF 置位此位让发送器停止发送数据。(R/W)

UART_SEND_XON 置位此位发送 XON 字符。此位由硬件自动清除。(R/W/SS/SC)

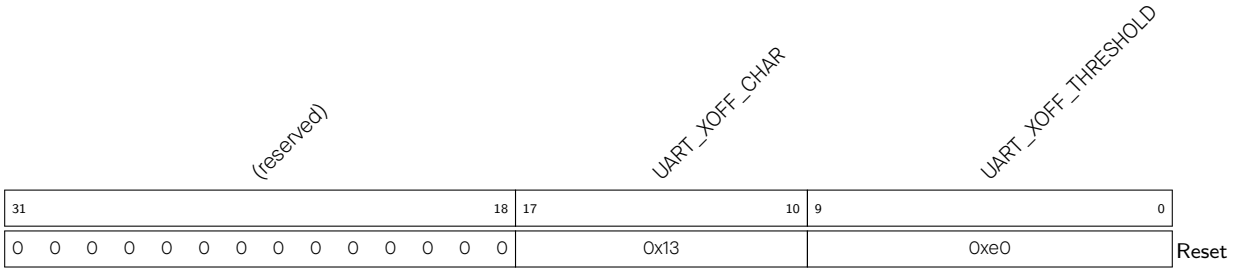
UART_SEND_XOFF 置位此位发送 XOFF 字符。此位由硬件自动清除。(R/W/SS/SC)

Register 26.12. UART_SLEEP_CONF_REG (0x0038)



UART_ACTIVE_THRESHOLD 输入 RXD 沿变化次数超过该字段的值加 3 时，UART 从 Light-sleep 模式唤醒。(R/W)

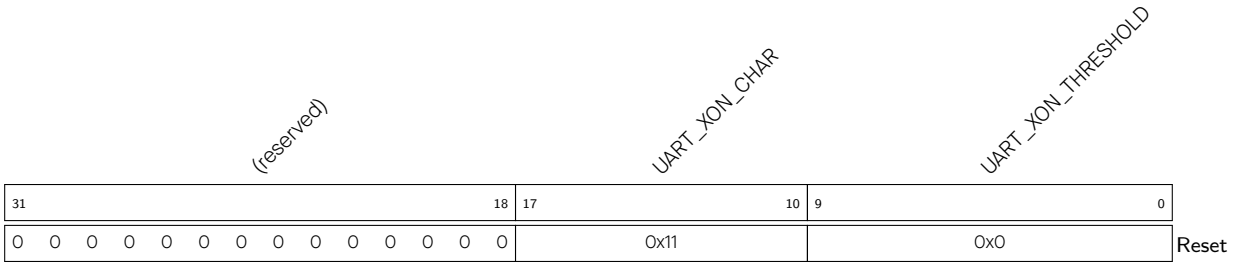
Register 26.13. UART_SWFC_CONF0_REG (0x003C)



UART_XOFF_THRESHOLD RX FIFO 中的数据超过该字段的值且 UART_SW_FLOW_CON_EN 置 1 时，发送 XOFF 字符。(R/W)

UART_XOFF_CHAR 存储 XOFF 流控字符。(R/W)

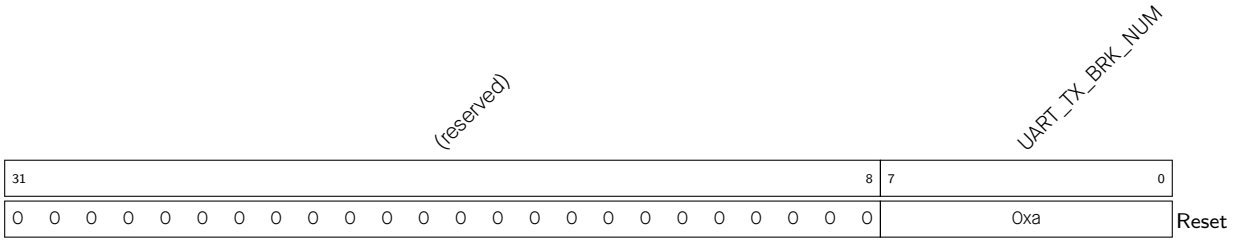
Register 26.14. UART_SWFC_CONF1_REG (0x0040)



UART_XON_THRESHOLD RX FIFO 中的数据小于该字段的值且 UART_SW_FLOW_CON_EN 置 1 时，发送 XON 字符。(R/W)

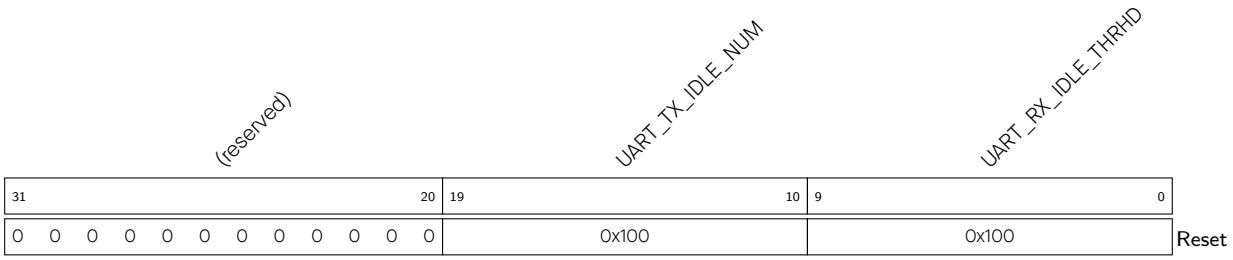
UART_XON_CHAR 存储 XON 流控字符。(R/W)

Register 26.15. UART_TXBRK_CONF_REG (0x0044)



UART_TX_BRK_NUM 配置数据发完后待发 NULL 字符的数量。UART_TXD_BRK 置 1 时有意义。(R/W)

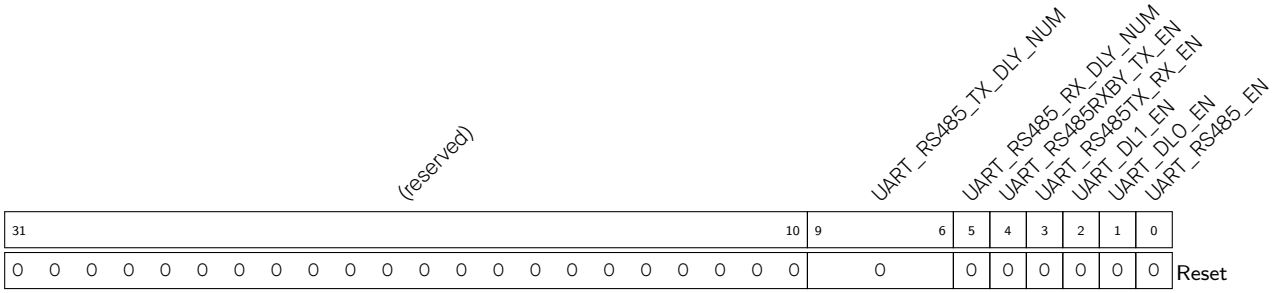
Register 26.16. UART_IDLE_CONF_REG (0x0048)



UART_RX_IDLE_THRHD 接收器接收一字节数据所需时间超过该字段的值时产生帧结束信号，单位是比特时间（即传输一个比特所需的时间）。(R/W)

UART_TX_IDLE_NUM 配置两次数据传输的间隔时间，单位是比特时间（即传输一个比特所需的时间）。(R/W)

Register 26.17. UART_RS485_CONF_REG (0x004C)



UART_RS485_EN 置位此位选择 RS485 模式。(R/W)

UART_DL0_EN 配置是否在起始位之前增加一位延时。

0：不增加 1：增加

(R/W)

UART_DL1_EN 配置是否在停止位之后增加一位延时。

0：不增加

1：增加

(R/W)

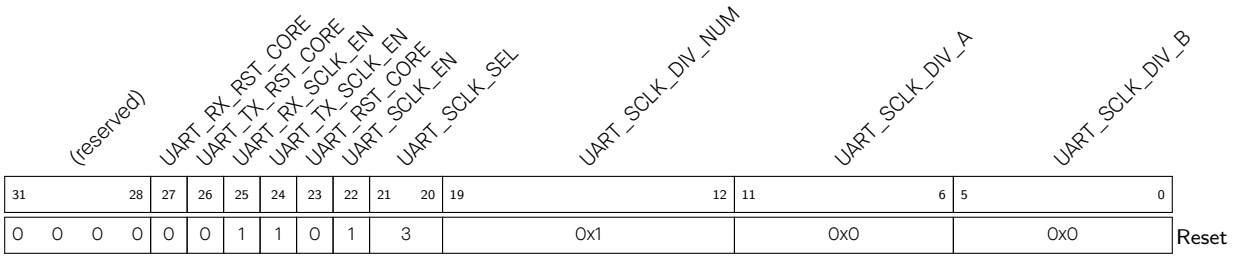
UART_RS485TX_RX_EN 发送器在 RS485 模式下发送数据时，置位此位使能接收器接收数据。(R/W)

UART_RS485RXBY_TX_EN 1'h1：RS485 接收器线路繁忙时使能 RS485 发送器发送数据。(R/W)

UART_RS485_RX_DLY_NUM 延迟接收器的内部数据信号。(R/W)

UART_RS485_TX_DLY_NUM 延迟发送器的内部数据信号。(R/W)

Register 26.18. UART_CLK_CONF_REG (0x0078)



UART_SCLK_DIV_B 分频系数的分母。(R/W)

UART_SCLK_DIV_A 分频系数的分子。(R/W)

UART_SCLK_DIV_NUM 分频系数的整数部分。(R/W)

UART_SCLK_SEL 选择 UART 时钟源。1：APB_CLK；2：RC_FAST_CLK；3：XTAL_CLK。(R/W)

UART_SCLK_EN 置位此位，使能 UART TX/RX 使能。(R/W)

UART_RST_CORE 向此位先写 1 后写 0，复位 UART TX/RX。(R/W)

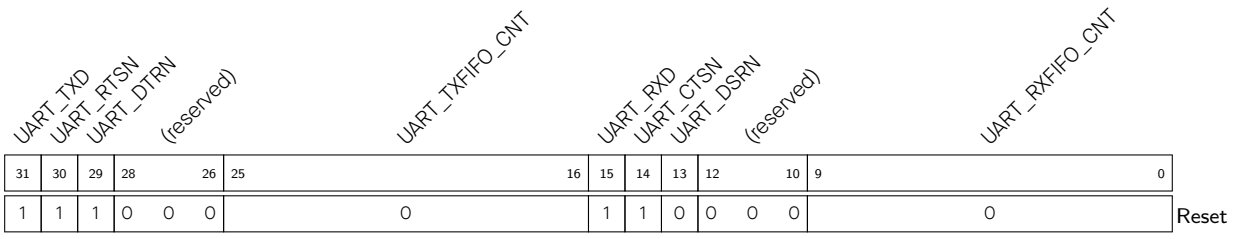
UART_TX_SCLK_EN 置位此位，使能 UART TX 时钟。(R/W)

UART_RX_SCLK_EN 置位此位，使能 UART RX 时钟。(R/W)

UART_TX_RST_CORE 向此位先写 1 后写 0，复位 UART TX。(R/W)

UART_RX_RST_CORE 向此位先写 1 后写 0，复位 UART RX。(R/W)

Register 26.19. UART_STATUS_REG (0x001C)



UART_RXFIFO_CNT 存储 RX FIFO 中有效数据的字节数。(RO)

UART_DSRN 该位表示内部 UART DSR 信号的电平值。(RO)

UART_CTSN 该位表示内部 UART CTS 信号的电平值。(RO)

UART_RXD 该位表示内部 UART RXD 信号的电平值。(RO)

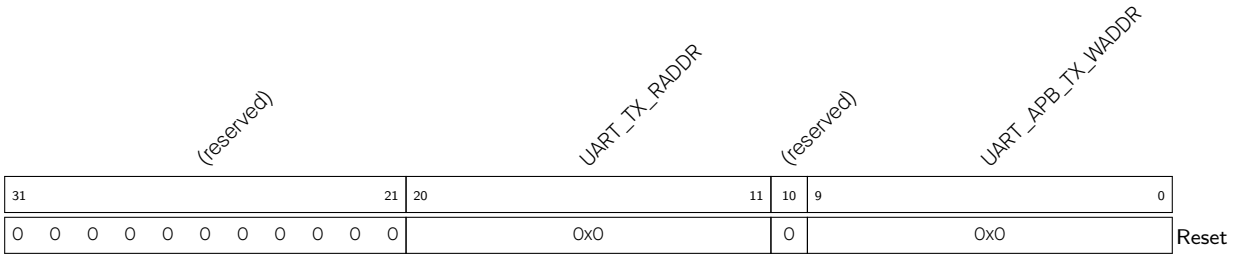
UART_TXFIFO_CNT 存储 TX FIFO 中数据的字节数。(RO)

UART_DTRN 此位表示内部 UART DTR 信号的电平。(RO)

UART_RTSN 此位表示内部 UART RTS 信号的电平。(RO)

UART_TXD 此位表示内部 UART TXD 信号的电平。(RO)

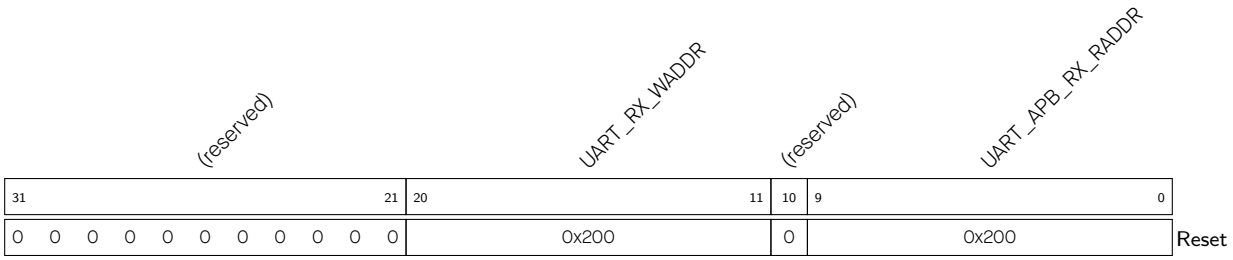
Register 26.20. UART_MEM_TX_STATUS_REG (0x0064)



UART_APB_TX_WADDR 在软件通过 APB 总线写 TX FIFO 时存储 TX FIFO 的偏移地址。(RO)

UART_TX_RADDR 在 TX FSM 通过 Tx_FIFO_Ctrl 读取数据时存储 TX FIFO 的偏移地址。(RO)

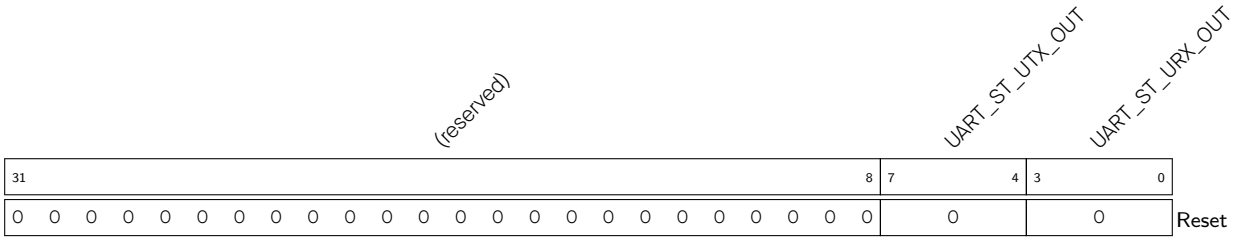
Register 26.21. UART_MEM_RX_STATUS_REG (0x0068)



UART_APB_RX_RADDR 在软件通过 APB 总线读取 RX FIFO 数据时存储 RX FIFO 的偏移地址。UART0 为 0x200，UART1 为 0x280，UART2 为 0x300。(RO)

UART_RX_WADDR 在 Rx_FIFO_Ctrl 写 RX FIFO 时存储 RX FIFO 的偏移地址。UART0 为 0x200，UART1 为 0x280，UART2 为 0x300。(RO)

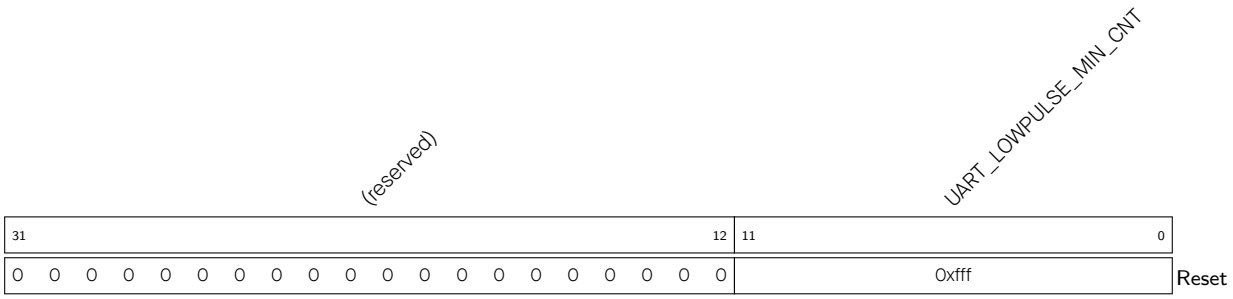
Register 26.22. UART_FSM_STATUS_REG (0x006C)



UART_ST_URX_OUT 接收器的状态字段。(RO)

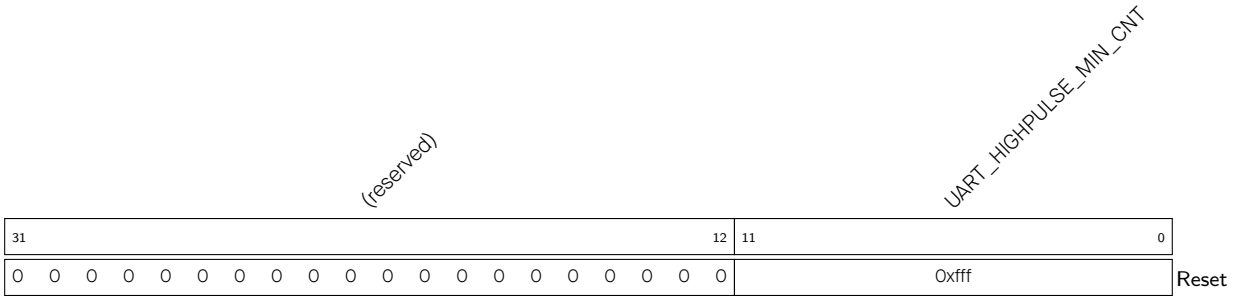
UART_ST_UTX_OUT 发送器的状态字段。(RO)

Register 26.23. UART_LOWPULSE_REG (0x0028)



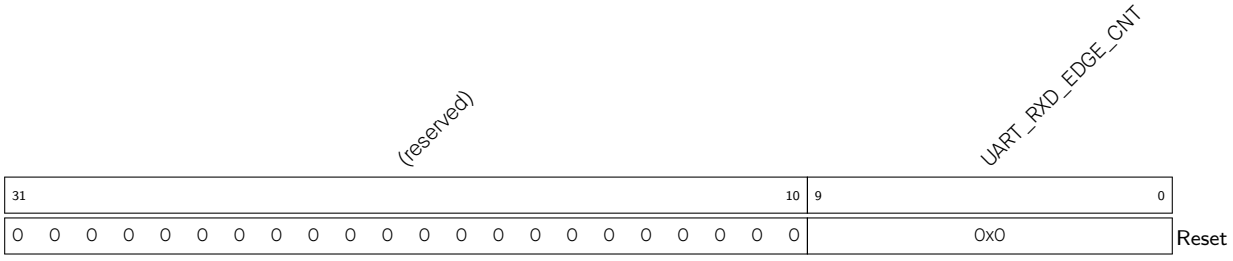
UART_LOWPULSE_MIN_CNT 存储低电平脉冲的最短持续时间，用于波特率检测，单位是 APB_CLK 时钟周期。(RO)

Register 26.24. UART_HIGHPULSE_REG (0x002C)



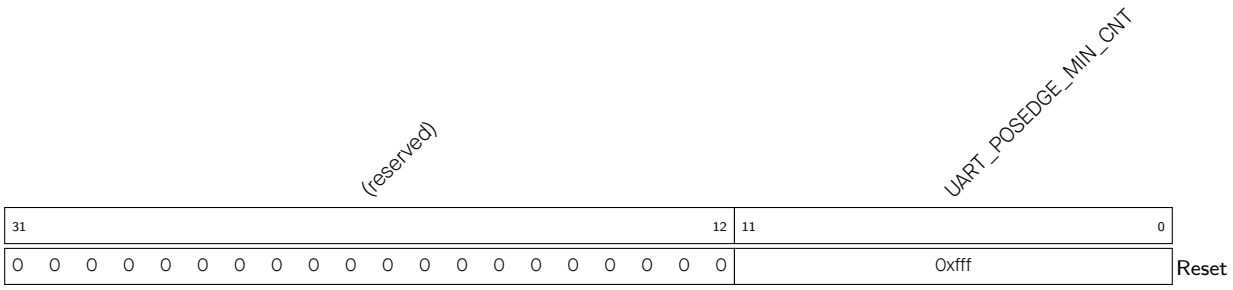
UART_HIGHPULSE_MIN_CNT 存储最长高电平脉冲持续时间。用于波特率检测，单位是 APB_CLK 时钟周期。(RO)

Register 26.25. UART_RXD_CNT_REG (0x0030)



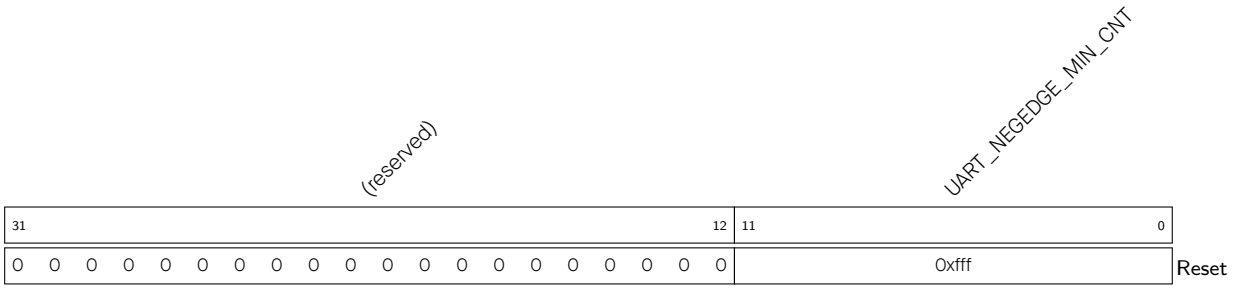
UART_RXD_EDGE_CNT 存储 RXD 沿变化的次数。用于波特率检测。(RO)

Register 26.26. UART_POSPULSE_REG (0x0070)



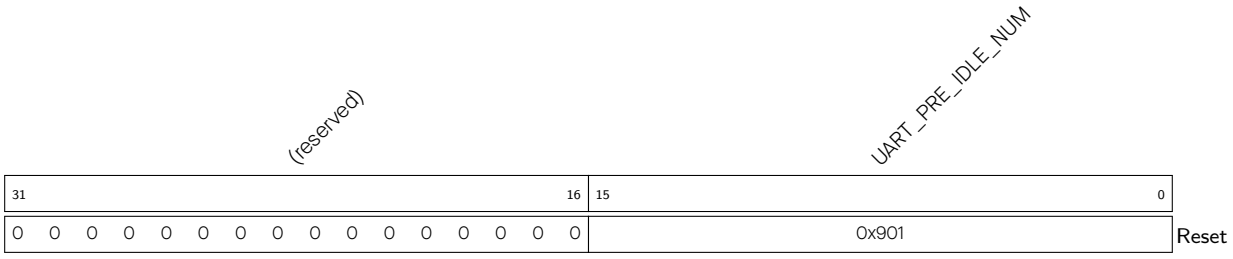
UART_POSEDGE_MIN_CNT 存储两个上升沿之间的最小输入时钟计数值。用于波特率检测。(RO)

Register 26.27. UART_NEGPULSE_REG (0x0074)



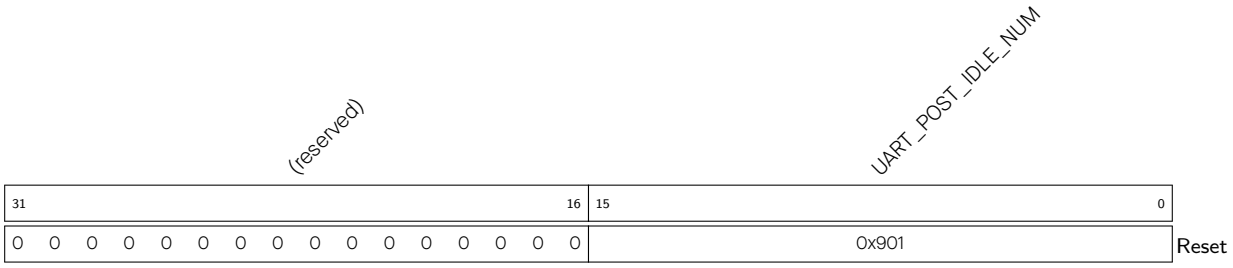
UART_NEGEDGE_MIN_CNT 存储两个下降沿之间的最小输入时钟计数值。用于波特率检测。(RO)

Register 26.28. UART_AT_CMD_PRECNT_REG (0x0050)



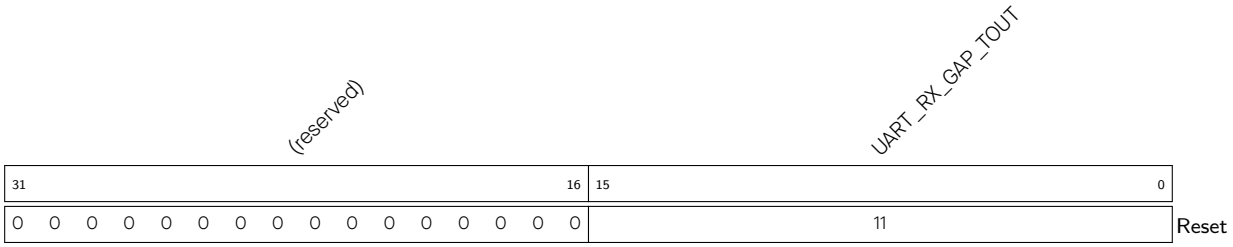
UART_PRE_IDLE_NUM 配置接收器接收第一个 AT_CMD 字符前的空闲时间，单位是比特时间（即传输一个比特所需的时间）。(R/W)

Register 26.29. UART_AT_CMD_POSTCNT_REG (0x0054)



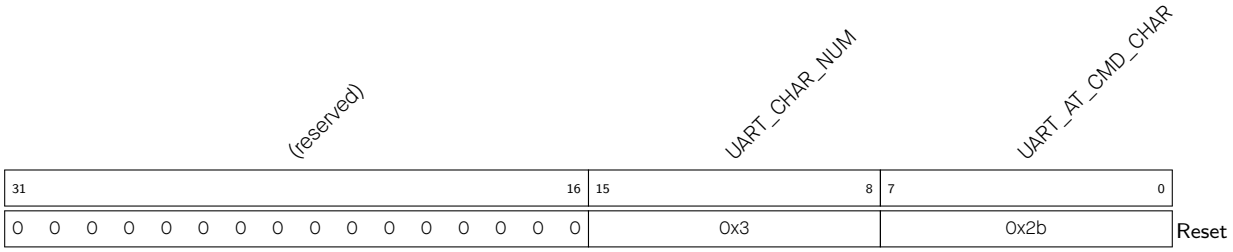
UART_POST_IDLE_NUM 配置最后一个 AT_CMD 字符和后续数据的间隔时间，单位是比特时间（即传输一个比特所需的时间）。(R/W)

Register 26.30. UART_AT_CMD_GAPTOUT_REG (0x0058)



UART_RX_GAP_TOUT 配置 AT_CMD 字符的间隔时间，单位是比特时间（即传输一个比特所需的时间）。(R/W)

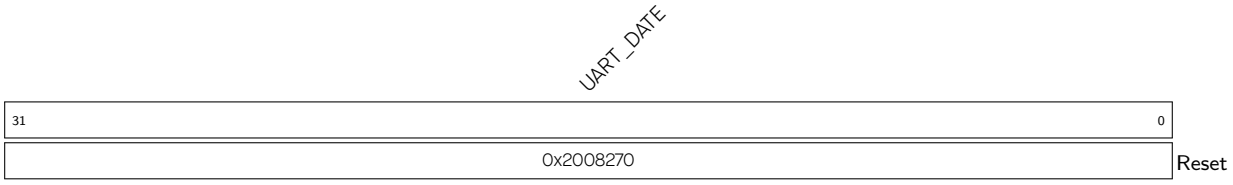
Register 26.31. UART_AT_CMD_CHAR_REG (0x005C)



UART_AT_CMD_CHAR 配置 AT_CMD 字符的内容。(R/W)

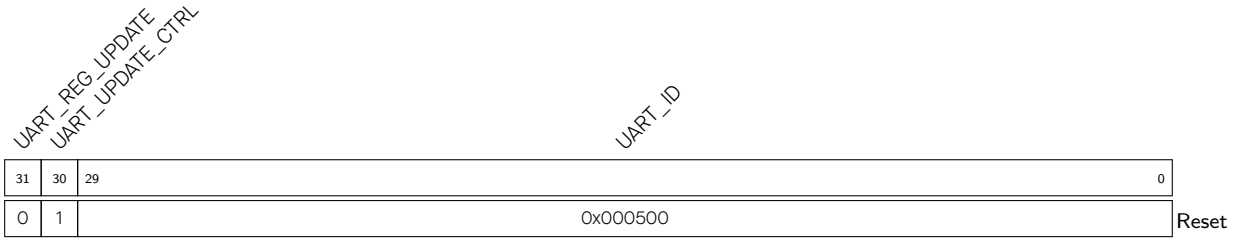
UART_CHAR_NUM 配置接收器接收连续 AT_CMD 字符的个数。(R/W)

Register 26.32. UART_DATE_REG (0x007C)



UART_DATE 版本控制寄存器。(R/W)

Register 26.33. UART_ID_REG (0x0080)



UART_ID 配置 UART_ID。(R/W)

UART_UPDATE_CTRL 用于控制同步模式。0：寄存器配置后，软件需向 UART_REG_UPDATE 写 1 同步寄存器。；1：寄存器自动同步至 UART Core 时钟域。(R/W)

UART_REG_UPDATE 软件向该字段写 1，将寄存器值同步到 UART Core 时钟域。该字段在同步完成后由硬件自清。(R/W/SC)

26.7.2 UHCI 寄存器

本小节的所有地址均为相对于 [UHCI 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 26.34. UHCI_CONF0_REG (0x0000)



UHCI_TX_RST 向此位先写 1 再写 0 复位解码状态机。(R/W)

UHCI_RX_RST 向此位先写 1 再写 0 复位编码状态机。(R/W)

UHCI_UART0_CE 置位此位，将 UHCI 和 UART0 相连。(R/W)

UHCI_UART1_CE 置位此位，将 UHCI 和 UART1 相连。(R/W)

UHCI_UART2_CE 置位此位，将 UHCI 和 UART2 相连。(R/W)

UHCI_SEPER_EN 置位此位，使用特殊字符分隔数据帧。(R/W)

UHCI_HEAD_EN 置位此位，用格式报头编码数据包。(R/W)

UHCI_CRC_REC_EN 置位此位，使能 UHCI 接收 16 位 CRC。(R/W)

UHCI_UART_IDLE_EOF_EN 若此位置 1，UHCI 在 UART 空闲时停止接收有效载荷。(R/W)

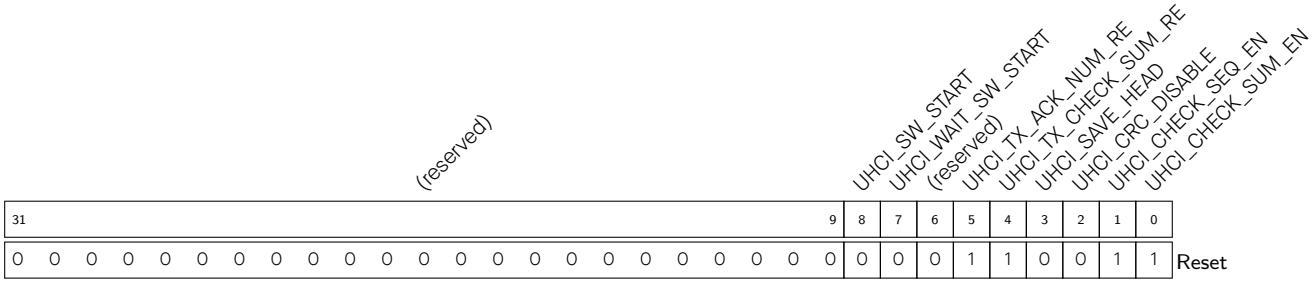
UHCI_LEN_EOF_EN 若此位置 1，UHCI 解码器接收字节数达到指定值时停止接收有效载荷数据。UHCI_HEAD_EN 为 1 时，该值是 UCHI 数据包报头明确的有效负载长度；UHCI_HEAD_EN 为 0 时，该值为配置值。若此位置 0，UHCI 解码器在接收到 0xC0 后停止接收有效载荷数据。(R/W)

UHCI_ENCODE_CRC_EN 置位此位，在有效载荷末尾加 16 位 CCITT-CRC 开始数据完整性检测。(R/W)

UHCI_CLK_EN 0：仅在应用写寄存器时支持时钟；1：强制为寄存器开启时钟。(R/W)

UHCI_UART_RX_BRK_EOF_EN 若此位置 1，UART 收到 NULL 帧后 UHCI 会停止接收有效载荷。(R/W)

Register 26.35. UHCI_CONF1_REG (0x0018)



UHCI_CHECK_SUM_EN UHCI 接收数据包时检查报头校验和的使能位。(R/W)

UHCI_CHECK_SEQ_EN UHCI 接收数据包时检查序列号的使能位。(R/W)

UHCI_CRC_DISABLE 置位此位，支持 CRC 计算。UHCI 包中的数据完整性检测位应为 1。(R/W)

UHCI_SAVE_HEAD 置位此位，在 UHCI 接收数据包时保存数据包报头。(R/W)

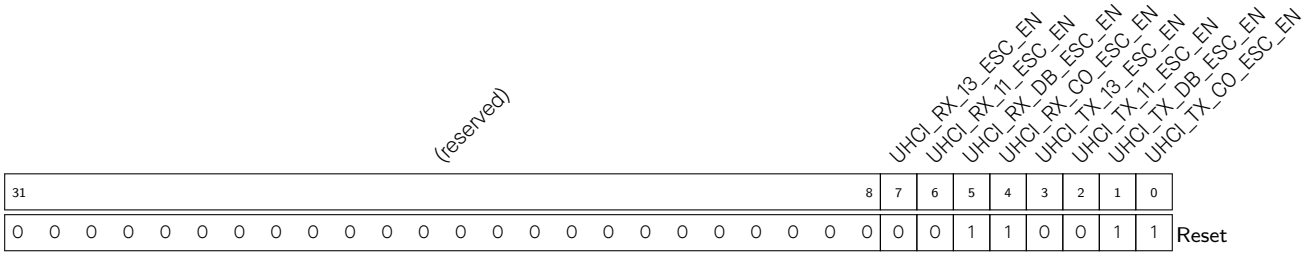
UHCI_TX_CHECK_SUM_RE 置位此位，用校验和编码数据包。(R/W)

UHCI_TX_ACK_NUM_RE 准备发送可靠数据包时，置位此位用 ACK 编码该数据包。(R/W)

UHCI_WAIT_SW_START 此位置 1 时，UHCI 编码器跳至 ST_SW_WAIT 状态。(R/W)

UHCI_SW_START 若当前 UHCI_ENCODE_STATE 为 ST_SW_WAIT 状态，此位置 1 时 UHCI 开始发送数据包。(R/W/SC)

Register 26.36. UHCI_ESCAPE_CONF_REG (0x0024)



UHCI_TX_C0_ESC_EN 置位此位，在 DMA 接收数据时解析字符 0xC0。(R/W)

UHCI_TX_DB_ESC_EN 置位此位，在 DMA 接收数据时解析字符 0xDB。(R/W)

UHCI_TX_11_ESC_EN 置位此位，在 DMA 接收数据时解析流控字符 0x11。(R/W)

UHCI_TX_13_ESC_EN 置位此位，在 DMA 接收数据时解析流控字符 0x13。(R/W)

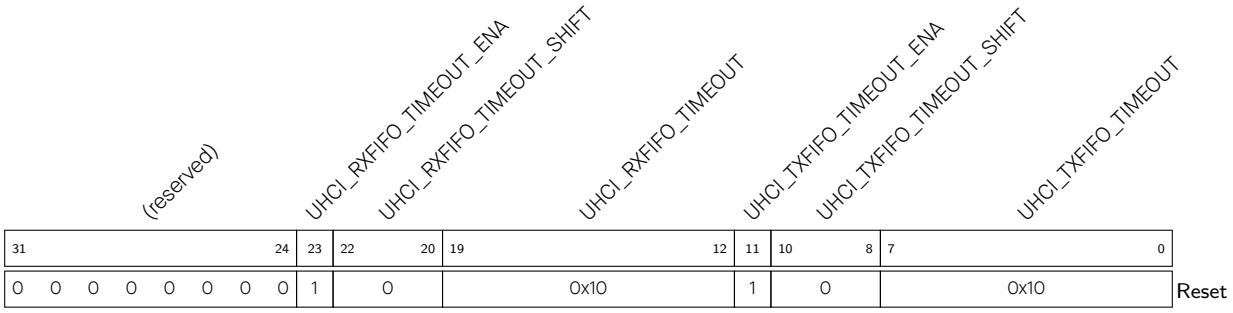
UHCI_RX_C0_ESC_EN 置位此位，在 DMA 发送数据时用特殊字符替换字符 0xC0。(R/W)

UHCI_RX_DB_ESC_EN 置位此位，在 DMA 发送数据时用特殊字符替换字符 0xDB。(R/W)

UHCI_RX_11_ESC_EN 置位此位，在 DMA 发送数据时用特殊字符替换流控字符 0x11。(R/W)

UHCI_RX_13_ESC_EN 置位此位，在 DMA 发送数据时用特殊字符替换流控字符 0x13。(R/W)

Register 26.37. UHCI_HUNG_CONF_REG (0x0028)



UHCI_TXFIFO_TIMEOUT 存储超时值。DMA 接收数据超时时产生 UHCI_TX_HUNG_INT 中断。(R/W)

UHCI_TXFIFO_TIMEOUT_SHIFT 用于配置计数最大值。(R/W)

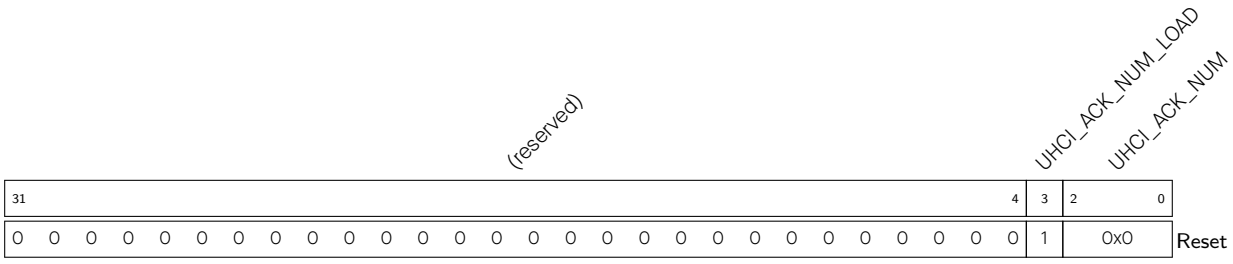
UHCI_TXFIFO_TIMEOUT_ENA TX FIFO 接收数据超时的使能位。(R/W)

UHCI_RXFIFO_TIMEOUT 存储超时值。DMA 读取 RAM 数据超时时产生 UHCI_RX_HUNG_INT 中断。(R/W)

UHCI_RXFIFO_TIMEOUT_SHIFT 用于配置计数最大值。(R/W)

UHCI_RXFIFO_TIMEOUT_ENA DMA 发送数据超时的使能位。(R/W)

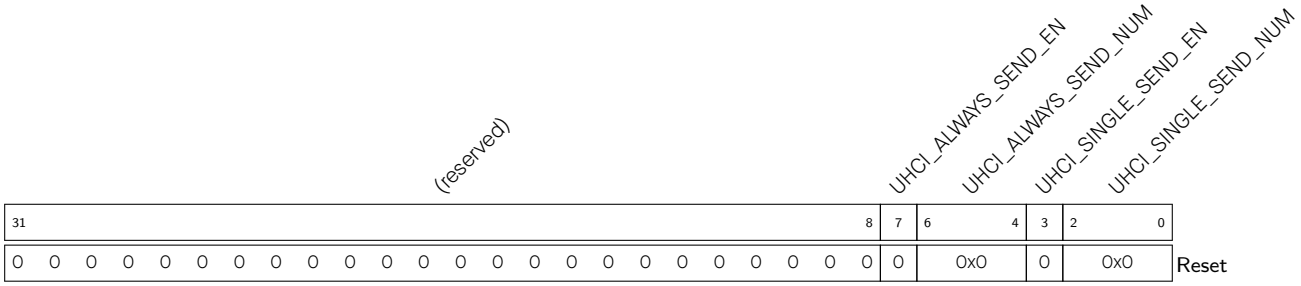
Register 26.38. UHCI_ACK_NUM_REG (0x002C)



UHCI_ACK_NUM 软件流控中使用的 ACK 值。(R/W)

UHCI_ACK_NUM_LOAD 置位此位，加载 UHCI_ACK_NUM 配置的值。(WT)

Register 26.39. UHCI_QUICK_SENT_REG (0x0034)



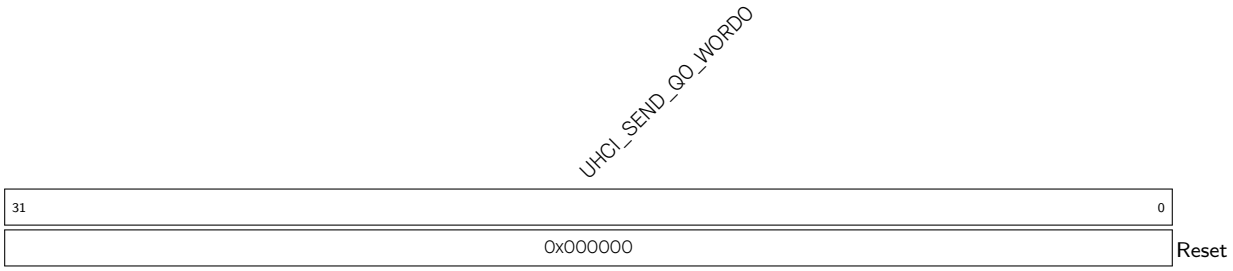
UHCI_SINGLE_SEND_NUM 设定 single_send 模式。(R/W)

UHCI_SINGLE_SEND_EN 置位此位使能 single_send 模式发送短包。(R/W/SC)

UHCI_ALWAYS_SEND_NUM 设定 always_send 模式。(R/W)

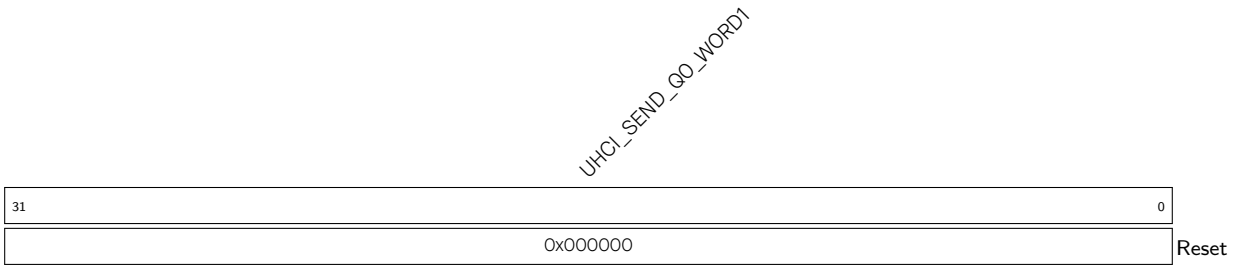
UHCI_ALWAYS_SEND_EN 置位此位使能 always_send 模式发送短包。(R/W)

Register 26.40. UHCI_REG_Q0_WORD0_REG (0x0038)



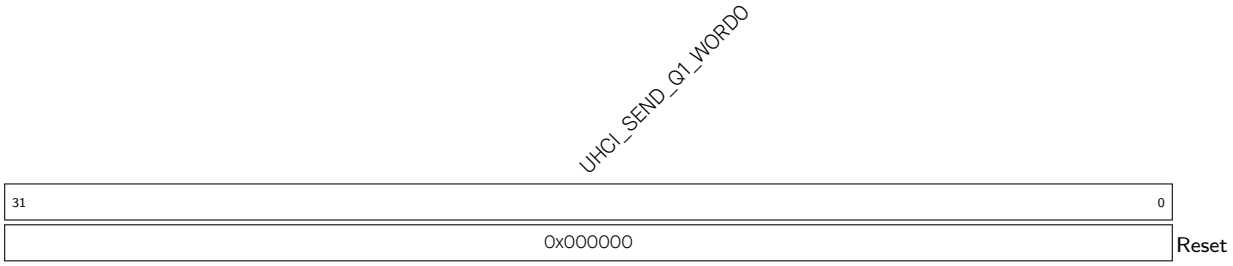
UHCI_SEND_Q0_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.41. UHCI_REG_Q0_WORD1_REG (0x003C)



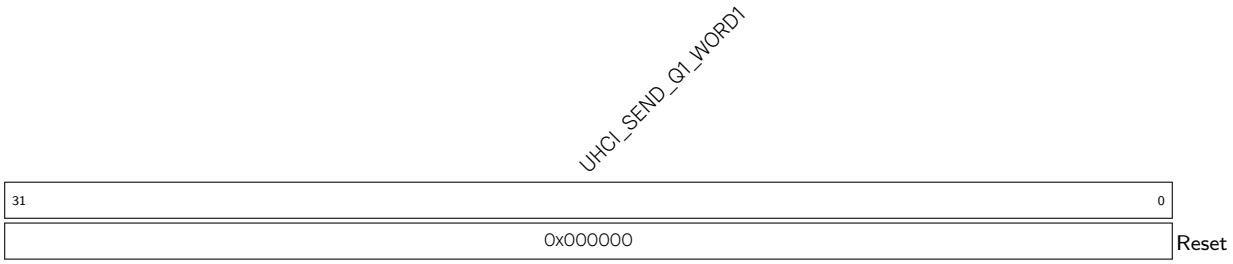
UHCI_SEND_Q0_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.42. UHCI_REG_Q1_WORD0_REG (0x0040)



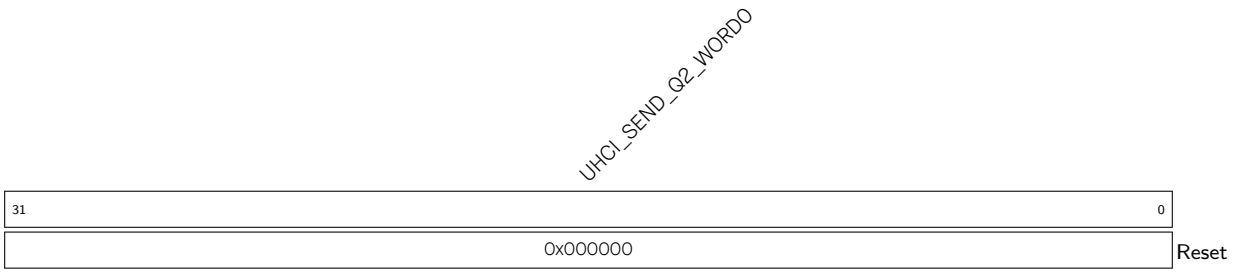
UHCI_SEND_Q1_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.43. UHCI_REG_Q1_WORD1_REG (0x0044)



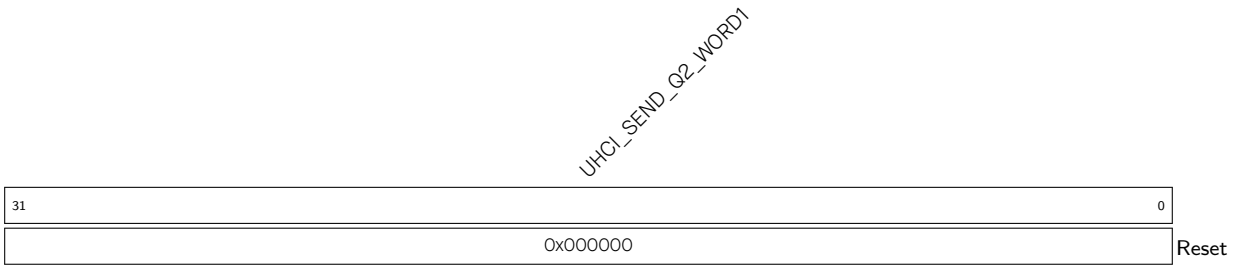
UHCI_SEND_Q1_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.44. UHCI_REG_Q2_WORD0_REG (0x0048)



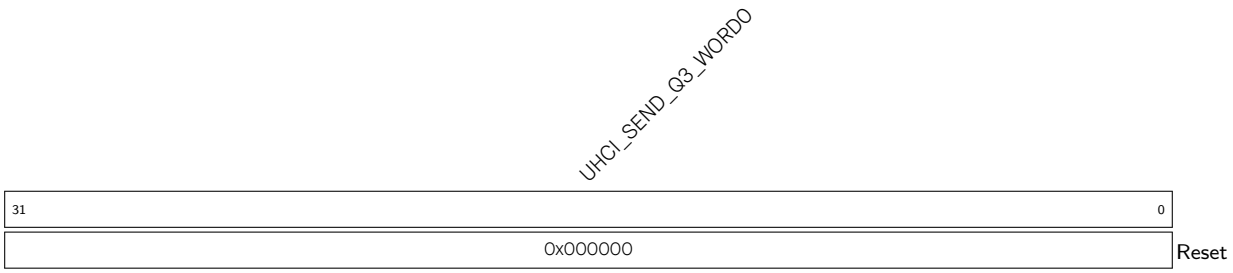
UHCI_SEND_Q2_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.45. UHCI_REG_Q2_WORD1_REG (0x004C)



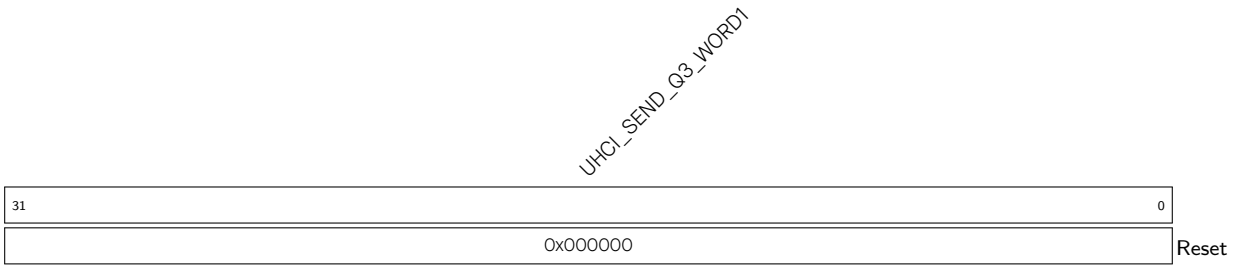
UHCI_SEND_Q2_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.46. UHCI_REG_Q3_WORD0_REG (0x0050)



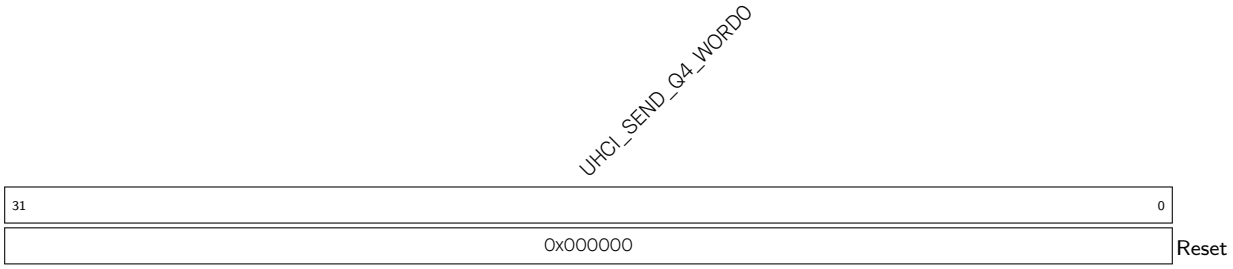
UHCI_SEND_Q3_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.47. UHCI_REG_Q3_WORD1_REG (0x0054)



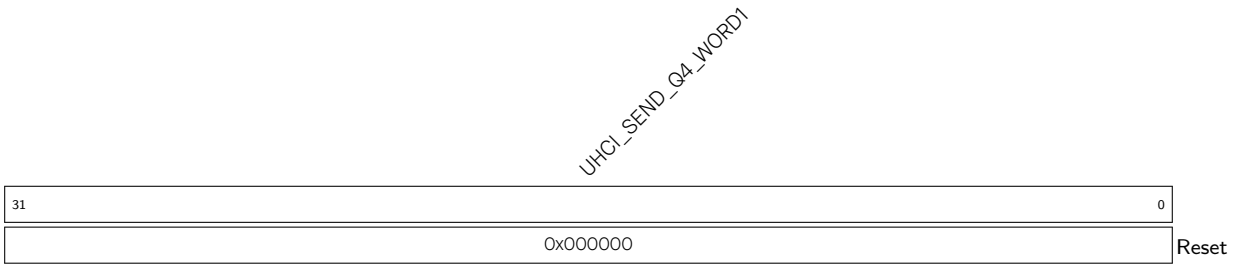
UHCI_SEND_Q3_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.48. UHCI_REG_Q4_WORD0_REG (0x0058)



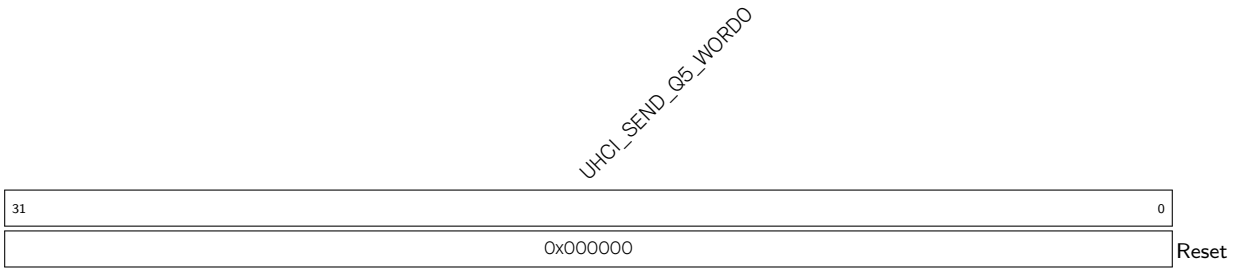
UHCI_SEND_Q4_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.49. UHCI_REG_Q4_WORD1_REG (0x005C)



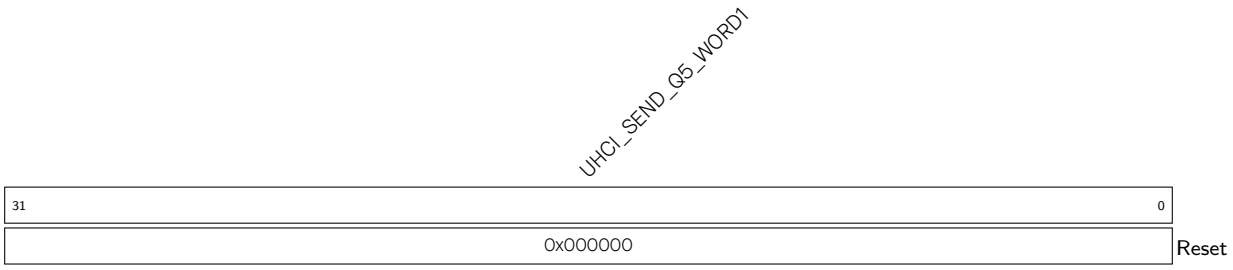
UHCI_SEND_Q4_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.50. UHCI_REG_Q5_WORD0_REG (0x0060)



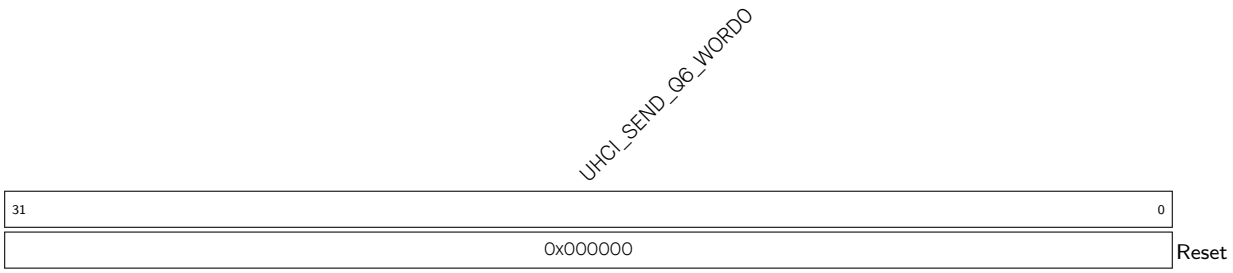
UHCI_SEND_Q5_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.51. UHCI_REG_Q5_WORD1_REG (0x0064)



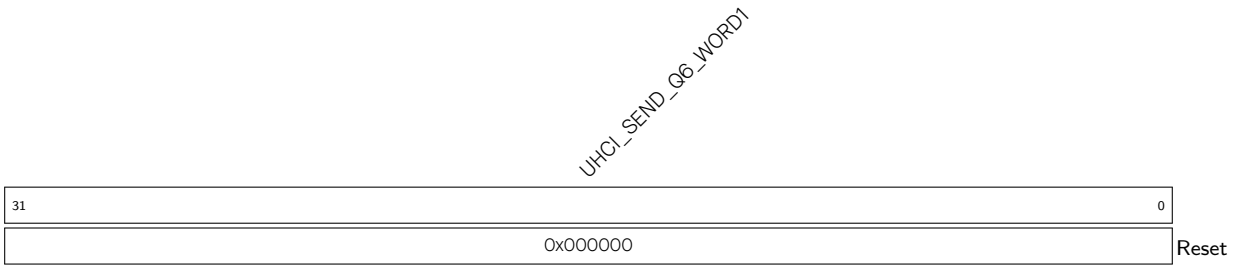
UHCI_SEND_Q5_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.52. UHCI_REG_Q6_WORD0_REG (0x0068)



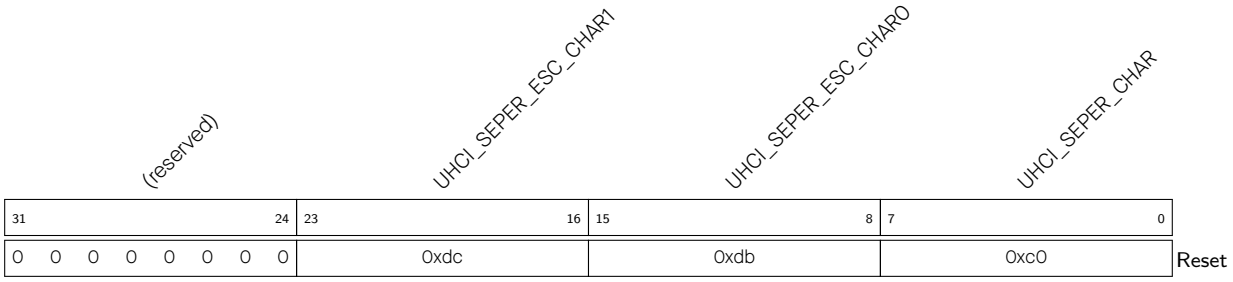
UHCI_SEND_Q6_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.53. UHCI_REG_Q6_WORD1_REG (0x006C)



UHCI_SEND_Q6_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定模式时用作快速发送寄存器。(R/W)

Register 26.54. UHCI_ESC_CONF0_REG (0x0070)

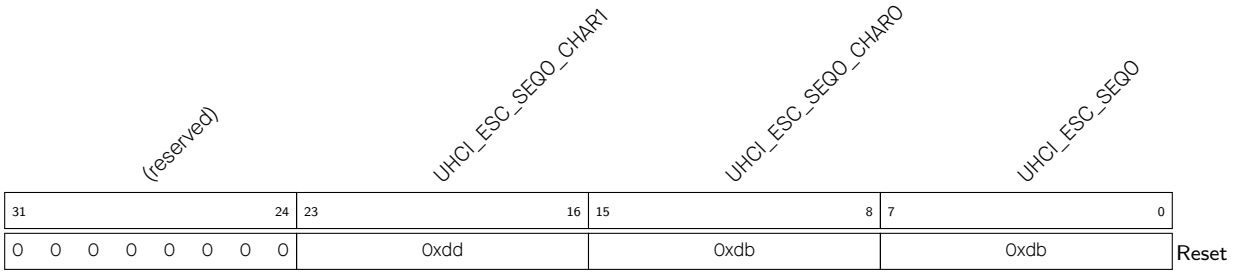


UHCI_SEPER_CHAR 定义用于编码的分隔符，默认为 0xC0。(R/W)

UHCI_SEPER_ESC_CHAR0 定义 SLIP 转义序列的第一个字符，默认为 0xDB。(R/W)

UHCI_SEPER_ESC_CHAR1 定义 SLIP 转义序列的第二个字符，默认为 0xDC。(R/W)

Register 26.55. UHCI_ESC_CONF1_REG (0x0074)

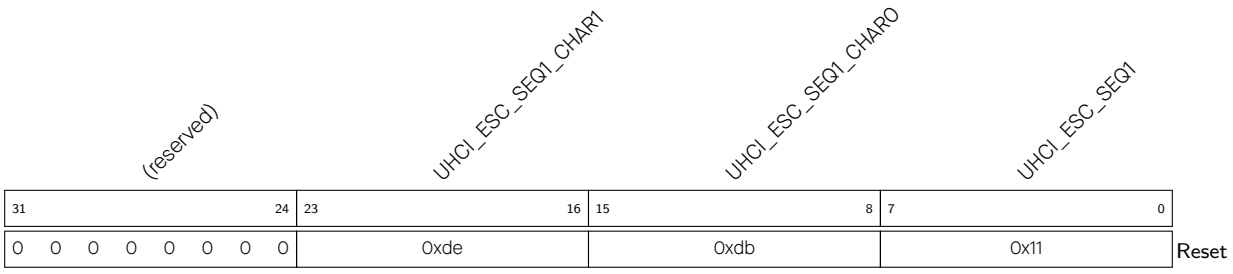


UHCI_ESC_SEQ0 定义需编码的字符，默认为用作 SLIP 转义序列第一个字符的 0xDB。(R/W)

UHCI_ESC_SEQ0_CHAR0 定义 SLIP 转义序列的第一个字符，默认为 0xDB。(R/W)

UHCI_ESC_SEQ0_CHAR1 时定义 SLIP 转义序列的第二个字符，默认为 0xDD。(R/W)

Register 26.56. UHCI_ESC_CONF2_REG (0x0078)

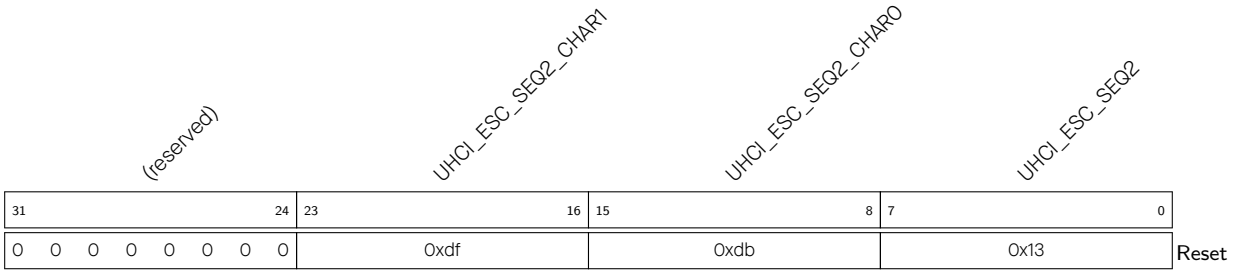


UHCI_ESC_SEQ1 定义需编码的字符，默认为用作流控字符的 0x11。(R/W)

UHCI_ESC_SEQ1_CHAR0 定义 SLIP 转义序列的第一个字符，默认为 0xDB。(R/W)

UHCI_ESC_SEQ1_CHAR1 定义 SLIP 转义序列的第二个字符，默认为 0xDE。(R/W)

Register 26.57. UHCI_ESC_CONF3_REG (0x007C)

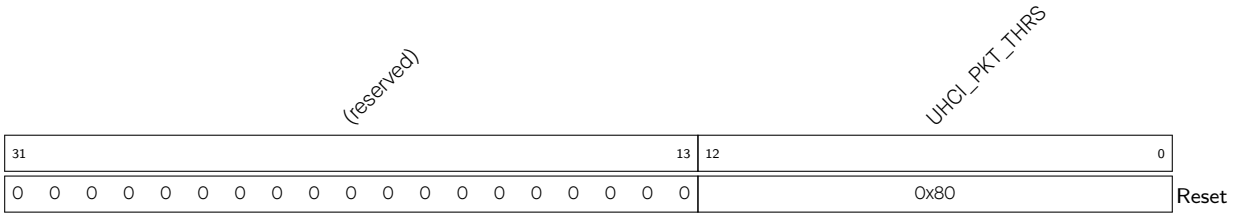


UHCI_ESC_SEQ2 定义需编码的字符，默认为用作流控字符的 0x13。(R/W)

UHCI_ESC_SEQ2_CHAR0 定义 SLIP 转义序列的第一个字符，默认为 0xDB。(R/W)

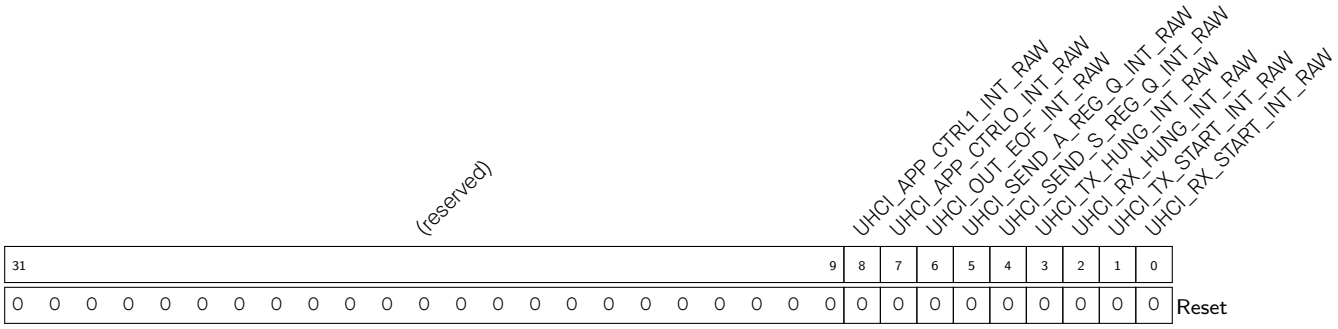
UHCI_ESC_SEQ2_CHAR1 定义 SLIP 转义序列的第二个字符，默认为 0xDF。(R/W)

Register 26.58. UHCI_PKT_THRES_REG (0x0080)



UHCI_PKT_THRS UHCI_HEAD_EN 为 0 时配置包长度的最大值。(R/W)

Register 26.59. UHCI_INT_RAW_REG (0x0004)



UHCI_RX_START_INT_RAW UHCI_RX_START_INT 中断的原始中断位。分隔符成功发送时触发中断。(R/WTC/SS)

UHCI_TX_START_INT_RAW UHCI_TX_START_INT 中断的原始中断位。DMA 检测到分隔符时触发中断。(R/WTC/SS)

UHCI_RX_HUNG_INT_RAW UHCI_RX_HUNG_INT 中断的原始中断位。DMA 接收数据所需时间超过配置值时触发中断。(R/WTC/SS)

UHCI_TX_HUNG_INT_RAW UHCI_TX_HUNG_INT 中断的原始中断位。DMA 读取 RAM 数据所需时间超过配置值时触发中断。(R/WTC/SS)

UHCI_SEND_S_REG_Q_INT_RAW UHCI_SEND_S_REG_Q_INT 中断的原始中断位。UHCI 使用 single_send 模式成功发送短包时触发中断。(R/WTC/SS)

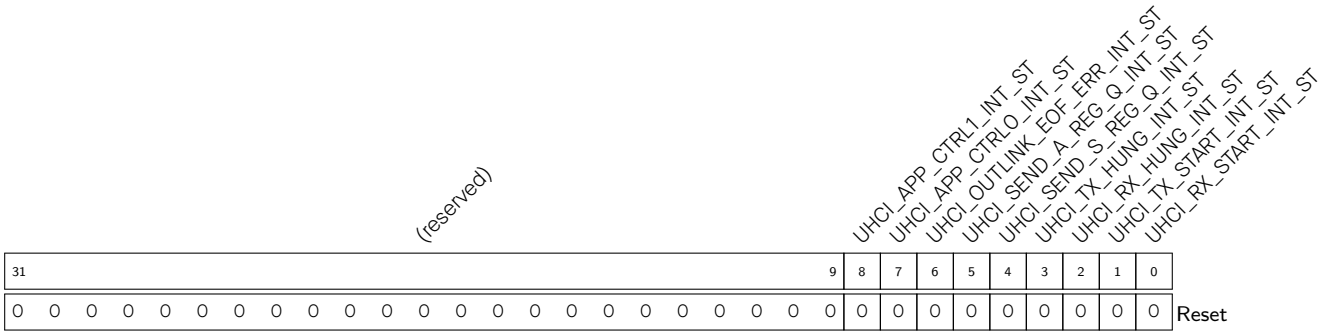
UHCI_SEND_A_REG_Q_INT_RAW UHCI_SEND_A_REG_Q_INT 中断的原始中断位。UHCI 使用 always_send 模式成功发送短包时触发中断。(R/WTC/SS)

UHCI_OUT_EOF_INT_RAW UHCI_OUT_EOF_INT 中断的原始中断位。接收数据的 EOF 有错误时触发中断。(R/WTC/SS)

UHCI_APP_CTRL0_INT_RAW UHCI_APP_CTRL0_INT 中断的原始中断位，UHCI_APP_CTRL0_IN_SET 置 1 时触发中断。(R/W)

UHCI_APP_CTRL1_INT_RAW UHCI_APP_CTRL1_INT 中断的原始中断位，UHCI_APP_CTRL1_IN_SET 置 1 时触发中断。(R/W)

Register 26.60. UHCI_INT_ST_REG (0x0008)



UHCI_RX_START_INT_ST UHCI_RX_START_INT_ENA 置 1 时 UHCI_RX_START_INT 中断的屏蔽中断位。(RO)

UHCI_TX_START_INT_ST UHCI_TX_START_INT_ENA 置 1 时 UHCI_TX_START_INT 中断的屏蔽中断位。(RO)

UHCI_RX_HUNG_INT_ST UHCI_RX_HUNG_INT_ENA 置 1 时 UHCI_RX_HUNG_INT 中断的屏蔽中断位。(RO)

UHCI_TX_HUNG_INT_ST UHCI_TX_HUNG_INT_ENA 置 1 时 UHCI_TX_HUNG_INT 中断的屏蔽中断位。(RO)

UHCI_SEND_S_REG_Q_INT_ST UHCI_SEND_S_REG_Q_INT_ENA 置 1 时 UHCI_SEND_S_REG_Q_INT 中断的屏蔽中断位。(RO)

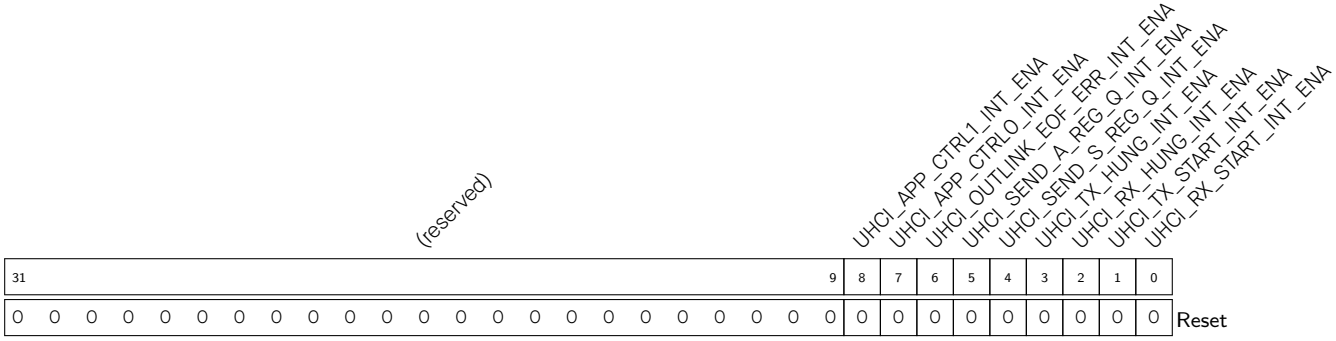
UHCI_SEND_A_REG_Q_INT_ST UHCI_SEND_A_REG_Q_INT_ENA 置 1 时 UHCI_SEND_A_REG_Q_INT 中断的屏蔽中断位。(RO)

UHCI_OUTLINK_EOF_ERR_INT_ST UHCI_OUTLINK_EOF_ERR_INT_ENA 置 1 时 UHCI_OUTLINK_EOF_ERR_INT 中断的屏蔽中断位。(RO)

UHCI_APP_CTRL0_INT_ST UHCI_APP_CTRL0_INT_ENA 置 1 时 UHCI_APP_CTRL0_INT 中断的屏蔽中断位。(RO)

UHCI_APP_CTRL1_INT_ST UHCI_APP_CTRL1_INT_ENA 置 1 时 UHCI_APP_CTRL1_INT 中断的屏蔽中断位。(RO)

Register 26.61. UHCI_INT_ENA_REG (0x000C)



UHCI_RX_START_INT_ENA UHCI_RX_START_INT 中断的使能位。(R/W)

UHCI_TX_START_INT_ENA UHCI_TX_START_INT 中断的使能位。(R/W)

UHCI_RX_HUNG_INT_ENA UHCI_RX_HUNG_INT 中断的使能位。(R/W)

UHCI_TX_HUNG_INT_ENA UHCI_TX_HUNG_INT 中断的使能位。(R/W)

UHCI_SEND_S_REG_Q_INT_ENA UHCI_SEND_S_REG_Q_INT 中断的使能位。(R/W)

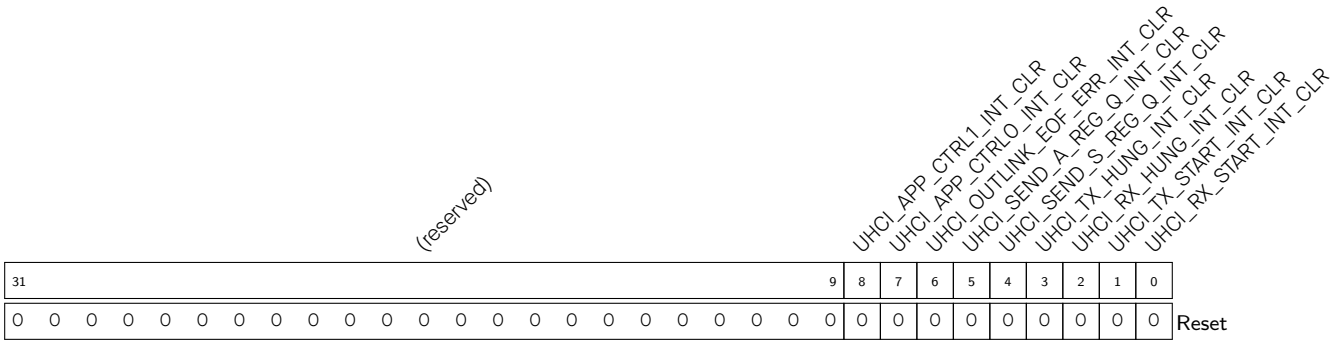
UHCI_SEND_A_REG_Q_INT_ENA UHCI_SEND_A_REG_Q_INT 中断的使能位。(R/W)

UHCI_OUTLINK_EOF_ERR_INT_ENA UHCI_OUTLINK_EOF_ERR_INT 中断的使能位。(R/W)

UHCI_APP_CTRL0_INT_ENA UHCI_APP_CTRL0_INT 中断的使能位。(R/W)

UHCI_APP_CTRL1_INT_ENA UHCI_APP_CTRL1_INT 中断的使能位。(R/W)

Register 26.62. UHCI_INT_CLR_REG (0x0010)



UHCI_RX_START_INT_CLR 置位此位清除 UHCI_RX_START_INT 中断。(WT)

UHCI_TX_START_INT_CLR 置位此位清除 UHCI_TX_START_INT 中断。(WT)

UHCI_RX_HUNG_INT_CLR 置位此位清除 UHCI_RX_HUNG_INT 中断。(WT)

UHCI_TX_HUNG_INT_CLR 置位此位清除 UHCI_TX_HUNG_INT 中断。(WT)

UHCI_SEND_S_REG_Q_INT_CLR 置位此位清除 UHCI_SEND_S_REQ_Q_INT 中断。(WT)

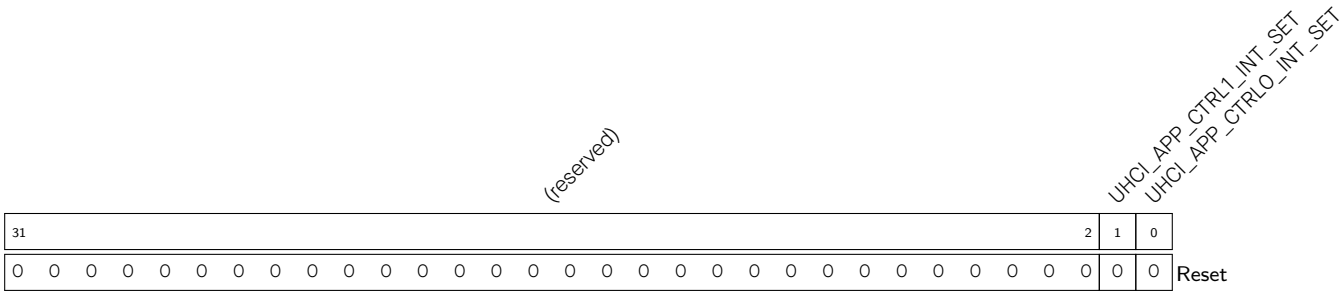
UHCI_SEND_A_REG_Q_INT_CLR 置位此位清除 UHCI_SEND_A_REQ_Q_INT 中断。(WT)

UHCI_OUTLINK_EOF_ERR_INT_CLR 置位此位清除 UHCI_OUTLINK_EOF_ERR_INT 中断。(WT)

UHCI_APP_CTRL0_INT_CLR 置位此位清除 UHCI_APP_CTRL0_INT 中断。(WT)

UHCI_APP_CTRL1_INT_CLR 置位此位清除 UHCI_APP_CTRL1_INT 中断。(WT)

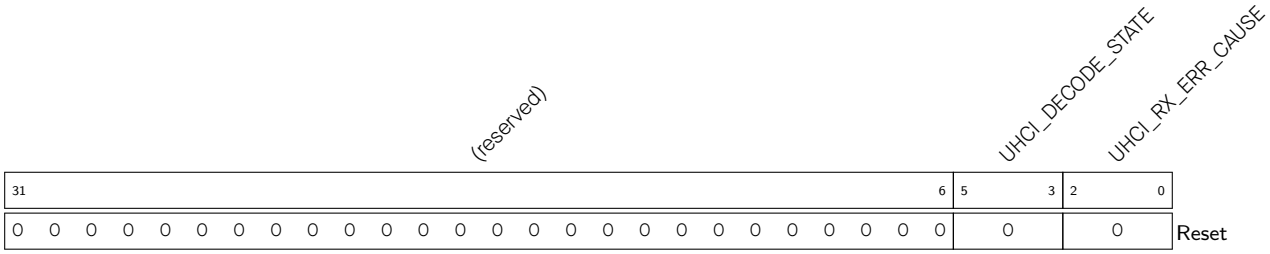
Register 26.63. UHCI_APP_INT_SET_REG (0x0014)



UHCI_APP_CTRL0_INT_SET UHCI_APP_CTRL0_INT 中断的软件触发源。(WT)

UHCI_APP_CTRL1_INT_SET UHCI_APP_CTRL1_INT 中断的软件触发源。(WT)

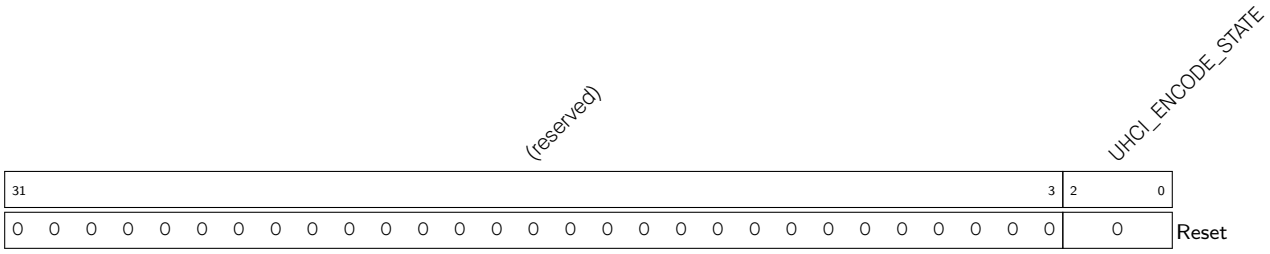
Register 26.64. UHCI_STATE0_REG (0x001C)



UHCI_RX_ERR_CAUSE 在 DMA 接收到错误帧时表示错误类型。3'b001：HCI 包校验和错误；3'b010：HCI 包序列号错误。3'b011：HCI 包 CRC 位错误；3'b100：找到 0xC0 但接收的 HCI 包不完整；3'b101：未找到 0xC0 但接收的 HCI 包完整；3'b110：CRC 检测错误。(RO)

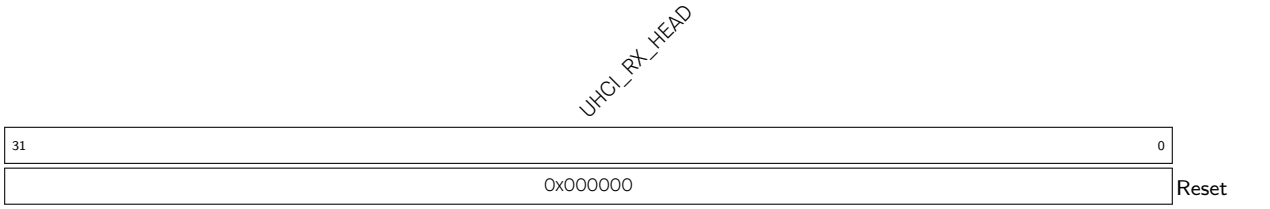
UHCI_DECODE_STATE UHCI 解码器状态。(RO)

Register 26.65. UHCI_STATE1_REG (0x0020)



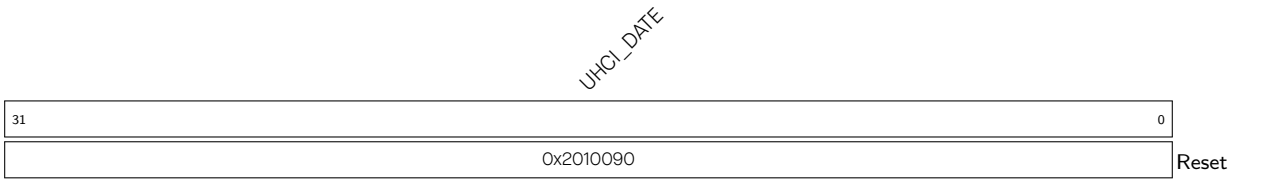
UHCI_ENCODE_STATE UHCI 编码器状态。(RO)

Register 26.66. UHCI_RX_HEAD_REG (0x0030)



UHCI_RX_HEAD 存储当前接收包的报头。(RO)

Register 26.67. UHCI_DATE_REG (0x0084)



UHCI_DATE 版本控制寄存器。(R/W)

第 27 章

I2C 控制器 (I2C)

I2C (Inter-Integrated Circuit) 总线用于使 ESP32-S3 和多个外部设备进行通信。多个外部设备可以共用一个 I2C 总线。

27.1 概述

I2C 是一个两线总线，由 SDA 线和 SCL 线构成。这些线设置为漏极开漏 (open-drain) 输出。因此，I2C 总线上可以挂载多个外设，通常是和一个或多个主机以及一个或多个从机。但同一时刻只有一个主机能占用总线访问一个从机。

主机发出开始信号，则通讯开始：在 SCL 为高电平时拉低 SDA 线，主机将通过 SCL 线发出 9 个时钟脉冲。前 8 个脉冲用于传输 7 位地址和 1 个读写位。如果从机地址与该 7 位地址一致，那么从机可以通过在第 9 个脉冲拉低 SDA 线来应答。接下来，根据读 / 写标志位，主机和从机之间可以传输更多的数据。根据应答位的逻辑电平决定是否停止发送数据。在数据传输中，SDA 线仅在 SCL 线为低电平时才发生变化。当主机完成通讯，发送一个停止标志：在 SCL 为高电平时，拉高 SDA 线。如果一次通信中主机既有写操作又有读操作，则主机需在读写操作变化前，发送一个重新开始信号、从机地址和读写标志位。重新开始信号不仅用于一次通信中切换方向，也用于切换设备模式（主机或从机模式）。

27.2 主要特性

I2C 具有以下几个特点：

- 支持主机模式和从机模式
- 支持多主机和从机通信
- 支持标准模式 (100 Kbit/s)
- 支持快速模式 (400 Kbit/s)
- 支持 7 位以及 10 位地址寻址
- 支持拉低 SCL 时钟实现连续数据传输
- 支持可编程数字噪声滤波功能
- 支持从机地址和从机内存或寄存器地址的双寻址模式

27.3 I2C 架构

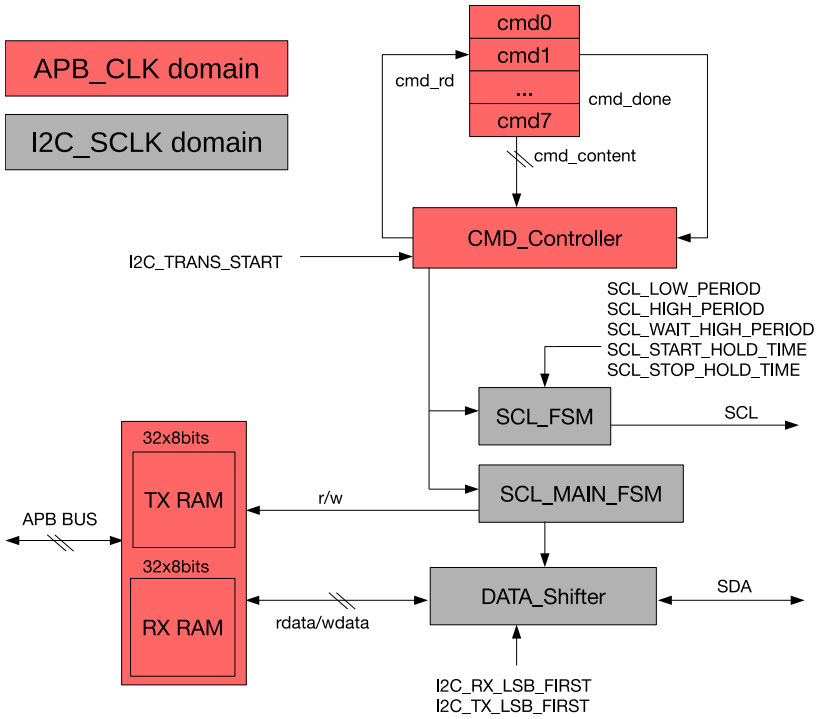


图 27.3-1. I2C 主机基本架构

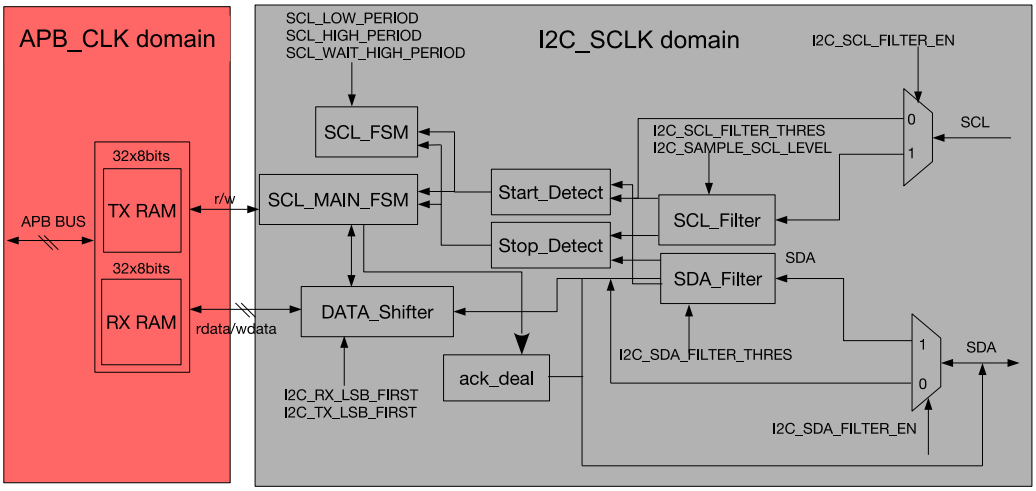


图 27.3-2. I2C 从机基本架构

I2C 控制器可以工作于主机模式或者从机模式，I2C_MS_MODE 寄存器用于模式选择。图 27.3-1 为 I2C 主机基本架构图，图 27.3-2 为 I2C 从机基本架构图。I2C 控制器内部包括的模块主要有：

- 接收和发送存储器 TX/RX RAM
- 命令控制器 CMD_Controller
- SCL 时钟控制器 SCL_FSM
- SDA 数据控制器 SCL_MAIN_FSM

- 串并转换器 DATA_Shifter
- SCL 滤波器 SCL_Filter
- SDA 滤波器 SDA_Filter

另外，还有产生 I2C 内部时钟的时钟模块，以及在 APB 总线和 I2C 模块之间同步的同步模块。

时钟模块的作用是进行时钟源选择，时钟开关和时钟分频。SCL_Filter 和 SDA_Filter 分别用于消除 SCL 及 SDA 输入信号上的噪声。同步模块用来同步不同时钟域之间信号的传输。

图 27.3-3 和图 27.3-4 是 I2C 协议的时序图和对应的参数表。SCL_FSM 用来产生满足 I2C 协议的 SCL 时钟。

SCL_MAIN_FSM 模块用来控制 I2C 指令的执行，和 SDA 线的序列。I2C 主机通过 CMD_Controller 产生 (R)START、STOP、WRITE、READ 和 END 指令。TX/RX RAM 分别用来存储 I2C 要发送和接收到的数据。DATA_Shifter 用来完成串行数据和并行数据之间的转换。

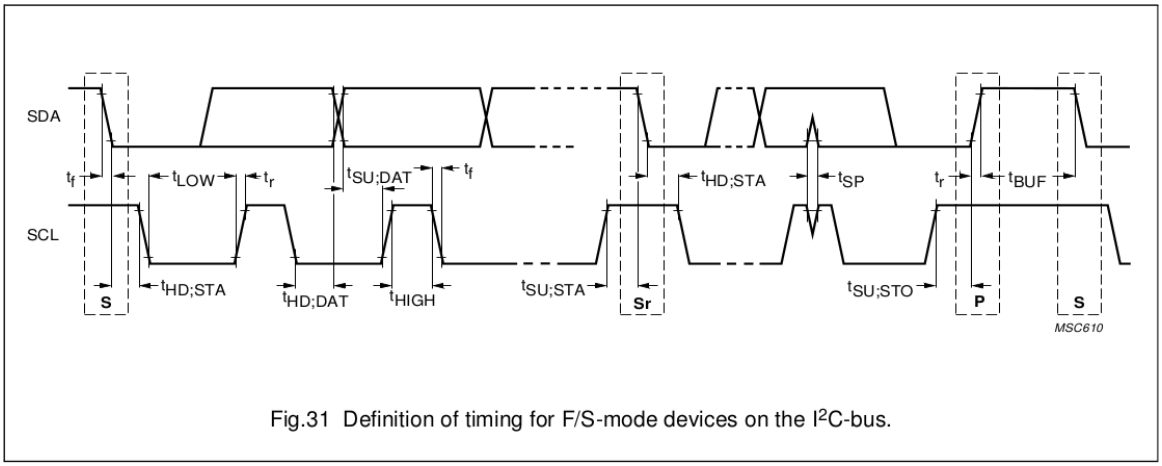


Fig.31 Definition of timing for F/S-mode devices on the I²C-bus.

图 27.3-3. I2C 协议时序（引自 The I2C-bus specification Version 2.1 Fig.31）

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: for CBUS compatible masters (see NOTE, Section 10.1.3) for I²C-bus devices	$t_{HD;DAT}$	 5.0 0(2)	 – 3.45(3)	 – 0(2)	 – 0.9(3)	 µs µs
Data set-up time	$t_{SU;DAT}$	250	–	100(4)	–	ns
Rise time of both SDA and SCL signals	t_r	–	1000	20 + 0.1C_b(5)	300	ns
Fall time of both SDA and SCL signals	t_f	–	300	20 + 0.1C_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

图 27.3-4. I2C 时序参数（引自 The I2C-bus specification Version 2.1 Table5）

27.4 功能描述

需要注意的是，I2C 总线上其他主机或者从机的操作可能与 ESP32-S3 I2C 外设有所不同，具体请参考各个 I2C 设备的技术规格书。

27.4.1 时钟配置

寄存器配置和 TX/RX RAM 部分的时钟域为 APB_CLK，时钟范围是 1 ~ 80 MHz。I2C 主要逻辑部分，包括 SCL_FSM、SCL_MAIN_FSM、SCL_FILTER、SDA_FILTER 和 DATA_SHIFTER 都为 I2C_SCLK 时钟域。

用户可以通过配置 I2C_SCLK_SEL 选择 I2C_SCLK 的时钟源：XTAL_CLK 或 RC_FAST_CLK，I2C_SCLK_SEL 为 0 时选择时钟源 XTAL_CLK，I2C_SCLK_SEL 为 1 时选择时钟源 RC_FAST_CLK。配置 I2C_SCLK_ACTIVE 为高电平来打开 I2C_SCLK 的时钟源。选择后的时钟经过小数分频得到 I2C 的工作时钟 I2C_SCLK，分频系数为：

$$I2C\_SCLK\_DIV\_NUM + 1 + \frac{I2C\_SCLK\_DIV\_A}{I2C\_SCLK\_DIV\_B}$$

XTAL_CLK 的频率是 40 MHz，RC_FAST_CLK 的频率是 17.5 MHz。根据时序参数的限制，分频后的 I2C_SCLK 的频率要满足大于 SCL 频率的 20 倍的关系。

27.4.2 滤除 SCL 和 SDA 噪声

SCL_Filter 和 SDA_Filter 滤波器模块实现方式相同，用于滤除 SCL 及 SDA 输入信号上的噪声。通过配置 I2C_SCL_FILTER_EN 以及 I2C_SDA_FILTER_EN 寄存器可以开启或关闭滤波器。

以 SCL_Filter 为例，当使能 SCL_Filter 功能时，滤波器会连续采样输入信号 SCL，如果输入信号在连续 I2C_SCL_FILTER_THRES 个 I2C_SCLK 时钟周期内保持不变，则输入信号有效，否则输入信号无效。只有有效的输入信号才能通过滤波器。因此，SCL_Filter 和 SDA_Filter 滤波器会过滤脉冲宽度小于 I2C_SCL_FILTER_THRES 以及 I2C_SDA_FILTER_THRES 个 I2C_SCLK 时钟周期的线路毛刺。

27.4.3 SCL 时钟拉伸

从机模式下，可以通过拉低 SCL 线，给软件足够的时间处理数据。置位 I2C_SLAVE_SCL_STRETCH_EN 位使能 SCL 时钟拉伸功能，配置 I2C_STRETCH_PROTECT_NUM 字段来控制 SCL 拉伸后释放的时长以防出现时序错误。出现以下四种情况时从机会拉低 SCL 线：

1. 地址命中：从机模式下，从机地址与主机发送到 SDA 线上的地址相匹配，且读写标志位为 1。
2. 写满：从机模式下，I2C 控制器的 RX RAM 为满。注意，从机在接收少于 32 个字节时，可以不开启时钟拉伸功能；当要接收不少于 32 个字节时，可以通过 FIFO 阈值中断写 RAM 的乒乓操作，或者开始时钟拉伸功能，给软件提供处理时间。开启时钟拉伸功能时，必须将 I2C_RX_FULL_ACK_LEVEL 置 0，来保证功能正确，否则可能会出现不可预计的后果。
3. 读空：从机模式下，I2C 控制器要发送数据，但 TX RAM 为空。
4. 发送 ACK 时：从机模式下置位 I2C_SLAVE_BYTE_ACK_CTL_EN，从机会在发送 ACK 时拉低 SCL。软件在此阶段进行一些操作，如数据校验，并通过配置 I2C_SLAVE_BYTE_ACK_LVL 控制将要发送的 ACK 的电平高低。要注意的是，当出现从机接收的 RX RAM 满时，要发送的 ACK 电平将由 I2C_RX_FULL_ACK_LEVEL 而不是 I2C_SLAVE_BYTE_ACK_LVL 决定。此时同样需要将 I2C_RX_FULL_ACK_LEVEL 置 0，以保证 SCL 时钟拉伸功能的正常产生。

SCL 线拉低后，软件可读取 I2C_STRETCH_CAUSE 位获取 SCL 时钟拉伸的原因。置位 I2C_SLAVE_SCL_STRETCH_CLR 位关闭 SCL 时钟拉伸。

27.4.4 SCL 空闲时产生 SCL 脉冲

通常情况下，在 I2C 总线空闲时，SCL 线一直为高。ESP32-S3 I2C 支持在 I2C 主机处于空闲状态时，可编程配置产生 SCL 脉冲的功能。这个功能仅在 I2C 控制器作为主机时有效。置位 I2C_SCL_RST_SLV_EN，硬件会发送 I2C_SCL_RST_SLV_NUM 个 SCL 脉冲。一段时间后，软件读取到 I2C_SCL_RST_SLV_EN 位的值为 0 后，再置位 I2C_CONF_UPGATE，来停止这个功能。

27.4.5 同步

I2C 的寄存器配置用 APB 时钟，I2C 主模块用 I2C_SCLK，这之间存在异步处理，需要增加同步的步骤将配置寄存器的值更新进入 I2C 主模块。步骤为先写配置寄存器，再向 I2C_CONF_UPGATE 位写 1。需要通过这种方法更新的配置寄存器详见表 27.4-1。

表 27.4-1. I2C 同步寄存器

配置寄存器	配置参数	地址
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	

27.4.6 漏级开路输出

SCL 及 SDA 线采用漏级开路的驱动方式。I2C 控制器有两种配置方式实现漏级开路驱动方式：

1. 置位 I2C_SCL_FORCE_OUT、I2C_SDA_FORCE_OUT 并配置相应 SCL 及 SDA PAD 的 GPIO_PIN*n*_PAD_DRIVER 寄存器为漏级开路驱动。
2. 清零 I2C_SCL_FORCE_OUT 以及 I2C_SDA_FORCE_OUT。

SCL 和 SDA 配置成开漏方式时，从低电平转向高电平的时间会较长，这个转变时间由线上的上拉电阻以及电容共同决定。开漏模式下，I2C 输出频率的占空比受限于 SCL 上拉速度，主要受 SCL 的速度限制。

另外，在 I2C_SCL_FORCE_OUT 和 I2C_SCL_PD_EN 置 1 时，可以强制拉低 SCL 线；在 I2C_SDA_FORCE_OUT 和 I2C_SDA_PD_EN 置 1 时，可以强制拉低 SDA 线。

27.4.7 时序参数配置

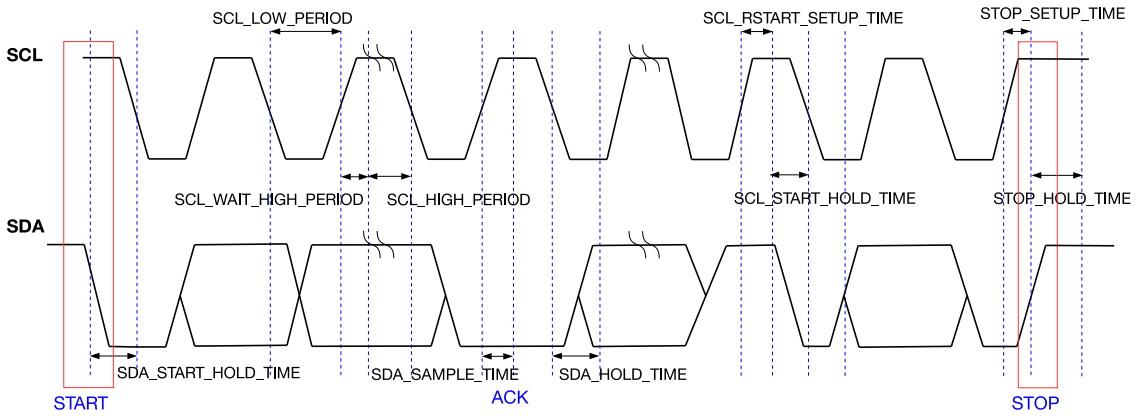


图 27.4-1. I2C 时序图

图 27.4-1 为实现 I2C 协议的 I2C 主机的时序图，图中的寄存器均用来配置时序参数。I2C 控制器的 START 位、STOP 位、数据保持时间、数据采样时间、SCL 上升沿等待时间等时序均可以通过图 27.4-1 中所示的寄存器进行配置。这些寄存器以模块时钟 (I2C_SCLK) 为单位，与各时序参数的对应关系为：

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. $t_{SU:DAT} = (I2C\_SCL\_LOW\_PERIOD - I2C\_SDA\_HOLD\_TIME) \cdot T_{I2C\_SCLK}$

根据在何种模式下有意义，下列时序寄存器可分为两组：

- 主机模式：
  1. I2C_SCL_START_HOLD_TIME：生成 I2C 协议中的 start 位时，SDA 信号拉低到 SCL 信号拉低的时间间隔。该时间间隔为（I2C_SCL_START_HOLD_TIME +1）个模块时钟周期。仅控制器工作在主机模式时有意义。
  2. I2C_SCL_LOW_PERIOD：SCL 低电平持续时间。SCL 低电平时间为 (I2C_SCL_LOW_PERIOD + 1) 个模块时钟周期。但是，如果外设拉低 SCL ，I2C 控制器执行 END 命令拉低 SCL，或者控制器发生 SCL 时钟拉伸则可能会导致 SCL 低电平时间变长。仅控制器工作在主机模式时有意义。
  3. I2C_SCL_WAIT_HIGH_PERIOD：等待 SCL 线拉高的模块时钟周期数。请确保在该时间内 SCL 线可以完成上拉。否则会导致 SCL 高电平持续时间不可预测。仅控制器工作在主机模式时有意义。
  4. I2C_SCL_HIGH_PERIOD：SCL 线拉高后维持高电平的模块时钟周期数。仅控制器工作在主机模式时有意义。当 SCL 线在 I2C_SCL_WAIT_HIGH_PERIOD + 1 个模块时钟内完成上拉，则 SCL 线的频率为：

$$f_{scl} = \frac{f_{\text{I2C\_SCLK}}}{\text{I2C\_SCL\_LOW\_PERIOD} + \text{I2C\_SCL\_HIGH\_PERIOD} + \text{I2C\_SCL\_WAIT\_HIGH\_PERIOD} + 3}$$

- 主机模式和从机模式：
  1. I2C_SDA_SAMPLE_TIME：SCL 上升沿到采样 SDA 线电平值的时间间隔。推荐设置在 SCL 高电平持续时间的中间值，以保证能够正确采样到 SDA 线上电平。控制器工作在主机模式及从机模式时都有意义。
  2. I2C_SDA_HOLD_TIME：SDA 输出数据变化与 SCL 下降沿的时间间隔。控制器工作在主机模式及从机模式时都有意义。

根据时序参数的限制，对时序寄存器的配置范围也有约束。

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

27.4.8 超时控制

I2C 内部有三种超时控制，分别是对 SCL_FSM 状态的超时控制、SCL_MAIN_FSM 状态的超时控制和对 SCL 线状态的超时控制。其中前两种是一直打开的，第三种是可编程配置的。

当 SCL_FSM 长时间处于同一状态不变，且时间超过 $2^{I2C\_SCL\_ST\_TO\_I2C}$ 个时钟周期后，会触发 I2C_SCL_ST_TO_INT 中断，状态机会回到空闲状态。I2C_SCL_ST_TO_I2C 的配置值最大为 22，即最大在时间超过 2^{22} 个 I2C_SCLK 时钟周期后会产生超时中断。

当 SCL_MAIN_FSM 长时间处于同一状态不变，且时间超过 $2^{I2C\_SCL\_MAIN\_ST\_TO\_I2C}$ 个 I2C_SCLK 时钟周期后，会触发 I2C_SCL_MAIN_ST_TO_INT 中断，状态机会回到空闲状态。I2C_SCL_MAIN_ST_TO_I2C 的配置值最大为 22，即最大在时间超过 2^{22} 个模块时钟后会产生超时中断。

使能 I2C_TIME_OUT_EN 打开 SCL 线状态的超时控制。当 SCL 线状态长时间维持同一电平不变，且时间超过 $2^{I2C\_TIME\_OUT\_VALUE}$ 后，会触发 I2C_TIME_OUT_INT 中断，I2C 总线回到空闲状态。

27.4.9 指令配置

I2C 控制器工作于主机模式时，CMD_Controller 会依次从八个命令寄存器中读出命令并按照命令来控制 SCL_FSM 及 SCL_MAIN_FSM。

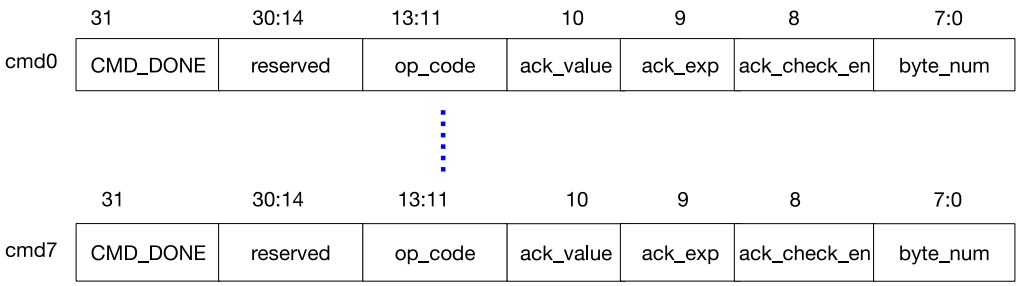


图 27.4-2. I2C 命令寄存器结构

命令寄存器只在 I2C 控制器工作于主机模式时才有效，其内部结构如图 27.4-2 所示。命令寄存器的参数为:

1. CMD_DONE: 命令执行完成标识。每条命令执行完硬件会将对应命令寄存器中的 CMD_DONE 置 1。软件可以通过读取每条命令的 CMD_DONE 位来判断该命令是否执行完毕。每次更新命令时，软件需要将 CMD_DONE 位清零。
2. op_code: 命令编码，共有五种命令:
   - RSTART: op_code 等于 6 时为 RSTART 命令，该命令指示 I2C 控制器发送 I2C 协议中的 START 位或 RESTART 位。
   - WRITE: op_code 等于 1 时为 WRITE 命令，该命令指示 I2C 控制器向从机发送从机地址、被访问的寄存器地址（仅限双寻址模式）、数据。
   - READ: op_code 等于 3 时为 READ 命令，该命令指示 I2C 控制器从从机读取数据。
   - STOP: op_code 等于 2 时为 STOP 命令，该命令指示 I2C 控制器发送 I2C 协议中的 STOP 位。此条命令也标识本次命令序列执行完成，CMD_Controller 将会停止取指令。软件再次启动

CMD_Controller 后，会重新从命令寄存器 0 开始去取指令。

- END: op_code 等于 4 时为 END 命令，该命令指示 I2C 控制器将 SCL 信号拉低，暂停 I2C 通信。该命令也标识本次命令序列执行完成，CMD_Controller 将会停止取指令。软件在更新命令寄存器和 RAM 数据后可重新启动 CMD_Controller，继续进行 I2C 协议传输。再次启动后 CMD_Controller 会重新从命令寄存器 0 开始去取指令。

3. ack_value: 该位设置读操作时 I2C 控制器在 I2C 协议中的 ACK 位发送的电平值。RSTART、STOP、END、WRITE 命令中该位没有意义。
4. ack_exp: 该位用于设置写操作时 I2C 控制器在 I2C 协议中的 ACK 位期望接收的电平值。RSTART、STOP、END、READ 命令中该位没有意义。
5. ack_check_en: 该位使能写操作中 I2C 控制器检查从机发送的 ACK 位电平与命令中的 ack_exp 是否一致。如果接收的 ACK 值与 WRITE 命令中的 ack_exp 电平不一致时，I2C 主机会产生 I2C_NACK_INT 中断，停止发送数据并产生 STOP。此位为 1 时，检测从机发送的 ACK 位电平；此位为 0 时，不检测从机发送的 ACK 位电平。RSTART、STOP、END、READ 命令中该位没有意义。
6. byte_num: 读写数据的长度（单位字节），最大为 255，最小为 1。RSTART、STOP、END 命令中 byte_num 无意义。

每次命令序列的执行都是从命令寄存器 0 开始，到 STOP 或 END 命令结束。所以需要保证每个命令序列中必须有 STOP 或 END 命令。

一次完整的 I2C 协议传输应该起始于 START 命令，结束于 STOP 命令。可通过 END 命令将一次 I2C 协议传输分为多个命令序列来完成。每个命令序列可以改变数据传输的方向、时钟频率、从机地址和数据长度等。这样可以弥补 RAM 大小不足的问题，也可以实现更灵活的 I2C 通信。

27.4.10 TX/RX RAM 数据存储

TX/RX RAM 大小均为 32 x 8 位。TX RAM 和 RX RAM 均可以通过 FIFO 和直接地址 (non-FIFO) 两种方式访问。将 I2C_NONFIFO_EN 位设置成 0，为 FIFO 方式；I2C_NONFIFO_EN 位设置成 1 时为直接地址方式。

TX RAM 用于存储 I2C 控制器需要发送的数据。在 I2C 通信的过程中，当 I2C 控制器需要发送数据时（不包括 ACK 位响应），会依次读出 TX RAM 中的数据并串行输出到 SDA 线上。当 I2C 控制器工作于主机模式时，所有需要发送给从机的数据都必须按照发送顺序依次存储在 TX RAM 中。包括被访问的从机地址、读写标志位、被访问的寄存器地址（仅限双地址寻址模式下）、写数据。当 I2C 控制器工作于从机模式时，TX RAM 中只存放写数据。

TX RAM 可被 CPU 读写。CPU 可通过两种方式写 TX RAM: FIFO 访问和直接地址访问。FIFO 访问方式是通过固定地址 I2C_DATA_REG 写 TX RAM，硬件自动进行 TX RAM 写地址自增。直接地址访问是通过地址段 (I2C 基地址 + 0x100) ~(I2C 基地址 + 0x17C) 直接访问 TX RAM。TX RAM 的每一个字节占据一个字 (word) 的地址。因此，第一个字节访问地址为 I2C 基地址 + 0x100，第二字节访问地址为 I2C 基地址 + 0x104，第三字节访问地址为 I2C 基地址 + 0x108，以此类推。CPU 只可通过直接地址访问方式读 TX RAM，读 TX RAM 的地址和写 TX RAM 的地址相同。

RX RAM 存储的是 I2C 通信过程中，I2C 控制器接收到的数据。当 I2C 控制器工作于从机模式时，主机发送的从机地址及被访问的寄存器地址（仅限双地址寻址模式下）都不会存储在 RX RAM 中。软件可以在 I2C 通信结束后，读出 RX RAM 的值。

RX RAM 只可被 CPU 读。CPU 可通过两种方式读 RX RAM: FIFO 访问和直接地址访问。FIFO 访问方式是通过固定地址 I2C_DATA_REG 读 RX RAM，硬件自动完成 RX RAM 读地址自增。直接地址访问是通过地址段 (I2C 基地址 + 0x180) ~(I2C 基地址 + 0x1FC) 直接访问 RX RAM。RX RAM 的每一个字节占据一个字的地址。因此，第一

个字节访问地址为 I2C 基地址 + 0x180，第二字节访问地址为 I2C 基地址 + 0x184，第三字节访问地址为 I2C 基地址 + 0x188，以此类推。

在 FIFO 模式下可以对 TX RAM 进行乒乓操作，来实现发送大于 32 个字节的数据。置位 I2C_FIFO_PRT_EN，当 RAM 中剩下的待发送数据字节数小于 I2C_TXFIFO_WM_THRHD 时，会产生 I2C_TXFIFO_WM_INT 中断。软件收到中断后，继续向 I2C_DATA_REG（主机）中写数。需要保证向 TX RAM 写数或更新数据的操作提前于 I2C 发送数据的动作，否则会产生不可预计的后果。

在 FIFO 模式下也可以对 RX RAM 进行乒乓操作，来实现接收大于 32 个字节的数据。置位 I2C_FIFO_PRT_EN，将 I2C_RX_FULL_ACK_LEVEL 置 0。当 RAM 中收到的数据字节数大于等于 I2C_RXFIFO_WM_THRHD（从机）时，会产生 I2C_RXFIFO_WM_INT 中断。软件收到中断后，从 I2C_DATA_REG（从机）中读数。

27.4.11 数据转换

DATA_Shifter 模块用于串并转换，当 I2C 发送数据时，将 TX RAM 中的字节数据转化成比特流；当 I2C 接收数据时，将比特流转化成字节数据并存入 RX RAM。I2C_RX_LSB_FIRST 和 I2C_TX_LSB_FIRST 用于配置最高有效位或最低有效位的优先储存或传输。

27.4.12 寻址模式

除了 7 位寻址模式，ESP32-S3 I2C 还支持 10 位寻址模式和双寻址模式。10 位寻址和 7 位寻址可结合使用。

假设从机地址为 SLV_ADDR 。ESP32-S3 I2C 控制器可以使用 7 位寻址（SLV_ADDR[6:0]），也可以使用 10 位寻址（SLV_ADDR[9:0]）。

对于主机模式而言，7 位寻址下只要发送一个字节地址段 SLV_ADDR[6:0] 和读写标志。7 位寻址模式下，有种特殊情况是广播寻址。在从机中，将 I2C_ADDR_BROADCASTING_EN 置 1，开启广播寻址模式。当接收到主机发送的地址为广播地址 (0x00) 且读写标志位为 0 时，此时无论从机自己的地址是多少，都会响应主机。

10 位寻址需要发送两字节地址段。第一个要发送的数是从机地址的第一个 7 位 slave_addr_first_7bits 和读写标志，slave_addr_first_7bits 的值应该配置为 (0x78 | SLV_ADDR[9:8])。第二个要发送的数是 slave_addr_second_byte，slave_addr_second_byte 的值为 SLV_ADDR[7:0]。

在从机中，可以通过配置 I2C_ADDR_10BIT_EN 寄存器开启 10 位寻址模式。I2C_SLAVE_ADDR 用于配置 I2C 从机地址。I2C_SLAVE_ADDR[14:7] 的值应配置为 SLV_ADDR[7:0]，I2C_SLAVE_ADDR[6:0] 的值应配置为 (0x78 | SLV_ADDR[9:8]) 。由于 10 位从机地址比 7 位地址多一个字节，所以 WRITE 命令对应的 byte_num 以及 RAM 中数据数量都相应增加 1。

控制器处于从机模式时，还支持双地址访问方式。双地址的第一个地址是 I2C 从机地址，第二个地址是 I2C 从机的内存地址。双地址模式下，RAM 必须采用 non-FIFO 方式访问。通过置位 I2C_FIFO_ADDR_CFG_EN 来使能双地址访问功能。

27.4.13 10 位寻址的读写标志位检查

在 10 位寻址模式下，将 I2C_ADDR_10BIT_RW_CHECK_EN 置 1，会对发送的第一个数 slave_addr_first_7bits 和读写标志做检查。当读写标志不是写，此时与协议不符合，会结束传输。若不打开此功能，当读写标志不是写，还能支持继续传输，但可能出现传输错误。

27.4.14 启动控制器

对于主机，配置成主机模式和命令寄存器等相关配置后，通过向 I2C_TRANS_START 写 1，启动主机解析，执行命令序列。一组命令总是从命令寄存器 0 开始，顺序执行到 STOP 或者 END 命令。当另一组命令需要从命令寄存器 0 开始执行时，需要重新向 I2C_TRANS_START 写 1 来更新命令。

对于从机，有两种启动方式：

- **自动启动传输：** 置位 I2C_SLV_TX_AUTO_START_EN，则从机会在被主机寻址后自动启动传输。此模式适用于主机需要持续访问从机数据的场景。
- **手动启动传输：** 清零 I2C_SLV_TX_AUTO_START_EN，且在每次接收传输前必须显式置位 I2C_TRANS_START。此模式适用于主机需要选择性读取从机数据的场景。

  例如：

  1. 从机准备 5 字节数据；
  2. 主机仅读取前 3 字节；
  3. 主机读取完成后，从机需置位 I2C_TX_FIFO_RST 清除 TXFIFO 中的剩余数据，随后置位 I2C_TRANS_START 重新初始化从机状态。

注意： 在手动启动模式下，必须在下次地址匹配前置位 I2C_TRANS_START。若主机在 I2C_TRANS_START 未置位时向从机发起寻址，从机可能会应答该地址，但状态机无法正常运行，从而导致从机控制器出现错误行为或时序紊乱。

27.5 编程示例

本节提供一些典型通信场景的编程示例。ESP32-S3 中有两个 I2C 外设控制器，为了便于描述，下文所有图示中的 I2C 主机和从机都假定为 ESP32-S3 I2C 外设控制器。

27.5.1 I2C 主机写入从机，7 位寻址，单次命令序列

27.5.1.1 场景介绍

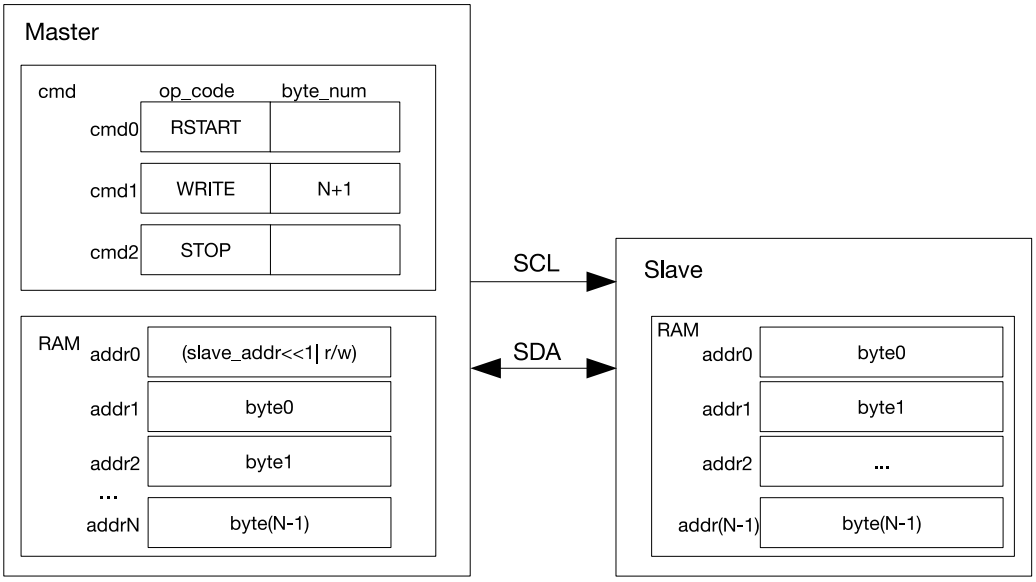


图 27.5-1. I2C 主机写 7 位寻址的从机

图 27.5-1 为 I2C 主机采用 7 位寻址写 N 个字节数据到 I2C 从机的寄存器或 RAM。如图 27.5-1 所示，主机 RAM 中第一个字节数据为 7 位从机地址 + 1 位读写标志位，其中读写标志位为 0 时表示写操作，接下来的连续空间存储待发送的数据。cmd 框中包含了相应的命令序列。

对于主机，在软件配置好命令序列以及 RAM 数据后，置位 I2C_TRANS_START 寄存器启动控制器进行数据传输。控制器的行为可分为四步：

1. 等待 SCL 线为高电平，以避免 SCL 线被其他主机或者从机占用。
2. 执行 RSTART 命令发送 START 位。
3. 执行 WRITE 命令从 RAM 的首地址开始取出 N+1 个字节并依次发送给从机，其中第一个字节为地址。
4. 发送 STOP。当 I2C 主机完成 STOP 位的传输后，会产生 I2C_TRANS_COMPLETE_INT 中断。

27.5.1.2 配置示例

1. 参照章节 27.4.7，配置 I2C 主机和 I2C 从机的时序参数寄存器。
2. 设置 I2C_MS_MODE（主机）为 1，I2C_MS_MODE（从机）为 0。
3. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。

4. 配置 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	RSTART	—	—	—	—
I2C_COMMAND1（主机）	WRITE	ack_value	ack_exp	1	N+1
I2C_COMMAND2（主机）	STOP	—	—	—	—

5. 参考章节 27.4.10，向 I2C 主机的 TX RAM 写入从机地址和要发送的数据。可选 FIFO 方式和直接访问方式。
6. 在 I2C_SLAVE_ADDR_REG（从机）的 I2C_SLAVE_ADDR（从机）设置 I2C 从机的地址。
7. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
8. 向 I2C_TRANS_START（主机）和 I2C_TRANS_START（从机）位写 1 开始传输。
9. I2C 从机比较 I2C 主机发送的从机地址和自己的 I2C_SLAVE_ADDR（从机），当 I2C 主机 WRITE 命令中的 ack_check_en（主机）配置为 1 时，I2C 主机会在发送完每个字节之后进行 ACK 检测。若 ack_check_en 配置为 0，则不会对 ACK 检测，会默认为匹配。
   - 匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平一致，传输继续。
   - 不匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平不一致，I2C 主机产生 I2C_NACK_INT（主机）中断，停止发送数据并且产生 STOP。
10. I2C 主机发送数据，并根据 ack_check_en（主机）配置的不同进行或不进行 ACK 检测。
11. 若发送数据 N 大于 32 字节，在 FIFO 模式下可以对 I2C 主机的 TX RAM 进行乒乓操作，具体做法参照章节 27.4.10。
12. 若接收数据 N 大于 32 字节，在 FIFO 模式下可以对 I2C 从机的 RX RAM 进行乒乓操作，具体做法参照章节 27.4.10。

    若接收数据 N 大于 32 字节，另一种处理方式是置位 I2C_SLAVE_SCL_STRETCH_EN（从机）使能 SCL 时钟拉伸，同时将 I2C_RX_FULL_ACK_LEVEL 置 0。RX RAM 满时会产生 I2C_SLAVE_STRETCH_INT（从机）中断。此时 I2C 从机会将 SCL 拉低，软件在此期间可以读取数据。等完成操作后再将 I2C_SLAVE_STRETCH_INT_CLR（从机）置 1 清除中断，并将 I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1 释放 SCL 总线。
13. 当整个传输正常结束，I2C 主机执行 STOP 命令，并产生 I2C_TRANS_COMPLETE_INT（主机）中断。

27.5.2 I2C 主机写入从机，10 位寻址，单次命令序列

27.5.2.1 场景介绍

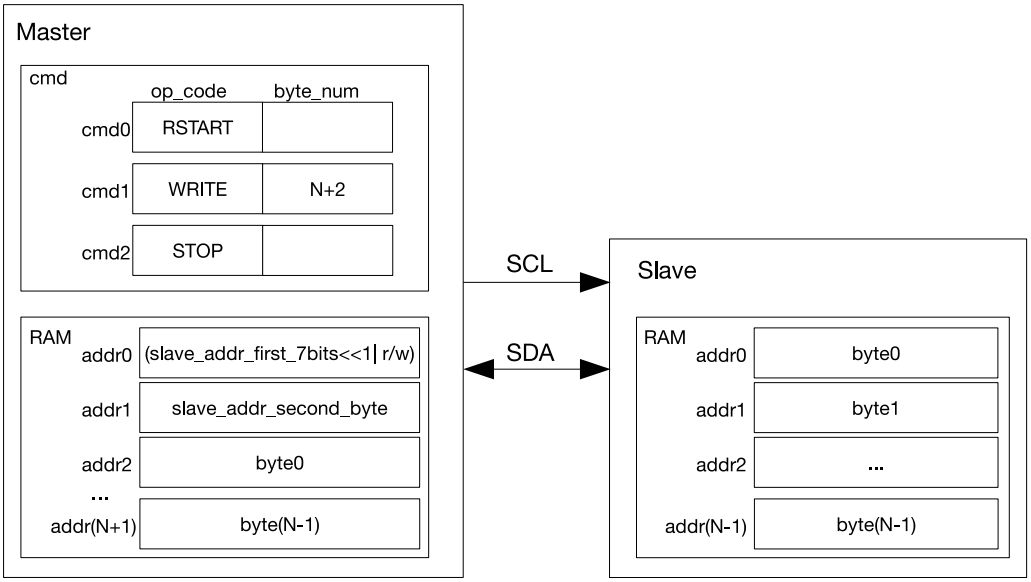


图 27.5-2. I2C 主机写 10 位寻址的从机

图 27.5-2 为 I2C 主机写 N 个字节到 10 位地址 I2C 从机的配置图。整个配置和传输过程都和 27.5.1 中类似，除了在传输的开始主机在 10 位寻址模式下需要发送两字节地址段。由于 10 位从机地址比 7 位地址多一个字节，所以 WRITE 命令对应的 byte_num 以及 TX RAM 中数据数量都相应增加 1。

27.5.2.2 配置示例

1. 设置 I2C_MS_MODE（主机）为 1，I2C_MS_MODE（从机）为 0。
2. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
3. 配置 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	RSTART	—	—	—	—
I2C_COMMAND1（主机）	WRITE	ack_value	ack_exp	1	N+2
I2C_COMMAND2（主机）	STOP	—	—	—	—

4. 在 I2C_SLAVE_ADDR_REG（从机）的 I2C_SLAVE_ADDR（从机）设置 I2C 从机的 10 位从机地址，并将 I2C_ADDR_10BIT_EN（从机）置 1 使能 10 位寻址模式。
5. 向 I2C 主机的 TX RAM 写入从机地址和要发送的数据，第一个地址字节是 ((0x78 | I2C_SLAVE_ADDR[9:8])«1)|R/W，第二个地址字节是 I2C_SLAVE_ADDR[7:0]。之后就是要发送的数据，可选 FIFO 方式和直接访问方式。
6. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
7. 向 I2C_TRANS_START（主机）和 I2C_TRANS_START（从机）位写 1 开始传输。
8. I2C 从机比较 I2C 主机发送的从机地址和自己的 I2C_SLAVE_ADDR（从机），当 I2C 主机 WRITE 命令中的 ack_check_en（主机）配置为 1 时，I2C 主机会在发送完每个字节之后进行 ACK 检测。若 ack_check_en 配置为 0，则不会对 ACK 检测，会默认为匹配。

- 匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平一致，传输继续。
- 不匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平不一致，I2C 主机产生 I2C_NACK_INT（主机）中断，停止发送数据并且产生 STOP。

9. I2C 主机发送数据，并根据 ack_check_en（主机）配置的不同进行或不进行 ACK 检测。
10. 若发送数据 N 大于 32 字节，在 FIFO 模式下可以对 I2C 主机的 TX RAM 进行乒乓操作，具体做法参照章节 27.4.10。
11. 若接收数据 N 大于 32 字节，在 FIFO 模式下可以对 I2C 从机的 RX RAM 进行乒乓操作，具体做法参照章节 27.4.10。

    若接收数据 N 大于 32 字节，另一种处理方式是置位 I2C_SLAVE_SCL_STRETCH_EN（从机）使能 SCL 时钟拉伸，同时将 I2C_RX_FULL_ACK_LEVEL 置 0。RX RAM 满时会产生 I2C_SLAVE_STRETCH_INT（从机）中断。此时 I2C 从机会将 SCL 拉低，软件在此期间可以读取数据。等完成操作后再将 I2C_SLAVE_STRETCH_INT_CLR（从机）置 1 清除中断，并将I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1 释放 SCL 总线。
12. 当整个传输正常结束，I2C 主机执行 STOP 命令，并产生 I2C_TRANS_COMPLETE_INT（主机）中断。

27.5.3 I2C 主机写入从机，7 位双地址寻址，单次命令序列

27.5.3.1 场景介绍

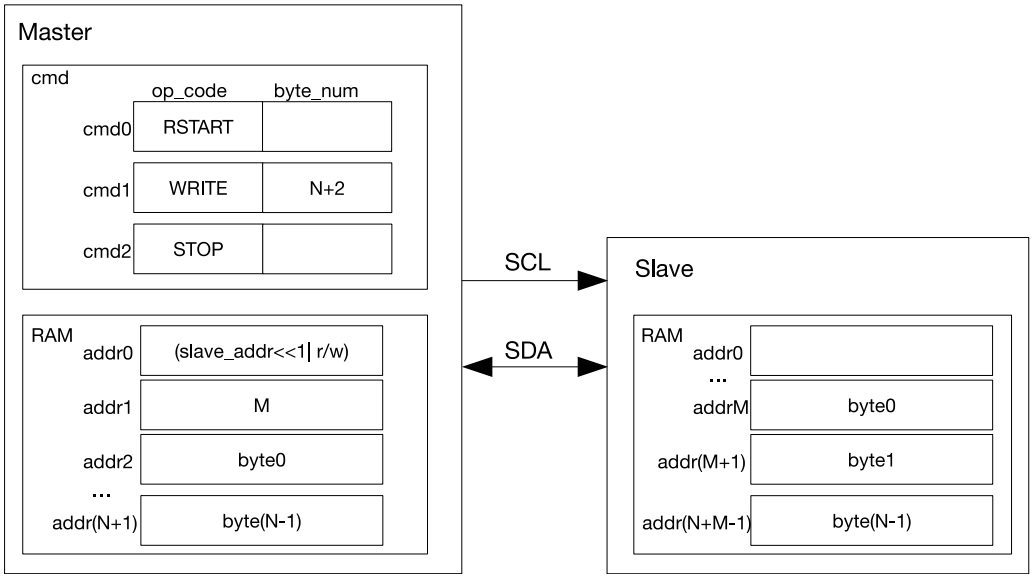


图 27.5-3. I2C 主机写 7 位双地址寻址从机

图 27.5-3 为 I2C 主机采用 7 位双寻址模式写 N 个字节数据到 I2C 从机的寄存器或 RAM 的值。整个配置和传输过程都和章节 27.5.1 中类似，区别是传输的开始主机在 7 位双寻址模式下需要发送两个字节。双地址的第一个地址是 7 位从机地址，第二个地址是 I2C 从机的内存地址（即图 27.5-3 中的 addrM）。双地址模式下，RX RAM 必须采用 non-FIFO 方式访问。另一个区别是，I2C 从机将接收到的数据 byte0 ~ byte (N-1) 从 RX RAM 中的 addrM 开始依次存储，addrM 就是主机发送的 I2C 内存地址。当超出地址 31 后会从地址 0 开始继续存储。

27.5.3.2 配置示例

1. 设置 I2C_MS_MODE（主机）为 1，I2C_MS_MODE（从机）为 0。
2. 设置 I2C_FIFO_ADDR_CFG_EN（从机）为 1 来使能双寻址模式。
3. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
4. 配置 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	RSTART	—	—	—	—
I2C_COMMAND1（主机）	WRITE	ack_value	ack_exp	1	N+2
I2C_COMMAND2（主机）	STOP	—	—	—	—

5. 向 I2C 主机的 TX RAM 写入从机地址和要发送的数据，可以用 FIFO 方式或直接访问方式。
6. 在 I2C_SLAVE_ADDR_REG（从机）的 I2C_SLAVE_ADDR（从机）设置 I2C 从机的地址。
7. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
8. 向 I2C_TRANS_START（主机）和 I2C_TRANS_START（从机）位写 1 开始传输。
9. I2C 从机比较 I2C 主机发送的从机地址和自己的 I2C_SLAVE_ADDR（从机），当 I2C 主机 WRITE 命令中的 ack_check_en（主机）配置为 1 时，I2C 主机会在发送完每个字节之后进行 ACK 检测。若 ack_check_en 配置为 0，则不会对 ACK 检测，会默认为匹配。
   - 匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平一致，传输继续。
   - 不匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平不一致，I2C 主机产生 I2C_NACK_INT（主机）中断，停止发送数据并且产生 STOP。
10. I2C 从机接收到 I2C 主机发送的内存地址，完成 RX RAM 的地址偏移。
11. I2C 主机发送数据，并根据 ack_check_en（主机）配置的不同进行或不进行 ACK 检测。
12. 若发送数据 N 大于 32 字节，在 FIFO 模式下可以对 I2C 主机的 TX RAM 进行乒乓操作，具体做法参照章节 27.4.10。
13. 若接收数据 N 大于 32 字节，置位 I2C_SLAVE_SCL_STRETCH_EN（从机）使能 SCL 时钟拉伸，同时将 I2C_RX_FULL_ACK_LEVEL 置 0。RX RAM 满时会产生 I2C_SLAVE_STRETCH_INT（从机）中断。此时 I2C 从机会将 SCL 拉低，软件在此期间可以读取数据。等完成操作后再将 I2C_SLAVE_STRETCH_INT_CLR（从机）置 1 清除中断，并将 I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1 释放 SCL 总线。
14. 当整个传输正常结束，I2C 主机执行 STOP 命令，并产生 I2C_TRANS_COMPLETE_INT（主机）中断。

27.5.4 I2C 主机写入从机，7 位寻址，多次命令序列

27.5.4.1 场景介绍

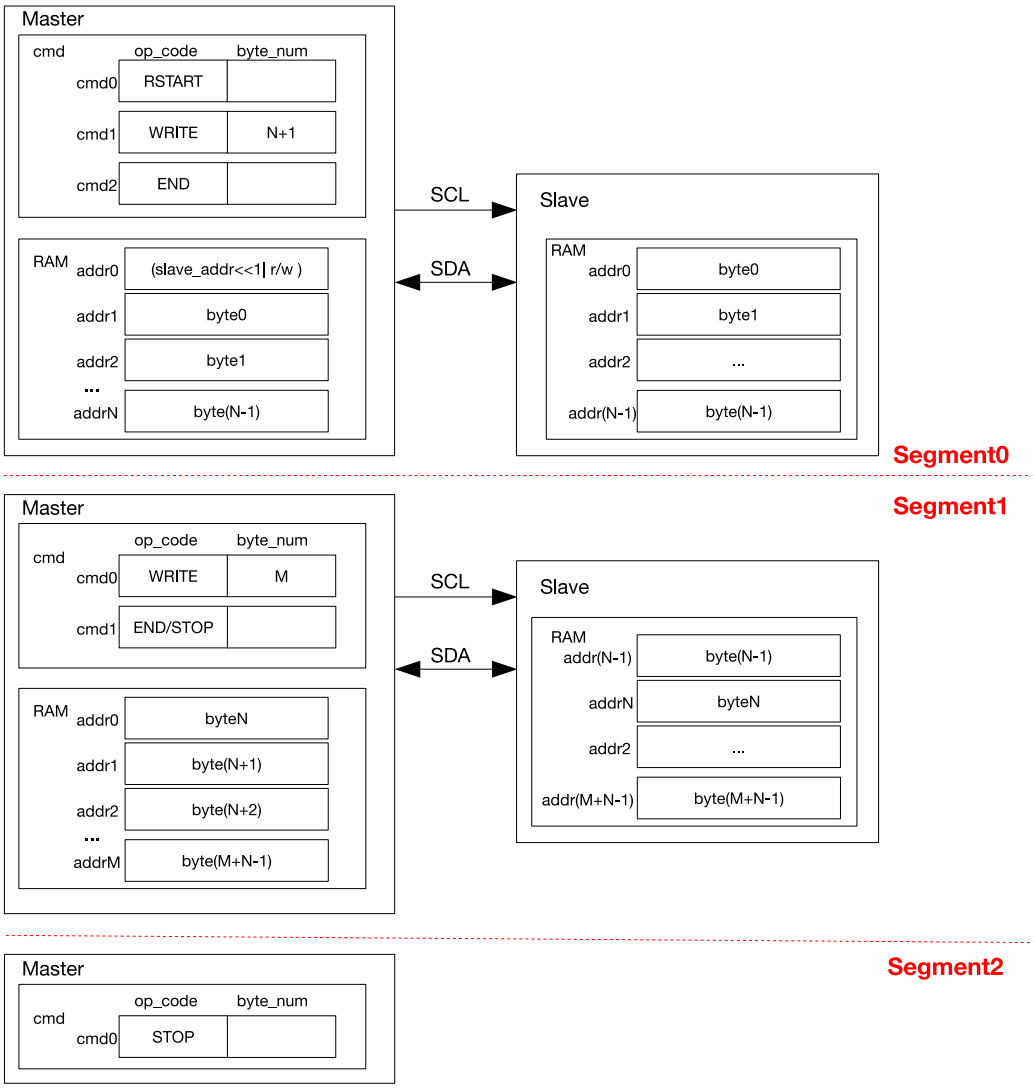


图 27.5-4. I2C 主机分段写 7 位寻址的从机

RAM 的大小只有 32 字节，对于大量的数据传输当 RAM 乒乓操作也不能满足要求时，建议使用多次命令序列进行分段传输。每次命令序列以 END 命令结尾，这样控制器会执行 END 命令拉低 SCL 线，软件此时可以更新命令序列寄存器和 RAM 的内容以用于下一次命令序列的传输。

以两段和三段传输为例，如图 27.5-4 所示为 I2C 主机分成三段或者两段写从机。配置 I2C 主机的命令序列如第一段所示，并且在主机的 RAM 中准备好数据，置位 I2C_TRANS_START，I2C 主机即开始数据传输。在执行到 END 命令后，I2C 主机会关闭 SCL 时钟，并将 SCL 线拉低来防止其他设备占用 I2C 总线。此时控制器产生 I2C_END_DETECT_INT 中断。

在检测到 I2C_END_DETECT_INT 中断后，软件可以更新命令序列以及 RAM 中的内容如第二段所示，并清除 I2C_END_DETECT_INT 中断。当第二段中 cmd1 为 STOP 时，不需要第三段，即为两段写从机。置位 I2C_TRANS_START 后，I2C 主机继续发送数据，并在最后发送 STOP 位。当为三段写从机时，I2C 主机在第二段发送完数据，并检测到 I2C 主机的 I2C_END_DETECT_INT 中断后，即可配置 cmd 如第三段所示。置位 I2C_TRANS_START 后，I2C 主机即产生 STOP 位，从而停止传输。

请注意，在两个分段之间，I2C 总线上的其他主机设备不会占用总线。只有在发送了 STOP 信号后总线才会被释放。任何情况下，置位 I2C_FSM_RST 可复位 I2C 控制器，硬件自清 I2C_FSM_RST。

27.5.4.2 配置示例

1. 设置 I2C_MS_MODE（主机）为 1，I2C_MS_MODE（从机）为 0。
2. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
3. 配置 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	RSTART	—	—	—	—
I2C_COMMAND1（主机）	WRITE	ack_value	ack_exp	1	N+1
I2C_COMMAND2（主机）	END	—	—	—	—

4. 参考章节 27.4.10，向 I2C 主机的 TX RAM 写入从机地址和要发送的数据。可选 FIFO 方式和直接访问方式。
5. 在I2C_SLAVE_ADDR_REG（从机）的I2C_SLAVE_ADDR（从机）设置 I2C 从机的地址。
6. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
7. 向 I2C_TRANS_START（主机）和 I2C_TRANS_START（从机）位写 1 开始传输。
8. I2C 从机比较 I2C 主机发送的从机地址和自己的 I2C_SLAVE_ADDR（从机），当 I2C 主机 WRITE 命令中的 ack_check_en（主机）配置为 1 时，I2C 主机会在发送完每个字节之后进行 ACK 检测。若 ack_check_en 配置为 0，则不会对 ACK 检测，会默认为匹配。
   - 匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平一致，传输继续。
   - 不匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平不一致，I2C 主机产生 I2C_NACK_INT（主机）中断，停止发送数据并且产生 STOP。
9. I2C 主机发送数据，并根据 ack_check_en（主机）配置的不同进行或不进行 ACK 检测。
10. 等到 I2C_END_DETECT_INT（主机）中断产生后，设置 I2C_END_DETECT_INT_CLR（主机）为 1 来清除中断。
11. 更新 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	WRITE	ack_value	ack_exp	1	M
I2C_COMMAND1（主机）	END/STOP	—	—	—	—

12. 向 I2C 主机的 TX RAM 写入 M 个要发送的数据，可以用 FIFO 方式或直接访问方式。
13. 向 I2C_TRANS_START（主机）位写 1 开始传输，并重复步骤 9 的流程。
14. 若指令为 STOP，I2C 主机执行 STOP 命令结束传输，并产生 I2C_TRANS_COMPLETE_INT（主机）中断。
15. 若 I2C_COMMAND1（主机）的指令为 END，则重复步骤 10。
16. 更新 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND1（主机）	STOP	—	—	—	—

17. 向 I2C_TRANS_START（主机）位写 1 开始传输。
18. I2C 主机执行 STOP 命令结束传输，并产生 I2C_TRANS_COMPLETE_INT（主机）中断。

27.5.5 I2C 主机读取从机，7 位寻址，单次命令序列

27.5.5.1 场景介绍

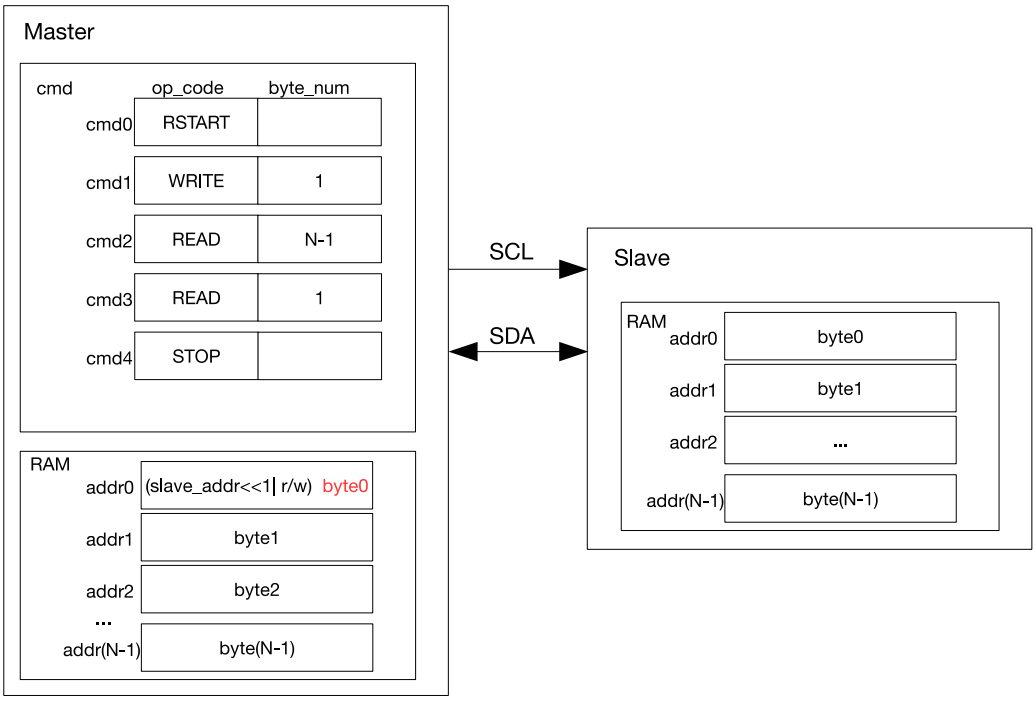


图 27.5-5. I2C 主机读 7 位寻址的从机

图 27.5-5 I2C 主机从 7 位寻址 I2C 从机读取 N 个字节数据的寄存器或 RAM 的值。cmd1 为 WRITE 命令，I2C 主机会将 I2C 从机的地址发送出去。该命令发送的字节是 7 位 I2C 从机地址以及读写标志位。读写标志位为 1 表示读操作。I2C 从机在地址匹配成功之后即开始发送数据给 I2C 主机。I2C 主机根据 READ 命令中设置的 ack_value，在接收完一个字节的数据之后回复 ACK。

图 27.5-5 中 READ 分成两次，I2C 主机对 cmd2 中 N-1 个数据均回复 ACK，对 cmd3 中的数据即传输的最后一个数据回复 NACK，实际使用时可以根据需要进行配置。在存储接收的数据时，I2C 主机从 RX RAM 的首地址开始存储，即为图中红色 byte0 存储的位置。

27.5.5.2 配置示例

1. 设置 I2C_MS_MODE（主机）为 1，I2C_MS_MODE（从机）为 0。
2. 推荐将 I2C_SLAVE_SCL_STRETCH_EN（从机）置 1，以便 I2C 从机在需要发送数据时可以先把 SCL 拉低来给软件向 I2C 从机的 TX RAM 中写数提供时间，否则 I2C 从机需要在主机开始传输前提前准备好数据。以下配置流程均按照 I2C_SLAVE_SCL_STRETCH_EN（从机）为 1 的情况进行。
3. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
4. 配置 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	RSTART	—	—	—	—
I2C_COMMAND1（主机）	WRITE	0	0	1	1
I2C_COMMAND2（主机）	READ	0	0	1	N-1
I2C_COMMAND3（主机）	READ	1	0	1	1
I2C_COMMAND4（主机）	STOP	—	—	—	—

5. 参考章节 27.4.10，向 I2C 主机的 TX RAM 写入从机地址。可选 FIFO 方式和直接访问方式。
6. 在 I2C_SLAVE_ADDR_REG（从机）的 I2C_SLAVE_ADDR（从机）设置 I2C 从机的地址。
7. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
8. 向 I2C_TRANS_START（主机）写 1 开始主机的传输。
9. 参考章节 27.4.14，启动从机的传输。
10. I2C 从机比较 I2C 主机发送的从机地址和自己的 I2C_SLAVE_ADDR（从机），当 I2C 主机 WRITE 命令中的 ack_check_en（主机）配置为 1 时，I2C 主机会在发送完每个字节之后进行 ACK 检测。若 ack_check_en 配置为 0，则不会对 ACK 检测，会默认为匹配。
    - 匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平一致，传输继续。
    - 不匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平不一致，I2C 主机产生 I2C_NACK_INT（主机）中断，停止发送数据并且产生 STOP。
11. 等待 I2C_SLAVE_STRETCH_INT（从机），读取 I2C_STRETCH_CAUSE 的值为 0，此时从机地址与 SDA 线上发送的地址相匹配，且从机要发送数据。
12. 参考章节 27.4.10，向 I2C 从机的 TX RAM 写入要发送的数据。可选 FIFO 方式和直接访问方式。
13. 将 I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1，释放 SCL 总线。
14. I2C 从机发送数据，I2C 主机会根据当前 READ 指令对应的 ack_check_en（主机）配置的不同进行或不进行 ACK 检测。
15. 若 I2C 主机要读取的数超过 32 个，当发送数据全部发完，在 I2C 从机的 TX RAM 空时产生 I2C_SLAVE_STRETCH_INT（从机）中断。此时 I2C 从机会将 SCL 拉低，软件在此期间可以继续向 I2C 从机的 TX RAM 填充数据，也可以从 I2C 主机的 RX RAM 读取数据。等完成操作后再将 I2C_SLAVE_STRETCH_INT_CLR（从机）置 1 清除中断，将 I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1 释放 SCL 总线。
16. 当 I2C 主机接收最后一个数据时，将 ack_value（主机）设成 1，I2C 从机接收到 NACK 中断，停止发送。
17. 当整个传输正常结束，I2C 主机执行 STOP 命令，并产生 I2C_TRANS_COMPLETE_INT（主机）中断。

27.5.6 I2C 主机读取从机，10 位寻址，单次命令序列

27.5.6.1 场景介绍

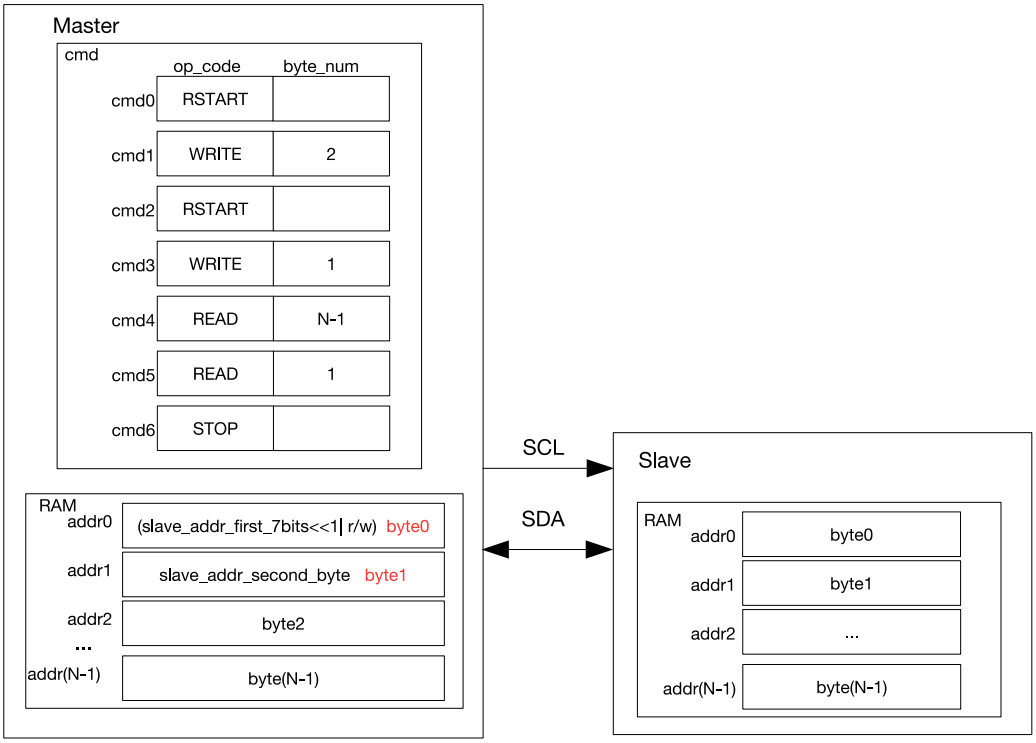


图 27.5-6. I2C 主机读 10 位寻址的从机

图 27.5-6 为 I2C 主机从 10 位寻址的 I2C 从机中读取数据的寄存器或 RAM 的值。相比于 7 位寻址，I2C 主机的第一写命令的字节数为两个字节，相应 TX RAM 中存储两个字节的 I2C 从机 10 位地址，且第一个地址字节的 R/W 位为 W（主机）。之后再次发送 RSTART，并重复发送第一个地址字节，R/W 位为 R（从机）。之后主机从从机中读取数据。两个字节地址的配置方式与章节 27.5.2 的相同。

27.5.6.2 配置示例

1. 设置 I2C_MS_MODE（主机）为 1，I2C_MS_MODE（从机）为 0。
2. 推荐将 I2C_SLAVE_SCL_STRETCH_EN（从机）置 1，以便 I2C 从机在需要发送数据时可以先把 SCL 拉低来给软件向 I2C 从机的 TX RAM 中写数提供时间，否则 I2C 从机需要在主机开始传输前提前准备好数据。以下配置流程均按照I2C_SLAVE_SCL_STRETCH_EN（从机）为 1 的情况进行。
3. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
4. 配置 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	RSTART	—	—	—	—
I2C_COMMAND1（主机）	WRITE	0	0	1	2
I2C_COMMAND2（主机）	RSTART	—	—	—	—
I2C_COMMAND3（主机）	WRITE	0	0	1	1
I2C_COMMAND4（主机）	READ	0	0	1	N-1
I2C_COMMAND5（主机）	READ	1	0	1	1
I2C_COMMAND6（主机）	STOP	—	—	—	—

5. 在 I2C_SLAVE_ADDR_REG（从机）的 I2C_SLAVE_ADDR（从机）设置 I2C 从机的 10 位从机地址，并将 I2C_ADDR_10BIT_EN（从机）置 1 使能 10 位寻址模式。
6. 向 I2C 主机的 TX RAM 写入从机地址和要发送的数据，第一个地址字节是 ((0x78 | I2C_SLAVE_ADDR[9:8])«1)|R/W，R/W 位为 W（主机）；第二个地址字节是I2C_SLAVE_ADDR[7:0]。第三个字节是重复的第一个地址字节加上 R/W 位，其中 R/W 位为 R（从机）。可选 FIFO 方式和直接访问方式。
7. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
8. 向 I2C_TRANS_START（主机）写 1 开始主机的传输。
9. 参考章节 27.4.14，启动从机的传输。
10. I2C 从机比较 I2C 主机发送的从机地址和自己的 I2C_SLAVE_ADDR（从机），当 I2C 主机 WRITE 命令中的 ack_check_en（主机）配置为 1 时，I2C 主机会在发送完每个字节之后进行 ACK 检测。若 ack_check_en 配置为 0，则不会对 ACK 检测，会默认为匹配。
    - 匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平一致，传输继续。
    - 不匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平不一致，I2C 主机产生 I2C_NACK_INT（主机）中断，停止发送数据并且产生 STOP。
11. I2C 主机发送 RSTART 命令，并发送 TX RAM 里的第三个字节，即为重复的地址字节和 R 位。
12. I2C 从机重复执行步骤 10，若地址匹配，继续后面的步骤。
13. 等待 I2C_SLAVE_STRETCH_INT（从机），读取 I2C_STRETCH_CAUSE 的值为 0，此时从机地址与 SDA 线上发送的地址相匹配，且从机要发送数据。
14. 参考章节 27.4.10，向 I2C 从机的 TX RAM 写入要发送的数据。可选 FIFO 方式和直接访问方式。
15. 将 I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1，释放 SCL 总线。
16. I2C 从机发送数据，I2C 主机会根据当前 READ 指令对应的 ack_check_en（主机）配置的不同进行或不进行 ACK 检测。
17. 若 I2C 主机要读取的数超过 32 个字节，当发送数据全部发完，在 I2C 从机的 TX RAM 空时产生 I2C_SLAVE_STRETCH_INT（从机）中断。此时 I2C 从机会将 SCL 拉低，软件在此期间可以继续向 I2C 从机的 TX RAM 填充数据，也可以从 I2C 主机的 RX RAM 读取数据。等完成操作后再将 I2C_SLAVE_STRETCH_INT_CLR（从机）置 1 清除中断，将 I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1 释放 SCL 总线。
18. 当 I2C 主机接收最后一个数据时，将 ack_value（主机）设成 1，I2C 从机接收到 NACK 中断，停止发送。
19. 当整个传输正常结束，I2C 主机执行 STOP 命令，并产生 I2C_TRANS_COMPLETE_INT（主机）中断。

27.5.7 I2C 主机读取从机，7 位双寻址，单次命令序列

27.5.7.1 场景介绍

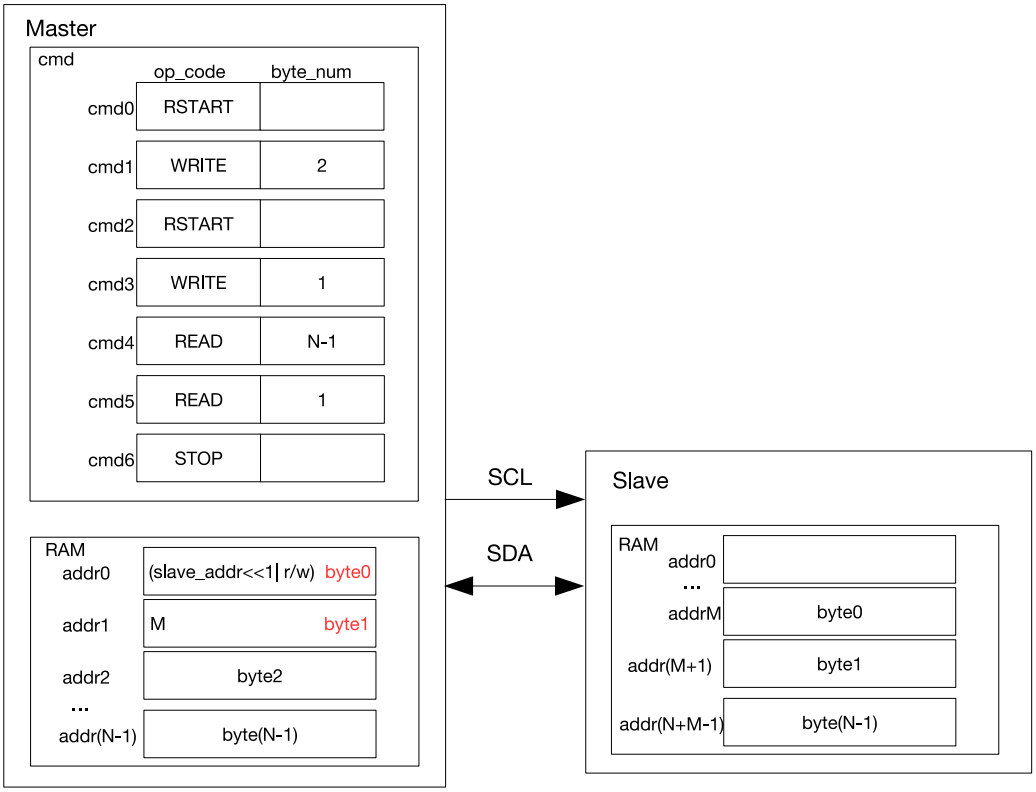


图 27.5-7. I2C 主机从 7 位寻址从机的 M 地址读取 N 个数据

图 27.5-7 为 I2C 主机从 I2C 从机中指定地址读取数据的寄存器或 RAM 的值。主机在传输开始发送 2 个地址字节，第一个地址字节是从机的 7 位地址加 R/W 位，R/W 位为 W（主机）；第二个地址字节是从机的内存地址 M。之后再次发送 RSTART，并重复发送第一个地址字节，R/W 位变为 R（从机）。之后主机从从机的 AddrM 地址开始读取数据。

27.5.7.2 配置示例

1. 设置 I2C_MS_MODE（主机）为 1，I2C_MS_MODE（从机）为 0。
2. 推荐将 I2C_SLAVE_SCL_STRETCH_EN（从机）置 1，以便 I2C 从机在需要发送数据时可以先把 SCL 拉低来给软件向 I2C 从机的 TX RAM 中写数提供时间，否则 I2C 从机需要在主机开始传输前提前准备好数据。以下配置流程均按照 I2C_SLAVE_SCL_STRETCH_EN（从机）为 1 的情况进行。
3. 设置 I2C_FIFO_ADDR_CFG_EN（从机）为 1 来使能双寻址模式。
4. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
5. 配置 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	RSTART	—	—	—	—
I2C_COMMAND1（主机）	WRITE	0	0	1	2
I2C_COMMAND2（主机）	RSTART	—	—	—	—
I2C_COMMAND3（主机）	WRITE	0	0	1	1
I2C_COMMAND4（主机）	READ	0	0	1	N-1

I2C_COMMAND5（主机）	READ	1	0	1	1
I2C_COMMAND6（主机）	STOP	—	—	—	—

6. 在 I2C_SLAVE_ADDR_REG（从机）的 I2C_SLAVE_ADDR（从机）设置 I2C 从机的 7 位地址，I2C_ADDR_10BIT_EN（从机）置 0 使能 7 位寻址模式。
7. 参考章节 27.4.10，向 I2C 主机的 TX RAM 写入从机地址和要发送的数据，第一个地址字节是 (I2C_SLAVE_ADDR[6:0])«1)|R/W，R/W 位为 W（主机）；第二个地址字节是 I2C 从机的内存地址 M。第三个字节是重复的第一个地址字节加上 R/W 位，其中 R/W 位为 R（从机）。可选 FIFO 方式和直接访问方式。
8. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
9. 向 I2C_TRANS_START（主机）写 1 开始主机的传输。
10. 参考章节 27.4.14，启动 I2C 从机的传输。
11. I2C 从机比较 I2C 主机发送的从机地址和自己的 I2C_SLAVE_ADDR（从机），当 I2C 主机 WRITE 命令中的 ack_check_en（主机）配置为 1 时，I2C 主机会在发送完每个字节之后进行 ACK 检测。若 ack_check_en 配置为 0，则不会对 ACK 检测，会默认为匹配。
    - 匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平一致，传输继续。
    - 不匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平不一致，I2C 主机产生 I2C_NACK_INT（主机）中断，停止发送数据并且产生 STOP。
12. I2C 从机接收到 I2C 主机发送的内存地址，完成 TX RAM 的地址偏移。
13. I2C 主机发送 RSTART 命令，并发送 TX RAM 里的第三个字节，即为重复的地址字节和 R 位。
14. I2C 从机重复执行步骤 11，若地址匹配，继续后面的步骤。
15. 等待 I2C_SLAVE_STRETCH_INT（从机），读取 I2C_STRETCH_CAUSE 的值为 0，此时从机地址与 SDA 线上发送的地址相匹配，且从机要发送数据。
16. 参考章节 27.4.10，向 I2C 从机的 TX RAM 写入要发送的数据。可选 FIFO 方式和直接访问方式。
17. 将 I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1，释放 SCL 总线。
18. I2C 从机发送数据，I2C 主机会根据当前 READ 指令对应的 ack_check_en（主机）配置的不同进行或不进行 ACK 检测。
19. 若 I2C 主机要读取的数超过 32 个，当发送数据全部发完，在 I2C 从机的 TX RAM 空时产生I2C_SLAVE_STRETCH_INT（从机）中断。此时 I2C 从机会将 SCL 拉低，软件在此期间可以继续向 I2C 从机的 TX RAM 填充数据，也可以从 I2C 主机的 RX RAM 读取数据。等完成操作后再将I2C_SLAVE_STRETCH_INT_CLR（从机）置 1，清除中断；I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1，释放 SCL 总线。
20. 当 I2C 主机接收最后一个数据时，将 ack_value（主机）设成 1，I2C 从机接收到 NACK 中断，停止发送。
21. 当整个传输正常结束，I2C 主机执行 STOP 命令，并产生 I2C_TRANS_COMPLETE_INT（主机）中断。

27.5.8 I2C 主机读取从机，7 位寻址，多次命令序列

27.5.8.1 场景介绍

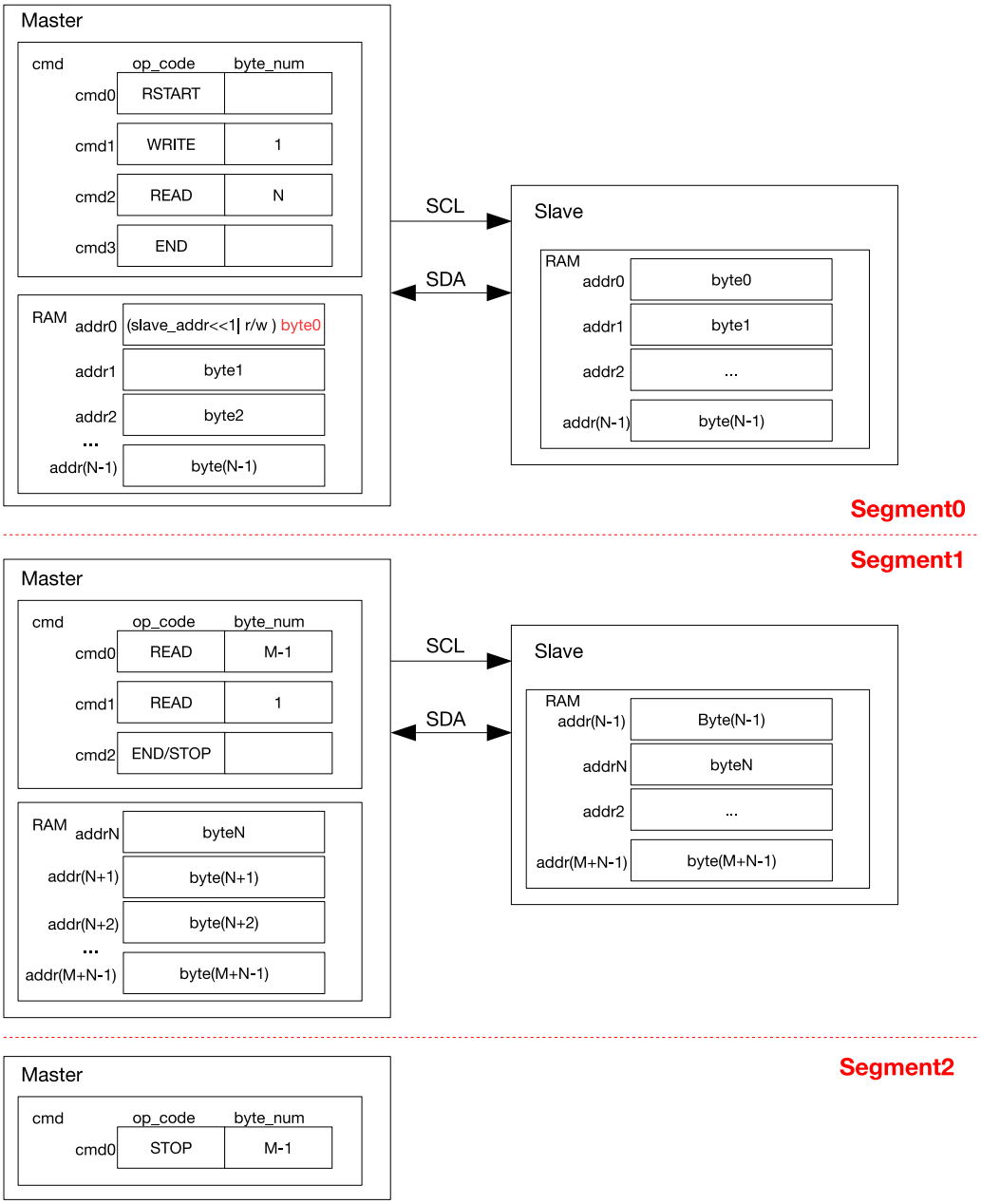


图 27.5-8. I2C 主机分段读 7 位寻址的从机

图 27.5-8 为 I2C 主机通过 END 命令分三段或者分两段，从 I2C 从机读取 N+M 个数据的流程图。

1. 第一段流程和 27.5-5 类似，只是最后一个命令变为 END。
2. 接着在从机的 TX RAM 中准备好数据，置位 I2C_TRANS_START，当执行到 END 命令时，I2C 主机可以更新命令寄存器和 RAM 的内容，如第二段所示，并且清零其对应的 I2C_END_DETECT_INT 中断。当第二段中 cmd2 为 STOP 时，即两段读 I2C 从机，置位 I2C_TRANS_START，I2C 主机继续传输数据，最后发送 STOP 位来停止传输。
3. 当第二段中 cmd2 为 END 时，在 I2C 主机完成第二次数据传输，并检测到 I2C 主机的 I2C_END_DETECT_INT 中断后，配置 cmd 如第三段所示。置位 I2C_TRANS_START，I2C 主机发送 STOP 位停止传输。

27.5.8.2 配置示例

1. 设置 I2C_MS_MODE（主机）为 1，I2C_MS_MODE（从机）为 0。
2. 推荐将 I2C_SLAVE_SCL_STRETCH_EN（从机）置 1，以便 I2C 从机在需要发送数据时可以先把 SCL 拉低来给软件向 I2C 从机的 TX RAM 中写数提供时间，否则 I2C 从机需要在主机开始传输前提前准备好数据。以下配置流程均按照 I2C_SLAVE_SCL_STRETCH_EN（从机）为 1 的情况进行。
3. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
4. 配置 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	RSTART	—	—	—	—
I2C_COMMAND1（主机）	WRITE	0	0	1	1
I2C_COMMAND2（主机）	READ	0	0	1	N
I2C_COMMAND3（主机）	END	—	—	—	—

5. 向 I2C 主机的 TX RAM 写入从机地址，可选 FIFO 方式和直接访问方式。
6. 在 I2C_SLAVE_ADDR_REG（从机）的I2C_SLAVE_ADDR（从机）设置 I2C 从机的地址。
7. 向 I2C_CONF_UPGATE（主机）和 I2C_CONF_UPGATE（从机）写 1 来同步寄存器。
8. 向 I2C_TRANS_START（主机）写 1 开始主机的传输。
9. 参考章节 27.4.14，启动从机的传输。
10. I2C 从机比较 I2C 主机发送的从机地址和自己的 I2C_SLAVE_ADDR（从机），当 I2C 主机 WRITE 命令中的 ack_check_en（主机）配置为 1 时，I2C 主机会在发送完每个字节之后进行 ACK 检测。若 ack_check_en 配置为 0，则不会对 ACK 检测，会默认为匹配。
    - 匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平一致，传输继续。
    - 不匹配：接收的 ACK 值与 WRITE 命令中的 ack_exp（主机）电平不一致，I2C 主机产生 I2C_NACK_INT（主机）中断，停止发送数据并且产生 STOP。
11. 等待 I2C_SLAVE_STRETCH_INT（从机），读取 I2C_STRETCH_CAUSE 的值为 0，此时从机地址与 SDA 线上发送的地址相匹配，且从机要发送数据。
12. 参考章节 27.4.10，向 I2C 从机的 TX RAM 写入要发送的数据。可选 FIFO 方式和直接访问方式。
13. 将 I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1，释放 SCL 总线。
14. I2C 从机发送数据，I2C 主机会根据当前 READ 指令对应的 ack_check_en（主机）配置的不同进行或不进行 ACK 检测。
15. 若 I2C 主机一个 READ 指令要读取的数 N 或 M 超过 32 个字节，当 I2C 从机的 TX RAM 中发送数据全部发完，为空时产生 I2C_SLAVE_STRETCH_INT（从机）中断。此时 I2C 从机会将 SCL 拉低，软件在此期间可以继续向 I2C 从机的 TX RAM 填充数据，也可以从 I2C 主机的 RX RAM 读取数据。等完成操作后再将 I2C_SLAVE_STRETCH_INT_CLR（从机）置 1 清除中断，将 I2C_SLAVE_SCL_STRETCH_CLR（从机）置 1 释放 SCL 总线。
16. 等到一次 READ 指令完成，I2C 主机执行 END 指令，I2C_END_DETECT_INT（主机）中断产生后，设置 I2C_END_DETECT_INT_CLR（主机）为 1 来清除中断。

17. 更新 I2C 主机的指令寄存器，有两种设置方式：

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	READ	ack_value	ack_exp	1	M
I2C_COMMAND1（主机）	END	—	—	—	—

或者

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND0（主机）	READ	0	0	1	M-1
I2C_COMMAND1（主机）	READ	1	0	1	1
I2C_COMMAND2（主机）	STOP	—	—	—	—

18. 向 I2C 从机的 TX RAM 写入 M 个字节要发送的数据，若 M 大于 32，重复步骤 12，可以用 FIFO 方式或直接访问方式。
19. 向 I2C_TRANS_START（主机）位写 1 开始传输，并重复步骤 14 的流程。
20. 若最后一个指令为 STOP，则当 I2C 主机接收最后一个数据时，将 ack_value（主机）设成 1，I2C 从机接收到 NACK 中断，停止发送。I2C 主机执行 STOP 命令结束传输，并产生 I2C_TRANS_COMPLETE_INT（主机）中断。
21. 若最后一个指令为为 END，则重复步骤 16，并在完成后继续下面的步骤。
22. 更新 I2C 主机的指令寄存器。

指令寄存器	op_code	ack_value	ack_exp	ack_check_er	byte_num
I2C_COMMAND1（主机）	STOP	—	—	—	—

23. 向 I2C_TRANS_START（主机）位写 1 开始传输。
24. I2C 主机执行 STOP 命令结束传输，并产生 I2C_TRANS_COMPLETE_INT（主机）中断。

27.6 中断

- I2C_SLAVE_STRETCH_INT: 从机模式下，当出现可触发时钟拉伸的时，产生此中断。
- I2C_DET_START_INT：主机或从机检测到 I2C START 位时，触发此中断。
- I2C_SCL_MAIN_ST_TO_INT: 当 I2C 主状态机 SCL_MAIN_FSM 保持某个状态超过 I2C_SCL_MAIN_ST_TO_I2C[23:0] 个模块时钟周期时，触发此中断。
- I2C_SCL_ST_TO_INT: 当 I2C 状态机 SCL_FSM 保持某个状态超过 I2C_SCL_ST_TO_I2C[23:0] 个模块时钟周期时，触发此中断。
- I2C_RXFIFO_UDF_INT: 当 I2C 通过 APB 总线读取 RX FIFO，但 RX FIFO 为空时，触发该中断。
- I2C_TXFIFO_OVF_INT: 当 I2C 通过 APB 总线写 TX FIFO，但 TX FIFO 为满时，触发该中断。
- I2C_NACK_INT: 当 I2C 配置为主机时，接收到的 ACK 与命令中期望的 ACK 值不一致时，即触发该中断；当 I2C 配置为从机时，接收到的 ACK 值为 1 时即触发该中断。
- I2C_TRANS_START_INT: 当 I2C 发送一个 START 位时，即触发该中断。

- I2C_TIME_OUT_INT: 在传输过程中，当 I2C SCL 保持为高或为低电平的时间超过 $2^{I2C\_TIME\_OUT\_VALUE}$ 个模块时钟后，即触发该中断。
- I2C_TRANS_COMPLETE_INT: 当 I2C 检测到 STOP 位时，即触发该中断。
- I2C_MST_TXFIFO_UDF_INT: 当 I2C 主机的 TX FIFO 下溢时，触发此中断。
- I2C_ARBITRATION_LOST_INT: 当 I2C 主机的 SCL 为高电平，SDA 输出值与输入值不相等时，即触发该中断。
- I2C_BYTE_TRANS_DONE_INT: 当 I2C 发送或接收一个字节，即触发该中断。
- I2C_END_DETECT_INT：当 I2C 主机命令的 op_code 为 END，且检测到 I2C END 状态时，触发此中断。
- I2C_RXFIFO_OVF_INT：当 I2C RX FIFO 上溢时，触发此中断。
- I2C_TXFIFO_WM_INT：I2C TX FIFO 水标中断。当 I2C_FIFO_PRT_EN 为 1，且 TX FIFO 指针小于 I2C_TXFIFO_WM_THRHD[4:0] 时，触发此中断。
- I2C_RXFIFO_WM_INT：I2C RX FIFO 水标中断。当 I2C_FIFO_PRT_EN 为 1，且 RX FIFO 指针大于 I2C_RXFIFO_WM_THRHD[4:0] 时，触发此中断。

27.7 寄存器列表

本小节的所有地址均为相对于 I2C 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

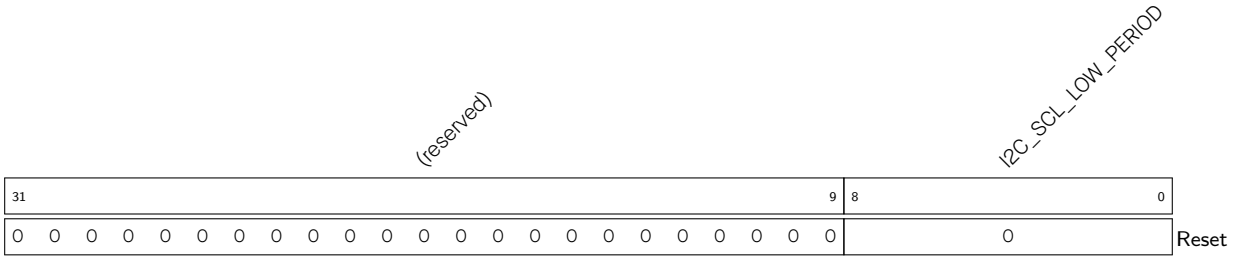
名称	描述	地址	访问
时序寄存器			
I2C_SCL_LOW_PERIOD_REG	配置 SCL 的低电平宽度	0x0000	R/W
I2C_SDA_HOLD_REG	配置 SCL 下降沿后的保持时间	0x0030	R/W
I2C_SDA_SAMPLE_REG	配置 SCL 上升沿后的采样时间	0x0034	R/W
I2C_SCL_HIGH_PERIOD_REG	配置 SCL 时钟的高电平宽度	0x0038	R/W
I2C_SCL_START_HOLD_REG	配置 START 命令产生时 SDA 下降沿和 SCL 下降沿之间的间隔时间	0x0040	R/W
I2C_SCL_RSTART_SETUP_REG	配置 SCL 上升沿和 SDA 下降沿之间的延迟	0x0044	R/W
I2C_SCL_STOP_HOLD_REG	配置 STOP 命令生成时 SCL 边沿的延迟	0x0048	R/W
I2C_SCL_STOP_SETUP_REG	配置 STOP 命令生成时 SDA 和 SCL 上升沿之间的间隔时间	0x004C	R/W
I2C_SCL_ST_TIME_OUT_REG	SCL 状态超时寄存器	0x0078	R/W
I2C_SCL_MAIN_ST_TIME_OUT_REG	SCL 主要状态超时寄存器	0x007C	R/W
配置寄存器			
I2C_CTR_REG	传输配置寄存器	0x0004	varies
I2C_TO_REG	超时控制寄存器	0x000C	R/W
I2C_SLAVE_ADDR_REG	从机地址配置寄存器	0x0010	R/W
I2C_FIFO_CONF_REG	FIFO 配置寄存器	0x0018	R/W
I2C_FILTER_CFG_REG	SCL 和 SDA 滤波配置寄存器	0x0050	R/W
I2C_CLK_CONF_REG	I2C 时钟配置寄存器	0x0054	R/W
I2C_SCL_SP_CONF_REG	电源配置寄存器	0x0080	varies
I2C_SCL_STRETCH_CONF_REG	配置 I2C 从机 SCL 时钟拉伸	0x0084	varies
状态寄存器			
I2C_SR_REG	描述 I2C 的工作状态	0x0008	RO
I2C_FIFO_ST_REG	FIFO 状态寄存器	0x0014	RO
I2C_DATA_REG	读/写 FIFO 寄存器	0x001C	R/W
中断寄存器			
I2C_INT_RAW_REG	原始中断状态	0x0020	R/ SS/ WTC
I2C_INT_CLR_REG	中断清除位	0x0024	WT
I2C_INT_ENA_REG	中断使能位	0x0028	R/W
I2C_INT_STATUS_REG	捕捉 I2C 通信事件的状态	0x002C	RO
命令寄存器			
I2C_COMD0_REG	I2C 命令寄存器 0	0x0058	varies
I2C_COMD1_REG	I2C 命令寄存器 1	0x005C	varies
I2C_COMD2_REG	I2C 命令寄存器 2	0x0060	varies
I2C_COMD3_REG	I2C 命令寄存器 3	0x0064	varies

名称	描述	地址	访问
I2C_COMD4_REG	I2C 命令寄存器 4	0x0068	varies
I2C_COMD5_REG	I2C 命令寄存器 5	0x006C	varies
I2C_COMD6_REG	I2C 命令寄存器 6	0x0070	varies
I2C_COMD7_REG	I2C 命令寄存器 7	0x0074	varies
版本寄存器			
I2C_DATE_REG	版本控制寄存器	0x00F8	R/W

27.8 寄存器

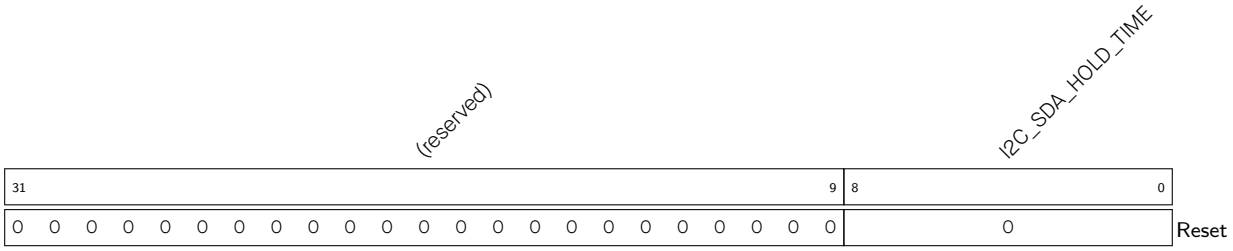
本小节的所有地址均为相对于 I2C 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 27.1. I2C_SCL_LOW_PERIOD_REG (0x0000)



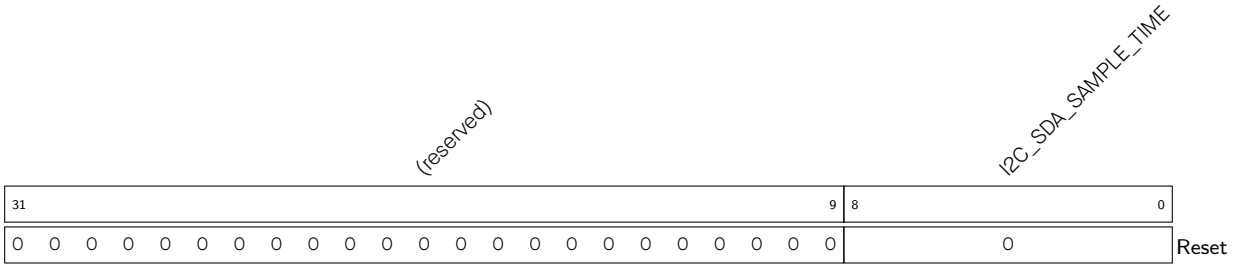
I2C_SCL_LOW_PERIOD 用于配置主机模式下 SCL 低电平的保持时间，以 I2C 控制器时钟周期数为单位。(R/W)

Register 27.2. I2C_SDA_HOLD_REG (0x0030)



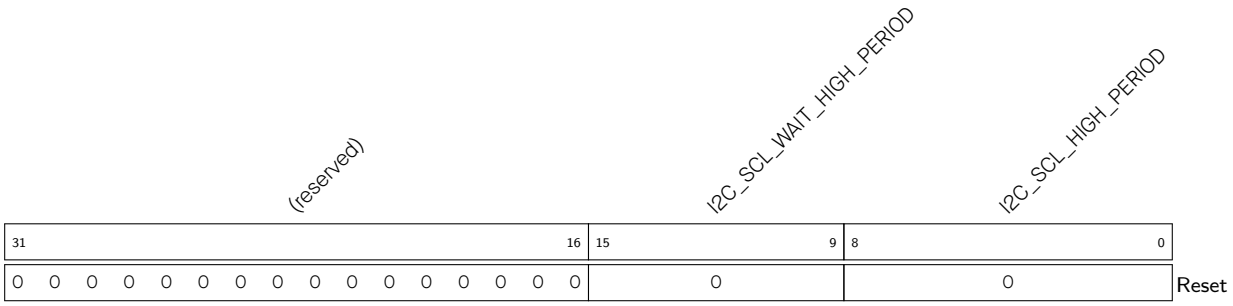
I2C_SDA_HOLD_TIME 用于配置 SCL 下降沿后的数据保持时间，以 I2C 控制器时钟周期数为单位。(R/W)

Register 27.3. I2C_SDA_SAMPLE_REG (0x0034)



I2C_SDA_SAMPLE_TIME 用于配置采样 SDA 的时间，以 I2C 控制器时钟周期数为单位。(R/W)

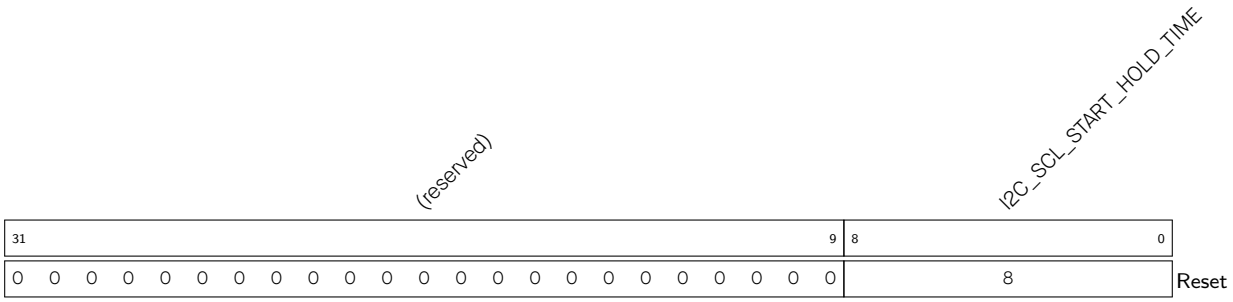
Register 27.4. I2C_SCL_HIGH_PERIOD_REG (0x0038)



I2C_SCL_HIGH_PERIOD 用于配置 SCL 在主机模式下保持高电平的时间，以 I2C 控制器时钟周期数为单位。(R/W)

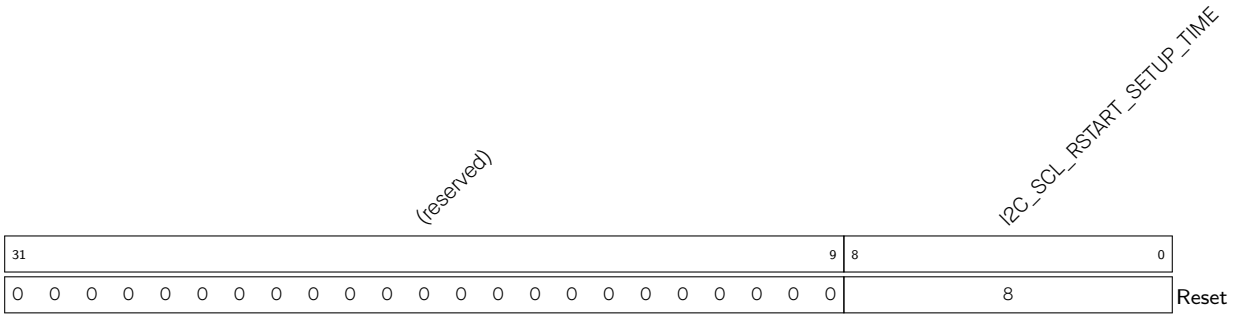
I2C_SCL_WAIT_HIGH_PERIOD 用于配置 SCL_FSM 等待 SCL 在主机模式下翻转至高电平的时间，以 I2C 控制器时钟周期数为单位。(R/W)

Register 27.5. I2C_SCL_START_HOLD_REG (0x0040)



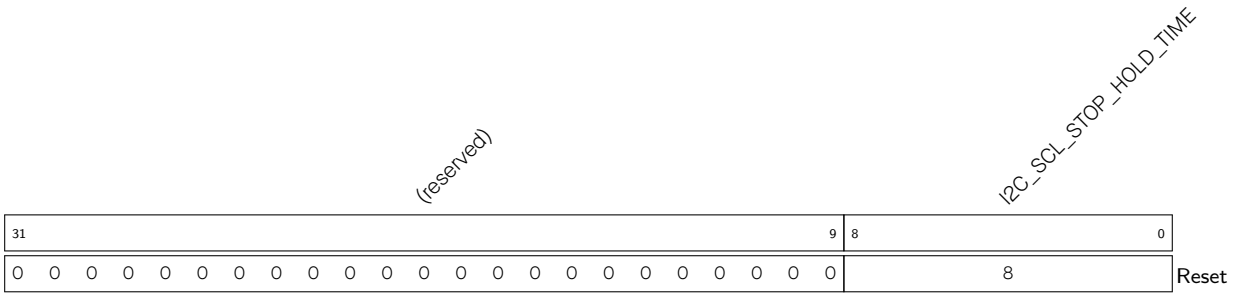
I2C_SCL_START_HOLD_TIME 配置 START 命令产生时 SDA 下降沿和 SCL 下降沿的间隔时间，以 I2C 控制器时钟周期数为单位。(R/W)

Register 27.6. I2C_SCL_RSTART_SETUP_REG (0x0044)



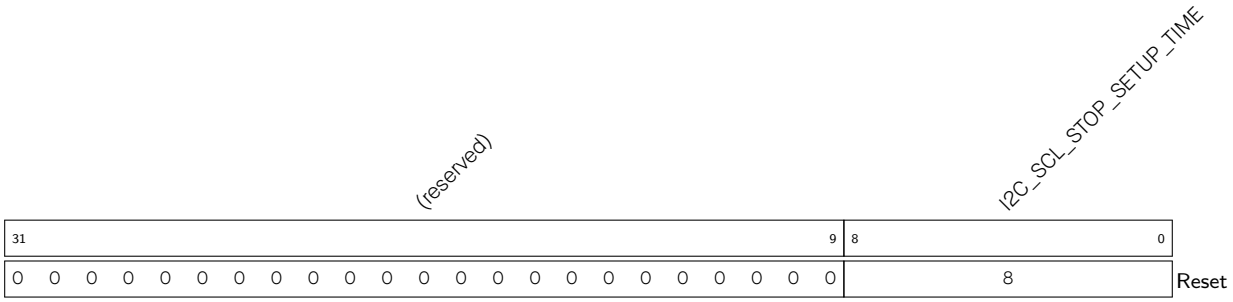
I2C_SCL_RSTART_SETUP_TIME 配置 RSTART 命令产生时 SCL 上升沿和 SDA 下降沿的间隔时间，以 I2C 控制器的时钟周期数为单位。(R/W)

Register 27.7. I2C_SCL_STOP_HOLD_REG (0x0048)



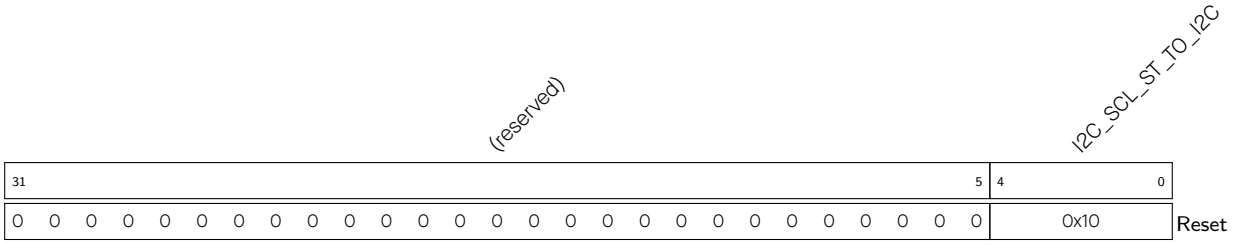
I2C_SCL_STOP_HOLD_TIME 配置 STOP 命令后的延迟，以 I2C 控制器时钟周期数为单位。(R/W)

Register 27.8. I2C_SCL_STOP_SETUP_REG (0x004C)



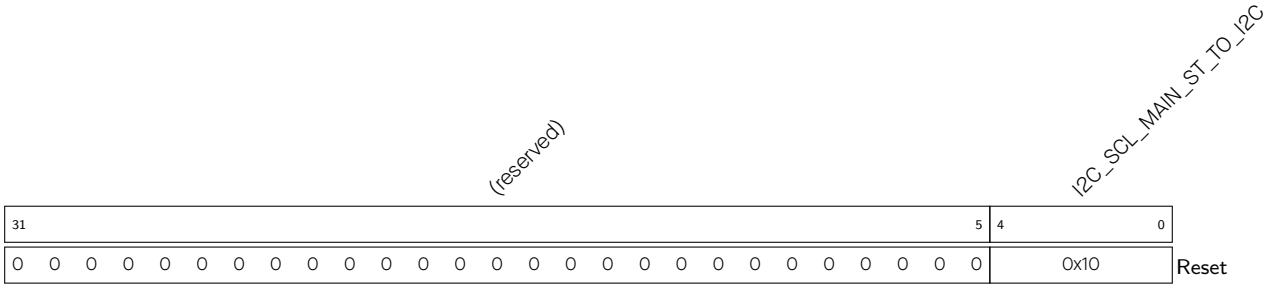
I2C_SCL_STOP_SETUP_TIME 配置 SCL 上升沿和 SDA 上升沿的间隔时间，以 I2C 控制器时钟周期数为单位。(R/W)

Register 27.9. I2C_SCL_ST_TIME_OUT_REG (0x0078)



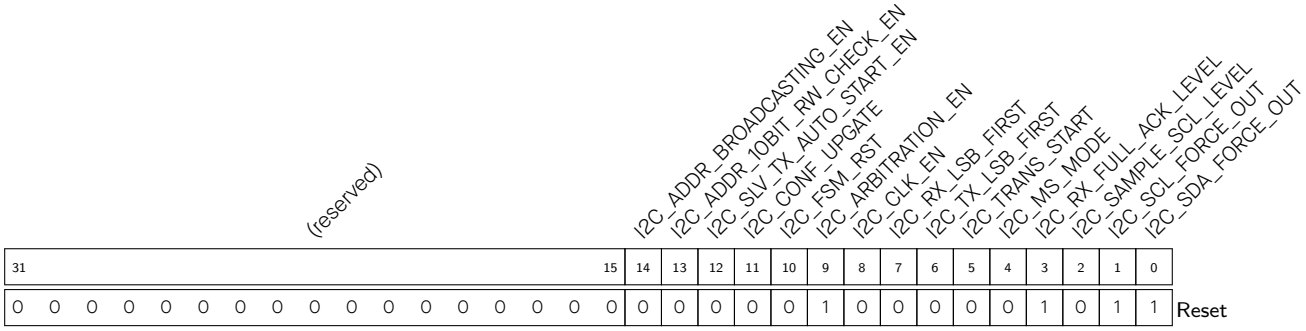
I2C_SCL_ST_TO_I2C SCL_FSM 状态不变的最大时间，不能大于 23。(R/W)

Register 27.10. I2C_SCL_MAIN_ST_TIME_OUT_REG (0x007C)



I2C_SCL_MAIN_ST_TO_I2C SCL_MAIN_FSM 状态不变的最大时间，不能大于 23。(R/W)

Register 27.11. I2C_CTR_REG (0x0004)



I2C_SDA_FORCE_OUT 配置 SDA 输出模式。

0: 开漏输出

1: 直接输出

(R/W)

I2C_SCL_FORCE_OUT 配置 SCL 输出模式。

0: 开漏输出

1: 直接输出

(R/W)

I2C_SAMPLE_SCL_LEVEL 用于选择采样模式。0：SCL 为高电平时采样 SDA 数据；1：SCL 为低电平时采样 SDA 数据。(R/W)

I2C_RX_FULL_ACK_LEVEL 用于配置主机在 I2C_RXFIFO_CNT 达到阈值时需发送的 ACK 电平值。(R/W)

I2C_MS_MODE 置位此位，将 I2C 控制器配置为主机。清零此位，将 I2C 控制器配置为从机。(R/W)

I2C_TRANS_START 置位此位，开始发送 TX FIFO 中的数据。(WT)

I2C_TX_LSB_FIRST 用于控制待发送数据的发送顺序。0：从最高有效位开始发送数据；1：从最低有效位开始发送数据。(R/W)

I2C_RX_LSB_FIRST 用于控制接收数据的存储顺序。0：从最高有效位开始接收数据；1：从最低有效位开始接收数据。(R/W)

I2C_CLK_EN 用于控制 APB_CLK 时钟门控。0：APB_CLK 时钟门控使能，以便节能；1：APB_CLK 时钟一直开启。(R/W)

I2C_ARBITRATION_EN I2C 总线仲裁的使能位。(R/W)

I2C_FSM_RST 用于复位 SCL_FSM。(WT)

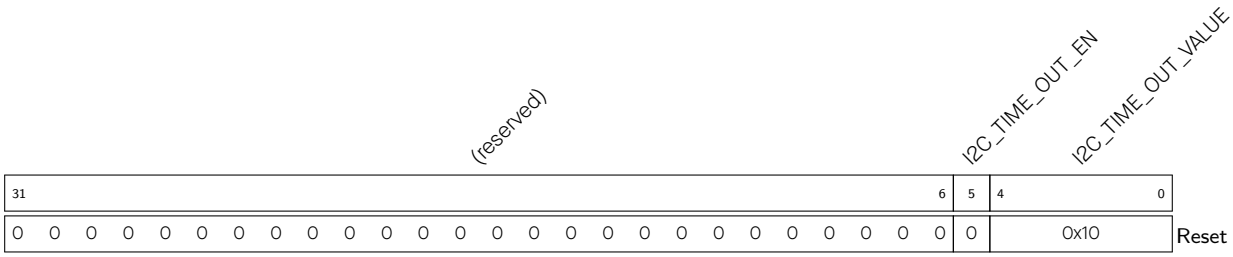
I2C_CONF_UPGATE 同步位。(WT)

I2C_SLV_TX_AUTO_START_EN 从机自动发送数据的使能位。(R/W)

I2C_ADDR_10BIT_RW_CHECK_EN 使能 10 位寻址模式的读写标志位检查功能，检查读写标志是否符合协议。(R/W)

I2C_ADDR_BROADCASTING_EN 使能 7 位寻址模式的广播功能。(R/W)

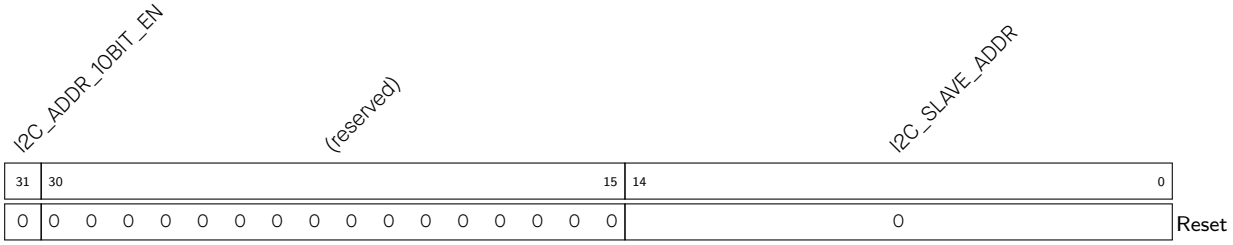
Register 27.12. I2C_TO_REG (0x000C)



I2C_TIME_OUT_VALUE 用于配置接收一位数据的超时时间，以 I2C_SCLK 时钟周期为单位。配置的超时时间为 $2^{I2C\_TIME\_OUT\_VALUE}$ 个时钟周期。(R/W)

I2C_TIME_OUT_EN 超时控制使能位。(R/W)

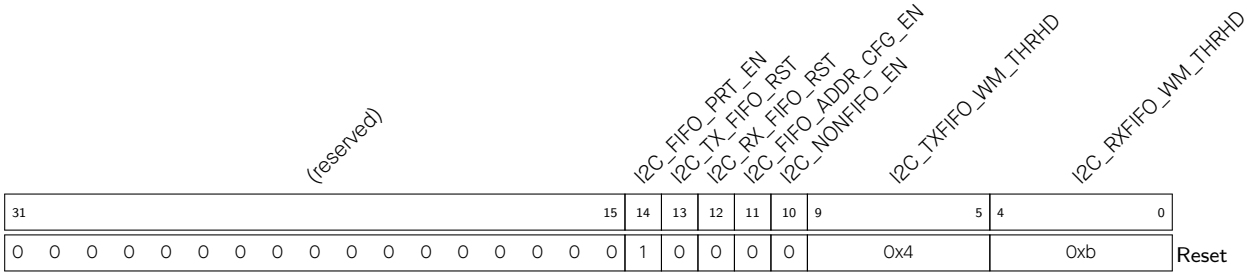
Register 27.13. I2C_SLAVE_ADDR_REG (0x0010)



I2C_SLAVE_ADDR I2C 控制器配置为从机时，该字段用于配置从机地址。(R/W)

I2C_ADDR_10BIT_EN 用于在主机模式下使能从机的 10 位寻址模式。(R/W)

Register 27.14. I2C_FIFO_CONF_REG (0x0018)



I2C_RXFIFO_WM_THRHD 直接访问模式下，RX FIFO 的水标阈值。I2C_FIFO_PRT_EN 为 1 且 RX FIFO 计数值大于 I2C_TXFIFO_WM_THRHD[4:0] 时，I2C_TXFIFO_WM_INT_RAW 位有效。(R/W)

I2C_TXFIFO_WM_THRHD 直接访问模式下，TX FIFO 的水标阈值。I2C_FIFO_PRT_EN 为 1 且 TX FIFO 计数值小于 I2C_TXFIFO_WM_THRHD[4:0] 时，I2C_TXFIFO_WM_INT_RAW 位有效。(R/W)

I2C_NONFIFO_EN 置位此位，使能 APB 直接访问。(R/W)

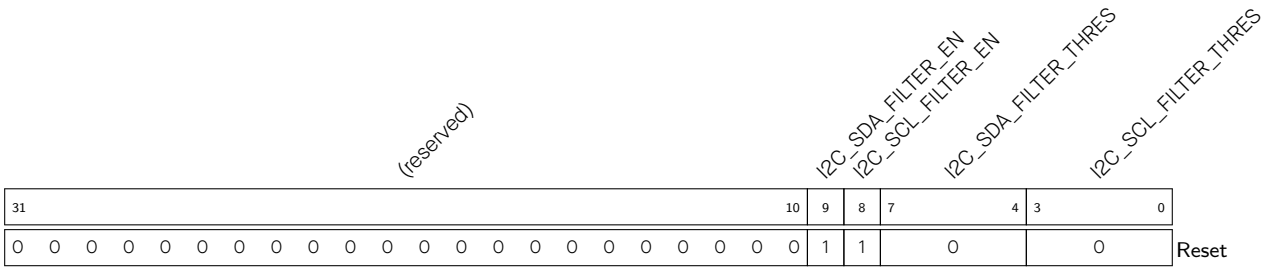
I2C_FIFO_ADDR_CFG_EN 此位置 1 时，从机接收地址字节的后一个字节为从机 RAM 中的偏移地址。(R/W)

I2C_RX_FIFO_RST 置位此位，复位 RX FIFO 。(R/W)

I2C_TX_FIFO_RST 置位此位，复位 TX FIFO 。(R/W)

I2C_FIFO_PRT_EN 直接访问模式下 FIFO 指针的控制使能位。该位控制 TX FIFO 和 RX FIFO 溢出、下溢、为满、为空时的有效位和中断。(R/W)

Register 27.15. I2C_FILTER_CFG_REG (0x0050)



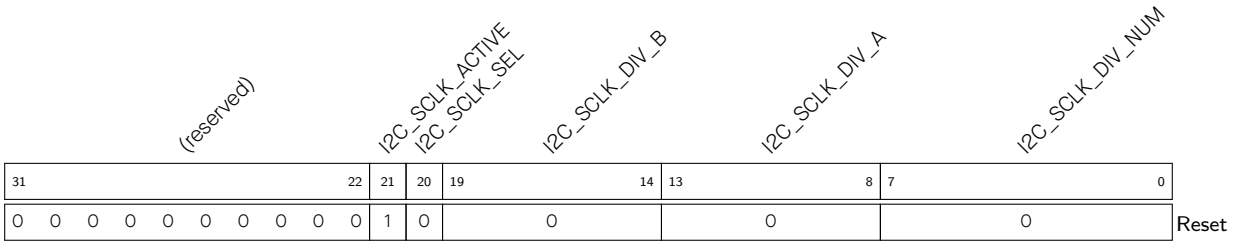
I2C_SCL_FILTER_THRES SCL 输入信号的脉冲宽度小于该字段的值时，I2C 控制器忽略此脉冲。该寄存器的值以 I2C 控制器时钟周期数为单位。(R/W)

I2C_SDA_FILTER_THRES SDA 输入信号的脉冲宽度小于该字段的值时，I2C 控制器忽略此脉冲。该寄存器的值以 I2C 控制器时钟周期数为单位。(R/W)

I2C_SCL_FILTER_EN SCL 的滤波使能位。(R/W)

I2C_SDA_FILTER_EN SDA 的滤波使能位。(R/W)

Register 27.16. I2C_CLK_CONF_REG (0x0054)



I2C_SCLK_DIV_NUM 分频系数的整数部分。(R/W)

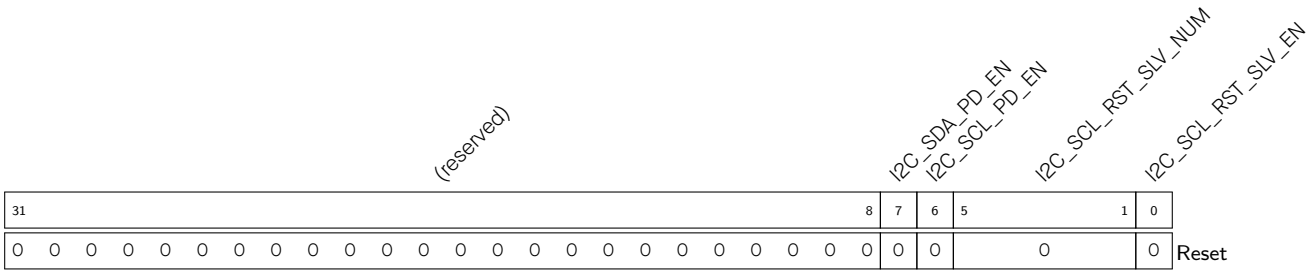
I2C_SCLK_DIV_A 分频系数小数部分的分子。(R/W)

I2C_SCLK_DIV_B 分频系数小数部分的分母。(R/W)

I2C_SCLK_SEL 选择 I2C 控制器的时钟源。0：XTAL_CLK；1：RC_FAST_CLK。(R/W)

I2C_SCLK_ACTIVE I2C 控制器的时钟开关。(R/W)

Register 27.17. I2C_SCL_SP_CONF_REG (0x0080)



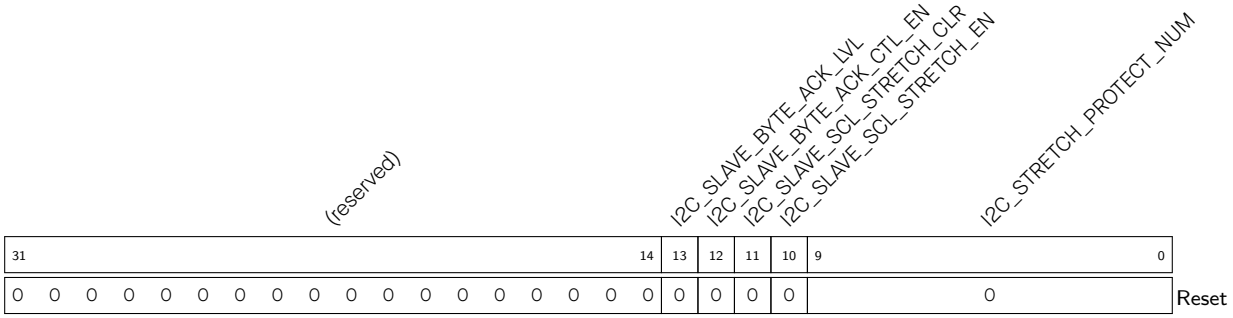
I2C_SCL_RST_SLV_EN I2C 主机处于空闲状态时，置位此位发送 SCL 脉冲。脉冲数量为 I2C_SCL_RST_SLV_NUM[4:0]。(R/W/SC)

I2C_SCL_RST_SLV_NUM 配置主机模式下生成的 SCL 脉冲。I2C_SCL_RST_SLV_EN 为 1 时有效。(R/W)

I2C_SCL_PD_EN 降低 I2C SCL 输出功耗的使能位。0：正常工作；1：不工作，降低功耗。将 I2C_SCL_FORCE_OUT 和 I2C_SCL_PD_EN 置 1 拉伸 SCL。(R/W)

I2C_SDA_PD_EN 降低 I2C SDA 输出功耗的使能位。0：正常工作；1：不工作，降低功耗。将 I2C_SDA_FORCE_OUT 和 I2C_SDA_PD_EN 置 1 拉伸 SDA 。(R/W)

Register 27.18. I2C_SCL_STRETCH_CONF_REG (0x0084)



I2C_STRETCH_PROTECT_NUM 配置 SCL 时钟拉伸释放后的保护时间，通常设置为大于 SDA 的建立时间即可。(R/W)

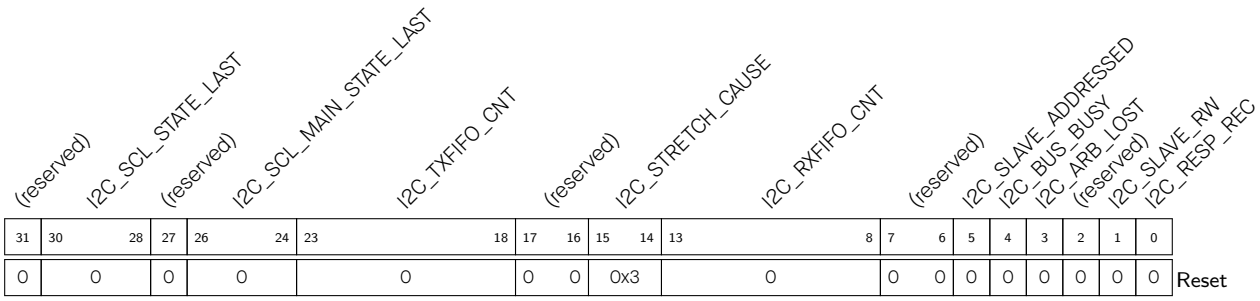
I2C_SLAVE_SCL_STRETCH_EN SCL 时钟拉伸的使能位。0：关闭；1：使能。I2C_SLAVE_SCL_STRETCH_EN 为 1，且出现可触发时钟拉伸的事件时，拉伸 SCL 时钟。SCL 时钟拉伸的原因可见 I2C_STRETCH_CAUSE。(R/W)

I2C_SLAVE_SCL_STRETCH_CLR 置位此位，清除 CL 时钟拉伸。(WT)

I2C_SLAVE_BYTE_ACK_CTL_EN 使能从机控制 ACK 电平。(R/W)

I2C_SLAVE_BYTE_ACK_LVL I2C_SLAVE_BYTE_ACK_CTL_EN 置 1 时，设置 ACK 的电平值。(R/W)

Register 27.19. I2C_SR_REG (0x0008)



I2C_RESP_REC 主机模式或从机模式下接收的 ACK 电平值。0：ACK；1：NACK。(RO)

I2C_SLAVE_RW 从机模式下，0：主机向从机写入数据；1：主机读取从机数据。(RO)

I2C_ARB_LOST I2C 控制器不控制 SCL 线时，该寄存器变为 1。(RO)

I2C_BUS_BUSY 0：I2C 总线处于空闲状态；1：I2C 总线正在传输数据。(RO)

I2C_SLAVE_ADDRESSED 配置成 I2C 从机、且主机发送地址与从机地址匹配时，该位翻转为高电平。(RO)

I2C_RXFIFO_CNT 该字段为需发送数据的字节数。(RO)

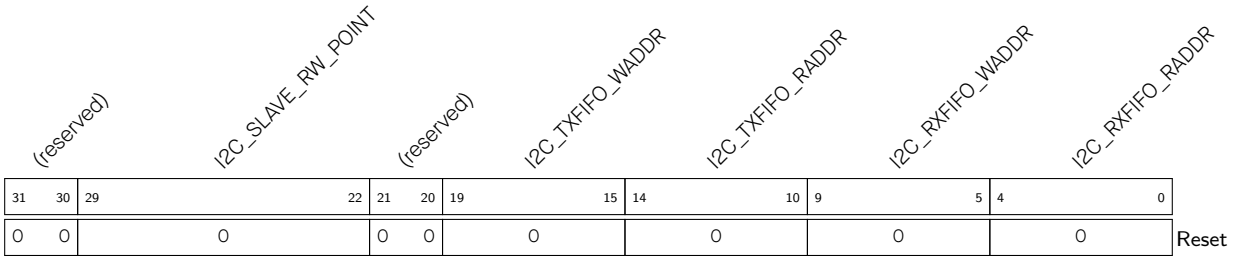
I2C_STRETCH_CAUSE 从机模式下 SCL 时钟拉伸的原因。0：主机开始读取数据时拉伸 SCL 时钟；1：从机模式下 I2C TX FIFO 读空时拉伸 SCL 时钟；2：从机模式下 I2C RX FIFO 写满时拉伸 SCL 时钟。(RO)

I2C_TXFIFO_CNT 该字段存储 RAM 接收数据的字节数。(RO)

I2C_SCL_MAIN_STATE_LAST 该字段为 I2C 控制器状态机的状态。0：空闲；1：地址偏移；2：ACK 地址；3：接收数据；4：发送数据；5：发送 ACK；6：等待 ACK (RO)

I2C_SCL_STATE_LAST 该字段为生成 SCL 的状态机状态。0：空闲状态；1：开始；2：下降沿；3：低电平；4：上升沿；5：高电平；6：停止 (RO)

Register 27.20. I2C_FIFO_ST_REG (0x0014)



I2C_RXFIFO_RADDR APB 总线读 RX FIFO 的偏移地址。(RO)

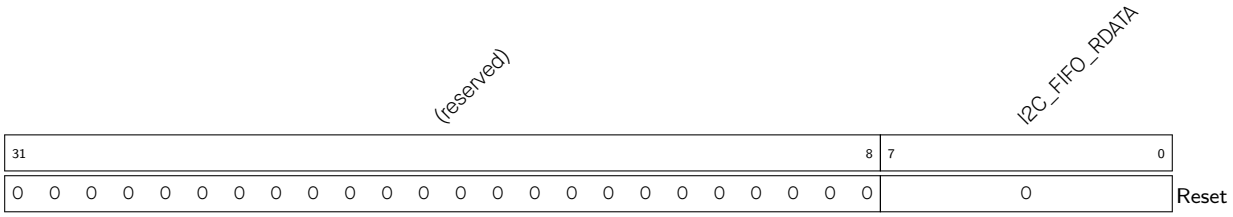
I2C_RXFIFO_WADDR I2C 控制器接收数据和写 RX FIFO 的偏移地址。(RO)

I2C_TXFIFO_RADDR I2C 控制器读 TX FIFO 的偏移地址。(RO)

I2C_TXFIFO_WADDR APB 总线写 TX FIFO 的偏移地址。(RO)

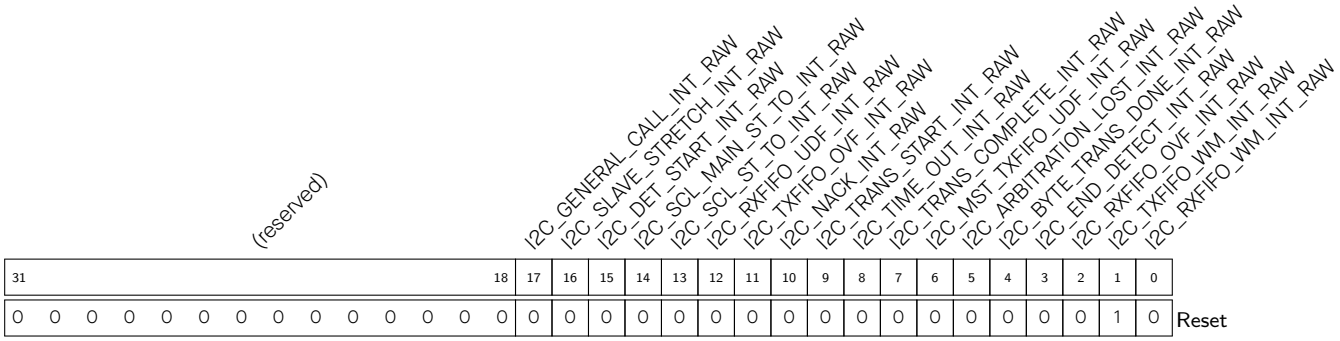
I2C_SLAVE_RW_POINT 从机模式下接收的数据。(RO)

Register 27.21. I2C_DATA_REG (0x001C)



I2C_FIFO_RDATA 用于读取 RX FIFO 的数据，或向 TX FIFO 写数据。(R/W)

Register 27.22. I2C_INT_RAW_REG (0x0020)



I2C_RXFIFO_WM_INT_RAW I2C_RXFIFO_WM_INT 的原始中断位。(R/SS/WTC)

I2C_TXFIFO_WM_INT_RAW I2C_TXFIFO_WM_INT 的原始中断位。(R/SS/WTC)

I2C_RXFIFO_OVF_INT_RAW I2C_RXFIFO_OVF_INT 的原始中断位。(R/SS/WTC)

I2C_END_DETECT_INT_RAW I2C_END_DETECT_INT 的原始中断位。(R/SS/WTC)

I2C_BYTE_TRANS_DONE_INT_RAW I2C_BYTE_TRANS_DONE_INT 的原始中断位。(R/SS/WTC)

I2C_ARBITRATION_LOST_INT_RAW I2C_ARBITRATION_LOST_INT 的原始中断位。(R/SS/WTC)

I2C_MST_TXFIFO_UDF_INT_RAW I2C_MST_TXFIFO_UDF_INT 的原始中断位。(R/SS/WTC)

I2C_TRANS_COMPLETE_INT_RAW I2C_TRANS_COMPLETE_INT 的原始中断位。(R/SS/WTC)

I2C_TIME_OUT_INT_RAW I2C_TIME_OUT_INT 的原始中断位。(R/SS/WTC)

I2C_TRANS_START_INT_RAW I2C_TRANS_START_INT 的原始中断位。(R/SS/WTC)

I2C_NACK_INT_RAW I2C_NACK_INT 的原始中断位。(R/SS/WTC)

I2C_TXFIFO_OVF_INT_RAW I2C_TXFIFO_OVF_INT 的原始中断位。(R/SS/WTC)

I2C_RXFIFO_UDF_INT_RAW I2C_RXFIFO_UDF_INT 的原始中断位。(R/SS/WTC)

I2C_SCL_ST_TO_INT_RAW I2C_SCL_ST_TO_INT 的原始中断位。(R/SS/WTC)

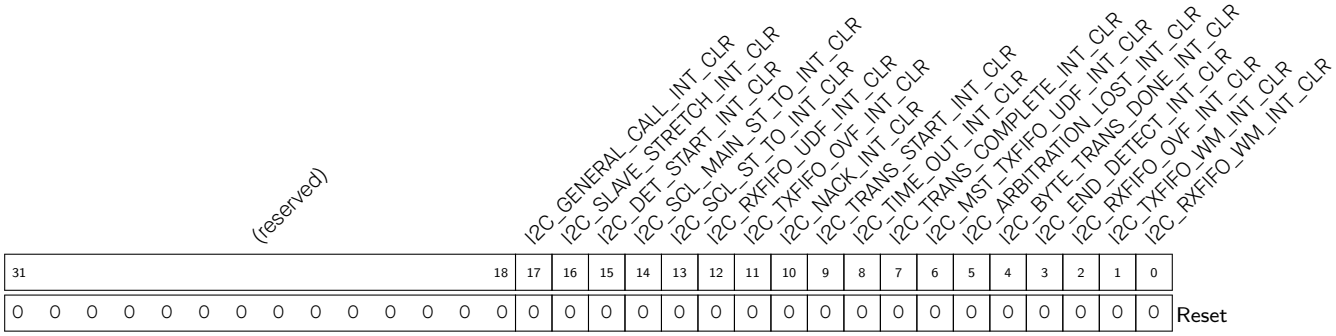
I2C_SCL_MAIN_ST_TO_INT_RAW I2C_SCL_MAIN_ST_TO_INT 的原始中断位。(R/SS/WTC)

I2C_DET_START_INT_RAW I2C_DET_START_INT 的原始中断位。(R/SS/WTC)

I2C_SLAVE_STRETCH_INT_RAW I2C_SLAVE_STRETCH_INT 的原始中断位。(R/SS/WTC)

I2C_GENERAL_CALL_INT_RAW I2C_GENARAL_CALL_INT 的原始中断位。(R/SS/WTC)

Register 27.23. I2C_INT_CLR_REG (0x0024)



I2C_RXFIFO_WM_INT_CLR 置位此位，清除 I2C_RXFIFO_WM_INT 中断。(WT)

I2C_TXFIFO_WM_INT_CLR 置位此位，清除 I2C_TXFIFO_WM_INT 中断。(WT)

I2C_RXFIFO_OVF_INT_CLR 置位此位，清除 I2C_RXFIFO_OVF_INT 中断。(WT)

I2C_END_DETECT_INT_CLR 置位此位，清除 I2C_END_DETECT_INT 中断。(WT)

I2C_BYTE_TRANS_DONE_INT_CLR 置位此位，清除 I2C_BYTE_TRANS_DONE_INT 中断。(WT)

I2C_ARBITRATION_LOST_INT_CLR 置位此位，清除 I2C_ARBITRATION_LOST_INT 中断。(WT)

I2C_MST_TXFIFO_UDF_INT_CLR 置位此位，清除 I2C_MST_TXFIFO_UDF_INT 中断。(WT)

I2C_TRANS_COMPLETE_INT_CLR 置位此位，清除 I2C_TRANS_COMPLETE_INT 中断。(WT)

I2C_TIME_OUT_INT_CLR 置位此位，清除 I2C_TIME_OUT_INT 中断。(WT)

I2C_TRANS_START_INT_CLR 置位此位，清除 I2C_TRANS_START_INT 中断。(WT)

I2C_NACK_INT_CLR 置位此位，清除 I2C_NACK_INT 中断。(WT)

I2C_TXFIFO_OVF_INT_CLR 置位此位，清除 I2C_TXFIFO_OVF_INT 中断。(WT)

I2C_RXFIFO_UDF_INT_CLR 置位此位，清除 I2C_RXFIFO_UDF_INT 中断。(WT)

I2C_SCL_ST_TO_INT_CLR 置位此位，清除 I2C_SCL_ST_TO_INT 中断。(WT)

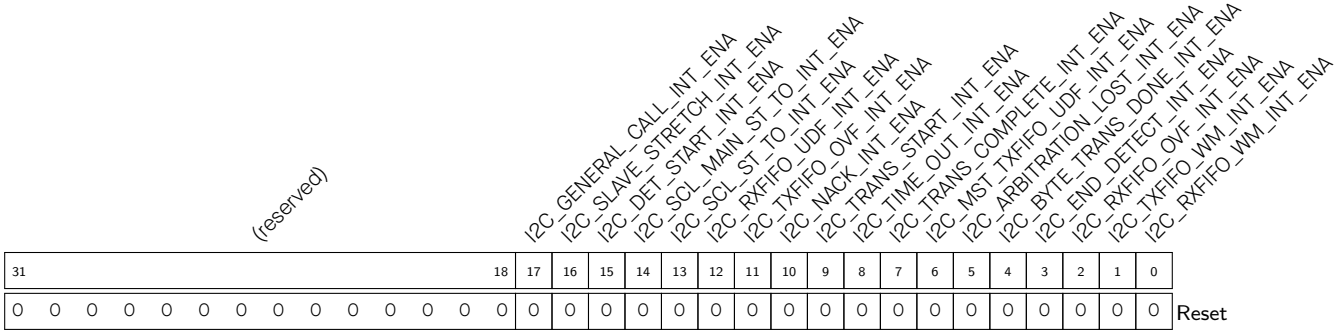
I2C_SCL_MAIN_ST_TO_INT_CLR 置位此位，清除 I2C_SCL_MAIN_ST_TO_INT 中断。(WT)

I2C_DET_START_INT_CLR 置位此位，清除 I2C_DET_START_INT 中断。(WT)

I2C_SLAVE_STRETCH_INT_CLR 置位此位，清除 I2C_SLAVE_STRETCH_INT 中断。(WT)

I2C_GENERAL_CALL_INT_CLR 置位此位，清除 I2C_GENARAL_CALL_INT 中断。(WT)

Register 27.24. I2C_INT_ENA_REG (0x0028)



I2C_RXFIFO_WM_INT_ENA I2C_RXFIFO_WM_INT 的使能位。(R/W)

I2C_TXFIFO_WM_INT_ENA I2C_TXFIFO_WM_INT 的使能位。(R/W)

I2C_RXFIFO_OVF_INT_ENA I2C_RXFIFO_OVF_INT 的使能位。(R/W)

I2C_END_DETECT_INT_ENA I2C_END_DETECT_INT 的使能位。(R/W)

I2C_BYTE_TRANS_DONE_INT_ENA I2C_BYTE_TRANS_DONE_INT 的使能位。(R/W)

I2C_ARBITRATION_LOST_INT_ENA I2C_ARBITRATION_LOST_INT 的使能位。(R/W)

I2C_MST_TXFIFO_UDF_INT_ENA I2C_MST_TXFIFO_UDF_INT 的使能位。(R/W)

I2C_TRANS_COMPLETE_INT_ENA I2C_TRANS_COMPLETE_INT 的使能位。(R/W)

I2C_TIME_OUT_INT_ENA I2C_TIME_OUT_INT 的使能位。(R/W)

I2C_TRANS_START_INT_ENA I2C_TRANS_START_INT 的使能位。(R/W)

I2C_NACK_INT_ENA I2C_NACK_INT 的使能位。(R/W)

I2C_TXFIFO_OVF_INT_ENA I2C_TXFIFO_OVF_INT 的使能位。(R/W)

I2C_RXFIFO_UDF_INT_ENA I2C_RXFIFO_UDF_INT 的使能位。(R/W)

I2C_SCL_ST_TO_INT_ENA I2C_SCL_ST_TO_INT 的使能位。(R/W)

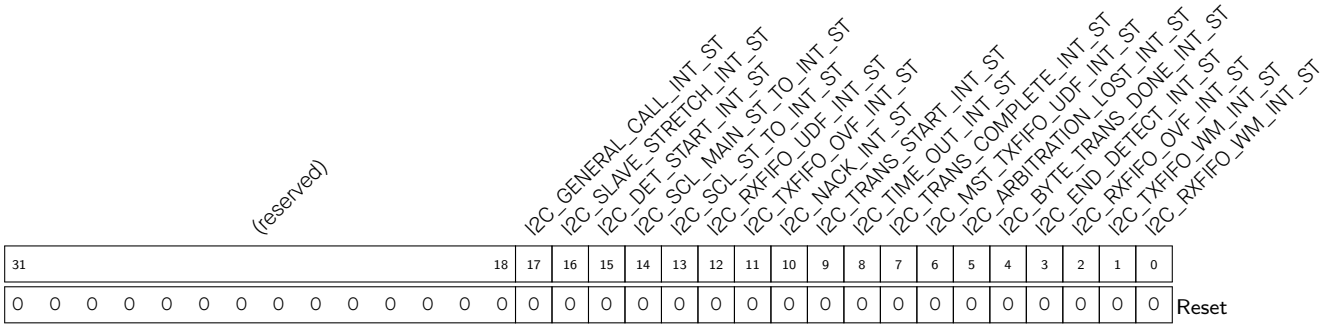
I2C_SCL_MAIN_ST_TO_INT_ENA I2C_SCL_MAIN_ST_TO_INT 的使能位。(R/W)

I2C_DET_START_INT_ENA I2C_DET_START_INT 的使能位。(R/W)

I2C_SLAVE_STRETCH_INT_ENA I2C_SLAVE_STRETCH_INT 的使能位。(R/W)

I2C_GENERAL_CALL_INT_ENA I2C_GENARAL_CALL_INT 的使能位。(R/W)

Register 27.25. I2C_INT_STATUS_REG (0x002C)



I2C_RXFIFO_WM_INT_ST I2C_RXFIFO_WM_INT 的屏蔽状态位。(RO)

I2C_TXFIFO_WM_INT_ST I2C_TXFIFO_WM_INT 的屏蔽状态位。(RO)

I2C_RXFIFO_OVF_INT_ST I2C_RXFIFO_OVF_INT 的屏蔽状态位。(RO)

I2C_END_DETECT_INT_ST I2C_END_DETECT_INT 的屏蔽状态位。(RO)

I2C_BYTE_TRANS_DONE_INT_ST I2C_BYTE_TRANS_DONE_INT 的屏蔽状态位。(RO)

I2C_ARBITRATION_LOST_INT_ST I2C_ARBITRATION_LOST_INT 的屏蔽状态位。(RO)

I2C_MST_TXFIFO_UDF_INT_ST I2C_MST_TXFIFO_UDF_INT 的屏蔽状态位。(RO)

I2C_TRANS_COMPLETE_INT_ST I2C_TRANS_COMPLETE_INT 的屏蔽状态位。(RO)

I2C_TIME_OUT_INT_ST I2C_TIME_OUT_INT 的屏蔽状态位。(RO)

I2C_TRANS_START_INT_ST I2C_TRANS_START_INT 的屏蔽状态位。(RO)

I2C_NACK_INT_ST I2C_NACK_INT 的屏蔽状态位。(RO)

I2C_TXFIFO_OVF_INT_ST I2C_TXFIFO_OVF_INT 的屏蔽状态位。(RO)

I2C_RXFIFO_UDF_INT_ST I2C_RXFIFO_UDF_INT 的屏蔽状态位。(RO)

I2C_SCL_ST_TO_INT_ST I2C_SCL_ST_TO_INT 的屏蔽状态位。(RO)

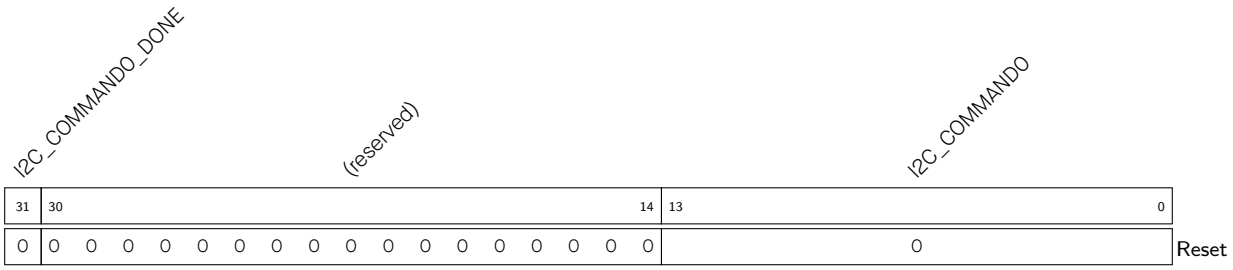
I2C_SCL_MAIN_ST_TO_INT_ST I2C_SCL_MAIN_ST_TO_INT 的屏蔽状态位。(RO)

I2C_DET_START_INT_ST I2C_DET_START_INT 的屏蔽状态位。(RO)

I2C_SLAVE_STRETCH_INT_ST I2C_SLAVE_STRETCH_INT 的屏蔽状态位。(RO)

I2C_GENERAL_CALL_INT_ST I2C_GENARAL_CALL_INT 的屏蔽状态位。(RO)

Register 27.26. I2C_COMD0_REG (0x0058)



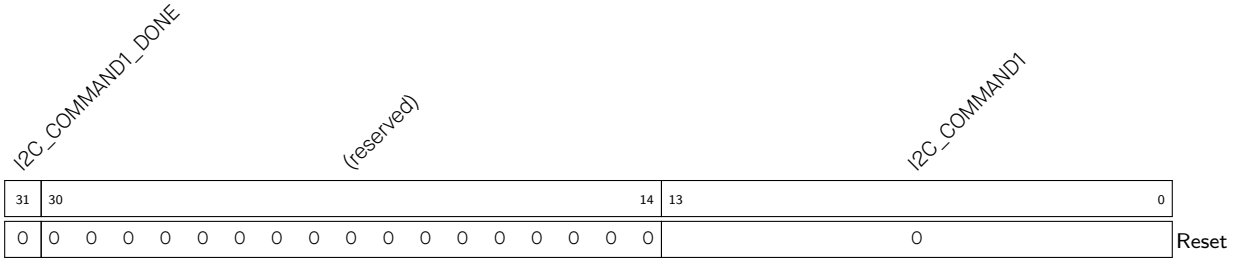
I2C_COMMAND0 命令寄存器 0 的内容。该命令包括三个部分：

- op_code 为命令，1：WRITE；2：STOP；3：READ；4：END；6：RSTART。
- byte_num 表示需发送或接收的字节数。
- ack_check_en、ack_exp 和 ack 用于控制 ACK 位。更多信息详见章节 27.4.9。

(R/W)

I2C_COMMAND0_DONE 在 I2C 主机模式下完成命令 0 时，该位翻转为高电平。(R/W/SS)

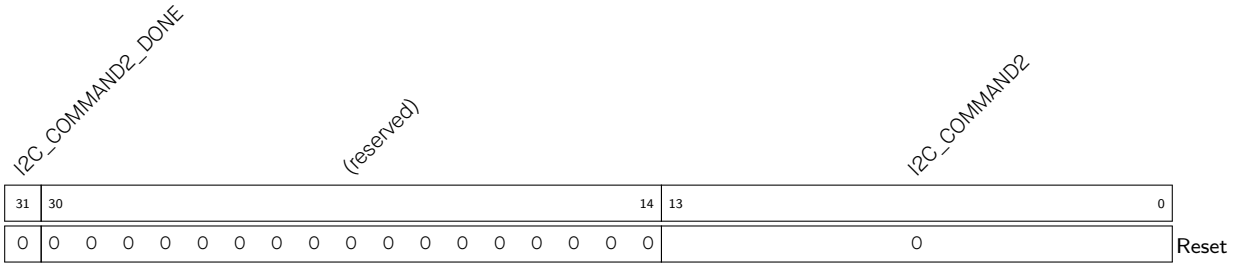
Register 27.27. I2C_COMD1_REG (0x005C)



I2C_COMMAND1 命令寄存器 1 的内容，同 I2C_COMMAND0。(R/W)

I2C_COMMAND1_DONE 在 I2C 主机模式下完成命令 1 时，该位翻转为高电平。(R/W/SS)

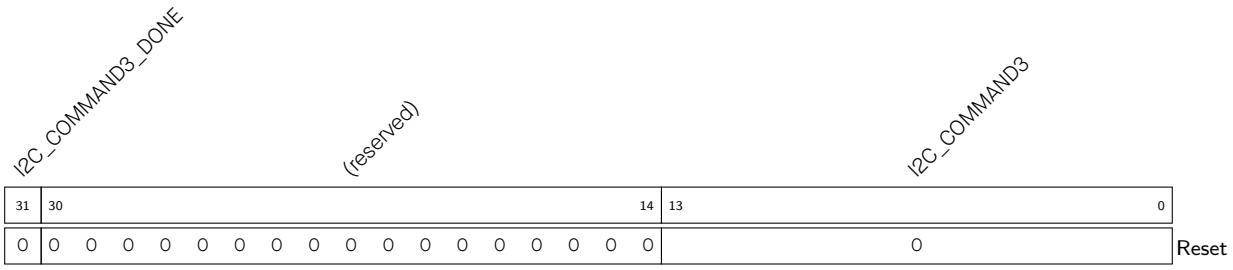
Register 27.28. I2C_COMD2_REG (0x0060)



I2C_COMMAND2 命令寄存器 2 的内容，同 I2C_COMMAND0。(R/W)

I2C_COMMAND2_DONE 在 I2C 主机模式下完成命令 2 时，该位翻转为高电平。(R/W/SS)

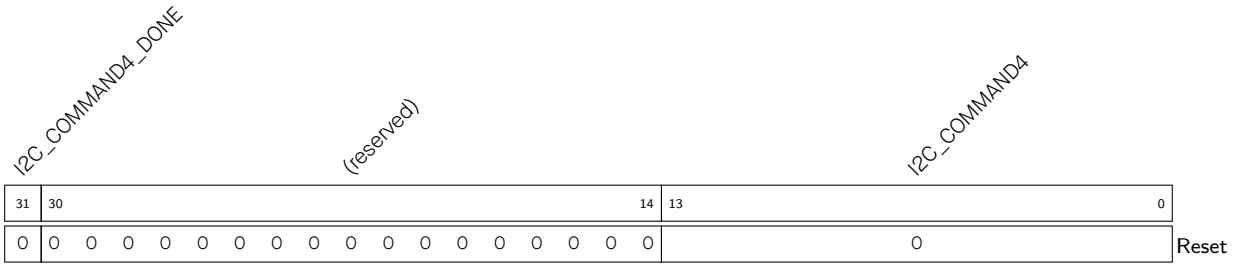
Register 27.29. I2C_COMD3_REG (0x0064)



I2C_COMMAND3 命令寄存器 3 的内容，同 I2C_COMMAND0。(R/W)

I2C_COMMAND3_DONE 在 I2C 主机模式下完成命令 3 时，该位翻转为高电平。(R/W/SS)

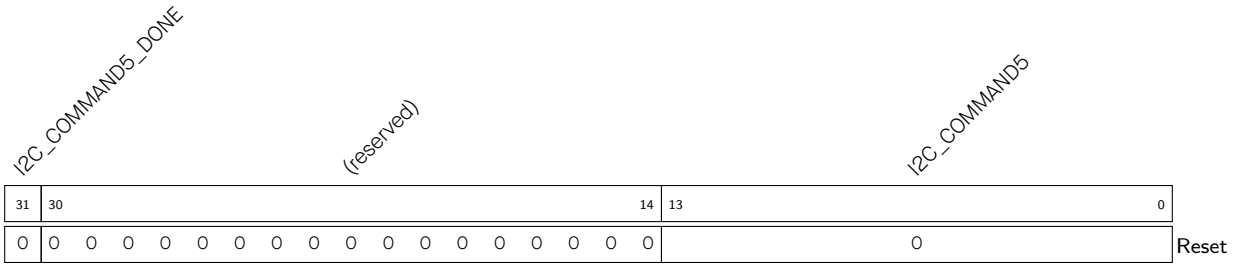
Register 27.30. I2C_COMD4_REG (0x0068)



I2C_COMMAND4 命令寄存器 4 的内容，同 I2C_COMMAND0。(R/W)

I2C_COMMAND4_DONE 在 I2C 主机模式下完成命令 4 时，该位翻转为高电平。(R/W/SS)

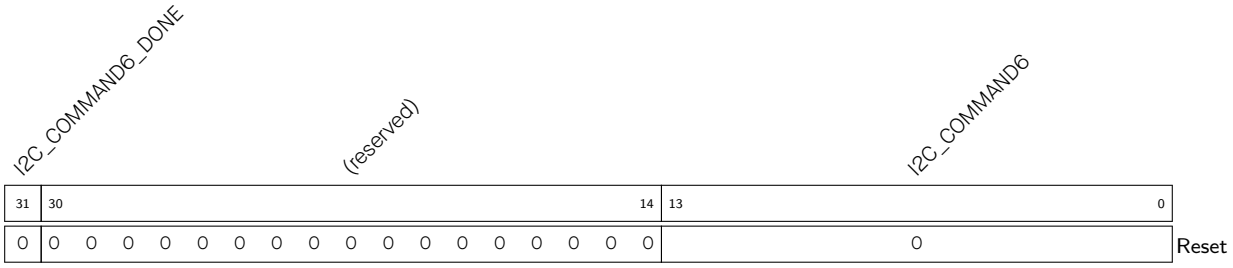
Register 27.31. I2C_COMD5_REG (0x006C)



I2C_COMMAND5 命令寄存器 5 的内容，同 I2C_COMMAND0。(R/W)

I2C_COMMAND5_DONE 在 I2C 主机模式下完成命令 5 时，该位翻转为高电平。(R/W/SS)

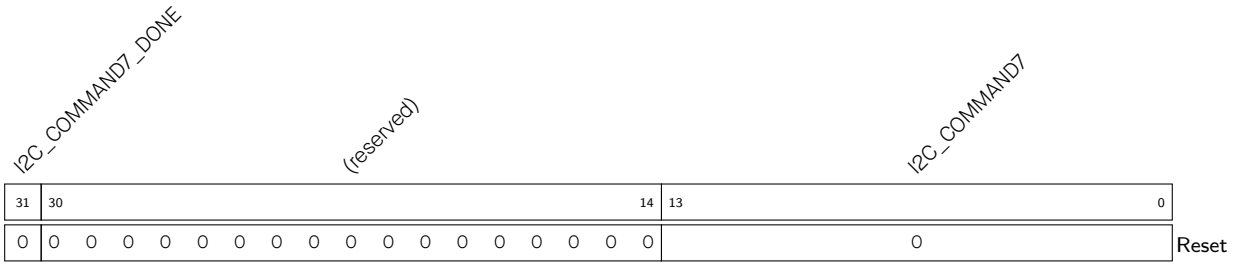
Register 27.32. I2C_COMD6_REG (0x0070)



I2C_COMMAND6 命令寄存器 6 的内容，同 I2C_COMMAND0。(R/W)

I2C_COMMAND6_DONE 在 I2C 主机模式下完成命令 6 时，该位翻转为高电平。(R/W/SS)

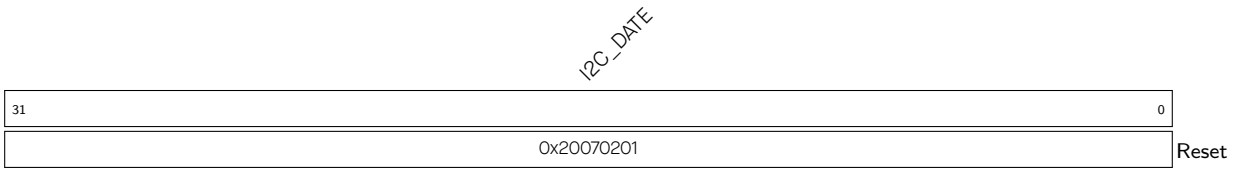
Register 27.33. I2C_COMD7_REG (0x0074)



I2C_COMMAND7 命令寄存器 7 的内容，同 I2C_COMMAND0。(R/W)

I2C_COMMAND7_DONE 在 I2C 主机模式下完成命令 7 时，该位翻转为高电平。(R/W/SS)

Register 27.34. I2C_DATE_REG (0x00F8)



I2C_DATE 版本控制寄存器。(R/W)

第 28 章

I2S 控制器 (I2S)

28.1 概述

ESP32-S3 内置两个 I2S 接口（即 I2S*0* 和 I2S*1*），为多媒体应用，尤其是为数字音频应用提供了灵活的数据通信接口。

I2S 标准总线定义了三种信号：串行时钟信号 BCK、字选择信号 WS 和串行数据信号 SD。一个基本的 I2S 数据总线有一个主机和一个从机。主机和从机的角色在通信过程中保持不变。ESP32-S3 的 I2S 模块包含独立的发送单元和接收单元，能够保证优良的通信性能。

说明：

本章节提供的信息适用于 I2S*0* 和 I2S*1*。除非另有说明，本章节中 I2S 及 I2S*n* 均指代 I2S*0* 和 I2S*1*。

28.2 术语

为了更好地说明 I2S*n* 的功能，本章使用了以下术语。

主机模式 I2S*n* 作为主机，BCK/WS 向外部输出，向从机发送或从其接收数据。

从机模式 I2S*n* 作为从机，BCK/WS 从外部输入，从主机接收或向其发送数据。

全双工 主机与从机之间的发送线和接收线各自独立，发送数据和接收数据同时进行。

半双工 主机和从机只能有一方先发送数据，另一方接收数据。发送数据和接收数据不能同时进行。

TDM RX 模式 利用时分复用方式接收脉冲编码调制 (PCM) 数据，并将其通过 DMA 存入储存器的模式。信号线包括 BCK、WS 和 DATA。可以接收最多 16 个通道的数据。通过用户配置，可支持 TDM Philips 格式、TDM MSB 对齐格式、TDM PCM 格式等。

PDM RX 模式 接收脉冲密度调制 (PDM) 数据，并将其通过 DMA 存入储存器的模式。信号线包括 WS 和 DATA。通过用户配置，可支持 PDM 标准格式等。

TDM TX 模式 通过 DMA 从储存器中取得脉冲编码调制 (PCM) 数据，并利用时分复用方式将其发送的模式。信号线包括 BCK、WS 和 DATA，可以发送最多 16 个通道的数据。通过用户配置，可支持 TDM Philips 格式、TDM MSB 对齐格式、TDM PCM 格式等。

PDM TX 模式 通过 DMA 从储存器中取得脉冲密度调制 (PDM) 数据，并将其发送的模式。信号线包括 WS 和 DATA。通过用户配置，可支持 PDM 标准格式等。

PCM-to-PDM TX 模式（仅对 I2S*0* 有效） 通过 DMA 从储存器中取得脉冲编码调制 (PCM) 数据，将其转换为脉冲密度调制 (PDM) 数据，并将其发送的**主机**模式。信号线包括 WS 和 DATA。通过用户配置，可支持 PDM 标准格式等。

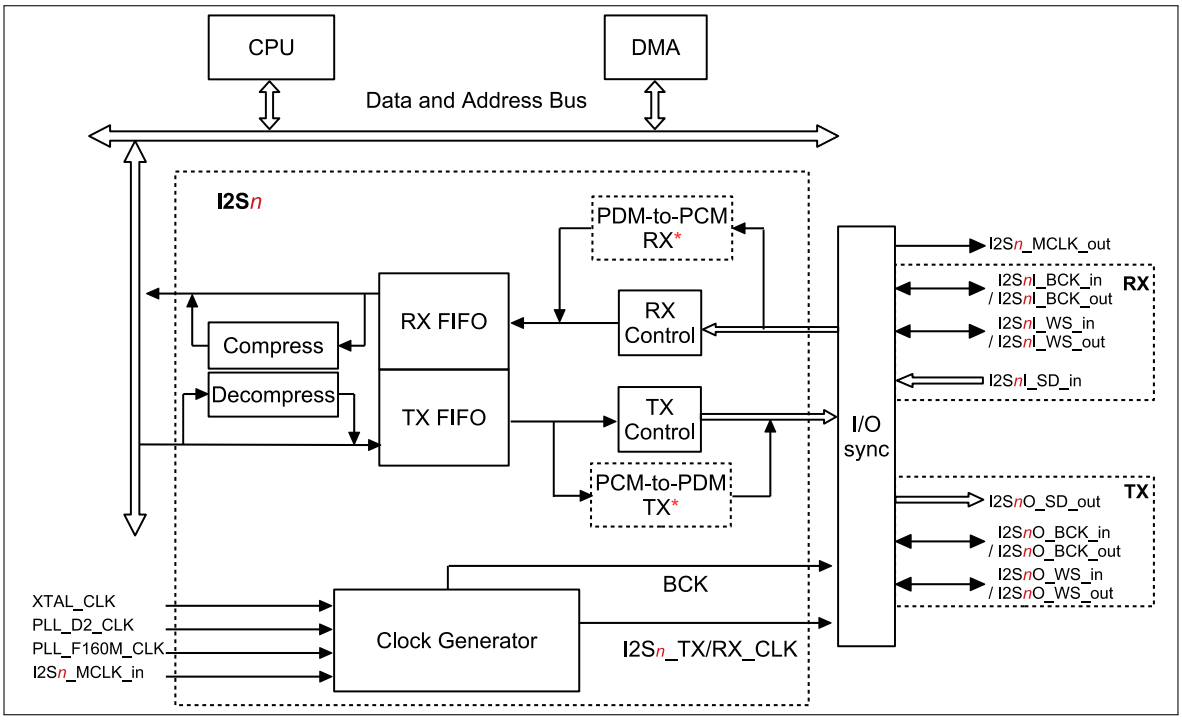
PDM-to-PCM RX 模式（仅对 I2S*0* 有效） 接收脉冲密度调制 (PDM) 数据，将其转换为脉冲编码调制 (PCM) 数据，并将其通过 DMA 存入储存器的**主机**模式或**从机**模式。信号线包括 WS 和 DATA。通过用户配置，可支持 PDM 标准格式等。

28.3 特性

I2S*n* 具有以下特性：

- 支持主机模式和从机模式
- 支持全双工和半双工通信
- TX 模块和 RX 模块相互独立
- TX 模块和 RX 模块可独立工作或同时工作
- 支持多种音频标准：
  - TDM Philips 标准
  - TDM MSB 对齐标准
  - TDM PCM 标准
  - PDM 标准
- 支持多种 TX/RX 模式
  - TDM TX 模式
  - TDM RX 模式
  - PDM TX 模式
  - PDM RX 模式
  - PCM-to-PDM TX 模式（仅对 I2S*0* 有效）
  - PDM-to-PCM RX 模式（仅对 I2S*0* 有效）
- 高精度采样时钟可配置
- 支持如下采样频率：8 kHz、16 kHz、32 kHz、44.1 kHz、48 kHz、88.2 kHz、96 kHz、128 kHz 和 192 kHz。注意：不支持从机 192 kHz 32 位模式。
- 支持 8/16/24/32 位数据通信
- 支持 DMA
- 支持 I2S 接口中断

28.4 系统架构



注：图中 PDM-to-PCM RX 和 PCM-to-PDM TX 仅适用于 I2S*0*。

图 28.4-1. ESP32-S3 I2S 系统框图

图 28.4-1 是 ESP32-S3 I2S*n* 模块的结构框图。ESP32-S3 I2S*n* 模块包含：

- 独立的发送单元 (TX control)
- 独立的接收单元 (RX control)
- 输入输出时序调节单元 (I/O sync)
- 时钟分频器 (Clock Generator)
- 64 x 32-bit TX FIFO
- 64 x 32-bit RX FIFO
- 压缩/解压缩模块 (Compress/Decompress)

ESP32-S3 I2S*n* 模块支持 GDMA，可访问内部存储器和外部存储器，更多信息见章节 3 *通用 DMA 控制器 (GDMA)*。

发送单元和接收单元各自有一组三线接口，分别为串行时钟线 BCK，字选择线 WS 和串行数据线 SD。其中，发送单元的 SD 线固定为输出，接收单元的 SD 线固定为接收。发送单元和接收单元的 BCK 和 WS 信号线均可配置为主机输出模式或从机输入模式。

图 28.4-1 右侧为 I2S*n* 模块的信号总线。RX 和 TX 模块的信号命名规则为：I2S*n*A_B_C，例如 I2S*n*I_BCK_in。其中：

- “A” 表示 I2S 模块的数据总线的方向
  - “I” 表示输入

  - “O” 表示输出
- “B” 表示信号功能，包括：
  - 串行时钟信号 (BCK)
  - 字选择信号 (WS)
  - 串行数据信号 (SD)
- “C” 表示该信号的方向
  - “in” 表示该信号输入 I2S*n* 模块
  - “out” 表示该信号自 I2S*n* 模块输出

I2S*n* 各信号的具体描述见表 28.4-1。

表 28.4-1. 模块信号描述

信号*	方向	功能
I2S*n*I_BCK_in	输入	从机模式下，输入 BCK 信号，用于 RX 模块
I2S*n*I_BCK_out	输出	主机模式下，输出 BCK 信号，用于 RX 模块
I2S*n*I_WS_in	输入	从机模式下，输入 WS 信号，用于 RX 模块
I2S*n*I_WS_out	输出	主机模式下，输出 WS 信号，用于 RX 模块
I2S*n*I_Data_in	输入	RX 模块的串行输入数据线
I2S*n*O_Data_out	输出	TX 模块的串行输出数据线
I2S*n*O_BCK_in	输入	从机模式下，输入 BCK 信号，用于 TX 模块
I2S*n*O_BCK_out	输出	主机模式下，输出 BCK 信号，用于 TX 模块
I2S*n*O_WS_in	输入	从机模式下，输入 WS 信号，用于 TX 模块
I2S*n*O_WS_out	输出	主机模式下，输出 WS 信号，用于 TX 模块
I2S*n*_MCLK_in	输入	从机模式下，来自外部芯片的时钟源
I2S*n*_MCLK_out	输出	主机模式下，作为外部芯片的时钟源
I2S*0*I_Data1_in	输入	RX 模块在 PDM-to-PCM 模式下的串行输入数据线
I2S*0*I_Data2_in	输入	RX 模块在 PDM-to-PCM 模式下的串行输入数据线
I2S*0*I_Data3_in	输入	RX 模块在 PDM-to-PCM 模式下的串行输入数据线
I2S*0*O_Data1_out	输出	TX 模块在 PCM-to-PDM 模式下的串行输出数据线

* I2S*n* 的所有信号均需要经过 GPIO 交换矩阵映射到芯片的管脚。更多信息请参考章节 6 *IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)*。

28.5 I2S*n* 模块支持的音频协议

ESP32-S3 I2S*n* 模块支持多种音频标准，包括 TDM Philips 标准、TDM MSB 对齐标准、TDM PCM 标准以及 PDM 标准。

用户可通过配置以下寄存器，选择所需的音频标准：

- I2S*n*_TX/RX_TDM_EN
  - 0：禁用 TDM 模式
  - 1：选择 TDM 模式

- I2S*n*_TX/RX_PDM_EN
  - 0：禁用 PDM 模式
  - 1：选择 PDM 模式

  I2S*n*_TX/RX_MSB_SHIFT
  - 0：配置 WS 信号和 SD 信号同时开始变化，即选择 MSB 对齐标准
  - 1：配置 WS 信号先于 SD 信号一个 BCK 时钟周期开始变化，即选择 Philips 标准或 PCM 标准
- I2S*n*_TX/RX_PCM_BYPASS
  - 0：选择 PCM 标准
  - 1：禁用 PCM 标准

28.5.1 TDM Philips 标准模式

在 Philips 标准下，在 BCK 的下降沿，WS 信号先于 SD 信号一个 BCK 时钟周期开始变化，即 WS 信号从当前通道数据的第一个位之前的一个时钟开始有效，并在当前通道数据发送结束前一个 BCK 时钟周期开始变化。SD 信号线上首先传输音频数据的最高位。

与 Philips 标准相比，TDM Philips 标准支持更多的通道，见图 28.5-1。

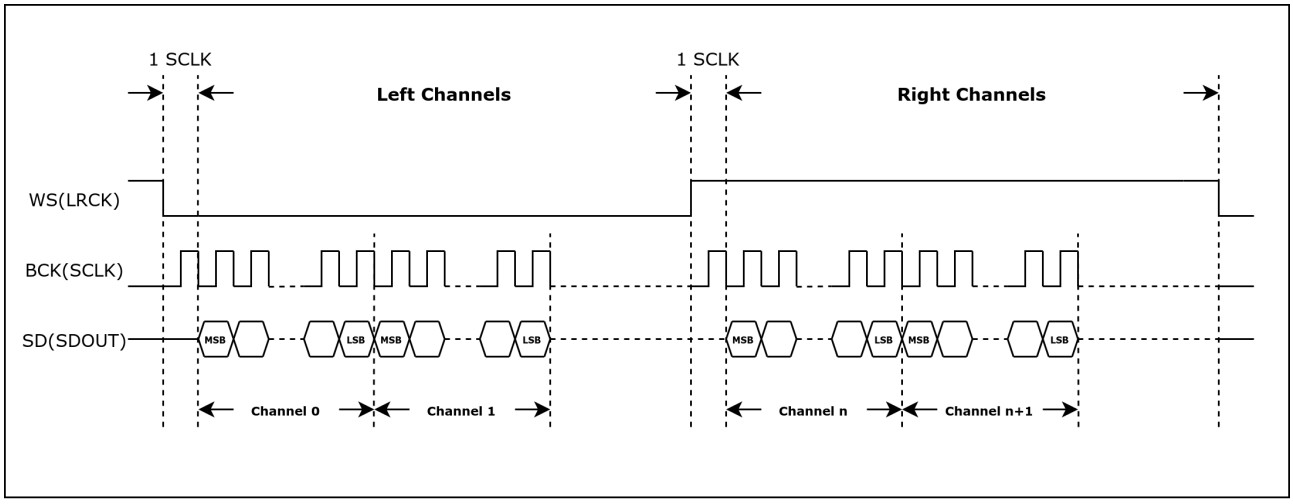


图 28.5-1. 时序图 – TDM Philips 标准

28.5.2 TDM MSB 对齐标准模式

MSB 对齐标准下，在 BCK 下降沿，WS 信号和 SD 信号同时变化。WS 持续到当前通道数据发送结束，SD 信号线上首先传输音频数据的最高位。

与 MSB 对齐标准相比，TDM MSB 对齐标准支持更多的通道，见图 28.5-2。

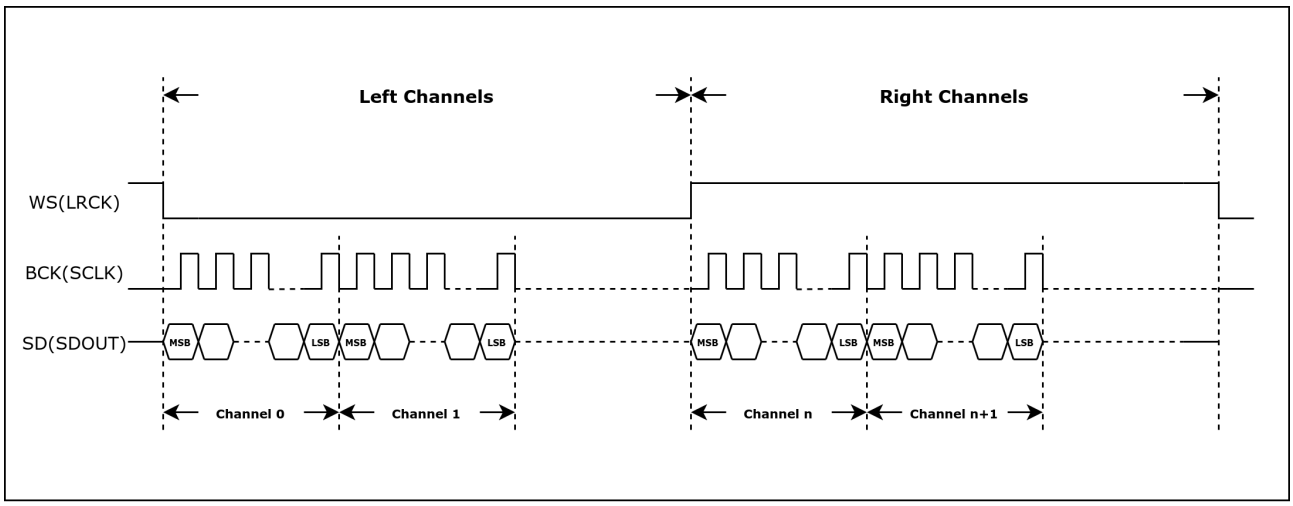


图 28.5-2. 时序图 – TDM MSB 对齐标准

28.5.3 TDM PCM 标准模式

在 PCM 标准的短帧同步模式下，在 BCK 的下降沿，WS 信号先于 SD 信号一个 BCK 时钟周期开始变化，即 WS 信号从当前通道数据的第一个位之前的一个时钟开始有效，并持续一个 BCK 时钟周期。SD 信号线上首先传输音频数据的最高位。

与 PCM 标准相比，TDM PCM 标准支持更多的通道，见图 28.5-3 所示。

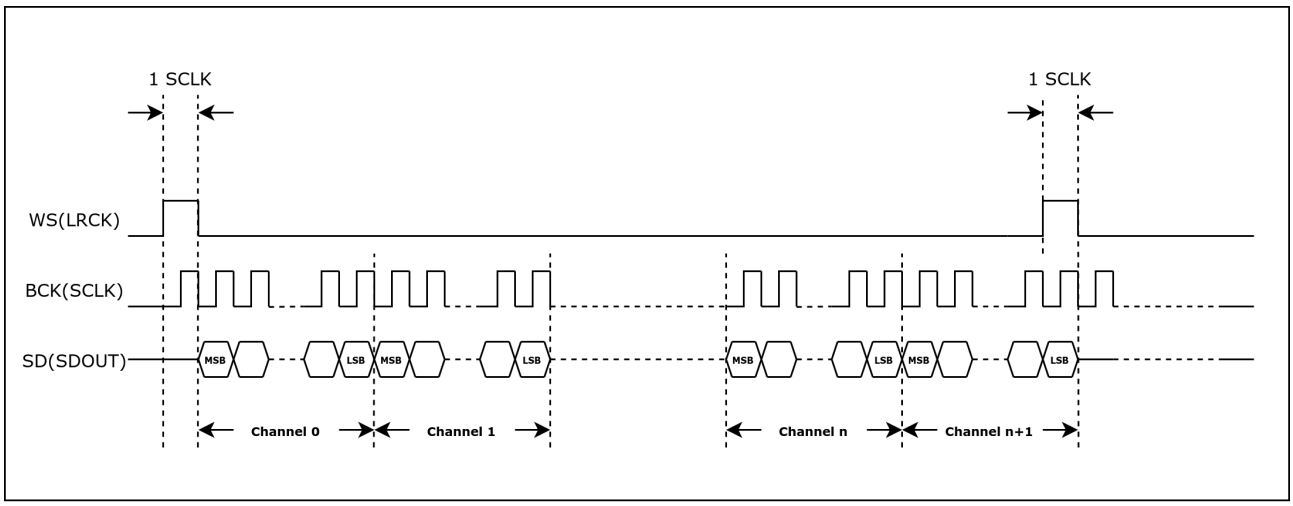


图 28.5-3. 时序图 – TDM PCM 标准

28.5.4 PDM 标准模式

如图 28.5-4 所示，在 PDM 标准下，WS 代表左/右声道，在 BCK 的下降沿，WS 与 SD 同时变化。WS 在数据发送过程中持续变化，WS 的高低对应两个声道。

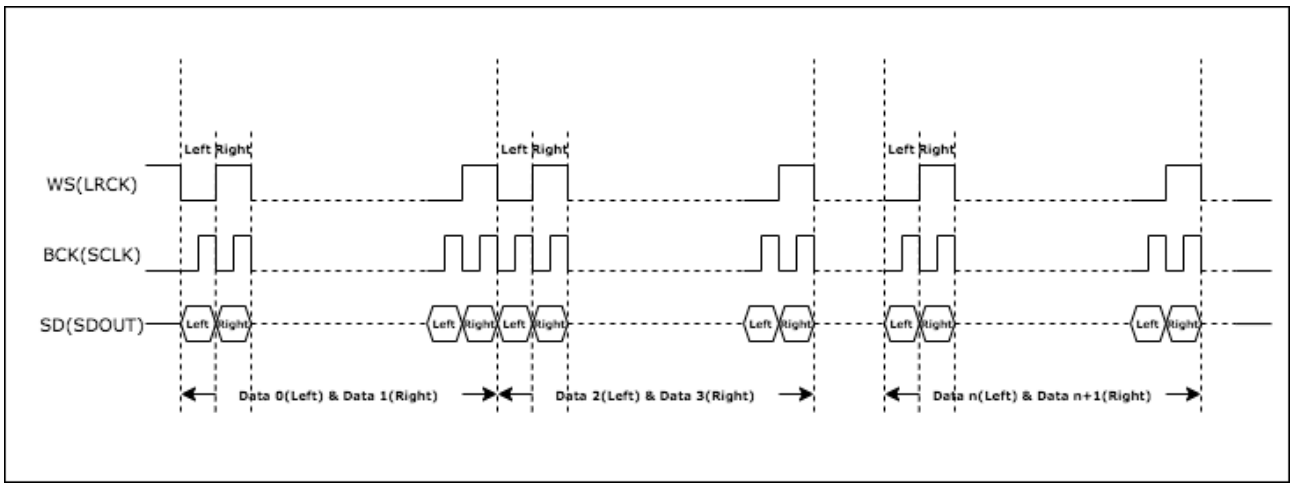


图 28.5-4. 时序图 – PDM 标准

28.6 TX/RX 模块时钟

I2S*n*_TX/RX_CLK 作为 I2S*n* TX/RX 模块的主时钟，可由以下时钟分频所得（见图 28.6-1）：

- 40 MHz XTAL_CLK
- 160 MHz PLL_F160M_CLK
- 240 MHz PLL_D2_CLK
- 或外部输入时钟：I2S*n*_MCLK_in

I2S*n*_TX/RX_CLK_SEL 用于选择 I2S*n* TX/RX 的时钟源，I2S*n*_TX/RX_CLK_ACTIVE 用于使能或者关闭 I2S*n* TX/RX 模块的时钟源。

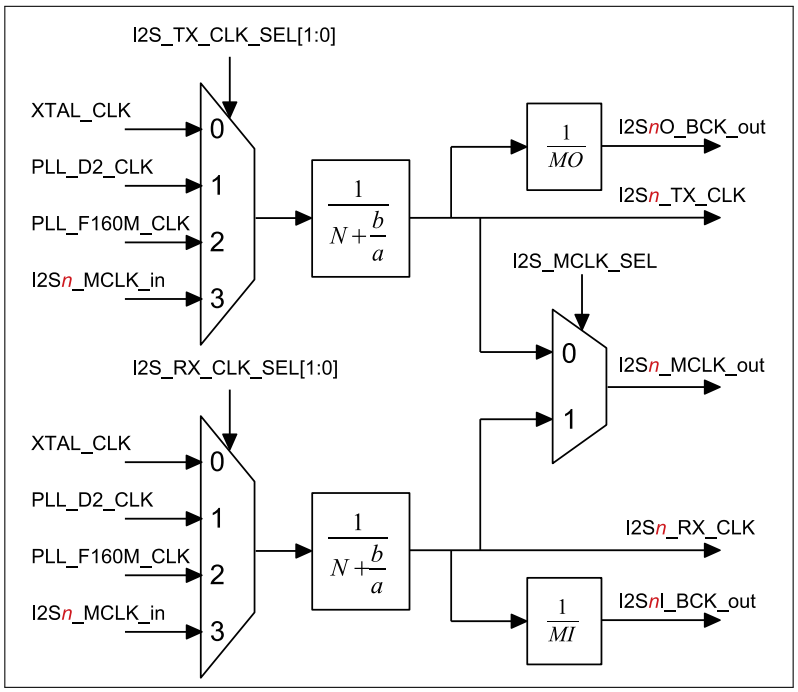


图 28.6-1. I2S*n* 时钟

I2S*n*_TX/RX_CLK 的频率 f_{I2Sn_TX/RX_CLK} 与分频器时钟源频率 $f_{I2Sn\_CLK\_S}$ 间的关系如下：

$$f_{I2Sn\_TX/RX\_CLK} = \frac{f_{I2Sn\_CLK\_S}}{N + \frac{b}{a}}$$

其中，2 =< N <= 256，N 对应为 I2S*n*_TX/RX_CLKM_CONF_REG 寄存器中 I2S*n*_TX/RX_CLKM_DIV_NUM 的值，具体为：

- I2S*n*_TX/RX_CLKM_DIV_NUM = 0 时，N = 256；
- I2S*n*_TX/RX_CLKM_DIV_NUM = 1 时，N = 2；
- I2S*n*_TX/RX_CLKM_DIV_NUM 为其它值时，N = I2S*n*_TX/RX_CLKM_DIV_NUM 的值。

分数部分分频系数 a、b 唯一对应系数 x、y、z、yn1。系数对应公式为：

- 当 b <= $\frac{a}{2}$ 时，yn1 = 0，x = $floor([\frac{a}{b}]) - 1$，y = a%b，z = b；
- 当 b > $\frac{a}{2}$ 时，yn1 = 1，x = $floor([\frac{a}{a-b}]) - 1$，y = a%(a - b)，z = a - b；

系数 x、y、z、yn1 在 I2S*n*_TX/RX_CLKM_DIV_X、I2S*n*_TX/RX_CLKM_DIV_Y、I2S*n*_TX/RX_CLKM_DIV_Z、I2S*n*_TX/RXCLKM_DIV_YN1 中配置。
对于整数分频，I2S*n*_TX/RX_CLKM_DIV_X 和 I2S*n*_TX/RX_CLKM_DIV_Z 清零，I2S*n*_TX/RX_CLKM_DIV_Y 置为 1。

> **说明：**
> 使用小数分频功能可能会产生时钟抖动。

I2S*n* TX/RX 模块的串行时钟 BCK 再由 I2S*n*_TX/RX_CLK 分频获得，如图 28.6-1 所示。

在主机发送模式下，I2S*n* TX 模块的串行时钟 BCK 为 I2S*n*O_BCK_out 信号，由 I2S*n*_TX_CLK 分频获得。即：

$$f_{I2SnO\_BCK\_out} = \frac{f_{I2Sn\_TX\_CLK}}{MO}$$

其中，MO 的值为 I2S*n*_TX_BCK_DIV_NUM 的值 + 1，即：

$$MO = I2Sn\_TX\_BCK\_DIV\_NUM + 1$$

注意：

I2S*n*_TX_BCK_DIV_NUM 不可配置为 1。

在主机接收模式下，I2S*n* RX 模块的串行时钟 BCK 为 I2S*n*I_BCK_out 信号，由 I2S*n*_RX_CLK 分频获得。即：

$$f_{I2SnI\_BCK\_out} = \frac{f_{I2Sn\_RX\_CLK}}{MI}$$

其中 MI 对应 I2S*n*_RX_BCK_DIV_NUM 的值 + 1，即：

$$MI = I2Sn\_RX\_BCK\_DIV\_NUM + 1$$

注意：

- I2S*n*_RX_BCK_DIV_NUM 不可配置为 1；
- 当模块处于从机模式时，必须保证 f_{I2Sn_TX/RX_CLK} >= 8 * f_{BCK}。另外模块可以输出 I2S*n*_MCLK_out 作为外部设备的主时钟。

28.7 I2S*n* 模块复位

I2S*n* 模块中各个单元以及 FIFO 可通过配置相关位进行复位：

- I2S*n* TX/RX 单元：可配置 I2S*n*_TX_RESET 和 I2S*n*_RX_RESET 位进行复位；
- I2S*n* TX/RX FIFO：可配置 I2S*n*_TX_FIFO_RESET 和 I2S*n*_RX_FIFO_RESET 位进行复位。

注意：在模块和 FIFO 复位之前，需要先配置 I2S*n* 模块时钟。

28.8 I2S*n* 主/从机模式

ESP32-S3 I2S*n* 模块可作为主机或从机。用户可配置 I2S*n*_TX_SLAVE_MOD 和 I2S*n*_RX_SLAVE_MOD 选择需要的模式。

- I2S*n*_TX_SLAVE_MOD
  - 0：主机发送模式
  - 1：从机发送模式
- I2S*n*_RX_SLAVE_MOD
  - 0：主机接收模式
  - 1：从机接收模式

28.8.1 主/从机发送模式

- 主机发送模式
  - 置位 I2S*n*_TX_START 位启动一次发送操作。
  - 置位该位，发送单元会一直输出时钟信号和串行数据。
  - 置位 I2S*n*_TX_STOP_EN 时，如果 FIFO 中的数据全部发送完毕，则主机停止发送数据。
  - 清零 I2S*n*_TX_STOP_EN 时，如果 FIFO 中的数据全部发送完毕，并且没有新数据填入，发送模块将一直发送最后一帧数据。
  - 当 I2S*n*_TX_START 位被清零时，主机停止发送数据。
- 从机发送模式
  - 置位 I2S*n*_TX_START。
  - 发送单元等待主机 BCK 时钟，来启动发送操作。
  - 置位 I2S*n*_TX_STOP_EN 时，如果 FIFO 中的数据全部发送完毕，则从机发送数据一直为零，直到主机停止发送 BCK 时钟为止。

  - 清零 I2S*n*_TX_STOP_EN 时，如果 FIFO 中的数据全部发送完毕，并且没有新数据填入，发送单元将一直发送最后一帧数据。
  - 当 I2S*n*_TX_START 位被清零时，从机发送数据一直为零，直到主机停止发送 BCK 时钟为止。

28.8.2 主/从机接收模式

- 主机接收模式
  - 置位 I2S*n*_RX_START 启动一次接收操作。
  - 接收单元会一直输出时钟信号，并对输入数据进行采样。
  - 清零 I2S*n*_RX_START，接收单元停止接收数据。
- 从机接收模式
  - 置位 I2S*n*_RX_START。
  - 等待主机 BCK 时钟，来启动接收操作。
  - 清零 I2S*n*_RX_START，接收单元停止接收数据。

28.9 发送数据

说明：
本小节以及后续小节所述的配置，均需要通过置位 I2S*n*_TX_UPDATE 的方式来进行更新，从而将 I2S*n* TX 寄存器数据从 APB 时钟域同步到 I2S*n* TX 时钟域。详细配置见第 28.11.1 小节。

ESP32-S3 I2S*n* 发送数据时，从 DMA 读取数据，经过数据格式控制和通道模式控制，从外设输出信号输出对应数据。

28.9.1 数据格式控制

数据格式控制分为三个阶段：

- 第一阶段从内存中读出有效数据并写入 TX FIFO；
- 第二阶段将待发送数据从 TX FIFO 中读出，并进行输出数据模式转换；
- 第三阶段，将待发送数据转换为串行数据流输出。

28.9.1.1 通道有效数据位宽

I2S*n*_TX_BITS_MOD 和 I2S*n*_TX_24_FILL_EN 决定了每个通道的有效数据位宽，其可取值和对应的有效数据位宽如下表：

表 28.9-1. 通道有效数据位宽控制

通道有效数据位宽	I2S*n*_TX_BITS_MOD	I2S*n*_TX_24_FILL_EN
32	31	x[1]
	23	1
24	23	0
16	15	x
8	7	x

[1] 该值被忽略。

28.9.1.2 通道有效数据字节序

I2S*n* 通过 DMA 读取数据之后，I2S*n*_TX_BIG_ENDIAN 用于控制从 DMA 读取数据的字节序。下表描述了不同通道有效数据位宽下，该寄存器对读取数据的控制。

表 28.9-2. 通道有效数据字节序控制

通道有效数据位宽	原始数据	控制后数据	I2S*n*_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

28.9.1.3 A 率/μ 率压缩/解压缩

ESP32-S3 I2S*n* 对排列好字节序的有效数据会按照 32-bit（缺省高位补 0*）的方式进行 A 率/μ 率压缩/解压缩。

> **说明：**
> 缺省高位补 0，即如果有效数据位宽小于 32，则待压缩/解压缩数据的 [31: 有效位宽] 位自动填充 0。

配置 I2S*n*_TX_PCM_BYPASS 为：

- 0，则不进行压缩/解压缩
- 1，则进行压缩/解压缩

配置 I2S*n*_TX_PCM_CONF 为：

- 0，A 律解压缩
- 1，A 律压缩
- 2，μ 律解压缩
- 3，μ 律压缩

至此，数据格式控制的第一阶段完成。

28.9.1.4 通道发送数据位宽

ESP32-S3 I2S*n* 中，I2S*n*_TX_TDM_CHAN_BITS 决定了每个通道发送数据的位宽。

- 当每个通道发送数据的位宽大于有效数据位宽时，会在剩余的位置补充 0。此时，配置 I2S*n*_TX_LEFT_ALIGN 为：
  - 0，有效数据位于发送数据低位。
  - 1，有效数据位于发送数据高位。
- 当每个通道发送数据的位宽小于有效数据位宽时，每次发送数据仅发送低位有效数据部分，高位部分将被舍弃。

至此，数据格式控制的第二阶段完成。

28.9.1.5 通道数据比特顺序

ESP32-S3 I2S*n* 通道数据比特顺序由 I2S*n*_TX_BIT_ORDER 控制：

- 0：不调换有效数据比特顺序；
- 1：调换有效数据比特顺序。

至此，数据格式控制部分全部完成。经过数据格式控制的数据将从高位向低位依次发送。图 28.9-1 所示即为一次完整的 TX 数据格式控制过程。

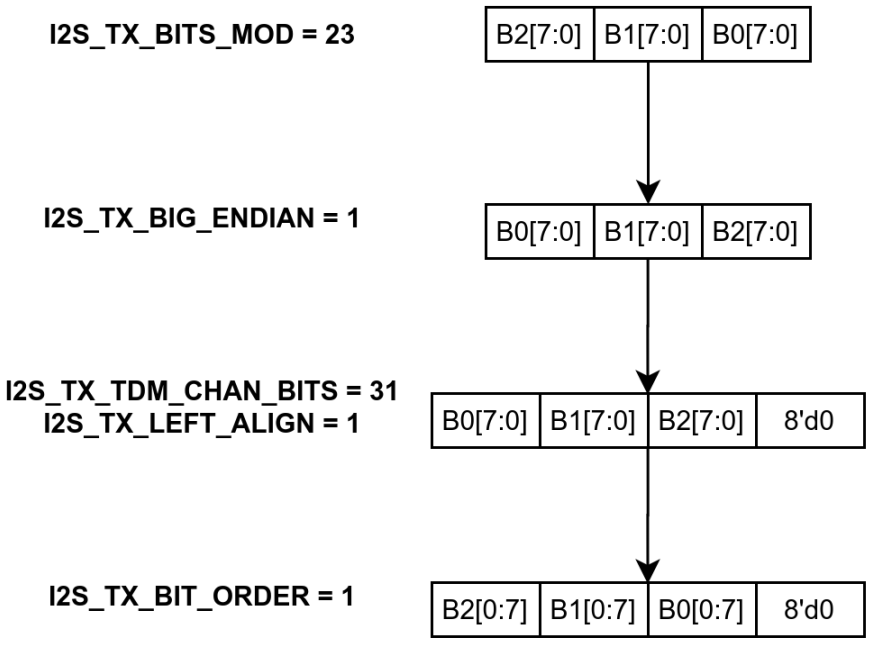


图 28.9-1. TX 数据格式控制

28.9.2 通道模式控制

ESP32-S3 I2S*n* 支持 TDM 和 PDM 两种发送模式。置位 I2S*n*_TX_TDM_EN 则为 TDM 发送模式，置位 I2S*n*_TX_PDM_EN 则为 PDM 发送模式。

说明：

- I2S*n*_TX_TDM_EN 和 I2S*n*_TX_PDM_EN 不能同时置位或同时清零。
- 将 I2S*n* 模块设置为 TDM 双通道模式，可实现控制大多数 I2S 双声道编解码器。

28.9.2.1 TDM 模式下 I2S*n* 通道模式

在 TDM 模式下，I2S*n* 支持最多 16 通道数据输出。发送通道数由 I2S*n*_TX_TDM_TOT_CHAN_NUM 控制。例如，配置 I2S*n*_TX_TDM_TOT_CHAN_NUM 为 5，则六个通道（通道 0 ~ 5）将用于发送数据，见图 28.9-2。

在发送数据的通道中，如果其对应的 I2S*n*_TX_TDM_CHAN*n*_EN 为：

- 1，则该通道发送通道数据。
- 0，则该通道发送数据由 I2S*n*_TX_CHAN_EQUAL 控制，当该值为：
  - 1，则发送上个通道的数据。
  - 0，则发送 I2S*n*_SINGLE_DATA 的值。

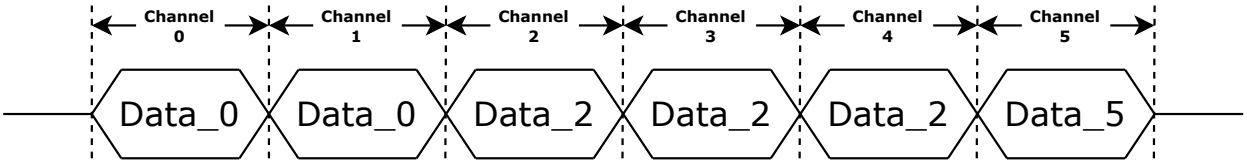
当 I2S*n* 处于主机 TDM 模式下，WS 信号由 I2S*n*_TX_WS_IDLE_POL 和 I2S*n*_TX_TDM_WS_WIDTH 控制。其中，I2S*n*_TX_WS_IDLE_POL 的值为 WS 信号的默认电平，I2S*n*_TX_TDM_WS_WIDTH 的值为在发送所有通道数据的过程中，WS 为默认电平的周期数。另外，I2S*n*_TX_HALF_SAMPLE_BITS 的值乘以 2，即为一个 WS 周期对应的 BCK 周期数。

TDM 通道配置示例

在本示例中，寄存器的配置如下：

- 配置 I2S_TX_TDM_CHAN_NUM 为 5，即选择使用通道 0 ~ 5 进行数据发送。
- 配置 I2S_TX_CHAN_EQUAL 为 1，即如果相应通道的 I2S_TX_TDM_CHAN*n*_EN 被置位，则该通道将发送上一通道的数据。*n* = 0 ~ 5。
- 配置 I2S_TX_TDM_CHAN0/2/5_EN 为 1，即这些通道将用于发送通道数据。
- 配置 I2S_TX_TDM_CHAN1/3/4_EN 为 0，则这些通道将发送上一个通道的数据。

配置完成后，则数据将按下图方式进行发送。



I2S_TX_TDM_CHAN_NUM = 5; I2S_TX_CHAN_EQUAL = 1;
I2S_TX_TDM_CHAN0_EN = 1; I2S_TX_TDM_CHAN1_EN = 0; I2S_TX_TDM_CHAN2_EN = 1;
I2S_TX_TDM_CHAN3_EN = 0; I2S_TX_TDM_CHAN4_EN = 0; I2S_TX_TDM_CHAN5_EN = 1;

图 28.9-2. TDM 通道控制

28.9.2.2 PDM 模式下 I2S*n* 通道模式

在 PDM 模式下，从 DMA 取得数据的过程由 I2S*n*_TX_MONO 和 I2S*n*_TX_MONO_FST_VLD 控制，具体如下表。请根据内存中存储的数据为单/双通道数据来配置该寄存器。

表 28.9-3. PDM 模式下 I2S*n* 取数逻辑

取数逻辑	模式	I2S*n*_TX_MONO	I2S*n*_TX_MONO_FST_VLD
每个 WS 沿都向 DMA 发起取数请求	双声道	0	x
只在 WS 后半周期向 DMA 发起取数请求	单声道	1	0
只在 WS 前半周期向 DMA 发起取数请求	单声道	1	1

在 PDM 模式下，I2S*n* 通道数据由 I2S*n*_TX_CHAN_MOD 和 I2S*n*_TX_WS_IDLE_POL 控制，具体如下表。

表 28.9-4. PDM 模式下 I2S*n* 通道模式控制

模式	左声道	右声道	模式控制字段[1]	声道选择位[2]
双声道	发送左通道数据	发送右通道数据	0	x
单声道	发送左通道数据	发送左通道数据	1	0
	发送右通道数据	发送右通道数据	1	1
	发送右通道数据	发送右通道数据	2	0
	发送左通道数据	发送左通道数据	2	1
	发送 I2S*n*_SINGLE_DATA 的值	发送右通道数据	3	0
	发送左通道数据	发送 I2S*n*_SINGLE_DATA 的值	3	1
	发送左通道数据	发送 I2S*n*_SINGLE_DATA 的值	4	0
	发送 I2S*n*_SINGLE_DATA 的值	发送右通道数据	4	1

[1] I2S*n*_TX_CHAN_MOD
[2] I2S*n*_TX_WS_IDLE_POL

当 I2S*n* 处于主机 PDM 模式下，WS 信号的默认电平由 I2S*n*_TX_WS_IDLE_POL 控制，WS 信号频率为 BCK 信号频率的一半。请参考 28.6 小节配置 BCK 信号的方式配置 WS 信号，见图 28.9-3。

I2S*O* 模块还支持 PCM 转 PDM 输出模式，可以将 DMA 中的 PCM 数据转为 PDM 数据，并按照 PDM 信号格式输出，配置 I2S*O*_PCM2PDM_CONV_EN 启动该模式。PCM 转 PDM 输出模式的寄存器配置如下：

- 配置一线 PDM 输出格式或一/二线 DAC 输出格式，具体如下表。

表 28.9-5. PCM 转 PDM 输出模式

通道输出格式	I2S*O*_TX_PDM_DAC_MODE_EN	I2S*O*_TX_PDM_DAC_2OUT_EN
一线 PDM 输出格式[1]	0	x
一线 DAC 输出格式[2]	1	0
二线 DAC 输出格式	1	1

[1] 此处定义的 PDM 输出格式是指一个 WS 周期发送两个通道的 SD 数据。
[2] 此处定义的 DAC 输出格式是指一个 WS 周期发送一个通道的 SD 数据。

- 配置采样频率和上采样率。
  ESP32-S3 I2S*O* PCM 转 PDM 模式下，PDM 时钟频率即为 BCK 时钟频率。采样频率 ($f_{Sampling}$) 与 BCK 时钟频率的关系如下：

$$f_{Sampling} = \frac{f_{BCK}}{OSR}$$

其中，上采样率 OSR 和 I2S*0*_TX_PDM_SINC_OSR2 的关系为：

$$\text{OSR} = \text{I2S0\_TX\_PDM\_SINC\_OSR2} \times 64$$

采样频率 $f_Sampling$ 和 I2S*0*_TX_PDM_FS 的对应关系为：

$$f_{\text{Sampling}} = \text{I2S0\_TX\_PDM\_FS} \times 100$$

请根据需要的采样频率、上采样率以及 PDM 时钟频率进行寄存器配置。

PDM 通道配置示例

在本示例中，寄存器的配置如下：

- 配置 I2S*n*_TX_CHAN_MOD 为 2，即选择单声道模式。
- 配置 I2S*n*_TX_WS_IDLE_POL 为 1，则左声道和右声道均发送左通道数据。

配置完成后，则数据将按照下图方式进行发送。

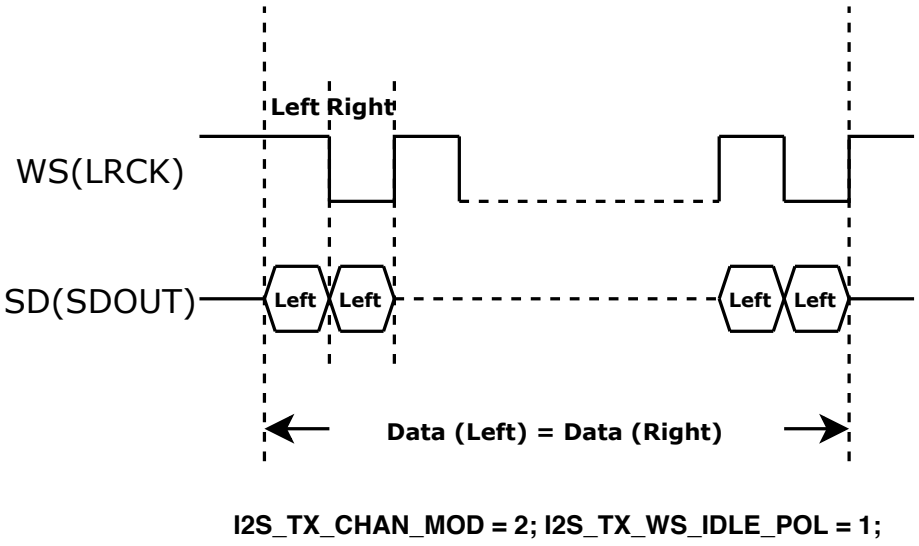


图 28.9-3. PDM 通道控制

28.10 接收数据

> **说明：**
> 本小节以及后续小节所述的配置，均需要通过置位 I2S*n*_RX_UPDATE 的方式来进行更新，从而将 I2S*n* RX 寄存器数据从 APB 时钟域同步到 I2S*n* RX 时钟域。详细配置见第 28.11.2 小节。

I2S*n* 接收数据时，从外设接口读取数据，经过通道模式控制和数据格式控制，通过 DMA 输入数据至内存。

28.10.1 通道模式控制

ESP32-S3 I2S*n* 支持 TDM 和 PDM 两种接收模式。置位 I2S*n*_RX_TDM_EN 则为 TDM 接收模式，置位 I2S*n*_RX_PDM_EN 则为 PDM 接收模式。

注意：I2S*n*_RX_TDM_EN 和 I2S*n*_RX_PDM_EN 不能同时置位或同时清零。

28.10.1.1 TDM 模式下 I2S*n* 通道模式

在 TDM 模式下，I2S*n* 支持最多 16 通道数据输入。接收通道数由 I2S*n*_RX_TDM_TOT_CHAN_NUM 控制。例如，配置 I2S*n*_RX_TDM_TOT_CHAN_NUM 为 5，则通道 0 ~ 5 接收数据。

在接收数据的通道中，如果其对应的 I2S*n*_RX_TDM_CHAN*n*_EN 为：

- 1，则该通道数据有效，会被存入 RX FIFO。
- 0，则该通道数据无效，不会存入 RX FIFO。

当 I2S*n* 处于主机 TDM 模式下，WS 信号由 I2S*n*_RX_WS_IDLE_POL 和 I2S*n*_RX_TDM_WS_WIDTH 控制。其中，I2S*n*_RX_WS_IDLE_POL 的值为 WS 信号的默认电平，I2S*n*_RX_TDM_WS_WIDTH 的值为在接收所有通道的过程中，WS 为默认电平的周期数。另外，I2S*n*_RX_HALF_SAMPLE_BITS 的值乘以 2，即为一个 WS 周期对应的 BCK 周期数。

28.10.1.2 PDM 模式下 I2S*n* 通道模式

在 PDM 模式下，I2S*n* 将通道中的串行数据转换为待输入数据。

当 I2S*n* 处于主机 PDM 模式下，WS 信号的默认电平由 I2S*n*_RX_WS_IDLE_POL 控制，WS 信号频率为 BCK 信号频率的一半，请参考 28.6 小节配置 BCK 信号的方式配置 WS 信号。注意，在 PDM 接收模式下，请配置 I2S*n*_RX_HALF_SAMPLE_BITS 的值与 I2S*n*_RX_BITS_MOD 的值相同。

I2S*0* 模块还支持 PDM 转 PCM 输入模式，可以将接收的 PDM 数据转为 PCM 数据，再进行数据控制，配置 I2S*0*_RX_PDM2PCM_EN 启动该模式。PDM 转 PCM 输入模式的寄存器配置如下：

- 配置采样频率和下采样率。
  ESP32-S3 I2S*0* PDM 转 PCM 模式下，PDM 时钟频率与主从机模式的关系如下：
  - 主机模式：PDM 时钟频率即为 BCK 时钟频率。
  - 从机模式：PDM 时钟频率由外部提供。

  采样频率 ($f_{Sampling}$) 与 PDM 时钟频率的关系如下：

  $$f_{Sampling} = \frac{f_{PDM}}{DSR}$$

  其中，下采样率 DSR 和 I2S*0*_RX_PDM_SINC_DSR_16_EN 的关系为：

  $$DSR = I2S0\_RX\_PDM\_SINC\_DSR\_16\_EN \times 64$$

  请根据需要的主从机模式、采样频率以及下采样率进行寄存器配置。
- 配置有效通道。
  ESP32-S3 I2S PDM 转 PCM 模式下，最多支持 8 个通道的输入信号，寄存器配置以及对应通道如下表：

表 28.10-1. PDM 转 PCM 输入模式

输入数据信号	通道	使能寄存器
I2S0I_Data_in	左声道	I2S0_RX_TDM_PDM_CHAN0_EN
	右声道	I2S0_RX_TDM_PDM_CHAN1_EN
I2S0I1_Data_in	左声道	I2S0_RX_TDM_PDM_CHAN2_EN
	右声道	I2S0_RX_TDM_PDM_CHAN3_EN
I2S0I2_Data_in	左声道	I2S0_RX_TDM_PDM_CHAN4_EN
	右声道	I2S0_RX_TDM_PDM_CHAN5_EN
I2S0I3_Data_in	左声道	I2S0_RX_TDM_PDM_CHAN6_EN
	右声道	I2S0_RX_TDM_PDM_CHAN7_EN

28.10.2 数据格式控制

数据格式控制分为两个阶段：

- 第一阶段从串行数据流输入转换为待输入数据，并存入 RX FIFO；
- 第二阶段将待输入数据从 RX FIFO 中读出，并进行输入数据模式转换。

28.10.2.1 通道数据比特顺序

ESP32-S3 I2S*n* 通道数据将从高位到低位依次存入待输入数据，并由 I2S*n*_RX_BIT_ORDER 控制是否调换比特序，当该值为：

- 0：不调换待输入数据的比特序；
- 1：调换待输入数据的比特序。

至此，数据格式控制的第一阶段完成，经过比特顺序控制的待输入数据被存入 RX FIFO。

28.10.2.2 通道储存数据位宽

I2S*n*_RX_BITS_MOD 和 I2S*n*_RX_24_FILL_EN 决定了每个通道的储存数据位宽，其可取值和对应的储存数据位宽如下表：

表 28.10-2. 通道储存数据位宽控制

通道储存数据位宽	I2S*n*_RX_BITS_MOD	I2S*n*_RX_24_FILL_EN
32	31	x
	23	1
24	23	0
16	15	x
8	7	x

28.10.2.3 通道接收数据位宽

ESP32-S3 I2S*n* 中，I2S*n*_RX_TDM_CHAN_BITS 决定了每个通道接收数据的位宽。

- 当每个通道储存数据位宽小于接收数据的位宽时，接收时仅会在接收数据中取储存位宽作为储存数据。此时，配置 I2S*n*_RX_LEFT_ALIGN 为：

  - 0，储存数据位于接收数据低位。
  - 1，储存数据位于接收数据高位。

- 当每个通道接收数据的位宽小于储存数据位宽时，会将接收数据高位补 0，作为储存数据。

28.10.2.4 通道储存数据字节序

通道接收数据经过截取/补全后，成为了待储存数据，I2S*n*_RX_BIG_ENDIAN 用于控制待储存数据的字节序。下表描述了不同通道储存数据位宽下，该寄存器对待储存数据的控制。

表 28.10-3. 通道储存数据字节序控制

通道储存数据位宽	原始数据	控制后数据	I2S*n*_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

28.10.2.5 A 率/μ 率压缩/解压缩

ESP32-S3 I2S*n* 对排列好字节序的待储存数据会按照 32-bit（缺省高位补 0）的方式进行 A 率/μ 率压缩/解压缩。

配置 I2S*n*_RX_PCM_BYPASS 为：

- 0，则不进行压缩/解压缩
- 1，则进行压缩/解压缩

配置 I2S*n*_RX_PCM_CONF 为：

- 0，A 律解压缩
- 1，A 律压缩
- 2，μ 律解压缩
- 3，μ 律压缩

至此，数据格式控制部分全部完成，数据通过 DMA 存入内存。

28.11 软件配置流程

28.11.1 软件配置 I2S*n* 发送流程

软件配置 I2S*n* 发送的流程如下：

1. 根据 28.6 小节的描述，配置时钟。
2. 根据表 28.4-1 的描述，配置信号管脚。

3. 根据主从机模式配置 I2S*n*_TX_SLAVE_MOD：
   - 0：主机发送模式
   - 1：从机发送模式
4. 根据 28.9 小节的描述，配置正确的发送数据模式和发送通道模式，置位 I2S*n*_TX_UPDATE。
5. 根据 28.7 小节的描述，复位发送单元和发送 FIFO。
6. 根据 28.12 小节的描述使能相应的中断。
7. 配置 DMA 发送链表。
8. 根据需要置位 I2S*n*_TX_STOP_EN，更多信息见章节 28.8.1。
9. 开始发送数据：
   - 主机模式下，等待 I2S*n* 从设备配置完成后，置位 I2S*n*_TX_START 开始发送数据；
   - 从机模式下，置位 I2S*n*_TX_START。I2S*n* 主设备提供 BCK 和 WS 信号后，开始发送数据。
10. 等待步骤 6 设置的中断信号，或查询 I2S*n*_TX_IDLE 检查传输是否结束：
    - 0：发送设备为工作状态；
    - 1：发送设备为空闲状态。
11. 清零 I2S*n*_TX_START。

28.11.2 软件配置 I2S*n* 接收流程

软件配置 I2S*n* 接收模式的流程如下：

1. 根据 28.6 小节的描述，配置时钟。
2. 根据表 28.4-1 的描述，配置信号管脚。
3. 配置 I2S*n*_RX_SLAVE_MOD 选择需要的模式：
   - 0：主机接收模式
   - 1：从机接收模式
4. 根据 28.10 小节的描述，配置正确的接收通道模式和接收数据模式，置位 I2S*n*_RX_UPDATE。
5. 根据 28.7 小节的描述，复位接收单元和接收 FIFO。
6. 根据 28.12 小节的描述使能相应的中断。
7. 配置 DMA 接收链表，并在 I2S*n*_RXEOF_NUM_REG 中配置接收数据长度。
8. 开始接收数据：
   - 在主机模式下，等待从机准备好后，置位 I2S*n*_RX_START 开始接收数据；
   - 在从机模式下，置位 I2S*n*_RX_START，等待主机提供 BCK 和 WS 信号后开始接收数据。
9. 接收的数据由 DMA 根据配置，存到 ESP32-S3 存储器的指定地址。最终产生步骤 6 中设置的中断。

28.12 I2S*n* 中断

- I2S_TX_HUNG_INT：当发送数据超时即触发此中断。例如，I2S*n* 配置为从机发送模式，但主机长时间未提供 BCK 或 WS 信号，则将触发该中断。超时配置见寄存器 I2S*n*_LC_HUNG_CONF_REG。
- I2S_RX_HUNG_INT：当接收数据超时即触发此中断。例如，I2S*n* 配置为从机接收模式，但主机长时间未发送数据，则将触发该中断。超时配置见寄存器 I2S*n*_LC_HUNG_CONF_REG。
- I2S_TX_DONE_INT：当发送数据完成即触发此中断。
- I2S_RX_DONE_INT：当接收数据完成即触发此中断。

28.13 寄存器列表

本小节的所有地址均为相对于 [I2S*n*] 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

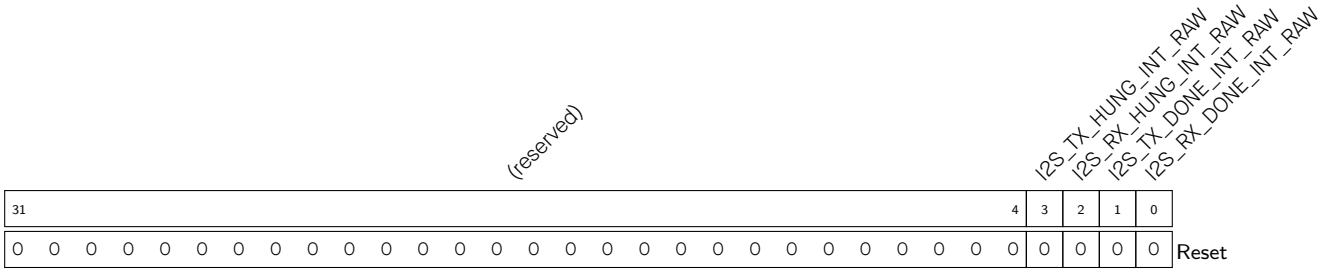
请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	I2S0 地址	I2S1 地址	访问
中断寄存器				
I2S_INT_RAW_REG	原始中断寄存器	0x000C	0x000C	RO/WTC/SS
I2S_INT_ST_REG	中断状态寄存器	0x0010	0x0010	RO
I2S_INT_ENA_REG	中断使能寄存器	0x0014	0x0014	R/W
I2S_INT_CLR_REG	中断清除寄存器	0x0018	0x0018	WT
RX/TX 控制和配置寄存器				
I2S_RX_CONF_REG	RX 配置寄存器	0x0020	0x0020	varies
I2S_RX_CONF1_REG	RX 配置寄存器 1	0x0028	0x0028	R/W
I2S_RX_CLKM_CONF_REG	RX 时钟配置寄存器	0x0030	0x0030	R/W
I2S_TX_PCM2PDM_CONF_REG	TX PCM-to-PDM 配置寄存器	0x0040	—	R/W
I2S_TX_PCM2PDM_CONF1_REG	TX PCM-to-PDM 配置寄存器	0x0044	—	R/W
I2S_RX_TDM_CTRL_REG	TX TDM 模式控制寄存器	0x0050	0x0050	R/W
I2S_RXEOF_NUM_REG	RX 数据大小控制寄存器	0x0064	0x0064	R/W
I2S_TX_CONF_REG	TX 配置寄存器	0x0024	0x0024	varies
I2S_TX_CONF1_REG	TX 配置寄存器 1	0x002C	0x002C	R/W
I2S_TX_CLKM_CONF_REG	TX 时钟配置寄存器	0x0034	0x0034	R/W
I2S_TX_TDM_CTRL_REG	TX TDM 模式控制寄存器	0x0054	0x0054	R/W
RX 时序和时钟寄存器				
I2S_RX_CLKM_DIV_CONF_REG	RX 时钟分频配置寄存器	0x0038	0x0038	R/W
I2S_RX_TIMING_REG	RX 时序控制寄存器	0x0058	0x0058	R/W
TX 时序和时钟寄存器				
I2S_TX_CLKM_DIV_CONF_REG	TX 时钟分频配置寄存器	0x003C	0x003C	R/W
I2S_TX_TIMING_REG	TX 时序控制寄存器	0x005C	0x005C	R/W
控制和配置寄存器				

名称	描述	I2S0 地址	I2S1 地址	访问
I2S_LC_HUNG_CONF_REG	超时配置寄存器	0x0060	0x0060	R/W
I2S_CONF_SIGLE_DATA_REG	single 数据寄存器	0x0068	0x0068	R/W
TX 状态寄存器				
I2S_STATE_REG	TX 状态寄存器	0x006C	0x006C	RO
版本寄存器				
I2S_DATE_REG	版本控制寄存器	0x0080	0x0080	R/W

28.14 寄存器

Register 28.1. I2S_INT_RAW_REG (0x000C)



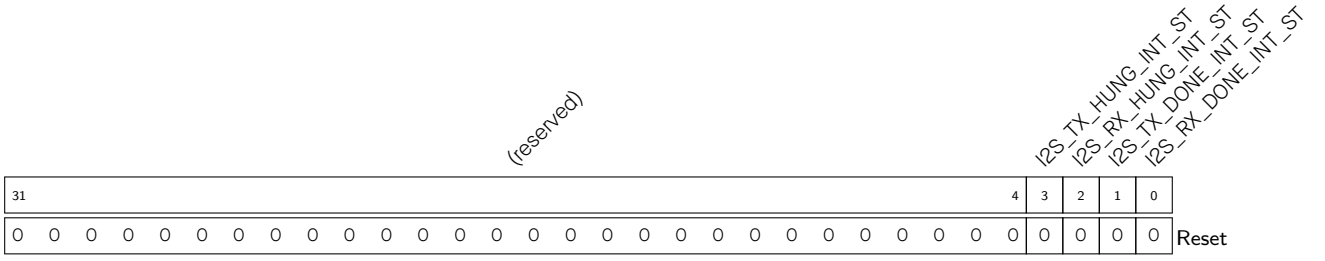
I2S_RX_DONE_INT_RAW I2S_RX_DONE_INT 中断的原始中断状态位。(RO/WTC/SS)

I2S_TX_DONE_INT_RAW I2S_TX_DONE_INT 中断的原始中断状态位。(RO/WTC/SS)

I2S_RX_HUNG_INT_RAW I2S_RX_HUNG_INT 中断的原始中断状态位。(RO/WTC/SS)

I2S_TX_HUNG_INT_RAW I2S_TX_HUNG_INT 中断的原始中断状态位。(RO/WTC/SS)

Register 28.2. I2S_INT_ST_REG (0x0010)



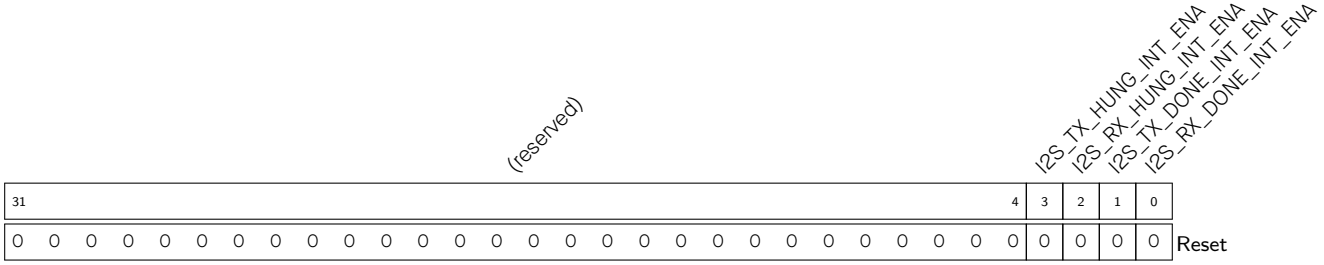
I2S_RX_DONE_INT_ST I2S_RX_DONE_INT 中断的屏蔽中断状态位。(RO)

I2S_TX_DONE_INT_ST I2S_TX_DONE_INT 中断的屏蔽中断状态位。(RO)

I2S_RX_HUNG_INT_ST I2S_RX_HUNG_INT 中断的屏蔽中断状态位。(RO)

I2S_TX_HUNG_INT_ST I2S_TX_HUNG_INT 中断的屏蔽中断状态位。(RO)

Register 28.3. I2S_INT_ENA_REG (0x0014)



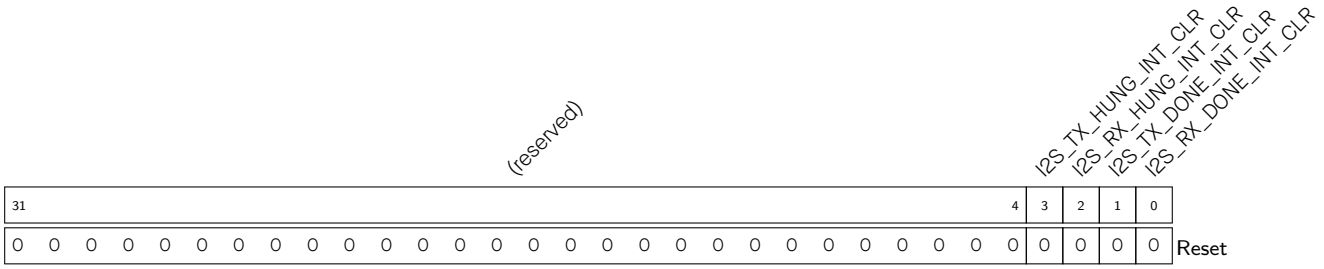
I2S_RX_DONE_INT_ENA I2S_RX_DONE_INT 的中断使能位。(R/W)

I2S_TX_DONE_INT_ENA I2S_TX_DONE_INT 的中断使能位。(R/W)

I2S_RX_HUNG_INT_ENA I2S_RX_HUNG_INT 的中断使能位。(R/W)

I2S_TX_HUNG_INT_ENA I2S_TX_HUNG_INT 的中断使能位。(R/W)

Register 28.4. I2S_INT_CLR_REG (0x0018)



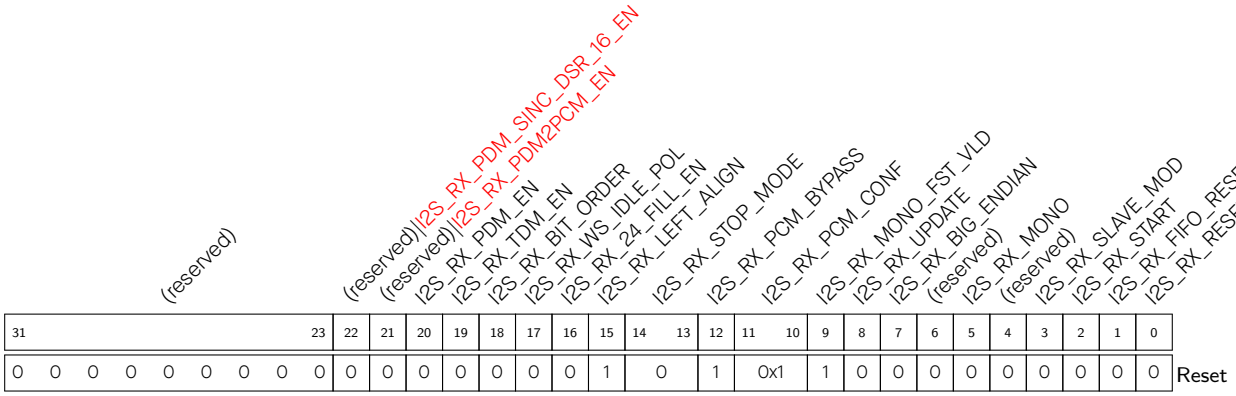
I2S_RX_DONE_INT_CLR I2S_RX_DONE_INT 的中断清除位。(WT)

I2S_TX_DONE_INT_CLR I2S_TX_DONE_INT 的中断清除位。(WT)

I2S_RX_HUNG_INT_CLR I2S_RX_HUNG_INT 的中断清除位。(WT)

I2S_TX_HUNG_INT_CLR I2S_TX_HUNG_INT 的中断清除位。(WT)

Register 28.5. I2S_RX_CONF_REG (0x0020)



I2S_RX_RESET 此位置 1，复位接收模块。(WT)

I2S_RX_FIFO_RESET 此位置 1，复位 RX FIFO。(WT)

I2S_RX_START 此位置 1，开始接收数据。(R/W)

I2S_RX_SLAVE_MOD 此位置 1，使能从机接收模式。(R/W)

I2S_RX_MONO 此位置 1，使能接收模块的单声道模式。(R/W)

I2S_RX_BIG_ENDIAN I2S RX 字节序。1：低字节数据写入高位地址；0：低字节数据写入低位地址。(R/W)

I2S_RX_UPDATE 写 1 将 I2S RX 寄存器从 APB 时钟域同步到 I2S RX 时钟域。寄存器更新完成后，此位将由硬件清除。(R/W/SC)

I2S_RX_MONO_FST_VLD 1：在 I2S RX 单声道模式下，第一个通道数据有效。0：在 I2S RX 单声道模式下，第二个通道数据有效。(R/W)

I2S_RX_PCM_CONF I2S RX 压缩/解压缩配置位。0 (atol)：A 率解压缩；1 (ltoa)：A 率压缩；2 (utol)：μ 率解压缩；3 (ltou)：μ 率压缩。(R/W)

I2S_RX_PCM_BYPASS 置位此位，接收数据将绕过压缩/解压缩模块。(R/W)

I2S_RX_STOP_MODE 0：只有在 I2S_RX_START 被清除时，I2S RX 才会停止工作；1：当 I2S_RX_START 为 0 或 in_suc_eof 为 1 时，I2S RX 停止工作；2：当 I2S_RX_STAR 为 0 或 RX FIFO 为满时，I2S RX 停止工作。(R/W)

I2S_RX_LEFT_ALIGN 1：使能 I2S RX 左对齐模式。0：使能 I2S RX 右对齐模式。(R/W)

I2S_RX_24_FILL_EN 1：将 24 位通道数据位保存到 32 位储存数据（多余的位填充为 0）。1：将 24 位通道数据位保存到 24 位储存数据。(R/W)

I2S_RX_WS_IDLE_POL 0：WS 为 0 时，接收左通道数据；WS 为 1 时，接收右通道数据。1：WS 为 1 时，接收左通道数据；WS 为 0 时，接收右通道数据。(R/W)

I2S_RX_BIT_ORDER 配置是否调换 I2S RX 待接收数据的位顺序。0：不调换；1：调换。(R/W)

见下页

Register 28.5. I2S_RX_CONF_REG (0x0020)

接上页

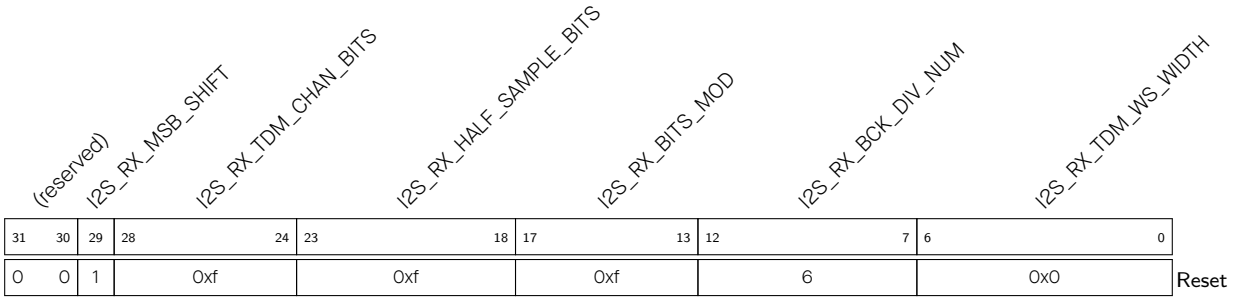
I2S_RX_TDM_EN 1：使能 I2S TDM RX 模式。0：禁用 I2S TDM RX 模式。(R/W)

I2S_RX_PDM_EN 1：使能 I2S PDM RX 模式。0：禁用 I2S PDM RX 模式。(R/W)

I2S_RX_PDM2PCM_EN（仅适用于 I2S0） 1：使能 PDM-to-PCM RX 模式。0：禁用 PDM-to-PCM RX 模式。(R/W)

I2S_RX_PDM_SINC_DSR_16_EN（仅适用于 I2S0） 配置 PDM RX 滤波器组 1 模块的下采样率。1：下采样率为 128。0：下采样率为 64。(R/W)

Register 28.6. I2S_RX_CONF1_REG (0x0028)



I2S_RX_TDM_WS_WIDTH 在 TDM 模式下，rx_ws_out（WS 默认电平）时长等于 (I2S_RX_TDM_WS_WIDTH + 1) * T_BCK。(R/W)

I2S_RX_BCK_DIV_NUM 在 RX 模式下，配置 BCK 时钟的分频系数。注意，不可配置为 1。(R/W)

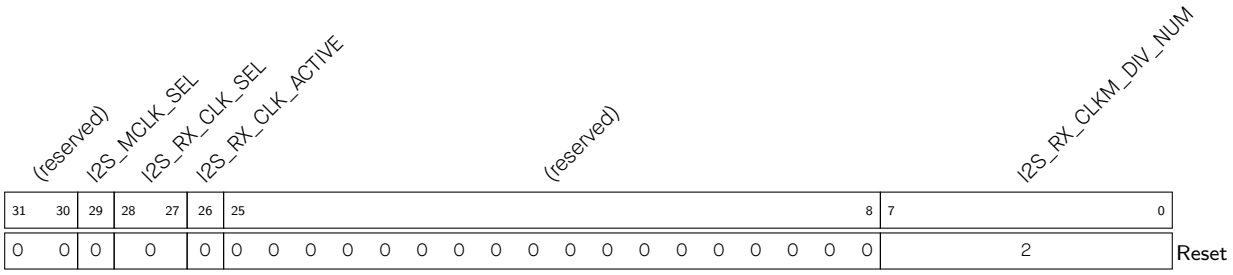
I2S_RX_BITS_MOD RX 模式下，配置接收通道的有效数据位长度。7：所有有效的通道数据均为 8 位模式。15：所有有效的通道数据均为 16 位模式。23：所有有效的通道数据均为 24 位模式。31：所有有效的通道数据均为 32 位模式。(R/W)

I2S_RX_HALF_SAMPLE_BITS I2S RX 单次采样比特数的一半。I2S_TX_HALF_SAMPLE_BITS x 2 等于在一个 WS 信号中，BCK 持续的周期长。(R/W)

I2S_RX_TDM_CHAN_BITS 在 TDM RX 模式下，每个通道的 RX 数据位等于该值 + 1。(R/W)

I2S_RX_MSB_SHIFT 控制 WS 和数据的 MSB 位之间的时序关系。1：相隔一个周期；0：上升沿对齐。(R/W)

Register 28.7. I2S_RX_CLKM_CONF_REG (0x0030)



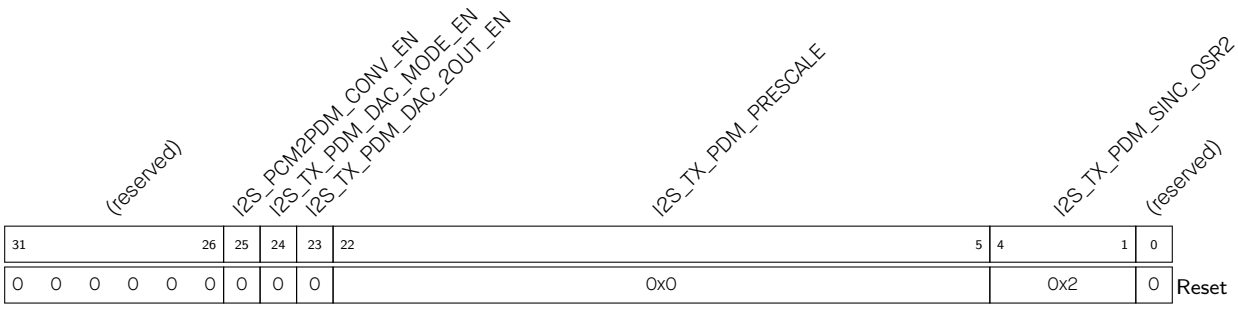
I2S_RX_CLKM_DIV_NUM I2S RX 时钟分频器的整数值。(R/W)

I2S_RX_CLK_ACTIVE I2S RX 单元时钟使能信号。(R/W)

I2S_RX_CLK_SEL 选择 I2S RX 单元的时钟源。0：XTAL_CLK；1：PLL_D2_CLK；2：PLL_F160M_CLK；3：I2S_MCLK_in。(R/W)

I2S_MCLK_SEL 0：使用 I2S TX 单元的时钟用作 I2S_MCLK_OUT。1：使用 I2S RX 单元的时钟用作 I2S_MCLK_OUT。(R/W)

Register 28.8. I2S_TX_PCM2PDM_CONF_REG（仅适用于 I2S0） (0x0040)



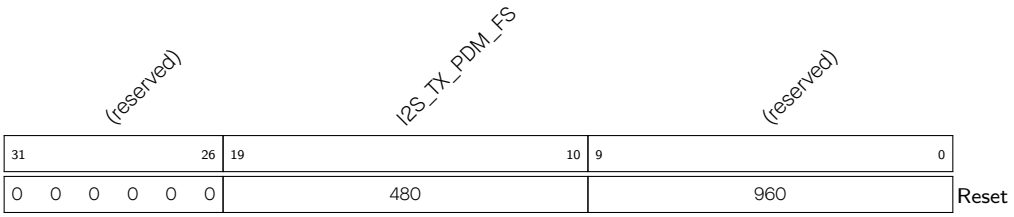
I2S_TX_PDM_SINC_OSR2 I2S TX PDM OSR 的值。(R/W)

I2S_TX_PDM_DAC_2OUT_EN 0：1-line DAC 输出模式；1：2-line DAC 输出模式；仅在 I2S_TX_PDM_DAC_MODE_EN 为 1 时有效。(R/W)

I2S_TX_PDM_DAC_MODE_EN 0：1-line PDM 输出模式；1：DAC 输出模式。(R/W)

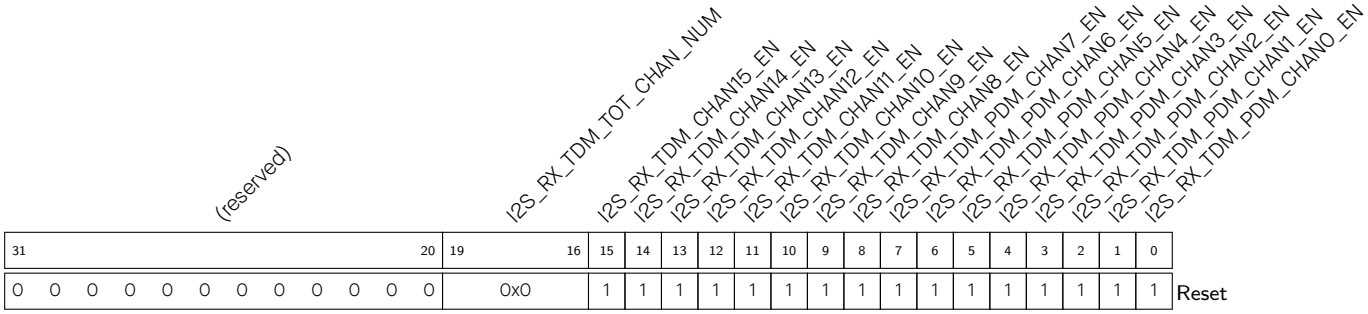
I2S_PCM2PDM_CONV_EN 使能 I2S TX PCM-to-PDM 转换器。(R/W)

Register 28.9. I2S_TX_PCM2PDM_CONF1_REG（仅适用于 I2S0） (0x0044)



I2S_TX_PDM_FS 配置 I2S TX PDM 上采样率。(R/W)

Register 28.10. I2S_RX_TDM_CTRL_REG (0x0050)



I2S_RX_TDM_PDM_CHAN0_EN 1：使能 I2S TDM 或 PDM RX 通道 0 的有效输入数据。0：禁用该功能，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_PDM_CHAN1_EN 1：使能 I2S TDM 或 PDM RX 通道 1 的有效输入数据。0：禁用该功能，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_PDM_CHAN2_EN 1：使能 I2S TDM 或 PDM RX 通道 2 的有效输入数据。0：禁用该功能，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_PDM_CHAN3_EN 1：使能 I2S TDM 或 PDM RX 通道 3 的有效输入数据。0：禁用该功能，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_PDM_CHAN4_EN 1：使能 I2S TDM 或 PDM RX 通道 4 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_PDM_CHAN5_EN 1：使能 I2S TDM 或 PDM RX 通道 5 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_PDM_CHAN6_EN 1：使能 I2S TDM 或 PDM RX 通道 6 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_PDM_CHAN7_EN 1：使能 I2S TDM 或 PDM RX 通道 7 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_CHAN8_EN 1：使能 I2S TDM RX 通道 8 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_CHAN9_EN 1：使能 I2S TDM RX 通道 9 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_CHAN10_EN 1：使能 I2S TDM RX 通道 10 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_CHAN11_EN 1：使能 I2S TDM RX 通道 11 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_CHAN12_EN 1：使能 I2S TDM RX 通道 12 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_CHAN13_EN 1：使能 I2S TDM RX 通道 13 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

见下页

Register 28.10. I2S_RX_TDM_CTRL_REG (0x0050)

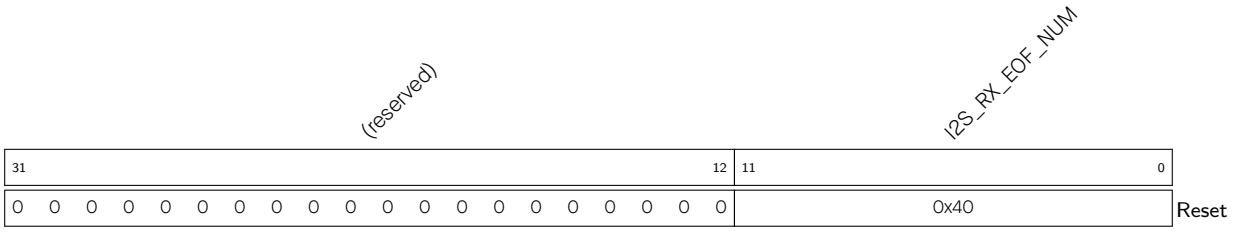
接上页

I2S_RX_TDM_CHAN14_EN 1：使能 I2S TDM RX 通道 14 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

I2S_RX_TDM_CHAN15_EN 1：使能 I2S TDM RX 通道 15 的有效输入数据。0：禁用该模式，在该通道中仅输入 0。(R/W)

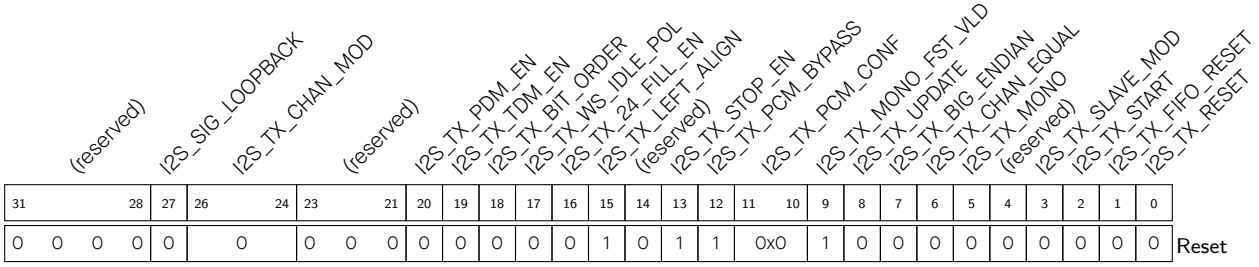
I2S_RX_TDM_TOT_CHAN_NUM 在 I2S TDM RX 模式下，使用的通道总数 = 该值 + 1。(R/W)

Register 28.11. I2S_RXEOF_NUM_REG (0x0064)



I2S_RX_EOF_NUM 用于配置接收数据的长度。接收数据长度等于 (I2S_RX_BITS_MOD + 1) * (I2S_RX_EOF_NUM + 1)。如果接收数据的长度达到该值，则将在已配置的 DMA RX 通道中触发 in_suc_eof 中断。(R/W)

Register 28.12. I2S_TX_CONF_REG (0x0024)



I2S_TX_RESET 此位置 1，复位发送单元。(WT)

I2S_TX_FIFO_RESET 此位置 1，复位 TX FIFO。(WT)

I2S_TX_START 此位置 1，开始发送数据。(R/W)

I2S_TX_SLAVE_MOD 此位置 1，使能从机发送模式。(R/W)

I2S_TX_MONO 此位置 1，使能发送单元的单声道模式。(R/W)

I2S_TX_CHAN_EQUAL 1：在 I2S TX 单声道模式或 TDM 模式下，左声道数据等于右声道数据。0：在 I2S TX 单声道模式或 TDM 模式下，I2S_SINGLE_DATA 为无效的通道数据。(R/W)

I2S_TX_BIG_ENDIAN I2S TX 字节序。1：低字节数据写入高位地址；0：低字节数据写入低位地址。(R/W)

I2S_TX_UPDATE 写 1 将 I2S TX 寄存器从 APB 时钟域同步到 I2S TX 时钟域。寄存器更新完成后，此位将由硬件清除。(R/W/SC)

I2S_TX_MONO_FST_VLD 1：在 I2S TX 单声道模式下，第一个通道数据有效。0：在 I2S TX 单声道模式下，第二个通道数据有效。(R/W)

I2S_TX_PCM_CONF I2S TX 压缩/解压缩配置位。0 (atol)：A 率解压缩；1 (ltoa)：A 率压缩；2 (utol)：μ 率解压缩；3 (ltou)：μ 率压缩。(R/W)

I2S_TX_PCM_BYPASS 置位此位，发送数据将绕过压缩/解压缩模块。(R/W)

I2S_TX_STOP_EN 将此位置 1，当 TX FIFO 为空时，发送单元停止输出 BCK 和 WS 信号。(R/W)

I2S_TX_LEFT_ALIGN 1：使能 I2S TX 左对齐模式。0：使能 I2S TX 右对齐模式。(R/W)

I2S_TX_24_FILL_EN 1：将 24 位数据以 32 位的格式发送出去（不足的位用 0 填充）；0：将 24 位数据以 24 位的格式发送出去。(R/W)

I2S_TX_WS_IDLE_POL 0：WS 为 0 时，发送左通道数据；WS 为 1 时，发送右通道数据。1：WS 为 1 时，发送左通道数据；WS 为 0 时，发送右通道数据。(R/W)

I2S_TX_BIT_ORDER 配置是否调换 I2S TX 待发送有效数据的位顺序。0：不调换；1：调换。(R/W)

I2S_TX_TDM_EN 1：使能 I2S TDM TX 模式。0：禁用 I2S TDM TX 模式。(R/W)

见下页

Register 28.12. I2S_TX_CONF_REG (0x0024)

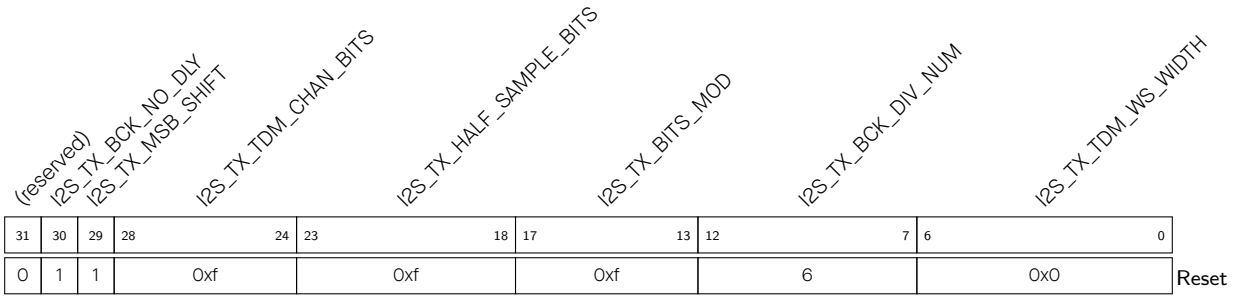
接上页

I2S_TX_PDM_EN 1：使能 I2S PDM TX 模式。0：禁用 I2S PDM TX 模式。(R/W)

I2S_TX_CHAN_MOD I2S TX 通道配置位。更多信息见表 28.9-4。(R/W)

I2S_SIG_LOOPBACK 置 1 时，发送单元和接收单元共享 WS 和 BCK 信号。(R/W)

Register 28.13. I2S_TX_CONF1_REG (0x002C)



I2S_TX_TDM_WS_WIDTH 在 TDM 模式下，tx_ws_out（WS 默认电平）时长等于 (I2S_TX_TDM_WS_WIDTH + 1) * T_BCK。(R/W)

I2S_TX_BCK_DIV_NUM 在 TX 模式下，配置 BCK 时钟的分频系数。注意，不可配置为 1。(R/W)

I2S_TX_BITS_MOD TX 模式下，配置发送通道的有效数据位长度。7：所有有效的通道数据均为 8 位模式。15：所有有效的通道数据均为 16 位模式。23：所有有效的通道数据均为 24 位模式。31：所有有效的通道数据均为 32 位模式。(R/W)

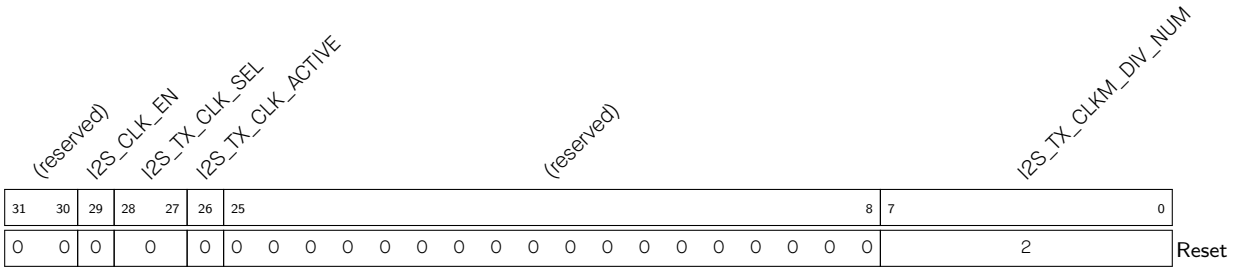
I2S_TX_HALF_SAMPLE_BITS TX 单次采样比特数的一半。I2S_TX_HALF_SAMPLE_BITS x 2 等于在一个 WS 信号中，BCK 持续的周期长。(R/W)

I2S_TX_TDM_CHAN_BITS 在 TDM TX 模式下，每个通道的 TX 数据位等于该值 + 1。(R/W)

I2S_TX_MSB_SHIFT 控制 WS 和数据的 MSB 位之间的时序关系。1：相隔一个周期；0：上升沿对齐。(R/W)

I2S_TX_BCK_NO_DLY 1：在主机模式下，BCK 上升沿和下降沿没有延迟。0：在主机模式下，BCK 上升沿和下降沿有延迟。(R/W)

Register 28.14. I2S_TX_CLKM_CONF_REG (0x0034)



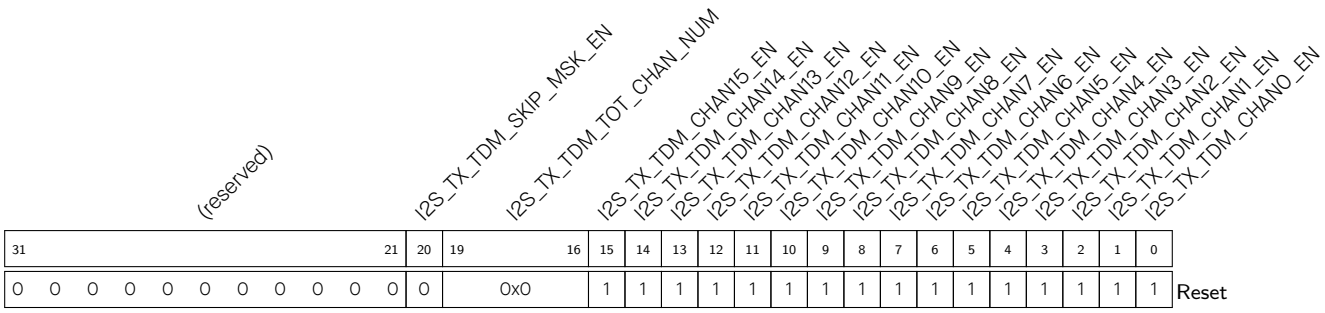
I2S_TX_CLKM_DIV_NUM I2S TX 时钟分频器的整数值。(R/W)

I2S_TX_CLK_ACTIVE I2S TX 单元时钟使能信号。(R/W)

I2S_TX_CLK_SEL 选择 I2S TX 单元的时钟源。0：XTAL_CLK；1：PLL_D2_CLK；2：PLL_F160M_CLK；3：I2S_MCLK_in。(R/W)

I2S_CLK_EN 置位此位，使能时钟门控。(R/W)

Register 28.15. I2S_TX_TDM_CTRL_REG (0x0054)



I2S_TX_TDM_CHAN0_EN 1：使能 I2S TDM TX 通道 0 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN1_EN 1：使能 I2S TDM TX 通道 1 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN2_EN 1：使能 I2S TDM TX 通道 2 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN3_EN 1：使能 I2S TDM TX 通道 3 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN4_EN 1：使能 I2S TDM TX 通道 4 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN5_EN 1：使能 I2S TDM TX 通道 5 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN6_EN 1：使能 I2S TDM TX 通道 6 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN7_EN 1：使能 I2S TDM TX 通道 7 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN8_EN 1：使能 I2S TDM TX 通道 8 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN9_EN 1：使能 I2S TDM TX 通道 9 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN10_EN 1：使能 I2S TDM TX 通道 10 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN11_EN 1：使能 I2S TDM TX 通道 11 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN12_EN 1：使能 I2S TDM TX 通道 12 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN13_EN 1：使能 I2S TDM TX 通道 13 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

见下页

Register 28.15. I2S_TX_TDM_CTRL_REG (0x0054)

接上页

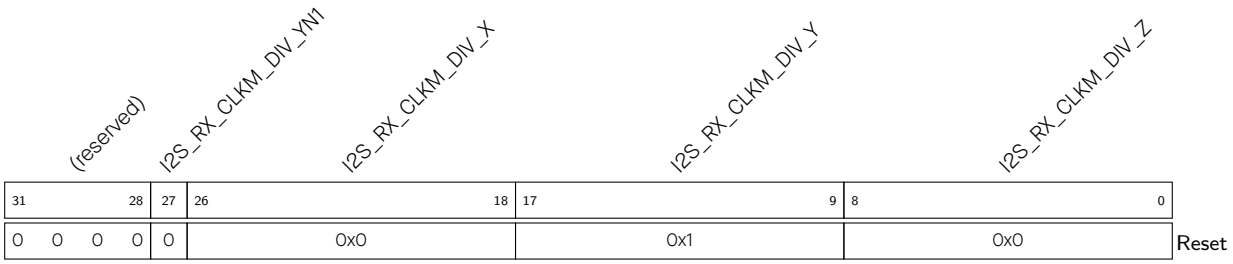
I2S_TX_TDM_CHAN14_EN 1：使能 I2S TDM TX 通道 14 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_CHAN15_EN 1：使能 I2S TDM TX 通道 15 的有效输出数据。0：通道发送数据由 I2S_TX_CHAN_EQUAL 和 I2S_SINGLE_DATA 控制，参考章节 28.9.2.1。(R/W)

I2S_TX_TDM_TOT_CHAN_NUM 在 I2S TDM TX 模式下，使用的通道总数等于该值 + 1。(R/W)

I2S_TX_TDM_SKIP_MSK_EN 置位此位，则当 DMA TX buffer 存储了 (I2S_TX_TDM_TOT_CHAN_NUM + 1) 个通道的数据时，仅有启用的通道的数据会被发送。清除此位，则 DMA TX buffer 中的所有数据都用于启用的通道。(R/W)

Register 28.16. I2S_RX_CLKM_DIV_CONF_REG (0x0038)



I2S_RX_CLKM_DIV_Z b <= a/ 2 时，I2S_RX_CLKM_DIV_Z 的值为 b。b > a/ 2 时，I2S_RX_CLKM_DIV_Z 的值为 a - b。(R/W)

I2S_RX_CLKM_DIV_Y b <= a/ 2 时，I2S_RX_CLKM_DIV_Y 的值为 a%b。b > a/ 2 时，I2S_RX_CLKM_DIV_Y 的值为 a%(a-b)。(R/W)

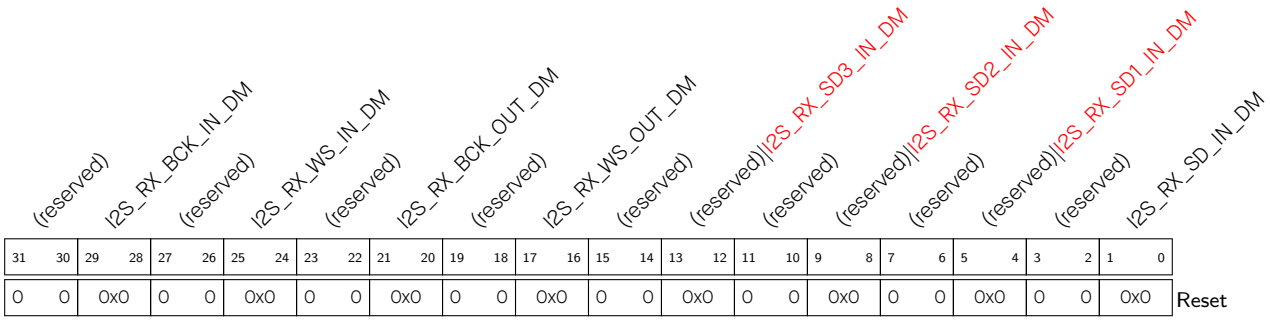
I2S_RX_CLKM_DIV_X b <= a/ 2 时，I2S_RX_CLKM_DIV_X 的值为 floor(a/ b) - 1。b > a/ 2，I2S_RX_CLKM_DIV_X 的值为 floor(a/(a - b)) - 1。(R/W)

I2S_RX_CLKM_DIV_YN1 b <= a/ 2 时，I2S_RX_CLKM_DIV_YN1 的值为 0。b > a/ 2 时，I2S_RX_CLKM_DIV_YN1 的值为 1。(R/W)

说明：

上文所述的 “a” 和 “b” 分别为小数分频的分母部分和分子部分。更多信息，见第 28.6 小节。

Register 28.17. I2S_RX_TIMING_REG (0x0058)



I2S_RX_SD_IN_DM I2S RX SD 输入信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

I2S_RX_SD1_IN_DM（仅适用于 I2S0） I2S RX SD1 输入信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

I2S_RX_SD2_IN_DM（仅适用于 I2S0） I2S RX SD2 输入信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

I2S_RX_SD3_IN_DM（仅适用于 I2S0） I2S RX SD3 输入信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

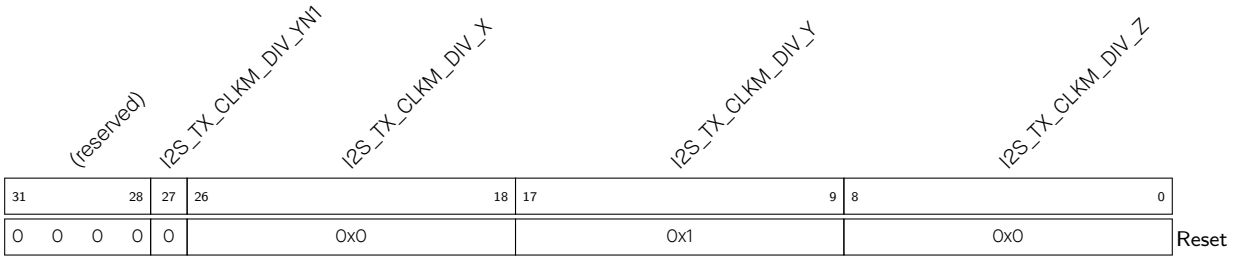
I2S_RX_WS_OUT_DM I2S RX WS 输出信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

I2S_RX_BCK_OUT_DM I2S RX BCK 输出信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

I2S_RX_WS_IN_DM I2S RX WS 输入信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

I2S_RX_BCK_IN_DM I2S RX BCK 输入信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

Register 28.18. I2S_TX_CLKM_DIV_CONF_REG (0x003C)



I2S_TX_CLKM_DIV_Z b <= a/ 2 时，I2S_TX_CLKM_DIV_Z 的值为 b。b > a/ 2 时，I2S_TX_CLKM_DIV_Z 的值为 a - b。(R/W)

I2S_TX_CLKM_DIV_Y b <= a/ 2 时，I2S_TX_CLKM_DIV_Y 的值为 a%b。b > a/ 2 时，I2S_TX_CLKM_DIV_Y 的值为 a%(a - b)。(R/W)

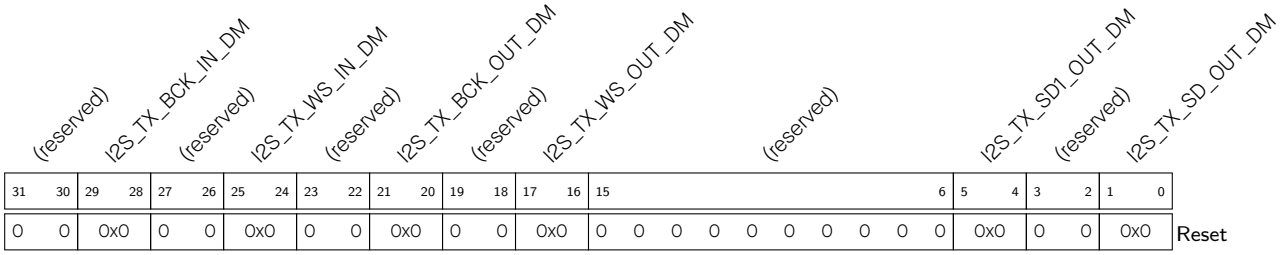
I2S_TX_CLKM_DIV_X b <= a/ 2 时，I2S_TX_CLKM_DIV_X 的值为 floor(a/ b) - 1。b > a/ 2，I2S_TX_CLKM_DIV_X 的值为 floor(a/(a - b)) - 1。(R/W)

I2S_TX_CLKM_DIV_YN1 b <= a/ 2 时，I2S_TX_CLKM_DIV_YN1 的值为 0。b > a/ 2 时，I2S_TX_CLKM_DIV_YN1 的值为 1。(R/W)

说明：

上文所述的 “a” 和 “b” 分别为小数分频的分母部分和分子部分。更多信息，见第 28.6 小节。

Register 28.19. I2S_TX_TIMING_REG (0x005C)



I2S_TX_SD_OUT_DM I2S TX SD 输出信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

I2S_TX_SD1_OUT_DM I2S TX SD1 输出信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

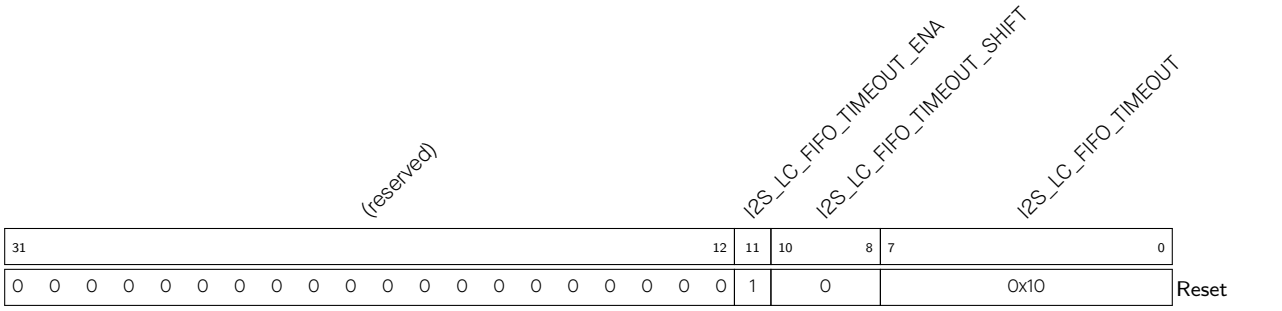
I2S_TX_WS_OUT_DM I2S TX WS 输出信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

I2S_TX_BCK_OUT_DM I2S TX BCK 输出信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

I2S_TX_WS_IN_DM I2S TX WS 输入信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

I2S_TX_BCK_IN_DM I2S TX BCK 输入信号的延迟模式。0：旁路。1：上升沿延迟。2：下降沿延迟。3：不使用该功能。(R/W)

Register 28.20. I2S_LC_HUNG_CONF_REG (0x0060)

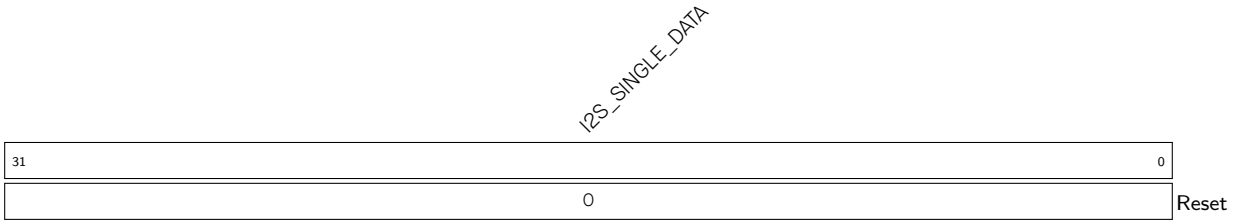


I2S_LC_FIFO_TIMEOUT FIFO Hung 计数器等于该值时，将触发 I2S_TX_HUNG_INT 中断或 I2S_RX_HUNG_INT 中断。(R/W)

I2S_LC_FIFO_TIMEOUT_SHIFT 用于分频滴答计数器的阈值。计数器值大于等于 $88000/2^{I2S\_LC\_FIFO\_TIMEOUT\_SHIFT}$ 时，复位滴答计数器。(R/W)

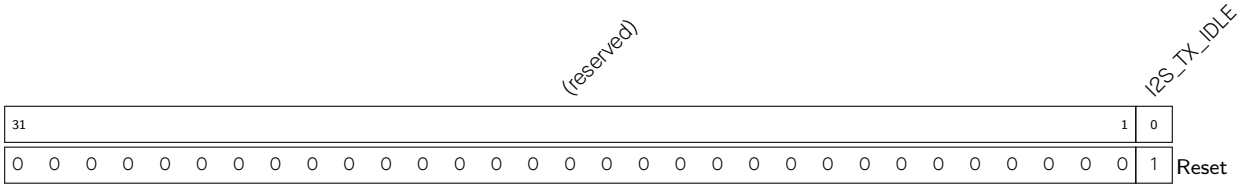
I2S_LC_FIFO_TIMEOUT_ENA FIFO 超时使能位。(R/W)

Register 28.21. I2S_CONF_SIGLE_DATA_REG (0x0068)



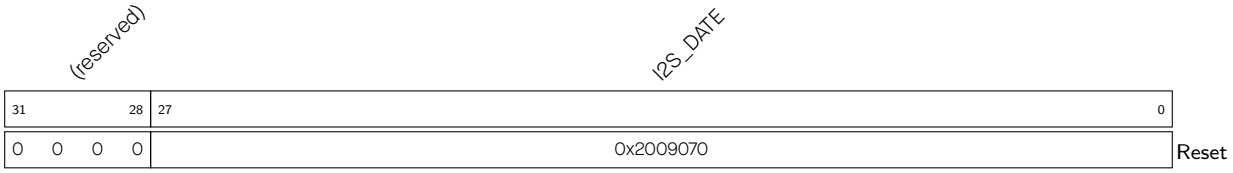
I2S_SINGLE_DATA 配置用于发送的通道常量数据。(R/W)

Register 28.22. I2S_STATE_REG (0x006C)



I2S_TX_IDLE 1：I2S TX 单元处于空闲状态。0：I2S TX 单元处于工作状态。(RO)

Register 28.23. I2S_DATE_REG (0x0080)



I2S_DATE 版本控制寄存器。(R/W)

第 29 章

LCD 与 Camera 控制器 (LCD_CAM)

29.1 概述

ESP32-S3 的 LCD_CAM 控制器包含独立的 LCD 模块和 Camera 模块。其中 LCD 模块用于发送并行视频数据信号，其总线支持 RGB、MOTO6800 和 I8080 等接口时序。Camera 模块用于接收并行视频数据信号，其总线支持 DVP 8-/16-bit 模式。

29.2 特性

- 支持以下工作模式：
  - LCD 主机发送模式
  - Camera 从机接收模式
  - Camera 主机接收模式
- 支持同时外接 LCD 和 Camera
- 支持单独外接 LCD，
  - 可配置为 8-bit 或 16-bit 并行输出模式
  - 支持 RGB、MOTO6800、I8080 等多种 LCD 模式
  - LCD 数据可由 GDMA 取自内部存储器
- 支持单独外接 Camera（即 DVP 图像传感器），
  - 可配置为 8-bit 或 16-bit 位并行输入模式
  - Camera 数据可由 GDMA 存入内部存储器
- 支持 LCD_CAM 接口中断

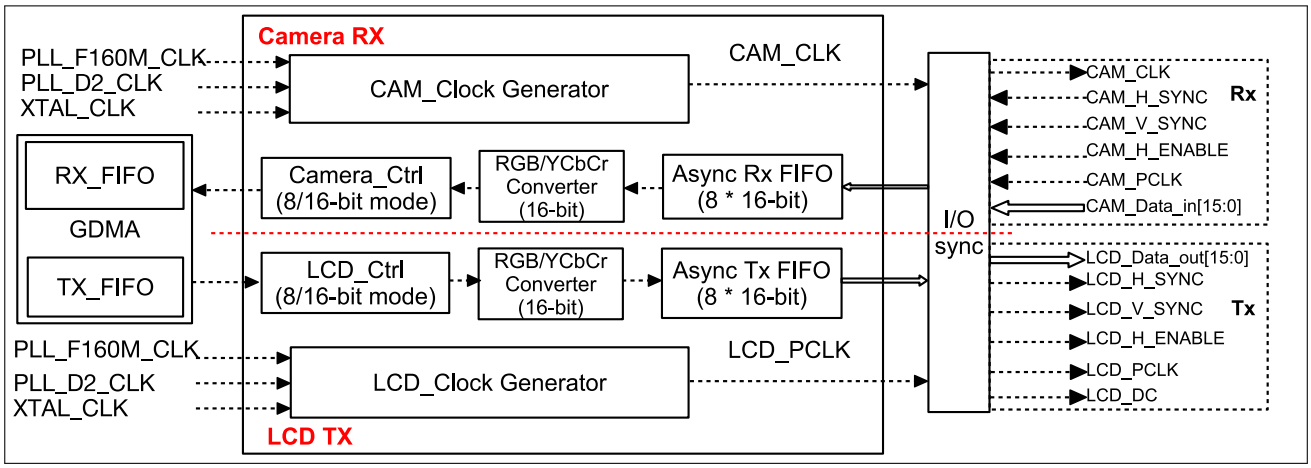
29.3 功能描述

29.3.1 功能框图

图 29.3-1 是 ESP32-S3 LCD_CAM 模块的结构框图，包含：

- 一个独立的发送控制单元 (LCD_Ctrl)
- 一个独立的接收控制单元 (Camera_Ctrl)
- 一个发送异步 FIFO (Async Tx FIFO)，用于与外部设备进行交互

- 一个接收异步 FIFO (Async Rx FIFO)，用于与外部设备进行交互
- 两个时钟生成模块（LCD_Clock Generator 和 CAM_Clock Generator），用于生成对应模块的时钟
- 两个格式转换模块 (RGB/YCbCr Converter)，用于各种格式的视频数据互相转换



-→: 控制信号 =>: 数据流

图 29.3-1. LCD_CAM 功能框图

29.3.2 信号描述

表 29.3-1. 信号描述

工作模式	信号[1]	方向	功能
Camera 从机接收模式	CAM_PCLK	输入	像素时钟输入信号
	CAM_V_SYNC	输入	帧同步输入信号 (VSYNC)[3]
	CAM_H_SYNC	输入	行同步输入信号 (HSYNC)[3]
	CAM_H_ENABLE	输入	行有效输入信号 (DE)[3]
	CAM_Data_in[*N*:0][2]	输入	并行输入数据总线，支持 8/16 位并行数据输入
Camera 主机接收模式	CAM_PCLK	输入	像素时钟输入信号
	CAM_CLK	输出	主机时钟输出信号
	CAM_V_SYNC	输入	帧同步输入信号[3]
	CAM_H_SYNC	输入	行同步输入信号[3]
	CAM_H_ENABLE	输入	行有效输入信号[3]
	CAM_Data_in[*N*:0][2]	输入	并行输入数据总线，支持 8/16 位并行数据输入
LCD 主机发送模式	LCD_PCLK	输出	LCD 像素时钟输出信号
	LCD_H_SYNC	输出	RGB 格式下，用作行同步信号
	LCD_V_SYNC	输出	RGB 格式下，用作帧同步信号
	LCD_H_ENABLE	输出	RGB 格式下，用作行有效信号
	LCD_CD	输出	I8080 格式下，用作命令和数据信号 (CD)
	LCD_CS	输出	I8080/MOTO6800 格式下，用作片选信号 (CS)
	LCD_Data_out[*N*:0][2]	输出	并行输出数据总线，支持 8/16 位并行数据输出

见下页

表 29.3-1 – 接上页

工作模式	信号[1]	方向	功能

[1] LCD_CAM 的所有信号均需要经过 GPIO 交换矩阵映射到芯片管脚。更多信息请参考章节 *6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)*。

[2] 信号位宽 8 或 16 时，则 *N* 分别为 7 或 15。

[3] LCD_CAM_CAM_VH_DE_MODE_EN 为 1（VSYNC + HSYNC 模式）时：VSYNC、HSYNC、DE 线同时控制数据有效，因此用户需要接 VSYNC、HSYNC、DE 信号线；LCD_CAM_CAM_VH_DE_MODE_EN 为 0（DE 模式）时：VSYNC、DE 线同时控制数据有效，HSYNC 线可以不接。但是在此模式下，Camera 的 YUV-RGB 转换功能将不可用。

29.3.3 LCD_CAM 模块时钟

29.3.3.1 LCD 时钟

时钟源经时钟生成模块处理后，生成 LCD 模块所需的时钟，如图 29.3-2 所示。

其中，

- LCD_CLK 为 LCD 模块的主时钟，由时钟源分频获得；
- 像素时钟 LCD_PCLK 再由 LCD_CLK 分频获得。

寄存器 LCD_CAM_LCD_CLOCK_REG 中 LCD_CAM_LCD_CLK_SEL 用于选择时钟源：

- 0：关闭 LCD 时钟源；
- 1：选择 XTAL_CLK；
- 2：选择 PLL_D2_CLK；
- 3：选择 PLL_F160M_CLK。

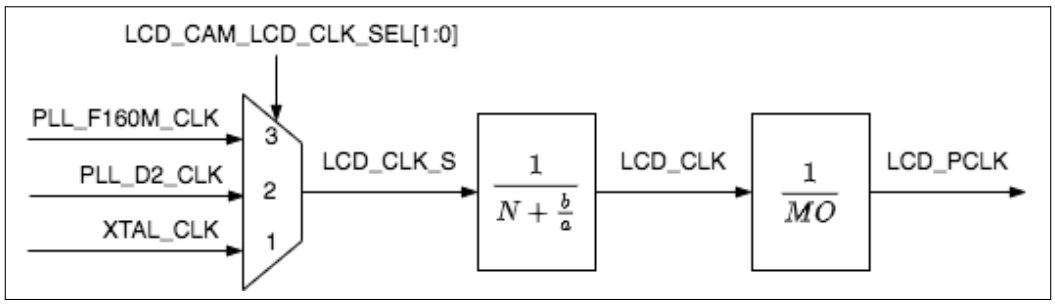


图 29.3-2. LCD 时钟

LCD_CLK 的频率 f_{LCD_CLK} 与分频器时钟源频率 $f_{LCD\_CLK\_S}$ 间的关系如下：

$$f_{LCD\_CLK} = \frac{f_{LCD\_CLK\_S}}{N + \frac{b}{a}}$$

其中，2 <= N <= 256，N 对应为 LCD_CAM_LCD_CLOCK_REG 寄存器中 LCD_CAM_LCD_CLKM_DIV_NUM 的值，具体为：

- LCD_CAM_LCD_CLKM_DIV_NUM = 0 时，N = 256；
- LCD_CAM_LCD_CLKM_DIV_NUM = 1 时，N = 2；

- LCD_CAM_LCD_CLKM_DIV_NUM 为其它值时，N = LCD_CAM_LCD_CLKM_DIV_NUM 的值。

b 对应 LCD_CAM_LCD_CLKM_DIV_B 的值，a 对应 LCD_CAM_LCD_CLKM_DIV_A 的值。对于整数分频，LCD_CAM
_LCD_CLKM_DIV_A 和 LCD_CAM_LCD_CLKM_DIV_B 都清零；对于小数分频，LCD_CAM_LCD_CLKM_DIV_B 的值应小于 LCD_CAM_LCD_CLKM_DIV_A 的值。

LCD 模块的像素时钟 PCLK 为 LCD_PCLK 信号，由 LCD_CLK 分频获得。即：

$$f_{\text{LCD\_PCLK}} = \frac{f_{\text{LCD\_CLK}}}{\text{MO}}$$

其中，MO 由 LCD_CAM_LCD_CLK_EQU_SYSCLK 和 LCD_CAM_LCD_CLKCNT_N 决定，具体为：

- LCD_CAM_LCD_CLK_EQU_SYSCLK = 1 时，MO = 1；
- LCD_CAM_LCD_CLK_EQU_SYSCLK = 0 时，MO = LCD_CAM_LCD_CLKCNT_N + 1。

注意：

- LCD_CAM_LCD_CLKCNT_N 不可配置为 0；
- 使用小数分频功能会产生时钟抖动。当 LCD_CLK 和 LCD_PCLK 无法通过 PLL_F160M_CLK 整数分频产生时，可使用 PLL_D2_CLK 作为时钟源，详情请参考章节 7 复位和时钟。

29.3.3.2 Camera 时钟

时钟源经时钟生成模块处理后，生成 Camera 模块所需的时钟，如图 29.3-3 所示。

其中，

- CAM_CLK 为 Camera 模块的主机时钟输出，由时钟源分频获得；
- 像素时钟 LCD_PCLK 由 Camera 从机输入获得。

寄存器 LCD_CAM_CAM_CTRL_REG 中 LCD_CAM_CAM_CLK_SEL 用于选择时钟源：

- 0：关闭 CAM 时钟源；
- 1：选择 XTAL_CLK；
- 2：选择 PLL_D2_CLK；
- 3：选择 PLL_F160M_CLK。

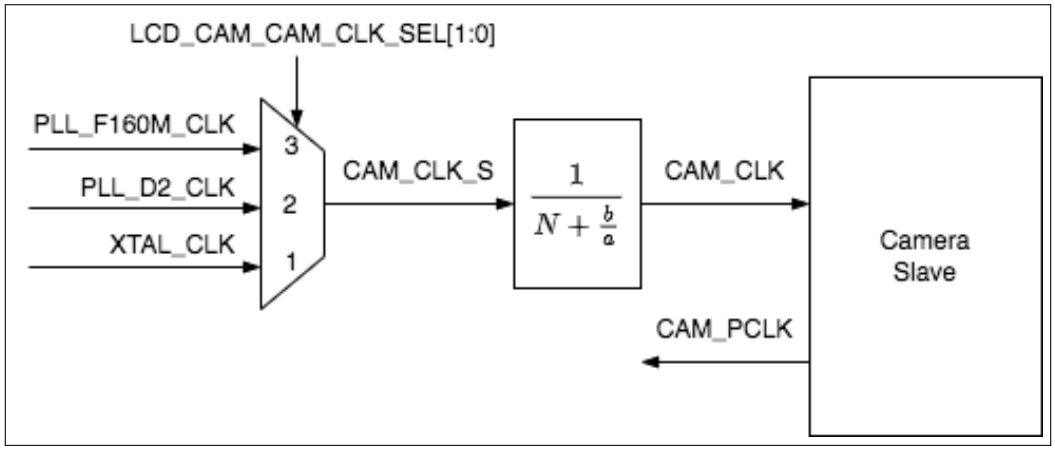


图 29.3-3. Camera 时钟

CAM_CLK 的频率 f_{CAM_CLK} 与分频器时钟源频率 $f_{CAM\_CLK\_S}$ 间的关系如下：

$$f_{CAM\_CLK} = \frac{f_{CAM\_CLK\_S}}{N + \frac{b}{a}}$$

其中，2 <= N <= 256，N 对应为 LCD_CAM_CAM_CTRL_REG 寄存器中 LCD_CAM_CAM_CLKM_DIV_NUM 的值，具体为：

- LCD_CAM_CAM_CLKM_DIV_NUM = 0 时，N = 256；
- LCD_CAM_CAM_CLKM_DIV_NUM = 1 时，N = 2；
- LCD_CAM_CAM_CLKM_DIV_NUM 为其它值时，N = LCD_CAM_CAM_CLKM_DIV_NUM 的值。

b 对应 LCD_CAM_CAM_CLKM_DIV_B 的值，a 对应 LCD_CAM_CAM_CLKM_DIV_A 的值。对于整数分频，LCD_CAM
_CAM_CLKM_DIV_A 和 LCD_CAM_CAM_CLKM_DIV_B 都清零；对于小数分频，
LCD_CAM_CAM_CLKM_DIV_B 的值应小于 LCD_CAM_CAM_CLKM_DIV_A 的值。

29.3.4 LCD_CAM 模块复位

LCD_CAM 模块中各个单元可通过配置下列寄存器进行复位：

- 发送控制单元 (LCD_Ctrl) 及其格式转换模块 (RGB/YCbCr Converter)：可配置 LCD_CAM_LCD_RESET 进行复位；
- 接收控制单元 (Camera_Ctrl) 及其格式转换模块 (RGB/YCbCr Converter)：可配置 LCD_CAM_CAM_RESET 进行复位；
- Async Tx FIFO：可配置 LCD_CAM_LCD_AFIFO_RESET 进行复位；
- Async Rx FIFO：可配置 LCD_CAM_CAM_AFIFO_RESET 进行复位。

注意：

- 上述复位寄存器均为硬件自清，即写 1 后，硬件会自动清零；
- 在模块和 FIFO 复位之前，需要先配置 LCD 或 Camera 模块时钟。

29.3.5 LCD_CAM 数据格式控制

29.3.5.1 LCD 数据格式控制

在使用 LCD 发送数据时，配置以下寄存器可以调整来自 DMA 的数据的比特/字节序：

- LCD_CAM_LCD_2BYTE_EN
  - 1:：LCD 输出的数据位宽为 16 个比特；
  - 0：LCD 输出的数据位宽为 8 个比特。
- LCD_CAM_LCD_BIT_ORDER
  - 反转数据位顺序：
    * 在 8-bit 模式下，LCD_DATA_out[7:0] 反转为 LCD_DATA_out[0:7]；
    * 在 16-bit 模式下，LCD_DATA_out[15:0] 反转为 LCD_DATA_out[0:15]。

  - 不反转。

- LCD_CAM_LCD_BYTE_ORDER
  - 1：反转数据字节顺序。仅在 16-bit 模式下有效；
  - 0：不反转。
- LCD_CAM_LCD_8BITS_ORDER
  - 1：两个字节反转位置。仅在 8-bit 模式下有效；
  - 0：不反转。

具体配置方式见下表。

表 29.3-2. LCD 数据格式控制

DMA 数据序列[1]	LCD_CAM_LCD_2BYTE_EN	LCD_CAM_LCD_BIT_ORDER	LCD_CAM_LCD_BYTE_ORDER[2]	LCD_CAM_LCD_8BITS_ORDER[2]	发送数据序列[3,4]
B0,B1,B2,B3	0	0	0	0	{B0}{B1}{B2}{B3}
			0	1	{B1}{B0}{B3}{B2}
		1	0	0	{B0'}{B1'}{B2'}{B3'}
			0	1	{B1'}{B0'}{B3'}{B2'}
	1	0	0	0	{B1,B0}{B3,B2}
			1	0	{B0,B1}{B2,B3}
		1	0	0	{B1',B0'}{B3',B2'}
			1	0	{B0',B1'}{B2',B3'}

[1] DMA 数据序列中的 B0 ~ B3 每一个代表一个字节的数据，靠左的数据对应低地址，即 B0 ~ B3 的地址为从低到高。
[2] 只有表格中的配置组合是有效的，所有表格所示之外的配置组合可能会导致不可预期的数据错误。
[3] 发送数据序列中，每个 {} 中的数据为大端并行数据，各个 {} 中的数据为串行关系，发送顺序为从左至右。以 {B0}{B1}{B2}{B3} 为例，{B0} 中的各个数据位之间为并行关系，而 {B0} 中的数据与 {B1}、{B2}、{B3} 的数据为串行关系。发送顺序为先发送 {B0}。
[4] 发送数据序列中，Bn'[7:0] = Bn[0:7]，其中 n = 0,1,2,3。

说明：
注意：当发送序列为每次发送 1 个字节数据时，LCD_Data_out[7:0] 为有效数据，LCD_Data_out[15:8] 为无效数据。

29.3.5.2 Camera 数据格式控制

在使用 Camera 接收数据时，配置以下寄存器可以调整送往 DMA 的数据的比特/字节序：

- LCD_CAM_CAM_2BYTE_EN
  - 1：接收数据的位宽为 16 个比特；
  - 0：接收数据的位宽为 8 个比特。
- LCD_CAM_CAM_BIT_ORDER
  - 1：反转数据位顺序：
    * 在 8-bit 模式下，CAM_DATA_in[7:0] 反转为 CAM_DATA_in[0:7]；

* 在 16-bit 模式下，CAM_DATA_in[15:0] 反转为 CAM_DATA_in[0:15]。

- 0：不反转。

- LCD_CAM_CAM_BYTE_ORDER
  - 1：反转数据字节顺序。仅在 16-bit 模式下有效；
  - 0：不反转。

具体配置方式见下表。

表 29.3-3. Camera 数据格式控制

接收数据序列[1]	LCD_CAM_CAM_2BYTE_EN	LCD_CAM_CAM_BIT_ORDER[2]	LCD_CAM_CAM_BYTE_ORDER[2]	DMA 数据序列[3,4]
{B0}{B1}{B2}{B3}	0	0	0	B0,B1,B2,B3
		1	0	B0',B1',B2',B3'
{B1,B0}{B3,B2}	1	0	0	B0,B1,B2,B3
			1	B1,B0,B3,B2
		1	0	B0',B1',B2',B3'
			1	B1',B0',B3',B2'

[1] 接收数据序列中，每个 {} 中的数据为大端并行数据，各个 {} 中的数据为串行关系，接收顺序为从左至右。以 {B0}{B1}{B2}{B3} 为例，{B0} 中的各个数据位之间为并行关系，而 {B0} 中的数据与 {B1}、{B2}、{B3} 的数据为串行关系。接收顺序为先接收 {B0}。

[2] 只有表格中的配置组合是有效的，所有表格所示之外的配置组合可能会导致不可预期的数据错误。

[3] DMA 数据序列中的 B0 ~ B3 每一个代表一个字节的数据，靠左的数据对应低地址，即 B0 ~ B3 的地址为从低到高。

[4] DMA 数据序列中，Bn'[7:0] = Bn[0:7]，其中 n = 0,1,2,3。

说明：

注意：当接收序列为每次接收 1 个字节数据时，CAM_Data_in[7:0] 为有效数据，因此用户需要将 CAM_Data_in[7:0] 与主机相连。

29.3.6 YUV-RGB 数据格式转换

ESP32-S3 LCD_CAM 模块支持 YUV-RGB 数据格式互相转换。LCD 模块以及 Camera 模块各有一个数据格式转换模块，其功能包括：

- 支持在 BT601 和 BT709 两种标准下进行格式转换；
- RGB565 (full/limited range) 格式与 YUV422/420/411 (full/limited range) 格式的互相转换；
- YUV422/420/411 (full/limited range) 各种格式的互相转换。

29.3.6.1 YUV 时序

在 ESP32-S3 LCD_CAM 模块中我们作以下规定：

假设 8 个依次待传输像素点对应的 YUV 数据为 $[Y_i, U_i, V_i]$ (i = 1 ~ 8)，则在：

- YUV422 模式下，LCD 依次发送或 Camera 依次接收如下数据：

Y_1	U_1	Y_2	V_2	Y_3	U_3	Y_4	V_4	Y_5	U_5	Y_6	V_6	Y_7	U_7	Y_8	V_8

- YUV420 模式下，LCD 依次发送或 Camera 依次接收如下数据：

Y_1	U_1	Y_2	Y_3	U_3	Y_4	Y_5	V_5	Y_6	Y_7	V_7	Y_8

- YUV411 模式下，LCD 依次发送或 Camera 依次接收如下数据：

Y_1	U_1	Y_2	Y_3	V_3	Y_4	Y_5	U_5	Y_6	Y_7	V_7	Y_8

29.3.6.2 格式转换模块配置流程

Camera 模块中的格式转换模块与 LCD 模块中的格式转换模块完全相同。因此，下文以 LCD 模块中的格式转换模块为例说明配置流程：

- 使能 YUV-RGB 格式转换模块：置位 LCD_CAM_LCD_CONV_BYPASS；
- 配置数据传输模式：配置 LCD_CAM_LCD_CONV_MODE_8BITS_ON 为：
  - 0：使用 16 线数据传输模式
  - 1：使用 8 线数据传输模式
- 配置标准：配置 LCD_CAM_LCD_CONV_PROTOCOL_MODE 为：
  - 0：使用 BT601 标准
  - 1：使用 BT709 标准
- 配置转换模式：

表 29.3-4. 转换模式配置

转换模式	TRANS_MODE[1]	YUV_MODE[2]	YUV2YUV_MODE[3]
RGB565 → YUV422	1	0	3
RGB565 → YUV420	1	1	3
RGB565 → YUV411	1	2	3
YUV422 → RGB565	0	0	3
YUV420 → RGB565	0	1	3
YUV411 → RGB565	0	2	3
YUV422 → YUV420	1	0	1
YUV422 → YUV411	1	0	2
YUV420 → YUV422	1	1	0
YUV420 → YUV411	1	1	2
YUV411 → YUV422	1	2	0
YUV411 → YUV420	1	2	1

[1] 对应 LCD_CAM_LCD_CONV_TRANS_MODE 的值
[2] 对应 LCD_CAM_LCD_CONV_YUV_MODE 的值
[3] 对应 LCD_CAM_LCD_CONV_YUV2YUV_MODE 的值

- 配置输入数据色彩空间：配置 LCD_CAM_LCD_CONV_DATA_IN_MODE 为：
  - 0：有限色彩空间 (limited range)
  - 1：全色彩空间 (full range)
- 配置输出数据色彩空间：配置 LCD_CAM_LCD_CONV_DATA_OUT_MODE 为：
  - 1：全色彩空间 (full range)[1]
  - 0：有限色彩空间 (limited range)[2]

说明：

1. 如果选择全色彩空间，则 RGB 或 YUV 的取值范围为：0 ~ 255；
2. 如果选择有限色彩空间，则
   - RGB 的取值范围为：16 ~ 240；
   - YUV 的取值范围为：
     - Y：16 ~ 240；
     - U-V：16 ~ 235。

29.4 软件配置流程

说明：
LCD 模块和 Camera 模块的寄存器相关配置，均需要通过置位对应的 LCD_CAM_LCD_UPDATE 和 LCD_CAM_CAM_UPDATE 的方式来进行更新，从而将 LCD 寄存器数据从 APB 时钟域同步到 LCD/Camera 时钟域。见下文示例。

29.4.1 软件配置 LCD（RGB 格式）发送流程

软件配置 LCD 以 RGB 格式发送的流程如下：

1. 根据 29.3.3 小节的描述，配置时钟；
2. 根据表 29.3-1 的描述，配置信号管脚；
3. 根据 29.5 小节的描述使能相应的中断；
4. 置位 LCD_CAM_LCD_RGB_MODE_EN，使能 RGB 格式；
5. 配置以下帧格式寄存器，如下图所示：
   - LCD_CAM_LCD_VT_HEIGHT
   - LCD_CAM_LCD_VA_HEIGHT
   - LCD_CAM_LCD_HB_FRONT
   - LCD_CAM_LCD_HT_WIDTH
   - LCD_CAM_LCD_HA_WIDTH
   - LCD_CAM_VB_FRONT
   - LCD_CAM_LCD_VSYNC_WIDTH

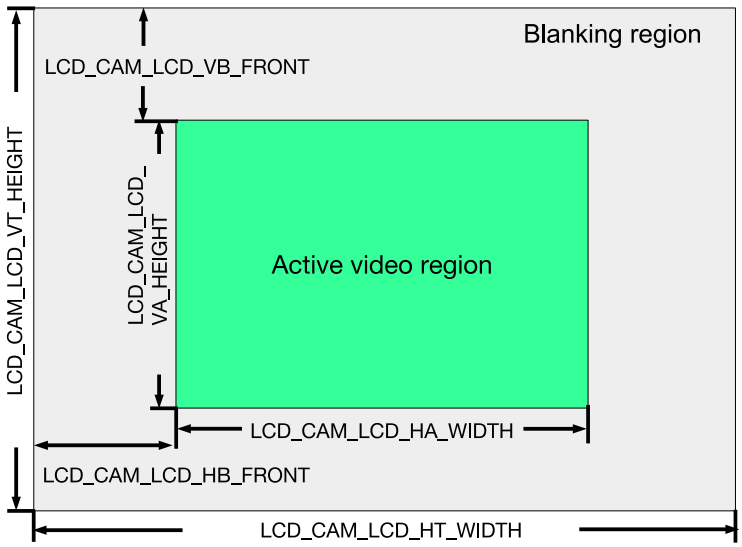


图 29.4-1. LCD 视频帧结构

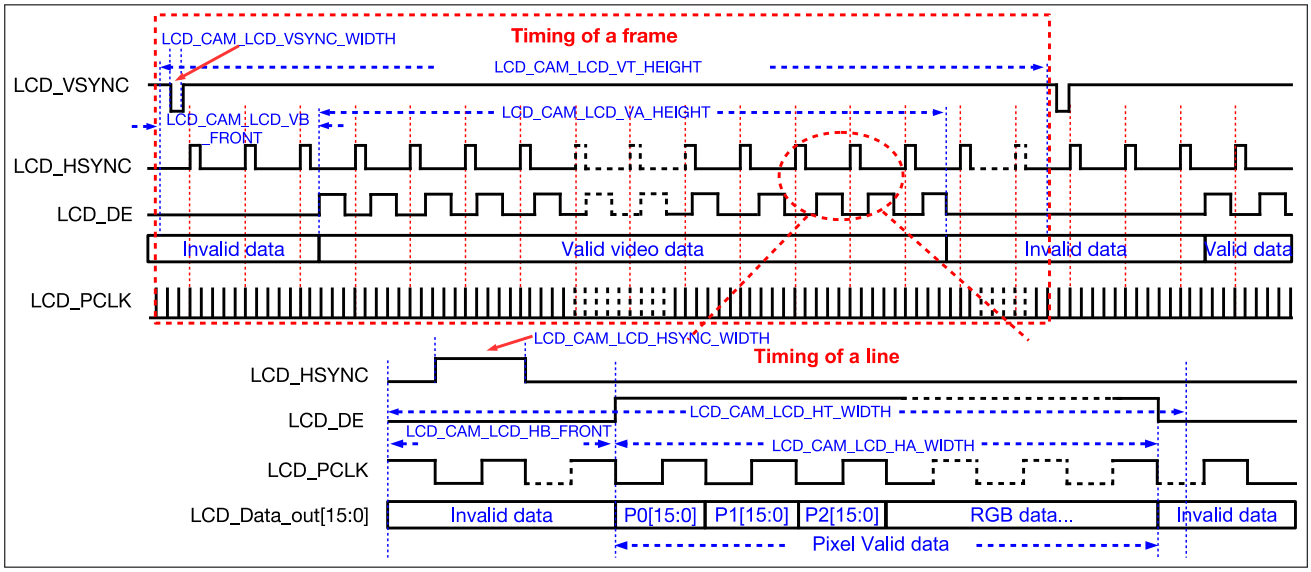


图 29.4-2. LCD 时序（RGB 格式）

> **说明：**
> 配置上图所示的时序参数时，请特别注意参数与寄存器的关系。参数值等于寄存器值 + 1。例如，如果希望 VSYNC 宽度为 1，则需要配置 LCD_CAM_LCD_VSYNC_WIDTH 为 0。更多信息见寄存器描述部分。

6. 置位 LCD_CAM_LCD_UPDATE；
7. 根据 29.3.4 小节的描述，复位发送控制单元 (LCD_Ctrl) 和 Async Tx FIFO；
8. 配置 GDMA 发送链表；
9. 开始发送数据：
   - 等待 LCD 从设备配置完成；
   - 置位 LCD_CAM_LCD_START 开始发送数据。

10. 等待步骤 3 设置的中断信号；
11. 在数据发送的帧间隔，检查 LCD_CAM_LCD_NEXT_FRAME_EN 是否被置位：
    - 如果被置位，则继续发送下一帧数据：置位 LCD_CAM_LCD_UPDATE，重复上述步骤；
    - 如果没有被置位，则 LCD 停止发送数据。
12. 若数据发送完成，则清零 LCD_CAM_LCD_START。

29.4.2 软件配置 LCD（I8080/MOTO6800 格式）发送流程

图 29.4-3 所示为 LCD 时序图（I8080 格式）。

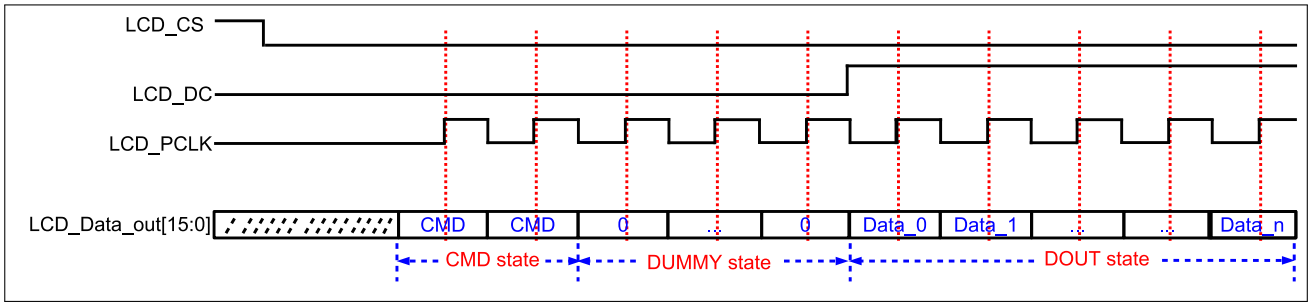


图 29.4-3. LCD 时序（I8080 格式）

软件配置 LCD 以 I8080 格式发送的流程如下：

1. 根据 29.3.3 小节的描述，配置时钟；
2. 根据表 29.3-1 的描述，配置信号管脚；
3. 根据 29.5 小节的描述使能相应的中断；
4. 清零 LCD_CAM_LCD_RGB_MODE_EN；
5. 配置 CMD 阶段，包括 LCD_CAM_LCD_CMD、LCD_CAM_LCD_CMD_2_CYCLE_EN、LCD_CAM_LCD_CMD_VALUE；
6. 配置 DUMMY 阶段，包括 LCD_CAM_LCD_DUMMY、LCD_CAM_LCD_DUMMY_CYCLELEN；
7. 配置 DOUT 阶段，包括 LCD_CAM_LCD_DOUT，根据输出模式不同进行以下配置：
    - 定长输出模式[a]，在 LCD_CAM_LCD_DOUT_CYCLELEN 中配置数据长度；
    - 持续输出模式[b]，则需：
        - 置位 LCD_CAM_LCD_ALWAYS_OUT_EN 使能持续输出模式；
        - 不需要配置 LCD_CAM_LCD_DOUT_CYCLELEN。

说明：

(a) 定长输出模式是指：LCD 发送完成 LCD_CAM_LCD_DOUT_CYCLELEN 中配置的数据长度即结束 LCD 发送。

(b) 持续输出模式是指：LCD 持续发送数据，直到

i. LCD_CAM_LCD_START 被清零；

ii. LCD_CAM_LCD_RESET 被置位；

iii. GDMA 中所有数据被发送完。

8. 配置 CD 信号模式，包括 CD 信号的默认值以及在各阶段的值，详见寄存器 LCD_CAM_LCD_MISC_REG 描述；
9. 置位 LCD_CAM_LCD_UPDATE；
10. 根据 29.3.4 小节的描述，复位发送控制单元 (LCD_Ctrl) 和 Async Tx FIFO；
11. 配置 GDMA 发送链表；
12. 开始发送数据：
    - 等待 LCD 从设备配置完成；
    - 置位 LCD_CAM_LCD_START 开始发送数据。
13. 等待步骤 3 设置的中断信号；
14. 若数据发送完成，则清零 LCD_CAM_LCD_START。

注意：

无论 LCD 配置为 RGB 格式，还是 I8080/MOTO6800 格式，访问内部存储器时均应注意以下限制：

- 如果 LCD 数据总线设置为 8-bit 并行输出时，则
  - 像素时钟频率需小于 80 MHz；
  - 如果同时使用了 YUV-RGB 格式转换，则像素时钟频率需小于 60 MHz。
- 如果 LCD 数据总线设置为 16-bit 并行输出时，则
  - 像素时钟频率需小于 40 MHz；
  - 如果同时使用了 YUV-RGB 格式转换，则像素时钟频率需小于 30 MHz。

29.4.3 软件配置 Camera 接收流程

软件配置 Camera 接收模式的流程如下：

1. 根据 29.3.3 小节的描述，配置时钟。注意，在从机模式下，模块时钟频率需要大于 2 倍的图形传感器的 PCLK 频率；
2. 根据表 29.3-1 的描述，配置信号管脚；
3. 根据控制信号 HSYNC 是否有效置位或清零 LCD_CAM_CAM_VH_DE_MODE_EN；
4. 配置正确的接收通道模式和接收数据模式，置位 LCD_CAM_CAM_UPDATE；
5. 根据 29.3.4 小节的描述，复位接收控制单元 (Camera_Ctrl) 和 Async Rx FIFO；
6. 根据 29.5 小节的描述使能相应的中断；
7. 配置 GDMA 接收链表，并在 LCD_CAM_CAM_REC_DATA_BYTELEN 中配置接收数据长度；
8. 开始接收数据：
   - 在主机模式下，等待从机准备好后，置位 LCD_CAM_CAM_START 开始接收数据；

- 在从机模式下，置位 LCD_CAM_CAM_START，等待主机提供时钟和控制信号后开始接收数据。

9. 接收数据，并存到 ESP32-S3 存储器的指定地址。最终产生步骤 6 中设置的中断。

注意：

- 无论 Camera 配置为主机模式还是从机模式，均应注意以下限制：
  - 如果选择 8-bit 并行输入，则
    * 像素时钟频率需小于 80 MHz；
    * 如果同时使用了 YUV-RGB 格式转换，则像素时钟频率需小于 60 MHz。
  - 如果选择 16-bit 并行输入，则
    * 像素时钟频率需小于 40 MHz；
    * 如果同时使用了 YUV-RGB 格式转换，则像素时钟频率需小于 30 MHz。
- 如果同时外接了 LCD 和 Camera，则访问内部存储器时，需保证接口上最大的数据吞吐率小于 GDMA 总数据带宽，即 80 MB/s。此处默认 APB_CLK 频率为 80 MHz，更多信息见章节 7 复位和时钟。

29.5 LCD_CAM 中断

- LCD_CAM_CAM_HS_INT：当 Camera 接收行数大于等于 LCD_CAM_CAM_LINE_INT_NUM 配置的值 + 1 即触发此中断。
- LCD_CAM_CAM_VSYNC_INT：当 Camera 接收到帧指示信号 VSYNC 即触发此中断。
- LCD_CAM_LCD_TRANS_DONE_INT：当 LCD 发送数据完成即触发此中断。
- LCD_CAM_LCD_VSYNC_INT：当 LCD 发送帧指示信号 VSYNC 即触发此中断。

29.6 寄存器列表

本小节的所有地址均为相对于 LCD_CAM 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

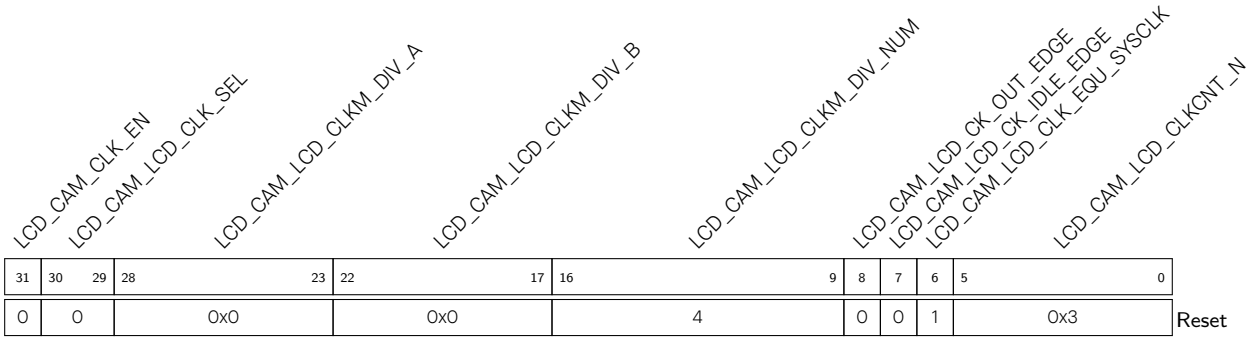
请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
LCD 配置寄存器			
LCD_CAM_LCD_CLOCK_REG	LCD 时钟配置寄存器	0x0000	R/W
LCD_CAM_LCD_RGB_YUV_REG	LCD 数据格式转换配置寄存器	0x0010	R/W
LCD_CAM_LCD_USER_REG	LCD 用户配置寄存器	0x0014	varies
LCD_CAM_LCD_MISC_REG	LCD MISC 配置寄存器	0x0018	varies
LCD_CAM_LCD_CTRL_REG	LCD 信号配置寄存器	0x001C	R/W
LCD_CAM_LCD_CTRL1_REG	LCD 信号配置寄存器 1	0x0020	R/W
LCD_CAM_LCD_CTRL2_REG	LCD 信号配置寄存器 2	0x0024	R/W
LCD_CAM_LCD_CMD_VAL_REG	LCD CMD 值配置寄存器	0x0028	R/W
LCD_CAM_LCD_DLY_MODE_REG	LCD 信号延迟模式配置寄存器	0x0030	R/W
LCD_CAM_LCD_DATA_DOUT_MODE_REG	LCD 数据延迟模式配置寄存器	0x0038	R/W
Camera 配置寄存器			
LCD_CAM_CAM_CTRL_REG	Camera 时钟配置寄存器	0x0004	R/W
LCD_CAM_CAM_CTRL1_REG	Camera 控制寄存器	0x0008	varies
LCD_CAM_CAM_RGB_YUV_REG	Camera 数据格式转换配置寄存器	0x000C	R/W
中断寄存器			
LCD_CAM_LC_DMA_INT_ENA_REG	LCD_CAM GDMA 中断使能寄存器	0x0064	R/W
LCD_CAM_LC_DMA_INT_RAW_REG	LCD_CAM GDMA 原始中断寄存器	0x0068	RO
LCD_CAM_LC_DMA_INT_ST_REG	LCD_CAM GDMA 屏蔽中断状态寄存器	0x006C	RO
LCD_CAM_LC_DMA_INT_CLR_REG	LCD_CAM GDMA 中断清除寄存器	0x0070	WO
版本寄存器			
LCD_CAM_LC_DATE_REG	版本控制寄存器	0x00FC	R/W

29.7 寄存器

本小节的所有地址均为相对于 LCD_CAM 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 29.1. LCD_CAM_LCD_CLOCK_REG (0x0000)



LCD_CAM_LCD_CLKCNT_N LCD_CAM_LCD_CLK_EQU_SYSCLK = 0 时，$f_{LCD\_PCLK} = f_{LCD\_CLK} / (LCD\_CAM\_LCD\_CLKCNT\_N + 1)$。注意，该字段不可配置为 0。(R/W)

LCD_CAM_LCD_CLK_EQU_SYSCLK 1：$f_{LCD\_PCLK} = f_{LCD\_CLK}$；0：$f_{LCD\_PCLK} = f_{LCD\_CLK} / (LCD\_CAM\_LCD\_CLKCNT\_N + 1)$。(R/W)

LCD_CAM_LCD_CK_IDLE_EDGE 1：空闲状态下，LCD_PCLK 为高电平；0：空闲状态下，LCD_PCLK 为低电平。(R/W)

LCD_CAM_LCD_CK_OUT_EDGE 1：在前半个时钟周期内，LCD_PCLK 为高电平；0：在前半个时钟周期内，LCD_PCLK 为低电平。(R/W)

LCD_CAM_LCD_CLKM_DIV_NUM LCD 时钟分频器的整数部分。(R/W)

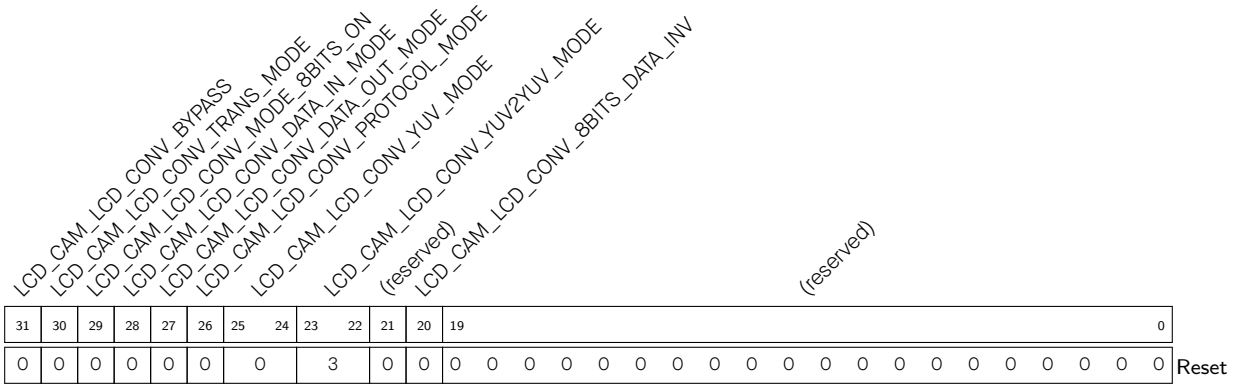
LCD_CAM_LCD_CLKM_DIV_B LCD 时钟分频器小数分频的分子部分。(R/W)

LCD_CAM_LCD_CLKM_DIV_A LCD 时钟分频器小数分频的分母部分。(R/W)

LCD_CAM_LCD_CLK_SEL 选择 LCD 模块时钟源。0：关闭时钟源；1：选择 XTAL_CLK 时钟；2：选择 PLL_D2_CLK 时钟；3：PLL_F160M_CLK。(R/W)

LCD_CAM_CLK_EN 置位此位，则强制打开所有配置寄存器的时钟，不再使用时钟门控。(R/W)

Register 29.2. LCD_CAM_LCD_RGB_YUV_REG (0x0010)



LCD_CAM_LCD_CONV_8BITS_DATA_INV 1：交换两个 8 位的输入数据；0：不交换。(R/W)

LCD_CAM_LCD_CONV_YUV2YUV_MODE 在 YUV-to-YUV 模式下，0：数据被转换为 YUV422 格式；1：数据被转换为 YUV420 格式；2：数据被转换为 YUV411 格式；3：禁用格式转换。仅在 LCD_CAM_LCD_CONV_TRANS_MODE = 1 时有效。(R/W)

LCD_CAM_LCD_CONV_YUV_MODE 在 YUV-to-YUV 模式下或在 YUV-to-RGB 模式下，输入数据具体的 YUV 格式由该字段决定。0：输入数据为 YUV422 格式；1：输入数据为 YUV420 格式；2：输入数据为 YUV411 格式。(R/W)

LCD_CAM_LCD_CONV_PROTOCOL_MODE 0：选择 BT601；1：选择 BT709。(R/W)

LCD_CAM_LCD_CONV_DATA_OUT_MODE 配置输出数据的色彩空间。0：有限色彩空间；1：全色彩空间。(R/W)

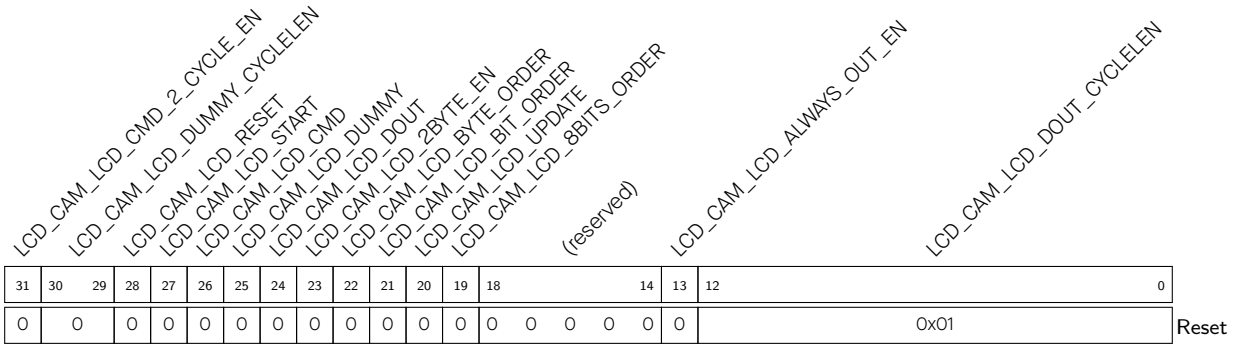
LCD_CAM_LCD_CONV_DATA_IN_MODE 配置输入数据的色彩空间。0：有限色彩空间；1：全色彩空间。(R/W)

LCD_CAM_LCD_CONV_MODE_8BITS_ON 配置数据模式。0：16-bit 模式；1：8-bit 模式。(R/W)

LCD_CAM_LCD_CONV_TRANS_MODE 配置格式转换模式。0：转换成 RGB 格式；1：转换成 YUV 格式。(R/W)

LCD_CAM_LCD_CONV_BYPASS 0：旁路格式转换模块；1：使能格式转换模块。(R/W)

Register 29.3. LCD_CAM_LCD_USER_REG (0x0014)



LCD_CAM_LCD_DOUT_CYCLELEN 配置 LCD 模块输出数据阶段 (DOUT) 的周期时长。实际周期时长 = 本字段的值 + 1。(R/W)

LCD_CAM_LCD_ALWAYS_OUT_EN 使能 LCD 模块的持续输出模式。在该模式下，LCD 模块处于 DOUT 阶段时，持续输出数据，直至 LCD_CAM_LCD_START 被清除或 LCD_CAM_LCD_RESET 被置位。(R/W)

LCD_CAM_LCD_8BITS_ORDER 1：两个字节反转位置。仅在 8-bit 模式下有效；0：不反转。(R/W)

LCD_CAM_LCD_UPDATE 1：更新 LCD 寄存器。LCD 寄存器将由硬件清除。0：无关。(R/W)

LCD_CAM_LCD_BIT_ORDER 1：反转数据位顺序。在 8-bit 模式下，LCD_DATA_out[7:0] 反转为 LCD_DATA_out[0:7]；在 16-bit 模式下，LCD_DATA_out[15:0] 反转为 LCD_DATA_out[0:15]。0：不反转。(R/W)

LCD_CAM_LCD_BYTE_ORDER 1：反转数据字节顺序。仅在 16-bit 模式下有效；0：不反转。(R/W)

LCD_CAM_LCD_2BYTE_EN 1：LCD 输出数据的位宽为 16 个比特；0：LCD 输出数据的数据位宽为 8 个比特。(R/W)

LCD_CAM_LCD_DOUT 1：LCD 启动后，开始以 LCD 模式发送数据；0：禁用该功能。(R/W)

LCD_CAM_LCD_DUMMY 1：LCD 启动后，使能 LCD 模式下的 DUMMY 阶段。0：禁用该功能。(R/W)

LCD_CAM_LCD_CMD 1：LCD 启动后，开始以 LCD 模式发送命令；0：禁用该功能。(R/W)

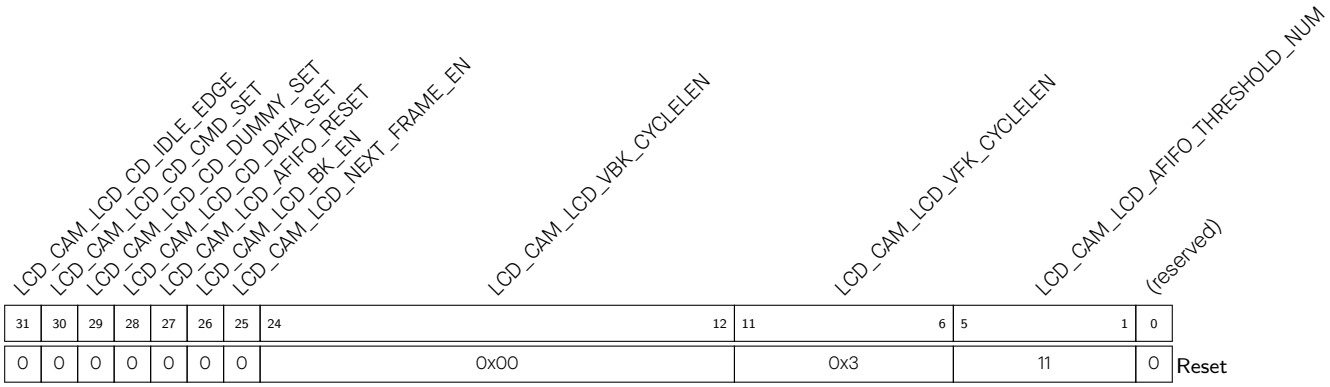
LCD_CAM_LCD_START LCD 开始发送数据使能信号，高电平有效。(R/W)

LCD_CAM_LCD_RESET LCD 模块复位寄存器。(WO)

LCD_CAM_LCD_DUMMY_CYCLELEN 设置 DUMMY 阶段的周期。实际周期 = 该字段的值 + 1。(R/W)

LCD_CAM_LCD_CMD_2_CYCLE_EN 设置 Command 阶段的周期。1：两个周期；0：一个周期。(R/W)

Register 29.4. LCD_CAM_LCD_MISC_REG (0x0018)



LCD_CAM_LCD_AFIFO_THRESHOLD_NUM 设置 Async Tx FIFO 满的阈值。(R/W)

LCD_CAM_LCD_VFK_CYCLELEN 在非 LCD RGB 模式下，设置建立 (Setup) 时间周期。实际周期 = 该字段的值 + 1。(R/W)

LCD_CAM_LCD_VBK_CYCLELEN 在非 LCD RGB 模式下，设置保持 (Hold) 时间周期。实际周期 = 该字段的值 + 1。(R/W)

LCD_CAM_LCD_NEXT_FRAME_EN 1：当前一帧数据发送完成后，继续发送下一帧数据；0：当前一帧数据发送完成后，LCD 停止工作。(R/W)

LCD_CAM_LCD_BK_EN 1：LCD 发送数据时，允许有消隐区；0：无消隐区。(R/W)

LCD_CAM_LCD_AFIFO_RESET Async Tx FIFO 复位信号。(WO)

LCD_CAM_LCD_CD_DATA_SET 1：当 LCD 状态 (lcd_st[2:0]) 处于 LCD DOUT 阶段时，LCD_CD = !LCD_CAM_LCD_CD_IDLE_EDGE；0：LCD_CD = LCD_CAM_LCD_CD_IDLE_EDGE。(R/W)

LCD_CAM_LCD_CD_DUMMY_SET 1：当 LCD 状态 (lcd_st[2:0]) 处于 LCD DUMMY 阶段时，LCD_CD = !LCD_CAM_LCD_CD_IDLE_EDGE；0：LCD_CD = LCD_CAM_LCD_CD_IDLE_EDGE。(R/W)

LCD_CAM_LCD_CD_CMD_SET 1：当 LCD 状态 (lcd_st[2:0]) 处于 LCD CMD 阶段时，LCD_CD = !LCD_CAM_LCD_CD_IDLE_EDGE；0：LCD_CD = LCD_CAM_LCD_CD_IDLE_EDGE。(R/W)

LCD_CAM_LCD_CD_IDLE_EDGE LCD_CD 的默认值。(R/W)

Register 29.5. LCD_CAM_LCD_CTRL_REG (0x001C)



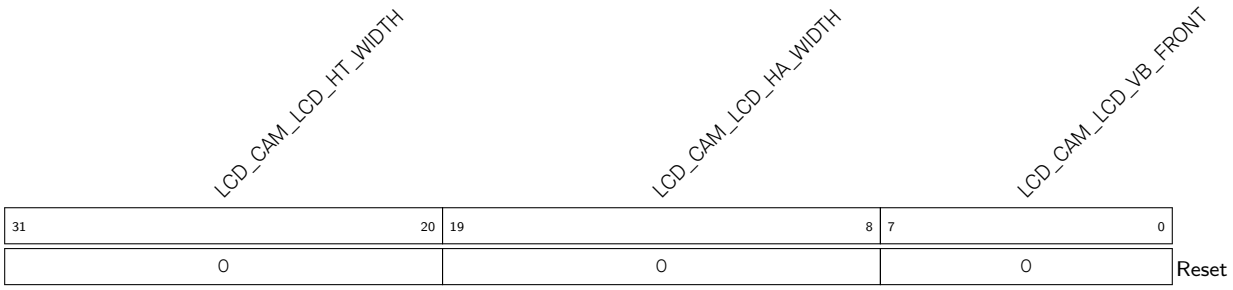
LCD_CAM_LCD_HB_FRONT 配置 (HSYNC_POSITION + HSYNC_WIDTH + 水平后肩) 的值。(R/W)

LCD_CAM_LCD_VA_HEIGHT 帧有效垂直高度。(R/W)

LCD_CAM_LCD_VT_HEIGHT 帧总垂直高度。(R/W)

LCD_CAM_LCD_RGB_MODE_EN 1：使能 RGB 模式，输入 VSYNC、HSYNC 和 DE 信号。(R/W)

Register 29.6. LCD_CAM_LCD_CTRL1_REG (0x0020)

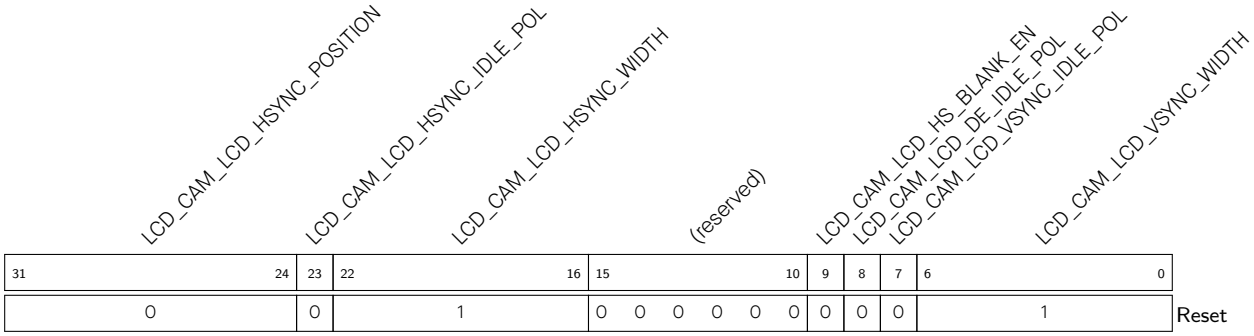


LCD_CAM_LCD_VB_FRONT 配置 (VSYNC_WIDTH + 垂直后肩) 的值。(R/W)

LCD_CAM_LCD_HA_WIDTH 帧有效水平宽度。(R/W)

LCD_CAM_LCD_HT_WIDTH 帧总水平宽度。(R/W)

Register 29.7. LCD_CAM_LCD_CTRL2_REG (0x0024)



LCD_CAM_LCD_VSYNC_WIDTH LCD_VSYNC 的有效脉冲宽度。(R/W)

LCD_CAM_LCD_VSYNC_IDLE_POL LCD_VSYNC 空闲值。(R/W)

LCD_CAM_LCD_DE_IDLE_POL LCD_DE 空闲值。(R/W)

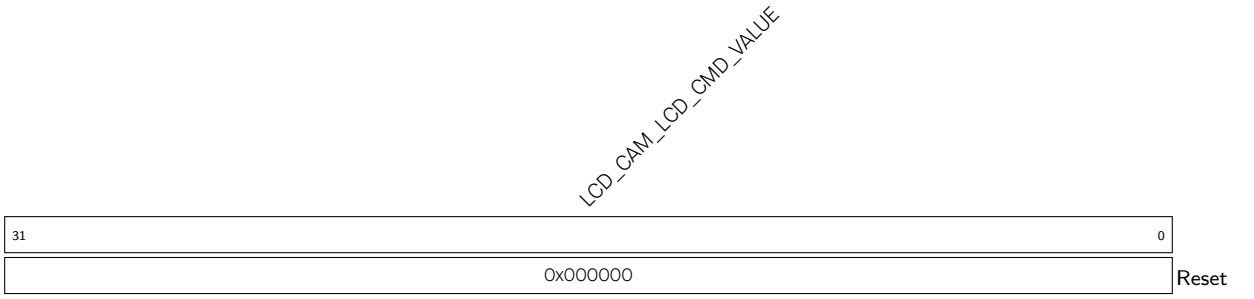
LCD_CAM_LCD_HS_BLANK_EN 1：RGB 模式下，LCD_HSYNC 可在场消隐行输出；0：RGB 模式下，LCD_HSYNC 仅在有效的视频行输出。(R/W)

LCD_CAM_LCD_HSYNC_WIDTH LCD_HSYNC 有效脉冲宽度。参数值 = 寄存器值 + 1。(R/W)

LCD_CAM_LCD_HSYNC_IDLE_POL LCD_HSYNC 空闲值。(R/W)

LCD_CAM_LCD_HSYNC_POSITION LCD_HSYNC 有效脉冲位置。参数值 = 寄存器值 + 1。单位为像素。(R/W)

Register 29.8. LCD_CAM_LCD_CMD_VAL_REG (0x0028)



LCD_CAM_LCD_CMD_VALUE LCD 写命令值。(R/W)

Register 29.9. LCD_CAM_LCD_DLY_MODE_REG (0x0030)

(reserved)	LCD_CAM_LCD_VSYNC_MODE	LCD_CAM_LCD_HSYNC_MODE	LCD_CAM_LCD_DE_MODE	LCD_CAM_LCD_CD_MODE	
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	0x0	0x0	0x0	0x0	Reset

LCD_CAM_LCD_CD_MODE 配置 LCD_CD 输出信号的延迟模式。0：无延迟；1：在 LCD_CLK 上升沿延迟；2：在 LCD_CLK 下降沿延迟。(R/W)

LCD_CAM_LCD_DE_MODE 配置 LCD_DE 输出信号的延迟模式。0：无延迟；1：在 LCD_CLK 上升沿延迟；2：在 LCD_CLK 下降沿延迟。(R/W)

LCD_CAM_LCD_HSYNC_MODE 配置 LCD_HSYNC 输出信号的延迟模式。0：无延迟；1：在 LCD_CLK 上升沿延迟；2：在 LCD_CLK 下降沿延迟。(R/W)

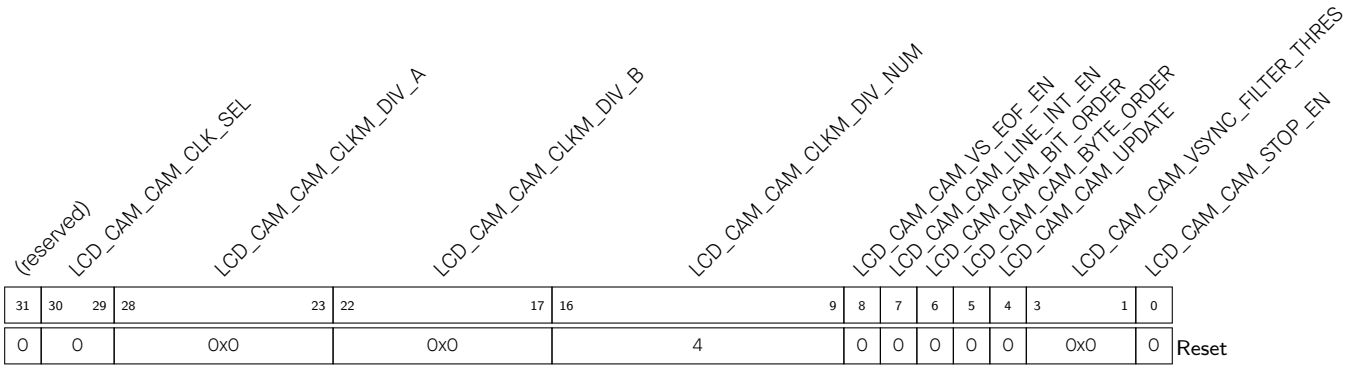
LCD_CAM_LCD_VSYNC_MODE 配置 LCD_VSYNC 输出信号的延迟模式。0：无延迟；1：在 LCD_CLK 上升沿延迟；2：在 LCD_CLK 下降沿延迟。(R/W)

Register 29.10. LCD_CAM_LCD_DATA_DOUT_MODE_REG (0x0038)

LCD_CAM_DOUT15_MODE	LCD_CAM_DOUT14_MODE	LCD_CAM_DOUT13_MODE	LCD_CAM_DOUT12_MODE	LCD_CAM_DOUT11_MODE	LCD_CAM_DOUT10_MODE	LCD_CAM_DOUT9_MODE	LCD_CAM_DOUT8_MODE	LCD_CAM_DOUT7_MODE	LCD_CAM_DOUT6_MODE	LCD_CAM_DOUT5_MODE	LCD_CAM_DOUT4_MODE	LCD_CAM_DOUT3_MODE	LCD_CAM_DOUT2_MODE	LCD_CAM_DOUT1_MODE	LCD_CAM_DOUT0_MODE	
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	
0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	Reset

LCD_CAM_DOUT*n*_MODE (*n* = 0 - 15) 配置输出数据位 (*n*) 的延迟模式。0：无延迟；1：在 LCD_CLK 上升沿延迟；2：在 LCD_CLK 下降沿延迟。(R/W)

Register 29.11. LCD_CAM_CAM_CTRL_REG (0x0004)



LCD_CAM_CAM_STOP_EN Camera 停止信号使能位。1：GDMA RX FIFO 存满后，Camera 停止工作；0：Camera 不停止工作。(R/W)

LCD_CAM_CAM_VSYNC_FILTER_THRES 设置 CAM_VSYNC 信号的滤波阈值。(R/W)

LCD_CAM_CAM_UPDATE 1：更新 Camera 寄存器，并由硬件清零。0：不使用该功能。(R/W)

LCD_CAM_CAM_BYTE_ORDER 1：反转数据字节顺序。仅在 16-bit 模式下有效。0：不反转。(R/W)

LCD_CAM_CAM_BIT_ORDER 1：反转数据位顺序。在 8-bit 模式下，CAM_DATA_in[7:0] 反转为 CAM_DATA_in[0:7]；在 16-bit 模式下，CAM_DATA_in[15:0] 反转为 CAM_DATA_in[0:15]。0：不反转。(R/W)

LCD_CAM_CAM_LINE_INT_EN 1：使能 LCD_CAM_CAM_HS_INT 中断；0：禁用。(R/W)

LCD_CAM_CAM_VS_EOF_EN 1：CAM_VSYNC 用于控制生成 in_suc_eof；0：in_suc_eof 由 LCD_CAM_CAM_REC_DATA_BYTELEN 控制。(R/W)

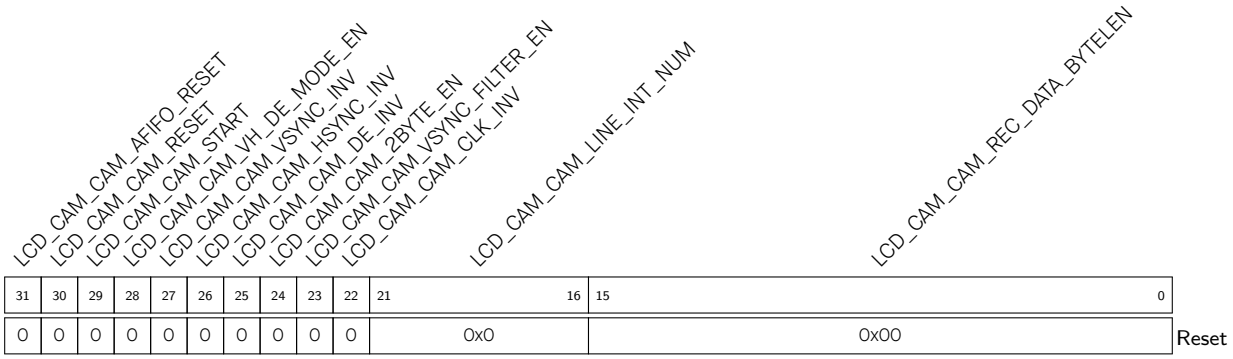
LCD_CAM_CAM_CLKM_DIV_NUM Camera 时钟分频器的整数部分。(R/W)

LCD_CAM_CAM_CLKM_DIV_B Camera 时钟分频器小数分频的分子部分。(R/W)

LCD_CAM_CAM_CLKM_DIV_A Camera 时钟分频器小数分频的分母部分。(R/W)

LCD_CAM_CAM_CLK_SEL 选择 Camera 时钟源。0：关闭时钟；1：选择 XTAL_CLK；2：选择 PLL_D2_CLK；3：选择 PLL_F160M_CLK。(R/W)

Register 29.12. LCD_CAM_CAM_CTRL1_REG (0x0008)



LCD_CAM_CAM_REC_DATA_BYTELEN 设置 Camera 触发 GDMA in_suc_eof_int 中断的接收数据字节长度。接收的数据长度 = 该字段的值 + 1。(R/W)

LCD_CAM_CAM_LINE_INT_NUM 设置触发 LCD_CAM_CAM_HS_INT 中断的视频行数。视频行数 = 该字段 + 1。(R/W)

LCD_CAM_CAM_CLK_INV 1：反转 CAM_PCLK 输入信号；0：不反转。(R/W)

LCD_CAM_CAM_VSYNC_FILTER_EN 1：使能 CAM_VSYNC 滤波功能；0：不使用该功能。(R/W)

LCD_CAM_CAM_2BYTE_EN 1：接收数据的位宽为 16 个比特；0：接收数据的位宽为 8 个比特。(R/W)

LCD_CAM_CAM_DE_INV CAM_DE 反转使能信号，高电平有效。(R/W)

LCD_CAM_CAM_HSYNC_INV CAM_HSYNC 反转使能信号，高电平有效。(R/W)

LCD_CAM_CAM_VSYNC_INV CAM_VSYNC 反转使能信号，高电平有效。(R/W)

LCD_CAM_CAM_VH_DE_MODE_EN 1：输入控制信号为 CAM_DE 和 CAM_HSYNC，且 CAM_VSYNC 为 1。0：输入控制信号为 CAM_DE 和 CAM_VSYNC，且 CAM_HSYNC 和 CAM_DE 均为 1。(R/W)

LCD_CAM_CAM_START Camera 模块使能信号。(R/W)

LCD_CAM_CAM_RESET Camera 模块复位信号。(WO)

LCD_CAM_CAM_AFIFO_RESET Camera Async Rx FIFO 复位信号。(WO)

Register 29.13. LCD_CAM_CAM_RGB_YUV_REG (0x000C)

Bits	Field	Reset
31	LCD_CAM_CAM_CONV_BYPASS	0
30	LCD_CAM_CAM_CONV_TRANS_MODE	0
29	LCD_CAM_CAM_CONV_MODE_8BITS_ON	0
28	LCD_CAM_CAM_CONV_DATA_IN_MODE	0
27	LCD_CAM_CAM_CONV_DATA_OUT_MODE	0
26	LCD_CAM_CAM_CONV_PROTOCOL_MODE	0
25:24	LCD_CAM_CAM_CONV_YUV_MODE	0
23:22	LCD_CAM_CAM_CONV_YUV2YUV_MODE	3
21	LCD_CAM_CAM_CONV_8BITS_DATA_INV	0
20:0	(reserved)	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

LCD_CAM_CAM_CONV_8BITS_DATA_INV 1：反转每两个 8-bit 输入数据；0：禁用该功能。(R/W)

LCD_CAM_CAM_CONV_YUV2YUV_MODE 在 YUV-to-YUV 模式下，0：数据被转换为 YUV422 格式；1：数据被转换为 YUV420 格式；2：数据被转换为 YUV411 格式；3：禁用格式转换。仅在 LCD_CAM_CAM_CONV_TRANS_MODE = 1 时有效。(R/W)

LCD_CAM_CAM_CONV_YUV_MODE 在 YUV-to-YUV 模式下或在 YUV-to-RGB 模式下，输入数据具体的 YUV 格式由该字段决定。0：输入数据为 YUV422 格式；1：输入数据为 YUV420 格式；2：输入数据为 YUV411 格式。(R/W)

LCD_CAM_CAM_CONV_PROTOCOL_MODE 0：选择 BT601；1：选择 BT709。(R/W)

LCD_CAM_CAM_CONV_DATA_OUT_MODE 配置输出数据的色彩空间。0：有限色彩空间；1：全色彩空间。(R/W)

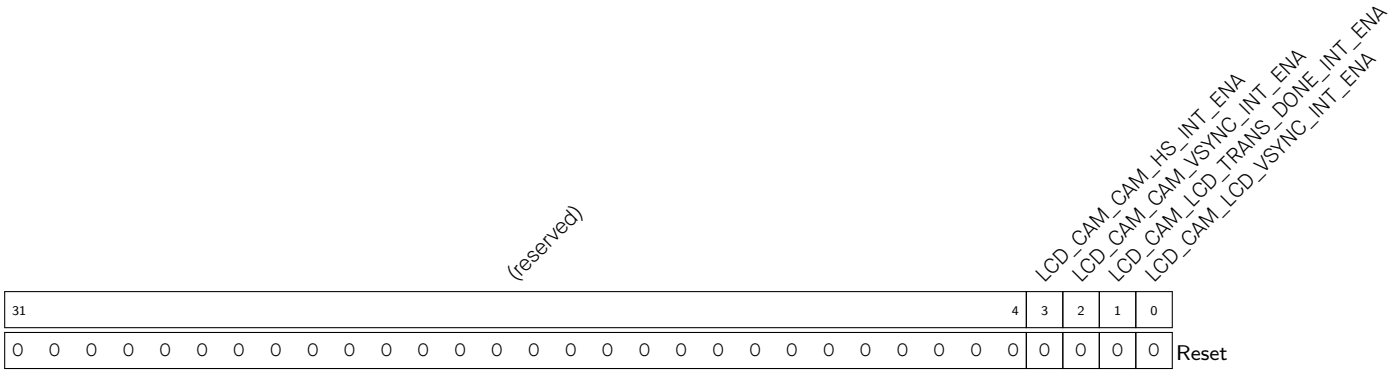
LCD_CAM_CAM_CONV_DATA_IN_MODE 配置输出数据的色彩空间。0：有限色彩空间；1：全色彩空间。(R/W)

LCD_CAM_CAM_CONV_MODE_8BITS_ON 配置数据模式。0：16-bit 模式；1：8-bit 模式。(R/W)

LCD_CAM_CAM_CONV_TRANS_MODE 配置格式转换模式。0：转换成 RGB 格式；1：转换成 YUV 格式。(R/W)

LCD_CAM_CAM_CONV_BYPASS 0：旁路格式转换模块；1：使能格式转换模块。(R/W)

Register 29.14. LCD_CAM_LC_DMA_INT_ENA_REG (0x0064)



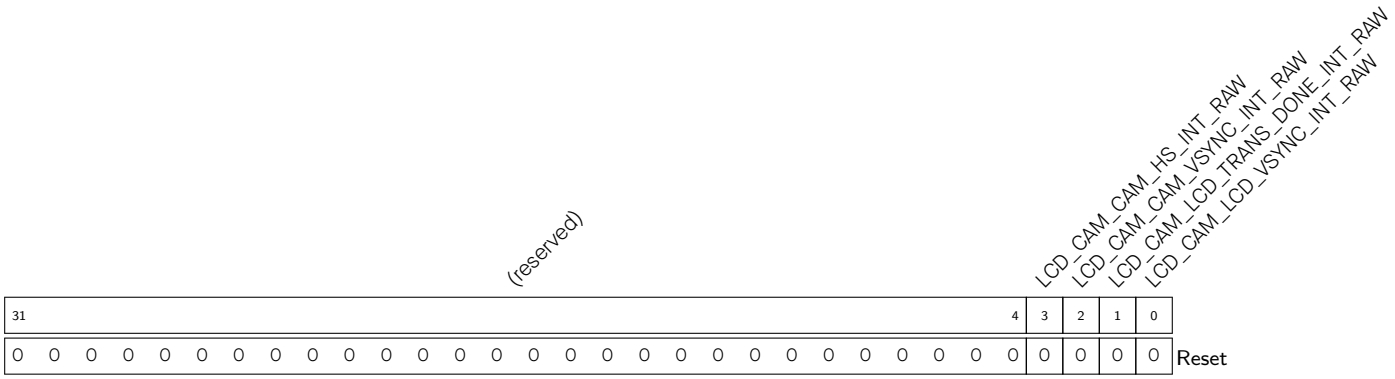
LCD_CAM_LCD_VSYNC_INT_ENA LCD_CAM_LCD_VSYNC_INT 的中断使能位。(R/W)

LCD_CAM_LCD_TRANS_DONE_INT_ENA LCD_CAM_LCD_TRANS_DONE_INT 的中断使能位。(R/W)

LCD_CAM_CAM_VSYNC_INT_ENA LCD_CAM_CAM_VSYNC_INT 的中断使能位。(R/W)

LCD_CAM_CAM_HS_INT_ENA LCD_CAM_CAM_HS_INT 的中断使能位。(R/W)

Register 29.15. LCD_CAM_LC_DMA_INT_RAW_REG (0x0068)



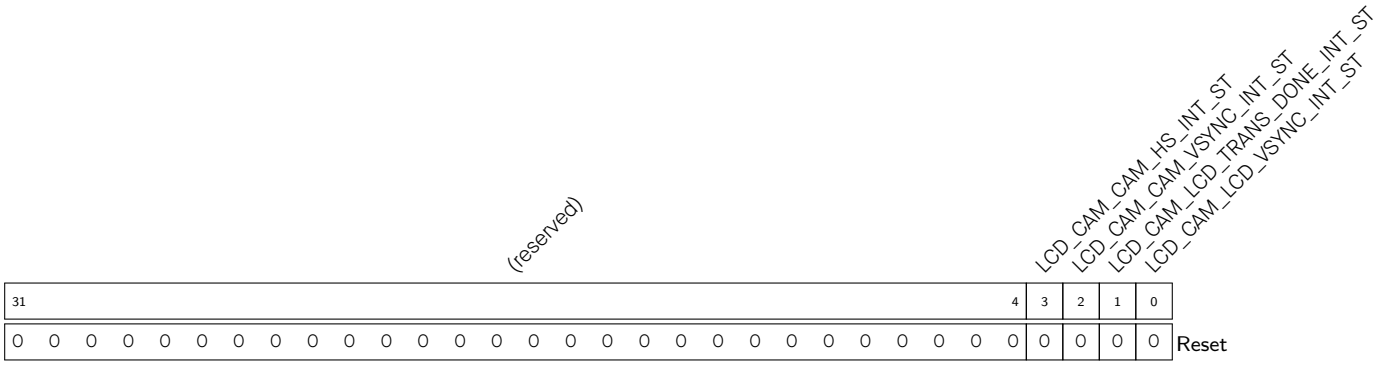
LCD_CAM_LCD_VSYNC_INT_RAW LCD_CAM_LCD_VSYNC_INT 的原始中断位。(RO)

LCD_CAM_LCD_TRANS_DONE_INT_RAW LCD_CAM_LCD_TRANS_DONE_INT 的原始中断位。(RO)

LCD_CAM_CAM_VSYNC_INT_RAW LCD_CAM_CAM_VSYNC_INT 的原始中断位。(RO)

LCD_CAM_CAM_HS_INT_RAW LCD_CAM_CAM_HS_INT 的原始中断位。(RO)

Register 29.16. LCD_CAM_LC_DMA_INT_ST_REG (0x006C)



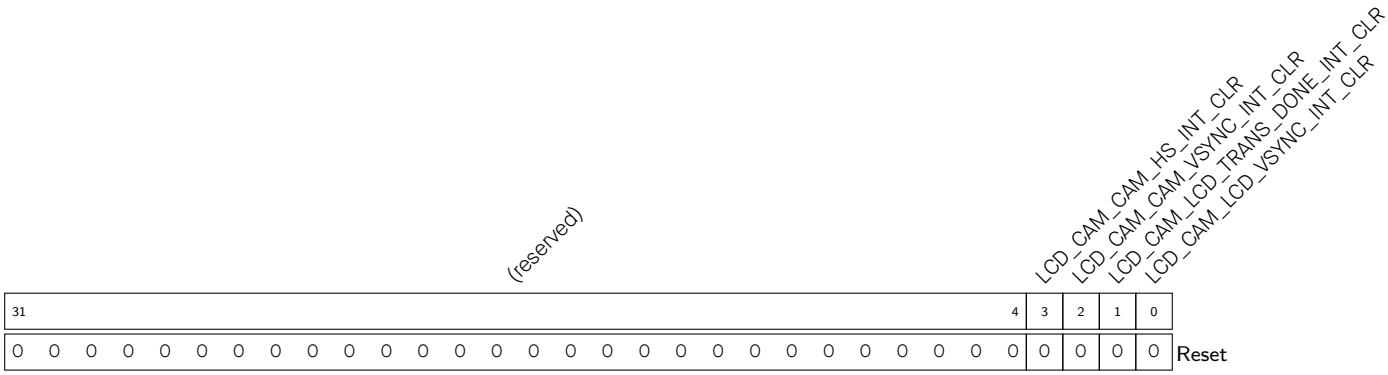
LCD_CAM_LCD_VSYNC_INT_ST LCD_CAM_LCD_VSYNC_INT 的中断状态位。(RO)

LCD_CAM_LCD_TRANS_DONE_INT_ST LCD_CAM_LCD_TRANS_DONE_INT 的中断状态位。(RO)

LCD_CAM_CAM_VSYNC_INT_ST LCD_CAM_CAM_VSYNC_INT 的中断状态位。(RO)

LCD_CAM_CAM_HS_INT_ST LCD_CAM_CAM_HS_INT 的中断状态位。(RO)

Register 29.17. LCD_CAM_LC_DMA_INT_CLR_REG (0x0070)



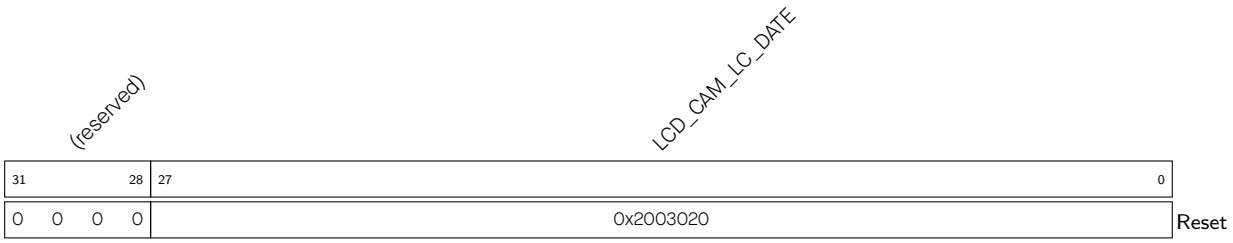
LCD_CAM_LCD_VSYNC_INT_CLR LCD_CAM_LCD_VSYNC_INT 的中断清除位。(WO)

LCD_CAM_LCD_TRANS_DONE_INT_CLR LCD_CAM_LCD_TRANS_DONE_INT 的中断清除位。(WO)

LCD_CAM_CAM_VSYNC_INT_CLR LCD_CAM_CAM_VSYNC_INT 的中断清除位。(WO)

LCD_CAM_CAM_HS_INT_CLR LCD_CAM_CAM_HS_INT 的中断清除位。(WO)

Register 29.18. LCD_CAM_LC_DATE_REG (0x00FC)



LCD_CAM_LC_DATE 版本控制寄存器。(R/W)

第 30 章

SPI 控制器 (SPI)

30.1 概述

串行外设接口 (SPI) 是一种同步串行接口，可用于与外围设备进行通信。ESP32-S3 芯片集成了四个 SPI 控制器：

- SPI0
- SPI1
- 通用 SPI2，即 GP-SPI2
- 和通用 SPI3，即 GP-SPI3

SPI0 和 SPI1 控制器主要供内部使用以访问外部 flash 及 PSRAM。本章节主要介绍 GP-SPI 控制器，即 GP-SPI2 和 GP-SPI3。**除非另有说明，否则在本章节中，GP-SPI 同时指 GP-SPI2 和 GP-SPI3**。

30.2 术语

为了更好地说明 GP-SPI 的功能，本章使用了以下术语。

主机模式	GP-SPI 用作 SPI 主机，发起 SPI 传输事务。
从机模式	GP-SPI 用作 SPI 从机，当其 CS 被拉低时，与 SPI 主机进行数据传输。
MISO	主机输入，从机输出。数据从从机发送至主机。
MOSI	主机输出，从机输入。数据从主机发送至从机。
传输事务	一次完整的传输事务：主机拉低从机的 CS 线，开始传输数据，然后再拉高从机的 CS 线。传输事务为原子操作，即不可打断。
SPI 传输	SPI 主机与从机完成数据交换的一次完整过程。一次 SPI 传输可以包含一个或多个 SPI 传输事务。
单次传输	在这种传输模式下，仅包含一次传输事务。
CPU 控制的传输	由 CPU 控制，SPI_W0_REG ~ SPI_W15_REG 与 SPI 设备之间的数据传输。
DMA 控制的传输	由 DMA 引擎控制，DMA 与 SPI 设备之间的数据传输。
分段配置传输	主机模式下，DMA 控制的数据传输。此类传输包含多个传输事务（分段），每个传输事务均可独立配置。
从机连续传输	从机模式下，DMA 控制的数据传输。此类传输包含多个传输事务（分段）。
全双工	主机与从机之间的发送线和接收线各自独立，发送数据和接收数据同时进行。

半双工	主机和从机只能有一方先发送数据，另一方接收数据。发送数据和接收数据不能同时进行。
四线全双工	四线包括：时钟线、片选线和两条数据线。其中，可使用两条数据线同时发送和接收数据。
四线半双工	四线包括：时钟线、片选线和两条数据线。其中，分时使用两条数据线，不可同时使用。
三线半双工	三线包括：时钟线、片选线和一条数据线。使用数据线分时发送和接收数据。
1-bit SPI	一个时钟周期传输一位数据。
(2-bit) Dual SPI	一个时钟周期传输两个数据位。
Dual Output Read	Dual SPI 的一种数据模式，一个时钟周期可传输一位命令、或一位地址、或两位数据。
Dual I/O Read	Dual SPI 的另外一种数据模式，一个时钟周期可传输一位命令、或两位地址、或两位数据。
(4-bit) Quad SPI	一个时钟周期传输四个数据位。
Quad Output Read	Quad SPI 的一种数据模式，一个时钟周期可传输一位命令、或一位地址、或四位数据。
Quad I/O Read	Quad SPI 的一种数据模式，一个时钟周期可传输一位命令、或四位地址、或四位数据。
QPI	一个时钟周期可传输四位命令、或四位地址、或四位数据。
(8-bit) Octal SPI	一个时钟周期传输八个数据位。
Octal Output Read	Octal SPI 的一种数据模式，一个时钟周期可传输一位命令、或一位地址、或八位数据。
Octal I/O Read	Octal SPI 的另外一种数据模式，一个时钟周期可传输一位命令、或八位地址、或八位数据。
OPI	一个时钟周期可传输八位命令、或八位地址、或八位数据。
FSPI	Fast SPI，GP-SPI2 输入输出信号的前缀。FSPI 总线信号可通过 GPIO 交换矩阵或 IO MUX 与 GPIO 管脚相连。
SPI3	GP-SPI3 输入输出信号的前缀。SPI3 总线信号仅可通过 GPIO 交换矩阵连接到 GPIO 管脚。

30.3 特性

GP-SPI 具体以下特性：

- 支持主机模式和从机模式
- 支持半双工通信和全双工通信
- 支持 CPU 控制的传输模式以及 DMA 控制的传输模式
- 支持多种数据模式：
  - **GP-SPI2**：
    - 1-bit SPI 模式
    - 2-bit Dual SPI 模式
    - 4-bit Quad SPI 模式

    * QPI 模式
    * 8-bit Octal SPI 模式
    * OPI 模式
  - **GP-SPI3**：
    * 1-bit SPI 模式
    * 2-bit Dual SPI 模式
    * 4-bit Quad SPI 模式
    * QPI 模式
- 时钟频率可配置：
  - 在主机模式下：时钟频率可达 80 MHz
  - 在从机模式下：时钟频率可达 60 MHz
- 数据长度可配置：
  - 在主机和从机 CPU 控制的传输模式下：数据长度为 1 ~ 64 B
  - 在主机 DMA 控制的单次传输模式下：数据长度为 1 ~ 32 KB
  - 在主机 DMA 控制的分段配置传输模式下：数据长度字节数无限制
  - 在从机 DMA 控制的单次或连续传输模式下：数据长度字节数无限制
- 数据位的读写顺序可配置
- 为 CPU 控制的传输和 DMA 控制的传输分别提供独立中断
- 时钟极性和相位可配置
- 四种 SPI 时钟模式：模式 0 ~ 模式 3
- 在主机模式下，提供多条 CS 线：
  - **GP-SPI2**：CS0 ~ CS5
  - **GP-SPI3**：CS0 ~ CS2
- 支持访问 SPI 接口的传感器、显示屏控制器、flash 或 RAM 芯片

30.4 架构概览

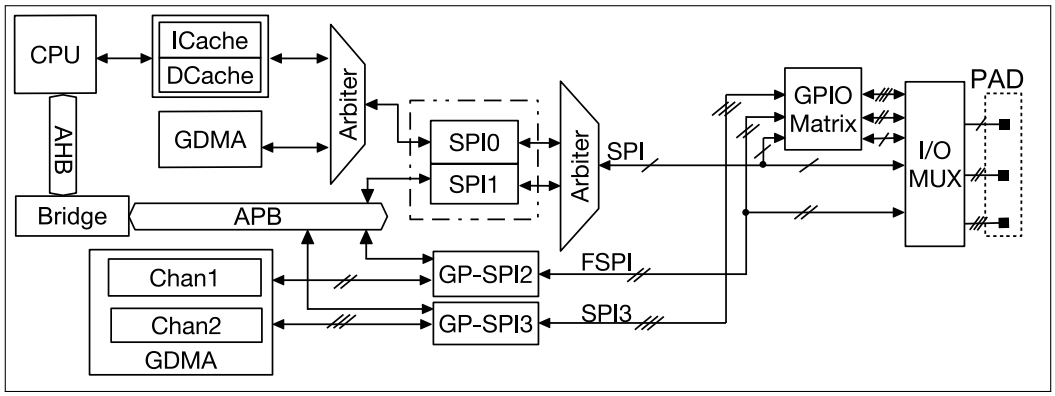


图 30.4-1. SPI 模块概览

图 30.4-1 所示为 SPI 模块的概览。GP-SPI2/GP-SPI3 通过以下方式与 SPI 设备进行数据交换：

- 在 CPU 控制的传输模式下：CPU ↔ GP-SPI2 (GP-SPI3) ↔ SPI 设备
- 在 DMA 控制的传输模式下：GDMA ↔ GP-SPI2 (GP-SPI3) ↔ SPI 设备

GP-SPI2 和 GP-SPI3 输入输出信号的前缀分别为 “FSPI” (Fast SPI) 和 “SPI3”。FSPI 总线信号可通过 GPIO 交换矩阵或 IO MUX 与 GPIO 管脚相连。SPI3 总线信号仅可通过 GPIO 交换矩阵连接到 GPIO 管脚。更多信息，见章节 *6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)*。

GP-SPI3 与 GP-SPI2 功能基本相同。GP-SPI2 功能描述见章节 30.5。GP-SPI2 和 GP-SPI3 的功能差异见章节 30.5.1 和章节 30.9。

30.5 功能描述

30.5.1 数据模式

GP-SPI 可配置成主机或从机模式，采用表 30.5-1 所示的数据模式与其它 SPI 设备进行通信。GP-SPI 用作主机时，表中列出的各种数据模式见章节 30.5.8；GP-SPI 用作从机时，表中列出的各种数据模式见章节 30.5.9。

表 30.5-1. GP-SPI2 和 GP-SPI3 支持的数据模式

数据模式		命令阶段	地址阶段	数据阶段	GP-SPI2	GP-SPI3
1-bit SPI		1-bit	1-bit	1-bit	Y	Y
Dual SPI	Dual Output Read	1-bit	1-bit	2-bit	Y	Y
	Dual I/O Read	1-bit	2-bit	2-bit	Y	Y
Quad SPI	Quad Output Read	1-bit	1-bit	4-bit	Y	Y
	Quad I/O Read	1-bit	4-bit	4-bit	Y	Y
Octal SPI	Octal Output Read	1-bit	1-bit	8-bit	Y	—
	Octal I/O Read	1-bit	8-bit	8-bit	Y	—
QPI		4-bit	4-bit	4-bit	Y	Y
OPI		8-bit	8-bit	8-bit	Y	—

30.5.2 FSPI 总线信号和 SPI3 总线信号描述

FSPI/SPI3 总线信号的功能描述如表 30.5-2。各种 SPI 模式下使用到的信号见表 30.5-3 和表 30.5-4。

表 30.5-2. FSPI/SPI3 总线信号功能描述

FSPI 总线信号	SPI3 总线信号	描述
FSPICLK	SPI3_CLK	主从机模式，输入输出时钟
FSPICS0	SPI3_CS0	主从机模式，输入输出片选信号
FSPICS1 ~ 5	SPI3_CS1 ~ 2	主机模式，输出片选信号
FSPID	SPI3_D	MOSI/SIO0：串行输入输出数据，比特 0
FSPIQ	SPI3_Q	MISO/SIO1：串行输入输出数据，比特 1
FSPIWP	SPI3_WP	SIO2：串行输入输出数据，比特 2
FSPIHD	SPI3_HD	SIO3：串行输入输出数据，比特 3
FSPIIO4 ~ 7	—	SIO4 ~ 7：串行输入输出数据，比特 4 ~ 7
FSPIDQS	—	主机模式下，输出数据屏蔽信号

表 30.5-3. 各种 SPI 模式下使用到的 FSPI 总线信号

FSPI 总线信号	主机模式								从机模式					
	1-bit SPI			Dual SPI	Quad SPI	QPI	Octal SPI	OPI	1-bit SPI			Dual SPI	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	Y	Y
FSPICS0	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
FSPICS1	Y	Y	Y	Y	Y	Y	Y	Y						
FSPICS2	Y	Y	Y	Y	Y	Y	Y	Y						
FSPICS3	Y	Y	Y	Y	Y	Y	Y	Y						
FSPICS4	Y	Y	Y	Y	Y	Y	Y	Y						
FSPICS5	Y	Y	Y	Y	Y	Y	Y	Y						
FSPID	Y	Y	(Y)[3]	Y[4]	Y[5]	Y	Y	Y	Y	Y	(Y)[6]	Y[7]	Y[8]	Y
FSPIQ	Y		(Y)[3]	Y[4]	Y[5]	Y	Y	Y	Y		(Y)[6]	Y[7]	Y[8]	Y
FSPIWP					Y[5]	Y	Y	Y					Y[8]	Y
FSPIHD					Y[5]	Y	Y	Y					Y[8]	Y
FSPIIO4 ~ 7							Y	Y						
FSPIDQS							Y	Y						

[1] FD：全双工

[2] HD：半双工

[3] 一次只使用两个信号中的一个

[4] 两个信号并行使用

[5] 四个信号并行使用

[6] 一次只使用两个信号中的一个

[7] 两个信号并行使用

[8] 四个信号并行使用

表 30.5-4. 各种 SPI 模式下使用到的 SPI3 总线信号

SPI3 总线信号	主机模式						从机模式					
	1-bit SPI			Dual SPI	Quad SPI	QPI	1-bit SPI			Dual SPI	Quad SPI	QPI
	FD[1]	3-line HD[2]	4-line HD				FD	3-line HD	4-line HD			
SPI3_CLK	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
SPI3_CS0	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
SPI3_CS1	Y	Y	Y	Y	Y	Y						
SPI3_CS2	Y	Y	Y	Y	Y	Y						
SPI3_D	Y	Y	(Y)[3]	Y[4]	Y[5]	Y	Y	Y	(Y)[6]	Y[7]	Y[8]	Y
SPI3_Q	Y		(Y)[3]	Y[4]	Y[5]	Y	Y		(Y)[6]	Y[7]	Y[8]	Y
SPI3_WP					Y[5]	Y					Y[8]	Y
SPI3_HD					Y[5]	Y					Y[8]	Y

[1] FD：全双工

[2] HD：半双工

[3] 一次只使用两个信号中的一个

[4] 两个信号并行使用

[5] 四个信号并行使用

[6] 一次只使用两个信号中的一个

[7] 两个信号并行使用

[8] 四个信号并行使用

30.5.3 数据位读/写顺序控制

在主机模式下：

- GP-SPI 主机发送的命令、地址和数据的位顺序由 SPI_WR_BIT_ORDER 控制；
- 接收数据的位顺序由 SPI_RD_BIT_ORDER 控制。

在从机模式下：

- GP-SPI 从机发送数据的位顺序由 SPI_WR_BIT_ORDER 控制；
- 接收的命令、地址和数据的位顺序由 SPI_RD_BIT_ORDER 控制。

表 30.5-5 所示为 SPI_RD/WR_BIT_ORDER 的功能。

表 30.5-5. GP-SPI 主机模式和从机模式下的数据位控制

位模式	FSPI 总线信号	SPI_RD/WR_BIT_ORDER = 0 (MSB)	SPI_RD/WR_BIT_ORDER = 2 (MSB)	SPI_RD/WR_BIT_ORDER = 1 (LSB)	SPI_RD/WR_BIT_ORDER = 3 (LSB)
1-bit 模式	FSPID or FSPIQ	B7→B6→B5→B4→B3→B2→B1→B0	B7→B6→B5→B4→B3→B2→B1→B0	B0→B1→B2→B3→B4→B5→B6→B7	B0→B1→B2→B3→B4→B5→B6→B7
2-bit 模式	FSPIQ	B7→B5→B3→B1	B6→B4→B2→B0	B1→B3→B5→B7	B0→B2→B4→B6
	FSPID	B6→B4→B2→B0	B7→B5→B3→B1	B0→B2→B4→B6	B1→B3→B5→B7
4-bit 模式	FSPIHD	B7→B3	B4→B0	B3→B7	B0→B4
	FSPIWP	B6→B2	B5→B1	B2→B6	B1→B5
	FSPIQ	B5→B1	B6→B2	B1→B5	B2→B6
	FSPID	B4→B0	B7→B3	B0→B4	B3→B7
8-bit 模式	FSPIO7	B7	B7	B0	B0
	FSPIO6	B6	B6	B1	B1
	FSPIO5	B5	B5	B2	B2
	FSPIO4	B4	B4	B3	B3
	FSPIHD	B3	B3	B4	B4
	FSPIWP	B2	B2	B5	B5
	FSPIQ	B1	B1	B6	B6
	FSPID	B0	B0	B7	B7

GoBack

30.5.4 传输方式

GP-SPI 在主机模式和从机模式下支持的传输方式见下表。

表 30.5-6. 主机模式和从机模式下支持的传输方式

模式		CPU 控制的单次传输	DMA 控制的单次传输	DMA 控制的分段配置传输*	DMA 控制的从机连续传输
主机	全双工	Y	Y	Y	–
	半双工	Y	Y	Y	–
从机	全双工	Y	Y	–	Y
	半双工	Y	Y	–	Y

* GP-SPI3 不支持 DMA 控制的分段配置传输。

以下章节将详细介绍上表中所列的各种传输方式。

30.5.5 CPU 控制的数据传输

GP-SPI 提供了 16 个 32-bit 的数据 buffer，即 SPI_W0_REG ~ SPI_W15_REG，见图 30.5-1。CPU 控制的传输表示在该次传输中发送的数据来自数据 buffer 或接收的数据存入数据 buffer。在这种传输方式下，每次传输事务均需要先配置相关寄存器，然后由 CPU 来触发。因此，CPU 控制的传输只能是单次传输，即仅包含一次传输事务。CPU 控制的传输支持全双工通信和半双工通信。

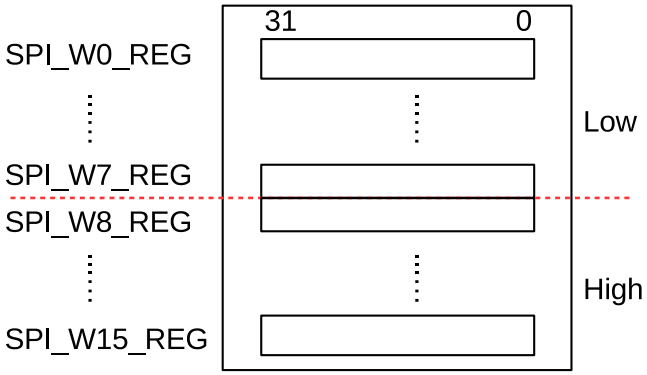


图 30.5-1. CPU 控制的传输中使用的数据 Buffer

30.5.5.1 CPU 控制的主机模式

主机模式下，无论全双工通信还是半双工通信，CPU 控制的数据传输均通过 SPI_W0_REG ~ SPI_W15_REG 完成。此外，用户可通过配置 SPI_USR_MOSI_HIGHPART 和 SPI_USR_MISO_HIGHPART 选择“高位模式”仅使用其中部分寄存器，具体见下方列表描述。

- TX 数据：
  - 未使能高位模式（SPI_USR_MOSI_HIGHPART 置 0）：此时，TX 数据取自 SPI_W0_REG ~ SPI_W15_REG，且每传输一个字节，取 TX 数据的地址即递增 1。如果数据长度大于 64 字节，则 SPI_W_REG ~ SPI_W15_REG 中的数据可能会被多次发送。例如，需要发送 66 个字节（字节 0 ~ 字节 65），则字节 65 的地址为 65 对 64 取模的结果（65 % 64 = 1），即字节 65 和字节 64 将分别取自地址 1 (SPI_W0_REG[15:8]) 和地址 0 (SPI_W0_REG[7:0])。在这种情况下，SPI_W0_REG[15:0] 中存放的数据将被多次发送。

  - 使能高位模式（SPI_USR_MOSI_HIGHPART 置 1）：此时，TX 数据取自 SPI_W8_REG ~ SPI_W15_REG，且每传输一个字节，取 TX 数据的地址即递增 1。如果数据长度大于 32 字节，则 SPI_W8_REG ~ SPI_W15_REG 中的数据将被多次发送。

- RX 数据：
  - 未使能高位模式（SPI_USR_MISO_HIGHPART 置 0）：此时，RX 数据存入 SPI_WO_REG ~ SPI_W15_REG，且每传输一个字节，存 RX 数据的地址即递增 1。如果数据长度大于 64 字节，SPI_WO_REG ~ SPI_W15_REG 中的数据可能被覆盖。例如，需要接收 66 个字节（字节 0 ~ 字节 65），则字节 65 的地址为 65 对 64 取模的结果（65 % 64 = 1），即字节 65 和字节 64 将分别存入地址 1 (SPI_WO_REG[15:8]) 和地址 0 (SPI_WO_REG[7:0])。在这种情况下，SPI_WO_REG[15:0] 中存放的数据将被覆盖。
  - 使能高位模式（SPI_USR_MISO_HIGHPART 置 1）：此时，RX 数据存入 SPI_W8_REG ~ SPI_W15_REG，且每传输一个字节，存 RX 数据的地址即递增 1。如果数据长度大于 32 字节，则 SPI_W8_REG ~ SPI_W15_REG 中的数据将被覆盖。

说明：

- 上述的 TX/RX 数据均按字节寻址。地址 0 为 SPI_WO_REG[7:0]，地址 1 为 SPI_WO_REG[15:8]，以此类推。最大地址 63 为 SPI_W15_REG[31:24]。
- 为避免 TX/RX 数据传输错误，如 TX 数据重复发送或 RX 数据被覆盖等问题，请确保寄存器配置正确。

30.5.5.2 CPU 控制的从机模式

从机模式下，无论全双工通信或半双工通信，CPU 控制的数据传输均通过 SPI_WO_REG ~ SPI_W15_REG 完成，均采用按字节寻址。

- 全双工方式下：SPI_WO_REG ~ SPI_W15_REG 地址从 0 开始，且每传输一个字节，地址即递增 1。如果数据地址大于 63，则 SPI_W15_REG[31:24] 中的数据会被覆盖。
- 半双工方式下：传输格式中 ADDR 的值即为 RX 数据或 TX 数据的起始地址，对应 SPI_WO_REG ~ SPI_W15_REG。每传输一个字节，则 RX 或 TX 地址即递增 1。如果地址大于 63（即大于最高地址：SPI_W15_REG[31:24]），溢出的数据将会被一直存入地址 63，即 SPI_W15_REG[31:24] 的数据被覆盖。

用户可根据具体应用，将 SPI_WO_REG ~ SPI_W15_REG

- 全部用作数据 buffer
- 部分用作数据 buffer，部分用作状态 buffer
- 全部用作状态 buffer

30.5.6 DMA 控制的数据传输

在 DMA 控制的传输中，GDMA RX 模块接收数据，GDMA TX 模块发送数据。主机模式和从机模式均支持这种传输方式。

DMA 控制的传输可以是：

- 一次单次传输，仅包含一次传输事务。GP-SPI 主机模式和从机模式均支持这种单次传输。

- 分段配置传输，包含多个传输事务（即多个分段）。仅有 GP-SPI2 主机模式支持这种分段配置传输。更多信息，见章节 30.5.8.5。
- 从机连续传输，包含多次传输事务。仅有 GP-SPI 从机模式支持这种从机连续传输。更多信息，见章节 30.5.9.3。

DMA 控制的传输只需由 CPU 触发一次即可完成多次传输事务。此类传输一旦被触发，GDMA 引擎从 DMA 链接的内存中发送数据，或将收到的数据存入 DMA 链接的内存中，无需 CPU 的干预。

DMA 控制的传输支持全双工通信、半双工通信以及章节 30.5.8 和章节 30.5.9 所描述的功能。同时，GDMA RX 模块与 GDMA TX 模块互不影响，即支持四种全双工通信：

- 在 DMA 控制模式下接收数据，并在 DMA 控制模式下发送数据；
- 在 DMA 控制模式下接收数据，但在 CPU 控制模式下发送数据；
- 在 CPU 控制模式下接收数据，但在 DMA 控制模式下发送数据；
- 在 CPU 控制模式下接收数据，并在 CPU 控制模式下发送数据。

30.5.6.1 GDMA 配置

- 选择 GDMA 通道 *n*，并配置 GDMA TX/RX 描述符，见章节 3 *通用 DMA 控制器 (GDMA)*；
- 置位 GDMA_INLINK_START_CH*n*/GDMA_OUTLINK_START_CH*n* 启动 GDMA RX/TX 引擎；
- 如果置位 GDMA_OUTLINK_RESTART_CH*n*，则在所有 GDMA TX buffer 用完之前，或在 GDMA TX 引擎重置之前，新的 TX buffer 将会被添加到最后使用中的 TX buffer 结尾；
- GDMA RX buffer 的链接方式与 GDMA TX buffer 的链接方式相同，可通过置位 GDMA_INLINK_START_CH*n* 或 GDMA_INLINK_RESTART_CH*n* 来实现；
- TX 数据长度和 RX 数据长度分别由 GDMA TX buffer 和 RX buffer 决定，TX 数据长度和 RX 数据长度范围无限制；
- 启动 GDMA 前，先初始化 GDMA 接收链表 (inlink) 和发送链表 (outlink)。请置位寄存器 SPI_DMA_CONF_REG 中的 SPI_DMA_RX_ENA 和 SPI_DMA_TX_ENA 位，否则读/写数据将存至或取自寄存器 SPI_W0_REG ~ SPI_W15_REG。

主机模式下，如果置位了 GDMA_IN_SUC_EOF_CH*n*_INT_ENA ，则一次单次传输结束或一次分段配置传输结束，就会触发 GDMA_IN_SUC_EOF_CH*n*_INT 中断。

从机模式下，如果置位了 GDMA_IN_SUC_EOF_CH*n*_INT_ENA，则下表中任一情况均可触发 GDMA_IN_SUC_EOF
_CH*n*_INT 中断。

表 30.5-7. GP-SPI 从机模式下数据传输中断触发条件

传输类型	控制位[1]	控制位[2]	触发条件
从机单次传输	0	0	一次单次传输结束即触发该中断。
	1	0	一次单次传输结束，或接收的数据长度等于 SPI_MS_DATA_BITLEN + 1，即触发该中断。
从机连续传输	0	1	正确接收 CMD7 或 End_SEG_TRANS 命令即触发该中断。

	1	1	正确接收 CMD7 或 End_SEG_TRANS 命令、或接收的数据长度等于 SPI_MS_DATA_BITLEN + 1，即触发该中断。

[1] SPI_RX_EOF_EN
[2] SPI_DMA_SLV_SEG_TRANS_EN

30.5.6.2 GDMA TX/RX Buffer 长度控制

配置的 GDMA TX/RX buffer 长度最好应等于实际传输数据的长度。

- 如果配置的 GDMA TX buffer 长度小于实际传输的数据长度，则多出来的数据将与最后传输的 TX buffer 数据相同。同时触发 SPI_OUTFIFO_EMPTY_ERR_INT 和 GDMA_OUT_EOF_CH*n*_INT 中断。
- 如果配置的 GDMA TX buffer 长度大于实际传输的数据长度，则 TX buffer 中的数据未被完全使用，即使稍后链接了新的 TX buffer，上个 TX buffer 中剩余的数据也将参与后续传输。请特别注意上述情况，或保存未使用的数据并复位 DMA。
- 如果配置的 GDMA RX buffer 长度小于实际传输的数据长度，则多出来的数据将会丢失。同时触发 SPI_INFIFO _FULL_ERR_INT 和 SPI_TRANS_DONE_INT 中断。但不会触发 GDMA_IN_SUC_EOF_CH*n*_INT 中断。
- 如果配置的 GDMA RX buffer 长度大于实际传输的数据长度，则 RX buffer 未被使用的部分被丢弃，下次传输直接使用后面链接的 RX buffer。

30.5.7 GP-SPI 主机模式和从机模式下的数据流控制

GP-SPI 主机模式和从机模式均支持 CPU 控制的数据传输和 DMA 控制的数据传输。CPU 控制的数据传输发生在 SPI_W0_REG ~ SPI_W15_REG 和外围 SPI 设备之间。DMA 控制的数据传输发生在配置好的 GDMA TX/RX buffer 和外围 SPI 设备之间。用户可在传输开始之前，配置 SPI_DMA_RX_ENA 和 SPI_DMA_TX_ENA 来选择需要的传输方式。

30.5.7.1 GP-SPI 功能块图

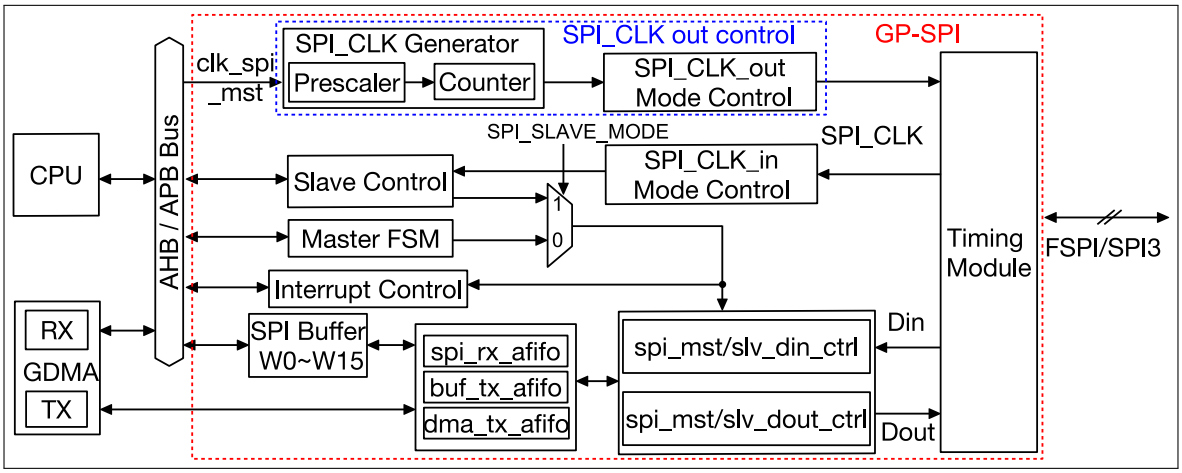


图 30.5-2. GP-SPI 功能块图

图 30.5-2 所示为 GP-SPI 主要的功能模块，包括：

- Master FSM：GP-SPI 的主机状态机。主机模式下支持的所有功能，均由该状态机与寄存器共同控制。
- SPI Buffer: SPI_W0_REG ~ SPI_W15_REG，见图 30.5-1。CPU 控制模式下传输的数据在 SPI buffer 中准备。
- 时序模块 (Timing Module)：捕获 FSPI/SPI3 总线上的数据。
- spi_mst/slv_din/dout_ctrl：用于将 TX/RX 数据转换成字节。
- spi_rx_afifo：暂存接收到的数据。
- buf_tx_afifo：暂存待发送的数据。
- dma_tx_afifo：暂存来自 GDMA 的数据。
- clk_spi_mst：GP-SPI 模块时钟，由 PLL_CLK 分频所得。在 GP-SPI 主机模式下用于生成数据传输以及从机所需的 SPI_CLK 信号。
- SPI_CLK 生成器 (SPI_CLK Generator)：对 clk_spi_mst 进行分频生成 SPI_CLK 信号。分频系数由 SPI_CLKCNT
  _N 和 SPI_CLKDIV_PRE 共同决定。
- SPI_CLK_out Mode Control：发送数据传输以及从机所需的 SPI_CLK 信号。
- SPI_CLK_in Mode Control：当 GP-SPI 用作从机时，用于捕获 SPI 主机发出的 SPI_CLK 信号。

30.5.7.2 主机模式下的数据流控制

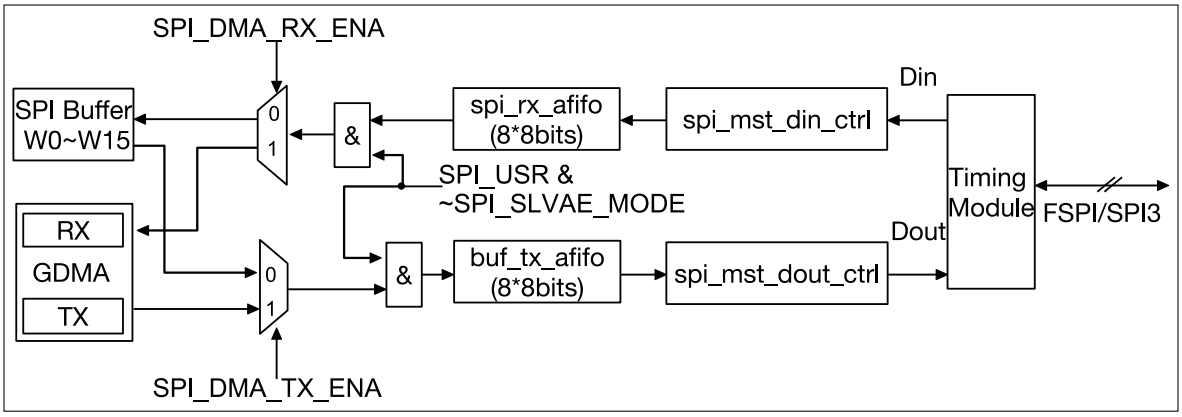


图 30.5-3. 主机模式下的数据流控制

图 30.5-3 所示为 GP-SPI 在主机模式下的数据流。其控制逻辑如下：

- RX 数据：时序模块捕获 FSPI/SPI3 总线上的数据，然后 spi_mst_din_ctrl 模块将比特数据转化为字节数据，暂存于 spi_rx_afifo 中，此后根据控制方式转存至不同的接收位置：
  - CPU 控制：转存至 SPI_W0_REG ~ SPI_W15_REG。
  - DMA 控制：转存至 GDMA RX buffer。
- TX 数据：buf_tx_afifo 模块暂存待发送数据。根据控制方式不同，待发送数据来自不同的位置：
  - CPU 控制：TX 数据来自 SPI_W0_REG ~ SPI_W15_REG。
  - DMA 控制：TX 数据来自 GDMA TX buffer。

buf_tx_afifo 中的数据会由时序模块以 1/2/4-bit 的模式发送出去。具体数据模式由 GP-SPI 状态机控制。时序模块可用于时序补偿。更多信息，见章节 30.8。

30.5.7.3 从机模式下的数据流控制

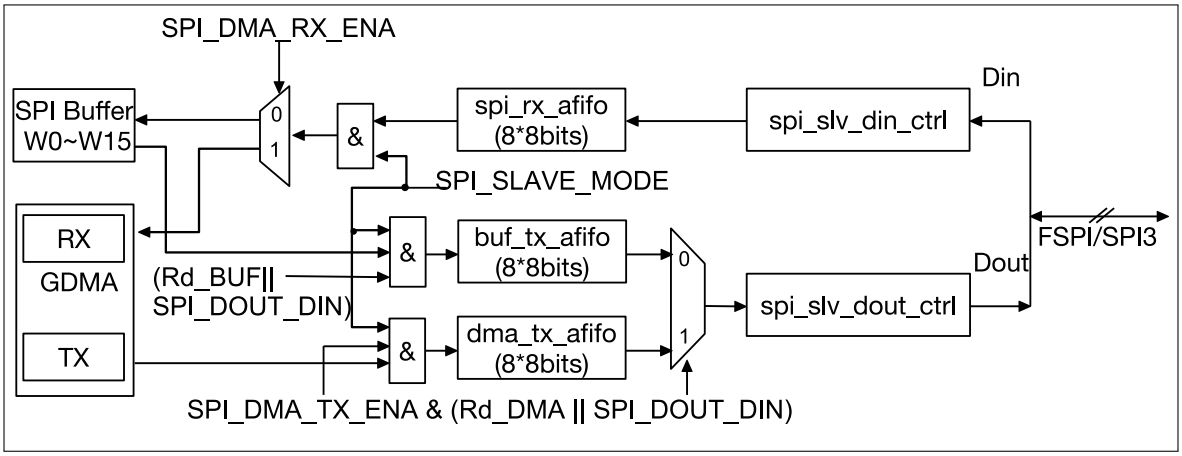


图 30.5-4. 从机模式下的数据流控制

图 30.5-4 所示为 GP-SPI 在从机模式下的数据流控制。其控制逻辑如下：

- 在 CPU/DMA 控制的全双工/半双工传输下，当外部 SPI 主机发起 SPI 传输后，FSPI/SPI3 总线上的数据将被捕获，然后由 spi_slv_din_ctrl 模块转换为字节，暂存于 spi_rx_afifo 中。
  - 在 CPU 控制的全双工传输中，暂存于 spi_rx_afifo 中的 RX 数据之后会被转存到 SPI_W0_REG ~ SPI_W15_REG。
  - 在半双工 Wr_BUF 传输中，收到地址值 (SLV_ADDR[7:0]) 后，spi_rx_afifo 中暂存的 RX 数据将转存至寄存器 SPI_W0_REG ~ SPI_W15_REG 的相应地址中。
  - 在 DMA 控制的全双工传输中，或在半双工 Wr_DMA 传输中，spi_rx_afifo 中暂存的 RX 数据将转存至配置好的 GDMA RX buffer 中。
- 在 CPU 控制的全双工/半双工传输中，待发送的数据暂存在 buf_tx_afifo 中；而在 DMA 控制的全双工/半双工传输中，待发送的数据暂存在 dma_tx_afifo 中。因此，在一次从机连续传输中，CPU 控制的 Rd_BUF 传输事务和 DMA 控制的 Rd_DMA 传输事务可同时发生。根据传输模式不同，TX 数据取自不同的地址。
  - 在 CPU 控制的全双工传输中，如果置位了 SPI_SLAVE_MODE 和 SPI_DOUTDIN，同时清零了 SPI_DMA
    _TX_ENA，则 SPI_W0_REG ~ SPI_W15_REG 中的数据将被转存至 buf_tx_afifo 中。
  - 在 CPU 控制的半双工传输中，如果置位了 SPI_SLAVE_MODE，清零了 SPI_DOUTDIN，且收到指令 Rd_BUF 和地址 SLV_ADDR[7:0]，则 SPI_W0_REG ~ SPI_W15_REG 相应地址中的数据将被转存至 buf_tx_afifo 中。
  - 在 DMA 控制的全双工传输中，如果置位了 SPI_SLAVE_MODE、SPI_DOUTDIN 和 SPI_DMA_TX_ENA，则 GDMA TX buffer 中的数据将被转存至 dma_tx_afifo 中。
  - 在 DMA 控制的半双工传输中，如果置位了 SPI_SLAVE_MODE，清零了 SPI_DOUTDIN，且收到指令 Rd_DMA，则 GDMA TX buffer 中的数据将被转存至 dma_tx_afifo 中。

  buf_tx_afifo 或 dma_tx_afifo 中的数据将由 spi_slv_dout_ctrl 模块以 1/2/4-bit 的模式发送出去。

30.5.8 GP-SPI 主机模式

说明：

- 数据以字节为单位进行传输，否则多余的位将丢失。此处多余的位表示总位长对 8 取模的结果。
- 如果需要传输非字节比特，推荐使用 CMD 状态或 ADDR 状态来实现。

清零 SPI_SLAVE_REG 中 SPI_SLAVE_MODE 位可将 GP-SPI 配置成主机模式。在这种模式下，GP-SPI 提供时钟信号（GP-SPI 模块时钟的分频时钟）和六条 CS 线 (CS0 ~ CS5)。

30.5.8.1 主机模式状态机

GP-SPI 用作主机时，状态机在数据传输中控制其各个阶段，包括配置阶段 (CONF)、准备阶段 (PREP)、命令阶段 (CMD)、地址阶段 (ADDR)、空闲阶段 (DUMMY)、发送数据阶段 (DOUT) 和接收数据阶段 (DIN)。GP-SPI 主要用于访问 1/2/4/8-bit SPI 设备，如 flash、外部 RAM 等。因此，GP-SPI 各个阶段的命名规则应与 flash 以及外部 RAM 的时序名称保持一致。每个阶段的描述如下，GP-SPI 状态机的工作流程见图 30.5-5。

1. 空闲阶段 (IDLE)：GP-SPI 未处于工作状态或处于从机模式。
2. 配置阶段 (CONF)：仅用于 DMA 控制的分段配置传输（仅对 GP-SPI2 有效）。置位 SPI_USR 和 SPI_USR_CONF 使能该阶段。如果未使能该阶段，则说明当前传输为单次传输。
3. 准备阶段 (PREP)：准备 SPI 传输事务，控制 SPI CS 建立时间。置位 SPI_USR 和 SPI_USR_SETUP 使能该阶段。
4. 命令阶段 (CMD)：发送命令序列。置位 SPI_USR 和 SPI_USR_COMMAND 使能该阶段。
5. 地址阶段 (ADDR)：发送地址序列。置位 SPI_USR 和 SPI_USR_ADDR 使能该阶段。
6. 等待阶段 (DUMMY)：发送 DUMMY 序列。置位 SPI_USR 和 SPI_USR_DUMMY 使能该阶段。
7. 传输数据阶段 (DATA)：传输数据。
   - DOUT：发送数据。置位 SPI_USR 和 SPI_USR_MOSI 使能该阶段。
   - DIN：接收数据。置位 SPI_USR 和 SPI_USR_MISO 使能该阶段。
8. 结束阶段 (DONE)：控制 SPI CS 保持时间。置位 SPI_USR 使能该阶段。

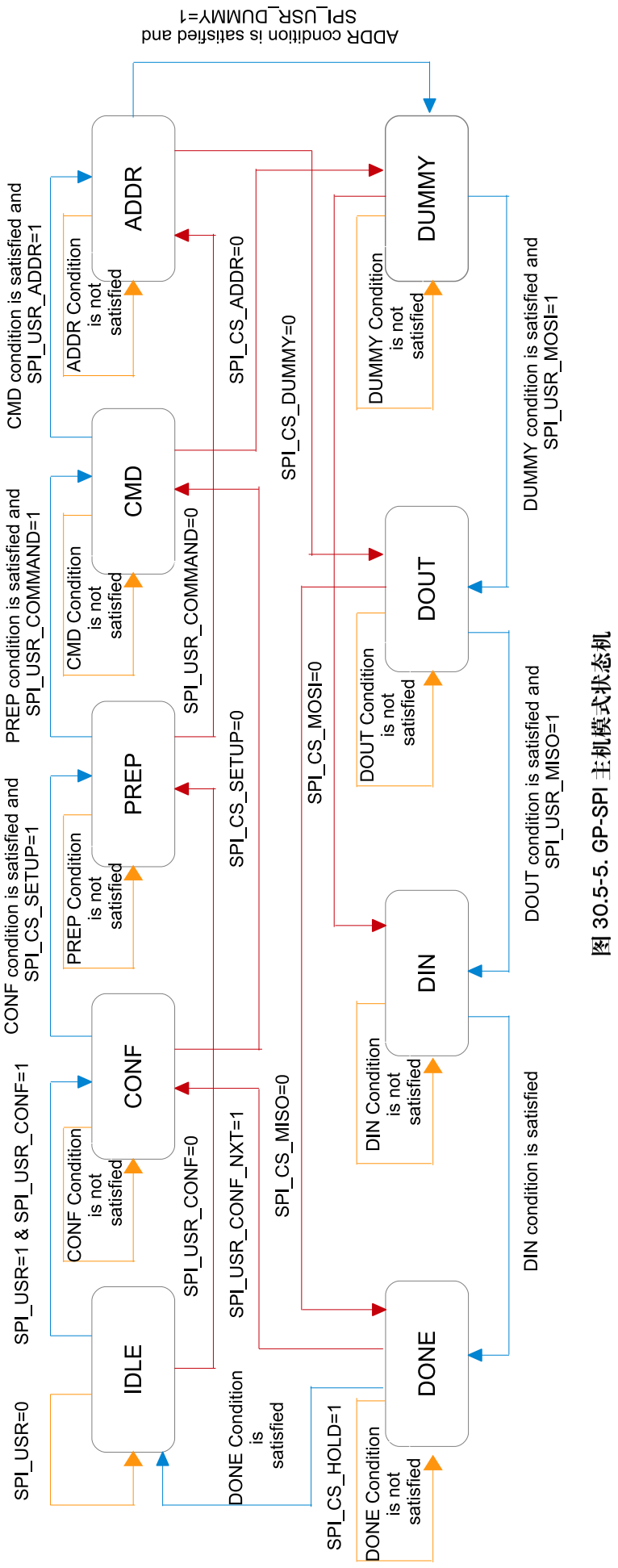


图 30.5-5. GP-SPI 主机模式状态机

图标说明：

- —: 表示相应的状态条件不满足，重复当前状态。
- —: 表示相应的寄存器已配置，状态条件已满足，将进行下一个状态。
- —: 表示相应的寄存器未配置，跳过下一个状态，或跳过后续多个状态。

上图中的各个状态条件描述如下：

- 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)

状态机中用到了一个计数器 (gpc[17:0]) 来控制每个状态的周期长度。CONF、PREP、CMD、ADDR、DUMMY、DOUT 和 DIN 各状态可单独使能或禁用，也可以单独配置其周期长度。

30.5.8.2 状态控制和位模式控制寄存器

概述

表 30.5-8 列出了与 GP-SPI 状态控制相关的寄存器配置。如需使能 GP-SPI 的 QPI 模式，请置位寄存器 SPI_USER_REG 中 SPI_QPI_MODE 位。

表 30.5-8. 1/2/4/8-bit 模式下状态控制寄存器

状态	1-bit FSPI/SPI3 总线控制寄存器	2-bit FSPI/SPI3 总线控制寄存器	4-bit FSPI/SPI3 总线控制寄存器	8-bit FSPI/SPI3 总线控制寄存器
CMD	SPI_USR_COMMAND_VALUE SPI_USR_COMMAND_BITLEN SPI_USR_COMMAND	SPI_USR_COMMAND_VALUE SPI_USR_COMMAND_BITLEN SPI_FCMD_DUAL SPI_USR_COMMAND	SPI_USR_COMMAND_VALUE SPI_USR_COMMAND_BITLEN SPI_FCMD_QUAD SPI_USR_COMMAND	SPI_USR_COMMAND_VALUE SPI_USR_COMMAND_BITLEN SPI_FCMD_OCT SPI_USR_COMMAND
ADDR	SPI_USR_ADDR_VALUE SPI_USR_ADDR_BITLEN SPI_USR_ADDR	SPI_USR_ADDR_VALUE SPI_USR_ADDR_BITLEN SPI_USR_ADDR SPI_FADDR_DUAL	SPI_USR_ADDR_VALUE SPI_USR_ADDR_BITLEN SPI_USR_ADDR SPI_FADDR_QUAD	SPI_USR_ADDR_VALUE SPI_USR_ADDR_BITLEN SPI_USR_ADDR SPI_FADDR_OCT
DUMMY	SPI_USR_DUMMY_CYCLELEN SPI_USR_DUMMY	SPI_USR_DUMMY_CYCLELEN SPI_USR_DUMMY	SPI_USR_DUMMY_CYCLELEN SPI_USR_DUMMY	SPI_USR_DUMMY_CYCLELEN SPI_USR_DUMMY
DIN	SPI_USR_MISO SPI_MS_DATA_BITLEN	SPI_USR_MISO SPI_MS_DATA_BITLEN SPI_FREAD_DUAL	SPI_USR_MISO SPI_MS_DATA_BITLEN SPI_FREAD_QUAD	SPI_USR_MISO SPI_MS_DATA_BITLEN SPI_FREAD_OCT
DOUT	SPI_USR_MOSI SPI_MS_DATA_BITLEN	SPI_USR_MOSI SPI_MS_DATA_BITLEN SPI_FWRITE_DUAL	SPI_USR_MOSI SPI_MS_DATA_BITLEN SPI_FWRITE_QUAD	SPI_USR_MOSI SPI_MS_DATA_BITLEN SPI_FWRITE_OCT

如表 30.5-8 所示，如果希望在表格第一栏所示的状态中将 FSPI/SPI3 总线设置为相应的位模式（见表头），则需要配置该行中每一单元格的寄存器。

配置

例如，当 GP-SPI 读取数据时，且希望实现：

- CMD 为 1-bit 模式
- ADDR 为 2-bit 模式
- DUMMY 为 8 个时钟周期
- DIN 为 4-bit 模式

则具体的寄存器配置如下：

1. 配置 CMD 状态相关寄存器。
   - 配置 SPI_USR_COMMAND_VALUE 为需要的命令值；
   - 配置 SPI_USR_COMMAND_BITLEN。SPI_USR_COMMAND_BITLEN 为所需要的命令位长 - 1；
   - 置位 SPI_USR_COMMAND；
   - 清除 SPI_FCMD_DUAL 和 SPI_FCMD_QUAD。
2. 配置 ADDR 状态相关寄存器。
   - 配置 SPI_USR_ADDR_VALUE 为需要的地址值；
   - 配置 SPI_USR_ADDR_BITLEN。SPI_USR_ADDR_BITLEN 为所需要的地址位长 - 1；
   - 置位 SPI_USR_ADDR 和 SPI_FADDR_DUAL；
   - 清除 SPI_FADDR_QUAD。
3. 配置 DUMMY 状态相关寄存器。
   - 在 SPI_USR_DUMMY_CYCLELEN 中配置 DUMMY 周期，其中 SPI_USR_DUMMY_CYCLELEN 的值等于 DUMMY 阶段所需要的时钟周期数 - 1；
   - 置位 SPI_USR_DUMMY。
4. 配置 DIN 状态相关寄存器。
   - 在 SPI_MS_DATA_BITLEN 中配置读数据的位长。SPI_MS_DATA_BITLEN 的值等于所需要的位长 - 1；
   - 置位 SPI_FREAD_QUAD 和 SPI_USR_MISO；
   - 清除 SPI_FREAD_DUAL；
   - 如果选择了 DAM 控制的传输模式，则需要配置 GDMA。如果选择了 CPU 控制的传输模式，则无需任何操作。
5. 清除 SPI_USR_MOSI。
6. 置位 SPI_DMA_AFIFO_RST、SPI_BUF_AFIFO_RST 和 SPI_RX_AFIFO_RST 复位 buffer。
7. 置位 SPI_USR 开始 GP-SPI 传输。

写数据时 (DOUT)，需要配置 SPI_USR_MOSI，同时清除 SPI_USR_MISO。输出数据的位长等于 SPI_MS_DATA_BITLEN 加 1。在 CPU 控制的传输模式下，需要在数据 buffer (SPI_W0_REG ~ SPI_W15_REG)

中准备数据；在 DMA 控制的数据传输下，需要在 GDMA TX buffer 中准备输出数据。字节顺序从 LSB (byte 0) 到 MSB 递增。

需特别注意 SPI_USR_COMMAND_VALUE 中的命令值以及 SPI_USR_ADDR_VALUE 中的地址值。

命令值的配置如下：

表 30.5-9. 命令值的发送顺序

COMMAND_BITLEN[1]	COMMAND_VALUE[2]	BIT_ORDER[3]	命令值的发送顺序
0 - 7	[7:0]	1	先发送 COMMAND_VALUE[COMMAND_BITLEN:0]。
		0	先发送 COMMAND_VALUE[7:7 - COMMAND_BITLEN]。
8 - 15	[15:0]	1	先发送 COMMAND_VALUE[7:0]，再发送 COMMAND_VALUE[COMMAND_BITLEN:8]。
		0	先发送 COMMAND_VALUE[7:0]，再发送 COMMAND_VALUE[15:15 - COMMAND_BITLEN]。

[1] SPI_USR_COMMAND_BITLEN：用于配置命令的位长。
[2] SPI_USR_COMMAND_VALUE：命令值写入的字段，见上表。
[3] SPI_WR_BIT_ORDER：0：先发送 LSB；1：先发送 MSB。

地址值配置如下：

表 30.5-10. 地址值的发送顺序

ADDR_BITLEN[1]	ADDR_VALUE[2]	BIT_ORDER[3]	地址值的发送顺序
0 - 7	[31:24]	1	先发送 COMMAND_VALUE[ADDR_BITLEN + 24:24]。
		0	先发送 ADDR_VALUE[31:31 - ADDR_BITLEN]。
8 - 15	[31:16]	1	先发送 ADDR_VALUE[31:24]，再发送 ADDR_VALUE[ADDR_BITLEN + 8:16]。
		0	先发送 ADDR_VALUE[31:24]，再发送 ADDR_VALUE[23:31 - ADDR_BITLEN]。
16 - 23	[31:8]	1	先发送 ADDR_VALUE[31:16]，再发送 ADDR_VALUE[ADDR_BITLEN - 8:8]。
		0	先发送 ADDR_VALUE[31:16]，再发送 ADDR_VALUE[15:31 - ADDR_BITLEN]。
24 - 31	[31:0]	1	先发送 ADDR_VALUE[31:8]，再发送 ADDR_VALUE[ADDR_BITLEN - 24:0]。
		0	先发送 ADDR_VALUE[31:8]，再发送 ADDR_VALUE[7:31 - ADDR_BITLEN]。

[1] SPI_USR_ADDR_BITLEN：用于配置地址值的位长。
[2] SPI_USR_ADDR_VALUE：地址值写入的字段，见上表。
[3] SPI_WR_BIT_ORDER：0：先发送 LSB；1：先发送 MSB。

30.5.8.3 主机全双工通信（仅支持 1-bit 模式）

概述

GP-SPI 支持 SPI 全双工通信。在这种模式下，SPI 主机提供 CLK 和 CS 信号，然后与从机使用 1-bit 模式同时交换数据：MOSI（FSPID/SPI3_D，发送），MISO（FSPIQ/SPI3_Q，接收）。用户可通过置位寄存器 SPI_USER_REG 中 SPI_DOUTDIN 位使能全双工通信。GP-SPI2 与从机使用全双工通信时的连接方式见图 30.5-6。

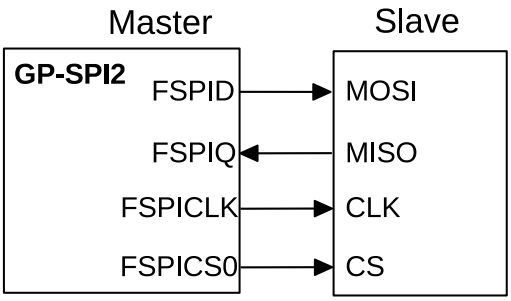


图 30.5-6. GP-SPI2 主机使用全双工模式与 SPI 从机通信框图

在全双工通信中，CMD、ADDR、DUMMY、DOUT 和 DIN 各个状态的具体行为可配置。通常，全双工模式跳过 CMD、ADDR 和 DUMMY 状态。传输数据的位长可在 SPI_MS_DATA_BITLEN 中配置。通信中使用的实际位长等于 (SPI_MS_DATA_BITLEN + 1)。

配置（以 GP-SPI2 为例）

按照以下操作步骤，开始数据传输：

- 经 IO MUX 或 GPIO 交换矩阵配置 GP-SPI2 与外部 SPI 设备之间的 IO 通道；
- 配置 APB 时钟（即 APB_CLK，见章节 7 复位和时钟），并为 GP-SPI 配置模块时钟 (clk_spi_mst)；
- 置位 SPI_DOUTDIN 同时清除 SPI_SLAVE_MODE，使能主机模式下的全双工通信方式；
- 配置表 30.5-8 中所列的 GP-SPI 寄存器；
- 配置 SPI CS 建立时间和保持时间，见章节 30.6；
- 设置 FSPICLK 的极性，见章节 30.7；
- 根据选定的传输模式准备数据：
  - 如果选择的传输模式为 CPU 控制的 MOSI 传输，则需要在 SPI_W0_REG ~ SPI_W15_REG 中准备数据。
  - 如果选择了 DAM 控制的传输模式，则需要：
    * 配置 SPI_DMA_TX_ENA/SPI_DMA_RX_ENA；
    * 配置 GDMA TX/RX 链表；
    * 启动 GDMA TX/RX 引擎，更多描述见章节 30.5.6 和章节 30.5.7。
- 配置中断，然后等待 SPI 从机做好传输准备；
- 置位 SPI_DMA_AFIFO_RST、SPI_BUF_AFIFO_RST 和 SPI_RX_AFIFO_RST 复位 buffer；
- 置位寄存器 SPI_CMD_REG 中 SPI_USR 位，开始数据传输，然后等待之前配置的中断。

30.5.8.4 主机半双工通信（支持 1/2/4/8-bit 模式）

概述

在半双工模式下，GP-SPI 发送 CLK 和 CS 信号。在同一时刻，SPI 主机或从机只能有一个可以发送数据，另一个接收数据。用户可通过清除寄存器 SPI_USER_REG 中 SPI_DOUTDIN 位使能半双工通信。SPI 半双工通信的通用格式为 CMD + [ADDR +] [DUMMY +] [DOUT or DIN]。其中，ADDR、DUMMY、DOUT 和 DIN 状态非必选，可单独禁用或启用。

如章节 30.5.8.2 所述，CMD、ADDR、DUMMY、DOUT 和 DIN 各个状态的周期、具体值和并行总线位模式等可独立配置。更多寄存器配置信息，见表 30.5-8。

半双工 GP-SPI 的详细属性如下：

1. CMD：0 ~ 16 位，主机发送，从机接收 (MOSI)。
2. ADDR：0 ~ 32 位，主机发送，从机接收。
3. DUMMY：0 ~ 256 个 FSPICLK/SPI3_CLK 周期，主机发送，从机接收。
4. DOUT：在 CPU 控制的模式下，可传输 0 ~ 512 位（64 字节）数据；在 DMA 控制的单次传输模式下，可传输 0 ~ 256 Kbit (32 KB)；在 DMA 控制的分段配置传输模式下，可传输的数据长度无限制。主机发送，从机接收。
5. DIN：在 CPU 控制的模式下，可传输 0 ~ 512 位（64 字节）数据；在 DMA 控制的单次传输模式下，可传输 0 ~ 256 Kbit (32 KB)；在 DMA 控制的分段配置传输模式下，可传输的数据长度无限制。从机发送，主机接收 (MISO)。

配置（以 GP-SPI2 为例）

则具体的寄存器配置如下：

1. 经 IO MUX 或 GPIO 交换矩阵配置 GP-SPI2 与外部 SPI 设备之间的 IO 通道；
2. 配置 APB 时钟 (APB_CLK)，并为 GP-SPI 配置模块时钟 (clk_spi_mst)；
3. 清除 SPI_DOUTDIN 和 SPI_SLAVE_MODE 位，使能主机模式下的半双工通信方式；
4. 配置表 30.5-8 中所列的 GP-SPI 寄存器；
5. 配置 SPI CS 建立时间和保持时间，见章节 30.6；
6. 设置 FSPICLK 的极性，见章节 30.7；
7. 根据选定的传输模式准备数据：
   - 如果选择的传输模式为 CPU 控制的 MOSI 传输，则需要在 SPI_W0_REG ~ SPI_W15_REG 中准备数据。
   - 如果选择了 DAM 控制的传输模式，则需要：
     - 配置 SPI_DMA_TX_ENA/SPI_DMA_RX_ENA；
     - 配置 GDMA TX/RX 链表；
     - 启动 GDMA TX/RX 引擎，更多描述见章节 30.5.6 和章节 30.5.7。
8. 配置中断，然后等待 SPI 从机做好传输准备；
9. 置位 SPI_DMA_AFIFO_RST、SPI_BUF_AFIFO_RST 和 SPI_RX_AFIFO_RST 复位 buffer；
10. 置位寄存器 SPI_CMD_REG 中 SPI_USR 位，开始数据传输，然后等待之前配置的中断。

应用示例（以 GP-SPI2 为例）

以下示例展示了 GP-SPI2 如何在主机半双工模式下访问 flash 和外部 RAM。

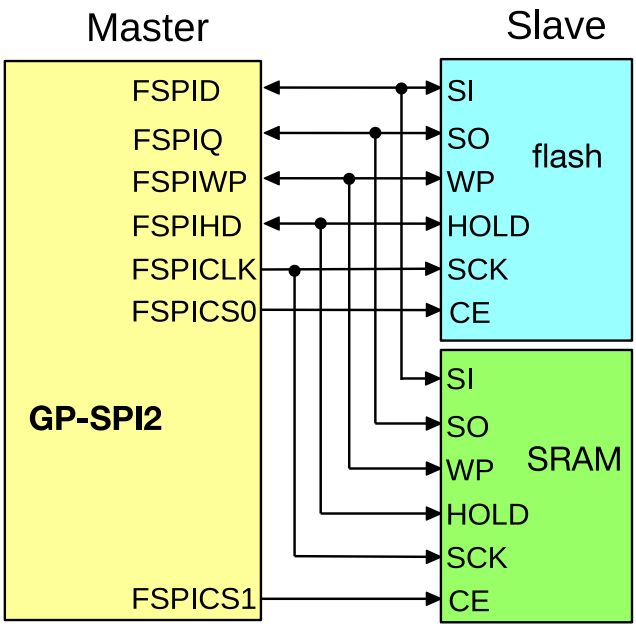


图 30.5-7. 4-bit 模式下 GP-SPI2 与 flash 以及外部 RAM 的连接方式

图 30.5-8 所示为 GP-SPI2 按照标准 flash 规范进行 Quad I/O Read 操作。其它 GP-SPI 命令序列可以根据 SPI 从机的要求实现。

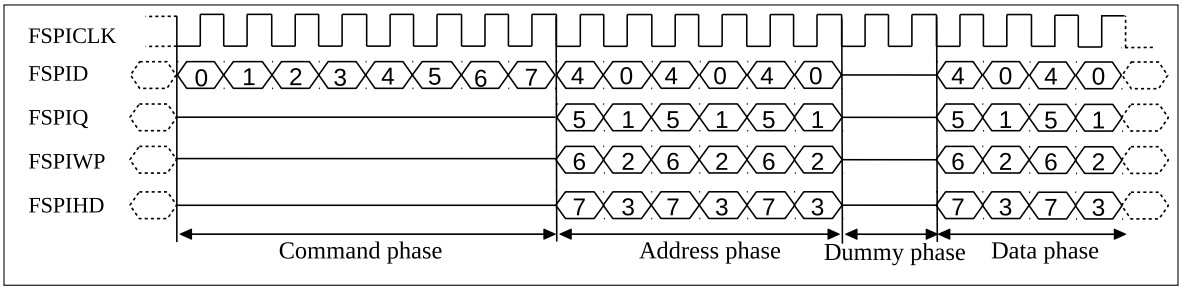


图 30.5-8. GP-SPI2 发送到 flash 的 SPI Quad I/O Read 命令序列

30.5.8.5 DMA 控制的分段配置传输

说明：

- GP-SPI3 不支持 DMA 控制的分段配置传输。
- 由于跳过 CONF 阶段即可实现单次传输，因此不再另起章节单独介绍如何在主机模式下配置单次传输。

概述

GP-SPI2 用作主机时，可采用 DMA 控制的分段配置传输模式。

DMA 控制的主机传输可以是：

- 一次单次传输，仅包含一次传输事务。
- 分段配置传输，包含多个传输事务（即多个分段）。

如果选择了分段配置传输模式，则在每个分段中，寄存器均可单独配置。在分段配置传输模式下，仅需 CPU 触发一次，即可完成多次传输事务。具体工作流程见图 30.5-9。

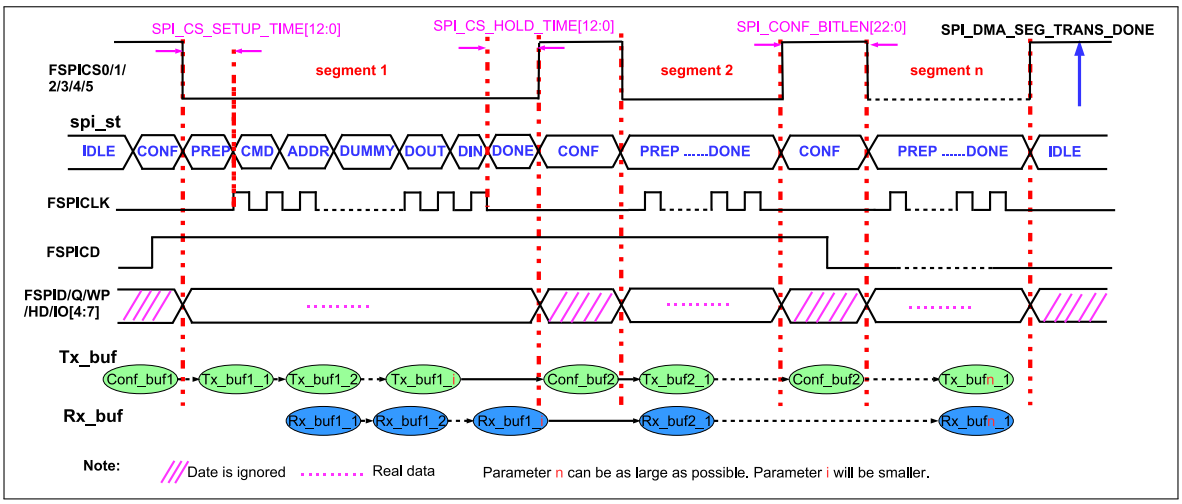


图 30.5-9. 主机模式下 DAM 控制的分段配置传输

如图 30.5-9 所示，在分段配置传输模式中的某个单次传输事务 (segment *n*) 开始前，GP-SPI2 可在 CONF 阶段将寄存器重新按照 Conf_buf*n* 定义的内容进行配置。

建议为每个传输事务的 CONF 阶段提供单独的 GDMA CONF 链表和 CONF buffer（即图 30.5-9 中的 Conf_buf*i*）。GDMA TX 链表将所有的 CONF buffer 和 TX data buffer（即图 30.5-9 中的 Tx_buf*i*）链接起来，因此可以独立控制每个传输事务中的 FSPI 总线行为。

例如，在一次完整的分段配置传输中，传输事务 *i*、传输事务 *j* 和传输事务 *k* 可分别配置为全双工、半双工 MISO 和半双工 MOSI 模式。*i*、*j*、*k* 均为整数变量，代表传输事务的编号。

同时，每个传输事务中，GP-SPI2 所使用到的各个阶段、各个阶段的相关值和 FSPI 总线周期长、以及 GDMA 行为等，均可独立配置。当整个 DMA 控制的传输（包括多个传输事务）完成后，即触发 GP-SPI2 中断 SPI_DMA_SEG_TRANS_DONE_INT。

配置

1. 经 IO MUX 或 GPIO 交换矩阵配置 GP-SPI2 与外部 SPI 设备之间的 IO 通道；
2. 配置 APB 时钟 (APB_CLK)，并为 GP-SPI2 配置模块时钟 (clk_spi_mst)；
3. 清除 SPI_DOUTDIN 和 SPI_SLAVE_MODE 位，使能主机模式下的半双工通信方式；
4. 配置表 30.5-8 中所列的 GP-SPI 寄存器；
5. 配置 SPI CS 建立时间和保持时间，见章节 30.6；
6. 设置 FSPICLK 的极性，见章节 30.7；
7. 为每个传输事务准备 GDMA CONF buffer 描述符和 TX data 描述符（可选）。把 CONF buffer 描述符和几次传输事务需要的 TX buffer 链接成一个链表；
8. 同样，为每个传输事务准备 RX buffer 描述符，并链接成一个链表；
9. 在该 DMA 控制的分段配置传输开始之前，为每个传输事务配置所需的 CONF buffer、TX buffer 和 RX buffer；
10. 配置 GDMA_OUTLINK_ADDR_CH*n* 指向 CONF 和 TX buffer 描述符链表的首地址，之后置位 GDMA_OUTLINK

_START_CH*n*，启动 TX GDMA；

11. 清除 SPI_DMA_CONF_REG 中 SPI_RX_EOF_EN 位。配置 GDMA_INLINK_ADDR_CH*n* 指向 RX buffer 描述符链表的首地址，之后置位 GDMA_INLINK_START_CH*n* 启动 RX GDMA；
12. 置位 SPI_USR_CONF 使能 CONF 阶段；
13. 置位 SPI_DMA_SEG_TRANS_DONE_INT_ENA 使能 SPI_DMA_SEG_TRANS_DONE_INT 中断。如需配置其它中断，请参考章节 30.10；
14. 等待所有从机做好传输准备；
15. 置位 SPI_DMA_AFIFO_RST、SPI_BUF_AFIFO_RST 和 SPI_RX_AFIFO_RST 复位 buffer；
16. 置位 SPI_USR 开始本次 DMA 控制的分段配置传输；
17. 等待 SPI_DMA_SEG_TRANS_DONE_INT 中断，即 DMA 分段配置传输结束，数据已存储至相应内存。

配置 CONF Buffer 和 Magic 值

在 GP-SPI2 分段配置传输中，仅有较上次传输事务有变动的寄存器会在 CONF 阶段被重新配置。为节省时间和芯片资源，其它寄存器配置则保持不变。

GDMA CONF buffer*i* 中第一个字，即 SPI_BIT_MAP
_WORD，记录传输事务 *i* 中，寄存器是否有改动。SPI_BIT_MAP_WORD 和待更新的 GP-SPI2 寄存器的对应关系见表 30.5-11，即位图 (BM) 表。如果位图表中某一位为 1，则在此次传输事务中，该位对应寄存器的值将被更新。位表中为 0 的寄存器则保持不变。

表 30.5-11. CONF 阶段 BM 位图

BM 位	寄存器	BM 位	寄存器
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

所有待修改的寄存器新值应紧跟在 SPI_BIT_MAP_WORD 之后，在 CONF buffer 中用连续的字表示。

为确保每个 CONF buffer 中内容正确，SPI_BIT_MAP_WORD[31:28] 位将用作 Magic 值，与寄存器 SPI_SLAVE_REG 中 SPI_DMA_SEG_MAGIC_VALUE 的值进行比较。SPI_DMA_SEG_MAGIC_VALUE 的值应在此 DMA 控制的分段配置传输开始之前配置，且在任何传输事务过程中均不可更改。

- 经比较，如果 SPI_BIT_MAP_WORD[31:28] == SPI_DMA_SEG_MAGIC_VALUE，则分段配置传输继续正常进行，整个传输过程结束则触发 SPI_DMA_SEG_TRANS_DONE_INT 中断。
- 如果 SPI_BIT_MAP_WORD[31:28] != SPI_DMA_SEG_MAGIC_VALUE，则 GP-SPI2 状态，即 spi_st 将返回至 IDLE 状态，分段配置传输立即结束。同时触发 SPI_DMA_SEG_TRANS_DONE_INT 中断，SPI_SEG_MAGIC_
  ERR_INT_RAW 位也将置 1。

CONF Buffer 配置示例

在一次分段配置传中，传输事务 *i* 有 SPI_ADDR_REG、SPI_CTRL_REG、SPI_CLOCK_REG、SPI_USER_REG 和 SPI_USER1_REG 五个寄存器需要更新，则其 CONF buffer*i* 具体的配置示例见表 30.5-12 和表 30.5-13。

表 30.5-12. 传输事务 *i* 中 CONF buffer*i* 配置示例

CONF buffer*i*	说明
SPI_BIT_MAP_WORD	Buffer 中的第一个字如果 SPI_DMA_SEG_MAGIC_VALUE 设置为 0xA，则本示例中该字的值为 0xA000001F。由表 30.5-13 可知，被置 1 的位有第 0、1、2、3 和 4 位，表示下列寄存器将被更新
SPI_ADDR_REG	CONF buffer*i* 的第二个字，存储 SPI_ADDR_REG 寄存器的更新值
SPI_CTRL_REG	CONF buffer*i* 的第三个字，存储 SPI_CTRL_REG 寄存器的更新值
SPI_CLOCK_REG	CONF buffer*i* 的第四个字，存储 SPI_CLOCK_REG 寄存器的更新值
SPI_USER_REG	CONF buffer*i* 的第五个字，存储 SPI_USER_REG 寄存器的更新值
SPI_USER1_REG	CONF buffer*i* 的第六个字，存储 SPI_USER1_REG 寄存器的更新值

表 30.5-13. BM 位图与待更新的寄存器

BM 位	值	寄存器	BM 位	值	寄存器
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

说明

使用 DMA 分段配置传输功能时，应注意以下寄存器相关位：

- SPI_USR_CONF：在置位 SPI_USR 之前，需先置位 SPI_USR_CONF，以使能本次传输。
- SPI_USR_CONF_NXT：如果传输事务 *i* 不是本次 DMA 控制的分段配置传输中的最后一次传输事务，则需要置位 SPI_USR_CONF_NXT。
- SPI_CONF_BITLEN：此外，在每个单独的传输事务中，GP-SPI2 的 CS 建立时间和保持时间可独立编程，更多配置信息见章节 30.6。在每次传输事务中，CS 保持高电平的时长约为：

$$(SPI\_CONF\_BITLEN + 5) \times T_{APB\_CLK}$$

f_{APB_CLK} 为 80 MHz 时，CONF 阶段的 CS 高电平时长可配置为 62.5 ns ~ 3.2768 ms。如果 SPI_CONF_BITLEN 大于 0x3FFFA，(SPI_CONF_BITLEN + 5) 将溢出 (0x40000 - SPI_CONF_BITLEN - 5)。

30.5.9 GP-SPI 从机模式

GP-SPI 可用作从机与另一 SPI 主机进行通信。用作从机时，GP-SPI 支持特定格式的 1-bit SPI、2-bit Dual SPI、4-bit Quad SPI 和 QPI 模式。用户可置位寄存器 SPI_SLAVE_REG 中 SPI_SLAVE_MODE 位使能 GP-SPI 从机模式。

在传输过程中，CS 信号应保持低电平，CS 信号的下降沿和上升沿代表一次单次传输或从机连续传输的开始和结束。

说明：
数据以字节为单位进行传输，否则多余的位将丢失。此处多余的位表示总位长对 8 取模的结果。

30.5.9.1 可配置的通信格式

GP-SPI 从机模式支持全双工通信和半双工通信。用户可配置寄存器 SPI_USER_REG 中 SPI_DOUTDIN 位选择需要的通信方式。

全双工模式下，传输一开始，则数据同时输入和输出。在此模式下，所有数据位均被视为输入/输出数据，即不需要命令、地址或 DUMMY 阶段。传输结束即触发 SPI_TRANS_DONE_INT 中断。

在半双工通信模式下，通信格式为 CMD+ADDR+DUMMY+DATA (DIN or DOUT)。

- “DIN” 表示 SPI 主机从 GP-SPI 中读取数据；
- “DOUT” 表示 SPI 主机向 GP-SPI 中写入数据。

每个阶段的详细特性如下：

1. CMD：
   - 表明 SPI 从机用于何种功能；
   - 一个字节，主机输出，从机输入；
   - 仅支持表 30.5-14 和表 30.5-15 所列的命令值；
   - 以 1-bit SPI 模式或 4-bit QPI 模式发送。
2. ADDR：
   - 在 CPU 控制的传输中，可以为 Wr_BUF 和 Rd_BUF 命令提供地址，或在其它命令中用作占位符，具体由应用定义；
   - 一个字节，主机输出，从机输入；
   - 可根据命令，以 1-bit，2-bit 或 4-bit 模式发送。
3. DUMMY：
   - DUMMY 的值无实际意义；SPI 从机在这个阶段准备数据；
   - FSPI 总线的位模式在这里也没有实际意义：
   - 持续八个 SPI_CLK 时钟周期。
4. DIN 或 DOUT：
   - 在 CPU 控制的模式下，可传输 0 ~ 64 字节数据；在 DMA 控制的模式下，传输数据长度无限制。
   - 可根据具体的 CMD 值，以 1-bit、2-bit 或 4-bit 模式发送。

说明：
半双工通信模式下，ADDR 和 DUMMY 阶段不可跳过。

半双工传输结束后，传输的 CMD 和 ADDR 的值分别锁存至 SPI_SLV_LAST_COMMAND 和 SPI_SLV_LAST_ADDR。如果 GP-SPI 从机模式不支持传输的 CMD 值，SPI_SLV_CMD_ERR_INT_RAW 将被置位。SPI_SLV_CMD_ERR_INT

_RAW 仅可由软件清零。

30.5.9.2 半双工通信支持的 CMD 值

在半双工传输中，CMD 定义的值将决定传输类型。不支持的 CMD 值及其相关数据传输均被忽略，且 SPI_SLV_CMD _ERR_INT_RAW 将被置 1。传输格式为：CMD（8 位）+ ADDR（8 位）+ DUMMY（8 个 SPI_CLK 周期）+ DATA（单位：字节），CMD[3:0] 的详细说明如下：

- 0x1 (Wr_BUF)：CPU 控制的写操作模式。主机发送数据，GP-SPI 接收数据。数据将存储至相应地址的寄存器 SPI_W0_REG ~ SPI_W15_REG。
- 0x2 (Rd_BUF)：CPU 控制的读操作模式。主机接收 GP-SPI 发送的数据。数据来自相应地址的寄存器 SPI_W0_REG ~ SPI_W15_REG。
- 0x3 (Wr_DMA)：DMA 控制的写操作模式。主机发送数据，GP-SPI 接收数据。数据将存储至 GP-SPI 的 GDMA RX buffer 中。
- 0x4 (Rd_DMA)：DMA 控制的读操作模式。主机接收 GP-SPI 发送的数据。数据来自 GP-SPI 接口的 GDMA TX buffer。
- 0x7 (CMD7)：用于生成 SPI_SLV_CMD7_INT 中断。在从机连续传输模式下，使用 GDMA RX 链表时，也可用于生成 GDMA_IN_SUC_EOF_CH*n*_INT 中断。但不会结束 GP-SPI 的从机连续传输。
- 0x8 (CMD8)：仅用于生成 SPI_SLV_CMD8_INT 中断，但不会结束 GP-SPI 的从机连续传输。
- 0x9 (CMD9)：仅用于生成 SPI_SLV_CMD9_INT 中断，但不会结束 GP-SPI 的从机连续传输。
- 0xA (CMDA)：仅用于生成 SPI_SLV_CMDA_INT 中断，但不会结束 GP-SPI 的从机连续传输。

CMD7、CMD8、CMD9 和 CMDA 的具体用途可由用户自定义。这些命令可用作握手信号、某些特定功能的密码、或某些用户自定义操作的触发器等。

CMD、ADDR 和 DATA 阶段均支持 1/2/4-bit 模式，具体由 CMD[7:4] 决定。DUMMY 仅支持 1-bit 模式，且持续八个 SPI_CLK 时钟周期。CMD[7:4] 的具体定义如下：

- 0x0：CMD、ADDR 和 DATA 阶段均为 1-bit 模式。
- 0x1：CMD 和 ADDR 均为 1-bit 模式。DATA 为 2-bit 模式。
- 0x2：CMD 和 ADDR 均为 1-bit 模式。DATA 为 4-bit 模式。
- 0x5：CMD 为 1-bit 模式。ADDR 和 DATA 均为 2-bit 模式。
- 0xA：CMD 为 1-bit 模式，ADDR 和 DATA 均为 4-bit 模式。或 QPI 模式。

此外，CMD[7:0] 的值为 0x05、0xA5、0x06 和 0xDD 时，将跳过 DUMMY 和 DATA 阶段。CMD[7:0] 的具体定义如下：

- 0x05 (End_SEG_TRANS)：主机发送 0x05 命令，结束 SPI 模式下从机连续传输。
- 0xA5 (End_SEG_TRANS)：主机发送 0xA5 命令，结束 QPI 模式下从机连续传输。
- 0x06 (En_QPI)：GP-SPI 接收到 0x06 命令后，进入 QPI 模式。此时，寄存器 SPI_USER_REG 中 SPI_QPI_MODE 置位。
- 0xDD (Ex_QPI)：GP-SPI 接收到 0xDD 命令后，退出 QPI 模式。此时，SPI_QPI_MODE 位清零。

GP-SPI 支持的所有 CMD 值见表 30.5-14 和表 30.5-15。注意，DUMMY 仅支持 1-bit 模式，且持续八个 SPI_CLK 时钟周期。

表 30.5-14. GP-SPI 从机 SPI 模式支持的 CMD 值

传输类型	CMD[7:0]	CMD 阶段	ADDR 阶段	DATA 阶段
Wr_BUF	0x01	1-bit 模式	1-bit 模式	1-bit 模式
	0x11	1-bit 模式	1-bit 模式	2-bit 模式
	0x21	1-bit 模式	1-bit 模式	4-bit 模式
	0x51	1-bit 模式	2-bit 模式	2-bit 模式
	0xA1	1-bit 模式	4-bit 模式	4-bit 模式
Rd_BUF	0x02	1-bit 模式	1-bit 模式	1-bit 模式
	0x12	1-bit 模式	1-bit 模式	2-bit 模式
	0x22	1-bit 模式	1-bit 模式	4-bit 模式
	0x52	1-bit 模式	2-bit 模式	2-bit 模式
	0xA2	1-bit 模式	4-bit 模式	4-bit 模式
Wr_DMA	0x03	1-bit 模式	1-bit 模式	1-bit 模式
	0x13	1-bit 模式	1-bit 模式	2-bit 模式
	0x23	1-bit 模式	1-bit 模式	4-bit 模式
	0x53	1-bit 模式	2-bit 模式	2-bit 模式
	0xA3	1-bit 模式	4-bit 模式	4-bit 模式
Rd_DMA	0x04	1-bit 模式	1-bit 模式	1-bit 模式
	0x14	1-bit 模式	1-bit 模式	2-bit 模式
	0x24	1-bit 模式	1-bit 模式	4-bit 模式
	0x54	1-bit 模式	2-bit 模式	2-bit 模式
	0xA4	1-bit 模式	4-bit 模式	4-bit 模式
CMD7	0x07	1-bit 模式	1-bit 模式	-
	0x17	1-bit 模式	1-bit 模式	-
	0x27	1-bit 模式	1-bit 模式	-
	0x57	1-bit 模式	2-bit 模式	-
	0xA7	1-bit 模式	4-bit 模式	-
CMD8	0x08	1-bit 模式	1-bit 模式	-
	0x18	1-bit 模式	1-bit 模式	-
	0x28	1-bit 模式	1-bit 模式	-
	0x58	1-bit 模式	2-bit 模式	-
	0xA8	1-bit 模式	4-bit 模式	-
CMD9	0x09	1-bit 模式	1-bit 模式	-
	0x19	1-bit 模式	1-bit 模式	-
	0x29	1-bit 模式	1-bit 模式	-
	0x59	1-bit 模式	2-bit 模式	-
	0xA9	1-bit 模式	4-bit 模式	-
CMDA	0x0A	1-bit 模式	1-bit 模式	-
	0x1A	1-bit 模式	1-bit 模式	-
	0x2A	1-bit 模式	1-bit 模式	-
	0x5A	1-bit 模式	2-bit 模式	-
	0xAA	1-bit 模式	4-bit 模式	-

表 30.5-14. GP-SPI 从机 SPI 模式支持的 CMD 值

传输类型	CMD[7:0]	CMD 阶段	ADDR 阶段	DATA 阶段
End_SEG_TRANS	0x05	1-bit 模式	-	-
En_QPI	0x06	1-bit 模式	-	-

表 30.5-15. GP-SPI 从机 QPI 支持的 CMD 值

传输类型	CMD[7:0]	CMD 阶段	ADDR 阶段	DATA 阶段
Wr_BUF	0xA1	4-bit 模式	4-bit 模式	4-bit 模式
Rd_BUF	0xA2	4-bit 模式	4-bit 模式	4-bit 模式
Wr_DMA	0xA3	4-bit 模式	4-bit 模式	4-bit 模式
Rd_DMA	0xA4	4-bit 模式	4-bit 模式	4-bit 模式
CMD7	0xA7	4-bit 模式	4-bit 模式	-
CMD8	0xA8	4-bit 模式	4-bit 模式	-
CMD9	0xA9	4-bit 模式	4-bit 模式	-
CMDA	0xAA	4-bit 模式	4-bit 模式	-
End_SEG_TRANS	0xA5	4-bit 模式	4-bit 模式	-
Ex_QPI	0xDD	4-bit 模式	4-bit 模式	-

GP-SPI 收到主机发送的 0x06 CMD (En_QPI) 命令后，将进入 QPI 模式。GP-SPI 在 QPI 模式下支持的传输类型，其后续所有阶段均为 4-bit 模式。如果收到 0xDD CMD (Ex_QPI)，则 GP-SPI 从机将返回到 SPI 模式。

未在表 30.5-14 和表 30.5-15 中列出的传输类型将被忽略掉。如果传输的数据不以字节为单位，GP-SPI 会发送或接收不足 8 位的比特，但不确保数据的正确性。但如果 CS 低电平持续时长大于 2 个 APB_CLK 时钟周期，则将触发 SPI_TRANS_DONE_INT 中断。有关传输结束时触发的中断信息，请参考章节 30.10。

30.5.9.3 从机单次传输和从机连读传输

GP-SPI 用作从机时，支持由 DMA 和 CPU 控制的全双工和半双工通信。DMA 控制的从机传输，可以是一次单次传输，也可以是从机连续传输（包含多次传输事务）。CPU 控制的传输只能是单次传输，因为每次传输均需由 CPU 触发。

一次从机连续传输包含多个传输事务，每个传输事务可以是表 30.5-14 和 30.5-15 列出的任一传输类型。即在一次完整的连续传输过程中，可以包含 CPU 控制的数据传输，也可以包含 DMA 控制的数据传输。

在一次完整的连续传输过程中，推荐操作如下：

- CPU 控制的数据传输可用于握手通信以及少量数据传输。
- DMA 控制的数据传输可用于大量数据传输。

30.5.9.4 配置从机单次传输模式

在从机模式下，GP-SPI 支持 CPU 控制的和 DMA 控制的全/半双工单次传输。

具体的寄存器配置如下（以 GP-SPI2 为例）：

1. 经 IO MUX 或 GPIO 交换矩阵配置 GP-SPI2 与外部 SPI 设备之间的 IO 通道；
2. 配置 APB 时钟 (APB_CLK)；

3. 置位 SPI_SLAVE_MODE 使能从机模式；
4. 配置 SPI_DOUTDIN：
   - 1：使能全双工通信。
   - 0：使能半双工通信。
5. 准备数据：
   - 如果选择的传输模式为 CPU 控制的传输，且 GP-SPI 发送数据，则在寄存器 SPI_W0_REG ~ SPI_W15_REG 中准备数据。
   - 如果选择的传输模式为 DMA 控制的传输模式，则需要：
     - 配置 SPI_DMA_TX_ENA/SPI_DMA_RX_ENA 和 SPI_RX_EOF_EN；
     - 配置 GDMA TX/RX 链表；
     - 启动 GDMA TX/RX 引擎，更多描述见章节 30.5.6 和章节 30.5.7。
6. 置位 SPI_DMA_AFIFO_RST、SPI_BUF_AFIFO_RST 和 SPI_RX_AFIFO_RST 复位 buffer；
7. 清零寄存器 SPI_DMA_CONF_REG 中 SPI_DMA_SLV_SEG_TRANS_EN 使能从机单次传输；
8. 置位寄存器 SPI_DMA_INT_ENA_REG 中 SPI_TRANS_DONE_INT_ENA，使能中断，并等待 SPI_TRANS_DONE_INT。在 DMA 控制模式下，使用 DMA RX buffer 时，推荐等待 GDMA_IN_SUC_EOF_CH*n*_INT 中断，即数据已存储至相应内存。其它中断见章节 30.10。

30.5.9.5 配置半双工模式下从机连续传输

此模式必须使用 GDMA。具体的寄存器配置如下（以 GP-SPI2 为例）：

1. 经 IO MUX 或 GPIO 交换矩阵配置 GP-SPI2 与外部 SPI 设备之间的 IO 通道；
2. 配置 APB 时钟 (APB_CLK)；
3. 置位 SPI_SLAVE_MODE 使能从机模式；
4. 清除 SPI_DOUTDIN 使能半双工通信方式；
5. 根据需求，确定是否需要在寄存器 SPI_W0_REG ~ SPI_W15_REG 中准备数据；
6. 置位 SPI_DMA_AFIFO_RST、SPI_BUF_AFIFO_RST 和 SPI_RX_AFIFO_RST 复位 buffer；
7. 置位 SPI_DMA_RX_ENA 和 SPI_DMA_TX_ENA。清零 SPI_RX_EOF_EN。配置 GDMA TX/RX 链表，并启动 GDMA TX/RX 引擎，更多描述见章节 30.5.6 和章节 30.5.7；
8. 置位寄存器 SPI_DMA_CONF_REG 中 SPI_DMA_SLV_SEG_TRANS_EN，使能从机连续传输；
9. 置位寄存器 SPI_DMA_INT_ENA_REG 中 SPI_DMA_SEG_TRANS_DONE_INT_ENA，使能中断，并等待 SPI_DMA_SEG_TRANS_DONE_INT 中断。中断发生，即表明从机连续传输已结束，且数据已放入相应的内存中。其它中断见章节 30.10。

GP-SPI 收到 End_SEG_TRANS 命令（SPI 模式下为 0x05，QPI 模式下为 0xA5），从机连续传输结束，并触发 SPI_DMA_SEG_TRANS_DONE_INT 中断。

30.5.9.6 配置全双工模式下从机连续传输

此模式必须使用 GDMA。数据从 GDMA buffer 中输入输出。传输结束，触发 GDMA_IN_SUC_EOF_CH*n*_INT 中断。具体的配置程序如下（以 GP-SPI2 为例）：

1. 经 IO MUX 或 GPIO 交换矩阵配置 GP-SPI 与外部 SPI 设备之间的 IO 通道；
2. 配置 APB 时钟 (APB_CLK)；
3. 置位 SPI_SLAVE_MODE 和 SPI_DOUTDIN，使能从机全双工通信模式；
4. 置位 SPI_DMA_AFIFO_RST、SPI_BUF_AFIFO_RST 和 SPI_RX_AFIFO_RST 复位 buffer；
5. 置位 SPI_DMA_TX_ENA/SPI_DMA_RX_ENA。配置 GDMA TX/RX 链表，并启动 GDMA TX/RX 引擎，更多描述见章节 30.5.6 和章节 30.5.7；
6. 置位寄存器 SPI_DMA_CONF_REG 中 SPI_RX_EOF_EN 位。在寄存器 SPI_MS_DLEN_REG 的 SPI_MS_DATA_BITLEN[17:0] 中配置 DMA 接收数据长度（单位：字节）；
7. 置位寄存器 SPI_DMA_CONF_REG 中 SPI_DMA_SLV_SEG_TRANS_EN，使能从机连续传输；
8. 置位 GDMA_IN_SUC_EOF_CH*n*_INT_ENA 使能中断，然后等待 GDMA_IN_SUC_EOF_CH*n*_INT 中断。

30.6 CS 建立时间和保持时间控制

SPI CS 建立时间和保持时间对于满足各种 SPI 设备（如 flash 或 PSRAM）的时序要求非常重要。

CS 建立时间为 CS 下降沿至 SPI_CLK 第一个锁存边沿的时间。模式 0 和模式 3 的第一锁存边沿为上升沿，模式 1 和模式 2 的第一锁存边沿为下降沿。

CS 保持时间为 SPI_CLK 最后一个锁存边沿到 CS 上升沿之间的时间。

从机模式下，CS 建立时间和保持时间应大于 0.5 x T_SPI_CLK，否则 SPI 传输可能出错。这里的 T_SPI_CLK 指 SPI_CLK 时钟周期。

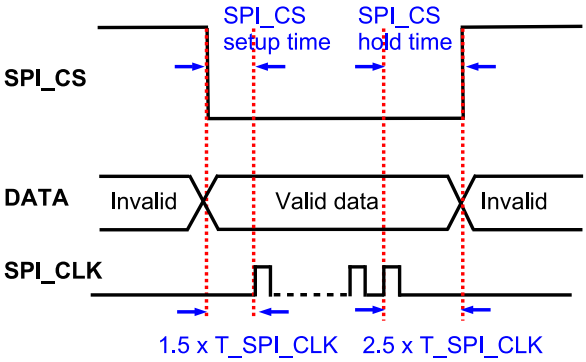
主机模式下，CS 建立时间由寄存器 SPI_USER_REG 中的 SPI_CS_SETUP 位和寄存器 SPI_USER1_REG 中的 SPI_CS_SETUP_TIME 位控制：

- 清零 SPI_CS_SETUP，则 SPI CS 建立时间为 0.5 x T_SPI_CLK。
- 置位 SPI_CS_SETUP，则 SPI CS 建立时间为 (SPI_CS_SETUP_TIME + 1.5) x T_SPI_CLK。

CS 保持时间由寄存器 SPI_USER_REG 中的 SPI_CS_HOLD 位和寄存器 SPI_USER1_REG 中的 SPI_CS_HOLD_TIME 位控制：

- 清零 SPI_CS_HOLD，则 SPI CS 保持时间为 0.5 x T_SPI_CLK。
- 置位 SPI_CS_HOLD，则 SPI CS 保持时间为 (SPI_CS_HOLD_TIME + 1.5) x T_SPI_CLK。

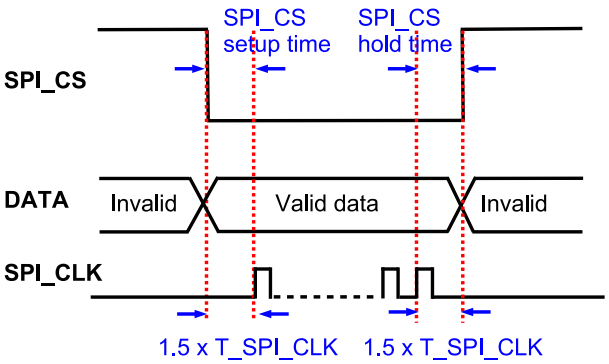
图 30.6-1 和图 30.6-2 所示为访问外部 RAM 和 flash 时推荐的 CS 时序配置和寄存器配置。



Register Configurations:

SPI_CS_SETUP = 1; SPI_CS_SETUP_TIME = 0;
SPI_CS_HOLD = 1; SPI_CS_HOLD_TIME = 1.

图 30.6-1. GP-SPI 访问外部 RAM 时推荐的 CS 时序配置



Register Configurations:

SPI_CS_SETUP = 1; SPI_CS_SETUP_TIME = 0;
SPI_CS_HOLD = 1; SPI_CS_HOLD_TIME = 0.

图 30.6-2. GP-SPI 访问 flash 时推荐的 CS 时序配置

30.7 GP-SPI 时钟控制

GP-SPI 中有以下四个时钟：

- clk_spi_mst：GP-SPI 模块时钟，由 PLL_CLK 或 XTAL_CLK 分频所得，见寄存器 SPI_MST_CLK_ACTIVE 和 SPI_MST_CLK_SEL。在 GP-SPI 主机模式下用于生成数据传输以及从机所需的 SPI_CLK 信号；
- clk_hclk：GP-SPI 模块的时序补偿时钟。如果 PLL_CLK 可用且置位了 SPI_TIMING_HCLK_ACTIVE，则 clk_hclk 时钟的频率为 160 MHz，否则，该时钟关闭。
- SPI_CLK：主机模式输出时钟；
- APB_CLK：用于寄存器配置的时钟。

主机模式下 GP-SPI 最高输出时钟频率为 $f_{clk\_spi\_mst}$。如果需要较低的时钟频率，可以采用如下分频方式：

$$f_{SPI\_CLK} = \frac{f_{clk\_spi\_mst}}{(SPI\_CLKCNT\_N + 1)(SPI\_CLKDIV\_PRE + 1)}$$

用户可配置寄存器 SPI_CLOCK_REG 中 SPI_CLKCNT_N 和 SPI_CLKDIV_PRE 设置分频系数。寄存器 SPI_CLOCK
_REG 中 SPI_CLK_EQU_SYSCLK 置 1 时，GP-SPI 的输出时钟频率为 $f_{clk\_spi\_mst}$。如果采用其它整数分频，则 SPI_CLK_EQU_SYSCLK 应置 0。

从机模式下，GP-SPI 支持的输入时钟频率 (f_{SPI_CLK}) 为：

- 如果 f_{APB_CLK} >= 60 MHz，则输入时钟频率为：f_{SPI_CLK} <= 60 MHz。
- 如果 f_{APB_CLK} < 60 MHz，则输入时钟频率为：f_{SPI_CLK} <= f_{APB_CLK}。

30.7.1 时钟相位和极性

SPI 协议支持四种时钟模式，即模式 0 ~ 3，见图 30.7-1 和图 30.7-2。注，图片来源于 SPI 协议。

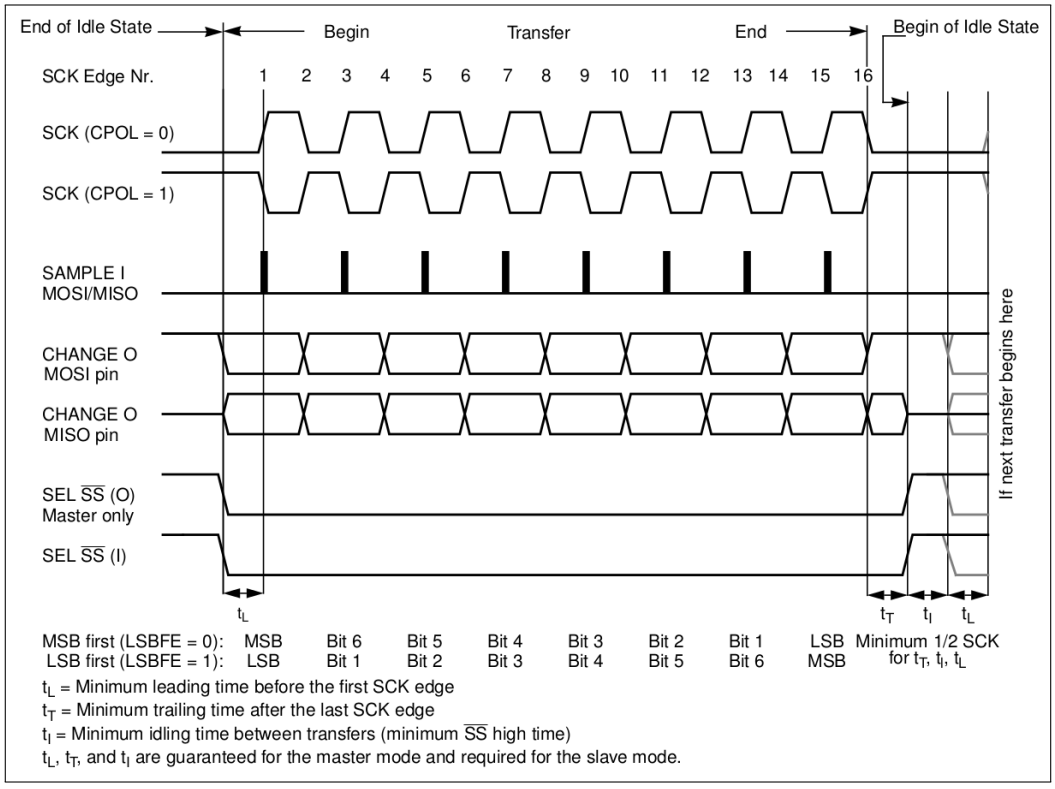


图 30.7-1. SPI 时钟模式 0 和时钟模式 2

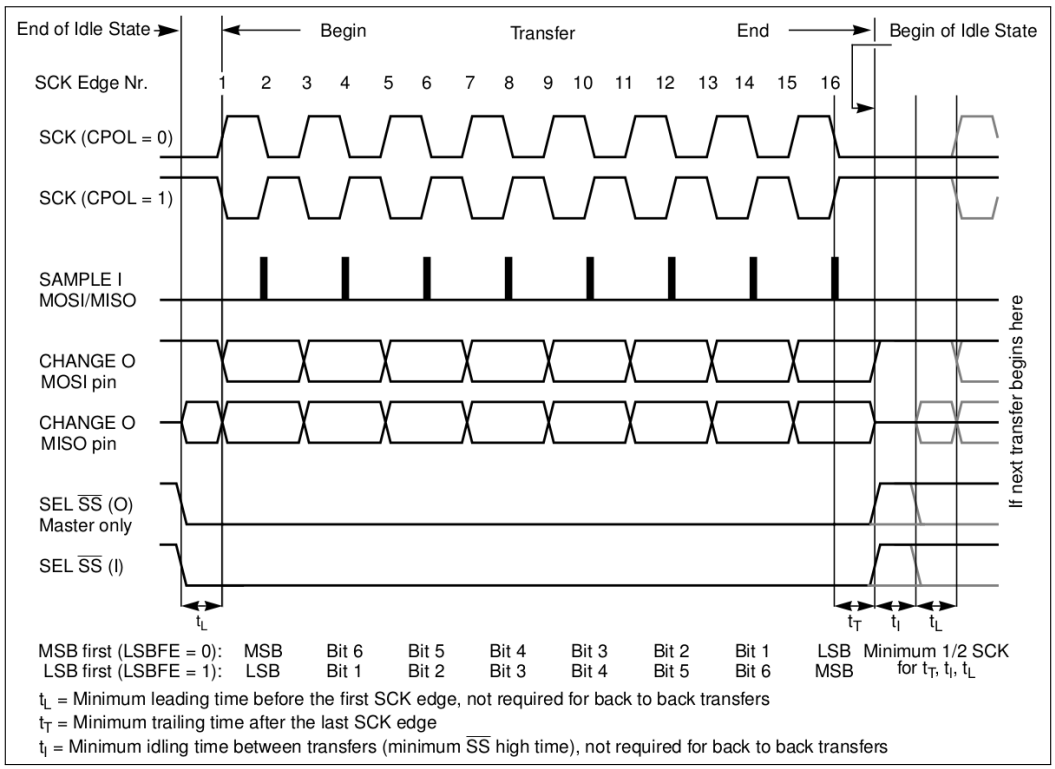


图 30.7-2. SPI 时钟模式 1 和时钟模式 3

1. 模式 0：CPOL = 0，CPHA = 0；SPI 处于空闲模式时，SCK 为 0；数据在 SCK 下降沿变化，在上升沿采样。第一个数据在 SCK 的第一个下降沿之前被移出。
2. 模式 1：CPOL = 0，CPHA = 1；SPI 处于空闲模式时，SCK 为 0；数据在 SCK 上升沿变化，在下降沿采样。
3. 模式 2：CPOL = 1，CPHA = 0；SPI 处于空闲模式时，SCK 为 1；数据在 SCK 上升沿变化，在下降沿采样。第一个数据在 SCK 的第一个上升沿之前被移出。
4. 模式 3：CPOL = 1，CPHA = 1；SPI 处于空闲模式时，SCK 为 1；数据在 SCK 下降沿变化，在上升沿采样。

30.7.2 主机模式下的时钟控制

GP-SPI 主机支持多种 SPI 时钟模式：模式 0 ~ 3。GP-SPI 极性和相位由寄存器 SPI_MISC_REG 中 SPI_CK_IDLE_EDGE 位和寄存器 SPI_USER_REG 中 SPI_CK_OUT_EDGE 位控制。SPI 时钟模式 0 ~ 3 的寄存器配置见表 30.7-1，可根据应用的路径延迟进行更改。

表 30.7-1. 主机模式下的时钟相位和极性配置

寄存器控制位	模式 0	模式 1	模式 2	模式 3
SPI_CK_IDLE_EDGE	0	0	1	1
SPI_CK_OUT_EDGE	0	1	1	0

此外，SPI_CLK_MODE 可用于选择 CS 拉高时 SPI_CLK 的上升沿个数：0、1、2 或 SPI_CLK 一直有效。

> **说明：**
> SPI_CLK_MODE 配置成 1 或 2 时，必须置位 SPI_CS_HOLD 且 SPI_CS_HOLD_TIME 的值需大于 1。

30.7.3 从机模式下的时钟控制

GP-SPI 从机也支持四种 SPI 时钟模式：即模式 0 ~ 3。寄存器 SPI_USER_REG 中 SPI_TSCK_I_EDGE 和 SPI_RSCK_I_EDGE 位可用于配置时钟极性和相位。数据的输出沿则由寄存器 SPI_SLAVE_REG 中的 SPI_CLK_MODE_13 位控制。寄存器具体配置见表 30.7-2。

表 30.7-2. 从机模式下的时钟相位和极性配置

寄存器控制位	模式 0	模式 1	模式 2	模式 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

30.8 GP-SPI 时序补偿

概述

SPI 输入输出信号可通过 GPIO 矩阵或 IO MUX 映射到芯片管脚，但 IO MUX 不支持时序调整。输入输出数据在 GPIO 矩阵模块中，可在上升沿或下降沿延迟 1 或 2 个 APB_CLK 周期。更多寄存器配置信息，见章节 *6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)*。

图 30.8-1 所示为 GP-SPI 主机模式下的时序补偿控制，包括以下路径：

- “CLK”：GP-SPI 总线时钟信号的输出路径。时钟由 SPI_CLK 输出控制模块发送，经过 GPIO 矩阵或 IO MUX，然后到达外部 SPI 设备。
- “IN”：GP-SPI 的数据输入路径，即图 30.8-1 中路径 3（紫色）。来自外部 SPI 设备的输入数据通过 GPIO 矩阵或 IO MUX，由时序模块（见图 30.5-2）进行调整，最后存储到 spi_rx_afifo。
- “OUT”：GP-SPI 的数据输出路径，即图 30.8-1 中路径 2（玫红色）。输出数据发送到时序模块，经过 GPIO 矩阵或 IO MUX，最后由外部 SPI 设备捕获。

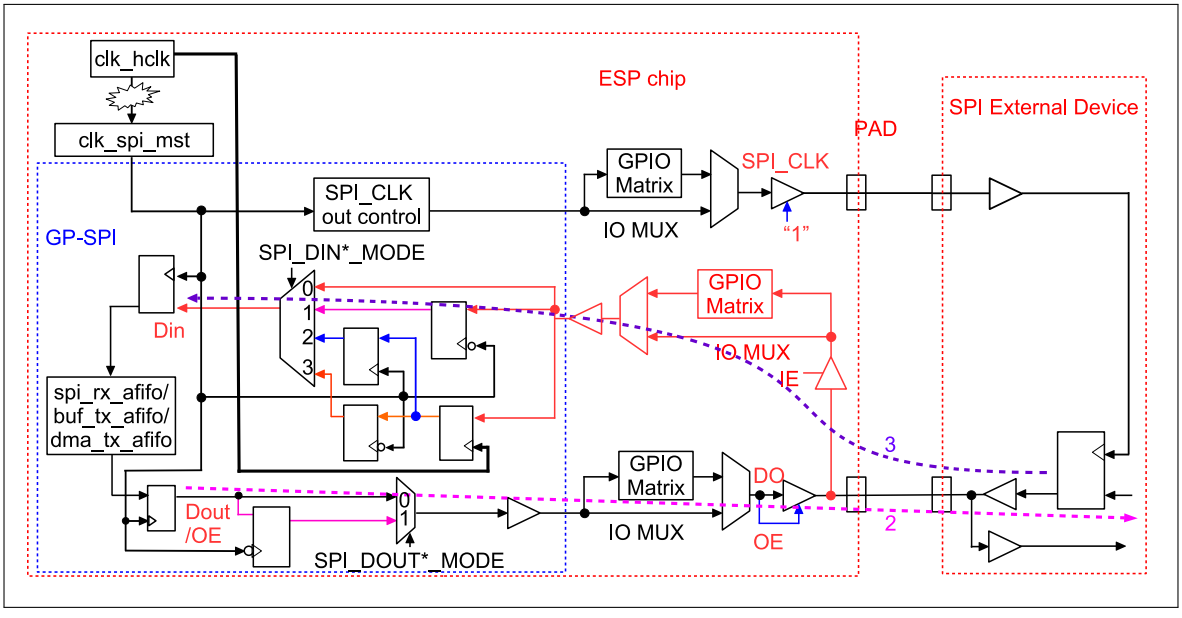


图 30.8-1. GP-SPI 主机模式下时序补偿控制图

时序调制模块同时适用于输入数据和输出数据，数据可在时钟上升沿或下降沿延迟整数个 clk_spi_mst 周期，即整数个 $T_{clk\_spi\_mst}$。

关键寄存器

- SPI_DIN_MODE_REG：用于选择输入数据的锁存沿
- SPI_DIN_NUM_REG：用于选择输入数据的延迟周期
- SPI_DOUT_MODE_REG：用于选择输出数据的锁存沿

时序补偿应用示例（以 GP-SPI2 为例）

图 30.8-2 所示为 GP-SPI2 主机模式下时序补偿示例。同时，DUMMY 周期长可更改，用于补偿实际的 I/O 线路延迟，从而提高 GP-SPI2 性能。

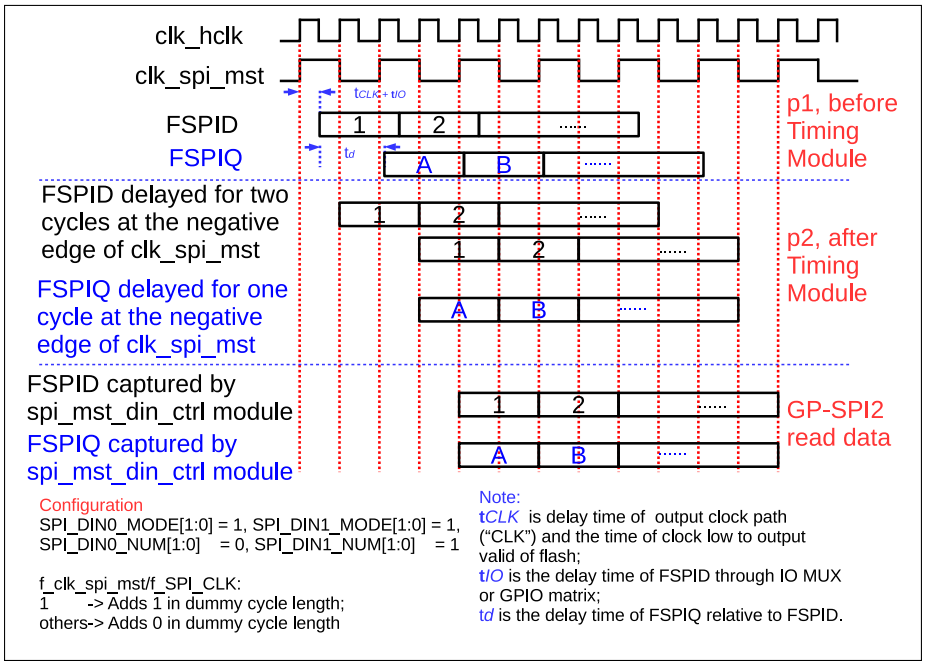


图 30.8-2. GP-SPI2 主机模式下时序补偿

图 30.8-2 中，"p1" 为时序模块输入数据的时间点；"p2" 为输出数据的时间点。由于 FSPIQ 输入数据并未与 FSPID 输入数据对齐，如果没有时间补偿的话，GP-SPI2 读取数据将会出错。

为了正确读取数据，需要进行如下配置。其中，假设 $f_{clk\_spi\_mst}$ 等于 f_{SPI_CLK}：

- 将 FSPID 在 clk_spi_mst 下降沿延迟两个时钟周期
- 将 FSPIQ 在 clk_spi_mst 下降沿延迟一个时钟周期
- 增加一个额外的 DUMMY 周期

在 GP-SPI 从机模式下，如果寄存器 SPI_SLAVE_REG 中 SPI_RSCK_DATA_OUT 置 1，则在锁存沿发送输出数据，即提前半个 SPI 时钟周期。上述功能可用于从机模式时序补偿。

30.9 GP-SPI2 和 GP-SPI3 功能差异

GP-SPI2 和 GP-SPI3 的功能差异如下：

- GP-SPI2 各个阶段，如 CMD、ADDR、DOUT 或 DIN，其通信模式可单独配置。在主机模式下，数据可配置成 1-bit、2-bit、4-bit 或 8-bit 模式；在从机模式下，可配置为 1-bit、2-bit 和 4-bit 模式。但 GP-SPI3 仅支持主机 1/2/4-bit 模式和从机 1/2/4-bit 模式。
- DMA 控制的分段配置传输仅在 GP-SPI2 中可用。因此，GP-SPI3 中没有 CONF 阶段。
- GP-SPI2 I/O 可用过 GPIO 交换矩阵或 IO MUX 映射到芯片管脚。但 GP-SPI3 信号线仅可通过 GPIO 交换矩阵进行配置。
- GP-SPI2 在主机模式下具有六个 CS 信号。GP-SPI3 在主机模式下仅有三个 CS 信号。

除此之外，GP-SPI2 和 GP-SPI3 的功能相同。GP-SPI2 可使用所有通用 GP-SPI 寄存器，而 GP-SPI3 仅可使用部分 GP-SPI 寄存器，更多信息见表 30.9-1。

表 30.9-1. 对 GP-SPI3 无效的字段

无效寄存器	无效字段
SPI_USER_REG	SPI_OPI_MODE SPI_FWRITE_OCT
SPI_CTRL_REG	SPI_FADDR_OCT SPI_FCMD_OCT SPI_FREAD_OCT
SPI_MISC_REG	SPI_CS3_DIS SPI_CS4_DIS SPI_CS5_DIS SPI_MASTER_CS_POL[5:3]
SPI_DIN_MODE_REG	SPI_DIN4_MODE SPI_DIN5_MODE SPI_DIN6_MODE SPI_DIN7_MODE
SPI_DIN_NUM_REG	SPI_DIN4_NUM SPI_DIN5_NUM SPI_DIN6_NUM SPI_DIN7_NUM
SPI_DOUT_MODE_REG	SPI_DOUT4_MODE SPI_DOUT5_MODE SPI_DOUT6_MODE SPI_DOUT7_MODE

GP-SPI3 1/2/4-bit 模式功能与 GP-SPI2 相同，对应的寄存器配置规则也相同。GP-SPI3 接口相当于 1/2/4-bit 模式的 GP-SPI2 接口，但不支持 DMA 控制的分段配置传输。

30.10 中断

中断描述

GP-SPI 提供：SPI_INTR_2/3 中断接口。一次 SPI 传输结束时，GP-SPI 即生成一次中断。这里的中断可能是以下一个或多个中断：

- SPI_DMA_INFIFO_FULL_ERR_INT：GDMA RX FIFO 小于实际传输的数据长度时即触发此中断。
- SPI_DMA_OUTFIFO_EMPTY_ERR_INT：GDMA TX FIFO 小于实际传输的数据长度时即触发此中断。
- SPI_SLV_EX_QPI_INT：GP-SPI 从机模式下，正确接收 Ex_QPI 命令，且 SPI 传输结束即触发此中断。
- SPI_SLV_EN_QPI_INT：在 GP-SPI 从机模式下，正确接收 En_QPI 命令，且 SPI 传输结束即触发此中断。
- SPI_SLV_CMD7_INT：GP-SPI 从机模式下，正确接收 CMD7 命令，且 SPI 传输结束即触发此中断。
- SPI_SLV_CMD8_INT：GP-SPI 从机模式下，正确接收 CMD8 命令，且 SPI 传输结束即触发此中断。
- SPI_SLV_CMD9_INT：GP-SPI 从机模式下，正确接收 CMD9 命令，且 SPI 传输结束即触发此中断。
- SPI_SLV_CMDA_INT：GP-SPI 从机模式下，正确接收 CMDA 命令，且 SPI 传输结束即触发此中断。
- SPI_SLV_RD_DMA_DONE_INT：从机模式下，Rd_DMA 传输结束即触发此中断。
- SPI_SLV_WR_DMA_DONE_INT：从机模式下，Wr_DMA 传输结束即触发此中断。
- SPI_SLV_RD_BUF_DONE_INT：从机模式下，Rd_BUF 传输结束即触发此中断。
- SPI_SLV_WR_BUF_DONE_INT：从机模式下，Wr_BUF 传输结束即触发此中断。
- SPI_TRANS_DONE_INT：主从机模式下，SPI 总线传输结束均会触发此中断。
- SPI_DMA_SEG_TRANS_DONE_INT：GP-SPI 从机连续传输模式下，End_SEG_TRANS 传输结束即触发此中断。主机模式下，分段配置传输结束也将触发此中断。
- SPI_SEG_MAGIC_ERR_INT：在主机分段配置传输模式下，CONF buffer 中的 Magic 值有误即触发此中断。仅对 GP-SPI2 有效。
- SPI_MST_RX_AFIFO_WFULL_ERR_INT：GP-SPI 主机模式下，如果发生 RX AFIFO write-full 错误，即触发此中断。
- SPI_MST_TX_AFIFO_REMPTY_ERR_INT：GP-SPI 主机模式下，如果发生 TX AFIFO read-empty 错误即触发此中断。
- SPI_SLV_CMD_ERR_INT：GP-SPI 从机模式下，如果接收命令值 GP-SPI 不支持，即触发此中断。
- SPI_APP2_INT：置位 SPI_APP2_INT_SET 触发该中断。仅用于用户自定义的功能。
- SPI_APP1_INT：置位 SPI_APP1_INT_SET 触发该中断。仅用于用户自定义的功能。

主机模式和从机模式分别用到的中断

表 30.10-1 和表 30.10-2 分别列出了 GP-SPI 在主机模式下和从机模式下用到的中断。置位寄存器 SPI_DMA_INT_ENA_REG 中 SPI_*_INT_ENA 位，使能相应中断，并等待 SPI_INT 中断。传输结束时，将触发相关中断。注意，在下次传输之前，需软件清除中断。

表 30.10-1. GP-SPI 主机模式中断表

传输类型	通信模式	控制方式	中断
单次传输	全双工	DMA	GDMA_IN_SUC_EOF_CH*n*_INT [1]
		CPU	SPI_TRANS_DONE_INT [2]
	半双工 MOSI 模式	DMA	SPI_TRANS_DONE_INT
		CPU	SPI_TRANS_DONE_INT
	半双工 MISO 模式	DMA	GDMA_IN_SUC_EOF_CH*n*_INT

接上页

表 30.10-1 – 见下页

传输类型	通信模式	控制方式	中断
		CPU	SPI_TRANS_DONE_INT
分段配置传输	全双工	DMA	SPI_DMA_SEG_TRANS_DONE_INT [3]
		CPU	不支持
	半双工 MOSI 模式	DMA	SPI_DMA_SEG_TRANS_DONE_INT
		CPU	不支持
	半双工 MISO 模式	DMA	SPI_DMA_SEG_TRANS_DONE_INT
		CPU	不支持

[1] 如果触发了 GDMA_IN_SUC_EOF_CH*n*_INT 中断，则表示 GP-SPI 的所有 RX 数据已保存至 RX buffer，且所有 TX 数据已发送至从机。

[2] CS 拉高，则将触发 SPI_TRANS_DONE_INT 中断，表明主机与从机已完成 SPI_W0_REG ~ SPI_W15_REG 的数据交换。

[3] 如果触发了 SPI_DMA_SEG_TRANS_DONE_INT 中断，则表明整个分段配置传输，包括若干个传输事务，已完成。即 RX 数据已全部存入 RX buffer 且所有 TX 数据已发送完毕。

表 30.10-2. GP-SPI 从机模式中断表

传输类型	通信模式	控制方式	中断
单次传输	全双工	DMA	GDMA_IN_SUC_EOF_CH*n*_INT [1]
		CPU	SPI_TRANS_DONE_INT [2]
	半双工 MOSI 模式	DMA (Wr_DMA)	GDMA_IN_SUC_EOF_CH*n*_INT[3]
		CPU (Wr_BUF)	SPI_TRANS_DONE_INT[4]
	半双工 MISO 模式	DMA (Rd_DMA)	SPI_TRANS_DONE_INT[5]
		CPU (Rd_BUF)	SPI_TRANS_DONE_INT[6]
从机连续传输	Full-duplex	DMA	GDMA_IN_SUC_EOF_CH*n*_INT[7]
		CPU	不支持[8]
	半双工 MOSI 模式	DMA (Wr_DMA)	SPI_DMA_SEG_TRANS_DONE_INT[9]
		CPU (Wr_BUF)	不支持[10]
	半双工 MISO 模式	DMA (Rd_DMA)	SPI_DMA_SEG_TRANS_DONE_INT[11]
		CPU (Rd_BUF)	不支持[12]

[1] 如果触发了 GDMA_IN_SUC_EOF_CH*n*_INT 中断，则表示所有 RX 数据已保存至 RX buffer，且所有 TX 数据已发送至从机。
[2] CS 拉高，则将触发 SPI_TRANS_DONE_INT 中断，表明主机与从机已完成 SPI_W0_REG ~ SPI_W15_REG 的数据交换。
[3] 触发 SPI_SLV_WR_DMA_DONE_INT 中断仅表示 SPI 总线上的数据传输已完成，但并不能保证所有入栈数据已存至 RX buffer。因此，推荐使用 GDMA_IN_SUC_EOF_CH*n*_INT 中断。
[4] 或等待 SPI_SLV_WR_BUF_DONE_INT 中断。
[5] 或等待 SPI_SLV_RD_DMA_DONE_INT 中断。
[6] 或等待 SPI_SLV_RD_BUF_DONE_INT 中断。
[7] 传输开始前，从机应在 SPI_MS_DATA_BITLEN 中设置读数据的总长度。并在中断程序结束前，置位 SPI_RX_EOF_EN。
[8] 主机和从机需定义连续传输结束的方式，比如配置 GPIO 用作中断等。
[9] 主机发送 COM5 结束连续传输，或从机在 SPI_MS_DATA_BITLEN 中配置总的读数据长度，然后等待 GDMA_IN_SUC _EOF_CH*n*_INT 中断。
[10] 半双工 Wr_BUF 单次传输也可用于 DMA 控制的从机连续传输中。
[11] 主机发送 End_SEG_TRAN 结束从机连续传输。
[12] 半双工 Rd_BUF 单次传输也可用于 DMA 控制的从机连续传输中。

30.11 寄存器列表

本小节的所有地址均为相对于 SPI2/SPI3 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 *4 系统和存储器* 中的表 4.3-3。

请查看章节 *寄存器的访问类型*，了解“访问”列缩写的含义。

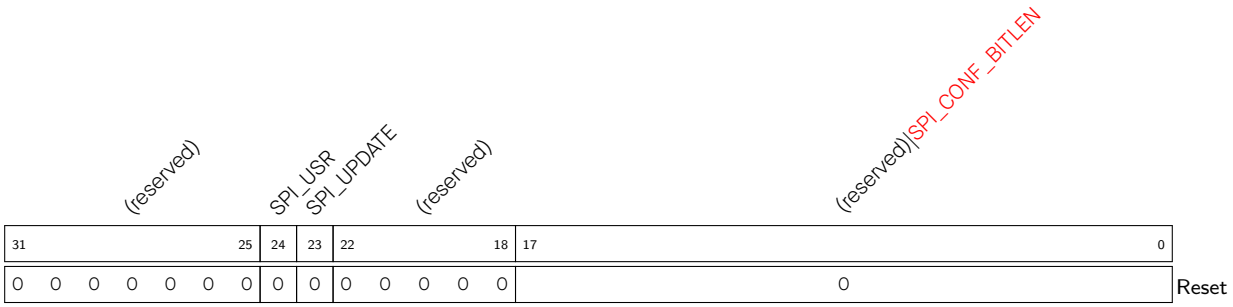
名称	描述	SPI2 地址	SPI3 地址	访问
自定义控制寄存器				
SPI_CMD_REG	命令控制寄存器	0x0000	0x0000	varies
SPI_ADDR_REG	地址值寄存器	0x0004	0x0004	R/W
SPI_USER_REG	SPI 用户控制寄存器	0x0010	0x0010	varies
SPI_USER1_REG	SPI 用户控制寄存器 1	0x0014	0x0014	R/W
SPI_USER2_REG	SPI 用户控制寄存器 2	0x0018	0x0018	R/W
控制和配置寄存器				
SPI_CTRL_REG	SPI 控制寄存器	0x0008	0x0008	R/W
SPI_MS_DLEN_REG	SPI 数据位长控制寄存器	0x001C	0x001C	R/W
SPI_MISC_REG	SPI MISC 寄存器	0x0020	0x0020	R/W
SPI_DMA_CONF_REG	SPI DMA 控制寄存器	0x0030	0x0030	varies
SPI_SLAVE_REG	SPI 从机控制寄存器	0x00E0	0x00E0	varies
SPI_SLAVE1_REG	SPI 从机控制寄存器 1	0x00E4	0x00E4	R/W/SS
时钟控制寄存器				
SPI_CLOCK_REG	SPI 时钟控制寄存器	0x000C	0x000C	R/W
SPI_CLK_GATE_REG	SPI 模块时钟和寄存器时钟控制	0x00E8	0x00E8	R/W
时序寄存器				

名称	描述	SPI2 地址	SPI3 地址	访问
SPI_DIN_MODE_REG	SPI 输入延迟模式配置寄存器	0x0024	0x0024	R/W
SPI_DIN_NUM_REG	SPI 输入延迟周期配置寄存器	0x0028	0x0028	R/W
SPI_DOUT_MODE_REG	SPI 输出延迟模式配置寄存器	0x002C	0x002C	R/W
中断寄存器				
SPI_DMA_INT_ENA_REG	SPI 中断使能寄存器	0x0034	0x0034	R/W
SPI_DMA_INT_CLR_REG	SPI 中断清除寄存器	0x0038	0x0038	WT
SPI_DMA_INT_RAW_REG	SPI 原始中断寄存器	0x003C	0x003C	R/ WTC/ SS
SPI_DMA_INT_ST_REG	SPI 中断状态寄存器	0x0040	0x0040	RO
SPI_DMA_INT_SET_REG	SPI 中断软件置位寄存器	0x0044	0x0044	WT
CPU 数据 Buffer				
SPI_W0_REG	SPI CPU 控制的 buffer 0	0x0098	0x0098	R/W/SS
SPI_W1_REG	SPI CPU 控制的 buffer 1	0x009C	0x009C	R/W/SS
SPI_W2_REG	SPI CPU 控制的 buffer 2	0x00A0	0x00A0	R/W/SS
SPI_W3_REG	SPI CPU 控制的 buffer 3	0x00A4	0x00A4	R/W/SS
SPI_W4_REG	SPI CPU 控制的 buffer 4	0x00A8	0x00A8	R/W/SS
SPI_W5_REG	SPI CPU 控制的 buffer 5	0x00AC	0x00AC	R/W/SS
SPI_W6_REG	SPI CPU 控制的 buffer 6	0x00B0	0x00B0	R/W/SS
SPI_W7_REG	SPI CPU 控制的 buffer 7	0x00B4	0x00B4	R/W/SS
SPI_W8_REG	SPI CPU 控制的 buffer 8	0x00B8	0x00B8	R/W/SS
SPI_W9_REG	SPI CPU 控制的 buffer 9	0x00BC	0x00BC	R/W/SS
SPI_W10_REG	SPI CPU 控制的 buffer 10	0x00C0	0x00C0	R/W/SS
SPI_W11_REG	SPI CPU 控制的 buffer 11	0x00C4	0x00C4	R/W/SS
SPI_W12_REG	SPI CPU 控制的 buffer 12	0x00C8	0x00C8	R/W/SS
SPI_W13_REG	SPI CPU 控制的 buffer 13	0x00CC	0x00CC	R/W/SS
SPI_W14_REG	SPI CPU 控制的 buffer 14	0x00D0	0x00D0	R/W/SS
SPI_W15_REG	SPI CPU 控制的 buffer 15	0x00D4	0x00D4	R/W/SS
版本寄存器				
SPI_DATE_REG	版本控制寄存器	0x00F0	0x00F0	R/W

30.12 寄存器

本小节的所有地址均为相对于 SPI2/SPI3 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 30.1. SPI_CMD_REG (0x0000)

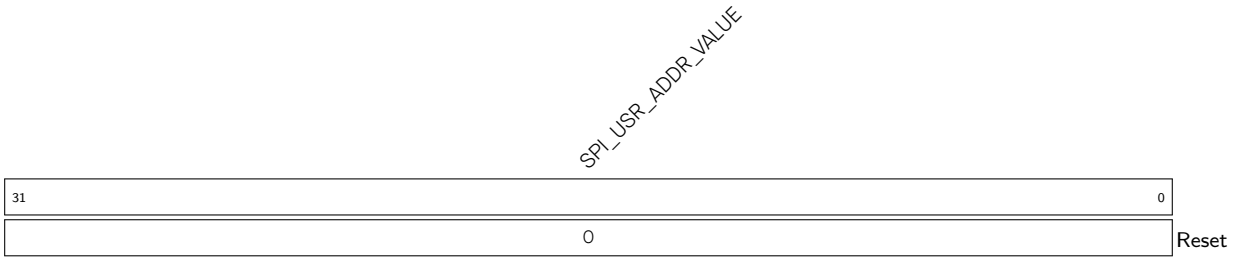


SPI_CONF_BITLEN（仅对 SPI2 有效） 定义 SPI CONF 阶段的 APB_CLK 周期。可在 CONF 阶段配置。(R/W)

SPI_UPDATE 置位此位，将 SPI 寄存器从 APB 时钟域同步到 SPI 模块时钟域。该位仅用于 SPI 主机模式。(WT)

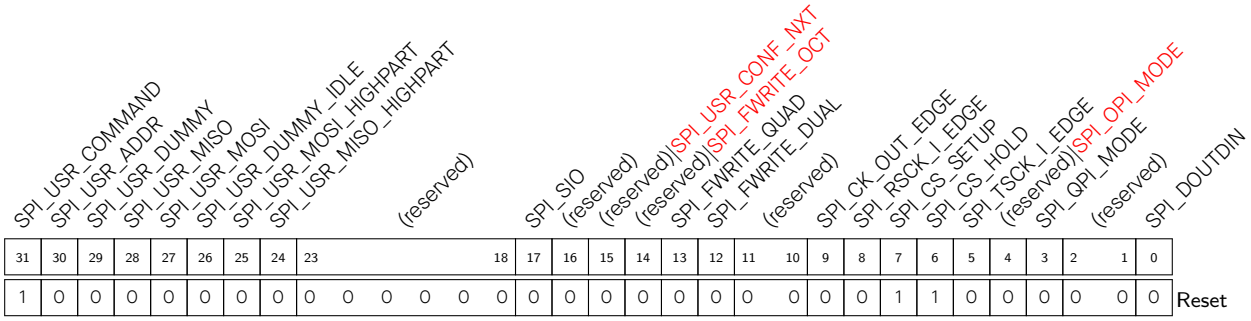
SPI_USR 使能用户自定义命令。置位此位将触发一次 SPI 操作。操作完成后，此位自动清零。1：使能；0：禁用。CONF_buf 不可更改该配置。(R/W/SC)

Register 30.2. SPI_ADDR_REG (0x0004)



SPI_USR_ADDR_VALUE 从机地址。可在 CONF 阶段配置。(R/W)

Register 30.3. SPI_USER_REG (0x0010)



SPI_DOUTDIN 使能全双工模式。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_QPI_MODE 1：使能 QPI 模式。0：禁用 QPI 模式。SPI 主机模式和从机模式均支持该配置。可在 CONF 阶段配置。(R/W/SS/SC)

SPI_OPI_MODE（仅对 SPI2 有效） 1：使能 OPI 模式，则所有阶段均采用 8-bit 模式。0：禁用 OPI 模式。仅有 SPI 主机模式支持该配置。可在 CONF 阶段配置。(R/W)

SPI_TSCK_I_EDGE 在从机模式下，此位可用于配置 SPI 四种时钟模式，见章节 30.7.3。(R/W)

SPI_CS_HOLD SPI 处于完成 (DONE) 阶段时，SPI CS 保持低电平。1：使能此功能；0：禁用此功能。可在 CONF 阶段配置。(R/W)

SPI_CS_SETUP SPI 处于准备 (PREP) 阶段时，使能 SPI CS。1：启用此功能；0：禁用此功能。可在 CONF 阶段配置。(R/W)

SPI_RSCK_I_EDGE 在从机模式下，此位可用于配置四种 SPI 时钟模式，见章节 30.7.3。(R/W)

SPI_CK_OUT_EDGE 该位与 SPI_MOSI_DELAY_MODE 一起用于控制 MOSI 信号延迟模式。可在 CONF 阶段配置。(R/W)

SPI_FWRITE_DUAL 在写操作 (DOUT) 阶段，读数据的方式为 2-bit 方式。可在 CONF 阶段配置。(R/W)

SPI_FWRITE_QUAD 在写操作 (DOUT) 阶段，读数据的方式为 4-bit 方式。可在 CONF 阶段配置。(R/W)

SPI_FWRITE_OCT（仅对 SPI2 有效） 在写操作 (DOUT) 阶段，读数据的方式为 8-bit 方式。可在 CONF 阶段配置。(R/W)

SPI_USR_CONF_NXT（仅对 SPI2 有效） 使能下一次传输事务的 CONF 阶段。可在 CONF 阶段配置。(R/W)

- 置位此位，则本次分段配置传输继续进行，开始下一次传输事务。
- 清除此位，则当前传输事务结束后，本次分段配置传输结束。或者，当前的传输模式不是分段配置传输。

SPI_SIO 配置三线半双工通信。MOSI 和 MISO 信号共用一个管脚。1：使能三线半双工通信；0：禁用三线半双工通信。可在 CONF 阶段配置。(R/W)

SPI_USR_MISO_HIGHPART 在读数据阶段，仅访问高位 buffer：SPI_W8_REG ~ SPI_W15_REG。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

见下页

Register 30.3. SPI_USER_REG (0x0010)

接上页

SPI_USR_MOSI_HIGHPART 在写数据阶段，仅访问高位 buffer：SPI_W8_REG ~ SPI_W15_REG。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_USR_DUMMY_IDLE 置位此位，在 DUMMY 阶段禁用 SPI 时钟。可在 CONF 阶段配置。(R/W)

SPI_USR_MOSI 置位此位，使能一次操作的写数据 (DOUT) 阶段。可在 CONF 阶段配置。(R/W)

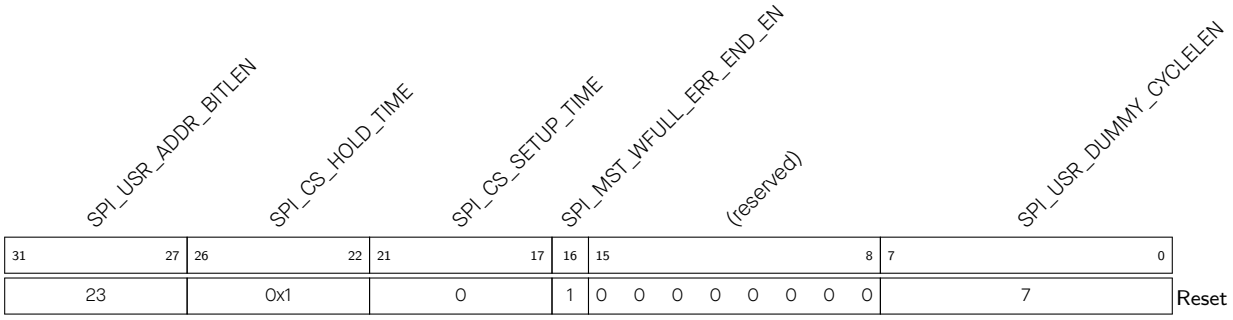
SPI_USR_MISO 置位此位，使能一次操作的读数据 (DIN) 阶段。可在 CONF 阶段配置。(R/W)

SPI_USR_DUMMY 置位此位，使能一次操作的 DUMMY 阶段。可在 CONF 阶段配置。(R/W)

SPI_USR_ADDR 置位此位，使能一次操作的地址 (ADDR) 阶段。可在 CONF 阶段配置。(R/W)

SPI_USR_COMMAND 置位此位，使能一次操作的命令 (CMD) 阶段。可在 CONF 阶段配置。(R/W)

Register 30.4. SPI_USER1_REG (0x0014)



SPI_USR_DUMMY_CYCLELEN DUMMY 阶段的时长，单位：SPI_CLK 时钟周期。寄存器值为（实际需要的周期数 - 1）。可在 CONF 阶段配置。(R/W)

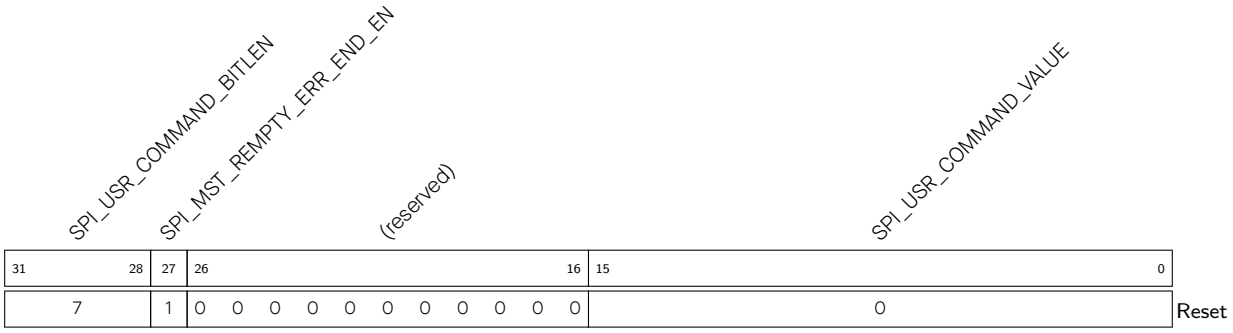
SPI_MST_WFULL_ERR_END_EN 1：在 GP-SPI 主机全双工或半双工模式下，如果发生 SPI RX AFIFO 满错误，则 SPI 传输将终止。0：在 GP-SPI 主机全双工或半双工模式下，如果发生 SPI RX AFIFO 满错误，SPI 传输将不被终止。(R/W)

SPI_CS_SETUP_TIME 准备 (PREP) 阶段的时长，单位：SPI_CLK 时钟周期。此值等于预期周期数 - 1。此字段与 SPI_CS_SETUP 搭配使用。可在 CONF 阶段配置。(R/W)

SPI_CS_HOLD_TIME CS 的延迟周期。单位：SPI_CLK 时钟周期。此字段与 SPI_CS_HOLD 搭配使用。可在 CONF 阶段配置。(R/W)

SPI_USR_ADDR_BITLEN 地址阶段的位长。此值为（预期位数 - 1）。可在 CONF 阶段配置。(R/W)

Register 30.5. SPI_USER2_REG (0x0018)

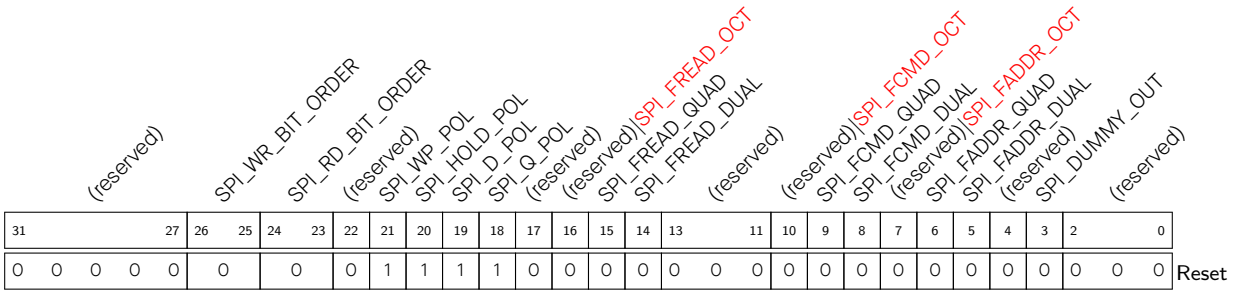


SPI_USR_COMMAND_VALUE 命令值。可在 CONF 阶段配置。(R/W)

SPI_MST_REMPTY_ERR_END_EN 1：在 GP-SPI 主机全双工或半双工模式下，如果发生 SPI TX AFIFO 空错误，则 SPI 传输将终止。0：在 GP-SPI 主机全双工或半双工模式下，如果发生 SPI TX AFIFO 空错误，则 SPI 传输将不会被终止。(R/W)

SPI_USR_COMMAND_BITLEN 命令阶段的位长。此值为（预期位长 - 1）。可在 CONF 阶段配置。(R/W)

Register 30.6. SPI_CTRL_REG (0x0008)



SPI_DUMMY_OUT 可在 CONF 阶段配置。(R/W)

- **SPI2**
  - 0：在 DUMMY 阶段，不输出 FSPI 总线信号。
  - 1：在 DUMMY 阶段，输出 FSPI 总线信号。
- **SPI3**：表示在 DUMMY 阶段，SPI3 输出的信号电平。

SPI_FADDR_DUAL 在地址 (ADDR) 阶段采用 2-bit 模式。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_FADDR_QUAD 在地址 (ADDR) 阶段采用 4-bit 模式。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_FADDR_OCT（仅对 SPI2 有效） 在地址 (ADDR) 阶段采用 8-bit 模式。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_FCMD_DUAL 在命令 (CMD) 阶段采用 2-bit 模式。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_FCMD_QUAD 在命令 (CMD) 阶段采用 4-bit 模式。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_FCMD_OCT (for SP2 only) 在命令 (CMD) 阶段采用 8-bit 模式。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_FREAD_DUAL 在读数据 (DIN) 阶段采用 2-bit 模式。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_FREAD_QUAD 在读数据 (DIN) 阶段采用 4-bit 模式。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_FREAD_OCT (for SP2 only) 在读数据 (DIN) 阶段采用 8-bit 模式。1：使能；0：禁用。可在 CONF 阶段配置。(R/W)

SPI_Q_POL 此位用于设置 MISO 的极性。1：高；0：低。可在 CONF 阶段配置。(R/W)

SPI_D_POL 此位用于设置 MOSI 的极性。1：高；0：低。可在 CONF 阶段配置。(R/W)

见下页

Register 30.6. SPI_CTRL_REG (0x0008)

接上页

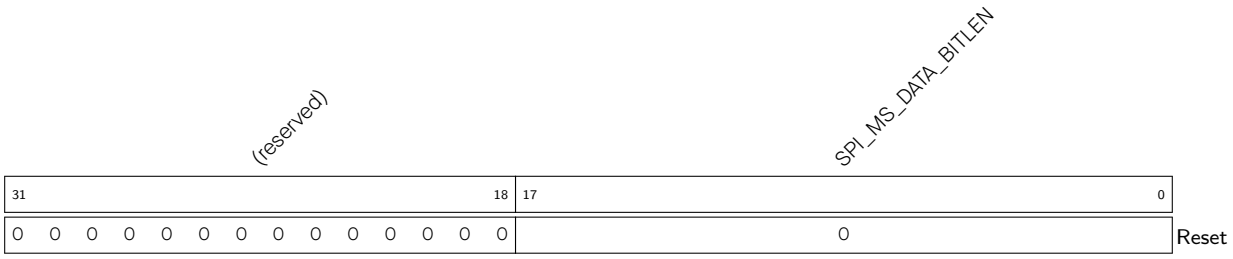
SPI_HOLD_POL 此位用于设置在 SPI 空闲状态下，SPI_HOLD 的输出值。1：输出高电平；0：输出低电平。可在 CONF 阶段配置。(R/W)

SPI_WP_POL 此位用于设置在 SPI 空闲状态下，WP 信号的输出值。1：输出高电平；0：输出低电平。可在 CONF 阶段配置。(R/W)

SPI_RD_BIT_ORDER 在读数据 (MISO) 阶段，1：先读低有效位；0：先读高有效位。可在 CONF 阶段配置。(R/W)

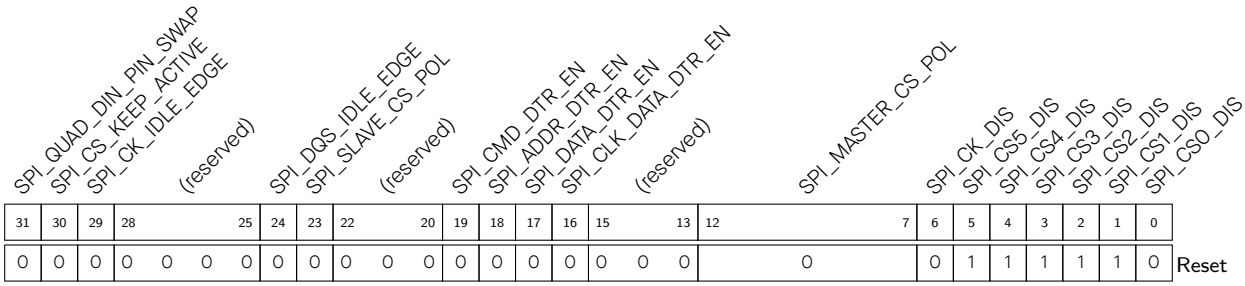
SPI_WR_BIT_ORDER 在命令 (CMD)、地址 (ADDR) 和写数据 (MOSI) 阶段，1：先发送低有效位；0：先发送高有效位。可在 CONF 阶段配置。(R/W)

Register 30.7. SPI_MS_DLEN_REG (0x001C)



SPI_MS_DATA_BITLEN 该字段用于配置主机模式下 DMA 控制或 CPU 控制的 SPI 传输的数据位长。也可用于配置从机模式下 DMA 控制的传输中接收数据的位长。该值等于需要的位长 - 1。可在 CONF 阶段配置。(R/W)

Register 30.8. SPI_MISC_REG（仅对 SPI2 有效） (0x0020)



SPI_CS0_DIS SPI CS0 管脚使能。1：禁用 CS0；0：SPI CS0 信号来自 CS0 管脚或输出至 CS0 管脚。可在 CONF 阶段配置。(R/W)

SPI_CS1_DIS SPI CS1 管脚使能。1：禁用 CS1；0：SPI CS1 信号来自 CS1 管脚或输出至 CS1 管脚。可在 CONF 阶段配置。(R/W)

SPI_CS2_DIS SPI CS2 管脚使能。1：禁用 CS2；0：SPI CS2 信号来自 CS2 管脚或输出至 CS2 管脚。可在 CONF 阶段配置。(R/W)

SPI_CS3_DIS SPI CS3 管脚使能。1：禁用 CS3；0：SPI CS3 信号来自 CS3 管脚或输出至 CS3 管脚。可在 CONF 阶段配置。(R/W)

SPI_CS4_DIS SPI CS4 管脚使能。1：禁用 CS4；0：SPI CS4 信号来自 CS4 管脚或输出至 CS4 管脚。可在 CONF 阶段配置。(R/W)

SPI_CS5_DIS SPI CS5 管脚使能。1：禁用 CS5；0：SPI CS5 信号来自 CS5 管脚或输出至 CS5 管脚。可在 CONF 阶段配置。(R/W)

SPI_CK_DIS 1：停止 SPI_CLK 输出信号；0：使能 SPI_CLK 输出信号。可在 CONF 阶段配置。(R/W)

SPI_MASTER_CS_POL 主机模式下，SPI_MASTER_CS_POL[*i*] 用于配置 SPI CS*i* 的极性，*i* = 0 ~ 5。0：CS*i* 低电平有效。1：CS*i* 高电平有效。可在 CONF 阶段配置。(R/W)

SPI_CLK_DATA_DTR_EN 1：在 SPI 主机模式下，SPI_CLK、DATA 和 SPI_DQS 采用 DDR 模式。0：在 SPI 主机模式下，仅 SPI_DQS 采用 DDR 模式。该位与 bit 17/18/19 一起使用。(R/W)

SPI_DATA_DTR_EN 1：SPI 在主机 1/2/4/8-bit 模式下，DIN 和 DOUT 阶段的时钟和数据均采用 DDR 模式。0：SPI DIN 和 DOUT 阶段的时钟和数据均采用 SDR 模式。可在 CONF 阶段配置。(R/W)

SPI_ADDR_DTR_EN 1：SPI 在主机 1/2/4/8-bit 模式下，SPI_SEND_ADDR 阶段的时钟和数据均采用 DDR 模式。0：SPI_SEND_ADDR 阶段的时钟和数据模式均采用 SDR 模式。可在 CONF 阶段配置。(R/W)

SPI_CMD_DTR_EN 1：SPI 在主机 1/2/4/8-bit 模式下，SPI_SEND_CMD 阶段的时钟和数据均采用 DDR 模式。0：SPI_SEND_CMD 阶段的时钟和数据模式均采用 SDR 模式。可在 CONF 阶段配置。(R/W)

SPI_SLAVE_CS_POL 选择 SPI 从机输入信号 CS 的极性。1：反相；0：保持不变。可在 CONF 阶段配置。(R/W)

SPI_DQS_IDLE_EDGE SPI_DQS 的默认值。0：低电平；1：高电平。可在 CONF 阶段配置。(R/W)

见下页

Register 30.8. SPI_MISC_REG（仅对 SPI2 有效） (0x0020)

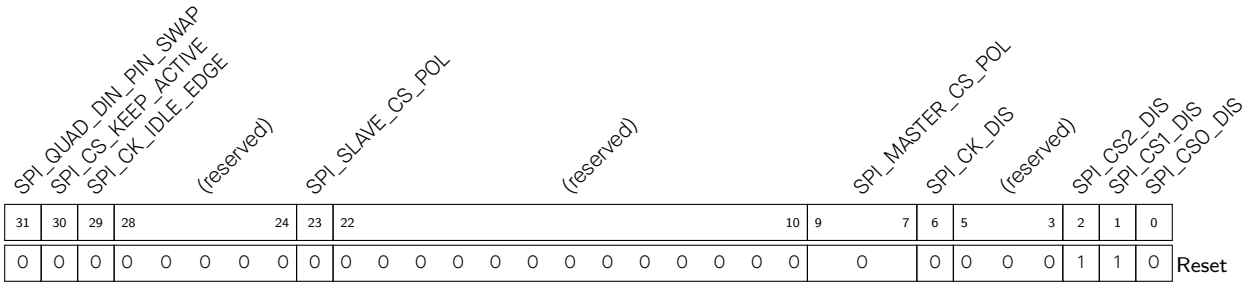
接上页

SPI_CK_IDLE_EDGE 1：SPI CLK 线在空闲状态时保持高电平；0：SPI CLK 线在空闲状态时保持低电平。可在 CONF 阶段配置。(R/W)

SPI_CS_KEEP_ACTIVE 置位此位，则 SPI CS 线保持低电平。可在 CONF 阶段配置。(R/W)

SPI_QUAD_DIN_PIN_SWAP 使能 SPI Quad 输入交换。1：FSPID 与 FSPIQ 交换，FSPIWP 与 FSPIHD 交换。0：禁止 SPI Quad 输入交换。可在 CONF 阶段配置。(R/W)

Register 30.9. SPI_MISC_REG（仅对 SPI3 有效） (0x0020)



SPI_CS0_DIS SPI CS0 管脚使能。1：禁用 CS0；0：SPI CS0 信号来自 CS0 管脚或输出至 CS0 管脚。可在 CONF 阶段配置。(R/W)

SPI_CS1_DIS SPI CS1 管脚使能。1：禁用 CS1；0：SPI CS1 信号来自 CS1 管脚或输出至 CS1 管脚。可在 CONF 阶段配置。(R/W)

SPI_CS2_DIS SPI CS2 管脚使能。1：禁用 CS2；0：1：禁用 CS2；0：SPI CS2 信号来自 CS2 管脚或输出至 CS2 管脚。可在 CONF 阶段配置。(R/W)

SPI_CK_DIS 1：停止 SPI_CLK 输出信号；0：使能 SPI_CLK 输出信号。可在 CONF 阶段配置。(R/W)

SPI_MASTER_CS_POL 主机模式下，SPI_MASTER_CS_POL[*i*] 用于配置 SPI CS*i* 的极性，*i* = 0 ~ 2。0：CS*i* 低电平有效。1：CS*i* 高电平有效。可在 CONF 阶段配置。(R/W)

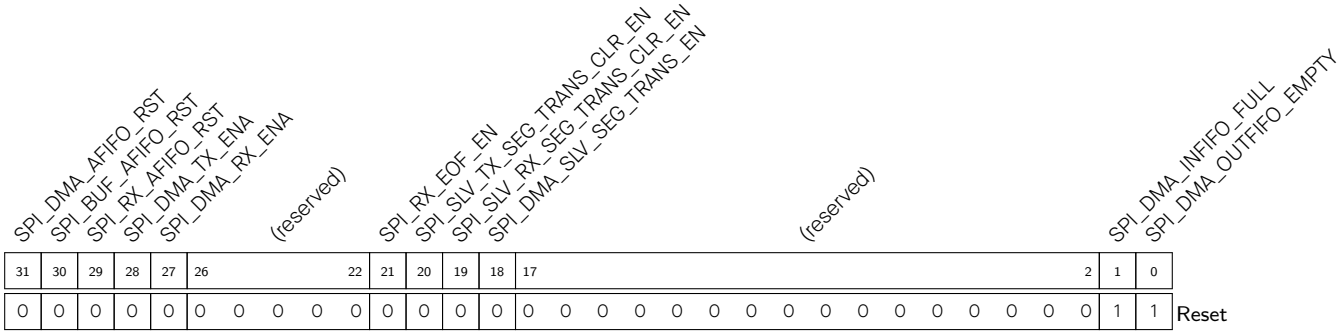
SPI_SLAVE_CS_POL 选择 SPI 从机输入信号 CS 的极性。1：反相；0：保持不变。可在 CONF 阶段配置。(R/W)

SPI_CK_IDLE_EDGE 1：SPI CLK 线在空闲状态时保持高电平；0：SPI CLK 线在空闲状态时保持低电平。可在 CONF 阶段配置。(R/W)

SPI_CS_KEEP_ACTIVE 置位此位，则 SPI CS 线保持低电平。可在 CONF 阶段配置。(R/W)

SPI_QUAD_DIN_PIN_SWAP 1：使能 SPI Quad 输入交换；0：禁止 SPI Quad 输入交换。可在 CONF 阶段配置。(R/W)

Register 30.10. SPI_DMA_CONF_REG (0x0030)



SPI_DMA_OUTFIFO_EMPTY 记录 DMA TX FIFO 的状态。1：DMA TX FIFO 尚未就绪，不能发送数据。0：DMA TX FIFO 已就绪，可以发送数据。(RO)

SPI_DMA_INFIFO_FULL 记录 DMA RX FIFO 的状态。1：DMA RX FIFO 尚未就绪，不能接收数据。0：DMA RX FIFO 已就绪，可以接收数据。(RO)

SPI_DMA_SLV_SEG_TRANS_EN 1：使能从机半双工通信方式下，DMA 控制的连续传输模式。0：禁用。(R/W)

SPI_SLV_RX_SEG_TRANS_CLR_EN 在 DMA 控制的半双工从机模式下，当 DMA RX buffer 小于实际接收的数据长度时，1：后续传输的数据都不接收；0：本次传输的数据不接收，下次传输时，如果 DMA RX buffer 长度不为零，则继续接收，否则不接收。(R/W)

SPI_SLV_TX_SEG_TRANS_CLR_EN 在 DMA 控制的半双工从机模式下，当 DMA TX buffer 大小小于实际发送的数据长度时，1：后续传输的数据都不更新，发送同一个旧数据；0：本次传输的数据都不更新，下次传输时，如果 DMA TX FIFO 填充了新的数据，则继续发送新数据，否则发送数据不更新。(R/W)

SPI_RX_EOF_EN 1：在 DMA 控制的数据传输过程中，如果 DMA 传输的数据比特数等于 (SPI_MS_DATA_BITLEN + 1)，则硬件会置位 GDMA_IN_SUC_EOF_CH*n*_INT_RAW。0：在非分段配置传输模式下，GDMA_IN_SUC_EOF_CH*n*_INT_RAW 由 SPI_TRANS_DONE_INT 置位；或在分段配置传输模式下，由 SPI_DMA_SEG_TRANS_DONE_INT 置位。(R/W)

SPI_DMA_RX_ENA 置位此位，使能 DMA 控制的接收数据模式。(R/W)

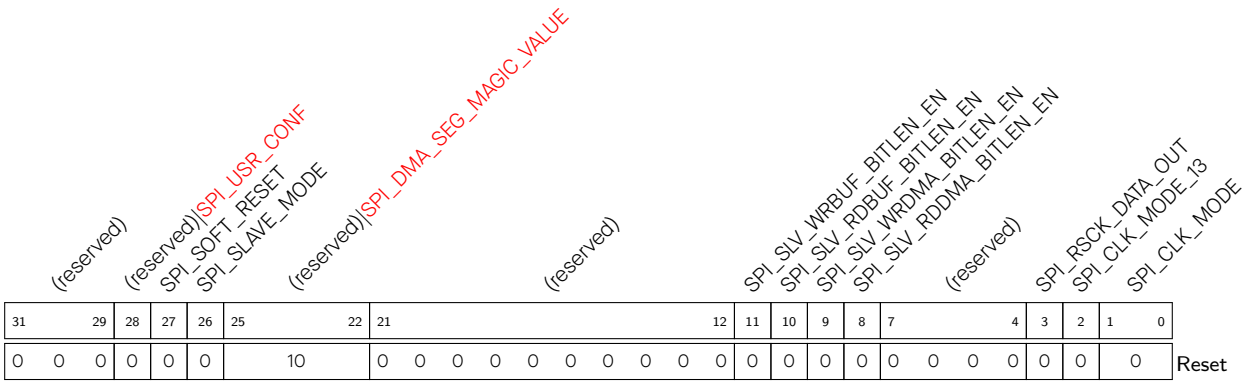
SPI_DMA_TX_ENA 置位此位，使能 DMA 控制的发送数据模式。(R/W)

SPI_RX_AFIFO_RST 置位此位，复位图 30.5-3 和图 30.5-4 中的 spi_rx_afifo。spi_rx_afifo 将在 SPI 主机和从机传输中用于接收数据。(WT)

SPI_BUF_AFIFO_RST 置位此位，复位图 30.5-3 和图 30.5-4 中的 buf_tx_afifo。buf_tx_afifo 将在 CPU 控制的从机传输或主机传输中用于发送数据。(WT)

SPI_DMA_AFIFO_RST 置位此位，复位图 30.5-4 中的 dma_tx_afifo，dma_tx_afifo 在 DMA 控制的从机传输中用于发送数据。(WT)

Register 30.11. SPI_SLAVE_REG (0x00E0)



SPI_CLK_MODE SPI 时钟模式控制位。可在 CONF 阶段配置。(R/W)

- 0：CS 信号无效时，SPI 时钟关闭；
- 1：CS 信号无效后，SPI 时钟延迟一个时钟周期；
- 2：CS 信号无效后，SPI 时钟延迟两个时钟周期；
- 3：SPI 时钟一直有效。

SPI_CLK_MODE_13 配置时钟模式。(R/W)

- 1：使用 SPI 时钟模式 1 或 3，在第一个跳变沿输出 B[0]/B[7] 数据；
- 0：使用 SPI 时钟模式 0 或 2，在第一个跳变沿输出 B[1]/B[6] 数据。

SPI_RSCK_DATA_OUT TSCK 与 RSCK 相同时，将节省半个周期。1：在 RSCK 上升沿输出数据。0：在 TSCK 上升沿输出数据。(R/W)

SPI_SLV_RDDMA_BITLEN_EN 置位此位，则在 DMA 控制的 Rd_DMA 传输过程中，SPI_SLV_DATA_BITLEN 将用于存储 Rd_DMA 传输的数据位长度。(R/W)

SPI_SLV_WRDMA_BITLEN_EN 置位此位，则在 DMA 控制的 Wr_DMA 传输过程中，SPI_SLV_DATA_BITLEN 将用于存储 Wr_DMA 传输的数据位长度。(R/W)

SPI_SLV_RDBUF_BITLEN_EN 置位此位，则在 CPU 控制的 Rd_BUF 传输过程中，SPI_SLV_DATA_BITLEN 将用于存储 Rd_BUF 传输的数据位长度。(R/W)

SPI_SLV_WRBUF_BITLEN_EN 置位此位，则在 CPU 控制的 Wr_BUF 传输过程中，SPI_SLV_DATA_BITLEN 将用于存储 Wr_BUF 传输的数据位长度。(R/W)

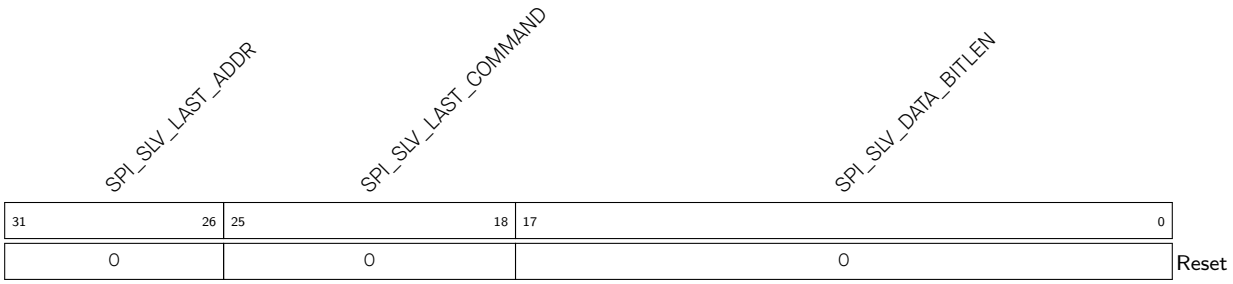
SPI_DMA_SEG_MAGIC_VALUE（仅对 SPI2 有效） 在主机分段配置传输中，用于配置位图表的 Magic 值。(R/W)

SPI_SLAVE_MODE 配置 SPI 工作模式。1：从机模式；0：主机模式。(R/W)

SPI_SOFT_RESET 软件置位使能位。置位此位，可复位 SPI 时钟线、CS 线和数据线。可在 CONF 阶段配置。(WT)

SPI_USR_CONF（仅对 SPI2 有效） 1：置位此位，使能当前分段配置传输的 CONF 阶段，开始分段配置传输。0：清除此位，则表明当前传输不是分段配置传输。(R/W)

Register 30.12. SPI_SLAVE1_REG (0x00E4)

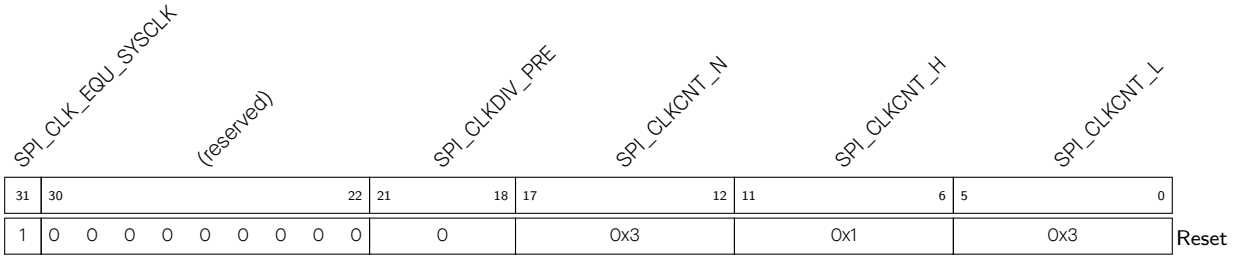


SPI_SLV_DATA_BITLEN 在 SPI 从机全双工和半双工传输中，传输的数据位长。(R/W/SS)

SPI_SLV_LAST_COMMAND 从机模式下的命令值。(R/W/SS)

SPI_SLV_LAST_ADDR 从机模式下的地址值。(R/W/SS)

Register 30.13. SPI_CLOCK_REG (0x000C)



SPI_CLKCNT_L 主机模式下，必须与 SPI_CLKCNT_N 相等。在从机模式下，必须为 0。可在 CONF 阶段配置。(R/W)

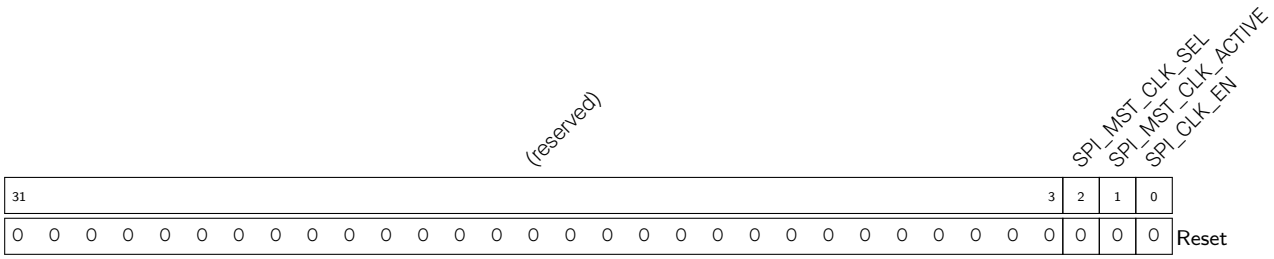
SPI_CLKCNT_H 主机模式下，必须为 (SPI_CLKCNT_N + 1)/2 - 1 的向下取整值。在从机模式下，必须为 0。可在 CONF 阶段配置。(R/W)

SPI_CLKCNT_N 主机模式下，SPI_CLK 的分频系数。因此 SPI_CLK 频率为 fapb_clk/(SPI_CLKDIV_PRE + 1)/(SPI_CLKCNT_N + 1)。可在 CONF 阶段配置。(R/W)

SPI_CLKDIV_PRE 主机模式下，SPI_CLK 的预分频系数。可在 CONF 阶段配置。(R/W)

SPI_CLK_EQU_SYSCLK 在主机模式下，1: SPI_CLK 与 APB_CLK 频率相同；0: SPI_CLK 为 APB_CLK 的分频时钟。可在 CONF 阶段配置。(R/W)

Register 30.14. SPI_CLK_GATE_REG (0x00E8)

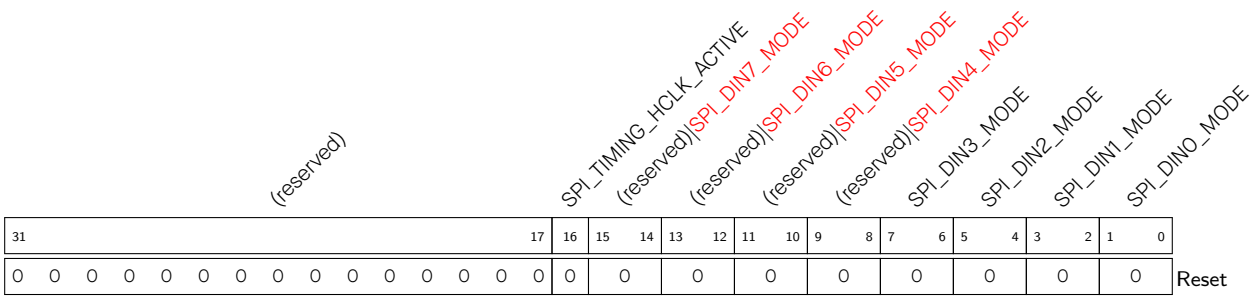


SPI_CLK_EN 置位此位，使能时钟门控。(R/W)

SPI_MST_CLK_ACTIVE 置位此位，SPI 模块时钟上电。(R/W)

SPI_MST_CLK_SEL 该位用于选择主机模式下 SPI 模块的时钟源。1：选择 PLL_F80M_CLK；0：选择 XTAL_CLK。(R/W)

Register 30.15. SPI_DIN_MODE_REG (0x0024)



SPI_DIN0_MODE 配置输入数据 bit0 的输入模式。可在 CONF 阶段配置。(R/W)

- 0：无输入延迟
- 1：输入数据在 SPI_CLK 下降沿延迟 (SPI_DIN0_NUM +1) 个周期
- 2：输入数据在 SPI_CLK 上升沿延迟 (SPI_DIN0_NUM + 1) 个周期
- 3：输入数据在 clk_hclk 上升沿延迟 (SPI_DIN0_NUM + 1) 个周期，然后再在 SPI_CLK 下降沿延迟一个时钟周期

SPI_DIN1_MODE 配置输入数据 bit1 的输入模式。可在 CONF 阶段配置。(R/W)

- 0：无输入延迟
- 1：输入数据在 SPI_CLK 下降沿延迟 (SPI_DIN1_NUM +1) 个周期
- 2：输入数据在 SPI_CLK 上升沿延迟 (SPI_DIN1_NUM + 1) 个周期
- 3：输入数据在 clk_hclk 上升沿延迟 (SPI_DIN1_NUM + 1) 个周期，然后再在 SPI_CLK 下降沿延迟一个时钟周期

SPI_DIN2_MODE 配置输入数据 bit2 的输入模式。可在 CONF 阶段配置。(R/W)

- 0：无输入延迟
- 1：输入数据在 SPI_CLK 下降沿延迟 (SPI_DIN2_NUM +1) 个周期
- 2：输入数据在 SPI_CLK 上升沿延迟 (SPI_DIN2_NUM + 1) 个周期
- 3：输入数据在 clk_hclk 上升沿延迟 (SPI_DIN2_NUM + 1) 个周期，然后再在 SPI_CLK 下降沿延迟一个时钟周期

SPI_DIN3_MODE 配置输入数据 bit3 的输入模式。可在 CONF 阶段配置。(R/W)

- 0：无输入延迟
- 1：输入数据在 SPI_CLK 下降沿延迟 (SPI_DIN3_NUM +1) 个周期
- 2：输入数据在 SPI_CLK 上升沿延迟 (SPI_DIN3_NUM + 1) 个周期
- 3：输入数据在 clk_hclk 上升沿延迟 (SPI_DIN3_NUM + 1) 个周期，然后再在 SPI_CLK 下降沿延迟一个时钟周期

见下页

Register 30.15. SPI_DIN_MODE_REG (0x0024)

接上页

SPI_DIN4_MODE（仅对 SPI2 有效） 配置输入数据 bit4 的输入模式。可在 CONF 阶段配置。(R/W)

- 0：无输入延迟
- 1：输入数据在 SPI_CLK 下降沿延迟 (SPI_DIN4_NUM +1) 个周期
- 2：输入数据在 SPI_CLK 上升沿延迟 (SPI_DIN4_NUM + 1) 个周期
- 3：输入数据在 clk_hclk 上升沿延迟 (SPI_DIN4_NUM + 1) 个周期，然后再在 SPI_CLK 下降沿延迟一个时钟周期

SPI_DIN5_MODE（仅对 SPI2 有效） 配置输入数据 bit5 的输入模式。可在 CONF 阶段配置。(R/W)

- 0：无输入延迟
- 1：输入数据在 SPI_CLK 下降沿延迟 (SPI_DIN5_NUM +1) 个周期
- 2：输入数据在 SPI_CLK 上升沿延迟 (SPI_DIN5_NUM + 1) 个周期
- 3：输入数据在 clk_hclk 上升沿延迟 (SPI_DIN5_NUM + 1) 个周期，然后再在 SPI_CLK 下降沿延迟一个时钟周期

SPI_DIN6_MODE（仅对 SPI2 有效） 配置输入数据 bit6 的输入模式。可在 CONF 阶段配置。(R/W)

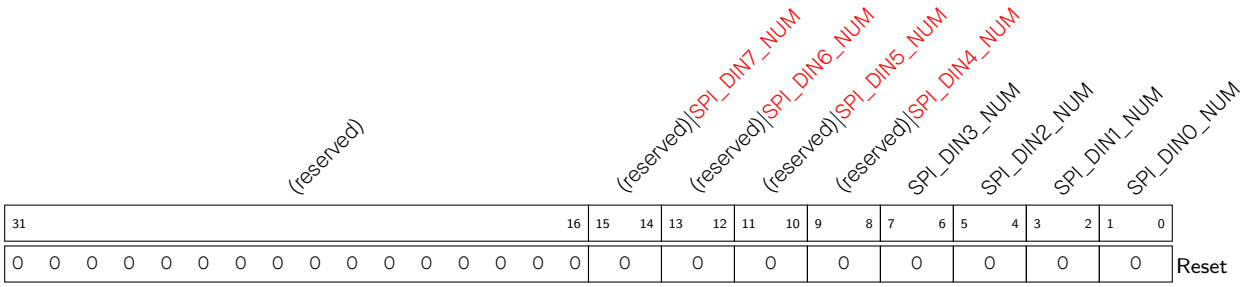
- 0：无输入延迟
- 1：输入数据在 SPI_CLK 下降沿延迟 (SPI_DIN6_NUM +1) 个周期
- 2：输入数据在 SPI_CLK 上升沿延迟 (SPI_DIN6_NUM + 1) 个周期
- 3：输入数据在 clk_hclk 上升沿延迟 (SPI_DIN6_NUM + 1) 个周期，然后再在 SPI_CLK 下降沿延迟一个时钟周期

SPI_DIN7_MODE（仅对 SPI2 有效） 配置输入数据 bit7 的输入模式。可在 CONF 阶段配置。(R/W)

- 0：无输入延迟
- 1：输入数据在 SPI_CLK 下降沿延迟 (SPI_DIN7_NUM +1) 个周期
- 2：输入数据在 SPI_CLK 上升沿延迟 (SPI_DIN7_NUM + 1) 个周期
- 3：输入数据在 clk_hclk 上升沿延迟 (SPI_DIN7_NUM + 1) 个周期，然后再在 SPI_CLK 下降沿延迟一个时钟周期

SPI_TIMING_HCLK_ACTIVE 1：使能 SPI 输入信号时序模块的高频时钟 clk_hclk；0：禁用 clk_hclk。可在 CONF 阶段配置。(R/W)

Register 30.16. SPI_DIN_NUM_REG (0x0028)



SPI_DIN0_NUM 配置输入信号 bit0 的延迟周期数。具体的延迟模式由 SPI_DIN0_MODE 配置。可在 CONF 阶段配置。(R/W)

SPI_DIN1_NUM 配置输入信号 bit1 的延迟周期数。具体的延迟模式由 SPI_DIN2_MODE 配置。可在 CONF 阶段配置。(R/W)

SPI_DIN2_NUM 配置输入信号 bit2 的延迟周期数。具体的延迟模式由 SPI_DIN0_MODE 配置。可在 CONF 阶段配置。(R/W)

SPI_DIN3_NUM 配置输入信号 bit3 的延迟周期数。具体的延迟模式由 SPI_DIN3_MODE 配置。可在 CONF 阶段配置。(R/W)

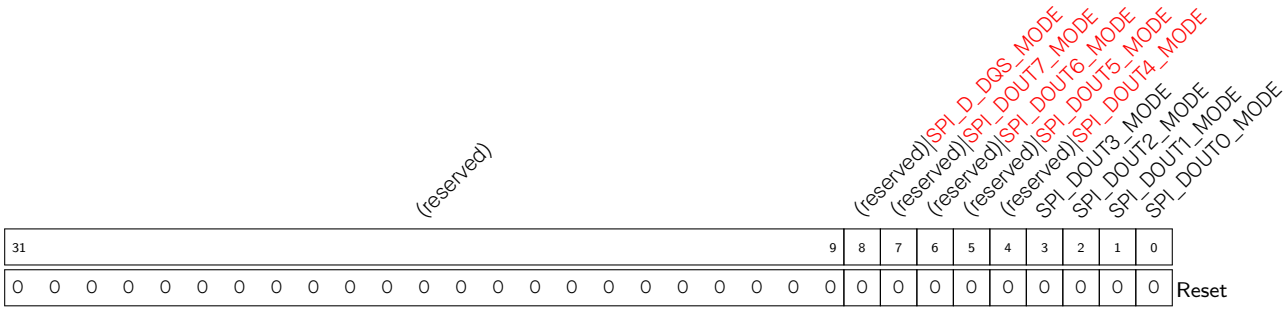
SPI_DIN4_NUM（仅对 SPI2 有效） 配置输入信号 bit4 的延迟周期数。具体的延迟模式由 SPI_DIN4_MODE 配置。可在 CONF 阶段配置。(R/W)

SPI_DIN5_NUM（仅对 SPI2 有效） 配置输入信号 bit5 的延迟周期数。具体的延迟模式由 SPI_DIN5_MODE 配置。可在 CONF 阶段配置。(R/W)

SPI_DIN6_NUM（仅对 SPI2 有效） 配置输入信号 bit6 的延迟周期数。具体的延迟模式由 SPI_DIN6_MODE 配置。可在 CONF 阶段配置。(R/W)

SPI_DIN7_NUM（仅对 SPI2 有效） 配置输入信号 bit7 的延迟周期数。具体的延迟模式由 SPI_DIN7_MODE 配置。可在 CONF 阶段配置。(R/W)

Register 30.17. SPI_DOUT_MODE_REG (0x002C)



SPI_DOUT0_MODE 配置输出数据 bit0 信号的输出模式。可在 CONF 阶段配置。(R/W)

- 0：无输出延迟
- 1：在 SPI_CLK 时钟下降沿，延迟一个时钟周期后输出

SPI_DOUT1_MODE 配置输出数据 bit1 信号的输出模式。可在 CONF 阶段配置。(R/W)

- 0：无输出延迟
- 1：在 SPI_CLK 时钟下降沿，延迟一个时钟周期后输出

SPI_DOUT2_MODE 配置输出数据 bit2 信号的输出模式。可在 CONF 阶段配置。(R/W)

- 0：无输出延迟
- 1：在 SPI_CLK 时钟下降沿，延迟一个时钟周期后输出

SPI_DOUT3_MODE 配置输出数据 bit3 信号的输出模式。可在 CONF 阶段配置。(R/W)

- 0：无输出延迟
- 1：在 SPI_CLK 时钟下降沿，延迟一个时钟周期后输出

SPI_DOUT4_MODE（仅对 SPI2 有效） 配置输出数据 bit4 信号的输出模式。可在 CONF 阶段配置。(R/W)

- 0：无输出延迟
- 1：在 SPI_CLK 时钟下降沿，延迟一个时钟周期后输出

SPI_DOUT5_MODE（仅对 SPI2 有效） 配置输出数据 bit5 信号的输出模式。可在 CONF 阶段配置。(R/W)

- 0：无输出延迟
- 1：在 SPI_CLK 时钟下降沿，延迟一个时钟周期后输出

SPI_DOUT6_MODE（仅对 SPI2 有效） 配置输出数据 bit6 信号的输出模式。可在 CONF 阶段配置。(R/W)

- 0：无输出延迟
- 1：在 SPI_CLK 时钟下降沿，延迟一个时钟周期后输出

见下页

Register 30.17. SPI_DOUT_MODE_REG (0x002C)

接上页

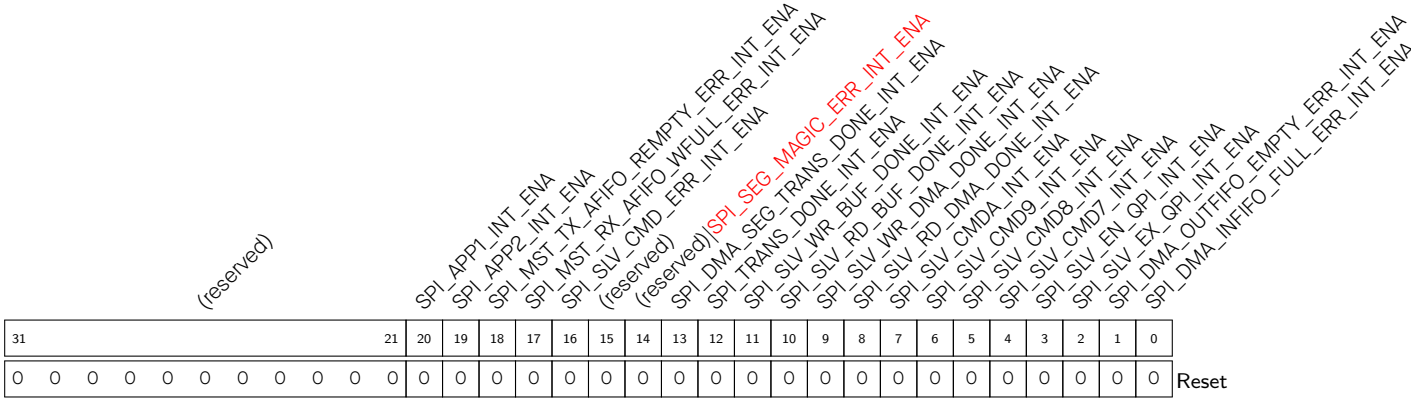
SPI_DOUT7_MODE（仅对 SPI2 有效） 配置输出数据 bit7 信号的输出模式。可在 CONF 阶段配置。(R/W)

- 0：无输出延迟
- 1：在 SPI_CLK 时钟下降沿，延迟一个时钟周期后输出

SPI_D_DQS_MODE（仅对 SPI2 有效） 配置输出信号 SPI_DQS 的输出模式。可在 CONF 阶段配置。(R/W)

- 0：无输出延迟
- 1：在 SPI_CLK 时钟下降沿，延迟一个时钟周期后输出

Register 30.18. SPI_DMA_INT_ENA_REG (0x0034)



SPI_DMA_INFIFO_FULL_ERR_INT_ENA SPI_DMA_INFIFO_FULL_ERR_INT 的中断使能位。(R/W)

SPI_DMA_OUTFIFO_EMPTY_ERR_INT_ENA SPI_DMA_OUTFIFO_EMPTY_ERR_INT 的中断使能位。(R/W)

SPI_SLV_EX_QPI_INT_ENA SPI_SLV_EX_QPI_INT 的中断使能位。(R/W)

SPI_SLV_EN_QPI_INT_ENA SPI_SLV_EN_QPI_INT 的中断使能位。(R/W)

SPI_SLV_CMD7_INT_ENA SPI_SLV_CMD7_INT 的中断使能位。(R/W)

SPI_SLV_CMD8_INT_ENA SPI_SLV_CMD8_INT 的中断使能位。(R/W)

SPI_SLV_CMD9_INT_ENA SPI_SLV_CMD9_INT 的中断使能位。(R/W)

SPI_SLV_CMDA_INT_ENA SPI_SLV_CMDA_INT 的中断使能位。(R/W)

SPI_SLV_RD_DMA_DONE_INT_ENA SPI_SLV_RD_DMA_DONE_INT 的中断使能位。(R/W)

SPI_SLV_WR_DMA_DONE_INT_ENA SPI_SLV_WR_DMA_DONE_INT 的中断使能位。(R/W)

SPI_SLV_RD_BUF_DONE_INT_ENA SPI_SLV_RD_BUF_DONE_INT 的中断使能位。(R/W)

SPI_SLV_WR_BUF_DONE_INT_ENA SPI_SLV_WR_BUF_DONE_INT 的中断使能位。(R/W)

SPI_TRANS_DONE_INT_ENA SPI_TRANS_DONE_INT 的中断使能位。(R/W)

SPI_DMA_SEG_TRANS_DONE_INT_ENA SPI_DMA_SEG_TRANS_DONE_INT 的中断使能位。(R/W)

SPI_SEG_MAGIC_ERR_INT_ENA（仅对 SPI2 有效） SPI_SEG_MAGIC_ERR_INT 的中断使能位。(R/W)

SPI_SLV_CMD_ERR_INT_ENA SPI_SLV_CMD_ERR_INT 的中断使能位。(R/W)

见下页

Register 30.18. SPI_DMA_INT_ENA_REG (0x0034)

接上页

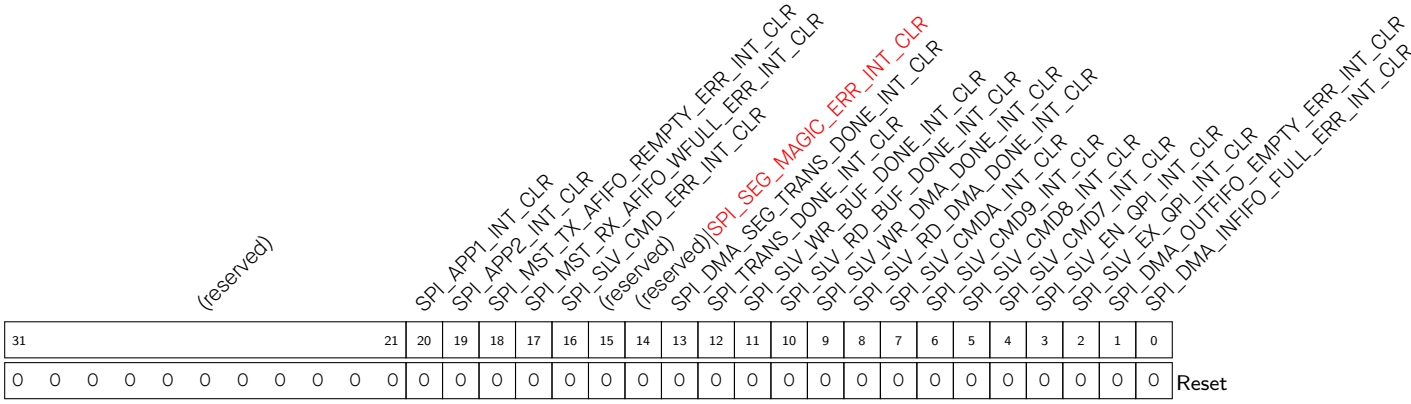
SPI_MST_RX_AFIFO_WFULL_ERR_INT_ENA SPI_MST_RX_AFIFO_WFULL_ERR_INT 的中断使能位。(R/W)

SPI_MST_TX_AFIFO_REMPTY_ERR_INT_ENA SPI_MST_TX_AFIFO_REMPTY_ERR_INT 的中断使能位。(R/W)

SPI_APP2_INT_ENA SPI_APP2_INT 的中断使能位。(R/W)

SPI_APP1_INT_ENA SPI_APP1_INT 的中断使能位。(R/W)

Register 30.19. SPI_DMA_INT_CLR_REG (0x0038)



SPI_DMA_INFIFO_FULL_ERR_INT_CLR SPI_DMA_INFIFO_FULL_ERR_INT 的中断清除位。(WT)

SPI_DMA_OUTFIFO_EMPTY_ERR_INT_CLR SPI_DMA_OUTFIFO_EMPTY_ERR_INT 的中断清除位。(WT)

SPI_SLV_EX_QPI_INT_CLR SPI_SLV_EX_QPI_INT 的中断清除位。(WT)

SPI_SLV_EN_QPI_INT_CLR SPI_SLV_EN_QPI_INT 的中断清除位。(WT)

SPI_SLV_CMD7_INT_CLR SPI_SLV_CMD7_INT 的中断清除位。(WT)

SPI_SLV_CMD8_INT_CLR SPI_SLV_CMD8_INT 的中断清除位。(WT)

SPI_SLV_CMD9_INT_CLR SPI_SLV_CMD9_INT 的中断清除位。(WT)

SPI_SLV_CMDA_INT_CLR SPI_SLV_CMDA_INT 的中断清除位。(WT)

SPI_SLV_RD_DMA_DONE_INT_CLR SPI_SLV_RD_DMA_DONE_INT 的中断清除位。(WT)

SPI_SLV_WR_DMA_DONE_INT_CLR SPI_SLV_WR_DMA_DONE_INT 的中断清除位。(WT)

SPI_SLV_RD_BUF_DONE_INT_CLR SPI_SLV_RD_BUF_DONE_INT 的中断清除位。(WT)

SPI_SLV_WR_BUF_DONE_INT_CLR SPI_SLV_WR_BUF_DONE_INT 的中断清除位。(WT)

SPI_TRANS_DONE_INT_CLR SPI_TRANS_DONE_INT 的中断清除位。(WT)

SPI_DMA_SEG_TRANS_DONE_INT_CLR SPI_DMA_SEG_TRANS_DONE_INT 的中断清除位。(WT)

SPI_SEG_MAGIC_ERR_INT_CLR（仅对 SPI2 有效） SPI_SEG_MAGIC_ERR_INT 的中断清除位。(WT)

SPI_SLV_CMD_ERR_INT_CLR SPI_SLV_CMD_ERR_INT 的中断清除位。(WT)

见下页

Register 30.19. SPI_DMA_INT_CLR_REG (0x0038)

接上页

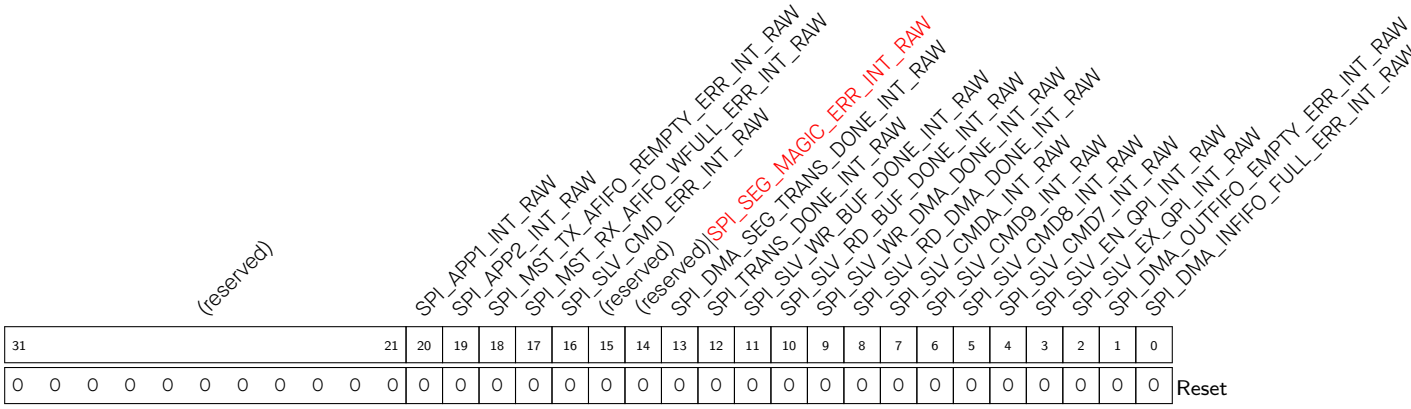
SPI_MST_RX_AFIFO_WFULL_ERR_INT_CLR SPI_MST_RX_AFIFO_WFULL_ERR_INT 的中断清除位。(WT)

SPI_MST_TX_AFIFO_REMPTY_ERR_INT_CLR SPI_MST_TX_AFIFO_REMPTY_ERR_INT 的中断清除位。(WT)

SPI_APP2_INT_CLR SPI_APP2_INT 的中断清除位。(WT)

SPI_APP1_INT_CLR SPI_APP1_INT 的中断清除位。(WT)

Register 30.20. SPI_DMA_INT_RAW_REG (0x003C)



SPI_DMA_INFIFO_FULL_ERR_INT_RAW SPI_DMA_INFIFO_FULL_ERR_INT 的原始中断位。(R/WTC/SS)

SPI_DMA_OUTFIFO_EMPTY_ERR_INT_RAW SPI_DMA_OUTFIFO_EMPTY_ERR_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_EX_QPI_INT_RAW SPI_SLV_EX_QPI_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_EN_QPI_INT_RAW SPI_SLV_EN_QPI_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_CMD7_INT_RAW SPI_SLV_CMD7_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_CMD8_INT_RAW SPI_SLV_CMD8_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_CMD9_INT_RAW SPI_SLV_CMD9_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_CMDA_INT_RAW SPI_SLV_CMDA_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_RD_DMA_DONE_INT_RAW SPI_SLV_RD_DMA_DONE_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_WR_DMA_DONE_INT_RAW SPI_SLV_WR_DMA_DONE_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_RD_BUF_DONE_INT_RAW SPI_SLV_RD_BUF_DONE_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_WR_BUF_DONE_INT_RAW SPI_SLV_WR_BUF_DONE_INT 的原始中断位。(R/WTC/SS)

SPI_TRANS_DONE_INT_RAW SPI_TRANS_DONE_INT 的原始中断位。(R/WTC/SS)

见下页

Register 30.20. SPI_DMA_INT_RAW_REG (0x003C)

接上页

SPI_DMA_SEG_TRANS_DONE_INT_RAW SPI_DMA_SEG_TRANS_DONE_INT 的原始中断位。(R/WTC/SS)

SPI_SEG_MAGIC_ERR_INT_RAW（仅对 SPI2 有效） SPI_SEG_MAGIC_ERR_INT 的原始中断位。(R/WTC/SS)

SPI_SLV_CMD_ERR_INT_RAW SPI_SLV_CMD_ERR_INT 的原始中断位。(R/WTC/SS)

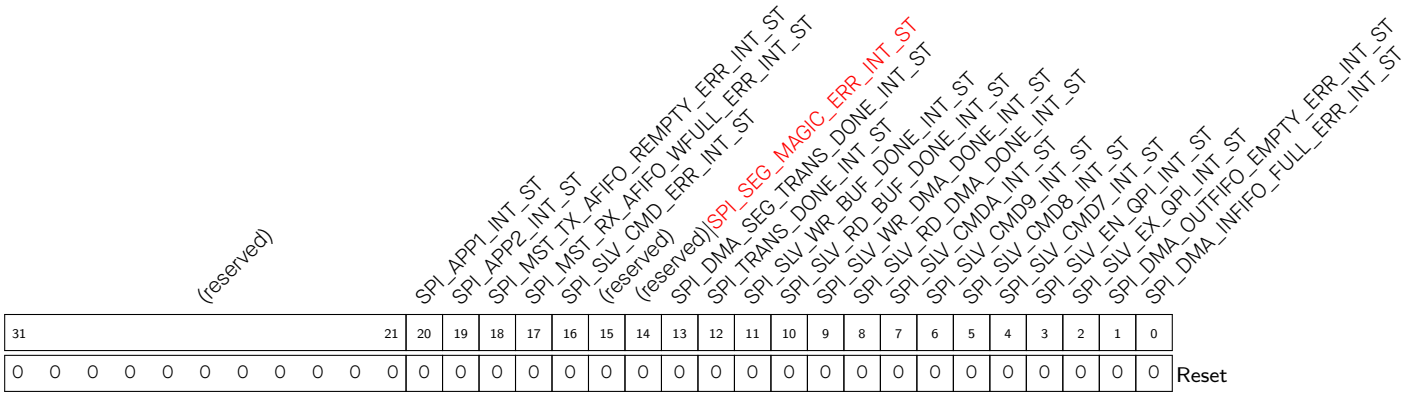
SPI_MST_RX_AFIFO_WFULL_ERR_INT_RAW SPI_MST_RX_AFIFO_WFULL_ERR_INT 的原始中断位。(R/WTC/SS)

SPI_MST_TX_AFIFO_REMPTY_ERR_INT_RAW SPI_MST_TX_AFIFO_REMPTY_ERR_INT 的原始中断位。(R/WTC/SS)

SPI_APP2_INT_RAW SPI_APP2_INT 的原始中断位。该值仅由软件控制。(R/WTC/SS)

SPI_APP1_INT_RAW SPI_APP1_INT 的中断使能位。该值仅由软件控制。(R/WTC/SS)

Register 30.21. SPI_DMA_INT_ST_REG (0x0040)



SPI_DMA_INFIFO_FULL_ERR_INT_ST SPI_DMA_INFIFO_FULL_ERR_INT 的中断状态位。(RO)

SPI_DMA_OUTFIFO_EMPTY_ERR_INT_ST SPI_DMA_OUTFIFO_EMPTY_ERR_INT 的中断状态位。(RO)

SPI_SLV_EX_QPI_INT_ST SPI_SLV_EX_QPI_INT 的中断状态位。(RO)

SPI_SLV_EN_QPI_INT_ST SPI_SLV_EN_QPI_INT 的中断状态位。(RO)

SPI_SLV_CMD7_INT_ST SPI_SLV_CMD7_INT 的中断状态位。(RO)

SPI_SLV_CMD8_INT_ST SPI_SLV_CMD8_INT 的中断状态位。(RO)

SPI_SLV_CMD9_INT_ST SPI_SLV_CMD9_INT 的中断状态位。(RO)

SPI_SLV_CMDA_INT_ST SPI_SLV_CMDA_INT 的中断状态位。(RO)

SPI_SLV_RD_DMA_DONE_INT_ST SPI_SLV_RD_DMA_DONE_INT 的中断状态位。(RO)

SPI_SLV_WR_DMA_DONE_INT_ST SPI_SLV_WR_DMA_DONE_INT 的中断状态位。(RO)

SPI_SLV_RD_BUF_DONE_INT_ST SPI_SLV_RD_BUF_DONE_INT 的中断状态位。(RO)

SPI_SLV_WR_BUF_DONE_INT_ST SPI_SLV_WR_BUF_DONE_INT 的中断状态位。(RO)

SPI_TRANS_DONE_INT_ST SPI_TRANS_DONE_INT 的中断状态位。(RO)

SPI_DMA_SEG_TRANS_DONE_INT_ST SPI_DMA_SEG_TRANS_DONE_INT 的中断状态位。(RO)

SPI_SEG_MAGIC_ERR_INT_ST（仅对 SPI2 有效） SPI_SEG_MAGIC_ERR_INT 的中断状态位。(RO)

SPI_SLV_CMD_ERR_INT_ST SPI_SLV_CMD_ERR_INT 的中断状态位。(RO)

SPI_MST_RX_AFIFO_WFULL_ERR_INT_ST SPI_MST_RX_AFIFO_WFULL_ERR_INT 的中断状态位。(RO)

见下页

Register 30.21. SPI_DMA_INT_ST_REG (0x0040)

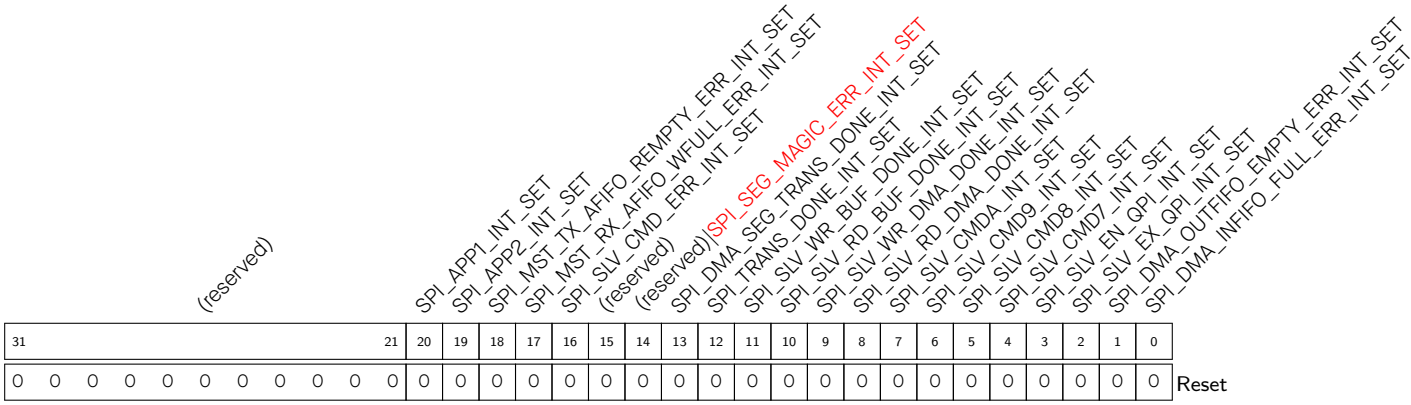
接上页

SPI_MST_TX_AFIFO_REMPTY_ERR_INT_ST SPI_MST_TX_AFIFO_REMPTY_ERR_INT 的中断状态位。(RO)

SPI_APP2_INT_ST SPI_APP2_INT 的中断状态位。(RO)

SPI_APP1_INT_ST SPI_APP1_INT 的中断状态位。(RO)

Register 30.22. SPI_DMA_INT_SET_REG (0x0044)



SPI_DMA_INFIFO_FULL_ERR_INT_SET 软件置位 SPI_DMA_INFIFO_FULL_ERR_INT 中断。(WT)

SPI_DMA_OUTFIFO_EMPTY_ERR_INT_SET 软件置位SPI_DMA_OUTFIFO_EMPTY_ERR_INT 中断。(WT)

SPI_SLV_EX_QPI_INT_SET 软件置位 SPI_SLV_EX_QPI_INT 中断。(WT)

SPI_SLV_EN_QPI_INT_SET 软件置位 SPI_SLV_EN_QPI_INT 中断。(WT)

SPI_SLV_CMD7_INT_SET 软件置位 SPI_SLV_CMD7_INT 中断。(WT)

SPI_SLV_CMD8_INT_SET 软件置位 SPI_SLV_CMD8_INT 中断。(WT)

SPI_SLV_CMD9_INT_SET 软件置位 SPI_SLV_CMD9_INT 中断。(WT)

SPI_SLV_CMDA_INT_SET 软件置位 SPI_SLV_CMDA_INT 中断。(WT)

见下页

Register 30.22. SPI_DMA_INT_SET_REG (0x0044)

接上页

SPI_SLV_RD_DMA_DONE_INT_SET 软件置位 SPI_SLV_RD_DMA_DONE_INT 中断。(WT)

SPI_SLV_WR_DMA_DONE_INT_SET 软件置位 SPI_SLV_WR_DMA_DONE_INT 中断。(WT)

SPI_SLV_RD_BUF_DONE_INT_SET 软件置位 SPI_SLV_RD_BUF_DONE_INT 中断。(WT)

SPI_SLV_WR_BUF_DONE_INT_SET 软件置位 SPI_SLV_WR_BUF_DONE_INT 中断。(WT)

SPI_TRANS_DONE_INT_SET 软件置位 SPI_TRANS_DONE_INT 中断。(WT)

SPI_DMA_SEG_TRANS_DONE_INT_SET 软件置位 SPI_DMA_SEG_TRANS_DONE_INT 中断。(WT)

SPI_SEG_MAGIC_ERR_INT_SET（仅对 SPI2 有效） 软件置位 SPI_SEG_MAGIC_ERR_INT 中断。(WT)

SPI_SLV_CMD_ERR_INT_SET 软件置位 SPI_SLV_CMD_ERR_INT 中断。(WT)

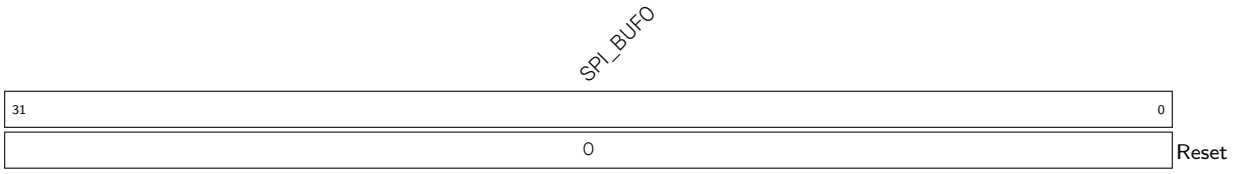
SPI_MST_RX_AFIFO_WFULL_ERR_INT_SET 软件置位 SPI_MST_RX_AFIFO_WFULL_ERR_INT 中断。(WT)

SPI_MST_TX_AFIFO_REMPTY_ERR_INT_SET 软件置位 SPI_MST_TX_AFIFO_REMPTY_ERR_INT 中断。(WT)

SPI_APP2_INT_SET 软件置位 SPI_APP2_INT 中断。(WT)

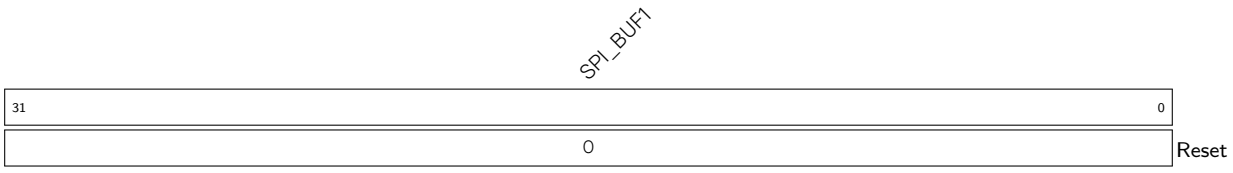
SPI_APP1_INT_SET 软件置位 SPI_APP1_INT 中断。(WT)

Register 30.23. SPI_W0_REG (0x0098)



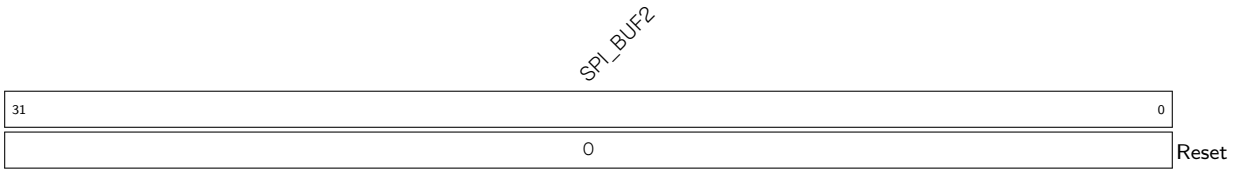
SPI_BUF0 数据 buffer 0，32 位。(R/W/SS)

Register 30.24. SPI_W1_REG (0x009C)



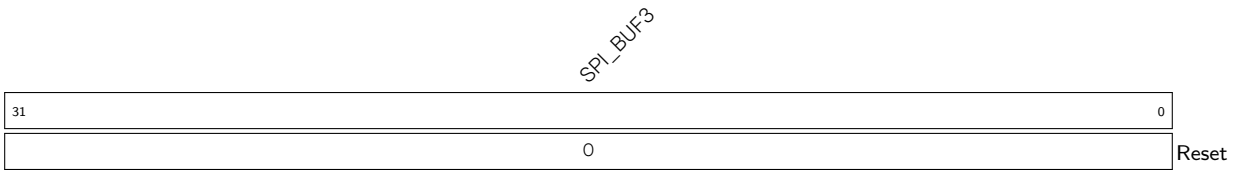
SPI_BUF1 数据 buffer 1，32 位。(R/W/SS)

Register 30.25. SPI_W2_REG (0x00A0)



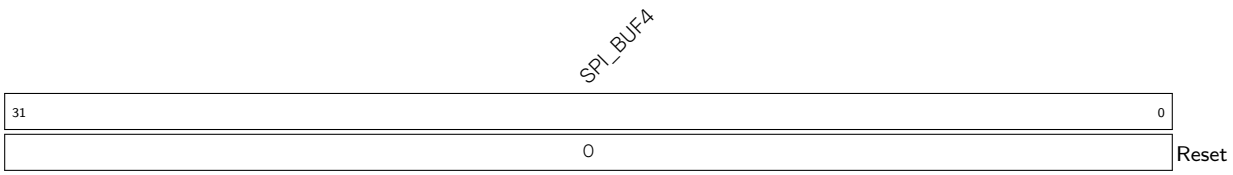
SPI_BUF2 数据 buffer 2，32 位。(R/W/SS)

Register 30.26. SPI_W3_REG (0x00A4)



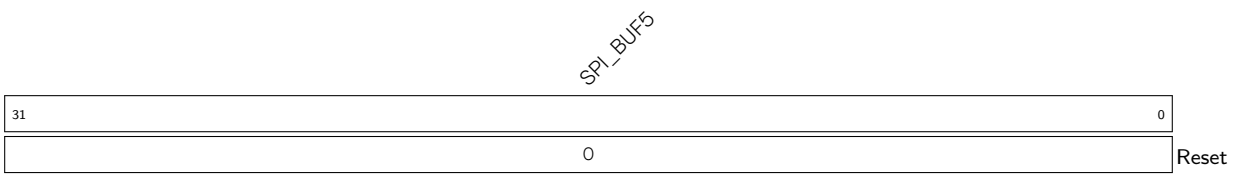
SPI_BUF3 数据 buffer 3，32 位。(R/W/SS)

Register 30.27. SPI_W4_REG (0x00A8)



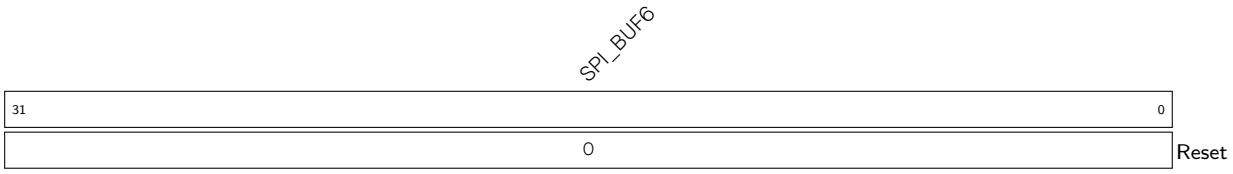
SPI_BUF4 数据 buffer 4，32 位。(R/W/SS)

Register 30.28. SPI_W5_REG (0x00AC)



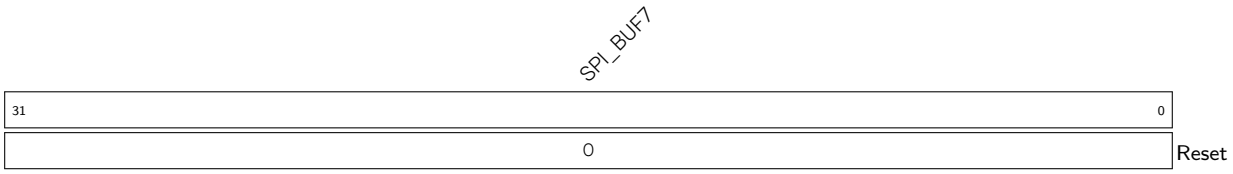
SPI_BUF5 数据 buffer 5，32 位。(R/W/SS)

Register 30.29. SPI_W6_REG (0x00B0)



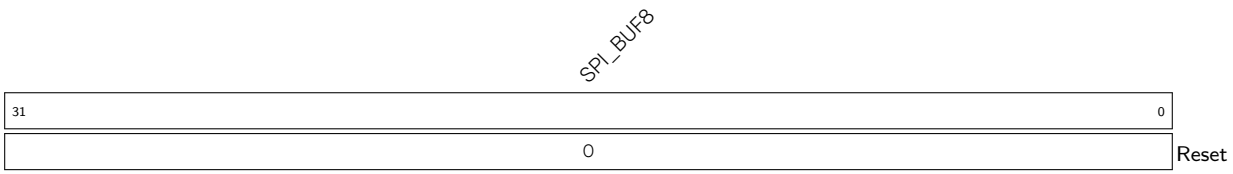
SPI_BUF6 数据 buffer 6，32 位。(R/W/SS)

Register 30.30. SPI_W7_REG (0x00B4)



SPI_BUF7 数据 buffer 7，32 位。(R/W/SS)

Register 30.31. SPI_W8_REG (0x00B8)



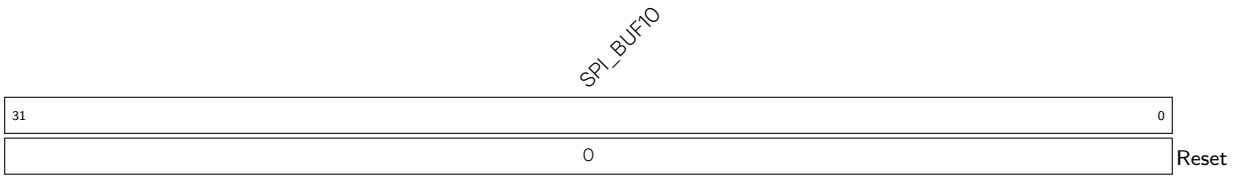
SPI_BUF8 数据 buffer 8，32 位。(R/W/SS)

Register 30.32. SPI_W9_REG (0x00BC)



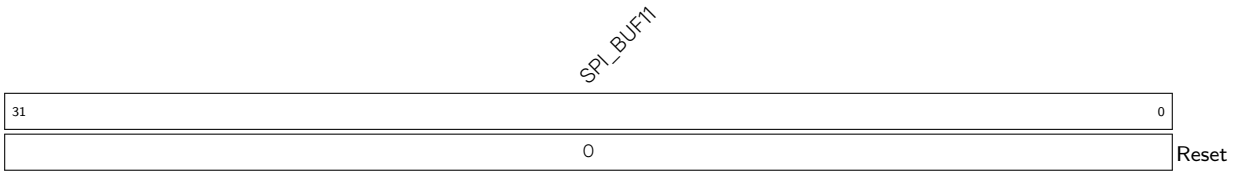
SPI_BUF9 数据 buffer 9，32 位。(R/W/SS)

Register 30.33. SPI_W10_REG (0x00C0)



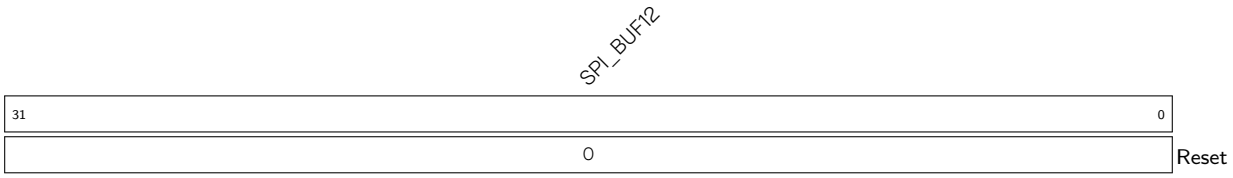
SPI_BUF10 数据 buffer 10，32 位。(R/W/SS)

Register 30.34. SPI_W11_REG (0x00C4)



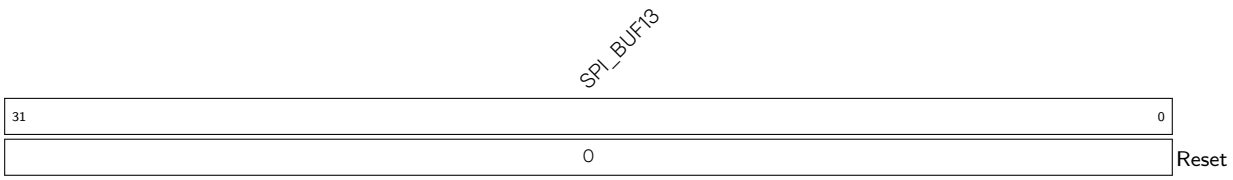
SPI_BUF11 数据 buffer 11，32 位。(R/W/SS)

Register 30.35. SPI_W12_REG (0x00C8)



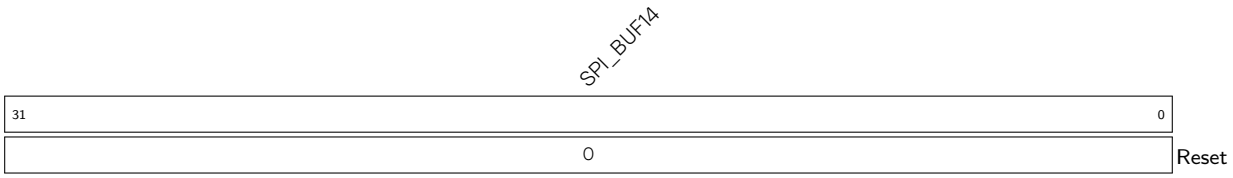
SPI_BUF12 数据 buffer 12，32 位。(R/W/SS)

Register 30.36. SPI_W13_REG (0x00CC)



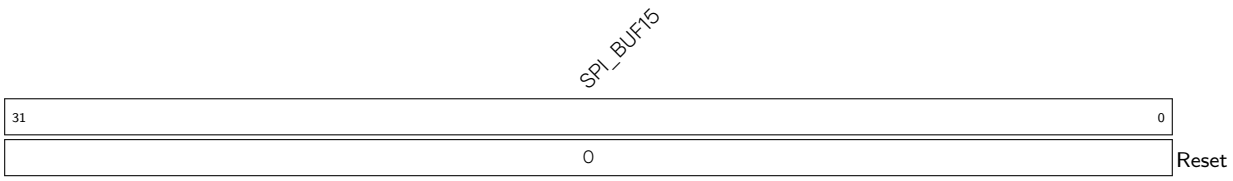
SPI_BUF13 数据 buffer 13，32 位。(R/W/SS)

Register 30.37. SPI_W14_REG (0x00D0)



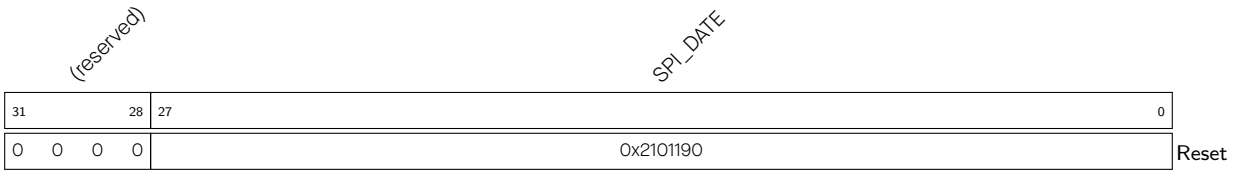
SPI_BUF14 数据 buffer 14，32 位。(R/W/SS)

Register 30.38. SPI_W15_REG (0x00D4)



SPI_BUF15 数据 buffer 15，32 位。(R/W/SS)

Register 30.39. SPI_DATE_REG (0x00F0)



SPI_DATE 版本寄存器。(R/W)

第 31 章

双线汽车接口 (TWAI®)

31.1 概述

双线车载串口 (Two-wire Automotive Interface, TWAI®) 协议是一种多主机、多播的通信协议，具有检测错误、发送错误信号以及内置报文优先仲裁等功能。TWAI 协议适用于汽车和工业应用（可参见第 31.3 章）。

ESP32-S3 包含一个 TWAI 控制器，可通过外部收发器连接到 TWAI 总线。TWAI 控制器包含一系列先进的功能，用途广泛，可用于如汽车产品、工业自动化控制、楼宇自动化等。

31.2 主要特性

ESP32-S3 TWAI 控制器具有以下特性：

- 兼容 ISO 11898-1 协议（CAN 规范 2.0）
- 支持标准格式（11-bit 标识符）和扩展格式（29-bit 标识符）
- 支持 1 Kbit/s ~ 1 Mbit/s 位速率
- 支持多种操作模式
  - 正常模式
  - 只听模式（不影响总线）
  - 自测模式（发送数据时不需应答）
- 64-byte 接收 FIFO
- 特殊发送
  - 单次发送（发生错误时不会自动重新发送）
  - 自发自收（TWAI 控制器同时发送和接收报文）
- 接收滤波器（支持单滤波器和双滤波器模式）
- 错误检测与处理
  - 错误计数
  - 错误报警限制可配置
  - 错误代码捕捉
  - 仲裁丢失捕捉

31.3 功能性协议

31.3.1 TWAI 性能

TWAI 协议连接网络中的两个或多个节点，并允许各节点以延迟限制的形式进行报文交互。TWAI 总线具有以下性能：

单通道通信与不归零编码： TWAI 总线由承载着位的单通道组成，因此为半双工通信。同步调整也在单通道中进行，因此不需其他通道（如时钟通道和使能通道）。TWAI 上报文的位流采用不归零编码 (NRZ) 方式。

位值： 单通道可处于显性状态或隐性状态，显性状态的逻辑值为 0，隐性状态的逻辑值为 1。发送显性状态数据的节点总是比发送隐性状态数据的节点优先级高。总线上的其他物理功能（如，差分电平、单线）由其各自应用实现。

位填充： TWAI 报文的某些域已经过位填充。每发送某个相同值（如显性数值或隐性数值）的连续五个位后，需自动插入一个互补位。同理，接收到 5 个连续位的接收器应将下一个位视为填充位。位填充应用于以下域：SOF、仲裁域、控制域、数据域和 CRC 序列（可参见第 31.3.2 章）。

多播： 当各节点连接到同个总线上时，这些节点将接收相同的位。各节点上的数据将保持一致，除非发生总线错误（可参见第 31.3.3 章）。

多主机： 任意节点都可发起数据传输。如果当前已有正在进行的数据传输，则节点将等待当前传输结束后再发起其数据传输。

报文优先级与仲裁： 若两个或多个节点同时发起数据传输，则 TWAI 协议将确保其中一个节点获得总线的优先仲裁权。各节点所发送报文的仲裁域决定哪个节点可以获得优先仲裁。

错误检测与通报： 各节点将积极检测总线上的错误，并通过发送错误帧来通报检测到的错误。

故障限制： 若一组错误计数依据规定增加/减少时，各节点将维护该组错误计数。当错误计数超过一定阈值时，对应节点将自动关闭以退出网络。

可配置位速率： 单个 TWAI 总线的位速率是可配置的。但是，同个总线中的所有节点须以相同位速率工作。

发送器与接收器： 不论何时，任意 TWAI 节点都可作为发送器和接收器。

- 产生报文的节点为发送器。且该节点将一直作为发送器，直到总线空闲或该节点失去仲裁。请注意，未丢失仲裁的多个节点都可作为发送器。
- 所有非发送器的节点都将作为接收器。

31.3.2 TWAI 报文

TWAI 节点使用报文发送数据，并在监测到总线上存在错误时向其他节点发送错误信号。报文分为不同的帧类型，某些帧类型将具有不同的帧格式。

TWAI 协议有以下帧类型：

- 数据帧
- 远程帧
- 错误帧
- 过载帧

- 帧间距

TWAI 协议有以下帧格式：

- 标准格式 (SFF) 由 11-bit 标识符组成
- 扩展格式 (EFF) 由 29-bit 标识符组成

31.3.2.1 数据帧和远程帧

节点使用数据帧向其他节点发送数据，可负载 0 ~ 8 字节数据。节点使用远程帧向其他节点请求具有相同标识符的数据帧，因此远程帧中不包含任何数据字节。但是，数据帧和远程帧中包含许多相同域。下图 31.3-1 所示为不同帧类型和不同帧格式中包含的域和子域。

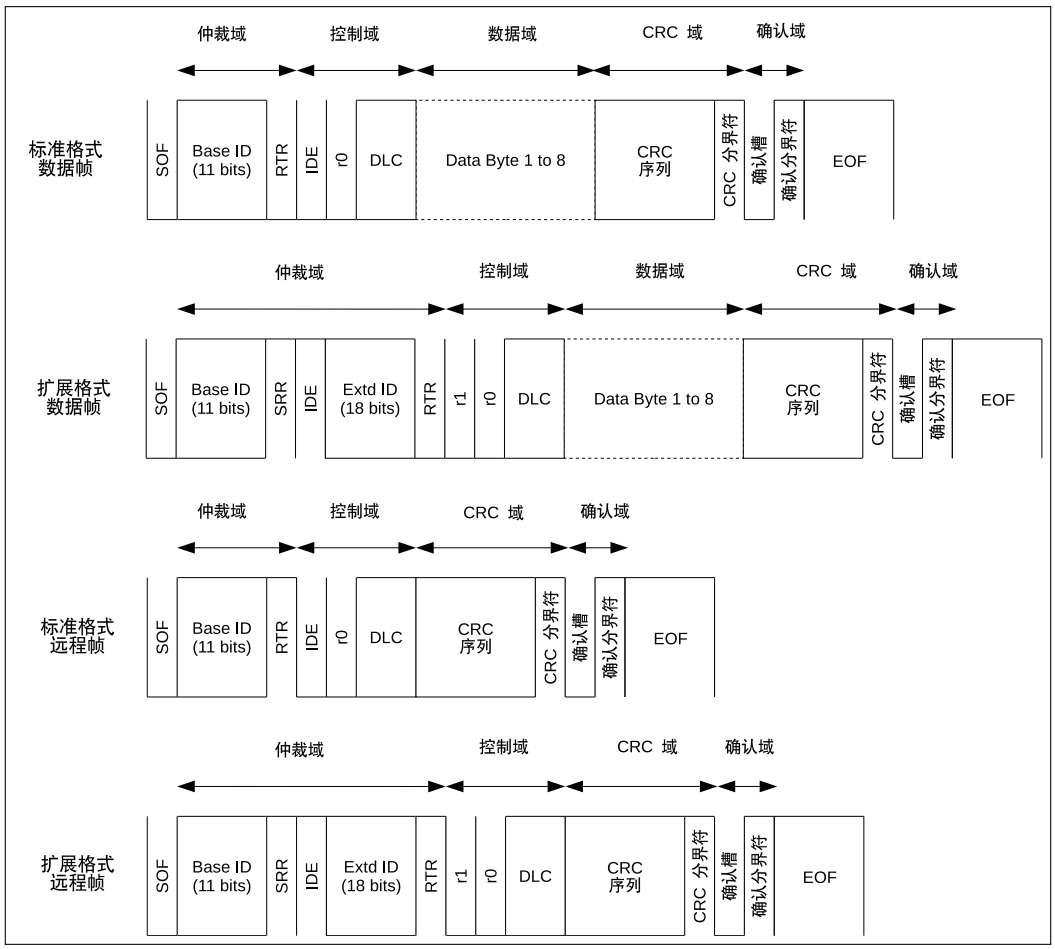


图 31.3-1. 数据帧和远程帧中的位域

仲裁域

当两个或多个节点同时发送数据帧和远程帧时，将根据仲裁域的位信息来决定总线上获得优先仲裁的节点。在仲裁域作用时，如果一个节点在发送隐性位的同时检测到了一个显性位，这表示有其他节点优先于了这个隐性位。那么，这个发送隐性位的节点已丢失总线仲裁，应立即转为接收器。

仲裁域主要由优先发送的最高有效位的帧标识符组成。根据显性位代表的逻辑值为 0，隐性位代表的逻辑值为 1，有以下规律:

- ID 值最小的帧将总是获得仲裁。
- 如果 ID 和格式相同，由于数据帧的 RTR 位为显性位，数据帧将优先于远程帧。

- 如果 ID 的前 11 位相同，由于扩展帧的 SRR 位是隐性，因而标准格式帧将总优先于扩展格式帧。

控制域

控制域主要由数据长度代码 (DLC) 组成，DLC 表示一个数据帧中的负载的数据字节数量，或一个远程帧请求的数据字节数量。DLC 优先发送最高有效位。

数据域

数据域中包含一个数据帧真实负载的数据字节。远程帧中不包含数据域。

CRC 域

CRC 域主要由 CRC 序列组成。CRC 序列是一个 15-bit 的循环冗余校验编码，根据数据帧或远程帧中的未填充内容（从 SOF 到数据域末尾的所有内容）中计算而来。

确认域

确认 (ACK) 域由确认槽和确认分界符组成，主要功能为：接收器向发送器报告已正确接收到有效报文。

表 31.3-1. SFF 和 EFF 中的数据帧和远程帧

数据/远程帧	描述
SOF	帧起始 (SOF) 是一个用于同步总线上节点的单个显性位。
Base ID	基标识符 (ID.28 ~ ID.18) 是 SFF 的 11-bit 标识符，或者是 EFF 中 29-bit 标识符的前 11-bit。
RTR	远程发送请求位 (RTR) 显示当前报文是数据帧（显性）还是远程帧（隐性）。这意味着，当某个数据帧和一个远程帧有相同标识符时，数据帧始终优先于远程帧仲裁。
SRR	在 EFF 中发送替代远程请求位 (SRR)，以替代 SFF 中相同位置的 RTR 位。
IDE	标识符扩展位 (IED) 显示当前报文是 SFF（显性）还是 EFF（隐性）。这意味着，当某 SFF 帧和 EFF 帧有相同基标识符时，SFF 帧将始终优先于 EFF 帧仲裁。
Extd ID	扩展标识符 (ID.17 ~ ID.0) 是 EFF 中 29-bit 标识符的剩余 18-bit。
r1	r1（保留位 1）始终是显性位。
r0	r0（保留位 0）始终是显性位。
DLC	数据长度代码 (DLC) 为 4-bits，且应包含 0 ~ 8 中任一数值。数据帧使用 DLC 表示自身包含的数据字节数量。远程帧使用 DLC 表示从其他节点请求的数据字节数量。
数据字节	表示数据帧的数据负载量。该字节数量应与 DLC 的值匹配。首先发送数据字节 0，各数据字节优先发送最高有效位。
CRC 序列	CRC 序列是一个 15-bit 的循环冗余校验编码。
CRC 分界符	CRC 分界符是遵循 CRC 序列的单一隐性位。
确认槽	确认槽用于接收器节点，表示是否已成功接收数据帧或远程帧。发送器节点将在确认槽中发送一个隐性位，如果接收到的帧没有错误，则接收器节点应用一个显性位替代确认槽。
确认分界符	确认分界符是一个单一的隐性位。
EOF	帧结束 (EOF) 标志着数据帧或远程帧的结束，由七个隐性位组成。

31.3.2.2 错误帧和过载帧

错误帧

当某节点检测到总线错误时，将发送一个错误帧。错误帧由一个特殊的错误标志构成，该标志由某相同值的六个连续位组成，因而违反了位填充的规则。所以，当某节点检测到总线错误并发送错误帧时，其余节点也将相

应地检测到一个填充错误并各自发送错误帧。也就是说，当发生总线错误时，通过上述过程可将该报文传递至总线上的所有节点。

当某节点检测到总线错误时，该节点将于下一个位发送错误帧。特例：如果总线错误类型为 CRC 错误，那么错误帧将从确认分界符的下一个位开始（可参见第 31.3.3 章）。下图 31.3-2 所示为一个错误帧所包含的不同域：

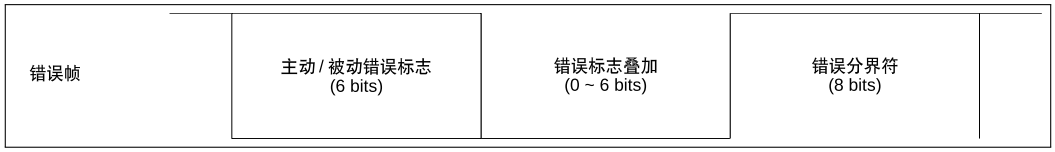


图 31.3-2. 错误帧中的位域

表 31.3-2. 错误帧

错误帧	描述
错误标志	错误标志包括两种形式: 主动错误标志和被动错误标志，主动错误标志由 6 个显性位组成，被动错误标志由 6 个隐性位组成（被其他节点的显性位优先仲裁时除外）。主动错误节点发送主动错误标志，被动错误节点发送被动错误标志。
错误标志叠加	错误标志叠加域的主要目的是允许总线上的其他节点发送各自的主动错误标志。叠加域的范围可以是 0 ~ 6 位，在检测到第一个隐性位时结束（如检测到分界符上的第一个位时）。
错误分界符	分界符域标志着错误/过载帧结束，由 8 个隐性位构成。

过载帧

过载帧与包含主动错误标志的错误帧有着相同的位域。二者主要区别在于触发发送过载帧的条件。下图 31.3-3 所示为过载帧中包含的位域：

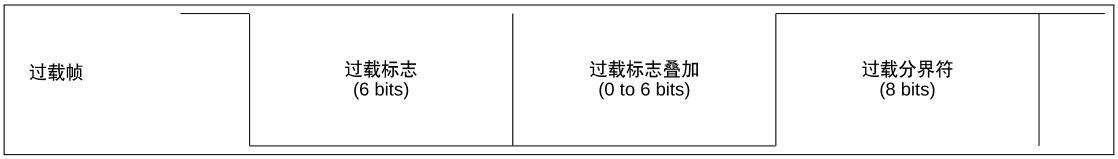


图 31.3-3. 过载帧中的位域

表 31.3-3. 过载帧

过载帧	描述
过载标志	由 6 个显性位构成。与主动错误标志相同。
过载标志叠加	允许其他节点发送过载标志的叠加，与错误标志叠加相似。
过载分界符	由 8 个隐性位构成。与错误分界符相同。

下列情况将触发发送过载帧：

1. 接收器内部要求延迟发送下一个数据帧或远程帧。
2. 在间歇域后的首个和第二个位上检测到显性位。

3. 如果在错误分界符的第八个（最后一个）位上检测到显性位。请注意，在这种情况下 TEC 和 REC 的值将不会增加（可参见第 31.3.3 章）。

由于上述情况发送过载帧时，须满足以下规定：

- 第 1 条情况下发送的过载帧只能从间歇域后的第一个位开始。
- 第 2、3 条情况下发送的过载帧须从检测到显性位的后一个位开始。
- 要延迟发送下一个数据帧或远程帧，最多可生成两个过载帧。

31.3.2.3 帧间距

帧间距充当各帧之间的分隔符。数据帧和远程帧必须与前一帧用一个帧间距分隔开，不论前面的帧是何类型（数据帧、远程帧、错误帧、过载帧）。但是，错误帧和过载帧则无需与前一个帧分隔开。

下图 31.3-4 所示为帧间距中包含的域：

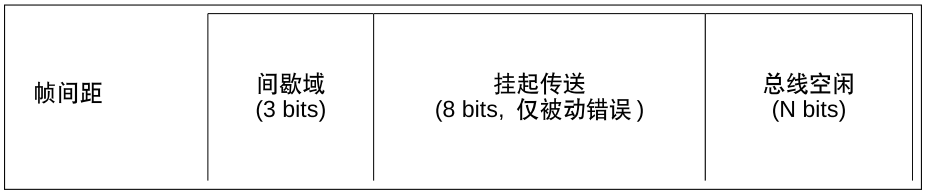


图 31.3-4. 帧间距中的域

表 31.3-4. 帧间距

帧间距	描述
间歇域	间歇域由 3 个隐性位构成。
挂起传送	被动错误节点发送报文后，节点中须包含一个挂起传送域，由 8 个隐性位构成。主动错误节点中不含这个域。
总线空闲	总线空闲域长度任意。发送 SOF 时，总线空闲结束。若节点中有挂起传送，则 SOF 应在间歇域后的第一位发送。

31.3.3 TWAI 错误

31.3.3.1 错误类型

TWAI 中的总线错误包括以下类型：

位错误

当节点发送一个位值（显性位或隐性位）但检测到相反的位时（如，发送显性位时检测到了隐性位），就会发生位错误。但是，如果发送的位是隐性位，且位于仲裁域或确认槽或被动错误标志中，那么此时检测到显性位的话也不会认定为位错误。

填充错误

当检测到相同值的 6 个连续位时（违反位填充的编码规则），发生填充错误。

CRC 错误

数据帧和远程帧的接收器将根据接收到的位计算 CRC 值。当接收器计算的值与接收到的数据帧和远程帧中的 CRC 序列不匹配时，会发生 CRC 错误。

格式错误

当某个报文中的固定格式位中包含非法位时，可检测到格式错误。比如，r1 和 r0 域必须固定为显性。

确认错误

当发送器无法在确认槽中检测到显性位时，将发生确认错误。

31.3.3.2 错误状态

TWAI 通过每个节点维护两个错误计数来实现故障界定，计数数值决定错误状态。这两个错误计数分别为：发送错误计数 (TEC) 和接收错误计数 (REC)。TWAI 包含以下错误状态。

主动错误
主动错误节点可参与到总线交互中，且在检测到错误时可以发送主动错误标志。

被动错误
被动错误节点可参与到总线交互中，但在检测到错误时只能发送一次被动错误标志。被动错误节点发送数据帧或远程帧后，须在后续的帧间距中设置挂起传送域。

离线
禁止离线节点以任意方式干扰总线（如，不允许其进行数据传输）。

31.3.3.3 错误计数

TEC 和 REC 根据以下规则递增/递减。**请注意，一条报文传输中可应用多个规则**。

1. 当接收器检测到错误时，REC 数值将增加 1。当检测到的错误为发送主动错误标志或过载标志期间的位错误除外。
2. 发送错误标志后，当接收器第一个检测到的位是显性位时，REC 数值将增加 8。
3. 当发送器发送错误标志时，TEC 数值增加 8。但是，以下情况不适用于该规则：
   - 发送器为被动错误状态，因为在应答槽未检测到显性位而产生应答错误，且在发送被动错误标志时检测到显性位时，则 TEC 数值不应增加。
   - 发送器在仲裁期间因填充错误而发送错误标志，且填充位本该是隐性位但是检测到显性位，则 TEC 数值不应增加。
4. 若发送器在发送主动错误标志和过载标志时检测到位错误，则 TEC 数值增加 8。
5. 若接收器在发送主动错误标志和过载标志时检测到位错误，则 REC 数值增加 8。
6. 任意节点在发送主动/被动错误标志或过载标志后，节点仅能承载最多 7 个连续显性位。在（发送主动错误标志或过载标志时）检测到第 14 个连续显性位，或在被动错误标志后检测到第 8 个连续显性位后，发送器将使其 TEC 数值增加 8，而接收器将使其 REC 数值增加 8。每增加 8 个连续显性位的同时，（发送器的）TEC 和（接收器的）REC 数值也将增加 8。
7. 每当发送器成功发送报文后（接收到 ACK，且直到 EOF 完成未发生错误），TEC 数值将减小 1，除非 TEC 的数值已经为 0。
8. 当接收器成功接收报文后（确认槽前未检测到错误，且成功发送 ACK），则 REC 数值将相应减小。
   - 若 REC 数值位于 1 ~ 127 之间，则其值减小 1。
   - 若 REC 数值大于 127，则其值减小到 127。
   - 若 REC 数值为 0，则仍保持为 0。
9. 当一个节点的 TEC 和/或 REC 数值大于等于 128 时，该节点变为被动错误节点。导致节点发生上述状态切换的错误，该节点仍发送主动错误标志。请注意，一旦 REC 数值到达 128，后续任何增加该值的动作都是无效的，直到 REC 数值返回到 128 以下。
10. 当某节点的 TEC 数值大于等于 256 时，该节点将变为离线节点。
11. 当某被动错误节点的 TEC 和 REC 数值都小于等于 127，则该节点将变为主动错误节点。
12. 当离线节点在总线上检测到 128 次 11 个连续隐性位后，该节点可变为主动错误节点（TEC 和 REC 数值都重设为 0）。

31.3.4 TWAI 位时序

31.3.4.1 名义位

TWAI 协议允许 TWAI 总线以特定的位速率运行。但是，总线内的所有节点必须以统一位速率运行。

- **名义位速率**为每秒发送比特数量。
- **名义位时间**为 1/**名义位速率**。

每个名义位时间中含多个段，每段由多个时间定额 (Time Quanta) 组成。**时间定额**为最小时间单位，作为一种预分频时钟信号应用于各个节点中。下图 31.3-5 所示为一个名义位时间内所包含的段。

TWAI 控制器将在一个时间定额的时间步长中进行操作，每个时间定额中都会分析 TWAI 的总线状态。如果两个连续的时间定额中总线状态不同（隐性-显性，或反之），意味着有边沿产生。PBS1 和 PBS2 的交点将被视为采样点，且采样的总线数值即为这个位的数值。

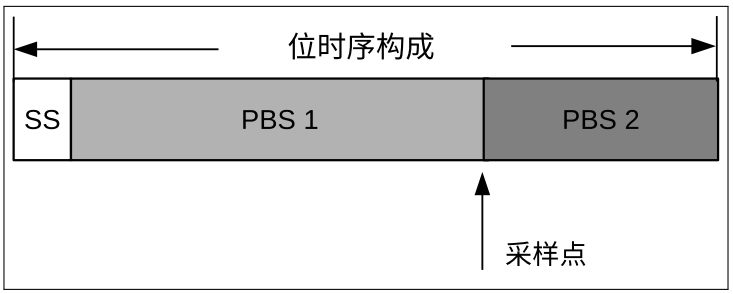


图 31.3-5. 位时序构成

表 31.3-5. 名义位时序中包含的段

段	描述
同步段 (SS)	SS（同步段）的长度为 1 个时间定额。若所有节点都同步正常，则位边沿应位于该段内。
缓冲时期段 1 (PBS1)	PBS1 的长度可为 1 ~ 16 个时间定额，用于补偿网络中的物理延迟时间。可增加 PBS1 的长度，从而更好地实现同步。
缓冲时期段 2 (PBS2)	PBS2 的长度可为 1 ~ 8 个时间定额，用于补偿节点中的信息处理时间。可缩短 PBS2 的长度，从而更好地实现同步。

31.3.4.2 硬同步与再同步

由于时钟偏移和抖动，同一总线上节点的位时序可能会脱离相位段。因而，位边沿可能会偏移到同步段的前后。针对上述位边沿偏移的问题 TWAI 提供多种同步方式。设位边沿偏移的 TQ（时间定额）数量为**相位错误 “e”**，该值与 SS 相关。

- 主动相位错误 (e > 0)：位边沿位于同步段之后采样点之前（即，边沿向后偏移）。
- 被动相位错误 (e < 0)：位边沿位于前个位的采样点之后同步段之前（即，边沿向前偏移）。

为解决相位错误，可进行两种同步方式，即**硬同步**与**再同步**。**硬同步**与**再同步**遵守以下规则：

- 单个位时序中仅可发生一次同步。
- 同步仅可发生在隐性位到显性位的边沿上。

硬同步

总线空闲期间，硬同步发生在隐性位到显性位的变化边沿上（如总线空闲后的第一个 SOF 位）。此时，所有节点都将重启其内部位时序，从而使该变化边沿位于重启位时序的同步段内。

再同步

非总线空闲期间，再同步发生在隐性位到显性位的变化边沿上。如果边沿上有主动相位错误 (e > 0)，则 PBS1 长度将增加。如果边沿上有被动相位错误 (e < 0)，则 PBS2 长度将减小。

PBS1/PBS2 具体增加和减小的时间定额取决于相位错误的绝对值，同时也受可配置的同步跳宽 (SJW) 数值限制。

- 当相位错误的绝对值小于等于 SJW 数值时，PBS1/PBS2 将增加/减小 e 个时间定额。该过程与硬同步具有相同效果。
- 当相位错误的绝对值大于 SJW 数值时，PBS1/PBS2 将增加/减小与 SJW 相同数值的时间定额。这意味着，在完全解决相位错误之前，可能需要多个同步位。

31.4 结构概述

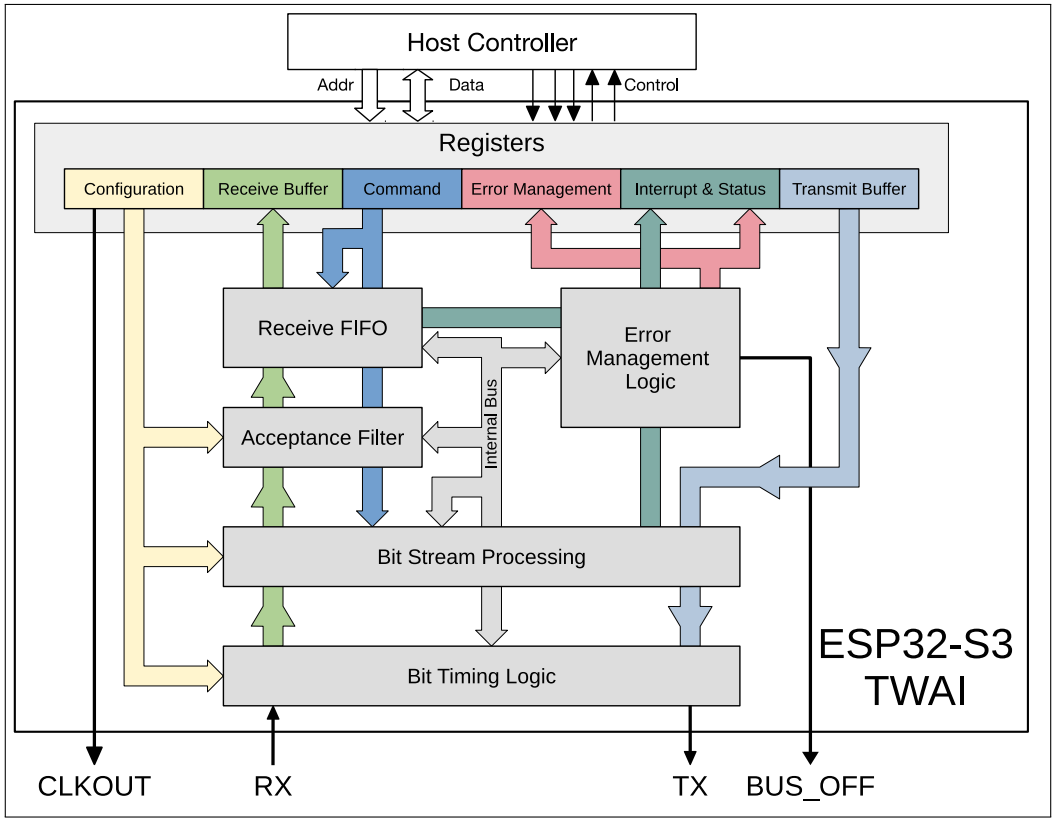


图 31.4-1. TWAI 概略图

TWAI 控制器的主要功能模块如图 31.4-1。

31.4.1 寄存器模块

ESP32-S3 的 CPU 使用 32-bit 对齐字访问外设。但是，TWAI 控制器中的大部分寄存器仅存储最低有效字节 (bits [7:0]) 上的有用数据。因此在这些寄存器中，bits [31:8] 在写入时被忽略，在读取时返回 0。

配置寄存器

配置寄存器存储 TWAI 控制器的各配置项，如位速率、操作模式、接收滤波器等。只有在 TWAI 控制器处于复位模式时，才可修改配置寄存器（可参见第 31.5.1 章）。

指令寄存器

CPU 通过指令寄存器驱动 TWAI 控制器执行任务，如发送报文或清除接收缓冲器。只有在 TWAI 控制器处于操作模式时，才可修改指令寄存器（可参见第 31.5.1 章）。

中断 & 状态寄存器

中断寄存器显示 TWAI 控制器中发生的事件（每个事件由一个单独的位表示）。状态寄存器显示 TWAI 控制器的当前状态。

错误管理寄存器

错误管理寄存器包括错误计数和捕捉寄存器。错误计数寄存器表示 TEC 和 REC 的数值。捕捉寄存器负责记录相关信息，如 TWAI 控制器在何处检测到总线错误，或何时丢失仲裁。

发送缓冲寄存器

发送缓冲器大小为 13 字节，用于存储 TWAI 的待发送报文。

接收缓冲寄存器

接收缓冲器大小为 13 字节，用于存储单个报文。接收缓冲器是进入接收 FIFO 的窗口，接收 FIFO 中的第一个报文将被映射到接收缓冲器中。

请注意，发送缓冲寄存器、接收缓冲寄存器和接收滤波寄存器的地址范围相同（地址偏移包含 0x0040 ~ 0x0070）。这些寄存器的访问权限遵循以下规则：

- 当 TWAI 控制器处于复位模式时，该地址范围被映射到接收滤波寄存器中。
- TWAI 控制器处于操作模式时：
  - 对地址范围的所有读取都映射于接收缓冲寄存器中。
  - 对地址范围的所有写入都映射于发送缓冲寄存器中。

31.4.2 位流处理器

位流处理 (BSP) 模块负责对发送缓冲器的数据进行帧处理 (如，位填充和附加 CRC 域) 并为位时序逻辑 (BTL) 模块生成位流。同时，BSP 模块还负责处理从 BTL 模块中接收的位流（如，去填充和验证 CRC），并将处理报文置于接收 FIFO。BSP 还负责检测 TWAI 总线上的错误并将此类错误报告给错误管理逻辑 (EML)。

31.4.3 错误管理逻辑

错误管理逻辑 (EML) 模块负责更新 TEC 和 REC 数值，记录错误信息（如，错误类型和错误位置），更新控制器的错误状态，确保 BSP 模块发送正确的错误标志。此外，该模块还负责记录 TWAI 控制器丢失仲裁时的 bit 位置。

31.4.4 位时序逻辑

位时序逻辑 (BTL) 模块负责以预先配置的位速率发送和接受报文。BTL 模块还负责同步位时序，确保数据传输的稳定性。位速率由多个可编程的段组成，且用户可设置每个段的 TQ（时间定额）长度，来调整传播延迟、控制器处理时间等因素。

31.4.5 接收滤波器

接收滤波器是一个可编程的报文过滤单元，允许 TWAI 控制器根据报文的标识符域接收或拒绝该报文。通过接收滤波器的报文才能被存储到接收 FIFO 中。用户可配置接收滤波器的模式：单滤波器、双滤波器。

31.4.6 接收 FIFO

接收 FIFO 是大小为 64-byte 的缓冲器（位于 TWAI 控制器内部），负责存储通过接收滤波器的接收报文。接收 FIFO 中存储的报文大小可以不同（3 ~ 13 byte 范围之间）。当接收 FIFO 为满时（或剩余的空间不足以完全存储下一个接收报文），将触发溢出中断，后续的接收报文将丢失，直到接收 FIFO 中清除出足够的存储空间。接收 FIFO 中的第一条报文将被映射到 13-byte 的接收缓冲器中，直到该报文被清除（通过释放接收缓冲器指令）。清除后，接收缓冲器将继续映射接收 FIFO 中的下一条报文，接收 FIFO 中上一条已清除报文的空间将被释放。

31.5 功能描述

31.5.1 模式

ESP32-S3 TWAI 控制器有两种工作模式：复位模式和操作模式。将 TWAI_RESET_MODE 位置 1，进入复位模式；置 0，进入操作模式。

31.5.1.1 复位模式

要修改 TWAI 控制器的各种配置寄存器，需进入复位模式。进入复位模式时，TWAI 控制器彻底与 TWAI 总线断开连接。复位模式下，TWAI 控制器将无法发送任何报文（包括错误信号）。任何正在进行的报文传输将立即被终止。同样的，TWAI 控制器在该模式下也将无法接收任何报文。

31.5.1.2 操作模式

进入操作模式后，TWAI 控制器与总线相连，并且写保护各配置寄存器，以确保控制器的配置在运行期间保持一致。操作模式下，TWAI 控制器可以发送和接收报文（包括错误信号），但具体取决于 TWAI 控制器配置于哪种运行子模式。TWAI 控制器支持以下三种子模式：

- **正常模式：** TWAI 控制器可以发送和接收包含错误信号在内的报文（如，错误帧和过载帧）。
- **自测模式：** 与正常模式相同，但在该模式下，TWAI 控制器发送报文时，即使在 CRC 域之后没有接收到应答信号，也不会产生应答错误。通常在 TWAI 控制器自测时使用该模式。
- **只听模式：** TWAI 控制器可以接收报文，但在 TWAI 总线上保持完全被动。因此，TWAI 控制器将无法发送任何报文、应答或错误信号。错误计数将保持冻结状态。该模式用于 TWAI 总线监控。

请注意，退出复位模式后（如，进入操作模式时），TWAI 控制器需等待 11 个连续隐性位出现，才能完全连接上 TWAI 总线（即，可以发送或接收报文）。

31.5.2 位时序

TWAI 控制器的工作位速率必须在控制器处于复位模式时进行配置。在寄存器 TWAI_BUS_TIMING_0_REG 和 TWAI_BUS_TIMING_1_REG 中配置位速率，这两个寄存器包含以下域：

下表 31.5-1 所示为 TWAI_BUS_TIMING_0_REG 包含的位域。

表 31.5-1. TWAI_BUS_TIMING_0_REG 的 bit 信息 (0x18)

Bit 31-16	Bit 15	Bit 14	Bit 13	Bit 12	.....	Bit 1	Bit 0
保留	SJW.1	SJW.0	保留	BRP.12	.....	BRP.1	BRP.0

说明：

- 预分频值 (BRP)：TWAI 时间定额时钟由 APB 时钟分频得到，APB 时钟通常为 80 MHz。可通过以下公式计算分频数值，其中 t_{Tq} 为时间定额的时钟周期，t_{CLK} 为 APB 时钟周期：
  $t_{Tq} = 2 \times t_{CLK} \times (2^{12} \times BRP.12 + 2^{11} \times BRP.11 + ... + 2^{1} \times BRP.1 + 2^{0} \times BRP.0 + 1)$
- 同步跳宽 (SJW)：SJW 数值在 SJW.0 和 SJW.1 中配置，计算公式为：SJW = (2 x SJW.1 + SJW.0 + 1)。

下表 31.5-2 所示为 TWAI_BUS_TIMING_1_REG 包含的位域。

表 31.5-2. TWAI_BUS_TIMING_1_REG 的 bit (0x1c)

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	SAM	PBS2.2	PBS2.1	PBS2.0	PBS1.3	PBS1.2	PBS1.1	PBS1.0

说明：

- PBS1: 根据以下公式计算缓冲时期段 1 中的时间定额数量：(8 x PBS1.3 + 4 x PBS1.2 + 2 x PBS1.1 + PBS1.0 + 1)。
- PBS2: 根据以下公式计算缓冲时期段 2 中的时间定额数量：(4 x PBS2.2 + 2 x PBS2.1 + PBS2.0 + 1)。
- SAM: 该值置 1 启动三点采样。用于低/中速总线，有利于过滤总线上的尖峰信号。

31.5.3 中断管理

ESP32-S3 TWAI 控制器提供了八种中断，每种中断由寄存器 TWAI_INT_RAW_REG 中的一个位表示。要触发某个特定的中断，须设置 TWAI_INT ENA_REG 中相应的使能位。

TWAI 控制器提供了以下八种中断：

- 接收中断
- 发送中断
- 错误报警中断
- 数据溢出中断
- 被动错误中断
- 仲裁丢失中断
- 总线错误中断
- 总线状态中断

只要在 TWAI_INT_RAW_REG 一个或多个中断位为 1，TWAI 控制器中的中断信号即为有效，当 TWAI_INT_RAW_REG 中的所有位都被清除时，TWAI 控制器中的中断信号则失效。寄存器 TWAI_INT_RAW_REG 被读取后，其中的大多数中断位将自动清除。但是，只有通过 TWAI_RELEASE_BUF 指令位清除所有接收报文后，接收中断位才能被清除。

31.5.3.1 接收中断 (RXI)

当 TWAI 接收 FIFO 中有待读取报文时（TWAI_RX_MESSAGE_CNT_REG > 0），都会触发 RXI。TWAI_RX_MESSAGE_CNT_REG 中记录的报文数量包括接收 FIFO 中的有效报文和溢出报文。直到通过 TWAI_RELEASE_BUF 指令位清除所有挂起接收报文后，RXI 才会失效。

31.5.3.2 发送中断 (TXI)

每当发送缓冲器空闲，将其他报文加载到发送缓冲器中等待发送时，都会触发 TXI。以下情况下，发送缓冲器将变为空闲，同时 TXI 将失效：

- 报文发送已成功完成（如，应答未发现错误）。任何发送失败将自动重发。
- 单次发送已完成（TWAI_TX_COMPLETE 位指示发送成功与否）。
- 使用 TWAI_ABORT_TX 指令位终止报文发送。

31.5.3.3 错误报警中断 (EWI)

每当寄存器 TWAI_STATUS_REG 中 TWAI_ERR_ST 和 TWAI_BUS_OFF_ST 的位值改变时（如，从 0 变为 1 或反之），都会触发 EWI。根据 EWI 触发时 TWAI_ERR_ST 和 TWAI_BUS_OFF_ST 的值分成以下几种情况：

- 如果 TWAI_ERR_ST = 0 且 TWAI_BUS_OFF_ST = 0：
  - 如果 TWAI 控制器处于主动错误状态，则表示 TEC 和 REC 的值都返回到了 TWAI_ERR_WARNING_LIMIT_REG 所设的阈值之下。
  - 如果 TWAI 控制器此前正处于总线恢复状态，则表示此时总线恢复已成功完成。
- 如果 TWAI_ERR_ST = 1 且 TWAI_BUS_OFF_ST = 0：表示 TEC 或 REC 数值已超过 TWAI_ERR_WARNING_LIMIT_REG 所设的阈值。
- 如果 TWAI_ERR_ST = 1 且 TWAI_BUS_OFF_ST = 1：表示 TWAI 控制器已进入 BUS_OFF 状态（因 TEC >= 256）。
- 如果 TWAI_ERR_ST = 0 且 TWAI_BUS_OFF_ST = 1：表示 BUS_OFF 恢复期间，TWAI 控制器的 TEC 数值已低于 TWAI_ERR_WARNING_LIMIT_REG 所设的阈值。

31.5.3.4 数据溢出中断 (DOI)

每当接收 FIFO 中有溢出发生时，都会触发 DOI。DOI 表示接收 FIFO 已满且应立即进行读取，以防出现更多溢出报文。

只有导致接收 FIFO 溢出的第一条报文可触发 DOI（如，当接收 FIFO 从未满变为开始溢出时）。任意后续的溢出报文将不会再次重复触发 DOI。只有当所有接收的（有效报文或溢出）报文都被读取后，才能再次触发 DOI。

31.5.3.5 被动错误中断 (TXI)

每当 TWAI 控制器从主动错误变为被动错误，或反之时，都会触发 EPI。

31.5.3.6 仲裁丢失中断 (ALI)

每当 TWAI 控制器尝试发送报文且丢失仲裁时，都会触发 ALI。TWAI 控制器丢失仲裁的 bit 位置将自动记录在仲裁丢失捕捉寄存器 (TWAI_ARB LOST CAP_REG) 中。仲裁丢失捕捉寄存器被清除（通过 CPU 读取该寄存器）之前，将不会再记录新发生的仲裁失败时的 bit 位置。

31.5.3.7 总线错误中断 (BEI)

每当 TWAI 控制器在 TWAI 总线上检测到错误时，都会触发 BEI。发生总线错误时，总线错误的类型和发生错误时的 bit 位置都将自动记录在错误捕捉寄存器 (TWAI_ERR_CODE_CAP_REG) 中。错误捕捉寄存器被清除（通过 CPU 的读取）之前，将不会再记录新的总线错误信息。

31.5.3.8 总线状态中断 (BSI)

每当 TWAI 控制器在收发总线数据状态与空闲状态之间切换时，都会触发 BSI。当该中断发生时，可通过读取 TWAI_STATUS_REG 寄存器 TWAI_RX_ST 和 TWAI_TX_ST 两个域来判断 TWAI 控制器当前的状态。

31.5.4 发送缓冲器与接收缓冲器

31.5.4.1 缓冲器概述

表 31.5-3. SFF 与 EFF 的缓冲器布局

标准格式 (SFF)		扩展格式 (EFF)	
TWAI 地址	内容	TWAI 地址	内容
0x40	TX/RX 帧信息	0x40	TX/RX 帧信息
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	保留	0x6c	TX/RX data byte 7
0x70	保留	0x70	TX/RX data byte 8

表 31.5-3 所示为发送缓冲器和接收缓冲器的寄存器布局。发送和接收缓冲寄存器的访问地址范围相同，且只有当 TWAI 控制器处于操作模式时才可访问。CPU 的写入操作将访问发送缓冲寄存器，CPU 的读取操作将访问接收缓冲寄存器。发送缓冲器和接受缓冲器中存储报文（接收报文或待发送报文）的寄存器布局和域完全一致。

发送缓冲寄存器用于配置 TWAI 的待发送报文。CPU 会在发送缓冲寄存器进行写入操作，指定报文的帧类型、帧格式、帧 ID 和帧数据（有效载荷）。一旦发送缓冲器配置完成后，CPU 会将 TWAI_CMD_REG 中的 TWAI_TX_REQ 位置 1，以开始报文发送。

- 若是自发自收请求，变更为将 TWAI_SELF_RX_REQ 置 1。
- 若是单次发送，需要同时将 TWAI_TX_REQ 和 TWAI_ABORT_TX 置 1。

接收缓冲寄存器将映射接收 FIFO 中的第一条报文。CPU 会在接收缓冲寄存器中进行读取操作，获取第一条报文的帧类型、帧格式、帧 ID 和帧数据（有效载荷）。读取完接收缓冲寄存器中的报文后，CPU 通过将 TWAI_CMD_REG 中的 TWAI_RELEASE_BUF 位置 1 来来清除接收缓冲寄存器，若接收 FIFO 中仍有待处理的报文，按照接收报文的先后次序将最早接收到的报文映射到接收缓冲寄存器。

31.5.4.2 帧信息

帧信息的长度为 1-byte，主要用于明确报文的帧类型、帧格式以及数据长度。下表 31.5-4 所示为帧信息域。

表 31.5-4. TX/RX 帧信息 (SFF/EFF)；TWAI 地址 0x40

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	FF	RTR	X	X	DLC.3	DLC.2	DLC.1	DLC.0

说明：

1. FF: 主要明确某报文属于 EFF 还是 SFF。当 FF 位为 1 时，该报文为 EFF，当 FF 位为 0 时，该报文为 SFF。
2. RTR: 主要明确某报文是数据帧还是远程帧。当 RTR 位为 1 时，该报文为远程帧，当 RTR 位为 0 时，该报文为数据帧。
3. X: 无关 bit，可以是任意值。
4. DLC: 主要明确数据帧中的数据字节数量，或从远程帧中请求的数据字节数量。TWAI 数据帧的最大载荷为 8 个数据字节，因此 DLC 的数值范围应是 0 ~ 8。

31.5.4.3 帧标识符

若报文为 SFF，则对应的帧标识符域为 2-bytes (11-bits)；若报文为 EFF，则对应的帧标识符域为 4-bytes (29-bits)。

下表 Table 31.5-5-31.5-6 所示为 SFF (11-bits) 报文的帧标识符域。

表 31.5-5. TX/RX 标识符 1 (SFF); TWAI 地址 0x44

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3

表 31.5-6. TX/RX 标识符 2 (SFF); TWAI 地址 0x48

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.2	ID.1	ID.0	X[1]	X[2]	X[2]	X[2]	X[2]

说明：

1. 无关项。建议设置为与接收缓冲器兼容（设为 RTR），以防需使用自接收功能（或与自接收功能一起使用）。
2. 无关项。建议设置为与接收缓冲器兼容（设为 0），以防需使用自接收功能（或与自接收功能一起使用）。

下表 31.5-7-31.5-10 所示为 EFF (29-bits) 报文的帧标识符域。

表 31.5-7. TX/RX 标识符 1 (EFF); TWAI 地址 0x44

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.28	ID.27	ID.26	ID.25	ID.24	ID.23	ID.22	ID.21

表 31.5-8. TX/RX 标识符 2 (EFF); TWAI 地址 0x48

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.20	ID.19	ID.18	ID.17	ID.16	ID.15	ID.14	ID.13

表 31.5-9. TX/RX 标识符 3 (EFF); TWAI 地址 0x4c

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.12	ID.11	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5

表 31.5-10. TX/RX 标识符 4 (EFF); TWAI 地址 0x50

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.4	ID.3	ID.2	ID.1	ID.0	X[1]	X[2]	X[2]

说明：

1. 无关项。建议设置为与接收缓冲器兼容（设为 RTR），以防需使用自接收功能（或与自接收功能一起使用）。
2. 无关项。建议设置为与接收缓冲器兼容（设为 0），以防需使用自接收功能（或与自接收功能一起使用）。

31.5.4.4 帧数据

帧数据域包含发送或接收的数据帧，范围为 0 ~ 8 bytes。其中的有效字节数应与 DLC 相同。但是，如果 DLC 数值大于 8，则帧数据域的有效字节数仍为 8。远程帧中不包含数据载荷，因此不存在帧数据域。

比如，当发送 5 个字节的数据帧时，CPU 应在 DLC 域中写入数值 5，并将数据写入数据域 1 ~ 5 字节对应的寄存器。同样，当接收 DLC 为 5 的数据帧时，只有 1 ~ 5 数据字节中包含 CPU 可以读取的有效载荷数据。

31.5.5 接收 FIFO 和数据溢出

接收 FIFO 是一个 64-byte 的内部缓冲器，用于以先进先出的原则存储接收到的报文。一条接收报文可在接收 FIFO 中占 3 ~ 13 bytes 空间，且其中字节序与接收缓冲器的寄存器地址顺序相同。接收缓冲寄存器将被映射到接收 FIFO 中第一条报文。

当 TWAI 控制器接收到一条报文时，TWAI_RX_MESSAGE_COUNTER 的值将增加 1，最大值为 64。如果接收 FIFO 中有足够的剩余空间，报文内容将被写入到接收 FIFO 中。读取接收缓冲器中的消息后，通过将 TWAI_RELEASE_BUF 的位置 1，释放接收 FIFO 第一条报文所占的空间，TWAI_RX_MESSAGE_COUNTER 的值也将减小 1。然后，接收缓冲器将映射接收 FIFO 中的下一条报文。

当 TWAI 控制器接收到一条报文，但接收 FIFO 没有足够空间完整地存储这条接收报文时（不论是因为报文内容大小大于接收 FIFO 中的空闲空间，还是因为接收 FIFO 已满），便会发生数据溢出。

数据溢出发生时：

- 接收 FIFO 中剩余的空间将填满溢出报文的内容。如果接收 FIFO 已满，则无法存储溢出报文的任何内容。

- 接收 FIFO 首次发生数据溢出时，将触发数据溢出中断。
- 溢出报文仍将增加 TWAI_RX_MESSAGE_COUNTER 的值到最大值 64。
- 接收 FIFO 将在内部将溢出报文标记为无效。可使用 TWAI_MISS_ST 位，确认目前接收缓冲器映射的报文是有效报文还是溢出报文。

为了清除接收 FIFO 中的溢出报文，应重复调用 TWAI_RELEASE_BUF，直到 TWAI_RX_MESSAGE_COUNTER 为 0。这样可以读取接收 FIFO 中的所有有效报文，并清除所有溢出报文。

31.5.6 接收滤波器

接收滤波器允许 TWAI 控制器根据报文 ID 过滤接收报文（有时可以过滤报文的第一个数据字节和帧类型）。只有通过过滤的报文才能存储到接收 FIFO 中。接收滤波器的使用可以一定程度地减轻 TWAI 控制器的运行负荷（如，可减少使用接收 FIFO 和发生接收中断的次数），因为 TWAI 控制器将只需要操作一小部分过滤后的报文。

只有当 TWAI 控制器处于复位模式时，才可以访问接收滤波器的配置寄存器，因为这些配置寄存器和发送/接收缓冲寄存器的地址空间相同。

接收滤波器的配置寄存器由 32-bit 的 Code 值和 32-bit 的 Mask 值组成。Code 值将指定一种位排列模式，每条过滤报文中的位都必须匹配该模式，才能使该报文通过过滤。Mask 值可屏蔽 Code 值中的某些位（将屏蔽位设置为“不相关”的位）。如图 31.5-1 所示，为了使报文通过过滤，每条过滤报文的 ID 都必须匹配 Code 值所设模式或者被 Mask 值屏蔽。

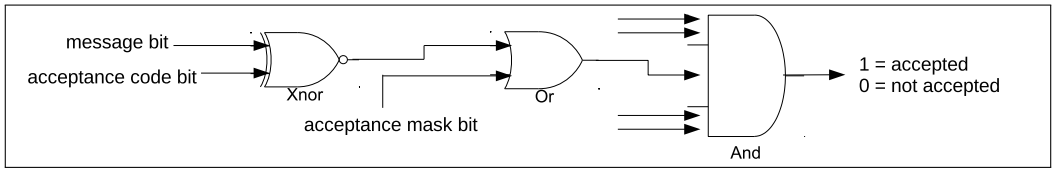


图 31.5-1. 接收滤波器

TWAI 控制器的接收滤波器允许 32-bit 的 Code 值和 Mask 值定义单个滤波器（单滤波模式），或两个滤波器（双滤波模式）。接收滤波器如何解析 32-bit 的 code 值和 mask 值，取决于滤波模式以及接收报文的格式（如，SFF 还是 EFF）。

31.5.6.1 单滤波模式

将 TWAI_RX_FILTER_MODE 的位置 1，可启动单滤波模式。此后，32-bit code/mask 的值将定义单个滤波器。

单个滤波器可过滤数据帧和远程帧中的以下位：

- SFF
  - 11-bit ID 整体
  - RTR bit
  - 数据字节 1 和数据字节 2
- EFF
  - 29-bit ID 整体
  - RTR bit

下图 31.5-2 所示为单滤波模式下如何解析 32-bit code/mask 的值。

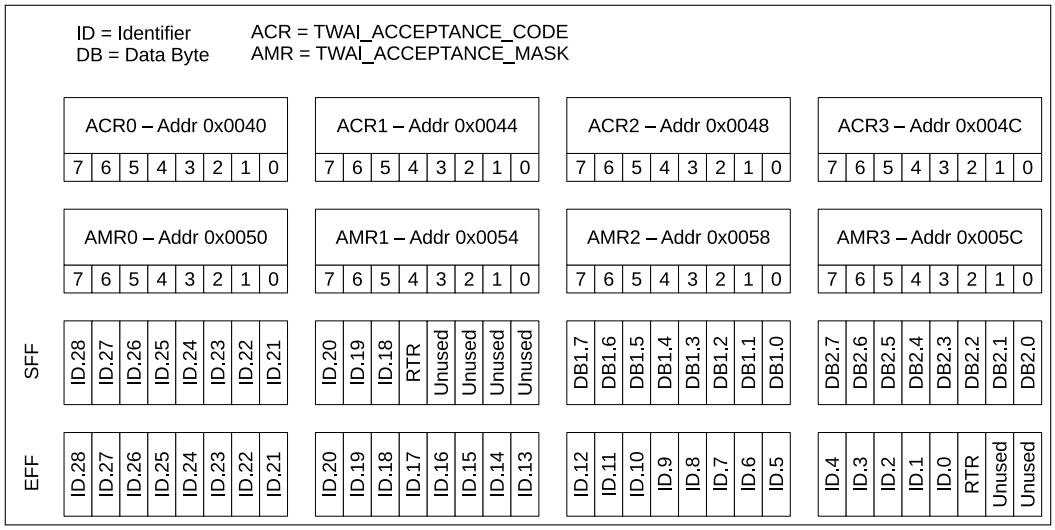


图 31.5-2. 单滤波模式

31.5.6.2 双滤波模式

将 TWAI_RX_FILTER_MODE 的位置 0，可启动双滤波模式。此后，32-bit code/mask 的值将定义两个滤波器之一，即滤波器 1 或滤波器 2。双滤波模式下，如果报文通过这两个滤波器中的至少一个，则表示该报文已成功通过过滤。

这两个滤波器可以过滤数据帧和远程帧中的以下位：

- SFF
  - 11-bit ID 整体
  - RTR bit
  - 数据字节 1（仅适用于滤波器 1）
- EFF
  - 29-bit ID 的前 16-bit

下图 31.5-3 所示为双滤波模式下如何解析 32-bit code/mask 的值。

31.5.7 错误管理

TWAI 协议要求每个 TWAI 节点中都包含发送错误计数 (TEC) 和接收错误计数 (REC)。这两个错误计数的数值决定了 TWAI 控制器当前的错误状态（如，主动错误、被动错误、离线）。TWAI 控制器将 TEC 和 REC 的数值分别存储在 TWAI_TX_ERR_CNT_REG 和 TWAI_RX_ERR_CNT_REG 中，CPU 可随时进行读取。除了错误状态之外，TWAI 控制器还提供错误报警限制 (EWL) 的功能，这个功能可在 TWAI 控制器进入被动错误状态之前，提醒用户当前发生的严重总线错误。

TWAI 控制器的当前错误状态通过以下各数值和状态位体现，即：TEC、REC、TWAI_ERR_ST 和 TWAI_BUS_OFF_ST。这些数值和状态位的变化也将触发中断，从而提醒用户当前的错误状态变化（可参见第 31.5.3 章）。下图 31.5-4 所示为错误状态、上述数值和状态位以及错误状态相关中断之间的关系。

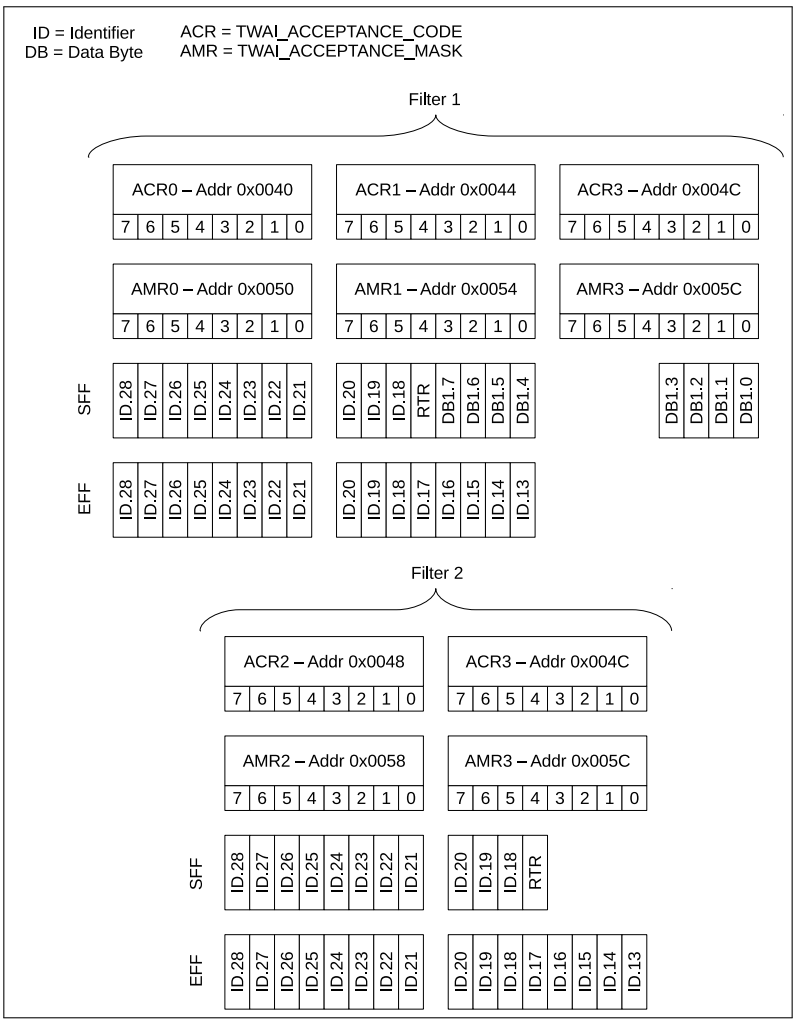


图 31.5-3. 双滤波模式

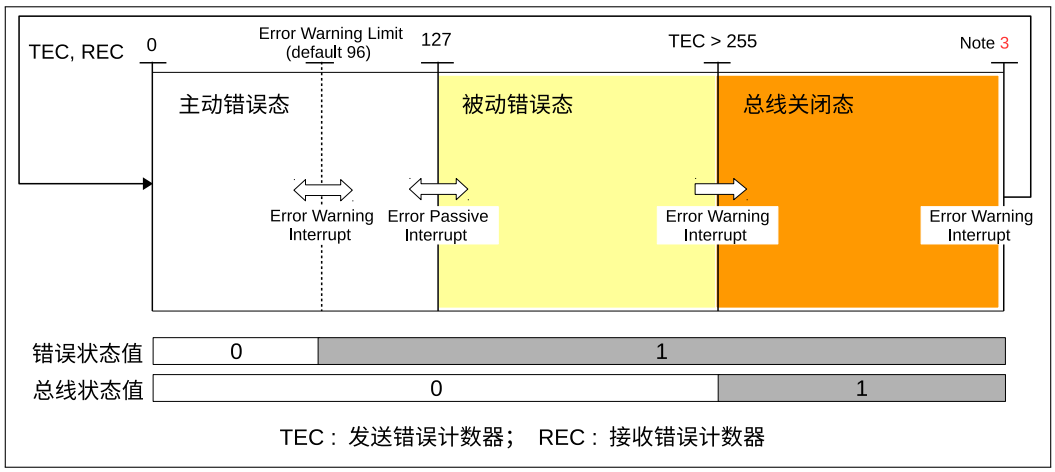


图 31.5-4. 错误状态变化

31.5.7.1 错误报警限制

错误报警限制 (EWL) 为 TEC 和 REC 的可配置阈值，若错误计数数值超过该阈值，将触发 EWI 中断。EWL 将作为一个报警功能提示当前发生的严重 TWAI 总线错误，且在 TWAI 控制器进入被动错误状态之前被触发。EWL 数值应在寄存器 TWAI_ERR_WARNING_LIMIT_REG 中进行配置，配置同时 TWAI 控制器必须处于复位模式下。

TWAI_ERR_WARNING_LIMIT_REG 默认数值为 96。

当 TEC 和/或 REC 数值大于等于 EWL 数值时，TWAI_ERR_ST 位将立即被置 1。同理，当 TEC 和 REC 数值都小于 EWL 数值时，TWAI_ERR_ST 位将立即复位为 0。只要 TWAI_ERR_ST（或 TWAI_BUS_OFF_ST）位值发生变化，便会触发错误报警中断。

31.5.7.2 被动错误

当 TEC 或 REC 数值大于 127 时，TWAI 控制器处于被动错误状态。同理，当 TEC 和 REC 数值都小于等于 127 时，TWAI 控制器进入主动错误状态。每当 TWAI 控制器从主动错误状态变为被动错误状态，或反之时，都将触发被动错误中断。

31.5.7.3 离线状态与离线恢复

当 TEC 数值大于 255 时，TWAI 控制器将进入离线状态。进入离线状态后，TWAI 控制器将自动进行以下动作：

- REC 数值置为 0
- TEC 数值置为 127
- TWAI_BUS_OFF_ST 位置 1
- 进入复位模式

每当 TWAI_BUS_OFF_ST 位（或 TWAI_ERR_ST 位）数值发生变化时，都将触发错误报警中断。

为了返回主动错误状态，TWAI 控制器必须进行离线恢复。要启动离线恢复，首先需要退出复位模式，进入操作模式。然后要求 TWAI 控制器在总线上检测到 128 次 11 个连续隐性位。

每一次 TWAI 控制器检测到 11 个连续隐性位时，TEC 数值都将减小，以追踪离线恢复进程。当离线恢复完成后（TEC 数值从 127 减小到 0），TWAI_BUS_OFF_ST 位将自动复位为 0，从而触发错误报警中断。

31.5.8 错误捕捉

错误捕捉 (ECC) 功能允许 TWAI 控制器以错误代码的形式记录 TWAI 总线错误的错误类型和 bit 位置。当检测到一个 TWAI 总线错误时，总线错误中断将被触发，相应的错误代码将记录在 TWAI_ERR_CODE_CAP_REG 中。寄存器 TWAI_ERR_CODE_CAP_REG 中存储的当前错误代码被读取之前，后续的总线错误中断触发时，将不会再记录错误代码。

下表 31.5-11 所示为寄存器 TWAI_ERR_CODE_CAP_REG 中的域：

表 31.5-11. TWAI_ERR_CODE_CAP_REG 中的位信息 (0x30)

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ERRC.1	ERRC.0	DIR	SEG.4	SEG.3	SEG.2	SEG.1	SEG.0

说明：

- 错误代码 (ERRC)：表示总线错误的类型。00 代表位错误，01 代表格式错误，10 代表填充错误，11 代表其他错误类型。
- 传输方向 (DIR)：表示总线错误发生时，TWAI 控制器处于发送器状态还是接收器状态。0 代表发送器，1 代表接收器。
- 错误段 (SEG)：表示总线错误发生在 TWAI 报文的哪个段。

下表 31.5-12 所示为 SEG.0 ~ SEG.4 的位信息。

表 31.5-12. SEG.4 - SEG.0 的位信息

Bit SEG.4	Bit SEG.3	Bit SEG.2	Bit SEG.1	Bit SEG.0	描述
0	0	0	1	1	帧起始
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	保留位 1
0	1	0	0	1	保留位 0
0	1	0	1	1	数据长度代码
0	1	0	1	0	数据域
0	1	0	0	0	CRC 序列
1	1	0	0	0	CRC 分界符
1	1	0	0	1	确认槽
1	1	0	1	1	确认分界符
1	1	0	1	0	帧结束
1	0	0	1	0	间歇域
1	0	0	0	1	主动错误标志
1	0	1	1	0	被动错误标志
1	0	0	1	1	兼容显性位
1	0	1	1	1	错误分界符
1	1	1	0	0	过载标志

说明：

- Bit SRTR: 标准格式 RTR bit。
- Bit IDE: 标识符扩展位。0 表示标准格式。

31.5.9 仲裁丢失捕捉

仲裁丢失捕捉 (ALC) 功能允许 TWAI 控制器记录丢失仲裁的 bit 位置。当 TWAI 控制器丢失仲裁时，bit 位置将被记录在寄存器 TWAI_ARB LOST CAP_REG 中，同时触发仲裁丢失中断。

后续的仲裁丢失中断触发时，bit 位置将不会被记录在 TWAI_ARB LOST CAP_REG 中，直到 TWAI_ERR_CODE_CAP_REG 中的当前仲裁丢失捕捉被读取。

下表 31.5-13 所示为 TWAI_ERR_CODE_CAP_REG 中的位域；下图 31.5-5 所示为一条 TWAI 报文的 bit 位置。

表 31.5-13. TWAI_ARB LOST CAP_REG 中的位信息 (0x2c)

Bit 31-5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	BITNO.4	BITNO.3	BITNO.2	BITNO.1	BITNO.0

说明：

- 位号 (BITNO)：表示丢失仲裁的 TWAI 报文的第 n 个位。

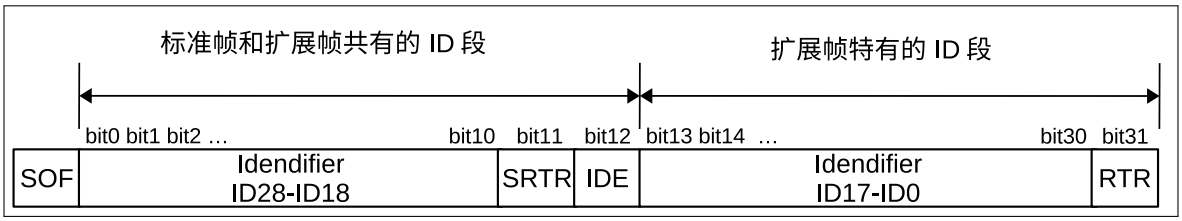


图 31.5-5. 丢失仲裁的 bit 位置

31.6 寄存器列表

请注意，“访问权限”一栏中，“|”用于区分第 31.5.1 中描述的工作模式，其中左侧为操作模式下的访问权限，右侧标红字体为复位模式下的访问权限。本小节的所有地址均为相对于 Two-wire Automotive Interface 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

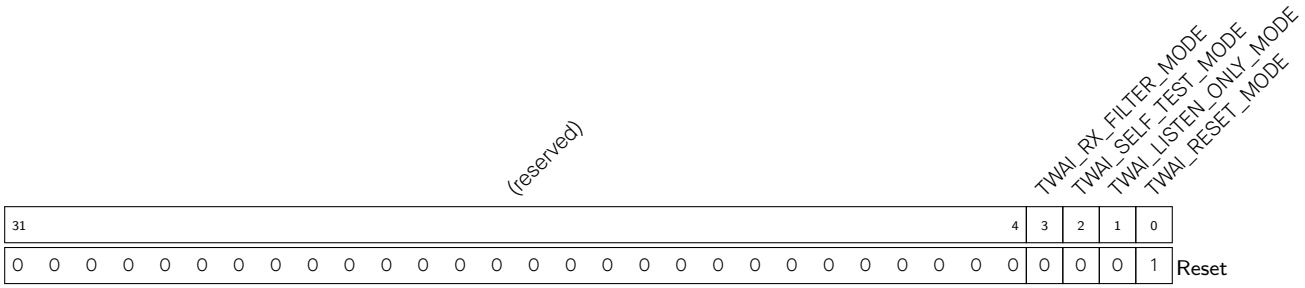
请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问权限
配置寄存器			
TWAI_MODE_REG	模式寄存器	0x0000	R/W
TWAI_BUS_TIMING_0_REG	时序配置寄存器 0	0x0018	RO | | R/W
TWAI_BUS_TIMING_1_REG	时序配置寄存器 1	0x001C	RO | | R/W
TWAI_ERR_WARNING_LIMIT_REG	错误寄存器	0x0034	RO | | R/W
TWAI_DATA_0_REG	数据寄存器 0	0x0040	WO | R/W
TWAI_DATA_1_REG	数据寄存器 1	0x0044	WO | R/W
TWAI_DATA_2_REG	数据寄存器 2	0x0048	WO | R/W
TWAI_DATA_3_REG	数据寄存器 3	0x004C	WO | R/W
TWAI_DATA_4_REG	数据寄存器 4	0x0050	WO | R/W
TWAI_DATA_5_REG	数据寄存器 5	0x0054	WO | R/W
TWAI_DATA_6_REG	数据寄存器 6	0x0058	WO | R/W
TWAI_DATA_7_REG	数据寄存器 7	0x005C	WO | R/W
TWAI_DATA_8_REG	数据寄存器 8	0x0060	WO | RO
TWAI_DATA_9_REG	数据寄存器 9	0x0064	WO | RO
TWAI_DATA_10_REG	数据寄存器 10	0x0068	WO | RO
TWAI_DATA_11_REG	数据寄存器 11	0x006C	WO | RO
TWAI_DATA_12_REG	数据寄存器 12	0x0070	WO | RO
TWAI_CLOCK_DIVIDER_REG	时钟分频寄存器	0x007C	不定
控制寄存器			
TWAI_CMD_REG	指令寄存器	0x0004	WO
状态寄存器			
TWAI_STATUS_REG	状态寄存器	0x0008	RO
TWAI_ARB LOST CAP_REG	仲裁丢失寄存器	0x002C	RO
TWAI_ERR_CODE_CAP_REG	错误捕获寄存器	0x0030	RO
TWAI_RX_ERR_CNT_REG	接收错误寄存器	0x0038	RO | R/W
TWAI_TX_ERR_CNT_REG	发送错误寄存器	0x003C	RO | R/W
TWAI_RX_MESSAGE_CNT_REG	接收数据寄存器	0x0074	RO
中断寄存器			
TWAI_INT_RAW_REG	中断寄存器	0x000C	RO
TWAI_INT ENA_REG	中断使能寄存器	0x0010	R/W

31.7 寄存器

请注意“访问权限”一栏中，“|”左侧为操作模式下的访问权限，右侧标红字体为复位模式下的访问权限。本小节的所有地址均为相对于 Two-wire Automotive Interface 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 31.1. TWAI_MODE_REG (0x0000)



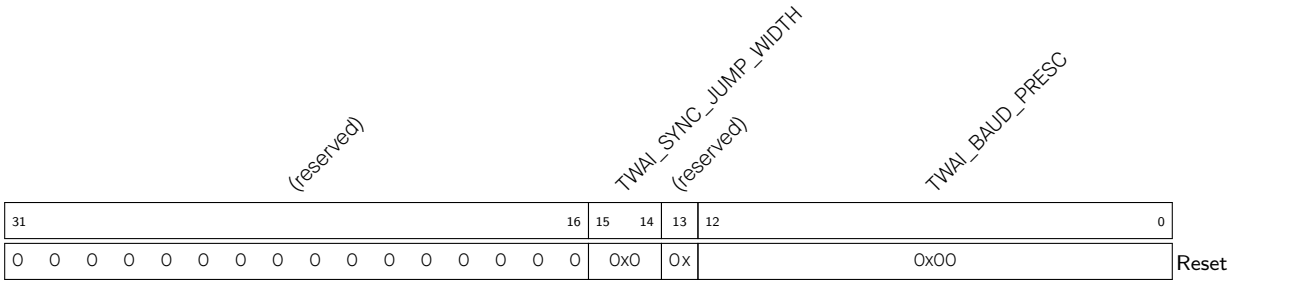
TWAI_RESET_MODE 配置 TWAI 控制器操作模式。1：复位模式；0：操作模式。(R/W)

TWAI_LISTEN_ONLY_MODE 置 1 进入只听模式，处于该模式下的节点只接收总线上数据，不产生应答信号，也不更新接收错误计数。(R/W)

TWAI_SELF_TEST_MODE 置 1 启动自测模式，此模式下发送节点发送完数据后无需应答信号反馈。该模式常配合自接自收指令测试某个节点。(R/W)

TWAI_RX_FILTER_MODE 配置滤波模式。0: 双滤波模式；1: 单滤波模式。(R/W)

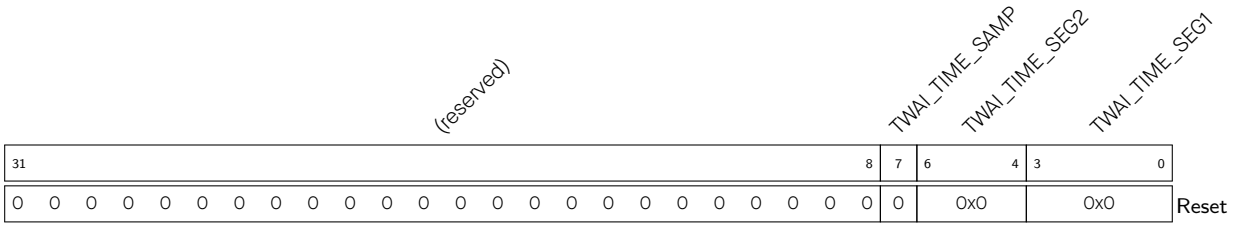
Register 31.2. TWAI_BUS_TIMING_0_REG (0x0018)



TWAI_BAUD_PRESC 预分频值，决定分频比例。(RO | R/W)

TWAI_SYNC_JUMP_WIDTH 同步跳宽 (SJW)，范围为 1 ~ 4 个时间定额。(RO | R/W)

Register 31.3. TWAI_BUS_TIMING_1_REG (0x001C)

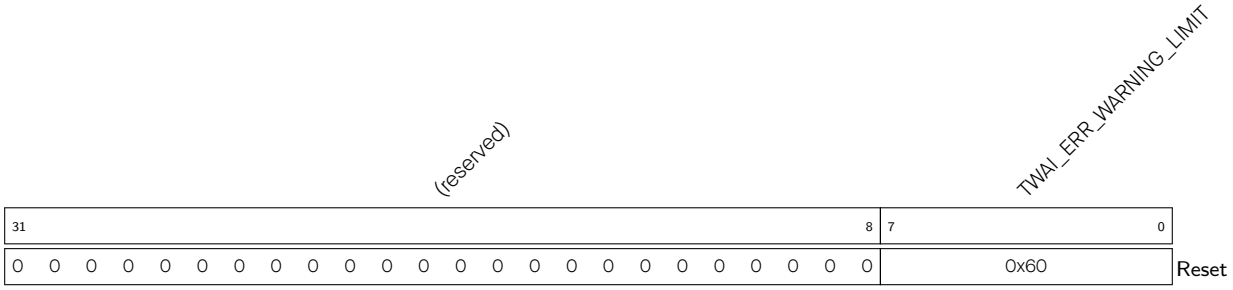


TWAI_TIME_SEG1 缓冲时期段 1 的宽度。(RO | R/W)

TWAI_TIME_SEG2 缓冲时期段 2 的宽度。(RO | R/W)

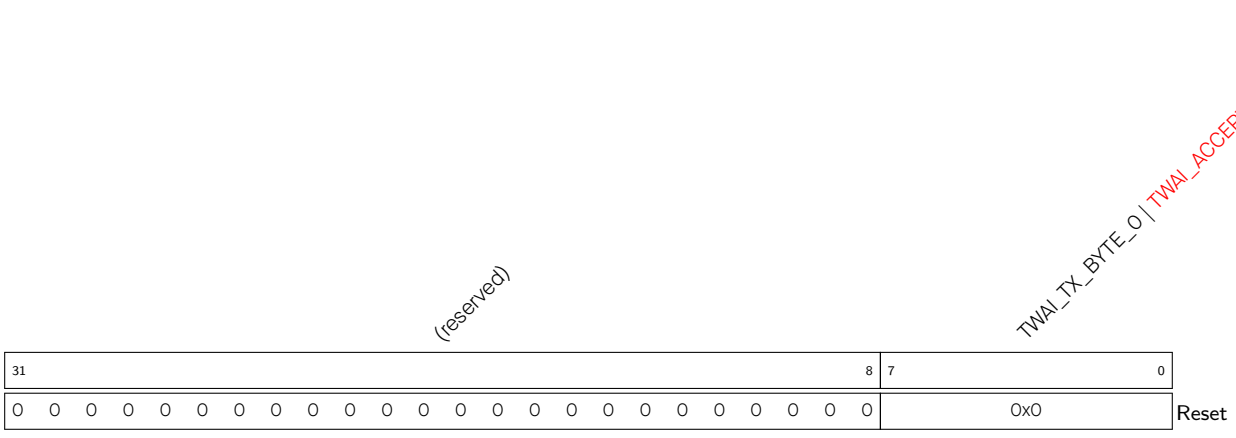
TWAI_TIME_SAMP 采样点数目。0：采样 1 次；1：采样三次 (RO | R/W)

Register 31.4. TWAI_ERR_WARNING_LIMIT_REG (0x0034)



TWAI_ERR_WARNING_LIMIT 错误报警阈值，当任一错误计数数值超过该阈值或者所有错误计数数值都小于该阈值时，将触发错误报警中断（使能信号有效情况下）。(RO | R/W)

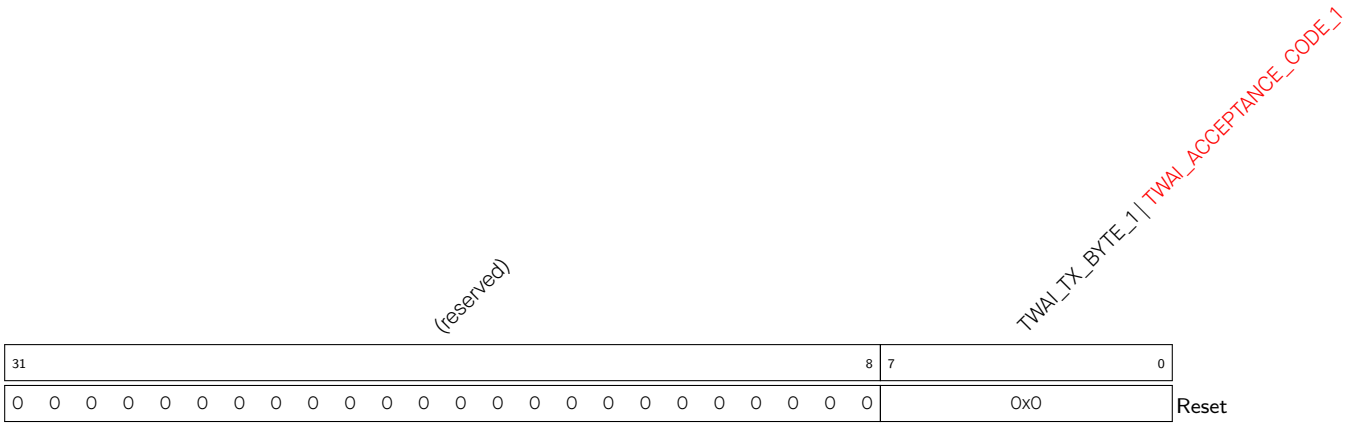
Register 31.5. TWAI_DATA_0_REG (0x0040)



TWAI_TX_BYTE_0 操作模式下，存储着待发送数据的第 0 个字节内容。(WO)

TWAI_ACCEPTANCE_CODE_0 复位模式下，存储着滤波编码的第 0 个字节。(R/W)

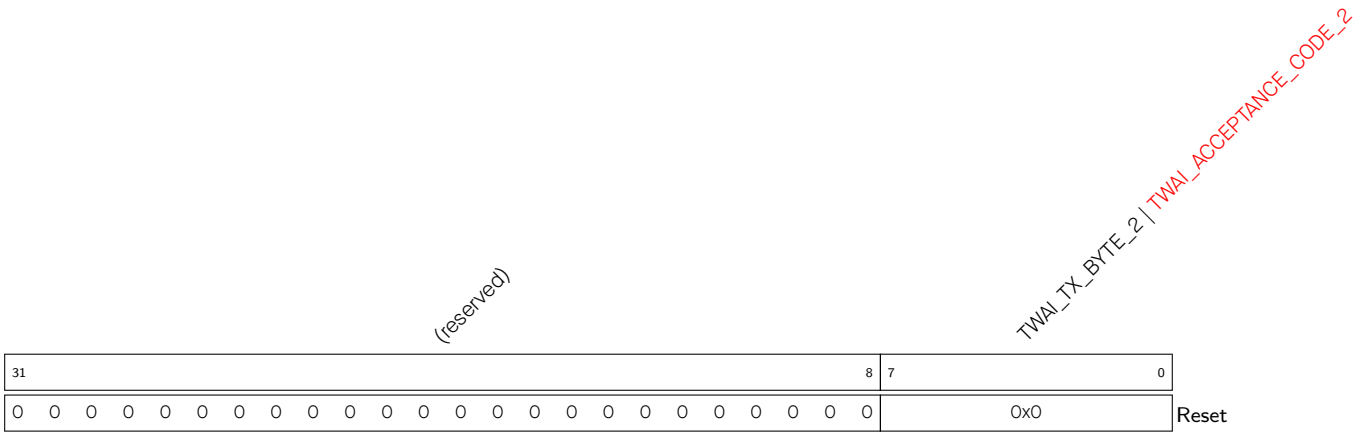
Register 31.6. TWAI_DATA_1_REG (0x0044)



TWAI_TX_BYTE_1 操作模式下，存储着待发送数据的第 1 个字节内容。(WO)

TWAI_ACCEPTANCE_CODE_1 复位模式下，存储着滤波编码的第 1 个字节。(R/W)

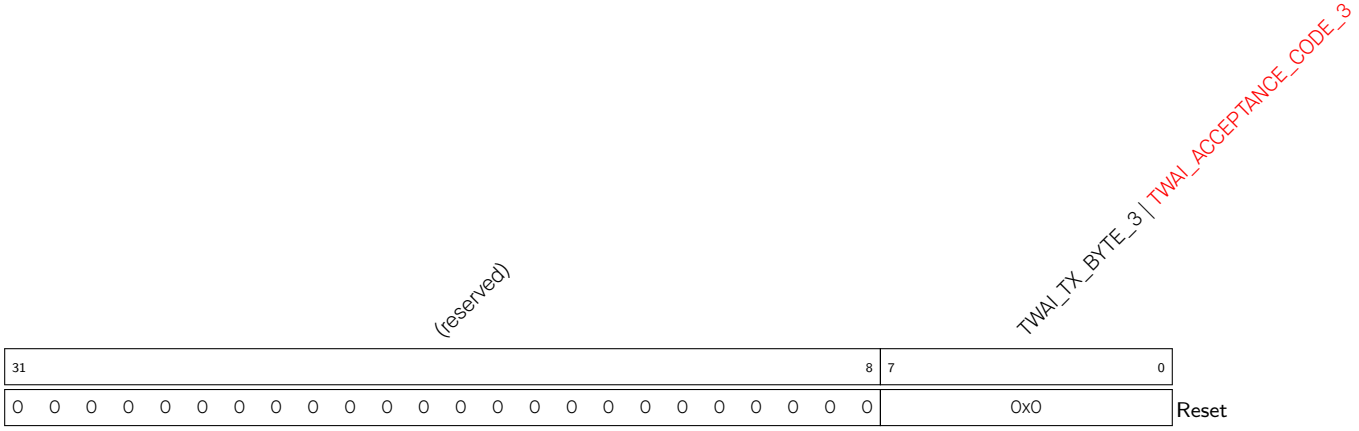
Register 31.7. TWAI_DATA_2_REG (0x0048)



TWAI_TX_BYTE_2 操作模式下，存储着待发送数据的第 2 个字节内容。(WO)

TWAI_ACCEPTANCE_CODE_2 复位模式下，存储着滤波编码的第 2 个字节。(R/W)

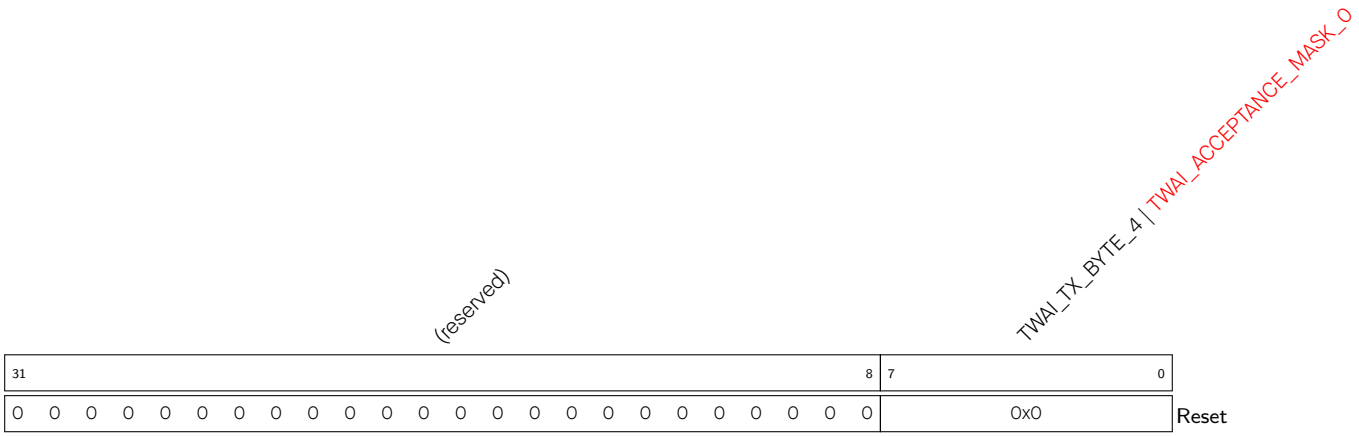
Register 31.8. TWAI_DATA_3_REG (0x004C)



TWAI_TX_BYTE_3 操作模式下，存储着待发送数据的第 3 个字节内容。(WO)

TWAI_ACCEPTANCE_CODE_3 复位模式下，存储着滤波编码的第 3 个字节。(R/W)

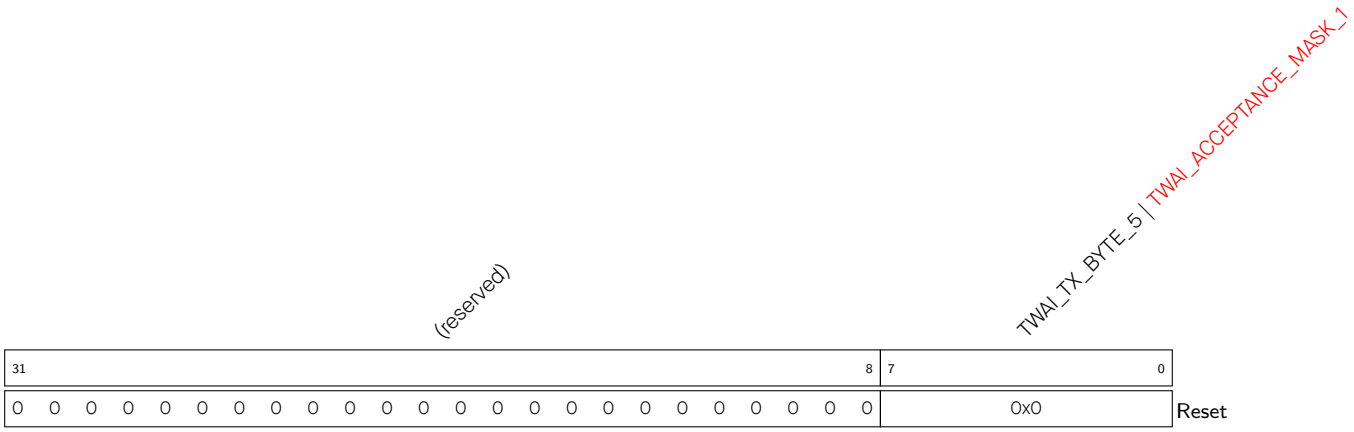
Register 31.9. TWAI_DATA_4_REG (0x0050)



TWAI_TX_BYTE_4 操作模式下，存储着待发送数据的第 4 个字节内容。(WO)

TWAI_ACCEPTANCE_MASK_0 复位模式下，存储着滤波编码的第 0 个字节。(R/W)

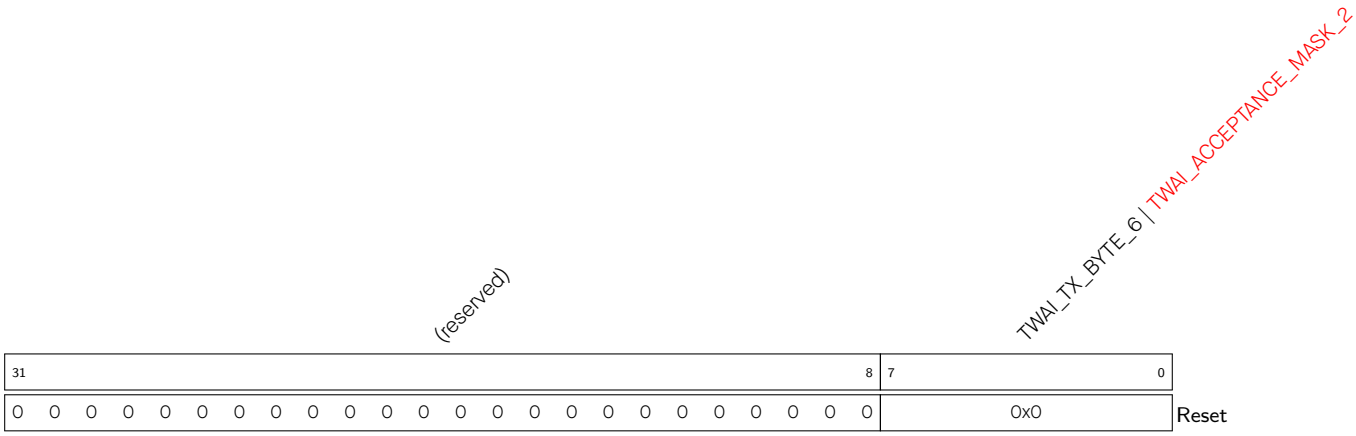
Register 31.10. TWAI_DATA_5_REG (0x0054)



TWAI_TX_BYTE_5 操作模式下，存储着待发送数据的第 5 个字节内容。(WO)

TWAI_ACCEPTANCE_MASK_1 复位模式下，存储着滤波编码的第 1 个字节。(R/W)

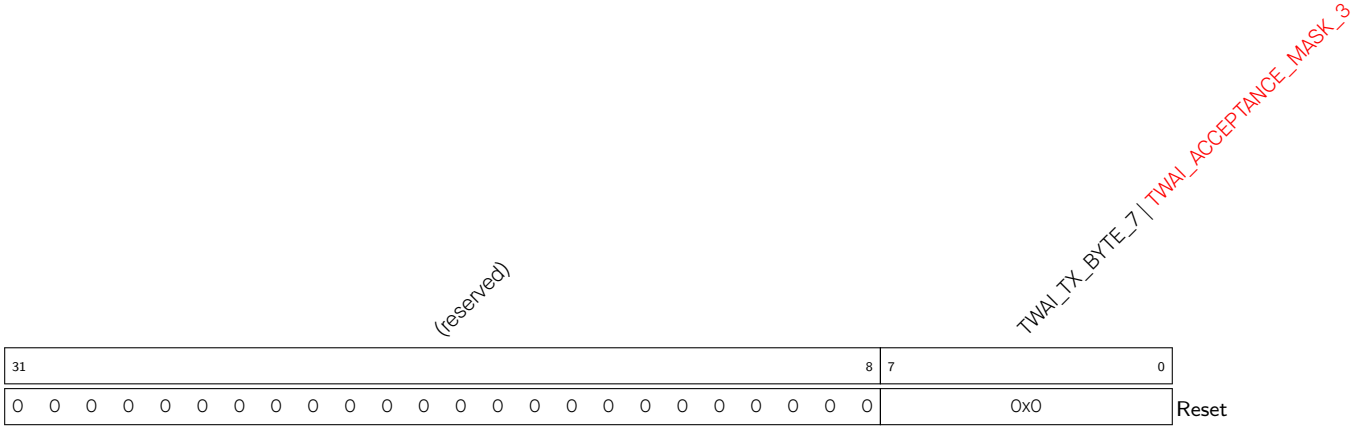
Register 31.11. TWAI_DATA_6_REG (0x0058)



TWAI_TX_BYTE_6 操作模式下，存储着待发送数据的第 6 个字节内容。(WO)

TWAI_ACCEPTANCE_MASK_2 复位模式下，存储着滤波编码的第 2 个字节。(R/W)

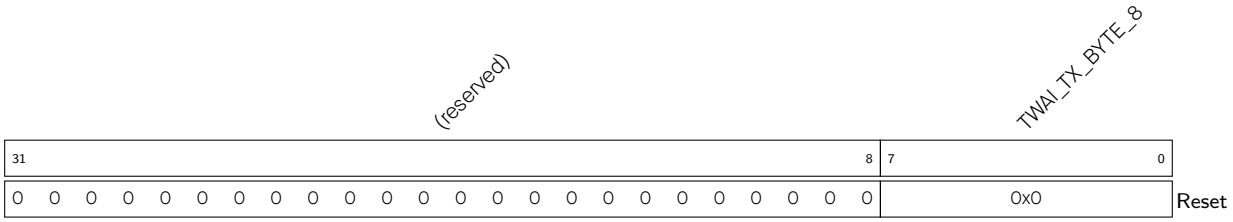
Register 31.12. TWAI_DATA_7_REG (0x005C)



TWAI_TX_BYTE_7 操作模式下，存储着待发送数据的第 7 个字节内容。(WO)

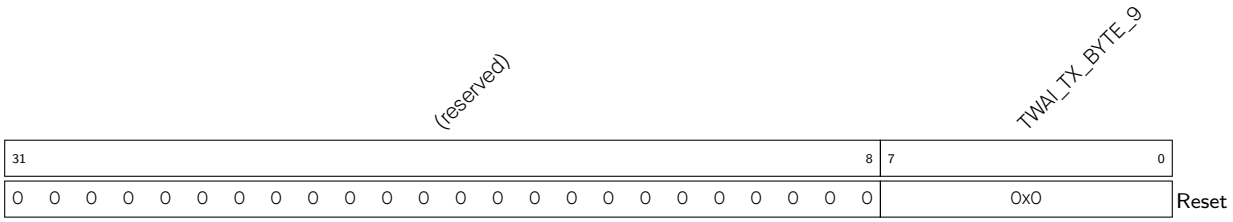
TWAI_ACCEPTANCE_MASK_3 复位模式下，存储着滤波编码的第 3 个字节。(R/W)

Register 31.13. TWAI_DATA_8_REG (0x0060)



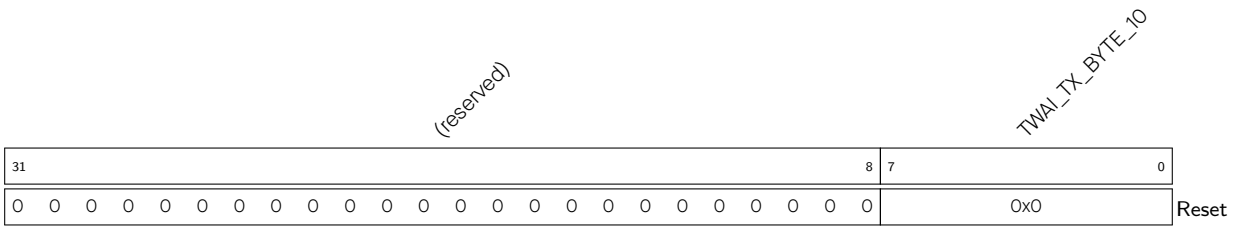
TWAI_TX_BYTE_8 操作模式下，存储着待发送数据的第 8 个字节内容。(WO)

Register 31.14. TWAI_DATA_9_REG (0x0064)



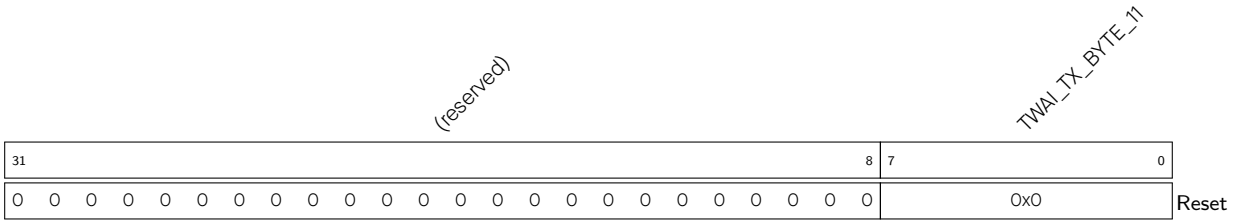
TWAI_TX_BYTE_9 操作模式下，存储着待发送数据的第 9 个字节内容。(WO)

Register 31.15. TWAI_DATA_10_REG (0x0068)



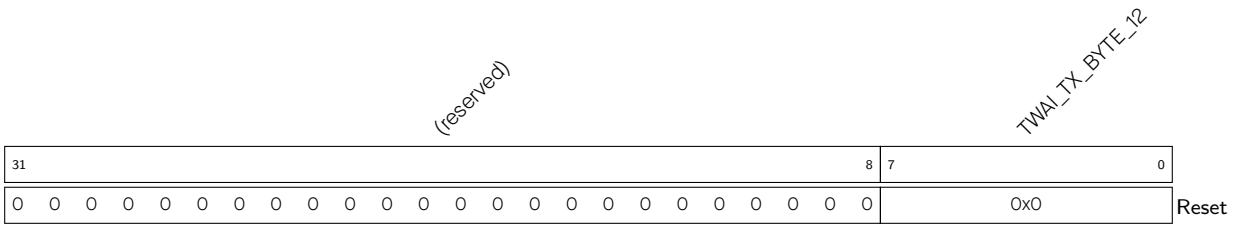
TWAI_TX_BYTE_10 操作模式下，存储着待发送数据的第 10 个字节内容。(WO)

Register 31.16. TWAI_DATA_11_REG (0x006C)



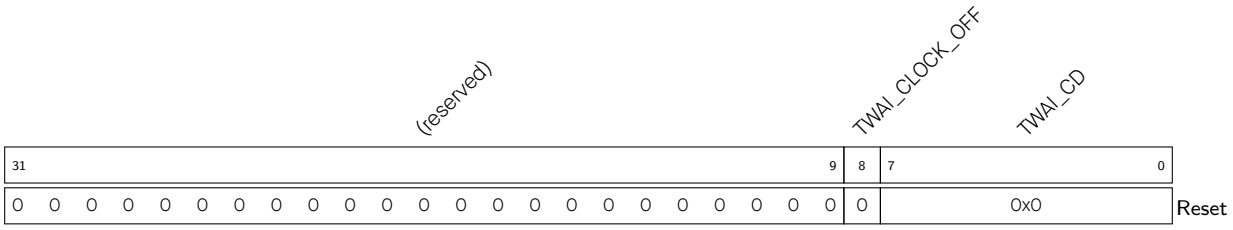
TWAI_TX_BYTE_11 操作模式下，存储着待发送数据的第 11 个字节内容。(WO)

Register 31.17. TWAI_DATA_12_REG (0x0070)



TWAI_TX_BYTE_12 操作模式下，存储着待发送数据的第 12 个字节内容。(WO)

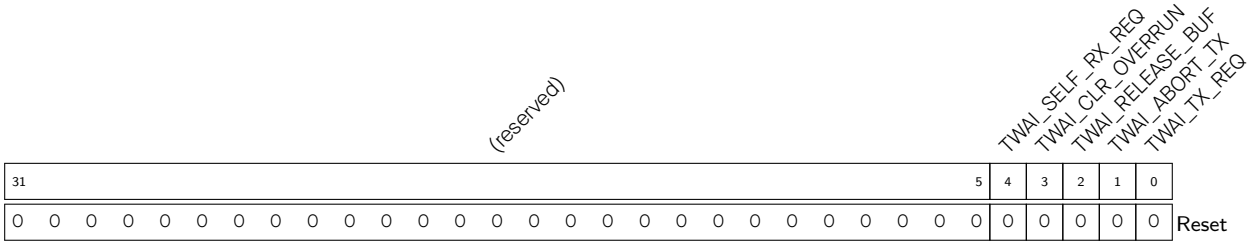
Register 31.18. TWAI_CLOCK_DIVIDER_REG (0x007C)



TWAI_CD 配置输出时钟 CLKOUT 的分频系数。(R/W)

TWAI_CLOCK_OFF 复位模式下可配。1：关闭输出的 CLKOUT 时钟；0：打开 CLKOUT 时钟（RO | R/W）

Register 31.19. TWAI_CMD_REG (0x0004)



TWAI_TX_REQ 置 1 驱动节点开始发送数据任务。(WO)

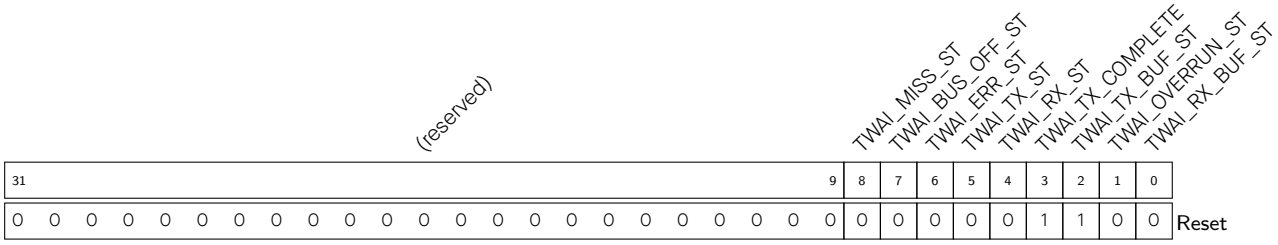
TWAI_ABORT_TX 置 1 取消当前还未开始的发送任务。(WO)

TWAI_RELEASE_BUF 置 1 释放接收缓冲器。(WO)

TWAI_CLR_OVERRUN 置 1 清除数据溢出状态。(WO)

TWAI_SELF_RX_REQ 自接自收命令。置 1 允许发送节点发送数据的同时接收总线上的数据。(WO)

Register 31.20. TWAI_STATUS_REG (0x0008)



TWAI_RX_BUF_ST 若值为 1, 表明接收缓冲器中数据不为空, 至少有一个已经接收到的数据包。(RO)

TWAI_OVERRUN_ST 若值为 1，表明接收 FIFO 中存储的数据已满，产生了溢出。(RO)

TWAI_TX_BUF_ST 若值为 1，表明发送缓冲器为空，允许写入待发送数据。(RO)

TWAI_TX_COMPLETE 若值为 1，表明成功从总线上接收到一个数据包。(RO)

TWAI_RX_ST 若值为 1，表明节点正在从总线上接收数据。(RO)

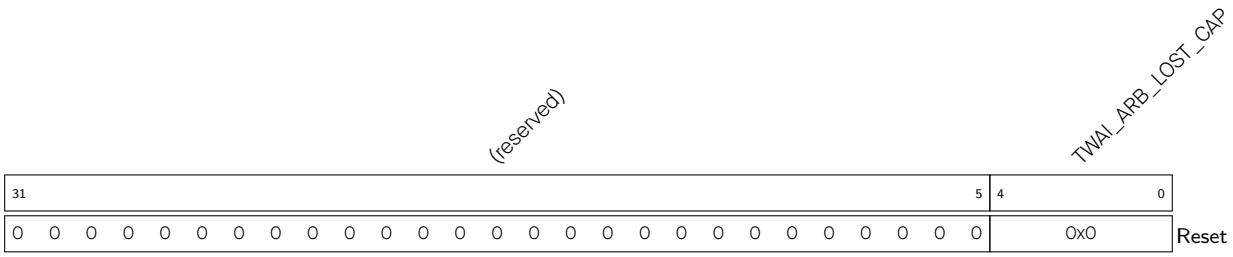
TWAI_TX_ST 若值为 1，表明节点正在往总线上发送数据。(RO)

TWAI_ERR_ST 若值为 1，表明接收错误计数和发送错误计数中至少有一个数值大于等于寄存器 TWAI_ERR_WARNING_LIMIT_REG 中配置的数值 (RO)

TWAI_BUS_OFF_ST 若值为 1，表明节点处于离线状态，不再响应总线上的数据传输。(RO)

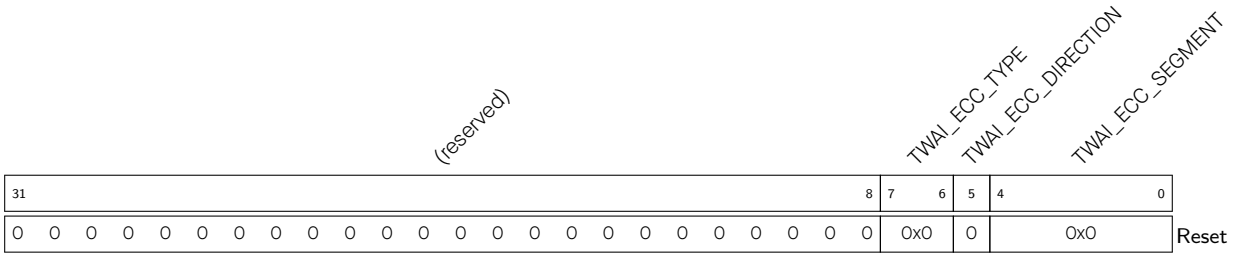
TWAI_MISS_ST 反映了从接收 FIFO 中取出数据包的完整状态。1: 当前数据包是缺失的；0：当前数据包是完整的 (RO)

Register 31.21. TWAI_ARB LOST CAP_REG (0x002C)



TWAI_ARB_LOST_CAP 记录着发送节点仲裁丢失的 bit 位置。(RO)

Register 31.22. TWAI_ERR_CODE_CAP_REG (0x0030)

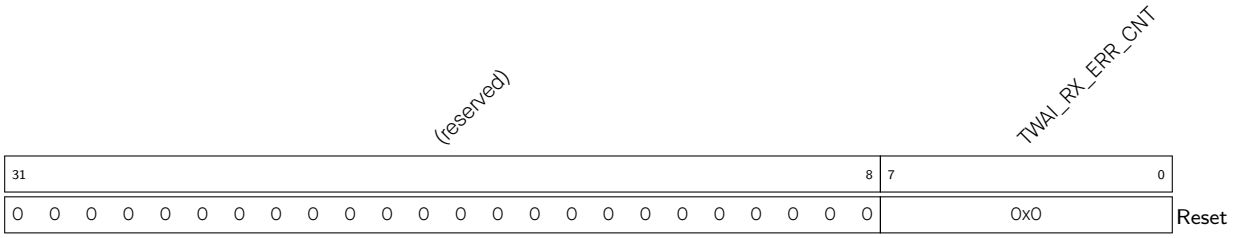


TWAI_ECC_SEGMENT 记录错误发生的位置，详见表 31.5-11。(RO)

TWAI_ECC_DIRECTION 记录错误时节点的数据传输方向。1：接收数据时发生错误；0：发送数据时发生错误 (RO)

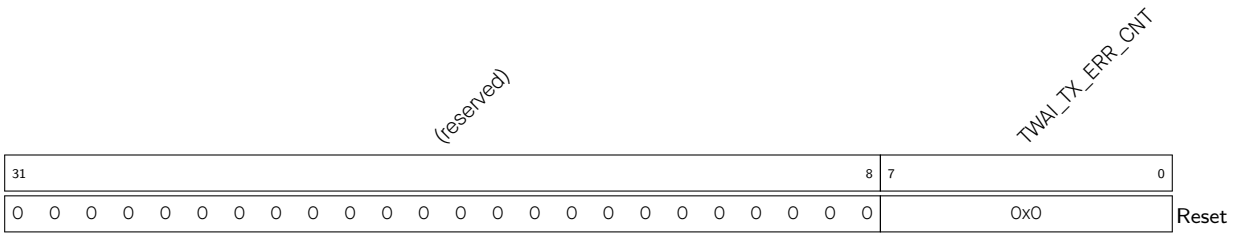
TWAI_ECC_TYPE 记录错误类别：00：位错误；01：格式错误；10：填充错误；11：其他错误 (RO)

Register 31.23. TWAI_RX_ERR_CNT_REG (0x0038)



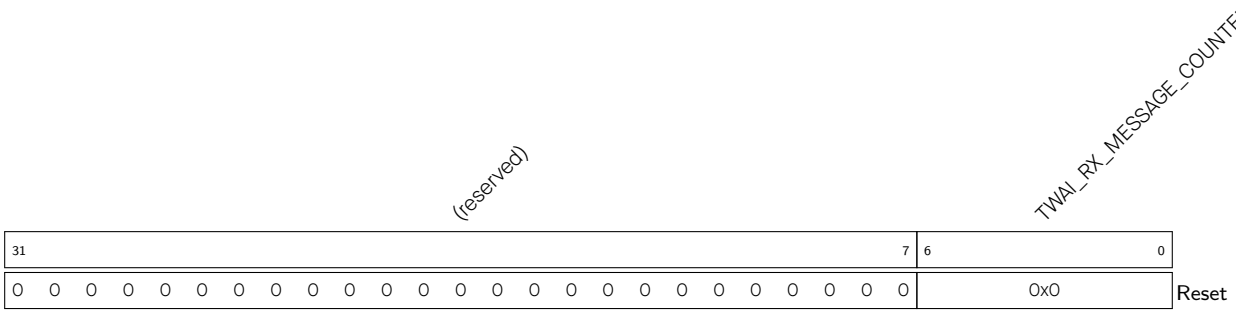
TWAI_RX_ERR_CNT 接收错误计数，数值变化发生在接收状态下。(RO | R/W)

Register 31.24. TWAI_TX_ERR_CNT_REG (0x003C)



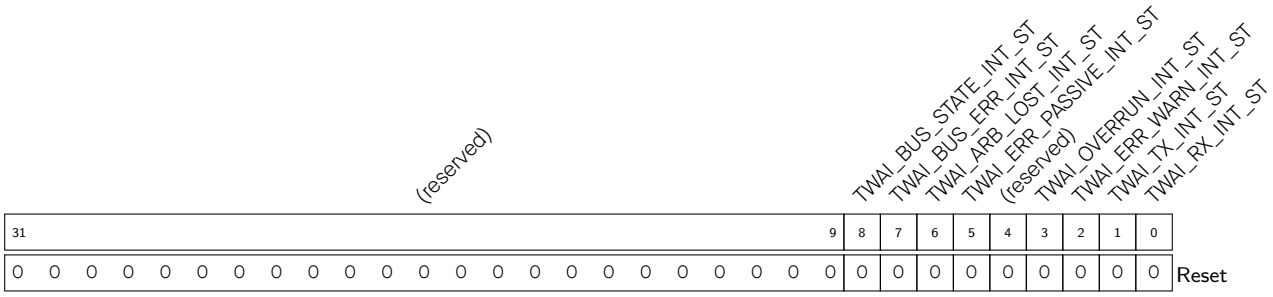
TWAI_TX_ERR_CNT 发送错误计数，数值变化发生在发送状态下。(RO | R/W)

Register 31.25. TWAI_RX_MESSAGE_CNT_REG (0x0074)



TWAI_RX_MESSAGE_COUNTER 存储着接收 FIFO 中数据包的个数。(RO)

Register 31.26. TWAI_INT_RAW_REG (0x000C)



TWAI_RX_INT_ST 接收中断。若值为 1，表明接收 FIFO 不为空，有接收数据待处理。(RO)

TWAI_TX_INT_ST 发送中断。若值为 1，表明节点数据发送任务结束，可以执行新的数据发送任务。(RO)

TWAI_ERR_WARN_INT_ST 错误报警中断。若值为 1，表明状态寄存器中错误状态信号和离线信号发生变化（0 变为 1 或 1 变为 0）。(RO)

TWAI_OVERRUN_INT_ST 数据溢出中断。若值为 1，表明节点的接收 FIFO 数据溢出。(RO)

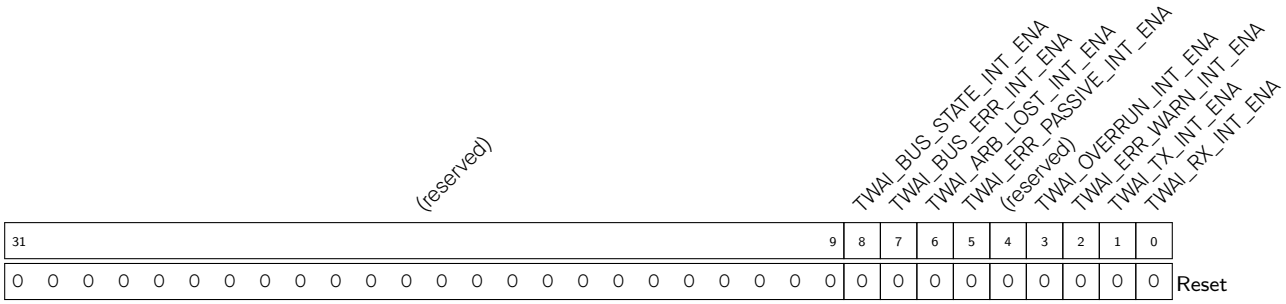
TWAI_ERR_PASSIVE_INT_ST 被动错误中断。若值为 1，表明节点由于错误计数数值的变化，在主动错误状态与被动错误状态间发生了切换。(RO)

TWAI_ARB_LOST_INT_ST 仲裁丢失中断。若值为 1，表明发送节点丢失仲裁。(RO)

TWAI_BUS_ERR_INT_ST 错误中断。若值为 1，表明节点检测到总线上发生了错误。(RO)

TWAI_BUS_STATE_INT_ST 总线状态中断。若值为 1，表明控制器状态发生了变化。(RO)

Register 31.27. TWAI_INT ENA_REG (0x0010)



TWAI_RX_INT_ENA 置 1 使能接收中断。(R/W)

TWAI_TX_INT_ENA 置 1 使能发送中断。(R/W)

TWAI_ERR_WARN_INT_ENA 置 1 使能错误报警中断。(R/W)

TWAI_OVERRUN_INT_ENA 置 1 使能数据溢出中断。(R/W)

TWAI_ERR_PASSIVE_INT_ENA 置 1 使能被动错误中断。(R/W)

TWAI_ARB_LOST_INT_ENA 置 1 使能仲裁丢失中断。(R/W)

TWAI_BUS_ERR_INT_ENA 置 1 使能总线错误中断。(R/W)

TWAI_BUS_STATE_INT_ENA 置 1 使能总线状态中断。(R/W)

第 32 章

USB OTG (USB)

32.1 概述

ESP32-S3 带有一个集成了收发器的 USB On-The-Go（下文将称为 OTG_FS）外设。该 OTG_FS 外设可配置成主机模式 (Host mode) 或设备模式 (Device mode)，完全符合 USB 2.0 协议规范。它支持传输速率为 12 Mbit/s 的全速模式 (Full-Speed, FS) 和传输速率为 1.5 Mbit/s 的低速模式 (Low-Speed, LS)，还支持主机协商协议 (Host Negotiation Protocol, HNP) 和会话请求协议 (Session Request Protocol, SRP)。

32.2 特性

32.2.1 通用特性

- 支持全速和低速速率
- 主机协商协议 (HNP) 和会话请求协议 (SRP)，均可作为 A 或 B 设备
- 动态 FIFO (DFIFO) 大小
- 支持多种存储器访问模式
  - Scatter/Gather DMA 模式
  - 缓冲 (Buffer) DMA 模式
  - Slave 模式
- 可选择集成收发器或外部收发器
- 当仅使用集成收发器时，可通过时分复用技术，和 USB Serial/JTAG 控制器共用集成收发器
- 当集成收发器和外部收发器同时投入使用时，支持 USB OTG 和 USB Serial/JTAG 控制器两外设各自挑选不同的收发器使用

32.2.2 设备模式 (Device mode) 特性

- 端点 0 永远存在（双向控制，由 EP0 IN 和 EP0 OUT 组成）
- 6 个附加端点 (1 ~ 6)，可配置为 IN 或 OUT
- 最多 5 个 IN 端点同时工作（包括 EP0 IN）
- 所有 OUT 端点共享一个 RX FIFO
- 每个 IN 端点都有专用的 TX FIFO

32.2.3 主机模式 (Host mode) 特性

- 8 个通道（管道）
  - 由 IN 与 OUT 两个通道组成的一个控制管道，因为 IN 和 OUT 必须分开处理。仅支持控制传输类型。
  - 其余 7 个管道可被配置为 IN 或 OUT，支持批量、同步、中断中的任意传输类型。
- 所有通道共用一个 RX FIFO、一个非周期性 TX FIFO、和一个周期性 TX FIFO。每个 FIFO 大小可配置。

32.3 功能描述

32.3.1 控制器内核与接口

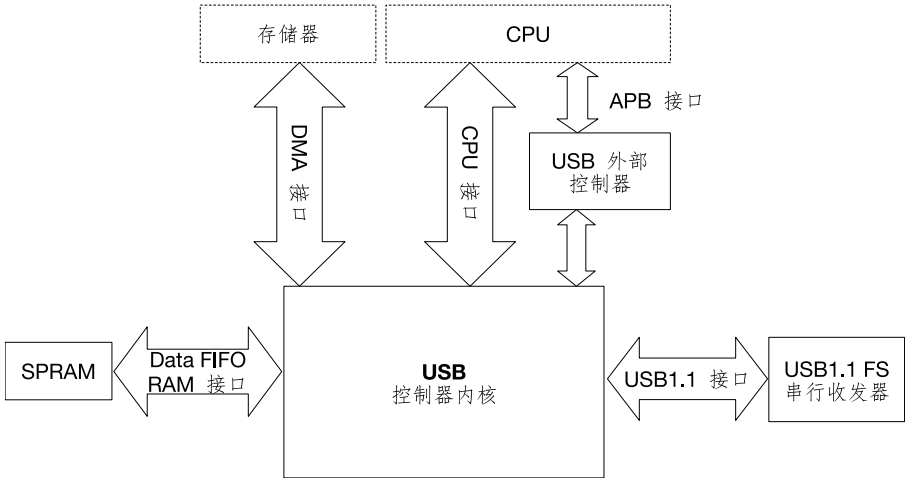


图 32.3-1. OTG_FS 系统架构

OTG_FS 外设的核心称为 USB 控制器内核。如图 32.3-1 所示，控制器内核有以下 4 个接口：

- **CPU 接口**
  CPU 可以通过该接口读写控制器内核的多个寄存器和 FIFO。该接口在内部实现为 AHB 从机接口。通过该接口访问 FIFO 的方式称为 Slave 模式。
- **APB 接口**
  CPU 可以通过 USB 外部控制器 (USB external controller) 来控制 USB 控制器内核的接口。
- **DMA 接口**
  控制器内核的内部 DMA 可以通过该接口读写系统存储器（例如在 DMA 模式下获取和写入有效数据）。该接口在内部实现为 AHB 主机接口。
- **USB 1.1 接口**
  控制器内核通过该接口连接 USB 1.1 全速串行收发器。除 USB OTG 之外，ESP32-S3 还内置一个 USB 串行/JTAG 控制器（请参阅章节 *33 USB 串口/JTAG 控制器 (USB_SERIAL_JTAG)*）。这两个 USB 控制器可通过时分复用使用内部集成收发器，或者一个控制器连接内部收发器，另一个控制器连接外部收发器。

  当只使用内部收发器时，USB OTG 和 USB 串行/JTAG 外设共用这个收发器。默认情况下，内部收发器与 USB 串行/JTAG 外设相连。当 RTC_CNTL_SW_HW_USB_PHY_SEL_CFG 为 0 时，eFuse 中的 EFUSE_USB_PHY_SEL 位决定内部收发器与哪个外设相连。若该位为 0，内部收发器与 USB 串行/JTAG 外设相连；若该位为 1，内部收发器与 USB OTG 外设相连。当 RTC_CNTL_SW_HW_USB_PHY_SEL_CFG

为 1 时，由 RTC_CNTL_SW_USB_PHY_SEL_CFG 控制内部收发器与哪个外设相连（与 EFUSE_USB_PHY_SEL 位的使用方式相同）。

当内部和外部收发器都被使用时，一个 USB 控制器选择其中一个收发器使用，另一个 USB 控制器则使用剩余的收发器。具体的地址映射信息，请参阅章节 33 *USB 串口/JTAG 控制器 (USB_SERIAL_JTAG)*。

- **USB 外部控制器**
  USB 外部控制器主要用于将 USB 2.0 全速串行接口连接到内部或外部收发器。USB 外部控制器还能实现省电模式，具体做法是控制器内核时钟（AHB 时钟）采用门控时钟，或关闭所连接的 SPRAM 时钟。需要注意的是此节能模式与通过 SRP 实现的节能方式有所不同。
- **数据 FIFO RAM 接口**
  控制器内核使用的多个 FIFO 实际上并不位于控制器内核内部，而是位于 SPRAM（单端口 RAM）上。FIFO 的大小可动态配置，因此在运行时在 SPRAM 中进行分配。CPU、DMA 或控制器通过该接口读写 FIFO。

32.3.2 存储器布局

图 32.3-2 显示了用于配置和控制 USB 控制器内核的寄存器布局。请注意，USB 外部控制器使用另外一组寄存器（称为 wrap 寄存器）。

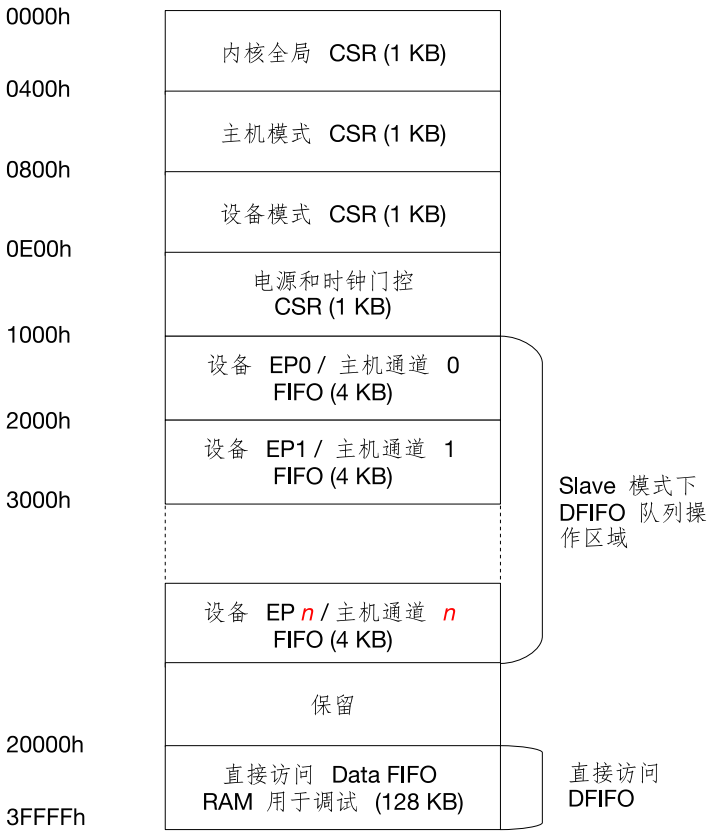


图 32.3-2. 内核地址映射

32.3.2.1 控制 & 状态寄存器 (CSR)

- **内核全局 CSR**
  内核全局寄存器用于配置/控制 OTG_FS 的通用功能（即主机和设备模式共同的功能），并表示其内部状态。通用功能包括 OTG 控制（HNP，SRP 和 A/B 设备检测），USB 配置（选择主机或设备模式，PHY 选

择），以及系统级中断。在主机和设备模式下，软件均可以对内核全局 CSR 进行访问。

- **主机模式 CSR**
  主机模式寄存器用于主机模式下的配置/控制/状态表示，只能在主机模式下被访问。每个通道各自有一组主机模式寄存器。
- **设备模式 CSR**
  设备模式寄存器用于设备模式下的配置/控制/状态表示，只能在设备模式下被访问。每个端点各自有一组设备模式寄存器。
- **电源和时钟门控寄存器**
  此单一寄存器用于控制模块电源和门控时钟。

32.3.2.2 FIFO 访问

OTG_FS 利用多个 FIFO 缓冲发送或接收的有效数据。FIFO 的数量和类型取决于主机或设备模式，以及所使用的通道或端点的数量（参考章节 32.3.3）。FIFO 访问有两种方式：DMA 模式和 Slave 模式。当使用 Slave 模式时，CPU 需要通过读写 DFIFO 的推入/弹出操作 (push/pop) 区域或读写调试区域来访问这些 FIFO。FIFO 访问遵循以下规则：

- 对 4 KB 推入/弹出区域中任何地址的读访问将在共享 RX FIFO 中产生一个弹出 (pop) 操作。
- 对特定 4 KB 推入/弹出区域的写访问将写入相应的端点或通道的 TX FIFO（前提是该端点是 IN 端点，或者该通道是 OUT 通道）。
  - 在设备模式下，数据写入相应的 IN 端点的专用 TX FIFO。
  - 在主机模式下，根据通道是非周期性通道还是周期性通道，数据写入非周期性 TX FIFO 或周期性 TX FIFO。
- 访问 128 KB 读写调试区域将直接读/写，而不是进行推入/弹出操作。此种访问通常仅用于调试目的。

请注意，仅在 Slave 模式下，才需要由 CPU 直接向 FIFO 进行数据操作。在 DMA 模式下，内部 DMA 将处理 TX FIFO 和 RX FIFO 的数据推入/弹出操作。

32.3.3 FIFO 和队列组织

OTG_FS 中的 FIFO 主要用于保存有效数据（USB 数据包的数据字段）。TX FIFO 用于存储将由主机模式下的 OUT 事务或设备模式下的 IN 事务发送的有效数据。RX FIFO 用于存储主机模式下 IN 事务或设备模式下 OUT 事务的已接收的有效数据。除了存储有效数据之外，RX FIFO 还存储每个有效数据的**状态条目**。状态条目中包含关于有效数据的信息，如：通道编号、字节数、是否有效等。在 Slave 模式下，状态条目还用于指示各类通道事件。

可用于 FIFO 分配的 SPRAM 大小为 256×35 位（35 位包括 32 个数据位加 3 个控制位）。各个通道（在主机模式下）或端点（在设备模式下）使用的多个 FIFO 被分配到 SPRAM 中，并且可以动态调整大小。

32.3.3.1 主机模式 FIFO 和队列

如图 32.3-3 所示，主机模式使用以下 FIFO：

- **非周期性 TX FIFO**：存储所有通道的批量和控制类型的 OUT 事务的有效数据。
- **周期性 TX FIFO**：存储所有通道的中断或同步类型的 OUT 事务的有效数据。

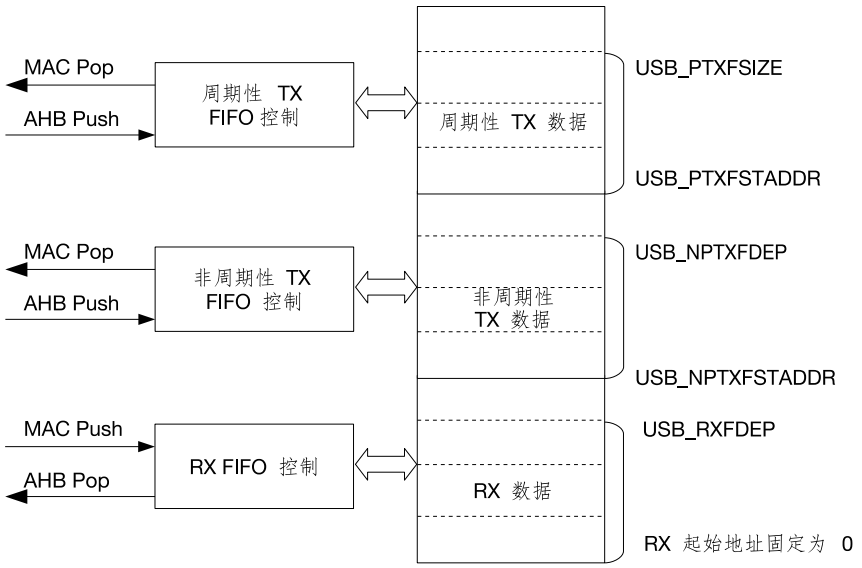


图 32.3-3. 主机模式 FIFO

- **RX FIFO**：存储所有 IN 事务的有效数据，以及用于指示有效数据大小和事务/通道事件（例如传输完成或通道暂停）的状态条目。

除 FIFO 外，主机模式还包含两个请求队列，用于存储来自多个通道的事务请求。请求队列中的每个条目都包含 IN/OUT 通道编号以及执行事务的其他信息（例如事务类型）。请求队列也用于存储其他请求类型，例如通道暂停请求。

与 FIFO 不同，请求队列的大小是固定的，且不能由软件直接访问。相反，一旦启用了通道，主机内核会自动将请求写入请求队列。请求写入队列的顺序决定了 USB 事务的处理顺序。

主机模式包含以下请求队列：

- **非周期性请求队列**：针对非周期性事务（批量和控制）的请求队列，最多可以存储 4 个条目。
- **周期性请求队列**：针对周期性事务（中断和同步）的请求队列，最多可以存储 8 个条目。

调度事务时，硬件将首先执行周期性请求队列上的所有请求，再执行非周期性请求队列上的请求。

32.3.3.2 设备模式 FIFO

如图 32.3-4 所示，设备模式使用以下 FIFO：

- **RX FIFO**：存储数据包内接收的有效数据和状态条目（用于指示有效数据的大小）。
- **专用 TX FIFO**：每个使能的 IN 端点都有一个专用的 TX FIFO，用于存储该端点的所有 IN 有效数据，而无论事务类型（周期性或非周期性 IN 事务）。

由于有专用的 FIFO，设备模式不使用任何请求队列。IN 事务的顺序由主机确定。

32.3.4 中断层次结构

OTG_FS 有一条中断线，可以通过中断矩阵连接到一个 CPU。可以通过置位 USB_GLBLINTRMSK 来显示中断信号。OTG_FS 中断是 USB_GINTSTS_REG 寄存器中所有位的或 (OR)，且置位 USB_GINTMSK_REG 寄存器中的相应位可以使能 USB_GINTSTS_REG 中的位。USB_GINTSTS_REG 包含系统级中断，还包含主机或设备模式专有的中断位以及 OTG 有关中断。OTG_FS 中断源的层次结构如图 32.3-5 所示。

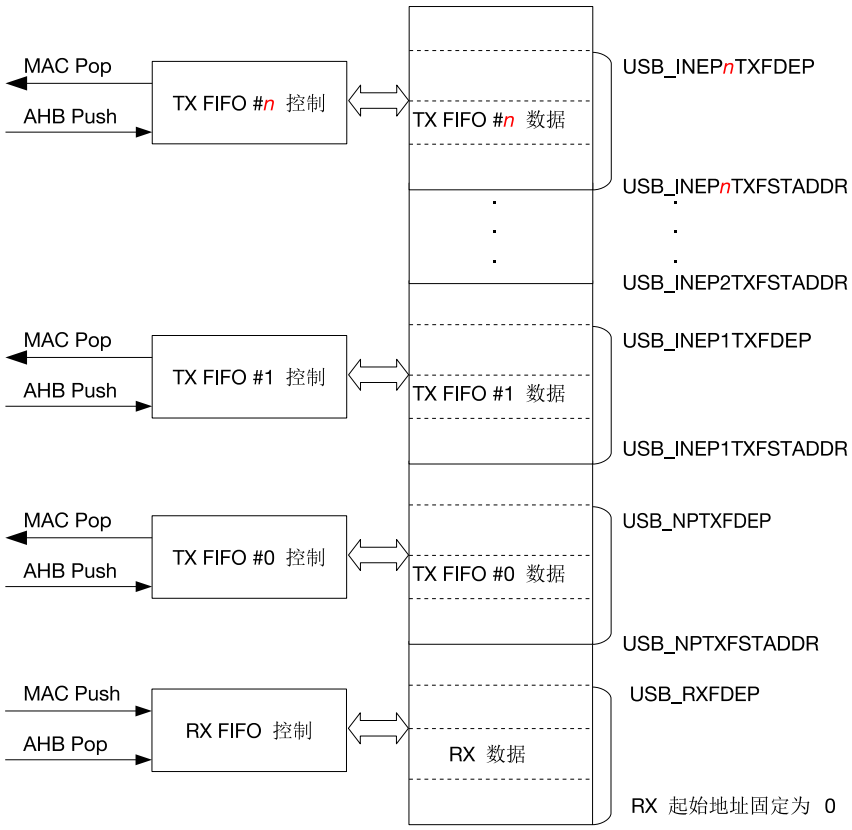


图 32.3-4. 设备模式 FIFO

USB_GINTSTS_REG 寄存器的以下位指示较低层级的中断源：

- **USB_PRTINT** 表示主机端口有未处理的中断。USB_HPRT_REG 寄存器指示中断源。
- **USB_HCHINT** 表示一个或多个主机通道有未处理的中断。通过读取 USB_HAINT_REG 寄存器可以确定哪些通道有未处理的中断，然后查询该通道的 USB_HCINT*n*_REG 寄存器以确定中断源。
- **USB_OEPINT** 表示一个或多个 OUT 端点有未处理的中断。通过读取 USB_DAINT_REG 寄存器可以确定哪些 OUT 端点有未处理的中断，然后查询 OUT 端点的 USB_DOEPINT*n*_REG 寄存器以确定中断源。
- **USB_IEPINT** 表示一个或多个 IN 端点有未处理的中断。通过读取 USB_DAINT_REG 寄存器可以确定哪些 IN 端点有未处理的中断，然后查询 IN 端点的 USB_DIEPINT*n*_REG 寄存器以确定中断源。
- **USB_OTGINT** 表示 OTG 事件已触发中断。查询 USB_GOTGINT_REG 寄存器以确定哪些 OTG 事件触发了中断。

32.3.5 DMA 模式和 Slave 模式

USB OTG 支持 3 种存储器访问方式：Scatter/Gather DMA 模式、缓冲 DMA 模式，和 Slave 模式。

32.3.5.1 Slave 模式

在 Slave 模式下，所有有效数据放入 FIFO 或从 FIFO 中取出都必须通过 CPU 进行。

- 使用 IN 端点或 OUT 通道传输数据包时，必须将有效数据放入相应的端点或通道的 TX FIFO 中。
- 接收到数据包时，必须先通过读取 USB_GRXSTSP_REG 从 RX FIFO 中取出数据包的状态条目，以确定数

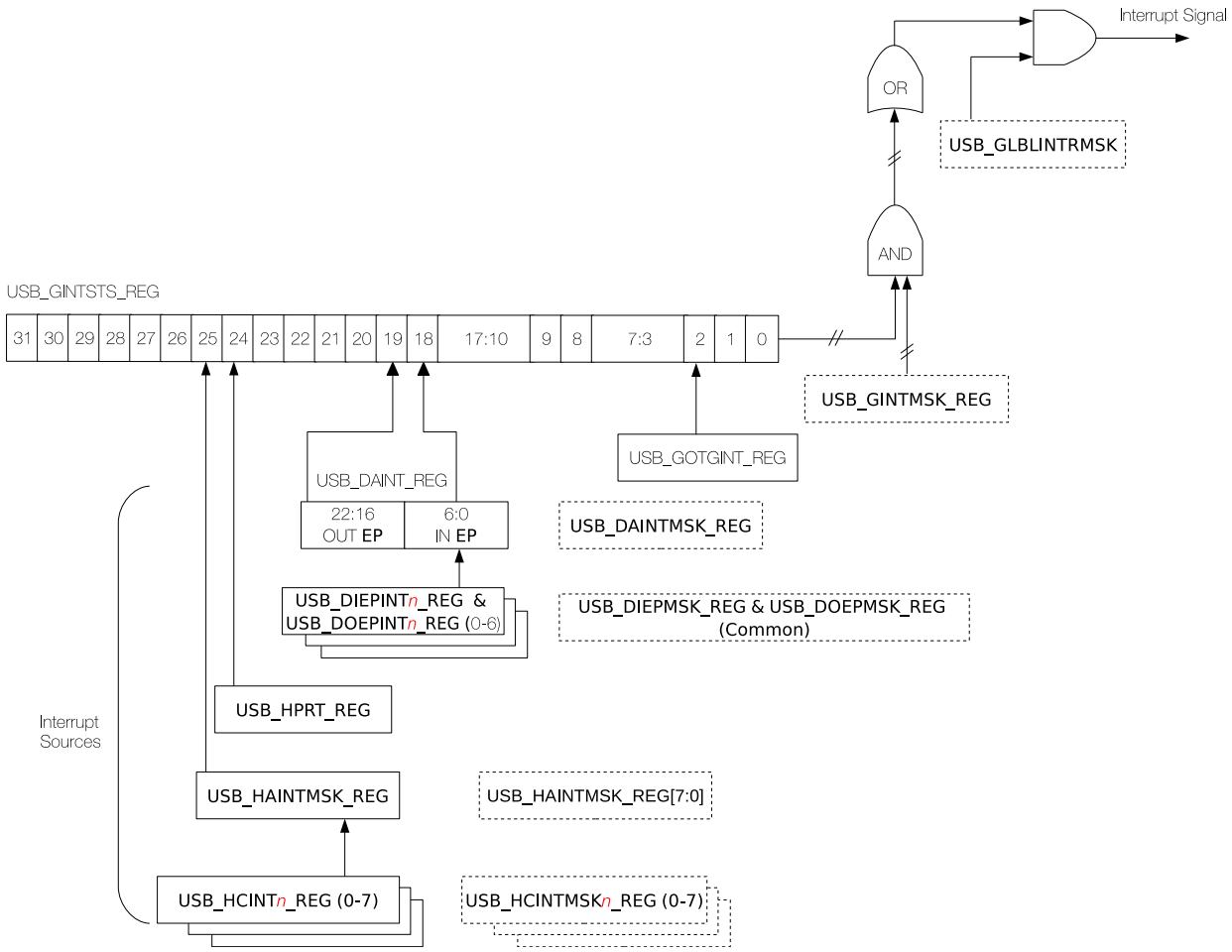


图 32.3-5. OTG_FS 中断层次结构图

据包中有效数据的长度（以字节为单位）。然后必须由 CPU 手动从 RX FIFO 中取出相应的字节数（通过读取 RX FIFO 中的存储区域）。

32.3.5.2 缓冲 DMA 模式

缓冲模式类似于 Slave 模式，不同之处在于该模式为利用内部 DMA 将有效数据放入 FIFO 或从 FIFO 中取出。

- 使用 IN 端点或 OUT 通道传输数据包时，应将有效数据的存储地址写入 USB_HCDMA*n*_REG（主机模式）或 USB_DOEPDMA*n*_REG（设备模式）寄存器。启用端点或通道后，内部 DMA 会将有效数据从存储器中推送到通道或端点的 TX FIFO 中。
- 使用 OUT 端点或 IN 通道接收数据包时，应将存储器中空缓冲区的地址写入 USB_HCDMA*n*_REG（主机模式）或 USB_DOEPDMA*n*_REG（设备模式）寄存器。启用端点或通道后，内部 DMA 将把有效数据从 RX FIFO 弹出到相应缓冲区中。

32.3.5.3 Scatter/Gather DMA 模式

在 Scatter/Gather DMA 模式下，包含有效数据的接收缓冲区可能分散在整个存储器各处。每个端点或通道都有一个连续的 DMA 描述符列表。每个描述符包含一个指向有效数据或接收缓冲区的 32 位指针和一个 32 位的缓冲区描述符 (BufferStatus Quadlet)。有效数据和接收缓冲区可以对应单个事务（即 < 1 MPS 字节，MPS：maximum packet size）或整个传输（即 > 1 MPS 字节）。该列表的实现为环形缓冲区，意味着 DMA 在遇到列表

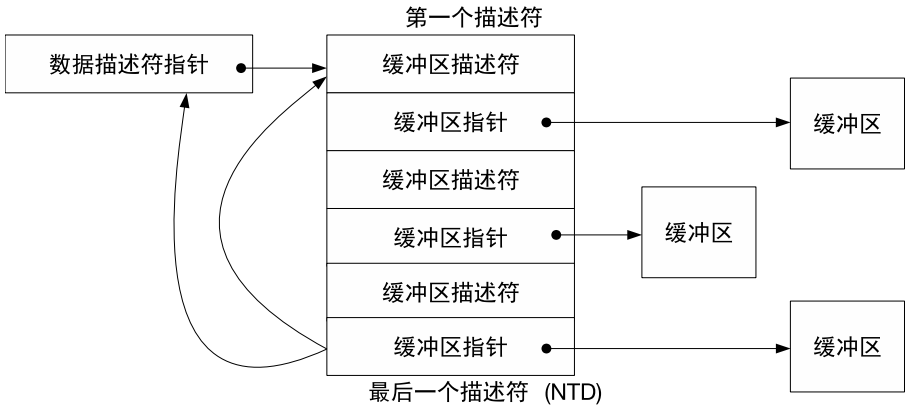


图 32.3-6. Scatter/Gather DMA 链表结构

中的最后一个条目时将返回至第一个条目。

- 当使用 IN 端点或 OUT 通道发送传输/事务时，DMA 将从多个缓冲区收集有效数据并将其放入 TX FIFO。
- 当使用 OUT 端点或 IN 通道接收传输/事务时，DMA 将取出来自 RX FIFO 的接收有效数据，并将它们分别存放到 DMA 列表条目所指向的多个缓冲区中。

32.3.6 事务和传输级操作

在主机或设备模式下，通信可以在事务级或传输级进行。

32.3.6.1 DMA 模式下的事务和传输级操作

在 DMA 模式下的传输级操作，只有当一个传输通道被终止时，中断才会发生。以下三种情况下发生传输通道终止：传输通道中所有要求传送的数据全部成功传送、接收到 STALL 或出现连续事物级错误（如，3 个连续的事务级错误）。在 DMA 设备模式下操作时，所有错误均由控制器内核进行处理。

在 DMA 模式下的事务级操作，传输大小是一个数据包的大小（最大数据包大小或短数据包大小）。

32.3.6.2 Slave 模式下的事务和传输级操作

在 Slave 模式下的事务级操作，一次只能处理一个事务。每组有效数据应对应一个数据包，软件必须根据 USB 接收到的握手应答（例如 ACK 或 NAK）来确定是否需要重启事务。

下表描述了 Slave 模式下进行 IN 和 OUT 事务级操作的方法。

表 32.3-1. Slave 模式下的 IN 和 OUT 事务级操作

主机模式	设备模式
OUT 事务	
1. 软件配置 USB_HCTSIZ*n*_REG 寄存器，指定数据包的大小和数量 (1 个)，使能该通道，然后将数据包的有效数据复制到 TX FIFO 中。 2. 软件在写完每个数据包的最后一个 DWORD 之后，控制器内核将自动把请求条目写入相应的请求队列。 3. 如果该事务成功，将生成 USB_XFERCOMPL 中断。如果该事务失败，则会发生错误中断 (例如 USB_H_NACK*n*)。	1. 软件配置 USB_DIEPTSIZ*n*_REG 寄存器，指定数据包的大小 (1 MPS) 和数量 (1 个)。端点使能后，将等待主机向其发送数据包。 2. 接收到的数据包将与数据包状态条目一起放入 RX FIFO。 3. 如果该事务失败 (例如，由于 RX FIFO 已满)，则端点将回传 NAK。
IN 事务	
1. 软件配置 USB_HCTSIZ*n*_REG 寄存器，指定数据包的大小和数量 (1 个)，然后使能该通道。 2. 控制器内核自动将请求条目写入相应的请求队列。 3. 如果该事务成功，接收数据以及状态条目将写入 RX FIFO。否则，会产生错误中断 (例如 USB_H_NACK*n*)。	1. 软件配置 USB_DIEPTSIZ*n*_REG 寄存器，指定数据包的大小和数量 (1 个)。端点使能后，将等待主机从其读取数据包。 2. 数据包传输完成后，将产生 USB_XFERCOMPL 中断。

在 Slave 模式下进行传输级操作时，可以在队列中一次性排入一个或多个事务级操作，达到类似于 DMA 模式下的传输级操作的效果。在同一次触发的传输中，多个事务的数据包均可以从 FIFO 中读写，这样就无需以数据包为单位触发中断。

Slave 模式下进行传输级操作的方法类似于事务级操作，不同之处在于，需要配置 USB_HCTSIZ*n*_REG 或
USB_DOEPTSIZ*n*_REG 寄存器以指定整次传输的大小和数据包个数。使能通道或端点后，应分别向 TX FIFO 或 RX FIFO 写入或读取对应于多个数据包的有效数据（假设有足够的空间或足够的数据量）。

32.4 OTG

USB OTG 允许 OTG 设备作为 USB 主机或 USB 设备。因此，OTG 设备上一般都有一个 Mini-AB 或 Micro-AB 接口，可用于连接 A-plug 或 B-plug。当 OTG 设备连接上 A-plug/B-plug 时，其将成为 A 设备/B 设备。

- A 设备默认为主机模式（A 主机），B 设备默认为设备模式（B 外设）。
- 通过使用主机协商协议 (NHP)，A、B 设备可互相交换角色，即变更为 A 外设和 B 主机。
- A 设备可关闭 Vbus 省电。然后，B 设备可通过请求 A 设备启动 Vbus 并发起一个新的会话来唤醒 A 设备。该机制称为会话请求协议 (SRP)。
- Vbus 只能由 A 设备供电，即使 A 设备为外设模式。

OTG 设备可通过接头的 ID 管脚确定其连接的是 A-plug 还是 B-plug。A-plug 中的 ID 管脚为接地，B-plug 中的 ID 管脚则为悬空。

32.4.1 OTG 接口

OTG_FS 支持 OTG Revision 1.3 规范的 SRP 和 HNP 协议。OTG_FS 控制器内核通过 UTMI+ OTG 接口与收发器（内部或外部）连接。UTMI+ OTG 接口允许控制器内核操作收发器（比如启用/禁用 HNP 中的上拉和下拉）以实现 OTG 的功能，并且还允许收发器指示与 OTG 相关的事件。如果改用外部收发器，那么 UTMI+ OTG 将通过 GPIO 交换矩阵连接到 ESP32-S3 的 GPIO，请参阅章节 *6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)*。表 32.4-1 描述了 UTMI+ OTG 接口信号。

表 32.4-1. UTMI OTG 接口

接口信号	I/O	描述
usb_otg_iddig_in	I	迷你 A/B 插头指示器。指示所连接的插头是 mini-A 还是 mini-B。仅在 usb_otg_idpullup 被采样断言时有效。 1'b0：连接 mini-A 1'b1：连接 mini-B
usb_otg_avalid_in	I	A 类外设会话有效。指示 Vbus 电压是否在 A 类外设会话的有效电平上。比较器阈值为： 1'b0: Vbus <0.8 V 1'b1: Vbus > 2 V
usb_otg_bvalid_in	I	B 类外设会话有效。指示 Vbus 电压是否在 B 类外设会话的有效电平上。比较器阈值为： 1'b0: Vbus <0.8 V 1'b1: Vbus > 4 V
usb_otg_vbusvalid_in	I	Vbus 有效。指示 Vbus 电压是否在 A/B 设备/外设操作的有效电平上。比较器阈值为： 1'b0: Vbus <4.4 V 1'b1: Vbus >4.75 V
usb_srp_sessend_in	I	B 设备会话结束。指示 Vbus 电压是否在 B 设备会话结束的阈值以下。比较器阈值为： 1'b0: Vbus >0.8 V 1'b1: Vbus <0.2 V
usb_otg_idpullup	O	模拟 ID 输入采样使能。使能采样模拟 ID 线。 1'b0: ID 管脚采样禁能 1'b1: ID 管脚采样使能
usb_otg_dppulldown	O	D+ 下拉电阻使能。使能 D+ 线上的 15 kΩ 下拉电阻。
usb_otg_dmpulldown	O	D- 下拉电阻使能。使能 D- 线上的 15 kΩ 下拉电阻。
usb_otg_drvvbus	O	驱动 Vbus。驱动 Vbus 到 5 V。 1'b0：不驱动 Vbus 1'b1：驱动 Vbus
usb_srp_chrgvbus	O	Vbus 输入充电使能。指示 PHY 为 Vbus 充电。 1'b0：不通过电阻为 Vbus 充电 1'b1：通过电阻为 Vbus 充电（需激活至少 30 ms）

接口信号	I/O	描述
usb_srp_dischrgvbus	O	Vbus 输入放电使能。指示 PHY 为 Vbus 放电。 1'b0：不通过电阻为 Vbus 放电 1'b1：通过电阻为 Vbus 放电（需激活至少 50 ms）

32.4.2 ID 管脚检测

寄存器 USB_GOTGCTL_REG 中的 USB_CONIDSTS 位指示 OTG 控制器为 A 设备 (1'b0) 还是 B 设备 (1'b1)。当 USB_CONIDSTS 发生改变（即连接或断开插头时），会产生 USB_CONIDSTSCHNG 中断。

32.4.3 会话请求协议 (SRP)

32.4.3.1 A 设备 SRP

图 32.4-1 说明了 OTG_FS 充当 A 设备（即默认主机并为 Vbus 供电）时的 SRP 流程。

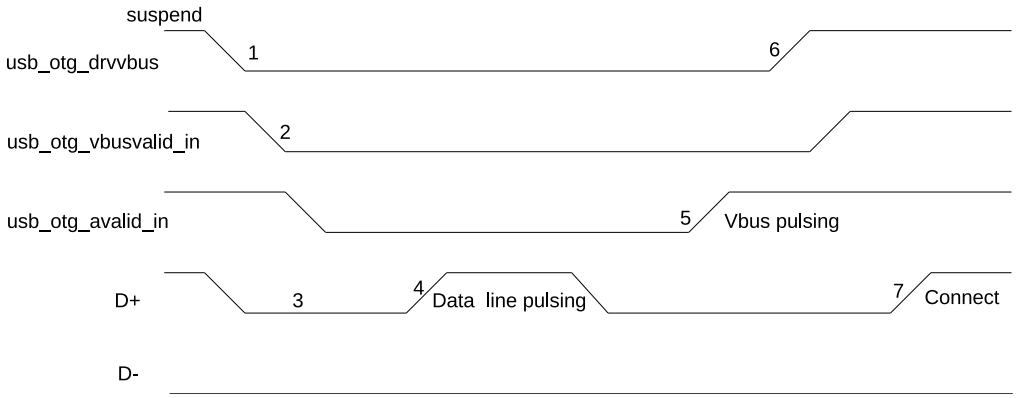


图 32.4-1. A 设备 SRP

1. 为了节省电能，当总线空闲时，应用程序将挂起并关闭端口电源，方法是写入主机端口控制和状态寄存器中的端口挂起位（USB_PRTSUSP 置为 1'b0）和端口电源位（USB_PRTPWR 置为 1'b0）。
2. PHY 通过使 usb_otg_vbusvalid_in 信号无效来指示端口断电。
3. 当 Vbus 电源关闭时，A 设备必须检测到 SEO 至少 2 ms 才能启动 SRP。
4. 要启动 SRP，B 设备会打开其数据线上拉电阻 5 到 10 ms。OTG_FS 内核将检测数据线脉冲。
5. 设备将 Vbus 驱动到 A 设备会话有效阈值之上（至少 2.0 V）以执行 Vbus 脉冲。OTF_FS 内核在检测 SRP 时中断应用程序。全局中断状态寄存器中的会话请求检测位 (USB_SESSREQINT) 将被置位。
6. 应用程序必须处理会话请求检测中断，并通过写入主机端口控制和状态寄存器中的端口电源位来打开端口电源位。PHY 通过确认 usb_otg_vbusvalid_in 信号来指示端口上电。
7. 当 USB 上电时，B 设备连接，完成 SRP 过程。

32.4.3.2 B 设备 SRP

图 32.4-2 说明了 OTG_FS 充当 B 设备（即不为 Vbus 供电）时的 SRP 流程。

1. 为了节省电能，当总线空闲时，主机（A 设备）将挂起并关闭端口电源。PHY 通过使 usb_otg_vbusvalid_in 信号无效来指示端口掉电。在检测到 3 ms 总线空闲后，OTG_FS 内核将内核中断

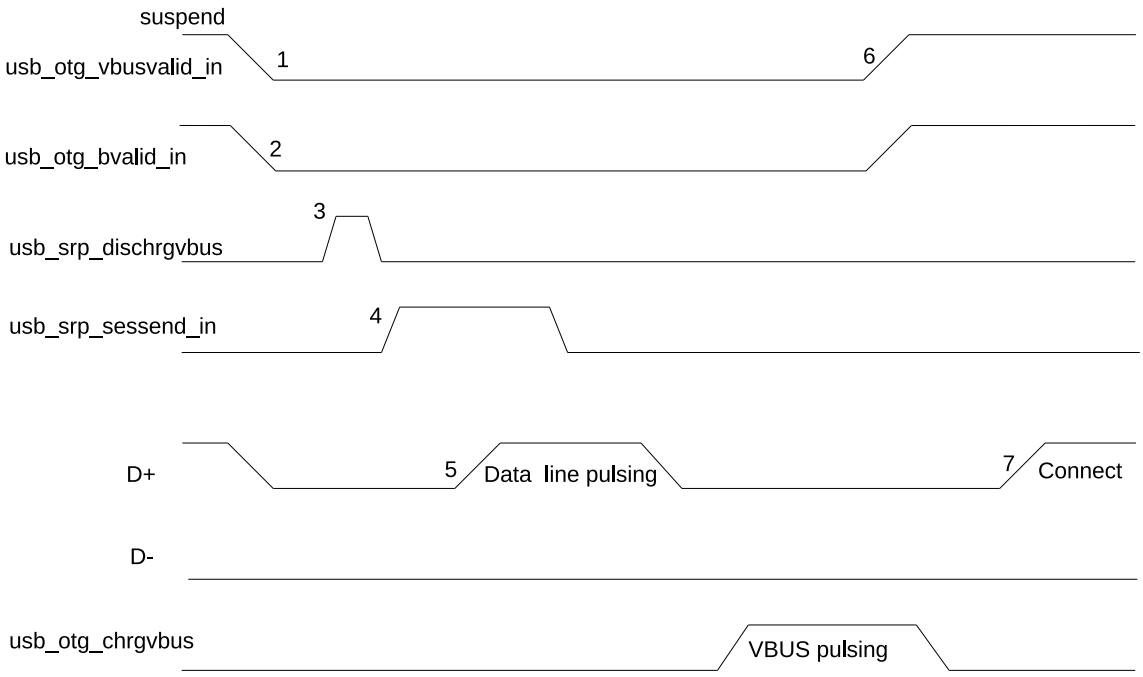


图 32.4-2. B 设备 SRP

寄存器中的早期挂起位（USB_ERLYSUSP 中断）置 1。之后，OTF_FS 内核将内核中断寄存器中的 USB 挂起位 (USB_USBSUSP) 置 1。PHY 通过使 usb_otg_bvalid_in 信号无效来指示 B 设备会话结束。

2. OTG_FS 内核确认 usb_otg_dischrgvbus 信号，指示 PHY 加快 Vbus 放电。
3. PHY 通过确认 usb_otg_sessend_in 信号来指示会话结束。这是 SRP 的初始条件。OTG_FS 内核在启动 SRP 之前需要检测到 SEO 2 ms。对于 USB 2.0 全速串行收发器，在 USB_BSESVLD 无效后，应用程序必须等待 Vbus 放电至 0.2 V。
4. 应用程序等待 1.5 秒（TB_SEO_SRP 时间），然后写入 OTG 控制和状态寄存器中的会话请求位 (USB_SESREQ) 并启动 SRP。OTG_FS 内核执行数据线脉冲，然后执行 Vbus 脉冲。
5. 主机（A 设备）从数据线或 Vbus 脉冲检测到 SRP，然后打开 Vbus。PHY 确认 usb_otg_vbusvalid_in 信号指示 Vbus 上电。
6. OTG_FS 内核确认 usb_srp_chrgvbus 并执行 Vbus 脉冲。主机（A 设备）打开 Vbus，启动新会话，指示 SRP 成功。OTG_FS 内核通过置位 OTG 中断状态寄存器中的会话请求成功状态改变位 (USB_SESREQSC) 来中断应用程序。应用程序读取 OTG 控制和状态寄存器中的会话请求成功位。
7. 当 USB 通电时，OTG_FS 内核连接，从而完成 SRP 过程。

32.4.4 主机协商协议 (HNP)

32.4.4.1 A 设备 HNP

图 32.4-3 说明了 OTG_FS 充当 A 设备时的 HNP 流程。

1. OTG_FS 内核向 B 设备发送 SetFeature b_hnp_enable 描述符以启用 HNP 支持。B 设备回复 ACK 则表明其支持 HNP。应用程序必须置位 OTG 控制和状态寄存器中的主机设置 HNP 使能位 (USB_HSTSETHNPEN) 向 OTG_FS 内核说明 B 设备支持 HNP。

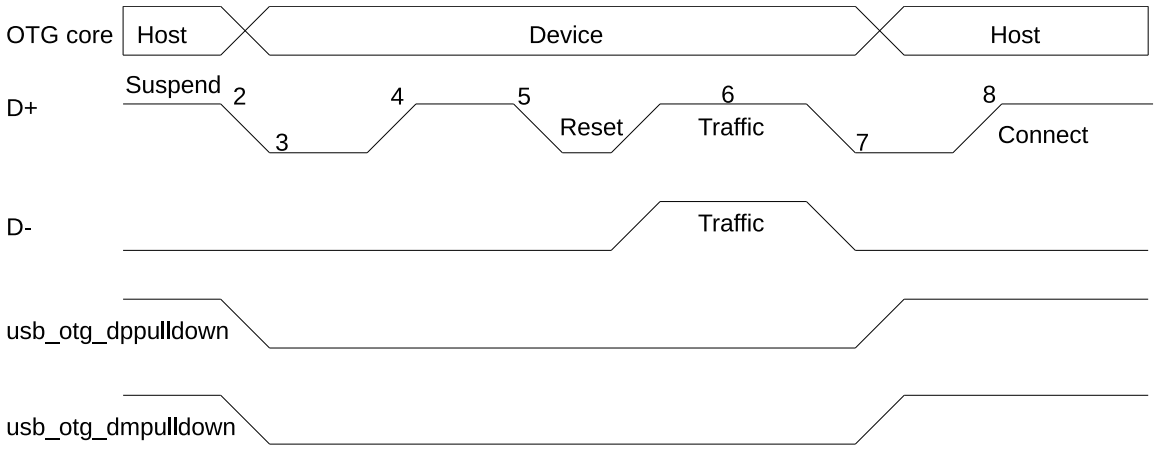


图 32.4-3. A 设备 HNP

2. 使用完总线后，应用程序写入主机端口控制和状态寄存器中的端口挂起位 (USB_PRTSUSP) 进入挂起状态。
3. 当 B 设备观察到 USB 挂起时，它将断开连接，表明 HNP 的初始状态。B 设备仅在必须切换到主机角色时才启动 HNP；否则，总线将继续挂起。OTG_FS 内核在 OTG 中断状态寄存器中置位主机协商中断位 (USB_HSTNEGDET)，指示 HNP 的开始。OTG_FS 内核将 usb_otg_dppulldown 和 usb_otg_dmpulldown 信号置为无效，指示设备角色。PHY 启用 D+ 上拉电阻，指示 B 设备的连接。应用程序必须读取 OTG 控制和状态寄存器中的当前模式位 (USB_CURMOD_INT) 以确定设备模式。
4. B 设备检测到连接，发出 USB 复位，对 OTG_FS 内核进行枚举以开始数据通信。
5. B 设备继续充当主机角色，启动数据通信，并在完成后挂起总线。OTG_FS 内核在检测到 3 ms 总线空闲之后，将内核中断寄存器中的早期挂起位 (USB_ERLYSUSP) 置 1。之后，OTG_FS 内核将内核中断寄存器中的 USB 挂起位 (USB_USBSUSP) 置 1。
6. 在协商模式下，OTG_FS 内核检测到挂起，断开连接并切换回主机角色。OTG_FS 内核确认 usb_otg_dppulldown 和 usb_otg_dmpulldown 信号，以表明其承担了主机角色。
7. OTG_FS 内核将 OTG 中断状态寄存器中的连接器 ID 状态改变中断 (USB_CONIDSTS) 置位。应用程序必须读取 OTG 控制和状态寄存器中的连接器 ID 状态，以确定 OTG_FS 内核作为 A 设备。这表明该应用程序已完成 HNP。应用程序必须读取 OTG 控制和状态寄存器中的当前模式位，以确定主机模式操作。
8. B 设备连接，完成 HNP 过程。

32.4.4.2 B 设备 HNP

图 32.4-4 说明了 OTG_FS 充当 B 设备时的 HNP 流程。

1. A 设备发送 SetFeature b_hnp_enable 描述符以启用 HNP 支持。OTG_FS 内核回复 ACK 响应以表明其支持 HNP。应用程序必须将 OTG 控制和状态寄存器中的设备 HNP 使能位 (USB_DEVHNPEN) 置 1，以表明支持 HNP。应用程序将 OTG 控制和状态寄存器中的 HNP 请求位 (USB_DEVHNPEN) 置 1，以指示 OTG_FS 内核启动 HNP。
2. A 设备使用完总线后，将挂起总线。
   (a) OTG_FS 内核在总线空闲 3 ms 之后将内核中断寄存器中的早期挂起位 (USB_ERLYSUSP) 置 1。之后，OTG_FS 内核将内核中断寄存器中的 USB 挂起位 (USB_USBSUSP) 置 1。OTG_FS 内核断开连接，并且 A 设备检测到总线上的 SE0，指示 HNP。

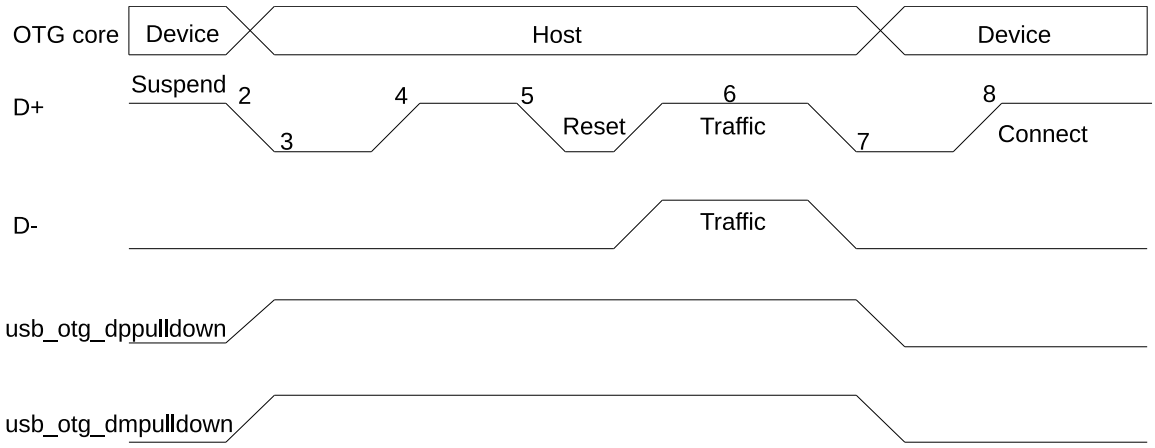


图 32.4-4. B 设备 HNP

(b) OTG_FS 内核确认 usb_otg_dppulldown 和 usb_otg_dmpulldown 信号，以表明其承担了主机角色。

(c) A 设备通过在检测到 SEO 的 3 ms 内激活 D+ 上拉电阻来做出响应。OTG_FS 内核将检测到连接。

(d) OTG_FS 内核将 OTG 中断状态寄存器中的主机协商成功状态改变位 (USB_CONIDSTS) 置 1，以指示 HNP 状态。应用程序必须读取 OTG 控制和状态寄存器中的主机协商成功位 (USB_HSTNEGSCS)，才能确定主机协商成功。应用程序必须读取内核中断寄存器中的当前模式位 (USB_CURMOD_INT)，才能确定主机模式操作。

3. 将 USB_PRTPWR 位设置为 1'b1，以驱动 USB 上的 Vbus。
4. 等待 USB_PRTCONNDET 中断。这表明设备已连接到端口。
5. 应用程序将重置位 (USB_PRTRST) 置为 1，OTG_FS 内核将发出 USB 重置，对 A 设备进行枚举并开始数据通信。
6. 等待 USB_PRTENCHNG 中断。
7. OTG_FS 内核继续充当启动数据通信的主机角色，完成后，通过写入主机端口控制和状态寄存器中的端口挂起位 (USB_PRTSUSP) 将总线挂起。
8. 在协商模式下，当 A 设备检测到挂起时，它将断开连接并切换回主机角色。OTG_FS 内核将 usb_otg_dppulldown 和 usb_otg_dmpulldown 信号置为无效，以指示承担设备角色。
9. 应用程序必须读取内核中断寄存器中的当前模式位 (USB_CURMOD_INT)，以确定主机模式操作。
10. OTG_FS 内核连接，完成 HNP 过程。

32.5 寄存器

USB OTG 寄存器列表及详细描述受 IP 供应商保密协议 (NDA) 的约束。若需获取某一寄存器的支持信息，请通过 Technical Inquires 联系乐鑫技术支持团队。

第 33 章

USB 串口/JTAG 控制器 (USB_SERIAL_JTAG)

ESP32-S3 中包含一个 USB 串口/JTAG 控制器，可用于烧录芯片的 flash、读取程序输出的数据以及将调试器连接到正在运行的程序中。任何带有 USB 主机（下文将简称为“主机”）的计算机都可以实现上述功能，无需其他外部组件辅助。

33.1 概述

开发 ESP32-S2 及之前版本的芯片通常使用两种通信方式：串口通信和 JTAG 调试端口通信。串口是一个双线接口，传统上用于将开发中的新固件烧录到 ESP32 上。由于大多数现代计算机上已没有兼容的串口，因此需要一个 USB 转串口集成电路或开发板来解决这一问题。固件烧录完成后，该端口即被用于监视程序中的调试输出数据，从而关注程序运行的总体状态。当程序运行中出现异常情况（程序崩溃）时，需使用 JTAG 调试端口检查程序及其变量的状态，并设置断点和观察点。此时便需要利用一个外部 JTAG 适配器使 SoC 与 JTAG 调试端口建立连接。

上述外部接口共需占用 6 个管脚，且在调试过程中，这些管脚便不能用于其他功能。然而，对于 ESP32-S3 这种小封装的设备，不能使用上述管脚会限制其设计。

为解决这一问题，同时尽可能减少对外部设备的需求，ESP32-S3 中包含了一个 USB 串口/JTAG 控制器，同时集成 USB-串口转换器和 USB-JTAG 适配器功能。由于该模块仅使用 USB Specification 2.0 所需的两条数据线直接连接外部 USB 主机，因此 ESP32-S3 仅需占用 2 个管脚用于调试。

33.2 特性

- USB 全速标准
- 可配置为使用 ESP32-S3 内部 USB PHY 或通过 GPIO 交换矩阵使用外部 PHY
- 固定功能。包含连接的 CDC-ACM（通信设备类抽象控制模型）和 JTAG 适配器功能
- 共 2 个 OUT 端点、3 个 IN 端点和 1 个控制端点 EP_0，可实现最大 64 字节的数据载荷
- 有内部 PHY，基本无需其他外部组件连接主机计算机
- CDC-ACM 的虚拟串行功能在大多数现代操作系统上可实现即插即用
- JTAG 接口可使用紧凑的 JTAG 指令实现与 CPU 调试内核的快速通信
- CDC-ACM 支持主机控制芯片复位和进入下载模式

如图 33.2-1 所示，USB 串口/JTAG 控制器包含一个 USB PHY、USB 设备接口、JTAG 命令处理器、响应捕捉单元以及若干个 CDC_ACM 寄存器。PHY 和部分 USB 设备接口使用主 PLL 时钟产生的 48 MHz 时钟作为时钟源，除此以外的其他部分则使用 APB_CLK 作为时钟源。JTAG 命令处理器与 ESP32-S3 主处理器中的 JTAG 调试单元相连；CDC-ACM 寄存器则连接至 APB 总线，主 CPU 上运行的软件可对其进行读写访问。

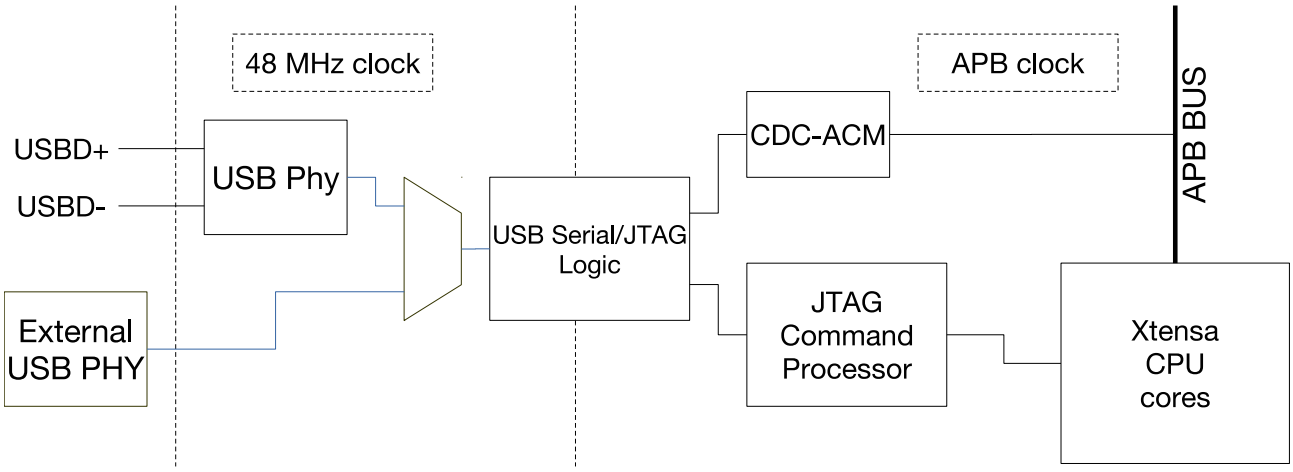


图 33.2-1. USB Serial/JTAG 高层框图

请注意，USB 串口/JTAG 控制器为 USB 2.0 全速标准 (12 Mbps)，不支持 USB 2.0 标准的其他模式（如，480 Mbps 的高速模式）。

图 33.2-2 显示了 USB 串口/JTAG 控制器中 USB 部分的内部详细信息。USB 串口/JTAG 控制器由一个 USB 2.0 全速设备组成，包含 1 个控制端点、1 个虚拟中断端点、2 个批量输入端点和 2 个批量输出端点。这些端点共同组成了该复合型 USB 设备，具体可分为 CDC-ACM USB 设备和实现 JTAG 接口功能的供应商特定设备。JTAG 接口直接与芯片的 Xtensa CPU 的调试接口相连，可对运行在该 CPU 上的程序进行调试。同时，CDC-ACM 中包含一组寄存器，CPU 上运行的程序可对其进行读写操作。此外，芯片上的 ROM 启动代码可允许用户通过使用该接口重新烧录 flash。

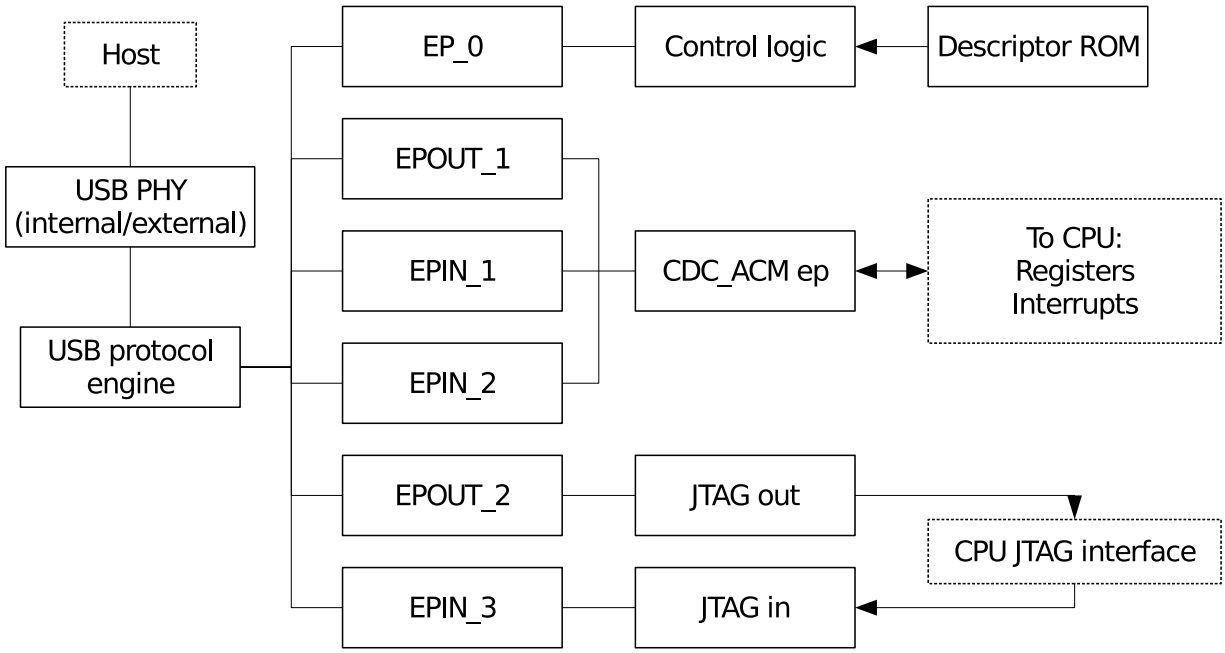


图 33.2-2. USB Serial/JTAG 框图

33.3 功能描述

USB 串口/JTAG 控制器一边与 USB 主机处理器连接，另一边与 CPU 调试硬件以及在 USB 端口上运行的软件连接。

33.3.1 USB 串口/JTAG 主机连接

USB 串口/JTAG 主机控制器通过 PHY 实现与 USB 主机物理层面的连接。ESP32-S3 有一个内部 PHY，由 USB-OTG 和 USB 串口/JTAG 硬件共用，二者都可使用该内部 PHY。另外，不使用内部 PHY 的信号单元可以通过 GPIO 交换矩阵连接到外部 IO 焊盘上。在这些焊盘上添加一个外部 USB PHY，即可产生另一个可用的 USB 端口。

使用 eFuse 可决定 USB 串口/JTAG 控制器和 USB-OTG 与内部 PHY 或外部 PHY 的连接方式，如表 33.4-1 所示。之后，可通过配置寄存器改变连接方式。

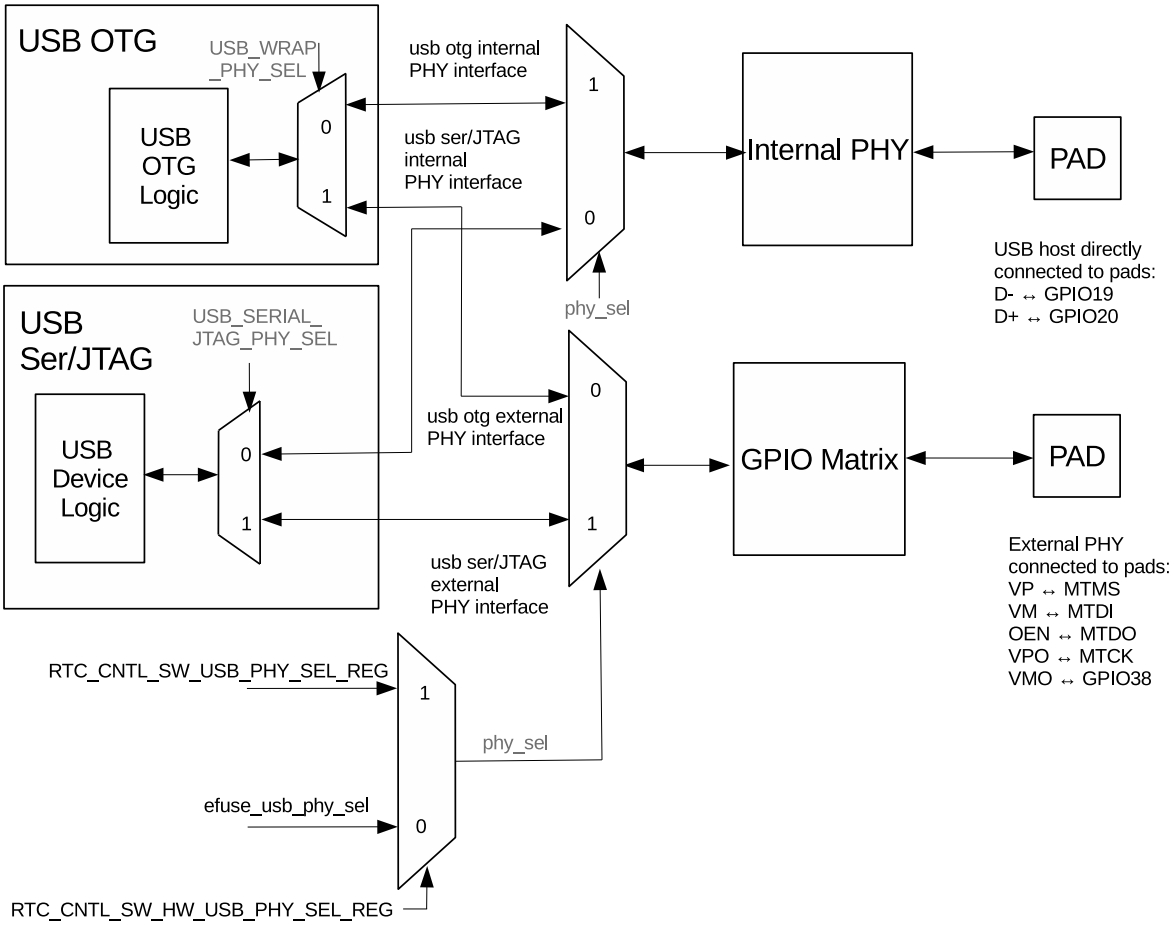


图 33.3-1. USB 串口/JTAG 与 USB-OTG 内部/外部 PHY 连接图

CPU 上的 JTAG 信号可通过 eFuse 传输到 USB 串口/JTAG 控制器或外部 GPIO 焊盘，也可以通过软件控制实现该信号传输。此时，USB 串口/JTAG 上的信号也可以传输到 GPIO 交换矩阵上。这样，就可以实现使用 ESP32-S3 的 USB 串口/JTAG 控制器模块通过 JTAG 调试另一块芯片。

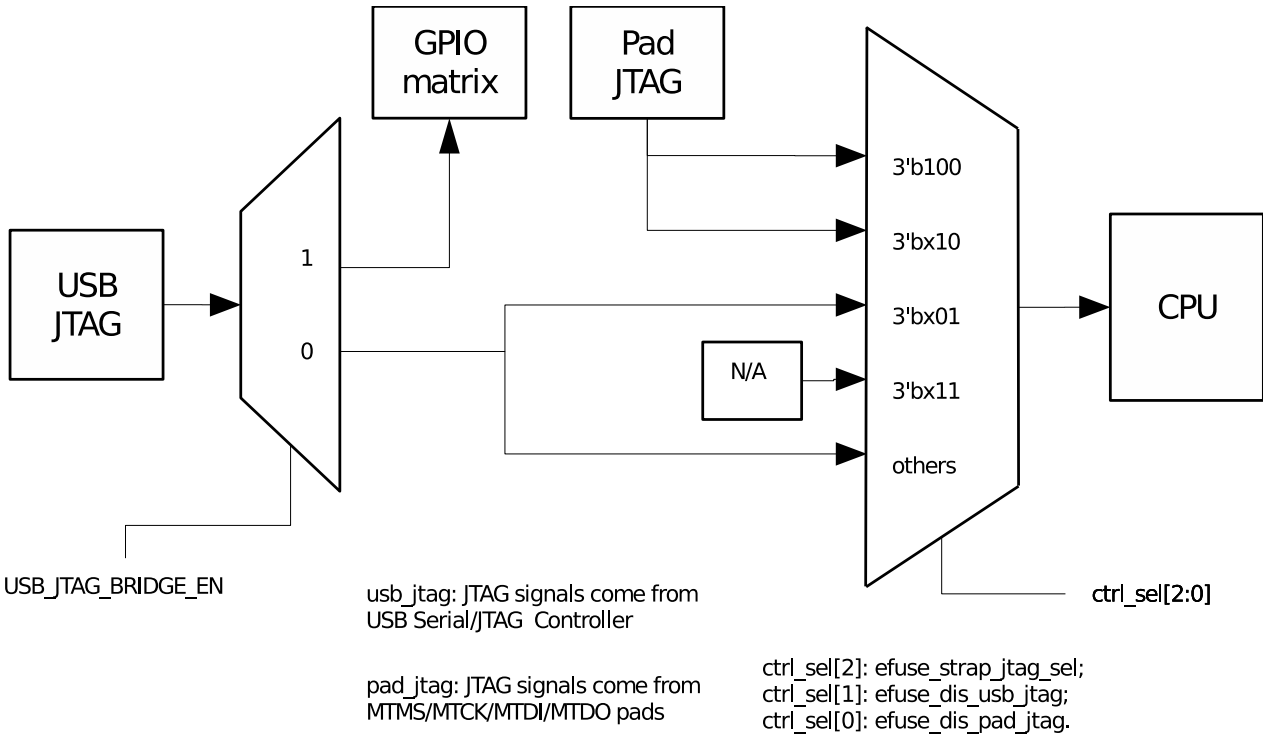


图 33.3-2. JTAG 信号传输

33.3.2 CDC-ACM USB 接口描述

CDC-ACM 接口遵循标准 USB CDC-ACM 类别进行虚拟串口通信，包含一个虚拟中断端点（不会发送任何事件，无使用需求）以及一个批量输入端点 (Bulk IN) 和批量输出端点 (Bulk OUT) 进行数据接收和发送。这些端点一次可以处理最高 64 字节的数据包，实现高吞吐量。CDC-ACM 为标准的 USB 设备类型，主机一般无需任何特殊安装程序就能正常工作，也就是说，当一个 USB 调试设备正确连接至主机时，操作系统应能在片刻后显示新的串口信息。

CDC-ACM 接口可以接收以下标准 CDC-ACM 控制请求：

表 33.3-1. 标准 CDC-ACM 控制请求

命令	操作
SEND_BREAK	接收但忽略（虚拟命令）
SET_LINE_CODING	接收但忽略（虚拟命令）
GET_LINE_CODING	总是返回 9600 baud，无奇偶校验，8 个数据位，1 个停止位
SET_CONTROL_LINE_STATE	设置 RTC/DTR 线的状态，如表 33.3-2 所示

除了通用的通信之外，CDC-ACM 接口还可以复位 ESP32-S3 并选择使其进入下载模式，从而烧录新的固件。这一功能可通过设置虚拟串口的 RTS 和 DTR 线来实现。

表 33.3-2. CDC-ACM 中 RTS 和 DTR 的设置

RTS	DTR	操作
0	0	清除下载模式标志
0	1	置位下载模式标志
1	0	复位 ESP32-S3
1	1	无操作

请注意，当 ESP32-S3 复位时，如果下载模式标志已置位，则 ESP32-S3 重启时将直接进入下载模式；如果下载模式标志已清除，则 ESP32-S3 将从 flash 启动。具体操作流程，请参见章节 33.4。除此之外，也可以通过烧写相应 eFuse 来禁用上述功能，详细信息请参见章节 5 *eFuse 控制器 (EFUSE)*。

33.3.3 CDC-ACM 固件接口描述

由于 USB 串口/JTAG 控制器与 ESP32-S3 的内部 APB 总线相连，因此 CPU 可直接与该模块交互，主要对连接的主机上的虚拟串口进行读写操作。

CPU 向主机发送并从主机接收 0 ~ 64 字节大小的 USB CDC-ACM 串口数据包。主机已接收到足够多的 CDC-ACM 数据时，将向 CDC-ACM 的接收端点发送一个数据包，如果 USB 串行/JTAG 控制器中有空闲缓冲区，该缓冲区将接收这一数据包。反之，主机会定期检查 USB 串口/JTAG 控制器内是否有待向主机发送的数据包，如果有，主机将接收这个数据包。

固件可通过以下两种方式之一获知是否有来自主机的新数据：第一，只要缓冲区中还有来自主机的未读数据，USB_SERIAL_JTAG_SERIAL_OUT_EP_DATA_AVAIL 位将保持为 1；第二，如果有新的未读数据，USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT 中断也将被触发。

当新数据可用时，固件可通过重复从 USB_SERIAL_JTAG_EP1_REG 读取字节来获取该数据。读取每个字节后，可通过检查 USB_SERIAL_JTAG_SERIAL_OUT_EP_DATA_AVAIL 位来看是否还有其他可读取的数据，从而确定需要读取的总字节数。读取完所有数据后，USB 调试设备会自动做好准备，以接收来自主机的新数据包。

当固件需要发送数据时，可将待发送数据置于发送缓冲区并触发刷写，从而使主机以 USB 数据包的形式接收该数据。在此之前，需确保发送缓冲区有可用空间存储待发送数据。固件可通过读取 USB_REG_SERIAL_IN_EP_DATA_FREE 位检查发送缓冲区是否有可用空间：当该值为 1 时，发送缓冲区有可用空间。此时，固件可通过向 USB_SERIAL_JTAG_EP1_REG 寄存器写入字节从而向缓冲区中写入数据。

但是，数据写入后并不会立即触发向主机发送数据，还需对缓冲区执行刷写操作。刷写操作后，整个缓冲区可准备好被 USB 主机立即接收。可通过两种方式触发刷写：将第 64 个字节写入缓冲区后，USB 硬件会自动将缓冲区刷写到主机；固件可通过向 USB_REG_SERIAL_WR_DONE 写入 1 来触发刷写。

不论以何种方式触发刷写操作，在此期间固件都无法向缓冲区写入数据，直到缓冲区中的所有数据都已被主机读取完成。主机读取完成后，将触发 USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT 中断，此时可向缓冲区内写入新的 64 字节数据。

33.3.4 USB-JTAG 接口

USB-JTAG 接口使用一种供应商特定接口类型实现命令解析的功能。它由两个端点组成：一个用于接收命令，一个用于发送响应。此外，一些对时效性要求不高的命令也可以作为控制请求发出。

33.3.5 JTAG 命令处理器

JTAG 命令处理器负责解析主机发送至 JTAG 接口的命令。JTAG 命令处理器内部包含一个全四线 JTAG 总线，包括发送信号到 Xtensa CPU 的 TCK、TMS 和 TDI 输出线，以及从 CPU 返回信号至 JTAG 响应捕捉单元的 TDO 线。这些信号都符合 IEEE 1149.1 JTAG 标准。此外，还有一条 SRST 线用于复位 ESP32-S3。

JTAG 命令处理器会将每个接收到的半字节（4 位）解析为一条命令。由于 USB 以 8 位为一个字节接收数据，这就意味着每个字节中都包含两条命令。USB 命令处理器将先解析高 4 位字节，然后再解析低 4 位字节。这些命令用于控制内部 JTAG 总线的 TCK、TMS、TDI、SRST 线，以及向 JTAG 响应捕捉单元发出信号，说明需要捕捉 TDO 线（由 CPU 调试逻辑驱动）状态。

JTAG 总线中，TCK、TMS、TDI 和 TDO 线直接与 Xtensa CPU 的 JTAG 调试逻辑相连。上文提到的 SRST 线则与 ESP32-S3 数字电路中的复位逻辑相连，该线电平拉高，芯片将进行系统复位。请注意，SRST 线并不会对 USB 串口/JTAG 控制器模块产生影响。

1 个半字节中可包含以下命令：

表 33.3-3. 半字节中的命令

位	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：将 TDI 和 TMS 设置为指示值，并在 TCK 上发出一个时钟脉冲。如果 CAP 位为 1，它将指示 JTAG 响应捕捉单元捕捉 TDO 线的状态。该指令构成了 JTAG 通信的基础。
- CMD_RST：将 SRST 线的状态设置为指示值。该命令可用于复位 ESP32-S3。
- CMD_FLUSH：指示 JTAG 响应捕捉单元对接收到的所有位的缓冲区进行刷写操作，以便主机可以读取这些位。请注意在某些情况下，一次 JTAG 通信会结束于第奇数个命令，即结束于第奇数个半字节。此时，可重复执行该命令直到获得偶数个半字节，使其组成整数个字节。
- CMD_RSV：该版本中保留。ESP32-S3 在接收到该命令时会自动忽略。
- CMD_REP：重复上一条指令（非 CMD_REP）一定的次数。该命令的目的是压缩多次重复 CMD_CLK 的命令流。因此，CMD_CLK 命令后可能跟随着多个 CMD_REP。一次 CMD_REP 命令产生的重复次数可表示为 $no\_repetitions = (R1 \times 2 + R0) \times (4^{cmd\_rep\_count})$，其中 cmd_rep_count 表示该命令之前的 CMD_REP 数量。请注意，CMD_REP 仅用于重复 CMD_CLK 命令。也就是说，如果在 CMD_FLUSH 后使用该命令，USB 设备将无法响应，需进行 USB 复位后才可恢复正常。

33.3.6 USB-JTAG 接口：CMD_REP 使用示例

下列命令用于演示如何使用 CMD_REP 命令。请注意，该示例中每个命令为半字节，命令流的每个字节为 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)

每一步骤的具体操作为：

1. TCK 上发出一个时钟脉冲，TDI 和 TMS 设置为 0。未捕捉到任何数据。
2. TCK 上再次发出 $(0 \times 2 + 1) \times (4^0) = 1$ 个时钟脉冲，其他设置与步骤 1 相同。
3. TCK 上发出一个时钟脉冲，TDI 和 TMS 设置为 0。捕捉到 TDO 线上的数据。
4. TCK 上再次发出 $(1 \times 2 + 0) \times (4^0) = 2$ 个时钟脉冲，其他设置与步骤 3 相同。
5. 未发生任何动作：$(0 \times 2 + 0) \times (4^1) = 0$。请注意，该步骤操作将增加下一步骤中的 cmd_rep_count 数值。
6. TCK 上再次发出 $(1 \times 2 + 1) \times (4^2) = 48$ 个时钟脉冲，其他设置与步骤 3 相同。

换言之，该命令流示例的操作结果等同于执行 2 次命令 1，然后执行 51 次命令 3。

33.3.7 USB-JTAG 接口：响应捕捉单元

响应捕捉单元首先读取内部 JTAG 总线的 TDO 线，并在命令处理器执行 CMD_CLK (cap=1) 命令时捕捉 TDO 线的值。然后，响应捕捉单元将这个值放入内部移位寄存器中，且在接收到 8 位时向 USB 缓冲区写入 1 个字节。这 8 位中的最低有效位即为最先从 TDO 线读取的值。

一旦接收到 64 字节（512 位）数据或执行 CMD_FLUSH 命令后，响应捕捉单元将使缓冲区可被主机接收。请注意，USB 逻辑的接口为双缓冲。这样，只要 USB 的吞吐量充足，响应捕捉单元就可以随时接收更多数据，即当一个缓冲区等待发送给主机时，另一个缓冲区可以继续接收数据。当主机从缓冲区成功接收数据且响应捕捉单元对缓冲区执行刷写操作后，这两个缓冲区便可以交换位置。

同时，这也意味着一个命令流可导致最多 128 字节（若该命令流中有刷写命令，则该数字会减小）的捕捉数据生成，而不需要主机主动接收这些数据。如果还是生成了超过该阈值数量的捕捉数据，则命令流将被暂停，且在这些数据被读取之前设备不会接收其他命令。

另需注意，一般情况下，响应捕捉单元的逻辑会尽量不发送 0 字节响应。例如，当发送一系列 CMD_FLUSH 命令后并不会产生一系列 0 字节 USB 响应。但是，当前版本中一些特殊情况下也可能产生 0 字节响应，建议您可直接忽略这些响应信息。

33.3.8 USB-JTAG 接口：控制传输请求

除命令处理器和响应捕捉单元之外，USB-JTAG 接口也可接收一些控制请求，具体如下表所示：

表 33.3-4. USB-JTAG 控制请求

bmRequestType	bRequest	wValue	wIndex	wLength	Data
01000000b	0 (VEND_JTAG_SETDIV)	[divider]	接口	0	None
01000000b	1 (VEND_JTAG_SETIO)	[iobits]	接口	0	None
11000000b	2 (VEND_JTAG_GETTDO)	0	接口	1	[iostate]
10000000b	6 (GET_DESCRIPTOR)	0x2000	0	256	[jtag cap desc]

- VEND_JTAG_SETDIV：设置使用的分频器。该命令将直接影响 TCK 时钟脉冲的持续时间。TCK 时钟脉冲来自于由内部分频器向下分频得到的内部 APB 时钟。该控制请求允许主机来设置这个内部分频器。请注意，该分频器在启动时的初始值为 2，即 TCK 时钟速率一般为 40 MHz。
- VEND_JTAG_SETIO：跳过 JTAG 命令处理器直接将内部 TDI、TDO、TMS 和 SRST 线设置为指定值。这些指定值在 wValue 字段中以 11'b0, srst, trst, tck, tms, tdi 的格式编码。
- VEND_JTAG_GETTDO：跳过 JTAG 响应捕捉单元直接读取内部 TDO 信号。该请求将返回 1 个字节，其中的最低有效位代表 TDO 线的状态。
- GET_DESCRIPTOR：为标准 USB 请求，该请求也可使用供应商专用的 0x2000 wValue 获取 JTAG 功能描述符。该请求将返回一定字节，具体字节数所代表的 USB-JTAG 适配器功能如表 33.3-5 所示。这一固定结构允许主机软件自动支持未来新的硬件版本，无需再次更新。

ESP32-S3 包含的 JTAG 功能描述符如下表所示。请注意，所有 16 位值都为小端序存储。

表 33.3-5. JTAG 功能描述符

字节	数值	描述
0	1	JTAG 协议功能结构的版本
1	10	JTAG 协议功能长度
2	1	结构类型：1 代表高速功能结构类型
3	8	该高速功能结构长度
4 ~ 5	8000	以 10 KHz 为增量的 APB_CLK 速度，其基础速度为该值的一半
6 ~ 7	1	最小分频系数，可通过 VEND_JTAG_SETDIV 设置
8 ~ 9	255	最大分频系数，可通过 VEND_JTAG_SETDIV 设置

33.4 操作建议

33.4.1 内部/外部 PHY 选择

由于 ESP32-S3 只有一个内部 PHY，因此在首次烧录时，用户需要通过烧录 eFuse 相应位配置 USB 初始设置来决定如何在预定的应用中使用内部 PHY。这一配置也会影响 ROM 下载模式，因为虽然 USB-OTG 和 USB 串口/JTAG 控制器都支持串口烧录，但只有 USB-OTG 支持 DFU 协议，且这二者中只有 USB 串口/JTAG 控制器支持 JTAG 调试功能。即使不使用 USB，在使用外部 JTAG 适配器时也需要进行相应的 eFuse 配置。

表 33.4-1 列出了进行相应配置时需要烧录的 eFuse 位。请注意，下表中的配置主要是在下载模式和引导加载程序中进行，因为用户代码一旦开始运行，配置代码可能在此时被改变。

表 33.4-1. 内部/外部 PHY 选择与相应 eFuse 配置

用例	需烧录的 eFuse 位	说明
仅 USB 串口/JTAG 使用内部 PHY	无	-
仅 USB-OTG 使用内部 PHY	EFUSE_USB_PHY_SEL + EFUSE_DIS_USB_JTAG	配置 GPIO 进行 JTAG 调试
USB 串口/JTAG 使用内部 PHY，USB-OTG 使用外部 PHY	无	-
USB-OTG 使用内部 PHY，USB 串口/JTAG 使用外部 PHY	EFUSE_USB_PHY_SEL	-
USB 串口/JTAG 使用内部 PHY，USB-OTG 使用外部 PHY，使用 strapping 管脚配置	EFUSE_USB_PHY_SEL + EFUSE_STRAP_JTAG_SEL	拉高 GPIO3
USB-OTG 使用内部 PHY，USB 串口/JTAG 使用外部 PHY，使用 strapping 管脚配置	EFUSE_USB_PHY_SEL + EFUSE_STRAP_JTAG_SEL	拉低 GPIO3

用户程序运行后，可通过配置寄存器修改初始配置。即可通过配置 RTC_CNTL_SW_HW_USB_PHY_SEL 覆盖 EFUSE_USB_PHY_SEL 的值：当置位该位时，USB PHY 的选择逻辑将使用 RTC_CNTL_SW_USB_PHY_SEL 配置的值，而非 EFUSE_USB_PHY_SEL 的值。

如图 33.3-1 所示，默认状态下 (phy_sel = 0)，ESP32-S3 USB 串口/JTAG 控制器连接至内部 PHY, USB-OTG 连接至外部 PHY。但在启用 USB-OTG Download 下载模式时，芯片上电后会在 ROM 中初始化用于连接外部 PHY 的 IO 焊盘，初始化后各 IO 焊盘状态如下：

表 33.4-2. USB-OTG Download 下载模式下芯片初始化后 IO 焊盘状态

IO 焊盘	输入/输出模式	电平状态
VP (MTMS)	INPUT	-
VM (MTDI)	INPUT	-
RCV (GPIO21)	INPUT	-
OEN (MTDO)	OUTPUT	HIGH
VPO (MTCK)	OUTPUT	LOW
VMO(GPIO38)	OUTPUT	LOW

若无需使用 USB-OTG Download 下载模式，可烧写 eFuse 位 EFUSE_DIS_USB_OTG_DOWNLOAD_MODE 禁用该模式，以避免 IO 状态的变化。

33.4.2 运行操作

使用 USB 串口/JTAG 控制器之前，几乎不需要多余的配置。除了主机操作系统已经完成的标准 USB 初始化之外，USB-JTAG 硬件本身不需要进行任何配置。而 CDC-ACM 虚拟串口在主机端也是即插即用的。

固件方面也几乎不需要初始化：USB 硬件是自初始化的，在其启动后，如果固件连接了一台主机并在 CDC-ACM 接口上监听，无需任何特定设置就可以实现前文所述的数据交换，除非固件选择设置了中断处理程序。

需要注意的是，可能会出现主机未连接或 CDC-ACM 虚拟串口未打开的情况。在这种情况下，发送至主机的数据包永远无法被接收，发送缓冲区也永远不会为空。因而，对此进行检测以及执行超时操作便十分重要，这是检测主机侧的端口是否关闭的唯一方式。

其次，需知 USB 设备依赖于产生 48 MHz USB PHY 时钟的 PLL 时钟和 APB 时钟。具体来说，正确的 USB 顺序操作要求 APB 时钟至少为 40 MHz，但 APB 时钟低至 10 MHz 时，USB 设备和大多数主机也能工作。此时 USB 是否会出现其他问题则取决于主机的硬件和驱动条件，可能会有设备出现无法响应或在首次访问时消失的情况。

换言之，在 Light-sleep 模式下对 USB 串口/JTAG 控制器进行时钟门控操作时，APB 时钟将受到影响。除此之外，在 Deep-sleep 模式下，USB 串口/JTAG 控制器（及其连接的 Xtensa CPU）将完全断电。如果有设备需要在这两种模式下进行调试，最好使用一个外部 JTAG 调试器和串行接口。

CDC-ACM 接口还可用于复位芯片，使其进入或退出下载模式。产生正确的握手信号序列则有些复杂，因为大多数操作系统仅支持分别设置或重置 DTR 和 RTS，无法同时进行。此外，一些驱动程序（如，Windows 系统上的标准 CDC-ACM 驱动程序）须先设置 RTS 后才可设置 DTR，因此您必须明确设置 RTS 才能传播 DTR 的值。推荐遵循以下程序进行设置：

复位芯片使其进入下载模式：

表 33.4-3. 复位芯片进入下载模式

操作	内部状态	备注
清除 DTR	RTS=?, DTR=0	初始化以获取数值
清除 RTS	RTS=0, DTR=0	-
设置 DTR	RTS=0, DTR=1	设置下载模式标志
清除 RTS	RTS=0, DTR=1	传播 DTR
设置 RTS	RTS=1, DTR=1	-
清除 DTR	RTS=1, DTR=0	复位芯片
设置 RTS	RTS=1, DTR=0	传播 DTR
清除 RTS	RTS=0, DTR=0	清除下载标志

复位 SoC 使其从 flash 启动：

表 33.4-4. 复位 SoC 进行启动

操作	内部状态	备注
清除 DTR	RTS=?, DTR=0	-
清除 RTS	RTS=0, DTR=0	清除下载标志
设置 RTS	RTS=1, DTR=0	复位 SoC
清除 RTS	RTS=0, DTR=0	退出复位

33.5 寄存器列表

本小节的所有地址均为相对于 USB Serial/JTAG 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

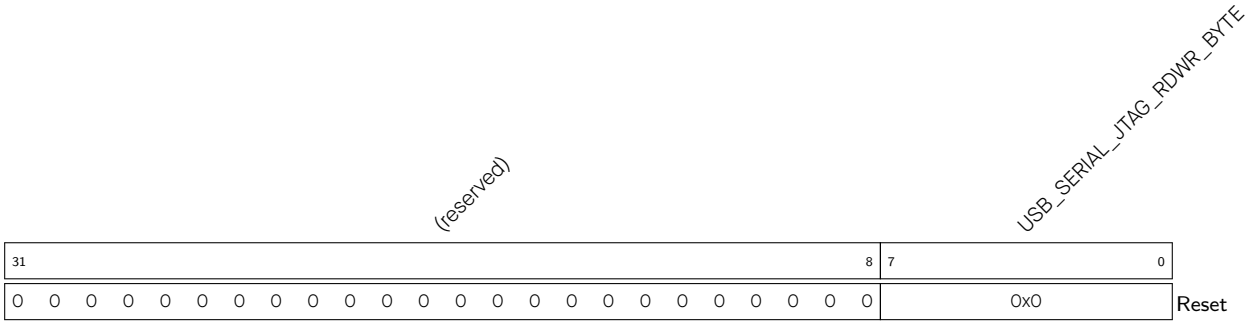
请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问权限
配置寄存器			
USB_SERIAL_JTAG_EP1_REG	输出端点 1 FIFO 访问配置寄存器	0x0000	R/W
USB_SERIAL_JTAG_EP1_CONF_REG	输出端点 1 配置和状态寄存器	0x0004	varies
USB_SERIAL_JTAG_CONF0_REG	配置 0 寄存器	0x0018	R/W
USB_SERIAL_JTAG_MISC_CONF_REG	时钟使能控制寄存器	0x0044	R/W
USB_SERIAL_JTAG_MEM_CONF_REG	存储器配置寄存器	0x0048	R/W
USB_SERIAL_JTAG_TEST_REG	内部 PHY 调试寄存器	0x001C	varies
中断寄存器			
USB_SERIAL_JTAG_INT_RAW_REG	中断原始状态寄存器	0x0008	R/ WTC/ SS
USB_SERIAL_JTAG_INT_ST_REG	中断状态寄存器	0x000C	RO
USB_SERIAL_JTAG_INT_ENA_REG	中断使能状态寄存器	0x0010	R/W
USB_SERIAL_JTAG_INT_CLR_REG	中断清除状态寄存器	0x0014	WT
状态寄存器			
USB_SERIAL_JTAG_JFIFO_ST_REG	JTAG FIFO 状态与控制寄存器	0x0020	varies
USB_SERIAL_JTAG_FRAM_NUM_REG	接收 SOF 帧索引寄存器	0x0024	RO
USB_SERIAL_JTAG_IN_EP0_ST_REG	输入端点 0 状态寄存器	0x0028	RO
USB_SERIAL_JTAG_IN_EP1_ST_REG	输入端点 1 状态寄存器	0x002C	RO
USB_SERIAL_JTAG_IN_EP2_ST_REG	输入端点 2 状态寄存器	0x0030	RO
USB_SERIAL_JTAG_IN_EP3_ST_REG	输入端点 3 状态寄存器	0x0034	RO
USB_SERIAL_JTAG_OUT_EP0_ST_REG	输出端点 0 状态寄存器	0x0038	RO
USB_SERIAL_JTAG_OUT_EP1_ST_REG	输出端点 1 状态寄存器	0x003C	RO
USB_SERIAL_JTAG_OUT_EP2_ST_REG	输出端点 2 状态寄存器	0x0040	RO
版本寄存器			
USB_SERIAL_JTAG_DATE_REG	版本寄存器	0x0080	R/W

33.6 寄存器

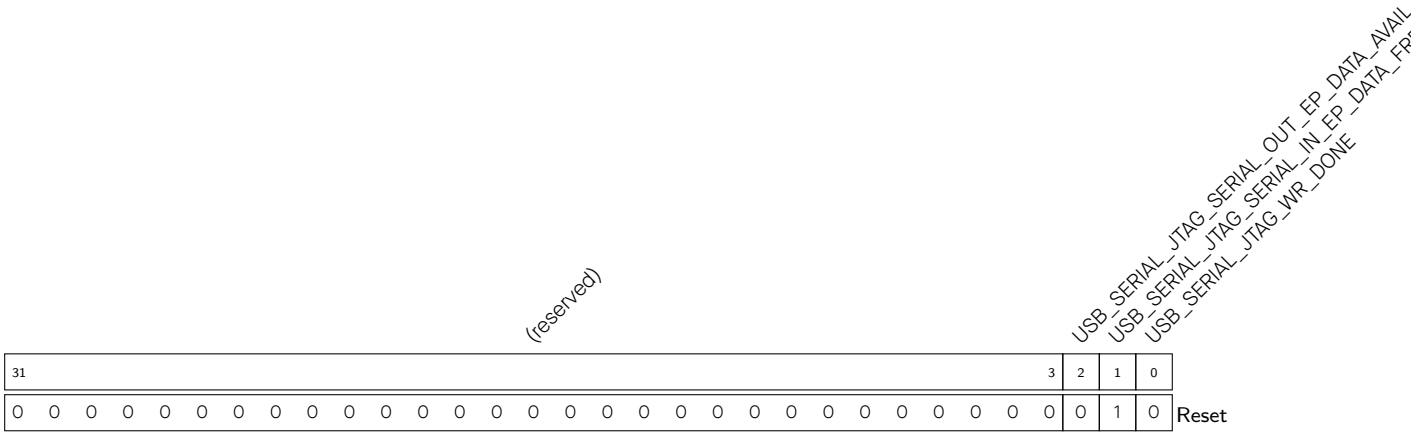
本小节的所有地址均为相对于 USB Serial/JTAG 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 33.1. USB_SERIAL_JTAG_EP1_REG (0x0000)



USB_SERIAL_JTAG_RDWR_BYTE 通过该字段向 UART Tx FIFO 中写入数据，或者从 UART Rx FIFO 中读取数据。USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT 置位时，用户可向 UART Tx FIFO 中写入数据（最大 64 字节）。当 USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT 置位时，用户可通过查看 USB_SERIAL_JTAG_OUT_EP1_WR_ADDR 和 USB_SERIAL_JTAG_OUT_EP1_RD_ADDR 的值获知接收到的数据量，然后从 UART Rx FIFO 中读取这些数据。(R/W)

Register 33.2. USB_SERIAL_JTAG_EP1_CONF_REG (0x0004)

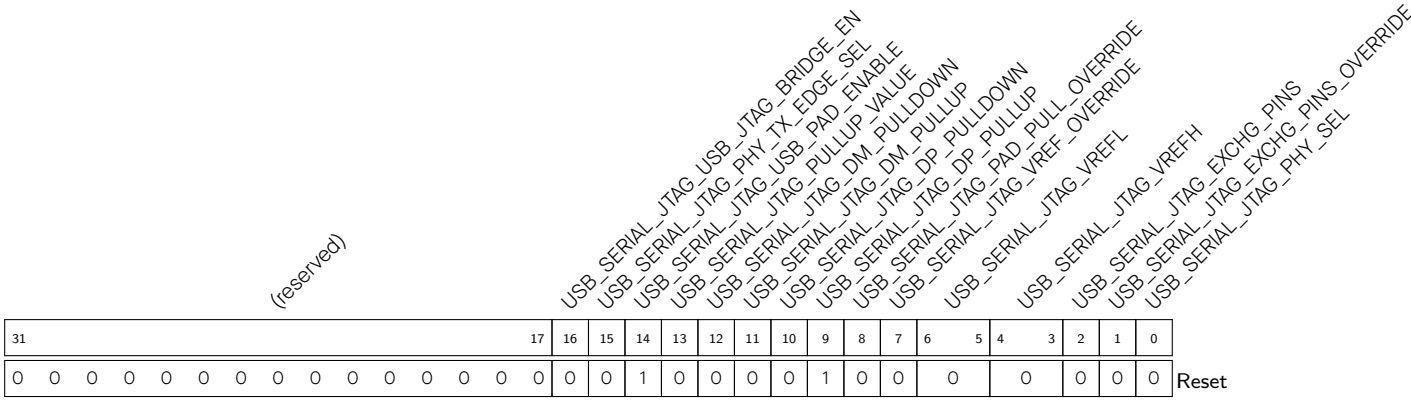


USB_SERIAL_JTAG_WR_DONE 置位表示已完成向 UART Tx FIFO 写入字节。(WT)

USB_SERIAL_JTAG_SERIAL_IN_EP_DATA_FREE 1'b1：UART Tx FIFO 不为空且可以写入数据。写入 USB_SERIAL_JTAG_WR_DONE 后，该位将保持为 1' b0，直到其中的数据已发送至 USB 主机。(RO)

USB_SERIAL_JTAG_SERIAL_OUT_EP_DATA_AVAIL 1'b1：UART Rx FIFO 中有数据。(RO)

Register 33.3. USB_SERIAL_JTAG_CONF0_REG (0x0018)



USB_SERIAL_JTAG_PHY_SEL 选择使用内部 PHY 或外部 PHY。1’ b0：内部 PHY；1’ b1：外部 PHY。(R/W)

USB_SERIAL_JTAG_EXCHG_PINS_OVERRIDE 使能软件控制 USB D+ 和 D- 管脚交换。(R/W)

USB_SERIAL_JTAG_EXCHG_PINS 交换 USB D+ 和 D- 管脚。(R/W)

USB_SERIAL_JTAG_VREFH 控制单端输入高阈值，1.76 V ~ 2 V，步长 80 mV。(R/W)

USB_SERIAL_JTAG_VREFL 控制单端输入低阈值，0.8 V ~ 1.04 V，步长 80 mV。(R/W)

USB_SERIAL_JTAG_VREF_OVERRIDE 使能软件控制输入阈值。(R/W)

USB_SERIAL_JTAG_PAD_PULL_OVERRIDE 使能软件控制 USB D+ 和 D- 管脚的上下拉。(R/W)

USB_SERIAL_JTAG_DP_PULLUP USB D+ 管脚的上拉电阻。(R/W)

USB_SERIAL_JTAG_DP_PULLDOWN USB D+ 管脚的下拉电阻。(R/W)

USB_SERIAL_JTAG_DM_PULLUP USB D- 管脚的上拉电阻。(R/W)

USB_SERIAL_JTAG_DM_PULLDOWN USB D- 管脚的下拉电阻。(R/W)

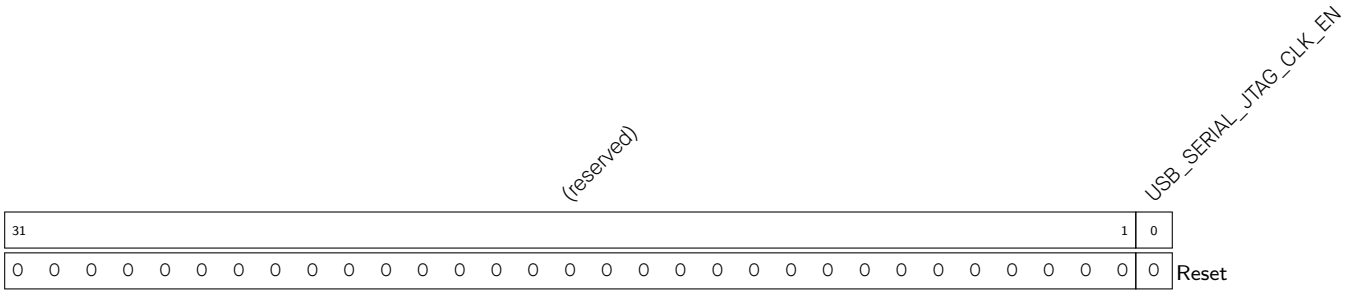
USB_SERIAL_JTAG_PULLUP_VALUE 控制上拉数值。0：2.2 K；1：1.1 K。(R/W)

USB_SERIAL_JTAG_USB_PAD_ENABLE 使能 USB 管脚。(R/W)

USB_SERIAL_JTAG_PHY_TX_EDGE_SEL 0：时钟下降沿 TX 输出；1：时钟上升沿 TX 输出。(R/W)

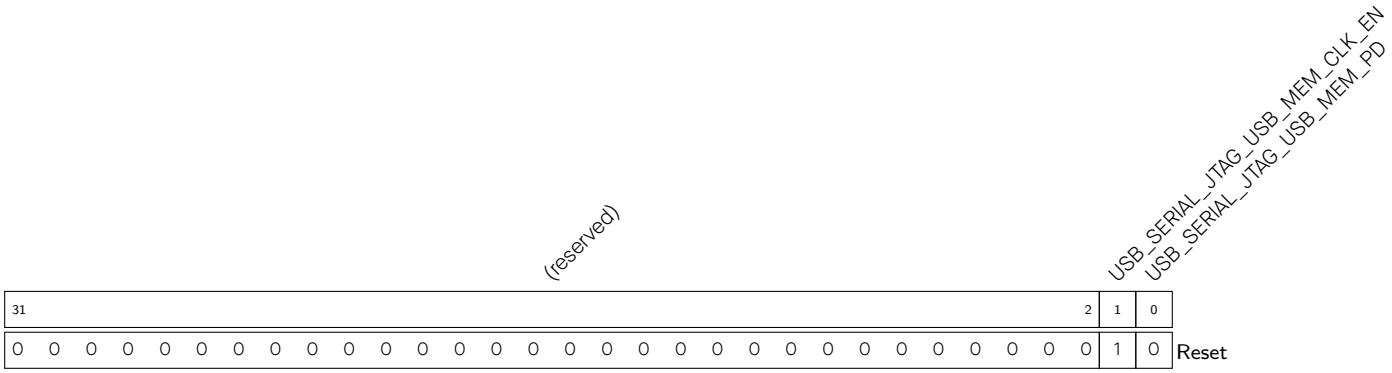
USB_SERIAL_JTAG_USB_JTAG_BRIDGE_EN 置位 usb_jtag 和内部 JTAG 之间断开连接，MTMS, MTDI, MTCK 为通过 GPIO 交换矩阵的输出，MTDO 为通过 GPIO 交换矩阵的输入。(R/W)

Register 33.4. USB_SERIAL_JTAG_MISC_CONF_REG (0x0044)



USB_SERIAL_JTAG_CLK_EN 1'h1：强制打开寄存器的时钟；1'h0：支持仅当应用程序向寄存器写入数据时打开时钟。(R/W)

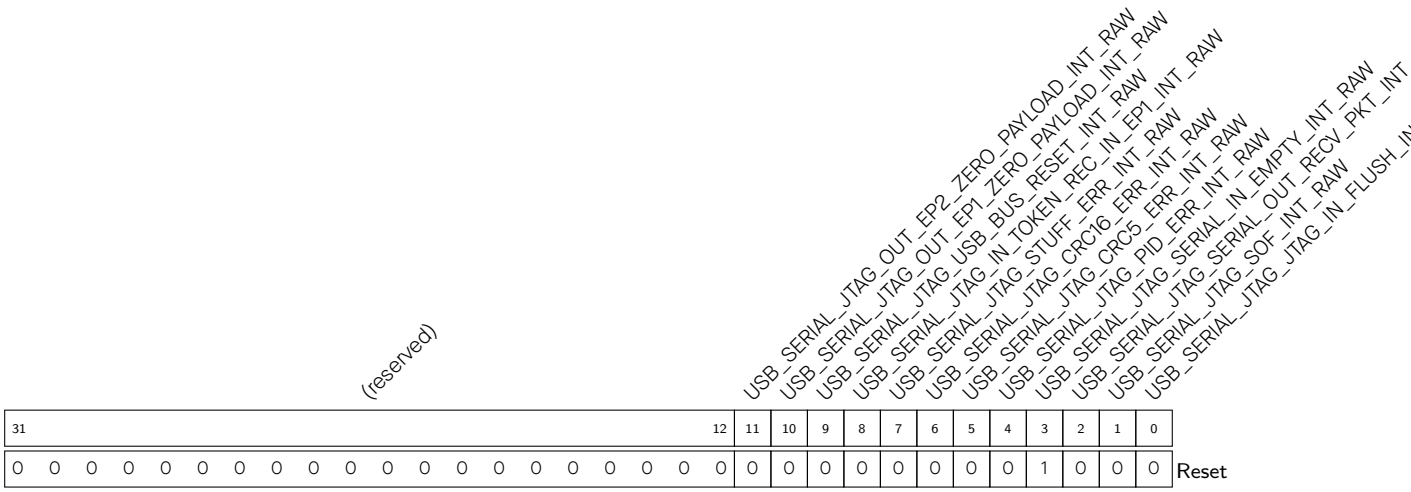
Register 33.5. USB_SERIAL_JTAG_MEM_CONF_REG (0x0048)



USB_SERIAL_JTAG_USB_MEM_PD 置位关闭 USB 存储器。(R/W)

USB_SERIAL_JTAG_USB_MEM_CLK_EN 置位强制对 USB 存储器进行时钟分频。(R/W)

Register 33.6. USB_SERIAL_JTAG_INT_RAW_REG (0x0008)



USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_RAW JTAG 输入端口 2 接收到刷写命令时，原始中断位变为高电平。(R/WTC/SS)

USB_SERIAL_JTAG_SOF_INT_RAW 接收到 SOF 帧时，原始中断位变为高电平。(R/WTC/SS)

USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_RAW 串口输出端点接收到 1 个数据包时，原始中断位变为高电平。(R/WTC/SS)

USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_RAW 串口输入端点为空时，原始中断位变为高电平。(R/WTC/SS)

USB_SERIAL_JTAG_PID_ERR_INT_RAW 检测到 PID 错误时，原始中断位变为高电平。(R/WTC/SS)

USB_SERIAL_JTAG_CRC5_ERR_INT_RAW 检测到 CRC5 错误时，原始中断位变为高电平。(R/WTC/SS)

USB_SERIAL_JTAG_CRC16_ERR_INT_RAW 检测到 CRC16 错误时，原始中断位变为高电平。(R/WTC/SS)

USB_SERIAL_JTAG_STUFF_ERR_INT_RAW 检测到位填充错误时，原始中断位变为高电平。(R/WTC/SS)

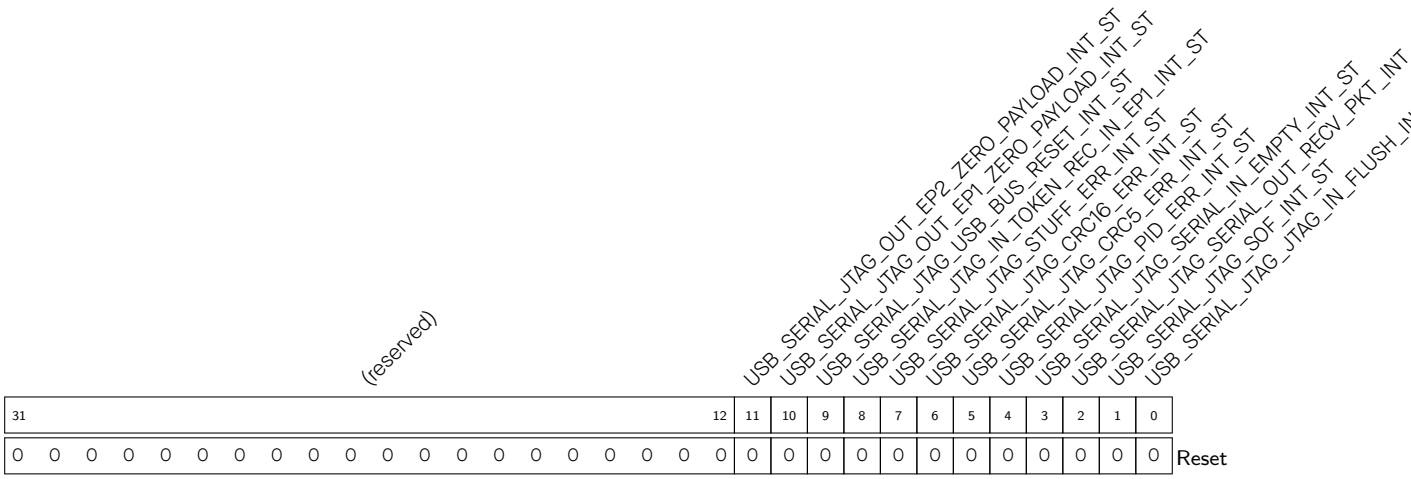
USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_RAW 输入端点 1 接收到一个 IN 令牌时，原始中断位变为高电平。(R/WTC/SS)

USB_SERIAL_JTAG_USB_BUS_RESET_INT_RAW 检测到 USB 总线复位时，原始中断位变为高电平。(R/WTC/SS)

USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_RAW 输出端点 1 接收到有效载荷为 0 的数据包时，原始中断位变为高电平。(R/WTC/SS)

USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_RAW 输出端点 2 接收到有效载荷为 0 的数据包时，原始中断位变为高电平。(R/WTC/SS)

Register 33.7. USB_SERIAL_JTAG_INT_ST_REG (0x000C)



USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_ST USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT 的原始中断状态位。(RO)

USB_SERIAL_JTAG_SOF_INT_ST USB_SERIAL_JTAG_SOF_INT 的原始中断状态位。(RO)

USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_ST USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT 的原始中断状态位。(RO)

USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_ST USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT 中断的原始中断状态位。(RO)

USB_SERIAL_JTAG_PID_ERR_INT_ST USB_SERIAL_JTAG_PID_ERR_INT 的原始中断状态位。(RO)

USB_SERIAL_JTAG_CRC5_ERR_INT_ST USB_SERIAL_JTAG_CRC5_ERR_INT 的原始中断状态位。(RO)

USB_SERIAL_JTAG_CRC16_ERR_INT_ST USB_SERIAL_JTAG_CRC16_ERR_INT 的原始中断状态位。(RO)

USB_SERIAL_JTAG_STUFF_ERR_INT_ST USB_SERIAL_JTAG_STUFF_ERR_INT 的原始中断状态位。(RO)

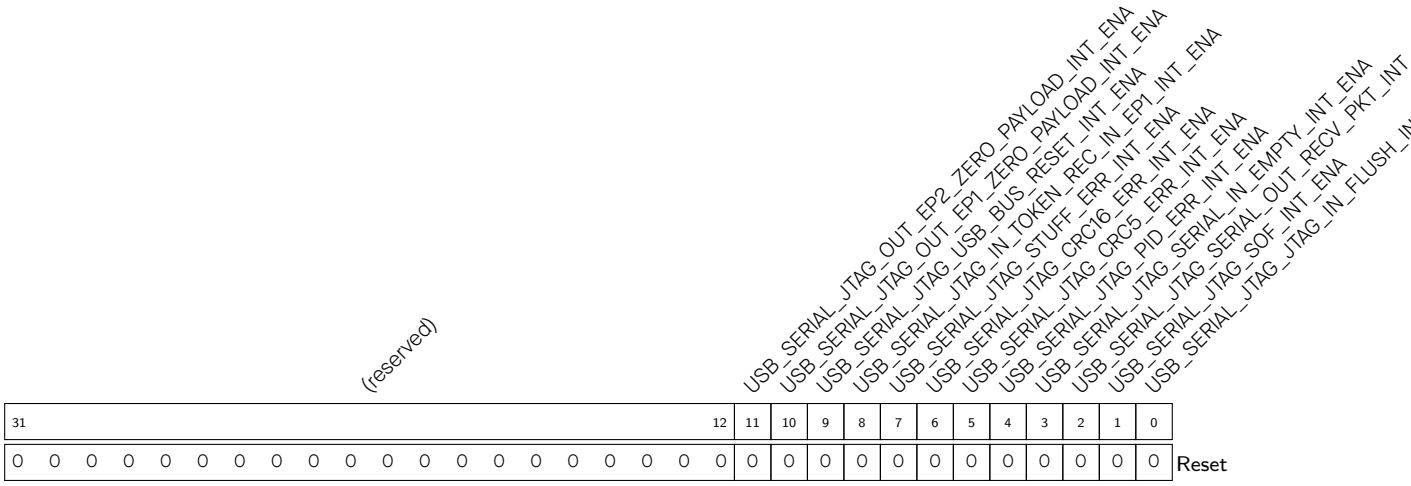
USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_ST USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT 的原始中断状态位。(RO)

USB_SERIAL_JTAG_USB_BUS_RESET_INT_ST USB_SERIAL_JTAG_USB_BUS_RESET_INT 的原始中断状态位。(RO)

USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_ST USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT 的原始中断状态位。(RO)

USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_ST USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT 的原始中断状态位。(RO)

Register 33.8. USB_SERIAL_JTAG_INT_ENA_REG (0x0010)



USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_ENA USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT 的中断使能位。(R/W)

USB_SERIAL_JTAG_SOF_INT_ENA USB_SERIAL_JTAG_SOF_INT 的中断使能位。(R/W)

USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_ENA USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT 的中断使能位。(R/W)

USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_ENA USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT 的中断使能位。(R/W)

USB_SERIAL_JTAG_PID_ERR_INT_ENA USB_SERIAL_JTAG_PID_ERR_INT 的中断使能位。(R/W)

USB_SERIAL_JTAG_CRC5_ERR_INT_ENA USB_SERIAL_JTAG_CRC5_ERR_INT 的中断使能位。(R/W)

USB_SERIAL_JTAG_CRC16_ERR_INT_ENA USB_SERIAL_JTAG_CRC16_ERR_INT 的中断使能位。(R/W)

USB_SERIAL_JTAG_STUFF_ERR_INT_ENA USB_SERIAL_JTAG_STUFF_ERR_INT 的中断使能位。(R/W)

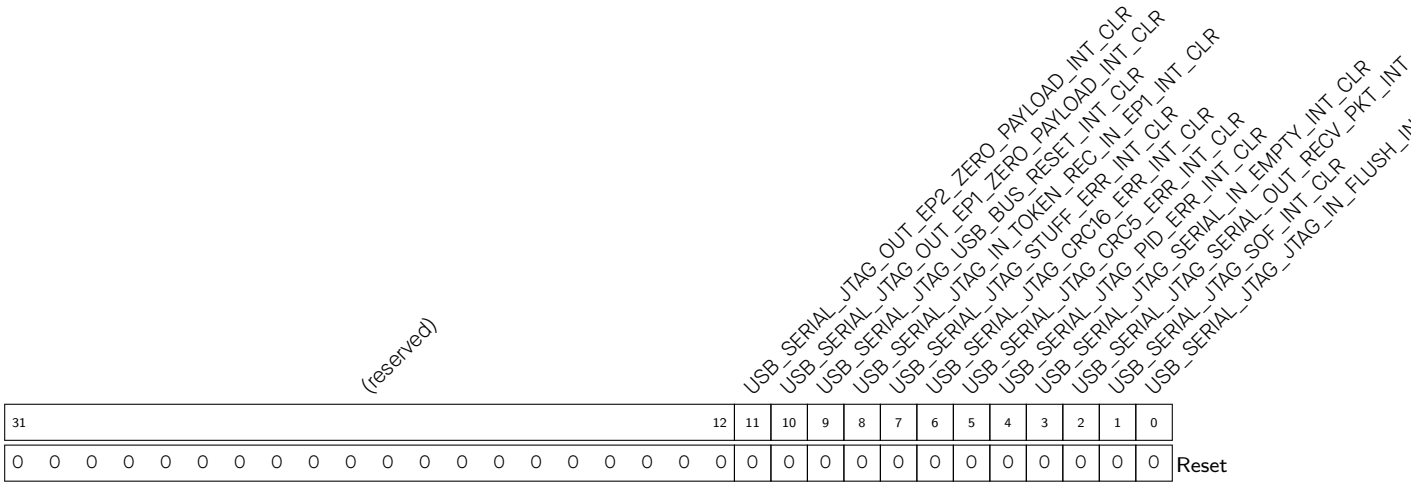
USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_ENA USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT 的中断使能位。(R/W)

USB_SERIAL_JTAG_USB_BUS_RESET_INT_ENA USB_SERIAL_JTAG_USB_BUS_RESET_INT 的中断使能位。(R/W)

USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_ENA USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT 的中断使能位。(R/W)

USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_ENA USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT 的中断使能位。(R/W)

Register 33.9. USB_SERIAL_JTAG_INT_CLR_REG (0x0014)



USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_CLR 置位清除 USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT 中断。(WT)

USB_SERIAL_JTAG_SOF_INT_CLR 置位清除 USB_SERIAL_JTAG_JTAG_SOF_INT 中断。(WT)

USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_CLR 置位清除 USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT 中断。(WT)

USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_CLR 置位清除 USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT 中断。(WT)

USB_SERIAL_JTAG_PID_ERR_INT_CLR 置位清除 USB_SERIAL_JTAG_PID_ERR_INT 中断。(WT)

USB_SERIAL_JTAG_CRC5_ERR_INT_CLR 置位清除 USB_SERIAL_JTAG_CRC5_ERR_INT 中断。(WT)

USB_SERIAL_JTAG_CRC16_ERR_INT_CLR 置位清除 USB_SERIAL_JTAG_CRC16_ERR_INT 中断。(WT)

USB_SERIAL_JTAG_STUFF_ERR_INT_CLR 置位清除 USB_SERIAL_JTAG_STUFF_ERR_INT 中断。(WT)

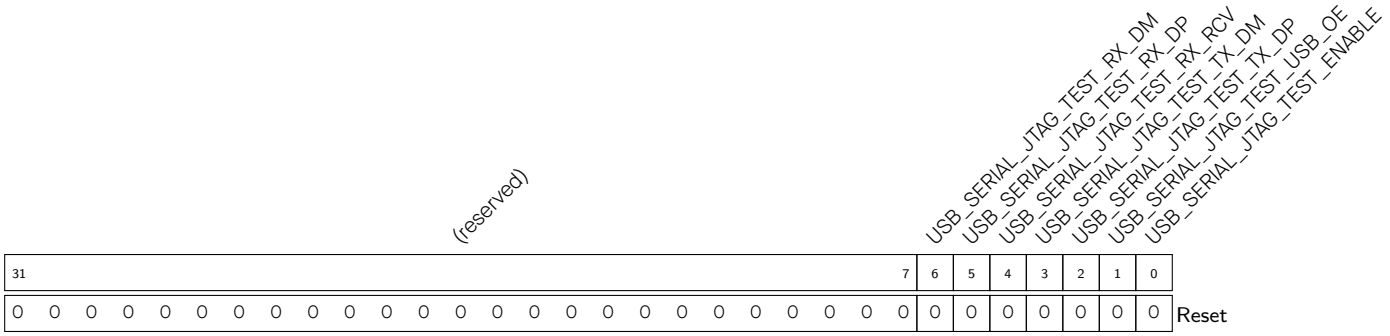
USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_CLR 置位清除 USB_SERIAL_JTAG_IN_TOKEN_IN_EP1_INT 中断。(WT)

USB_SERIAL_JTAG_USB_BUS_RESET_INT_CLR 置位清除 USB_SERIAL_JTAG_USB_BUS_RESET_INT 中断。(WT)

USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_CLR 置位清除 USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT 中断。(WT)

USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_CLR 置位清除 USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT 中断。(WT)

Register 33.10. USB_SERIAL_JTAG_TEST_REG (0x001C)



USB_SERIAL_JTAG_TEST_ENABLE 使能测试 USB 管脚。(R/W)

USB_SERIAL_JTAG_TEST_USB_OE 测试 USB oe 填充。(R/W)

USB_SERIAL_JTAG_TEST_TX_DP 测试 USB D+ 管脚的发送值。(R/W)

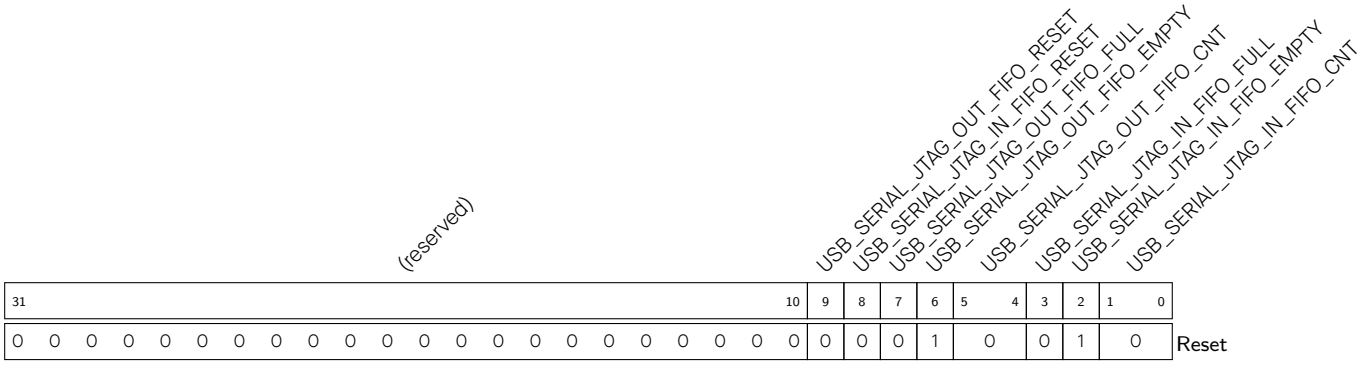
USB_SERIAL_JTAG_TEST_TX_DM 测试 USB D- 管脚的发送值。(R/W)

USB_SERIAL_JTAG_TEST_RX_RCV 测试 USB 发送差值。(RO)

USB_SERIAL_JTAG_TEST_RX_DP 测试 USB D+ 管脚的接收值。(RO)

USB_SERIAL_JTAG_TEST_RX_DM 测试 USB D- 管脚的接收值。(RO)

Register 33.11. USB_SERIAL_JTAG_JFIFO_ST_REG (0x0020)



USB_SERIAL_JTAG_IN_FIFO_CNT FIFO 计数器中的 JTAG。(RO)

USB_SERIAL_JTAG_IN_FIFO_EMPTY 置位表示 FIFO 中的 JTAG 为空。(RO)

USB_SERIAL_JTAG_IN_FIFO_FULL 置位表示 FIFO 中的 JTAG 为满。(RO)

USB_SERIAL_JTAG_OUT_FIFO_CNT JTAG 输出 FIFO 计数器。(RO)

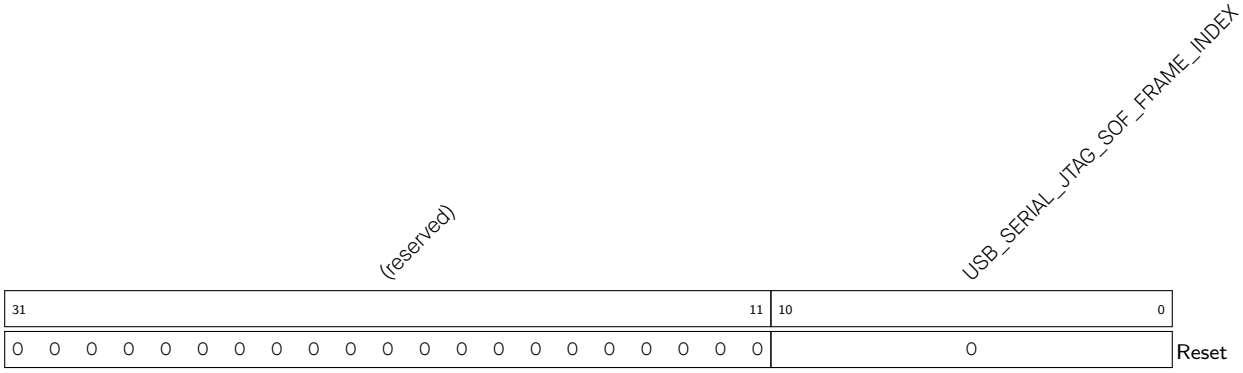
USB_SERIAL_JTAG_OUT_FIFO_EMPTY 置位表示 JTAG 输出 FIFO 为空。(RO)

USB_SERIAL_JTAG_OUT_FIFO_FULL 置位表示 JTAG 输出 FIFO 为满。(RO)

USB_SERIAL_JTAG_IN_FIFO_RESET 置位复位 FIFO 中的 JTAG。(R/W)

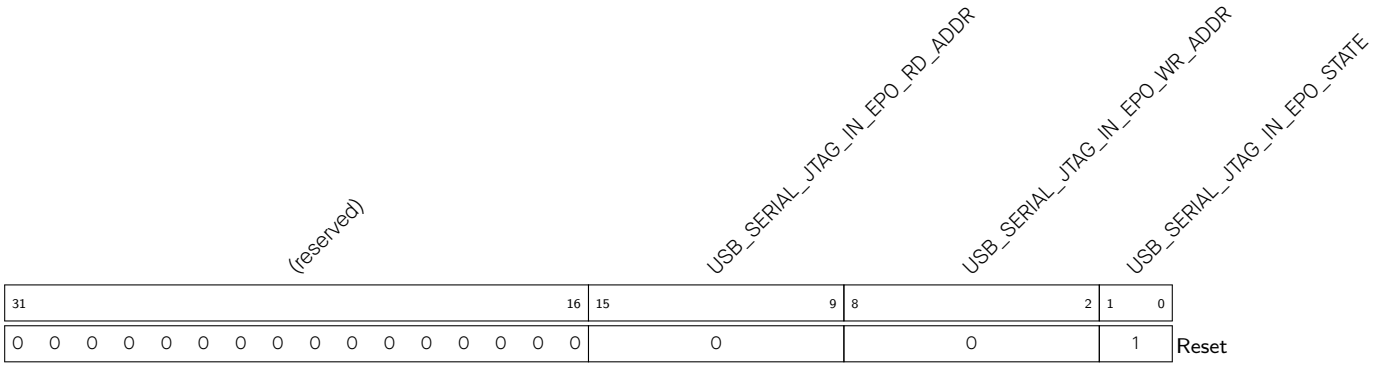
USB_SERIAL_JTAG_OUT_FIFO_RESET 置位复位 JTAG 输出 FIFO。(R/W)

Register 33.12. USB_SERIAL_JTAG_FRAM_NUM_REG (0x0024)



USB_SERIAL_JTAG_SOF_FRAME_INDEX 接收 SOF 帧的帧索引。(RO)

Register 33.13. USB_SERIAL_JTAG_IN_EP0_ST_REG (0x0028)

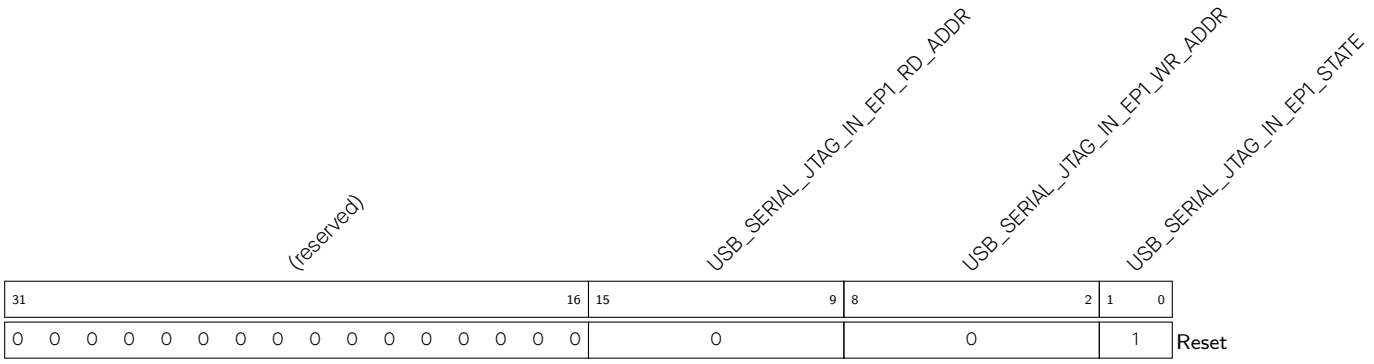


USB_SERIAL_JTAG_IN_EP0_STATE 输入端点 0 的状态。(RO)

USB_SERIAL_JTAG_IN_EP0_WR_ADDR 输入端点 0 的写入数据地址。(RO)

USB_SERIAL_JTAG_IN_EP0_RD_ADDR 输入端点 0 的读取数据地址。(RO)

Register 33.14. USB_SERIAL_JTAG_IN_EP1_ST_REG (0x002C)

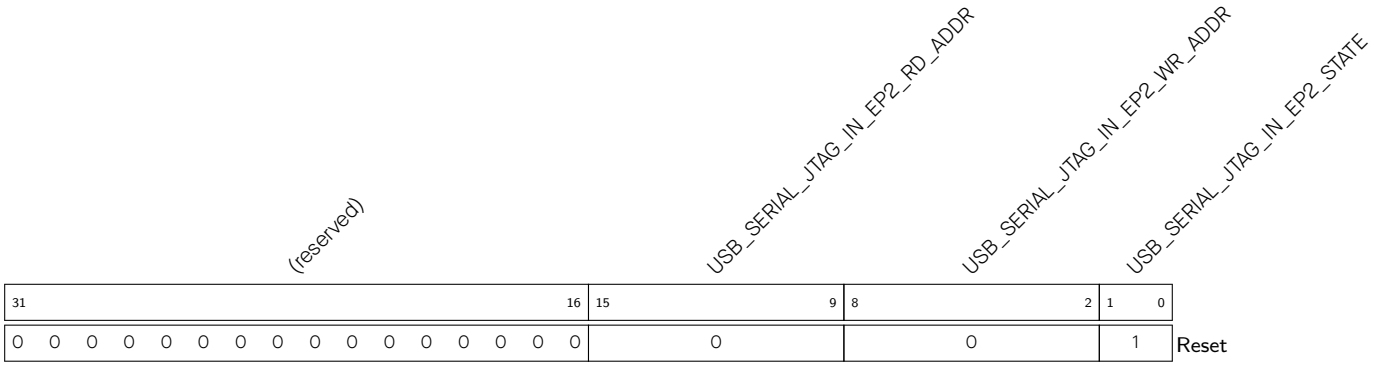


USB_SERIAL_JTAG_IN_EP1_STATE 输入端点 1 的状态。(RO)

USB_SERIAL_JTAG_IN_EP1_WR_ADDR 输入端点 1 的写入数据地址。(RO)

USB_SERIAL_JTAG_IN_EP1_RD_ADDR 输入端点 1 的读取数据地址。(RO)

Register 33.15. USB_SERIAL_JTAG_IN_EP2_ST_REG (0x0030)

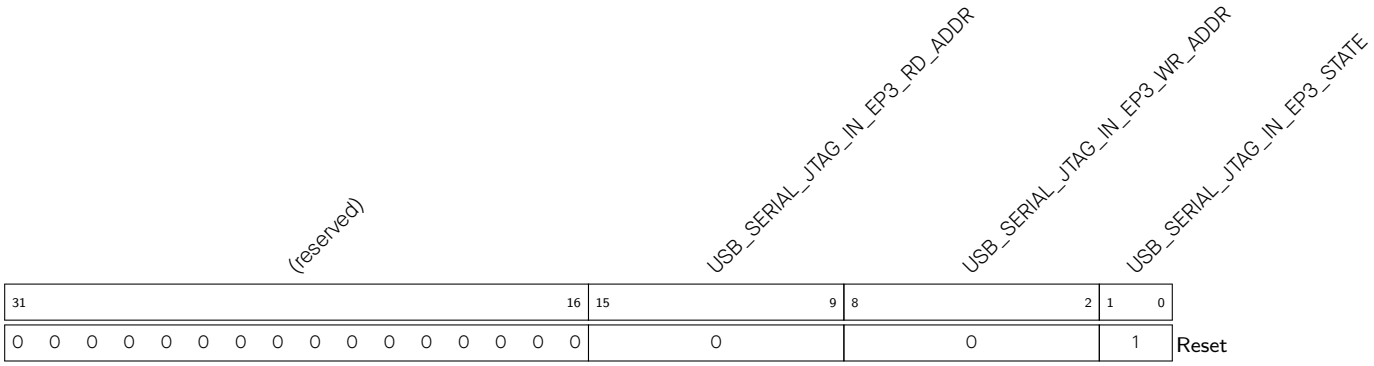


USB_SERIAL_JTAG_IN_EP2_STATE 输入端点 2 的状态。(RO)

USB_SERIAL_JTAG_IN_EP2_WR_ADDR 输入端点 2 的写入数据地址。(RO)

USB_SERIAL_JTAG_IN_EP2_RD_ADDR 输入端点 2 的读取数据地址。(RO)

Register 33.16. USB_SERIAL_JTAG_IN_EP3_ST_REG (0x0034)

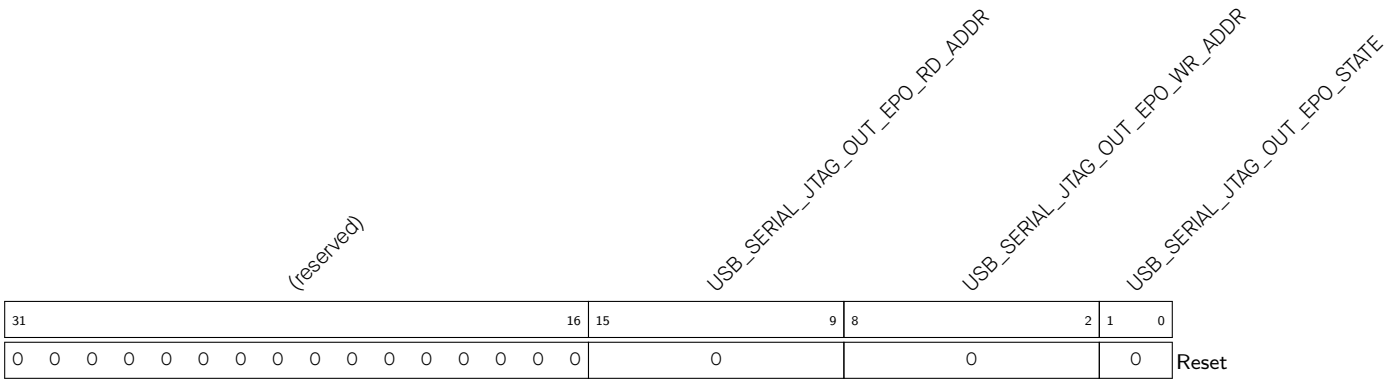


USB_SERIAL_JTAG_IN_EP3_STATE 输入端点 3 的状态。(RO)

USB_SERIAL_JTAG_IN_EP3_WR_ADDR 输入端点 3 的写入数据地址。(RO)

USB_SERIAL_JTAG_IN_EP3_RD_ADDR 输入端点 3 的读取数据地址。(RO)

Register 33.17. USB_SERIAL_JTAG_OUT_EP0_ST_REG (0x0038)

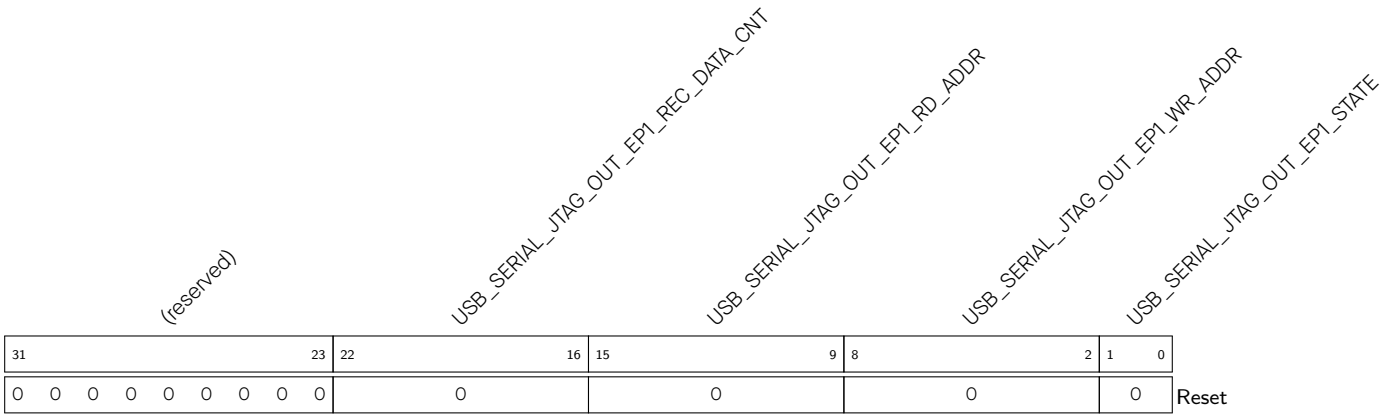


USB_SERIAL_JTAG_OUT_EP0_STATE 输出端点 0 的状态。(RO)

USB_SERIAL_JTAG_OUT_EP0_WR_ADDR 输出端点 0 的写入数据地址。当检测到 USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT 时，输出端点 0 中有 USB_SERIAL_JTAG_OUT_EP0_WR_ADDR-2 个字节数据。(RO)

USB_SERIAL_JTAG_OUT_EP0_RD_ADDR 输出端点 0 的读取数据地址。(RO)

Register 33.18. USB_SERIAL_JTAG_OUT_EP1_ST_REG (0x003C)



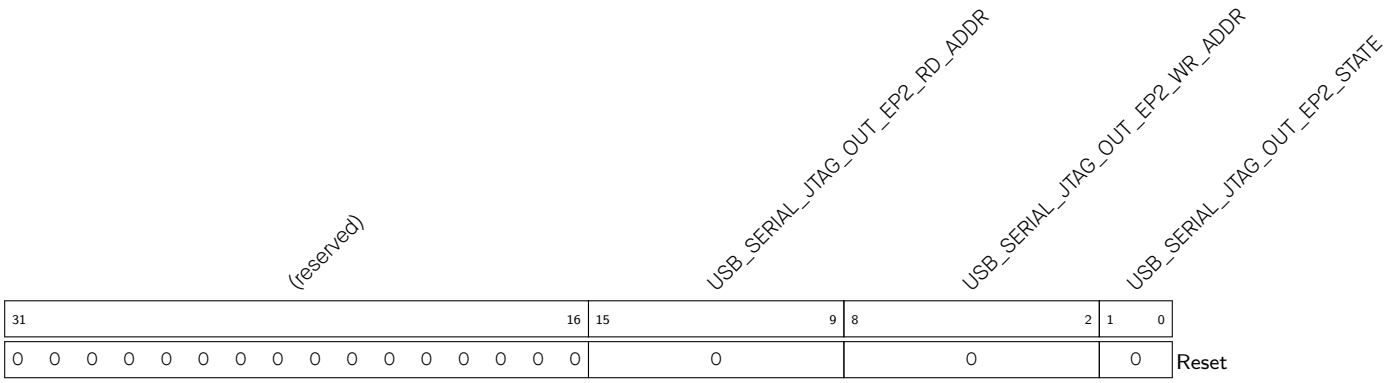
USB_SERIAL_JTAG_OUT_EP1_STATE 输出端点 1 的状态。(RO)

USB_SERIAL_JTAG_OUT_EP1_WR_ADDR OUT_RECV_PKT_INT 时，输出端点 1 中有 USB_SERIAL_JTAG_OUT_EP1_WR_ADDR-2 个字节数据。(RO)

USB_SERIAL_JTAG_OUT_EP1_RD_ADDR 输出端点 1 的读取数据地址。(RO)

USB_SERIAL_JTAG_OUT_EP1_REC_DATA_CNT 当接收到 1 个数据包时输出端点 1 中的数据计数器。(RO)

Register 33.19. USB_SERIAL_JTAG_OUT_EP2_ST_REG (0x0040)

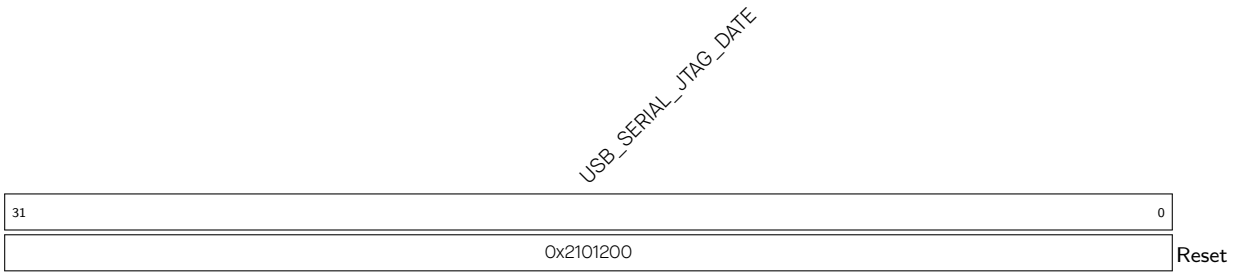


USB_SERIAL_JTAG_OUT_EP2_STATE 输出端点 2 的状态。(RO)

USB_SERIAL_JTAG_OUT_EP2_WR_ADDR 输出端点 2 的写入数据地址。当检测到 USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT 时，输出端点 2 中有 USB_SERIAL_JTAG_OUT_EP2_WR_ADDR-2 个字节数据。(RO)

USB_SERIAL_JTAG_OUT_EP2_RD_ADDR 输出端点 2 的读取数据地址。(RO)

Register 33.20. USB_SERIAL_JTAG_DATE_REG (0x0080)



USB_SERIAL_JTAG_DATE 版本控制寄存器。(R/W)

第 34 章

SD/MMC 主机控制器 (SDHOST)

34.1 概述

ESP32-S3 存储卡接口控制器提供了一个访问安全数字输入输出卡 (SDIO)、MMC 卡和 CE-ATA 设备的硬件接口，用于连接高级外围设备总线 (APB) 和外部存储设备。该控制器支持两个外部卡（卡 0 和卡 1）。所有 SD/MMC 模块接口信号都须通过 GPIO 矩阵传输至 GPIO pad。

34.2 主要特性

该模块支持以下特性：

- 支持两个外部卡
- 支持 3.0、3.01 版本 SD 存储卡标准
- 支持 4.41、4.5、4.51 版本 MMC
- 支持 1.1 版本 CE-ATA
- 高达 80 MHz 的时钟输出
- 支持 1-bit、4-bit 和 8-bit 位宽模式

说明：
在 80 MHz 工作条件下，时钟相位调节能力受限，仅支持 0° 和 180° 两种相位配置。PCB 布局设计时应充分考虑该限制，以确保时序收敛。更多细节请参考 34.10 时钟相位选择 小节。

SD/MMC 控制器连接的拓扑结构如图 34.2-1 所示。该控制器支持两组外设工作，但不支持同时工作。

34.3 SD/MMC 外部接口信号

SD/MMC 的外部接口信号主要为时钟信号 (sdhost_cclk_out_1.eg:card1)、命令信号 (sdhost_ccmd_out_1)、数据信号 (sdhost_cdata_in_1[7:0]/sdhost_cdata_out_1[7:0])，SD/MMC 控制器可通过这些外部接口信号与外部设备通信。其它信号还包括卡中断信号、卡检测信号和写保护信号等。各个信号的方向如图 34.3-1 所示。每个管脚的方向和描述如表 34.3-1 所示。

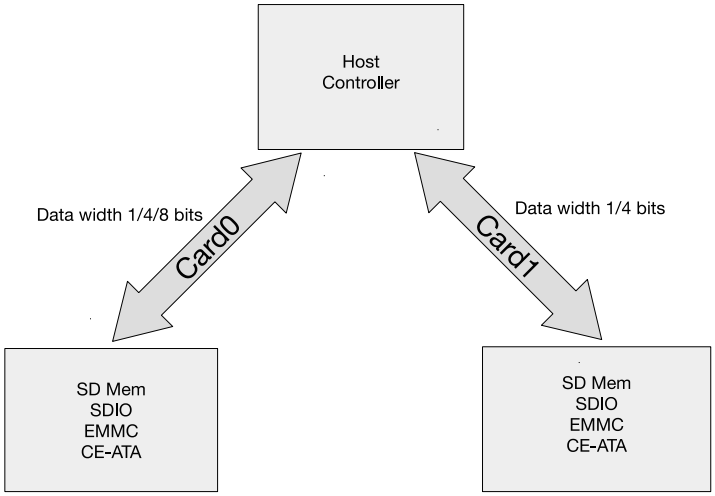


图 34.2-1. SD/MMC 控制器连接的拓扑结构

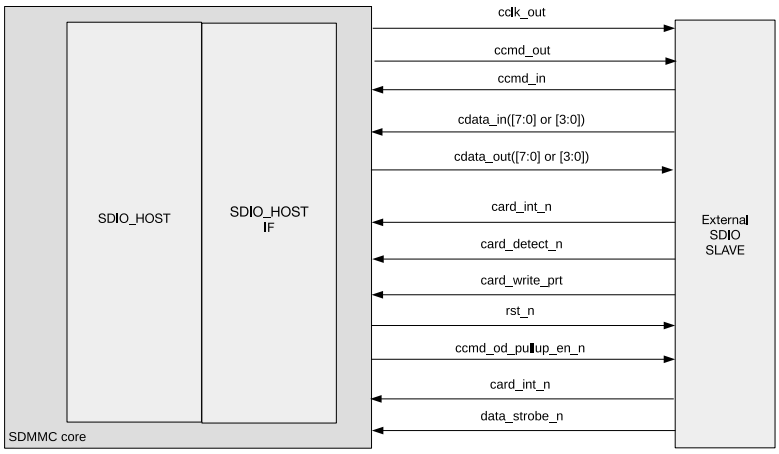


图 34.3-1. SD/MMC 控制器外部接口信号

表 34.3-1. SD/MMC 信号描述

管脚	方向	描述
sdhost_cclk_out	输出	向从机发送时钟信号
sdhost_ccmd	双向	双向命令/响应线
sdhost_cdata	双向	双向数据读/写线
sdhost_card_detect_n	输入	卡检测输入线
sdhost_card_write_prt	输入	卡写保护状态输入

34.4 功能描述

34.4.1 SD/MMC 主机控制器结构

SD/MMC 主机控制器主要由两大功能块组成，如图 34.4-1 所示，分别为：

- 总线接口单元 (BIU)：提供 APB 总线访问寄存器的接口、RAM 数据访问方式和 DMA 数据读写操作
- 卡接口单元 (CIU)：处理外部存储卡的接口协议，控制时钟。

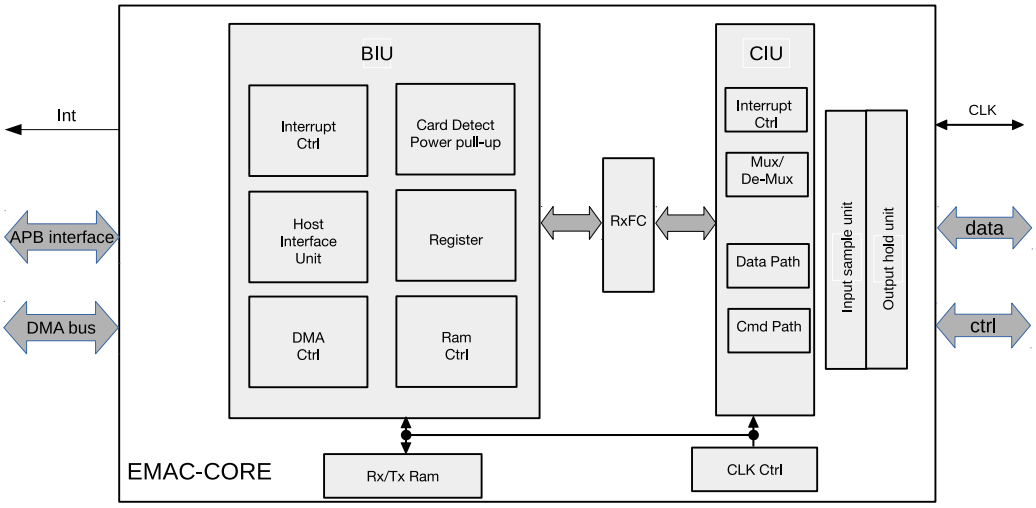


图 34.4-1. SDIO 主机结构框图

34.4.1.1 总线接口单元 (BIU)

该模块提供了通过主机接口单元 (HIU) 访问寄存器和 RAM 数据的方式。除此之外，它通过 DMA 接口提供独立的 RAM 数据访问。它通过 DMA 接口访问 Memory 中的数据。图 34.4-1 描述了该模块的内部结构。图 34.10-1 描述了时钟相位选择。

BIU 支持以下功能：

- 主机接口
- DMA 接口
- 中断控制
- 寄存器访问
- FIFO 访问
- 上电/上拉控制和卡检测

34.4.1.2 卡接口单元 (CIU)

该模块实现卡专用接口协议。在 CIU 中，命令通路 (Cmd Path) 控制单元和数据通路 (Data Path) 控制单元将控制器分别连接到 SD/MMC/CE-ATA 卡的命令接口和数据接口。CIU 还提供时钟控制。图 34.4-1 描述了 CIU 的内部结构。

CIU 由以下主要功能模块组成：

- 命令通路
- 数据通路
- SDIO 中断控制
- 时钟控制
- Mux/De-Mux 单元

34.4.2 命令通路

命令通路具有以下功能：

- 配置时钟参数
- 配置卡命令参数
- 向卡总线发送命令 (sdhost_ccmd_out)
- 从卡总线接收响应 (sdhost_ccmd_in)
- 向 BIU 发送响应
- 在命令线上发送 P-bit

命令通路状态机如图 34.4-2 所示。

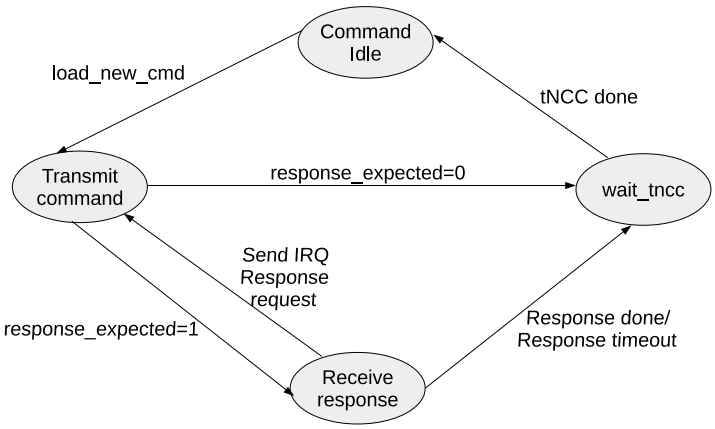


图 34.4-2. 命令通路状态机

34.4.3 数据通路

发送时，数据通路模块会取出 RAM 中的数据，并将其发送至 sdhost_cdata_out；接收时，数据通路会接收 sdhost_cdata_in 上的数据，并将其导入 RAM 中。数据发送命令未运行时，数据通路将加载新的数据参数，即希望发送的数据、读/写数据发送、流/块发送、块大小、字节数、卡类型、超时寄存器等。

如果在命令寄存器 (SDHOST_CMD_REG) 中置了 SDHOST_DATA_EXPECTED 位，则新命令为数据传输命令，数据通路将执行以下操作之一：

- 若 SDHOST_READ_WRITE 位为 1，发送数据
- 若 SDHOST_READ_WRITE 位为 0，接收数据

34.4.3.1 数据发送

接收到数据写入命令后，该模块将在两个时钟周期开始发送数据。即使命令通路在响应信息中检测到响应错误或循环冗余检查 (CRC) 错误，数据发送也会正常进行。但是，如果直到响应超时仍没有从卡接收到响应，则不发送数据。根据 SDHOST_CMD_REG 寄存器中 SDHOST_TRANSFER_MODE 位的值，数据发送状态机将数据以流或块的形式加至卡数据总线上。数据发送状态机如图 34.4-3 所示。

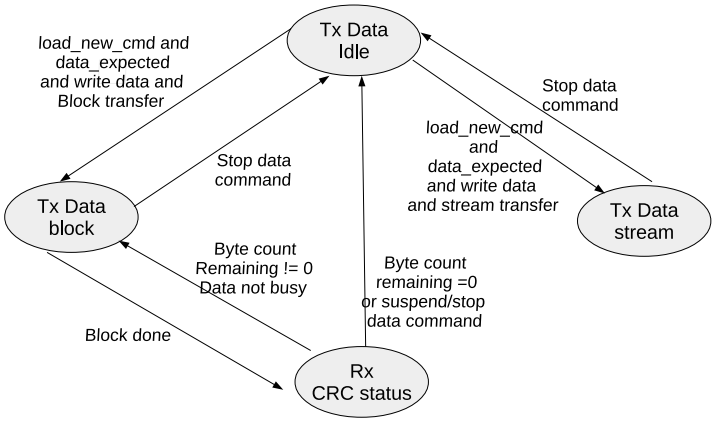


图 34.4-3. 数据发送状态机

34.4.3.2 数据接收

数据读取命令完成两个时钟周期后，该模块将开始接收数据。即使命令通路检测到响应错误或 CRC 错误，数据发送也会正常进行。如果一直未从卡接收到响应而发生了响应超时，则 BIU 无法接收到 CIU 数据发送完成的信号。如果 CIU 发送的命令是卡所禁止的非法操作，那么将无法开始读取从卡发送的数，且 BIU 也不会接收到 CIU 数据发送完成的信号。

若在数据超时前未接收到数据，数据通路将向 BIU 发出数据超时信号并结束数据传输。根据 SDHOST_CMD_REG 寄存器中的 SDHOST_TRANSFER_MODE 位的值，数据接收状态机以流或块的形式从卡数据总线获取数据。数据接收状态机如图 34.4-4 所示。

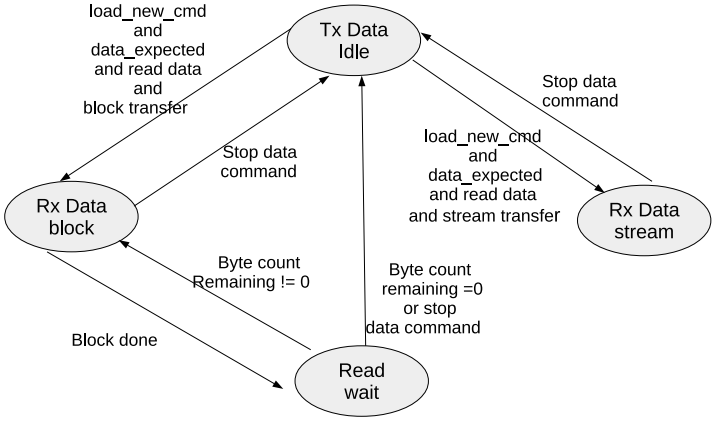


图 34.4-4. 数据接收状态机

34.5 CIU 操作的软件限制

- 一次只能选择一个卡执行命令或发送数据。例如，当向某张卡发送数据或从这张卡接收数据时，不可以给另一张卡发送其它新的命令。但是，可以将新的命令发送到同一张卡上，使其读取设备状态或停止数据传输。
- 一次只能发出一个数据传输命令。
- 在开放式写卡操作期间，如果由于 RAM 为空导致卡时钟停止，那么软件必须先将数据填充到 RAM 中并启动卡时钟，然后才可以向卡发出一个停止/中止命令。
- 在 SDIO/Combo 卡数据传输期间，如果卡功能暂停，并且软件想要恢复所暂停的数据传输，则其必须先重置 RAM（置位 SDHOST_FIFO_RESET）并启动恢复命令，这和启动一个新的数据传输命令相似。
- 要在进行卡数据传输中发出卡复位命令（CMD0、CMD15 或 CMD52_reset），软件必须在 SDHOST_CMD_REG 寄存器上置位 SDHOST_STOP_ABORT_CMD，保证 CIU 可以在发出卡复位命令后停止数据传输。
- 当在 SDHOST_RINTSTS_REG 寄存器中设置数据结束位错误时，CIU 无法控制 SDIO 中断。因此在该情况下，软件应忽略 SDIO 中断，并向卡发出停止/中止命令使其停止发送数据。
- 在一次读卡过程中，如果由于 RAM 已满而导致卡时钟停止，那么软件将至少读取两个 RAM 地址来重启卡时钟。
- 在一次命令/数据传输中只能选取一个 CE-ATA 设备。例如，当一个 CE-ATA 设备已经在传输数据，则其它 CE-ATA 设备不可传输新命令。

- 使能 CE-ATA 设备的中断 (nIEN=0) 后，如果该设备已经有正在执行的 SDHODT_RW_BLK 命令，则不可以再向这一设备发送新的 SDHODT_RW_BLK 命令。只有在等待 CCS 时可以发送 CCSD。
- 但是，如果一个 CE-ATA (nIEN=1) 设备未使能，则可以向同一设备发送新的命令来读取状态信息。
- CE-ATA 设备不支持开放式数据传输。
- CE-ATA 数据传输不支持 sdhost_send_auto_stop 信号（禁止软件置位 SDHOST_SEND_AUTO_STOP）。

置位命令起始位后，在发送出命令之前以下寄存器的值不可改变：

- CMD — 命令
- CMDARG — 命令参数
- BYTCNT — 字节计数
- BLKSIZ — 块大小
- CLKDIV — 时钟分频器
- CKLENA — 时钟使能
- CLKSRC — 时钟源
- TMOUT — 超时
- CTYPE — 卡类型

34.6 收发数据 RAM

RAM 子模块是一个收发数据缓冲区，分为接收和发送两个单元。也可通过 CPU 和 DMA 实现数据的收发，从而进行读写操作，可参阅章节 34.8。

34.6.1 TX RAM 模块

可通过两种方式使能写入操作：DMA 和 CPU 读写。

如果使能 SDIO 发送，那么可以通过 APB 接口将数据写入到 TX RAM 模块中。此时，数据将通过 APB 接口直接从 CPU 写入到 SDHOST_BUFFIFO_REG 寄存器中。

除此之外，还可以通过 DMA 实现数据发送。

34.6.2 RX RAM 模块

可通过两种方式使能读取操作：DMA 和 CPU 读写。

当数据通路接收到数据时，该数据将被写入 RX RAM 中。可在读取端通过 APB 读取这些数据，APB 可直接读取寄存器 SDHOST_BUFFIFO_REG。

除此之外，还可以通过 DMA 实现数据接收。

34.7 DMA 链表环

链表环中的每个链表模块都由两部分组成：链表和一个数据缓冲区。也就是说，每一个链表都指向一个独特的数据缓冲区，同时还连接至下一个链表模块。链表环结构如图 34.7-1 所示。

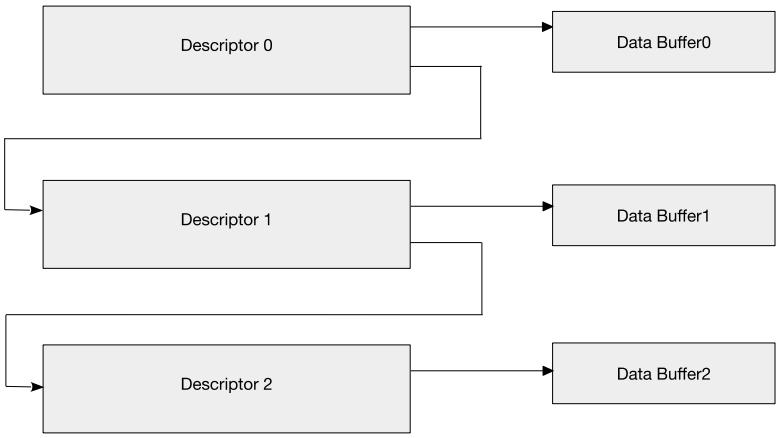


图 34.7-1. 链表环结构

34.8 DMA 链表结构

每个链表由 4 个字组成，如图 34.8-1 所示。表 34.8-1、表 34.8-2、表 34.8-3、表 34.8-4 为这 4 个字的描述。

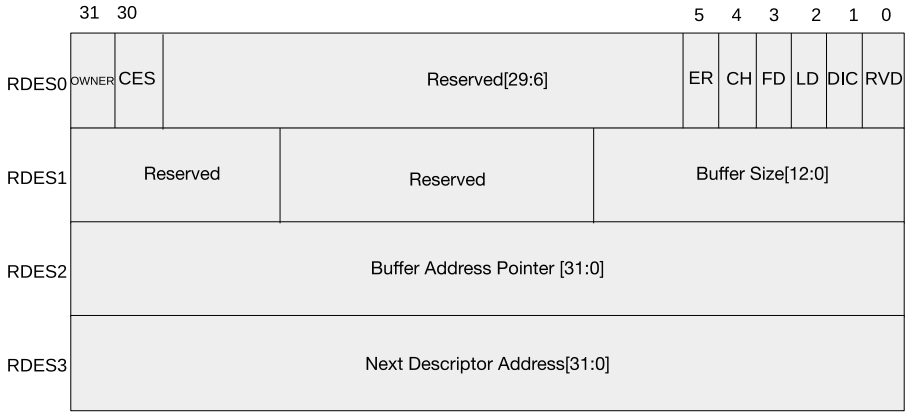


图 34.8-1. 链表结构

DESO 单元包含控制和状态信息。

表 34.8-1. DESO 单元描述

位	名称	描述
31	OWNER	置位时，表明该描述符允许的操作者为 DMA 控制器。该位被重置时，表明该描述符允许的操作者为主机。DMA 控制器在完成数据传输后清除该位。
30	CES (Card Error Summary)	这些错误位显示卡的读/写状态。 以下各位也存储于 SDHOST_RINTSTS_REG 中，为或运算： • EBE：结束位错误 • RTO：响应超时 • RCRC：响应 CRC • SBE：起始位错误 • DRTO：读取数据超时 • DCRC：对接收数据进行循环冗余校验 • RE：响应错误
29:6	保留	保留
5	ER (End of Ring)	置位时，表示当前描述符为数据包的最后一个描述符。此时，DMA 控制器返回到链表的基地址，创建一个链表环。
4	CH (Second Address Chained)	置位时，表示该描述符中的第二个地址为下一描述符的地址。此时，BS2 (DES1[25:13]) 应全部归零。
3	FD (First Descriptor)	置位时，表示该描述符内包含了数据的第一个缓冲区。若该缓冲区大小为 0，则下一个描述符内为数据的开始。
2	LD (Last Descriptor)	该位与 DMA 传输的最后一个数据块相关。置位时，表示该描述符指向的缓冲区是数据的最后一个缓冲区。在该描述符完成之后，剩余字节数为 0。也就是说，带有被置位的 LD 位的描述符完成之后，剩余字节数应为 0。
1	DIC (Disable Interrupt on Completion)	置位时，为了保留在该描述符指向的缓冲区中结束的数据，该位将阻止置位 DMA 状态寄存器 (IDSTS) 上 TI/RI 位。
0	保留	保留

DES1 单元包含缓冲区大小。

表 34.8-2. DES1 单元描述

位	名称	描述
31:26	保留	保留
25:13	保留	保留
12:0	BS（缓冲区大小）	表示数据缓冲区的字节大小。该数值必须为 4 的倍数。否则，其大小将无法被定义成功。该字段不可设置为 0。

DES2 单元包含指向数据缓冲区的地址指针。

表 34.8-3. DES2 单元描述

位	名称	描述
31:0	Buffer Address Pointer（数据缓冲区地址指针）	表示数据缓冲区的物理地址，须按字对齐。

如果当前的描述符不是链表环中的最后一个，则 DES3 单元包含指向下一个描述符的地址指针。

表 34.8-4. DES3 单元描述

位	名称	描述
31:0	Next Descriptor Address（下一个链表地址）	如果置位 CH (DES0[4])，则该位中有指向包含下一个描述符位置的物理内存的指针。 如果该描述符不是链表环结构中的最后一个描述符，则下一个描述符的地址指针必须满足 DES3[1:0] = 0。

34.9 初始化

34.9.1 DMA 控制器初始化

DMA 控制器初始化过程如下：

1. 向 DMA 控制器的总线模式寄存器 (SDHOST_BMOD_REG) 写入数据，设置主机总线访问参数。
2. 向 DMA 控制器的中断使能寄存器 (SDHOST_IDINTEN_REG) 写入数据，屏蔽不必要的中断类型。
3. 软件驱动器创建发送链表或接收链表。然后向 DMA 控制器链表基地址寄存器 (SDHOST_DBADDR_REG) 中写入链表的起始地址。
4. DMA 控制器引擎尝试从链表中获取描述符。

34.9.2 DMA 控制器数据发送初始化

DMA 控制器发送数据的过程如下：

1. 主机设置发送数据的描述符单元 (DES0 ~ DES3)，置位 OWNER 位 (DES0[31])，同时准备数据缓冲区。
2. 主机在 BIU 的命令 (CMD) 寄存器中写入数据命令。
3. 主机设置所需的数据发送阈值（在 SDHOST_FIFOTH_REG 寄存器中的 SDHOST_TX_WMARK 字段进行）。
4. DMA 控制器引擎读取描述符并检查 OWNER 位。如果 OWNER 位未置位，表明该描述符所允许的操作者为主机。此时，DMA 控制器进入挂起状态，并在 SDHOST_IDSTS_REG 寄存器中产生禁能描述符中断。然后，主机需在 SDHOST_PLDMND_REG 中写入任意值来释放 DMA 控制器资源。
5. DMA 控制器等待 DHOST_RINTSTS_REG 寄存器中的命令完成 (CD) 位，如果 BIU 无报错，则表明数据发送已完成。

6. 然后，DMA 控制器引擎等待 BIU 发送的 DMA 总线请求，该请求将基于配置的数据发送阈值生成。对于使用突发传输不能访问的最后一个字节，则在 AHB 总线上执行单次发送。
7. DMA 控制器从主机存储器的数据缓冲区获取发送数据，并通过 RAM 将其发送至卡中。
8. 当数据跨越多个描述符时，DMA 控制器将获取下一个描述符，并继续使用后面的描述符进行后续操作。最后一个描述符位将显示该数据是否跨越多个描述符。
9. 当数据发送完成且 SDHOST_IDSTS_TI 位已使能，将通过置位 SDHOST_IDSTS_TI 将状态信息更新至寄存器 SDHOST_IDSTS_REG。同时，DMA 控制器将通过对 DESO 执行写操作来清除 OWNER 位。

34.9.3 DMA 控制器数据接收初始化

DMA 控制器接收数据的过程如下：

1. 主机设置接收数据的描述符单元 (DESO ~ DES3)，置位 OWNER 位 (DESO[31])。
2. 主机在 BIU 的命令寄存器中写入读数据命令。
3. 主机设置所需的数据接收阈值（在 SDHOST_FIFOTH_REG 寄存器中的 SDHOST_RX_WMARK 字段进行）。
4. DMA 控制器引擎读取描述符并检查 OWNER 位。如果 OWNER 位未置位，表明该描述符所允许的操作者为主机。此时，DMA 控制器进入挂起状态，并在 SDHOST_IDSTS_REG 寄存器中产生禁能描述符中断。然后，软件需在 SDHOST_PLDMND_REG 中写入任意值释放 DMA 控制器资源。
5. DMA 控制器等待命令完成位 (CD)，如果 BIU 无报错，则表明数据接收已完成。
6. DMA 控制器引擎等待 BIU 发送的 DMA 总线请求。该请求将基于配置的数据接收阈值生成。对于使用突发传输不能访问的最后一个字节，则在 AHB 总线上执行单次传输。
7. DMA 控制器从 RAM 获取数据，并将其发送至主机存储器。
8. 当数据跨越多个描述符时，DMA 控制器将获取下一个描述符，并继续使用后面的描述符进行后续操作。最后一个描述符位将显示该数据是否跨越多个描述符。
9. 当数据接收完成且 SDHOST_IDSTS_RI 位已使能，将通过置位 SDHOST_IDSTS_RI 将状态信息更新至寄存器 SDHOST_IDSTS_REG。同时，DMA 控制器将通过对 DESO 执行写操作来清除 OWNER 位。

34.10 时钟相位选择

如果输入数据信号或输出数据信号的建立时间的时序不符合要求，用户可以参照下图 34.10-1 改变时钟相位。

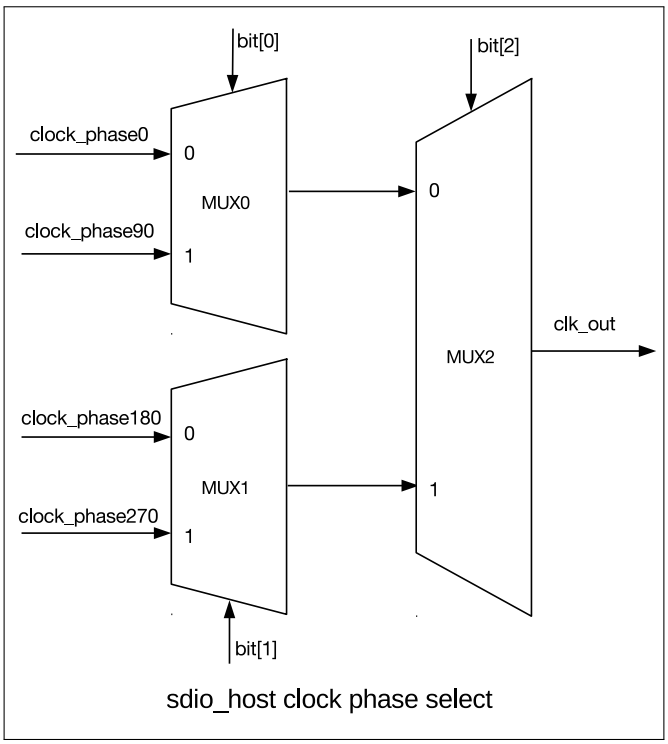


图 34.10-1. 时钟相位选择

可通过设置寄存器 SDHOST_CLK_DIV_EDGE_REG 来解决时序问题。例如，清除 CCLKIN_EDGE_DRV_SEL 位发送相位 0 中的输出数据，然后置位 CCLKIN_EDGE_SAM_SEL 选择 90 度相位采样 SDIO 从机中的数据。如果此时仍有时序问题，还可通过向 CCLKIN_EDGE_SAM_SEL 写入 4 或 6 分别选择 180 度或 270 度相位对 SDIO 从机进行数据采样。

有关时钟相位选择寄存器 SDHOST_CLK_DIV_EDGE_REG 的说明，请见寄存器章节 34.13。

表 34.10-1. SDHOST 时钟相位选择

时钟相位	phase_select 数值
0	0
90	1
180	4
270	6

34.11 中断

中断可在多个事件中产生。SDHOST_IDSTS_REG 寄存器中包含所有可能导致中断的位。SDHOST_IDINTEN_REG 寄存器中包含所有可导致中断的事件的使能位。

SDHOST_IDSTS_REG 寄存器中有两组中断汇总：正常中断汇总 (第 8 位：SDHOST_IDSTS_NIS) 和异常中断汇总 (第 9 位：SDHOST_IDSTS_AIS)。置位相应的位，可以清除中断。当某组中的所有使能中断都被清除，相应的汇总位将被清零。当两组的汇总位都被清零，连接 CPU 的中断信号将撤销（停止发送信号）。

中断不排序，如果中断在驱动程序响应之前发生，则不会产生其他中断。例如，SDHOST_IDSTS_RI 表示一个或多个数据已发送至主机缓冲区。

并发事件只会产生一个中断。驱动程序须扫描 SDHOST_IDSTS_REG 寄存器来查找导致中断的原因。

34.12 寄存器列表

本小节的所有地址均为相对于 SD/MMC Host Controller 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

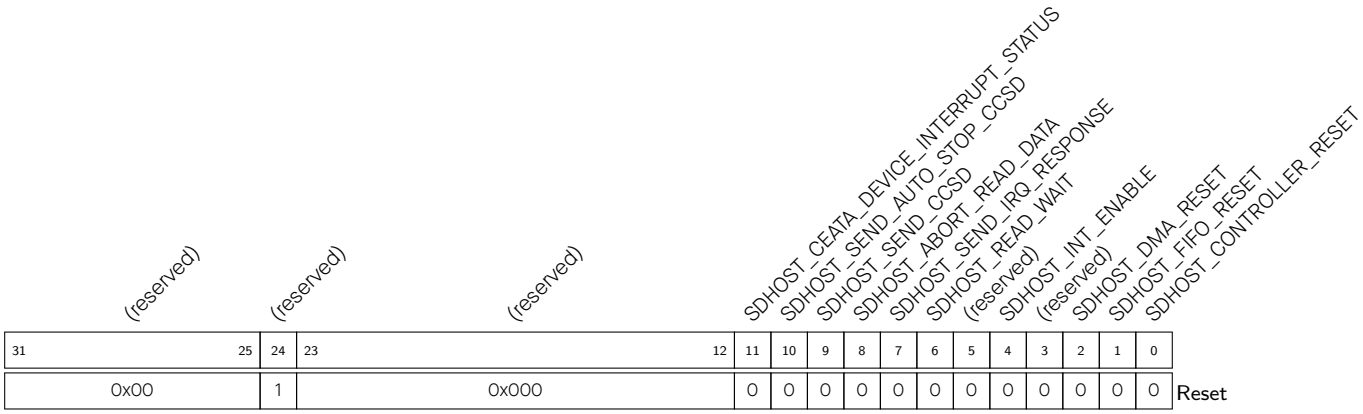
名称	描述	地址	访问
SDHOST_CTRL_REG	控制寄存器	0x0000	R/W
SDHOST_CLKDIV_REG	时钟分频器配置寄存器	0x0008	R/W
SDHOST_CLKSRC_REG	时钟源选择寄存器	0x000C	R/W
SDHOST_CLKENA_REG	时钟使能寄存器	0x0010	R/W
SDHOST_TMOUT_REG	数据和响应超时配置寄存器	0x0014	R/W
SDHOST_CTYPE_REG	卡总线宽度配置寄存器	0x0018	R/W
SDHOST_BLKSIZ_REG	卡数据块大小配置寄存器	0x001C	R/W
SDHOST_BYTCNT_REG	数据传输长度配置寄存器	0x0020	R/W
SDHOST_INTMASK_REG	SDIO 中断屏蔽寄存器	0x0024	R/W
SDHOST_CMDARG_REG	指令参数数据寄存器	0x0028	R/W
SDHOST_CMD_REG	指令和启动配置寄存器	0x002C	R/W
SDHOST_RESP0_REG	响应数据寄存器	0x0030	RO
SDHOST_RESP1_REG	长响应数据寄存器 0	0x0034	RO
SDHOST_RESP2_REG	长响应数据寄存器 1	0x0038	RO
SDHOST_RESP3_REG	长响应数据寄存器 2	0x003C	RO
SDHOST_MINTSTS_REG	屏蔽的中断状态寄存器	0x0040	RO
SDHOST_RINTSTS_REG	原始中断状态寄存器	0x0044	R/W
SDHOST_STATUS_REG	SD/MMC 状态寄存器	0x0048	RO
SDHOST_FIFOTH_REG	FIFO 配置寄存器	0x004C	R/W
SDHOST_CDETECT_REG	卡检测寄存器	0x0050	RO
SDHOST_WRTPRT_REG	卡写保护状态寄存器	0x0054	RO
SDHOST_TCBCNT_REG	传输字节计数寄存器	0x005C	RO
SDHOST_TBBCNT_REG	传输字节计数寄存器	0x0060	RO
SDHOST_DEBNCE_REG	去抖过滤器时间配置寄存器	0x0064	R/W
SDHOST_USRID_REG	用户 ID（暂存）寄存器	0x0068	R/W
SDHOST_VERID_REG	版本 ID（暂存）寄存器	0x006C	RO
SDHOST_HCON_REG	硬件特性寄存器	0x0070	RO
SDHOST_UHS_REG	UHS-1 寄存器	0x0074	R/W
SDHOST_RST_N_REG	卡重置寄存器	0x0078	R/W
SDHOST_BMOD_REG	突发模式传输配置寄存器	0x0080	R/W
SDHOST_PLDMND_REG	轮询命令配置寄存器	0x0084	WO
SDHOST_DBADDR_REG	链表基地址寄存器	0x0088	R/W
SDHOST_IDSTS_REG	IDMAC 状态寄存器	0x008C	R/W
SDHOST_IDINTEN_REG	IDMAC 中断使能寄存器	0x0090	R/W
SDHOST_DSCADDR_REG	主机描述符地址指针寄存器	0x0094	RO
SDHOST_BUFADDR_REG	主机缓冲区地址指针寄存器	0x0098	RO
SDHOST_CARDTHRCTL_REG	卡阈值控制寄存器	0x0100	R/W

名称	描述	地址	访问
SDHOST_EMMCDDR_REG	双倍数据速率寄存器	0x010C	R/W
SDHOST_ENSHIFT_REG	相位偏移使能寄存器	0x0110	R/W
SDHOST_BUFFIFO_REG	CPU 通过 FIFO 读写发送数据	0x0200	R/W
SDHOST_CLK_DIV_EDGE_REG	时钟相位选择寄存器	0x0800	R/W

34.13 寄存器

本小节的所有地址均为相对于 SD/MMC Host Controller 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 34.1. SDHOST_CTRL_REG (0x0000)



SDHOST_CEATA_DEVICE_INTERRUPT_STATUS 软件应在 CE-ATA 设备上电复位或其他复位后写此位。CE-ATA 设备的中断通常被禁能 (nIEN = 1)。如果主机使能中断，则软件应将此位置为 1。(R/W)

SDHOST_SEND_AUTO_STOP_CCSD 始终将 SDHOST_SEND_AUTO_STOP_CCSD 和 SDHOST_SEND_CCSD 一起置位，不能分开单独置位。置位时，SD/ MMC 自动发送内部生成的 STOP 命令 (CMD12) 给 CE-ATA 设备。发送 STOP 命令后，SDHOST_RINTSTS_REG 里的 Auto Command Done (ACD) 位被置为 1，如果 ACD 中断没有屏蔽，则用于主机的中断将会生成。发送 Command Completion Signal Disable (CCSD) 后，SD/ MMC 自动清零 SDHOST_SEND_AUTO_STOP_CCSD 位。(R/W)

SDHOST_SEND_CCSD 置位时，SD/MMC 发送 CCSD 给 CE-ATA 设备。只有在当前命令等待 CCS 并且 CE-ATA 设备的中断使能时软件才会将此位置位。一旦 CCSD 模式发送给设备，SD/MMC 就会自动清零 SDHOST_SEND_CCSD 位。如果 ACD 中断没有屏蔽，则寄存器 SDHOST_RINTSTS_REG 里的 Command Done (CD) 位会被置为 1，且主机的中断将会生成。

说明：一旦 SDHOST_SEND_CCSD 被置位，需要两个卡时钟周期来驱动 CMD 线上的 CCSD。因此，即使设备已经发送 CCS 信号，在边界条件内 CCSD 信号可能会发送给 CE-ATA 设备。(R/W)

SDHOST_ABORT_READ_DATA 在读操作期间，挂起命令发送后，软件会轮询卡并找出挂起事件是从什么时候开始的。一旦挂起事件开始，软件会复位数据状态机，此时状态机正在等待下一个数据块。当数据状态机复位为空闲状态时，此位自动清零。(R/W)

SDHOST_SEND_IRQ_RESPONSE 响应发送后此位自动清零。为了等待 MMC 卡发送的中断，主机发送 CMD40，然后等待 MMC 卡的中断响应。同时，如果主机想要 SD/MMC 退出等待中断的状态，则会将此位置 1，此时 SD/MMC 命令状态机发送 CMD40 响应并返回空闲状态。(R/W)

见下页...

Register 34.1. SDHOST_CTRL_REG (0x0000)

接上页...

SDHOST_READ_WAIT 发送读操作等待给 SDIO 卡。(R/W)

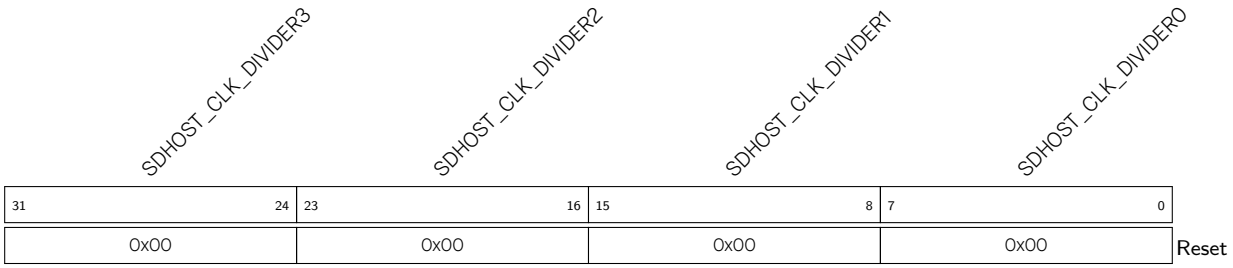
SDHOST_INT_ENABLE 全局中断使能/禁能位。0：禁能；1：使能。(R/W)

SDHOST_DMA_RESET 要复位 DMA 接口，软件应将此位置为 1。两个 AHB 时钟后此位自动清零。(R/W)

SDHOST_FIFO_RESET 要复位 FIFO，软件应将此位置为 1。复位操作结束后自动清零。
说明：清零后，再过两个系统时钟周期和同步延迟（两个卡时钟周期），FIFO 指针将会退出复位。(R/W)

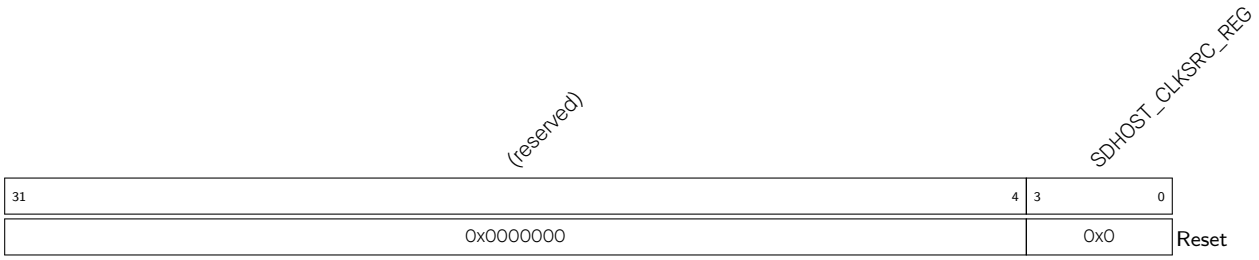
SDHOST_CONTROLLER_RESET 要复位控制器，软件应将此位置为 1。两个 AHB 时钟周期和两个 sdhost_cclk_in 时钟周期后此位自动清零。(R/W)

Register 34.2. SDHOST_CLKDIV_REG (0x0008)



SDHOST_CLK_DIVIDER*m* Clock divider-*m* 的值。时钟分频系数为 2**n*，*n*=0 旁路分频器（分频系数为 1）。例如，值为 1 代表分频系数为 2*1 = 2，值为 0xFF 代表分频系数为 2*255 = 510，以此类推。m 的取值范围为 0 至 3。(R/W)

Register 34.3. SDHOST_CLKSRC_REG (0x000C)



SDHOST_CLKSRC_REG 时钟分频源可以支持 2 个 SD 卡。每个卡分配有两个位。例如，bit[1:0] 分配给卡 0，bit[3:2] 分配给卡 1。卡 0 根据位值将时钟分频器 [0:3] 的输出信号传输给 cclk_out[1:0] 管脚。(R/W)

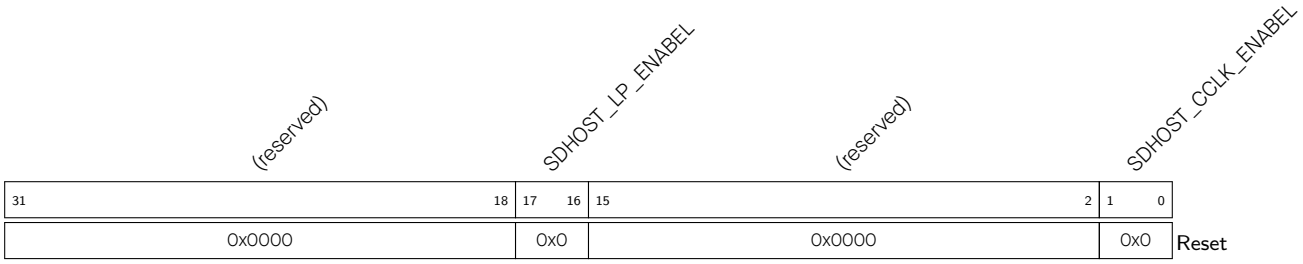
00：时钟分频器 0；

01：时钟分频器 1；

10：时钟分频器 2；

11：时钟分频器 3。

Register 34.4. SDHOST_CLKENA_REG (0x0010)



SDHOST_LP_ENABLE 当卡处于 IDLE 状态时，把时钟停掉。每个卡分配有一个位。(R/W)

0：时钟禁用；

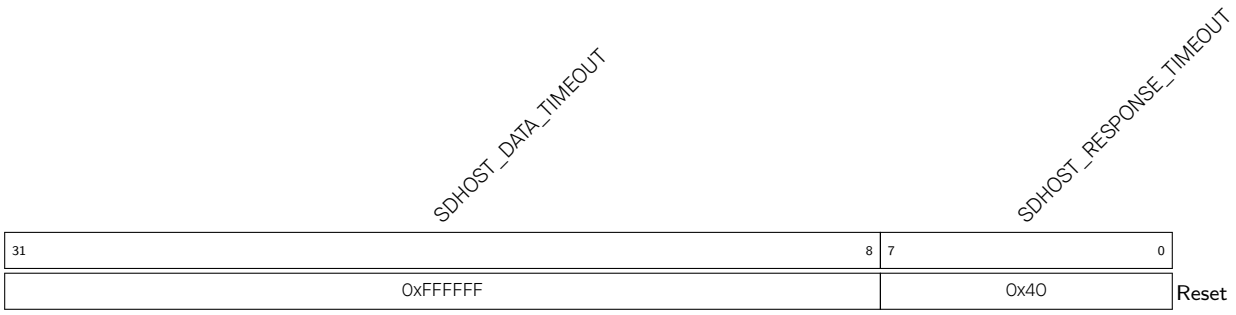
1：时钟使能。

SDHOST_CCLK_ENABLE 时钟使能控制可支持 2 个 SD 卡时钟和一个 MMC 卡时钟。每个卡分配有一个位。(R/W)

0：时钟禁用；

1：时钟使能。

Register 34.5. SDHOST_TMOUT_REG (0x0014)

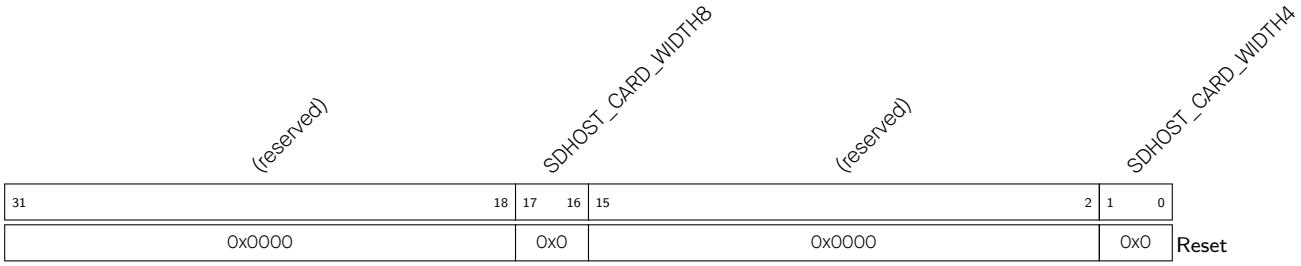


SDHOST_DATA_TIMEOUT 此位用于设置卡数据读取超时的值，还可以用来设置主机超时的定时器的值。只有当卡时钟停止后超时计数器才开始启动。此位的值以卡输出时钟数来表示，即被选取卡的 sdhost_cclk_out 时钟。(R/W)

说明：如果超时值是 100 ms 左右，则应该使用软件定时器。这种情况下，读数据超时中断应该被禁能。

SDHOST_RESPONSE_TIMEOUT 此位用于设置响应超时的值，以卡输出时钟数来表示，即 sdhost_cclk_out 时钟。(R/W)

Register 34.6. SDHOST_CTYPE_REG (0x0018)



SDHOST_CARD_WIDTH8 每个卡一个位，表明卡是否处于 8-bit 模式。(R/W)

0：非 8-bit 模式；

1：8-bit 模式。

Bit[17:16] 分别对应卡 [1:0]。

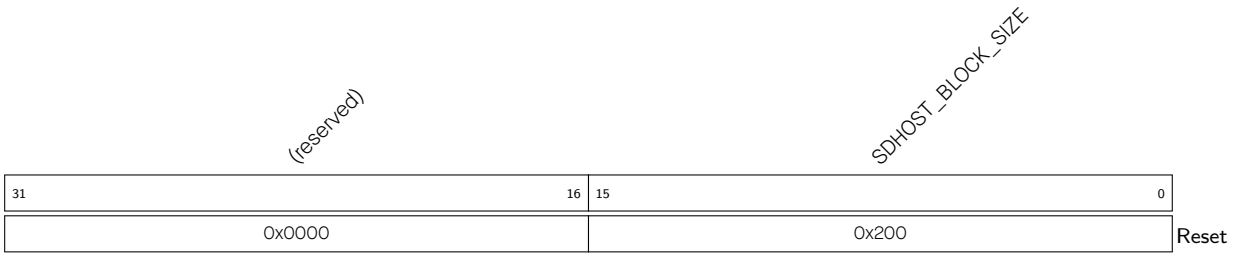
SDHOST_CARD_WIDTH4 每个卡一个位，表明卡处于 1-bit 模式还是 4-bit 模式。(R/W)

0：1-bit 模式；

1：4-bit 模式。

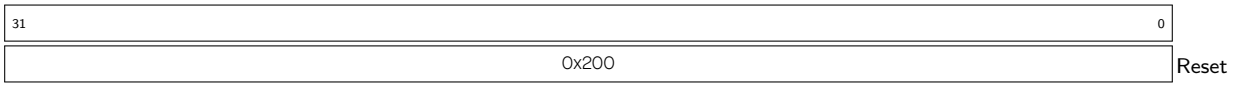
Bit[1:0] 分别对应卡 [1:0]。

Register 34.7. SDHOST_BLKSIZ_REG (0x001C)



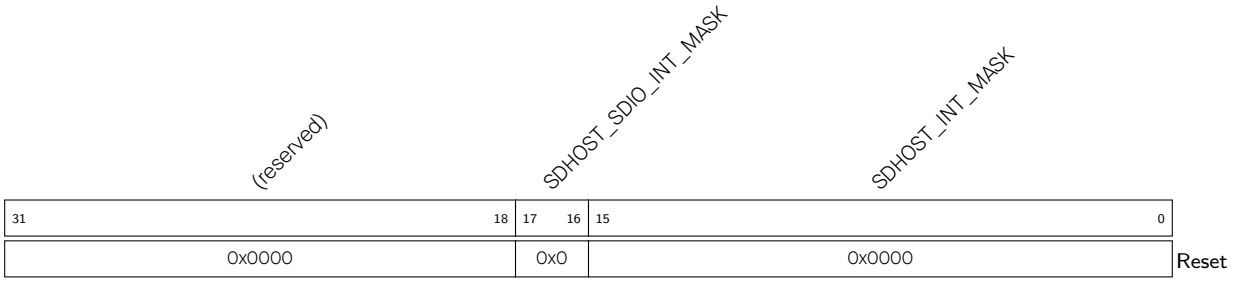
SDHOST_BLOCK_SIZE 块大小。(R/W)

Register 34.8. SDHOST_BYTCNT_REG (0x0020)



SDHOST_BYTCNT_REG 表明传输的字节数。对于块的传输，值应为块大小的整数倍。对于未定义字节长度的数据传输，字节计数应该设置为 0。当字节计数为 0 时，主机应当明确发送停止/终止命令来停止数据传输。(R/W)

Register 34.9. SDHOST_INTMASK_REG (0x0024)

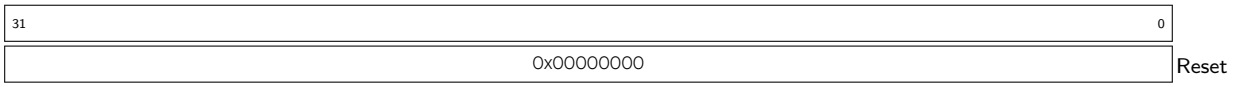


SDHOST_SDIO_INT_MASK SDIO 中断的屏蔽位，每个卡一个位。Bit[17:16] 分别对应卡 [1:0]。当被屏蔽时，SDIO 中断的检测被禁能。0 屏蔽中断，1 使能中断。(R/W)

SDHOST_INT_MASK 这些位用于屏蔽不想要的中断。0 屏蔽中断，1 使能中断。(R/W)

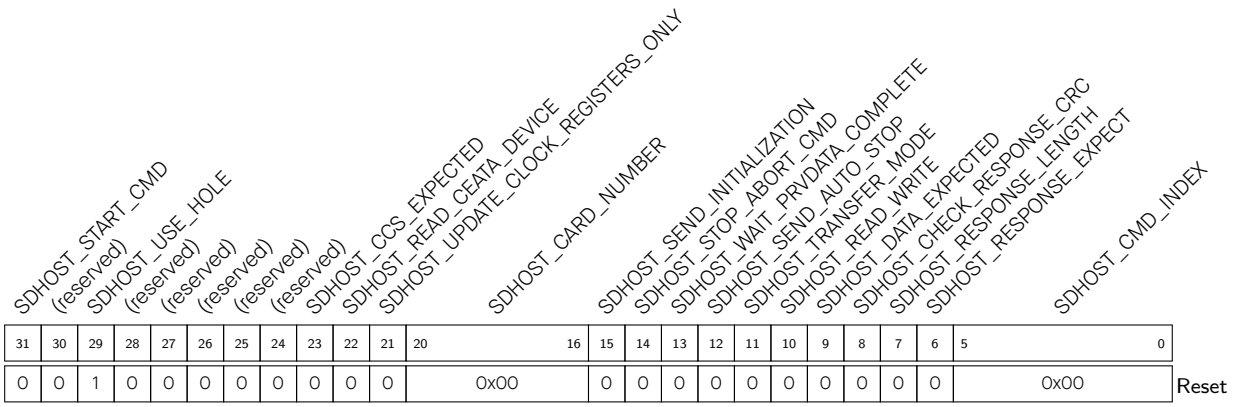
Bit 15 (EBE): 结束位错误/无 CRC 错误；
Bit 14 (ACD): 自动命令结束；
Bit 13 (SBE/BCI): 接收启动位错误；
Bit 12 (HLE): 硬件锁定写入错误
Bit 11 (FRUN): FIFO 空/满错误；
Bit 10 (HTO): 主机填充数据超时；
Bit 9 (DRTO): 数据读取超时；
Bit 8 (RTO): 响应超时；
Bit 7 (DCRC): 数据 CRC 错误；
Bit 6 (RCRC): 响应 CRC 错误；
Bit 5 (RXDR): 接收 FIFO 数据请求；
Bit 4 (TXDR): 发送 FIFO 数据请求；
Bit 3 (DTO): 数据传输结束；
Bit 2 (CD): 命令执行完毕；
Bit 1 (RE): 响应错误；
Bit 0 (CD): 卡检测。

Register 34.10. SDHOST_CMDARG_REG (0x0028)



SDHOST_CMDARG_REG 传递给卡的命令参数。(R/W)

Register 34.11. SDHOST_CMD_REG (0x002C)



SDHOST_START_CMD 开始发送命令。一旦 CIU 响应命令，此位自动清零。当此位置为 1 时，主机不应尝试写任何命令寄存器。如果尝试写寄存器，原始中断寄存器的硬件锁定错误位将会被置为 1。一旦命令被发送并且接收到 SD_MMC_CEATA 卡的响应，则原始中断寄存器的 Command Done 位将会被置为 1。(R/W)

SDHOST_USE_HOLE Use Hold 寄存器。(R/W)

0：发送给卡的 CMD 和 DATA 旁路 Hold 寄存器；

1：发送给卡的 CMD 和 DATA 经过 Hold 寄存器。

SDHOST_CCS_EXPECTED 预期命令完成信号 (CCS) 的配置。(R/W)

0：CE-ATA 设备的中断不使能（ATA 控制寄存器中 nIEN = 1）；或者命令不等待设备端的 CCS 信号；

1：CE-ATA 设备的中断使能 (nIEN = 0)，并且 RW_BLK 命令等待 CE-ATA 设备的 CCS 信号。

如果命令等待 CE-ATA 设备的 CCS 信号，软件应该将此位置为 1。SD/MMC 置位 RINTSTS 寄存器里的 Data Transfer Over (DTO) 位并且如果 DTO 中断没有被屏蔽，会产生对主机的中断。

SDHOST_READ_CEATA_DEVICE 读操作标志位。(R/W)

0：主机不进行对于 CE-ATA 设备的读操作（RW_REG 或 RW_BLK）；

1：主机进行对于 CE-ATA 设备的读操作（RW_REG 或 RW_BLK）。

软件应将此位置为 1 来表明 CE-ATA 设备正在被访问用于读传输。此位用于在执行 CE-ATA 读传输时禁能读数据超时指示。I/O 传输延迟的最大值至少为 10 秒。SD/MMC 在等待来自 CE-ATA 设备的数据时不应指示读取数据超时。

见下页...

Register 34.11. SDHOST_CMD_REG (0x002C)

接上页...

SDHOST_UPDATE_CLOCK_REGISTERS_ONLY 0：正常命令序列；1：不发送命令，仅更新时钟寄存器的值到卡时钟域内。(R/W)

以下寄存器的值被传输到卡时钟域内：CLKDIV、CLRSRC 和 CLKENA。可改变卡时钟（改变时钟频率、设置是否截断、以及设置低频模式）。这是为了改变时钟频率或停止时钟，而不必发送命令给卡。

在正常命令序列下，当 SDHOST_UPDATE_CLOCK_REGISTERS_ONLY = 0，以下控制寄存器从 BIU 传输到 CIU：CMD、CMDARG、TMOUT、CTYPE、BLKSIZ 和 BYTCNT。CIU 为新的命令序列使用新的寄存器的值。当此位置为 1 时，由于没有命令被发送给 SD_MMC_CEATA 卡，所以没有 Command Done 中断。

SDHOST_CARD_NUMBER 使用中的卡号，表示正在访问的卡的物理插槽编号。在仅 SD 模式下，支持 2 张卡。(R/W)

SDHOST_SEND_INITIALIZATION 0：在发送命令前不发送初始化序列（80 个时钟周期的 1）；1：在发送命令前发送初始化序列。(R/W)

上电后，发送任何命令到卡之前，必须向卡发送 80 个时钟进行初始化。在向卡发送第一个命令时应该将此位置为 1，以便控制器在向卡发送命令之前初始化时钟。

SDHOST_STOP_ABORT_CMD 0：停止或中止命令，停止命令和中止命令都不会停止当前的数据传输。如果中止命令发送到当前选择的功能号或不在数据传输模式，则该位应设置为 0；1：停止或中止命令，用于停止当前的数据传输。(R/W)

当开放式预定义数据传输正在进行时，并且主机发出停止或中止命令以停止数据传输时，应将此位置为 1，以使 CIU 的命令/数据状态机可以正确返回到空闲状态。

SDHOST_WAIT_PRVDATA_COMPLETE 0：即使以前的数据传输尚未完成，也立即发送命令；1：等待前面的数据传输完成后再发送命令。(R/W)

SDHOST_WAIT_PRVDATA_COMPLETE = 0 选项通常用来在数据传输时询问卡的状态或停止当前的数据传输。SDHOST_CARD_NUMBER 应该与上一个命令相同。

SDHOST_SEND_AUTO_STOP 0：在数据传输结束时不发送停止命令；1：在数据传送结束时发送停止命令。(R/W)

见下页...

Register 34.11. SDHOST_CMD_REG (0x002C)

接上页...

SDHOST_TRANSFER_MODE 0：模块数据传输命令；1: 流数据传输命令。(R/W)
如果不等待数据则为无关项。

SDHOST_READ_WRITE 0：读卡；1：写卡。(R/W)
如果不等待数据则为无关项。

SDHOST_DATA_EXPECTED 0：不等待数据传输；1：等待数据传输。(R/W)

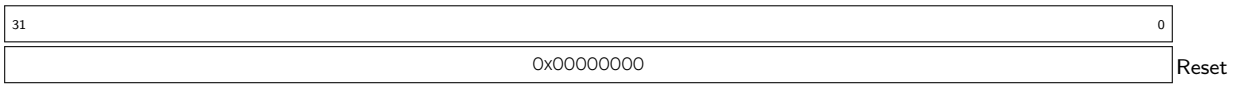
SDHOST_CHECK_RESPONSE_CRC 0：不检查；1：检查响应 CRC。(R/W)
有些命令响应不会返回有效的 CRC 位。软件应禁能对于这些命令的 CRC 检查以便禁能控制器进行 CRC 检查。

SDHOST_RESPONSE_LENGTH 0：等待卡的短响应；1：等待卡的长响应。(R/W)

SDHOST_RESPONSE_EXPECT 0：不等待卡的响应；1：等待卡的响应。(R/W)

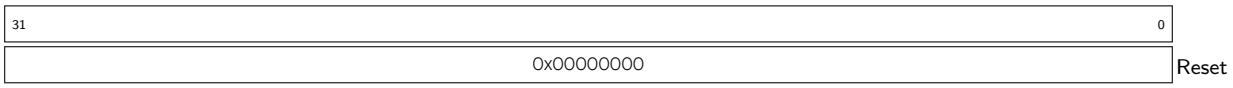
SDHOST_CMD_INDEX 命令指数。(R/W)

Register 34.12. SDHOST_RESP0_REG (0x0030)



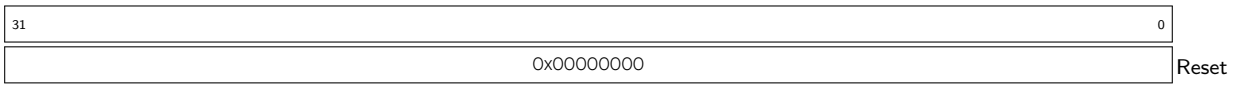
SDHOST_RESP0_REG 响应的 bit[31:0]。(RO)

Register 34.13. SDHOST_RESP1_REG (0x0034)



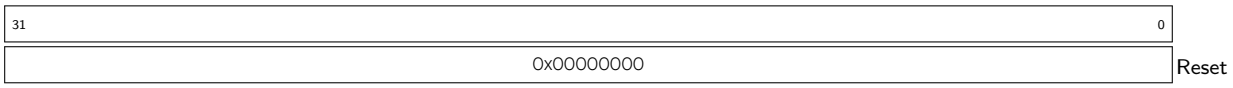
SDHOST_RESP1_REG 长响应的 bit[63:32]。(RO)

Register 34.14. SDHOST_RESP2_REG (0x0038)



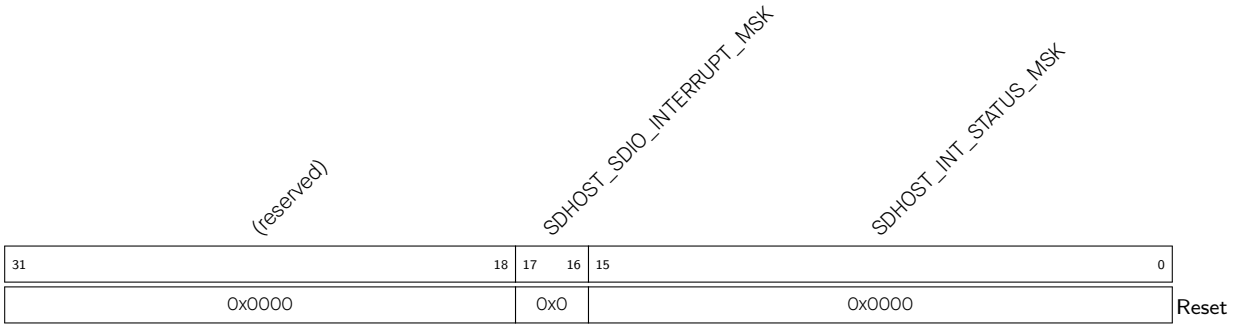
SDHOST_RESP2_REG 长响应的 bit[95:64]。(RO)

Register 34.15. SDHOST_RESP3_REG (0x003C)



SDHOST_RESP3_REG 长响应的 bit[127:96]。(RO)

Register 34.16. SDHOST_MINTSTS_REG (0x0040)

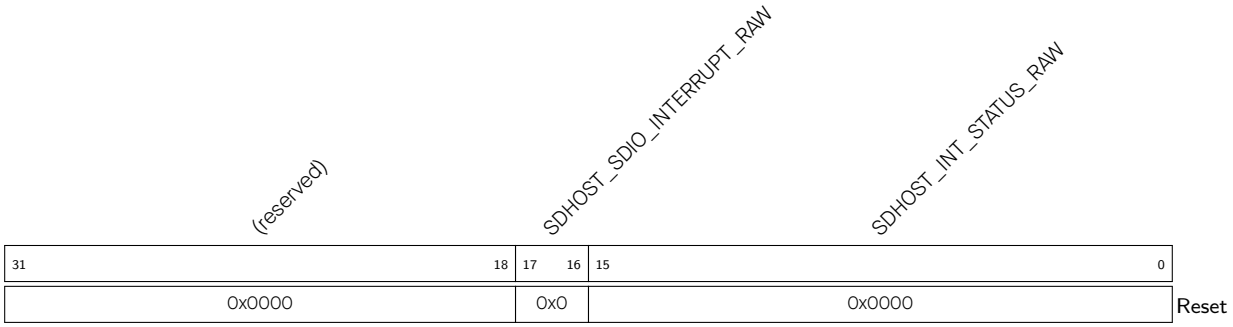


SDHOST_SDIO_INTERRUPT_MSK SDIO 中断的屏蔽位，每个卡占一个位。Bit[17:16] 分别对应卡 1 和卡 0。只有对应的 sdhost_sdio_int_mask 位被置为 1 时，SDIO 中断才会使能（置位屏蔽位使能中断）。(RO)

SDHOST_INT_STATUS_MSK 只有当中断屏蔽寄存器中的对应位被置为 1 时，中断才会使能。(RO)

Bit 15 (EBE): 结束位错误 / 无 CRC 错误；
Bit 14 (ACD): 自动命令结束；
Bit 13 (SBE/BCI): 接收启动位错误；
Bit 12 (HLE): 硬件锁定写入错误
Bit 11 (FRUN): FIFO 空/满错误；
Bit 10 (HTO): 主机填充数据超时；
Bit 9 (DRTO): 数据读取超时；
Bit 8 (RTO): 响应超时；
Bit 7 (DCRC): 数据 CRC 错误；
Bit 6 (RCRC): 响应 CRC 错误；
Bit 5 (RXDR): 接收 FIFO 数据请求；
Bit 4 (TXDR): 发送 FIFO 数据请求；
Bit 3 (DTO): 数据传输结束；
Bit 2 (CD): 命令执行完毕；
Bit 1 (RE): 响应错误；
Bit 0 (CD): 卡检测。

Register 34.17. SDHOST_RINTSTS_REG (0x0044)



SDHOST_SDIO_INTERRUPT_RAW 来自 SDIO 卡的中断，一个卡占一个位。Bit[17:16] 分别对应卡 1 和卡 0。置位某位就把相应的中断位清零，写 0 无效。(R/W)

0：没有来自卡的 SDIO 中断；

1：有来自卡的 SDIO 中断。

SDHOST_INT_STATUS_RAW 置位某位就把相应的中断位清零，写 0 无效。无论中断屏蔽状态如何，这些中断位都会被记录。(R/W)

Bit 15 (EBE): 结束位错误 / 无 CRC 错误；

Bit 14 (ACD): 自动命令结束；

Bit 13 (SBE/BCI): 接收启动为错误；

Bit 12 (HLE): 硬件锁定写入错误

Bit 11 (FRUN): FIFO 空/满错误；

Bit 10 (HTO): 主机填充数据超时；

Bit 9 (DRTO): 数据读取超时；

Bit 8 (RTO): 响应超时；

Bit 7 (DCRC): 数据 CRC 错误；

Bit 6 (RCRC): 响应 CRC 错误；

Bit 5 (RXDR): 接收 FIFO 数据请求；

Bit 4 (TXDR): 发送 FIFO 数据请求；

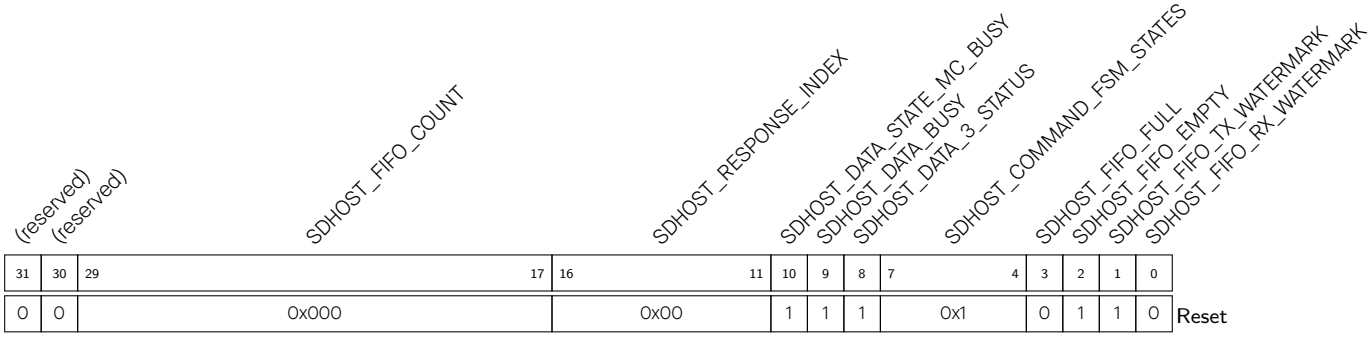
Bit 3 (DTO): 数据传输结束；

Bit 2 (CD): 命令执行完毕；

Bit 1 (RE): 响应错误；

Bit 0 (CD): 卡检测。

Register 34.18. SDHOST_STATUS_REG (0x0048)



SDHOST_FIFO_COUNT FIFO 计数器，FIFO 中被填充的地址的数量。(RO)

SDHOST_RESPONSE_INDEX 前一个响应的指数，包括内核发送的任何自动停止的响应。(RO)

SDHOST_DATA_STATE_MC_BUSY 数据发送或接收状态机忙。(RO)

SDHOST_DATA_BUSY 数据线 sdhost_card_data[0] 的值取反。(RO)

0：卡数据不忙；

1：卡数据忙。

SDHOST_DATA_3_STATUS 数据线 sdhost_card_data[3] 上的值，检查卡是否存在。(RO)

0：卡不存在；

1：卡存在。

SDHOST_COMMAND_FSM_STATES 命令状态机状态。(RO)

0: 空闲；

1: 发送初始序列；

2: 发送命令开始位；

3: 发送命令发送位；

4: 发送命令指数和参数；

5: 发送命令 CRC7；

6: 发送命令结束位；

7: 接收响应开始位；

8: 接收响应 IRQ 响应；

9: 接收响应发送位；

10: 接收响应命令指数；

11: 接收响应数据；

12: 接收响应 CRC7；

13: 接收响应结束位；

14: 命令路径等待 NCC；

15: 等待，命令-响应回转。

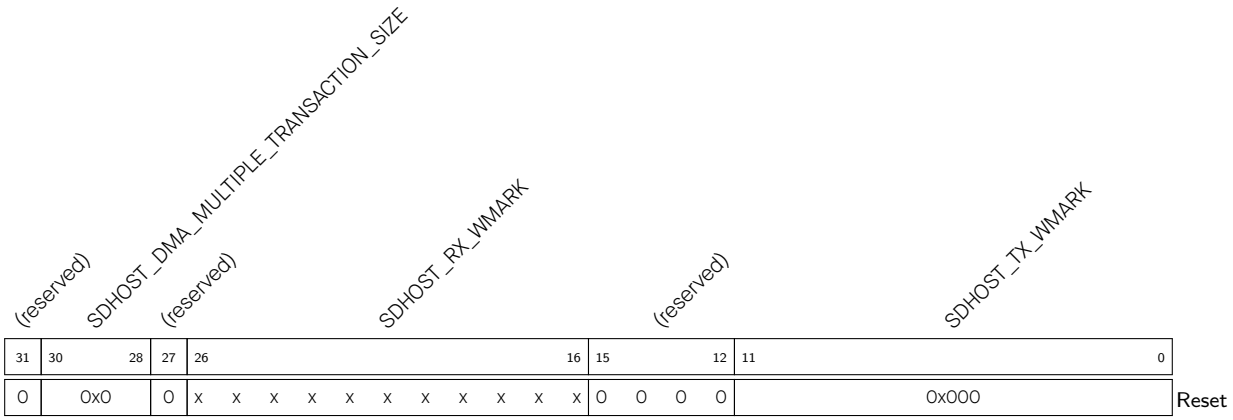
SDHOST_FIFO_FULL FIFO 为满的状态。(RO)

SDHOST_FIFO_EMPTY FIFO 为空的状态。(RO)

SDHOST_FIFO_TX_WATERMARK FIFO 达到发送阈值，不是数据传输的必要条件。(RO)

SDHOST_FIFO_RX_WATERMARK FIFO 达到接收阈值，不是数据传输的必要条件。(RO)

Register 34.19. SDHOST_FIFOTH_REG (0x004C)

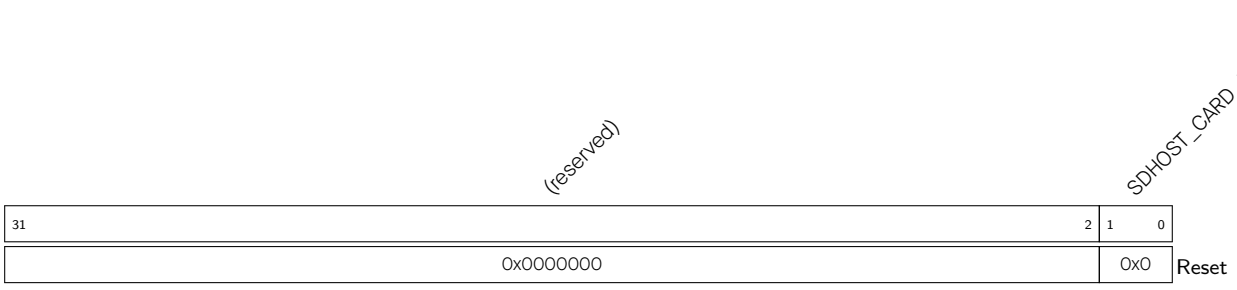


SDHOST_DMA_MULTIPLE_TRANSACTION_SIZE 突发传输的字节数，配置的值应与 DMA 控制器多个传输大小 SDHOST_SRC/DEST_MSIZE 相同。000: 1 字节传输；001: 4 字节传输；010: 8 字节传输；011: 16 字节传输；100: 32 字节传输；101: 64 字节传输；110: 128 字节传输；111: 256 字节传输。(R/W)

SDHOST_RX_WMARK 当接收数据给卡时 FIFO 的阈值。当 FIFO 数据计数大于该数值时，DMA/FIFO 请求被提出。在数据包结束期间，无论阈值大小如何，都会生成请求，以完成剩余的数据传输。在非 DMA 模式下，当接收 FIFO 阈值 (RXDR) 中断使能时，则会产生中断，而不是 DMA 请求。如果阈值大于任何剩余数据，则不产生中断。当主机看见数据发送结束中断时，主机应该读取剩下的数据。在 DMA 模式下，在数据包结束时，即使剩余的字节数少于阈值，DMA 请求也会在数据传输结束中断设置之前进行单次传输以清除所有剩余的字节。(R/W)

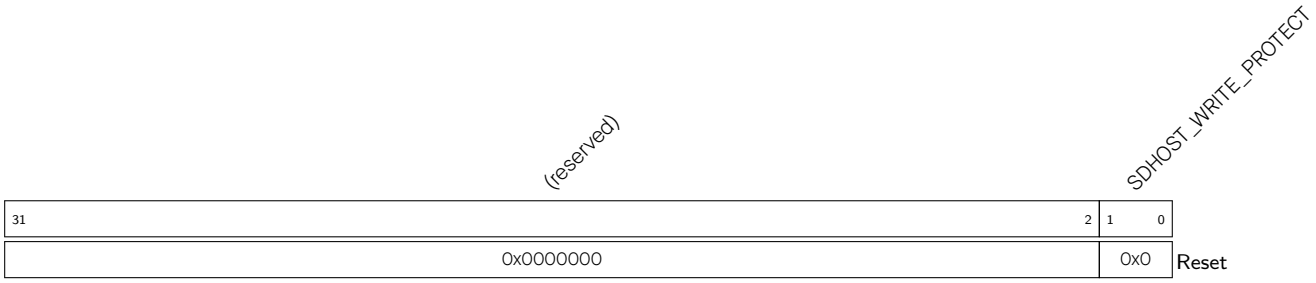
SDHOST_TX_WMARK 当发送数据给卡时 FIFO 的阈值。当 FIFO 数据计数小于等于该数值时，DMA/FIFO 请求被提出。如果使能中断，则中断发生。在数据包结束期间，无论阈值大小如何，都会生成请求。在非 DMA 模式下，当发送 FIFO 阈值 (TXDR) 中断使能时，则会产生中断，而不是 DMA 请求。在数据包结束期间，在最后一个中断时，主机负责仅用所需的剩余字节填充 FIFO（不是在 FIFO 满之前或在 CIU 完成数据传输之后，因为 FIFO 可能不为空）。在 DMA 模式下，在数据包结束时，如果最后一次传输比突发传输小，DMA 控制器将执行单个周期，直到传输所需的字节。(R/W)

Register 34.20. SDHOST_CDETECT_REG (0x0050)



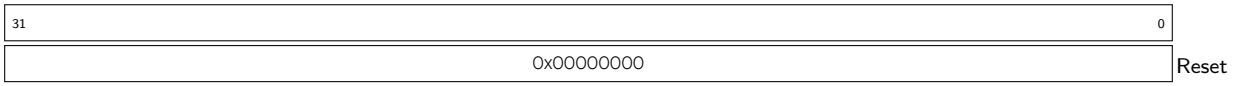
SDHOST_CARD_DETECT_N 信号线 sdhost_card_detect_n 输入端口（每个卡一个位）的值。0 代表卡存在。只有相应位被执行。(RO)

Register 34.21. SDHOST_WRTPRT_REG (0x0054)



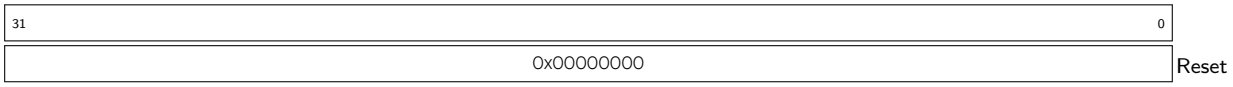
SDHOST_WRITE_PROTECT 信号线 sdhost_card_write_prt 输入端口（每个卡一个位）的值。1 表示写保护。只有的相应位被执行。(RO)

Register 34.22. SDHOST_TCBCNT_REG (0x005C)



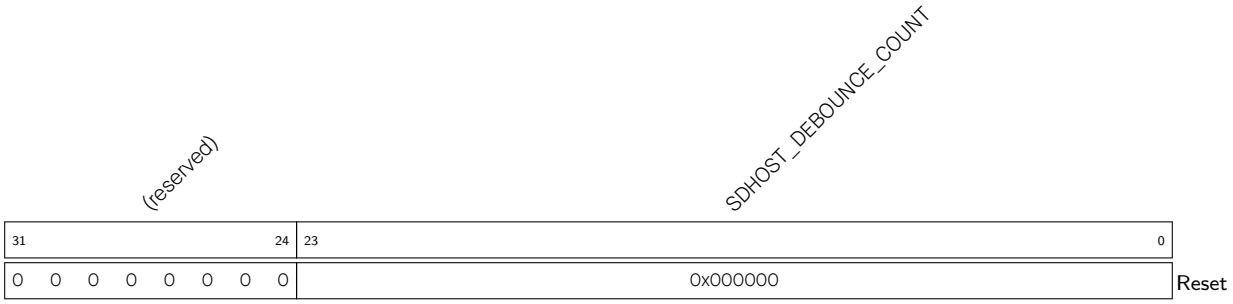
SDHOST_TCBCNT_REG CIU 模块以及传输给卡的字节数。(RO)

Register 34.23. SDHOST_TBBCNT_REG (0x0060)



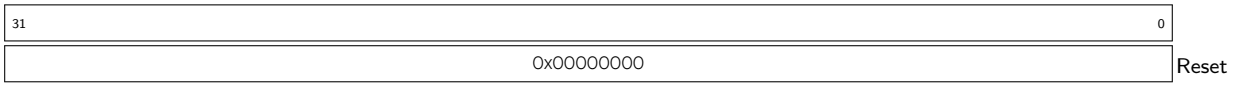
SDHOST_TBBCNT_REG 主机/DMA 和 BIU FIFO 之间传输的字节数。(RO)

Register 34.24. SDHOST_DEBNCE_REG (0x0064)



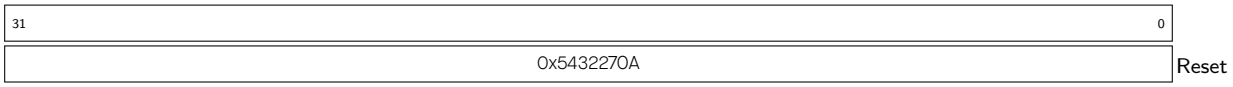
SDHOST_DEBOUNCE_COUNT 抖动消除滤波器逻辑使用的主机时钟数。典型的去抖动时间为 5 ~ 25 ms，防止插卡或移除卡的时候的不稳定性。(R/W)

Register 34.25. SDHOST_USRID_REG (0x0068)



SDHOST_USRID_REG 用户识别寄存器。此寄存器也可以作为用户的寄存器使用。(R/W)

Register 34.26. SDHOST_VERID_REG (0x006C)



SDHOST_VERSIONID_REG 表示硬件版本。固件也可以读取该寄存器。(RO)

Register 34.27. SDHOST_HCON_REG (0x0070)



SDHOST_NUM_CLK_DIV_REG 表示设计中有 4 个时钟分频器。(RO)

SDHOST_HOLD_REG 表示数据路径中有一个保持寄存器。(RO)

SDHOST_RAM_INDISE_REG 表示 SDMMC 模块内部包含 RAM。(RO)

SDHOST_DMA_WIDTH_REG 表示 DMA 数据宽度为 32。(RO)

SDHOST_ADDR_WIDTH_REG 表示寄存器地址宽度为 32。(RO)

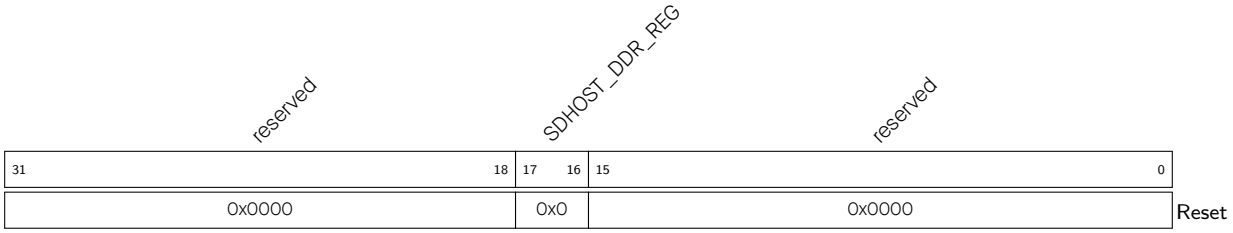
SDHOST_DATA_WIDTH_REG 表示寄存器数据宽度为 32。(RO)

SDHOST_BUS_TYPE_REG 表示寄存器配置为 APB 总线。(RO)

SDHOST_CARD_NUM_REG 表示支持的卡数量为 2。(RO)

SDHOST_CARD_TYPE_REG 表示硬件支持 SDIO 和 MMC。(RO)

Register 34.28. SDHOST_UHS_REG (0x0074)



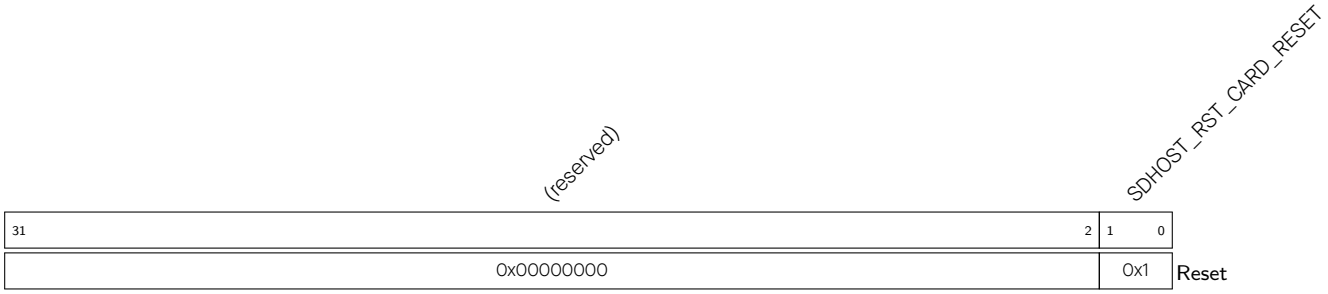
SDHOST_DDR_REG 配置卡是否使用 DDR（双倍数据速率）模式。每张卡一位，位 0 对应卡 0。

0：非 DDR 模式

1：DDR 模式

(R/W)

Register 34.29. SDHOST_RST_N_REG (0x0078)



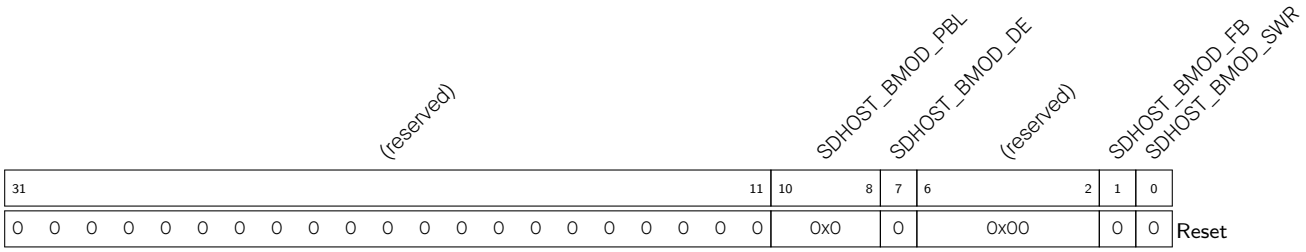
SDHOST_RST_CARD_RESET 硬件复位。(R/W)

1：工作模式；

0：复位。

这些位让卡进入空闲状态，主机工作时需要被重新初始化。SDHOST_RST_CARD_RESET[0] 应被置为 1'b0 来复位卡 0。SDHOST_RST_CARD_RESET[1] 应被置为 1'b0 来复位卡 1。

Register 34.30. SDHOST_BMOD_REG (0x0080)



SDHOST_BMOD_PBL 可编程突发长度。这些位指示在一个 IDMAC (Internal DMA Control) 传输中要执行的最大节拍数。IDMAC 每次在主机总线上开始突发传输时将总是尝试在 PBL 中指定的突发值。允许的值为 1、4、8、16、32、64、128 和 256。该值是 FIFOTH 寄存器的 MSIZE 的镜像。要修改此值，将所需的值写入 FIFOTH 寄存器。这是一个编码值，如下所示：(RO)

000: 1 字节传输；

001: 4 字节传输；

010: 8 字节传输；

011: 16 字节传输；

100: 32 字节传输；

101: 64 字节传输；

110: 128 字节传输；

111: 256 字节传输。

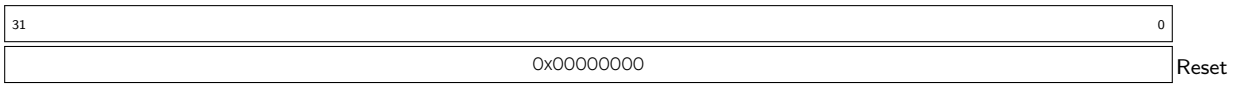
PBL 是只读值，并且仅适用于数据访问，不适用于链表访问。

SDHOST_BMOD_DE IDMAC 使能位。置位后 IDMAC 使能。(RO)

SDHOST_BMOD_FB 固定突发。控制 AHB 主接口是否执行固定突发传输。置位时，AHB 将在正常突发传输开始期间仅使用 SINGLE、INCR4、INCR8 或 INCR16。当复位时，AHB 将使用 SINGLE 和 INCR 突发传输操作。(R/W)

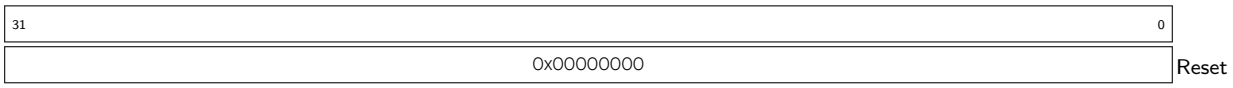
SDHOST_BMOD_SWR 软件复位。当置位时，DMA 控制器复位所有内部寄存器。一个时钟周期后自动清零。(R/W)

Register 34.31. SDHOST_PLDMND_REG (0x0080)



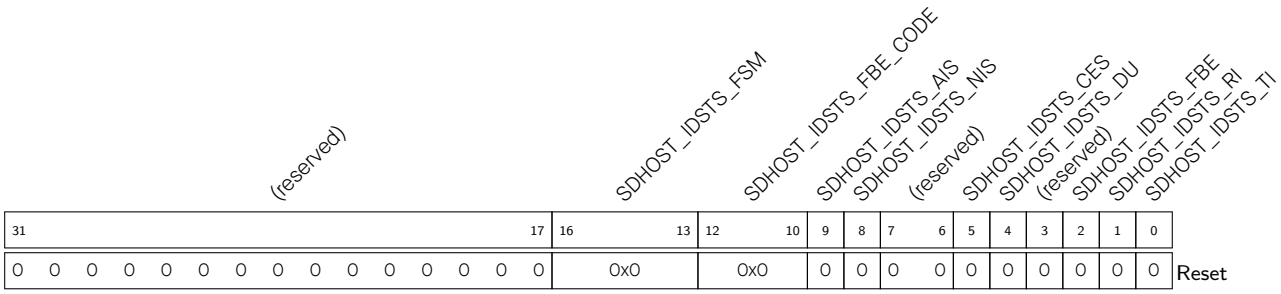
SDHOST_PLDMND_REG 轮询需求。如果链表的 OWNER 位未置位，则状态机进入挂起状态。主机需要对这个寄存器写入任意值，以使 IDMAC 状态机恢复正常读取链表的操作。这是一个只写寄存器。(WO)

Register 34.32. SDHOST_DBADDR_REG (0x0088)



SDHOST_DBADDR_REG 链表的开始。包含第一个链表的基址。最低效位 (LSB bit) [1:0] 被忽略，并由 IDMAC 内部全部取为零，因此这些 LSB 位可被视为只读。(R/W)

Register 34.33. SDHOST_IDSTS_REG (0x008C)



SDHOST_IDSTS_FSM DMAC 状态机当前状态。(RO)

0: DMA_IDLE（状态机空闲状态）;

1: DMA_SUSPEND（状态机暂停状态）;

2: DESC_RD（读链表状态）;

3: DESC_CHK（检查链表状态）;

4: DMA_RD_REQ_WAIT（状态机读数据请求等待状态）;

5: DMA_WR_REQ_WAIT（状态机写数据请求等待状态）;

6: DMA_RD（状态机读数据状态）;

7: DMA_WR（状态机写数据状态）;

8: DESC_CLOSE（当前链表使用完关闭状态）。

SDHOST_IDSTS_FBE_CODE 致命总线错误代码。表明导致总线错误的错误类型。仅当设置致命总线错误位 IDSTS[2] 被置位时有效。此字段不生成中断。(RO)

001：发送期间接收到主机中止发送;

010：接收期间接收到主机中止接收;

其他：保留。

SDHOST_IDSTS_AIS 异常中断汇总。以下各项的逻辑或：(R/W)

IDSTS[2]：致命总线中断;

IDSTS[4]：SDHOST_IDSTS_DU 位中断。

只有未屏蔽的位影响该位。这是一个粘滞位，必须在每次清零可以引起 AIS 置 1 的相应位时清零。

写 1 清零该位。

SDHOST_IDSTS_NIS 正常中断汇总。以下各项的逻辑或：(R/W)

IDSTS[0]：发送中断;

IDSTS[1]：接收中断。

只有未屏蔽的位影响该位。这是一个粘滞位，必须在每次清零可以引起 NIS 置 1 的相应位时清零。

写 1 清零该位。

见下页...

Register 34.33. SDHOST_IDSTS_REG (0x008C)

接上页...

SDHOST_IDSTS_CES 卡错误汇总。指示发送/接收卡的传输状态，也出现在 RINTSTS 中。表示以下位的逻辑或：(R/W)
EBE：结束位错误；
RTO：响应超时/引导确认超时错误；
RCRC：响应 CRC 错误；
SBE：启动位错误；
DRTO：数据读取超时/BDS 超时错误；
DCRC：接收的数据 CRC 错误；
RE：响应错误。
写 1 清零该位。IDMAC 的中止条件取决于此 CES 位的配置。如果 CES 位被使能，则 IDMAC 在响应错误时中止。

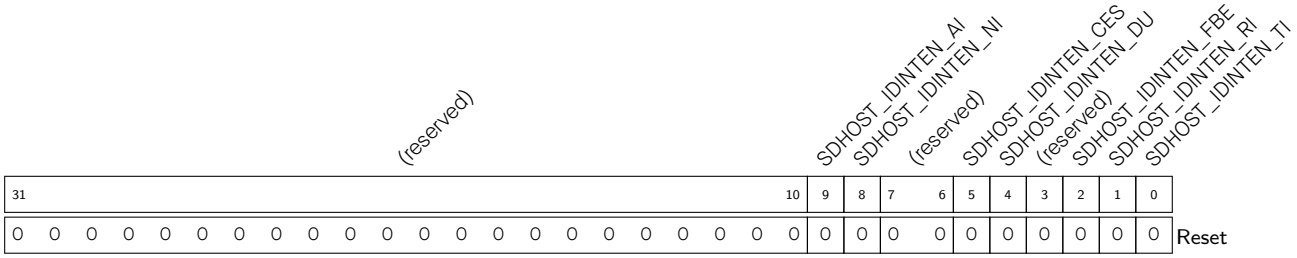
SDHOST_IDSTS_DU 链表不可用中断。当链表由于 OWNER 位 = 0 (DESO [31] = 0) 而不可用时，该位置 1。写 1 清零该位。(R/W)

SDHOST_IDSTS_FBE 致命总线错误中断。表示发生总线错误 (IDSTS[12:10])。当该位置 1 时，DMA 禁止所有总线访问。写 1 清零该位。(R/W)

SDHOST_IDSTS_RI 接收中断。表示链表的数据接收完成。写 1 清零该位。(R/W)

SDHOST_IDSTS_TI 发送中断。表示链表的数据发送完成。写 1 清零该位。(R/W)

Register 34.34. SDHOST_IDINTEN_REG (0x0090)



SDHOST_IDINTEN_AI 异常中断汇总使能位。置 1 时，使能异常中断。该位使能以下位：(R/W)

IDINTEN [2]：致命总线错误中断；

IDINTEN [4]：DU 中断。

SDHOST_IDINTEN_NI 正常中断汇总使能位。置 1 时，使能正常中断。重置时，禁能正常中断。该位使能以下位：(R/W)

IDINTEN[0]：发送中断；

IDINTEN[1]：接收中断。

SDHOST_IDINTEN_CES 卡错误汇总中断使能位。置 1 时使能卡中断汇总。(R/W)

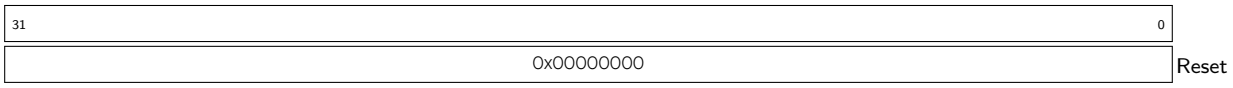
SDHOST_IDINTEN_DU 链表不可用中断。当与异常中断汇总使能位一起设置时，将使能 DU 中断。(R/W)

SDHOST_IDINTEN_FBE 致命总线错误使能位。当与异常中断汇总使能位一起设置时，将使能致命总线错误中断。复位时，致命总线错误使能中断被禁能。(R/W)

SDHOST_IDINTEN_RI 接收中断使能位。当与正常中断汇总使能位一起设置时，将使能接收中断。复位时，接收中断被禁能。(R/W)

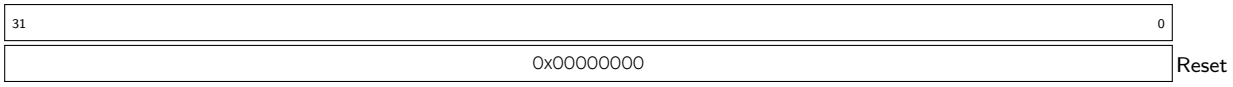
SDHOST_IDINTEN_TI 发送中断使能位。当与正常中断汇总使能位一起设置时，将使能发送中断。复位时，发送中断被禁能。(R/W)

Register 34.35. SDHOST_DSCADDR_REG (0x0094)



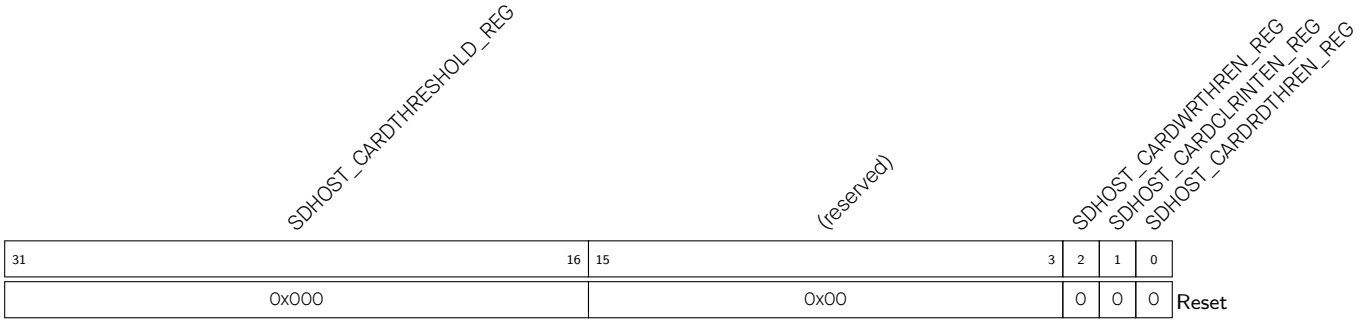
SDHOST_DSCADDR_REG 主机链表地址指针，在操作期间由 IDMAC 更新，并在复位时清零。该寄存器指向由 IDMAC 读取的当前链表的起始地址。(RO)

Register 34.36. SDHOST_BUFADDR_REG (0x0098)



SDHOST_BUFADDR_REG 主机缓冲区地址指针，在操作期间由 IDMAC 更新，并在复位时清零。该寄存器指向由 IDMAC 访问的当前数据缓冲区地址。(RO)

Register 34.37. SDHOST_CARDTHRCTL_REG (0x0100)



SDHOST_CARDTHRESHOLD_REG 配置卡读取阈值。内部 FIFO 大小为 512。当 SDHOST_CARDERTHREN_REG 被设置为 1 或 SDHOST_CARDRDTHREN_REG 被设置为 1 时，该寄存器生效。(R/W)

SDHOST_CARDWRTHREN_REG 在 HS400 模式启用时适用。

0：关闭卡写阈值功能。

1：启用卡写阈值功能。

(R/W)

SDHOST_CARDCLRINTEN_REG 配置是否使能忙碌清除中断生成。

0：不使能

1：使能

(R/W)

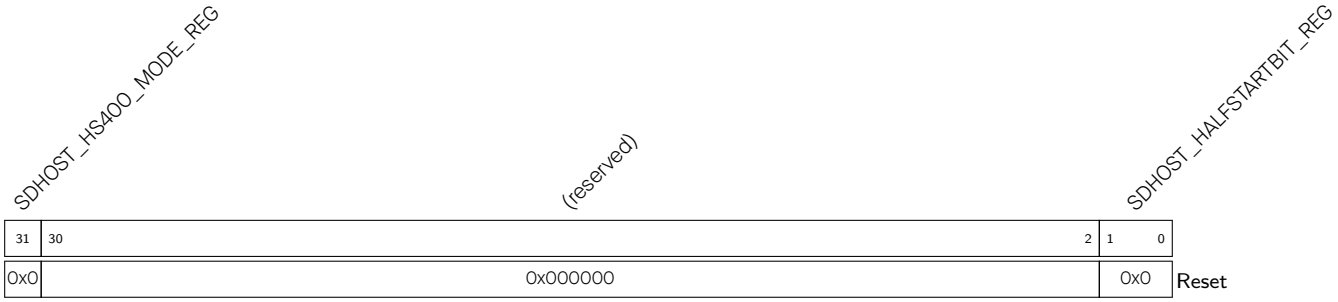
SDHOST_CARDRDTHREN_REG 配置是否使能卡读取阈值。

0：不使能

1：使能

(R/W)

Register 34.38. SDHOST_EMMC_DDR_REG (0x010C)



SDHOST_HS400_MODE_REG 置 1 启用 HS400 模式。

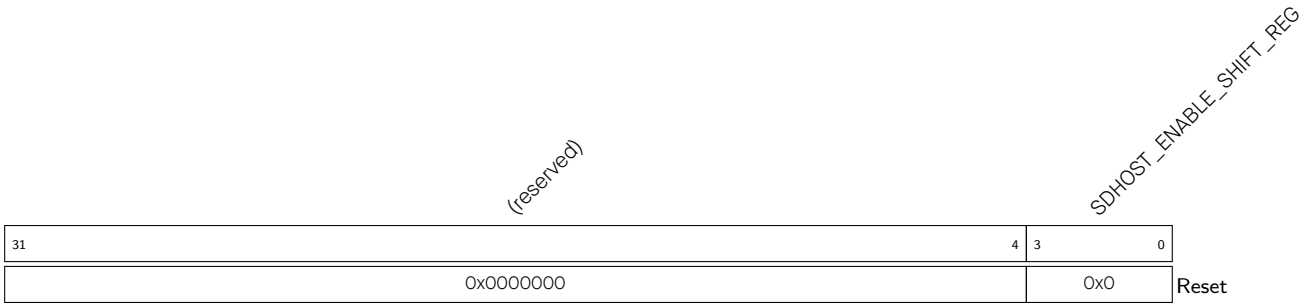
SDHOST_HALFSTARTBIT_REG 配置开始位检测机制的开始位检测长度。每张卡一位，位 0 对应卡 0。eMMC4.5 及以上版本此位需为 1，SD 应用此位需为 0。对于 eMMC4.5，起始位检测模式可配置为：

0：完整周期

1：少于一个完整周期

(R/W)

Register 34.39. SDHOST_ENSHIFT_REG (0x0110)



SDHOST_ENABLE_SHIFT_REG 配置设计中默认的相位偏移量。每张卡 2 位，位 1:0 对应卡 0。

0x0：默认相位偏移

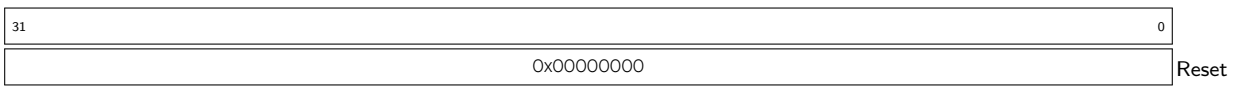
0x1：偏移到下个上升沿

0x2：偏移到下个下降沿

0x3：保留

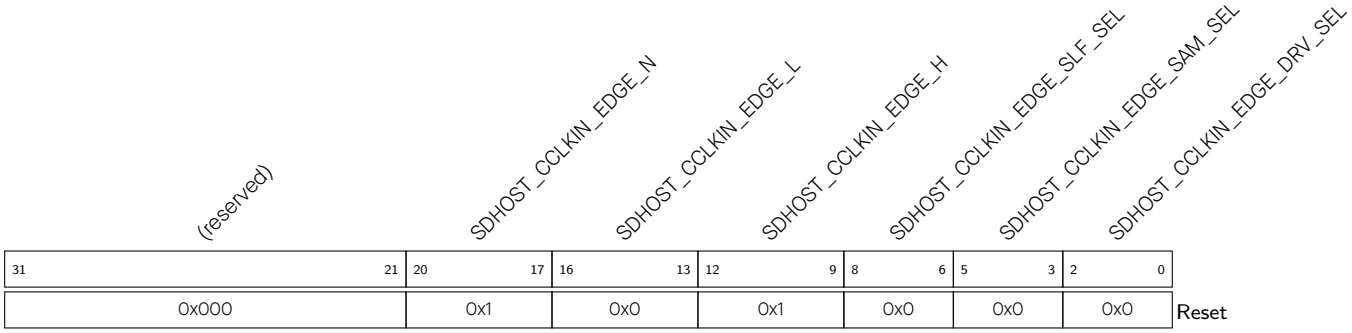
(R/W)

Register 34.40. SDHOST_BUFFIFO_REG (0x0200)



SDHOST_BUFFIFO_REG CPU 通过 FIFO 读写发送的数据。该寄存器指向当前数据 FIFO。(RO)

Register 34.41. SDHOST_CLK_DIV_EDGE_REG (0x0800)



CCLKIN_EDGE_N 值应与 CCLKIN_EDGE_L 相同。(R/W)

CCLKIN_EDGE_L 分频时钟的低电平，值应比 CCLKIN_EDGE_H 大。(R/W)

CCLKIN_EDGE_H 分频时钟的高电平，值应比 CCLKIN_EDGE_L 小。(R/W)

CCLKIN_EDGE_SLF_SEL 用于选择内部时钟信号的相位，90 度相位、180 度相位或 270 度相位。(R/W)

CCLKIN_EDGE_SAM_SEL 用于选择输入时钟信号的相位，90 度相位、180 度相位或 270 度相位。(R/W)

CCLKIN_EDGE_DRV_SEL 用于选择输出时钟信号的相位，90 度相位、180 度相位或 270 度相位。(R/W)

说明：SD/MMC 使用该寄存器分频 160M 时钟 (CCLKIN_EDGE_H/CCLKIN_EDGE_L)。输出时钟通过该寄存器和寄存器 SDHOST_CLKDIV_REG 连接 SDIO 从机分频器，使用 SDHOST_CLKSRC_REG 时可选择 4 个时钟源。

第 35 章

LED PWM 控制器 (LEDC)

LED PWM 控制器用于生成控制 LED 的脉冲宽度调制信号 (PWM)，具有占空比自动渐变等专门功能。该外设也可成生成 PWM 信号用作其他用途。

35.1 主要特性

LED PWM 控制器具有如下特性:

- 八个独立的 PWM 生成器（即八个通道）
- 四个独立定时器，可实现小数分频
- 占空比自动渐变（即 PWM 信号占空比可逐渐增加或减小，无须处理器干预），渐变完成时产生中断
- 输出 PWM 信号相位可调
- 低功耗模式 (Light-sleep mode) 下可输出 PWM 信号
- PWM 最大精度为 14 位

四个定时器具有相同的功能和运行方式，下文将四个定时器统称为定时器*x*（*x* 的范围是 0 到 3）。八个 PWM 生成器的功能和运行方式也相同，下文将统称为 PWM *n*（n 的范围是 0 到 7）。

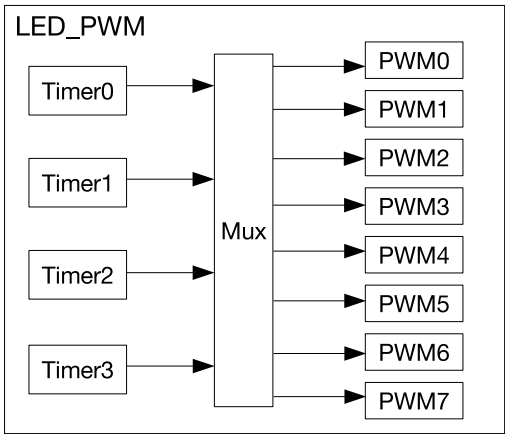


图 35.1-1. LED PWM 控制器架构

35.2 功能描述

35.2.1 架构

图 35.1-1 为 LED PWM 控制器的架构。四个定时器可独立配置（时钟分频器和计数器最大值），每个定时器内部有一个时基计数器（即基于基准时钟周期计数的计数器）。每个 PWM 生成器会在四个定时器中择一，以该定时

器的计数值为基准生成 PWM 信号。

图 35.2-1 为定时器和 PWM 生成器的主要功能块。

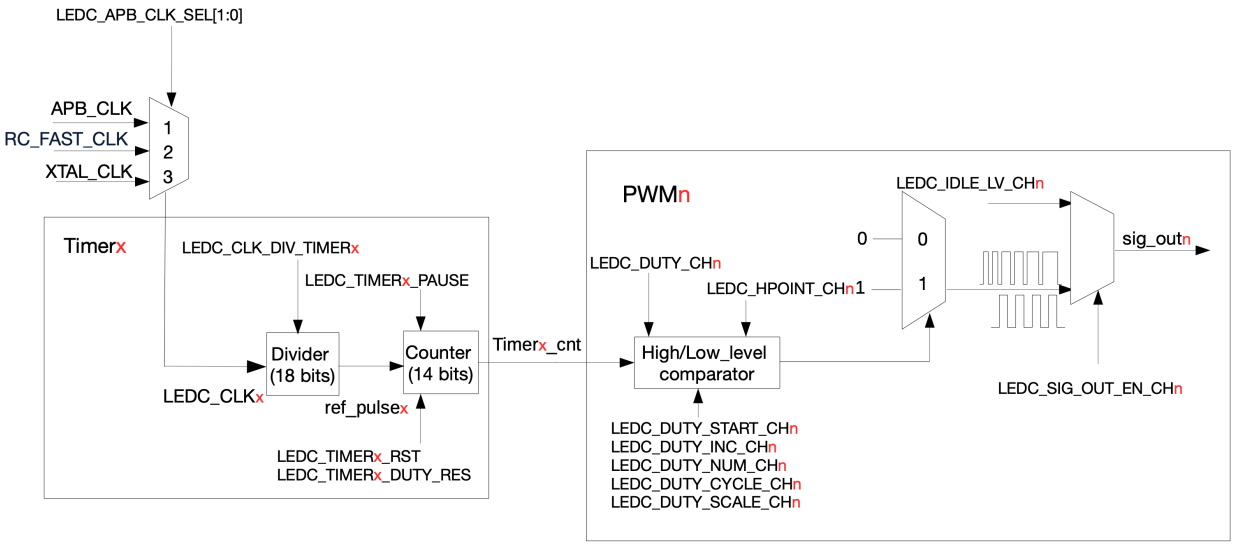


图 35.2-1. 定时器和 PWM 生成器功能块

35.2.2 定时器

LED PWM 控制器的每个定时器内部都有一个时基计数器，该计数器在时钟信号的每个周期加 1。图 35.2-1 中时基计数器使用的时钟信号称为 ref_pulse*x*。所有定时器使用同一个时钟源信号（LEDC_CLK*x*），该时钟源信号经分频器分频后产生 ref_pulse*x* 供计数器使用。

35.2.2.1 时钟源

软件配置 LED PWM 寄存器由 APB_CLK 驱动。更多关于 APB_CLK 的信息，详见章节 7 *复位和时钟*。要使用 LED PWM 控制器，需使能 LED PWM 的 APB_CLK 时钟信号，该时钟信号可通过置位 SYSTEM_PERIP_CLK_EN0_REG 寄存器的 SYSTEM_LEDC_CLK_EN 使能，通过软件置位 SYSTEM_PERIP_RST_EN0_REG 寄存器的 SYSTEM_LEDC_RST 位复位。更多信息，请参阅章节 17 *系统寄存器 (SYSTEM)* 的表 17.3-2

LED PWM 控制器的定时器有三个时钟源信号可以选择：APB_CLK、RC_FAST_CLK 和 XTAL_CLK（更多有关时钟源的信息详见章节 7 *复位和时钟*）。为 LEDC_CLK*x* 选择时钟源信号的配置如下：

- APB_CLK：将 LEDC_APB_CLK_SEL[1:0] 置 1
- RC_FAST_CLK：将 LEDC_APB_CLK_SEL[1:0] 置 2
- XTAL_CLK：将 LEDC_APB_CLK_SEL[1:0] 置 3

之后，LEDC_CLK*x* 信号会进入时钟分频器。

35.2.2.2 时钟分频器配置

LEDC_CLK*x* 信号传输到时钟分频器，产生 ref_pulse*x* 信号供计数器使用。ref_pulse*x* 的频率等于 LEDC_CLK*x* 的频率经分频系数 LEDC_CLK_DIV 分频后的结果（见图 35.2-1）。

分频系数 LEDC_CLK_DIV 为小数，因此其值可为非整数。分频系数 LEDC_CLK_DIV 可根据下列等式配置：

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

- 整数部分 *A* 为 LEDC_CLK_DIV_TIMER*x* 字段的高 10 位（即 LEDC_TIMER*x*_CONF_REG[21:12]）
- 小数部分 *B* 为 LEDC_CLK_DIV_TIMER*x* 字段的低 8 位（即 LEDC_TIMER*x*_CONF_REG[11:4]）

小数部分 *B* 为 0 时，LEDC_CLK_DIV 的值为整数（整数分频）。也就是说，每 *A* 个 LEDC_CLK*x* 时钟周期产生一个 ref_pulse*x* 时钟脉冲。

小数部分 *B* 不为 0 时，LEDC_CLK_DIV 的值非整数。时钟分频器按照 *A* 个 LEDC_CLK*x* 时钟周期和 (*A*+1) 个 LEDC_CLK*x* 时钟周期轮流进行非整数分频。这样一来，ref_pulse*x* 时钟脉冲的平均频率便会是理想值（非整数分频的频率）。每 256 个 ref_pulse*x* 时钟脉冲中：

- 有 *B* 个以 (*A*+1) 个 LEDC_CLK*x* 时钟周期分频
- 有 (256-*B*) 个以 *A* 个 LEDC_CLK*x* 时钟周期分频
- 以 (*A*+1) 个 LEDC_CLK*x* 时钟周期分频的时钟脉冲均匀分布在以 *A* 分频的时钟脉冲中

图 35.2-2 展示了分频系数 LEDC_CLK_DIV 非整数时，LEDC_CLK*x* 时钟周期和 ref_pulse*x* 时钟脉冲的关系。

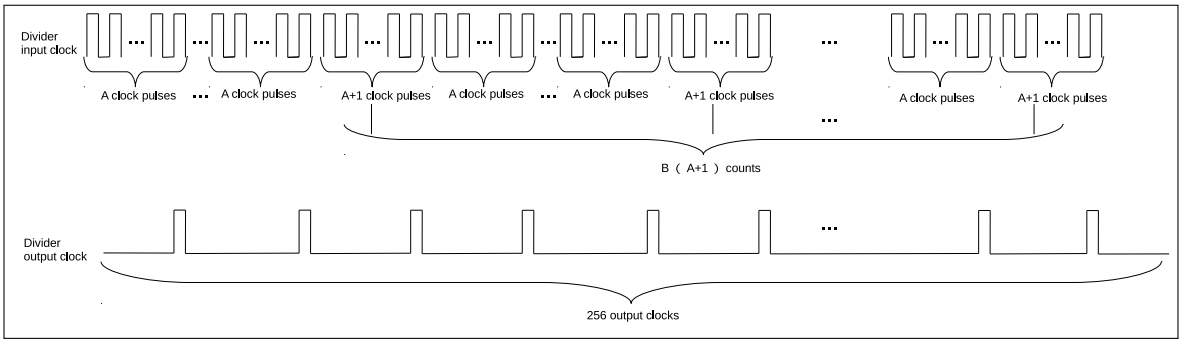


图 35.2-2. LEDC_CLK_DIV 非整数时的分频

重新设置分频器的分频系数最大值，需先配置 LEDC_CLK_DIV_TIMER*x* 字段，然后置位 LEDC_TIMER*x*_PARA_UP 字段应用新配置。新配置会在计数器下次溢出时生效。LEDC_TIMER*x*_PARA_UP 字段由硬件自动清除。

35.2.2.3 14 位计数器

每个定时器有一个以 ref_pulse*x* 为基准时钟的 14 位时基计数器（见图 35.2-1）。LEDC_TIMER*x*_DUTY_RES 字段用于配置 14 位计数器的最大值。因此，PWM 信号的最大精确度为 14 位。计数器最大可计数至 $2^{LEDC\_TIMERx\_DUTY\_RES} - 1$，然后溢出重新从 0 开始计数。软件可以读取、复位、暂停计数器。

计数器可在每次溢出时触发 (LEDC_TIMER*x*_OVF_INT) 中断，这个中断为硬件自动产生，不需要配置。计数器也可配置为在溢出 $LEDC\_OVF\_NUM\_CHn + 1$) 次时触发 LEDC_OVF_CNT_CH*n*_INT 中断，该中断配置步骤如下：

1. 配置 LEDC_TIMER_SEL_CH*n* 为 PWM 生成器选择该计数器
2. 置位 LEDC_OVF_CNT_EN_CH*n* 使能计数器
3. 把 LEDC_OVF_NUM_CH*n* 的值设为计数器触发中断的溢出次数减 1

4. 置位 LEDC_OVF_CNT_CH*n*_INT_ENA 使能溢出中断
5. 置位 LEDC_TIMER*x*_DUTY_RES 使能定时器，等待 LEDC_OVF_CNT_CH*n*_INT 中断产生

如图 35.2-1 所示，PWM 生成器输出信号 sig_out*n* 的频率取决于定时器时钟源 LEDC_CLK*x* 的频率、时钟分频系数 LEDC_CLK_DIV 以及占空比精度（计数器位宽）LEDC_TIMER*x*_DUTY_RES：

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

上述公式变形后，可得到以下公式计算预期的占空比精度：

$$\text{LEDC\_TIMER}x\text{\_DUTY\_RES} = \log_2\left(\frac{f_{\text{LEDC\_CLK}x}}{f_{\text{PWM}} \cdot \text{LEDC\_CLK\_DIV}}\right)$$

表 35.2-1 列出了常用配置频率及其对应精度。

表 35.2-1. 常用配置频率及精度

LEDC_CLK*x*	PWM 频率	最高精度（位）[1]	最低精度（位）[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
RC_FAST_CLK (17.5 MHz)	1 kHz	14	5
RC_FAST_CLK (17.5 MHz)	1.75 kHz	13	4

[1] 最高精度指时钟分频系数 LEDC_CLK_DIV 为 1 时的精度，向下取整。如果经公式计算出的最高精度超过了计数器位宽 14 位，则最高精度为 14。

[2] 最低精度指时钟分频系数 LEDC_CLK_DIV 为 $1023 + \frac{255}{256}$ 时的精度，向上取整。如果经公式计算出的最低精度小于 0，则最低精度为 1。

在运行时改变计数器的最大值，需先置位 LEDC_TIMER*x*_DUTY_RES 字段，然后置位 LEDC_TIMER*x*_PARA_UP 字段。新的配置在计数器下一次溢出时生效。如果重新配置 LEDC_OVF_CNT_EN_CH*n* 字段，需置位 LEDC_PARA_UP_CH*n* 应用新配置。总之，更改配置时需置位 LEDC_TIMER*x*_PARA_UP 或 LEDC_PARA_UP_CH*n* 应用新配置。LEDC_TIMER*x*_PARA_UP 和 LEDC_PARA_UP_CH*n* 字段由硬件自动清除。

35.2.3 PWM 生成器

要生成 PWM 信号，PWM 生成器 (PWM*n*) 需选择一个定时器 (Timer*x*)。每个 PWM 生成器均可通过置位 LEDC_TIMER_SEL_CH*n* 单独配置，在四个定时器中选择一个输出 PWM 信号。

如图 35.2-1 所示，每个 PWM 生成器主要包括一个高低电平比较器和两个选择器。PWM 生成器将定时器的 14 位计数值 (Timer*x*_cnt) 与高低电平比较器的值 Hpoint*n* 和 Lpoint*n* 比较。如果定时器的计数值等于 Hpoint*n* 或 Lpoint*n*，PWM 信号可以输出高低电平：

- 如果 Timer*x*_cnt == Hpoint*n*，则 sig_out*n* 为 1。

- 如果 Timer*x*_cnt == Lpoint*n*，则 sig_out*n* 为 0。

图 35.2-3 展示了如何使用 Hpoint*n* 和 Lpoint*n* 生成占空比固定的 PWM 信号。

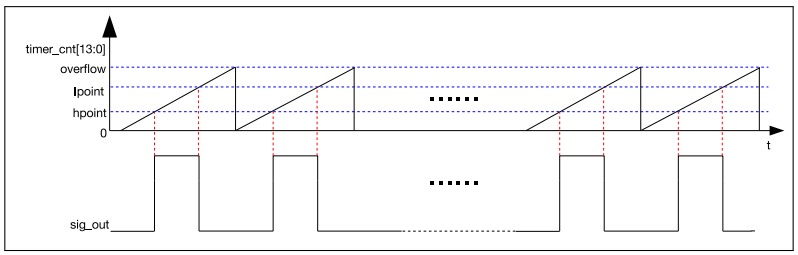


图 35.2-3. LED PWM 输出信号图

每当所选定时器的计数器溢出时，PWM 生成器 (PWM*n*) 的 Hpoint*n* 值更新为 LEDC_HPOINT_CH*n*。Lpoint*n* 的值同样在计数器每次溢出时更新，为 LEDC_DUTY_CH*n*[18:4] 和 LEDC_HPOINT_CH*n* 的和。通过配置以上两个字段，可设置 PWM 输出的相对相位和占空比。

置位 LEDC_SIG_OUT_EN_CH*n*，开启 PWM 信号（sig_out*n*）输出；清除 LEDC_SIG_OUT_EN_CH*n*，关闭 PWM 信号输出，输出信号 sig_out*n* 输出恒定电平，电平值为 LEDC_IDLE_LV_CH*n*.

LEDC_DUTY_CH*n*[3:0] 通过周期性改变 PWM 输出信号 sig_out*n* 的占空比实现微调。如 LEDC_DUTY_CH*n*[3:0] 不为 0，那么 sig_out*n* 每 16 个周期中，有 LEDC_DUTY_CH*n*[3:0] 个周期的 PWM 脉冲占空比要比 (16 - LEDC_DUTY_CH*n*[3:0]) 个周期的脉冲占空比多一个定时器的计数周期。比如，如果 LEDC_DUTY_CH*n*[18:4] 设为 10，LEDC_DUTY_CH*n*[3:0] 设为 5，则 16 个周期中，有 5 个周期的 PWM 脉冲占空比为 11，剩余 11 个周期的 PWM 脉冲占空比为 10。16 个周期的平均占空比为 10.3125。

如果重新配置 LEDC_TIMER_SEL_CH*n*、LEDC_HPOINT_CH*n*、LEDC_DUTY_CH*n*[18:4] 和 LEDC_SIG_OUT_EN_CH*n* 字段，需置位 LEDC_PARA_UP_CH*n* 应用新配置。新配置在计数器下次溢出时生效。LEDC_TIMER*x*_PARA_UP 字段由硬件自动清除。

35.2.4 占空比渐变

PWM 生成器可以渐变 PWM 输出信号的占空比，即由一种占空比逐渐变为为另一种占空比。如果开启占空比渐变功能，Lpoint*n* 的值会在计数器溢出固定次数后递增或递减。图 35.2-4 展示了占空比渐变功能。

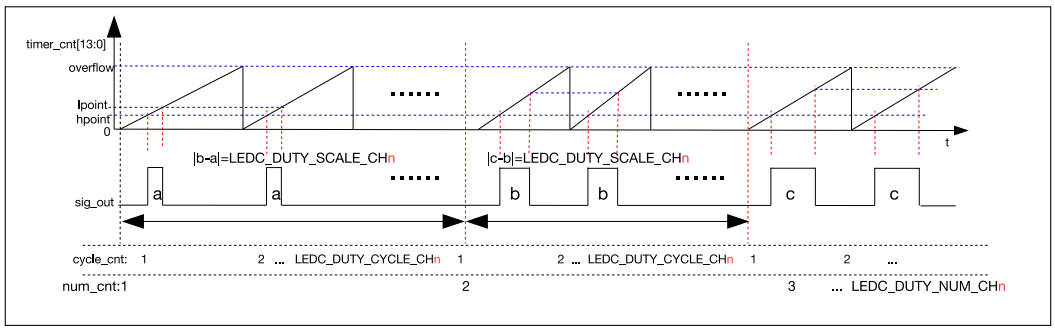


图 35.2-4. 输出信号占空比渐变图

占空比渐变功能可通过以下寄存器字段配置：

- LEDC_DUTY_CH*n* 用于设置 Lpoint*n* 的初始值。
- LEDC_DUTY_START_CH*n* 置 1 或清零，使能或关闭占空比渐变功能。

- LEDC_DUTY_CYCLE_CH*n* 用于设置 Lpoint*n* 在计数器溢出多少次时递增或递减。也就是说，Lpoint*n* 会在计数器溢出 LEDC_DUTY_CYCLE_CH*n* 次时递增或递减。
- LEDC_DUTY_INC_CH*n* 置 1 或清零，Lpoint*n* 递增或递减。
- LEDC_DUTY_SCALE_CH*n* 用于设置 Lpoint*n* 递增或递减的值。
- LEDC_DUTY_NUM_CH*n* 用于设置占空比渐变停止前，Lpoint*n* 递增或递减的最大次数。

如果重新配置 LEDC_DUTY_CH*n*、LEDC_DUTY_START_CH*n*、LEDC_DUTY_CYCLE_CH*n*、LEDC_DUTY_INC_CH*n*、LEDC_DUTY_SCALE_CH*n* 和 LEDC_DUTY_NUM_CH*n* 字段，需置位 LEDC_PARA_UP_CH*n* 应用新配置。
LEDC_PARA_UP_CH*n* 置位后，新配置立即生效。LEDC_TIMER*x*_PARA_UP 字段由硬件自动清除。

35.2.5 中断

- LEDC_OVF_CNT_CH*n*_INT：定时器计数器溢出 (LEDC_OVF_NUM_CH*n* + 1) 次且寄存器 LEDC_OVF_CNT_EN_CH*n* 置 1 时触发中断。
- LEDC_DUTY_CHNG_END_CH*n*_INT：PWM 生成器渐变完成后触发中断。
- LEDC_TIMER*x*_OVF_INT：定时器达到最大计数值时触发中断。

35.3 寄存器列表

本小节的所有地址均为相对于 LED PWM 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

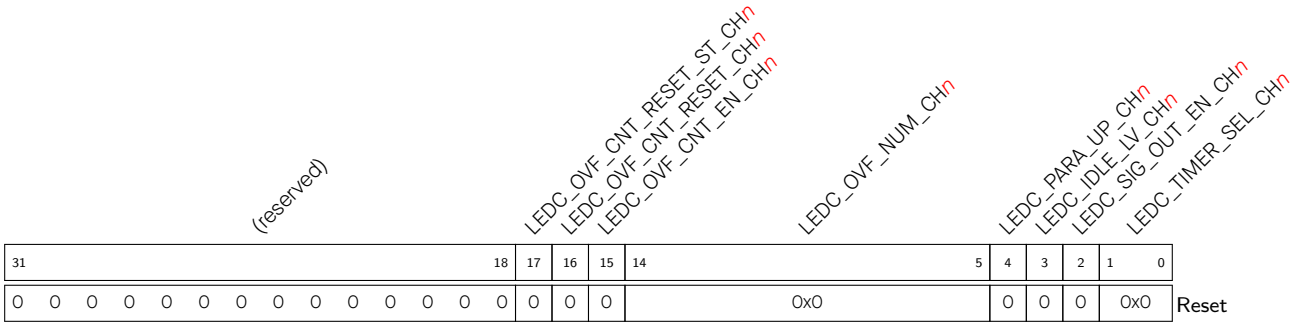
名称	描述	地址	访问
配置寄存器			
LEDC_CH0_CONF0_REG	通道 0 的配置寄存器 0	0x0000	不定
LEDC_CH0_CONF1_REG	通道 0 的配置寄存器 1	0x000C	读/写
LEDC_CH1_CONF0_REG	通道 1 的配置寄存器 0	0x0014	不定
LEDC_CH1_CONF1_REG	通道 1 的配置寄存器 1	0x0020	读/写
LEDC_CH2_CONF0_REG	通道 2 的配置寄存器 0	0x0028	不定
LEDC_CH2_CONF1_REG	通道 2 的配置寄存器 1	0x0034	读/写
LEDC_CH3_CONF0_REG	通道 3 的配置寄存器 0	0x003C	不定
LEDC_CH3_CONF1_REG	通道 3 的配置寄存器 1	0x0048	读/写
LEDC_CH4_CONF0_REG	通道 4 的配置寄存器 0	0x0050	不定
LEDC_CH4_CONF1_REG	通道 4 的配置寄存器 1	0x005C	读/写
LEDC_CH5_CONF0_REG	通道 5 的配置寄存器 0	0x0064	不定
LEDC_CH5_CONF1_REG	通道 5 的配置寄存器 1	0x0070	读/写
LEDC_CH6_CONF0_REG	通道 6 的配置寄存器 0	0x0078	不定
LEDC_CH6_CONF1_REG	通道 6 的配置寄存器 1	0x0084	读/写
LEDC_CH7_CONF0_REG	通道 7 的配置寄存器 0	0x008C	不定
LEDC_CH7_CONF1_REG	通道 7 的配置寄存器 1	0x0098	读/写
LEDC_CONF_REG	LEDC 全局配置寄存器	0x00D0	读/写
高位点寄存器			
LEDC_CH0_HPOINT_REG	通道 0 的高位点寄存器	0x0004	读/写
LEDC_CH1_HPOINT_REG	通道 1 的高位点寄存器	0x0018	读/写
LEDC_CH2_HPOINT_REG	通道 2 的高位点寄存器	0x002C	读/写
LEDC_CH3_HPOINT_REG	通道 3 的高位点寄存器	0x0040	读/写
LEDC_CH4_HPOINT_REG	通道 4 的高位点寄存器	0x0054	读/写
LEDC_CH5_HPOINT_REG	通道 5 的高位点寄存器	0x0068	读/写
LEDC_CH6_HPOINT_REG	通道 6 的高位点寄存器	0x007C	读/写
LEDC_CH7_HPOINT_REG	通道 7 的高位点寄存器	0x0090	读/写
占空比寄存器			
LEDC_CH0_DUTY_REG	通道 0 的初始占空比	0x0008	读/写
LEDC_CH0_DUTY_R_REG	通道 0 的当前占空比	0x0010	只读
LEDC_CH1_DUTY_REG	通道 1 的初始占空比	0x001C	读/写
LEDC_CH1_DUTY_R_REG	通道 1 的当前占空比	0x0024	只读
LEDC_CH2_DUTY_REG	通道 2 的初始占空比	0x0030	读/写
LEDC_CH2_DUTY_R_REG	通道 2 的当前占空比	0x0038	只读
LEDC_CH3_DUTY_REG	通道 3 的初始占空比	0x0044	读/写
LEDC_CH3_DUTY_R_REG	通道 3 的当前占空比	0x004C	只读
LEDC_CH4_DUTY_REG	通道 4 的初始占空比	0x0058	读/写

名称	描述	地址	访问
LEDC_CH4_DUTY_R_REG	通道 4 的当前占空比	0x0060	只读
LEDC_CH5_DUTY_REG	通道 5 的初始占空比	0x006C	读/写
LEDC_CH5_DUTY_R_REG	通道 5 的当前占空比	0x0074	只读
LEDC_CH6_DUTY_REG	通道 6 的初始占空比	0x0080	读/写
LEDC_CH6_DUTY_R_REG	通道 6 的当前占空比	0x0088	只读
LEDC_CH7_DUTY_REG	通道 7 的初始占空比	0x0094	读/写
LEDC_CH7_DUTY_R_REG	通道 7 的当前占空比	0x009C	只读
定时器寄存器			
LEDC_TIMER0_CONF_REG	定时器 0 配置	0x00A0	不定
LEDC_TIMER0_VALUE_REG	定时器 0 的当前计数器值	0x00A4	只读
LEDC_TIMER1_CONF_REG	定时器 1 配置	0x00A8	不定
LEDC_TIMER1_VALUE_REG	定时器 1 的当前计数器值	0x00AC	只读
LEDC_TIMER2_CONF_REG	定时器 2 配置	0x00B0	不定
LEDC_TIMER2_VALUE_REG	定时器 2 的当前计数器值	0x00B4	只读
LEDC_TIMER3_CONF_REG	定时器 3 配置	0x00B8	不定
LEDC_TIMER3_VALUE_REG	定时器 3 的当前计数器值	0x00BC	只读
中断寄存器			
LEDC_INT_RAW_REG	原始中断状态	0x00C0	只读
LEDC_INT_ST_REG	屏蔽中断状态	0x00C4	只读
LEDC_INT_ENA_REG	中断使能位	0x00C8	读/写
LEDC_INT_CLR_REG	中断清除位	0x00CC	只写
版本寄存器			
LEDC_DATE_REG	版本控制寄存器	0x00FC	读/写

35.4 寄存器

本小节的所有地址均为相对于 LED PWM 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 35.1. LEDC_CH*n*_CONF0_REG (*n*: 0-7) (0x0000+0x14**n*)



LEDC_TIMER_SEL_CH*n* 用于选择通道 *n* 的定时器。

0：选择定时器 0

1：选择定时器 1

2：选择定时器 2

3：选择定时器 3 (R/W)

LEDC_SIG_OUT_EN_CH*n* 置位此位，使能通道 *n* 的信号输出。(R/W)

LEDC_IDLE_LV_CH*n* 控制通道 *n* 不工作时（LEDC_SIG_OUT_EN_CH*n* 为 0 时）的输出电平。(R/W)

LEDC_PARA_UP_CH*n* 用于更新通道 *n* 的的下列字段，由硬件自动清除。(WO)

- 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*

见下页...

Register 35.1. LEDC_CH*n*_CONF0_REG (*n*: 0-7) (0x0000+0x14**n*)

接上页...

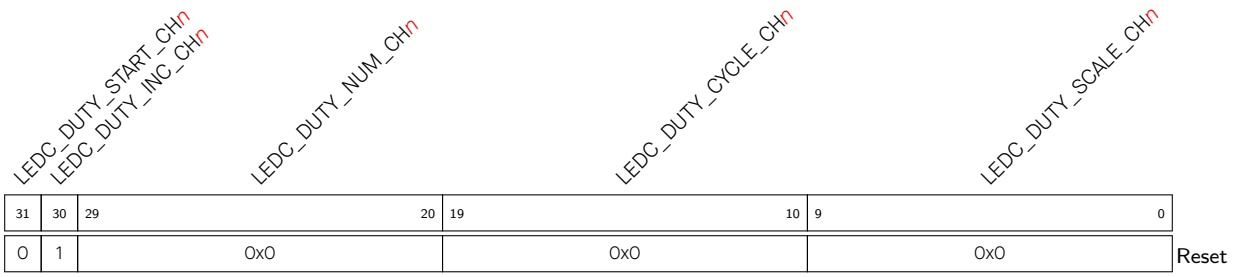
LEDC_OVF_NUM_CH*n* 用于配置定时器溢出次数的最大值减 1。通道 *n* 的定时器溢出次数达到 (LEDC_OVF_NUM_CH*n* +1) 次时，触发 LEDC_OVF_CNT_CH*n*_INT 中断。(R/W)

LEDC_OVF_CNT_EN_CH*n* 用于计算通道 *n* 选择的定时器溢出的次数。(R/W)

LEDC_OVF_CNT_RESET_CH*n* 置位此位，复位通道 *n* 的定时器溢出计数器。(WO)

LEDC_OVF_CNT_RESET_ST_CH*n* LEDC_OVF_CNT_RESET_CH*n* 的状态位。(RO)

Register 35.2. LEDC_CH*n*_CONF1_REG (*n*: 0-7) (0x000C+0x14**n*)



LEDC_DUTY_SCALE_CH*n* 用于配置通道 *n* 占空比的变化步长。(R/W)

LEDC_DUTY_CYCLE_CH*n* 通道 *n* 占空比每隔 LEDC_DUTY_CYCLE_CH*n* 变化一次。(R/W)

LEDC_DUTY_NUM_CH*n* 用于控制占空比变化的次数。(R/W)

LEDC_DUTY_INC_CH*n* 用于递增或递减通道 *n* 输出信号的占空比。1：递增；0：递减。(R/W)

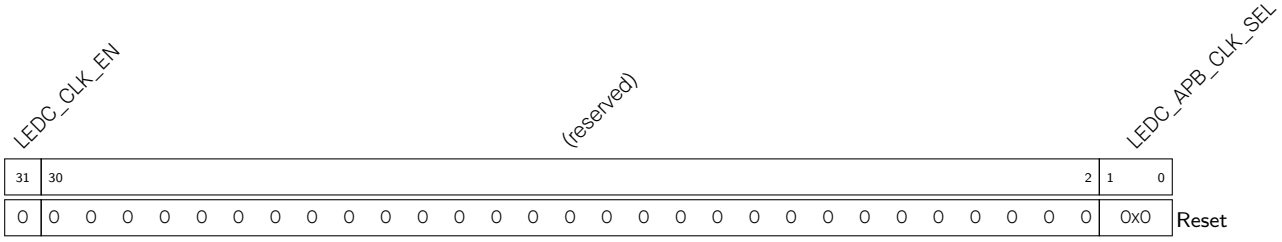
LEDC_DUTY_START_CH*n* 配置是否开启占空比渐变。

0：关闭

1：开启

(R/W)

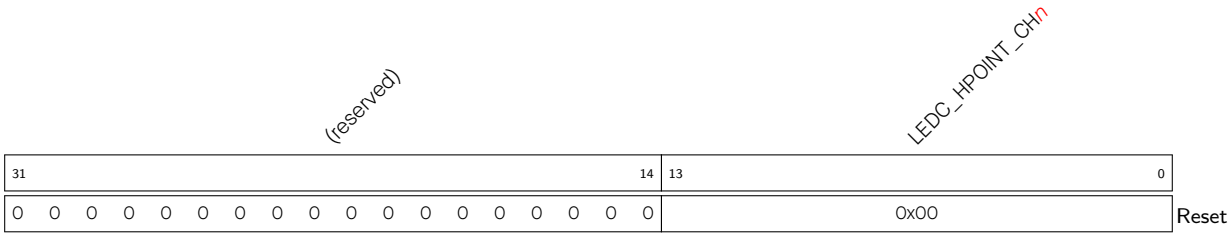
Register 35.3. LEDC_CONF_REG (0x00D0)



LEDC_APB_CLK_SEL 用于设置 4 个定时器的共同时钟源。1：APB_CLK；2：RC_FAST_CLK；3：XTAL_CLK。(读/写)

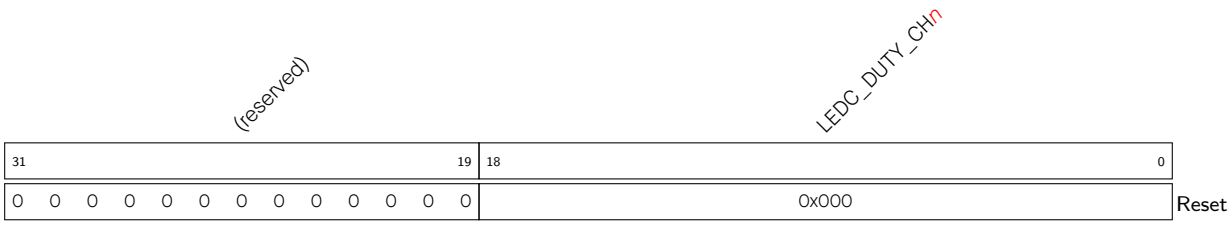
LEDC_CLK_EN 用于控制时钟。1：强制开启寄存器时钟。0：仅在应用写寄存器时支持时钟。(读/写)

Register 35.4. LEDC_CH*n*_HPOINT_REG (*n*: 0-7) (0x0004+0x14*n)



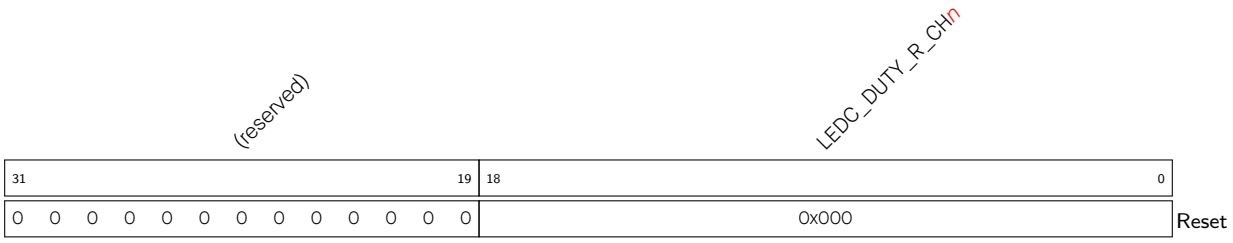
LEDC_HPOINT_CH*n* 所选定时器计数值达到该值时，输出信号翻转为高电平。(R/W)

Register 35.5. LEDC_CH*n*_DUTY_REG (*n*: 0-7) (0x0008+0x14*n)



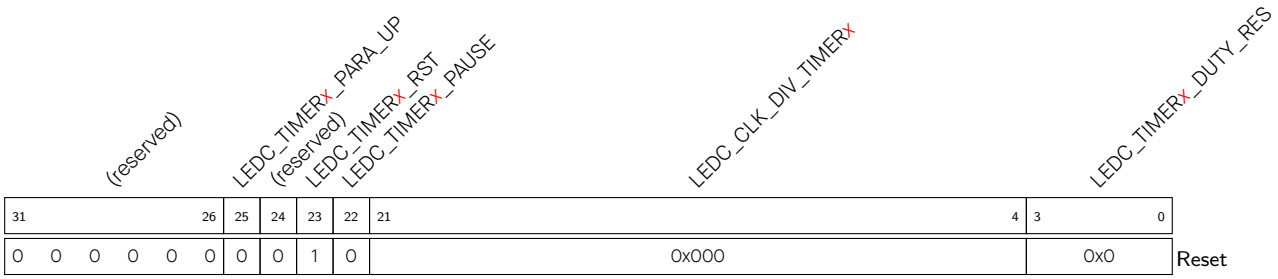
LEDC_DUTY_CH*n* 通过控制低位点改变输出信号占空比。所选定时器达到低位点时，输出信号翻转为低电平。(R/W)

Register 35.6. LEDC_CH*n*_DUTY_R_REG (*n*: 0-7) (0x0010+0x14**n*)



LEDC_DUTY_R_CH*n* 存储通道 *n* 输出信号的当前占空比。(RO)

Register 35.7. LEDC_TIMER*x*_CONF_REG (*x*: 0-3) (0x00A0+0x8**x*)



LEDC_TIMER*x*_DUTY_RES 用于控制定时器 *x* 计数器的计数范围。(R/W)

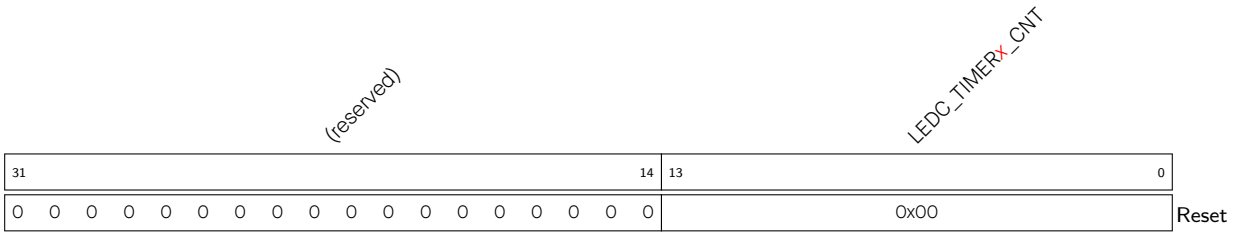
LEDC_CLK_DIV_TIMER*x* 用于配置定时器 *x* 分频器的分频系数。低 8 位为小数部分。(R/W)

LEDC_TIMER*x*_PAUSE 用于暂停定时器 *x* 的计数器。(R/W)

LEDC_TIMER*x*_RST 用于复位定时器 *x*。复位后计数器为 0。(R/W)

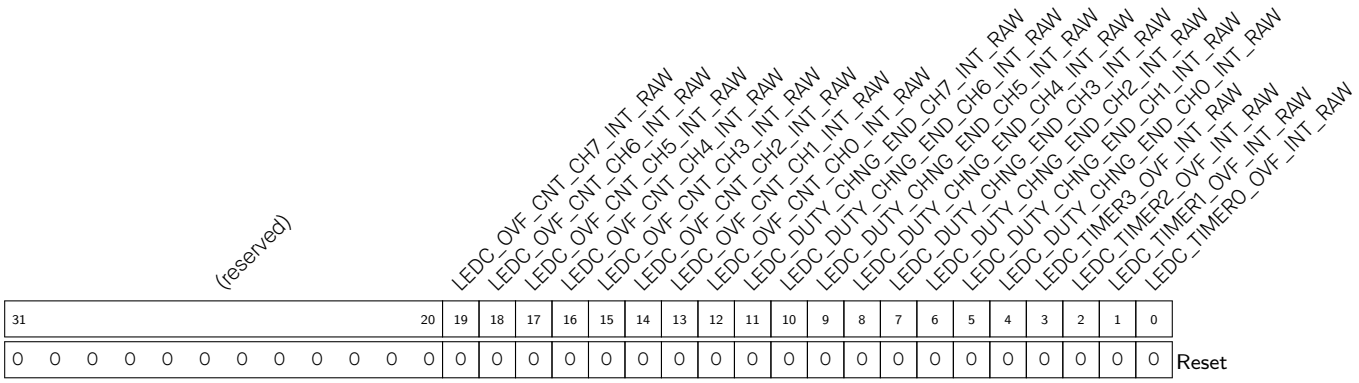
LEDC_TIMER*x*_PARA_UP 置位此位，更新 LEDC_CLK_DIV_TIMER*x* 和 LEDC_TIMER*x*_DUTY_RES。(WO)

Register 35.8. LEDC_TIMER*x*_VALUE_REG (*x*: 0-3) (0x00A4+0x8**x*)



LEDC_TIMER*x*_CNT 存储定时器 *x* 的当前计数器值 (RO)

Register 35.9. LEDC_INT_RAW_REG (0x00C0)

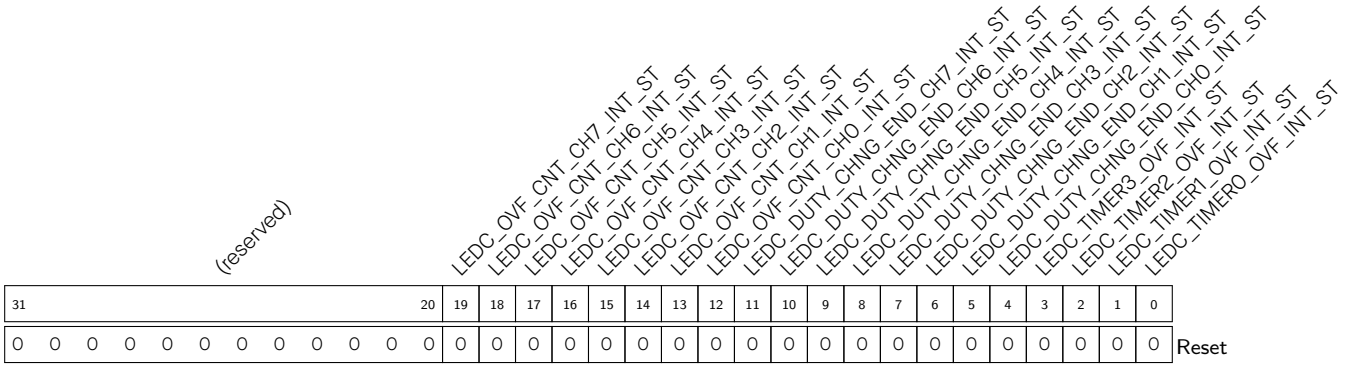


LEDC_TIMER*x*_OVF_INT_RAW 定时器 *x* 达到最大计数值时触发中断。(RO)

LEDC_DUTY_CHNG_END_CH*n*_INT_RAW 通道 *n* 的原始中断位。占空比渐变结束时触发。(RO)

LEDC_OVF_CNT_CH*n*_INT_RAW 通道 *n* 的原始中断位。ovf_cnt 达到 LEDC_OVF_NUM_CH*n* 指定值时触发。(RO)

Register 35.10. LEDC_INT_ST_REG (0x00C4)

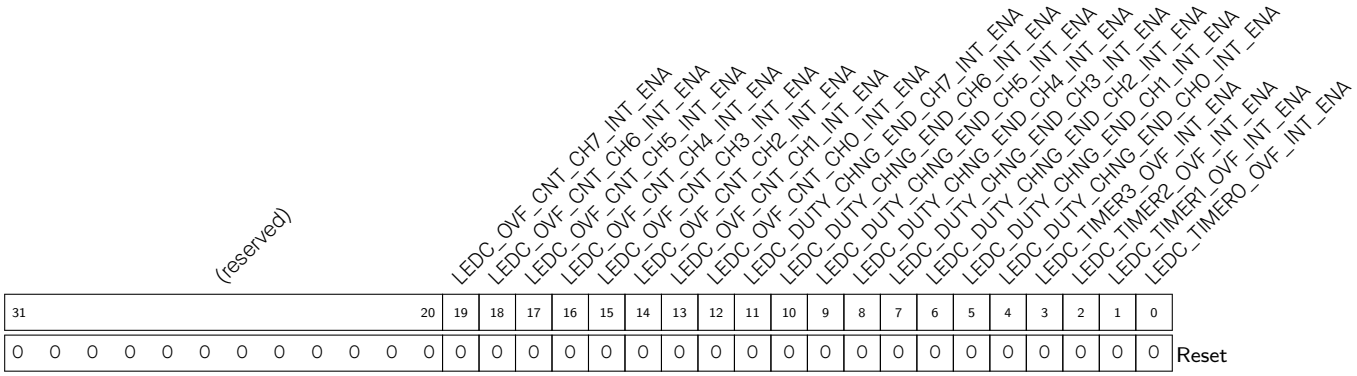


LEDC_TIMER*x*_OVF_INT_ST LEDC_TIMER*x*_OVF_INT_ENA 置 1 时，LEDC_TIMER*x*_OVF_INT 中断的屏蔽中断状态位。(RO)

LEDC_DUTY_CHNG_END_CH*n*_INT_ST LEDC_DUTY_CHNG_END_CH*n*_INT_ENA 置 1 时，LEDC_DUTY_CHNG_END_CH*n*_INT 中断的屏蔽中断状态位。(RO)

LEDC_OVF_CNT_CH*n*_INT_ST LEDC_OVF_CNT_CH*n*_INT_ENA 置 1 时，LEDC_OVF_CNT_CH*n*_INT 中断的屏蔽中断状态位。(RO)

Register 35.11. LEDC_INT_ENA_REG (0x00C8)

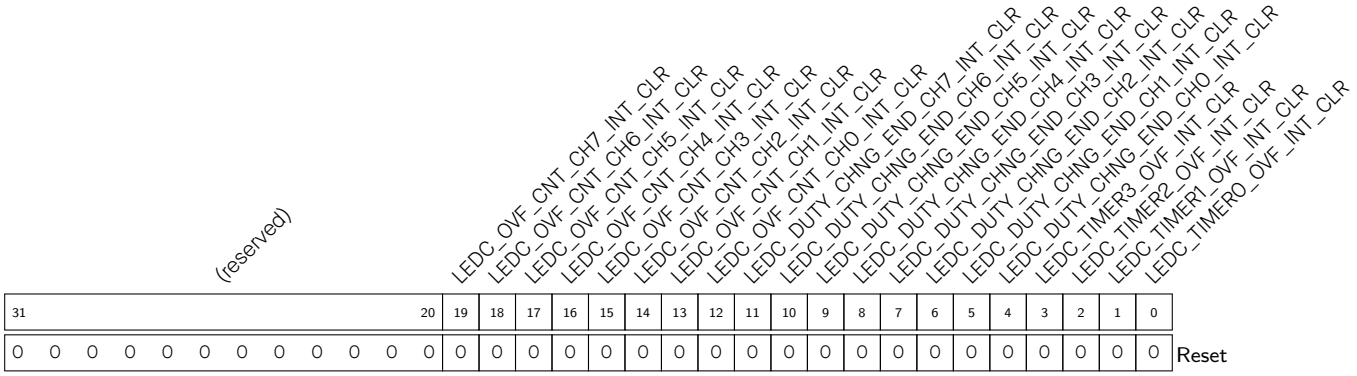


LEDC_TIMER*x*_OVF_INT_ENA LEDC_TIMER*x*_OVF_INT 中断的使能位。(R/W)

LEDC_DUTY_CHNG_END_CH*n*_INT_ENA LEDC_DUTY_CHNG_END_CH*n*_INT 中断的使能位。(R/W)

LEDC_OVF_CNT_CH*n*_INT_ENA LEDC_OVF_CNT_CH*n*_INT 中断的使能位。(R/W)

Register 35.12. LEDC_INT_CLR_REG (0x00CC)

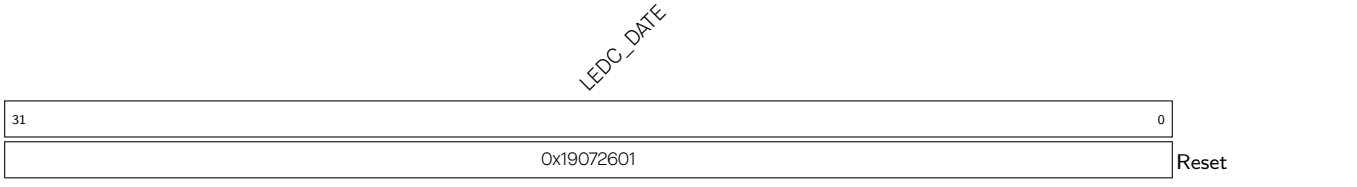


LEDC_TIMER*x*_OVF_INT_CLR 置位此位，清除 LEDC_TIMER*x*_OVF_INT 中断。(WO)

LEDC_DUTY_CHNG_END_CH*n*_INT_CLR 置位此位，清除 LEDC_DUTY_CHNG_END_CH*n*_INT 中断。(WO)

LEDC_OVF_CNT_CH*n*_INT_CLR 置位此位，清除 LEDC_OVF_CNT_CH*n*_INT 中断。(WO)

Register 35.13. LEDC_DATE_REG (0x00FC)



LEDC_DATE 版本控制寄存器。(R/W)

第 36 章

电机控制脉宽调制器 (MCPWM)

36.1 概述

电机控制脉宽调制器 (MCPWM) 外设用于电机和电源控制。该外设提供了 6 个 PWM 输出，可在几种拓扑结构中运行。常见的拓扑结构之一是用一对 PWM 输出来驱动 H 桥以控制电机旋转速度和旋转方向。

MCPWM 的时序和控制资源主要分为两种子模块：PWM 定时器和 PWM 操作器。每个 PWM 定时器提供参考时序，可以自由运行，或同步到其他定时器或外部源。每个 PWM 操作器具有为一个 PWM 通道生成波形对的所有控制资源。MCPWM 外设还包含专用捕获模块，用于需要精确定时外部事件的系统。

ESP32-S3 有两个 MCPWM 外设，分别是 MCPWM0 和 MCPWM1。

36.2 主要特性

每个 MCPWM 外设都有一个时钟分频器（预分频器），三个 PWM 定时器，三个 PWM 操作器和一个捕获模块。图 36.2-1 描述了 MCPWM 内的模块和接口上的信号。PWM 定时器用于生成定时参考。PWM 操作器将根据定时参考生成所需的波形。通过配置，任一 PWM 操作器可以使用任一 PWM 定时器的定时参考。不同的 PWM 操作器可以使用相同的 PWM 定时器的定时参考来产生 PWM 信号。此外，不同的 PWM 操作器也可以使用不同的 PWM 定时器的值来生成单独的 PWM 信号。不同的 PWM 定时器也可进行同步。

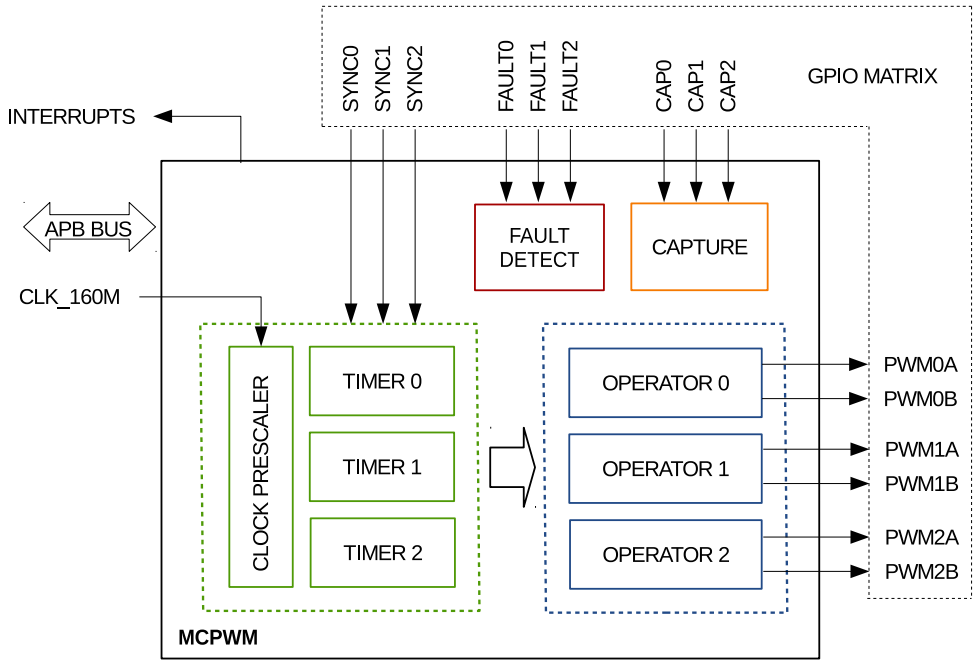


图 36.2-1. MCPWM 外设概览

以下是图 36.2-1 中模块的功能概述：

- PWM 定时器 0、1 和 2
  - 每个 PWM 定时器都有一个专用的 8 位时钟预分频器。
  - PWM 定时器中的 16 位计数器的工作模式包括：递增计数模式，递减计数模式，递增递减循环计数模式。
  - 硬件/软件同步可以触发 PWM 定时器重载，重载值位于相位寄存器中；同时触发预分频的重启，从而同步定时器的时钟。硬件同步源可以来自任何 GPIO 或任何其他 PWM 定时器的 sync_out 信号。软件同步源通过向 MCPWM_TIMER*x*_SYNC_SW 位写入取反值获取。
- PWM 操作器 0，1 和 2
  - 每个 PWM 操作器有两个 PWM 输出（PWM*x*A 和 PWM*x*B），可以在对称和非对称配置中独立工作。
  - 可以通过异步方式更新对 PWM 信号的控制。
  - 死区时间在上升沿和下降沿可配置，并可分别设置。
  - 所有事件都可触发 CPU 中断。
  - 通过高频载波信号调制 PWM 输出，在使用变压器隔离栅极驱动器时可发挥巨大作用。
  - 周期、时间戳寄存器和其他主要的控制寄存器有影子寄存器，具有灵活的更新方式。
- 故障检测模块
  - 出现故障时，可选择在逐周期模式或一次性模式下处理。
  - 故障条件可强制 PWM 输出高或低电平。
- 捕获模块
  - 旋转电机的速度测量（例如，用霍尔传感器检测的齿形链轮）。
  - 位置传感器脉冲之间的间隔时间测量。
  - 脉冲序列信号的周期和占空比测量。
  - 从电流/电压传感器的占空比编码信号导出的解码电流或电压振幅。
  - 3 个独立的捕获通道，各具备一个 32 位的时间戳寄存器。
  - 输入捕获信号可以预分频，边沿极性可选。
  - 捕获定时器可以与 PWM 定时器或外部信号同步。
  - 3 个捕获通道上都可以产生中断。

36.3 模块

36.3.1 概述

以下提供 MCPWM 关键模块的主要配置参数。调整特定参数，例如 PWM 定时器的同步源，请参考章节 36.3.2。

36.3.1.1 预分频器模块



图 36.3-1. 预分频器模块

配置参数：

- 对 CRYPTO_PWM_CLK 进行分频。

36.3.1.2 定时器模块

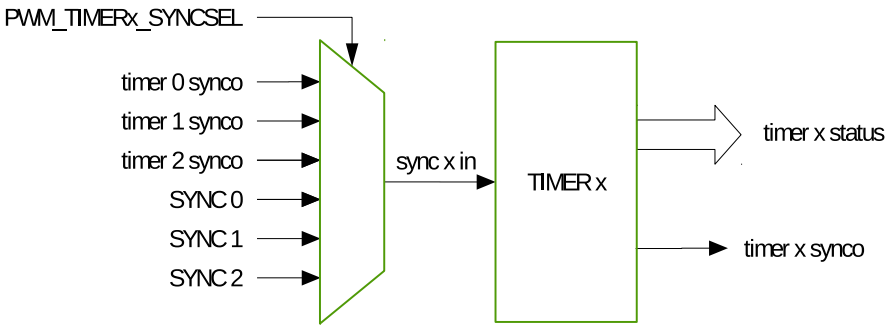


图 36.3-2. 定时器模块

配置参数：

- 设置 PWM 定时器的频率或周期。
- 配置定时器的工作模式：
  - 递增计数模式：用于非对称 PWM 输出
  - 递减计数模式：用于非对称 PWM 输出
  - 递增递减循环计数模式：用于于对称 PWM 输出
- 配置软件或硬件同步发生时的重载相位，包括值和方向。
- 通过硬件或软件同步使 PWM 定时器彼此同步。
- 设置 PWM 定时器的同步输入源，共 7 个可选输入源：

  - 3 个 PWM 定时器的同步输出
  - 来自 GPIO 交换矩阵的 3 个同步信号：PWMn_SYNC0_IN、PWMn_SYNC1_IN、PWMn_SYNC2_IN
  - 未选择同步输入信号
- 配置 PWM 定时器的同步输出源，共 4 个可选输出源：
  - 同步输入信号
  - PWM 定时器的值为 0 时生成的事件
  - PMW 定时器的值与时钟周期的值相同时生成的事件
  - 向 MCPWM_TIMERx_SYNC_SW 位写入取反值时生成的事件
- 配置周期更新方式。

36.3.1.3 操作器模块

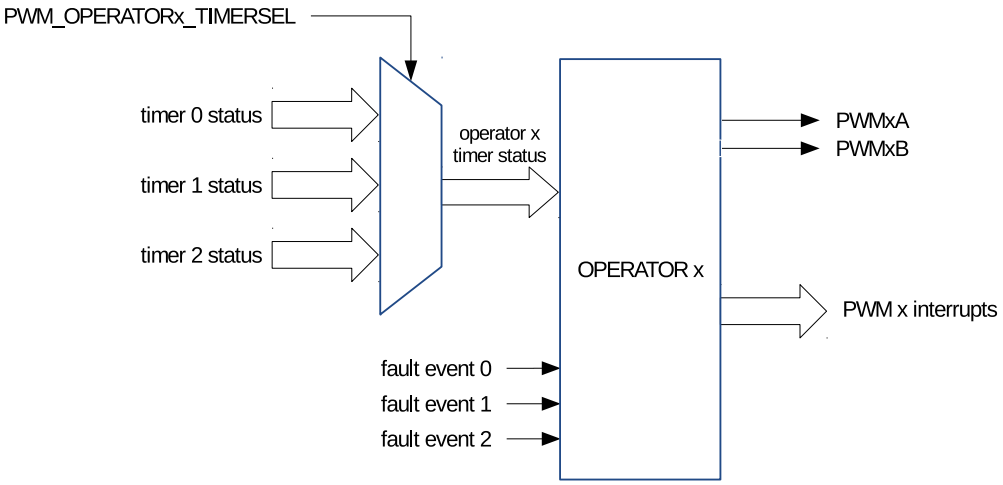


图 36.3-3. 操作器模块

表 36.3-1 列举操作器模块的主要配置参数。

表 36.3-1. 操作器模块的配置参数

模块	配置参数/选项
PWM 生成器	• 设置 PWM*x*A 和/或 PWM*x*B 输出的 PWM 占空比 • 设置定时事件发生的时间 • 设置发生定时事件时采取的行动： – 改变 PWM*x*A 和/或 PWM*x*B 输出为高或低 – 将 PWM*x*A 和/或 PWM*x*B 取反 – 不对输出执行任何操作 • 通过直接软件控制强制 PWM 输出的状态 • 在 PWM 输出的上升和/或下降边沿上增加死区 • 配置该模块的更新方式
死区生成器	• 控制高侧和低侧开关之间的互补死区关系 • 指定上升沿死区 • 指定下降沿死区 • 绕过死区发生器模块，PWM 波形不插入死区 • 可根据 PWM*x*A 输出进行 PWM*x*B 相移 • 配置该模块的更新方式
PWM 载波	• 使能载波，设置载波频率 • 设置载波波形中第一个脉冲的持续时间 • 设置第二个以及之后的脉冲的占空比 • 绕过 PWM 载波模块，PWM 波形无变动
故障处理器	• 配置 PWM 模块是否以及如何响应故障事件信号 • 指定发生故障事件时采取的操作： – 强制 PWM*x*A 和/或 PWM*x*B 为高电平 – 强制 PWM*x*A 和/或 PWM*x*B 为低电平 – 配置 PWM*x*A 和/或 PWM*x*B 忽略任何故障事件 • 配置 PWM 应对故障事件的间隔模式： – 一次性模式 – 逐周期模式 • 生成中断 • 绕过故障处理器模块 • 设置逐周期操作清除的方式 • 当时基计数器采取向上和向下时，可采取不同操作

36.3.1.4 故障检测模块

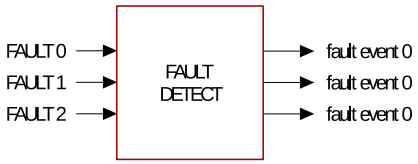


图 36.3-4. 故障检测模块

配置参数：

- 开启故障事件的生成，并为每个故障信号配置故障事件生成的极性
- 生成故障事件中断

36.3.1.5 捕获模块

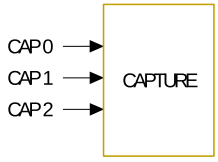


图 36.3-5. 捕获模块

配置参数：

- 选择捕获模块输入的边沿极性和预分频
- 设置软件触发捕获
- 配置捕获定时器同步触发和同步相位
- 软件同步捕获定时器

36.3.2 PWM 定时器模块

每个 MCPWM 外设都有三个 PWM 定时器模块。它们中的任何一个都可以决定三个 PWM 操作器模块中任意一个的必要事件时序。通过使用 GPIO 交换矩阵的同步信号，内置同步逻辑允许一个或多个 MCPWM 外设中的多个 PWM 定时器模块作为一个系统协同工作。

36.3.2.1 PWM 定时器模块的配置

用户可配置 PWM 定时器模块的以下功能：

- 通过指定 PWM 定时器频率或周期来控制事件发生的频率。
- 配置某个 PWM 定时器与其他 PWM 定时器或模块同步。
- 使 PWM 定时器与其他 PWM 定时器或模块同相。
- 设置定时器计数模式：递增，递减，或递增递减循环计数模式。
- 使用预分频器更改 PWM 定时器时钟（PT_clk）的速率。每个定时器都有自己的预分频器，通过寄存器 MCPWM_TIMER0_CFG0_REG 的 MCPWM_TIMER*x*_PRESCALE 配置。PWM 定时器根据该寄存器的设置以较慢的速度递增或递减。

36.3.2.2 PWM 定时器工作模式和定时事件生成

PWM 定时器有三种工作模式，由 PWM*x* 定时器模式字段配置：

- 递增计数模式：

定时器从零增加到周期字段中配置的值。一旦到达周期值，PWM 定时器清零，并再次开始递增。PWM 周期等于周期寄存器中的周期值 + 1。

说明：周期寄存器为 MCPWM_TIMER*x*_PERIOD (*x* = 0, 1, 2)，对应 MCPWM_TIMER0_PERIOD、MCPWM_TIMER1_PERIOD、MCPWM_TIMER2_PERIOD。

- 递减计数模式：
  PWM 定时器从周期寄存器中的值开始递减到零。达到零后，将恢复为周期值，再次开始递减。在这种情况下，PWM 周期等于周期寄存器中的周期值 + 1。
- 递增-递减循环模式：
  此模式结合了上述两种模式。PWM 定时器从零开始递增，直到达到周期值，再次递减为零。PWM 定时器按照此模式循环递增递减。PWM 周期为（周期寄存器的周期值 × 2 + 1）。

图 36.3-6 至 36.3-9 显示不同的模式下 PWM 定时器波形，包括同步事件期间的定时器行为。递增计数模式中，同步后永远保持递增计数；递减计数模式中，同步后永远保持递减计数；递增-递减循环模式中，同步后由 MCPWM_TIMER*x*_PHASE_DIRECTION 配置计数方向。

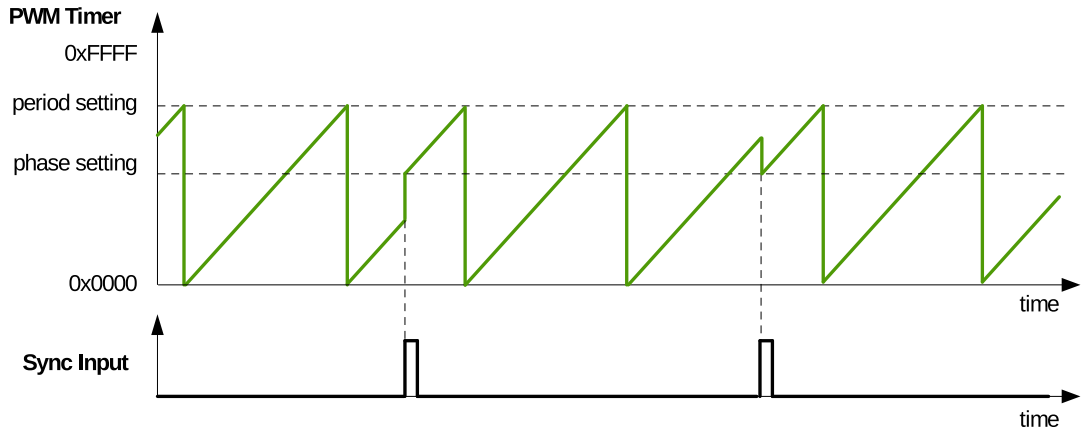


图 36.3-6. 递增计数模式波形

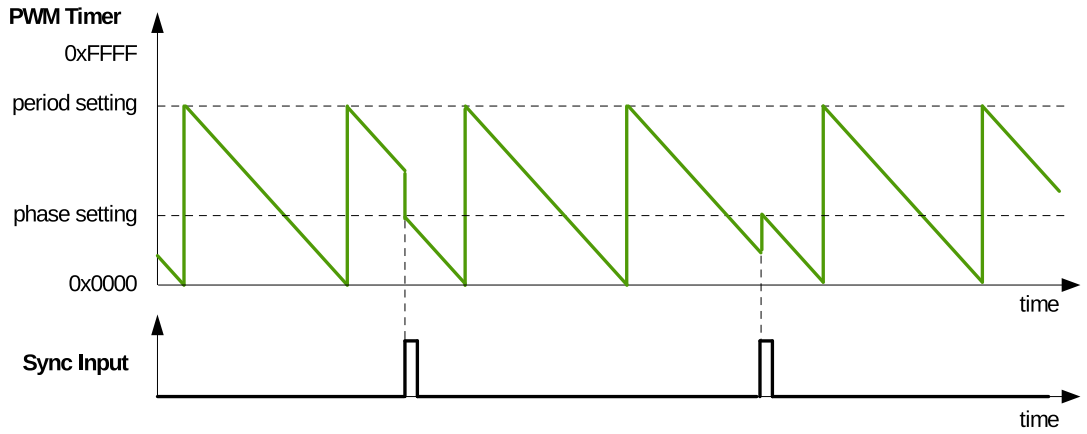


图 36.3-7. 递减计数模式波形

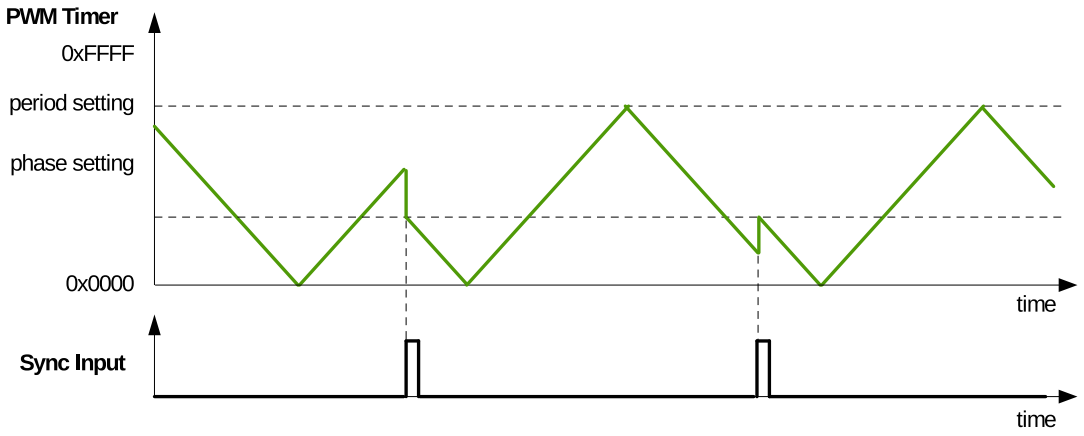


图 36.3-8. 递增递减循环模式波形，同步事件后递减

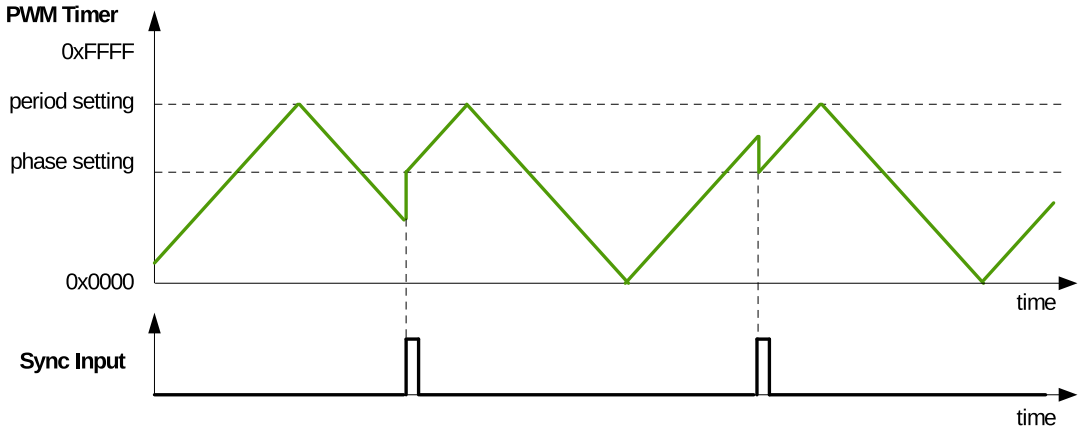


图 36.3-9. 递增递减循环模式波形，同步事件后递增

PWM 定时器运行时，定期自动生成以下定时事件：

- UTEP
  当 PWM 定时器等于周期字段的值 (MCPWM_TIMERx_PERIOD) 且 PWM 定时器递增计数时生成的定时事件。
- UTEZ
  当 PWM 定时器等于零且 PWM 定时器递增计数时生成的定时事件。
- DTEP
  当 PWM 定时器等于周期字段的值 (MCPWM_TIMERx_PERIOD) 且 PWM 定时器递减时生成的定时事件。
- DTEZ
  当 PWM 定时器等于零且 PWM 定时器递减时生成的定时事件。

图 36.3-10 至 36.3-12 为 U/DTEP 和 U/DTEZ 定时事件的时序波形。

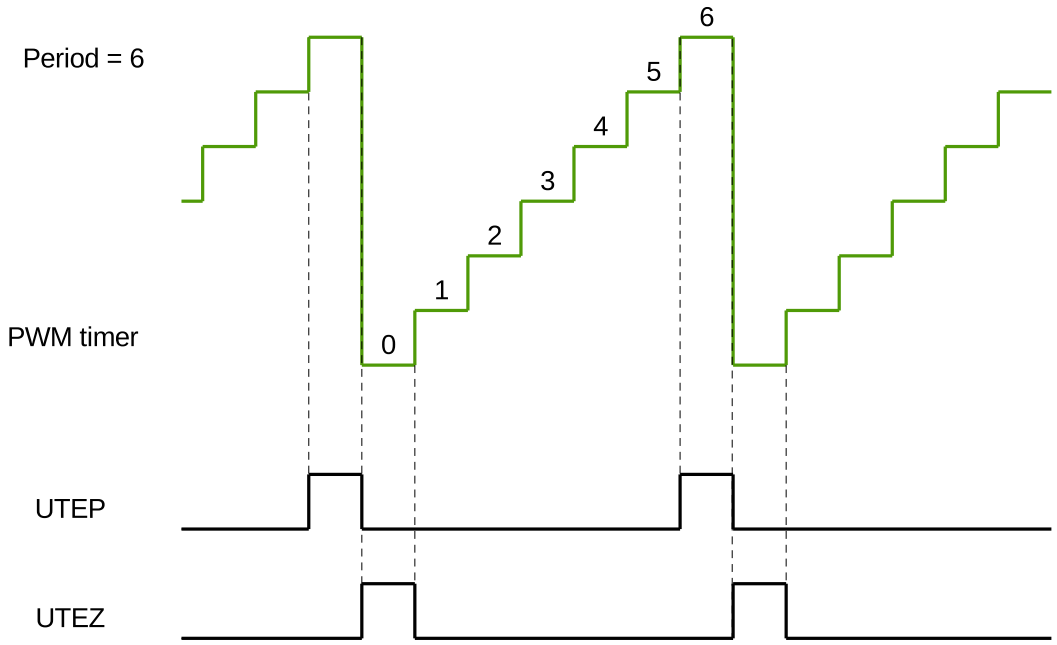


图 36.3-10. 递增模式中生成的 UTEP 和 UTEZ

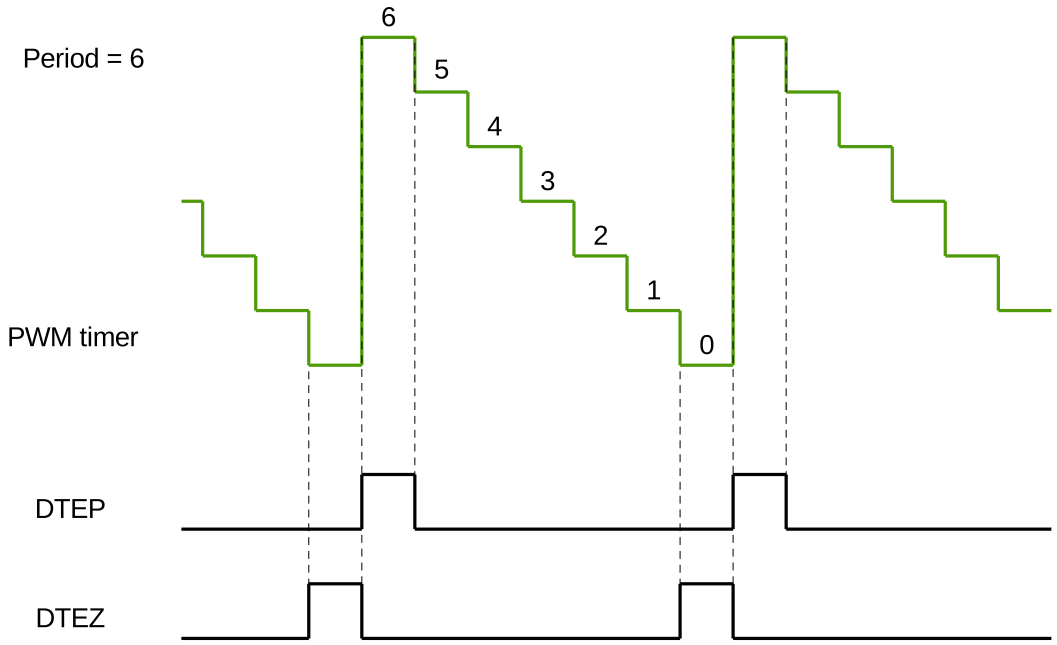


图 36.3-11. 递减模式中生成的 DTEP 和 DTEZ

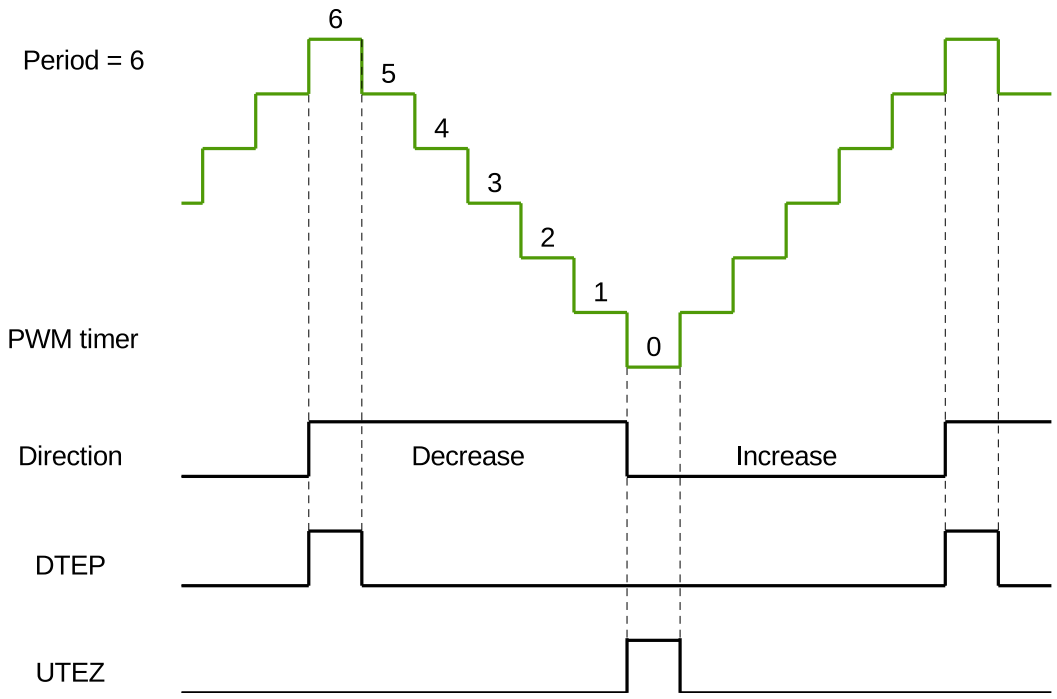


图 36.3-12. 递增递减模式中生成的 DTEP 和 UTEZ

需注意，在递增递减模式下，当定时器递增计数时，定时器范围为 [0，周期值 - 1]；递减计数时，定时器范围为 [周期值，1]。也就是说，在该模式下，将定时器同步为 0 时，则不支持递减计数，即 MCPWM_TIMER*n*_PHASE_DIRECTION 不能为 1；定时器同步为周期值时，也不支持递增计数，即 MCPWM_TIMER*n*_PHASE_DIRECTION 不能为 0。所以，当定时器同步为 0 时，计数方向只能为递增，即将 MCPWM_TIMER*n*_PHASE_DIRECTION 配置为 0；定时器同步为周期值时，计数方向只能为递减，即将 MCPWM_TIMER*n*_PHASE_DIRECTION 配置为 1。

36.3.2.3 PWM 定时器影子寄存器

PWM 定时器周期寄存器和 PWM 定时器时钟预分频器寄存器具有影子寄存器。影子寄存器的作用是备份即将写入有效寄存器的值，并在硬件同步的特定时刻写入有效寄存器。两种寄存器类型定义如下：

- 有效寄存器
  有效寄存器直接控制硬件执行的所有操作。
- 影子寄存器
  影子寄存器充当存储即将写入有效寄存器的值的临时缓冲区。在用户配置的某个时间点，影子寄存器中的值被写入有效寄存器。在此之前，影子寄存器的内容对受控硬件没有任何直接影响。这有助于防止寄存器由软件异步修改时可能发生的错误硬件操作。影子寄存器和有效寄存器具有相同的存储器地址。软件写入或读取影子寄存器。

  当定时器开始工作时，时钟预分频器的有效寄存器更新。当 MCPWM_GLOBAL_UP_EN 置 1 时，可通过以下方式选择更新有效寄存器的时间点：将 MCPWM_TIMER*x*_PERIOD_UPMETHOD 置 1，当 PWM 定时器值为 0 时，开始更新，当 PWM 定时器值等于周期时，同步或立即更新；软件也可以触发全局强制更新位 MCPWM_GLOBAL_FORCE_UP，该位将触发模块中的所有寄存器根据影子寄存器进行更新。

36.3.2.4 PWM 定时器同步和锁相

PWM 模块采用灵活的同步方法。每个 PWM 定时器都有一个同步输入和一个同步输出。同步输入可以从 GPIO 矩阵的三个同步输出和三个同步信号中选择。同步输出可以使用同步输入信号、在 PWM 定时器等于周期、PWM 定时器等于零或软件同步时产生。因此，PWM 定时器可以通过相位锁定而相连。在同步期间，PWM 定时器时钟预分频器将复位其计数器，以同步 PWM 定时器时钟。

36.3.3 PWM 操作器模块

PWM 操作器模块具备以下功能：

- 根据相应 PWM 定时器的定时参考生成 PWM 信号对。
- PWM 信号对的每个信号都可以独立设置特定的死区时间。
- 可通过配置将载波叠加到 PWM 信号上。
- 故障条件下处理响应。

图 36.3-13 为 PWM 操作器的框图。

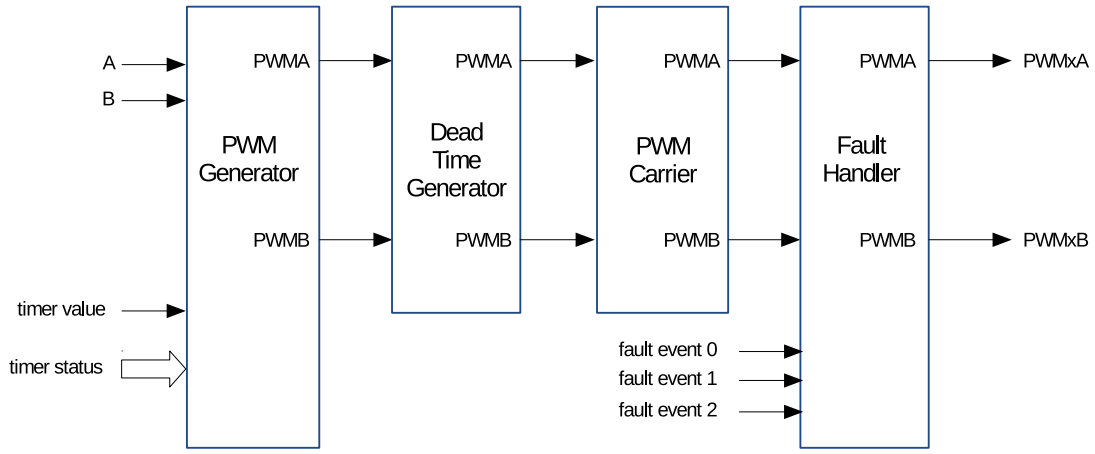


图 36.3-13. PWM 操作器的子模块

36.3.3.1 PWM 生成器模块

PWM 生成器模块的作用

此模块中生成或导入重要的时序事件，并转化为特定操作，在 PWM*x*A 和 PWM*x*B 输出处生成所需的波形。

PWM 生成器模块执行以下操作：

- 基于使用寄存器 A 和 B 配置的时间戳生成定时事件。满足以下条件时发生定时事件：
  - UTEA：PWM 定时器递增计数并且其值等于寄存器 A。
  - UTEB：PWM 定时器递增计数并且其值等于寄存器 B。
  - DTEA：PWM 定时器递减计数并且其值等于寄存器 A。

  - DTEB：PWM 定时器递减计数并且其值等于寄存器 B。
- 基于故障或同步事件生成 U/DT1，U/DT2 定时事件。
- 当这些定时事件同时发生时管理优先级。
- 基于定时事件产生置 1，置 0 和取反操作。
- 根据 PWM 生成器模块的配置来控制 PWM 占空比。
- 使用影子寄存器处理新的时间戳值，以防止 PWM 波形中的毛刺干扰。

PWM 操作器影子寄存器

时间戳寄存器 A 和 B，以及操作配置寄存器 MCPWM_GEN*x*_A_REG 和 MCPWM_GEN*x*_B_REG 都有影子寄存器。影子寄存器提供了一种与硬件同步更新寄存器的方法。

当 MCPWM_GLOBAL_UP_EN 置 1 时，影子寄存器中的值可在某个特定时间写入有效寄存器中。时间戳寄存器 A 和 B 的更新方式字段分别为 MCPWM_GEN_A_UPMETHOD 和 MCPWM_GEN_B_UPMETHOD。MCPWM_GEN*x*_A_REG 和 MCPWM_GEN*x*_B_REG 的更新方式字段为 MCPWM_GEN_CFG_UPMETHOD。软件也可以触发全局强制更新位 MCPWM_GLOBAL_FORCE_UP，该位将触发模块中的所有寄存器根据影子寄存器进行更新。更多关于影子寄存器的描述请参考第 36.3.2.3 章。

定时事件

表 36.3-2 概括了所有定时信号和事件。

表 36.3-2. PWM 生成器中的所有定时事件

信号	事件描述	PWM 定时器操作
DTEP	PWM 定时器的值等于周期寄存器的值	PWM 定时器递减计数
DTEZ	PWM 定时器的值等于 0	
DTEA	PWM 定时器的值等于寄存器 A	
DTEB	PWM 定时器的值等于寄存器 B	
DTO 事件	基于故障或同步事件	
DT1 事件	基于故障或同步事件	
UTEP	PWM 定时器的值等于周期寄存器的值	PWM 定时器递增计数
UTEZ	定时器的值等于 0	
UTEA	PWM 定时器的值等于寄存器 A	
UTEB	PWM 定时器的值等于寄存器 B	
UTO 事件	基于故障或同步事件	
UT1 事件	基于故障或同步事件	
软件强制事件	软件触发的异步事件	-

软件强制事件用于在 PWM*x*A 和 PWM*x*B 输出上施加非连续或连续的强制电平。此更改是异步完成的。软件强制由寄存器 MCPWM_GEN*x*_FORCE_REG 控制。

PWM 生成器模块中 TO/T1 的选择和配置独立于故障处理模块中的故障事件的配置。跳闸事件可以不被配置为在故障处理器模块中引起跳闸动作，但相同的事件可以由 PWM 生成器用于触发 TO/T1 以控制 PWM 波形。

需要注意的是，当 PWM 定时器处于递增递减循环计数模式时，它将在 TEP 事件后递减，在 TEZ 事件后递增。因此，当 PWM 定时器处于此模式时，将出现 DTEP 和 UTEZ，但 UTEP 和 DTEZ 不会出现。

PWM 生成器可以同时处理多个事件。事件优先级由硬件决定，详见表 36.3-3 和表 36.3-4。优先级从 1（最高）到 7（最低）排列。需要注意的是，TEP 和 TEZ 事件的优先级取决于 PWM 定时器的计数模式。

如果 A 或 B 的值设置为大于周期，则 U/DTEA 和 U/DTEB 将永远不会发生。

表 36.3-3. PWM 定时器递增计数时，定时事件的优先级

优先级	事件
1（最高）	软件强制事件
2	UTEP
3	UTO
4	UT1
5	UTEB
6	UTEA
7（最低）	UTEZ

表 36.3-4. PWM 定时器递减计数时，定时事件的优先级

优先级	事件
1（最高）	软件强制事件
2	DTEZ
3	DTO
4	DT1
5	DTEB
6	DTEA
7（最低）	DTEP

说明：

1. UTEP 和 UTEZ 不同时发生。当 PWM 定时器处于递增计数模式，UTEP 将始终比 UTEZ 提前一个周期发生，如图 36.3-10 所示，因此它们对 PWM 信号的作用不会彼此干扰。当 PWM 定时器处于递增递减循环模式时，UTEP 不会发生。
2. DTEP 和 DTEZ 不同时发生。当 PWM 定时器处于递减计数模式时，DTEZ 始终比 DTEP 早一个周期发生，如图 36.3-11 所示，因此它们对 PWM 信号的作用不会彼此干扰。当 PWM 定时器处于递增递减循环模式时，DTEZ 不会发生。

PWM 信号生成

当某个定时事件发生时，PWM 生成器控制输出 PWM*x*A 和 PWM*x*B 的电平。定时事件通过 PWM 定时器计数方向（递增或递减）进一步限定。根据定时器计数方向，模块可以对 PWM 定时器递增或递减计数的阶段执行不同的操作。

可以在 PWM*x*A 和 PWM*x*B 输出上配置以下操作：

- 置为高电平：
  将 PWM*x*A 或 PWM*x*B 的输出设置为高电平。

- 置为低电平：
  通过将 PWM*x*A 或 PWM*x*B 的输出设置为低电平来清除 PWM*x*A 或 PWM*x*B 的输出。
- 取反：
  将 PWM*x*A 或 PWM*x*B 的当前输出电平更改为相反的值。如果它当前被拉高，则拉低，或反之。
- 不进行操作：
  保持 PWM*x*A 和 PWM*x*B 输出电平不变。在这种状态下，仍然可以触发中断。

输出上的操作通过寄存器 MCPWN_GEN*x*_A_REG 和 MCPWN_GEN*x*_B_REG 配置。每一次输出的操作都独立配置。此外，基于事件在某个输出上灵活地执行不同的操作。表 36.3-2 中列举的任何事件都可以作用于 PWM*x*A 或 PWM*x*B 输出上。关于生成器 0，1 或 2 的寄存器信息，请参考第 36.4 章。

常见配置的波形

图 36.3-14 为 PWM 定时器在递增递减循环计数时生成的对称 PWM 波形。该模式下的直流 0%-100% 调制可由以下公式获得：

$$Duty = (Period - A) \div Period$$

如果 A 的值等于 PWM 定时器的值，并且 PWM 定时器递增，则 PWM 输出被上拉。如果 A 的值在 PWM 定时器递减时等于 PWM 定时器的值，则 PWM 输出被拉低。

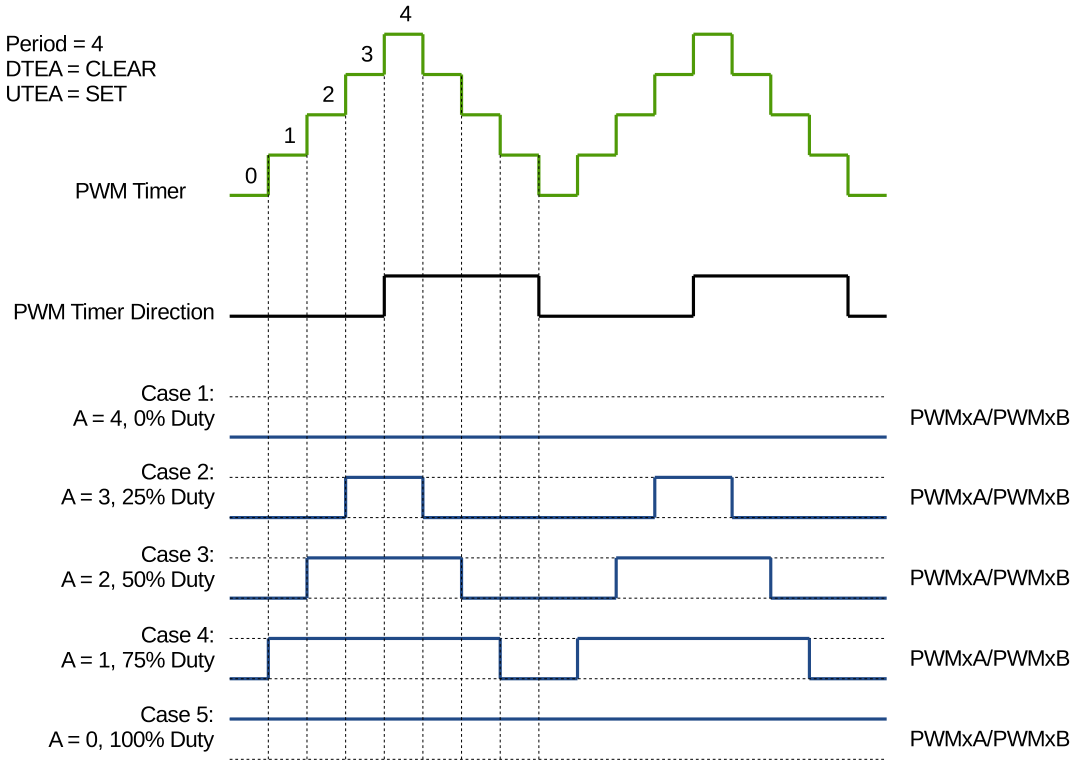


图 36.3-14. 递增递减模式下的对称波形

图 36.3-15 至图 36.3-18 的 PWM 波形描述了常见的 PWM 操作器配置。图中数据说明如下：

- Period A 和 B 分别表示写入周期寄存器 A 和 B 的值。
- PWM*x*A 和 PWM*x*B 是 PWM 操作器 *x* 的输出信号。

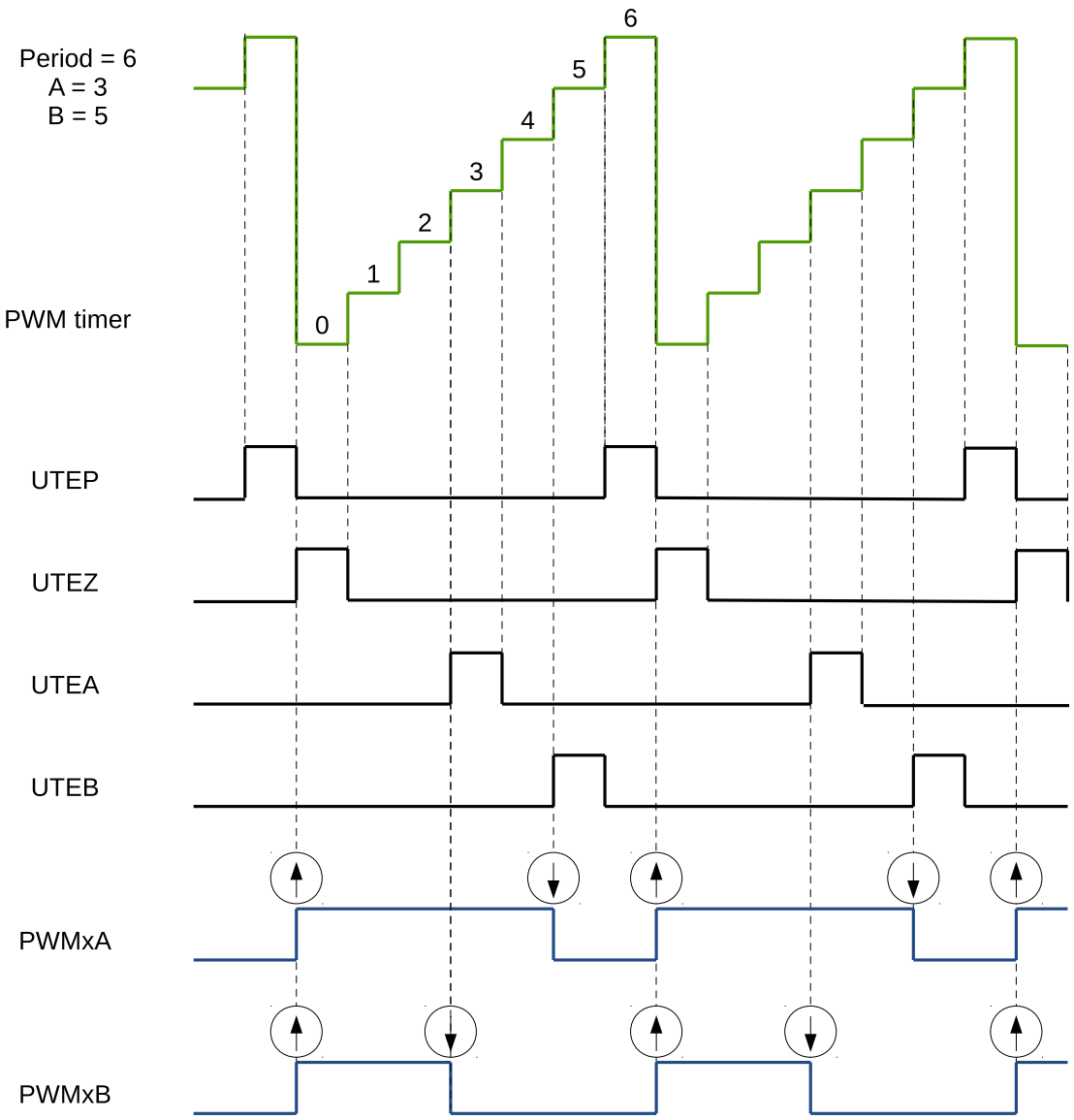


图 36.3-15. 递增计数模式，单边不对称波形，PWM*x*A 和 PWM*x*B 独立调制–高电平

PWM*x*A 的占空比调制由 B 设置，高电平有效，与 B 成正比。
PWM*x*B 的占空比调制由 A 设置，高电平有效，与 A 成正比。

$$Period = (MCPWM\_TIMERx\_PERIOD + 1) \times T_{PT\_clk}$$

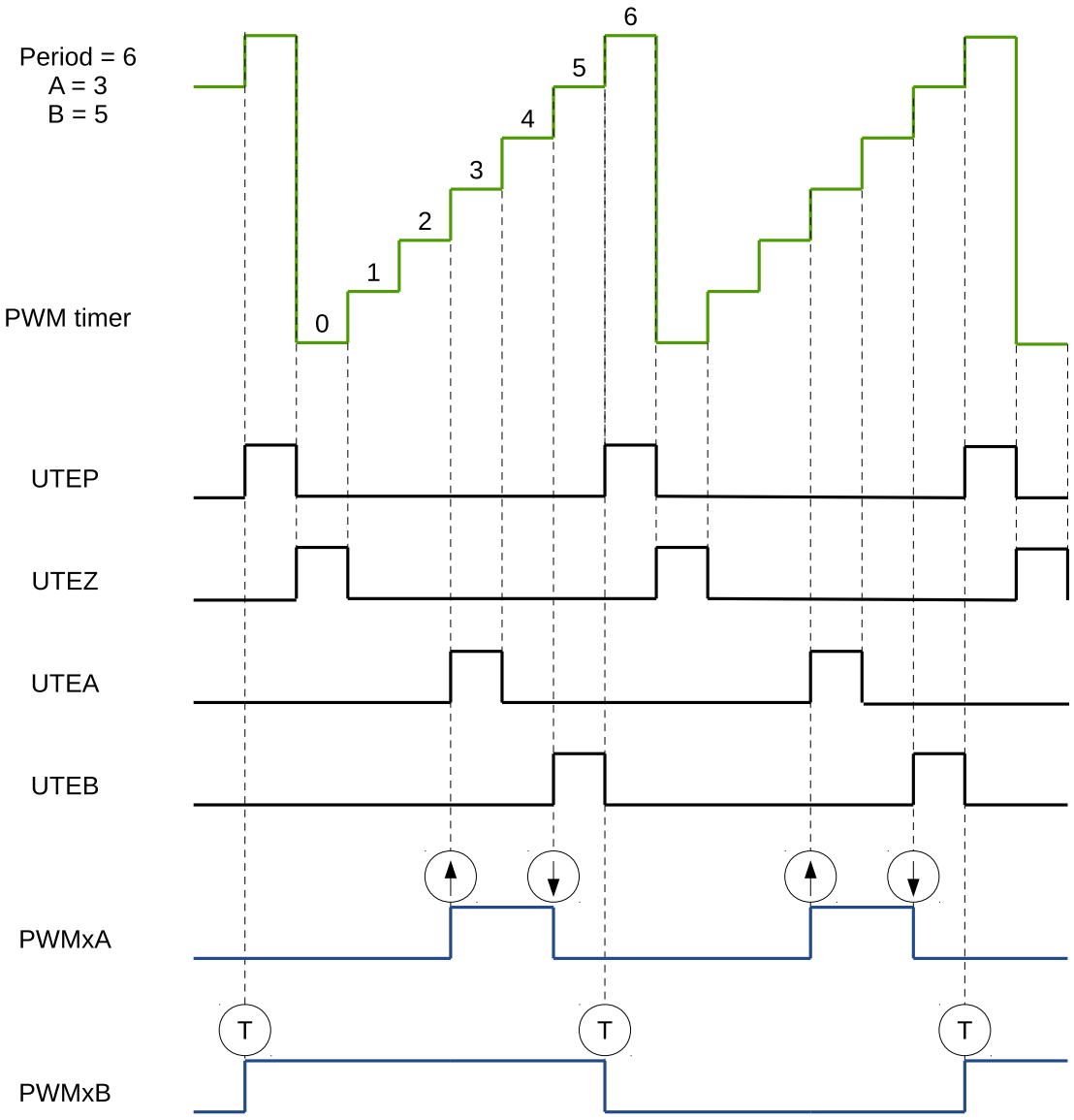


图 36.3-16. 递增计数模式，脉冲位置不对称波形，PWM*x*A 独立调制

脉冲可以在 PWM 波形内（零至周期值之间）的任何地方生成。

PWM*x*A 占空比与 (B –A) 成正比。

$$Period = (MCPWM\_TIMERx\_PERIOD + 1) \times T_{PT\_clk}$$

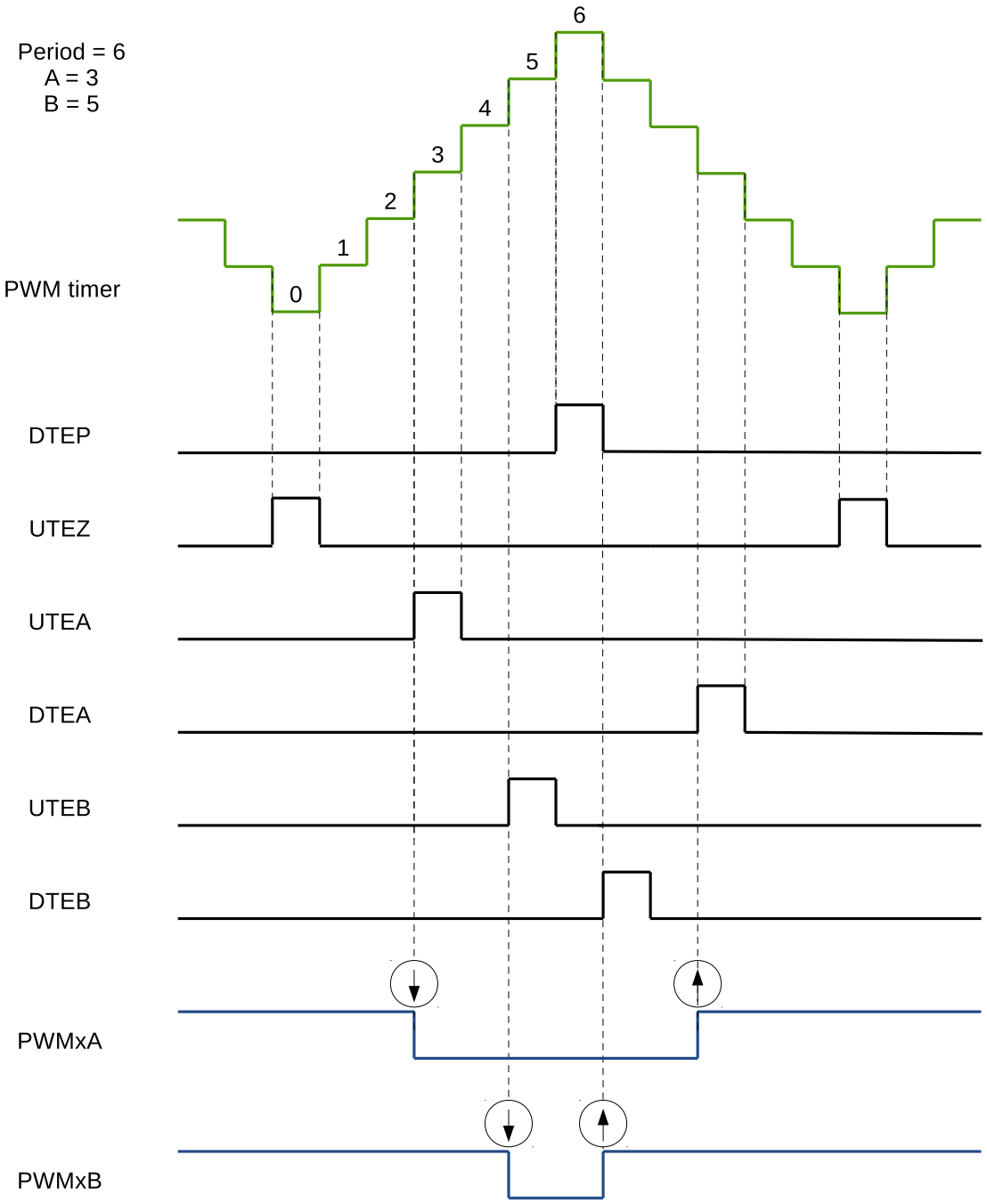


图 36.3-17. 递增递减循环计数模式，双沿对称波形，在 PWM*x*A 和 PWM*x*B 上独立调制–高电平有效

PWM*x*A 的占空比调制由 A 设置，高电平有效，与 A 成正比。
PWM*x*B 的占空比调制由 B 设置，高电平有效，与 B 成正比。
输出 PWM*x*A 和 PWM*x*B 可驱动不同开关。

$$Period = (2 \times PMCWM\_TIMERx\_PERIOD) \times T_{PT\_clk}$$

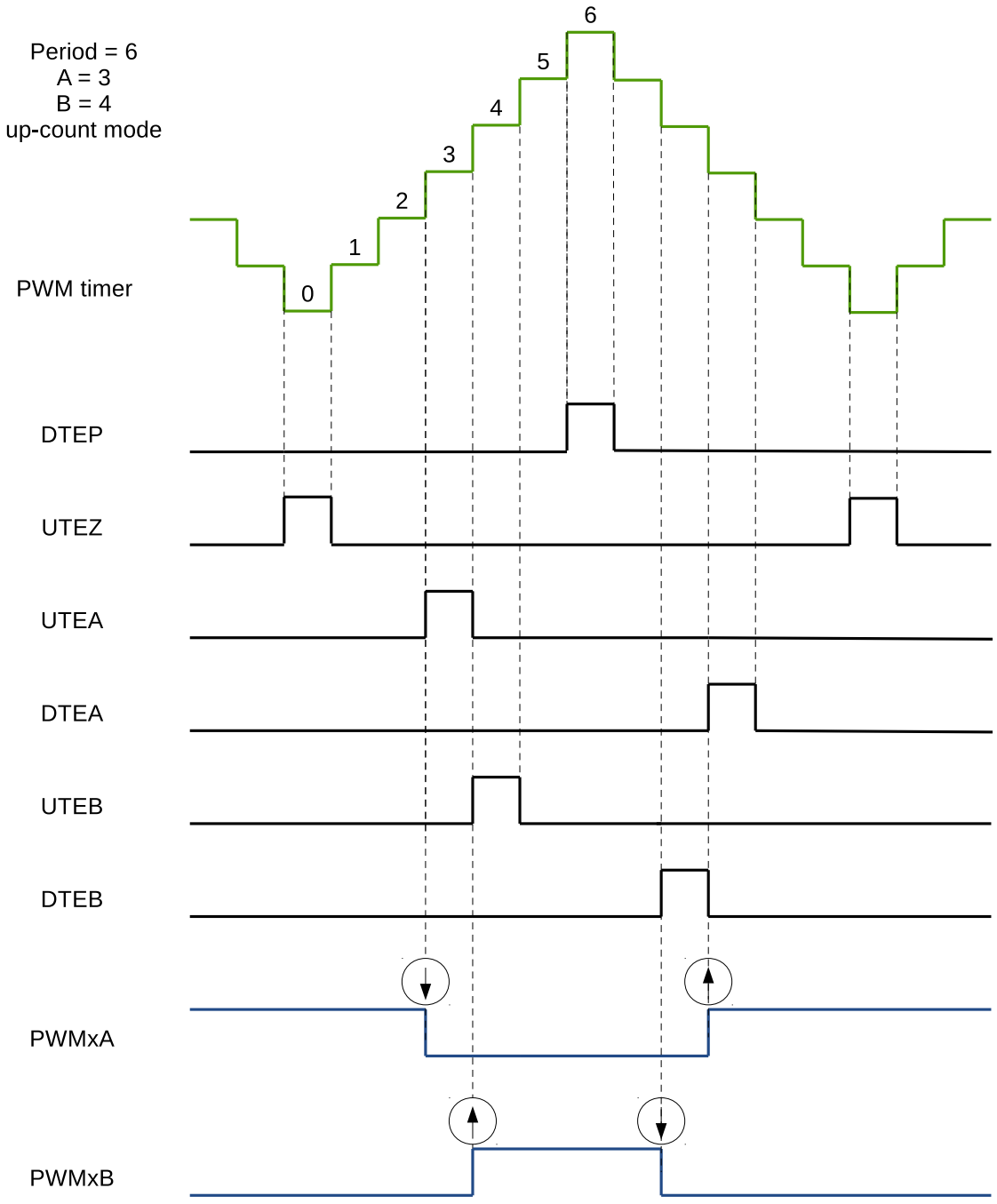


图 36.3-18. 递增递减循环计数模式，双沿对称波形，在 PWM*x*A 和 PWM*x*B 上独立调制–互补

PWM*x*A 的占空比调制由 A 设置，高电平有效，与 A 成正比。
PWM*x*B 的占空比调制由 B 设置，高电平有效，与 B 成正比。
PWM*x*A/B 输出可驱动上/下（互补）开关。
死区 = B –A，边沿位置完全可由软件配置。必要时，可使用死区生成器模块设置其他边沿延迟方式。

$$Period = (2 \times MCPWM\_TIMERx\_PERIOD) \times T_{PT\_clk}$$

软件强制事件

在 PWM 生成器内有 2 种软件强制事件：

- 非连续即时 (NCI) 软件强制事件
  当由软件触发时，这些类型的事件在 PWM 输出上立即生效。并且强制是不连续的，这意味着下一个激活的定时事件能够改变 PWM 输出。
- 连续 (CNTU) 软件强制事件
  这一类型事件是连续的。直到通过软件释放，强制 PWM 持续输出。事件触发器可配置。这一类事件可配置为定时或即时发生。

图 36.3-19 为 NCI 软件强制事件的一种波形。NCI 用于单独强制 PWM*x*A 输出为低电平，PWM*x*B 不受强制。

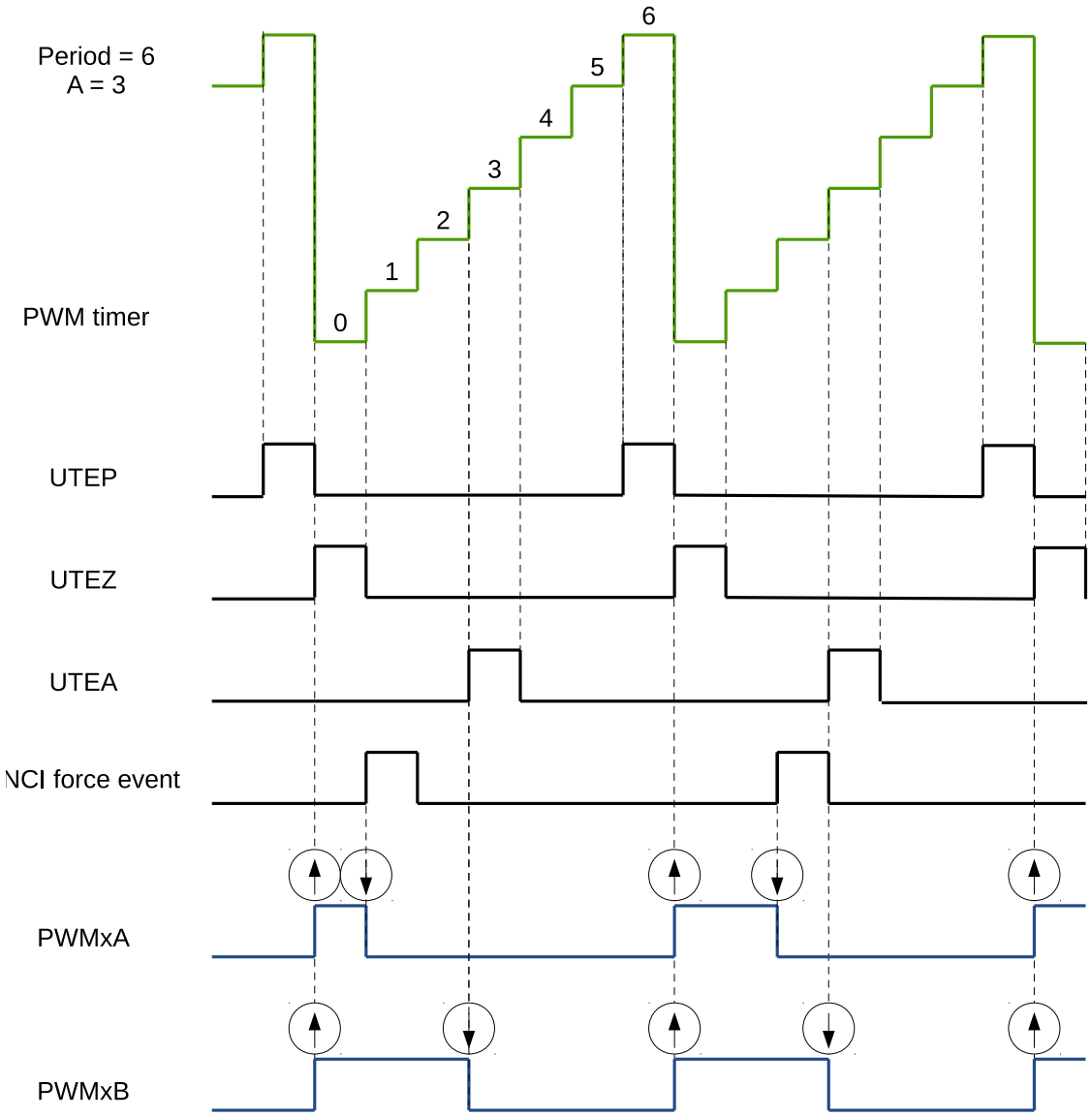


图 36.3-19. NCI 在 PWM*x*A 输出上软件强制事件示例

图 36.3-20 为 CNTU 软件强制事件的波形。UTEZ 事件被选为 CNTU 软件强制事件的触发器。CNTU 用于单独强制 PWM*x*B 输出为低电平，但 PWM*x*A 不受强制。

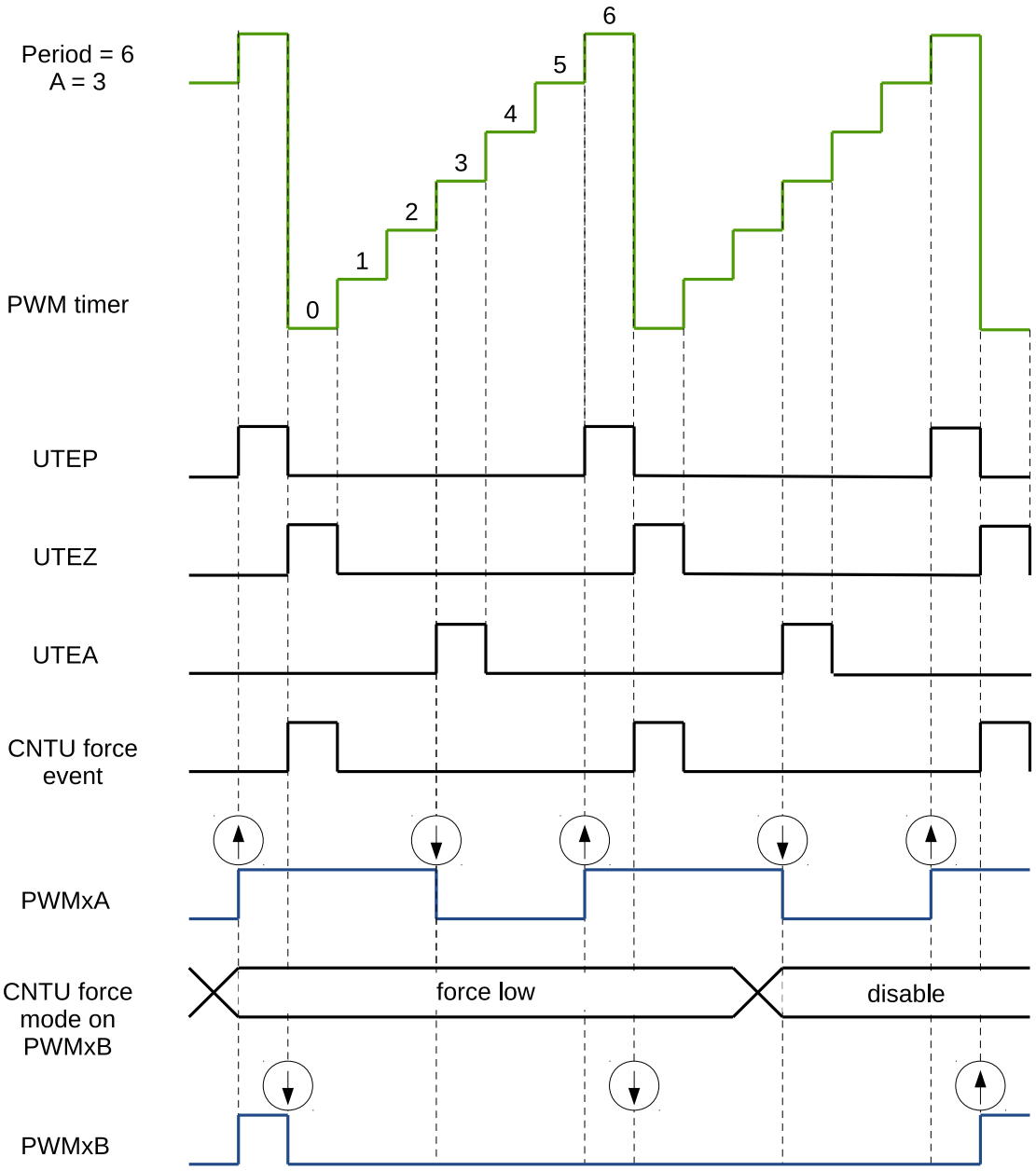


图 36.3-20. CNTU 在 PWM*x*B 输出上软件强制事件示例

36.3.3.2 死区生成器模块

死区生成器模块作用

章节 36.3.3.1 讲述了在 PWM*x*A 和 PWM*x*B 输出上生成信号的几种方式/事件，包括信号边沿的特定位置。通过改变信号之间的边沿位置以及设置信号的占空比，可获得所需的死区。另一种方式是使用专门的死区生成器模块控制死区。

死区生成器模块的主要功能如下：

- 根据单个 PWM*x*A 输入的死区生成信号对（PWM*x*A 和 PWM*x*B）
- 通过在信号边沿增加延迟来生成死区：
  - 上升沿延迟 (RED)
  - 下降沿延迟 (FED)
- 配置信号对为：
  - 高电平有效互补 (AHC)
  - 低电平有效互补 (ALC)
  - 高电平有效 (AH)
  - 低电平有效 (AL)
- 如果死区已经在生成器中配置，死区发生器也可以不进行配置。

死区模块生成器影子寄存器

延迟寄存器 RED 和 FED 的影子寄存器为 MCPWM_DT*x*_RED_CFG_REG 和 MCPWM_DT*x*_FED_CFG_REG。
MCPWM_GLOBAL_UP_EN 置 1 时，影子寄存器中的值可在某个特定时间写入有效寄存器中。
MCPWM_DT*x*_RED_CFG_REG 的更新方式寄存器为 MCPWM_DT_RED_UPMETHOD；
MCPWM_DT*x*_FED_CFG_REG 的更新方式寄存器为 MCPWM_DT_FED_UPMETHOD。软件也可以触发全局强制更新位
MCPWM_GLOBAL_FORCE_UP，该位将触发模块中的所有寄存器根据影子寄存器进行更新。寄存器描述详见 36.3.2.3。

死区生成器模块的操作要点

图 36.3-21 描述了创建死区模块的开关拓扑。

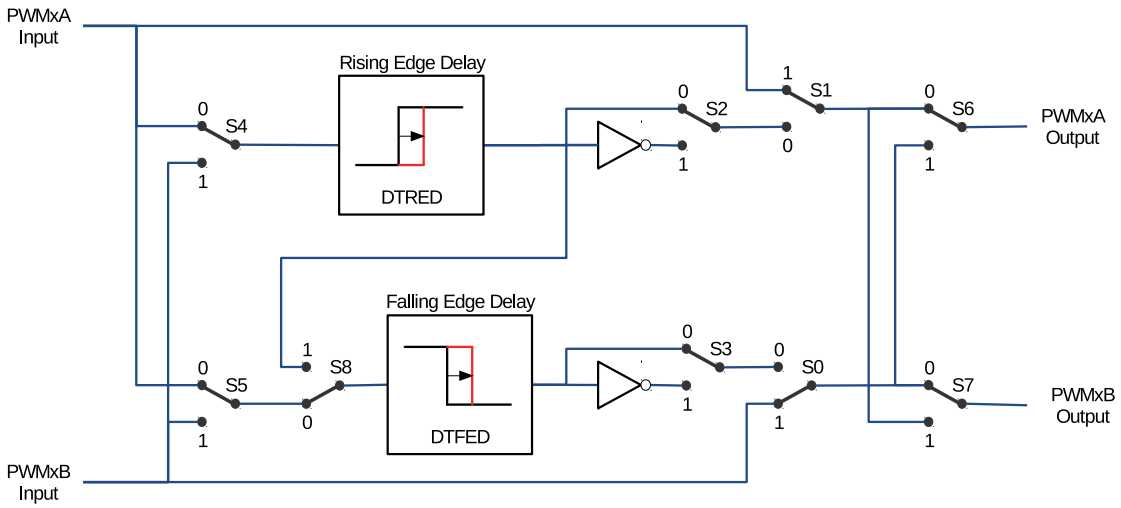


图 36.3-21. 死区模块的开关拓扑

上图中的 S0 - S8 开关由 MCPWM_DT*x*_CFG_REG 寄存器中的字段控制，详细信息见表 36.3-5。

表 36.3-5. 控制死区时间生成器开关的字段

开关	字段
S0	MCPWM_DT*x*_B_OUTBYPASS
S1	MCPWM_DT*x*_A_OUTBYPASS
S2	MCPWM_DT*x*_RED_OUTINVERT
S3	MCPWM_DT*x*_FED_OUTINVERT
S4	MCPWM_DT*x*_RED_INSEL
S5	MCPWM_DT*x*_FED_INSEL
S6	MCPWM_DT*x*_A_OUTSWAP
S7	MCPWM_DT*x*_B_OUTSWAP
S8	MCPWM_DT*x*_DEB_MODE

支持所有开关组合，但不是所有的开关模式都是典型的使用模式。表 36.3-6 列举了一些典型的死区配置。在这些配置中，S4 和 S5 的开关位置将 PWM*x*A 设置为下降沿和上升沿延迟的公共源。表 36.3-6 中的模式可分为以下几类：

- **模式 1：绕过下降沿 (FED) 和上升沿 (RED) 的延迟**
  在该模式下，死区模块被关闭。PWM*x*A 和 PWM*x*B 信号的波形无变化。
- **模式 2-5：经典死区极性设置**
  这些模式为典型极性配置，涵盖工业电源栅极驱动器中的高 / 低电平有效模式。图 36.3-22 至 36.3-25 为典型波形。
- **模式 6 和 7：绕过下降沿 (FED) 或上升沿 (RED) 的延迟**
  此模式下，绕过上升沿延迟 (RED) 或下降沿延迟 (FED)。因此，不使用对应延迟。

表 36.3-6. 死区生成器的典型操作模式

模式	描述	S0	S1	S2	S3
1	PWM*x*A 和 PWM*x*B 波形无变化	1	1	X	X
2	高电平有效互补 (AHC)，参见图 36.3-22	0	0	0	1
3	低电平有效互补 (ALC)，参见图 36.3-23	0	0	1	0
4	高电平有效 (AH)，参见图 36.3-24	0	0	0	0
5	低电平有效 (AL)，参见图 36.3-25	0	0	1	1
6	PWM*x*A 输出 = PWM*x*A 输入（无延迟） PWM*x*B 输出 = PWM*x*A 输入，下降沿延迟	0	1	0 或 1	0 或 1
7	PWM*x*A 输出 = PWM*x*A 输入，上升沿延迟 PWM*x*B 输出 = PWM*x*B 输入（无延迟）	1	0	0 或 1	0 或 1

说明：

以上所有模式中，S4 - S8 的开关位置都置 0。

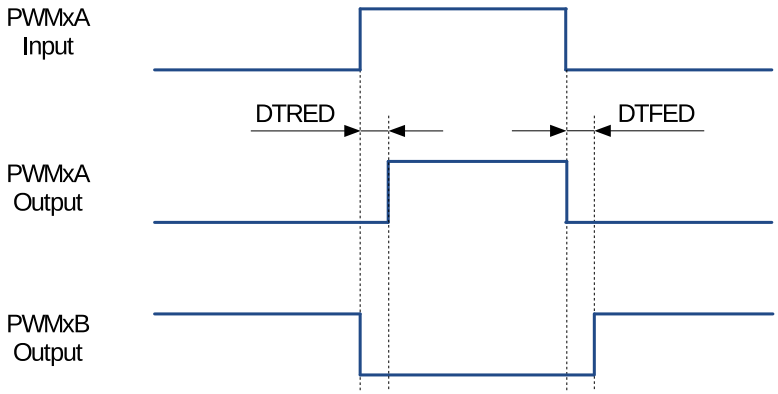


图 36.3-22. 高电平有效互补 (AHC) 死区波形

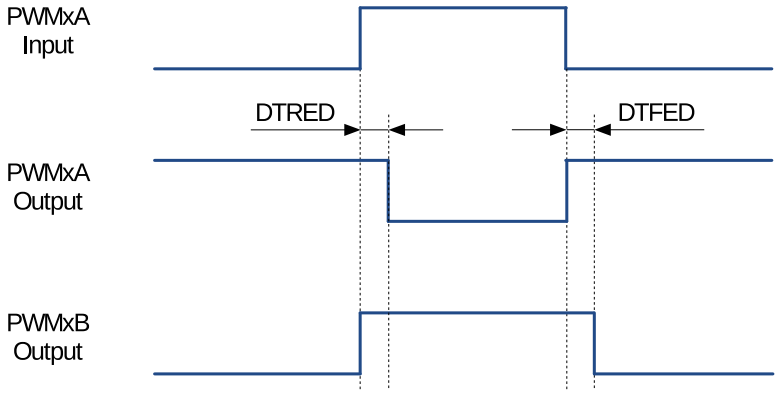


图 36.3-23. 低电平有效互补 (ALC) 死区波形

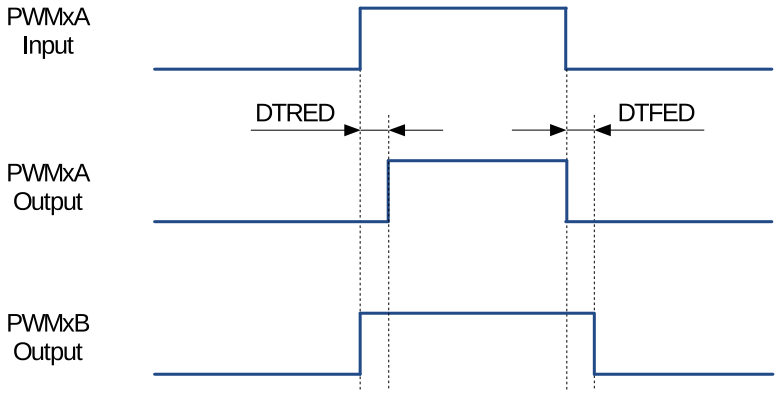


图 36.3-24. 高电平有效 (AH) 死区波形

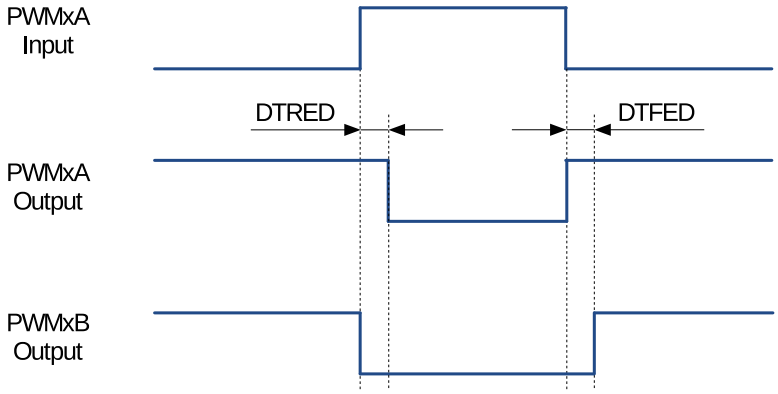


图 36.3-25. 低电平有效 (AL) 死区波形

上升沿延迟 (RED) 和下降沿延迟 (FED) 可分别设置。延迟的值通过 16 位寄存器 MCPWM_DT*x*_*RED* 和 MCPWM_DT*x*_*FED* 配置。寄存器值表示一个信号边沿可以延迟的 DT_clk 时钟周期值。DT_clk 可通过寄存器 MCPWM_DT*x*_CLK_SEL 从 PWM_clk 或 PT_clk 中选择。

通过以下公式计算下降沿延迟 (FED) 和上升沿延迟 (RED) 的值:

$$FED = MCPWM\_DTx\_FED \times T_{DT\_clk}$$

$$RED = MCPWM\_DTx\_RED \times T_{DT\_clk}$$

36.3.3.3 PWM 载波模块

将 PWM 输出耦合到电机驱动器可能需要使用变压器隔离。变压器只提供交流信号，而 PWM 信号的占空比可能在 0% 到 100% 之间变化。PWM 载波模块可以通过使用高频载波对其进行调制，将该信号传递给变压器。

功能概述

此模块的以下关键功能可配置:

- 载波频率
- 第一个脉冲的脉宽
- 第二个以及之后的脉冲的占空比
- 开启/关闭载波

操作要点

PWM 载波时钟 (PC_clk) 来自于 PWM_clk。通过寄存器 MCPWM_CARRIER*x*_CFG_REG 的 MCPWM_CARRIER*x*_PRESCALE 和 MCPWM_CARRIER*x*_DUTY 位配置频率和占空比。一次性脉冲的功能在于提供高能量脉冲以接通电源开关。随后的脉冲用于保持上电的状态。一次性脉冲宽度可通过 MCPWM_CARRIER*x*_OSHTWTH 位进行配置。通过 MCPWM_CARRIER*x*_EN 位来使能/禁用载波模块。

载波示例

图 36.3-26 描述了载波叠加在原始 PWM 脉冲上的示例波形。该图不显示第一个脉冲和占空比控制，相关详细信息将在后两节中介绍。

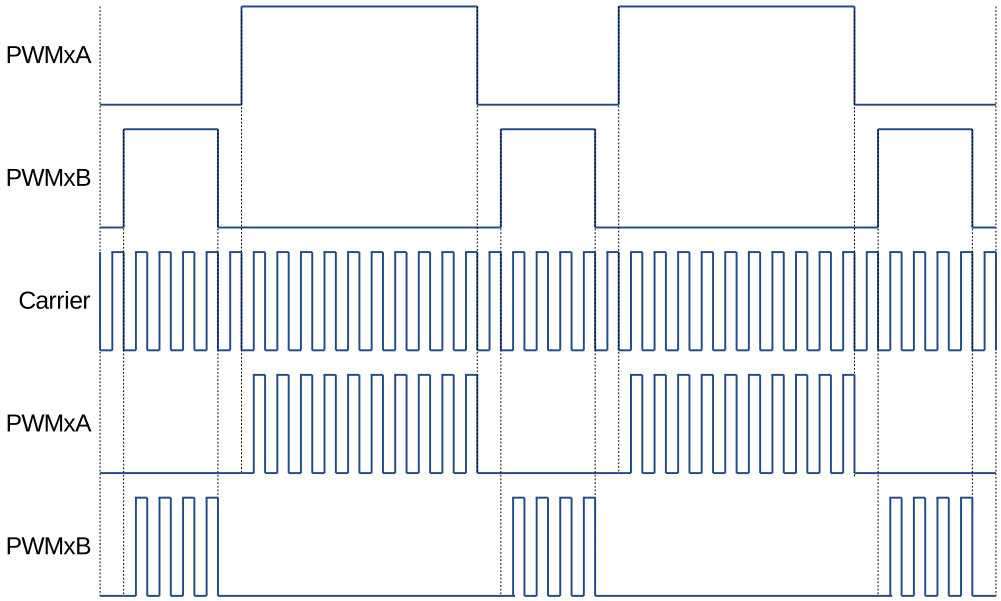


图 36.3-26. PWM 载波操作的波形示例

第一个脉冲

第一个脉冲的宽度可配置，其值有 16 种可能，可通过下公式计算：

$$T_{1stpulse} = T_{PWM\_clk} \times 8 \times (MCPWM\_CARRIERx\_PRESCALE + 1) \times (MCPWM\_CARRIERx\_OSHTWTH + 1)$$

其中：

- T_{PMW_clk} 为 PWM 时钟周期 (PWM_clk)
- $(MCPWM\_CARRIERx\_OSHTWTH + 1)$ 为一次性脉冲宽度值（取值范围：1-16）
- $(MCPWM\_CARRIERx\_PRESCALE + 1)$ PWM 载波时钟 (PC_clk) 预分频值

图 36.3-27 展示了第一个脉冲和之后持续的脉冲。

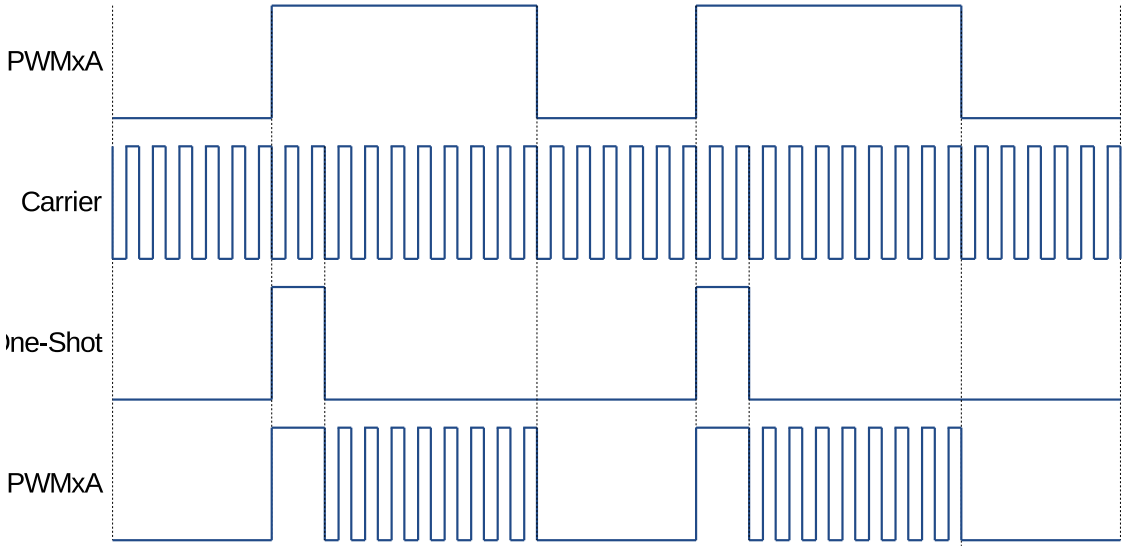


图 36.3-27. 载波模块的第一个脉冲和之后持续的脉冲示例

占空比控制

在发出第一个一次性脉冲之后，根据载波频率调制剩余的 PWM 信号。用户可配置该信号的占空比。在一定情况下，调整占空比可使信号通过隔离变压器后仍然可以开启或关闭电动机驱动器，改变电机旋转速度和方向。

占空比通过 MCPWM_CARRIERx_DUTY 或寄存器 MCPWM_CARRIERx_CFG_REG 的 [7:5] 位设置，其值有 7 种可能性。

占空比的值可通过以下方式计算：

$$Duty = MCPWM\_CARRIERx\_DUTY \div 8$$

图 36.3-28 为所有 7 种占空比设置。

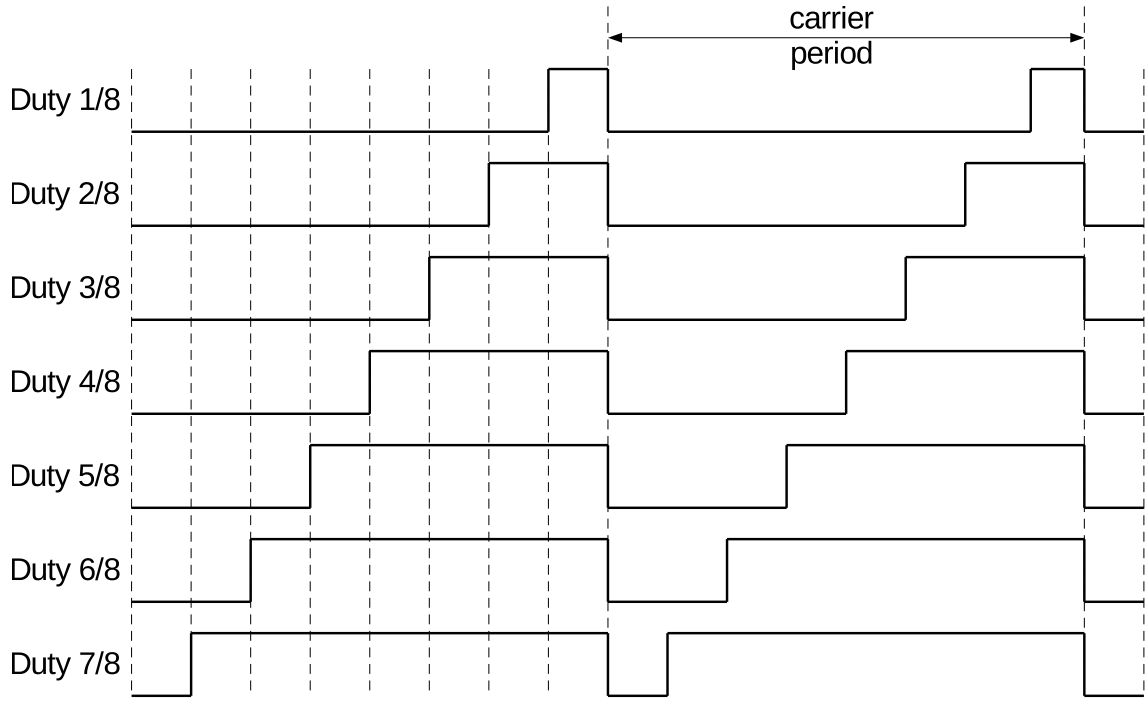


图 36.3-28. PWM 载波模块中持续脉冲的 7 种占空比设置

36.3.3.4 故障处理器模块

每个 MCPWM 外设都连接来自 GPIO 矩阵的 3 个故障信号（FAULT0，FAULT1 和 FAULT2）。这些信号用于指示外部故障状况，并且可由故障检测模块预处理后生成故障事件。故障事件通过执行用户代码，针对特定故障调整 MCPWM 输出。

故障处理模块功能

故障处理模块的主要功能为：

- 在检测到故障时强制 PWM*x*A 和 PWM*x*B 输出信号进入以下状态之一：
  - 高
  - 低
  - 取反
  - 无
- 在检测到过电流/短路时执行一次性跳闸 (OST)。
- 逐周期跳闸 (CBC) 以提供限流操作。
- 每个故障信号单独分配一次性或逐周期操作。
- 每个故障输入都生成中断。
- 支持软件强制跳闸。
- 根据需要开启或关闭模块功能。

操作与配置要点

本节提供故障处理模块的操作要点和配置选项。

来自管脚的故障信号在 GPIO 矩阵中采样和同步。为了保证故障脉冲采样的成功，每个脉冲持续时间必须至少为 2 个 APB 时钟周期。故障检测模块使用 PWM_clk 对故障信号进行采样，因此来自 GPIO 矩阵的故障脉冲持续时间必须至少为 1 个 PWM_clk 周期。换句话说，无论 APB 时钟周期和 PWM_clk 周期的大小关系如何，管脚上的故障信号脉冲的宽度必须至少等于两个 APB 时钟周期与一个 PWM_clk 周期的和。

故障处理器模块可以使用故障信号 FAULT0 至 FAULT2 中的高电平或低电平来生成故障事件 fault_event0 至 fault_event2。每个故障事件可以单独配置为进行 CBC 操作，OST 操作或无操作。

- **逐周期 (CBC) 操作：**
  当 CBC 操作被触发，PWM*x*A 和 PWM*x*B 的状态立即根据字段 MCPWM_FH*x*_A_CBC_U/D 和 MCPWM_FH*x*_B_CBC_U/D 的设置改变。PWM 定时器递增或递减计数时，可指定不同的操作。不同的故障事件可触发不同的逐周期操作中断。通过状态字段 MCPWM_FH*x*_CBC_ON 开启或关闭 CBC 操作。在没有故障事件时，将在指定时间点，即发生 D/UTEP 或 D/UTEZ 事件时清除 PWM*x*A/B 上的 CBC 操作。字段 MCPWM_FH*x*_CBCPULSE 控制决定 PWM*x*A 和 PWM*x*B 恢复正常的事件。因此，在此模式下，CBC 操作在每个 PWM 循环后清除或刷新。
- **一次性 (OST) 操作：**
  当 OST 操作被触发时，PWM*x*A 和 PWM*x*B 的状态立即根据字段 MCPWM_FH*x*_A_OST_U/D 和 MCPWM_FH*x*_B_OST_U/D 改变。PWM 定时器递增或递减计数时，可配置不同的操作。不同的故障事件可触发不同的 OST 操作中断。通过状态字段 MCPWM_FH*x*_OST_ON 开启或关闭 OST 操作。PWM*x*A/B 上的 OST 操作将在没有故障事件时不能自动清除。一次性操作须通过将 MCPWM_FH*x*_CLR_OST 位置 1 来清除。

36.3.4 捕获模块

36.3.4.1 介绍

捕获模块包含 3 个完整的捕获通道。通道输入信号 CAP0，CAP1 和 CAP2 来自于 GPIO 矩阵。由于 GPIO 矩阵的灵活性，CAP0，CAP1 和 CAP2 可以通过任一管脚输入配置。多个捕获通道可以来自同一管脚输入，而每个通道的预分频可以分别设置。此外，每个捕获通道还可以来自不同的管脚输入。因此，可以通过后台硬件用多种方式处理捕获信号，而不直接由 CPU 处理。

捕捉模块都有以下独立资源：

- 一个 32 位定时器（计数器），可与 PWM 定时器、另一个模块或软件同步。
- 三个捕获通道，每个通道配有一个 32 位时间戳和一个捕获预分频器。
- 任何捕获通道的边沿极性（上升/下降沿）可独立选择。
- 输入捕获信号预分频（分频取值范围：1 ~ 256）。
- 三个捕获事件都有中断功能。

36.3.4.2 捕获定时器

捕获定时器是一个 32 位计数器，使能时不断递增计数。将 MCPWM_CAP_TIMER_EN 置 1 使能捕获寄存器。其操作时钟源为 APB_CLK。配置 MCPWM_CAP_SYNCI_EN 后，发生同步事件时，存储在

MCPWM_CAP_TIMER_PHASE_REG 寄存器中的相位将被加载至计数器中。同步事件可来自 PWM 定时器同步输出或 PWM 模块同步输入，通过配置 MCPWM_CAP_SYNCI_SEL 选择。同步事件也可通过将 MCPWM_CAP_SYNC_SW 置 1 生成。该捕获定时器为所有 3 个捕获通道提供定时参考。

36.3.4.3 捕获通道

必要时，到达捕获通道的捕获信号可先被反相，然后预分频。每个捕获通道都有一个预分频寄存器 MCPWM_CAP*x*_PRESCALE。最后，预处理后的捕获信号的指定边沿将触发捕获事件。将 MCPWM_CAP*x*_EN 置 1 可使能捕获通道。捕获事件将在 MCPWM_CAP*x*_MODE 配置的时间发生。在捕获事件发生时，捕获定时器的值存储在时间戳寄存器 MCPWM_CAP_CH*x*_REG 中。捕获事件中的不同捕获通道可生成不同的中断。触发捕获事件的边沿储存在寄存器 MCPWM_CAP*x*_EDGE 中。捕获事件可通过将 MCPWM_CAP*x*_SW 置 1 由软件强制发生。

36.4 寄存器列表

本小节的所有地址均为相对于电机控制器 0 和电机控制器 1 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
预分频器配置			
MCPWM_CLK_CFG_REG	配置预分频器	0x0000	R/W
PWM 定时器 0 配置与状态			
MCPWM_TIMER0_CFG0_REG	定时器周期与更新方法	0x0004	R/W
MCPWM_TIMER0_CFG1_REG	工作模式与开始/停止控制	0x0008	R/W
MCPWM_TIMER0_SYNC_REG	PWM 定时器 0 同步功能配置寄存器	0x000C	R/W
MCPWM_TIMER0_STATUS_REG	PWM 定时器 0 状态	0x0010	RO
PWM 定时器 1 配置与状态			
MCPWM_TIMER1_CFG0_REG	定时器更新方式与周期	0x0014	R/W
MCPWM_TIMER1_CFG1_REG	工作模式与开始/停止控制	0x0018	varies
MCPWM_TIMER1_SYNC_REG	同步设置	0x001C	R/W
MCPWM_TIMER1_STATUS_REG	PWM 定时器 1 状态	0x0020	RO
PWM 定时器 2 配置与状态			
MCPWM_TIMER2_CFG0_REG	定时器方式与周期	0x0024	R/W
MCPWM_TIMER2_CFG1_REG	工作模式与开始/停止控制	0x0028	varies
MCPWM_TIMER2_SYNC_REG	同步设置	0x002C	R/W
MCPWM_TIMER2_STATUS_REG	PWM 定时器 2 状态	0x0030	RO
PWM 定时器常见配置			
MCPWM_TIMER_SYNCI_CFG_REG	定时器同步输入选择	0x0034	R/W
MCPWM_OPERATOR_TIMERSEL_REG	为 PWM 操作器选择特定的计时器	0x0038	R/W
PWM 操作器 0 配置与状态			
MCPWM_GEN0_STMP_CFG_REG	时间戳寄存器 A 和 B 的传输状态和更新方式	0x003C	varies
MCPWM_GEN0_TSTMP_A_REG	PWM 操作器 0 时间戳寄存器 A 的影子寄存器	0x0040	R/W
MCPWM_GEN0_TSTMP_B_REG	PWM 操作器 0 时间戳寄存器 B 的影子寄存器	0x0044	R/W
MCPWM_GEN0_CFG0_REG	故障事件 T0 和 T1 处理	0x0048	R/W
MCPWM_GEN0_FORCE_REG	软件强制 PWM0A 和 PWM0B 输出	0x004C	R/W
MCPWM_GEN0_A_REG	PWM0A 输出上事件触发的操作	0x0050	R/W
MCPWM_GEN0_B_REG	PWM0B 输出上事件触发的操作	0x0054	R/W
MCPWM_DT0_CFG_REG	死区与类型的选择与配置	0x0058	R/W
MCPWM_DT0_FED_CFG_REG	FED 的影子寄存器	0x005C	R/W
MCPWM_DT0_RED_CFG_REG	RED 的影子寄存器	0x0060	R/W
MCPWM_CARRIER0_CFG_REG	载波使能与配置	0x0064	R/W
MCPWM_FH0_CFG0_REG	故障事件中 PWM0A 和 PWM0B 上的操作	0x0068	R/W
MCPWM_FH0_CFG1_REG	故障处理的软件触发	0x006C	R/W
MCPWM_FH0_STATUS_REG	故障事件状态	0x0070	RO
PWM 操作器 1 配置与状态			
MCPWM_GEN1_STMP_CFG_REG	时间戳寄存器 A 和 B 的传输状态和更新方式	0x0074	varies

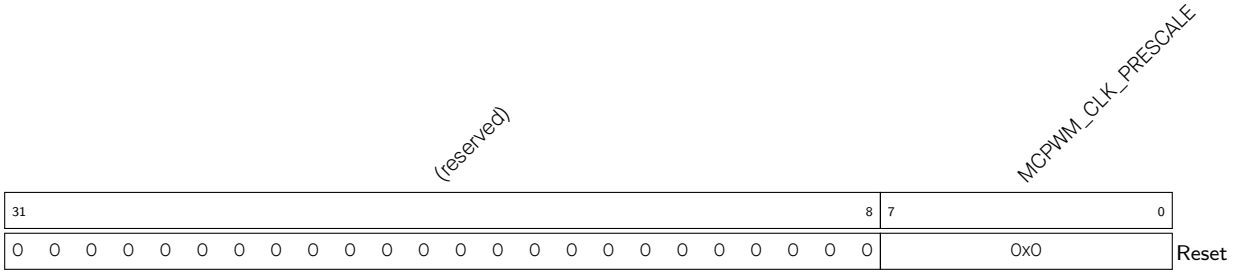
名称	描述	地址	访问
MCPWM_GEN1_TSTMP_A_REG	PWM 操作器 1 时间戳寄存器 A 的影子寄存器	0x0078	R/W
MCPWM_GEN1_TSTMP_B_REG	PWM 操作器 1 时间戳寄存器 B 的影子寄存器	0x007C	R/W
MCPWM_GEN1_CFG0_REG	故障事件 T0 和 T1 处理	0x0080	R/W
MCPWM_GEN1_FORCE_REG	软件强制 PWM1A 和 PWM1B 输出	0x0084	R/W
MCPWM_GEN1_A_REG	PWM1A 输出上的事件触发的操作	0x0088	R/W
MCPWM_GEN1_B_REG	PWM1B 输出上的事件触发的操作	0x008C	R/W
MCPWM_DT1_CFG_REG	死区类型的选择与配置	0x0090	R/W
MCPWM_DT1_FED_CFG_REG	FED 的影子寄存器	0x0094	R/W
MCPWM_DT1_RED_CFG_REG	RED 的影子寄存器	0x0098	R/W
MCPWM_CARRIER1_CFG_REG	使能与配置载波	0x009C	R/W
MCPWM_FH1_CFG0_REG	故障事件中 PWM1A 和 PWM1B 输出上的操作	0x00A0	R/W
MCPWM_FH1_CFG1_REG	故障处理的软件触发	0x00A4	R/W
MCPWM_FH1_STATUS_REG	故障事件状态	0x00A8	RO
PWM 操作器 2 的配置与状态			
MCPWM_GEN2_STMP_CFG_REG	时间戳寄存器 A 和 B 的传输状态和更新方式	0x00AC	varies
MCPWM_GEN2_TSTMP_A_REG	PWM 操作器 2 时间戳寄存器 A 的影子寄存器	0x00B0	R/W
MCPWM_GEN2_TSTMP_B_REG	PWM 操作器 2 时间戳寄存器 B 的影子寄存器	0x00B4	R/W
MCPWM_GEN2_CFG0_REG	故障事件 T0 和 T1 处理	0x00B8	R/W
MCPWM_GEN2_FORCE_REG	软件强制 PWM2A 和 PWM2B 输出	0x00BC	R/W
MCPWM_GEN2_A_REG	PWM2A 输出上的事件触发的操作	0x00C0	R/W
MCPWM_GEN2_B_REG	PWM2B 输出上的事件触发的操作	0x00C4	R/W
MCPWM_DT2_CFG_REG	死区类型的选择与配置	0x00C8	R/W
MCPWM_DT2_FED_CFG_REG	FED 影子寄存器	0x00CC	R/W
MCPWM_DT2_RED_CFG_REG	RED 影子寄存器	0x00D0	R/W
MCPWM_CARRIER2_CFG_REG	使能与配置载波	0x00D4	R/W
MCPWM_FH2_CFG0_REG	故障事件中 PWM2A 和 PWM2B 输出上的操作	0x00D8	R/W
MCPWM_FH2_CFG1_REG	故障处理的软件触发	0x00DC	R/W
MCPWM_FH2_STATUS_REG	故障事件状态	0x00E0	RO
故障检测与配置			
MCPWM_FAULT_DETECT_REG	故障检测与配置	0x00E4	varies
捕获配置与状态			
MCPWM_CAP_TIMER_CFG_REG	配置捕获定时器	0x00E8	varies
MCPWM_CAP_TIMER_PHASE_REG	捕获定时器同步相位	0x00EC	R/W
MCPWM_CAP_CH0_CFG_REG	捕获通道 0 的配置与使能	0x00F0	varies
MCPWM_CAP_CH1_CFG_REG	捕获通道 1 的配置与使能	0x00F4	varies
MCPWM_CAP_CH2_CFG_REG	捕获通道 2 的配置与使能	0x00F8	varies
MCPWM_CAP_CH0_REG	捕获通道 0 值的状态	0x00FC	RO
MCPWM_CAP_CH1_REG	捕获通道 1 值的状态	0x0100	RO
MCPWM_CAP_CH2_REG	捕获通道 2 值的状态	0x0104	RO
MCPWM_CAP_STATUS_REG	上一次捕获触发器的边沿	0x0108	RO
使能有效寄存器的更新			
MCPWM_UPDATE_CFG_REG	使能更新	0x010C	R/W
管理中断			

名称	描述	地址	访问
MCPWM_INT_ENA_REG	中断使能位	0x0110	R/W
MCPWM_INT_RAW_REG	原始中断状态	0x0114	R/WTC/SS
MCPWM_INT_ST_REG	屏蔽中断状态	0x0118	RO
MCPWM_INT_CLR_REG	中断清除位	0x011C	WT
MCPWM APB 配置			
MCPWM_CLK_REG	MCPWM APB 配置	0x0120	R/W
版本寄存器			
MCPWM_VERSION_REG	版本控制寄存器	0x0124	R/W

36.5 寄存器

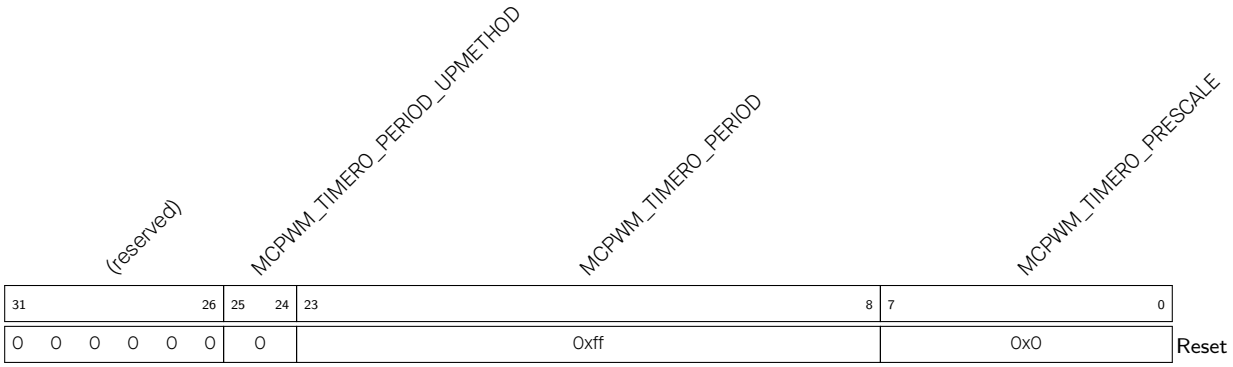
本小节的所有地址均为相对于电机控制器 0 和电机控制器 1 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 36.1. MCPWM_CLK_CFG_REG (0x0000)



MCPWM_CLK_PRESCALE PWM_clk 的周期 = 6.25ns * (PWM_CLK_PRESCALE + 1)。(R/W)

Register 36.2. MCPWM_TIMER0_CFG0_REG (0x0004)

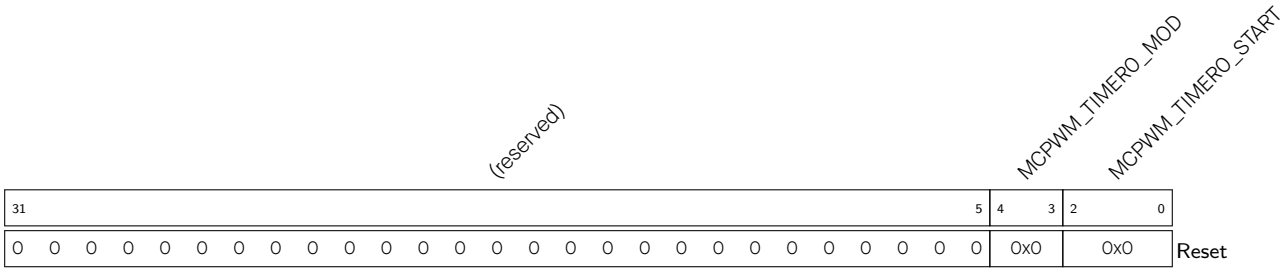


MCPWM_TIMER0_PRESCALE PT0_clk 的周期 = PWM_clk 的周期 * (PWM_TIMER0_PRESCALE + 1)。(R/W)

MCPWM_TIMER0_PERIOD 计时器 0 的影子周期寄存器。(R/W)

MCPWM_TIMER0_PERIOD_UPMETHOD PWM 定时器 0 周期有效寄存器的更新方式。0：立即更新；1：发生 TEZ 事件时更新；2：同步时更新；3：发生 TEZ 事件或同步时更新。本文档中，TEZ 指定时器为 0 时的事件。(R/W)

Register 36.3. MCPWM_TIMER0_CFG1_REG (0x0008)



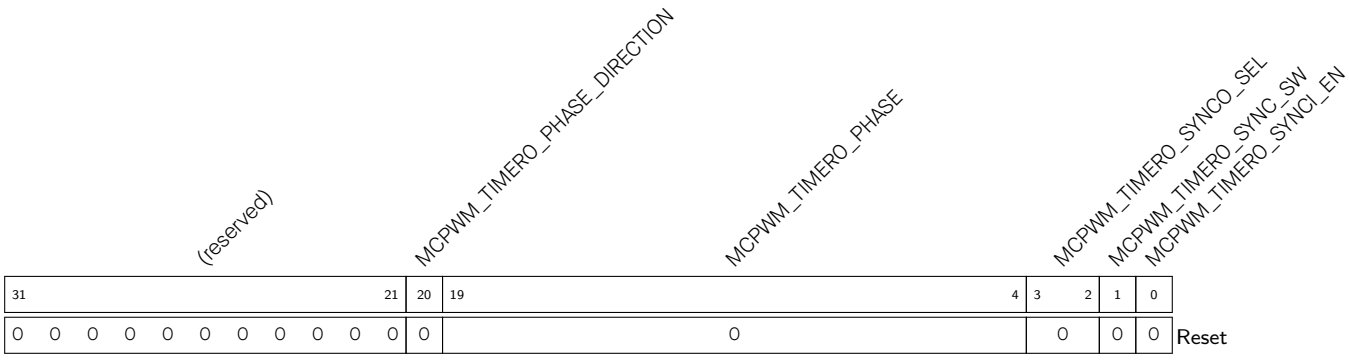
MCPWM_TIMER0_START 控制 PWM 定时器的开启与关闭。(R/W/SC)

- 0：如果开启，在 TEZ 事件发生时停止；
- 1：如果开启，在 TEP 事件发生时停止；
- 2：开启；
- 3：开启，并在下一个 TEZ 事件发生时停止；
- 4：开启，并在下一个 TEP 事件发生时停止。

本文档中，TEP 指定时器为周期值时发生的事件。

MCPWM_TIMER0_MOD PWM 定时器 0 工作模式。0：暂停；1：递增模式；2：递减模式；3：递增递减循环模式。(R/W)

Register 36.4. MCPWM_TIMER0_SYNC_REG (0x000C)



MCPWM_TIMER0_SYNCI_EN 置 1 时，使能在同步输入事件发生时的定时器相位重载。(R/W)

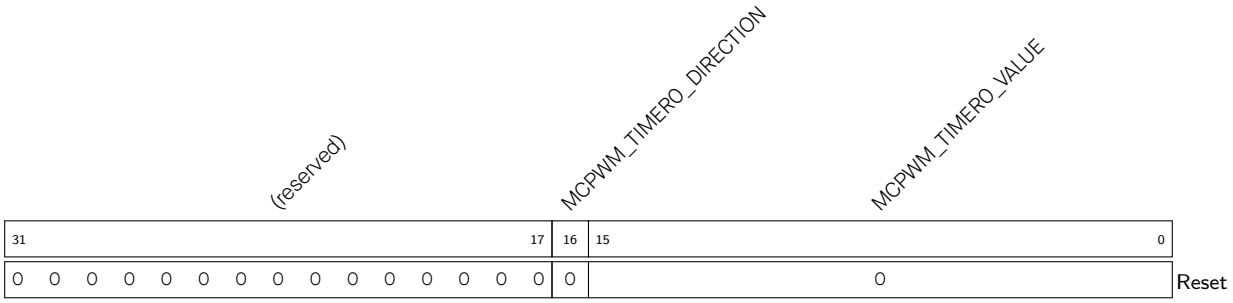
MCPWM_TIMER0_SYNC_SW 此位取反，触发软件同步。(R/W)

MCPWM_TIMER0_SYNCO_SEL 选择 PWM 定时器 0 的同步输出来源。0：同步；1：TEZ；2：TEP。取反 MCPWM_TIMER0_SYNC_SW 位时始终生成同步输出。(R/W)

MCPWM_TIMER0_PHASE 同步事件中定时器重载的相位。(R/W)

MCPWM_TIMER0_PHASE_DIRECTION 当定时器 0 为递增-递减循环模式时，配置该定时器方向。0：递增；1：递减。(R/W)

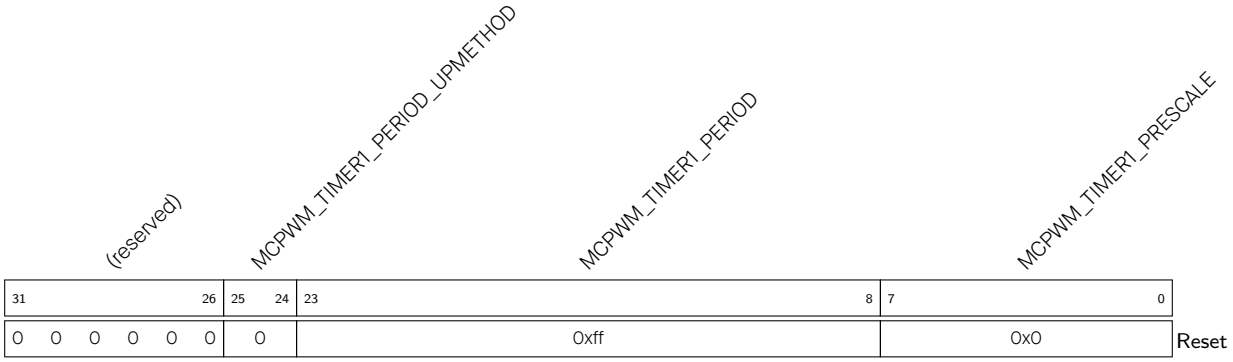
Register 36.5. MCPWM_TIMER0_STATUS_REG (0x0010)



MCPWM_TIMER0_VALUE 当前 PWM 定时器 0 计数器的值。(RO)

MCPWM_TIMER0_DIRECTION 当前 PWM 定时器 0 的计数器模式。0：递增模式；1：递减模式。(RO)

Register 36.6. MCPWM_TIMER1_CFG0_REG (0x0014)

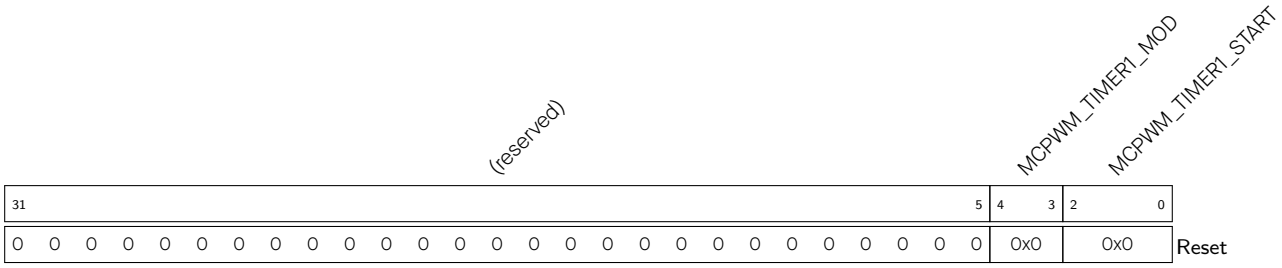


MCPWM_TIMER1_PRESCALE PT0_clk 周期 = PWM_clk 周期 * (PWM_timer1_PRESCALE + 1)。(R/W)

MCPWM_TIMER1_PERIOD 定时器 1 的影子周期寄存器。(R/W)

MCPWM_TIMER1_PERIOD_UPMETHOD PWM 定时器 1 周期有效寄存器的更新方式。0：立即更新；1：发生 TEZ 事件时更新；2：同步时更新；3：发生 TEZ 事件或同步时更新。本文档中，TEZ 指定时器为 0 时的事件。(R/W)

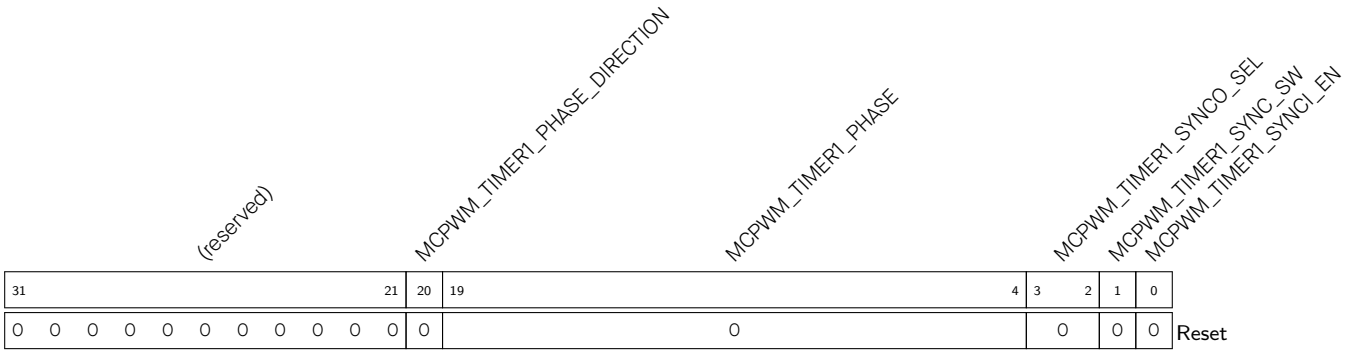
Register 36.7. MCPWM_TIMER1_CFG1_REG (0x0018)



MCPWM_TIMER1_START PWM 控制定时器 1 的开启与停止。0：如果开启，在发生 TEZ 事件时停止；1：如果开启，在发生 TEP 事件时停止；2：开启；3：开启并在下一次 TEZ 事件中停止；4：开启并在下一次 TEP 事件中停止。本文档中，TEP 指定时器为周期数时的事件。(R/W/SC)

MCPWM_TIMER1_MOD PWM 计时器 1 的工作模式。0：暂停；1：递增模式；2：递减模式；3：递增-递减循环模式。(R/W)

Register 36.8. MCPWM_TIMER1_SYNC_REG (0x001C)



MCPWM_TIMER1_SYNCI_EN 置 1 时，使能在同步输入事件时的定时器相位重载。(R/W)

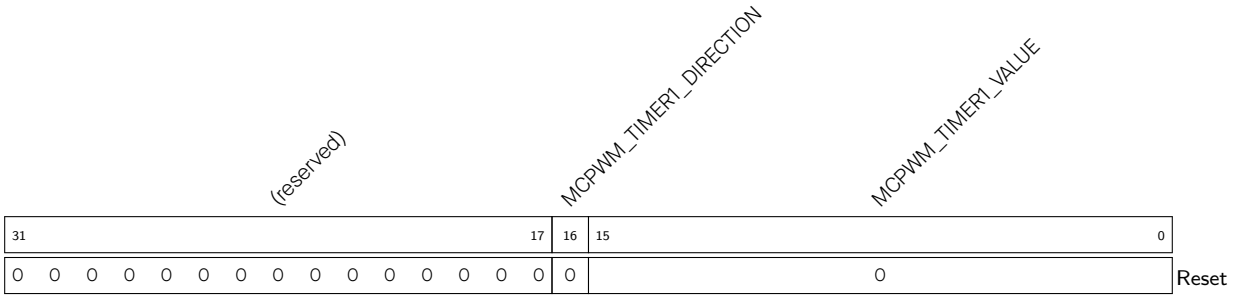
MCPWM_TIMER1_SYNC_SW 此位取反，触发软件同步事件。(R/W)

MCPWM_TIMER1_SYNCO_SEL 选择 PWM 定时器 1 的同步输出来源。0：同步；1：TEZ；2：TEP。取反 reg_timer1_sw 位时始终生成同步输出。(R/W)

MCPWM_TIMER1_PHASE 同步时间中计时器重载的相位。(R/W)

MCPWM_TIMER1_PHASE_DIRECTION 当定时器 1 为递增-递减循环模式时，配置该定时器方向。0：递增；1：递减。(R/W)

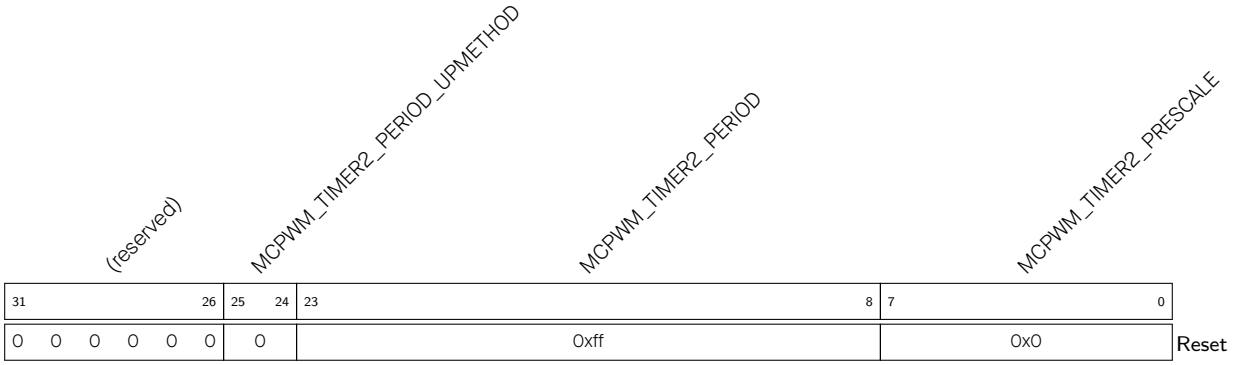
Register 36.9. MCPWM_TIMER1_STATUS_REG (0x0020)



MCPWM_TIMER1_VALUE 当前 PWM 计时器 1 的计数器值。(RO)

MCPWM_TIMER1_DIRECTION 当前 PWM 计时器 1 的计数器模式。0：递增；1：递减。(RO)

Register 36.10. MCPWM_TIMER2_CFG0_REG (0x0024)

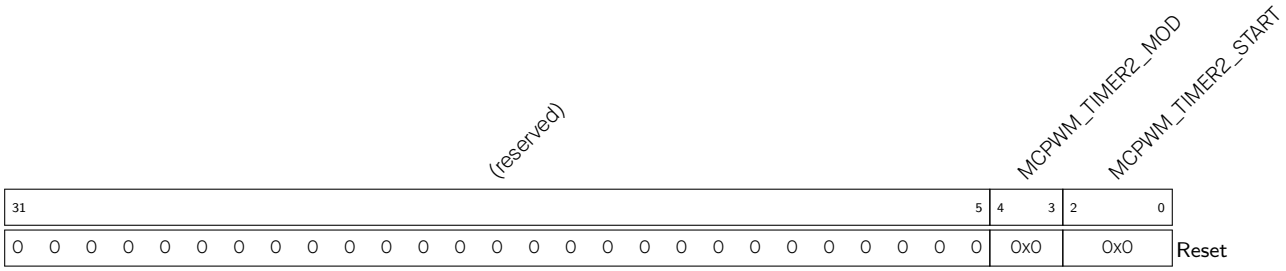


MCPWM_TIMER2_PRESCALE PTO_clk 周期 = PWM_clk 周期 * (PWM_timer2_PRESCALE + 1)。(R/W)

MCPWM_TIMER2_PERIOD PWM 定时器 2 的影子周期寄存器。(R/W)

MCPWM_TIMER2_PERIOD_UPMETHOD PWM 定时器 2 周期有效寄存器的更新方式。0：立即更新；1：发生 TEZ 事件时更新；2：同步时更新；3：发生 TEZ 事件或同步时更新。本文档中，TEZ 指定时器为 0 时的事件。(R/W)

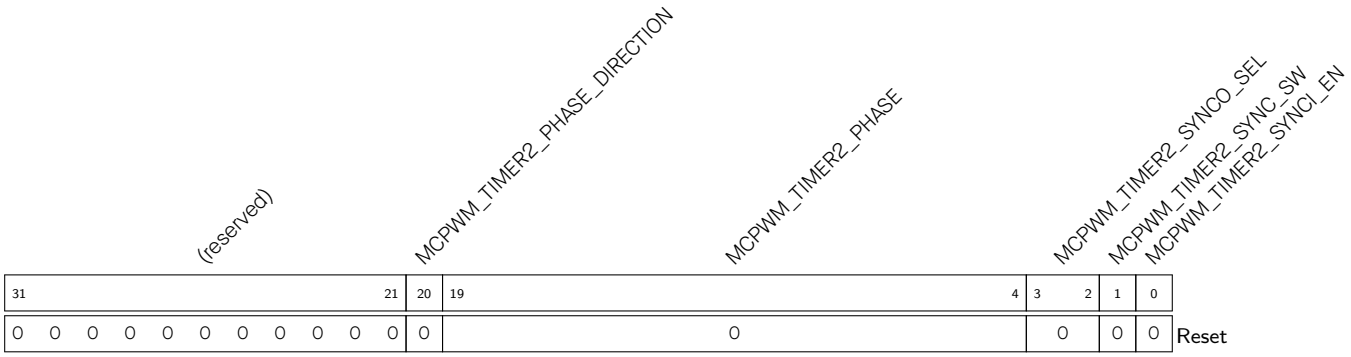
Register 36.11. MCPWM_TIMER2_CFG1_REG (0x0028)



MCPWM_TIMER2_START PWM 控制定时器 2 的开启与停止。0：如果开启，在发生 TEZ 事件时停止；1：如果开启，在发生 TEP 事件时停止；2：开启；3：开启并在下一次 TEZ 事件中停止；4：开启并在下一次 TEP 事件中停止。本文档中，TEP 指定时器为周期数时的事件。(R/W/SC)

MCPWM_TIMER2_MOD PWM 计时器 1 的工作模式。0：暂停；1：递增模式；2：递减模式；3：递增-递减循环模式。(R/W)

Register 36.12. MCPWM_TIMER2_SYNC_REG (0x002C)



MCPWM_TIMER2_SYNCI_EN 置 1 时使能在同步输入事件时的定时器相位重载。(R/W)

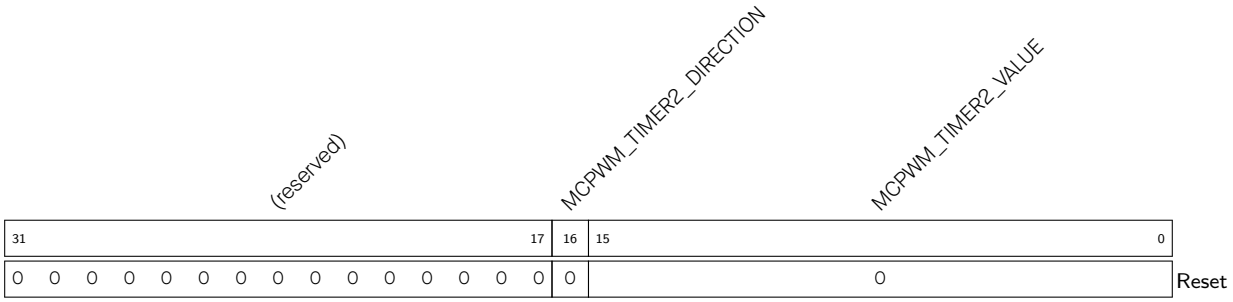
MCPWM_TIMER2_SYNC_SW 此位取反，触发软件同步事件。(R/W)

MCPWM_TIMER2_SYNCO_SEL 选择 PWM 定时器 2 的同步输出来源。0：同步；1：TEZ；2：TEP。取反 reg_timer2_sw 位时始终生成同步输出。(R/W)

MCPWM_TIMER2_PHASE 同步事件中定时器重载相位。(R/W)

MCPWM_TIMER2_PHASE_DIRECTION 当定时器 2 为递增-递减循环模式时，配置该定时器方向。0：递增；1：递减。(R/W)

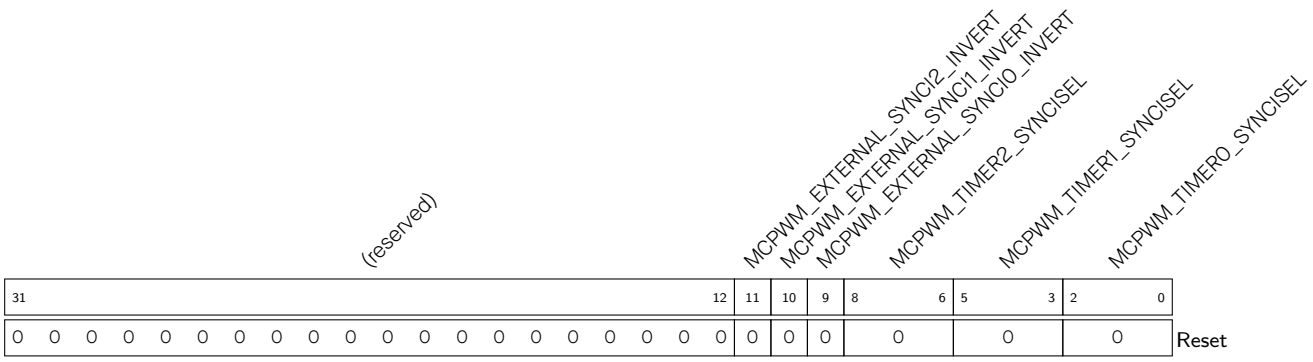
Register 36.13. MCPWM_TIMER2_STATUS_REG (0x0030)



MCPWM_TIMER2_VALUE 当前 PWM 定时器 2 计数器的值。(RO)

MCPWM_TIMER2_DIRECTION 当前 PWM 计时器 2 的计数器模式。0：递增；1：递减。(RO)

Register 36.14. MCPWM_TIMER_SYNCI_CFG_REG (0x0034)



MCPWM_TIMER0_SYNCISEL 选择 PWM 定时器 0 的同步输入来源。(R/W)

- 1：PWM 定时器 0 同步输出；
- 2：PWM 定时器 1 同步输出；
- 3：PWM 定时器 2 同步输出；
- 4：来自 GPIO 矩阵的 SYNC0；
- 5：来自 GPIO 矩阵的 SYNC1；
- 6：来自 GPIO 矩阵的 SYNC2；
- 其他值：未选择任何同步输入。

MCPWM_TIMER1_SYNCISEL 选择 PWM 定时器 1 的同步输入来源。(R/W)

- 1：PWM 定时器 0 同步输出；
- 2：PWM 定时器 1 同步输出；
- 3：PWM 定时器 2 同步输出；
- 4：来自 GPIO 矩阵的 SYNC0；
- 5：来自 GPIO 矩阵的 SYNC1；
- 6：来自 GPIO 矩阵的 SYNC2；
- 其他值：未选择任何同步输入。

见下页

Register 36.14. MCPWM_TIMER_SYNCI_CFG_REG (0x0034)

接上页

MCPWM_TIMER2_SYNCISEL 选择 PWM 定时器 2 的同步输入来源。(R/W)

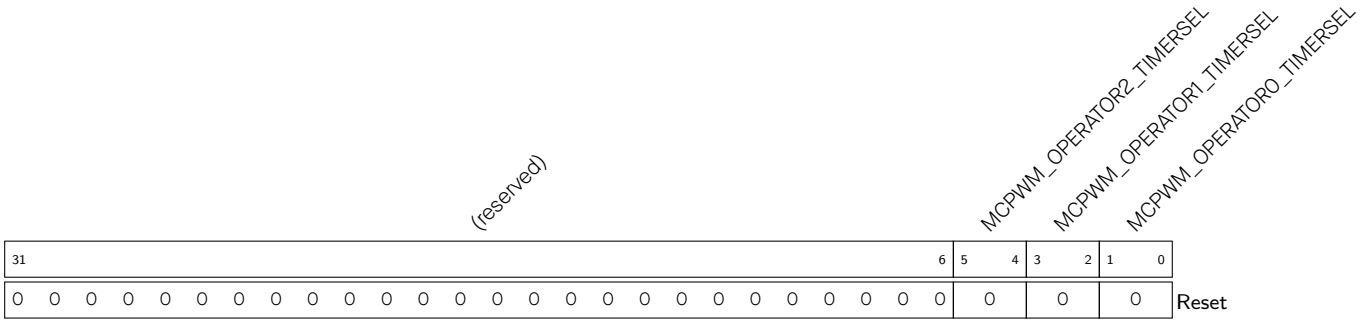
- 1：PWM 定时器 0 同步输出；
- 2：PWM 定时器 1 同步输出；
- 3：PWM 定时器 2 同步输出；
- 4：来自 GPIO 矩阵的 SYNC0；
- 5：来自 GPIO 矩阵的 SYNC1；
- 6：来自 GPIO 矩阵的 SYNC2；
- 其他值：未选择任何同步输入。

MCPWM_EXTERNAL_SYNCI0_INVERT 将来自 GPIO 矩阵的 SYNC0 反相。(R/W)

MCPWM_EXTERNAL_SYNCI1_INVERT 将来自 GPIO 矩阵的 SYNC1 反相。(R/W)

MCPWM_EXTERNAL_SYNCI2_INVERT 将来自 GPIO 矩阵的 SYNC2 反相。(R/W)

Register 36.15. MCPWM_OPERATOR_TIMERSEL_REG (0x0038)

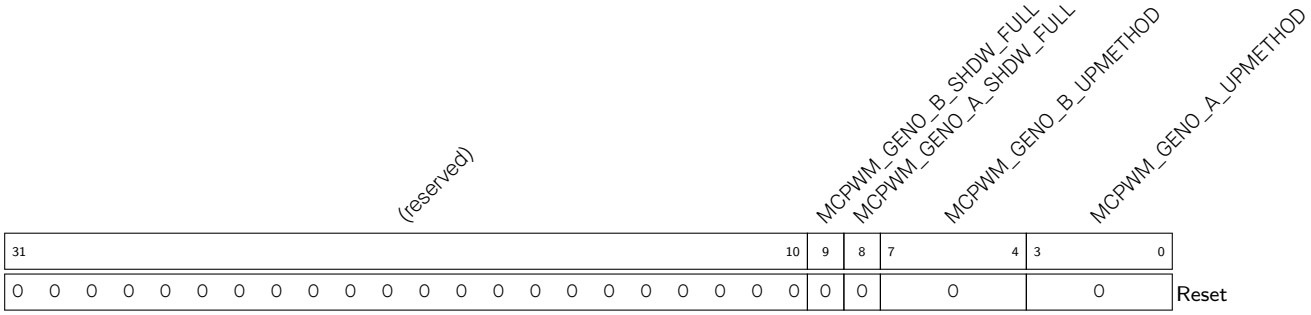


MCPWM_OPERATOR0_TIMERSEL 选择 PWM 操作器 0 的定时参考来源。0：定时器 0；1：定时器 1；2：定时器 2。(R/W)

MCPWM_OPERATOR1_TIMERSEL 选择 PWM 操作器 1 的定时参考来源。0：定时器 0；1：定时器 1；2：定时器 2。(R/W)

MCPWM_OPERATOR2_TIMERSEL 选择 PWM 操作器 2 的定时参考来源。0：定时器 0；1：定时器 1；2：定时器 2。(R/W)

Register 36.16. MCPWM_GEN0_STMP_CFG_REG (0x003C)



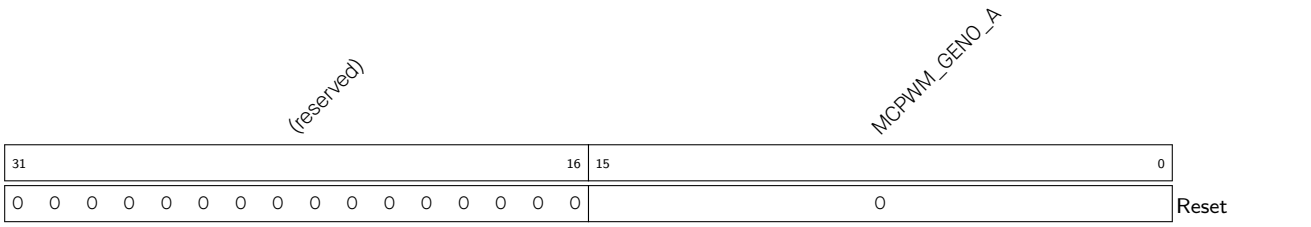
MCPWM_GEN0_A_UPMETHOD PWM 生成器 0 时间戳寄存器 A 的有效寄存器的更新方式。所有位值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_GEN0_B_UPMETHOD PWM 生成器 0 时间戳寄存器 B 有效寄存器的更新方式。所有位值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_GEN0_A_SHDW_FULL 硬件置 1 或复位。置 1 时，PWM 生成器 0 时间戳寄存器 A 的影子寄存器被写入，写入的值等待传输给有效寄存器 A。清零时，有效寄存器 A 中写入其影子寄存器最新的值。(R/WTC/SC)

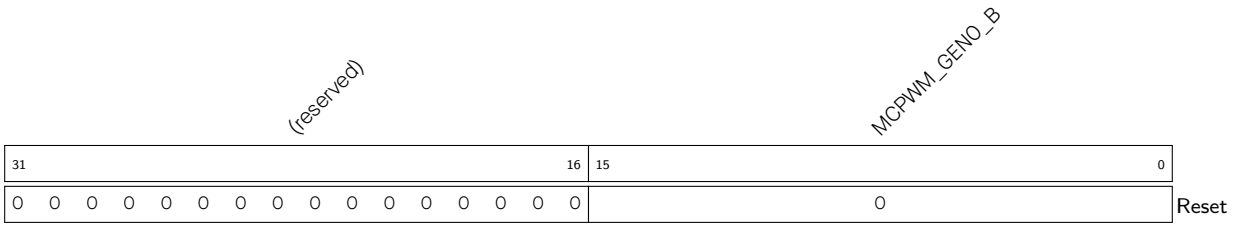
MCPWM_GEN0_B_SHDW_FULL 由硬件置 1 和清零。置 1 时，PWM 生成器 0 时间戳寄存器 B 的影子寄存器被写入，写入的值将传输给有效寄存器 B。清零时，有效寄存器 B 中写入其影子寄存器最新的值。(R/WTC/SC)

Register 36.17. MCPWM_GEN0_TSTMP_A_REG (0x0040)



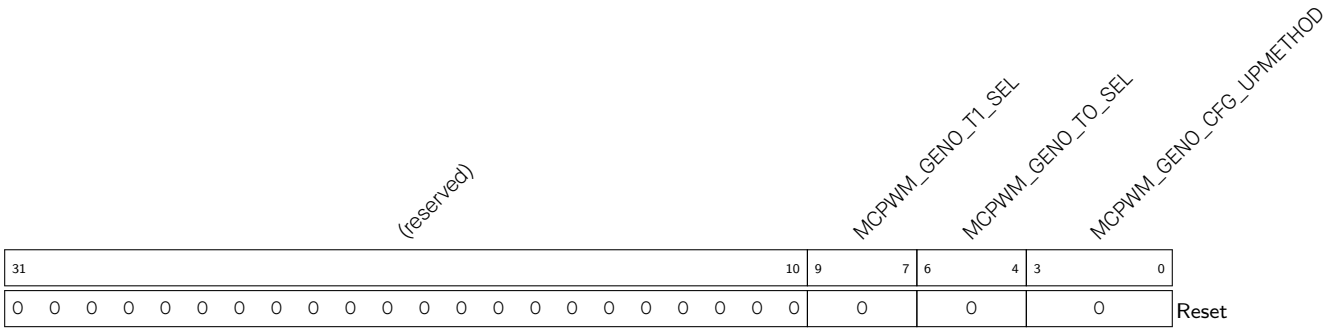
MCPWM_GEN0_A PWM 生成器 0 时间戳寄存器 A 的影子寄存器。(R/W)

Register 36.18. MCPWM_GEN0_TSTMP_B_REG (0x0044)



MCPWM_GEN0_B PWM 生成器 0 时间戳寄存器 B 的影子寄存器。(R/W)

Register 36.19. MCPWM_GEN0_CFG0_REG (0x0048)

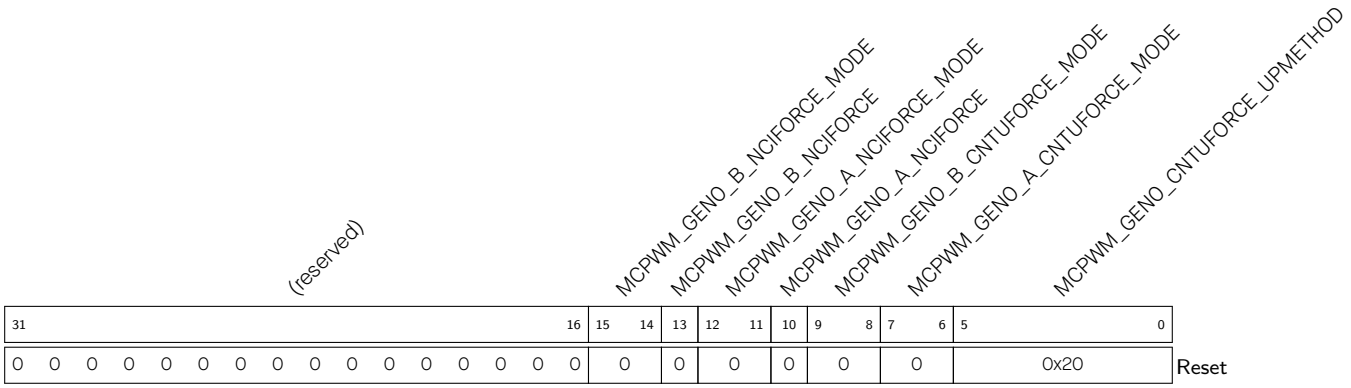


MCPWM_GEN0_CFG_UPMETHOD PWM 生成器 0 有效配置寄存器的更新方式。所有 bit 值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_GEN0_T0_SEL 选择 PWM 生成器 0 event_t0 的信号源，立即生效。0：fault_event0；1：fault_event1；2：fault_event2；3：sync_taken；4：无。(R/W)

MCPWM_GEN0_T1_SEL 选择 PWM 生成器 0 event_t1 的信号源，立即生效。0：fault_event0；1：fault_event1；2：fault_event2；3：sync_taken；4：无。(R/W)

Register 36.20. MCPWM_GEN0_FORCE_REG (0x004C)



MCPWM_GEN0_CNTUFORCE_UPMETHOD 生成器 0 的连续软件强制事件更新方式。所有 bit 为 0 时：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生 TEA 事件时更新；bit3 为 1：发生 TEB 事件时更新；bit4：发生同步事件时更新；bit5 为 1：关闭更新。(本文档中，TEA/B 指定时器值为寄存器 A/B 的值时生成的事件。) (R/W)

MCPWM_GEN0_A_CNTUFORCE_MODE 设置 PWM0A 的连续软件强制模式。0：关闭；1：低电平；2：高电平；3：关闭。(R/W)

MCPWM_GEN0_B_CNTUFORCE_MODE 设置 PWM0B 的连续软件强制模式。0：关闭；1：低电平；2：高电平；3：关闭。(R/W)

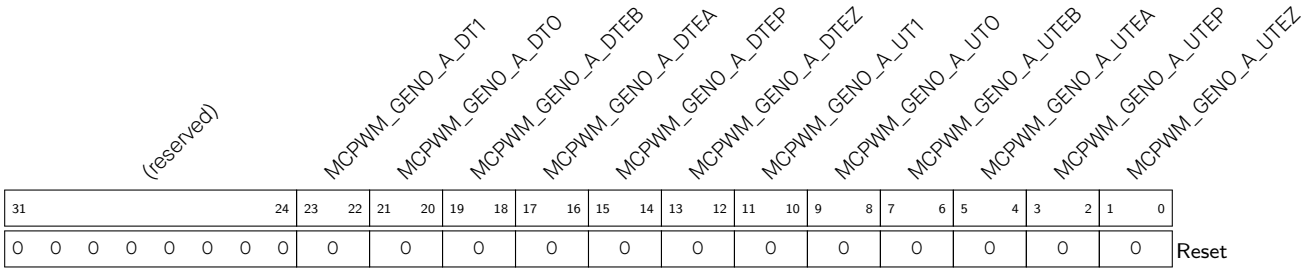
MCPWM_GEN0_A_NCIFORCE 该位的值取反时将触发 PWM0A 上的非连续即时软件强制事件。(R/W)

MCPWM_GEN0_A_NCIFORCE_MODE 设置用于 PWM0A 的非连续即时软件强制模式。0：关闭；1：低电平；2：高电平；3：关闭。(R/W)

MCPWM_GEN0_B_NCIFORCE 该位的值取反时将触发 PWM0B 上的非连续即时软件强制事件。(R/W)

MCPWM_GEN0_B_NCIFORCE_MODE 设置用于 PWM0B 的非连续即时软件强制模式。0：关闭；1：低电平；2：高电平；3：关闭。(R/W)

Register 36.21. MCPWM_GEN0_A_REG (0x0050)



MCPWM_GEN0_A_UTEZ 定时器递增时，TEZ 事件在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_A_UTEP 定时器递增时，TEP 事件在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_A_UTEA 定时器递增时，TEA 事件在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_A_UTEB 定时器递增时，TEB 事件在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_A_UT0 定时器递增时，event_t0 在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_A_UT1 定时器递增时，event_t1 在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_A_DTEZ 定时器递减时，TEZ 事件在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_A_DTEP 定时器递减时，TEP 事件在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

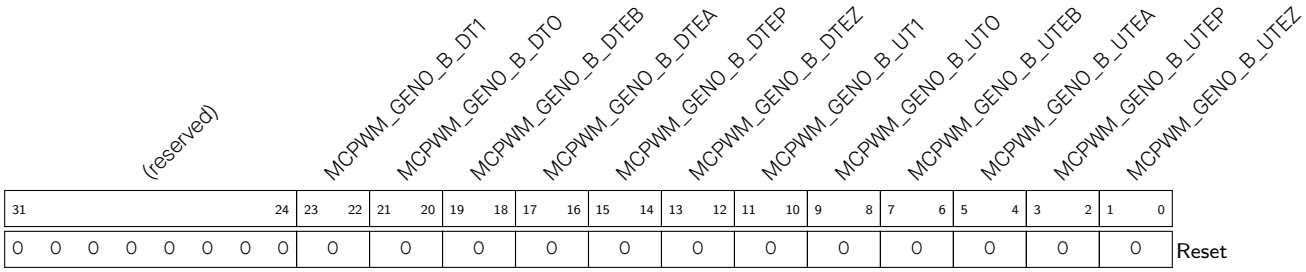
MCPWM_GEN0_A_DTEA 定时器递减时，TEA 事件在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_A_DTEB 定时器递减时，TEB 事件在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_A_DT0 定时器递减时，event_t0 在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_A_DT1 定时器递减时，event_t1 在 PWM0A 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

Register 36.22. MCPWM_GEN0_B_REG (0x0054)



MCPWM_GEN0_B_UTEZ 定时器递增时，TEZ 在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_B_UTEP 定时器递增时，TEP 在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_B_UTEA 定时器递增时，TEA 在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_B_UTEB 定时器递增时，TEB 在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_B_UT0 定时器递增时，event_t0 在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_B_UT1 定时器递增时，event_t1 在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_B_DTEZ 定时器递减时，TEZ 事件在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_B_DTEP 定时器递减时，TEP 事件在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

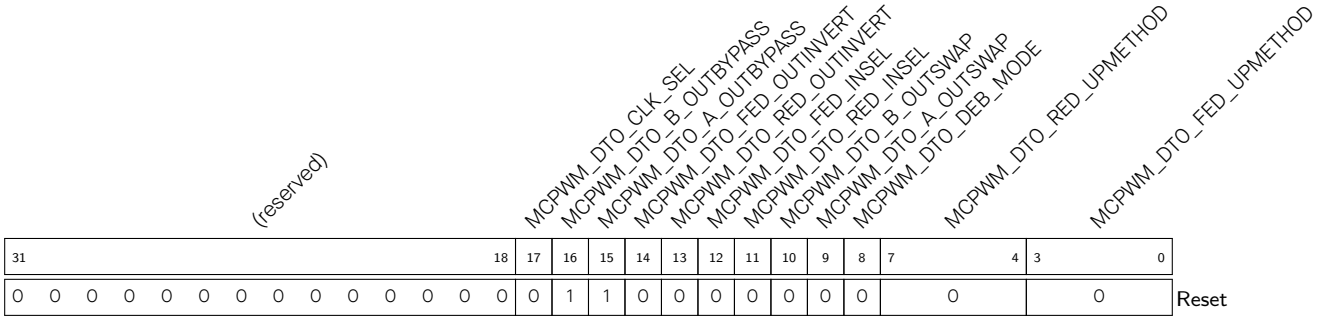
MCPWM_GEN0_B_DTEA 定时器递减时，TEA 事件在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_B_DTEB 定时器递减时，TEB 事件在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_B_DT0 定时器递减时，event_t0 事件在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN0_B_DT1 定时器递减时，event_t1 在 PWM0B 上触发的操作。0：波形无改变；1：拉低；2：拉高；3：取反。(R/W)

Register 36.23. MCPWM_DTO_CFG_REG (0x0058)



MCPWM_DTO_FED_UPMETHOD 下降沿延迟有效寄存器的更新方式。所有 bit 值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_DTO_RED_UPMETHOD 上升沿延迟有效寄存器的更新方式。所有 bit 值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_DTO_DEB_MODE 表 36.3-5 中的 S8, B 路双边沿模式。0：下降沿延迟/下降沿延迟分别在不同的路径中生效；1：下降沿延迟/下降沿延迟在路径 B 上生效，A 输出绕过或为 dulpB 模式。(R/W)

MCPWM_DTO_A_OUTSWAP 表 36.3-5 中的 S6。(R/W)

MCPWM_DTO_B_OUTSWAP 表 36.3-5 中的 S7。(R/W)

MCPWM_DTO_RED_INSEL 表 36.3-5 中的 S4。(R/W)

MCPWM_DTO_FED_INSEL 表 36.3-5 中的 S5。(R/W)

MCPWM_DTO_RED_OUTINVERT 表 36.3-5 中的 S2。(R/W)

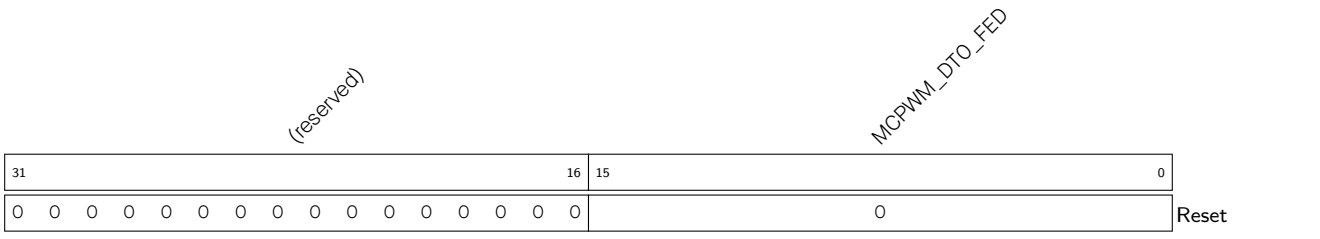
MCPWM_DTO_FED_OUTINVERT 表 36.3-5 中的 S3。(R/W)

MCPWM_DTO_A_OUTBYPASS 表 36.3-5 中的 S1。(R/W)

MCPWM_DTO_B_OUTBYPASS 表 36.3-5 中的 S0。(R/W)

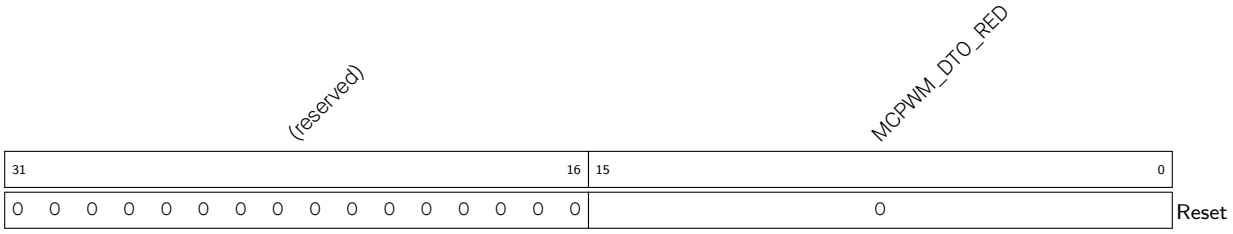
MCPWM_DTO_CLK_SEL 选择死区时间生成器 0 的时钟。0：PWM_clk；1：PT_clk。(R/W)

Register 36.24. MCPWM_DTO_FED_CFG_REG (0x005C)



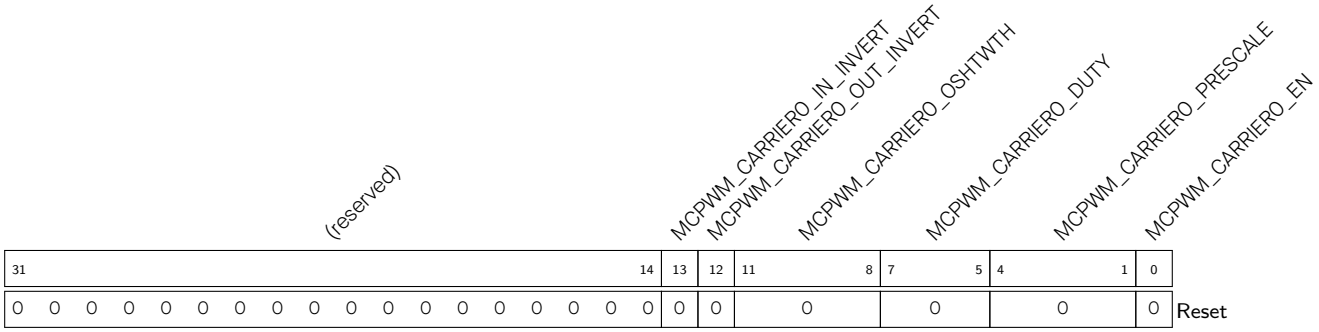
MCPWM_DTO_FED FED 的影子寄存器。(R/W)

Register 36.25. MCPWM_DT0_RED_CFG_REG (0x0060)



MCPWM_DT0_RED RED 的影子寄存器。(R/W)

Register 36.26. MCPWM_CARRIER0_CFG_REG (0x0064)



MCPWM_CARRIER0_EN 置 1 时，使能载波 0 的功能。清零时，载波 0 被绕过。(R/W)

MCPWM_CARRIER0_PRESCALE PWM 载波 0 时钟 (PC_clk) 的预分频值。PC_clk 周期 = PWM_clk 周期 * (PWM_CARRIER0_PRESCALE + 1)。(R/W)

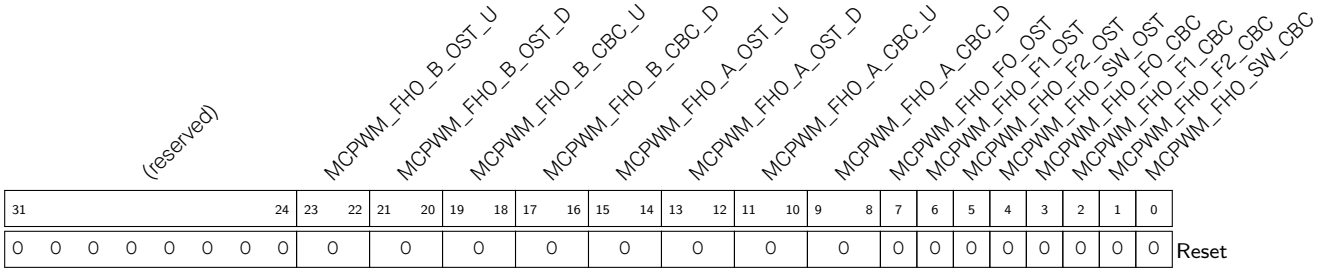
MCPWM_CARRIER0_DUTY 选择载波占空比。占空比 = PWM_CARRIER0_DUTY/8。(R/W)

MCPWM_CARRIER0_OSHTWTH 载波第一个脉冲的宽度，单位为载波周期。(R/W)

MCPWM_CARRIER0_OUT_INVERT 置 1 时，将此模块的 PWM0A 和 PWM0B 输出反相。(R/W)

MCPWM_CARRIER0_IN_INVERT 置 1 时，将此模块的 PWM0A 和 PWM0B 输入反相。(R/W)

Register 36.27. MCPWM_FH0_CFG0_REG (0x0068)



MCPWM_FH0_SW_CBC 使能软件强制逐周期模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH0_F2_CBC 设置 fault_event2 是否触发逐周期模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH0_F1_CBC 设置 fault_event1 是否触发逐周期模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH0_F0_CBC 设置 fault_event0 是否触发逐周期模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH0_SW_OST 软件强制一次性模式操作的使能寄存器。0：关闭；1：使能。(R/W)

MCPWM_FH0_F2_OST 设置 fault_event2 是否触发一次性模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH0_F1_OST 设置 fault_event1 是否触发一次性模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH0_F0_OST 设置 fault_event0 是否触发一次性模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH0_A_CBC_D 定时器递减计数并且发生故障事件时，PWM0A 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH0_A_CBC_U 定时器递增计数并且发生故障事件时，PWM0A 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH0_A_OST_D 定时器递减计数并且发生故障事件时，PWM0A 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH0_A_OST_U 定时器递增计数并且发生故障事件时，PWM0A 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

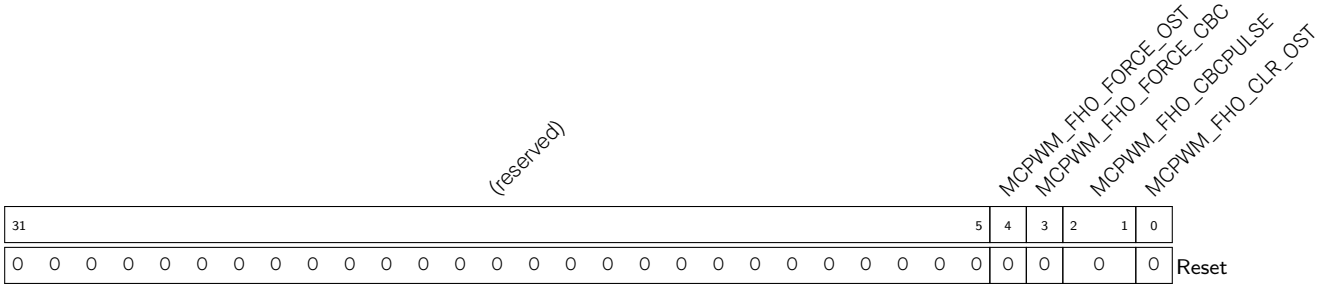
MCPWM_FH0_B_CBC_D 定时器递减计数并且发生故障事件时，PWM0B 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH0_B_CBC_U 定时器递增计数并且发生故障事件时，PWM0B 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH0_B_OST_D 定时器递减计数并且发生故障事件时，PWM0B 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH0_B_OST_U 定时器递增计数并且发生故障事件时，PWM0B 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

Register 36.28. MCPWM_FH0_CFG1_REG (0x006C)



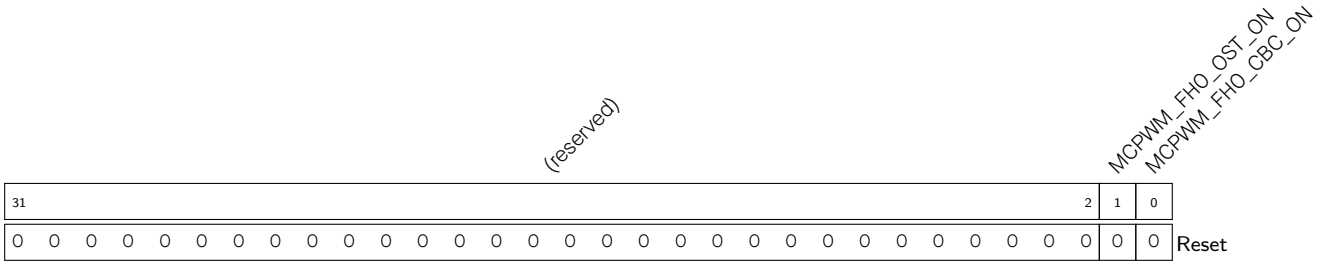
MCPWM_FH0_CLR_OST 上升沿时将清除持续一次性模式操作。(R/W)

MCPWM_FH0_CBCPULSE 设置逐周期模式操作更新的时间点。bit0 为 1：发生 TEZ 事件时；bit1 为 1：发生 TEP 事件时。(R/W)

MCPWM_FH0_FORCE_CBC 通过软件将此位的值取反，可触发逐周期模式的操作。(R/W)

MCPWM_FH0_FORCE_OST 通过软件将此位的值取反，可触发一次性模式的操作。(R/W)

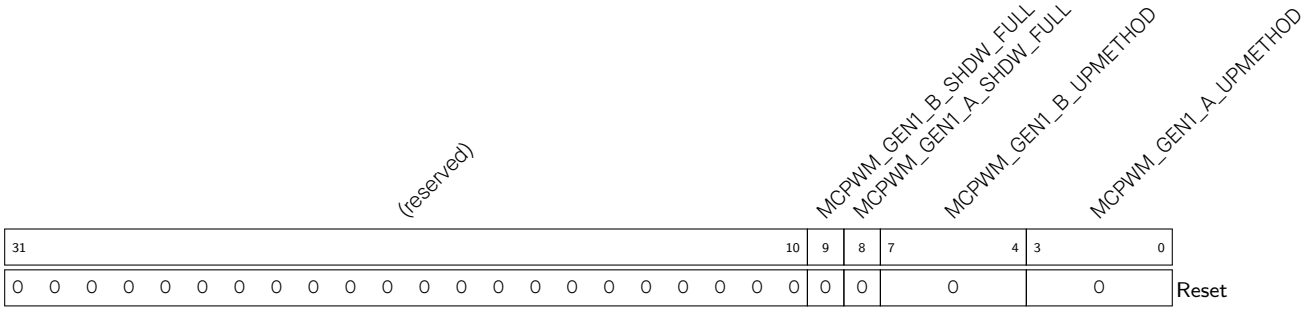
Register 36.29. MCPWM_FH0_STATUS_REG (0x0070)



MCPWM_FH0_CBC_ON 由硬件置 1 和清零。置 1 时，逐周期模式的操作正在进行进行。(RO)

MCPWM_FH0_OST_ON 由硬件置 1 和清零。置 1 时，一次性模式的操作正在进行进行。(RO)

Register 36.30. MCPWM_GEN1_STMP_CFG_REG (0x0074)



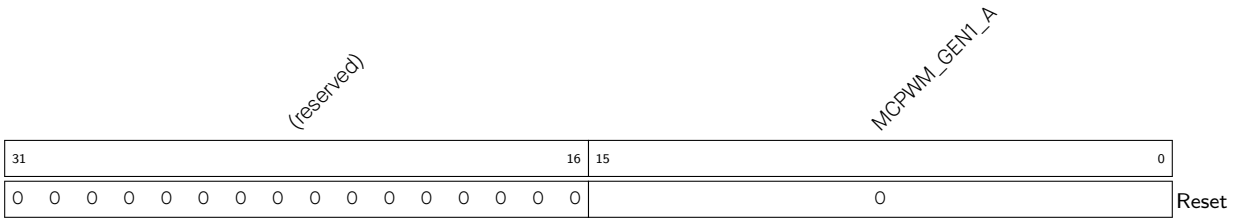
MCPWM_GEN1_A_UPMETHOD PWM 生成器 1 时间戳寄存器 A 有效寄存器的更新方式。所有 bit 值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_GEN1_B_UPMETHOD PWM 生成器 1 时间戳寄存器 B 有效寄存器的更新方式。所有 bit 值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_GEN1_A_SHDW_FULL 由硬件置 1 和清零。置 1 时，PWM 生成器 1 时间戳寄存器 A 的影子寄存器被写入，写入的值将传输给有效寄存器 A。清零时，有效寄存器 A 中写入其影子寄存器最新的值。(R/WTC/SC)

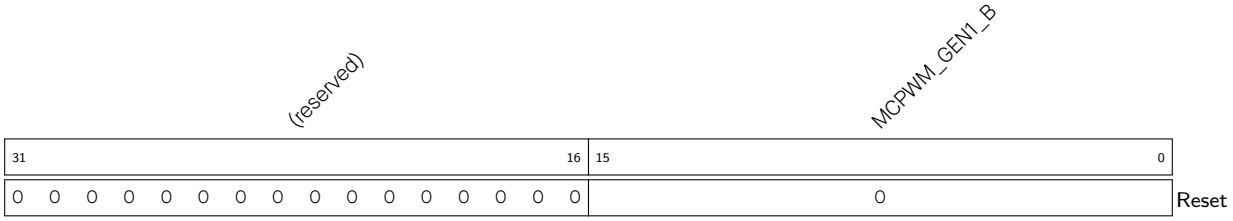
MCPWM_GEN1_B_SHDW_FULL 由硬件置 1 和清零。置 1 时，PWM 生成器 1 时间戳寄存器 B 的影子寄存器被写入，写入的值将传输给有效寄存器 B。清零时，有效寄存器 B 中写入其影子寄存器最新的值。(R/WTC/SC)

Register 36.31. MCPWM_GEN1_TSTMP_A_REG (0x0078)



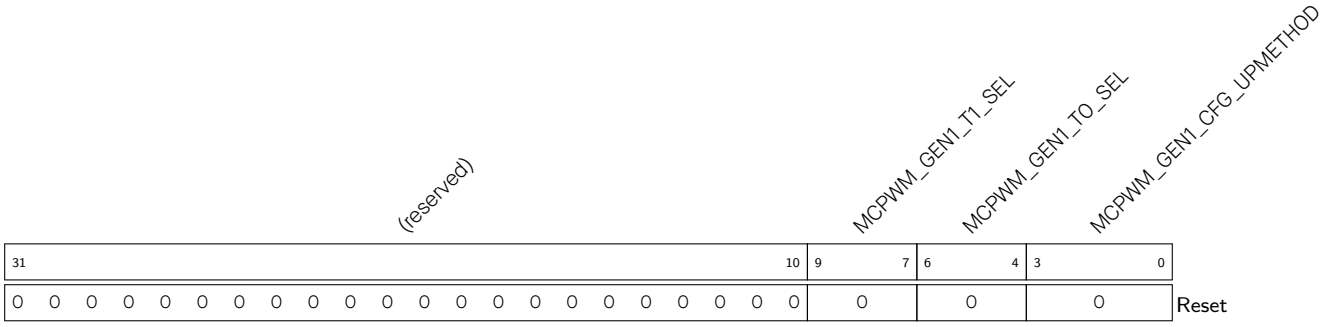
MCPWM_GEN1_A PWM 生成器 1 时间戳寄存器 A 的影子寄存器。(R/W)

Register 36.32. MCPWM_GEN1_TSTMP_B_REG (0x007C)



MCPWM_GEN1_B PWM 生成器 1 时间戳寄存器 B 的影子寄存器。(R/W)

Register 36.33. MCPWM_GEN1_CFG0_REG (0x0080)

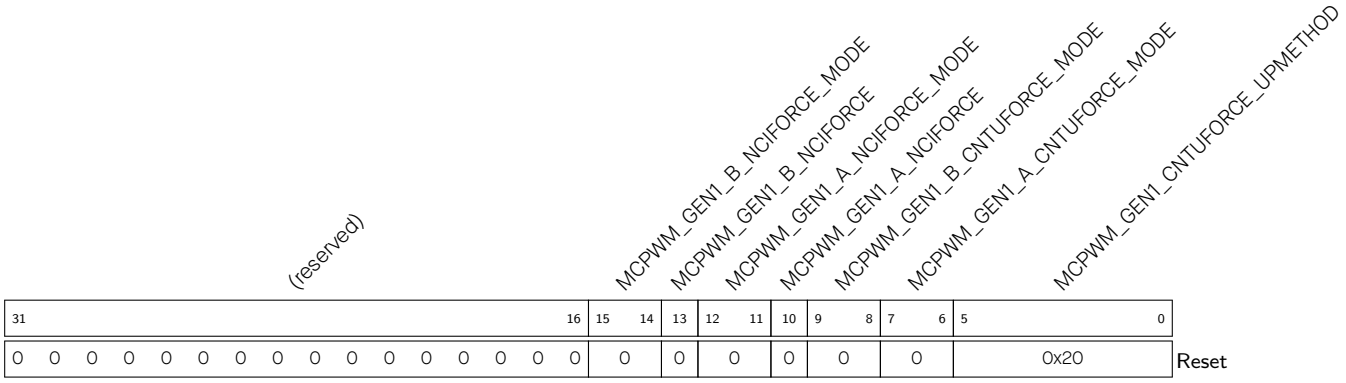


MCPWM_GEN1_CFG_UPMETHOD PWM 生成器 1 有效寄存器的更新方式。所有 bit 值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_GEN1_T0_SEL 选择 PWM 生成器 1 event_t0 的信号源，立即生效。0：fault_event0；1：fault_event1；2：fault_event2；3：sync_taken；4：无。(R/W)

MCPWM_GEN1_T1_SEL 选择 PWM 生成器 1 event_t1 的信号源，立即生效。0：fault_event0；1：fault_event1；2：fault_event2；3：sync_taken；4：无。(R/W)

Register 36.34. MCPWM_GEN1_FORCE_REG (0x0084)



MCPWM_GEN1_CNTUFORCE_UPMETHOD PWM 生成器 1 持续软件强制的更新方式。0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生 TEA 事件时更新；bit3 为 1：发生 TEB 事件时更新；bit4 为 1：发生同步时间时更新；bit5 为 1：关闭更新。(本文档中的 TEA/B 表示计时器值等于寄存器 A/B 生成的事件。) (R/W)

MCPWM_GEN1_A_CNTUFORCE_MODE 用于 PWM1A 的连续软件强制事件。0：关闭；1：拉低；2：拉高；3：关闭。(R/W)

MCPWM_GEN1_B_CNTUFORCE_MODE 用于 PWM1B 的连续软件强制事件。0：关闭；1：拉低；2：拉高；3：关闭。(R/W)

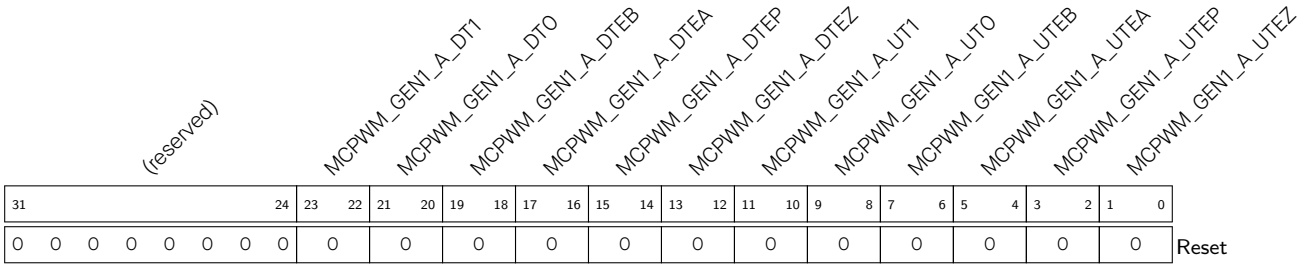
MCPWM_GEN1_A_NCIFORCE 该位的值取反时将触发 PWM1A 上的非连续即时软件强制事件。(R/W)

MCPWM_GEN1_A_NCIFORCE_MODE 用于 PWM1A 的非持续性即时软件强制事件。0：关闭；1：拉低；2：拉高；3：关闭。(R/W)

MCPWM_GEN1_B_NCIFORCE 该位的值取反时将触发 PWM1B 上的非连续即时软件强制事件。(R/W)

MCPWM_GEN1_B_NCIFORCE_MODE 用于 PWM1B 的非持续性即时软件强制事件。0：关闭；1：拉低；2：拉高；3：关闭。(R/W)

Register 36.35. MCPWM_GEN1_A_REG (0x0088)



MCPWM_GEN1_A_UTEZ 定时器递增计数时，TEZ 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_A_UTEP 定时器递增计数时，TEP 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_A_UTEA 定时器递增计数时，TEA 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_A_UTEB 定时器递增计数时，TEB 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_A_UT0 定时器递增计数时，event_t0 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_A_UT1 定时器递增计数时，event_t1 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_A_DTEZ 定时器递减计数时，TEZ 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_A_DTEP 定时器递减计数时，TEP 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

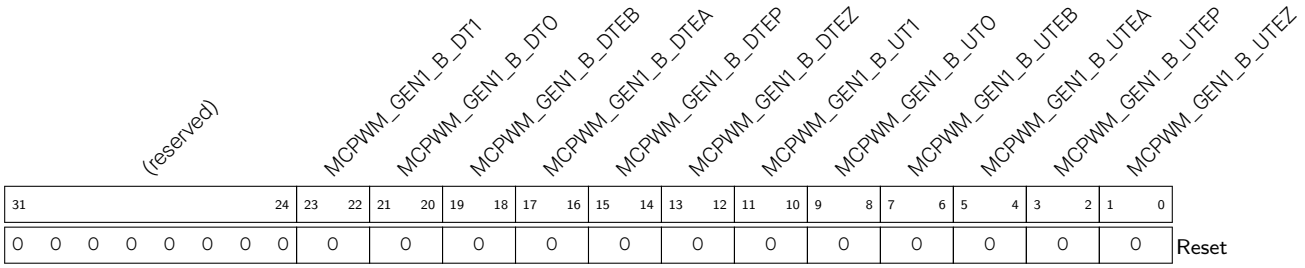
MCPWM_GEN1_A_DTEA 定时器递减计数时，TEA 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_A_DTEB 定时器递减计数时，TEB 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_A_DT0 定时器递减计数时，event_t0 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_A_DT1 定时器递减计数时，event_t1 在 PWM1A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

Register 36.36. MCPWM_GEN1_B_REG (0x008C)



MCPWM_GEN1_B_UTEZ 定时器递增计数时，TEZ 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_UTEP 定时器递增计数时，TEP 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_UTEA 定时器递增计数时，TEA 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_UTEB 定时器递增计数时，TEB 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_UT0 定时器递增计数时，event_t0 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_UT1 定时器递增计数时，event_t1 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_DTEZ 定时器递减计数时，TEZ 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_DTEP 定时器递减计数时，TEP 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_DTEA 定时器递减计数时，TEA 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_DTEB 定时器递减计数时，TEB 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_DT0 定时器递减计数时，event_t0 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN1_B_DT1 定时器递减计数时，event_t1 在 PWM1B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

Register 36.37. MCPWM_DT1_CFG_REG (0x0090)



MCPWM_DT1_FED_UPMETHOD FED（下降沿延迟）有效寄存器的更新方式。0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步事件时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_DT1_RED_UPMETHOD RED（上升沿延迟）有效寄存器的更新方式。0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步事件时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_DT1_DEB_MODE 表 36.3-5 中的 S8, B 路双边沿模式。0：下降沿延迟/下降沿延迟分别在不同的路径中生效；1：下降沿延迟/下降沿延迟在路径 B 上生效，A 输出绕过或为 dulpB 模式。(R/W)

MCPWM_DT1_A_OUTSWAP 表 36.3-5 中的 S6。(R/W)

MCPWM_DT1_B_OUTSWAP 表 36.3-5 中的 S7。(R/W)

MCPWM_DT1_RED_INSEL 表 36.3-5 中的 S4。(R/W)

MCPWM_DT1_FED_INSEL 表 36.3-5 中的 S5。(R/W)

MCPWM_DT1_RED_OUTINVERT 表 36.3-5 中的 S2。(R/W)

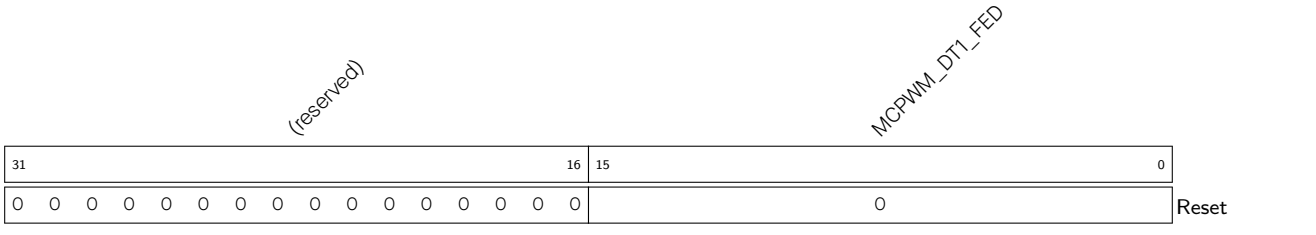
MCPWM_DT1_FED_OUTINVERT 表 36.3-5 中的 S3。(R/W)

MCPWM_DT1_A_OUTBYPASS 表 36.3-5 中的 S1。(R/W)

MCPWM_DT1_B_OUTBYPASS 表 36.3-5 中的 S0。(R/W)

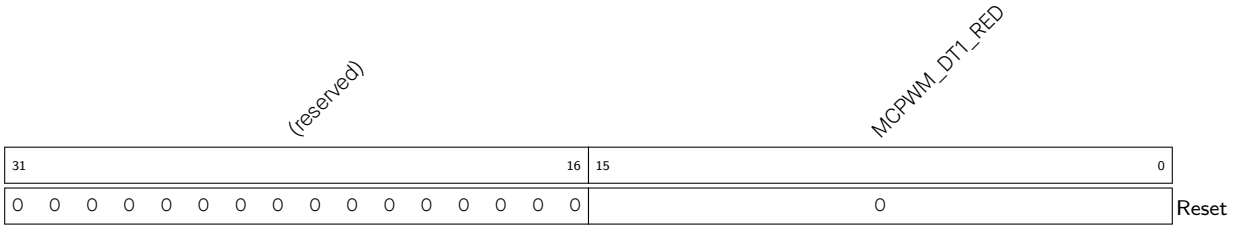
MCPWM_DT1_CLK_SEL 设置死区时间生成器时钟。0：PWM_clk；1：PT_clk。(R/W)

Register 36.38. MCPWM_DT1_FED_CFG_REG (0x0094)



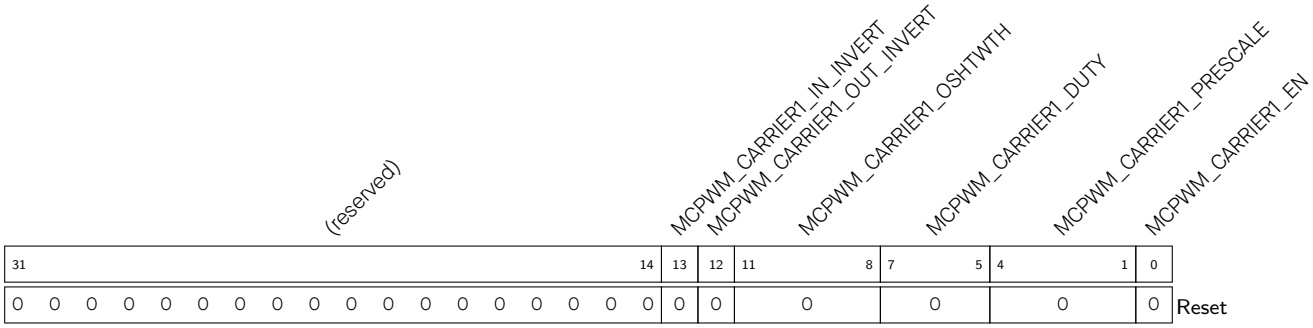
MCPWM_DT1_FED FED 影子寄存器。(R/W)

Register 36.39. MCPWM_DT1_RED_CFG_REG (0x0098)



MCPWM_DT1_RED RED 影子寄存器。(R/W)

Register 36.40. MCPWM_CARRIER1_CFG_REG (0x009C)



MCPWM_CARRIER1_EN 置 1 时，使能载波 1 功能。此位清零时，绕过载波 1。(R/W)

MCPWM_CARRIER1_PRESCALE PWM 载波 1 时钟 (PC_clk) 预分频值。PC_clk 周期 = PWM_clk 周期 * (PWM_CARRIER0_PRESCALE + 1)。(R/W)

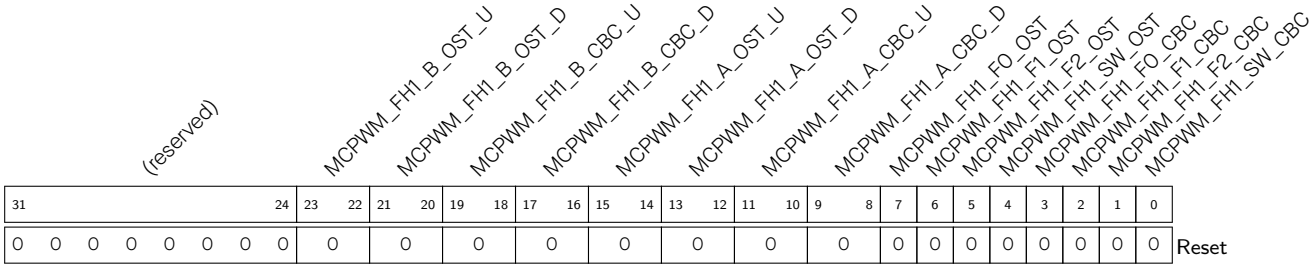
MCPWM_CARRIER1_DUTY 设置载波占空比。占空比 = PWM_CARRIER0_DUTY/8。(R/W)

MCPWM_CARRIER1_OSHTWTH 载波第一个脉冲的宽度，单位为载波周期。(R/W)

MCPWM_CARRIER1_OUT_INVERT 置 1 时，将此模块的 PWM1A 和 PWM1B 输出反相。(R/W)

MCPWM_CARRIER1_IN_INVERT 置 1 时，将此模块的 PWM1A 和 PWM1B 输入反相。(R/W)

Register 36.41. MCPWM_FH1_CFG0_REG (0x00A0)



MCPWM_FH1_SW_CBC 软件强制逐周期模式操作的使能寄存器。0：关闭；1：使能。(R/W)

MCPWM_FH1_F2_CBC 设置 fault_event2 触发逐周期模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH1_F1_CBC 设置 fault_event1 触发逐周期模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH1_F0_CBC 设置 fault_event0 触发逐周期模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH1_SW_OST 软件强制一次性模式操作的使能寄存器。0：关闭；1：使能。(R/W)

MCPWM_FH1_F2_OST 设置 fault_event2 触发一次性模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH1_F1_OST 设置 fault_event1 触发一次性模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH1_F0_OST 设置 fault_event0 触发一次性模式操作。0：disable, 1：enable (R/W)

MCPWM_FH1_A_CBC_D 定时器递减计数并且发生故障事件时，PWM1A 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH1_A_CBC_U 定时器递增计数并且发生故障事件时，PWM1A 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH1_A_OST_D 定时器递减计数并且发生故障事件时，PWM1A 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH1_A_OST_U 定时器递增计数并且发生故障事件时，PWM1A 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

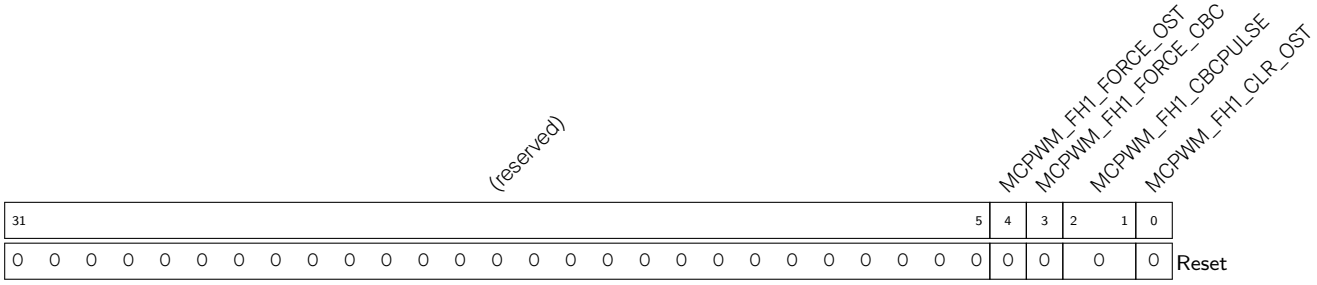
MCPWM_FH1_B_CBC_D 定时器递减计数并且发生故障事件时，PWM1B 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH1_B_CBC_U 定时器递增计数并且发生故障事件时，PWM1B 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH1_B_OST_D 定时器递减计数并且发生故障事件时，PWM1B 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH1_B_OST_U 定时器递增计数并且发生故障事件时，PWM1B 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

Register 36.42. MCPWM_FH1_CFG1_REG (0x00A4)



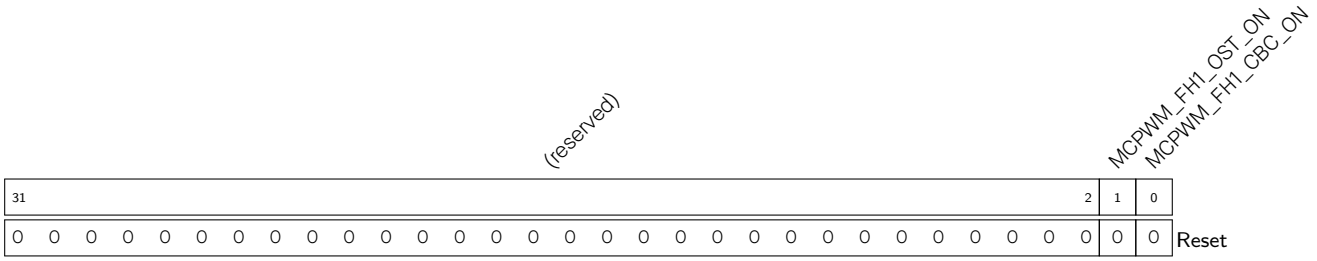
MCPWM_FH1_CLR_OST 置 1 清除正在进行的一次性模式操作。(R/W)

MCPWM_FH1_CBCPULSE 设置逐周期模式操作的更新方式。bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新。(R/W)

MCPWM_FH1_FORCE_CBC 通过软件将该位取反时，触发逐周期模式操作。(R/W)

MCPWM_FH1_FORCE_OST 通过软件将该位取反时，触发一次性模式操作。(R/W)

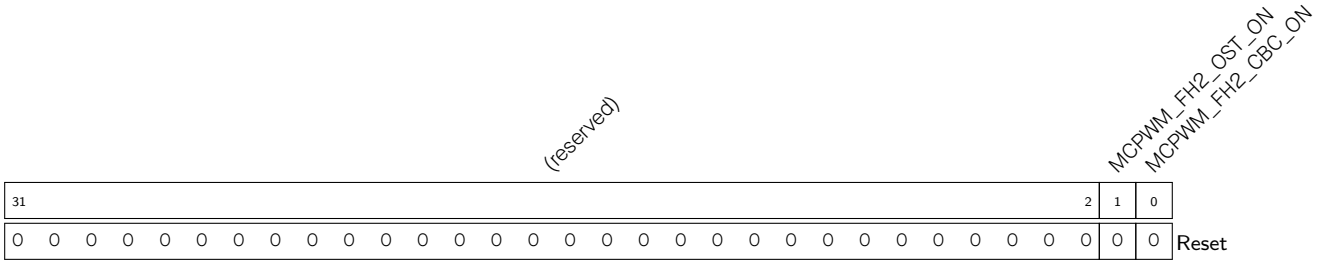
Register 36.43. MCPWM_FH1_STATUS_REG (0x00A8)



MCPWM_FH1_CBC_ON 通过硬件置位或清零。置 1 时，逐周期模式操作正在进行。(RO)

MCPWM_FH1_OST_ON 通过硬件置位或清零。置 1 时，一次性模式操作正在进行。(RO)

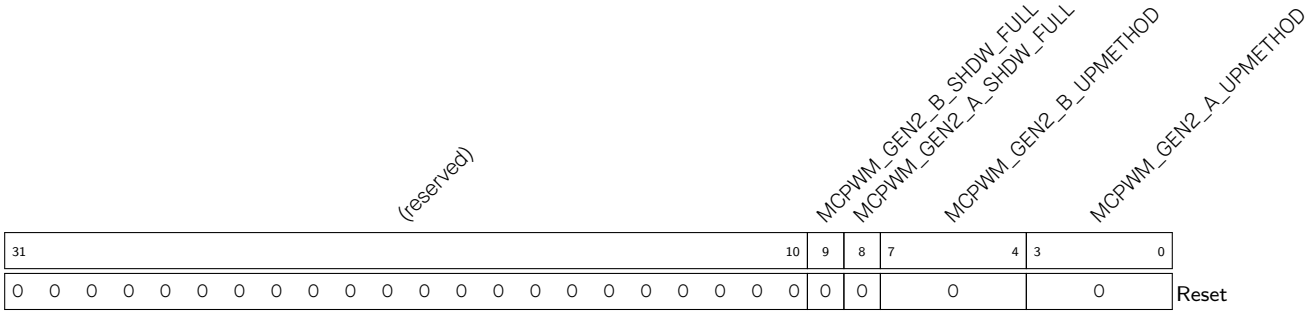
Register 36.44. MCPWM_FH2_STATUS_REG (0x00E0)



MCPWM_FH2_CBC_ON 通过硬件置位或清零。置 1 时，逐周期模式操作正在进行。(RO)

MCPWM_FH2_OST_ON 通过硬件置位或清零。置 1 时，一次性模式操作正在进行。(RO)

Register 36.45. MCPWM_GEN2_STMP_CFG_REG (0x00AC)



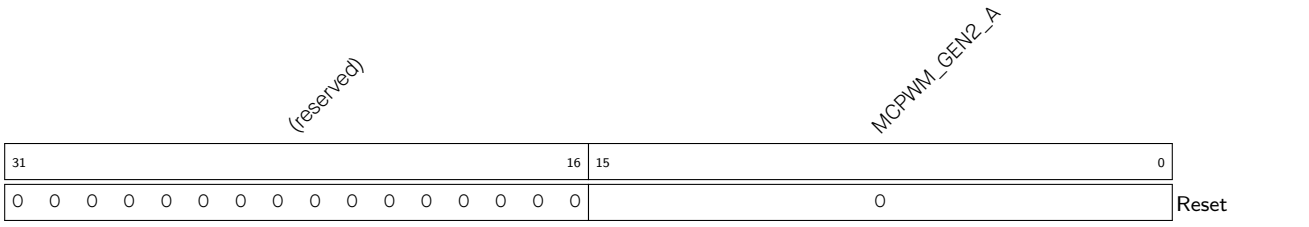
MCPWM_GEN2_A_UPMETHOD 生成器 2 时间戳寄存器 A 有效寄存器的更新方式。所有 bit 值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_GEN2_B_UPMETHOD 生成器 2 时间戳寄存器 B 有效寄存器的更新方式。所有 bit 值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_GEN2_A_SHDW_FULL 由硬件置 1 和清零。置 1 时，PWM 生成器 2 时间戳寄存器 A 的影子寄存器被写入，写入的值将传输给有效寄存器 A。清零时，有效寄存器 A 中写入其影子寄存器最新的值。(R/WTC/SC)

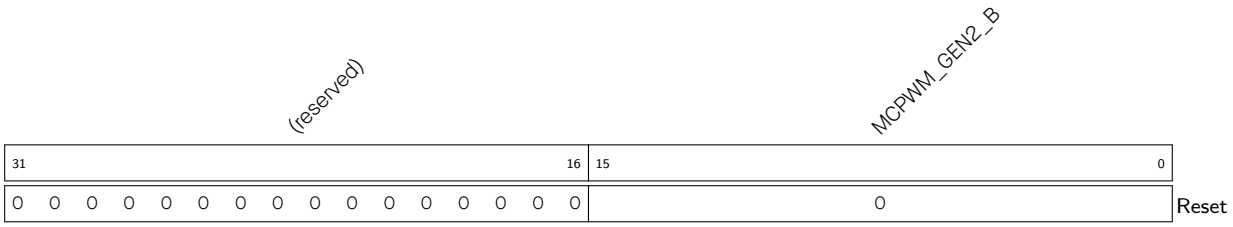
MCPWM_GEN2_B_SHDW_FULL 由硬件置 1 和清零。置 1 时，PWM 生成器 2 时间戳寄存器 B 的影子寄存器被写入，写入的值将传输给有效寄存器 B。清零时，有效寄存器 B 中写入其影子寄存器最新的值。(R/WTC/SC)

Register 36.46. MCPWM_GEN2_STMP_A_REG (0x00B0)



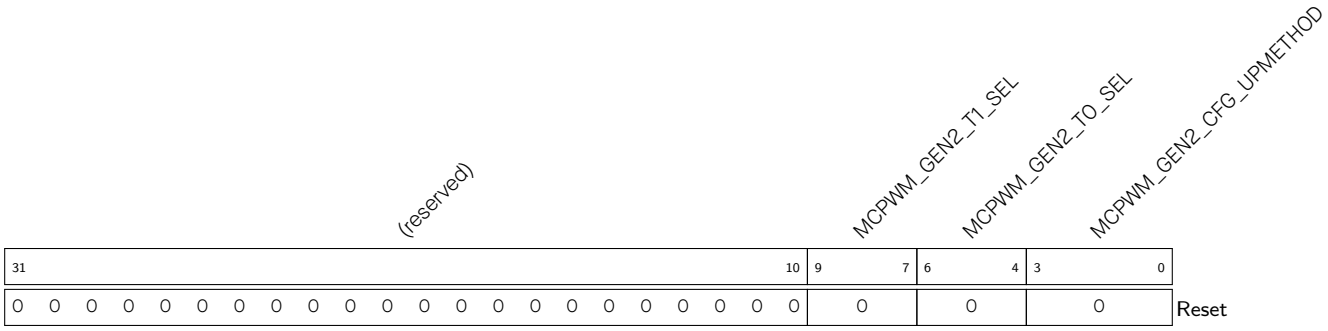
MCPWM_GEN2_A PWM 生成器 2 时间戳 A 的影子寄存器。(R/W)

Register 36.47. MCPWM_GEN2_STMP_B_REG (0x00B4)



MCPWM_GEN2_B PWM 生成器 2 时间戳 B 的影子寄存器。(R/W)

Register 36.48. MCPWM_GEN2_CFG0_REG (0x00B8)

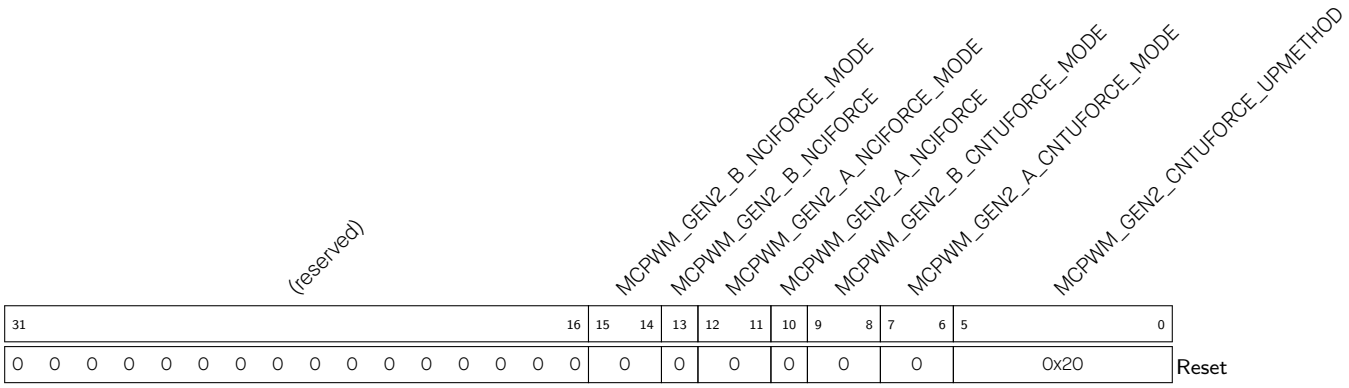


MCPWM_GEN2_CFG_UPMETHOD PWM 生成器 2 有效配置寄存器的更新方式。所有 bit 值为 0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步时间时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_GEN2_T0_SEL 设置 PWM 操作器 2 event_t0 信号源，立即生效。0：fault_event0；1：fault_event1；2：fault_event2；3：sync_taken；4：无。(R/W)

MCPWM_GEN2_T1_SEL 设置 PWM 操作器 2 event_t1 信号源，立即生效。0：fault_event0；1：fault_event1；2：fault_event2；3：sync_taken；4：无。(R/W)

Register 36.49. MCPWM_GEN2_FORCE_REG (0x00BC)



MCPWM_GEN2_CNTUFORCE_UPMETHOD PWM 生成器 2 的持续性软件强制事件的更新方式。0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生 TEA 事件时更新；bit3 为 1：发生 TEB 事件时更新；bit4 为 1：发生同步时间时更新；bit5 为 1：关闭更新。(本文档中的 TEA/B 表示计时器值等于寄存器 A/B 生成的事件。) (R/W)

MCPWM_GEN2_A_CNTUFORCE_MODE 用于 PWM2A 的持续性即时软件强制事件。0：关闭；1：拉低；2：拉高；3：关闭。(R/W)

MCPWM_GEN2_B_CNTUFORCE_MODE 用于 PWM2B 的持续性即时软件强制事件。0：关闭；1：拉低；2：拉高；3：关闭。(R/W)

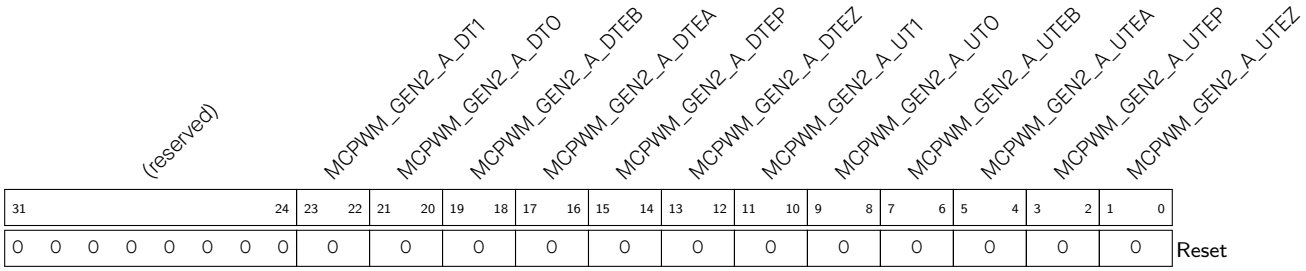
MCPWM_GEN2_A_NCIFORCE 该位的值取反时将触发 PWM2A 上的非连续即时软件强制事件。(R/W)

MCPWM_GEN2_A_NCIFORCE_MODE 用于 PWM2A 的非持续性即时软件强制事件。0：关闭；1：拉低；2：拉高；3：关闭。(R/W)

MCPWM_GEN2_B_NCIFORCE 该位的值取反时将触发 PWM2B 上的非连续即时软件强制事件。(R/W)

MCPWM_GEN2_B_NCIFORCE_MODE 用于设置 PWM2B 的非持续性即时软件强制模式。0：关闭；1：拉低；2：拉高；3：关闭。(R/W)

Register 36.50. MCPWM_GEN2_A_REG (0x00C0)



MCPWM_GEN2_A_UTEZ 定时器递增时，TEZ 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_A_UTEP 定时器递增时，TEP 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_A_UTEA 定时器递增时，TEA 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_A_UTEB 定时器递增时，TEB 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_A_UT0 定时器递增时，event_t0 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_A_UT1 定时器递增时，event_t1 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_A_DTEZ 定时器递减时，TEZ 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_A_DTEP 定时器递减时，TEP 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

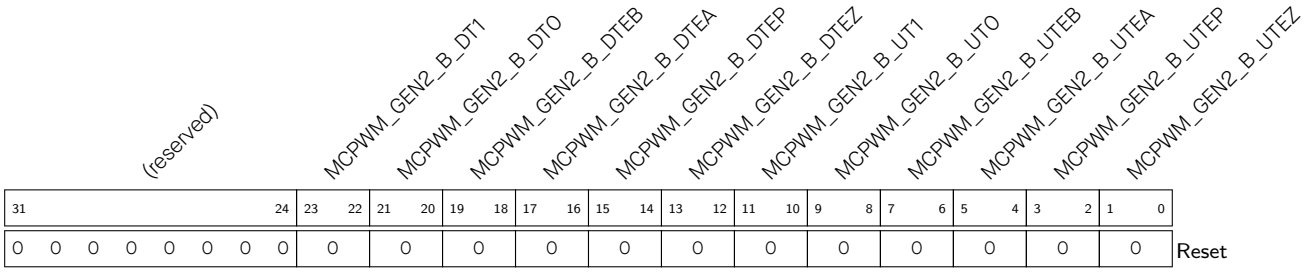
MCPWM_GEN2_A_DTEA 定时器递减时，TEA 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_A_DTEB 定时器递减时，TEB 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_A_DT0 定时器递减时，event_t0 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_A_DT1 定时器递减时，event_t1 在 PWM2A 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

Register 36.51. MCPWM_GEN2_B_REG (0x00C4)



MCPWM_GEN2_B_UTEZ 定时器递增时，TEZ 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_B_UTEP 定时器递增时，TEP 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_B_UTEA 定时器递增时，TEA 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_B_UTEB 定时器递增时，TEB 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_B_UTO 定时器递增时，event_t0 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_B_UT1 定时器递增时，event_t1 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_B_DTEZ 定时器递减时，TEZ 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_B_DTEP 定时器递减时，TEP 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

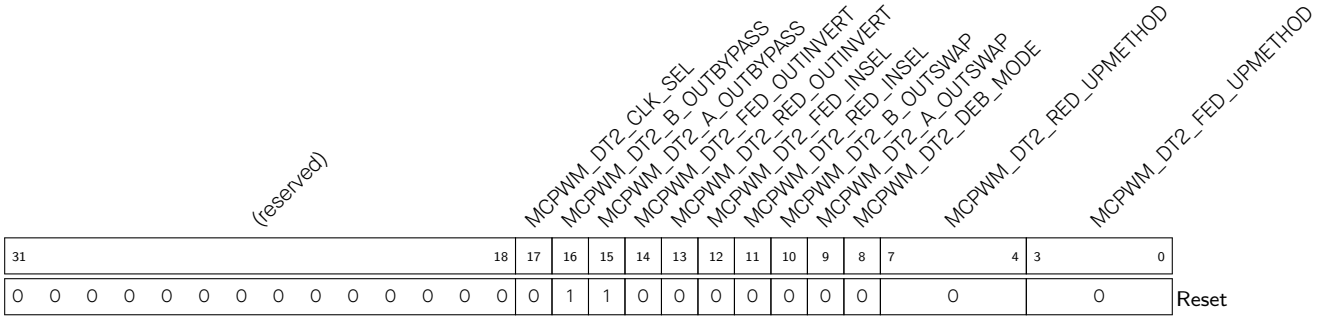
MCPWM_GEN2_B_DTEA 定时器递减时，TEA 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_B_DTEB 定时器递减时，TEB 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_B_DTO 定时器递减时，event_t0 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

MCPWM_GEN2_B_DT1 定时器递减时，event_t1 在 PWM2B 上触发的操作。0：无；1：拉低；2：拉高；3：取反。(R/W)

Register 36.52. MCPWM_DT2_CFG_REG (0x00C8)



MCPWM_DT2_FED_UPMETHOD FED（下降沿延迟）有效寄存器的更新方式。0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步事件时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_DT2_RED_UPMETHOD RED（上升沿延迟）有效寄存器的更新方式。0：立即更新；bit0 为 1：发生 TEZ 事件时更新；bit1 为 1：发生 TEP 事件时更新；bit2 为 1：发生同步事件时更新；bit3 为 1：关闭更新。(R/W)

MCPWM_DT2_DEB_MODE 表 36.3-5 中的 S8, B 路双边沿模式。0：下降沿延迟/下降沿延迟分别在不同的路径中生效；1：下降沿延迟/下降沿延迟在路径 B 上生效，A 输出绕过或为 dulpB 模式。(R/W)

MCPWM_DT2_A_OUTSWAP 表 36.3-5 中的 S6。(R/W)

MCPWM_DT2_B_OUTSWAP 表 36.3-5 中的 S7。(R/W)

MCPWM_DT2_RED_INSEL 表 36.3-5 中的 S4。(R/W)

MCPWM_DT2_FED_INSEL 表 36.3-5 中的 S5。(R/W)

MCPWM_DT2_RED_OUTINVERT 表 36.3-5 中的 S2。(R/W)

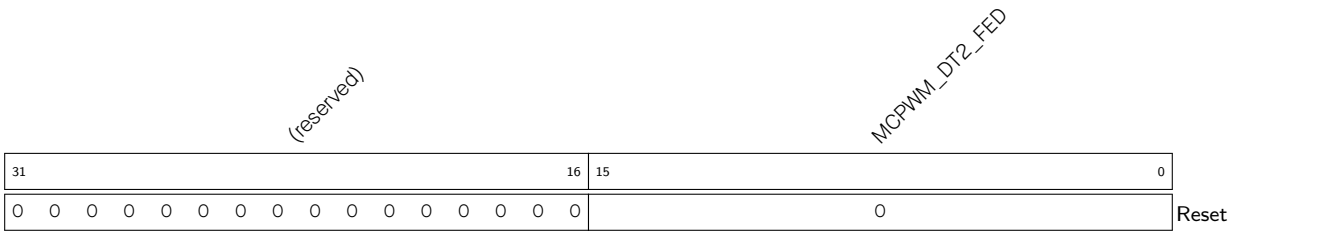
MCPWM_DT2_FED_OUTINVERT 表 36.3-5 中的 S3。(R/W)

MCPWM_DT2_A_OUTBYPASS 表 36.3-5 中的 S1。(R/W)

MCPWM_DT2_B_OUTBYPASS 表 36.3-5 中的 S0。(R/W)

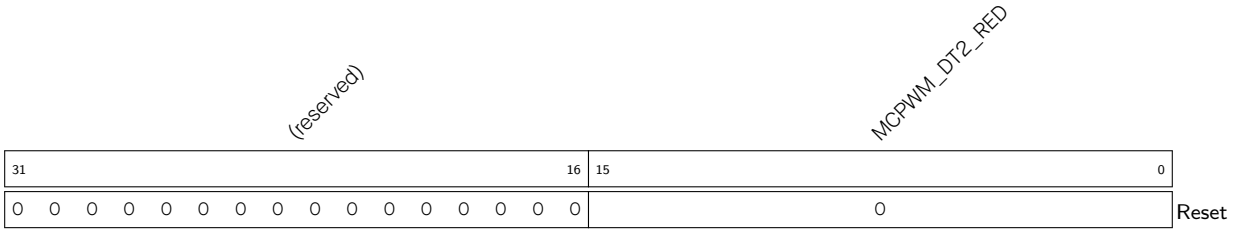
MCPWM_DT2_CLK_SEL 设置死区时间生成器时钟。0：PWM_clk；1：PT_clk。(R/W)

Register 36.53. MCPWM_DT2_FED_CFG_REG (0x00CC)



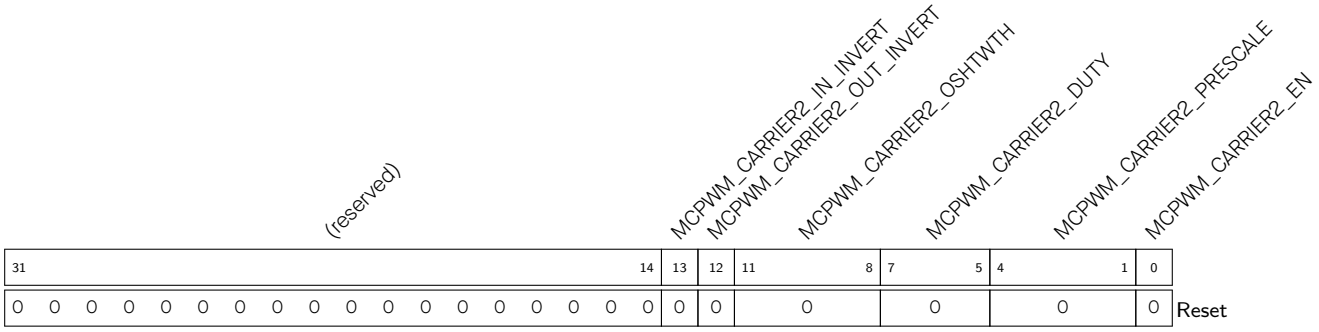
MCPWM_DT2_FED FED 影子寄存器。(R/W)

Register 36.54. MCPWM_DT2_RED_CFG_REG (0x00D0)



MCPWM_DT2_RED RED 影子寄存器。(R/W)

Register 36.55. MCPWM_CARRIER2_CFG_REG (0x00D4)



MCPWM_CARRIER2_EN 置 1 时，使能载波 2 功能。清零时，载波 2 被绕过。(R/W)

MCPWM_CARRIER2_PRESCALE PWM 载波 2 时钟 (PC_clk) 的预分频值。PC_clk 周期 = PWM_clk 周期 * (PWM_CARRIER0_PRESCALE + 1)。(R/W)

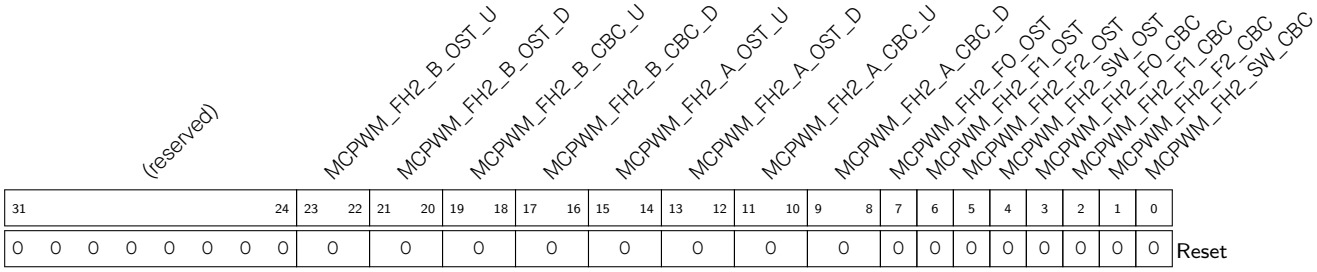
MCPWM_CARRIER2_DUTY 设置载波占空比。占空比 = PWM_CARRIER0_DUTY / 8。(R/W)

MCPWM_CARRIER2_OSHTWTH 载波第一个脉冲的宽度，单位为载波周期。(R/W)

MCPWM_CARRIER2_OUT_INVERT 置 1 时，将此模块的 PWM2A 和 PWM2B 输出反相。(R/W)

MCPWM_CARRIER2_IN_INVERT 置 1 时，将此模块的 PWM2A 和 PWM2B 输入反相。(R/W)

Register 36.56. MCPWM_FH2_CFG0_REG (0x00D8)



MCPWM_FH2_SW_CBC 软件强制逐周期模式操作的使能寄存器。0：关闭；1：使能。(R/W)

MCPWM_FH2_F2_CBC 设置 fault_event2 触发逐周期模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH2_F1_CBC 设置 fault_event1 触发逐周期模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH2_F0_CBC 设置 fault_event0 触发逐周期模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH2_SW_OST 软件强制一次性模式操作的使能寄存器。0：关闭；1：使能。(R/W)

MCPWM_FH2_F2_OST 设置 fault_event2 触发一次性模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH2_F1_OST 设置 fault_event1 触发一次性模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH2_F0_OST 设置 fault_event0 触发一次性模式操作。0：关闭；1：使能。(R/W)

MCPWM_FH2_A_CBC_D 发生故障事件并且定时器递减时，PWM2A 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH2_A_CBC_U 发生故障事件并且定时器递增时，PWM2A 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH2_A_OST_D 发生故障事件并且定时器递减时，PWM2A 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH2_A_OST_U 发生故障事件并且定时器递增时，PWM2A 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

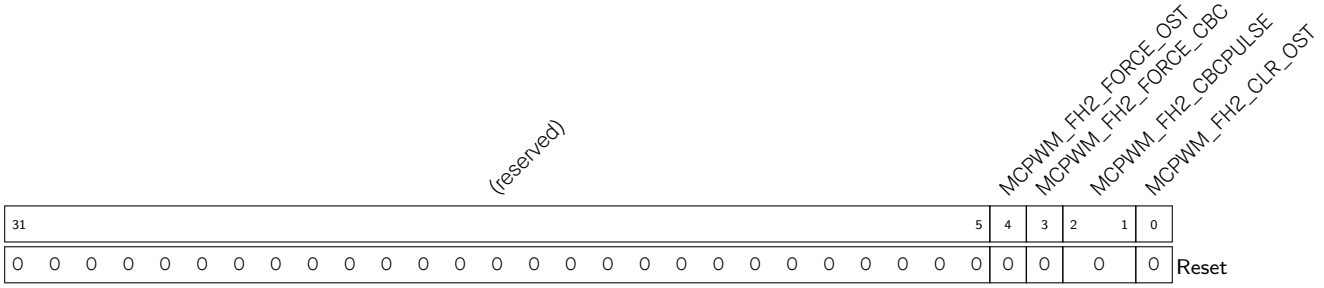
MCPWM_FH2_B_CBC_D 发生故障事件并且定时器递减时，PWM2B 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH2_B_CBC_U 发生故障事件并且定时器递增时，PWM2B 上的逐周期模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH2_B_OST_D 发生故障事件并且定时器递减时，PWM2B 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

MCPWM_FH2_B_OST_U 发生故障事件并且定时器递增时，PWM2B 上的一次性模式操作。0：无；1：强制拉低；2：强制拉高；3：取反。(R/W)

Register 36.57. MCPWM_FH2_CFG1_REG (0x00DC)



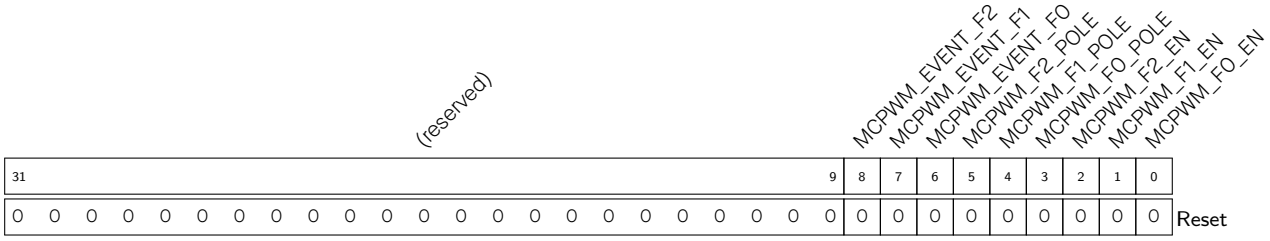
MCPWM_FH2_CLR_OST 上升沿清除正在进行的一次性模式的操作。(R/W)

MCPWM_TZ2_CBCPULSE 设置逐周期模式的更新方式。bit0 为 1：发生 TEZ 事件时；bit1 为 1：发生 TEP 事件时。(R/W)

MCPWM_TZ2_FORCE_CBC 通过软件取反此位的值触发逐周期模式操作。(R/W)

MCPWM_TZ2_FORCE_OST 通过软件取反此位的值触发一次性模式操作。(R/W)

Register 36.58. MCPWM_FAULT_DETECT_REG (0x00E4)



MCPWM_F0_EN 置 1 使能 fault_event0 的生成。(R/W)

MCPWM_F1_EN 置 1 使能 fault_event1 的生成。(R/W)

MCPWM_F2_EN 置 1 使能 fault_event2 的生成。(R/W)

MCPWM_F0_POLE 设置来自 GPIO 矩阵的 FAULT0 信号源触发 fault_event2 时极性。0：低电平触发；1：高电平触发。(R/W)

MCPWM_F1_POLE 设置来自 GPIO 矩阵的 FAULT0 信号源触发 fault_event1 时极性。0：低电平触发；1：高电平触发。(R/W)

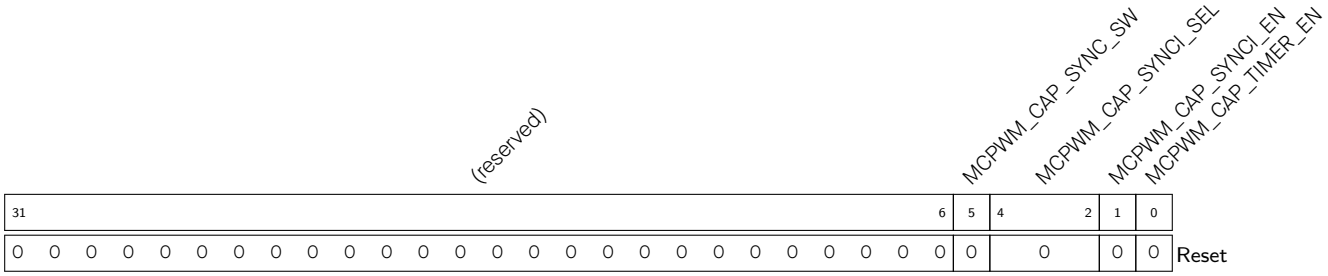
MCPWM_F2_POLE 设置来自 GPIO 矩阵的 FAULT0 信号源触发 fault_event0 时极性。0：低电平触发；1：高电平触发。(R/W)

MCPWM_EVENT_F0 由硬件置 1 和清零。置 1 时，fault_event0 事件持续。(RO)

MCPWM_EVENT_F1 由硬件置 1 和清零。置 1 时，fault_event1 事件持续。(RO)

MCPWM_EVENT_F2 由硬件置 1 和清零。置 1 时，fault_event2 事件持续。(RO)

Register 36.59. MCPWM_CAP_TIMER_CFG_REG (0x00E8)



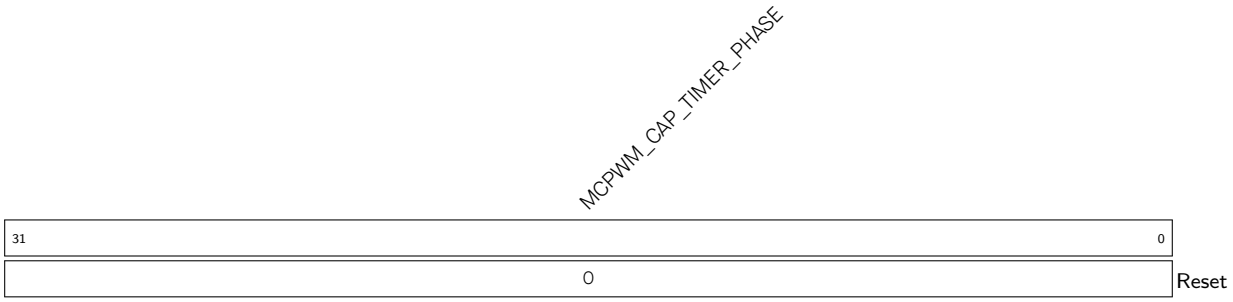
MCPWM_CAP_TIMER_EN 置 1 时，使能捕获定时器在 APB_clk 下的递增。(R/W)

MCPWM_CAP_SYNCI_EN 置 1 时，使能捕获定时器同步。(R/W)

MCPWM_CAP_SYNCI_SEL 选择捕获模块的同步输入。0：无；1：定时器 0 的同步输出；2：定时器 1 的同步输出；3：定时器 2 的同步输出；4：来自 GPIO 矩阵的 SYNC0；5：来自 GPIO 矩阵的 SYNC1；6：来自 GPIO 矩阵的 SYNC2。(R/W)

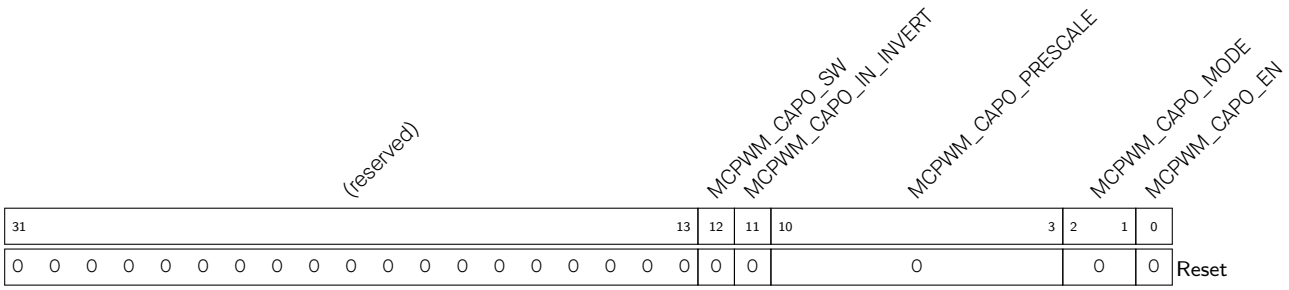
MCPWM_CAP_SYNC_SW 当 reg_cap_synci_en 置位时，该位置 1 同步捕获定时器，捕获定时器中写入相位寄存器的值。(WT)

Register 36.60. MCPWM_CAP_TIMER_PHASE_REG (0x00EC)



MCPWM_CAP_TIMER_PHASE 捕获定时器同步操作的相位值。(R/W)

Register 36.61. MCPWM_CAP_CH0_CFG_REG (0x00F0)



MCPWM_CAP0_EN 置 1 时，使能信道 0 上的捕获。(R/W)

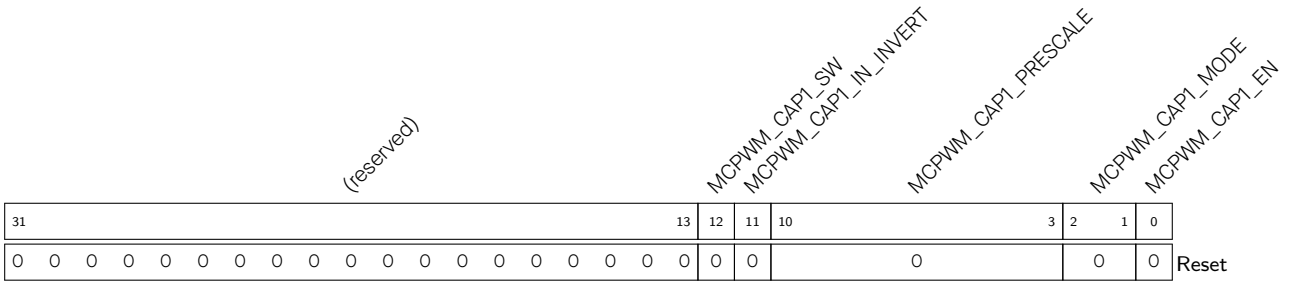
MCPWM_CAP0_MODE 预分频后信道 0 上的捕获边缘。bit0 为 1：使能下降沿捕获；bit1 为 1：使能上升沿捕获。(R/W)

MCPWM_CAP0_PRESCALE CAP0 上升沿的预分频值。预分频值 = PWM_CAP0_PRESCALE + 1。(R/W)

MCPWM_CAP0_IN_INVERT 置 1 时，来自 GPIO 矩阵的 CAP0 在预分频之前反相。(R/W)

MCPWM_CAP0_SW 置 1 触发信道 0 上的软件强制捕获。(WT)

Register 36.62. MCPWM_CAP_CH1_CFG_REG (0x00F4)



MCPWM_CAP1_EN 置 1 时，使能信道 1 上的捕获事件。(R/W)

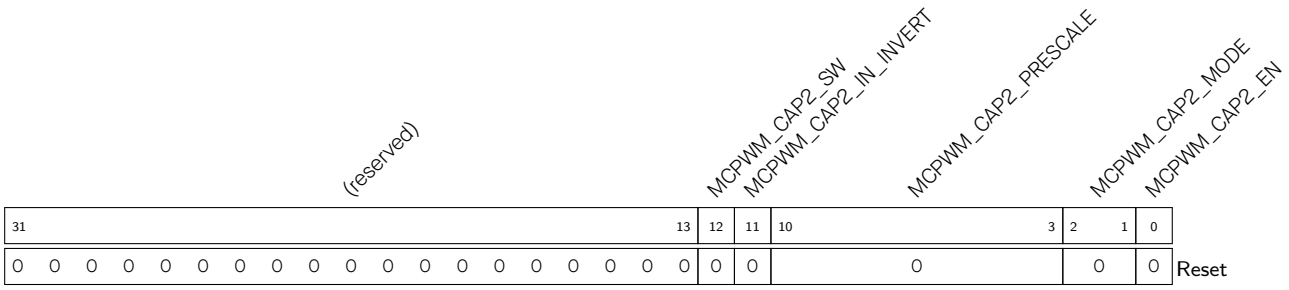
MCPWM_CAP1_MODE 预分频后信道 1 上的捕获沿。bit0 为 1：使能下降沿捕获；bit1 为 1：使能上升沿捕获。(R/W)

MCPWM_CAP1_PRESCALE CAP1 上升沿的预分频值。预分频值 = PWM_CAP1_PRESCALE + 1。(R/W)

MCPWM_CAP1_IN_INVERT 置 1 时，来自于 GPIO 矩阵的 CAP1 在预分频前被反相。(R/W)

MCPWM_CAP1_SW 置 1 时，触发信道 1 上的软件强制捕获事件。(WT)

Register 36.63. MCPWM_CAP_CH2_CFG_REG (0x00F8)



MCPWM_CAP2_EN 置 1 时，使能信道 2 上的捕获。(R/W)

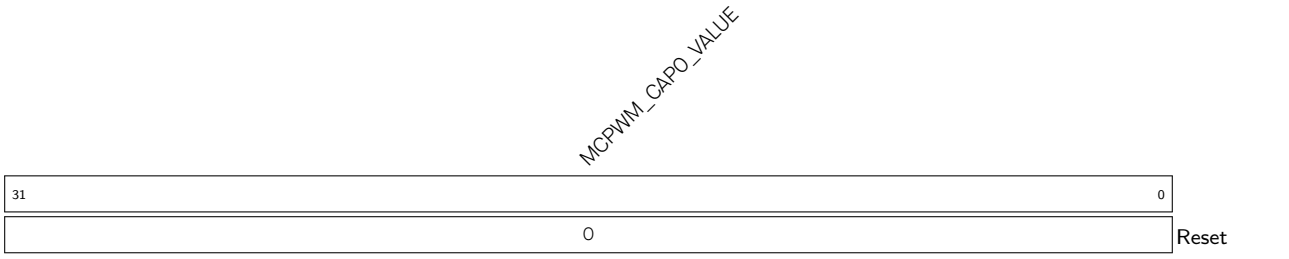
MCPWM_CAP2_MODE 预分频后信道 2 上的捕获沿。bit0 为 1：使能下降沿捕获；bit1 为 1：使能上升沿捕获。(R/W)

MCPWM_CAP2_PRESCALE CAP2 上升沿的预分频值。该预分频值 = PWM_CAP2_PRESCALE + 1. (R/W)

MCPWM_CAP2_IN_INVERT 置 1 时，来自 GPIO 矩阵的 CAP2 在预分频前被反相。(R/W)

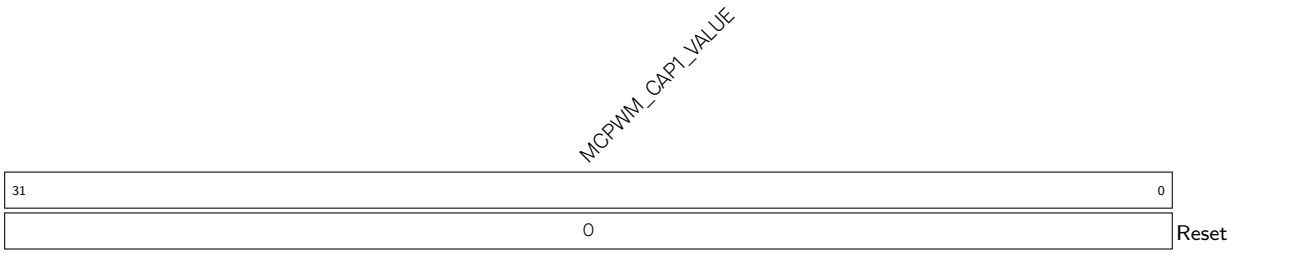
MCPWM_CAP2_SW 置 1 触发信道 2 上的软件强制捕获事件。(WT)

Register 36.64. MCPWM_CAP_CH0_REG (0x00FC)



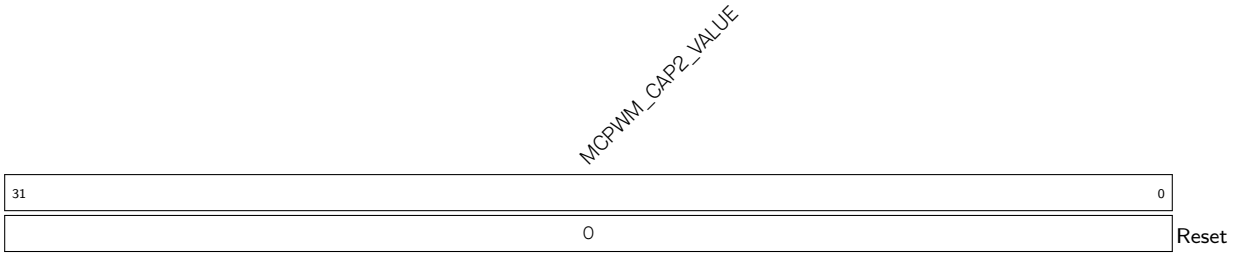
MCPWM_CAP0_VALUE 信道 0 上一次捕获的值。(RO)

Register 36.65. MCPWM_CAP_CH1_REG (0x0100)



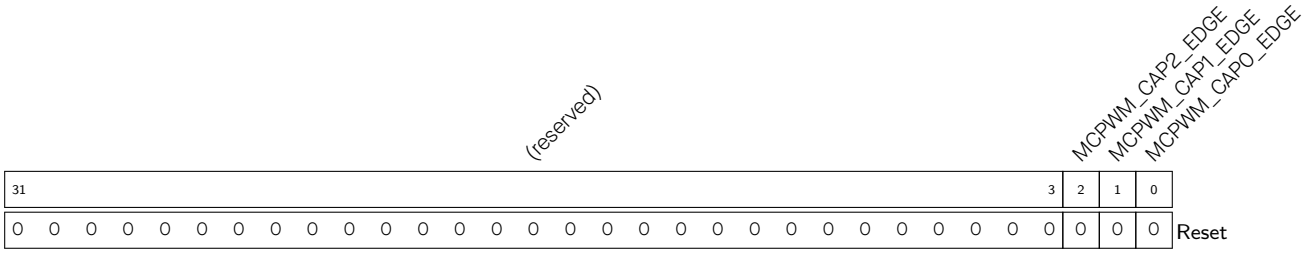
MCPWM_CAP1_VALUE 信道 1 上一次捕获的值。(RO)

Register 36.66. MCPWM_CAP_CH2_REG (0x0104)



MCPWM_CAP2_VALUE 信道 2 上一次捕获的值。(RO)

Register 36.67. MCPWM_CAP_STATUS_REG (0x0108)

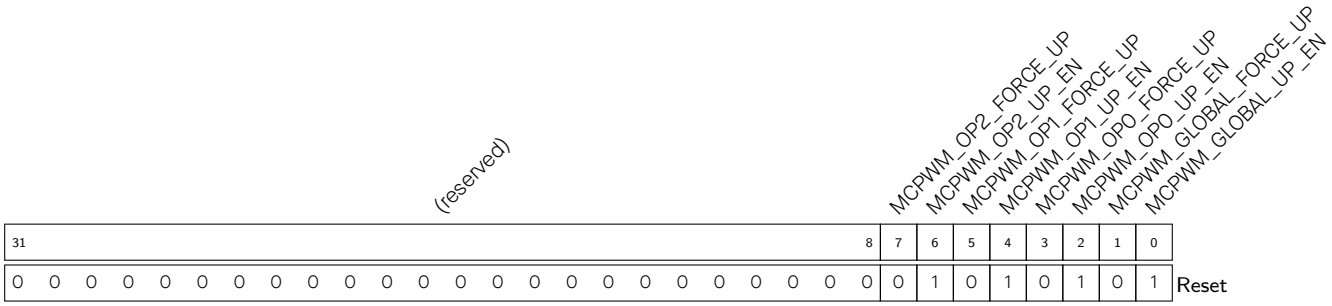


MCPWM_CAP0_EDGE 信道 0 上一次捕获触发事件的边沿。0：上升沿；1：下降沿。(RO)

MCPWM_CAP1_EDGE 信道 1 上一次捕获触发事件的边沿。0：上升沿；1：下降沿。(RO)

MCPWM_CAP2_EDGE 信道 2 上一次捕获触发事件的边沿。0：上升沿；1：下降沿。(RO)

Register 36.68. MCPWM_UPDATE_CFG_REG (0x010C)



MCPWM_GLOBAL_UP_EN MCPWM 模块所有有效寄存器的更新使能位。(R/W)

MCPWM_GLOBAL_FORCE_UP 通过软件取反此位的值将触发 MCPWM 模块所有有效寄存器的强制更新。(R/W)

MCPWM_OP0_UP_EN 此位以及 MCPWM_GLOBAL_UP_EN 置 1 时，使能 PWM 操作器 0 有效寄存器的更新。(R/W)

MCPWM_OP0_FORCE_UP 通过软件取反此位的值将触发 PWM 操作器 0 有效寄存器强制更新。(R/W)

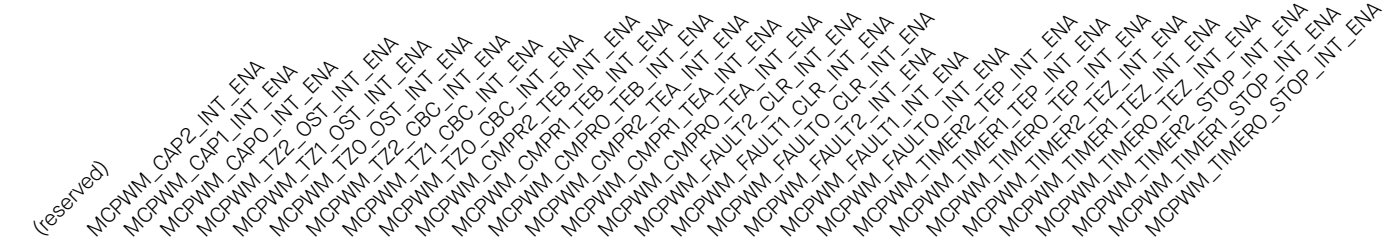
MCPWM_OP1_UP_EN 此位以及 PWM_GLOBAL_UP_EN 置 1 时，使能 PWM 操作器 1 有效寄存器的更新。(R/W)

MCPWM_OP1_FORCE_UP 通过软件取反此位的值将触发 PWM 操作器 1 有效寄存器强制更新。(R/W)

MCPWM_OP2_UP_EN 此位以及 PWM_GLOBAL_UP_EN 置 1 时，使能 PWM 操作器 2 有效寄存器的更新。(R/W)

MCPWM_OP2_FORCE_UP 通过软件取反此位的值将触发 PWM 操作器 2 有效寄存器强制更新。(R/W)

Register 36.69. MCPWM_INT_ENA_REG (0x0110)



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

MCPWM_TIMER0_STOP_INT_ENA 该位用于使能定时器 0 停止时触发的中断。(R/W)

MCPWM_TIMER1_STOP_INT_ENA 该位用于使能定时器 1 停止时触发的中断。(R/W)

MCPWM_TIMER2_STOP_INT_ENA 该位用于使能定时器 2 停止时触发的中断。(R/W)

MCPWM_TIMER0_TEZ_INT_ENA 该位用于使能 PWM 定时器 0 TEZ 事件触发的中断。(R/W)

MCPWM_TIMER1_TEZ_INT_ENA 该位用于使能 PWM 定时器 1 TEZ 事件触发的中断。(R/W)

MCPWM_TIMER2_TEZ_INT_ENA 该位用于使能 PWM 定时器 2 TEZ 事件触发的中断。(R/W)

MCPWM_TIMER0_TEP_INT_ENA 该位用于使能 PWM 定时器 0 TEP 事件触发的中断。(R/W)

MCPWM_TIMER1_TEP_INT_ENA 该位用于使能 PWM 定时器 1 TEP 事件触发的中断。(R/W)

MCPWM_TIMER2_TEP_INT_ENA 该位用于使能 PWM 定时器 2 TEP 事件触发的中断。(R/W)

MCPWM_FAULT0_INT_ENA 该位用于使能 fault_event0 开始时触发的中断。(R/W)

MCPWM_FAULT1_INT_ENA 该位用于使能 fault_event1 开始时触发的中断。(R/W)

MCPWM_FAULT2_INT_ENA 该位用于使能 fault_event2 开始时触发的中断。(R/W)

MCPWM_FAULT0_CLR_INT_ENA 该位用于使能 fault_event0 结束时触发的中断。(R/W)

MCPWM_FAULT1_CLR_INT_ENA 该位用于使能 fault_event1 结束时触发的中断。(R/W)

MCPWM_FAULT2_CLR_INT_ENA 该位用于使能 fault_event2 结束时触发的中断。(R/W)

MCPWM_CMPR0_TEA_INT_ENA 该位用于使能由 PWM 操作器 0 TEA 事件触发的中断。(R/W)

MCPWM_CMPR1_TEA_INT_ENA 该位用于使能由 PWM 操作器 1 TEA 事件触发的中断。(R/W)

MCPWM_CMPR2_TEA_INT_ENA 该位用于使能由 PWM 操作器 2 TEA 事件触发的中断。(R/W)

见下页

Register 36.69. MCPWM_INT_ENA_REG (0x0110)

接上页

MCPWM_CMPR0_TEB_INT_ENA 该位用于使能由 PWM 操作器 0 TEB 事件触发的中断。(R/W)

MCPWM_CMPR1_TEB_INT_ENA 该位用于使能由 PWM 操作器 1 TEB 事件触发的中断。(R/W)

MCPWM_CMPR2_TEB_INT_ENA 该位用于使能由 PWM 操作器 2 TEB 事件触发的中断。(R/W)

MCPWM_TZ0_CBC_INT_ENA 该位用于使能由 PWM0 上的逐周期模式操作触发的中断。(R/W)

MCPWM_TZ1_CBC_INT_ENA 该位用于使能由 PWM1 上的逐周期模式操作触发的中断。(R/W)

MCPWM_TZ2_CBC_INT_ENA 该位用于使能由 PWM2 上的逐周期模式操作触发的中断。(R/W)

MCPWM_TZ0_OST_INT_ENA 该位用于使能由 PWM0 上的一次性模式操作触发的中断。(R/W)

MCPWM_TZ1_OST_INT_ENA 该位用于使能由 PWM1 上的一次性模式操作触发的中断。(R/W)

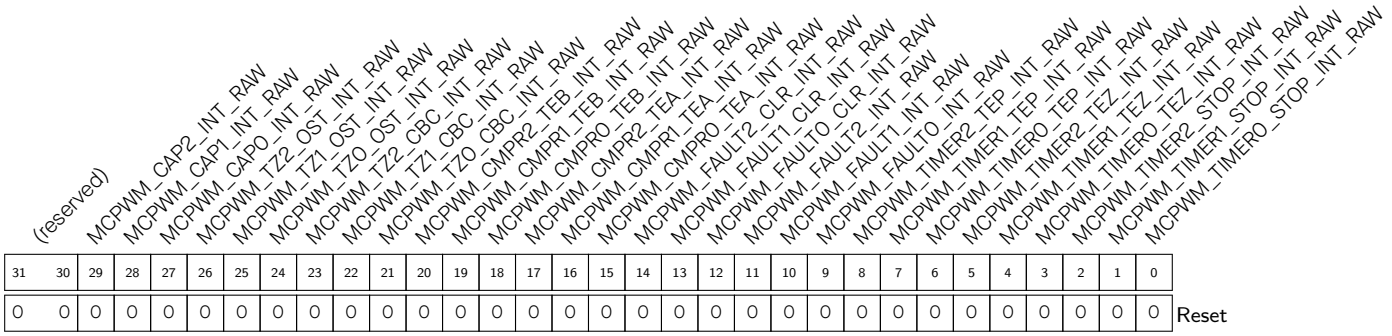
MCPWM_TZ2_OST_INT_ENA 该位用于使能由 PWM2 上的一次性模式操作触发的中断。(R/W)

MCPWM_CAP0_INT_ENA 该位用于使能由信道 0 上的捕获事件触发的中断。(R/W)

MCPWM_CAP1_INT_ENA 该位用于使能由信道 1 上的捕获事件触发的中断。(R/W)

MCPWM_CAP2_INT_ENA 该位用于使能由信道 2 上的捕获事件触发的中断。(R/W)

Register 36.70. MCPWM_INT_RAW_REG (0x0114)



MCPWM_TIMER0_STOP_INT_RAW 定时器 0 停止后触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TIMER1_STOP_INT_RAW 定时器 1 停止后触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TIMER2_STOP_INT_RAW 定时器 2 停止后触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TIMER0_TEZ_INT_RAW PWM 定时器 0 TEP 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TIMER1_TEZ_INT_RAW PWM 定时器 1 TEP 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TIMER2_TEZ_INT_RAW PWM 定时器 2 TEP 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TIMER0_TEP_INT_RAW PWM 定时器 0 TEP 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TIMER1_TEP_INT_RAW PWM 定时器 1 TEP 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TIMER2_TEP_INT_RAW PWM 定时器 2 TEP 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_FAULT0_INT_RAW fault_event0 开始后触发的中断的原始状态位。(R/WTC/SS)

MCPWM_FAULT1_INT_RAW fault_event1 开始后触发的中断的原始状态位。(R/WTC/SS)

MCPWM_FAULT2_INT_RAW fault_event2 开始后触发的中断的原始状态位。(R/WTC/SS)

MCPWM_FAULT0_CLR_INT_RAW fault_event0 结束后触发的中断的原始状态位。(R/WTC/SS)

MCPWM_FAULT1_CLR_INT_RAW fault_event1 结束后触发的中断的原始状态位。(R/WTC/SS)

MCPWM_FAULT2_CLR_INT_RAW fault_event2 结束后触发的中断的原始状态位。(R/WTC/SS)

见下页

Register 36.70. MCPWM_INT_RAW_REG (0x0114)

接上页

MCPWM_CMPR0_TEA_INT_RAW PWM 操作器 0 TEA 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_CMPR1_TEA_INT_RAW PWM 操作器 1 TEA 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_CMPR2_TEA_INT_RAW PWM 操作器 2 TEA 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_CMPR0_TEB_INT_RAW PWM 操作器 0 TEB 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_CMPR1_TEB_INT_RAW PWM 操作器 1 TEB 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_CMPR2_TEB_INT_RAW PWM 操作器 2 TEB 事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TZ0_CBC_INT_RAW 由 PWM0 逐周期操作触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TZ1_CBC_INT_RAW 由 PWM1 逐周期操作触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TZ2_CBC_INT_RAW 由 PWM2 逐周期操作触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TZ0_OST_INT_RAW 由 PWM0 一次性操作触发的中断的原始状态位。(R/WTC/SS)

MCPWM_TZ1_OST_INT_RAW 由 PWM1 一次性操作触发的中断的原始状态位。(R/WTC/SS)

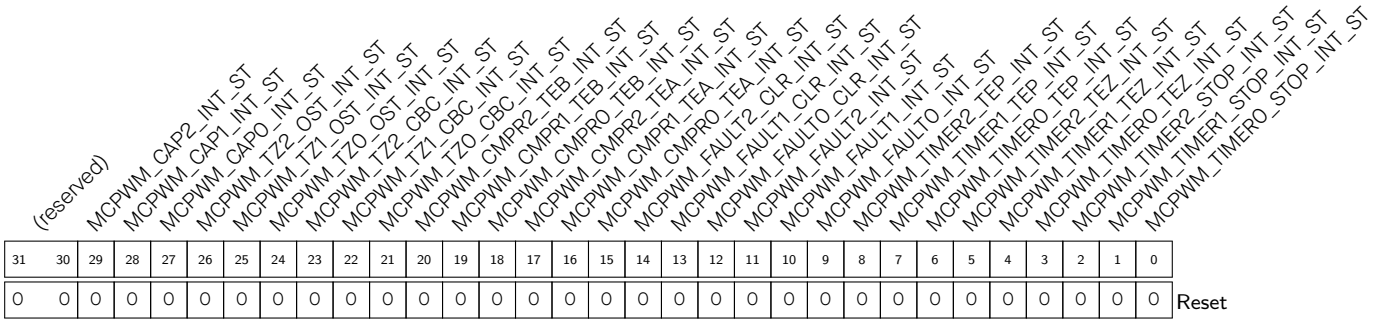
MCPWM_TZ2_OST_INT_RAW 由 PWM2 一次性操作触发的中断的原始状态位。(R/WTC/SS)

MCPWM_CAP0_INT_RAW 由信道 0 上捕获事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_CAP1_INT_RAW 由信道 1 上捕获事件触发的中断的原始状态位。(R/WTC/SS)

MCPWM_CAP2_INT_RAW 由信道 2 上捕获事件触发的中断的原始状态位。(R/WTC/SS)

Register 36.71. MCPWM_INT_ST_REG (0x0118)



MCPWM_TIMER0_STOP_INT_ST 定时器 0 停止后触发的中断的屏蔽状态位。(RO)

MCPWM_TIMER1_STOP_INT_ST 定时器 1 停止后触发的中断的屏蔽状态位。(RO)

MCPWM_TIMER2_STOP_INT_ST 定时器 2 停止后触发的中断的屏蔽状态位。(RO)

MCPWM_TIMER0_TEZ_INT_ST PWM 定时器 0 TEZ 事件触发的中断的屏蔽状态位。(RO)

MCPWM_TIMER1_TEZ_INT_ST PWM 定时器 1 TEZ 事件触发的中断的屏蔽状态位。(RO)

MCPWM_TIMER2_TEZ_INT_ST PWM 定时器 2 TEZ 事件触发的中断的屏蔽状态位。(RO)

MCPWM_TIMER0_TEP_INT_ST PWM 定时器 0 TEP 事件触发的中断的屏蔽状态位。(RO)

MCPWM_TIMER1_TEP_INT_ST PWM 定时器 1 TEP 事件触发的中断的屏蔽状态位。(RO)

MCPWM_TIMER2_TEP_INT_ST PWM 定时器 2 TEP 事件触发的中断的屏蔽状态位。(RO)

MCPWM_FAULT0_INT_ST fault_event0 开始时触发的中断的屏蔽状态位。(RO)

MCPWM_FAULT1_INT_ST fault_event1 开始时触发的中断的屏蔽状态位。(RO)

MCPWM_FAULT2_INT_ST fault_event2 开始时触发的中断的屏蔽状态位。(RO)

MCPWM_FAULT0_CLR_INT_ST fault_event0 结束时触发的中断的屏蔽状态位。(RO)

MCPWM_FAULT1_CLR_INT_ST fault_event1 结束时触发的中断的屏蔽状态位。(RO)

MCPWM_FAULT2_CLR_INT_ST fault_event2 结束时触发的中断的屏蔽状态位。(RO)

MCPWM_CMPR0_TEA_INT_ST PWM 操作器 0 TEA 事件触发的中断的屏蔽状态位。(RO)

MCPWM_CMPR1_TEA_INT_ST PWM 操作器 1 TEA 事件触发的中断的屏蔽状态位。(RO)

MCPWM_CMPR2_TEA_INT_ST PWM 操作器 2 TEA 事件触发的中断的屏蔽状态位。(RO)

见下页

Register 36.71. MCPWM_INT_ST_REG (0x0118)

接上页

MCPWM_CMPR0_TEB_INT_ST PWM 操作器 0 TEB 事件触发的中断的屏蔽状态位。(RO)

MCPWM_CMPR1_TEB_INT_ST PWM 操作器 1 TEB 事件触发的中断的屏蔽状态位。(RO)

MCPWM_CMPR2_TEB_INT_ST PWM 操作器 2 TEB 事件触发的中断的屏蔽状态位。(RO)

MCPWM_TZ0_CBC_INT_ST PWM0 逐周期操作触发的中断的屏蔽状态位。(RO)

MCPWM_TZ1_CBC_INT_ST PWM1 逐周期操作触发的中断的屏蔽状态位。(RO)

MCPWM_TZ2_CBC_INT_ST PWM2 逐周期操作触发的中断的屏蔽状态位。(RO)

MCPWM_TZ0_OST_INT_ST PWM0 一次性操作触发的中断的屏蔽状态位。(RO)

MCPWM_TZ1_OST_INT_ST PWM1 一次性操作触发的中断的屏蔽状态位。(RO)

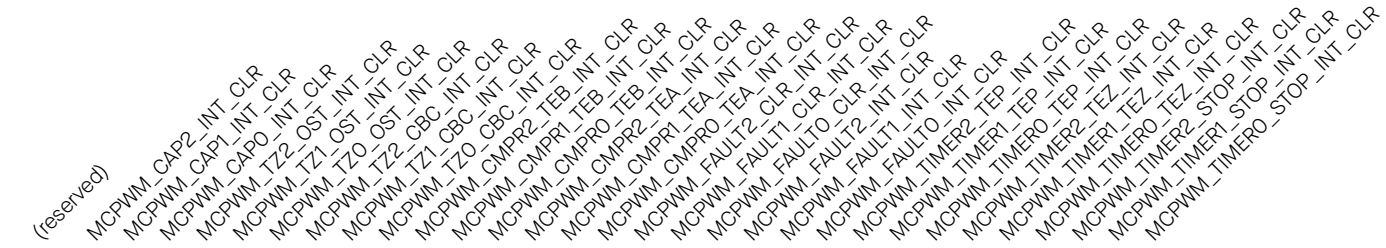
MCPWM_TZ2_OST_INT_ST PWM2 一次性操作触发的中断的屏蔽状态位。(RO)

MCPWM_CAP0_INT_ST 信道 0 上捕获事件触发的中断的屏蔽状态位。(RO)

MCPWM_CAP1_INT_ST 信道 1 上捕获事件触发的中断的屏蔽状态位。(RO)

MCPWM_CAP2_INT_ST 信道 2 上捕获事件触发的中断的屏蔽状态位。(RO)

Register 36.72. MCPWM_INT_CLR_REG (0x011C)



MCPWM_TIMER0_STOP_INT_CLR 置 1 时清除定时器 0 停止后触发的中断。(WT)

MCPWM_TIMER1_STOP_INT_CLR 置 1 时清除定时器 1 停止后触发的中断。(WT)

MCPWM_TIMER2_STOP_INT_CLR 置 1 时清除定时器 2 停止后触发的中断。(WT)

MCPWM_TIMER0_TEZ_INT_CLR 置 1 时清除 PWM 定时器 0 TEZ 事件触发的中断。(WT)

MCPWM_TIMER1_TEZ_INT_CLR 置 1 时清除 PWM 定时器 1 TEZ 事件触发的中断。(WT)

MCPWM_TIMER2_TEZ_INT_CLR 置 1 时清除 PWM 定时器 2 TEZ 事件触发的中断。(WT)

MCPWM_TIMER0_TEP_INT_CLR 置 1 时清除 PWM 定时器 0 TEP 事件触发的中断。(WT)

MCPWM_TIMER1_TEP_INT_CLR 置 1 时清除 PWM 定时器 1 TEP 事件触发的中断。(WT)

MCPWM_TIMER2_TEP_INT_CLR 置 1 时清除 PWM 定时器 2 TEP 事件触发的中断。(WT)

MCPWM_FAULT0_INT_CLR 置位清除 fault_event0 开始时触发的中断。(WT)

MCPWM_FAULT1_INT_CLR 置位清除 fault_event1 开始时触发的中断。(WT)

MCPWM_FAULT2_INT_CLR 置位清除 fault_event2 开始时触发的中断。(WT)

MCPWM_FAULT0_CLR_INT_CLR 置位清除 fault_event0 结束时触发的中断。(WT)

MCPWM_FAULT1_CLR_INT_CLR 置位清除 fault_event1 结束时触发的中断。(WT)

MCPWM_FAULT2_CLR_INT_CLR 置位清除 fault_event2 结束时触发的中断。(WT)

见下页

Register 36.72. MCPWM_INT_CLR_REG (0x011C)

接上页

MCPWM_CMPR0_TEA_INT_CLR 置 1 时清除 PWM 操作器 0 TEA 事件触发的中断。(WT)

MCPWM_CMPR1_TEA_INT_CLR 置 1 时清除 PWM 操作器 1 TEA 事件触发的中断。(WT)

MCPWM_CMPR2_TEA_INT_CLR 置 1 时清除 PWM 操作器 2 TEA 事件触发的中断。(WT)

MCPWM_CMPR0_TEB_INT_CLR 置 1 时清除 PWM 操作器 0 TEB 事件触发的中断。(WT)

MCPWM_CMPR1_TEB_INT_CLR 置 1 时清除 PWM 操作器 1 TEB 事件触发的中断。(WT)

MCPWM_CMPR2_TEB_INT_CLR 置 1 时清除 PWM 操作器 2 TEB 事件触发的中断。(WT)

MCPWM_TZ0_CBC_INT_CLR 置 1 时清除 PWM0 逐周期操作触发的中断。(WT)

MCPWM_TZ1_CBC_INT_CLR 置 1 时清除 PWM1 逐周期操作触发的中断。(WT)

MCPWM_TZ2_CBC_INT_CLR 置 1 时清除 PWM2 逐周期操作触发的中断。(WT)

MCPWM_TZ0_OST_INT_CLR 置 1 时清除 PWM0 一次性操作触发的中断。(WT)

MCPWM_TZ1_OST_INT_CLR 置 1 时清除 PWM1 一次性操作触发的中断。(WT)

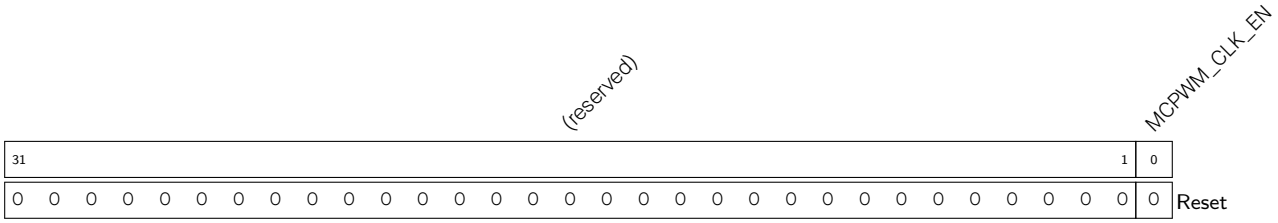
MCPWM_TZ2_OST_INT_CLR 置 1 时清除 PWM2 一次性操作触发的中断。(WT)

MCPWM_CAP0_INT_CLR 置 1 时清除信道 0 上捕获事件触发的中断。(WT)

MCPWM_CAP1_INT_CLR 置 1 时清除信道 1 上捕获事件触发的中断。(WT)

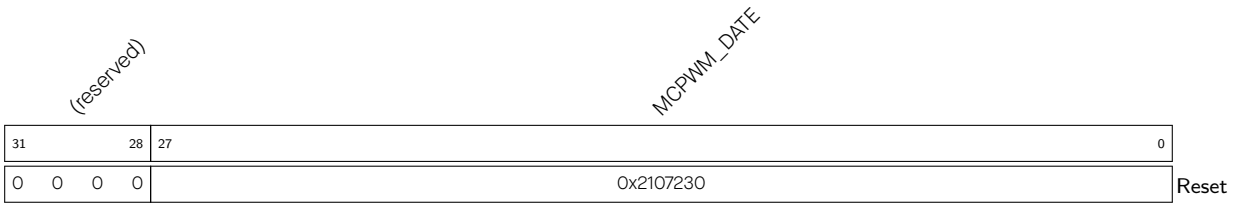
MCPWM_CAP2_INT_CLR 置 1 时清除信道 2 上捕获事件触发的中断。(WT)

Register 36.73. MCPWM_CLK_REG (0x0120)



MCPWM_CLK_EN 强制使能时钟。(R/W)

Register 36.74. MCPWM_VERSION_REG (0x0124)



MCPWM_DATE 版本控制寄存器。(R/W)

第 37 章

红外遥控 (RMT)

37.1 概述

RMT 是一个红外发送和接收控制器，可通过软件加解密多种红外协议。RMT 模块可以实现将模块内置 RAM 中的脉冲编码转换为信号输出，或将模块的输入信号转换为脉冲编码存入 RAM 中。此外，RMT 模块可以选择是否对输出信号进行载波调制，也可以选择是否对输入信号进行滤波和去噪处理。

RMT 共有八个通道，编码为 0 ~ 7，各通道可独立用于发送或接收信号：

- 0 ~ 3 通道专门用于发送信号；
- 4 ~ 7 通道专门用于接收信号。

每个发送通道和接收通道分别有一组功能相同的寄存器。另外，发送通道 3 和接收通道 7 对应的 RAM 支持 DMA 访问，因此还有 DMA 相关的控制和状态寄存器。为了方便叙述，以 *n* 表示各个发送通道，以 *m* 表示各个接收通道。

37.2 特性

- 四个通道支持发送
- 四个通道支持接收
- 可编程配置多个通道同时发送
- RMT 的八个通道共享 384 x 32-bit 的 RAM
- 发送脉冲支持载波调制
- 接收脉冲支持滤波和载波解调
- 乒乓发送模式
- 乒乓接收模式
- 发射器支持持续发送
- 发送通道 3 支持 DMA 访问
- 接收通道 7 支持 DMA 访问

37.3 功能描述

37.3.1 架构

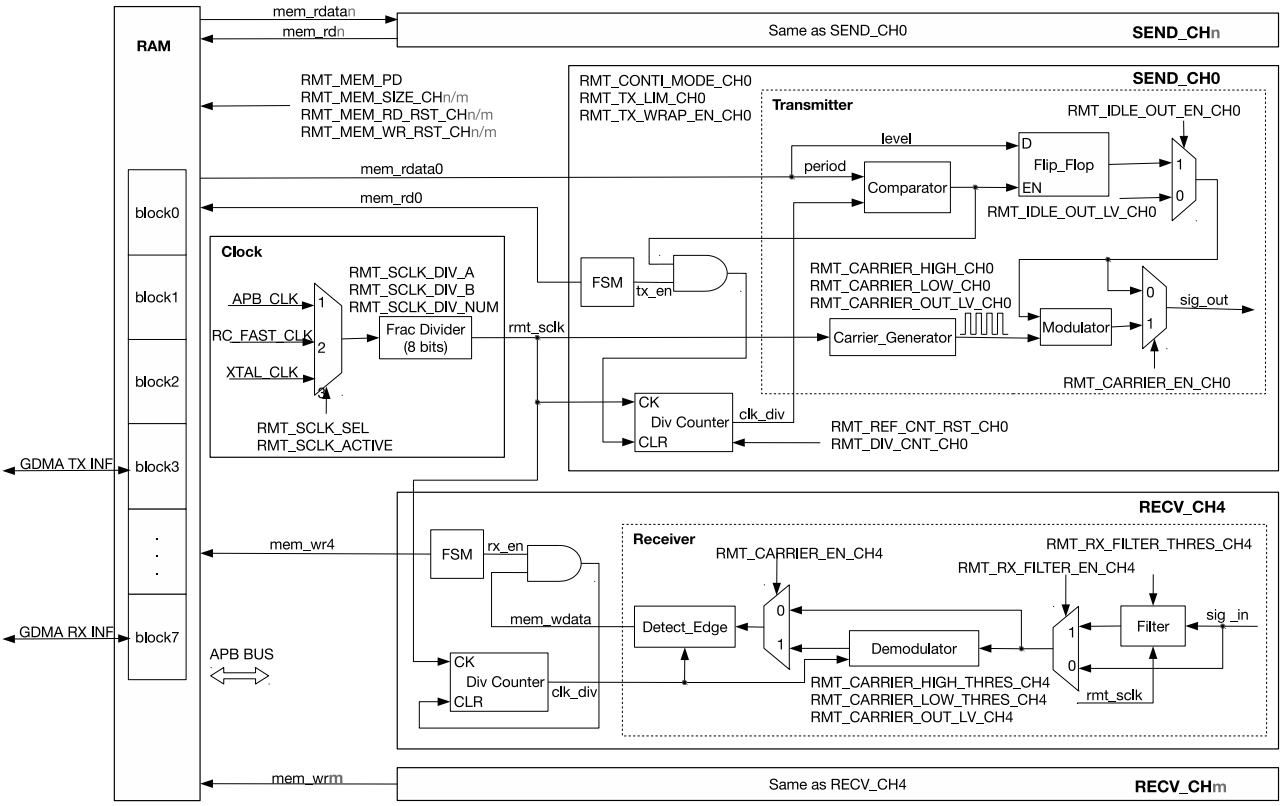


图 37.3-1. RMT 结构框图

如图 37.3-1 所示，每个发送通道 (SEND_CH*n*) 内部各有：

- 一个时钟分频计数器 (Div Counter)
- 一个状态机 (FSM)
- 一个发射器 (Transmitter)

每个接收通道 (RECV_CH*m*) 内部也各有：

- 一个时钟分频计数器 (Div Counter)
- 一个状态机 (FSM)
- 一个接收器 (Receiver)

八个通道共享一块 384 x 32 位的 RAM。

37.3.2 RAM

37.3.2.1 RAM 结构

RAM 中脉冲编码结构如图 37.3-2 所示。每个脉冲编码为 16 位，由 level 与 period 两部分组成。其中 level 表示输入或输出信号的逻辑电平值（0 或 1），period 表示该电平信号持续的时钟（图37.3-1 clk_div）周期数。

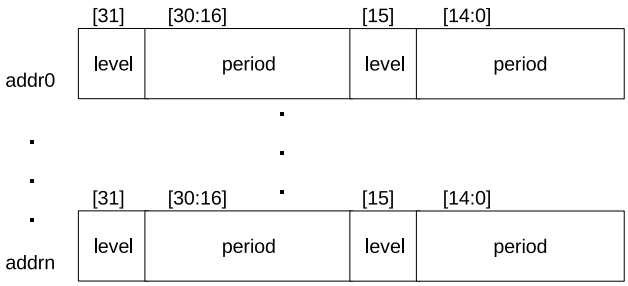


图 37.3-2. RAM 中脉冲编码结构

period 的最小值为 0，period = 0 是一次传输的结束标志。对于非零的 period，即不是结束标志的 period，其值需要满足与 APB 时钟和 RMT 时钟相关的限制条件，见下述公式：

$$5 \times T_{apb\_clk} + 6 \times T_{rmt\_sclk} < period \times T_{clk\_div} \tag{37.1}$$

说明：

RMT 时钟与 APB 时钟需满足以下关系：

$$1.5 \times T_{apb\_clk} < 9 \times T_{rmt\_sclk} \tag{37.2}$$

RMT 可捕获的脉冲宽度，即 $period \times T_{clk\_div}$ 受上述公式以及 rmt_sclk 时钟频率的限制。具体如下：

- 最小脉冲长度应大于 $(5 \times T_{apb\_clk} + 6 \times T_{rmt\_sclk})$；
- 最大脉冲长度应小于或等于（最大 period × 最大 T_{clk_div}），即 $((2^{15} - 1) \times$ 最大 $T_{rmt\_sclk} \times 256)$。

更多有关 rmt_sclk 时钟频率，或 APB_CLK 时钟频率，请参考小节 37.3.3，或章节 7 复位和时钟。

37.3.2.2 RAM 使用说明

RAM 按照 48 x 32-bit 分成八个 block。默认情况下每个通道只能使用一个 block（固定为通道 0 使用 block 0，通道 1 使用 block 1，以此类推）。

当发送通道 *n* 或接收通道 *m* 单次传输的脉冲编码数大于一个 block 时，可以：

- 置位 RMT_MEM_TX/RX_WRAP_EN_CH*n*/*m* 使能乒乓操作；
- 或通过配置 RMT_MEM_SIZE_CH*n*/*m* 寄存器，允许该通道占用多个 block。

当设置 RMT_MEM_SIZE_CH*n*/*m* > 1 时，通道 *n*/*m* 将占用 block (*n*/*m*) ~ block (*n*/*m* + RMT_MEM_SIZE_CH*n*/*m* -1) 的存储空间。通道 *n*/*m* + 1 ~ *n*/*m* + RMT_MEM_SIZE_CH*n*/*m* - 1 因为对应的 RAM block 被占用而无法使用。例如，如果通道 0 配置使用了 block 0 和 block 1，则通道 1 无法使用，通道 2 和通道 3 不受影响。

注意，每个通道使用 RAM 的空间是根据地址从低到高进行映射的，因此通道 0 可以通过配置 RMT_MEM_SIZE_CHO 寄存器来使用通道 1、2、3、...、7 的 RAM 空间，但是通道 7 不能使用通道 0、1、2、...、6 的 RAM 空间。因此 RMT_MEM_SIZE_CH*n* 的最大值不应超过 (8 - *n*)，RMT_MEM_SIZE_CH*m* 的最大值不应超过 (8 - *m*)。

RAM 可被 APB 总线及通道的发射器或接收器访问，为了防止接收通道访问 RAM 和 APB 访问时发生冲突，例如 APB 读 RAM 时，通道向 RAM 发起写操作，用户可以通过配置 RMT_MEM_OWNER_CH*m* 来决定当前 RAM 的使用权。当接收通道发生越权访问时会产生 RMT_MEM_OWNER_ERR_CH*m* 标志信号。

当 RMT 模块不工作时，可以通过配置 RMT_MEM_FORCE_PD 寄存器使 RAM 工作于低功耗模式。

37.3.2.3 RAM 访问方式

APB 总线访问 RAM 有 FIFO 和 NONFIFO（直接地址）两种模式：

- RMT_APB_FIFO_MASK 置 1 时，选择 NONFIFO 模式；
- RMT_APB_FIFO_MASK 置 0 时选择 FIFO 模式。

另外，通道 3 和通道 7 还分别支持 DMA 访问方式。

FIFO 模式

在 FIFO 模式下，APB 通过固定地址 RMT_CH*n*/*m*DATA_REG 向 RAM 写数据或从 RAM 读数据。

NONFIFO 模式

在 NONFIFO 模式下，APB 向连续地址段写入数据或从连续地址段读出数据：

- 发送通道 *n* 对应的写地址段的首地址是：RMT 基地址 + 0x800 + *n*x48，第 2 个数据的访问地址是 RMT 基地址 + 0x800 + *n*x48 + 0x4，以此类推，后面的地址依次加上 0x4。
- 接收通道 *m* 对应的读地址段的首地址是：RMT 基地址 + 0x8c0 + *m*x48，第 2 个数据的访问地址是 RMT 基地址 + 0x8c0 + *m*x48 + 0x4，以此类推，后面的地址依次加上 0x4。

DMA 模式

不同于其它几个发送通道，通道 3 还支持 DMA 访问方式。如果将 RMT_DMA_ACCESS_EN_CH3 置 1，则通道 3 对应的 RAM 仅支持从 DMA 取数。APB 对通道 3 的 FIFO 方式访问将被忽略，同时也不允许 APB 通过 NONFIFO 方式访问通道 3 的 RAM 区域，**否则会出现不可预计的后果**。

为保证传输数据的正确性，需要先启动 DMA，在 DMA 通道已经接收到数据后，再启动 RMT 发送。普通发送模式下，DMA 写满通道 3 的 RAM 后会触发 RMT_APB_MEM_WR_ERR_CH3。使能 RMT_MEM_TX_WRAP_EN_CH3 后，不需要软件进行乒乓操作就可以连续发送大于一个 block 的数据，但要保证通道准备发送数据前 DMA 已经将待发送数据准备好，否则可能会发出与期望不符的数据。

不同于其它几个接收通道，通道 7 还支持 DMA 访问方式。如果将 RMT_DMA_ACCESS_EN_CH7 置 1，通道 7 对应的 RAM 支持向 DMA 发送数据，同时也支持 APB 的 NONFIFO 方式访问。

普通接收模式下，DMA 读到通道 7 RAM 大小的数据后就会触发 RMT_APB_MEM_RD_ERR_CH7，之后收到的数会被丢弃。使能 RMT_MEM_TX_WRAP_EN_CH7 后，不需要软件进行乒乓操作就可以连续接收大于一个 block 的数据。如果通道 7 的 RAM 接收满后 DMA 还没有接收，新收到的数据会替换上一个数据。

> **说明：**
> 当通道 7 接收到结束标志时，会产生 DMA 的 in_suc_eof 中断。如果是 period[14:0] 为 0，则会向 DMA 写两个字节；如果是 period[30:16] 为 0，则会向 DMA 写四个字节。

37.3.3 时钟

用户可以通过配置 RMT_SCLK_SEL 选择 RMT 的时钟源：APB_CLK、RC_FAST_CLK 或 XTAL_CLK，配置 RMT_SCLK_ACTIVE 为高电平来打开 RMT 的时钟。选择后的时钟经过小数分频得到 RMT 的工作时钟（图 37.3-1 rmt_sclk），分频系数为：

$$RMT\_SCLK\_DIV\_NUM + 1 + RMT\_SCLK\_DIV\_A/RMT\_SCLK\_DIV\_B$$

更多信息，请参考章节 7 复位和时钟。RMT_DIV_CNT_CH*n*/*m* 用于配置 RMT 通道内部的时钟分频器的分频系数，除 0 表示 256 分频外，其他分频数等同于寄存器 RMT_DIV_CNT_CH*n*/*m* 的值。时钟分频器可以通过配置 RMT_REF_CNT_RST_CH*n*/*m* 进行复位。时钟分频器的分频时钟可供计数器使用，见图 37.3-1。

37.3.4 发射器

说明：
本小节以及后续小节所述的配置，均需要通过置位 RMT_CONF_UPDATE_CH*n*/*m* 的方式来进行更新，详情见第 37.3.6 小节。

37.3.4.1 普通发送模式

当 RMT_TX_START_CH*n* 置为 1 时，通道 *n* 的发射器开始从通道对应 RAM block 的起始地址，按照地址从低到高依次读取脉冲编码进行发送。当遇到结束标志（period 等于 0）时，发射器将结束发送返回空闲状态，并产生 RMT_CH*n*_TX_END_INT 中断。配置 RMT_TX_STOP_CH*n* 可以使发射器立刻停止发送并进入空闲状态。发射器空闲状态发送的电平由结束标志中的 level 段或者是 RMT_IDLE_OUT_LV_CH*n* 决定。用户可以配置 RMT_IDLE_OUT_EN_CH*n* 来选择这两种方式：

- 清除 RMT_IDLE_OUT_EN_CH*n*，发射器空闲状态发送的电平由结束标志中的 level 段决定；
- 置位 RMT_IDLE_OUT_EN_CH*n*，发射器空闲状态发送的电平由 RMT_IDLE_OUT_LV_CH*n* 决定；

37.3.4.2 乒乓发送模式

当发送的脉冲编码较多时，可通过置位 RMT_MEM_TX_WRAP_EN_CH*n* 使能通道 *n* 的乒乓模式。在乒乓操作模式下，发射器会循环从通道对应的 RAM 区域取出脉冲编码进行发送，直到遇到结束标识为止。例如，当 RMT_MEM_SIZE_CH*n* = 1 时，发射器将从 48 * *n* 地址开始发送，然后对应 RAM 的地址递增。发完 (48 * (*n* + 1) - 1) 地址的数据后，下次继续从 48 * *n* 地址开始递增发送数据，依此类推，遇到结束标识时停止发送。RMT_MEM_SIZE_CH*n* > 1 的情形下，乒乓操作同样适用。

每当发射器发送的脉冲编码数大于等于 RMT_TX_LIM_CH*n* 时，会产生 RMT_CH*n*_TX_THR_EVENT_INT 中断。在乒乓模式下，可以设置 RMT_TX_LIM_CH*n* 为每个通道对应 RAM 空间的一半或几分之一。软件在检测到 RMT_
CH*n*_TX_THR_EVENT_INT 中断之后，可以更新已使用过的 RAM 区域的脉冲编码，从而实现乒乓操作。

说明：
当 RAM 采用 DMA 方式访问时，不需要额外操作就能支持多于一个 RAM 的脉冲编码发送。而采用 APB 方式访问时，需要软件进行乒乓操作。

37.3.4.3 发送加载波

此外，发射器还可以对输出信号进行载波调制，置位 RMT_CARRIER_EN_CH*n* 可以使能该功能。载波的波形可配置。一个载波周期中高电平持续时间为 (RMT_CARRIER_HIGH_CH*n* + 1) 个 rmt_sclk 时钟周期，低电平持续的时间为 (RMT_CARRIER_LOW_CH*n* + 1) 个 rmt_sclk 时钟周期。置位 RMT_CARRIER_OUT_LV_CH*n* 时在输出

信号高电平上加载波信号，清零 RMT_CARRIER_OUT_LV_CH*n* 时在输出信号低电平上加载波信号。同时，在进行载波调制时，载波可以一直加载在输出信号上，也可以仅加载在有效的脉冲编码（RAM 中的数据）上。通过配置 RMT_CARRIER_EFF_EN_CH*n*，可以选择这两种模式。RMT_CARRIER_EFF_EN_CH*n* 设置为 0 时在所有信号上加载波，设置为 1 时在有效信号上加载波。

37.3.4.4 持续发送模式

置位 RMT_TX_CONTI_MODE_CH*n* 可以使能发射器的持续发送功能。置位后，发射器会循环发送 RAM 中的脉冲编码。

持续发送模式下，

- 如果遇到结束标志，则重新从该通道 RAM 中的第一个数据开始发送；
- 如果没有结束标志，会在发送到最后一个数据处回卷，重新开始发送第一个数据。

置位 RMT_TX_LOOP_CNT_EN_CH*n* 后，发射器每遇到一次结束标志，循环发送的次数会加 1。当该次数达到 RMT_TX _LOOP_NUM_CH*n* 设定的值时，会产生 RMT_CH*n*_TX_LOOP_INT 中断。如果置位 RMT_LOOP_STOP_EN_CH*n*，则发送会在产生 RMT_CH*n*_TX_LOOP_INT 中断后立即停止，否则会继续发送。持续发送模式下，如果遇到的结束标志类型是 period[14:0] 为 0，那么这个结束标志前一个数据的 period 需要满足：

$$10 \times T_{apb\_clk} + 19 \times T_{rmt\_sclk} < period \times T_{clk\_div} \tag{37.3}$$

而其它数据的 period，满足非 0 period 关系式 即可。

37.3.4.5 多通道同时发送

RMT 模块支持多通道同时发送，具体配置步骤如下所示：

- 首先配置 RMT_TX_SIM_CH*n* 用于选择同步发送的通道；
- 然后置位 RMT_TX_SIM_EN 使能发射器多个通道同步发送的功能；
- 最后将所选同步发送通道的 RMT_TX_START_CH*n* 置为 1。

当最后一个通道完成配置时，此时多个通道会同时启动发送。另外，RMT 模块还支持通道 0 ~ 2 的 RAM 采用 APB 访问和通道 3 的 RAM 采用 DMA 方式访问下的多通道同时发送。

37.3.5 接收器

37.3.5.1 普通接收模式

RMT_RX_EN_CH*m* 置为 1 时接收器开始工作，置为 0 时会停止接收。接收器会从信号的第一个跳变沿开始计数，并检测信号电平及其持续的时钟周期数，将其按照脉冲编码的格式存入 RAM 中。当信号在一个电平下持续的时钟周期数超过 RMT_IDLE_THRES_CH*m* 时，接收器结束接收过程，返回空闲状态，并产生 RMT_CH*m*_RX_END_INT 中断。RMT_IDLE_THRES_CH*m* 的值需要大于输入信号的高电平或低电平的最大时钟周期，否则接收器会将该信号误判为空闲信号而进入空闲状态。当接收数据存满了接收通道设置的 RAM 空间时，会停止接收，并产生 RMT_CH*m*_ERR_INT 中断（由 RAM 满事件触发）。

37.3.5.2 乒乓接收模式

当接收的脉冲编码较多时，可通过置位 RMT_MEM_RX_WRAP_EN_CH*m* 使能通道 *m* 的乒乓模式。当 RAM 采用 DMA 方式访问时，不需要额外操作就能支持多于一个 RAM 的脉冲编码接收。而采用 APB 方式访问时，需要软件进行乒乓操作。在乒乓操作模式下，接收器会将接收到的脉冲编码循环存入通道对应的 RAM 区域，当信号在一个电平下持续的时钟周期数超过 RMT_IDLE_THRES_CH*m* 时，接收器结束接收过程，返回空闲状态，并产生 RMT_CH*m*_RX_END_INT 中断。例如，当 RMT_MEM_SIZE_CH*m* = 1 时，接收器将从 48 * *m* 地址开始接收，然后对应 RAM 的地址递增。收完 (48 * (*m* +1) - 1) 地址的数据后，下次继续从 48 * *m* 地址开始递增接收数据，以此类推，遇到信号在一个电平下持续的时钟周期数超过 RMT_IDLE_THRES_CH*m* 时停止接收。RMT_MEM_SIZE_CH*m* > 1 的情形下，乒乓操作同样适用。

每当接收器接收的脉冲编码数大于或等于 RMT_CH*m*_RX_LIM_REG 时，会产生 RMT_CH*m*_RX_THR_EVENT_INT 中断。在乒乓模式下，可以设置 RMT_CH*m*_RX_LIM_REG 为每个通道对应 RAM 空间的一半或几分之一。软件在检测到 RMT_CH*m*_RX_THR_EVENT_INT 中断之后，可以回收已使用过的 RAM 区域的脉冲编码，从而实现乒乓操作。

37.3.5.3 接收滤波

置位 RMT_RX_FILTER_EN_CH*m* 可使能通道 *m* 的接收器对输入信号进行滤波。滤波器的功能为连续采样输入信号，如果输入信号在连续 RMT_RX_FILTER _THRES_CH*m* 个 rmt_sclk 时钟周期内保持不变，则输入信号有效，否则输入信号无效。只有有效的输入信号才能通过滤波器。因此，滤波器会滤除脉冲宽度小于 RMT_RX_FILTER_THR
ES_CH*m* 个 rmt_sclk 时钟周期的线路毛刺。

37.3.5.4 接收去载波

此外，接收器还可以对输入信号或滤波后的信号进行去载波调制，置位 RMT_CARRIER_EN_CH*m* 可以使能该功能。去载波可以对高电平调制或低电平调制的载波进行滤除：

- 置位 RMT_CARRIER_OUT_LV_CH*m*，配置滤除高电平载波；
- 清零 RMT_CARRIER_OUT_LV_CH*m* 配置滤除低电平载波。

配置 RMT_CARRIER_HIGH_THRES_CH*m* 和 RMT_CARRIER_LOW_THRES_CH*m* 可设置去载波的高电平阈值和低电平阈值。当信号的高电平持续时间小于 RMT_CARRIER_HIGH_THRES_CH*m* 个 clk_div 分频时钟周期，或者信号的低电平持续时间小于 RMT_CARRIER _LOW_THRES_CH*m* 个 clk_div 分频时钟周期，则信号会被认为是载波而被滤除。

37.3.6 配置参数更新

RMT 的配置寄存器均需要配置各自通道的 RMT_CONF_UPDATE_CH*n*/*m* 位来更新进入各自通道，更新方法是向 RMT_CONF_UPDATE_CH*n*/*m* 写入高电平。发送通道和接收通道需要通过这种方法更新的配置参数见表 37.3-1。

表 37.3-1. 更新配置参数

配置寄存器	配置参数
发送通道	
	RMT_CARRIER_OUT_LV_CH*n*
	RMT_CARRIER_EN_CH*n*

见下页

表 37.3-1 – 接上页

配置寄存器	配置参数
	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_TX_SIM_REG	RMT_TX_SIM_EN
接收通道	
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*

37.4 中断

- RMT_CH*n*/*m*_ERR_INT：当通道 *n*/*m* 发生读写数据不正确，或内存空满错误时，即触发此中断。
- RMT_CH*n*_TX_THR_EVENT_INT：发射器每发送 RMT_CH*n*_TX_LIM_REG 的数据，即触发一次此中断。
- RMT_CH*m*_RX_THR_EVENT_INT：接收器每接收 RMT_CH*m*_RX_LIM_REG 的数据，即触发一次此中断。
- RMT_CH*n*_TX_END_INT：当发射器停止发送信号时，即触发此中断。
- RMT_CH*m*_RX_END_INT：当接收器停止接收信号时，即触发此中断。
- RMT_CH*n*_TX_LOOP_INT：发射器处于循环发送模式时，当循环次数达到 RMT_TX_LOOP_NUM_CH*n* 的值后，会产生此中断。
- RMT_CH3_DMA_ACCESS_FAIL_INT：当 DMA 向通道 3 RAM 区域写入数据个数减去通道 3 实际发送个数超过通道 3 RAM 大小，且 DMA 又写入数据，会产生此中断。
- RMT_CH7_DMA_ACCESS_FAIL_INT：当通道 7 RAM 接收数据个数减去 DMA 取走的数据超过通道 7 RAM 大小，且通道 7 又接收到数据，会产生此中断。

37.5 寄存器列表

本小节的所有地址均为相对于 RMT 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
FIFO 读/写寄存器			
RMT_CH0DATA_REG	通道 0 通过 APB FIFO 进行读写操作时用到的数据寄存器	0x0000	RO
RMT_CH1DATA_REG	通道 1 通过 APB FIFO 进行读写操作时用到的数据寄存器	0x0004	RO
RMT_CH2DATA_REG	通道 2 通过 APB FIFO 进行读写操作时用到的数据寄存器	0x0008	RO
RMT_CH3DATA_REG	通道 3 通过 APB FIFO 进行读写操作时用到的数据寄存器	0x000C	RO
RMT_CH4DATA_REG	通道 4 通过 APB FIFO 进行读写操作时用到的数据寄存器	0x0010	RO
RMT_CH5DATA_REG	通道 5 通过 APB FIFO 进行读写操作时用到的数据寄存器	0x0014	RO
RMT_CH6DATA_REG	通道 6 通过 APB FIFO 进行读写操作时用到的数据寄存器	0x0018	RO
RMT_CH7DATA_REG	通道 7 通过 APB FIFO 进行读写操作时用到的数据寄存器	0x001C	RO
配置寄存器			
RMT_CH0CONF0_REG	通道 0 的配置寄存器 0	0x0020	varies
RMT_CH1CONF0_REG	通道 1 的配置寄存器 0	0x0024	varies
RMT_CH2CONF0_REG	通道 2 的配置寄存器 0	0x0028	varies
RMT_CH3CONF0_REG	通道 3 的配置寄存器 0	0x002C	varies
RMT_CH4CONF0_REG	通道 4 的配置寄存器 0	0x0030	R/W
RMT_CH4CONF1_REG	通道 4 的配置寄存器 1	0x0034	varies
RMT_CH5CONF0_REG	通道 5 的配置寄存器 0	0x0038	R/W
RMT_CH5CONF1_REG	通道 5 的配置寄存器 1	0x003C	varies
RMT_CH6CONF0_REG	通道 6 的配置寄存器 0	0x0040	R/W
RMT_CH6CONF1_REG	通道 6 的配置寄存器 1	0x0044	varies
RMT_CH7CONF0_REG	通道 7 的配置寄存器 0	0x0048	R/W
RMT_CH7CONF1_REG	通道 7 的配置寄存器 1	0x004C	varies
RMT_CH4_RX_CARRIER_RM_REG	通道 4 的去载波寄存器	0x0090	R/W
RMT_CH5_RX_CARRIER_RM_REG	通道 5 的去载波寄存器	0x0094	R/W
RMT_CH6_RX_CARRIER_RM_REG	通道 6 的去载波寄存器	0x0098	R/W
RMT_CH7_RX_CARRIER_RM_REG	通道 7 的去载波寄存器	0x009C	R/W
RMT_SYS_CONF_REG	RMT APB 配置寄存器	0x00C0	R/W
RMT_REF_CNT_RST_REG	RMT 时钟分频寄存器复位寄存器	0x00C8	WT
状态寄存器			

名称	描述	地址	访问
RMT_CH0STATUS_REG	通道 0 的状态寄存器	0x0050	RO
RMT_CH1STATUS_REG	通道 1 的状态寄存器	0x0054	RO
RMT_CH2STATUS_REG	通道 2 的状态寄存器	0x0058	RO
RMT_CH3STATUS_REG	通道 3 的状态寄存器	0x005C	RO
RMT_CH4STATUS_REG	通道 4 的状态寄存器	0x0060	RO
RMT_CH5STATUS_REG	通道 5 的状态寄存器	0x0064	RO
RMT_CH6STATUS_REG	通道 6 的状态寄存器	0x0068	RO
RMT_CH7STATUS_REG	通道 7 的状态寄存器	0x006C	RO
中断寄存器			
RMT_INT_RAW_REG	原始中断状态寄存器	0x0070	R/WTC/SS
RMT_INT_ST_REG	屏蔽中断状态寄存器	0x0074	RO
RMT_INT_ENA_REG	中断使能寄存器	0x0078	R/W
RMT_INT_CLR_REG	中断清除寄存器	0x007C	WT
载波占空比寄存器			
RMT_CH0CARRIER_DUTY_REG	通道 0 的占空比配置寄存器	0x0080	R/W
RMT_CH1CARRIER_DUTY_REG	通道 1 的占空比配置寄存器	0x0084	R/W
RMT_CH2CARRIER_DUTY_REG	通道 2 的占空比配置寄存器	0x0088	R/W
RMT_CH3CARRIER_DUTY_REG	通道 3 的占空比配置寄存器	0x008C	R/W
TX 事件配置寄存器			
RMT_CH0_TX_LIM_REG	通道 0 的 TX 事件配置寄存器	0x00A0	varies
RMT_CH1_TX_LIM_REG	通道 1 的 TX 事件配置寄存器	0x00A4	varies
RMT_CH2_TX_LIM_REG	通道 2 的 TX 事件配置寄存器	0x00A8	varies
RMT_CH3_TX_LIM_REG	通道 3 的 TX 事件配置寄存器	0x00AC	varies
RMT_TX_SIM_REG	RMT TX 同步发送寄存器	0x00C4	R/W
RX 事件配置寄存器			
RMT_CH4_RX_LIM_REG	通道 4 的 RX 事件配置寄存器	0x00B0	R/W
RMT_CH5_RX_LIM_REG	通道 5 的 RX 事件配置寄存器	0x00B4	R/W
RMT_CH6_RX_LIM_REG	通道 6 的 RX 事件配置寄存器	0x00B8	R/W
RMT_CH7_RX_LIM_REG	通道 7 的 RX 事件配置寄存器	0x00BC	R/W
版本寄存器			
RMT_DATE_REG	版本控制寄存器	0x00CC	R/W

37.6 寄存器

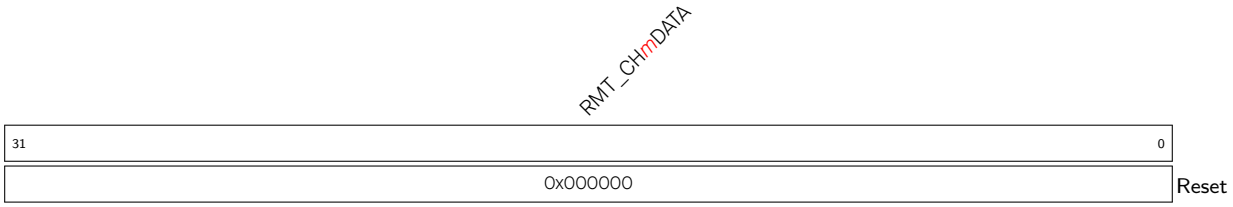
本小节的所有地址均为相对于 RMT 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 37.1. RMT_CH*n*DATA_REG (*n*: 0-3) (0x0000+0x4**n*)



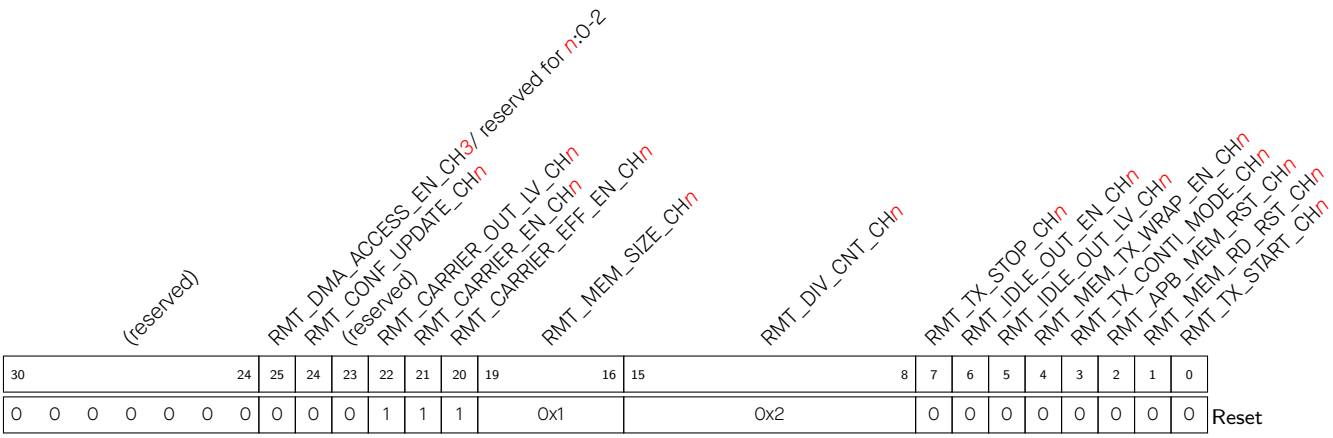
RMT_CH*n*DATA 通道 *n* 通过 APB FIFO 进行读写操作时用到的数据寄存器。(RO)

Register 37.2. RMT_CH*m*DATA_REG (*m* = 4, 5, 6, 7) (0x0010, 0x0014, 0x0018, 0x001C)



RMT_CH*m*DATA 通道 *m* 通过 APB FIFO 进行读写操作时用到的数据寄存器。(RO)

Register 37.3. RMT_CHnCONF0_REG (n: 0-3) (0x0020+0x4*n)



RMT_TX_START_CHn 置位此位，通道 n 开始发送数据。(WT)

RMT_MEM_RD_RST_CHn 置位此位，则通道 n 通过发射器访问的 RAM 读地址将被复位。(WT)

RMT_APB_MEM_RST_CHn 置位此位，则通道 n 通过 APB FIFO 访问的读/写 RAM 地址将被复位。(WT)

RMT_TX_CONTI_MODE_CHn 置位此位，使能通道 n 的持续发送模式。在这种模式下，发射器从第一个数据开始发送，如果遇到结束标志，则再次发送第一个数据；如果没有遇到结束标志，发送完最后一个数据后，回卷到第一个数据再次继续发送。(R/W)

RMT_MEM_TX_WRAP_EN_CHn 置位此位，使能通道 n 的乒乓发送模式。在这种模式下，如果待发送的数据长度大于该通道的 RAM Block 长度，则发射器将继续从第一个数据开始循环发送。(R/W)

RMT_IDLE_OUT_LV_CHn 配置通道 n 在空闲模式下的输出信号电平。(R/W)

RMT_IDLE_OUT_EN_CHn 通道 n 在空闲状态下的输出使能控制位。0：输出由结束标志的电平决定。1：由 RMT_IDLE_OUT_LV_CHn 决定。(R/W)

RMT_TX_STOP_CHn 置位此位，则通道 n 的发射器停止发送数据。(R/W/SC)

RMT_DIV_CNT_CHn 配置通道 n 的时钟分频器。(R/W)

RMT_MEM_SIZE_CHn 配置通道 n 可用的最大 RAM Block 数量。(R/W)

RMT_CARRIER_EFF_EN_CHn 1：配置通道 n 仅在发送数据状态下对输出信号载波调制；0：配置通道 n 对发送数据状态和空闲状态均加载载波。仅在 RMT_CARRIER_EN_CHn 为 1 时有效。(R/W)

RMT_CARRIER_EN_CHn 通道 n 的载波调制使能控制位。1：对输出信号进行载波调制；0：禁止对输出信号进行载波调制。(R/W)

RMT_CARRIER_OUT_LV_CHn 配置通道 n 的载波调制方式。(R/W)

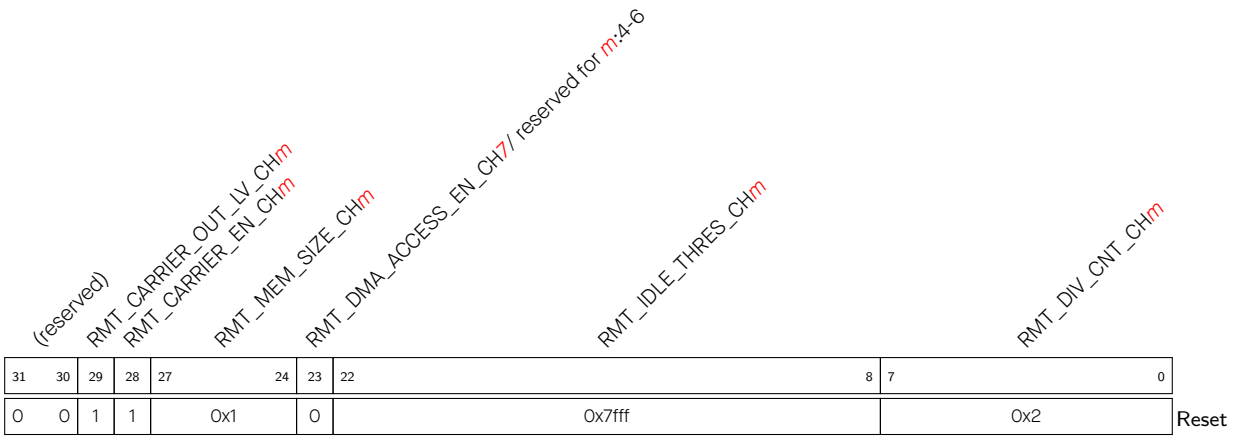
1'h0：载波加载在低电平上；

1'h1：载波加载在高电平上。

RMT_CONF_UPDATE_CHn 通道 n 的同步位。(WT)

RMT_DMA_ACCESS_EN_CH3 (Reserved for channel 0 - 2) 置位此位，使能通道 3 的 DMA 访问方式。(R/W)

Register 37.4. RMT_CH*m*CONF0_REG (*m* = 4, 5, 6, 7) (0x0030, 0x0038, 0x0040, 0x0048)



RMT_DIV_CNT_CH*m* 配置通道 *m* 的时钟分频器。(R/W)

RMT_IDLE_THRES_CH*m* 配置通道 *m* 的接收阈值。接收器长时间检测不到信号变化，且持续的时间大于 RMT_IDLE_THRES_CH*m* 的值，则接收器停止接收过程。(R/W)

RMT_DMA_ACCESS_EN_CH7 (Reserved for channel 4 - 6) 置位此位，使能通道 7 的 DMA 访问方式。(R/W)

RMT_MEM_SIZE_CH*m* 配置通道 *m* 可用的最大 RAM Block 数量。(R/W)

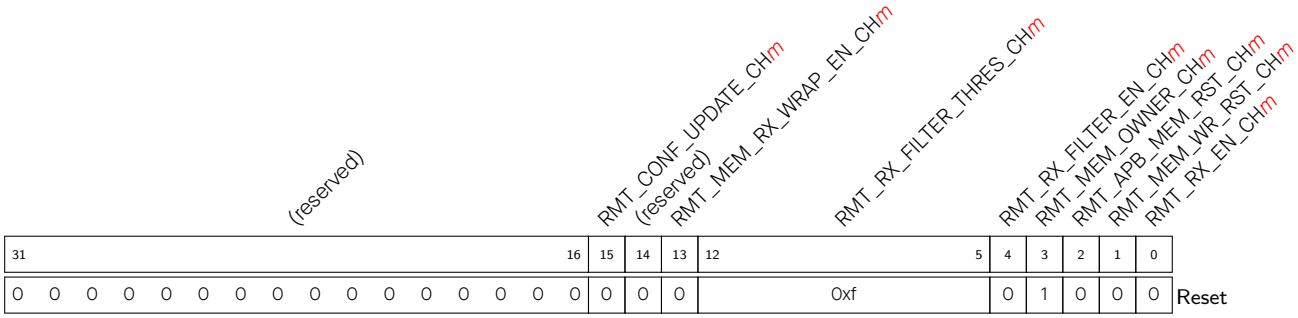
RMT_CARRIER_EN_CH*m* 通道 *m* 的去载波使能控制位。1：对输入信号进行去载波；0：禁止对输入信号进行去载波。(R/W)

RMT_CARRIER_OUT_LV_CH*m* 配置通道 *m* 的去载波方式。(R/W)

1'h0：滤除低电平载波；

1'h1：滤除高电平载波。

Register 37.5. RMT_CH*m*CONF1_REG (*m* = 4, 5, 6, 7) (0x0034, 0x003C, 0x0044, 0x004C)



RMT_RX_EN_CH*m* 置位此位，使能通道 *m* 的接收器，接收器开始接收数据。(R/W)

RMT_MEM_WR_RST_CH*m* 置位此位，则通道 *m* 通过接收器访问的 RAM 写地址将被复位。(WT)

RMT_APB_MEM_RST_CH*m* 置位此位，则通道 *m* 通过 APB FIFO 访问的读/写 RAM 地址将被复位。(WT)

RMT_MEM_OWNER_CH*m* 标志通道 *m* 的 RAM 使用权。(R/W/SC)

1'h1：接收器有权使用该通道的 RAM；

1'h0：APB 总线有权使用该通道的 RAM。

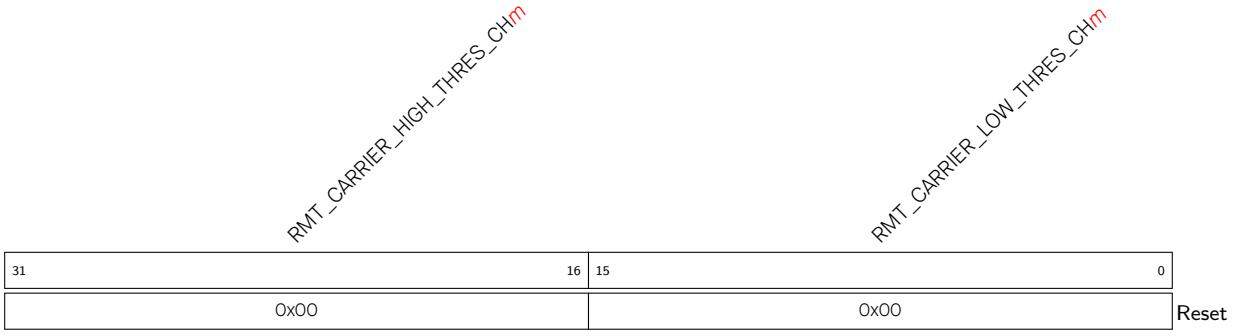
RMT_RX_FILTER_EN_CH*m* 通道 *m* 的接收器滤波功能使能位。(R/W)

RMT_RX_FILTER_THRES_CH*m* 接收模式下，忽略宽度小于 RMT_RX_FILTER_THRES_CH*m* 个 rmt_sclk 周期的输入脉冲。(R/W)

RMT_MEM_RX_WRAP_EN_CH*m* 置位此位，使能通道 *m* 的乒乓接收模式。在这种模式下，如果待接收的数据长度大于该通道的 RAM Block 长度，则接收器将继续把接收数据存入 RAM 的第一个地址，依次循环。(R/W)

RMT_CONF_UPDATE_CH*m* 通道 *m* 的同步位。(WT)

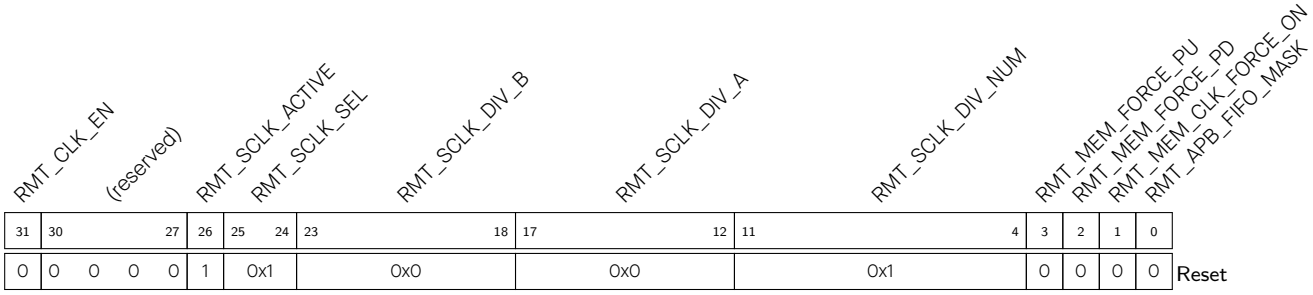
Register 37.6. RMT_CH*m*_RX_CARRIER_RM_REG (*m* = 4, 5, 6, 7) (0x0090, 0x0094, 0x0098, 0x009C)



RMT_CARRIER_LOW_THRES_CH*m* 载波调制模式下，通道 *m* 低电平周期为 RMT_CARRIER_LOW_THRES_CH*m* + 1。(R/W)

RMT_CARRIER_HIGH_THRES_CH*m* 载波调制模式下，通道 *m* 高电平周期为 RMT_CARRIER_HIGH_THRES_CH*m* + 1。(R/W)

Register 37.7. RMT_SYS_CONF_REG (0x00C0)



RMT_APB_FIFO_MASK 1'h1：直接访问 RAM（NOFIFO 模式）；1'h0：通过 FIFO 访问 RAM（FIFO 模式）。(R/W)

RMT_MEM_CLK_FORCE_ON 置位此位，使能 RMT 的 RAM 时钟。(R/W)

RMT_MEM_FORCE_PD 置位此位，关闭 RMT RAM。(R/W)

RMT_MEM_FORCE_PU 1：禁用 RMT RAM 的 Light-sleep 低功耗模式；0：RMT 处于 Light-sleep 模式时，关闭 RMT RAM。(R/W)

RMT_SCLK_DIV_NUM 小数分频器的整数部分。(R/W)

RMT_SCLK_DIV_A 小数分频器的分子。(R/W)

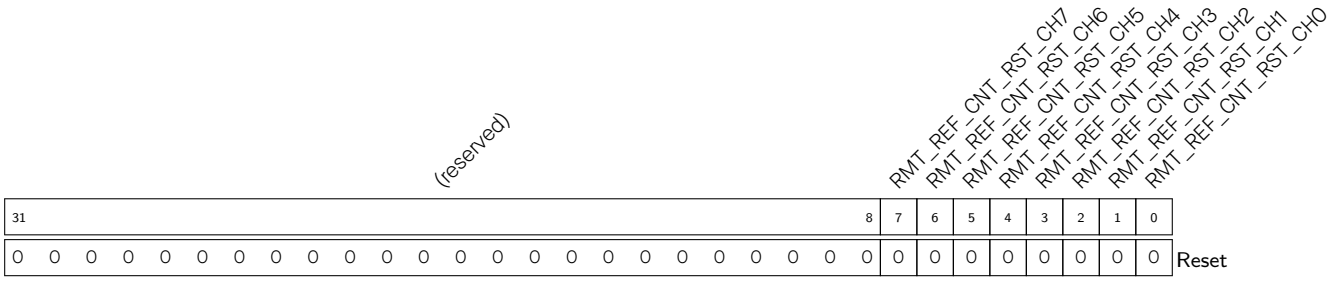
RMT_SCLK_DIV_B 小数分频器的分母。(R/W)

RMT_SCLK_SEL 设置 rmt_sclk 的时钟源：1：APB_CLK；2：RC_FAST_CLK；3：XTAL_CLK。(R/W)

RMT_SCLK_ACTIVE rmt_sclk 控制开关。(R/W)

RMT_CLK_EN RMT 寄存器的时钟门控使能位。1：打开寄存器的驱动时钟；0：关闭寄存器的驱动时钟。(R/W)

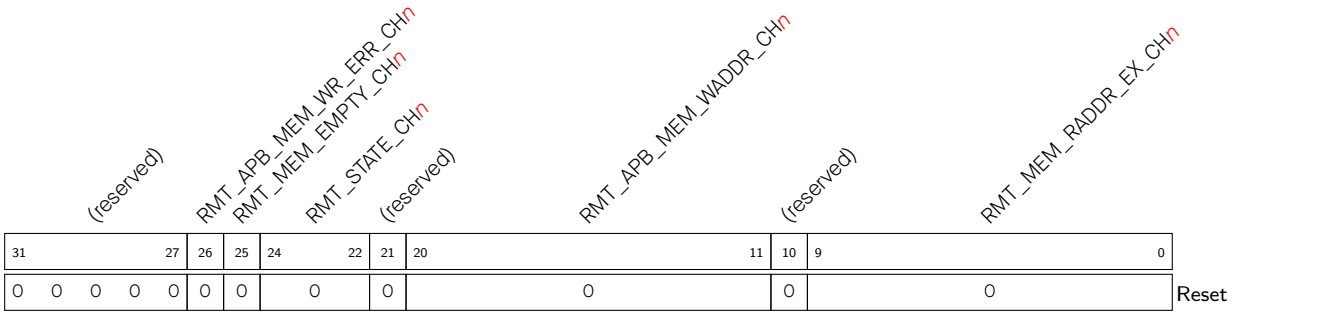
Register 37.8. RMT_REF_CNT_RST_REG (0x00C8)



RMT_REF_CNT_RST_CH*n*(*n* = 0, 1, 2, 3) 复位通道 *n* 的时钟分频器。(WT)

RMT_REF_CNT_RST_CH*m* (*m* = 4, 5, 6, 7) 复位通道 *m* 的时钟分频器。(WT)

Register 37.9. RMT_CH*n*STATUS_REG (*n*: 0-3) (0x0050+0x4**n*)



RMT_MEM_RADDR_EX_CH*n* 记录通道 *n* 发射器使用 RAM 时的地址偏移量。(RO)

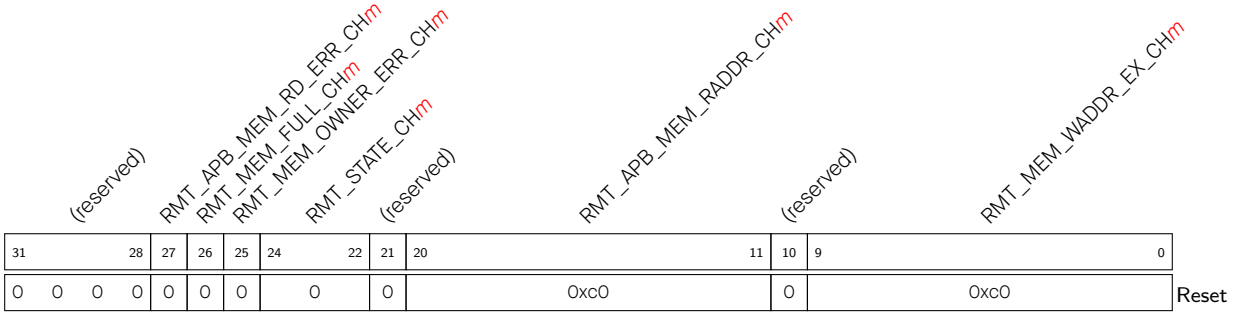
RMT_APB_MEM_WADDR_CH*n* 记录 RMT 使用 APB 总线访问 RAM 时的地址偏移量。(RO)

RMT_STATE_CH*n* 记录通道 *n* 的 FSM 状态。(RO)

RMT_MEM_EMPTY_CH*n* 发送的数据长度大于 RAM block 且乒乓模式未启用时，该状态位将被置位。(RO)

RMT_APB_MEM_WR_ERR_CH*n* RMT 使用 APB 总线进行写 RAM 操作时，如果偏移地址溢出 RAM block，则该状态位将被置位。(RO)

Register 37.10. RMT_CH*m*STATUS_REG (*m* = 4, 5, 6, 7) (0x0060, 0x0064, 0x0068, 0x006C)



RMT_MEM_WADDR_EX_CH*m* 记录通道 *m* 接收器使用 RAM 时的地址偏移量。(RO)

RMT_APB_MEM_RADDR_CH*m* 记录 RMT 使用 APB 总线访问 RAM 时的地址偏移量。(RO)

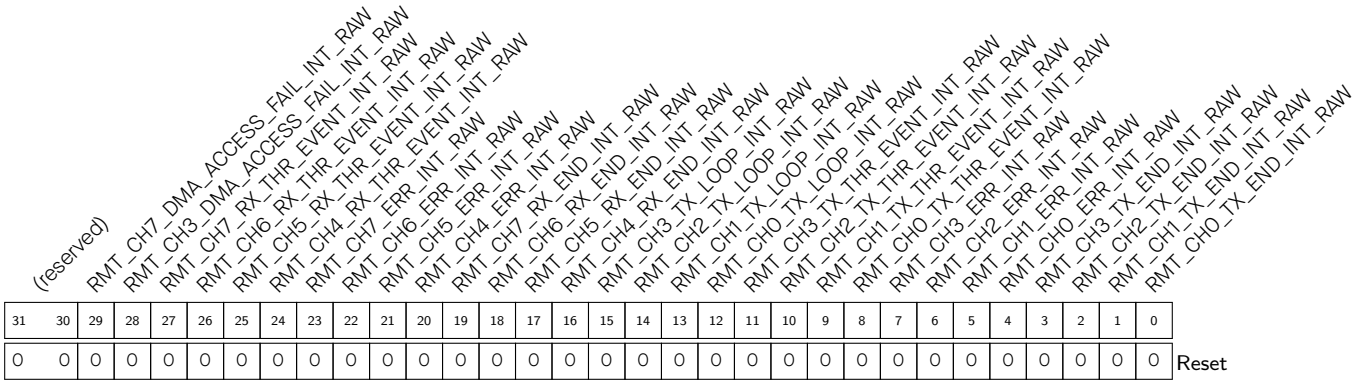
RMT_STATE_CH*m* 记录通道 *m* 的 FSM 状态。(RO)

RMT_MEM_OWNER_ERR_CH*m* RAM block 使用权发生错误时，该状态位将被置位。(RO)

RMT_MEM_FULL_CH*m* 接收器接收的数据长度大于 RAM block 时，此状态位将被置位。(RO)

RMT_APB_MEM_RD_ERR_CH*m* RMT 使用 APB 总线执行 RAM 读操作时，如果偏移地址溢出 RAM block，则该状态位将被置位。(RO)

Register 37.11. RMT_INT_RAW_REG (0x0070)



RMT_CH*n*_TX_END_INT_RAW (*n* = 0-3) RMT_CH*n*_TX_END_INT 的原始中断位。(R/WTC/SS)

RMT_CH*n*_ERR_INT_RAW (*n* = 0-3) RMT_CH*n*_ERR_INT 的原始中断位。(R/WTC/SS)

RMT_CH*n*_TX_THR_EVENT_INT_RAW (*n* = 0-3) RMT_CH*n*_TX_THR_EVENT_INT 的原始中断位。(R/WTC/SS)

RMT_CH*n*_TX_LOOP_INT_RAW (*n* = 0-3) RMT_CH*n*_TX_LOOP_INT 的原始中断位。(R/WTC/SS)

RMT_CH*m*_RX_END_INT_RAW(*m* = 4-7) RMT_CH*m*_RX_END_INT 的原始中断位。(R/WTC/SS)

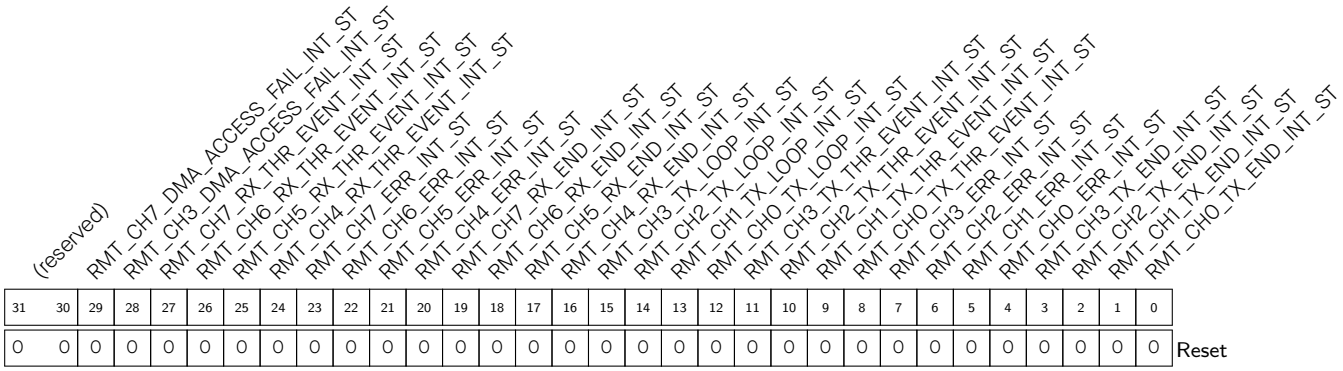
RMT_CH*m*_ERR_INT_RAW (*m* = 4-7) RMT_CH*m*_ERR_INT 的原始中断位。(R/WTC/SS)

RMT_CH*m*_RX_THR_EVENT_INT_RAW(*m* = 4-7) RMT_CH*m*_RX_THR_EVENT_INT 的原始中断位。(R/WTC/SS)

RMT_CH3_DMA_ACCESS_FAIL_INT_RAW RMT_CH3_DMA_ACCESS_FAIL_INT 的原始中断位。(R/WTC/SS)

RMT_CH7_DMA_ACCESS_FAIL_INT_RAW RMT_CH7_DMA_ACCESS_FAIL_INT 的原始中断位。(R/WTC/SS)

Register 37.12. RMT_INT_ST_REG (0x0074)



RMT_CH*n*_TX_END_INT_ST (*n* = 0-3) RMT_CH*n*_TX_END_INT 的屏蔽中断状态位。(RO)

RMT_CH*n*_ERR_INT_ST (*n* = 0-3) RMT_CH*n*_ERR_INT 的屏蔽中断状态位。(RO)

RMT_CH*n*_TX_THR_EVENT_INT_ST (*n* = 0-3) RMT_CH*n*_TX_THR_EVENT_INT 的屏蔽中断状态位。(RO)

RMT_CH*n*_TX_LOOP_INT_ST (*n* = 0-3) RMT_CH*n*_TX_LOOP_INT 的屏蔽中断状态位。(RO)

RMT_CH*m*_RX_END_INT_ST (*m* = 4-7) RMT_CH*m*_RX_END_INT 的屏蔽中断状态位。(RO)

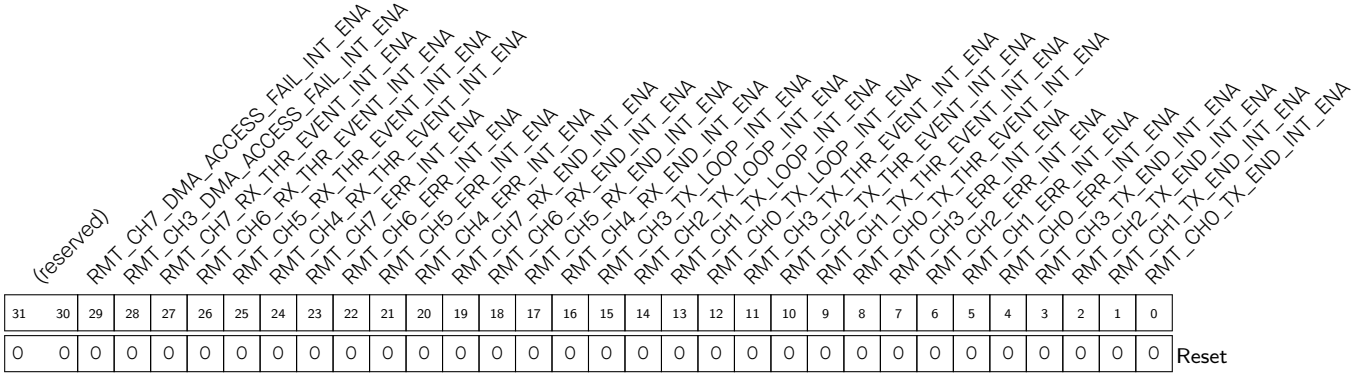
RMT_CH*m*_ERR_INT_ST (*m* = 4-7) RMT_CH*m*_ERR_INT 的屏蔽中断状态位。(RO)

RMT_CH*m*_RX_THR_EVENT_INT_ST (*m* = 4-7) RMT_CH*m*_RX_THR_EVENT_INT 的屏蔽中断状态位。(RO)

RMT_CH3_DMA_ACCESS_FAIL_INT_ST RMT_CH3_DMA_ACCESS_FAIL_INT 的屏蔽中断状态位。(RO)

RMT_CH7_DMA_ACCESS_FAIL_INT_ST RMT_CH7_DMA_ACCESS_FAIL_INT 的屏蔽中断状态位。(RO)

Register 37.13. RMT_INT_ENA_REG (0x0078)



RMT_CH*n*_TX_END_INT_ENA (*n* = 0-3) RMT_CH*n*_TX_END_INT 的中断使能位。(R/W)

RMT_CH*n*_ERR_INT_ENA (*n* = 0-3) RMT_CH*n*_ERR_INT 的中断使能位。(R/W)

RMT_CH*n*_TX_THR_EVENT_INT_ENA (*n* = 0-3) RMT_CH*n*_TX_THR_EVENT_INT 的中断使能位。(R/W)

RMT_CH*n*_TX_LOOP_INT_ENA (*n* = 0-3) RMT_CH*n*_TX_LOOP_INT 的中断使能位。(R/W)

RMT_CH*m*_RX_END_INT_ENA (*m* = 4-7) RMT_CH*m*_RX_END_INT 的中断使能位。(R/W)

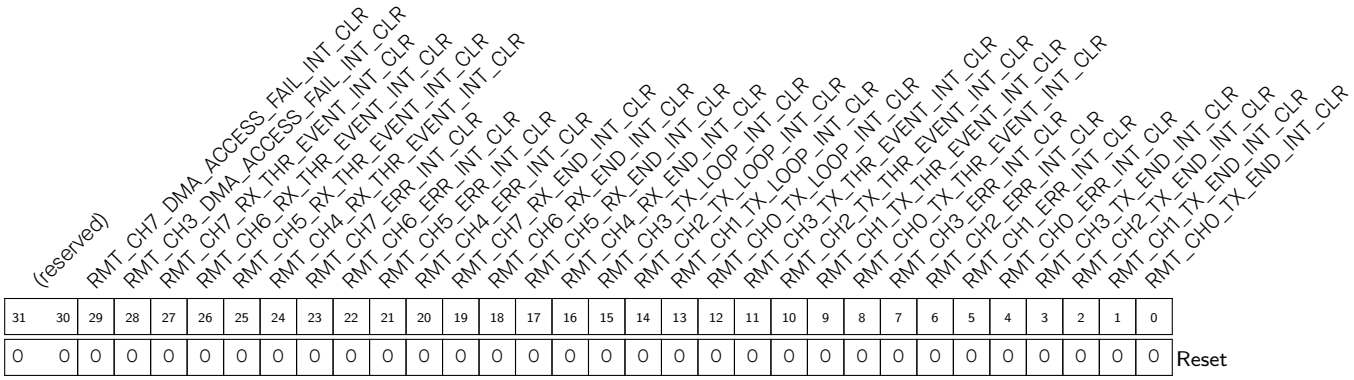
RMT_CH*m*_ERR_INT_ENA (*m* = 4-7) RMT_CH*m*_ERR_INT 的中断使能位。(R/W)

RMT_CH*m*_RX_THR_EVENT_INT_ENA (*m* = 4-7) RMT_CH*m*_RX_THR_EVENT_INT 的中断使能位。(R/W)

RMT_CH3_DMA_ACCESS_FAIL_INT_ENA RMT_CH3_DMA_ACCESS_FAIL_INT 的中断使能位。(R/W)

RMT_CH7_DMA_ACCESS_FAIL_INT_ENA RMT_CH7_DMA_ACCESS_FAIL_INT 的中断使能位。(R/W)

Register 37.14. RMT_INT_CLR_REG (0x007C)



RMT_CH*n*_TX_END_INT_CLR (*n* = 0-3) RMT_CH*n*_TX_END_INT 的中断清除位。(WT)

RMT_CH*n*_ERR_INT_CLR (*n* = 0-3) RMT_CH*n*_ERR_INT 的中断清除位。(WT)

RMT_CH*n*_TX_THR_EVENT_INT_CLR (*n* = 0-3) RMT_CH*n*_TX_THR_EVENT_INT 的中断清除位。(WT)

RMT_CH*n*_TX_LOOP_INT_CLR (*n* = 0-3) RMT_CH*n*_TX_LOOP_INT 的中断清除位。(WT)

RMT_CH*m*_RX_END_INT_CLR (*m* = 4-7) RMT_CH*m*_RX_END_INT 的中断清除位。(WT)

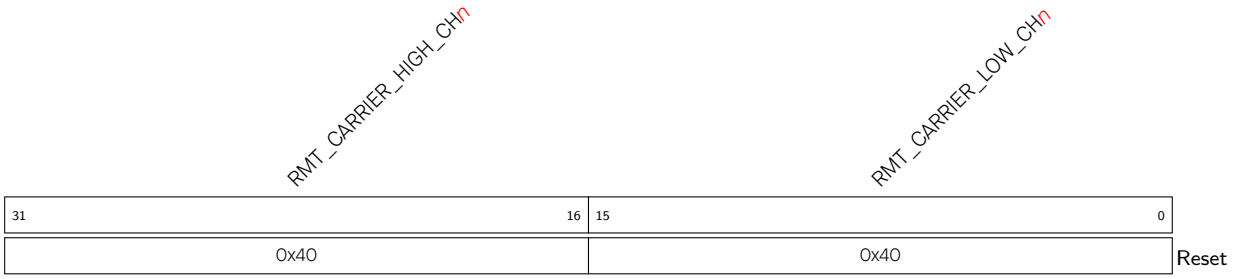
RMT_CH*m*_ERR_INT_CLR (*m* = 4-7) RMT_CH*m*_ERR_INT 的中断清除位。(WT)

RMT_CH*m*_RX_THR_EVENT_INT_CLR (*m* = 4-7) RMT_CH*m*_RX_THR_EVENT_INT 的中断清除位。(WT)

RMT_CH3_DMA_ACCESS_FAIL_INT_CLR RMT_CH3_DMA_ACCESS_FAIL_INT 的中断清除位。(WT)

RMT_CH7_DMA_ACCESS_FAIL_INT_CLR RMT_CH7_DMA_ACCESS_FAIL_INT 的中断清除位。(WT)

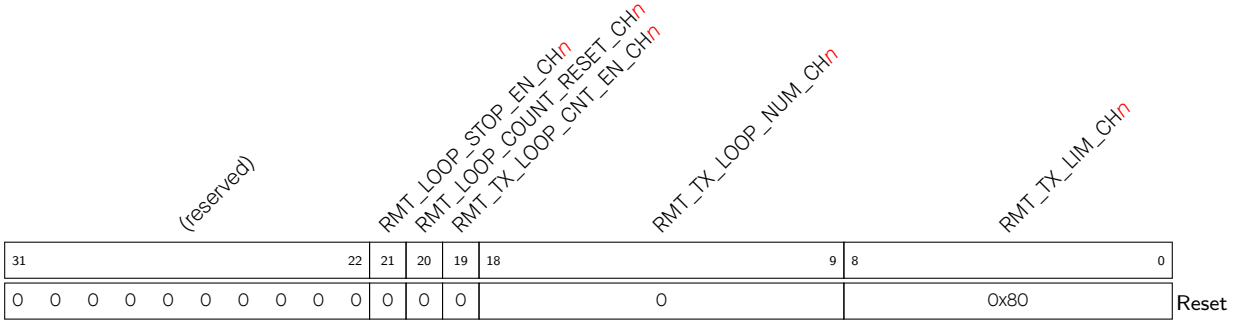
Register 37.15. RMT_CH*n*CARRIER_DUTY_REG (*n*: 0-3) (0x0080+0x4***n*)



RMT_CARRIER_LOW_CH*n* 配置通道 *n* 载波的低电平时钟周期。(R/W)

RMT_CARRIER_HIGH_CH*n* 配置通道 *n* 载波的高电平时钟周期。(R/W)

Register 37.16. RMT_CH*n*_TX_LIM_REG (*n*: 0-3) (0x00A0+0x4*n*)



RMT_TX_LIM_CH*n* 配置通道 *n* 发送脉冲编码数量的上限值。(R/W)

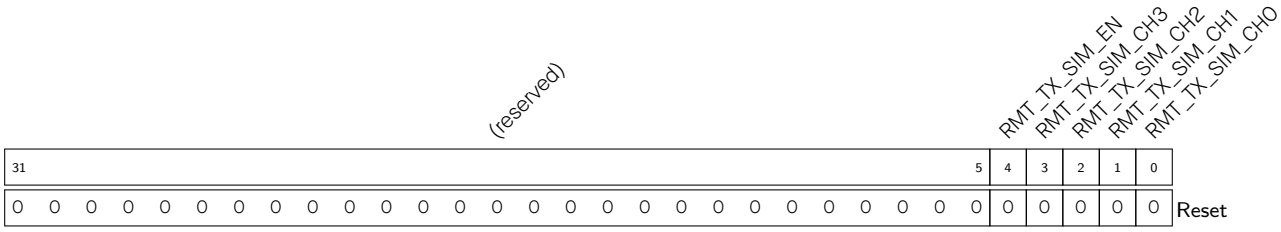
RMT_TX_LOOP_NUM_CH*n* 配置持续发送模式下最大循环发送次数。(R/W)

RMT_TX_LOOP_CNT_EN_CH*n* 置位此位，使能循环次数计数。(R/W)

RMT_LOOP_COUNT_RESET_CH*n* 重置持续发送模式下的循环计数器。(WT)

RMT_LOOP_STOP_EN_CH*n* 置位此位，当通道 *n* 在持续发送模式下，循环发送次数超过 RMT_TX_LOOP_NUM_CH*n* 后，将停止循环发送。(R/W)

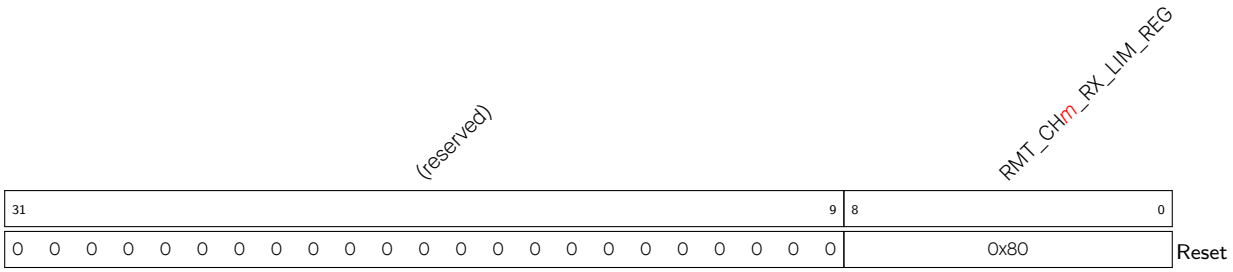
Register 37.17. RMT_TX_SIM_REG (0x00C4)



RMT_TX_SIM_CH*n* (*n* = 0-3) 置位此位，使能通道 *n* 与其它启用的通道同步开始发送数据。(R/W)

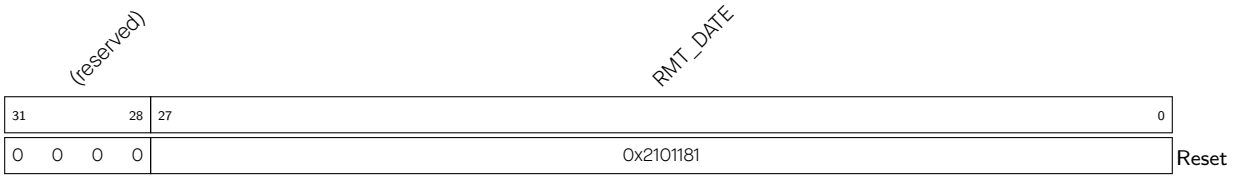
RMT_TX_SIM_EN 置位此位，多个通道开始同步发送数据。(R/W)

Register 37.18. RMT_CH*m*_RX_LIM_REG (*m* = 4, 5, 6, 7) (0x00B0, 0x00B4, 0x00B8, 0x00BC)



RMT_RX_LIM_CH*m* 配置通道 *m* 最大可接收的脉冲编码数量。(R/W)

Register 37.19. RMT_DATE_REG (0x00CC)



RMT_DATE 版本控制寄存器 (R/W)

第 38 章

脉冲计数控制器 (PCNT)

脉冲计数控制器 (Pulse Count Controller, PCNT) 用于对输入脉冲计数，通过记录输入脉冲信号的上升沿或下降沿进行递增或递减计数。PCNT 有四个称为“单元”的独立脉冲计数控制器，这些单元拥有自己的寄存器。PCNT 模块仅有一个时钟，为 APB_CLK。下文描述中 *n* 表示单元编号 0 ~ 3。

每个单元有两个通道（ch0 和 ch1），可以独立配置为递增或递减计数。两个通道功能相同，下文以通道 0 (ch0) 为例进行介绍。

如图 38.0-1 所示，每个通道有两个输入信号：

1. 一个脉冲输入信号（如 sig_ch0_u*n* 为单元 *n* ch0 的脉冲输入信号）
2. 一个控制信号（如 ctrl_ch0_u*n* 为单元 *n* ch0 的控制信号）

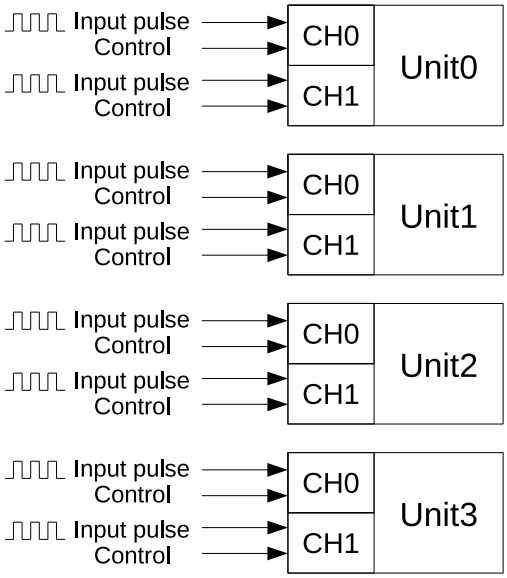


图 38.0-1. PCNT 框图

38.1 主要特性

PCNT 有如下特性：

- 四个脉冲计数控制器（单元），各自独立工作，计数范围是 1 ~ 65535
- 每个单元有两个独立的通道，共用一个脉冲计数控制器
- 所有通道均有输入脉冲信号（如 sig_ch0_u*n*）和相应的控制信号（如 ctrl_ch0_u*n*）
- 滤波器独立工作，过滤每个单元输入脉冲信号（sig_ch0_u*n* 和 sig_ch1_u*n*）控制信号（ctrl_ch0_u*n* 和 ctrl_ch1_u*n*）的毛刺

- 每个通道参数如下：
  1. 选择在输入脉冲信号的上升沿或下降沿计数
  2. 在控制信号为高电平或低电平时可将计数模式配置为递增、递减或停止计数
- 最大脉冲频率：40 MHz

38.2 功能描述

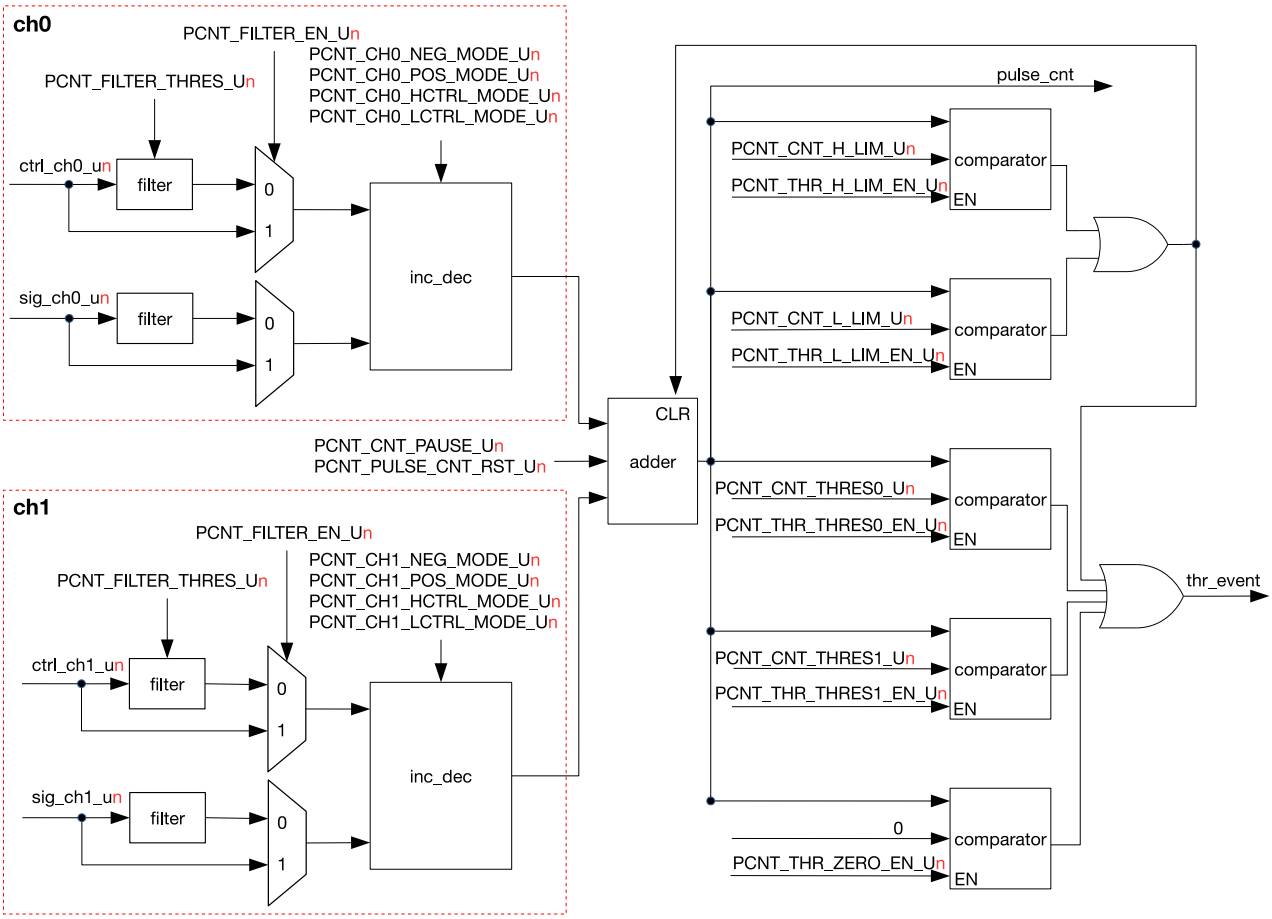


图 38.2-1. PCNT 单元基本架构图

图 38.2-1 为 PCNT 单元的基本架构图。如上所述，ctrl_ch0_u*n* 为单元 *n* ch0 的控制信号，控制信号 ctrl_ch0_u*n* 为高电平或低电平时可配置不同的计数模式，在输入脉冲信号 sig_ch0_u*n* 的上升沿和下降沿计数。可选计数模式如下：

- 递增模式：通道检测到 sig_ch0_u*n* 的有效边沿（软件可配）时，计数器的值 pulse_cnt 加 1。pulse_cnt 的值达到 PCNT_CNT_H_LIM_U*n* 时被清零。如果在 pulse_cnt 达到 PCNT_CNT_H_LIM_U*n* 前，该通道的计数模式改变或 PCNT_CNT_PAUSE_U*n* 置 1，则 pulse_cnt 停止计数，计数模式改变。
- 递减模式：通道检测到 sig_ch0_u*n* 的有效边沿（软件可配）时，计数器的值 pulse_cnt 减 1。pulse_cnt 的值达到 PCNT_CNT_L_LIM_U*n* 时被清零。如果在 pulse_cnt 达到 PCNT_CNT_H_LIM_U*n* 前，该通道的计数模式改变或 PCNT_CNT_PAUSE_U*n* 置 1，则 pulse_cnt 停止计数，计数模式改变。
- 停止计数：计数停止，计数器的值 pulse_cnt 保持不变。

表 38.2-1 至表 38.2-4 说明了如何配置通道 0 的计数模式。

表 38.2-1. 控制信号为低电平时输入脉冲信号上升沿的计数模式

PCNT_CHO_POS_MODE_Un	PCNT_CHO_LCTRL_MODE_Un	计数模式
1	0	递增模式
	1	递减模式
	其他	停止计数
2	0	递减模式
	1	递增模式
	其他	停止计数
其他	N/A	停止计数

表 38.2-2. 控制信号为高电平时输入脉冲信号上升沿的计数模式

PCNT_CHO_POS_MODE_Un	PCNT_CHO_HCTRL_MODE_Un	计数模式
1	0	递增模式
	1	递减模式
	其他	停止计数
2	0	递减模式
	1	递增模式
	其他	停止计数
其他	N/A	停止计数

表 38.2-3. 控制信号为低电平时输入脉冲信号下降沿的计数模式

PCNT_CHO_NEG_MODE_Un	PCNT_CHO_LCTRL_MODE_Un	计数模式
1	0	递增模式
	1	递减模式
	其他	停止计数
2	0	递减模式
	1	递增模式
	其他	停止计数
其他	N/A	停止计数

表 38.2-4. 控制信号为高电平时输入脉冲信号下降沿的计数模式

PCNT_CHO_NEG_MODE_Un	PCNT_CHO_HCTRL_MODE_Un	计数模式
1	0	递增模式
	1	递减模式
	其他	停止计数
2	0	递减模式
	1	递增模式
	其他	停止计数
其他	N/A	停止计数

每个单元均有一个滤波器，用于该单元的所有控制信号和输入脉冲信号。置位 PCNT_FILTER_EN_U*n* 位使能滤波器。滤波器监测信号，滤除脉冲宽度小于 PCNT_FILTER_THRES_U*n* 个 APB 时钟周期的线路毛刺。

如前文所述，每个单元有通道 0 和通道 1 两个通道，处理不同的输入脉冲信号，并通过各自的 inc_dec 模块递增或递减计数值。之后，两个通道将计数值发送给加法器模块，该模块有一个带符号位的 16 位宽寄存器。软件可以通过置位 PCNT_CNT_PAUSE_U*n* 暂停加法器，也可以通过置位 PCNT_PULSE_CNT_RST_U*n* 清零加法器。

PCNT 可以设置五个观察点，五个观察点共用一个中断，可以通过每个观察点各自的中断使能信号开启或屏蔽中断。

- 最大计数值：当 pulse_cnt 大于等于 PCNT_CNT_H_LIM_U*n* 时，产生上限中断，同时 PCNT_CNT_THR_H_LIM_LAT_U*n* 为高。
- 最小计数值：当 pulse_cnt 小于等于 PCNT_CNT_L_LIM_U*n* 时，产生下限中断，同时 PCNT_CNT_THR_L_LIM_LAT_U*n* 为高。
- 两个中间阈值：当 pulse_cnt 等于 PCNT_CNT_THRESO_U*n* 或者 PCNT_CNT_THRES1_U*n* 时，产生中断，同时 PCNT_CNT_THR_THRESO_LAT_U*n* 或 PCNT_CNT_THR_THRES1_LAT_U*n* 为高。
- 零：当 pulse_cnt 等于 0 时，产生中断，同时 PCNT_CNT_THR_ZERO_LAT_U*n* 有效。

38.3 应用实例

每个单元的通道 0 和通道 1 可配置为独立工作或一起工作。下文详细说明了通道 0 独自递增计数、通道 0 独自递减计数和两个通道一起递增计数的应用实例。本节中未详述的通道工作模式（如通道 1 独自递减或递减、双通道一增一减），可参考这三种模式。

38.3.1 通道 0 独自递增计数

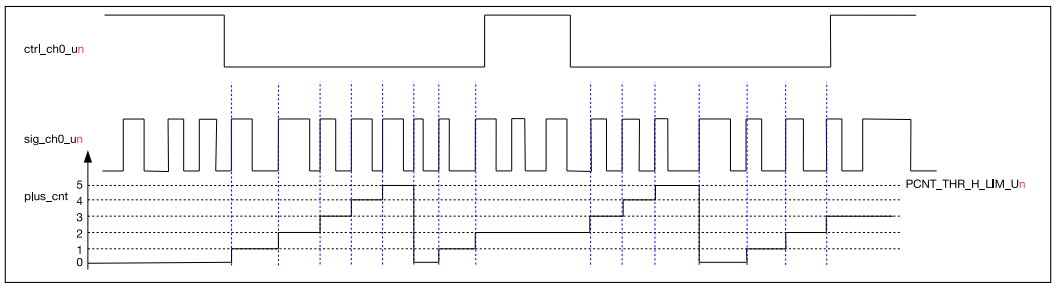


图 38.3-1. 通道 0 递增计数图

图 38.3-1 为通道 0 在 sig_ch0_u*n* 上升沿独立递增计数的示意图，此时通道 1 关闭（请参阅 38.2 一节查看如何关闭通道 1）。通道 0 的配置如下所示。

- PCNT_CH0_LCTRL_MODE_U*n*=0：当 ctrl_ch0_u*n* 为低电平时，递增计数。
- PCNT_CH0_HCTRL_MODE_U*n*=2：当 ctrl_ch0_u*n* 为高电平时，停止计数。
- PCNT_CH0_POS_MODE_U*n*=1：在 sig_ch0_u*n* 的上升沿递增计数。
- PCNT_CH0_NEG_MODE_U*n*=0：在 sig_ch0_u*n* 的下降沿不计数。
- PCNT_CNT_H_LIM_U*n*=5：pulse_cnt 的值递增至 PCNT_CNT_H_LIM_U*n* 时被清零。

38.3.2 通道 0 独自递减计数

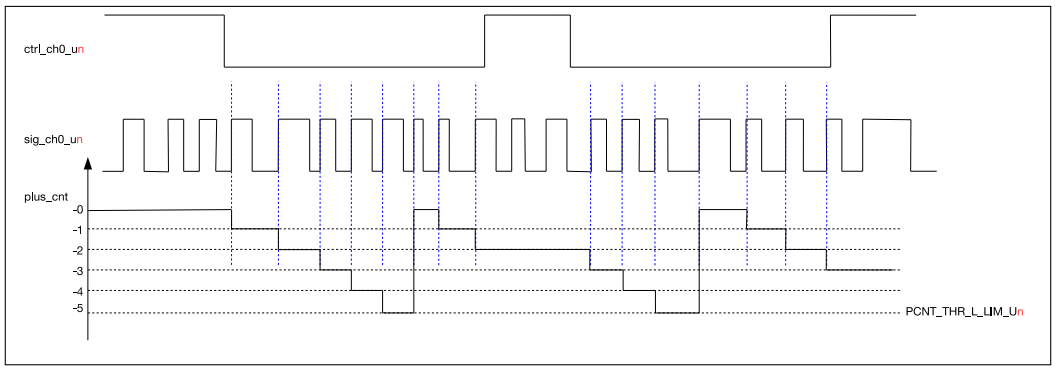


图 38.3-2. 通道 0 递减计数图

图 38.3-2 为通道 0 在 sig_ch0_u*n* 上升沿独立递减计数的示意图，此时通道 1 关闭。此时通道 0 的配置与图 38.3-1 相比有如下区别：

- PCNT_CH0_POS_MODE_U*n*=2：即在 sig_ch0_u*n* 的上升沿递减计数。
- PCNT_CNT_L_LIM_U*n*=-5：pulse_cnt 的值递减到 PCNT_CNT_L_LIM_U*n* 时被清零。

38.3.3 通道 0 和通道 1 同时递增计数

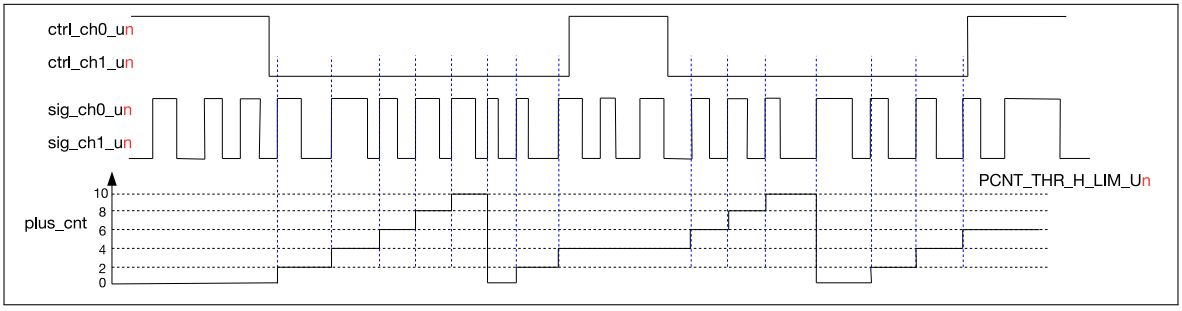


图 38.3-3. 双通道递增计数图

图 38.3-3 为通道 0 和通道 1 分别在 sig_ch0_u*n* 和 sig_ch1_u*n* 上升沿同时递增计数的示意图。如图 38.3-3 所示，控制信号 ctrl_ch0_u*n* 与 ctrl_ch1_u*n* 的波形一致，输入脉冲信号 sig_ch0_u*n* 和 sig_ch1_u*n* 波形一致。具体配置如下：

- 通道 0：
  - PCNT_CH0_LCTRL_MODE_U*n*=0：当 ctrl_ch0_u*n* 为低电平时，递增计数。
  - PCNT_CH0_HCTRL_MODE_U*n*=2：当 ctrl_ch0_u*n* 为高电平时，停止计数。
  - PCNT_CH0_POS_MODE_U*n*=1：在 sig_ch0_u*n* 的上升沿递增计数。
  - PCNT_CH0_NEG_MODE_U*n*=0：在 sig_ch0_u*n* 的下降沿不计数。
- 通道 1：
  - PCNT_CH1_LCTRL_MODE_U*n*=0：当 ctrl_ch1_u*n* 为低电平时，递增计数。
  - PCNT_CH1_HCTRL_MODE_U*n*=2：当 ctrl_ch1_u*n* 为高电平时，停止计数。
  - PCNT_CH1_POS_MODE_U*n*=1：在 sig_ch1_u*n* 的上升沿递增计数。

  - PCNT_CH1_NEG_MODE_U*n*=0：在 sig_ch1_u*n* 的下降沿不计数。

- PCNT_CNT_H_LIM_U*n*=10：pulse_cnt 递增至 PCNT_CNT_H_LIM_U*n* 时被清零。

38.4 寄存器列表

本小节的所有地址均为相对于 脉冲计数控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

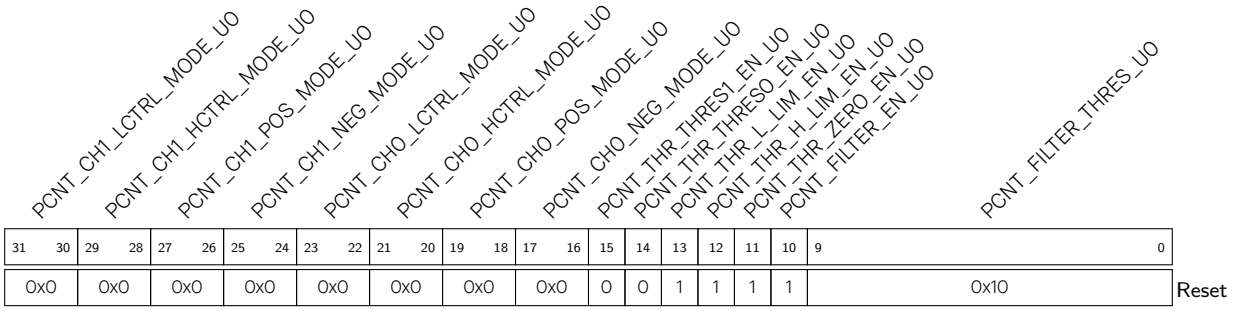
请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
配置寄存器			
PCNT_U0_CONF0_REG	单元 0 的配置寄存器 0	0x0000	读/写
PCNT_U0_CONF1_REG	单元 0 的配置寄存器 1	0x0004	读/写
PCNT_U0_CONF2_REG	单元 0 的配置寄存器 2	0x0008	读/写
PCNT_U1_CONF0_REG	单元 1 的配置寄存器 0	0x000C	读/写
PCNT_U1_CONF1_REG	单元 1 的配置寄存器 1	0x0010	读/写
PCNT_U1_CONF2_REG	单元 1 的配置寄存器 2	0x0014	读/写
PCNT_U2_CONF0_REG	单元 2 的配置寄存器 0	0x0018	读/写
PCNT_U2_CONF1_REG	单元 2 的配置寄存器 1	0x001C	读/写
PCNT_U2_CONF2_REG	单元 2 的配置寄存器 2	0x0020	读/写
PCNT_U3_CONF0_REG	单元 3 的配置寄存器 0	0x0024	读/写
PCNT_U3_CONF1_REG	单元 3 的配置寄存器 1	0x0028	读/写
PCNT_U3_CONF2_REG	单元 3 的配置寄存器 2	0x002C	读/写
PCNT_CTRL_REG	所有计数器的控制寄存器	0x0060	读/写
状态寄存器			
PCNT_U0_CNT_REG	单元 0 的计数器值	0x0030	只读
PCNT_U1_CNT_REG	单元 1 的计数器值	0x0034	只读
PCNT_U2_CNT_REG	单元 2 的计数器值	0x0038	只读
PCNT_U3_CNT_REG	单元 3 的计数器值	0x003C	只读
PCNT_U0_STATUS_REG	脉冲计数器单元 0 的状态寄存器	0x0050	只读
PCNT_U1_STATUS_REG	脉冲计数器单元 1 的状态寄存器	0x0054	只读
PCNT_U2_STATUS_REG	脉冲计数器单元 2 的状态寄存器	0x0058	只读
PCNT_U3_STATUS_REG	脉冲计数器单元 3 的状态寄存器	0x005C	只读
中断寄存器			
PCNT_INT_RAW_REG	原始中断状态寄存器	0x0040	只读
PCNT_INT_ST_REG	中断状态寄存器	0x0044	只读
PCNT_INT_ENA_REG	中断使能寄存器	0x0048	读/写
PCNT_INT_CLR_REG	中断清除寄存器	0x004C	只写
版本寄存器			
PCNT_DATE_REG	脉冲计数器的版本控制寄存器	0x00FC	读/写

38.5 寄存器

本小节的所有地址均为相对于 脉冲计数控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 38.1. PCNT_U*n*_CONF0_REG (*n*: 0-3) (0x0000+0xC*n*)



PCNT_FILTER_THRES_U*n* 滤波器设置的最大阈值，以 APB_CLK 时钟周期为单位。

滤波器启动时，任何小于该值的脉冲都会被过滤。(R/W)

PCNT_FILTER_EN_U*n* 单元 *n* 输入滤波器的使能位。(R/W)

PCNT_THR_ZERO_EN_U*n* 单元 *n* 过零比较器的使能位。(R/W)

PCNT_THR_H_LIM_EN_U*n* 单元 *n* 上限比较器的使能位。(R/W)

PCNT_THR_L_LIM_EN_U*n* 单元 *n* 下限比较器的使能位。(R/W)

PCNT_THR_THRES0_EN_U*n* 单元 *n* 阈值 0 比较器的使能位。(R/W)

PCNT_THR_THRES1_EN_U*n* 单元 *n* 阈值 1 比较器的使能位。(R/W)

PCNT_CH0_NEG_MODE_U*n* 用于设置通道 0 输入信号检测下降沿的工作模式。

1：增加计数器；2：减少计数器；0、3：对计数器无任何影响。(R/W)

PCNT_CH0_POS_MODE_U*n* 用于设置通道 0 输入信号检测上升沿的工作模式。

1：增加计数器；2：减少计数器；0、3：对计数器无任何影响。(R/W)

PCNT_CH0_HCTRL_MODE_U*n* 控制信号为高电平时，用于改变 CH*n*_POS_MODE 和 CH*n*_NEG_MODE 的设置。

0：不做修改；1：反转（增加转为减少，减少转为增加）；2、3：禁止计数器修改。(R/W)

PCNT_CH0_LCTRL_MODE_U*n* 控制信号为低电平时，用于改变 CH*n*_POS_MODE 和 CH*n*_NEG_MODE 的设置。

0：不做修改；1：反转（增加转为减少，减少转为增加）；2、3：禁止计数器修改。(R/W)

PCNT_CH1_NEG_MODE_U*n* 用于设置通道 1 输入信号检测下降沿的工作模式。

1：计数器递增；2：计数器递减；0、3：对计数器无任何影响。(R/W)

见下页...

Register 38.1. PCNT_U*n*_CONF0_REG (*n*: 0-3) (0x0000+0xC*n*)

接上页...

PCNT_CH1_POS_MODE_U*n* 用于设置通道 1 输入信号检测上升沿的工作模式。

1：计数器递增；2：计数器递减；0、3：对计数器无任何影响。(R/W)

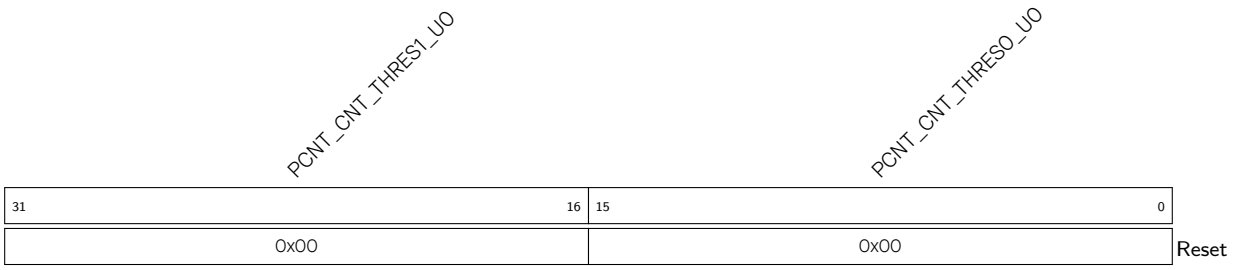
PCNT_CH1_HCTRL_MODE_U*n* 控制信号为高电平时，用于改变 CH*n*_POS_MODE 和 CH*n*_NEG_MODE 的设置。

0：不做修改；1：反转（增加转为减少，减少转为增加）；2、3：禁止计数器修改。(R/W)

PCNT_CH1_LCTRL_MODE_U*n* 控制信号为低电平时，用于改变 CH*n*_POS_MODE 和 CH*n*_NEG_MODE 的设置。

0：不做修改；1：反转（增加转为减少，减少转为增加）；2、3：禁止计数器修改。(R/W)

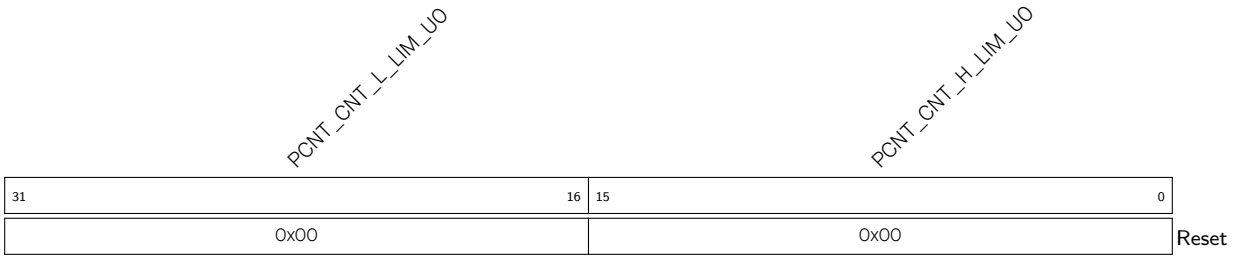
Register 38.2. PCNT_U*n*_CONF1_REG (*n*: 0-3) (0x0004+0xC*n*)



PCNT_CNT_THRES0_U*n* 用于配置单元 *n* 阈值 0 的值。(R/W)

PCNT_CNT_THRES1_U*n* 用于配置单元 *n* 阈值 1 的值。(R/W)

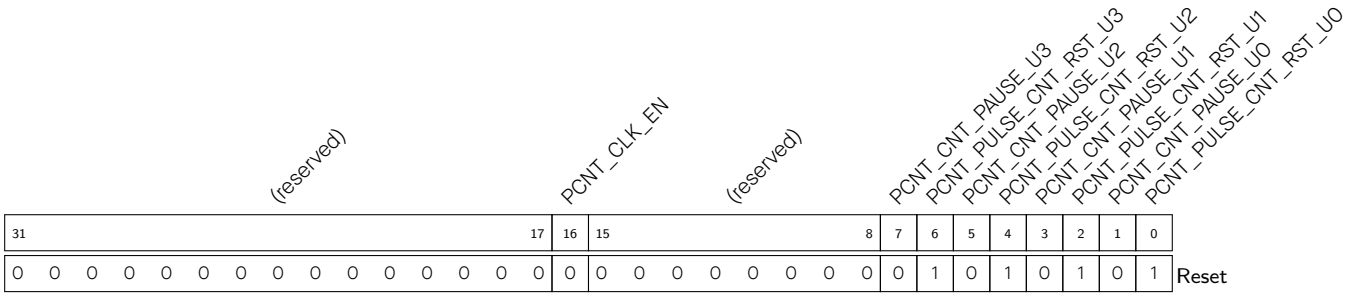
Register 38.3. PCNT_U*n*_CONF2_REG (*n*: 0-3) (0x0008+0xC*n*)



PCNT_CNT_H_LIM_U*n* 用于配置单元 *n* 的计数上限阈值。(R/W)

PCNT_CNT_L_LIM_U*n* 用于配置单元 *n* 的计数下限阈值。(R/W)

Register 38.4. PCNT_CTRL_REG (0x0060)

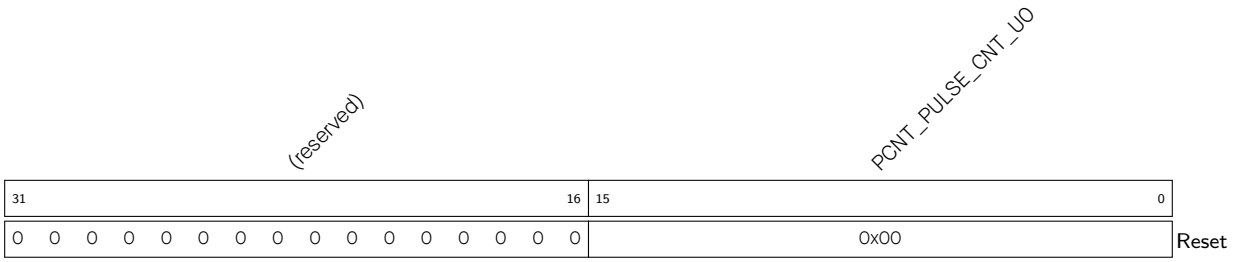


PCNT_PULSE_CNT_RST_U*n* 置位此位，清零单元 *n* 的计数器。(R/W)

PCNT_CNT_PAUSE_U*n* 置位此位，暂停单元 *n* 的计数器。(R/W)

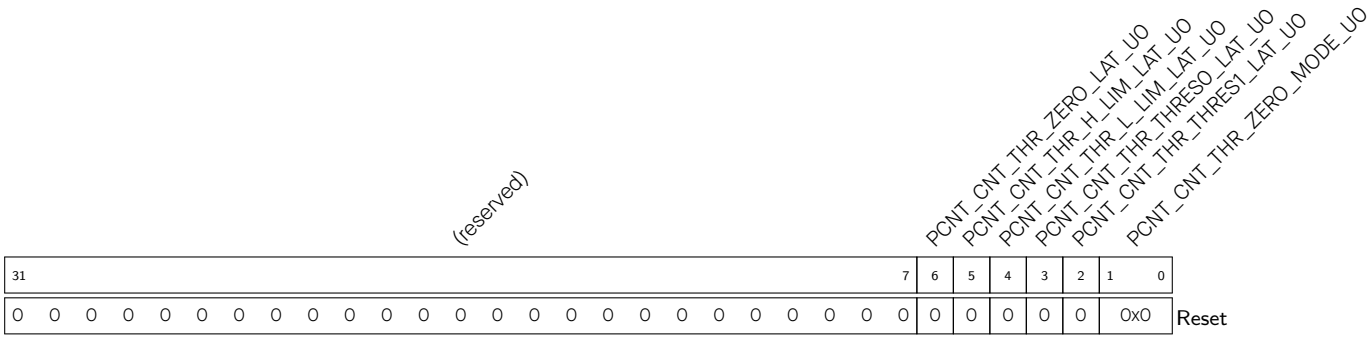
PCNT_CLK_EN 脉冲计数器模块寄存器时钟门控的使能信号 1：寄存器可通过应用读取、写值。0：寄存器无法通过应用读取、写值。(R/W)

Register 38.5. PCNT_U*n*_CNT_REG (*n*: 0-3) (0x0030+0x4**n*)



PCNT_PULSE_CNT_U*n* 存储单元 *n* 脉冲计数器的当前值。(RO)

Register 38.6. PCNT_U*n*_STATUS_REG (*n*: 0-3) (0x0050+0x4*n*)



PCNT_CNT_THR_ZERO_MODE_U*n* PCNT_U*n* 为 0 时的脉冲计数器状态。0：脉冲计数器的值由正数减至 0。1：脉冲计数器的值由负数增至 0。2：脉冲计数器为负。3：脉冲计数器为正。(RO)

PCNT_CNT_THR_THRES1_LAT_U*n* 阈值中断有效时，PCNT_U*n* 阈值 1 的锁存值。1：脉冲计数器的当前值与阈值 1 相等，阈值 1 有效。0：其他。(RO)

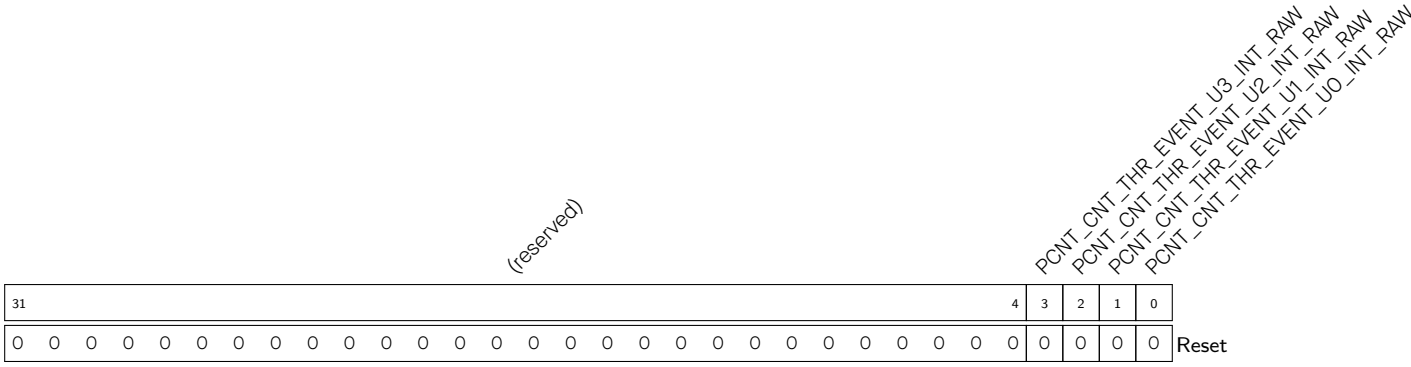
PCNT_CNT_THR_THRESO_LAT_U*n* 阈值中断有效时，PCNT_U*n* 阈值 0 的锁存值。1：脉冲计数器的当前值与阈值 0 相等，阈值 0 有效。0：其他。(RO)

PCNT_CNT_THR_L_LIM_LAT_U*n* 下限中断有效时，PCNT_U*n* 下限的锁存值。1：脉冲计数器的当前值与下限阈值相等，下限有效。0：其他。(RO)

PCNT_CNT_THR_H_LIM_LAT_U*n* 上限中断有效时，PCNT_U*n* 上限的锁存值。1：脉冲计数器的当前值与上限阈值相等，上限有效。0：其他。(RO)

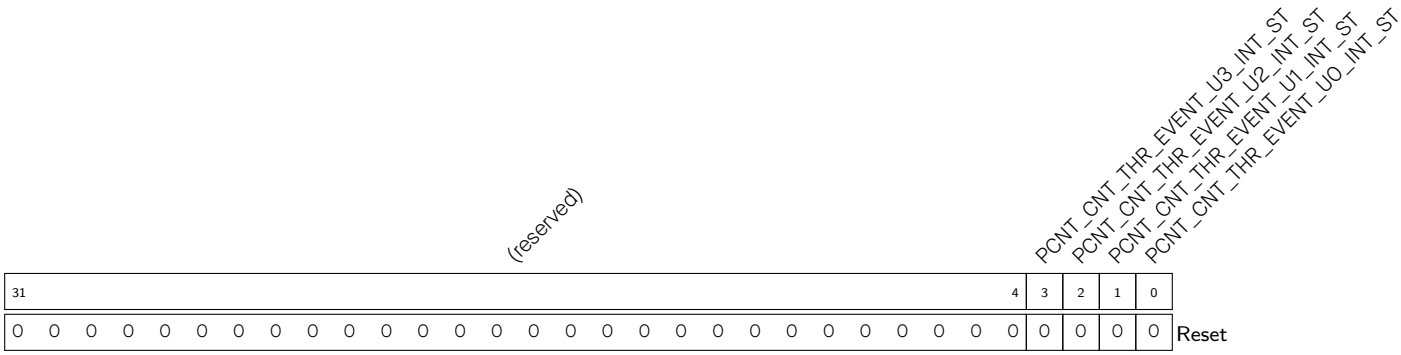
PCNT_CNT_THR_ZERO_LAT_U*n* 阈值中断有效时，PCNT_U*n* 阈值 0 的锁存值。1：脉冲计数器的当前值为 0，阈值 0 有效。0：其他。(RO)

Register 38.7. PCNT_INT_RAW_REG (0x0040)



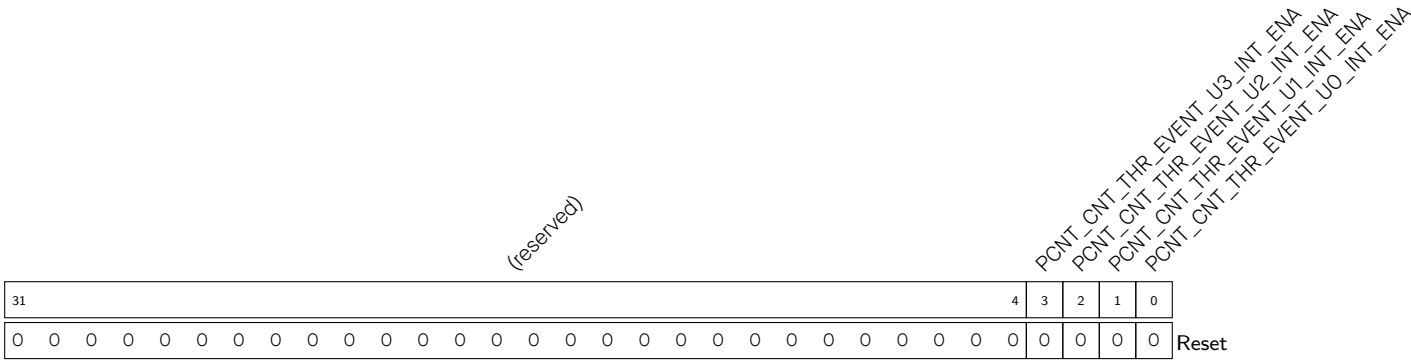
PCNT_CNT_THR_EVENT_U*n*_INT_RAW 单元 *n* 事件中断的原始中断状态位。(RO)

Register 38.8. PCNT_INT_ST_REG (0x0044)



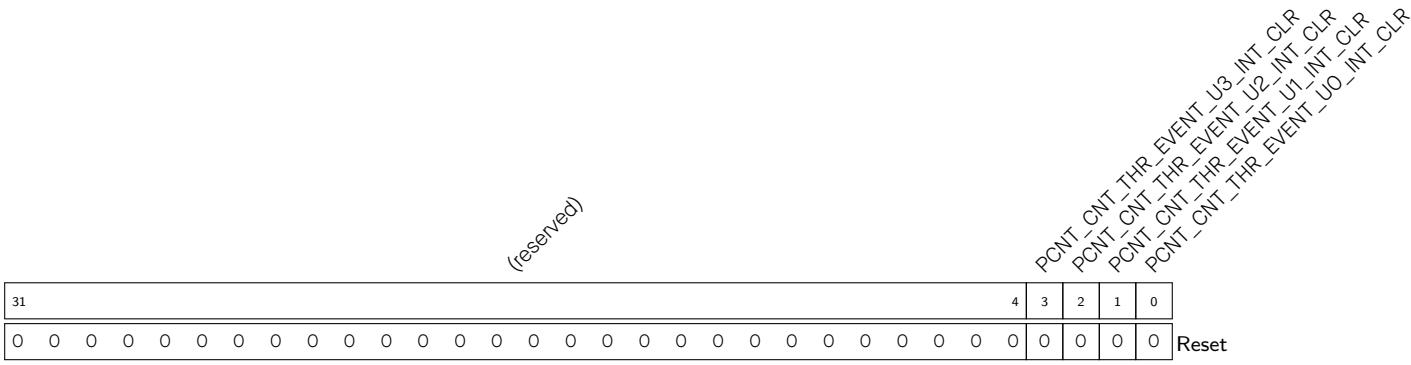
PCNT_CNT_THR_EVENT_U*n*_INT_ST 单元 *n* 事件中断的屏蔽中断状态位。(RO)

Register 38.9. PCNT_INT_ENA_REG (0x0048)



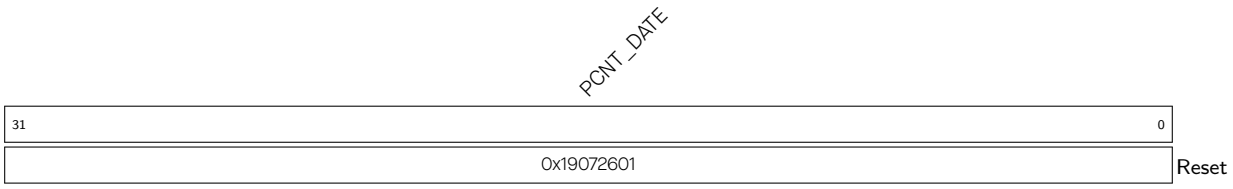
PCNT_CNT_THR_EVENT_U*n*_INT_ENA 单元 *n* 事件中断的中断使能位。(R/W)

Register 38.10. PCNT_INT_CLR_REG (0x004C)



PCNT_CNT_THR_EVENT_U*n*_INT_CLR 置位此位，清除单元 *n* 事件中断。(WO)

Register 38.11. PCNT_DATE_REG (0x00FC)



PCNT_DATE 脉冲计数器的版本控制寄存器。(R/W)

第 VI 卷

模拟信号处理

该部分描述了与模数转换、片上传感器以及温度感应等功能相关的模块，展示了系统在处理模拟信号方面的性能。

第 39 章

片上传感器与模拟信号处理

39.1 概述

ESP32-S3 搭载了以下片上传感器和模拟信号处理设备：

- 14 个电容式触摸传感器：常用于手指触摸检测，支持 14 个通道电容检测、支持防潮功能、遇水保护功能和接近感应模式。
- 温度传感器：用于测量 ESP32-S3 芯片内部温度。
- 两个 12 位逐次逼近型模数转换器 (SAR ADC)：支持 20 个通道的模拟信号检测。系统专门内置了五个专用控制器，可在转换模拟输入信号时支持高性能与低功耗两种模式。

39.2 电容式触摸传感器

39.2.1 术语

为了更好地对电容式触摸传感器进行说明，本章及后续章节使用了以下术语。

- 触摸管脚 (Touch Pin)：指 ESP32-S3 芯片提供的具有触摸传感功能的 GPIO 接口。
- 触摸传感器 (Touch Sensor)：指 ESP32-S3 提供的实现触摸传感功能的内部电路。
- 触摸板 (Touch Panel)：指芯片外部连接的用于接收触摸动作的设备。
- 触摸传感器系统 (Touch Sensor System)：触摸管脚、触摸传感器、触摸板、以及连接各部分的电路共同构成触摸传感器系统。

说明：
为方便描述，对触摸板进行采样、扫描、测量、充放电等操作，即表示对触摸管脚进行采样、扫描、测量、充放电等操作。

39.2.2 概述

ESP32-S3 内置电容式触摸传感器，通过触摸管脚外接触摸板，构成触摸传感器系统。触摸传感器系统主要应用于人机交互场景中，检测手指触碰或接近。其中，触摸板主要由三部分组成：

- 电极 (Electrode)：手指触碰时，电极电容发生变化。
- 基片 (Substrate)：基础材料，承载平面保护层、电极及接入相应通道需要的连接部件。
- 平面保护层 (Protective cover)：将其余部件与外界环境隔开。

用户需自行设计触摸板，确定触摸板用材及排列方式等，具体信息可参考触摸传感器应用方案简介。触摸板经触摸管脚（通道）连接至 ESP32-S3 提供的触摸传感器，如下图 39.2-1 所示。

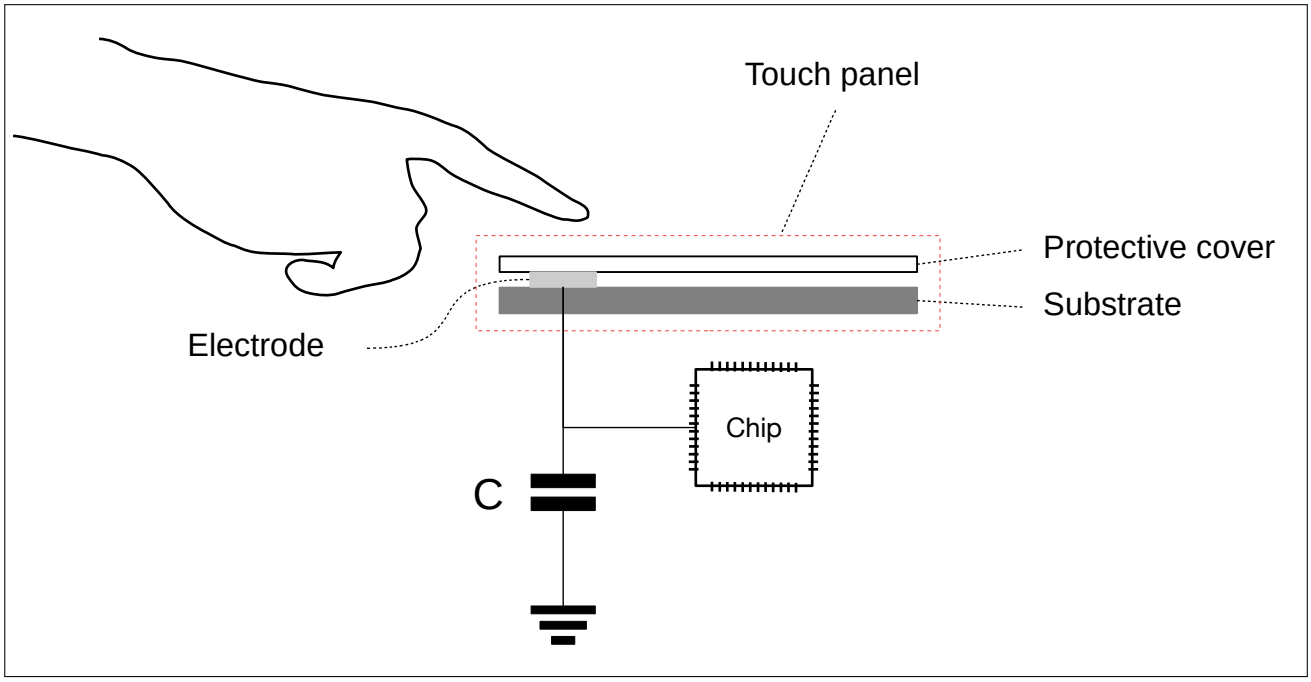


图 39.2-1. 触摸传感器

用户触碰保护层时，电极电容增加。如果该电极通过通道已连接至 ESP32-S3 触摸传感器，则传感器将检测到电极的电容变化。电容变化超过阈值（可配置），传感器将触发中断。

39.2.3 主要特性

- 集成 14 个触摸传感器 (T1 ~ T14)，每个传感器通过专用通道，可连接至外部触摸板。触摸传感器 T0 为内部通道，无法外接触摸板，因此常用于噪声检测，详细信息见章节 39.2.8。
- 触摸传感器由触摸有限状态机 (Touch FSM) 控制。Touch FSM 可由软件或专用硬件定时器触发，然后对某个触摸管脚执行测量操作，即采样某个触摸管脚。
- 用户可配置 Touch FSM 按照特定顺序对多个触摸管脚进行采样，即扫描 (Scan) 模式。Scan 模式可用于同时监控多个触摸管脚。
- 支持最多三个通道配置为接近感应模式。
- 支持下面两种触摸动作检测方式，用于检测触摸管脚是否被触碰：
  - 软件轮询触摸传感器采样值
  - 使用内置硬件算法
- 触摸传感器可在 CPU 处于睡眠模式时正常工作。
- 协助 ESP32-S3 芯片在下面两种低功耗场景下工作：
  - CPU 处于 Deep-sleep 模式时，可配置触摸管脚为唤醒源，详细信息见 RTC_CNTL_TOUCH_SLP_PAD。如果 RTC 外设电源域关闭（参考表 10.4-1），则只能将一个触摸管脚配置为唤醒源。

  - 触摸管脚可由 ULP 协处理器控制。配置 ULP 协处理器扫描多个触摸管脚，达到触摸阈值，ULP 协处理器唤醒主 CPU。

- 防潮设计，可缓和小水滴对传感器的影响。
- 遇水保护设计，如果检测到传感器触摸板阵列表面已被水覆盖，将触发软件关停传感器。
- 内部噪声过滤

说明：
ESP32-S3 触摸传感器目前尚无法通过射频抗扰度测试系统 (CS) 认证，应用场景有所限制。

39.2.4 电容触摸管脚

ESP32-S3 搭载 14 个电容触摸传感器 (T1 ~ T14)，通过芯片电容触摸管脚与外部触摸板连接，用于检测外部的触摸情况。对应的连接关系见表 39.2-1。另一个传感器 (T0) 未引出，可用于检测芯片内部的噪声变化，见章节 39.2.8。

表 39.2-1. ESP32-S3 电容式触摸传感器的管脚

触摸传感信号名称	管脚
T0	内部通道，无对应管脚
T1	GPIO1
T2	GPIO2
T3	GPIO3
T4	GPIO4
T5	GPIO5
T6	GPIO6
T7	GPIO7
T8	GPIO8
T9	GPIO9
T10	GPIO10
T11	GPIO11
T12	GPIO12
T13	GPIO13
T14	GPIO14

39.2.5 触摸传感器工作原理和工作信号

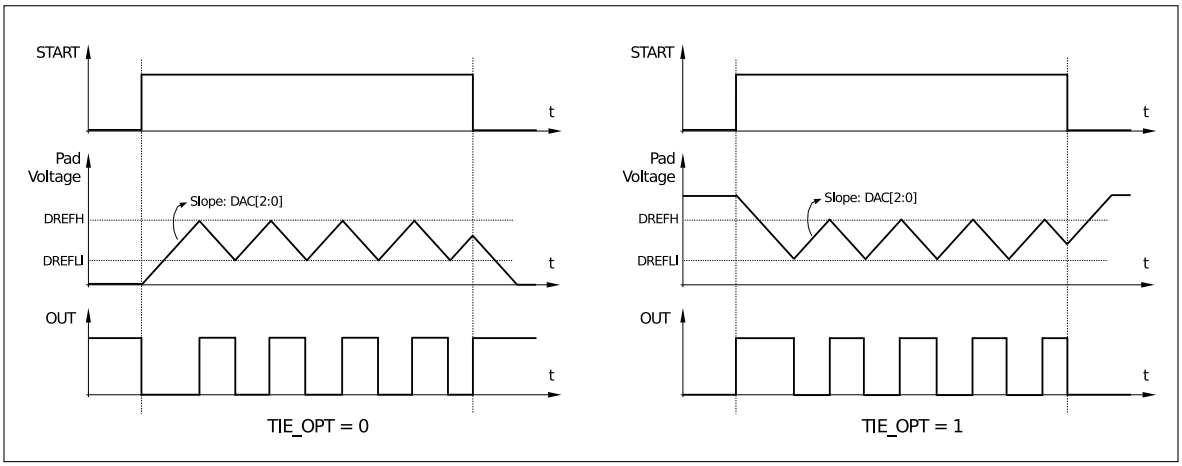


图 39.2-2. 触摸传感器工作原理

图 39.2-2 展示了触摸传感器的工作原理。手指触碰或靠近触摸管脚时，管脚电容增大。触摸传感器可检测到触摸管脚上的电容变化。

触摸传感器使用固定电源对触摸管脚进行充放电，电压范围：参考高值 DREFH ~ 参考低值 DREFL。触摸管脚被触碰时，其电容增大，因此充放电时间延长。通过测量触摸管脚在固定周期内充放电所需时间，可判断管脚是否被触碰。

每次电压变化，触摸传感器生成一个脉冲。一个 OUT 信号包含若干个脉冲。一次测量操作包括：对触摸管脚进行 N 次（固定次数，可配置）的充放电，然后测量 OUT 信号生成 N 个脉冲所需时间。

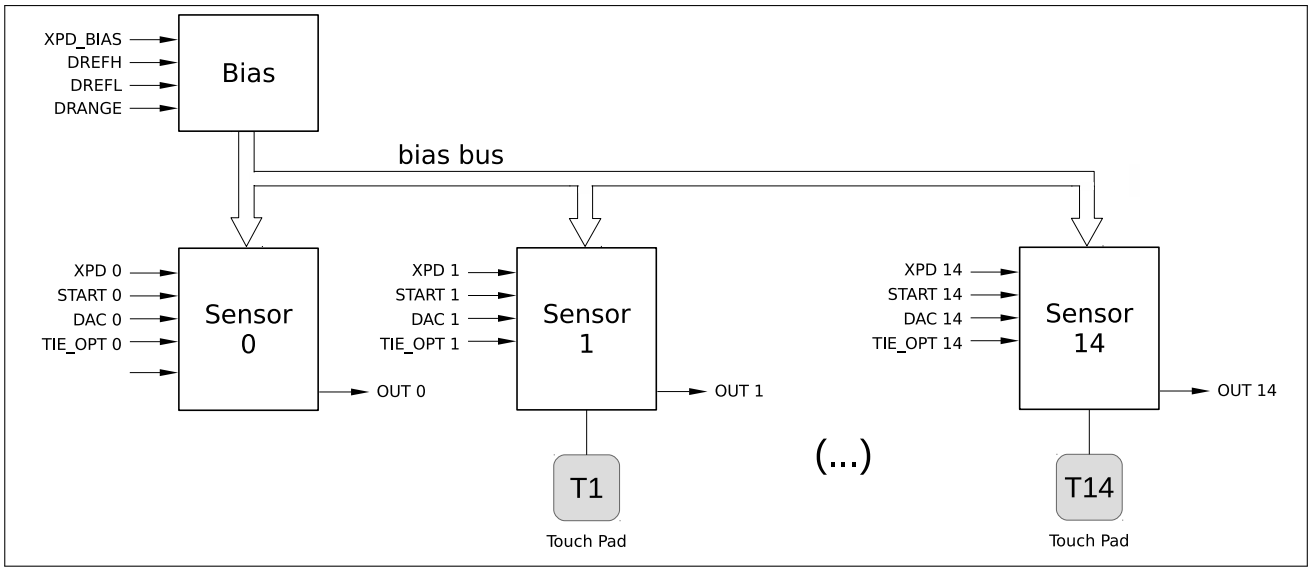


图 39.2-3. 触摸传感器的内部结构

图 39.2-3 展示了触摸传感器的内部结构。每个触摸传感器均有一组输入输出信号，其中一些信号连接至 Touch FSM，见章节 39.2.6。

- START 信号：由 Touch FSM 产生，控制触摸传感器发起一次测量。Touch FSM 拉低 START 信号，测量结束。

- OUT 信号：由触摸传感器产生，发送给 Touch FSM。Touch FSM 对 OUT 信号中脉冲进行计数，测量产生 N 个脉冲需要的时间。
- TIE_OPT 信号：控制触摸传感器的初始电压（DREFH 或 DREFL）。用户可以通过 RTCIO_TOUCH_PADn_TIE_OPT (TIE_OPT) 设置开始充放电的初始电压电平。
- DAC 信号：触摸传感器斜率控制，用户可通过 RTC_CNTL_TOUCH_PADn_DAC 来配置斜率。
- XPD 信号：触摸传感器上电信号，由 RTCIO_TOUCH_PADn_XPD 控制。

39.2.6 Touch FSM

Touch FSM 负责执行测量操作，包括选择要测量的触摸传感器，为触摸传感器生成需要的信号，以及处理触摸传感器输出的信号等。Touch FSM 的内部结构见图 39.2-4，时钟源为 RTC_FAST_CLK。关于 RTC_FAST_CLK 更多信息，请参考章节 7 复位和时钟。

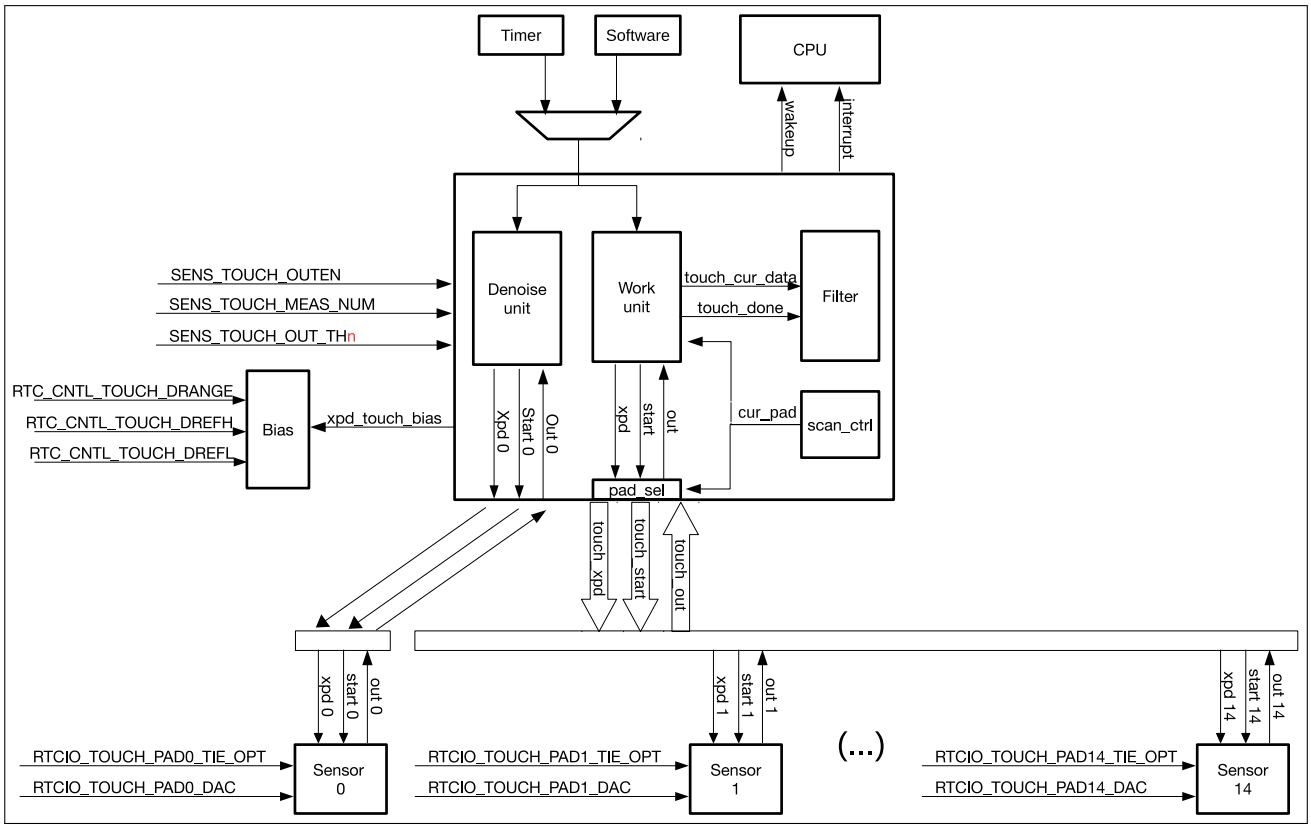


图 39.2-4. Touch FSM 的内部结构

Touch FSM 各个模块的功能如下：

- SCAN_CTRL 模块：SCAN 模式下，选择要测量的触摸管脚。
- WORK_UNIT 模块：测量过程中，负责驱动已选择的触摸管脚。
- DENOISE_UNIT 模块：与 WORK_UNIT 结构几乎一样，但连接的是内部触摸传感器 TO。其它触摸传感器可使用 TO 的测量数据改善噪声带来的影响，见章节 39.2.8。
- Filter 模块：使能此模块，则触摸传感器在测量时将使用 IIR 滤波器，返回的滤波值即为采样值。详细信息见章节 39.2.7.1。

39.2.6.1 测量过程

触摸传感器的一次测量操作包括以下流程：

1. Touch FSM 选择需要测量的触摸传感器对应的管脚，将相关信号发送到该传感器。
2. Touch FSM 发送 START 信号到触摸传感器，初始化测量操作。Touch FSM 启动内部触摸计数器 (touch counter) 记录测量持续时长。
3. Touch FSM 内部的脉冲计数器 (pulse counter) 开始对 OUT 信号的脉冲数量进行计数。
4. 脉冲计数器达到 RTC_CNTL_TOUCH_MEAS_NUM 中设定的计数阈值，则测量结束。START 信号拉低，触摸计数器停止计数。触摸计数器的值即为对触摸管脚进行 RTC_CNTL_TOUCH_MEAS_NUM 次充放电所需时间。

采样值（测量值）为触摸计数器的值，表示为原始触摸数据 (touch_raw_data)。用户可从 SENS_TOUCH_PAD*n*_DATA 中读取原始触摸数据的值。注意：SENS_TOUCH_PAD*n*_DATA 返回的值也可能是滤波后的原始触摸数据，即平滑触摸数据 (touch_smooth_data) 和基准数据 (benchmark)。用户可配置 SENS_TOUCH_DATA_SEL 选择具体的返回值类型。有关采样值返回类型的详细信息以及如何使用这些数据判断触摸动作，请见章节 39.2.7.1。

注意：如果脉冲计数器长时间未达到设置的计数阈值，而触摸计数器达到 RTC_CNTL_TOUCH_TIMEOUT_NUM 中设定的超时阈值，则将触发TOUCH_TIME_OUT_INT 超时中断，提示电路异常。如果已启用 TOUCH Timeout 作为睡眠模式的唤醒源，则同时触发唤醒信号（更多信息，参考小节 10.4.4）。

39.2.6.2 测量操作的触发源

Touch FSM 通过发送 START 信号发起一次测量操作。START 信号可由软件触发，或由触摸定时器 (touch timer) 触发。使用触摸定时器可发起周期性测量，而无需软件介入。

触摸定时器的时钟源为 RTC_SLOW_CLK，时长可设置为若干个 RTC_SLOW_CLK 周期。定时器过期则生成 START 信号。测量操作执行完毕后，定时器计数复位，开始重新计数，直至下次过期。

- 配置 START 信号由软件触发：
  - 置位 RTC_CNTL_TOUCH_START_FORCE，选择 START 信号由软件触发。
  - 软件置位 RTC_CNTL_TOUCH_START_EN，产生 START 信号，初始化一次测量操作。
- 配置 START 信号由触摸定时器触发：
  - 清零 RTC_CNTL_TOUCH_START_FORCE。
  - 配置定时器过期时间（单位：RTC_SLOW_CLK 周期），见 RTC_CNTL_TOUCH_SLEEP_CYCLES。
  - 置位 RTC_CNTL_TOUCH_SLP_TIMER_EN 使能定时器。

39.2.6.3 SCAN 模式

在 SCAN 模式下，Touch FSM 对多个触摸传感器按照一定顺序进行测量。每生成一个 START 信号，就选择一个触摸传感器进行测量。因此，利用多个 START 信号即可监控多个触摸管脚。具体的扫描过程见图 39.2-5。

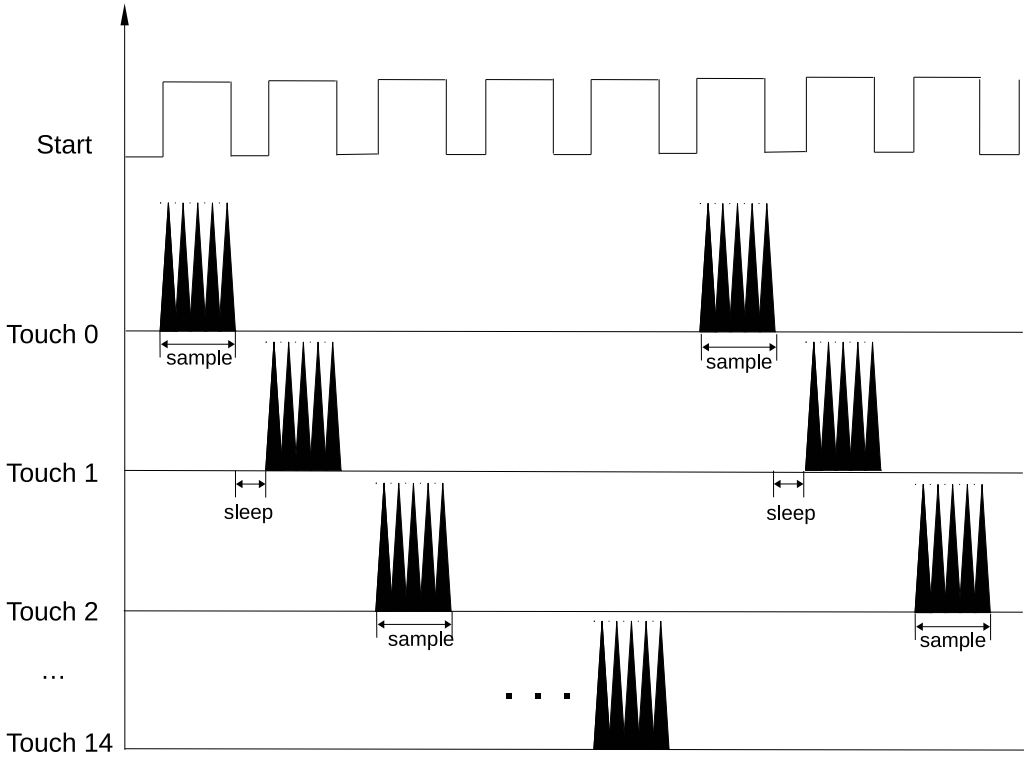


图 39.2-5. TOUCH SCAN 时序图

如需使能 SCAN 模式，请在寄存器 RTC_CNTL_TOUCH_SCAN_PAD_MAP 中指定已使能触摸管脚的位图。对于每个 START 信号，Touch FSM 将从位图中依次选择一个触摸管脚进行测量。一个 START 信号仅执行一次测量操作。经过多个 START 信号，Touch FSM 按照顺序依次对使能的触摸管脚进行测量。

注意：如果使用触摸定时器生成 START 信号，则整个扫描过程无需软件介入。

39.2.7 触摸检测

39.2.7.1 采样值

如章节 39.2.6.1 所述，用户可从 SENS_TOUCH_PAD*n*_DATA 中读取测量所得采样值。但 SENS_TOUCH_PAD*n*_DATA 中所存储的具体采样值类型由 SENS_TOUCH_DATA_SEL 决定。具体支持下面三种类型的采样值：

原始触摸数据 (touch_raw_data)：测量结束时，触摸计数器的计数值，表示对触摸管脚进行固定次数的充放电所需的时间，单位：RTC_SLOW_CLK 周期。

平滑触摸数据 (touch_smooth_data)：一系列来自某个触摸传感器的原始触摸数据经 IIR 滤波生成的移动平均线。平滑触摸数据不易被噪声或采样异常值干扰，因此常用于硬件触摸检测。生成平滑触摸数据的 IIR 滤波设置可在 RTC_CNTL_TOUCH_SMOOTH_LVL 中配置。具体信息见表 39.2-2。

基准数据 (benchmark)：同样由原始触摸数据经 IIR 滤波处理而生成。但基准数据的移动平均线窗口更广。因此基准数据常用于表示触摸管脚稳定读数，而不受触摸动作的影响。生成基准数据的 IIR 滤波设置可在 RTC_CNTL_TOUCH_FILTER_MODE 中配置，具体信息见表 39.2-3。

表 39.2-2. 平滑触摸数据算法

RTC_CNTL_TOUCH_SMOOTH_LVL	类型	公式
0	—	touch_raw_data
1	IIR 1/2	1/2 touch_raw_data + 1/2 touch_smooth_data
2	IIR 1/4	1/4 touch_raw_data + 3/4 touch_smooth_data
3	IIR 1/8	1/8 touch_raw_data + 7/8 touch_smooth_data

表 39.2-3. 基准数据算法

RTC_CNTL_TOUCH_FILTER_MODE	类型	公式
0	IIR 1/2	1/2 touch_raw_data + 1/2 benchmark
1	IIR 1/4	1/4 touch_raw_data + 3/4 benchmark
2	IIR 1/8	1/8 touch_raw_data + 7/8 benchmark
3	IIR 1/16	1/16 touch_raw_data + 15/16 benchmark
4	IIR 1/32	1/32 touch_raw_data + 31/32 benchmark
5	IIR 1/64	1/64 touch_raw_data + 63/64 benchmark
6	IIR 1/128	1/128 touch_raw_data + 127/128 benchmark
7	JITTER	touch_raw_data +/- RTC_CNTL_TOUCH_JITTER_STEP

39.2.7.2 硬件触摸检测

硬件触摸检测可检测到触摸动作或触摸释放动作，然后触发中断。如果采用硬件触摸检测，则不再需要指定触摸动作软件检测算法，也无需不断轮询采样值。

使用硬件触摸检测，需定义**触碰阈值** (finger_threshold) 和**噪声阈值** (active_noise_threshold)。当平滑触摸数据大于触碰阈值与**迟滞** (hysteresis) 的和，或小于触碰阈值与迟滞的差，将触发触摸中断。**触碰阈值**和**噪声阈值**均为基准数据的偏移，而不是绝对阈值。采用这种方法可以避免将原始触摸数据的渐变（缓慢变化）误测为触摸。多种环境因素，比如温度、电源、或噪声均有可能造成原始触摸数据的渐变。

- 触碰阈值由寄存器 SENS_TOUCH_OUT_TH*n* 配置。
- 噪声阈值由 RTC_CNTL_TOUCH_NOISE_THRES 配置，详见表 39.2-4。
- 迟滞由 RTC_CNTL_TOUCH_CONFIG3 配置，详见表 39.2-5。

表 39.2-4. 噪声算法

RTC_CNTL_TOUCH_NOISE_THRES	公式
0	4/8 finger_threshold
1	3/8 finger_threshold
2	2/8 finger_threshold
3	1/8 finger_threshold

表 39.2-5. 迟滞算法

RTC_CNTL_TOUCH_CONFIG3	公式
0	1/8 finger_threshold
1	3/32 finger_threshold
2	1/32 finger_threshold
3	0

39.2.8 噪声检测

触摸传感器 T0 未连接至 GPIO，即未连接至外部触摸板。因此，触摸传感器 T0 检测到的电容浮动可视为内部噪声。噪声检测功能使用触摸传感器 T0 用作噪声参考。当其它触摸传感器 T(N) (1 ~ 14) 执行测量操作时，触摸传感器 T0 也将同时进行测量。触摸传感器 T(N) 的采样值自动减去触摸传感器 0 的采样值，即可降低噪声带来的影响。

噪声检测过程如下：

- 用户可通过配置 RTC_CNTL_TOUCH_REFC 调节触摸传感器 T0 的参考电容，从而调节其驱动强度。
- 置位 RTC_CNTL_TOUCH_DENOISE_EN，其它触摸传感器 T(N) 开始执行测量操作时，触摸传感器 T0 同时开始测量。
- 最终的采样值为：

$$DATA(TOUCH[N]) - DATA(TOUCH0)$$

  其中，DATA(TOUCH[N]) 为触摸传感器 T(N) 的采样值，DATA(TOUCH0) 为触摸传感器 T0 的采样值，分辨率可配置为 12/10/8/4 位。具体配置见 RTC_CNTL_TOUCH_DENOISE_RES。

39.2.9 接近模式

物体靠近，但未接触触摸管脚时，触摸管脚电容将发生轻微变动，远小于物理触摸引起的电容变化。接近模式可用于检测电容轻微变动。

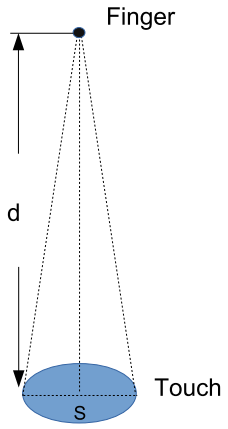


图 39.2-6. 感测面积示意图

- 检测距离 (d) 最大为 16 cm（感应面积 (S) 为 20 cm^2），如图 39.2-6 所示。
- 最多三个触摸管脚可同时配置为接近模式。

在接近模式下，触摸传感器进行固定次数的采样，然后累计采样值。如果最终累计采样值超出设定的阈值，则认为检测到了物体靠近，并触发中断。注意：由于是累计采样值，处于接近模式的触摸传感器不会生成跟章节 39.2.7.1 所述一样的数值，即原始触摸数据、平滑触摸数据和基准数据。

接近模式操作流程如下：

- 通过配置 SENS_TOUCH_APPROACH_PAD0、SENS_TOUCH_APPROACH_PAD1 或 SENS_TOUCH_APPROACH_PAD2，设置触摸传感器为接近模式。
- 通过配置 RTC_CNTL_TOUCH_APPROACH_MEAS_TIME，设置采样次数，用于生成累计值。触摸传感器内置采样计数器，记录采样次数。
- 配置寄存器 SENS_TOUCH_OUT_TH*n*，设置阈值。
- 采样计数器达到 RTC_CNTL_TOUCH_APPROACH_MEAS_TIME 设置的采样次数：
  - 如果累计值大于设定的阈值，则触发中断。
  - 采样计数器和累计值复位归零。触摸传感器重新开始累计采样值。

39.2.10 防潮功能和遇水保护功能

如果有小水珠，触摸传感器可能会误测为有触摸动作。**防潮设计**可缓和小水珠带来的影响。

如果传感器触摸板阵列变潮，即大部分触摸板已浸水，触摸管脚将无法继续检测触摸动作。**遇水保护设计**可检测到阵列已浸水，可关停传感器阵列。

39.2.10.1 防潮功能

触摸板上的小水珠，如果面积大到足以连通两个或更多触摸板，则可能导致临近触摸板出现电耦合现象。耦合现象导致电容变化，因此耦合的触摸板可能会误测触摸动作。用户可配置使用防潮功能。具体设置如下：

- 设置触摸传感器 T14 的驱动强度，见 RTC_CNTL_TOUCH_BUFDRV；
- 使能触摸传感器 T14 用于防潮功能，见 RTC_CNTL_TOUCH_SHIELD_PAD_EN。

39.2.10.2 遇水保护功能

传感器阵列遇水后，全部或大部分触摸管脚因存在误测触摸动作的可能，因此不可用。

用户可配置 RTC_CNTL_TOUCH_OUT_RING 选择用于遇水保护功能的触摸管脚。

39.3 SAR ADC

39.3.1 概述

ESP32-S3 内置了两个 12 位的 SAR ADC，可测量最多来自 20 个管脚的模拟信号。图 39.3-1 所示为 SAR ADC 概图。

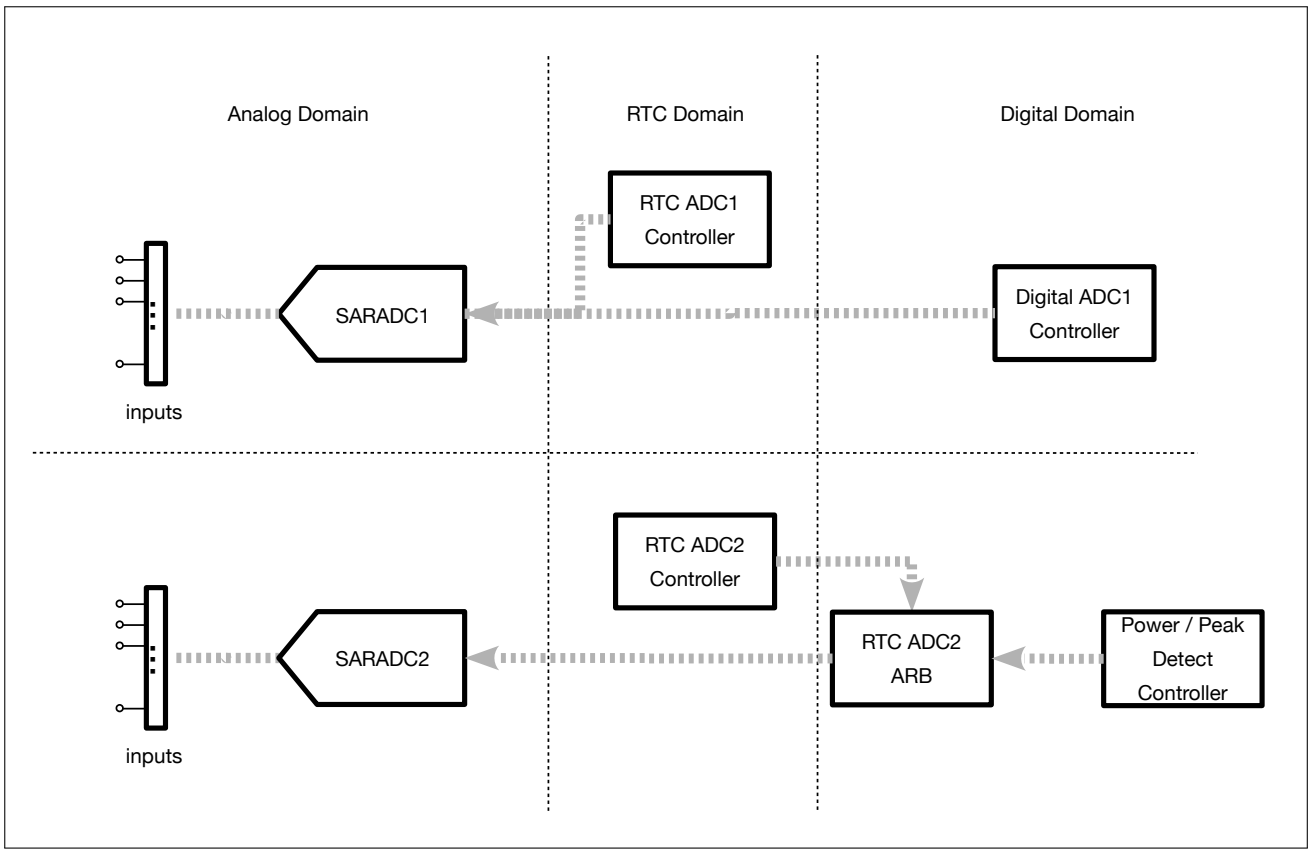


图 39.3-1. SAR ADC 概图

如图 39.3-1 所示，SAR ADC 由四个专用控制器控制，包括：

- 一个数字控制器，支持高性能多通道扫描和 DMA 连续转换，包括：
  - DIG ADC1 控制器 (Digital ADC1 Controller)
- 两个 RTC 控制器，支持在低功耗模式下工作和单次转换，包括：
  - RTC ADC1 控制器 (RTC ADC1 Controller)
  - RTC ADC2 控制器 (RTC ADC2 Controller)
- 一个内部控制器，即 PWDET 控制器 (Power/Peak Detect Controller)，用于 RF 功率检测。注意，此控制器仅供 RF 内部使用。

说明：
ESP32-S3 的 DIG ADC2 控制器无法正常工作，该章节中已删除其相关信息，详见 ESP32-S3 系列芯片勘误表。

39.3.2 主要特性

SAR ADC 具有以下特性：

- 四个控制器均配有独立的 ADC Reader 模块，见图 39.3-2
- 支持 DIG ADC1 控制器和 RTC ADC1 控制器通过软件选择的方式获取 SAR ADC1 的控制权

- 支持 RTC ADC2 控制器和 PWDET 控制器通过仲裁的方式按照指定的仲裁方式，轮流获取 SAR ADC2 的控制权
- 支持 12 位采样分辨率
- 支持采集最多 20 个管脚上的模拟电压
- RTC ADC1/2 控制器：
  - 支持单次转换
  - 支持在低功耗模式下工作，如 Deep-sleep
  - 可由 ULP 协处理器配置
- DIG ADC1 控制器：
  - 配有多通道扫描控制模块，支持多通道扫描模式
  - 提供模式控制模块，支持单 SAR ADC 采样模式
  - 在多通道扫描模式下，支持自定义扫描通道顺序
  - 提供两个滤波器，滤波系数可配
  - 支持阈值监控，采样值大于设置的高阈值或小于设置的低阈值将产生中断
  - 支持 DMA
- PWDET 控制器：用于 RF 功率检测（仅供内部使用）

39.3.3 SAR ADC 架构

SAR ADC 的主要元件与连接情况见图 39.3-2。

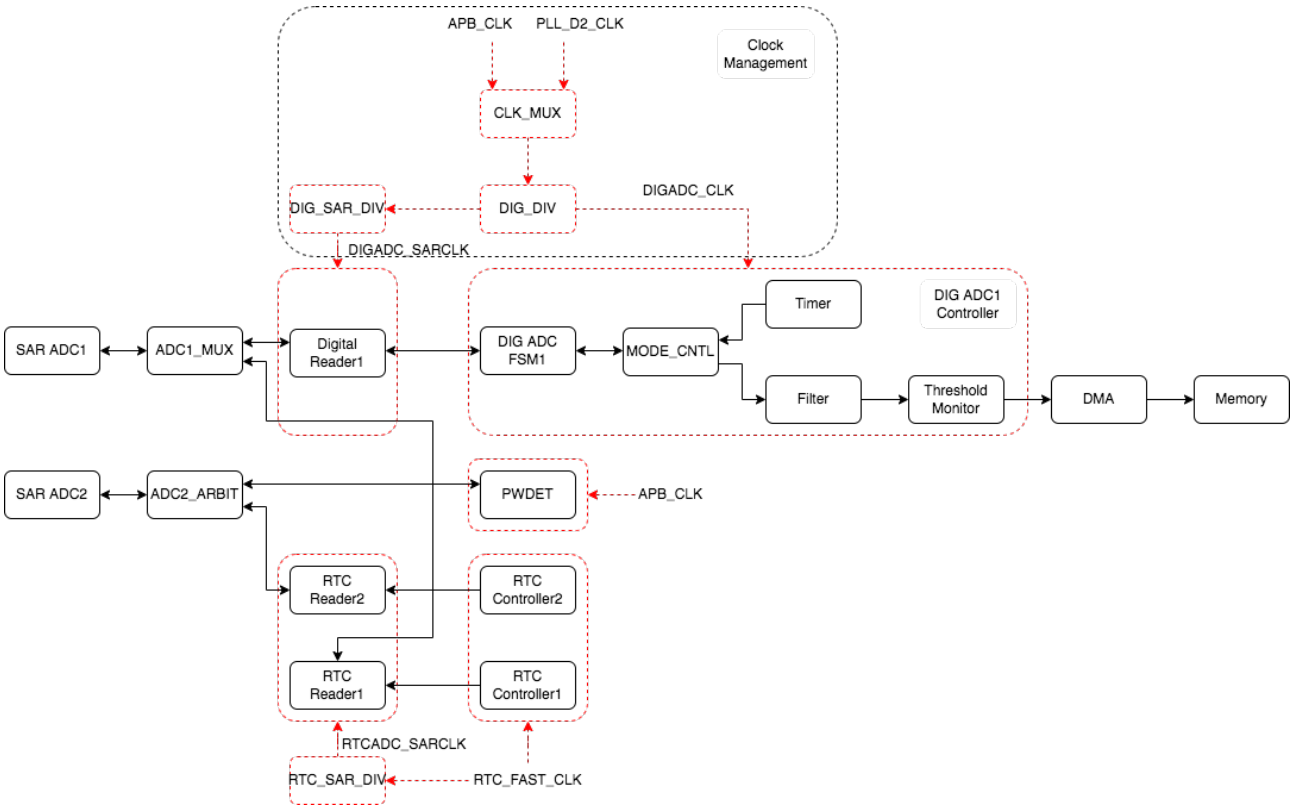


图 39.3-2. SAR ADC 的功能概况

- —>：时钟信号
-  ：时钟分频器、多路器、以及时钟作用的模块

如图 39.3-2 所示，SAR ADC 模块主要包括以下部分：

- SAR ADC1：可对 10 个通道进行电压检测；
- SAR ADC2：可对 10 个通道进行电压检测；
- 时钟管理 (Clock Management)：对时钟源进行选择和分频：
  - DIG ADC1 控制器时钟源：可选择 APB_CLK 或 PLL_D2_CLK；
  - DIG ADC1 控制器分频时钟：
    * DIGADC_SARCLK：SAR ADC1、SAR ADC2、Digital Reader1 的工作时钟；其中控制 DIGADC_SARCLK 分频的 DIG_SAR_DIV 的分频系数至少是 2 分频，且 DIGADC_SARCLK 的频率不能超过 5 MHz，具体配置见 APB_SARADC_SAR_CLK_DIV；
    * DIGADC_CLK：DIG ADC FSM1 的工作时钟。
  - RTC ADC 控制器时钟源：RTC_FAST_CLK
  - RTC ADC 控制器分频时钟：
    * RTCADC_SARCLK：SAR ADC1、SAR ADC2、RTC Reader1 和 RTC Reader2 的工作时钟；其中控制 RTCADC_SARCLK 分频的 RTC_SAR_DIV 的分频系数至少是 2 分频，且 RTCADC_SARCLK 的频率不能超过 5 MHz；
- 仲裁器 (ADC2_ARBIT)：用于选择使用 SAR ADC2 的控制器，可以是 RTC ADC2 控制器或 PWDET 控制器。仲裁器不仅仲裁控制信号，还会根据授权的控制器选择 SAR ADC2 的时钟。

- 定时器 (Timer)：DIG ADC1 专用定时器，可发起采样使能信号。
- DIG ADC FSM1：用于
  - 接收定时器提供的采样使能信号
  - 根据样式表生成 SAR ADC1 的采样配置
  - 驱动 Digital Reader1 模块读取 ADC 采样值
  - 并将采样值传输到滤波器、存储器等后续数据处理模块
- Filter：滤波器 0 和滤波器 1，可对目标通道的采样数据进行滤波处理。
- Threshold Monitor：阈值监控器 0 和阈值监控器 1。可在采样值大于设定的高阈值，或小于设定的低阈值时触发中断。
- 模式控制模块 (MODE CNTL)：用于过滤定时器触发的采样信号，支持单 SAR ADC 采样模式。
- Digital Reader1：由 DIG ADC FSM1 驱动，读取 SAR ADC1 的数值。
- RTC Controller1/2：提供采样使能信号，根据配置驱动 RTC Reader1/2 模块读取 SAR ADC1/2 的采样值，并存储采样数据。
- RTC Reader1/2：由 RTC Controller1/2 驱动，读取 SAR ADC1/2 的数值。

39.3.4 输入信号

SAR ADC 需首先通过内部多路器选择待测量的模拟管脚，然后才能采样模拟信号。表 39.3-1 列出了所有可能需要经过 SAR ADC1 或 SAR ADC2 处理的模拟信号。

表 39.3-1. SAR ADC 的信号输入

管脚/信号	通道	ADC 选择
GPIO1	0	SAR ADC1
GPIO2	1	
GPIO3	2	
GPIO4	3	
GPIO5	4	
GPIO6	5	
GPIO7	6	
GPIO8	7	
GPIO9	8	
GPIO10	9	
GPIO11	0	SAR ADC2
GPIO12	1	
GPIO13	2	
GPIO14	3	
GPIO15	4	
GPIO16	5	
GPIO17	6	
GPIO18	7	
GPIO19	8	

管脚/信号	通道	ADC 选择
GPIO20	9	

39.3.5 ADC 转换和衰减

SAR ADC 转换模拟信号时，转换分辨率（12 位）电压范围为 0 mV ~ V_{ref}。其中，V_{ref} 为 SAR ADC 内部参考电压，出厂设定为 1100 mV。因此，转换结果 (data) 可以使用以下公式转换成模拟电压输出 V_{data}：

$$V_{data} = \frac{V_{ref}}{k} \times \frac{data}{4095}$$

其中 k 为衰减对应的系数。

如需转换大于 V_{ref} 的电压，信号输入 SAR ADC 前可进行衰减。支持的衰减等级包括：

- 0 dB (k≈100%)
- 2.5 dB (k≈75%)
- 6 dB (k≈50%)
- 12 dB (k≈25%)

39.3.6 RTC ADC 控制器

RTC 电源域中的 SAR ADC 控制器（RTC ADC1 控制器和 RTC ADC2 控制器）可在低频状态下提供最小功耗 ADC 测量。

RTC ADC1/2 控制器的具体功能概况见图 39.3-3。转换是由 SENS_SAR_MEAS*n*_START_SAR 触发，测量结果见 SENS_SAR_MEAS*n*_DATA_SAR。

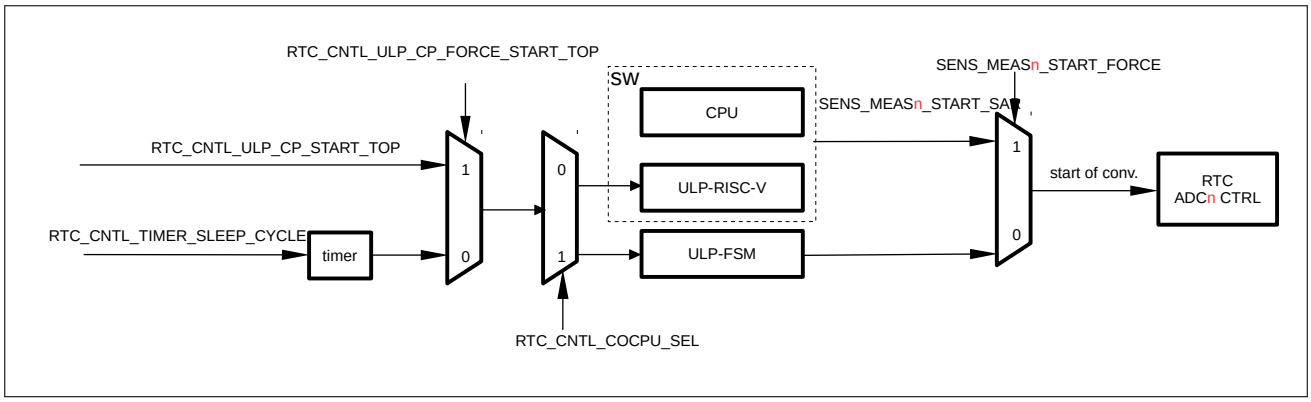


图 39.3-3. RTC SAR ADC 的功能概况

ULP 协处理器与控制器之间的关系非常紧密，已经内置了指令来使用 ADC。很多情况下，控制器均需要与 ULP 协处理器协同工作，例如：

- 可在 Deep-sleep 模式下对通道进行周期性检测。Deep-sleep 模式下，只能通过 ULP 协处理器触发 ADC 采样。ULP 协处理器可以通过配置 RTC 外设的寄存器，触发 ADC。
- 可按一定顺序对通道进行连续性扫描。尽管控制器无法支持连续性扫描或 DMA，但协处理器可通过软件的方式协助实现这部分功能。

用户可配置 SENS_SAR1/2_EN_PAD 来选择采样通道。如果选用 ULP 专用指令触发，则需要清空 SENS_SAR1/2
_EN_PAD_FORCE 选择由 ULP 指令使能采样通道。

39.3.7 DIG ADC 控制器

DIG ADC1 控制器使用快速时钟，实现了采样速率大幅提升。控制器最高支持 12 位采样分辨率，支持专用定时器触发的多通道扫描。更多参数和性能信息见 《ESP32-S3 系列芯片技术规格书》 中的 ADC 特性章节。

如果选择专用定时器驱动的多通道扫描，可采用如下配置。注意，在多通道扫描模式下，扫描序列可根据样式表的描述进行，样式表可配置。

- 配置 APB_SARADC_TIMER_TARGET 设置 DIG ADC 定时器的触发周期。当计数器计数到配置周期数的 2 倍时，触发采样。计数器的工作时钟见章节 39.3.7.1；
- 配置 APB_SARADC_TIMER_EN 使能定时器；
- 定时器超时则将驱动 DIG ADC FSM 根据样式表进行采样；
- 数据通过 DMA 自动存储到内存空间中，扫描完成将产生中断。

39.3.7.1 DIG ADC 控制器时钟

用户可配置 APB_SARADC_CLK_SEL 选择 DIG ADC1 控制器的时钟源：

- 0：关闭时钟；
- 1：选择 PLL_D2_CLK 用作时钟源。此时 DIG ADC1 的工作时钟为该时钟源的分频时钟：DIGADC_CLK；
- 2：选择 APB_CLK 用作时钟源。

如果选择使用 DIGADC_CLK，用户可配置 APB_SARADC_CLKM_DIV_NUM 选择分频系数。

注意，由于 SAR ADC 有速度限制，所以 Digital Reader1（包括 SAR ADC1）的工作时钟是 DIGADC_SARCLK。DIGADC_SARCLK 频率会影响采样精度。当 DIGADC_SARCLK 的频率高于 5 MHz 时，采样精度会变低。DIGADC_SARCLK 由 DIGADC_CLK 经过专用分频器分频所得。分频系数通过 APB_SARADC_SAR_CLK_DIV 配置。ADC 每采样一个数据需要 25 个 DIGADC_SARCLK 时钟周期数，所以最大采样速率受到 DIGADC_SARCLK 的频率限制。更多时钟信息，见章节 7 复位和时钟。

39.3.7.2 DMA 支持

DIG ADC1 控制器允许通过外设 DMA 实现直接内存访问，由 DIG ADC1 专用定时器产生触发信号。用户可通过软件配置 APB_SARADC_APB_ADC_TRANS 将 DMA 的数据通路切换到 DIG ADC。关于 DMA 的具体配置，请参考章节 3 通用 *DMA* 控制器 *(GDMA)*。

39.3.7.3 DIG ADC FSM

DIG ADC FSM1 按照样式表指定顺序和通道号循环使用 SAR ADC1：

- DIG ADC FSM1 接收定时器发出的采样使能信号；
- 然后根据当前样式表指针 PR 所指的通道和衰减信息向 SAR ADC1 发起采样请求；
- 采样完成后，更新指针 PR：

- 如果当前指针 PR 达到配置的 APB_SARADC_SAR1_PATT_LEN，则 PR 归零，重新采样样式表第一项的指令。
- 否则，PR 加 1，指向下一个样式指令。

39.3.7.4 样式表结构

DIG ADC FSM1 包含一个独立样式表，由 APB_SARADC_SAR1_PATT_TAB*x*_REG 配置，其中 *x* 表示样式表的寄存器序号 1 ~ 4，如下图所示：

(reserved)	cmd0	cmd1	cmd2	cmd3
31 ... 24	23 ... 18	17 ... 12	11 ... 6	5 ... 0
0 0 0 0 0 0 0 0	0x0000	0x0000	0x0000	0x0000

cmd *n* (*n* = 0 - 3) 表示样式表中的样式，即样式 0 ~ 样式 3。

图 39.3-4. APB_SARADC_SAR1_PATT_TAB1_REG 与样式 0 - 3

(reserved)	cmd4	cmd5	cmd6	cmd7
31 ... 24	23 ... 18	17 ... 12	11 ... 6	5 ... 0
0 0 0 0 0 0 0 0	0x0000	0x0000	0x0000	0x0000

cmd *n* (*n* = 4 - 7) 表示样式表中的样式，即样式 4 ~ 样式 7。

图 39.3-5. APB_SARADC_SAR1_PATT_TAB2_REG 与样式 4 - 7

(reserved)	cmd11	cmd10	cmd9	cmd8
31 ... 24	23 ... 18	17 ... 12	11 ... 6	5 ... 0
0 0 0 0 0 0 0 0	0x0000	0x0000	0x0000	0x0000

cmd *n* (*n* = 8 - 11) 表示样式表中的样式，即样式 8 ~ 样式 11。

图 39.3-6. APB_SARADC_SAR1_PATT_TAB3_REG 与样式 8 - 11

(reserved)	cmd15	cmd14	cmd13	cmd12
31 ... 24	23 ... 18	17 ... 12	11 ... 6	5 ... 0
0 0 0 0 0 0 0 0	0x0000	0x0000	0x0000	0x0000

cmd *n* (*n* = 12 - 15) 表示样式表中的样式，即样式 12 ~ 样式 15。

图 39.3-7. APB_SARADC_SAR1_PATT_TAB4_REG 与样式 12 - 15

每个寄存器包含四个样式，每个样式长度为六位，共包括 2 个字段，分别存储通道和衰减信息，具体见图 39.3-8。

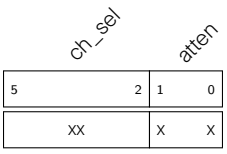


图 39.3-8. 样式表中的样式结构

atten 衰减配置信息。0：0 dB；1：2.5 dB；2：6 dB；3：12 dB。

ch_sel 扫描通道选择信息，更多信息见表 39.3-1。

39.3.7.5 多通道扫描配置示例

例如，希望实现如下所示的 SAR ADC1 多通道扫描方式：

- 扫描 SAR ADC1 的通道 2，且衰减配置为 12 dB（见图 39.3-9）；
- 扫描 SAR ADC1 的通道 0，且衰减配置为 2.5 dB（见图 39.3-10）。

则具体的配置如下：

- 配置 SAR ADC1
  - 配置 SAR ADC1 第一个样式 cmd0（即 APB_SARADC_SAR1_PATT_TAB1_REG[5:0]），如下图所示：

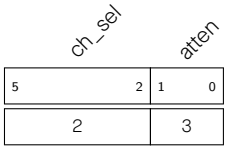


图 39.3-9. SAR ADC1 cmd0 配置示例

atten 配置该字段的值为 3，即衰减配置为 12 dB。

ch_sel 配置该字段的值为 2，即选择通道 2（见表 39.3-1）。

  - 配置 SAR ADC1 第二个样式 cmd1（即 APB_SARADC_SAR1_PATT_TAB1_REG[11:6]），如下图所示：

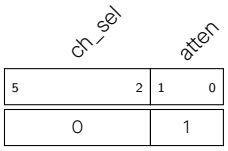


图 39.3-10. SAR ADC1 cmd1 配置示例

atten 配置该字段的值为 1，即衰减配置为 2.5 dB。

ch_sel 配置该字段的值为 0，即选择通道 0（见表 39.3-1）。

- 配置 APB_SARADC_SAR1_PATT_LEN 为 1，即选择使用上述配置好的 SAR ADC1 样式 0 和样式 1；
- 使能定时器，则 DIG ADC1 控制器将根据上述样式配置，周期性采样 SAR ADC1 的通道 2 和通道 0。

39.3.7.6 DMA 数据格式

ADC 最终向 DMA 传递 32 位数据，如下图所示：

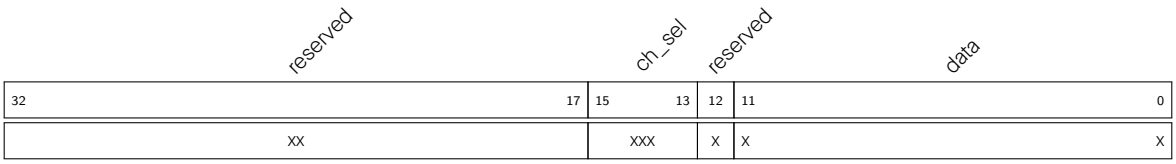


图 39.3-11. DMA 数据格式

data ADC 转换结果，12 位

ch_sel 通道信息，3 位

39.3.7.7 ADC 滤波器

DIG ADC1 控制器支持滤波功能，提供两个滤波器，可配置给 SAR ADC1 的任意两个通道，然后对目标通道的采样数据进行滤波。滤波公式如下所示：

$$data_{cur} = \frac{(k-1)data_{prev}}{k} + \frac{data_{in}}{k} + 0.5$$

- $data_{cur}$：滤波后数据
- $data_{in}$：ADC 采样值
- $data_{prev}$：上次滤波数据
- k：滤波系数

配置滤波器如下：

- 配置 APB_SARADC_FILTER_CHANNEL*x* 设置滤波器 *x* 作用的 ADC 通道
- 配置 APB_SARADC_FILTER_FACTOR*x* 设置滤波器 *x* 的滤波系数

注意，这里的 *x* 为滤波器编号：*x* 为 0 表示滤波器 *0*；为 1 表示滤波器 *1*。

39.3.7.8 阈值监控

DIG ADC1 包含两个阈值监控器，可配置到 SAR ADC1 的任意通道上。当 ADC 采样值大于设定的高阈值，则触发高阈值中断；若采样值小于设定的低阈值，则触发低阈值中断。

阈值监控配置如下：

- 配置 APB_SARADC_THRES*x*_EN 使能阈值监控 *x* 的功能；
- 配置 APB_SARADC_THRES*x*_LOW 设置低阈值；
- 配置 APB_SARADC_THRES*x*_HIGH 设置高阈值；
- 配置 APB_SARADC_THRES*x*_CHANNEL 设置监控的 SAR ADC 以及通道。

注意，这里的 *x* 为阈值监控器编号：*x* 为 0 表示阈值监控器 *0*；为 1 表示阈值监控器 *1*。

39.3.8 SAR ADC2 仲裁器

SAR ADC2 可选两种控制器：RTC ADC2 控制器或 PWDET 控制器。为防止出现冲突，同时提升 SAR ADC2 的使用效率，ESP32-S3 提供了 SAR ADC2 的访问仲裁。仲裁器有公平仲裁和固定优先级仲裁两种模式可供选择。

- 公平仲裁模式，即循环优先级仲裁。清零 APB_SARADC_ADC_ARB_FIX_PRIORITY 即可进入公平仲裁模式。
- 固定优先级仲裁模式下，配置 APB_SAR ADC_ADC_ARB_RTC_PRIORITY（RTC ADC2 控制器）或 APB_SARADC_ADC_ARB_WIFI_PRIORITY（PWDET 控制器）可分别配置对应控制器的优先级。值越大，优先级越高。

仲裁器规定，无论低优先级控制器是否已开始转换数据，高优先级控制器均可随时开始自己的数据转换。如果 ADC 已经接受低优先级控制器的转换请求，开始转换数据，但高优先级控制器也需要转换数据，则可以中断或终止低优先级控制器的数据转换，开始高优先级控制器的数据转换。如果已经有高优先级控制器正在进行数据转换，而低优先级控制器则无法启动数据转换。

转换中断或转换启动失败返回的转换数据无效，因此在返回的转换结果中增加数据标记位，表示转换是否有效。

- RTC ADC2 CTRL 的数据标记位为 SENS_MEAS2_DATA_SAR 的高两位
  - 2'b10：数据转换中断
  - 2'b01：数据转换启动失败
  - 2'b00：转换数据有效
- PWDET 控制器的数据标记位为采样结果的高两位
  - 2'b10：数据转换中断
  - 2'b01：数据转换启动失败
  - 2'b00：转换数据有效

除了上述两种模式以外，用户可以通过配置 APB_SARADC_ADC_ARB_GRANT_FORCE 屏蔽仲裁器，配置 APB_SARADC_ADC_ARB_WIFI_FORCE 或 APB_SARADC_ADC_ARB_RTC_FORCE 决定授权控制器。

说明：

- 屏蔽仲裁器时，只能有一个 APB_SARADC_ADC_ARB_XXX_FORCE 位配置为 1。
- 仲裁器的工作时钟为 APB_CLK，当 APB_CLK 时钟降低到 8 MHz 及以下时，必须屏蔽仲裁器。
- 进入睡眠时，需要配置寄存器 SENS_SAR_MEAS2_MUX_REG 的 SENS_SAR2_RTC_FORCE 位为 1，屏蔽仲裁器以及除 RTC 控制器以外的所有控制器信号。

39.4 温度传感器

39.4.1 概述

ESP32-S3 搭载了温度传感器可以实时监测温度。

39.4.2 主要特性

温度传感器的主要特性包括：

- 支持在低功耗状态下，通过 ULP 协处理器实时监控

- 支持软件和 ULP 触发
- 可根据使用环境配置温度偏移，提高测试精度
- 测量范围可调节

39.4.3 功能描述

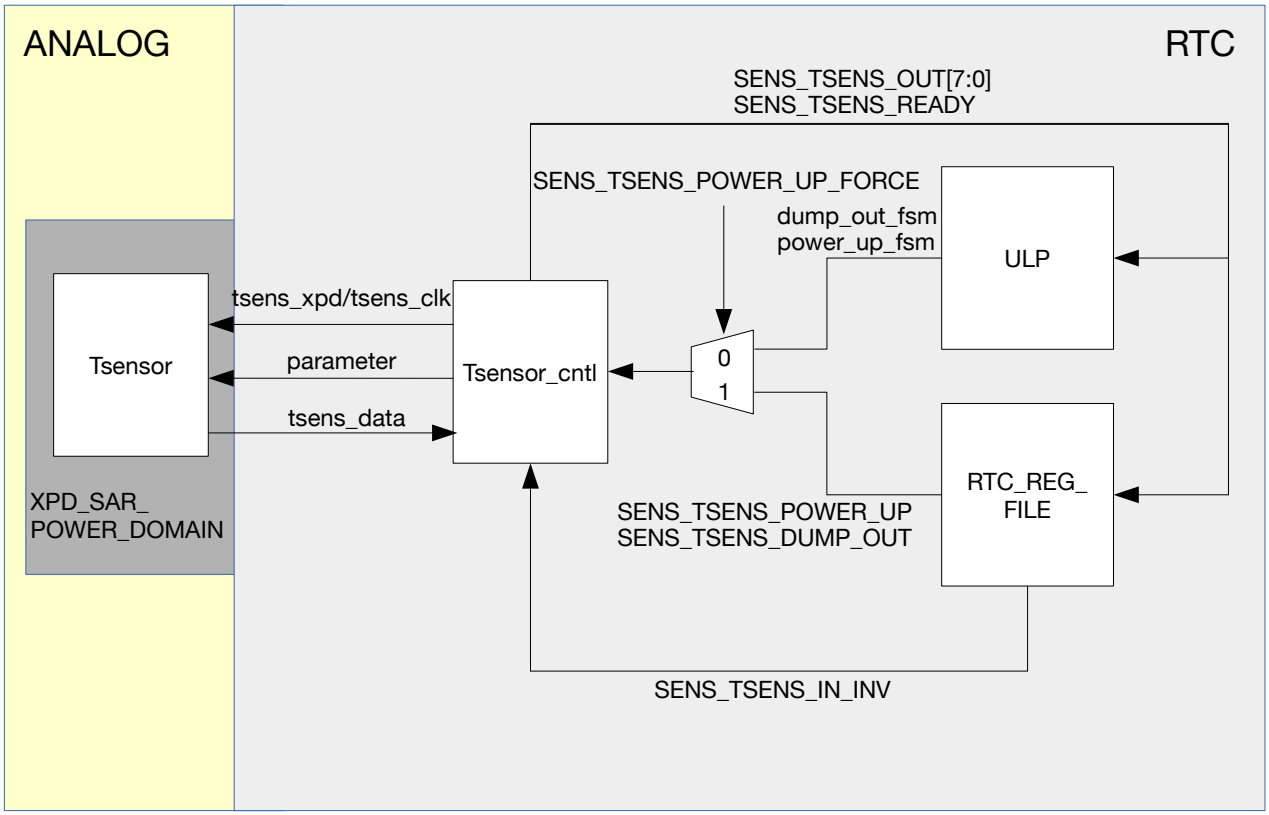


图 39.4-1. 温度传感器结构

如图 39.4-1 所示，温度传感器的触发方式分为软件启动和 ULP-FSM 启动：

- 软件启动，即由 CPU 或 ULP-RISC-V 通过配置寄存器启动：
  - 置位 SENS_TSENS_POWER_UP_FORCE、SENS_TSENS_POWER_UP 使能温度传感器；
  - 置位 SENS_FORCE_XPD_SAR 强制开启 SAR ADC 电源，置位 SENS_TSENS_CLK_EN 使能温度传感器时钟；
  - 等待一段时间后（输出值会随着测量时间的增加而逐渐线性逼近真实的温度值），配置 SENS_TSENS_DUMP_OUT；
  - 等待 SENS_TSENS_READY，读取 SENS_TSENS_OUT 获取转换结果。
- ULP-FSM 启动：
  - 清空 SENS_TSENS_POWER_UP_FORCE；
  - ULP-FSM 内置高度集成的温度采集指令，所以只需要执行该指令即可完成温度采样，详见章节 *2 超低功耗协处理器 (ULP-FSM, ULP-RISC-V)*。

温度传感器的输出值需要使用转换公式转换成实际的温度值 (°C)。转换公式如下：

$$T(°C) = 0.4386 * VALUE - 27.88 * offset - 20.52$$

其中 VALUE 即温度传感器的输出值，offset 由温度偏移决定。温度传感器在不同的实际使用环境（测量温度范围）下，温度偏移不同，见表 39.4-1。

表 39.4-1. 温度测量范围与温度偏移

测量范围 (°C)	温度偏移
50 ~ 125	-2
20 ~ 100	-1
-10 ~ 80	0
-30 ~ 50	1
-40 ~ 20	2

39.5 中断

- APB_SARADC_THRES*x*_HIGH_INT：超过阈值监控器 *x* 的高阈值，即触发此中断；
- APB_SARADC_THRES*x*_LOW_INT：低于阈值监控器 *x* 的低阈值，即触发此中断。
- APB_SARADC_ADC1_DONE_INT：SAR ADC1 完成一次转换，即触发此中断；

更多 ULP-RISC-V 中断信息，请参考章节 *2 超低功耗协处理器 (ULP-FSM, ULP-RISC-V)*中的 ULP-RISC-V 中断部分。

39.6 寄存器列表

- SENSOR (ALWAYS_ON) 为非掉电寄存器，寄存器不会随着 RTC_PERI 的电源域掉电而复位。更多信息，见章节 *10 低功耗管理 (RTC_CNTL)*。
- SENSOR (RTC_PERI) 处于 RTC_PERI 电源域下，如果 RTC_PERI 的电源域掉电，寄存器值将会被复位。更多信息，见章节 *10 低功耗管理 (RTC_CNTL)*。
- SENSOR (DIG_PERI) 处于数字系统的电源域下，如果数字系统的电源域掉电，寄存器值将会被复位。更多信息，见章节 *10 低功耗管理 (RTC_CNTL)*。

39.6.1 SENSOR (ALWAYS_ON) 寄存器列表

本小节的所有地址均为相对于 [低功耗管理基地址] 的地址偏移量（相对地址），具体基地址请见章节 *4 系统和存储器* 中的表 4.3-3。

名称	描述	地址	访问
触摸控制寄存器			
RTC_CNTL_TOUCH_CTRL1_REG	触摸控制寄存器 1	0x0108	R/W
RTC_CNTL_TOUCH_CTRL2_REG	触摸控制寄存器 2	0x010C	R/W
RTC_CNTL_TOUCH_SCAN_CTRL_REG	触摸扫描配置寄存器	0x0110	R/W
RTC_CNTL_TOUCH_SLP_THRES_REG	睡眠管脚配置寄存器	0x0114	R/W
RTC_CNTL_TOUCH_APPROACH_REG	接近模式配置寄存器	0x0118	varies

名称	描述	地址	访问
RTC_CNTL_TOUCH_FILTER_CTRL_REG	噪声过滤配置寄存器	0x011C	R/W
RTC_CNTL_TOUCH_TIMEOUT_CTRL_REG	噪声过滤超时配置寄存器	0x0124	R/W
RTC_CNTL_TOUCH_DAC_REG	触摸传感器的斜率配置寄存器	0x014C	R/W
RTC_CNTL_TOUCH_DAC1_REG	触摸传感器的斜率配置寄存器 1	0x0150	R/W

39.6.2 SENSOR (RTC_PERI) 寄存器列表

本小节的所有地址均为相对于 [低功耗管理基地址 + 0x800] 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
配置寄存器			
SENS_SAR_READER1_CTRL_REG	SAR ADC1 数据和采样控制寄存器	0x0000	R/W
SENS_SAR_MEAS1_CTRL2_REG	SAR ADC1 转换状态控制寄存器	0x000C	varies
SENS_SAR_MEAS1_MUX_REG	SAR ADC1 控制器选择寄存器	0x0010	R/W
SENS_SAR_ATTEN1_REG	SAR ADC1 衰减配置寄存器	0x0014	R/W
SENS_SAR_READER2_CTRL_REG	SAR ADC2 数据和采样控制寄存器	0x0024	R/W
SENS_SAR_MEAS2_CTRL2_REG	SAR ADC2 转换状态控制寄存器	0x0030	varies
SENS_SAR_MEAS2_MUX_REG	SAR ADC2 控制器选择寄存器	0x0034	R/W
SENS_SAR_ATTEN2_REG	SAR ADC2 衰减配置寄存器	0x0038	R/W
SENS_SAR_POWER_XPD_SAR_REG	SAR ADC 电源控制寄存器	0x003C	R/W
SENS_SAR_TSENS_CTRL_REG	温度传感器数据控制寄存器	0x0050	varies
SENS_SAR_TOUCH_CONF_REG	触摸传感器配置寄存器	0x005C	varies
SENS_SAR_TOUCH_DENOISE_REG	去噪数据寄存器	0x0060	RO
SENS_SAR_TOUCH_THRES1_REG	触摸管脚 1 的阈值配置寄存器	0x0064	R/W
SENS_SAR_TOUCH_THRES2_REG	触摸管脚 2 的阈值配置寄存器	0x0068	R/W
SENS_SAR_TOUCH_THRES3_REG	触摸管脚 3 的阈值配置寄存器	0x006C	R/W
SENS_SAR_TOUCH_THRES4_REG	触摸管脚 4 的阈值配置寄存器	0x0070	R/W
SENS_SAR_TOUCH_THRES5_REG	触摸管脚 5 的阈值配置寄存器	0x0074	R/W
SENS_SAR_TOUCH_THRES6_REG	触摸管脚 6 的阈值配置寄存器	0x0078	R/W
SENS_SAR_TOUCH_THRES7_REG	触摸管脚 7 的阈值配置寄存器	0x007C	R/W
SENS_SAR_TOUCH_THRES8_REG	触摸管脚 8 的阈值配置寄存器	0x0080	R/W
SENS_SAR_TOUCH_THRES9_REG	触摸管脚 9 的阈值配置寄存器	0x0084	R/W
SENS_SAR_TOUCH_THRES10_REG	触摸管脚 10 的阈值配置寄存器	0x0088	R/W
SENS_SAR_TOUCH_THRES11_REG	触摸管脚 11 的阈值配置寄存器	0x008C	R/W
SENS_SAR_TOUCH_THRES12_REG	触摸管脚 12 的阈值配置寄存器	0x0090	R/W
SENS_SAR_TOUCH_THRES13_REG	触摸管脚 13 的阈值配置寄存器	0x0094	R/W
SENS_SAR_TOUCH_THRES14_REG	触摸管脚 14 的阈值配置寄存器	0x0098	R/W
SENS_SAR_TOUCH_CHN_ST_REG	触摸传感器通道状态寄存器	0x009C	varies
SENS_SAR_PERI_CLK_GATE_CONF_REG	RTC 外设时钟门控寄存器	0x0104	R/W
SENS_SAR_PERI_RESET_CONF_REG	RTC 外设复位寄存器	0x0108	R/W
状态寄存器			

名称	描述	地址	访问
SENS_SAR_TOUCH_STATUS0_REG	触摸扫描状态寄存器	0x00A0	RO
SENS_SAR_TOUCH_STATUS1_REG	触摸管脚 1 的通道状态寄存器	0x00A4	RO
SENS_SAR_TOUCH_STATUS2_REG	触摸管脚 2 的通道状态寄存器	0x00A8	RO
SENS_SAR_TOUCH_STATUS3_REG	触摸管脚 3 的通道状态寄存器	0x00AC	RO
SENS_SAR_TOUCH_STATUS4_REG	触摸管脚 4 的通道状态寄存器	0x00B0	RO
SENS_SAR_TOUCH_STATUS5_REG	触摸管脚 5 的通道状态寄存器	0x00B4	RO
SENS_SAR_TOUCH_STATUS6_REG	触摸管脚 6 的通道状态寄存器	0x00B8	RO
SENS_SAR_TOUCH_STATUS7_REG	触摸管脚 7 的通道状态寄存器	0x00BC	RO
SENS_SAR_TOUCH_STATUS8_REG	触摸管脚 8 的通道状态寄存器	0x00C0	RO
SENS_SAR_TOUCH_STATUS9_REG	触摸管脚 9 的通道状态寄存器	0x00C4	RO
SENS_SAR_TOUCH_STATUS10_REG	触摸管脚 10 的通道状态寄存器	0x00C8	RO
SENS_SAR_TOUCH_STATUS11_REG	触摸管脚 11 的通道状态寄存器	0x00CC	RO
SENS_SAR_TOUCH_STATUS12_REG	触摸管脚 12 的通道状态寄存器	0x00D0	RO
SENS_SAR_TOUCH_STATUS13_REG	触摸管脚 13 的通道状态寄存器	0x00D4	RO
SENS_SAR_TOUCH_STATUS14_REG	触摸管脚 14 的通道状态寄存器	0x00D8	RO
SENS_SAR_TOUCH_STATUS15_REG	睡眠管脚的通道状态寄存器	0x00DC	RO
SENS_SAR_TOUCH_APPR_STATUS_REG	接近模式管脚的通道状态寄存器	0x00E0	RO

39.6.3 SENSOR (DIG_PERI) 寄存器列表

本小节的所有地址均为相对于 ADC 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

请查看章节 寄存器的访问类型，了解“访问”列缩写的含义。

名称	描述	地址	访问
配置寄存器			
APB_SARADC_CTRL_REG	DIG ADC 控制器配置寄存器 1	0x0000	R/W
APB_SARADC_CTRL2_REG	DIG ADC 控制器配置寄存器 2	0x0004	R/W
APB_SARADC_FILTER_CTRL1_REG	SAR ADC 滤波器配置寄存器 1	0x0008	R/W
APB_SARADC_SAR1_PATT_TAB1_REG	SAR ADC1 样式表寄存器 1	0x0018	R/W
APB_SARADC_SAR1_PATT_TAB2_REG	SAR ADC1 样式表寄存器 2	0x001C	R/W
APB_SARADC_SAR1_PATT_TAB3_REG	SAR ADC1 样式表寄存器 3	0x0020	R/W
APB_SARADC_SAR1_PATT_TAB4_REG	SAR ADC1 样式表寄存器 4	0x0024	R/W
APB_SARADC_APB_ADC_ARB_CTRL_REG	SAR ADC2 仲裁器配置寄存器	0x0038	R/W
APB_SARADC_FILTER_CTRL0_REG	SAR ADC 滤波器配置寄存器 0	0x003C	R/W
APB_SARADC_THRES0_CTRL_REG	采样阈值控制寄存器 0	0x0044	R/W
APB_SARADC_THRES1_CTRL_REG	采样阈值控制寄存器 1	0x0048	R/W
APB_SARADC_THRES_CTRL_REG	阈值监控使能寄存器	0x0058	R/W
APB_SARADC_DMA_CONF_REG	SAR ADC DMA 配置寄存器	0x006C	R/W
APB_SARADC_APB_ADC_CLKM_CONF_REG	SAR ADC 时钟配置寄存器	0x0070	R/W
状态寄存器			

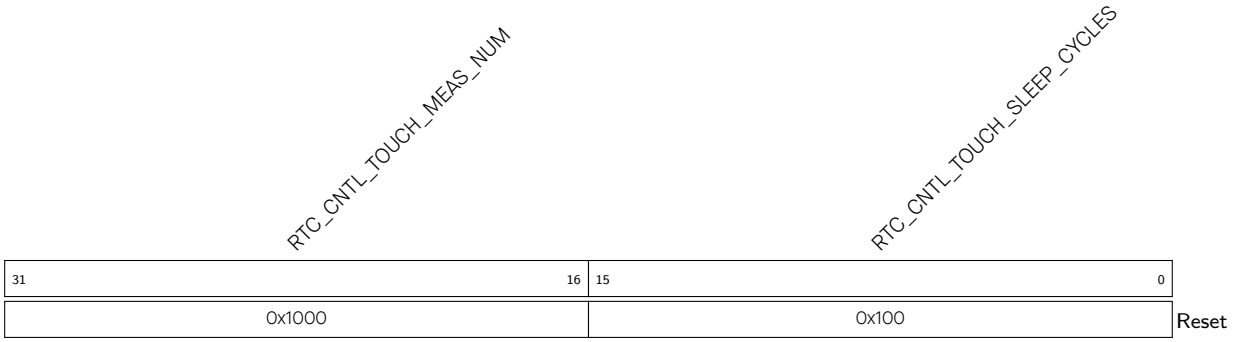
名称	描述	地址	访问
APB_SARADC_APB_SARADC1_DATA_STATUS_REG	SAR ADC1 采样数据	0x0040	RO
APB_SARADC_APB_SARADC2_DATA_STATUS_REG	SAR ADC2 采样数据	0x0078	RO
中断寄存器			
APB_SARADC_INT_ENA_REG	SAR ADC 中断使能寄存器	0x005C	R/W
APB_SARADC_INT_RAW_REG	SAR ADC 原始中断寄存器	0x0060	RO
APB_SARADC_INT_ST_REG	SAR ADC 中断状态寄存器	0x0064	RO
APB_SARADC_INT_CLR_REG	SAR ADC 中断清除寄存器	0x0068	WO
版本寄存器			
APB_SARADC_APB_CTRL_DATE_REG	版本控制寄存器	0x03FC	R/W

39.7 寄存器

39.7.1 SENSOR (ALWAYS_ON) 寄存器

本小节的所有地址均为相对于 [低功耗管理] 基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

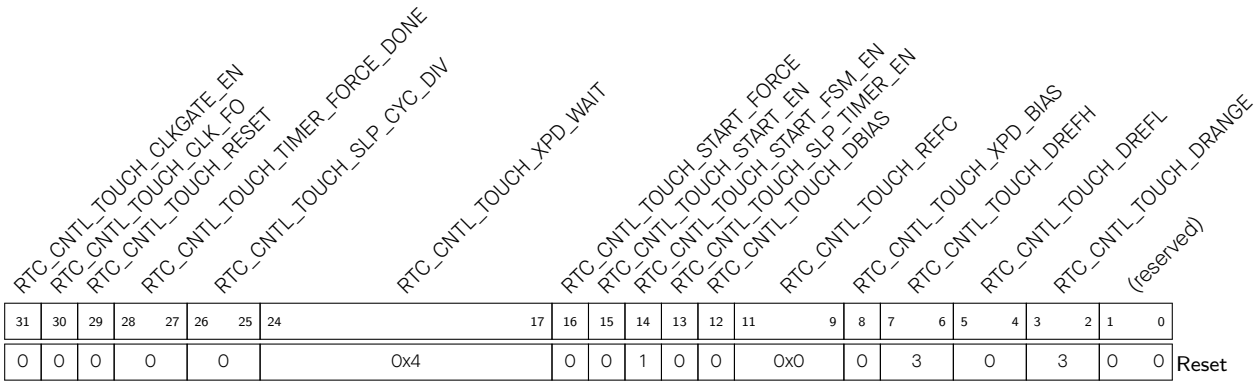
Register 39.1. RTC_CNTL_TOUCH_CTRL1_REG (0x0108)



RTC_CNTL_TOUCH_SLEEP_CYCLES 设置触摸定时器的睡眠周期。时钟：RTC_SLOW_CLK。(R/W)

RTC_CNTL_TOUCH_MEAS_NUM 设置测量阈值，即充放电的次数。(R/W)

Register 39.2. RTC_CNTL_TOUCH_CTRL2_REG (0x010C)



RTC_CNTL_TOUCH_DRANGE 触摸传感器充放电衰减。(R/W)

RTC_CNTL_TOUCH_DREFL 触摸传感器充放电低阈值。(R/W)

- 0: 0.5 V
- 1: 0.6 V
- 2: 0.7 V
- 3: 0.8 V

RTC_CNTL_TOUCH_DREFH 触摸传感器充放电高阈值。(R/W)

- 0: 2.4 V
- 1: 2.5 V
- 2: 2.6 V
- 3: 2.7 V

RTC_CNTL_TOUCH_XPD_BIAS 触摸传感器的偏置电源开关。(R/W)

RTC_CNTL_TOUCH_REFC 触摸管脚 0 的参考电容。(R/W)

RTC_CNTL_TOUCH_DBIAS 0：选择使用自身偏置。1：选择使用带隙偏置。(R/W)

RTC_CNTL_TOUCH_SLP_TIMER_EN 触摸传感器的定时器使能位。(R/W)

RTC_CNTL_TOUCH_START_FSM_EN 1：TOUCH_START 和 TOUCH_XPD 信号由 Touch FSM 控制；0：TOUCH_START 和 TOUCH_XPD 信号由软件控制。(R/W)

RTC_CNTL_TOUCH_START_EN 1：启动 Touch FSM，仅当 RTC_CNTL_TOUCH_START_FORCE = 1 时有效。(R/W)

RTC_CNTL_TOUCH_START_FORCE 1：Touch FSM 由软件启动；0：Touch FSM 由定时器启动。(R/W)

RTC_CNTL_TOUCH_XPD_WAIT 从 TOUCH_START 到 TOUCH_XPD 需等待的时钟周期。时钟：RTC_FAST_CLK。(R/W)

见下页

Register 39.2. RTC_CNTL_TOUCH_CTRL2_REG (0x010C)

接上页

RTC_CNTL_TOUCH_SLP_CYC_DIV 触摸管脚处于活动状态时，睡眠周期可以除以此处设置的数值。(R/W)

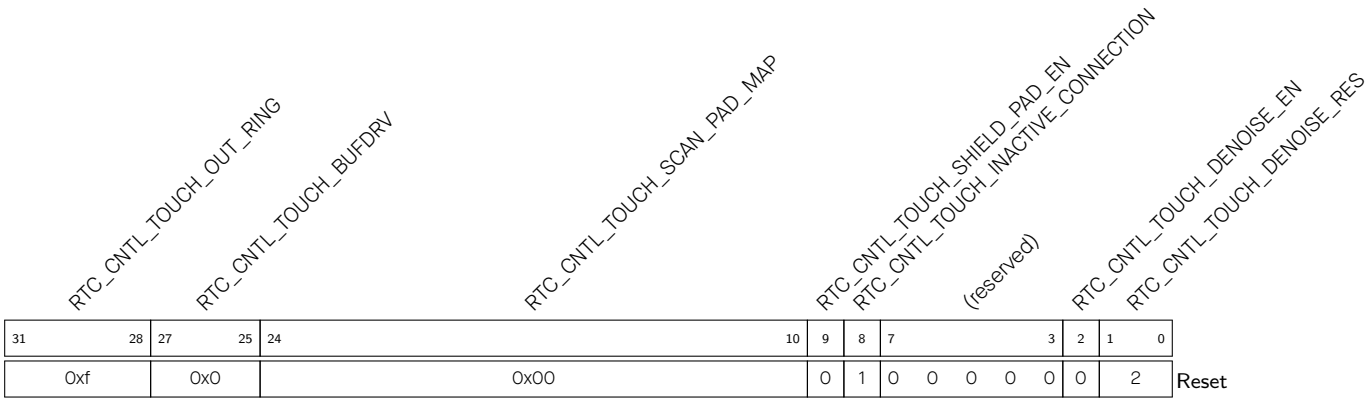
RTC_CNTL_TOUCH_TIMER_FORCE_DONE 强制结束 Touch 定时器。(R/W)

RTC_CNTL_TOUCH_RESET 软件复位 Touch FSM。(R/W)

RTC_CNTL_TOUCH_CLK_FO 强制开启触摸传感器的时钟。(R/W)

RTC_CNTL_TOUCH_CLKGATE_EN 触摸传感器时钟的使能位。(R/W)

Register 39.3. RTC_CNTL_TOUCH_SCAN_CTRL_REG (0x0110)



RTC_CNTL_TOUCH_DENOISE_RES 设置去噪分辨率。(R/W)

- 0: 12-bit
- 1: 10-bit
- 2: 8-bit
- 3: 4-bit

RTC_CNTL_TOUCH_DENOISE_EN 触摸管脚 0 将用于去噪。(R/W)

RTC_CNTL_TOUCH_INACTIVE_CONNECTION 配置非活动状态的触摸管脚的连接方式。0：连接至 HighZ。1：连接至 GND。(R/W)

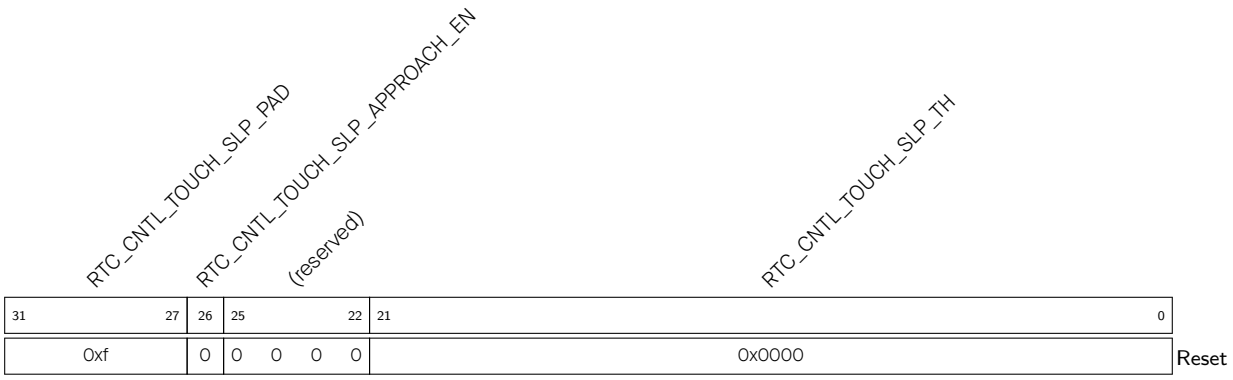
RTC_CNTL_TOUCH_SHIELD_PAD_EN 配置触摸管脚 14 用作屏蔽垫。(R/W)

RTC_CNTL_TOUCH_SCAN_PAD_MAP 触摸扫描模式下触摸管脚的使能位图。(R/W)

RTC_CNTL_TOUCH_BUFDRV 设置触摸管脚 14 的驱动强度。(R/W)

RTC_CNTL_TOUCH_OUT_RING 选择一个触摸管脚用作保护环。(R/W)

Register 39.4. RTC_CNTL_TOUCH_SLP_THRES_REG (0x0114)

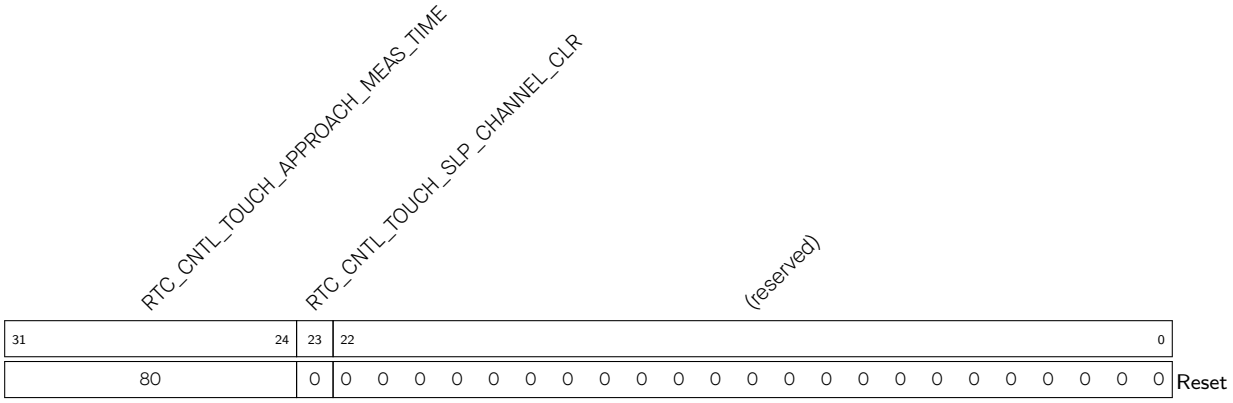


RTC_CNTL_TOUCH_SLP_TH 设置睡眠管脚的阈值。(R/W)

RTC_CNTL_TOUCH_SLP_APPROACH_EN 使能睡眠管脚的接近模式。(R/W)

RTC_CNTL_TOUCH_SLP_PAD 选择一个触摸管脚用作睡眠管脚。(R/W)

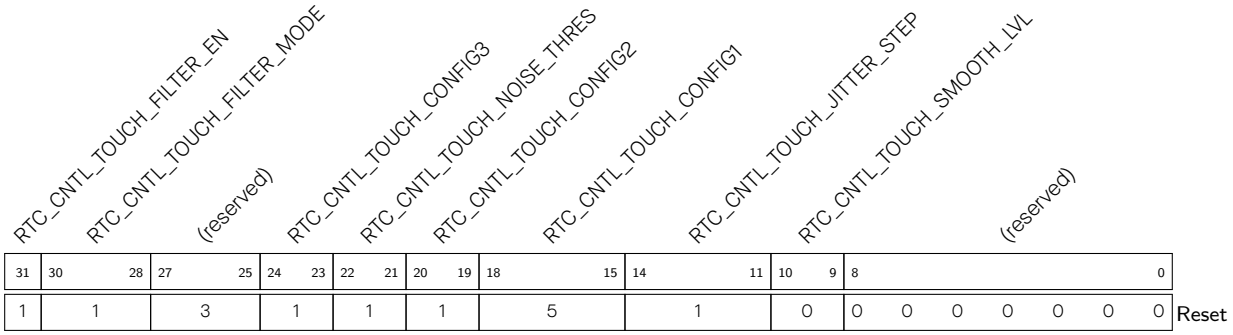
Register 39.5. RTC_CNTL_TOUCH_APPROACH_REG (0x0118)



RTC_CNTL_TOUCH_SLP_CHANNEL_CLR 清除睡眠通道。(WO)

RTC_CNTL_TOUCH_APPROACH_MEAS_TIME 设置成接近模式的管脚执行的所有测量次数。范围：0 ~ 255。(R/W)

Register 39.6. RTC_CNTL_TOUCH_FILTER_CTRL_REG (0x011C)



RTC_CNTL_TOUCH_SMOOTH_LVL 配置平滑过滤系数。0：原始数据；1：IIR1/2；2：IIR1/4；3：IIR1/8。(R/W)

RTC_CNTL_TOUCH_JITTER_STEP 配置抖动幅度。范围：0 ~ 15。(R/W)

RTC_CNTL_TOUCH_CONFIG1 内部配置。(R/W)

RTC_CNTL_TOUCH_CONFIG2 内部配置。(R/W)

RTC_CNTL_TOUCH_NOISE_THRES 正噪声阈值。(R/W)

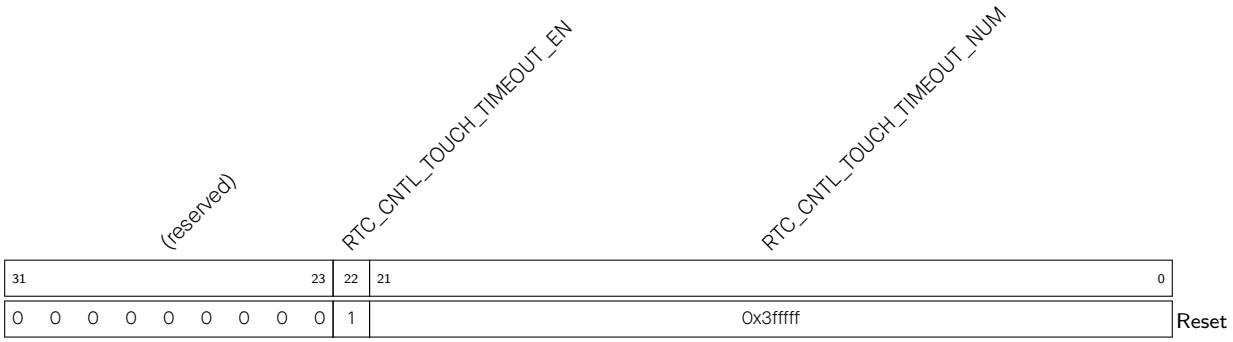
RTC_CNTL_TOUCH_CONFIG3 触摸迟滞。(R/W)

RTC_CNTL_TOUCH_FILTER_MODE 滤波模式选择。(R/W)

- 0: IIR 1/2
- 1: IIR 1/4
- 2: IIR 1/8
- 3: IIR 1/16
- 4: IIR 1/32
- 5: IIR 1/64
- 6: IIR 1/128
- 7: Jitter

RTC_CNTL_TOUCH_FILTER_EN 使能噪声过滤功能。(R/W)

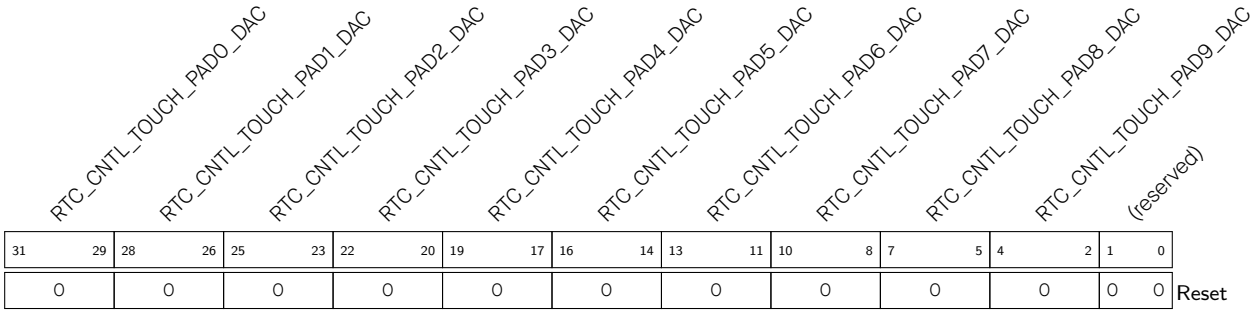
Register 39.7. RTC_CNTL_TOUCH_TIMEOUT_CTRL_REG (0x0124)



RTC_CNTL_TOUCH_TIMEOUT_NUM 设置触摸超时阈值。(R/W)

RTC_CNTL_TOUCH_TIMEOUT_EN 使能触摸超时。(R/W)

Register 39.8. RTC_CNTL_TOUCH_DAC_REG (0x014C)



RTC_CNTL_TOUCH_PAD9_DAC 配置触摸传感器 9 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD8_DAC 配置触摸传感器 8 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD7_DAC 配置触摸传感器 7 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD6_DAC 配置触摸传感器 6 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD5_DAC 配置触摸传感器 5 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD4_DAC 配置触摸传感器 4 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD3_DAC 配置触摸传感器 3 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD2_DAC 配置触摸传感器 2 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD1_DAC 配置触摸传感器 1 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD0_DAC 配置触摸传感器 0 的斜率。(R/W)

Register 39.9. RTC_CNTL_TOUCH_DAC1_REG (0x0150)

RTC_CNTL_TOUCH_PAD10_DAC	RTC_CNTL_TOUCH_PAD11_DAC	RTC_CNTL_TOUCH_PAD12_DAC	RTC_CNTL_TOUCH_PAD13_DAC	RTC_CNTL_TOUCH_PAD14_DAC	(reserved)	
31 29	28 26	25 23	22 20	19 17	16 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_TOUCH_PAD14_DAC 配置触摸传感器 14 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD13_DAC 配置触摸传感器 13 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD12_DAC 配置触摸传感器 12 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD11_DAC 配置触摸传感器 11 的斜率。(R/W)

RTC_CNTL_TOUCH_PAD10_DAC 配置触摸传感器 10 的斜率。(R/W)

39.7.2 SENSOR (RTC_PERI) 寄存器

本小节的所有地址均为相对于 [低功耗管理基地址 + 0x800] 的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 39.10. SENS_SAR_READER1_CTRL_REG (0x0000)

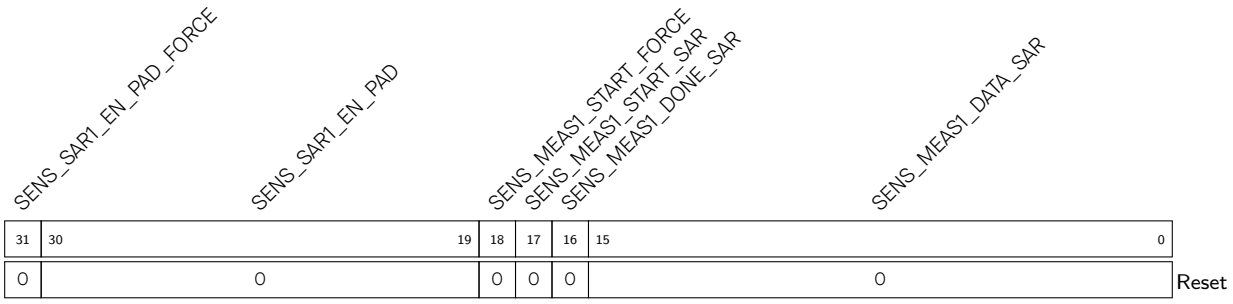
(reserved)	SENS_SAR1_INT_EN	SENS_SAR1_DATA_INV	(reserved)	SENS_SAR1_CLK_DIV	
31 30	29	28	27 8	7 0	
0 0	1	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	2	Reset

SENS_SAR1_CLK_DIV 配置时钟分频器。(R/W)

SENS_SAR1_DATA_INV 反转 SAR ADC1 数据。(R/W)

SENS_SAR1_INT_EN 配置 SAR ADC1 中断。0：禁用中断。1：使能中断。(R/W)

Register 39.11. SENS_SAR_MEAS1_CTRL2_REG (0x000C)



SENS_MEAS1_DATA_SAR SAR ADC1 数据。(RO)

SENS_MEAS1_DONE_SAR 用于标记 SAR ADC1 转换已完成。(RO)

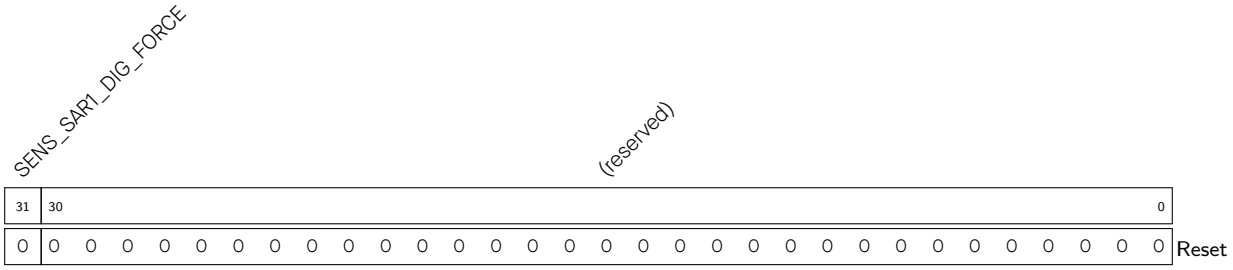
SENS_MEAS1_START_SAR RTC ADC1 控制器开始进行转换数据，仅在 SENS_MEAS1_START_FORCE = 1 时有效。(R/W)

SENS_MEAS1_START_FORCE 1：RTC ADC1 控制器由软件启动；0：RTC ADC1 控制器由 ULP 协处理器启动。(R/W)

SENS_SAR1_EN_PAD SAR ADC1 的管脚使能位图，仅在 SENS_SAR1_EN_PAD_FORCE = 1 时有效。(R/W)

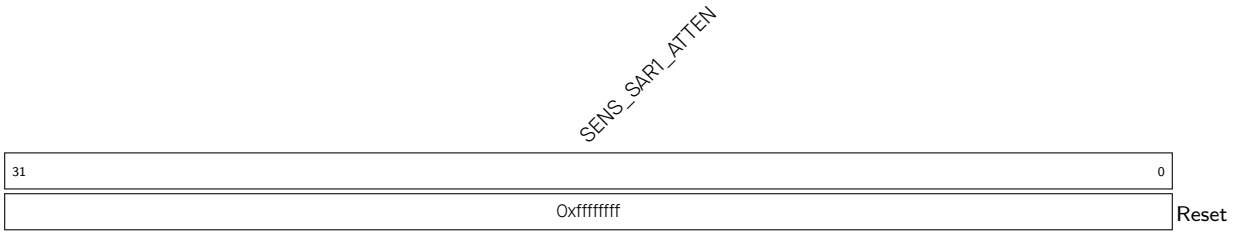
SENS_SAR1_EN_PAD_FORCE 1：SAR ADC1 管脚使能位图由软件控制；0：SAR ADC1 管脚使能位图由 ULP 协处理器控制。(R/W)

Register 39.12. SENS_SAR_MEAS1_MUX_REG (0x0010)



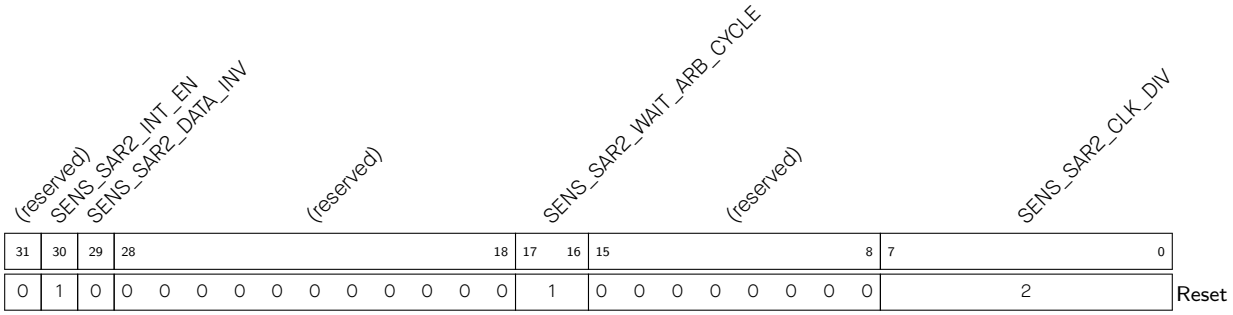
SENS_SAR1_DIG_FORCE 写入 1，则 SAR ADC1 将由 DIG ADC1 控制器控制。(R/W)

Register 39.13. SENS_SAR_ATTEN1_REG (0x0014)



SENS_SAR1_ATTEN SAR ADC1 每个管脚的 2 位衰减。[1:0] 用于通道 0，[3:2] 用于通道 1，以此类推。(R/W)

Register 39.14. SENS_SAR_READER2_CTRL_REG (0x0024)



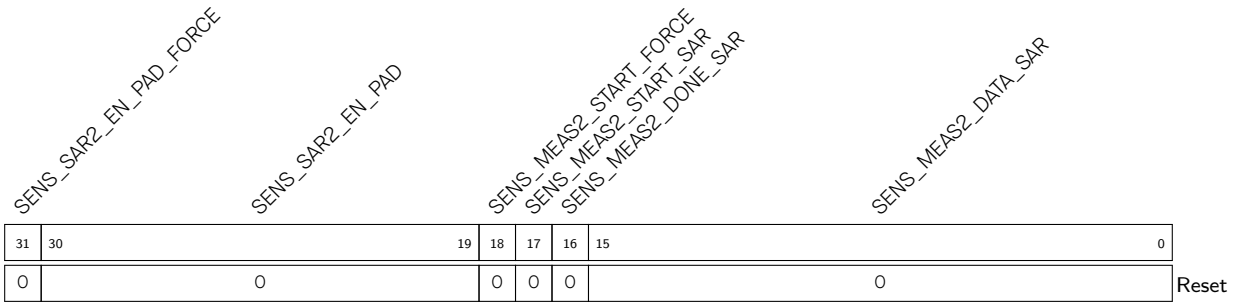
SENS_SAR2_CLK_DIV 配置时钟分配系数。(R/W)

SENS_SAR2_WAIT_ARB_CYCLE SAR ADC2 转换完成后，等待仲裁器稳定需要的周期数。(R/W)

SENS_SAR2_DATA_INV 反转 SAR ADC2 数据。(R/W)

SENS_SAR2_INT_EN 使能 SAR ADC2 中断。(R/W)

Register 39.15. SENS_SAR_MEAS2_CTRL2_REG (0x0030)



SENS_MEAS2_DATA_SAR SAR ADC2 数据。(RO)

SENS_MEAS2_DONE_SAR 标记 SAR ADC2 数据转换完成。(RO)

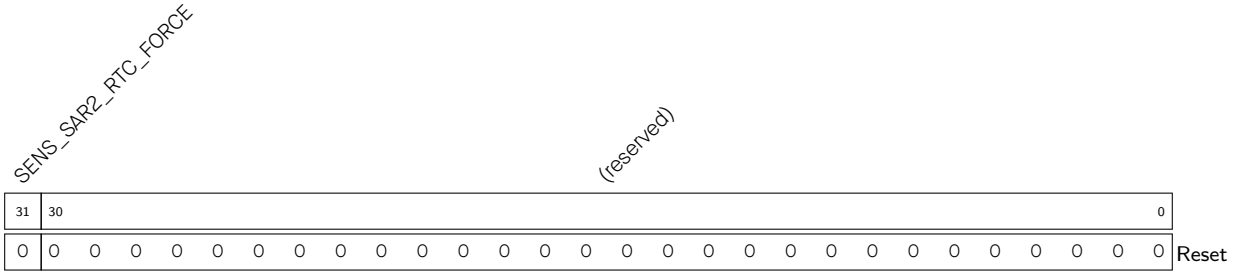
SENS_MEAS2_START_SAR RTC ADC2 控制器开始数据转换。仅当 SENS_MEAS2_START_FORCE = 1 时有效。(R/W)

SENS_MEAS2_START_FORCE 1：RTC ADC2 控制器由软件启动；0：RTC ADC2 控制器由 ULP 协处理器启动。(R/W)

SENS_SAR2_EN_PAD SAR ADC2 管脚使能位图。仅当 SENS_SAR2_EN_PAD_FORCE = 1 时有效。(R/W)

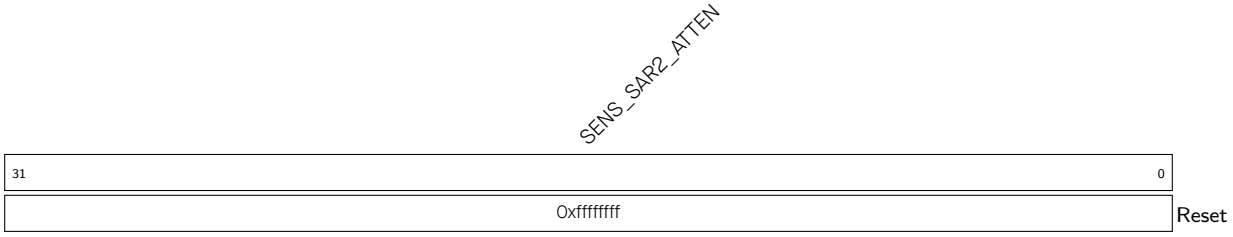
SENS_SAR2_EN_PAD_FORCE 1：SAR ADC2 管脚使能位图由软件控制；0：SAR ADC2 管脚使能位图由 ULP 协处理器控制。(R/W)

Register 39.16. SENS_SAR_MEAS2_MUX_REG (0x0034)



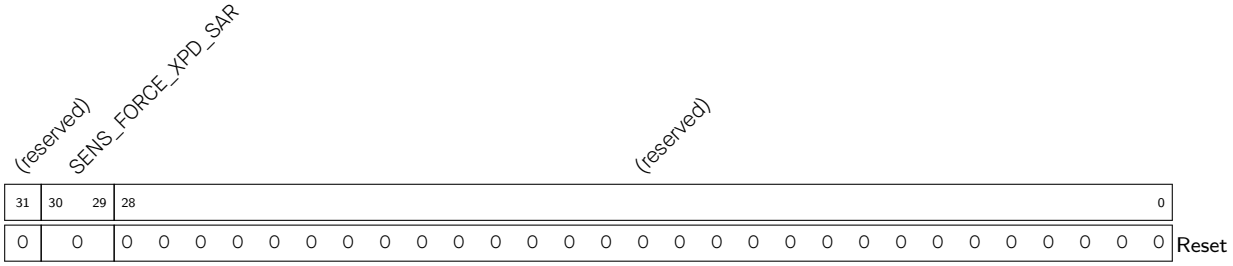
SENS_SAR2_RTC_FORCE 睡眠状态下，强制使用 RTC ADC2 控制器。(R/W)

Register 39.17. SENS_SAR_ATTEN2_REG (0x0038)



SENS_SAR2_ATTEN SAR ADC2 每个管脚的 2 位衰减。[1:0] 用于通道 0，[3:2] 用于通道 1，以此类推。(R/W)

Register 39.18. SENS_SAR_POWER_XPD_SAR_REG (0x003C)



SENS_FORCE_XPD_SAR 配置是否使能强制上电/断电 SAR ADC。

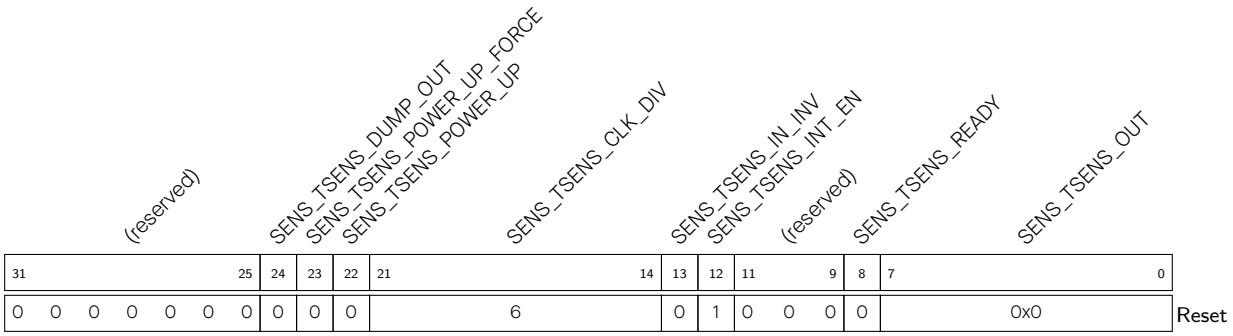
0/1：禁用强制上电/断电功能

2：强制上电

3：强制断电

(R/W)

Register 39.19. SENS_SAR_TSENS_CTRL_REG (0x0050)



SENS_TSENS_OUT 温度传感器的输出数值。(RO)

SENS_TSENS_READY 提示温度传感器已准备好输出数据。(RO)

SENS_TSENS_INT_EN 使能温度传感器中断。(R/W)

SENS_TSENS_IN_INV 反转温度传感器的输入数据。(R/W)

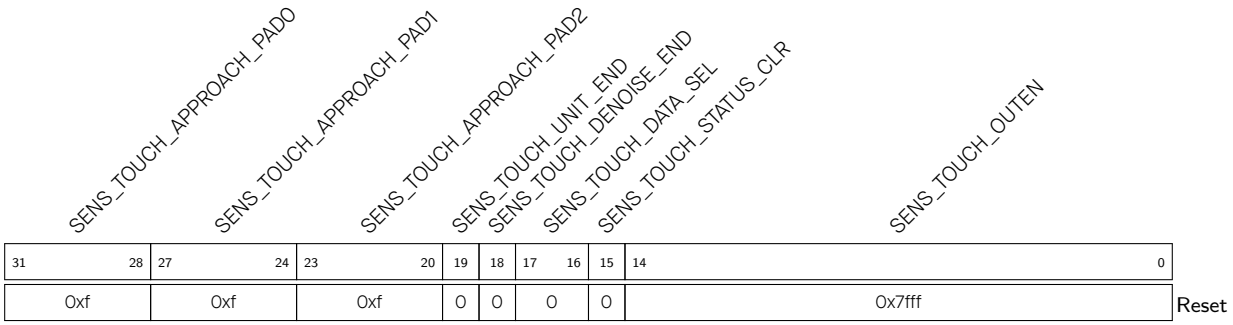
SENS_TSENS_CLK_DIV 温度传感器的时钟分频系数。(R/W)

SENS_TSENS_POWER_UP 温度传感器上电。(R/W)

SENS_TSENS_POWER_UP_FORCE 0：温度传感器的数据转储和上电由 FSM 控制。1：温度传感器的数据转储和上电由软件控制。(R/W)

SENS_TSENS_DUMP_OUT 温度传感器数据转储，仅在 SNES_TSENS_POWER_UP_FORCE = 1 时有效。(R/W)

Register 39.20. SENS_SAR_TOUCH_CONF_REG (0x005C)



SENS_TOUCH_OUTEN 使能触摸传感器输出。(R/W)

SENS_TOUCH_STATUS_CLR 清除所有触摸传感器的触摸状态。(WO)

SENS_TOUCH_DATA_SEL 选择触摸数据类型。(R/W)

- 0 或 1：原始数据
- 2：基准数据
- 3：平滑数据

SENS_TOUCH_DENOISE_END 标记除噪完成。(RO)

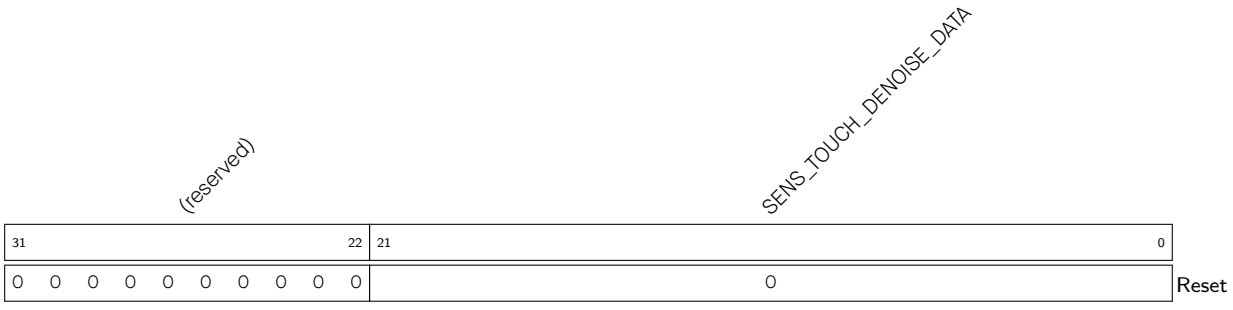
SENS_TOUCH_UNIT_END 标记触摸传感器采样完成。(RO)

SENS_TOUCH_APPROACH_PAD2 选择一个管脚配置成接近模式。该管脚将用作接近模式管脚 2。(R/W)

SENS_TOUCH_APPROACH_PAD1 选择一个管脚配置成接近模式。该管脚将用作接近模式管脚 1。(R/W)

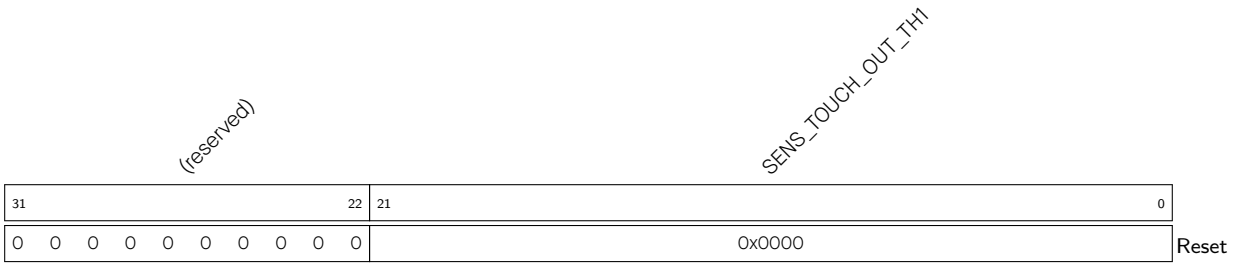
SENS_TOUCH_APPROACH_PAD0 选择一个管脚配置成接近模式。该管脚将用作接近模式管脚 0。(R/W)

Register 39.21. SENS_SAR_TOUCH_DENOISE_REG (0x0060)



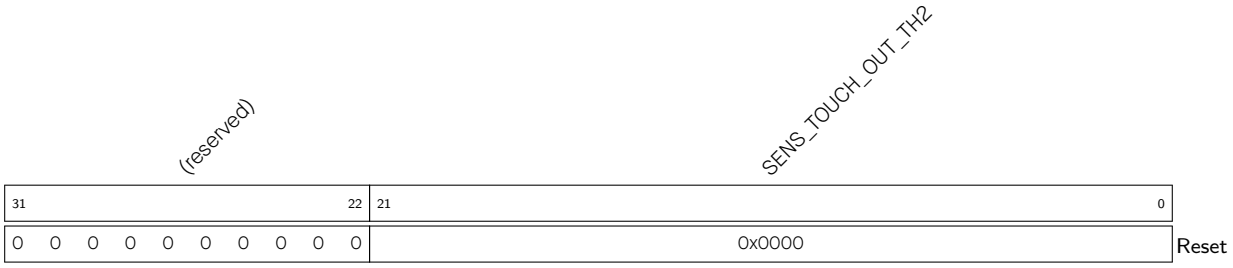
SENS_TOUCH_DENOISE_DATA 触摸传感器 0 测得的去噪数据。(RO)

Register 39.22. SENS_SAR_TOUCH_THRES1_REG (0x0064)



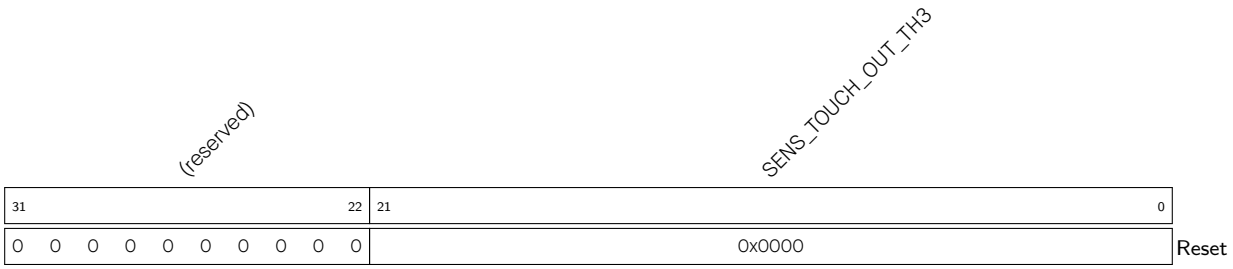
SENS_TOUCH_OUT_TH1 管脚 1 的触摸阈值。(R/W)

Register 39.23. SENS_SAR_TOUCH_THRES2_REG (0x0068)



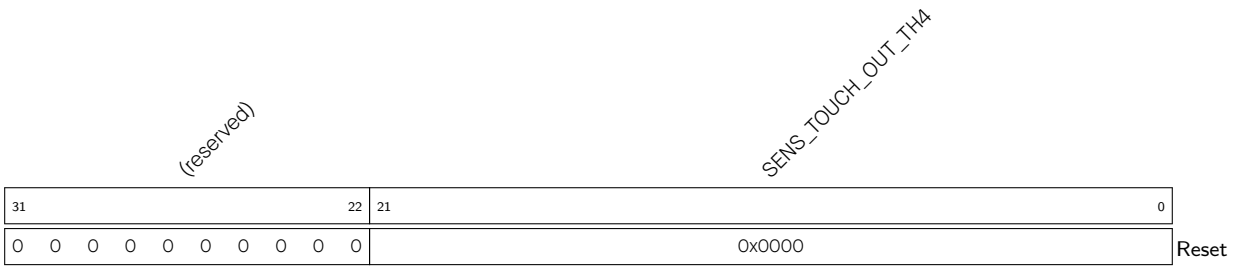
SENS_TOUCH_OUT_TH2 管脚 2 的触摸阈值。(R/W)

Register 39.24. SENS_SAR_TOUCH_THRES3_REG (0x006C)



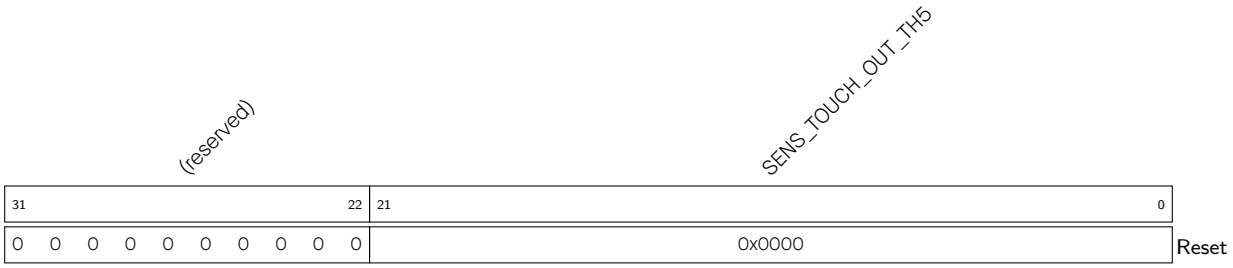
SENS_TOUCH_OUT_TH3 管脚 3 的触摸阈值。(R/W)

Register 39.25. SENS_SAR_TOUCH_THRES4_REG (0x0070)



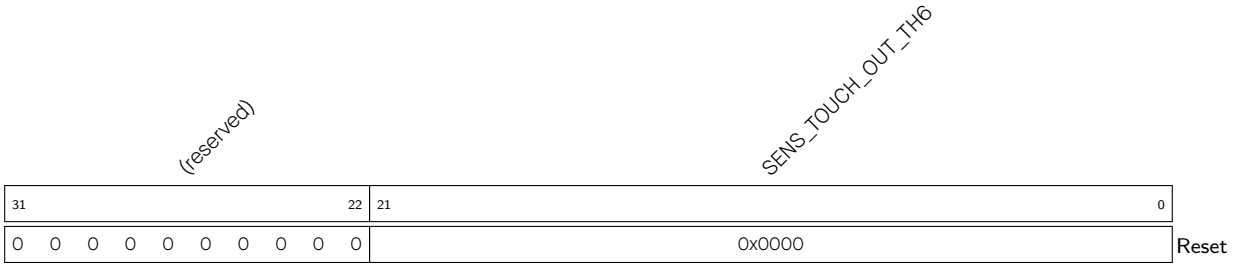
SENS_TOUCH_OUT_TH4 管脚 4 的触摸阈值。(R/W)

Register 39.26. SENS_SAR_TOUCH_THRES5_REG (0x0074)



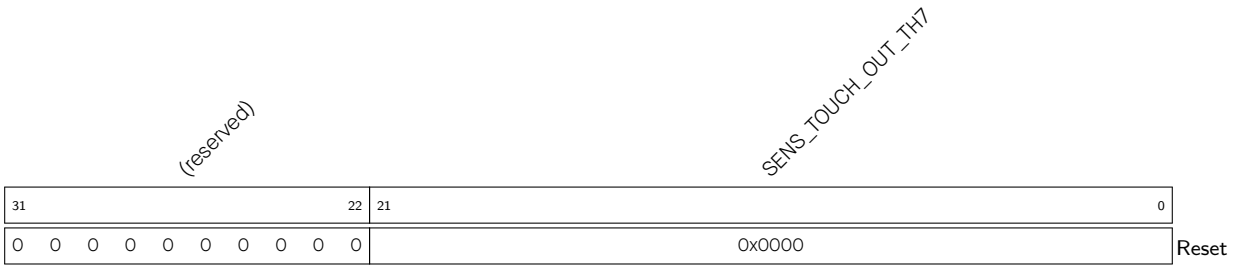
SENS_TOUCH_OUT_TH5 管脚 5 的触摸阈值。(R/W)

Register 39.27. SENS_SAR_TOUCH_THRES6_REG (0x0078)



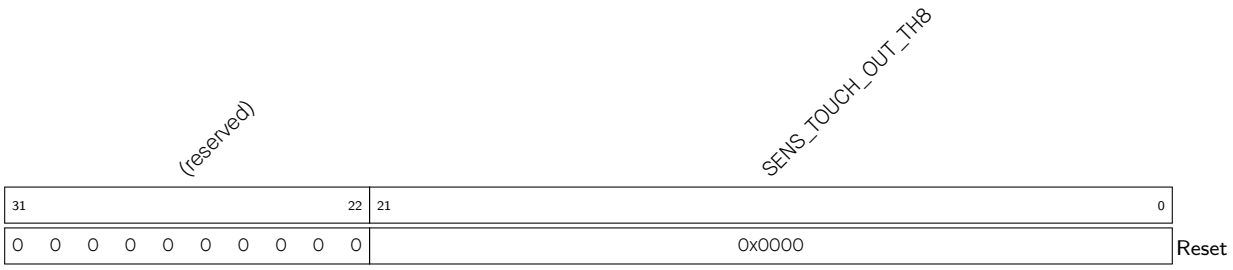
SENS_TOUCH_OUT_TH6 管脚 6 的触摸阈值。(R/W)

Register 39.28. SENS_SAR_TOUCH_THRES7_REG (0x007C)



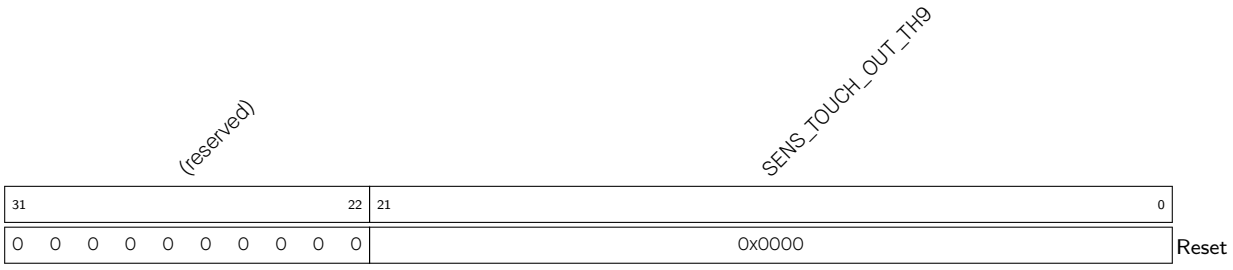
SENS_TOUCH_OUT_TH7 管脚 7 的触摸阈值。(R/W)

Register 39.29. SENS_SAR_TOUCH_THRES8_REG (0x0080)



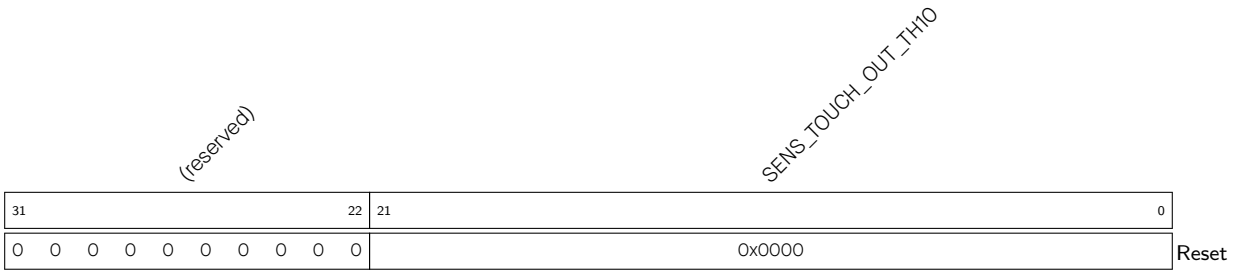
SENS_TOUCH_OUT_TH8 管脚 8 的触摸阈值。(R/W)

Register 39.30. SENS_SAR_TOUCH_THRES9_REG (0x0084)



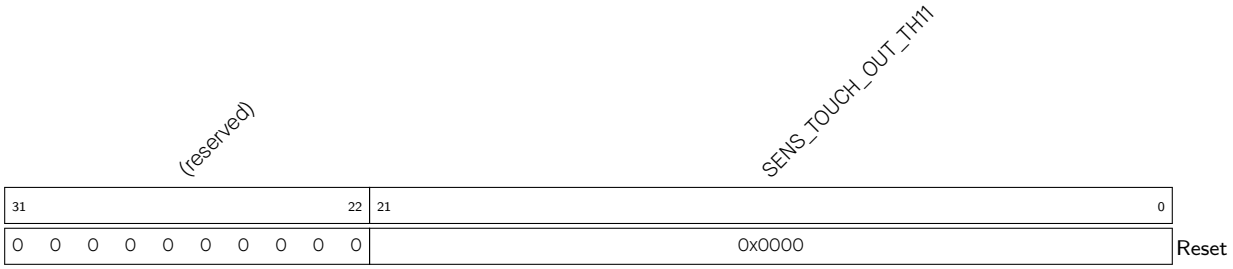
SENS_TOUCH_OUT_TH9 管脚 9 的触摸阈值。(R/W)

Register 39.31. SENS_SAR_TOUCH_THRES10_REG (0x0088)



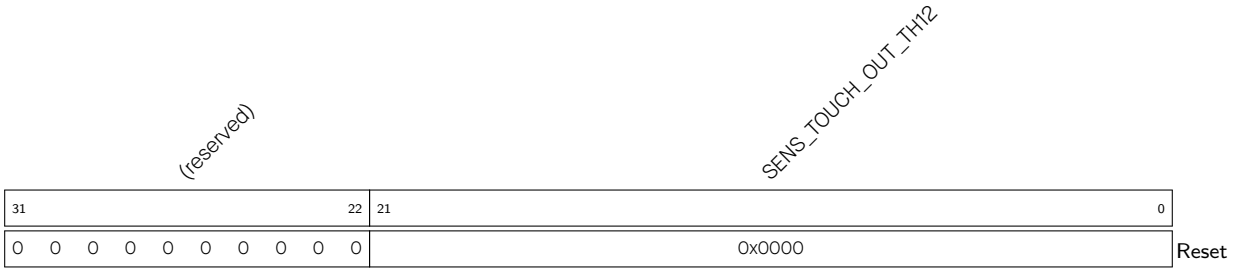
SENS_TOUCH_OUT_TH10 管脚 10 的触摸阈值。(R/W)

Register 39.32. SENS_SAR_TOUCH_THRES11_REG (0x008C)



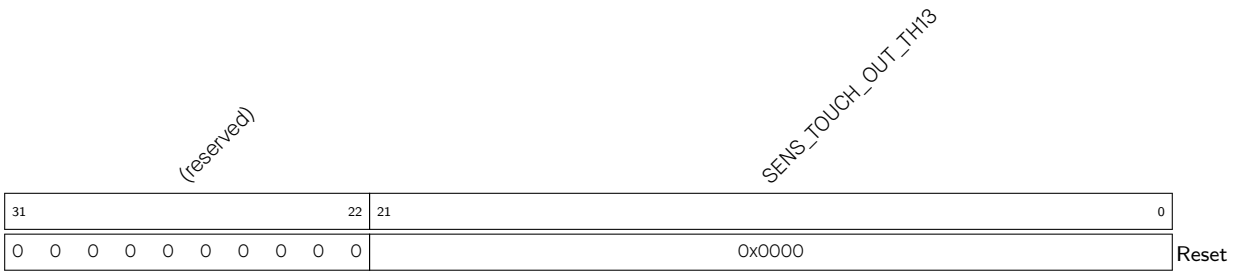
SENS_TOUCH_OUT_TH11 管脚 11 的触摸阈值。(R/W)

Register 39.33. SENS_SAR_TOUCH_THRES12_REG (0x0090)



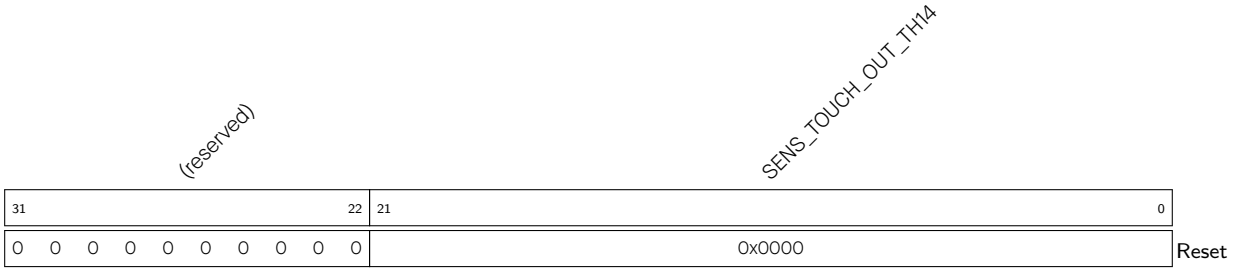
SENS_TOUCH_OUT_TH12 管脚 12 的触摸阈值。(R/W)

Register 39.34. SENS_SAR_TOUCH_THRES13_REG (0x0094)



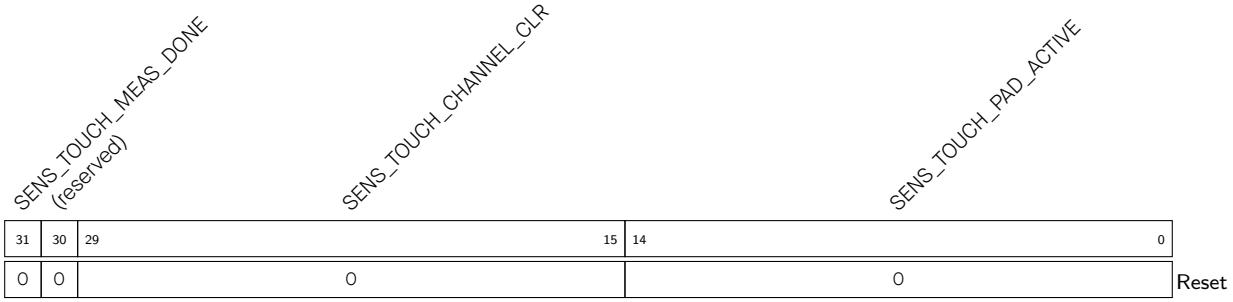
SENS_TOUCH_OUT_TH13 管脚 13 的触摸阈值。(R/W)

Register 39.35. SENS_SAR_TOUCH_THRES14_REG (0x0098)



SENS_TOUCH_OUT_TH14 管脚 14 的触摸阈值。(R/W)

Register 39.36. SENS_SAR_TOUCH_CHN_ST_REG (0x009C)

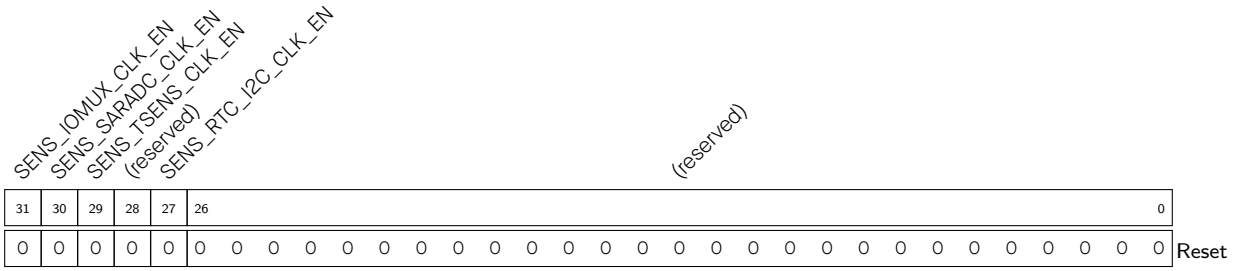


SENS_TOUCH_PAD_ACTIVE 触摸传感器的有效触摸状态。(RO)

SENS_TOUCH_CHANNEL_CLR 清除触摸通道。(WO)

SENS_TOUCH_MEAS_DONE 标记触摸管脚已完成测量操作。(RO)

Register 39.37. SENS_SAR_PERI_CLK_GATE_CONF_REG (0x0104)



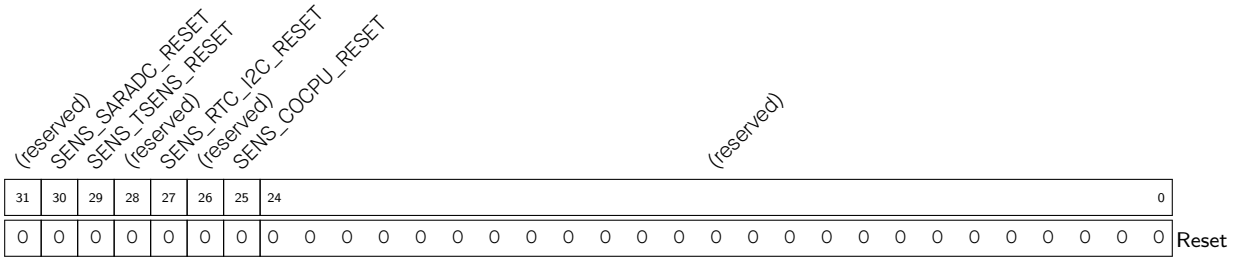
SENS_RTC_I2C_CLK_EN 使能 RTC I2C 时钟。(R/W)

SENS_TSENS_CLK_EN 使能温度传感器时钟。(R/W)

SENS_SARADC_CLK_EN 使能 SAR ADC 时钟。(R/W)

SENS_IOMUX_CLK_EN 使能 IO MUX 时钟。(R/W)

Register 39.38. SENS_SAR_PERI_RESET_CONF_REG (0x0108)



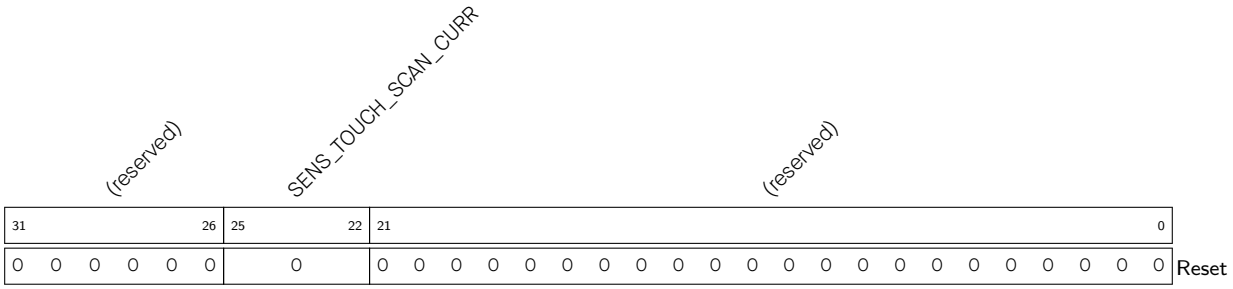
SENS_COCPU_RESET ULP-RISC-V 复位使能。(R/W)

SENS_RTC_I2C_RESET RTC_I2C 复位使能。(R/W)

SENS_TSENS_RESET SAR ADC 复位使能。(R/W)

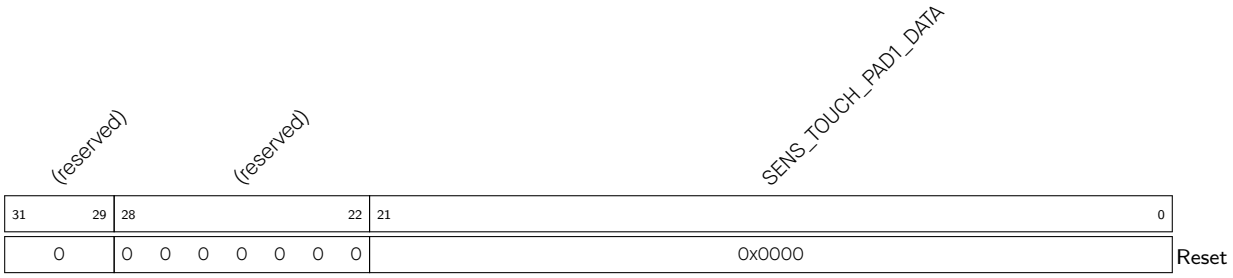
SENS_SARADC_RESET IO MUX 复位使能。(R/W)

Register 39.39. SENS_SAR_TOUCH_STATUS0_REG (0x00A0)



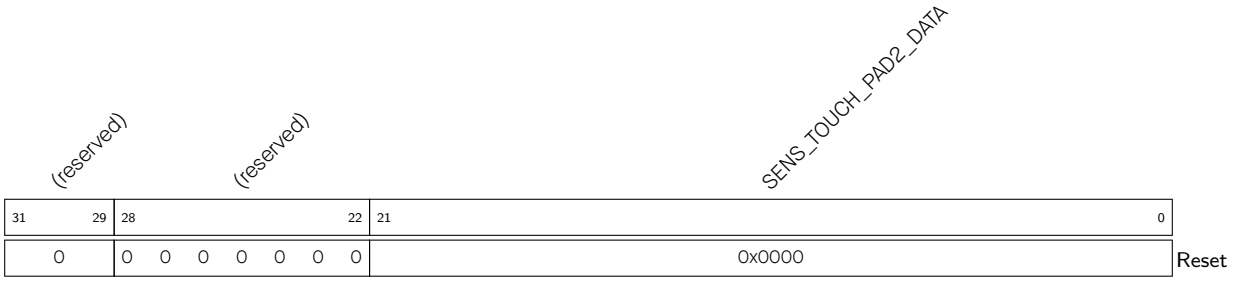
SENS_TOUCH_SCAN_CURR 表示正处于扫描状态的管脚。(RO)

Register 39.40. SENS_SAR_TOUCH_STATUS1_REG (0x00A4)



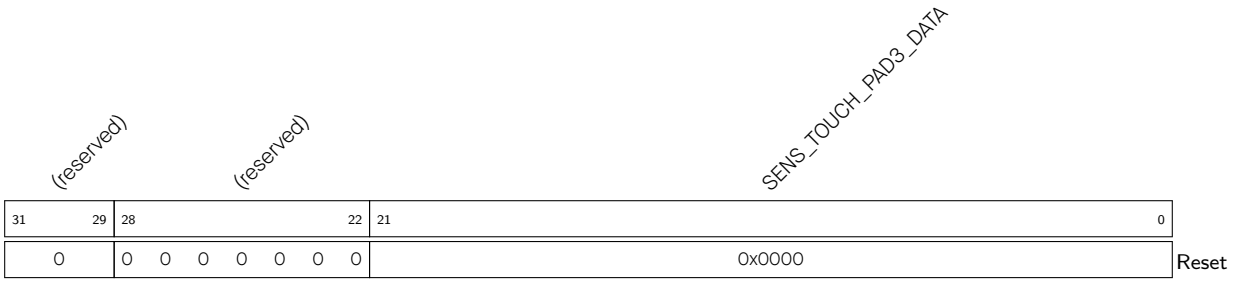
SENS_TOUCH_PAD1_DATA 触摸管脚 1 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.41. SENS_SAR_TOUCH_STATUS2_REG (0x00A8)



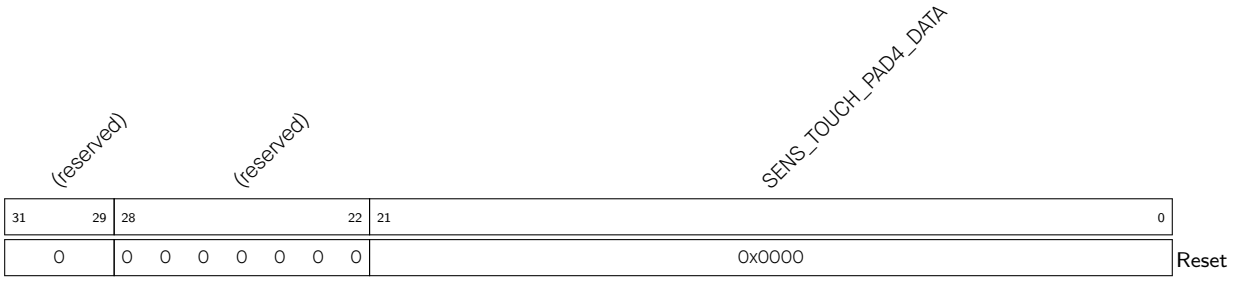
SENS_TOUCH_PAD2_DATA 触摸管脚 2 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.42. SENS_SAR_TOUCH_STATUS3_REG (0x00AC)



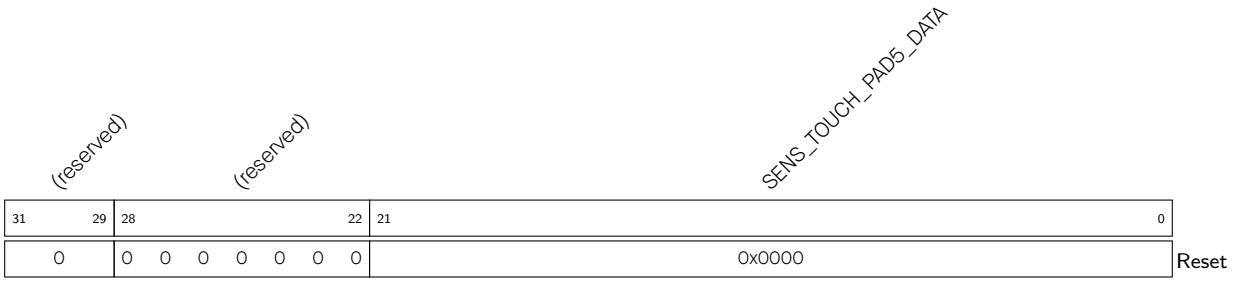
SENS_TOUCH_PAD3_DATA 触摸管脚 3 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.43. SENS_SAR_TOUCH_STATUS4_REG (0x00B0)



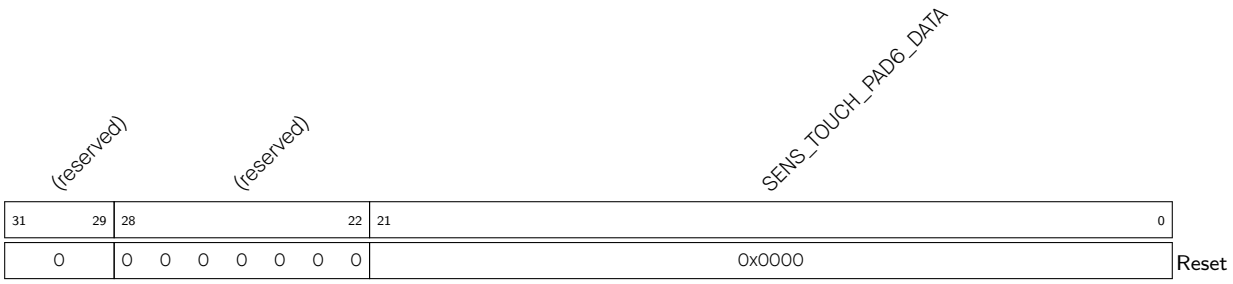
SENS_TOUCH_PAD4_DATA 触摸管脚 4 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.44. SENS_SAR_TOUCH_STATUS5_REG (0x00B4)



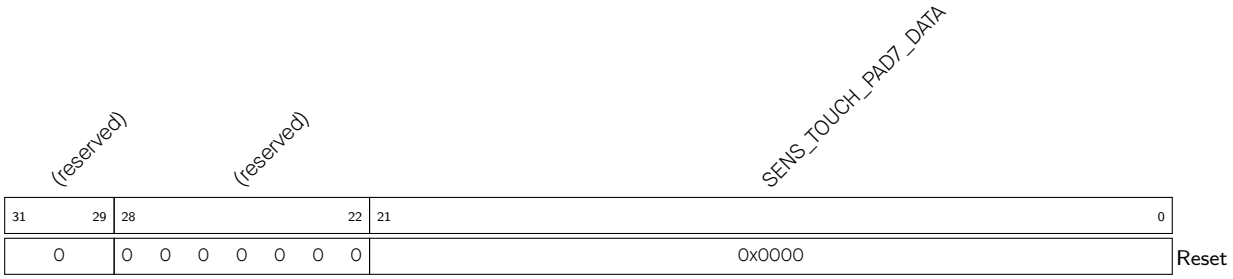
SENS_TOUCH_PAD5_DATA 触摸管脚 5 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.45. SENS_SAR_TOUCH_STATUS6_REG (0x00B8)



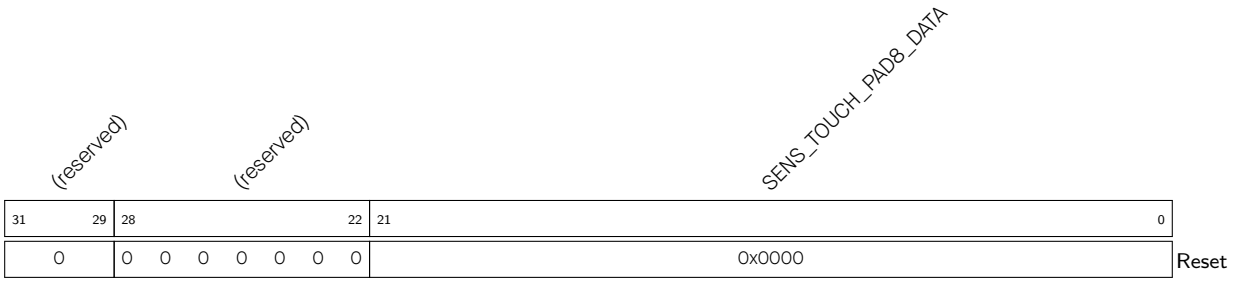
SENS_TOUCH_PAD6_DATA 触摸管脚 6 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.46. SENS_SAR_TOUCH_STATUS7_REG (0x00BC)



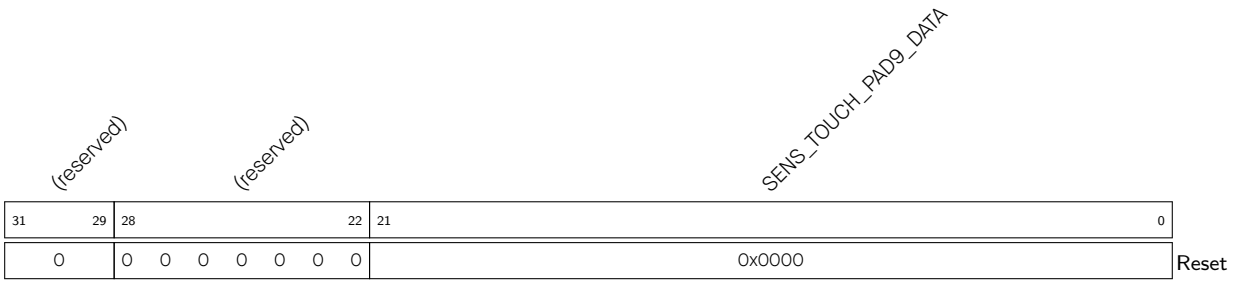
SENS_TOUCH_PAD7_DATA 触摸管脚 7 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.47. SENS_SAR_TOUCH_STATUS8_REG (0x00C0)



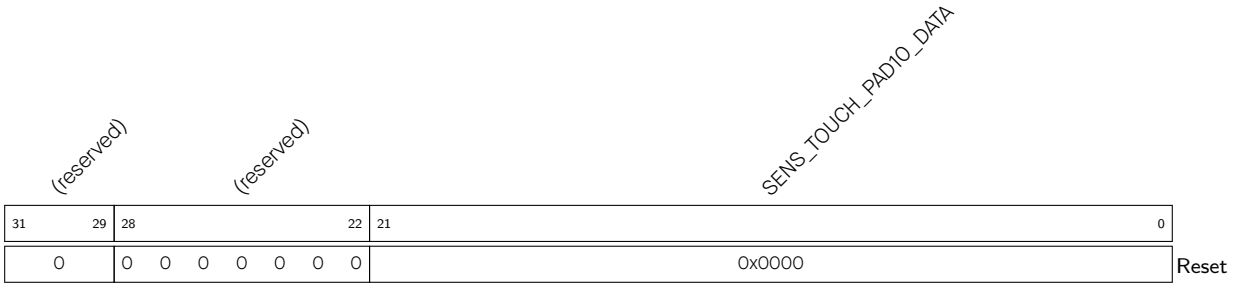
SENS_TOUCH_PAD8_DATA 触摸管脚 8 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.48. SENS_SAR_TOUCH_STATUS9_REG (0x00C4)



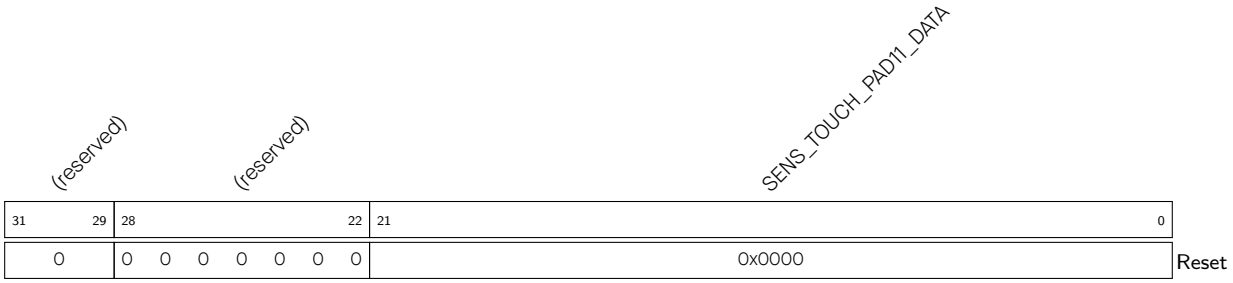
SENS_TOUCH_PAD9_DATA 触摸管脚 9 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.49. SENS_SAR_TOUCH_STATUS10_REG (0x00C8)



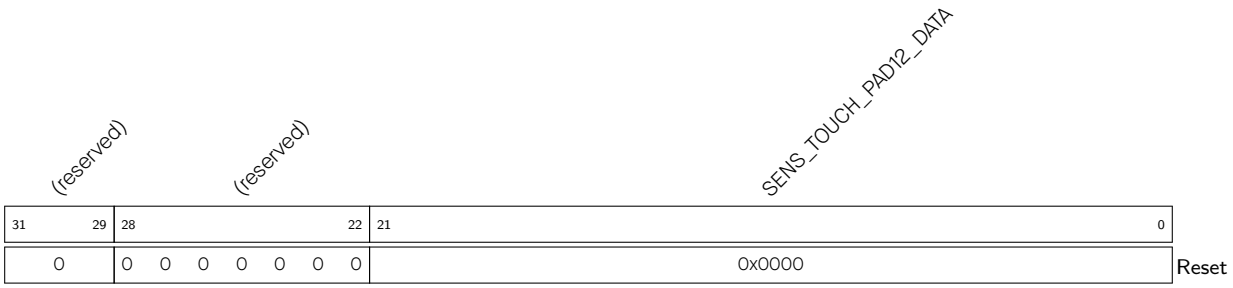
SENS_TOUCH_PAD10_DATA 触摸管脚 10 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.50. SENS_SAR_TOUCH_STATUS11_REG (0x00CC)



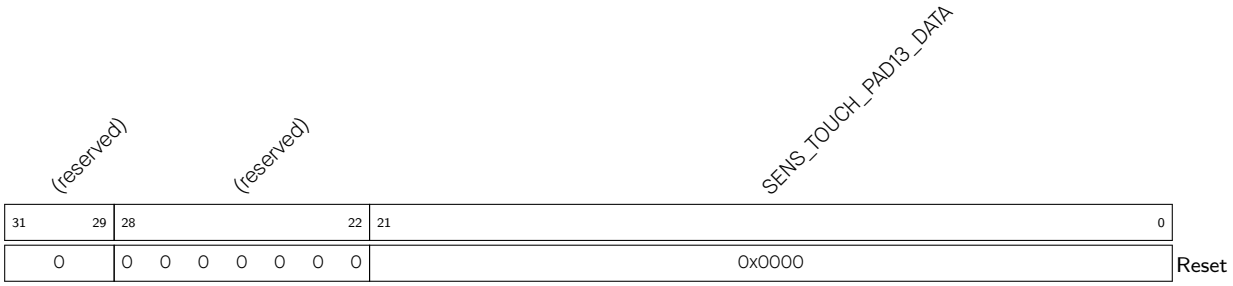
SENS_TOUCH_PAD11_DATA 触摸管脚 11 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.51. SENS_SAR_TOUCH_STATUS12_REG (0x00D0)



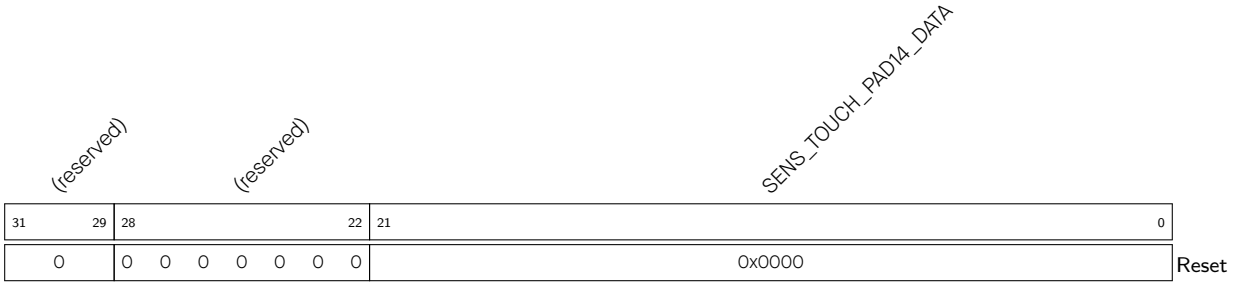
SENS_TOUCH_PAD12_DATA 触摸管脚 12 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.52. SENS_SAR_TOUCH_STATUS13_REG (0x00D4)



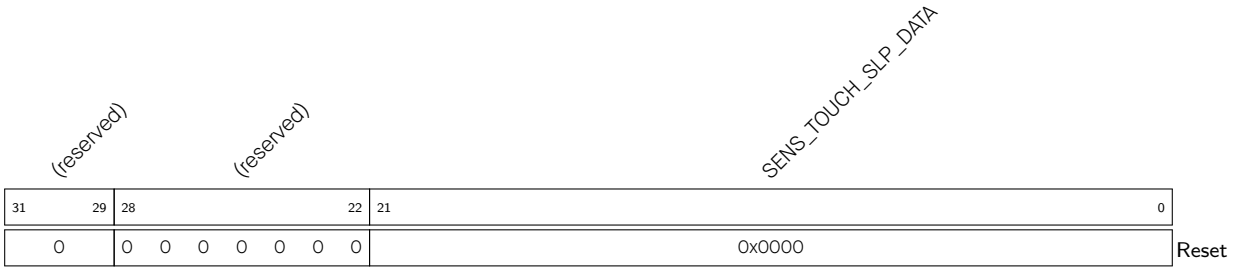
SENS_TOUCH_PAD13_DATA 触摸管脚 13 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.53. SENS_SAR_TOUCH_STATUS14_REG (0x00D8)



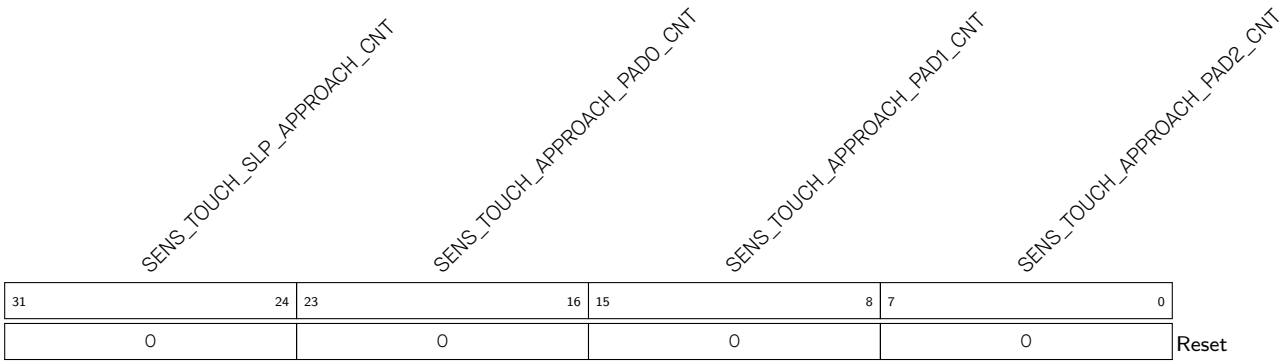
SENS_TOUCH_PAD14_DATA 触摸管脚 14 的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.54. SENS_SAR_TOUCH_STATUS15_REG (0x00DC)



SENS_TOUCH_SLP_DATA 睡眠管脚的数据，具体类型由 SENS_TOUCH_DATA_SEL 决定。(RO)

Register 39.55. SENS_SAR_TOUCH_APPR_STATUS_REG (0x00E0)



SENS_TOUCH_APPROACH_PAD2_CNT 接近模式管脚 2 的触摸计数值。(RO)

SENS_TOUCH_APPROACH_PAD1_CNT 接近模式管脚 1 的触摸计数值。(RO)

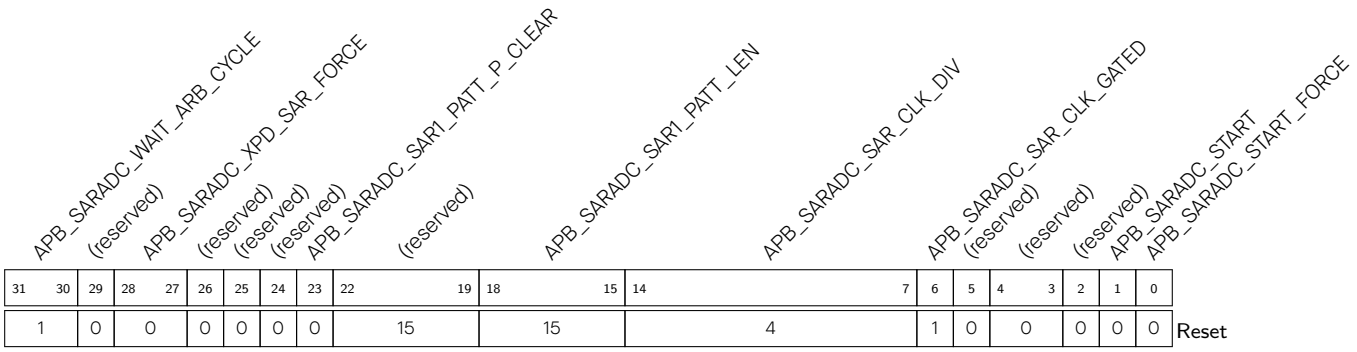
SENS_TOUCH_APPROACH_PAD0_CNT 接近模式管脚 0 的触摸计数值。(RO)

SENS_TOUCH_SLP_APPROACH_CNT 接近模式睡眠管脚的触摸计数值。(RO)

39.7.3 SENSOR (DIG_PERI) 寄存器

本小节的所有地址均为相对于 ADC 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 4 系统和存储器 中的表 4.3-3。

Register 39.56. APB_SARADC_CTRL_REG (0x0000)



APB_SARADC_START_FORCE 0：SAR ADC 由 FSM 启动；1：SAR ADC 由软件启动。(R/W)

APB_SARADC_START 通过软件启动 SAR ADC，仅当 APB_SARADC_START_FORCE = 1 时有效。(R/W)

APB_SARADC_SAR_CLK_GATED 当 SAR ADC 处于空闲状态时使能 SAR ADC 时钟门控。(R/W)

APB_SARADC_SAR_CLK_DIV SAR ADC 的时钟分频系数。(R/W)

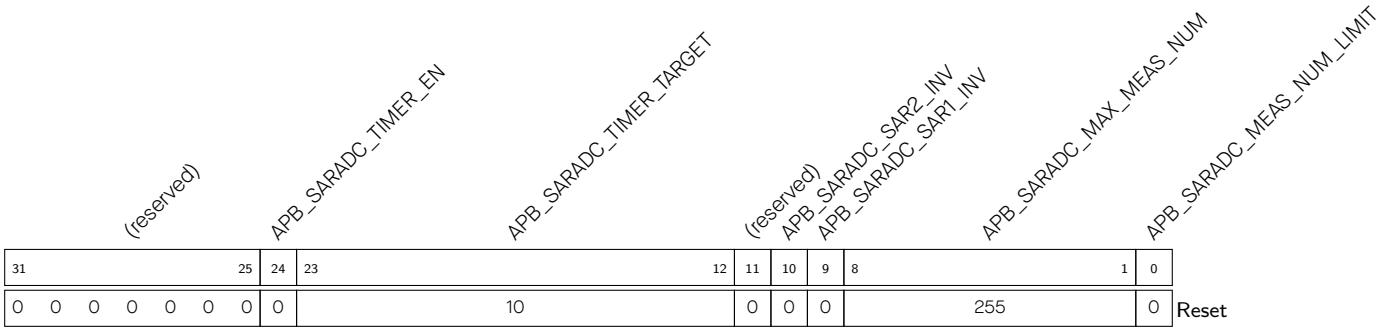
APB_SARADC_SAR1_PATT_LEN 配置 SAR ADC1 需要使用的样式数量。如果此字段设置为 1，则将使用样式表中的样式 0 (cmd0) 和样式 1 (cmd1)。(R/W)

APB_SARADC_SAR1_PATT_P_CLEAR 清除 DIG ADC1 控制器的样式表指针。(R/W)

APB_SARADC_XPD_SAR_FORCE 强制选择 XPD SAR。(R/W)

APB_SARADC_WAIT_ARB_CYCLE SAR_DONE 信号发出后至仲裁信号稳定需等待的时钟周期。(R/W)

Register 39.57. APB_SARADC_CTRL2_REG (0x0004)



APB_SARADC_MEAS_NUM_LIMIT 使能 SAR ADC 最大转换次数限制。(R/W)

APB_SARADC_MAX_MEAS_NUM 设置最大转换次数。(R/W)

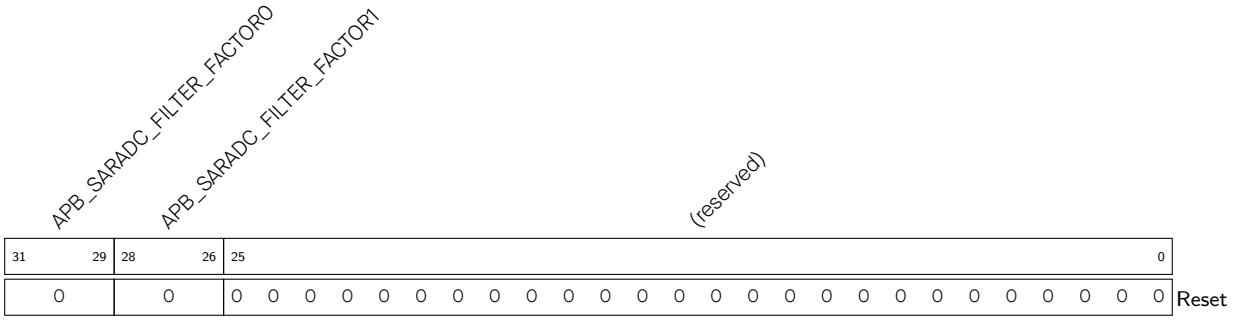
APB_SARADC_SAR1_INV 写 1 反转 DIG ADC1 数据。(R/W)

APB_SARADC_SAR2_INV 写 1 反转 DIG ADC1 数据。(R/W)

APB_SARADC_TIMER_TARGET 设置 SAR ADC 定时器目标，即定时器的触发周期。(R/W)

APB_SARADC_TIMER_EN 使能 SAR ADC 定时器触发。(R/W)

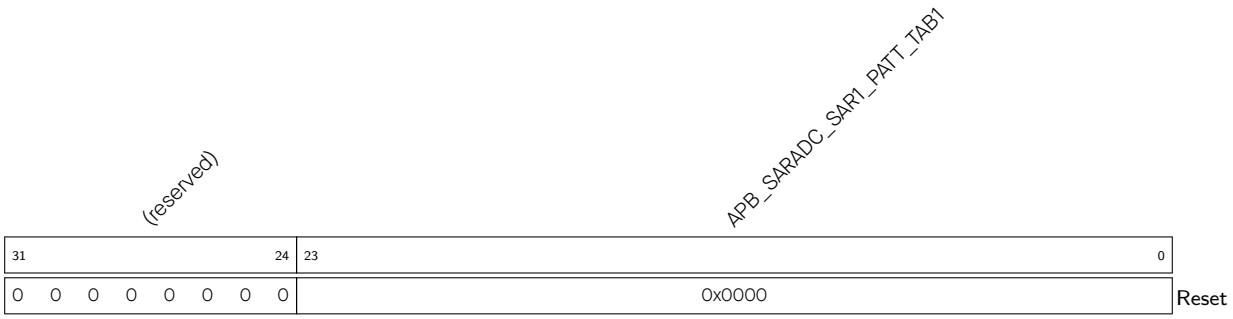
Register 39.58. APB_SARADC_FILTER_CTRL1_REG (0x0008)



APB_SARADC_FILTER_FACTOR1 配置 SAR ADC 滤波器 1 的滤波系数。(R/W)

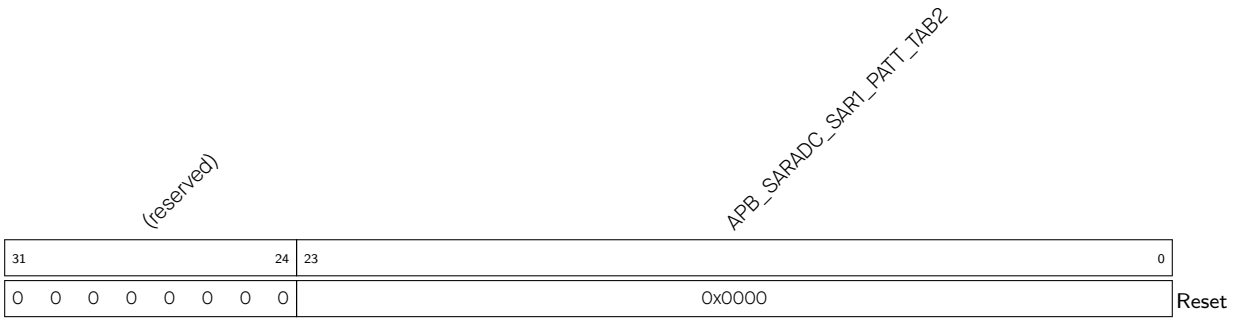
APB_SARADC_FILTER_FACTOR0 配置 SAR ADC 滤波器 0 的滤波系数。(R/W)

Register 39.59. APB_SARADC_SAR1_PATT_TAB1_REG (0x0018)



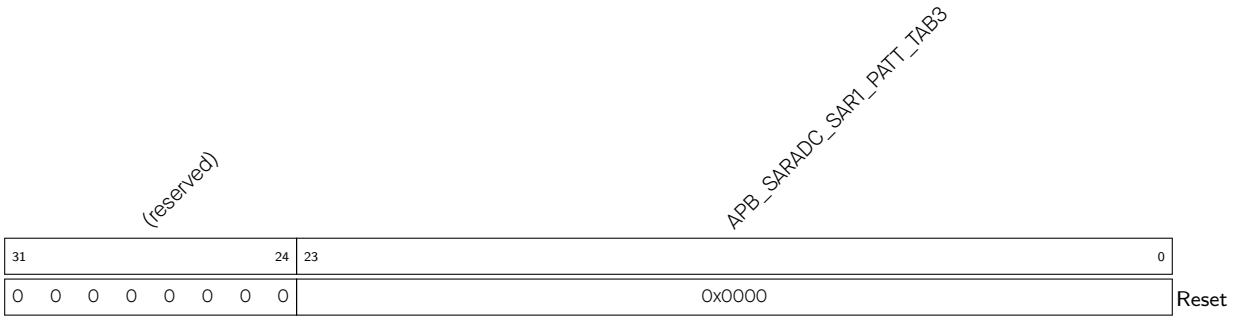
APB_SARADC_SAR1_PATT_TAB1 样式表 1 的样式 0 ~ 3。每个样式占 6 位。(R/W)

Register 39.60. APB_SARADC_SAR1_PATT_TAB2_REG (0x001C)



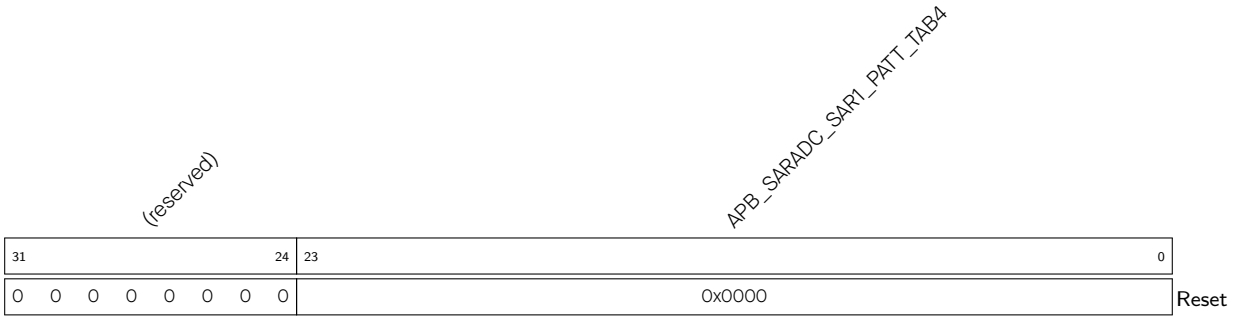
APB_SARADC_SAR1_PATT_TAB2 样式表 1 的样式 4 ~ 7。每个样式占 6 位。(R/W)

Register 39.61. APB_SARADC_SAR1_PATT_TAB3_REG (0x0020)



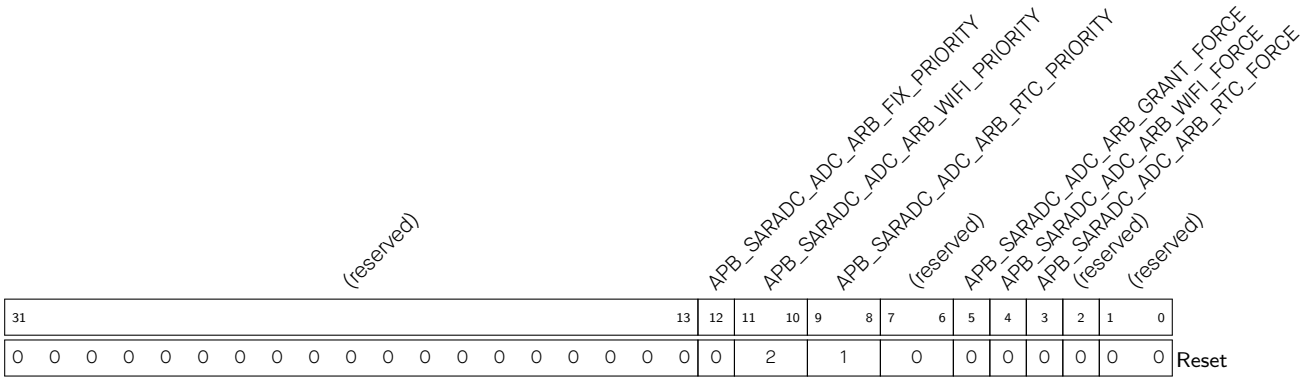
APB_SARADC_SAR1_PATT_TAB3 样式表 1 的样式 8 ~ 11。每个样式占 6 位。(R/W)

Register 39.62. APB_SARADC_SAR1_PATT_TAB4_REG (0x0024)



APB_SARADC_SAR1_PATT_TAB4 样式表 1 的样式 12 ~ 15。每个样式占 6 位。(R/W)

Register 39.63. APB_SARADC_APB_ADC_ARB_CTRL_REG (0x0038)



APB_SARADC_ADC_ARB_RTC_FORCE SAR ADC2 仲裁器强制使能 RTC ADC2 控制器。(R/W)

APB_SARADC_ADC_ARB_WIFI_FORCE SAR ADC2 仲裁器强制使能 PWDET 控制器。(R/W)

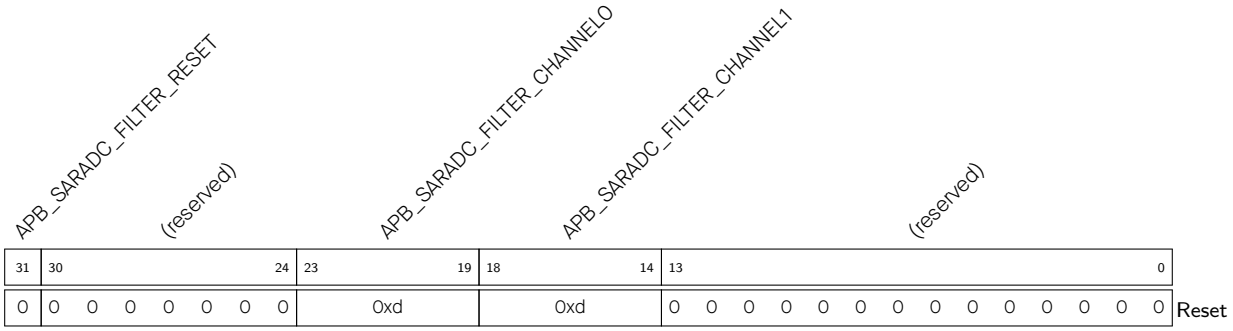
APB_SARADC_ADC_ARB_GRANT_FORCE SAR ADC2 仲裁器强制授权。(R/W)

APB_SARADC_ADC_ARB_RTC_PRIORITY 设置 RTC ADC2 控制器的优先级。(R/W)

APB_SARADC_ADC_ARB_WIFI_PRIORITY 设置 PWDET 控制器的优先级。(R/W)

APB_SARADC_ADC_ARB_FIX_PRIORITY 设置 SAR ADC2 仲裁器使用固定优先级。(R/W)

Register 39.64. APB_SARADC_FILTER_CTRL0_REG (0x003C)

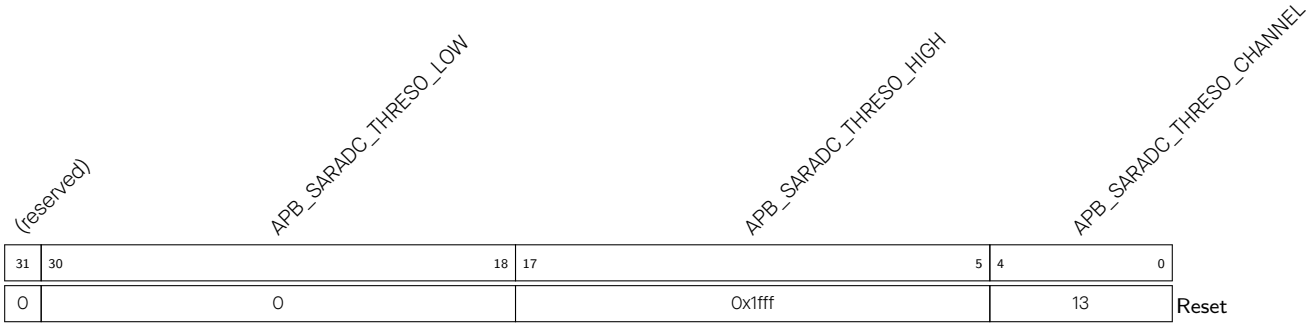


APB_SARADC_FILTER_CHANNEL1 配置 SAR ADC 滤波器 1 的滤波通道。(R/W)

APB_SARADC_FILTER_CHANNEL0 配置 SAR ADC 滤波器 0 的滤波通道。(R/W)

APB_SARADC_FILTER_RESET 复位 SAR ADC 滤波器。(R/W)

Register 39.65. APB_SARADC_THRES0_CTRL_REG (0x0044)

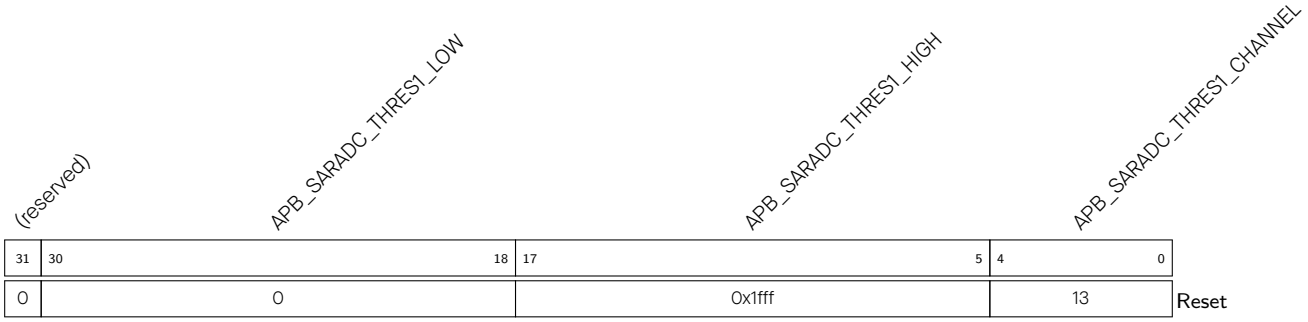


APB_SARADC_THRES0_CHANNEL 配置 SAR ADC 阈值监控器 0 需要监控的通道。(R/W)

APB_SARADC_THRES0_HIGH 设置 SAR ADC 阈值监控器 0 的高阈值。(R/W)

APB_SARADC_THRES0_LOW 设置 SAR ADC 阈值监控器 0 的低阈值。(R/W)

Register 39.66. APB_SARADC_THRES1_CTRL_REG (0x0048)

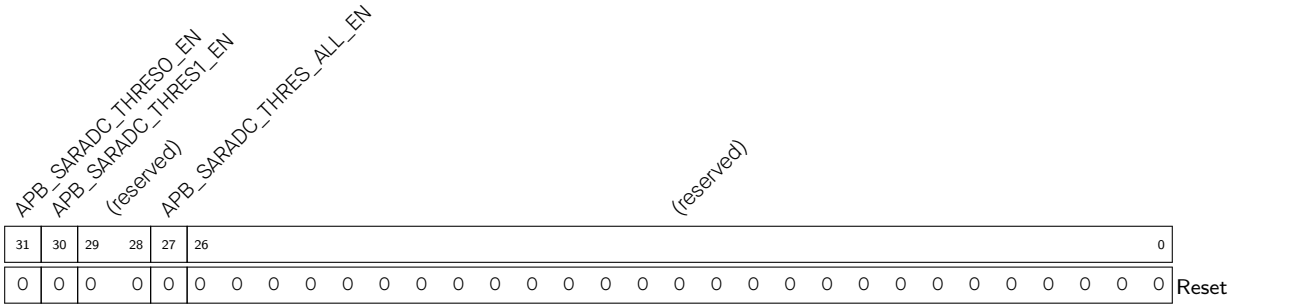


APB_SARADC_THRES1_CHANNEL 配置 SAR ADC 阈值监控器 1 需要监控的通道。(R/W)

APB_SARADC_THRES1_HIGH 设置 SAR ADC 阈值监控器 1 的高阈值。(R/W)

APB_SARADC_THRES1_LOW 设置 SAR ADC 阈值监控器 1 的低阈值。(R/W)

Register 39.67. APB_SARADC_THRES_CTRL_REG (0x0058)

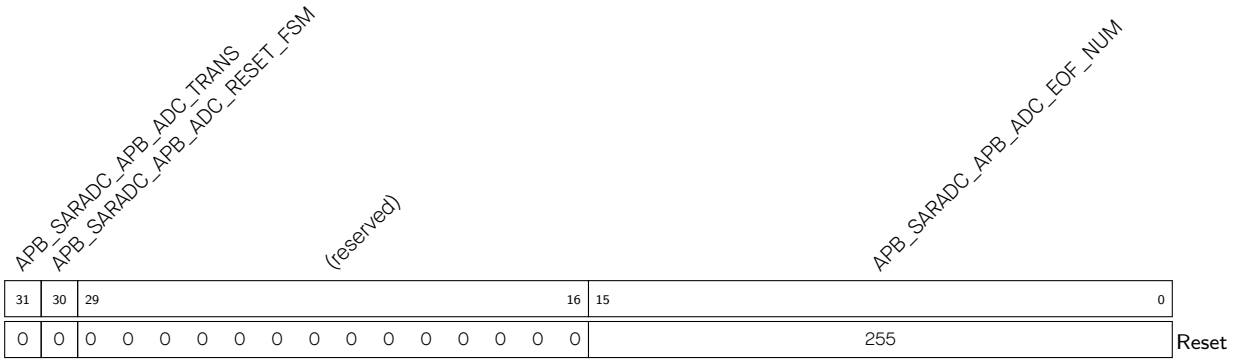


APB_SARADC_THRES_ALL_EN 使能所有已配置通道的阈值监控。(R/W)

APB_SARADC_THRES1_EN 使能阈值监控器 1。(R/W)

APB_SARADC_THRES0_EN 使能阈值监控器 0。(R/W)

Register 39.68. APB_SARADC_DMA_CONF_REG (0x006C)

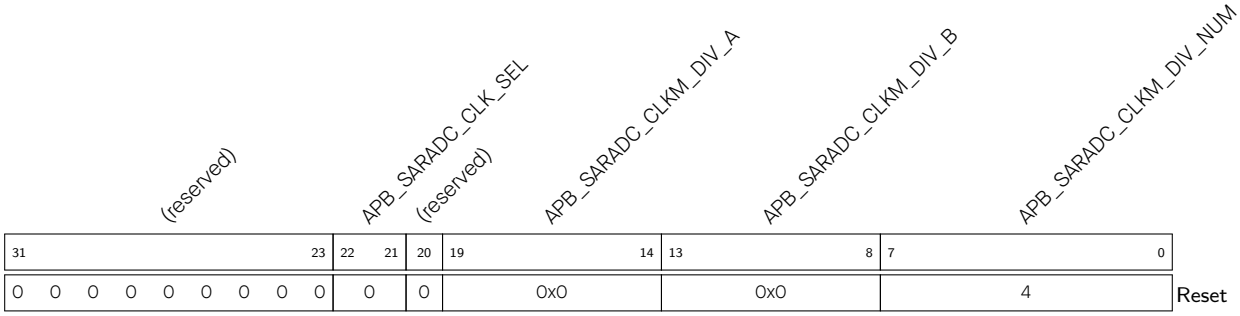


APB_SARADC_APB_ADC_EOF_NUM 当采样计数等于 apb_adc_eof_num 时，将生成 DMA_IN_SUC_EOF。(R/W)

APB_SARADC_APB_ADC_RESET_FSM 复位 DIG ADC 控制器状态。(R/W)

APB_SARADC_APB_ADC_TRANS 配置 DIG ADC 控制器使用 DMA。(R/W)

Register 39.69. APB_SARADC_APB_ADC_CLKM_CONF_REG (0x0070)



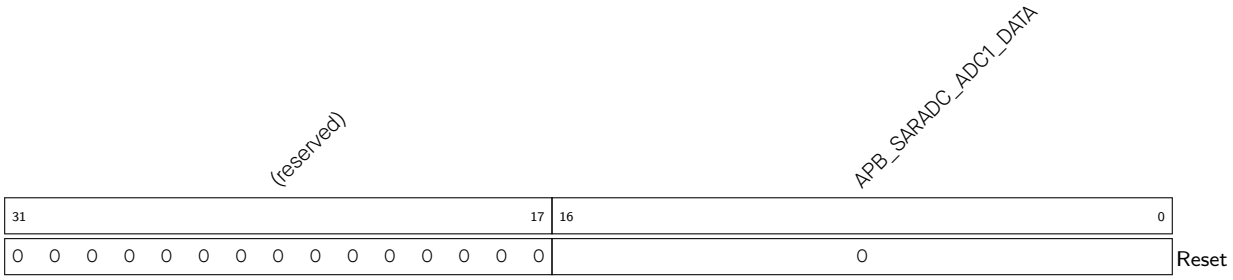
APB_SARADC_CLKM_DIV_NUM ADC 时钟分频系数的整数部分。分频系数 = APB_SARADC_CLKM_DIV_NUM + APB_SARADC_CLKM_DIV_B/ APB_SARADC_CLKM_DIV_A。(R/W)

APB_SARADC_CLKM_DIV_B 小数分频系数中的分子。(R/W)

APB_SARADC_CLKM_DIV_A 小数分频系数中的分母。(R/W)

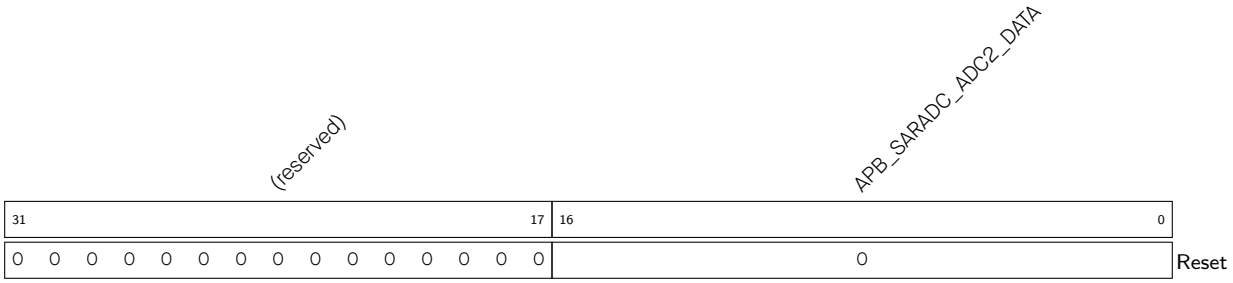
APB_SARADC_CLK_SEL 选择 DIG ADC 控制器的时钟源。0：关闭时钟；1：选择 PLL_D2_CLK；2：选择 APB_CLK。(R/W)

Register 39.70. APB_SARADC_APB_SARADC1_DATA_STATUS_REG (0x0040)



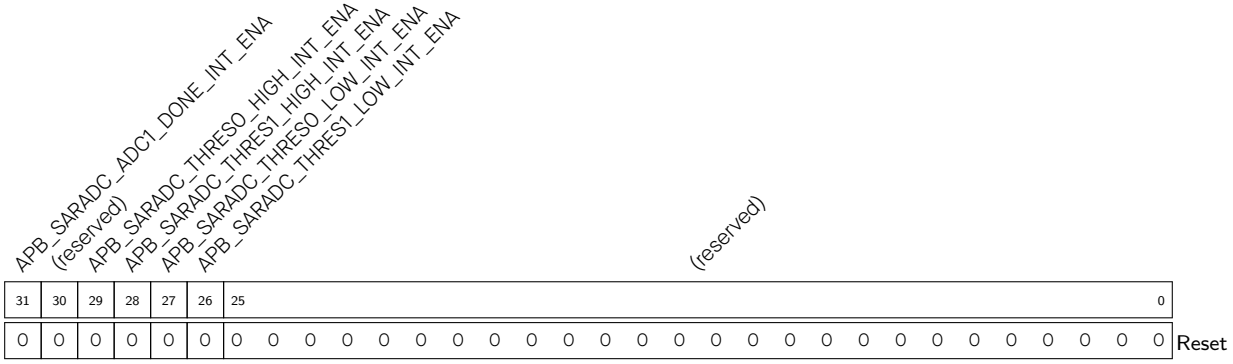
APB_SARADC_ADC1_DATA SAR ADC1 原始采样数据。(RO)

Register 39.71. APB_SARADC_APB_SARADC2_DATA_STATUS_REG (0x0078)



APB_SARADC_ADC2_DATA SAR ADC2 原始采样数据。(RO)

Register 39.72. APB_SARADC_INT_ENA_REG (0x005C)



APB_SARADC_THRES1_LOW_INT_ENA APB_SARADC_THRES1_LOW_INT 的中断使能位。(R/W)

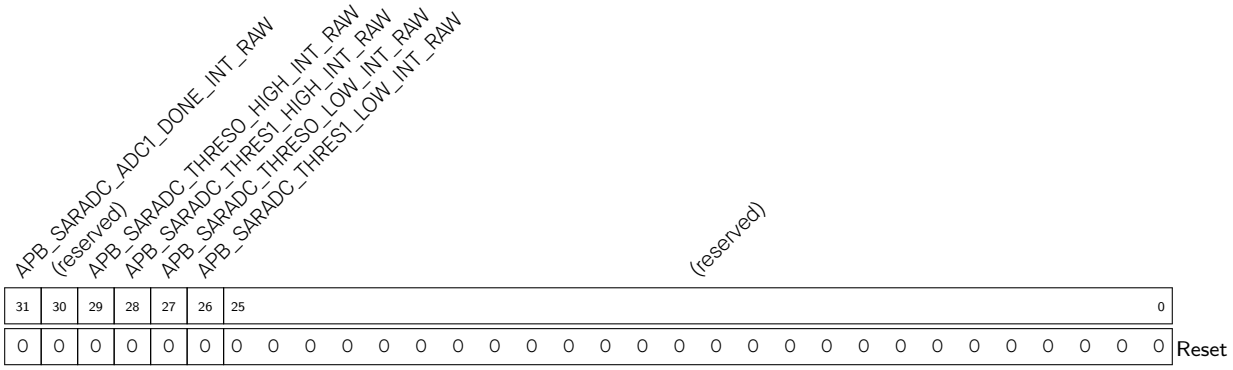
APB_SARADC_THRES0_LOW_INT_ENA APB_SARADC_THRES0_LOW_INT 的中断使能位。(R/W)

APB_SARADC_THRES1_HIGH_INT_ENA APB_SARADC_THRES1_HIGH_INT 的中断使能位。(R/W)

APB_SARADC_THRES0_HIGH_INT_ENA APB_SARADC_THRES0_HIGH_INT 的中断使能位。(R/W)

APB_SARADC_ADC1_DONE_INT_ENA APB_SARADC_ADC1_DONE_INT 的中断使能位。(R/W)

Register 39.73. APB_SARADC_INT_RAW_REG (0x0060)



APB_SARADC_THRES1_LOW_INT_RAW APB_SARADC_THRES1_LOW_INT 的原始中断位。(RO)

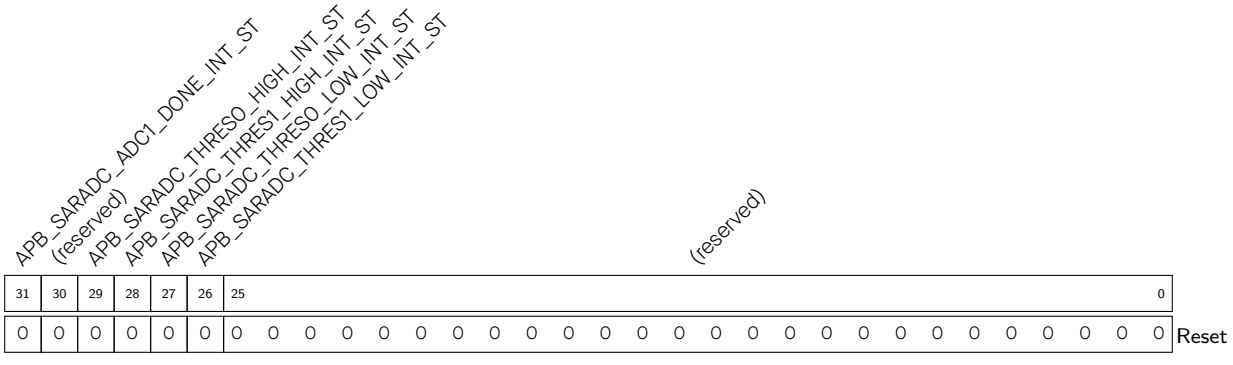
APB_SARADC_THRES0_LOW_INT_RAW APB_SARADC_THRES0_LOW_INT 的原始中断位。(RO)

APB_SARADC_THRES1_HIGH_INT_RAW APB_SARADC_THRES1_HIGH_INT 的原始中断位。(RO)

APB_SARADC_THRES0_HIGH_INT_RAW APB_SARADC_THRES0_HIGH_INT 的原始中断位。(RO)

APB_SARADC_ADC1_DONE_INT_RAW APB_SARADC_ADC1_DONE_INT 的原始中断位。(RO)

Register 39.74. APB_SARADC_INT_ST_REG (0x0064)



APB_SARADC_THRES1_LOW_INT_ST APB_SARADC_THRES1_LOW_INT 的中断状态位。(RO)

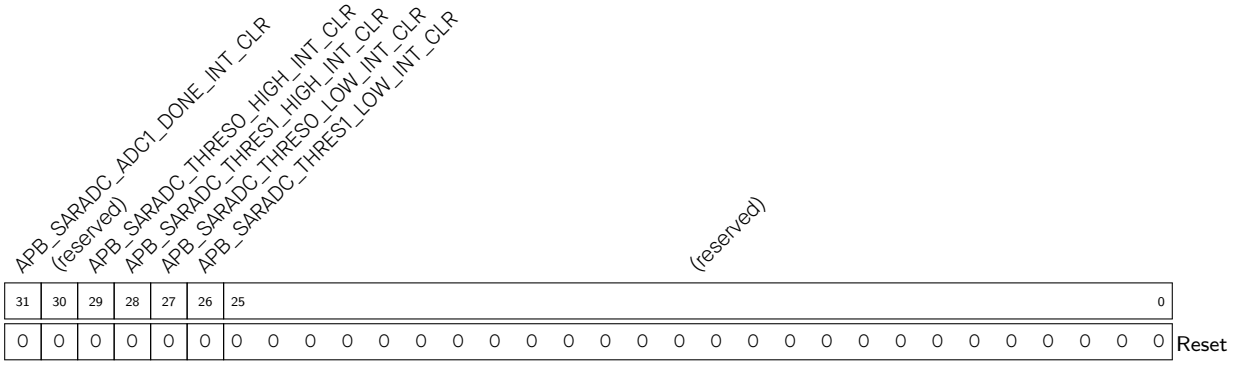
APB_SARADC_THRES0_LOW_INT_ST APB_SARADC_THRES0_LOW_INT 的中断状态位。(RO)

APB_SARADC_THRES1_HIGH_INT_ST APB_SARADC_THRES1_HIGH_INT 的中断状态位。(RO)

APB_SARADC_THRES0_HIGH_INT_ST APB_SARADC_THRES0_HIGH_INT 的中断状态位。(RO)

APB_SARADC_ADC1_DONE_INT_ST APB_SARADC_ADC1_DONE_INT 的中断状态位。(RO)

Register 39.75. APB_SARADC_INT_CLR_REG (0x0068)



APB_SARADC_THRES1_LOW_INT_CLR APB_SARADC_THRES1_LOW_INT 的中断清除位。(WO)

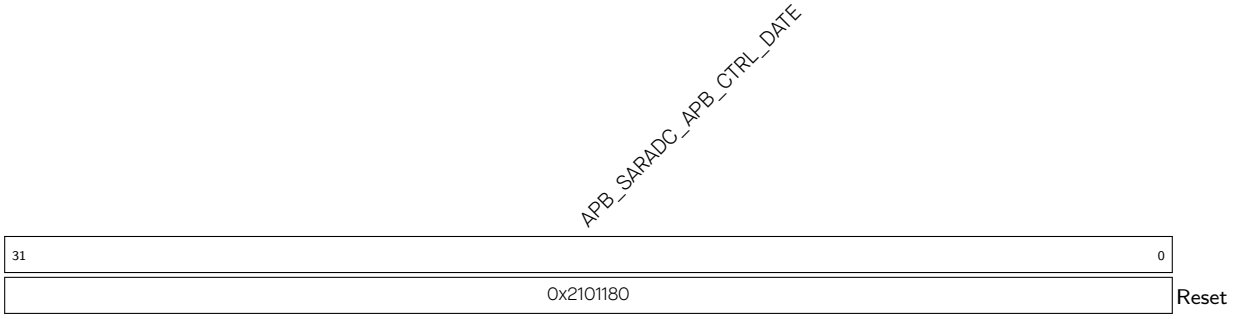
APB_SARADC_THRES0_LOW_INT_CLR APB_SARADC_THRES0_LOW_INT 的中断清除位。(WO)

APB_SARADC_THRES1_HIGH_INT_CLR APB_SARADC_THRES1_HIGH_INT 的中断清除位。(WO)

APB_SARADC_THRES0_HIGH_INT_CLR APB_SARADC_THRES0_HIGH_INT 的中断清除位。(WO)

APB_SARADC_ADC1_DONE_INT_CLR APB_SARADC_ADC1_DONE_INT 的中断清除位。(WO)

Register 39.76. APB_SARADC_APB_CTRL_DATE_REG (0x03FC)



APB_SARADC_APB_CTRL_DATE 版本控制寄存器。(R/W)

第 VII 卷

附录

附录部分包括以下内容：

- 相关文档和资源
- 词汇列表
- 如何配置寄存器的保留域
- 中断配置寄存器
- 修订历史

相关文档和资源

相关文档

- 《ESP32-S3 技术规格书》 – 提供 ESP32-S3 芯片的硬件技术规格。
- 《ESP32-S3 硬件设计指南》 – 提供基于 ESP32-S3 芯片的产品设计规范。
- 《ESP32-S3 系列芯片勘误表》 – 描述 ESP32-S3 系列芯片的已知错误。
- 证书
  https://espressif.com/zh-hans/support/documents/certificates
- ESP32-S3 产品/工艺变更通知 (PCN)
  https://espressif.com/zh-hans/support/documents/pcns?keys=ESP32-S3
- ESP32-S3 公告 – 提供有关安全、bug、兼容性、器件可靠性的信息
  https://espressif.com/zh-hans/support/documents/advisories?keys=ESP32-S3
- 文档更新和订阅通知
  https://espressif.com/zh-hans/support/download/documents

开发者社区

- 《ESP32-S3 ESP-IDF 编程指南》 – ESP-IDF 开发框架的文档中心。
- ESP-IDF 及 GitHub 上的其它开发框架
  https://github.com/espressif
- ESP32 论坛 – 工程师对工程师 (E2E) 的社区，您可以在这里提出问题、解决问题、分享知识、探索观点。
  https://esp32.com/
- *ESP-FAQ* – 由乐鑫官方推出的针对常见问题的总结。
  https://espressif.com/projects/esp-faq/zh_CN/latest/index.html
- *The ESP Journal* – 分享乐鑫工程师的最佳实践、技术文章和工作随笔。
  https://blog.espressif.com/
- SDK 和演示、App、工具、AT 等下载资源
  https://espressif.com/zh-hans/support/download/sdks-demos

产品

- ESP32-S3 系列芯片 – ESP32-S3 全系列芯片。
  https://espressif.com/zh-hans/products/socs?id=ESP32-S3
- ESP32-S3 系列模组 – ESP32-S3 全系列模组。
  https://espressif.com/zh-hans/products/modules?id=ESP32-S3
- ESP32-S3 系列开发板 – ESP32-S3 全系列开发板。
  https://espressif.com/zh-hans/products/devkits?id=ESP32-S3
- ESP Product Selector（乐鑫产品选型工具） – 通过筛选性能参数、进行产品对比快速定位您所需要的产品。
  https://products.espressif.com/#/product-selector?language=zh

联系我们

- 商务问题、技术支持、电路原理图 & PCB 设计审阅、购买样品（线上商店）、成为供应商、意见与建议
  https://espressif.com/zh-hans/contact-us/sales-questions

词汇列表

外设相关词汇

AES	AES 加速器
BOOTCTRL	芯片 Boot 控制
DS	数字签名
DMA	DMA 控制器
eFuse	eFuse 控制器
HMAC	HMAC 加速器
I2C	I2C 控制器
I2S	I2S 控制器
LEDC	LED 控制 PWM
MCPWM	电机控制 PWM
PCNT	脉冲计数器控制器
RMT	红外遥控
RNG	随机数生成器
RSA	RSA 加速器
SDHOST	SD/MMC 主机控制器
SHA	SHA 加速器
SPI	SPI 控制器
SYSTIMER	系统定时器
TIMG	定时器组
TWAI	双线汽车接口
UART	UART 控制器
ULP 协处理器	超低功耗协处理器
USB OTG	USB On-The-Go
WDT	看门狗定时器

寄存器相关缩写

REG	**寄存器**。
SYSREG	**系统寄存器**是一组控制系统复位、存储器、时钟、软件中断、电源管理、时钟门控等的寄存器。
ISO	**隔离**。如果外设或其他芯片组件断电，其输出信号的管脚（若有）将会浮空。ISO 寄存器会隔离上述引脚并令其保持在某个确定值，以使连接到这些引脚的其他非断电外设/设备免受影响。
NMI	**非屏蔽中断**是一种 CPU 指令无法禁用或忽略的硬件中断。出现此类中断说明发生严重错误。
W1TS	添加到寄存器/字段名称中的缩写，表示此类寄存器/字段用于置位名称相似寄存器中的相应字段。例如，寄存器 `GPIO_ENABLE_W1TS_REG` 用于置位寄存器 `GPIO_ENABLE_REG` 中的相应字段。
W1TC	与 *W1TS* 相同，但用于清除相应寄存器中的字段。

寄存器的访问类型

TRM 章节 寄存器列表 和 寄存器 详述了寄存器及其字段的访问类型。

常用访问类型及组合如下：

- 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

下文提供了所有访问类型的具体描述。

R	**软件可读**。用户软件可以读取此寄存器/字段；通常与其他访问类型结合使用。
RO	**软件只读**。用户软件只可读取此寄存器/字段。
HRO	**硬件只读**。仅硬件可以读取此寄存器/字段；用于存储变量参数的默认设置。
W	**软件可写**。用户软件可以写入此寄存器/字段；通常与其他访问类型结合使用。
WO	**软件只写**。用户软件只可写入此寄存器/字段。
W1	**软件只写一次**。用户软件只可写入一次此寄存器/字段；只允许写入 1，写入 0 无效。
SS	**硬件置位**。在指定事件中，硬件自动将 1 写入此寄存器/字段；与一位字段一同使用。
SC	**硬件清零**。在指定事件中，硬件自动将 0 写入此寄存器/字段；与一位和多位字段一同使用。
SM	**硬件修改**。在指定事件中，硬件自动将指定值写入此寄存器/字段；与多位字段一同使用。
SU	**硬件更新**。在指定事件中，硬件自动更新此寄存器/字段；与多位字段一同使用。
RS	**软件读置位**。如果用户软件读取此寄存器/字段，硬件会自动写 1。
RC	**软件读清零**。如果用户软件读取此寄存器/字段，硬件会自动写 0。
RF	**软件读 FIFO**。如果用户软件将新数据写入 FIFO，寄存器/字段会自动读取。
WF	**软件写 FIFO**。如果用户软件将新数据写入此寄存器/字段，寄存器/字段会自动通过 APB 总线将数据传递到 FIFO。
WS	**软件写置位**。如果用户软件写入此寄存器/字段，硬件会自动置位此寄存器/字段。
W1S	**软件写 1 置位**。如果用户软件将 1 写入此寄存器/字段，硬件会自动置位此寄存器/字段。
WOS	**软件写 0 置位**。如果用户软件将 0 写入此寄存器/字段，硬件会自动置位此寄存器/字段。
WC	**软件写清零**。如果用户软件写入此寄存器/字段，硬件会自动清零此寄存器/字段。
W1C	**软件写 1 清零**。如果用户软件将 1 写入此寄存器/字段，硬件会自动清零此寄存器/字段。
WOC	**软件写 0 清零**。如果用户软件将 0 写入此寄存器/字段，硬件会自动清零此寄存器/字段。
WT	**软件写产生边沿触发信号**。如果用户软件将 1 写入此字段，将会产生边沿触发信号（APB 总线中的脉冲）或清除相应的 WTC 字段（详见 WTC）。

WTC **软件写其他寄存器位清零本寄存器位**。如果用户软件将 1 写入相应的 WT 字段，硬件会自动清除此字段（详见 WT）。

W1T **软件写 1 取反**。如果用户软件将 1 写入此字段，硬件会自动取反相应字段，否则不会取反。

W0T **软件写 0 取反**。如果用户软件将 0 写入此字段，硬件会自动取反相应字段，否则不会取反。

WL **软件仅在锁禁用时写**。如果锁被禁用，用户软件可以写入此寄存器/字段。

varies **访问类型不定**。此寄存器中的不同字段访问类型可能不同。

如何配置寄存器的保留域

概述

寄存器的保留域指的是寄存器中不对用户开放、或者配置为非默认值时会导致不可预测的结果的域。

如何配置保留域

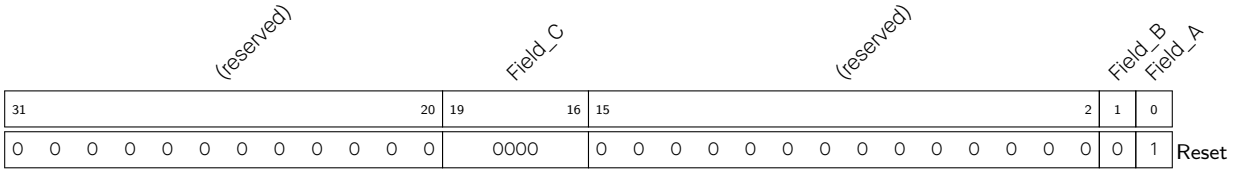
保留域的值不能修改。由于写寄存器时必须整体写，不能只写部分域，因此，在写带有保留域的寄存器时需要特别注意，只能采用以下两种方式：

1. 读取寄存器的值，仅修改需要配置的域，然后将修改后的值以及其他未修改的值一起写回寄存器，这样保留域的值就会保持不变。

   或者

2. 仅修改需要配置的域，然后将保留域的默认值写回寄存器。默认值即寄存器图表中的“Reset”值。例如，寄存器 X 中 Field_A 的默认值为 1。

Register 39.77. 寄存器 X (地址)



假设您要将 寄存器 X 的 Field_A、Field_B 和 Field_C 设置为 0x0、0x1 和 0x2，您可以：

- 使用第 1 种方式，修改这三个域的值，然后把读到的值写回寄存器。假设寄存器读取的值为 0x0000_0003，修改三个域的值后，将值 0x0002_0002 写入寄存器。
- 使用第 2 种方式，修改这三个域的值，然后把保留域的默认值写回寄存器，即把 0x0002_0002 写入寄存器。

中断配置寄存器

大部分外设的内部中断源都有以下配置寄存器：

- **RAW**（原始状态）寄存器：该寄存器指示原始中断状态，每个位对应一个内部中断源。当中断源触发时，其 RAW 位为 1。
- **ENA**（使能）寄存器：该寄存器用于启用或禁用内部中断源，每个位对应一个内部中断源。

  通过操作 ENA 寄存器，可以根据需要屏蔽或取消屏蔽某个内部中断源。当中断源被屏蔽（禁用）时，它不会生成中断信号，但仍可以从 RAW 寄存器中读取其值。
- **ST**（状态）寄存器：该寄存器指示中断源的屏蔽状态，每个位对应一个内部中断源。ST 位为 1 代表 RAW 位和 ENA 位都为 1，即中断源已生成且未被屏蔽。RAW 位和 ENA 位的值为其他组合时，ST 位为 0。

  ENA/RAW/ST 寄存器的配置见表 39.7-4。
- **CLR**（清除）寄存器：CLR 寄存器负责清除内部中断源。写 1 将清除该位对应的中断源。

表 39.7-4. ENA/RAW/ST 寄存器的配置

ENA 位的值	RAW 位的值	ST 位的值
0	忽略	0
1	0	0
	1	1

修订历史

日期	版本	发布说明
2026-03-04	v1.8	更新以下章节： • 章节 4 系统和存储器：更新图 4.2-1，增加图例 • 章节 6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)：更新寄存器 GPIO_STRAP_REG 的描述 • 章节 27 I2C 控制器 (I2C)：更新从机启动方式的描述 • 章节 32 USB OTG (USB)：更新表 32.4-1 • 章节 34 SD/MMC 主机控制器 (SDHOST)： – 更新寄存器列表 – 更新时钟配置相关描述 • 章节 37 红外遥控 (RMT)：更新 RMT 时钟频率限制相关公式
2025-06-09	v1.7	更新以下章节： • 章节 4 系统和存储器：在图 4.3-2 中新增 RMT 相关信息 • 章节 6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)：更新图 6.3-1 • 章节 8 芯片 Boot 控制：更新 strapping 管脚的锁存条件 • 章节 29 LCD 与 Camera 控制器 (LCD_CAM)：更新寄存器 LCD_CAM_LCD_HB_FRONT、LCD_CAM_LCD_VB_FRONT、LCD_CAM_LCD_HSYNC_POSITION、LCD_CAM_LCD_HSYNC_WIDTH 的描述 • 章节 39 片上传感器与模拟信号处理：将 ADC 滤波公式中的 “-0.5” 更正为 “+0.5”
2024-12-10	v1.6	更新以下章节： • 章节 6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)：更新 Hold 特性相关的描述 • 章节 10 低功耗管理 (RTC_CNTL)： – 更新预设功耗模式相关的描述 – 增加表 10.4-3 下方有关 EXT1 的说明 – 将表 10.4-3 中 TOUCH Active 标为可用作 Deep-sleep 的唤醒源 • 章节 35 LED PWM 控制器 (LEDC)：更新 LEDC_DUTY_START_CH*n* 位的描述 • 章节 39 片上传感器与模拟信号处理： – 删除 SAR ADC2 测量内部电压/信号的相关描述 – 新增关于 RTC 外设电源域关闭时触摸唤醒源配置以及启用 TOUCH Timeout 作为唤醒源的说明 – 更新表 39.4-1 温度测量范围与温度偏移 中温度传感器的测量范围

见下页

接上页

日期	版本	发布说明
2024-4-18	v1.5	更新以下章节： • 章节 3 通用 DMA 控制器 (GDMA)：更新 suc_eof 和 EOF 标志的相关描述 • 章节 6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)：更新 GPIO17 和 GPIO18 的驱动强度 • 章节 22 数字签名 (DS) 和章节 8 芯片 Boot 控制：修正笔误 • 章节 26 UART 控制器 (UART)：更新清除 wake_up 的说明 • 章节 35 LED PWM 控制器 (LEDC)：更新表 35.2-1 中的最低精度
2024-01-30	v1.4	更新以下章节： • 章节 1 处理器指令拓展 (PIE)：在章节 1.6 增加数据交换指令列表；增加 LD.QR、ST.QR、以及 MV.QR 指令的描述；修正一处笔误 • 章节 5 eFuse 控制器 (EFUSE)：修改 EFUSE_PIN_POWER_SELECTION 寄存器的描述 • 章节 6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)：从表格 4.3-3 中删除辅助调试；更新章节 6.9 和表 6.14-2 中的描述 • 章节 8 芯片 Boot 控制：在小节 8.2 中添加 SPI Download Boot 相关信息，并将 Download Boot 重命名为 Joint Download Boot；新增 EFUSE_DIS_FORCE_DOWNLOAD 和 EFUSE_DIS_DOWNLOAD_MODE 对芯片 Boot 控制的细节描述 • 章节 9 中断矩阵 (INTERRUPT)：删除 ASSIST_DEBUG_INTR 中断源 • 章节 10 低功耗管理 (RTC_CNTL)：更新寄存器 RTC_CNTL_WDT_WKEY 的描述 • 章节 12 定时器组 (TIMG)：更新 TIMG_WDT_CLK_PRESCALE 的描述 • 章节 15 权限控制 (PMS)：删除 Debug Assist 的访问权限配置 • 章节 17 系统寄存器 (SYSTEM)：优化 SYSTEM_CONTROL_CORE_1_MESSAGE 寄存器域的描述 • 章节 26 UART 控制器 (UART)：更新产生 wake_up 所需的上升沿个数 • 章节 27 I2C 控制器 (I2C)：更新 I2C_COMD0_REG、I2C_SDA_FORCE_OUT 和 I2C_SCL_FORCE_OUT 的描述 • 章节 36 电机控制脉宽调制器 (MCPWM)：增加递增递减模式下相关配置的说明 新增章节 中断配置寄存器

见下页

接上页

日期	版本	发布说明
2023-07-04	v1.3	更新以下章节： • 章节 2 超低功耗协处理器 (ULP-FSM, ULP-RISC-V)：增加 RTC I2C op_code 的描述；在 SENS_SAR_I2C_CTRL_REG 寄存器中增加一些域；增加章节“中断处理” • 章节 3 通用 DMA 控制器 (GDMA)：更新 GDMA_IN_SUC_EOF_CH*n*_INT 中断和 GDMA_INLINK_DSCR_ADDR_CH*n* 字段的描述 • 章节 4 系统和存储器：更新表 4.3-3 和图 4.2-1 • 章节 6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)：更新对 IO_MUX_PAD_POWER_CTRL 的描述 • 章节 10 低功耗管理 (RTC_CNTL)：更新一处错误链接 • 章节 15 权限控制 (PMS) 和 17 系统寄存器 (SYSTEM)：更新寄存器前缀 APB_CTRL 为 SYSCON • 章节 26 UART 控制器 (UART)：更新图 UART 架构概况、图 UART 基本架构图、图 硬件流控图；更新停止位最大位数和相关描述 • 章节 32 USB OTG (USB)：增加对 USB OTG 寄存器获取方式的说明 • 章节 33 USB 串口/ JTAG 控制器 (USB_SERIAL_JTAG)：在 USB_SERIAL_JTAG_PULLUP_VALUE 位的描述中增加具体的配置上拉电阻值 • 章节 39 片上传感器与模拟信号处理：更新图 39.3-4 和图 39.3-5 的样式表编号 新增章节 如何配置寄存器的保留域
2023-03-02	v1.2	更新以下章节： • 章节 4 系统和存储器：更新章节 4.3.2 内部存储器 中 Internal ROM 1 的描述 • 章节 10 低功耗管理 (RTC_CNTL)：删除 UART 作为拒绝睡眠原因 • 章节 12 定时器组 (TIMG)：更新读取定时器值的步骤 • 章节 17 系统寄存器 (SYSTEM)：更新寄存器 SYSTEM_PERIP_RST_EN0_REG 和 SYSTEM_PERIP_CLK_EN0_REG 的描述 • 章节 26 UART 控制器 (UART)：新增终止状态 (break condition) 的相关描述 • 章节 33 USB 串口/JTAG 控制器 (USB_SERIAL_JTAG)：新增 USB-OTG Download 下载模式下 IO 焊盘状态变化相关描述 • 章节 35 LED PWM 控制器 (LEDC)：新增占空比精度计算公式和表 常用配置频率及精度 • 章节 39 片上传感器与模拟信号处理： 新增 APB_SARADC_APB_SARADC1_DATA_STATUS_REG 寄存器相关描述，更新 SENS_FORCE_XPD_SAR 相关描述，并新增关于触摸传感器限制的说明 其他微小改动

见下页

接上页

日期	版本	发布说明
2022-10-31	v1.1	更新以下章节： • 章节 1 处理器指令拓展 (PIE) • 章节 5 eFuse 控制器 (EFUSE) • 章节 39 片上传感器与模拟信号处理 更新 词汇列表 小节
2022-09-08	v1.0	新增以下章节： • 章节 1 处理器指令拓展 (PIE) 更新以下章节： • 章节 9 中断矩阵 (INTERRUPT) • 章节 5 eFuse 控制器 (EFUSE) • 章节 25 随机数发生器 (RNG) • 章节 32 USB OTG (USB) • 章节 10 低功耗管理 (RTC_CNTL)
2022-08-04	v0.8	新增以下章节： • 章节 28 I2S 控制器 (I2S) • 章节 30 SPI 控制器 (SPI) 更新以下章节： • 章节 5 eFuse 控制器 (EFUSE) • 章节 10 低功耗管理 (RTC_CNTL)
2022-06-30	v0.7	新增以下章节： • 章节 39 片上传感器与模拟信号处理 • 章节 10 低功耗管理 (RTC_CNTL) 更新以下章节： • 章节 4 系统和存储器 • 更新时钟名称： – FOSC_CLK 更新为 RC_FAST_CLK – FOSC_DIV_CLK 更新为 RC_FAST_DIV_CLK – RTC_CLK 更新为 RC_SLOW_CLK – SLOW_CLK 更新为 RTC_SLOW_CLK – FAST_CLK 更新为 RTC_FAST_CLK – PLL_80M_CLK 更新为 PLL_F80M_CLK – PLL_160M_CLK 更新为 PLL_F160M_CLK – PLL_240M_CLK 更新为 PLL_D2_CLK
2022-06-01	v0.6	更新以下章节： • 章节 3 通用 DMA 控制器 (GDMA) • 章节 5 eFuse 控制器 (EFUSE) • 章节 8 芯片 Boot 控制

见下页

接上页

日期	版本	发布说明
2022-04-06	v0.5	新增以下章节： • 章节 15 权限控制 (PMS) • 章节 16 World 控制器 (WCL) 更新以下章节： • 章节 4 系统和存储器
2022-02-24	v0.4	新增章节 29 LCD 与 Camera 控制器 (LCD_CAM) 更新以下章节： • 章节 2 超低功耗协处理器 (ULP-FSM, ULP-RISC-V) • 章节 5 eFuse 控制器 (EFUSE) • 章节 6 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX) • 章节 7 复位和时钟 • 章节 8 芯片 Boot 控制 • 章节 32 USB OTG (USB)
2021-12-16	v0.3	新增以下章节： • 章节 2 超低功耗协处理器 (ULP-FSM, ULP-RISC-V) • 章节 3 通用 DMA 控制器 (GDMA) • 章节 11 系统定时器 (SYSTIMER) • 章节 24 时钟毛刺检测 • 章节 27 I2C 控制器 (I2C) • 章节 36 电机控制脉宽调制器 (MCPWM) • 章节 37 红外遥控 (RMT) 更新以下章节： • 章节 5 eFuse 控制器 (EFUSE) • 章节 20 RSA 加速器 (RSA) • 章节 21 HMAC 加速器 (HMAC) • 章节 22 数字签名 (DS)
2021-09-30	v0.2	新增以下章节： • 章节 17 系统寄存器 (SYSTEM) • 章节 21 HMAC 加速器 (HMAC) • 章节 23 片外存储器加密与解密 (XTS_AES) • 章节 26 UART 控制器 (UART) • 章节 33 USB 串口/JTAG 控制器 (USB_SERIAL_JTAG) 更新以下章节： • 章节 5 eFuse 控制器 (EFUSE) • 章节 31 双线汽车接口 (TWAI®)
2021-07-09	v0.1	首次预发布



免责声明和版权公告

本文档中的信息，包括供参考的 URL 地址，如有变更，恕不另行通知。

本文档可能引用了第三方的信息，所有引用的信息均为“按现状”提供，乐鑫不对信息的准确性、真实性做任何保证。

乐鑫不对本文档的内容做任何保证，包括内容的适销性、是否适用于特定用途，也不提供任何其他乐鑫提案、规格书或样品在他处提到的任何保证。

乐鑫不对本文档是否侵犯第三方权利做任何保证，也不对使用本文档内信息导致的任何侵犯知识产权的行为负责。本文档在此未以禁止反言或其他方式授予任何知识产权许可，不管是明示许可还是暗示许可。

Wi-Fi 联盟成员标志归 Wi-Fi 联盟所有。蓝牙标志是 Bluetooth SIG 的注册商标。

文档中提到的所有商标名称、商标和注册商标均属其各自所有者的财产，特此声明。

版权归 © 2026 乐鑫信息科技（上海）股份有限公司。保留所有权利。

www.espressif.com