ESP32-S2

技术参考手册

预发布 V0.4

关于本手册

《ESP32-S2 技术参考手册》的目标读者群体是使用 ESP32-S2 芯片的应用开发工程师。本手册提供了关于 ESP32-S2 的具体信息，包括各个功能模块的内部架构、功能描述和寄存器配置等。

芯片的管脚描述、电气特性和封装信息等可以从《ESP32-S2 技术规格书》获取。

文档版本

请至乐鑫官网 https://www.espressif.com/zh-hans/support/download/documents 下载最新版本文档。

修订历史

请至文档最后页查看修订历史。

文档变更通知

用户可以通过乐鑫官网订阅页面 www.espressif.com/zh-hans/subscribe 订阅技术文档变更的电子邮件通知。

证书下载

用户可以通过乐鑫官网证书下载页面 www.espressif.com/zh-hans/certificates 下载产品证书。

免责申明和版权公告

目录

表格

插图

1. 系统和存储器

1.1 概述

ESP32-S2 采用哈佛结构 Xtensa® LX7 CPU 构成单核系统。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。

1.2 主要特性

- **地址空间**
  - 数据总线与指令总线总共有 4 GB（32 位）地址空间
  - 464 KB 内部存储器指令地址空间
  - 400 KB 内部存储器数据地址空间
  - 1.77 MB 外设地址空间
  - 7.5 MB 外部存储器指令虚地址空间
  - 14.5 MB 外部存储器数据虚地址空间
  - 320 KB 内部 DMA 地址空间
  - 10.5 MB 外部 DMA 地址空间
- **内部存储器**
  - 128 KB Internal ROM
  - 320 KB Internal SRAM
  - 8 KB RTC FAST Memory
  - 8 KB RTC SLOW Memory
- **外部存储器**
  - 最大支持 1 GB 外部 SPI flash
  - 最大支持 1 GB 外部 SPI RAM
- **DMA**
  - 9 个具有 DMA 功能的模块 / 外设

图 1-1 描述了系统结构与地址映射结构。

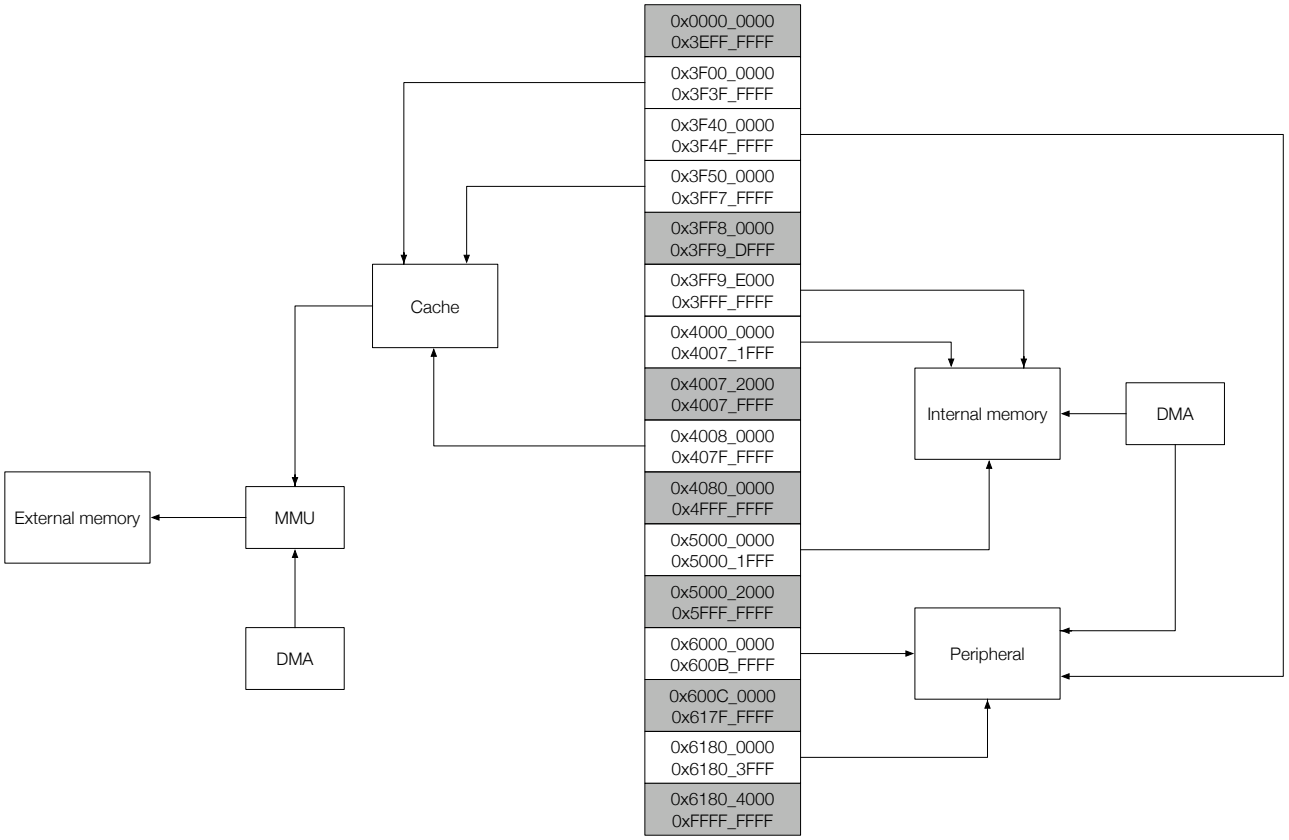


图 1-1. 系统结构与地址映射结构

说明：

- 图中灰色背景标注的地址空间不可用。
- 地址空间中可用的地址范围可能大于或小于实际可用的内存。

1.3 功能描述

1.3.1 地址映射

系统由一个哈佛结构 Xtensa® LX7 CPU 构成，CPU 具有 4 GB（32 位）的地址空间寻址能力。

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字节、半字、字对齐的数据访问。CPU 也可以通过指令总线进行数据访问，但必须是字对齐方式；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。

表 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。

表 1: 地址映射

总线类型	边界地址		容量	目标
	低位地址	高位地址		
	0x0000_0000	0x3EFF_FFFF		保留
数据	0x3F00_0000	0x3F3F_FFFF	4 MB	外部存储器
	0x3F40_0000	0x3F4F_FFFF	1 MB	外设
	0x3F50_0000	0x3FF7_FFFF	10.5 MB	外部存储器
	0x3FF8_0000	0x3FF9_DFFF		保留
数据	0x3FF9_E000	0x3FFF_FFFF	392 KB	内部存储器
指令	0x4000_0000	0x4007_1FFF	456 KB	内部存储器
	0x4007_2000	0x4007_FFFF		保留
指令	0x4008_0000	0x407F_FFFF	7.5 MB	外部存储器
	0x4080_0000	0x4FFF_FFFF		保留
数据 / 指令	0x5000_0000	0x5000_1FFF	8 KB	内部存储器
	0x5000_2000	0x5FFF_FFFF		保留
数据 / 指令	0x6000_0000	0x600B_FFFF	768 KB	外设
	0x600C_0000	0x617F_FFFF		保留
数据 / 指令	0x6180_0000	0x6180_3FFF	16 KB	外设
	0x6180_4000	0xFFFF_FFFF		保留

1.3.2 内部存储器

内部存储器分为四个部分：Internal ROM (128 KB)、Internal SRAM (320 KB)、RTC FAST Memory (8 KB)、RTC SLOW Memory (8 KB)。

128 KB Internal ROM 分为 64 KB Internal ROM 0、64 KB Internal ROM 1 两部分。

320 KB Internal SRAM 分为 32 KB Internal SRAM 0、288 KB Internal SRAM 1 两部分。

RTC FAST Memory 与 RTC SLOW Memory 都为 SRAM。

表 2 列出了所有内部存储器以及访问内部存储器的数据总线与指令总线地址段。

表 2: 内部存储器地址映射

总线类型	边界地址		容量	目标	权限管理
	低位地址	高位地址			
数据	0x3FF9_E000	0x3FF9_FFFF	8 KB	RTC FAST Memory	YES
	0x3FFA_0000	0x3FFA_FFFF	64 KB	Internal ROM 1	NO
	0x3FFB_0000	0x3FFB_7FFF	32 KB	Internal SRAM 0	YES
	0x3FFB_8000	0x3FFF_FFFF	288 KB	Internal SRAM 1	YES

总线类型	边界地址		容量	目标	权限管理
	低位地址	高位地址			
指令	0x4000_0000	0x4000_FFFF	64 KB	Internal ROM 0	NO
	0x4001_0000	0x4001_FFFF	64 KB	Internal ROM 1	NO
	0x4002_0000	0x4002_7FFF	32 KB	Internal SRAM 0	YES
	0x4002_8000	0x4006_FFFF	288 KB	Internal SRAM 1	YES
	0x4007_0000	0x4007_1FFF	8 KB	RTC FAST Memory	YES
总线类型	边界地址		容量	目标	权限管理
	低位地址	高位地址			
数据 / 指令	0x5000_0000	0x5000_1FFF	8 KB	RTC SLOW Memory	YES

说明：

表 2 中的“权限管理”一栏中标注为 YES，指总线在访问目标存储器时需要获取权限。用户可以配置权限控制寄存器对指令 / 数据总线访问目标存储器进行限制。

1.3.2.1 Internal ROM 0

Internal ROM 0 的容量为 64 KB，只读。如表 2 所示，CPU 只可以通过指令总线访问这部分存储器。

1.3.2.2 Internal ROM 1

Internal ROM 1 的容量为 64 KB，只读。如表 2 所示，CPU 可以通过数据或指令总线进行访问。

这两段地址同序访问 Internal ROM 1，即地址 0x3FFA_0000 与 0x4001_0000 访问到相同的字，0x3FFA_0004 与 0x4001_0004 访问到相同的字，0x3FFA_0008 与 0x4001_0008 访问到相同的字，以此类推。

1.3.2.3 Internal SRAM 0

Internal SRAM 0 的容量为 32 KB，可读可写。如表 2 所示，CPU 可以通过数据或指令总线同序访问。

通过配置，这部分存储器中的 8 KB、16 KB、24 KB 或全部 32 KB 可以被 cache 占用，用来缓存外部存储器的数据。被 cache 占用的部分不可以被 CPU 访问，未被 cache 占用的部分仍然可以被 CPU 访问。

1.3.2.4 Internal SRAM 1

Internal SRAM 1 容量为 288 KB，可读可写。如表 2 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 288 KB 地址空间由 18 个容量为 16 KB 的小存储器（子存储器）组成。其中至多有 1 个可以作为 Trace Memory 用于 CPU 的 Trace 功能，但 Trace Memory 无法被 CPU 访问。

1.3.2.5 RTC FAST Memory

RTC FAST Memory 为 8 KB SRAM，可读可写。如表 2 所示，CPU 可以通过数据或指令总线同序访问。

1.3.2.6 RTC SLOW Memory

RTC SLOW Memory 为 8 KB SRAM，可读可写。如表 2 所示，CPU 可以通过数据 / 指令总线的共用地址段访问这部分存储器。

RTC SLOW Memory 也可以被当作一个外设来使用，此时 CPU 可以通过数据总线的地址段 0x3F42_1000 ~ 0x3F42_2FFF 或地址段 0x6002_1000 ~ 0x6002_2FFF 来访问它。

1.3.3 外部存储器

ESP32-S2 支持多个外部 QSPI / OSPI flash 和随机存储器 (RAM)。ESP32-S2 还支持基于 XTS-AES 算法的硬件加解密功能，从而保护开发者 flash 和片外 RAM 中的程序和数据。

1.3.3.1 外部存储器地址映射

CPU 通过高速缓存 (cache) 来访问外部存储器。Cache 将根据 MMU 中的信息把 CPU 指令总线或数据总线的地址变换为访问外部 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 的外部 flash 与 1 GB 的片外 RAM。

通过高速缓存，ESP32-S2 一次最多可以同时有：

- 7.5 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 flash 或片外 RAM，支持字节、半字、字对齐的读访问。
- 4 MB 的只读数据总线地址空间，通过指令缓存 (ICache) 以 64 KB 为单位映射到 flash 或片外 RAM，支持字节、半字、字对齐的读访问。
- 10.5 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，支持字节、半字、字对齐的读写访问。这部分地址空间也可以用作只读数据空间，映射到 flash 与片外 RAM。

表 3 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 3: 外部存储器地址映射

总线类型	边界地址		容量	目标	权限管理
	低位地址	高位地址			
数据	0x3F00_0000	0x3F3F_FFFF	4 MB	ICache	YES
数据	0x3F50_0000	0x3FF7_FFFF	10.5 MB	DCache	YES
指令	0x4008_0000	0x407F_FFFF	7.5 MB	ICache	YES

说明：

表 3 中的“权限管理”一栏中标注为 YES，指总线在访问目标存储器时需要获取权限。用户可以配置权限控制寄存器对指令 / 数据总线访问目标存储器进行限制。

1.3.3.2 高速缓存

如图 1-2 所示，ESP32-S2 cache 采用分立结构，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复用（详见章节 1.3.2.3）。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 和 DCache 的缓存大小可配置为 8 KB 和 16 KB，块大小可配置为 16 B 和 32 B。

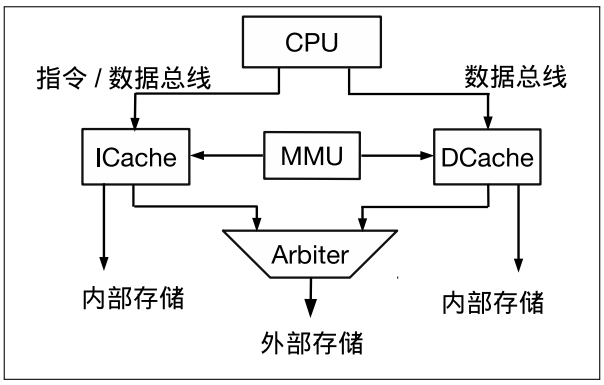


图 1-2. Cache 系统框图

1.3.3.3 Cache 操作

ESP32-S2 cache 共有如下几种操作：

1. **失效** (Invalidate)：清除 Tag 的有效标志位 (Valid bit)。如果 CPU 接着去访问该数据，那么需要访问外部存储器。失效分为自动失效和手动失效。手动失效仅对 cache 中落入指定区域的地址对应的数据做失效处理，而自动失效会对 cache 中的所有数据做失效处理。ICache 和 DCache 均具有此功能。
2. **清除** (Clean)：清除 Tag 的脏标志位 (Dirty bit)，保留有效标志位。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. **写回** (Write-back)：清除 Tag 的脏块标志位，保留有效标志位，并强制将对应地址的数据写回到外部存储器。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。
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。预取分为手动预取和自动预取，手动预取是指硬件按软件指定的虚地址预取一段连续的数据；自动预取是指硬件根据当前命中 / 缺失（取决于配置）的地址，自动预取一段连续的数据。
5. **锁定 / 解锁** (Lock / Unlock)：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 memory 时将落在指定区域的数据锁定，手动锁定开启时，cache 检查已填充到 cache memory 的数据，并将落在指定区域的数据锁定。锁定区域的数据会一直保存在 cache 中，而不会被替换掉。但当所有路都被锁定时, cache 将进行正常替换，就像 cache 没有被锁定一样。解锁是锁定的逆操作。Cache 的手动失效操作、清除和写回操作均需要在解锁后才可使用。

1.3.4 DMA 地址空间

ESP32-S2 可以借助直接存储访问 (direct memory access, DMA) 完成：

- 模块 / 外设与内部存储器之间的数据搬运；
- 内部存储器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；
- 内部存储器与外部存储器之间的数据搬运。

DMA 可以通过与 CPU 数据总线完全相同的地址访问 Internal SRAM 0 与 Internal SRAM 1，即使用地址 0x3FFB_0000 ~ 0x3FFB_7FFF 访问 Internal SRAM 0，使用地址 0x3FFB_8000 ~ 0x3FFF_FFFF 访问 Internal SRAM 1。但 DMA 无法访问被 cache 占用的内部存储器。

另外，DMA 可以使用与 CPU 访问 DCache 相同的地址 (0x3F50_0000 ~ 0x3FF7_FFFF) 来访问外部存储器，但只可以访问片外 RAM。当 DCache 与 DMA 同时访问外部存储器时，务必要注意数据一致性问题。

系统中具有 DMA 功能的模块 / 外设总共有 9 个，如表 4 所示。其中，有些模块 / 外设通过 DMA 只可以访问内部存储器，有些既可以访问内部存储器又可以访问外部存储器。

表 4: 具有 DMA 功能的模块 / 外设

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

1.3.5 模块 / 外设地址空间

CPU 可以通过数据总线 0x3F40_0000 ~ 0x3F4F_FFFF、数据 / 指令总线的共用地址段 0x6000_0000 ~ 0x600B_FFFF 和 0x6180_0000 ~ 0x6180_3FFF 来访问模块 / 外设。

1.3.5.1 外设总线名称约定

为了后续描述的方便，作如下约定：

- **PeriBus1**：总线地址段 0x3F40_0000 ~ 0x3F4F_FFFF 被记作“PeriBus1 总线”，简记为“PeriBus1”，它的基地址为 0x3F40_0000。
- **PeriBus2**：总线地址段 0x6000_0000 ~ 0x600B_FFFF 和 0x6180_0000 ~ 0x6180_3FFF 整体被记作“PeriBus2 总线”，简记为“PeriBus2”，它的基地址为 0x6000_0000。

在后续章节中出现的所有“PeriBus1”和“PeriBus2”都指代对应的总线地址段。

1.3.5.2 外设总线区别

相比于 CPU 通过 PeriBus2 访问模块 / 外设，CPU 通过 PeriBus1 访问模块 / 外设效率更高。但是 PeriBus1 有预测性读 (speculative read) 的特点，不能保证每一次的读访问都是真实有效的，因此，在访问模块 / 外设中的某些特殊寄存器如 FIFO 寄存器时，必须使用 PeriBus2 进行访问。

另外，PeriBus1 会打乱总线上的读写操作的先后顺序以提升性能，这可能会导致对读写操作的先后顺序有严格要求的程序发生崩溃，对于这种情况，请在程序语句之前增加 volatile，也可以改用 PeriBus2 进行访问。

1.3.5.3 模块 / 外设地址空间列表

表 5 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。请注意，“边界地址”一栏中的数值是相对于总线基地址的地址偏移量，而非绝对地址。访问某一模块 / 外设的绝对地址等于总线基地址加上相应的地址偏移量。

表 5: 模块 / 外设地址空间映射表

目标	边界地址		Size	说明
	低位地址	高位地址		
UART0	0x0000_0000	0x0000_0FFF	4 KB	1, 2, 3
保留	0x0000_1000	0x0000_1FFF		
SPI1	0x0000_2000	0x0000_2FFF	4 KB	1, 2
SPI0	0x0000_3000	0x0000_3FFF	4 KB	1, 2
GPIO	0x0000_4000	0x0000_4FFF	4 KB	1, 2
保留	0x0000_5000	0x0000_6FFF		
TIMER	0x0000_7000	0x0000_7FFF	4 KB	1, 2
RTC	0x0000_8000	0x0000_8FFF	4 KB	1, 2
IO MUX	0x0000_9000	0x0000_9FFF	4 KB	1, 2
保留	0x0000_A000	0x0000_EFFF		
I2S0	0x0000_F000	0x0000_FFFF	4 KB	1, 2, 3
UART1	0x0001_0000	0x0001_0FFF	4 KB	1, 2, 3
保留	0x0001_1000	0x0001_2FFF		
I2C0	0x0001_3000	0x0001_3FFF	4 KB	1, 2, 3
UHCI0	0x0001_4000	0x0001_4FFF	4 KB	1, 2
保留	0x0001_5000	0x0001_5FFF		
RMT	0x0001_6000	0x0001_6FFF	4 KB	1, 2, 3
PCNT	0x0001_7000	0x0001_7FFF	4 KB	1, 2
保留	0x0001_8000	0x0001_8FFF		
LED PWM Controller	0x0001_9000	0x0001_9FFF	4 KB	1, 2
eFuse Controller	0x0001_A000	0x0001_AFFF	4 KB	1, 2
保留	0x0001_B000	0x0001_EFFF		
Timer Group 0	0x0001_F000	0x0001_FFFF	4 KB	1, 2
Timer Group 1	0x0002_0000	0x0002_0FFF	4 KB	1, 2
RTC SLOW Memory	0x0002_1000	0x0002_2FFF	8 KB	1, 2, 3
System Timer	0x0002_3000	0x0002_3FFF	4 KB	1, 2
SPI2	0x0002_4000	0x0002_4FFF	4 KB	1, 2
SPI3	0x0002_5000	0x0002_5FFF	4 KB	1, 2
APB Controller	0x0002_6000	0x0002_6FFF	4 KB	1, 2
I2C1	0x0002_7000	0x0002_7FFF	4 KB	1, 2, 3
保留	0x0002_8000	0x0002_AFFF		
TWAI Controller	0x0002_B000	0x0002_BFFF	4 KB	1, 2
保留	0x0002_C000	0x0003_8FFF		
USB OTG	0x0003_9000	0x0003_9FFF	4 KB	1, 2, 3, 4
AES Accelerator	0x0003_A000	0x0003_AFFF	4 KB	1, 2
SHA Accelerator	0x0003_B000	0x0003_BFFF	4 KB	1, 2
RSA Accelerator	0x0003_C000	0x0003_CFFF	4 KB	1, 2
Digital Signature	0x0003_D000	0x0003_DFFF	4 KB	1, 2
HMAC	0x0003_E000	0x0003_EFFF	4 KB	1, 2
Crypto DMA	0x0003_F000	0x0003_FFFF	4 KB	1, 2
保留	0x0004_4000	0x000C_DFFF		

目标	边界地址		Size	说明
	低位地址	高位地址		
ADC Controller	0x0004_0000	0x0004_0FFF	4 KB	1, 2
保留	0x0004_1000	0x0007_FFFF		
USB	0x0008_0000	0x000B_FFFF	256 KB	1, 2, 3, 4
System Registers	0x000C_0000	0x000C_0FFF	4 KB	1
Sensitive Register	0x000C_1000	0x000C_1FFF	4 KB	1
Interrupt Matrix	0x000C_2000	0x000C_2FFF	4 KB	1
Copy DMA	0x000C_3000	0x000C_3FFF	4 KB	1
保留	0x000C_4000	0x000C_EFFF		
Dedicated GPIO	0x000C_F000	0x000C_FFFF	4 KB	1
保留	0x000D_1000	0x000F_FFFF		
Configure Cache	0x0180_0000	0x0180_3FFF	16 KB	2

说明：

1. 该模块 / 外设可以被 PeriBus1 总线访问。
2. 该模块 / 外设可以被 PeriBus2 总线访问。
3. 当 PeriBus1 总线访问该模块 / 外设时，不能读访问一些特殊的地址（详见 1.3.5.4 章节）。
4. 该模块 / 外设的地址空间不连续。

1.3.5.4 PeriBus1 访问受限地址列表

如前文 1.3.5.2 所述，PeriBus1 有预测性读 (speculative read) 的特点，因此被禁止读访问模块 / 外设的 FIFO 寄存器。表 6 列出了禁止被 PeriBus1 读访问的地址。另外，4 个用户自定义寄存器可以根据用户需求进行配置，用于将更多地址添加到访问受限地址列表中。

表 6: 访问受限的地址

目标外设	受限地址值 / 区间
UART0	0x3F40_0000
UART1	0x3F41_0000
I2S0	0x3F40_F004
RMT	0x3F41_6000 ~ 0x3F41_600F
I2C0	0x3F41_301C
I2C1	0x3F42_701C
USB OTG	0x3F48_0020, 0x3F48_1000 ~ 0x3F49_0FFF

2. 复位和时钟

2.1 复位

2.1.1 概述

系统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。

除芯片复位外其他复位方式不影响片上内存存储的数据。图 2-1 展示了整个芯片系统的结构以及四种复位等级。

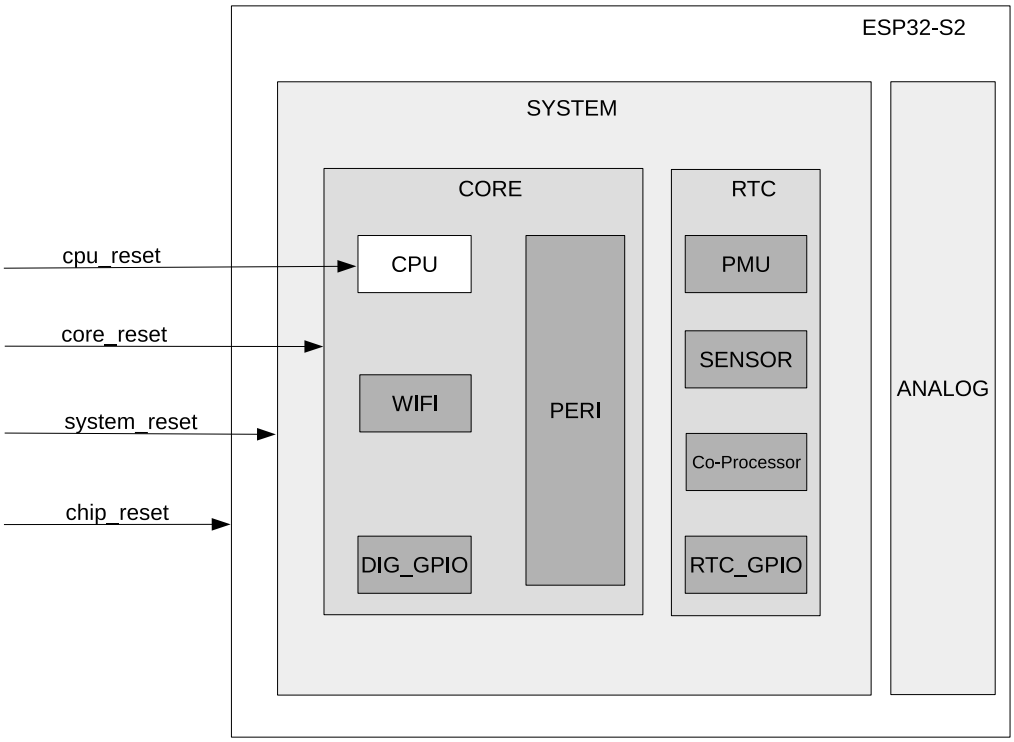


图 2-1. 四种复位等级

- CPU 复位：只复位 CPU core，复位释放后，程序将从 CPU reset vector 开始执行；
- 内核复位：复位除 RTC 以外的其他数字系统，包括 CPU、外设、Wi-Fi 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。

2.1.2 复位源

上述任一复位源产生时，CPU 均将立刻复位。复位释放后，CPU 可以通过读取寄存器 RTC_CNTL_RESET_CAUSE_PROCPU 来获取复位源。

表 7 列出了从这一寄存器中可能读出的复位源。

表 7: 复位源

编码	复位源	复位方式	注释
0x01	芯片复位	芯片复位	见表下方芯片复位的触发源
0x0F	欠压系统复位	系统复位	欠压检测器触发的系统复位

编码	复位源	复位方式	注释
0x10	RWDT 系统复位	系统复位	详见章节 13 看门狗定时器
0x13	GLITCH 复位	系统复位	-
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 28 低功耗管理
0x07	MWDT0 全局复位	内核复位	详见章节 13 看门狗定时器
0x08	MWDT1 全局复位	内核复位	详见章节 13 看门狗定时器
0x09	RWDT 内核复位	内核复位	详见章节 13 看门狗定时器
0x0B	MWDT0 CPU 复位	CPU 复位	详见章节13 看门狗定时器
0x0C	软件 CPU 复位	CPU 复位	配置 RTC_CNTL_SW_PROCPU_RST 寄存器触发
0x0D	RWDT CPU 复位	CPU 复位	详见章节 13 看门狗定时器
0x11	MWDT1 CPU 复位	CPU 复位	详见章节 13 看门狗定时器

注意：

- 芯片复位的触发源包括以下三项：
  - 芯片上电
  - 欠压检测器触发的芯片复位
  - 超级看门狗 (SWD) 触发的芯片复位
- 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 28 低功耗管理。

2.2 系统时钟

2.2.1 概述

ESP32-S2 提供了多种不同频率的时钟选择，可以灵活配置 CPU、外设、以及 RTC 的工作频率，以满足不同功耗和性能需求。下图 2-2 为系统时钟结构。

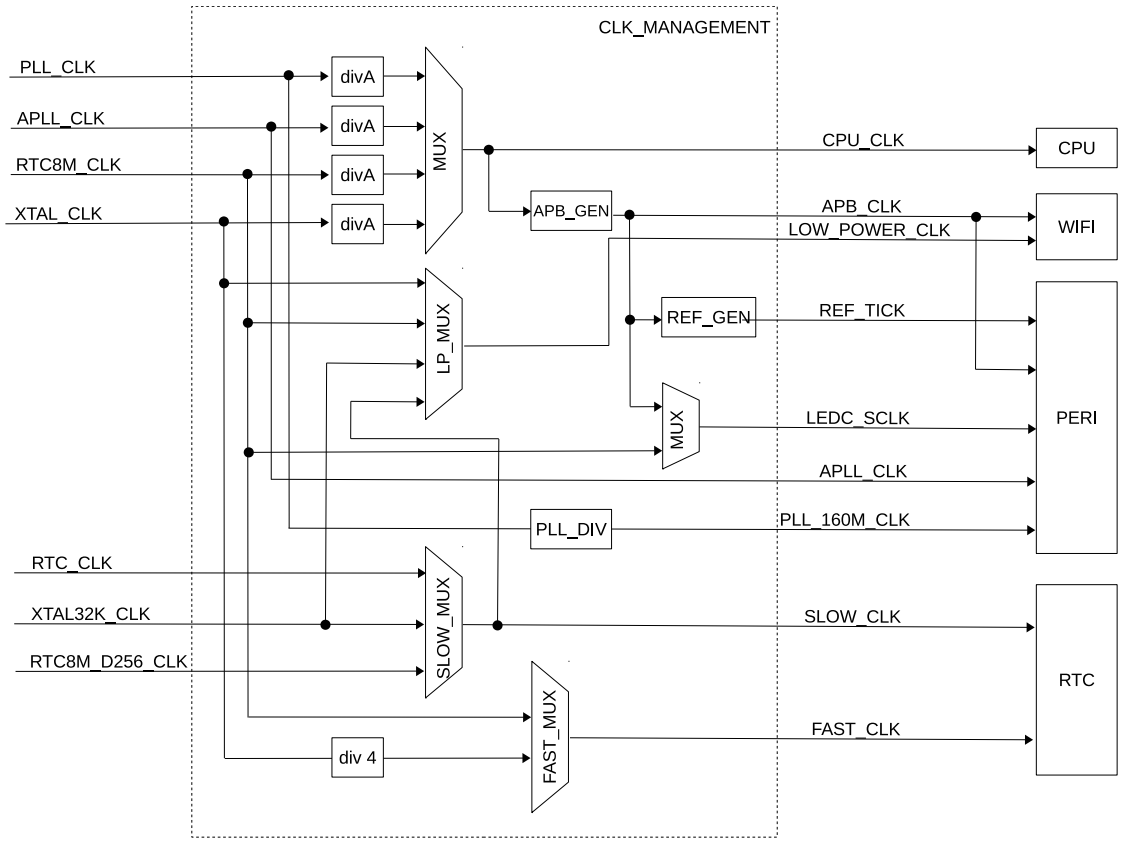


图 2-2. 系统时钟

2.2.2 时钟源

ESP32-S2 有三种时钟源：外部晶振、内部 PLL 和震荡电路，用于生成多种时钟。这些时钟根据频率不同可分为以下三种类型。

- 快速时钟：供 CPU 及数字外设等高速设备使用
  - PLL_CLK：320 MHz 或 480 MHz 内部 PLL 时钟；
  - XTAL_CLK：40 MHz 外部晶振时钟。
- 低功耗慢速时钟：供低功耗设备使用，包括功耗管理单元以及低功耗外设等
  - XTAL32K_CLK：32 kHz 外部晶振时钟；
  - RTC8M_CLK：8 MHz 内部时钟，频率可调；
  - RTC8M_D256_CLK：由 RTC8M_CLK 经 256 分频所得，频率为 RTC8M_CLK / 256。当 RTC8M_CLK 的初始频率为 8 MHz 时，该时钟以 31.250 kHz 的频率运行；
  - RTC_CLK：90 kHz 内部低功耗时钟，频率可调。
- 音频时钟：供音频相关设备使用
  - APLL_CLK：16 MHz ~ 128 MHz 内部 Audio PLL 时钟。

2.2.3 CPU 时钟

如图 2-2 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、APLL_CLK、RTC8M_CLK 或 XTAL_CLK 作为 CPU_CLK 的时钟源。具体请参考表 8 和表 9。

表 8: CPU_CLK 源

SYSTEM_SOC_CLK_SEL 值	时钟源
0	XTAL_CLK
1	PLL_CLK
2	RTC8M_CLK
3	APLL_CLK

表 9: CPU_CLK 选择

时钟源	SEL_0*	SEL_1*	SEL_2*	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。
RTC8M_CLK	2	-	-	CPU_CLK = RTC8M_CLK / (SYSTEM_PRE_DIV_CNT + 1) SYSTEM_PRE_DIV_CNT 默认值为 1，范围 0 ~ 1023。
APLL_CLK	3	0	0	CPU_CLK = APLL_CLK / 4
APLL_CLK	3	0	1	CPU_CLK = APLL_CLK / 2

*SEL_0: 寄存器 SYSTEM_SOC_CLK_SEL 的值；
*SEL_1: 寄存器 SYSTEM_PLL_FREQ_SEL 的值；
*SEL_2: 寄存器 SYSTEM_CPUPERIOD_SEL 的值。

注意：

- 当 CPU 选择 XTAL_CLK 用作时钟源，通过配置寄存器 SYSTEM_PRE_DIV_CNT 调节 XTAL_CLK 的时钟分频系数时，需要遵循以下规则：
  - 目标分频系数为 x 分频（x 不等于 1 分频）且当前分频系数为 2 分频时（SYSTEM_PRE_DIV_CNT = 1），需要先将分频系数配置为 1 分频（SYSTEM_PRE_DIV_CNT = 0），再调节分频系数为 x 分频（SYSTEM_PRE_DIV_CNT = x - 1）；
  - 目标分频系数为 2 分频且当前分频系数为 x 分频（SYSTEM_PRE_DIV_CNT = x - 1），需要先将分频系数调节为 1 分频（SYSTEM_PRE_DIV_CNT = 0），再调节分频系数为 2 分频（SYSTEM_PRE_DIV_CNT = 1）；
  - 如需调至其它目标分频系数 x，直接调整分频系数即可（SYSTEM_PRE_DIV_CNT = x - 1）。

2.2.4 外设时钟

外设所需要的时钟包括 APB_CLK、REF_TICK、LEDC_PWM_CLK、APLL_CLK 和 PLL_160M_CLK。下表 10 为接入各个外设的时钟。

表 10: 外设时钟用法

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

2.2.4.1 APB_CLK 源

如表 11 所示，APB_CLK 的频率由 CPU_CLK 的时钟源决定。

表 11: APB_CLK 源

CPU_CLK 源	APB_CLK 频率
PLL_CLK	80 MHz
APLL_CLK	CPU_CLK / 2
XTAL_CLK	CPU_CLK
RTC8M_CLK	CPU_CLK

2.2.4.2 REF_TICK 源

REF_TICK 由 XTAL_CLK 或 RTC8M_CLK 分频产生。当 CPU 时钟源为 PLL_CLK、APLL_CLK、XTAL_CLK 时，REF_TICK 由 XTAL_CLK 分频获得; 当 CPU 时钟源为内部 RTC8M_CLK 时，REF_TICK 由内部 RTC8M_CLK 时钟分频获得。由此可以保证 REF_TICK 在 APB_CLK 切换时维持频率不变。寄存器配置如表 12 所示:

表 12: REF_TICK 源

CPU_CLK 源	时钟分频寄存器
PLL_CLK | XTAL_CLK | APLL_CLK	APB_CTRL_XTAL_TICK_NUM
RTC8M_CLK	APB_CTRL_CK8M_TICK_NUM

通常 REF_TICK 的周期为 1 μs，所以 APB_CTRL_XTAL_TICK_NUM 需配置为 39；
APB_CTRL_CK8M_TICK_NUM 需配置为 7。

2.2.4.3 LEDC_PWM_CLK 源

LEDC_PWM_CLK 时钟源由寄存器 LEDC_APB_CLK_SEL 决定，如表 13 所示。

表 13: LEDC_PWM_CLK 源

LEDC_APB_CLK_SEL 值	LEDC_PWM_CLK 源
0（默认）	-
1	APB_CLK
2	RTC8M_CLK
3	XTAL_CLK

2.2.4.4 APLL_SCLK 源

APLL_CLK 来自内部 PLL_CLK，其输出频率通过使用 APLL 配置寄存器来配置。配置方式见第 2.2.7 节。

2.2.4.5 PLL_160M_CLK 源

PLL_160M_CLK 是 PLL_CLK 根据当前 PLL 的频率分频所得。

2.2.4.6 时钟源注意事项

需要与其他时钟配合工作的外设（如 RMT、I²C 等）一般在选择 PLL_CLK 时钟源的情况下工作。若频率发生变化，外设需要通过修改配置才能以同样的频率工作。接入 REF_TICK 的外设允许在切换时钟源的情况下，不修改外设配置即可工作。详情请参考表 10。

LED 模块能将 RTC8M_CLK 作为时钟源使用，即在 APB_CLK 关闭的时候，LED 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 LED 仍然可以通过 RTC8M_CLK 来正常工作。

2.2.5 Wi-Fi 时钟

Wi-Fi 必须在 APB_CLK 时钟源选择 PLL_CLK 下才能工作。只有当 Wi-Fi 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、RTC8M_CLK、SLOW_CLK（RTC 当前所选的慢速时钟）用于 Wi-Fi 的低功耗模式。

2.2.6 RTC 时钟

SLOW_CLK 和 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。

SLOW_CLK 允许选择 RTC_CLK、XTAL32K_CLK 或 RTC8M_D256_CLK，用于驱动功耗模块。

FAST_CLK 允许选择 XTAL_CLK 的分频时钟或 RTC8M_CLK，用于驱动片上传感器模块。

2.2.7 音频 PLL 时钟

音频应用和其他对于数据传输时效性要求很高的应用都需要高度可配置、低抖动并且精确的时钟源。来自系统时钟的时钟源可能会携带抖动，并且不支持高精度的时钟频率配置。

在满足应用需求的前提下，为了最大限度地降低系统成本，ESP32-S2 集成了专门用于 I²S 外设的音频 PLL。用户可使用 APLL 时钟对 I²S 模块进行计时。

Audio PLL 公式如下：

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

其中，

- f_{xtal}：晶振频率，通常为 40 MHz
- sdm0：可配参数 0 ~ 255
- sdm1：可配参数 0 ~ 255
- sdm2：可配参数 0 ~ 63
- odiv：可配参数 0 ~ 31
- 公式的分子频率工作范围在 350 MHz ~ 500 MHz

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

Audio PLL 可通过使能寄存器 RTC_CNTL_PLLA_FORCE_PU 强行打开，或者使能寄存器 RTC_CNTL_PLLA_FORCE_PD 强行关闭，关闭优先级大于打开优先级。当 RTC_CNTL_PLLA_FORCE_PU 和 RTC_CNTL_PLLA_FORCE_PD 同时为 0 时，PLL 会跟随系统状态，当系统进入睡眠模式时自动关闭，系统被唤醒时自动打开。

3. 芯片 Boot 控制

3.1 概述

ESP32-S2 共有三个 Strapping 管脚：

- GPIO0
- GPIO45
- GPIO46

软件可以从 GPIO_STRAPPING 寄存器中读取这三个 Strapping 管脚的值。在芯片复位过程中，包括上电复位、欠压复位和模拟超级看门狗复位（请参考章节 2 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、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 14 所示。

表 14: Strapping 管脚默认上拉/下拉

管脚	GPIO0	GPIO45	GPIO46
默认值	上拉	下拉	下拉

如需改变 Strapping 管脚的值，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-S2 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。

3.2 Boot 控制

复位释放后，GPIO0 和 GPIO46 共同控制 Boot 模式。

表 15: 系统启动模式

管脚	SPI Boot 模式	Download Boot 模式
GPIO0	1	0
GPIO46	x	0

表 15 列出了 GPIO0 和 GPIO46 的 Strapping 值及其对应的系统启动模式。此处"x" 表示该项为无关项。ESP32-S2 芯片当前仅支持 SPI Boot 模式和 Download Boot 模式。GPIO0、GPIO46 组合为 (0, 1) 不可使用。

在 SPI Boot 模式下，CPU 通过从 SPI Flash 中读取程序来启动系统；在 Download Boot 模式下，用户可以通过 UART0、UART1、QPI 或 USB 接口将代码下载到 SRAM 或 Flash 中，或者将程序加载到 SRAM 中并在 Download Boot 模式下执行程序。

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），软件可通过配置 RTC_CNTL_FORCE_DOWNLOAD_BOOT 寄存器，触发 CPU 复位，将芯片启动模式强制从 SPI Boot 模式切换至 Download Boot 模式；如果此 eFuse 设置为 1，则禁用 RTC_CNTL_FORCE_DOWNLOAD_BOOT 寄存器。

- EFUSE_DIS_DOWNLOAD_MODE：如果此 eFuse 设置为 1，则关闭 Download Boot 模式。
- EFUSE_ENABLE_SECURITY_DOWNLOAD：如果此 eFuse 设置为 1，则在 Download Boot 模式下，只允许读取、写入和擦除明文 flash，不支持 SRAM 或寄存器操作。如已禁用 Download Boot 模式，请忽略此 eFuse。

3.3 ROM Code 打印

在系统启动过程中，GPIO46 与 eFuse UART_PRINT_CONTROL 一起控制 ROM Code 打印。

表 16: ROM Code 打印控制

UART_PRINT_CONTROL	GPIO46	ROM Code 打印
0	-	ROM Code 打印至 UART，此时未使用 GPIO46
1	0	使能打印
	1	关闭打印
2	0	关闭打印
	1	使能打印
3	-	系统启动过程中始终关闭打印，此时未使用 GPIO46

ROM Code 上电打印默认通过 U0TXD 管脚，可以由 eFuse UART_PRINT_CHANNEL (0: UART0; 1: DAC_1) 控制切换到 DAC_1 管脚。

3.4 VDD_SPI 电压

芯片复位时，GPIO45 可用于选择 VDD_SPI 电压：

- GPIO45 = 0 时，VDD_SPI 由 VDD3P3_RTC_IO 通过电阻 R_{SPI} 后供电（电压典型值为 3.3 V）；
- GPIO45 = 1 时，VDD_SPI 可选择由内置 LDO 供电（电压为 1.8 V）。

eFuse 中 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：

- VDD_SPI_FORCE = 1 且 VDD_SPI_TIEH = 0 时，VDD_SPI 连接 1.8 V LDO；
- VDD_SPI_FORCE = 1 且 VDD_SPI_TIEH = 1 时，VDD_SPI 连接 VDD3P3_RTC_IO。

4. 中断矩阵

4.1 概述

ESP32-S2 中断矩阵 (Interrupt Matrix) 可以将任一外部中断源单独分配至 CPU 任一外部中断，以便在外设中断信号产生后，及时通知 CPU 进行处理。这一功能灵活强，可以满足不同的应用需求。

4.2 主要特性

- 接收 95 个外部中断源作为输入
- 生成 26 个外部中断作为输出
- 关闭 CPU 的 NMI 类型中断源
- 查询外部中断源当前的中断状态

中断矩阵的结构如图 4-1 所示。

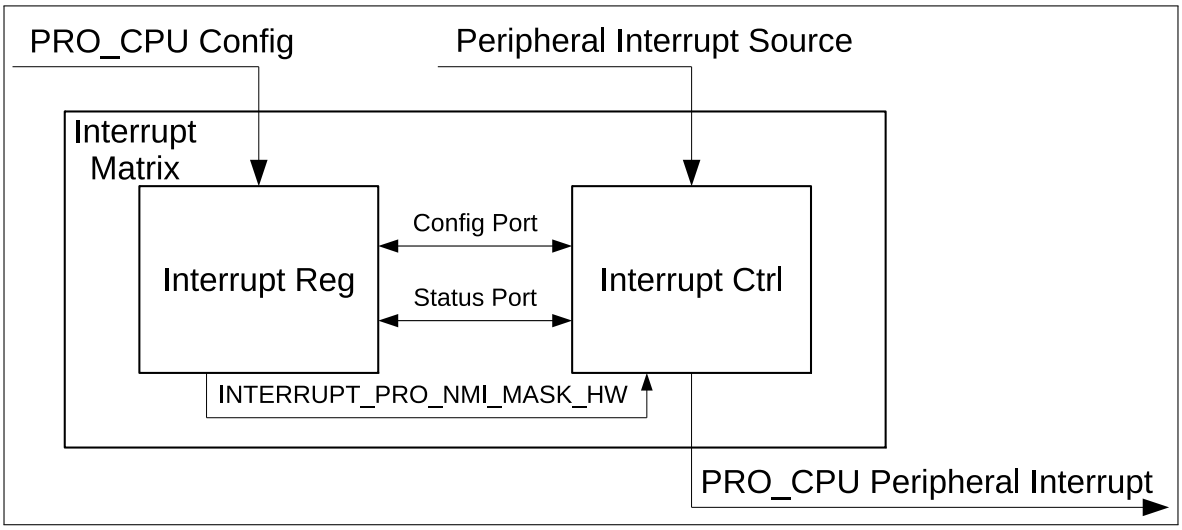


图 4-1. 中断矩阵结构图

4.3 功能描述

4.3.1 外部中断源

ESP32-S2 共有 95 个外部中断源。表 17 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。上述 95 个外部中断源均可分配至 CPU 外部中断。

表 17: 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_PRO_MAC_INTR_MAP_REG	0	INTERRUPT_PRO_INTR_STATUS_REG_0_REG
1	MAC_NMI	INTERRUPT_PRO_MAC_NMI_MAP_REG	1	
2	PWR_INTR	INTERRUPT_PRO_PWR_INTR_MAP_REG	2	
3	BB_INT	INTERRUPT_PRO_BB_INT_MAP_REG	3	
4	BT_MAC_INT	INTERRUPT_PRO_BT_MAC_INT_MAP_REG	4	
5	BT_BB_INT	INTERRUPT_PRO_BT_BB_INT_MAP_REG	5	
6	BT_BB_NMI	INTERRUPT_PRO_BT_BB_NMI_MAP_REG	6	
7	RWBT_IRQ	INTERRUPT_PRO_RWBT_IRQ_MAP_REG	7	
8	RWBLE_IRQ	INTERRUPT_PRO_RWBLE_IRQ_MAP_REG	8	
9	RWBT_NMI	INTERRUPT_PRO_RWBT_NMI_MAP_REG	9	
10	RWBLE_NMI	INTERRUPT_PRO_RWBLE_NMI_MAP_REG	10	
11	SLC0_INTR	INTERRUPT_PRO_SLC0_INTR_MAP_REG	11	
12	SLC1_INTR	INTERRUPT_PRO_SLC1_INTR_MAP_REG	12	
13	UHCI0_INTR	INTERRUPT_PRO_UHCI0_INTR_MAP_REG	13	
14	UHCI1_INTR	INTERRUPT_PRO_UHCI1_INTR_MAP_REG	14	
15	TG_T0_LEVEL_INT	INTERRUPT_PRO_TG_T0_LEVEL_INT_MAP_REG	15	
16	TG_T1_LEVEL_INT	INTERRUPT_PRO_TG_T1_LEVEL_INT_MAP_REG	16	
17	TG_WDT_LEVEL_INT	INTERRUPT_PRO_TG_WDT_LEVEL_INT_MAP_REG	17	
18	TG_LACT_LEVEL_INT	INTERRUPT_PRO_TG_LACT_LEVEL_INT_MAP_REG	18	
19	TG1_T0_LEVEL_INT	INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP_REG	19	
20	TG1_T1_LEVEL_INT	INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP_REG	20	
21	TG1_WDT_LEVEL_INT	INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP_REG	21	
22	TG1_LACT_LEVEL_INT	INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP_REG	22	
23	GPIO_INTERRUPT_PRO	INTERRUPT_PRO_GPIO_INTERRUPT_PRO_MAP_REG	23	
24	GPIO_INTERRUPT_PRO_NMI	INTERRUPT_PRO_GPIO_INTERRUPT_PRO_NMI_MAP_REG	24	
25	GPIO_INTERRUPT_APP	INTERRUPT_PRO_GPIO_INTERRUPT_APP_MAP_REG	25	
26	GPIO_INTERRUPT_APP_NMI	INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP_REG	26	
27	DEDICATED_GPIO_IN_INTR	INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP_REG	27	
28	CPU_INTR_FROM_CPU_0	INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP_REG	28	
29	CPU_INTR_FROM_CPU_1	INTERRUPT_PRO_CPU_INTR_FROM_CPU_1_MAP_REG	29	
30	CPU_INTR_FROM_CPU_2	INTERRUPT_PRO_CPU_INTR_FROM_CPU_2_MAP_REG	30	
31	CPU_INTR_FROM_CPU_3	INTERRUPT_PRO_CPU_INTR_FROM_CPU_3_MAP_REG	31	
32	SPI_INTR_1	INTERRUPT_PRO_SPI_INTR_1_MAP_REG	0	INTERRUPT_PRO_INTR_STATUS_REG_1_REG
33	SPI_INTR_2	INTERRUPT_PRO_SPI_INTR_2_MAP_REG	1	
34	SPI_INTR_3	INTERRUPT_PRO_SPI_INTR_3_MAP_REG	2	

No.	中断源	配置寄存器	状态寄存器	
			位	名称
35	I2S0_INT	INTERRUPT_PRO_I2S0_INT_MAP_REG	3	INTERRUPT_PRO_INTR_STATUS_REG_1_REG
36	I2S1_INT	INTERRUPT_PRO_I2S1_INT_MAP_REG	4	
37	UART_INTR	INTERRUPT_PRO_UART_INTR_MAP_REG	5	
38	UART1_INTR	INTERRUPT_PRO_UART1_INTR_MAP_REG	6	
39	UART2_INTR	INTERRUPT_PRO_UART2_INTR_MAP_REG	7	
40	SDIO_HOST_INTERRUPT	INTERRUPT_PRO_SDIO_HOST_INTERRUPT_MAP_REG	8	
41	PWM0_INTR	INTERRUPT_PRO_PWM0_INTR_MAP_REG	9	
42	PWM1_INTR	INTERRUPT_PRO_PWM1_INTR_MAP_REG	10	
43	PWM2_INTR	INTERRUPT_PRO_PWM2_INTR_MAP_REG	11	
44	PWM3_INTR	INTERRUPT_PRO_PWM3_INTR_MAP_REG	12	
45	LEDC_INT	INTERRUPT_PRO_LEDC_INT_MAP_REG	13	
46	EFUSE_INT	INTERRUPT_PRO_EFUSE_INT_MAP_REG	14	
47	CAN_INT	INTERRUPT_PRO_CAN_INT_MAP_REG	15	
48	USB_INTR	INTERRUPT_PRO_USB_INTR_MAP_REG	16	
49	RTC_CORE_INTR	INTERRUPT_PRO_RTC_CORE_INTR_MAP_REG	17	
50	RMT_INTR	INTERRUPT_PRO_RMT_INTR_MAP_REG	18	
51	PCNT_INTR	INTERRUPT_PRO_PCNT_INTR_MAP_REG	19	
52	I2C_EXT0_INTR	INTERRUPT_PRO_I2C_EXT0_INTR_MAP_REG	20	
53	I2C_EXT1_INTR	INTERRUPT_PRO_I2C_EXT1_INTR_MAP_REG	21	
54	RSA_INTR	INTERRUPT_PRO_RSA_INTR_MAP_REG	22	
55	SHA_INTR	INTERRUPT_PRO_SHA_INTR_MAP_REG	23	
56	AES_INTR	INTERRUPT_PRO_AES_INTR_MAP_REG	24	
57	SPI2_DMA_INT	INTERRUPT_PRO_SPI2_DMA_INT_MAP_REG	25	
58	SPI3_DMA_INT	INTERRUPT_PRO_SPI3_DMA_INT_MAP_REG	26	
59	WDG_INT	INTERRUPT_PRO_WDG_INT_MAP_REG	27	
60	TIMER_INT	INTERRUPT_PRO_TIMER_INT1_MAP_REG	28	
61	TIMER_INT2	INTERRUPT_PRO_TIMER_INT2_MAP_REG	29	
62	TG_T0_EDGE_INT	INTERRUPT_PRO_TG_T0_EDGE_INT_MAP_REG	30	
63	TG_T1_EDGE_INT	INTERRUPT_PRO_TG_T1_EDGE_INT_MAP_REG	31	
64	TG_WDT_EDGE_INT	INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP_REG	0	INTERRUPT_PRO_INTR_STATUS_REG_2_REG
65	TG_LACT_EDGE_INT	INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP_REG	1	
66	TG1_T0_EDGE_INT	INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP_REG	2	
67	TG1_T1_EDGE_INT	INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP_REG	3	
68	TG1_WDT_EDGE_INT	INTERRUPT_PRO_TG1_WDT_EDGE_INT_MAP_REG	4	
69	TG1_LACT_EDGE_INT	INTERRUPT_PRO_TG1_LACT_EDGE_INT_MAP_REG	5	
70	CACHE_IA_INT	INTERRUPT_PRO_CACHE_IA_INT_MAP_REG	6	
71	SYSTIMER_TARGET0_INT	INTERRUPT_PRO_SYSTIMER_TARGET0_INT_MAP_REG	7	

No.	中断源	配置寄存器	状态寄存器	
			位	名称
72	SYSTIMER_TARGET1_INT	INTERRUPT_PRO_SYSTIMER_TARGET1_INT_MAP_REG	8	
73	SYSTIMER_TARGET2_INT	INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP_REG	9	
74	ASSIST_DEBUG_INTR	INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP_REG	10	
75	PMS_PRO_IRAM0_ILG_INTR	INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP_REG	11	
76	PMS_PRO_DRAM0_ILG_INTR	INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP_REG	12	
77	PMS_PRO_DPORT_ILG_INTR	INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP_REG	13	
78	PMS_PRO_AHB_ILG_INTR	INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP_REG	14	
79	PMS_PRO_CACHE_ILG_INTR	INTERRUPT_PRO_PMS_PRO_CACHE_ILG_INTR_MAP_REG	15	
80	PMS_DMA_APB_I_ILG_INTR	INTERRUPT_PRO_PMS_DMA_APB_I_ILG_INTR_MAP_REG	16	
81	PMS_DMA_RX_I_ILG_INTR	INTERRUPT_PRO_PMS_DMA_RX_I_ILG_INTR_MAP_REG	17	
82	PMS_DMA_TX_I_ILG_INTR	INTERRUPT_PRO_PMS_DMA_TX_I_ILG_INTR_MAP_REG	18	
83	SPI_MEM_REJECT_INTR	INTERRUPT_PRO_SPI_MEM_REJECT_INTR_MAP_REG	19	
84	DMA_COPY_INTR	INTERRUPT_PRO_DMA_COPY_INTR_MAP_REG	20	
85	SPI4_DMA_INT	INTERRUPT_PRO_SPI4_DMA_INT_MAP_REG	21	
86	DCACHE_PRELOAD_INT	INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP_REG	22	
87	DCACHE_PRELOAD_INT	INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP_REG	23	INTERRUPT_PRO_INTR_STATUS_REG_2_REG
88	ICACHE_PRELOAD_INT	INTERRUPT_PRO_ICACHE_PRELOAD_INT_MAP_REG	24	
89	APB_ADC_INT	INTERRUPT_PRO_APB_ADC_INT_MAP_REG	25	
90	CRYPTO_DMA_INT	INTERRUPT_PRO_CRYPTO_DMA_INT_MAP_REG	26	
91	CPU_PERI_ERROR_INT	INTERRUPT_PRO_CPU_PERI_ERROR_INT_MAP_REG	27	
92	APB_PERI_ERROR_INT	INTERRUPT_PRO_APB_PERI_ERROR_INT_MAP_REG	28	
93	DCACHE_SYNC_INT	INTERRUPT_PRO_DCACHE_SYNC_INT_MAP_REG	29	
94	ICACHE_SYNC_INT	INTERRUPT_PRO_ICACHE_SYNC_INT_MAP_REG	30	

4.3.2 CPU 中断

CPU 共有 32 个中断，其中包括 26 个外部中断和六个内部中断。表 18 列出了所有中断：

- 外部中断
  - 电平触发型中断：由高电平触发，要求保持中断的电平状态直到 CPU 响应。
  - 边沿触发型中断：由上升沿触发，此中断一旦发生，CPU 即可响应。
  - NMI 类型中断：软件不可使用 CPU 寄存器屏蔽此类中断。
- 内部中断
  - 定时器类型中断：由内部定时器触发，可用于产生周期性中断。
  - 软件类型中断：软件写特殊寄存器时将触发此中断。
  - 解析类型中断：用于性能监测与分析。

ESP32-S2 支持 6 级中断，在下表优先级一栏中，数字越大代表中断优先级越高。其中，NMI 拥有最高优先级，一旦 NMI 中断发生，CPU 必定会响应。

表 18: CPU 中断

编号	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
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

编号	类别	种类	优先级
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

4.3.3 分配外部中断源至 CPU 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_X：用于表示某个外部中断源，其中 X 为中断源编号，详见表 17。
- INTERRUPT_PRO_X_MAP_REG：用于表示 CPU 的某个外部中断配置寄存器。此外部中断配置寄存器与外部中断源 Source_X 相对应。即表 17 中“配置寄存器”一栏与“中断源”一栏应一一对应。例如：MAC_INTR 的中断配置寄存器为处于同一行的 INTERRUPT_PRO_MAC_INTR_MAP_REG。
- Interrupt_P：表示 CPU 中断序号为 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 18。
- Interrupt_I：表示 CPU 中断序号为 Num_I 的内部中断，Num_I 的取值范围为 6、7、11、15、16、29，详见表 18。

4.3.3.1 分配一个外部中断源 Source_X 至 CPU 外部中断

将外部中断源 Source_X 对应的寄存器 INTERRUPT_PRO_X_MAP_REG 配置成 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 可以取任一 CPU 外部中断值，包括 0 ~ 5、8 ~ 10、12 ~ 14、17 ~ 28、30 ~ 31。一个 CPU 外部中断可以被多个外设共享。

4.3.3.2 分配多个外部中断源 Source_X*n* 至 CPU 外部中断

将各个中断源对应的寄存器 INTERRUPT_PRO_X*n*_MAP_REG 均配置成相同的 Num_P，即可将多个中断源 Source_X*n* 分配至同一 CPU 外部中断 Interrupt_P。上述任一外设中断源均会触发 CPU 外部中断 Interrupt_P。待中断触发后，软件需要查询中断状态寄存器，用于判断哪个外设产生中断。

4.3.3.3 关闭 CPU 外部中断源 Source_X

将中断源对应的寄存器 INTERRUPT_PRO_X_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，而且选择任一内部中断值 (6、7、11、15、16、29) 不会造成其他影响，可用于关闭外部中断。

4.3.4 关闭 CPU 的 NMI 类型中断源

中断矩阵内部的 Interrupt Reg 寄存器配置子模块可以产生 INTERRUPT_PRO_NMI_MASK_HW 信号，中断矩阵根据此内部信号可以通过硬件关闭所有被分配到 CPU 第 14 号 NMI 中断的外部中断源。信号 INTERRUPT_PRO_NMI_MASK_HW 可以配置为高电平，此时 CPU 不响应 NMI 中断，也可以配置为低电平，则 CPU 响应 NMI 中断。具体结构请参考图 4-1 所示。

4.3.5 查询外部中断源当前的中断状态

读取寄存器 INTERRUPT_PRO_INTR_STATUS_REG_*n*（只读）中的特定 Bit 值可以获取外部中断源当前的中断状态。寄存器 INTERRUPT_PRO_INTR_STATUS_REG_*n* 与外部中断源的对应关系如表 17 所示。

4.4 基地址

用户可以通过寄存器基地址访问中断矩阵，如表 19 所示。更多信息，请访问章节 1 *系统和存储器*。

表 19: 中断矩阵基地址

访问总线	基地址
PeriBUS1	0x3F4C2000

4.5 寄存器列表

请注意，下表中的地址都是相对于中断矩阵基地址的地址偏移量（相对地址）。更多有关中断矩阵基地址的信息，请前往第 4.4 节。

名称	描述	地址	访问
配置寄存器			
INTERRUPT_PRO_MAC_INTR_MAP_REG	MAC_INTR 中断配置寄存器	0x0000	读/写
INTERRUPT_PRO_MAC_NMI_MAP_REG	MAC_NMI 中断配置寄存器	0x0004	读/写
INTERRUPT_PRO_PWR_INTR_MAP_REG	PWR_INTR 中断配置寄存器	0x0008	读/写
INTERRUPT_PRO_BB_INT_MAP_REG	BB_INT 中断配置寄存器	0x000C	读/写
INTERRUPT_PRO_BT_MAC_INT_MAP_REG	BT_MAC 中断配置寄存器	0x0010	读/写
INTERRUPT_PRO_BT_BB_INT_MAP_REG	BT_BB_INT 中断配置寄存器	0x0014	读/写
INTERRUPT_PRO_BT_BB_NMI_MAP_REG	BT_BB_NMI 中断配置寄存器	0x0018	读/写
INTERRUPT_PRO_RWBT_IRQ_MAP_REG	RWBT_IRQ 中断配置寄存器	0x001C	读/写
INTERRUPT_PRO_RWBLE_IRQ_MAP_REG	RWBLE_IRQ 中断配置寄存器	0x0020	读/写
INTERRUPT_PRO_RWBT_NMI_MAP_REG	RWBT_NMI 中断配置寄存器	0x0024	读/写
INTERRUPT_PRO_RWBLE_NMI_MAP_REG	RWBLE_NMI 中断配置寄存器	0x0028	读/写
INTERRUPT_PRO_SLC0_INTR_MAP_REG	SLC0_INTR 中断配置寄存器	0x002C	读/写
INTERRUPT_PRO_SLC1_INTR_MAP_REG	SLC1_INTR 中断配置寄存器	0x0030	读/写
INTERRUPT_PRO_UHCI0_INTR_MAP_REG	UHCI0_INTR 中断配置寄存器	0x0034	读/写
INTERRUPT_PRO_UHCI1_INTR_MAP_REG	UHCI1_INTR 中断配置寄存器	0x0038	读/写
INTERRUPT_PRO_TG_T0_LEVEL_INT_MAP_REG	TG_T0_LEVEL_INT 中断配置寄存器	0x003C	读/写
INTERRUPT_PRO_TG_T1_LEVEL_INT_MAP_REG	TG_T1_LEVEL_INT 中断配置寄存器	0x0040	读/写
INTERRUPT_PRO_TG_WDT_LEVEL_INT_MAP_REG	TG_WDT_LEVEL_INT 中断配置寄存器	0x0044	读/写
INTERRUPT_PRO_TG_LACT_LEVEL_INT_MAP_REG	TG_LACT_LEVEL_INT 中断配置寄存器	0x0048	读/写
INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP_REG	TG1_T0_LEVEL_INT 中断配置寄存器	0x004C	读/写
INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP_REG	TG1_T1_LEVEL_INT 中断配置寄存器	0x0050	读/写
INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP_REG	TG1_WDT_LEVEL_INT 中断配置寄存器	0x0054	读/写
INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP_REG	TG1_LACT_LEVEL_INT 中断配置寄存器	0x0058	读/写
INTERRUPT_PRO_GPIO_INTERRUPT_PRO_MAP_REG	GPIO_INTERRUPT_PRO 中断配置寄存器	0x005C	读/写
INTERRUPT_PRO_GPIO_INTERRUPT_PRO_NMI_MAP_REG	GPIO_INTERRUPT_PRO_NMI 中断配置寄存器	0x0060	读/写
INTERRUPT_PRO_GPIO_INTERRUPT_APP_MAP_REG	GPIO_INTERRUPT_APP 中断配置寄存器	0x0064	读/写
INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP_REG	GPIO_INTERRUPT_APP_NMI 中断配置寄存器	0x0068	读/写
INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP_REG	DEDICATED_GPIO_IN_INTR 中断配置寄存器	0x006C	读/写

名称	描述	地址	访问
INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP_REG	CPU_INTR_FROM_CPU_0 中断配置寄存器	0x0070	读/写
INTERRUPT_PRO_CPU_INTR_FROM_CPU_1_MAP_REG	CPU_INTR_FROM_CPU_1 中断配置寄存器	0x0074	读/写
INTERRUPT_PRO_CPU_INTR_FROM_CPU_2_MAP_REG	CPU_INTR_FROM_CPU_2 中断配置寄存器	0x0078	读/写
INTERRUPT_PRO_CPU_INTR_FROM_CPU_3_MAP_REG	CPU_INTR_FROM_CPU_3 中断配置寄存器	0x007C	读/写
INTERRUPT_PRO_SPI_INTR_1_MAP_REG	SPI_INTR_1 中断配置寄存器	0x0080	读/写
INTERRUPT_PRO_SPI_INTR_2_MAP_REG	SPI_INTR_2 中断配置寄存器	0x0084	读/写
INTERRUPT_PRO_SPI_INTR_3_MAP_REG	SPI_INTR_3 中断配置寄存器	0x0088	读/写
INTERRUPT_PRO_I2S0_INT_MAP_REG	I2S0_INT 中断配置寄存器	0x008C	读/写
INTERRUPT_PRO_I2S1_INT_MAP_REG	I2S1_INT 中断配置寄存器	0x0090	读/写
INTERRUPT_PRO_UART_INTR_MAP_REG	UART_INTR 中断配置寄存器	0x0094	读/写
INTERRUPT_PRO_UART1_INTR_MAP_REG	UART1_INTR 中断配置寄存器	0x0098	读/写
INTERRUPT_PRO_UART2_INTR_MAP_REG	UART2_INTR 中断配置寄存器	0x009C	读/写
INTERRUPT_PRO_SDIO_HOST_INTERRUPT_MAP_REG	SDIO_HOST_INTERRUPT 中断配置寄存器	0x00A0	读/写
INTERRUPT_PRO_PWM0_INTR_MAP_REG	PWM0_INTR 中断配置寄存器	0x00A4	读/写
INTERRUPT_PRO_PWM1_INTR_MAP_REG	PWM1_INTR 中断配置寄存器	0x00A8	读/写
INTERRUPT_PRO_PWM2_INTR_MAP_REG	PWM2_INTR 中断配置寄存器	0x00AC	读/写
INTERRUPT_PRO_PWM3_INTR_MAP_REG	PWM3_INTR 中断配置寄存器	0x00B0	读/写
INTERRUPT_PRO_LEDC_INT_MAP_REG	LEDC_INT 中断配置寄存器	0x00B4	读/写
INTERRUPT_PRO_EFUSE_INT_MAP_REG	EFUSE_INT 中断配置寄存器	0x00B8	读/写
INTERRUPT_PRO_CAN_INT_MAP_REG	CAN_INT 中断配置寄存器	0x00BC	读/写
INTERRUPT_PRO_USB_INTR_MAP_REG	USB_INTR 中断配置寄存器	0x00C0	读/写
INTERRUPT_PRO_RTC_CORE_INTR_MAP_REG	RTC_CORE_INTR 中断配置寄存器	0x00C4	读/写
INTERRUPT_PRO_RMT_INTR_MAP_REG	RMT_INTR 中断配置寄存器	0x00C8	读/写
INTERRUPT_PRO_PCNT_INTR_MAP_REG	PCNT_INTR 中断配置寄存器	0x00CC	读/写
INTERRUPT_PRO_I2C_EXT0_INTR_MAP_REG	I2C_EXT0_INTR 中断配置寄存器	0x00D0	读/写
INTERRUPT_PRO_I2C_EXT1_INTR_MAP_REG	I2C_EXT1_INTR 中断配置寄存器	0x00D4	读/写
INTERRUPT_PRO_RSA_INTR_MAP_REG	RSA_INTR 中断配置寄存器	0x00D8	读/写
INTERRUPT_PRO_SHA_INTR_MAP_REG	SHA_INTR 中断配置寄存器	0x00DC	读/写
INTERRUPT_PRO_AES_INTR_MAP_REG	AES_INTR 中断配置寄存器	0x00E0	读/写

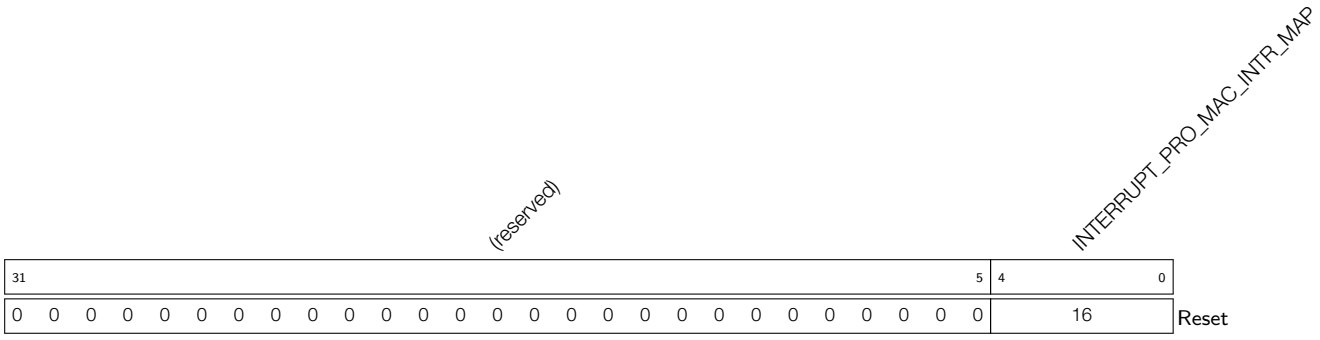
名称	描述	地址	访问
INTERRUPT_PRO_SPI2_DMA_INT_MAP_REG	SPI2_DMA_INT 中断配置寄存器	0x00E4	读/写
INTERRUPT_PRO_SPI3_DMA_INT_MAP_REG	SPI3_DMA_INT 中断配置寄存器	0x00E8	读/写
INTERRUPT_PRO_WDG_INT_MAP_REG	WDG_INT 中断配置寄存器	0x00EC	读/写
INTERRUPT_PRO_TIMER_INT1_MAP_REG	TIMER_INT1 中断配置寄存器	0x00F0	读/写
INTERRUPT_PRO_TIMER_INT2_MAP_REG	TIMER_INT2 中断配置寄存器	0x00F4	读/写
INTERRUPT_PRO_TG_T0_EDGE_INT_MAP_REG	TG_T0_EDGE_INT 中断配置寄存器	0x00F8	读/写
INTERRUPT_PRO_TG_T1_EDGE_INT_MAP_REG	TG_T1_EDGE_INT 中断配置寄存器	0x00FC	读/写
INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP_REG	TG_WDT_EDGE_INT 中断配置寄存器	0x0100	读/写
INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP_REG	TG_LACT_EDGE_INT 中断配置寄存器	0x0104	读/写
INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP_REG	TG1_T0_EDGE_INT 中断配置寄存器	0x0108	读/写
INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP_REG	TG1_T1_EDGE_INT 中断配置寄存器	0x010C	读/写
INTERRUPT_PRO_TG1_WDT_EDGE_INT_MAP_REG	TG1_WDT_EDGE_INT 中断配置寄存器	0x0110	读/写
INTERRUPT_PRO_TG1_LACT_EDGE_INT_MAP_REG	TG1_LACT_EDGE_INT 中断配置寄存器	0x0114	读/写
INTERRUPT_PRO_CACHE_IA_INT_MAP_REG	CACHE_IA_INT 中断配置寄存器	0x0118	读/写
INTERRUPT_PRO_SYSTIMER_TARGET0_INT_MAP_REG	SYSTIMER_TARGET0_INT 中断配置寄存器	0x011C	读/写
INTERRUPT_PRO_SYSTIMER_TARGET1_INT_MAP_REG	SYSTIMER_TARGET1 中断配置寄存器	0x0120	读/写
INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP_REG	SYSTIMER_TARGET2 中断配置寄存器	0x0124	读/写
INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP_REG	ASSIST_DEBUG_INTR 中断配置寄存器	0x0128	读/写
INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP_REG	PMS_PRO_IRAM0_ILG_INTR 中断配置寄存器	0x012C	读/写
INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP_REG	PMS_PRO_DRAM0_ILG_INTR 中断配置寄存器	0x0130	读/写
INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP_REG	PMS_PRO_DPORT_ILG_INTR 中断配置寄存器	0x0134	读/写
INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP_REG	PMS_PRO_AHB_ILG_INTR 中断配置寄存器	0x0138	读/写
INTERRUPT_PRO_PMS_PRO_CACHE_ILG_INTR_MAP_REG	PMS_PRO_CACHE_ILG_INTR 中断配置寄存器	0x013C	读/写
INTERRUPT_PRO_PMS_DMA_APB_I_ILG_INTR_MAP_REG	PMS_DMA_APB_I_ILG_INTR 中断配置寄存器	0x0140	读/写
INTERRUPT_PRO_PMS_DMA_RX_I_ILG_INTR_MAP_REG	PMS_DMA_RX_I_ILG_INTR 中断配置寄存器	0x0144	读/写
INTERRUPT_PRO_PMS_DMA_TX_I_ILG_INTR_MAP_REG	PMS_DMA_TX_I_ILG_INTR 中断配置寄存器	0x0148	读/写
INTERRUPT_PRO_SPI_MEM_REJECT_INTR_MAP_REG	SPI_MEM_REJECT_INTR 中断配置寄存器	0x014C	读/写
INTERRUPT_PRO_DMA_COPY_INTR_MAP_REG	DMA_COPY_INTR 中断配置寄存器	0x0150	读/写
INTERRUPT_PRO_SPI4_DMA_INT_MAP_REG	SPI4_DMA_INT 中断配置寄存器	0x0154	读/写

名称	描述	地址	访问
INTERRUPT_PRO_SPI_INTR_4_MAP_REG	SPI_INTR_4 中断配置寄存器	0x0158	读/写
INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP_REG	DCACHE_PRELOAD_INT 中断配置寄存器	0x015C	读/写
INTERRUPT_PRO_ICACHE_PRELOAD_INT_MAP_REG	ICACHE_PRELOAD_INT 中断配置寄存器	0x0160	读/写
INTERRUPT_PRO_APB_ADC_INT_MAP_REG	APB_ADC_INT 中断配置寄存器	0x0164	读/写
INTERRUPT_PRO_CRYPTO_DMA_INT_MAP_REG	CRYPTO_DMA_INT 中断配置寄存器	0x0168	读/写
INTERRUPT_PRO_CPU_PERI_ERROR_INT_MAP_REG	cpu peri error 中断配置寄存器	0x016C	读/写
INTERRUPT_PRO_APB_PERI_ERROR_INT_MAP_REG	CPU_PERI_ERROR_INT 中断配置寄存器	0x0170	读/写
INTERRUPT_PRO_DCACHE_SYNC_INT_MAP_REG	DCACHE_SYNC_INT 中断配置寄存器	0x0174	读/写
INTERRUPT_PRO_ICACHE_SYNC_INT_MAP_REG	ICACHE_SYNC_INT 中断配置寄存器	0x0178	读/写
INTERRUPT_CLOCK_GATE_REG	NMI 中断信号屏蔽寄存器	0x0188	读/写
中断状态寄存器			
INTERRUPT_PRO_INTR_STATUS_REG_0_REG	中断状态寄存器 0	0x017C	只读
INTERRUPT_PRO_INTR_STATUS_REG_1_REG	中断状态寄存器 1	0x0180	只读
INTERRUPT_PRO_INTR_STATUS_REG_2_REG	中断状态寄存器 2	0x0184	只读
版本寄存器			
INTERRUPT_REG_DATE_REG	版本控制寄存器	0x0FFC	读/写

4.6 寄存器

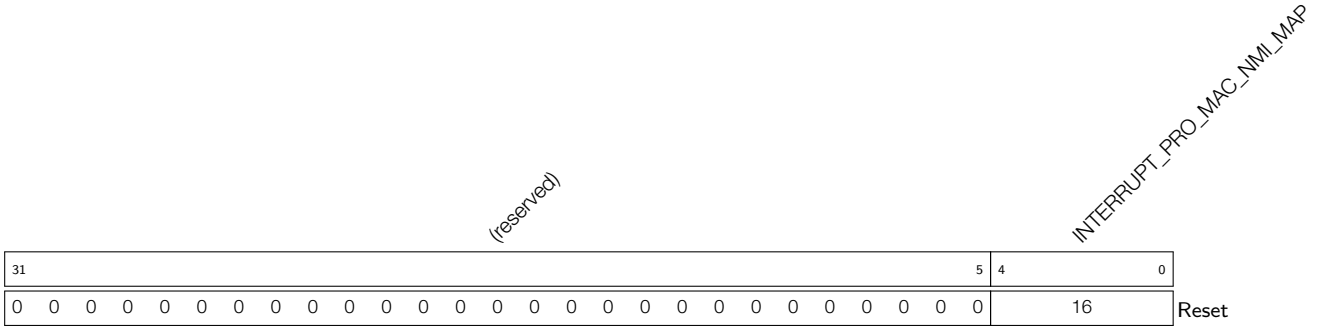
请注意，下表中的地址都是相对于中断矩阵基地址的地址偏移量（相对地址）。更多有关中断矩阵基地址的信息，请前往第 4.4 节。

Register 4.1: INTERRUPT_PRO_MAC_INTR_MAP_REG (0x0000)



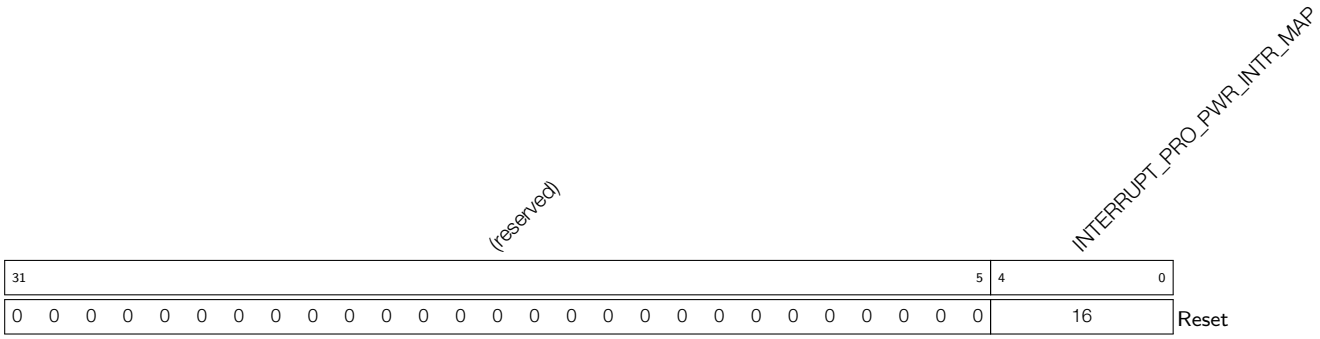
INTERRUPT_PRO_MAC_INTR_MAP 用于将 MAC_INTR 中断信号映射至 CPU 中断。（读/写）

Register 4.2: INTERRUPT_PRO_MAC_NMI_MAP_REG (0x0004)



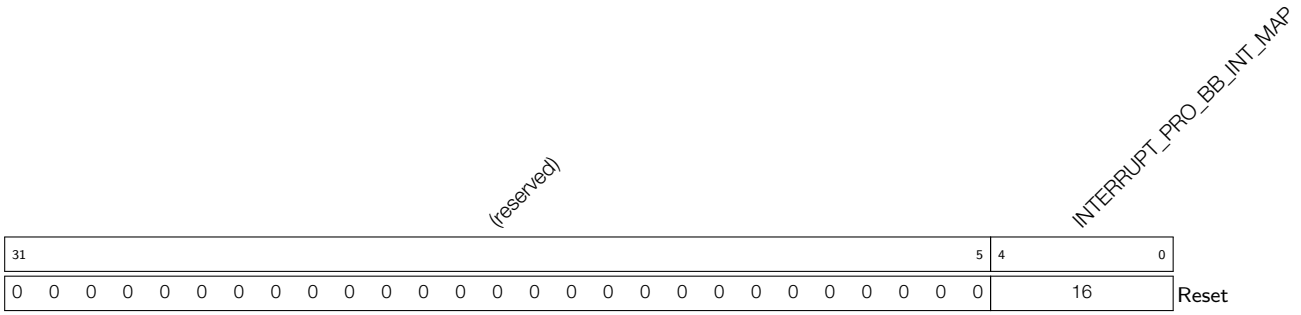
INTERRUPT_PRO_MAC_NMI_MAP 用于将 MAC_NMI 中断信号映射至 CPU 中断。（读/写）

Register 4.3: INTERRUPT_PRO_PWR_INTR_MAP_REG (0x0008)



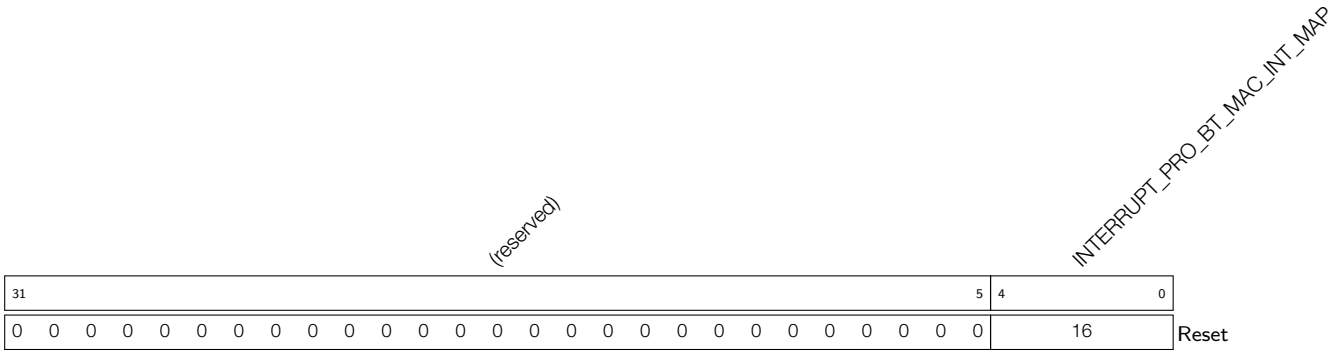
INTERRUPT_PRO_PWR_INTR_MAP 用于将 PWR_INTR 中断信号映射至 CPU 中断。（读/写）

Register 4.4: INTERRUPT_PRO_BB_INT_MAP_REG (0x000C)



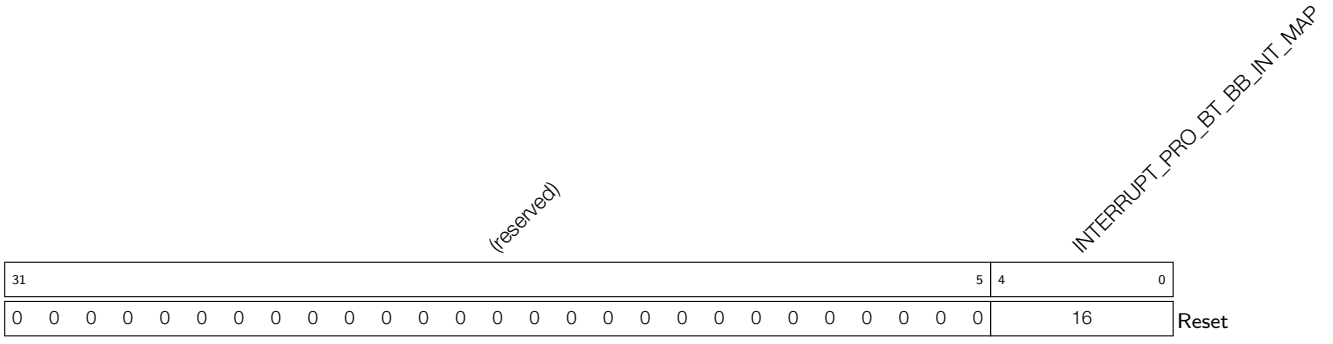
INTERRUPT_PRO_BB_INT_MAP 用于将 BB_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.5: INTERRUPT_PRO_BT_MAC_INT_MAP_REG (0x0010)



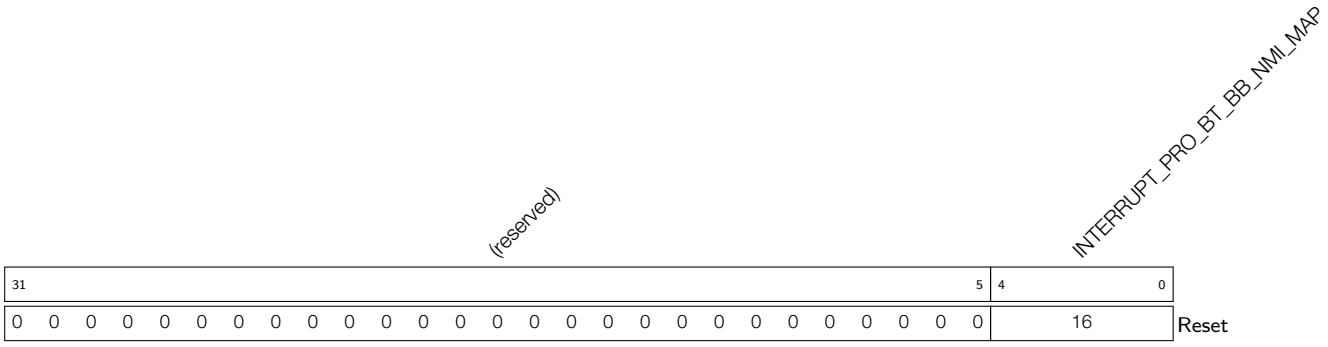
INTERRUPT_PRO_BT_MAC_INT_MAP 用于将 BT_MAC_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.6: INTERRUPT_PRO_BT_BB_INT_MAP_REG (0x0014)



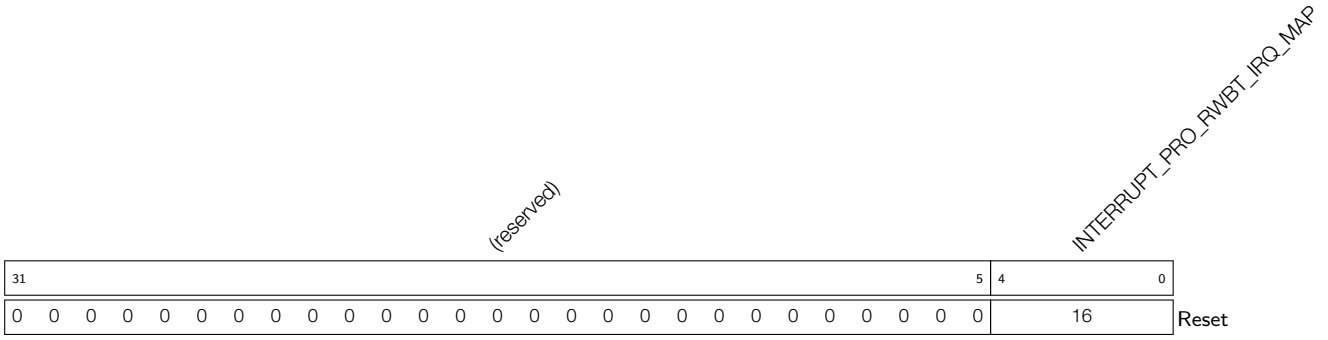
INTERRUPT_PRO_BT_BB_INT_MAP 用于将 BT_BB_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.7: INTERRUPT_PRO_BT_BB_NMI_MAP_REG (0x0018)



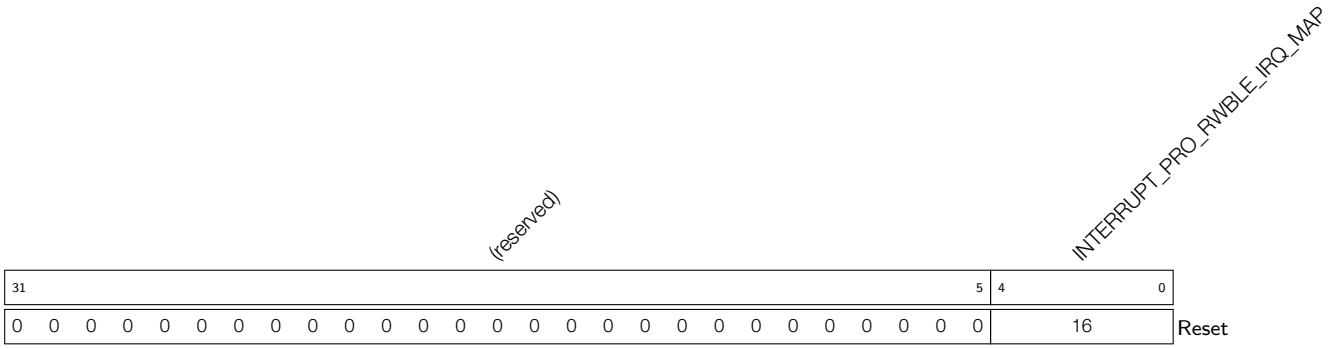
INTERRUPT_PRO_BT_BB_NMI_MAP 用于将 BT_BB_NMI 中断信号映射至 CPU 中断。(读/写)

Register 4.8: INTERRUPT_PRO_RWBT_IRQ_MAP_REG (0x001C)



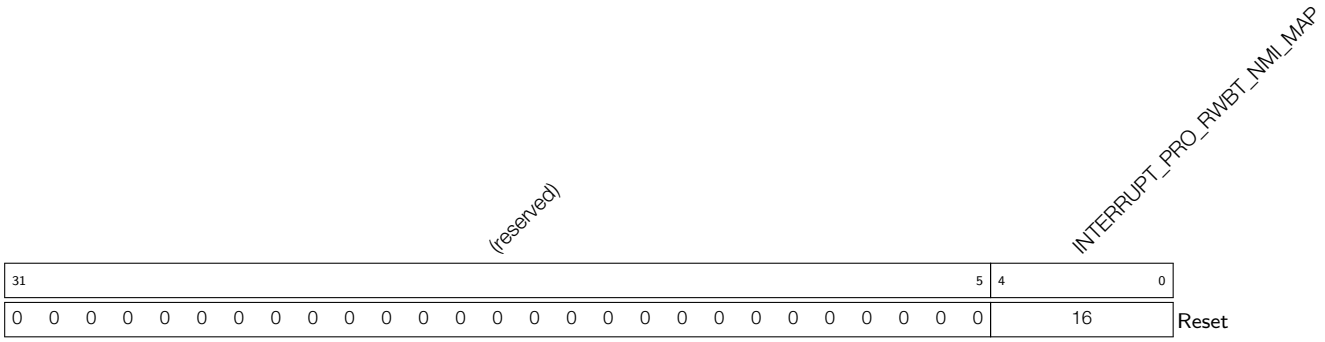
INTERRUPT_PRO_RWBT_IRQ_MAP 用于将 RWBT_IRQ 中断信号映射至 CPU 中断。(读/写)

Register 4.9: INTERRUPT_PRO_RWBLE_IRQ_MAP_REG (0x0020)



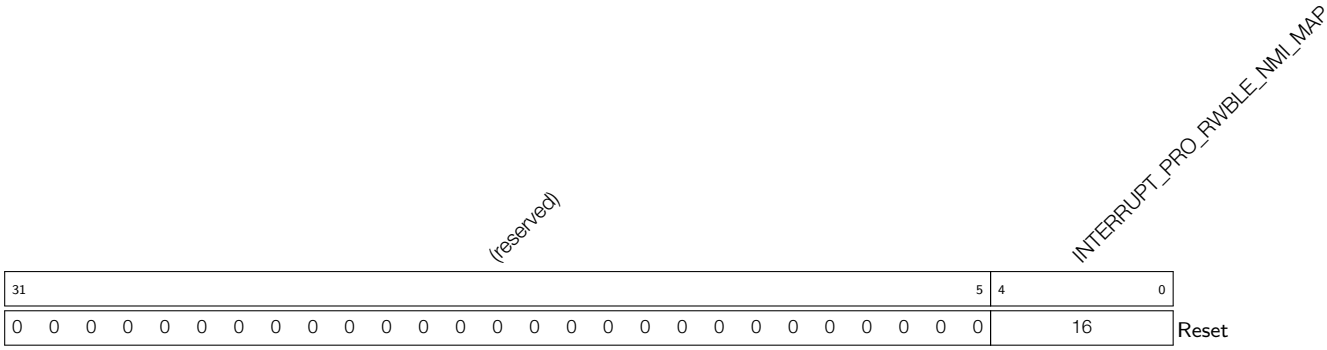
INTERRUPT_PRO_RWBLE_IRQ_MAP 用于将 RWBLE_IRQ 中断信号映射至 CPU 中断。(读/写)

Register 4.10: INTERRUPT_PRO_RWBT_NMI_MAP_REG (0x0024)



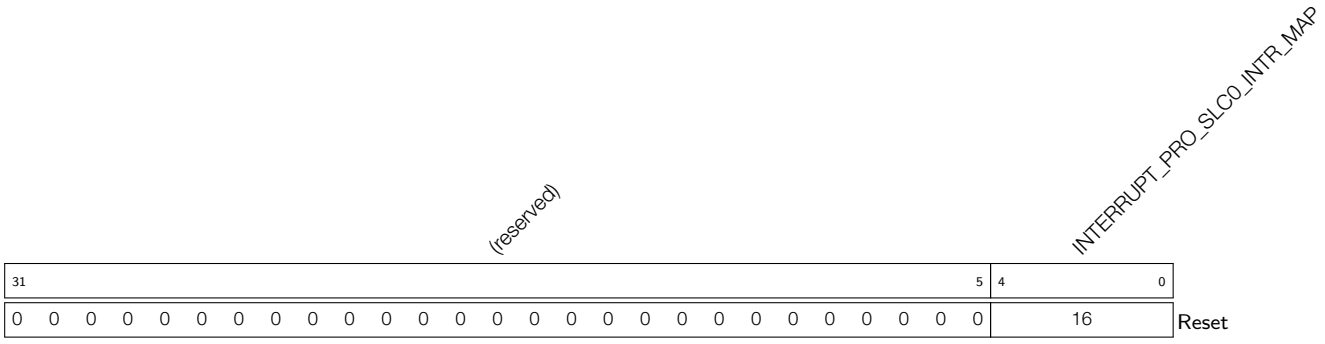
INTERRUPT_PRO_RWBT_NMI_MAP 用于将 RWBT_NMI 中断信号映射至 CPU 中断。(读/写)

Register 4.11: INTERRUPT_PRO_RWBLE_NMI_MAP_REG (0x0028)



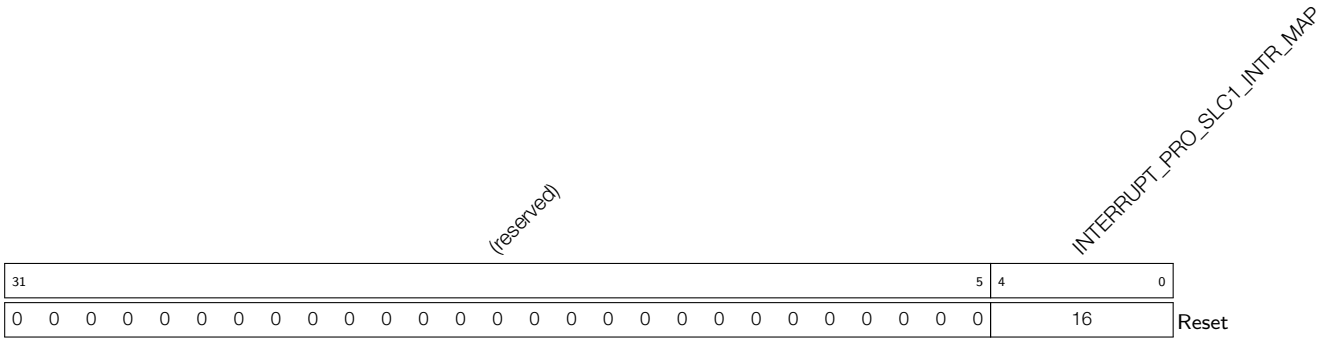
INTERRUPT_PRO_RWBLE_NMI_MAP 用于将 RWBLE_NMI 中断信号映射至 CPU 中断。(读/写)

Register 4.12: INTERRUPT_PRO_SLC0_INTR_MAP_REG (0x002C)



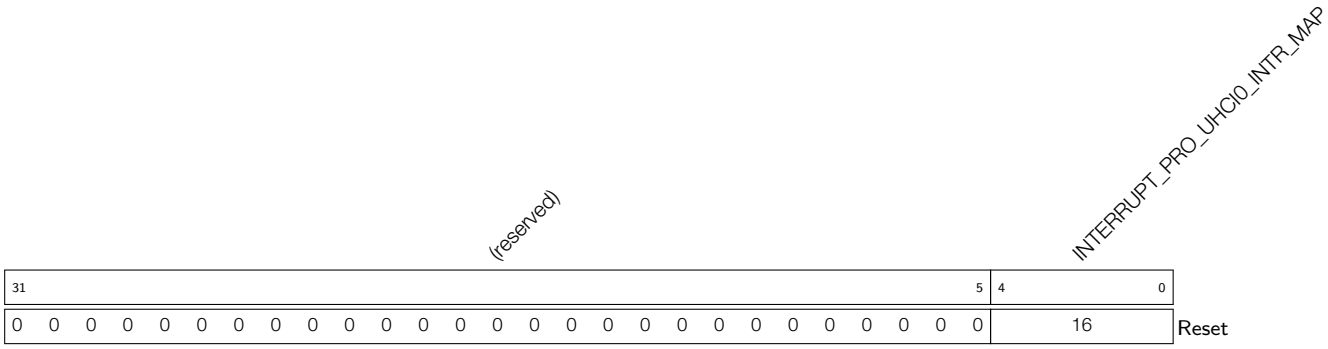
INTERRUPT_PRO_SLC0_INTR_MAP 用于将 SLC0_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.13: INTERRUPT_PRO_SLC1_INTR_MAP_REG (0x0030)



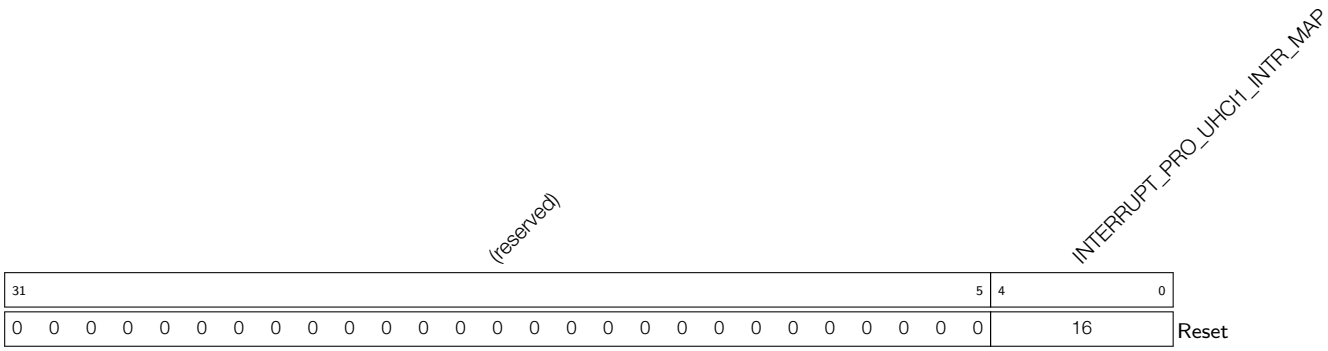
INTERRUPT_PRO_SLC1_INTR_MAP 用于将 SLC1_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.14: INTERRUPT_PRO_UHCI0_INTR_MAP_REG (0x0034)



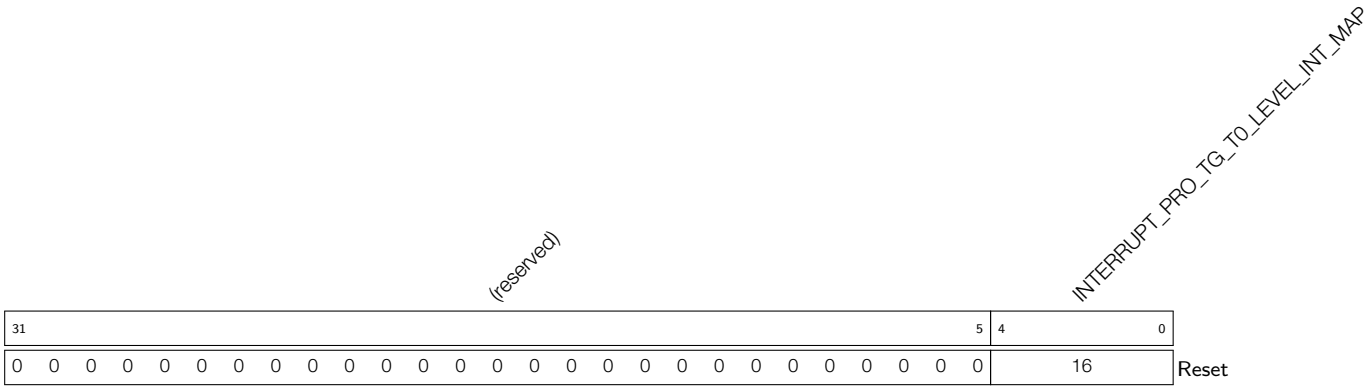
INTERRUPT_PRO_UHCI0_INTR_MAP 用于将 UHCI0_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.15: INTERRUPT_PRO_UHCI1_INTR_MAP_REG (0x0038)



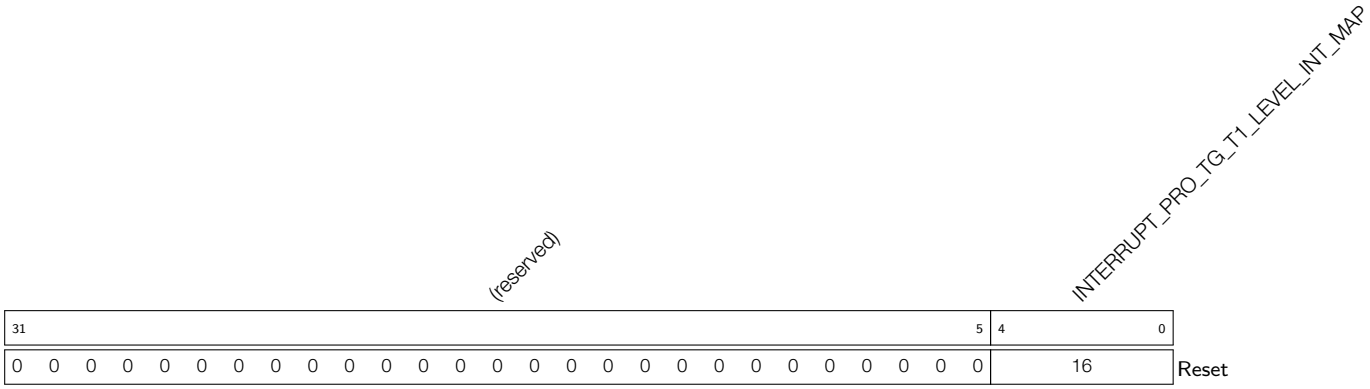
INTERRUPT_PRO_UHCI1_INTR_MAP 用于将 UHCI1_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.16: INTERRUPT_PRO_TG_T0_LEVEL_INT_MAP_REG (0x003C)



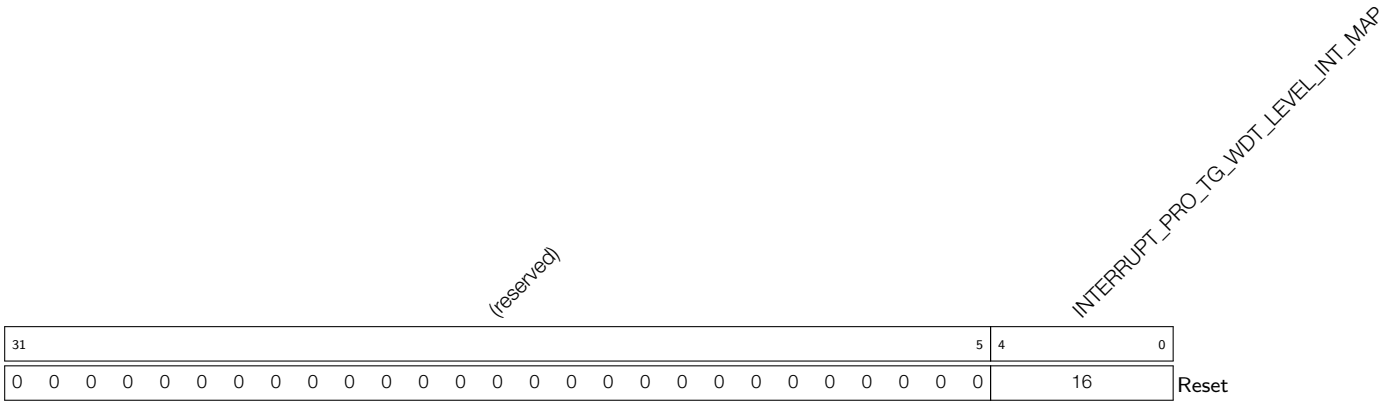
INTERRUPT_PRO_TG_T0_LEVEL_INT_MAP 用于将 TG_T0_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.17: INTERRUPT_PRO_TG_T1_LEVEL_INT_MAP_REG (0x0040)



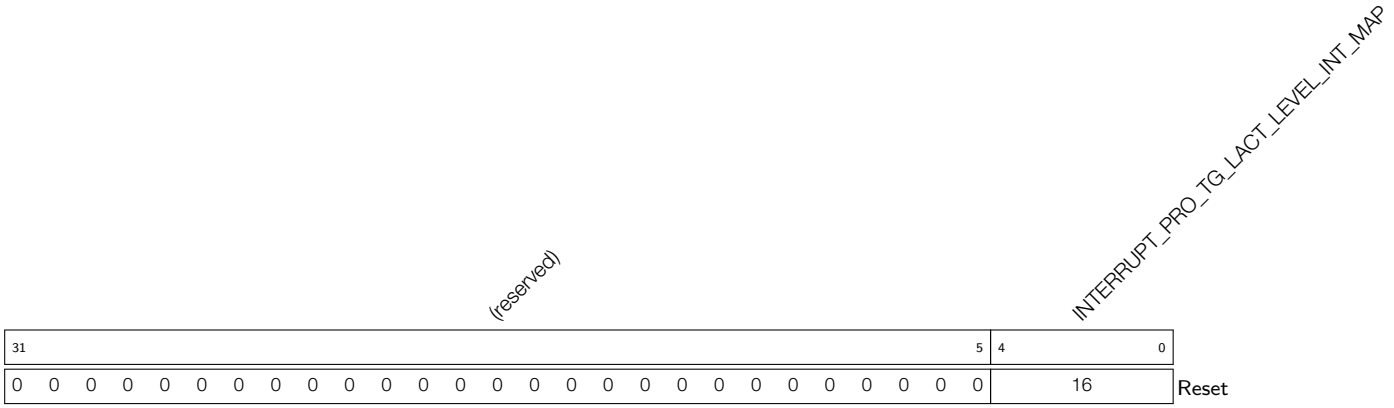
INTERRUPT_PRO_TG_T1_LEVEL_INT_MAP 用于将 TG_T1_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.18: INTERRUPT_PRO_TG_WDT_LEVEL_INT_MAP_REG (0x0044)



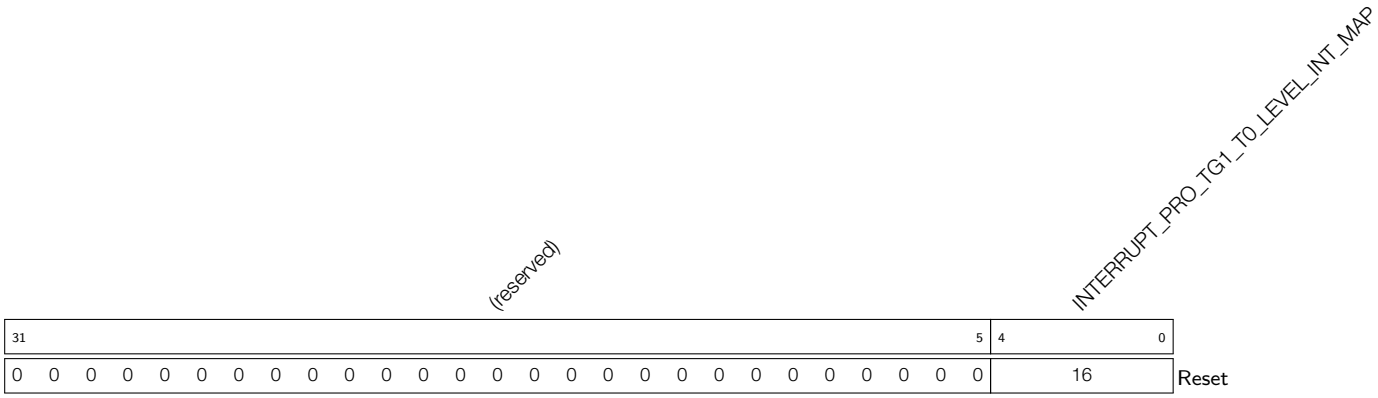
INTERRUPT_PRO_TG_WDT_LEVEL_INT_MAP 用于将 TG_WDT_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.19: INTERRUPT_PRO_TG_LACT_LEVEL_INT_MAP_REG (0x0048)



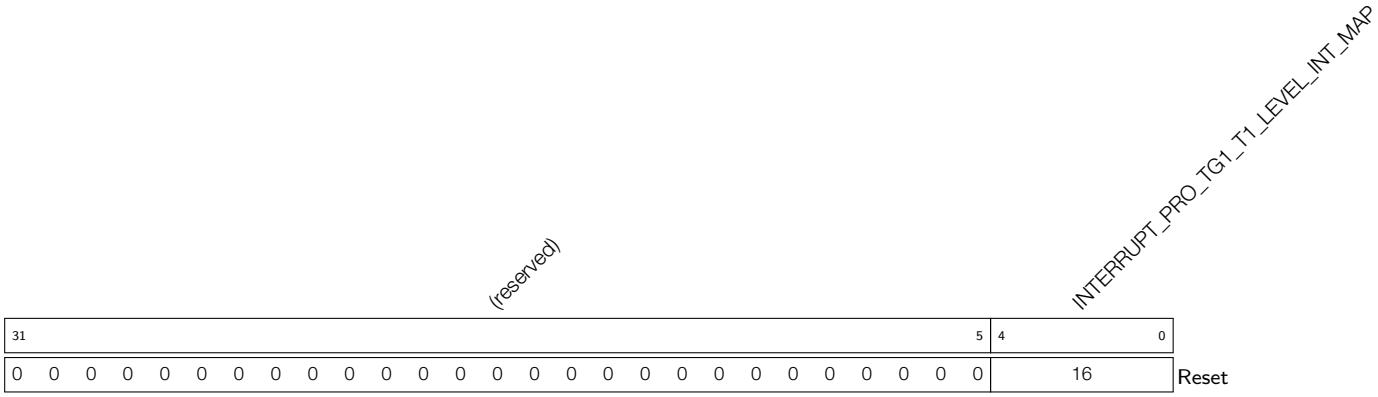
INTERRUPT_PRO_TG_LACT_LEVEL_INT_MAP 用于将 TG_LACT_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.20: INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP_REG (0x004C)



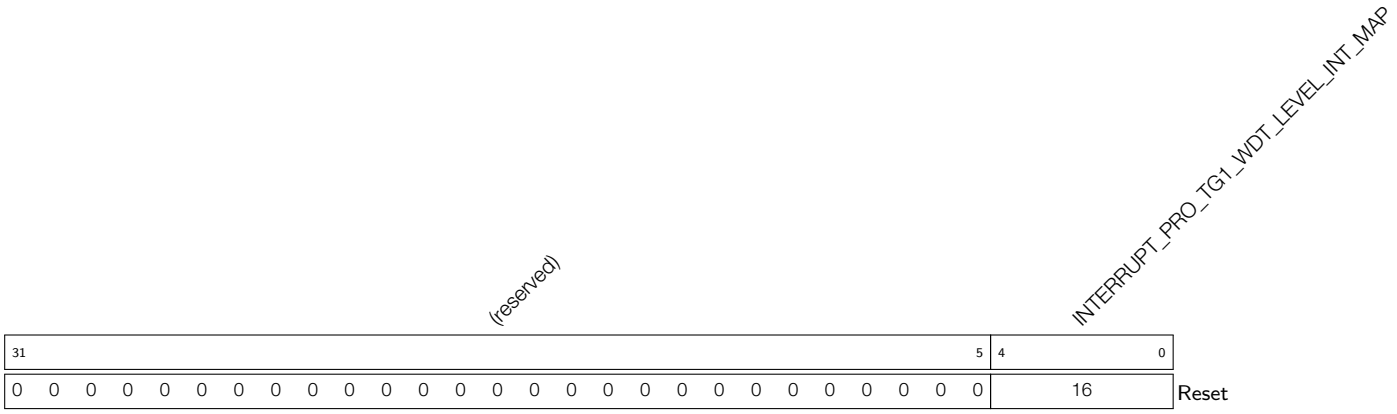
INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP 用于将 TG1_T0_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.21: INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP_REG (0x0050)



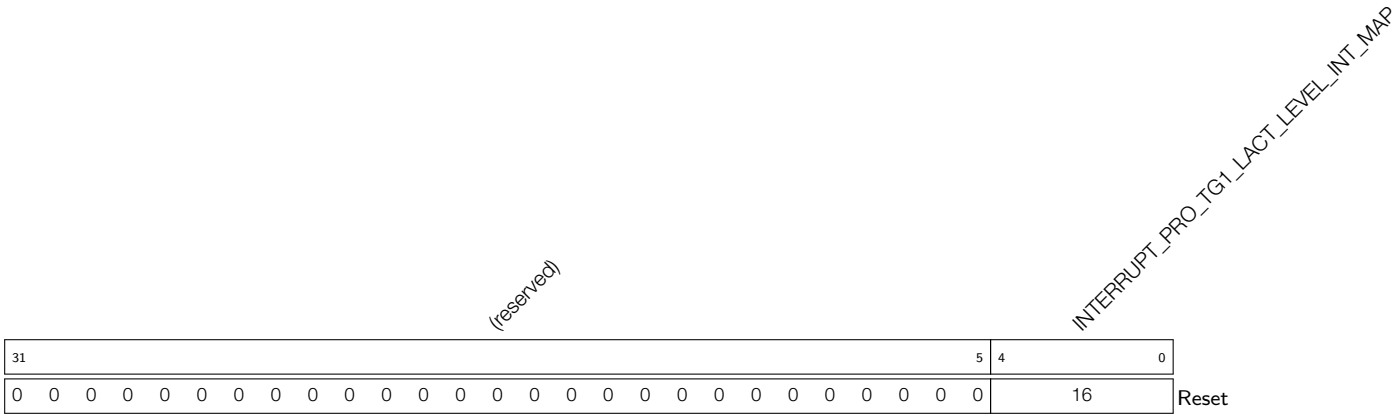
INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP 用于将 TG1_T1_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.22: INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP_REG (0x0054)



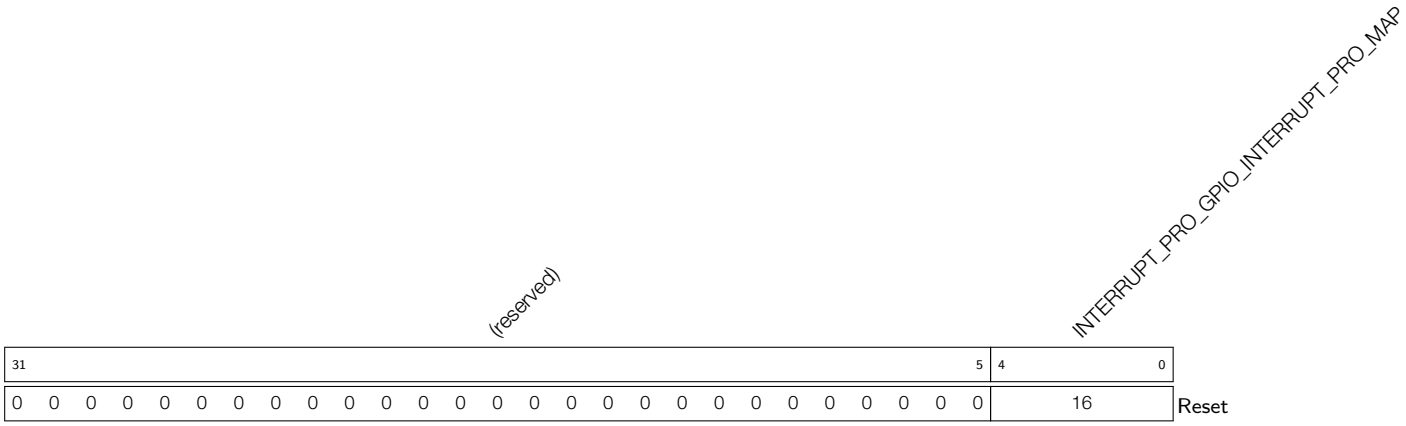
INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP 用于将 TG1_WDT_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.23: INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP_REG (0x0058)



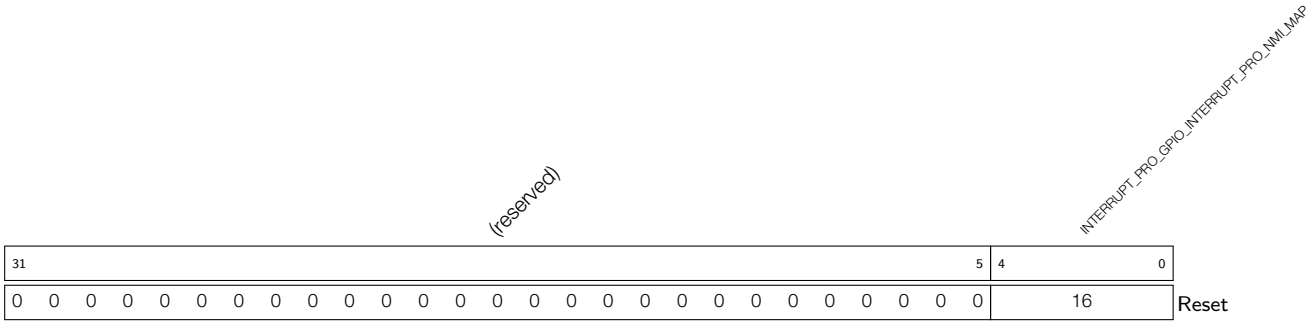
INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP 用于将 TG1_LACT_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.24: INTERRUPT_PRO_GPIO_INTERRUPT_PRO_MAP_REG (0x005C)



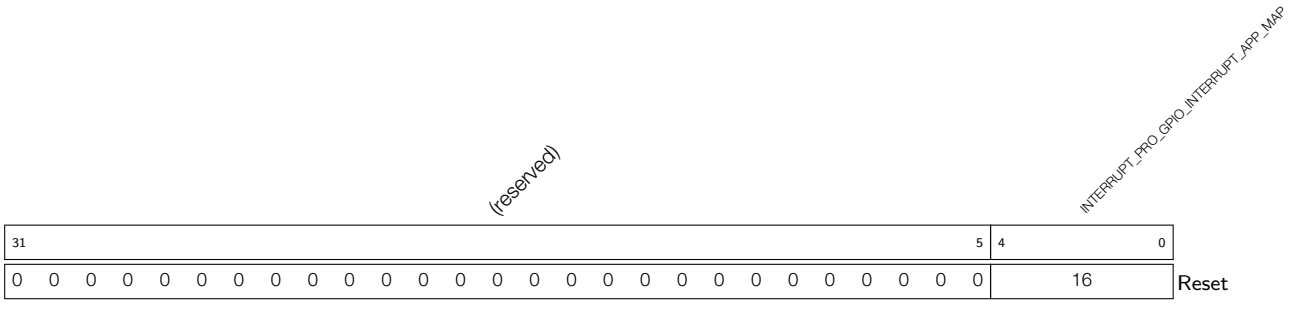
INTERRUPT_PRO_GPIO_INTERRUPT_PRO_MAP 用于将 GPIO_INTERRUPT_PRO 中断信号映射至 CPU 中断。(读/写)

Register 4.25: INTERRUPT_PRO_GPIO_INTERRUPT_PRO_NMI_MAP_REG (0x0060)



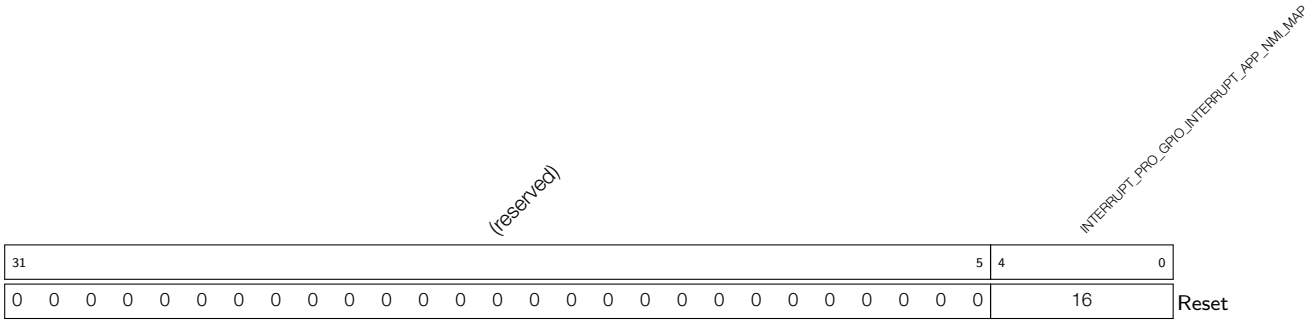
INTERRUPT_PRO_GPIO_INTERRUPT_PRO_NMI_MAP 用于将 GPIO_INTERRUPT_PRO_NMI 中断信号映射至 CPU 中断。(读/写)

Register 4.26: INTERRUPT_PRO_GPIO_INTERRUPT_APP_MAP_REG (0x0064)



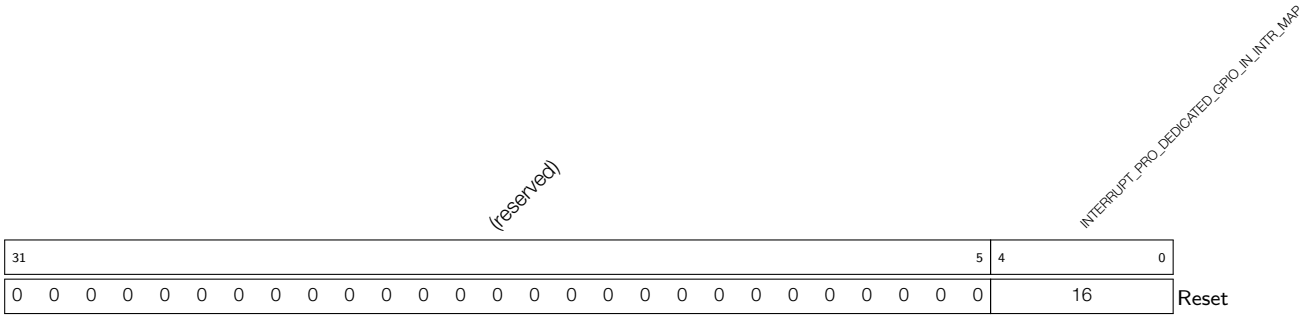
INTERRUPT_PRO_GPIO_INTERRUPT_APP_MAP 用于将 GPIO_INTERRUPT_APP 中断信号映射至 CPU 中断。(读/写)

Register 4.27: INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP_REG (0x0068)



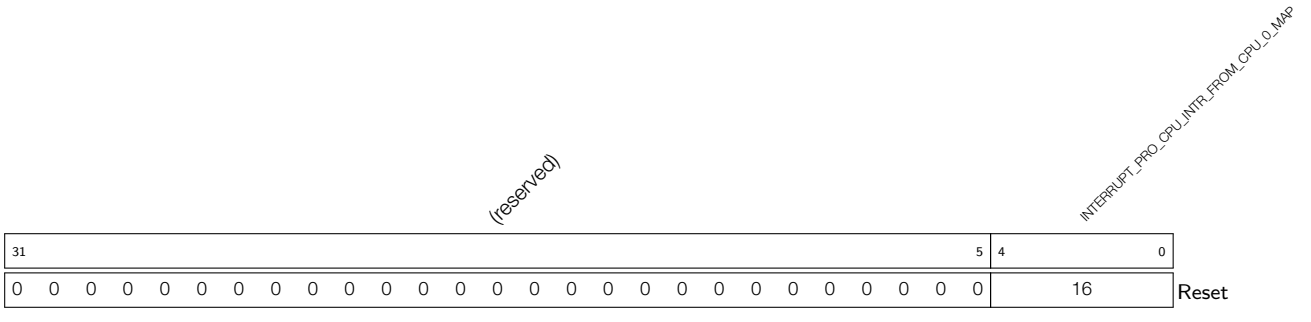
INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP 用于将 GPIO_INTERRUPT_APP 中断信号映射至 CPU 中断。(读/写)

Register 4.28: INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP_REG (0x006C)



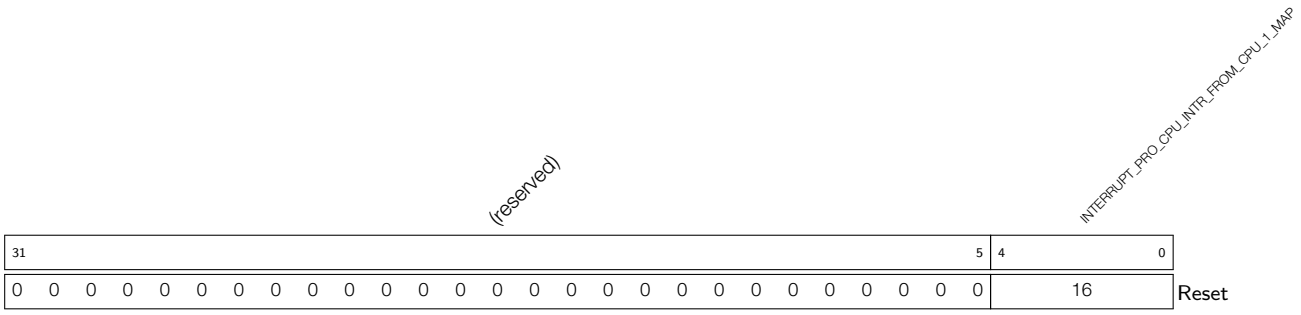
INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP 用于将 DEDICATED_GPIO_IN_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.29: INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP_REG (0x0070)



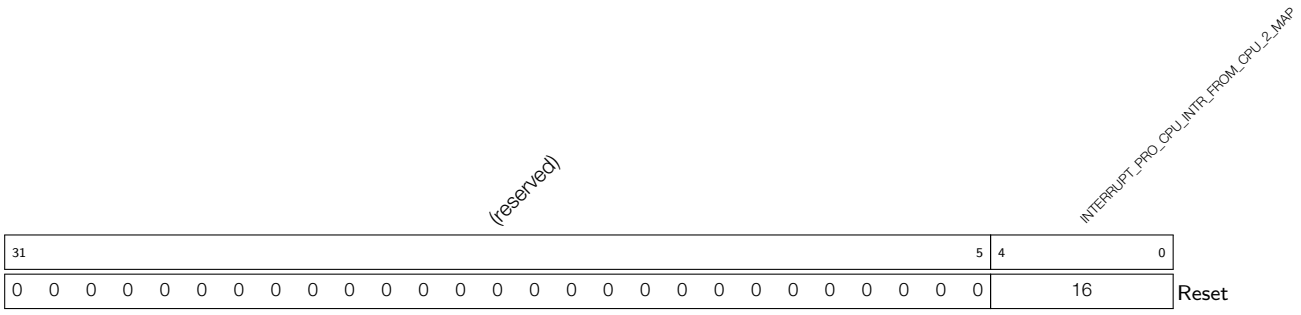
INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP 用于将 CPU_INTR_FROM_CPU_0 中断信号映射至 CPU 中断。(读/写)

Register 4.30: INTERRUPT_PRO_CPU_INTR_FROM_CPU_1_MAP_REG (0x0074)



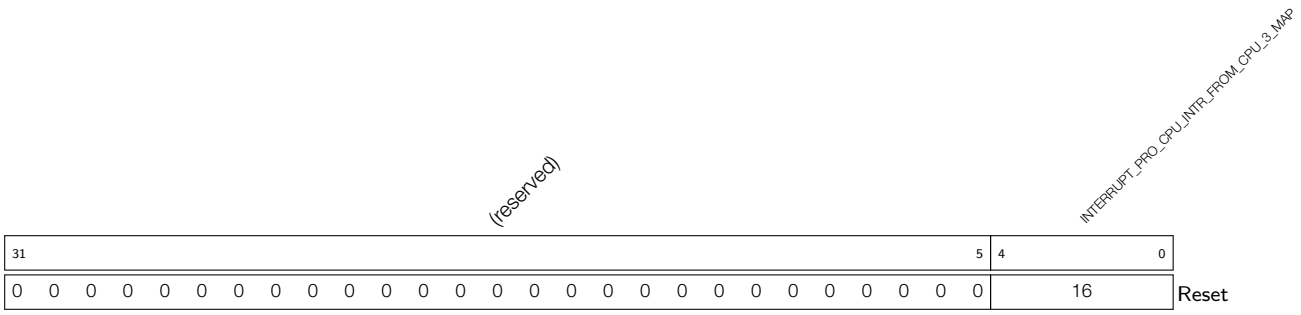
INTERRUPT_PRO_CPU_INTR_FROM_CPU_1_MAP 用于将 CPU_INTR_FROM_CPU_1 中断信号映射至 CPU 中断。(读/写)

Register 4.31: INTERRUPT_PRO_CPU_INTR_FROM_CPU_2_MAP_REG (0x0078)



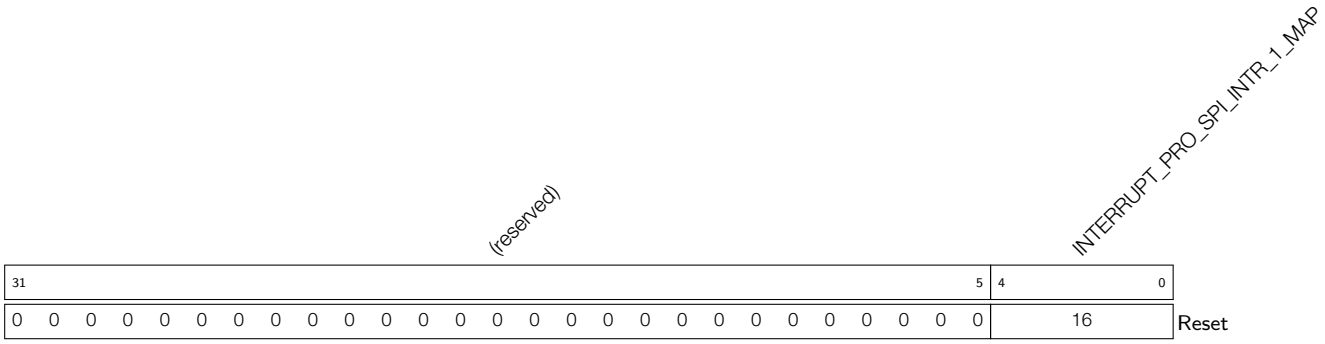
INTERRUPT_PRO_CPU_INTR_FROM_CPU_2_MAP 用于将 CPU_INTR_FROM_CPU_2 中断信号映射至 CPU 中断。(读/写)

Register 4.32: INTERRUPT_PRO_CPU_INTR_FROM_CPU_3_MAP_REG (0x007C)



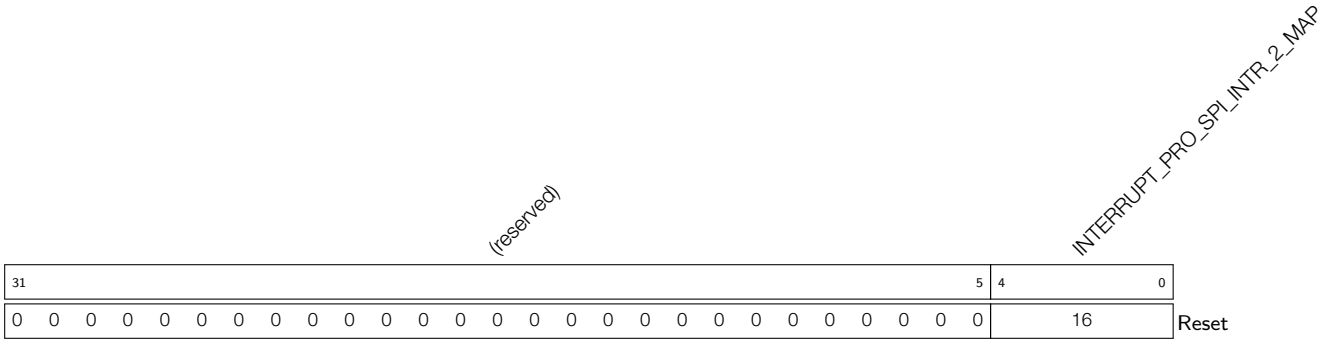
INTERRUPT_PRO_CPU_INTR_FROM_CPU_3_MAP 用于将 CPU_INTR_FROM_CPU_3 中断信号映射至 CPU 中断。(读/写)

Register 4.33: INTERRUPT_PRO_SPI_INTR_1_MAP_REG (0x0080)



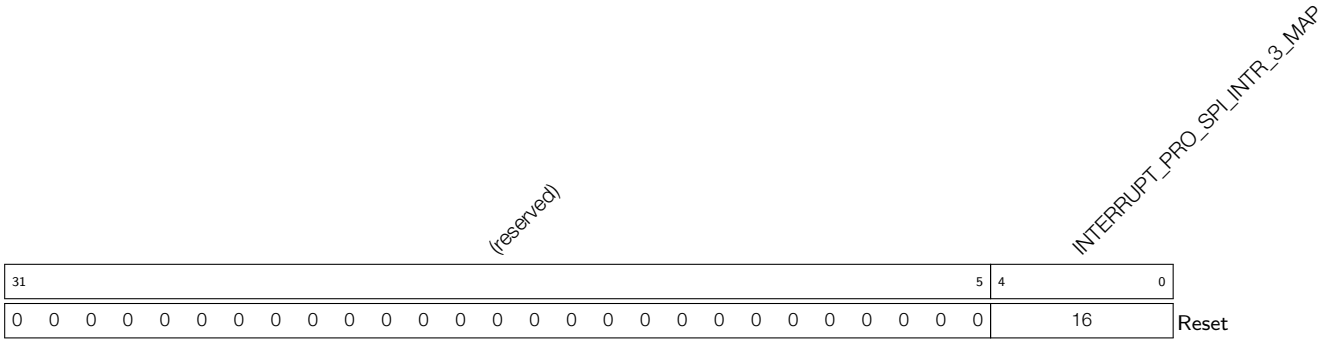
INTERRUPT_PRO_SPI_INTR_1_MAP 用于将 SPI_INTR_1 中断信号映射至 CPU 中断。(读/写)

Register 4.34: INTERRUPT_PRO_SPI_INTR_2_MAP_REG (0x0084)



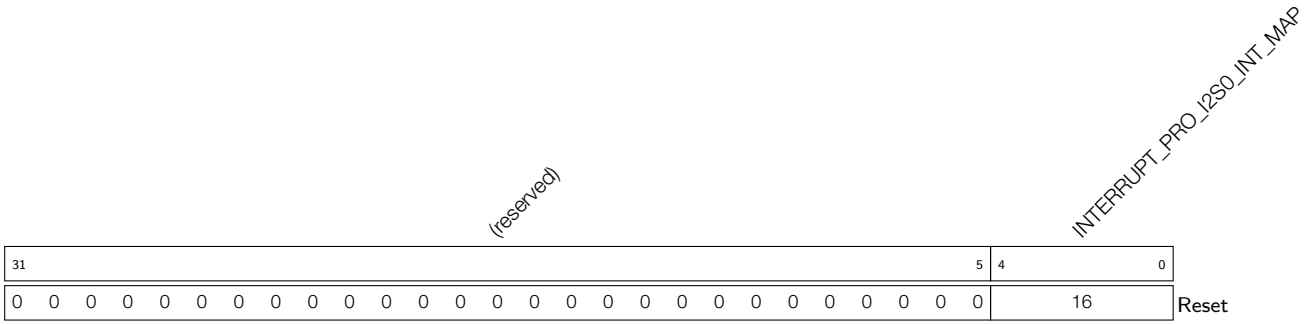
INTERRUPT_PRO_SPI_INTR_2_MAP 用于将 SPI_INTR_2 中断信号映射至 CPU 中断。(读/写)

Register 4.35: INTERRUPT_PRO_SPI_INTR_3_MAP_REG (0x0088)



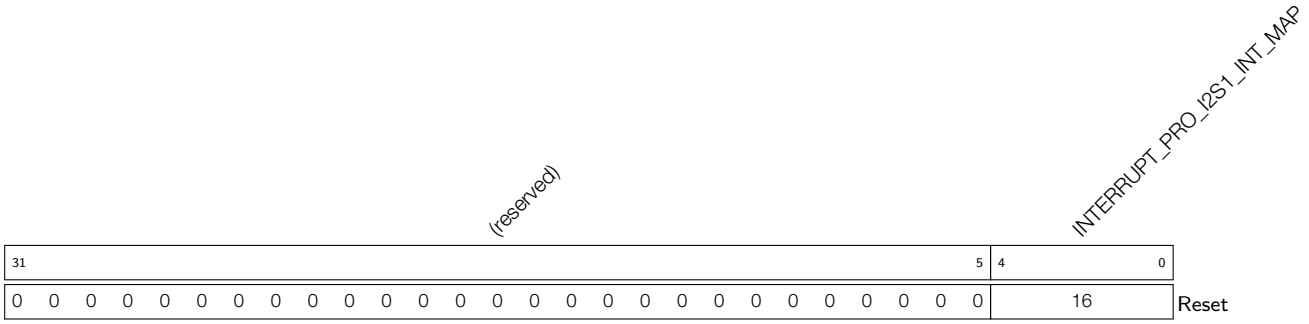
INTERRUPT_PRO_SPI_INTR_3_MAP 用于将 SPI_INTR_3 中断信号映射至 CPU 中断。(读/写)

Register 4.36: INTERRUPT_PRO_I2S0_INT_MAP_REG (0x008C)



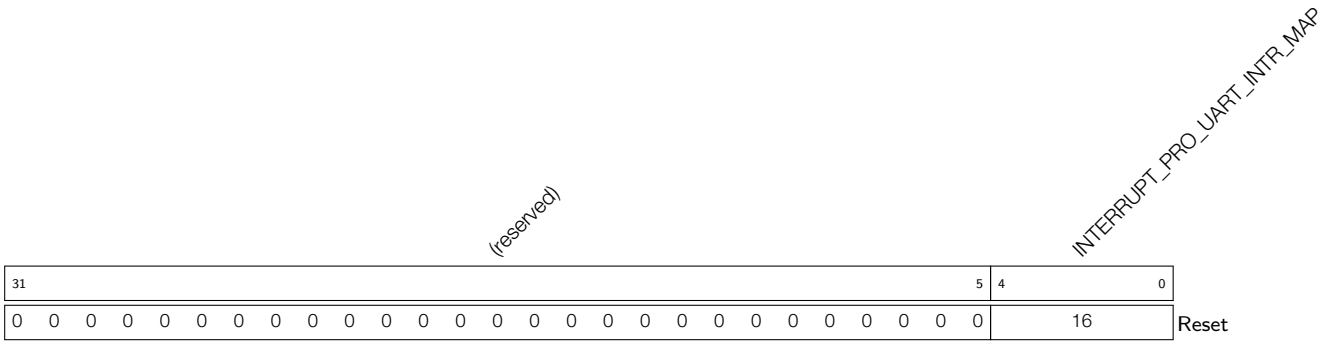
INTERRUPT_PRO_I2S0_INT_MAP 用于将 I2S0_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.37: INTERRUPT_PRO_I2S1_INT_MAP_REG (0x0090)



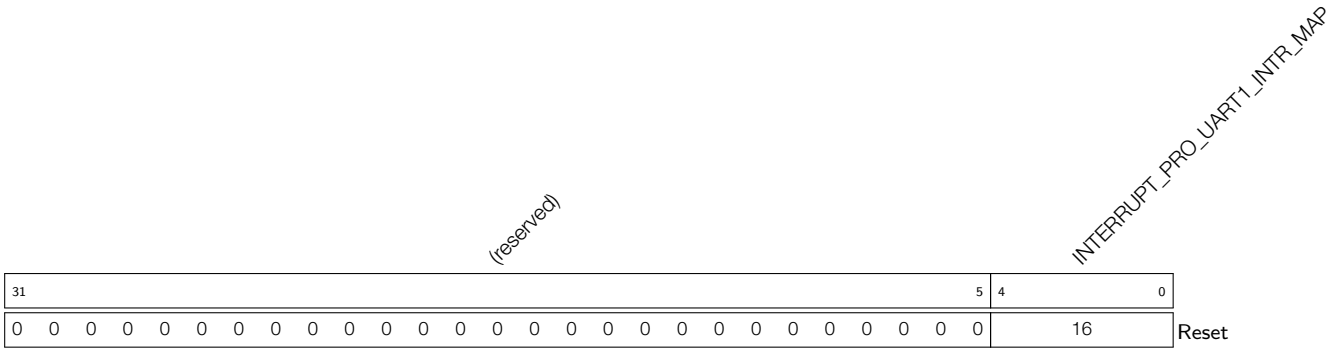
INTERRUPT_PRO_I2S1_INT_MAP 用于将 I2S1_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.38: INTERRUPT_PRO_UART_INTR_MAP_REG (0x0094)



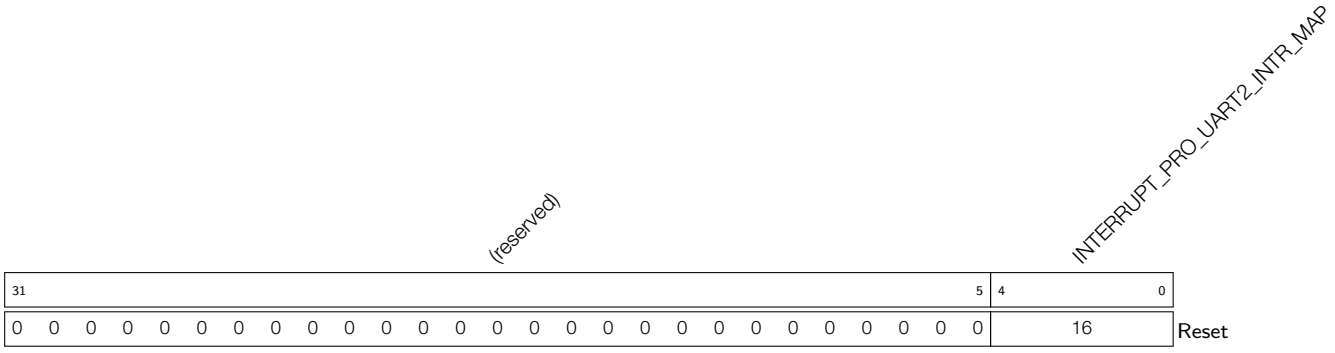
INTERRUPT_PRO_UART_INTR_MAP 用于将 UART_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.39: INTERRUPT_PRO_UART1_INTR_MAP_REG (0x0098)



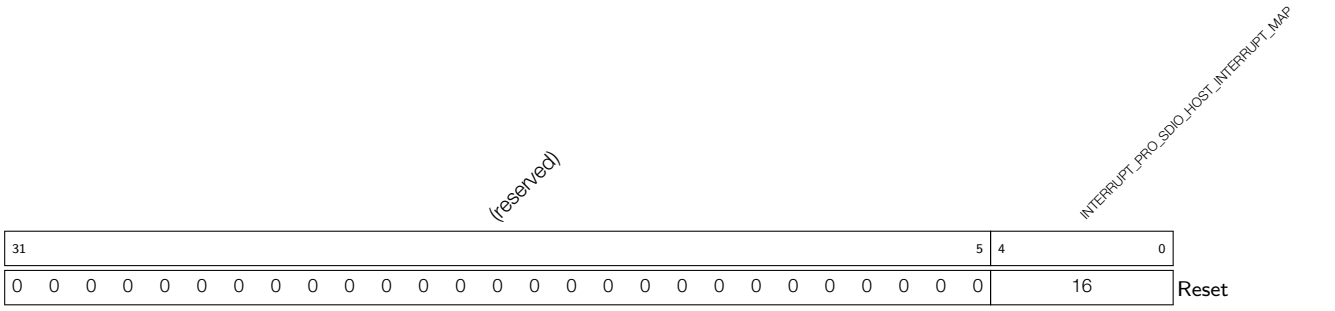
INTERRUPT_PRO_UART1_INTR_MAP 用于将 UART1_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.40: INTERRUPT_PRO_UART2_INTR_MAP_REG (0x009C)



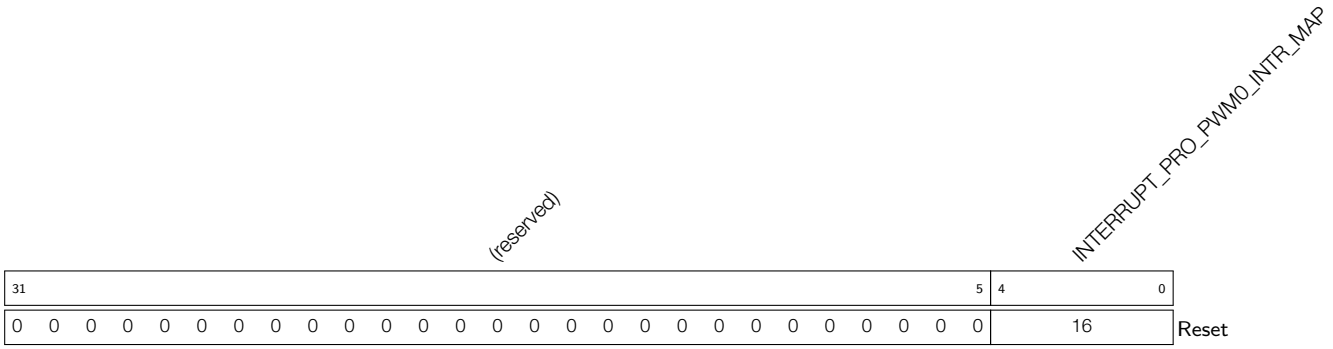
INTERRUPT_PRO_UART2_INTR_MAP 用于将 UART2_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.41: INTERRUPT_PRO_SDIO_HOST_INTERRUPT_MAP_REG (0x00A0)



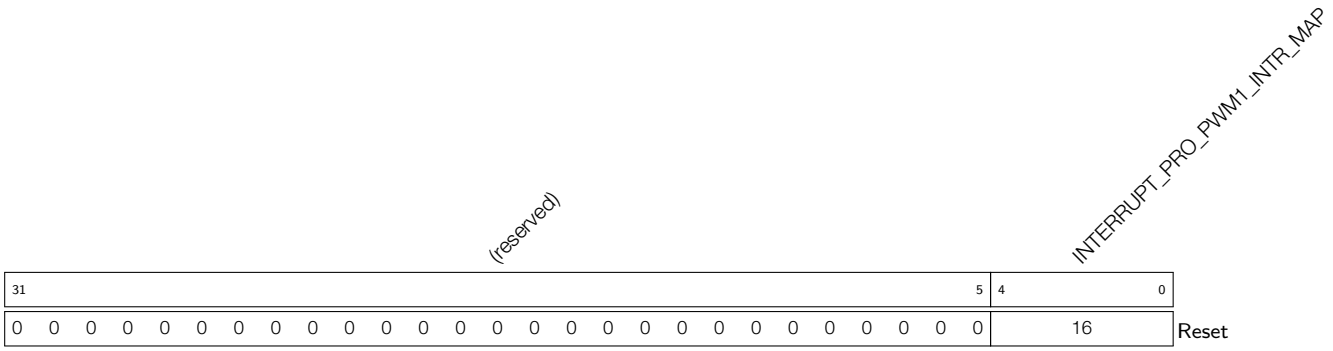
INTERRUPT_PRO_SDIO_HOST_INTERRUPT_MAP 用于将 SDIO_HOST 中断信号映射至 CPU 中断。(读/写)

Register 4.42: INTERRUPT_PRO_PWM0_INTR_MAP_REG (0x00A4)



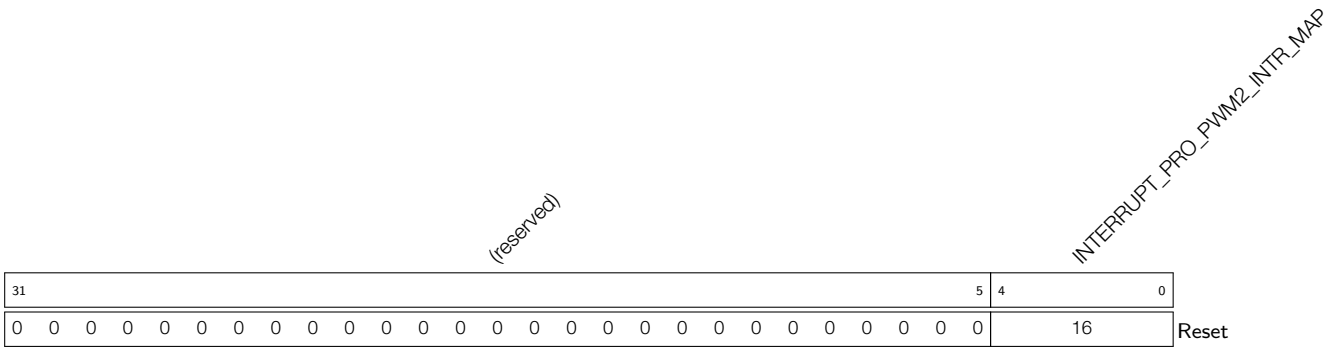
INTERRUPT_PRO_PWM0_INTR_MAP 用于将 PWM0_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.43: INTERRUPT_PRO_PWM1_INTR_MAP_REG (0x00A8)



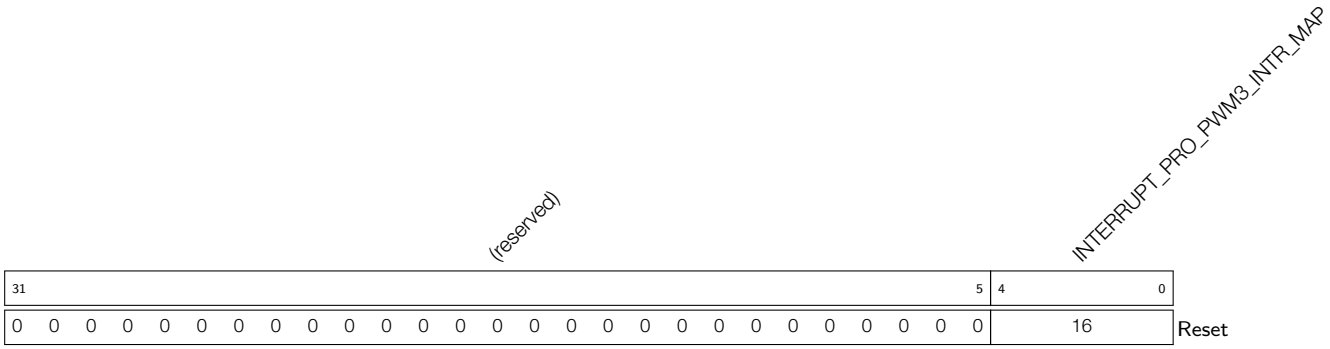
INTERRUPT_PRO_PWM1_INTR_MAP 用于将 PWM1_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.44: INTERRUPT_PRO_PWM2_INTR_MAP_REG (0x00AC)



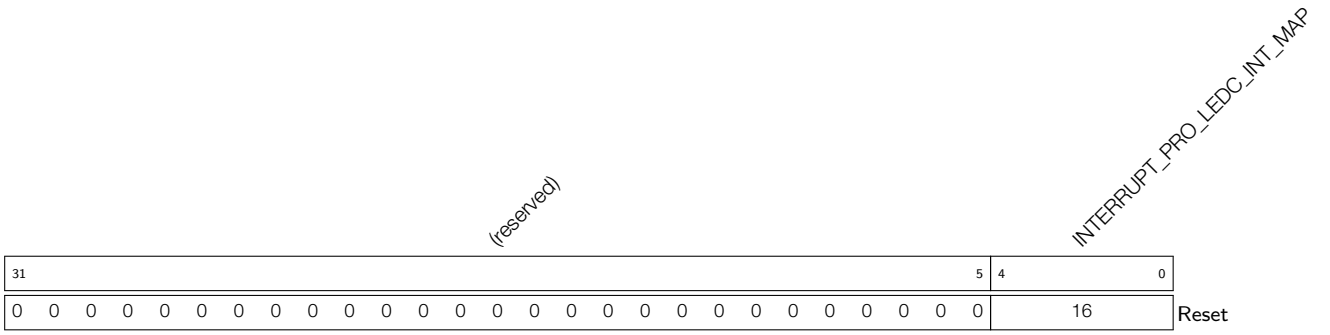
INTERRUPT_PRO_PWM2_INTR_MAP 用于将 PWM2_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.45: INTERRUPT_PRO_PWM3_INTR_MAP_REG (0x00B0)



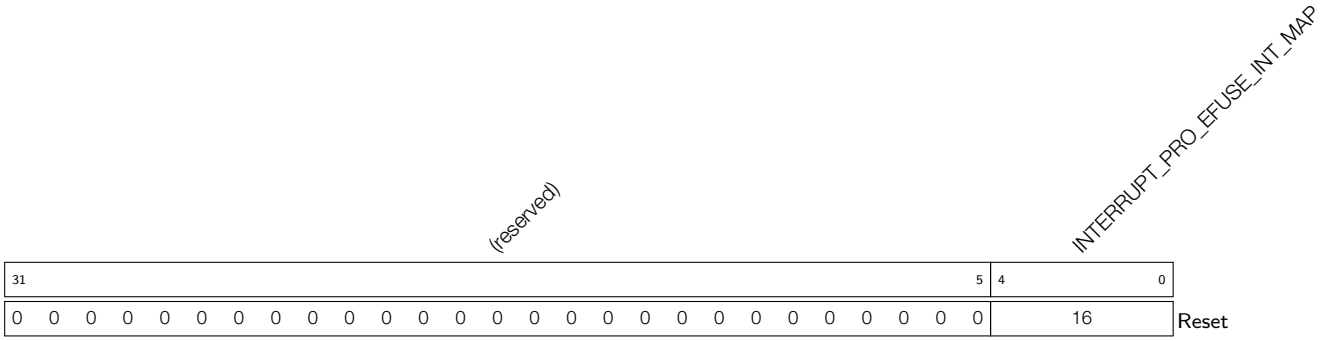
INTERRUPT_PRO_PWM3_INTR_MAP 用于将 PWM3_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.46: INTERRUPT_PRO_LEDC_INT_MAP_REG (0x00B4)



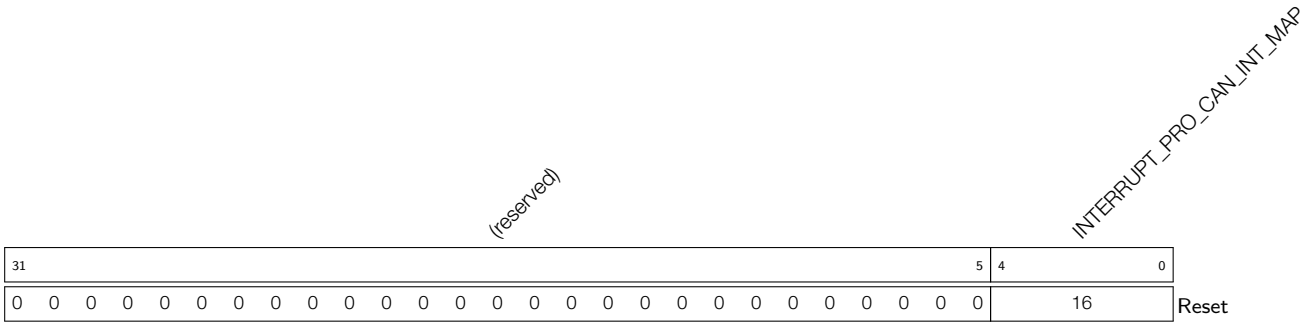
INTERRUPT_PRO_LEDC_INT_MAP 用于将 LEDC_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.47: INTERRUPT_PRO_EFUSE_INT_MAP_REG (0x00B8)



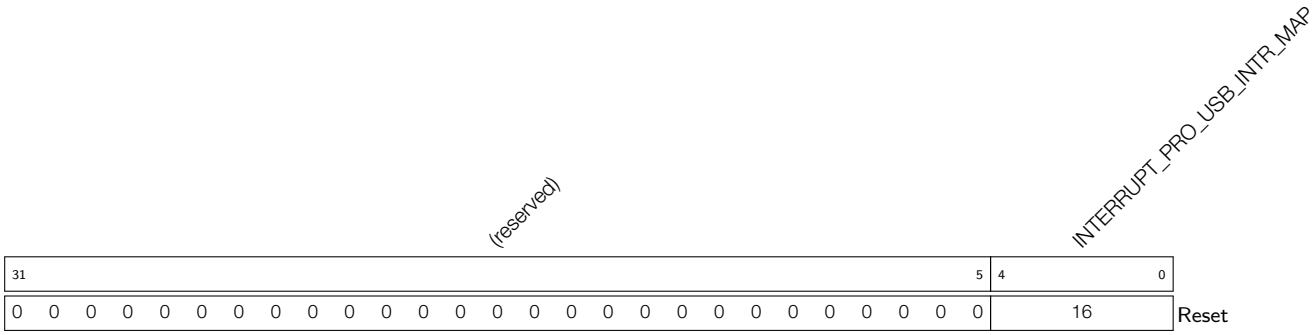
INTERRUPT_PRO_EFUSE_INT_MAP 用于将 EFUSE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.48: INTERRUPT_PRO_CAN_INT_MAP_REG (0x00BC)



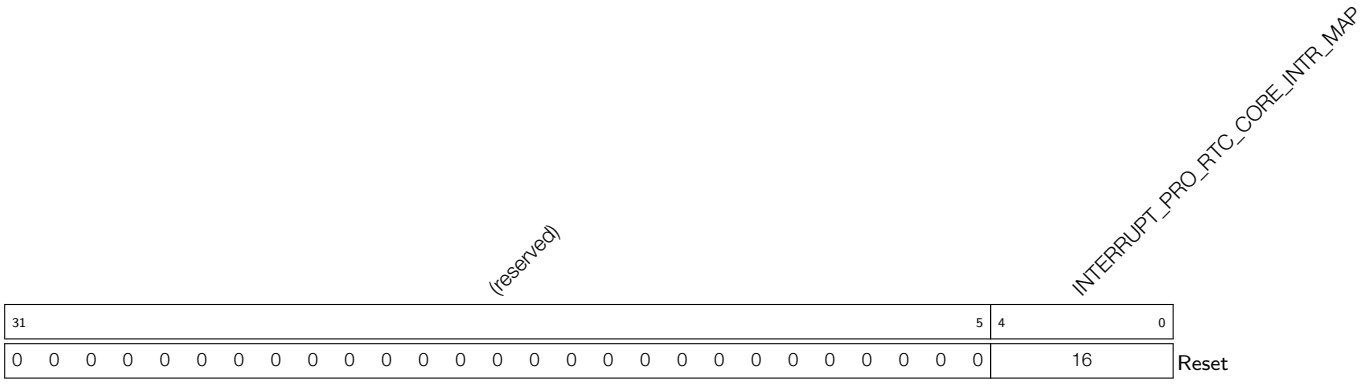
INTERRUPT_PRO_CAN_INT_MAP 用于将 CAN_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.49: INTERRUPT_PRO_USB_INTR_MAP_REG (0x00C0)



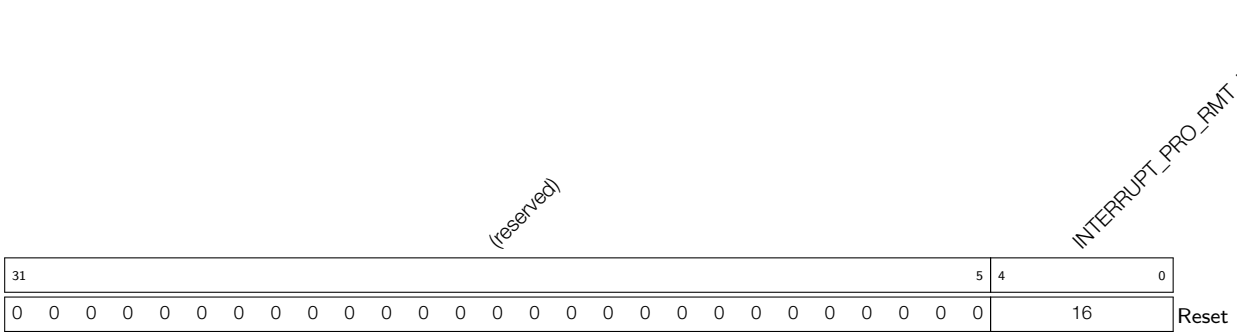
INTERRUPT_PRO_USB_INTR_MAP 用于将 USB_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.50: INTERRUPT_PRO_RTC_CORE_INTR_MAP_REG (0x00C4)



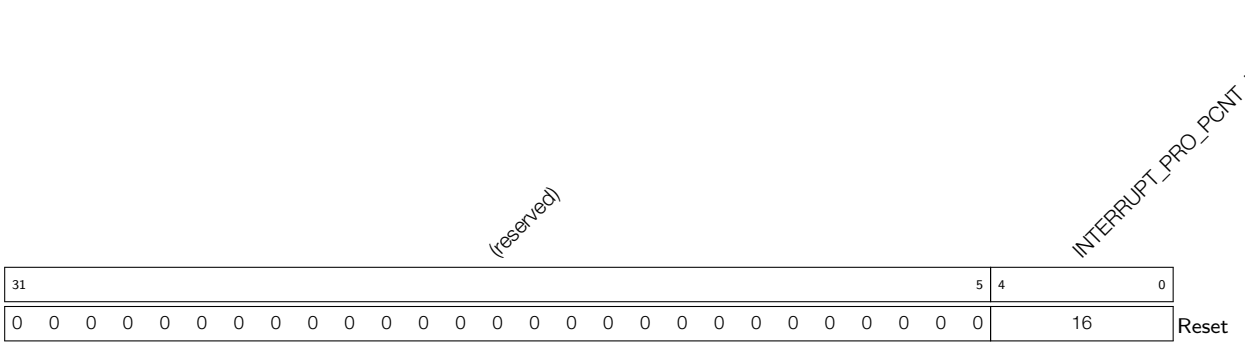
INTERRUPT_PRO_RTC_CORE_INTR_MAP 用于将 RTC_CORE_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.51: INTERRUPT_PRO_RMT_INTR_MAP_REG (0x00C8)



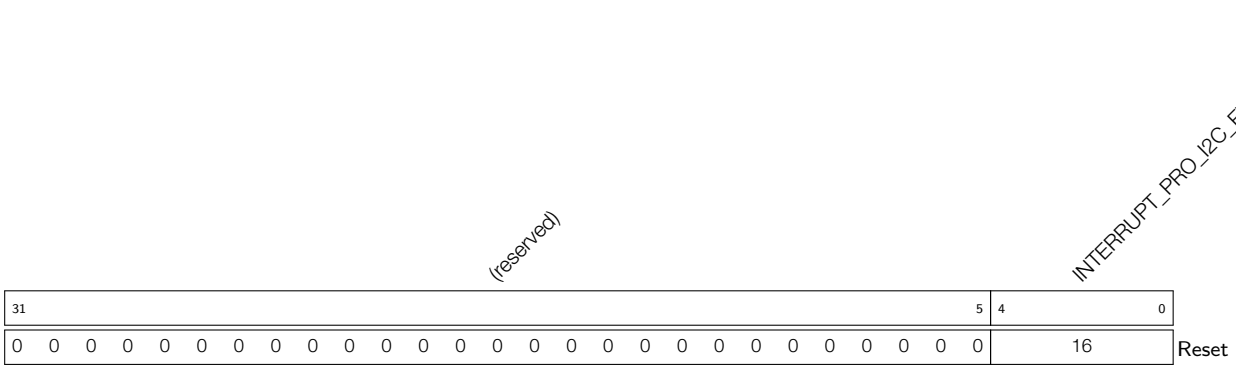
INTERRUPT_PRO_RMT_INTR_MAP 用于将 RMT_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.52: INTERRUPT_PRO_PCNT_INTR_MAP_REG (0x00CC)



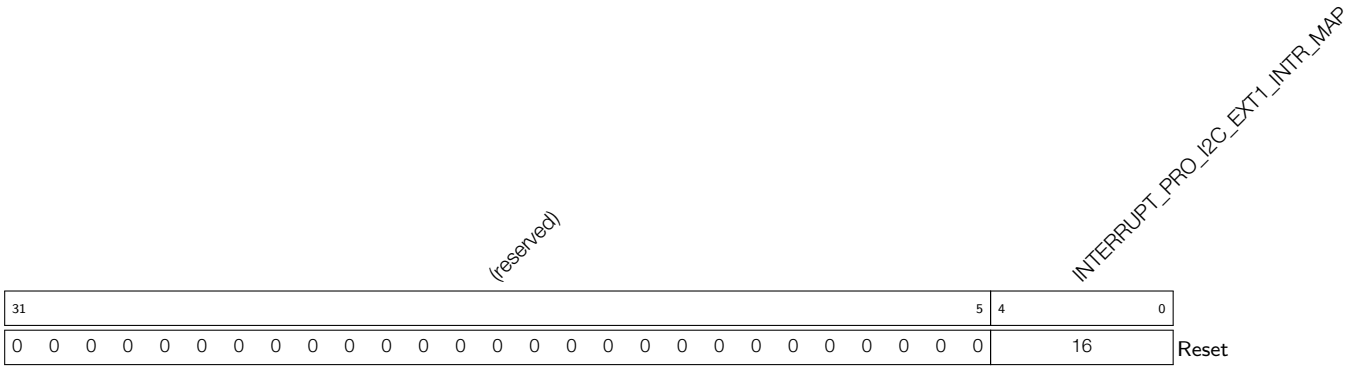
INTERRUPT_PRO_PCNT_INTR_MAP 用于将 PCNT_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.53: INTERRUPT_PRO_I2C_EXT0_INTR_MAP_REG (0x00D0)



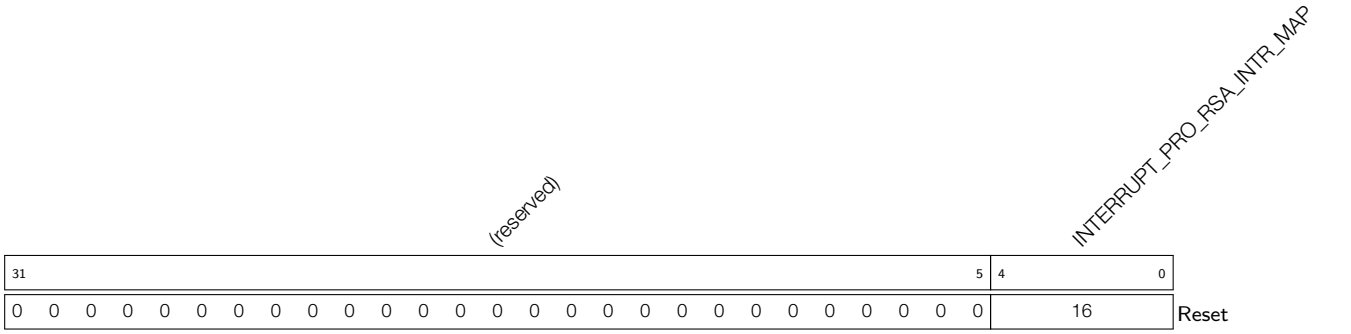
INTERRUPT_PRO_I2C_EXT0_INTR_MAP 用于将 I2C_EXT0_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.54: INTERRUPT_PRO_I2C_EXT1_INTR_MAP_REG (0x00D4)



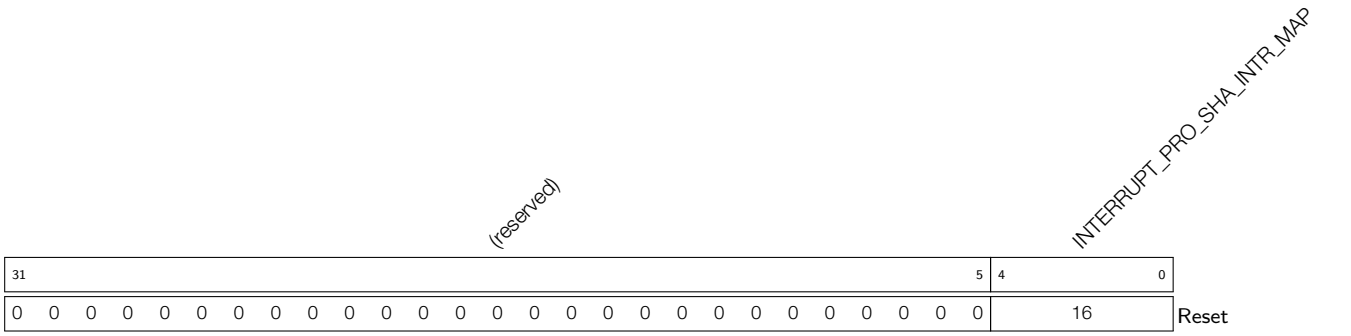
INTERRUPT_PRO_I2C_EXT1_INTR_MAP 用于将 I2C_EXT1_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.55: INTERRUPT_PRO_RSA_INTR_MAP_REG (0x00D8)



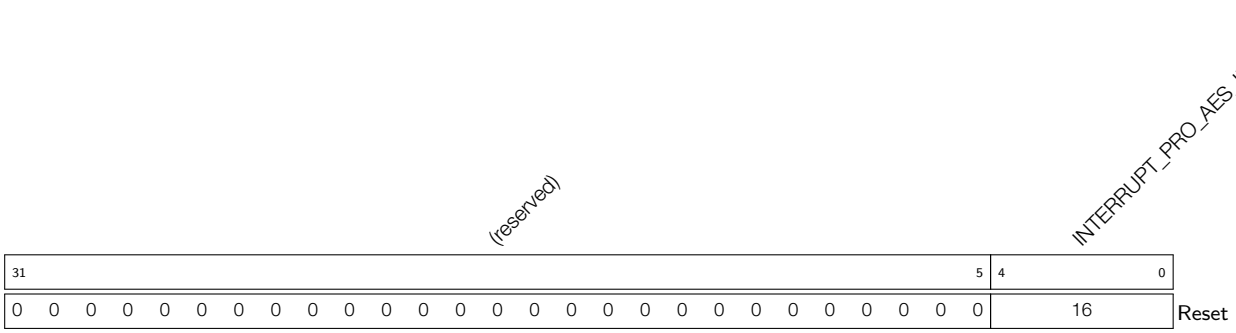
INTERRUPT_PRO_RSA_INTR_MAP 用于将 RSA_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.56: INTERRUPT_PRO_SHA_INTR_MAP_REG (0x00DC)



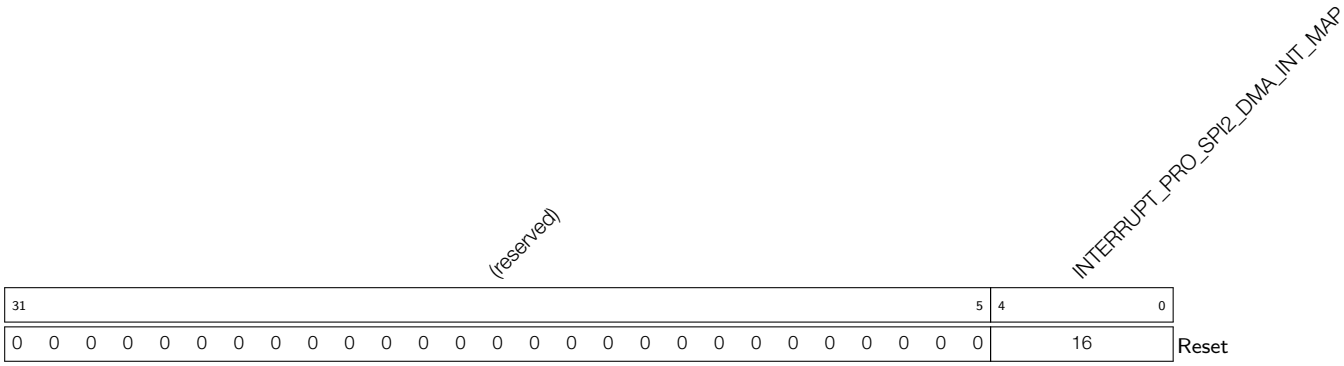
INTERRUPT_PRO_SHA_INTR_MAP 用于将 SHA_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.57: INTERRUPT_PRO_AES_INTR_MAP_REG (0x00E0)



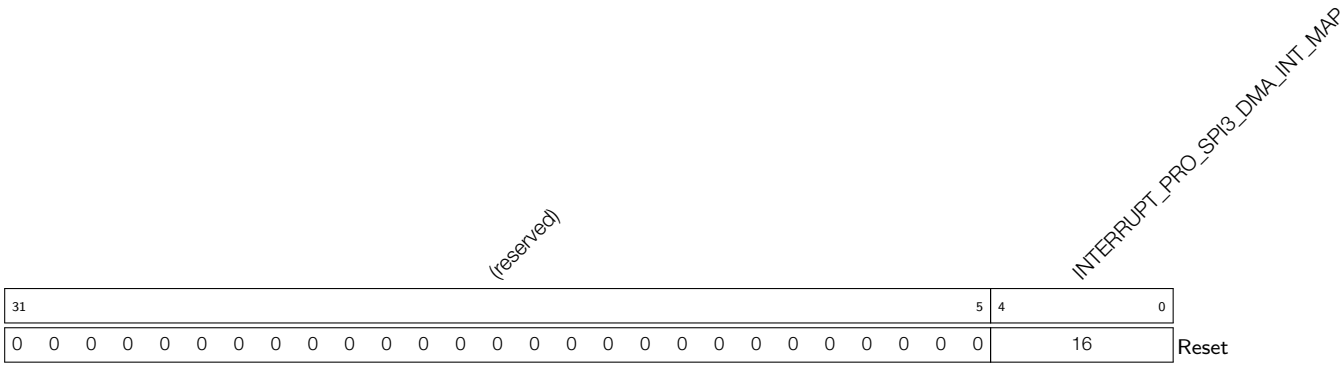
INTERRUPT_PRO_AES_INTR_MAP 用于将 AES_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.58: INTERRUPT_PRO_SPI2_DMA_INT_MAP_REG (0x00E4)



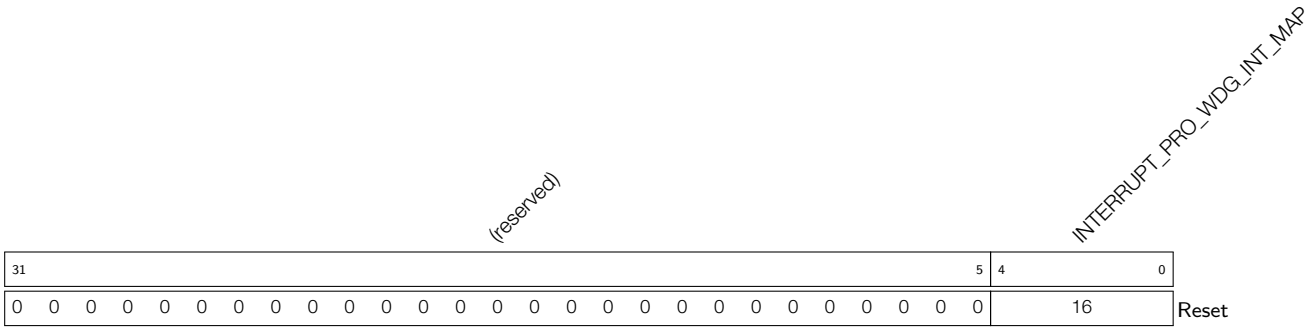
INTERRUPT_PRO_SPI2_DMA_INT_MAP 用于将 SPI2_DMA_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.59: INTERRUPT_PRO_SPI3_DMA_INT_MAP_REG (0x00E8)



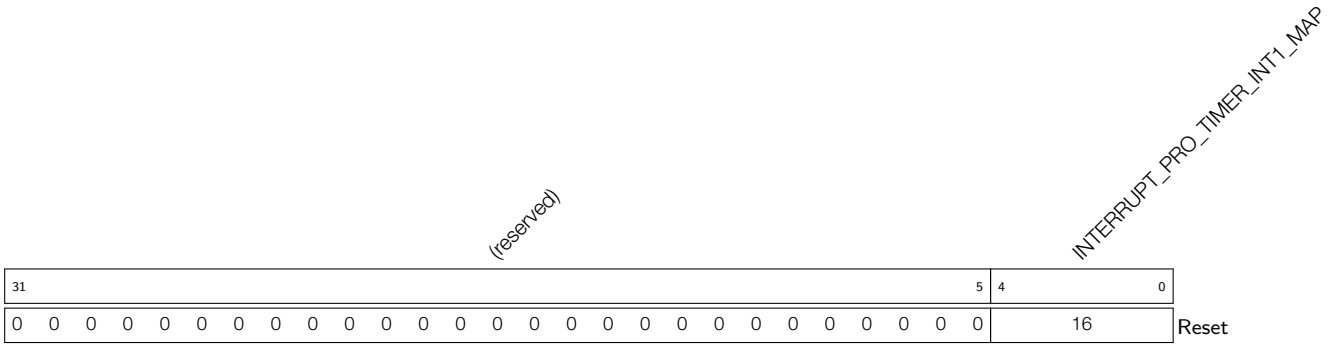
INTERRUPT_PRO_SPI3_DMA_INT_MAP 用于将 SPI3_DMA_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.60: INTERRUPT_PRO_WDG_INT_MAP_REG (0x00EC)



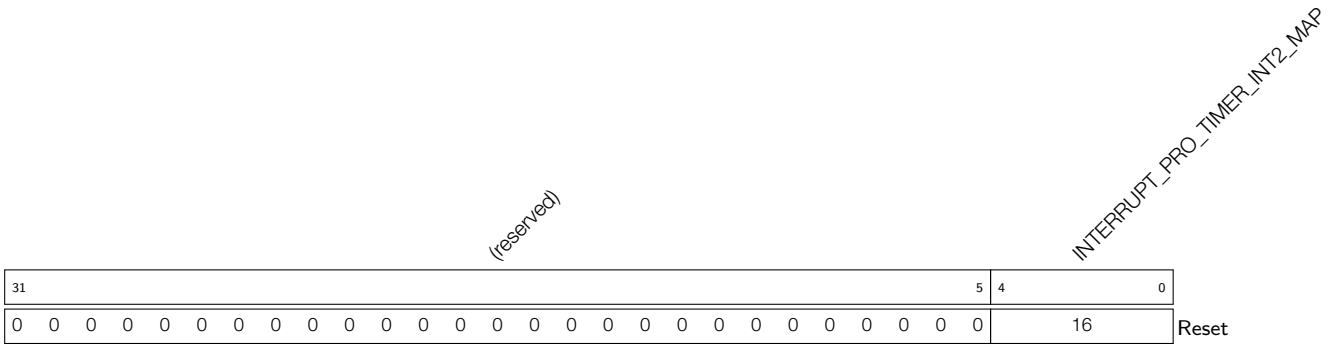
INTERRUPT_PRO_WDG_INT_MAP 用于将 WDG_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.61: INTERRUPT_PRO_TIMER_INT1_MAP_REG (0x00F0)



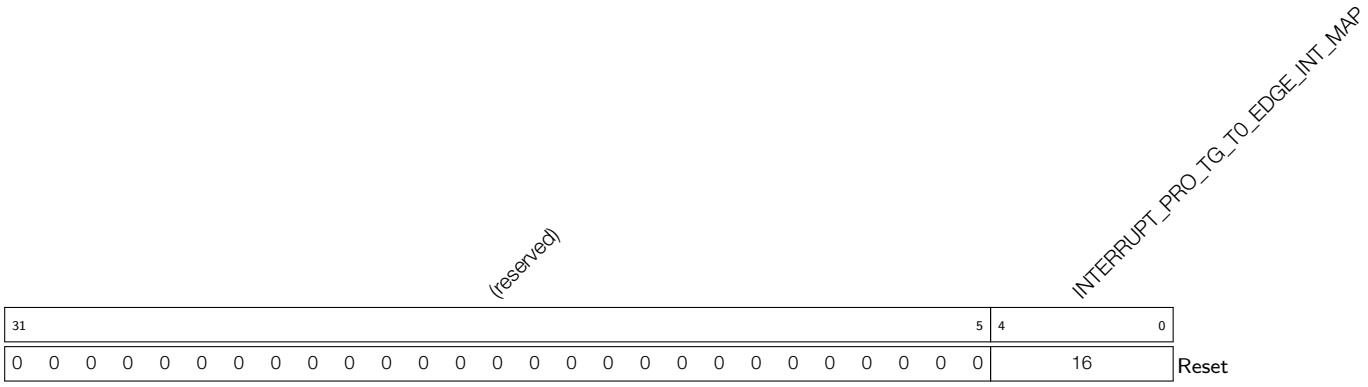
INTERRUPT_PRO_TIMER_INT1_MAP 用于将 TIMER_INT1 中断信号映射至 CPU 中断。(读/写)

Register 4.62: INTERRUPT_PRO_TIMER_INT2_MAP_REG (0x00F4)



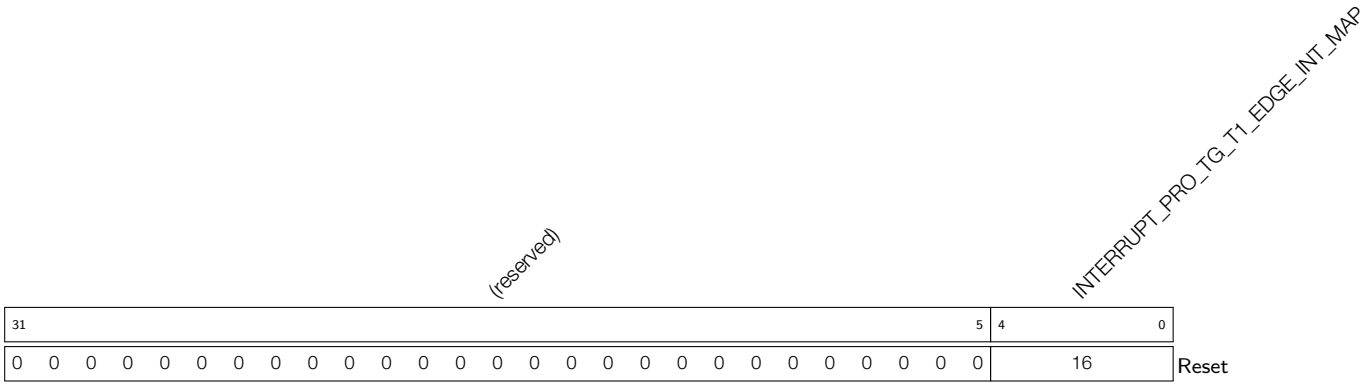
INTERRUPT_PRO_TIMER_INT2_MAP 用于将 TIMER_INT2 中断信号映射至 CPU 中断。(读/写)

Register 4.63: INTERRUPT_PRO_TG_T0_EDGE_INT_MAP_REG (0x00F8)



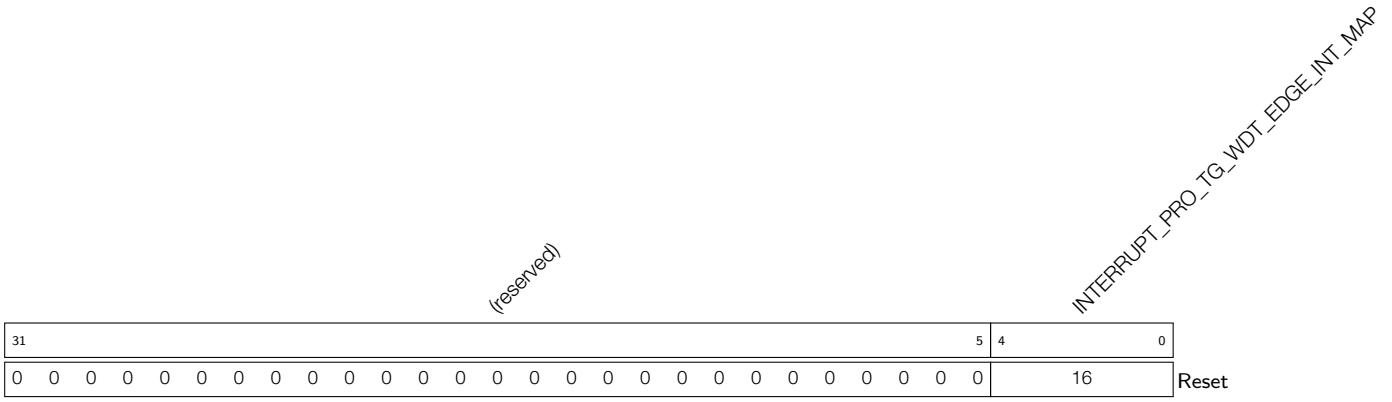
INTERRUPT_PRO_TG_T0_EDGE_INT_MAP 用于将 TG_T0_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.64: INTERRUPT_PRO_TG_T1_EDGE_INT_MAP_REG (0x00FC)



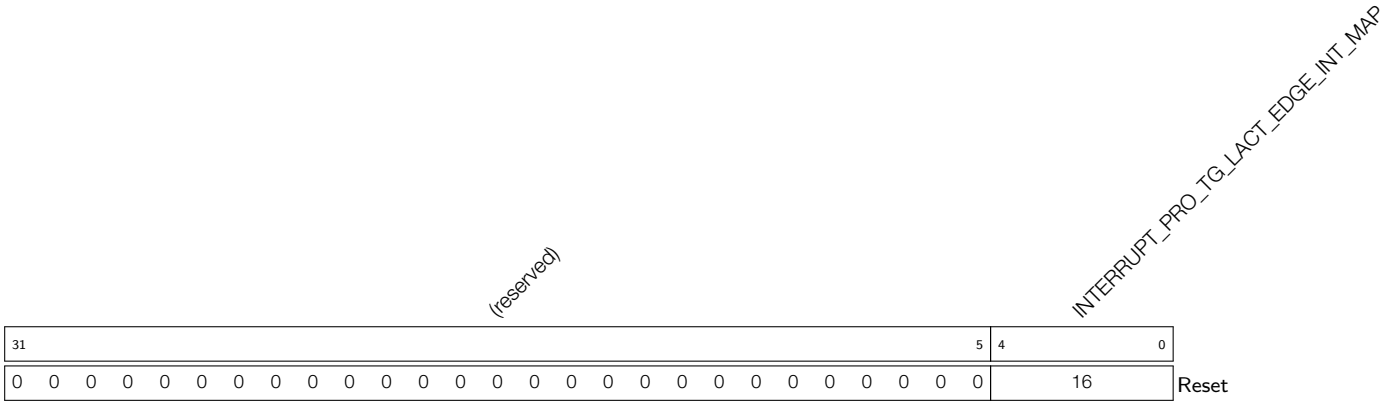
INTERRUPT_PRO_TG_T1_EDGE_INT_MAP 用于将 TG_T1_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.65: INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP_REG (0x0100)



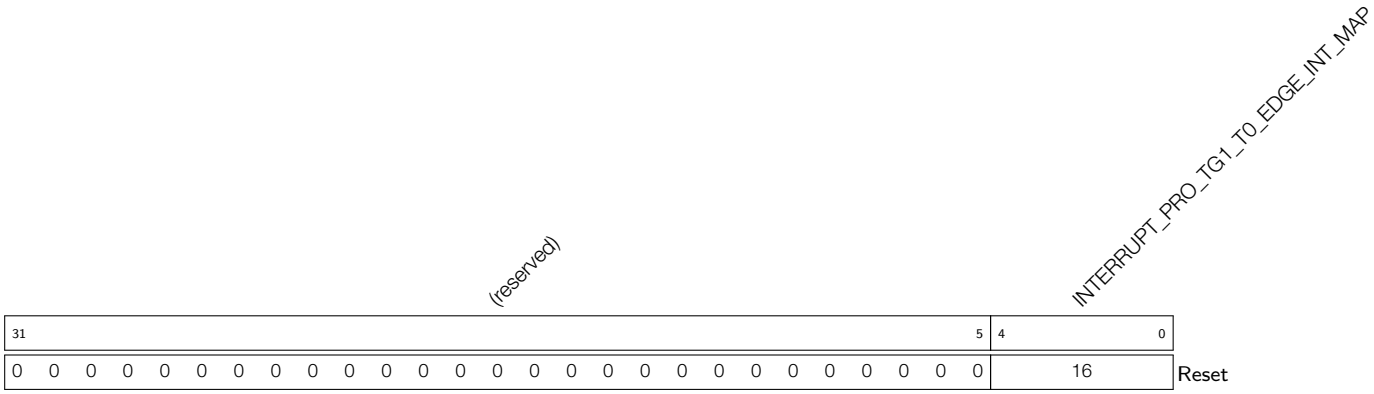
INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP 用于将 TG_WDT_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.66: INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP_REG (0x0104)



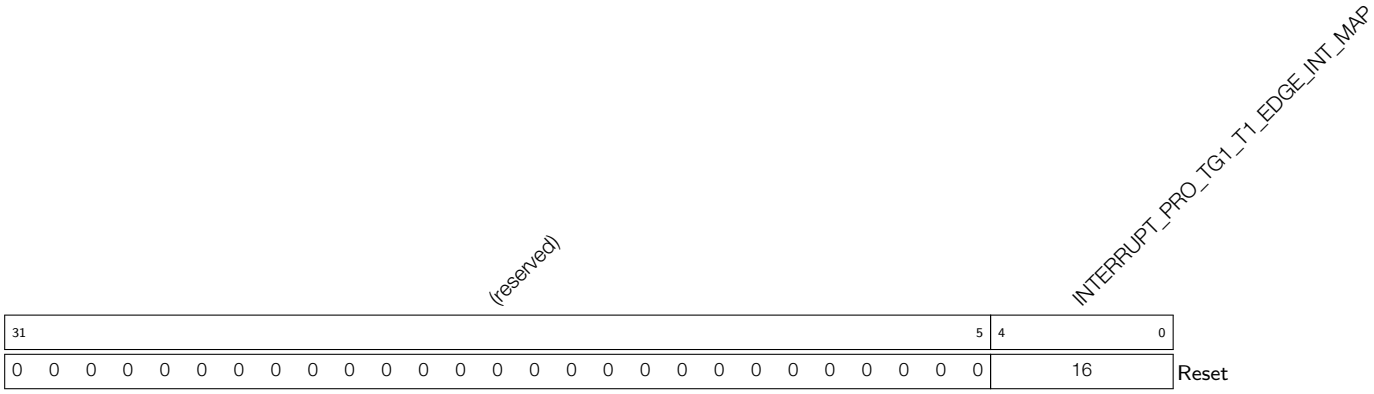
INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP 用于将 TG_LACT_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.67: INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP_REG (0x0108)



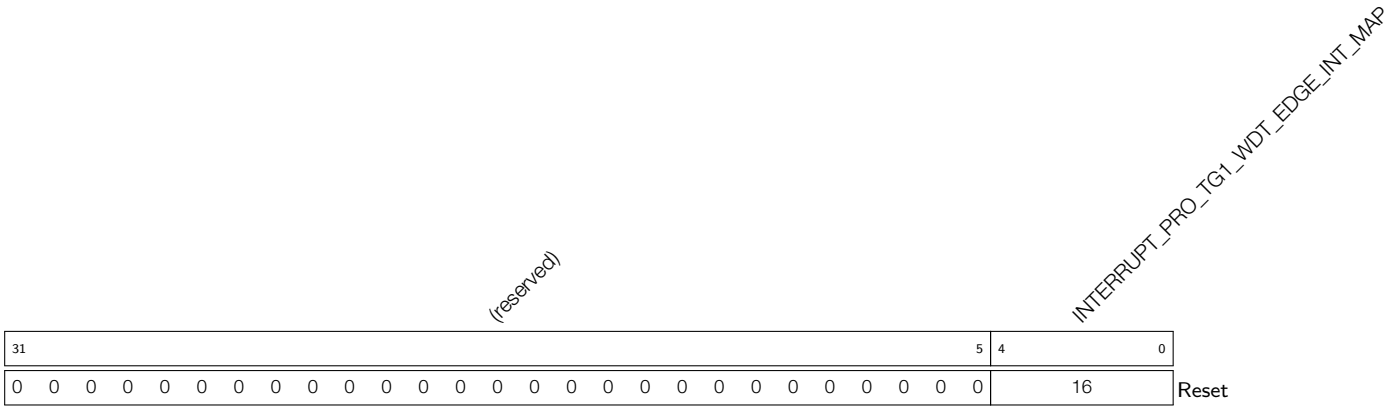
INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP 用于将 TG1_T0_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.68: INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP_REG (0x010C)



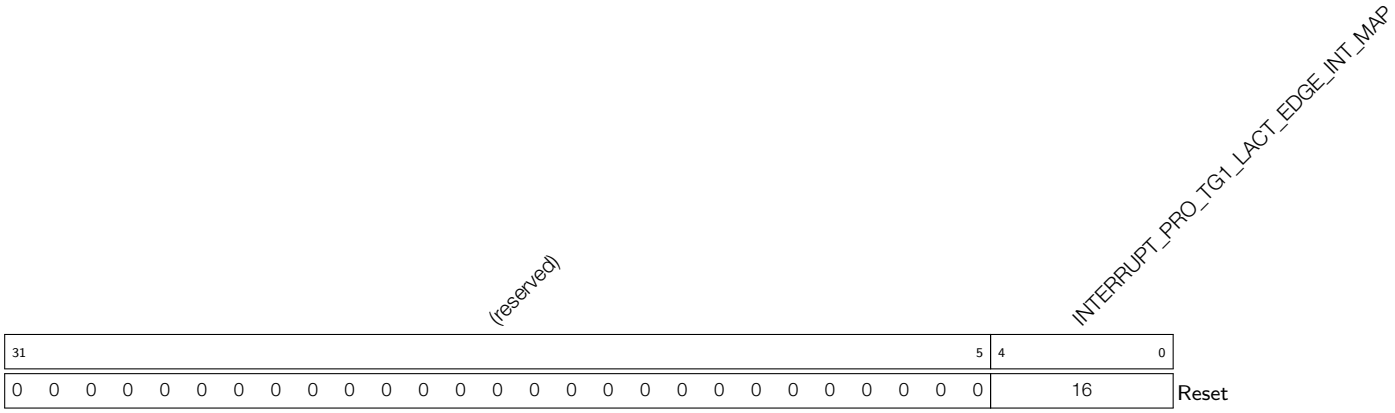
INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP 用于将 TG1_T1_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.69: INTERRUPT_PRO_TG1_WDT_EDGE_INT_MAP_REG (0x0110)



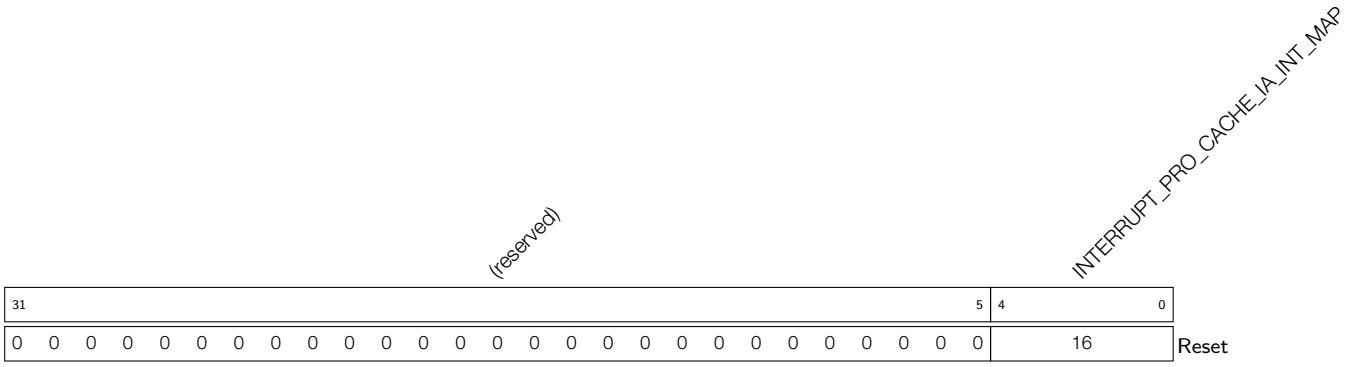
INTERRUPT_PRO_TG1_WDT_EDGE_INT_MAP 用于将 TG1_WDT_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.70: INTERRUPT_PRO_TG1_LACT_EDGE_INT_MAP_REG (0x0114)



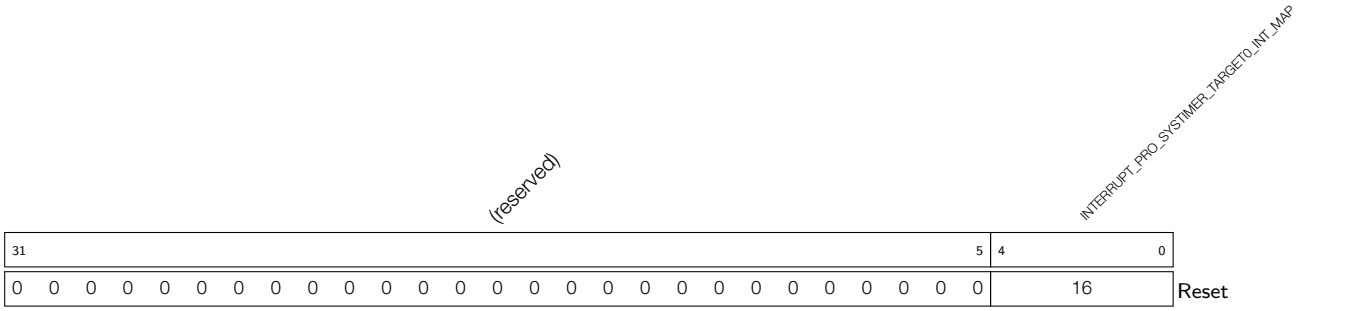
INTERRUPT_PRO_TG1_LACT_EDGE_INT_MAP 用于将 TG1_LACT_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.71: INTERRUPT_PRO_CACHE_IA_INT_MAP_REG (0x0118)



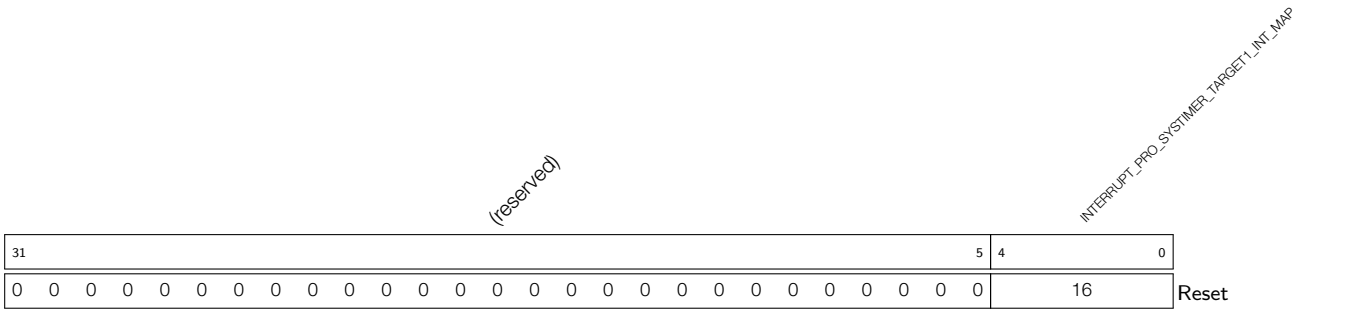
INTERRUPT_PRO_CACHE_IA_INT_MAP 用于将 CACHE_IA_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.72: INTERRUPT_PRO_SYSTIMER_TARGET0_INT_MAP_REG (0x011C)



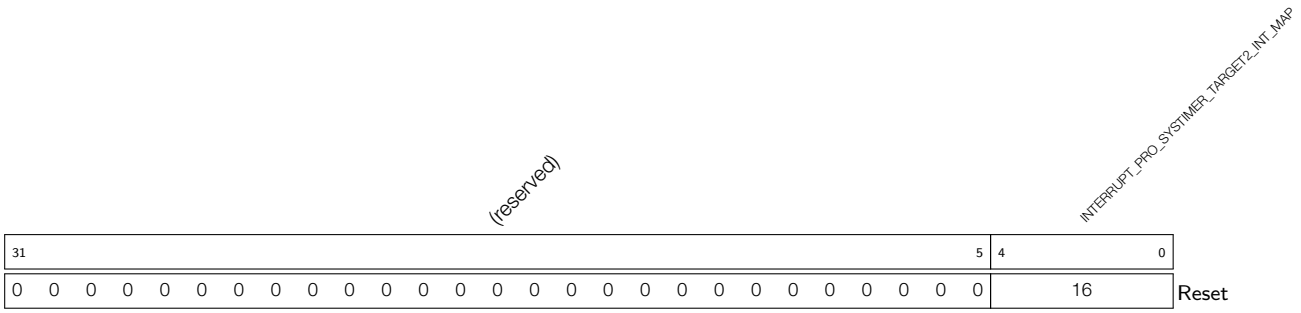
INTERRUPT_PRO_SYSTIMER_TARGET0_INT_MAP 用于将 SYSTIMER_TARGET0 中断信号映射至 CPU 中断。(读/写)

Register 4.73: INTERRUPT_PRO_SYSTIMER_TARGET1_INT_MAP_REG (0x0120)



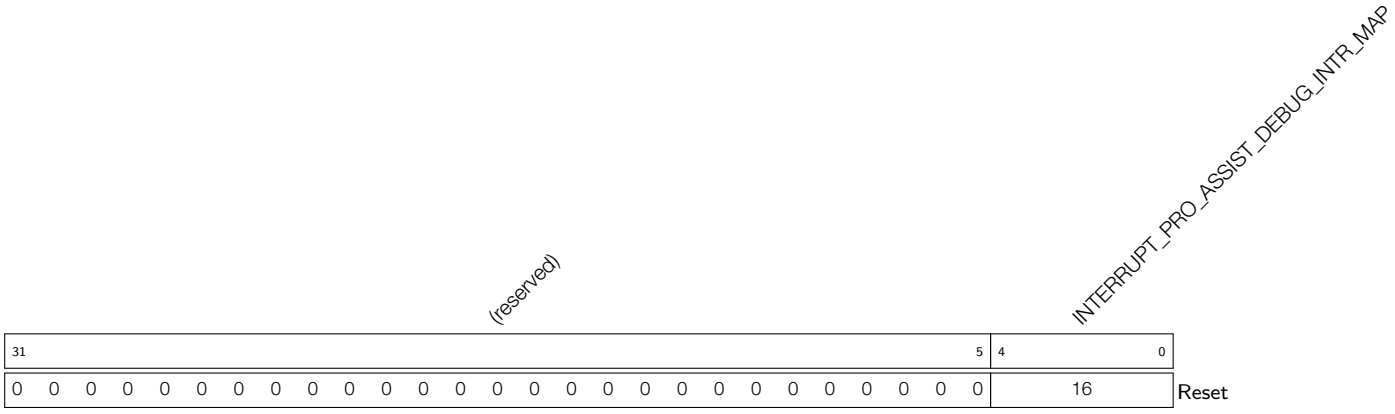
INTERRUPT_PRO_SYSTIMER_TARGET1_INT_MAP 用于将 SYSTIMER_TARGET1 中断信号映射至 CPU 中断。(读/写)

Register 4.74: INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP_REG (0x0124)



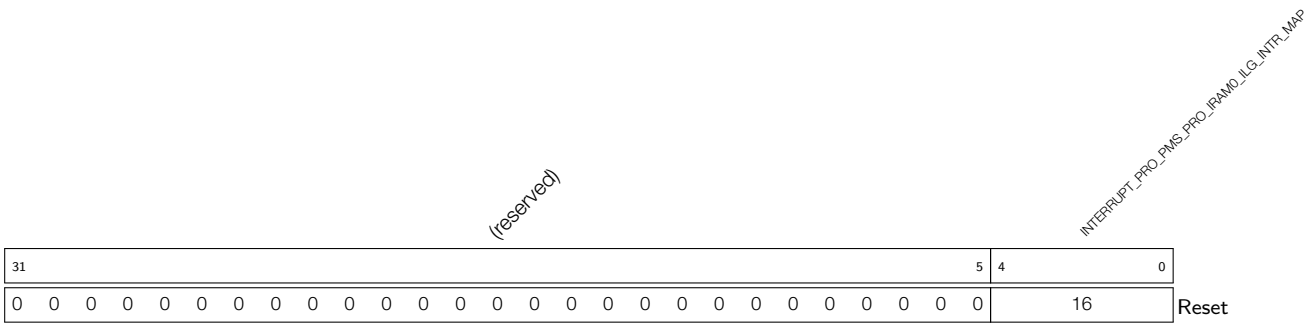
INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP 用于将 SYSTIMER_TARGET2 中断信号映射至 CPU 中断。(读/写)

Register 4.75: INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP_REG (0x0128)



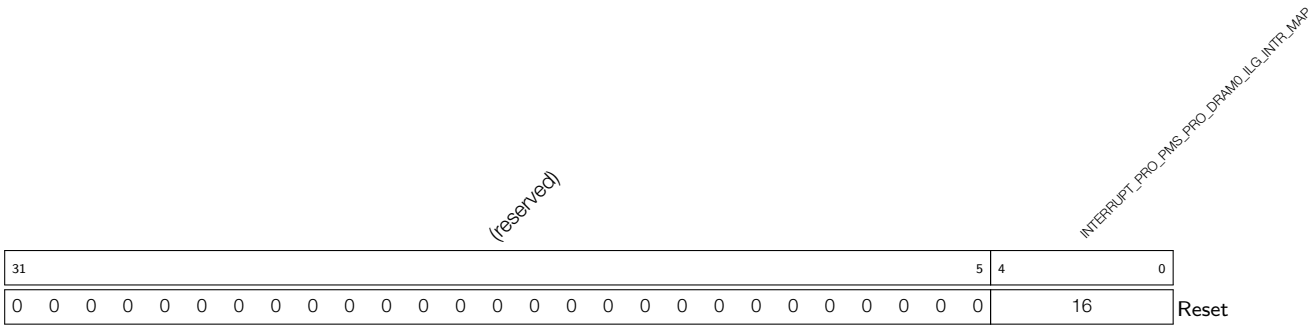
INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP 用于将 ASSIST_DEBUG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.76: INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP_REG (0x012C)



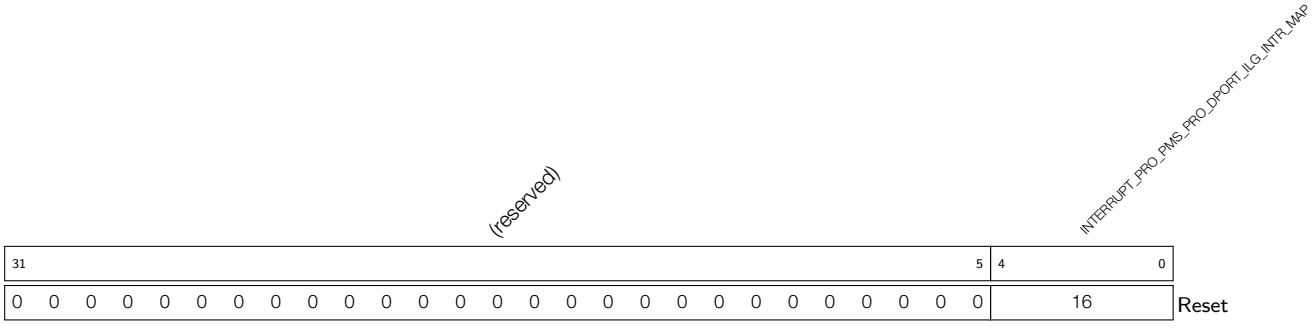
INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP 用于将 PMS_PRO_IRAM0_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.77: INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP_REG (0x0130)



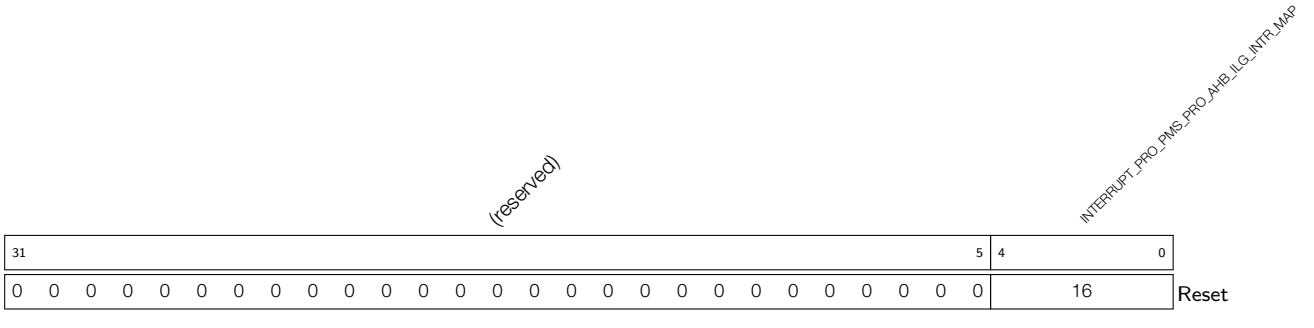
INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP 用于将 PMS_PRO_DRAM0_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.78: INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP_REG (0x0134)



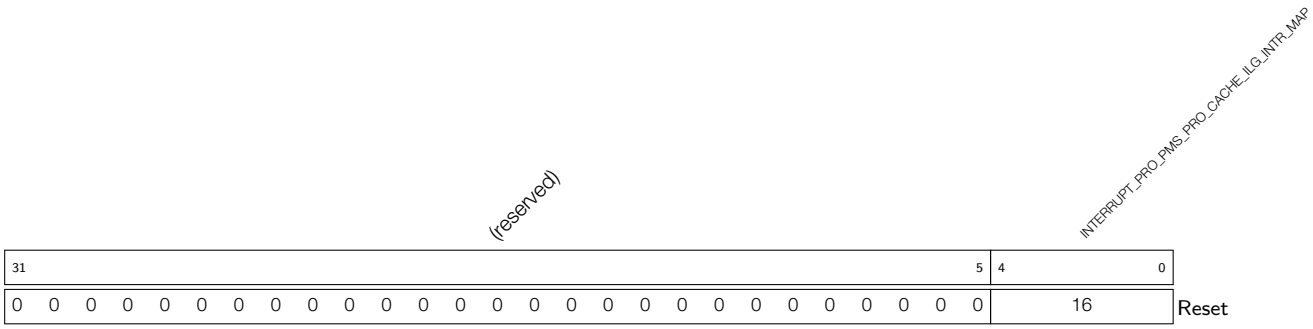
INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP 用于将 PMS_PRO_DPORT_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.79: INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP_REG (0x0138)



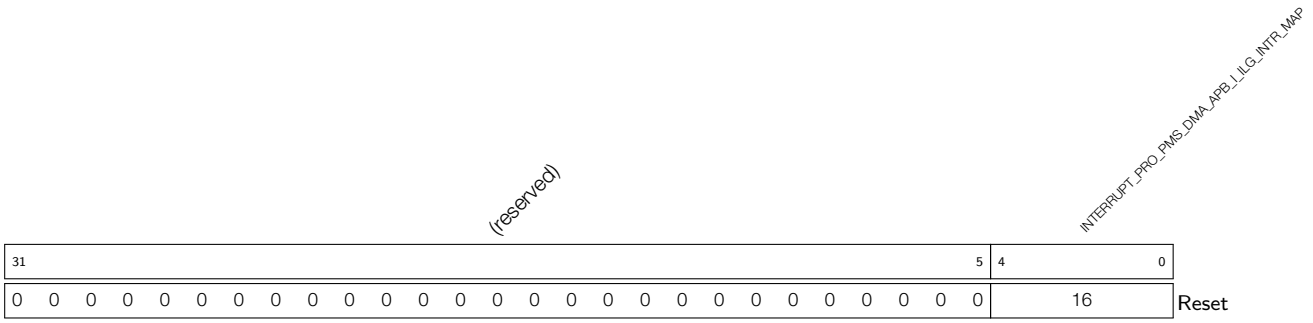
INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP 用于将 PMS_PRO_AHB_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.80: INTERRUPT_PRO_PMS_PRO_CACHE_ILG_INTR_MAP_REG (0x013C)



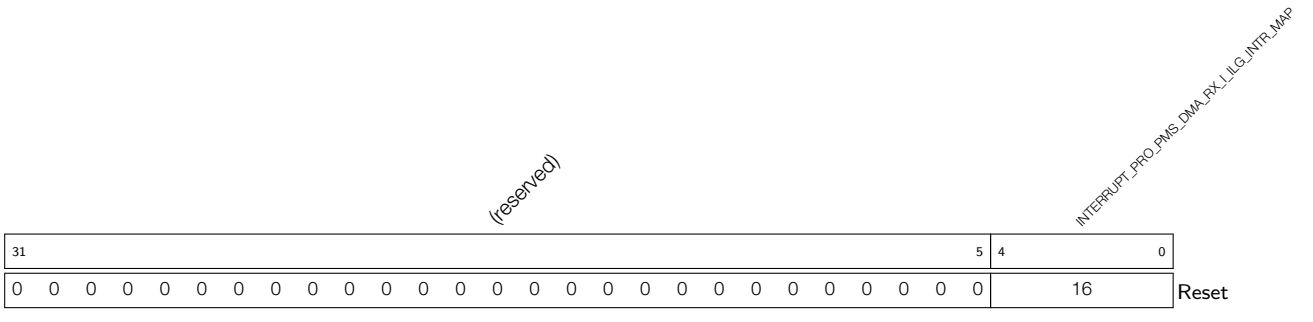
INTERRUPT_PRO_PMS_PRO_CACHE_ILG_INTR_MAP 用于将 PMS_PRO_CACHE_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.81: INTERRUPT_PRO_PMS_DMA_APB_I_ILG_INTR_MAP_REG (0x0140)



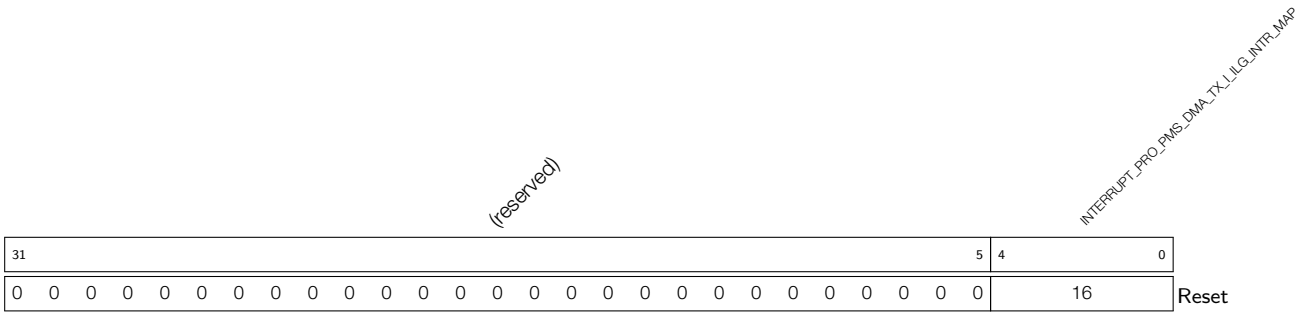
INTERRUPT_PRO_PMS_DMA_APB_I_ILG_INTR_MAP 用于将 PMS_DMA_APB_I_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.82: INTERRUPT_PRO_PMS_DMA_RX_I_ILG_INTR_MAP_REG (0x0144)



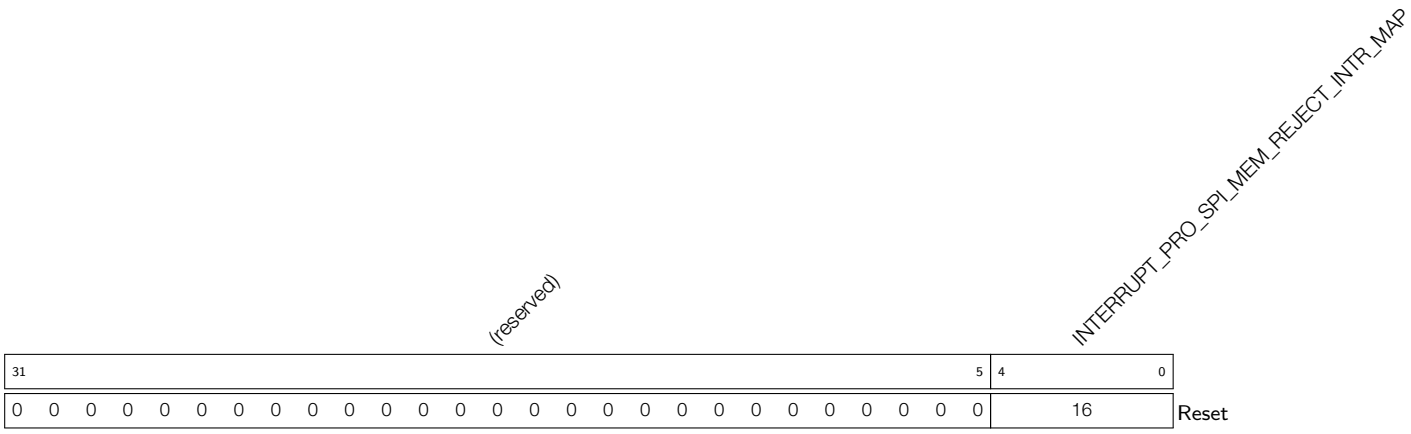
INTERRUPT_PRO_PMS_DMA_RX_I_ILG_INTR_MAP 用于将 PMS_DMA_RX_I_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.83: INTERRUPT_PRO_PMS_DMA_TX_I_ILG_INTR_MAP_REG (0x0148)



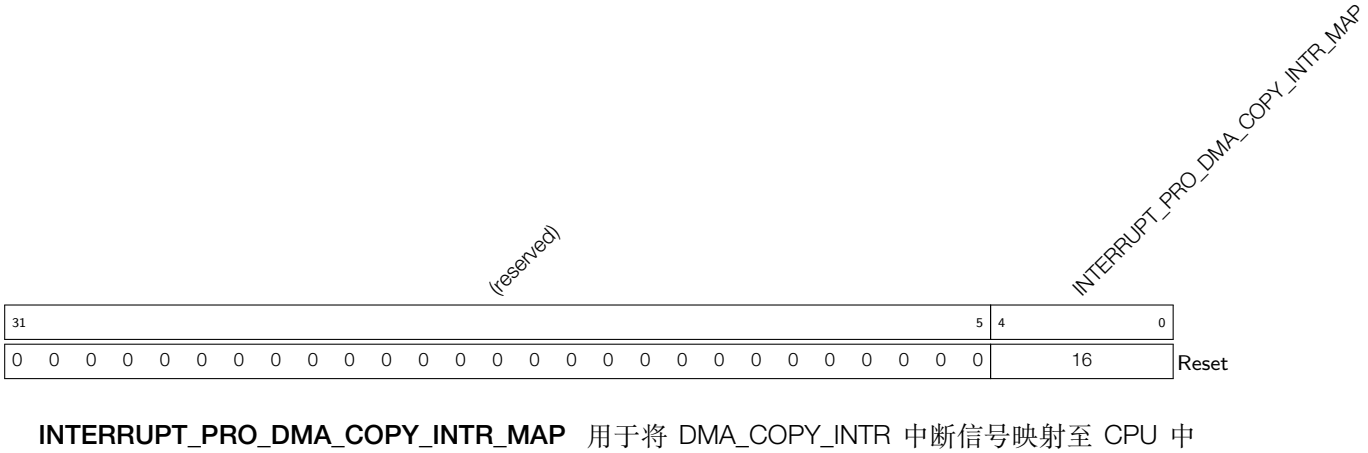
INTERRUPT_PRO_PMS_DMA_TX_I_ILG_INTR_MAP 用于将 PMS_DMA_TX_I_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.84: INTERRUPT_PRO_SPI_MEM_REJECT_INTR_MAP_REG (0x014C)



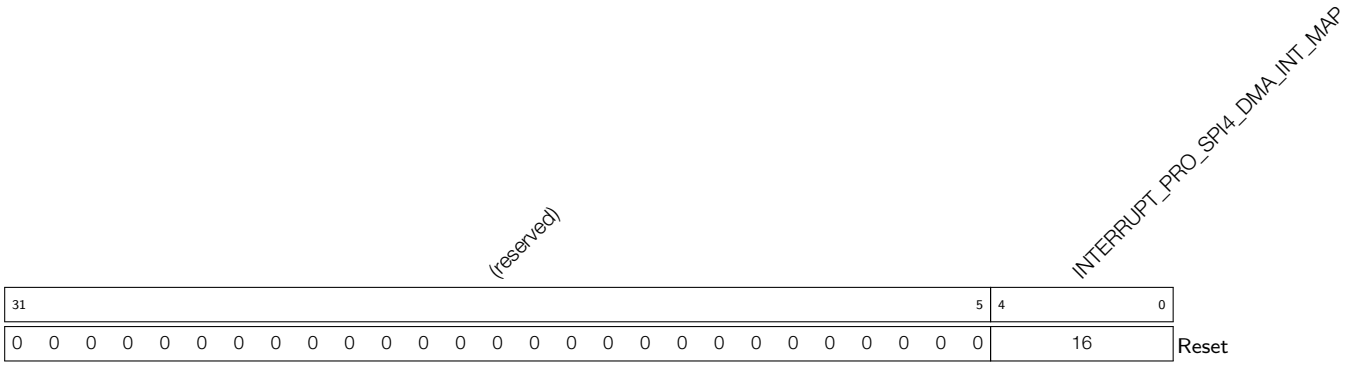
INTERRUPT_PRO_SPI_MEM_REJECT_INTR_MAP 用于将 SPI_MEM_REJECT_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.85: INTERRUPT_PRO_DMA_COPY_INTR_MAP_REG (0x0150)



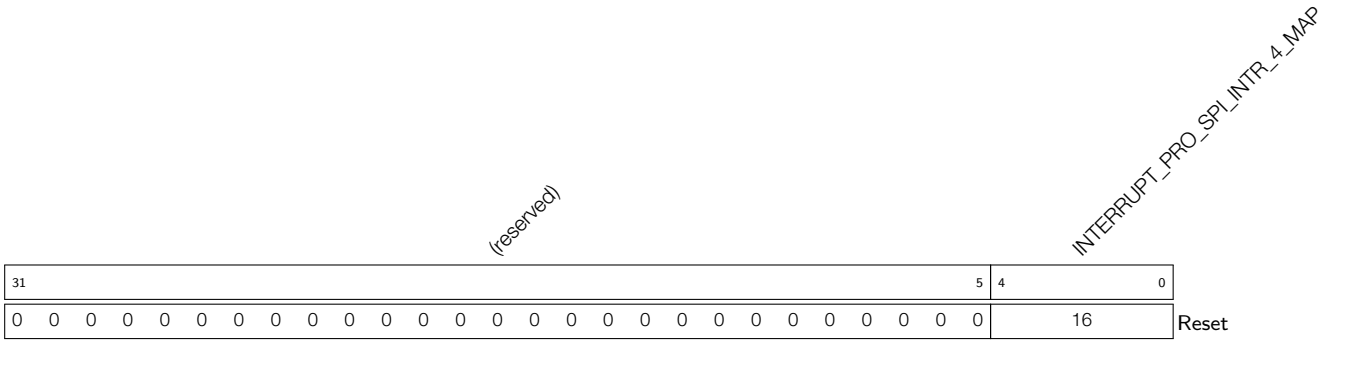
INTERRUPT_PRO_DMA_COPY_INTR_MAP 用于将 DMA_COPY_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.86: INTERRUPT_PRO_SPI4_DMA_INT_MAP_REG (0x0154)



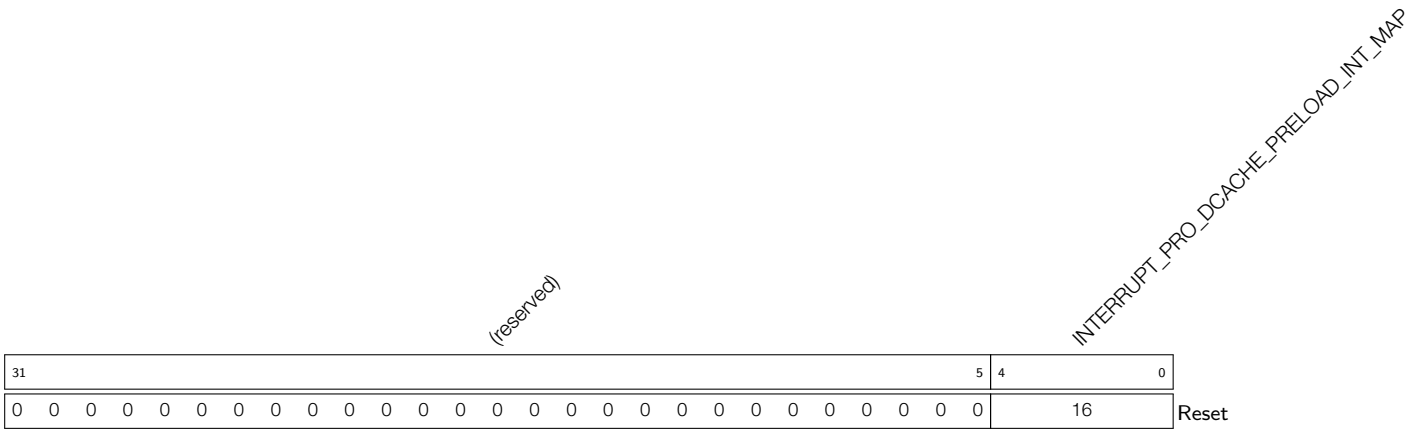
INTERRUPT_PRO_SPI4_DMA_INT_MAP 用于将 SPI4_DMA_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.87: INTERRUPT_PRO_SPI_INTR_4_MAP_REG (0x0158)



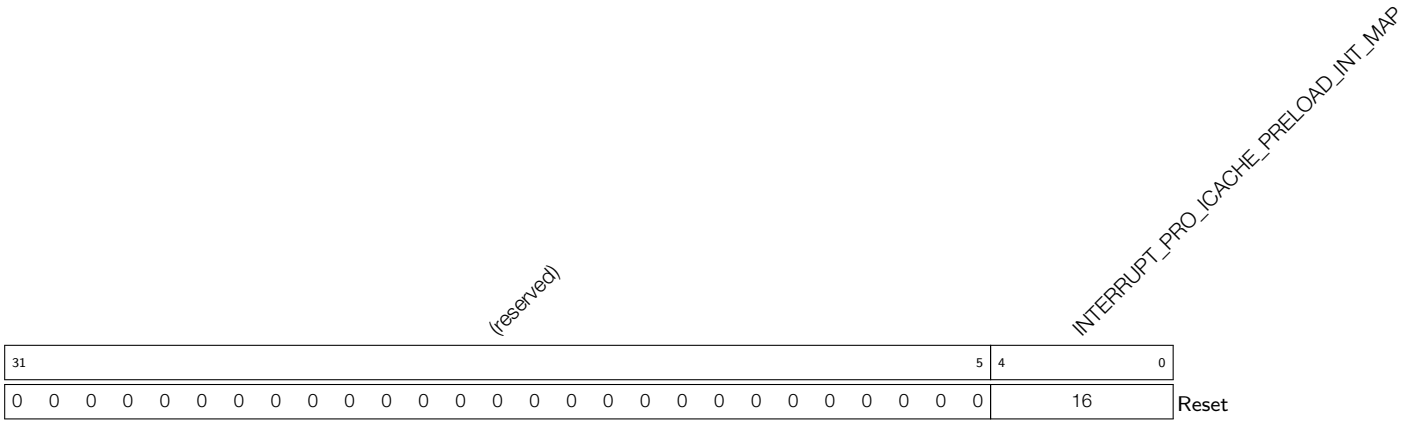
INTERRUPT_PRO_SPI_INTR_4_MAP 用于将 SPI_INTR_4 中断信号映射至 CPU 中断。(读/写)

Register 4.88: INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP_REG (0x015C)



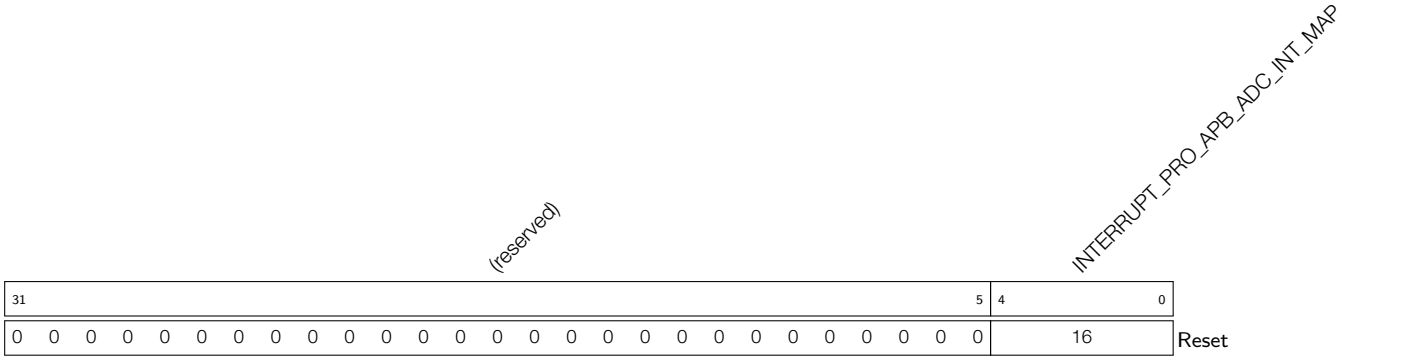
INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP 用于将 DCACHE_PRELOAD_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.89: INTERRUPT_PRO_ICACHE_PRELOAD_INT_MAP_REG (0x0160)



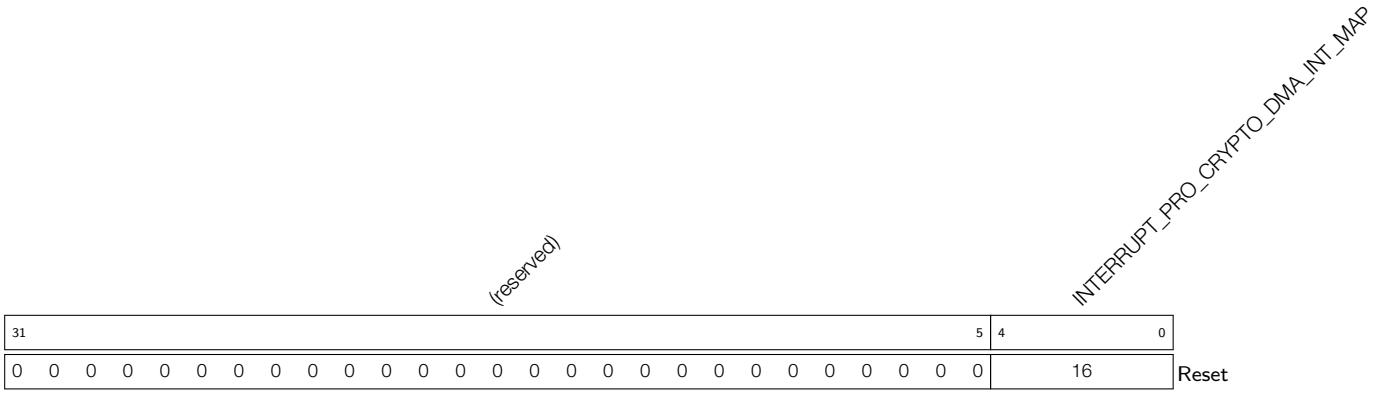
INTERRUPT_PRO_ICACHE_PRELOAD_INT_MAP 用于将 ICACHE_PRELOAD_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.90: INTERRUPT_PRO_APB_ADC_INT_MAP_REG (0x0164)



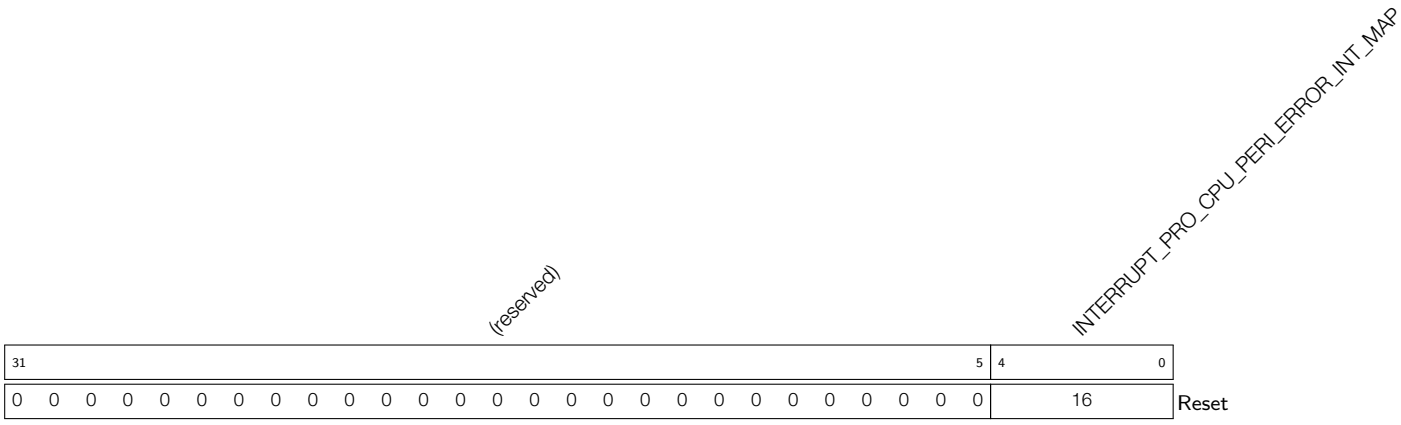
INTERRUPT_PRO_APB_ADC_INT_MAP 用于将 APB_ADC_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.91: INTERRUPT_PRO_CRYPTO_DMA_INT_MAP_REG (0x0168)



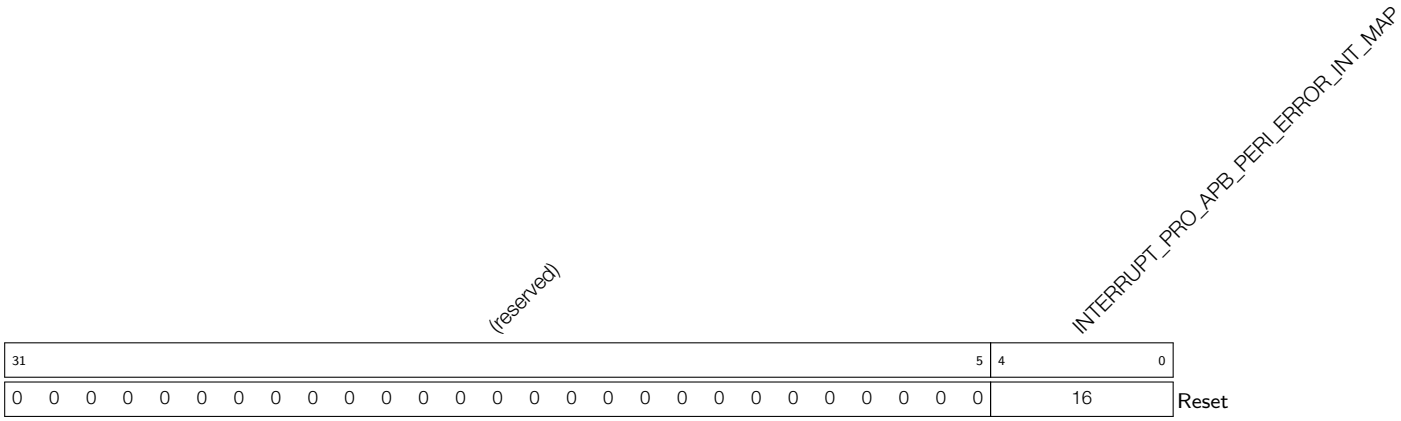
INTERRUPT_PRO_CRYPTO_DMA_INT_MAP 用于将 CRYPTO_DMA_INT 中断信号映射至 CPU 中断。（读/写）

Register 4.92: INTERRUPT_PRO_CPU_PERI_ERROR_INT_MAP_REG (0x016C)



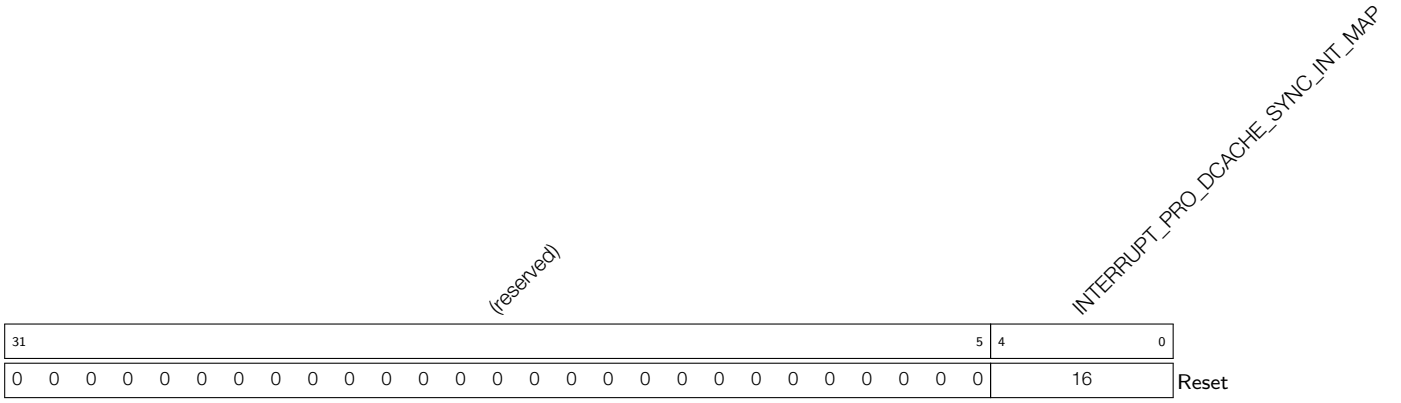
INTERRUPT_PRO_CPU_PERI_ERROR_INT_MAP 用于将 CPU_PERI_ERROR_INT 中断信号映射至 CPU 中断。（读/写）

Register 4.93: INTERRUPT_PRO_APB_PERI_ERROR_INT_MAP_REG (0x0170)



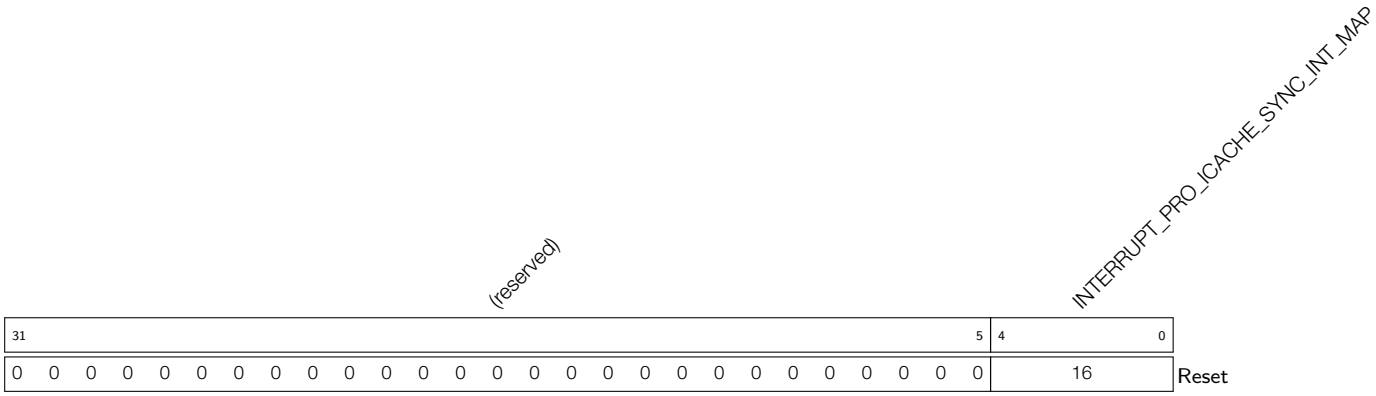
INTERRUPT_PRO_APB_PERI_ERROR_INT_MAP 用于将 APB_PERI_ERROR_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.94: INTERRUPT_PRO_DCACHE_SYNC_INT_MAP_REG (0x0174)



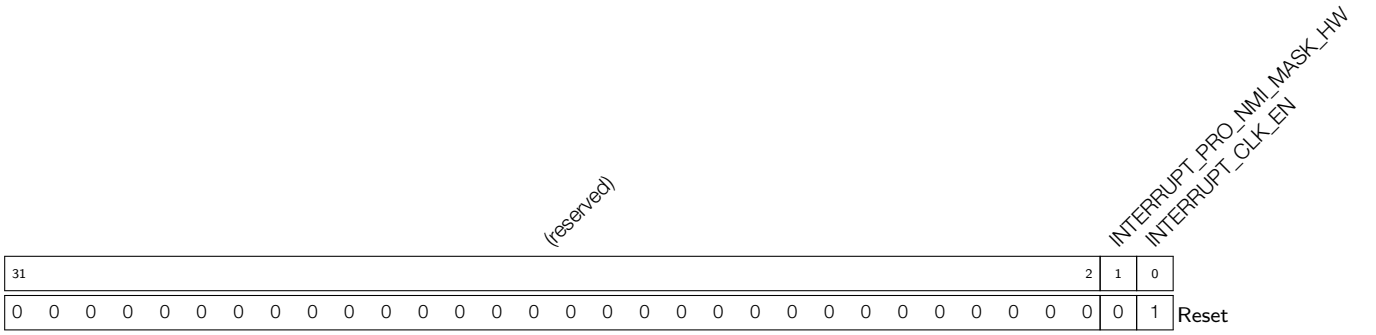
INTERRUPT_PRO_DCACHE_SYNC_INT_MAP 用于将 DCACHE_SYNC_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.95: INTERRUPT_PRO_ICACHE_SYNC_INT_MAP_REG (0x0178)



INTERRUPT_PRO_ICACHE_SYNC_INT_MAP 用于将 ICACHE_SYNC_INT 中断信号映射至 CPU 中断。(读/写)

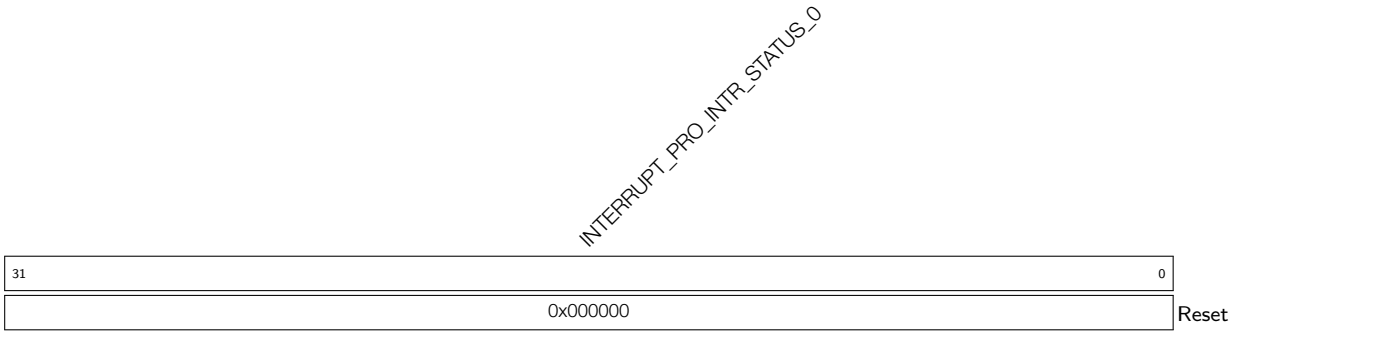
Register 4.96: INTERRUPT_CLOCK_GATE_REG (0x0188)



INTERRUPT_CLK_EN 使能或关闭中断矩阵的时钟。1：使能时钟；0：关闭时钟。(读/写)

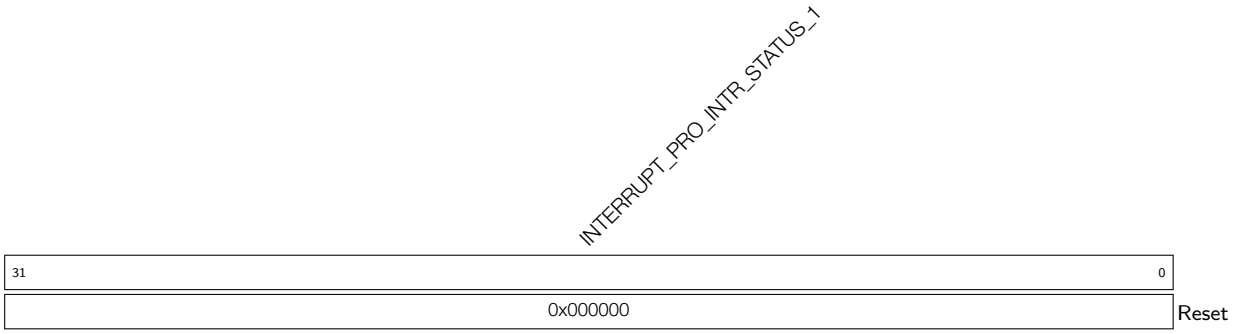
INTERRUPT_PRO_NMI_MASK_HW 屏蔽所有 NMI 中断信号映射至 CPU。(读/写)

Register 4.97: INTERRUPT_PRO_INTR_STATUS_REG_0_REG (0x017C)



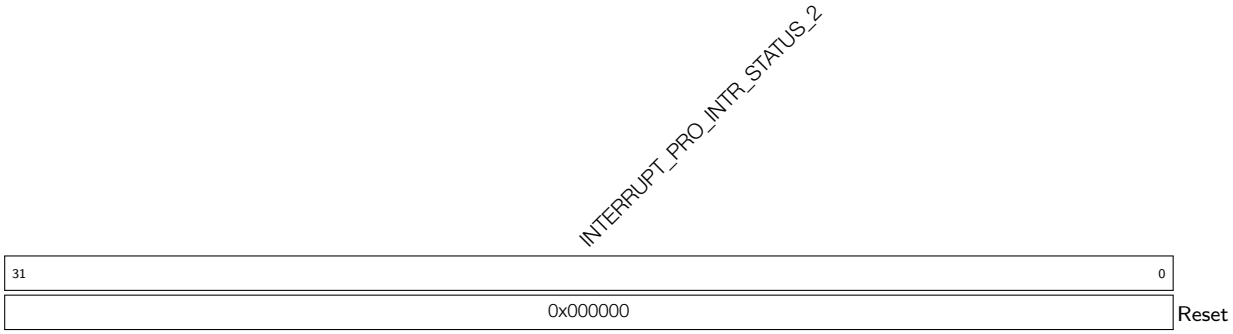
INTERRUPT_PRO_INTR_STATUS_0 用于存储前 32 个中断源的状态。(只读)

Register 4.98: INTERRUPT_PRO_INTR_STATUS_REG_1_REG (0x0180)



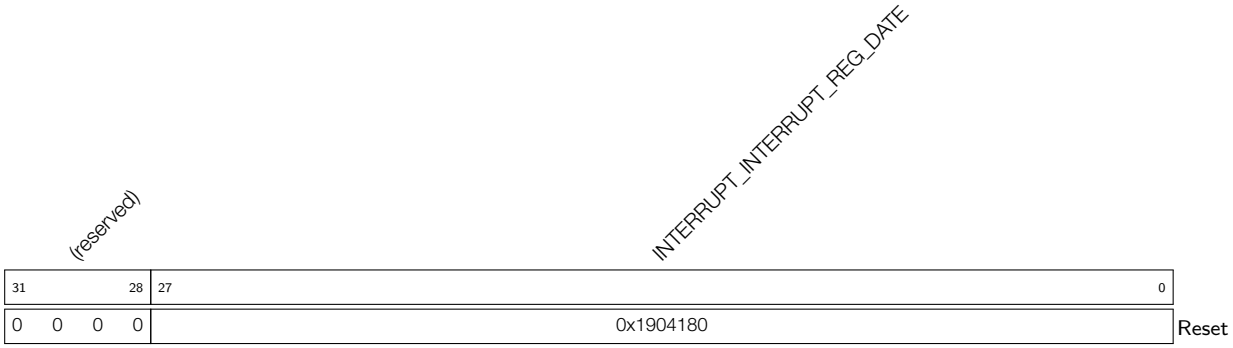
INTERRUPT_PRO_INTR_STATUS_1 用于存储中间 32 个中断源的状态。(只读)

Register 4.99: INTERRUPT_PRO_INTR_STATUS_REG_2_REG (0x0184)



INTERRUPT_PRO_INTR_STATUS_2 用于存储最后 31 个中断源的状态。(只读)

Register 4.100: INTERRUPT_REG_DATE_REG (0x0FFC)



INTERRUPT_DATE 版本控制寄存器。(读/写)

5. IO MUX 和 GPIO 交换矩阵

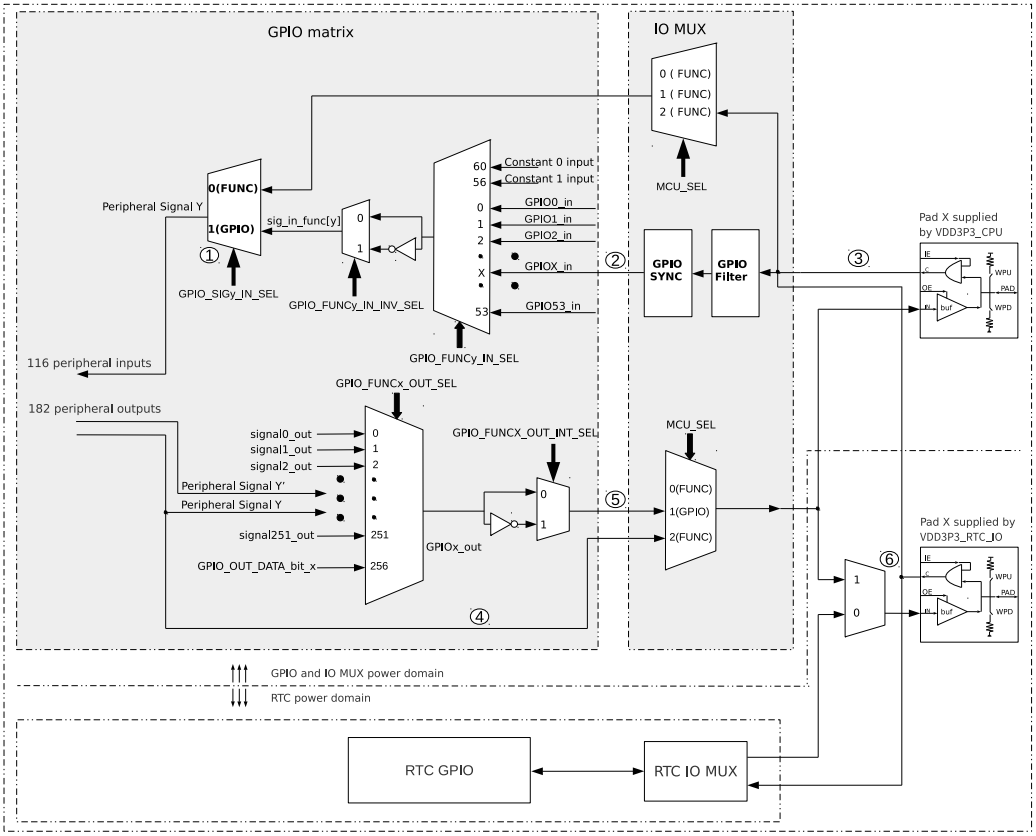
5.1 概述

ESP32-S2 芯片有 43 个物理 GPIO pad。每个 pad 都可用作一个通用 IO，或连接一个内部的外设信号。IO MUX、RTC IO MUX 和 GPIO 交换矩阵用于将信号从外设传输至 GPIO pad。这些模块共同组成了芯片的 IO 控制。

注意：这 43 个物理 GPIO pad 的序列号为：0 ~ 21, 26 ~ 46。其中 GPIO46 仅用作输入管脚，其他的既可以作为输入又可以作为输出管脚。

本章节主要描述了如何通过 GPIO 交换矩阵、IO MUX 或 RTC IO MUX 实现 43 个数字 pad（控制信号：DRV、IE、OE、WPU、WPD 等）与内部信号的选择和连接，这些内部信号包括：

- 116 个数字外设输入信号（控制信号：SIG_IN_SEL、SIG_OUT_SEL、IE、OE 等）
- 182 个数字外设输出信号（控制信号：SIG_IN_SEL、SIG_OUT_SEL、IE、OE 等）
- 快速外设输入和输出信号（控制信号：IE、OE 等）
- 22 个 RTC GPIO 信号



①Not every peripheral input has the MUX, only peripheral input signals (Y: 0~3, 7~11, 14~15, 17~18, 108~117, 127~130, 167~171) have this MUX. The other peripheral inputs only get the input from GPIO Matrix.

②There are 43 GPIO pads(X: 0~21, 26~46), therefore only 43 inputs from GPIO SYNC to GPIO Matrix.

③There are few control signals, IE, OE, WPU, WPD, connected to Pad supplied by VDD_3P3_CPU.

④Only few peripheral outputs have this path, please refer to IO MUX Pad Table in this chapter.

⑤There are only 42 outputs (GPIO pad X: 0~21, 26~45) from GPIO matrix to IO MUX.

⑥There are few control signals, IE, OE, WPU, WPD, connected to Pad supplied by VDD3P3_RTC_IO.

图 5-1. IO MUX、RTC IO MUX 和 GPIO 交换矩阵结构框图

图 5-1为 IO MUX、RTC IO MUX 和 GPIO 交换矩阵结构框图。

1. IO MUX 中每个 GPIO pad 有一个寄存器 IO_MUX_*n*_REG，每个 pad 可以配置成：
   - GPIO 功能，连接 GPIO 交换矩阵；
   - 直连功能，旁路 GPIO 交换矩阵。

   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。

   章节 5.11 列出了所有 GPIO pad 的 IO MUX 功能。
2. GPIO 交换矩阵是外设输入输出信号和管脚之间的全交换矩阵。
   - 芯片输入方向：116 个外设输入信号都可以选择任意一个 GPIO pad 的输入信号。
   - 芯片输出方向：每个 GPIO pad 的输出信号可来自 182 个外设输出信号中的任意一个。

   章节 5.10 列出了 GPIO 交换矩阵的外设信号。
3. RTC IO MUX 用于控制 GPIO pad 的低功耗和模拟功能。只有部分 GPIO pad 具有这些功能。

   章节 5.12 列出了 RTC IO MUX 功能。

5.2 通过 GPIO 交换矩阵的外设输入

5.2.1 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 43 个 GPIO（0 ~ 21, 26 ~ 46）中获取外设输入信号的索引号，见交换矩阵表格 22。并需要配置外设输入选择通过 GPIO 交换矩阵接收输入信号。

5.2.2 信号同步

如图 5-1 所示，对于信号输入，外部输入信号从 pad 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。

GPIO SYNC 模块的功能如图 5-2 所示。其中，negative sync 为 GPIO input 经过 APB clock 的下降沿同步，positive sync 为 GPIO input 经过 APB clock 上升沿同步。

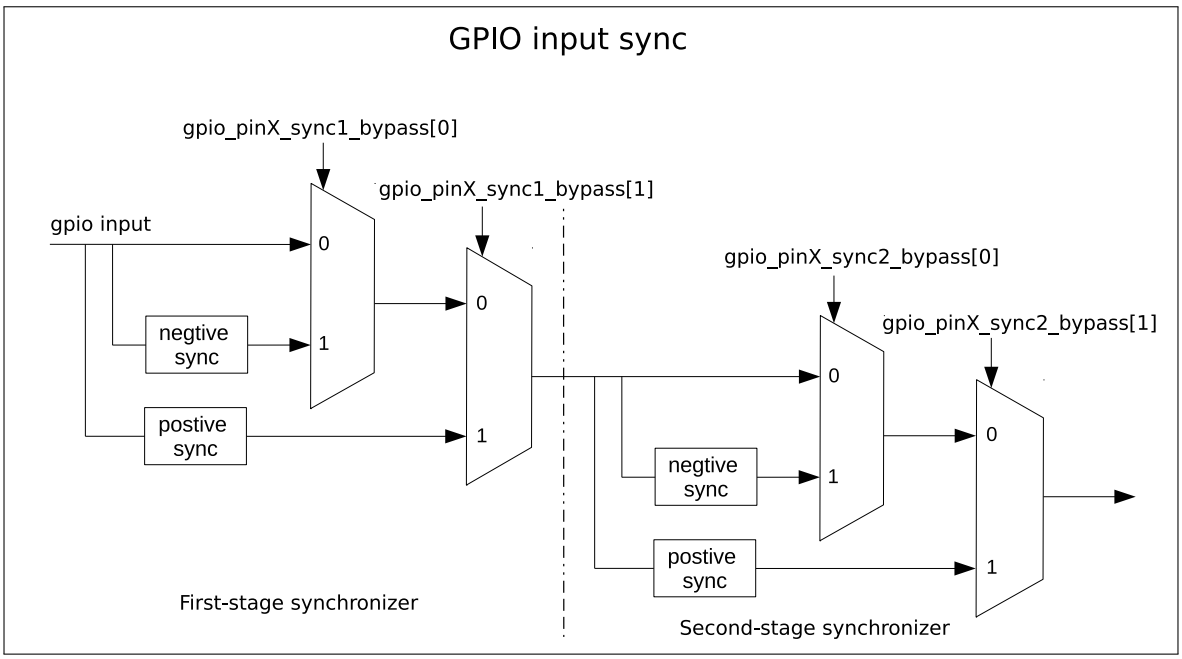


图 5-2. GPIO 输入经 clock 上升沿或下降沿同步

5.2.3 功能描述

把某个外设输入信号 *Y* 绑定到某个 GPIO pad *X* 的配置过程为：

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 为要读取的 GPIO pad *X* 的值。

   注意： 并不是所有外设信号都有有效的 GPIO_SIG*y*_IN_SEL，即有些外设信号没有图 5-1 中所示的 MUX（见图下方的注释 ①）。这些外设信号只能通过 GPIO 交换矩阵接收外部输入信号。
2. 置位寄存器 IO_MUX_FILTER_EN 使能 pad 输入信号滤波，如图 5-3 所示。只有当输入信号的有效宽度大于两个时钟周期时，输入信号才会被采样。否则，输入信号将会被滤掉。

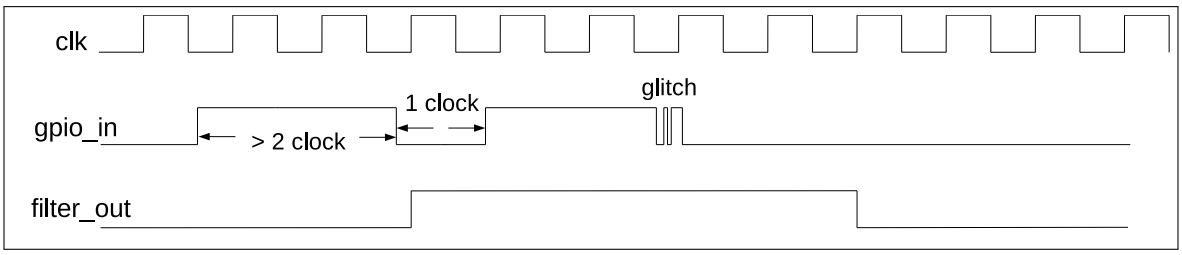


图 5-3. GPIO 输入信号滤波时序图

3. 配置 GPIO pad *X* 的 GPIO_PIN*x*_REG 来同步 GPIO input，过程如下：
    - 如图 5-2 所示，配置 GPIO_PIN*x*_SYNC1_BYPASS 使能输入信号第一拍为上升沿或者下降沿同步。
    - 如图 5-2 所示，配置 GPIO_PIN*x*_SYNC2_BYPASS 使能输入信号第二拍为上升沿或者下降沿同步。
4. 配置 GPIO pad *X* 的 IO_MUX_*x*_REG 来使能 pad 输入功能，过程如下：
    - 置位 IO_MUX_FUN_IE 使能输入；
    - 置位或清零 IO_MUX_FUN_WPU 和 IO_MUX_FUN_WPD，使能或关闭内部上拉/下拉电阻器。

例如，要把 RMT 外设通道 0 的输入信号 rmt_sig_in0（信号索引号 83）绑定到 GPIO40，请按照以下步骤操作（请注意 GPIO40 也叫做 MTDO 管脚）：

1. 置位 GPIO_FUNC83_IN_SEL_CFG_REG 寄存器的 GPIO_SIG83_IN_SEL，使能通过 GPIO 交换矩阵接收外部输入信号；
2. 将 GPIO_FUNC83_IN_SEL_CFG_REG 寄存器的 GPIO_FUNC83_IN_SEL 字段设置为 40；
3. 置位 IO_MUX_GPIO40_REG 寄存器的 IO_MUX_FUN_IE（使能输入模式）。

说明：

- 同一个输入 pad 上可以同时绑定多个内部 input_signals。
- 置位 GPIO_FUNC*y*_IN_INV_SEL 可以把输入的信号取反。
- 无需将输入信号绑定到一个 pad 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，input_signal_*X* 始终为 0；
    - 当 GPIO_FUNC*y*_IN_SEL 是 0x38 时，input_signal_*X* 始终为 1。

5.2.4 简单 GPIO 输入

GPIO_IN_REG/GPIO_IN1_REG 寄存器存储着每一个 GPIO pad 的输入值。

任意 GPIO pad 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。但是需要为 pad *X* 的 IO_MUX_*n*_REG 寄存器配置 IO_MUX_FUN_IE 位以使能输入，如章节 5.2.2 所述。

5.3 通过 GPIO 交换矩阵的外设输出

5.3.1 概述

为实现通过 GPIO 交换矩阵输出外设信号，需要配置 GPIO 交换矩阵将输出索引号为 (0 ~ 11，14 ~ 18, ...) 的外设信号输出到 42 个 GPIO (0 ~ 21, 26 ~ 45)。具体输出索引号见交换矩阵表格 22。

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 pad 为 GPIO 功能，这样输出 GPIO 信号就能连接到相应 pad。

说明：
输出索引号为 223 ~ 227 的外设信号，可配置为从一个 GPIO 管脚输入后，直接由另一个 GPIO 管脚输出。

5.3.2 功能描述

如图 5-1 所示，对于信号输出，182 个输出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 pad。

输出外设信号 *Y* 到某一 GPIO pad *X* 的步骤为：

1. 在 GPIO 交换矩阵里配置 GPIO pad *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 W1TS 或 W1TC 寄存器来更新 GPIO_ENABLE_REG 寄存器中的值：
   - 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 pad *X* 对应的 GPIO_FUNC*x*_OUT_SEL_CFG_REG 寄存器的 GPIO_FUNC*x*_OEN_SEL 字段置位；同时需要将 GPIO_ENABLE_W1TS_REG 或 GPIO_ENABLE1_W1TS_REG 中的相应字段置位。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。详细信息见表 22 中“输出信号的输出使能信号”一栏。
   - 清零 GPIO_ENABLE_W1TC_REG 或 GPIO_ENABLE1_W1TC_REG 中相应位可以关闭 GPIO pad 的输出。
2. 要选择以开漏方式输出，可以设置 GPIO pad *X* 的 GPIO_PIN*x*_REG 寄存器中的 GPIO_PIN*x*_PAD_DRIVER 位。
3. 配置 IO MUX 寄存器来选择 GPIO 交换矩阵。配置 GPIO pad *X* 的 IO_MUX_*x*_REG 的过程如下：
   - 设置功能字段 IO_MUX_MCU_SEL 为 GPIO pad *X* 的 IO MUX 功能（所有管脚的 Function 1，数值为 1）。
   - 设置 IO_MUX_FUN_DRV 字段为特定的输出强度值 (0 ~ 3)，值越大，输出驱动能力越强：
     - 0: ~5 mA
     - 1: ~10 mA
     - 2: ~20 mA（默认值）
     - 3: ~40 mA
   - 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/下拉电阻器。

说明：

- 某一个外设的输出信号可以同时从多个 pad 输出。
- GPIO46 不可用于输出信号。
- 置位 GPIO_FUNC*n*_OUT_INV_SEL 可以把输出的信号取反。

5.3.3 简单 GPIO 输出

GPIO 交换矩阵也可以用于简单 GPIO 输出，具体配置如下：

- 设置 GPIO 交换矩阵 GPIO_FUNC*n*_OUT_SEL 寄存器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] 寄存器中某一位的值为期望 GPIO 输出的值。

说明：

- GPIO_OUT_REG[0] ~ GPIO_OUT_REG[31] 对应 GPIO0 ~ GPIO31，GPIO_OUT1_REG[25:22] 无效；
- GPIO_OUT1_REG[0] ~ GPIO_OUT1_REG[13] 对应 GPIO32 ~ GPIO45，GPIO_OUT1_REG[21:14] 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 来改写 GPIO_OUT_REG 或 GPIO_OUT1_REG 中的值。

5.3.4 Sigma Delta 调制输出 (SDM)

5.3.4.1 功能描述

182 个数字外设输出中有 8 个通道支持 1 比特二阶 Sigma Delta 调制输出，这 8 个通道对应的外设输出信号索引号为 100 ~ 107，默认使能输出。该调制器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：

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

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 GPIOSD_FUNCTION_CLK_EN 使能调制器时钟；
- 配置寄存器 GPIOSD_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。

GPIOSD_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]。

- GPIOSD_SD*n*_IN = -128，调制器输出信号占空比为 0%；
- GPIOSD_SD*n*_IN = 0，调制器输出信号占空比接近 50%；
- GPIOSD_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：

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

> **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。

5.3.4.2 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 pad，见 5.3.2 章节。
- 置位 GPIOSD_FUNCTION_CLK_EN，使能 SDM 时钟。
- 配置 GPIOSD_SD*n*_PRESCALE 寄存器设置时钟分频系数。
- 配置 GPIOSD_SD*n*_IN 设置 SDM 输出信号的占空比。

5.4 专用 GPIO

5.4.1 概述

专用 GPIO 模块是为 CPU 与 GPIO 交换矩阵和 IO MUX 交互而设计的专用通道，该模块包括 8 个输入输出通道，其中输入通道对应的外设输入信号索引号为 235 ~ 242；输出通道对应的外设输出信号索引号也为 235 ~ 242，输出通道默认使能输出。

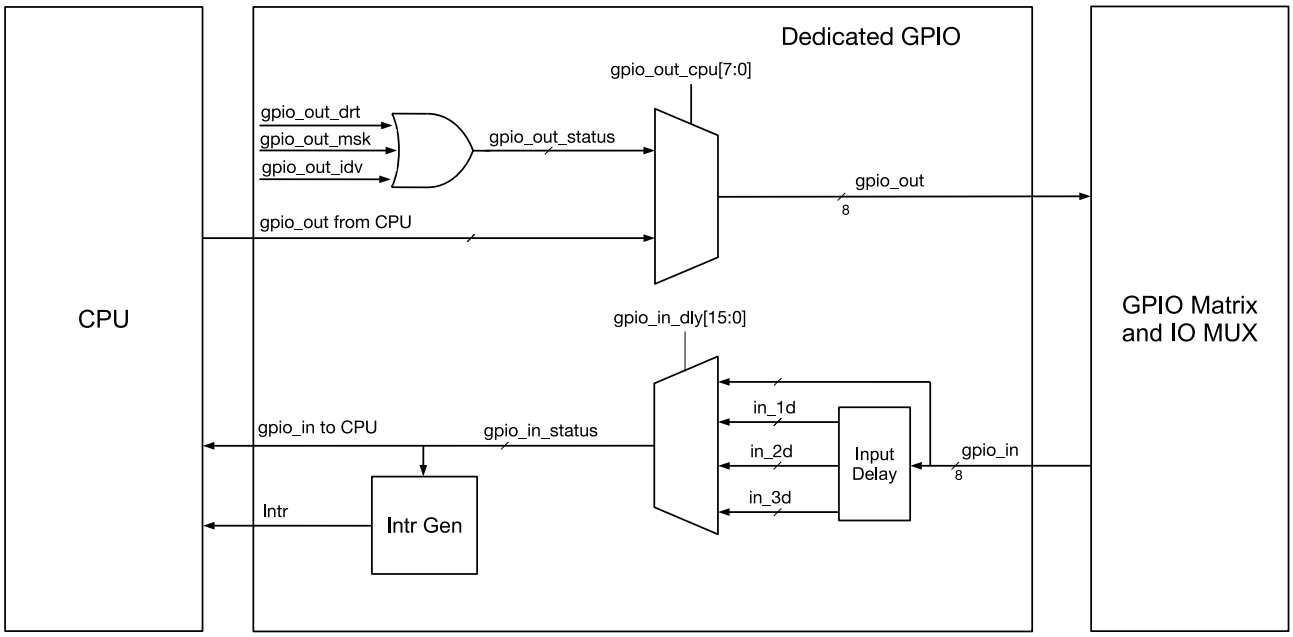


图 5-4. 专用 GPIO 模块图

图 5-4 为专用 GPIO 结构图。用户可通过置位寄存器 SYSTEM_CPU_PERI_CLK_EN_REG 中的 SYSTEM_CLK_EN_DEDICATED_GPIO 位使能专用 GPIO 模块的时钟，也可以通过先置位，再清零寄存器 SYSTEM_CPU_PERI_RST_EN_REG 中的 SYSTEM_RST_EN_DEDICATED_GPIO 位复位专用 GPIO 模块。详细信息可参考章节 6 *系统寄存器* 中表 33 *外设时钟门控与复位控制位*。

5.4.2 主要特性

专用 GPIO 模块具备以下特性：

- 8 个输出通道和 8 个输入通道
- 每个通道均可通过寄存器或 CPU 指令访问
- 输入通道延迟时间可配置

- 输入通道支持中断

5.4.3 功能描述

用户可通过寄存器访问专用 GPIO，或直接调用特定的 CPU 指令访问专用 GPIO。

如图 5-4 所示，对于输出通道，专用 GPIO 模块提供两种方式来驱动 GPIO。用户可通过配置寄存器 DEDIC_GPIO_OUT_CPU_REG 中相应位选择具体的 GPIO 输出驱动方式：

- DEDIC_GPIO_OUT_CPU_SEL*n* = 0，则通过配置寄存器的方式来驱动 GPIO 的输出；
- DEDIC_GPIO_OUT_CPU_SEL*n* = 1，则通过 CPU 指令来驱动 GPIO 的输出。

对于输入通道，专用 GPIO 模块也提供两种方式来获取专用 GPIO 输入值：

- 通过寄存器查询专用 GPIO 输入值；
- 通过 CPU 指令读取专用 GPIO 的输入值。

5.4.3.1 通过寄存器访问专用 GPIO

寄存器可通过下面三种方式驱动 GPIO 输出：

- 直接写 GPIO 输出值，通过配置寄存器 DEDIC_GPIO_OUT_DRT_REG 实现；
- MASK 写 GPIO 输出值，通过配置寄存器 DEDIC_GPIO_OUT_MSK_REG 实现；
- 单独 BIT 写 GPIO 输出值，通过配置寄存器 DEDIC_GPIO_OUT_IDV_REG 实现。

软件可通过寄存器 DEDIC_GPIO_OUT_SCAN_REG 查询 GPIO 的状态，即为图 5-4 中的 gpio_out_status。

软件可以通过读寄存器 DEDIC_GPIO_IN_SCAN_REG 获取专用 GPIO 输入值，即为图 5-4 中的 gpio_in_status。

专用 GPIO 模块支持对输入信号零延时及一个时钟/两个时钟/三个时钟周期的延时，可通过配置寄存器 DEDIC_GPIO_IN_DLY_REG 来控制各个通道的延时。专用 GPIO 模块支持可配置的中断方式来指示 GPIO 的输入状态。配置寄存器 DEDIC_GPIO_INTR_RCGN_REG 设置输入信号中断产生方式，包括：

- 0/1：无中断
- 2：低电平触发
- 3：高电平触发
- 4：下降沿触发
- 5：上升沿触发
- 6/7：沿触发

5.4.3.2 通过 CPU 指令访问专用 GPIO

CPU 也可通过指令来读/写专用 GPIO，共包括六条指令：

- 置位输出通道中相应位
  汇编语法：SET_BIT_GPIO_OUT mask

功能为写 1 置位用户寄存器 GPIO_OUT 对应的位，mask 的位宽为 8。即 GPIO_OUT 中某些位将被置位，这些位即为 mask 中为 1 的位，其余位则保持不变。GPIO_OUT 对应图 5-4 中的 gpio_out。

- 清除输出通道中相应位
  汇编语法：CLR_BIT_GPIO_OUT mask
  功能为写 1 清零用户寄存器 GPIO_OUT 对应的位，mask 的位宽为 8。即 GPIO_OUT 中某些位将被清零，这些位即为 mask 中为 1 的位，其余位则保持不变。

- 以 MASK 方式置位或清除输出通道中相应位
  汇编语法：WR_MASK_GPIO_OUT value, mask
  功能为按 mask 方式向用户寄存器 GPIO_OUT 写入 value。value 位宽为 8，表示将要写入的值；mask 位宽为 8，表示需要操作的 GPIO_OUT。只有 mask 中为 1 的位被更新，即更新为 value 的值。如 mask 0x03 (0000 0011), 表示写入值只对 GPIO_OUT[0] 与 GPIO_OUT[1] 有效。

- 写地址寄存器 art 的值到输出通道
  汇编语法：WUR.GPIO_OUT art
  功能为写地址寄存器 art 的值到用户寄存器 GPIO_OUT。寄存器 art 的位宽为 32，使用该指令时只有低 8 位有效。

- 读输出通道的值到 arr 寄存器
  汇编语法：RUR.GPIO_OUT arr
  功能为读用户寄存器 GPIO_OUT 的值到地址寄存器 arr。寄存器 arr 的位宽为 32，使用该指令时只有低 8 位有效。

- 读输入通道的值至地址寄存器 l
  汇编语法：GET_GPIO_IN l
  功能为读用户寄存器 GPIO_IN 的值到地址寄存器 l。寄存器 l 的位宽为 32, 其高 24 位为 0，低 8 位对应用户寄存器 GPIO_IN 的值。

5.5 IO MUX 的直接 I/O 功能

5.5.1 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 pad 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。

5.5.2 功能描述

对于外设输入信号，I/O 旁路 GPIO 交换矩阵必须配置两个寄存器：

1. GPIO pad 的 IO_MUX_MCU_SEL 必须设置为相应的 pad 功能，章节 5.11 列出了 pad 功能。
2. 对于输入信号，必须将 GPIO_SIG*n*_IN_SEL 寄存器设置为低电平，则输入信号直接连接到外设。

对于外设输出信号，I/O 旁路 GPIO 交换矩阵只需将 GPIO pad 的 IO_MUX_MCU_SEL 配置为相应的 pad 功能即可。

说明：

- 对于外设输入信号，并不是所有的外部输入信号都可以通过 IO MUX 连接到外设输入。某些外部输入信号只能通过 GPIO 交换矩阵连接到外设输入。
- 对于外设输出信号，并不是所有的外设输出都可以直接通过 IO MUX 连接到 pad。某些外设输出只能通过 GPIO 交换矩阵连接到 pad。

5.6 RTC IO MUX 的低功耗和模拟 I/O 功能

5.6.1 概述

ESP32-S2 中有 22 个 GPIO 管脚具有低功耗（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 I/O 指向 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。

章节 5.12 列出了 RTC_MUX 管脚和功能。

5.6.2 功能描述

每个 pad 的模拟和 RTC 功能是由 RTCIO_TOUCH_PAD*n*_REG 寄存器中的 RTCIO_TOUCH_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 RTCIO_TOUCH_PAD*n*_MUX_SEL，则输入输出信号会经过 RTC 子系统。在这种模式下，RTCIO_TOUCH_PAD*n*_REG 寄存器用于数字 I/O，pad 的模拟功能也可以实现。章节 5.12 列出了 RTC 管脚的功能。

表 5.12 列出了 GPIO pad 与相应的 RTC 管脚和模拟功能的映射关系。请注意 RTCIO_TOUCH_PAD*n*_REG 寄存器使用的是 RTC GPIO 管脚的序号，不是 GPIO pad 的序号。

5.7 Light-sleep 模式管脚功能

当 ESP32-S2 处于 Light-sleep 模式时管脚可以有不同的功能。如果某一 GPIO pad 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 pad。

表 21: IO MUX Light-sleep 管脚功能寄存器

IO MUX 功能	正常工作模式 或者 IO_MUX_SLP_SEL = 0	Light-sleep 模式 并且 IO_MUX_SLP_SEL = 1
输出驱动强度	IO_MUX_FUN_DRV	IO_MUX_FUN_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	(From GPIO Matrix _OEN field)[1]	IO_MUX_MCU_OE

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。

说明：
正常工作模式或者 IO_MUX_SLP_SEL = 0 时，输出使能配置请参考 5.3.2 章节。

5.8 Pad Hold 特性

每个 IO pad（包括 RTC pad）都有单独的 hold 功能，由 RTC 寄存器控制。pad 的 hold 功能被置上后，pad 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 pad 的状态。应用如果希望在看门狗超时触发内核复位和系统复位时或者 Deep-sleep 时 pad 的状态不被改变，就需要提前把 hold 置上。

说明：

- 对于数字 pad 而言，若要在深度睡眠掉电之后保持 pad 输入输出的状态值，需要在掉电之前把寄存器 RTC_CNTL_DG_PAD_FORCE_UNHOLD 设置成 0。对于 RTC pad 而言，pad 的输入输出值，由寄存器 RTC_CNTL_PAD_HOLD_REG 中相应的位来控制 Hold 和 Unhold pad 的值。
- 在芯片被唤醒之后，若要关闭 Hold 功能，将寄存器 RTC_CNTL_DG_PAD_FORCE_UNHOLD 设置成 1。若想继续保持 pad 的值，可把 RTC_CNTL_PAD_HOLD_REG 寄存器中相应的位设置成 1。

5.9 I/O Pad 供电

IO pad 供电请参考 《ESP32-S2 技术规格书》 中管脚定义章节。

5.9.1 电源管理

ESP32-S2 的数字管脚可分为如下 4 种不同的电源域。

- VDD3P3_RTC_IO：RTC 和 CPU 的输入电源
- VDD3P3_CPU：CPU 的输入电源
- VDD3P3_RTC：RTC 模拟的输入电源
- 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_IO 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-S2 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。

5.10 外设信号列表

表 22 列出了 GPIO 交换矩阵的外设输入输出信号。

表 22: GPIO 交换矩阵

信号索引	输入信号	默认值 *	信号可经由 IO MUX 输出	输出信号	输出信号的输出使能信号
0	SPIQ_in	0	yes	SPIQ_out	SPIQ_oe
1	SPID_in	0	yes	SPID_out	SPID_oe
2	SPIHD_in	0	yes	SPIHD_out	SPIHD_oe
3	SPIWP_in	0	yes	SPIWP_out	SPIWP_oe
4	-	-	-	SPICLK_out_mux	SPICLK_oe
5	-	-	-	SPICS0_out	SPICS0_oe
6	-	-	-	SPICS1_out	SPICS1_oe
7	SPID4_in	0	yes	SPID4_out	SPID4_oe

信号索引	输入信号	默认值 *	信号可经由 IO MUX 输出	输出信号	输出信号的输出使能信号
8	SPID5_in	0	yes	SPID5_out	SPID5_oe
9	SPID6_in	0	yes	SPID6_out	SPID6_oe
10	SPID7_in	0	yes	SPID7_out	SPID7_oe
11	SPIDQS_in	0	yes	SPIDQS_out	SPIDQS_oe
14	U0RXD_in	0	yes	U0TXD_out	1'd1
15	U0CTS_in	0	yes	U0RTS_out	1'd1
16	U0DSR_in	0	no	U0DTR_out	1'd1
17	U1RXD_in	0	yes	U1TXD_out	1'd1
18	U1CTS_in	0	yes	U1RTS_out	1'd1
21	U1DSR_in	0	no	U1DTR_out	1'd1
23	I2S0O_BCK_in	0	no	I2S0O_BCK_out	1'd1
25	I2S0O_WS_in	0	no	I2S0O_WS_out	1'd1
27	I2S0I_BCK_in	0	no	I2S0I_BCK_out	1'd1
28	I2S0I_WS_in	0	no	I2S0I_WS_out	1'd1
29	I2CEXT0_SCL_in	1	no	I2CEXT0_SCL_out	I2CEXT0_SCL_oe
30	I2CEXT0_SDA_in	1	no	I2CEXT0_SDA_out	I2CEXT0_SDA_oe
39	pcnt_sig_ch0_in0	0	no	gpio_wlan_prio	1'd1
40	pcnt_sig_ch1_in0	0	no	gpio_wlan_active	1'd1
41	pcnt_ctrl_ch0_in0	0	no	-	1'd1
42	pcnt_ctrl_ch1_in0	0	no	-	1'd1
43	pcnt_sig_ch0_in1	0	no	-	1'd1
44	pcnt_sig_ch1_in1	0	no	-	1'd1
45	pcnt_ctrl_ch0_in1	0	no	-	1'd1
46	pcnt_ctrl_ch1_in1	0	no	-	1'd1
47	pcnt_sig_ch0_in2	0	no	-	1'd1
48	pcnt_sig_ch1_in2	0	no	-	1'd1
49	pcnt_ctrl_ch0_in2	0	no	-	1'd1
50	pcnt_ctrl_ch1_in2	0	no	-	1'd1
51	pcnt_sig_ch0_in3	0	no	-	1'd1
52	pcnt_sig_ch1_in3	0	no	-	1'd1
53	pcnt_ctrl_ch0_in3	0	no	-	1'd1
54	pcnt_ctrl_ch1_in3	0	no	-	1'd1
64	usb_otg_iddig_in	0	no	-	1'd1
65	usb_otg_avalid_in	0	no	-	1'd1
66	usb_srp_bvalid_in	0	no	usb_otg_idpullup	1'd1
67	usb_otg_vbusvalid_in	0	no	usb_otg_dppulldown	1'd1
68	usb_srp_sessend_in	0	no	usb_otg_dmpulldown	1'd1
69	-	-	-	usb_otg_drvvbus	1'd1
70	-	-	-	usb_srp_chrgvbus	1'd1
71	-	-	-	usb_srp_dischrgvbus	1'd1
72	SPI3_CLK_in	0	no	SPI3_CLK_out_mux	SPI3_CLK_oe
73	SPI3_Q_in	0	no	SPI3_Q_out	SPI3_Q_oe

信号索引	输入信号	默认值 *	信号可经由 IO MUX 输出	输出信号	输出信号的输出使能信号
74	SPI3_D_in	0	no	SPI3_D_out	SPI3_D_oe
75	SPI3_HD_in	0	no	SPI3_HD_out	SPI3_HD_oe
76	SPI3_CS0_in	0	no	SPI3_CS0_out	SPI3_CS0_oe
77	-	-	-	SPI3_CS1_out	SPI3_CS1_oe
78	-	-	-	SPI3_CS2_out	SPI3_CS2_oe
79	-	-	-	ledc_ls_sig_out0	1'd1
80	-	-	-	ledc_ls_sig_out1	1'd1
81	-	-	-	ledc_ls_sig_out2	1'd1
82	-	-	-	ledc_ls_sig_out3	1'd1
83	rmt_sig_in0	0	no	ledc_ls_sig_out4	1'd1
84	rmt_sig_in1	0	no	ledc_ls_sig_out5	1'd1
85	rmt_sig_in2	0	no	ledc_ls_sig_out6	1'd1
86	rmt_sig_in3	0	no	ledc_ls_sig_out7	1'd1
87	-	-	-	rmt_sig_out0	1'd1
88	-	-	-	rmt_sig_out1	1'd1
89	-	-	-	rmt_sig_out2	1'd1
90	-	-	-	rmt_sig_out3	1'd1
95	I2CEXT1_SCL_in	1	no	I2CEXT1_SCL_out	I2CEXT1_SCL_oe
96	I2CEXT1_SDA_in	1	no	I2CEXT1_SDA_out	I2CEXT1_SDA_oe
100	-	-	-	gpio_sd0_out	1'd1
101	-	-	-	gpio_sd1_out	1'd1
102	-	-	-	gpio_sd2_out	1'd1
103	-	-	-	gpio_sd3_out	1'd1
104	-	-	-	gpio_sd4_out	1'd1
105	-	-	-	gpio_sd5_out	1'd1
106	-	-	-	gpio_sd6_out	1'd1
107	-	-	-	gpio_sd7_out	1'd1
108	FSPICLK_in	0	yes	FSPICLK_out_mux	FSPICLK_oe
109	FSPIQ_in	0	yes	FSPIQ_out	FSPIQ_oe
110	FSPID_in	0	yes	FSPID_out	FSPID_oe
111	FSPIHD_in	0	yes	FSPIHD_out	FSPIHD_oe
112	FSPIWP_in	0	yes	FSPIWP_out	FSPIWP_oe
113	FSPIIO4_in	0	yes	FSPIIO4_out	FSPIIO4_oe
114	FSPIIO5_in	0	yes	FSPIIO5_out	FSPIIO5_oe
115	FSPIIO6_in	0	yes	FSPIIO6_out	FSPIIO6_oe
116	FSPIIO7_in	0	yes	FSPIIO7_out	FSPIIO7_oe
117	FSPICS0_in	0	yes	FSPICS0_out	FSPICS0_oe
118	-	-	-	FSPICS1_out	FSPICS1_oe
119	-	-	-	FSPICS2_out	FSPICS2_oe
120	-	-	-	FSPICS3_out	FSPICS3_oe
121	-	-	-	FSPICS4_out	FSPICS4_oe
122	-	-	-	FSPICS5_out	FSPICS5_oe

信号索引	输入信号	默认值 *	信号可经由 IO MUX 输出	输出信号	输出信号的输出使能信号
123	can_rx	1	no	can_tx	1'd1
124	-	-	-	can_bus_off_on	1'd1
125	-	-	-	can_clkout	1'd1
126	-	-	-	SUBSPICLK_out_mux	SUBSPICLK_oe
127	SUBSPIQ_in	0	yes	SUBSPIQ_out	SUBSPIQ_oe
128	SUBSPID_in	0	yes	SUBSPID_out	SUBSPID_oe
129	SUBSPIHD_in	0	yes	SUBSPIHD_out	SUBSPIHD_oe
130	SUBSPIWP_in	0	yes	SUBSPIWP_out	SUBSPIWP_oe
131	-	-	-	SUBSPICS0_out	SUBSPICS0_oe
132	-	-	-	SUBSPICS1_out	SUBSPICS1_oe
133	-	-	-	FSPIDQS_out	FSPIDQS_oe
134	-	-	-	FSPI_HSYNC_out	FSPI_HSYNC_oe
135	-	-	-	FSPI_VSYNC_out	FSPI_VSYNC_oe
136	-	-	-	FSPI_DE_out	FSPI_DE_oe
137	-	-	-	FSPICD_out	FSPICD_oe
139	-	-	-	SPI3_CD_out	SPI3_CD_oe
140	-	-	-	SPI3_DQS_out	SPI3_DQS_oe
143	I2S0I_DATA_in0	0	no	I2S0O_DATA_out0	1'd1
144	I2S0I_DATA_in1	0	no	I2S0O_DATA_out1	1'd1
145	I2S0I_DATA_in2	0	no	I2S0O_DATA_out2	1'd1
146	I2S0I_DATA_in3	0	no	I2S0O_DATA_out3	1'd1
147	I2S0I_DATA_in4	0	no	I2S0O_DATA_out4	1'd1
148	I2S0I_DATA_in5	0	no	I2S0O_DATA_out5	1'd1
149	I2S0I_DATA_in6	0	no	I2S0O_DATA_out6	1'd1
150	I2S0I_DATA_in7	0	no	I2S0O_DATA_out7	1'd1
151	I2S0I_DATA_in8	0	no	I2S0O_DATA_out8	1'd1
152	I2S0I_DATA_in9	0	no	I2S0O_DATA_out9	1'd1
153	I2S0I_DATA_in10	0	no	I2S0O_DATA_out10	1'd1
154	I2S0I_DATA_in11	0	no	I2S0O_DATA_out11	1'd1
155	I2S0I_DATA_in12	0	no	I2S0O_DATA_out12	1'd1
156	I2S0I_DATA_in13	0	no	I2S0O_DATA_out13	1'd1
157	I2S0I_DATA_in14	0	no	I2S0O_DATA_out14	1'd1
158	I2S0I_DATA_in15	0	no	I2S0O_DATA_out15	1'd1
159	-	-	-	I2S0O_DATA_out16	1'd1
160	-	-	-	I2S0O_DATA_out17	1'd1
161	-	-	-	I2S0O_DATA_out18	1'd1
162	-	-	-	I2S0O_DATA_out19	1'd1
163	-	-	-	I2S0O_DATA_out20	1'd1
164	-	-	-	I2S0O_DATA_out21	1'd1
165	-	-	-	I2S0O_DATA_out22	1'd1
166	-	-	-	I2S0O_DATA_out23	1'd1
167	SUBSPID4_in	0	yes	SUBSPID4_out	SUBSPID4_oe

信号索引	输入信号	默认值 *	信号可经由 IO MUX 输出	输出信号	输出信号的输出使能信号
168	SUBSPID5_in	0	yes	SUBSPID5_out	SUBSPID5_oe
169	SUBSPID6_in	0	yes	SUBSPID6_out	SUBSPID6_oe
170	SUBSPID7_in	0	yes	SUBSPID7_out	SUBSPID7_oe
171	SUBSPIDQS_in	0	yes	SUBSPIDQS_out	SUBSPIDQS_oe
193	I2S0I_H_SYNC	0	no	-	1'd1
194	I2S0I_V_SYNC	0	no	-	1'd1
195	I2S0I_H_ENABLE	0	no	-	1'd1
215	-	-	-	ant_sel0	1'd1
216	-	-	-	ant_sel1	1'd1
217	-	-	-	ant_sel2	1'd1
218	-	-	-	ant_sel3	1'd1
219	-	-	-	ant_sel4	1'd1
220	-	-	-	ant_sel5	1'd1
221	-	-	-	ant_sel6	1'd1
222	-	-	-	ant_sel7	1'd1
223	sig_in_func_223	0	no	sig_in_func223	1'd1
224	sig_in_func_224	0	no	sig_in_func224	1'd1
225	sig_in_func_225	0	no	sig_in_func225	1'd1
226	sig_in_func_226	0	no	sig_in_func226	1'd1
227	sig_in_func_227	0	no	sig_in_func227	1'd1
235	pro_alonegpio_in0	0	no	pro_alonegpio_out0	1'd1
236	pro_alonegpio_in1	0	no	pro_alonegpio_out1	1'd1
237	pro_alonegpio_in2	0	no	pro_alonegpio_out2	1'd1
238	pro_alonegpio_in3	0	no	pro_alonegpio_out3	1'd1
239	pro_alonegpio_in4	0	no	pro_alonegpio_out4	1'd1
240	pro_alonegpio_in5	0	no	pro_alonegpio_out5	1'd1
241	pro_alonegpio_in6	0	no	pro_alonegpio_out6	1'd1
242	pro_alonegpio_in7	0	no	pro_alonegpio_out7	1'd1
251	-	-	-	clk_i2s_mux	1'd1

5.11 IO MUX Pad 列表

表 23 列出了每个 I/O pad 的 IO MUX 功能。

表 23: IO MUX Pad 列表

GPIO	Pad Name	Function 0	Function 1	Function 2	Function 3	Function 4	Reset	Notes
0	GPIO0	GPIO0	GPIO0	-	-	-	3	R
1	GPIO1	GPIO1	GPIO1	-	-	-	1	R
2	GPIO2	GPIO2	GPIO2	-	-	-	1	R
3	GPIO3	GPIO3	GPIO3	-	-	-	0	R
4	GPIO4	GPIO4	GPIO4	-	-	-	0	R
5	GPIO5	GPIO5	GPIO5	-	-	-	0	R
6	GPIO6	GPIO6	GPIO6	-	-	-	0	R

GPIO	Pad Name	Function 0	Function 1	Function 2	Function 3	Function 4	Reset	Notes
7	GPIO7	GPIO7	GPIO7	-	-	-	0	R
8	GPIO8	GPIO8	GPIO8	-	SUBSPICS1	-	0	R
9	GPIO9	GPIO9	GPIO9	-	SUBSPIHD	FSPIHD	1	R
10	GPIO10	GPIO10	GPIO10	FSPIIO4	SUBSPICS0	FSPICS0	1	R
11	GPIO11	GPIO11	GPIO11	FSPIIO5	SUBSPID	FSPID	1	R
12	GPIO12	GPIO12	GPIO12	FSPIIO6	SUBSPICLK	FSPICLK	1	R
13	GPIO13	GPIO13	GPIO13	FSPIIO7	SUBSPIQ	FSPIQ	1	R
14	GPIO14	GPIO14	GPIO14	FSPIDQS	SUBSPIWP	FSPIWP	1	R
15	XTAL_32K_P	XTAL_32K_P	GPIO15	U0RTS	-	-	0	R
16	XTAL_32K_N	XTAL_32K_N	GPIO16	U0CTS	-	-	0	R
17	DAC_1	DAC_1	GPIO17	U1TXD	-	-	1	R
18	DAC_2	DAC_2	GPIO18	U1RXD	CLK_OUT3	-	1	R
19	GPIO19	GPIO19	GPIO19	U1RTS	CLK_OUT2	-	0	R
20	GPIO20	GPIO20	GPIO20	U1CTS	CLK_OUT1	-	0	R
21	GPIO21	GPIO21	GPIO21	-	-	-	0	R
26	SPICS1	SPICS1	GPIO26	-	-	-	3	-
27	SPIHD	SPIHD	GPIO27	-	-	-	3	-
28	SPIWP	SPIWP	GPIO28	-	-	-	3	-
29	SPICS0	SPICS0	GPIO29	-	-	-	3	-
30	SPICLK	SPICLK	GPIO30	-	-	-	3	-
31	SPIQ	SPIQ	GPIO31	-	-	-	3	-
32	SPID	SPID	GPIO32	-		-	3	-
33	GPIO33	GPIO33	GPIO33	FSPIHD	SUBSPIHD	SPIIO4	1	-
34	GPIO34	GPIO34	GPIO34	FSPICS0	SUBSPICS0	SPIIO5	1	-
35	GPIO35	GPIO35	GPIO35	FSPID	SUBSPID	SPIIO6	1	-
36	GPIO36	GPIO36	GPIO36	FSPICLK	SUBSPICLK	SPIIO7	1	-
37	GPIO37	GPIO37	GPIO37	FSPIQ	SUBSPIQ	SPIDQS	1	-
38	GPIO38	GPIO38	GPIO38	FSPIWP	SUBSPIWP	-	1	-
39	MTCK	MTCK	GPIO39	CLK_OUT3	SUBSPICS1	-	1	-
40	MTDO	MTDO	GPIO40	CLK_OUT2	-	-	1	-
41	MTDI	MTDI	GPIO41	CLK_OUT1	-	-	1	-
42	MTMS	MTMS	GPIO42	-	-	-	1	-
43	U0TXD	U0TXD	GPIO43	CLK_OUT1	-	-	3	-
44	U0RXD	U0RXD	GPIO44	CLK_OUT2	-	-	3	-
45	GPIO45	GPIO45	GPIO45	-	-	-	2	-
46	GPIO46	GPIO46	GPIO46	-	-	-	2	I

复位配置

“Reset” 一栏是每个 pad 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）

说明

- **R** - Pad 通过 RTC IO MUX 具有 RTC/模拟功能。
- **I** - Pad 只能配置为输入 GPIO。

请参考 《ESP32-S2 技术规格书》的附录查看管脚功能的完整表格。

5.12 RTC IO MUX 管脚清单

表 24 列出了 RTC 管脚和对应 GPIO pad。

表 24: RTC IO MUX 管脚清单

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

说明： TOUCH_PAD0 为内部通道，其模拟功能未引出至外部管脚。

5.13 基地址

用户可通过表 25 所示的基地址访问本章节所述模块。更多信息见章节 1 *系统和存储器*。

表 25: 模块基地址

模块名	访问方式	基地址值
GPIO	PeriBUS1	0x3F404000
	PeriBUS2	0x60004000
IO MUX	PeriBUS1	0x3F409000
	PeriBUS2	0x60009000
GPIOSD	PeriBUS2	0x60004F00
专用 GPIO	PeriBUS1	0x3F4CF000
RTCIO	PeriBUS1	0x3F408400
	PeriBUS2	0x60008400

5.14 寄存器列表

5.14.1 GPIO 交换矩阵寄存器列表

请注意，下面的地址是相对于 GPIO 基地址的地址偏移量（相对地址），请参阅章节 5.13 获取有关 GPIO 基地址的信息。

名称	描述	地址	权限
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 输出清除寄存器	0x000C	只写
GPIO_OUT1_REG	GPIO32 ~ 53 输出寄存器	0x0010	读/写
GPIO_OUT1_W1TS_REG	GPIO32 ~ 53 输出置位寄存器	0x0014	只写
GPIO_OUT1_W1TC_REG	GPIO32 ~ 53 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 ~ 53 输出使能寄存器	0x002C	读/写
GPIO_ENABLE1_W1TS_REG	GPIO32 ~ 53 输出使能置位寄存器	0x0030	只写
GPIO_ENABLE1_W1TC_REG	GPIO32 ~ 53 输出使能清零寄存器	0x0034	只写
GPIO_STRAP_REG	Bootstrap 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 ~ 53 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
GPIO_PIN2_REG	配置 GPIO pin 2	0x007C	读/写
...	...	...	...
GPIO_PIN51_REG	配置 GPIO pin 51	0x0140	读/写
GPIO_PIN52_REG	配置 GPIO pin 52	0x0144	读/写
GPIO_PIN53_REG	配置 GPIO pin 53	0x0148	读/写
GPIO_FUNC0_IN_SEL_CFG_REG	外设 function 0 输入选择寄存器	0x0154	读/写

名称	描述	地址	权限
GPIO_FUNC1_IN_SEL_CFG_REG	外设 function 1 输入选择寄存器	0x0158	读/写
GPIO_FUNC2_IN_SEL_CFG_REG	外设 function 2 输入选择寄存器	0x015C	读/写
...	...	...	...
GPIO_FUNC253_IN_SEL_CFG_REG	外设 function 253 输入选择寄存器	0x0548	读/写
GPIO_FUNC254_IN_SEL_CFG_REG	外设 function 254 输入选择寄存器	0x054C	读/写
GPIO_FUNC255_IN_SEL_CFG_REG	外设 function 255 输入选择寄存器	0x0550	读/写
GPIO_FUNC0_OUT_SEL_CFG_REG	GPIO0 外设输出选择	0x0554	读/写
GPIO_FUNC1_OUT_SEL_CFG_REG	GPIO1 外设输出选择	0x0558	读/写
GPIO_FUNC2_OUT_SEL_CFG_REG	GPIO2 外设输出选择	0x055C	读/写
..	...	...	...
GPIO_FUNC51_OUT_SEL_CFG_REG	GPIO51 外设输出选择	0x0620	读/写
GPIO_FUNC52_OUT_SEL_CFG_REG	GPIO52 外设输出选择	0x0624	读/写
GPIO_FUNC53_OUT_SEL_CFG_REG	GPIO53 外设输出选择	0x0628	读/写
GPIO_CLOCK_GATE_REG	GPIO 时钟门控寄存器	0x062C	读/写
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 ~ 53 中断状态置位寄存器	0x0054	只写
GPIO_STATUS1_W1TC_REG	GPIO32 ~ 53 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 ~ 53 中断源寄存器	0x0150	只读
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 ~ 53 中断状态寄存器	0x0050	读/写
GPIO_PCPU_INT_REG	GPIO0 ~ 31 PRO_CPU 中断状态寄存器	0x005C	只读
GPIO_PCPU_NMI_INT_REG	GPIO0 ~ 31 PRO_CPU 非屏蔽中断状态寄存器	0x0060	只读
GPIO_PCPU_INT1_REG	GPIO32 ~ 53 PRO_CPU 中断状态寄存器	0x0068	只读
GPIO_PCPU_NMI_INT1_REG	GPIO 32 ~ 53 PRO_CPU 非屏蔽中断状态寄存器	0x006C	只读

5.14.2 IO MUX 寄存器列表

请注意，下面的地址是相对于 IO MUX 基地址的地址偏移量（相对地址），请参阅章节 5.13 获取有关 IO MUX 基地址的信息。

名称	描述	地址	访问
IO_MUX_PIN_CTRL	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

名称	描述	地址	访问
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_XTAL_32K_P_REG	XTAL_32K_P 配置寄存器	0x0040	读/写
IO_MUX_XTAL_32K_N_REG	XTAL_32K_N 配置寄存器	0x0044	读/写
IO_MUX_DAC_1_REG	DAC_1 配置寄存器	0x0048	读/写
IO_MUX_DAC_2_REG	DAC_2 配置寄存器	0x004C	读/写
IO_MUX_GPIO_19_REG	GPIO19 配置寄存器	0x0050	读/写
IO_MUX_GPIO_20_REG	GPIO20 配置寄存器	0x0054	读/写
IO_MUX_GPIO_21_REG	GPIO21 配置寄存器	0x0058	读/写
IO_MUX_SPICS1_REG	SPICS1 配置寄存器	0x006C	读/写
IO_MUX_SPIHD_REG	SPIHD 配置寄存器	0x0070	读/写
IO_MUX_SPIWP_REG	SPIWP 配置寄存器	0x0074	读/写
IO_MUX_SPICS0_REG	SPICS0 配置寄存器	0x0078	读/写
IO_MUX_SPICLK_REG	SPICLK 配置寄存器	0x007C	读/写
IO_MUX_SPIQ_REG	SPIQ 配置寄存器	0x0080	读/写
IO_MUX_SPID_REG	SPID 配置寄存器	0x0084	读/写
IO_MUX_GPIO_33_REG	GPIO33 配置寄存器	0x0088	读/写
IO_MUX_GPIO_34_REG	GPIO34 配置寄存器	0x008C	读/写
IO_MUX_GPIO_35_REG	GPIO35 配置寄存器	0x0090	读/写
IO_MUX_GPIO_36_REG	GPIO36 配置寄存器	0x0094	读/写
IO_MUX_GPIO_37_REG	GPIO37 配置寄存器	0x0098	读/写
IO_MUX_GPIO_38_REG	GPIO38 配置寄存器	0x009C	读/写
IO_MUX_MTCK_REG	MTCK 配置寄存器	0x00A0	读/写
IO_MUX_MTDO_REG	MTDO 配置寄存器	0x00A4	读/写
IO_MUX_MTDI_REG	MTDI 配置寄存器	0x00A8	读/写
IO_MUX_MTMS_REG	MTMS 配置寄存器	0x00AC	读/写
IO_MUX_U0TXD_REG	U0TXD 配置寄存器	0x00B0	读/写
IO_MUX_U0RXD_REG	U0RXD 配置寄存器	0x00B4	读/写
IO_MUX_GPIO_45_REG	GPIO45 配置寄存器	0x00B8	读/写
IO_MUX_GPIO_46_REG	GPIO46 配置寄存器	0x00BC	读/写

5.14.3 SDM 寄存器列表

请注意，下面的地址是相对于 GPIOSD 基地址的地址偏移量（相对地址），请参阅章节 5.13 获取有关 GPIOSD 基地址的信息。

名称	描述	地址	权限
配置寄存器			
GPIOSD_SIGMADELTA0_REG	SDM0 占空比配置寄存器	0x0000	读/写

名称	描述	地址	权限
GPIOSD_SIGMADELTA1_REG	SDM1 占空比配置寄存器	0x0004	读/写
GPIOSD_SIGMADELTA2_REG	SDM2 占空比配置寄存器	0x0008	读/写
GPIOSD_SIGMADELTA3_REG	SDM3 占空比配置寄存器	0x000C	读/写
GPIOSD_SIGMADELTA4_REG	SDM4 占空比配置寄存器	0x0010	读/写
GPIOSD_SIGMADELTA5_REG	SDM5 占空比配置寄存器	0x0014	读/写
GPIOSD_SIGMADELTA6_REG	SDM6 占空比配置寄存器	0x0018	读/写
GPIOSD_SIGMADELTA7_REG	SDM7 占空比配置寄存器	0x001C	读/写
GPIOSD_SIGMADELTA_CG_REG	时钟门控配置寄存器	0x0020	读/写
GPIOSD_SIGMADELTA_MISC_REG	MISC 寄存器	0x0024	读/写
GPIOSD_SIGMADELTA_VERSION_REG	版本控制寄存器	0x0028	读/写

5.14.4 专用 GPIO 寄存器列表

请注意，下面的地址是相对于专用 GPIO 基地址的地址偏移量（相对地址），请参阅章节 5.13 获取有关专用 GPIO 基地址的信息。

名称	描述	地址	访问
配置寄存器			
DEDIC_GPIO_OUT_DRT_REG	专用 GPIO 直接输出寄存器	0x0000	WO
DEDIC_GPIO_OUT_MSK_REG	专用 GPIO 屏蔽输出寄存器	0x0004	WO
DEDIC_GPIO_OUT_IDV_REG	专用 GPIO 单独输出寄存器	0x0008	WO
DEDIC_GPIO_OUT_CPU_REG	专用 GPIO 输出模式选择寄存器	0x0010	R/W
DEDIC_GPIO_IN_DLY_REG	专用 GPIO 输入延时配置寄存器	0x0014	R/W
DEDIC_GPIO_INTR_RCGN_REG	专用 GPIO 中断产生模式配置寄存器	0x001C	R/W
状态寄存器			
DEDIC_GPIO_OUT_SCAN_REG	专用 GPIO 输出状态寄存器	0x000C	RO
DEDIC_GPIO_IN_SCAN_REG	专用 GPIO 输入状态寄存器	0x0018	RO
中断寄存器			
DEDIC_GPIO_INTR_RAW_REG	原始中断状态	0x0020	RO
DEDIC_GPIO_INTR_RLS_REG	中断使能位	0x0024	R/W
DEDIC_GPIO_INTR_ST_REG	屏蔽中断状态	0x0028	RO
DEDIC_GPIO_INTR_CLR_REG	中断清零位	0x002C	WO

5.14.5 RTC IO MUX 寄存器列表

请注意，下面的地址是相对于 RTCIO 基地址的地址偏移量（相对地址），请参阅章节 5.13 获取有关 RTCIO 基地址的信息。

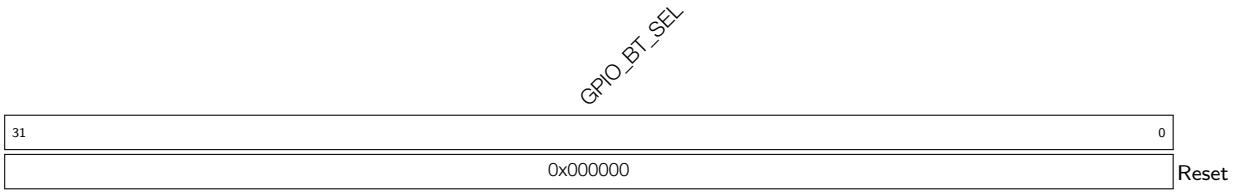
名称	描述	地址	权限
GPIO 配置 / 数据寄存器			
RTCIO_RTC_GPIO_OUT_REG	RTC GPIO 输出寄存器	0x0000	读/写
RTCIO_RTC_GPIO_OUT_W1TS_REG	RTC GPIO 输出置位寄存器	0x0004	只写
RTCIO_RTC_GPIO_OUT_W1TC_REG	RTC GPIO 输出清零寄存器	0x0008	只写
RTCIO_RTC_GPIO_ENABLE_REG	RTC GPIO 输出使能寄存器	0x000C	读/写
RTCIO_RTC_GPIO_ENABLE_W1TS_REG	RTC GPIO 输出使能置位寄存器	0x0010	只写

名称	描述	地址	权限
RTCIO_RTC_GPIO_ENABLE_W1TC_REG	RTC GPIO 输出使能清零寄存器	0x0014	只写
RTCIO_RTC_GPIO_STATUS_REG	RTC GPIO 中断状态寄存器	0x0018	读/写
RTCIO_RTC_GPIO_STATUS_W1TS_REG	RTC GPIO 中断状态置位寄存器	0x001C	只写
RTCIO_RTC_GPIO_STATUS_W1TC_REG	RTC GPIO 中断状态清零寄存器	0x0020	只写
RTCIO_RTC_GPIO_IN_REG	RTC GPIO 输入寄存器	0x0024	只读
RTCIO_RTC_GPIO_PIN0_REG	Pin0 RTC 配置	0x0028	读/写
RTCIO_RTC_GPIO_PIN1_REG	Pin1 RTC 配置	0x002C	读/写
RTCIO_RTC_GPIO_PIN2_REG	Pin2 RTC 配置	0x0030	读/写
RTCIO_RTC_GPIO_PIN3_REG	Pin3 RTC 配置	0x0034	读/写
...	...	...	...
RTCIO_RTC_GPIO_PIN19_REG	Pin19 RTC 配置	0x0074	读/写
RTCIO_RTC_GPIO_PIN20_REG	Pin20 RTC 配置	0x0078	读/写
RTCIO_RTC_GPIO_PIN21_REG	Pin21 RTC 配置	0x007C	读/写
RTC GPIO 功能配置寄存器			
RTCIO_TOUCH_PAD0_REG	Touch pad 0 配置寄存器	0x0084	读/写
RTCIO_TOUCH_PAD1_REG	Touch pad 1 配置寄存器	0x0088	读/写
RTCIO_TOUCH_PAD2_REG	Touch pad 2 配置寄存器	0x008C	读/写
...	...	...	...
RTCIO_TOUCH_PAD13_REG	Touch pad 13 配置寄存器	0x00B8	读/写
RTCIO_TOUCH_PAD14_REG	Touch pad 14 配置寄存器	0x00BC	读/写
RTCIO_XTAL_32P_PAD_REG	32KHz crystal P-pad 配置寄存器	0x00C0	读/写
RTCIO_XTAL_32N_PAD_REG	32KHz crystal N-pad 配置寄存器	0x00C4	读/写
RTCIO_PAD_DAC1_REG	DAC1 配置寄存器	0x00C8	读/写
RTCIO_PAD_DAC2_REG	DAC2 配置寄存器	0x00CC	读/写
RTCIO_RTC_PAD19_REG	Touch pad 19 配置寄存器	0x00D0	读/写
RTCIO_RTC_PAD20_REG	Touch pad 20 配置寄存器	0x00D4	读/写
RTCIO_RTC_PAD21_REG	Touch pad 21 配置寄存器	0x00D8	读/写
RTCIO_XTL_EXT_CTR_REG	晶振断电 GPIO 使能源	0x00E0	读/写
RTCIO_SAR_I2C_IO_REG	RTC I2C pad 选择寄存器	0x00E4	读/写

5.15 寄存器

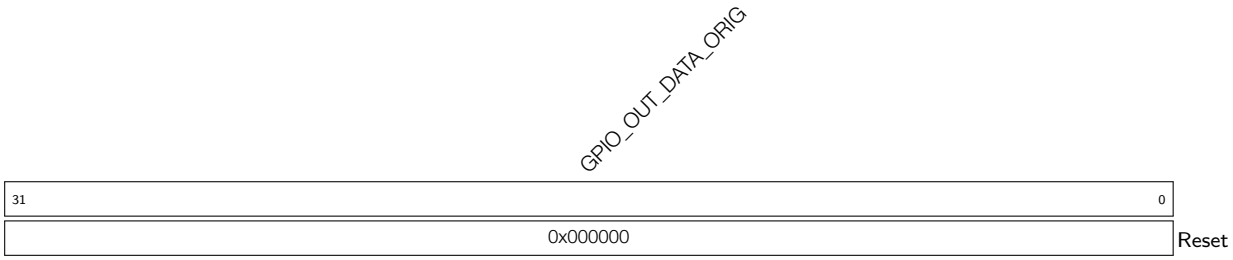
5.15.1 GPIO 交换矩阵寄存器

Register 5.1: GPIO_BT_SELECT_REG (0x0000)



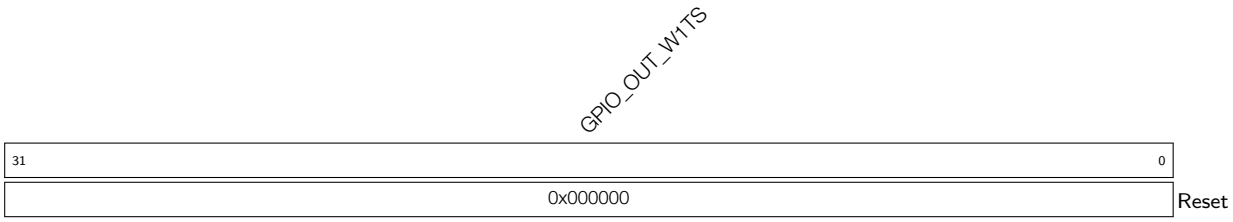
GPIO_BT_SEL 保留（读/写）

Register 5.2: GPIO_OUT_REG (0x0004)



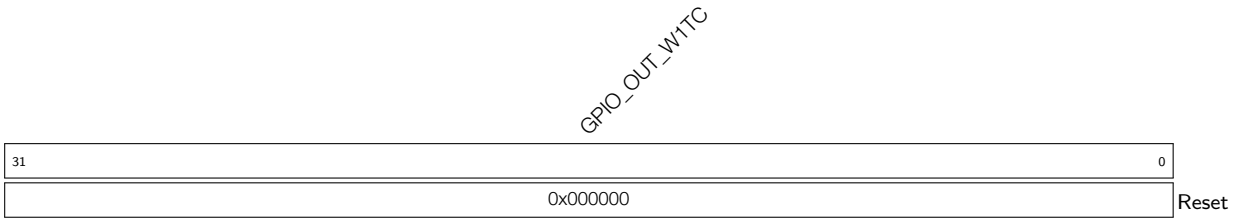
GPIO_OUT_DATA_ORIG 简单 GPIO 输出模式下 GPIO0 ~ GPIO31 的输出值。bit0 ~ bit31 的值分别对应 GPIO0 ~ 31 的输出值，bit22 ~ bit25 无效。(读/写)

Register 5.3: GPIO_OUT_W1TS_REG (0x0008)



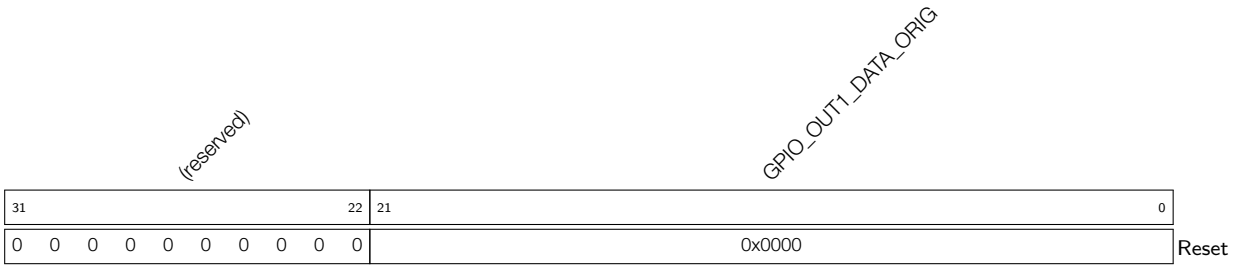
GPIO_OUT_W1TS GPIO0 ~ 31 输出置位寄存器。每一位置 1，GPIO_OUT_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_OUT_REG。(只写)

Register 5.4: GPIO_OUT_W1TC_REG (0x000C)



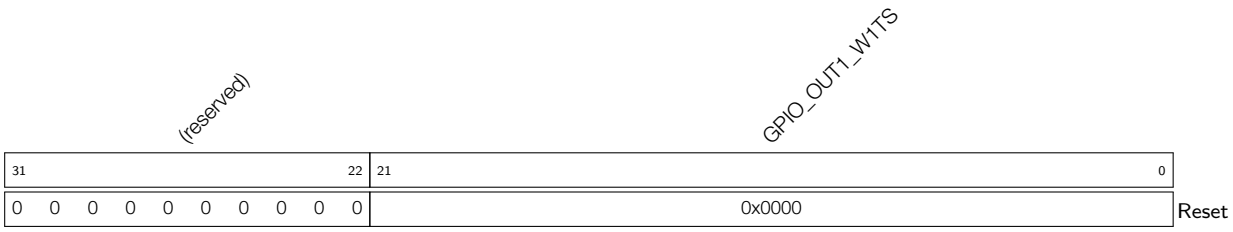
GPIO_OUT_W1TC GPIO0 ~ 31 输出清零寄存器。每一位置 1，则 GPIO_OUT_REG 中的相应位会清零。注：推荐使用此寄存器来清零 GPIO_OUT_REG。(只写)

Register 5.5: GPIO_OUT1_REG (0x0010)



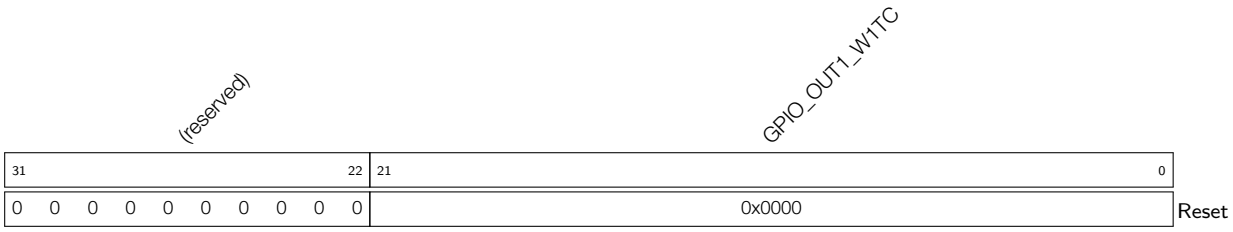
GPIO_OUT1_DATA_ORIG 简单 GPIO 输出模式下 GPIO32 ~ 53 的输出值。bit0 ~ bit13 的值分别对应 GPIO32 ~ GPIO45 的输出值，bit14 ~ bit21 无效。（读/写）

Register 5.6: GPIO_OUT1_W1TS_REG (0x0014)



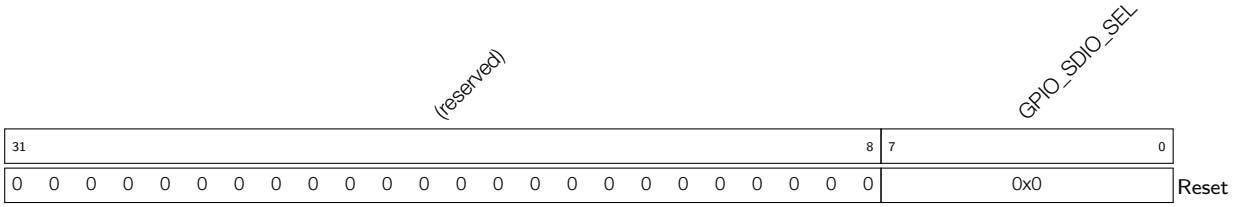
GPIO_OUT1_W1TS GPIO32 ~ 53 输出置位寄存器。每一位置 1，则 GPIO_OUT1_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_OUT1_REG。（只写）

Register 5.7: GPIO_OUT1_W1TC_REG (0x0018)



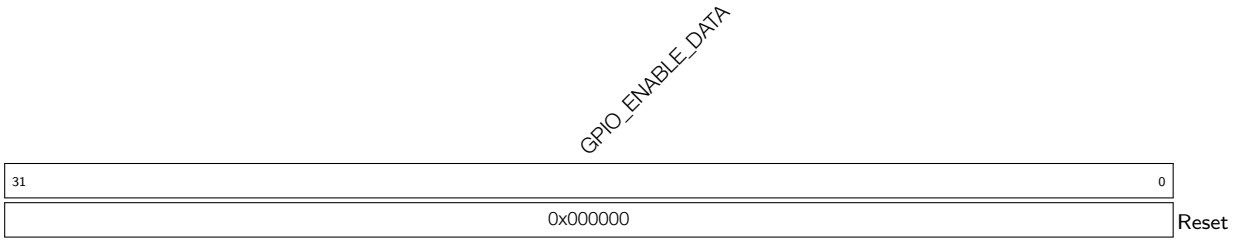
GPIO_OUT1_W1TC GPIO32 ~ 53 输出清零寄存器。每一位置 1，则 GPIO_OUT1_REG 中的相应位会清零。注：推荐使用此寄存器来清零 GPIO_OUT1_REG。（只写）

Register 5.8: GPIO_SDIO_SELECT_REG (0x001C)



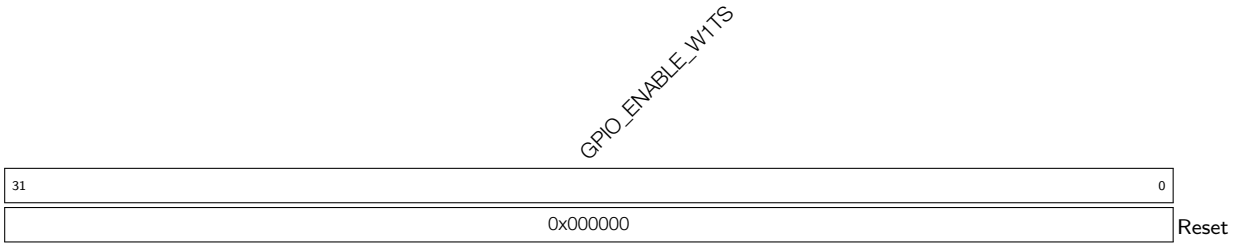
GPIO_SDIO_SEL 保留（读/写）

Register 5.9: GPIO_ENABLE_REG (0x0020)



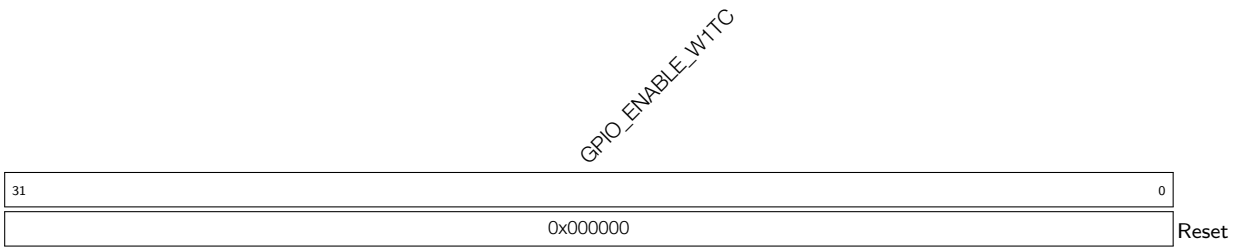
GPIO_ENABLE_DATA GPIO0 ~ 31 输出使能寄存器。（读/写）

Register 5.10: GPIO_ENABLE_W1TS_REG (0x0024)



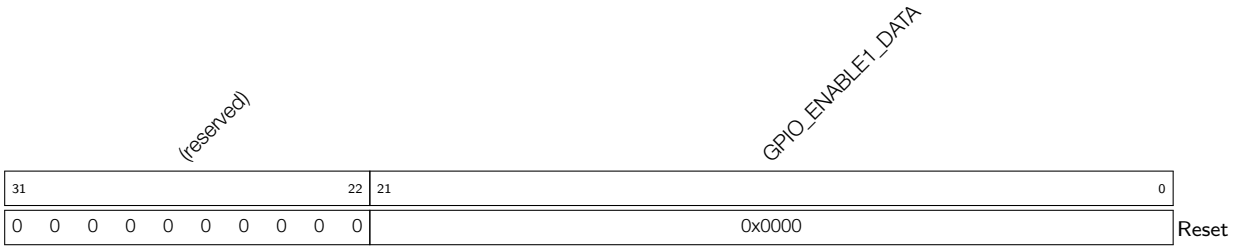
GPIO_ENABLE_W1TS GPIO0 ~ 31 输出使能置位寄存器。每一位置 1，则 GPIO_ENABLE_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_ENABLE_REG。（只写）

Register 5.11: GPIO_ENABLE_W1TC_REG (0x0028)



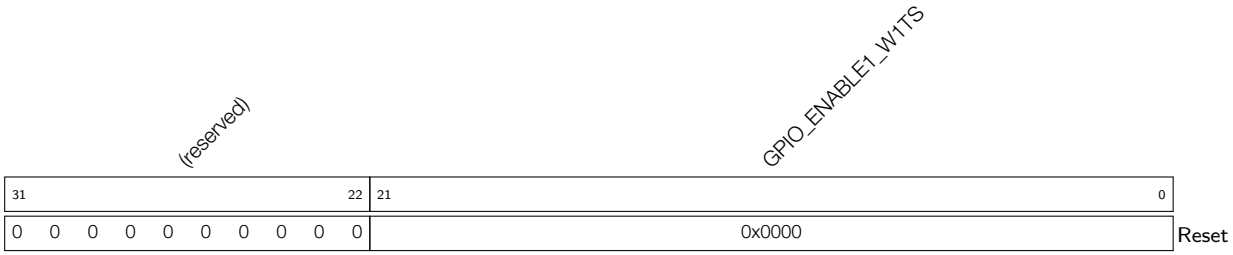
GPIO_ENABLE_W1TC GPIO0 ~ 31 输出使能清零寄存器。每一位置 1，则 GPIO_ENABLE_REG 中的相应位会清零。注：推荐使用此寄存器清零 GPIO_ENABLE_REG。（只写）

Register 5.12: GPIO_ENABLE1_REG (0x002C)



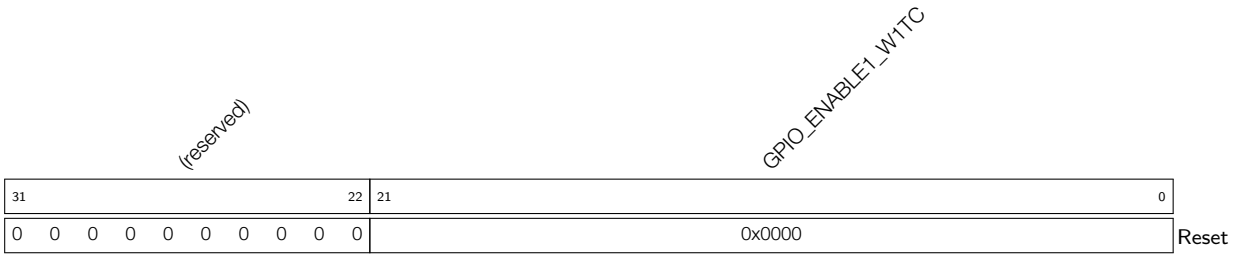
GPIO_ENABLE1_DATA GPIO32 ~ 53 输出使能寄存器。(读/写)

Register 5.13: GPIO_ENABLE1_W1TS_REG (0x0030)



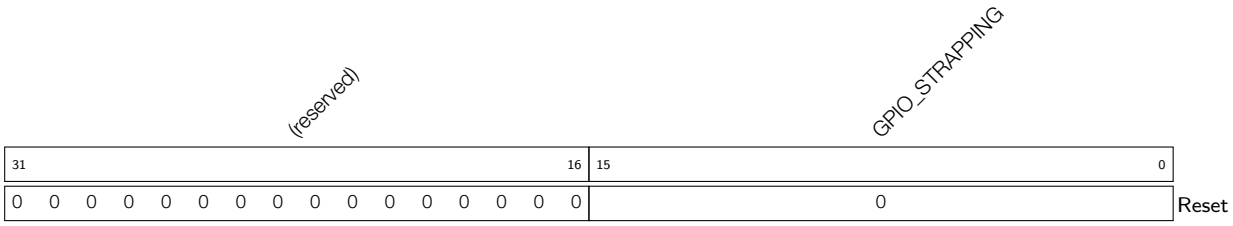
GPIO_ENABLE1_W1TS GPIO32 ~ 53 输出使能置位寄存器。每一位置 1，则 GPIO_ENABLE1_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_ENABLE1_REG。(只写)

Register 5.14: GPIO_ENABLE1_W1TC_REG (0x0034)



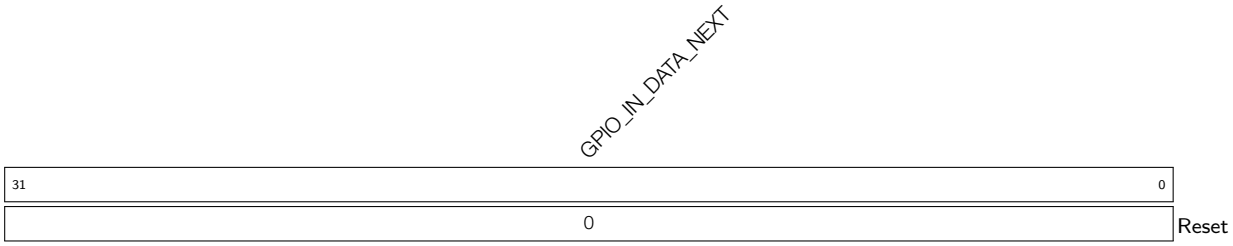
GPIO_ENABLE1_W1TC GPIO32 ~ 53 输出使能清零寄存器。每一位置 1，则 GPIO_ENABLE1_REG 中的相应位会清零。注：推荐使用此寄存器清零 GPIO_ENABLE1_REG。(只写)

Register 5.15: GPIO_STRAP_REG (0x0038)



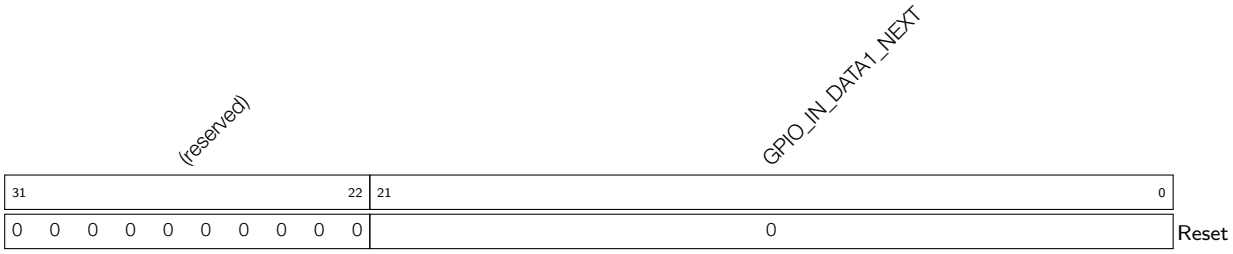
GPIO_STRAPPING GPIO Strapping 值：bit4 ~ bit2 分别对应 GPIO45、GPIO0 和 GPIO46。（只读）

Register 5.16: GPIO_IN_REG (0x003C)



GPIO_IN_DATA_NEXT GPIO0 ~ 31 输入值。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。（只读）

Register 5.17: GPIO_IN1_REG (0x0040)



GPIO_IN_DATA1_NEXT GPIO32 ~ 53 输入值。每个 bit 代表 pad 的片外输入值。（只读）

Register 5.18: GPIO_PIN*n*_REG (*n*:0-53) (0x0074+4n*)**

GPIO_PIN*n*_SYNC2_BYPASS 使能 GPIO 输入信号第二拍为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。（读/写）

GPIO_PIN*n*_PAD_DRIVER pad 驱动选择。0：正常输出；1：开漏输出。（读/写）

GPIO_PIN*n*_SYNC1_BYPASS 使能 GPIO 输入信号第一拍为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。（读/写）

GPIO_PIN*n*_INT_TYPE 中断类型选择。（读/写）

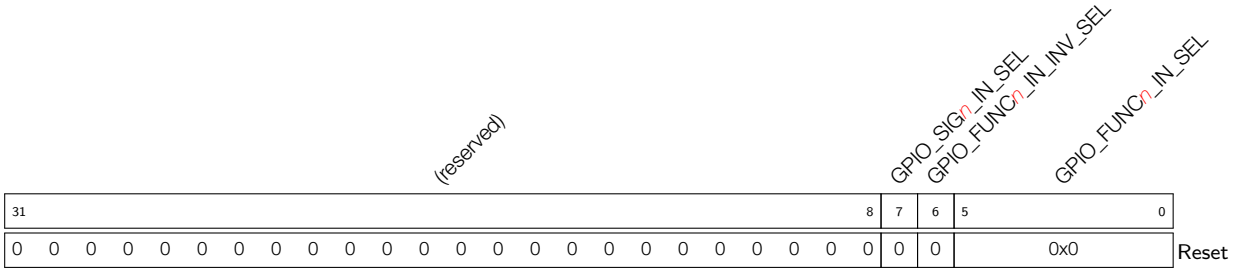
0：禁用 GPIO 中断
1：上升沿触发
2：下降沿触发
3：任一沿触发
4：低电平触发
5：高电平触发

GPIO_PIN*n*_WAKEUP_ENABLE 使能 GPIO 唤醒，仅能将 CPU 从 Light-sleep 模式唤醒。（读/写）

GPIO_PIN*n*_CONFIG 保留。（读/写）

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。（读/写）

Register 5.19: GPIO_FUNC*n*_IN_SEL_CFG_REG (*n*:0-255) (0x0154+4n*)**

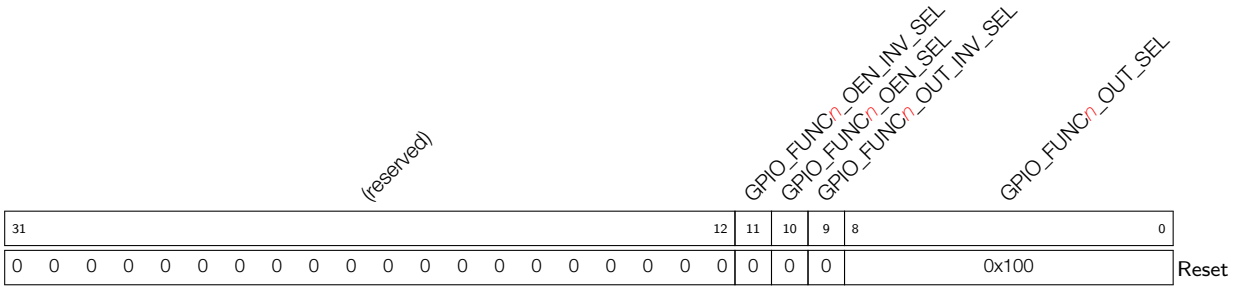


GPIO_FUNC*n*_IN_SEL 外设输入 *m* 的选择控制位。此位选择 1 个 GPIO 交换矩阵输入管脚与信号连接，或者选择 0x38 与恒高电平输入信号连接，或者选择 0x3C 与恒低电平输入信号连接。(读/写)

GPIO_FUNC*n*_IN_INV_SEL 反转输入值。1：反转；0：不反转。(读/写)

GPIO_SIG*n*_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。(读/写)

Register 5.20: GPIO_FUNC*n*_OUT_SEL_CFG_REG (*n*:0-53) (0x0554+4n*)**



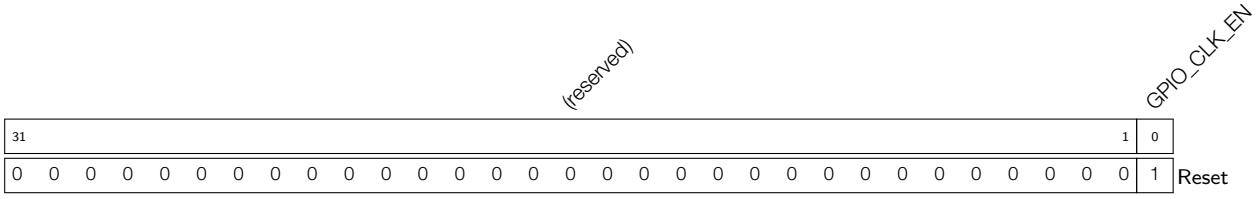
GPIO_FUNC*n*_OUT_SEL GPIO 输出 *n* 的选择控制位。值为 *s* (0<=*s*<256) 连接外设输出 *s* 与 GPIO 输出 *n*。值为 256 选择 GPIO_OUT_REG/GPIO_OUT1_REG 和 GPIO_ENABLE_REG/GPIO_ENABLE1_REG 的 bit *n* 作为输出值和输出使能。(读/写)

GPIO_FUNC*n*_OUT_INV_SEL 0：不反转输出值；1：反转输出值。(读/写)

GPIO_FUNC*n*_OEN_SEL 0：采用外设的输出使能信号；1：强制使用 GPIO_ENABLE_REG 的 bit *n* 用作输出使能信号。(读/写)

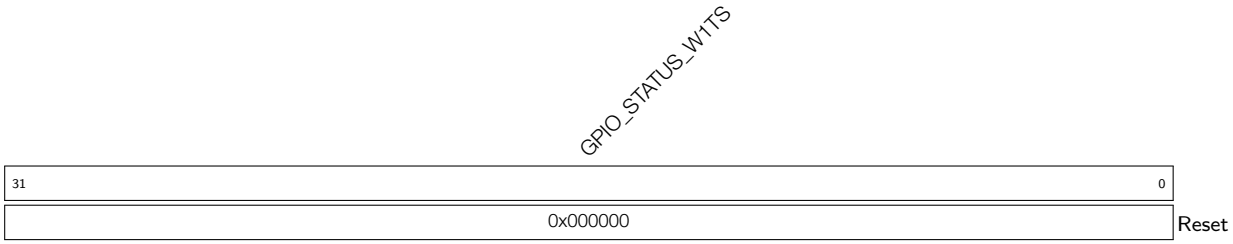
GPIO_FUNC*n*_OEN_INV_SEL 0：不反转输出使能信号；1：反转输出使能信号。(读/写)

Register 5.21: GPIO_CLOCK_GATE_REG (0x062C)



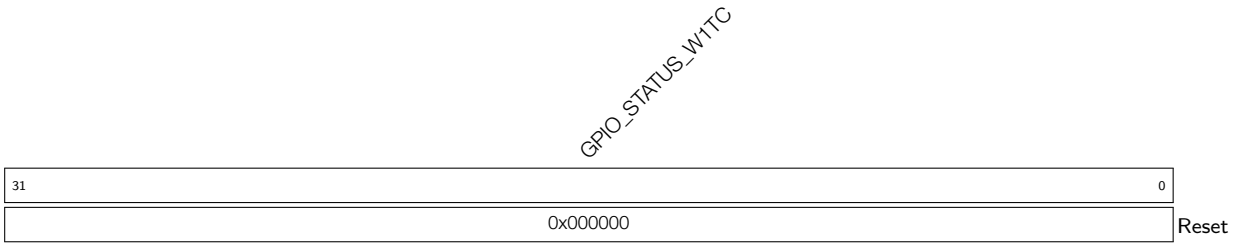
GPIO_CLK_EN 时钟门控使能。此位置 1，则时钟自由运转。(读/写)

Register 5.22: GPIO_STATUS_W1TS_REG (0x0048)



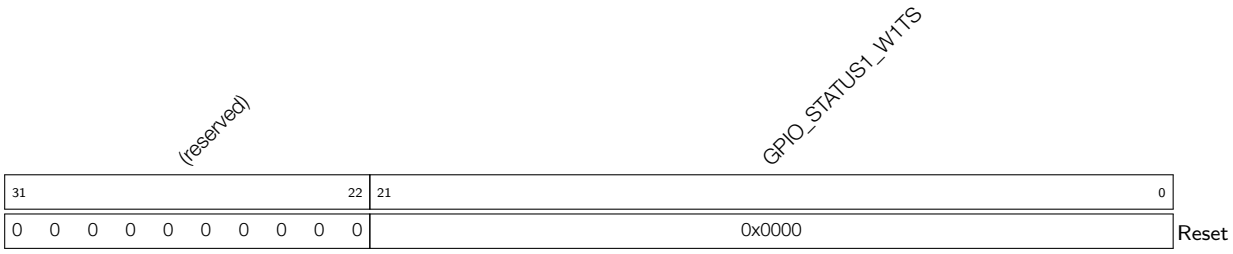
GPIO_STATUS_W1TS GPIO0 ~ 31 中断状态置位寄存器。每位置 1，则 GPIO_STATUS_INTERRUPT 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_STATUS_INTERRUPT。(只写)

Register 5.23: GPIO_STATUS_W1TC_REG (0x004C)



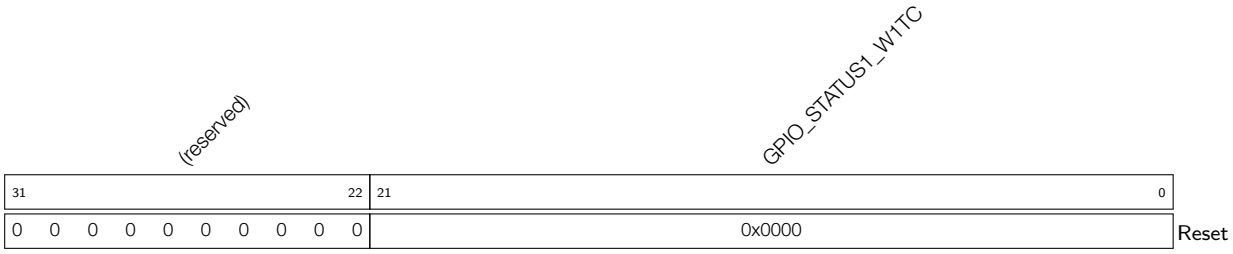
GPIO_STATUS_W1TC GPIO0 ~ 31 中断状态清除寄存器。每一位置 1，则 GPIO_STATUS_INTERRUPT 中的相应位也会清零。注：推荐使用此寄存器来清零 GPIO_STATUS_INTERRUPT。(只写)

Register 5.24: GPIO_STATUS1_W1TS_REG (0x0054)



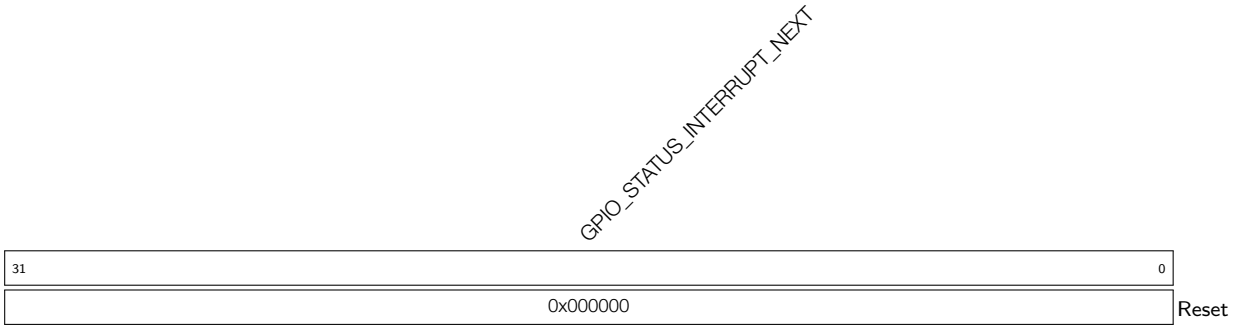
GPIO_STATUS1_W1TS GPIO32 ~ 53 中断状态置位寄存器。每一位置 1，GPIO_STATUS_INTERRUPT1 中相应位也置 1。注：推荐使用此寄存器来置位 GPIO_STATUS_INTERRUPT1。(只写)

Register 5.25: GPIO_STATUS1_W1TC_REG (0x0058)



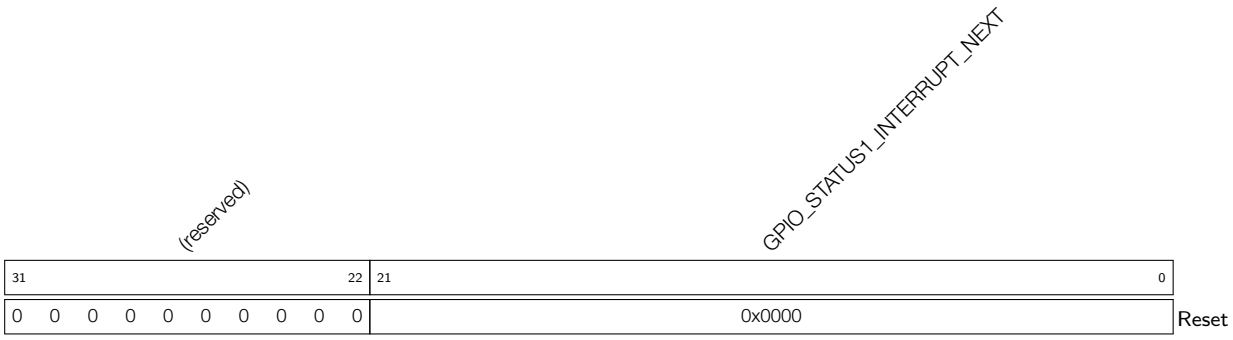
GPIO_STATUS1_W1TC GPIO32 ~ 53 中断状态清除寄存器。每一位置 1，则 GPIO_STATUS_INTERRUPT1 中的相应位也将清零。注：推荐使用此寄存器来清零 GPIO_STATUS_INTERRUPT1。(只写)

Register 5.26: GPIO_STATUS_NEXT_REG (0x014C)



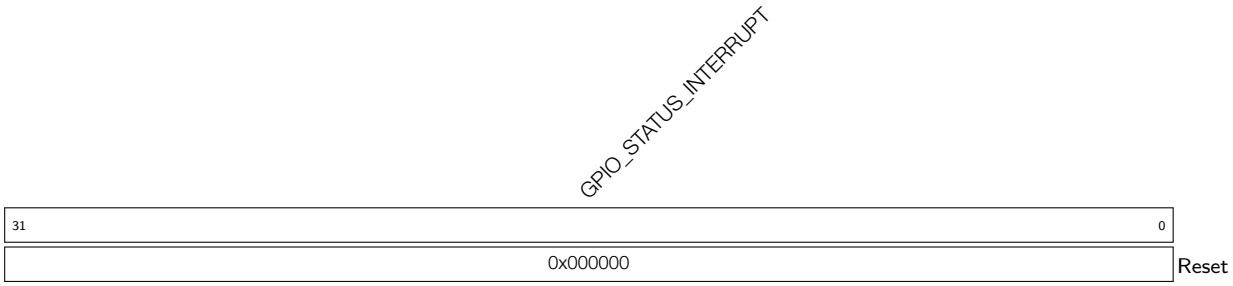
GPIO_STATUS_INTERRUPT_NEXT GPIO0 ~ 31 中断源信号，可以设置为上升沿中断、下降沿中断、电平敏感中断或任一沿中断。(只读)

Register 5.27: GPIO_STATUS_NEXT1_REG (0x0150)



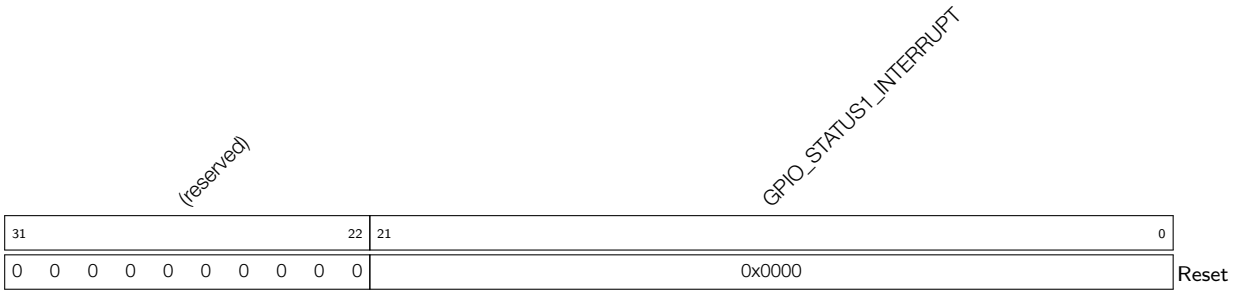
GPIO_STATUS1_INTERRUPT_NEXT GPIO32 ~ 53 的中断源信号。（只读）

Register 5.28: GPIO_STATUS_REG (0x0044)



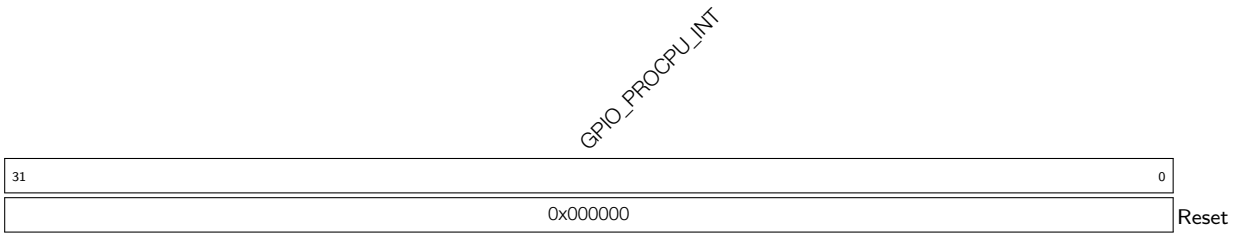
GPIO_STATUS_INTERRUPT GPIO0 ~ 31 中断状态寄存器。（读/写）

Register 5.29: GPIO_STATUS1_REG (0x0050)



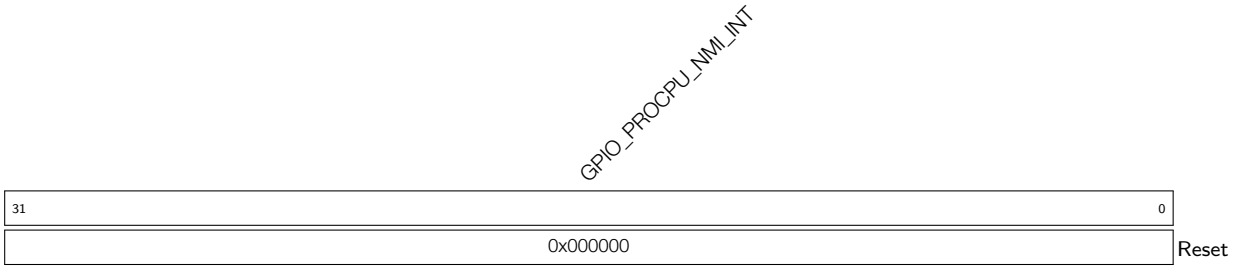
GPIO_STATUS1_INTERRUPT GPIO32 ~ 53 中断状态寄存器。（读/写）

Register 5.30: GPIO_PCPU_INT_REG (0x005C)



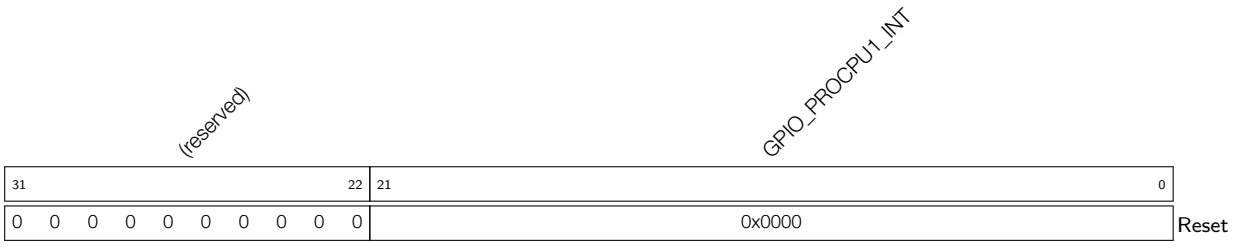
GPIO_PROCPU_INT GPIO0 ~ 31 PRO_CPU 中断状态。如果 GPIO_PIN*n*_REG 中 bit13 高电平有效，即使能 CPU 中断，则此寄存器所示的中断状态应与 GPIO_STATUS_REG 中相应 bit 的中断状态一致。(只读)

Register 5.31: GPIO_PCPU_NMI_INT_REG (0x0060)



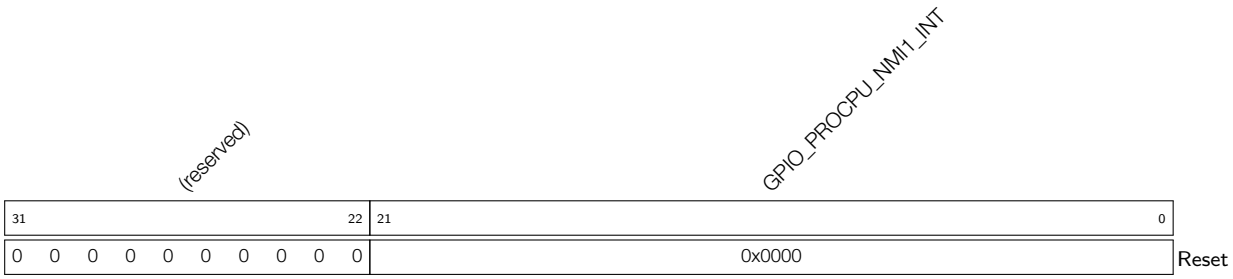
GPIO_PROCPU_NMI_INT GPIO0 ~ 31 PRO_CPU 非屏蔽中断状态寄存器。如果 GPIO_PIN*n*_REG 中 bit14 高电平有效，即使能 CPU 非屏蔽中断，则此寄存器所示的中断状态应与 GPIO_STATUS_REG 中相应 bit 的中断状态一致。(只读)

Register 5.32: GPIO_PCPU_INT1_REG (0x0068)



GPIO_PROCPU1_INT GPIO32 ~ 53 PRO_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。(只读)

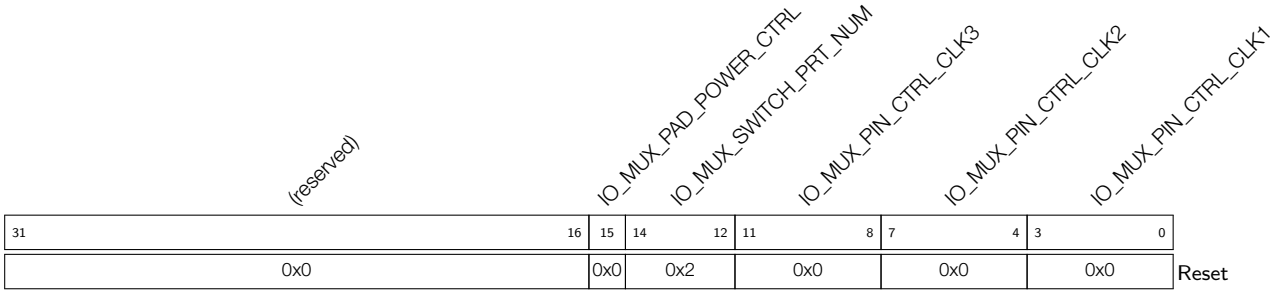
Register 5.33: GPIO_PCPU_NMI_INT1_REG (0x006C)



GPIO_PROCPU_NMI1_INT GPIO32 ~ 53 PRO_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。(只读)

5.15.2 IO MUX 寄存器

Register 5.34: IO_MUX_PIN_CTRL (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[3:0] = 0x0；

CLK_OUT2，配置 IO_MUX_PIN_CTRL[3:0] = 0x0 and IO_MUX_PIN_CTRL[7:4] = 0x0；

CLK_OUT3，配置 IO_MUX_PIN_CTRL[3:0] = 0x0 and IO_MUX_PIN_CTRL[11:8] = 0x0。

说明：

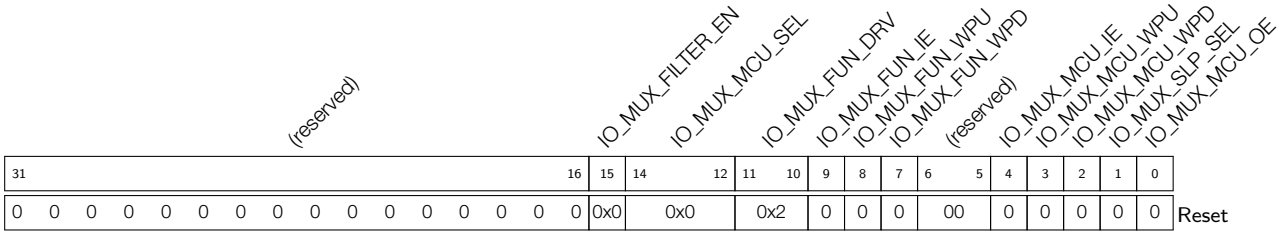
只能有上述配置组合。

CLK_OUT1 ~ 3 可在 IO_MUX Pad 列表中查询。

IO_MUX_SWITCH_PRT_NUM IO Pad 电源切换延时，延时单位为一个 APB 时钟周期。

IO_MUX_PAD_POWER_CTRL 选择 GPIO33 ~ 37 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。

Register 5.35: IO_MUX_*n*_REG (*n*: GPIO0-GPIO21, GPIO26-GPIO46) (0x0010+4n*)**



IO_MUX_MCU_OE 睡眠模式下 pad 的输出使能。1：输出使能；0：输出关闭。(读/写)

IO_MUX_SLP_SEL Pad 的睡眠模式选择。置 1 将使能睡眠模式。(读/写)

IO_MUX_MCU_WPD 睡眠模式下 pad 的下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

IO_MUX_MCU_WPU 睡眠模式下 pad 的上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

IO_MUX_MCU_IE 睡眠模式下 pad 的输入使能。1：输入使能；0：输入关闭。(读/写)

IO_MUX_FUN_WPD Pad 的下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

IO_MUX_FUN_WPU Pad 的上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

IO_MUX_FUN_IE Pad 的输入使能。1：输入使能；0：输入关闭。(读/写)

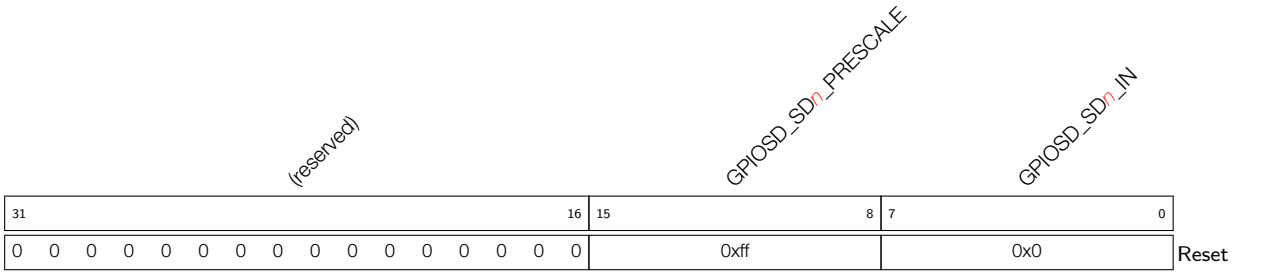
IO_MUX_FUN_DRV 选择 Pad 驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

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。(读/写)

IO_MUX_FILTER_EN Pad 输入信号滤波使能。1：滤波使能；0：滤波关闭。(读/写)

5.15.3 SDM 寄存器

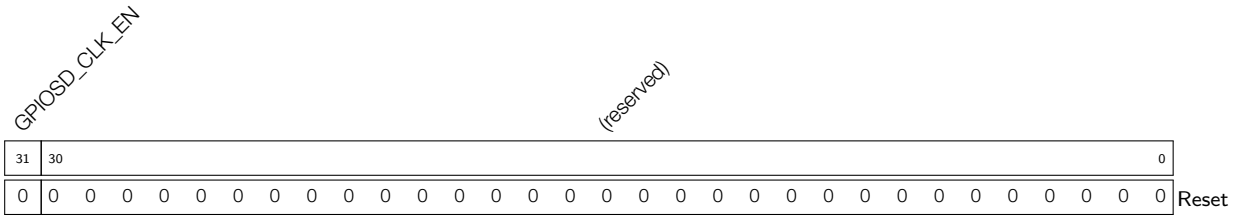
Register 5.36: GPIOSD_SIGMADELTA*n*_REG (*n*: 0-7) (0x0000+4n*)**



GPIOSD_SD*n*_IN 配置 SDM 输出信号的占空比。(读/写)

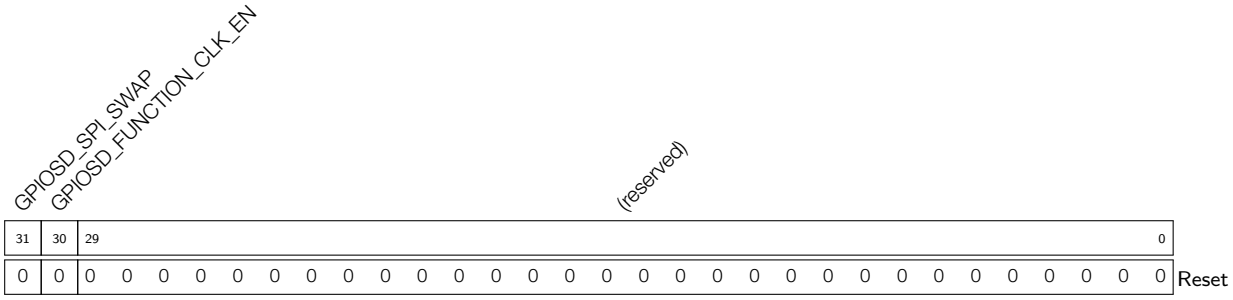
GPIOSD_SD*n*_PRESCALE 配置 APB_CLK 分频系数。(读/写)

Register 5.37: GPIOSD_SIGMADELTA_CG_REG (0x0020)



GPIOSD_CLK_EN 使能 SDM 配置寄存器的时钟。(读/写)

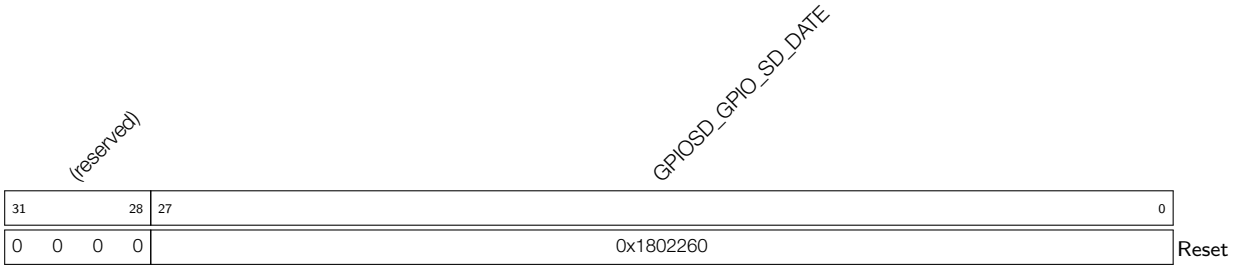
Register 5.38: GPIOSD_SIGMADELTA_MISC_REG (0x0024)



GPIOSD_FUNCTION_CLK_EN 使能 SDM 的时钟。(读/写)

GPIOSD_SPI_SWAP 保留。(读/写)

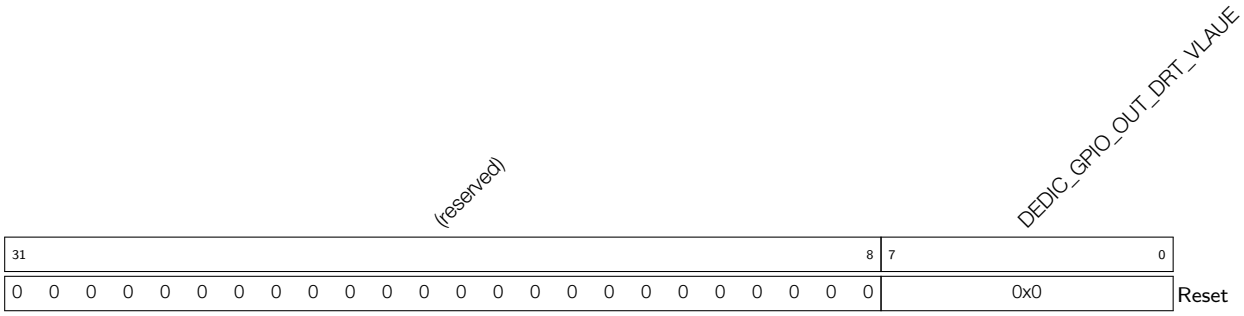
Register 5.39: GPIOSD_SIGMADELTA_VERSION_REG (0x0028)



GPIOSD_GPIO_SD_DATE 版本控制寄存器。(读/写)

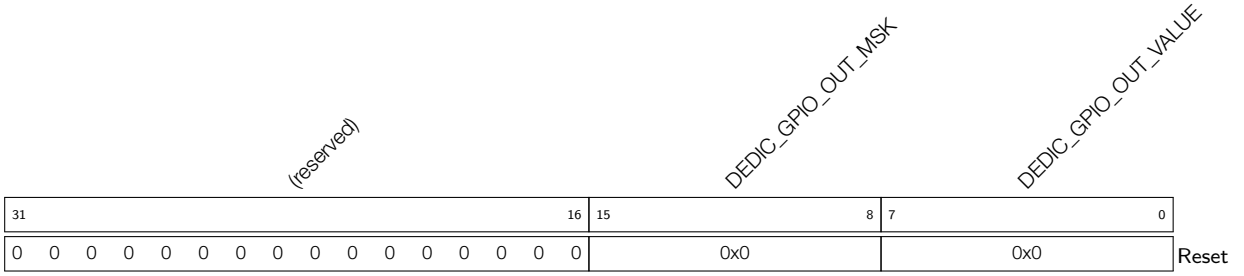
5.15.4 专用 GPIO 寄存器

Register 5.40: DEDIC_GPIO_OUT_DRT_REG (0x0000)



DEDIC_GPIO_OUT_DRT_VLAUE 配置 8 通道专用 GPIO 的直接输出值。(只写)

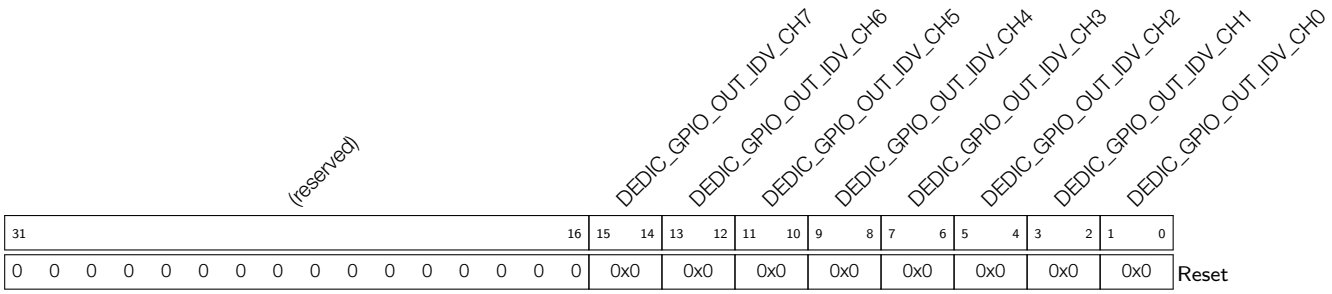
Register 5.41: DEDIC_GPIO_OUT_MSK_REG (0x0004)



DEDIC_GPIO_OUT_VALUE 配置 8 通道专用 GPIO 需更新的输出值。(只写)

DEDIC_GPIO_OUT_MSK 配置要更新的通道(位选)。置 1 则相应通道的输出被选择。(只写)

Register 5.42: DEDIC_GPIO_OUT_IDV_REG (0x0008)



DEDIC_GPIO_OUT_IDV_CH0 配置通道 0 输出值。0：保持输出值。1：置位输出值。2：清零输出值。3：反转输出值。（只写）

DEDIC_GPIO_OUT_IDV_CH1 配置通道 1 输出值。0：保持输出值。1：置位输出值。2：清零输出值。3：反转输出值。（只写）

DEDIC_GPIO_OUT_IDV_CH2 配置通道 2 输出值。0：保持输出值。1：置位输出值。2：清零输出值。3：反转输出值。（只写）

DEDIC_GPIO_OUT_IDV_CH3 配置通道 3 输出值。0：保持输出值。1：置位输出值。2：清零输出值。3：反转输出值。（只写）

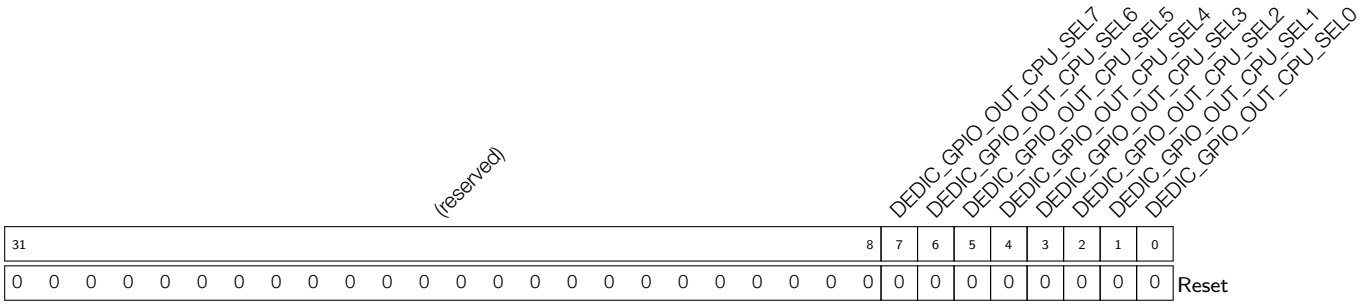
DEDIC_GPIO_OUT_IDV_CH4 配置通道 4 输出值。0：保持输出值。1：置位输出值。2：清零输出值。3：反转输出值。（只写）

DEDIC_GPIO_OUT_IDV_CH5 配置通道 5 输出值。0：保持输出值。1：置位输出值。2：清零输出值。3：反转输出值。（只写）

DEDIC_GPIO_OUT_IDV_CH6 配置通道 6 输出值。0：保持输出值。1：置位输出值。2：清零输出值。3：反转输出值。（只写）

DEDIC_GPIO_OUT_IDV_CH7 配置通道 7 输出值。0：保持输出值。1：置位输出值。2：清零输出值。3：反转输出值。（只写）

Register 5.43: DEDIC_GPIO_OUT_CPU_REG (0x0010)



DEDIC_GPIO_OUT_CPU_SEL0 选择通道 0 GPIO 输出值是由寄存器配置还是由 CPU 指令配置。0：选择使用寄存器。1：选择使用 CPU 指令。（读/写）

DEDIC_GPIO_OUT_CPU_SEL1 选择通道 1 GPIO 输出值是由寄存器配置还是由 CPU 指令配置。0：选择使用寄存器。1：选择使用 CPU 指令。（读/写）

DEDIC_GPIO_OUT_CPU_SEL2 选择通道 2 GPIO 输出值是由寄存器配置还是由 CPU 指令配置。0：选择使用寄存器。1：选择使用 CPU 指令。（读/写）

DEDIC_GPIO_OUT_CPU_SEL3 选择通道 3 GPIO 输出值是由寄存器配置还是由 CPU 指令配置。0：选择使用寄存器。1：选择使用 CPU 指令。（读/写）

DEDIC_GPIO_OUT_CPU_SEL4 选择通道 4 GPIO 输出值是由寄存器配置还是由 CPU 指令配置。0：选择使用寄存器。1：选择使用 CPU 指令。（读/写）

DEDIC_GPIO_OUT_CPU_SEL5 选择通道 5 GPIO 输出值是由寄存器配置还是由 CPU 指令配置。0：选择使用寄存器。1：选择使用 CPU 指令。（读/写）

DEDIC_GPIO_OUT_CPU_SEL6 选择通道 6 GPIO 输出值是由寄存器配置还是由 CPU 指令配置。0：选择使用寄存器。1：选择使用 CPU 指令。（读/写）

DEDIC_GPIO_OUT_CPU_SEL7 选择通道 7 GPIO 输出值是由寄存器配置还是由 CPU 指令配置。0：选择使用寄存器。1：选择使用 CPU 指令。（读/写）

Register 5.44: DEDIC_GPIO_IN_DLY_REG (0x0014)

DEDIC_GPIO_IN_DLY_CH0 配置 GPIO0 的输入延时。0：无延时。1：延时一个时钟周期。2：延时两个时钟周期。3：延时三个时钟周期。(读/写)

DEDIC_GPIO_IN_DLY_CH1 配置 GPIO1 的输入延时。0：无延时。1：延时一个时钟周期。2：延时两个时钟周期。3：延时三个时钟周期。(读/写)

DEDIC_GPIO_IN_DLY_CH2 配置 GPIO2 的输入延时。0：无延时。1：延时一个时钟周期。2：延时两个时钟周期。3：延时三个时钟周期。(读/写)

DEDIC_GPIO_IN_DLY_CH3 配置 GPIO3 的输入延时。0：无延时。1：延时一个时钟周期。2：延时两个时钟周期。3：延时三个时钟周期。(读/写)

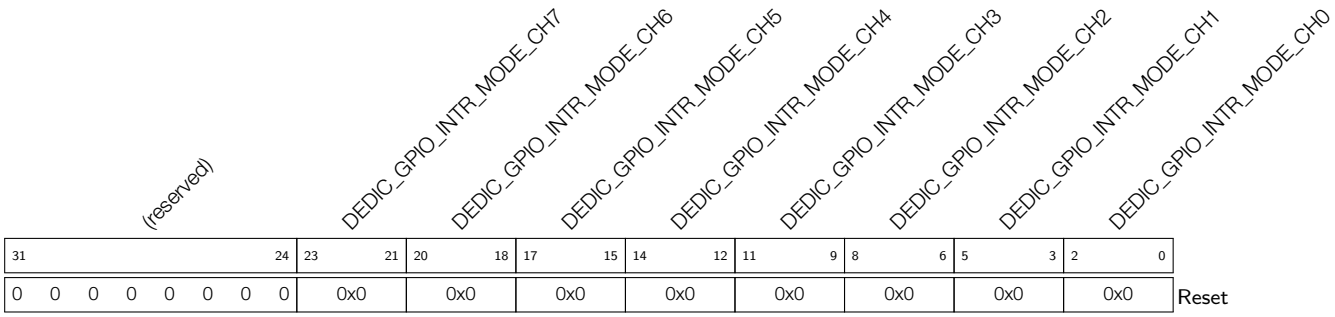
DEDIC_GPIO_IN_DLY_CH4 配置 GPIO4 的输入延时。0：无延时。1：延时一个时钟周期。2：延时两个时钟周期。3：延时三个时钟周期。(读/写)

DEDIC_GPIO_IN_DLY_CH5 配置 GPIO5 的输入延时。0：无延时。1：延时一个时钟周期。2：延时两个时钟周期。3：延时三个时钟周期。(读/写)

DEDIC_GPIO_IN_DLY_CH6 配置 GPIO6 的输入延时。0：无延时。1：延时一个时钟周期。2：延时两个时钟周期。3：延时三个时钟周期。(读/写)

DEDIC_GPIO_IN_DLY_CH7 配置 GPIO7 的输入延时。0：无延时。1：延时一个时钟周期。2：延时两个时钟周期。3：延时三个时钟周期。(读/写)

Register 5.45: DEDIC_GPIO_INTR_RCGN_REG (0x001C)



DEDIC_GPIO_INTR_MODE_CH0 配置通道 0 中断产生的模式。0/1：不产生中断。2：低电平触发。3：高电平触发。4：下降沿触发。5：上升沿触发。6/7：上升沿和下降沿触发。（读/写）

DEDIC_GPIO_INTR_MODE_CH1 配置通道 1 中断产生的模式。0/1：不产生中断。2：低电平触发。3：高电平触发。4：下降沿触发。5：上升沿触发。6/7：上升沿和下降沿触发。（读/写）

DEDIC_GPIO_INTR_MODE_CH2 配置通道 2 中断产生的模式。0/1：不产生中断。2：低电平触发。3：高电平触发。4：下降沿触发。5：上升沿触发。6/7：上升沿和下降沿触发。（读/写）

DEDIC_GPIO_INTR_MODE_CH3 配置通道 3 中断产生的模式。0/1：不产生中断。2：低电平触发。3：高电平触发。4：下降沿触发。5：上升沿触发。6/7：上升沿和下降沿触发。（读/写）

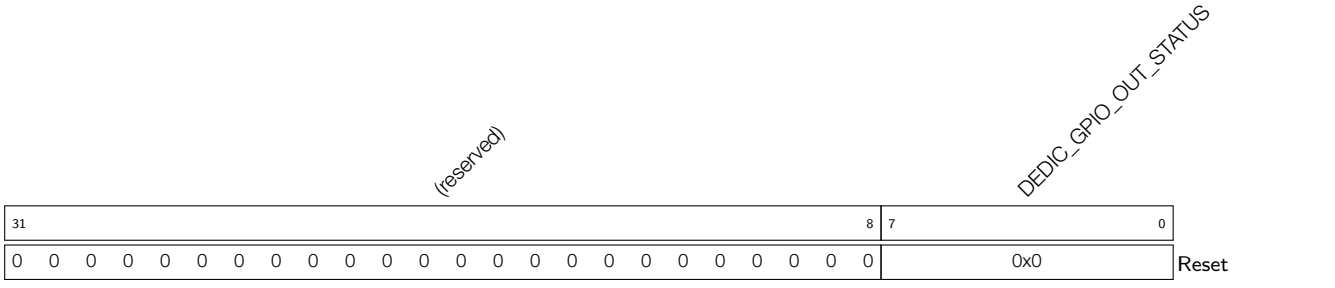
DEDIC_GPIO_INTR_MODE_CH4 配置通道 4 中断产生的模式。0/1：不产生中断。2：低电平触发。3：高电平触发。4：下降沿触发。5：上升沿触发。6/7：上升沿和下降沿触发。（读/写）

DEDIC_GPIO_INTR_MODE_CH5 配置通道 5 中断产生的模式。0/1：不产生中断。2：低电平触发。3：高电平触发。4：下降沿触发。5：上升沿触发。6/7：上升沿和下降沿触发。（读/写）

DEDIC_GPIO_INTR_MODE_CH6 配置通道 6 中断产生的模式。0/1：不产生中断。2：低电平触发。3：高电平触发。4：下降沿触发。5：上升沿触发。6/7：上升沿和下降沿触发。（读/写）

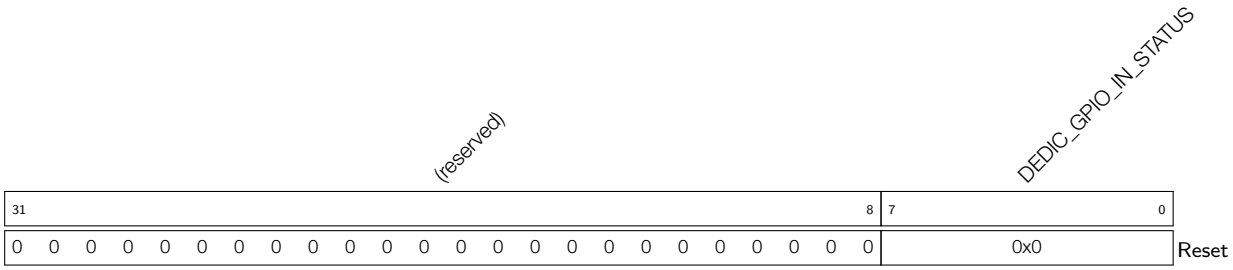
DEDIC_GPIO_INTR_MODE_CH7 配置通道 7 中断产生的模式。0/1：不产生中断。2：低电平触发。3：高电平触发。4：下降沿触发。5：上升沿触发。6/7：上升沿和下降沿触发。（读/写）

Register 5.46: DEDIC_GPIO_OUT_SCAN_REG (0x000C)



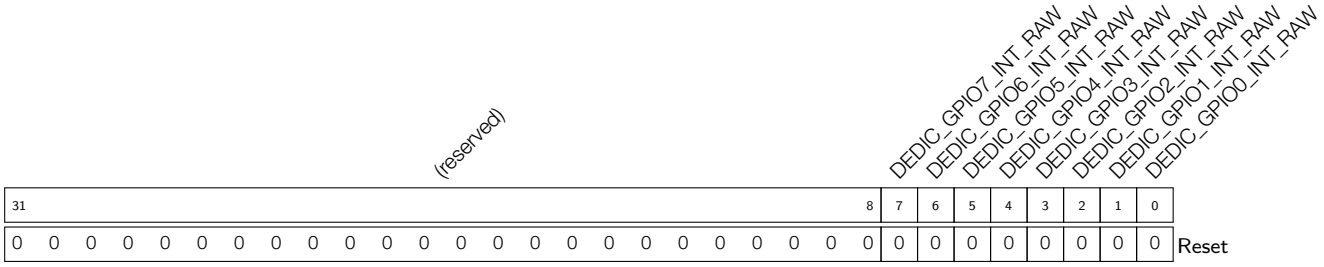
DEDIC_GPIO_OUT_STATUS 经 DEDIC_GPIO_OUT_DRT_REG、DEDIC_GPIO_OUT_MSK_REG 和 DEDIC_GPIO_OUT_IDV_REG 配置后，GPIO 的输出值。（只读）

Register 5.47: DEDIC_GPIO_IN_SCAN_REG (0x0018)



DEDIC_GPIO_IN_STATUS 经 DEDIC_GPIO_IN_DLY_REG 配置后，GPIO 的输入值。（只读）

Register 5.48: DEDIC_GPIO_INTR_RAW_REG (0x0020)



DEDIC_GPIO0_INT_RAW 专用 GPIO0 根据 DEDIC_GPIO_INTR_RCGN_REG 的配置，如果发生电平变化或沿变化，则该原始中断位将反转为高电平。（只读）

DEDIC_GPIO1_INT_RAW 专用 GPIO1 根据 DEDIC_GPIO_INTR_RCGN_REG 的配置，如果发生电平变化或沿变化，则该原始中断位将反转为高电平。（只读）

DEDIC_GPIO2_INT_RAW 专用 GPIO2 根据 DEDIC_GPIO_INTR_RCGN_REG 的配置，如果发生电平变化或沿变化，则该原始中断位将反转为高电平。（只读）

DEDIC_GPIO3_INT_RAW 专用 GPIO3 根据 DEDIC_GPIO_INTR_RCGN_REG 的配置，如果发生电平变化或沿变化，则该原始中断位将反转为高电平。（只读）

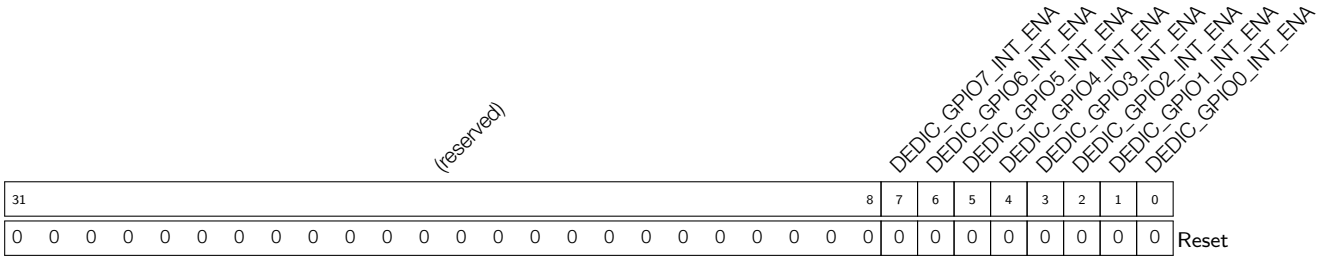
DEDIC_GPIO4_INT_RAW 专用 GPIO4 根据 DEDIC_GPIO_INTR_RCGN_REG 的配置，如果发生电平变化或沿变化，则该原始中断位将反转为高电平。（只读）

DEDIC_GPIO5_INT_RAW 专用 GPIO5 根据 DEDIC_GPIO_INTR_RCGN_REG 的配置，如果发生电平变化或沿变化，则该原始中断位将反转为高电平。（只读）

DEDIC_GPIO6_INT_RAW 专用 GPIO6 根据 DEDIC_GPIO_INTR_RCGN_REG 的配置，如果发生电平变化或沿变化，则该原始中断位将反转为高电平。（只读）

DEDIC_GPIO7_INT_RAW 专用 GPIO7 根据 DEDIC_GPIO_INTR_RCGN_REG 的配置，如果发生电平变化或沿变化，则该原始中断位将反转为高电平。（只读）

Register 5.49: DEDIC_GPIO_INTR_RLS_REG (0x0024)



DEDIC_GPIO0_INT_ENA DEDIC_GPIO0_INT_ST 的使能位。（读/写）

DEDIC_GPIO1_INT_ENA DEDIC_GPIO1_INT_ST 的使能位。（读/写）

DEDIC_GPIO2_INT_ENA DEDIC_GPIO2_INT_ST 的使能位。（读/写）

DEDIC_GPIO3_INT_ENA DEDIC_GPIO3_INT_ST 的使能位。（读/写）

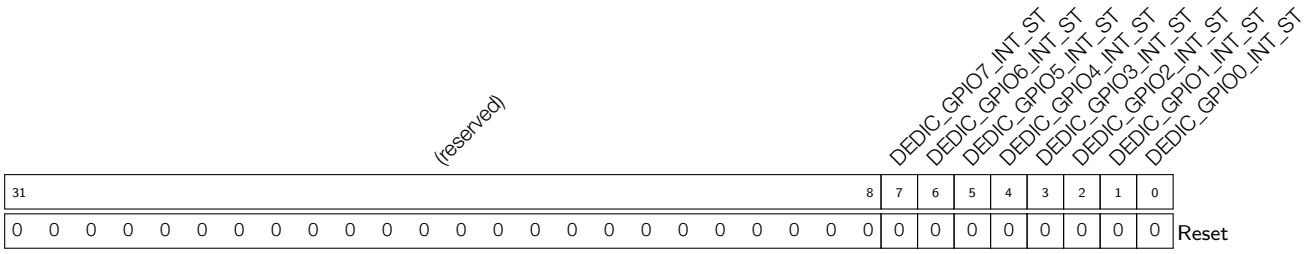
DEDIC_GPIO4_INT_ENA DEDIC_GPIO4_INT_ST 的使能位。（读/写）

DEDIC_GPIO5_INT_ENA DEDIC_GPIO5_INT_ST 的使能位。（读/写）

DEDIC_GPIO6_INT_ENA DEDIC_GPIO6_INT_ST 的使能位。（读/写）

DEDIC_GPIO7_INT_ENA DEDIC_GPIO7_INT_ST 的使能位。（读/写）

Register 5.50: DEDIC_GPIO_INTR_ST_REG (0x0028)



DEDIC_GPIO0_INT_ST DEDIC_GPIO0_INT_RAW 的状态位，仅在 DEDIC_GPIO0_INT_ENA = 1 时有效。（只读）

DEDIC_GPIO1_INT_ST DEDIC_GPIO1_INT_RAW 的状态位，仅在 DEDIC_GPIO1_INT_ENA = 1 时有效。（只读）

DEDIC_GPIO2_INT_ST DEDIC_GPIO2_INT_RAW 的状态位，仅在 DEDIC_GPIO2_INT_ENA = 1 时有效。（只读）

DEDIC_GPIO3_INT_ST DEDIC_GPIO3_INT_RAW 的状态位，仅在 DEDIC_GPIO3_INT_ENA = 1 时有效。（只读）

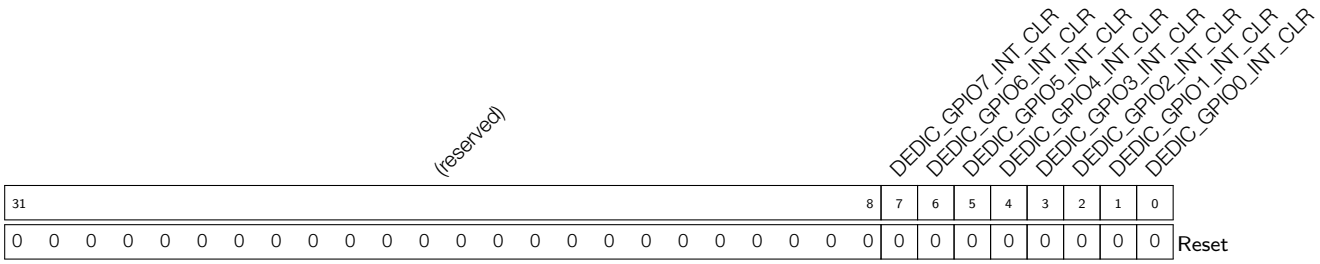
DEDIC_GPIO4_INT_ST DEDIC_GPIO4_INT_RAW 的状态位，仅在 DEDIC_GPIO4_INT_ENA = 1 时有效。（只读）

DEDIC_GPIO5_INT_ST DEDIC_GPIO5_INT_RAW 的状态位，仅在 DEDIC_GPIO5_INT_ENA = 1 时有效。（只读）

DEDIC_GPIO6_INT_ST DEDIC_GPIO6_INT_RAW 的状态位，仅在 DEDIC_GPIO6_INT_ENA = 1 时有效。（只读）

DEDIC_GPIO7_INT_ST DEDIC_GPIO7_INT_RAW 的状态位，仅在 DEDIC_GPIO7_INT_ENA = 1 时有效。（只读）

Register 5.51: DEDIC_GPIO_INTR_CLR_REG (0x002C)



DEDIC_GPIO0_INT_CLR 置位此位清除 DEDIC_GPIO0_INT_RAW 中断。（只写）

DEDIC_GPIO1_INT_CLR 置位此位清除 DEDIC_GPIO1_INT_RAW 中断。（只写）

DEDIC_GPIO2_INT_CLR 置位此位清除 DEDIC_GPIO2_INT_RAW 中断。（只写）

DEDIC_GPIO3_INT_CLR 置位此位清除 DEDIC_GPIO3_INT_RAW 中断。（只写）

DEDIC_GPIO4_INT_CLR 置位此位清除 DEDIC_GPIO4_INT_RAW 中断。（只写）

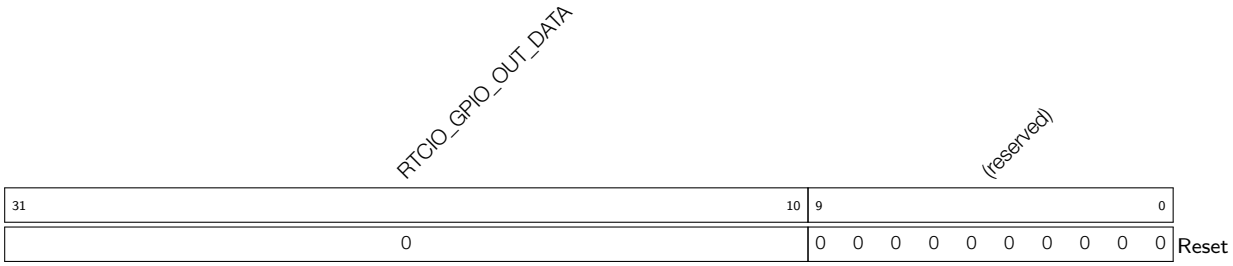
DEDIC_GPIO5_INT_CLR 置位此位清除 DEDIC_GPIO5_INT_RAW 中断。（只写）

DEDIC_GPIO6_INT_CLR 置位此位清除 DEDIC_GPIO6_INT_RAW 中断。（只写）

DEDIC_GPIO7_INT_CLR 置位此位清除 DEDIC_GPIO7_INT_RAW 中断。（只写）

5.15.5 RTC IO MUX 寄存器

Register 5.52: RTCIO_RTC_GPIO_OUT_REG (0x0000)

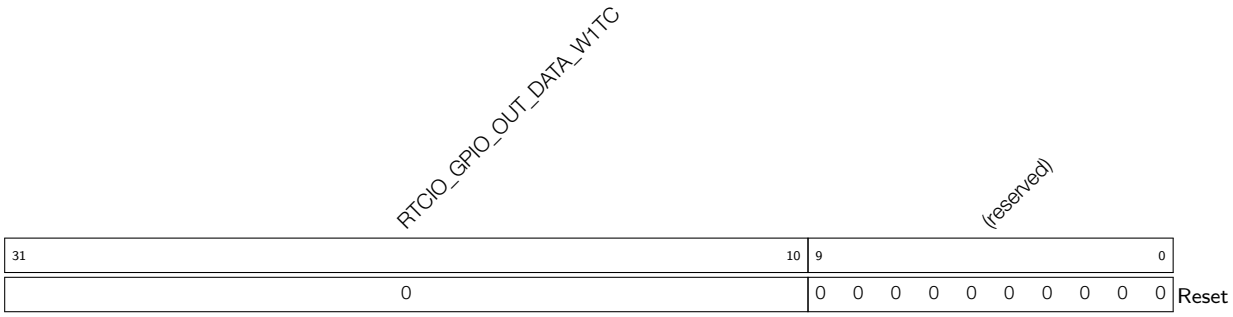


RTCIO_GPIO_OUT_DATA GPIO0 ~ 21 输出寄存器。bit10 对应 GPIO0，bit11 对应 GPIO1，以此类推。（读/写）

Register 5.53: RTCIO_RTC_GPIO_OUT_W1TS_REG (0x0004)

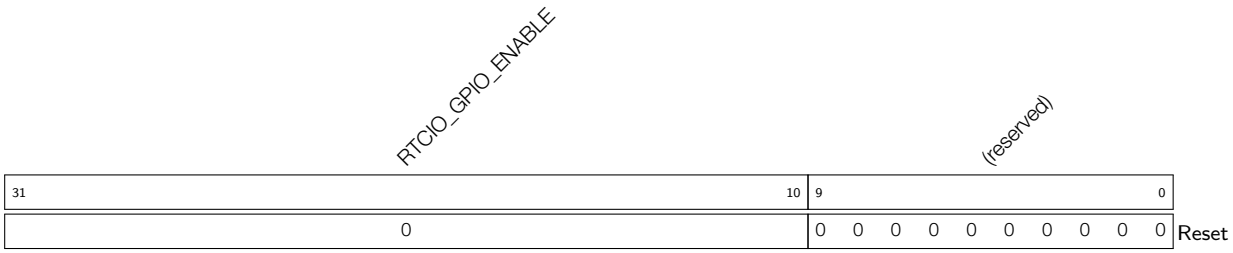
RTCIO_GPIO_OUT_DATA_W1TS GPIO0 ~ 21 输出置位寄存器。每一位置 1，RTCIO_RTC_GPIO_OUT_REG 中相应位也置 1。注：推荐使用此寄存器来置位 RTCIO_RTC_GPIO_OUT_REG。(只写)

Register 5.54: RTCIO_RTC_GPIO_OUT_W1TC_REG (0x0008)



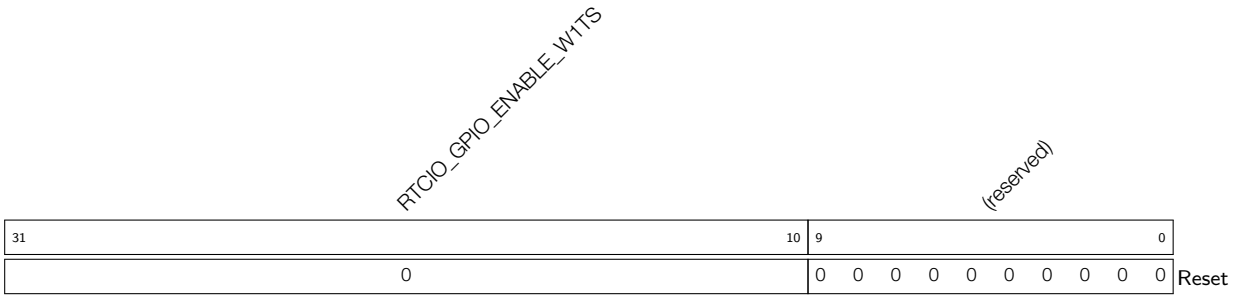
RTCIO_GPIO_OUT_DATA_W1TC GPIO0 ~ 21 输出清零寄存器。每一位置 1，则 RTCIO_RTC_GPIO_OUT_REG 中相应位将被清零。注：推荐使用此寄存器来清零 RTCIO_RTC_GPIO_OUT_REG。(只写)

Register 5.55: RTCIO_RTC_GPIO_ENABLE_REG (0x000C)



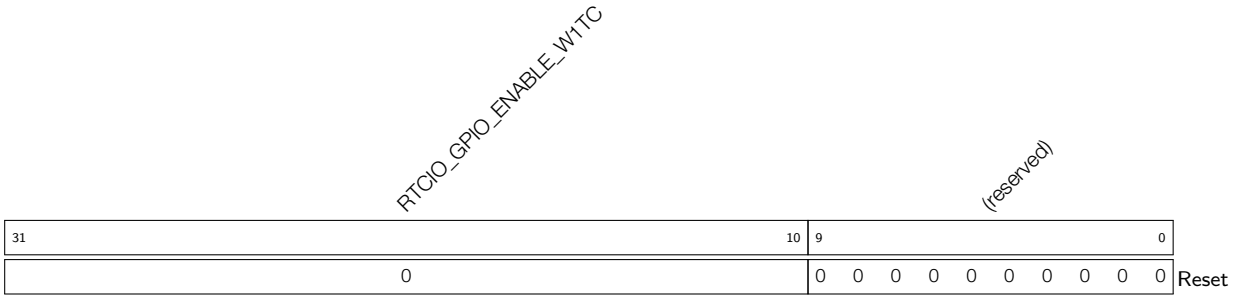
RTCIO_GPIO_ENABLE GPIO0 ~ 21 输出使能。bit10 对应 GPIO0, bit11 对应 GPIO1，以此类推。此位置 1，即该 GPIO pad 为输出。(读/写)

Register 5.56: RTCIO_RTC_GPIO_ENABLE_W1TS_REG (0x0010)



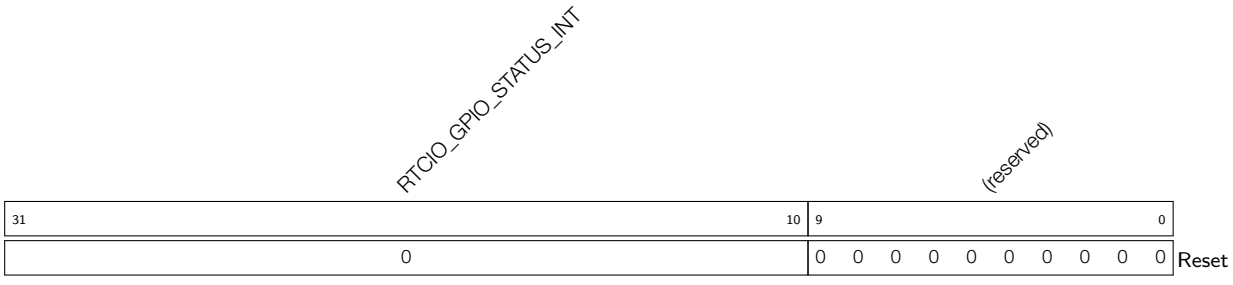
RTCIO_GPIO_ENABLE_W1TS GPIO0 ~ 21 输出使能置位寄存器。每一位置 1，则 RTCIO_RTC_GPIO_ENABLE_REG 中相应位也将置 1。注：推荐使用此寄存器来置位 RTCIO_RTC_GPIO_ENABLE_REG。(只写)

Register 5.57: RTCIO_RTC_GPIO_ENABLE_W1TC_REG (0x0014)



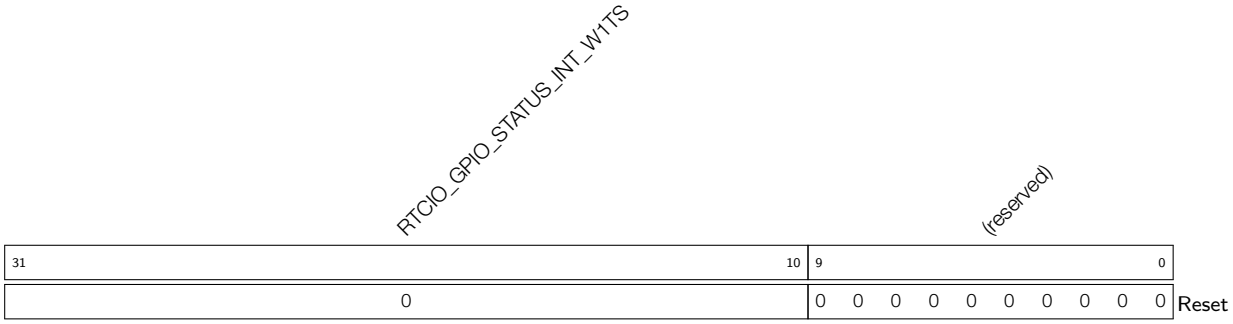
RTCIO_GPIO_ENABLE_W1TC GPIO0 ~ 21 输出使能清零寄存器。每一位置 1，则 RTCIO_RTC_GPIO_ENABLE_REG 中相应位将被清零。注：推荐使用此寄存器来清零 RTCIO_RTC_GPIO_ENABLE_REG。(只写)

Register 5.58: RTCIO_RTC_GPIO_STATUS_REG (0x0018)



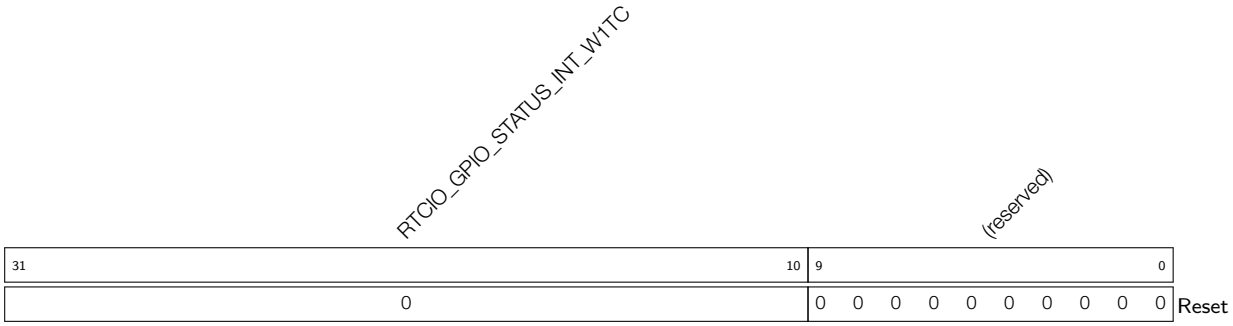
RTCIO_GPIO_STATUS_INT GPIO0 ~ 21 中断状态寄存器。bit10 对应 GPIO0，bit11 对应 GPIO1，以此类推。此寄存器应同时与 RTCIO_RTC_GPIO_PIN*n*_REG 寄存器中的 RTCIO_RTC_GPIO_PIN*n*_INT_TYPE 中断类型配合使用。0：代表没有中断；1：代表有相应中断。(读/写)

Register 5.59: RTCIO_RTC_GPIO_STATUS_W1TS_REG (0x001C)



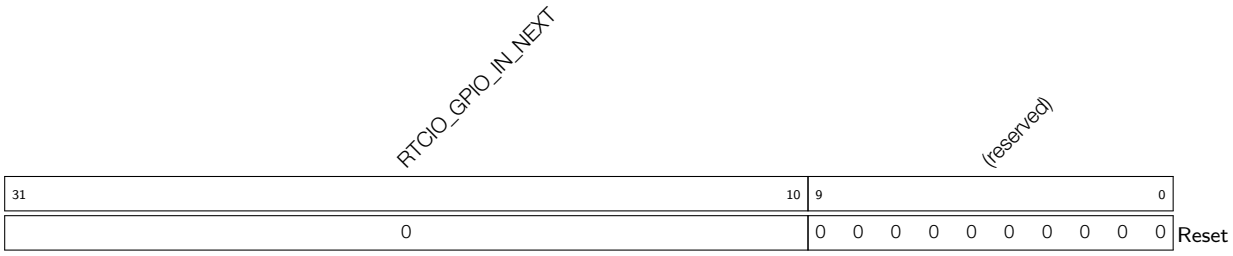
RTCIO_GPIO_STATUS_INT_W1TS GPIO0 ~ 21 中断状态置位寄存器。每一位置 1，则 RTCIO_GPIO_STATUS_INT 中相应位也将置 1。注：推荐使用此寄存器来置位 RTCIO_GPIO_STATUS_INT。(只写)

Register 5.60: RTCIO_RTC_GPIO_STATUS_W1TC_REG (0x0020)



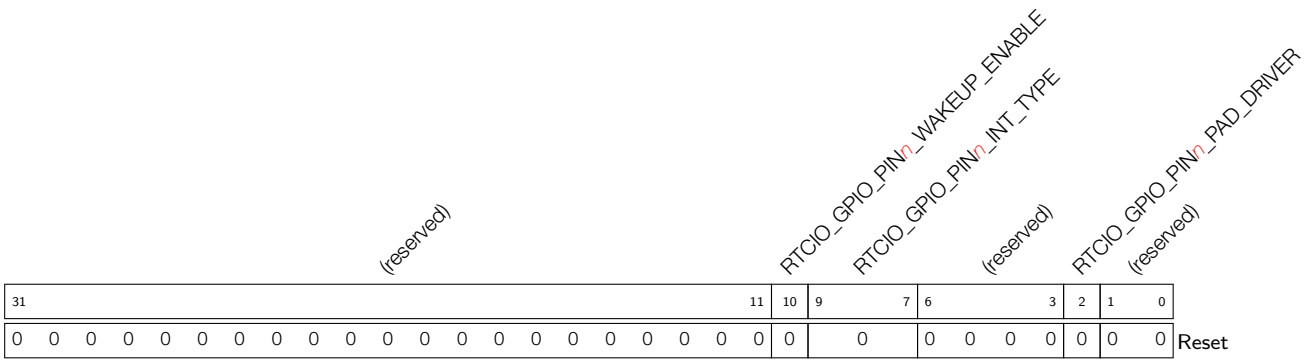
RTCIO_GPIO_STATUS_INT_W1TC GPIO0 ~ 21 中断状态清零寄存器。每一位置 1，则 RTCIO_GPIO_STATUS_INT 中的相应位也将清零。注：推荐使用此寄存器来清零 RTCIO_GPIO_STATUS_INT。(只写)

Register 5.61: RTCIO_RTC_GPIO_IN_REG (0x0024)



RTCIO_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。(只读)

Register 5.62: RTCIO_RTC_GPIO_PIN*n*_REG (*n*: 0-21) (0x0028+4n*)**



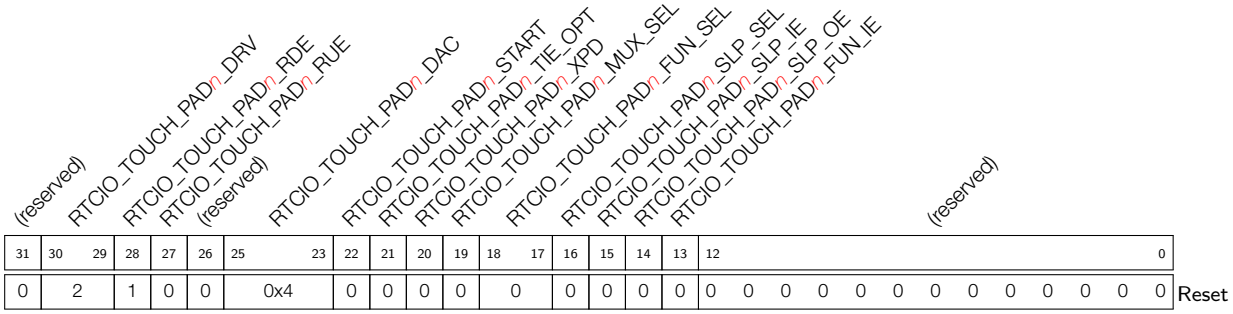
RTCIO_GPIO_PIN*n*_PAD_DRIVER Pad 驱动选择寄存器。0：正常输出；1：开漏输出。(读/写)

RTCIO_GPIO_PIN*n*_INT_TYPE GPIO 中断类型选择寄存器。(读/写)

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

RTCIO_GPIO_PIN*n*_WAKEUP_ENABLE GPIO 唤醒使能。只能将 ESP32-S2 从 Light-sleep 中唤醒。(读/写)

Register 5.63: RTCIO_TOUCH_PAD*n*_REG (*n*: 0-14) (0x0084+4n*)**



RTCIO_TOUCH_PAD*n*_FUN_IE 工作模式下输入使能。(读/写)

RTCIO_TOUCH_PAD*n*_SLP_OE 睡眠模式下输出使能。(读/写)

RTCIO_TOUCH_PAD*n*_SLP_IE 睡眠模式下输入使能。(读/写)

RTCIO_TOUCH_PAD*n*_SLP_SEL 0：无睡眠模式；1：使能睡眠模式。(读/写)

RTCIO_TOUCH_PAD*n*_FUN_SEL Function 选择 (读/写)

RTCIO_TOUCH_PAD*n*_MUX_SEL 连接 RTC pad 输入与数字 pad 输入，可以置 0。(读/写)

RTCIO_TOUCH_PAD*n*_XPD 触摸传感器上电。(读/写)

RTCIO_TOUCH_PAD*n*_TIE_OPT 默认触摸传感器初始电压位。0：拉高；1：拉低。(读/写)

RTCIO_TOUCH_PAD*n*_START 启动触摸传感器。(读/写)

RTCIO_TOUCH_PAD*n*_DAC 触摸传感器斜率控制。每个触摸 pad 为 3-bit，默认为 0x4。(读/写)

RTCIO_TOUCH_PAD*n*_RUE 使能 pad 上拉。1：内部上拉使能；0：内部上拉关闭。(读/写)

RTCIO_TOUCH_PAD*n*_RDE 使能 pad 下拉。1：内部下拉使能；0：内部下拉关闭。(读/写)

RTCIO_TOUCH_PAD*n*_DRV 选择 pad 的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

Register 5.64: RTCIO_XTAL_32P_PAD_REG (0x00C0)

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

RTCIO_X32P_FUN_IE 使能工作模式下 pad 输入。（读/写）

RTCIO_X32P_SLP_OE 使能睡眠模式下 pad 输出。（读/写）

RTCIO_X32P_SLP_IE 使能睡眠模式下 pad 输入。（读/写）

RTCIO_X32P_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。（读/写）

RTCIO_X32P_FUN_SEL Function 选择。（读/写）

RTCIO_X32P_MUX_SEL 1：选择使用 RTC GPIO；0：选择使用数字 GPIO。（读/写）

RTCIO_X32P_RUE Pad 上拉使能。1：内部上拉使能；0：内部上拉关闭。（读/写）

RTCIO_X32P_RDE Pad 下拉使能。1：内部下拉使能；0：内部下拉关闭。（读/写）

RTCIO_X32P_DRV 选择 pad 的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。（读/写）

Register 5.65: RTCIO_XTAL_32N_PAD_REG (0x00C4)

RTCIO_X32N_FUN_IE 工作模式下输入使能。(读/写)

RTCIO_X32N_SLP_OE 睡眠模式下输出使能。(读/写)

RTCIO_X32N_SLP_IE 睡眠模式下输入使能。(读/写)

RTCIO_X32N_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(读/写)

RTCIO_X32N_FUN_SEL Function 选择。(读/写)

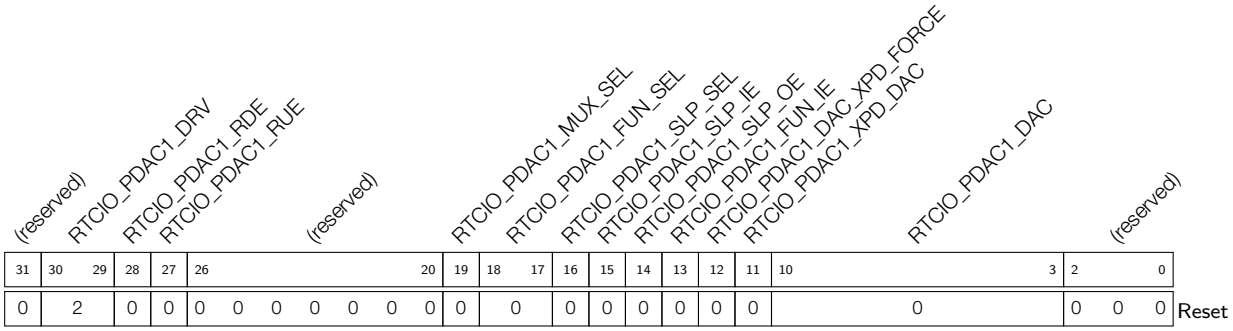
RTCIO_X32N_MUX_SEL 1：选择使用 RTC GPIO；0：选择使用数字 GPIO。(读/写)

RTCIO_X32N_RUE Pad 上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

RTCIO_X32N_RDE Pad 下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

RTCIO_X32N_DRV 选择 pad 的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

Register 5.66: RTCIO_PAD_DAC1_REG (0x00C8)



RTCIO_PDAC1_DAC RTCIO_PDAC1_DAC_XPD_FORCE 置 1 时，配置 DAC_1 输出。(读/写)

RTCIO_PDAC1_XPD_DAC RTCIO_PDAC1_DAC_XPD_FORCE 置 1 时，1：使能 DAC_1 输出；0：关闭 DAC_1 输出。(读/写)

RTCIO_PDAC1_DAC_XPD_FORCE 1：选择使用 RTCIO_PDAC1_XPD_DAC 控制 DAC_1 输出；0：选择使用 SAR ADC FSM 控制 DAC_1 输出。(读/写)

RTCIO_PDAC1_FUN_IE 工作模式下输入使能。(读/写)

RTCIO_PDAC1_SLP_OE 睡眠模式下输出使能。(读/写)

RTCIO_PDAC1_SLP_IE 睡眠模式下输入使能。(读/写)

RTCIO_PDAC1_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(读/写)

RTCIO_PDAC1_FUN_SEL DAC_1 功能选择。(读/写)

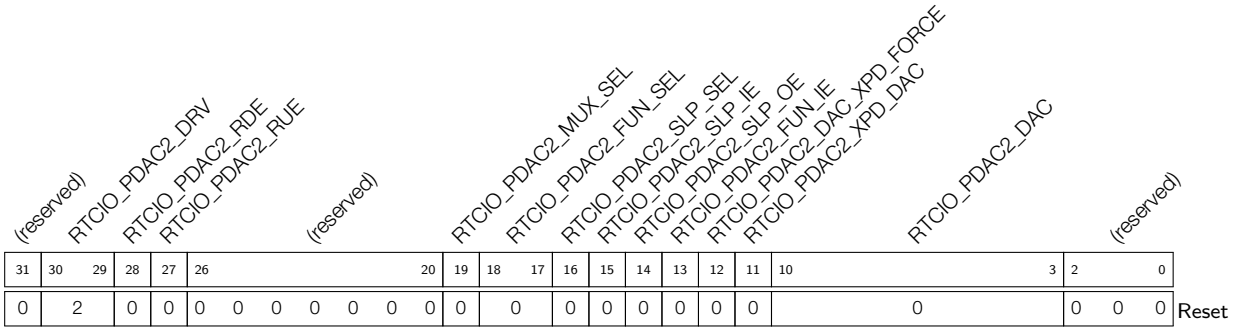
RTCIO_PDAC1_MUX_SEL 1：选择使用 RTC GPIO；0：选择使用数字 GPIO。(读/写)

RTCIO_PDAC1_RUE Pad 上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

RTCIO_PDAC1_RDE Pad 下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

RTCIO_PDAC1_DRV 选择 pad 的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

Register 5.67: RTCIO_PAD_DAC2_REG (0x00CC)



RTCIO_PDAC2_DAC RTCIO_PDAC2_DAC_XPD_FORCE 置 1 时，配置 DAC_2 输出。(读/写)

RTCIO_PDAC2_XPD_DAC RTCIO_PDAC2_DAC_XPD_FORCE 置 1 时，1：使能 DAC_2 输出；0：关闭 DAC_2 输出。(读/写)

RTCIO_PDAC2_DAC_XPD_FORCE 1：使用 RTCIO_PDAC2_XPD_DAC 控制 DAC_2 输出；0：使用 SAR ADC FSM 控制 DAC_2 输出。(读/写)

RTCIO_PDAC2_FUN_IE 工作模式下输入使能。(读/写)

RTCIO_PDAC2_SLP_OE 睡眠模式下输出使能。(读/写)

RTCIO_PDAC2_SLP_IE 睡眠模式下输入使能。(读/写)

RTCIO_PDAC2_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(读/写)

RTCIO_PDAC2_FUN_SEL DAC_2 功能选择。(读/写)

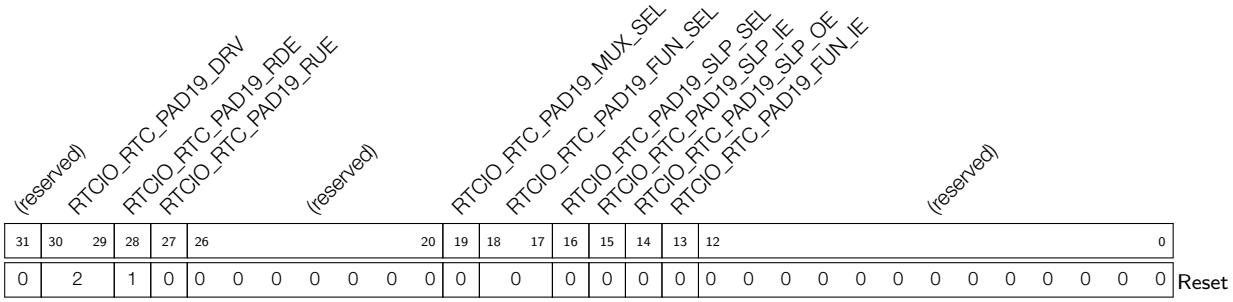
RTCIO_PDAC2_MUX_SEL 1：选择使用 RTC GPIO；0：选择使用数字 GPIO。(读/写)

RTCIO_PDAC2_RUE Pad 上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

RTCIO_PDAC2_RDE Pad 下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

RTCIO_PDAC2_DRV 选择 pad 的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

Register 5.68: RTCIO_RTC_PAD19_REG (0x00D0)



RTCIO_RTC_PAD19_FUN_IE 工作模式下输入使能。(读/写)

RTCIO_RTC_PAD19_SLP_OE 睡眠模式下输出使能。(读/写)

RTCIO_RTC_PAD19_SLP_IE 睡眠模式下输入使能。(读/写)

RTCIO_RTC_PAD19_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(读/写)

RTCIO_RTC_PAD19_FUN_SEL 功能选择。(读/写)

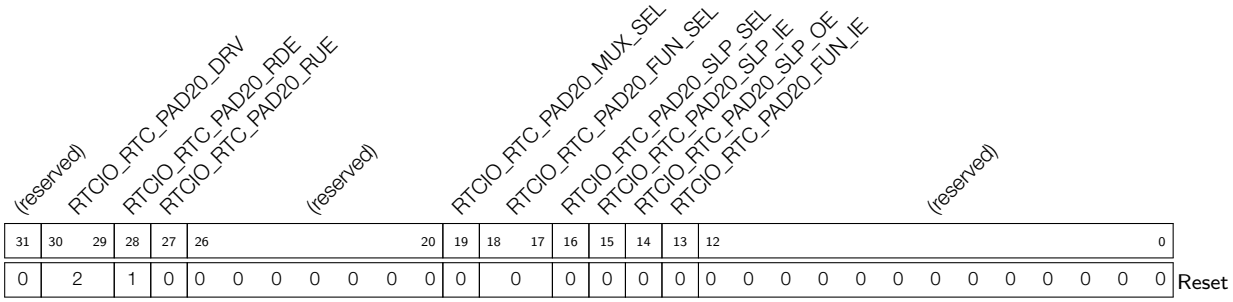
RTCIO_RTC_PAD19_MUX_SEL 1：选择使用 RTC GPIO；2：选择使用数字 GPIO。(读/写)

RTCIO_RTC_PAD19_RUE Pad 上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

RTCIO_RTC_PAD19_RDE Pad 下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

RTCIO_RTC_PAD19_DRV 选择 Pad 的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

Register 5.69: RTCIO_RTC_PAD20_REG (0x00D4)



RTCIO_RTC_PAD20_FUN_IE 工作模式下输入使能。(读/写)

RTCIO_RTC_PAD20_SLP_OE 睡眠模式下输出使能。(读/写)

RTCIO_RTC_PAD20_SLP_IE 睡眠模式下输入使能。(读/写)

RTCIO_RTC_PAD20_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(读/写)

RTCIO_RTC_PAD20_FUN_SEL 功能选择。(读/写)

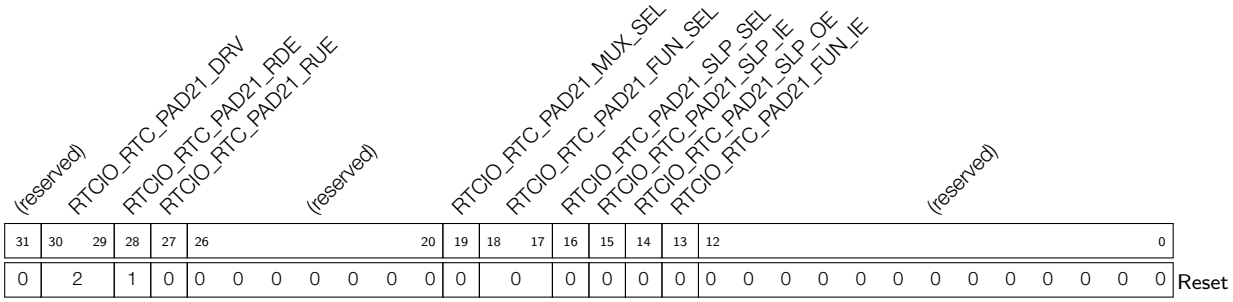
RTCIO_RTC_PAD20_MUX_SEL 1：选择使用 RTC GPIO；0：选择使用数字 GPIO。(读/写)

RTCIO_RTC_PAD20_RUE Pad 上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

RTCIO_RTC_PAD20_RDE Pad 下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

RTCIO_RTC_PAD20_DRV 选择 Pad 的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

Register 5.70: RTCIO_RTC_PAD21_REG (0x00D8)



RTCIO_RTC_PAD21_FUN_IE 工作模式下输入使能。(读/写)

RTCIO_RTC_PAD21_SLP_OE 睡眠模式下输出使能。(读/写)

RTCIO_RTC_PAD21_SLP_IE 睡眠模式下输入使能。(读/写)

RTCIO_RTC_PAD21_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(读/写)

RTCIO_RTC_PAD21_FUN_SEL 功能选择。(读/写)

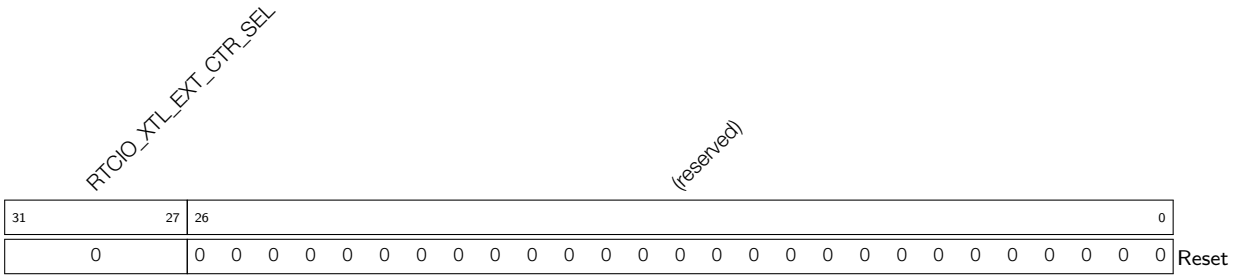
RTCIO_RTC_PAD21_MUX_SEL 1：选择使用 RTC GPIO；0：选择使用数字 GPIO。(读/写)

RTCIO_RTC_PAD21_RUE Pad 上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

RTCIO_RTC_PAD21_RDE Pad 下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

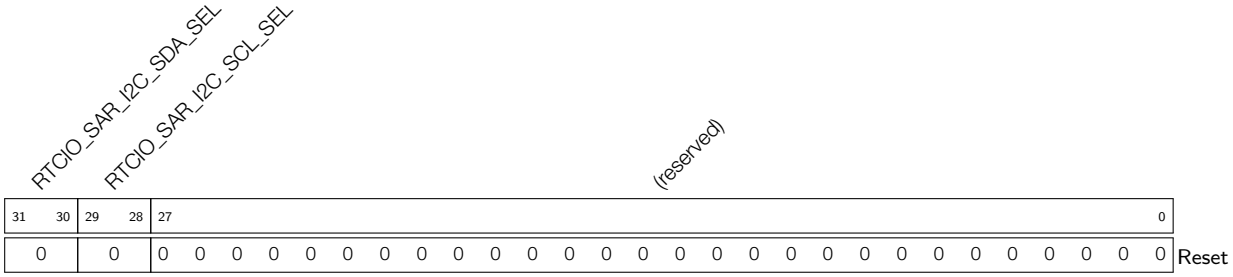
RTCIO_RTC_PAD21_DRV 选择 Pad 的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

Register 5.71: RTCIO_XTL_EXT_CTR_REG (0x00E0)



RTCIO_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。(读/写)

Register 5.72: RTCIO_SAR_I2C_IO_REG (0x00E4)



RTCIO_SAR_I2C_SCL_SEL 选择 RTC I2C SCL 信号连接的 pad。0：选择 TOUCH_PAD0；1：选择 TOUCH_PAD2。（读/写）

RTCIO_SAR_I2C_SDA_SEL 选择 RTC I2C SDA 信号连接的 pad。0：选择 TOUCH_PAD1；1：选择 TOUCH_PAD3。（读/写）

6. 系统寄存器

6.1 概述

ESP32-S2 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-S2 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。

6.2 主要特性

ESP32-S2 的系统寄存器可用于控制以下外设和模块：

- 系统和存储器
- 复位和时钟
- 中断矩阵
- eFuse 控制器
- 低功耗管理寄存器
- 外设时钟门控和复位

6.3 功能描述

6.3.1 系统和存储器寄存器

以下系统寄存器用于系统和存储器的配置，例如 cache 配置和存储器掉电控制等。更多信息，请见章节 *1 系统和存储器*。

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

ROM 功耗控制

寄存器 SYSTEM_ROM_CTRL_0_REG 和 SYSTEM_ROM_CTRL_1_REG 用以控制 ESP32-S2 内部 ROM 存储的功耗，具体来说：

- 寄存器 SYSTEM_ROM_CTRL_0_REG 中 SYSTEM_ROM_FO 的相应位可用以控制 ROM 不同 block 的时钟门控。
- 寄存器 SYSTEM_ROM_CTRL_1_REG 中 SYSTEM_ROM_FORCE_PD 的相应位可用以控制 ROM 不同 block 的掉电。
- 寄存器 SYSTEM_ROM_CTRL_1_REG 中 SYSTEM_ROM_FORCE_PU 的相应位可用以控制 ROM 不同 block 的上电。

更多有关 ROM 控制位的对应关系，请见下方表 31。

表 31: ROM 控制位

ROM	低地址 1	高地址 1	低地址 2	高地址 2	控制位
Block0	0x4000_0000	0x4000_FFFF	-	-	Bit0
Block1	0x4001_2000	0x4001_FFFF	0x3FFA_0000	0x3FFA_FFFF	Bit1

SRAM 功耗控制

寄存器 SYSTEM_SRAM_CTRL_0_REG、SYSTEM_SRAM_CTRL_1_REG 和 SYSTEM_SRAM_CTRL_2_REG 用以控制 ESP32-S2 内部 SRAM 存储的功耗，具体来说：

- 寄存器 SYSTEM_SRAM_CTRL_0_REG 中 SYSTEM_SRAM_FO 的相应位可用以控制 SRAM 不同 block 的时钟门控。
- 寄存器 SYSTEM_SRAM_CTRL_1_REG 中 SYSTEM_SRAM_FORCE_PD 的相应位可用以控制 SRAM 不同 block 的掉电。
- 寄存器 SYSTEM_SRAM_CTRL_2_REG 中 SYSTEM_SRAM_FORCE_PU 的相应位可用以控制 SRAM 不同 block 的上电。

更多有关 SRAM 控制位的对应关系，请见下方表 32。

表 32: SRAM 控制位

SRAM	低地址 1	高地址 1	低地址 2	高地址 2	控制位
Block0	0x4002_0000	0x4002_1FFF	0x3FFB_0000	0x3FFB_1FFF	Bit0
Block1	0x4002_2000	0x4002_3FFF	0x3FFB_2000	0x3FFB_3FFF	Bit1
Block2	0x4002_4000	0x4002_5FFF	0x3FFB_4000	0x3FFB_5FFF	Bit2
Block3	0x4002_6000	0x4002_7FFF	0x3FFB_6000	0x3FFB_7FFF	Bit3
Block4	0x4002_8000	0x4002_BFFF	0x3FFB_8000	0x3FFB_BFFF	Bit4
Block5	0x4002_C000	0x4002_FFFF	0x3FFB_C000	0x3FFB_FFFF	Bit5
Block6	0x4003_0000	0x4003_3FFF	0x3FFC_0000	0x3FFC_3FFF	Bit6
Block7	0x4003_4000	0x4003_7FFF	0x3FFC_4000	0x3FFC_7FFF	Bit7
Block8	0x4003_8000	0x4003_BFFF	0x3FFC_8000	0x3FFC_BFFF	Bit8
Block9	0x4003_C000	0x4003_FFFF	0x3FFC_C000	0x3FFC_FFFF	Bit9

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

6.3.2 复位和时钟寄存器

以下系统寄存器用于复位和时钟的配置。更多信息，请见章节 2 复位和时钟。

- SYSTEM_CPU_PER_CONF_REG
- SYSTEM_SYSCLK_CONF_REG
- SYSTEM_BT_LPCK_DIV_FRAC_REG

6.3.3 中断矩阵寄存器

以下系统寄存器用于产生中断矩阵中的 CPU 中断信号。更多信息，请见章节 4 中断矩阵。

- SYSTEM_CPU_INTR_FROM_CPU_0_REG
- SYSTEM_CPU_INTR_FROM_CPU_1_REG
- SYSTEM_CPU_INTR_FROM_CPU_2_REG
- SYSTEM_CPU_INTR_FROM_CPU_3_REG

6.3.4 JTAG 软件使能寄存器

以下系统寄存器用于撤销 eFuse 对 JTAG 的临时关闭功能。

- SYSTEM_JTAG_CTRL_0_REG
- SYSTEM_JTAG_CTRL_1_REG
- SYSTEM_JTAG_CTRL_2_REG
- SYSTEM_JTAG_CTRL_3_REG
- SYSTEM_JTAG_CTRL_4_REG
- SYSTEM_JTAG_CTRL_5_REG
- SYSTEM_JTAG_CTRL_6_REG
- SYSTEM_JTAG_CTRL_7_REG

6.3.5 低功耗管理寄存器

以下系统寄存器用于低功耗管理。

- SYSTEM_RTC_FASTMEM_CONFIG_REG
- SYSTEM_RTC_FASTMEM_CRC_REG

6.3.6 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，详见下方表 33。

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

表 33: 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]
CPU 外设	**SYSTEM_CPU_PERI_CLK_EN_REG**	**SYSTEM_CPU_PERI_RST_EN_REG**
DEDICATED GPIO	SYSTEM_CLK_EN_DEDICATED_GPIO	SYSTEM_RST_EN_DEDICATED_GPIO
外设	**SYSTEM_PERIP_CLK_EN0_REG**	**SYSTEM_PERIP_RST_EN0_REG**
Timers	SYSTEM_TIMERS_CLK_EN	SYSTEM_TIMERS_RST
Timer Group0	SYSTEM_TIMERGROUP_CLK_EN	SYSTEM_TIMERGROUP_RST
Timer Group1	SYSTEM_TIMERGROUP1_CLK_EN	SYSTEM_TIMERGROUP1_RST
System Timer	SYSTEM_SYSTIMER_CLK_EN	SYSTEM_SYSTIMER_RST
UART0	SYSTEM_UART_CLK_EN	SYSTEM_UART_RST
UART1	SYSTEM_UART1_CLK_EN	SYSTEM_UART1_RST
UART MEM	SYSTEM_UART_MEM_CLK_EN [4]	SYSTEM_UART_MEM_RST
SPI0, SPI1	SYSTEM_SPI01_CLK_EN	SYSTEM_SPI01_RST
SPI2	SYSTEM_SPI2_CLK_EN	SYSTEM_SPI2_RST
SPI3	SYSTEM_SPI3_DMA_CLK_EN	SYSTEM_SPI3_RST
SPI4	SYSTEM_SPI4_CLK_EN	SYSTEM_SPI4_RST
SPI2 DMA	SYSTEM_SPI2_DMA_CLK_EN	SYSTEM_SPI2_DMA_RST
SPI3 DMA	SYSTEM_SPI3_DMA_CLK_EN	SYSTEM_SPI3_DMA_RST
I2C0	SYSTEM_I2C_EXT0_CLK_EN	SYSTEM_I2C_EXT0_RST
I2C1	SYSTEM_I2C_EXT1_CLK_EN	SYSTEM_I2C_EXT1_RST
I2S0	SYSTEM_I2S0_CLK_EN	SYSTEM_I2S0_RST
I2S1	SYSTEM_I2S1_CLK_EN	SYSTEM_I2S1_RST
TWAI 控制器	SYSTEM_CAN_CLK_EN	SYSTEM_CAN_RST
UHCI0	SYSTEM_UHCI0_CLK_EN	SYSTEM_UHCI0_RST
UHCI1	SYSTEM_UHCI1_CLK_EN	SYSTEM_UHCI1_RST
USB	SYSTEM_USB_CLK_EN	SYSTEM_USB_RST
RMT	SYSTEM_RMT_CLK_EN	SYSTEM_RMT_RST

PCNT	SYSTEM_PCNT_CLK_EN	SYSTEM_PCNT_RST
PWM0	SYSTEM_PWM0_CLK_EN	SYSTEM_PWM0_RST
PWM1	SYSTEM_PWM1_CLK_EN	SYSTEM_PWM1_RST
PWM2	SYSTEM_PWM2_CLK_EN	SYSTEM_PWM2_RST
PWM3	SYSTEM_PWM3_CLK_EN	SYSTEM_PWM3_RST
LED_PWM 控制器	SYSTEM_LEDC_CLK_EN	SYSTEM_LEDC_RST
eFuse	SYSTEM_EFUSE_CLK_EN	SYSTEM_EFUSE_RST
APB SARADC	SYSTEM_APB_SARADC_CLK_EN	SYSTEM_APB_SARADC_RST
ADC2 ARB	SYSTEM_ADC2_ARB_CLK_EN	SYSTEM_ADC2_ARB_RST
WDG	SYSTEM_WDG_CLK_EN	SYSTEM_WDG_RST
加速器	**SYSTEM_PERIP_CLK_EN1_REG**	**SYSTEM_PERIP_RST_EN1_REG**
DMA	SYSTEM_CRYPTO_DMA_CLK_EN	SYSTEM_CRYPTO_DMA_RST [5]
HMAC	SYSTEM_CRYPTO_HMAC_CLK_EN	SYSTEM_CRYPTO_HMAC_RST [6]
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

说明：

1. 时钟控制寄存器置 1 打开对应时钟，置 0 关闭对应时钟。
2. 复位寄存器置 1 使能复位状态，对应外设进行复位，置 0 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。
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. 该 Crypto DMA 为 AES 加速器和 SHA 加速器所共用。
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。

6.4 基地址

用户可以通过两个不同的寄存器基地址访问系统寄存器，如表 34 所示。更多信息，请访问章节 1 系统和存储器。

表 34: 系统寄存器基地址

访问总线	基地址
PeriBUS1	0x3F4C0000

6.5 寄存器列表

请注意，这里的地址是相对于系统寄存器基地址的地址偏移量（相对地址）。请参阅章节 6.4 获取有关系统寄存器基地址的信息。

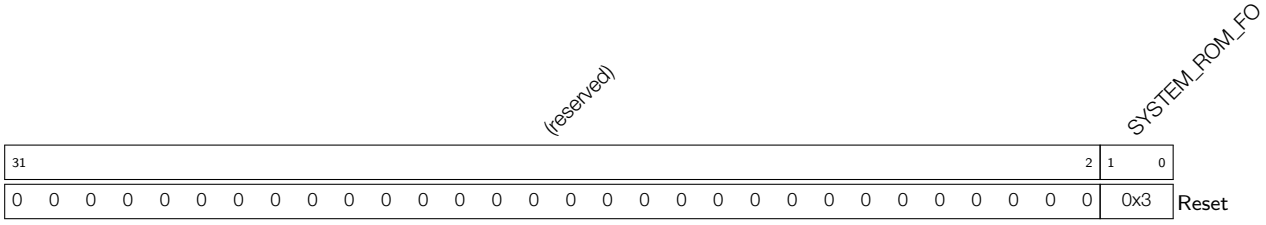
名称	描述	地址	访问权限
系统和存储器寄存器			
SYSTEM_ROM_CTRL_0_REG	系统 ROM 配置寄存器 0	0x0000	读写
SYSTEM_ROM_CTRL_1_REG	系统 ROM 配置寄存器 1	0x0004	读写
SYSTEM_SRAM_CTRL_0_REG	系统 SRAM 配置寄存器 0	0x0008	读写
SYSTEM_SRAM_CTRL_1_REG	系统 SRAM 配置寄存器 1	0x000C	读写
SYSTEM_SRAM_CTRL_2_REG	系统 SRAM 配置寄存器 2	0x0088	读写
SYSTEM_RSA_PD_CTRL_REG	RSA 存储掉电寄存器	0x0068	读写
SYSTEM_MEM_PD_MASK_REG	存储器掉电屏蔽寄存器（low-sleep 状态下）	0x003C	读写
SYSTEM_CACHE_CONTROL_REG	Cache 控制寄存器	0x0070	读写
SYSTEM_BUSTOEXTMEM_ENA_REG	EDMA 使能寄存器	0x006C	读写
SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG	外部设备加解密控制寄存器	0x0074	读写
复位和时钟寄存器			
SYSTEM_CPU_PER_CONF_REG	CPU 外设时钟配置寄存器	0x0018	读写
SYSTEM_SYSCLK_CONF_REG	系统时钟配置寄存器	0x008C	不定
SYSTEM_BT_LPCK_DIV_FRAC_REG	低功耗时钟分频分数配置寄存器	0x0054	读写
中断矩阵寄存器			
SYSTEM_CPU_INTR_FROM_CPU_0_REG	CPU 中断控制寄存器 0	0x0058	读写
SYSTEM_CPU_INTR_FROM_CPU_1_REG	CPU 中断控制寄存器 1	0x005C	读写
SYSTEM_CPU_INTR_FROM_CPU_2_REG	CPU 中断控制寄存器 2	0x0060	读写
SYSTEM_CPU_INTR_FROM_CPU_3_REG	CPU 中断控制寄存器 3	0x0064	读写
JTAG 软件使能寄存器			
SYSTEM_JTAG_CTRL_0_REG	JTAG 配置寄存器 0	0x001C	只写
SYSTEM_JTAG_CTRL_1_REG	JTAG 配置寄存器 1	0x0020	只写
SYSTEM_JTAG_CTRL_2_REG	JTAG 配置寄存器 2	0x0024	只写
SYSTEM_JTAG_CTRL_3_REG	JTAG 配置寄存器 3	0x0028	只写
SYSTEM_JTAG_CTRL_4_REG	JTAG 配置寄存器 4	0x002C	只写
SYSTEM_JTAG_CTRL_5_REG	JTAG 配置寄存器 5	0x0030	只写
SYSTEM_JTAG_CTRL_6_REG	JTAG 配置寄存器 6	0x0034	只写
SYSTEM_JTAG_CTRL_7_REG	JTAG 配置寄存器 7	0x0038	只写

名称	描述	地址	访问权限
低功耗管理寄存器			
SYSTEM_RTC_FASTMEM_CONFIG_REG	RTC 快速内存配置寄存器	0x0078	不定
SYSTEM_RTC_FASTMEM_CRC_REG	RTC 快速内存 CRC 校验寄存器	0x007C	只读
外设时钟门控和复位寄存器			
SYSTEM_CPU_PERI_CLK_EN_REG	CPU 外设时钟使能寄存器	0x0010	读写
SYSTEM_CPU_PERI_RST_EN_REG	CPU 外设复位寄存器	0x0014	读写
SYSTEM_PERIP_CLK_EN0_REG	系统外设时钟（硬件加速器）使能寄存器 0	0x0040	读写
SYSTEM_PERIP_CLK_EN1_REG	系统外设时钟（硬件加速器）使能寄存器 1	0x0044	读写
SYSTEM_PERIP_RST_EN0_REG	系统外设（硬件加速器）复位寄存器 0	0x0048	读写
SYSTEM_PERIP_RST_EN1_REG	系统外设（硬件加速器）复位寄存器 1	0x004C	读写
版本寄存器			
SYSTEM_REG_DATE_REG	版本控制寄存器	0x0FFC	读写

6.6 寄存器

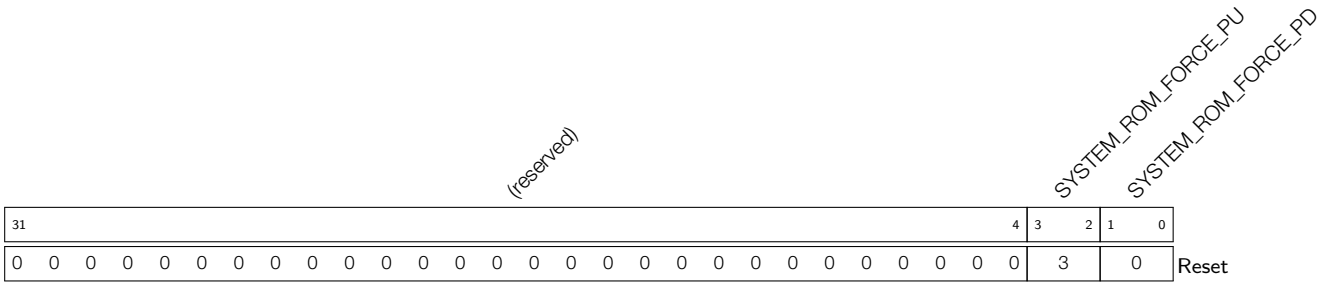
请注意，这里的地址是相对于系统寄存器基地址的地址偏移量（相对地址）。请参阅章节 6.4 获取有关系统寄存器基地址的信息。

Register 6.1: SYSTEM_ROM_CTRL_0_REG (0x0000)



SYSTEM_ROM_FO 强制打开内部 ROM 的时钟门控。详见表 31。（读写）

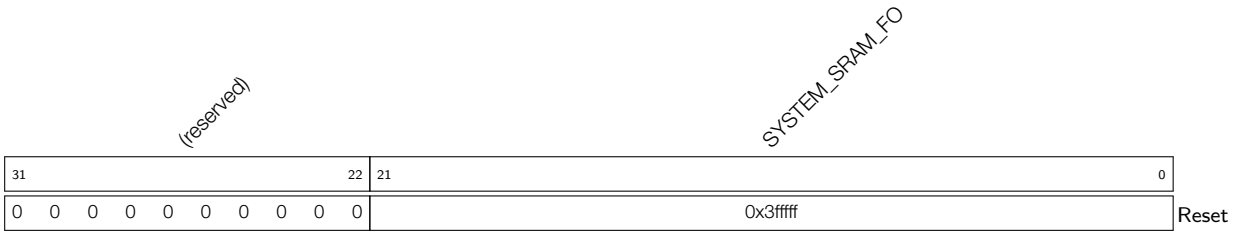
Register 6.2: SYSTEM_ROM_CTRL_1_REG (0x0004)



SYSTEM_ROM_FORCE_PD 控制内部 ROM 的掉电。详见表 31。（读写）

SYSTEM_ROM_FORCE_PU 控制内部 ROM 的上电。详见表 31。（读写）

Register 6.3: SYSTEM_SRAM_CTRL_0_REG (0x0008)

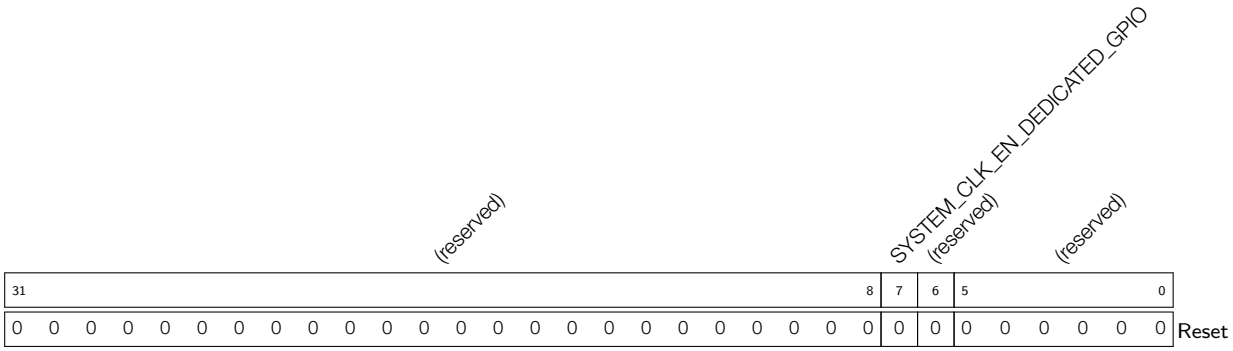


SYSTEM_SRAM_FO 强制打开内部 SRAM 的时钟门控。详见表 32。(读写)

Register 6.4: SYSTEM_SRAM_CTRL_1_REG (0x000C)

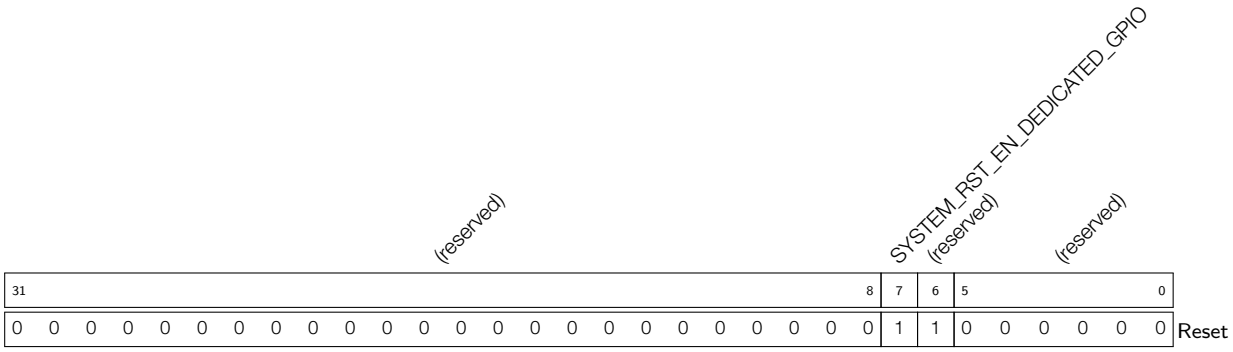
SYSTEM_SRAM_FORCE_PD 控制内部 SRAM 的掉电。详见表 32。(读写)

Register 6.5: SYSTEM_CPU_PERI_CLK_EN_REG (0x0010)



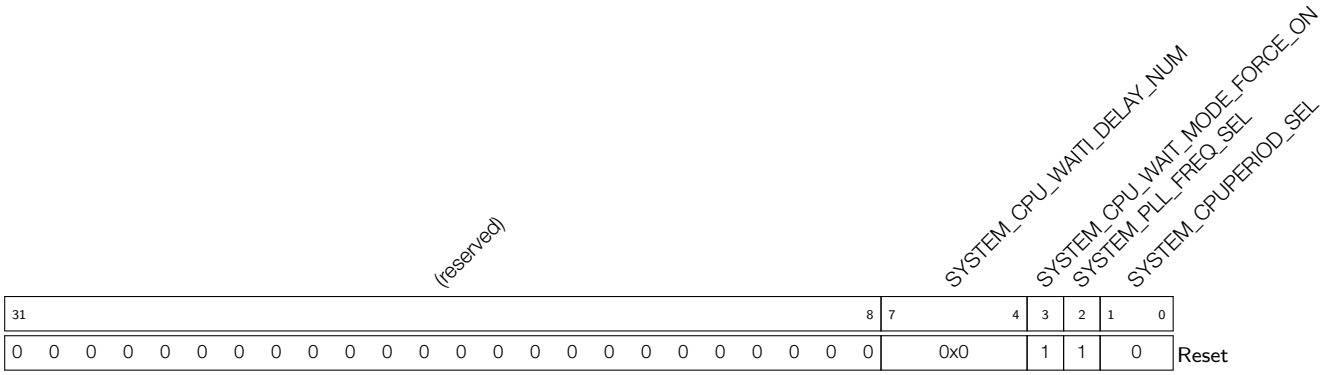
SYSTEM_CLK_EN_DEDICATED_GPIO 置 1 打开 DEDICATED GPIO 时钟。(读写)

Register 6.6: SYSTEM_CPU_PERI_RST_EN_REG (0x0014)



SYSTEM_RST_EN_DEDICATED_GPIO 置 1 复位 DEDICATED GPIO。(R/W)

Register 6.7: SYSTEM_CPU_PER_CONF_REG (0x0018)



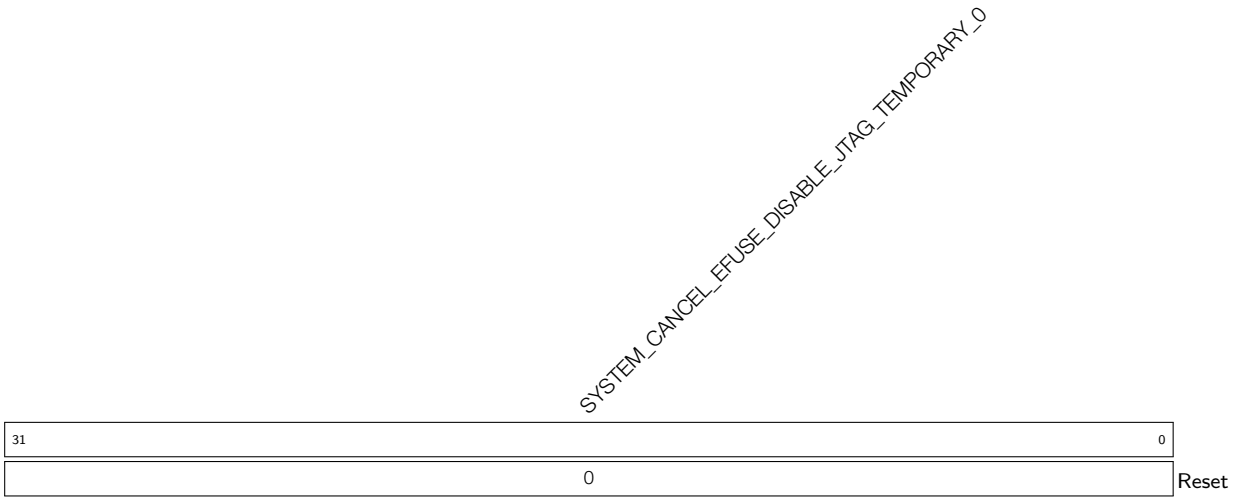
SYSTEM_CPUPERIOD_SEL 选择 CPU 时钟周期或时钟频率。详见章节 2 复位和时钟中表 8。(读写)

SYSTEM_PLL_FREQ_SEL 选择基于 CPU 时钟周期的 PPL 时钟频率。详见章节 2 复位和时钟。(读写)

SYSTEM_CPU_WAIT_MODE_FORCE_ON 置 1 强制打开 CPU 等待模式。在 CPU 等待模式下，CPU 时钟门控一直处于关闭状态，直到中断产生。用户也可通过 WAITI 指令强制打开 CPU 等待模式。(读写)

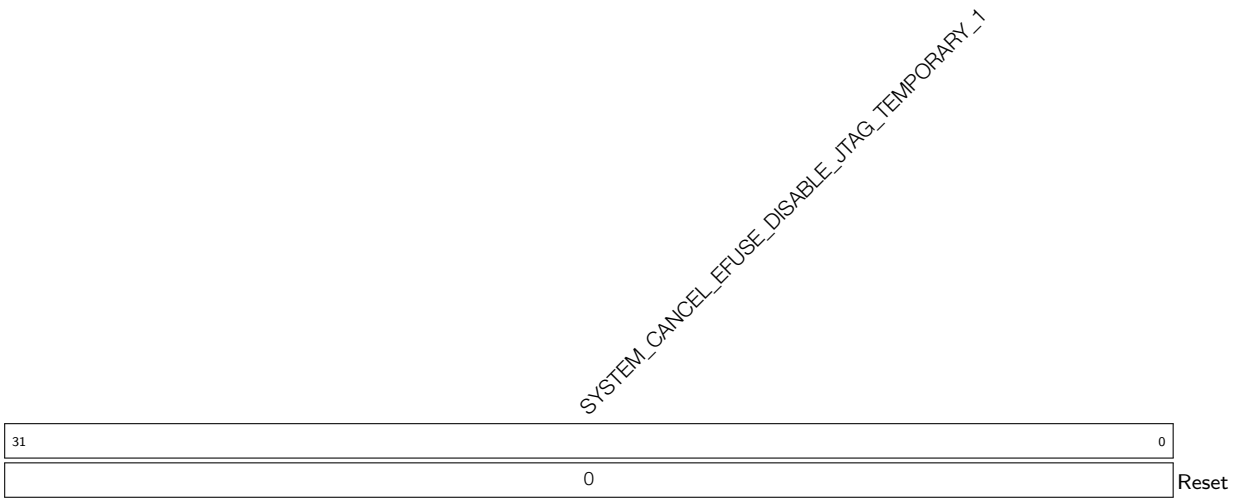
SYSTEM_CPU_WAITI_DELAY_NUM 设置 CPU 在收到 WAITI 指令后进入 CPU 等待模式前的等待周期。(读写)

Register 6.8: SYSTEM_JTAG_CTRL_0_REG (0x001C)



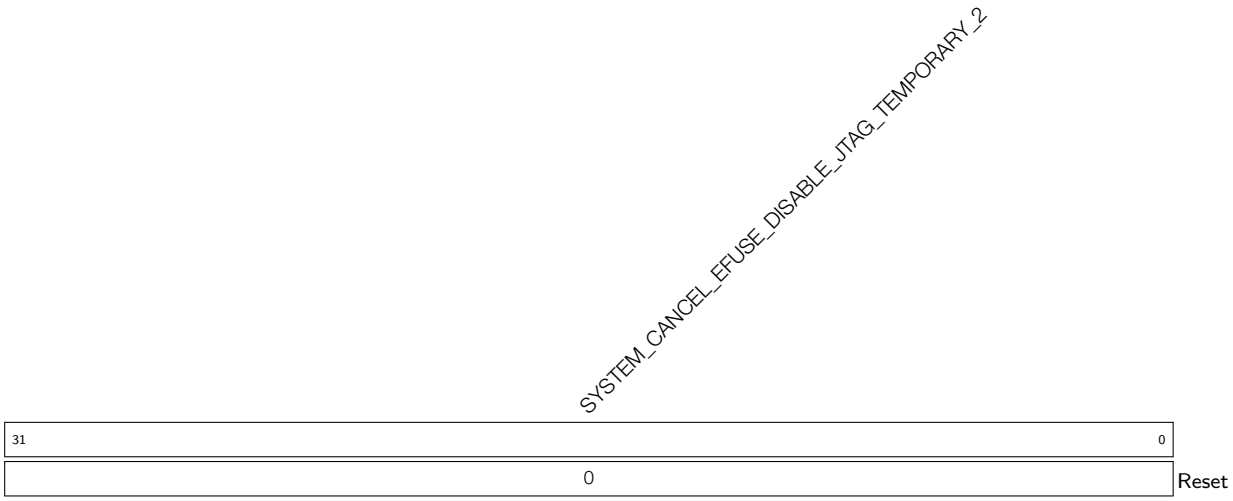
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_0 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 0 到 31 位。（只写）

Register 6.9: SYSTEM_JTAG_CTRL_1_REG (0x0020)



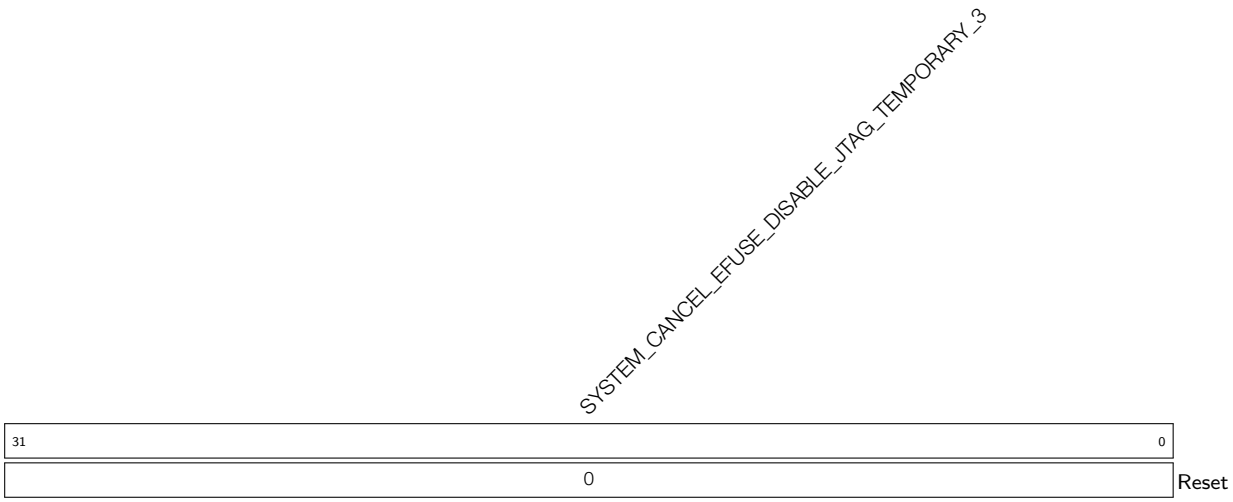
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_1 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 32 到 63 位。（只写）

Register 6.10: SYSTEM_JTAG_CTRL_2_REG (0x0024)



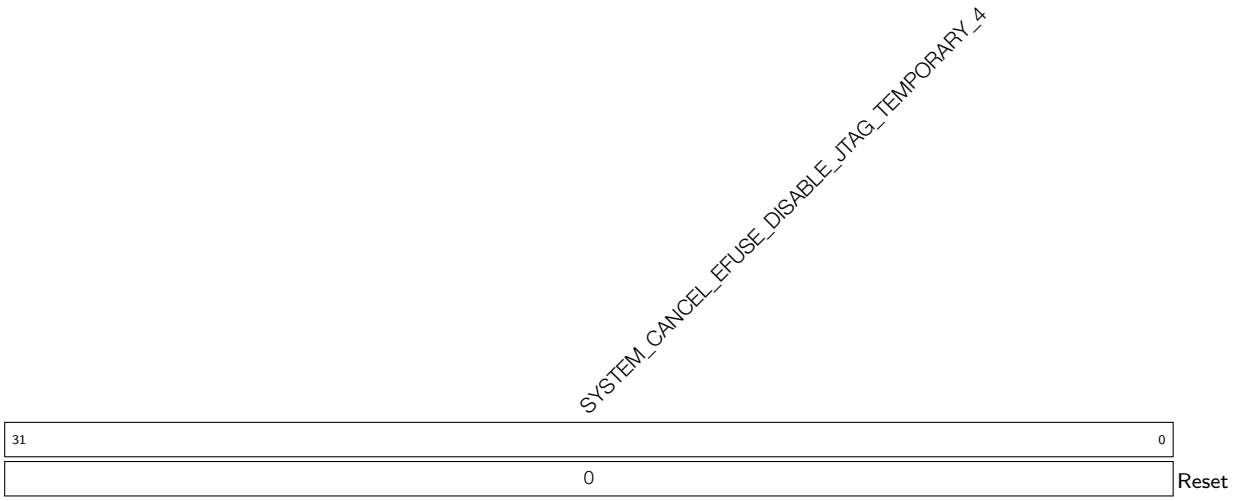
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_2 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 64 到 95 位。（只写）

Register 6.11: SYSTEM_JTAG_CTRL_3_REG (0x0028)



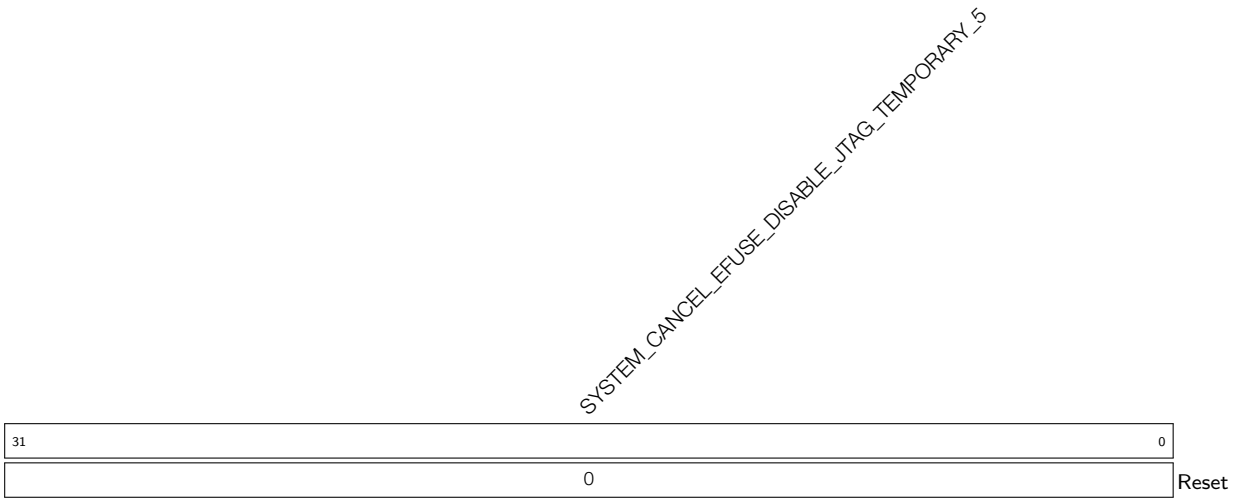
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_3 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 96 到 127 位。（只写）

Register 6.12: SYSTEM_JTAG_CTRL_4_REG (0x002C)



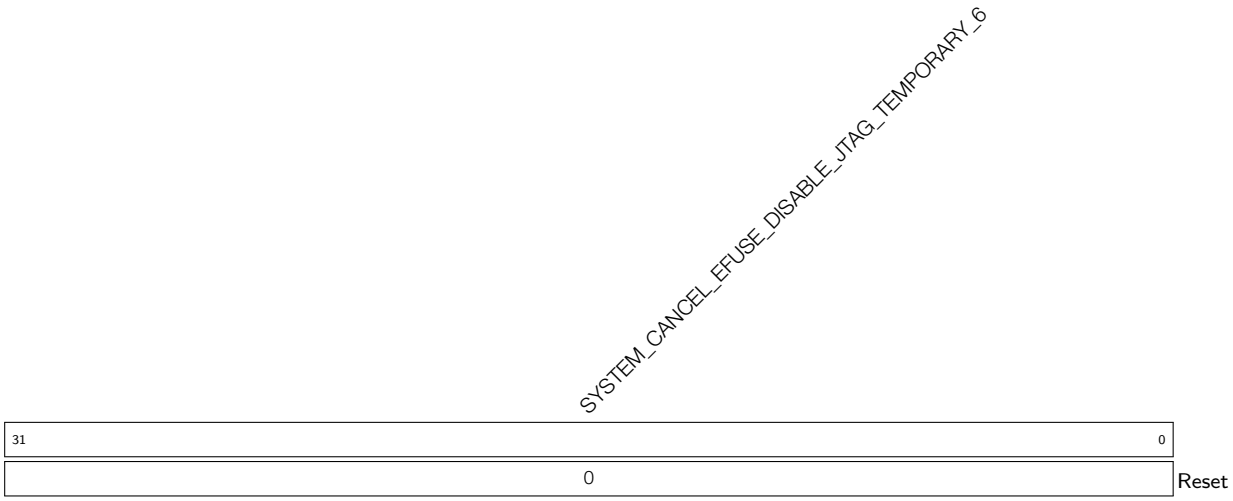
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_4 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 128 到 159 位。（只写）

Register 6.13: SYSTEM_JTAG_CTRL_5_REG (0x0030)



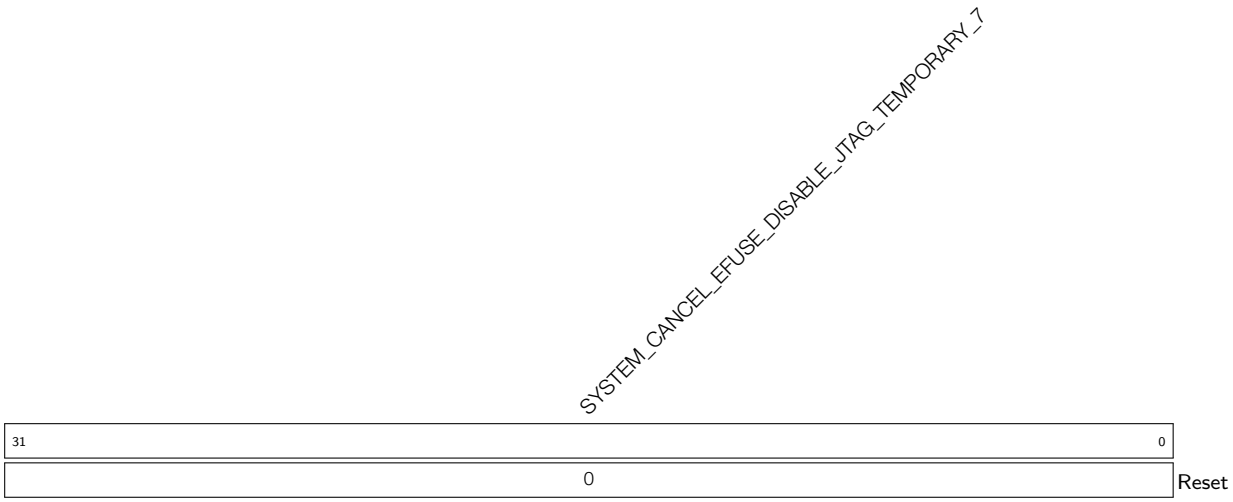
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_5 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 160 到 191 位。（只写）

Register 6.14: SYSTEM_JTAG_CTRL_6_REG (0x0034)



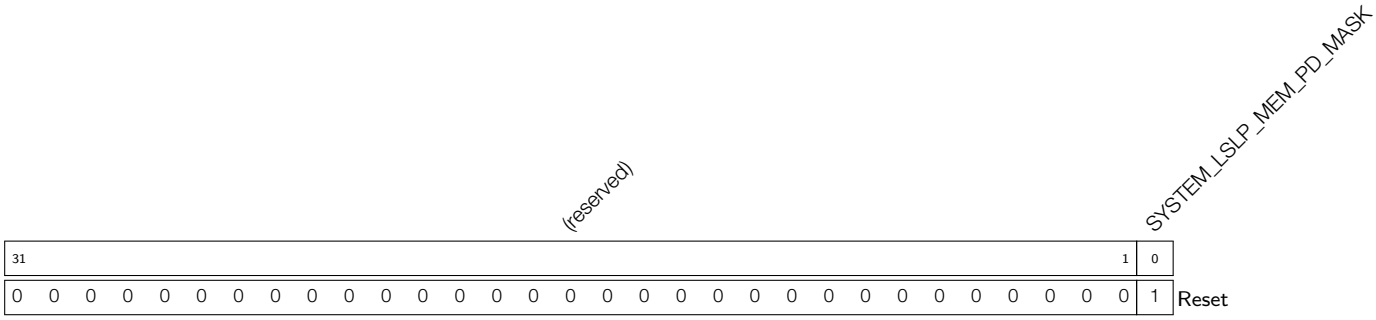
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_6 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 192 到 223 位。（只写）

Register 6.15: SYSTEM_JTAG_CTRL_7_REG (0x0038)



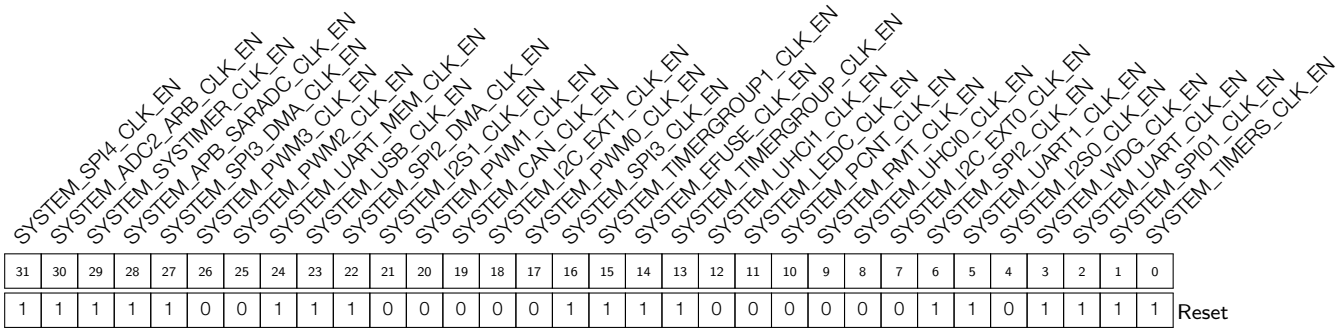
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_7 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 224 到 255 位。（只写）

Register 6.16: SYSTEM_MEM_PD_MASK_REG (0x003C)



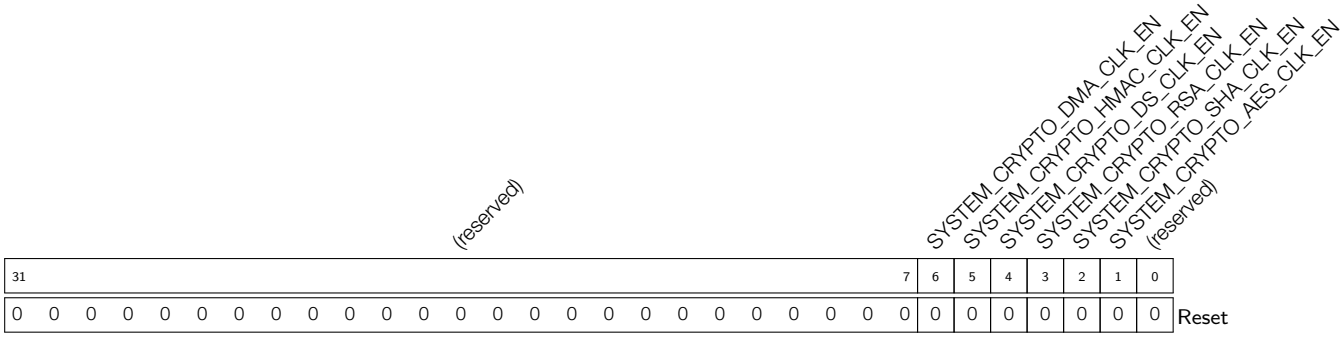
SYSTEM_LSLP_MEM_PD_MASK 置 1 允许存储器在 light-sleep 模式下仍正常工作。(读写)

Register 6.17: SYSTEM_PERIP_CLK_EN0_REG (0x0040)



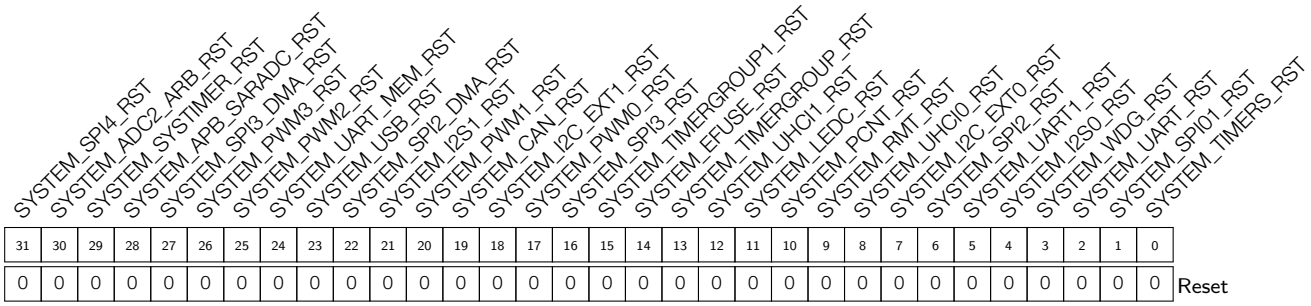
SYSTEM_CPU_PERI_CLK_EN0_REG 使能不同的外设时钟，详见表 33。

Register 6.18: SYSTEM_PERIP_CLK_EN1_REG (0x0044)



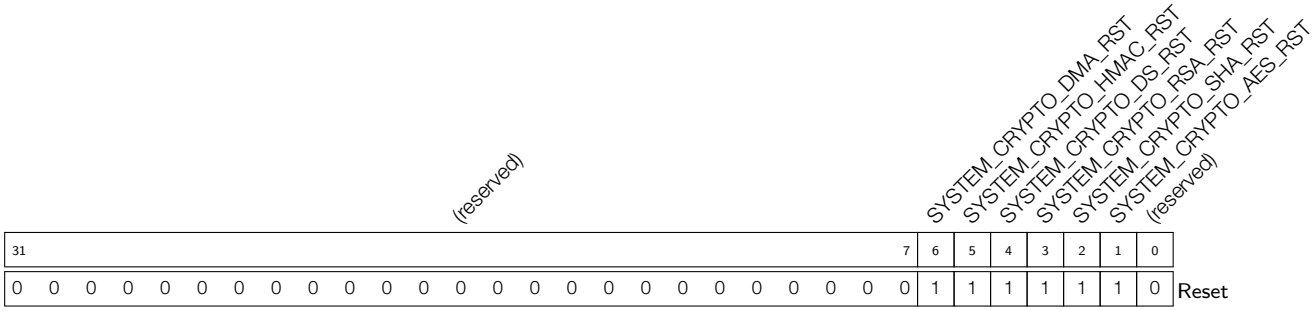
SYSTEM_PERIP_CLK_EN1_REG 使能不同的加速器时钟，详见表 33。

Register 6.19: SYSTEM_PERIP_RST_EN0_REG (0x0048)



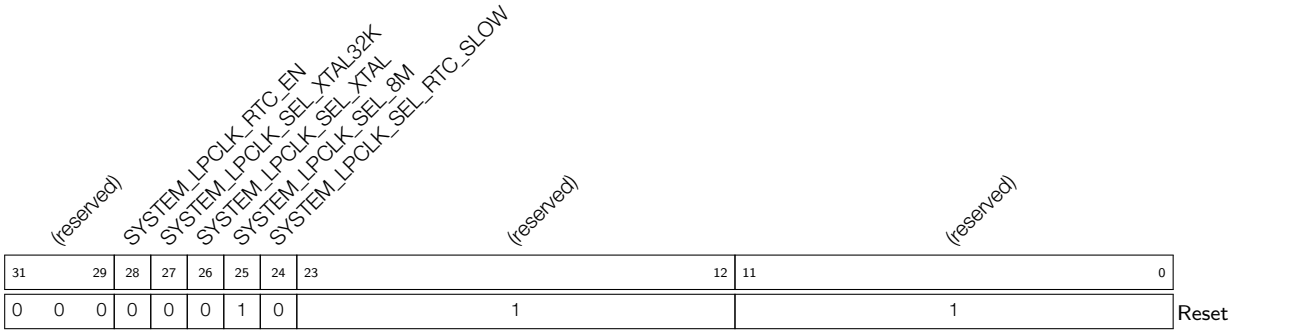
SYSTEM_PERIP_RST_EN0_REG 复位不同外设，详见表 33。

Register 6.20: SYSTEM_PERIP_RST_EN1_REG (0x004C)



SYSTEM_PERIP_RST_EN1_REG 复位不同加速器，详见表 33。

Register 6.21: SYSTEM_BT_LPCK_DIV_FRAC_REG (0x0054)



SYSTEM_LPCLK_SEL_RTC_SLOW 置 1 选择 RTC 慢速时钟为低功耗时钟。（读写）

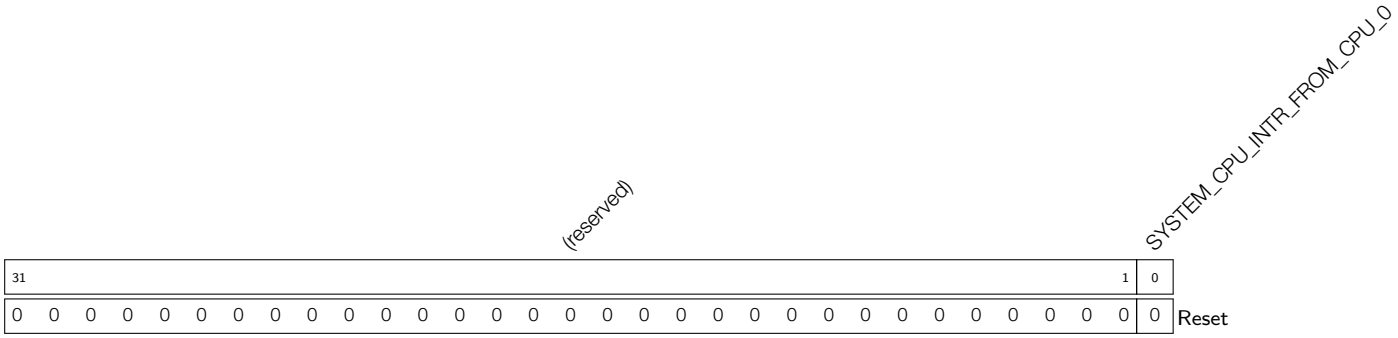
SYSTEM_LPCLK_SEL_8M 置 1 选择 8m 时钟为低功耗时钟。（读写）

SYSTEM_LPCLK_SEL_XTAL 置 1 选择 xtal 时钟为低功耗时钟。（读写）

SYSTEM_LPCLK_SEL_XTAL32K 置 1 选择 xtal32k 时钟为低功耗时钟。（读写）

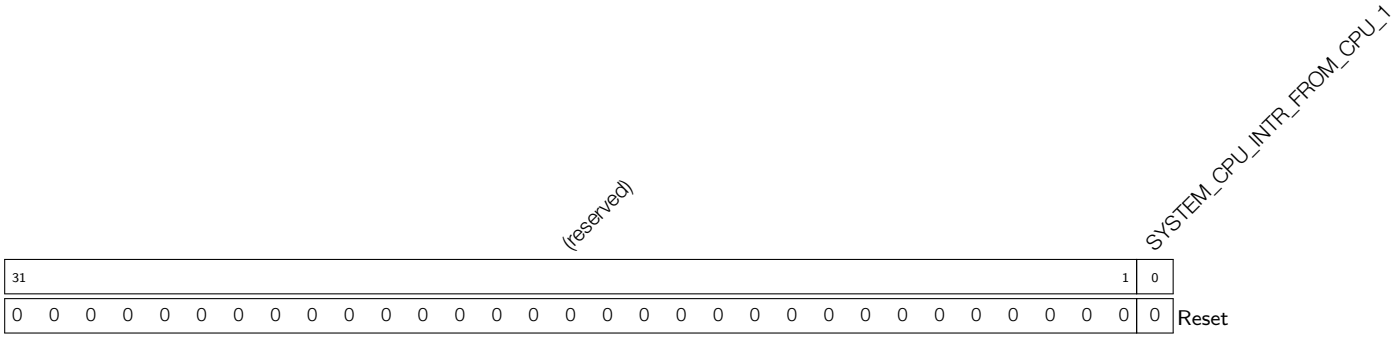
SYSTEM_LPCLK_RTC_EN 置 1 使能 RTC 低功耗时钟。（读写）

Register 6.22: SYSTEM_CPU_INTR_FROM_CPU_0_REG (0x0058)



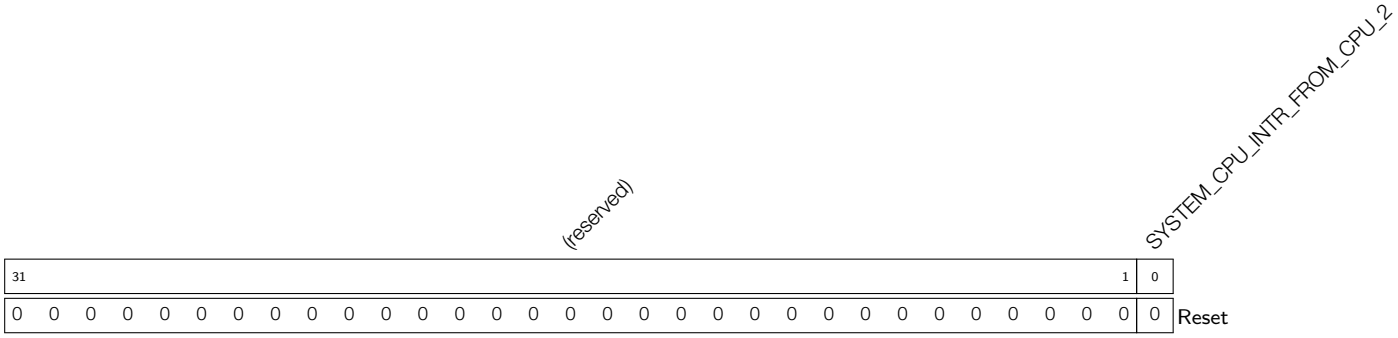
SYSTEM_CPU_INTR_FROM_CPU_0 置 1 生成 CPU 中断 0。该位需在 ISR 过程中由软件清 0。(读写)

Register 6.23: SYSTEM_CPU_INTR_FROM_CPU_1_REG (0x005C)



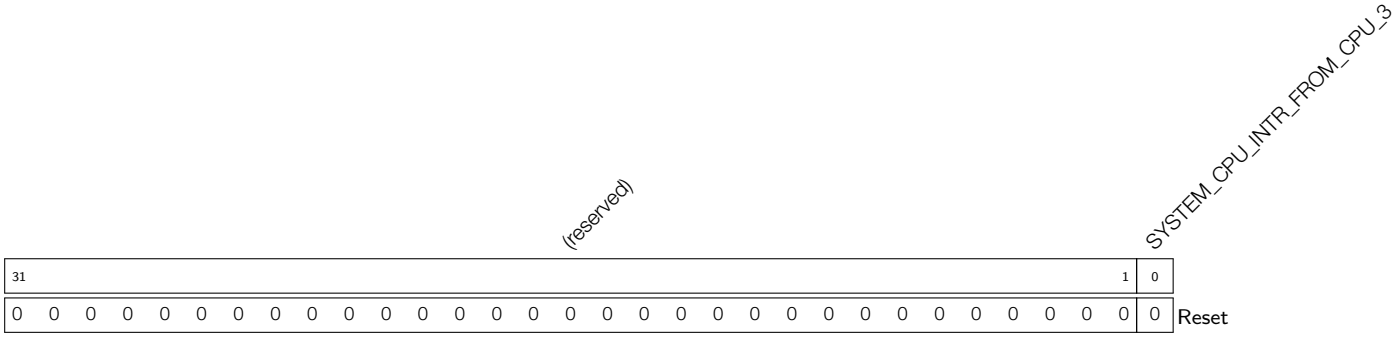
SYSTEM_CPU_INTR_FROM_CPU_1 置 1 生成 CPU 中断 1。该位需在 ISR 过程中由软件清 0。(读写)

Register 6.24: SYSTEM_CPU_INTR_FROM_CPU_2_REG (0x0060)



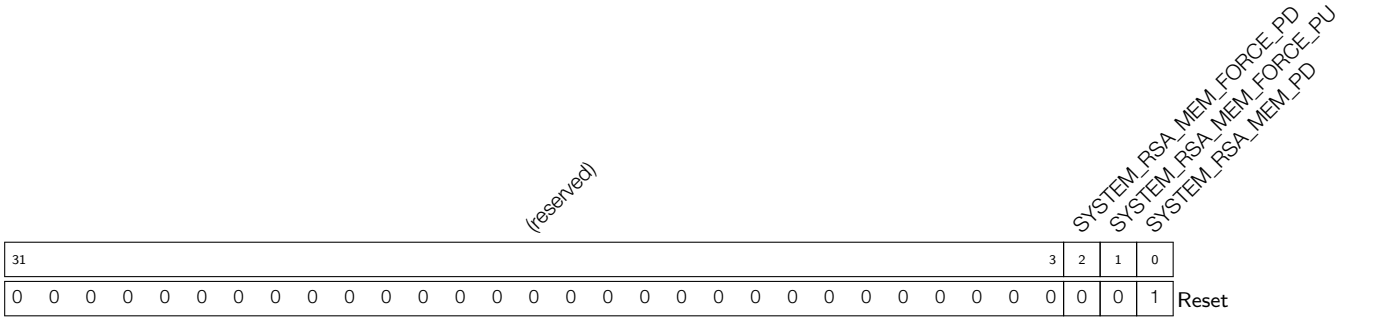
SYSTEM_CPU_INTR_FROM_CPU_2 置 1 生成 CPU 中断 2。该位需在 ISR 过程中由软件清 0。(读写)

Register 6.25: SYSTEM_CPU_INTR_FROM_CPU_3_REG (0x0064)



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。(读写)

Register 6.26: SYSTEM_RSA_PD_CTRL_REG (0x0068)

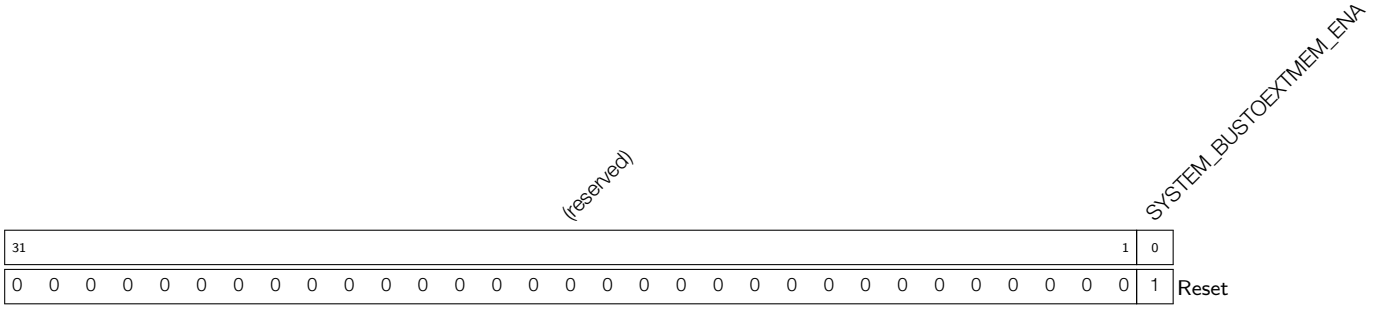


SYSTEM_RSA_MEM_PD 置 1 关闭 RSA 内存。该位优先级最低。当数字签名占用 RSA 加速器时，该位无效。(读写)

SYSTEM_RSA_MEM_FORCE_PU 置 1 关闭 RSA 内存。该位优先级第二高。(读写)

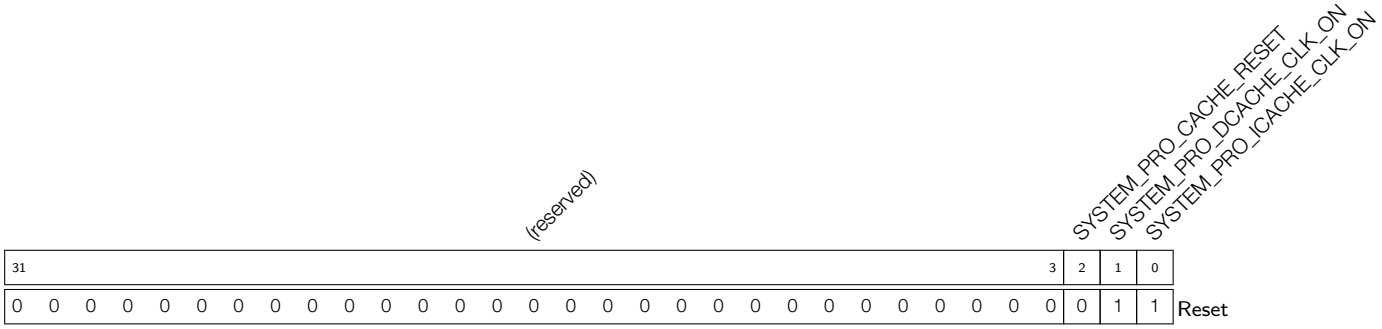
SYSTEM_RSA_MEM_FORCE_PD 置 1 关闭 RSA 内存。该位优先级最高。(读写)

Register 6.27: SYSTEM_BUSTOEXTMEM_ENA_REG (0x006C)



SYSTEM_BUSTOEXTMEM_ENA 置 1 使能 EDMA 总线。(读写)

Register 6.28: SYSTEM_CACHE_CONTROL_REG (0x0070)

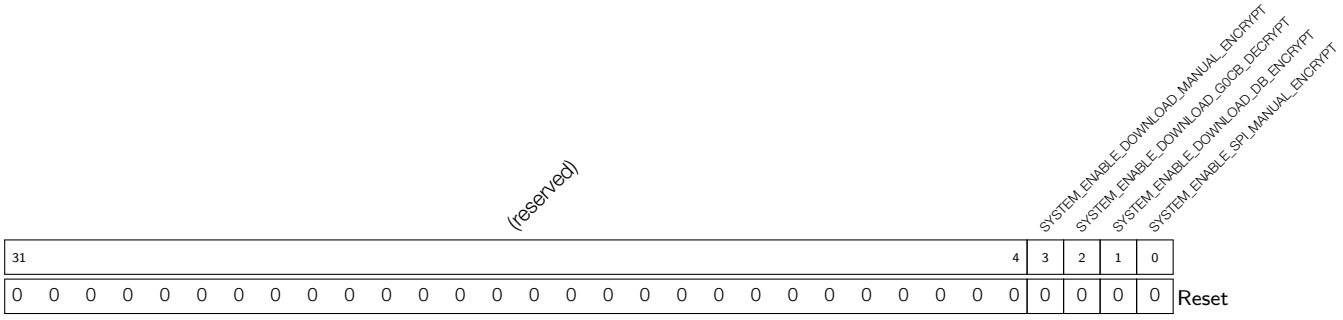


SYSTEM_PRO_ICACHE_CLK_ON 置 1 使能 i-cache 时钟。（读写）

SYSTEM_PRO_DCACHE_CLK_ON 置 1 使能 d-cache 时钟。（读写）

SYSTEM_PRO_CACHE_RESET 置 1 复位 cache。（读写）

Register 6.29: SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG (0x0074)



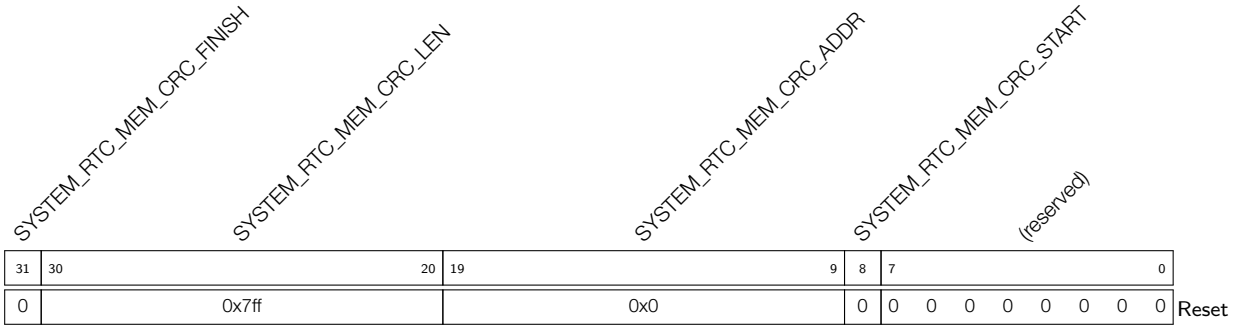
SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT 置 1 在 SPI Boot 模式下使能手动加密 (Manual Encryption)。（读写）

SYSTEM_ENABLE_DOWNLOAD_DB_ENCRYPT 置 1 在 Download Boot 模式下使能自动加密 (Auto Encryption)。（读写）

SYSTEM_ENABLE_DOWNLOAD_G0CB_DECRYPT 置 1 在 Download Boot 模式下使能自动解密 (Auto Decryption)。（读写）

SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT 置 1 在 Download Boot 模式下使能手动加密 (Manual Encryption)。（读写）

Register 6.30: SYSTEM_RTC_FASTMEM_CONFIG_REG (0x0078)



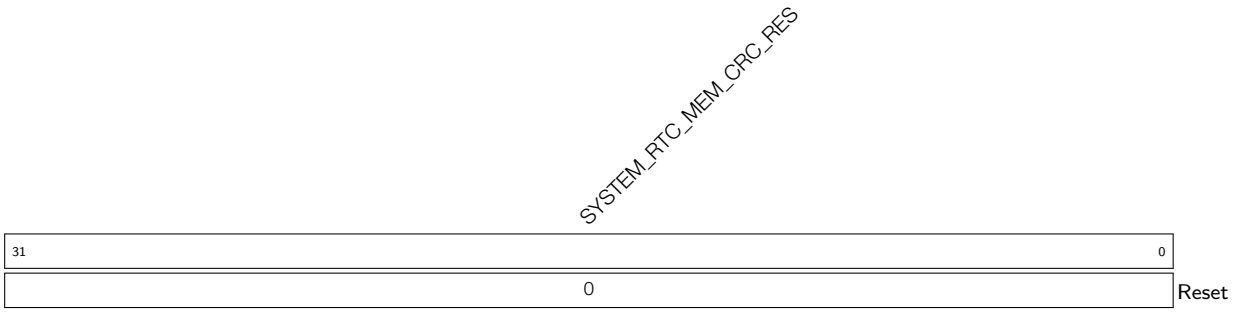
SYSTEM_RTC_MEM_CRC_START 置 1 启动 RTC 存储的 CRC 校验。(读写)

SYSTEM_RTC_MEM_CRC_ADDR 设置 CRC 校验的 RTC 存储地址。(读写)

SYSTEM_RTC_MEM_CRC_LEN 设置用于 CRC 校验的 RTC 存储长度(基于起始地址)。(读写)

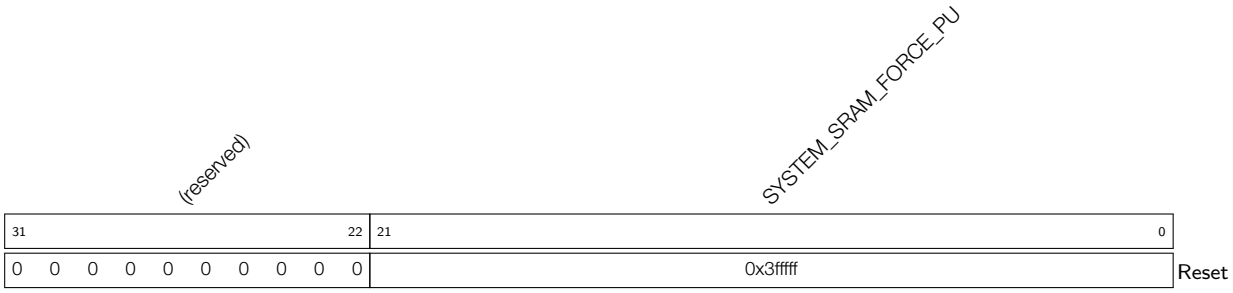
SYSTEM_RTC_MEM_CRC_FINISH 储存 RTC 存储 CRC 校验状态。高电平表示校验完成，低电平表示校验未完成。(只读)

Register 6.31: SYSTEM_RTC_FASTMEM_CRC_REG (0x007C)



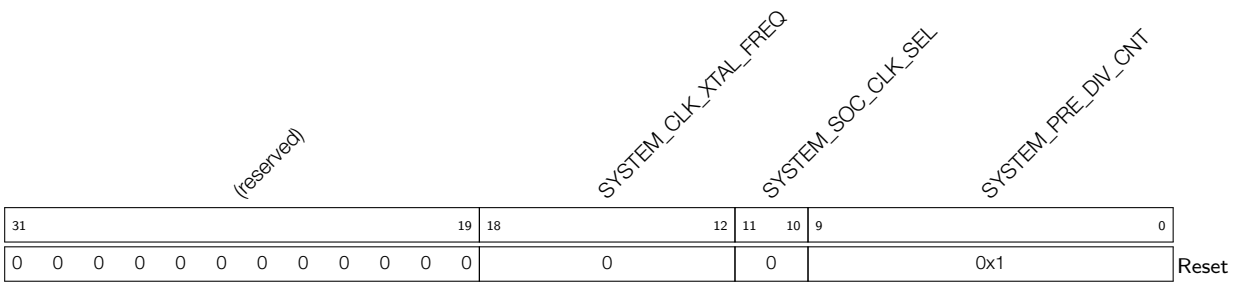
SYSTEM_RTC_MEM_CRC_RES 储存 RTC 存储的 CRC 校验结果。(只读)

Register 6.32: SYSTEM_SRAM_CTRL_2_REG (0x0088)



SYSTEM_SRAM_FORCE_PU 控制内部 SRAM 的上电。详见表 32。(读写)

Register 6.33: SYSTEM_SYSCLK_CONF_REG (0x008C)

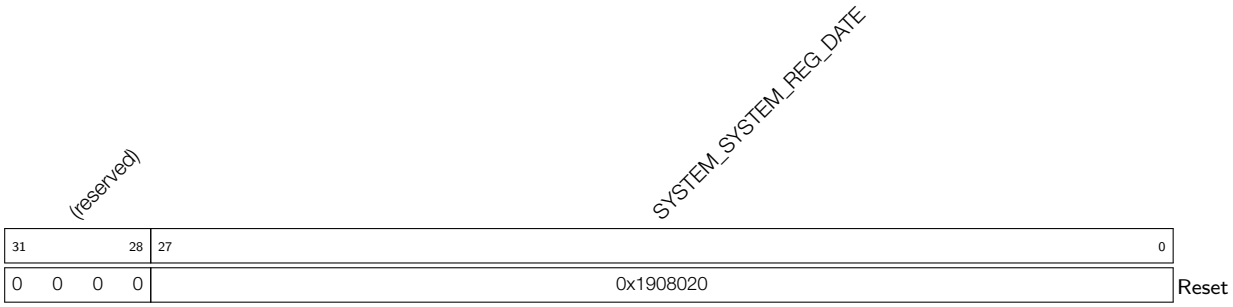


SYSTEM_PRE_DIV_CNT 设置预分频器计数器。具体配置，请见章节 2 复位和时钟中表 10。（读写）

SYSTEM_SOC_CLK_SEL 选择 SoC 时钟。具体配置，请见章节 2 复位和时钟中表 8。（读写）

SYSTEM_CLK_XTAL_FREQ 读取晶振频率（单位：MHz）。（只读）

Register 6.34: SYSTEM_REG_DATE_REG (0x0FFC)



SYSTEM_DATE 版本控制寄存器。（读写）

7. DMA 控制器

7.1 概述

直接存储访问 (Direct Memory Access, DMA) 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。可以在无需任何 CPU 操作的情况下通过 DMA 快速移动数据，从而提高了 CPU 的效率。

ESP32-S2 包括三种 DMA：Internal DMA，EDMA 和 Copy DMA。Internal DMA 只能访问片内 RAM，用于内存与外设之间的数据传输；EDMA 既能访问片内 RAM 也能访问片外 RAM，用于存储器与外设之间的数据传输；Copy DMA 只能访问片内 RAM，用于内存与内存之间的高速数据传输。

ESP32-S2 中有 8 个外设具有 DMA 功能。如表7-1所示，UART0 与 UART1 共用一个 Internal DMA；SPI3 与 ADC Controller 共用一个 Internal DMA；AES 与 SHA 共用一个 EDMA；SPI2 与 I2S0 各有一个单独的 EDMA。除此之外，CPU Peripheral 模块下还包括一个 Copy DMA。

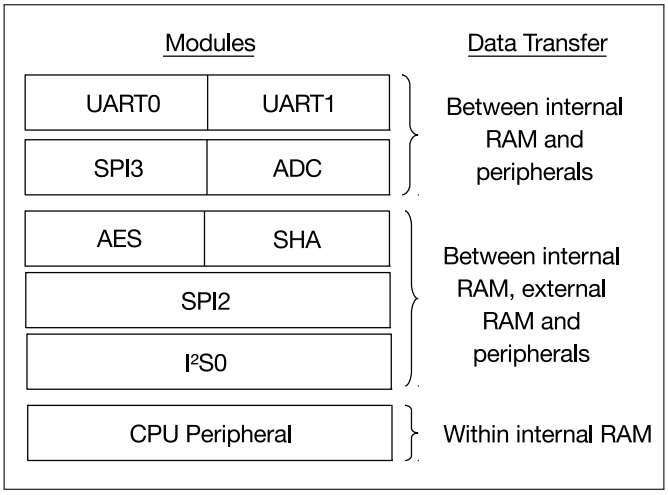


图 7-1. 具有 DMA 功能的模块和支持的数据传输类型

7.2 特性

DMA 控制器具有以下几个特点：

- AHB 总线架构
- 支持半双工和全双工收发数据
- 数据传输以字节为单位，传输数据量可软件编程
- 访问内部 RAM 时，支持 INCR burst 传输
- DMA 能够访问的内部 RAM 最大地址空间为 320 KB
- DMA 能够访问的最大外部地址空间为 10.5 MB
- 通过 DMA 实现高速数据传输

7.3 功能描述

ESP32-S2 中所有需要进行高速数据传输的模块都具有 DMA 功能。DMA 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。根据各自模块的需求，各个模块的 DMA 控制器功能有所差别，但是 DMA 引擎 (DMA_ENGINE) 的结构相同。

7.3.1 DMA 引擎的架构

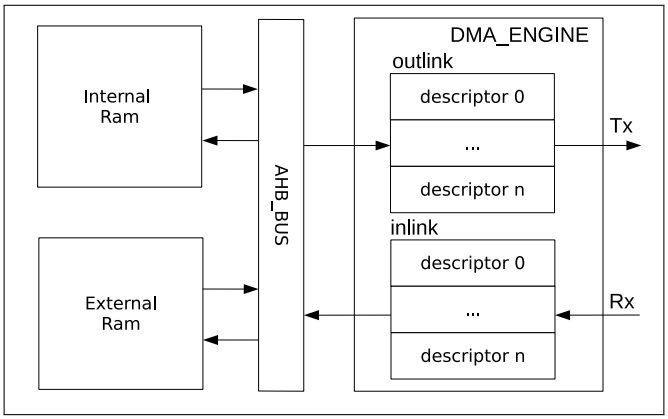


图 7-2. DMA 引擎的架构

DMA 引擎通过 AHB_BUS 将数据存入外部或内部 RAM 或者将数据从外部或内部 RAM 取出。图 7-2 为 DMA 引擎基本架构图。RAM 的具体使用范围详见章节 1 系统和存储器。软件可以通过挂载链表的方式来使用 DMA 引擎。DMA_ENGINE 根据 outlink 中的内容将相应 RAM 中的数据发送出去，也可根据 inlink 中的内容将接收的数据存入指定 RAM 地址空间。

7.3.2 链表

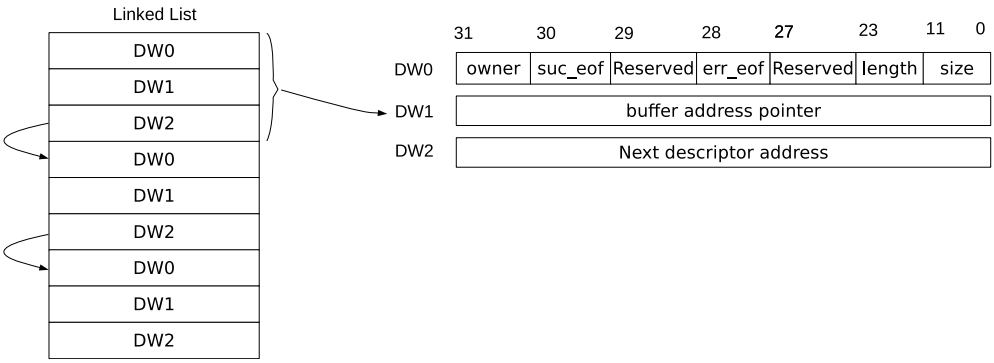


图 7-3. 链表结构图

图 7-3 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 3 个字组成。链表应存放在内部 RAM 中，供 DMA 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 DMA 控制器。
  在 DMA 使用完该描述符对应的 buffer 后，硬件会将该 bit 清零。也可通过置位 *PERI*_IN_LOOP_TEST 位关闭硬件自动清零的功能。软件在挂载链表时需要将该 bit 置 1。

注意: *PERI*指包含 DMA 的外设模块名，如 I²S，SPI，UHCI 等。

- suc_eof (DW0) [30]：表示结束标志。
  1'b0：当前描述符不是链表中的最后一个描述符；
  1'b1：当前描述符为数据包的最后一个描述符。
  对于接收描述符，硬件会在收完 1 个包后将该 bit 置 1。对于发送描述符，需要软件在包的最后一个描述符中的该 bit 置 1。
- Reserved (DW0) [29]：保留。
- err_eof (DW0) [28]：表示接收结束错误标志。
  该 bit 只用于 UART DMA 接收数据。对于接收描述符，硬件在收完 1 个包并检测到接收数据错误会将该 bit 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 中读取数据时能够读取的字节数；对于接收描述符该段由硬件使用完该 buffer 后自动填写，向 buffer 中存储数据时表示已存数据的字节数。
  当 DMA 访问片外 RAM 时，该大小必须是 16/32/64 bytes 的整数倍，详情见章节 7.3.6 *访问片外 RAM*。
- size (DW0) [11:0]：表示当前描述符对应的 buffer 的大小。
  当 DMA 访问片外 RAM 时，该大小必须是 16/32/64 bytes 的整数倍，详情见章节 7.3.6 *访问片外 RAM*。
- buffer address pointer (DW1)：buffer 地址指针。
  当 DMA 访问片外 RAM 时，写地址必须根据 *PERI*_EXT_MEM_BK_SIZE 大小对齐，详情见章节 7.3.6 *访问片外 RAM*。
- next descriptor address (DW2)：下一个描述符的地址指针。当前描述符为链表中最后一个描述符时 (suc_eof = 1)，该值可以为 0。并且描述符的地址指针只能在片内 RAM。

用 DMA 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。

7.3.3 启动 DMA

软件通过挂载链表的方式来使用 DMA。对于接收数据，软件挂载好接收链表，配置 *PERI*_INLINK_ADDR 字段指向第一个接收链表描述符。置位 *PERI*_INLINK_START 位启动 DMA。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 *PERI*_OUTLINK_ADDR 字段指向第一个发送链表描述符。置位 *PERI*_OUTLINK_START 位启动 DMA。*PERI*_INLINK_START 与 *PERI*_OUTLINK_START 位由硬件自动清零。

DMA 支持 Restart 功能。如果不确定已挂载链表的使用状态（使用完或正在使用中），又希望挂载新的链表，利用 Restart 功能，可以完成挂载新的链表而又不影响已挂载链表的正常使用。软件使用该功能时，需要重写已挂载链表的最后一个描述符，使其第三个 word 中的内容指向新链表的首地址；然后挂载新的链表，如图 7-4 所示；并置位 *PERI*_INLINK_RESTART 或者 *PERI*_OUTLINK_RESTART（这两个位由硬件自动清零）。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，从而继续读取新挂载的链表。

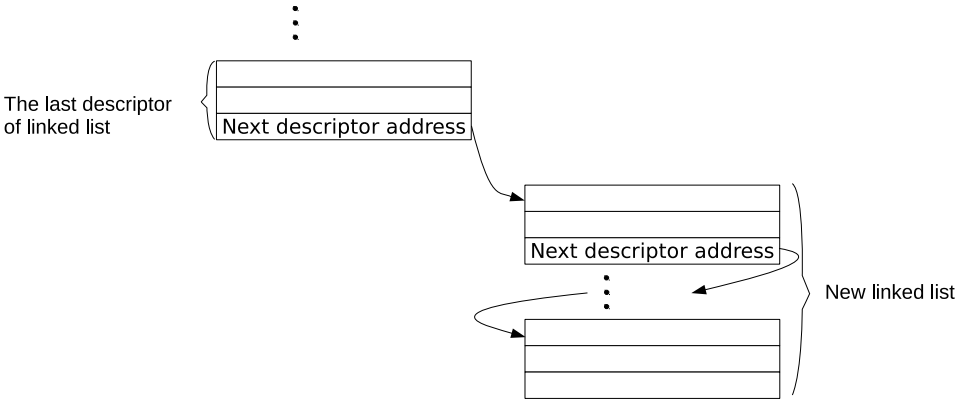


图 7-4. 链表关系图

7.3.4 读链表

软件在配置并启动 DMA 后，DMA 会从内部 RAM 读取链表。置位 *PERI*_IN_DSCR_ERR_INT_ENA 或者 *PERI*_OUT_DSCR_ERR_INT_ENA 位使能描述符错误中断。访问片内 RAM 时，当描述符中第二个 word 指示的地址不在 0x3FFB0000 ~ 0x3FFFFFFF；访问片外 RAM 时，当描述符中第二个 word 指示的地址不在 0x3F500000 ~ 0x3FF7FFFF 时，产生描述符错误中断。

注意: 描述符的第三个 word 指示的地址只能在片内，指向下一个可用描述符；所有描述符都需存在内存中。

7.3.5 数据传输结束标志

DMA 通过 EOF 来指示数据传输结束。发送数据时，置位 *PERI*_OUT_TOTAL_EOF_INT_ENA 位使能 *PERI*_OUT_TOTAL_EOF_INT 中断，当带有 EOF 标志的描述符对应 buffer 的数据传输完成后，DMA 会产生该中断。接收数据时，置位 *PERI*_IN_SUC_EOF_INT_ENA 位使能 *PERI*_IN_SUC_EOF_INT 中断，表示数据接收完成。对于 UART DMA 还支持中断 UHCI_IN_ERR_EOF_INT，置位 UHCI_IN_ERR_EOF_INT_ENA 使能该中断中断，表示接收完成但接收数据有错误。

软件在检测到 *PERI*_OUT_TOTAL_EOF_INT 或 *PERI*_IN_SUC_EOF_INT 中断时，可以记录 *PERI*_OUTLINK_DSCR_ADDR 或 *PERI*_INLINK_DSCR_ADDR 字段的值，即最后一个描述符的地址。需要注意，这两个地址为真实描述符地址右移两 bit 的结果。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。

7.3.6 访问片外 RAM

并不是所有外设的 DMA 都支持访问片外 RAM，ESP32-S2 只有 I²S0, SPI2, AES 和 SHA 的 DMA 支持访问片外 RAM。DMA 可访问的片外 RAM 地址空间为：0x3F500000 ~ 0x3FF7FFFF。软件配置地址时需要注意写地址（向片外 RAM 写入数据时的地址）必须按 16/32/64 字节大小对齐，表 36显示了写地址分别为 16 位、32 位、64 位对齐时 *PERI*_EXT_MEM_BK_SIZE 位的值。

注意: 读地址（从片外 RAM 读出数据时的地址）不需要进行地址对齐。

表 36: 配置寄存器与写地址的关系表

*PERI*_EXT_MEM_BK_SIZE	写地址对齐方式
0	16 位对齐
1	32 位对齐
2	64 位对齐

7.4 Copy DMA 控制器

Copy DMA 用于芯片内存与内存之间数据的传输，并且只能访问片内 RAM。图7-5所示为 Copy DMA 引擎的架构，与 Internal DMA 与 EDMA 不同的是，Copy DMA 通过发送链表将片内 RAM 的数据取出存入 DMA 内部 FIFO，并通过接收链表将内部 FIFO 中的数据写入目标片内 RAM。

Copy DMA 的软件配置流程如下：

1. 将 CP_DMA_IN_RST、CP_DMA_OUT_RST、CP_DMA_FIFO_RST 和 CP_DMA_CMDFIFO_RST 位先置 1 后置 0，复位 Copy DMA 状态机和 FIFO 指针；
2. 将 CP_DMA_FIFO_RST 位先置 1 然后置 0，复位 FIFO 指针；
3. 挂载好发送链表，配置 CP_DMA_OUTLINK_ADDR 指向第一个发送链表描述符；
4. 挂载好接收链表，配置 CP_DMA_INLINK_ADDR 指向第一个接收链表描述符；
5. 置位 CP_DMA_OUTLINK_START 启动 DMA 发送；
6. 置位 CP_DMA_INLINK_START 启动 DMA 接收。

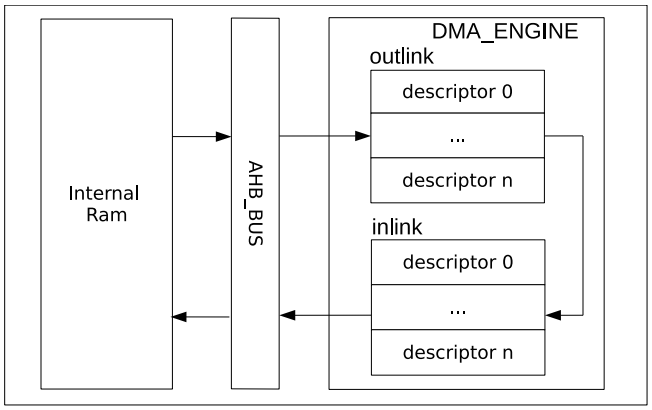


图 7-5. Copy DMA 引擎架构

7.5 UART DMA (UDMA) 控制器

ESP32-S2 中有 2 个 UART 接口，它们共用 1 个 UDMA 控制器。UHCI_UART_CE 寄存器用于选择哪个串口占用 UDMA。

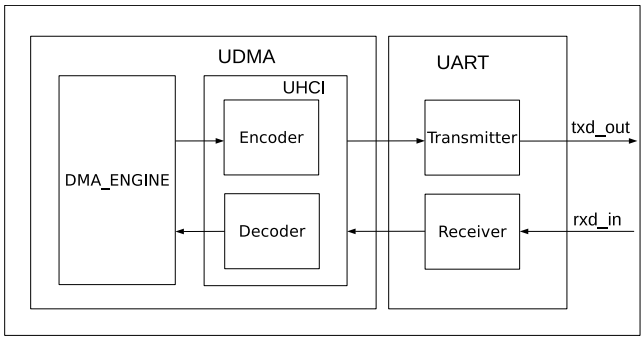


图 7-6. UDMA 模式数据传输

图 7-6 为 UDMA 方式数据传输图。在 UDMA 接收数据前，软件将接收链表准备好。UHCI_INLINK_ADDR 用于指向第一个接收链表描述符。置位 UHCI_INLINK_START 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 UDMA 的控制下存入接收链表指定的 RAM 空间。

在 UDMA 发送数据前，软件需要将发送链表和发送数据准备好，UHCI_OUTLINK_ADDR 用于指向第一个发送链表描述符。置位 UHCI_OUTLINK_START 之后，DMA 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。

UDMA 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替换。Decoder 用于去除数据包前后分隔符，并将数据中的特殊字符进行替换为分隔符。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 UHCI_OUT_TOTAL_EOF_INT 中断。当数据接收完成后，会产生 UHCI_IN_SUC_EOF_INT 中断。

7.6 SPI DMA 控制器

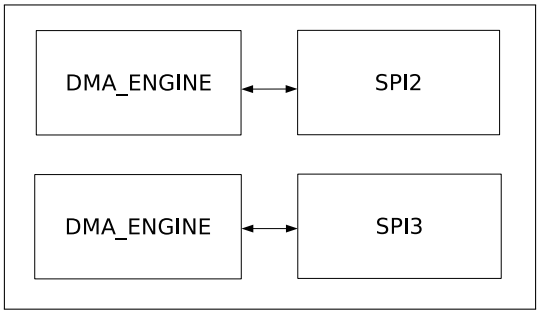


图 7-7. SPI DMA

ESP32-S2 SPI 除了使用 CPU 实现与外设的数据交换外，还可以使用 DMA 。如图 7-7 所示，SPI2 和 SPI3 分别单独使用一个 DMA。

ESP32-S2 SPI DMA 使用链表接收/发送数据，发送数据支持 burst 操作，一次接收/发送的数据长度至少为 1 个字节。

寄存器 SPI_DMA_OUT_LINK_REG 的 SPI_OUTLINK_START 位和寄存器 SPI_DMA_IN_LINK_REG 的 SPI_INLINK_START 位用于使能 DMA 引擎，这两个位由硬件清零。当 SPI_OUTLINK_START 位被置为 1 时，DMA 引擎开始处理发送链表，并准备发送数据；当 SPI_INLINK_START 位被置为 1 时，DMA 引擎开始处理接收链表，并准备接收数据。

SPI DMA 接收数据时软件配置流程如下：

1. 将 SPI_DMA_IN_RST、SPI_AHBM_FIFO_RST 和 SPI_AHBM_RST 位先置 1 后置 0，复位 DMA 状态机和 FIFO 指针；
2. 挂载好接收链表，配置 SPI_DMA_INLINK_ADDR 指向第一个接收链表描述符；
3. 置位 SPI_DMA_INLINK_START 启动 DMA 接收。

SPI DMA 发送数据时的软件配置流程如下：

1. 将 SPI_DMA_OUT_RST、SPI_AHBM_FIFO_RST 和 SPI_AHBM_RST 位先置 1 后置 0，复位 DMA 状态机和 FIFO 指针；

2. 挂载好发送链表，配置寄存器 SPI_DMA_OUTLINK_ADDR 指向第一个发送链表描述符；
3. 置位 SPI_DMA_OUTLINK_START 启动 DMA 发送。

注意: 在使用 SPI2 与 SPI3 DMA 实现片内 RAM 与片外 RAM 之间数据传输时，需要置位 SPI_MEM_TRANS_EN。

SPI2 与 SPI3 DMA 还支持分段传输模式。

7.7 I²S DMA 控制器

ESP32-S2 I²S 单独使用一个 DMA。寄存器 I2S_FIFO_CONF_REG 的 I2S_DSCR_EN 位用于使能 I²S 的 DMA 操作。ESP32-S2 I²S DMA 使用链表接收/发送数据，发送数据支持 burst 操作。寄存器 I2S_RXEOF_NUM_REG 的 I2S_RX_EOF_NUM[31:0] 位用于配置 DMA 一次接收数据个数，单位为字。

寄存器 I2S_OUT_LINK_REG 的 I2S_OUTLINK_START 位和寄存器 I2S_IN_LINK_REG 的 I2S_INLINK_START 位用于使能 DMA 引擎，这两个位由硬件清零。当 I2S_OUTLINK_START 位被置为 1 时，DMA 引擎开始处理发送链表，并准备发送数据，当 I2S_INLINK_START 位被置为 1 时，DMA 引擎开始处理接收链表，并准备接收数据。

I²S DMA 接收数据时的软件配置流程如下：

1. 将 I2S_IN_RST、I2S_AHBM_FIFO_RST 和 I2S_AHBM_RST 位先置 1 后置 0，复位 DMA 状态机和 FIFO 指针；
2. 挂载好接收链表，配置 I2S_INLINK_ADDR 指向第一个接收链表描述符；
3. 置位 I2S_INLINK_START 启动 DMA 接收。

I²S DMA 发送数据时的软件配置流程如下：

1. 对 I2S_OUT_RST、I2S_AHBM_FIFO_RST 和 I2S_AHBM_RST 位置 1 然后置 0，复位 DMA 状态机和 FIFO 指针；
2. 挂载好发送链表，配置 I2S_OUTLINK_ADDR 指向第一个发送链表描述符；
3. 置位 I2S_OUTLINK_START 启动 DMA 发送。

注意: 在使用 I²S DMA 实现片内 RAM 与片外 RAM 之间数据传输时，需要置位 I2S_MEM_TRANS_EN。

8. 通用异步收发传输器 (UART)

8.1 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。芯片中有 2 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。

2 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 2 个 UART 控制器，*n* 为 0、1。

8.2 主要特性

- 可编程收发波特率
- 2 个 UART 的发送 FIFO 以及接收 FIFO 共享 512 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 1/1.5/2/3 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 DMA 高速数据通信
- 支持 UART 唤醒模式
- 支持软件流控和硬件流控

8.3 功能描述

8.3.1 UART 简介

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。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 DMA，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。

8.3.2 UART 架构

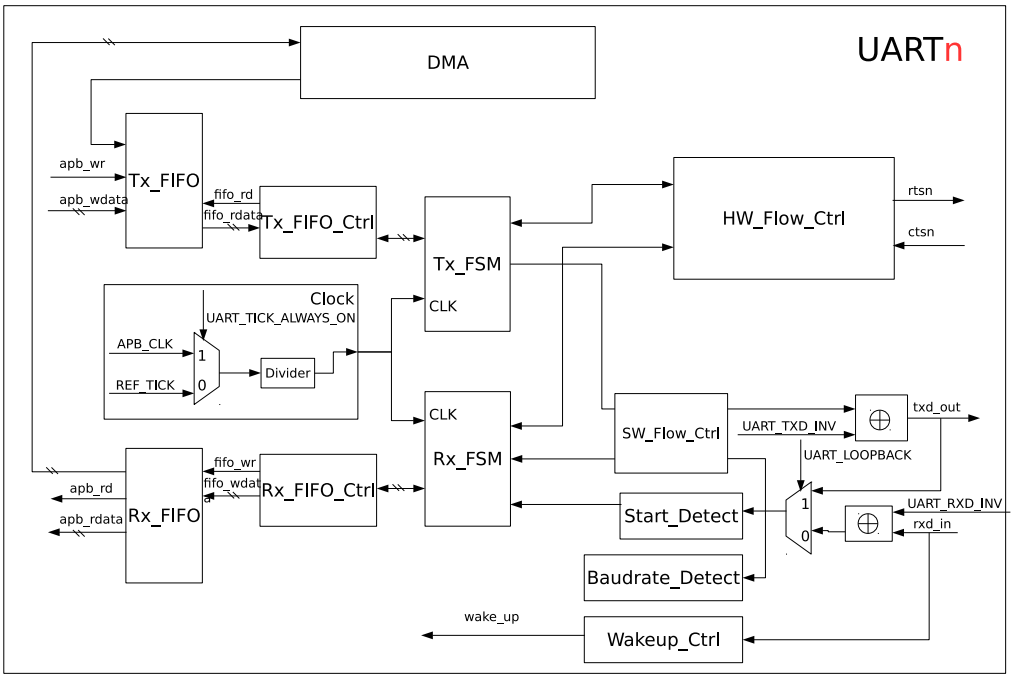


图 8-1. UART 基本架构图

图 8-1 为 UART 基本架构图。UART 有两个时钟源：80-MHz APB_CLK 以及参考时钟 REF_TICK （详情请参考章节 2 复位和时钟）。可以通过配置 UART_TICK_REF_ALWAYS_ON 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 模块。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。

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 DMA 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 DMA 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 状态时，Wakeup_Ctrl 开始计算 rxd_in 的上升沿个数，当上升沿个数大于 (UART_ACTIVE_THRESHOLD + 2）时产生 wake_up 信号给 RTC 模块，由 RTC 来唤醒芯片。

8.3.3 UART RAM

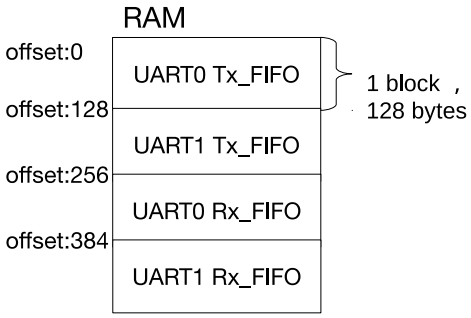


图 8-2. UART 共享 RAM 图

芯片中 2 个 UART 控制器共用 512x8-bit RAM 空间。如图 8-2 所示，RAM 以 block 为单位进行分配，1 block 为 128x8 bits。图 8-2 所示为默认情况下 2 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 进行扩展，通过配置 UART_RX_SIZE 可以对 UART*n* 的 Rx_FIFO 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，UART0 Rx_FIFO 可以从地址 256 扩展到 RAM 的尾地址，UART1 Rx_FIFO 则不支持地址空间扩展，以 1 block 为单位进行扩展。需要注意的是当扩展某一个 UART 的 FIFO 空间时可能会占用其他 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 2 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。

UART0 和 UART1 的 Tx_FIFO 可以通过置位 UART_TXFIFO_RST 来复位, UART0 和 UART1 的 Rx_FIFO 可以通过置位 UART_RXFIFO_RST 来复位。

对于 Tx FIFO，可以通过 APB 总线或 DMA 向其写入数据，硬件 Tx_FSM 自动从其中读取数据，数据将按照配置的帧格式转换成比特流；对于 Rx FIFO，可以通过 APB 总线或 DMA 读取其中的数据，并存储到内存，硬件 Rx_FSM 将接收到的比特流转换成字节并写入 Rx FIFO。两个 UART 共享同一个 DMA。

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。

8.3.4 波特率产生与检测

8.3.4.1 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。

波特率分频器系数由 UART_CLKDIV + (UART_CLKDIV_FRAG/16) 构成。也就是说，最终波特率为 INPUT_FREQ/(UART_CLKDIV + (UART_CLKDIV_FRAG/16))。例如，若 UART_CLKDIV = 694，UART_CLKDIV_FRAG = 7，则分频系数为 (694 + 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 8-3 所示，每 16 个输出脉冲，波特率发生器分频 (UART_CLKDIV + 1) 个输入脉冲或 UART_CLKDIV 个输入脉冲。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 8-3 所示，输出脉冲相互交错，使得输出时序更加统一。

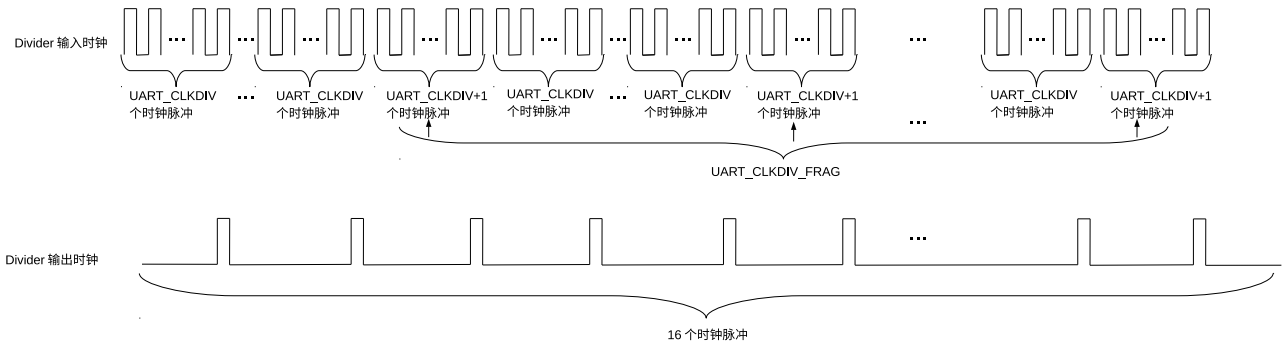


图 8-3. UART 控制器分频

为了支持 IrDA （详情见本节 8.3.7 *IrDA*），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。

8.3.4.2 波特率检测

置位 UART_AUTOBAUD_EN 可以开启 UART 波特率自检测功能。图 8-1 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。

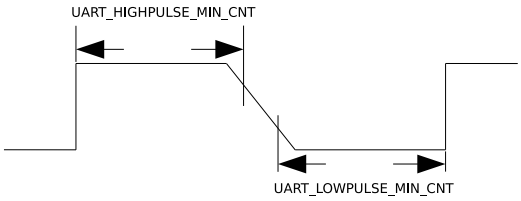


图 8-4. UART 信号下降沿较差时序图

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, 对于 UART 信号的下降沿信号比较差的情况，如图8-4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：

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

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：

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

8.3.5 UART 数据帧

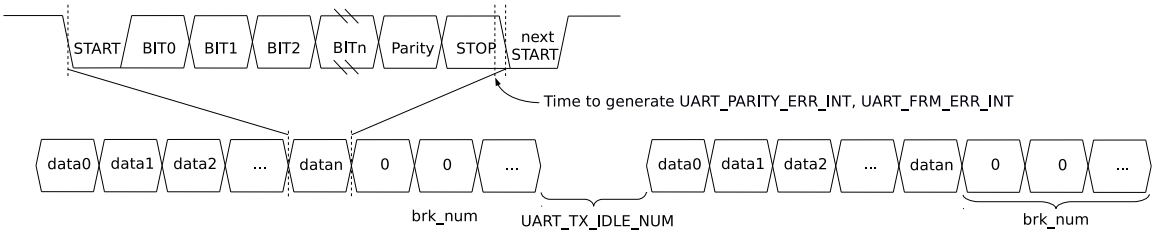


图 8-5. UART 数据帧结构

图 8-5 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、UART_DL1_EN 和 UART_DL0_EN 实现 1/1.5/2/3 位宽。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，输入数据仍会存入 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，输入数据会被存入 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 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 寄存器的配置值时则产生 UART_TX_BRK_IDLE_DONE_INT 中断。

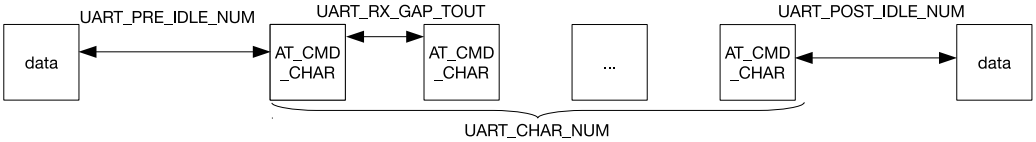


图 8-6. AT_CMD 字符格式

图 8-6 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。

- 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。

8.3.6 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。

8.3.6.1 驱动控制

如图 8-7 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 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。本设计 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 8-7 所示，DE 与 UART 的 dtrn_out 相连（详见本节 8.3.9.1 硬件流控）。

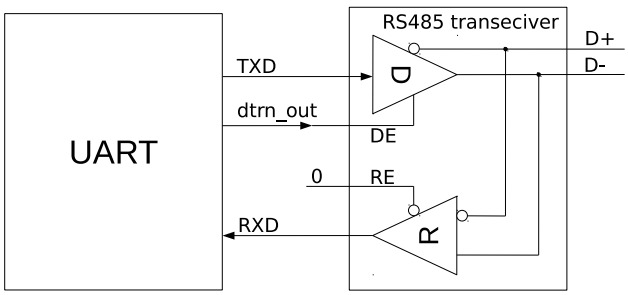


图 8-7. RS485 模式驱动控制结构图

8.3.6.2 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 turn around dealy。UART 发送模块支持在停止位之后增加两个波特率的延时。置位 UART_DL1_EN，增加一个波特率周期延时；置位 DL0_EN，增加另一个波特率周期延时。

8.3.6.3 总线侦听

RS485 两线 multidrop 系统中，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，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。

8.3.7 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 8-8 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。

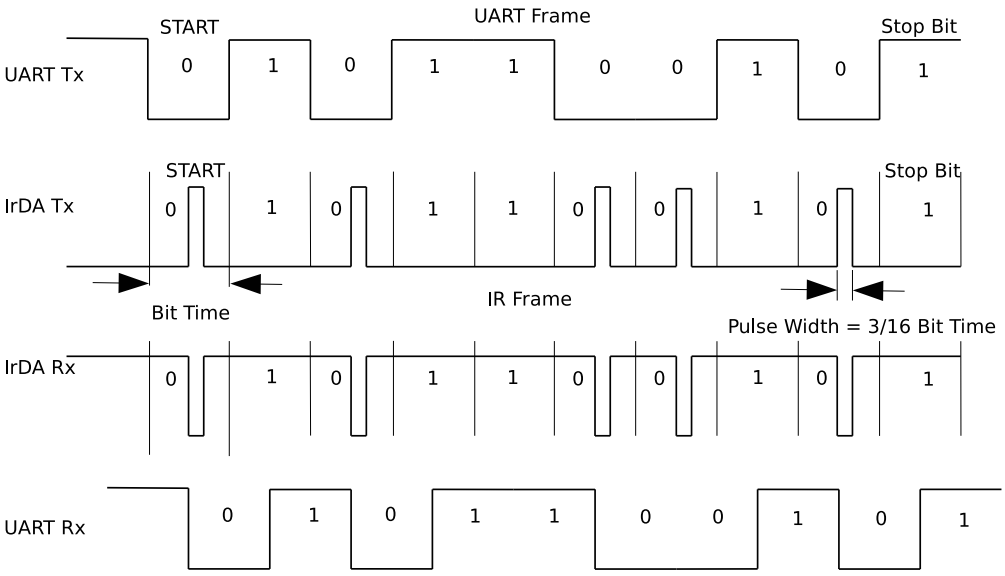


图 8-8. SIR 模式编解码时序图

IrDA 为半双工传输协议，不允许同时进行收发。如图8-9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。

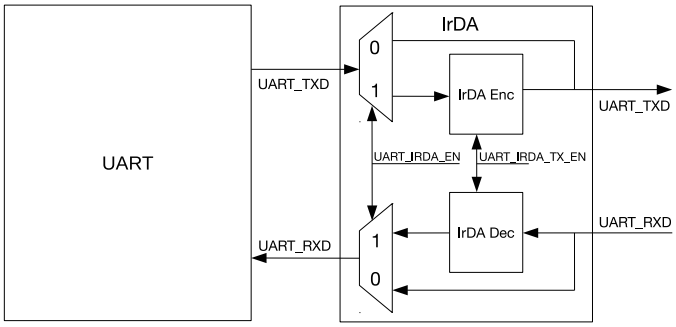


图 8-9. IrDA 编解码结构图

8.3.8 唤醒

UART0 和 UART1 支持唤醒功能。当 UART 处于 Light-sleep 状态时，Wakeup_Ctrl 开始计算 rxd_in 的上升沿个数，当上升沿个数大于（UART_ACTIVE_THRESHOLD + 2）时产生 wake_up 信号给 RTC 模块，由 RTC 来唤醒芯片。

8.3.9 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特殊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。

8.3.9.1 硬件流控

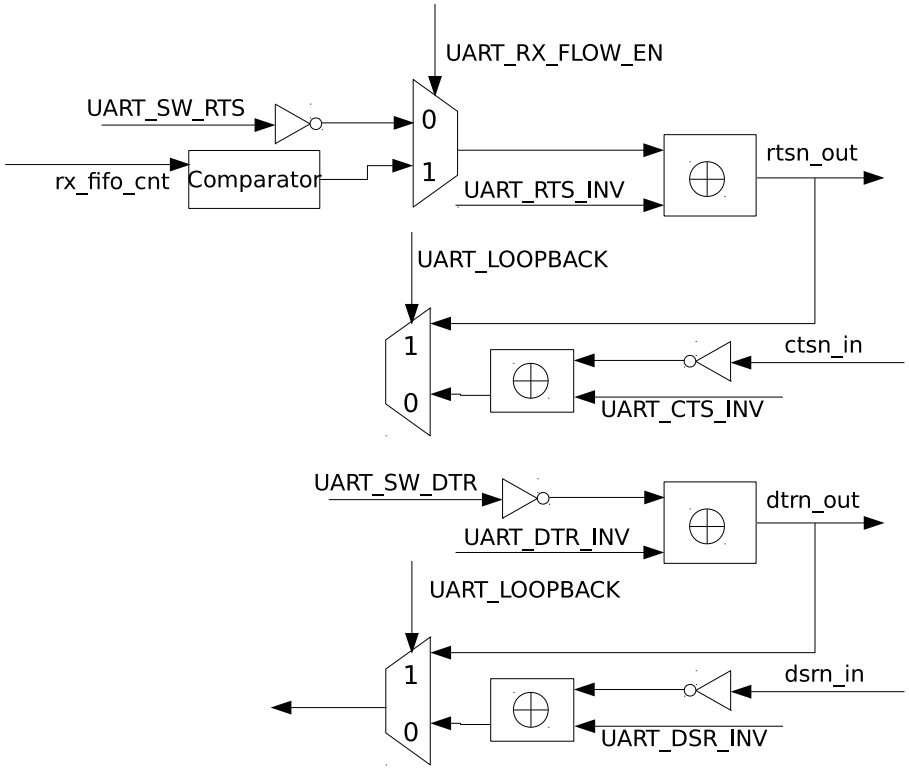


图 8-10. 硬件流控图

图 8-10 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 8-11 为两个 UART 之间硬件流控信号连接图。记 ESP32-S2 UART 为 IU0，External UART 为 EU0，下文将使用这两个标记来区分两个 UART。输出信号 rtsn_out （IU0）为低电平表示允许对方（EU0）发送数据，rtsn_out（IU0）为高电平表示通知对方（EU0）中止数据发送直到 rtsn_out （IU0）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。

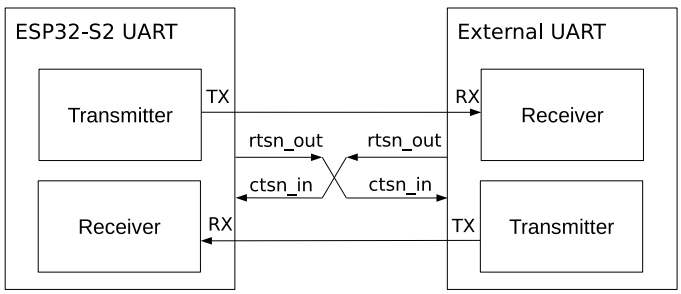


图 8-11. 硬件流控信号连接图

输入信号 ctsn_in（IU0）为低电平表示允许发送端（IU0）发送数据；ctsn_in（IU0）为高电平表示禁止发送端（IU0）发送数据。当 UART 检测到输入信号 ctsn_in （IU0）的沿变化时会产生 UART_CTS_CHG_INT 中

断。

UART 发送设备（IU0）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（IU0）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（EU0）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。

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。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。

8.3.9.2 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。

8.3.10 UDMA

UART 共用一个 UDMA (UART DMA)，UDMA 支持对 HCI 协议数据包的解析（decoder）及数据包封装（encoder）。更多信息请见章节 7 *DMA 控制器*。

8.3.11 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: 当发送器在最后一个数据发送后保持了最短的间隔时间时触发此中断。
- 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 glitch 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 NULL 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。

8.3.12 UCHI 中断

- UHCI_DMA_INFIFO_FULL_WM_INT：当 DMA 接收 FIFO 计数器的值大于 UHCI_DMA_INFIFO_FULL_THRS 时触发此中断。
- UHCI_SEND_A_REG_Q_INT: 当使用 always_send 发送一串短包，DMA 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，DMA 发送了短包后触发此中断。
- UHCI_OUT_TOTAL_EOF_INT: 当所有数据都已发送时触发此中断。
- 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_IN_DSCR_EMPTY_INT: 当 DMA 没有足够的接收链表描述符时触发此中断。
- UHCI_OUT_DSCR_ERR_INT: 当发送链表描述符里有错误时触发此中断。
- UHCI_IN_DSCR_ERR_INT: 当接收链表描述符里有错误时触发此中断。
- UHCI_OUT_EOF_INT: 当前描述符的 EOF 位为 1 时触发此中断。
- UHCI_OUT_DONE_INT: 当发送链表描述符完成时触发此中断。
- UHCI_IN_ERR_EOF_INT: 当接收链表描述符中的 EOF 有错误时触发此中断。
- UHCI_IN_SUC_EOF_INT: 当接收一个数据包时触发此中断。
- UHCI_IN_DONE_INT: 当一个接收链表描述符完成时触发此中断。

- UHCI_TX_HUNG_INT: 当 DMA 从 RAM 中读取数据的时间过长时触发此中断。
- UHCI_RX_HUNG_INT: 当 DMA 接收数据的时间过长时触发此中断。
- UHCI_TX_START_INT: 当 DMA 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。

8.4 基地址

用户可以通过两个不同的寄存器基地址访问 UART，如表 37 所示。更多信息，请访问 1 系统和存储器章节。

表 37: UART0、UART1 与 UHCI 基地址

模块名	访问总线	基地址
UART0	PeriBUS1	0x3F400000
	PeriBUS2	0x60000000
UART1	PeriBUS1	0x3F410000
	PeriBUS2	0x60010000
UHCI	PeriBUS1	0x3F414000
	PeriBUS2	0x60014000

8.5 寄存器列表

请注意，下表中的地址都是相对于 UART0、UART1 和 UHCI 基地址的地址偏移量（相对地址）。更多有关 UART0、UART1 和 UHCI 基地址的信息，请前往 8.4 基地址 章节。

名称	描述	地址	访问
FIFO 配置			
UART_FIFO_REG	FIFO 数据寄存器	0x0000	只读
UART_MEM_CONF_REG	UART 阈值和分配配置	0x005C	读/写
中断寄存器			
UART_INT_RAW_REG	原始中断状态	0x0004	只读
UART_INT_ST_REG	屏蔽中断状态	0x0008	只读
UART_INT_ENA_REG	中断使能位	0x000C	读/写
UART_INT_CLR_REG	中断清除位	0x0010	只写
配置寄存器			
UART_CLKDIV_REG	时钟分频配置	0x0014	读/写
UART_CONF0_REG	配置寄存器 0	0x0020	读/写
UART_CONF1_REG	配置寄存器 1	0x0024	读/写
UART_FLOW_CONF_REG	软件流控配置	0x0034	读/写
UART_SLEEP_CONF_REG	睡眠模式配置	0x0038	读/写
UART_SWFC_CONF0_REG	软件流控字符配置	0x003C	读/写
UART_SWFC_CONF1_REG	软件流控字符配置	0x0040	读/写
UART_IDLE_CONF_REG	帧结束空闲配置	0x0044	读/写
UART_RS485_CONF_REG	RS485 模式配置	0x0048	读/写
自动波特率检测寄存器			

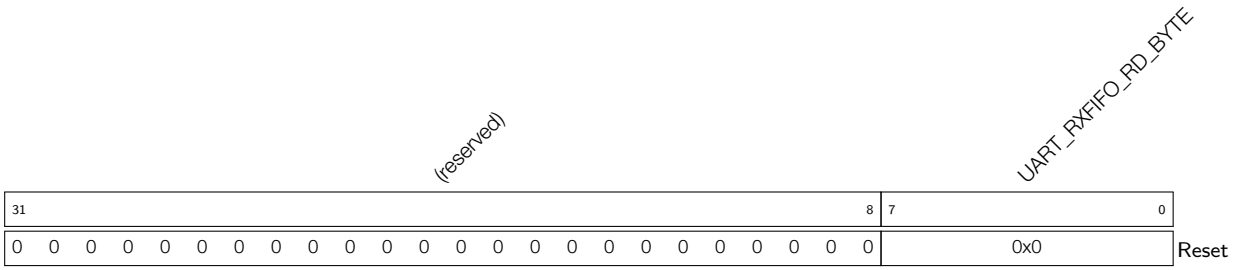
名称	描述	地址	访问
UART_AUTOBAUD_REG	自动波特率检测配置寄存器	0x0018	读/写
UART_LOWPULSE_REG	自动波特率检测最低低电平脉冲持续时间寄存器	0x0028	只读
UART_HIGHPULSE_REG	自动波特率检测最短高电平脉冲持续时间寄存器	0x002C	只读
UART_RXD_CNT_REG	自动波特率检测沿变化计数寄存器	0x0030	只读
UART_POSPULSE_REG	自动波特率检测高电平脉冲寄存器	0x006C	只读
UART_NEGPULSE_REG	自动波特率检测低电平脉冲寄存器	0x0070	只读
状态寄存器			
UART_STATUS_REG	UART 状态寄存器	0x001C	只读
UART_MEM_TX_STATUS_REG	TX FIFO 写入、读取偏移地址	0x0060	只读
UART_MEM_RX_STATUS_REG	RX FIFO 写入、读取偏移地址	0x0064	只读
UART_FSM_STATUS_REG	UART 发送和接收状态	0x0068	只读
AT 转义序列检测配置			
UART_AT_CMD_PRECNT_REG	AT_CMD 字符序列发送前的时序配置	0x004C	读/写
UART_AT_CMD_POSTCNT_REG	AT_CMD 字符发送后的时序配置	0x0050	读/写
UART_AT_CMD_GAPTOUT_REG	超时配置	0x0054	读/写
UART_AT_CMD_CHAR_REG	AT 转义序列检测配置	0x0058	读/写
版本寄存器			
UART_DATE_REG	UART 版本控制寄存器	0x0074	读/写

名称	描述	地址	访问
配置寄存器			
UHCI_CONF0_REG	UHCI 配置寄存器	0x0000	读/写
UHCI_CONF1_REG	UHCI 配置寄存器	0x002C	读/写
UHCI_AHB_TEST_REG	AHB 测试寄存器	0x0048	读/写
UHCI_ESCAPE_CONF_REG	转义符配置	0x0064	读/写
UHCI_HUNG_CONF_REG	超时配置	0x0068	读/写
UHCI_QUICK_SENT_REG	UHCI 快速发送配置寄存器	0x0074	读/写
UHCI_REG_Q0_WORD0_REG	Q0_WORD0 快速发送寄存器	0x0078	读/写
UHCI_REG_Q0_WORD1_REG	Q0_WORD1 快速发送寄存器	0x007C	读/写
UHCI_REG_Q1_WORD0_REG	Q1_WORD0 快速发送寄存器	0x0080	读/写
UHCI_REG_Q1_WORD1_REG	Q1_WORD1 快速发送寄存器	0x0084	读/写
UHCI_REG_Q2_WORD0_REG	Q2_WORD0 快速发送寄存器	0x0088	读/写
UHCI_REG_Q2_WORD1_REG	Q2_WORD1 快速发送寄存器	0x008C	读/写
UHCI_REG_Q3_WORD0_REG	Q3_WORD0 快速发送寄存器	0x0090	读/写
UHCI_REG_Q3_WORD1_REG	Q3_WORD1 快速发送寄存器	0x0094	读/写
UHCI_REG_Q4_WORD0_REG	Q4_WORD0 快速发送寄存器	0x0098	读/写
UHCI_REG_Q4_WORD1_REG	Q4_WORD1 快速发送寄存器	0x009C	读/写
UHCI_REG_Q5_WORD0_REG	Q5_WORD0 快速发送寄存器	0x00A0	读/写
UHCI_REG_Q5_WORD1_REG	Q5_WORD1 快速发送寄存器	0x00A4	读/写
UHCI_REG_Q6_WORD0_REG	Q6_WORD0 快速发送寄存器	0x00A8	读/写
UHCI_REG_Q6_WORD1_REG	Q6_WORD1 快速发送寄存器	0x00AC	读/写

名称	描述	地址	访问
UHCI_ESC_CONF0_REG	转义序列配置寄存器 0	0x00B0	读/写
UHCI_ESC_CONF1_REG	转义序列配置寄存器 1	0x00B4	读/写
UHCI_ESC_CONF2_REG	转义序列配置寄存器 2	0x00B8	读/写
UHCI_ESC_CONF3_REG	转义序列配置寄存器 3	0x00BC	读/写
UHCI_PKT_THRES_REG	包长度配置寄存器	0x00C0	读/写
中断寄存器			
UHCI_INT_RAW_REG	原始中断状态	0x0004	只读
UHCI_INT_ST_REG	屏蔽中断状态	0x0008	只读
UHCI_INT_ENA_REG	中断使能位	0x000C	读/写
UHCI_INT_CLR_REG	中断清除位	0x0010	只写
DMA 状态			
UHCI_DMA_OUT_STATUS_REG	DMA 数据输出状态寄存器	0x0014	只读
UHCI_DMA_IN_STATUS_REG	UHCI 数据输入状态寄存器	0x001C	只读
UHCI_STATE0_REG	UHCI 解码器状态寄存器	0x0030	只读
UHCI_DMA_OUT_EOF_DES_ADDR_REG	EOF 有效时发送链表描述符的地址	0x0038	只读
UHCI_DMA_IN_SUC_EOF_DES_ADDR_REG	EOF 有效时接收链表描述符的地址	0x003C	只读
UHCI_DMA_IN_ERR_EOF_DES_ADDR_REG	报错时接收链表描述符的地址	0x0040	只读
UHCI_DMA_OUT_EOF_BFR_DES_ADDR_REG	倒数第二个发送链表描述符	0x0044	只读
UHCI_DMA_IN_DSCR_REG	下一个接收链表描述符的第三个字	0x004C	只读
UHCI_DMA_IN_DSCR_BF0_REG	当前接收链表描述符的第三个字	0x0050	只读
UHCI_DMA_IN_DSCR_BF1_REG	当前接收链表描述符的第二个字	0x0054	只读
UHCI_DMA_OUT_DSCR_REG	下一个接收链表描述符的第三个字	0x0058	只读
UHCI_DMA_OUT_DSCR_BF0_REG	当前接收链表描述符的第三个字	0x005C	只读
UHCI_DMA_OUT_DSCR_BF1_REG	当前发送链表描述符的第二个字	0x0060	只读
UHCI_RX_HEAD_REG	UHCI 包报头寄存器	0x0070	只读
UHCI_STATE1_REG	UHCI 编码状态寄存器	0x0034	只读
DMA 配置			
UHCI_DMA_OUT_PUSH_REG	数据输出 FIFO 推送控制寄存器	0x0018	读/写
UHCI_DMA_IN_POP_REG	数据输入 FIFO 的弹出控制寄存器	0x0020	不定
UHCI_DMA_OUT_LINK_REG	链表描述符地址与控制	0x0024	不定
UHCI_DMA_IN_LINK_REG	链表描述符地址与控制	0x0028	不定
版本寄存器			
UHCI_DATE_REG	UHCI 版本控制寄存器	0x00FC	读/写

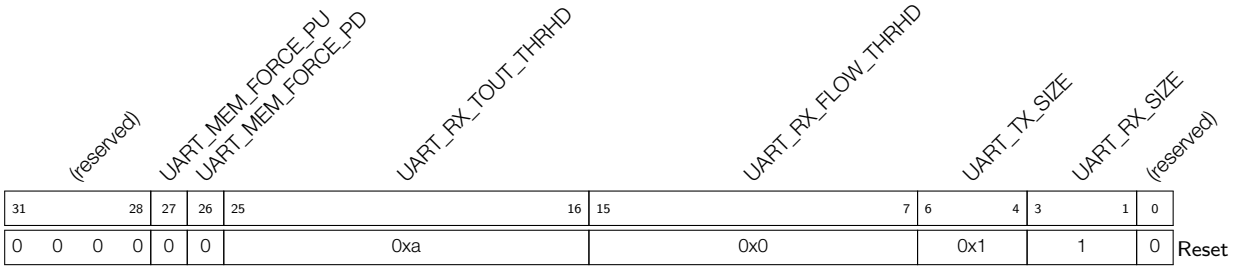
8.6 寄存器

Register 8.1: UART_FIFO_REG (0x0000)



UART_RXFIFO_RD_BYTE 存储从 RX FIFO 读取的一个字节数据。(只读)

Register 8.2: UART_MEM_CONF_REG (0x005C)



UART_RX_SIZE 配置存储器分配给 RX FIFO 的空间大小。默认为 128 字节。(读/写)

UART_TX_SIZE 配置存储器分配给 TX FIFO 的空间大小。默认为 128 字节。(读/写)

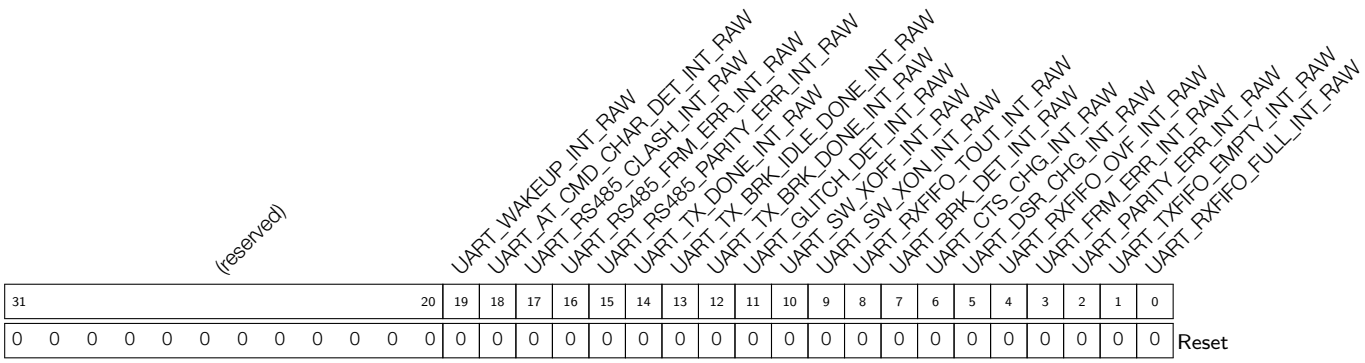
UART_RX_FLOW_THRHD 配置使用硬件流控时接收数据的最大值。(读/写)

UART_RX_TOUT_THRHD 配置接收器接收一个字节所需时间的阈值。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。(读/写)

UART_MEM_FORCE_PD 置位此位强制关闭 UART 存储器。(读/写)

UART_MEM_FORCE_PU 置位此位强制开启 UART 存储器。(读/写)

Register 8.3: UART_INT_RAW_REG (0x0004)



UART_RXFIFO_FULL_INT_RAW 接收器接收数据多于 UART_RXFIFO_FULL_THRHD 的指定值时，该原始中断位翻转至高电平。(只读)

UART_TXFIFO_EMPTY_INT_RAW TX FIFO 中的数据少于 UART_TXFIFO_EMPTY_THRHD 的指定值时，该原始中断位翻转至高电平。(只读)

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。(只读)

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。(只读)

UART_RXFIFO_OVF_INT_RAW 接收器接收数据超过 FIFO 的存储容量时，该原始中断位翻转至高电平。(只读)

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。(只读)

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。(只读)

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。(只读)

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。(只读)

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。(只读)

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。(只读)

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。(只读)

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。(只读)

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。(只读)

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。(只读)

见下页...

Register 8.3: UART_INT_RAW_REG (0x0004)

接上页...

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。(只读)

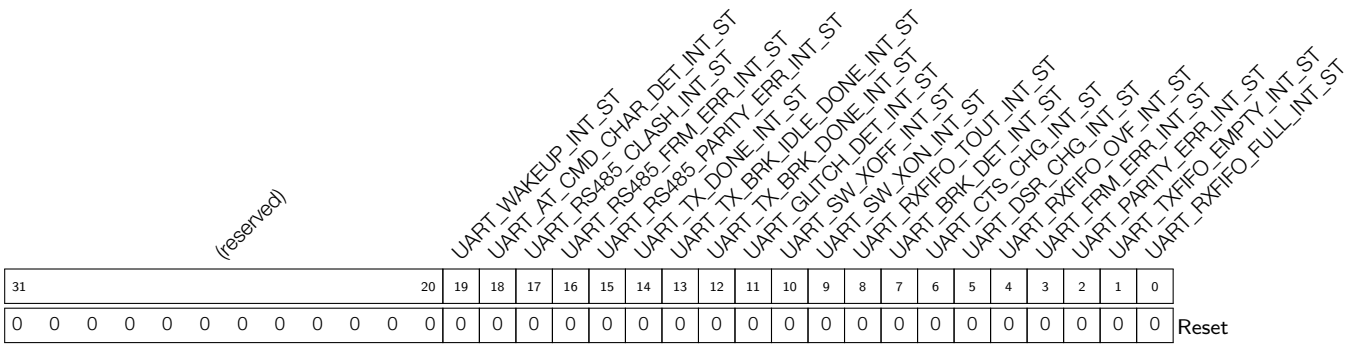
UART_RS485_FRM_ERR_INT_RAW RS485 模式下接收器检测到发送器回音的数据帧错误时，该原始中断位翻转至高电平。(只读)

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。(只读)

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。(只读)

UART_WAKEUP_INT_RAW 输入 RXD 沿变化次数超过 Light-sleep 模式指定的 UART_ACTIVE_THRESHOLD 值时，该原始中断位翻转至高电平。(只读)

Register 8.4: UART_INT_ST_REG (0x0008)



UART_RXFIFO_FULL_INT_ST UART_RXFIFO_FULL_INT_ENA 置 1 时 UART_RXFIFO_FULL_INT_RAW 的状态位。(只读)

UART_TXFIFO_EMPTY_INT_ST UART_TXFIFO_EMPTY_INT_ENA 置 1 时 UART_TXFIFO_EMPTY_INT_RAW 的状态位。(只读)

UART_PARITY_ERR_INT_ST UART_PARITY_ERR_INT_ENA 置 1 时 UART_PARITY_ERR_INT_RAW 的状态位。(只读)

UART_FRM_ERR_INT_ST UART_FRM_ERR_INT_ENA 置 1 时 UART_FRM_ERR_INT_RAW 的状态位。(只读)

UART_RXFIFO_OVF_INT_ST UART_RXFIFO_OVF_INT_ENA 置 1 时 UART_RXFIFO_OVF_INT_RAW 的状态位。(只读)

UART_DSR_CHG_INT_ST UART_DSR_CHG_INT_ENA 置 1 时 UART_DSR_CHG_INT_RAW 的状态位。(只读)

UART_CTS_CHG_INT_ST UART_CTS_CHG_INT_ENA 置 1 时 UART_CTS_CHG_INT_RAW 的状态位。(只读)

UART_BRK_DET_INT_ST UART_BRK_DET_INT_ENA 置 1 时 UART_BRK_DET_INT_RAW 的状态位。(只读)

UART_RXFIFO_TOUT_INT_ST UART_RXFIFO_TOUT_INT_ENA 置 1 时 UART_RXFIFO_TOUT_INT_RAW 的状态位。(只读)

UART_SW_XON_INT_ST UART_SW_XON_INT_ENA 置 1 时 UART_SW_XON_INT_RAW 的状态位。(只读)

UART_SW_XOFF_INT_ST UART_SW_XOFF_INT_ENA 置 1 时 UART_SW_XOFF_INT_RAW 的状态位。(只读)

UART_GLITCH_DET_INT_ST UART_GLITCH_DET_INT_ENA 置 1 时 UART_GLITCH_DET_INT_RAW 的状态位。(只读)

UART_TX_BRK_DONE_INT_ST UART_TX_BRK_DONE_INT_ENA 置 1 时 UART_TX_BRK_DONE_INT_RAW 的状态位。(只读)

UART_TX_BRK_IDLE_DONE_INT_ST UART_TX_BRK_IDLE_DONE_INT_ENA 置 1 时 UART_TX_BRK_IDLE_DONE_INT_RAW 的状态位。(只读)

见下页...

Register 8.4: UART_INT_ST_REG (0x0008)

接上页...

UART_TX_DONE_INT_ST UART_TX_DONE_INT_ENA 置 1 时 UART_TX_DONE_INT_RAW 的状态位。(只读)

UART_RS485_PARITY_ERR_INT_ST UART_RS485_PARITY_INT_ENA 置 1 时 UART_RS485_PARITY_ERR_INT_RAW 的状态位。(只读)

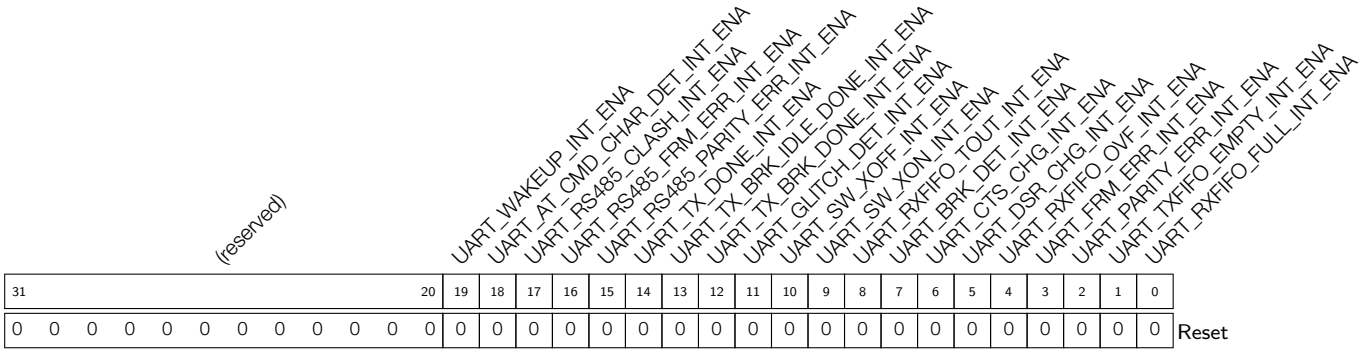
UART_RS485_FRM_ERR_INT_ST UART_RS485_FM_ERR_INT_ENA 置 1 时 UART_RS485_FRM_ERR_INT_RAW 的状态位。(只读)

UART_RS485_CLASH_INT_ST UART_RS485_CLASH_INT_ENA 置 1 时 UART_RS485_CLASH_INT_RAW 的状态位。(只读)

UART_AT_CMD_CHAR_DET_INT_ST UART_AT_CMD_CHAR_DET_INT_ENA 置 1 时 UART_AT_CMD_DET_INT_RAW 的状态位。(只读)

UART_WAKEUP_INT_ST UART_WAKEUP_INT_ENA 置 1 时 UART_WAKEUP_INT_RAW 的状态位。(只读)

Register 8.5: UART_INT_ENA_REG (0x000C)



UART_RXFIFO_FULL_INT_ENA UART_RXFIFO_FULL_INT_ST 寄存器的使能位。(读/写)

UART_TXFIFO_EMPTY_INT_ENA UART_TXFIFO_EMPTY_INT_ST 寄存器的使能位。(读/写)

UART_PARITY_ERR_INT_ENA UART_PARITY_ERR_INT_ST 寄存器的使能位。(读/写)

UART_FRM_ERR_INT_ENA UART_FRM_ERR_INT_ST 寄存器的使能位。(读/写)

UART_RXFIFO_OVF_INT_ENA UART_RXFIFO_OVF_INT_ST 寄存器的使能位。(读/写)

UART_DSR_CHG_INT_ENA UART_DSR_CHG_INT_ST 寄存器的使能位。(读/写)

UART_CTS_CHG_INT_ENA UART_CTS_CHG_INT_ST 寄存器的使能位。(读/写)

UART_BRK_DET_INT_ENA UART_BRK_DET_INT_ST 寄存器的使能位。(读/写)

UART_RXFIFO_TOUT_INT_ENA UART_RXFIFO_TOUT_INT_ST 寄存器的使能位。(读/写)

UART_SW_XON_INT_ENA UART_SW_XON_INT_ST 寄存器的使能位。(读/写)

UART_SW_XOFF_INT_ENA UART_SW_XOFF_INT_ST 寄存器的使能位。(读/写)

UART_GLITCH_DET_INT_ENA UART_GLITCH_DET_INT_ST 寄存器的使能位。(读/写)

UART_TX_BRK_DONE_INT_ENA UART_TX_BRK_DONE_INT_ST 寄存器的使能位。(读/写)

UART_TX_BRK_IDLE_DONE_INT_ENA UART_TX_BRK_IDLE_DONE_INT_ST 寄存器的使能位。(读/写)

UART_TX_DONE_INT_ENA UART_TX_DONE_INT_ST 寄存器的使能位。(读/写)

UART_RS485_PARITY_ERR_INT_ENA UART_RS485_PARITY_ERR_INT_ST 寄存器的使能位。(读/写)

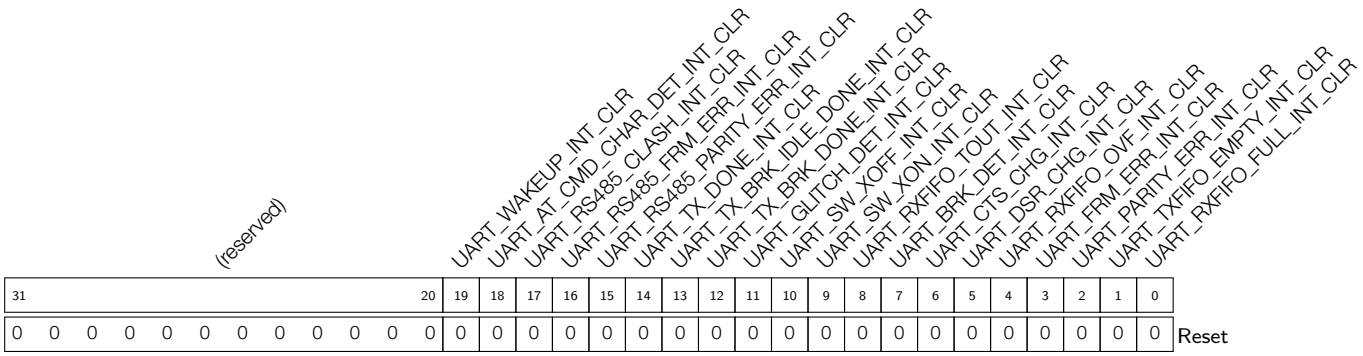
UART_RS485_FRM_ERR_INT_ENA UART_RS485_PARITY_ERR_INT_ST 寄存器的使能位。(读/写)

UART_RS485_CLASH_INT_ENA UART_RS485_CLASH_INT_ST 寄存器的使能位。(读/写)

UART_AT_CMD_CHAR_DET_INT_ENA UART_AT_CMD_CHAR_DET_INT_ST 寄存器的使能位。(读/写)

UART_WAKEUP_INT_ENA UART_WAKEUP_INT_ST 寄存器的使能位。(读/写)

Register 8.6: UART_INT_CLR_REG (0x0010)



UART_RXFIFO_FULL_INT_CLR 置位此位清除 UART_RXFIFO_FULL_INT_RAW 中断。(只写)

UART_TXFIFO_EMPTY_INT_CLR 置位此位清除 UART_TXFIFO_EMPTY_INT_RAW 中断。(只写)

UART_PARITY_ERR_INT_CLR 置位此位清除 UART_PARITY_ERR_INT_RAW 中断。(只写)

UART_FRM_ERR_INT_CLR 置位此位清除 UART_FRM_ERR_INT_RAW 中断。(只写)

UART_RXFIFO_OVF_INT_CLR 置位此位清除 UART_RXFIFO_OVF_INT_RAW 中断。(只写)

UART_DSR_CHG_INT_CLR 置位此位清除 UART_DSR_CHG_INT_RAW 中断。(只写)

UART_CTS_CHG_INT_CLR 置位此位清除 UART_CTS_CHG_INT_RAW 中断。(只写)

UART_BRK_DET_INT_CLR 置位此位清除 UART_BRK_DET_INT_RAW 中断。(只写)

UART_RXFIFO_TOUT_INT_CLR 置位此位清除 UART_RXFIFO_TOUT_INT_RAW 中断。(只写)

UART_SW_XON_INT_CLR 置位此位清除 UART_SW_XON_INT_RAW 中断。(只写)

UART_SW_XOFF_INT_CLR 置位此位清除 UART_SW_XOFF_INT_RAW 中断。(只写)

UART_GLITCH_DET_INT_CLR 置位此位清除 UART_GLITCH_DET_INT_RAW 中断。(只写)

UART_TX_BRK_DONE_INT_CLR 置位此位清除 UART_TX_BRK_DONE_INT_RAW 中断。(只写)

UART_TX_BRK_IDLE_DONE_INT_CLR 置位此位清除 UART_TX_BRK_IDLE_DONE_INT_RAW 中断。(只写)

UART_TX_DONE_INT_CLR 置位此位清除 UART_TX_DONE_INT_RAW 中断。(只写)

UART_RS485_PARITY_ERR_INT_CLR 置位此位清除 UART_RS485_PARITY_ERR_INT_RAW 中断。(只写)

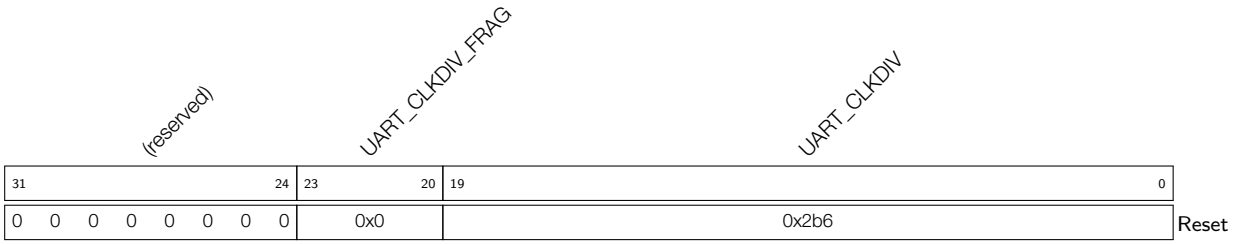
UART_RS485_FRM_ERR_INT_CLR 置位此位清除 UART_RS485_FRM_ERR_INT_RAW 中断。(只写)

UART_RS485_CLASH_INT_CLR 置位此位清除 UART_RS485_CLASH_INT_RAW 中断。(只写)

UART_AT_CMD_CHAR_DET_INT_CLR 置位此位清除 UART_AT_CMD_CHAR_DET_INT_RAW 中断。(只写)

UART_WAKEUP_INT_CLR 置位此位清除 UART_WAKEUP_INT_RAW 中断。(只写)

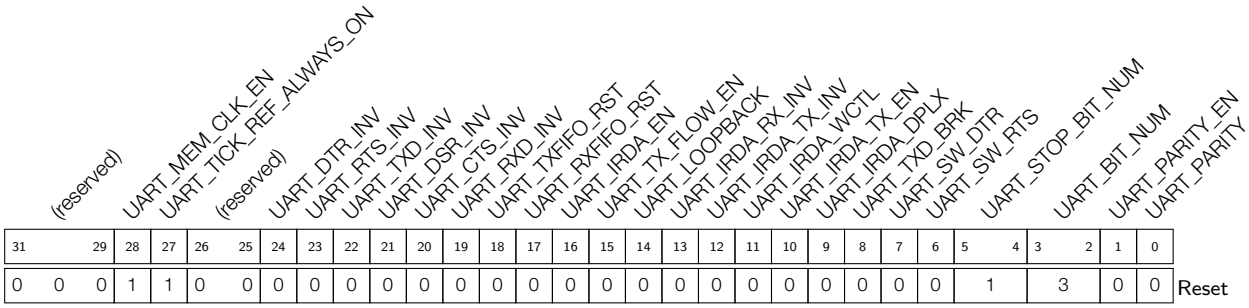
Register 8.7: UART_CLKDIV_REG (0x0014)



UART_CLKDIV 分频系数的整数部分。(读/写)

UART_CLKDIV_FRAG 分频系数的小数部分。(读/写)

Register 8.8: UART_CONF0_REG (0x0020)



UART_PARITY 配置奇偶检验方式。1’ h0：偶检验。1’h1：奇检验。(读/写)

UART_PARITY_EN 置位此位使能 UART 奇偶检验。(读/写)

UART_BIT_NUM 设置数据长度。0：5 位 1：6 位 2：7 位 3：8 位。(读/写)

UART_STOP_BIT_NUM 设置停止位的长度。1：1 位 2：1.5 位 3：2 位 (读/写)

UART_SW_RTS 该寄存器用于配置软件流控使用的软件 RTS 信号。(读/写)

UART_SW_DTR 该寄存器用于配置软件流控使用的软件 DTR 信号。(读/写)

UART_TXD_BRK 置位此位，使能发送器在发完数据后发送 NULL。(读/写)

UART_IRDA_DPLX 置位此位开启 IrDA 回环测试模式。(读/写)

UART_IRDA_TX_EN IrDA 发送器的启动使能位。(读/写)

UART_IRDA_WCTL 1’h1：IrDA 发送器的第 11 位与第 10 位相同。1’ h0：将 IrDA 发送器的第 11 位置 0。(读/写)

UART_IRDA_TX_INV 置位此位翻转 IrDA 发送器的电平。(读/写)

UART_IRDA_RX_INV 置位此位翻转 IrDA 接收器的电平。(读/写)

UART_LOOPBACK 置位此位开启 UART 回环测试模式。(读/写)

UART_TX_FLOW_EN 置位此位使能发送器的流控功能。(读/写)

UART_IRDA_EN 置位此位使能 IrDA 协议。(读/写)

UART_RXFIFO_RST 置位此位复位 UART RX FIFO。(读/写)

UART_TXFIFO_RST 置位此位复位 UART TX FIFO。(读/写)

UART_RXD_INV 置位此位翻转 UART RXD 信号电平。(读/写)

UART_CTS_INV 置位此位翻转 UART CTS 信号电平。(读/写)

UART_DSR_INV 置位此位翻转 UART DSR 信号电平。(读/写)

UART_TXD_INV 置位此位翻转 UART TXD 信号电平。(读/写)

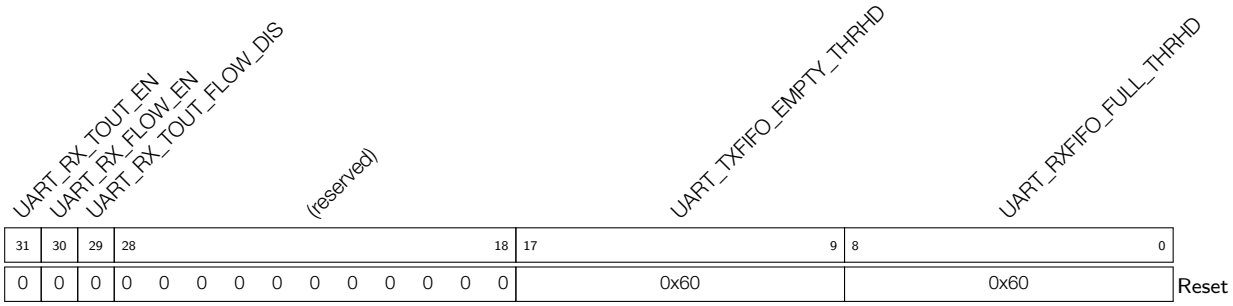
UART_RTS_INV 置位此位翻转 UART RTS 信号电平。(读/写)

UART_DTR_INV 置位此位翻转 UART DTR 信号电平。(读/写)

UART_TICK_REF_ALWAYS_ON 该寄存器用于选取时钟。1’h1：APB_CLK。1’h0：REF_TICK。(读/写)

UART_MEM_CLK_EN UART 存储器门控使能信号。1：UART 存储器开启，可读取、写入数据。0：UART 存储器关闭。(读/写)

Register 8.9: UART_CONF1_REG (0x0024)



UART_RXFIFO_FULL_THRHD 接收器接收数据多于该寄存器的值时产生 UART_RXFIFO_FULL_INT 中断。(读/写)

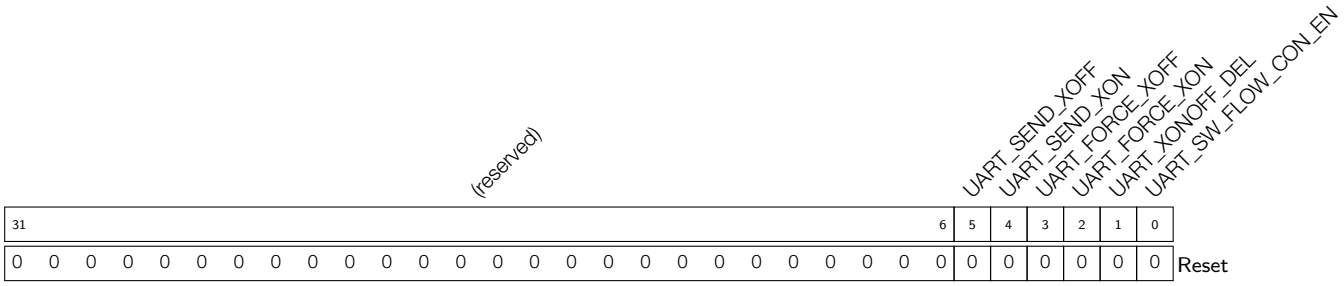
UART_TXFIFO_EMPTY_THRHD TX FIFO 中的数据少于该寄存器的值时产生 UART_TXFIFO_EMPTY_INT 中断。(读/写)

UART_RX_TOUT_FLOW_DIS 使用硬件流控时置位此位停止堆积 idle_cnt。(读/写)

UART_RX_FLOW_EN UART 接收器流控功能的使能位。1'h1：配置 sw_rts 信号选择软件流控。1'h0：关闭软件流控。(读/写)

UART_RX_TOUT_EN UART 接收器超时功能的使能位。(读/写)

Register 8.10: UART_FLOW_CONF_REG (0x0034)



UART_SW_FLOW_CON_EN 置位此位使能软件流控，与 SW_XON 或 SW_XOFF 寄存器一起使用。(读/写)

UART_XONOFF_DEL 置位此位移除接收数据中的流控字符。(读/写)

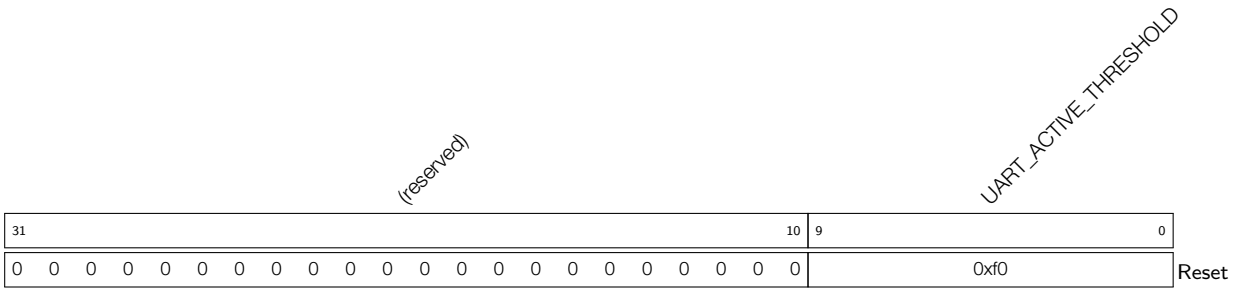
UART_FORCE_XON 置位此位让发送器继续发送数据。(读/写)

UART_FORCE_XOFF 置位此位让发送器停止发送数据。(读/写)

UART_SEND_XON 置位此位发送 XON 字符。此位由硬件自动清除。(读/写)

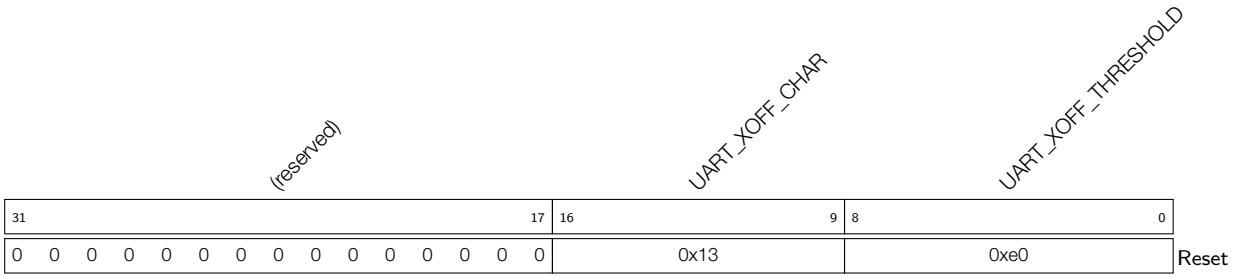
UART_SEND_XOFF 置位此位发送 XOFF 字符。此位由硬件自动清除。(读/写)

Register 8.11: UART_SLEEP_CONF_REG (0x0038)



UART_ACTIVE_THRESHOLD 输入 RXD 沿变化次数超过该寄存器的值时，UART 从 Light-sleep 模式唤醒。(读/写)

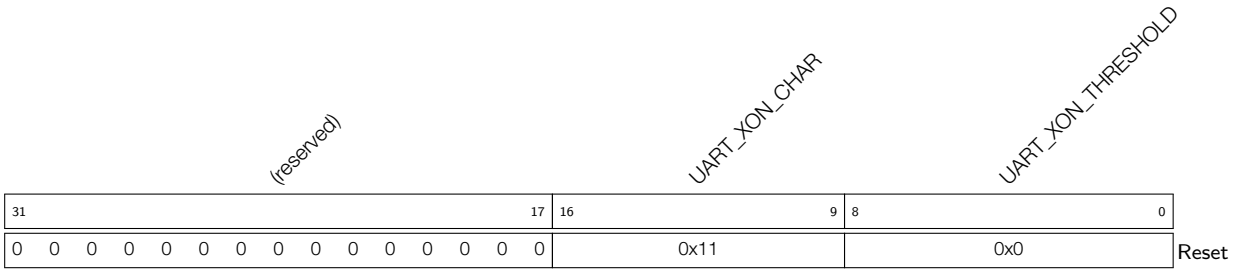
Register 8.12: UART_SWFC_CONF0_REG (0x003C)



UART_XOFF_THRESHOLD RX FIFO 中的数据超过该寄存器的值且 UART_SW_FLOW_CON_EN 置 1 时，发送 XOFF 字符。(读/写)

UART_XOFF_CHAR 存储 XOFF 流控字符。(读/写)

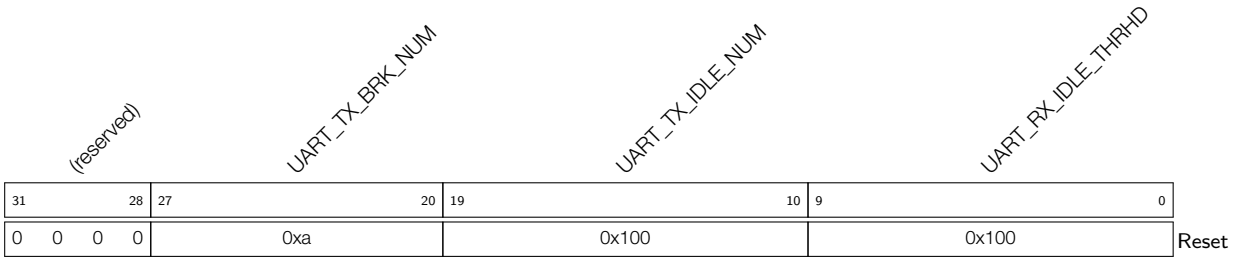
Register 8.13: UART_SWFC_CONF1_REG (0x0040)



UART_XON_THRESHOLD RX FIFO 中的数据小于该寄存器的值且 UART_SW_FLOW_CON_EN 置 1 时，发送 XON 字符。(读/写)

UART_XON_CHAR 存储 XON 流控字符。(读/写)

Register 8.14: UART_IDLE_CONF_REG (0x0044)

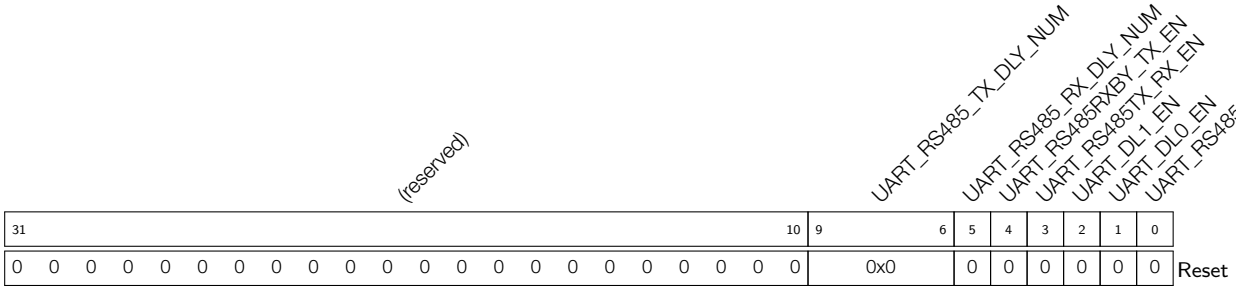


UART_RX_IDLE_THRHD 接收器接收一字节数据所需时间超过该寄存器的值时产生帧结束信号。(读/写)

UART_TX_IDLE_NUM 配置两次数据传输的间隔时间。(读/写)

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。(读/写)

Register 8.15: UART_RS485_CONF_REG (0x0048)



UART_RS485_EN 置位此位选择 RS485 模式。(读/写)

UART_DL0_EN 置位此位，延迟停止位 1 位。(读/写)

UART_DL1_EN 置位此位，延迟停止位 1 位。(读/写)

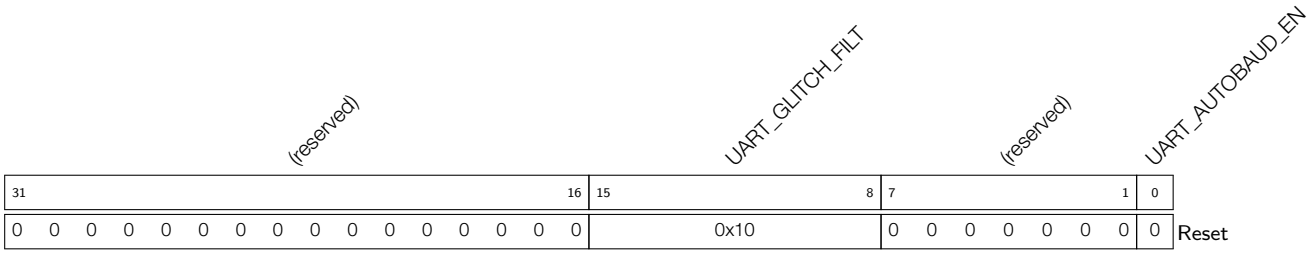
UART_RS485TX_RX_EN 发送器在 RS485 模式下发送数据时，置位此位使能接收器接收数据。(读/写)

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。1' h0：RS485 接收器线路繁忙时 RS485 发送器不发送数据。(读/写)

UART_RS485_RX_DLY_NUM 延迟接收器的内部数据信号。(读/写)

UART_RS485_TX_DLY_NUM 延迟发送器的内部数据信号。(读/写)

Register 8.16: UART_AUTOBAUD_REG (0x0018)



UART_AUTOBAUD_EN 波特率检测的使能位。(读/写)

UART_GLITCH_FILT 宽度小于该寄存器值的输入脉冲会被忽略。该寄存器用于自动波特率检测。(读/写)

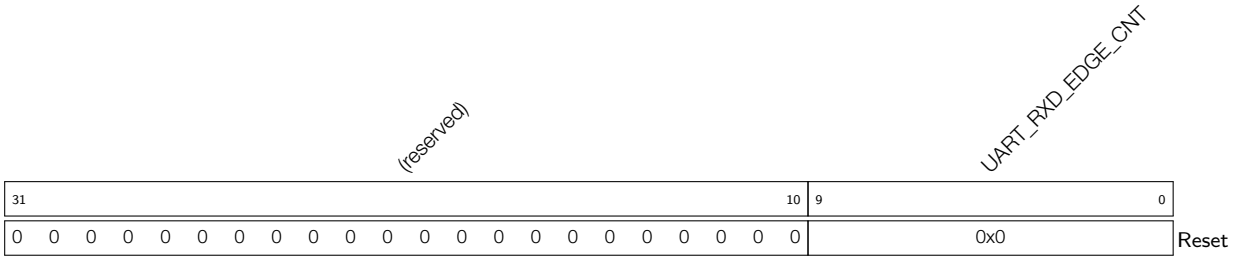
Register 8.17: UART_LOWPULSE_REG (0x0028)

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。(只读)

Register 8.18: 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。(只读)

Register 8.19: 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。(只读)

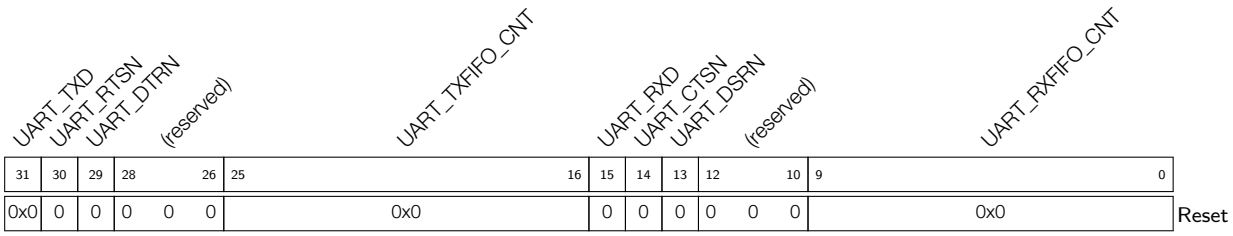
Register 8.20: UART_POSPULSE_REG (0x006C)

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。(只读)

Register 8.21: UART_NEGPULSE_REG (0x0070)

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。(只读)

Register 8.22: UART_STATUS_REG (0x001C)



UART_RXFIFO_CNT 存储 RX FIFO 中有效数据的字节数。(只读)

UART_DSRN 该寄存器表示内部 UART DSR 信号的电平值。(只读)

UART_CTSN 该寄存器表示内部 UART CTS 信号的电平值。(只读)

UART_RXD 该寄存器表示内部 UART RXD 信号的电平值。(只读)

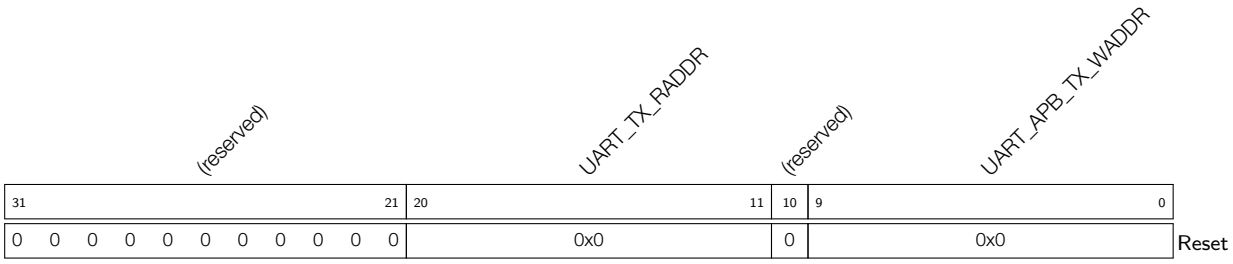
UART_TXFIFO_CNT 存储 TX FIFO 中数据的字节数。(只读)

UART_DTRN 此位表示内部 UART DTR 信号的电平。(只读)

UART_RTSN 此位表示内部 UART RTS 信号的电平。(只读)

UART_TXD 此位表示内部 UART TXD 信号的电平。(只读)

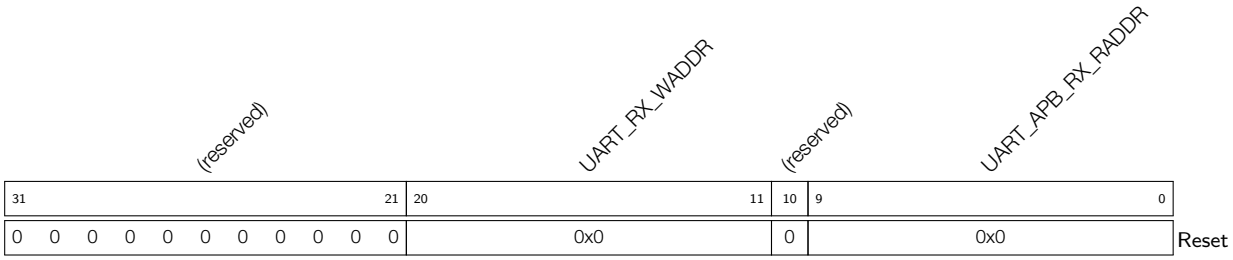
Register 8.23: UART_MEM_TX_STATUS_REG (0x0060)



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。(只读)

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。(只读)

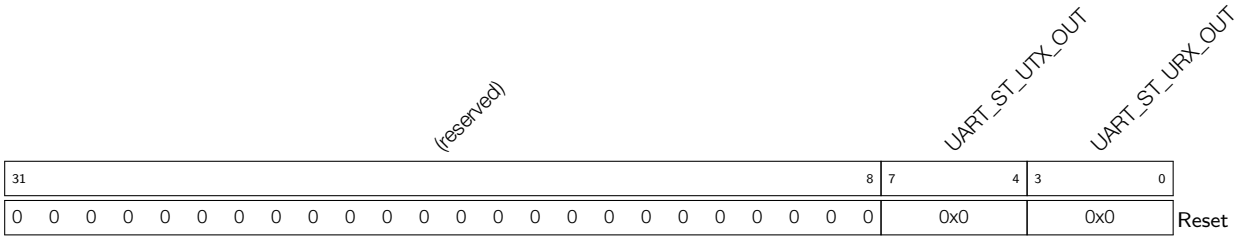
Register 8.24: UART_MEM_RX_STATUS_REG (0x0064)



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。(只读)

UART_RX_WADDR 在 Rx_FIFO_Ctrl 写 RX FIFO 时存储 RX FIFO 的偏移地址。(只读)

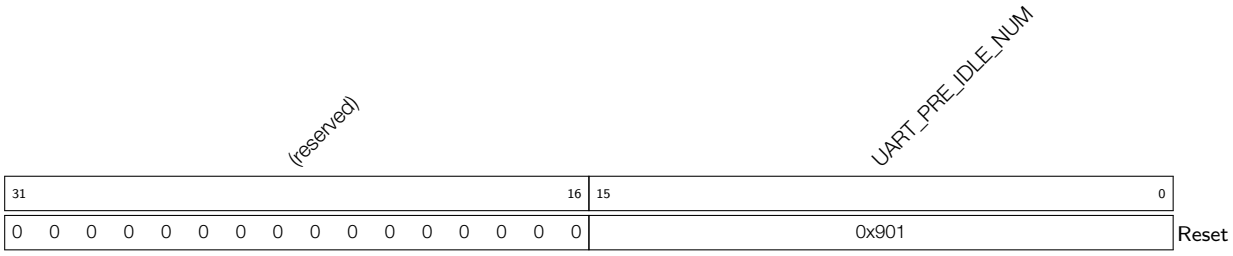
Register 8.25: UART_FSM_STATUS_REG (0x0068)



UART_ST_URX_OUT 接收器的状态寄存器。(只读)

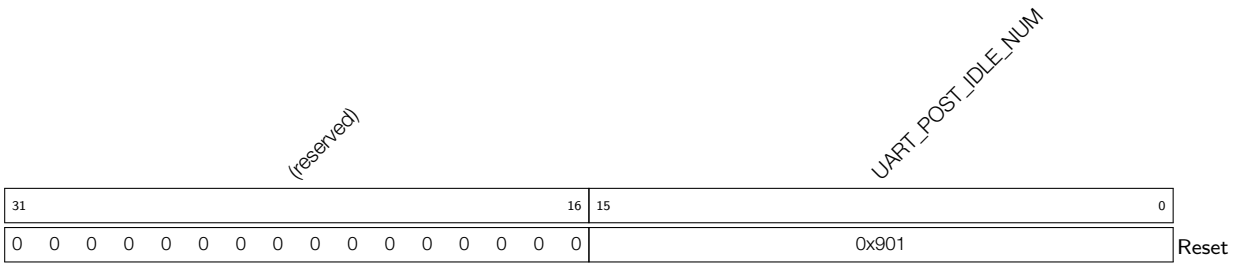
UART_ST_UTX_OUT 发送器的状态寄存器。(只读)

Register 8.26: UART_AT_CMD_PRECNT_REG (0x004C)



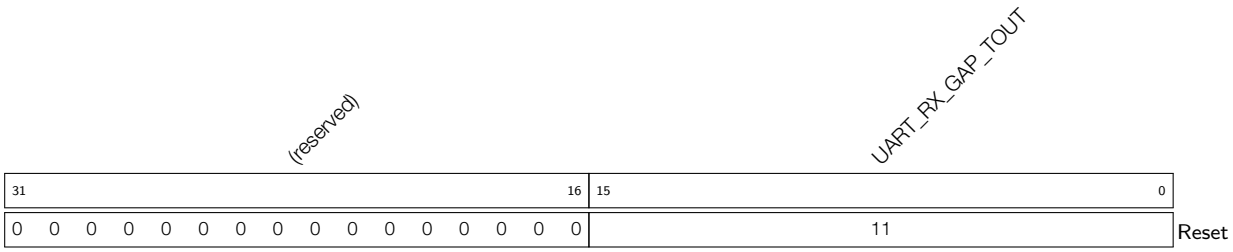
UART_PRE_IDLE_NUM 配置接收器接收第一个 AT_CMD 字符前的空闲时间。间隔时间小于该寄存器的值时，下一个接收数据不会被视作 AT_CMD 字符。(读/写)

Register 8.27: UART_AT_CMD_POSTCNT_REG (0x0050)



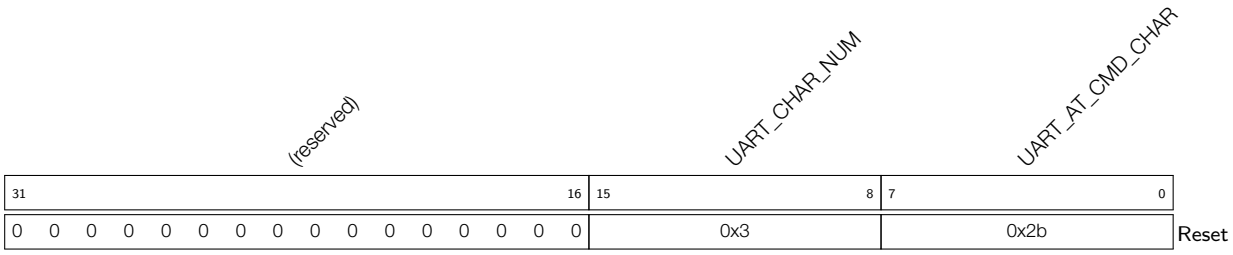
UART_POST_IDLE_NUM 配置最后一个 AT_CMD 字符和后续数据的间隔时间。间隔时间小于该寄存器的值时，前一个数据不会被视作 AT_CMD 字符。(读/写)

Register 8.28: UART_AT_CMD_GAPTOUT_REG (0x0054)



UART_RX_GAP_TOUT 配置 AT_CMD 字符的间隔时间。持续/间隔时间小于该寄存器的值时，数据不会被视作连续的 AT_CMD 字符。(读/写)

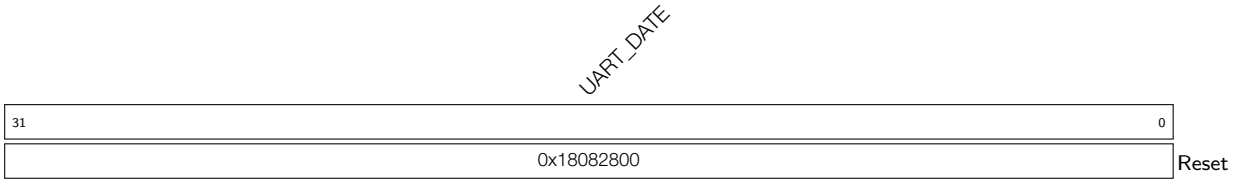
Register 8.29: UART_AT_CMD_CHAR_REG (0x0058)



UART_AT_CMD_CHAR 配置 AT_CMD 字符的内容。(读/写)

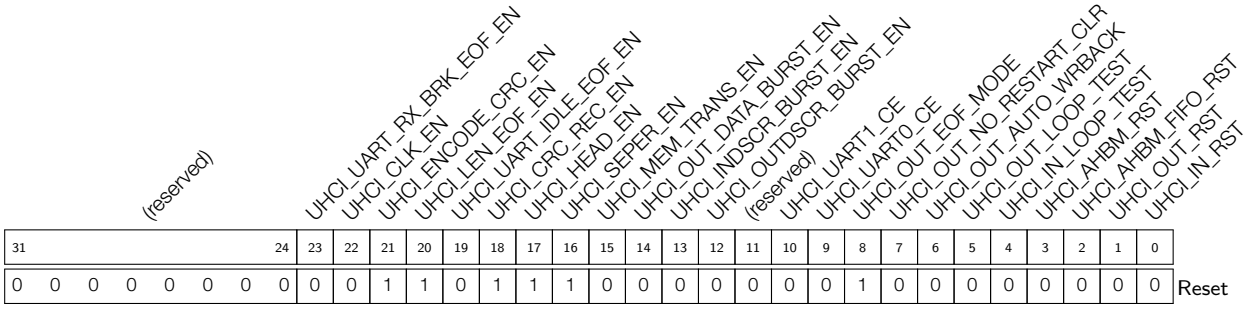
UART_CHAR_NUM 配置接收器接收连续 AT_CMD 字符的个数。(读/写)

Register 8.30: UART_DATE_REG (0x0074)



UART_DATE 版本控制寄存器。(读/写)

Register 8.31: UHCI_CONF0_REG (0x0000)



UHCI_IN_RST 置位此位，复位接收 DMA FSM。(读/写)

UHCI_OUT_RST 置位此位，复位发送 DMA FSM。(读/写)

UHCI_AHBM_FIFO_RST 置位此位，复位 DMA cmdFIFO 的 AHB 接口。(读/写)

UHCI_AHBM_RST 置位此位，复位 DMA 的 AHB 接口。(读/写)

UHCI_IN_LOOP_TEST 保留。(读/写)

UHCI_OUT_LOOP_TEST 保留。(读/写)

UHCI_OUT_AUTO_WRBACK 置位此位，在 TX Buffer 中所有数据传输完毕后自动回写发送链表。(读/写)

UHCI_OUT_NO_RESTART_CLR (读/写)

UHCI_OUT_EOF_MODE 用于选择 UHCI_OUT_EOF_INT 的产生条件。1：DMA 将所有数据从 FIFO 中弹出时产生中断。0：AHB 将所有数据推送到 FIFO 时产生中断。(读/写)

UHCI_UART0_CE 置位此位，将 HCI 和 UART0 相连。(读/写)

UHCI_UART1_CE 置位此位，将 HCI 和 UART1 相连。(读/写)

UHCI_OUTDSCR_BURST_EN 用于选择 DMA 发送链表描述符传输模式。1：突发模式。0：字节模式。(读/写)

UHCI_INDSCR_BURST_EN 用于选择 DMA 接收链表描述符传输模式。1：突发模式。0：字节模式。(读/写)

UHCI_OUT_DATA_BURST_EN 用于选择数据传输模式。1：突发模式传输数据。0：字节模式传输数据。(读/写)

UHCI_MEM_TRANS_EN 1：UHCI 发送数据写回 DMA INFIFO。(读/写)

UHCI_SEPER_EN 置位此位，使用特殊字符分隔数据帧。(读/写)

UHCI_HEAD_EN 置位此位，用格式报头编码数据包。(读/写)

UHCI_CRC_REC_EN 置位此位，使能 UHCI 接收 16 位 CRC。(读/写)

UHCI_UART_IDLE_EOF_EN 若此位置 1，UHCI 在 UART 空闲时停止接收有效载荷。(读/写)

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。(读/写)

见下页...

Register 8.31: UHCI_CONF0_REG (0x0000)

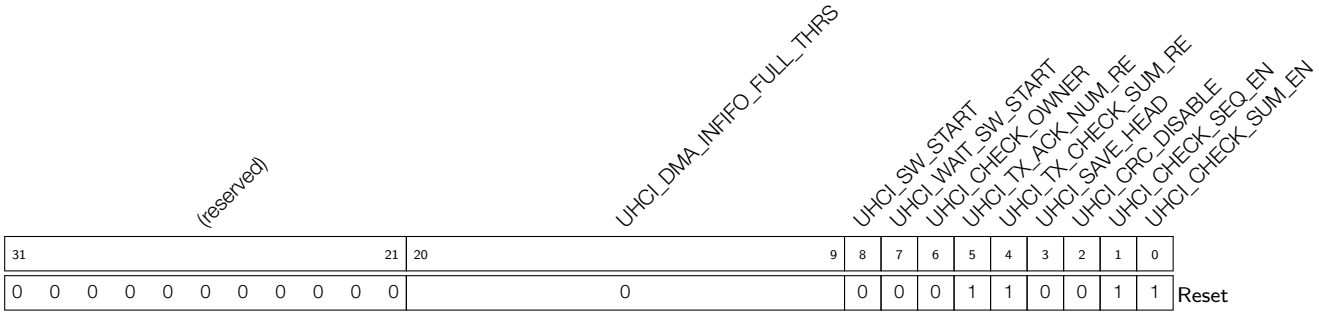
接上页...

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。(读/写)

UHCI_CLK_EN 1'b1：强制为寄存器开启时钟。1 ' b0：仅在应用写寄存器时支持时钟。(读/写)

UHCI_UART_RX_BRK_EOF_EN 若此位置 1，UART 收到 NULL 帧后 UHCI 结束 payload_rec。(读/写)

Register 8.32: UHCI_CONF1_REG (0x002C)



UHCI_CHECK_SUM_EN UHCI 接收数据包时检查报头校验和的使能位。(读/写)

UHCI_CHECK_SEQ_EN UHCI 接收数据包时检查序列号的使能位。(读/写)

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。(读/写)

UHCI_SAVE_HEAD 置位此位，在 HCI 接收数据包时保存数据包报头。(读/写)

UHCI_TX_CHECK_SUM_RE 置位此位，用校验和编码数据包。(读/写)

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。(读/写)

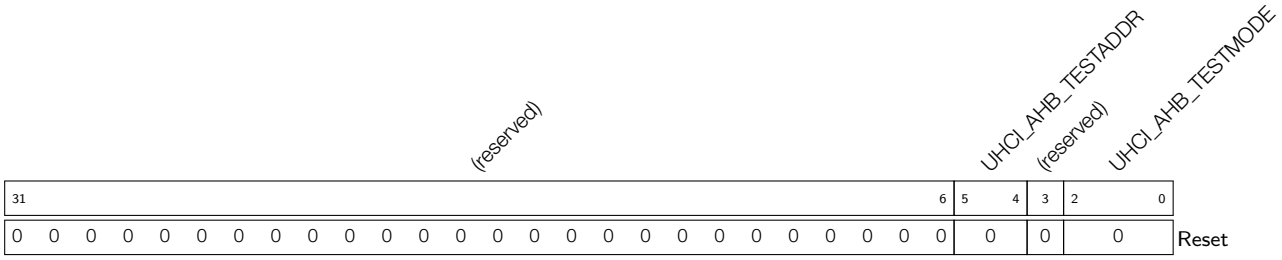
UHCI_CHECK_OWNER 1：链表操作方为 DMA 控制器时检查链表描述符；0：总是检查链表描述符。(读/写)

UHCI_WAIT_SW_START 此寄存器置 1 时，UHCI 编码器跳至 ST_SW_WAIT 状态。(读/写)

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。(读/写)

UHCI_DMA_INFIFO_FULL_THRS 接收 FIFO 的计数器值超过该寄存器的值时产生 UHCI_DMA_INFIFO_FULL_WM_INT 中断。(读/写)

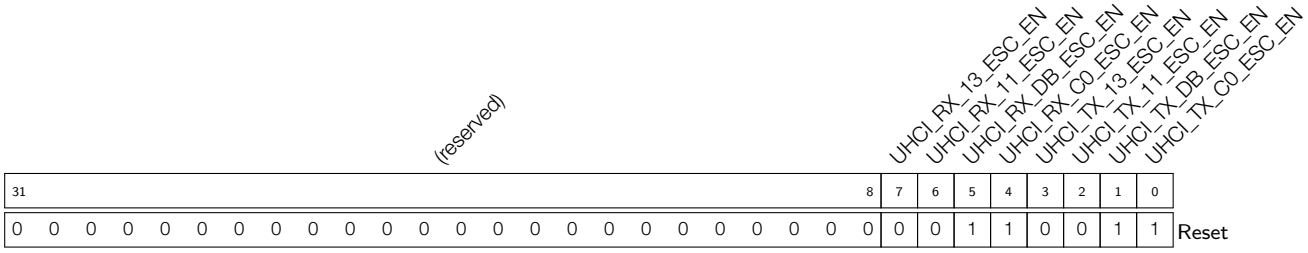
Register 8.33: UHCI_AHB_TEST_REG (0x0048)



UHCI_AHB_TESTMODE 保留。(读/写)

UHCI_AHB_TESTADDR 保留。(读/写)

Register 8.34: UHCI_ESCAPE_CONF_REG (0x0064)



UHCI_TX_C0_ESC_EN 置位此位，在 DMA 接收数据时解析字符 0xc0。(读/写)

UHCI_TX_DB_ESC_EN 置位此位，在 DMA 接收数据时解析字符 0xdb。(读/写)

UHCI_TX_11_ESC_EN 置位此位，在 DMA 接收数据时解析流控字符 0x11。(读/写)

UHCI_TX_13_ESC_EN 置位此位，在 DMA 接收数据时解析流控字符 0x13。(读/写)

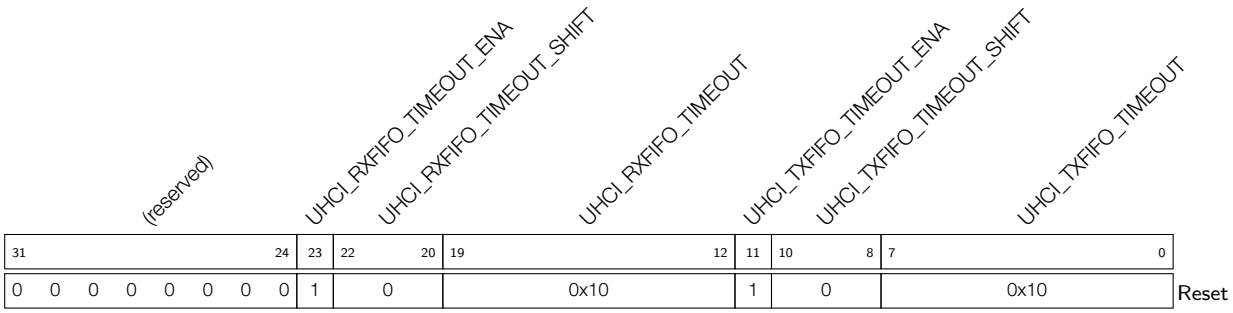
UHCI_RX_C0_ESC_EN 置位此位，在 DMA 发送数据时用特殊字符替换流控字符 0xc0。(读/写)

UHCI_RX_DB_ESC_EN 置位此位，在 DMA 发送数据时用特殊字符替换流控字符 0xdb。(读/写)

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。(读/写)

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。(读/写)

Register 8.35: UHCI_HUNG_CONF_REG (0x0068)



UHCI_TXFIFO_TIMEOUT 存储超时值。DMA 接收数据时间过长时产生 UHCI_TX_HUNG_INT 中断。(读/写)

UHCI_TXFIFO_TIMEOUT_SHIFT 配置计数器的最大值。(读/写)

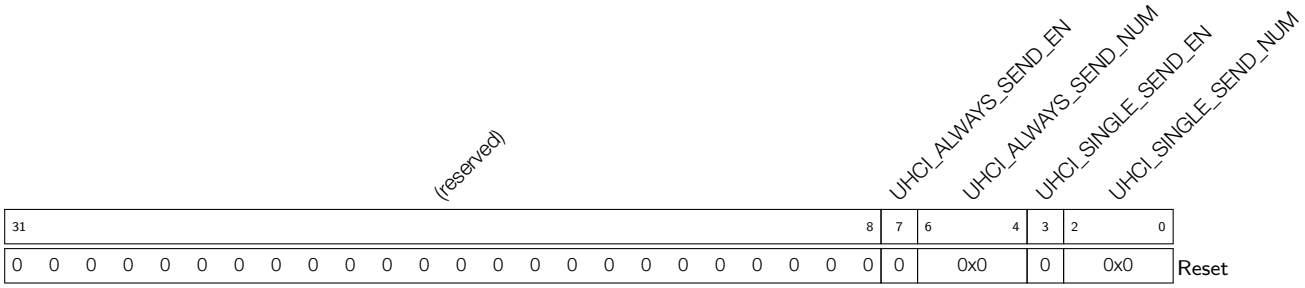
UHCI_TXFIFO_TIMEOUT_ENA TX FIFO 接收数据超时使能位。(读/写)

UHCI_RXFIFO_TIMEOUT 存储超时值。DMA 读取 RAM 数据时间过长时产生 UHCI_RX_HUNG_INT 中断。(读/写)

UHCI_RXFIFO_TIMEOUT_SHIFT 配置计数器的最大值。(读/写)

UHCI_RXFIFO_TIMEOUT_ENA DMA 发送数据超时的使能位。(读/写)

Register 8.36: UHCI_QUICK_SENT_REG (0x0074)



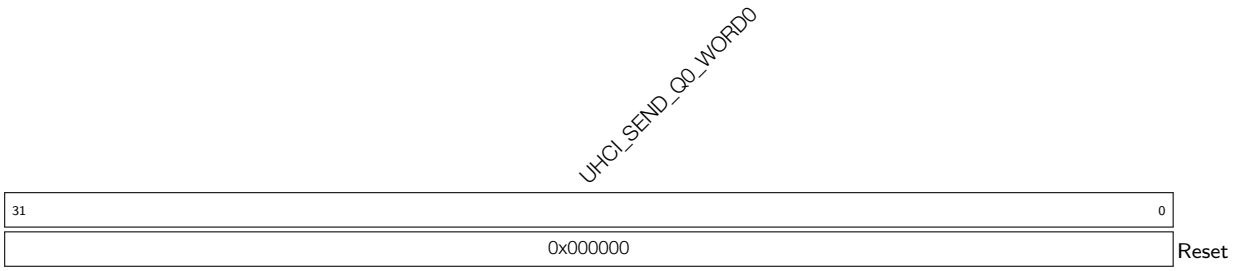
UHCI_SINGLE_SEND_NUM 设定 single_send 寄存器。(读/写)

UHCI_SINGLE_SEND_EN 置位此位使能 single_send 模式发送短包。(读/写)

UHCI_ALWAYS_SEND_NUM 设定 always_send 寄存器。(读/写)

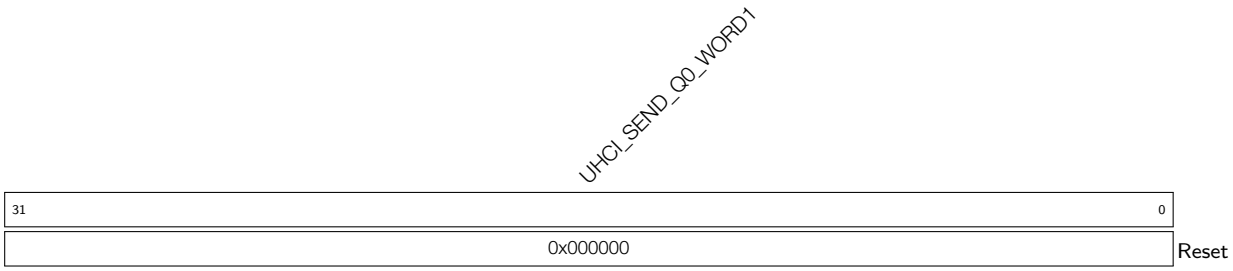
UHCI_ALWAYS_SEND_EN 置位此位使能 always_send 模式发送短包。(读/写)

Register 8.37: UHCI_REG_Q0_WORD0_REG (0x0078)



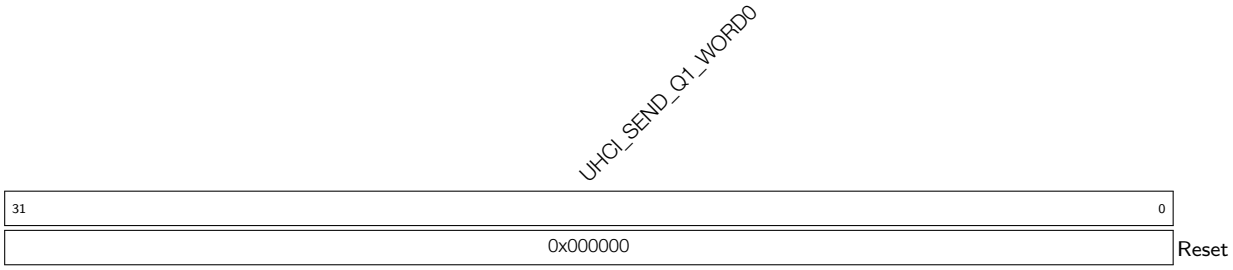
UHCI_SEND_Q0_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.38: UHCI_REG_Q0_WORD1_REG (0x007C)



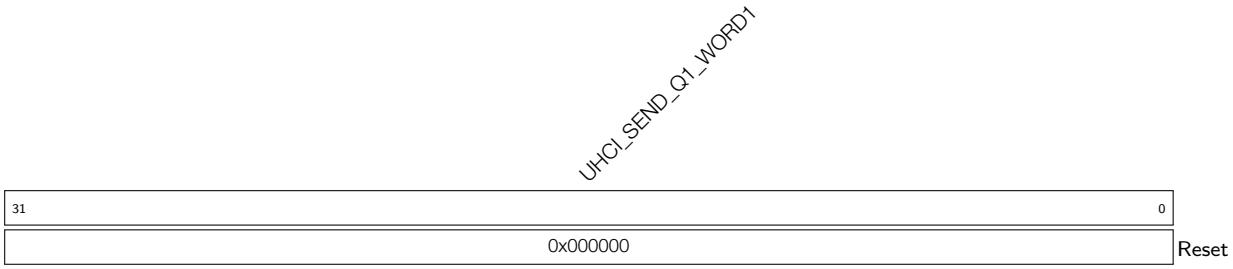
UHCI_SEND_Q0_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.39: UHCI_REG_Q1_WORD0_REG (0x0080)



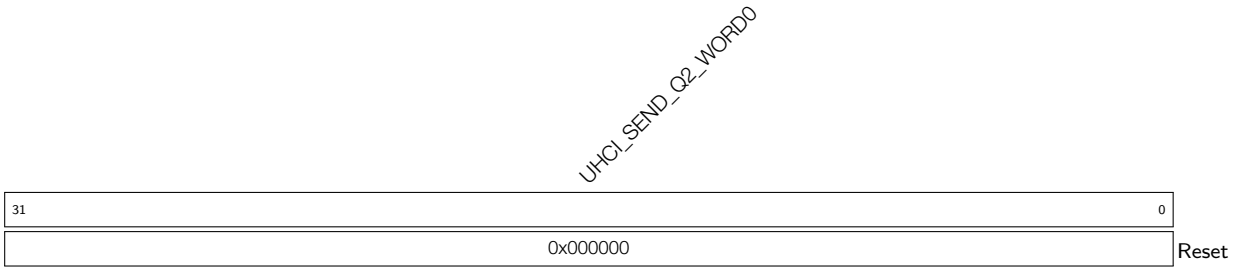
UHCI_SEND_Q1_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.40: UHCI_REG_Q1_WORD1_REG (0x0084)



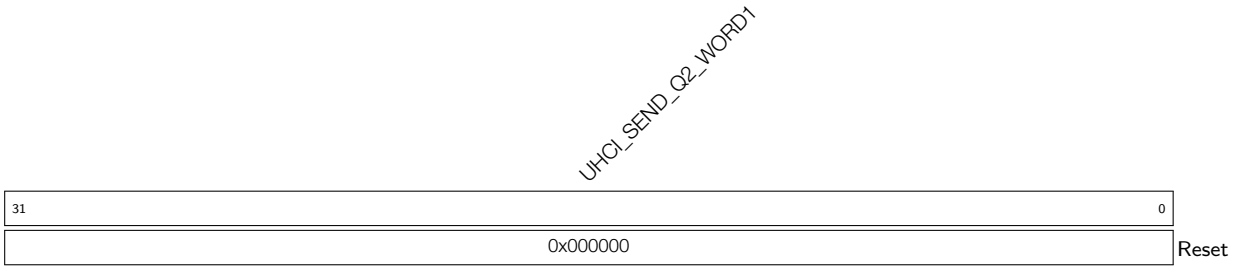
UHCI_SEND_Q1_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.41: UHCI_REG_Q2_WORD0_REG (0x0088)



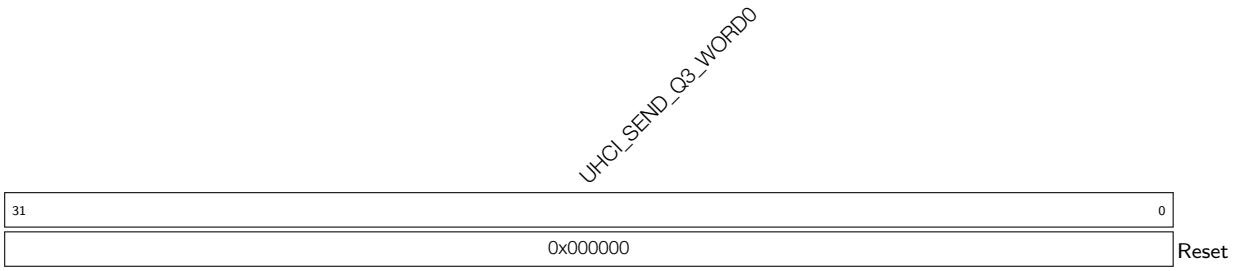
UHCI_SEND_Q2_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.42: UHCI_REG_Q2_WORD1_REG (0x008C)



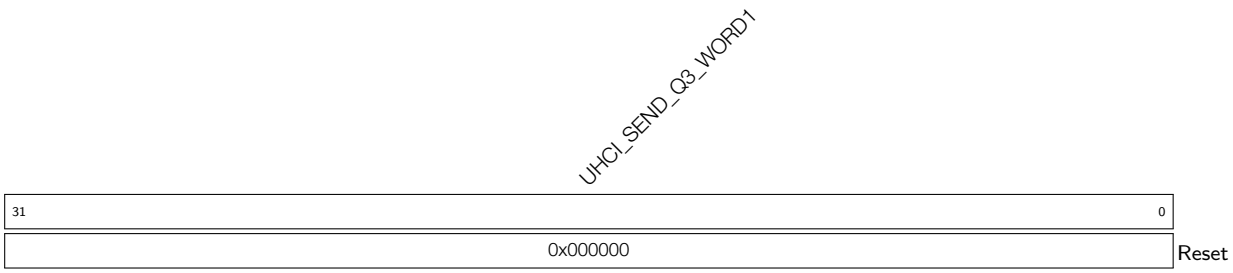
UHCI_SEND_Q2_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.43: UHCI_REG_Q3_WORD0_REG (0x0090)



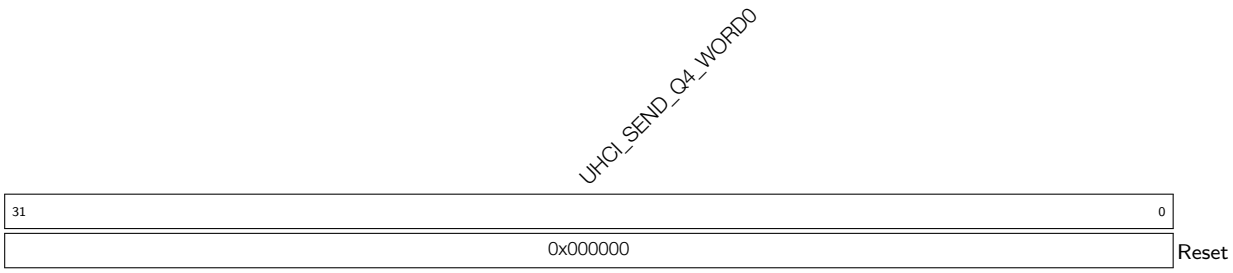
UHCI_SEND_Q3_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.44: UHCI_REG_Q3_WORD1_REG (0x0094)



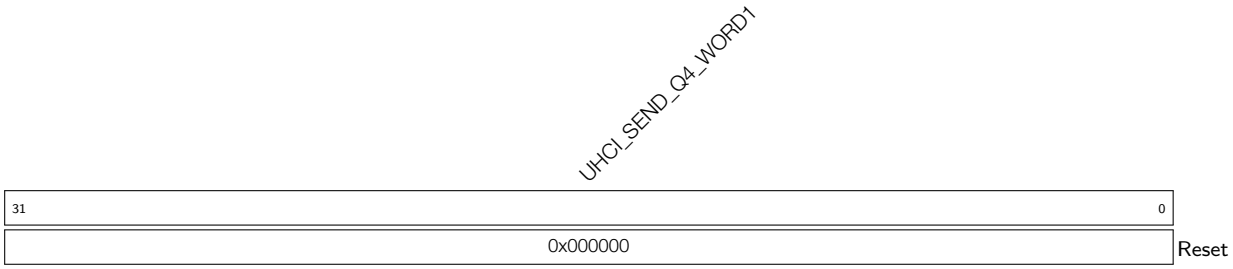
UHCI_SEND_Q3_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.45: UHCI_REG_Q4_WORD0_REG (0x0098)



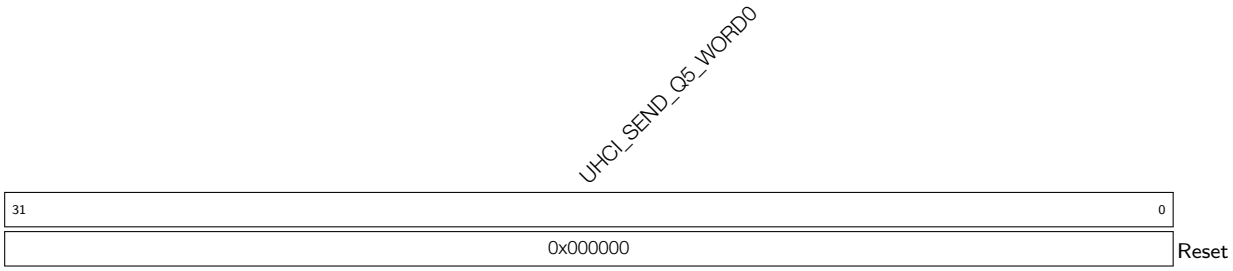
UHCI_SEND_Q4_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.46: UHCI_REG_Q4_WORD1_REG (0x009C)



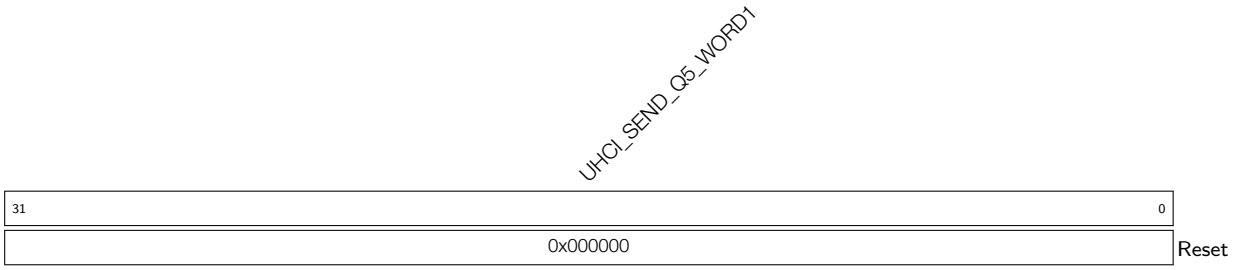
UHCI_SEND_Q4_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.47: UHCI_REG_Q5_WORD0_REG (0x00A0)



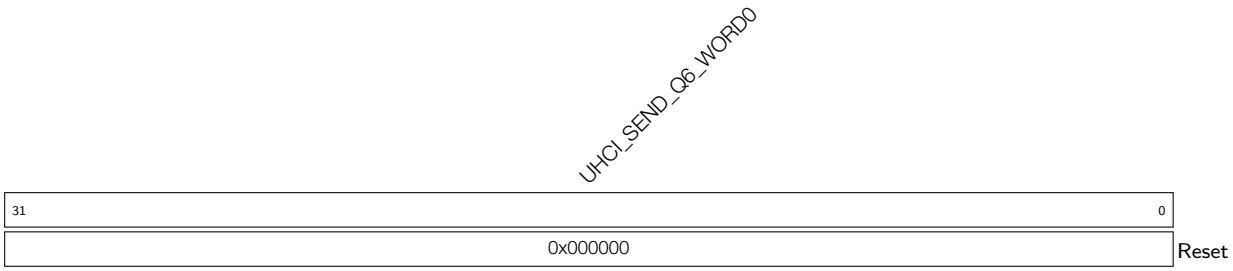
UHCI_SEND_Q5_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.48: UHCI_REG_Q5_WORD1_REG (0x00A4)



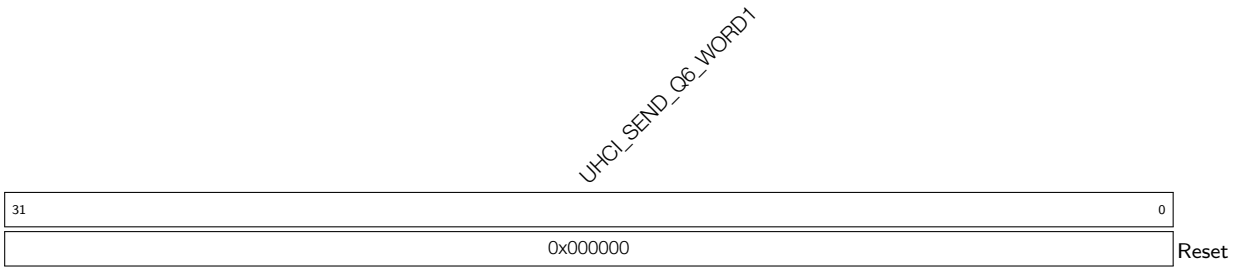
UHCI_SEND_Q5_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.49: UHCI_REG_Q6_WORD0_REG (0x00A8)



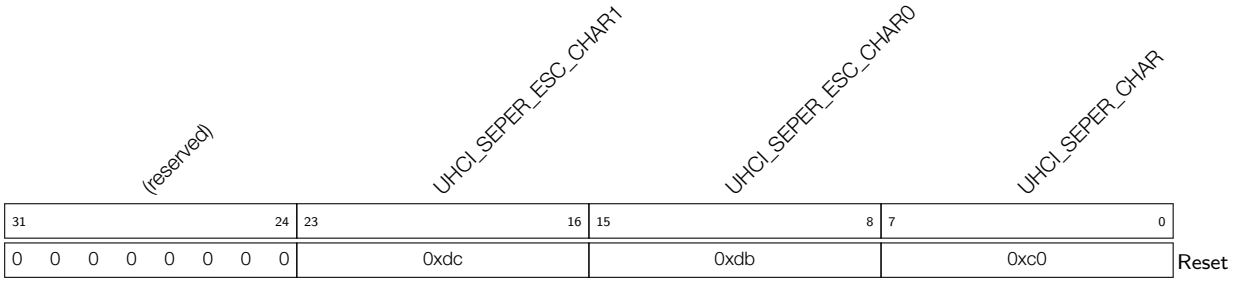
UHCI_SEND_Q6_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.50: UHCI_REG_Q6_WORD1_REG (0x00AC)



UHCI_SEND_Q6_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(读/写)

Register 8.51: UHCI_ESC_CONF0_REG (0x00B0)

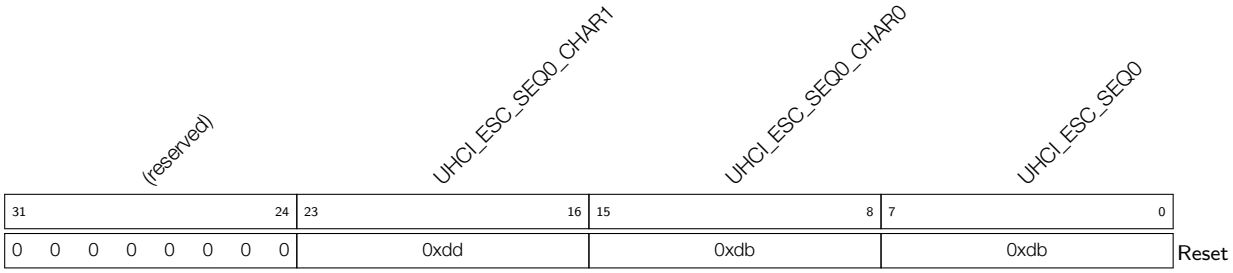


UHCI_SEPER_CHAR 定义需编码的分隔符，默认为 0xc0。(读/写)

UHCI_SEPER_ESC_CHAR0 编码分隔符时定义 Slip 转义序列的第一个字符，分隔符默认为 0xdb。(读/写)

UHCI_SEPER_ESC_CHAR1 编码分隔符时定义 Slip 转义序列的第二个字符，分隔符默认为 0xdc。(读/写)

Register 8.52: UHCI_ESC_CONF1_REG (0x00B4)

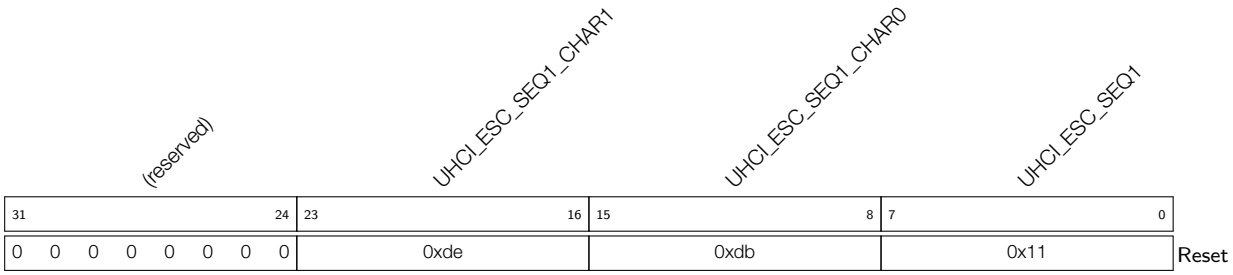


UHCI_ESC_SEQ0 定义需编码的字符，默认为用作 Slip 转义序列第一个字符的 0xdb。(读/写)

UHCI_ESC_SEQ0_CHAR0 编码 UHCI_ESC_SEQ0 时定义 Slip 转义序列的第一个字符，UHCI_ESC_SEQ0 默认为 0xdb。(读/写)

UHCI_ESC_SEQ0_CHAR1 编码 UHCI_ESC_SEQ0 时定义 Slip 转义序列的第二个字符，UHCI_ESC_SEQ0 默认为 0xdd。(读/写)

Register 8.53: UHCI_ESC_CONF2_REG (0x00B8)

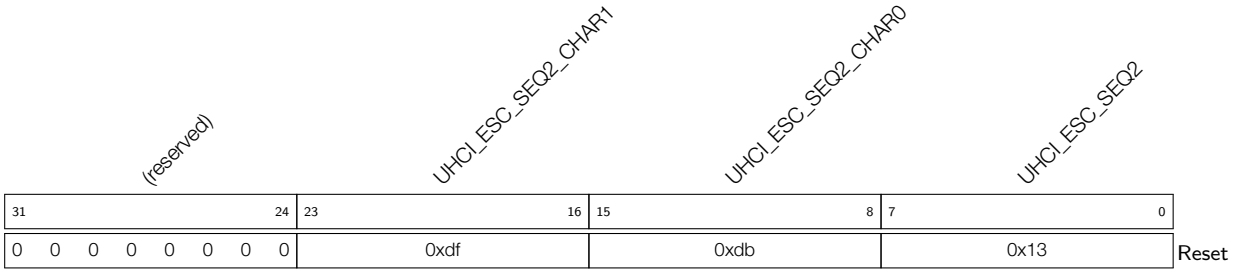


UHCI_ESC_SEQ1 定义需编码的字符，默认为用作流控字符的 0x11。(读/写)

UHCI_ESC_SEQ1_CHAR0 编码 UHCI_ESC_SEQ1 时定义 Slip 转义序列的第一个字符，UHCI_ESC_SEQ1 默认为 0xdb。(读/写)

UHCI_ESC_SEQ1_CHAR1 编码 UHCI_ESC_SEQ1 时定义 Slip 转义序列的第二个字符，UHCI_ESC_SEQ1 默认为 0xde。(读/写)

Register 8.54: UHCI_ESC_CONF3_REG (0x00BC)

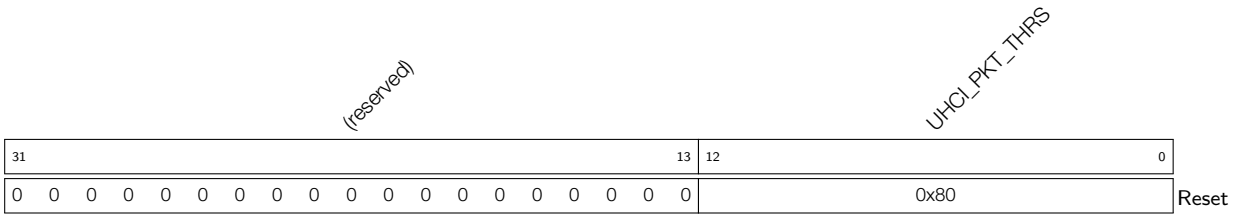


UHCI_ESC_SEQ2 定义需编码的字符，默认为用作流控字符的 0x13。(读/写)

UHCI_ESC_SEQ2_CHAR0 编码 UHCI_ESC_SEQ2 时定义 Slip 转义序列的第一个字符，UHCI_ESC_SEQ2 默认为 0xdb。(读/写)

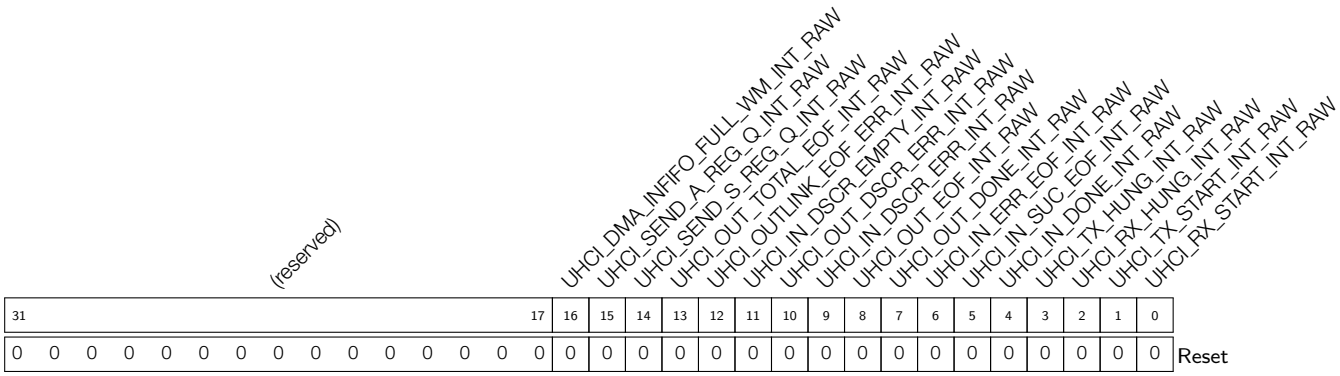
UHCI_ESC_SEQ2_CHAR1 编码 UHCI_ESC_SEQ2 时定义 Slip 转义序列的第二个字符，UHCI_ESC_SEQ2 默认为 0xdf。(读/写)

Register 8.55: UHCI_PKT_THRES_REG (0x00C0)



UHCI_PKT_THRS UHCI_HEAD_EN 为 0 时配置包长度的最大值。(读/写)

Register 8.56: UHCI_INT_RAW_REG (0x0004)



UHCI_RX_START_INT_RAW 原始中断位。分隔符成功发送时触发此中断。(只读)

UHCI_TX_START_INT_RAW 原始中断位。DMA 检测到分隔符时触发此中断。(只读)

UHCI_RX_HUNG_INT_RAW 原始中断位。DMA 接收数据所需时间超过配置值时触发此中断。(只读)

UHCI_TX_HUNG_INT_RAW 原始中断位。DMA 读取 RAM 数据所需时间超过配置值时触发此中断。(只读)

UHCI_IN_DONE_INT_RAW 原始中断位。接收链表描述符完成时触发此中断。(只读)

UHCI_IN_SUC_EOF_INT_RAW 原始中断位。成功接收数据包时触发此中断。(只读)

UHCI_IN_ERR_EOF_INT_RAW 原始中断位。接收链表描述符的 EOF 有错误时触发此中断。(只读)

UHCI_OUT_DONE_INT_RAW 原始中断位。发送链表描述符完成时触发此中断。(只读)

UHCI_OUT_EOF_INT_RAW 原始中断位。当前描述符的 EOF 有效时触发此中断。(只读)

UHCI_IN_DSCR_ERR_INT_RAW 原始中断位。接收链表描述符存在错误时触发此中断。(只读)

UHCI_OUT_DSCR_ERR_INT_RAW 原始中断位。发送链表描述符存在错误时触发此中断。(只读)

UHCI_IN_DSCR_EMPTY_INT_RAW 原始中断位。DMA 没有足够的接收链表时触发此中断。(只读)

UHCI_OUTLINK_EOF_ERR_INT_RAW 原始中断位。发送链表描述符的 EOF 有错误时触发此中断。(只读)

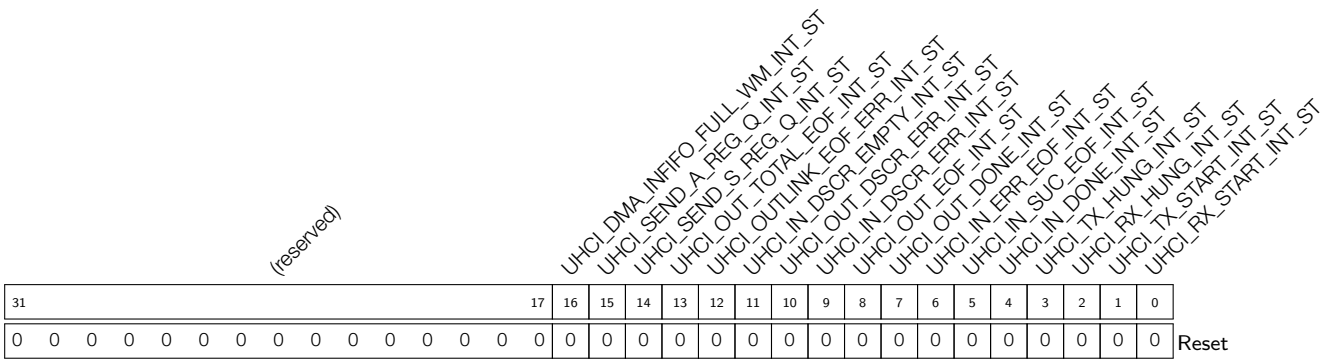
UHCI_OUT_TOTAL_EOF_INT_RAW 原始中断位。最后一个缓存区中的所有数据发送完毕时触发此中断。(只读)

UHCI_SEND_S_REG_Q_INT_RAW 原始中断位。DMA 使用 single_send 寄存器成功发送短包时触发此中断。(只读)

UHCI_SEND_A_REG_Q_INT_RAW 原始中断位。DMA 使用 always_send 寄存器成功发送短包时触发此中断。(只读)

UHCI_DMA_INFIFO_FULL_WM_INT_RAW 原始中断位。DMA INFIFO 存储量达到配置阈值时触发此中断。(只读)

Register 8.57: 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。(只读)

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。(只读)

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。(只读)

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。(只读)

UHCI_IN_DONE_INT_ST UHCI_IN_DONE_INT_ENA 置 1 时 UHCI_IN_DONE_INT 中断的屏蔽中断位。(只读)

UHCI_IN_SUC_EOF_INT_ST UHCI_IN_SUC_EOF_INT_ENA 置 1 时 UHCI_IN_SUC_EOF_INT 中断的屏蔽中断位。(只读)

UHCI_IN_ERR_EOF_INT_ST UHCI_IN_ERR_EOF_INT_ENA 置 1 时 UHCI_IN_ERR_EOF_INT 中断的屏蔽中断位。(只读)

UHCI_OUT_DONE_INT_ST UHCI_OUT_DONE_INT_ENA 置 1 时 UHCI_OUT_DONE_INT 中断的屏蔽中断位。(只读)

UHCI_OUT_EOF_INT_ST UHCI_OUT_EOF_INT_ENA 置 1 时 UHCI_OUT_EOF_INT 中断的屏蔽中断位。(只读)

UHCI_IN_DSCR_ERR_INT_ST UHCI_IN_DSCR_ERR_INT_ENA 置 1 时 UHCI_IN_DSCR_ERR_INT 中断的屏蔽中断位。(只读)

UHCI_OUT_DSCR_ERR_INT_ST UHCI_OUT_DSCR_ERR_INT_ENA 置 1 时 UHCI_OUT_DSCR_ERR_INT 中断的屏蔽中断位。(只读)

UHCI_IN_DSCR_EMPTY_INT_ST UHCI_IN_DSCR_EMPTY_INT_ENA 置 1 时 UHCI_IN_DSCR_EMPTY_INT 中断的屏蔽中断位。(只读)

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。(只读)

UHCI_OUT_TOTAL_EOF_INT_ST UHCI_OUT_TOTAL_EOF_INT_ENA 置 1 时 UHCI_OUT_TOTAL_EOF_INT 中断的屏蔽中断位。(只读)

见下页...

Register 8.57: UHCI_INT_ST_REG (0x0008)

接上页...

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。(只读)

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。(只读)

UHCI_DMA_INFIFO_FULL_WM_INT_ST UHCI_DMA_INFIFO_FULL_WM_INT_ENA 置 1 时 UHCI_DMA_INFIFO_FULL_WM_INT 中断的屏蔽中断位。(只读)

Register 8.58: UHCI_INT_ENA_REG (0x000C)

UHCI_RX_START_INT_ENA UHCI_RX_START_INT 的中断使能位。(读/写)

UHCI_TX_START_INT_ENA UHCI_TX_START_INT 的中断使能位。(读/写)

UHCI_RX_HUNG_INT_ENA UHCI_RX_HUNG_INT 的中断使能位。(读/写)

UHCI_TX_HUNG_INT_ENA UHCI_TX_HUNG_INT 的中断使能位。(读/写)

UHCI_IN_DONE_INT_ENA UHCI_IN_DONE_INT 的中断使能位。(读/写)

UHCI_IN_SUC_EOF_INT_ENA UHCI_IN_SUC_EOF_INT 的中断使能位。(读/写)

UHCI_IN_ERR_EOF_INT_ENA UHCI_IN_ERR_EOF_INT 的中断使能位。(读/写)

UHCI_OUT_DONE_INT_ENA UHCI_OUT_DONE_INT 的中断使能位。(读/写)

UHCI_OUT_EOF_INT_ENA UHCI_OUT_EOF_INT 的中断使能位。(读/写)

UHCI_IN_DSCR_ERR_INT_ENA UHCI_IN_DSCR_ERR_INT 的中断使能位。(读/写)

UHCI_OUT_DSCR_ERR_INT_ENA UHCI_OUT_DSCR_ERR_INT 的中断使能位。(读/写)

UHCI_IN_DSCR_EMPTY_INT_ENA UHCI_IN_DSCR_EMPTY_INT 的中断使能位。(读/写)

UHCI_OUTLINK_EOF_ERR_INT_ENA UHCI_OUTLINK_EOF_ERR_INT 的中断使能位。(读/写)

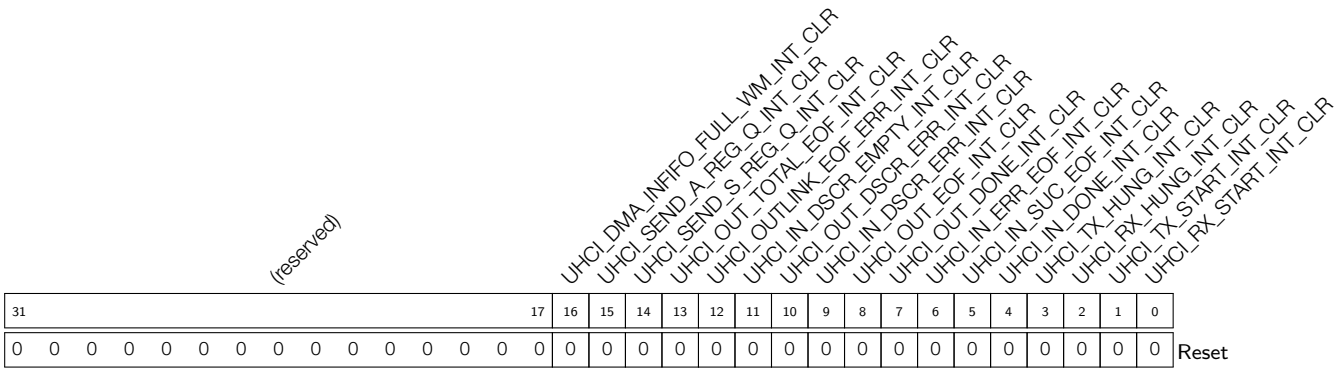
UHCI_OUT_TOTAL_EOF_INT_ENA UHCI_OUT_TOTAL_EOF_INT 的中断使能位。(读/写)

UHCI_SEND_S_REG_Q_INT_ENA UHCI_SEND_S_REG_Q_INT 的中断使能位。(读/写)

UHCI_SEND_A_REG_Q_INT_ENA UHCI_SEND_A_REG_Q_INT 的中断使能位。(读/写)

UHCI_DMA_INFIFO_FULL_WM_INT_ENA UHCI_DMA_INFIFO_FULL_WM_INT 的中断使能位。(读/写)

Register 8.59: UHCI_INT_CLR_REG (0x0010)



UHCI_RX_START_INT_CLR 置位此位清除 UHCI_RX_START_INT 中断。(只写)

UHCI_TX_START_INT_CLR 置位此位清除 UHCI_TX_START_INT 中断。(只写)

UHCI_RX_HUNG_INT_CLR 置位此位清除 UHCI_RX_HUNG_INT 中断。(只写)

UHCI_TX_HUNG_INT_CLR 置位此位清除 UHCI_TX_HUNG_INT 中断。(只写)

UHCI_IN_DONE_INT_CLR 置位此位清除 UHCI_IN_DONE_INT 中断。(只写)

UHCI_IN_SUC_EOF_INT_CLR 置位此位清除 UHCI_IN_SUC_EOF_INT 中断。(只写)

UHCI_IN_ERR_EOF_INT_CLR 置位此位清除 UHCI_IN_ERR_EOF_INT 中断。(只写)

UHCI_OUT_DONE_INT_CLR 置位此位清除 UHCI_OUT_DONE_INT 中断。(只写)

UHCI_OUT_EOF_INT_CLR 置位此位清除 UHCI_OUT_EOF_INT 中断。(只写)

UHCI_IN_DSCR_ERR_INT_CLR 置位此位清除 UHCI_IN_DSCR_ERR_INT 中断。(只写)

UHCI_OUT_DSCR_ERR_INT_CLR 置位此位清除 UHCI_OUT_DSCR_ERR_INT 中断。(只写)

UHCI_IN_DSCR_EMPTY_INT_CLR 置位此位清除 UHCI_IN_DSCR_EMPTY_INT 中断。(只写)

UHCI_OUTLINK_EOF_ERR_INT_CLR 置位此位清除 UHCI_OUTLINK_EOF_ERR_INT 中断。(只写)

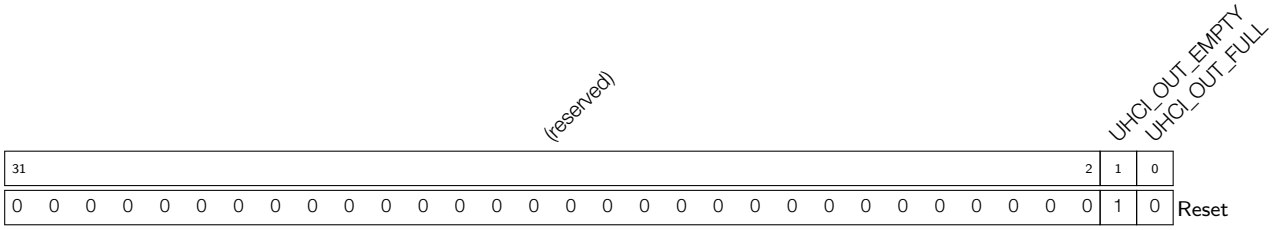
UHCI_OUT_TOTAL_EOF_INT_CLR 置位此位清除 UHCI_OUT_TOTAL_EOF_INT 中断。(只写)

UHCI_SEND_S_REG_Q_INT_CLR 置位此位清除 UHCI_SEND_S_UHCI_Q_INT 中断。(只写)

UHCI_SEND_A_REG_Q_INT_CLR 置位此位清除 UHCI_SEND_A_UHCI_Q_INT 中断。(只写)

UHCI_DMA_INFIFO_FULL_WM_INT_CLR 置位此位清除 UHCI_DMA_INFIFO_FULL_WM_INT 中断。(只写)

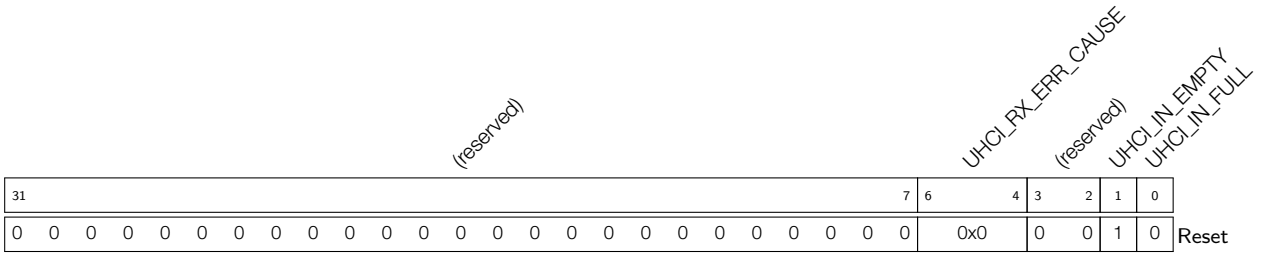
Register 8.60: UHCI_DMA_OUT_STATUS_REG (0x0014)



UHCI_OUT_FULL 1：DMA 数据输出 FIFO 为满。(只读)

UHCI_OUT_EMPTY 1：DMA 数据输出 FIFO 为空。(只读)

Register 8.61: UHCI_DMA_IN_STATUS_REG (0x001C)

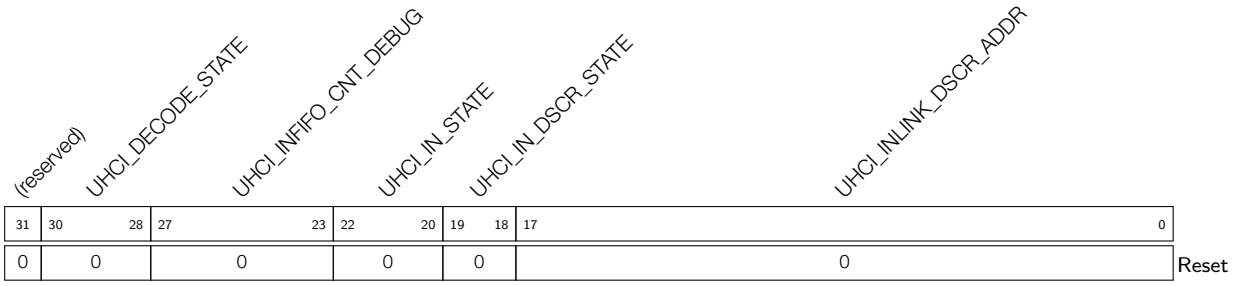


UHCI_IN_FULL 数据输入 FIFO 满信号。(只读)

UHCI_IN_EMPTY 数据输入 FIFO 空信号。(只读)

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。(只读)

Register 8.62: UHCI_STATE0_REG (0x0030)



UHCI_INLINK_DSCR_ADDR 存储当前接收链表描述符的地址。(只读)

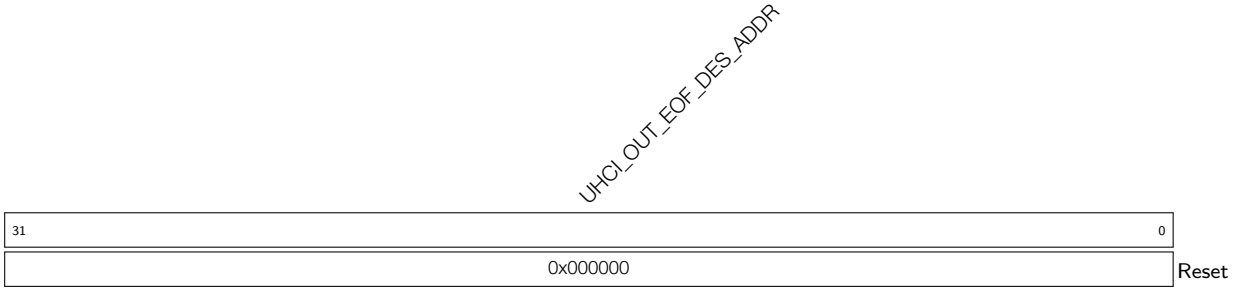
UHCI_IN_DSCR_STATE 保留。(只读)

UHCI_IN_STATE 保留。(只读)

UHCI_INFIFO_CNT_DEBUG 存储接收链表描述符 FIFO 中的数据字节数。(只读)

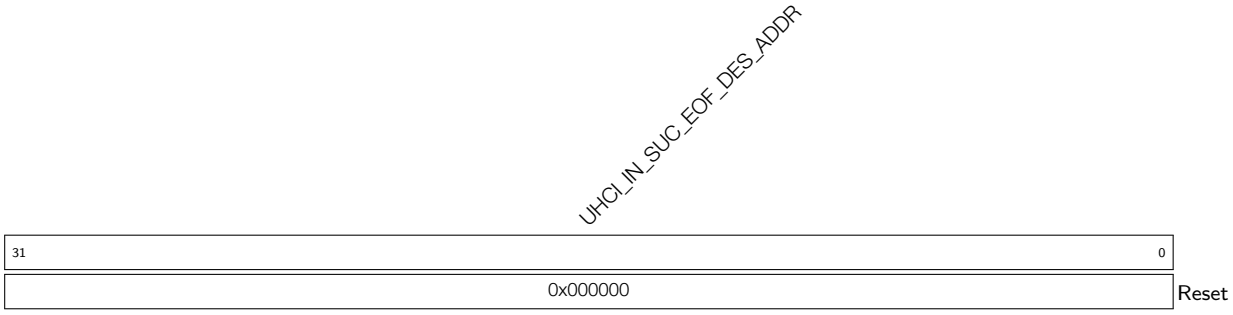
UHCI_DECODE_STATE UHCI 解码器状态。(只读)

Register 8.63: UHCI_DMA_OUT_EOF_DES_ADDR_REG (0x0038)



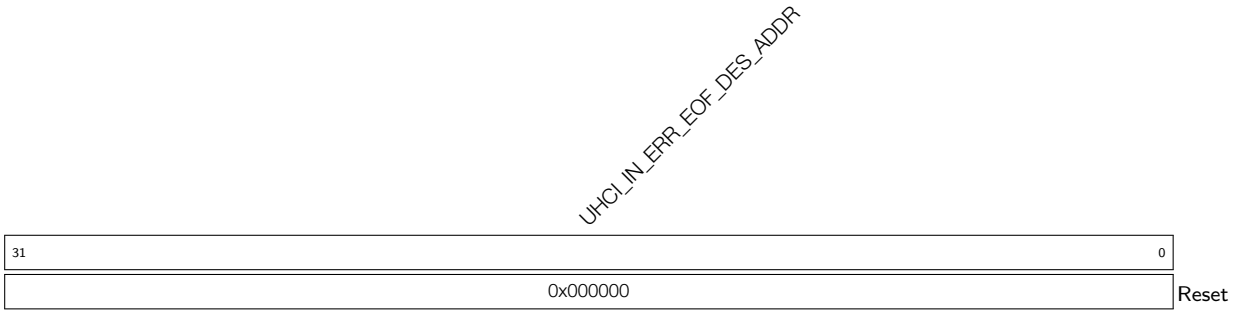
UHCI_OUT_EOF_DES_ADDR 存储发送链表描述符的 EOF 位为 1 时该描述符的地址。(只读)

Register 8.64: UHCI_DMA_IN_SUC_EOF_DES_ADDR_REG (0x003C)



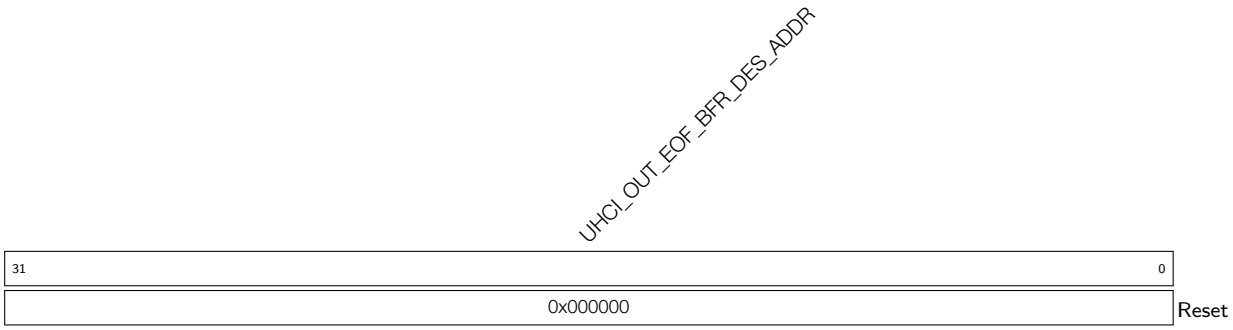
UHCI_IN_SUC_EOF_DES_ADDR 存储成功接收 EOF 时接收链表描述符的地址。(只读)

Register 8.65: UHCI_DMA_IN_ERR_EOF_DES_ADDR_REG (0x0040)



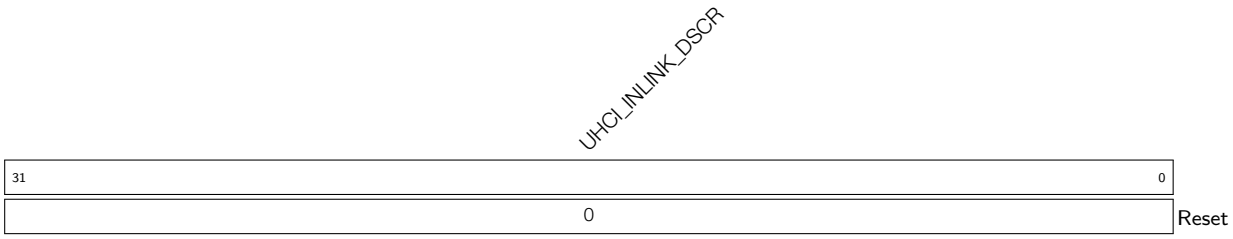
UHCI_IN_ERR_EOF_DES_ADDR 存储接收链表描述符存在错误时该描述符的地址。(只读)

Register 8.66: UHCI_DMA_OUT_EOF_BFR_DES_ADDR_REG (0x0044)



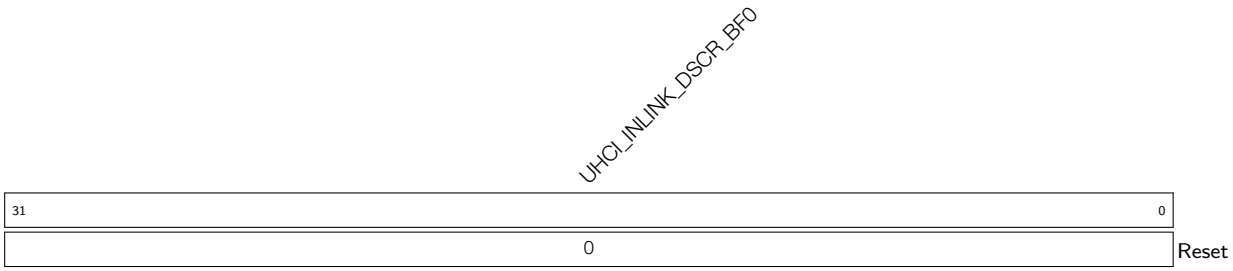
UHCI_OUT_EOF_BFR_DES_ADDR 存储上一个发送链表描述符之前的描述符地址。(只读)

Register 8.67: UHCI_DMA_IN_DSCR_REG (0x004C)



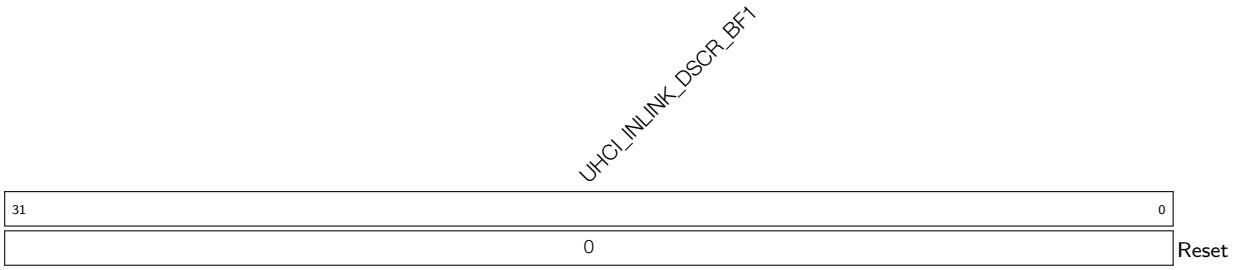
UHCI_INLINK_DSCR 存储下一个接收链表描述符的第三个字。(只读)

Register 8.68: UHCI_DMA_IN_DSCR_BF0_REG (0x0050)



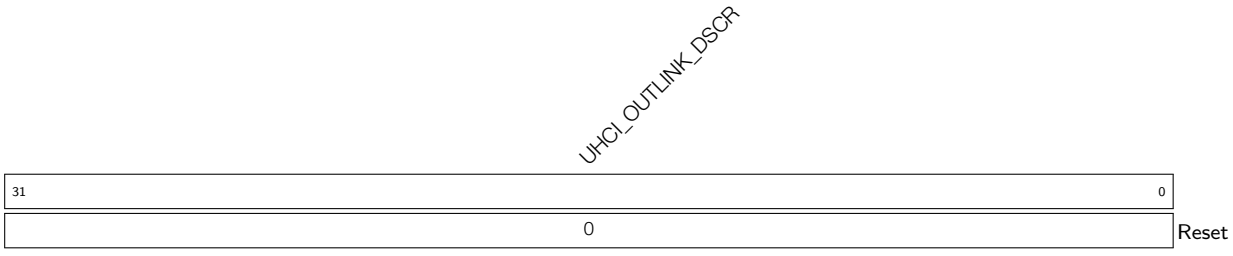
UHCI_INLINK_DSCR_BF0 存储当前接收链表描述符的第三个字。(只读)

Register 8.69: UHCI_DMA_IN_DSCR_BF1_REG (0x0054)



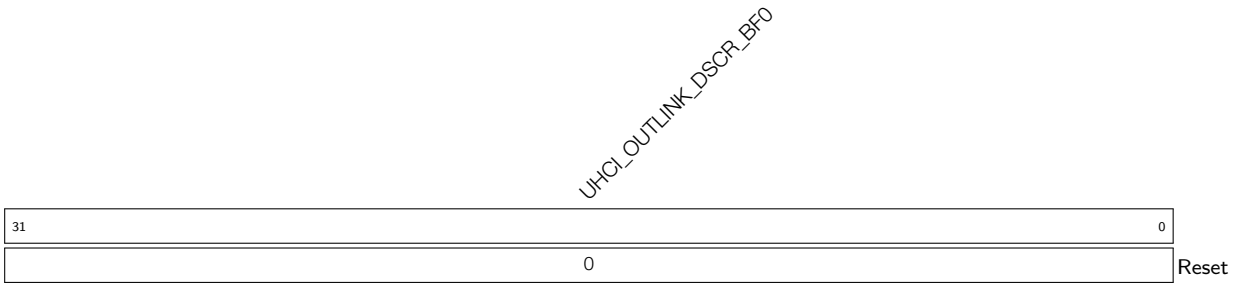
UHCI_INLINK_DSCR_BF1 存储当前接收链表描述符的第二个字。(只读)

Register 8.70: UHCI_DMA_OUT_DSCR_REG (0x0058)



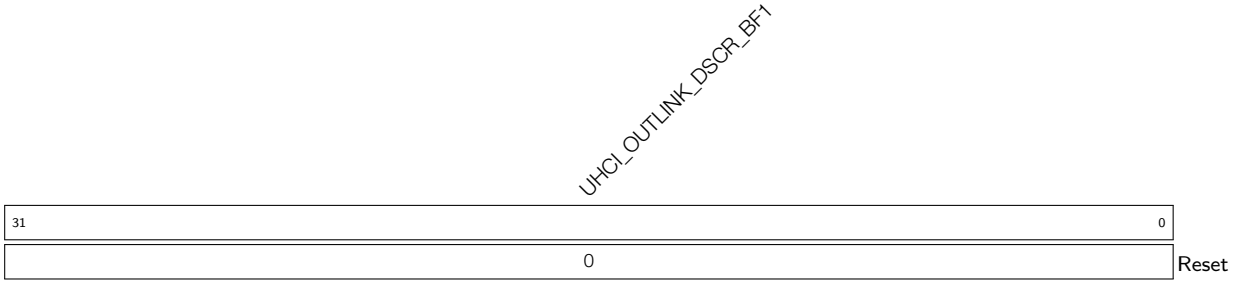
UHCI_OUTLINK_DSCR 存储下一个发送链表描述符的第三个字。(只读)

Register 8.71: UHCI_DMA_OUT_DSCR_BF0_REG (0x005C)



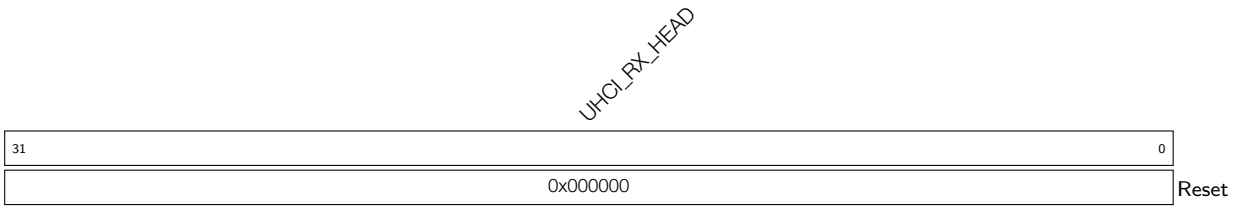
UHCI_OUTLINK_DSCR_BF0 存储当前发送链表描述符的第三个字。(只读)

Register 8.72: UHCI_DMA_OUT_DSCR_BF1_REG (0x0060)



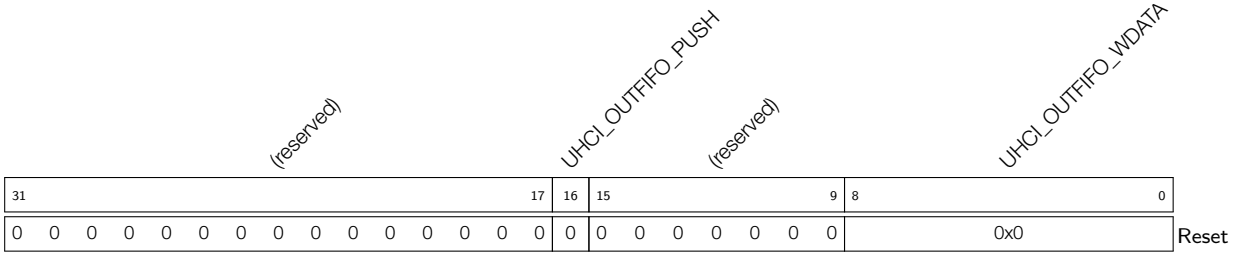
UHCI_OUTLINK_DSCR_BF1 存储当前发送链表描述符的第二个字。(只读)

Register 8.73: UHCI_RX_HEAD_REG (0x0070)



UHCI_RX_HEAD 存储当前接收包的报头 (只读)

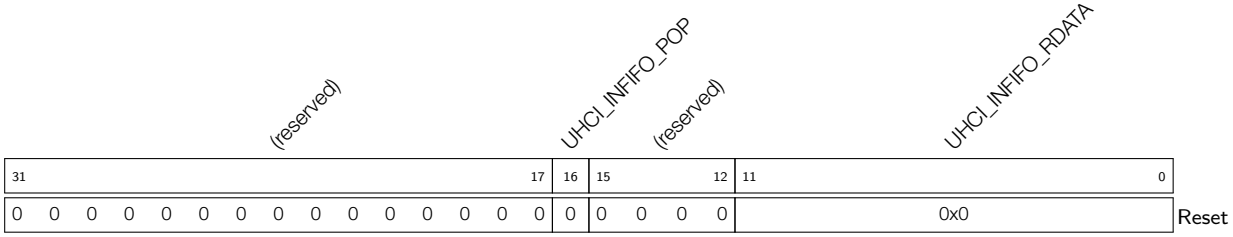
Register 8.74: UHCI_DMA_OUT_PUSH_REG (0x0018)



UHCI_OUTFIFO_WDATA 需推送到数据输出 FIFO 的数据。(读/写)

UHCI_OUTFIFO_PUSH 置位此位将数据推送到数据输出 FIFO。(读/写)

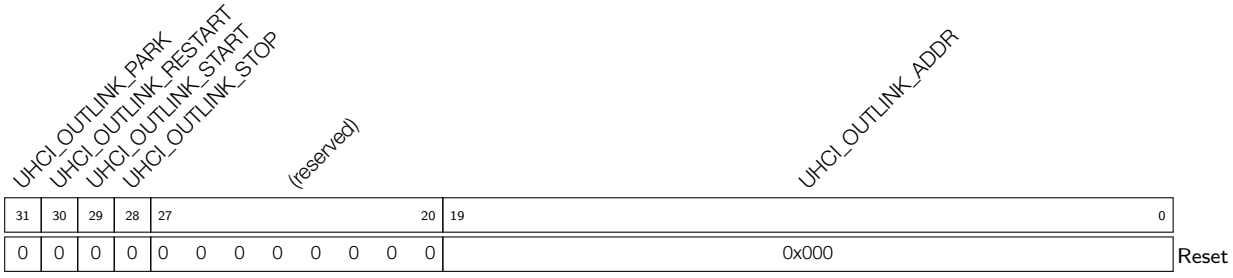
Register 8.75: UHCI_DMA_IN_POP_REG (0x0020)



UHCI_INFIFO_RDATA 存储从数据输入 FIFO 中弹出的数据。(只读)

UHCI_INFIFO_POP 置位此位，从数据输入 FIFO 中弹出数据。(读/写)

Register 8.76: UHCI_DMA_OUT_LINK_REG (0x0024)



UHCI_OUTLINK_ADDR 用于设定第一个发送链表描述符地址的低 20 位。(读/写)

UHCI_OUTLINK_STOP 置位此位，停止处理发送链表描述符。(读/写)

UHCI_OUTLINK_START 置位此位，启动新的发送链表描述符。(读/写)

UHCI_OUTLINK_RESTART 置位此位，在上一个地址重启发送链表描述符。(读/写)

UHCI_OUTLINK_PARK 1：发送链表描述符的 FSM 空闲。0：发送链表描述符的 FSM 正在工作。(只读)

Register 8.77: UHCI_DMA_IN_LINK_REG (0x0028)

UHCI_INLINK_ADDR 用于设定第一个接收链表描述符地址的低 20 位。(读/写)

UHCI_INLINK_AUTO_RET 当前包存在错误时恢复当前接收链表描述符地址的使能位。(读/写)

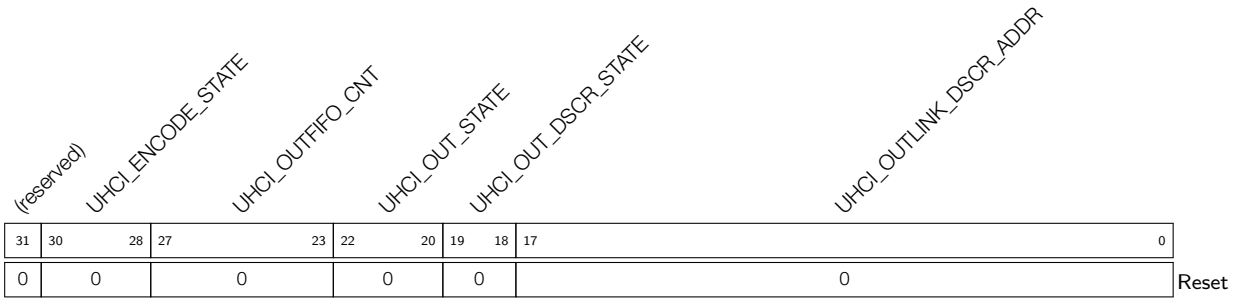
UHCI_INLINK_STOP 置位此位，停止处理接收链表描述符。(读/写)

UHCI_INLINK_START 置位此位，开始处理接收链表描述符。(读/写)

UHCI_INLINK_RESTART 置位此位，启动新的接收链表描述符。(读/写)

UHCI_INLINK_PARK 1：接收链表描述符的 FSM 空闲。0：接收链表描述符的 FSM 正在工作。(只读)

Register 8.78: UHCI_STATE1_REG (0x0034)



UHCI_OUTLINK_DSCR_ADDR 存储当前发送链表描述符的地址。(只读)

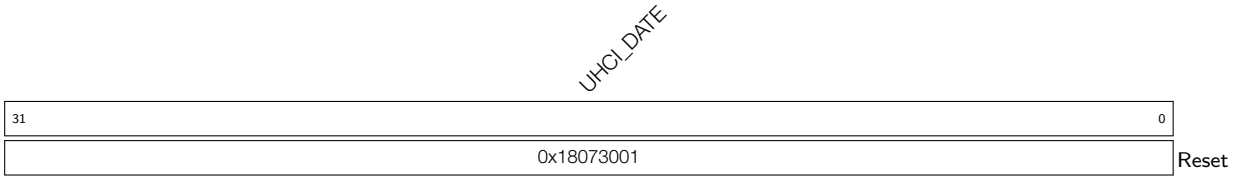
UHCI_OUT_DSCR_STATE 保留。(只读)

UHCI_OUT_STATE 保留。(只读)

UHCI_OUTFIFO_CNT 存储发送链表描述符 FIFO 中的数据字节数。(只读)

UHCI_ENCODE_STATE UHCI 编码器状态。(只读)

Register 8.79: UHCI_DATE_REG (0x00FC)



UHCI_DATE 版本控制寄存器。(读/写)

9. LED PWM 控制器

9.1 概述

LED PWM 控制器主要用于控制 LED 设备，也可以产生 PWM 信号用于控制其他开关设备。该控制器由 14 位定时器和波形发生器组成。

9.2 特性

LED PWM 控制器具有如下特性:

- 四个独立定时器，可实现小数分频
- 八个独立波形发生器，可产生 8 路 PWM 信号
- PWM 信号占空比可递增或递减渐变，无须处理器干预。渐变完成可产生中断事件
- 输出 PWM 信号相位可调
- 低功耗模式下可输出 PWM 信号

为方便描述，下文中八个 PWM 发生器统称为 PWM*n*，四个定时器统称为 Timer*x*。

9.3 功能描述

9.3.1 架构

图 9-1为 LED PWM 控制器的架构。图 9-2 为一个 PWM 生成器及其选取的定时器和计数器。

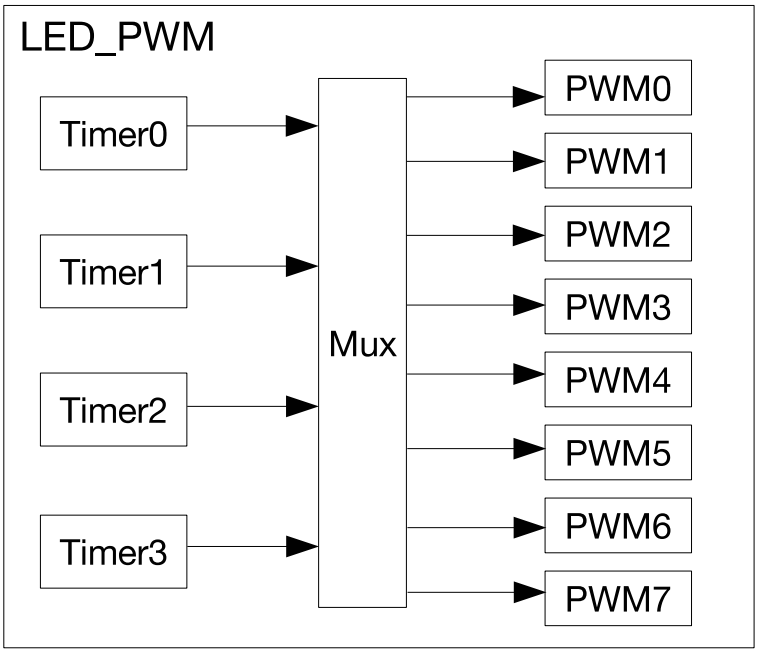


图 9-1. LED_PWM 架构

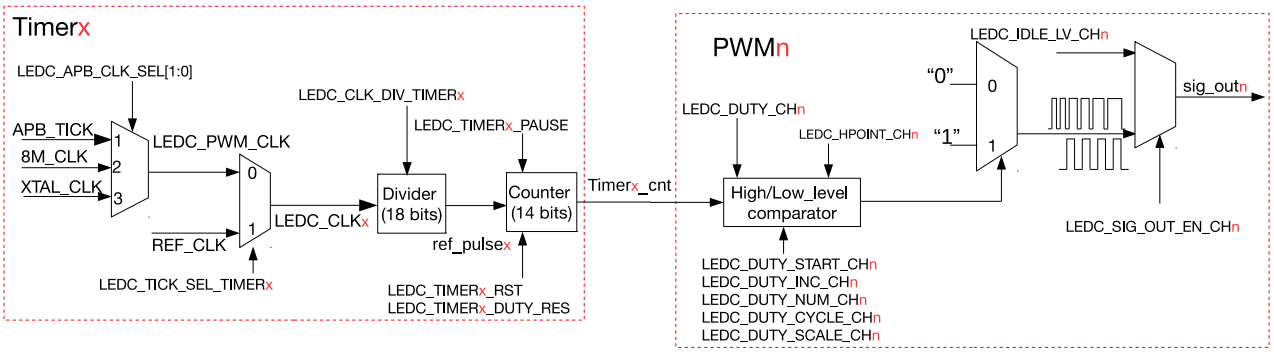


图 9-2. LED_PWM 生成器图

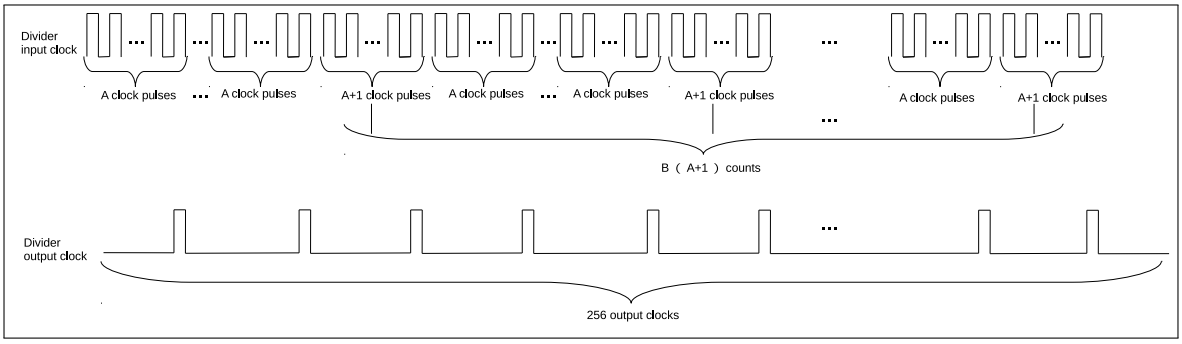


图 9-3. LED_PWM 分频器

9.3.2 定时器

LED PWM 控制器的时钟 LEDC_PWM_CLK 有三个时钟源：APB_CLK、RTC8M_CLK 和 XTAL_CLK，可通过配置 LEDC_APB_CLK_SEL[1:0] 来选择。LED PWM 定时器的时钟 LEDC_CLK*x* 有两个时钟源：LEDC_PWM_CLK 和 REF_TICK。如使用 REF_TICK 作定时器的时钟源，需将 LEDC_APB_CLK_SEL[1:0] 置 1，REF_TICK 的周期需为 APB_CLK 周期的整数倍，否则定时器的时钟会不精准。更多关于时钟源的信息请参考 复位与时钟一章。

LEDC_CLK*x* 经分频器分频后的输出时钟用作计数器的基准时钟。分频器的分频系数通过 LEDC_CLK_DIV_TIMER*x* 寄存器配置。该分频系数为定点数，高 10 位为整数部分 A，低 8 位为小数部分 B。分频系数 LEDC_CLK_DIV*x* 的公式为：

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

小数部分 B 不为 0 时，分频器的输入和输出时钟如图 9-3所示。256 个输出时钟中有 B 个以 (A+1) 分频，有 (256-B) 个以 A 分频。以 (A+1) 分频的 B 个输出时钟均匀分布在 256 个输出时钟中。

LED PWM 控制器的计数器为 14 位，计数最大值为 $2^{LEDC\_TIMERx\_DUTY\_RES} - 1$。每当计数值达到 $2^{LEDC\_TIMERx\_DUTY\_RES} - 1$ 时，计数器便会溢出，并重新从 0 开始计数。软件可以复位、暂停并读取计数器的计数值。计数器每次溢出时可产生 LEDC_TIMER*x*_OVF_INT 中断，也可在溢出 (LEDC_OVF_NUM_CH*n* + 1) 次时产生中断。寄存器配置流程如下：

1. 置位 LEDC_OVF_CNT_EN_CH*n*
2. 将 LEDC_OVF_NUM_CH*n* 配置为溢出次数减 1
3. 置位LEDC_OVF_CNT_CH*n*_INT_ENA
4. 置位 LEDC_TIMER*x*_DUTY_RES 启动定时器，等待 LEDC_OVF_CNT_CH*n*_INT 中断

PWM 生成器输出信号 sig_out*n* 的频率取决于分频器的分频系数以及计数器的计数范围：

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

重新设置分频器的分频系数和计数器的溢出值，需分别配置 LEDC_CLK_DIV_TIMER*x* 和 LEDC_TIMER*x*_DUTY_RES ，然后置位 LEDC_TIMER*x*_PARA_UP，否则配置无效。新的配置在计数器溢出时更新。

9.3.3 PWM 生成器

如图 9-2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 生成器获取 LED PWM 定时器的 14 位计数值，与高低电平比较器的值 Hpoint*n* 和 Lpoint*n* 比较，来控制 PWM 信号的电平。

- 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。

Hpoint*n* 由 LEDC_HPOINT_CH*n* 在计数器溢出时更新。Lpoint*n* 的初始值为计数器溢出时 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。

图 9-4 为占空比固定时的 PWM 波形。

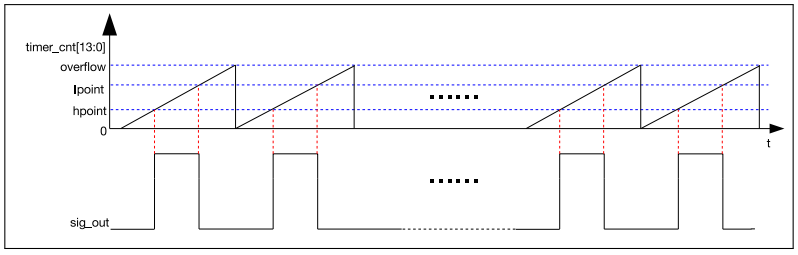


图 9-4. LED_PWM 输出信号图

LEDC_DUTY_CH*n* 是一个具有四位小数的定点寄存器。LEDC_DUTY_CH*n*[18..4] 为整数部分，可直接用于 PWM 计算。LEDC_DUTY_CH*n*[3..0] 为小数部分，可用于调整输出信号。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 脉冲宽度要比 (16 - LEDC_DUTY_CH*n*[3..0]) 个周期的脉冲宽度多一个定时器的计数周期。该功能可将 PWM 生成器的精度提至 18 位。

9.3.4 渐变占空比

LED PWM 输出信号可由一种占空比渐变为另一种占空比。渐变功能由 LEDC_DUTY_CH*n*、LEDC_DUTY_START_CH*n*、LEDC_DUTY_INC_CH*n* 、LEDC_DUTY_NUM_CH*n* 和 LEDC_DUTY_SCALE_CH*n* 配置。

LEDC_DUTY_START_CH*n* 用于更新 Lpoint*n* 的值。如置位该位，计数器溢出时 Lpoint*n* 会自动递增或递减。
LEDC_DUTY_INC_CH*n* 位用于设置渐变模式。

sig_out*n* 的占空比每隔 LEDC_DUTY_CYCLE_CH*n* 个 PWM 脉冲周期，便会在当前占空比上加上或减去 LEDC_DUTY_SCALE_CH*n*。

占空比渐变次数达到 LEDC_DUTY_NUM_CH*n* 时停止渐变，并产生 LEDC_DUTY_CHNG_END_CH*n*_INT。图 9-5为渐变的图例。该配置下，每 LEDC_DUTY_CYCLE_CH*n* 个 PWM 脉冲周期，sig_out*n*的占空比会增加 LEDC_DUTY_SCALE_CH*n*。

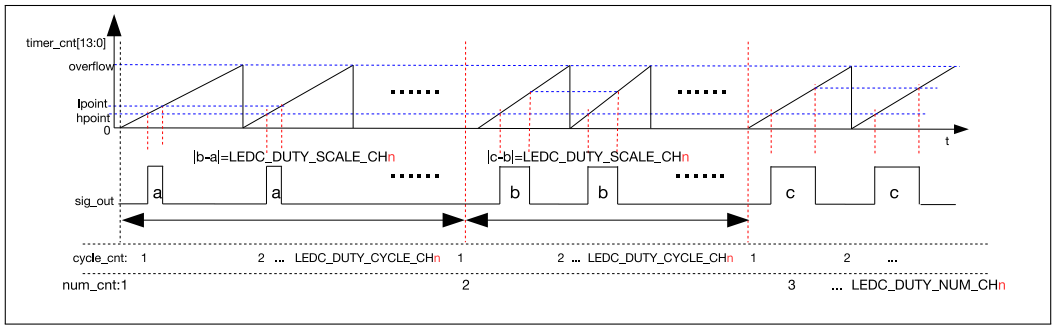


图 9-5. 渐变占空比输出信号图

LEDC_SIG_OUT_EN_CH*n* 用于启动 PWM 波形输出。LEDC_SIG_OUT_EN_CH*n* 为 0 时，sig_out*n*的电平值恒定为 LEDC_IDLE_LV_CH*n*。

软件更新 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*
和 LEDC_OVF_CNT_EN_CH*n* 的配置后，需置位 LEDC_PARA_UP_CH*n* 应用新配置。

9.3.5 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：LED PWM 生成器渐变完成后触发中断。
- LEDC_TIMER*x*_OVF_INT：LED PWM 定时器达到最大计数值时触发中断。

9.4 基地址

用户可以通过两个不同的寄存器基地址访问 LED PWM，如表 40 所示。更多信息，请访问章节 1 *系统和存储器*。

表 40: LED_PWM 基地址

访问总线	基地址
PeriBUS1	0x3F419000
PeriBUS2	0x60019000

9.5 寄存器列表

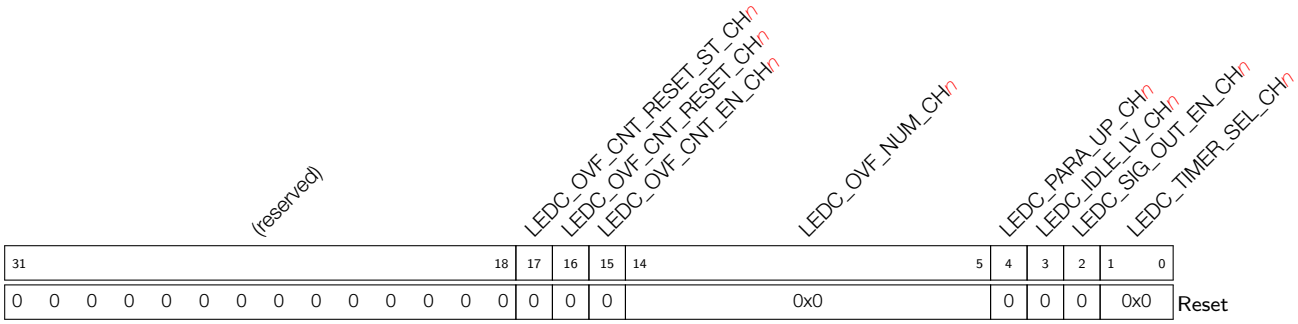
请注意，下表的地址指相对于 LED PWM 基地址的偏移量（相对地址）。请参阅章节 9.4 获取有关 LED PWM 基地址的信息。请注意，下表中的地址都是相对于 LED PWM 基地址的地址偏移量（相对地址）。更多有关 LED PWM 基地址的信息，请前往 9.4 章节。

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
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

名称	描述	地址	访问
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

9.6 寄存器

Register 9.1: LEDC_CH*n*_CONF0_REG (*n*: 0-7) (0x0000+20n*)**



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。(读/写)

LEDC_SIG_OUT_EN_CH*n* 置位此位，使能通道 *n* 的信号输出。(读/写)

LEDC_IDLE_LV_CH*n* 控制通道 *n* 不工作时的输出电平。(读/写)

LEDC_PARA_UP_CH*n* 用于更新通道 *n* 的 LEDC_CH*n*_HPOINT 和 LEDC_CH*n*_DUTY 寄存器。(只写)

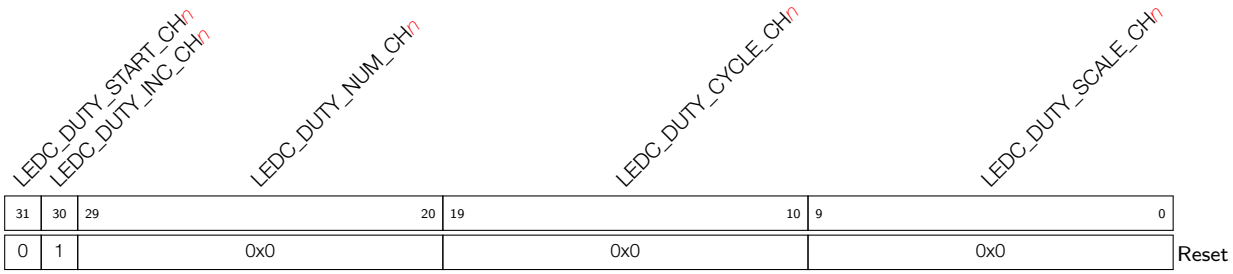
LEDC_OVF_NUM_CH*n* 用于配置定时器溢出次数的最大值减 1。通道 *n* 的定时器溢出次数达到 (LEDC_OVF_CNT_CH*n* + 1) 时触发 LEDC_OVF_CNT_CH*n*_INT 中断。(读/写)

LEDC_OVF_CNT_EN_CH*n* 用于使能通道 *n* 的 ovf_cnt。(读/写)

LEDC_OVF_CNT_RESET_CH*n* 置位此位，复位通道 *n* 的 ovf_cnt。(只写)

LEDC_OVF_CNT_RESET_ST_CH*n* LEDC_OVF_CNT_RESET_CH*n* 的状态位。(只读)

Register 9.2: LEDC_CH*n*_CONF1_REG (*n*: 0-7) (0x000C+20n*)**



LEDC_DUTY_SCALE_CH*n* 用于配置通道 *n* 占空比的变化步长。(读/写)

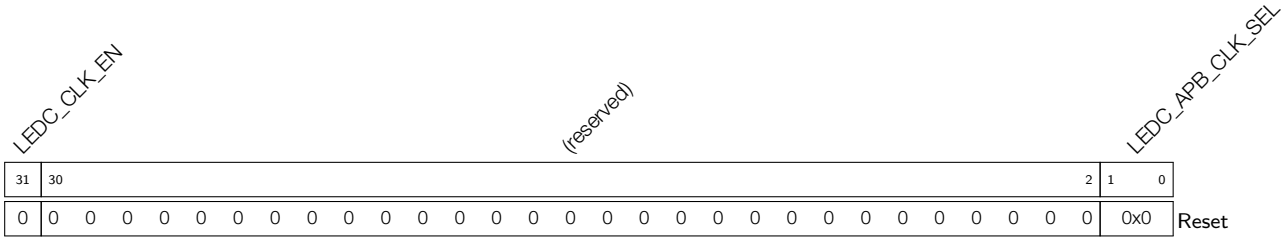
LEDC_DUTY_CYCLE_CH*n* 通道 *n* 占空比每隔 LEDC_DUTY_CYCLE_CH*n* 变化一次。(读/写)

LEDC_DUTY_NUM_CH*n* 用于控制占空比变化的次数。(读/写)

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。(读/写)

LEDC_DUTY_START_CH*n* 此位置 1 时，LEDC_CH*n*_CONF1_REG 中的其他字段生效。(读/写)

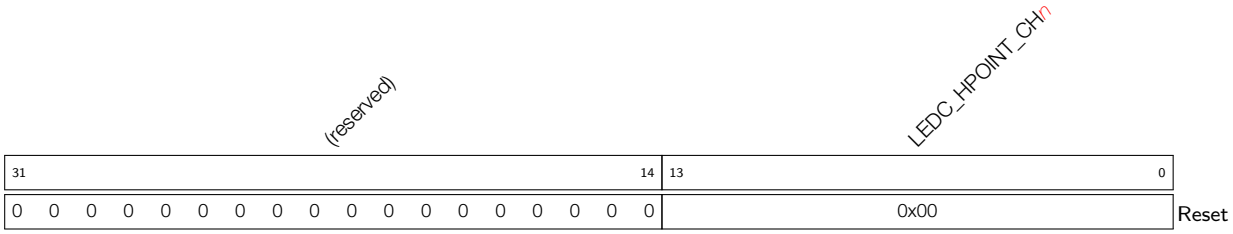
Register 9.3: LEDC_CONF_REG (0x00D0)



LEDC_APB_CLK_SEL 用于设置 4 个定时器的时钟源。1：APB_CLK；2：RTC8M_CLK；3：XTAL_CLK。(读/写)

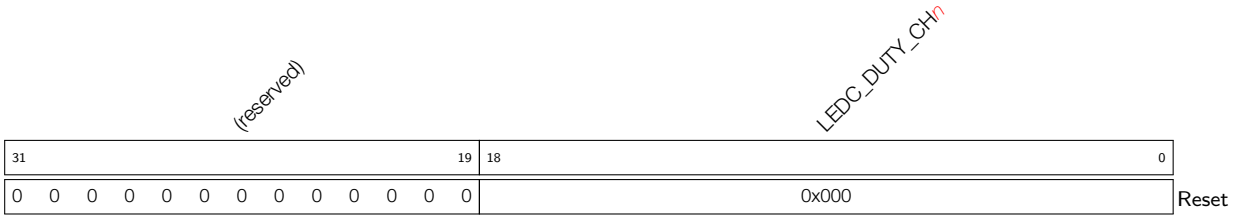
LEDC_CLK_EN 用于控制时钟。1'b1：强制开启寄存器时钟。1'b0：仅在应用写寄存器时支持时钟。(读/写)

Register 9.4: LEDC_CH*n*_HPOINT_REG (*n*: 0-7) (0x0004+20n*)**



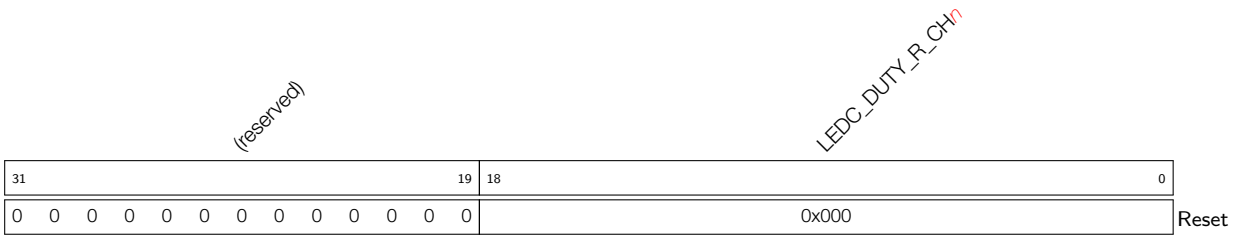
LEDC_HPOINT_CH*n* 所选定时器计数值达到该值时，输出信号翻转为高电平。(读/写)

Register 9.5: LEDC_CH*n*_DUTY_REG (*n*: 0-7) (0x0008+20n*)**



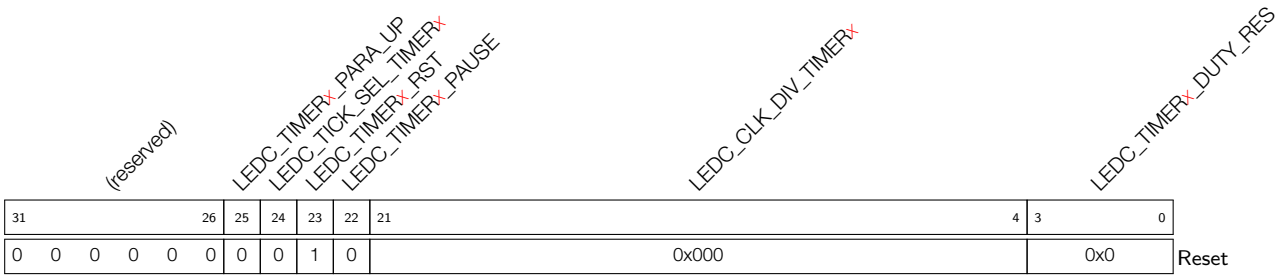
LEDC_DUTY_CH*n* 通过控制低位点改变输出信号占空比。所选定时器达到低位点时，输出信号翻转为低电平。(读/写)

Register 9.6: LEDC_CH*n*_DUTY_R_REG (*n*: 0-7) (0x0010+20**n*)



LEDC_DUTY_R_CH*n* 存储通道 *n* 输出信号的当前占空比。(只读)

Register 9.7: LEDC_TIMER*x*_CONF_REG (*x*: 0-3) (0x00A0+8**x*)



LEDC_TIMER*x*_DUTY_RES 用于控制定时器 *x* 计数器的计数范围。(读/写)

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。(读/写)

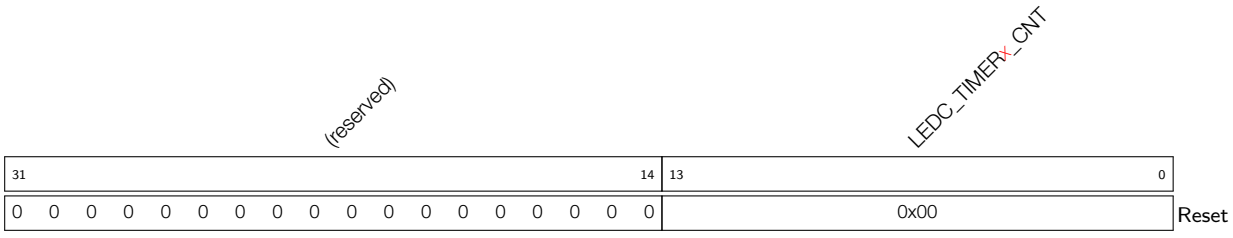
LEDC_TIMER*x*_PAUSE 用于暂停定时器 *x* 的计数器。(读/写)

LEDC_TIMER*x*_RST 用于复位定时器 *x*。复位后计数器为 0。(读/写)

LEDC_TICK_SEL_TIMER*x* 用于为定时器 *x* 选择时钟。此位置 1 时，LEDC_APB_CLK_SEL[1:0] 应为 1，否则定时器时钟会不精准。1'h0：SLOW_CLK 1'h1：REF_TICK (读/写)

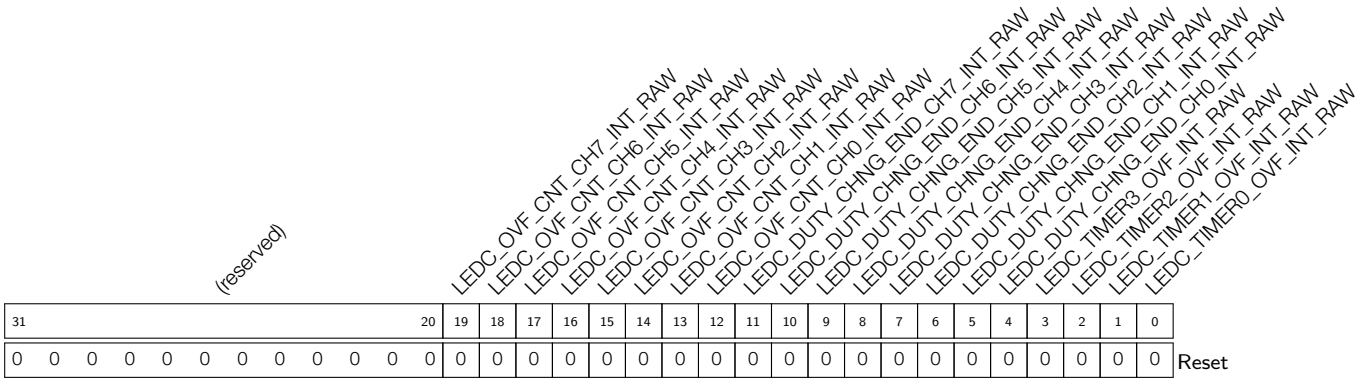
LEDC_TIMER*x*_PARA_UP 置位此位，更新 LEDC_CLK_DIV_TIMER*x* 和 LEDC_TIMER*x*_DUTY_RES。(只写)

Register 9.8: LEDC_TIMER*x*_VALUE_REG (*x*: 0-3) (0x00A4+8**x*)



LEDC_TIMER*x*_CNT 存储定时器 *x* 的当前计数器值 (只读)

Register 9.9: LEDC_INT_RAW_REG (0x00C0)

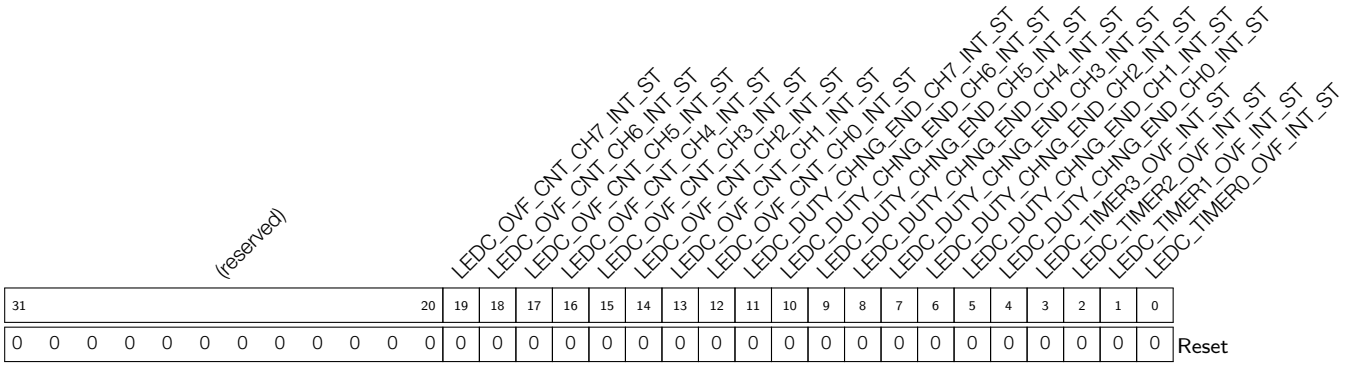


LEDC_TIMER*x*_OVF_INT_RAW 定时器 *x* 达到最大计数值时触发中断。(只读)

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。(只读)

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。(只读)

Register 9.10: LEDC_INT_ST_REG (0x00C4)

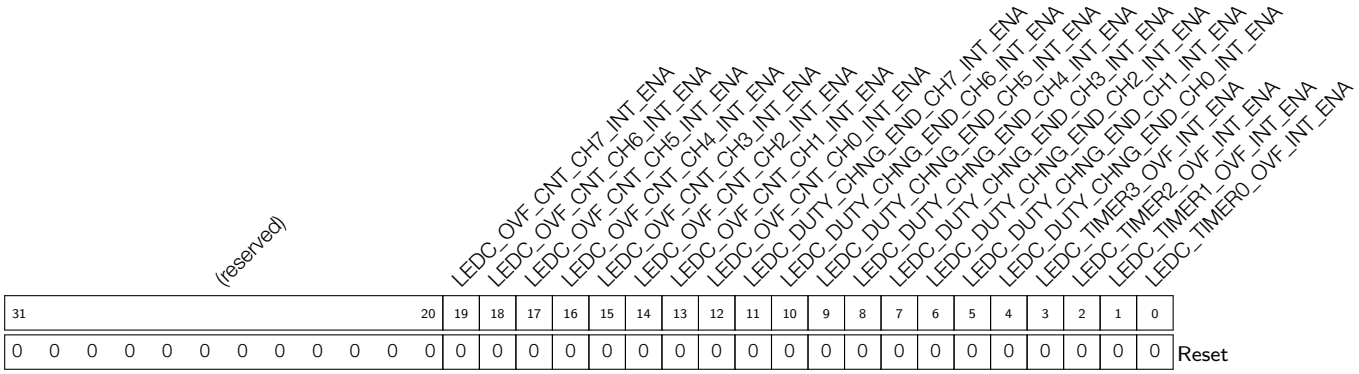


LEDC_TIMER*x*_OVF_INT_ST LEDC_TIMER*x*_OVF_INT_ENA 置 1 时，LEDC_TIMER*x*_OVF_INT 中断的屏蔽中断状态位。(只读)

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。(只读)

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。(只读)

Register 9.11: LEDC_INT_ENA_REG (0x00C8)

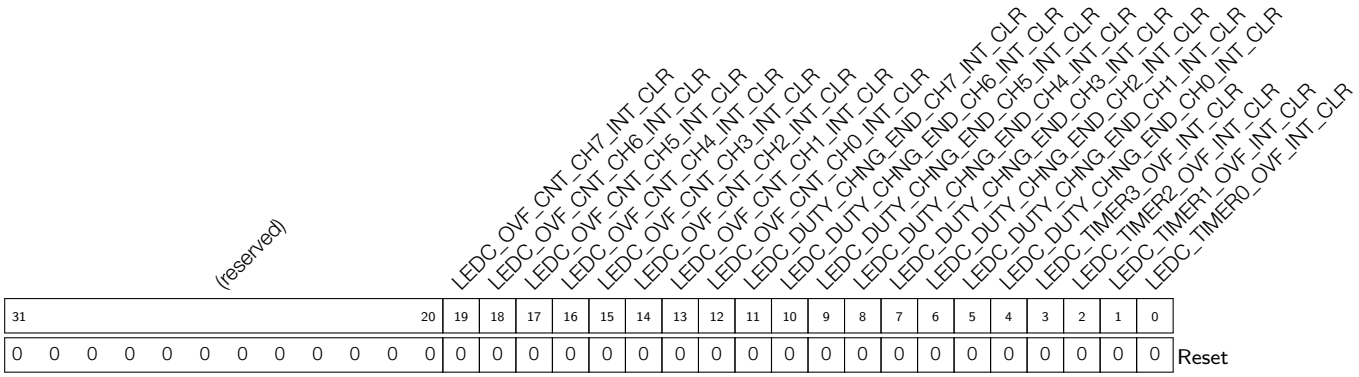


LEDC_TIMER*x*_OVF_INT_ENA LEDC_TIMER*x*_OVF_INT 中断的使能位。(读/写)

LEDC_DUTY_CHNG_END_CH*n*_INT_ENA LEDC_DUTY_CHNG_END_CH*n*_INT 中断的使能位。(读/写)

LEDC_OVF_CNT_CH*n*_INT_ENA LEDC_OVF_CNT_CH*n*_INT 中断的使能位。(读/写)

Register 9.12: LEDC_INT_CLR_REG (0x00CC)

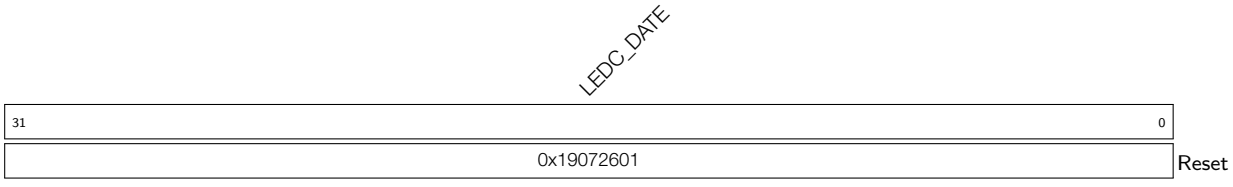


LEDC_TIMER*x*_OVF_INT_CLR 置位此位，清除 LEDC_TIMER*x*_OVF_INT 中断。(只写)

LEDC_DUTY_CHNG_END_CH*n*_INT_CLR 置位此位，清除 LEDC_DUTY_CHNG_END_CH*n*_INT 中断。(只写)

LEDC_OVF_CNT_CH*n*_INT_CLR 置位此位，清除 LEDC_OVF_CNT_CH*n*_INT 中断。(只写)

Register 9.13: LEDC_DATE_REG (0x00FC)



LEDC_DATE 版本控制寄存器。(读/写)

10. 红外遥控

10.1 概述

RMT（红外收发器）是一个红外发送/接收控制器，支持多种红外协议。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，或检测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处理。

RMT 共有四个通道，可独立用于信号发送或接收，编码为 0 ~ 3，每个通道有一组功能相同的寄存器。为了方便叙述，以 *n* 表示各个通道。

10.2 功能描述

10.2.1 RMT 架构

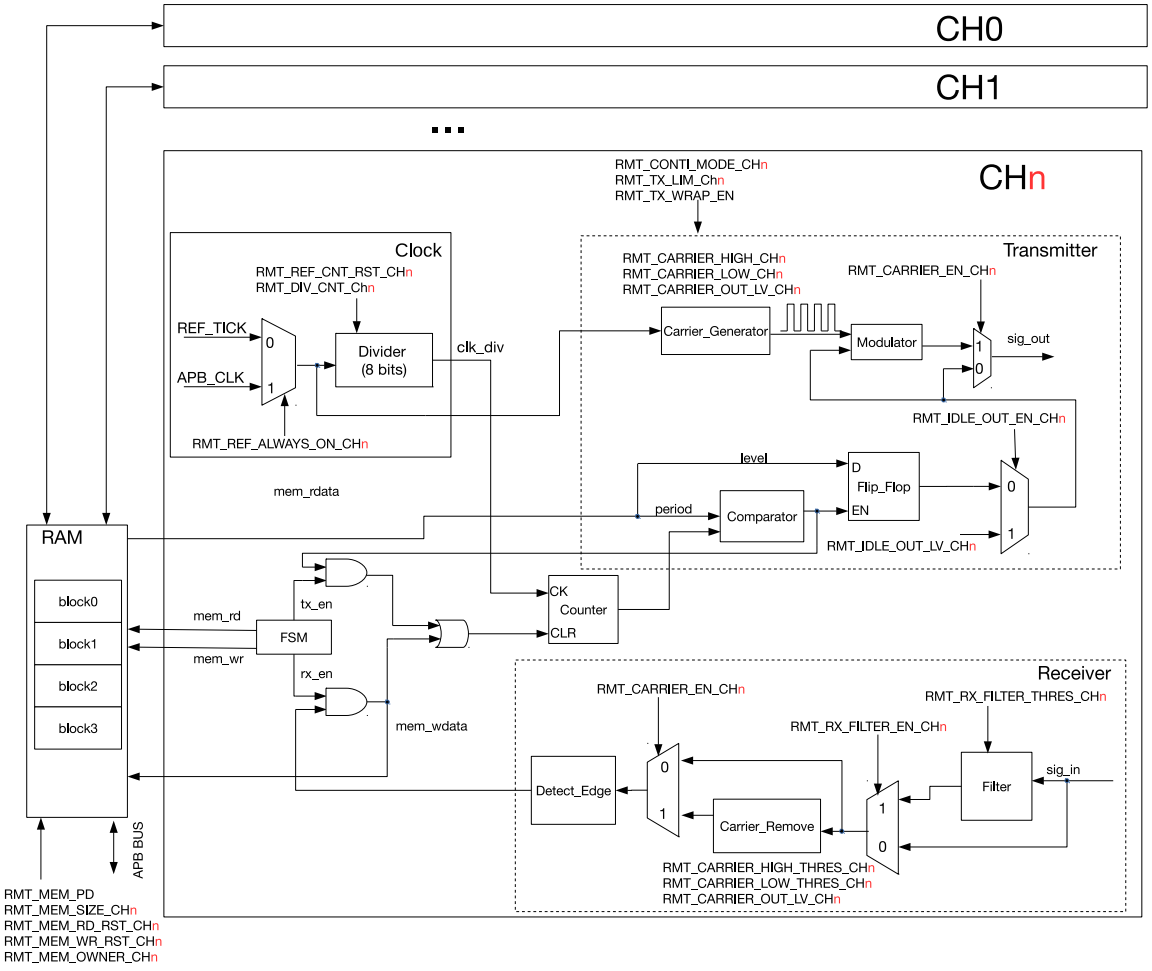


图 10-1. RMT 结构框图

RMT 模块有四个独立的通道，每个通道内部都有各自的一个时钟分频器，一个计数器，一个发射器和一个接收器。注意，同一个通道的发射器和接收器不可同时工作。四个通道共享一块 256 x 32 位的 RAM。

10.2.2 RMT RAM

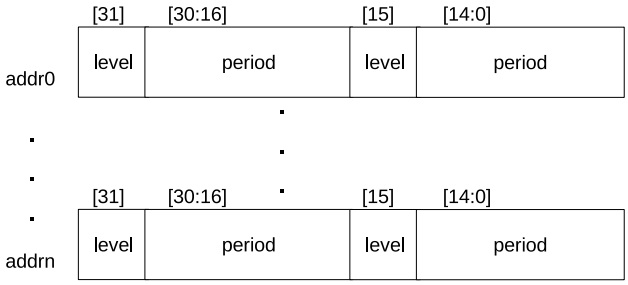


图 10-2. RAM 中脉冲编码结构

RAM 中脉冲编码结构如图 10-2 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10-1 clk_div）周期数。Period 为 0 是传输结束标志。

RAM 按照 64 x 32 位分成四个 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）。当通道 *n* 单次发送或接收的脉冲编码数大于一个 block 时，可以进行乒乓操作，或者通过配置 RMT_MEM_SIZE_CH*n* 寄存器，允许该通道占用多个 block。当设置 RMT_MEM_SIZE_CH*n* > 1 时，通道 *n* 将占用 block (*n*) ~ block (*n* + RMT_MEM_SIZE_CH*n* -1) 的存储空间。通道 *n* +1 ~ *n* + RMT_MEM_SIZE_CH*n* - 1 因为对应的 RAM block 被占用而无法使用。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 可以通过置位 RMT_MEM_SIZE_CH*n* 寄存器来使用通道 1、2、3 的 RAM 空间，但是通道 3 不能使用通道 0、1 或 2 的 RAM 空间。

RAM 可被 APB 总线及通道的发射器或接收器访问，为了防止发射器和接收器访问 RAM 时发生冲突，用户可以通过配置 RMT_MEM_OWNER_CH*n* 来决定当前 RAM 的使用权。当通道的发射器或接收器发生越权访问时会产生 RMT_MEM_OWNER_ERR_CH*n* 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。

10.2.3 时钟

用户可以通过配置 RMT_REF_ALWAYS_ON_CH*n* 选择时钟分频器的驱动时钟：APB_CLK 或者 REF_TICK，请参考复位和时钟章节。RMT_DIV_CNT_CH*n* 寄存器可以配置分频器的分频系数，除 0 表示 256 分频外，其他分频数等同于寄存器 RMT_DIV_CNT_CH*n* 的值。时钟分频器可以通过配置 RMT_REF_CNT_RST_CH*n* 进行复位。时钟分频器的分频时钟可供计数器使用。

10.2.4 发射器

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* 寄存器可以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* 寄存器决定。用户可以配置 RMT_IDLE_OUT_EN_CH*n* 寄存器来选择这两种方式。

当发送的脉冲编码较多时，可通过置位 RMT_MEM_TX_WRAP_EN 寄存器使能乒乓操作。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 64 * *n* 地址开始发送，然后对应 RAM 的地址递增。发完 (64 * (*n* +1) - 1) 地址的数据后，下次继续从 64 * *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。

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 APB_CLK 或者 REF_TICK 时钟周期，低电平持续的时间为 (RMT_CARRIER_LOW_CH*n* +1) 个 APB_CLK 或者 REF_TICK 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* 寄存器，可以选择这两种模式。

置位 RMT_TX_CONTI_MODE_CH*n* 寄存器可以使能发射器的持续发送功能。置位该寄存器后，发射器会循环发送 RAM 中的脉冲编码。配置 RMT_TX_LOOP_CNT_EN_CH*n* 后发射器会记录循环发送的次数。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。

置位 RMT_TX_SIM_EN 可以使能发射器多个通道同步发送的功能，此时多个通道会同时启动发送。RMT_TX_SIM_CH*n* 用于选择同步发送的通道。

10.2.5 接收器

RMT_RX_EN_CH*n* 置为 1 时接收器开始工作。接收器会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*n* 时，接收器结束接收过程，返回空闲状态，并产生 RMT_CH*n*_RX_END_INT 中断。

每个通道都可以通过置位 RMT_RX_FILTER_EN_CH*n* 使能接收器对输入信号进行滤波的功能。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*n* 个 APB 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_THRES_CH*n* 个 APB 时钟周期的线路毛刺。

10.2.6 中断

- RMT_CH*n*_ERR_INT：当通道 *n* 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*n*_TX_END_INT：当发射器停止发送信号时，即触发此中断。
- RMT_CH*n*_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。

10.3 基地址

用户可以通过两个不同的寄存器基地址访问 RMT，如表 42 所示。更多信息，请访问章节 1 *系统和存储器*。

表 42: RMT 基地址

访问总线	基地址
PeriBUS1	0x3F416000
PeriBUS2	0x60016000

10.4 寄存器列表

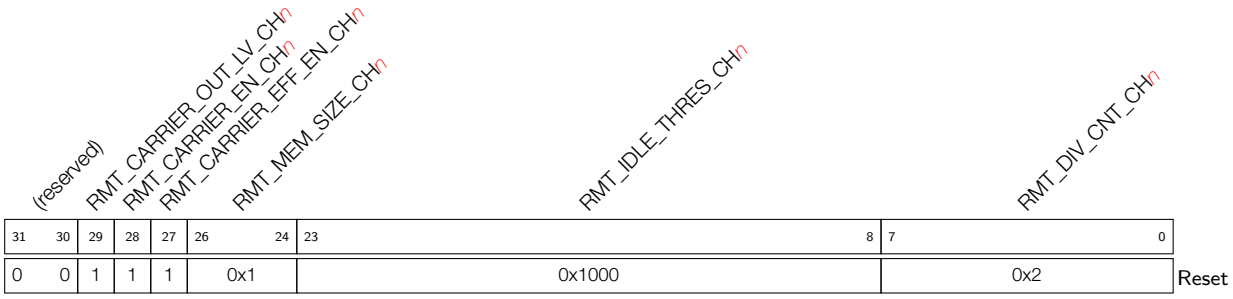
请注意，下表中的地址都是相对于 RMT 基地址的地址偏移量（相对地址）。更多有关 RMT 基地址的信息，请前往第 10.3 节。

名称	描述	地址	访问
配置寄存器			
RMT_CH0CONF0_REG	通道 0 配置寄存器 0	0x0010	读/写
RMT_CH0CONF1_REG	通道 0 配置寄存器 1	0x0014	不定
RMT_CH1CONF0_REG	通道 1 配置寄存器 0	0x0018	读/写
RMT_CH1CONF1_REG	通道 1 配置寄存器 1	0x001C	不定
RMT_CH2CONF0_REG	通道 2 配置寄存器 0	0x0020	读/写
RMT_CH2CONF1_REG	通道 2 配置寄存器 1	0x0024	不定
RMT_CH3CONF0_REG	通道 3 配置寄存器 0	0x0028	读/写
RMT_CH3CONF1_REG	通道 3 配置寄存器 1	0x002C	不定
RMT_APB_CONF_REG	RMT APB 配置寄存器	0x0080	读/写
RMT_REF_CNT_RST_REG	RMT 时钟分频器复位寄存器	0x0088	读/写
RMT_CH0_RX_CARRIER_RM_REG	通道 0 载波清除寄存器	0x008C	读/写
RMT_CH1_RX_CARRIER_RM_REG	通道 1 载波清除寄存器	0x0090	读/写
RMT_CH2_RX_CARRIER_RM_REG	通道 2 载波清除寄存器	0x0094	读/写
RMT_CH3_RX_CARRIER_RM_REG	通道 3 载波清除寄存器	0x0098	读/写
载波占空比寄存器			
RMT_CH0CARRIER_DUTY_REG	通道 0 占空比配置寄存器	0x0060	读/写
RMT_CH1CARRIER_DUTY_REG	通道 1 占空比配置寄存器	0x0064	读/写
RMT_CH2CARRIER_DUTY_REG	通道 2 占空比配置寄存器	0x0068	读/写
RMT_CH3CARRIER_DUTY_REG	通道 3 占空比配置寄存器	0x006C	读/写
发送事件配置寄存器			
RMT_CH0_TX_LIM_REG	通道 0 Tx 配置寄存器	0x0070	不定
RMT_CH1_TX_LIM_REG	通道 1 Tx 配置寄存器	0x0074	不定
RMT_CH2_TX_LIM_REG	通道 2 Tx 配置寄存器	0x0078	不定
RMT_CH3_TX_LIM_REG	通道 3 Tx 配置寄存器	0x007C	不定
RMT_TX_SIM_REG	RMT 同步发送寄存器	0x0084	读/写
状态寄存器			
RMT_CH0STATUS_REG	通道 0 状态寄存器	0x0030	只读
RMT_CH1STATUS_REG	通道 1 状态寄存器	0x0034	只读
RMT_CH2STATUS_REG	通道 2 状态寄存器	0x0038	只读
RMT_CH3STATUS_REG	通道 3 状态寄存器	0x003C	只读
RMT_CH0ADDR_REG	通道 0 地址寄存器	0x0040	只读
RMT_CH1ADDR_REG	通道 1 地址寄存器	0x0044	只读
RMT_CH2ADDR_REG	通道 2 地址寄存器	0x0048	只读
RMT_CH3ADDR_REG	通道 3 地址寄存器	0x004C	只读
版本寄存器			
RMT_DATE_REG	版本控制寄存器	0x00FC	读/写
FIFO 读/写寄存器			
RMT_CH0DATA_REG	配置 APB FIFO 对通道 0 进行读写操作	0x0000	只读
RMT_CH1DATA_REG	配置 APB FIFO 对通道 1 进行读写操作	0x0004	只读

名称	描述	地址	访问
RMT_CH2DATA_REG	配置 APB FIFO 对通道 2 进行读写操作	0x0008	只读
RMT_CH3DATA_REG	配置 APB FIFO 对通道 3 进行读写操作	0x000C	只读
中断寄存器			
RMT_INT_RAW_REG	原始中断状态寄存器	0x0050	只读
RMT_INT_ST_REG	隐蔽中断状态寄存器	0x0054	只读
RMT_INT_ENA_REG	中断使能寄存器	0x0058	读/写
RMT_INT_CLR_REG	中断清除寄存器	0x005C	只写

10.5 寄存器

Register 10.1: RMT_CH*n*CONF0_REG (*n*: 0-3) (0x0010+8n*)**



RMT_DIV_CNT_CH*n* 用于配置通道 *n* 分频器的分频系数。(读/写)

RMT_IDLE_THRES_CH*n* 当接收器长时间检测不到信号沿变化，即计数器的值大于等于此寄存器的值时，接收器结束接收过程。(读/写)

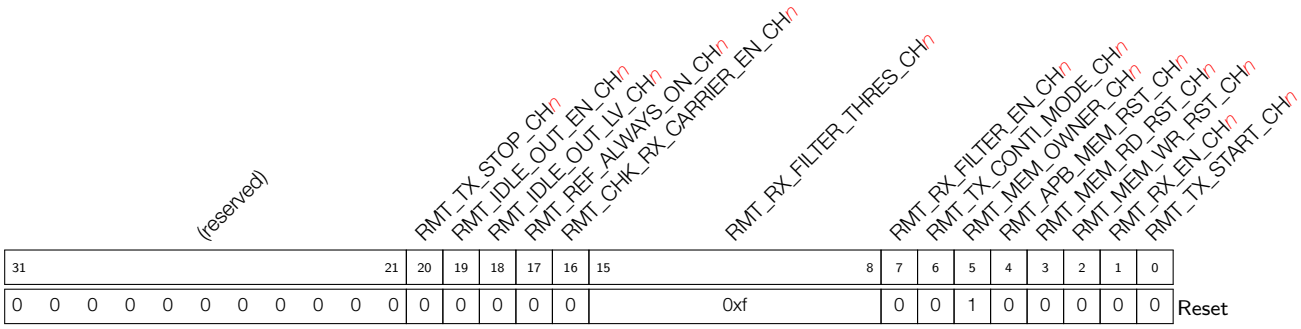
RMT_MEM_SIZE_CH*n* 配置通道 *n* 可以使用的 block 数量，范围为 1 ~ 4-*n*。(读/写)

RMT_CARRIER_EFF_EN_CH*n* 1：配置通道 *n* 仅在发送状态下对输出信号载波调制；0：配置通道 *n* 对所有状态（空闲状态、从 RAM 中读取数据和发送数据阶段）均加载载波。仅在 RMT_CARRIER_EN_CH*n* 为 1 时有效。(读/写)

RMT_CARRIER_EN_CH*n* 通道 *n* 的载波调制使能位。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。(读/写)

RMT_CARRIER_OUT_LV_CH*n* 用于配置通道 *n* 的载波调制方式。1'h0：载波加载在输出信号低电平上；1'h1：载波加载在输出信号高电平上。(读/写)

Register 10.2: RMT_CH*n*CONF1_REG (*n*: 0-3) (0x0014+8n*)**



RMT_TX_START_CH*n* 用于使能通道 *n* 的发射器。发射器开始发送数据。(读/写)

RMT_RX_EN_CH*n* 用于使能通道 *n* 的接收器。接收器开始接收数据。(读/写)

RMT_MEM_WR_RST_CH*n* 用于复位通道 *n* 中接收器访问的 RAM 写地址。(只写)

RMT_MEM_RD_RST_CH*n* 用于复位通道 *n* 中发射器访问的 RAM 读地址。(只写)

RMT_MEM_OWNER_CH*n* 标志通道 *n* 的 RAM 使用权。1'h1：接收器有权使用该通道的 RAM；1'h0：发射器有权使用该通道的 RAM。(读/写)

RMT_TX_CONTI_MODE_CH*n* 设置通道 *n* 发送结束后，发射器不进入空闲状态，而是继续从第一个字节开始发送数据。(读/写)

RMT_RX_FILTER_EN_CH*n* 使能通道 *n* 接收器的滤波功能。(读/写)

RMT_RX_FILTER_THRES_CH*n* 接收模式下，忽略宽度小于 RMT_RX_FILTER_THRES_CH*n* 个 APB 时钟周期的脉冲。(读/写)

RMT_CHK_RX_CARRIER_EN_CH*n* 用于使能 RAM 读循环模式（通道 *n* 中需使能载波调制模式）。(读/写)

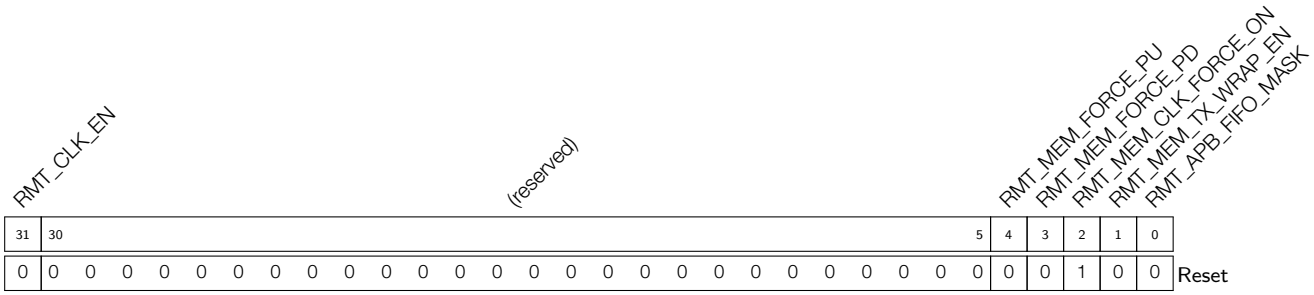
RMT_REF_ALWAYS_ON_CH*n* 用于选择通道 *n* 的基础时钟。1'h1：APB_CLK；1'h0：REF_TICK (读/写)

RMT_IDLE_OUT_LV_CH*n* 配置通道 *n* 处于空闲状态下的输出信号电平。(读/写)

RMT_IDLE_OUT_EN_CH*n* 通道 *n* 处于空闲状态下的输出使能位。(读/写)

RMT_TX_STOP_CH*n* 通道 *n* 中发射器停止发送信息。(读/写)

Register 10.3: RMT_APB_CONF_REG (0x0080)



RMT_APB_FIFO_MASK 1'h1：直接访问 RAM；1'h0：经 APB FIFO 访问 RAM。（读/写）

RMT_MEM_TX_WRAP_EN 乒乓模式使能位。该模式下，当发射器发送的数据量大于 block 大小，发射器将继续从第一字节开始发送数据。（读/写）

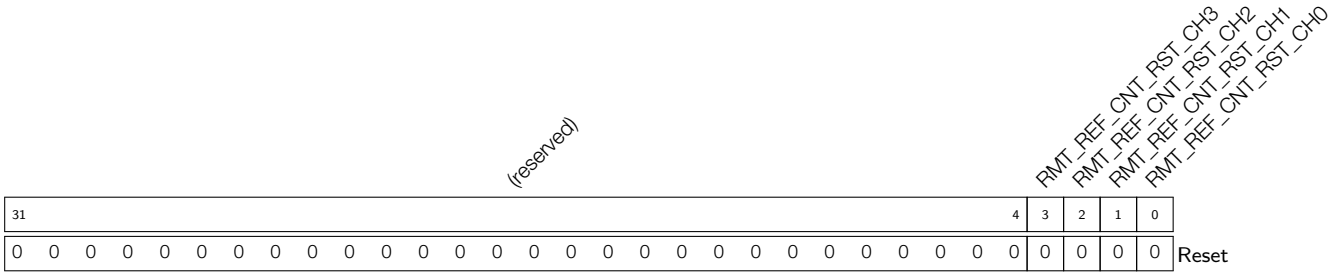
RMT_MEM_CLK_FORCE_ON RMT 模块工作时，使能 RAM 的时钟；RMT 模块不工作时，关闭 RAM 的时钟，从而实现低功耗。（读/写）

RMT_MEM_FORCE_PD 降低 RMT RAM 的功耗。（读/写）

RMT_MEM_FORCE_PU 1：禁用 RMT RAM 的 Light-sleep 低功耗功能；0：RMT 处于 Light-sleep 模式时，设置 RAM 进入低功耗模式。（读/写）

RMT_CLK_EN RMT 寄存器的时钟门控使能位，用于 RMT 寄存器低功耗控制。1：使能 RMT 寄存器驱动时钟；0：禁用 RMT 寄存器驱动时钟。（读/写）

Register 10.4: RMT_REF_CNT_RST_REG (0x0088)



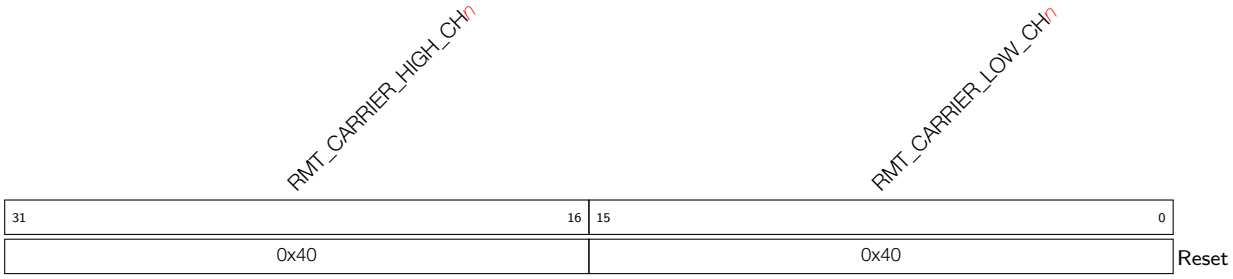
RMT_REF_CNT_RST_CH*n* 复位通道 *n* 的时钟分频器。（读/写）

Register 10.5: RMT_CH*n*_RX_CARRIER_RM_REG (*n*: 0-3) (0x008C+4*n)

RMT_CARRIER_LOW_THRES_CH*n* 载波调制模式下，通道 *n* 低电平周期为 (RMT_CARRIER_LOW_THRES_CH*n* + 1) 个时钟周期。(读/写)

RMT_CARRIER_HIGH_THRES_CH*n* 载波模式下，通道 *n* 高电平周期为 (RMT_CARRIER_HIGH_THRES_CH*n* + 1) 个时钟周期。(读/写)

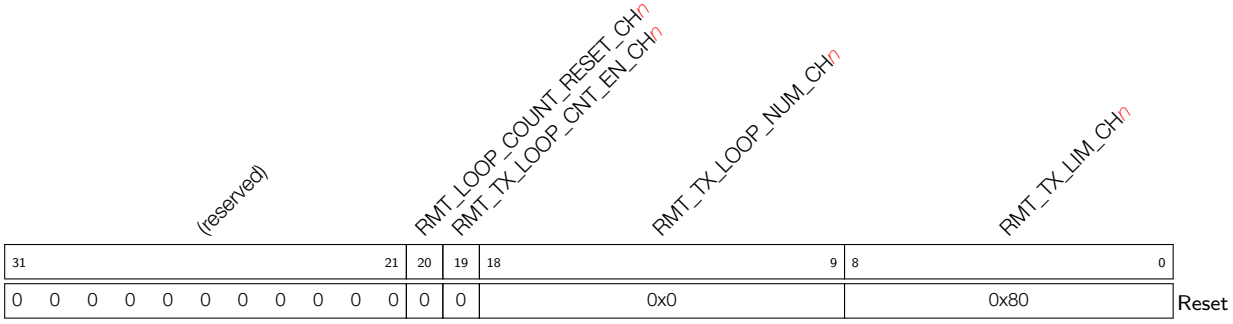
Register 10.6: RMT_CH*n*CARRIER_DUTY_REG (*n*: 0-3) (0x0060+4*n)



RMT_CARRIER_LOW_CH*n* 用于配置通道 *n* 载波的低电平时钟周期。(读/写)

RMT_CARRIER_HIGH_CH*n* 用于配置通道 *n* 载波的高电平时钟周期。(读/写)

Register 10.7: RMT_CH*n*_TX_LIM_REG (*n*: 0-3) (0x0070+4*n)



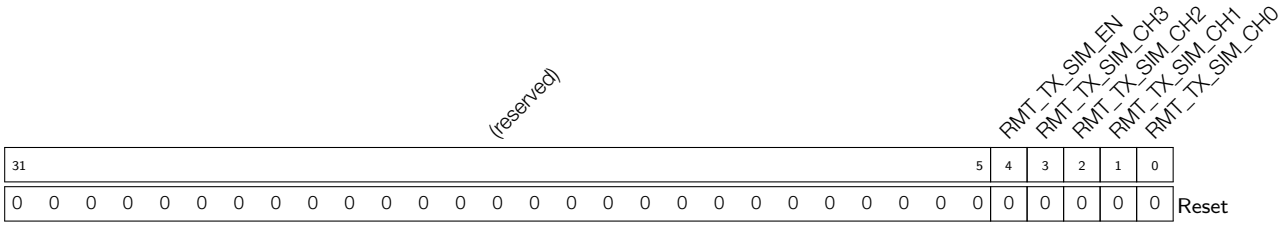
RMT_TX_LIM_CH*n* 用于配置通道 *n* 发送脉冲编码数量的上限值。当 RMT_MEM_SIZE_CH*n* = 1 时，此寄存器可设置为 0 ~ 128 (64 * 32 /16 = 128) 之间的值；当 RMT_MEM_SIZE_CH*n* > 1 时，此寄存器可设置为 (0 ~ 128) * RMT_MEM_SIZE_CH*n* 之间的值。(读/写)

RMT_TX_LOOP_NUM_CH*n* 用于配置连续发送模式下最大循环发送次数。(读/写)

RMT_TX_LOOP_CNT_EN_CH*n* 使能循环计数。(读/写)

RMT_LOOP_COUNT_RESET_CH*n* 重置 RMT_TX_CONTI_MODE_CH*n* 模式下循环计数器。(只写)

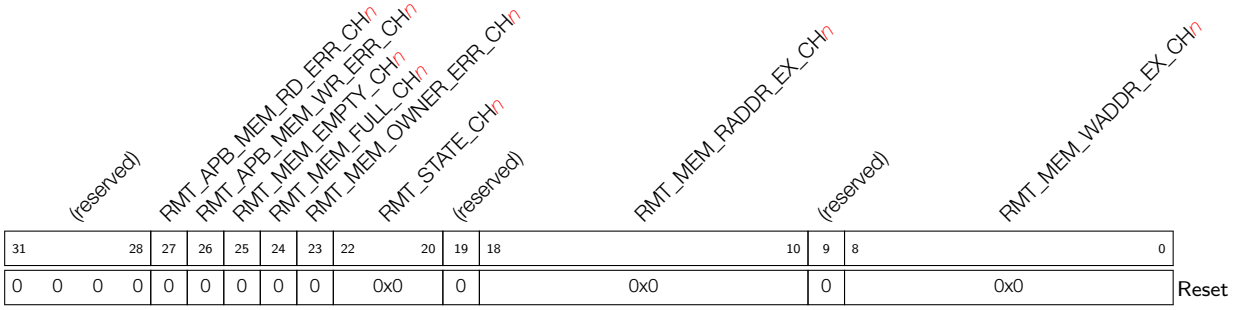
Register 10.8: RMT_TX_SIM_REG (0x0084)



RMT_TX_SIM_CH*n* 使能通道 *n* 与其他启用的通道同步开始发送数据。(读/写)

RMT_TX_SIM_EN 使能多通道同步发送数据。(读/写)

Register 10.9: RMT_CH*n*STATUS_REG (*n*: 0-3) (0x0030+4n*)**



RMT_MEM_WADDR_EX_CH*n* 记录通道 *n* 中接收器使用 RAM 时的地址偏移量。(只读)

RMT_MEM_RADDR_EX_CH*n* 记录通道 *n* 中发射器使用 RAM 时的地址偏移量。(只读)

RMT_STATE_CH*n* 记录通道 *n* 的 FSM 状态。(只读)

RMT_MEM_OWNER_ERR_CH*n* RAM block 使用权发生错误时，将触发此状态位。(只读)

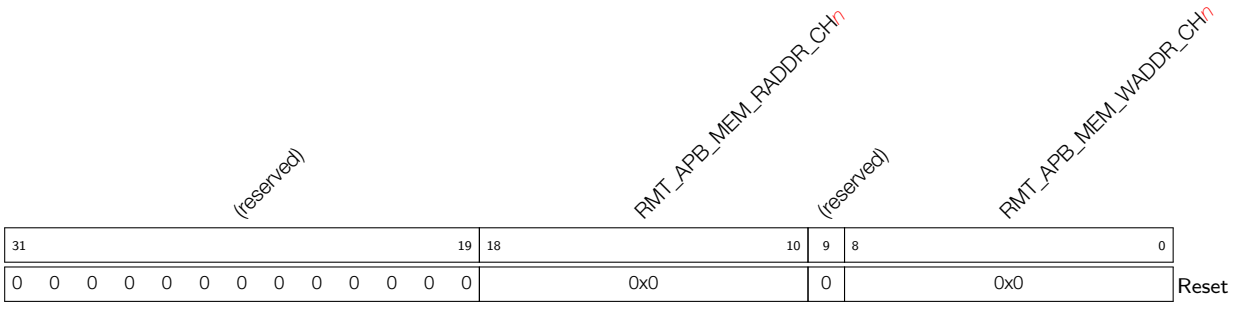
RMT_MEM_FULL_CH*n* 接收器接收的数据量大于 RAM block 时，将触发此状态位。(只读)

RMT_MEM_EMPTY_CH*n* 待发送的数据量大于 RAM block，且乒乓模式未启用时，将触发此状态位。(只读)

RMT_APB_MEM_WR_ERR_CH*n* 通过 APB 总线执行写操作时，如果偏移地址溢出 RAM block，将触发此状态位。(只读)

RMT_APB_MEM_RD_ERR_CH*n* 通过 APB 总线执行读操作时，如果偏移地址溢出 RAM block，将触发此状态位。(只读)

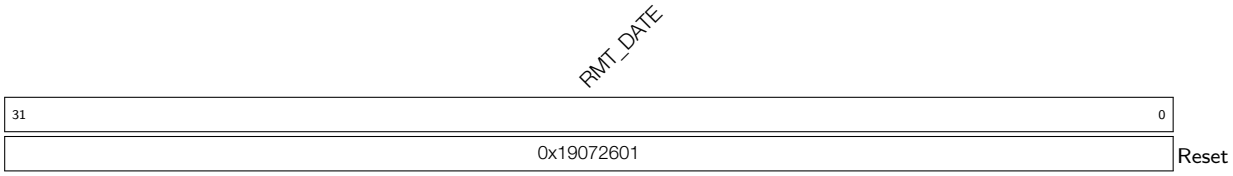
Register 10.10: RMT_CH*n*ADDR_REG (*n*: 0-3) (0x0040+4n*)**



RMT_APB_MEM_WADDR_CH*n* 记录通过 APB 总线对 RAM 执行写操作时，地址的偏移量。(只读)

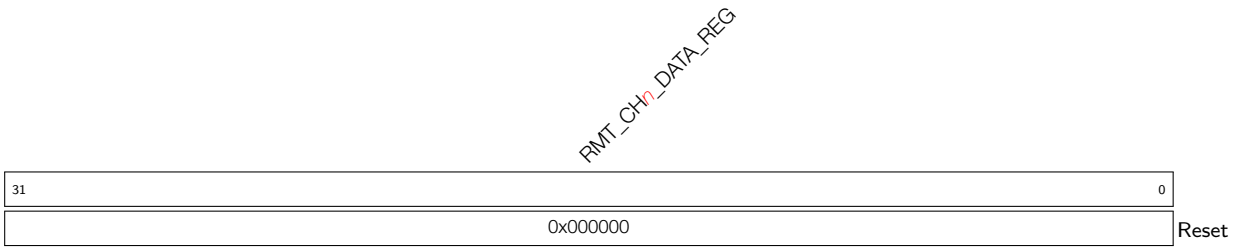
RMT_APB_MEM_RADDR_CH*n* 记录通过 APB 总线对 RAM 执行读操作时，地址的偏移量。(只读)

Register 10.11: RMT_DATE_REG (0x00FC)



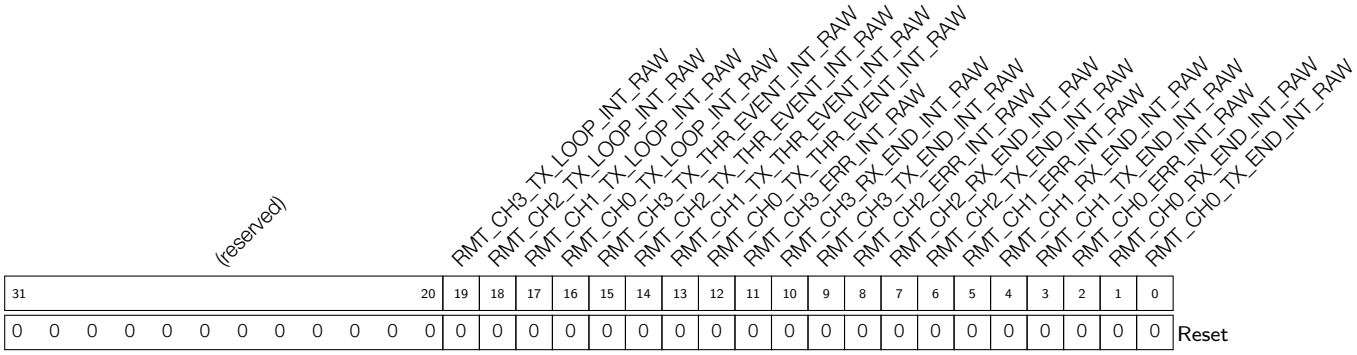
RMT_DATE 版本控制寄存器（读/写）

Register 10.12: RMT_CH*n*DATA_REG (*n*: 0-3) (0x0000+4n*)**



RMT_CH*n*_DATA_REG 通过 APB FIFO 对通道 *n* 进行读写操作。（只读）

Register 10.13: RMT_INT_RAW_REG (0x0050)



RMT_CH*n*_TX_END_INT_RAW 通道 *n* 的原始中断状态位。发送结束即触发该中断。（只读）

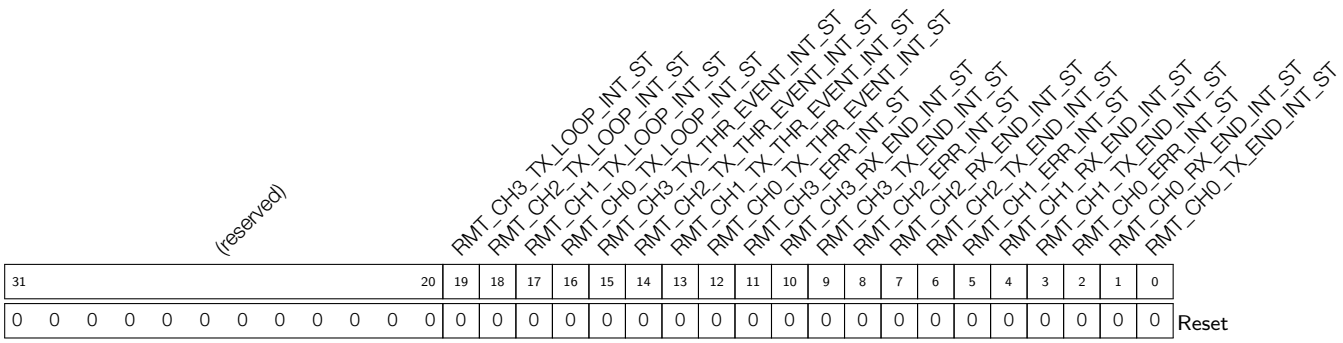
RMT_CH*n*_RX_END_INT_RAW 通道 *n* 的原始中断状态位。接收结束即触发该中断。（只读）

RMT_CH*n*_ERR_INT_RAW 通道 *n* 的原始中断状态位。发生错误即触发该中断。（只读）

RMT_CH*n*_TX_THR_EVENT_INT_RAW 通道 *n* 的原始中断状态位。发送的数据量大于预先设定的值时即触发该中断。（只读）

RMT_CH*n*_TX_LOOP_INT_RAW 通道 *n* 的原始中断状态位。循环次数达到预先设定的阈值时即触发该中断。（只读）

Register 10.14: RMT_INT_ST_REG (0x0054)



RMT_CH*n*_TX_END_INT_ST RMT_CH*n*_TX_END_INT 的隐蔽中断状态位。(只读)

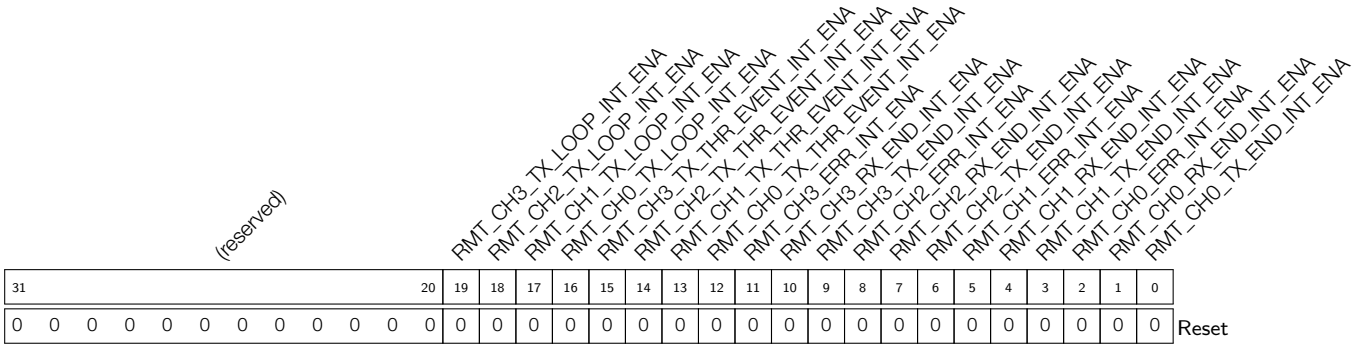
RMT_CH*n*_RX_END_INT_ST RMT_CH*n*_RX_END_INT 的隐蔽中断状态位。(只读)

RMT_CH*n*_ERR_INT_ST RMT_CH*n*_ERR_INT 的隐蔽中断状态位。(只读)

RMT_CH*n*_TX_THR_EVENT_INT_ST RMT_CH*n*_TX_THR_EVENT_INT 的隐蔽中断状态位。(只读)

RMT_CH*n*_TX_LOOP_INT_ST RMT_CH*n*_TX_LOOP_INT 的隐蔽中断状态位。(只读)

Register 10.15: RMT_INT_ENA_REG (0x0058)



RMT_CH*n*_TX_END_INT_ENA RMT_CH*n*_TX_END_INT 中断使能位。(读/写)

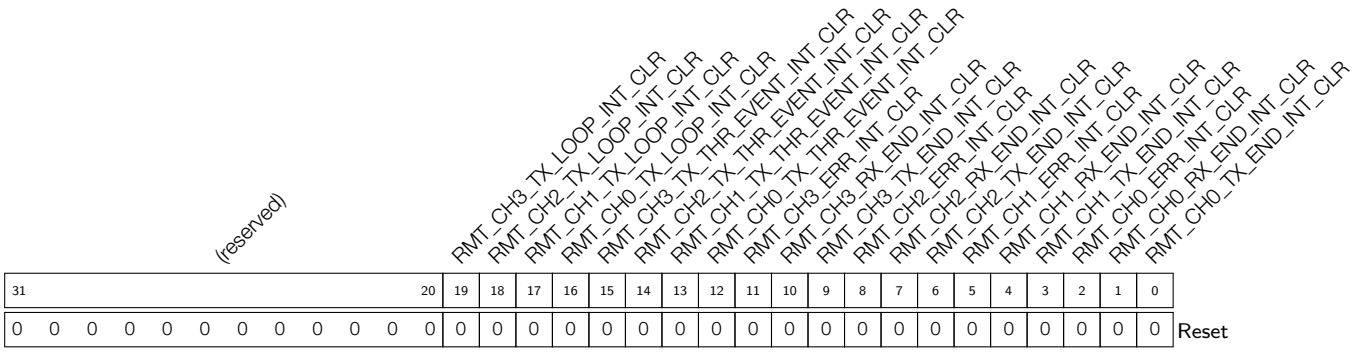
RMT_CH*n*_RX_END_INT_ENA RMT_CH*n*_RX_END_INT 中断使能位。(读/写)

RMT_CH*n*_ERR_INT_ENA RMT_CH*n*_ERR_INT 中断使能位。(读/写)

RMT_CH*n*_TX_THR_EVENT_INT_ENA RMT_CH*n*_TX_THR_EVENT_INT 中断使能位。(读/写)

RMT_CH*n*_TX_LOOP_INT_ENA RMT_CH*n*_TX_LOOP_INT 中断使能位。(读/写)

Register 10.16: RMT_INT_CLR_REG (0x005C)



RMT_CH*n*_TX_END_INT_CLR 用于清除 RMT_CH*n*_TX_END_INT 中断。(只写)

RMT_CH*n*_RX_END_INT_CLR 用于清除 RMT_CH*n*_RX_END_INT 中断。(只写)

RMT_CH*n*_ERR_INT_CLR 用于清除 RMT_CH*n*_ERR_INT 中断。(只写)

RMT_CH*n*_TX_THR_EVENT_INT_CLR 用于清除 RMT_CH*n*_TX_THR_EVENT_INT 中断。(只写)

RMT_CH*n*_TX_LOOP_INT_CLR 用于清除 RMT_CH*n*_TX_LOOP_INT 中断。(只写)

11. 脉冲计数器

脉冲计数器 (Pulse Count Controller, PCNT) 用于对输入脉冲计数和产生中断，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”的独立脉冲计数器，这些单元拥有自己的寄存器。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 11-1 所示，每个通道有两个输入信号：

1. 一个控制信号（如 ctrl_ch0_u*n* 为 ch0 的控制信号）
2. 一个脉冲输入信号（如 sig_ch0_u*n* 为 ch0 的脉冲输入信号）

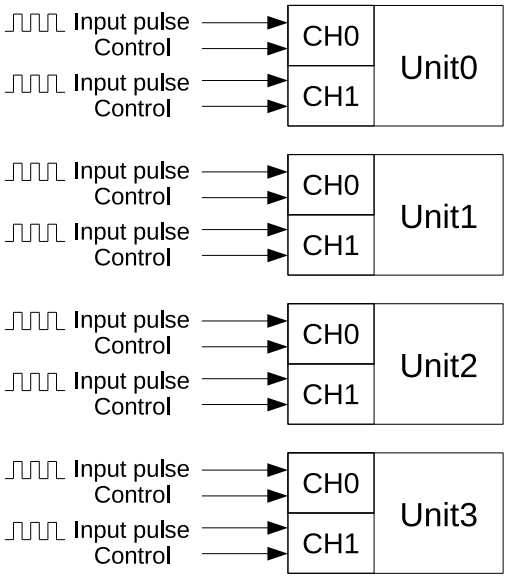


图 11-1. PCNT 框图

11.1 特性

PCNT 有如下特性：

- 四个脉冲计数器（单元），各自独立工作
- 每个单元有两个独立的通道，共用一个脉冲计数器
- 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，过滤每个单元的输入脉冲信号（sig_ch0_u*n* 和 sig_ch1_u*n*）和控制信号（ctrl_ch0_u*n* 和 ctrl_ch1_u*n*）
- 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。

11.2 功能描述

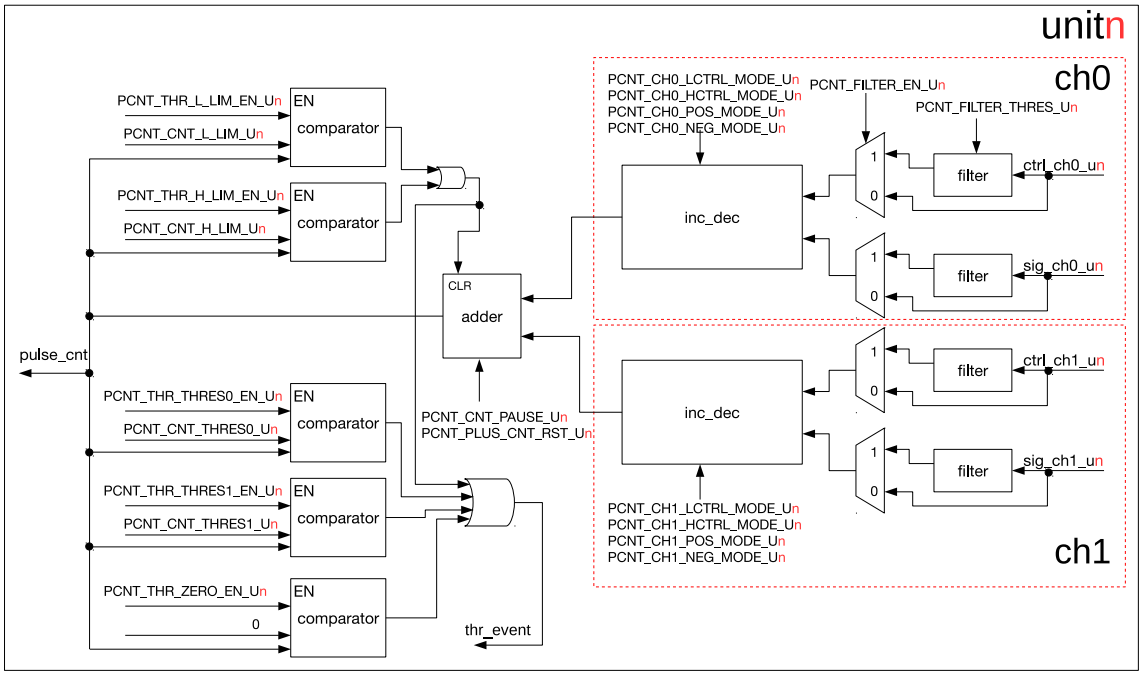


图 11-2. PCNT 单元基本架构图

图 11-2 为 PCNT 单元的基本架构图。如上所述，ctrl_ch0_u*n* 为 ch0 的控制信号，当控制信号 ctrl_ch0_u*n* 为高电平或低电平时，都可配置输入脉冲信号 sig_ch0_u*n* 在上升沿和下降沿的不同计数模式。可选计数模式如下：

- 递增模式：通道检测到 sig_ch0_u*n* 的有效边沿时，计数器的值 pulse_cnt 加 1。pulse_cnt 大于等于 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置位 PCNT_CNT_PAUSE_U*n*，则 pulse_cnt 计数模式改变，且不被清零。
- 递减模式：通道检测到 sig_ch0_u*n* 的有效边沿时，计数器的值 pulse_cnt 减 1。pulse_cnt 小于等于 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置位 PCNT_CNT_PAUSE_U*n* ，则 pulse_cnt 计数模式改变，且不被清零。
- 停止计数：计数停止，计数器的值 pulse_cnt 保持不变。

表 44 至表 47 说明了如何配置通道 0 的计数模式。

表 44: 控制信号为低电平时输入脉冲信号上升沿的计数模式

PCNT_CH0_POS_MODE_U*n*	PCNT_CH0_LCTRL_MODE_U*n*	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 45: 控制信号为高电平时输入脉冲信号上升沿的计数模式

PCNT_CH0_POS_MODE_U*n*	PCNT_CH0_HCTRL_MODE_U*n*	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 46: 控制信号为低电平时输入脉冲信号下降沿的计数模式

PCNT_CH0_NEG_MODE_U*n*	PCNT_CH0_LCTRL_MODE_U*n*	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 47: 控制信号为高电平时输入脉冲信号下降沿的计数模式

PCNT_CH0_NEG_MODE_U*n*	PCNT_CH0_HCTRL_MODE_U*n*	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 adder 模块。adder 模块是一个带符号位的 16 位加法器。软件可以通过置位 PCNT_CNT_PAUSE_U*n* 暂停 adder，也可以通过置位 PCNT_PULSE_CNT_RST_U*n* 清零 adder。

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_THRES0_U*n* 或者 PCNT_CNT_THRES1_U*n* 时，产生中断，同时 PCNT_CNT_THR_THRES0_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。

11.3 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 递减或递减、双通道一增一减），可参考这三种模式。

11.3.1 通道 0 独自递增计数

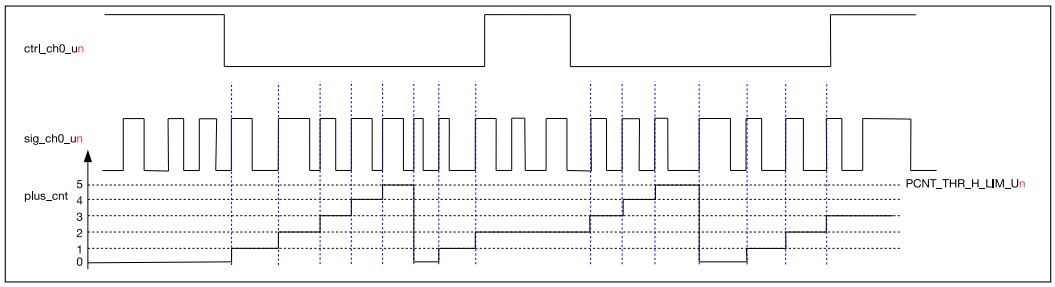


图 11-3. 通道 0 递增计数图

图 11-3 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 11.2 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。

11.3.2 通道 0 独自递减计数

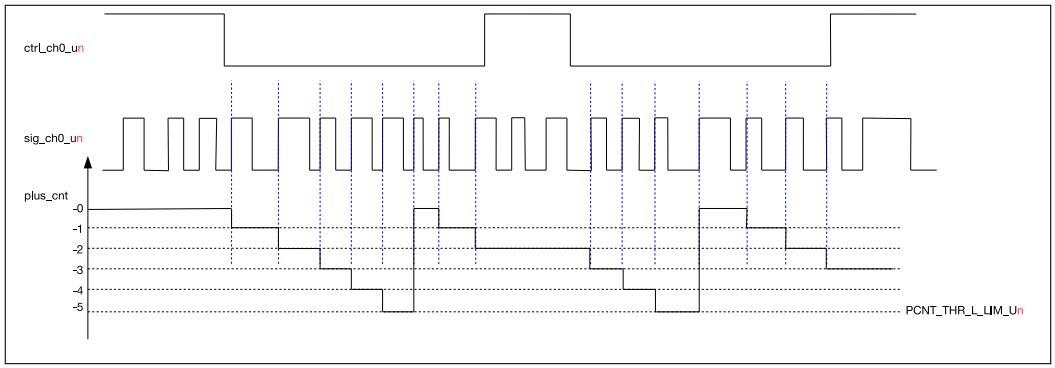


图 11-4. 通道 0 递减计数图

图 11-4 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 11-3 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。

11.3.3 通道 0 和通道 1 同时递增计数

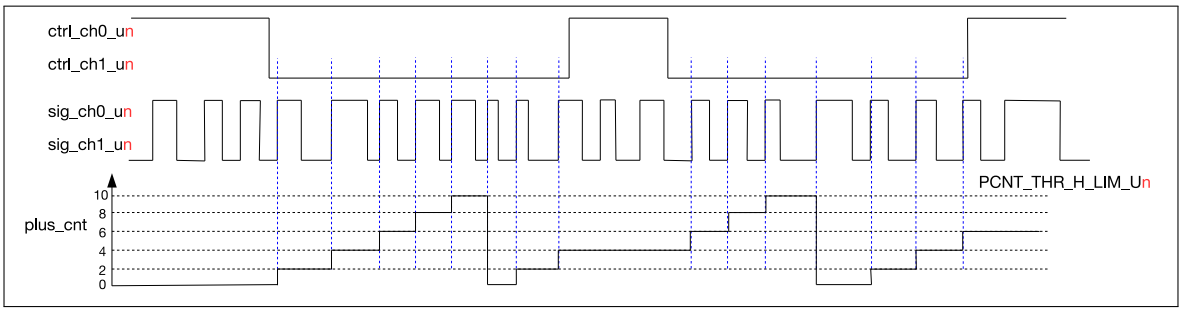


图 11-5. 双通道递增计数图

图 11-5 为通道 0 和通道 1 在 sig_ch0_u*n* 和 sig_ch1_u*n* 上升沿一同递增计数的示意图。如图 11-5 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。

11.4 基地址

用户可以通过两个不同的寄存器基地址访问 PCNT，如表 48 所示。更多信息，请访问章节 1 *系统和存储器*。

表 48: PCNT 基地址

访问总线	基地址
PeriBUS1	0x3F417000
PeriBUS2	0x60017000

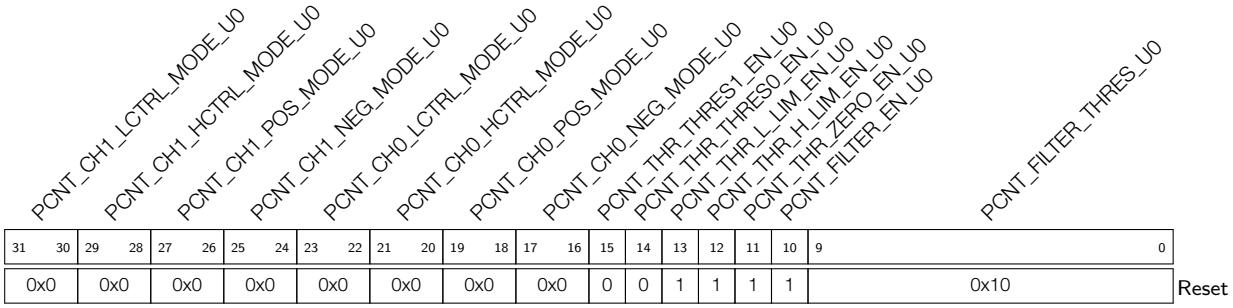
11.5 寄存器列表

请注意，下表中的地址都是相对于 PCNT 基地址的地址偏移量（相对地址）。更多有关 PCNT 基地址的信息，请前往 11.4 章节。

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

11.6 寄存器

Register 11.1: PCNT_U*n*_CONF0_REG (*n*: 0-3) (0x0000+12n*)**



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。(读/写)

PCNT_FILTER_EN_U*n* 单元 *n* 输入滤波器的使能位。(读/写)

PCNT_THR_ZERO_EN_U*n* 单元 *n* 过零比较器的使能位。(读/写)

PCNT_THR_H_LIM_EN_U*n* 单元 *n* 上限比较器的使能位。(读/写)

PCNT_THR_L_LIM_EN_U*n* 单元 *n* 下限比较器的使能位。(读/写)

PCNT_THR_THRES0_EN_U*n* 单元 *n* 阈值 0 比较器的使能位。(读/写)

PCNT_THR_THRES1_EN_U*n* 单元 *n* 阈值 1 比较器的使能位。(读/写)

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。(读/写)

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。(读/写)

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。(读/写)

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。(读/写)

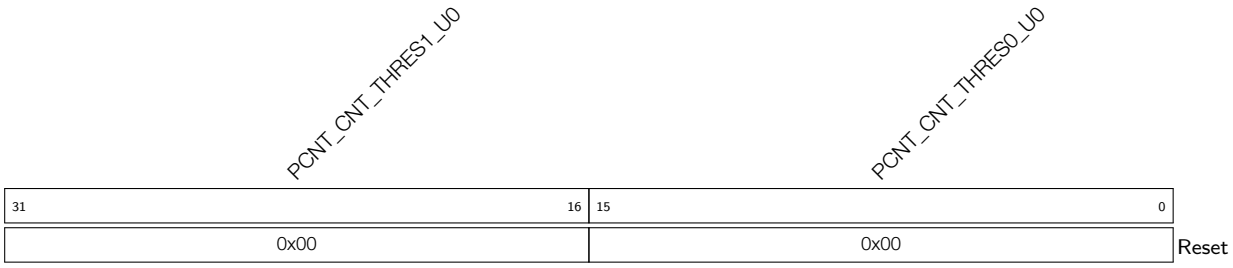
PCNT_CH1_NEG_MODE_U*n* 用于设置通道 1 输入信号检测下降沿的工作模式。1：计数器递增；2：计数器递减；0、3：对计数器无任何影响。(读/写)

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。(读/写)

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。(读/写)

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。(读/写)

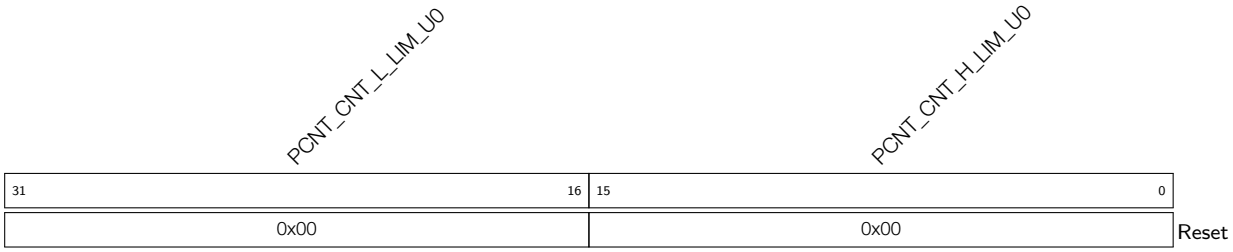
Register 11.2: PCNT_U*n*_CONF1_REG (*n*: 0-3) (0x0004+12n*)**



PCNT_CNT_THRES0_U*n* 用于配置单元 *n* 阈值 0 的值。(读/写)

PCNT_CNT_THRES1_U*n* 用于配置单元 *n* 阈值 1 的值。(读/写)

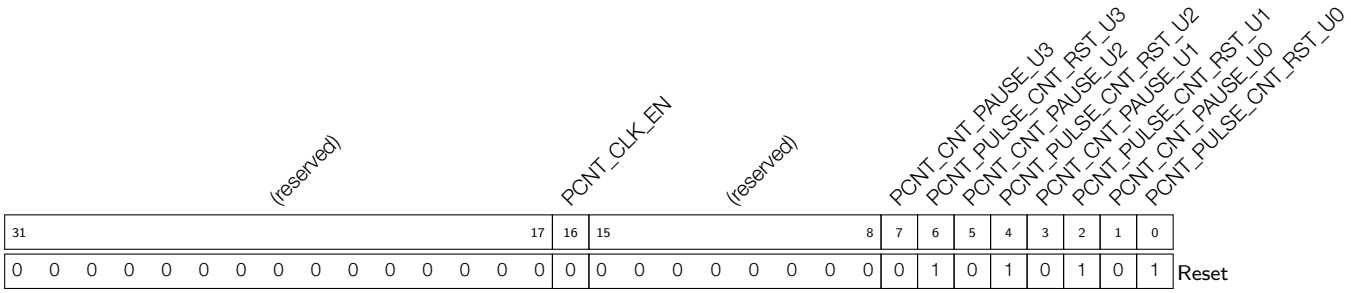
Register 11.3: PCNT_U*n*_CONF2_REG (*n*: 0-3) (0x0008+12n*)**



PCNT_CNT_H_LIM_U*n* 用于配置单元 *n* 的计数上限阈值。(读/写)

PCNT_CNT_L_LIM_U*n* 用于配置单元 *n* 的计数下限阈值。(读/写)

Register 11.4: PCNT_CTRL_REG (0x0060)

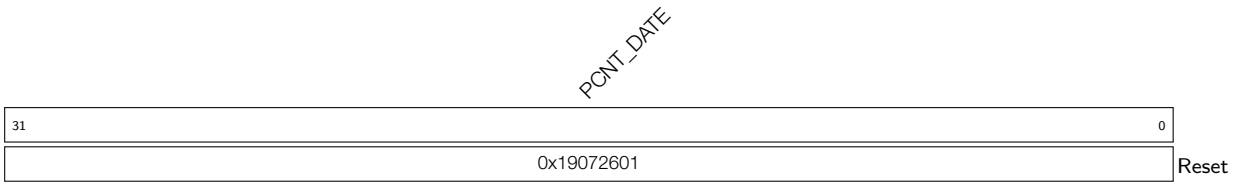


PCNT_PULSE_CNT_RST_U*n* 置位此位，清零单元 *n* 的计数器。(读/写)

PCNT_CNT_PAUSE_U*n* 置位此位，暂停单元 *n* 的计数器。(读/写)

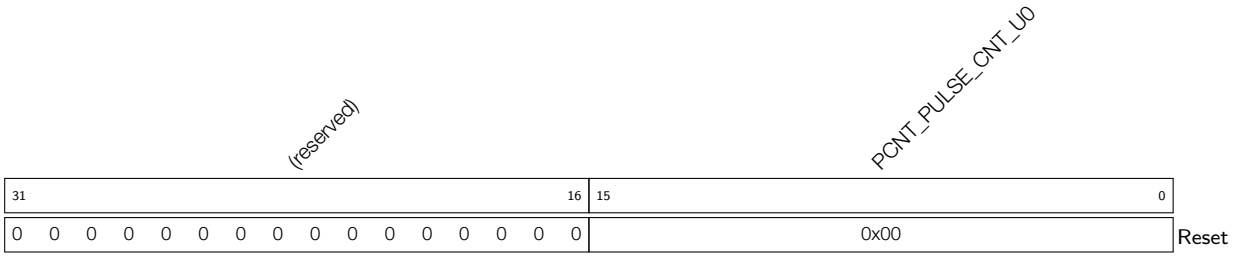
PCNT_CLK_EN 脉冲计数器模块寄存器时钟门控的使能信号 1：寄存器可通过应用读取、写值。0：寄存器无法通过应用读取、写值。(读/写)

Register 11.5: PCNT_DATE_REG (0x00FC)



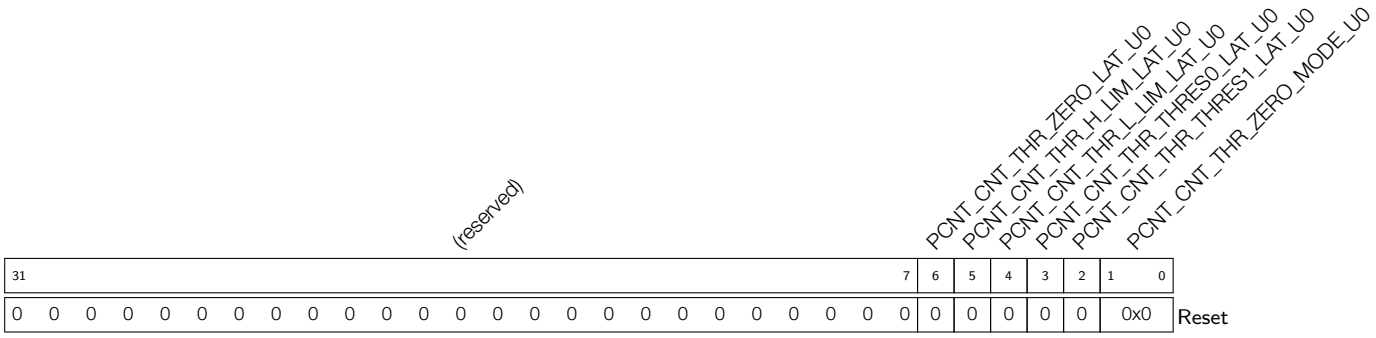
PCNT_DATE 脉冲计数器的版本控制寄存器。(读/写)

Register 11.6: PCNT_U*n*_CNT_REG (*n*: 0-3) (0x0030+4**n*)



PCNT_PULSE_CNT_U*n* 存储单元 *n* 脉冲计数器的当前值。(只读)

Register 11.7: PCNT_U*n*_STATUS_REG (*n*: 0-3) (0x0050+4*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。(只读)

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。(只读)

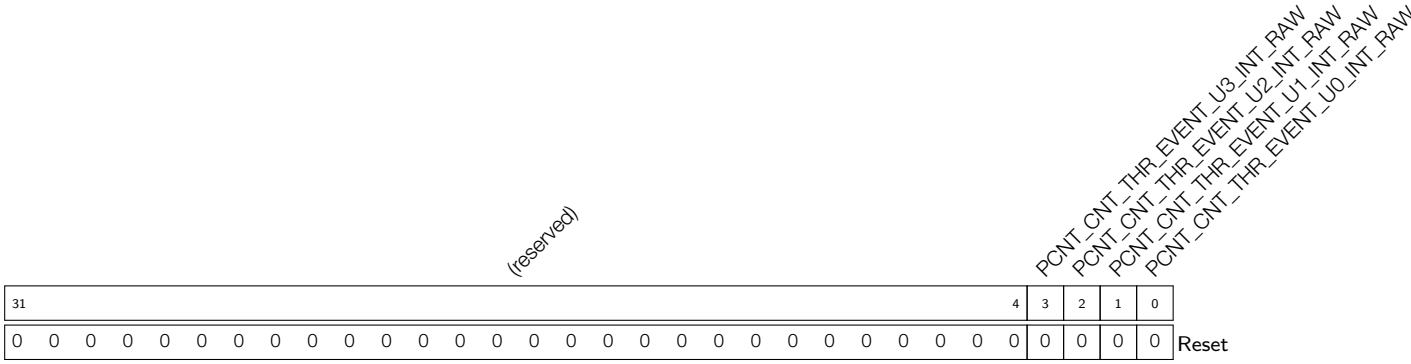
PCNT_CNT_THR_THRES0_LAT_U*n* 阈值中断有效时，PCNT_U*n* 阈值 0 的锁存值。1：脉冲计数器的当前值与阈值 0 相等，阈值 0 有效。0：其他。(只读)

PCNT_CNT_THR_L_LIM_LAT_U*n* 阈值中断有效时，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。(只读)

PCNT_CNT_THR_H_LIM_LAT_U*n* 阈值中断有效时，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。(只读)

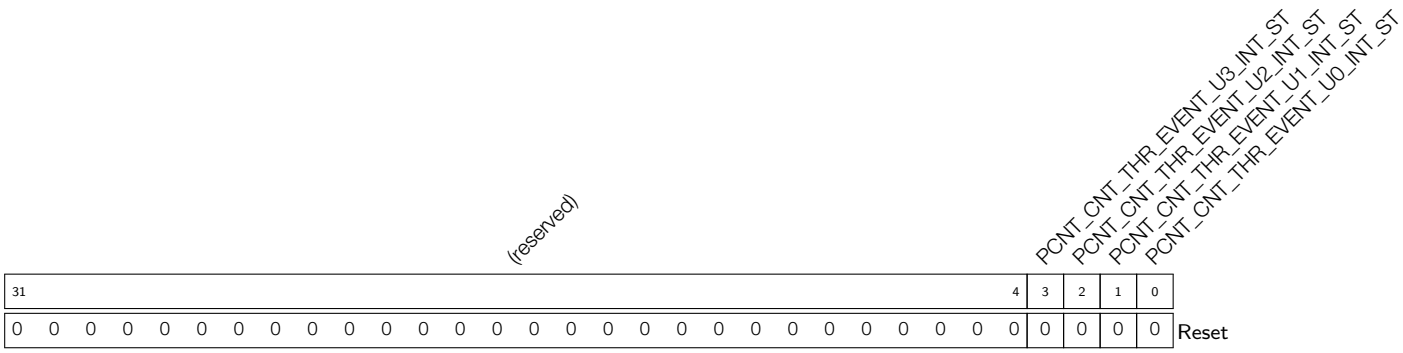
PCNT_CNT_THR_ZERO_LAT_U*n* 阈值中断有效时，PCNT_U*n* 阈值 0 的锁存值。1：脉冲计数器的当前值为 0，阈值 0 有效。0：其他。(只读)

Register 11.8: PCNT_INT_RAW_REG (0x0040)



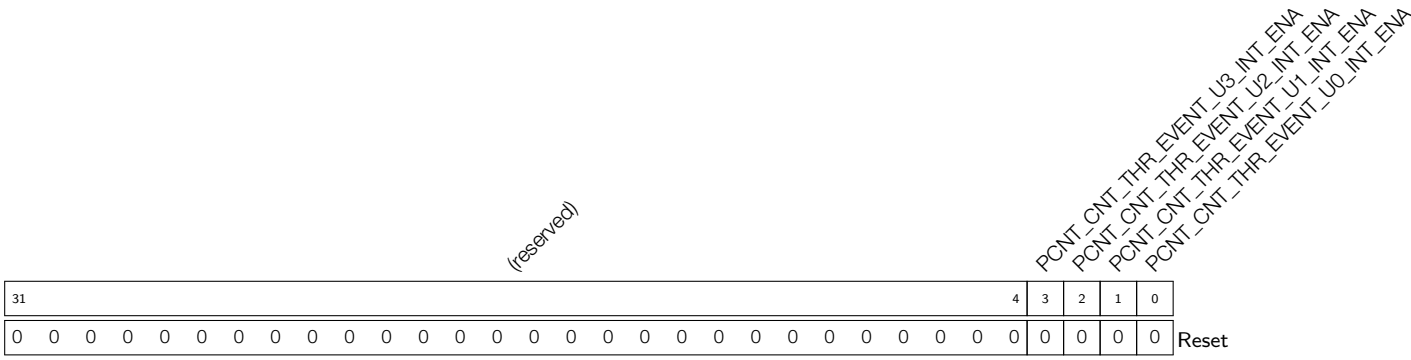
PCNT_CNT_THR_EVENT_U*n*_INT_RAW 单元 *n* 事件中断的原始中断状态位。(只读)

Register 11.9: PCNT_INT_ST_REG (0x0044)



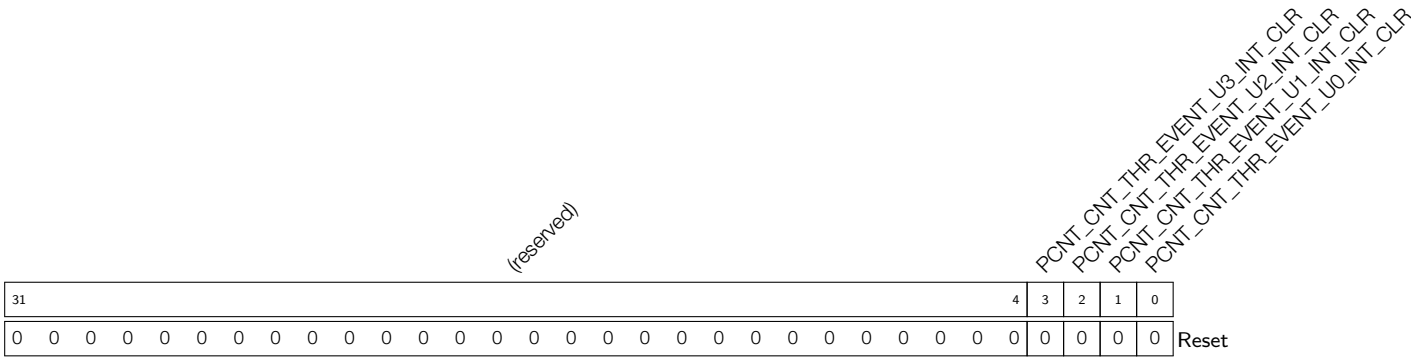
PCNT_CNT_THR_EVENT_U*n*_INT_ST 单元 *n* 事件中断的屏蔽中断状态位。(只读)

Register 11.10: 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。(读/写)

Register 11.11: 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。(只写)

12. 64 位定时器

12.1 概述

通用定时器可用于准确设定时间间隔、在一定间隔后触发中断（周期或非周期的）或充当硬件时钟。如图12-1所示，ESP32-S2 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 64 位可自动重新加载向上 / 向下计数器。

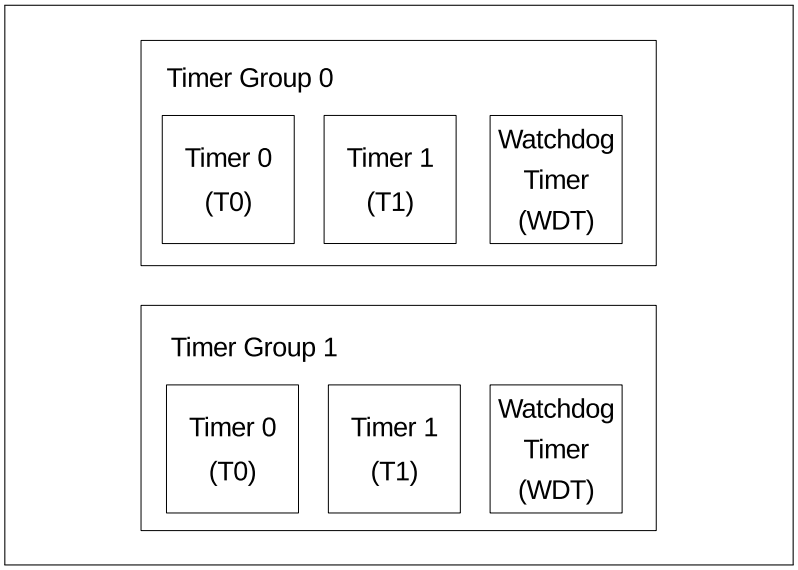


图 12-1. 定时器组

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。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 1-65536
- 64 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）
- 电平触发中断和边沿触发中断机制

12.2 功能描述

12.2.1 16 位预分频器与时钟选择

每个定时器可通过配置寄存器 TIMG_TxCONFIG_REG 的 TIMG_Tx_USE_XTAL 字段，选择 APB 时钟 (APB_CLK) 或外部时钟 (XTAL_CLK) 作为源时钟。源时钟经 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 位预分频器可通过 TIMG_Tx_DIVIDER 器字段配置，选取从 1 到 65536 之间的任意值。注意，将 TIMG_Tx_DIVIDER 置 0 后，分频系数会变为 65536。定时器必须关闭（即 TIMG_Tx_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 64 位时基计数器

64 位时基计数器基于 TB_CLK，可通过 TIMG_Tx_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_Tx_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_Tx_INCREASE 置位后，TIMG_Tx_EN 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 64 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）。在 TIMG_TxUPDATE_REG 上写任意值，64 位定时器的值可立即锁入寄存器 TIMG_TxLO_REG 和 TIMG_TxHI_REG ，两个寄存器分别锁存低 32 位和高 32 位。在 TIMG_TxUPDATE_REG 写入新值之前，寄存器 TIMG_TxLO_REG 和 TIMG_TxHI_REG 将保持不变，以便 CPU 读值。

12.2.3 报警产生

配置后，定时器的当前值与报警值相同时可触发报警。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）。

64 位报警值可在 TIMG_TxALARMLO_REG 和 TIMG_TxALARMHI_REG 配置，两者分别代表报警值的低 32 位和高 32 位。但是，只有置位 TIMG_Tx_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），定时器的当前值高于向上计数器或低于向下计数器的报警值时也会立即触发报警。

报警时，TIMG_Tx_ALARM_EN 字段自动清零，在置位 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 32 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 在寄存器上 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 中断

每个定时器都有一组连接至 CPU 的中断线（针对边沿中断和电平中断）。因此，每组定时器共有 6 个中断，命名如下：

- TIMG_WDT_LEVEL_INT：组内看门狗定时器的电平中断，在看门狗定时器中断阶段超时后产生
- TIMG_WDT_EDGE_INT：组内看门狗定时器的边沿中断，在看门狗定时器中断阶段超时后产生
- TIMG_T*x*_LEVEL_INT：通用定时器的电平中断，在报警时产生
- TIMG_T*x*_EDGE_INT：通用定时器的边沿中断，在报警时产生

中断在报警（或看门狗定时器阶段超时）时触发。报警（或阶段超时）后，电平中断线会被拉高，直至手动清除。边沿中断则会在报警（或阶段超时）后产生一个短脉冲。要使能定时器的电平中断或边沿中断，应分别置位 TIMG_T*x*_LEVEL_INT_EN 或 TIMG_T*x*_EDGE_INT_EN 位。

每个定时器组的中断由一组寄存器控制。组内的每个定时器在该组寄存器中都有对应的位：

- TIMG_T*x*_INT_RAW：报警时置 1。该位在写值到对应的 TIMG_T*x*_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_T*x*_INT_ST：反映每个定时器中断的状态，通过用 TIMG_T*x*_INT_ENA 屏蔽 TIMG_T*x*_INT_RAW 位来生成。对于电平中断而言，TIMG_T*x*_INT_ST 反映了 TIMG_T*x*_INT_RAW 的电平。
- 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。对于电平中断而言，TIMG_WDT_INT_ST 反映了 TIMG_WDT_INT_RAW 的电平。
- TIMG_T*x*_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_T*x*_INT_CLR：置 1 此位清除定时器中断，定时器对应的 TIMG_T*x*_INT_RAW 和 TIMG_T*x*_INT_ST 位会清零。注意，使用电平中断前，必须清除定时器中断。
- TIMG_WDT_INT_CLR：置 1 此位清除定时器中断，看门狗定时器对应的 TIMG_WDT_INT_RAW 和 TIMG_WDT_INT_ST 位会清零。注意，使用电平中断前，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 TIMG_T*x*_USE_XTAL 字段选择源时钟。
    - 置位 TIMG_T*x*_DIVIDER 配置 16 位预分频器。
    - 置位或清除 TIMG_T*x*_INCREASE 配置定时器方向。
    - 在 TIMG_T*x*_LOAD_LO 和 TIMG_T*x*_LOAD_HI 上写初始值设置定时器的初始值，然后在 TIMG_T*x*LOAD_REG 上写任意值将初始值重新加载进定时器。
2. 置位 TIMG_T*x*_EN 开启定时器。
3. 获得定时器的当前值。
    - 在 TIMG_T*x*UPDATE_REG 上写任意值锁存定时器的当前值。
    - 从 TIMG_T*x*LO_REG 和 TIMG_T*x*HI_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_LEVEL_INT_EN 或 TIMG_Tx_EDGE_INT_EN 分别使能电平中断和边沿中断。
3. 清零 TIMG_Tx_AUTORELOAD 关闭自动重新加载。
4. 置位 TIMG_Tx_EN 开启定时器。
5. 处理报警中断。
   - 置位定时器对应的 TIMG_Tx_INT_CLR 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_EN 开启定时器。
5. 处理报警中断（每次报警时重复）。
   - 置位定时器对应的 TIMG_Tx_INT_CLR 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。
   - 置位定时器对应的 TIMG_Tx_INT_CLR 位清除中断。
   - 清零 TIMG_Tx_EN 关闭定时器。

12.4 基地址

用户可通过四个不同的寄存器基地址访问 64 位定时器，如表 50 所示。更多信息，请访问章节 1 *系统和存储器*。

表 50: 64 位定时器基地址

模块名	访问总线	基地址
TIMG0	PeriBUS1	0x3F41F000
	PeriBUS2	0x6001F000
TIMG1	PeriBUS1	0x3F420000
	PeriBUS2	0x60020000

12.5 寄存器列表

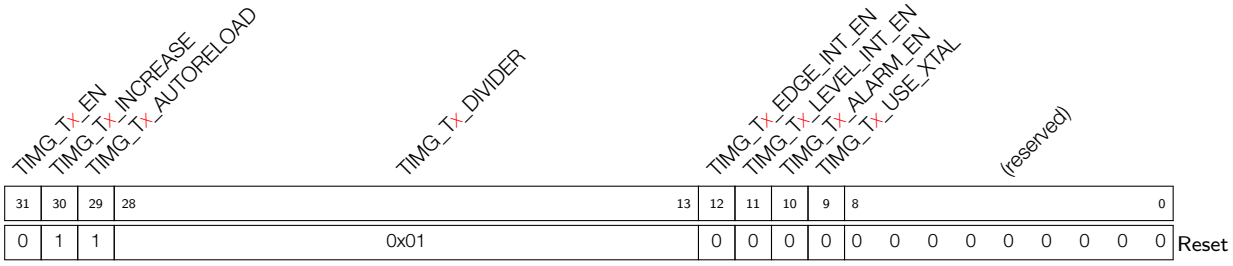
请注意，下表的地址指相对于 64 位定时器基地址的偏移量（相对地址）。请参阅第 12.4 节获取有关 64 位定时器基地址的信息。

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	读/写
TIMG_T0LO_REG	定时器 0 的当前值，低 32 位	0x0004	只读
TIMG_T0HI_REG	定时器 0 的当前值，高 32 位	0x0008	只读
TIMG_T0UPDATE_REG	将当前定时器的值写到 TIMG_T0LO_REG 和 TIMG_T0LO_REG	0x000C	读/写
TIMG_T0ALARMLO_REG	定时器 0 的报警值，低 32 位	0x0010	读/写
TIMG_T0ALARMHI_REG	定时器 0 的报警值，高位	0x0014	读/写
TIMG_T0LOADLO_REG	定时器 0 的重新加载值，低 32 位	0x0018	读/写
TIMG_T0LOADHI_REG	定时器 0 的重新加载值，高 32 位	0x001C	读/写
TIMG_T0LOAD_REG	在 TIMG_T0LOADLO_REG 和 TIMG_T0LOADHI_REG 上写值重新加载定时器	0x0020	只写
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	读/写
TIMG_T1LO_REG	定时器 1 的当前值，低 32 位	0x0028	只读
TIMG_T1HI_REG	定时器 1 的当前值，高 32 位	0x002C	只读
TIMG_T1UPDATE_REG	将当前定时器的值写到 TIMG_T1LO_REG 和 TIMG_T1LO_REG	0x0030	读/写
TIMG_T1ALARMLO_REG	定时器 1 的报警值，低 32 位	0x0034	读/写
TIMG_T1ALARMHI_REG	定时器 1 的报警值，高位	0x0038	读/写
TIMG_T1LOADLO_REG	定时器 1 的重新加载值，低 32 位	0x003C	读/写
TIMG_T1LOADHI_REG	定时器 1 的重新加载值，高 32 位	0x0040	读/写
TIMG_T1LOAD_REG	在 TIMG_T0LOADLO_REG 和 TIMG_T0LOADHI_REG 上写值重新加载定时器	0x0044	只写
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	读/写
TIMG_WDTCONFIG1_REG	看门狗定时器预分频器寄存器	0x004C	读/写
TIMG_WDTCONFIG2_REG	看门狗定时器阶段 0 超时值	0x0050	读/写
TIMG_WDTCONFIG3_REG	看门狗定时器阶段 1 超时值	0x0054	读/写
TIMG_WDTCONFIG4_REG	看门狗定时器阶段 2 超时值	0x0058	读/写

名称	描述	地址	访问
TIMG_WDTCONFIG5_REG	看门狗定时器阶段 3 超时值	0x005C	读/写
TIMG_WDTFEED_REG	写值驱动看门狗定时器	0x0060	只写
TIMG_WDTWPROTECT_REG	看门狗写保护寄存器	0x0064	读/写
RTC CALI 配置和控制寄存器			
TIMG_RTCCALICFG2_REG	定时器组校准寄存器	0x00A8	不定
中断寄存器			
TIMG_INT_ENA_TIMERS_REG	中断使能位	0x0098	读/写
TIMG_INT_RAW_TIMERS_REG	原始中断状态	0x009C	只读
TIMG_INT_ST_TIMERS_REG	屏蔽中断状态	0x00A0	只读
TIMG_INT_CLR_TIMERS_REG	中断清除位	0x00A4	只写
版本寄存器			
TIMG_TIMERS_DATE_REG	版本控制寄存器	0x00F8	读/写
配置寄存器			
TIMG_REGCLK_REG	定时器组时钟门控寄存器	0x00FC	读/写

12.6 寄存器

Register 12.1: TIMG_TxCONFIG_REG (x: 0-1) (0x0000+36*x)



TIMG_Tx_USE_XTAL 1：使用 XTAL_CLK 作为定时器组的源时钟。0：使用 APB_CLK 作为定时器组的源时钟。(读/写)

TIMG_Tx_ALARM_EN 置 1 后，报警使能。报警使能后，此位自动清零。(读/写)

TIMG_Tx_LEVEL_INT_EN 置 1 后，报警触发电平中断。(读/写)

TIMG_Tx_EDGE_INT_EN 置 1 后，报警触发边沿中断。(读/写)

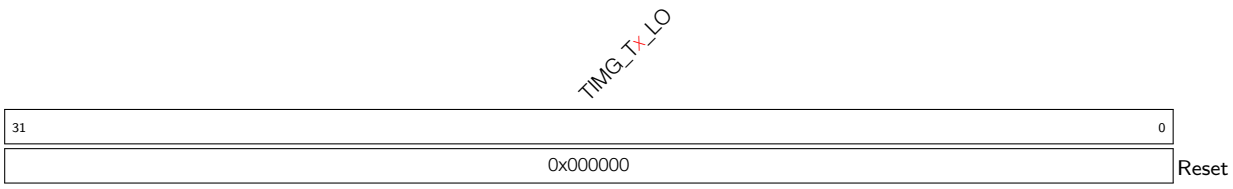
TIMG_Tx_DIVIDER 定时器 x 时钟 (Tx_clk) 的预分频器值。(读/写)

TIMG_Tx_AUTORELOAD 置 1 后，定时器 x 报警时自动重新加载使能。(读/写)

TIMG_Tx_INCREASE 置 1 后，定时器 x 的时基计数器在每个时钟周期递增计数。清零后，定时器 x 的时基计数器递减计数。(读/写)

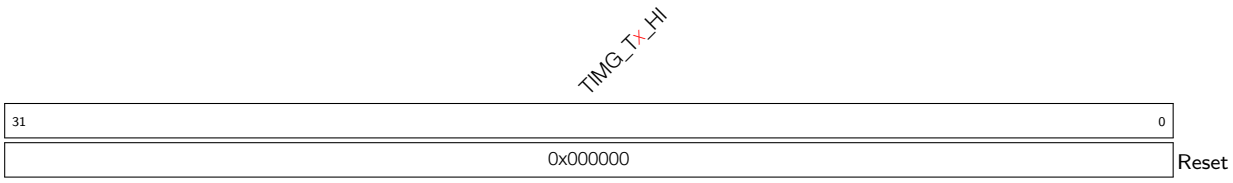
TIMG_Tx_EN 置 1 后，定时器 x 时基计数器使能。(读/写)

Register 12.2: TIMG_TxLO_REG (x: 0-1) (0x0004+36*x)



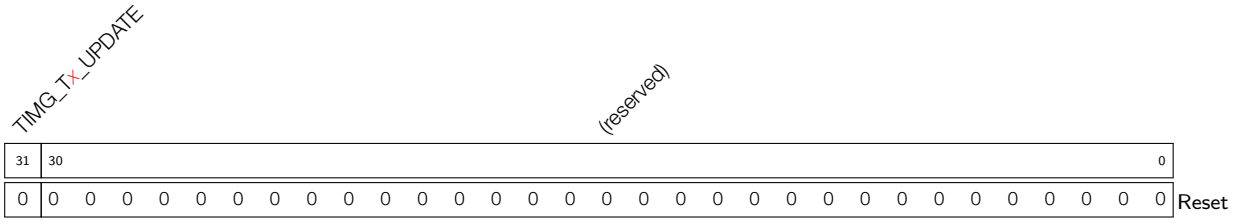
TIMG_Tx_LO 在 TIMG_TxUPDATE_REG 上写值后，可读取定时器 x 时基计数器的低 32 位。(只读)

Register 12.3: TIMG_TxHI_REG (x: 0-1) (0x0008+36*x)



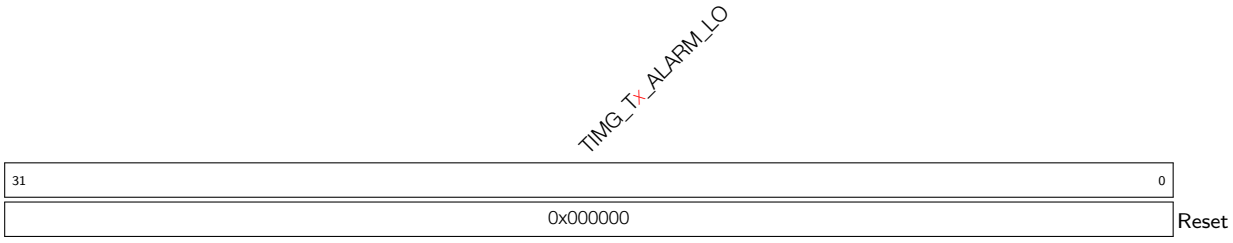
TIMG_Tx_HI 在 TIMG_TxUPDATE_REG 上写值后，可读取定时器 x 时基计数器的高 32 位。(只读)

Register 12.4: TIMG_TxUPDATE_REG (x: 0-1) (0x000C+36*x)



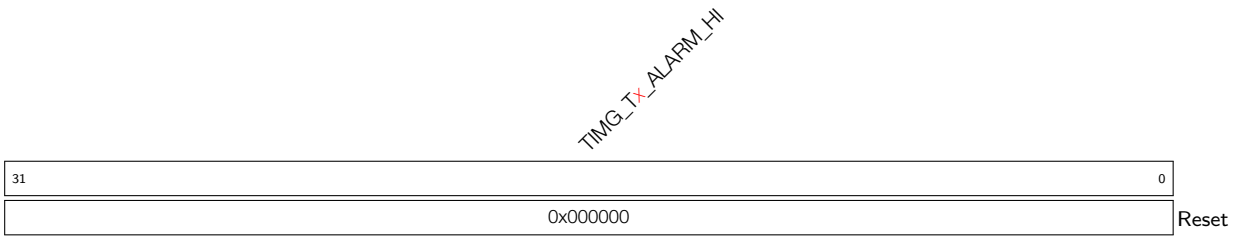
TIMG_Tx_UPDATE 在 TIMG_TxUPDATE_REG 上写 0 或 1，计数器的值被锁存 (读/写)

Register 12.5: TIMG_TxALARMLO_REG (x: 0-1) (0x0010+36*x)



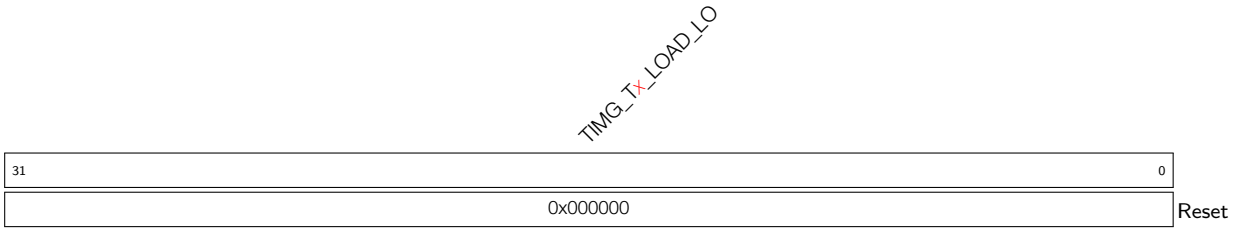
TIMG_Tx_ALARM_LO 定时器 x 时基计数器低 32 位的警报触发值。(读/写)

Register 12.6: TIMG_TxALARMHI_REG (x: 0-1) (0x0014+36*x)



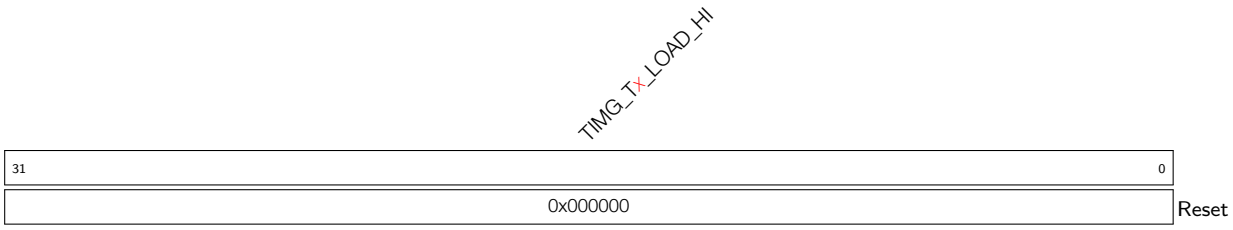
TIMG_Tx_ALARM_HI 定时器 x 时基计数器高 32 位的警报触发值。(读/写)

Register 12.7: TIMG_TxLOADLO_REG (x: 0-1) (0x0018+36*x)



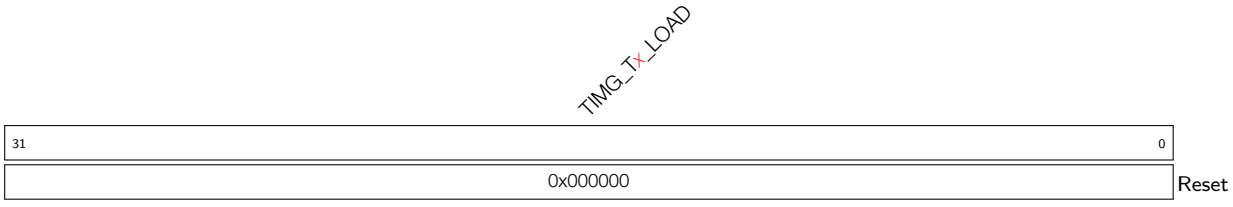
TIMG_Tx_LOAD_LO 定时器 x 时基计数器低 32 位的重新加载值。(读/写)

Register 12.8: TIMG_TxLOADHI_REG (x: 0-1) (0x001C+36*x)



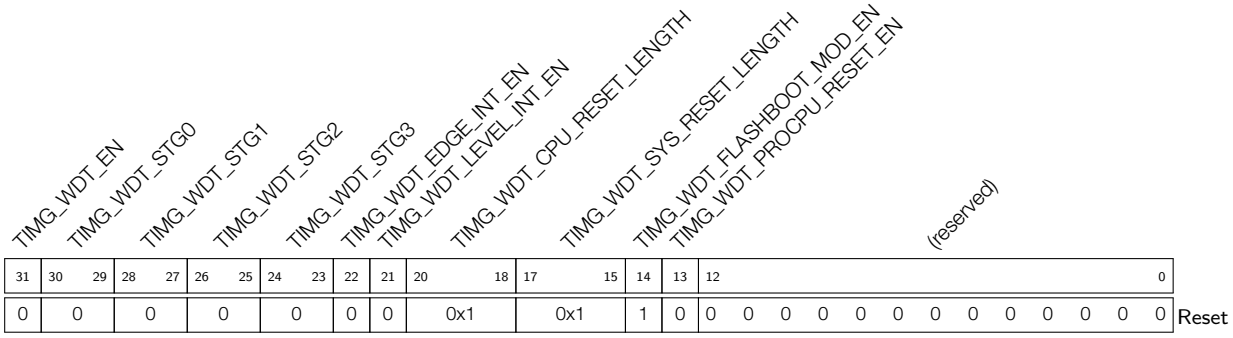
TIMG_Tx_LOAD_HI 定时器 x 时基计数器高 32 位的重新加载值。(读/写)

Register 12.9: TIMG_TxLOAD_REG (x: 0-1) (0x0020+36*x)



TIMG_Tx_LOAD 写任意值重新加载定时器 x 时基计数器。(只写)

Register 12.10: TIMG_WDTCONFIG0_REG (0x0048)



TIMG_WDT_PROCPU_RESET_EN WDT 复位 CPU 使能。(读/写)

TIMG_WDT_FLASHBOOT_MOD_EN 置 1 后, flash 启动保护使能。(读/写)

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 us，7：3.2 us。(读/写)

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 us，7：3.2 us。(读/写)

TIMG_WDT_LEVEL_INT_EN 置 1 后, 如超过设置的阶段中断产生时间，会产生电平中断。(读/写)

TIMG_WDT_EDGE_INT_EN 置 1 后, 如超过设置的阶段中断产生时间，会产生边沿中断。(读/写)

TIMG_WDT_STG3 阶段 3 配置。0：关闭，1：中断，2：复位 CPU，3：复位系统。(读/写)

TIMG_WDT_STG2 阶段 2 配置。0：关闭，1：中断，2：复位 CPU，3：复位系统。(读/写)

TIMG_WDT_STG1 阶段 1 配置。0：关闭，1：中断，2：复位 CPU，3：复位系统。(读/写)

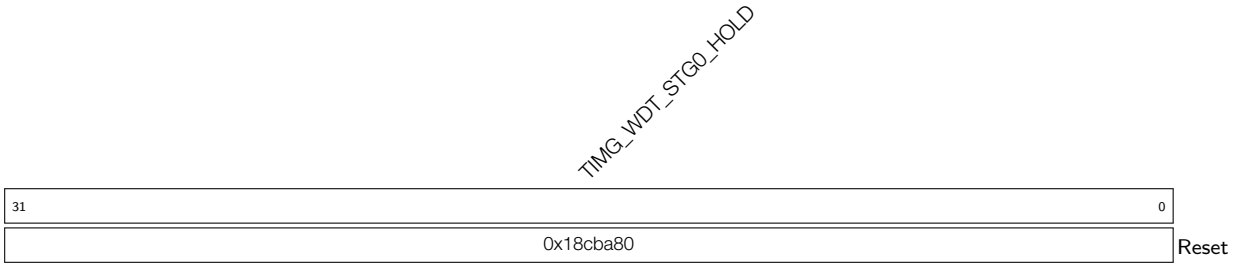
TIMG_WDT_STG0 阶段 0 配置。0：关闭，1：中断，2：复位 CPU，3：复位系统。(读/写)

TIMG_WDT_EN 置 1 后，MWDT 使能。(读/写)

Register 12.11: TIMG_WDTCONFIG1_REG (0x004C)

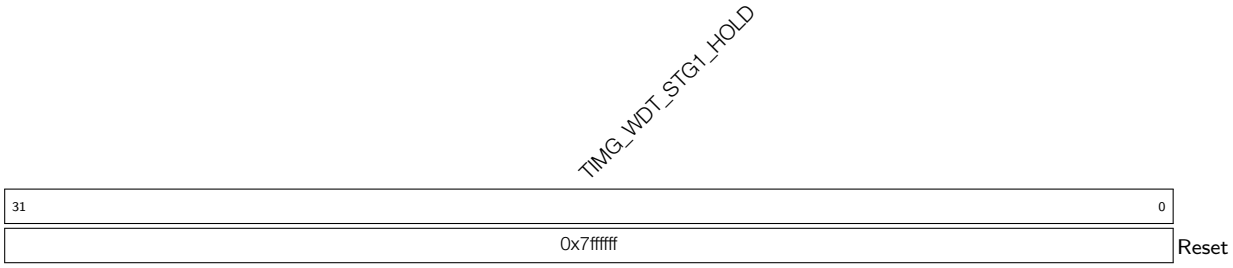
TIMG_WDT_CLK_PRESCALER MWDT 时钟预分频器值。MWDT 时钟长度 = 12.5 ns * TIMGn_WDT_CLK_PRESCALE。(读/写)

Register 12.12: TIMG_WDTCONFIG2_REG (0x0050)



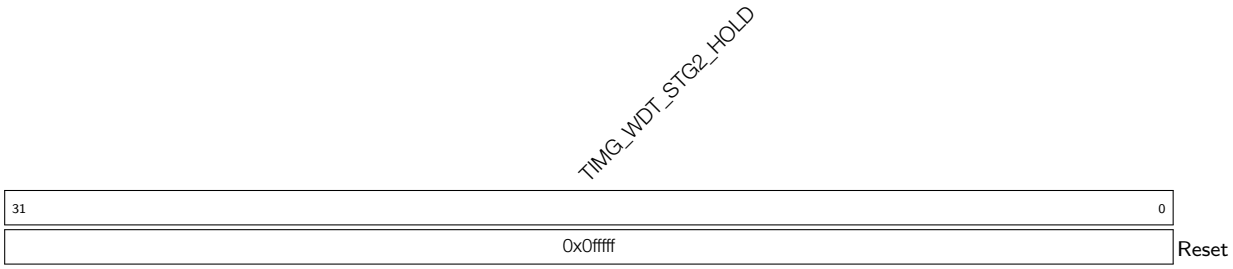
TIMG_WDT_STG0_HOLD MWDT 时钟周期中阶段 0 超时时间。(读/写)

Register 12.13: TIMG_WDTCONFIG3_REG (0x0054)



TIMG_WDT_STG1_HOLD MWDT 时钟周期中阶段 1 超时时间。(读/写)

Register 12.14: TIMG_WDTCONFIG4_REG (0x0058)

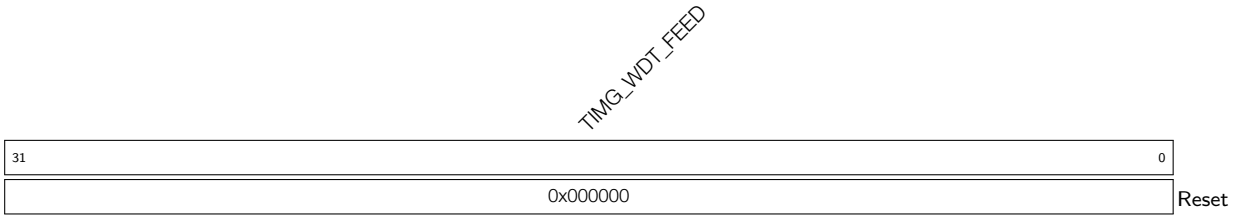


TIMG_WDT_STG2_HOLD MWDT 时钟周期中阶段 2 超时时间。(读/写)

Register 12.15: TIMG_WDTCONFIG5_REG (0x005C)

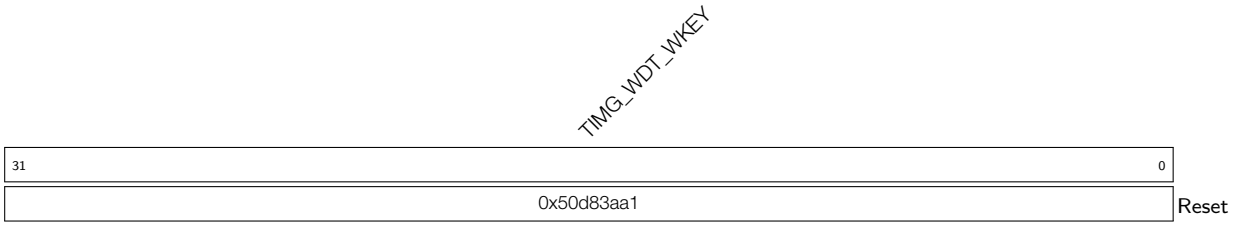
TIMG_WDT_STG3_HOLD MWDT 时钟周期中阶段 3 超时时间。(读/写)

Register 12.16: TIMG_WDTFEED_REG (0x0060)



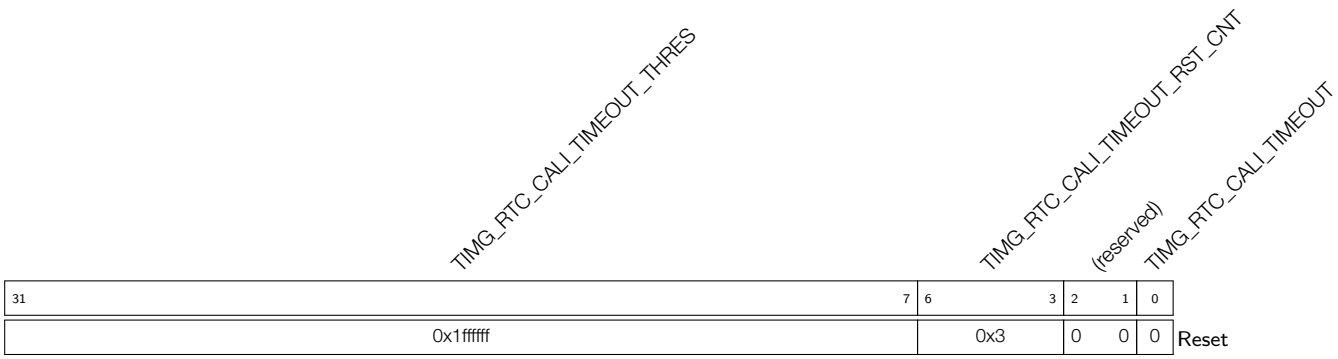
TIMG_WDT_FEED 写任意值驱动 MWDT。(只写)

Register 12.17: TIMG_WDTWPROTECT_REG (0x0064)



TIMG_WDT_WKEY 如果寄存器中有和复位值不同的值，写保护使能。(读/写)

Register 12.18: TIMG_RTCCALICFG2_REG (0x00A8)

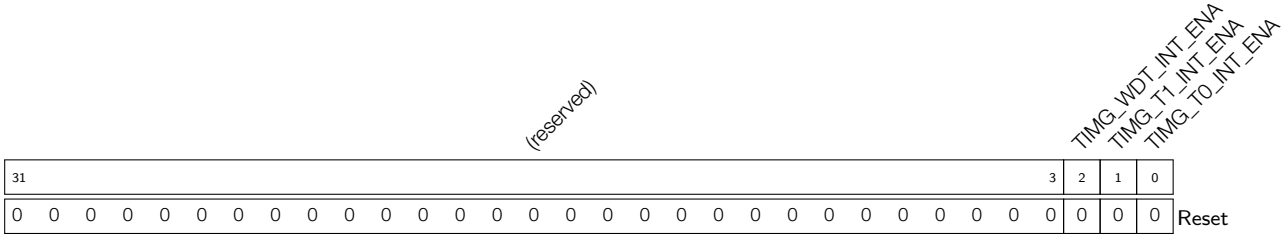


TIMG_RTC_CALI_TIMEOUT RTC 校准超时指示器 (只读)

TIMG_RTC_CALI_TIMEOUT_RST_CNT 校准超时复位周期 (读/写)

TIMG_RTC_CALI_TIMEOUT_THRES RTC 校准定时器的界限值。校准定时器的值超过此界限值时触发超时。(读/写)

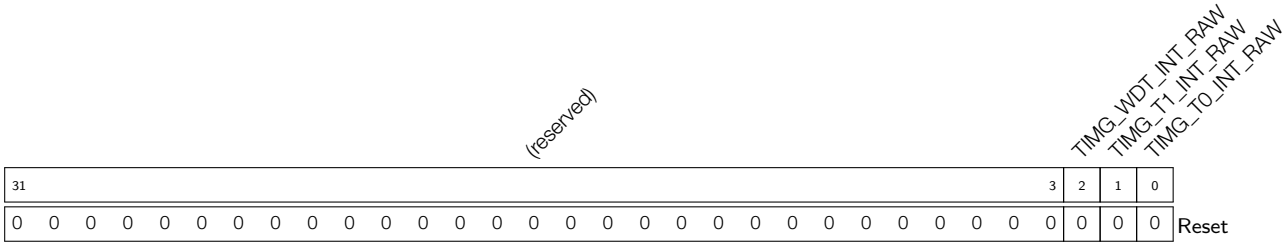
Register 12.19: TIMG_INT_ENA_TIMERS_REG (0x0098)



TIMG_T*x*_INT_ENA TIMG_T*x*_INT 中断的中断使能位 (读/写)

TIMG_WDT_INT_ENA TIMG_WDT_INT 中断的中断使能位 (读/写)

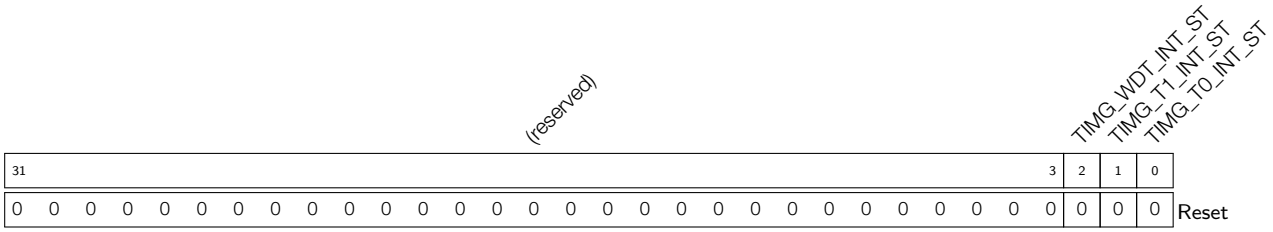
Register 12.20: TIMG_INT_RAW_TIMERS_REG (0x009C)



TIMG_T*x*_INT_RAW TIMG_T*x*_INT 中断的原始中断状态位。(只读)

TIMG_WDT_INT_RAW TIMG_WDT_INT 中断的原始中断状态位。(只读)

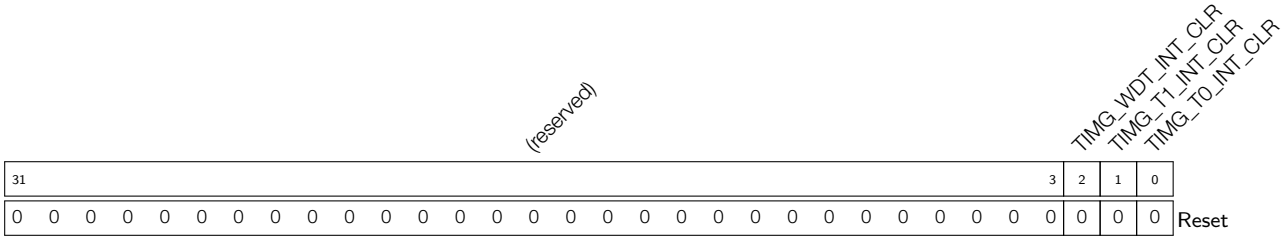
Register 12.21: TIMG_INT_ST_TIMERS_REG (0x00A0)



TIMG_T*x*_INT_ST TIMG_T*x*_INT 中断的屏蔽中断状态位。(只读)

TIMG_WDT_INT_ST TIMG_WDT_INT 中断的屏蔽中断状态位。(只读)

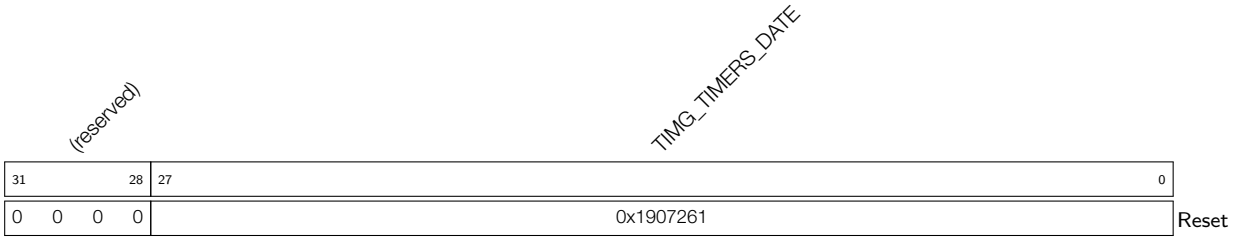
Register 12.22: TIMG_INT_CLR_TIMERS_REG (0x00A4)



TIMG_T*x*_INT_CLR 置位此位，清除 TIMG_T*x*_INT 中断。(只写)

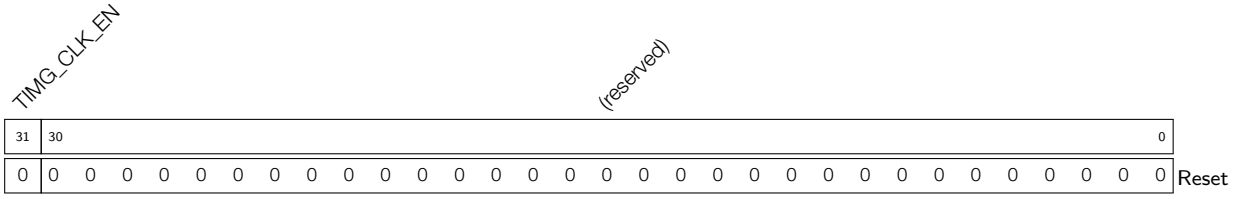
TIMG_WDT_INT_CLR 置位此位，清除 TIMG_WDT_INT 中断。(只写)

Register 12.23: TIMG_TIMERS_DATE_REG (0x00F8)



TIMG_TIMERS_DATE 版本控制寄存器。(读/写)

Register 12.24: TIMG_REGCLK_REG (0x00FC)



TIMG_CLK_EN 寄存器时钟门控信号。1：可以通过软件读取、写值的寄存器 0：无法通过软件读取、写值的寄存器 (读/写)

13. 看门狗定时器

13.1 概述

系统/软件若出现不可预知的问题（比如软件卡在某个循环中或事件逾期）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/软件的错误行为。

ESP32-S2 中有三个看门狗定时器：两个定时器组中各一个（称作主系统看门狗定时器，缩写为 MWDT），RTC 模块中一个（称作 RTC 看门狗定时器，缩写为 RWDT）。看门狗在运行期间会经历四个阶段（除非看门狗被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，其中除了 RWDT 支持四种超时动作外，其它两个看门狗仅支持三种。超时动作包括：中断、CPU 复位、内核复位和系统复位。其中，只有 RWDT 能够触发系统复位，即复位芯片内部所有的数字电路，包括 RTC 和主系统。每个阶段的超时时间都可单独设置。

在引导加载 flash 固件期间，RWDT 和第一个 MWDT 会自动使能，以检测引导过程中发生的错误，并恢复运行。

请注意，本章节仅包含看门狗定时器的功能描述，其寄存器部分详见第 12 章：*64 位定时器*。

13.2 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，则会采取三或四种（分别针对 MWDT 和 RWDT）动作中的一种（中断、CPU 复位、内核复位和系统复位）。
- 保护 32 位超时计数器，防止 RWDT 和 MWDT 的配置被无意间更改。
- 写保护，防止 RWDT 和 MWDT 配置无意间改动
- 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。

13.3 功能描述

13.3.1 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。而 RWDT 的时钟源则直接取自于 RTC 慢速时钟（没有预分频器），频率通常为 32 kHz。MWDT 的 16 位预分频器可通过 TIMG_WDTCONFIG1_REG 寄存器的 TIMG_WDT_CLK_PRESCALER 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。

13.3.2 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在 TIMG_WDTCONFIG*i*_REG （i 的范围是 2 到 5）寄存器中配置，RWDT 的超时时间可在 RTC_CNTL_WDT_STGG*j*_HOLD_REG（j 的范围是 0 到 3）寄存器中配置。值得注意的是，RWDT 在阶段 0 的超时时间 (T_{hold0}) 受 eFuse 寄存器 EFUSE_WDT_DELAY_SEL 字段和 RTC_CTRL_WDT_STG0_HOLD_REG 寄存器共同影响，关系如下：

T_{hold0} = RTC_CTRL_WDT_STG0_HOLD_REG « （EFUSE_WDT_DELAY_SEL + 1）

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。
- 复位 CPU 内核
  如阶段超时，复位 CPU 内核。
- 复位主系统
  如阶段超时，包括 MWDT 在内的主系统都会复位。RTC 不会复位。
- 复位主系统和 RTC
  如阶段超时，主系统和 RTC 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。与之类似，RWDT 的超时动作可在 RTC_WDTCONFIG0 寄存器配置。

13.3.3 写保护

看门狗定时器对于检测和处理系统/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，防止看门狗因偶然的错误访问而被关闭或篡改。

每个看门狗定时器都有一个写密钥寄存器，运行写保护机制（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保护寄存器写入 0x50D83AA1，才能修改其它看门狗寄存器。如果写密钥保护寄存器的值不是 0x50D83AA1，任何试图在看门狗定时器寄存器（除了密钥保护寄存器本身）上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保护寄存器，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保护寄存器上写入除 0x50D83AA1 以外的任意值，重新使能写保护。

13.3.4 Flash 引导保护

在 flash 引导过程中，定时器组 0（TIMG0）中的 MWDT 和 RWDT 会自动使能。MWDT 使能后，阶段 0 的默认超时动作为系统复位。RWDT 的阶段 0 超时动作为主系统和 RTC 复位。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。

13.4 寄存器

MWDT “寄存器” 是定时器模块的一部分，在第 12 章：*64 位定时器*定时器寄存器中有详细描述。

14. XTAL32K 看门狗

14.1 概述

ESP32 S2 的 XTAL32K 看门狗是用于检测 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 的慢速时钟源，若 XTAL32K_CLK 时钟停振，XTAL32K 看门狗会将 RTC 的慢速时钟替换为 RTC_CLK 的分频时钟并发送中断 (若芯片处于 deep sleep 状态则唤醒 CPU)，由软件重启 XTAL32K_CLK，并切回。

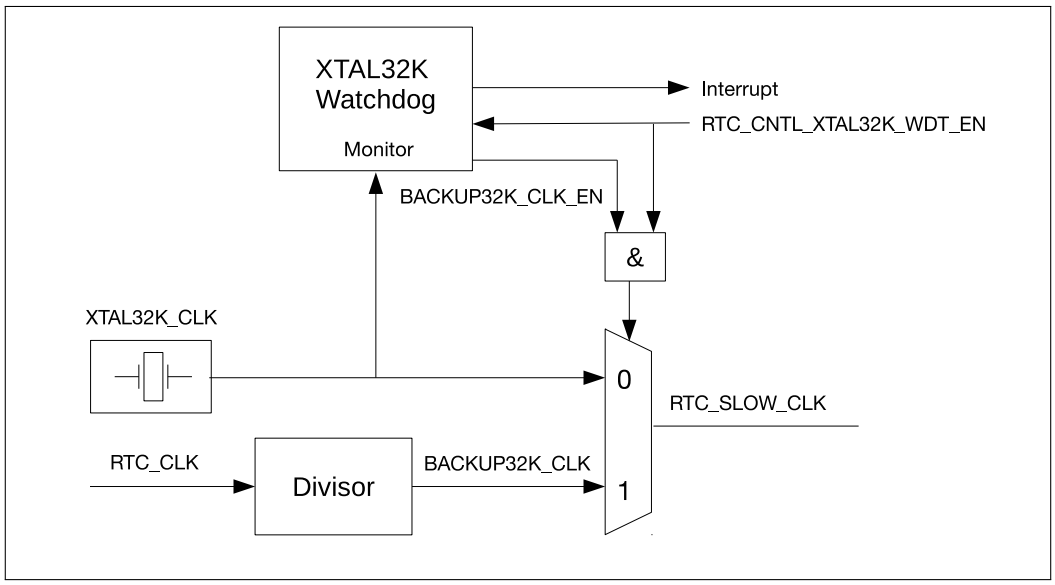


图 14-1. XTAL32K 看门狗

14.2 主要特性

14.2.1 XTAL32K 看门狗的中断及唤醒

XTAL32K 看门狗监控到 XTAL32K_CLK 停振时，将发起停振中断，如果 CPU 处于休眠状态，将唤醒 CPU。

14.2.2 BACKUP32K_CLK

XTAL32K 看门狗监控到 XTAL32K_CLK 停振后，将使用 RTC_CLK 的分频时钟 BACKUP32K_CLK (频率为 32KHz) 替代 XTAL32K_CLK 作为 RTC 的慢速时钟维持系统继续正常工作。

14.3 功能描述

14.3.1 工作流程

1. 使能 RTC_CNTL_XTAL_32K_WDT_EN，XTAL32K 看门狗将由空闲状态转入计数状态，计数器收到 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. XTAL32K 看门狗自动开启 BACKUP32K_CLK，替换 RTC 的慢速时钟源，保证系统能够正常运行，以及工作在 RTC 慢速时钟的定时器 (如 RTC_TIMER 等) 能够保持计时准确性。时钟频率的配置参见 14.3.2 *BACKUP32K_CLK 的分频系数配置方法*。

3. 软件通过 RTC_CNTL_XPD_XTAL_32K 位开关 XTAL32K_CLK 的 XPD 信号来重启 XTAL32K_CLK，然后通过将 RTC_CNTL_XTAL32K_WDT_EN 位置 0 把 RTC 的慢速时钟源从 BACKUP32K_CLK 切回到 XTAL32K_CLK。若是芯片处于 deep sleep 状态，则 XTAL32K 看门狗将唤醒 CPU，完成上述操作。

14.3.2 BACKUP32K_CLK 的分频系数配置方法

由于 RTC_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 慢速时钟工作的定时器依然能够准确计时，需要根据 RTC_CLK 的实际频率，配置 BACKUP32K_CLK 的分频系数。

设 RTC_CLK 的频率为 f_rtc_clk （单位：kHz），8 个分频因子为 x_0 x_1 x_2 x_3 x_4 x_5 x_6 和 x_7。
$S = x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$。

需满足以下条件：

$$S = f\_rtc\_clk \times (4/32)$$
$$M + 1 \geq x_n \geq M (0 \leq n \leq 7)$$
$$M = f\_rtc\_clk/32/2$$

x_n 为整数。M 和 S 四舍五入取整。分频因子 x_0 ~x_7 的宽度为 4 位，依次对应寄存器 RTC_CNTL_XTAL32K_CLK_FACTOR 的 32 位数据。

例如，RTC_CLK 的时钟频率为 163 kHz，则 $f\_rtc\_clk = 163$ $S = 20, M = 2$，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。

15. 系统定时器

15.1 概述

ESP32-S2 芯片内置一个 64 位系统定时器。该系统定时器专门用于操作系统，可生成周期性系统滴答或延时中断用于调度系统任务。在 RTC 定时器的协助下，系统定时器可在芯片从 Deep-sleep 或 Light-sleep 唤醒后及时更新。

15.2 主要特征

- 采用 64 位定时器。
- 使用 APB_CLK 时钟。
- 每个 APB_CLK 周期，定时器数值的递增幅度可配置。
- 如果 APB_CLK 时钟源发生切换，包括从 PLL_CLK 切换至 XTAL_CLK，或从 XTAL_CLK 切换至 PLL_CLK，系统定时器将执行自动时间补偿，以提高定时器准确性。
- 支持设置不同的报警值或周期（目标）生成三个独立中断。
- 支持 64 位报警值和 30 位报警周期。
- 芯片进入 Deep-sleep 或 Light-sleep 之后，RTC 定时器将记录睡眠时间。当芯片从上述睡眠模式唤醒之后，系统定时器可以通过软件加载 RTC 定时器记录的睡眠时间，然后进行更新。
- CPU 处于停止状态或处于在线调试状态时，系统定时器也将停止运行。

15.3 时钟源选择

用户可从 XTAL_CLK 或 PLL_CLK 中选择一个作为时钟源。选择方式见章节 2 复位和时钟中的表 8 *CPU_CLK 源*。具体用于系统定时器的时钟频率见表 11 *APB_CLK 源*。定时器将根据使用的具体时钟源，每一时钟周期按照 SYSTIMER_TIMER_XTAL_STEP 或 SYSTIMER_TIMER_PLL_STEP 中设定的幅度递增。

15.4 功能描述

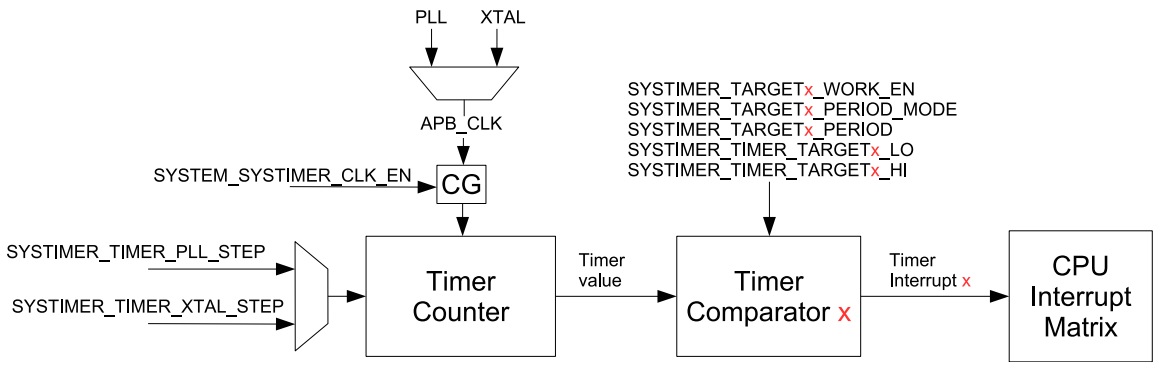


图 15-1. 系统定时器结构

上图 15-1 为系统定时器结构图。用户可通过置位寄存器 SYSTEM_PERIP_CLK_EN0_REG 中的 SYSTEM_SYSTIMER_CLK_EN 位使能系统定时器，或置位寄存器 SYSTEM_PERIP_RST_EN0_REG 中的

SYSTEM_SYSTIMER_RST 位通过软件复位系统定时器。详细信息可参考 6 系统寄存器中表 33 外设时钟门控与复位控制位。

15.4.1 读取系统定时器的值

1. 置位 SYSTIMER_TIMER_UPDATE 将系统定时器的值更新至寄存器。
2. 等待 SYSTIMER_TIMER_VALUE_VALID 置位。置位后则可以从寄存器中读取系统定时器的值。
3. 从 SYSTIMER_TIMER_VALUE_HI 中读取定时器数值的高 32 位，从 SYSTIMER_TIMER_VALUE_LO 中读取定时器数值的低 32 位。

15.4.2 设置一次性延时报警

1. 读取当前系统定时器的值，步骤见章节 15.4.1。读取的当前值可用于计算步骤 3 中的报警值。
2. 清零 SYSTIMER_TARGET*x*_PERIOD_MODE，设置定时器工作模式为一次性延时报警。
3. 将目标值（报警值）高 32 位写入 SYSTIMER_TIMER_TARGET*x*_HI，低 32 位写入 SYSTIMER_TIMER_TARGET*x*_LO。
4. 置位 SYSTIMER_TARGET*x*_WORK_EN，使能上述选择的工作模式。
5. 置位 SYSTIMER_INT*x*_ENA 使能定时器中断。定时器计数达到报警值则触发中断。

15.4.3 设置周期性报警

1. 置位 SYSTIMER_TARGET*x*_PERIOD_MODE，设置定时器工作模式为周期性报警。
2. 将目标值（报警周期）写入 SYSTIMER_TARGET*x*_PERIOD。
3. 置位 SYSTIMER_TARGET*x*_WORK_EN，使能上述选择的工作模式。
4. 置位 SYSTIMER_INT*x*_ENA 使能定时器中断。定时器计数达到目标值则触发中断。

15.4.4 定时器更新与睡眠模式

1. 在芯片进入 Deep-sleep 或 Light-sleep 之前，用户需配置 RTC 定时器用于精确记录睡眠时间，见低功耗管理章节。
2. 系统从睡眠模式唤醒后开始读取 RTC 定时器记录的睡眠时间，见低功耗管理章节。
3. 读取系统定时器当前值，具体步骤见章节 15.4.1。
4. 定时器当前值与睡眠时间相加。
5. 相加的和写入 SYSTIMER_TIMER_LOAD_HI（高 32 位）和 SYSTIMER_TIMER_LOAD_LO（低 32 位）。
6. 置位 SYSTIMER_TIMER_LOAD 将步骤 5 中寄存器的值加载至系统定时器，实现系统定时器的更新。

15.5 基地址

用户可以通过两个不同的寄存器基地址访问系统定时器相关的寄存器，如表 52 所示。更多信息，请访问章节 1 系统和存储器。

表 52: 系统定时器基地址

访问方式	基地址
PeriBUS1	0x3F423000
PeriBUS2	0x60023000

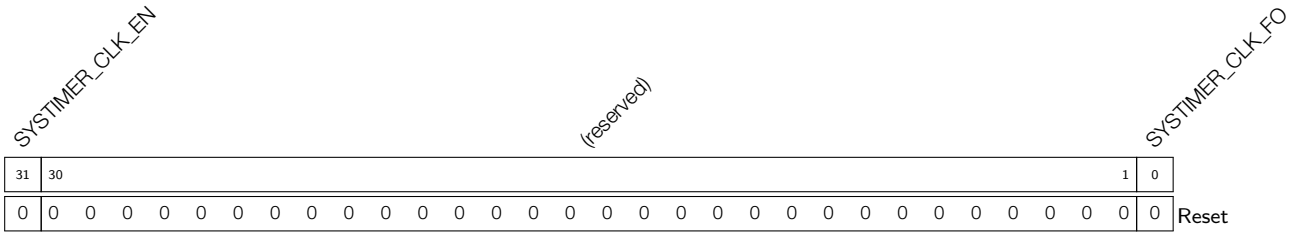
15.6 寄存器列表

请注意，表中的地址为相对于基地址的地址偏移量（相对地址）。请参考章节 15.5 获取基地址相关信息。

名称	描述	地址	访问
系统定时器寄存器			
SYSTIMER_CONF_REG	配置系统定时器的时钟	0x0000	读/写
SYSTIMER_LOAD_REG	加载数值至系统定时器	0x0004	只写
SYSTIMER_LOAD_HI_REG	待加载至系统定时器的值，高 32 位	0x0008	读/写
SYSTIMER_LOAD_LO_REG	待加载至系统定时器的值，低 32 位	0x000C	读/写
SYSTIMER_STEP_REG	系统定时器的递增幅度	0x0010	读/写
SYSTIMER_TARGET0_HI_REG	系统定时器的目标 0，高 32 位	0x0014	读/写
SYSTIMER_TARGET0_LO_REG	系统定时器的目标 0，低 32 位	0x0018	读/写
SYSTIMER_TARGET1_HI_REG	系统定时器的目标 1，高 32 位	0x001C	读/写
SYSTIMER_TARGET1_LO_REG	系统定时器的目标 1，低 32 位	0x0020	读/写
SYSTIMER_TARGET2_HI_REG	系统定时器的目标 2，高 32 位	0x0024	读/写
SYSTIMER_TARGET2_LO_REG	系统定时器的目标 2，低 32 位	0x0028	读/写
SYSTIMER_TARGET0_CONF_REG	配置系统定时器目标 0 的工作模式	0x002C	读/写
SYSTIMER_TARGET1_CONF_REG	配置系统定时器目标 1 的工作模式	0x0030	读/写
SYSTIMER_TARGET2_CONF_REG	配置系统定时器目标 2 的工作模式	0x0034	读/写
SYSTIMER_UPDATE_REG	读取系统定时器的值	0x0038	不定
SYSTIMER_VALUE_HI_REG	系统定时器的值，高 32 位	0x003C	只读
SYSTIMER_VALUE_LO_REG	系统定时器的值，低 32 位	0x0040	只读
SYSTIMER_INT_ENA_REG	使能系统定时器中断	0x0044	读/写
SYSTIMER_INT_RAW_REG	系统定时器原始中断位	0x0048	只读
SYSTIMER_INT_CLR_REG	系统定时器中断清零位	0x004C	只写
版本寄存器			
SYSTIMER_DATE_REG	版本控制寄存器	0x00FC	读/写

15.7 寄存器

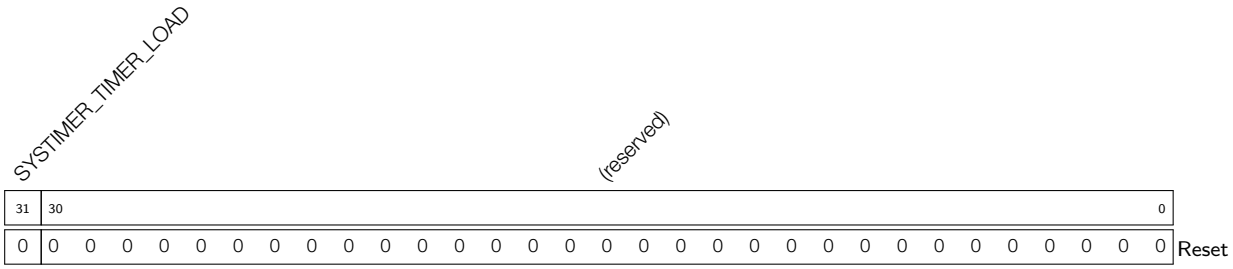
Register 15.1: SYSTIMER_CONF_REG (0x0000)



SYSTIMER_CLK_FO 系统定时器时钟强制使能。(读/写)

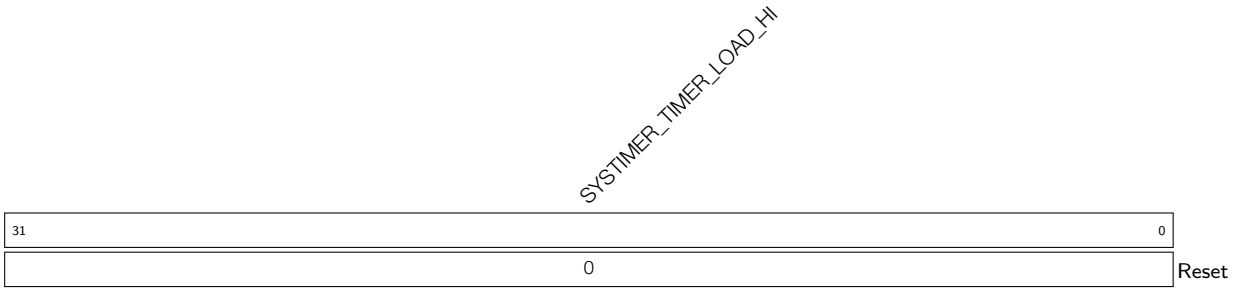
SYSTIMER_CLK_EN 使能寄存器时钟。(读/写)

Register 15.2: SYSTIMER_LOAD_REG (0x0004)



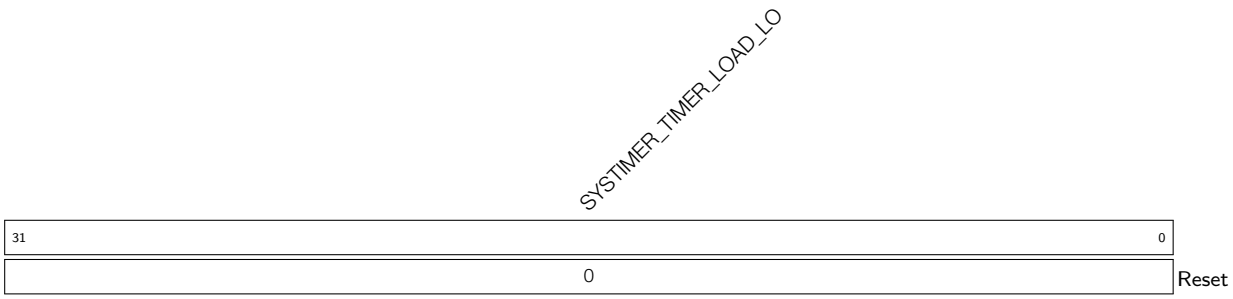
SYSTIMER_TIMER_LOAD 此位置 1 时，SYSTIMER_TIMER_LOAD_HI 和 SYSTIMER_TIMER_LOAD_LO 中存储的数值将加载至系统定时器。(只写)

Register 15.3: SYSTIMER_LOAD_HI_REG (0x0008)



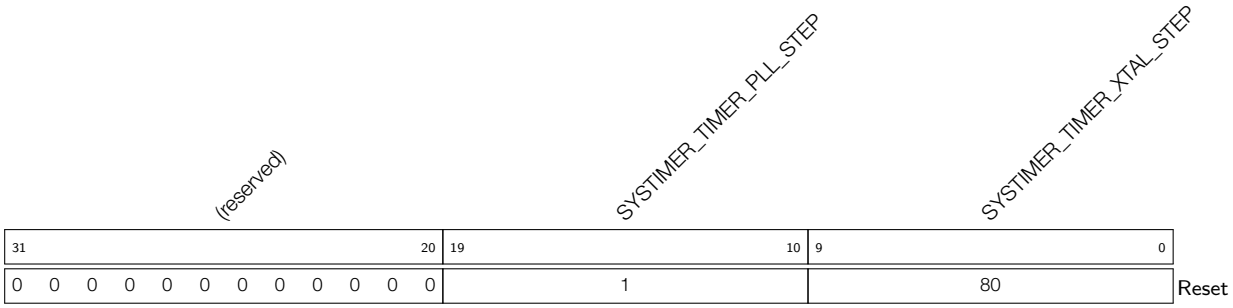
SYSTIMER_TIMER_LOAD_HI 待加载至系统定时器的数据，高 32 位。(读/写)

Register 15.4: SYSTIMER_LOAD_LO_REG (0x000C)



SYSTIMER_TIMER_LOAD_LO 待加载至系统定时器的数据，低 32 位。（读/写）

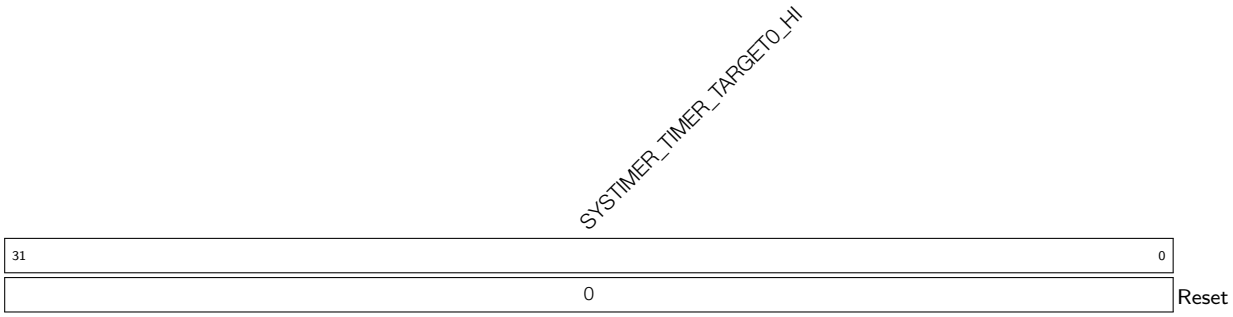
Register 15.5: SYSTIMER_STEP_REG (0x0010)



SYSTIMER_TIMER_XTAL_STEP 选择 XTAL_CLK 用作时钟源时，系统定时器的递增幅度。（读/写）

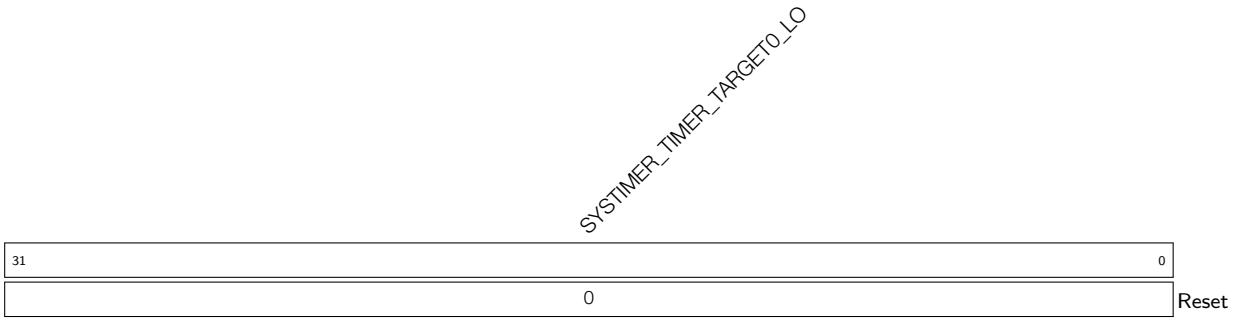
SYSTIMER_TIMER_PLL_STEP 选择 PLL_CLK 用作时钟源时，系统定时器的递增幅度。（读/写）

Register 15.6: SYSTIMER_TARGET0_HI_REG (0x0014)



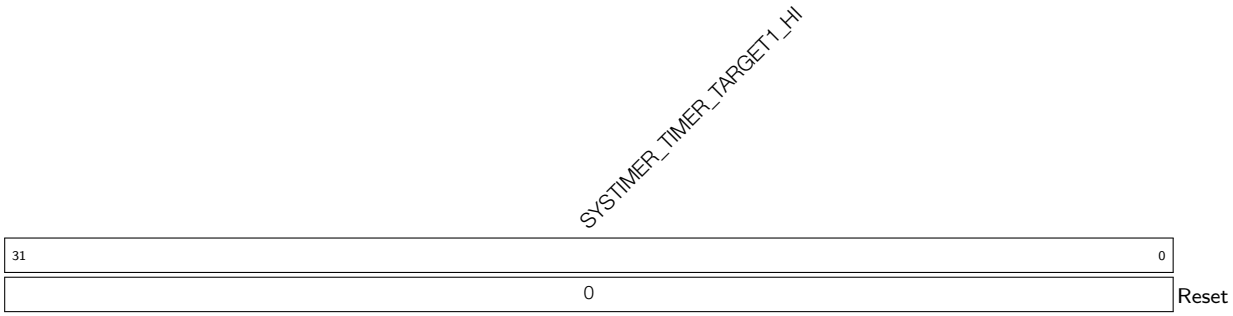
SYSTIMER_TIMER_TARGET0_HI 系统定时器目标 0，高 32 位。（读/写）

Register 15.7: SYSTIMER_TARGET0_LO_REG (0x0018)



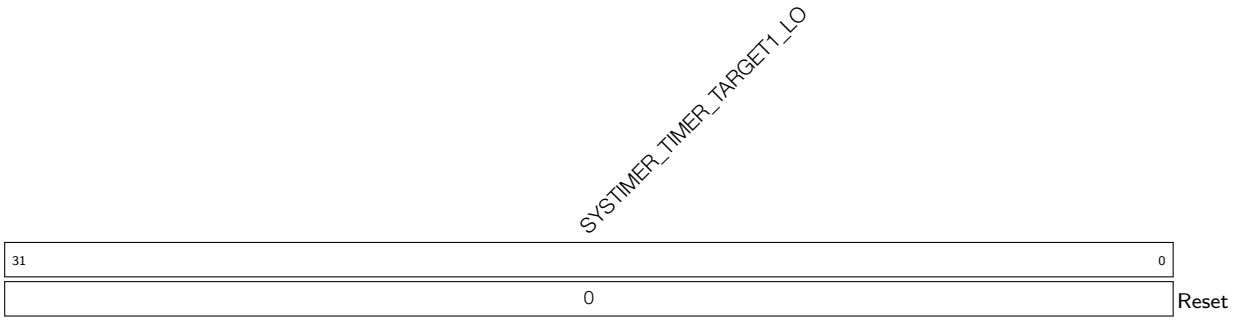
SYSTIMER_TIMER_TARGET0_LO 系统定时器目标 0，低 32 位。（读/写）

Register 15.8: SYSTIMER_TARGET1_HI_REG (0x001C)



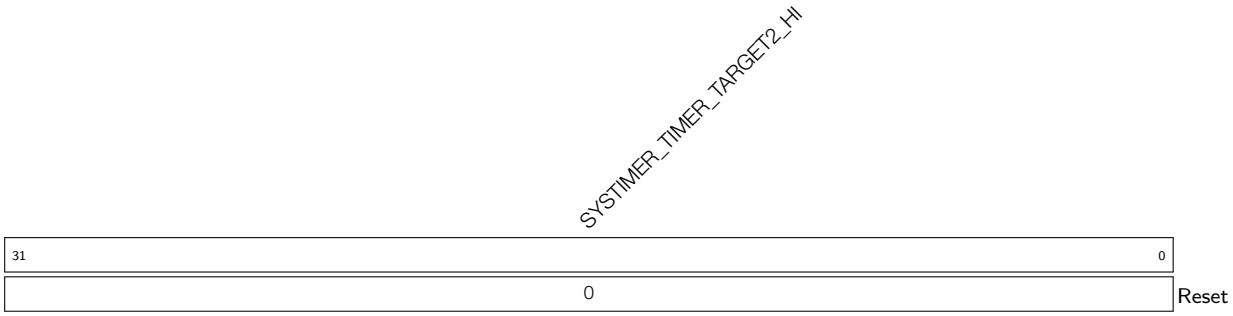
SYSTIMER_TIMER_TARGET1_HI 系统定时器目标 1，高 32 位。（读/写）

Register 15.9: SYSTIMER_TARGET1_LO_REG (0x0020)



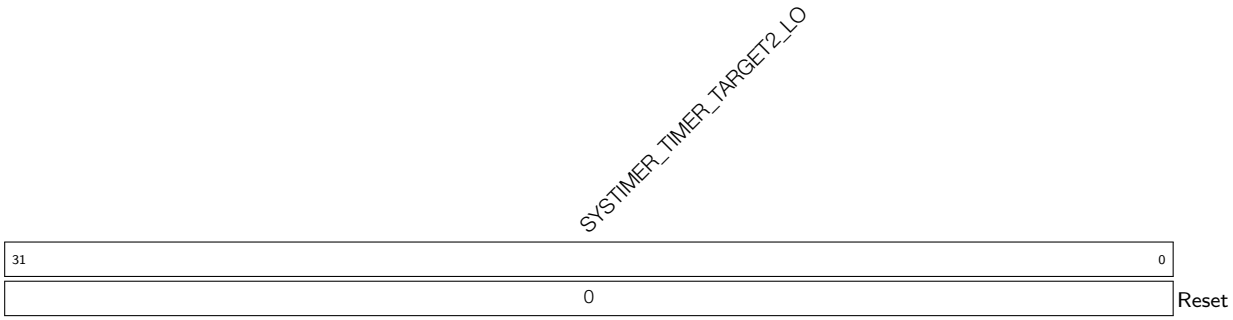
SYSTIMER_TIMER_TARGET1_LO 系统定时器目标 1，低 32 位。（读/写）

Register 15.10: SYSTIMER_TARGET2_HI_REG (0x0024)



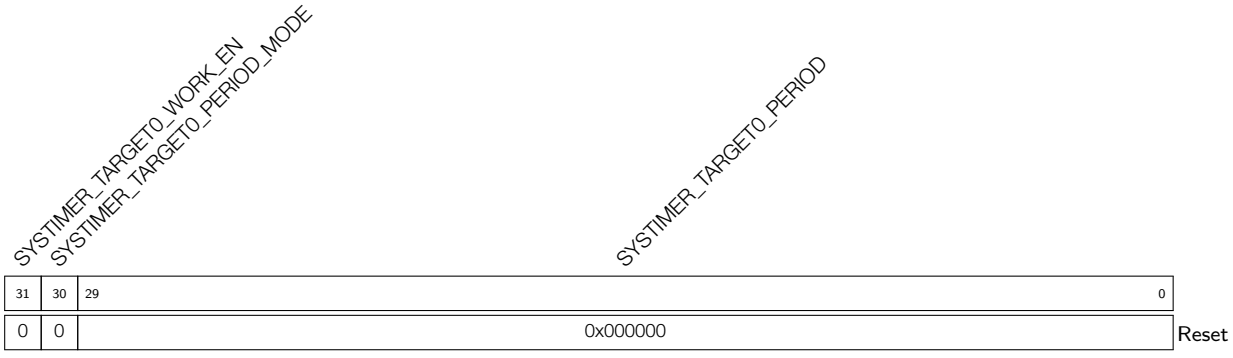
SYSTIMER_TIMER_TARGET2_HI 系统定时器目标 2，高 32 位。（读/写）

Register 15.11: SYSTIMER_TARGET2_LO_REG (0x0028)



SYSTIMER_TIMER_TARGET2_LO 系统定时器目标 2，低 32 位。（读/写）

Register 15.12: SYSTIMER_TARGET0_CONF_REG (0x002C)

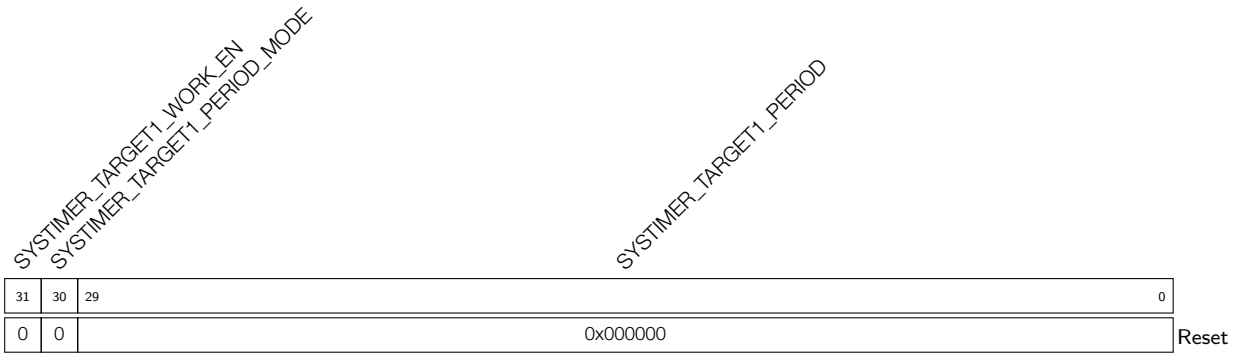


SYSTIMER_TARGET0_PERIOD 设置系统定时器目标 0 的报警周期。（读/写）

SYSTIMER_TARGET0_PERIOD_MODE 设置系统定时器目标 0 的工作模式。0：一次性延时报警；1：周期性报警。（读/写）

SYSTIMER_TARGET0_WORK_EN 系统定时器目标 0 的工作使能位。（读/写）

Register 15.13: SYSTIMER_TARGET1_CONF_REG (0x0030)

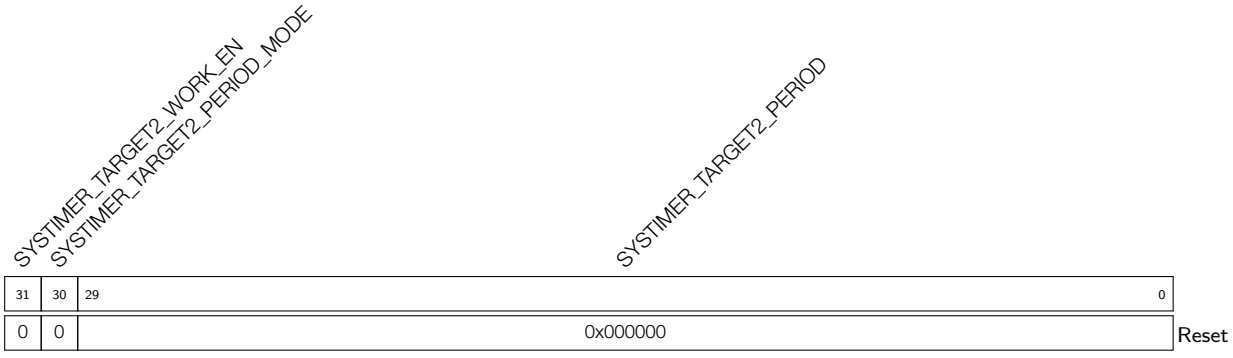


SYSTIMER_TARGET1_PERIOD 设置系统定时器目标 1 的报警周期。(读/写)

SYSTIMER_TARGET1_PERIOD_MODE 设置系统定时器目标 1 的工作模式。0：一次性延时报警；1：周期性报警。(读/写)

SYSTIMER_TARGET1_WORK_EN 系统定时器目标 1 的工作使能位。(读/写)

Register 15.14: SYSTIMER_TARGET2_CONF_REG (0x0034)

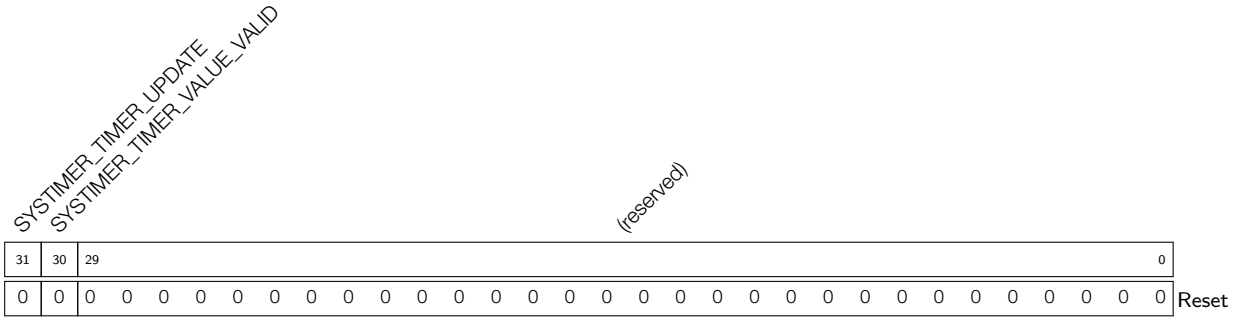


SYSTIMER_TARGET2_PERIOD 设置系统定时器目标 2 的报警周期。(读/写)

SYSTIMER_TARGET2_PERIOD_MODE 设置系统定时器目标 2 的工作模式。0：一次性延时报警；1：周期性报警。(读/写)

SYSTIMER_TARGET2_WORK_EN 系统定时器目标 2 的工作使能位。(读/写)

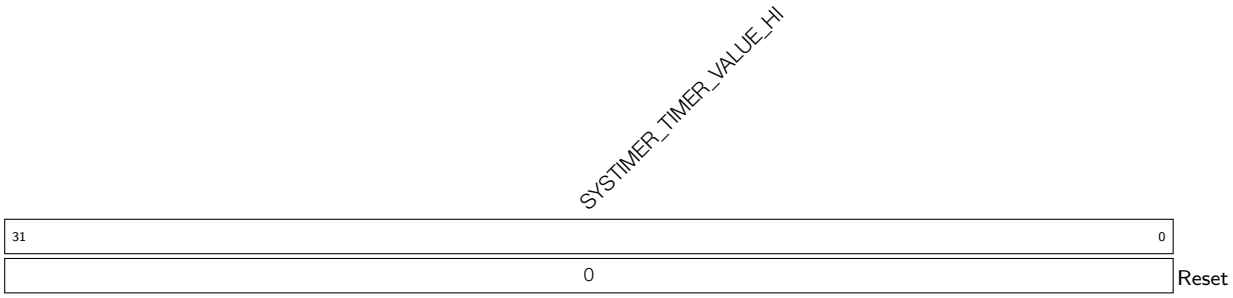
Register 15.15: SYSTIMER_UPDATE_REG (0x0038)



SYSTIMER_TIMER_VALUE_VALID 查看从寄存器读取定时器的值是否有效。0：无效，即暂时还不能从寄存器中读取系统定时器的值；1：有效，即此时可以从寄存器中读取系统定时器的值。（只读）

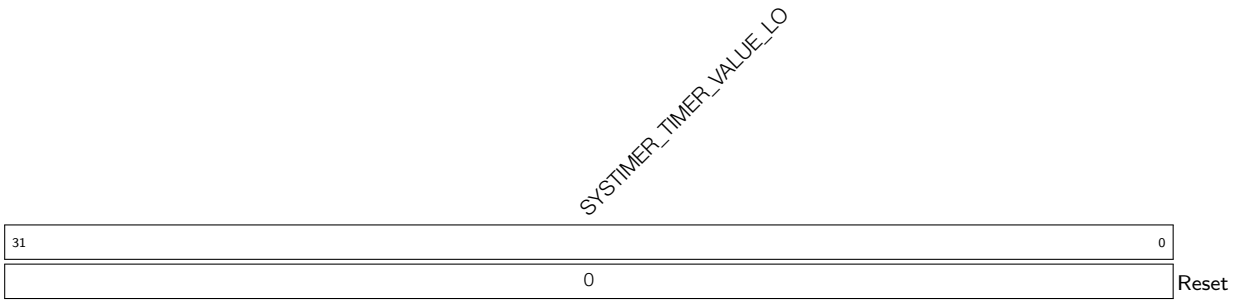
SYSTIMER_TIMER_UPDATE 将系统定时器的数值更新至寄存器。（只写）

Register 15.16: SYSTIMER_VALUE_HI_REG (0x003C)



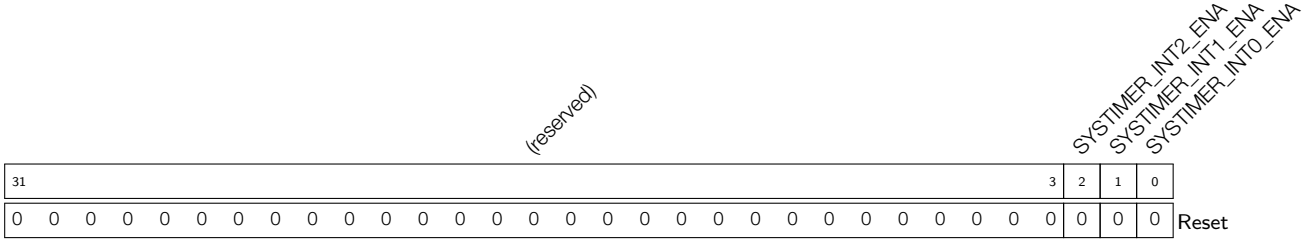
SYSTIMER_TIMER_VALUE_HI 系统定时器的值，高 32 位。（只读）

Register 15.17: SYSTIMER_VALUE_LO_REG (0x0040)



SYSTIMER_TIMER_VALUE_LO 系统定时器的值，低 32 位。（只读）

Register 15.18: SYSTIMER_INT_ENA_REG (0x0044)

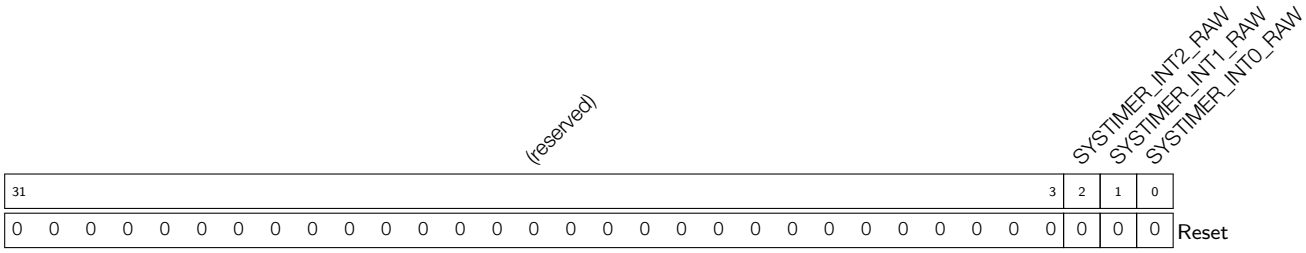


SYSTIMER_INT0_ENA 系统定时器目标 0 的中断使能位。(读/写)

SYSTIMER_INT1_ENA 系统定时器目标 1 的中断使能位。(读/写)

SYSTIMER_INT2_ENA 系统定时器目标 2 的中断使能位。(读/写)

Register 15.19: SYSTIMER_INT_RAW_REG (0x0048)

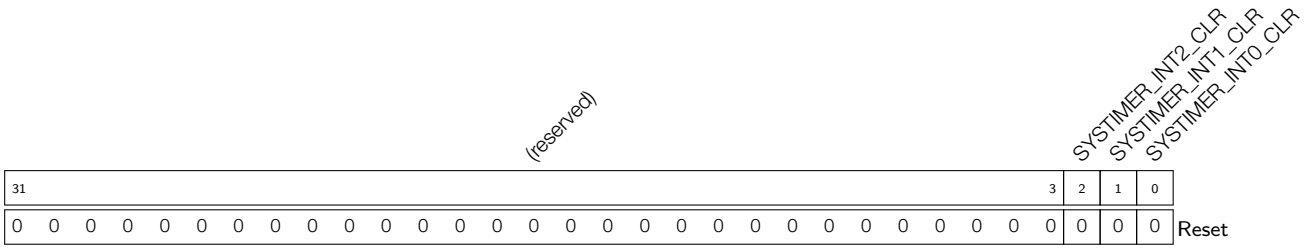


SYSTIMER_INT0_RAW 系统定时器目标 0 的原始中断位。(只读)

SYSTIMER_INT1_RAW 系统定时器目标 1 的原始中断位。(只读)

SYSTIMER_INT2_RAW 系统定时器目标 2 的原始中断位。(只读)

Register 15.20: SYSTIMER_INT_CLR_REG (0x004C)

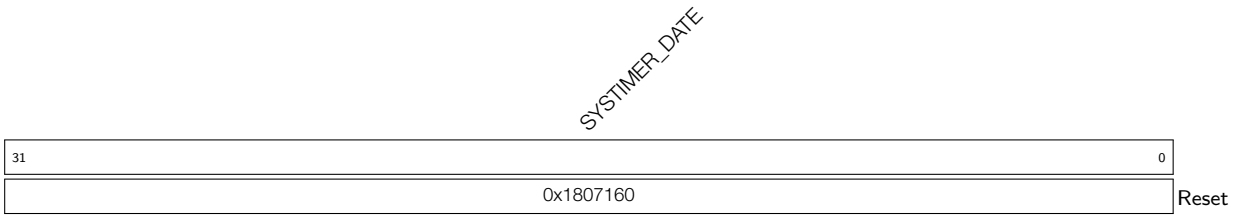


SYSTIMER_INT0_CLR 系统定时器目标 0 的中断清零位。(只写)

SYSTIMER_INT1_CLR 系统定时器目标 1 的中断清零位。(只写)

SYSTIMER_INT2_CLR 系统定时器目标 2 的中断清零位。(只写)

Register 15.21: SYSTIMER_DATE_REG (0x00FC)



SYSTIMER_DATE 版本控制寄存器。（读/写）

16. eFuse 控制器

16.1 概述

ESP32-S2 系统中有一块 4096 位的 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 控制器按照软件配置完成对 eFuse 中各参数中的各个位的烧写。这些参数有些可以通过 eFuse 控制器被软件读取，有些直接由硬件模块使用。

16.2 主要特性

- 一次性可编程存储
- 烧写保护可配置
- 软件读取保护可配置
- 使用多种硬件编码方式保护参数内容

16.3 功能描述

16.3.1 结构

eFuse 从结构上分成 11 个块 (BLOCK0 ~ BLOCK10)。

BLOCK0 存储大部分核心系统参数，其中 24 位供硬件使用，软件不可见；还有 38 位处于保留状态，留作未来使用。

表 54 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** 用于控制软件读取 BLOCK4 ~ BLOCK10。更多关于这两个参数的信息请见章节 16.3.1.1、16.3.1.2。

表 54: BLOCK0 参数

参数	偏移	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	描述
EFUSE_WR_DIS	0	32	Y	N/A	禁止 eFuse 烧写
EFUSE_RD_DIS	32	7	Y	0	禁止软件读取 eFuse BLOCK4-10 的内容
EFUSE_DIS_RTC_RAM_BOOT	39	1	N	1	禁止从 RTC RAM 启动
EFUSE_DIS_ICACHE	40	1	Y	2	关闭 ICache
EFUSE_DIS_DCACHE	41	1	Y	2	关闭 DCache
EFUSE_DIS_DOWNLOAD_ICACHE	42	1	Y	2	在 Download 模式下关闭 ICache
EFUSE_DIS_DOWNLOAD_DCACHE	43	1	Y	2	在 Download 模式下关闭 DCache
EFUSE_DIS_FORCE_DOWNLOAD	44	1	Y	2	禁止强制芯片进入 Download 模式
EFUSE_DIS_USB	45	1	Y	2	关闭 USB OTG 功能
EFUSE_DIS_CAN	46	1	Y	2	关闭 TWAI 控制器功能
EFUSE_DIS_BOOT_REMAP	47	1	Y	2	REMAP 指代 RAM 空间可以映射到 ROM 空间，此功能可被关闭

参数	偏移	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	描述
EFUSE_SOFT_DIS_JTAG	49	1	Y	2	软件禁用 JTAG，可通过 HMAC 重新启动
EFUSE_HARD_DIS_JTAG	50	1	Y	2	硬件永远禁用 JTAG
EFUSE_DIS_DOWNLOAD_ MANUAL_ENCRYPT	51	1	Y	2	在 download boot 模式下禁用 flash 加密功能
EFUSE_USB_EXCHG_PINS	56	1	Y	30	交换 USB D+ / D- 管脚
EFUSE_EXT_PHY_ENABLE	57	1	N	30	使能外部 USB PHY
EFUSE_USB_FORCE_NOPERSIST	58	1	N	30	强制设置 USB BVALID 为 1
EFUSE_VDD_SPI_XPD	68	1	Y	3	若 VDD_SPI_FORCE 为 1，控制 VDD_SPI 调节器上电
EFUSE_VDD_SPI_TIEH	69	1	Y	3	若 VDD_SPI_FORCE 为 1，选择 VDD_SPI 电压。0：VDD_SPI 连接 1.8 V LDO；1：VDD_SPI 连接 VDD_RTC_IO
EFUSE_VDD_SPI_FORCE	70	1	Y	3	置位使用 XPD_VDD_PSI_REG 和 VDD_SPI_TIEH 配置 VDD_SPI LDO
EFUSE_WDT_DELAY_SEL	80	2	Y	3	选择 RTC WDT 超时阈值
EFUSE_SPI_BOOT_CRYPT_CNT	82	3	Y	4	使能 SPI boot 加解密，奇数个 1：使能；偶数个 1：关闭
EFUSE_SECURE_BOOT_KEY_REVOKE0	85	1	N	5	使能撤销第一个 secure boot（安全启动）密钥
EFUSE_SECURE_BOOT_KEY_REVOKE1	86	1	N	6	使能撤销第二个 secure boot 密钥
EFUSE_SECURE_BOOT_KEY_REVOKE2	87	1	N	7	使能撤销第三个 secure boot 密钥
EFUSE_KEY_PURPOSE_0	88	4	Y	8	Key0 用途 (purpose)，见表 55
EFUSE_KEY_PURPOSE_1	92	4	Y	9	Key1 用途，见表 55
EFUSE_KEY_PURPOSE_2	96	4	Y	10	Key2 用途，见表 55
EFUSE_KEY_PURPOSE_3	100	4	Y	11	Key3 用途，见表 55
EFUSE_KEY_PURPOSE_4	104	4	Y	12	Key4 用途，见表 55
EFUSE_KEY_PURPOSE_5	108	4	Y	13	Key5 用途，见表 55
EFUSE_SECURE_BOOT_EN	116	1	N	15	使能 secure boot
EFUSE_SECURE_BOOT_ AGGRESSIVE_REVOKE	117	1	N	16	Secure boot 的撤销采用激进策略
EFUSE_FLASH_TPUW	124	4	N	18	上电后 flash 等待时间，单位为 (ms/2)，值为 15 时，等待时间为 7.5 ms
EFUSE_DIS_DOWNLOAD_MODE	128	1	N	18	关闭所有 download boot 模式
EFUSE_DIS_LEGACY_SPI_BOOT	129	1	N	18	关闭 Legacy SPI boot 模式
EFUSE_UART_PRINT_CHANNEL	130	1	N	18	选择打印 boot 信息的 UART 通道。0: UART0; 1: UART1
EFUSE_DIS_USB_DOWNLOAD_MODE	132	1	N	18	在 UART download boot 模式下关闭 USB OTG 下载功能

参数	偏移	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	描述
EFUSE_ENABLE_SECURITY_DOWNLOAD	133	1	N	18	使能 UART 安全下载模式（仅支持读写 flash）
EFUSE_UART_PRINT_CONTROL	134	2	N	18	控制 UART boot 信息输出模式。2'b00：强制打印；2'b01：由 GPIO 46 控制，低电平打印；2'b10：由 GPIO 46 控制，高电平打印；2'b11：强制关闭打印
EFUSE_PIN_POWER_SELECTION	136	1	N	18	选择 GPIO33 ~ GPIO37 的电源；0：VDD3P3_CPU；1：VDD_SPI
EFUSE_FLASH_TYPE	137	1	N	18	Flash 类型；0：4 根数据线；1：8 根数据线
EFUSE_FORCE_SEND_RESUME	138	1	N	18	强制 ROM 代码在 SPI 启动过程中发送 SPI flash 继续指令
EFUSE_SECURE_VERSION	139	16	N	18	安全版本（用于 ESP-IDF 的防回滚功能）

表 55 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 55: 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（仅为软件使用）
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 Downstream（下行）模式使用
6	指定为 HMAC Downstream 模式下的 JTAG 使用
7	指定为 HMAC Downstream 模式下的数字签名使用
8	指定为 HMAC Upstream（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 56 列出了 BLOCK1 ~ BLOCK10 中存储的参数的信息。

表 56: BLOCK1-10 参数

块	参数	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	EFUSE_RD_DIS 软件读取保护位	描述
BLOCK1	EFUSE_MAC	48	N	20	N/A	MAC 地址
	EFUSE_SPI_PAD_CONFIGURE	[0:5]	N	20	N/A	CLK
		[6:11]	N	20	N/A	Q (D1)
		[12:17]	N	20	N/A	D (D0)

块	参数	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	EFUSE_RD_DIS 软件读取保护位	描述
		[18:23]	N	20	N/A	CS
		[24:29]	N	20	N/A	HD (D3)
		[30:35]	N	20	N/A	WP (D2)
		[36:41]	N	20	N/A	DQS
		[42:47]	N	20	N/A	D4
		[48:53]	N	20	N/A	D5
		[54:59]	N	20	N/A	D6
		[60:65]	N	20	N/A	D7
	EFUSE_SYS_DATA_PART0	78	N	20	N/A	系统数据
BLOCK2	EFUSE_SYS_DATA_PART1	256	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
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 55）。例如，软件将用于 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 16.3.1.3：*数据存储方式*，和章节 16.3.2：*软件烧写参数*。

16.3.1.1 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 16.3.3 中：*更新 eFuse 读寄存器*）。

表 54 以及表 56 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，但已经被烧写的参数不能被重复烧写。

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。

16.3.1.2 EFUSE_RD_DIS

所有参数中，只有 BLCOK4 ~ BLOCK10 的参数受软件读取保护状态的约束，即表 56 中“EFUSE_RD_DIS 软件读取保护”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 16.3.3 中：*更新 eFuse 读寄存器*）。

参数 EFUSE_RD_DIS 中的某个位为 0，表示此位管理的参数未处于软件读取保护状态；某个位为 1，表示此位管理的参数处于软件读取保护状态。

除 BLOCK4 ~ BLOCK10 之外，其他参数不受软件读取保护状态的约束，均可被软件读取。

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* 被硬件使用。

16.3.1.3 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 备份机制对软件不可见。

BLOCK1 ~ BLOCK10 使用 RS (44, 32) 编码方式，最多支持自动校正 5 个字节。本文 RS (44, 32) 使用的本源多项式为 $p(x) = x^8 + x^4 + x^3 + x^2 + 1$，产生校验码的移位寄存器电路如图 16-1 所示，其中 gf_mul_n（n 为一个整数）为 $GF(2^8)$ 域中某一字节数据与元素 α^n 相乘的结果。

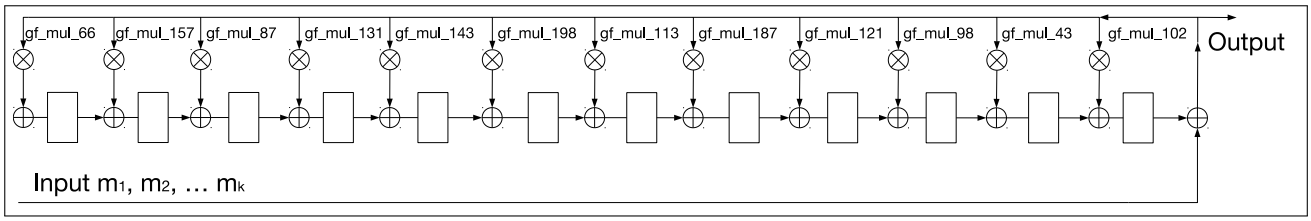


图 16-1. 移位寄存器电路图

软件需要先对 32 字节参数进行 RS (44, 32) 编码得到 12 字节验证码，然后将参数及验证码一起烧入 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。

16.3.2 软件烧写参数

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。

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 24 位为硬件可用、软件不可见的有效信息，必须写入 0，对应位置为：

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

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. 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. 确保 eFuse 时钟寄存器的配置正确，具体请参考章节 16.3.4.1：*eFuse 烧写时序*。
4. 确保 eFuse 烧写电压 VDDQ 的配置正确，具体请参考章节 16.3.4.2：*eFuse VDDQ 时序*。
5. 配置寄存器 EFUSE_CONF_REG 的 EFUSE_OP_CODE 位段为 0x5A5A。
6. 配置寄存器 EFUSE_CMD_REG 的 EFUSE_PGM_CMD 位段为 1。
7. 轮询寄存器 EFUSE_CMD_REG 直到其为 0x0，或者等待烧写完成中断产生。识别烧写/读取完成中断产生的方法详见章节 16.3.3 最后的说明。
8. 将寄存器中写入的参数清零。
9. 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 16.3.3：*软件读取参数*。

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。

16.3.3 软件读取参数

软件不能直接读取 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，软件再通过读取以 EFUSE_RD_ 开始的寄存器来获取 eFuse 信息。下表列出了读取数据的寄存器名称以及对应烧写时的烧写寄存器名称。

BLOCK	读寄存器	烧写寄存器

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，也可以根据需要由软件手动触发（例如在需要读取新烧写 eFuse 中的数据内容时）。软件触发 eFuse 读取操作的流程如下：

1. 配置 eFuse 读取时序，具体请参考章节 16.3.4.3：*eFuse 读取时序*。
2. 配置寄存器 EFUSE_CONF_REG 的 EFUSE_OP_CODE 位段为 0x5AA5。
3. 配置寄存器 EFUSE_CMD_REG 的 EFUSE_READ_CMD 位段为 1。
4. 轮询寄存器 EFUSE_CMD_REG 直到其为 0x0，或者等待 read_done interrupt（读取完成中断）产生，识别烧写/读取完成中断产生的方法详见下方说明。
5. 软件从 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软件检测 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。

软件只可以在更新 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/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。
  3. 等待烧写/读取完成中断产生。
  4. 对寄存器 EFUSE_INT_CLR_REG 的位 1/0 置 1 以清除烧写/读取完成中断。

16.3.4 时序

16.3.4.1 eFuse 烧写时序

图 16-2 是 eFuse 烧写的时序图。硬件提供 EFUSE_TSUP_A、EFUSE_TPGM、EFUSE_THP_A 和 EFUSE_TPGM_INACTIVE 四个寄存器用于控制 eFuse 的烧写时序。图中 CSB、VDDQ、PGENB 的含义如下：

- CSB：片选信号，低电平有效
- VDDQ：eFuse 烧写电压
- PGENB：烧写使能信号，低电平有效

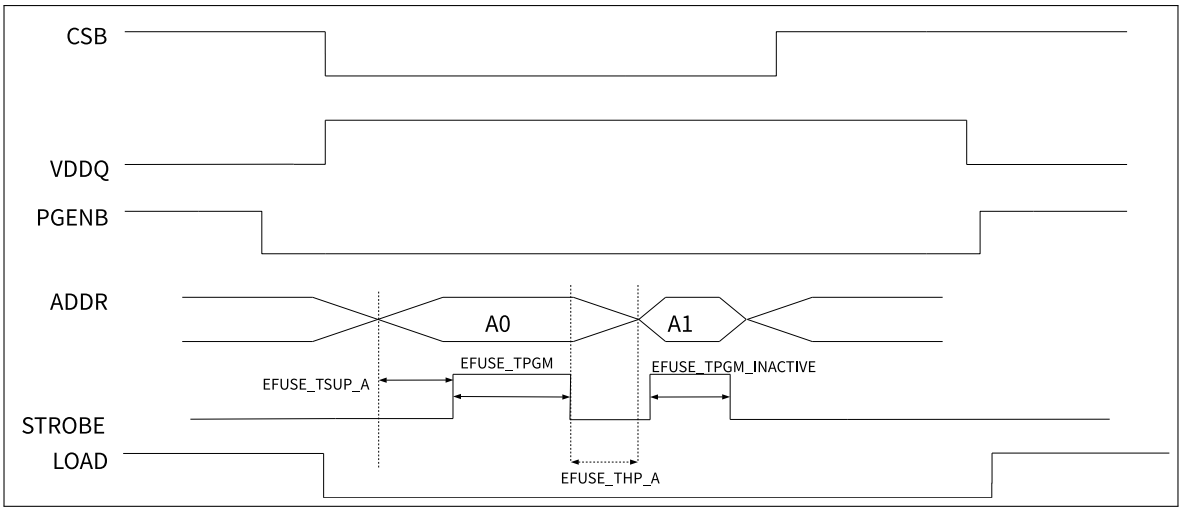


图 16-2. eFuse 烧写时序图

eFuse 模块使用的是 CLK_APB 时钟。由于 CLK_APB 时钟频率是可变的，需要根据不同的时钟频率来配置上述时序参数，具体请见表 58。复位后，eFuse 默认的参数配置对应的是 20 MHz 时钟频率。

表 58: eFuse 烧写时序参数配置

APB Frequency	EFUSE_TSUP_A (> 6.669 ns)	EFUSE_TPGM (9-11 μs, usually 10 μs)	EFUSE_THP_A (> 6.166 ns)	EFUSE_TPGM_INACTIVE (> 35.96 ns)
80 MHz	0x2	0x320	0x2	0x4
40 MHz	0x1	0x190	0x1	0x2
20 MHz	0x1	0xC8	0x1	0x1

图 16-2 中 A0 地址烧写 1，即 A0 对应的 eFuse 位为 1；A1 地址处于不烧写状态，A1 对应的 eFuse 位为 0。

16.3.4.2 eFuse VDDQ 时序

用户需要根据不同的 APB 时钟频率来配置 eFuse 的烧写电压 VDDQ 的时序参数，具体配置如下：

表 59: VDDQ 时序参数配置

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

16.3.4.3 eFuse 读取时序

图 16-3 是 eFuse 读取的时序图。硬件提供 EFUSE_TSUR_A、EFUSE_TRD 和 EFUSE_THR_A 三个寄存器用于控制 eFuse 的读时序。

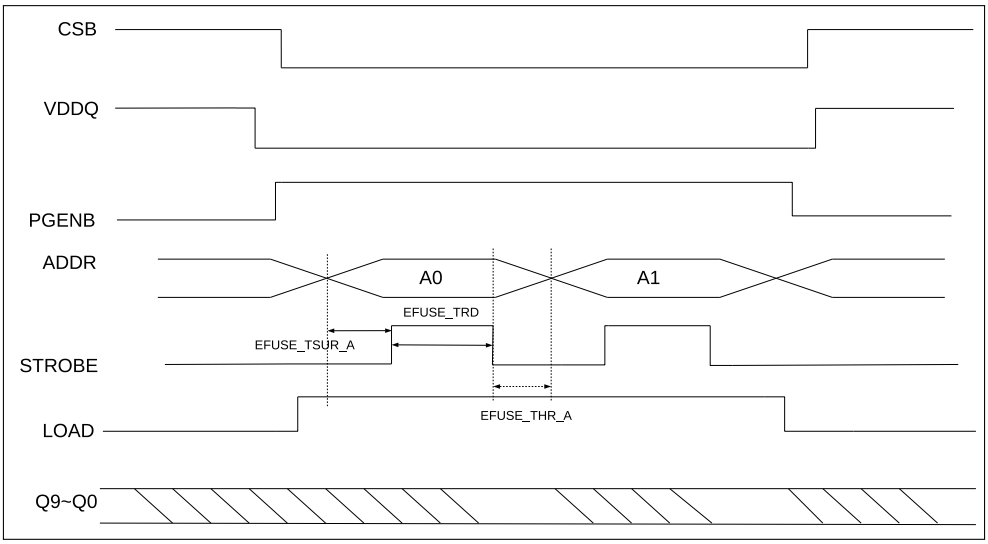


图 16-3. eFuse 读取时序图

用户需要根据不同的 APB 时钟频率来配置上述时序参数，具体请见表 60。

表 60: eFuse 读取时序参数配置

APB Frequency	EFUSE_TSUR_A (> 6.669 ns)	EFUSE_TRD (> 35.96 ns)	EFUSE_THR_A (> 6.166 ns)
80 MHz	0x2	0x4	0x2
40 MHz	0x1	0x2	0x1
20 MHz	0x1	0x1	0x1

16.3.5 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，软件无法干预这个过程。硬件使用的参数为表 54 和 56 “硬件使用”一栏中标记为 “Y” 的参数。

16.3.6 中断

- 烧写完成中断：当 eFuse 烧写完成后，此中断被触发。如果要启动该中断信号，需将寄存器 EFUSE_PGM_DONE_INT_ENA 置 1。
- 读取完成中断：当 eFuse 读取完成后，此中断被触发。如果要启动该中断信号，需将寄存器

EFUSE_READ_DONE_INT_ENA 置 1。

16.4 基地址

用户可以通过两个不同的寄存器基地址访问 eFuse 控制器，如表 61 所示。更多有关通过不同总线访问外设的信息，请参考章节 1 *系统和存储器*。

表 61: eFuse 控制器基地址

访问外设总线	地址值
PeriBUS1	0x6001A000
PeriBUS2	0x3FC1A000

16.5 寄存器列表

请注意，这里的地址是相对于 eFuse 控制器基地址的地址偏移量（相对地址）。请参阅章节 16.4 获取有关 eFuse 控制器的基地址的信息。

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	读/写
EFUSE_PGM_DATA1_REG	存放待烧写数据的第 1 个寄存器内容。	0x0004	读/写
EFUSE_PGM_DATA2_REG	存放待烧写数据的第 2 个寄存器内容。	0x0008	读/写
EFUSE_PGM_DATA3_REG	存放待烧写数据的第 3 个寄存器内容。	0x000C	读/写
EFUSE_PGM_DATA4_REG	存放待烧写数据的第 4 个寄存器内容。	0x0010	读/写
EFUSE_PGM_DATA5_REG	存放待烧写数据的第 5 个寄存器内容。	0x0014	读/写
EFUSE_PGM_DATA6_REG	存放待烧写数据的第 6 个寄存器内容。	0x0018	读/写
EFUSE_PGM_DATA7_REG	存放待烧写数据的第 7 个寄存器内容。	0x001C	读/写
EFUSE_PGM_CHECK_VALUE0_REG	存放待烧写 RS 代码的第 0 个寄存器数据。	0x0020	读/写
EFUSE_PGM_CHECK_VALUE1_REG	存放待烧写 RS 代码的第 1 个寄存器数据。	0x0024	读/写
EFUSE_PGM_CHECK_VALUE2_REG	存放待烧写 RS 代码的第 2 个寄存器数据。	0x0028	读/写
读取数据寄存器			
EFUSE_RD_WR_DIS_REG	BLOCK0 的第 0 个寄存器内容。	0x002C	只读
EFUSE_RD_REPEAT_DATA0_REG	BLOCK0 的第 1 个寄存器内容。	0x0030	只读
EFUSE_RD_REPEAT_DATA1_REG	BLOCK0 的第 2 个寄存器内容。	0x0034	只读
EFUSE_RD_REPEAT_DATA2_REG	BLOCK0 的第 3 个寄存器内容。	0x0038	只读
EFUSE_RD_REPEAT_DATA3_REG	BLOCK0 的第 4 个寄存器内容。	0x003C	只读
EFUSE_RD_REPEAT_DATA4_REG	BLOCK0 的第 5 个寄存器内容。	0x0040	只读
EFUSE_RD_MAC_SPI_SYS_0_REG	BLOCK1 的第 0 个寄存器内容。	0x0044	只读
EFUSE_RD_MAC_SPI_SYS_1_REG	BLOCK1 的第 1 个寄存器内容。	0x0048	只读
EFUSE_RD_MAC_SPI_SYS_2_REG	BLOCK1 的第 2 个寄存器内容。	0x004C	只读
EFUSE_RD_MAC_SPI_SYS_3_REG	BLOCK1 的第 3 个寄存器内容。	0x0050	只读
EFUSE_RD_MAC_SPI_SYS_4_REG	BLOCK1 的第 4 个寄存器内容。	0x0054	只读
EFUSE_RD_MAC_SPI_SYS_5_REG	BLOCK1 的第 5 个寄存器内容。	0x0058	只读
EFUSE_RD_SYS_DATA_PART1_0_REG	BLOCK2 (system) 的第 0 个寄存器内容。	0x005C	只读
EFUSE_RD_SYS_DATA_PART1_1_REG	BLOCK2 (system) 的第 1 个寄存器内容。	0x0060	只读
EFUSE_RD_SYS_DATA_PART1_2_REG	BLOCK2 (system) 的第 2 个寄存器内容。	0x0064	只读

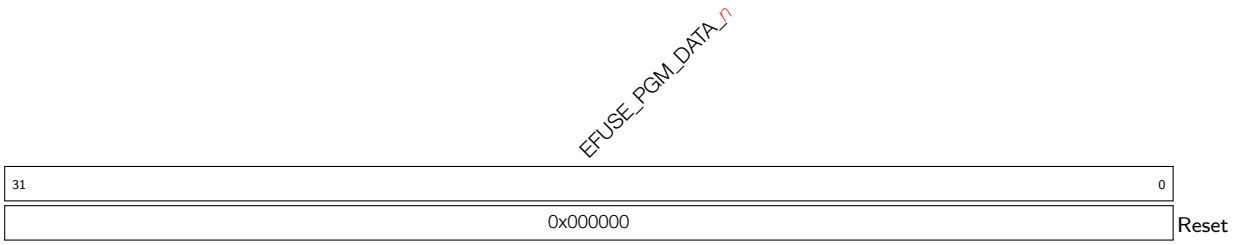
名称	描述	地址	访问
EFUSE_RD_SYS_DATA_PART1_3_REG	BLOCK2 (system) 的第 3 个寄存器内容。	0x0068	只读
EFUSE_RD_SYS_DATA_PART1_4_REG	BLOCK2 (system) 的第 4 个寄存器内容。	0x006C	只读
EFUSE_RD_SYS_DATA_PART1_5_REG	BLOCK2 (system) 的第 5 个寄存器内容。	0x0070	只读
EFUSE_RD_SYS_DATA_PART1_6_REG	BLOCK2 (system) 的第 6 个寄存器内容。	0x0074	只读
EFUSE_RD_SYS_DATA_PART1_7_REG	BLOCK2 (system) 的第 7 个寄存器内容。	0x0078	只读
EFUSE_RD_USR_DATA0_REG	BLOCK3 (user) 的第 0 个寄存器内容。	0x007C	只读
EFUSE_RD_USR_DATA1_REG	BLOCK3 (user) 的第 1 个寄存器内容。	0x0080	只读
EFUSE_RD_USR_DATA2_REG	BLOCK3 (user) 的第 2 个寄存器内容。	0x0084	只读
EFUSE_RD_USR_DATA3_REG	BLOCK3 (user) 的第 3 个寄存器内容。	0x0088	只读
EFUSE_RD_USR_DATA4_REG	BLOCK3 (user) 的第 4 个寄存器内容。	0x008C	只读
EFUSE_RD_USR_DATA5_REG	BLOCK3 (user) 的第 5 个寄存器内容。	0x0090	只读
EFUSE_RD_USR_DATA6_REG	BLOCK3 (user) 的第 6 个寄存器内容。	0x0094	只读
EFUSE_RD_USR_DATA7_REG	BLOCK3 (user) 的第 7 个寄存器内容。	0x0098	只读
EFUSE_RD_KEY0_DATA0_REG	BLOCK4 (KEY0) 的第 0 个寄存器内容。	0x009C	只读
EFUSE_RD_KEY0_DATA1_REG	BLOCK4 (KEY0) 的第 1 个寄存器内容。	0x00A0	只读
EFUSE_RD_KEY0_DATA2_REG	BLOCK4 (KEY0) 的第 2 个寄存器内容。	0x00A4	只读
EFUSE_RD_KEY0_DATA3_REG	BLOCK4 (KEY0) 的第 3 个寄存器内容。	0x00A8	只读
EFUSE_RD_KEY0_DATA4_REG	BLOCK4 (KEY0) 的第 4 个寄存器内容。	0x00AC	只读
EFUSE_RD_KEY0_DATA5_REG	BLOCK4 (KEY0) 的第 5 个寄存器内容。	0x00B0	只读
EFUSE_RD_KEY0_DATA6_REG	BLOCK4 (KEY0) 的第 6 个寄存器内容。	0x00B4	只读
EFUSE_RD_KEY0_DATA7_REG	BLOCK4 (KEY0) 的第 7 个寄存器内容。	0x00B8	只读
EFUSE_RD_KEY1_DATA0_REG	BLOCK5 (KEY1) 的第 0 个寄存器内容。	0x00BC	只读
EFUSE_RD_KEY1_DATA1_REG	BLOCK5 (KEY1) 的第 1 个寄存器内容。	0x00C0	只读
EFUSE_RD_KEY1_DATA2_REG	BLOCK5 (KEY1) 的第 2 个寄存器内容。	0x00C4	只读
EFUSE_RD_KEY1_DATA3_REG	BLOCK5 (KEY1) 的第 3 个寄存器内容。	0x00C8	只读
EFUSE_RD_KEY1_DATA4_REG	BLOCK5 (KEY1) 的第 4 个寄存器内容。	0x00CC	只读
EFUSE_RD_KEY1_DATA5_REG	BLOCK5 (KEY1) 的第 5 个寄存器内容。	0x00D0	只读
EFUSE_RD_KEY1_DATA6_REG	BLOCK5 (KEY1) 的第 6 个寄存器内容。	0x00D4	只读
EFUSE_RD_KEY1_DATA7_REG	BLOCK5 (KEY1) 的第 7 个寄存器内容。	0x00D8	只读
EFUSE_RD_KEY2_DATA0_REG	BLOCK6 (KEY2) 的第 0 个寄存器内容。	0x00DC	只读
EFUSE_RD_KEY2_DATA1_REG	BLOCK6 (KEY2) 的第 1 个寄存器内容。	0x00E0	只读
EFUSE_RD_KEY2_DATA2_REG	BLOCK6 (KEY2) 的第 2 个寄存器内容。	0x00E4	只读
EFUSE_RD_KEY2_DATA3_REG	BLOCK6 (KEY2) 的第 3 个寄存器内容。	0x00E8	只读
EFUSE_RD_KEY2_DATA4_REG	BLOCK6 (KEY2) 的第 4 个寄存器内容。	0x00EC	只读
EFUSE_RD_KEY2_DATA5_REG	BLOCK6 (KEY2) 的第 5 个寄存器内容。	0x00F0	只读
EFUSE_RD_KEY2_DATA6_REG	BLOCK6 (KEY2) 的第 6 个寄存器内容。	0x00F4	只读
EFUSE_RD_KEY2_DATA7_REG	BLOCK6 (KEY2) 的第 7 个寄存器内容。	0x00F8	只读
EFUSE_RD_KEY3_DATA0_REG	BLOCK7 (KEY3) 的第 0 个寄存器内容。	0x00FC	只读
EFUSE_RD_KEY3_DATA1_REG	BLOCK7 (KEY3) 的第 1 个寄存器内容。	0x0100	只读
EFUSE_RD_KEY3_DATA2_REG	BLOCK7 (KEY3) 的第 2 个寄存器内容。	0x0104	只读
EFUSE_RD_KEY3_DATA3_REG	BLOCK7 (KEY3) 的第 3 个寄存器内容。	0x0108	只读
EFUSE_RD_KEY3_DATA4_REG	BLOCK7 (KEY3) 的第 4 个寄存器内容。	0x010C	只读
EFUSE_RD_KEY3_DATA5_REG	BLOCK7 (KEY3) 的第 5 个寄存器内容。	0x0110	只读

名称	描述	地址	访问
EFUSE_RD_KEY3_DATA6_REG	BLOCK7 (KEY3) 的第 6 个寄存器内容。	0x0114	只读
EFUSE_RD_KEY3_DATA7_REG	BLOCK7 (KEY3) 的第 7 个寄存器内容。	0x0118	只读
EFUSE_RD_KEY4_DATA0_REG	BLOCK8 (KEY4) 的第 0 个寄存器内容。	0x011C	只读
EFUSE_RD_KEY4_DATA1_REG	BLOCK8 (KEY4) 的第 1 个寄存器内容。	0x0120	只读
EFUSE_RD_KEY4_DATA2_REG	BLOCK8 (KEY4) 的第 2 个寄存器内容。	0x0124	只读
EFUSE_RD_KEY4_DATA3_REG	BLOCK8 (KEY4) 的第 3 个寄存器内容。	0x0128	只读
EFUSE_RD_KEY4_DATA4_REG	BLOCK8 (KEY4) 的第 4 个寄存器内容。	0x012C	只读
EFUSE_RD_KEY4_DATA5_REG	BLOCK8 (KEY4) 的第 5 个寄存器内容。	0x0130	只读
EFUSE_RD_KEY4_DATA6_REG	BLOCK8 (KEY4) 的第 6 个寄存器内容。	0x0134	只读
EFUSE_RD_KEY4_DATA7_REG	BLOCK8 (KEY4) 的第 7 个寄存器内容。	0x0138	只读
EFUSE_RD_KEY5_DATA0_REG	BLOCK9 (KEY5) 的第 0 个寄存器内容。	0x013C	只读
EFUSE_RD_KEY5_DATA1_REG	BLOCK9 (KEY5) 的第 1 个寄存器内容。	0x0140	只读
EFUSE_RD_KEY5_DATA2_REG	BLOCK9 (KEY5) 的第 2 个寄存器内容。	0x0144	只读
EFUSE_RD_KEY5_DATA3_REG	BLOCK9 (KEY5) 的第 3 个寄存器内容。	0x0148	只读
EFUSE_RD_KEY5_DATA4_REG	BLOCK9 (KEY5) 的第 4 个寄存器内容。	0x014C	只读
EFUSE_RD_KEY5_DATA5_REG	BLOCK9 (KEY5) 的第 5 个寄存器内容。	0x0150	只读
EFUSE_RD_KEY5_DATA6_REG	BLOCK9 (KEY5) 的第 6 个寄存器内容。	0x0154	只读
EFUSE_RD_KEY5_DATA7_REG	BLOCK9 (KEY5) 的第 7 个寄存器内容。	0x0158	只读
EFUSE_RD_SYS_DATA_PART2_0_REG	BLOCK10 (system) 的第 0 个寄存器内容。	0x015C	只读
EFUSE_RD_SYS_DATA_PART2_1_REG	BLOCK10 (system) 的第 1 个寄存器内容。	0x0160	只读
EFUSE_RD_SYS_DATA_PART2_2_REG	BLOCK10 (system) 的第 2 个寄存器内容。	0x0164	只读
EFUSE_RD_SYS_DATA_PART2_3_REG	BLOCK10 (system) 的第 3 个寄存器内容。	0x0168	只读
EFUSE_RD_SYS_DATA_PART2_4_REG	BLOCK10 (system) 的第 4 个寄存器内容。	0x016C	只读
EFUSE_RD_SYS_DATA_PART2_5_REG	BLOCK10 (system) 的第 5 个寄存器内容。	0x0170	只读
EFUSE_RD_SYS_DATA_PART2_6_REG	BLOCK10 (system) 的第 6 个寄存器内容。	0x0174	只读
EFUSE_RD_SYS_DATA_PART2_7_REG	BLOCK10 (system) 的第 7 个寄存器内容。	0x0178	只读
错误状态寄存器			
EFUSE_RD_REPEAT_ERR0_REG	记录 BLOCK0 参数烧写错误第 0 个寄存器。	0x017C	只读
EFUSE_RD_REPEAT_ERR1_REG	记录 BLOCK0 参数烧写错误第 1 个寄存器。	0x0180	只读
EFUSE_RD_REPEAT_ERR2_REG	记录 BLOCK0 参数烧写错误第 2 个寄存器。	0x0184	只读
EFUSE_RD_REPEAT_ERR3_REG	记录 BLOCK0 参数烧写错误第 3 个寄存器。	0x0188	只读
EFUSE_RD_REPEAT_ERR4_REG	记录 BLOCK0 参数烧写错误第 4 个寄存器。	0x0190	只读
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	只读
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	只读
控制/状态寄存器			
EFUSE_CLK_REG	eFuse 时钟配置寄存器。	0x01C8	读/写
EFUSE_CONF_REG	eFuse 运行模式寄存器。	0x01CC	读/写
EFUSE_CMD_REG	eFuse 命令寄存器。	0x01D4	读/写
EFUSE_DAC_CONF_REG	eFuse 烧写电压控制寄存器。	0x01E8	读/写
EFUSE_STATUS_REG	eFuse 运行状态寄存器。	0x01D0	只读
中断寄存器			

名称	描述	地址	访问
EFUSE_INT_RAW_REG	eFuse 原始中断寄存器。	0x01D8	只读
EFUSE_INT_ST_REG	eFuse 中断状态寄存器。	0x01DC	只读
EFUSE_INT_ENA_REG	eFuse 中断使能寄存器。	0x01E0	读/写
EFUSE_INT_CLR_REG	eFuse 中断清除寄存器。	0x01E4	只写
配置寄存器			
EFUSE_RD_TIM_CONF_REG	eFuse 读取时序参数配置寄存器。	0x01EC	读/写
EFUSE_WR_TIM_CONF0_REG	eFuse 烧写时序参数第 0 个配置寄存器。	0x01F0	读/写
EFUSE_WR_TIM_CONF1_REG	eFuse 烧写时序参数第 1 个配置寄存器。	0x01F4	读/写
EFUSE_WR_TIM_CONF2_REG	eFuse 烧写时序参数第 2 个配置寄存器。	0x01F8	读/写
版本寄存器			
EFUSE_DATE_REG	版本控制寄存器。	0x01FC	读/写

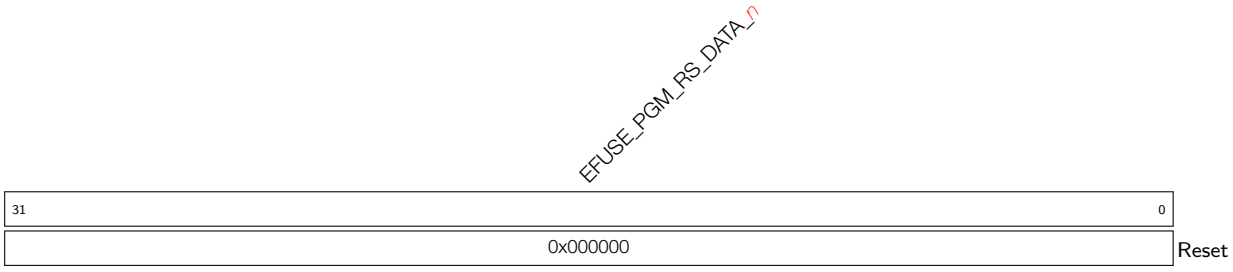
16.6 寄存器

Register 16.1: EFUSE_PGM_DATA*n*_REG (*n*: 0-7) (0x0000+4n*)**



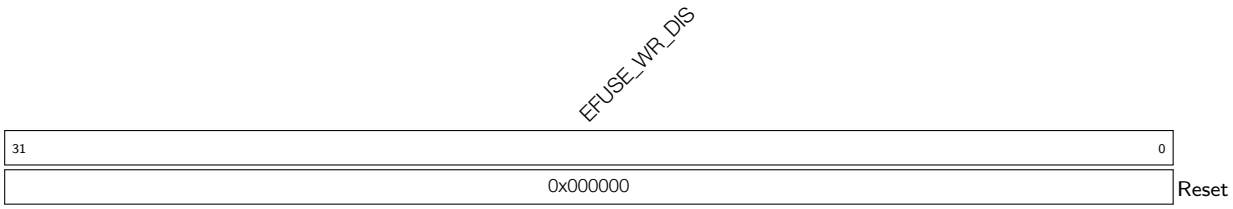
EFUSE_PGM_DATA_*n* 存放待烧写数据的第 *n* 个 32 位数据内容。(读/写)

Register 16.2: EFUSE_PGM_CHECK_VALUE*n*_REG (*n*: 0-2) (0x0020+4n*)**



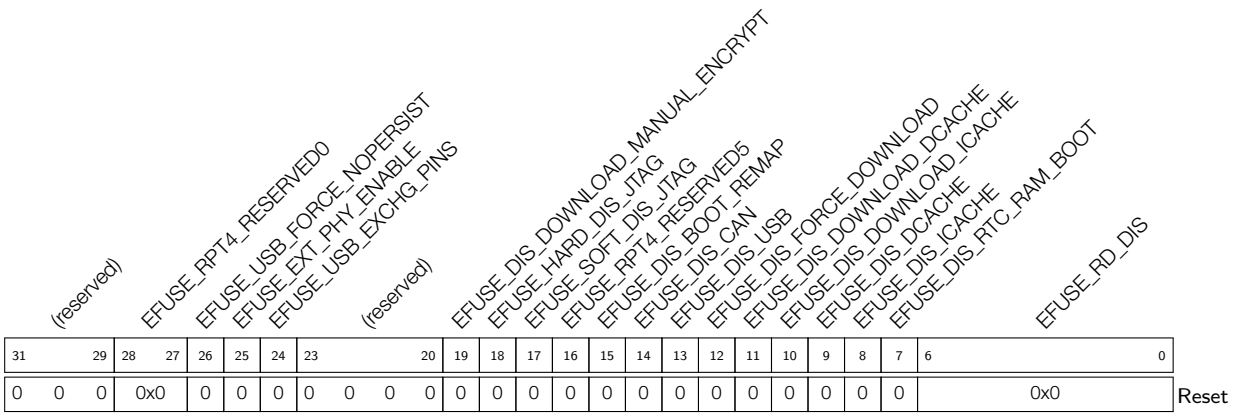
EFUSE_PGM_RS_DATA_*n* 存放待烧写 RS 代码的第 *n* 个 32 位数据内容。(读/写)

Register 16.3: EFUSE_RD_WR_DIS_REG (0x002C)



EFUSE_WR_DIS 置位禁用 eFuse 烧写。(只读)

Register 16.4: EFUSE_RD_REPEAT_DATA0_REG (0x0030)



EFUSE_RD_DIS 置位禁止软件读取 eFuse Block4-10 的内容。(只读)

EFUSE_DIS_RTC_RAM_BOOT 置位禁止从 RTC RAM 启动。(只读)

EFUSE_DIS_ICACHE 置位禁用 ICache。(只读)

EFUSE_DIS_DCACHE 置位禁用 DCache。(只读)

EFUSE_DIS_DOWNLOAD_ICACHE 置位在下载模式下关闭 ICache。(只读)

EFUSE_DIS_DOWNLOAD_DCACHE 置位在下载模式下关闭 DCache。(只读)

EFUSE_DIS_FORCE_DOWNLOAD 置位禁止强制芯片进入下载模式。(只读)

EFUSE_DIS_USB 置位关闭 USB OTG 功能。(只读)

EFUSE_DIS_CAN 置位关闭 TWAI 控制器功能。(只读)

EFUSE_DIS_BOOT_REMAP 置位禁止 RAM 空间映射到 ROM 空间。(只读)

EFUSE_RPT4_RESERVED5 保留（采用 4 备份编码)。(只读)

EFUSE_SOFT_DIS_JTAG 软关断 JTAG 功能，用户还可以通过 HAMC 模块再次打开 JTAG。(只读)

EFUSE_HARD_DIS_JTAG 硬关断 JTAG 功能，永久关断。(只读)

EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT 置位在 download boot 模式下关闭 flash 加密功能。(只读)

EFUSE_USB_EXCHG_PINS 置位交换 USB D+ 和 D- 管脚。(只读)

EFUSE_EXT_PHY_ENABLE 置位使能外部 USB PHY。(只读)

EFUSE_USB_FORCE_NOPERSIST 置位强制设置 USB BAVLID 为 1。(只读)

EFUSE_RPT4_RESERVED0 保留（采用 4 备份编码)。(只读)

Register 16.5: EFUSE_RD_REPEAT_DATA1_REG (0x0034)

EFUSE_VDD_SPI_XPD 当 EFUSE_VDD_SPI_FORCE 为 1 时，控制 SPI 调节器上电。(只读)

EFUSE_VDD_SPI_TIEH 当 EFUSE_VDD_SPI_FORCE 为 1 时，选择 VDD_SPI 电压。0：VDD_SPI 连接 1.8 V LDO；1：VDD_SPI 连接 VDD_RTC_IO。(只读)

EFUSE_VDD_SPI_FORCE 使用 EFUSE_VDD_SPI_XPD 和 EFUSE_VDD_SPI_TIEH 配置 VDD_SPI。(只读)

EFUSE_WDT_DELAY_SEL 选择 RTC 看门狗超时阈值。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。(只读)

EFUSE_SPI_BOOT_CRYPT_CNT 置位使能 SPI boot 加解密。奇数个 1：使能；偶数个 1：禁能。(只读)

EFUSE_SECURE_BOOT_KEY_REVOKE0 置位使能撤销第一个安全启动密钥。(只读)

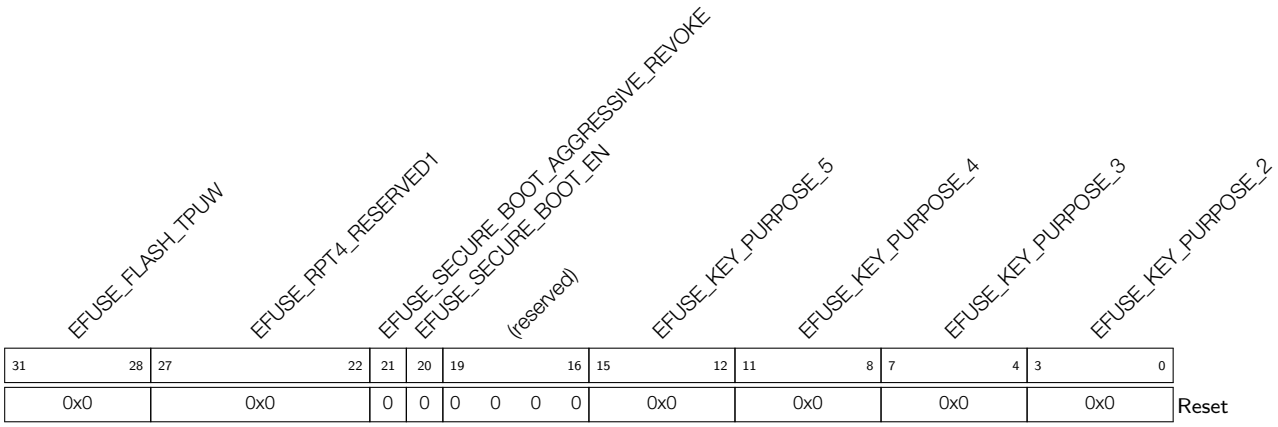
EFUSE_SECURE_BOOT_KEY_REVOKE1 置位使能撤销第二个安全启动密钥。(只读)

EFUSE_SECURE_BOOT_KEY_REVOKE2 置位使能撤销第三个安全启动密钥。(只读)

EFUSE_KEY_PURPOSE_0 KEY0 purpose，详见表 55：密钥用途数值对应的含义。(只读)

EFUSE_KEY_PURPOSE_1 KEY1 purpose，详见表 55：密钥用途数值对应的含义。(只读)

Register 16.6: EFUSE_RD_REPEAT_DATA2_REG (0x0038)



EFUSE_KEY_PURPOSE_2 KEY2 purpose，详见表 55：密钥用途数值对应的含义。(只读)

EFUSE_KEY_PURPOSE_3 KEY3 purpose，详见表 55：密钥用途数值对应的含义。(只读)

EFUSE_KEY_PURPOSE_4 KEY4 purpose，详见表 55：密钥用途数值对应的含义。(只读)

EFUSE_KEY_PURPOSE_5 KEY5 purpose，详见表 55：密钥用途数值对应的含义。(只读)

EFUSE_SECURE_BOOT_EN 置位使能安全启动。(只读)

EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE 置位使能密钥失效的激进策略。(只读)

EFUSE_RPT4_RESERVED1 保留（采用 4 备份编码)。(只读)

EFUSE_FLASH_TPUW 配置上电后 flash 等待时间，单位为 ms/2，值为 15 时，等待时间为 7.5 ms。(只读)

Register 16.7: EFUSE_RD_REPEAT_DATA3_REG (0x003C)

EFUSE_DIS_DOWNLOAD_MODE 置位关闭下载模式。(只读)

EFUSE_DIS_LEGACY_SPI_BOOT 置位关闭 Legacy SPI boot 模式。(只读)

EFUSE_UART_PRINT_CHANNEL 选择打印 boot 信息的 UART 通道。0：UART0；1：UART1。(只读)

EFUSE_RPT4_RESERVED3 保留（采用 4 备份编码)。(只读)

EFUSE_DIS_USB_DOWNLOAD_MODE 置位在 UART download boot 模式下关闭 USB 功能。(只读)

EFUSE_ENABLE_SECURITY_DOWNLOAD 置位使能安全 UART 下载模式（仅支持读写 flash)(只读)

EFUSE_UART_PRINT_CONTROL 控制 UART 打印方式。00：强制打印 01：由 GPIO 46 控制，低电平打印 10：由 GPIO 46 控制，高电平打印 11：强制关闭打印。(只读)

EFUSE_PIN_POWER_SELECTION SPI flash 启动时选择 GPIO33-GPIO37 的电源。0：VDD3P3_CPU；1：VDD_SPI。(只读)

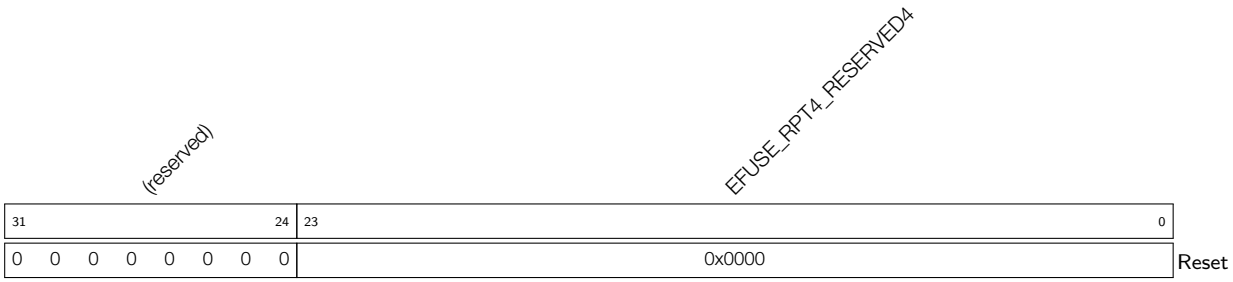
EFUSE_FLASH_TYPE Flash 类型。0：4 根数据线；1：8 根数据线。(只读)

EFUSE_FORCE_SEND_RESUME 置位强制 ROM 代码在 SPI 启动过程中发送恢复指令。(只读)

EFUSE_SECURE_VERSION 表明 IDF 安全版本（用于 ESP-IDF 的防回滚功能)。(只读)

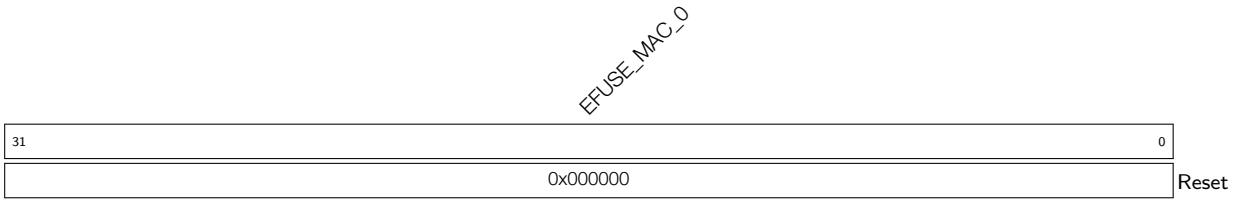
EFUSE_RPT4_RESERVED2 保留（采用 4 备份编码)。(只读)

Register 16.8: EFUSE_RD_REPEAT_DATA4_REG (0x0040)



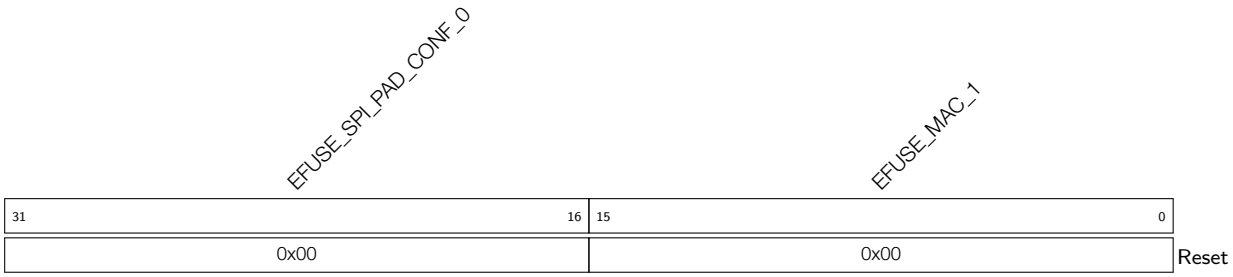
EFUSE_RPT4_RESERVED4 保留（采用 4 备份编码）。(只读)

Register 16.9: EFUSE_RD_MAC_SPI_SYS_0_REG (0x0044)



EFUSE_MAC_0 存储 MAC 地址低 32 位的内容。(只读)

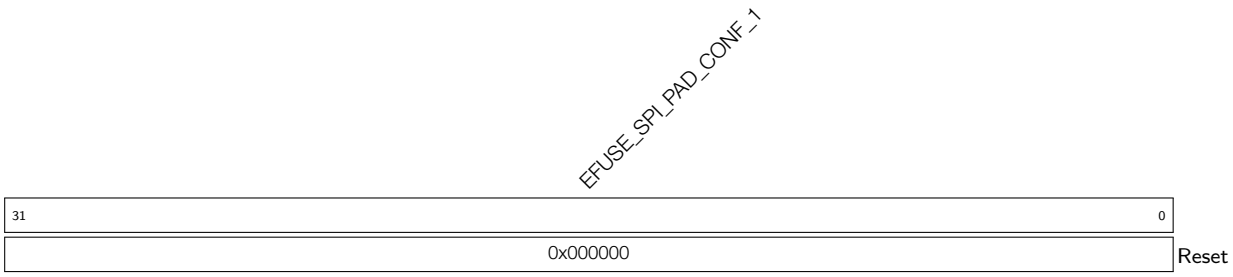
Register 16.10: EFUSE_RD_MAC_SPI_SYS_1_REG (0x0048)



EFUSE_MAC_1 存储 MAC 地址高 16 位的内容。(只读)

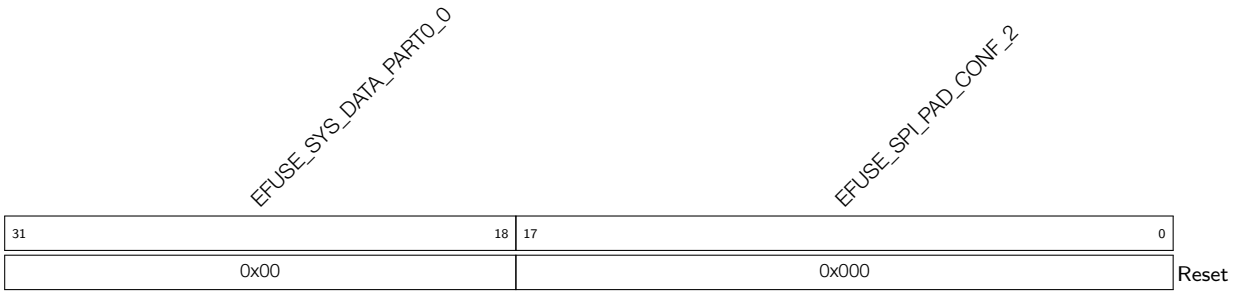
EFUSE_SPI_PAD_CONF_0 存储 SPI_PAD_CONF 第 0 部分的内容。(只读)

Register 16.11: EFUSE_RD_MAC_SPI_SYS_2_REG (0x004C)



EFUSE_SPI_PAD_CONF_1 存储 SPI_PAD_CONF 第 1 部分的内容。（只读）

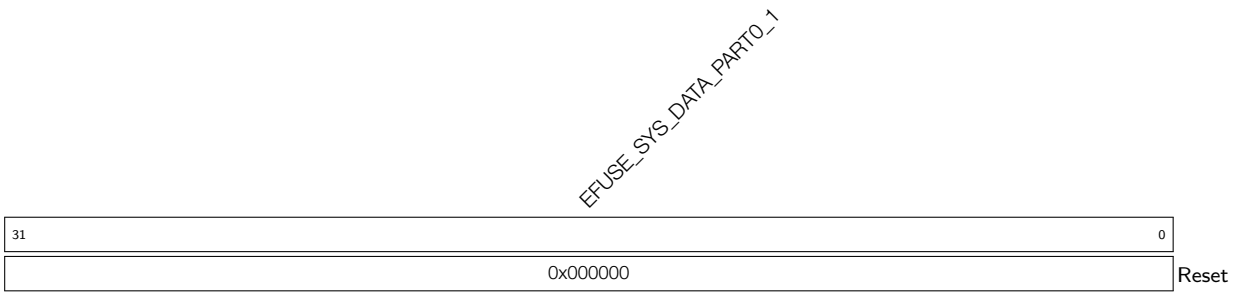
Register 16.12: EFUSE_RD_MAC_SPI_SYS_3_REG (0x0050)



EFUSE_SPI_PAD_CONF_2 存储 SPI_PAD_CONF 第 2 部分的内容。（只读）

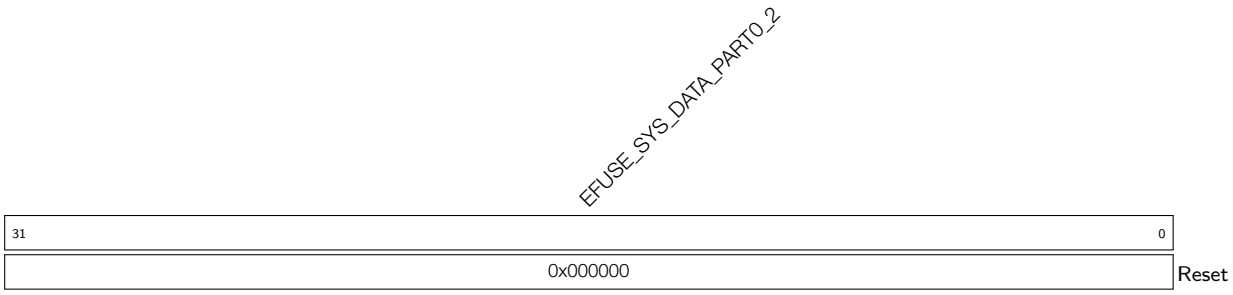
EFUSE_SYS_DATA_PART0_0 存储系统数据第 0 部分的第 0 部分的内容。（只读）

Register 16.13: EFUSE_RD_MAC_SPI_SYS_4_REG (0x0054)



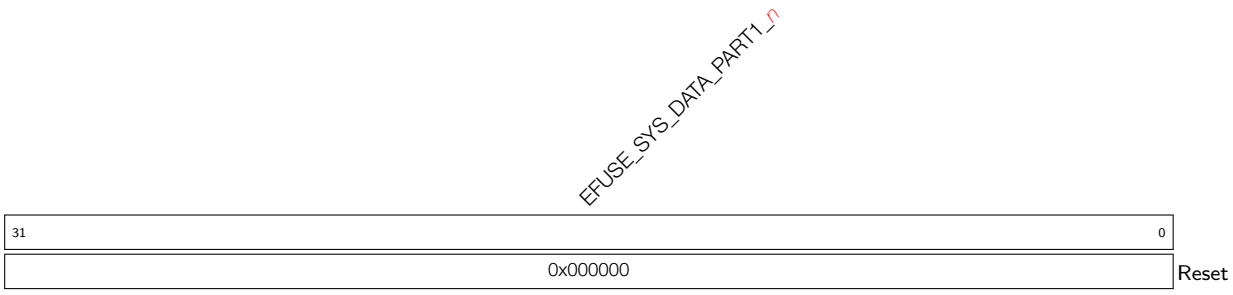
EFUSE_SYS_DATA_PART0_1 存储系统数据第 0 部分的第 1 部分的内容。（只读）

Register 16.14: EFUSE_RD_MAC_SPI_SYS_5_REG (0x0058)



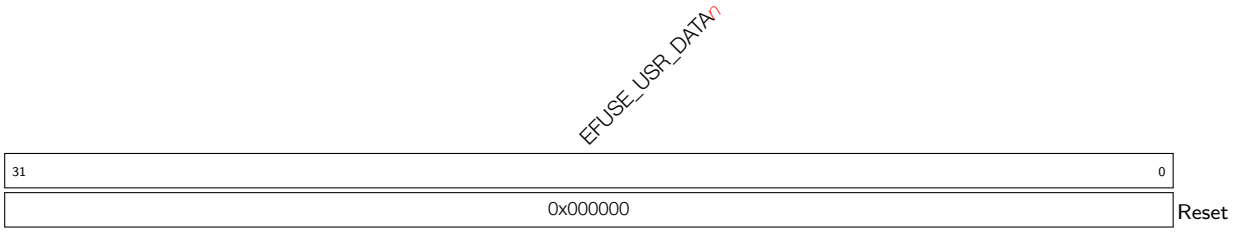
EFUSE_SYS_DATA_PART0_2 存储系统数据第 0 部分的第 2 部分的内容。(只读)

Register 16.15: EFUSE_RD_SYS_DATA_PART1_*n*_REG (*n*: 0-7) (0x005C+4n*)**



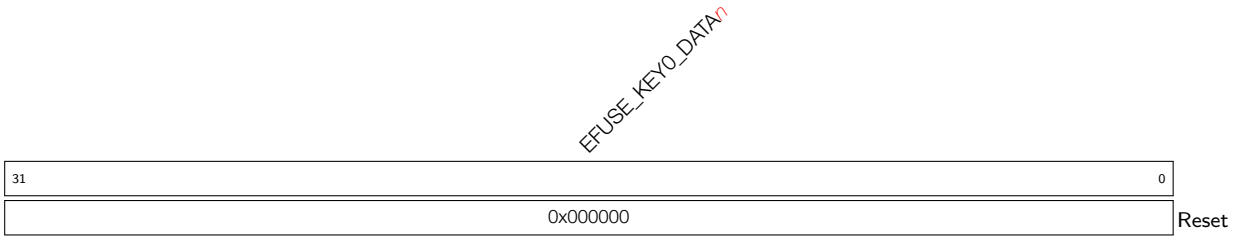
EFUSE_SYS_DATA_PART1_*n* 存储系统数据第 1 部分的第 *n* 个 32 位的内容。(只读)

Register 16.16: EFUSE_RD_USR_DATA*n*_REG (*n*: 0-7) (0x007C+4n*)**



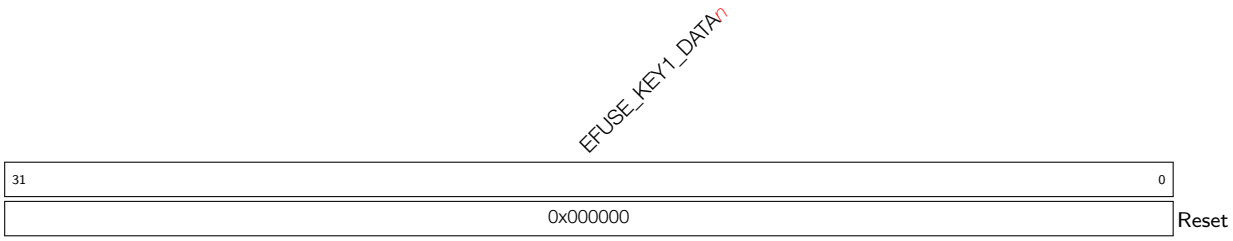
EFUSE_USR_DATA*n* 存储 BLOCK3 (user) 的第 *n* 个 32 位的内容。(只读)

Register 16.17: EFUSE_RD_KEY0_DATA*n*_REG (*n*: 0-7) (0x009C+4n*)**



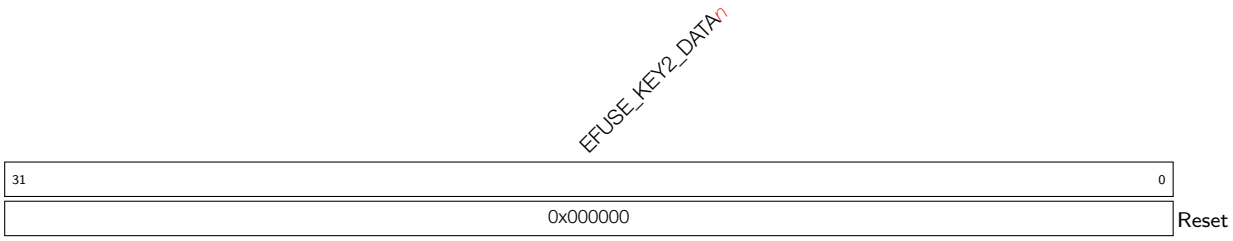
EFUSE_KEY0_DATA*n* 存储 KEY0 的第 *n* 个 32 位的内容。（只读）

Register 16.18: EFUSE_RD_KEY1_DATA*n*_REG (*n*: 0-7) (0x00BC+4n*)**



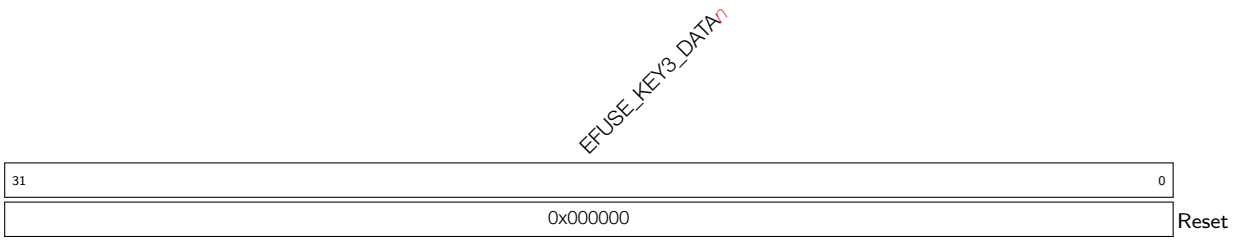
EFUSE_KEY1_DATA*n* 存储 KEY1 的第 *n* 个 32 位的内容。（只读）

Register 16.19: EFUSE_RD_KEY2_DATA*n*_REG (*n*: 0-7) (0x00DC+4n*)**



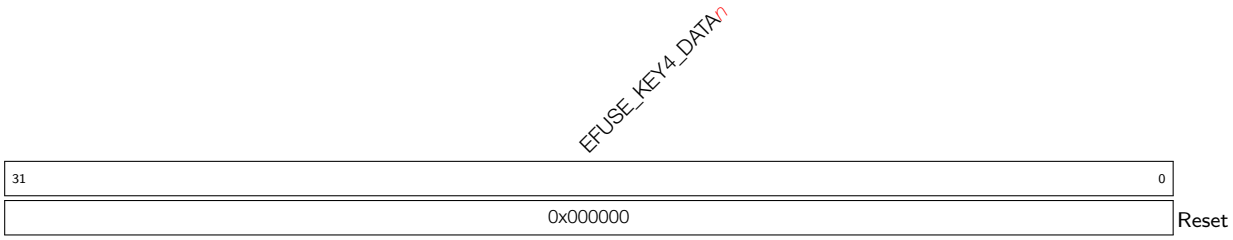
EFUSE_KEY2_DATA*n* 存储 KEY2 的第 *n* 个 32 位的内容。（只读）

Register 16.20: EFUSE_RD_KEY3_DATA*n*_REG (*n*: 0-7) (0x00FC+4n*)**



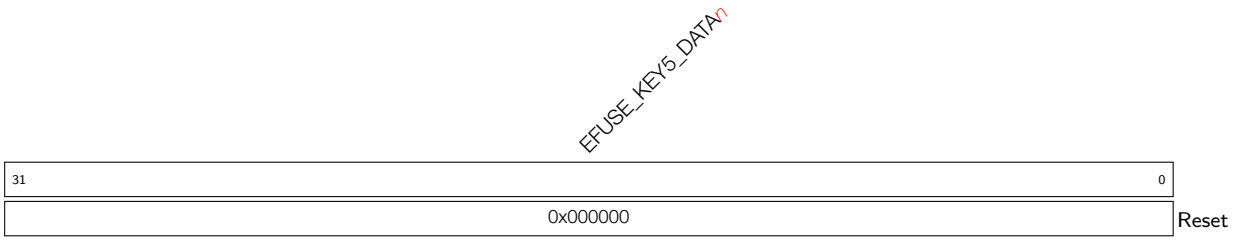
EFUSE_KEY3_DATA*n* 存储 KEY3 的第 *n* 个 32 位的内容。（只读）

Register 16.21: EFUSE_RD_KEY4_DATA*n*_REG (*n*: 0-7) (0x011C+4n*)**



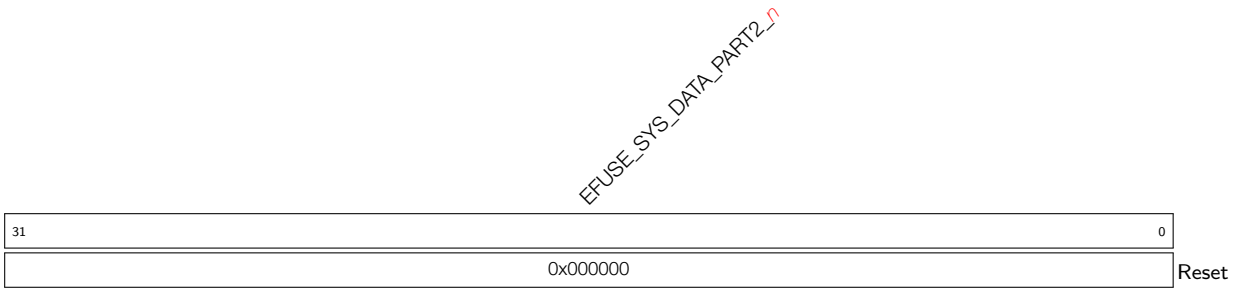
EFUSE_KEY4_DATA*n* 存储 KEY4 的第 *n* 个 32 位的内容。（只读）

Register 16.22: EFUSE_RD_KEY5_DATA*n*_REG (*n*: 0-7) (0x013C+4n*)**



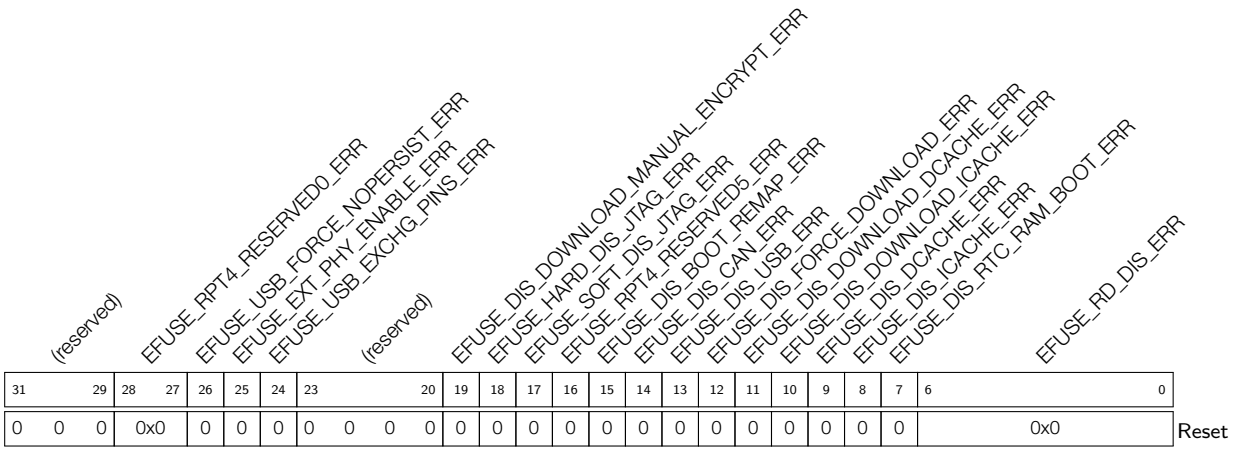
EFUSE_KEY5_DATA*n* 存储 KEY5 的第 *n* 个 32 位的内容。（只读）

Register 16.23: EFUSE_RD_SYS_DATA_PART2_*n*_REG (*n*: 0-7) (0x015C+4n*)**



EFUSE_SYS_DATA_PART2_*n* 存储系统数据第 2 部分的第 *n* 个 32 位的内容。（只读）

Register 16.24: EFUSE_RD_REPEAT_ERR0_REG (0x017C)



EFUSE_RD_DIS_ERR 若该参数中任意位为 1，表明对应 EFUSE_RD_DIS 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_RTC_RAM_BOOT_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_RTC_RAM_BOOT 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_ICACHE_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_ICACHE 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_DCACHE_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_DCACHE 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_DOWNLOAD_ICACHE_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_DOWNLOAD_ICACHE 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_DOWNLOAD_DCACHE_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_DOWNLOAD_DCACHE 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_FORCE_DOWNLOAD_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_FORCE_DOWNLOAD 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_USB_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_USB 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_CAN_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_CAN 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_BOOT_REMAP_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_BOOT_REMAP 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_RPT4_RESERVED5_ERR 若该参数中任意比特为 1，表明对应 EFUSE_RPT4_RESERVED5 中该比特的四备份不一致，参数不可靠。(只读)

EFUSE_SOFT_DIS_JTAG_ERR 若该参数中任意位为 1，表明对应 EFUSE_SOFT_DIS_JTAG 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_HARD_DIS_JTAG_ERR 若该参数中任意位为 1，表明对应 EFUSE_HARD_DIS_JTAG 中该位的四备份不一致，参数不可靠。(只读)

寄存器描述下一页继续。

Register 16.24: EFUSE_RD_REPEAT_ERR0_REG (0x017C)

继上一页寄存器描述。

EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT 中该位的四备份不一致，参数不可靠。（只读）

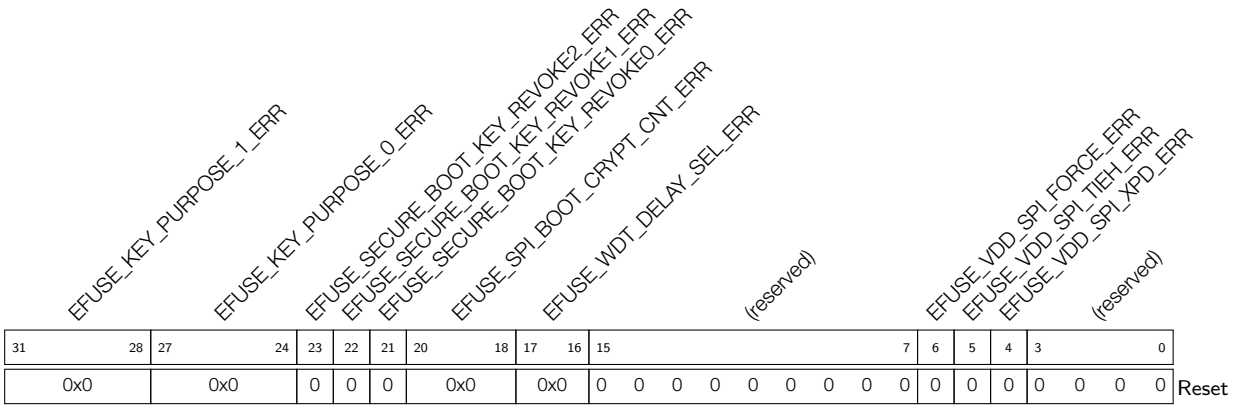
EFUSE_USB_EXCHG_PINS_ERR 若该参数中任意位为 1，表明对应 EFUSE_USB_EXCHG_PINS 中该位的四备份不一致，参数不可靠。（只读）

EFUSE_EXT_PHY_ENABLE_ERR 若该参数中任意位为 1，表明对应 EFUSE_EXT_PHY_ENABLE 中该位的四备份不一致，参数不可靠。（只读）

EFUSE_USB_FORCE_NOPERSIST_ERR 若该参数中任意位为 1，表明对应 EFUSE_USB_FORCE_B 中该位的四备份不一致，参数不可靠。（只读）

EFUSE_RPT4_RESERVED0_ERR 若该参数中任意位为 1，表明对应 EFUSE_RPT4_RESERVED0 中该位的四备份不一致，参数不可靠。（只读）

Register 16.25: EFUSE_RD_REPEAT_ERR1_REG (0x0180)



EFUSE_VDD_SPI_XPD_ERR 若该参数中任意位为 1，表明对应 EFUSE_VDD_SPI_XPD 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_VDD_SPI_TIEH_ERR 若该参数中任意位为 1，表明对应 EFUSE_VDD_SPI_TIEH 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_VDD_SPI_FORCE_ERR 若该参数中任意位为 1，表明对应 EFUSE_VDD_SPI_FORCE 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_WDT_DELAY_SEL_ERR 若该参数中任意位为 1，表明对应 EFUSE_WDT_DELAY_SEL 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_SPI_BOOT_CRYPT_CNT_ERR 若该参数中任意位为 1，表明对应 EFUSE_SPI_BOOT_CRYPT_CNT 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_SECURE_BOOT_KEY_REVOKE0_ERR 若该参数中任意位为 1，表明对应 EFUSE_SECURE_BOOT_KEY_REVOKE0 中该位的四备份不一致，参数不可靠。(只读)

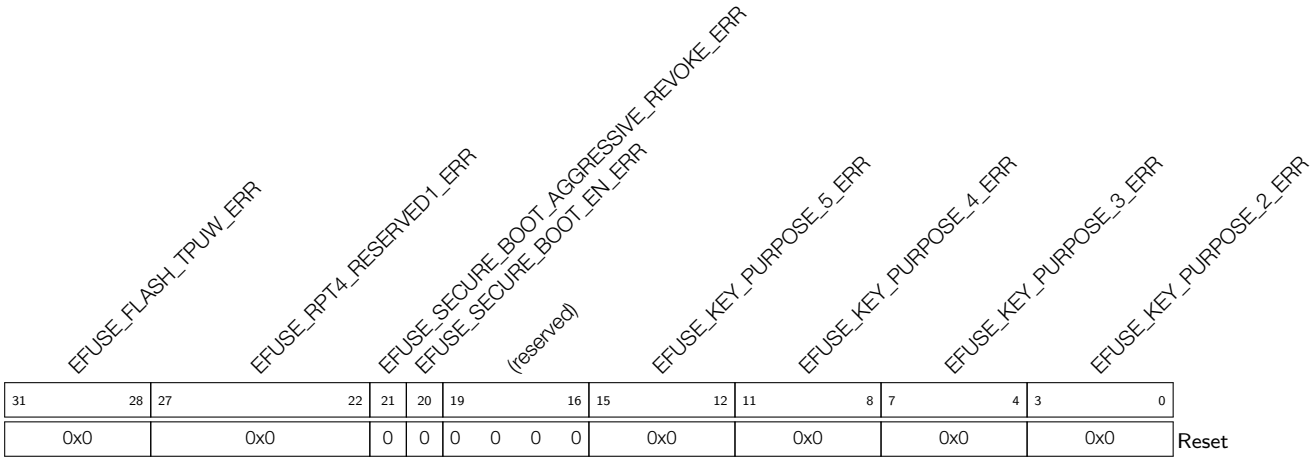
EFUSE_SECURE_BOOT_KEY_REVOKE1_ERR 若该参数中任意位为 1，表明对应 EFUSE_SECURE_BOOT_KEY_REVOKE1 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_SECURE_BOOT_KEY_REVOKE2_ERR 若该参数中任意位为 1，表明对应 EFUSE_SECURE_BOOT_KEY_REVOKE2 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_KEY_PURPOSE_0_ERR 若该参数中任意位为 1，表明对应 EFUSE_KEY_PURPOSE_0 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_KEY_PURPOSE_1_ERR 若该参数中任意位为 1，表明对应 EFUSE_KEY_PURPOSE_1 中该位的四备份不一致，参数不可靠。(只读)

Register 16.26: EFUSE_RD_REPEAT_ERR2_REG (0x0184)



EFUSE_KEY_PURPOSE_2_ERR 若该参数中任意位为 1，表明对应 EFUSE_KEY_PURPOSE_2 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_KEY_PURPOSE_3_ERR 若该参数中任意位为 1，表明对应 EFUSE_KEY_PURPOSE_3 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_KEY_PURPOSE_4_ERR 若该参数中任意位为 1，表明对应 EFUSE_KEY_PURPOSE_4 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_KEY_PURPOSE_5_ERR 若该参数中任意位为 1，表明对应 EFUSE_KEY_PURPOSE_5 中该位的四备份不一致，参数不可靠。(只读)

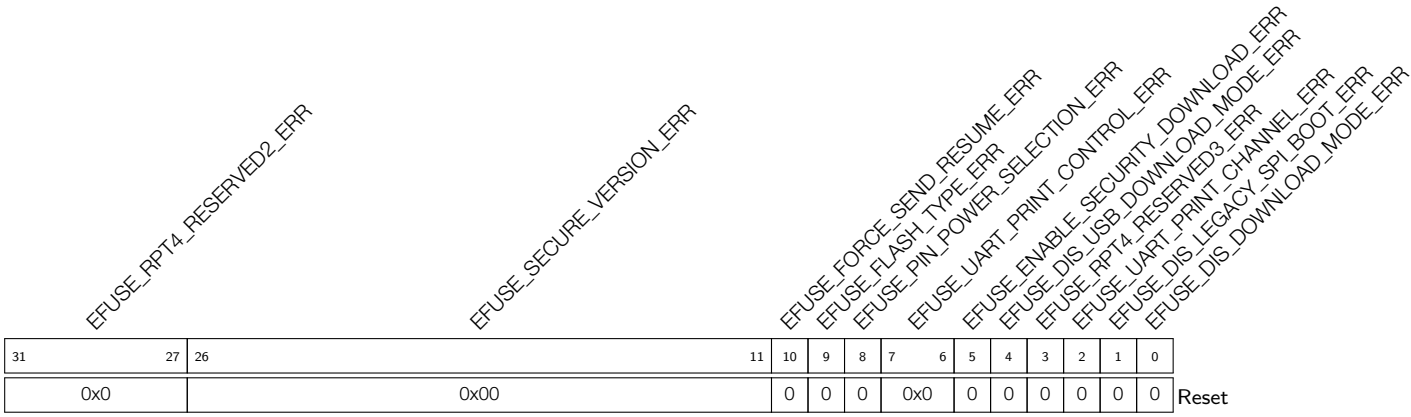
EFUSE_SECURE_BOOT_EN_ERR 若该参数中任意位为 1，表明对应 EFUSE_SECURE_BOOT_EN 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE_ERR 若该参数中任意位为 1，表明对应 EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_RPT4_RESERVED1_ERR 若该参数中任意位为 1，表明对应 EFUSE_RPT4_RESERVED1 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_FLASH_TPUW_ERR 若该参数中任意位为 1，表明对应 EFUSE_FLASH_TPUM 中该位的四备份不一致，参数不可靠。(只读)

Register 16.27: EFUSE_RD_REPEAT_ERR3_REG (0x0188)



EFUSE_DIS_DOWNLOAD_MODE_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_DOWNLOAD_MODE 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_LEGACY_SPI_BOOT_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_LEGACY_SPI_BOOT 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_UART_PRINT_CHANNEL_ERR 若该参数中任意位为 1，表明对应 EFUSE_UART_PRINT_CHANNEL 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_RPT4_RESERVED3_ERR 若该参数中任意位为 1，表明对应 EFUSE_RPT4_RESERVED3 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_DIS_USB_DOWNLOAD_MODE_ERR 若该参数中任意位为 1，表明对应 EFUSE_DIS_USB_DOWNLOAD_MODE 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_ENABLE_SECURITY_DOWNLOAD_ERR 若该参数中任意位为 1，表明对应 EFUSE_ENABLE_SECURITY_DOWNLOAD 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_UART_PRINT_CONTROL_ERR 若该参数中任意位为 1，表明对应 EFUSE_UART_PRINT_CONTROL 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_PIN_POWER_SELECTION_ERR 若该参数中任意位为 1，表明对应 EFUSE_PIN_POWER_SELECTION 中该位的四备份不一致，参数不可靠。(只读)

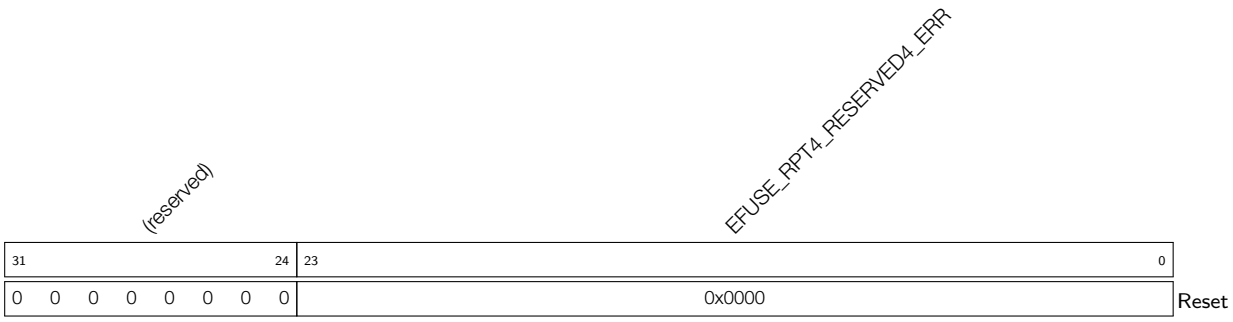
EFUSE_FLASH_TYPE_ERR 若该参数中任意位为 1，表明对应 EFUSE_FLASH_TYPE 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_FORCE_SEND_RESUME_ERR 若该参数中任意位为 1，表明对应 EFUSE_FORCE_SEND_RESUME 中该位的四备份不一致，参数不可靠。(只读)

EFUSE_SECURE_VERSION_ERR 若该参数中任意位为 1，表明对应 EFUSE_SECURE_VERSION 中该位的四备份不一致，参数不可靠。(只读)

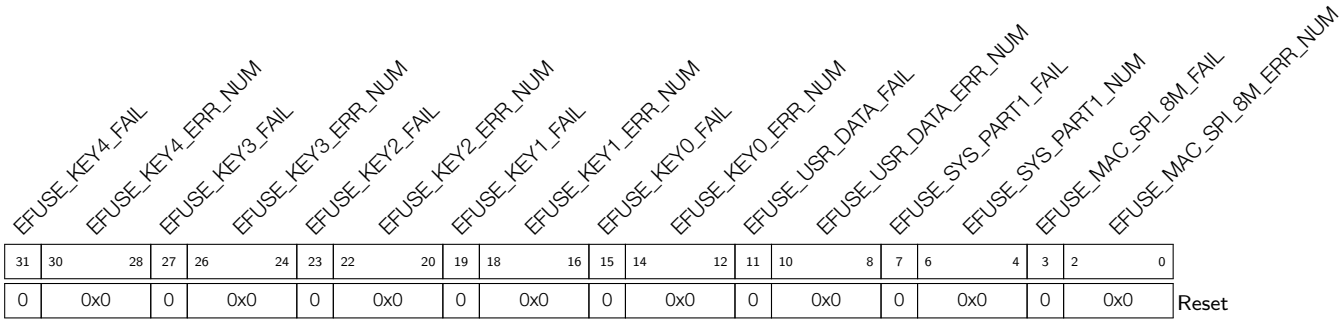
EFUSE_RPT4_RESERVED2_ERR 若该参数中任意位为 1，表明对应 EFUSE_RPT4_RESERVED2 中该位的四备份不一致，参数不可靠。(只读)

Register 16.28: EFUSE_RD_REPEAT_ERR4_REG (0x0190)



EFUSE_RPT4_RESERVED4_ERR 若该参数中任意位为 1，表明对应 EFUSE_RPT4_RESERVED4 中该位的四备份不一致，参数不可靠。(只读)

Register 16.29: EFUSE_RD_RS_ERR0_REG (0x01C0)



EFUSE_MAC_SPI_8M_ERR_NUM 指示 BLOCK1 中的错误字节的个数。(只读)

EFUSE_MAC_SPI_8M_FAIL 0：代表没有烧写错误，BLOCK1 里的数据是可靠的；1：代表烧写 BLOCK1 失败，错误字节数超过 5 个。(只读)

EFUSE_SYS_PART1_NUM 指示 BLOCK2 中的错误字节的个数。(只读)

EFUSE_SYS_PART1_FAIL 0：代表没有烧写错误，BLOCK2 里的数据是可靠的；1：代表烧写 BLOCK2 失败，错误字节数超过 5 个。(只读)

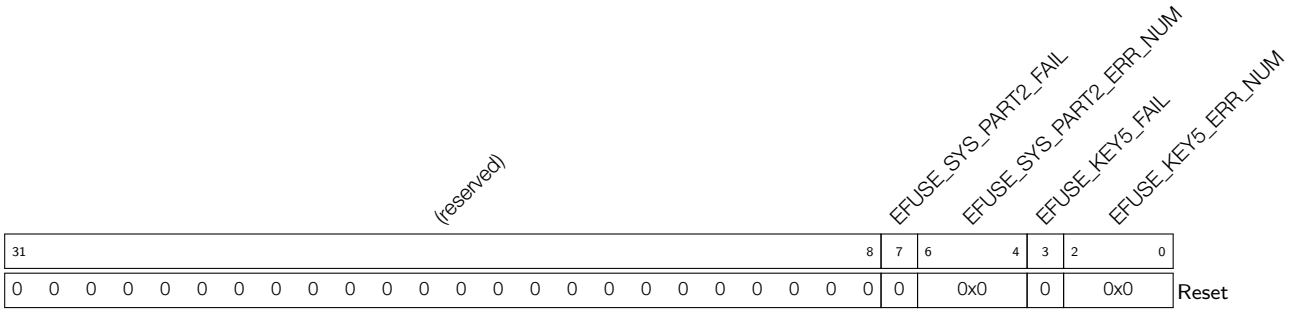
EFUSE_USR_DATA_ERR_NUM 指示 BLOCK3 中的错误字节的个数。(只读)

EFUSE_USR_DATA_FAIL 0：代表没有烧写错误，BLOCK3 里的数据是可靠的；1：代表烧写 BLOCK3 失败，错误字节数超过 5 个。(只读)

EFUSE_KEY*n*_ERR_NUM 指示 KEY*n* 中的错误字节的个数。(只读)

EFUSE_KEY*n*_FAIL 0：代表没有烧写错误，key*n* 数据是可靠的；1：代表 key*n* 烧写失败，错误字节数超过 5 个。(只读)

Register 16.30: EFUSE_RD_RS_ERR1_REG (0x01C4)



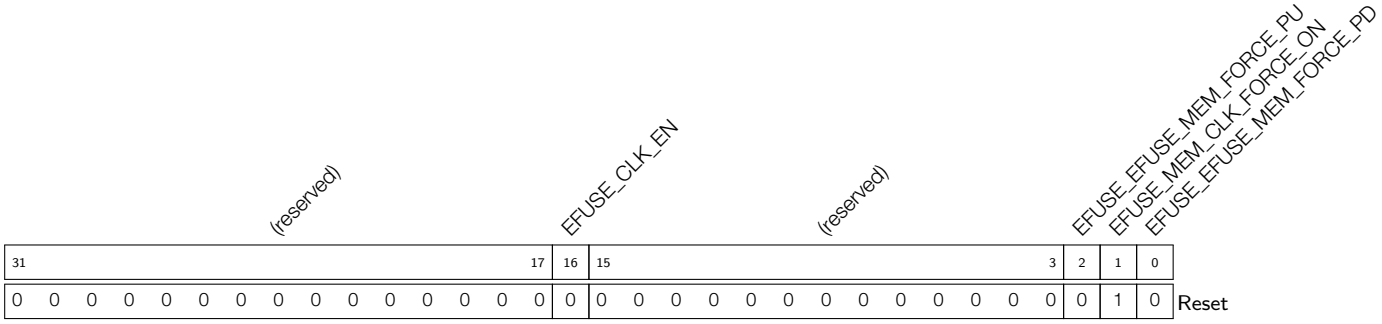
EFUSE_KEY5_ERR_NUM 指示 KEY5 中的错误字节的个数。(只读)

EFUSE_KEY5_FAIL 0：代表没有烧写错误，KEY5 里的数据是可靠的；1：代表烧写 KEY5 失败，错误字节数超过 5 个。(只读)

EFUSE_SYS_PART2_ERR_NUM 指示 BLOCK10 中的错误字节的个数。(只读)

EFUSE_SYS_PART2_FAIL 0：代表没有烧写错误，BLOCK10 里的数据是可靠的；1：代表烧写 BLOCK10 失败，错误字节数超过 5 个。(只读)

Register 16.31: EFUSE_CLK_REG (0x01C8)



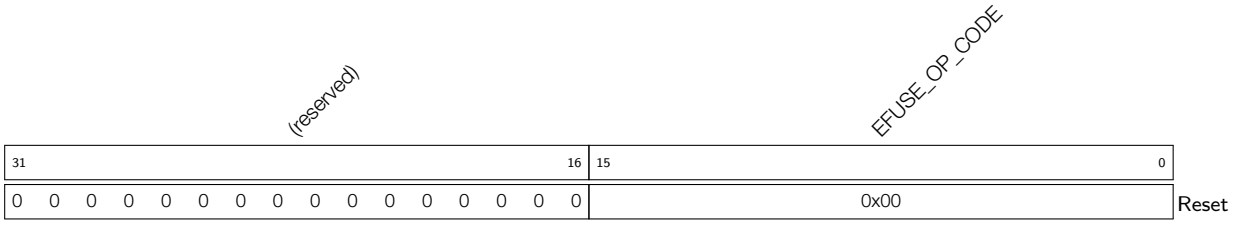
EFUSE_EFUSE_MEM_FORCE_PD 置位强制使 eFuse SRAM 进入低功耗模式。(读/写)

EFUSE_MEM_CLK_FORCE_ON 置位强制激活 eFuse SRAM 的时钟信号。(读/写)

EFUSE_EFUSE_MEM_FORCE_PU 置位强制使 eFuse SRAM 进入工作模式。(读/写)

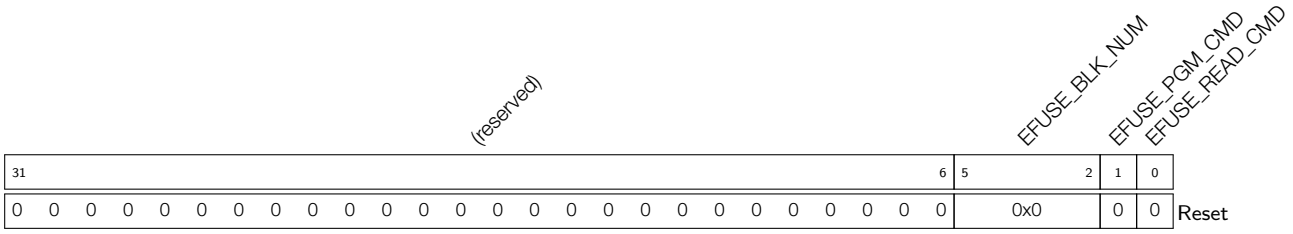
EFUSE_CLK_EN 置位强制使能 eFuse memory 的时钟信号。(读/写)

Register 16.32: EFUSE_CONF_REG (0x01CC)



EFUSE_OP_CODE 0x5A5A：运行烧写指令；0x5AA5：运行读取指令。(读/写)

Register 16.33: EFUSE_CMD_REG (0x01D4)

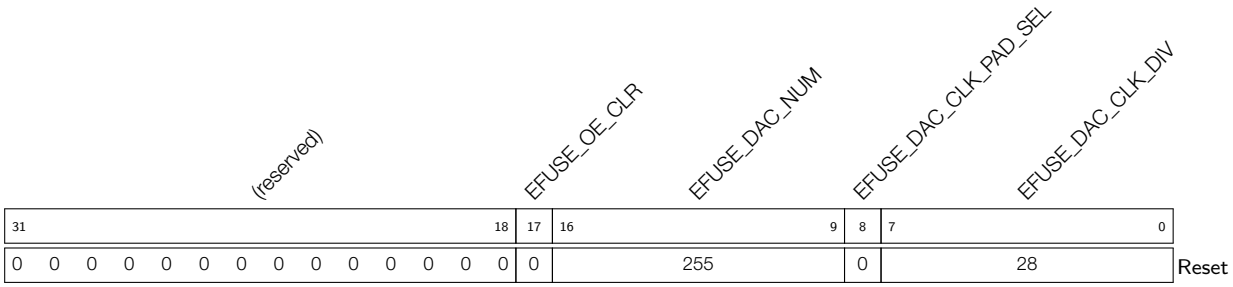


EFUSE_READ_CMD 置位发送读取指令。(读/写)

EFUSE_PGM_CMD 置位发送烧写指令。(读/写)

EFUSE_BLK_NUM 表明烧写哪个块，值 0-10 分别对应 BLOCK0-10。(读/写)

Register 16.34: EFUSE_DAC_CONF_REG (0x01E8)



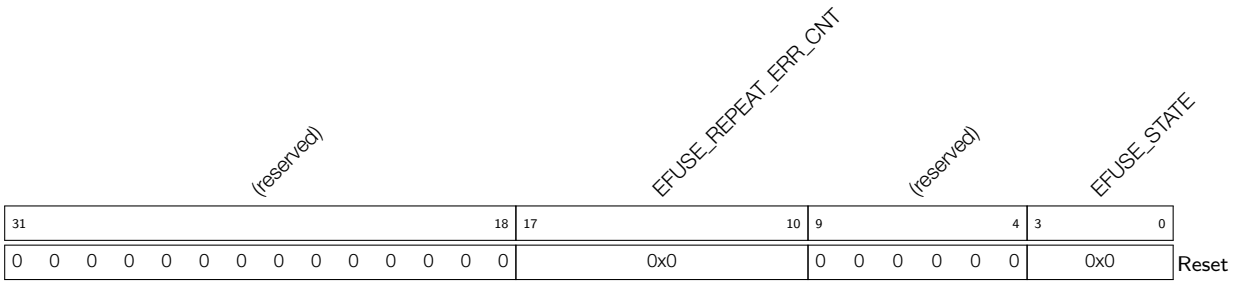
EFUSE_DAC_CLK_DIV 烧写电压的爬升时钟分频系数。(读/写)

EFUSE_DAC_CLK_PAD_SEL 无关项。(读/写)

EFUSE_DAC_NUM 烧写供电的上升周期。(读/写)

EFUSE_OE_CLR 降低烧写电压的供电能力。(读/写)

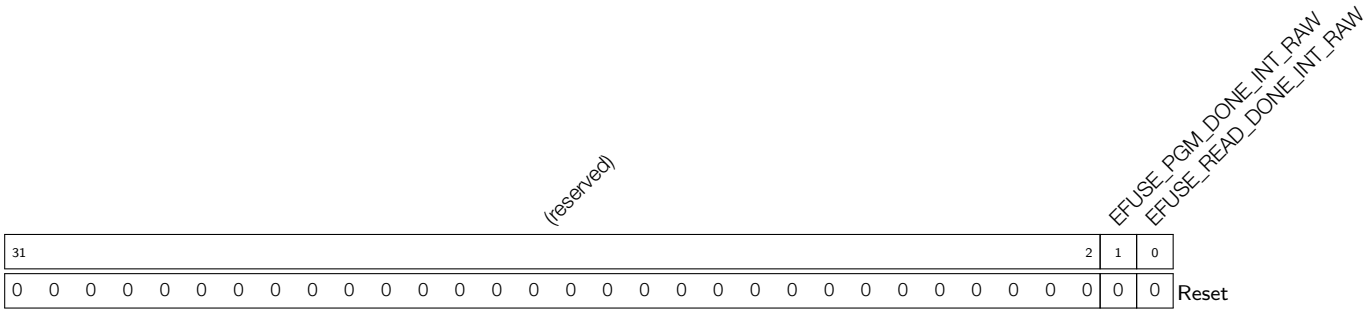
Register 16.35: EFUSE_STATUS_REG (0x01D0)



EFUSE_STATE 表明 eFuse 状态机所处的状态。(只读)

EFUSE_REPEAT_ERR_CNT 表明烧写 BLOCK0 时的错误位的个数。(只读)

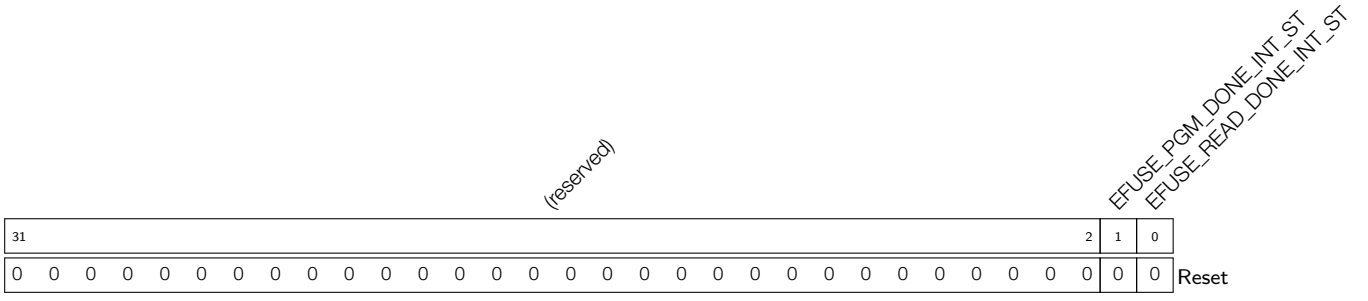
Register 16.36: EFUSE_INT_RAW_REG (0x01D8)



EFUSE_READ_DONE_INT_RAW 读取完成中断的原始中断状态位。(只读)

EFUSE_PGM_DONE_INT_RAW 烧写完成中断的原始中断状态位。(只读)

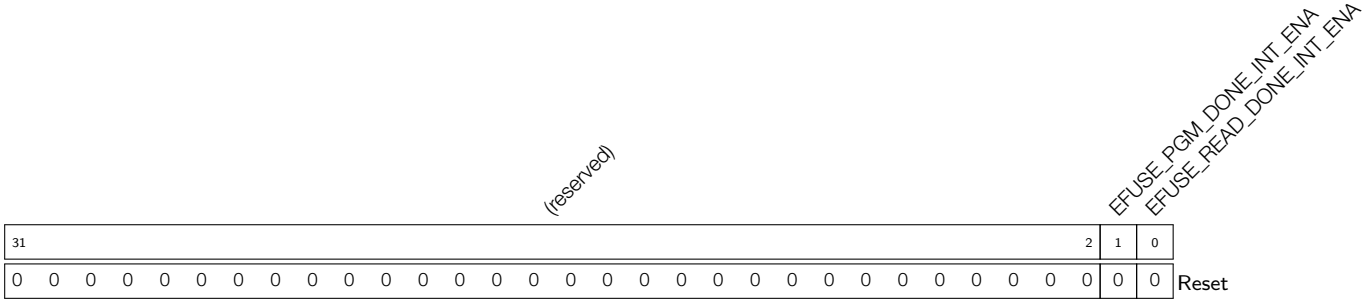
Register 16.37: EFUSE_INT_ST_REG (0x01DC)



EFUSE_READ_DONE_INT_ST 读取完成中断的状态位。(只读)

EFUSE_PGM_DONE_INT_ST 烧写完成中断的状态位。(只读)

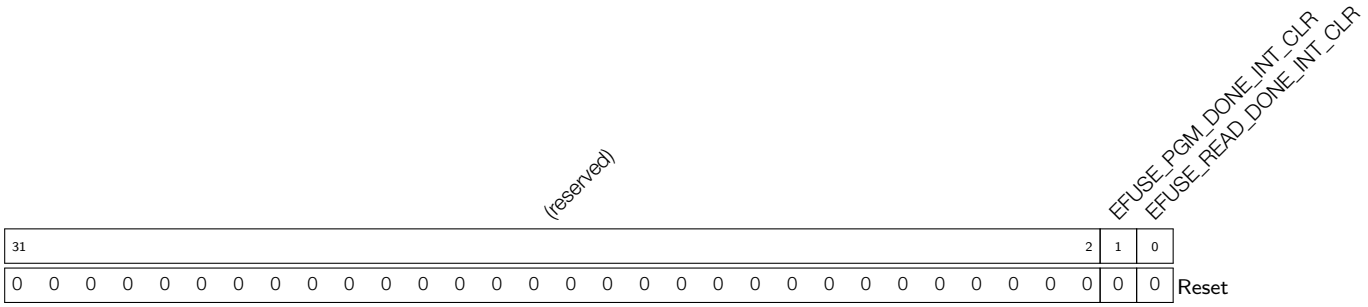
Register 16.38: EFUSE_INT_ENA_REG (0x01E0)



EFUSE_READ_DONE_INT_ENA 读取完成中断的使能位。(读/写)

EFUSE_PGM_DONE_INT_ENA 烧写完成中断的使能位。(读/写)

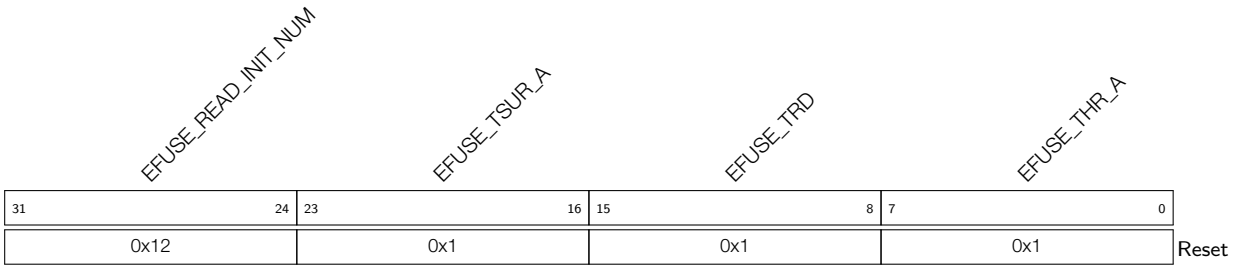
Register 16.39: EFUSE_INT_CLR_REG (0x01E4)



EFUSE_READ_DONE_INT_CLR 读取完成中断的清除位。(只写)

EFUSE_PGM_DONE_INT_CLR 烧写完成中断的清除位。(只写)

Register 16.40: EFUSE_RD_TIM_CONF_REG (0x01EC)



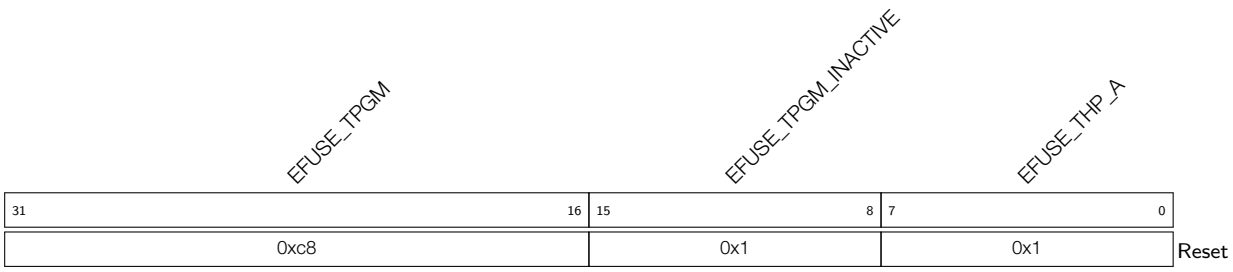
EFUSE_THR_A 配置读取操作的保持时间。(读/写)

EFUSE_TRD 配置读取操作的脉冲长度。(读/写)

EFUSE_TSUR_A 配置读取操作的建立时间。(读/写)

EFUSE_READ_INIT_NUM 配置 eFuse 的初始读取时间。(读/写)

Register 16.41: EFUSE_WR_TIM_CONF0_REG (0x01F0)

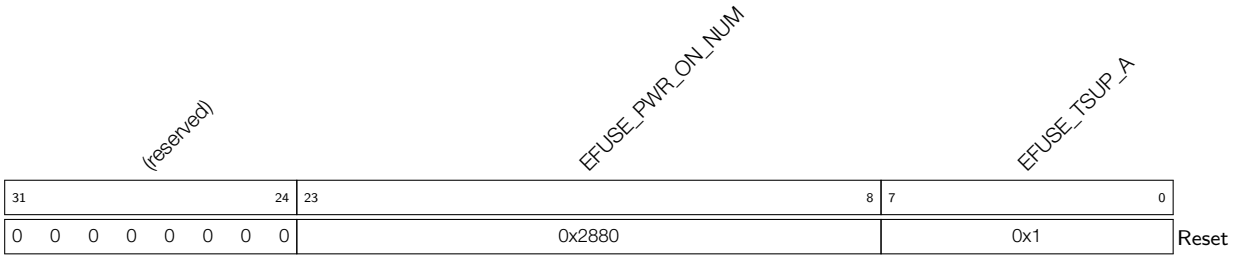


EFUSE_THP_A 配置烧写操作的保持时间。(读/写)

EFUSE_TPGM_INACTIVE 配置烧写 eFuse 为 0 时的脉冲长度。(读/写)

EFUSE_TPGM 配置烧写 eFuse 为 1 时的脉冲长度。(读/写)

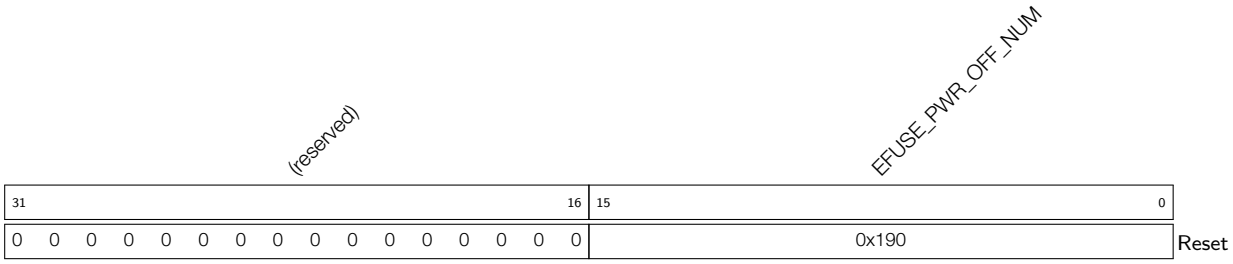
Register 16.42: EFUSE_WR_TIM_CONF1_REG (0x01F4)



EFUSE_TSUP_A 配置烧写操作的建立时间。(读/写)

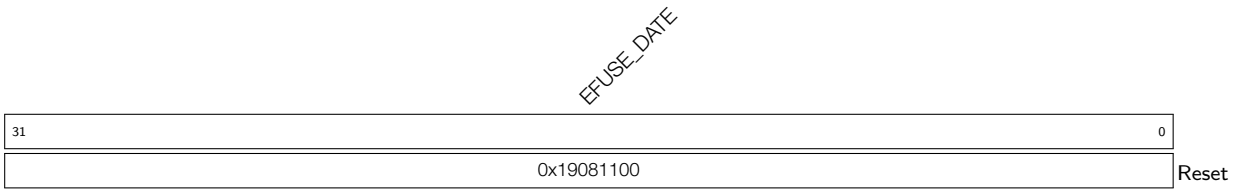
EFUSE_PWR_ON_NUM 配置烧写电压 VDDQ 的上升时间。(读/写)

Register 16.43: EFUSE_WR_TIM_CONF2_REG (0x01F8)



EFUSE_PWR_OFF_NUM 配置烧写电压 VDDQ 的下降时间。(读/写)

Register 16.44: EFUSE_DATE_REG (0x01FC)



EFUSE_DATE 版本控制寄存器。(读/写)

17. I²C 控制器

17.1 概述

I²C (Inter-Integrated Circuit) 总线用于使 ESP32-S2 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。

17.2 主要特性

I2C 具有以下几个特点。

- 支持主机模式以及从机模式
- 支持多主机多从机通信
- 支持标准模式 (100 kbit/s)
- 支持快速模式 (400 kbit/s)
- 支持 7-bit 以及 10-bit 模式寻址
- 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
- 支持双寻址模式

17.3 I²C 功能描述

17.3.1 I²C 简介

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 个脉冲用于按位传输，该字节包括 7-bit 地址和 1 个读写位。如果从机地址与该 7-bit 地址一致，那么从机可以通过在第 9 个脉冲上拉低 SDA 线来应答。接下来，根据读 / 写标志位，主机和从机可以发送 / 接收更多的数据。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。

17.3.2 I²C 架构

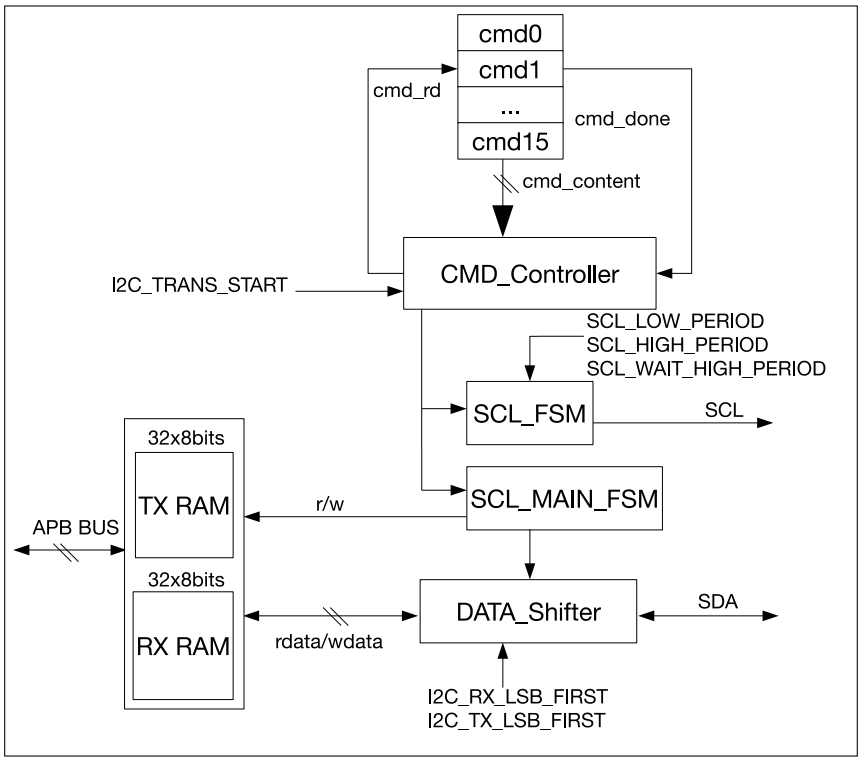


图 17-1. I²C Master 基本架构

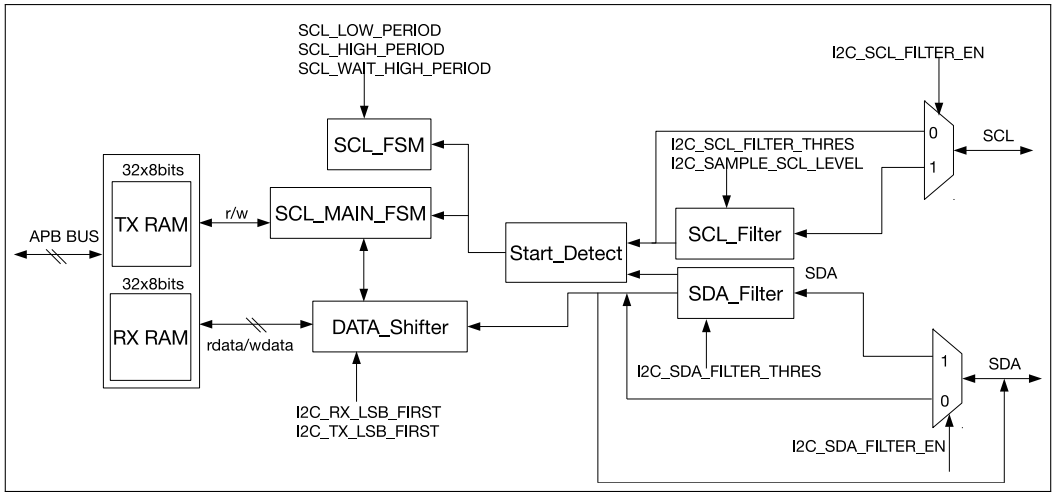


图 17-2. I²C Slave 基本架构

I²C 控制器可以工作于 Master 模式或者 Slave 模式，I2C_MS_MODE 寄存器用于模式选择。图 17-1 为 I²C Master 基本架构图，图 17-2 为 I²C Slave 基本架构图。I²C 控制器内部包括的模块主要有 TX/RX RAM、CMD_Controller、SCL_FSM、SCL_MAIN_FSM、DATA_Shifter、SCL_Filter 和 SDA_Filter 。

17.3.2.1 TX/RX RAM

TX/RX RAM 大小均为 32 x 8 bits。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。

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。

TX RAM 和 RX RAM 均可以通过 FIFO 和直接地址 (non-FIFO) 两种方式访问，具体访问方式通过 I2C_NONFIFO_EN 位配置。

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 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 的地址相比于写 TX RAM 地址需要减去 0x80 的偏移。

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 + 0x100) ~(I²C 基地址 + 0x17C) 直接访问 RX RAM。RX RAM 的每一个字节占据一个 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。

TX RAM 的写地址和 RX RAM 的读地址范围一样，因此可以把 TX RAM 和 RX RAM 看成一块 32 x 8 bits 的 RAM。本文在后续章节中都以 RAM 来代替 TX RAM 和 RX RAM。

17.3.2.2 CMD_Controller

I²C 控制器工作于主机模式时，CMD_Controller 会依次从 16 个命令寄存器中读出命令并按照命令来控制 SCL_FSM 及 SDA_FSM。

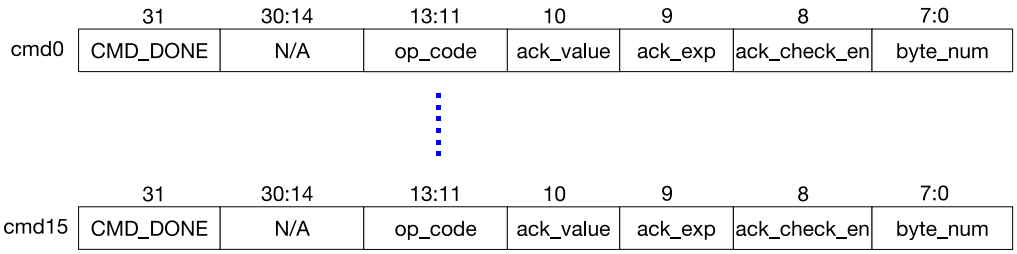


图 17-3. I²C 命令寄存器结构

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 17-3 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 5 种命令。
    - RSTART: op_code 等于 0 时为 RSTART 命令，该命令指示 I²C 控制器发送 I²C 协议中的 START 位以及 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 2 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 3 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 Master 会产生 I2C_NACK_INT 中断，停止发送数据并产生 STOP。1: 检测从机发送的 ACK 位电平; 0: 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 16 个命令寄存器中必须有 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、数据长度和发送数据等。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。

17.3.2.3 SCL_FSM

SCL_FSM 单元模块控制 SCL 时钟线。I2C_SCL_LOW_PERIOD_REG、I2C_SCL_HIGH_PERIOD_REG 和 I2C_SCL_WAIT_HIGH_PERIOD 用于配置 SCL 的频率和占空比。当 SCL_FSM 长时间处于非空闲状态，且时间超过 I2C_SCL_ST_TO 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。

17.3.2.4 SCL_MAIN_FSM

SCL_MAIN_FSM 单元模块控制 SDA 数据线以及数据的存取。当 SCL_MAIN_FSM 长时间处于非空闲状态，且时间超过 I2C_SCL_MAIN_ST_TO 个模块时钟后，会触发 I2C_SCL_MAIN_ST_TO_INT 中断，状态机会回到空闲状态。

17.3.2.5 DATA_Shifter

DATA_Shifter 模块用于串并转换，将字节数据转化成比特流或者将比特流转化成字节数据。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。

17.3.2.6 SCL_Filter 和 SDA_Filter

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，SCL_Filter 滤波器的功能为连续采样输入信号 SCL，如果输入信号在连续 I2C_SCL_FILTER_THRES 个 APB 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 APB 时钟周期的线路毛刺。

17.3.3 I²C 总线时序

I²C 控制器计时的时钟源可以用 APB_CLK，也可以用 REF_TICK。I2C_REF_ALWAYS_ON 置 1 选择 APB_CLK，清零选择 REF_TICK。

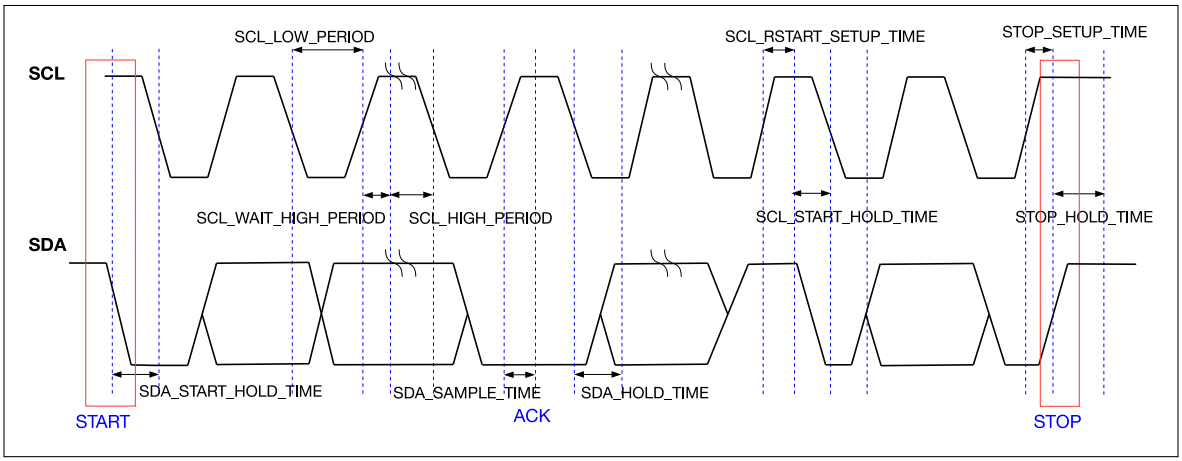


图 17-4. I²C 时序图

图 17-4 为 I2C 主机的时序图，图中的参数是以模块时钟 (I2C_CLK) 为单位的，即 I2C_REF_ALWAYS_ON 为 1 时，以 T_{APB_CLK} 为单位；I2C_REF_ALWAYS_ON 为 0 时，以 T_{REF_TICK} 为单位。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 17-4 中所示的寄存器进行配置。如图17-4 所示，各参数的含义如下:

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 延展传输则可能会导致 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:

$$f_{scl} = \frac{f_{I2C\_CLK}}{I2C\_SCL\_LOW\_PERIOD+1+I2C\_SCL\_HIGH\_PERIOD+I2C\_SCL\_WAIT\_HIGH\_PERIOD}$$

5. 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。
6. 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。

SCL 及 SDA 线采用 open-drain 的驱动方式。I²C 控制器有两种配置方式实现 open-drain 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 open-drain 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 的输出频率受限于 SCL 和 SDA 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。

17.4 典型应用

为了便于描述，下文所有图示中的 I²C Master 和 Slave 都假定为 I²C 外设控制器。

17.4.1 I²C 主机写入从机，7-bit 寻址，单次命令序列

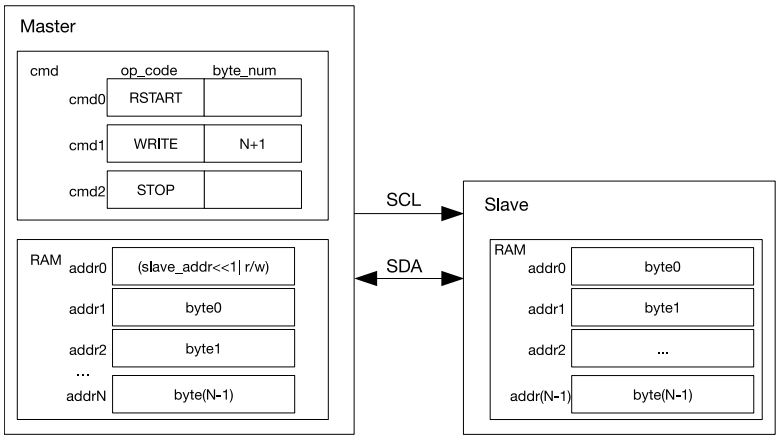


图 17-5. I²C Master 写 7-bit 寻址的 Slave

图 17-5 为 I²C Master 采用 7-bit 寻址写 N 个字节数据到 I²C Slave 的命令寄存器及 RAM 的值。如图 17-5 所示，主机 RAM 中第一个字节数据为 7-bit Slave 地址 + 1-bit 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. 控制器会等待 SCL 线为高电平，以避免 SCL 线被其他 Master 或者 Slave 占用。
2. 控制器执行 RSTART 命令发送 START 位。

3. 控制器执行 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 Master 完成 STOP 位的传输后，会产生 I2C_TRANS_COMPLETE_INT 中断。

当需要传输的数据量太大，超过 32 字节时，可以对 RAM 使用乒乓操作。在控制器发送数据的同时，软件更新已发送的数据。当 I²C Master 的 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。

软件在检测到该中断后，向使用完的 RAM 区域更新新的数据。当 RAM 采用 non-FIFO 访问时，软件可以配置 I2C_TX_UPDATE 锁存已发送数据在 RAM 中的首末地址，通过读取 I2C_FIFO_ST_REG 寄存器中 I2C_TXFIFO_START_ADDR 段及 I2C_TXFIFO_END_ADDR 得到已发送数据在 RAM 中的首末地址，从而更新 RAM 中的旧数据。当 RAM 采用 FIFO 访问时，直接通过 I2C_DATA_REG 寄存器写入新的数据就可以。

有如下两种情况会打断控制器执行命令序列：

1. 当 I²C Master WRITE 命令中的 ack_check_en 配置为 1 时，I²C Master 会在发送完每个字节之后进行 ACK 检测。如果接收的 ACK 值与 WRITE 命令中的 ack_exp 电平不一致时，I²C Master 会产生 I2C_NACK_INT 中断，停止发送数据并且产生 STOP。

2. I²C Master 在 SCL 为高电平期间，检测到 SDA 输入值与 SDA 输出值不等时，则 I²C Master 会产生 I2C_ARBITRATION_LOST_INT 中断，并停止命令序列的执行返回 IDLE 状态，释放对 SCL 及 SDA 线的控制。

I²C Slave 在检测到 I²C Master 发送的 START 位之后，开始接收地址并进行地址匹配。当 I²C Slave 接收的地址与其 I2C_SLAVE_ADDR[6:0] 的值不匹配时，I²C Slave 停止接收数据。当地址匹配后，I²C Slave 将接下来接收的数据按照顺序存储到 RAM 中。

当需要接收的数据量太大，超过 32 字节时，可以对 RAM 使用乒乓操作。在控制器接收数据的同时，软件回收已接收的数据。当 I²C Slave 的 RAM 中接收的待回收的数据字节数大于等于 I2C_RXFIFO_WM_THRHD 时，会产生 I2C_RXFIFO_WM_INT 中断。

软件在检测到该中断后，需要回收相应的 RAM 区域数据。当 RAM 采用 non-FIFO 访问时，软件可以配置 I2C
_RX_UPDATE 锁存已待回收数据在 RAM 中的首末地址，通过读取 RD_FIFO_ST_REG 寄存器中 RXFIFO_START_ADDR 段及 RXFIFO_END_ADDR 得到已发送数据在 RAM 中的首末地址，从而回收 RAM 中的接收数据。当 RAM 采用 FIFO 访问时，直接通过 I2C_DATA_REG 寄存器回收数据就可以。

17.4.2 I²C 主机写入从机，10-bit 寻址，单次命令序列

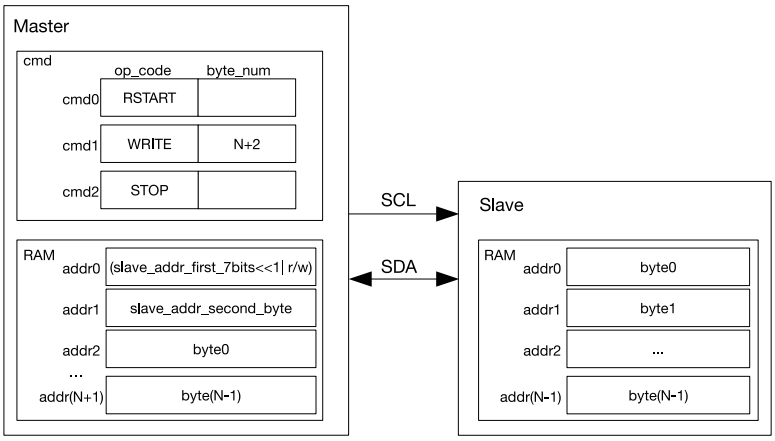


图 17-6. I²C Master 写 10-bit 寻址的 Slave

假设从机地址为 SLV_ADDR 。ESP32-S2 I²C 控制器可以使用 7-bit 寻址（SLV_ADDR[6:0]），也可以使用 10-bit 寻址（SLV_ADDR[9:0]）。

图 17-6 为 I²C Master 写 N 个字节到 10-bit 地址 I²C Slave 的配置图。主机模式下，相比于 7-bit 寻址，10-bit 寻址需要发送两字节地址段。将从机地址的第一个 7 bits slave_addr_first_7bits 和读写标志位存入 RAM 的 addr0 地址，slave_addr_first_7bits 的值应该配置为 (0x78 | SLV_ADDR[9:8])。接着将 slave_addr_second_byte 存入 RAM 的 addr1 地址，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-bit 寻址模式。I2C_SLAVE_ADDR 用于配置 I²C Slave 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-bit Slave 地址比 7-bit 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。

17.4.3 I²C 主机写入从机，7-bit 双地址寻址，单次命令序列

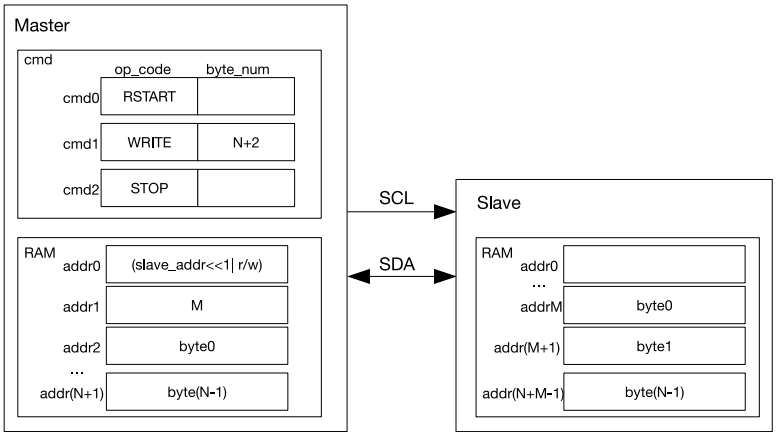


图 17-7. I²C Master 写 7-bit 寻址 Slave 的 M 地址 RAM

控制器处于 slave 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。如图 17-7 所示，I²C Slave 将接收到的数据 byte0 ~ byte(N-1) 从 Slave RAM 中的 addrM 开始依次存储。当超出地址 31 后会从地址 0 开始继续存储。

17.4.4 I²C 主机写入从机，7-bit 寻址，多次命令序列

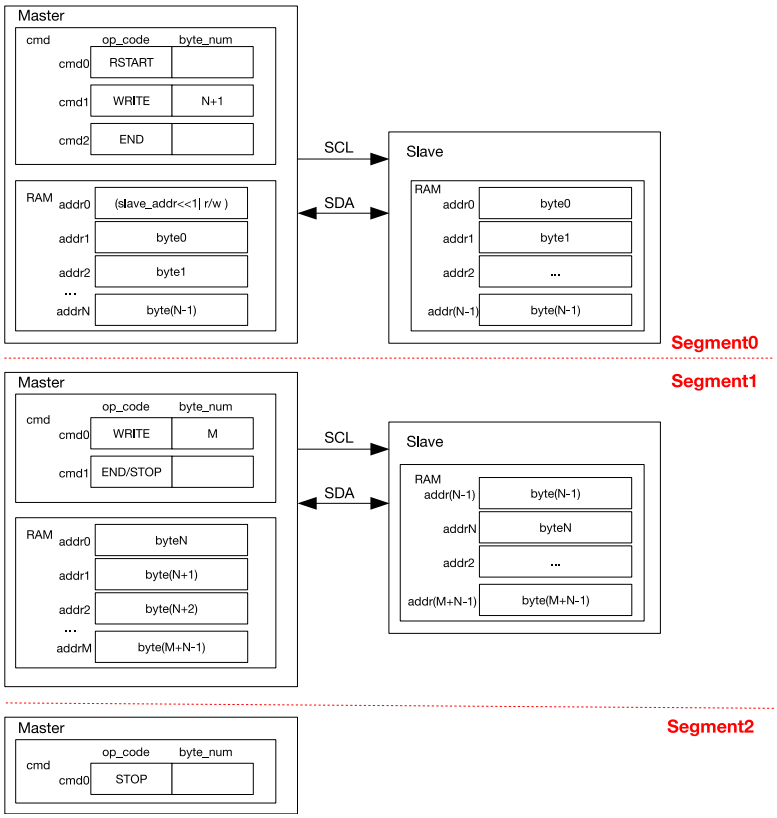


图 17-8. I²C Master 分段写 7-bit 寻址的 Slave

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 17-8 所示为 I²C Master 分成三段或者两段写 Slave。首先配置 I²C Master 的命令序列如第一段所示，并且在 Master 的 RAM 中准备好数据，置位 I2C_TRANS_START，I²C Master 即开始数据传输。在执行到 END 命令后，I²C Master 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 Slave。置位 I2C_TRANS_START 后，I²C Master 继续发送数据，并在最后发送 STOP 位。当为三段写 Slave 时,I²C Master 在第二段发送完数据，并检测到 I²C Master 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 Master 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 Master 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。

在 I²C Master 处于空闲状态时，置位 I2C_SCL_RST_SLV_EN，硬件会发送 I2C_ SCL_RST_SLV_NUM 个 SCL 脉冲，之后硬件会自清 I2C_SCL_RST_SLV_EN 位。

需要注意的是，总线上其他 Master 或者 Slave 的操作可能与 ESP32-S2 I²C 外设有所不同，具体请参考各个 I²C 设备的技术规格书。

17.4.5 I²C 主机读取从机，7-bit 寻址，单次命令序列

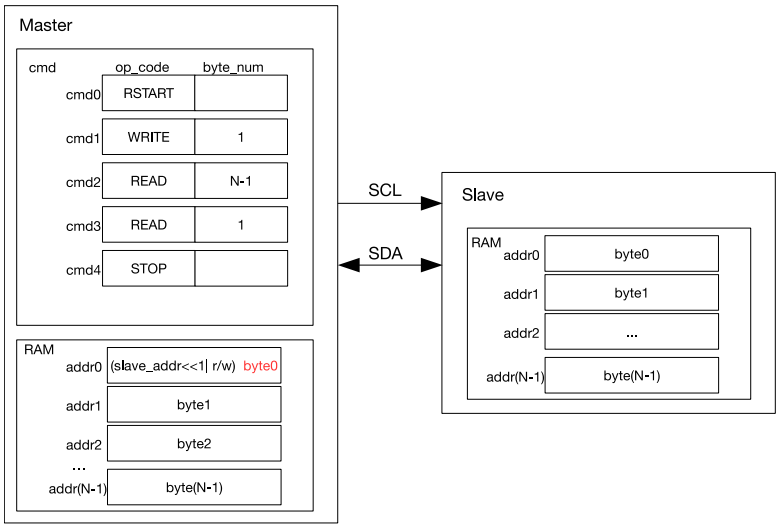


图 17-9. I²C Master 读 7-bit 寻址的 Slave

图 17-9 I²C Master 从 7-bit 寻址 I²C Slave 读取 N 个字节数据的的命令寄存器及 RAM 的值。cmd1 为 WRITE 命令，I²C Master 会将 I²C Slave 的地址发送出去。该命令发送的字节是 7-bit I²C Slave 地址以及读写标志位。读写标志位为 1 表示读操作。I²C Slave 在地址匹配成功之后即开始发送数据给 I²C Master。I²C Master 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 17-9 中 READ 分成两次，I²C Master 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 Master 从 RAM 的首地址开始存储，图中红色 byte0 会覆盖第一个字节的内容（Slave 地址 +1-bit 读写位）。

17.4.6 I²C 主机读取从机，10-bit 寻址，单次命令序列

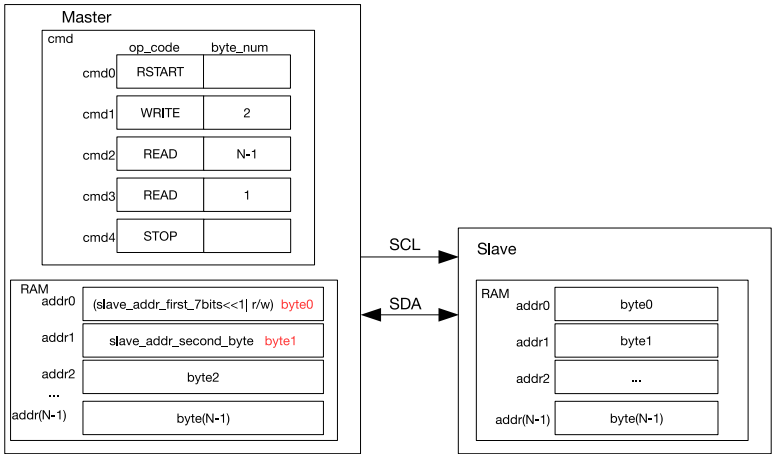


图 17-10. I²C Master 读 10-bit 寻址的 Slave

图 17-10 为 I²C Master 从 10-bit 寻址的 I²C Slave 中读取数据的命令寄存器及 RAM 的值。相比于 7-bit 寻址，I²C Master 的第一写命令的字节数为 2 字节，相应 RAM 中存储 2 个字节的 I²C Slave 10-bit 地址。相比于 7-bit 寻址，I²C Slave 需要置位 I2C_ADDR_10BIT_EN 和 I2C_SLAVE_ADDR[14：0]。具体配置方式与 17.4.2 小节的相同。

17.4.7 I²C 主机读取从机，7-bit 双寻址，单次命令序列

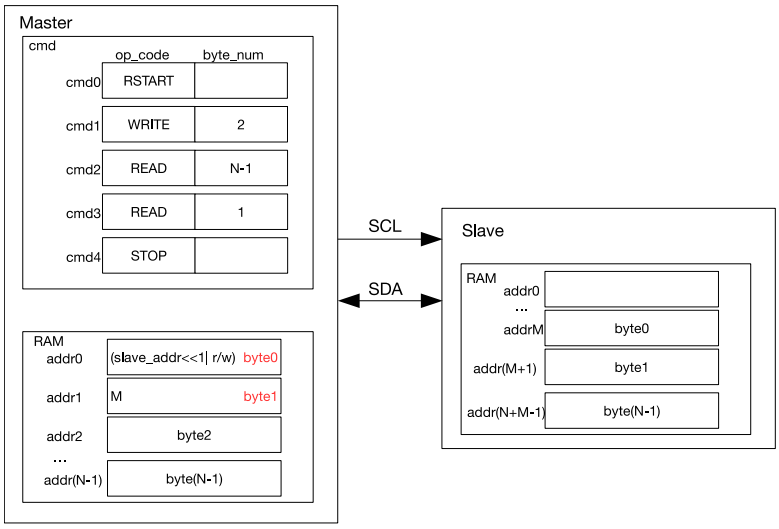


图 17-11. I²C Master 从 7-bit 寻址 Slave 的 M 地址读取 N 个数据

图 17-11 为 I²C Master 从 I²C Slave 中指定地址读取数据的命令寄存器及 RAM 的值。配置流程如下：

1. 在 I²C Slave 中置位 I2C_FIFO_ADDR_CFG_EN 并在其 RAM 中准备好待发送的数据。
2. 在 I²C Master 中准备好 I²C Slave 的地址以及其指定的寄存器地址 M。
3. 置位 I²C Master 的 I2C_TRANS_START，I²C Slave 会将从 RAM 中 M 地址开始取 N 个数据发送给 I²C Master。

17.4.8 I²C 主机读取从机，7-bit 寻址，多次命令序列

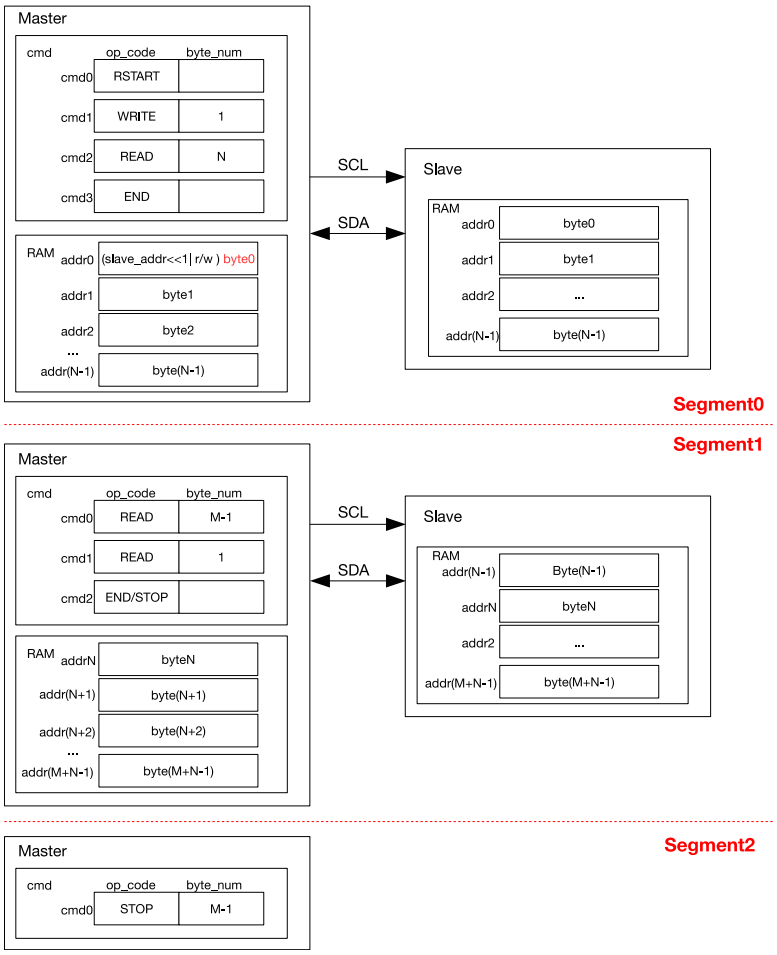


图 17-12. I²C Master 分段读 7-bit 寻址的 Slave

图 17-12 为 I²C Master 通过 END 命令分三段或者分两段，从 I²C Slave 读取 N+M 个数据的配置图。配置的流程如下：

1. 首先配置命令寄存器和 RAM 的内容，如第一段所示。
2. 接着在 Slave 的 RAM 中准备好数据，置位 I2C_TRANS_START，I²C 即开始工作。当执行到 END 命令时，I²C Master 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 Slave，置位 I2C_TRANS_START，I²C Master 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 Master 完成第二次数据传输，并检测到 I²C Master 的 I2C_END_DETECT_INT 中断后，配置 cmd 如第三段所示。置位 I2C_TRANS_START，I²C Master 发送 STOP 位停止传输。

17.5 SCL 延展传输

从机模式下，可以通过 SCL 线拉低，以给软件足够的时间进行处理。置位 I2C_SLAVE_SCL_STRETCH_EN 位使能延展传输，置位 I2C_STRETCH_PROTECT_NUM 位配置延展传输时长。出现以下三种情况会拉低 SCL 线：

1. 地址命中：从机模式下，I²C 控制器的地址与 SDA 线发送的地址相匹配。

2. 写满：从机模式下，I²C 控制器的 RX RAM 为满。
3. 读空：从机模式下，I²C 控制器的 TX RAM 为空。

SCL 线拉低后，可读取 I2C_STRETCH_CAUSE 位获取延展传输的原因。置位 I2C_SLAVE_SCL_STRETCH_CLR 位关闭 SCL 延展传输。

17.6 中断

- I2C_SLAVE_STRETCH_INT: 当 I²C 从机将 SCL 线拉低时产生此中断。
- I2C_DET_START_INT：当检测到 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[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[23:0] 个模块时钟周期时，触发此中断。
- I2C_RXFIFO_UDF_INT: 直接地址访问模式下，当 I²C 接收I2C_NONFIFO_RX_THRES 个数据，即触发该中断。
- I2C_TXFIFO_OVF_INT: 每当 I²C 发送I2C_NONFIFO_TX_THRES 个数据，即触发该中断。
- I2C_NACK_INT: 当 I²C 配置为 Master 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 Slave 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 I2C_TIME_OUT 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 Master 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。

17.7 基地址

用户可通过不同的寄存器基地址访问 I²C 控制器，如表 63 所示。更多信息，请访问章节 1 系统和存储器。

表 63: I²C 控制器基地址

	访问总线	基地址
I²C0	PeriBUS1	0x3F413000
	PeriBUS2	0x60013000
I²C1	PeriBUS1	0x3F427000
	PeriBUS2	0x60027000

17.8 寄存器列表

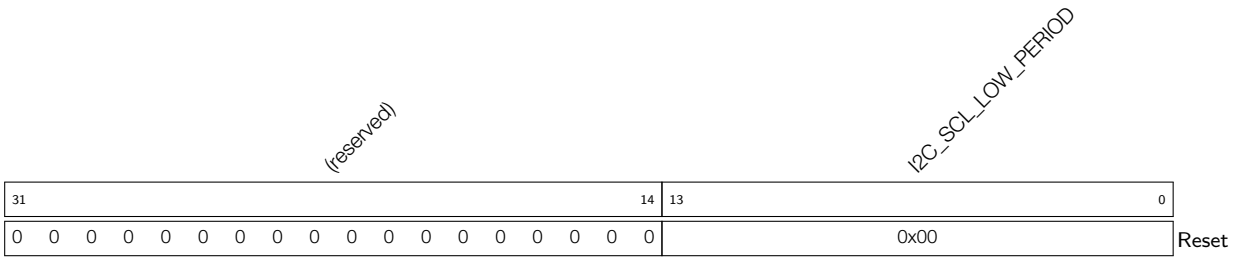
请注意，下表的地址指相对于 I²C 控制器基地址的偏移量（相对地址）。更多有关 I²C 控制器基地址的信息，请前往 17.7 章节。

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	读/写
I2C_SDA_HOLD_REG	配置 SCL 下降沿后的保持时间	0x0030	读/写
I2C_SDA_SAMPLE_REG	配置 SCL 上升沿后的采样时间	0x0034	读/写
I2C_SCL_HIGH_PERIOD_REG	配置 SCL 时钟的高电平宽度	0x0038	读/写
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	读/写
I2C_SCL_RSTART_SETUP_REG	配置 SCL 上升沿和 SDA 下降沿之间的延迟	0x0044	读/写
I2C_SCL_STOP_HOLD_REG	配置 STOP 命令生成时 SCL 边沿的延迟	0x0048	读/写
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	读/写
I2C_SCL_ST_TIME_OUT_REG	SCL 状态超时寄存器	0x0098	读/写
I2C_SCL_MAIN_ST_TIME_OUT_REG	SCL 主要状态超时寄存器	0x009C	读/写
配置寄存器			
I2C_CTR_REG	传输设置	0x0004	读/写
I2C_TO_REG	设置接收数据超时控制	0x000C	读/写
I2C_SLAVE_ADDR_REG	本地从机地址设置	0x0010	读/写
I2C_FIFO_CONF_REG	FIFO 配置寄存器	0x0018	读/写
I2C_SCL_SP_CONF_REG	电源配置寄存器	0x00A0	读/写
I2C_SCL_STRETCH_CONF_REG	配置 I2C 从机 SCL 延展传输	0x00A4	不定
状态寄存器			
I2C_SR_REG	描述 I2C 的工作状态	0x0008	只读
I2C_FIFO_ST_REG	FIFO 状态寄存器	0x0014	不定
I2C_DATA_REG	RX FIFO 读取数据	0x001C	只读
中断寄存器			
I2C_INT_RAW_REG	原始中断状态	0x0020	只读
I2C_INT_CLR_REG	中断清除位	0x0024	只写
I2C_INT_ENA_REG	中断使能位	0x0028	读/写
I2C_INT_STATUS_REG	捕捉 I2C 通信事件的状态	0x002C	只读

名称	描述	地址	访问
滤波寄存器			
I2C_SCL_FILTER_CFG_REG	SCL 滤波配置寄存器	0x0050	读/写
I2C_SDA_FILTER_CFG_REG	SDA 滤波配置寄存器	0x0054	读/写
命令寄存器			
I2C_COMD0_REG	I2C 命令寄存器 0	0x0058	读/写
I2C_COMD1_REG	I2C 命令寄存器 1	0x005C	读/写
I2C_COMD2_REG	I2C 命令寄存器 2	0x0060	读/写
I2C_COMD3_REG	I2C 命令寄存器 3	0x0064	读/写
I2C_COMD4_REG	I2C 命令寄存器 4	0x0068	读/写
I2C_COMD5_REG	I2C 命令寄存器 5	0x006C	读/写
I2C_COMD6_REG	I2C 命令寄存器 6	0x0070	读/写
I2C_COMD7_REG	I2C 命令寄存器 7	0x0074	读/写
I2C_COMD8_REG	I2C 命令寄存器 8	0x0078	读/写
I2C_COMD9_REG	I2C 命令寄存器 9	0x007C	读/写
I2C_COMD10_REG	I2C 命令寄存器 10	0x0080	读/写
I2C_COMD11_REG	I2C 命令寄存器 11	0x0084	读/写
I2C_COMD12_REG	I2C 命令寄存器 12	0x0088	读/写
I2C_COMD13_REG	I2C 命令寄存器 13	0x008C	读/写
I2C_COMD14_REG	I2C 命令寄存器 14	0x0090	读/写
I2C_COMD15_REG	I2C 命令寄存器 15	0x0094	读/写
版本寄存器			
I2C_DATE_REG	版本控制寄存器	0x00F8	读/写

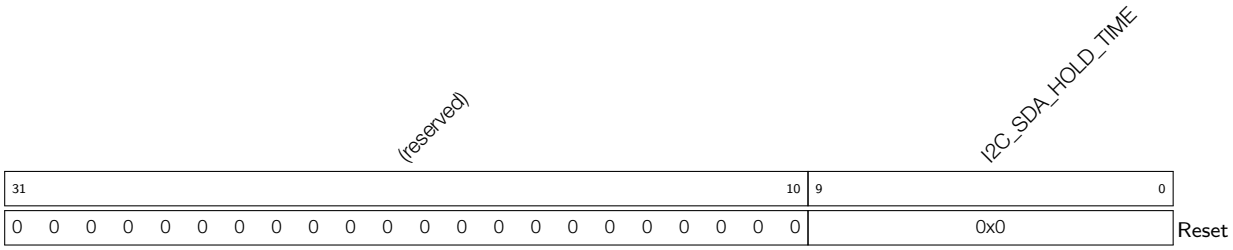
17.9 寄存器

Register 17.1: I2C_SCL_LOW_PERIOD_REG (0x0000)



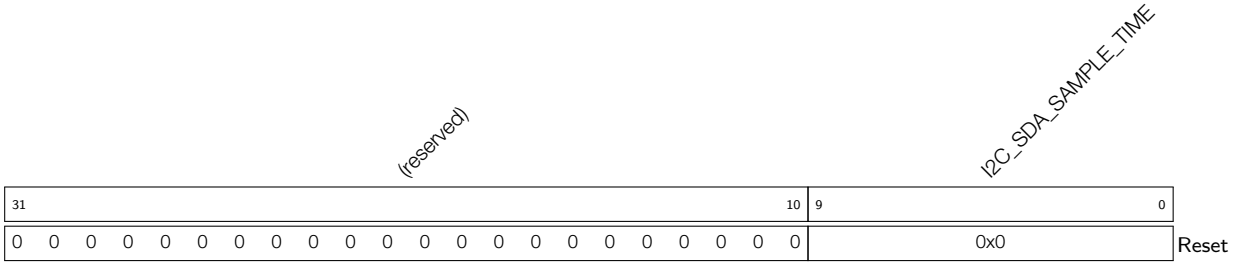
I2C_SCL_LOW_PERIOD 用于配置 SCL 低电平的保持时间，以 I2C 模块时钟周期数为单位。(读/写)

Register 17.2: I2C_SDA_HOLD_REG (0x0030)



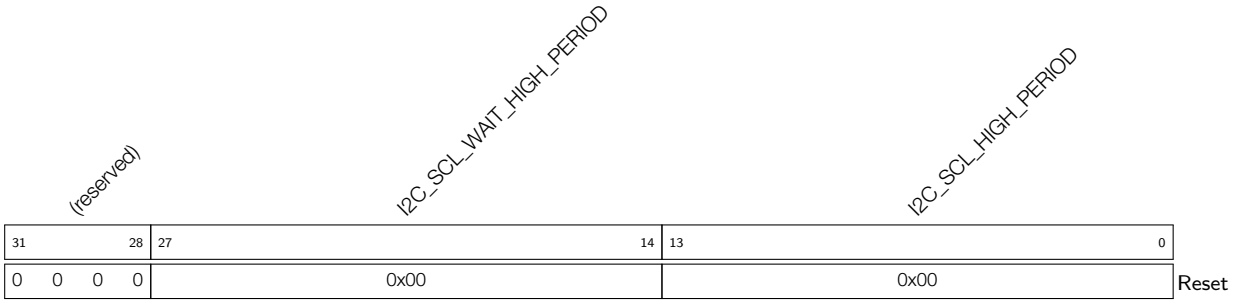
I2C_SDA_HOLD_TIME 用于配置 SCL 下降沿后的数据保持时间，以 I2C 模块时钟周期数为单位。(读/写)

Register 17.3: I2C_SDA_SAMPLE_REG (0x0034)



I2C_SDA_SAMPLE_TIME 用于配置采样 SDA 的时间，以 I2C 模块时钟周期数为单位。(读/写)

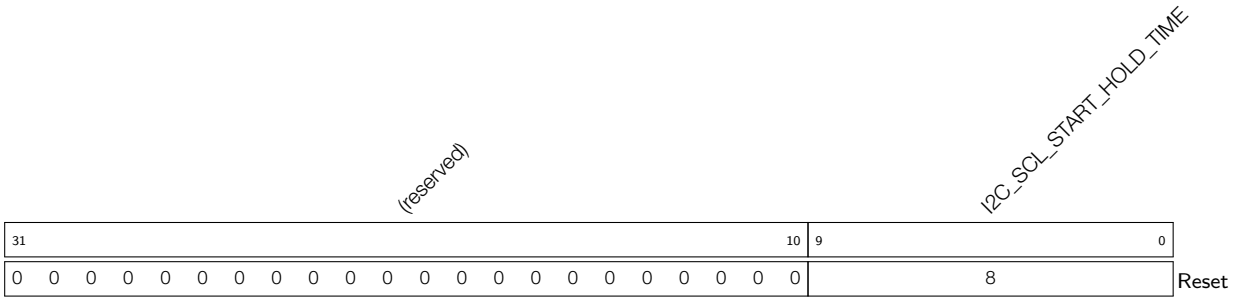
Register 17.4: I2C_SCL_HIGH_PERIOD_REG (0x0038)



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 模块时钟周期数为单位。(读/写)

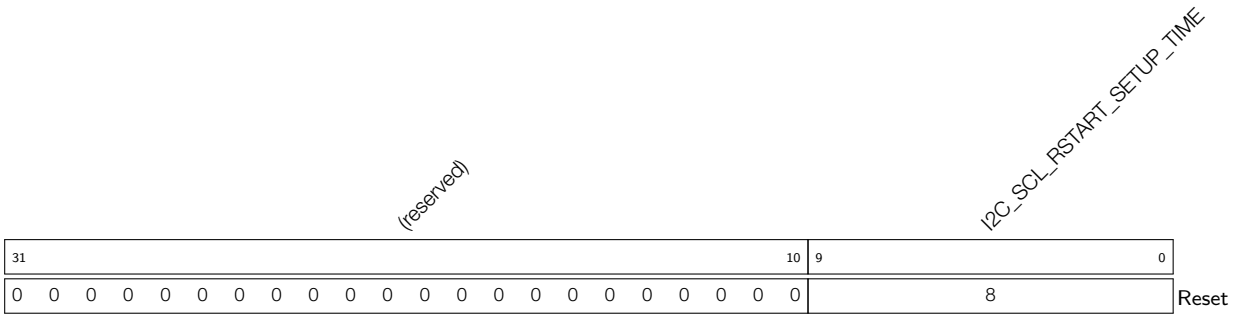
I2C_SCL_WAIT_HIGH_PERIOD 用于配置 SCL_FSM 等待 SCL 在主机模式下翻转至高电平的时间，以 I2C 模块时钟周期数为单位。(读/写)

Register 17.5: I2C_SCL_START_HOLD_REG (0x0040)



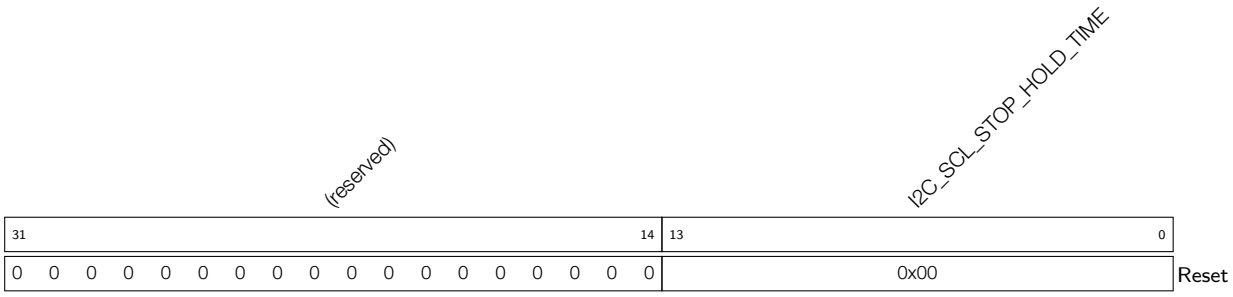
I2C_SCL_START_HOLD_TIME 配置 START 命令产生时 SDA 下降沿和 SCL 下降沿的间隔时间，以 I2C 模块时钟周期数为单位。(读/写)

Register 17.6: I2C_SCL_RSTART_SETUP_REG (0x0044)



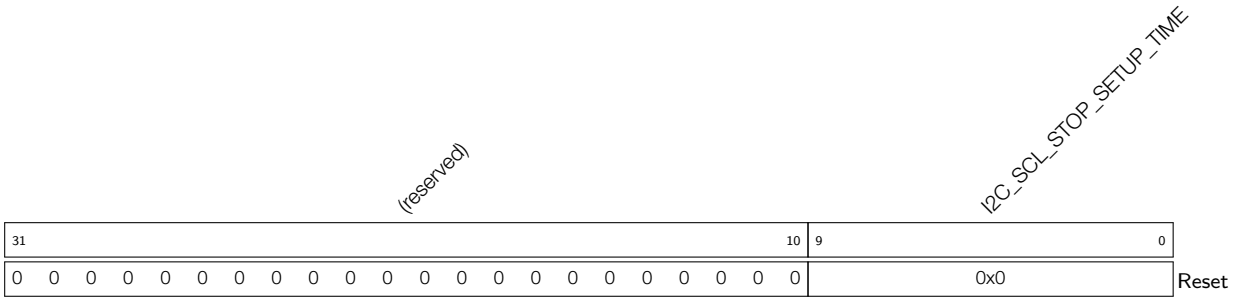
I2C_SCL_RSTART_SETUP_TIME 配置 RESTART 命令产生时 SCL 上升沿和 SDA 下降沿的间隔时间，以 I2C 模块的时钟周期数为单位。(读/写)

Register 17.7: I2C_SCL_STOP_HOLD_REG (0x0048)



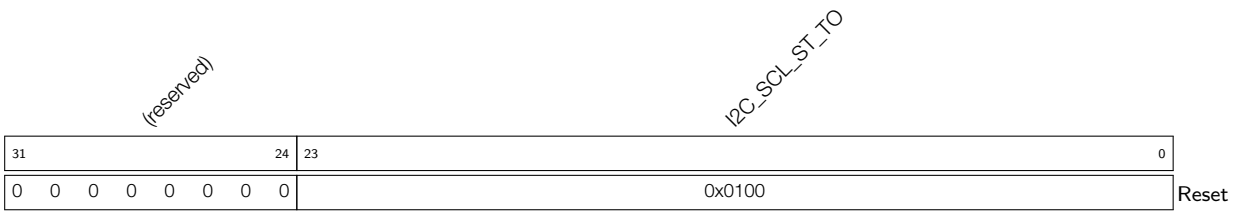
I2C_SCL_STOP_HOLD_TIME 配置 STOP 命令后的延迟，以 I2C 模块时钟周期数为单位。(读/写)

Register 17.8: I2C_SCL_STOP_SETUP_REG (0x004C)



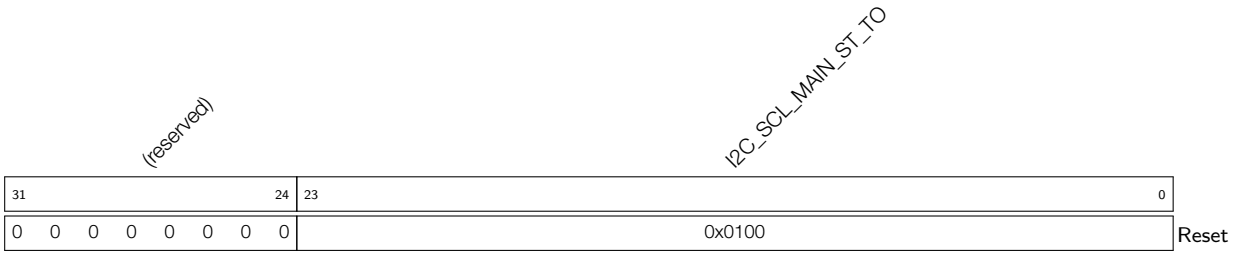
I2C_SCL_STOP_SETUP_TIME 配置 SCL 上升沿和 SDA 上升沿的间隔时间，以 I2C 模块时钟周期数为单位。(读/写)

Register 17.9: I2C_SCL_ST_TIME_OUT_REG (0x0098)



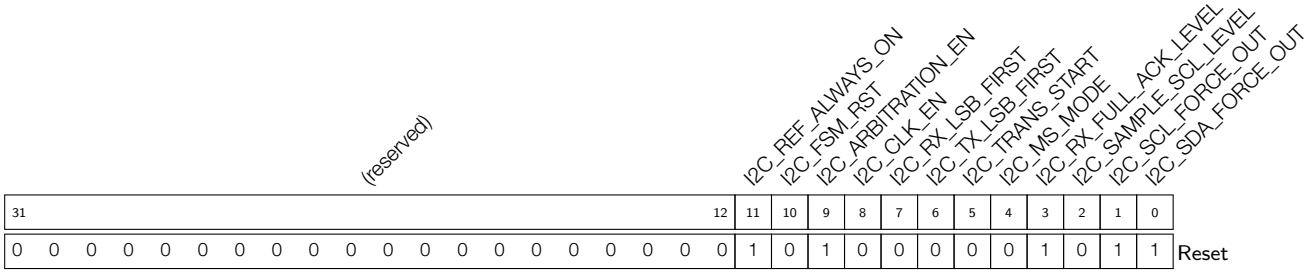
I2C_SCL_ST_TO SCL_FSM 状态不变的阈值。(读/写)

Register 17.10: I2C_SCL_MAIN_ST_TIME_OUT_REG (0x009C)



I2C_SCL_MAIN_ST_TO SCL_MAIN_FSM 状态不变的阈值。(读/写)

Register 17.11: I2C_CTR_REG (0x0004)



I2C_SDA_FORCE_OUT 0：直接输出；1：漏极开路输出。(读/写)

I2C_SCL_FORCE_OUT 0：直接输出；1：漏极开路输出。(读/写)

I2C_SAMPLE_SCL_LEVEL 用于选择采样模式。1：SCL 为低电平时采样 SDA 数据。0：SCL 为高电平时采样 SDA 数据。(读/写)

I2C_RX_FULL_ACK_LEVEL 用于配置主机在 rx_fifo_cnt 达到阈值时需发送的 ACK 电平值。(读/写)

I2C_MS_MODE 置位此位，将模块配置为 I2C 主机。清零此位，将模块配置为 I2C 从机。(读/写)

I2C_TRANS_START 置位此位，开始发送 TX FIFO 中的数据。(读/写)

I2C_TX_LSB_FIRST 用于控制待发送数据的发送模式。1：从最低有效位开始发送数据；0：从最高有效位开始发送数据。(读/写)

I2C_RX_LSB_FIRST 用于控制接收数据的存储模式。1：从最低有效位开始接收数据；0：从最高有效位开始接收数据。(读/写)

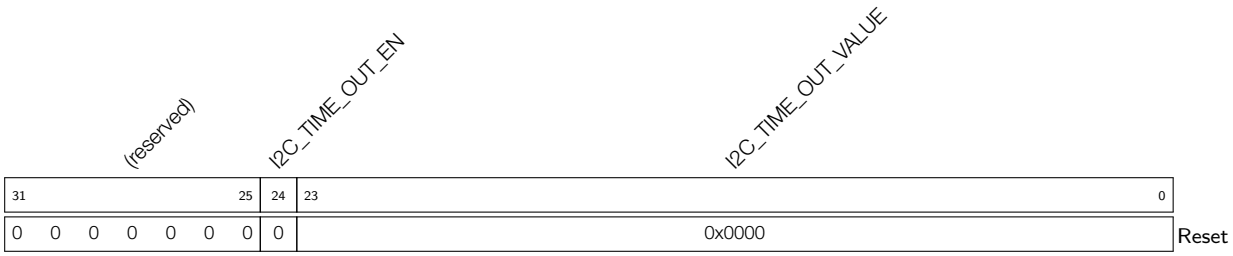
I2C_CLK_EN 保留 (读/写)

I2C_ARBITRATION_EN I2C 总线仲裁的使能位。(读/写)

I2C_FSM_RST 用于复位 SCL_FSM。(读/写)

I2C_REF_ALWAYS_ON 用于控制 REF_TICK。(读/写)

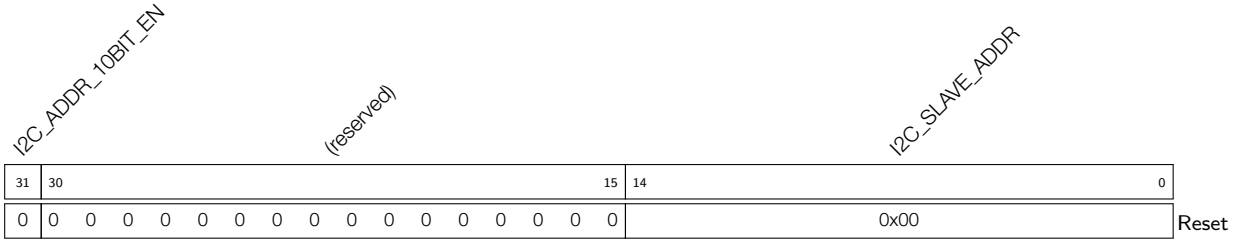
Register 17.12: I2C_TO_REG (0x000C)



I2C_TIME_OUT_VALUE 用于配置接收一位数据的超时时间，以 APB 时钟周期为单位。(读/写)

I2C_TIME_OUT_EN 超时控制使能位。(读/写)

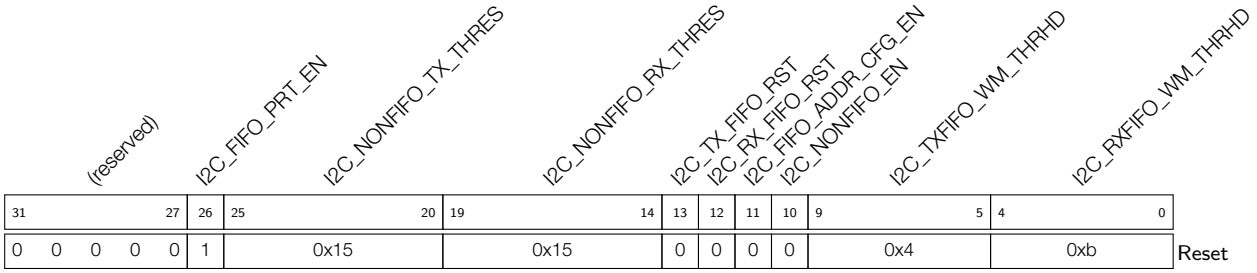
Register 17.13: I2C_SLAVE_ADDR_REG (0x0010)



I2C_SLAVE_ADDR 配置为 Slave 时，该字段用于配置从机地址。(读/写)

I2C_ADDR_10BIT_EN 用于在主机模式下使能从机的 10 位寻址模式。(读/写)

Register 17.14: I2C_FIFO_CONF_REG (0x0018)



I2C_RXFIFO_WM_THRHD non-FIFO 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。(读/写)

I2C_TXFIFO_WM_THRHD non-FIFO 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。(读/写)

I2C_NONFIFO_EN 置位此位，使能 APB non-FIFO 访问。(读/写)

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。(读/写)

I2C_RX_FIFO_RST 置位此位，复位 RX FIFO 。(读/写)

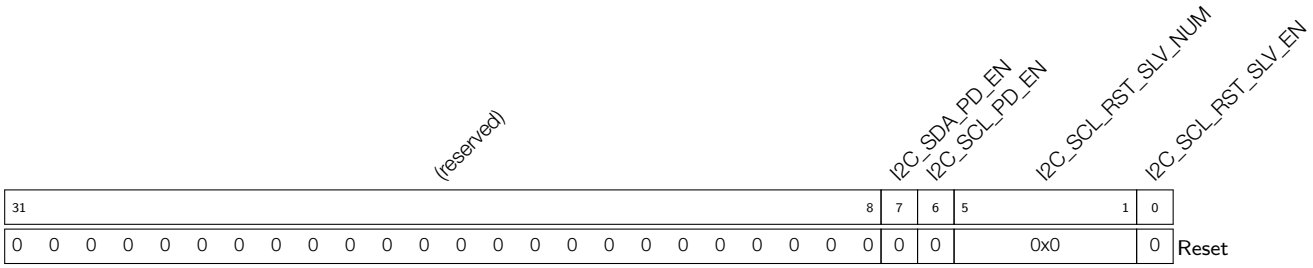
I2C_TX_FIFO_RST 置位此位，复位 TX FIFO 。(读/写)

I2C_NONFIFO_RX_THRES I2C 接收的数据超过 I2C_NONFIFO_TX_THRES 字节时，生成 I2C_RXFIFO_UDF_INT 中断，更新接收数据的当前偏移地址。(读/写)

I2C_NONFIFO_TX_THRES I2C 发送的数据超过 I2C_NONFIFO_TX_THRES 个字节时，生成 I2C_TXFIFO_OVF_INT 中断，更新发送数据的当前偏移地址。(读/写)

I2C_FIFO_PRT_EN non-FIFO 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。(读/写)

Register 17.15: I2C_SCL_SP_CONF_REG (0x00A0)



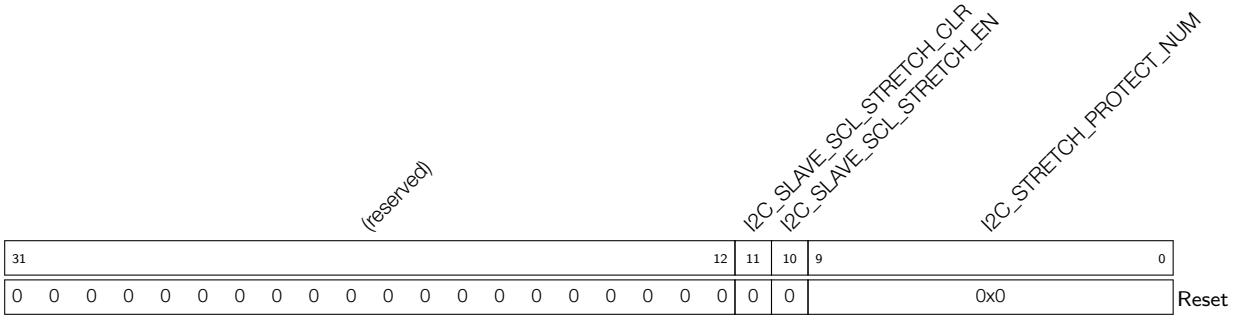
I2C_SCL_RST_SLV_EN I2C 主机处于空闲状态时，置位此位发送 SCL 脉冲。脉冲数量为 I2C_SCL_RST_SLV_NUM[4:0]。(读/写)

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。(读/写)

I2C_SCL_PD_EN 降低 I2C SCL 输出功耗的使能位。1：不工作，降低功耗。0：正常工作。将 I2C_SCL_FORCE_OUT 和 I2C_SCL_PD_EN 置 1 延长 SCL 的低电平时间。(读/写)

I2C_SDA_PD_EN 降低 I2C SDA 输出功耗的使能位。1：不工作，降低功耗。0：正常工作。将 I2C_SDA_FORCE_OUT 和 I2C_SDA_PD_EN 置 1 延长 SDA 的低电平时间。(读/写)

Register 17.16: I2C_SCL_STRETCH_CONF_REG (0x00A4)

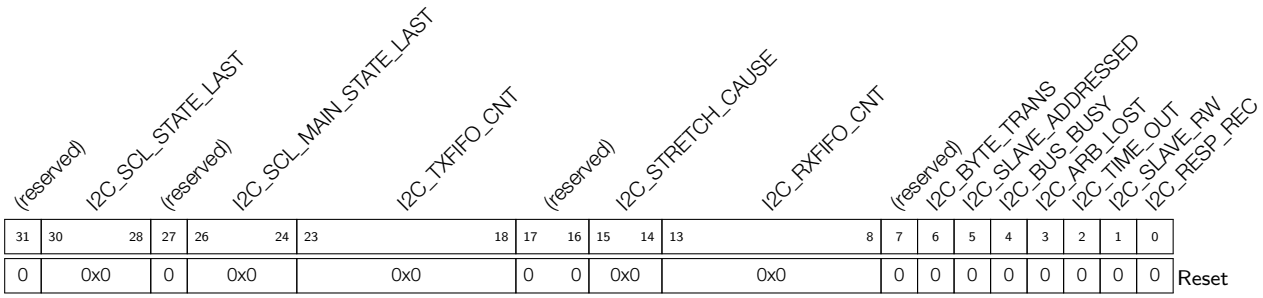


I2C_STRETCH_PROTECT_NUM 配置 I2C 从机延长 SCL 低电平时间，以时钟周期为单位。(读/写)

I2C_SLAVE_SCL_STRETCH_EN 从机 SCL 延展传输功能的使能位。1：使能。0：关闭。I2C_SLAVE_SCL_STRETCH_EN 为 1 时延展时钟，延长 SCL 输出线的低电平时间。延展传输的原因可见 I2C_STRETCH_CAUSE。(读/写)

I2C_SLAVE_SCL_STRETCH_CLR 置位此位，清除 I2C 从机的 SCL 延展传输功能。(只写)

Register 17.17: 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。(只读)

I2C_SLAVE_RW 从机模式下，1：主机读取从机数据；0：主机向从机写入数据。(只读)

I2C_TIME_OUT I2C 控制器接收一位数据的时间超过 I2C_TIME_OUT 周期时，该字段变为 1。(只读)

I2C_ARB_LOST I2C 控制器不控制 SCL 线时，该寄存器变为 1 。(只读)

I2C_BUS_BUSY 1：I2C 总线正在传输数据；0：I2C 总线处于空闲状态。(只读)

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。(只读)

I2C_BYTE_TRANS 传输一个字节后，该字段变为 1。(只读)

I2C_RXFIFO_CNT 该字段为需发送数据的字节数。(只读)

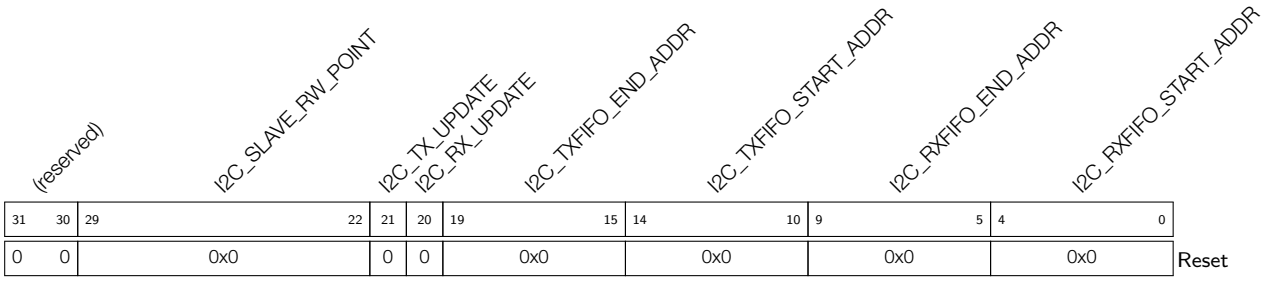
I2C_STRETCH_CAUSE 从机模式下延长 SCL 低电平时间的原因。0：I2C 开始读取数据时延长 SCL 的低电平时间。1：从机模式下 TX FIFO 为空时延长 SCL 的低电平时间。2：从机模式下 RX FIFO 为满时延长 SCL 的低电平时间。(只读)

I2C_TXFIFO_CNT 该字段存储 RAM 接收数据的字节数。(只读)

I2C_SCL_MAIN_STATE_LAST 该字段为 I2C 模块状态机的状态。0：空闲；1：地址转换；2：ACK 地址；3：接收数据；4：发送数据；5：发送 ACK；6：等待 ACK (只读)

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 (只读)

Register 17.18: I2C_FIFO_ST_REG (0x0014)



I2C_RXFIFO_START_ADDR 最后接收数据的偏移地址，如寄存器 I2C_NONFIFO_RX_THRES 所述。(只读)

I2C_RXFIFO_END_ADDR 最后接收数据的偏移地址，如寄存器 I2C_NONFIFO_RX_THRES 所述。该值在 I2C_RX_REC_FULL_INT 中断或 I2C_TRANS_COMPLETE_INT 中断产生时更新。(只读)

I2C_TXFIFO_START_ADDR 最先发送数据的偏移地址，如寄存器 I2C_NONFIFO_TX_THRES 所述。(只读)

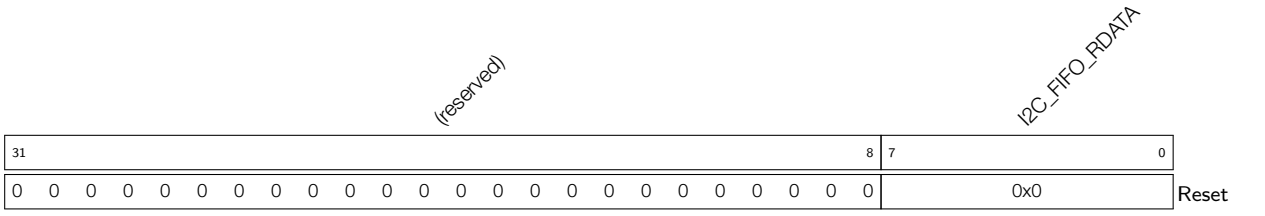
I2C_TXFIFO_END_ADDR 最后发送数据的偏移地址，如寄存器 I2C_NONFIFO_TX_THRES 所述。该值在 I2C_TX_SEND_EMPTY_INT 中断或 I2C_TRANS_COMPLETE_INT 中断产生时更新。(只读)

I2C_RX_UPDATE 在 I2C_RX_UPDATE 上写 0 或 1，更新 I2C_RXFIFO_END_ADDR 和 I2C_RXFIFO_START_ADDR 的值。(只写)

I2C_TX_UPDATE 在 I2C_TX_UPDATE 上写 0 或 1，更新 I2C_TXFIFO_END_ADDR 和 I2C_TXFIFO_START_ADDR 的值。(只写)

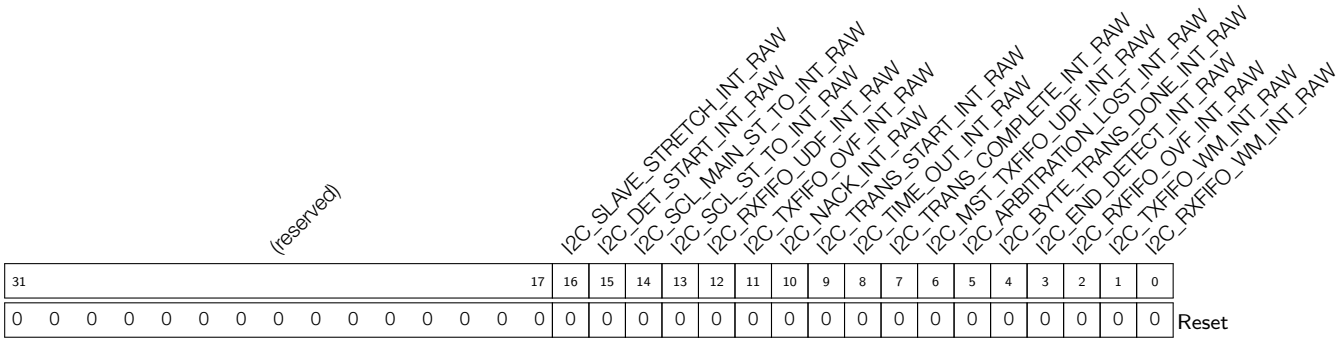
I2C_SLAVE_RW_POINT 从机模式下接收的数据。(只读)

Register 17.19: I2C_DATA_REG (0x001C)



I2C_FIFO_RDATA RX FIFO 读取数据的值。(只读)

Register 17.20: I2C_INT_RAW_REG (0x0020)



I2C_RXFIFO_WM_INT_RAW I2C_RXFIFO_WM_INT 的原始中断位。(只读)

I2C_TXFIFO_WM_INT_RAW I2C_TXFIFO_WM_INT 的原始中断位。(只读)

I2C_RXFIFO_OVF_INT_RAW I2C_RXFIFO_OVF_INT 的原始中断位。(只读)

I2C_END_DETECT_INT_RAW I2C_END_DETECT_INT 的原始中断位。(只读)

I2C_BYTE_TRANS_DONE_INT_RAW I2C_END_DETECT_INT 的原始中断位。(只读)

I2C_ARBITRATION_LOST_INT_RAW I2C_ARBITRATION_LOST_INT 的原始中断位。(只读)

I2C_MST_TXFIFO_UDF_INT_RAW I2C_TRANS_COMPLETE_INT 的原始中断位。(只读)

I2C_TRANS_COMPLETE_INT_RAW I2C_TRANS_COMPLETE_INT 的原始中断位。(只读)

I2C_TIME_OUT_INT_RAW I2C_TIME_OUT_INT 的原始中断位。(只读)

I2C_TRANS_START_INT_RAW I2C_TRANS_START_INT 的原始中断位。(只读)

I2C_NACK_INT_RAW I2C_SLAVE_STRETCH_INT 的原始中断位。(只读)

I2C_TXFIFO_OVF_INT_RAW I2C_TXFIFO_OVF_INT 的原始中断位。(只读)

I2C_RXFIFO_UDF_INT_RAW I2C_RXFIFO_UDF_INT 的原始中断位。(只读)

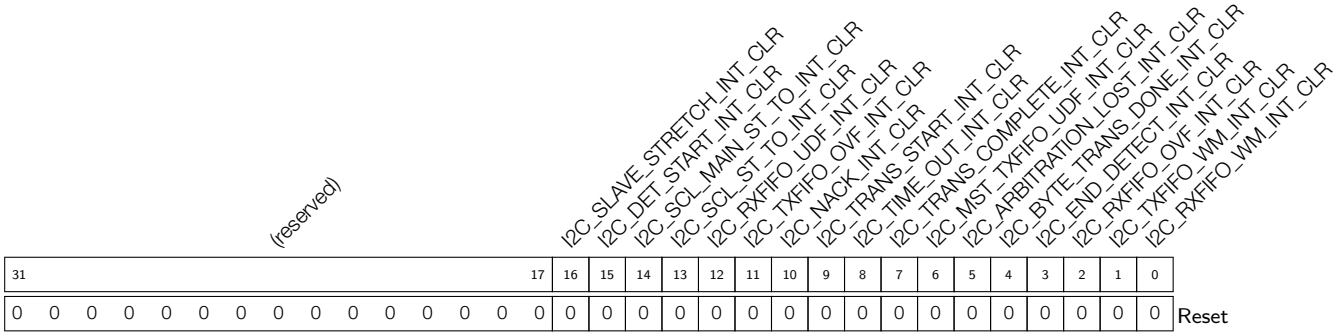
I2C_SCL_ST_TO_INT_RAW I2C_SCL_ST_TO_INT 的原始中断位。(只读)

I2C_SCL_MAIN_ST_TO_INT_RAW I2C_SCL_MAIN_ST_TO_INT 的原始中断位。(只读)

I2C_DET_START_INT_RAW I2C_DET_START_INT 的原始中断位。(只读)

I2C_SLAVE_STRETCH_INT_RAW I2C_SLAVE_STRETCH_INT 的原始中断位。(只读)

Register 17.21: I2C_INT_CLR_REG (0x0024)



I2C_RXFIFO_WM_INT_CLR 置位此位，清除 I2C_RXFIFO_WM_INT 中断。(只写)

I2C_TXFIFO_WM_INT_CLR 置位此位，清除 I2C_TXFIFO_WM_INT 中断。(只写)

I2C_RXFIFO_OVF_INT_CLR 置位此位，清除 I2C_RXFIFO_OVF_INT 中断。(只写)

I2C_END_DETECT_INT_CLR 置位此位，清除 I2C_END_DETECT_INT 中断。(只写)

I2C_BYTE_TRANS_DONE_INT_CLR 置位此位，清除 I2C_END_DETECT_INT 中断。(只写)

I2C_ARBITRATION_LOST_INT_CLR 置位此位，清除 I2C_ARBITRATION_LOST_INT 中断。(只写)

I2C_MST_TXFIFO_UDF_INT_CLR 置位此位，清除 I2C_TRANS_COMPLETE_INT 中断。(只写)

I2C_TRANS_COMPLETE_INT_CLR 置位此位，清除 I2C_TRANS_COMPLETE_INT 中断。(只写)

I2C_TIME_OUT_INT_CLR 置位此位，清除 I2C_TIME_OUT_INT 中断。(只写)

I2C_TRANS_START_INT_CLR 置位此位，清除 I2C_TRANS_START_INT 中断。(只写)

I2C_NACK_INT_CLR 置位此位，清除 I2C_SLAVE_STRETCH_INT 中断。(只写)

I2C_TXFIFO_OVF_INT_CLR 置位此位，清除 I2C_TXFIFO_OVF_INT 中断。(只写)

I2C_RXFIFO_UDF_INT_CLR 置位此位，清除 I2C_RXFIFO_UDF_INT 中断。(只写)

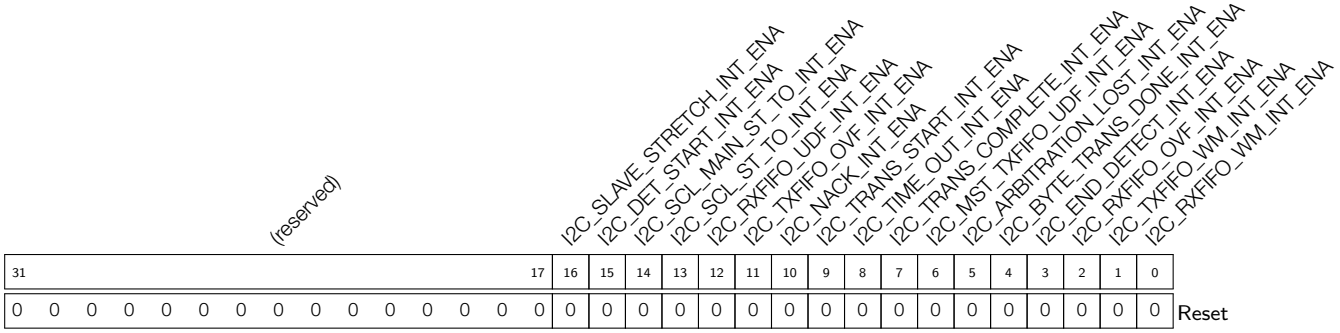
I2C_SCL_ST_TO_INT_CLR 置位此位，清除 I2C_SCL_ST_TO_INT 中断。(只写)

I2C_SCL_MAIN_ST_TO_INT_CLR 置位此位，清除 I2C_SCL_MAIN_ST_TO_INT 中断。(只写)

I2C_DET_START_INT_CLR 置位此位，清除 I2C_DET_START_INT 中断。(只写)

I2C_SLAVE_STRETCH_INT_CLR 置位此位，清除 I2C_SLAVE_STRETCH_INT 中断。(只写)

Register 17.22: I2C_INT_ENA_REG (0x0028)



I2C_RXFIFO_WM_INT_ENA I2C_RXFIFO_WM_INT 的原始中断位。(读/写)

I2C_TXFIFO_WM_INT_ENA I2C_TXFIFO_WM_INT 的原始中断位。(读/写)

I2C_RXFIFO_OVF_INT_ENA I2C_RXFIFO_OVF_INT 的原始中断位。(读/写)

I2C_END_DETECT_INT_ENA I2C_END_DETECT_INT 的原始中断位。(读/写)

I2C_BYTE_TRANS_DONE_INT_ENA I2C_END_DETECT_INT 的原始中断位。(读/写)

I2C_ARBITRATION_LOST_INT_ENA I2C_ARBITRATION_LOST_INT 的原始中断位。(读/写)

I2C_MST_TXFIFO_UDF_INT_ENA I2C_TRANS_COMPLETE_INT 的原始中断位。(读/写)

I2C_TRANS_COMPLETE_INT_ENA I2C_TRANS_COMPLETE_INT 的原始中断位。(读/写)

I2C_TIME_OUT_INT_ENA I2C_TIME_OUT_INT 的原始中断位。(读/写)

I2C_TRANS_START_INT_ENA I2C_TRANS_START_INT 的原始中断位。(读/写)

I2C_NACK_INT_ENA I2C_SLAVE_STRETCH_INT 的原始中断位。(读/写)

I2C_TXFIFO_OVF_INT_ENA I2C_TXFIFO_OVF_INT 的原始中断位。(读/写)

I2C_RXFIFO_UDF_INT_ENA I2C_RXFIFO_UDF_INT 的原始中断位。(读/写)

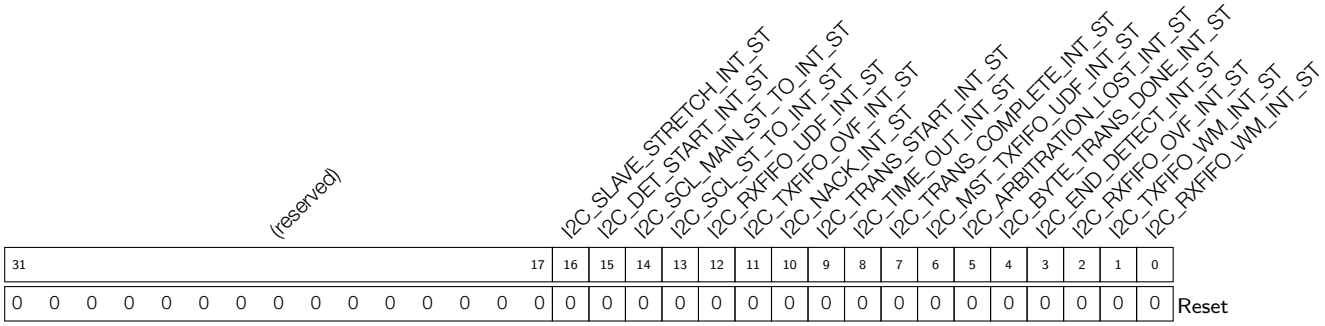
I2C_SCL_ST_TO_INT_ENA I2C_SCL_ST_TO_INT 的原始中断位。(读/写)

I2C_SCL_MAIN_ST_TO_INT_ENA I2C_SCL_MAIN_ST_TO_INT 的原始中断位。(读/写)

I2C_DET_START_INT_ENA I2C_DET_START_INT 的原始中断位。(读/写)

I2C_SLAVE_STRETCH_INT_ENA I2C_SLAVE_STRETCH_INT 的原始中断位。(读/写)

Register 17.23: I2C_INT_STATUS_REG (0x002C)



I2C_RXFIFO_WM_INT_ST I2C_RXFIFO_WM_INT 的屏蔽中断状态位。(只读)

I2C_TXFIFO_WM_INT_ST I2C_TXFIFO_WM_INT 的屏蔽中断状态位。(只读)

I2C_RXFIFO_OVF_INT_ST I2C_RXFIFO_OVF_INT 的屏蔽中断状态位。(只读)

I2C_END_DETECT_INT_ST I2C_END_DETECT_INT 的屏蔽中断状态位。(只读)

I2C_BYTE_TRANS_DONE_INT_ST I2C_END_DETECT_INT 的屏蔽中断状态位。(只读)

I2C_ARBITRATION_LOST_INT_ST I2C_ARBITRATION_LOST_INT 的屏蔽中断状态位。(只读)

I2C_MST_TXFIFO_UDF_INT_ST I2C_TRANS_COMPLETE_INT 的屏蔽中断状态位。(只读)

I2C_TRANS_COMPLETE_INT_ST I2C_TRANS_COMPLETE_INT 的屏蔽中断状态位。(只读)

I2C_TIME_OUT_INT_ST I2C_TIME_OUT_INT 的屏蔽中断状态位。(只读)

I2C_TRANS_START_INT_ST I2C_TRANS_START_INT 的屏蔽中断状态位。(只读)

I2C_NACK_INT_ST I2C_SLAVE_STRETCH_INT 的屏蔽中断状态位。(只读)

I2C_TXFIFO_OVF_INT_ST I2C_TXFIFO_OVF_INT 的屏蔽中断状态位。(只读)

I2C_RXFIFO_UDF_INT_ST I2C_RXFIFO_UDF_INT 的屏蔽中断状态位。(只读)

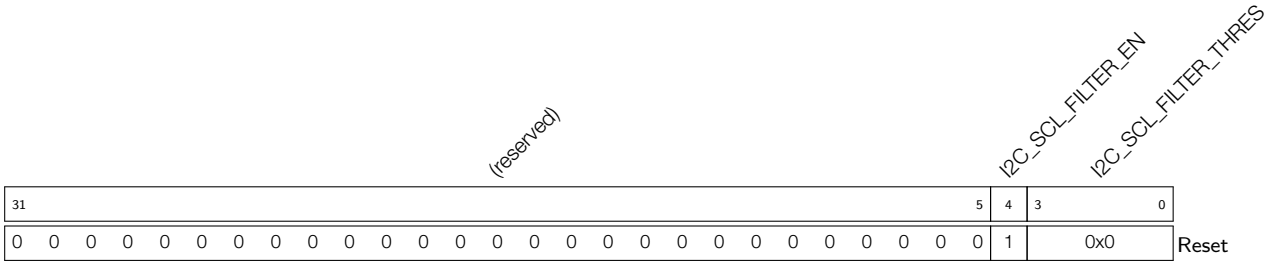
I2C_SCL_ST_TO_INT_ST I2C_SCL_ST_TO_INT 的屏蔽中断状态位。(只读)

I2C_SCL_MAIN_ST_TO_INT_ST I2C_SCL_MAIN_ST_TO_INT 的屏蔽中断状态位。(只读)

I2C_DET_START_INT_ST I2C_DET_START_INT 的屏蔽中断状态位。(只读)

I2C_SLAVE_STRETCH_INT_ST I2C_SLAVE_STRETCH_INT 的屏蔽中断状态位。(只读)

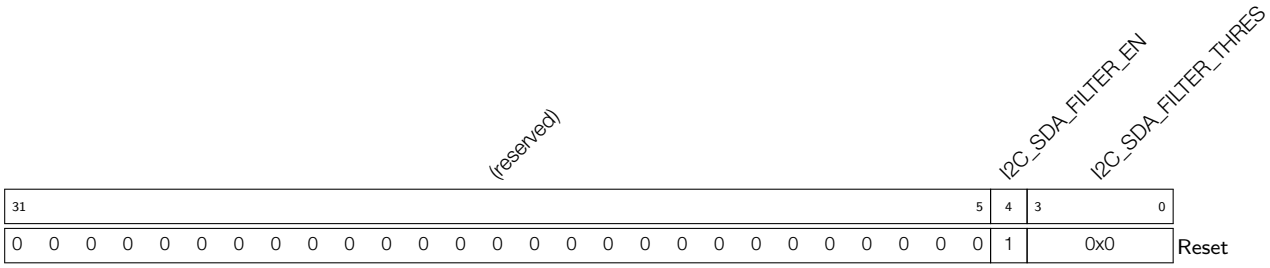
Register 17.24: I2C_SCL_FILTER_CFG_REG (0x0050)



I2C_SCL_FILTER_THRES SCL 输入信号的脉冲宽度小于该寄存器的值时，I2C 控制器忽略此脉冲。该寄存器的值以 I2C 模块时钟周期数为单位。(读/写)

I2C_SCL_FILTER_EN SCL 的滤波使能位。(读/写)

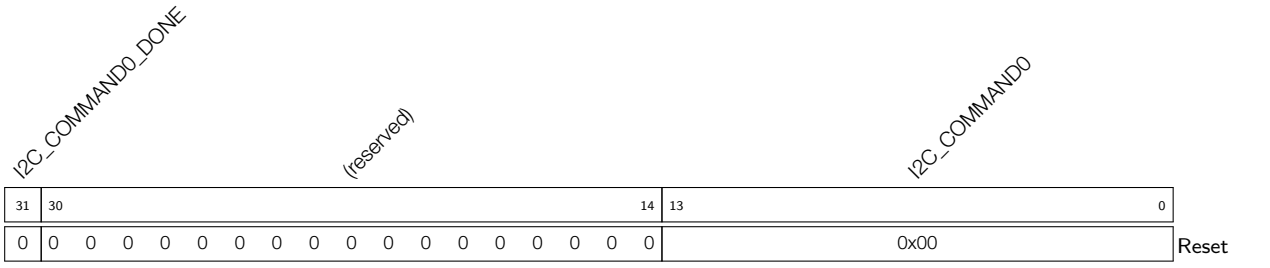
Register 17.25: I2C_SDA_FILTER_CFG_REG (0x0054)



I2C_SDA_FILTER_THRES SDA 输入信号的脉冲宽度小于该寄存器的值时，I2C 控制器忽略此脉冲。该寄存器的值以 I2C 模块时钟周期数为单位。(读/写)

I2C_SDA_FILTER_EN SDA 的滤波使能位。(读/写)

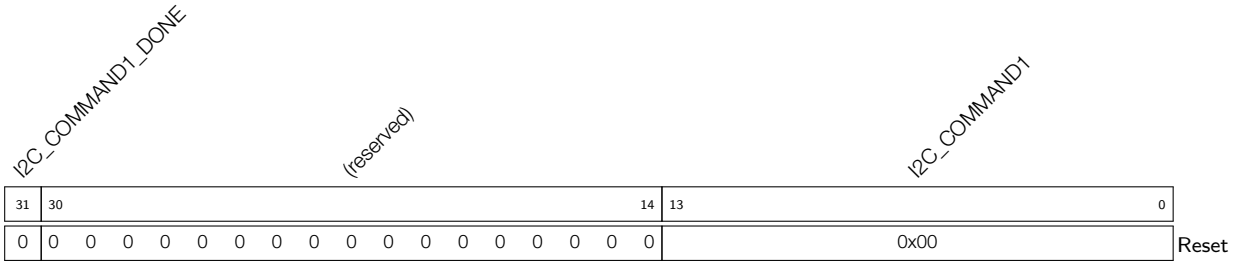
Register 17.26: I2C_COMD0_REG (0x0058)



I2C_COMMAND0 命令 0 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND0_DONE 在 I2C 主机模式下完成命令 0 时，该位翻转为高电平。(读/写)

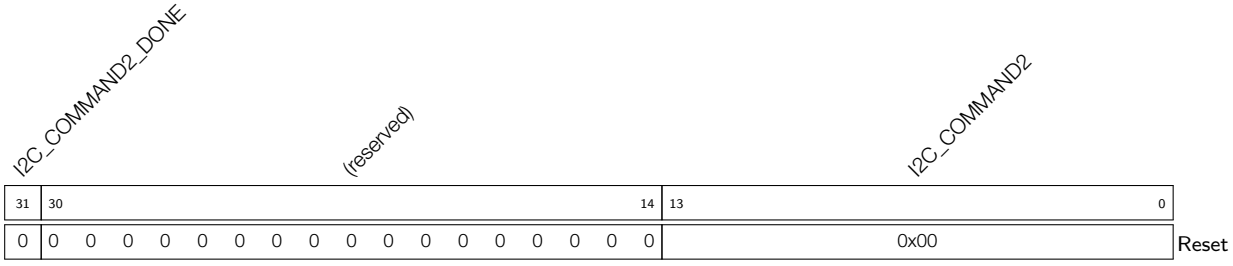
Register 17.27: I2C_COMD1_REG (0x005C)



I2C_COMMAND1 命令 1 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND1_DONE 在 I2C 主机模式下完成命令 1 时，该位翻转为高电平。(读/写)

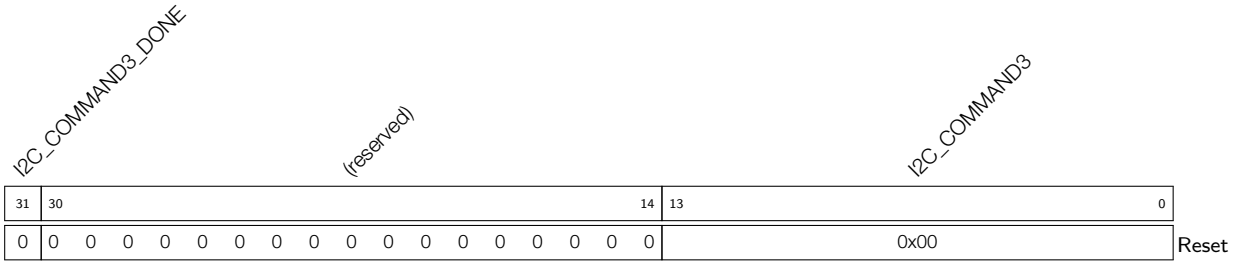
Register 17.28: I2C_COMD2_REG (0x0060)



I2C_COMMAND2 命令 2 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND2_DONE 在 I2C 主机模式下完成命令 2 时，该位翻转为高电平。(读/写)

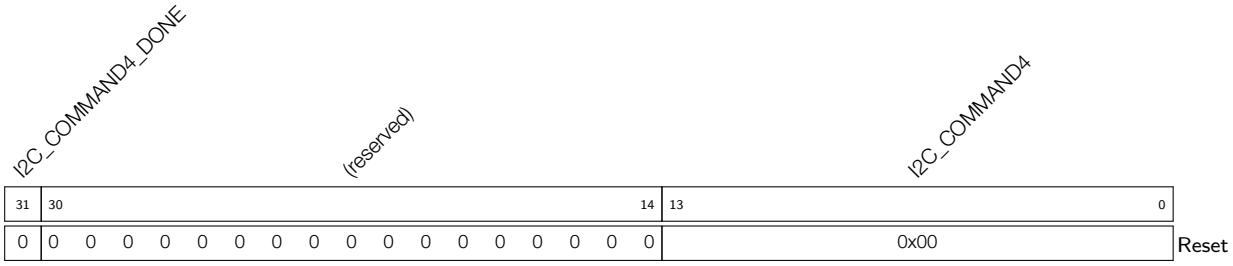
Register 17.29: I2C_COMD3_REG (0x0064)



I2C_COMMAND3 命令 3 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND3_DONE 在 I2C 主机模式下完成命令 3 时，该位翻转为高电平。(读/写)

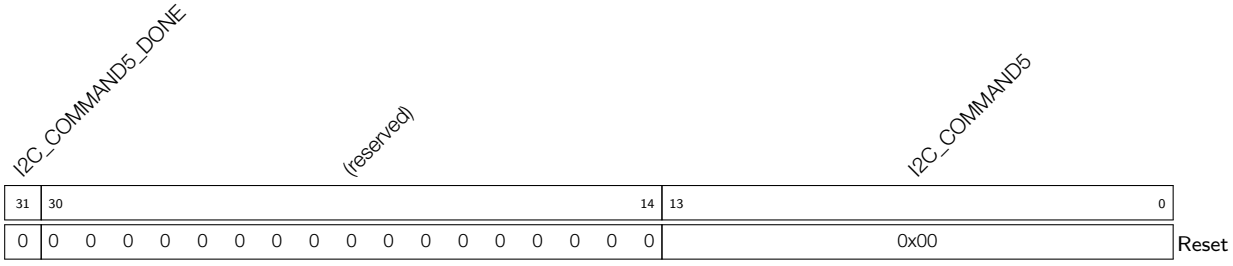
Register 17.30: I2C_COMD4_REG (0x0068)



I2C_COMMAND4 命令 4 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND4_DONE 在 I2C 主机模式下完成命令 4 时，该位翻转为高电平。(读/写)

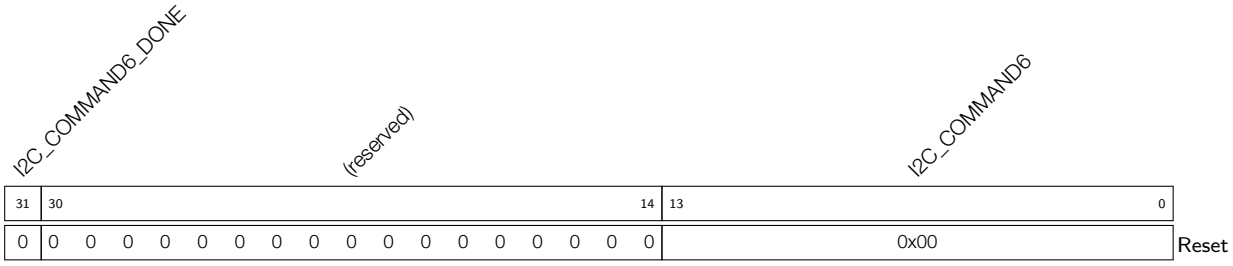
Register 17.31: I2C_COMD5_REG (0x006C)



I2C_COMMAND5 命令 5 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND5_DONE 在 I2C 主机模式下完成命令 5 时，该位翻转为高电平。(读/写)

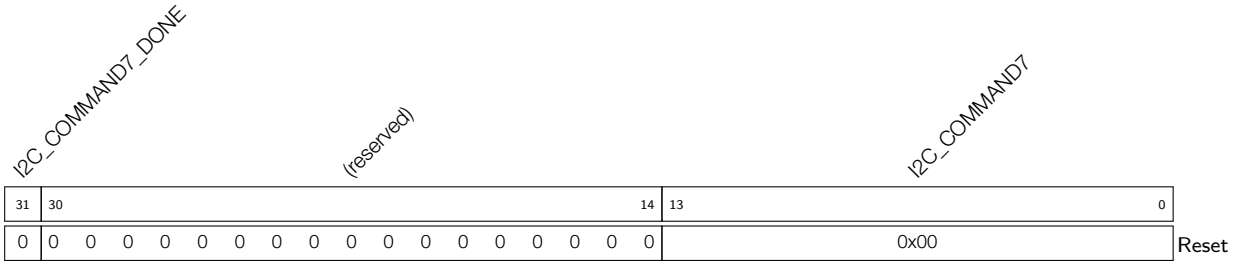
Register 17.32: I2C_COMD6_REG (0x0070)



I2C_COMMAND6 命令 6 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND6_DONE 在 I2C 主机模式下完成命令 6 时，该位翻转为高电平。(读/写)

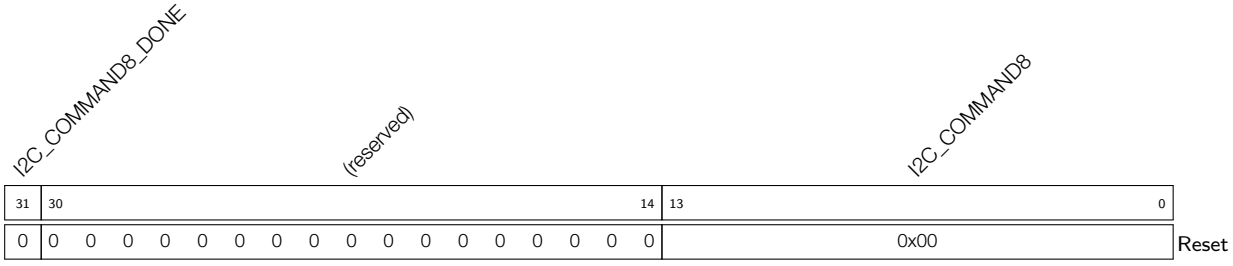
Register 17.33: I2C_COMD7_REG (0x0074)



I2C_COMMAND7 命令 7 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND7_DONE 在 I2C 主机模式下完成命令 7 时，该位翻转为高电平。(读/写)

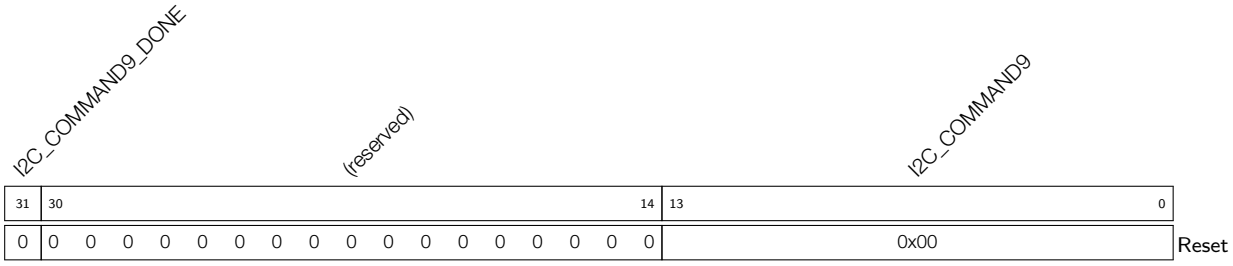
Register 17.34: I2C_COMD8_REG (0x0078)



I2C_COMMAND8 命令 8 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND8_DONE 在 I2C 主机模式下完成命令 8 时，该位翻转为高电平。(读/写)

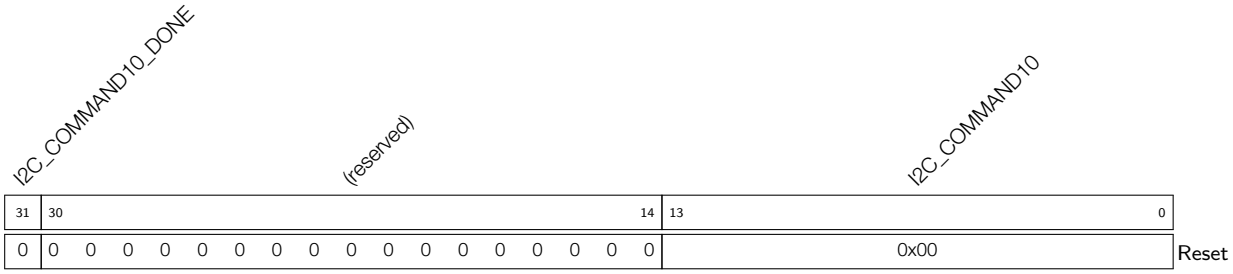
Register 17.35: I2C_COMD9_REG (0x007C)



I2C_COMMAND9 命令 9 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND9_DONE 在 I2C 主机模式下完成命令 9 时，该位翻转为高电平。(读/写)

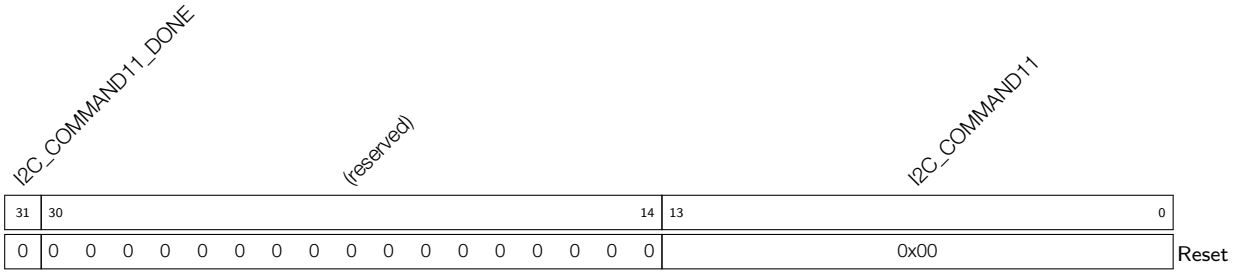
Register 17.36: I2C_COMD10_REG (0x0080)



I2C_COMMAND10 命令 10 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND10_DONE 在 I2C 主机模式下完成命令 10 时，该位翻转为高电平。(读/写)

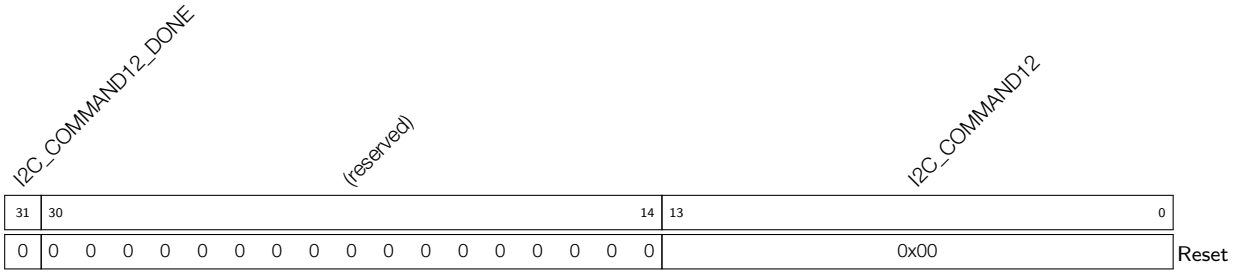
Register 17.37: I2C_COMD11_REG (0x0084)



I2C_COMMAND11 命令 11 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND11_DONE 在 I2C 主机模式下完成命令 11 时，该位翻转为高电平。(读/写)

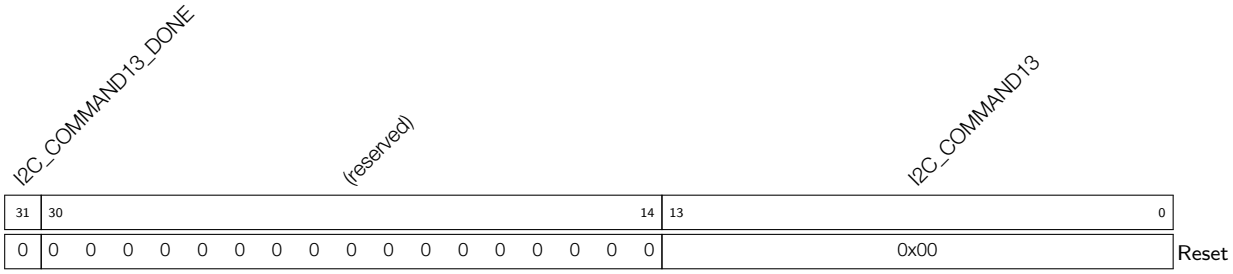
Register 17.38: I2C_COMD12_REG (0x0088)



I2C_COMMAND12 命令 12 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND12_DONE 在 I2C 主机模式下完成命令 12 时，该位翻转为高电平。(读/写)

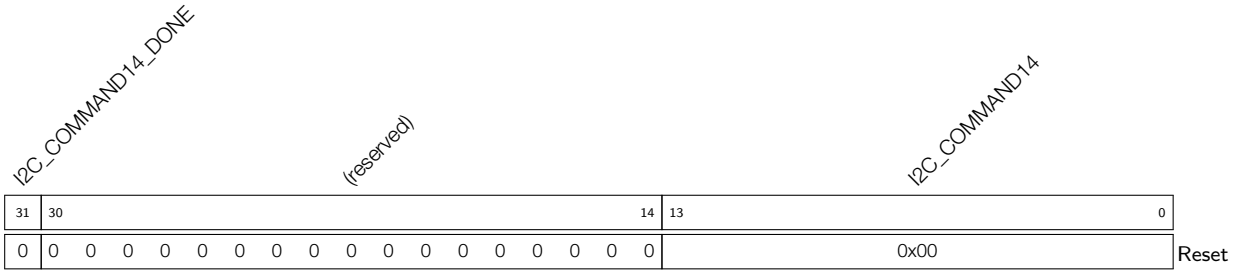
Register 17.39: I2C_COMD13_REG (0x008C)



I2C_COMMAND13 命令 13 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND13_DONE 在 I2C 主机模式下完成命令 13 时，该位翻转为高电平。(读/写)

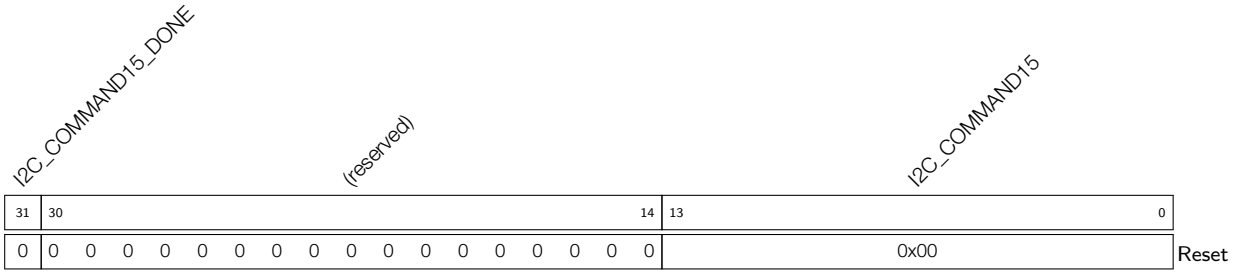
Register 17.40: I2C_COMD14_REG (0x0090)



I2C_COMMAND14 命令 14 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND14_DONE 在 I2C 主机模式下完成命令 14 时，该位翻转为高电平。(读/写)

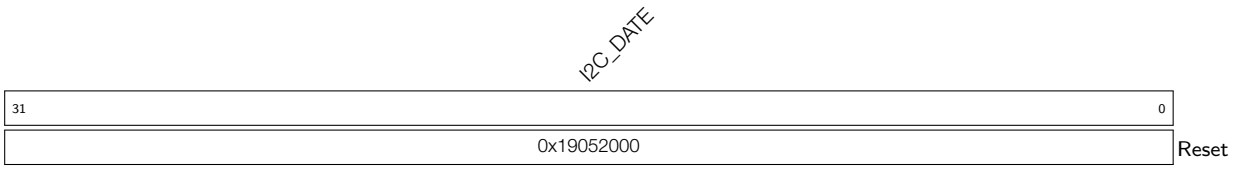
Register 17.41: I2C_COMD15_REG (0x0094)



I2C_COMMAND15 命令 15 的内容。该命令包括三个部分：op_code 为命令，0：START；1：WRITE；2：READ；3：STOP；4：END。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。参阅 I2C cmd 结构获取更多信息。(读/写)

I2C_COMMAND15_DONE 在 I2C 主机模式下完成命令 15 时，该位翻转为高电平。(读/写)

Register 17.42: I2C_DATE_REG (0x00F8)



I2C_DATE 版本控制寄存器。(读/写)

18. TWAI

18.1 概述

双线车载串口 (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 TWAI 协议描述）。

ESP32-S2 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。

18.2 主要特性

ESP32-S2 TWAI 控制器具有以下特性：

- 兼容 ISO 11898-1 协议
- 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

18.3 功能性协议

18.3.1 TWAI 性能

TWAI 协议连接总线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）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 18.3.2 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 18.3.3 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。

18.3.2 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

18.3.2.1 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 18-1 所示为不同帧类型和不同帧格式中包含的域和子域。

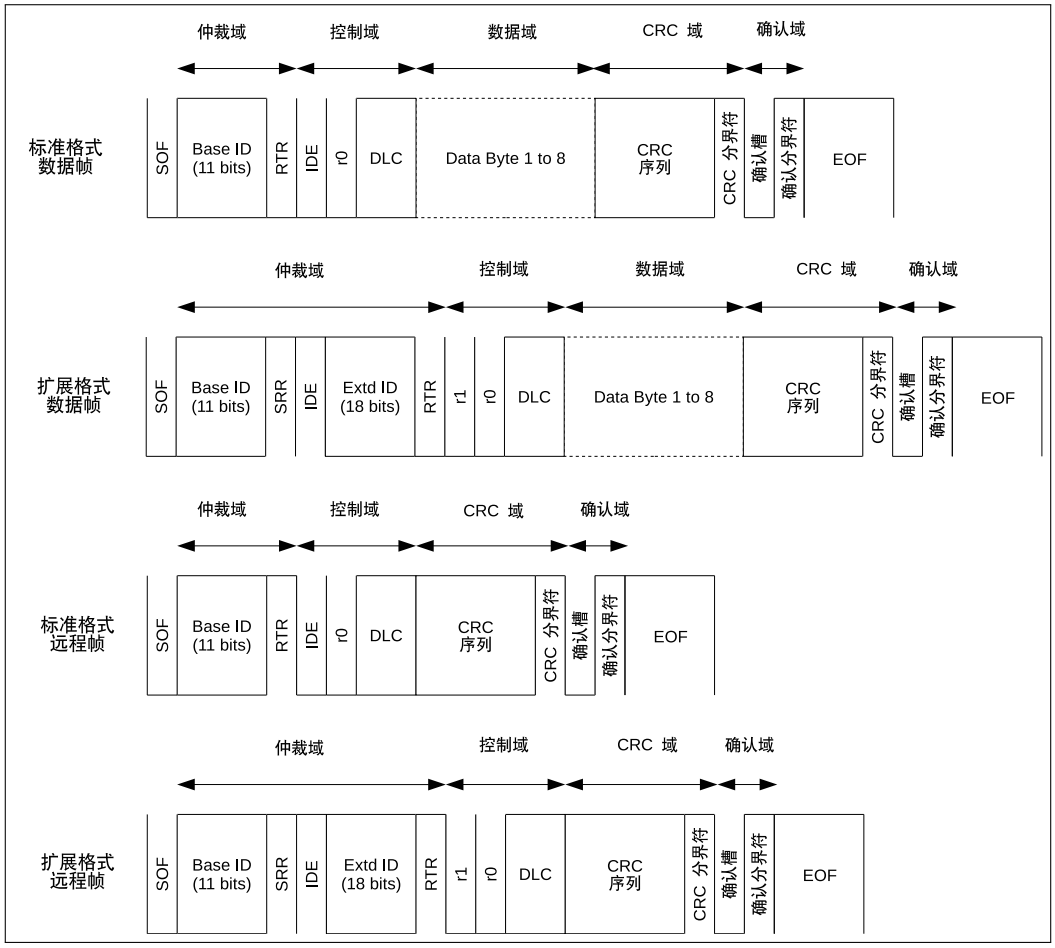


图 18-1. 数据帧和远程帧中的位域

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 65: 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。

18.3.2.2 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 18.3.3 章）。下图 18-2 所示为一个错误帧所包含的不同域：

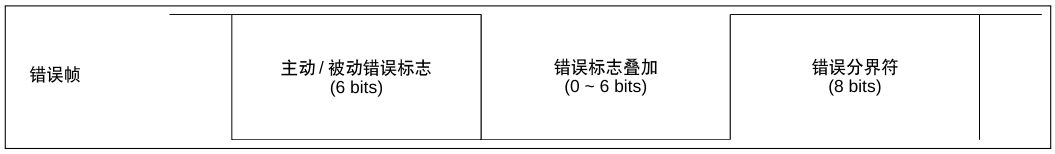


图 18-2. 错误帧中的位域

表 66: 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 18-3 所示为过载帧中包含的位域：

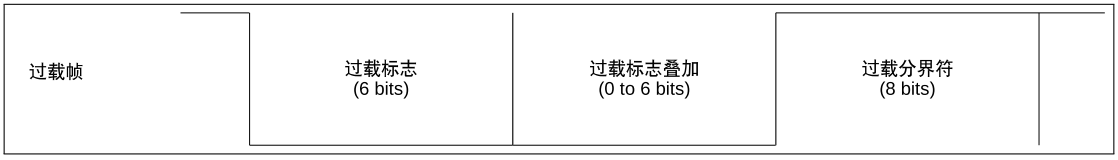


图 18-3. 过载帧中的位域

表 67: 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 18.3.3 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。

18.3.2.3 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 18-4 所示为帧间距中包含的域：

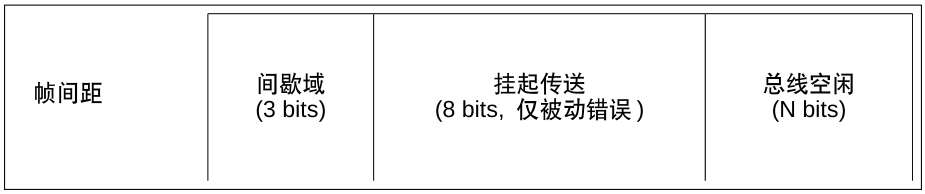


图 18-4. 帧间距中的域

表 68: 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。

18.3.3 TWAI 错误

18.3.3.1 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。

18.3.3.2 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）。

18.3.3.3 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）。

18.3.4 TWAI 位时序

18.3.4.1 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 18-5 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。

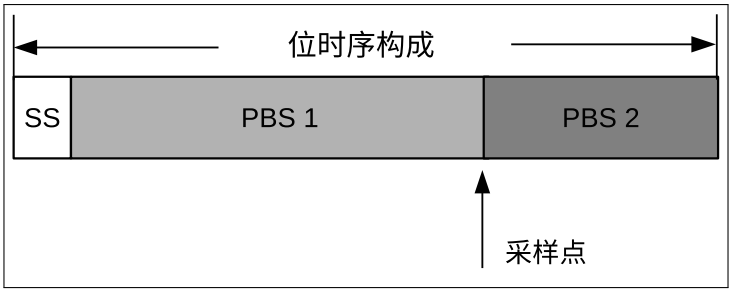


图 18-5. 位时序构成

表 69: 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。

18.3.4.2 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 SOF 位上）。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。

18.4 结构概述

ESP32-S2 中包含一个 TWAI 控制器。图 18-6 所示为 TWAI 控制器的主要功能模块。

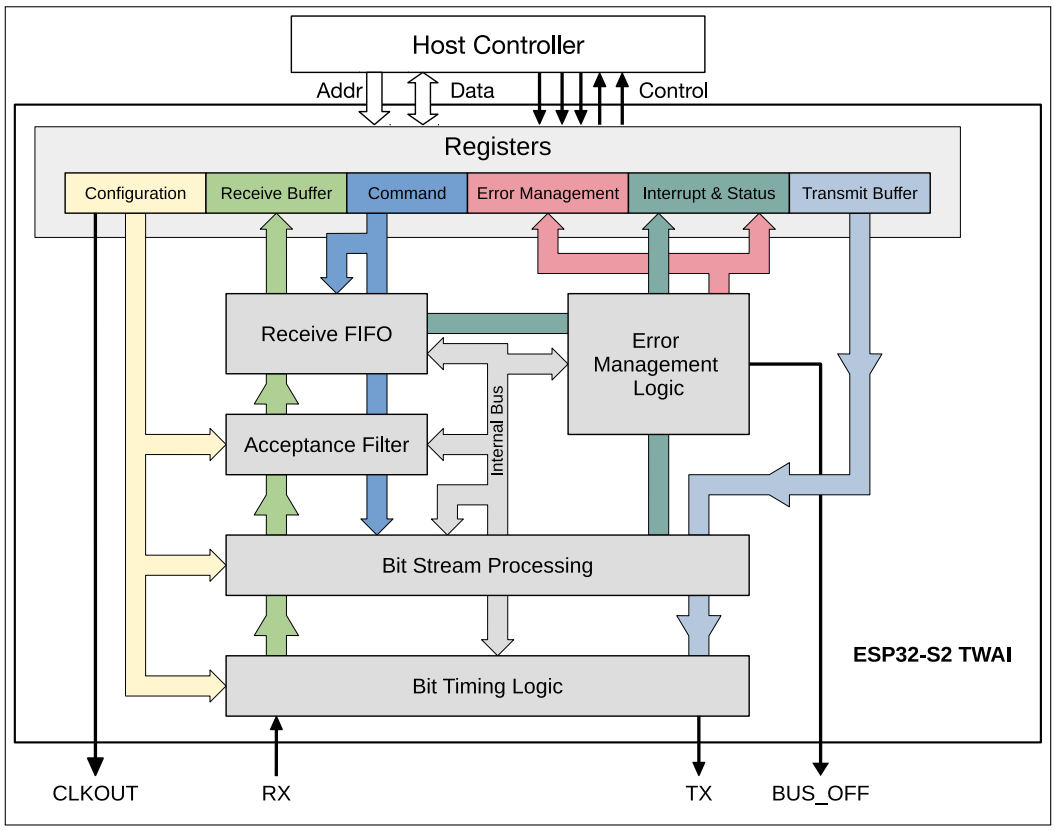


图 18-6. TWAI 概略图

18.4.1 寄存器模块

ESP32-S2 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 18.5.1 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 18.5.1 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。

18.4.2 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)。

18.4.3 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。

18.4.4 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，来调整报文传输速率。

18.4.5 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。

18.4.6 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。

18.5 功能描述

18.5.1 模式

ESP32-S2 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。

18.5.1.1 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。

18.5.1.2 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）。

18.5.2 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 70 所示为 TWAI_BUS_TIMING_0_REG 包含的位域。

表 70: TWAI_CLOCK_DIVIDER_REG 的 bit 信息; TWAI 地址 0x18

Bit 31-16	Bit 15	Bit 14	Bit 13	Bit 12	.....	Bit 1	Bit 0
保留	SJW.1	SJW.0	BRP.13	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：
  $t_{Tq} = 2 \times t_{CLK} \times (2^{13} \times BRP.13 + 2^{12} \times BRP.12 + ... + 2^{1} \times BRP.1 + 2^{0} \times BRP.0 + 1)$
- 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 71 所示为 TWAI_BUS_TIMING_1_REG 包含的位域。

表 71: TWAI_BUS_TIMING_1_REG 的 bit 信息; TWAI 地址 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。

18.5.3 中断管理

ESP32-S2 TWAI 控制器提供了七种中断，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 控制器提供了以下七种中断：

- 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

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，接收中断不包括在内，直到通过 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。

18.5.3.1 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。

18.5.3.2 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。

18.5.3.3 错误报警中断 (EWI)

每当寄存器 TWAI_STATUS_REG 中 TWAI_ERR_ST 或 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 或 TWAI_BUS_OFF_ST 的值分成以下几种情况：

- 如果 TWAI_ERR_ST = 0 或 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 或 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 或 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 或 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。

18.5.3.4 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。

18.5.3.5 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。

18.5.3.6 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。

18.5.3.7 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。

18.5.4 发送缓冲器与接收缓冲器

18.5.4.1 缓冲器概述

表 72: 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

标准格式 (SFF)		扩展格式 (EFF)	
TWAI 地址	内容	TWAI 地址	内容
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 72 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。

接收缓冲寄存器映射到接收 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。

18.5.4.2 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 73 所示为帧信息域。

表 73: 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	XDLC.3	DLC.2	DLC.1	DLC.0

说明：

- 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。
- 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。
- 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。
- X: 无关 bit，可以是任意值。

18.5.4.3 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 74-75 所示为 SFF (11-bits) 报文的帧标识符域。

表 74: 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.28	ID.27	ID.26	ID.25	ID.24	ID.23	ID.22	ID.21

表 75: 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.20	ID.19	ID.18	X[1]	X[2]	X[2]	X[2]	X[2]

下表 76-79 所示为 EFF (29-bits) 报文的帧标识符域。

表 76: 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 77: 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 78: 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 79: 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]

18.5.4.4 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 个数据字节的数据帧时，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。

18.5.5 接收 FIFO 和数据溢出

接收 FIFO 是一个 64-bytes 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。

18.5.6 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 18-7 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。

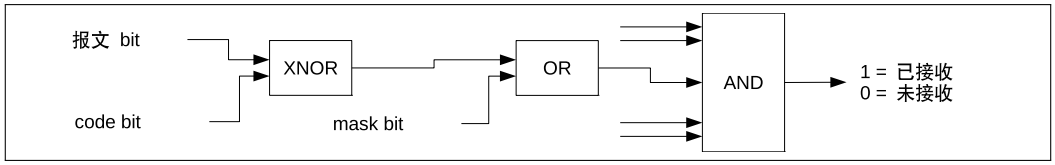


图 18-7. 接收滤波器

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）。

18.5.6.1 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 18-8 所示为单滤波模式下如何解析 32-bit code/mask 的值。

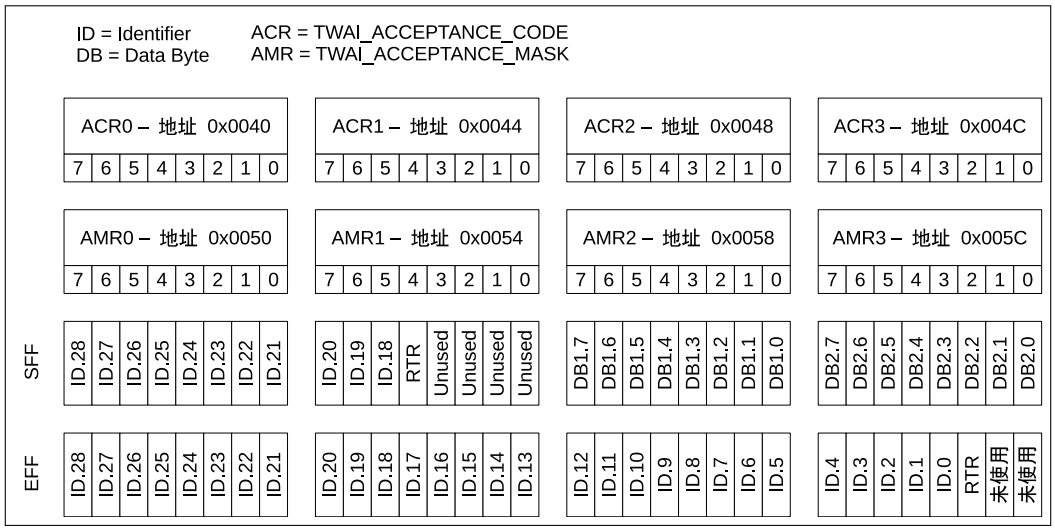


图 18-8. 单滤波模式

18.5.6.2 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 18-9 所示为双滤波模式下如何解析 32-bit code/mask 的值。

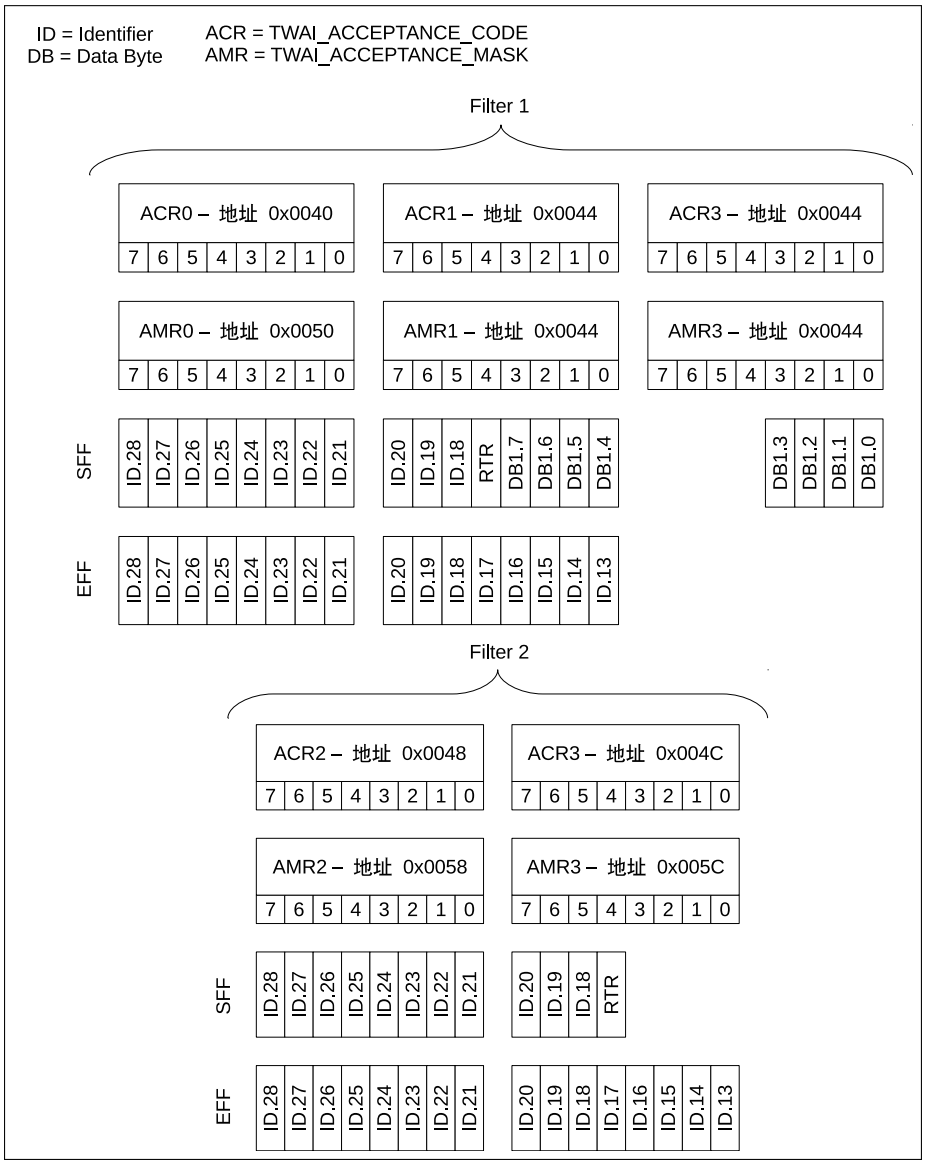


图 18-9. 双滤波模式

18.5.7 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 18.5.3 章）。下图 18-10 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。

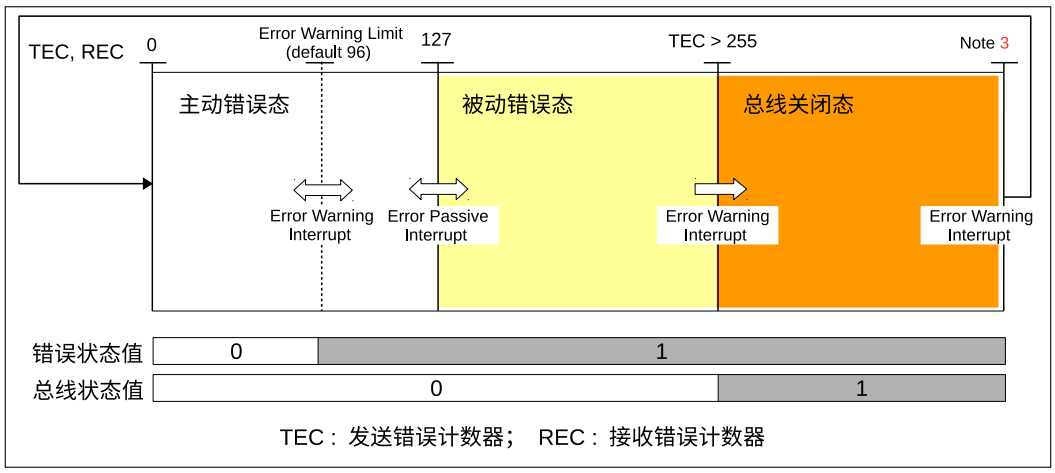


图 18-10. 错误状态变化

18.5.7.1 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。

18.5.7.2 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。

18.5.7.3 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。

18.5.8 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 80 所示为寄存器 TWAI_ERR_CODE_CAP_REG 中的域：

表 80: TWAI_ERR_CODE_CAP_REG 的 bit 信息; TWAI 地址 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 81 所示为 SEG.0 ~ SEG.4 的位信息。

表 81: 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

Bit SEG.4	Bit SEG.3	Bit SEG.2	Bit SEG.1	Bit SEG.0	描述
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。

18.5.9 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 82 所示为 TWAI_ERR_CODE_CAP_REG 中的位域；下图 18-11 所示为一条 TWAI 报文的 bit 位置。

表 82: TWAI_ARB LOST CAP_REG 中的位信息; TWAI 地址 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。

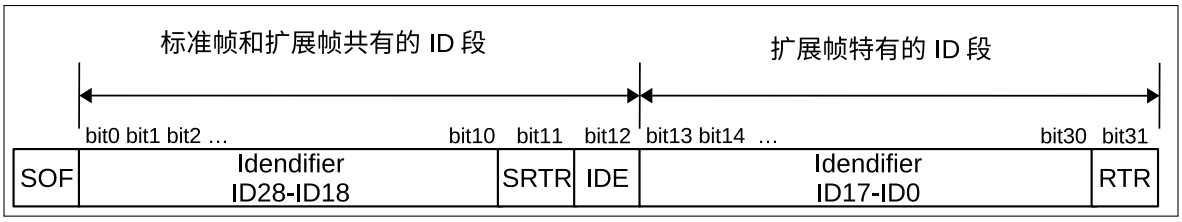


图 18-11. 丢失仲裁的 bit 位置

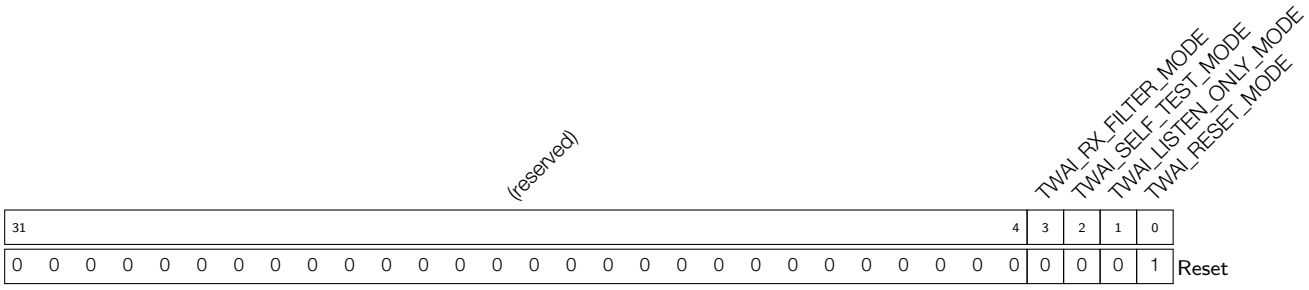
18.6 寄存器列表

名称	描述	地址	访问权限
配置寄存器			
TWAI_MODE_REG	模式寄存器	0x0000	读/写
TWAI_BUS_TIMING_0_REG	时序配置寄存器 0	0x0018	只读 I 读/写
TWAI_BUS_TIMING_1_REG	时序配置寄存器 1	0x001C	只读 I 读/写
TWAI_ERR_WARNING_LIMIT_REG	错误寄存器	0x0034	只读 I 读/写
TWAI_DATA_0_REG	数据寄存器 0	0x0040	只写 I 读/写
TWAI_DATA_1_REG	数据寄存器 1	0x0044	只写 I 读/写
TWAI_DATA_2_REG	数据寄存器 2	0x0048	只写 I 读/写
TWAI_DATA_3_REG	数据寄存器 3	0x004C	只写 I 读/写
TWAI_DATA_4_REG	数据寄存器 4	0x0050	只写 I 读/写

名称	描述	地址	访问权限
TWAI_DATA_5_REG	数据寄存器 5	0x0054	只写 I 读/写
TWAI_DATA_6_REG	数据寄存器 6	0x0058	只写 I 读/写
TWAI_DATA_7_REG	数据寄存器 7	0x005C	只写 I 读/写
TWAI_DATA_8_REG	数据寄存器 8	0x0060	只写 I 只读
TWAI_DATA_9_REG	数据寄存器 9	0x0064	只写 I 只读
TWAI_DATA_10_REG	数据寄存器 10	0x0068	只写 I 只读
TWAI_DATA_11_REG	数据寄存器 11	0x006C	只写 I 只读
TWAI_DATA_12_REG	数据寄存器 12	0x0070	只写 I 只读
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	只写
状态寄存器			
TWAI_STATUS_REG	状态寄存器	0x0008	只读
TWAI_ARB LOST CAP_REG	仲裁丢失寄存器	0x002C	只读
TWAI_ERR_CODE_CAP_REG	错误捕获寄存器	0x0030	只读
TWAI_RX_ERR_CNT_REG	接收错误寄存器	0x0038	只读 I 读/写
TWAI_TX_ERR_CNT_REG	发送错误寄存器	0x003C	只读 I 读/写
TWAI_RX_MESSAGE_CNT_REG	接收数据寄存器	0x0074	只读
中断寄存器			
TWAI_INT_RAW_REG	中断寄存器	0x000C	只读
TWAI_INT ENA_REG	中断使能寄存器	0x0010	读/写

18.7 寄存器

Register 18.1: TWAI_MODE_REG (0x0000)



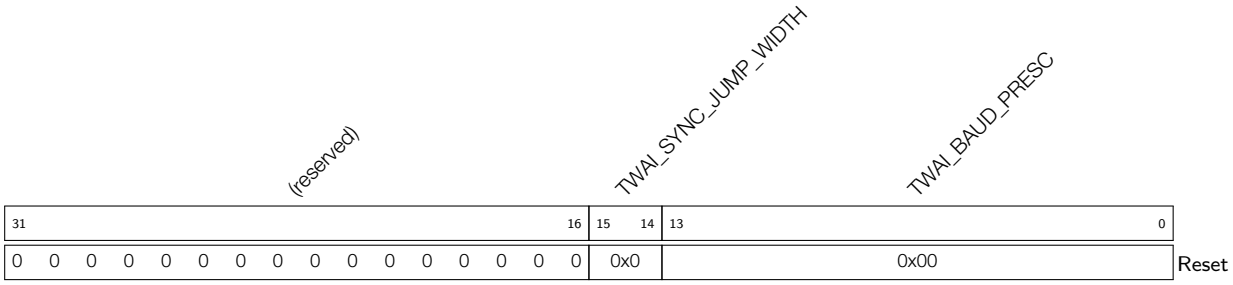
TWAI_RESET_MODE 配置 TWAI 控制器操作模式。1：复位模式；0：操作模式（读/写）

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。(读/写)

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。(读/写)

TWAI_RX_FILTER_MODE 配置滤波模式。0: 双滤波模式；1: 单滤波模式（读/写）

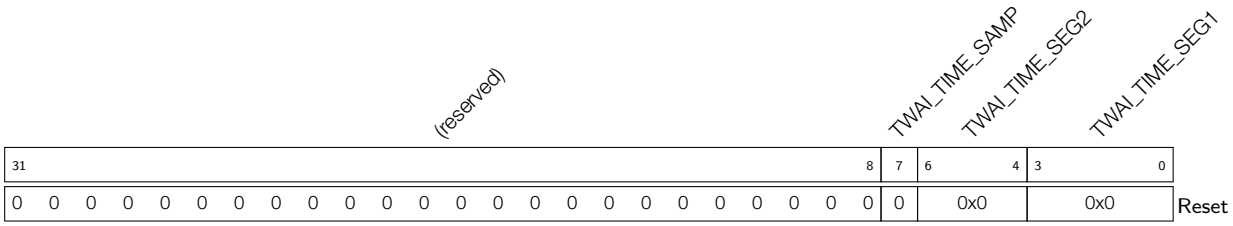
Register 18.2: TWAI_BUS_TIMING_0_REG (0x0018)



TWAI_BAUD_PRESC 预分频值，决定分频比例。(只读 | 读/写)

TWAI_SYNC_JUMP_WIDTH 同步跳宽 (SJW)，范围为 1 ~ 4 个时间定额。(只读 | 读/写)

Register 18.3: TWAI_BUS_TIMING_1_REG (0x001C)

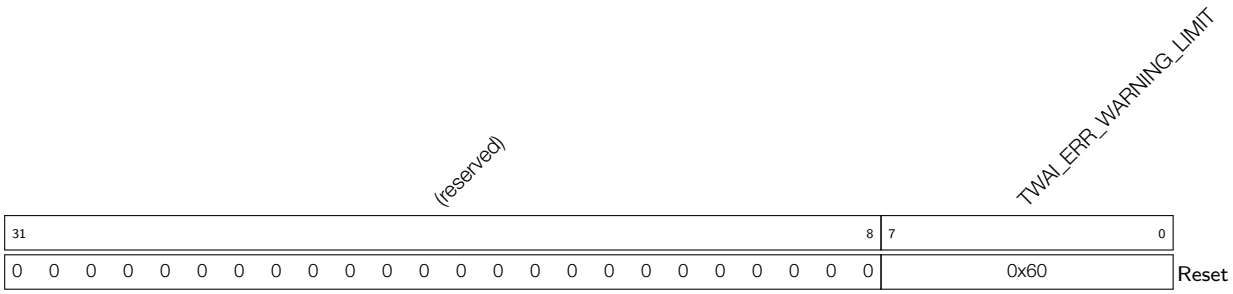


TWAI_TIME_SEG1 缓冲时期段 1 的宽度。(只读 | 读/写)

TWAI_TIME_SEG2 缓冲时期段 2 的宽度。(只读 | 读/写)

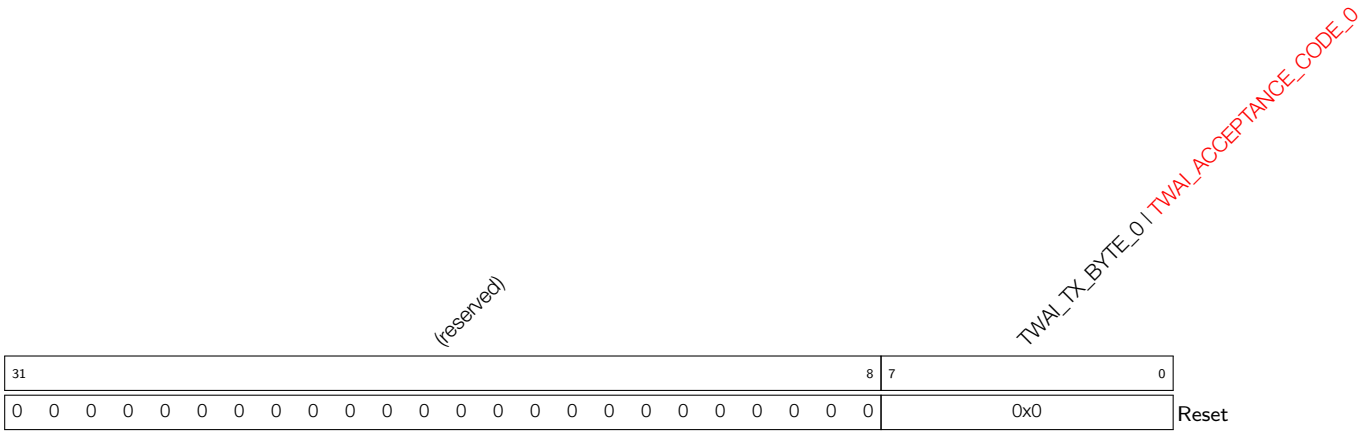
TWAI_TIME_SAMP 采样点数目。0：采样 1 次；1：采样三次（只读 | 读/写)

Register 18.4: 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)。(只读 | 读/写)

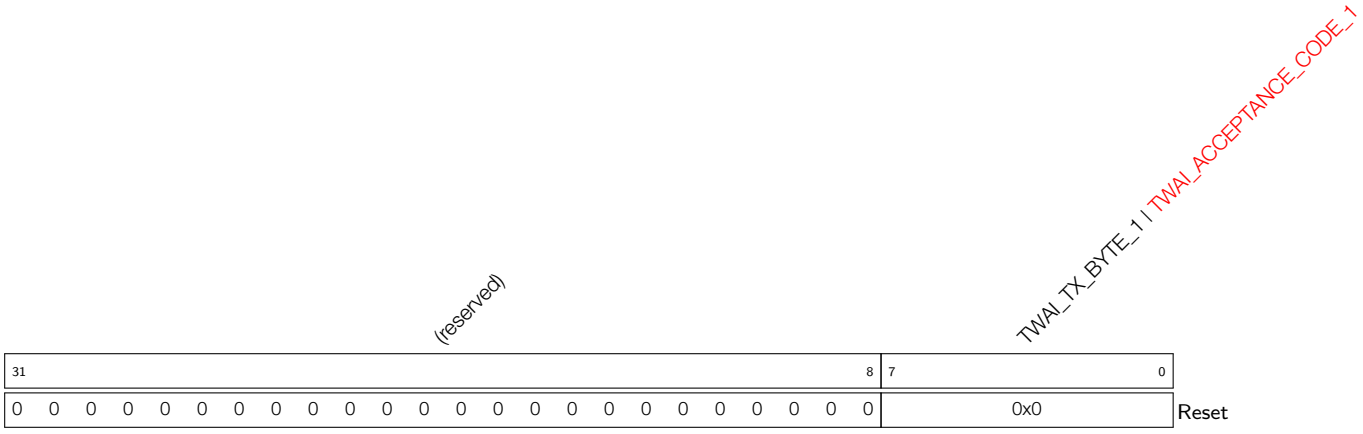
Register 18.5: TWAI_DATA_0_REG (0x0040)



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。（只写）

TWAI_ACCEPTANCE_CODE_0 复位模式下，存储着滤波编码的第 0 个字节。（读/写）

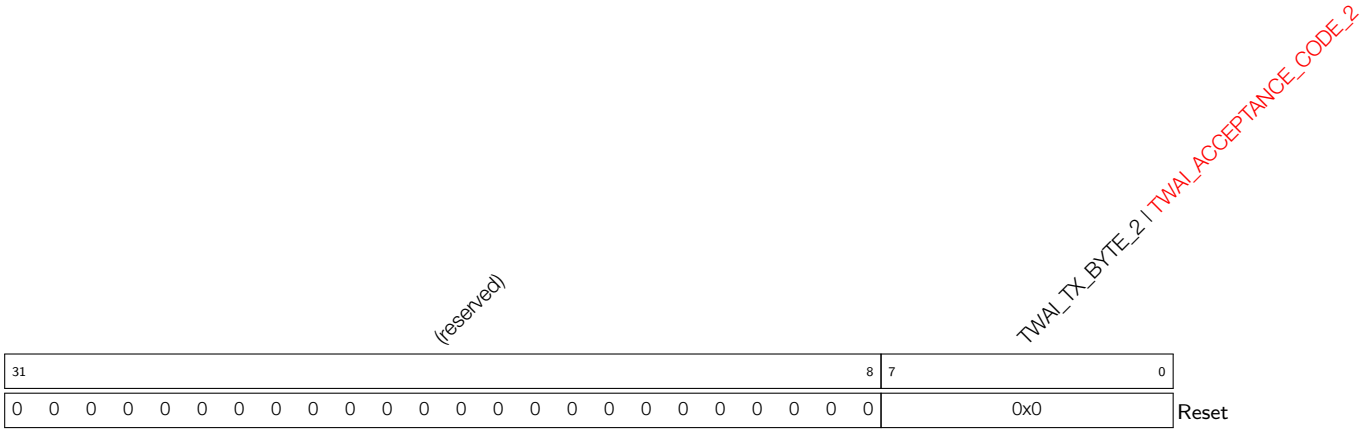
Register 18.6: TWAI_DATA_1_REG (0x0044)



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。（只写）

TWAI_ACCEPTANCE_CODE_1 复位模式下，存储着滤波编码的第 1 个字节。（读/写）

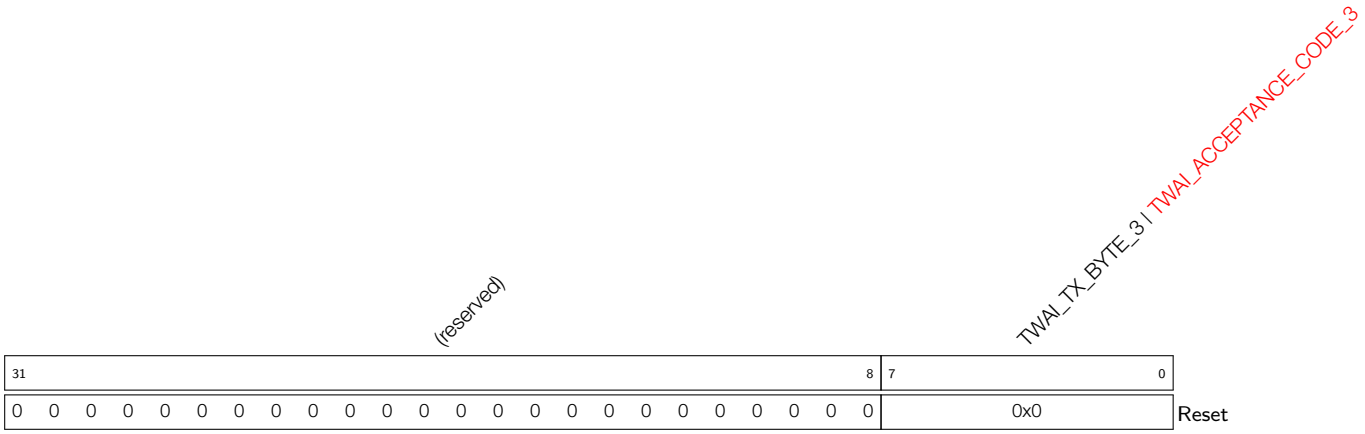
Register 18.7: TWAI_DATA_2_REG (0x0048)



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。（只写）

TWAI_ACCEPTANCE_CODE_2 复位模式下，存储着滤波编码的第 2 个字节。（读/写）

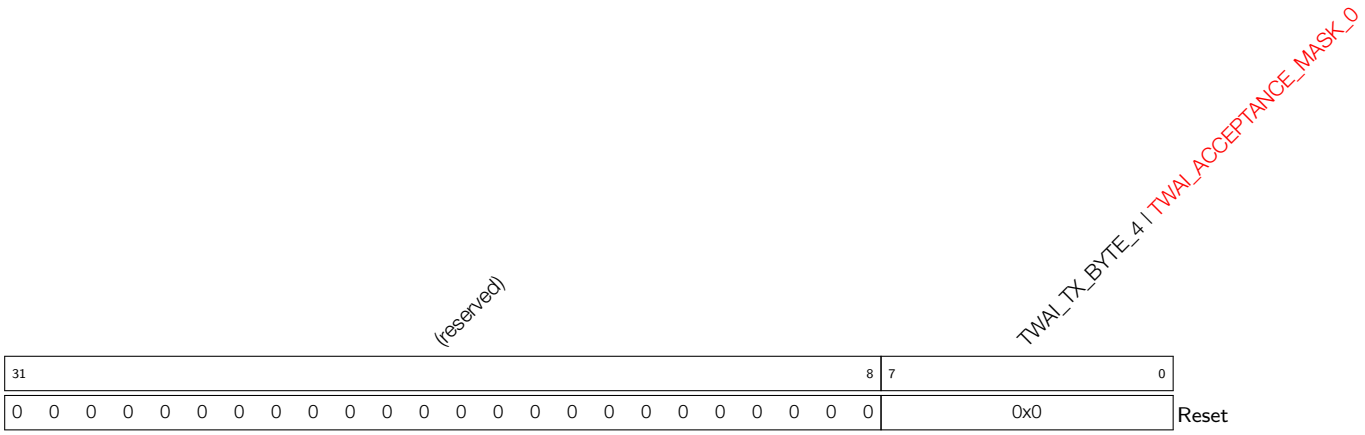
Register 18.8: TWAI_DATA_3_REG (0x004C)



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。（只写）

TWAI_ACCEPTANCE_CODE_3 复位模式下，存储着滤波编码的第 3 个字节。（读/写）

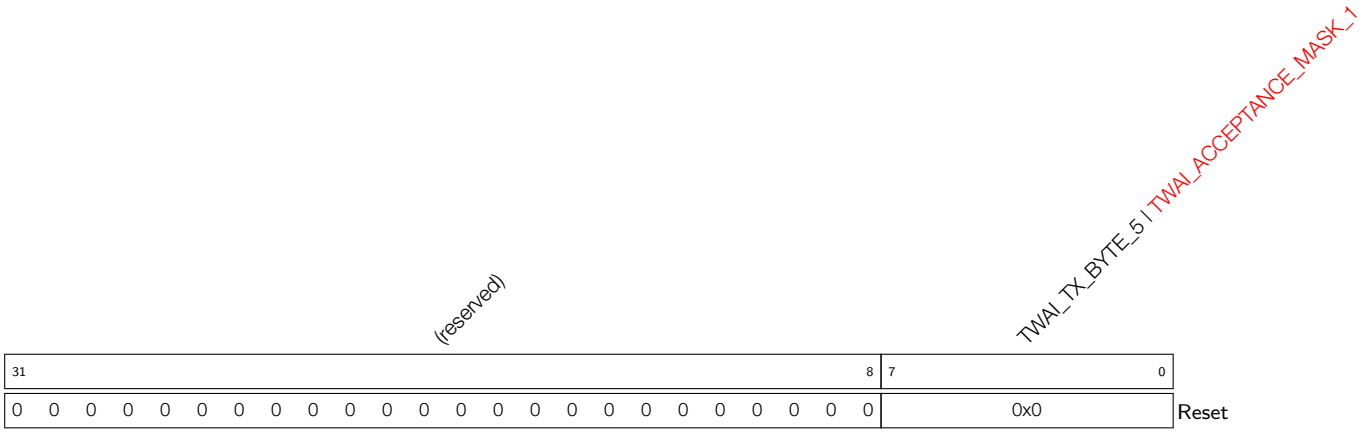
Register 18.9: TWAI_DATA_4_REG (0x0050)



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。（只写）

TWAI_ACCEPTANCE_MASK_0 复位模式下，存储着滤波编码的第 0 个字节。（读/写）

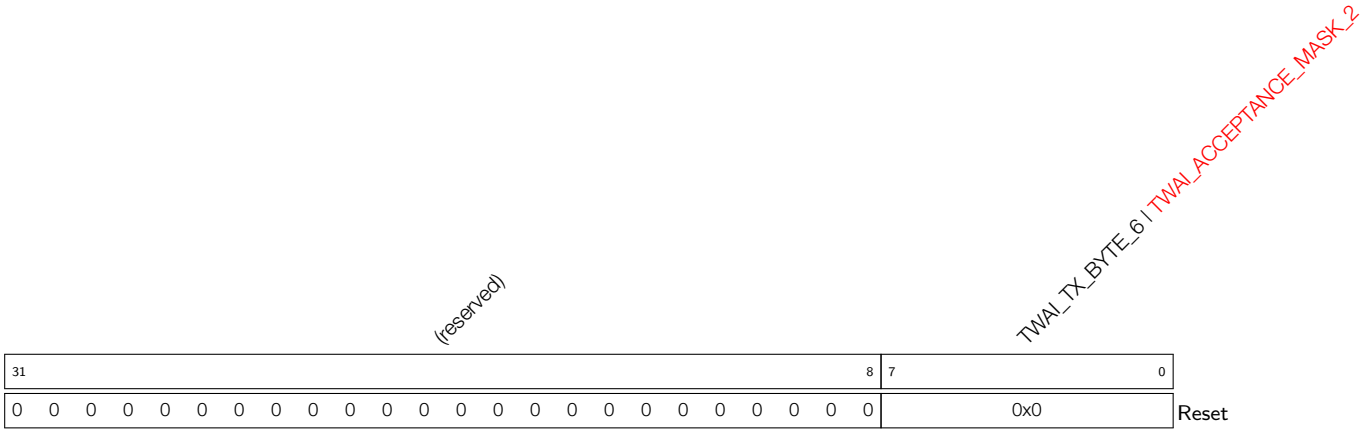
Register 18.10: TWAI_DATA_5_REG (0x0054)



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。（只写）

TWAI_ACCEPTANCE_MASK_1 复位模式下，存储着滤波编码的第 1 个字节。（读/写）

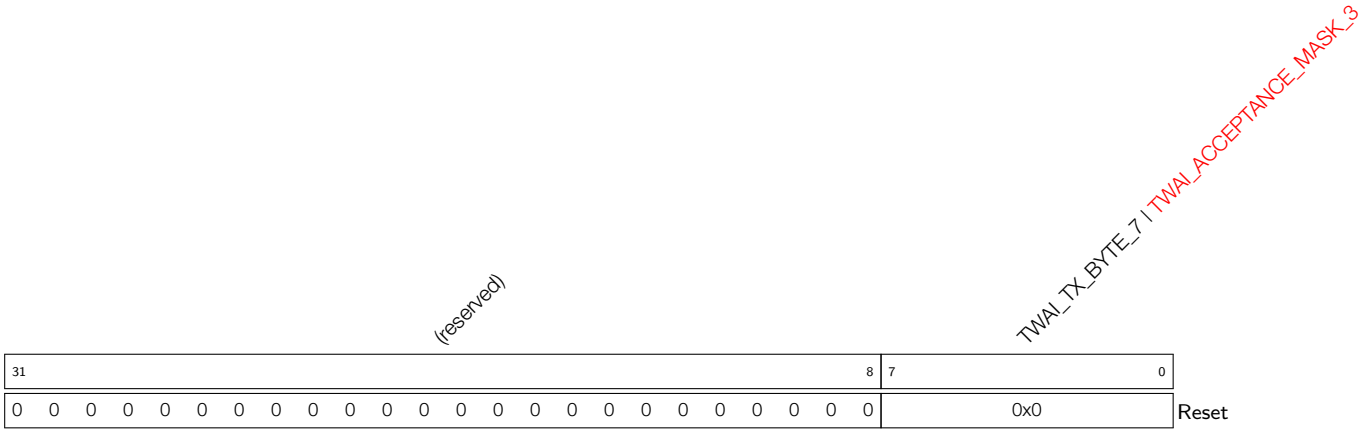
Register 18.11: TWAI_DATA_6_REG (0x0058)



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。（只写）

TWAI_ACCEPTANCE_MASK_2 复位模式下，存储着滤波编码的第 2 个字节。（读/写）

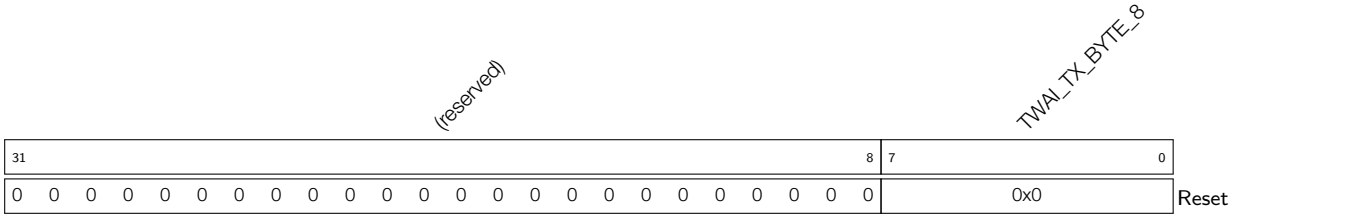
Register 18.12: TWAI_DATA_7_REG (0x005C)



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。（只写）

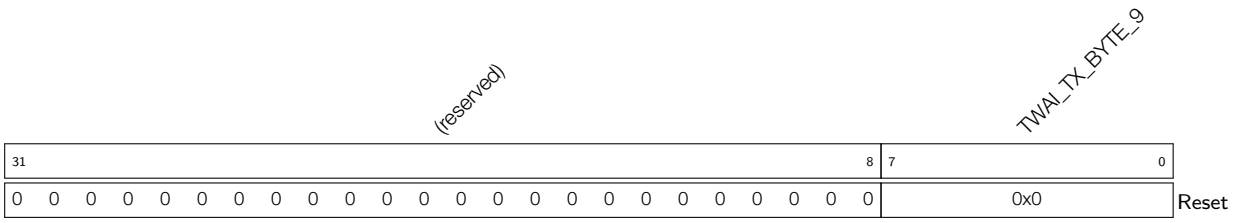
TWAI_ACCEPTANCE_MASK_3 复位模式下，存储着滤波编码的第 3 个字节。（读/写）

Register 18.13: TWAI_DATA_8_REG (0x0060)



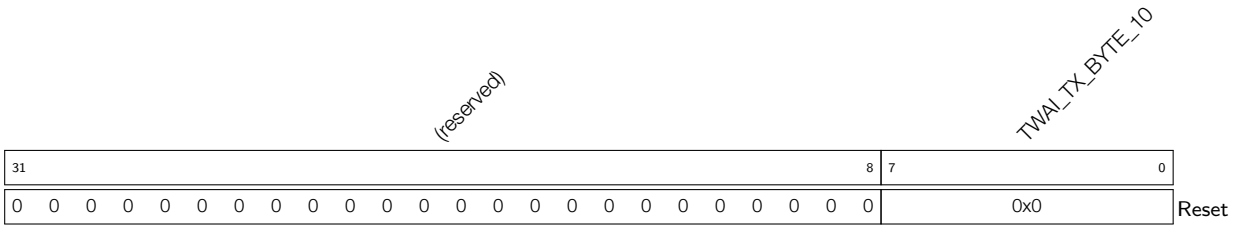
TWAI_TX_BYTE_8 操作模式下，存储着待发送数据的第 8 个字节内容。（只写）

Register 18.14: TWAI_DATA_9_REG (0x0064)



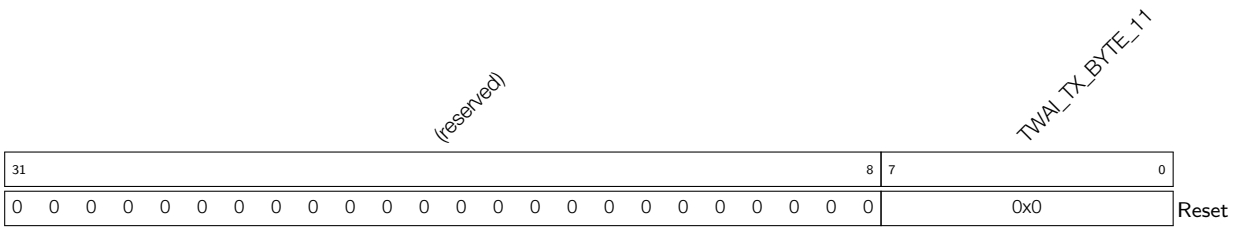
TWAI_TX_BYTE_9 操作模式下，存储着待发送数据的第 9 个字节内容。（只写）

Register 18.15: TWAI_DATA_10_REG (0x0068)



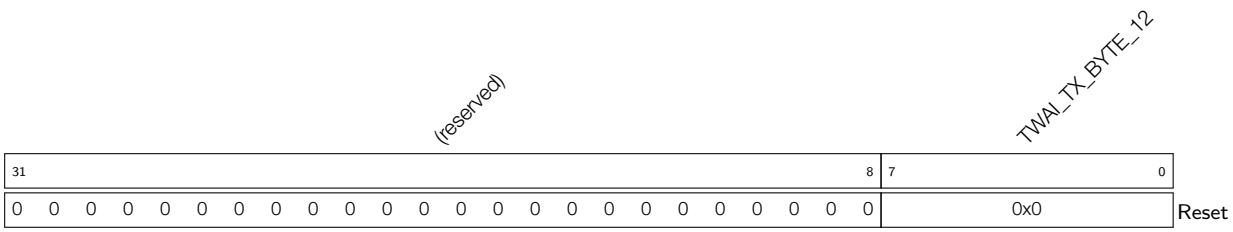
TWAI_TX_BYTE_10 操作模式下，存储着待发送数据的第 10 个字节内容。（只写）

Register 18.16: TWAI_DATA_11_REG (0x006C)



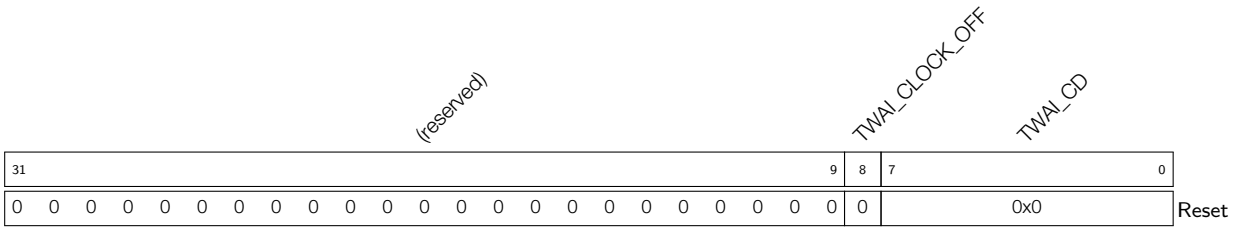
TWAI_TX_BYTE_11 操作模式下，存储着待发送数据的第 11 个字节内容。（只写）

Register 18.17: 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。（只写）

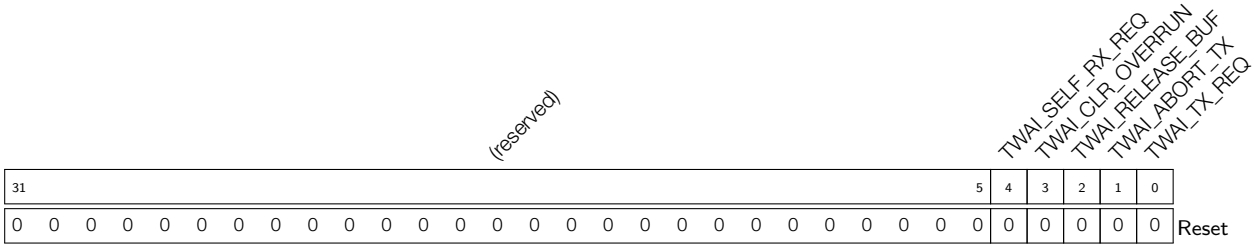
Register 18.18: TWAI_CLOCK_DIVIDER_REG (0x007C)



TWAI_CD 配置输出时钟 CLKOUT 的分频系数。(读/写)

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（只读 | 读/写）

Register 18.19: TWAI_CMD_REG (0x0004)



TWAI_TX_REQ 置 1 驱动节点开始发送数据任务。(只写)

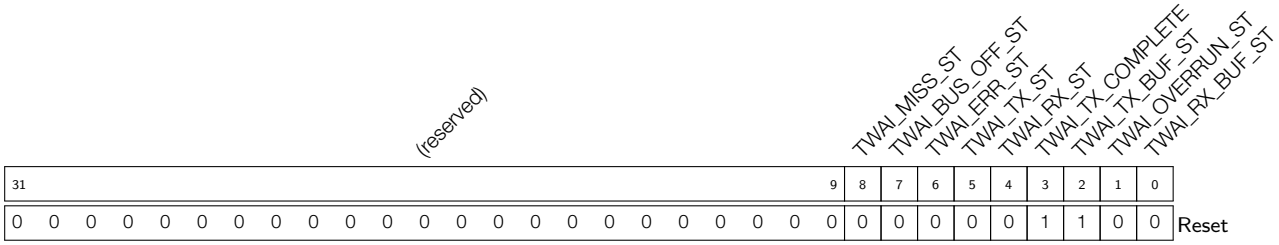
TWAI_ABORT_TX 置 1 取消当前还未开始的发送任务。(只写)

TWAI_RELEASE_BUF 置 1 释放接收缓冲器。(只写)

TWAI_CLR_OVERRUN 置 1 清除数据溢出状态。(只写)

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。(只写)

Register 18.20: 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。(只读)

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。(只读)

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。(只读)

TWAI_TX_COMPLETE 若值为 1，表明成功从总线上接收到一个数据包。(只读)

TWAI_RX_ST 若值为 1，表明节点正在从总线上接收数据。(只读)

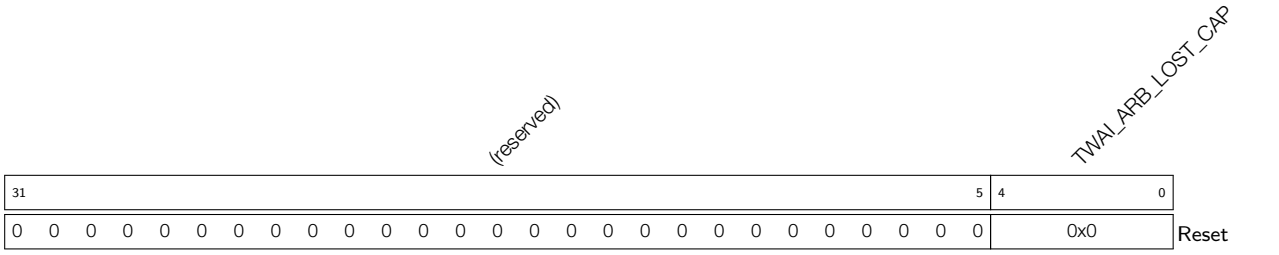
TWAI_TX_ST 若值为 1，表明节点正在往总线上发送数据。(只读)

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（只读）

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。(只读)

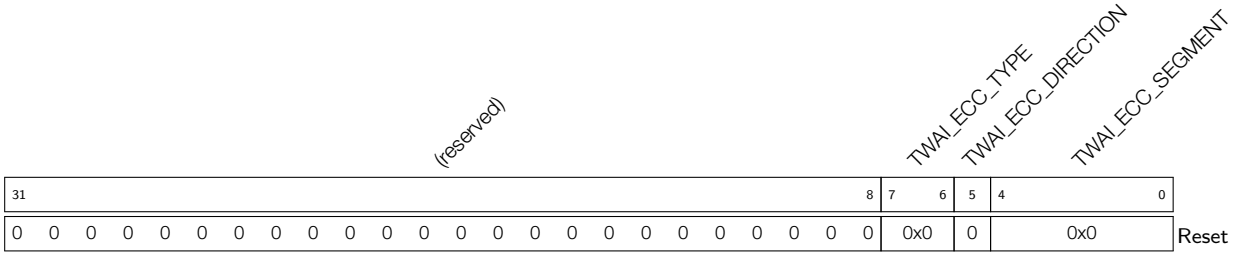
TWAI_MISS_ST 反映了从接收 FIFO 中取出数据包的完整状态。1: 当前数据包是缺失的；0：当前数据包是完整的（只读）

Register 18.21: 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。(只读)

Register 18.22: TWAI_ERR_CODE_CAP_REG (0x0030)

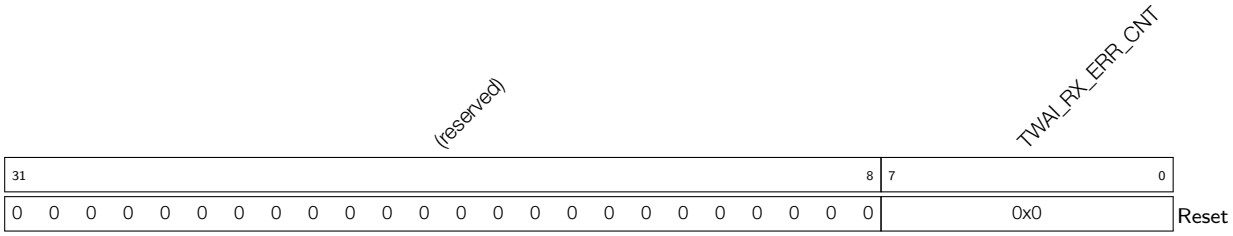


TWAI_ECC_SEGMENT 记录错误发生的位置，详见表 80。（只读）

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（只读）

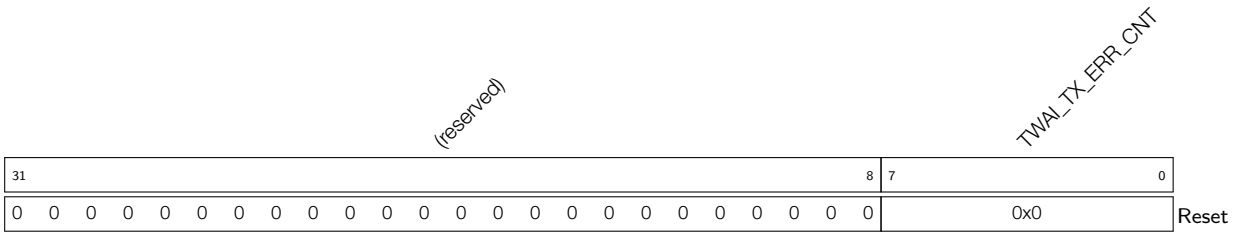
TWAI_ECC_TYPE 记录错误类别：00：位错误；01：格式错误；10：填充错误；11：其他错误（只读）

Register 18.23: TWAI_RX_ERR_CNT_REG (0x0038)



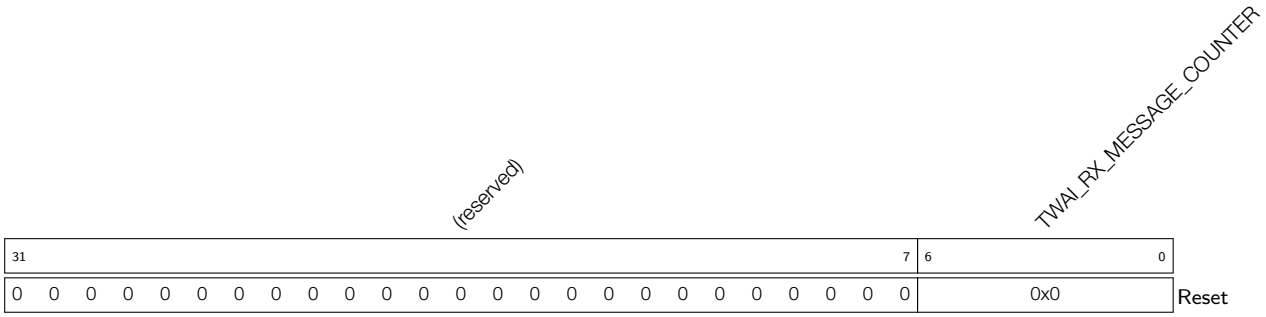
TWAI_RX_ERR_CNT 接收错误计数，数值变化发生在接收状态下。（只读 | 读/写）

Register 18.24: TWAI_TX_ERR_CNT_REG (0x003C)



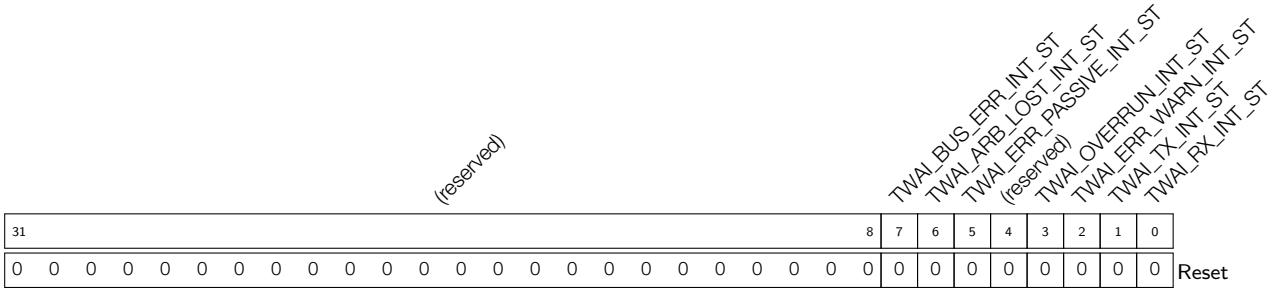
TWAI_TX_ERR_CNT 发送错误计数，数值变化发生在发送状态下。（只读 | 读/写）

Register 18.25: TWAI_RX_MESSAGE_CNT_REG (0x0074)



TWAI_RX_MESSAGE_COUNTER 存储着接收 FIFO 中数据包的个数。(只读)

Register 18.26: 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。(只读)

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。(只读)

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)。(只读)

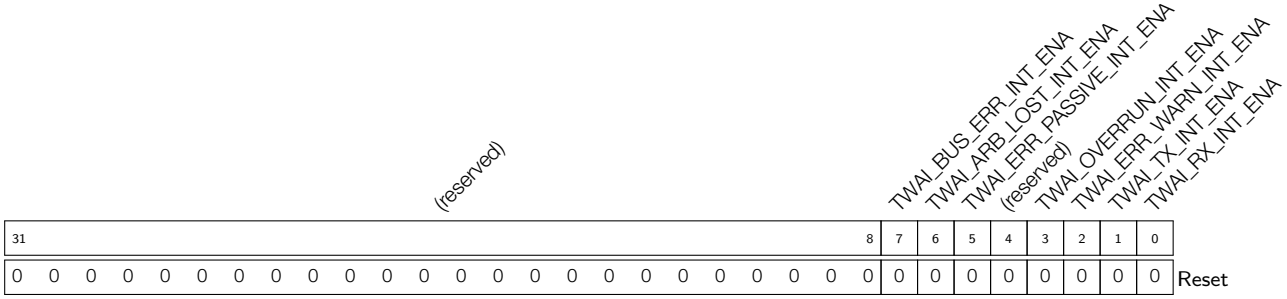
TWAI_OVERRUN_INT_ST 数据溢出中断。若值为 1，表明节点的接收 FIFO 数据溢出。(只读)

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。(只读)

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。(只读)

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。(只读)

Register 18.27: TWAI_INT ENA_REG (0x0010)



TWAI_RX_INT_ENA 置 1 使能接收中断。(读/写)

TWAI_TX_INT_ENA 置 1 使能发送中断。(读/写)

TWAI_ERR_WARN_INT_ENA 置 1 使能错误报警中断。(读/写)

TWAI_OVERRUN_INT_ENA 置 1 使能数据溢出中断。(读/写)

TWAI_ERR_PASSIVE_INT_ENA 置 1 使能被动错误中断。(读/写)

TWAI_ARB_LOST_INT_ENA 置 1 使能仲裁丢失中断。(读/写)

TWAI_BUS_ERR_INT_ENA 置 1 使能错误中断。(读/写)

19. AES 加速器

19.1 概述

ESP32-S2 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-S2 支持以下特性：

- Typical AES 工作模式
  - AES-128/AES-192/AES-256 加解密运算
  - 4 种密钥字节序和 4 种文本字节序
- DMA-AES 工作模式
  - 块模式
    - ECB (Electronic Codebook)
    - CBC (Cipher Block Chaining)
    - OFB (Output Feedback)
    - CTR (Counter)
    - CFB8 (8-bit Cipher Feedback)
    - CFB128 (128-bit Cipher Feedback)
  - GCM (Galois/Counter Mode)
  - 中断发生

19.3 工作模式简介

ESP32-S2 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 NIST FIPS 197，能够实现 AES-128、AES-192、AES-256 加密与解密运算。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 NIST SP 800-38A 标准中的 ECB/CBC/OFB/CTR/CFB8/CFB128 等块加密模式运算以及 NIST SP 800-38D 标准中的 GCM 运算。在这种情况下，明文/密文的传输通过硬件上的 crypto 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 84。

表 84: 工作模式

AES_DMA_ENABLE_REG	工作模式
0	Typical AES

1	DMA-AES

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-S2 的数字签名和片外存储手动加密模块也会调用 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 加速器支持 AES-128/AES-192/AES-256 加解密共 6 种运算类型。用户可通过配置 AES_MODE_REG 寄存器选择具体运算类型，具体可参考表 85。

表 85: 运算模式选择

AES_MODE_REG[2:0]	运算模式
0	AES-128 加密
1	AES-192 加密
2	AES-256 加密
4	AES-128 解密
5	AES-192 解密
6	AES-256 解密

AES 加速器的状态值可查看寄存器 AES_STATE_REG，具体见表 86 所示：

表 86: 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

在 Typical AES 工作模式下，AES 加速器每加密一个信息块需要 11 ~ 15 个时钟周期，每解密一个信息块需要 21 或 22 个时钟周期。

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-192 加解密运算，则 192 位密钥在寄存器 AES_KEY_0_REG ~ AES_KEY_5_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 用于存放明文或密文，各由 4 个 32 位寄存器组成。

- 如果为 AES-128/192/265 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-128/192/256 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。其中，输入文本的字节序由寄存器 AES_ENDIAN_REG 的 Bit 2 和 Bit 3 控制，输出文本字节序由 Bit 4 和 Bit 5 控制。具体来说，Bit 2 和 Bit 4 控制每个 word 内 4 个 byte 的顺序，Bit 3 和 Bit 5 控制每个 block 中 4 个 word 的顺序。

不难看出，通过配置 AES_ENDIAN_REG 寄存器，AES 加速器可允许四种文本字节序。表 87 指定了在四种不同字节序下，寄存器 AES_TEXT_IN_*m*_REG 和 AES_TEXT_OUT_*m*_REG 中的每个 word 所存放的明文或密文将如何构成 State。

表 87: Typical AES 文本字节序

Word Endian 控制位	Byte Endian 控制位	明文/密文 [2]				
0	0	State[1]		c		
				0	1	2	3
		r	0	AES_TEXT_*x*_3_REG[31:24]	AES_TEXT_*x*_2_REG[31:24]	AES_TEXT_*x*_1_REG[31:24]	AES_TEXT_*x*_0_REG[31:24]
			1	AES_TEXT_*x*_3_REG[23:16]	AES_TEXT_*x*_2_REG[23:16]	AES_TEXT_*x*_1_REG[23:16]	AES_TEXT_*x*_0_REG[23:16]
			2	AES_TEXT_*x*_3_REG[15:8]	AES_TEXT_*x*_2_REG[15:8]	AES_TEXT_*x*_1_REG[15:8]	AES_TEXT_*x*_0_REG[15:8]
			3	AES_TEXT_*x*_3_REG[7:0]	AES_TEXT_*x*_2_REG[7:0]	AES_TEXT_*x*_1_REG[7:0]	AES_TEXT_*x*_0_REG[7:0]
0	1	State		c			
				0	1	2	3
		r	0	AES_TEXT_*x*_3_REG[7:0]	AES_TEXT_*x*_2_REG[7:0]	AES_TEXT_*x*_1_REG[7:0]	AES_TEXT_*x*_0_REG[7:0]
			1	AES_TEXT_*x*_3_REG[15:8]	AES_TEXT_*x*_2_REG[15:8]	AES_TEXT_*x*_1_REG[15:8]	AES_TEXT_*x*_0_REG[15:8]
			2	AES_TEXT_*x*_3_REG[23:16]	AES_TEXT_*x*_2_REG[23:16]	AES_TEXT_*x*_1_REG[23:16]	AES_TEXT_*x*_0_REG[23:16]
			3	AES_TEXT_*x*_3_REG[31:24]	AES_TEXT_*x*_2_REG[31:24]	AES_TEXT_*x*_1_REG[31:24]	AES_TEXT_*x*_0_REG[31:24]
1	0	State		c			
				0	1	2	3
		r	0	AES_TEXT_*x*_0_REG[31:24]	AES_TEXT_*x*_1_REG[31:24]	AES_TEXT_*x*_2_REG[31:24]	AES_TEXT_*x*_3_REG[31:24]
			1	AES_TEXT_*x*_0_REG[23:16]	AES_TEXT_*x*_1_REG[23:16]	AES_TEXT_*x*_2_REG[23:16]	AES_TEXT_*x*_3_REG[23:16]
			2	AES_TEXT_*x*_0_REG[15:8]	AES_TEXT_*x*_1_REG[15:8]	AES_TEXT_*x*_2_REG[15:8]	AES_TEXT_*x*_3_REG[15:8]
			3	AES_TEXT_*x*_0_REG[7:0]	AES_TEXT_*x*_1_REG[7:0]	AES_TEXT_*x*_2_REG[7:0]	AES_TEXT_*x*_3_REG[7:0]
1	1	State		c			
				0	1	2	3
		r	0	AES_TEXT_*x*_0_REG[7:0]	AES_TEXT_*x*_1_REG[7:0]	AES_TEXT_*x*_2_REG[7:0]	AES_TEXT_*x*_3_REG[7:0]
			1	AES_TEXT_*x*_0_REG[15:8]	AES_TEXT_*x*_1_REG[15:8]	AES_TEXT_*x*_2_REG[15:8]	AES_TEXT_*x*_3_REG[15:8]
			2	AES_TEXT_*x*_0_REG[23:16]	AES_TEXT_*x*_1_REG[23:16]	AES_TEXT_*x*_2_REG[23:16]	AES_TEXT_*x*_3_REG[23:16]
			3	AES_TEXT_*x*_0_REG[31:24]	AES_TEXT_*x*_1_REG[31:24]	AES_TEXT_*x*_2_REG[31:24]	AES_TEXT_*x*_3_REG[31:24]

说明：

1. 有关 “State” 的详细定义，请参考 NIST FIPS 197 中 “3.4 The State” 章节。
2. 其中，
   - *x* = IN 时，AES_TEXT_IN_*m*_REG 的 Word Endian 和 Byte Endian 控制位分别为 AES_ENDIAN_REG 的 Bit 2 和 Bit 3；
   - *x* = OUT 时，AES_TEXT_OUT_*m*_REG 的 Word Endian 和 Byte Endian 控制位分别为 AES_ENDIAN_REG 的 Bit 4 和 Bit 5。

密钥字节序

在 Typical AES 工作模式下，AES 加速器的密钥字节序由寄存器 AES_ENDIAN_REG 的 Bit 0 和 Bit 1 控制，共可组成四种密钥字节序。

表 88、表 89、表 90 描述了在四种密钥字节序下，寄存器 AES_KEY_*n*_REG 中的每个 word 将如何构成 “the first Nk words of the expanded key”。

表 88: AES-128 密钥字节序

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

说明：

1. 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。
2. Bit 列代表 w[0] ~ w[3] 每个 word 中的各个字节。

表 89: AES-192 密钥字节序

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

说明：

1. w[0] ~ w[5] 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。
2. Bit 列代表 w[0] ~ w[5] 每个 word 中的各个字节。

表 90: AES-256 密钥字节序

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

说明：

1. 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。
2. Bit 列代表 w[0] ~ w[7] 每个 word 中的各个字节。

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、AES_ENDIAN_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_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、AES_ENDIAN_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、AES_KEY_*n*_REG、AES_ENDIAN_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 种块模式运算和 GCM 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 91。

表 91: 运算模式选择

AES_BLOCK_MODE_REG[2:0]	运算模式
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	GCM (Galois/Counter Mode)

AES 加速器的状态值可查看寄存器 AES_STATE_REG，具体见表 92 所示：

表 92: 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。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 (in_stream) 来自 DMA，结果数据 (out_stream) 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 93 所示。

表 93: 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 equals 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 94 所示。结果数据也遵从相同的存放规则，在此不多做介绍。

表 94: 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。

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 94 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 块运算模式的流程

1. 对寄存器 CRYPTO_DMA_AES_SHA_SELECT_REG 写入 0。
2. 配置 Crypto DMA 链表并启动 DMA。
3. 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。
   - 初始化寄存器 AES_MODE_REG、AES_KEY_*n*_REG、AES_ENDIAN_REG。
   - 初始化寄存器 AES_BLOCK_MODE_REG，请参照表 91。
   - 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）。
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 2。如果开启了中断功能，也可以等待 AES_INT 中断产生。
6. 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。
7. 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。
8. 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 5 之后。

19.5.7 GCM 运算模式的流程

1. 对寄存器 CRYPTO_DMA_AES_SHA_SELECT_REG 写入 0。
2. 配置 Crypto DMA 链表并启动 DMA。

3. 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。
   - 初始化寄存器 AES_MODE_REG、AES_KEY_*n*_REG、AES_ENDIAN_REG。
   - 初始化寄存器 AES_BLOCK_MODE_REG 为 6。
   - 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_AAD_BLOCK_NUM_REG，请参照章节 19.6.4。
   - 初始化寄存器 AES_REMAINDER_BIT_NUM_REG，请参照章节 19.6.5。解密时不使用。
4. 启动运算。对寄存器 AES_TRIGGER_REG 写入 1。
5. 轮询寄存器 AES_STATE_REG，直到读到 0。请参照表 92。此处无中断产生。
6. 读存储器 AES_H_MEM 获取 H。
7. 软件计算 J_0，然后将其写入存储器 AES_J0_MEM。
8. 继续运算。对寄存器 AES_CONTINUE_REG 写入 1。
9. 等待运算完成。轮询寄存器 AES_STATE_REG，直到读到 2。请参照表 92。如果开启了中断功能，也可以等待 AES_INT 中断产生。
10. 此时，T_0 已经准备好。读存储器 AES_T0_MEM 获取 T_0。
11. 确认 DMA 完成数据传输。此时，结果数据已经被 DMA 写入 memory，可以直接从中读取。
12. 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。
13. 对寄存器 AES_DMA_EXIT_REG 写入 1。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 9 之后。

19.6 GCM 算法

ESP32-S2 中 AES 加速器直接支持 GCM 算法（更多信息，请见 NIST SP 800-38D 标准）。值得注意的是，考虑到实际使用中很少会使用长度超过 $2^{32}-1$ 比特的 AAD 、C 和 P，我们在算法中将 AAD 、C 和 P 的长度限制在 $2^{32}-1$ 比特之内。图 19-12 显示了 ESP32-S2 内置 AES 加速器实现 GCM 加密算法的流程。

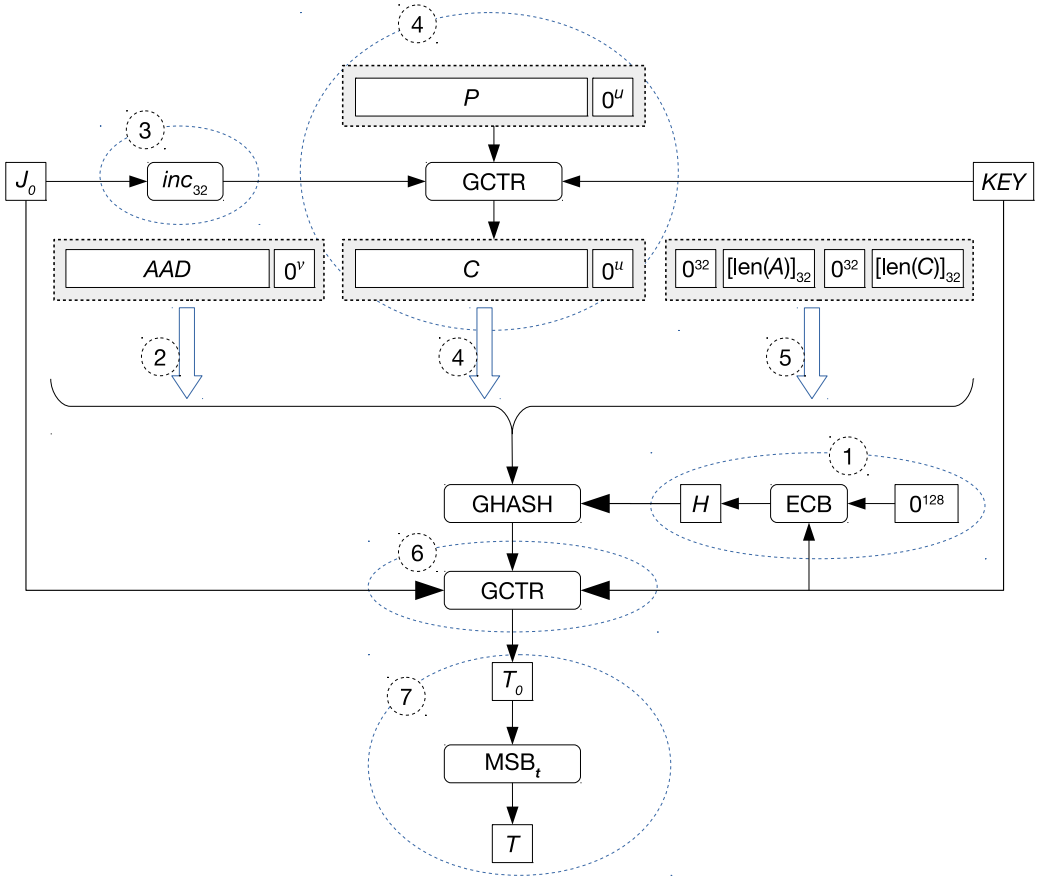


图 19-12. GCM 加密操作流程图

具体步骤描述见下：

1. 硬件调用 ECB 算法求解哈希子密钥 (H) 的值，为哈希计算做准备；
2. 硬件调用 GHASH 算法对封装后的 AAD 作哈希计算；
3. 硬件调用标准增量函数 INC_{32} 对 J_0 作运算，为 CTR 加密做准备；
4. 硬件调用 GCTR 算法对封装后的明文 P 做加密运算，同时调用 GHASH 算法对封装后的密文 C 作哈希计算；
5. 硬件调用 GHASH 算法对追加块做哈希计算，获得最终的 128 位哈希结果；
6. 硬件调用 GCTR 算法对 J_0 做加密运算，获得 T_0 ；
7. 软件读取硬件结果 T_0，并调用 MSB_t 算法计算最终的认证标签 T 。

GCM 解密运算与 GCM 加密运算基本相同，仅有一处差别：在图 19-12 中的步骤 4 中，使用 GCTR 算法对封装后的密文做解密运算，得出明文。此处不再赘述。

19.6.1 哈希子密钥（Hash subkey）

在 GCM 操作中，哈希子密钥 (H) 是一个 128-bit 的值，由硬件计算求出，即图 19-12 中的步骤 1 ，对应 NIST SP 800-38D 标准中 “7 GCM Specification” 中的 “Step 1. Let H = CIPH$_K(0^{128})$ ” 描述。

哈希子密钥 (H) 的值以字节序的形式存放在存储器 AES_H_MEM 中，且需遵循表 94 中的字节序规，即最左侧字节存放在 AES_H_MEM 中的最低地址，最右侧字节存放在 AES_H_MEM 中的最高地址。

19.6.2 J_0

在 GCM 操作中，J_0 是一个 128-bit 的值，由软件计算求出，将参与到图 19-12 中的步骤 3 和步骤 6 的计算中。具体的计算方式在 NIST SP 800-38D 标准中的 “7 GCM Specification” 章节部分有明确定义。请参阅 NIST SP 800-38D 获取更多算法标准的相关信息。

J_0 值以字节序的形式存放在存储器 AES_J0_MEM 中，且需遵循表 94 中的字节序规，即最左侧字节存放在 AES_J0_MEM 中的最低地址，最右侧字节存放在 AES_J0_MEM 中的最高地址。

19.6.3 认证标签（Authenticated Tag）

认证标签（Authenticated Tag，简称为 Tag）是 GCM 计算的最终结果之一，由软件完成计算，对应图 19-12 中的步骤 7 ，具体取值由长度参数 t ($1 <= t <= 128$) 决定：

- 当 $t = 128$ 时，Tag 的值记为 T_0。T_0 是一个 128-bit 的比特串，以字节序的形式存放在存储器 AES_T0_MEM 中，且需遵循表 94 中的字节序规，即最左侧字节存放在 AES_T0_MEM 中的最低地址，最右侧字节存放在 AES_T0_MEM 中的最高地址。
- 当 $1 <= t < 128$ 时，Tag 的值为 T_0 中最左侧的 t 个比特位，可以用 MSB$_t(T_0)$ 函数表示。例如，MSB$_4$(111011010) = 1110，MSB$_5$(11010011010) = 11010。有关 MSB$_t$( ) 函数的具体定义，请见 NIST SP 800-38D 的 “6 Mathematical Components of GCM” 章节。

19.6.4 附加认证消息块个数（AAD Block Number ）

寄存器 AES_AAD_BLOCK_NUM_REG 存放附加认证消息（Additional Authenticated Data, AAD）的块个数，其值等于 length(**TEXT-PADDING**(AAD))/128。该寄存器仅在 DMA-AES 工作模式下的 GCM 操作中有意义。

19.6.5 不完整块的有效比特数 (Remainder Bit Number)

寄存器 AES_REMAINDER_BIT_NUM_REG 存放明文中不完整块的有效比特数，其值等于 length(P)%128。该寄存器仅在 DMA-AES 工作模式下的 GCM 加密操作中有意义。寄存器 AES_REMAINDER_BIT_NUM_REG 的值不会影响密文或明文结果，但会影响 Tag 结果值（包含在 T_0 中）。依据 GCM 算法标准，GCM 运算是 GCTR 运算和 GHASH 运算的结合，GCTR 运算用于执行加解密计算，GHASH 运算用于求解 Tag。

- 对于 GCM 加密，硬件首先计算 C，而后将其封装为 **TEXT-PADDING**(C)，作为 GHASH 运算的输入。此时，硬件需要根据寄存器 AES_REMAINDER_BIT_NUM_REG 的值决定追加 0 的个数。
- 对于 GCM 解密，封装操作 **TEXT-PADDING**(C) 交由软件完成，此时寄存器 AES_REMAINDER_BIT_NUM_REG 的值不起作用。

19.7 基地址

用户可以通过两个不同的寄存器基地址访问 AES，如表 95 所示。更多信息，请前往章节 1 系统和存储器。

表 95: AES 基地址

访问总线	基地址

PeriBUS1	0x3F43A000
PeriBUS2	0x6003A000

19.8 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 AES 基地址的地址偏移量（相对地址）。请参阅章节 19.7 获取有关 AES 基地址的信息。

名称	描述	大小	起始地址	结束地址	访问
AES_IV_MEM	存储器 IV	16 字节	0x0050	0x005F	读 / 写
AES_H_MEM	存储器 H	16 字节	0x0060	0x006F	只读
AES_J0_MEM	存储器 J0	16 字节	0x0070	0x007F	读 / 写
AES_T0_MEM	存储器 T0	16 字节	0x0080	0x008F	只读

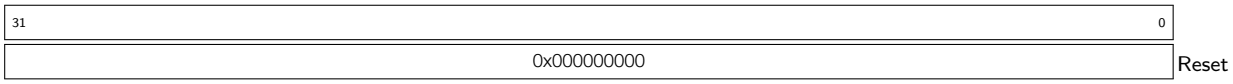
19.9 寄存器列表

请注意，这里的地址是相对于 AES 基地址的地址偏移量（相对地址）。请参阅章节 19.7 获取有关 AES 基地址的信息。

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	工作模式选择寄存器	0x0040	读 / 写
AES_ENDIAN_REG	字节序配置寄存器	0x0044	读 / 写
AES_DMA_ENABLE_REG	DMA 使能寄存器	0x0090	读 / 写
AES_BLOCK_MODE_REG	块模式配置寄存器	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
AES_AAD_BLOCK_NUM_REG	AAD 块数量配置寄存器	0x00A0	读 / 写
AES_REMAINDER_BIT_NUM_REG	明文或密文的不完整块的有效比特位数	0x00A4	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_CONTINUE_REG	继续运算寄存器	0x00A8	只写
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

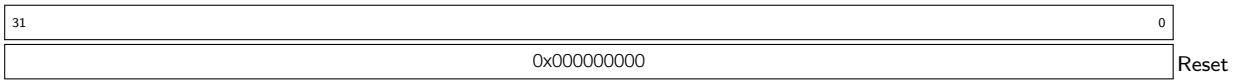
19.10 寄存器

Register 19.1: AES_KEY_*n*_REG (*n*: 0-7) (0x0000+4n*)**



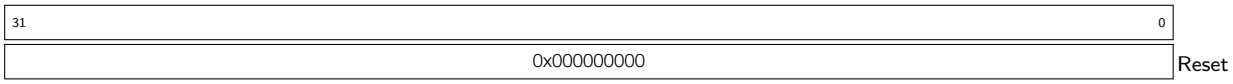
AES_KEY_*n*_REG (*n*: 0-7) AES 密钥寄存器。（读 / 写）

Register 19.2: AES_TEXT_IN_*m*_REG (*m*: 0-3) (0x0020+4m*)**



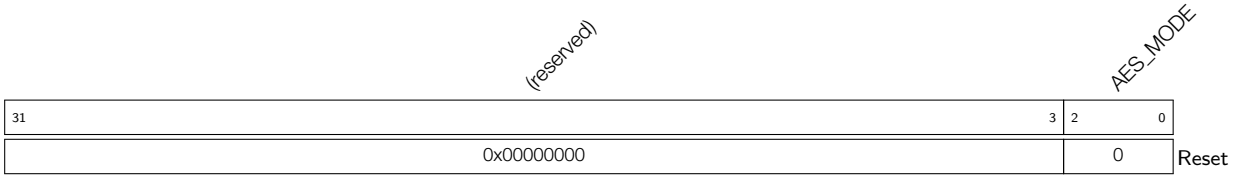
AES_TEXT_IN_*m*_REG (*m*: 0-3) Typical AES 文本输入寄存器。（读 / 写）

Register 19.3: AES_TEXT_OUT_*m*_REG (*m*: 0-3) (0x0030+4m*)**



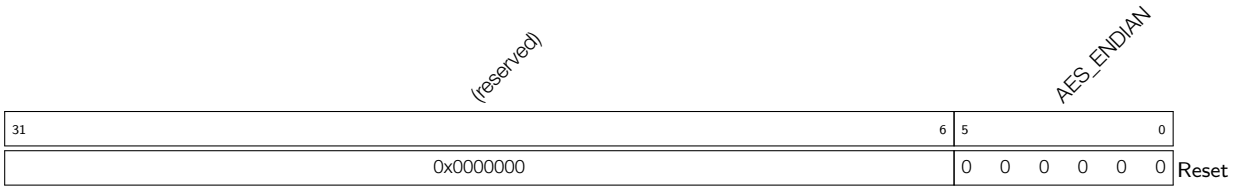
AES_TEXT_OUT_*m*_REG (*m*: 0-3) Typical AES 文本输出寄存器。（只读）

Register 19.4: AES_MODE_REG (0x0040)



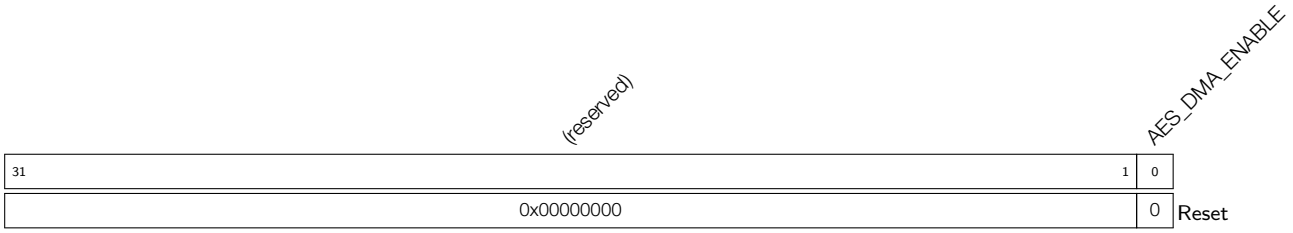
AES_MODE 选择 AES 加速器的工作模式。详情请见表 85。（读 / 写）

Register 19.5: AES_ENDIAN_REG (0x0044)



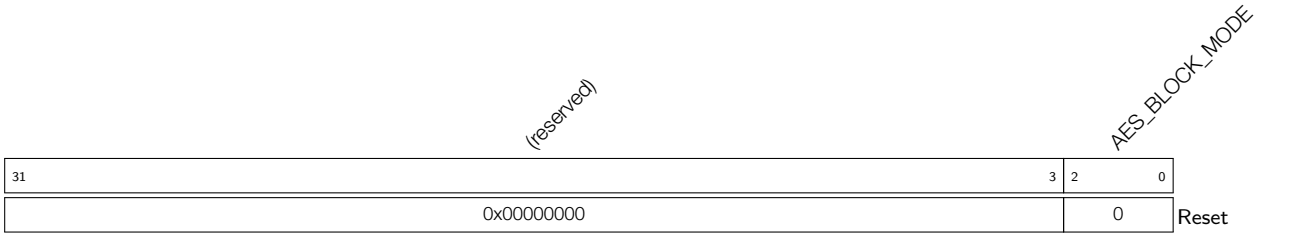
AES_ENDIAN 字节序选择寄存器。详情请见表 87。（读 / 写）

Register 19.6: AES_DMA_ENABLE_REG (0x0090)



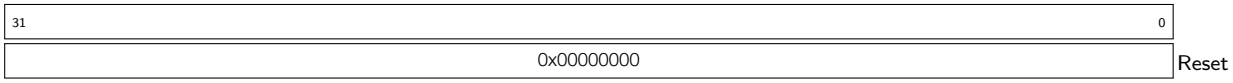
AES_DMA_ENABLE 选择工作模式。详情请见表 84。（读 / 写）

Register 19.7: AES_BLOCK_MODE_REG (0x0094)



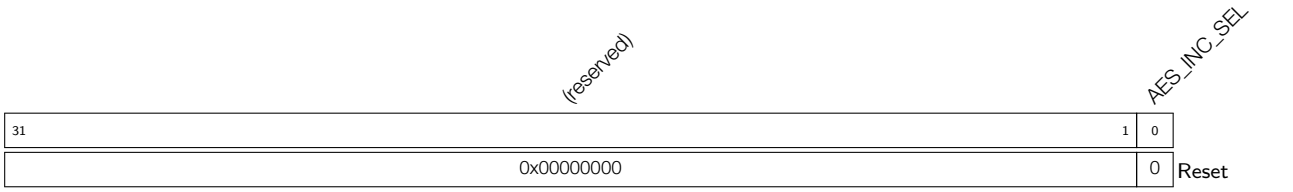
AES_BLOCK_MODE 选择 DMA-AES 使用的块模式。详情请见表 91。（读 / 写）

Register 19.8: AES_BLOCK_NUM_REG (0x0098)



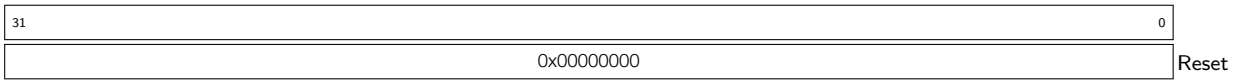
AES_BLOCK_NUM 在 DMA-AES 运算中待加解密的文本块数。详情请见章节 19.5.4。（读 / 写）

Register 19.9: AES_INC_SEL_REG (0x009C)



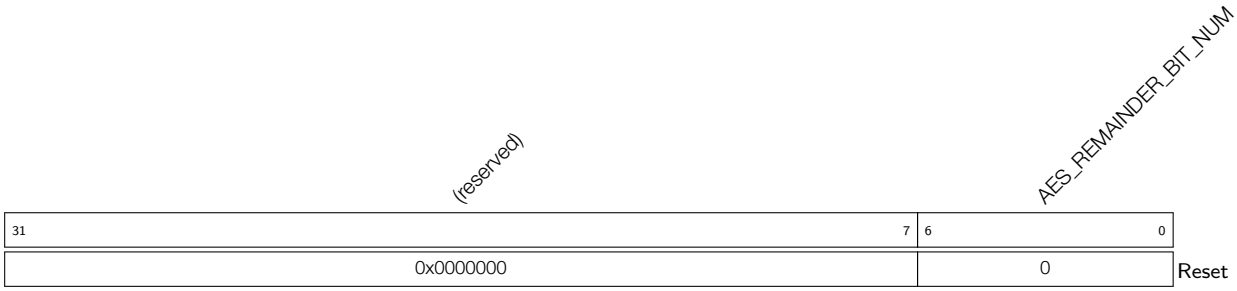
AES_INC_SEL 选择 CTR 块模式使用的标准增量函数。置 0 选择 INC_{32} 标准增量函数，置 1 选择 INC_{128} 标准增量函数。（读 / 写）

Register 19.10: AES_AAD_BLOCK_NUM_REG (0x00A0)



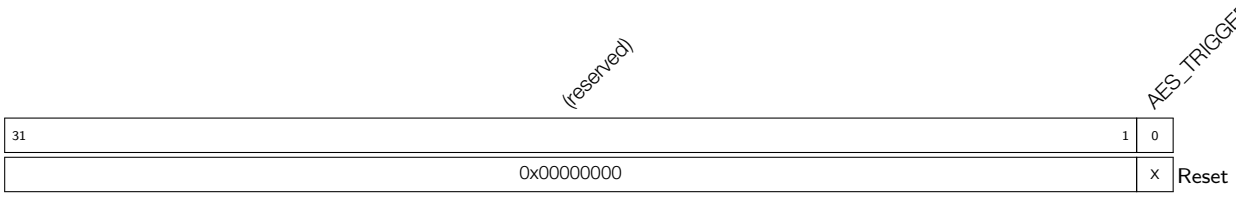
AES_AAD_BLOCK_NUM 在 GCM 运算中 AAD 的块数。详情请见章节 19.6.4。(读 / 写)

Register 19.11: AES_REMAINDER_BIT_NUM_REG (0x00A4)



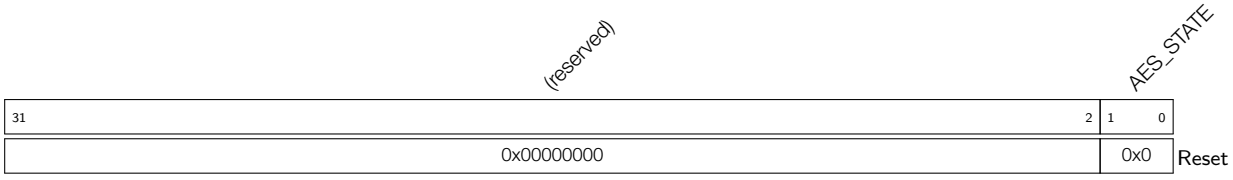
AES_REMAINDER_BIT_NUM 在 GCM 运算中输入文本的不完整块的有效比特数。详情请见章节 19.6.5。(读 / 写)

Register 19.12: AES_TRIGGER_REG (0x0048)



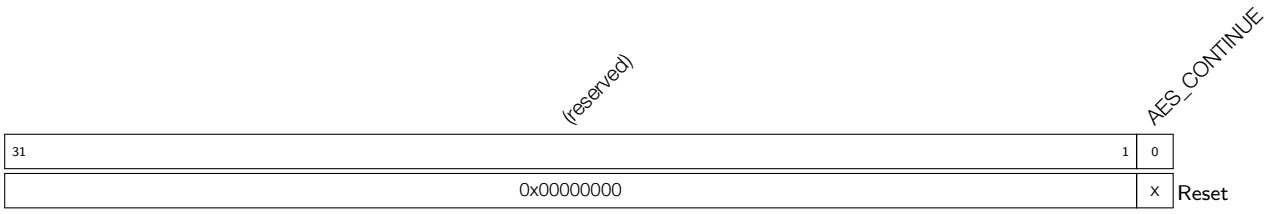
AES_TRIGGER 写入 1 使能 AES 运算。(只写)

Register 19.13: AES_STATE_REG (0x004C)



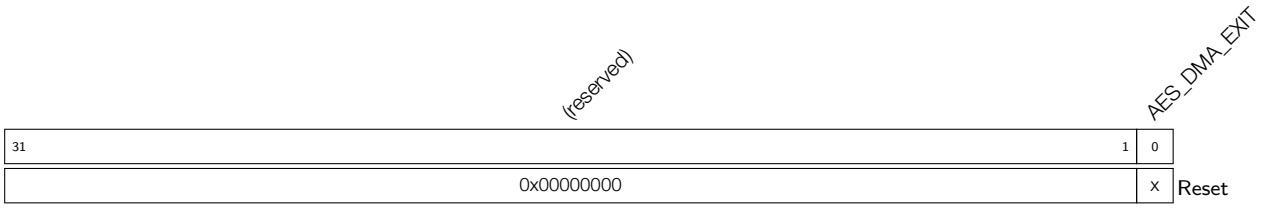
AES_STATE AES 状态寄存器。详见表 86 (Typical AES 工作模式) 和表 92 (DMA-AES 工作模式)。(只读)

Register 19.14: AES_CONTINUE_REG (0x00A8)



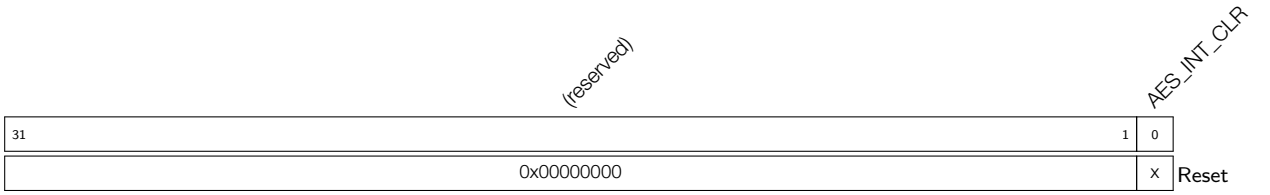
AES_CONTINUE 在 GCM 运算中，写入 1 继续运算。(只写)

Register 19.15: AES_DMA_EXIT_REG (0x00B8)



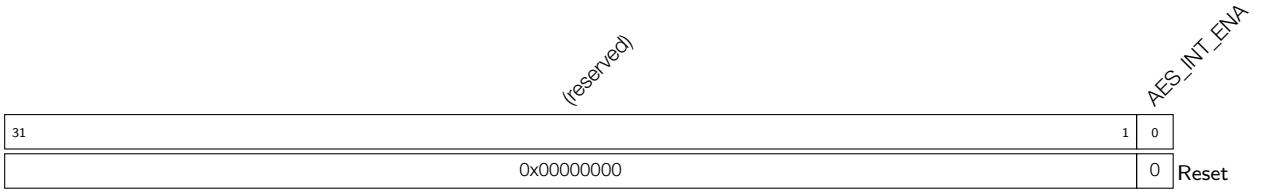
AES_DMA_EXIT 在 DMA-AES 运算完成后，在下一次配置 AES 任何寄存器之前，写入 1 使 AES 回到空闲状态。(只写)

Register 19.16: AES_INT_CLR_REG (0x00AC)



AES_INT_CLR 写入 1 清除 AES 中断。(只写)

Register 19.17: 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。(读 / 写)

20. SHA 加速器

20.1 概述

ESP32-S2 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。

20.2 主要特性

ESP32-S2 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 (interleave) 功能（仅限 Typical SHA 工作模式）
- 允许中断功能（仅限 DMA-SHA 工作模式）

20.3 工作模式简介

ESP32-S2 内置的 SHA 加速器支持 Typical SHA 和 DMA-SHA 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 crypto 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 98。

表 98: 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 99。

表 99: 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-S2 的数字签名和 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。

20.4 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。

20.4.1 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。

20.4.1.1 附加填充比特

SHA 加速器仅能处理长度为 512 位及其倍数或 1024 位及其倍数的信息。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。

20.4.1.2 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 $M_{15}^{(i)}$。

在 Typical SHA 工作模式下，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。

在 DMA-SHA 工作模式下，需要预先完成如下配置：

1. 创建外发链表；
2. 根据章节 *DMA Controller* 完成链表配置，包括但不限于将字节数组（完成填充后的信息）的首地址配置给发送链表的 buffer 地址指针；
3. 配置寄存器 CRYPTO_DMA_OUTLINK_ADDR 指向第一个发送链表；
4. 将数值 1 写入寄存器 CRYPTO_DMA_OUTLINK_START，启动 DMA 模块开始搬运数据；
5. 将数值 1 写入寄存器 CRYPTO_DMA_AES_SHA_SELECT_REG，将 AES 和 SHA 共用的 DMA 资源分配给 SHA 加速器使用。

20.4.1.3 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：

$8'h3t_0$	$1'b1$	$23'b0$

     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：

$8'h3t_1$	$8'h3t_0$	$1'b1$	$15'b0$

     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：

$8'h3t_2$	$8'h3t_1$	$8'h3t_0$	$1'b1$	$7'b0$

     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”。

20.4.2 哈希运算流程

在完成信息预处理后，ESP32-S2 SHA 加速器将正式开始哈希运算，并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-S2 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。

20.4.2.1 Typical SHA 模式下的运算流程

ESP32-S2 SHA 加速器在 Typical SHA 工作模式下支持两种运算方法：

- **“alone” 运算方法**：在计算完所有信息块并得到全部信息摘要前，用户不会插入其他运算任务。
- **“interleave” 运算方法**：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 99。
2. 处理当前信息块。
   (a) 将当前信息块写入 SHA_M_*n*_REG 寄存器堆；
   (b) 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，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。

   (c) 轮询寄存器 SHA_BUSY_REG 一直到读回的值为 0，代表 SHA 硬件加速器已完成对当前信息块的计算，进入“空闲”状态。然后执行步骤 3。
3. 选择是否要插入其他运算。
   - 如需插入，则准备移交 SHA 加速器使用权：
     (a) 读取并保存寄存器 SHA_MODE_REG 中的运算标准类型；
     (b) 读取并保存寄存器堆 SHA_H_*n*_REG 中的信息摘要；
     (c) 最后，跳转至相应流程进行插入运算。具体按照插入运行类型的不同，请见 Typical SHA 或 DMA-SHA 章节。
   - 如无需插入，则继续执行步骤 4。
4. 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，进入步骤 5。
5. 判断是否需要交还 SHA 加速器使用权（即判断本次运算是否为插入运算）：

- 如果是插入运算，则应交还 SHA 加速器的使用权：
  (a) 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;
  (b) 将获得使用权前保存的信息摘要写入寄存器堆 SHA_H_*n*_REG；
  (c) 然后执行步骤 2。
- 如果不是插入运算，则无需交还 SHA 加速器的使用权。此时，请执行步骤 6。

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 20.4.1.3 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。
   (a) 将当前信息块写入 SHA_M_*n*_REG 寄存器堆；
   (b) 启动 SHA 加速器 [1]：
   - 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。

   (c) 轮询寄存器 SHA_BUSY_REG 一直到读回的值为 0，代表 SHA 硬件加速器已完成对当前信息块的计算，进入“空闲”状态。然后执行步骤 4。
4. 选择是否要插入其他运算。
   - 如需插入，则准备移交 SHA 加速器使用权：
     (a) 读取并保存寄存器 SHA_MODE_REG 中的运算标准类型；
     (b) 读取并保存寄存器堆 SHA_H_*n*_REG 中的信息摘要；
     (c) 最后，跳转至相应流程进行插入运算。具体按照插入运行类型的不同，请见 Typical SHA 或 DMA-SHA 章节。
   - 如无需插入，则继续执行步骤 5。
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 3。
   - 否则，进入步骤 6。

6. 判断是否需要交还 SHA 加速器使用权（即判断本次运算是否为插入运算）：
    - 如果是插入运算，则应交还 SHA 加速器的使用权：
        (a) 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；
        (b) 将获得使用权前保存的信息摘要写入寄存器堆 SHA_H_*n*_REG；
        (c) 然后执行步骤 3。
    - 如果不是插入运算，则无需交还 SHA 加速器的使用权。此时，请执行步骤 7。
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. 在第 2b 步时，在 SHA 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 寄存器堆，以节省时间。

20.4.2.2 DMA-SHA 模式下的运算流程

ESP32-S2 SHA 加速器在 DMA-SHA 工作模式下不支持“interleave”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。用户先按照 20.4.1.2 章节配置 DMA 控制器。

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 99。
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 20.4.1.3 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。

20.4.3 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 103：

表 103: 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

说明：

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。

20.4.4 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。

20.5 基地址

用户可以通过两个不同的寄存器基地址访问 SHA，如表 104 所示。更多信息，请前往章节 1 *系统和存储器*。

表 104: SHA 基地址

基地址	地址值
PeriBUS1	0x3F43B000
PeriBUS2	0x6003B000

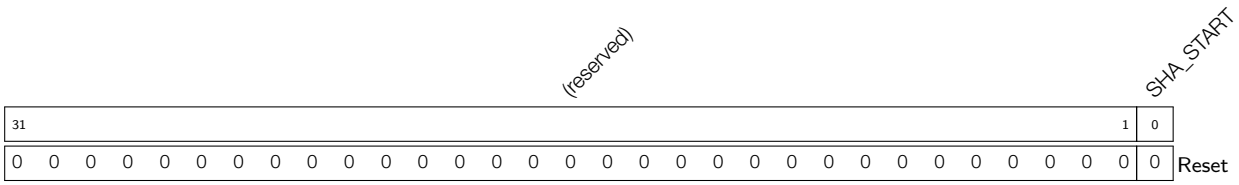
20.6 寄存器列表

名称	描述	地址	访问权限
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
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

名称	描述	地址	访问权限
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
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

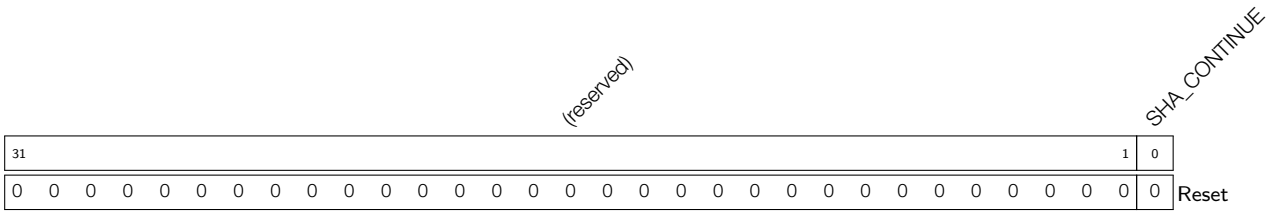
20.7 寄存器

Register 20.1: SHA_START_REG (0x0010)



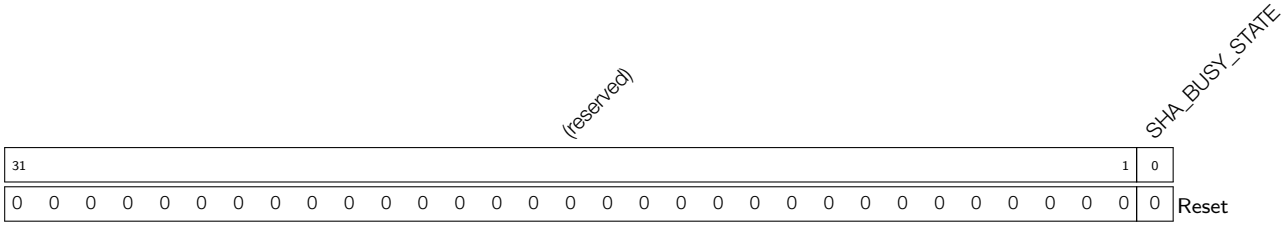
SHA_START 置 1 启动 SHA 加速器的 Typical SHA 模式。(只写)

Register 20.2: SHA_CONTINUE_REG (0x0014)



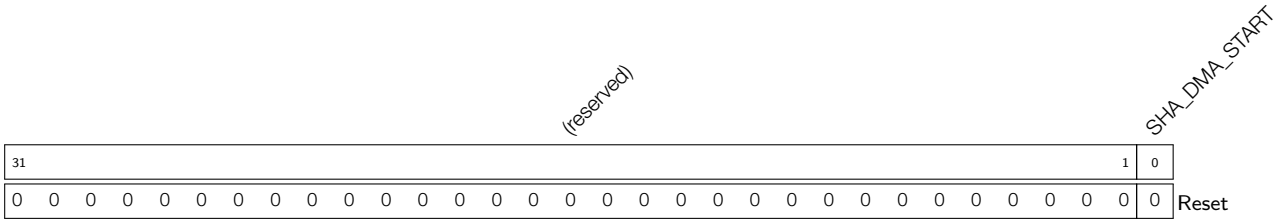
SHA_CONTINUE 置 1 继续 SHA 加速器的 Typical SHA 运算。（只写）

Register 20.3: SHA_BUSY_REG (0x0018)



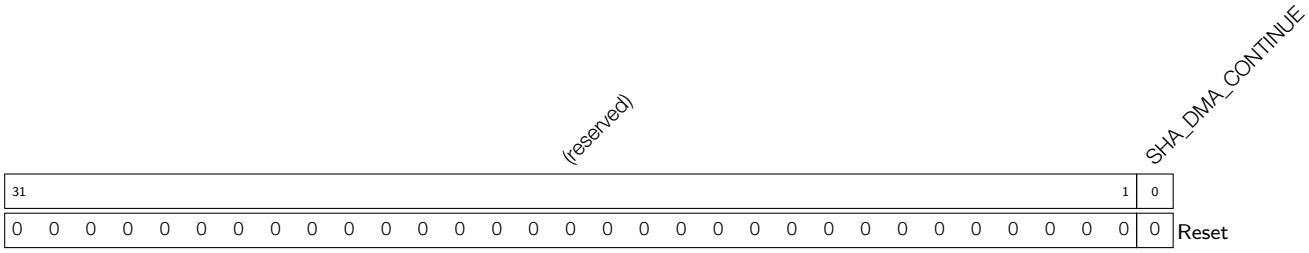
SHA_BUSY_STATE 指示 SHA 是否处于“忙碌”状态。（只读）1'h0: 空闲 1'h1: 忙碌

Register 20.4: SHA_DMA_START_REG (0x001C)



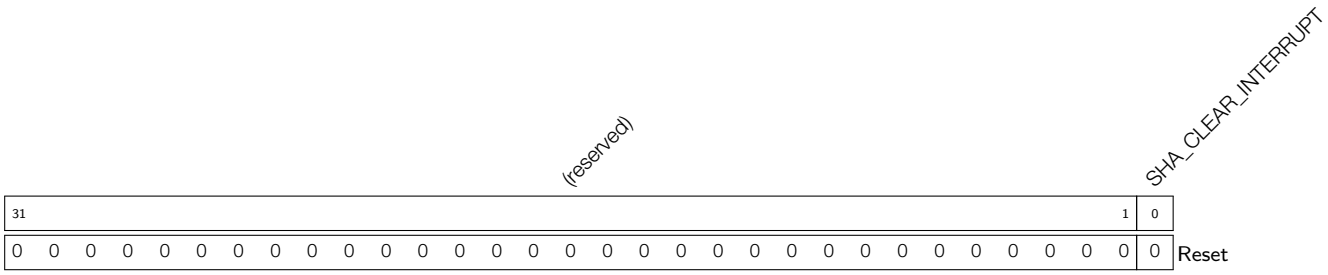
SHA_DMA_START 置 1 启动 SHA 加速器的 DMA-SHA 模式。（只写）

Register 20.5: SHA_DMA_CONTINUE_REG (0x0020)



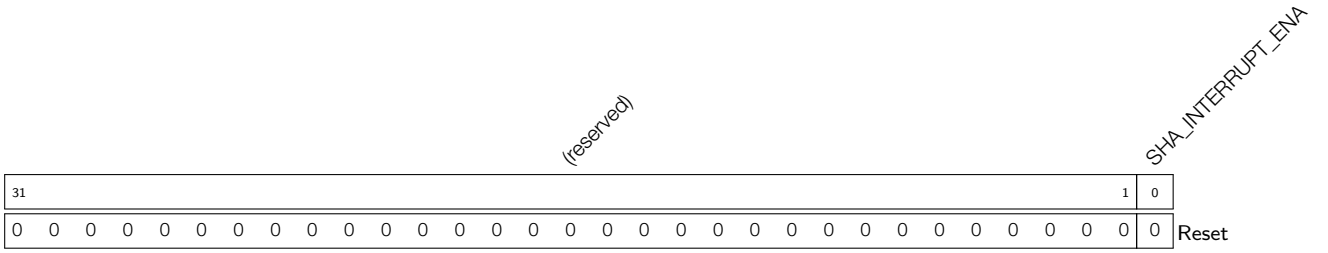
SHA_DMA_CONTINUE 置 1 继续 SHA 加速器的 DMA-SHA 运算。（只写）

Register 20.6: SHA_INT_CLEAR_REG (0x0024)



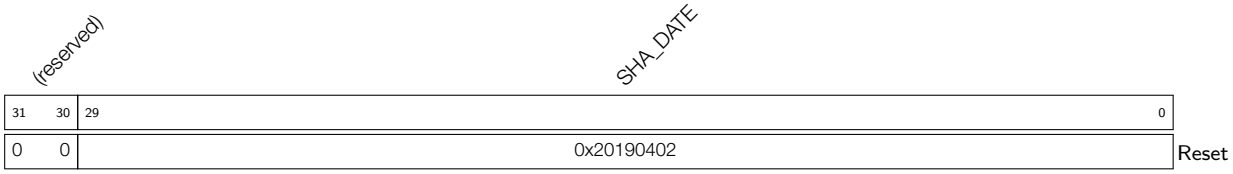
SHA_CLEAR_INTERRUPT 清除 DMA-SHA 中断。(只写)

Register 20.7: SHA_INT_ENA_REG (0x0028)



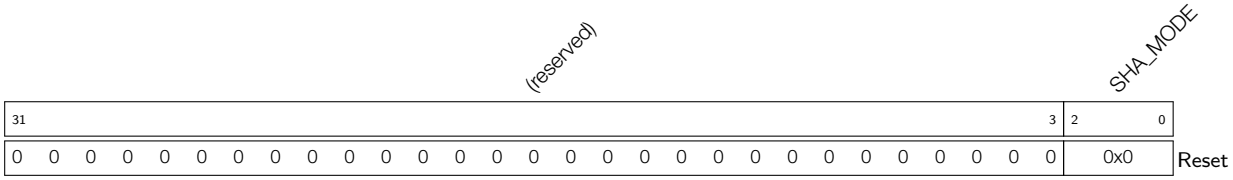
SHA_INTERRUPT_ENA 使能 DMA-SHA 中断。(读写)

Register 20.8: SHA_DATE_REG (0x002C)



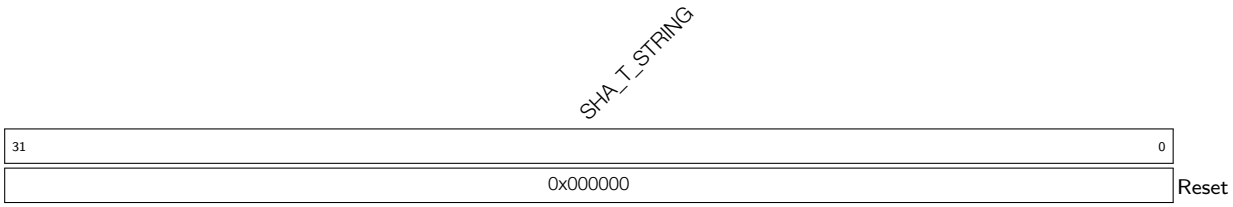
SHA_DATE 版本控制寄存器。(读写)

Register 20.9: SHA_MODE_REG (0x0000)



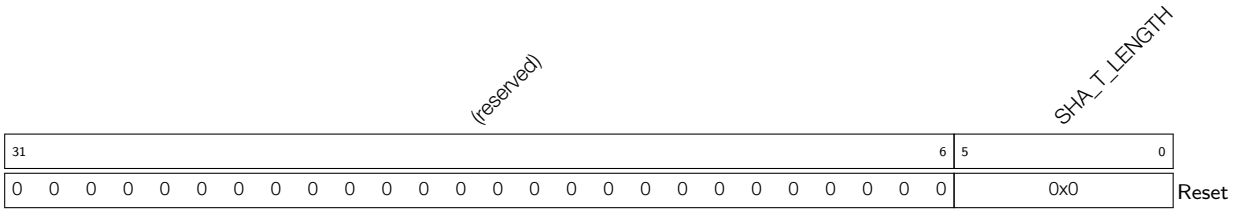
SHA_MODE 选择 SHA 加速器的运算标准，详见表 99。(R/W)

Register 20.10: SHA_T_STRING_REG (0x0004)



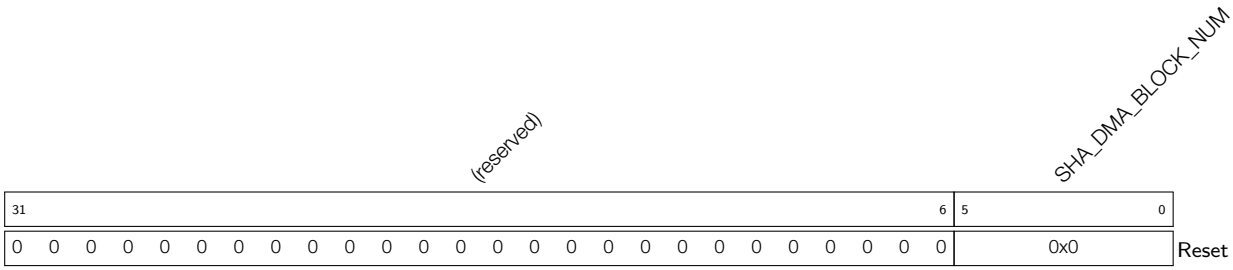
SHA_T_STRING 存储哈希字符串内容（仅用于计算 SHA-512/t 的哈希初始值）。（读写）

Register 20.11: SHA_T_LENGTH_REG (0x0008)



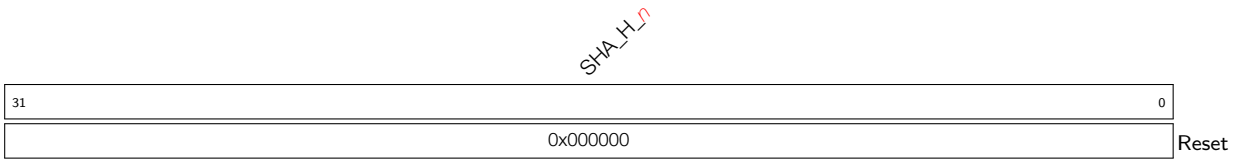
SHA_T_LENGTH 存储哈希字符串长度（仅用于计算 SHA-512/t 的哈希初始值）。（读写）

Register 20.12: SHA_DMA_BLOCK_NUM_REG (0x000C)



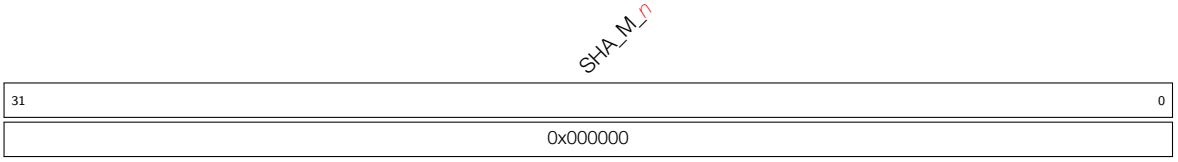
SHA_DMA_BLOCK_NUM 定义 DMA-SHA 工作模式下的信息块个数。（读写）

Register 20.13: SHA_H_*n*_REG (*n*: 0-15) (0x0040+4n*)**



SHA_H_*n* 存储第 *n* 个 32 位哈希值。（读写）

Register 20.14: SHA_M_*n*_REG (*n*: 0-31) (0x0080+4n*)**



SHA_M_*n* 存储第 *n* 个 32 位输入信息。（读写）

21. RSA 加速器

21.1 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。

21.2 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

21.3 功能描述

RSA 加速器的激活仅需使能 DPORT_CRYPTO_RSA_CLK_EN 外围时钟的 DPORT_PERIP_CLK_EN1_REG 位，并同时清零 DPORT_RSA_PD_CTRL_REG 寄存器中的 DPORT_RSA_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-S2 的数字签名模块也会调用 RSA 加速器。此时，用户无法正常访问 RSA 加速器。

21.3.1 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 21.3.4 获取详细信息。
3. 将 X_i、Y_i、M_i、$\overline{r}_i (i \in \{0, 1, \ldots, n\})$ 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 $Z_i (i \in \{0, 1, \ldots, n\})$。
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 $\bar{r}_i$ 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。

21.3.2 大数模乘运算

大数模乘运算也是基于 Montgomery Multiplication 实现运算 $Z = X \times Y \bmod M$，所以也需要预先通过软件计算得到 $\bar{r}$ 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、$\bar{r}_i$ $(i \in \{0, 1, \ldots, n\})$ 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 $(i \in \{0, 1, \ldots, n\})$。
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 $\bar{r}_i$ 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。

21.3.3 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长度为 $N = 32 \times x$ $(x \in \{1, 2, \ldots, 64\})$ 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 $(i \in \{0, 1, \ldots, n\})$ 分别写入存储器 RSA_X_MEM、RSA_Z_MEM。每块存储器的容量都是 128 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 ($i \in \{0, 1, \ldots, n\}$) 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 ($i \in \{0, 1, \ldots, n\}$) 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 ($i \in \{0, 1, \ldots, n\}$)。$\hat{n}$ 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。

21.3.4 加速选项

ESP32-S2 RSA 提供了两种加速选项，分别为 SEARCH 选项和 CONSTANT_TIME 选项，专用于加速大数模幂运算。这两个加速选项默认关闭，但可以同时使用。

当两个加速选项均关闭时，求解 $Z = X^Y \bmod M$ 的时间开销完全由运算子长度决定。然后，当任一加速选项开启时，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，
- $\widetilde{Y}_{N-1}, \widetilde{Y}_{N-2}, \ldots, \widetilde{Y}_{t+1}$ 的值均为 0，
- 且 $\widetilde{Y}_{t-1}, \widetilde{Y}_{t-2}, \ldots, \widetilde{Y}_0$ 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，当启动加速选项时：

- SEARCH 选项
  - 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，$\widetilde{Y}_{N-1}, \widetilde{Y}_{N-2}, \ldots, \widetilde{Y}_{t+1}$ 中的 0 位将在运算中全部被忽略。
- CONSTANT_TIME 选项
  - 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。

为了直观地展示加速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 106 展示了 4 种加速选项组合对应的时间开销。可以看到，相比于不开启任何加速选项，开启加速选项时的时间开销明显大幅度降低。注意，这里 SEARCH 选项开启时设定 α 为 16。

表 106: 加速效果

SEARCH 选项	CONSTANT_TIME 选项	时间开销	提速比例
关闭	关闭	376.405 ms	0%
开启	关闭	2.260 ms	99.41%
关闭	开启	1.203 ms	99.68%
开启	开启	1.165 ms	99.69%

21.4 基地址

用户可以通过两个不同的寄存器基地址访问 RSA，如表 107 所示。更多有关总线的信息，请见章节 1 *系统和存储器*。

表 107: RSA 基地址

外设访问方式	地址值
PeriBUS1	0x3F43C000
PeriBUS2	0x6003C000

21.5 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 RSA 基地址的地址偏移量（相对地址）。请参阅章节 21.4 获取有关 RSA 基地址的信息。

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

21.6 寄存器列表

请注意，这里的地址是相对于 RSA 基地址的地址偏移量（相对地址）。请参阅章节 21.4 获取有关 RSA 基地址的信息。

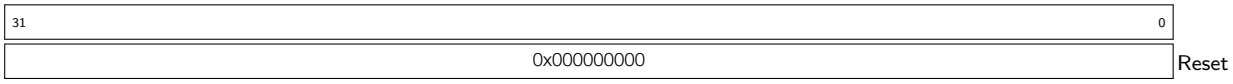
名称	描述	地址	访问
配置寄存器			
RSA_M_PRIME_REG	M' 存储器	0x0800	读 / 写
RSA_MODE_REG	RSA 长度模式	0x0804	读 / 写
RSA_CONSTANT_TIME_REG	固定时间选项	0x0820	读 / 写
RSA_SEARCH_ENABLE_REG	使能 search 加速选项	0x0824	读 / 写
RSA_SEARCH_POS_REG	search 起始位置	0x0828	读 / 写
状态/控制寄存器			
RSA_CLEAN_REG	RSA 清除寄存器	0x0808	只读

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

21.7 寄存器

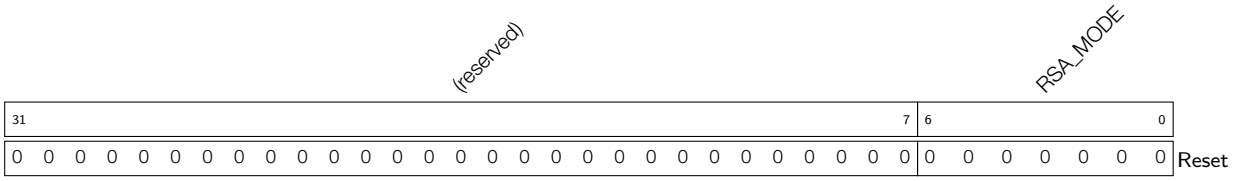
请注意，本章节的地址是相对于 RSA 基地址的地址偏移量（相对地址）。请参阅章节 21.4 获取有关 RSA 基地址的信息。

Register 21.1: RSA_M_PRIME_REG (0x0800)



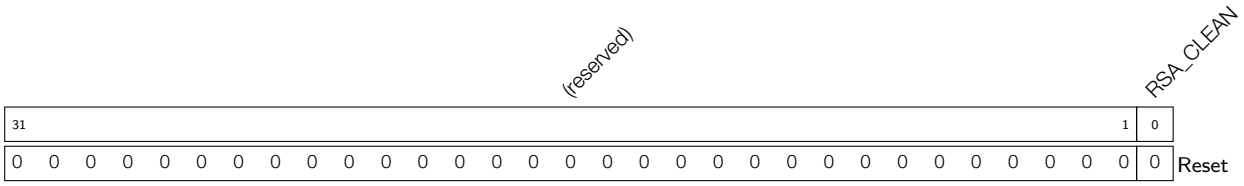
RSA_M_PRIME_REG 此寄存器存储 M'。(读 / 写)

Register 21.2: RSA_MODE_REG (0x0804)



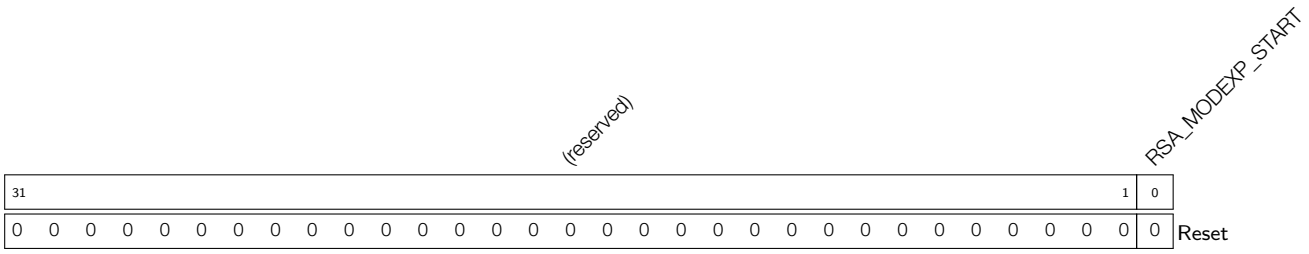
RSA_MODE 此寄存器存储模幂运算的模式。(读 / 写)

Register 21.3: RSA_CLEAN_REG (0x0808)



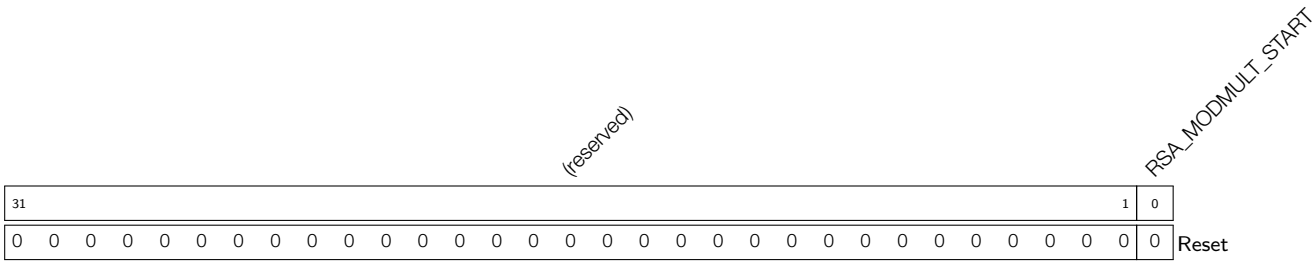
RSA_CLEAN 一旦存储器初始化结束，此位为 1。(只读)

Register 21.4: RSA_MODEXP_START_REG (0x080C)



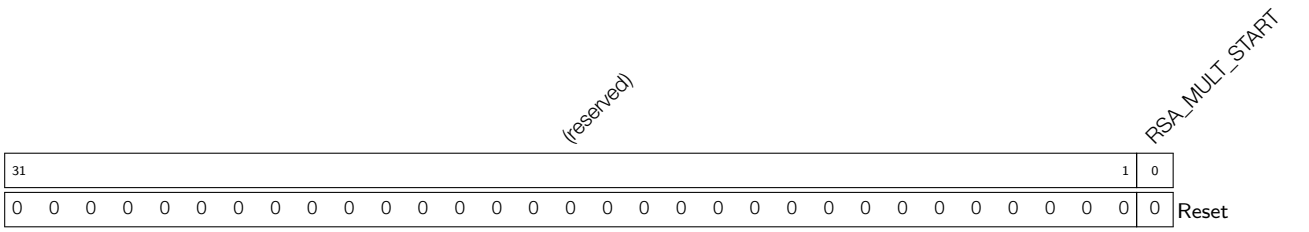
RSA_MODEXP_START 写入 1 以开始模幂运算。(只写)

Register 21.5: RSA_MODMULT_START_REG (0x0810)



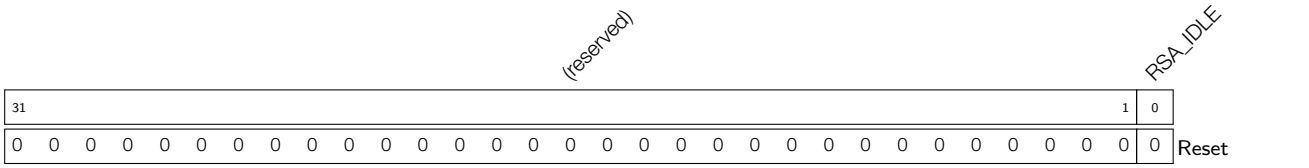
RSA_MODMULT_START 写入 1 以开始模乘运算。(只写)

Register 21.6: RSA_MULT_START_REG (0x0814)



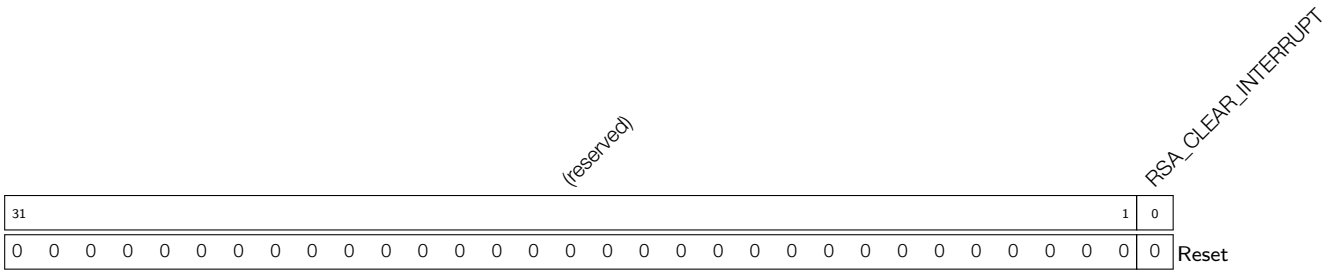
RSA_MULT_START 写入 1 以开始乘法运算。(只写)

Register 21.7: RSA_IDLE_REG (0x0818)



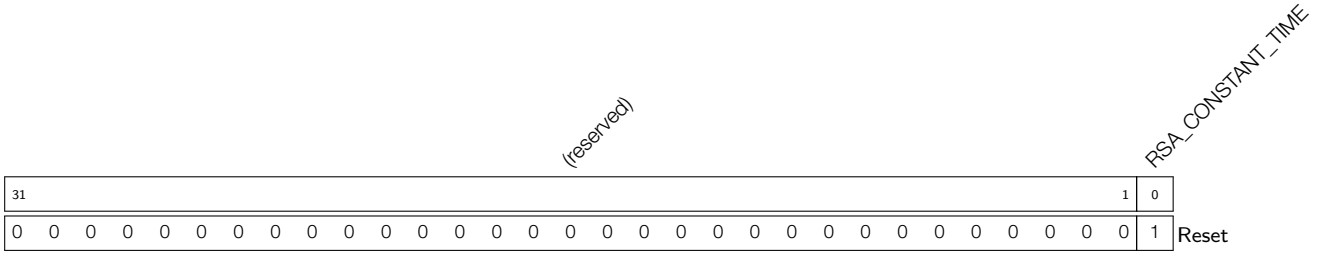
RSA_IDLE 当 RSA 空闲时，此位为 1。(只读)

Register 21.8: RSA_CLEAR_INTERRUPT_REG (0x081C)



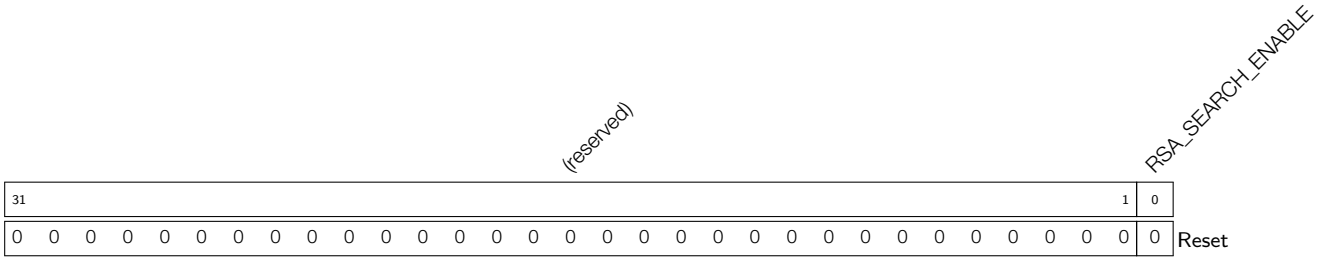
RSA_CLEAR_INTERRUPT RSA 中断清除寄存器。写入 1 清除中断。(只写)

Register 21.9: RSA_CONSTANT_TIME_REG (0x0820)



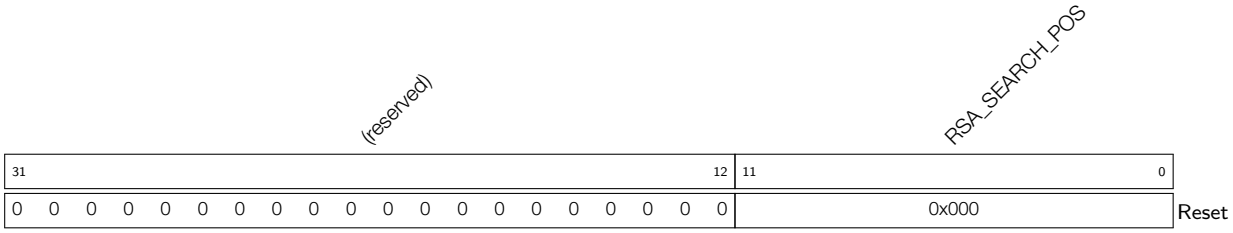
RSA_CONSTANT_TIME_REG 模幂运算中的 constant_time 加速选项。0: 开启; 1: 关闭 (默认)。(读 / 写)

Register 21.10: RSA_SEARCH_ENABLE_REG (0x0824)



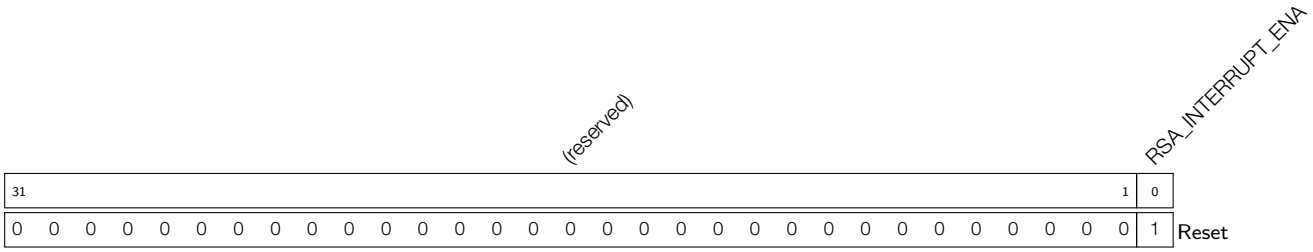
RSA_SEARCH_ENABLE 模幂运算中的 search 加速选项。1: 开启; 0: 关闭 (默认)。(读 / 写)

Register 21.11: RSA_SEARCH_POS_REG (0x0828)



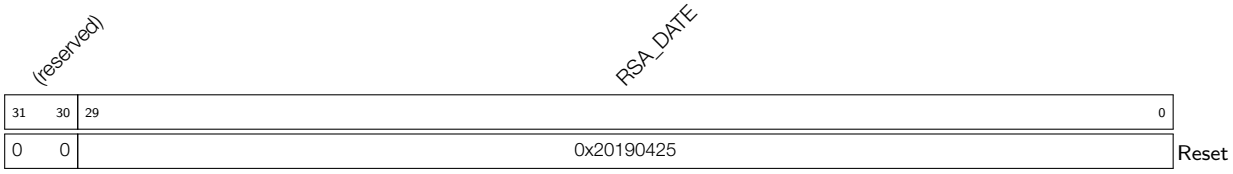
RSA_SEARCH_POS 模幂运算中的 search 加速选项。用于配置 search 的起始位置（读 / 写）。

Register 21.12: 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。（读 / 写）

Register 21.13: RSA_DATE_REG (0x0830)



RSA_DATE 版本控制寄存器（读 / 写）。

22. 随机数发生器

22.1 概述

ESP32-S2 内置一个真随机数发生器，其生成的随机数可作为加密等操作的基础。

22.2 主要特性

该随机数发生器可生成真随机数，即所有生成的随机数在特定范围内出现的概率完全一致。

22.3 功能描述

在正确使用的情况下，系统每次从随机数发生器中的寄存器 RNG_DATA_REG 读到的每一个 32 位值都是真随机数。这些真随机数来自系统中的热噪声。

噪声源可以来自高速 ADC 或 SAR ADC 或两者兼有。

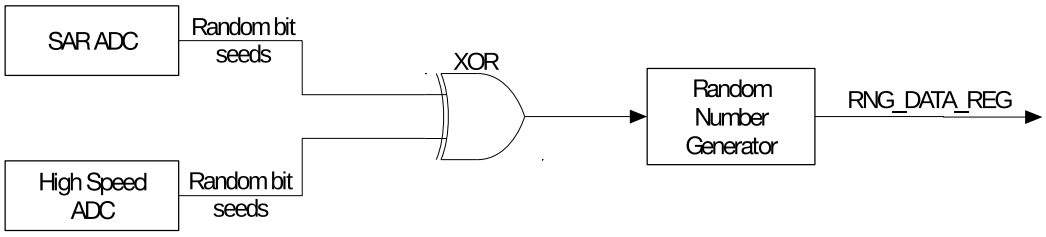


图 22-1. 噪声源

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 寄存器时的速度不超过 5 MHz。

当 SAR ADC 打开时，每个 8 MHz 时钟周期内（来自内部 RC 振荡器，详情可见 复位和时钟），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 寄存器时的速度不超过 500 kHz。

我们在仅高速 ADC 打开的状态下，以 5 MHz 的速率从 RNG_DATA_REG 读取了 2 GB 的数据样本，并使用 Dieharder 随机数测试套件（版本 3.31.1）对样本进行了测试。最终，样本通过了所有测试。

> **说明：**
> 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。所以建议在 Wi-Fi 开启时，使用 SAR ADC 产生随机数。

未来，我们将提供一个生成随机数的系统 API，建议客户直接调用该 API 生成随机数。

22.4 基地址

用户可以通过两个不同的寄存器基地址访问随机数发生器，如表 110 所示。更多信息，请访问章节 1 系统和存储器。

表 110: 随机数发生器基地址

访问总线	基地址
PeriBUS1	0x3F435000
PeriBUS2	0x60035000

22.5 寄存器列表

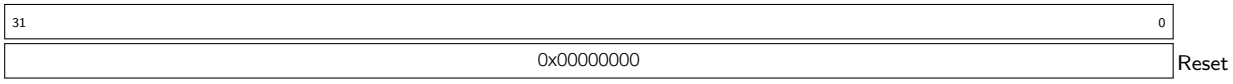
请注意，下表中的地址都是相对于随机数发生器基地址的地址偏移量（相对地址）。更多有关随机数发生器基地址的信息，请前往 22.4 章节。

名称	描述	地址	访问
RNG_DATA_REG	随机数数据	0x0110	只读

22.6 寄存器

请注意，这里的地址都是相对于随机数发生器基地址的地址偏移量（相对地址）。更多有关随机数发生器基地址的信息，请前往 22.4 章节。

Register 22.1: RNG_DATA_REG (0x0110)



RNG_DATA 随机数来源。(只读)

23. 片外存储器加密与解密

23.1 概述

ESP32-S2 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（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 功能描述

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 23-1 所示。

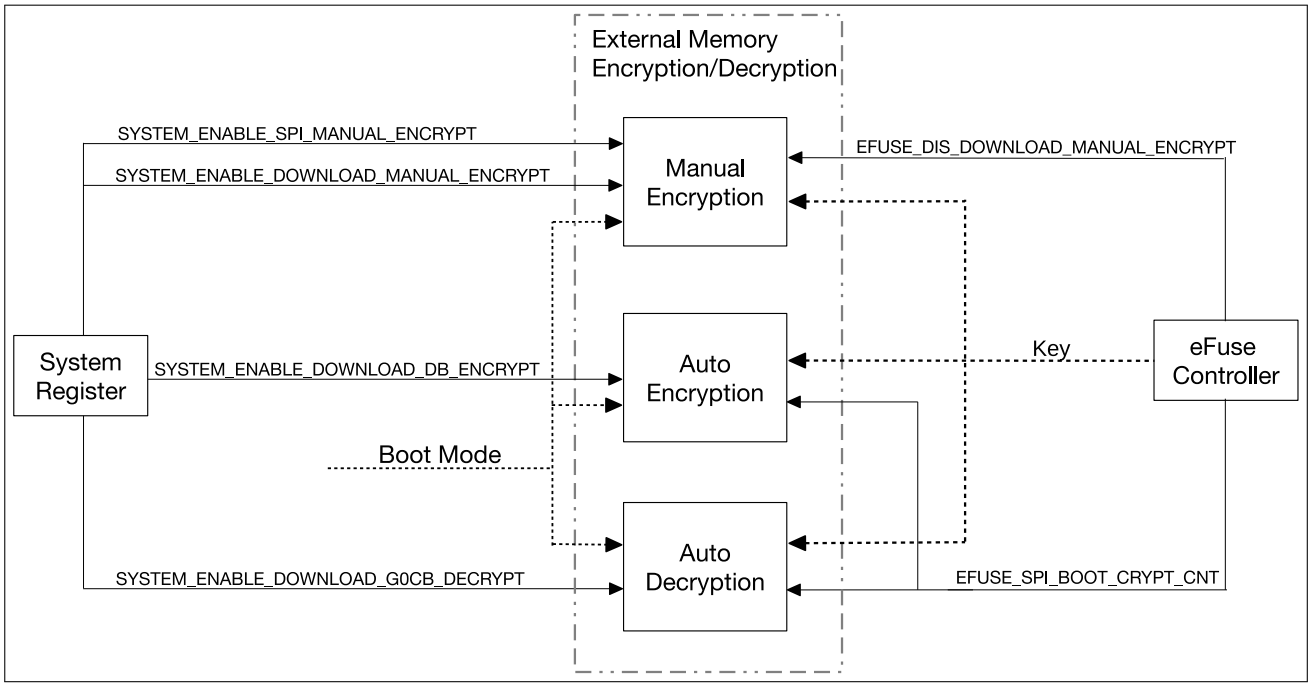


图 23-1. 片外存储器加解密模块架构

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。

外设 System 寄存器中与片外存储器加解密相关的是寄存器 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 中的下面 4 个位：

- SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT
- SYSTEM_ENABLE_DOWNLOAD_G0CB_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。

23.3.1 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 标准。

23.3.2 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 值取决于 eFuse 中的 BLOCK4 ~ BLOCK9 的某个 BLOCK 或两个 BLOCK 的内容。为方便描述，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。$Block_A$ 中存放 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。$Block_B$ 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。$Block_C$ 中存放 256 位的 Key_C。

根据 $Block_A$、$Block_B$ 和 $Block_C$ 存在与否，不同的组合将产生不同的 Key 值，如表 112 所示。

表 112: Key 值映射表

$Block_A$	$Block_B$	$Block_C$	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

注：表 112 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 16 *eFuse 控制器*。

23.3.3 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 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*)，这是一个物理地址，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 flash 中的地址段 0x130 ~ 0x13F 中，则目标空间为 0x130 ~ 0x13F，目标类型为 0 (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。

23.3.4 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) 构成的寄存器堆专用于数据填充，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堆中，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堆中。

目标空间映射到寄存器堆的方法为：

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，寄存器堆中的所有寄存器都将被使用，目标空间中的地址与寄存器堆之间的填充映射关系如表 113 所示。

表 113: 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

$offset$	寄存器	$offset$	寄存器
0x1C	XTS_AES_PLAIN_7_REG	0x3C	XTS_AES_PLAIN_15_REG

23.3.5 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、外设 System 寄存器、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 flash。

此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 23.3.3。
2. 用明文填充寄存器堆 XTS_AES_PLAIN_n_REG (n: 0-15)。请参考章节 23.3.4 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。
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。

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。

- 手动加密模块通过调用 AES 加速器完成计算，在此期间禁止用户访问 AES。

23.3.6 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。外设 System 寄存器、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。加密算法使用的密钥 Key 同样无法被软件获取。
- 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。

23.3.7 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。外设 System 寄存器、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_G0CB_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。解密算法使用的密钥 Key 同样无法被软件获取。

- 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。

23.4 基地址

用户可以通过两个不同的寄存器基地址访问手动加密模块，如表 114 所示。更多有关通过不同总线访问外设的信息，请参考章节 1 系统和存储器。

表 114: 手动加密模块基地址

访问外设总线	地址值
PeriBUS1	0x3F43A000
PeriBUS2	0x6003A000

23.5 寄存器列表

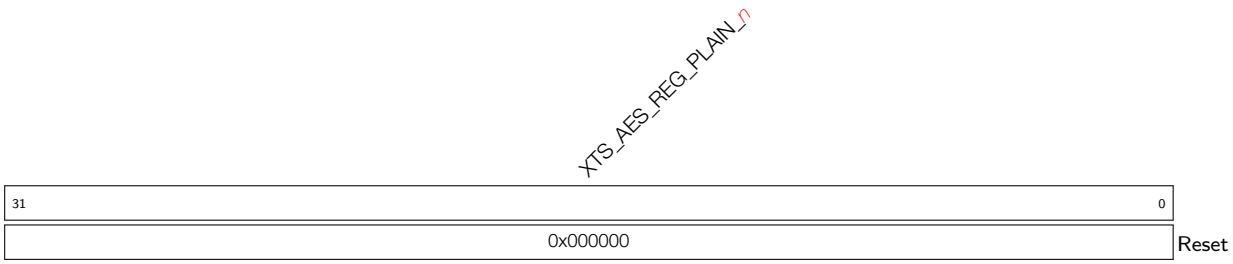
请注意，这里的地址是相对于手动加密模块基地址的地址偏移量（相对地址）。请参阅章节 23.4 获取有关手动加密模块的基地址的信息。

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	0x0100	读/写
XTS_AES_PLAIN_1_REG	明文寄存器 1	0x0104	读/写
XTS_AES_PLAIN_2_REG	明文寄存器 2	0x0108	读/写
XTS_AES_PLAIN_3_REG	明文寄存器 3	0x010C	读/写
XTS_AES_PLAIN_4_REG	明文寄存器 4	0x0110	读/写
XTS_AES_PLAIN_5_REG	明文寄存器 5	0x0114	读/写
XTS_AES_PLAIN_6_REG	明文寄存器 6	0x0118	读/写
XTS_AES_PLAIN_7_REG	明文寄存器 7	0x011C	读/写
XTS_AES_PLAIN_8_REG	明文寄存器 8	0x0120	读/写
XTS_AES_PLAIN_9_REG	明文寄存器 9	0x0124	读/写
XTS_AES_PLAIN_10_REG	明文寄存器 10	0x0128	读/写
XTS_AES_PLAIN_11_REG	明文寄存器 11	0x012C	读/写
XTS_AES_PLAIN_12_REG	明文寄存器 12	0x0130	读/写
XTS_AES_PLAIN_13_REG	明文寄存器 13	0x0134	读/写
XTS_AES_PLAIN_14_REG	明文寄存器 14	0x0138	读/写
XTS_AES_PLAIN_15_REG	明文寄存器 15	0x013C	读/写
配置寄存器			
XTS_AES_LINESIZE_REG	加密块大小寄存器	0x0140	读/写
XTS_AES_DESTINATION_REG	加密类型寄存器	0x0144	读/写
XTS_AES_PHYSICAL_ADDRESS_REG	物理地址寄存器	0x0148	读/写
控制/状态寄存器			
XTS_AES_TRIGGER_REG	启动运算	0x014C	只写
XTS_AES_RELEASE_REG	释放控制	0x0150	只写
XTS_AES_DESTROY_REG	销毁控制	0x0154	只写
XTS_AES_STATE_REG	状态寄存器	0x0158	只读

名称	描述	地址	访问
版本寄存器			
XTS_AES_DATE_REG	版本控制寄存器	0x015C	只读

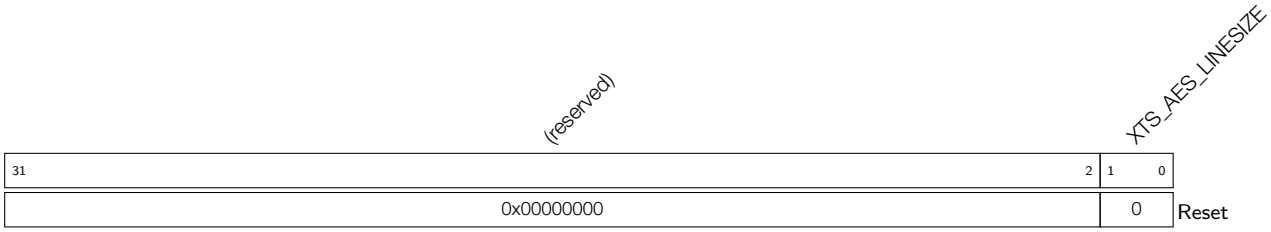
23.6 寄存器

Register 23.1: XTS_AES_PLAIN_*n*_REG (*n*: 0-15) (0x0100+4n*)**



XTS_AES_REG_PLAIN_*n* 存储明文的第 *n* 个 32 位部分。(读/写)

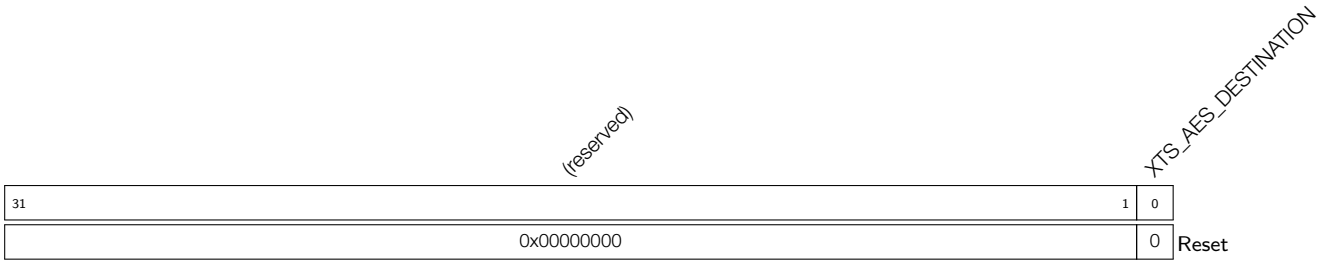
Register 23.2: XTS_AES_LINESIZE_REG (0x0140)



XTS_AES_LINESIZE 块大小寄存器，决定单次加密的数据量。(读/写)

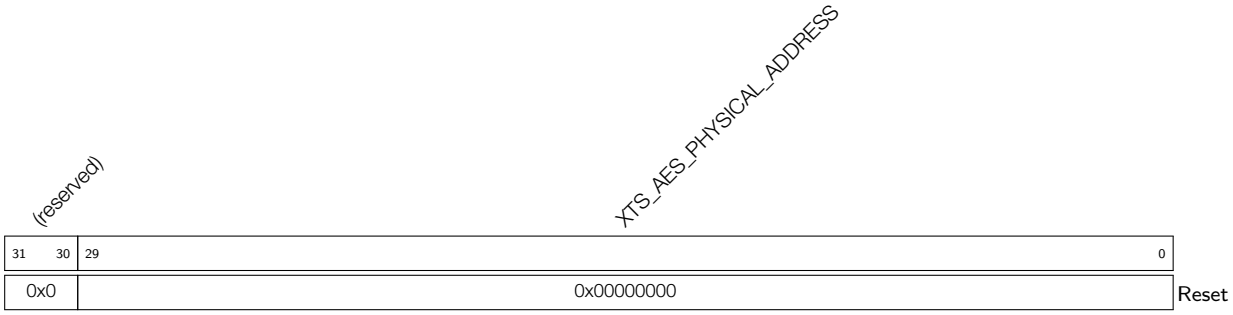
- 0：加密 128 位；
- 1：加密 256 位；
- 2：加密 512 位。

Register 23.3: XTS_AES_DESTINATION_REG (0x0144)



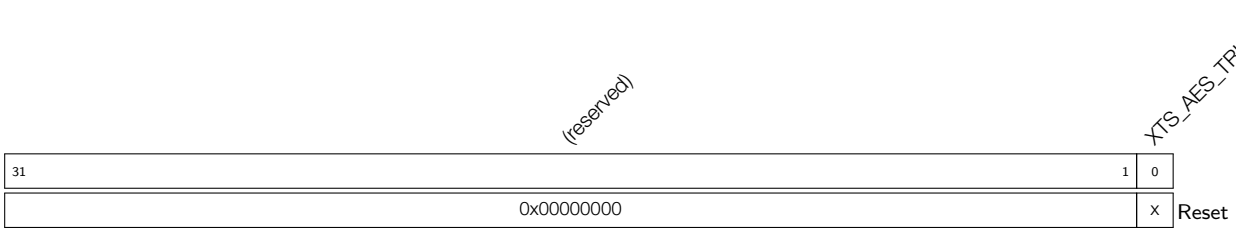
XTS_AES_DESTINATION 决定手动加密类型，目前只能手动加密 flash，所以只能为 0。用户不能写入 1，否则将发生错误。0：加密 flash；1：加密片外 RAM。(读/写)

Register 23.4: XTS_AES_PHYSICAL_ADDRESS_REG (0x0148)



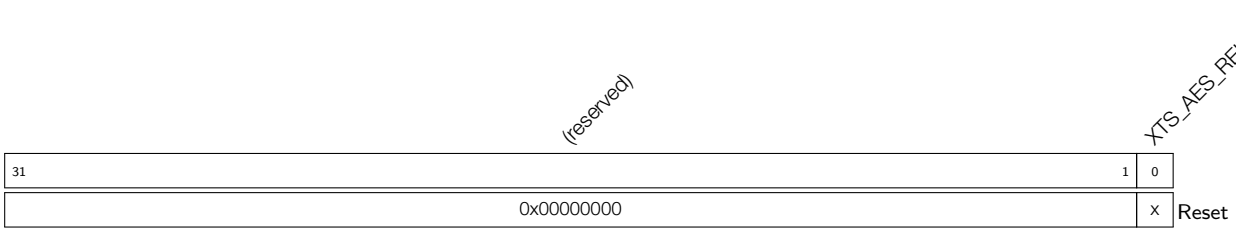
XTS_AES_PHYSICAL_ADDRESS 物理地址。(读/写)

Register 23.5: XTS_AES_TRIGGER_REG (0x014C)



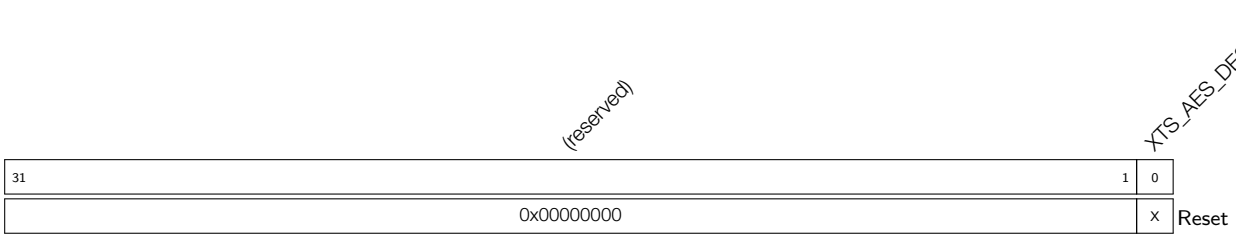
XTS_AES_TRIGGER 写入 1 使能手动加密运算。(只写)

Register 23.6: XTS_AES_RELEASE_REG (0x0150)



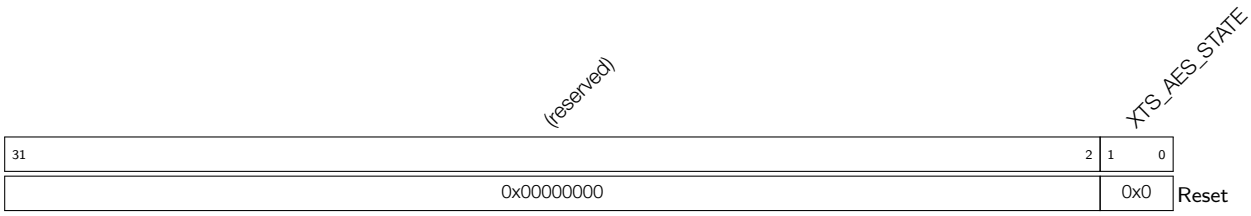
XTS_AES_RELEASE 写入 1 使加密结果对 SPI1 可见，因而 SPI1 可以获取到加密结果。(只写)

Register 23.7: XTS_AES_DESTROY_REG (0x0154)



XTS_AES_DESTROY 写入 1 销毁加密结果。(只写)

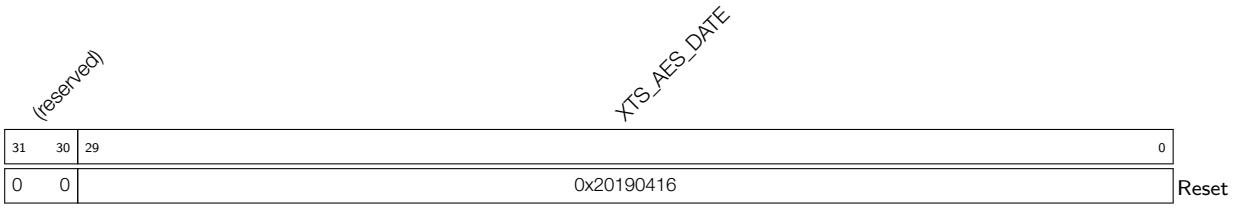
Register 23.8: XTS_AES_STATE_REG (0x0158)



XTS_AES_STATE 手动加密模块状态寄存器。(只读)

- 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。

Register 23.9: XTS_AES_DATE_REG (0x015C)



XTS_AES_DATE 版本控制寄存器。(只读)

24. 权限控制

24.1 概述

ESP32-S2 中不同总线类型、模块对存储器发起的访问权限是不同的。权限控制模块的功能就是通过相应的权限控制寄存器对上述所有的访问权限进行集中管理。

24.2 主要特性

- 内部存储器权限管理
  - SRAM 部分采用统一管理，部分采用分割管理
  - RTC FAST SRAM 采用分割管理
  - RTC SLOW SRAM 采用分割管理
- 片外存储器由 MMU 及权限控制模块管理

24.3 功能描述

24.3.1 内部存储器权限管理

内部存储器资源包括：ROM，SRAM，RTC FAST，RTC SLOW。各存储器的组成结构如下：

- ROM 总大小为 128 KB，由两个 64 KB 的块组成。
- SRAM 总大小为 320 KB，分成 4 个 8 KB 和 18 个 16 KB 的块，编号 Block 0 ~ 21，各 Block 的偏移地址范围如表 116 所示，不同总线访问的基地址不同，具体请见章节 24.3.1.1、24.3.1.2、24.3.1.3。对于 4 个 8 KB 的块 Block 0 ~ 3，每一个块都配备有独立的权限控制寄存器；而 16 KB 的块 Block 4 ~ 21 被视作一个大的存储空间，对该存储空间采用**分割管理**（将存储空间分割成高、低两片区域分别实施权限管理），分割地址不可落在 Block 4 ~ 5 的地址范围内。
- RTC FAST Memory 采用分割管理，高、低两片区域备有独立的权限管理寄存器。
- RTC SLOW Memory 采用分割管理，高、低两片区域备有独立的权限管理寄存器。

下文分别叙述 ESP32-S2 中能够访问上述内部存储资源的总线类型和模块。

表 116: SRAM Block 偏移地址范围

SRAM Block	偏移起始地址	偏移结尾地址
Block 0	0x0000	0x1FFF
Block 1	0x2000	0x3FFF
Block 2	0x4000	0x5FFF
Block 3	0x6000	0x7FFF
Block 4	0x8000	0xBFFF
Block 5	0xC000	0xFFFF
Block 6	0x10000	0x13FFF
Block 7	0x14000	0x17FFF
Block 8	0x18000	0x1BFFF
Block 9	0x1C000	0x1FFFF

SRAM Block	偏移起始地址	偏移结尾地址
Block 10	0x20000	0x23FFF
Block 11	0x24000	0x27FFF
Block 12	0x28000	0x2BFFF
Block 13	0x2C000	0x2FFFF
Block 14	0x30000	0x33FFF
Block 15	0x34000	0x37FFF
Block 16	0x38000	0x3BFFF
Block 17	0x3C000	0x3FFFF
Block 18	0x40000	0x43FFF
Block 19	0x44000	0x47FFF
Block 20	0x48000	0x4BFFF
Block 21	0x4C000	0x4FFFF

24.3.1.1 IBUS 指令总线的访问权限管理

具备权限管理的 IBUS 指令总线的地址范围为 0x4002_0000 ~ 0x4007_1FFF，该地址段包含有 SRAM 存储单元、RTC FAST Memory。访问不同存储单元都有独立的权限控制寄存器，由软件配置完成。

此外本章节中与 IBUS 总线相关的权限控制寄存器还被 PMS_PRO_IRAM0_LOCK 信号控制，当该信号配置为 1 时，权限寄存器中配置数值将被锁定，不允许再次更改。同时 PMS_PRO_IRAM0_LOCK 信号数值也将保持在 1，不允许更改。当且仅当 CPU 被复位时，PMS_PRO_IRAM0_LOCK 信号数值才被重新置 0。

SRAM 存储单元

CPU 通过 IBUS 总线对 SRAM 发起的访问类型有 R/W/X（读/写/取指），基地址是 0x4002_0000。软件可以提前配置好 IBUS 总线对 SRAM 每一个 Block 发起的合法的访问类型。相应的配置寄存器信息如表 117 所示。

表 117: IBUS 总线访问 SRAM 的权限管理

寄存器名	位的位置	含义
PMS_PRO_IRAM0_1_REG	[11:9]	配置 IBUS 总线对 SRAM Block 3 的访问权限（从高到低为 W/R/X）
	[8:6]	配置 IBUS 总线对 SRAM Block 2 的访问权限（从高到低为 W/R/X）
	[5:3]	配置 IBUS 总线对 SRAM Block 1 的访问权限（从高到低为 W/R/X）
	[2:0]	配置 IBUS 总线对 SRAM Block 0 的访问权限（从高到低为 W/R/X）
PMS_PRO_IRAM0_2_REG	[22:20]	配置 IBUS 总线对 SRAM Block 4 ~ 21 高地址段的访问权限（从高到低为 W/R/X）
	[19:17]	配置 IBUS 总线对 SRAM Block 4 ~ 21 低地址段的访问权限（从高到低为 W/R/X）
	[16:0]	配置 IBUS 总线在 SRAM Block 4 ~ 21 地址范围内的分割地址，基地址为 0x4000_0000。有关如何配置分割地址，请见下文说明。

说明：

* 分割地址为 32 位宽，其值为基地址 + 用户配置的域的值 ×4。比如，将 0x10000 写入 PMS_PRO_IRAM0_2_REG [16:0]，则 IBUS 总线在 SRAM Block 4 ~ 21 地址范围内的分割地址为 0x4004_0000。注意 IBUS 和 DBUS0 的分割地址不能落在 SRAM Block 0 ~ 5 的有效地址范围内。

若 CPU 通过 IBUS 总线对 SRAM 发起的访问类型与配置允许的访问类型不一致，例如软件配置 IBUS 总线只允许对 SRAM Block5 进行读写访问，但 CPU 通过 IBUS 总线对该 Block 发起了取指访问，权限控制模块将直接拒绝此次访问。

拒绝此次访问，但是 CPU 发起的访问不会被阻塞。具体表现如下：

- 写操作，将不会生效，不会改变存储单元内的数据内容；
- 读和取指令操作，得到的是无意义的数据。

如果 IBUS 总线的违法访问 SRAM 存储单元的中断被使能，权限控制模块还将记录当次访问地址和访问类型，同时给出中断信号。如果连续发生访问不通过情况，硬件只记录第一次出错的信息。访问错误中断为电平信号，需要软件清除。清除访问错误中断标志的同时，相关错误记录也会同时被清除。

RTC FAST Memory

CPU 通过 IBUS 总线可以对 RTC FAST Memory 进行 R/W/X 三种类型的访问。由于 RTC FAST 采用分割管理，软件需要分别为高、低地址段配置访问权限。相应的配置寄存器信息如表 118 所示。

表 118: IBUS 总线访问 RTC FAST 权限管理

寄存器名	位的位置	含义
PMS_PRO_IRAM0_3_REG	[16:14]	配置 IBUS 总线对 RTC FAST 高地址段的访问权限（从高到低为 W/R/X）
	[13:11]	配置 IBUS 总线对 RTC FAST 低地址段的访问权限（从高到低为 W/R/X）
	[10:0]	配置 IBUS 总线在 RTC FAST 地址范围内的分割地址，基地址为 0x4007_0000。关于如何配置分割地址，请见前文说明。

若 CPU 通过 IBUS 总线对 RTC FAST 发起的访问类型与配置允许的访问类型不一致，权限控制模块将直接拒绝此次访问。如果 IBUS 总线违法访问 RTC FAST SRAM 的中断被使能，权限控制模块还将记录当次访问地址和访问类型，同时给出中断信号。如果连续发生访问不通过情况，硬件只记录第一次出错的信息。访问错误中断为电平信号，需要软件清除。清除访问错误中断标志的同时，相关错误记录也会同时被清除。

24.3.1.2 DBUS0 总线的访问权限管理

CPU 通过 DBUS0 数据总线可以访问的地址范围有两段：0x3FFB_0000 ~ 0x3FFF_FFFF 和 0x3FF9_E000 ~ 0x3FF9_FFFF，包含 SRAM 存储单元、RTC FAST Memory。访问不同存储单元都有独立的权限控制寄存器，由软件配置完成。

此外本章节中与 DBUS0 总线相关的的权限控制寄存器还被 PMS_PRO_DRAM0_LOCK 信号控制，当该信号配置为 1 时，权限寄存器的配置数值将被锁定，不允许再次更改。同时 PMS_PRO_DRAM0_LOCK 信号数值也将保持在 1，不允许更改。当且仅当 CPU 被复位时，PMS_PRO_DRAM0_LOCK 信号数值才被重新置 0。

SRAM 存储单元

CPU 通过 DBUS0 总线对 SRAM 发起的访问类型有 R/W 两种，基地址是 0x3FFB_0000。相应地，软件可以提前配置好 DBUS0 总线对 SRAM 每一个 Block 发起的合法的访问类型。相应的配置寄存器信息如表 119 所示。

表 119: DBUS0 访问 SRAM 权限管理

寄存器名	位的位置	含义
PMS_PRO_DRAM0_1_REG	[28:27]	配置 DBUS0 总线对 SRAM Block 4 ~ 21 高地址段的访问权限（从高到低为 W/R）
	[26:25]	配置 DBUS0 总线对 SRAM Block 4 ~ 21 低地址段的访问权限（从高到低为 W/R）
	[24:8]	配置 DBUS0 总线在 SRAM Block 4 ~ 21 地址范围内的分割地址，基地址为 0x3FFB_0000。关于如何配置分割地址，请见章节 24.3.1.1 中的说明。
	[7:6]	配置 DBUS0 总线对 SRAM Block3 的访问权限（从高到低为 W/R）
	[5:4]	配置 DBUS0 总线对 SRAM Block2 的访问权限（从高到低为 W/R）
	[3:2]	配置 DBUS0 总线对 SRAM Block1 的访问权限（从高到低为 W/R）
	[1:0]	配置 DBUS0 总线对 SRAM Block0 的访问权限（从高到低为 W/R）

若 CPU 通过 DBUS0 总线对 SRAM 发起的访问类型与配置允许的访问类型不一致，权限控制模块将直接拒绝此次访问。如果 DBUS0 总线违法访问 SRAM 存储单元的中断被使能，权限控制模块还将记录当次访问地址、访问类型和访问大小（字节、半字、字访问），同时给出中断信号。如果连续发生访问不通过情况，硬件只记录第一次出错的信息。访问错误中断为电平信号，需要软件清除。清除访问错误中断标志的同时，相关错误记录也会同时被清除。

RTC FAST Memory

CPU 通过 DBUS0 总线可以对 RTC FAST Memory 进行 R/W 两种类型的访问。由于 RTC FAST 采用分割管理，软件需要分别为高、低地址段配置访问权限。相应的配置寄存器信息如表 120 所示。

表 120: DBUS0 总线访问 RTC FAST 权限管理

寄存器名	位的位置	含义
PMS_PRO_DRAM0_2_REG	[14:13]	配置 DBUS0 总线对 RTC FAST 高地址段的访问权限（从高到低为 W/R）
	[12:11]	配置 DBUS0 总线对 RTC FAST 低地址段的访问权限（从高到低为 W/R）
	[10:0]	配置 DBUS0 总线在 RTC FAST 地址范围内的分割地址，基地址为 0x3FF9_E000。关于如何配置分割地址，请见章节 24.3.1.1 中的说明。

若 CPU 通过 DBUS0 总线对 RTC FAST 发起的访问类型与配置允许的访问类型不一致，权限控制模块将直接拒绝此次访问。如果 DBUS0 总线违法访问 RTC FAST SRAM 的中断被使能，权限控制模块还将记录当次访问地址、访问类型和访问大小（字节、半字、字访问）（字节、半字、字访问），同时给出中断信号。如果连续发生访问不通过情况，硬件只记录第一次出错的信息。访问错误中断为电平信号，需要软件清除。清除访问错误中断标志的同时，相关错误记录也会同时被清除。

24.3.1.3 片上 DMA 访问权限控制

ESP32-S2 上 DMA 对 SRAM 的访问分为 3 类，分别为 Internal DMA，Copy DMA RX（接收）通道，Copy DMA TX（发送）通道，基地址是 0x3FFB_0000，地址范围为：0x3FFB_0000 ~ 0x3FFF_FFFF。软件可以提前配置允许 DMA 资源对 SRAM 每一个 Block 发起访问的类型。相应的配置寄存器信息如表 121 所示，其中 XX 可以是 APB，TX 和 RX，分别对应 Internal DMA, TX Copy DMA 和 RX Copy DMA 资源的配置寄存器。更多有关 DMA 的内容，请参考章节 7 *DMA 控制器*。

此外本章节中与 DMA 相关的的权限控制寄存器还被 PMS_DMA_XX_I_LOCK 信号控制，当该信号配置为 1 时，权限寄存器的配置数值将被锁定，不允许再次更改。同时 PMS_DMA_XX_I_LOCK 信号数值也将保持在 1，不允许更改。当且仅当 CPU 被复位时，PMS_DMA_XX_I_LOCK 信号数值才被重新置 0。

表 121: 片上 DMA 资源访问 SRAM 权限管理

寄存器名	位的位置	含义
PMS_DMA_XX_I_1_REG	[28:27]	配置 DMA 资源对 SRAM Block 4 ~ 21 高地址段的访问权限（从高到低为 W/R）
	[26:25]	配置 DMA 对 SRAM Block 4 ~ 21 低地址段的访问权限（从高到低为 W/R）
	[24:8]	配置 DMA 在 SRAM Block 4 ~ 21 地址范围内的分割地址，基地址为 0x3FFB_0000。关于如何配置分割地址，请见章节 24.3.1.1 中的说明。
	[7:6]	配置 DMA 对 SRAM Block3 的访问权限（从高到低为 W/R）
	[5:4]	配置 DMA 对 SRAM Block2 的访问权限（从高到低为 W/R）
	[3:2]	配置 DMA 对 SRAM Block1 的访问权限（从高到低为 W/R）
	[1:0]	配置 DMA 对 SRAM Block0 的访问权限（从高到低为 W/R）

片上 DMA 资源发起对 SRAM 的访问类型与配置允许的访问类型不一致时，权限控制模块将直接拒绝此次访问。如果 Internal DMA，TX Copy DMA 或 RX Copy DMA 违法访问 SRAM 的中断被使能，权限控制模块还将记录当次访问地址和访问类型，同时给出中断信号。如果连续发生访问不通过情况，硬件只记录第一次出错的信息。访问错误中断为电平信号，需要软件清除。清除访问错误中断标志的同时，相关错误记录也会同时被清除。

24.3.1.4 PeriBus1 总线的访问权限管理

CPU 通过 PeriBus1 总线访问地址范围为：0x3F40_0000 ~ 0x3F4F_FFFF，包含 RTC SLOW SRAM。PeriBus1 总线可以对 RTC SLOW 进行读/写访问，不支持取指。上述读/写权限皆由软件配置。

除此以外，软件还可以配置是否允许 PeriBus1 总线访问地址段 0x3F40_0000 ~ 0x3F4B_FFFF 下的外设。

由于 CPU 会通过 PeriBus1 总线发起预测性读操作，可能会产生外设的 FIFO 读取错误。为了防止 FIFO 读取错误，相应外设的 FIFO 地址已经固化在硬件中并且不可更改（如表 122 所示）。另外预留了 4 个可由软件配置的地址寄存器 PMS_PRO_DPORT_2~5_REG，用户可将读保护地址写入这些寄存器中。PeriBus1 总线对上述地址发起的读访问将被拒绝。相应的配置寄存器信息如表 123 所示。

表 122: 外设和 FIFO 地址

外设	FIFO 地址
ADDR_RTCSLOW	0x6002_1000
ADDR_FIFO_UART0	0x6000_0000

外设	FIFO 地址
ADDR_FIFO_UART1	0x6001_0000
ADDR_FIFO_UART2	0x6002_E000
ADDR_FIFO_I2S0	0x6000_F004
ADDR_FIFO_I2S1	0x6002_D004
ADDR_FIFO_RMT_CH0	0x6001_6000
ADDR_FIFO_RMT_CH1	0x6001_6004
ADDR_FIFO_RMT_CH2	0x6001_6008
ADDR_FIFO_RMT_CH3	0x6001_600C
ADDR_FIFO_I2C_EXT0	0x6001_301C
ADDR_FIFO_I2C_EXT1	0x6002_701C
ADDR_FIFO_USB_0	0x6008_0020
ADDR_FIFO_USB_1_L	0x6008_1000
ADDR_FIFO_USB_1_H	0x6009_0FFF

此外本章节中与 PeriBus1 总线相关的的权限控制寄存器还被 PMS_PRO_DPORT_LOCK 信号控制，当该信号配置为 1 时，权限寄存器的配置数值将被锁定，不允许再次更改。同时 PMS_PRO_DPORT_LOCK 信号数值也将保持在 1，不允许更改。当且仅当 CPU 被复位时，PMS_PRO_DPORT_LOCK 信号数值才被重新置 0。

表 123: PeriBus1 总线的访问权限管理

寄存器名	位位置	含义
PMS_PRO_DPORT_1_REG	[19:16]	每位的值决定了是否启用对应的预留的 FIFO 地址
	[15:14]	配置 PeriBus1 总线对 RTC SLOW 高地址段的访问权限（从高到低为 W/R）
	[13:12]	配置 PeriBus1 总线对 RTC SLOW 低地址段的访问权限（从高到低为 W/R）
	[11:1]	配置 PeriBus1 总线在 RTC SLOW 地址范围内的分割地址，基地址为 0x3F42_1000 。关于如何配置分割地址，请见章节 24.3.1.1 中的说明。
	[0]	配置 PeriBus1 总线是否具有地址段 0x3F40_0000～0x3F4B_FFFF 下外设的访问权限
PMS_PRO_DPORT_2_REG	[17:0]	第 0 个 PeriBus1 总线不可读地址，由 PMS_PRO_DPORT_RESERVE_FIFO_VALID [16] 启用
PMS_PRO_DPORT_3_REG	[17:0]	第 1 个 PeriBus1 总线不可读地址，由 PMS_PRO_DPORT_RESERVE_FIFO_VALID [17] 启用
PMS_PRO_DPORT_4_REG	[17:0]	第 2 个 PeriBus1 总线不可读地址，由 PMS_PRO_DPORT_RESERVE_FIFO_VALID [18] 启用
PMS_PRO_DPORT_5_REG	[17:0]	第 3 个 PeriBus1 总线不可读地址，由 PMS_PRO_DPORT_RESERVE_FIFO_VALID [19] 启用

当 PeriBus1 总线上发起对不可读地址的读访问或者访问 RTC SLOW Memory 的类型与配置允许的访问类型不一致时，权限控制模块都将拒绝此次访问。如果 PeriBus1 总线违法访问 RTC SLOW Memory 的中断被使能，权限控制模块还将记录当次访问地址、访问类型和访问大小（字节、半字、字访问），同时给出中断信号。如果连续发生访问不通过情况，硬件只记录第一次出错的信息。访问错误中断为电平信号，需要软件清除。清除访

问错误中断标志的同时，相关错误记录也会同时被清除。

24.3.1.5 PeriBus2 的访问权限管理

具备权限管理的 PeriBus2 总线地址范围有两段：0x5000_0000 ~ 0x5000_1FFF 和 0x6000_0000 ~ 0x600B_FFFF，包含 RTC SLOW 的两种访问地址、外设模块。

PeriBus2 总线访问 RTC SLOW Memory 时支持 R/W/X 三种访问类型。PeriBus2 总线上有两片地址区域可以访问 RTC SLOW Memory，分别是 RTCSlow_0、RTCSlow_1。软件可以配置允许 CPU 通过上述两片地址区域的发起的访问类型。相应的配置寄存器信息如表 124 所示。

PeriBus2 总线不支持对外设的进行取指操作。若对外设发起取指操作，得到的是无意义的数据。

此外本章节中与 PeriBus2 总线相关的的权限控制寄存器还被 PMS_PRO_AHB_LOCK 信号控制，当该信号配置为 1 时，权限寄存器的配置数值将被锁定，不允许再次更改。同时 PMS_PRO_AHB_LOCK 信号数值也将保持在 1，不允许更改。当且仅当 CPU 被复位时，PMS_PRO_AHB_LOCK 信号数值才被重新置 0。

表 124: PeriBus2 访问 RTC SLOW 权限管理

寄存器名	位位置	含义
PMS_PRO_AHB_1_REG	[16:14]	配置 PeriBus2 对 RTCSlow_0 高地址段的访问权限（从高到低为 W/R/X）
	[13:11]	配置 PeriBus2 对 RTCSlow_0 低地址段的访问权限（从高到低为 W/R/X）
	[10:0]	配置 PeriBus2 对 RTCSlow_0 地址范围内的分割地址，基地址为 0x5000_0000。关于如何配置分割地址，请见章节 24.3.1.1 中的说明。
PMS_PRO_AHB_2_REG	[16:14]	配置 PeriBus2 对 RTCSlow_1 高地址段的访问权限（从高到低为 W/R/X）
	[13:11]	配置 PeriBus2 对 RTCSlow_1 低地址段的访问权限（从高到低为 W/R/X）
	[10:0]	配置 PeriBus2 对 RTCSlow_1 地址范围内的分割地址，基地址为 0x6002_1000。关于如何配置分割地址，请见章节 24.3.1.1 中的说明。

若 CPU 通过 PeriBus2 总线对 RTCSlow_0 或 RTCSlow_1 发起的访问类型与配置允许的访问类型不一致，权限控制模块将直接拒绝此次访问。如果 PeriBus2 的违法访问中断被使能，权限控制模块还将记录当次访问地址和访问类型，同时给出中断信号。如果连续发生访问不通过情况，硬件只记录第一次出错的信息。访问错误中断为电平信号，需要软件清除。清除访问错误中断标志的同时，相关错误记录也会同时被清除。

24.3.1.6 Cache 的访问权限管理

用户只允许将 SRAM Block 0 ~ 3 配置给 Icache 和 Dcache 使用。由于 Icache 和 Dcache 对内部存储器的访问地址空间最大为 16 KB，意味着最多可以访问 2 个 Block。Icache 和 Dcache 的访问地址范围被分成高地址段和低地址段，各 8 KB，分别以后缀 “_H” 和 “_L” 表示。

用户可以通过配置寄存器 PMS_PRO_CACHE_1_REG 将 SRAM Block 0 ~ 3 分配给 Icache_H, Icache_L, Dcache_H 和 Dcache_L 访问。配置数值与分配结果的关系如下表所示，表中 *FIELD* 为 PMS_PRO_CACHE_1_REG 寄存器的域 PMS_PRO_CACHE_CONNECT。

表 125: PMS_PRO_CACHE_1_REG 寄存器配置

SRAM Block 0-3	Dcache_H	Dcache_L	Icache_H	Icache_L
Block 0	*FIELD*[3]	*FIELD*[2]	*FIELD*[1]	*FIELD*[0]
Block 1	*FIELD*[7]	*FIELD*[6]	*FIELD*[5]	*FIELD*[4]
Block 2	*FIELD*[11]	*FIELD*[10]	*FIELD*[9]	*FIELD*[8]
Block 3	*FIELD*[15]	*FIELD*[14]	*FIELD*[13]	*FIELD*[12]

注意：

- 对于同一个 Block 而言，只能分配给 Dcache_H、Dcache_L、Icache_H、Icache_L 中的一个使用，意味着上表中同一行至多有一个为 1。
- 对于 Dcache_H、Dcache_L、Icache_H、Icache_L 而言，至多占用一个 Block，意味着上表中同一个列向至多有一个为 1。
- 被 cache 占用的部分不可以被 CPU 访问，未被 cache 占用的部分仍然可以被 CPU 访问。

24.3.1.7 其它内部存储器权限管理

在 ESP32-S2 中，支持软件读取 Trace memory 来获取 CPU 运行状态、调试程序。软件可以通过寄存器 PMS_PRO_TRACE_1 启用 Trace memory 功能。该寄存器同时受 PMS_PRO_TRACE_LOCK 信号控制，当该信号配置为 1 时，寄存器中配置的数值将被锁定，不允许再次更改。同时 PMS_PRO_TRACE_LOCK 信号数值也将保持在 1，不允许更改。当且仅当 CPU 被复位时，PMS_PRO_TRACE_LOCK 信号数值才被重新置 0。

启用 Trace memory 功能后，还需要配置寄存器 PMS_OCCUPY_3 选择 SRAM Block 4 ~ 21 中的某一个 Block 作为 Trace memory 的存储单元（至多选择一个，且被选中的 Block 无法再被 CPU 访问。）上述寄存器受 PMS_OCCUPY_LOCK 信号控制。当该信号配置为 1 时，寄存器中配置的数值将被锁定，不允许再次更改。同时 PMS_OCCUPY_LOCK 信号数值也将保持在 1，不允许更改。当且仅当 CPU 被复位时，PMS_OCCUPY_LOCK 信号数值才被重新置 0。

当某个 SRAM Block 为 Trace Memory 时，CPU 的 IBUS, DBUS 以及 DMA 都无法访问此 Block。

24.3.2 片外存储器权限管理

CPU 可以通过 SPI1、EDMA 及 cache 访问片外 flash 和 SRAM。SPI1 和 EDMA 是直接访问，cache 是间接访问。

24.3.2.1 Cache MMU

Cache MMU 用于控制 cache 和 EDMA 访问外部存储器，主要完成虚拟地址到实地址的转换。在开启 cache 和 EDMA 之前需要配置 MMU。当 cache 发生缺失或者将数据写回外部存储器时，cache 控制器会自动访问 MMU 并且生成访问外部存储器的实地址。当 EDMA 读写外部 SRAM 时，EDMA 控制器也会自动访问 MMU 并且生成访问外部 SRAM 的实地址。

表 126: MMU 表项

MMU 表项	位位置	含义
SRAM	[16]	外部存储器属性，用于指示 cache/EDMA 访问 flash 还是外部 SRAM。如果第 15 位为 1，则访问的是 flash，如果第 16 位为 1，则访问的是 SRAM。第 15、16 位不能同时为 1。
Flash	[15]	
Invalid	[14]	标记 MMU 表项是否有效；0 代表有效。
Page number	[13:0]	外部存储器分成多个块，称为“页”，页大小固定为 64 KB。实地址空间的页号用于指示 cache/EDMA 访问哪一个页。

当 cache 或者 EDMA 访问到无效的 MMU 页或者 MMU 中未指明外部存储器属性时均会触发 MMU 错误中断。

24.3.2.2 外部存储器权限控制

硬件将外部存储器 (flash + SRAM) 的实地址以 64 KB 为单位分为 8 (4+4) 个区域。每个区域可配置 W/R/X 访问。软件需要预先设定各区域的大小及其访问属性。

8 个区域分别对应 8 组寄存器，flash 和 SRAM 各有 4 组。每组寄存器包含有 3 部分:

1. 属性列表: APB_CTRL_X_ACS_n_ATTR_REG 共 3 位，从高到低分别为 W/R/F；
2. 区域起始地址: APB_CTRL_X_ACS_n_ADDR_REG，表示实际地址；
3. 区域长度: APB_CTRL_X_ACS_n_SIZE_REG，以 64 KB 为单位。

其中“X”表示 flash 或 SRAM，“n”表示 0 ~ 3。软件配置时需要注意的是 flash 对应的 4 个区域的总大小必须为 1 GB 且 SRAM 对应的 4 个区域总大小也必须为 1 GB。

CPU 直接访问外部存储器时，权限控制模块只监控 CPU 的写请求，不监控读请求。控制模块会根据访问的物理地址检查 CPU 的访问属性和预先设定的访问属性是否一致。如果事先设定该区域不可写，那么控制模块直接拒绝此次写访问，并记录当前访问的相关信息，并触发访问错误中断。

Cache 缺失、cache 写回和 cache 预取等操作会触发 cache 对外部存储器发起 W/R/X 请求。此时权限控制模块会根据访问的物理地址查询其所在区域的访问属性，并比较 cache 请求和预先设定的访问属性是否一致。只有 cache 请求和设定的访问属性一致时，权限控制模块才会将请求传递到外部存储器。若 cache 发起的是 R/F 请求，控制模块还要将该区域对应的访问属性反馈给 cache，同时 cache 会将访问属性存起来。当访问请求与当前访问属性不一致时，硬件会记录下当前访问的相关信息，并触发访问错误中断。

CPU 访问 cache 时，权限控制模块均要检查 CPU 的访问属性，检查方法是比较 CPU 的访问属性和存储在本地的访问属性列表，只有两者一致时 CPU 的访问才会成功。当访问请求与当前访问属性不一致时，硬件会记录下当前访问的相关信息，并触发访问错误中断。

如果连续发生访问属性检查不通过情况，那么硬件只记录第一次出错的信息，后面错误信息将被直接丢弃。另外访问错误中断为电平信号，软件清除访问错误中断标志的同时，错误记录也会同时被清除。

24.3.3 非对齐访问权限管理

ESP32-S2 中加入了非字对齐访问的监控。PeriBus1 总线或者 PeriBus2 总线对外设发起非字地址对齐且非字访问时，可能会触发中断或者系统异常。下表列出了所有可能的访问情形以及产生的结果，其中 **IN** 表示中断，

EX 表示异常，√ 表示正常。

表 127: 非对齐访问外设情况汇总

访问来源	访问地址	访问大小	读访问	写访问
PeriBus2 总线	0xXXXX_XXX0	字节	IN	IN
		半字	IN	IN
		字	√	√
	0xXXXX_XXX1	字节	IN	IN
		半字	√	IN
		字	√	IN
	0xXXXX_XXX2	字节	IN	IN
		半字	IN	IN
		字	√	IN
PeriBus1 总线下地址段 0x3F40_0000 - 0x3F4B_FFFF	0xXXXX_XXX0	字节	IN	IN
		半字	IN	IN
		字	√	√
	0xXXXX_XXX1	字节	IN	IN
		半字	EX	EX
		字	EX	EX
	0xXXXX_XXX2	字节	IN	IN
		半字	IN	IN
		字	EX	EX
PeriBus1 总线下地址段 0x3F4C_0000 - 0x3F4F_FFFF	0xXXXX_XXX0	字节访问	√	√
		半字访问	√	√
		字访问	√	√
	0xXXXX_XXX1	字节访问	IN	IN
		半字访问	EX	EX
		字访问	EX	EX
	0xXXXX_XXX2	字节访问	IN	IN
		半字访问	IN	IN
		字访问	EX	EX

24.4 基地址

用户可以通过寄存器基地址访问权限管理，如表 128 所示。更多信息，请访问章节 1 系统和存储器。

表 128: 中断矩阵基地址

访问总线	基地址
PeriBUS1	0x3F4C1000

24.5 寄存器列表

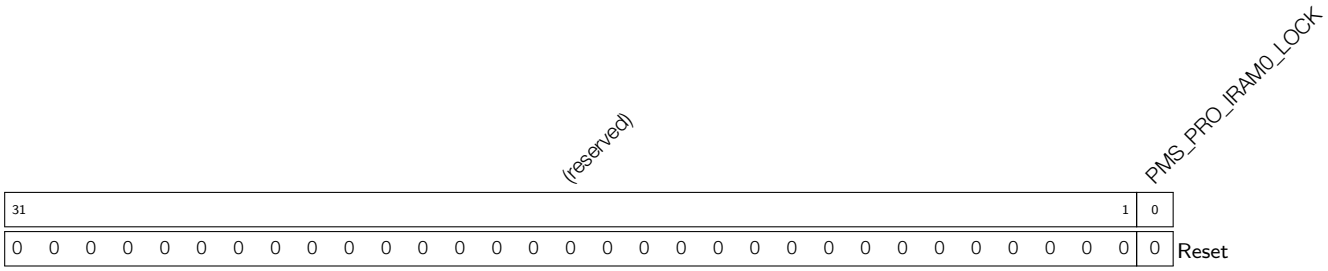
请注意，这里的地址是相对于权限管理基地址的地址偏移量（相对地址）。请参阅章节 24.4 获取有关权限管理的基地址的信息。

名称	描述	地址	访问
控制寄存器			
PMS_PRO_IRAM0_0_REG	IBUS 权限控制寄存器 0	0x0010	读/写
PMS_PRO_DRAM0_0_REG	DBUS 权限控制寄存器 0	0x0028	读/写
PMS_PRO_DPORT_0_REG	PeriBus1 权限控制寄存器 0	0x003C	读/写
PMS_PRO_AHB_0_REG	PeriBus2 权限控制寄存器 0	0x005C	读/写
PMS_PRO_TRACE_0_REG	Trace memory 功能权限控制寄存器 0	0x0070	读/写
PMS_PRO_CACHE_0_REG	Cache 权限控制寄存器 0	0x0078	读/写
PMS_DMA_APB_I_0_REG	内部 DMA 权限控制寄存器 0	0x008C	读/写
PMS_DMA_RX_I_0_REG	RX Copy DMA 权限控制寄存器 0	0x009C	读/写
PMS_DMA_TX_I_0_REG	TX Copy DMA 权限控制寄存器 0	0x00AC	读/写
PMS_CACHE_SOURCE_0_REG	Cache 访问权限控制寄存器 0	0x00C4	读/写
PMS_APB_PERIPHERAL_0_REG	外设访问权限控制寄存器 0	0x00CC	读/写
PMS_OCCUPY_0_REG	占用权限控制寄存器 0	0x00D4	读/写
PMS_CACHE_TAG_ACCESS_0_REG	Cache 标签权限控制寄存器 0	0x00E4	读/写
PMS_CACHE_MMU_ACCESS_0_REG	Cache MMU 权限控制寄存器 0	0x00EC	读/写
PMS_CLOCK_GATE_REG_REG	灵敏时钟门控寄存器	0x0104	读/写
配置寄存器			
PMS_PRO_IRAM0_1_REG	IBUS 权限控制寄存器 1	0x0014	读/写
PMS_PRO_IRAM0_2_REG	IBUS 权限控制寄存器 2	0x0018	读/写
PMS_PRO_IRAM0_3_REG	IBUS 权限控制寄存器 3	0x001C	读/写
PMS_PRO_DRAM0_1_REG	DBUS 权限控制寄存器 1	0x002C	读/写
PMS_PRO_DRAM0_2_REG	DBUS 权限控制寄存器 2	0x0030	读/写
PMS_PRO_DPORT_1_REG	PeriBus1 权限控制寄存器 1	0x0040	读/写
PMS_PRO_DPORT_2_REG	PeriBus1 权限控制寄存器 2	0x0044	读/写
PMS_PRO_DPORT_3_REG	PeriBus1 权限控制寄存器 3	0x0048	读/写
PMS_PRO_DPORT_4_REG	PeriBus1 权限控制寄存器 4	0x004C	读/写
PMS_PRO_DPORT_5_REG	PeriBus1 权限控制寄存器 5	0x0050	读/写
PMS_PRO_AHB_1_REG	PeriBus2 权限控制寄存器 1	0x0060	读/写
PMS_PRO_AHB_2_REG	PeriBus2 权限控制寄存器 2	0x0064	读/写
PMS_PRO_TRACE_1_REG	Trace memory 功能权限控制寄存器 1	0x0074	读/写
PMS_PRO_CACHE_1_REG	Cache 权限控制寄存器 1	0x007C	读/写
PMS_DMA_APB_I_1_REG	内部 DMA 权限控制寄存器 1	0x0090	读/写
PMS_DMA_RX_I_1_REG	RX Copy DMA 权限控制寄存器 1	0x00A0	读/写
PMS_DMA_TX_I_1_REG	TX Copy DMA 权限控制寄存器 1	0x00B0	读/写
PMS_APB_PERIPHERAL_1_REG	外设访问权限控制寄存器 1	0x00D0	读/写
PMS_OCCUPY_1_REG	占用权限控制寄存器 1	0x00D8	读/写
PMS_OCCUPY_3_REG	占用权限控制寄存器 3	0x00E0	读/写
PMS_CACHE_TAG_ACCESS_1_REG	Cache 标签权限控制寄存器 1	0x00E8	读/写
PMS_CACHE_MMU_ACCESS_1_REG	Cache MMU 权限控制寄存器 1	0x00F0	读/写
中断寄存器			
PMS_PRO_IRAM0_4_REG	IBUS 权限控制寄存器 4	0x0020	不定
PMS_PRO_IRAM0_5_REG	IBUS 状态寄存器	0x0024	只读
PMS_PRO_DRAM0_3_REG	DBUS 权限控制寄存器 3	0x0034	不定

名称	描述	地址	访问
PMS_PRO_DRAM0_4_REG	DBUS 状态寄存器	0x0038	只读
PMS_PRO_DPORT_6_REG	PeriBus1 权限控制寄存器 6	0x0054	不定
PMS_PRO_DPORT_7_REG	PeriBus1 状态寄存器	0x0058	只读
PMS_PRO_AHB_3_REG	PeriBus2 权限控制寄存器 3	0x0068	不定
PMS_PRO_AHB_4_REG	PeriBus2 状态寄存器	0x006C	只读
PMS_PRO_CACHE_2_REG	Cache 权限控制寄存器 2	0x0080	不定
PMS_PRO_CACHE_3_REG	Icache 状态寄存器	0x0084	只读
PMS_PRO_CACHE_4_REG	Dcache 状态寄存器	0x0088	只读
PMS_DMA_APB_I_2_REG	内部 DMA 权限控制寄存器 2	0x0094	不定
PMS_DMA_APB_I_3_REG	内部 DMA 状态寄存器	0x0098	只读
PMS_DMA_RX_I_2_REG	RX Copy DMA 权限控制寄存器 2	0x00A4	不定
PMS_DMA_RX_I_3_REG	RX Copy DMA 状态寄存器	0x00A8	只读
PMS_DMA_TX_I_2_REG	TX Copy DMA 权限控制寄存器 2	0x00B4	不定
PMS_DMA_TX_I_3_REG	TX Copy DMA 状态寄存器	0x00B8	只读
PMS_APB_PERIPHERAL_INTR_REG	PeriBus2 外设访问权限控制寄存器	0x00F4	不定
PMS_APB_PERIPHERAL_STATUS_REG	PeriBus2 外设访问状态寄存器	0x00F8	只读
PMS_CPU_PERIPHERAL_INTR_REG	PeriBus1 外设访问权限控制寄存器	0x00FC	不定
PMS_CPU_PERIPHERAL_STATUS_REG	PeriBus1 外设访问状态寄存器	0x0100	只读
版本控制寄存器			
PMS_DATE	版本控制寄存器。	0x0FFC	读/写

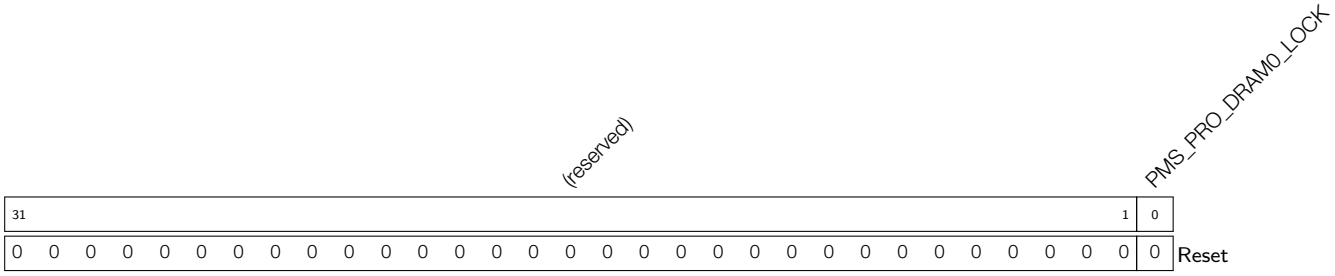
24.6 寄存器

Register 24.1: PMS_PRO_IRAM0_0_REG (0x0010)



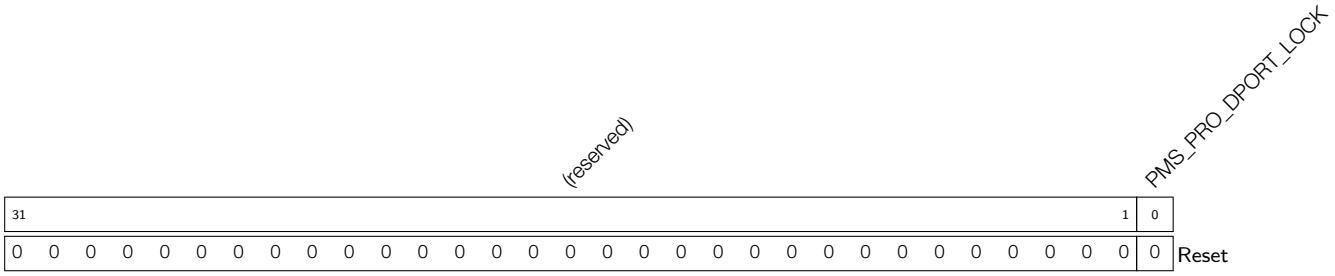
PMS_PRO_IRAM0_LOCK 锁定寄存器。设置为 1 将锁定 IBUS 权限控制寄存器。(读/写)

Register 24.2: PMS_PRO_DRAM0_0_REG (0x0028)



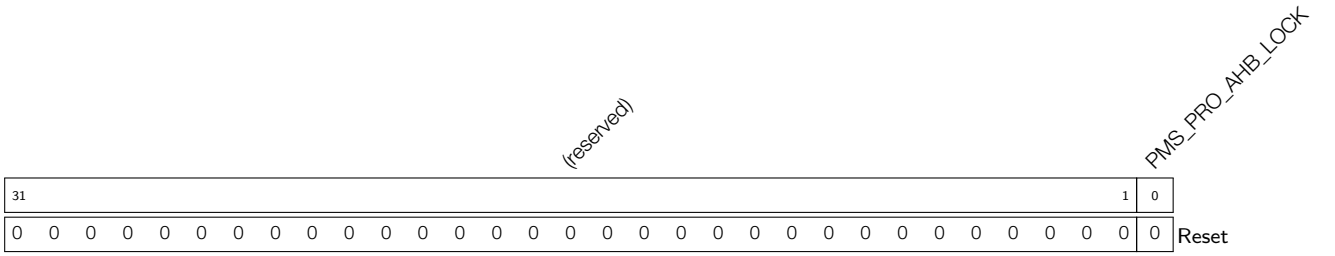
PMS_PRO_DRAM0_LOCK 锁定寄存器。设置为 1 将锁定 DBUS0 权限控制寄存器。(读/写)

Register 24.3: PMS_PRO_DPORT_0_REG (0x003C)



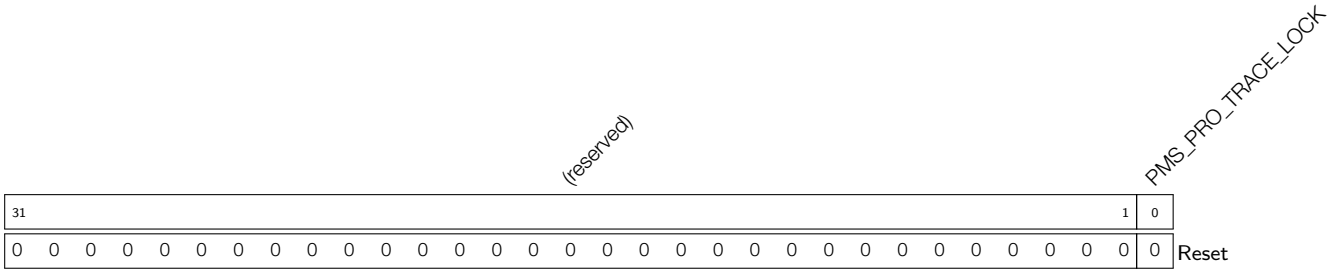
PMS_PRO_DPORT_LOCK 锁定寄存器。设置为 1 将锁定 PeriBus1 权限控制寄存器。(读/写)

Register 24.4: PMS_PRO_AHB_0_REG (0x005C)



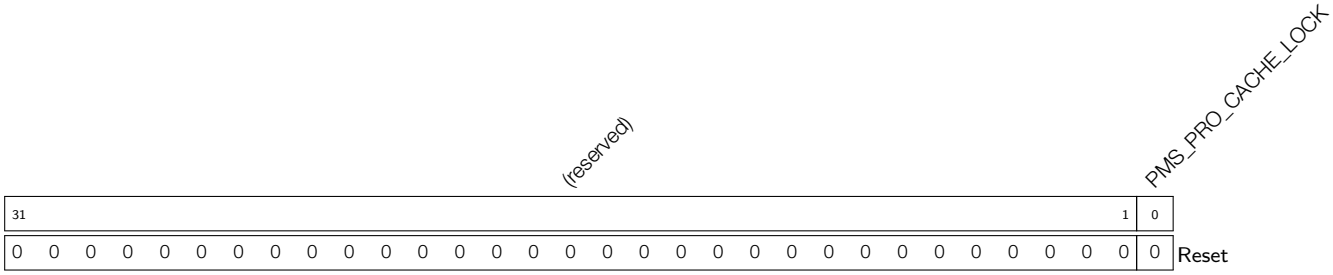
PMS_PRO_AHB_LOCK 锁定寄存器。设置为 1 将锁定 PeriBus2 权限控制寄存器。(读/写)

Register 24.5: PMS_PRO_TRACE_0_REG (0x0070)



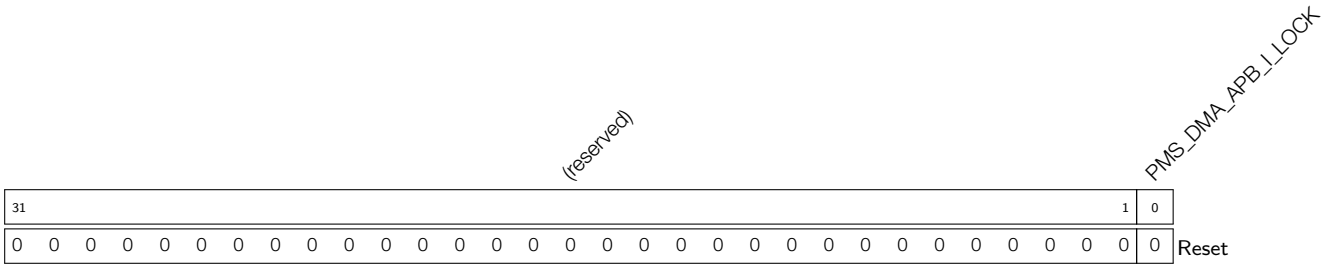
PMS_PRO_TRACE_LOCK 锁定寄存器。设置为 1 将锁定 Trace memory 功能权限控制寄存器。(读/写)

Register 24.6: PMS_PRO_CACHE_0_REG (0x0078)



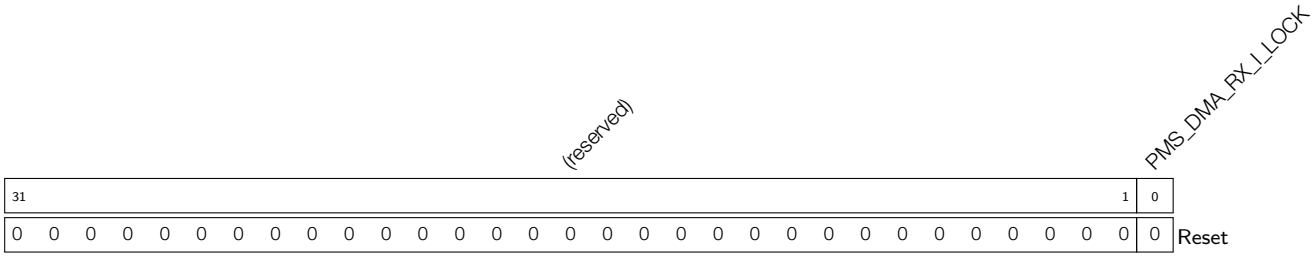
PMS_PRO_CACHE_LOCK 锁定寄存器。设置为 1 将锁定 cache 权限控制寄存器。(读/写)

Register 24.7: PMS_DMA_APB_I_0_REG (0x008C)



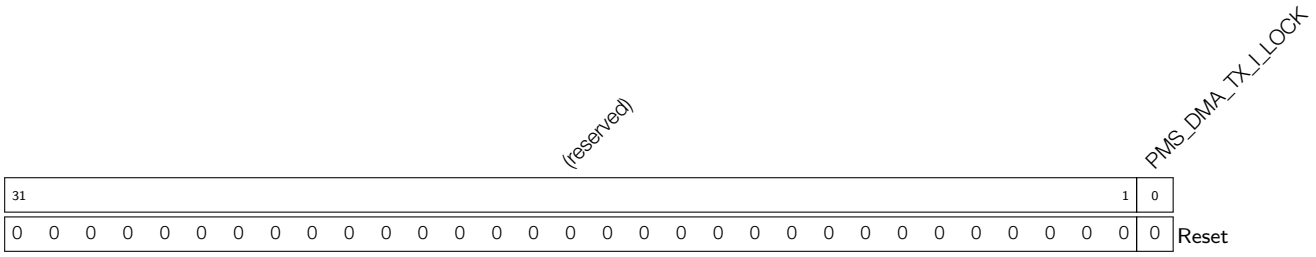
PMS_DMA_APB_I_LOCK 锁定寄存器。设置为 1 将锁定内部 DMA 权限控制寄存器。(读/写)

Register 24.8: PMS_DMA_RX_I_0_REG (0x009C)



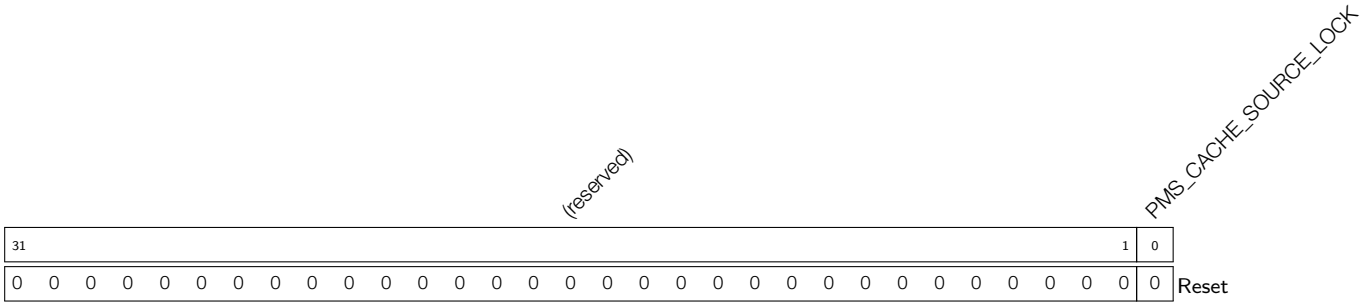
PMS_DMA_RX_I_LOCK 锁定寄存器。设置为 1 将锁定 RX Copy DMA 权限控制寄存器。(读/写)

Register 24.9: PMS_DMA_TX_I_0_REG (0x00AC)



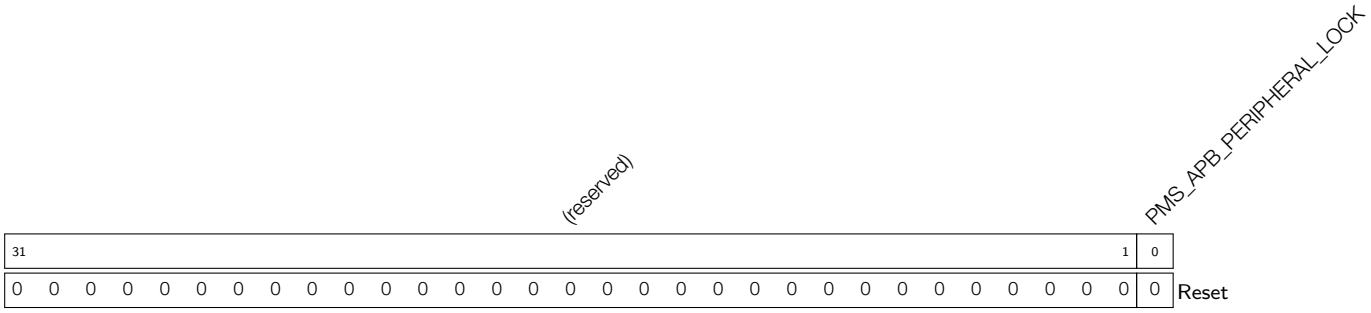
PMS_DMA_TX_I_LOCK 锁定寄存器。设置为 1 将锁定 TX Copy DMA 权限控制寄存器。(读/写)

Register 24.10: PMS_CACHE_SOURCE_0_REG (0x00C4)



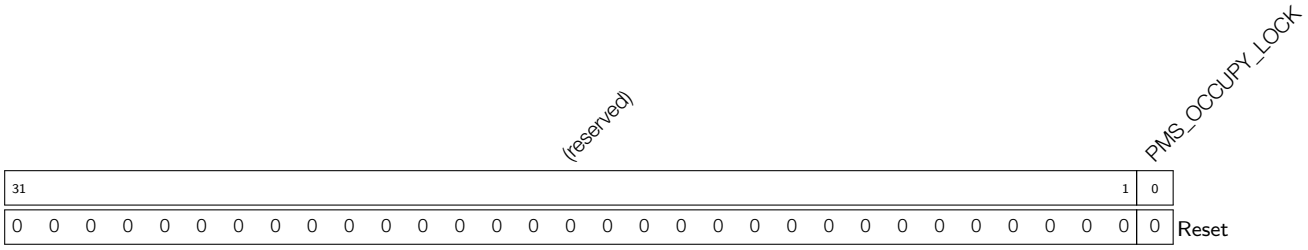
PMS_CACHE_SOURCE_LOCK 锁定寄存器。设置为 1 将锁定 cache 访问权限控制寄存器。(读/写)

Register 24.11: PMS_APB_PERIPHERAL_0_REG (0x00CC)



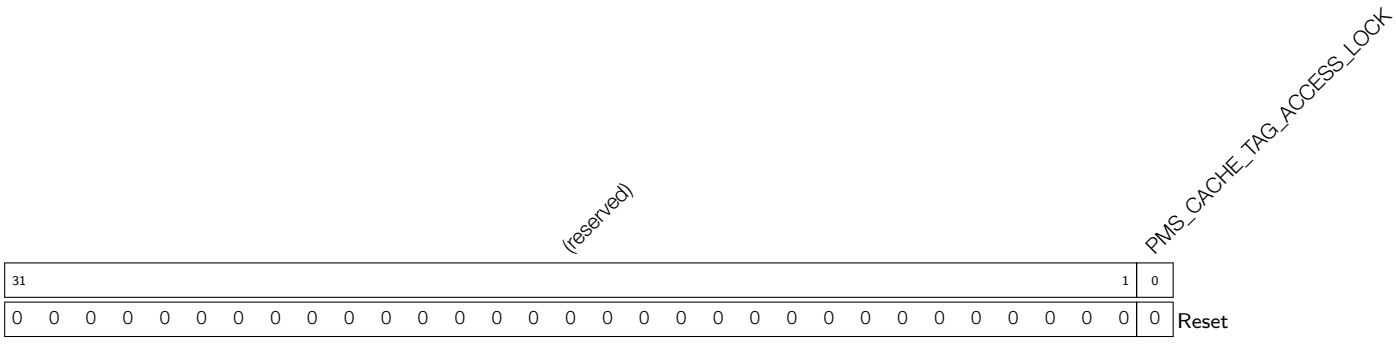
PMS_APB_PERIPHERAL_LOCK 锁定寄存器。设置为 1 将锁定 TX Copy DMA 权限控制寄存器。(读/写)

Register 24.12: PMS_OCCUPY_0_REG (0x00D4)



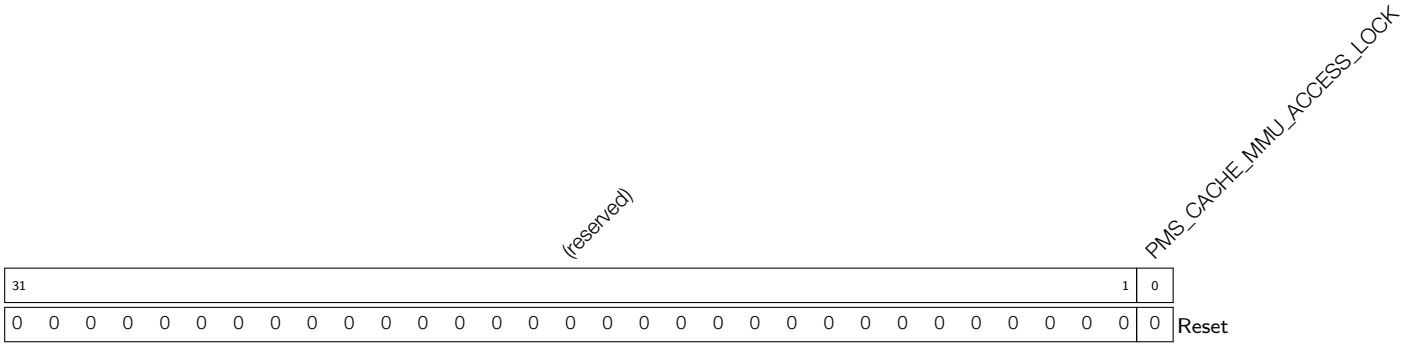
PMS_OCCUPY_LOCK 锁定寄存器。设置为 1 将锁定占用权限控制寄存器。(读/写)

Register 24.13: PMS_CACHE_TAG_ACCESS_0_REG (0x00E4)



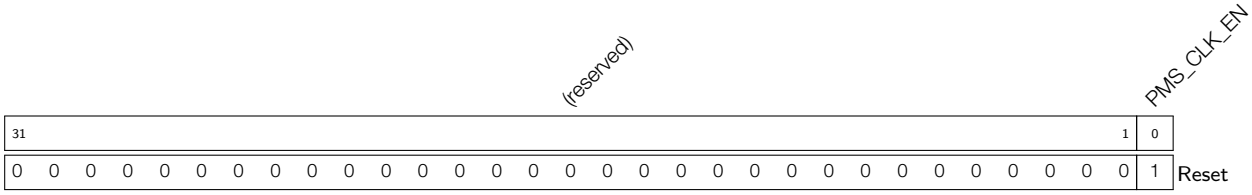
PMS_CACHE_TAG_ACCESS_LOCK 锁定寄存器。设置为 1 将锁定 cache 标签权限控制寄存器。(读/写)

Register 24.14: PMS_CACHE_MMU_ACCESS_0_REG (0x00EC)



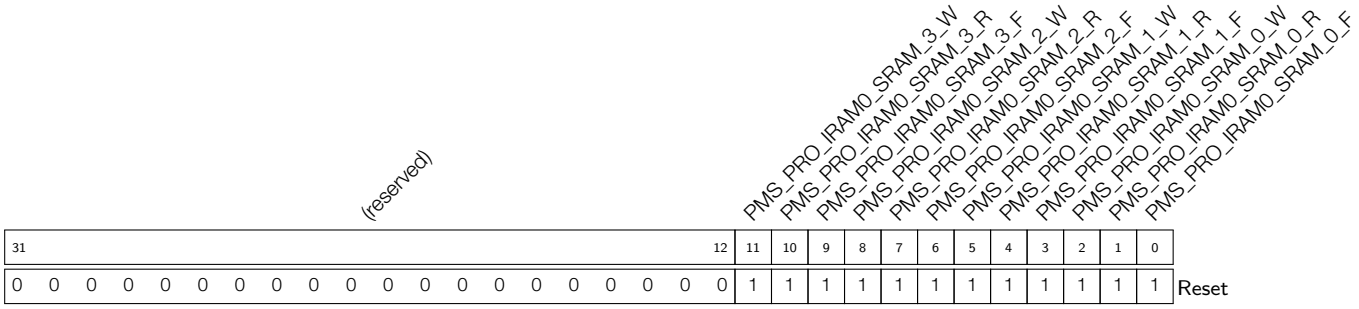
PMS_CACHE_MMU_ACCESS_LOCK 锁定寄存器。设置为 1 将锁定 cache MMU 权限控制寄存器。(读/写)

Register 24.15: PMS_CLOCK_GATE_REG_REG (0x0104)



PMS_CLK_EN 启用权限控制模块的时钟。(读/写)

Register 24.16: PMS_PRO_IRAM0_1_REG (0x0014)



PMS_PRO_IRAM0_SRAM_0_F 设置为 1 给予 IBUS 对 SRAM Block 0 进行取指操作的权限。(读/写)

PMS_PRO_IRAM0_SRAM_0_R 设置为 1 给予 IBUS 读 SRAM Block 0 的权限。(读/写)

PMS_PRO_IRAM0_SRAM_0_W 设置为 1 给予 IBUS 写 SRAM Block 0 的权限。(读/写)

PMS_PRO_IRAM0_SRAM_1_F 设置为 1 给予 IBUS 对 SRAM Block 1 进行取指操作的权限。(读/写)

PMS_PRO_IRAM0_SRAM_1_R 设置为 1 给予 IBUS 读 SRAM Block 1 的权限。(读/写)

PMS_PRO_IRAM0_SRAM_1_W 设置为 1 给予 IBUS 写 SRAM Block 1 的权限。(读/写)

PMS_PRO_IRAM0_SRAM_2_F 设置为 1 给予 IBUS 对 SRAM Block 2 进行取指操作的权限。(读/写)

PMS_PRO_IRAM0_SRAM_2_R 设置为 1 给予 IBUS 读 SRAM Block 2 的权限。(读/写)

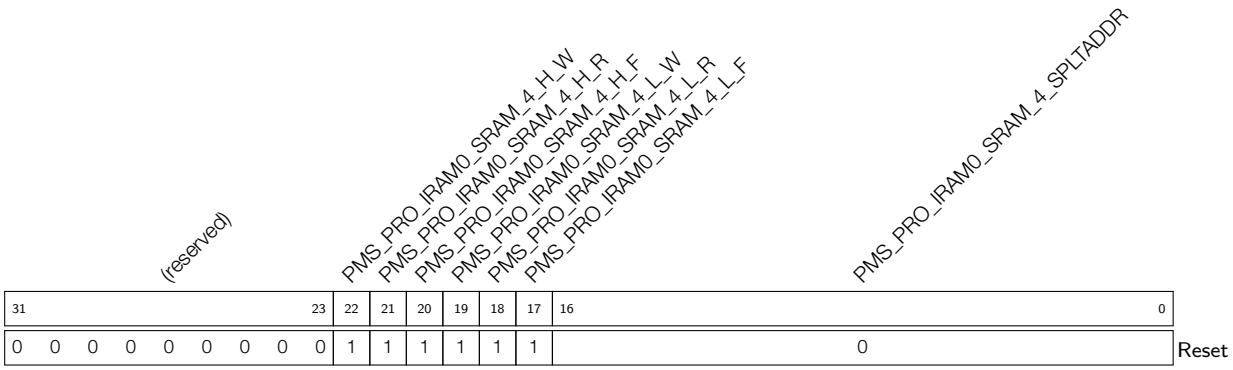
PMS_PRO_IRAM0_SRAM_2_W 设置为 1 给予 IBUS 写 SRAM Block 2 的权限。(读/写)

PMS_PRO_IRAM0_SRAM_3_F 设置为 1 给予 IBUS 对 SRAM Block 3 进行取指操作的权限。(读/写)

PMS_PRO_IRAM0_SRAM_3_R 设置为 1 给予 IBUS 读 SRAM Block 3 的权限。(读/写)

PMS_PRO_IRAM0_SRAM_3_W 设置为 1 给予 IBUS 写 SRAM Block 3 的权限。(读/写)

Register 24.17: PMS_PRO_IRAM0_2_REG (0x0018)



PMS_PRO_IRAM0_SRAM_4_SPLTADDR 配置 IBUS 访问 SRAM Block 4-21 的分割地址。(读/写)

PMS_PRO_IRAM0_SRAM_4_L_F 设置为 1 给予 IBUS 对 SRAM Block 4-21 的低地址区域进行取指操作的权限。(读/写)

PMS_PRO_IRAM0_SRAM_4_L_R 设置为 1 给予 IBUS 读 SRAM Block 4-21 低地址区域的权限。(读/写)

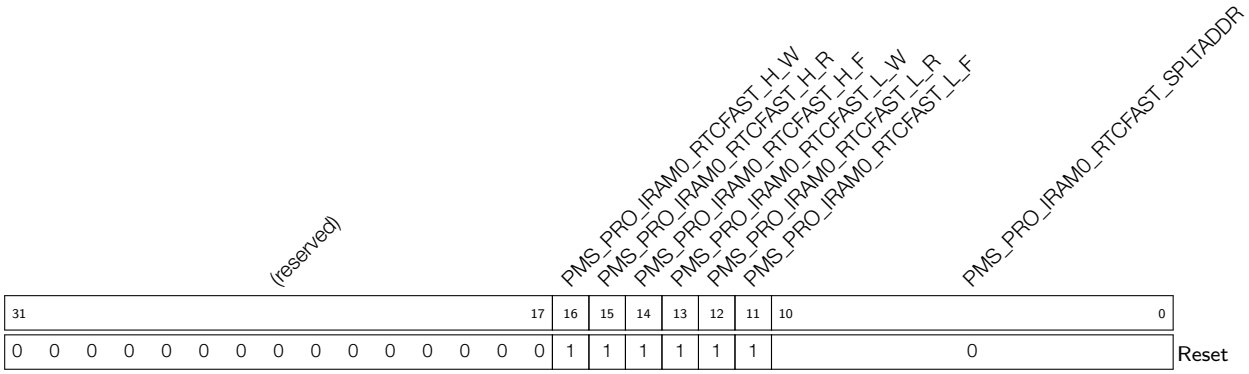
PMS_PRO_IRAM0_SRAM_4_L_W 设置为 1 给予 IBUS 写 SRAM Block 4-21 低地址区域的权限。(读/写)

PMS_PRO_IRAM0_SRAM_4_H_F 设置为 1 给予 IBUS 对 SRAM Block 4-21 的高地址区域进行取指操作的权限。(读/写)

PMS_PRO_IRAM0_SRAM_4_H_R 设置为 1 给予 IBUS 读 SRAM Block 4-21 高地址区域的权限。(读/写)

PMS_PRO_IRAM0_SRAM_4_H_W 设置为 1 给予 IBUS 写 SRAM Block 4-21 高地址区域的权限。(读/写)

Register 24.18: PMS_PRO_IRAM0_3_REG (0x001C)



PMS_PRO_IRAM0_RTCFAST_SPLTADDR 配置 IBUS 访问 RTC FAST Memory 的分割地址。(读/写)

PMS_PRO_IRAM0_RTCFAST_L_F 设置为 1 给予 IBUS 对 RTC FAST Memory 的低地址区域进行取指操作的权限。(读/写)

PMS_PRO_IRAM0_RTCFAST_L_R 设置为 1 给予 IBUS 读 RTC FAST Memory 低地址区域的权限。(读/写)

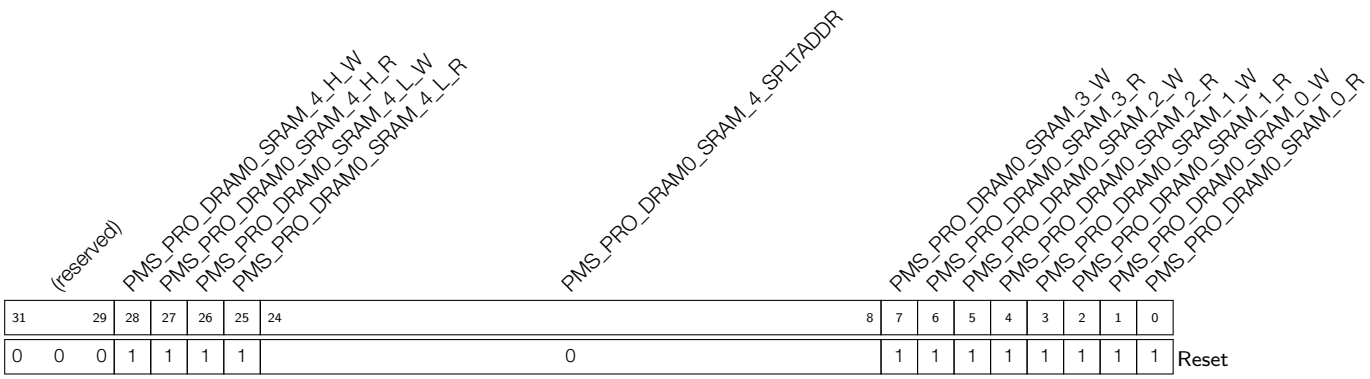
PMS_PRO_IRAM0_RTCFAST_L_W 设置为 1 给予 IBUS 写 RTC FAST Memory 低地址区域的权限。(读/写)

PMS_PRO_IRAM0_RTCFAST_H_F 设置为 1 给予 IBUS 对 RTC FAST Memory 的高地址区域进行取指操作的权限。(读/写)

PMS_PRO_IRAM0_RTCFAST_H_R 设置为 1 给予 IBUS 读 RTC FAST Memory 高地址区域的权限。(读/写)

PMS_PRO_IRAM0_RTCFAST_H_W 设置为 1 给予 IBUS 写 RTC FAST Memory 高地址区域的权限。(读/写)

Register 24.19: PMS_PRO_DRAM0_1_REG (0x002C)



PMS_PRO_DRAM0_SRAM_0_R 设置为 1 给予 DBUS0 读 SRAM Block 0 的权限。(读/写)

PMS_PRO_DRAM0_SRAM_0_W 设置为 1 给予 DBUS0 写 SRAM Block 0 的权限。(读/写)

PMS_PRO_DRAM0_SRAM_1_R 设置为 1 给予 DBUS0 读 SRAM Block 1 的权限。(读/写)

PMS_PRO_DRAM0_SRAM_1_W 设置为 1 给予 DBUS0 写 SRAM Block 1 的权限。(读/写)

PMS_PRO_DRAM0_SRAM_2_R 设置为 1 给予 DBUS0 读 SRAM Block 2 的权限。(读/写)

PMS_PRO_DRAM0_SRAM_2_W 设置为 1 给予 DBUS0 写 SRAM Block 2 的权限。(读/写)

PMS_PRO_DRAM0_SRAM_3_R 设置为 1 给予 DBUS0 读 SRAM Block 3 的权限。(读/写)

PMS_PRO_DRAM0_SRAM_3_W 设置为 1 给予 DBUS0 写 SRAM Block 3 的权限。(读/写)

PMS_PRO_DRAM0_SRAM_4_SPLTADDR 配置 DBUS0 访问 SRAM Block 4-21 的分割地址。(读/写)

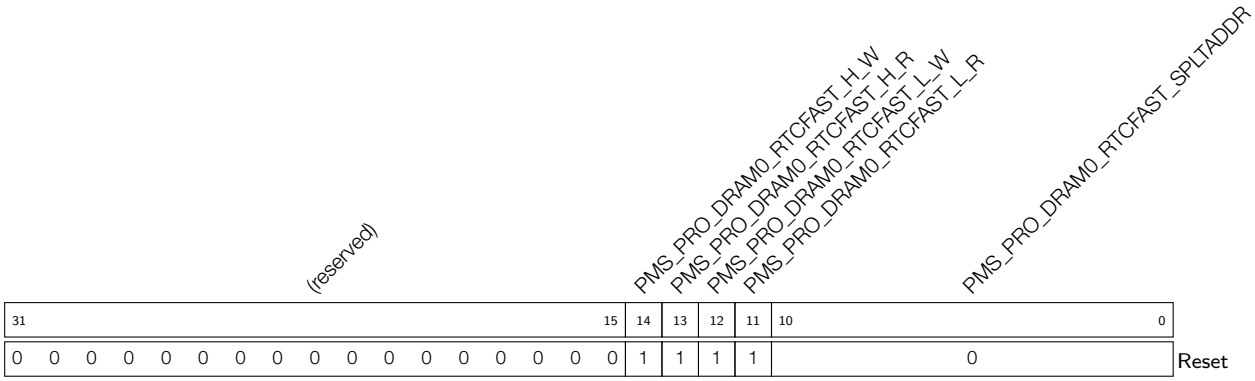
PMS_PRO_DRAM0_SRAM_4_L_R 设置为 1 给予 DBUS0 读 SRAM Block 4-21 低地址区域的权限。(读/写)

PMS_PRO_DRAM0_SRAM_4_L_W 设置为 1 给予 DBUS0 写 SRAM Block 4-21 低地址区域的权限。(读/写)

PMS_PRO_DRAM0_SRAM_4_H_R 设置为 1 给予 DBUS0 读 SRAM Block 4-21 高地址区域的权限。(读/写)

PMS_PRO_DRAM0_SRAM_4_H_W 设置为 1 给予 DBUS0 写 SRAM Block 4-21 高地址区域的权限。(读/写)

Register 24.20: PMS_PRO_DRAM0_2_REG (0x0030)



PMS_PRO_DRAM0_RTCFAST_SPLTADDR 配置 DBUS0 访问 RTC FAST Memory 的分割地址。(读/写)

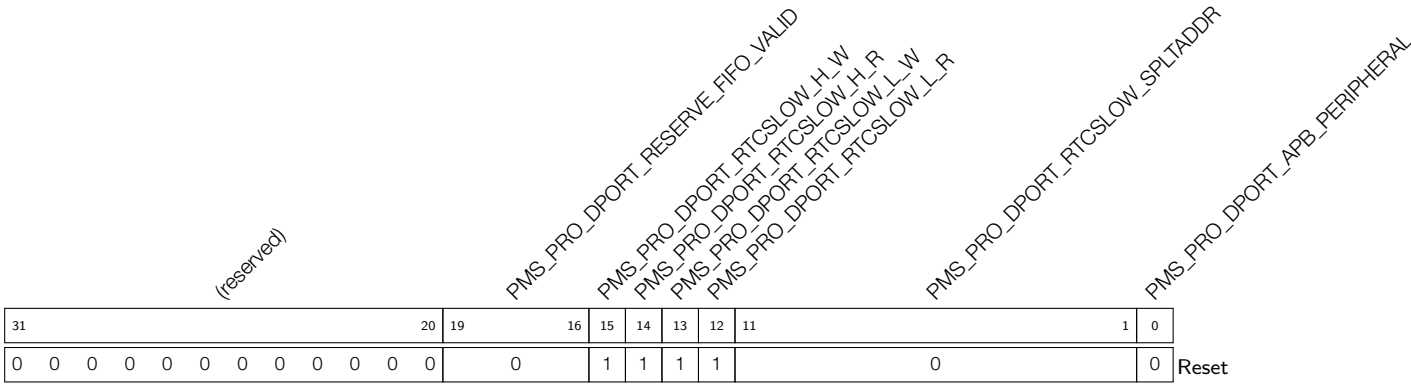
PMS_PRO_DRAM0_RTCFAST_L_R 设置为 1 给予 DBUS0 读 RTC FAST Memory 低地址区域的权限。(读/写)

PMS_PRO_DRAM0_RTCFAST_L_W 设置为 1 给予 DBUS0 写 RTC FAST Memory 低地址区域的权限。(读/写)

PMS_PRO_DRAM0_RTCFAST_H_R 设置为 1 给予 DBUS0 读 RTC FAST Memory 高地址区域的权限。(读/写)

PMS_PRO_DRAM0_RTCFAST_H_W 设置为 1 给予 DBUS0 写 RTC FAST Memory 高地址区域的权限。(读/写)

Register 24.21: PMS_PRO_DPORT_1_REG (0x0040)



PMS_PRO_DPORT_APB_PERIPHERAL_FORBID 设置为 1 将拒绝 PeriBus1 总线对 APB 外设的访问。

PMS_PRO_DPORT_RTCSLOW_SPLTADDR 配置 PeriBus1 访问 RTC FAST Memory 的分割地址。(读/写)

PMS_PRO_DPORT_RTCSLOW_L_R 设置为 1 给予 PeriBus1 读 RTC FAST Memory 低地址区域的权限。(读/写)

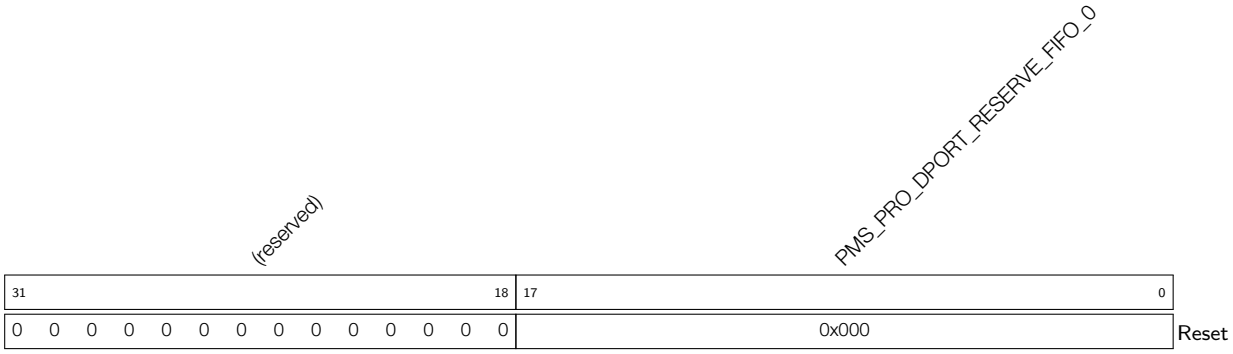
PMS_PRO_DPORT_RTCSLOW_L_W 设置为 1 给予 PeriBus1 写 RTC FAST Memory 低地址区域的权限。(读/写)

PMS_PRO_DPORT_RTCSLOW_H_R 设置为 1 给予 PeriBus1 读 RTC FAST Memory 高地址区域的权限。(读/写)

PMS_PRO_DPORT_RTCSLOW_H_W 设置为 1 给予 PeriBus1 写 RTC FAST Memory 高地址区域的权限。(读/写)

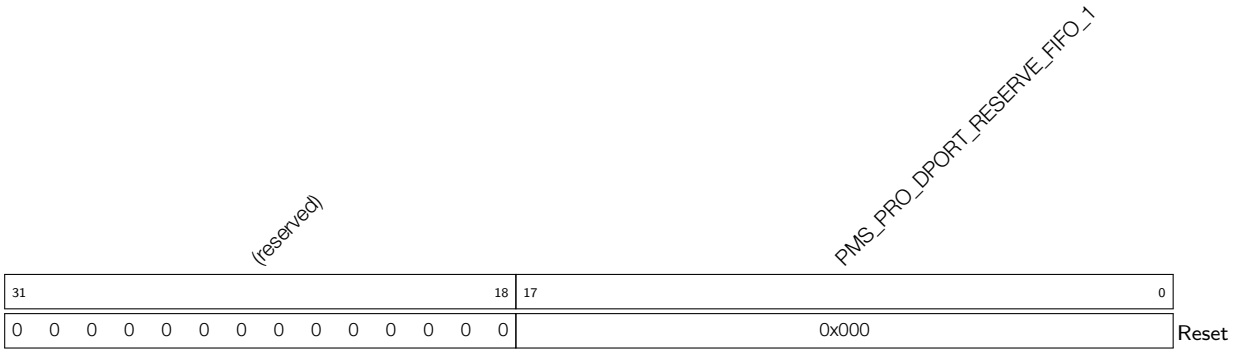
PMS_PRO_DPORT_RESERVE_FIFO_VALID 配置其否对用户配置的 FIFO 地址启用读保护。(读/写)

Register 24.22: PMS_PRO_DPORT_2_REG (0x0044)



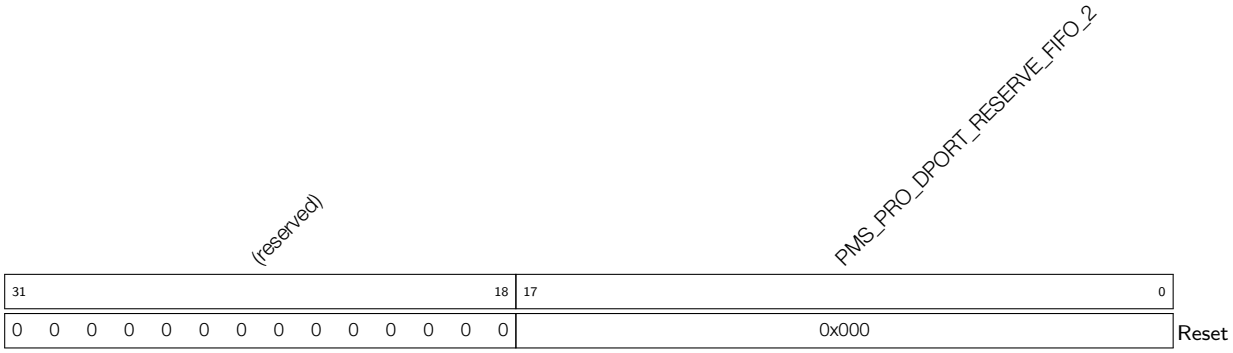
PMS_PRO_DPORT_RESERVE_FIFO_0 配置读保护地址 0。(读/写)

Register 24.23: PMS_PRO_DPORT_3_REG (0x0048)



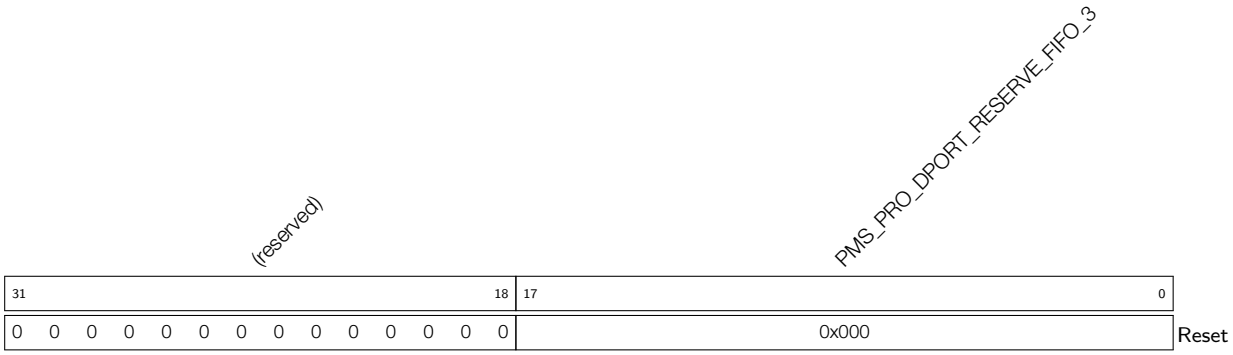
PMS_PRO_DPORT_RESERVE_FIFO_1 配置读保护地址 1。(读/写)

Register 24.24: PMS_PRO_DPORT_4_REG (0x004C)



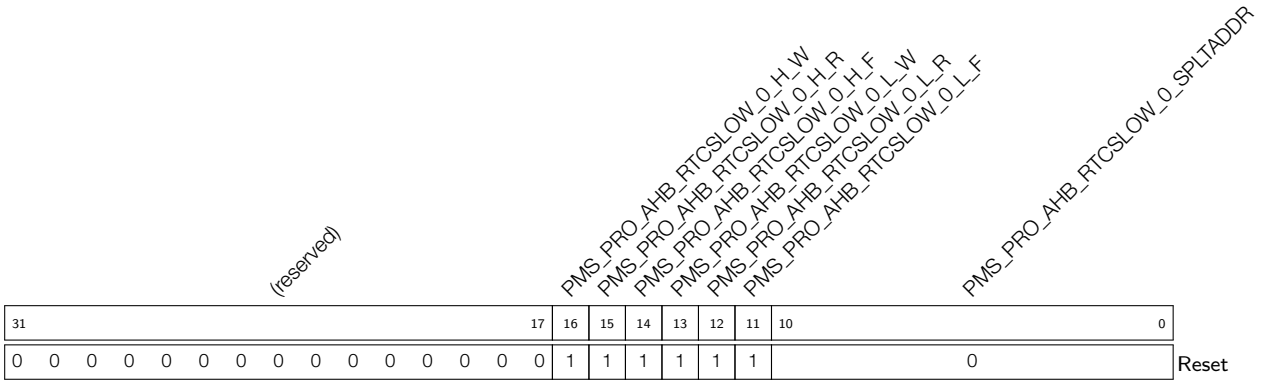
PMS_PRO_DPORT_RESERVE_FIFO_2 配置读保护地址 2。(读/写)

Register 24.25: PMS_PRO_DPORT_5_REG (0x0050)



PMS_PRO_DPORT_RESERVE_FIFO_3 配置读保护地址 3。(读/写)

Register 24.26: PMS_PRO_AHB_1_REG (0x0060)



PMS_PRO_AHB_RTCSLOW_0_SPLTADDR 配置 PeriBus2 访问 RTCSlow_0 的分割地址。(读/写)

PMS_PRO_AHB_RTCSLOW_0_L_F 设置为 1 给予 PeriBus2 对 RTCSlow_0 低地址区域进行取指操作的权限。(读/写)

PMS_PRO_AHB_RTCSLOW_0_L_R 设置为 1 给予 PeriBus2 读 RTCSlow_0 低地址区域的权限。(读/写)

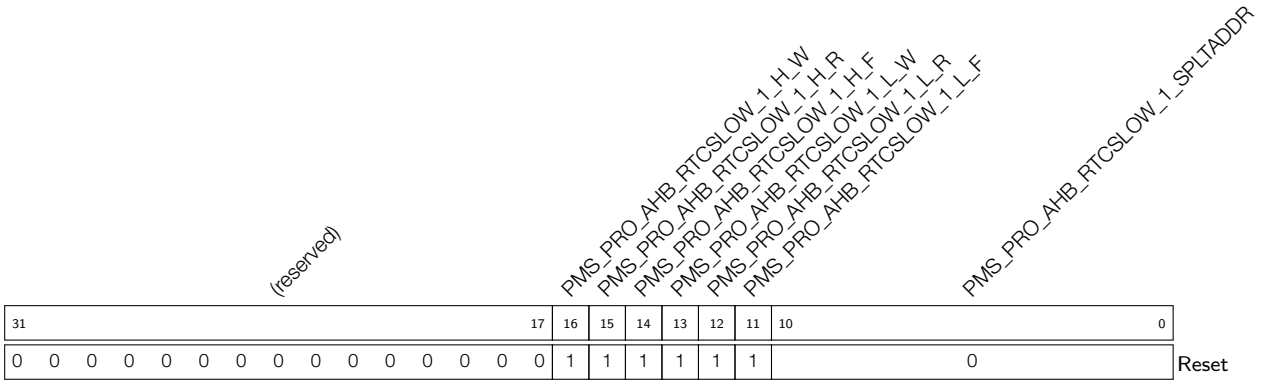
PMS_PRO_AHB_RTCSLOW_0_L_W 设置为 1 给予 PeriBus2 写 RTCSlow_0 低地址区域的权限。(读/写)

PMS_PRO_AHB_RTCSLOW_0_H_F 设置为 1 给予 PeriBus2 对 RTCSlow_0 高地址区域进行取指操作的权限。(读/写)

PMS_PRO_AHB_RTCSLOW_0_H_R 设置为 1 给予 PeriBus2 读 RTCSlow_0 高地址区域的权限。(读/写)

PMS_PRO_AHB_RTCSLOW_0_H_W 设置为 1 给予 PeriBus2 写 RTCSlow_0 高地址区域的权限。(读/写)

Register 24.27: PMS_PRO_AHB_2_REG (0x0064)



PMS_PRO_AHB_RTCSLOW_1_SPLTADDR 配置 PeriBus2 访问 RTCSlow_1 的分割地址。(读/写)

PMS_PRO_AHB_RTCSLOW_1_L_F 设置为 1 给予 PeriBus2 对 RTCSlow_1 低地址区域进行取指操作的权限。(读/写)

PMS_PRO_AHB_RTCSLOW_1_L_R 设置为 1 给予 PeriBus2 读 RTCSlow_1 低地址区域的权限。(读/写)

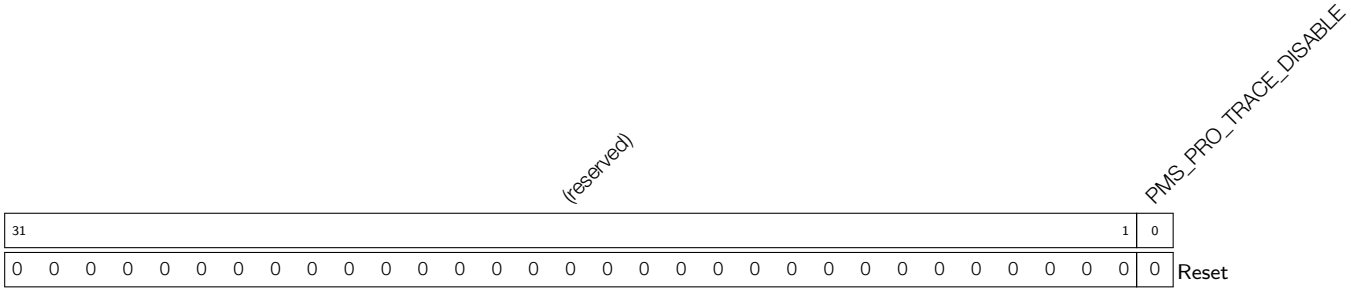
PMS_PRO_AHB_RTCSLOW_1_L_W 设置为 1 给予 PeriBus2 写 RTCSlow_1 低地址区域的权限。(读/写)

PMS_PRO_AHB_RTCSLOW_1_H_F 设置为 1 给予 PeriBus2 对 RTCSlow_1 高地址区域进行取指操作的权限。(读/写)

PMS_PRO_AHB_RTCSLOW_1_H_R 设置为 1 给予 PeriBus2 读 RTCSlow_1 高地址区域的权限。(读/写)

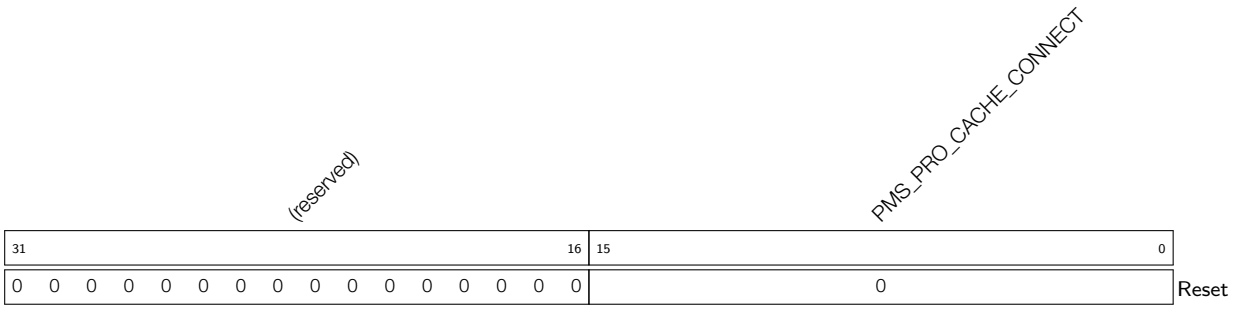
PMS_PRO_AHB_RTCSLOW_1_H_W 设置为 1 给予 PeriBus2 写 RTCSlow_1 高地址区域的权限。(读/写)

Register 24.28: PMS_PRO_TRACE_1_REG (0x0074)



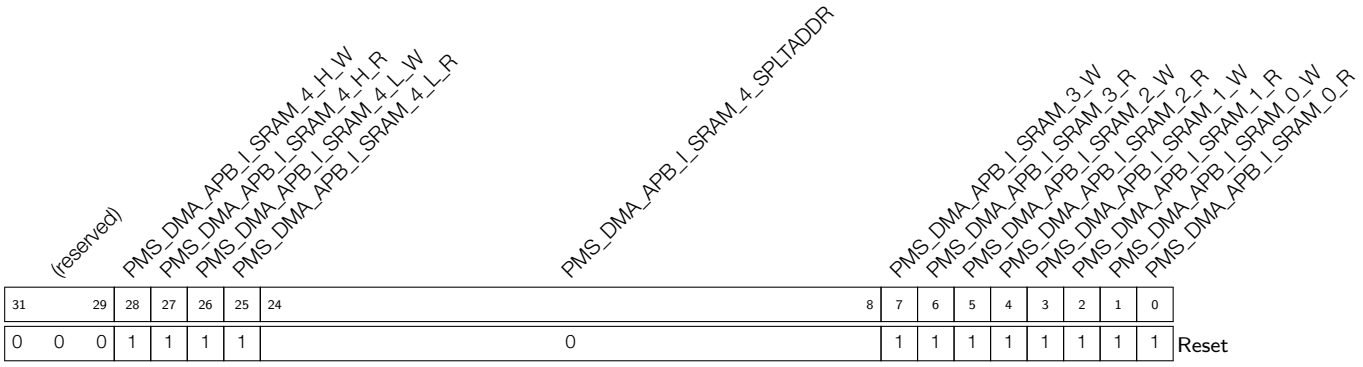
PMS_PRO_TRACE_DISABLE 设置为 1 禁用 Trace memory 功能。(读/写)

Register 24.29: PMS_PRO_CACHE_1_REG (0x007C)



PMS_PRO_CACHE_CONNECT 配置 Icache 或 Dcache 将占用 SRAM block 0-3 中的哪一个。(读/写)

Register 24.30: PMS_DMA_APB_I_1_REG (0x0090)



PMS_DMA_APB_I_SRAM_0_R 设置为 1 给予内部 DMA 读 SRAM Block 0 的权限。(读/写)

PMS_DMA_APB_I_SRAM_0_W 设置为 1 给予内部 DMA 写 SRAM Block 0 的权限。(读/写)

PMS_DMA_APB_I_SRAM_1_R 设置为 1 给予内部 DMA 读 SRAM Block 1 的权限。(读/写)

PMS_DMA_APB_I_SRAM_1_W 设置为 1 给予内部 DMA 写 SRAM Block 1 的权限。(读/写)

PMS_DMA_APB_I_SRAM_2_R 设置为 1 给予内部 DMA 读 SRAM Block 2 的权限。(读/写)

PMS_DMA_APB_I_SRAM_2_W 设置为 1 给予内部 DMA 写 SRAM Block 2 的权限。(读/写)

PMS_DMA_APB_I_SRAM_3_R 设置为 1 给予内部 DMA 读 SRAM Block 3 的权限。(读/写)

PMS_DMA_APB_I_SRAM_3_W 设置为 1 给予内部 DMA 写 SRAM Block 3 的权限。(读/写)

PMS_DMA_APB_I_SRAM_4_SPLTADDR 配置内部 DMA 访问 SRAM Block 4-21 的分割地址。(读/写)

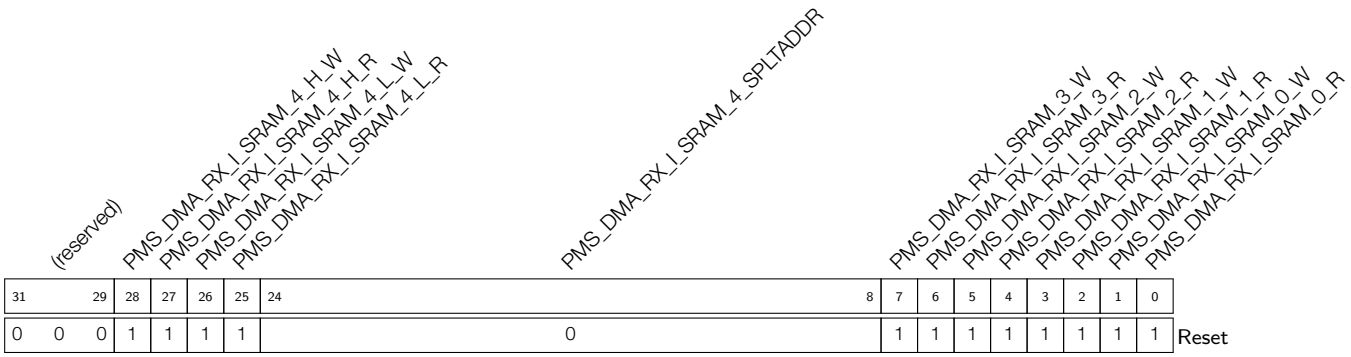
PMS_DMA_APB_I_SRAM_4_L_R 设置为 1 给予内部 DMA 读 SRAM Block 4-21 低地址区域的权限。(读/写)

PMS_DMA_APB_I_SRAM_4_L_W 设置为 1 给予内部 DMA 写 SRAM Block 4-21 低地址区域的权限。(读/写)

PMS_DMA_APB_I_SRAM_4_H_R 设置为 1 给予内部 DMA 读 SRAM Block 4-21 高地址区域的权限。(读/写)

PMS_DMA_APB_I_SRAM_4_H_W 设置为 1 给予内部 DMA 写 SRAM Block 4-21 高地址区域的权限。(读/写)

Register 24.31: PMS_DMA_RX_I_1_REG (0x00A0)



PMS_DMA_RX_I_SRAM_0_R 设置为 1 给予 RX Copy DMA 读 SRAM Block 0 的权限。(读/写)

PMS_DMA_RX_I_SRAM_0_W 设置为 1 给予 RX Copy DMA 写 SRAM Block 0 的权限。(读/写)

PMS_DMA_RX_I_SRAM_1_R 设置为 1 给予 RX Copy DMA 读 SRAM Block 1 的权限。(读/写)

PMS_DMA_RX_I_SRAM_1_W 设置为 1 给予 RX Copy DMA 写 SRAM Block 1 的权限。(读/写)

PMS_DMA_RX_I_SRAM_2_R 设置为 1 给予 RX Copy DMA 读 SRAM Block 2 的权限。(读/写)

PMS_DMA_RX_I_SRAM_2_W 设置为 1 给予 RX Copy DMA 写 SRAM Block 2 的权限。(读/写)

PMS_DMA_RX_I_SRAM_3_R 设置为 1 给予 RX Copy DMA 读 SRAM Block 3 的权限。(读/写)

PMS_DMA_RX_I_SRAM_3_W 设置为 1 给予 RX Copy DMA 写 SRAM Block 3 的权限。(读/写)

PMS_DMA_RX_I_SRAM_4_SPLTADDR 配置 RX Copy DMA 访问 SRAM Block 4-21 的分割地址。(读/写)

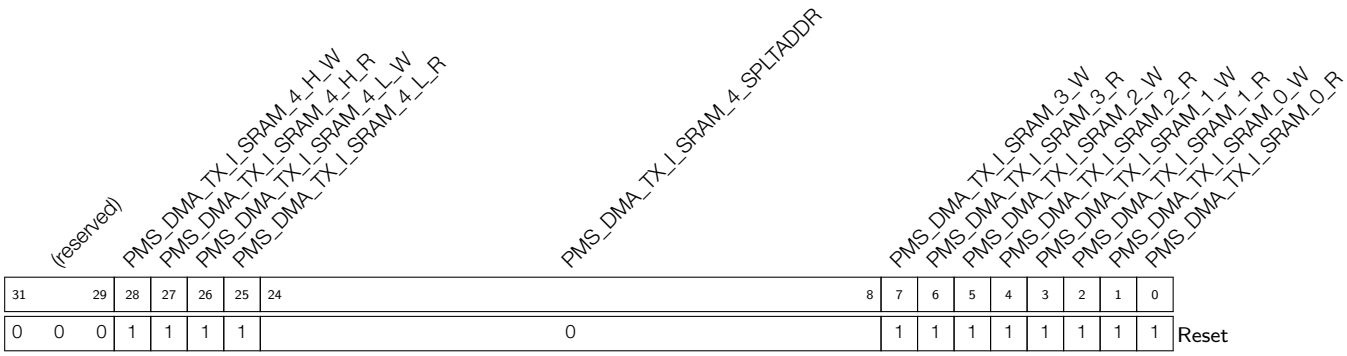
PMS_DMA_RX_I_SRAM_4_L_R 设置为 1 给予 RX Copy DMA 读 SRAM Block 4-21 低地址区域的权限。(读/写)

PMS_DMA_RX_I_SRAM_4_L_W 设置为 1 给予 RX Copy DMA 写 SRAM Block 4-21 低地址区域的权限。(读/写)

PMS_DMA_RX_I_SRAM_4_H_R 设置为 1 给予 RX Copy DMA 读 SRAM Block 4-21 高地址区域的权限。(读/写)

PMS_DMA_RX_I_SRAM_4_H_W 设置为 1 给予 RX Copy DMA 写 SRAM Block 4-21 高地址区域的权限。(读/写)

Register 24.32: PMS_DMA_TX_I_1_REG (0x00B0)



PMS_DMA_TX_I_SRAM_0_R 设置为 1 给予 TX Copy DMA 读 SRAM Block 0 的权限。(读/写)

PMS_DMA_TX_I_SRAM_0_W 设置为 1 给予 TX Copy DMA 写 SRAM Block 0 的权限。(读/写)

PMS_DMA_TX_I_SRAM_1_R 设置为 1 给予 TX Copy DMA 读 SRAM Block 1 的权限。(读/写)

PMS_DMA_TX_I_SRAM_1_W 设置为 1 给予 TX Copy DMA 写 SRAM Block 1 的权限。(读/写)

PMS_DMA_TX_I_SRAM_2_R 设置为 1 给予 TX Copy DMA 读 SRAM Block 2 的权限。(读/写)

PMS_DMA_TX_I_SRAM_2_W 设置为 1 给予 TX Copy DMA 写 SRAM Block 2 的权限。(读/写)

PMS_DMA_TX_I_SRAM_3_R 设置为 1 给予 TX Copy DMA 读 SRAM Block 3 的权限。(读/写)

PMS_DMA_TX_I_SRAM_3_W 设置为 1 给予 TX Copy DMA 写 SRAM Block 3 的权限。(读/写)

PMS_DMA_TX_I_SRAM_4_SPLTADDR 配置 TX Copy DMA 访问 SRAM Block 4-21 的分割地址。(读/写)

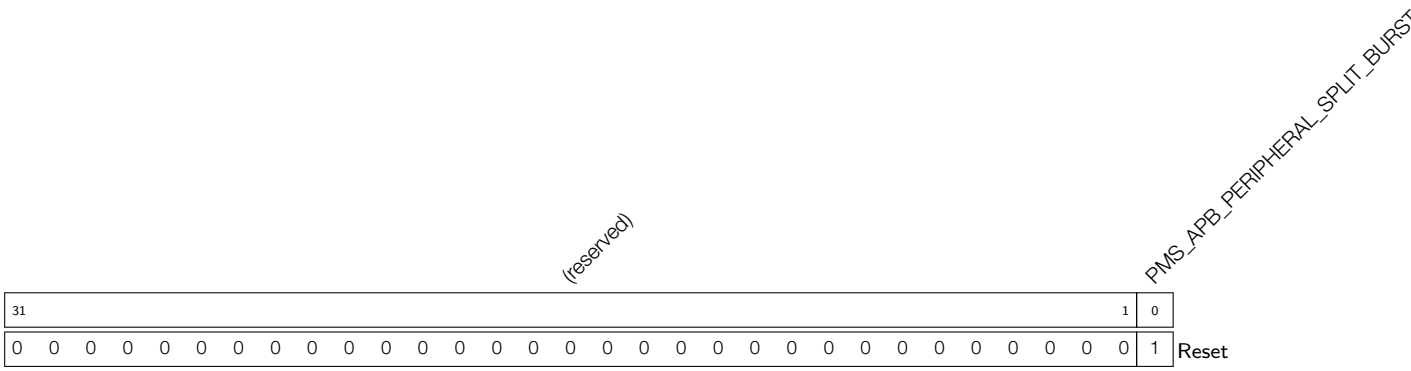
PMS_DMA_TX_I_SRAM_4_L_R 设置为 1 给予 TX Copy DMA 读 SRAM Block 4-21 低地址区域的权限。(读/写)

PMS_DMA_TX_I_SRAM_4_L_W 设置为 1 给予 TX Copy DMA 写 SRAM Block 4-21 低地址区域的权限。(读/写)

PMS_DMA_TX_I_SRAM_4_H_R 设置为 1 给予 TX Copy DMA 读 SRAM Block 4-21 高地址区域的权限。(读/写)

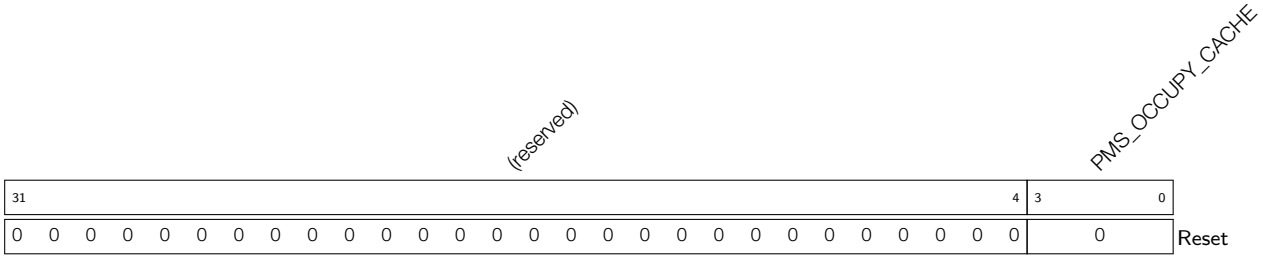
PMS_DMA_TX_I_SRAM_4_H_W 设置为 1 给予 TX Copy DMA 写 SRAM Block 4-21 高地址区域的权限。(读/写)

Register 24.33: PMS_APB_PERIPHERAL_1_REG (0x00D0)



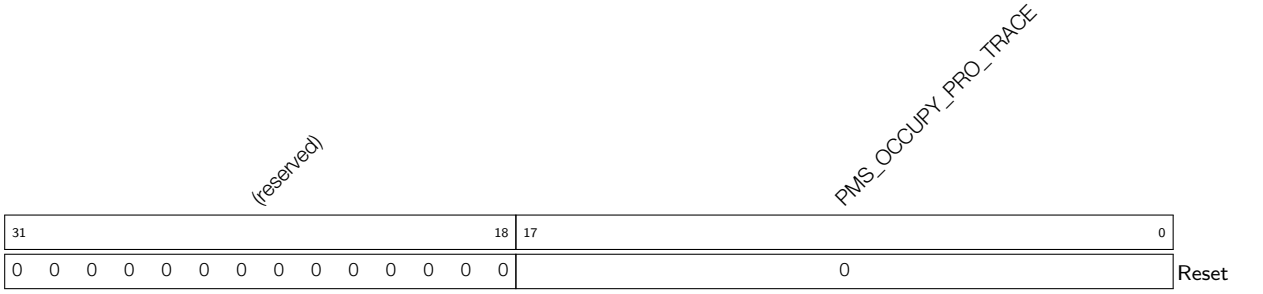
PMS_APB_PERIPHERAL_SPLIT_BURST 设置为 1 将上次访问的数据阶段和后续访问的地址阶段分开。(读/写)

Register 24.34: PMS_OCCUPY_1_REG (0x00D8)



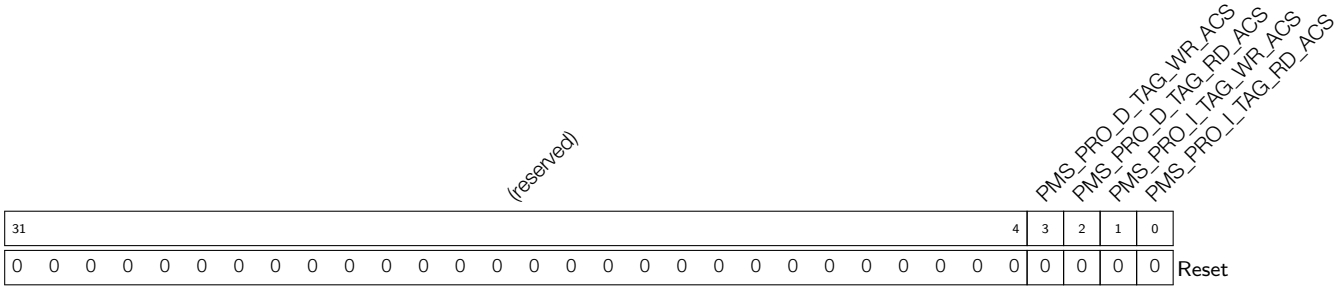
PMS_OCCUPY_CACHE 配置是否将 SRAM Block 0-3 用作 cache。(读/写)

Register 24.35: PMS_OCCUPY_3_REG (0x00E0)



PMS_OCCUPY_PRO_TRACE 配置 SRAM block 4-21 中的一个 block 用作 Trace memory。(读/写)

Register 24.36: PMS_CACHE_TAG_ACCESS_1_REG (0x00E8)



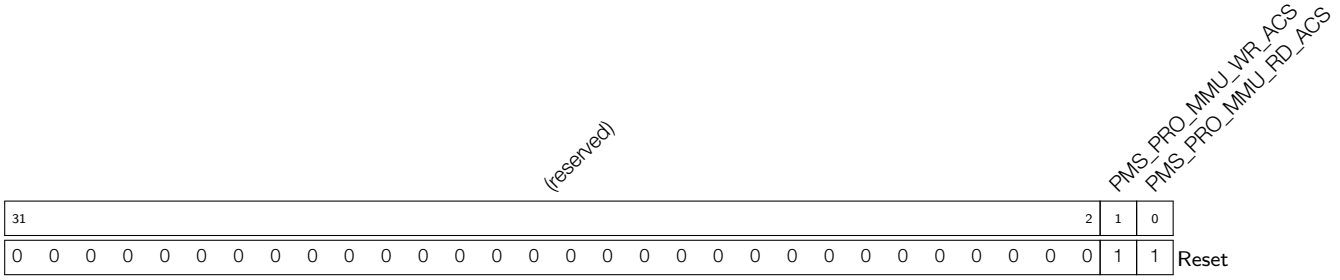
PMS_PRO_I_TAG_RD_ACS 设置为 1 允许对 Icache 标签存储器进行读访问。(读/写)

PMS_PRO_I_TAG_WR_ACS 设置为 1 允许对 Icache 标签存储器进行写访问。(读/写)

PMS_PRO_D_TAG_RD_ACS 设置为 1 允许对 Dcache 标签存储器进行读访问。(读/写)

PMS_PRO_D_TAG_WR_ACS 设置为 1 允许对 Dcache 标签存储器进行写访问。(读/写)

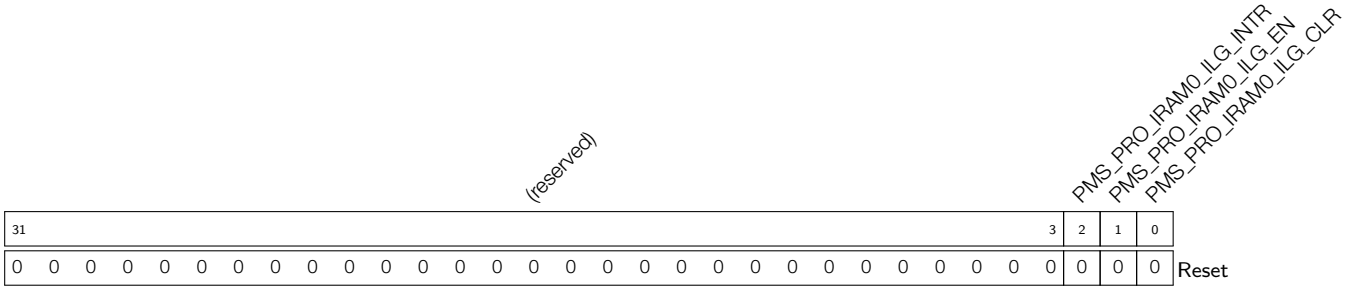
Register 24.37: PMS_CACHE_MMU_ACCESS_1_REG (0x00F0)



PMS_PRO_MMU_RD_ACS 设置为 1 允许对 MMU 进行读访问。(读/写))

PMS_PRO_MMU_WR_ACS 设置为 1 允许对 MMU 进行写访问。(读/写)

Register 24.38: PMS_PRO_IRAM0_4_REG (0x0020)

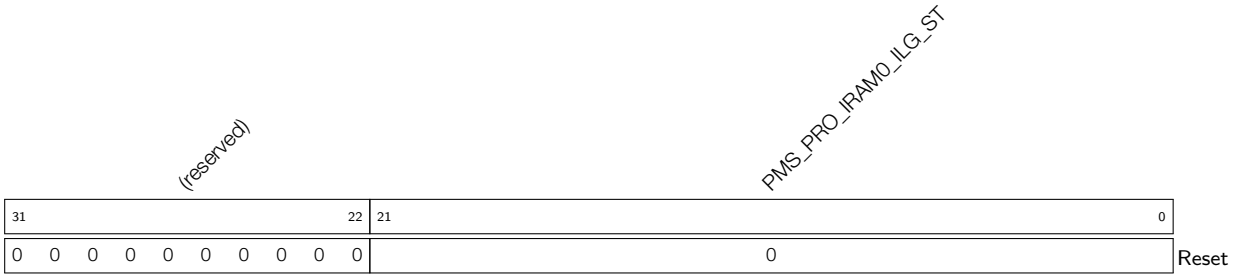


PMS_PRO_IRAM0_ILG_CLR 访问中断的清除信号。(读/写)

PMS_PRO_IRAM0_ILG_EN IBUS 访问中断的使能信号。(读/写)

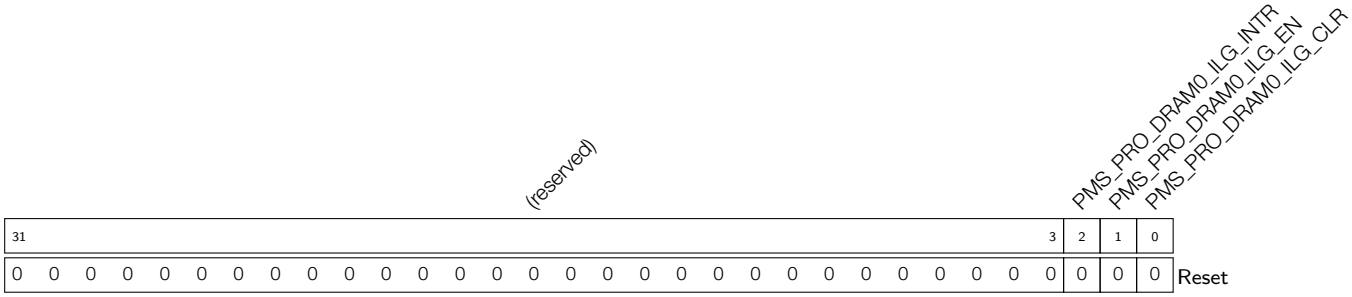
PMS_PRO_IRAM0_ILG_INTR IBUS 访问中断信号。(只读)

Register 24.39: PMS_PRO_IRAM0_5_REG (0x0024)



PMS_PRO_IRAM0_ILG_ST 记录 IBUS 访问的非法信息。[21:2]：存储 IBUS 地址的位 [21:2]；[1]：1 代表数据访问，0 代表取指访问；[0]：1 代表读操作，0 代表写操作。（只读）

Register 24.40: PMS_PRO_DRAM0_3_REG (0x0034)

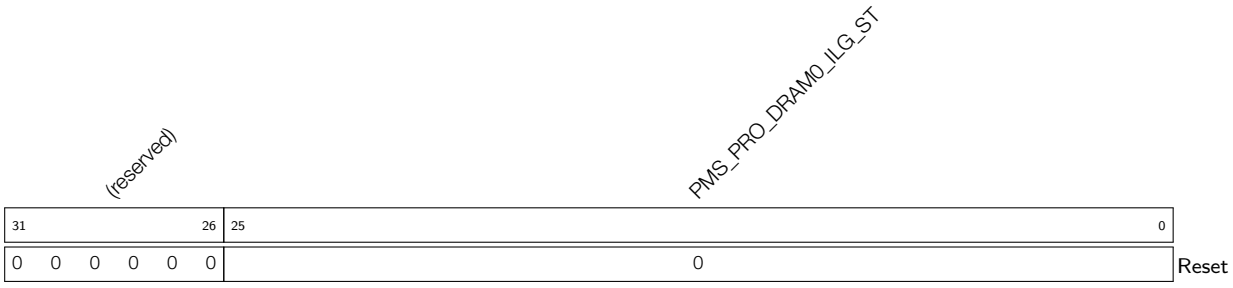


PMS_PRO_DRAM0_ILG_CLR DBUS0 访问中断的清除信号。（读/写）

PMS_PRO_DRAM0_ILG_EN DBUS0 访问中断的使能信号。（读/写）

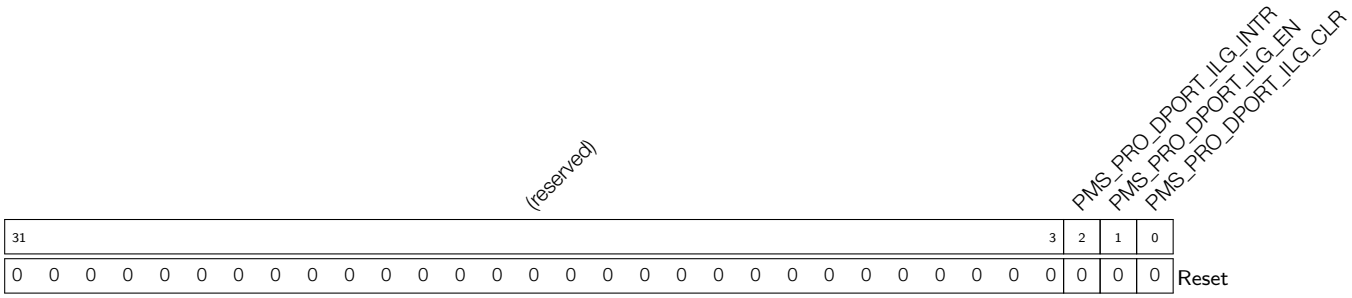
PMS_PRO_DRAM0_ILG_INTR DBUS0 访问中断信号。（只读）

Register 24.41: PMS_PRO_DRAM0_4_REG (0x0038)



PMS_PRO_DRAM0_ILG_ST 记录 DBUS0 访问的非法信息。[25:6]：存储 DBUS 地址的位 [21:2]；[5]：1 代表原子访问，0 代表非原子访问；[4]：1 代表读操作，0 代表写操作；[3:0]：DBUS0 总线字节使能 (byte enables)。（只读）

Register 24.42: PMS_PRO_DPORT_6_REG (0x0054)

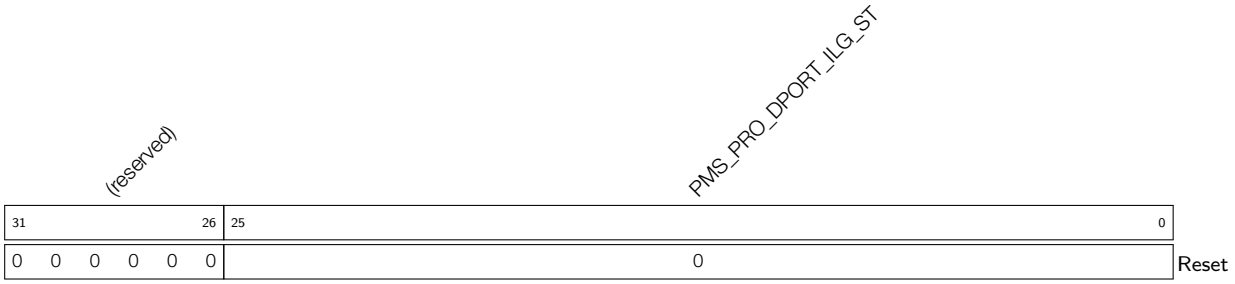


PMS_PRO_DPORT_ILG_CLR PeriBus1 访问中断的清除信号。(读/写)

PMS_PRO_DPORT_ILG_EN PeriBus1 访问中断的使能信号。(读/写)

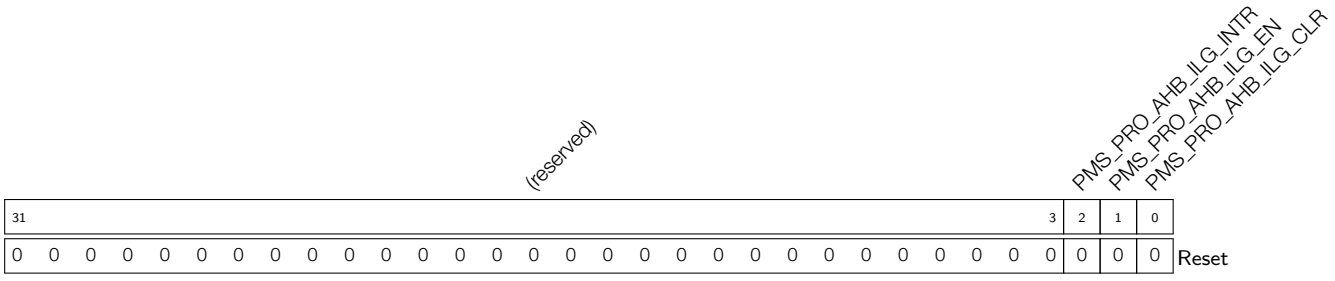
PMS_PRO_DPORT_ILG_INTR PeriBus1 访问中断信号。(只读)

Register 24.43: PMS_PRO_DPORT_7_REG (0x0058)



PMS_PRO_DPORT_ILG_ST 记录 PeriBus1 访问的非法信息。[25:6]：存储 PeriBus1 地址的位 [21:2]；[5]：1 代表原子访问，0 代表非原子访问；[4]：如果 PeriBus1 地址的位 [31:22] 为 0xfd，则为 1，否则为 0；[3:0]：PeriBus1 总线字节使能。(只读)

Register 24.44: PMS_PRO_AHB_3_REG (0x0068)

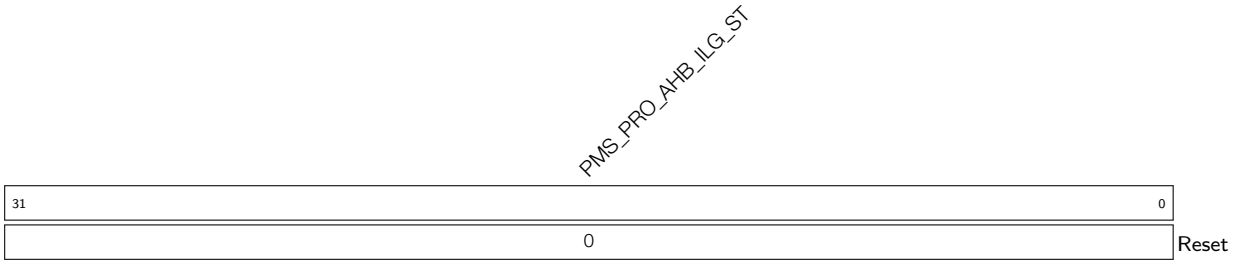


PMS_PRO_AHB_ILG_CLR PeriBus2 访问中断的清除信号。(读/写)

PMS_PRO_AHB_ILG_EN PeriBus2 访问中断的使能信号。(读/写)

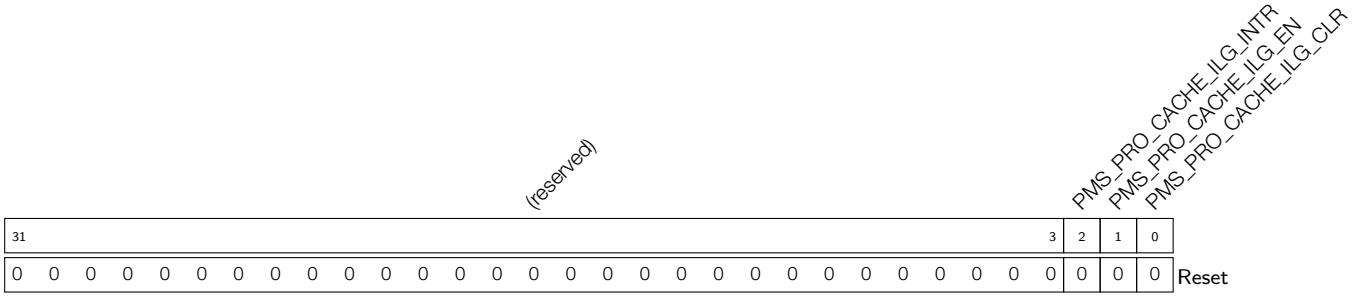
PMS_PRO_AHB_ILG_INTR PeriBus2 访问中断信号。(只读)

Register 24.45: PMS_PRO_AHB_4_REG (0x006C)



PMS_PRO_AHB_ILG_ST 记录 PeriBus2 访问的非法信息。[31:2]：存储 PeriBus2 地址的位 [31:2]；[1]：1 代表数据访问，0 代表取指访问；[0]：1 代表读操作，0 代表写操作。（只读）

Register 24.46: PMS_PRO_CACHE_2_REG (0x0080)

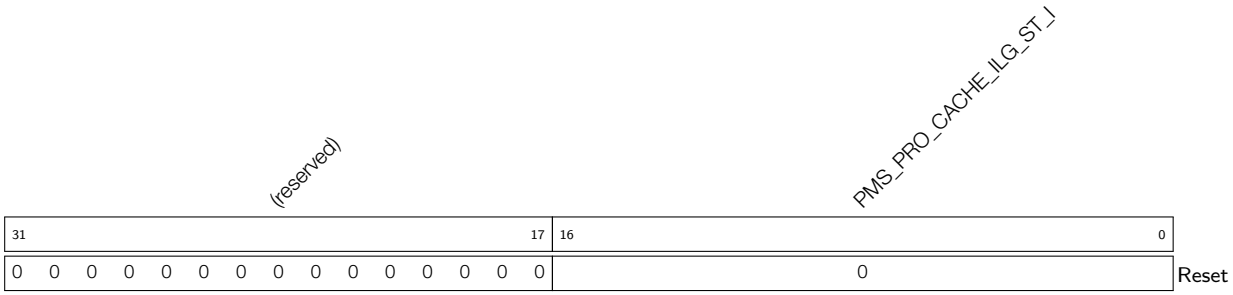


PMS_PRO_CACHE_ILG_CLR Cache 访问中断的清除信号。（读/写）

PMS_PRO_CACHE_ILG_EN Cache 访问中断的使能信号。（读/写）

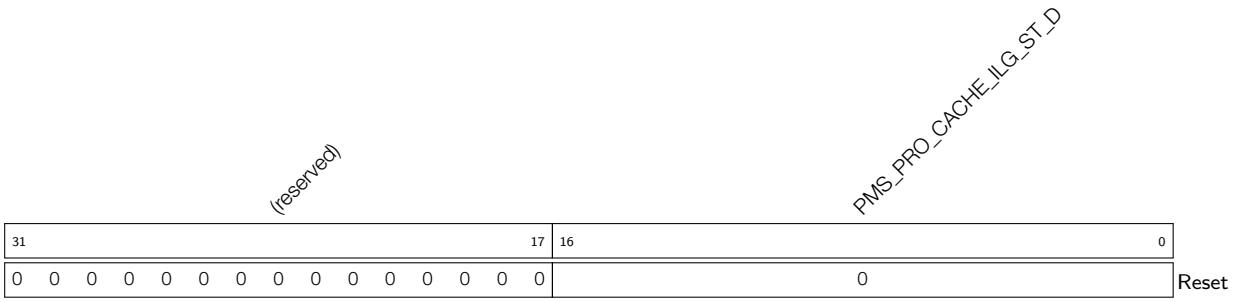
PMS_PRO_CACHE_ILG_INTR Cache 访问中断信号。（只读）

Register 24.47: PMS_PRO_CACHE_3_REG (0x0084)



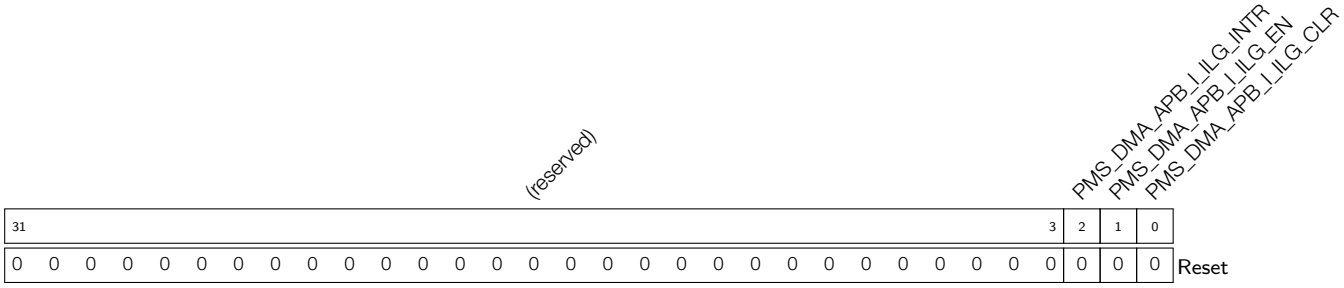
PMS_PRO_CACHE_ILG_ST_I 记录 Icache 访问的非法信息。[16]：访问使能，低电平有效；[15:4]：存储地址位 [11:0]；[3:0]：Icache 总线字节使能，低电平有效。（只读）

Register 24.48: PMS_PRO_CACHE_4_REG (0x0088)



PMS_PRO_CACHE_ILG_ST_D 记录 Dcache 访问的非法信息。[16]：访问使能，低电平有效；[15:4]：存储地址位 [11:0]；[3:0]：Dcache 总线字节使能，低电平有效。（只读）

Register 24.49: PMS_DMA_APB_I_2_REG (0x0094)

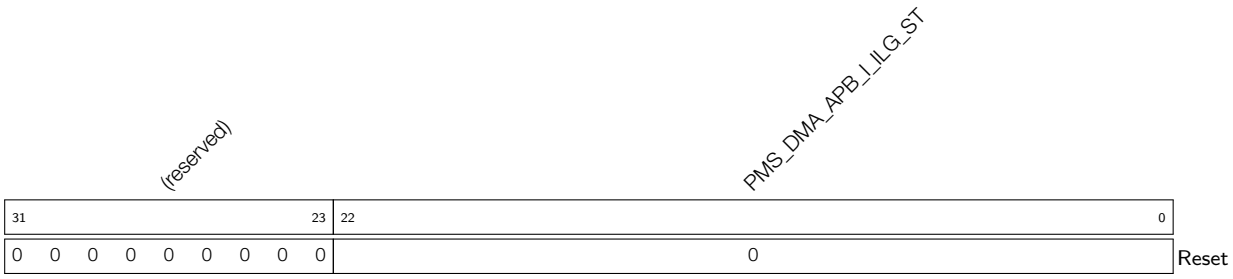


PMS_DMA_APB_I_ILG_CLR 内部 DMA 访问中断的清除信号。（读/写）

PMS_DMA_APB_I_ILG_EN 内部 DMA 访问中断的使能信号。（读/写）

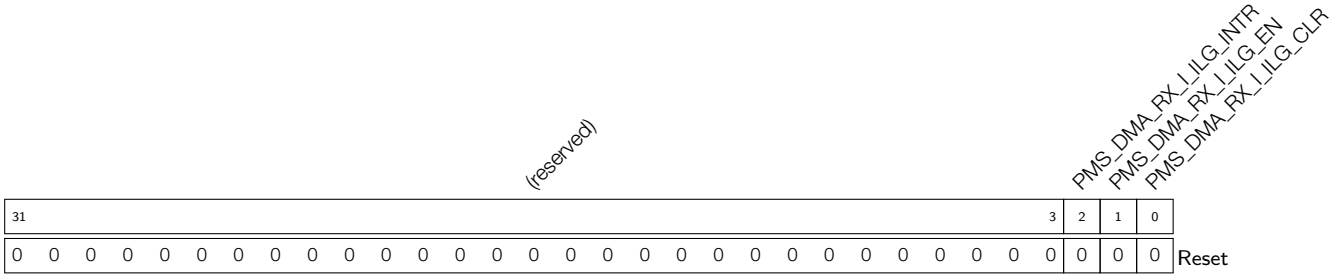
PMS_DMA_APB_I_ILG_INTR 内部 DMA 访问中断信号。（只读）

Register 24.50: PMS_DMA_APB_I_3_REG (0x0098)



PMS_DMA_APB_I_ILG_ST 记录内部 DMA 访问的非法信息。[22:6]：存储地址位 [18:2]；[5]：如果地址位 [31:19] 为 0x7ff，则值为 1，否则为 0；[4]：1 代表写操作，0 代表读操作；[3:0]：内部 DMA 总线字节使能。（只读）

Register 24.51: PMS_DMA_RX_I_2_REG (0x00A4)

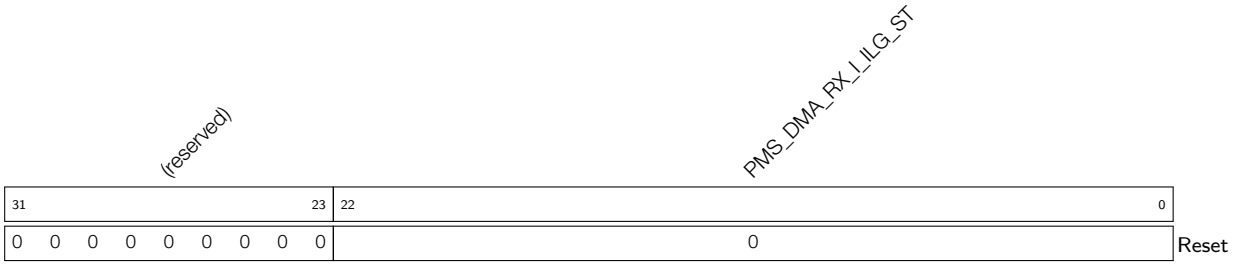


PMS_DMA_RX_I_ILG_CLR RX Copy DMA 访问中断的清除信号。(读/写)

PMS_DMA_RX_I_ILG_EN RX Copy DMA 访问中断的使能信号。(读/写)

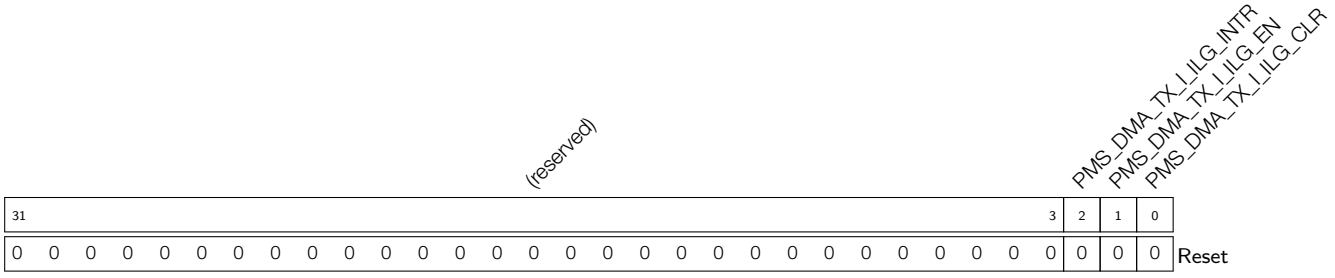
PMS_DMA_RX_I_ILG_INTR RX Copy DMA 访问中断信号。(只读)

Register 24.52: PMS_DMA_RX_I_3_REG (0x00A8)



PMS_DMA_RX_I_ILG_ST 记录 RX Copy DMA 访问的非法信息。[22:6]：存储地址位 [18:2]；[5]：如果地址位 [31:19] 为 0x7ff，则值为 1，否则为 0；[4]：1 代表写操作，0 代表读操作；[3:0]：RX Copy DMA 总线字节使能。(只读)

Register 24.53: PMS_DMA_TX_I_2_REG (0x00B4)

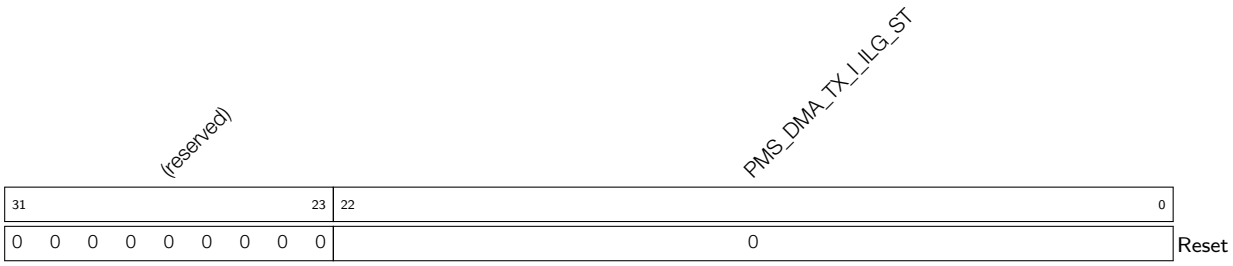


PMS_DMA_TX_I_ILG_CLR TX Copy DMA 访问中断的清除信号。(读/写)

PMS_DMA_TX_I_ILG_EN TX Copy DMA 访问中断的使能信号。(读/写)

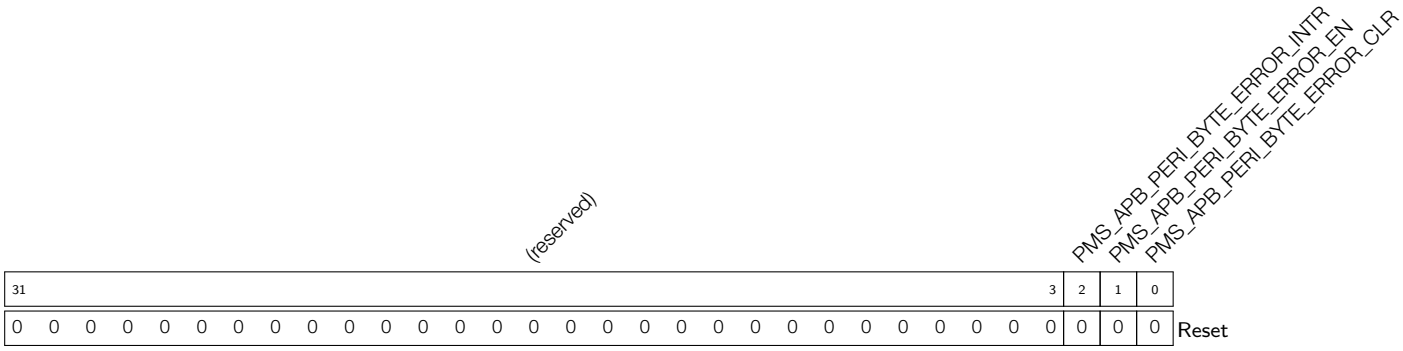
PMS_DMA_TX_I_ILG_INTR TX Copy DMA 访问中断信号。(只读)

Register 24.54: PMS_DMA_TX_I_3_REG (0x00B8)



PMS_DMA_TX_I_ILG_ST 记录 TX Copy DMA 访问的非法信息。[22:6]：存储地址位 [18:2]；[5]：如果地址位 [31:19] 为 0x7ff，则值为 1，否则为 0；[4]：1 代表写操作，0 代表读操作；[3:0]：TX Copy DMA 总线字节使能。（只读）

Register 24.55: PMS_APB_PERIPHERAL_INTR_REG (0x00F4)

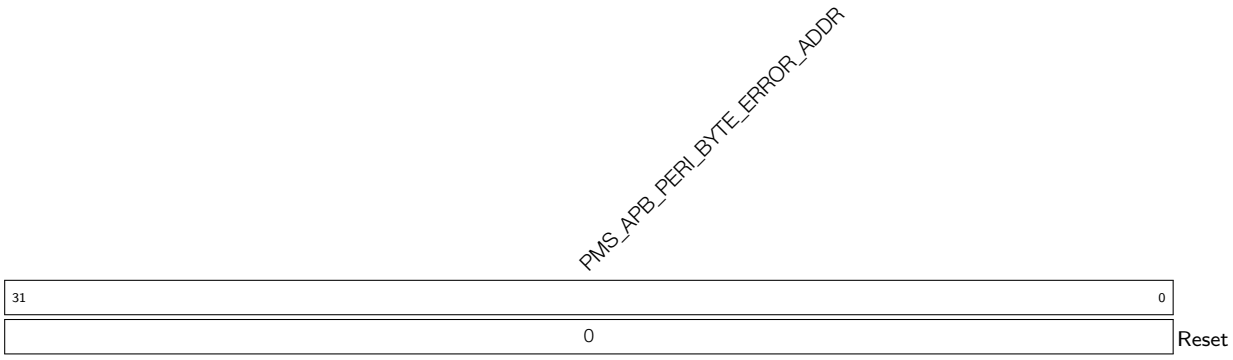


PMS_APB_PERI_BYTE_ERROR_CLR APB 外设访问中断的清除信号。（读/写）

PMS_APB_PERI_BYTE_ERROR_EN APB 外设访问中断的使能信号。（读/写）

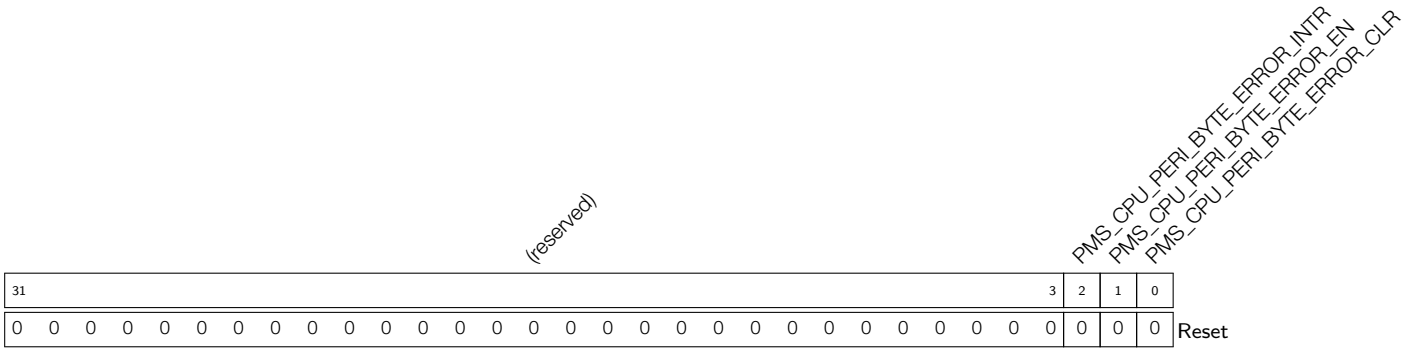
PMS_APB_PERI_BYTE_ERROR_INTR APB 外设访问中断信号。（只读）

Register 24.56: PMS_APB_PERIPHERAL_STATUS_REG (0x00F8)



PMS_APB_PERI_BYTE_ERROR_ADDR 记录 APB 外设访问的非法地址。

Register 24.57: PMS_CPU_PERIPHERAL_INTR_REG (0x00FC)

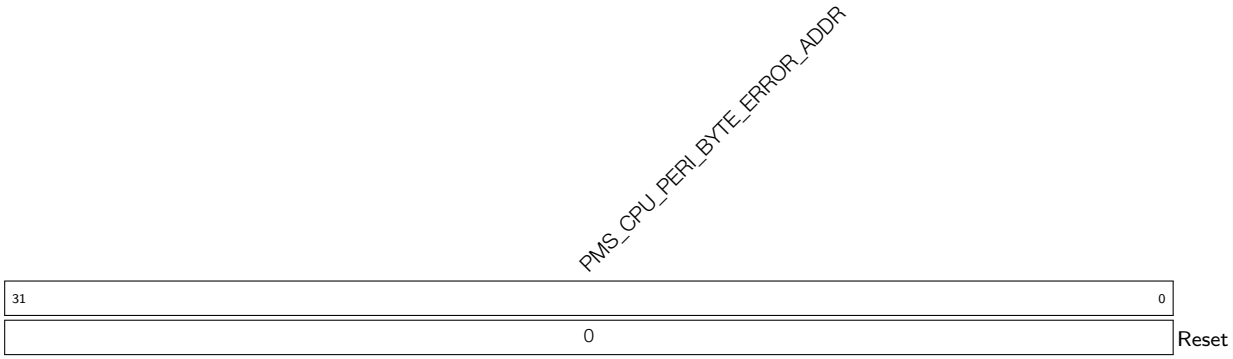


PMS_CPU_PERI_BYTE_ERROR_CLR CPU 外设访问中断的清除信号。(读/写)

PMS_CPU_PERI_BYTE_ERROR_EN CPU 外设访问中断的使能信号。(读/写)

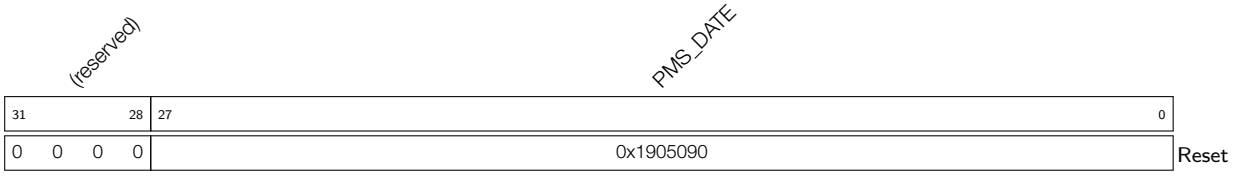
PMS_CPU_PERI_BYTE_ERROR_INTR CPU 外设访问中断信号。(只读)

Register 24.58: PMS_CPU_PERIPHERAL_STATUS_REG (0x0100)



PMS_CPU_PERI_BYTE_ERROR_ADDR 记录 CPU 外设访问的非法地址。(只读)

Register 24.59: PMS_DATE_REG (0x0FFC)



PMS_DATE 版本控制寄存器。(读/写)

25. 数字签名

25.1 概述

数字签名提供了一种使用私钥对消息进行加密、再使用公钥对消息进行验证的方法。数字签名可用于向服务器验证设备自身身份，或验证消息是否未被篡改。

ESP32-S2 包含一个数字签名 (DS) 模块，可高效生成 RSA 数字签名，而软件无法访问 RSA 私钥。

25.2 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

25.3 功能描述

25.3.1 概述

DS 模块计算 RSA 加密操作 $Z = X^Y \bmod M$，其中 Z 是签名，X 是输入消息，Y 和 M 是 RSA 私钥参数。

私钥参数以加密形式存储在 flash 或其他存储器中。这些参数使用一个密钥进行加密，该密钥只能由 DS 模块通过 HMAC 模块获取，并且，求解该密钥所需的一切输入信息只存放在 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，软件绝对不会获取私钥明文。

需要签名时，软件直接将输入消息 X 发送到 DS 外设。加密操作之后，软件将读取签名结果 Z。

25.3.2 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）。相应的公钥也将另外生成和存储，可独立用于验证 DS 签名。

加密操作还需要两个运算子，即参数 $\bar{r}$ 和 M'。这两个参数由 Y 和 M 得出，但需要通过软件提前运算得到。

运算子 Y、M、$\bar{r}$ 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 模块之后由硬件解密并使用密钥生成签名。具体的加密过程请参考章节 25.3.4。

DS 模块计算 RSA 加密操作 $Z = X^Y \bmod M$，更多信息请参考章节 21 *RSA 加速器*中第 21.3.1 节大数模幂运算。

25.3.3 约定

为方便描述，这里约定几个符号和函数，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。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 $[0x0C]_8$。根据需要，$[x]$ 右边可以加上 0，使长度变成 s 位。例如：$[0x5]_4 = 0101$，$[0x5]_8 = 00000101$，$[0x5]_{16} = 0000010100000000$，$[0x13]_8 = 00010011$，$[0x13]_{16} = 0001001100000000$。
- ||　　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。

25.3.4 软件存储私钥

软件要存储用于 DS 加密操作的私钥，需要做以下准备工作：

- 按照章节 25.3.2 所述准备 RSA 私钥 (Y, M) 和运算子 $\bar{r}$ 和 M'。
- 准备一个 256 位 HMAC 密钥 ($[HMAC\_KEY]_{256}$)，存储在 eFuse 中。HMAC 模块通过读取 HMAC 密钥来生成一个密钥，即 DS_KEY = HMAC-SHA256($[HMAC\_KEY]_{256}$, 1^{256})，该密钥用于加解密 RSA 私钥。
- 准备密文 C 形式的加密密钥参数，长度为 1584 字节。

下图描述了软件层面的准备工作和硬件层面的工作流程。

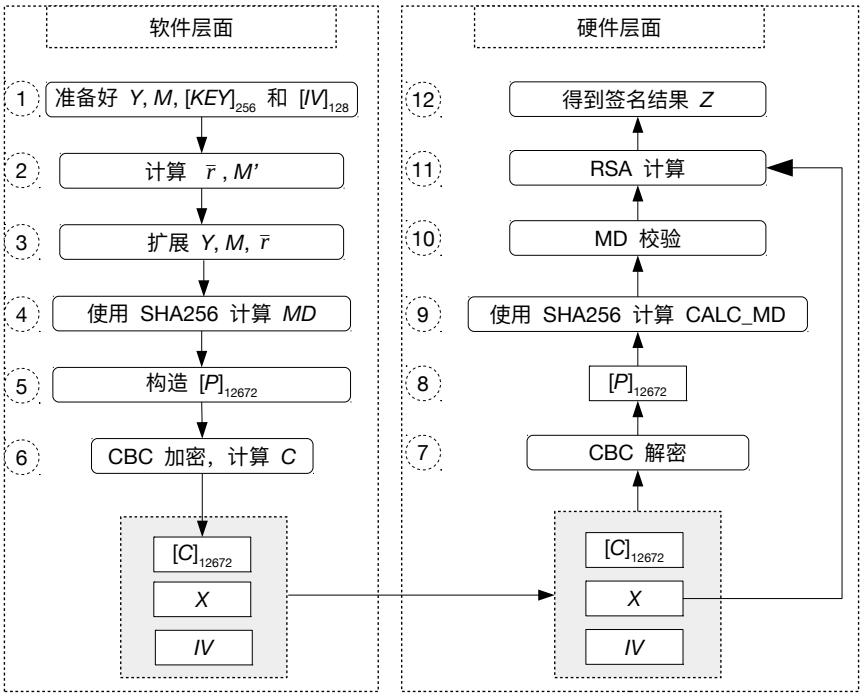


图 25-1. 软件准备工作与硬件工作流程

图 25-1 中左半边给出了计算 C 的过程。用户需要依照图 25-1 指定的步骤来计算 C。更加详细的过程描述如下：

- **步骤 1**：准备好大数 Y 和 M，它们应符合运算子的长度要求。记 $[L]_{32} = \frac{N}{32}$（比如，对于 RSA 4096，$[L]_{32}$ == $[0x80]_{32}$）。另外，准备好 $[DS\_KEY]_{256}$ 和一个随机的 $[IV]_{128}$，他们都要符合 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*。

- **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 个值为 0x08 的字节组成的一个 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 C = $[C]_{12672}$ = AES-CBC-ENC ($[P]_{12672}$, $[DS\_KEY]_{256}$, $[IV]_{128}$)。C 以密文状态包含诸多信息，包括 RSA 运算子 Y、M、$\overline{r}$、M' 和 L，用于校验的 MD 校验码和追加码 $[\beta]_{64}$ 等其他信息。如前文 25.3.4 所述，DS_KEY 由 eFuse 中的 $HMAC\_KEY$ 得出。

25.3.5 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。输入信息是预先生成的私钥密文 C、唯一的消息 X，和 IV。

DS 模块的工作流程可以视为 25.3.4 准备工作中计算 C 的逆过程。可以分为如下三个阶段：

1. **解析阶段，即图 25-1 中的步骤 7 和步骤 8**

   解析过程是图 25-1 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。

   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 25-1 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 25-1 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 25-1 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 $Z = X^Y$ mod M，Z 为签名结果。

25.3.6 软件层面的 DS 操作

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，和 IV。这些软件步骤触发章节 25.3.5 中描述的硬件工作流程。

1. **启动 DS**：对寄存器 DS_SET_START_REG 写 1。
2. **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。
3. **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*。
4. **将 X 写入存储器 DS_X_MEM**：将 X_i ($i \in [0, n) \cap \mathbb{N}$) 写入存储器 DS_X_MEM，容量为 128 个字 (word)。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。
5. **将 C 写入存储器 DS_C_MEM**：将 C_i ($i \in [0, 396) \cap \mathbb{N}$) 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。
6. **启动计算**：对寄存器 DS_SET_ME_REG 写入 1。
7. **等待运算结束**：轮询寄存器 DS_QUERY_BUSY_REG 直到读到 0。
8. **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 10。
   - 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。
   - 如果返回值为 3，则说明填充填充校验失败，且 MD 校验失败，Z 结果值全零无效，跳至步骤 10。
9. **读出运算结果**：从存储器 DS_Z_MEM 读出运算结果 Z_i ($i \in \{0, 1, 2..., n\}$)。Z 以小端字节序存储在存储器中。
10. **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）。

25.4 基地址

用户可以通过两个不同的寄存器基地址访问 DS 模块，如表 130 所示。更多有关通过不同总线访问外设的信息，请参考章节 1 *系统和存储器*。

表 130: 基地址

访问外设总线	地址值
PeriBUS1	0x3F43D000
PeriBUS2	0x6003D000

25.5 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基地址的地址偏移量（相对地址）。请参阅表 130 查询 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	只写
DS_X_MEM	存储器 X	512	0x0800	0x09FF	只写
DS_Z_MEM	存储器 Z	512	0x0A00	0x0BFF	只读

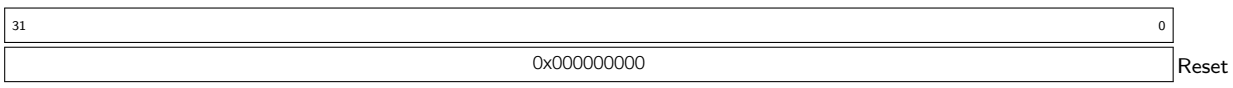
25.6 寄存器列表

请注意，这里的地址是相对于基地址的地址偏移量（相对地址）。请参阅表 130 查询 DS 模块的基地址。

名称	描述	地址	访问
配置寄存器			
DS_IV_0_REG	IV block 数据	0x0630	只写
DS_IV_1_REG	IV block 数据	0x0634	只写
DS_IV_2_REG	IV block 数据	0x0638	只写
DS_IV_3_REG	IV block 数据	0x063C	只写
状态/控制寄存器			
DS_SET_START_REG	启动 DS 模块	0x0E00	只写
DS_SET_ME_REG	开始计算	0x0E04	只写
DS_SET_FINISH_REG	结束计算	0x0E08	只写
DS_QUERY_BUSY_REG	DS 模块状态	0x0E0C	只读
DS_QUERY_KEY_WRONG_REG	查询 DS_KEY 未准备好的原因	0x0E10	只读
DS_QUERY_CHECK_REG	查询校验结果	0x0814	只读
版本寄存器			
DS_DATE_REG	版本控制寄存器	0x0820	读/写

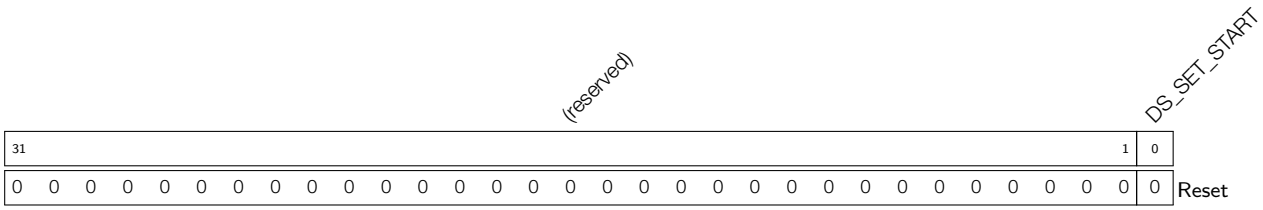
25.7 寄存器

Register 25.1: DS_IV_*m*_REG (*m*: 0-3) (0x0630+4m*)**



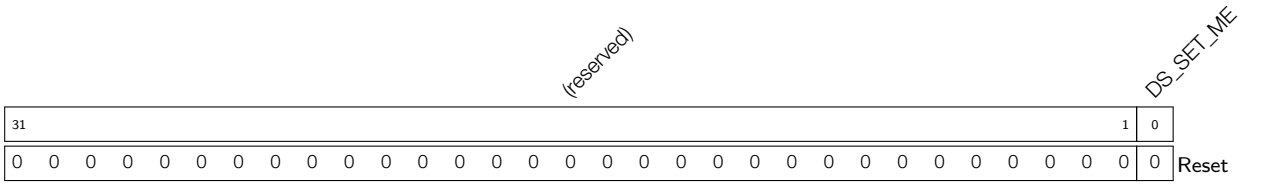
DS_IV_*m*_REG (*m*: 0-3) IV block 数据。（只写）

Register 25.2: DS_SET_START_REG (0x0E00)



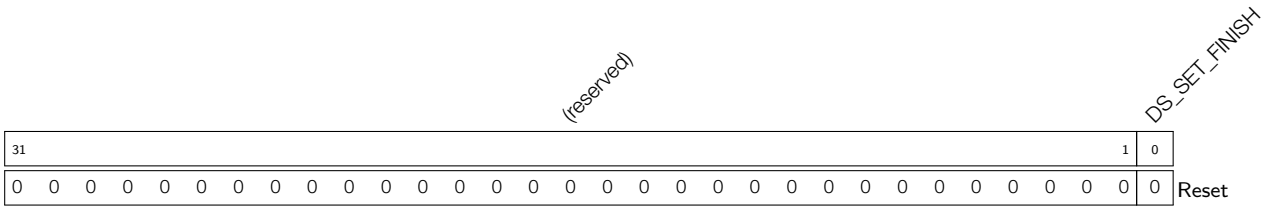
DS_SET_START 写入 1 启动 DS 模块。(只写)

Register 25.3: DS_SET_ME_REG (0x0E04)



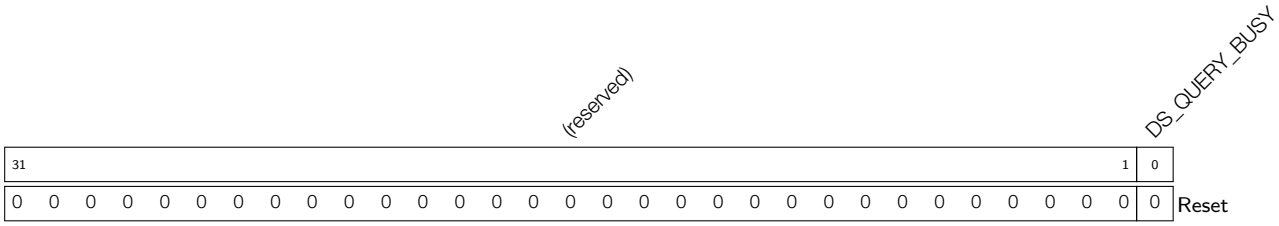
DS_SET_ME 写入 1 以开始计算。(只写)

Register 25.4: DS_SET_FINISH_REG (0x0E08)



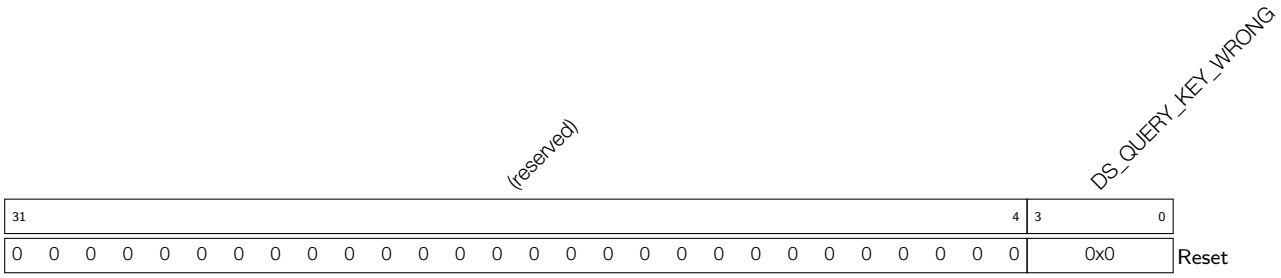
DS_SET_FINISH 写入 1 以结束运算。(只写)

Register 25.5: DS_QUERY_BUSY_REG (0x0E0C)



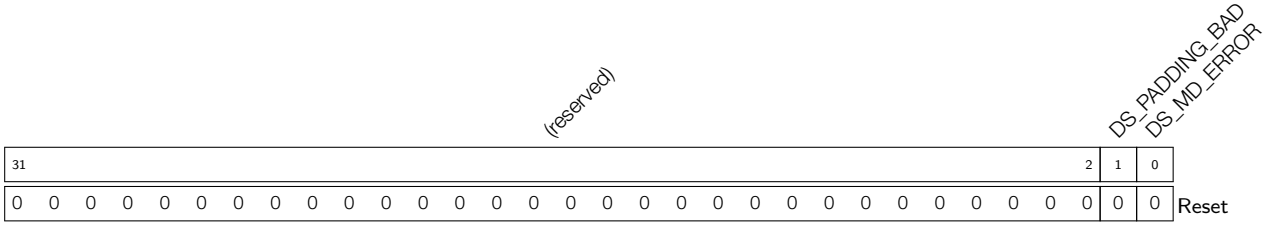
DS_QUERY_BUSY 1：DS 模块正在忙；0：DS 模块空闲。(只读)

Register 25.6: 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。（只读）

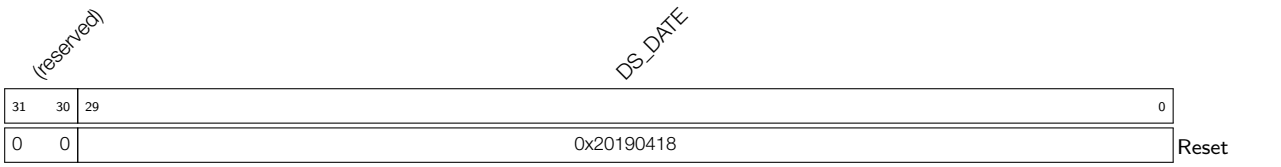
Register 25.7: DS_QUERY_CHECK_REG (0x0E14)



DS_PADDING_BAD 1：填充校验失败；0：填充校验通过。（只读）

DS_MD_ERROR 1：MD 校验失败；0：MD 校验通过。（只读）

Register 25.8: DS_DATE_REG (0x0E20)



DS_DATE 版本控制寄存器（读/写）。

26. HMAC 模块

26.1 概述

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，可配置成不能被软件读取。

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)。

26.2 主要特性

- 标准 HMAC-SHA-256 算法
- HMAC 计算的 hash 结果仅支持可配的硬件外设访问（下行模式）
- 支持挑战-应答算法进行身份验证
- 支持数字签名外设（下行模式）
- 支持重启软禁用的 JTAG（下行模式）

26.3 功能描述

26.3.1 上行模式

在上行模式中，由用户提供 HMAC 信息，且用户回读其计算结果。

该模式下，存储于 eFuse 中的密钥（可配置成无法被软件读取）仅由用户和设备共享。任何支持 HMAC-SHA-256 算法的挑战-应答协议都可以使用该方式。

挑战-应答协议的一般验证方式为：

- 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。如结果相同，则验证通过了 B 的身份

设置密钥：

1. 随机生成一个 256-bit HMAC 密钥，将其烧写到一个 eFuse 密钥块中，并声明该密钥块的功能为 EFUSE_KEY_PURPOSE_HMAC_UP。详细信息可参见章节 16。
2. 配置 eFuse 密钥块读保护功能，软件无法读取密钥值。请求验证该设备的任意一方应保存该密钥的副本。

计算 HMAC 值：

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，一次写入一个块中。
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 26.3.5。

26.3.2 下行 JTAG 启动模式

eFuse memory 中有两个参数可以关闭 JTAG 调试：EFUSE_HARD_DIS_JTAG 和 EFUSE_SOFT_DIS_JTAG。前者烧写为 1，JTAG 将被永久关闭；后者烧写为 1，JTAG 将被暂时关闭。详细信息可参见章节 16。

烧录 EFUSE_SOFT_DIS_JTAG 后，可以使用 HMAC 外设重启 JTAG。

设置密钥：

1. 随机生成一个 256-bit HMAC 密钥，将其烧写到一个 eFuse 密钥块中，并声明该密钥块的功能为 EFUSE_KEY_PURPOSE_HMAC_DOWN_JTAG 或 EFUSE_KEY_PURPOSE_HMAC_DOWN_ALL。若设置为 EFUSE_KEY_PURPOSE_HMAC_DOWN_ALL，则该密钥可同时用于 DS 和 JTAG 的重启功能。
2. 配置 eFuse 密钥块读保护功能，软件无法读取密钥值。用户将步骤中生成的随机密钥安全地存储在其他位置。
3. 将 EFUSE_SOFT_DIS_JTAG 烧写为 1。

重启 JTAG：

1. 用户预先在本地使用 SHA-256 和已知的随机密钥对 32 字节的 0x00 进行 HMAC 计算，并将计算的值输入至寄存器 SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_0 ~ SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_7 中。
2. 用户启动 HMAC 模块，并进入下行 TJAG 启动模式。
3. 如果 HMAC 计算的结果与寄存器 SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_0 ~ SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_7 中提供的值匹配，则 JTAG 重启。否则，JTAG 保持关闭状态。
4. 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 3 中的状态。

有关此过程的详细步骤，可参见章节 26.3.5。

26.3.3 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。

设置密钥：

1. 随机生成一个 256-bit HMAC 密钥，将其烧写到一个 eFuse 密钥块中，并声明该密钥块的功能为 EFUSE_KEY_PURPOSE_HMAC_DOWN_DIGITAL_SIGNATURE 或 EFUSE_KEY_PURPOSE_HMAC_DOWN_ALL。若设置为 EFUSE_KEY_PURPOSE_HMAC_DOWN_ALL，则该密钥可同时用于 DS 和 JTAG 的重启功能。
2. 配置 eFuse 密钥块读保护功能，软件无法读取密钥值。必要时，可将密钥的副本存储在安全的位置。

在启用 DS 模块之前，需要软件先开启 HMAC 模块下行 DS 模式的计算任务。上述计算结果将作为 DS 模块执行计算任务时的密钥。详细信息可参见章节 25。

26.3.4 HMAC eFuse 配置

HMAC 模块的正确执行取决于选取的 eFuse 密钥块与配置的 HMAC 功能是否一致。

配置 HMAC 的功能

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 133 列出了各功能对应的配置寄存器的数值，用户应将所使用功能对应的数值写入寄存器 HMAC_SET_PARA_PURPOSE_REG（可参见章节 26.3.5）。

表 133: 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

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。

详细信息可参见章节 16。

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 133 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。

26.3.5 调用 HMAC 流程（详细说明）

用户调用 ESP32-S2 中 HMAC 流程如下：

1. 启动 HMAC 模块
   (a) 启动 HMAC 和 SHA 的外设时钟位，清除相应的外设重启位。
   (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 55 描述了数值 m 对应的密钥功能，可参见章节 26.3.4。
   (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 26.3.4。
   (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。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，继续下一步。

(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，继续下一步。

(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 26.4.1 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。

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。

26.4 HMAC 算法细节

26.4.1 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 “5.1 Padding the Message” 相同，并在其章节中有详细描述。

如图26-1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。

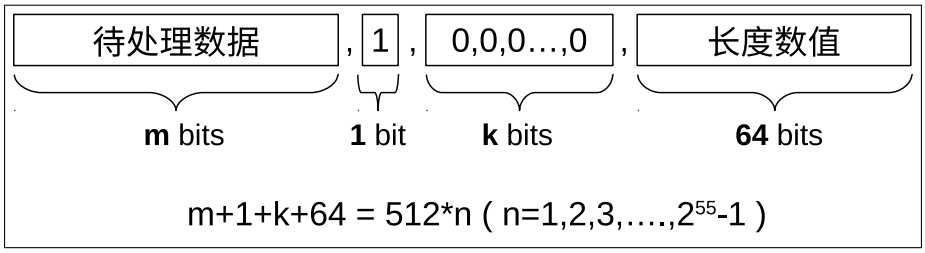


图 26-1. HMAC 附加填充比特示意图

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 26.3.5。

26.4.2 HMAC 算法结构

HMAC 模块中应用的算法结构示意图如 26-2 所示。这是 RFC 2104 中描述的标准 HMAC 算法。

图 26-2 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。

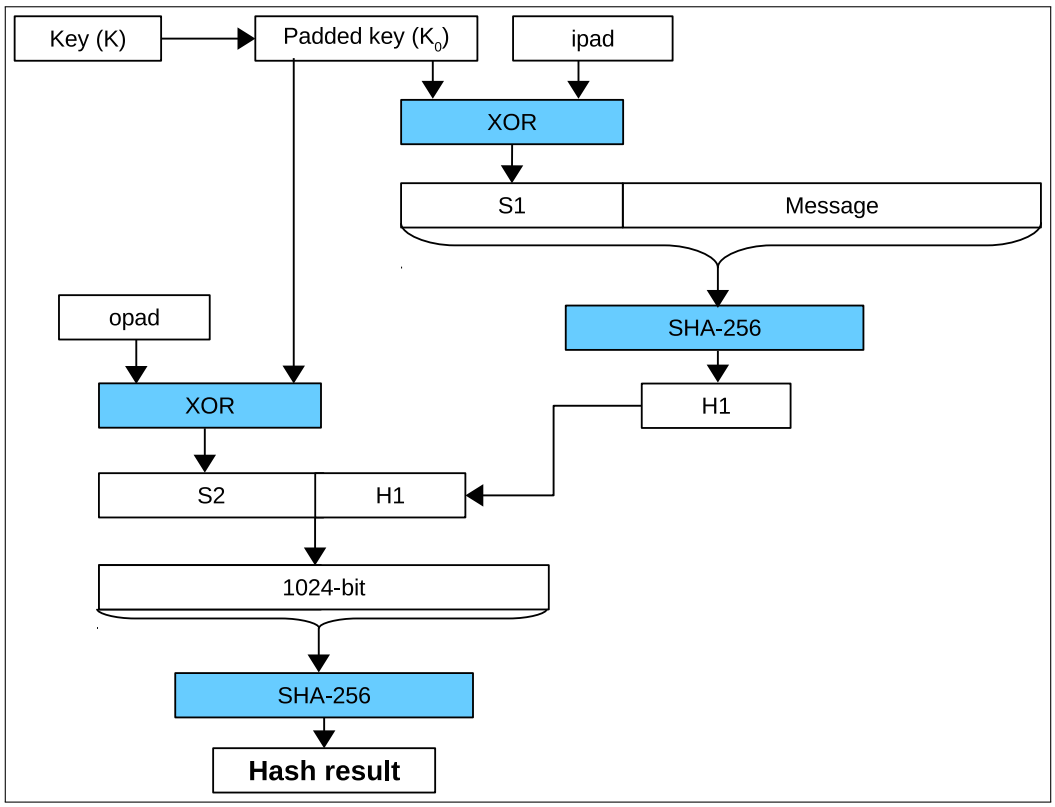


图 26-2. HMAC 结构示意图

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 26.4.1 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。

26.5 基地址

用户可以通过两个不同的寄存器基地址访问 HMAC 模块，如表 134 所示。

表 134: HMAC 基地址

基地址	地址值
PeriBUS1	0x3F43E000
PeriBUS2	0x6003E000

26.6 寄存器列表

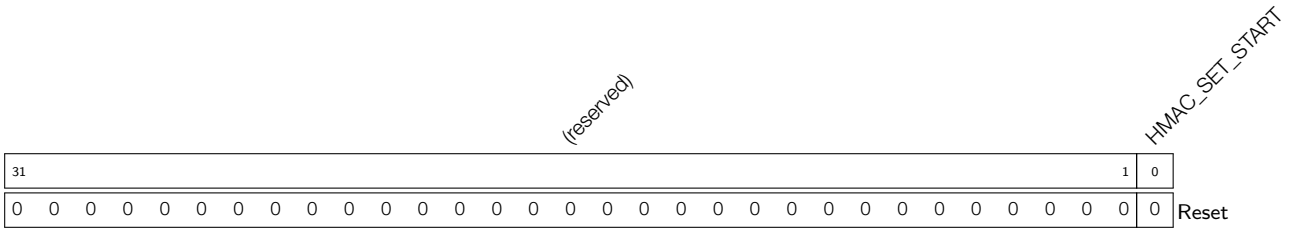
请注意，这里的地址是相对于系统寄存器基地址的地址偏移量 (相对地址)。请参阅章节 26.5 获取有关系统寄存器基地址的信息。

名称	描述	地址	访问权限
Control/Status Registers			
HMAC_SET_START_REG	HMAC 开始控制寄存器	0x0040	只写
HMAC_SET_PARA_FINISH_REG	HMAC 配置完成寄存器	0x004C	只写
HMAC_SET_MESSAGE_ONE_REG	HMAC 信息控制寄存器	0x0050	只写
HMAC_SET_MESSAGE_ING_REG	HMAC 信息继续寄存器	0x0054	只写
HMAC_SET_MESSAGE_END_REG	HMAC 信息终止寄存器	0x0058	只写
HMAC_SET_RESULT_FINISH_REG	HMAC 读取结果完成寄存器	0x005C	只写
HMAC_SET_INVALIDATE_JTAG_REG	注销 JTAG 结果寄存器	0x0060	只写
HMAC_SET_INVALIDATE_DS_REG	注销数字签名结果寄存器	0x0064	只写
HMAC_QUERY_ERROR_REG	存储用户配置的密钥和功能的配对结果	0x0068	只读
HMAC_QUERY_BUSY_REG	存储 HMAC 模块的忙碌状态	0x006C	只读
configuration Registers			
HMAC_SET_PARA_PURPOSE_REG	HMAC 参数配置寄存器	0x0044	只写
HMAC_SET_PARA_KEY_REG	HMAC 密钥配置寄存器	0x0048	只写
HMAC Message Block			
HMAC_WR_MESSAGE_0_REG	信息寄存器 0	0x0080	只写
HMAC_WR_MESSAGE_1_REG	信息寄存器 1	0x0084	只写
HMAC_WR_MESSAGE_2_REG	信息寄存器 2	0x0088	只写
HMAC_WR_MESSAGE_3_REG	信息寄存器 3	0x008C	只写
HMAC_WR_MESSAGE_4_REG	信息寄存器 4	0x0090	只写
HMAC_WR_MESSAGE_5_REG	信息寄存器 5	0x0094	只写
HMAC_WR_MESSAGE_6_REG	信息寄存器 6	0x0098	只写
HMAC_WR_MESSAGE_7_REG	信息寄存器 7	0x009C	只写
HMAC_WR_MESSAGE_8_REG	信息寄存器 8	0x00A0	只写
HMAC_WR_MESSAGE_9_REG	信息寄存器 9	0x00A4	只写
HMAC_WR_MESSAGE_10_REG	信息寄存器 10	0x00A8	只写
HMAC_WR_MESSAGE_11_REG	信息寄存器 11	0x00AC	只写
HMAC_WR_MESSAGE_12_REG	信息寄存器 12	0x00B0	只写
HMAC_WR_MESSAGE_13_REG	信息寄存器 13	0x00B4	只写
HMAC_WR_MESSAGE_14_REG	信息寄存器 14	0x00B8	只写
HMAC_WR_MESSAGE_15_REG	信息寄存器 15	0x00BC	只写
HMAC Upstream Result			
HMAC_RD_RESULT_0_REG	Hash 结果寄存器 0	0x00C0	只读
HMAC_RD_RESULT_1_REG	Hash 结果寄存器 1	0x00C4	只读
HMAC_RD_RESULT_2_REG	Hash 结果寄存器 2	0x00C8	只读
HMAC_RD_RESULT_3_REG	Hash 结果寄存器 3	0x00CC	只读
HMAC_RD_RESULT_4_REG	Hash 结果寄存器 4	0x00D0	只读
HMAC_RD_RESULT_5_REG	Hash 结果寄存器 5	0x00D4	只读
HMAC_RD_RESULT_6_REG	Hash 结果寄存器 6	0x00D8	只读
HMAC_RD_RESULT_7_REG	Hash 结果寄存器 7	0x00DC	只读
Control/Status registers			
HMAC_SET_MESSAGE_PAD_REG	软件填充寄存器	0x00F0	只写

名称	描述	地址	访问权限
HMAC_ONE_BLOCK_REG	One block 信息寄存器	0x00F4	只写
Version Register			
HMAC_DATE_REG	版本控制寄存器	0x00F8	读写

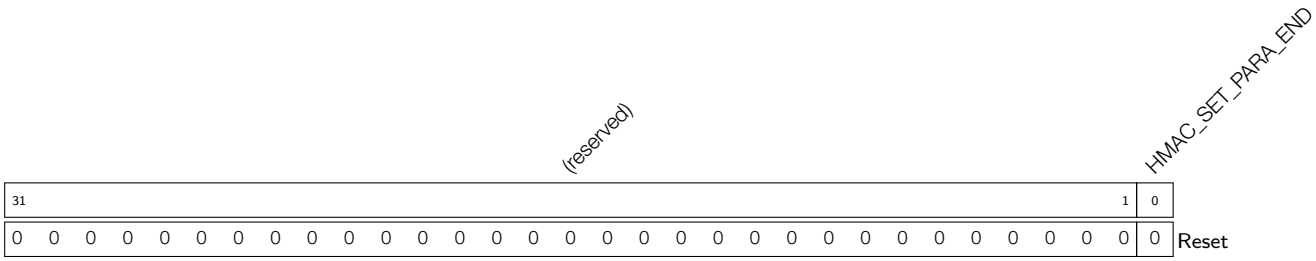
26.7 寄存器

Register 26.1: HMAC_SET_START_REG (0x0040)



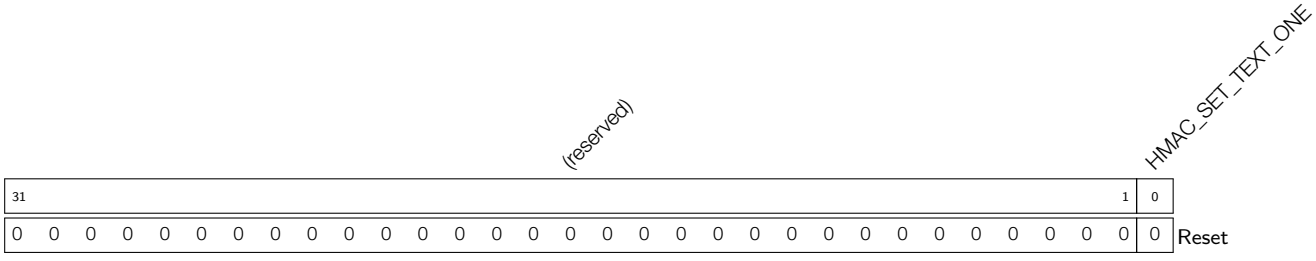
HMAC_SET_START 置 1 启动 HMAC。(只写)

Register 26.2: HMAC_SET_PARA_FINISH_REG (0x004C)



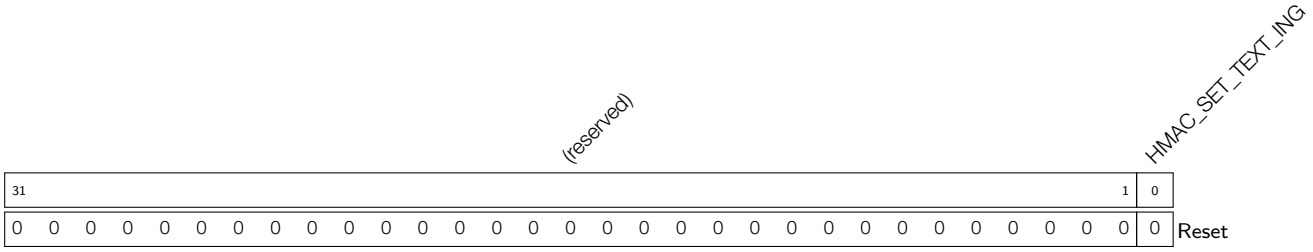
HMAC_SET_PARA_END 置 1 完成 HMAC 配置。(只写)

Register 26.3: HMAC_SET_MESSAGE_ONE_REG (0x0050)



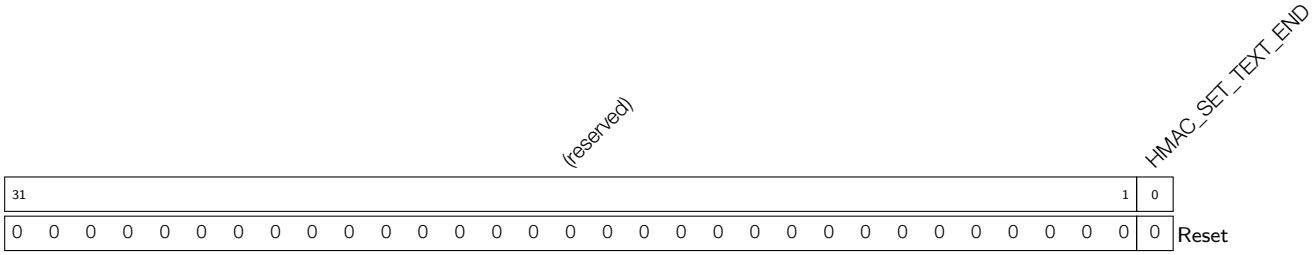
HMAC_SET_TEXT_ONE 调用 SHA 计算信息块。(只写)

Register 26.4: HMAC_SET_MESSAGE_ING_REG (0x0054)



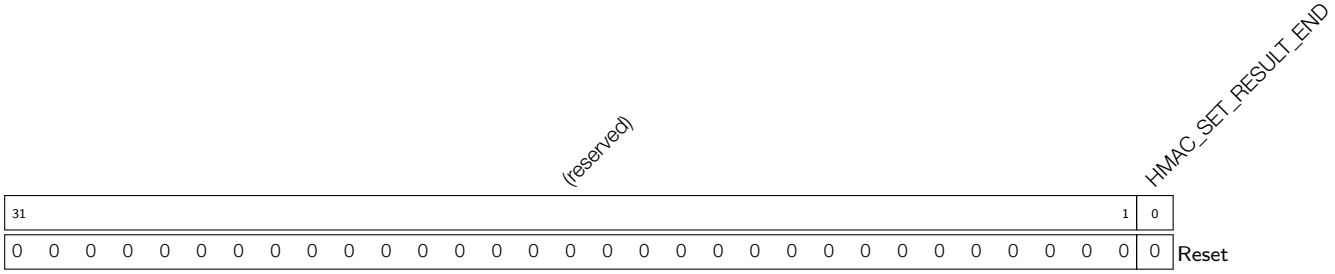
HMAC_SET_TEXT_ING 置 1 表明仍存在未处理信息块。(只写)

Register 26.5: HMAC_SET_MESSAGE_END_REG (0x0058)



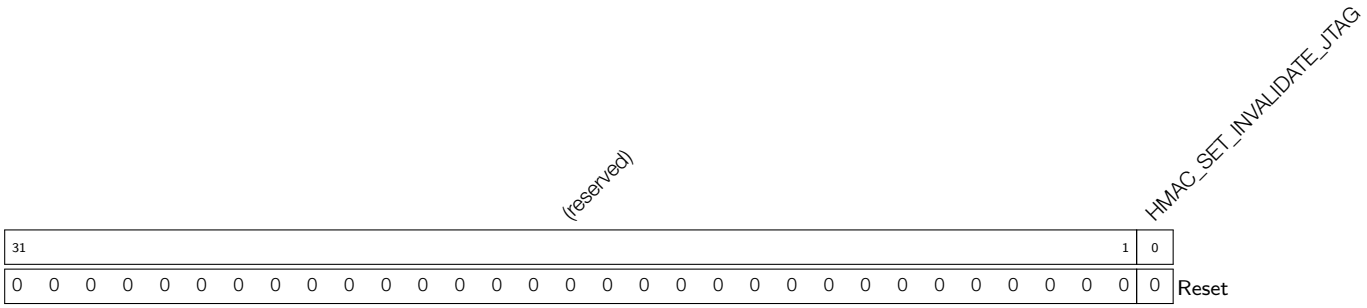
HMAC_SET_TEXT_END 置 1 开始硬件填充。(只写)

Register 26.6: HMAC_SET_RESULT_FINISH_REG (0x005C)



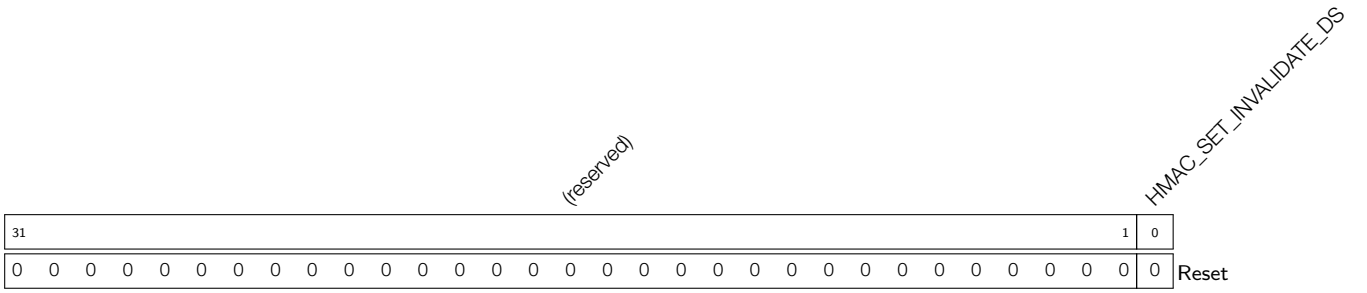
HMAC_SET_RESULT_END 置 1 结束上行模式，清空计算结果。(只写)

Register 26.7: HMAC_SET_INVALIDATE_JTAG_REG (0x0060)



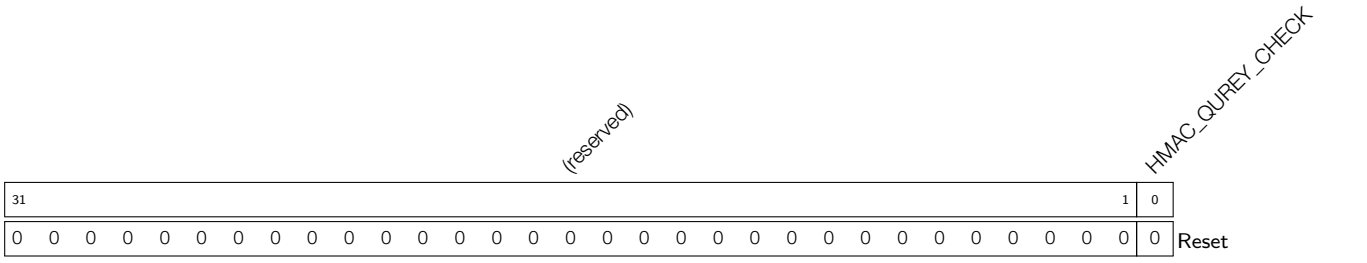
HMAC_SET_INVALIDATE_JTAG 置 1 清空下行模式下 JTAG 重启功能的计算结果。(只写)

Register 26.8: HMAC_SET_INVALIDATE_DS_REG (0x0064)



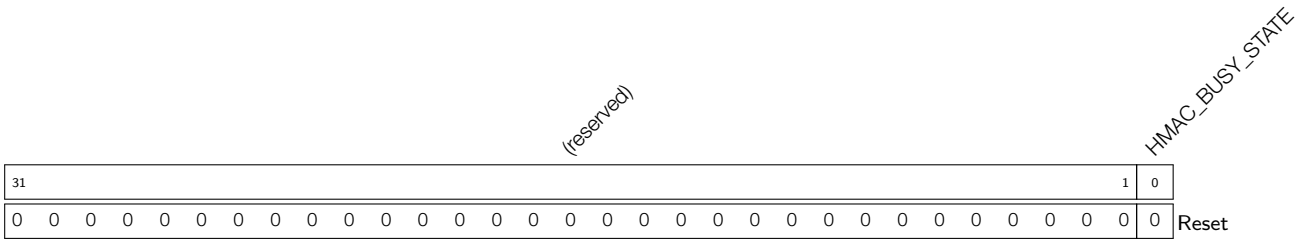
HMAC_SET_INVALIDATE_DS 置 1 清空下行模式下 DS 功能的计算结果。(只写)

Register 26.9: HMAC_QUERY_ERROR_REG (0x0068)



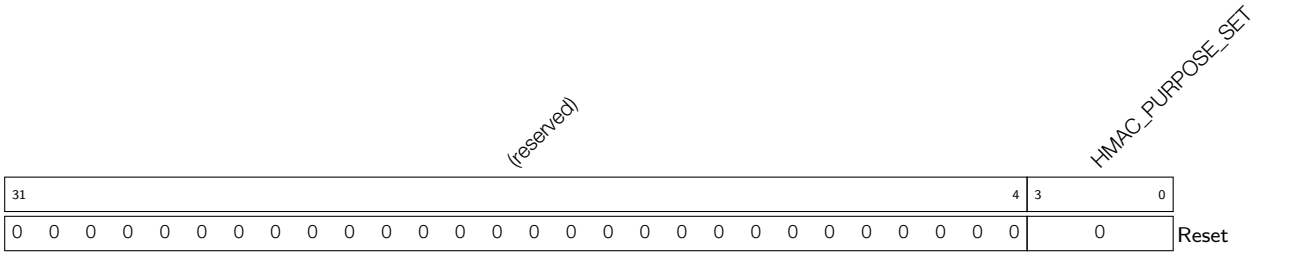
HMAC_QUREY_CHECK 指示 HMAC 错误状态。0: HMAC 密钥和功能匹配。1: 错误。(只读)

Register 26.10: HMAC_QUERY_BUSY_REG (0x006C)



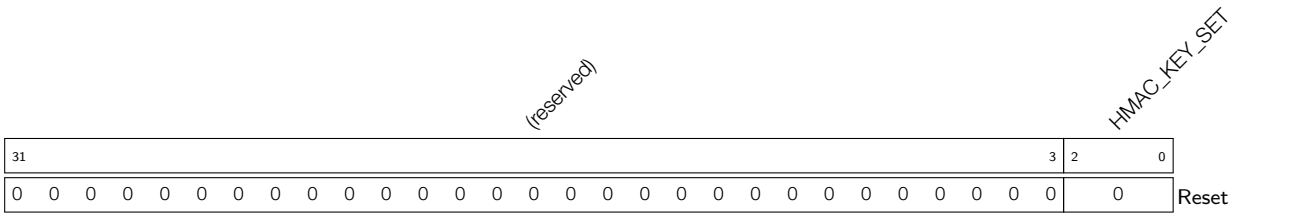
HMAC_BUSY_STATE 指示 HMAC 是否处于“忙碌”状态。1’b0: 空闲 1’b1: 忙碌（只读）

Register 26.11: HMAC_SET_PARA_PURPOSE_REG (0x0044)



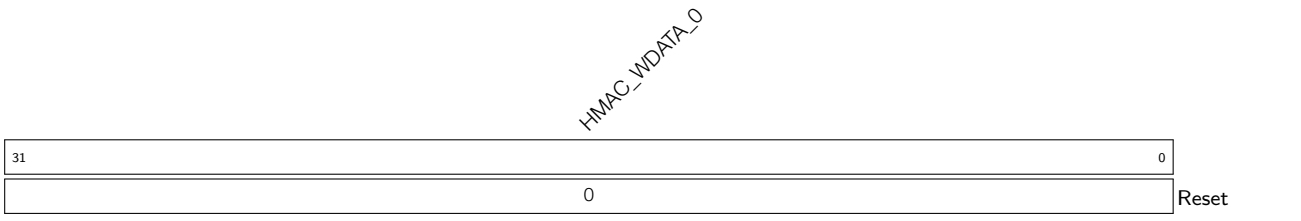
HMAC_PURPOSE_SET 设置 HMAC 功能。（只写）

Register 26.12: HMAC_SET_PARA_KEY_REG (0x0048)



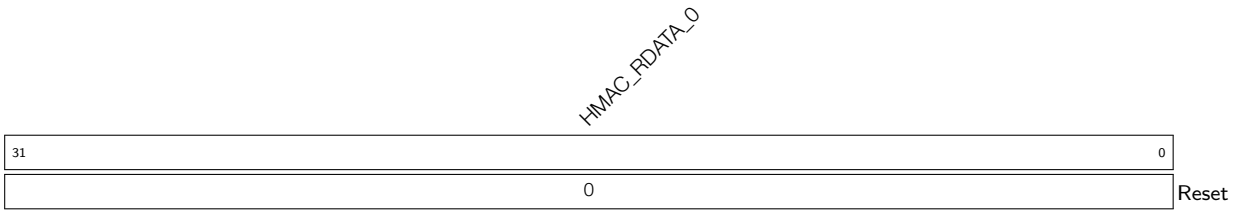
HMAC_KEY_SET 选择 HMAC 密钥。（只写）

Register 26.13: HMAC_WR_MESSAGE_*n*_REG (*n*: 0-15) (0x0080+4**n*)



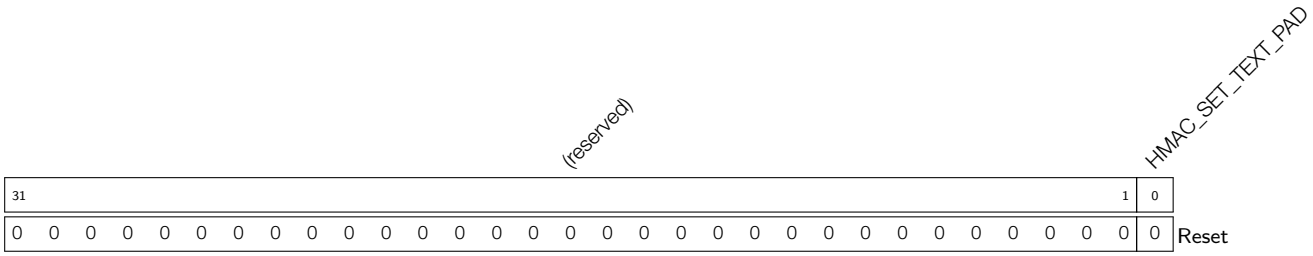
HMAC_WDATA_*n* 存储信息的第 *n* 个 32 位数据信息。（只写）

Register 26.14: HMAC_RD_RESULT_*n*_REG (*n*: 0-7) (0x00C0+4**n*)



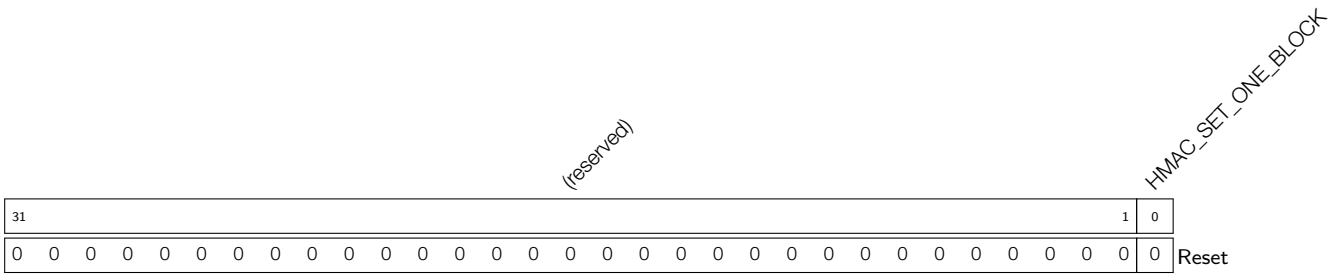
HMAC_RDATA_*n* 读取 hash 结果的第 *n* 个 32 位。（只读）

Register 26.15: HMAC_SET_MESSAGE_PAD_REG (0x00F0)



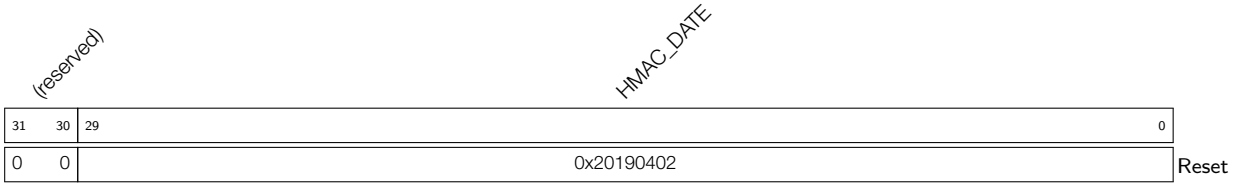
HMAC_SET_TEXT_PAD 置 1 表明由软件执行填充操作。（只写）

Register 26.16: HMAC_ONE_BLOCK_REG (0x00F4)



HMAC_SET_ONE_BLOCK 置 1 表明无需填充。（只写）

Register 26.17: HMAC_DATE_REG (0x00F8)



HMAC_DATE 存储 HMAC 版本信息。（读写）

27. 超低功耗协处理器

27.1 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（详见低功耗管理章节），允许开发者通过存储在 RTC 存储器中的专用程序，访问 RTC 外围设备、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 RTCIO、RTC I²C、SAR ADC、温度传感器 (TSENS) 等外设检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。

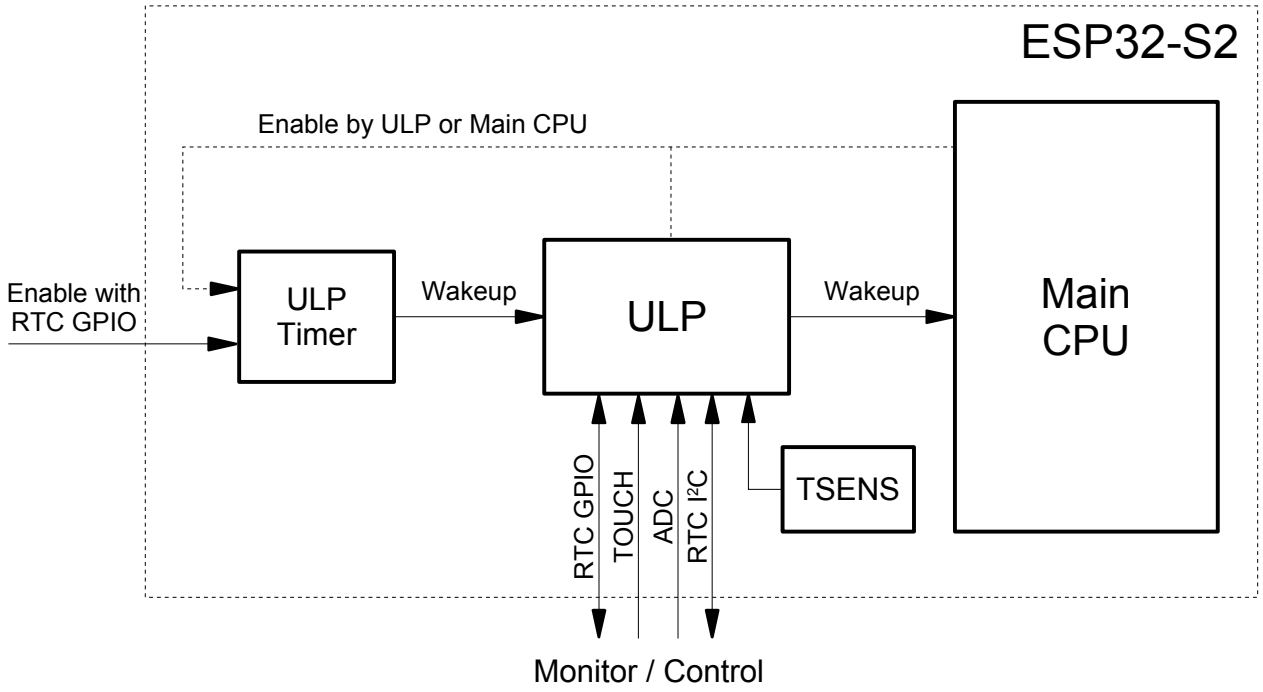


图 27-1. 超低功耗协处理器概图

ESP32-S2 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。

27.2 主要特性

- 可访问最多 8 KB SRAM RTC 慢速内存，储存指令和数据
- 采用 8 MHz RTC_FAST_CLK 时钟频率
- 支持正常模式和 Monitor 模式
- 可唤醒 CPU 或向 CPU 发送中断
- 可访问外围设备、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-S2 的超低功耗协处理器。
ULP-FSM 与 ULP-RISC-V 特性比较如下：

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		8 MHz	
唤醒源		ULP 定时器	
工作模式	正常模式	当系统唤醒时，协助主 CPU 完成部分任务	
	Monitor 模式	当系统休眠时，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		ADC0/ADC1	
		DAC0/DAC1	
		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		特殊指令	标准 C 编译器

表 136: 超低功耗协处理器特性比较

ESP32-S2 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-S2 协处理器的基本架构可见图 27-2。

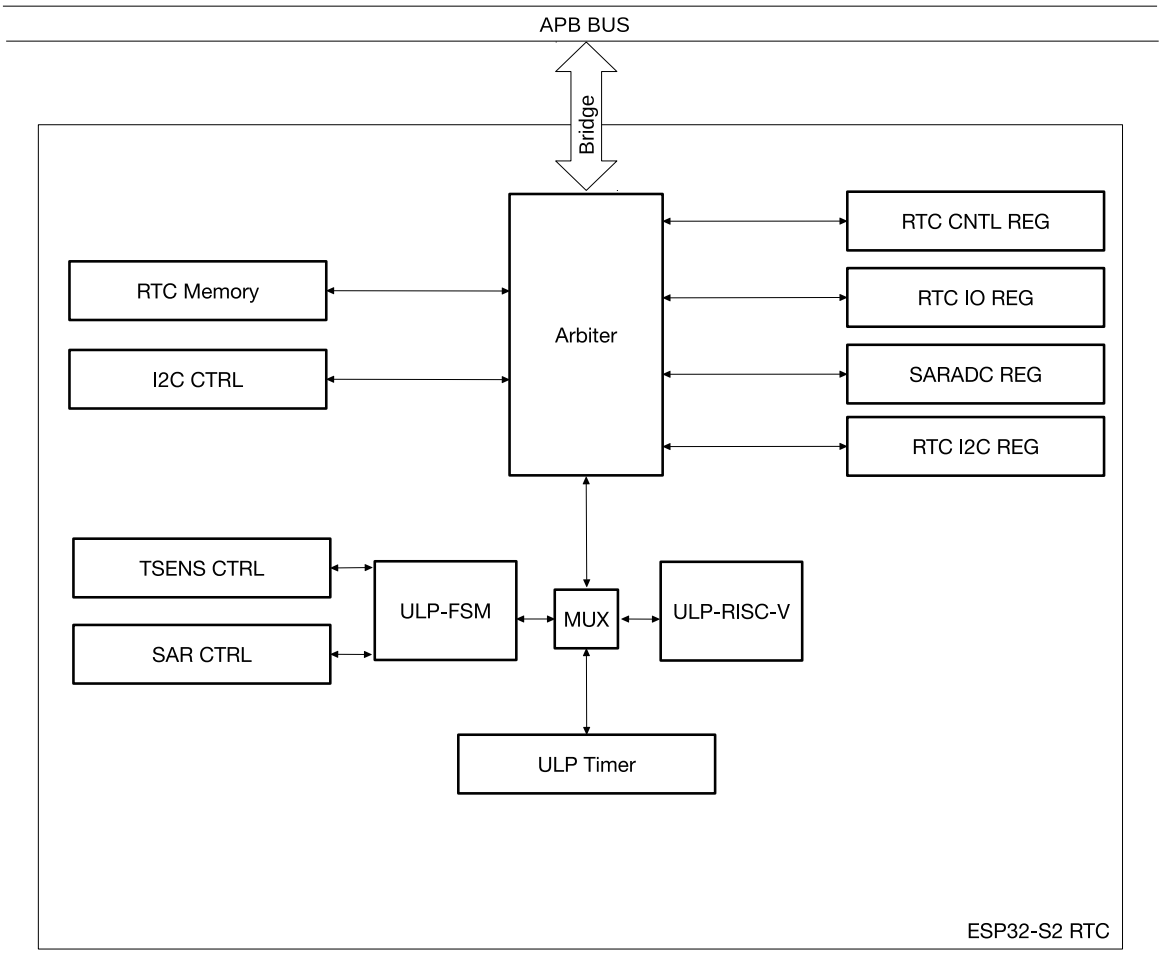


图 27-2. 超低功耗协处理器基本架构

27.3 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 27.5.2。

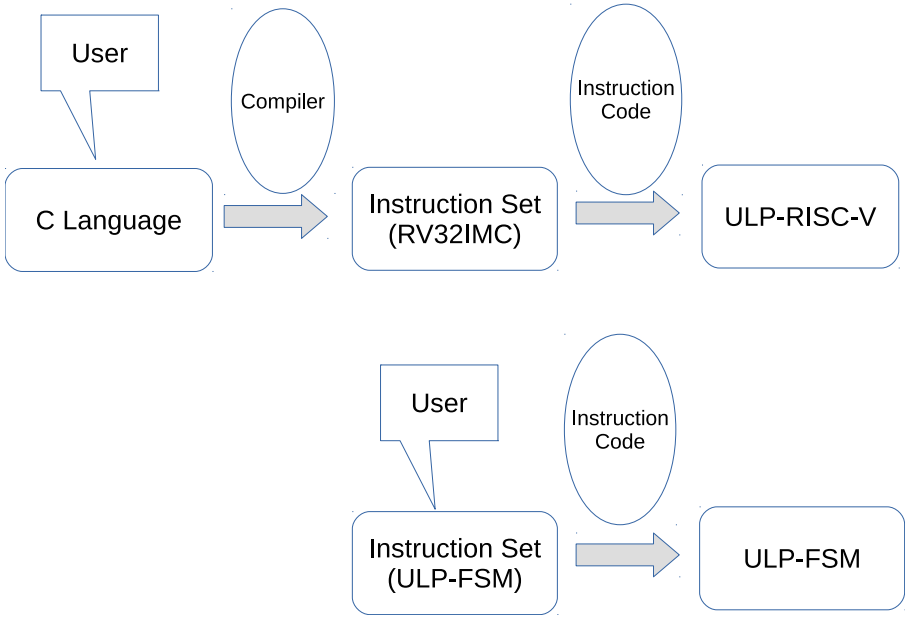


图 27-3. 编程流程图

27.4 协处理器的睡眠和唤醒流程

ESP32-S2 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烧写到 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. 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；
4. 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。
5. 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；
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 27-4，包括：

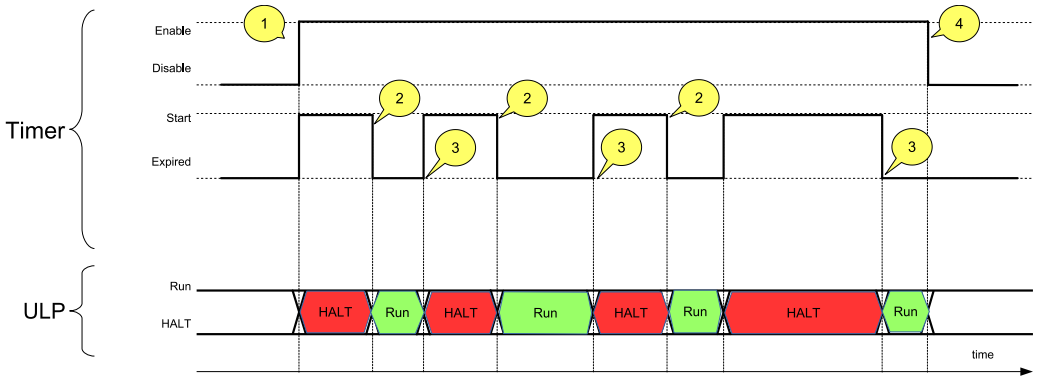


图 27-4. ULP 程序流程框图

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，系统将不再进入 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 状态需要配置寄存器 RTC_CNTL_COCPU_DONE，所以建议预先烧写的程序用下列代码结尾：

- 置位寄存器 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；
- 置位寄存器 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 27-5。

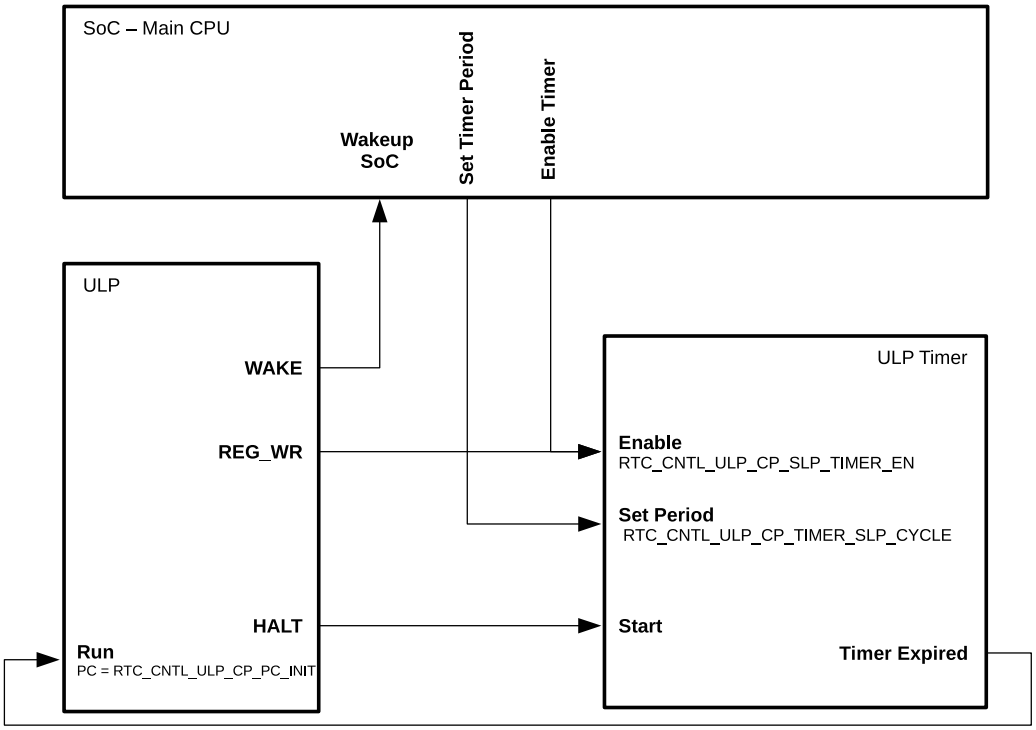


图 27-5. ULP 程序框图

27.5 ULP-FSM

27.5.1 ULP-FSM 主要特性

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 控制与外围设备控制指令。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。

27.5.2 ULP-FSM 指令集

ULP-FSM 可支持下列指令：

- ALU - 算数与逻辑
- 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 27-6。

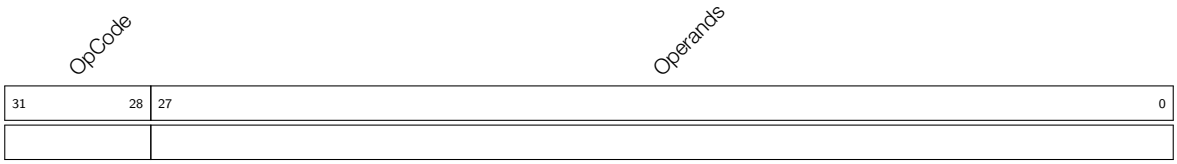


图 27-6. ULP-FSM 协处理器的指令格式

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 种不同的算数和逻辑运算，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。

27.5.2.1 ALU - 算数与逻辑运算

算数逻辑单元 (ALU) 可以进行算数和逻辑运算，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：

- 算数 - 加 (ADD) 和减 (SUB)
- 逻辑 - 与 (AND) 和或 (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，选择特定的算数和逻辑运算。

对寄存器数值的运算

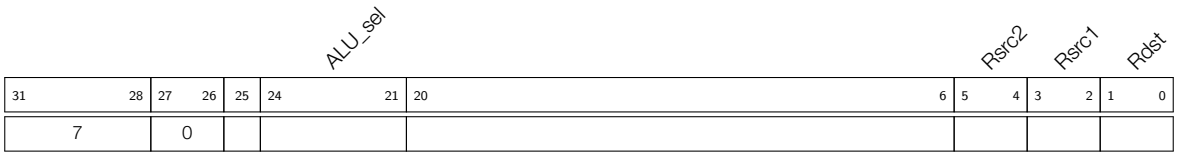


图 27-7. 指令类型 - 对寄存器数值的 ALU 运算

如图 27-7 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 137。

Operand	**描述** - 见图 27-7
Rdst	寄存器 R[0-3]，目标地址寄存器
Rsrc1	寄存器 R[0-3]，源地址寄存器
Rsrc2	寄存器 R[0-3]，源地址寄存器
ALU_sel	ALU 运算类型，具体见表 137

ALU_sel	指令	运算类型	描述
0	ADD	*Rdst* = *Rsrc1* + *Rsrc2*	加
1	SUB	*Rdst* = *Rsrc1* - *Rsrc2*	减
2	AND	*Rdst* = *Rsrc1* & *Rsrc2*	逻辑与
3	OR	*Rdst* = *Rsrc1* | *Rsrc2*	逻辑或
4	MOVE	*Rdst* = *Rsrc1*	寄存器赋值
5	LSH	*Rdst* = *Rsrc1* «␣*Rsrc2*	逻辑左移
6	RSH	*Rdst* = *Rsrc1* »␣*Rsrc2*	逻辑右移

表 137: 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

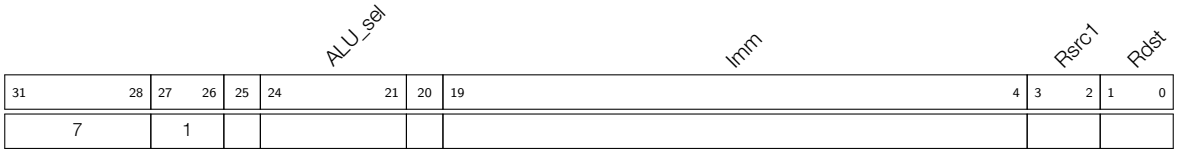


图 27-8. 指令类型 - 对指令立即值的 ALU 运算

如图 27-8 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 138。

Operand	**描述** - 见图 27-8
Rdst	寄存器 R[0-3]，目标地址寄存器
Rsrc1	寄存器 R[0-3]，源地址寄存器
Imm	指令立即值，16 位有符号数
ALU_sel	ALU 运算类型，具体见表 138

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*	逻辑与
3	OR	*Rdst* = *Rsrc1* | *Imm*	逻辑或
4	MOVE	*Rdst* = *Imm*	寄存器赋值
5	LSH	*Rdst* = *Rsrc1* «␣*Imm*	逻辑左移
6	RSH	*Rdst* = *Rsrc1* »␣*Imm*	逻辑右移

表 138: 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

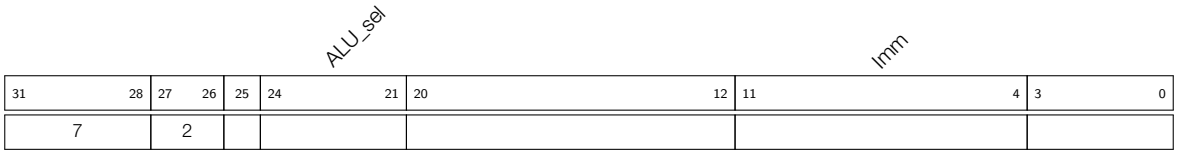


图 27-9. 指令类型 - 对阶段计数器寄存器的 ALU 运算

如图 27-9 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 27-9。Stage_cnt 是一个独立的寄存器，并不是图 27-9 所示指令的一部分。

Operand	**描述** - 见图 27-9
Imm	指令立即值，8 位数
ALU_sel	ALU 运算类型，具体见表 27-9
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 139: 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。

27.5.2.2 ST - 存储数据至内存

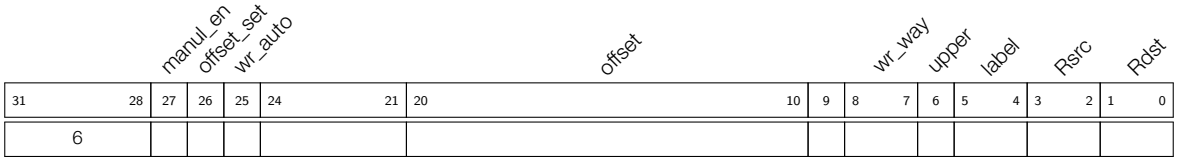


图 27-10. 指令类型 - ST

Operand	**描述** - 见图 27-10
Rdst	寄存器 R[0-3]，目标寄存器地址，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
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

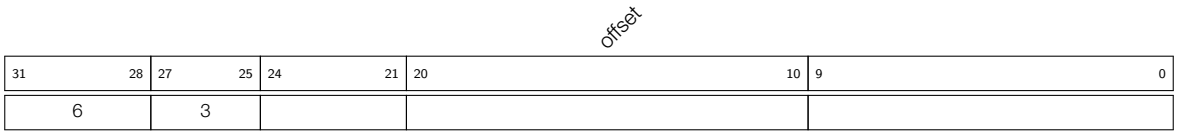


图 27-11. 指令类型 - 地址自增模式的基地址偏移 (ST-OFFSET)

Operand	**描述** - 见图 27-11
offset	初始地址偏移量，11 位有符号数，单位为 32 位字。

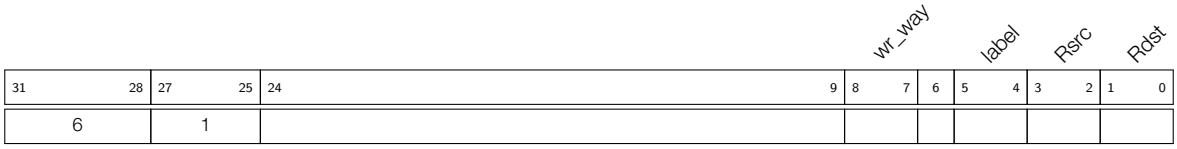


图 27-12. 指令类型 - 地址自增模式的数据存储 (ST-AUTO-DATA)

Operand	**描述** - 见图 27-12
Rdst	寄存器 R[0-3]，目标寄存器地址，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 140。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], 5'b0, Rsrc[15:0]}	写全字，包含指针和数据
1	奇数	Mem [Rdst + Offset]{15:0} = {Label[1:0],Rscr[13:0]}	低半字存储带 label 的数据
1	偶数	Mem [Rdst + Offset]{31:16} = {Label[1:0],Rscr[13:0]}	高半字存储带 label 的数据
3	奇数	Mem [Rdst + Offset]{15:0} = Rscr[15:0]	低半字存储不带 label 的数据
3	偶数	Mem [Rdst + Offset]{31:16} = Rscr[15:0]	高半字存储不带 label 的数据

表 140: 数据存储格式-地址自增模式

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

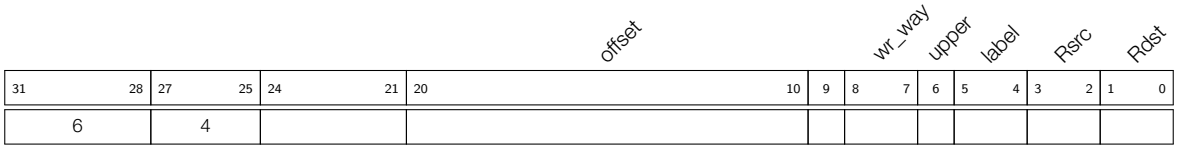


图 27-13. 指令类型 - 指定地址模式的数据存储

Operand	**描述** - 见图 27-13
Rdst	寄存器 R[0-3]，目标寄存器地址，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 141。

wr_way	upper	存储数据	运算
0	*	Mem [Rdst + Offset]{31:0} ={PC[10:0], 5'b0, Rsrc[15:0]}	写全字，包含指针和数据
1	0	Mem [Rdst + Offset]{15:0} = {Label[1:0],Rscr[13:0]}	低半字存储带 label 的数据
1	1	Mem [Rdst + Offset]{31:16} = {Label[1:0],Rscr[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 141: 数据存储格式-地址指定模式

27.5.2.3 LD - 从内存加载数据

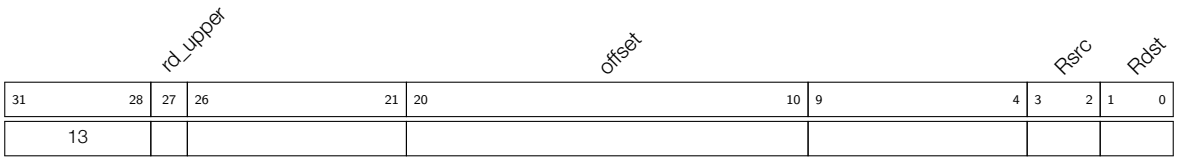


图 27-14. 指令类型 - LD

Operand	**描述** - 见图 27-14
Rdst	寄存器 R[0-3]，目标寄存器
Rsrc	寄存器 R[0-3]，保留目标寄存器的地址，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： 0 - 读取高半字 1 - 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。
- 加载的 Mem 将存储至 RTC_SLOW_MEM 慢速内存中，ULP 协处理器的地址 0 即相当于主 CPU 的地址 0x50000000。

27.5.2.4 JUMP - 跳转至绝对地址

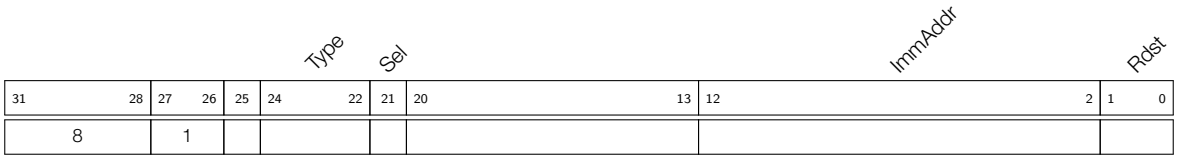


图 27-15. 指令类型 - JUMP

Operand	**描述** - 见图 27-15
Rdst	寄存器 R[0-3]，储存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。

27.5.2.5 JUMPR - 跳转至相对地址（基于 R0 寄存器判断）

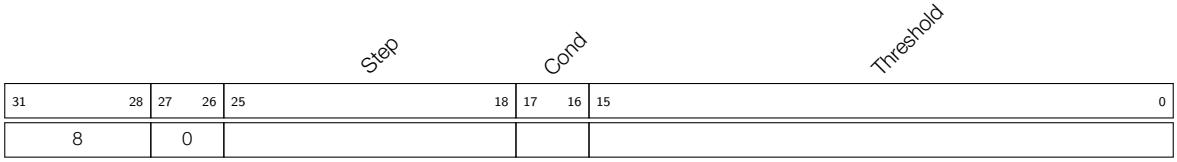


图 27-16. 指令类型 - JUMPR

Operand	**描述** - 见图 27-16
Threshold	跳转条件阈值（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。

27.5.2.6 JUMPS - 跳转至相对地址（基于阶段计数器寄存器判断）

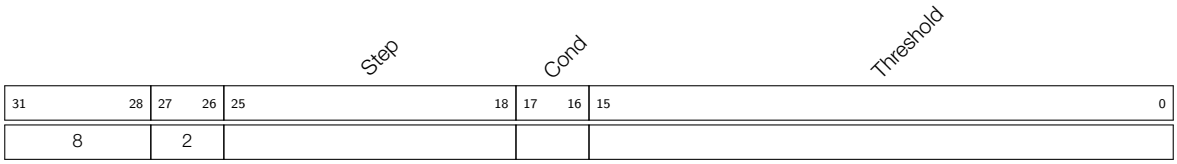


图 27-17. 指令类型 - JUMPS

Operand	**描述** - 见图 27-17
Threshold	跳转条件阈值（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 27.5.2.1 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。

27.5.2.7 HALT - 结束程序

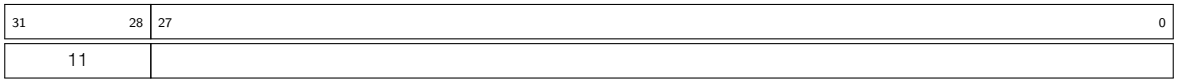


图 27-18. 指令类型 - HALT

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。

27.5.2.8 WAKE - 唤醒芯片

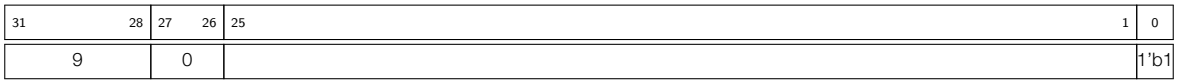


图 27-19. 指令类型 - WAKE

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。

27.5.2.9 WAIT - 等待若干个周期

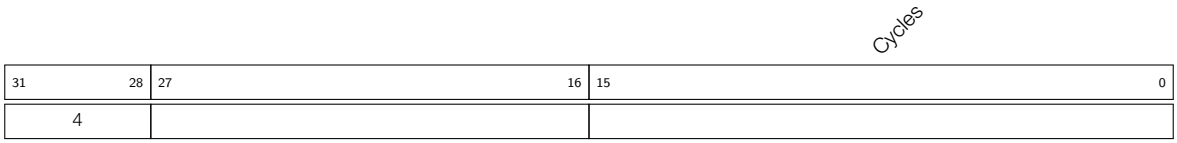


图 27-20. 指令类型 - WAIT

Operand	**描述** - 见图 27-20
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。

27.5.2.10 TSENS - 对温度传感器进行测量

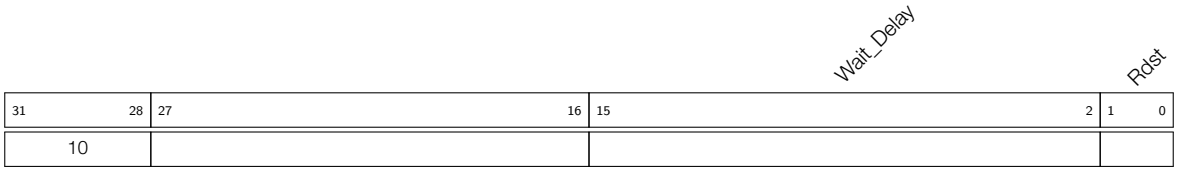


图 27-21. 指令类型 - TSENS

Operand	**描述** - 见图 27-21
Rdst	目标地址寄存器 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。

27.5.2.11 ADC - 对 ADC 进行测量

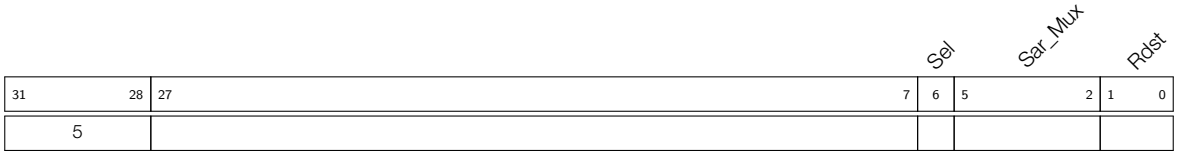


图 27-22. 指令类型 - ADC

Operand	**描述** - 见图 27-22
Rdst	目标地址寄存器 R[0-3]，将存储测量结果。
Sar_Mux	使能 SAR ADC 管脚 [Sar_Mux - 1]，见表 142。
Sel	选择 ADC。0：选择 SAR ADC1；1：选择 SAR ADC2，具体可见表 142。

表 142: 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	

管脚名 / 信号名 / GPIO	*Sar_Mux*	ADC 选择（*Sel*）
GPIO11	1	*Sel* = 1，选择 SAR ADC2
GPIO12	2	
GPIO13	3	
GPIO14	4	
XTAL_32k_P	5	
XTAL_32k_N	6	
DAC1	7	
DAC2	8	
GPIO19	9	
GPIO20	10	

27.5.2.12 REG_RD - 从外围寄存器读取

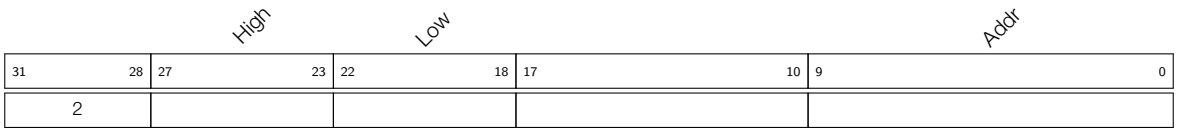


图 27-23. 指令类型 - REG_RD

Operand	**描述** - 见图 27-23
Addr	外围设备寄存器地址，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 外围设备中的寄存器。ULP 协处理器可通过相同寄存器在外设总线上的地址 (addr_peribus1)，计算外围寄存器的地址，具体方式见下：

  addr_ulp = (addr_peribus1 - 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围寄存器空间的 4096 字节，包括 DR_REG_RTCCNTL_BASE、DR_REG_RTCIO_BASE、DR_REG_SENS_BASE 及 DR_REG_RTC_I2C_BASE 区域。

27.5.2.13 REG_WR - 写入外围寄存器

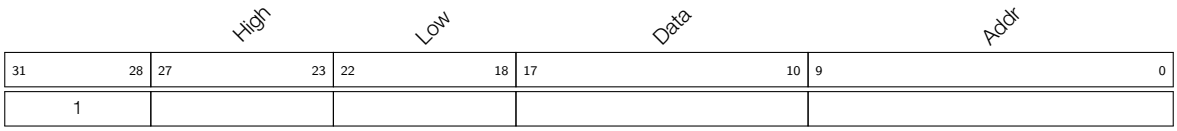


图 27-24. 指令类型 - REG_WR

Operand	**描述** - 见图 27-24
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 27.5.2.12 节。

27.6 ULP-RISC-V

27.6.1 ULP-RISC-V 主要特性

- 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

27.6.2 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 143: 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 位的有符号整数相乘，返回结果高 32 位
	MULHU	66	两个 32 位的无符号整数相乘，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 位的无符号整数相除，返回商
	REM	34	两个 32 位的有符号整数相除，返回余数
	REMU	34	两个 32 位的无符号整数相除，返回余数

27.6.3 ULP-RISC-V 中断

ESP32-S2 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 144 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

表 144: ULP-RISC-V 的中断列表

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，但只能选择电平类触发。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 27.7.4。

27.7 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。

27.7.1 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。

27.7.2 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。

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. 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。

27.7.3 使用 RTC I²C

27.7.3.1 I2C 指令编码格式

RTC I²C 的指令编码与 I²C0/I²C1 的编码方式保持一致（见 I²C 控制器章节的 CMD_Controller）。不同的地方在于，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。

27.7.3.2 I2C_RD - I²C 读取流程

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_CTRL[18:11] 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 27-25：

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。

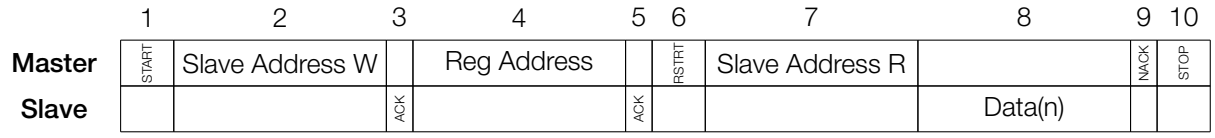


图 27-25. I²C 读操作

注意：

RTC I²C 控制器外围设备会对 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。

27.7.3.3 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_CTRL[18:11] 配置从机寄存器地址；
- 使用寄存器 SENS_SAR_I2C_CTRL[26:19]) 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 (SENS_SAR_I2C_CTRL[26:19])。

I2C_WR 指令的执行步骤如下，见图 27-26：

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。

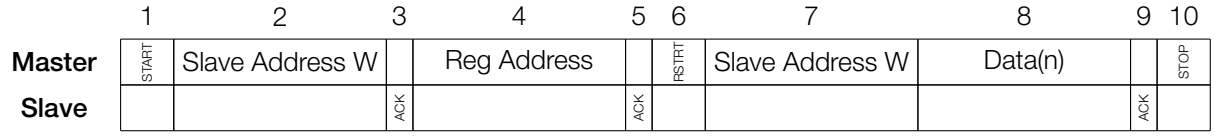


图 27-26. I²C 写操作

27.7.3.4 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。

27.7.4 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。

27.8 基地址

27.8.1 协处理器基地址

用户可以通过两个不同的寄存器基地址访问协处理器相关的寄存器，如表 145 所示：

表 145: 协处理器基地址

模块名	访问方式	基地址
ULP (ALWAYS_ON)	PeriBUS1	0x3F408000
	PeriBUS2	0x60008000
ULP (RTC_PERI)	PeriBUS1	0x3F408800
	PeriBUS2	0x60008800

其中：

- ULP(ALWAYS_ON) 为非掉电寄存器，寄存器不会随着 RTC_PERI 的电源域（见低功耗管理章节）掉电而复位。
- ULP(RTC_PERI) 处于 RTC_PERI 电源域下，如果 RTC_PERI 的电源域（见低功耗管理章节）掉电，寄存器值将会被复位。

27.8.2 RTC I²C 基地址

与 RTC I²C 相关的寄存器有两组：RTC_PERI 与 I²C，并且都处于 RTC_PERI 的电源域下。用户都可以通过两个不同的寄存器基地址进行访问，如表 146 所示：

表 146: RTC I²C 基地址

模块名	访问方式	基地址
RTC I²C (RTC_PERI)	PeriBUS1	0x3F408800
	PeriBUS2	0x60008800
RTC I²C (I²C)	PeriBUS1	0x3F408C00
	PeriBUS2	0x60008C00

27.9 寄存器列表

请注意，表中的地址为相对于基地址的地址偏移量（相对地址）。请参考章节 27.8 获取基地址相关信息。

27.9.1 ULP (ALWAYS_ON) 寄存器列表

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	0x00F8	不定
RTC_CNTL_ULP_CP_TIMER_1_REG	配置定时器睡眠周期	0x0130	读/写
ULP-FSM 寄存器			
RTC_CNTL_ULP_CP_CTRL_REG	ULP-FSM 配置寄存器	0x00FC	读/写
ULP-RISC-V 寄存器			
RTC_CNTL_COCPU_CTRL_REG	ULP-RISC-V 配置寄存器	0x0100	不定

27.9.2 ULP (RTC_PERI) 寄存器列表

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	0x0128	只读

名称	描述	地址	访问
SENS_SAR_COCPU_INT_ENA_REG	ULP-RISC-V 的中断使能位	0x012C	读/写
SENS_SAR_COCPU_INT_ST_REG	ULP-RISC-V 的中断状态位	0x0130	只读
SENS_SAR_COCPU_INT_CLR_REG	ULP-RISC-V 的中断清零位	0x0134	只写

27.9.3 RTC I²C (RTC_PERI) 寄存器列表

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

27.9.4 RTC I²C (I²C) 寄存器列表

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
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

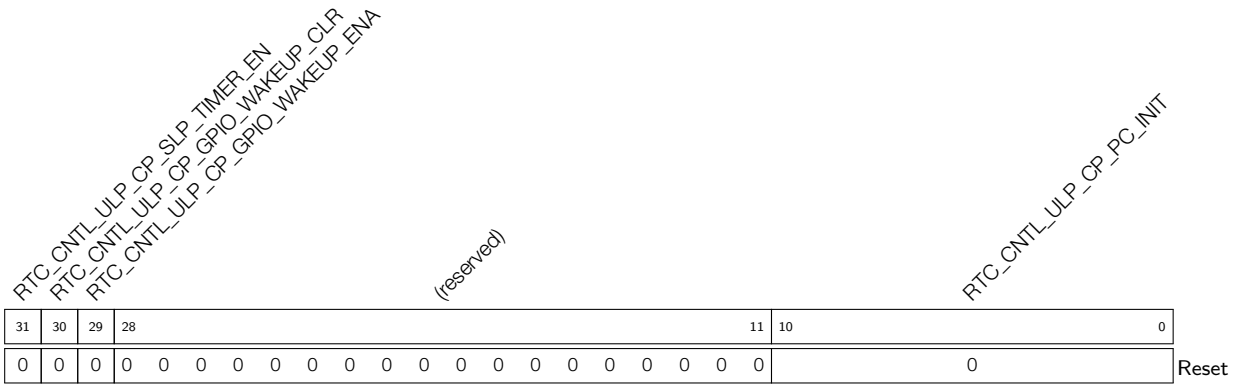
名称	描述	地址	访问
RTC_I2C_CMD5_REG	RTC I²C 命令 5	0x004C	不定
RTC_I2C_CMD6_REG	RTC I²C 命令 6	0x0050	不定
RTC_I2C_CMD7_REG	RTC I²C 命令 7	0x0054	不定
RTC_I2C_CMD8_REG	RTC I²C 命令 8	0x0058	不定
RTC_I2C_CMD9_REG	RTC I²C 命令 9	0x005C	不定
RTC_I2C_CMD10_REG	RTC I²C 命令 10	0x0060	不定
RTC_I2C_CMD11_REG	RTC I²C 命令 11	0x0064	不定
RTC_I2C_CMD12_REG	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

27.10 寄存器

请注意，表中的地址为相对于基地址的地址偏移量（相对地址）。请参考章节 27.8 获取基地址相关信息。

27.10.1 ULP (ALWAYS_ON) 寄存器

Register 27.1: RTC_CNTL_ULP_CP_TIMER_REG (0x00F8)



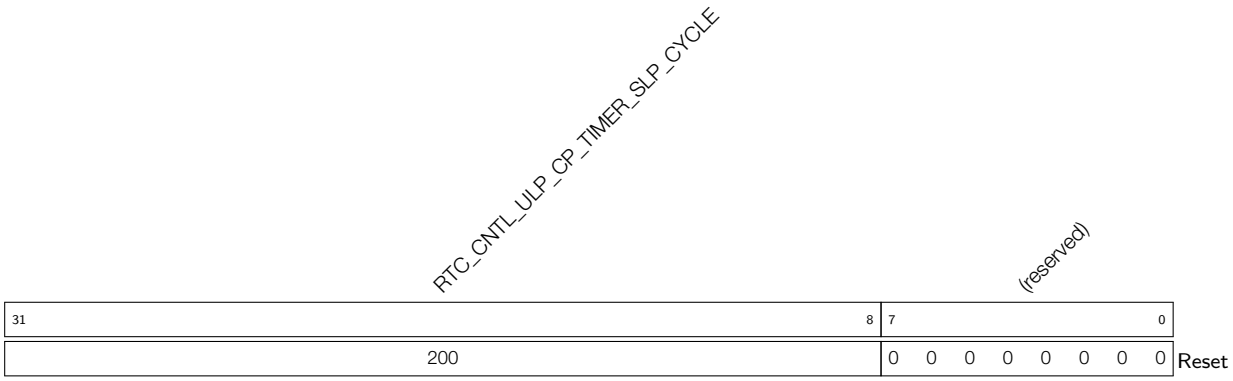
RTC_CNTL_ULP_CP_PC_INIT ULP 协处理器 PC 初始地址。（读/写）

RTC_CNTL_ULP_CP_GPIO_WAKEUP_ENA 开启使用 RTC GPIO 唤醒 ULP 协处理器的选项。（读/写）

RTC_CNTL_ULP_CP_GPIO_WAKEUP_CLR 关闭使用 RTC GPIO 唤醒 ULP 协处理器的选项。（只写）

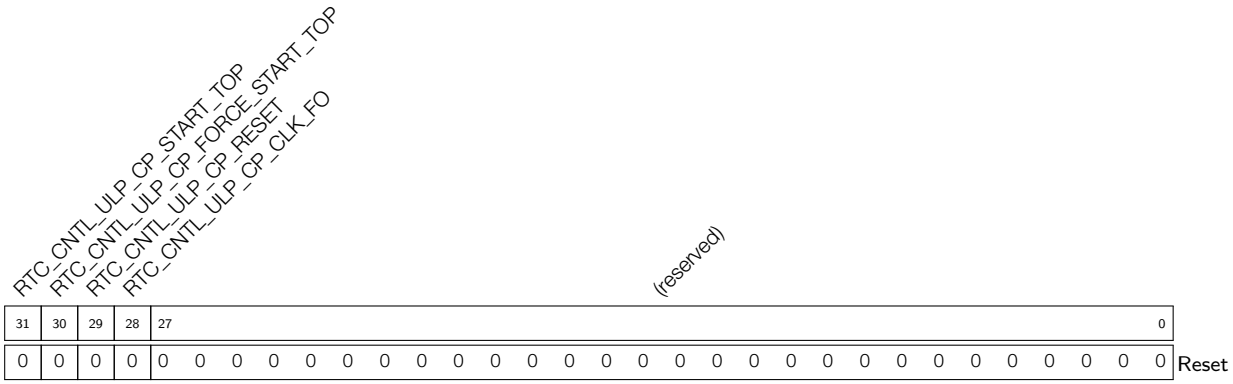
RTC_CNTL_ULP_CP_SLP_TIMER_EN ULP 协处理器定时器使能位。0：关闭硬件定时器；1：使能硬件定时器。（读/写）

Register 27.2: RTC_CNTL_ULP_CP_TIMER_1_REG (0x0130)



RTC_CNTL_ULP_CP_TIMER_SLP_CYCLE 设置 ULP 协处理器定时器的睡眠周期。（读/写）

Register 27.3: RTC_CNTL_ULP_CP_CTRL_REG (0x00FC)



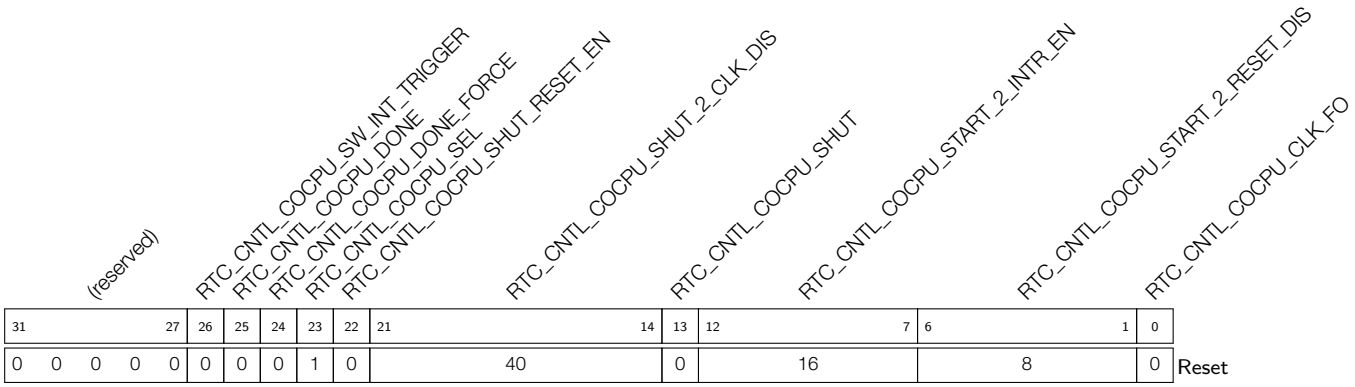
RTC_CNTL_ULP_CP_CLK_FO 强制使能 ULP-FSM 时钟。（读/写）

RTC_CNTL_ULP_CP_RESET ULP-FSM 时钟软件重置。（读/写）

RTC_CNTL_ULP_CP_FORCE_START_TOP 写入 1 时，ULP-FSM 由软件启动。（读/写）

RTC_CNTL_ULP_CP_START_TOP 写入 1 启动 ULP-FSM。（读/写）

Register 27.4: RTC_CNTL_COCPU_CTRL_REG (0x0100)



RTC_CNTL_COCPU_CLK_FO 强制使能 ULP-RISC-V 时钟。(读/写)

RTC_CNTL_COCPU_START_2_RESET_DIS 从 ULP-RISC-V 启动到下拉复位的时间。(读/写)

RTC_CNTL_COCPU_START_2_INTR_EN 从 ULP-RISC-V 启动到发出 RISCV_START_INT 中断的时间。(读/写)

RTC_CNTL_COCPU_SHUT 关闭 ULP-RISC-V。(读/写)

RTC_CNTL_COCPU_SHUT_2_CLK_DIS 关闭 ULP-RISC-V 至关闭时钟之间的延迟时间。(读/写)

RTC_CNTL_COCPU_SHUT_RESET_EN 复位 ULP-RISC-V。(读/写)

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。(读/写)

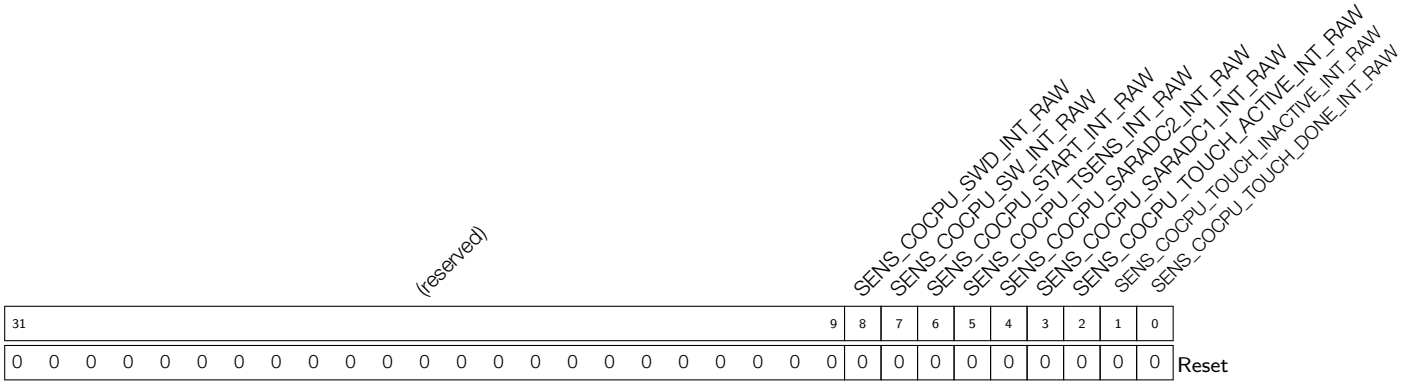
RTC_CNTL_COCPU_DONE_FORCE 0：选择 ULP-FSM 的完成信号；1：选择 ULP-RISC-V 的完成信号。(读/写)

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。(读/写)

RTC_CNTL_COCPU_SW_INT_TRIGGER 触发 ULP-RISC-V 寄存器中断。(只写)

27.10.2 ULP (RTC_PERI) 寄存器

Register 27.5: SENS_ SAR_COCPU_INT_RAW_REG (0x0128)



SENS_COCPU_TOUCH_DONE_INT_RAW TOUCH_DONE_INT 的原始中断位。(只读)

SENS_COCPU_TOUCH_INACTIVE_INT_RAW TOUCH_INACTIVE_INT 的原始中断位。(只读)

SENS_COCPU_TOUCH_ACTIVE_INT_RAW TOUCH_ACTIVE_INT 的原始中断位。(只读)

SENS_COCPU_SARADC1_INT_RAW SARADC1_DONE_INT 的原始中断位。(只读)

SENS_COCPU_SARADC2_INT_RAW SARADC2_DONE_INT 的原始中断位。(只读)

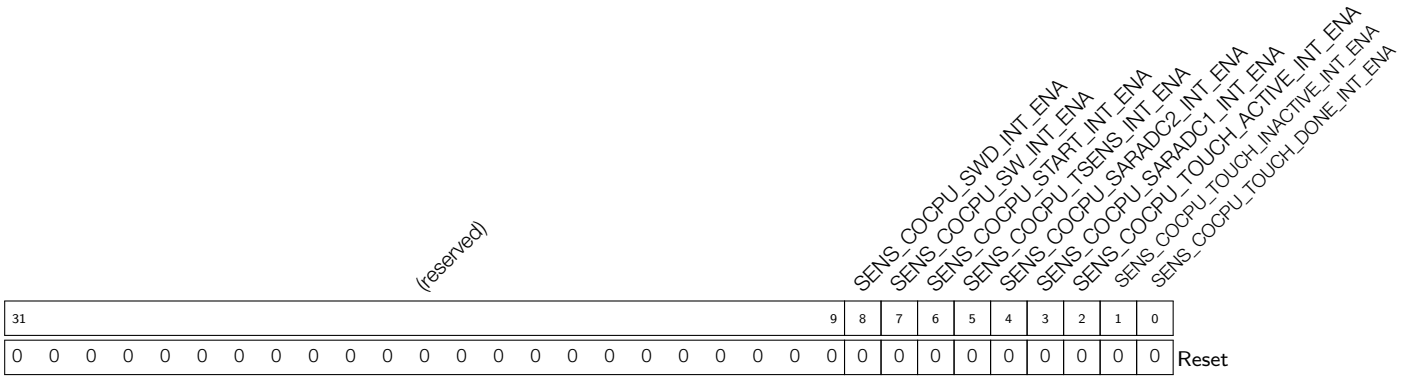
SENS_COCPU_TSENS_INT_RAW TSENS_DONE_INT 的原始中断位。(只读)

SENS_COCPU_START_INT_RAW RISCV_START_INT 的原始中断位。(只读)

SENS_COCPU_SW_INT_RAW SW_INT 的原始中断位。(只读)

SENS_COCPU_SWD_INT_RAW SWD_INT 的原始中断位。(只读)

Register 27.6: SENS_SAR_COCPU_INT_ENA_REG (0x012C)



SENS_COCPU_TOUCH_DONE_INT_ENA TOUCH_DONE_INT 的中断使能位。(读/写)

SENS_COCPU_TOUCH_INACTIVE_INT_ENA TOUCH_INACTIVE_INT 的中断使能位。(读/写)

SENS_COCPU_TOUCH_ACTIVE_INT_ENA TOUCH_ACTIVE_INT 的中断使能位。(读/写)

SENS_COCPU_SARADC1_INT_ENA SARADC1_DONE_INT 的中断使能位。(读/写)

SENS_COCPU_SARADC2_INT_ENA SARADC2_DONE_INT 的中断使能位。(读/写)

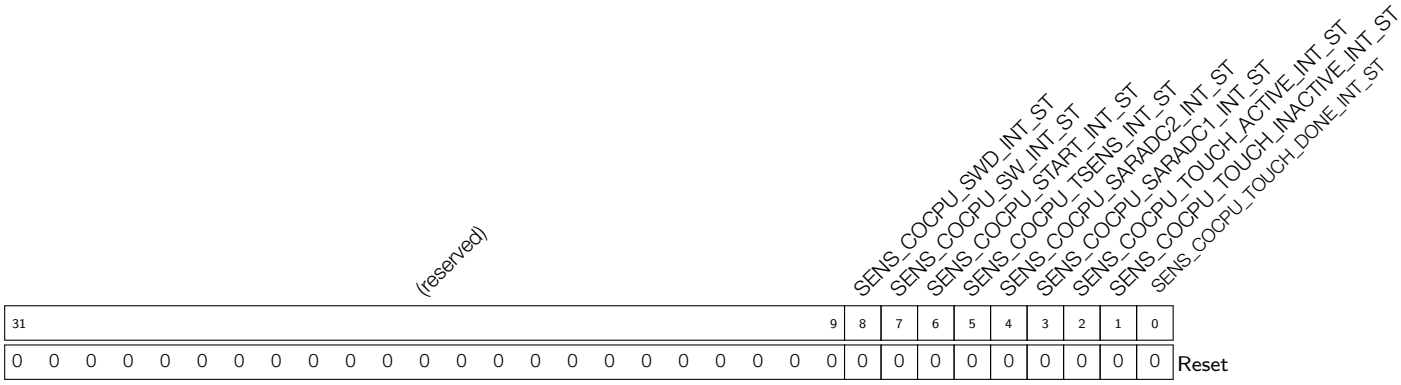
SENS_COCPU_TSENS_INT_ENA TSENS_DONE_INT 的中断使能位。(读/写)

SENS_COCPU_START_INT_ENA RISCV_START_INT 的中断使能位。(读/写)

SENS_COCPU_SW_INT_ENA SW_INT 的中断使能位。(读/写)

SENS_COCPU_SWD_INT_ENA SWD_INT 的中断使能位。(读/写)

Register 27.7: SENS_SAR_COCPU_INT_ST_REG (0x0130)



SENS_COCPU_TOUCH_DONE_INT_ST TOUCH_DONE_INT 的中断状态位（只读）

SENS_COCPU_TOUCH_INACTIVE_INT_ST TOUCH_INACTIVE_INT 的中断状态位（只读）

SENS_COCPU_TOUCH_ACTIVE_INT_ST TOUCH_ACTIVE_INT 的中断状态位（只读）

SENS_COCPU_SARADC1_INT_ST SARADC1_DONE_INT 的中断状态位（只读）

SENS_COCPU_SARADC2_INT_ST SARADC2_DONE_INT 的中断状态位（只读）

SENS_COCPU_TSENS_INT_ST TSENS_DONE_INT 的中断状态位（只读）

SENS_COCPU_START_INT_ST RISCV_START_INT 的中断状态位（只读）

SENS_COCPU_SW_INT_ST SW_INT 的中断状态位（只读）

SENS_COCPU_SWD_INT_ST SWD_INT 的中断状态位（只读）

Register 27.8: SENS_SAR_COCPU_INT_CLR_REG (0x0134)

(reserved)	SENS_COCPU_SWD_INT_CLR	SENS_COCPU_SW_INT_CLR	SENS_COCPU_START_INT_CLR	SENS_COCPU_TSENS_INT_CLR	SENS_COCPU_SARADC2_INT_CLR	SENS_COCPU_SARADC1_INT_CLR	SENS_COCPU_TOUCH_ACTIVE_INT_CLR	SENS_COCPU_TOUCH_INACTIVE_INT_CLR	SENS_COCPU_TOUCH_DONE_INT_CLR	
31 … 9	8	7	6	5	4	3	2	1	0	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0	0	0	0	0	0	0	0	0	Reset

SENS_COCPU_TOUCH_DONE_INT_CLR TOUCH_DONE_INT 的中断清零位（只写）

SENS_COCPU_TOUCH_INACTIVE_INT_CLR TOUCH_INACTIVE_INT 的中断清零位（只写）

SENS_COCPU_TOUCH_ACTIVE_INT_CLR TOUCH_ACTIVE_INT 的中断清零位（只写）

SENS_COCPU_SARADC1_INT_CLR SARADC1_DONE_INT 的中断清零位（只写）

SENS_COCPU_SARADC2_INT_CLR SARADC2_DONE_INT 的中断清零位（只写）

SENS_COCPU_TSENS_INT_CLR TSENS_DONE_INT 的中断清零位（只写）

SENS_COCPU_START_INT_CLR RISCV_START_INT 的中断清零位（只写）

SENS_COCPU_SW_INT_CLR SW_INT 的中断清零位（只写）

SENS_COCPU_SWD_INT_CLR SWD_INT 的中断清零位（只写）

27.10.3 RTC I²C (RTC_PERI) 寄存器

Register 27.9: SENS_SAR_I2C_CTRL_REG (0x0058)

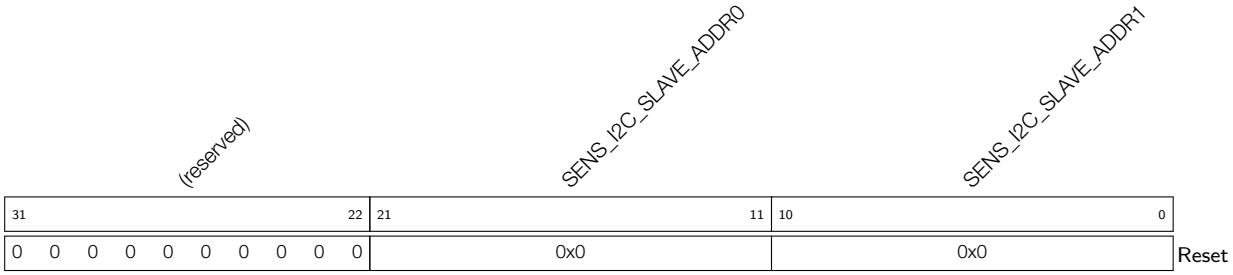
(reserved)	SENS_SAR_I2C_START_FORCE	SENS_SAR_I2C_START	SENS_SAR_I2C_CTRL	
31 30	29	28	27 … 0	
0 0	0	0	0	Reset

SENS_SAR_I2C_CTRL RTC I²C 控制数据，仅当 SENS_SAR_I2C_START_FORCE = 1 时有效。（读/写）

SENS_SAR_I2C_START 启动 RTC I²C，仅当 SENS_SAR_I2C_START_FORCE = 1 时有效。（读/写）

SENS_SAR_I2C_START_FORCE RTC I²C 启动模式。0：由 FSM 启动；1：由软件启动。（读/写）

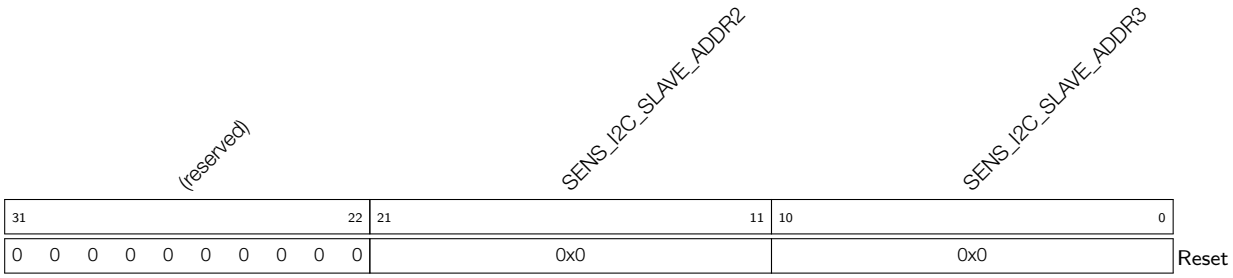
Register 27.10: SENS_SAR_SLAVE_ADDR1_REG (0x0040)



SENS_I2C_SLAVE_ADDR1 RTC I²C 从机地址 1。(读/写)

SENS_I2C_SLAVE_ADDR0 RTC I²C 从机地址 0。(读/写)

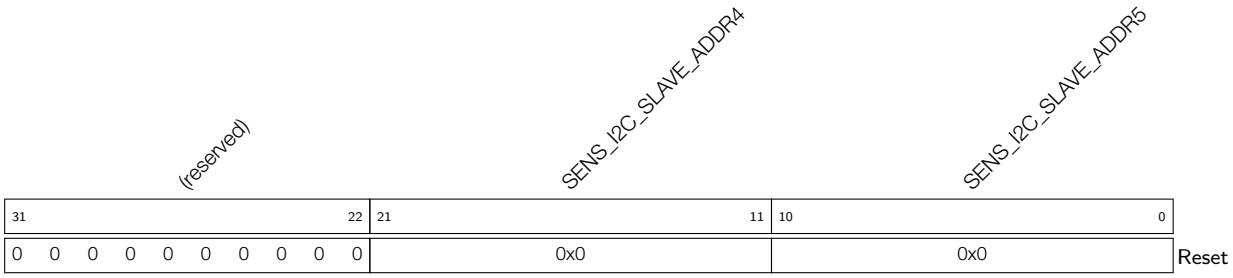
Register 27.11: SENS_SAR_SLAVE_ADDR2_REG (0x0044)



SENS_I2C_SLAVE_ADDR3 RTC I²C 从机地址 3。(读/写)

SENS_I2C_SLAVE_ADDR2 RTC I²C 从机地址 2。(读/写)

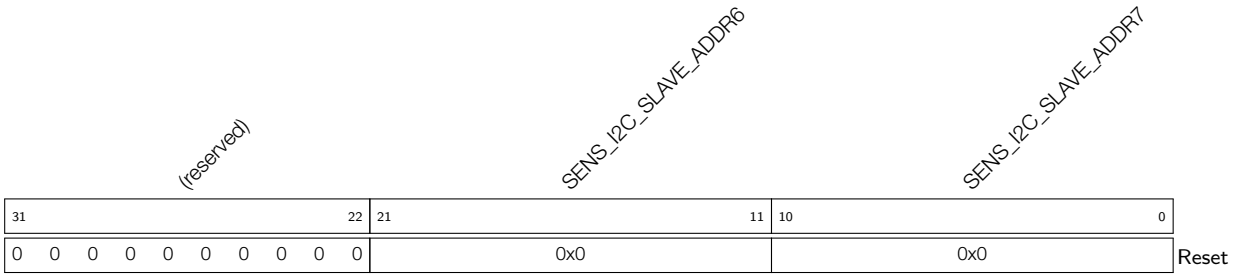
Register 27.12: SENS_SAR_SLAVE_ADDR3_REG (0x0048)



SENS_I2C_SLAVE_ADDR5 RTC I²C 从机地址 5。(读/写)

SENS_I2C_SLAVE_ADDR4 RTC I²C 从机地址 4。(读/写)

Register 27.13: SENS_SAR_SLAVE_ADDR4_REG (0x004C)

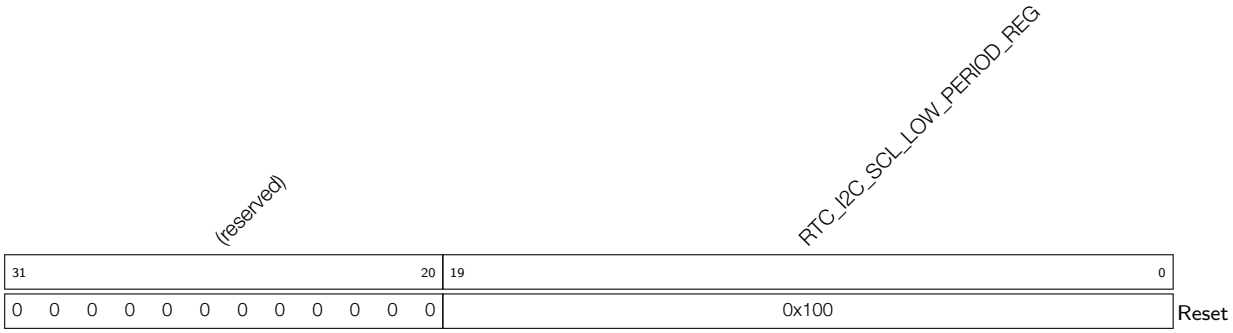


SENS_I2C_SLAVE_ADDR7 RTC I²C 从机地址 7。(读/写)

SENS_I2C_SLAVE_ADDR6 RTC I²C 从机地址 6。(读/写)

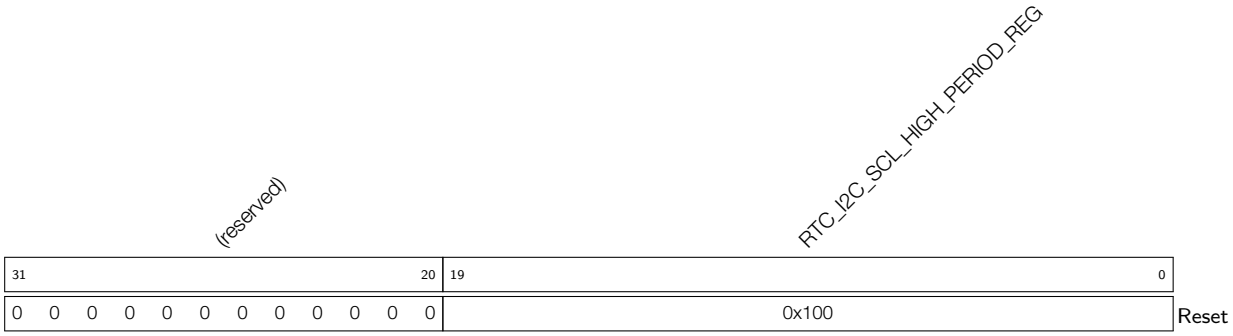
27.10.4 RTC I²C (I²C) 寄存器

Register 27.14: RTC_I2C_SCL_LOW_REG (0x0000)



RTC_I2C_SCL_LOW_PERIOD_REG 配置 SCL 低电平周期。(读/写)

Register 27.15: RTC_I2C_SCL_HIGH_REG (0x0014)

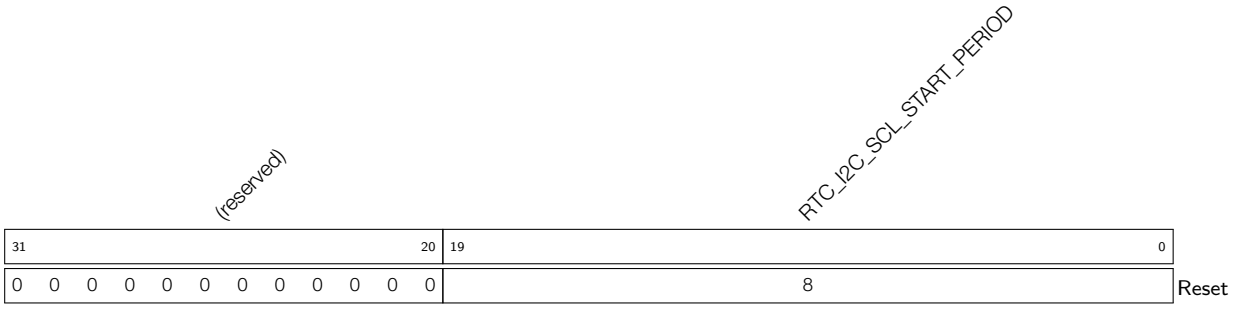


RTC_I2C_SCL_HIGH_PERIOD_REG 配置 SCL 高电平周期。(读/写)

Register 27.16: RTC_I2C_SDA_DUTY_REG (0x0018)

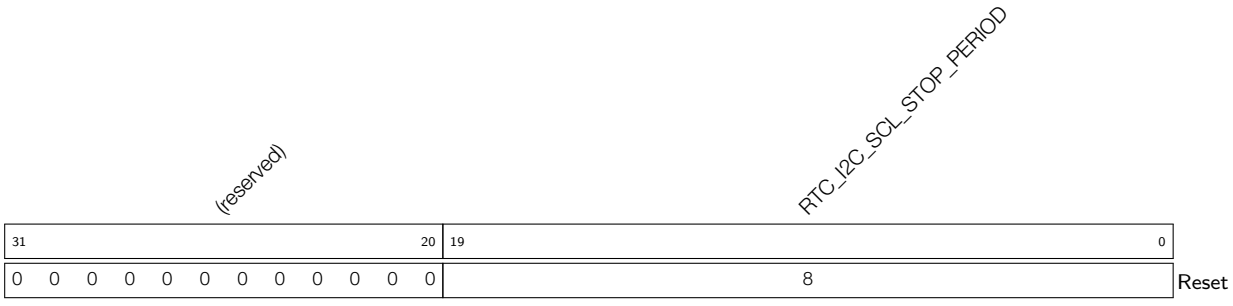
RTC_I2C_SDA_DUTY_NUM SCL 下降沿与 SDA 切换之间的时钟周期数。(读/写)

Register 27.17: 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。(读/写)

Register 27.18: 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。(读/写)

Register 27.19: 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。（读/写）

RTC_I2C_SCL_FORCE_OUT SCL 输出模式配置。0：开漏输出；1：推挽输出。（读/写）

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。（读/写）

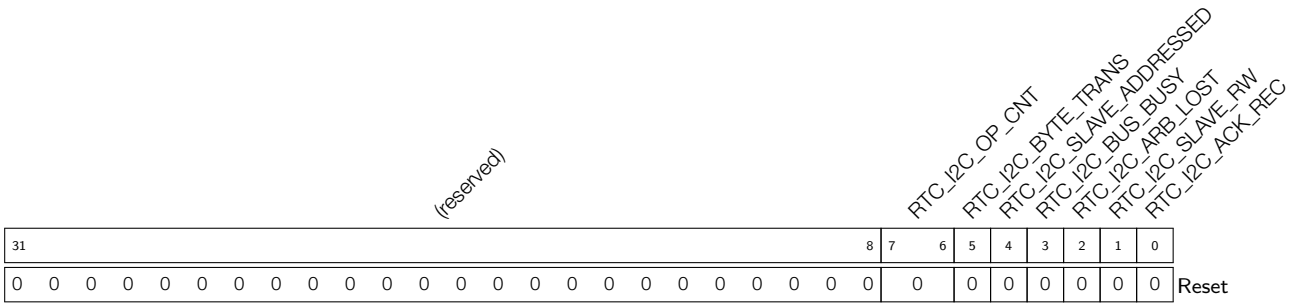
RTC_I2C_TX_LSB_FIRST 用于控制待发送数据的发送模式。0：从最高有效位发送数据；1：从最低有效位开始发送数据。（读/写）

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。（读/写）

RTC_I2C_CTRL_CLK_GATE_EN RTC I²C 控制器时钟门控。（读/写）

RTC_I2C_RESET 置位此位，RTC I²C 软件重置。（读/写）

Register 27.20: RTC_I2C_STATUS_REG (0x0008)



RTC_I2C_ACK_REC ACK 电平值。0：ACK；1：NACK。（只读）

RTC_I2C_SLAVE_RW 0：主机向从机写入数据；1：主机读取从机数据。（只读）

RTC_I2C_ARB_LOST RTC I²C 不控制 SCL 线时，该寄存器变为 1。（只读）

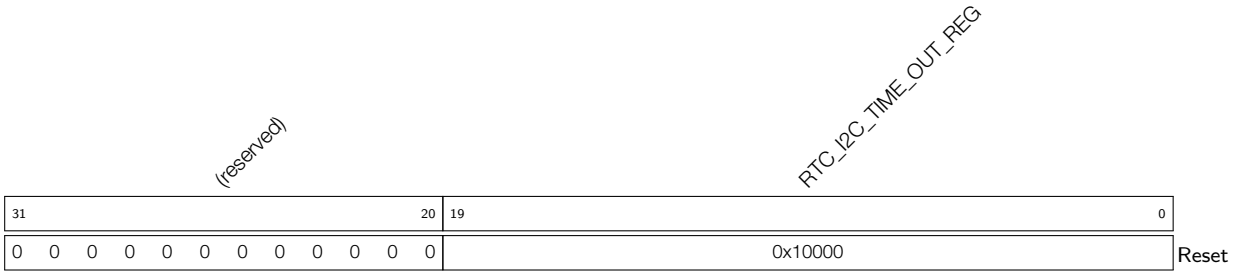
RTC_I2C_BUS_BUSY 0：RTC I²C 总线处于空闲状态；1：RTC I²C 总线正在传输数据。（只读）

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。（只读）

RTC_I2C_BYTE_TRANS 传输一个字节后，该字段变为 1。（只读）

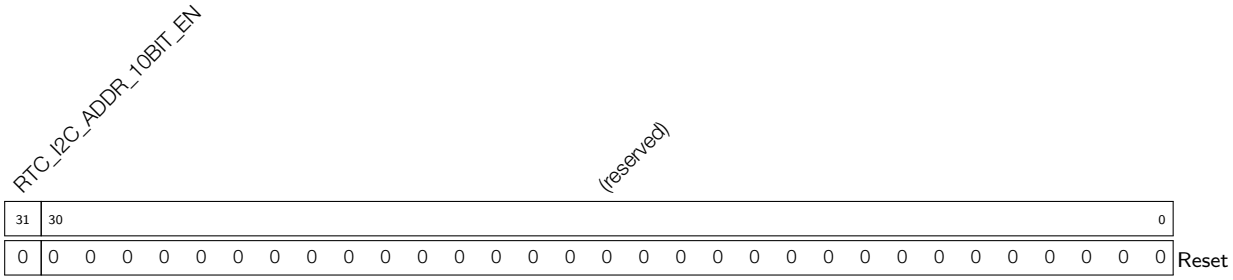
RTC_I2C_OP_CNT 表示正在执行的操作。（只读）

Register 27.21: RTC_I2C_TO_REG (0x000C)



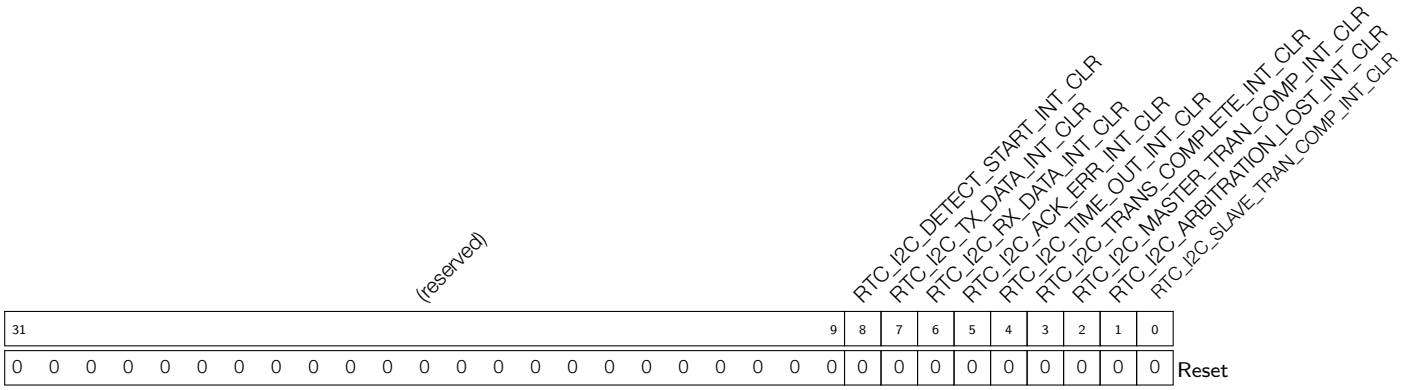
RTC_I2C_TIME_OUT_REG 超时阈值。（读/写）

Register 27.22: 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。（读/写）

Register 27.23: RTC_I2C_INT_CLR_REG (0x0024)



RTC_I2C_SLAVE_TRAN_COMP_INT_CLR RTC_I2C_SLAVE_TRAN_COMP_INT 中断清零位(只写)

RTC_I2C_ARBITRATION_LOST_INT_CLR RTC_I2C_ARBITRATION_LOST_INT 中断清零位（只写）

RTC_I2C_MASTER_TRAN_COMP_INT_CLR RTC_I2C_MASTER_TRAN_COMP_INT 中断清零位（只写）

RTC_I2C_TRANS_COMPLETE_INT_CLR RTC_I2C_TRANS_COMPLETE_INT 中断清零位（只写）

RTC_I2C_TIME_OUT_INT_CLR RTC_I2C_TIME_OUT_INT 中断清零位（只写）

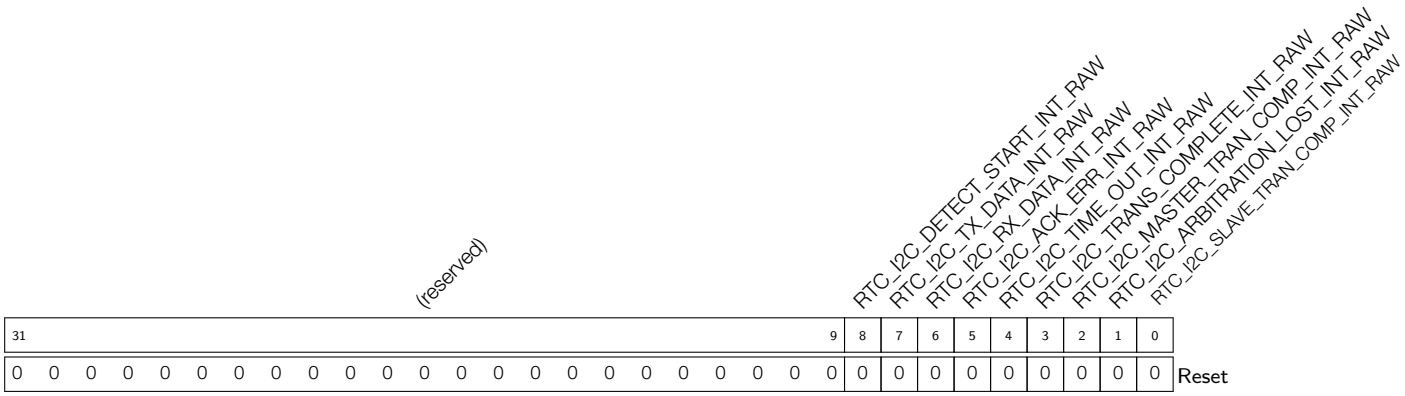
RTC_I2C_ACK_ERR_INT_CLR RTC_I2C_ACK_ERR_INT 中断清零位（只写）

RTC_I2C_RX_DATA_INT_CLR RTC_I2C_RX_DATA_INT 中断清零位（只写）

RTC_I2C_TX_DATA_INT_CLR RTC_I2C_TX_DATA_INT 中断清零位（只写）

RTC_I2C_DETECT_START_INT_CLR RTC_I2C_DETECT_START_INT 中断清零位（只写）

Register 27.24: RTC_I2C_INT_RAW_REG (0x0028)



RTC_I2C_SLAVE_TRAN_COMP_INT_RAW RTC_I2C_SLAVE_TRAN_COMP_INT 原始中断位。(只读)

RTC_I2C_ARBITRATION_LOST_INT_RAW RTC_I2C_ARBITRATION_LOST_INT 原始中断位。(只读)

RTC_I2C_MASTER_TRAN_COMP_INT_RAW RTC_I2C_MASTER_TRAN_COMP_INT 原始中断位。(只读)

RTC_I2C_TRANS_COMPLETE_INT_RAW RTC_I2C_TRANS_COMPLETE_INT 原始中断位。(只读)

RTC_I2C_TIME_OUT_INT_RAW RTC_I2C_TIME_OUT_INT 原始中断位。(只读)

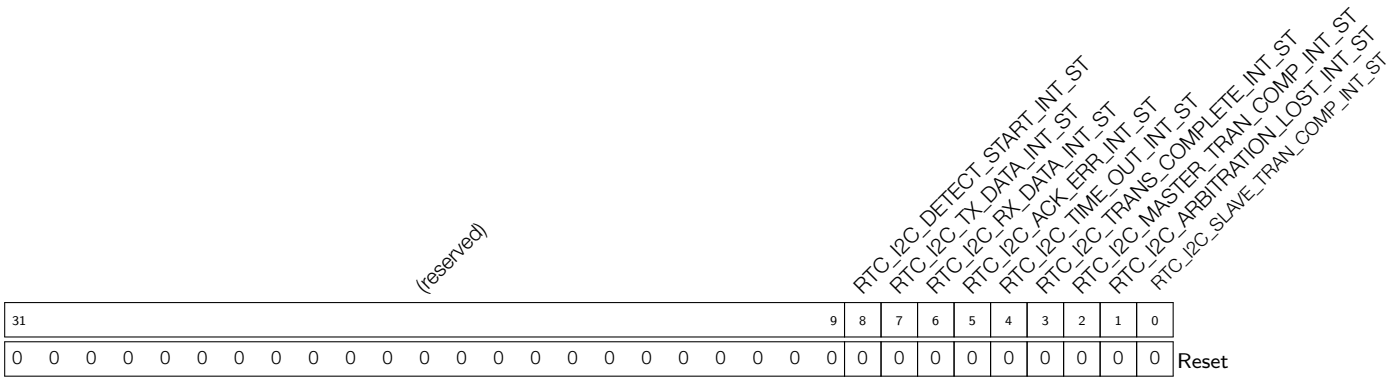
RTC_I2C_ACK_ERR_INT_RAW RTC_I2C_ACK_ERR_INT 原始中断位。(只读)

RTC_I2C_RX_DATA_INT_RAW RTC_I2C_RX_DATA_INT 原始中断位。(只读)

RTC_I2C_TX_DATA_INT_RAW RTC_I2C_TX_DATA_INT 原始中断位。(只读)

RTC_I2C_DETECT_START_INT_RAW RTC_I2C_DETECT_START_INT 原始中断位。(只读)

Register 27.25: RTC_I2C_INT_ST_REG (0x002C)



RTC_I2C_SLAVE_TRAN_COMP_INT_ST RTC_I2C_SLAVE_TRAN_COMP_INT 中断状态位。(只读)

RTC_I2C_ARBITRATION_LOST_INT_ST RTC_I2C_ARBITRATION_LOST_INT 中断状态位。(只读)

RTC_I2C_MASTER_TRAN_COMP_INT_ST RTC_I2C_MASTER_TRAN_COMP_INT 中断状态位。(只读)

RTC_I2C_TRANS_COMPLETE_INT_ST RTC_I2C_TRANS_COMPLETE_INT 中断状态位。(只读)

RTC_I2C_TIME_OUT_INT_ST RTC_I2C_TIME_OUT_INT 中断状态位。(只读)

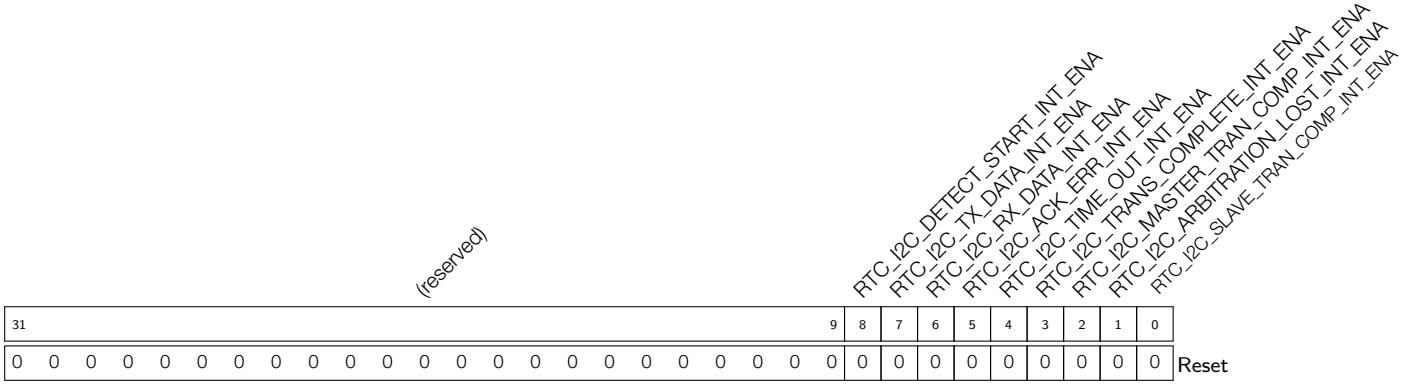
RTC_I2C_ACK_ERR_INT_ST RTC_I2C_ACK_ERR_INT 中断状态位。(只读)

RTC_I2C_RX_DATA_INT_ST RTC_I2C_RX_DATA_INT 中断状态位。(只读)

RTC_I2C_TX_DATA_INT_ST RTC_I2C_TX_DATA_INT 中断状态位。(只读)

RTC_I2C_DETECT_START_INT_ST RTC_I2C_DETECT_START_INT 中断状态位。(只读)

Register 27.26: RTC_I2C_INT_ENA_REG (0x0030)



RTC_I2C_SLAVE_TRAN_COMP_INT_ENA RTC_I2C_SLAVE_TRAN_COMP_INT 中断使能位。(读/写)

RTC_I2C_ARBITRATION_LOST_INT_ENA RTC_I2C_ARBITRATION_LOST_INT 中断使能位。(读/写)

RTC_I2C_MASTER_TRAN_COMP_INT_ENA RTC_I2C_MASTER_TRAN_COMP_INT 中断使能位。(读/写)

RTC_I2C_TRANS_COMPLETE_INT_ENA RTC_I2C_TRANS_COMPLETE_INT 中断使能位。(读/写)

RTC_I2C_TIME_OUT_INT_ENA RTC_I2C_TIME_OUT_INT 中断使能位。(读/写)

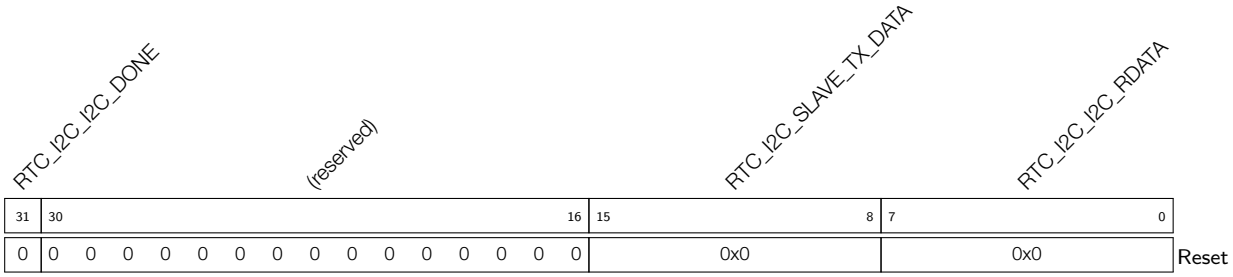
RTC_I2C_ACK_ERR_INT_ENA RTC_I2C_ACK_ERR_INT 中断使能位。(读/写)

RTC_I2C_RX_DATA_INT_ENA RTC_I2C_RX_DATA_INT 中断使能位。(读/写)

RTC_I2C_TX_DATA_INT_ENA RTC_I2C_TX_DATA_INT 中断使能位。(读/写)

RTC_I2C_DETECT_START_INT_ENA RTC_I2C_DETECT_START_INT 中断使能位。(读/写)

Register 27.27: RTC_I2C_DATA_REG (0x0034)

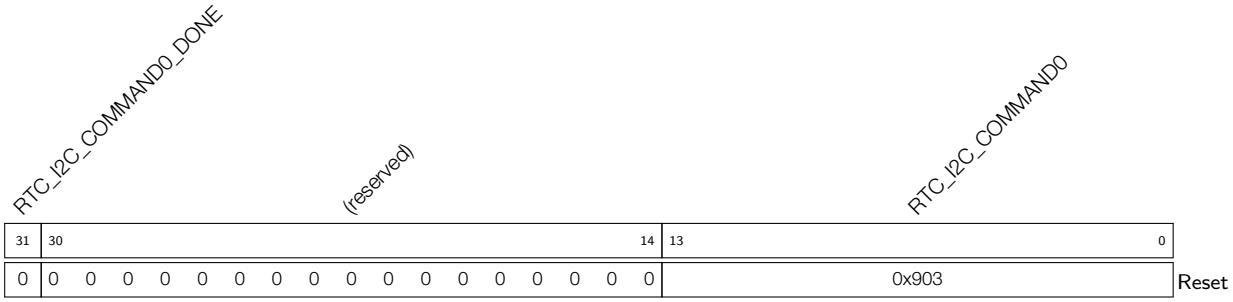


RTC_I2C_RDATA RTC I²C 主机接收的数据。(只读)

RTC_I2C_SLAVE_TX_DATA RTC I²C 从机发送的数据。(读/写)

RTC_I2C_DONE RTC I²C 数据传输结束。(只读)

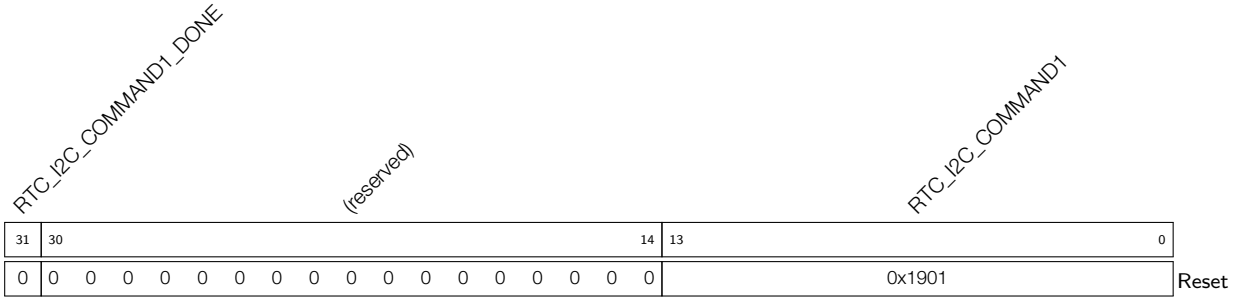
Register 27.28: RTC_I2C_CMD0_REG (0x0038)



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。(读/写)

RTC_I2C_COMMAND0_DONE 命令 0 完成时，该位翻转为高电平。(只读)

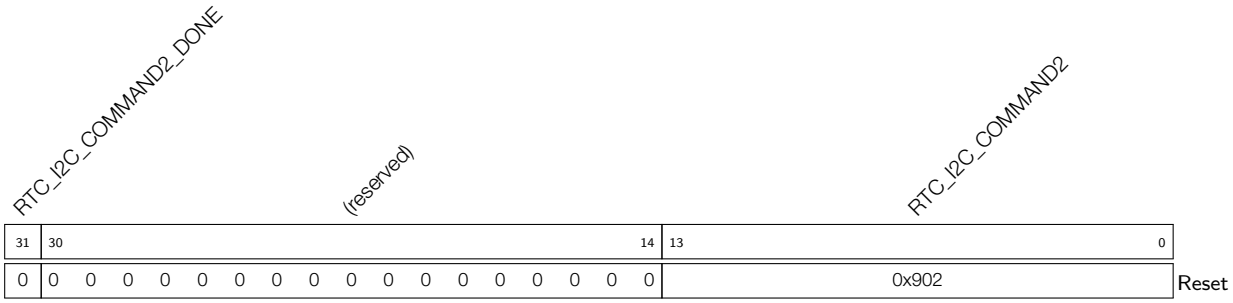
Register 27.29: RTC_I2C_CMD1_REG (0x003C)



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。(读/写)

RTC_I2C_COMMAND1_DONE 命令 1 完成时，该位翻转为高电平。(只读)

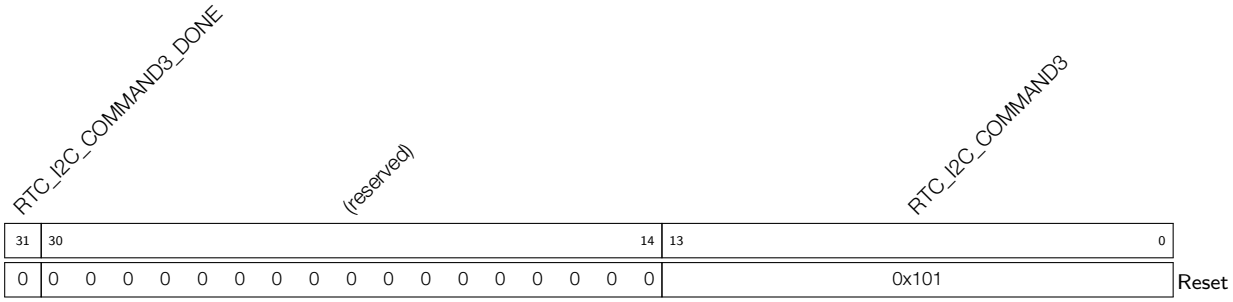
Register 27.30: RTC_I2C_CMD2_REG (0x0040)



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。(读/写)

RTC_I2C_COMMAND2_DONE 命令 2 完成时，该位翻转为高电平。(只读)

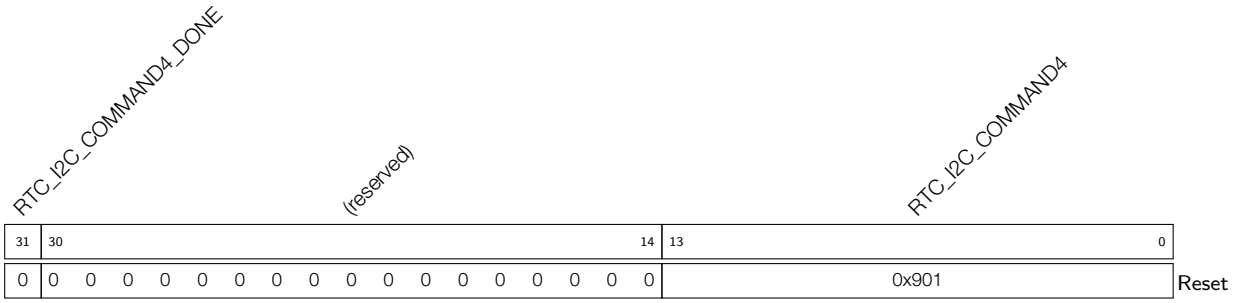
Register 27.31: RTC_I2C_CMD3_REG (0x0044)



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。(读/写)

RTC_I2C_COMMAND3_DONE 命令 3 完成时，该位翻转为高电平。(只读)

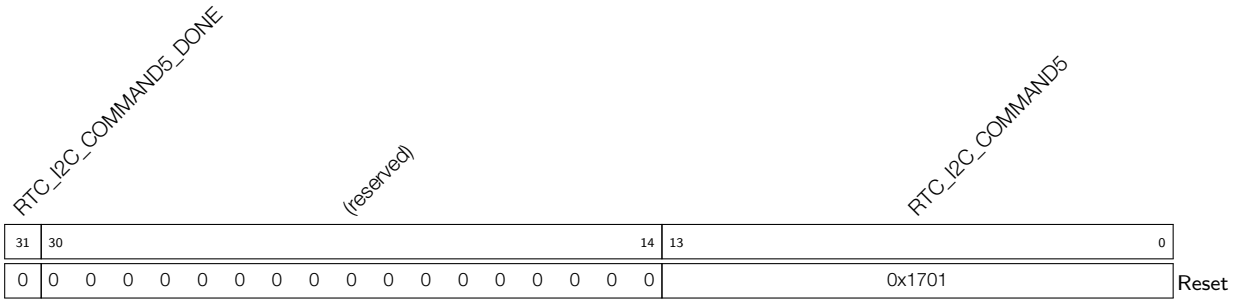
Register 27.32: RTC_I2C_CMD4_REG (0x0048)



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。(读/写)

RTC_I2C_COMMAND4_DONE 命令 4 完成时，该位翻转为高电平。(只读)

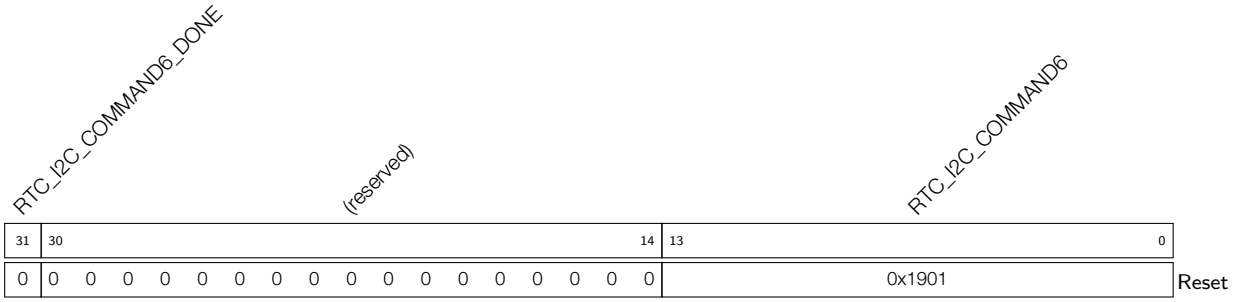
Register 27.33: RTC_I2C_CMD5_REG (0x004C)



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。(读/写)

RTC_I2C_COMMAND5_DONE 命令 5 完成时，该位翻转为高电平。(只读)

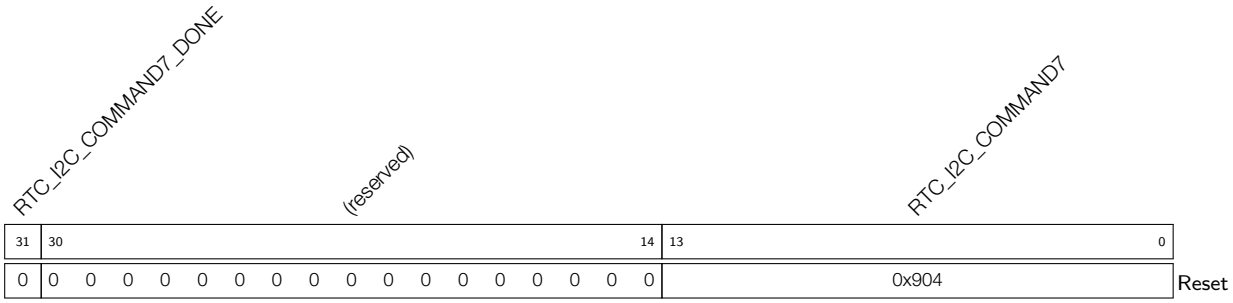
Register 27.34: RTC_I2C_CMD6_REG (0x0050)



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。(读/写)

RTC_I2C_COMMAND6_DONE 命令 6 完成时，该位翻转为高电平。(只读)

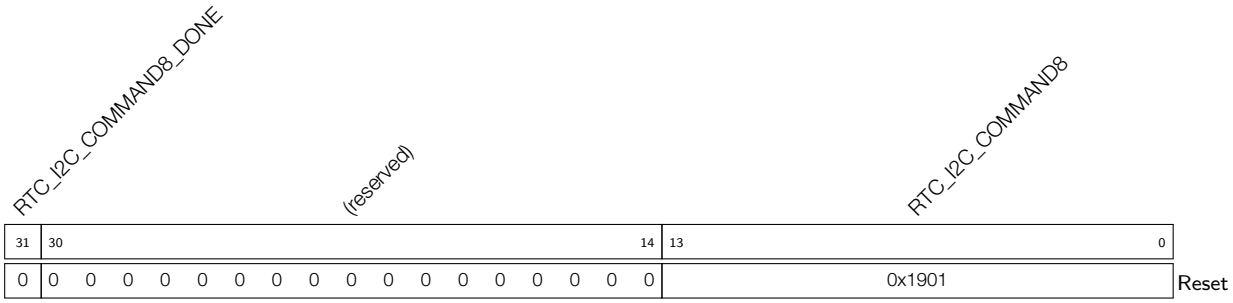
Register 27.35: RTC_I2C_CMD7_REG (0x0054)



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。(读/写)

RTC_I2C_COMMAND7_DONE 命令 7 完成时，该位翻转为高电平。(只读)

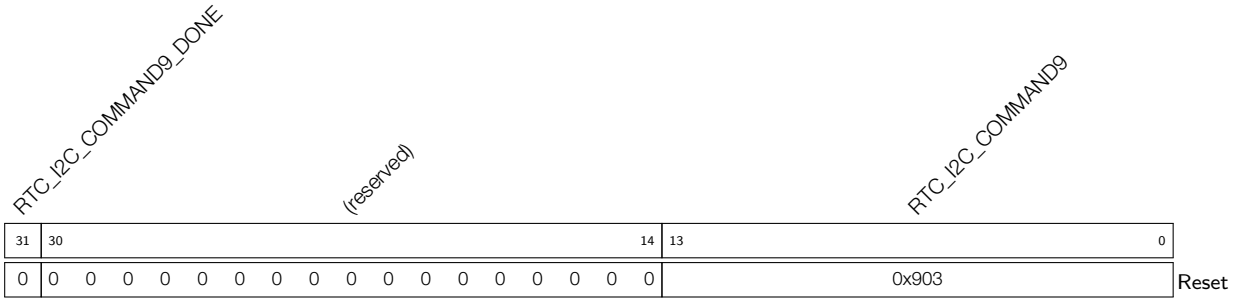
Register 27.36: RTC_I2C_CMD8_REG (0x0058)



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。(读/写)

RTC_I2C_COMMAND8_DONE 命令 8 完成时，该位翻转为高电平。(只读)

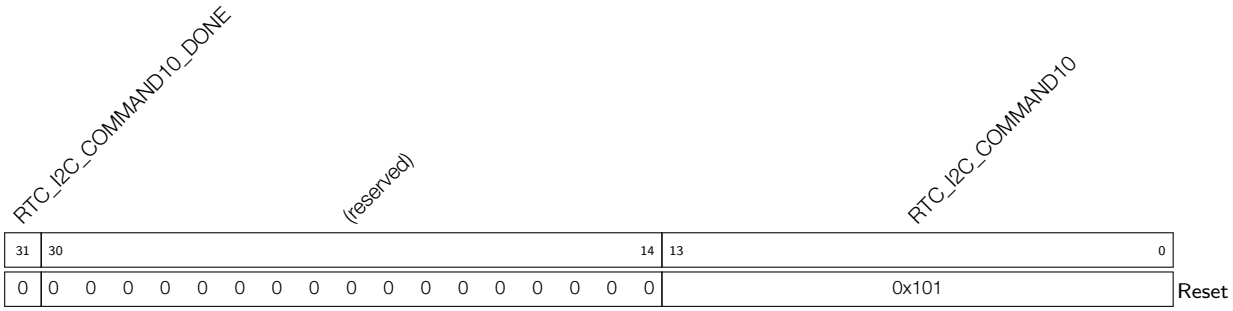
Register 27.37: RTC_I2C_CMD9_REG (0x005C)



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。(读/写)

RTC_I2C_COMMAND9_DONE 命令 9 完成时，该位翻转为高电平。(只读)

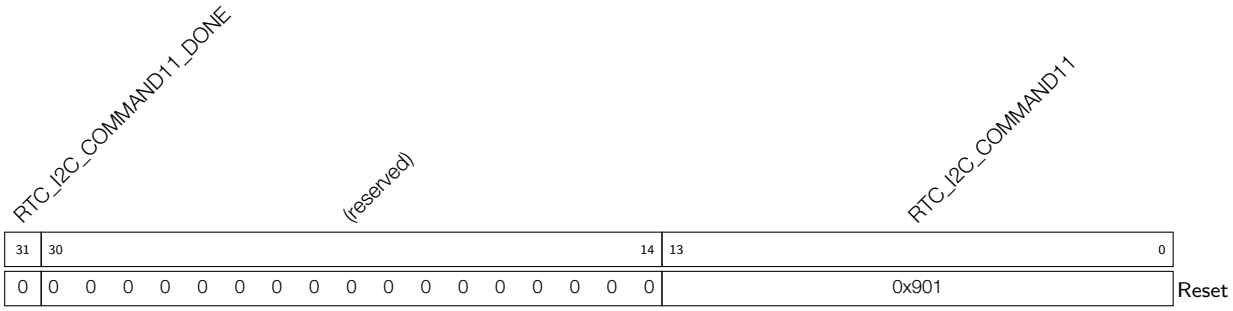
Register 27.38: RTC_I2C_CMD10_REG (0x0060)



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。(读/写)

RTC_I2C_COMMAND10_DONE 命令 10 完成时，该位翻转为高电平。(只读)

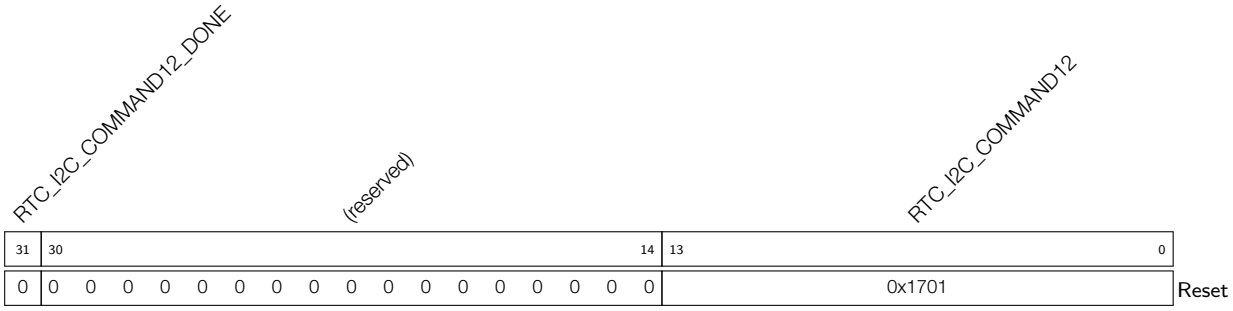
Register 27.39: RTC_I2C_CMD11_REG (0x0064)



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。(读/写)

RTC_I2C_COMMAND11_DONE 命令 11 完成时，该位翻转为高电平。(只读)

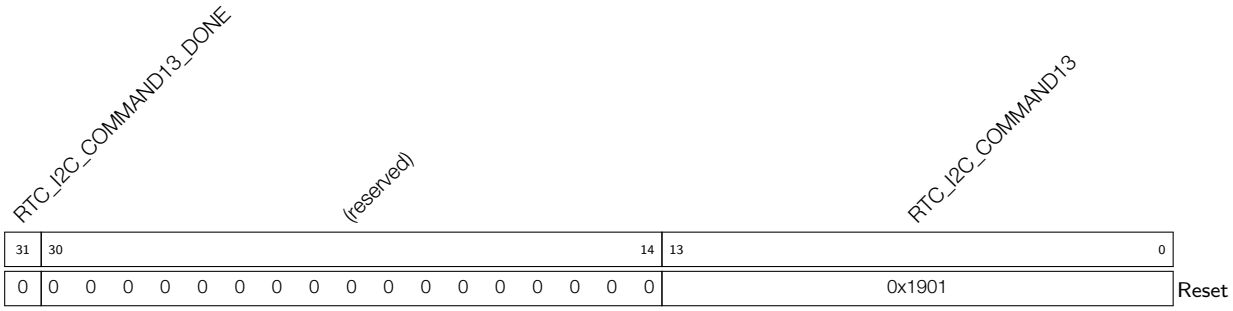
Register 27.40: RTC_I2C_CMD12_REG (0x0068)



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。(读/写)

RTC_I2C_COMMAND12_DONE 命令 12 完成时，该位翻转为高电平。(只读)

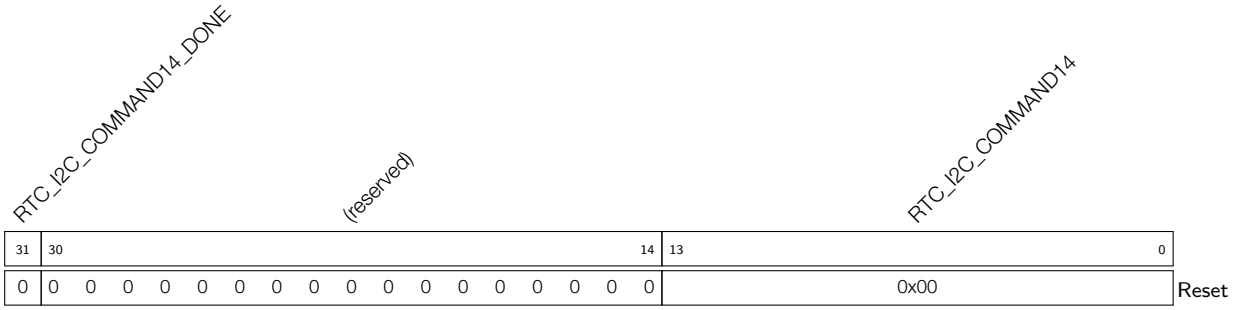
Register 27.41: RTC_I2C_CMD13_REG (0x006C)



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。(读/写)

RTC_I2C_COMMAND13_DONE 命令 13 完成时，该位翻转为高电平。(只读)

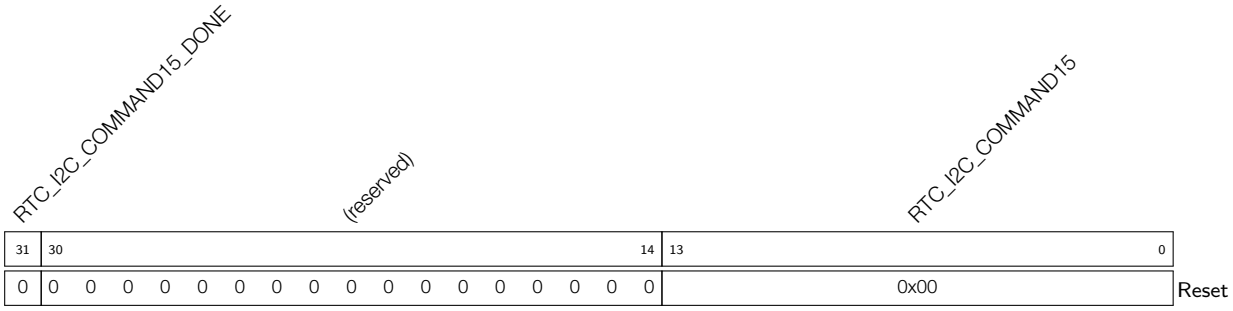
Register 27.42: RTC_I2C_CMD14_REG (0x0070)



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。(读/写)

RTC_I2C_COMMAND14_DONE 命令 14 完成时，该位翻转为高电平。(只读)

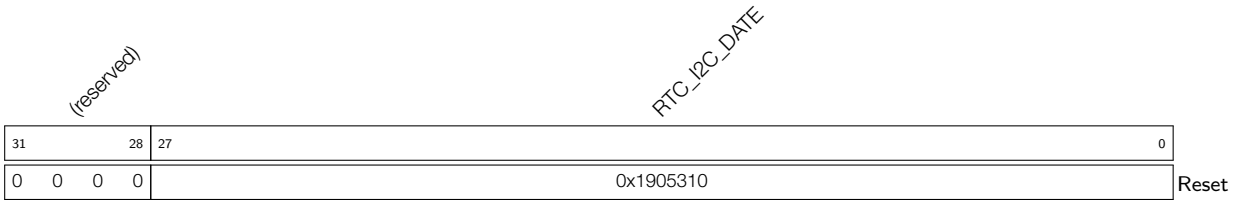
Register 27.43: RTC_I2C_CMD15_REG (0x0074)



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。（读/写）

RTC_I2C_COMMAND15_DONE 命令 15 完成时，该位翻转为高电平。（只读）

Register 27.44: RTC_I2C_DATE_REG (0x00FC)



RTC_I2C_DATE 版本控制寄存器。（读/写）

28. 低功耗管理

28.1 概述

ESP32-S2 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-S2 定义了五种最常见的电源域设置组合，对应 5 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-S2 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。

28.2 主要特性

ESP32-S2 的低功耗管理具有以下主要特性：

- 各功耗模式下均支持超低功耗协处理器
- 5 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)
- 8 个 32 位保留寄存器 (retention register)
- 支持 RTC Boot 功能，用于快速唤醒

在本章节中，我们将首先介绍 ESP32-S2 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。

28.3 功能描述

ESP32-S2 的低功耗管理具有以下主要特性：

- 功耗管理单元 (PMU)：控制向模拟、RTC 和数字类电源域的供电；
- 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 27 超低功耗协处理器。
  - 触摸传感器定时器：在预设时间唤醒触摸传感器。
- 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，不受芯片掉电影响，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，不受芯片掉电影响，可用作低功耗模式下的唤醒源（详见第 28.4.4 节），也作为正常 GPIO 使用（详见 5 *IO MUX 和 GPIO 交换矩阵* 章节）。
- RTC 慢速内存：支持 8 KB SRAM，可用作保留内存或存储 ULP 指令和数据内存。
- RTC 快速内存：支持 8 KB SRAM，可用作保留内存。
- 调压器：调节向不同电源域的供电。

ESP32-S2 低功耗管理的原理图可见图 28-1。

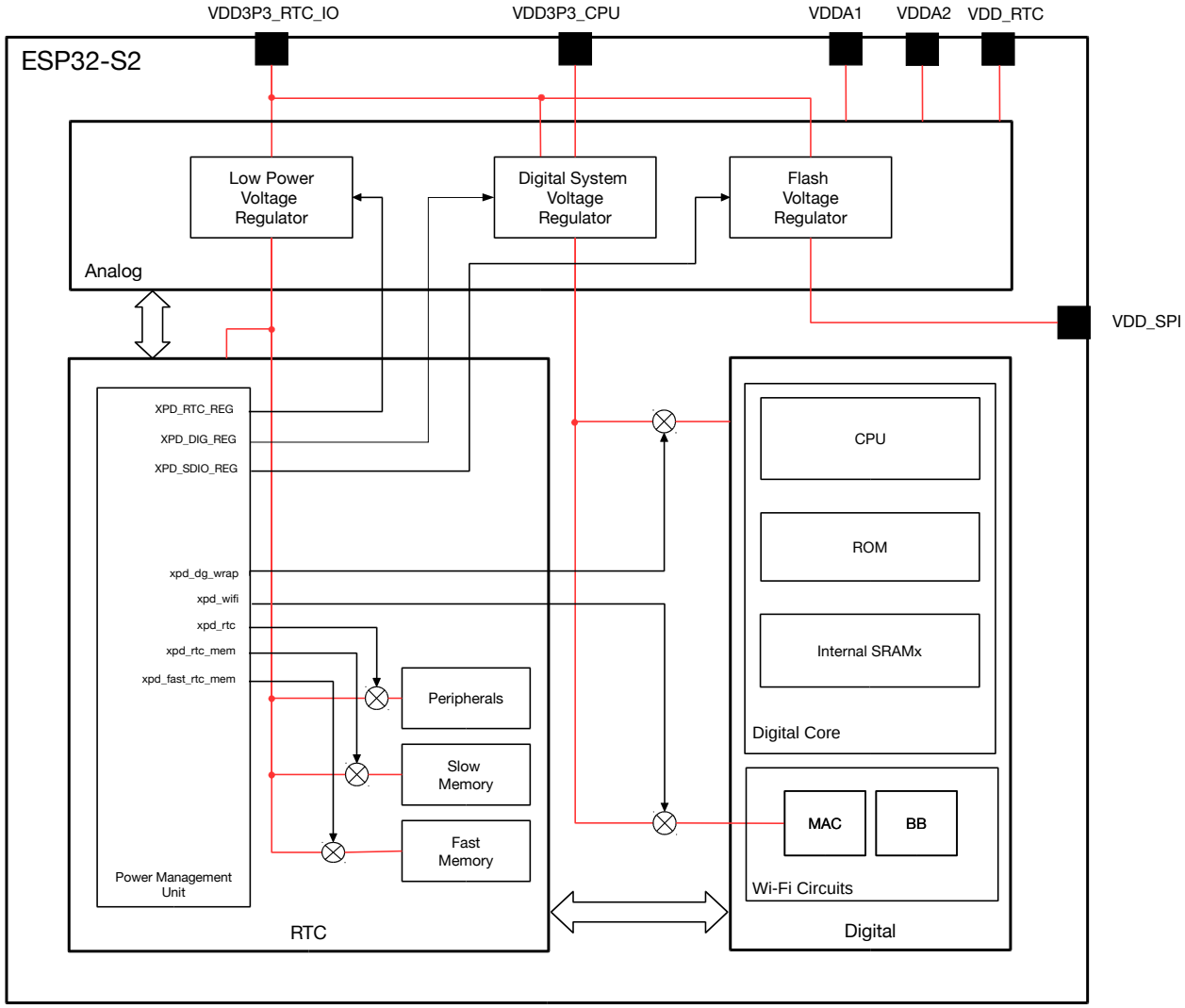


图 28-1. 低功耗管理原理图

28.3.1 功耗管理单元

ESP32-S2 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：产生电源门控、时钟门控和复位信号。
- 功耗控制器：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：向 RTC 主状态机发送睡眠或唤醒请求。
- 时钟控制器：选择并打开或关闭时钟源。

在 ESP32-S2 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 28-2。

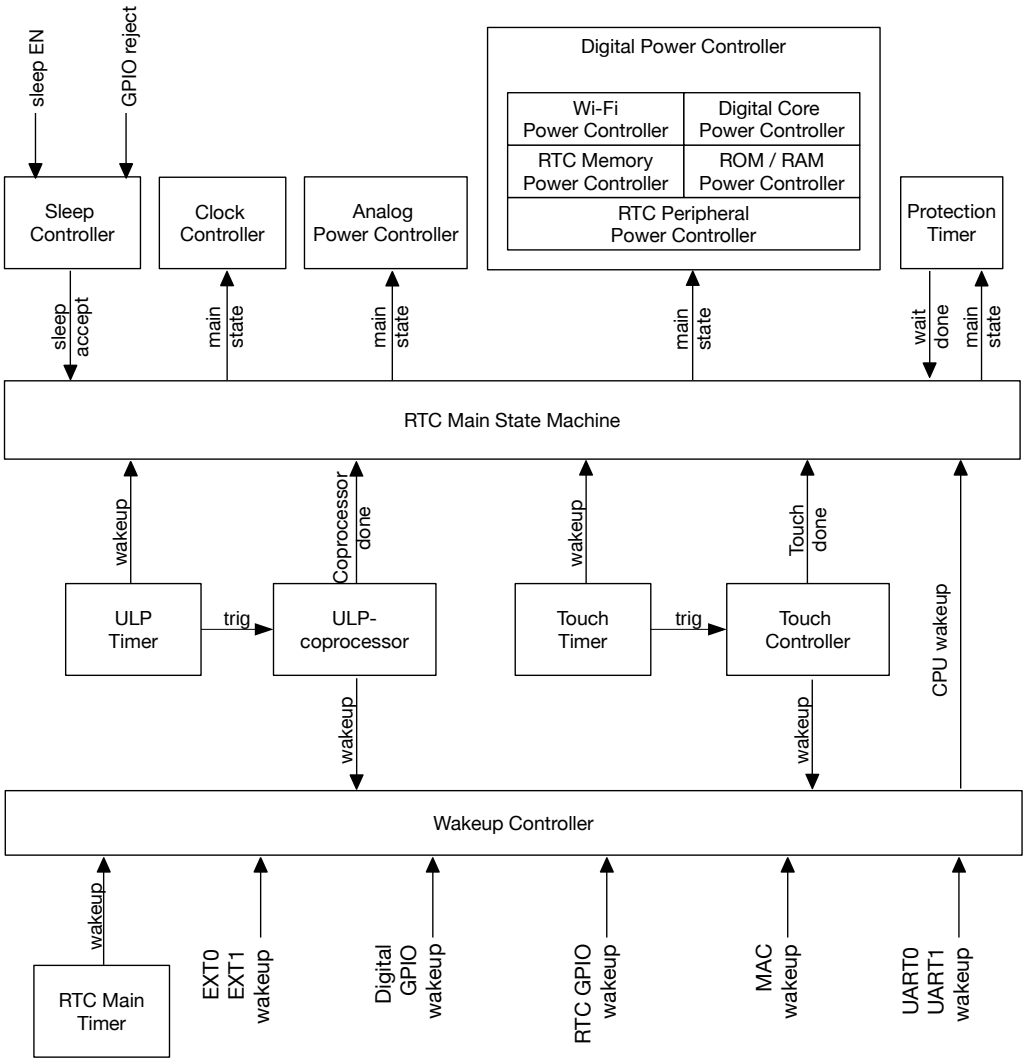


图 28-2. 电源管理单元的主要工作流程

说明：

更多有关不同电源域的描述，请见第 28.4.2 节。

28.3.2 低功耗时钟

通常情况下，当 ESP32-S2 处于低功耗模式下，芯片的 40 MHz 晶振和 PLL 将断电以降低功耗，但低功耗时钟仍保持开启，为不同电源域提供给时钟，比如电源管理单元、RTC 外设、RTC 高速内存、RTC 低速内存，及数字系统中 Wi-Fi 数字电路等，以确保芯片在低功耗模式下的正常工作。

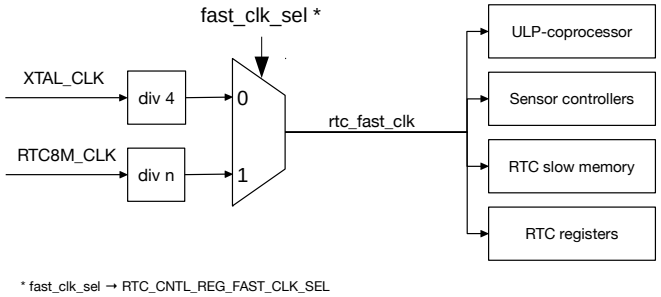


图 28-3. RTC 电源域的低功耗时钟

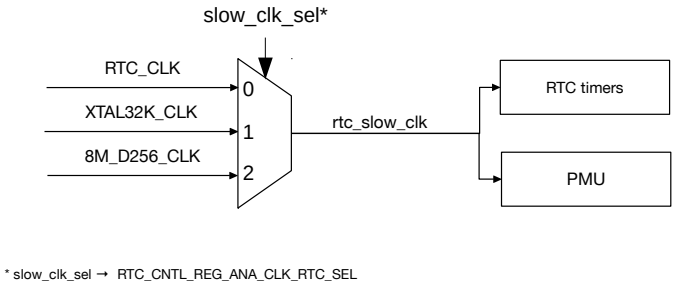


图 28-4. RTC 电源域的低功耗时钟

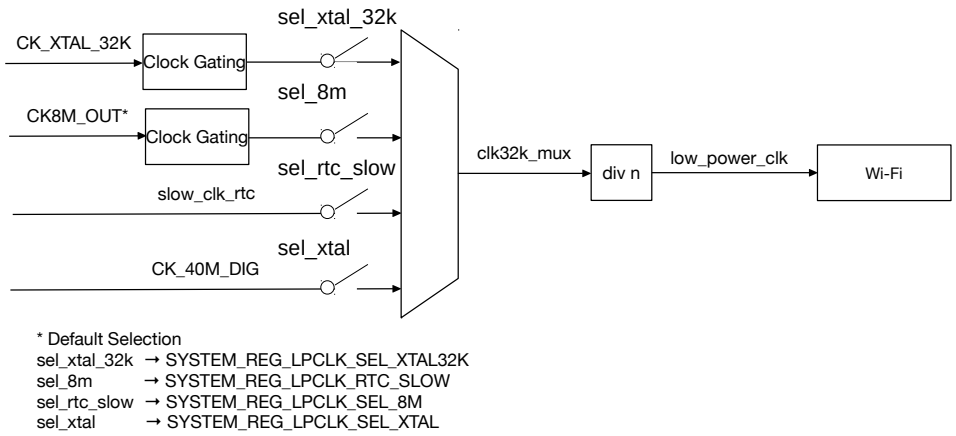


图 28-5. RTC 快速内存和慢速内存的低功耗时钟

表 151: 低功耗时钟

时钟类型	时钟源	时钟选择	电源域
RTC 快速时钟	RTC8M_CLK 的 n 分频 [1]	RTC_CNTL_REG_FAST_CLK_RTC_SEL	RTC 外设
	XTAL_CLK 的 4 分频		RTC 快速内存
			RTC 慢速内存
RTC 慢速时钟	XTAL32K_CLK	RTC_CNTL_REG_ANA_CLK_RTC_SEL	功耗管理单元
	RTC8M_D256_CLK		
	RTC_CLK [2]		
低功耗时钟	XTAL32K_CLK	SYSTEM_LPCLK_SEL_XTAL32K	低功耗模式下的数字系统 (Wi-Fi)
	RTC8M_CLK	SYSTEM_LPCLK_SEL_8M	
	RTC_CLK	SYSTEM_LPCLK_RTC_SLOW	
	XTAL_CLK	SYSTEM_LPCLK_SEL_XTAL	

说明：

1. 默认 RTC 快速时钟源；
2. 默认 RTC 慢速时钟源。

更多有关时钟的描述，请见章节 2 复位和时钟。

28.3.3 定时器

ESP32-S2 的低功耗管理使用三个定时器：

- RTC 定时器
- ULP 定时器
- 触摸传感器定时器

本章节主要介绍 RTC 定时器。有关 ULP 定时器和触摸传感器定时器的内容，请见章节 27 超低功耗协处理器。

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 152。

表 152: RTC 定时器的触发条件

使能条件	描述
RTC_CNTL_REG_TIMER_XTL_OFF	1.RTC 主状态机关闭，或 2. 打开 40 MHz 晶振时均触发。
RTC_CNTL_REG_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_REG_TIMER_SYS_RST	系统复位时触发。
RTC_CNTL_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_TIME_HIGH0_REG
  - RTC_CNTL_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_TIME_HIGH1_REG
  - RTC_CNTL_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 28.4.4 节。

28.3.4 调压器

ESP32-S2 共有三个调压器，负责调节向不同电源域的供电：

- 数字系统调压器：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 28.4.2 节。

28.3.4.1 数字系统调压器

ESP32-S2 的内置数字系统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，具有输出电压可调的特点，具体结构示意图可见下方图 28-6。

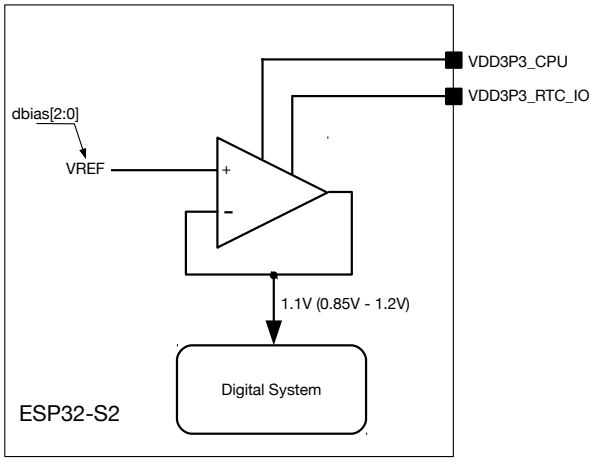


图 28-6. 数字系统调压器

1. 当 XPD_DIG_REG == 1 时，该调压器的输出电压为 1.1 V，此时数字类电源域可以正常工作；当 XPD_DIG_REG == 0 时，该调压器和数字类电源域均关闭。
2. DIG_REG_DBIAS[2:0] 可以调节数字类电源域的供电电压：

$$VDD\_DIG = 0.90 + DBIAS \times 0.05V$$

3. 数字类电源域的输入电压来自管脚 VDD3P3_CPU 和 VDD3P3_RTC_IO。

28.3.4.2 低功耗调压器

ESP32-S2 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，此外，该调压器的输出电压范围可调，可以在 Deep-sleep 和冬眠模式下输出更低的电压，从而降低功耗。具体结构示意图可见下方图 28-1。

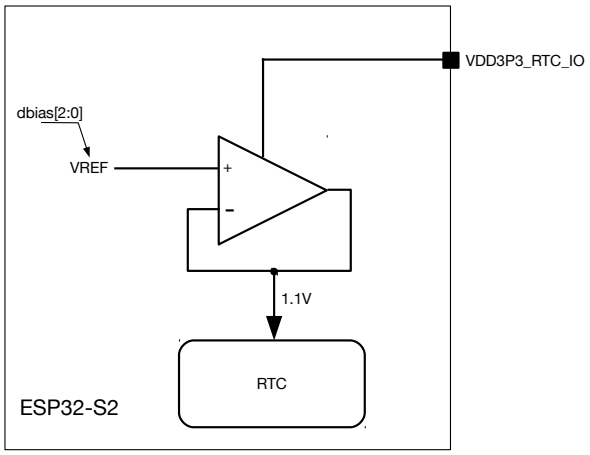


图 28-7. 低功耗调压器

1. 当管脚 CHIP_PU 为高电平时，低功耗调压器无法关闭，仅能在正常和 Deep-sleep 模式之间进行切换。
2. RTC_DBIAS[2:0] 可以调节输出电压：

$$VDD\_RTC = 0.90 + DBIAS \times 0.05V$$

3. RTC 类电源域的输入电压来源为管脚 VDD3P3_RTC_IO。

28.3.4.3 Flash 调压器

ESP32-S2 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 28-8。

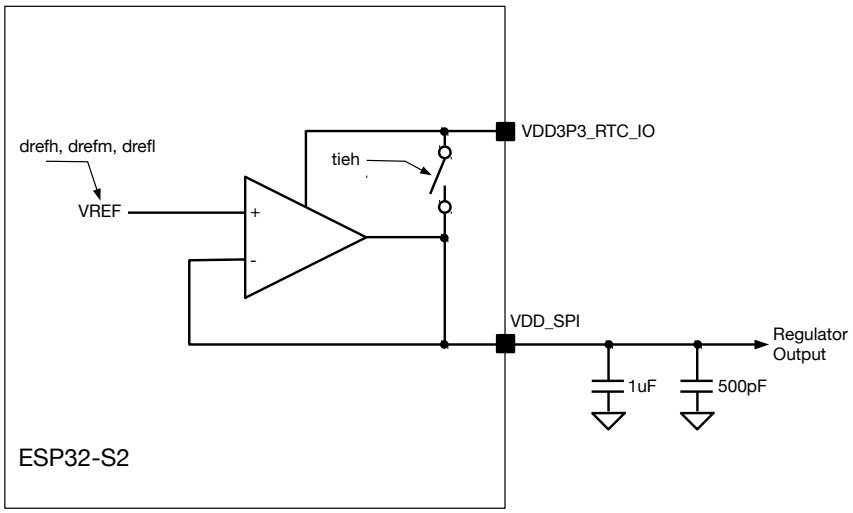


图 28-8. Flash 调压器

1. 当 XPD_SDIO_VREG == 1 时，调压器的输出电压为 3.3 V 或 1.8 V；当 XPD_SDIO_VREG == 0 时，调压器的输出电压为高阻抗，此时，电压由外部电源提供。
2. 当 SDIO_TIEH == 1 时，调压器将管脚 VDD_SDIO 和管脚 VDD3P3_RTC 短路，电压输出为 3.3 V，即管脚 VDD3P3_RTC 的电压。当 SDIO_TIEH == 0 时，调压器的电压输出为参考电压 VREF，通常为 1.8 V。
3. DREFH_SDIO、DREFM_SDIO 和 DREFL_SDIO 可以小幅调节参考电压 VREF，但这种操作可能会影响系统内环的稳定性，因此不推荐进行更改。
4. 当调压器输出为 3.3 V 或 1.8 V，输出电压来自管脚 VDD3P3_RTC_IO。

Flash 调压器的配置可以通过 RTC 寄存器配置，也可以通过操作 eFuse 控制相关参数的配置。

- XPD_SDIO_VREG 的配置：
  - 当芯片处于 Active 状态，且 RTC_CNTL_SDIO_FORCE == 0，VDD_SPI_FORCE(EFUSE) == 1 时，XPD_SDIO_VREG 由 XPD_VDD_SPI_REG(EFUSE) 决定；
  - 当芯片处于 sleep 状态，且 RTC_CNTL_SDIO_REG_PD_EN == 1 时，XPD_SDIO_VREG 为 0；
  - 当 RTC_CNTL_SDIO_FORCE == 1 时，XPD_SDIO_VREG 由 RTC_CNTL_XPD_SDIO_REG 决定。
- SDIO_TIEH 的配置：
  - 当 RTC_CNTL_SDIO_FORCE == 0 且 VDD_SPI_FORCE(EFUSE) == 1 时，SDIO_TIEH = VDD_SDIO_TIEH(EFUSE)；
  - 否则，SDIO_TIEH = RTC_CNTL_SDIO_TIEH。

28.3.4.4 欠压检测器

ESP32-S2 的欠压检测器可以检查管脚 VDD3P3_RTC_IO，VDD3P3_CPU，VDDA1 和 VDDA2 的电压，在电压快速下落至预设阈值以下时发出触发信号，并在触发信号持续一定时间后进行芯片或系统复位，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-S2 欠压检测器的具体结构示意图可见下方图 28-9。

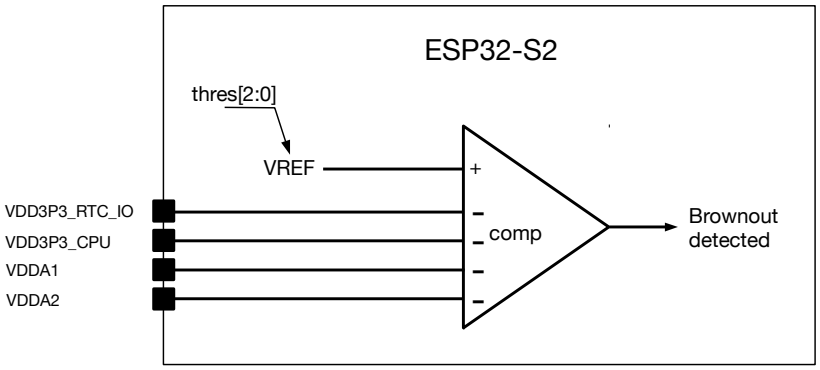


图 28-9. 欠压检测器

1. 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；
2. I2C 寄存器 ULP_CAL_REG5[2:0] 可用于配置欠压检测器的信号触发阈值，具体见表 153；

表 153: 欠压检测器配置

ULP_CAL_REG5[2:0]	VDD（单位：V）
0	2.67
1	3.30
2	3.19
3	2.98
4	2.84
5	2.67
6	2.56
7	2.44

3. RTC_CNTL_REG_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

说明：
更多有关芯片复位和系统复位的介绍，请前往章节 2 复位和时钟。

28.4 功耗模式管理

28.4.1 电源域

ESP32-S2 共有三大类共 10 个电源域：

- RTC 类
  - 功耗管理单元
  - RTC 外设
  - RTC 慢速内存
  - RTC 快速内存
- 数字类
  - 数字内核
  - Wi-Fi 数字电路
- 模拟类
  - 8 MHz 晶振
  - 40 MHz 晶振
  - PLL
  - RF 电路

28.4.2 RTC 状态

ESP32-S2 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 28-10。

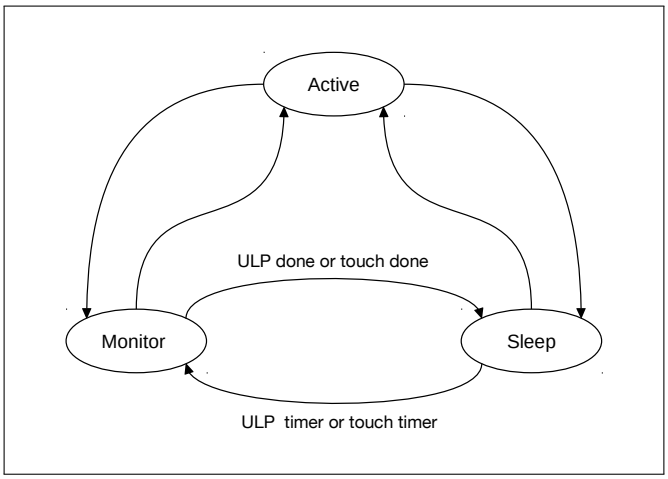


图 28-10. RTC 状态

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 154。

表 154: RTC 状态转换

电源域		RTC 状态			说明
类型	子类型	Active	Monitor	Sleep	
RTC	电源管理单元	ON	ON	ON	1
	RTC 外设	ON	ON	OFF	2
	RTC 慢速内存	ON	OFF	OFF	3
	RTC 快速内存	ON	OFF	OFF	4
数字类	数字内核	ON	OFF	OFF	5
	Wi-Fi 数字电路	ON	OFF	OFF	6
模拟类	8 MHz 晶振	ON	ON	OFF	-
	40 MHz 晶振	ON	OFF	OFF	-
	PLL	ON	OFF	OFF	-
	RF 电路	-	-	-	-

说明：

1. ESP32-S2 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。
2. RTC 外设包括 27 超低功耗协处理器 和片上传感器（比如温度传感器控制器、触摸传感器和 SAR ADC 控制器）。
3. RTC 低速内存支持 8 KB SRAM，可用作保留内存或存储 ULP 指令和数据内存，因此应强制打开。CPU 可通过 PeriBUS2 总线访问慢速内存，起始地址为 0x50000000。
4. 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。
5. 当数字内核关闭时，数字系统下的所有电源域均关闭。值得注意的是，ESP32-S2 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。
6. Wi-Fi 数字电路电源域包括 Wi-Fi MAC 和 BB（基带）。

28.4.3 预设功耗模式

如上文所示，ESP32-S2 定义了五种最常见的电源域设置组合，对应 5 种预设功耗模式，可满足用户的常见场景需求，详见表 155。

表 155: 预设功耗模式

Power Modes	电源域									
	PMU	RTC 外设	RTC 慢速内存	RTC 快速内存	数字内核	Wi-Fi 数字电路	8 MHz 晶振	40 MHz 晶振	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	ON	ON	ON *	OFF	OFF	OFF	OFF
Deep-sleep	ON	ON	ON	ON	OFF	OFF	OFF	OFF	OFF	OFF
冬眠	ON	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF

说明：

* 可配置

默认情况下，ESP32-S2 系统复位后将进入 Active 模式。此后，CPU 在停止工作一段时间后（比如等待外部活动唤醒时），可以进入 Modem-sleep、Light-sleep、Deep-sleep 和冬眠等低功耗模式。从 Active 到冬眠模式，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 155。
2. 具体功耗数据可见 《ESP32-S2 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 28.4.4 节。

28.4.4 唤醒源

ESP32-S2 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_REG_RTC_WAKEUP_ENA 决定，见表 156。

表 156: 唤醒源

WAKEUP_ENA	唤醒源	Light-sleep	Deep-sleep	Hibernation	说明 *
0x1	EXT0	Y	Y	-	1
0x2	EXT1	Y	Y	Y	2
0x4	GPIO	Y	Y	-	3
0x8	RTC 定时器	Y	Y	Y	-
0x20	Wi-Fi	Y	-	-	4
0x40	UART0	Y	-	-	5
0x80	UART1	Y	-	-	5
0x100	TOUCH	Y	Y	-	6
0x800	ULP-FSM	Y	Y	-	7
0x1000	XTAL_32K	Y	Y	Y	8
0x2000	ULP-RISCV Trap	Y	Y	-	9
0x8000	USB	Y	-	-	10

说明：

1. EXT0 仅可将芯片从 Light-sleep/Deep-sleep 模式中唤醒。当 RTC_CNTL_REG_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_REG_EXT_WAKEUP1_SEL[17:0]。当 RTC_CNTL_REG_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，所有选中管脚为高电压才能触发信号唤醒芯片。

3. 在 Deep-sleep 模式下，仅有 RTC GPIO 可以作为唤醒源。
4. 为了通过 Wi-Fi 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 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. 当 USB 主机发送指令让 USB 进入 RESUMING 状态时，即触发唤醒。

28.5 RTC Boot

在 Deep-sleep 和冬眠模式下，芯片的 ROM 和 RAM 均将断电，因此在唤醒时 ROM 解包与 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 和冬眠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 ROM 解包或 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 ROM 解包和 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_STORE6_REG[31:0] 中。
3. 设置寄存器 RTC_CNTL_STORE7_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_STORE6_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-S2 的 RTC 启动流程见下方图 28-11：

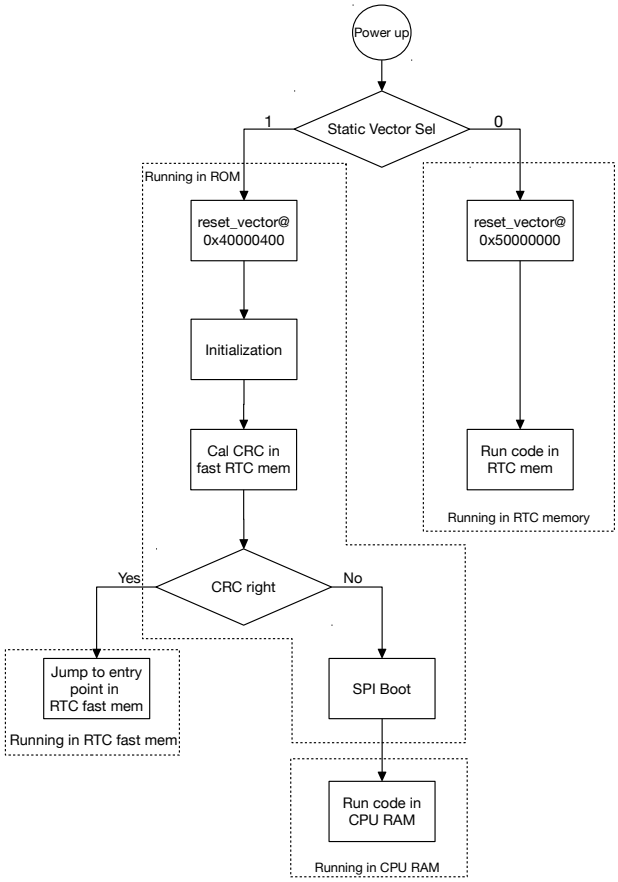


图 28-11. ESP32-S2 启动流程图

在低功耗模式下，ESP32-S2 的 40 MHz 晶振和 PLL 通常将断电以降低功耗，但低功耗时钟仍将开启，以确保芯片在低功耗模式下的正常工作。

28.6 基地址

用户可以通过两个不同的寄存器基地址访问 ESP32-S2 的低功耗管理模块，如表 157 所示。更多有关总线的信息，请见章节 1 *系统和存储器*。

表 157: 低功耗管理基地址

外设访问方式	基地址
PeriBUS1	0x3f408000
PeriBUS2	0x60008000

28.7 寄存器列表

请注意，本章节的地址是相对于低功耗管理模块基地址的地址偏移量（相对地址）。请参阅章节 28.6 获取有关低功耗管理模块基地址的信息。

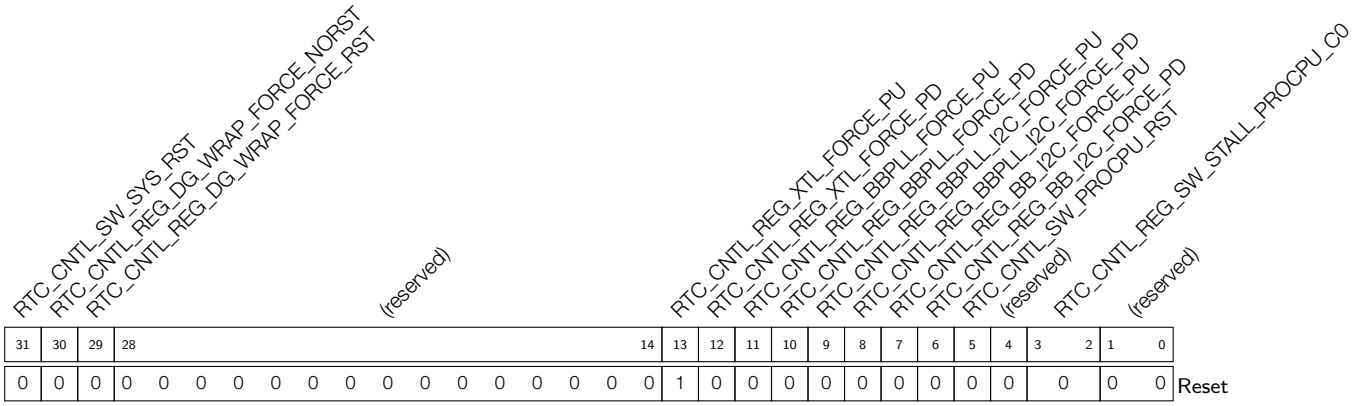
名称	描述	地址	访问权限
RTC 选项寄存器			
RTC_CNTL_OPTIONS0_REG	设置晶振和 PLL 时钟的电源选项，并启动软件复位	0x0000	可变
RTC_CNTL_OPTION1_REG	RTC 选项寄存器	0x0128	读/写
RTC 定时器寄存器			
RTC_CNTL_SLP_TIMER0_REG	RTC 定时器阈值寄存器 0	0x0004	读/写
RTC_CNTL_SLP_TIMER1_REG	RTC 定时器阈值寄存器 1	0x0008	可变
RTC_CNTL_TIME_UPDATE_REG	RTC 定时器更新控制寄存器	0x000C	可变
RTC_CNTL_TIME_LOW0_REG	存储 RTC 定时器 0 的低 32 位	0x0010	只读
RTC_CNTL_TIME_HIGH0_REG	存储 RTC 定时器 0 的高 16 位	0x0014	只读
RTC_CNTL_STATE0_REG	配置 sleep / reject / wakeup 状态	0x0018	可变
RTC_CNTL_TIMER1_REG	配置 CPU stall 选项	0x001C	读/写
RTC_CNTL_TIMER2_REG	配置 RTC 慢速时钟和触摸控制器	0x0020	读/写
RTC_CNTL_TIMER5_REG	配置最小睡眠周期	0x002C	读/写
RTC_CNTL_TIME_LOW1_REG	存储 RTC 定时器 1 的低 32 位	0x00E8	只读
RTC_CNTL_TIME_HIGH1_REG	存储 RTC 定时器 1 的高 16 位	0x00EC	只读
内部电源控制寄存器			
RTC_CNTL_ANA_CONF_REG	配置 I2C 和 PLLA 的电源选项	0x0034	读/写
RTC_CNTL_REG_REG	低功耗和数字系统调压器配置寄存器	0x0084	读/写
RTC_CNTL_PWC_REG	RTC 电源配置寄存器	0x0088	读/写
RTC_CNTL_DIG_PWC_REG	数字系统电源配置寄存器	0x008C	读/写
RTC_CNTL_DIG_ISO_REG	数字系统 ISO 配置寄存器	0x0090	可变
RTC_CNTL_LOW_POWER_ST_REG	RTC 主状态机状态寄存器	0x00CC	只读
复位控制寄存器			
RTC_CNTL_RESET_STATE_REG	指示 CPU 复位的来源	0x0038	可变
睡眠与唤醒控制寄存器			
RTC_CNTL_WAKEUP_STATE_REG	唤醒位图使能寄存器	0x003C	读/写
RTC_CNTL_EXT_WAKEUP_CONF_REG	GPIO 唤醒配置寄存器	0x0064	读/写
RTC_CNTL_SLP_REJECT_CONF_REG	睡眠/拒绝选项配置寄存器 / reject options	0x0068	读/写
RTC_CNTL_EXT_WAKEUP1_REG	EXT1 唤醒配置寄存器	0x00DC	可变
RTC_CNTL_EXT_WAKEUP1_STATUS_REG	EXT1 唤醒源寄存器	0x00E0	只读
RTC_CNTL_SLP_REJECT_CAUSE_REG	存储拒绝入睡的原因	0x0124	只读
RTC_CNTL_SLP_WAKEUP_CAUSE_REG	存储从睡眠中唤醒的原因	0x012C	只读
中断寄存器			
RTC_CNTL_INT_ENA_RTC_REG	RTC 中断使能寄存器	0x0040	读/写
RTC_CNTL_INT_RAW_RTC_REG	RTC 原始中断寄存器	0x0044	只读
RTC_CNTL_INT_ST_RTC_REG	RTC 中断状态寄存器	0x0048	只读
RTC_CNTL_INT_CLR_RTC_REG	RTC 中断清除寄存器	0x004C	只写
保留寄存器			
RTC_CNTL_STORE0_REG	保留寄存器 0	0x0050	读/写
RTC_CNTL_STORE1_REG	保留寄存器 1	0x0054	读/写
RTC_CNTL_STORE2_REG	保留寄存器 2	0x0058	读/写

名称	描述	地址	访问权限
RTC_CNTL_STORE3_REG	保留寄存器 3	0x005C	读/写
RTC_CNTL_STORE4_REG	保留寄存器 4	0x00BC	读/写
RTC_CNTL_STORE5_REG	保留寄存器 5	0x00C0	读/写
RTC_CNTL_STORE6_REG	保留寄存器 6	0x00C4	读/写
RTC_CNTL_STORE7_REG	保留寄存器 7	0x00C8	读/写
时钟控制寄存器			
RTC_CNTL_EXT_XTL_CONF_REG	32 kHz 晶振配置寄存器	0x0060	可变
RTC_CNTL_CLK_CONF_REG	RTC 时钟配置寄存器	0x0074	读/写
RTC_CNTL_SLOW_CLK_CONF_REG	RTC 慢速时钟配置寄存器	0x0078	读/写
RTC_CNTL_XTAL32K_CLK_FACTOR_REG	配置 32 kHz 晶振备用时钟的分频数	0x00F0	读/写
RTC_CNTL_XTAL32K_CONF_REG	32 kHz 晶振配置寄存器	0x00F4	读/写
RTC 看门狗控制寄存器			
RTC_CNTL_WDTCONFIG0_REG	RTC 看门狗配置寄存器	0x0094	读/写
RTC_CNTL_WDTCONFIG1_REG	配置 1 级 RTC 看门狗的保持时间	0x0098	读/写
RTC_CNTL_WDTCONFIG2_REG	配置 2 级 RTC 看门狗的保持时间	0x009C	读/写
RTC_CNTL_WDTCONFIG3_REG	配置 3 级 RTC 看门狗的保持时间	0x00A0	读/写
RTC_CNTL_WDTCONFIG4_REG	配置 4 级 RTC 看门狗的保持时间	0x00A4	读/写
RTC_CNTL_WDTFEED_REG	RTC 看门狗软件喂狗配置寄存器	0x00A8	只写
RTC_CNTL_WDTWPROTECT_REG	RTC 看门狗写保护配置寄存器	0x00AC	读/写
RTC_CNTL_SWD_CONF_REG	超级看门狗配置寄存器	0x00B0	可变
RTC_CNTL_SWD_WPROTECT_REG	超级看门狗写保护配置寄存器	0x00B4	读/写
其他寄存器			
RTC_CNTL_SW_CPU_STALL_REG	CPU stall 配置寄存器	0x00B8	读/写
RTC_CNTL_PAD_HOLD_REG	配置 RTC GPIO 的保持时间	0x00D4	读/写
RTC_CNTL_DIG_PAD_HOLD_REG	配置数字 GPIO 的保持时间	0x00D8	读/写
RTC_CNTL_BROWN_OUT_REG	欠压检测配置寄存器	0x00E4	可变

28.8 寄存器

请注意，本章节的地址是相对于低功耗管理模块基地址的地址偏移量（相对地址）。请参阅章节 28.6 获取有关低功耗管理模块基地址的信息。

Register 28.1: RTC_CNTL_OPTIONS0_REG (0x0000)



RTC_CNTL_REG_SW_STALL_PROCPU_C0 当 RTC_CNTL_REG_SW_STALL_PROCPU_C1 配置为 0x21 时，设置该位为 0x2 将软件使 CPU 进入 stall 状态。

RTC_CNTL_SW_PROCPU_RST 置 1 软件复位 CPU。(只写)

RTC_CNTL_REG_BB_I2C_FORCE_PD 置 1 强制打开 BB_I2C。(读/写)

RTC_CNTL_REG_BB_I2C_FORCE_PU 置 1 强制打开 BB_I2C。(读/写)

RTC_CNTL_REG_BBPLL_I2C_FORCE_PD 置 1 强制关闭 BB_PLL_I2C。(读/写)

RTC_CNTL_REG_BBPLL_I2C_FORCE_PU 置 1 强制打开 BB_PLL_I2C。(读/写)

RTC_CNTL_REG_BBPLL_FORCE_PD 置 1 强制关闭 BB_PLL。(读/写)

RTC_CNTL_REG_BBPLL_FORCE_PU 置 1 强制打开 BB_PLL。(读/写)

RTC_CNTL_REG_XTL_FORCE_PD 置 1 强制关闭晶振。(读/写)

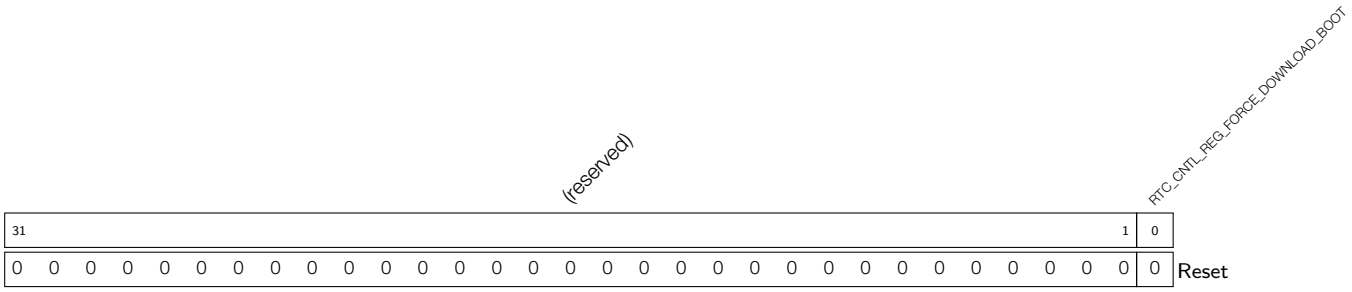
RTC_CNTL_REG_XTL_FORCE_PU 置 1 强制打开晶振。(读/写)

RTC_CNTL_REG_DG_WRAP_FORCE_RST 置 1 强制 deep sleep 中的数字系统复位。(读/写)

RTC_CNTL_REG_DG_WRAP_FORCE_NORST 置 1 强制 deep sleep 中的数字系统不复位。(读/写)

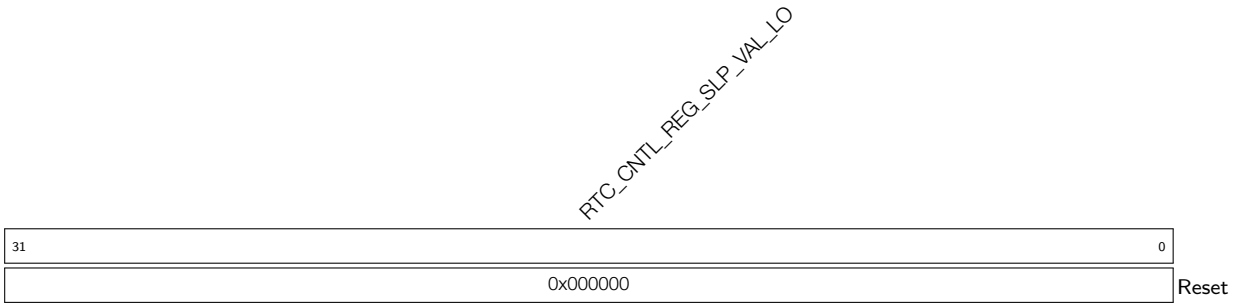
RTC_CNTL_SW_SYS_RST 置 1 通过软件复位系统。(只写)

Register 28.2: RTC_CNTL_OPTION1_REG (0x0128)



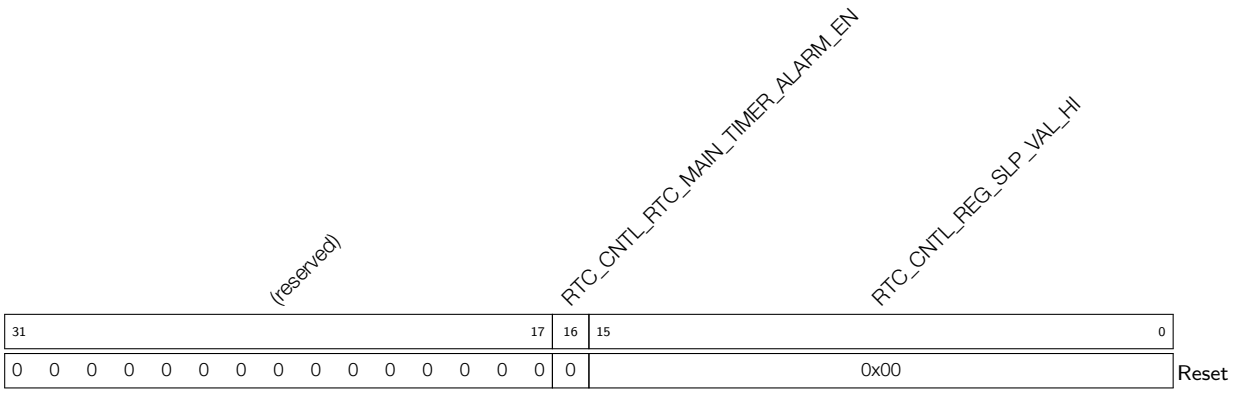
RTC_CNTL_REG_FORCE_DOWNLOAD_BOOT 置 1 强制芯片从下载模式启动。(读/写)

Register 28.3: RTC_CNTL_SLP_TIMER0_REG (0x0004)



RTC_CNTL_REG_SLP_VAL_LO 配置 RTC 计时器触发阈值的低 32 位。(读/写)

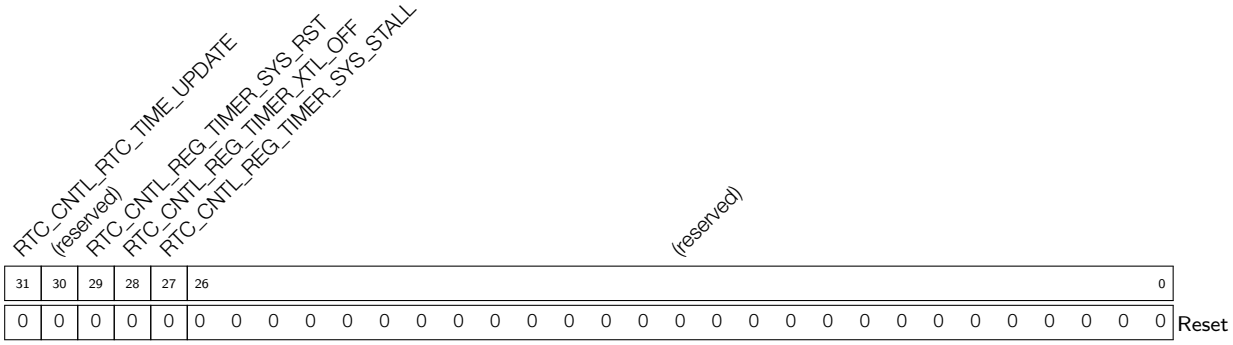
Register 28.4: RTC_CNTL_SLP_TIMER1_REG (0x0008)



RTC_CNTL_REG_SLP_VAL_HI 配置 RTC 计时器触发阈值的高 16 位。(读/写)

RTC_CNTL_RTC_MAIN_TIMER_ALARM_EN 置 1 使能定时器警报。(只写)

Register 28.5: RTC_CNTL_TIME_UPDATE_REG (0x000C)



RTC_CNTL_REG_TIMER_SYS_STALL 选择 RTC 计时器的触发条件。(读/写)

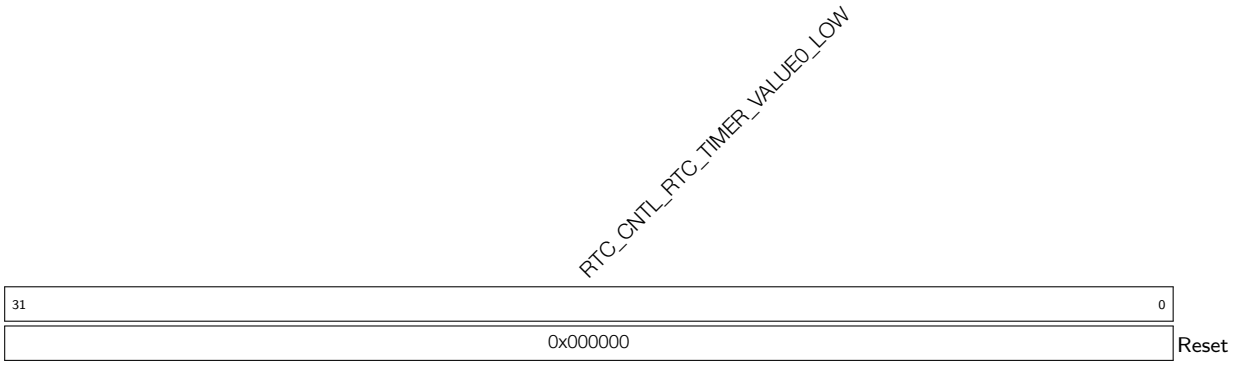
RTC_CNTL_REG_TIMER_XTL_OFF 选择 RTC 计时器的触发条件。(读/写)

RTC_CNTL_REG_TIMER_SYS_RST 选择 RTC 计时器的触发条件。(读/写)

RTC_CNTL_RTC_TIME_UPDATE 选择 RTC 计时器的触发条件。(只写)

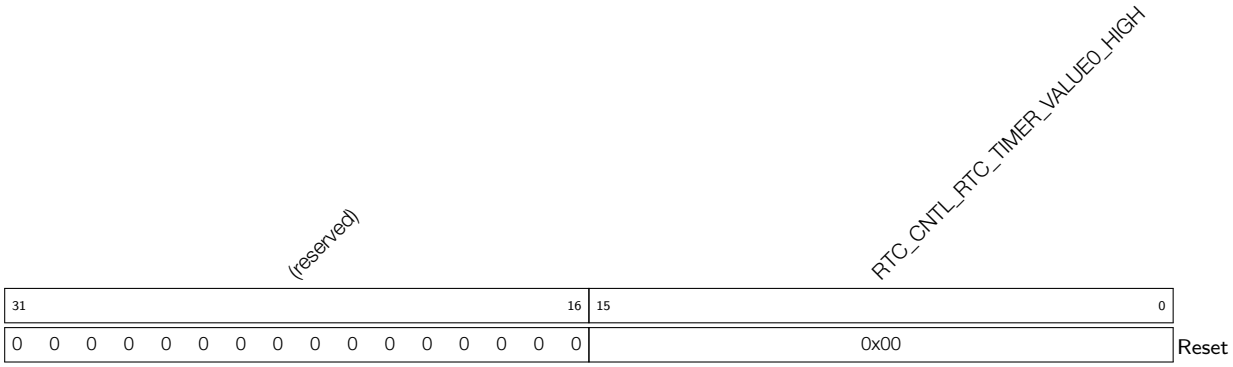
说明:
更多详情，请见表 152。

Register 28.6: RTC_CNTL_TIME_LOW0_REG (0x0010)



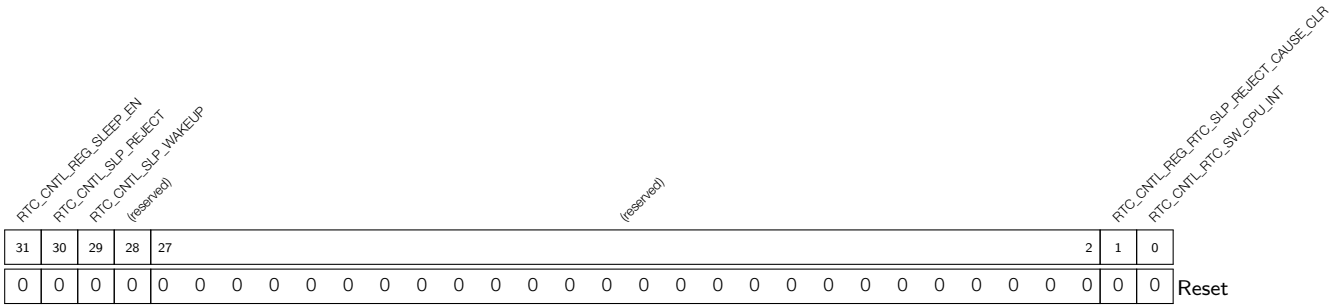
RTC_CNTL_RTC_TIMER_VALUE0_LOW 存储 RTC 计时器 0 的低 32 位。(只读)

Register 28.7: RTC_CNTL_TIME_HIGH0_REG (0x0014)



RTC_CNTL_RTC_TIMER_VALUE0_HIGH 存储 RTC 计时器 0 的高 16 位。(只读)

Register 28.8: RTC_CNTL_STATE0_REG (0x0018)



RTC_CNTL_RTC_SW_CPU_INT 发送软件 RTC 中断给 CPU。(只写)

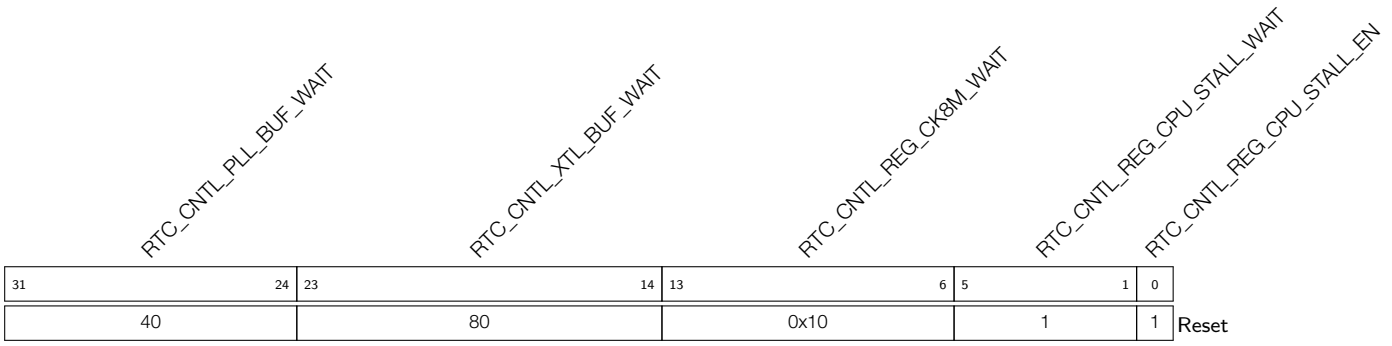
RTC_CNTL_REG_RTC_SLP_REJECT_CAUSE_CLR 清除 RTC 拒绝入睡原因。(只写)

RTC_CNTL_SLP_WAKEUP 睡眠唤醒位。(读/写)

RTC_CNTL_SLP_REJECT 拒绝入睡位。(读/写)

RTC_CNTL_REG_SLEEP_EN 使芯片进入睡眠。(读/写)

Register 28.9: RTC_CNTL_TIMER1_REG (0x001C)



RTC_CNTL_REG_CPU_STALL_EN 使能 CPU stall。(读/写)

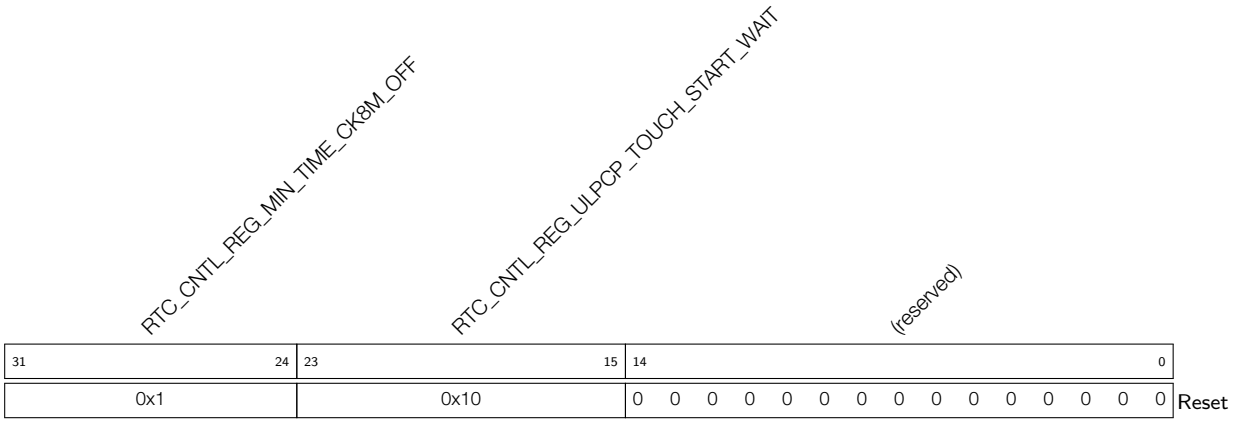
RTC_CNTL_REG_CPU_STALL_WAIT 设置 CPU stall 的等待周期（使用 RTC 快速时钟)。(读/写)

RTC_CNTL_REG_CK8M_WAIT 设置 8 MHz 时钟的等待周期（使用 RTC 慢速时钟)。(读/写)

RTC_CNTL_XTL_BUF_WAIT 设置 XTAL 的等待周期（使用 RTC 慢速时钟)。(读/写)

RTC_CNTL_PLL_BUF_WAIT 设置 PLL 的等待周期（使用 RTC 慢速时钟)。(读/写)

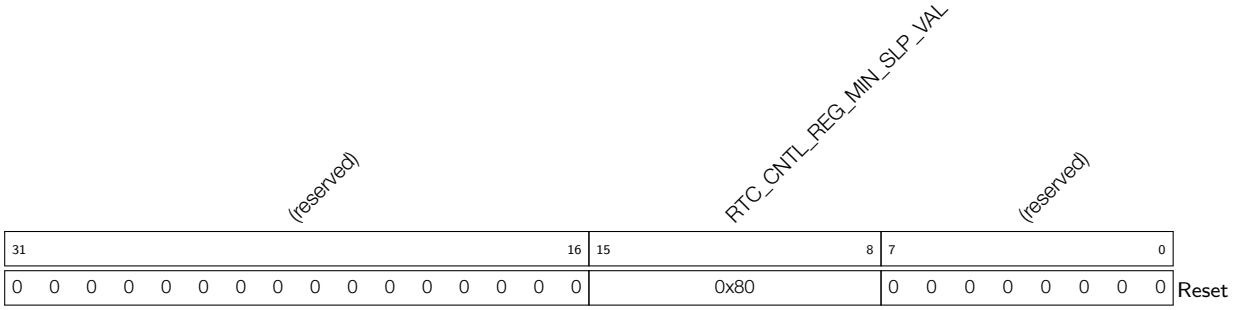
Register 28.10: RTC_CNTL_TIMER2_REG (0x0020)



RTC_CNTL_REG_ULPCP_TOUCH_START_WAIT 设置超低功耗协处理器或触摸控制器开始工作之前的等待周期（使用 RTC 慢速时钟)。(读/写)

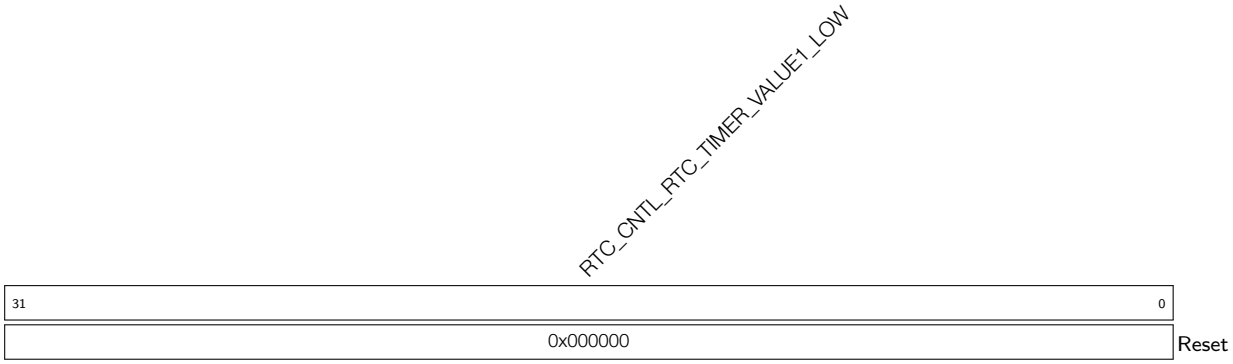
RTC_CNTL_REG_MIN_TIME_CK8M_OFF 8 MHz 时钟断电时的最小等待周期（使用 RTC 慢速时钟)。(读/写)

Register 28.11: RTC_CNTL_TIMER5_REG (0x002C)



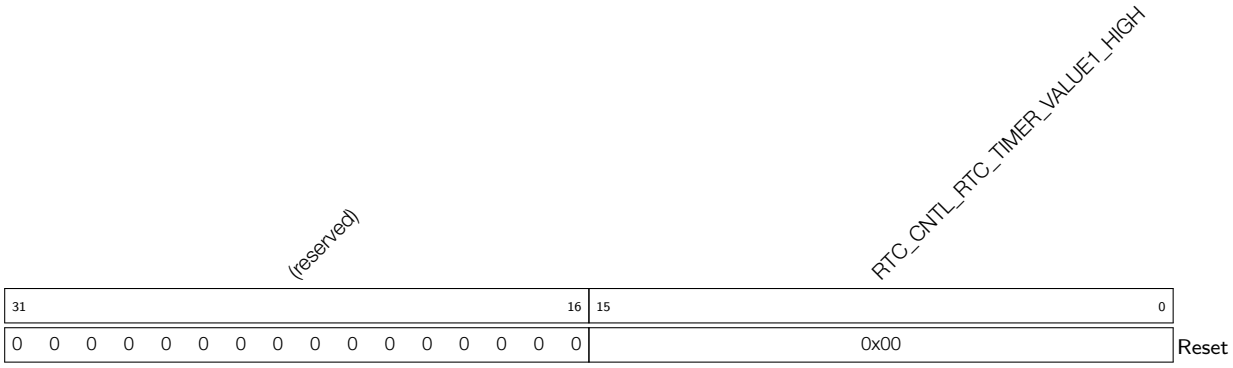
RTC_CNTL_REG_MIN_SLP_VAL 设置最小睡眠周期（使用 RTC 慢速时钟）。（读/写）

Register 28.12: RTC_CNTL_TIME_LOW1_REG (0x00E8)



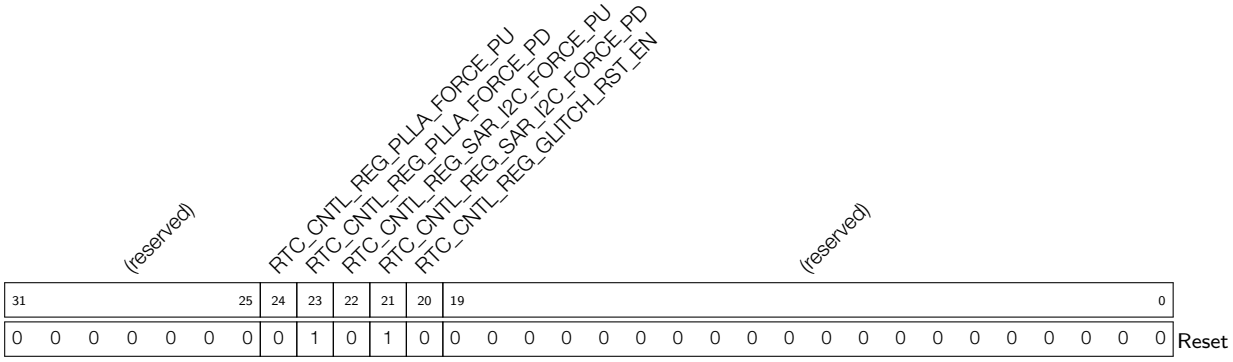
RTC_CNTL_RTC_TIMER_VALUE1_LOW 存储 RTC 计时器 1 的低 32 位。（只读）

Register 28.13: RTC_CNTL_TIME_HIGH1_REG (0x00EC)



RTC_CNTL_RTC_TIMER_VALUE1_HIGH 存储 RTC 计时器 1 的高 16 位。（只读）

Register 28.14: RTC_CNTL_ANA_CONF_REG (0x0034)



RTC_CNTL_REG_GLITCH_RST_EN 置 1 使能在系统监测到脉冲毛刺时进行复位。(读/写)

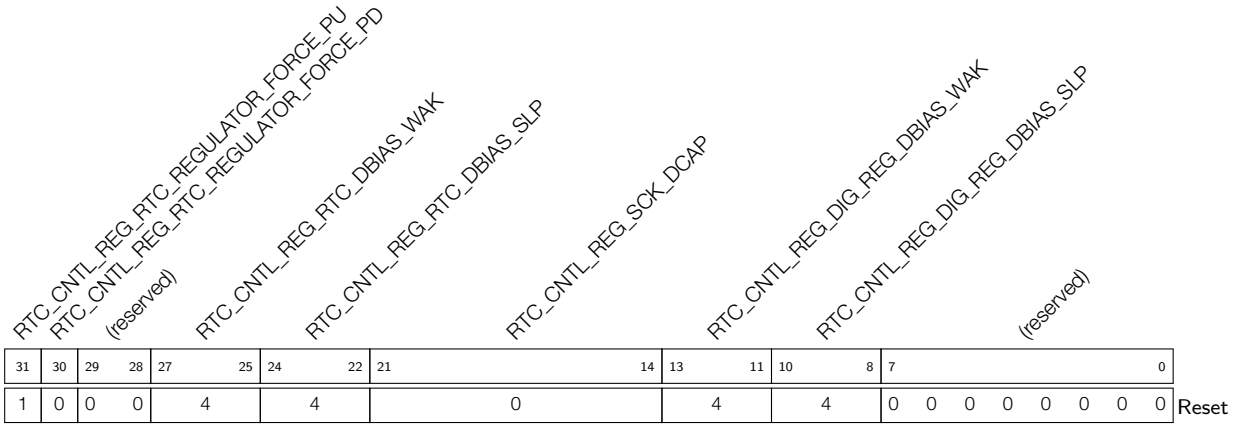
RTC_CNTL_REG_SAR_I2C_FORCE_PD 置 1 强制关闭 SAR_I2C。(读/写)

RTC_CNTL_REG_SAR_I2C_FORCE_PU 置 1 强制打开 SAR_I2C。(读/写)

RTC_CNTL_REG_PLLA_FORCE_PD 置 1 强制关闭 PLLA。(读/写)

RTC_CNTL_REG_PLLA_FORCE_PU 置 1 强制打开 PLLA。(读/写)

Register 28.15: RTC_CNTL_REG_REG (0x0084)



RTC_CNTL_REG_DIG_REG_DBIAS_SLP 配置 CPU 处于 Sleep 状态下数字系统调压器的调压系数。(读/写)

RTC_CNTL_REG_DIG_REG_DBIAS_WAK 配置 CPU 处于 Active 状态下数字系统调压器的调压系数。(读/写)

RTC_CNTL_REG_SCK_DCAP 配置 RTC 时钟频率。(读/写)

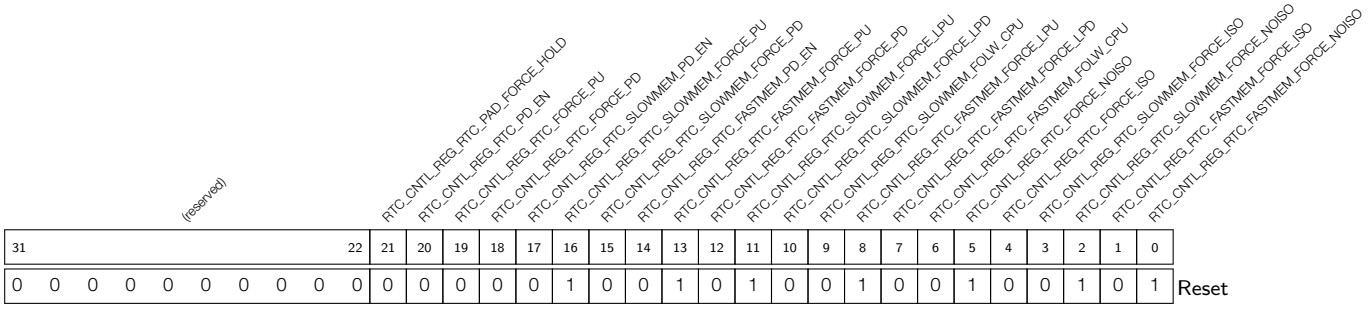
RTC_CNTL_REG_RTC_DBIAS_SLP 配置 CPU 处于 Sleep 状态下低功耗调压器的调压系数。(读/写)

RTC_CNTL_REG_RTC_DBIAS_WAK 配置 CPU 处于 Active 状态下低功耗调压器的调压系数。(读/写)

RTC_CNTL_REG_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)。(读/写)

RTC_CNTL_REG_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)。(读/写)

Register 28.16: RTC_CNTL_PWC_REG (0x0088)



RTC_CNTL_REG_RTC_FASTMEM_FORCE_NOISO 置 1 强制禁止隔离 RTC 快速内存。(读/写)

RTC_CNTL_REG_RTC_FASTMEM_FORCE_ISO 置 1 强制隔离 RTC 快速内存。(读/写)

RTC_CNTL_REG_RTC_SLOWMEM_FORCE_NOISO 置 1 强制禁止隔离 RTC 慢速内存。(读/写)

RTC_CNTL_REG_RTC_SLOWMEM_FORCE_ISO 置 1 强制隔离 RTC 慢速内存。(读/写)

RTC_CNTL_REG_RTC_FORCE_ISO 置 1 强制隔离 RTC 外设。(读/写)

RTC_CNTL_REG_RTC_FORCE_NOISO 置 1 强制禁止隔离 RTC 外设。(读/写)

RTC_CNTL_REG_RTC_FASTMEM_FOLW_CPU 置 1 在 CPU 掉电时强制关闭 RTC 快速内存，置 0 在 RTC 主状态机掉电时强制关闭 RTC 快速内存。(读/写)

RTC_CNTL_REG_RTC_FASTMEM_FORCE_LPD 置 1 强制不保留 RTC 快速内存。(读/写)

RTC_CNTL_REG_RTC_FASTMEM_FORCE_LPU 置 1 强制保留 RTC 快速内存。(读/写)

RTC_CNTL_REG_RTC_SLOWMEM_FOLW_CPU 置 1 在 CPU 掉电时强制关闭 RTC 慢速内存，置 0 在 RTC 主状态机掉电时强制关闭 RTC 慢速内存。(读/写)

RTC_CNTL_REG_RTC_SLOWMEM_FORCE_LPD 置 1 强制不保留 RTC 慢速内存。(读/写)

RTC_CNTL_REG_RTC_SLOWMEM_FORCE_LPU 置 1 强制保留 RTC 慢速内存。(读/写)

RTC_CNTL_REG_RTC_FASTMEM_FORCE_PD 置 1 强制关闭 RTC 快速内存。(读/写)

RTC_CNTL_REG_RTC_FASTMEM_FORCE_PU 置 1 强制打开 RTC 快速内存。(读/写)

RTC_CNTL_REG_RTC_FASTMEM_PD_EN 置 1 使能在 Sleep 状态中关闭 RTC 快速内存。(读/写)

接下页...

Register 28.16: RTC_CNTL_PWC_REG (0x0088)

接上页...

RTC_CNTL_REG_RTC_SLOWMEM_FORCE_PD 置 1 强制关闭 RTC 慢速内存。(读/写)

RTC_CNTL_REG_RTC_SLOWMEM_FORCE_PU 置 1 强制打开 RTC 慢速内存。(读/写)

RTC_CNTL_REG_RTC_SLOWMEM_PD_EN 置 1 使能在 Sleep 状态中关闭 RTC 慢速内存。(读/写)

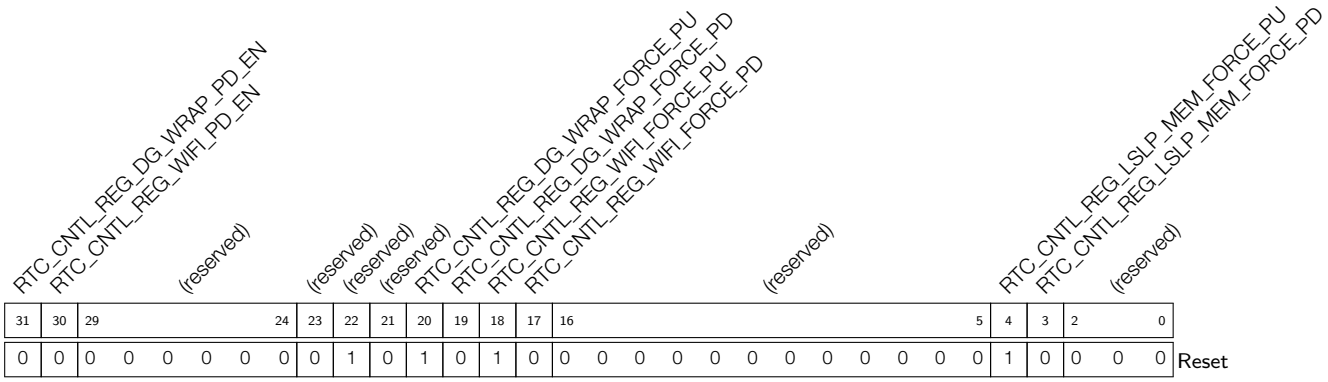
RTC_CNTL_REG_RTC_FORCE_PD 置 1 强制关闭 RTC 外设。(读/写)

RTC_CNTL_REG_RTC_FORCE_PU 置 1 强制打开 RTC 外设。(读/写)

RTC_CNTL_REG_RTC_PD_EN 置 1 使能在 Sleep 状态中关闭 RTC 外设。(读/写)

RTC_CNTL_REG_RTC_PAD_FORCE_HOLD 置 1 强制保留 RTC GPIO。(读/写)

Register 28.17: RTC_CNTL_DIG_PWC_REG (0x008C)



RTC_CNTL_REG_LSLP_MEM_FORCE_PD 置 1 在 Sleep 状态下强制关闭数字系统中的内存。(读/写)

RTC_CNTL_REG_LSLP_MEM_FORCE_PU 置 1 在 Sleep 状态下强制打开数字系统中的内存。(读/写)

RTC_CNTL_REG_WIFI_FORCE_PD 置 1 强制关闭 Wi-Fi 数字电路。(读/写)

RTC_CNTL_REG_WIFI_FORCE_PU 置 1 强制打开 Wi-Fi 电路。(读/写)

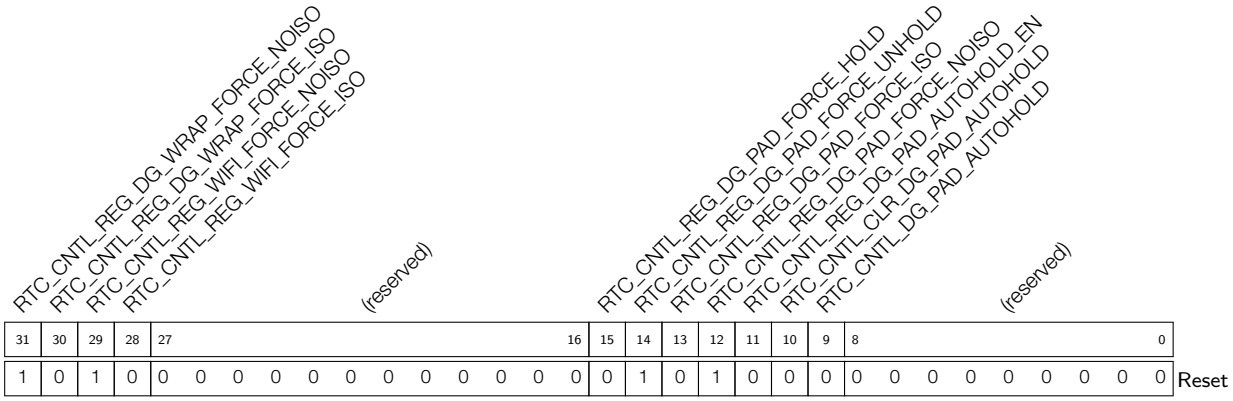
RTC_CNTL_REG_DG_WRAP_FORCE_PD 置 1 强制关闭数字系统。(读/写)

RTC_CNTL_REG_DG_WRAP_FORCE_PU 置 1 强制打开数字系统。(读/写)

RTC_CNTL_REG_WIFI_PD_EN 置 1 在 Sleep 状态下强制关闭数字系统中的 Wi-Fi 电路。(读/写)

RTC_CNTL_REG_DG_WRAP_PD_EN 置 1 使能在 Sleep 状态下关闭数字系统。(读/写)

Register 28.18: RTC_CNTL_DIG_ISO_REG (0x0090)



RTC_CNTL_DG_PAD_AUTOHOLD 指示数字 GPIO 的 auto-hold 状态。(只读)

RTC_CNTL_CLR_DG_PAD_AUTOHOLD 置 1 取消数字 GPIO 的 auto-hold 状态。(只写)

RTC_CNTL_REG_DG_PAD_AUTOHOLD_EN 置 1 允许数字 GPIO 进入 auto-hold 状态。(读/写)

RTC_CNTL_REG_DG_PAD_FORCE_NOISO 置 1 强制不隔离数字 GPIO。(读/写)

RTC_CNTL_REG_DG_PAD_FORCE_ISO 置 1 强制隔离数字 GPIO。(读/写)

RTC_CNTL_REG_DG_PAD_FORCE_UNHOLD 置 1 强制数字 GPIO 进入 unhold 状态。(读/写)

RTC_CNTL_REG_DG_PAD_FORCE_HOLD 置 1 强制数字 GPIO 进入 hold 状态。(读/写)

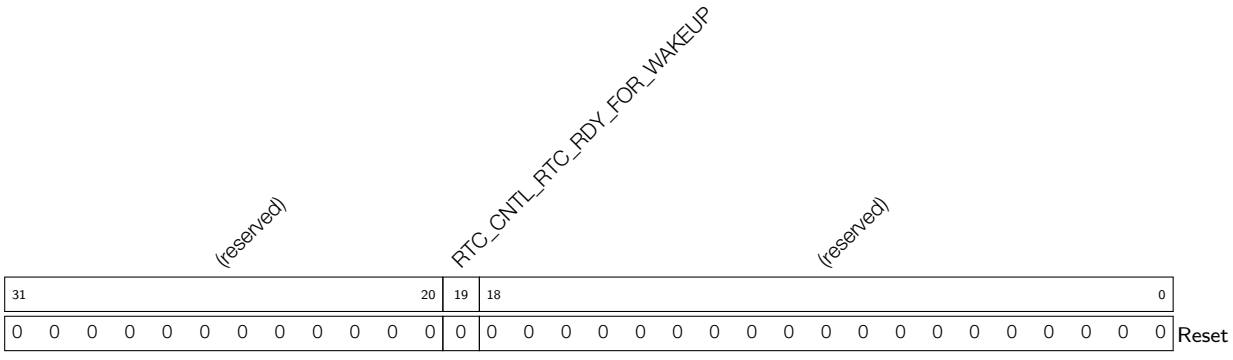
RTC_CNTL_REG_WIFI_FORCE_ISO 置 1 强制隔离 Wi-Fi 电路。(读/写)

RTC_CNTL_REG_WIFI_FORCE_NOISO 置 1 强制不隔离 Wi-Fi 电路。(读/写)

RTC_CNTL_REG_DG_WRAP_FORCE_ISO 置 1 强制隔离数字系统。(读/写)

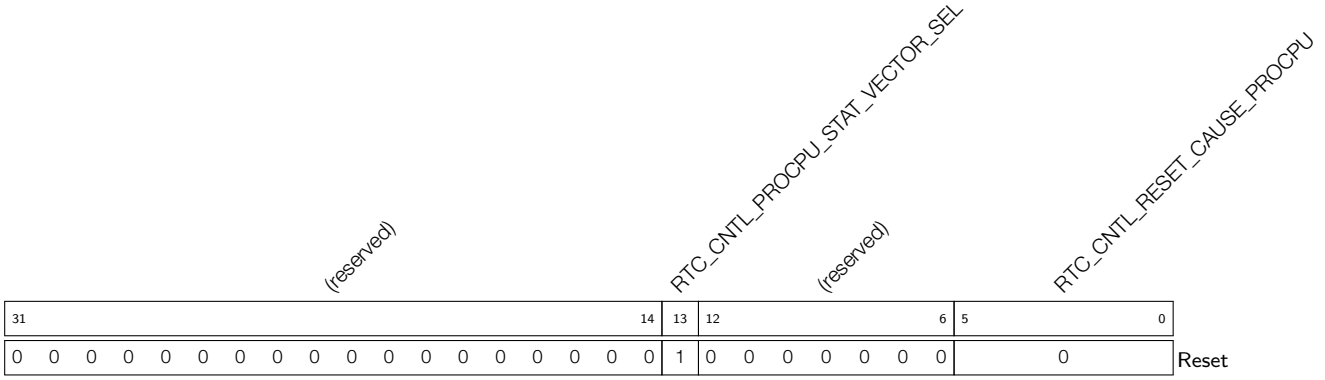
RTC_CNTL_REG_DG_WRAP_FORCE_NOISO 置 1 强制不隔离数字系统。(读/写)

Register 28.19: RTC_CNTL_LOW_POWER_ST_REG (0x00CC)



RTC_CNTL_RTC_RDY_FOR_WAKEUP 代表 RTC 已准备接受任何唤醒源的触发。(只读)

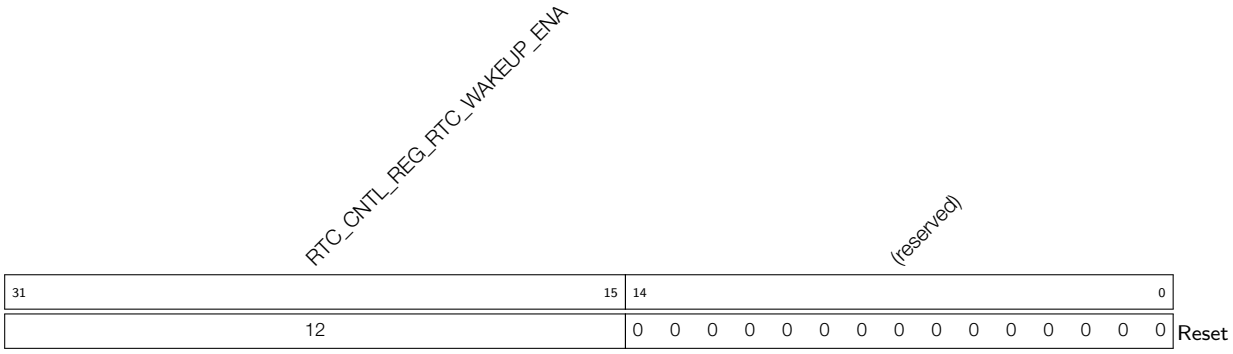
Register 28.20: RTC_CNTL_RESET_STATE_REG (0x0038)



RTC_CNTL_RESET_CAUSE_PROCPU 存储 CPU 复位原因。（只读）

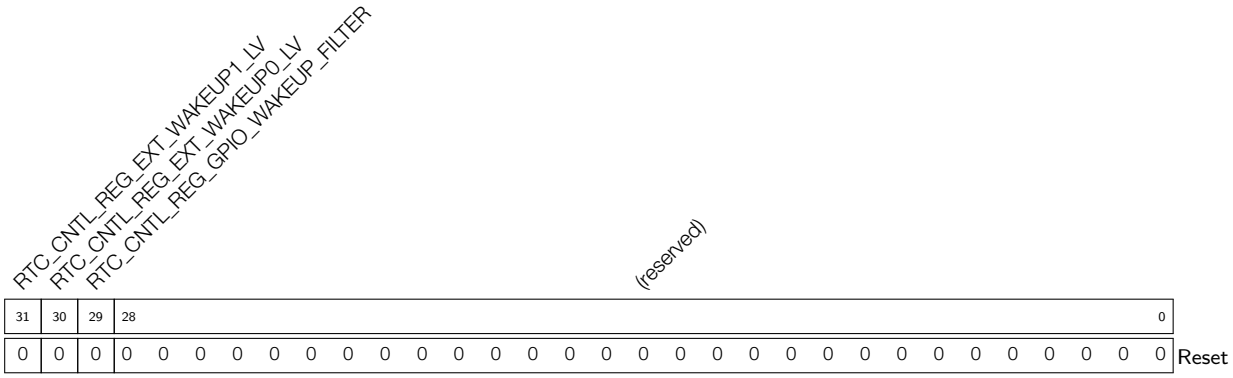
RTC_CNTL_PROCPU_STAT_VECTOR_SEL 选择 CPU 静态矢量 (static vector)。（读/写）

Register 28.21: RTC_CNTL_WAKEUP_STATE_REG (0x003C)



RTC_CNTL_REG_RTC_WAKEUP_ENA 选择唤醒源。详见表 156。（读/写）

Register 28.22: RTC_CNTL_EXT_WAKEUP_CONF_REG (0x0064)



RTC_CNTL_REG_GPIO_WAKEUP_FILTER 置 1 使能 GPIO 唤醒事件过滤器。(读/写)

RTC_CNTL_REG_EXT_WAKEUP0_LV 置 0 选择 EXT0 低电平唤醒，置 1 选择 EXT0 高电平唤醒。(读/写)

RTC_CNTL_REG_EXT_WAKEUP1_LV 置 0 选择 EXT1 低电平唤醒，置 1 选择 EXT1 高电平唤醒。(读/写)

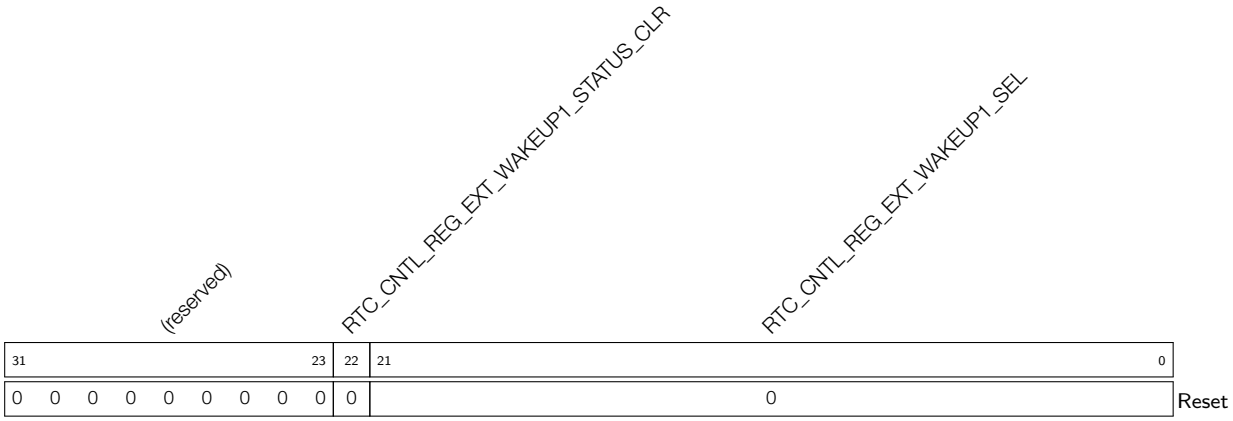
Register 28.23: RTC_CNTL_SLP_REJECT_CONF_REG (0x0068)

RTC_CNTL_REG_RTC_SLEEP_REJECT_ENA 置 1 使能“拒绝入睡”。(读/写)

RTC_CNTL_REG_LIGHT_SLP_REJECT_EN 置 1 使能“拒绝进入 Light-sleep”。(读/写)

RTC_CNTL_REG_DEEP_SLP_REJECT_EN 置 1 使能“拒绝进入 Deep-sleep”。(读/写)

Register 28.24: RTC_CNTL_EXT_WAKEUP1_REG (0x00DC)



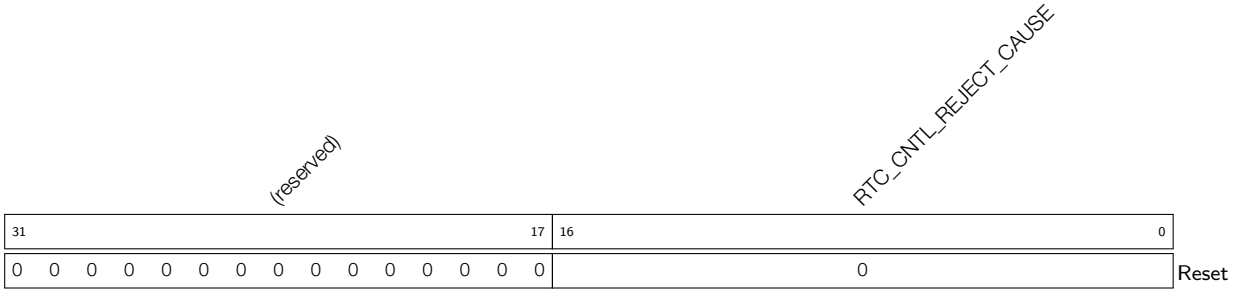
RTC_CNTL_REG_EXT_WAKEUP1_SEL 选择 RTC GPIO 为 EXT1 唤醒源。(读/写)

RTC_CNTL_REG_EXT_WAKEUP1_STATUS_CLR 清除 EXT1 唤醒状态。(只写)

Register 28.25: RTC_CNTL_EXT_WAKEUP1_STATUS_REG (0x00E0)

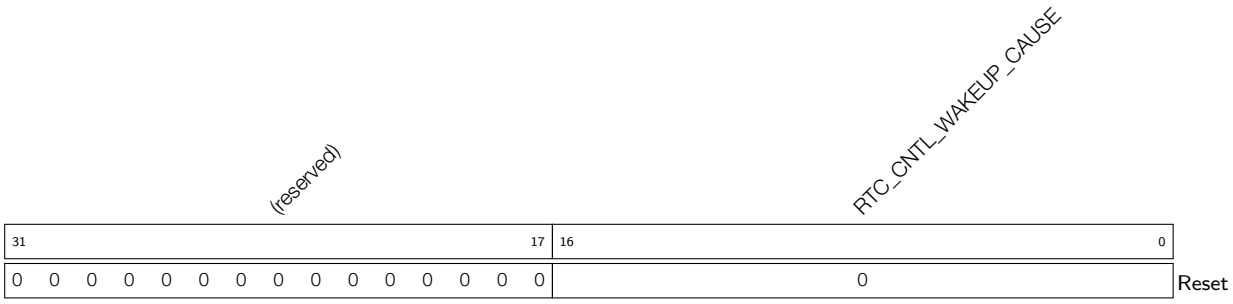
RTC_CNTL_REG_EXT_WAKEUP1_STATUS 指示 EXT1 唤醒状态。(只读)

Register 28.26: RTC_CNTL_SLP_REJECT_CAUSE_REG (0x0124)



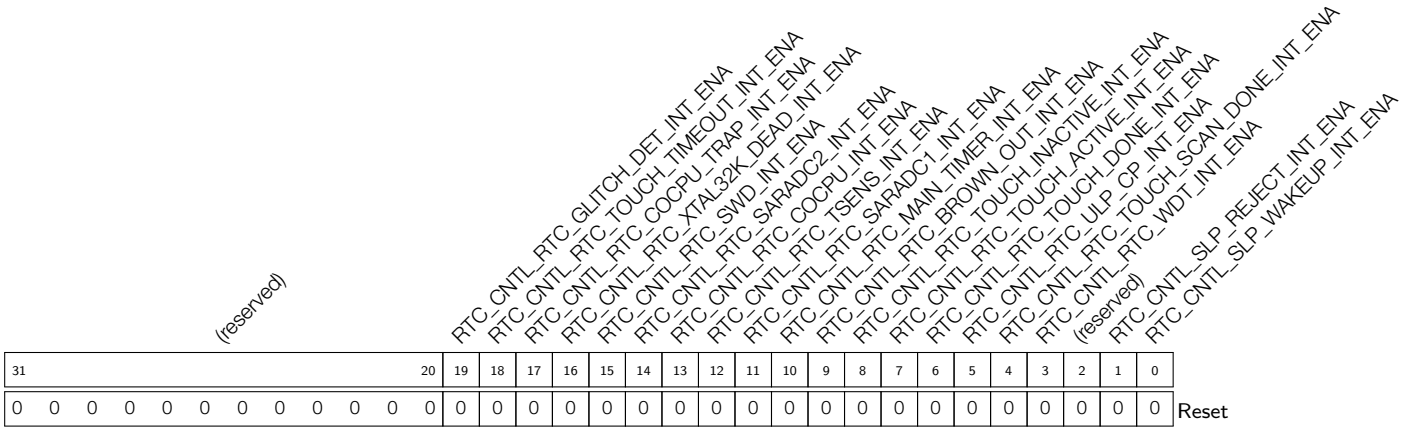
RTC_CNTL_REJECT_CAUSE 存储“拒绝入睡”的原因。(只读)

Register 28.27: RTC_CNTL_SLP_WAKEUP_CAUSE_REG (0x012C)



RTC_CNTL_WAKEUP_CAUSE 存储唤醒原因。(只读)

Register 28.28: 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。(读/写)

RTC_CNTL_SLP_REJECT_INT_ENA 使能在芯片“拒绝入睡”时发送中断。(读/写)

RTC_CNTL_RTC_WDT_INT_ENA 使能 RTC 看门狗中断。(读/写)

RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_ENA 使能在触摸完成时发送中断。(读/写)

RTC_CNTL_RTC_ULP_CP_INT_ENA 使能超低功耗协处理器中断。(读/写)

接下页...

Register 28.28: RTC_CNTL_INT_ENA_RTC_REG (0x0040)

接上页...

RTC_CNTL_RTC_TOUCH_DONE_INT_ENA 使能在单次触摸完成时发送中断。(读/写)

RTC_CNTL_RTC_TOUCH_ACTIVE_INT_ENA 使能在发生触摸时发送中断。(读/写)

RTC_CNTL_RTC_TOUCH_INACTIVE_INT_ENA 使能在触摸释放时发送中断。(读/写)

RTC_CNTL_RTC_BROWN_OUT_INT_ENA 使能欠压检测中断。(读/写)

RTC_CNTL_RTC_MAIN_TIMER_INT_ENA 使能 RTC 主计时器中断。(读/写)

RTC_CNTL_RTC_SARADC1_INT_ENA 使能 SAR ADC 1 中断。(读/写)

RTC_CNTL_RTC_TSENS_INT_ENA 使能温度传感器中断。(读/写)

RTC_CNTL_RTC_COCPU_INT_ENA 使能 ULP-RISCV 中断。(读/写)

RTC_CNTL_RTC_SARADC2_INT_ENA 使能 SAR ADC 2 中断。(读/写)

RTC_CNTL_RTC_SWD_INT_ENA 使能超级看门狗中断。(读/写)

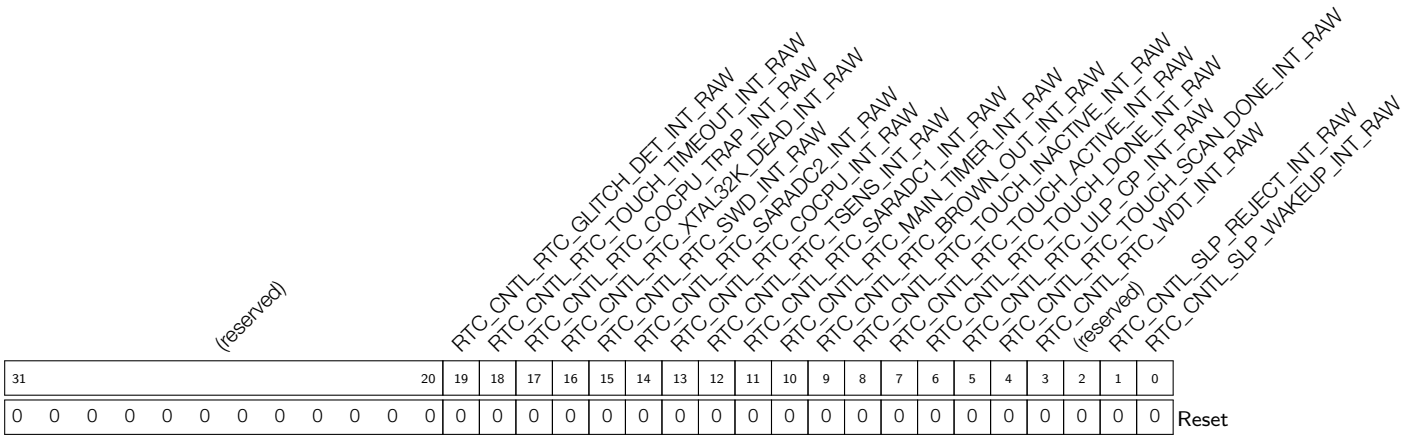
RTC_CNTL_RTC_XTAL32K_DEAD_INT_ENA 使能在 32 kHz 晶振掉电时发送中断。(读/写)

RTC_CNTL_RTC_COCPU_TRAP_INT_ENA 使能在 ULP-RISCV 被困时发送中断。(读/写)

RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_ENA 使能在触摸传感器超时时发送中断。(读/写)

RTC_CNTL_RTC_GLITCH_DET_INT_ENA 使能在检测到脉冲毛刺时发送中断。(读/写)

Register 28.29: 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。(只读)

RTC_CNTL_SLP_REJECT_INT_RAW 存储芯片“拒绝入睡”时中断的原始中断位。(只读)

RTC_CNTL_RTC_WDT_INT_RAW 存储 RTC 看门狗中断的原始中断位。(只读)

RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_RAW 存储触摸完成时中断的原始中断位。(只读)

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。(只读)

RTC_CNTL_RTC_TOUCH_DONE_INT_RAW 存储单次触摸完成时中断的原始中断位。(只读)

RTC_CNTL_RTC_TOUCH_ACTIVE_INT_RAW 存储检测到触摸时中断的原始中断位。(只读)

RTC_CNTL_RTC_TOUCH_INACTIVE_INT_RAW 存储触摸释放中断的原始中断位。(只读)

RTC_CNTL_RTC_BROWN_OUT_INT_RAW 存储欠压检测中断的原始中断位。(只读)

RTC_CNTL_RTC_MAIN_TIMER_INT_RAW 存储 RTC 主定时器中断的原始中断位。(只读)

RTC_CNTL_RTC_SARADC1_INT_RAW 存储 SAR ADC 1 中断的原始中断位。(只读)

RTC_CNTL_RTC_TSENS_INT_RAW 存储温度传感器中断的原始中断位。(只读)

RTC_CNTL_RTC_COCPU_INT_RAW 存储 ULP-RISCV 中断的原始中断位。(只读)

RTC_CNTL_RTC_SARADC2_INT_RAW 存储 SAR ADC 2 中断的原始中断位。(只读)

RTC_CNTL_RTC_SWD_INT_RAW 存储超级看门狗中断的原始中断位。(只读)

RTC_CNTL_RTC_XTAL32K_DEAD_INT_RAW 存储 32 kHz 晶振掉电中断的原始中断位。(只读)

接下页...

Register 28.29: RTC_CNTL_INT_RAW_RTC_REG (0x0044)

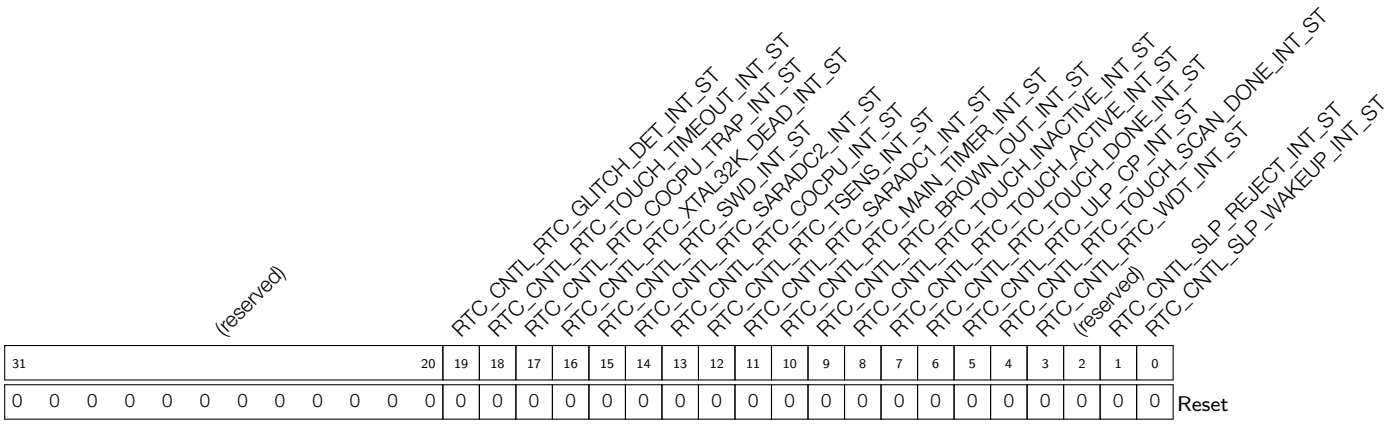
接上页...

RTC_CNTL_RTC_COCPU_TRAP_INT_RAW 存储 ULP-RISCV 受困时中断的原始中断位。(只读)

RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_RAW 存储触摸传感器超时中断的原始中断位。(只读)

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。(只读)

Register 28.30: 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。(只读)

RTC_CNTL_SLP_REJECT_INT_ST 存储芯片“拒绝进入睡眠”时中断的中断状态。(只读)

RTC_CNTL_RTC_WDT_INT_ST 存储 RTC 看门狗中断的中断状态。(只读)

RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_ST 存储触摸完成时中断的中断状态。(只读)

RTC_CNTL_RTC_ULP_CP_INT_ST 存储超低功耗协处理器中断的中断状态。(只读)

RTC_CNTL_RTC_TOUCH_DONE_INT_ST 存储单次触摸完成时中断的中断状态。(只读)

RTC_CNTL_RTC_TOUCH_ACTIVE_INT_ST 存储检测到触摸时中断的中断状态。(只读)

RTC_CNTL_RTC_TOUCH_INACTIVE_INT_ST 存储触摸释放时中断的中断状态。(只读)

RTC_CNTL_RTC_BROWN_OUT_INT_ST 存储欠压检测中断的中断状态。(只读)

RTC_CNTL_RTC_MAIN_TIMER_INT_ST 存储 RTC 主定时器中断的中断状态。(只读)

RTC_CNTL_RTC_SARADC1_INT_ST 存储 SAR ADC 1 中断的中断状态。(只读)

RTC_CNTL_RTC_TSENS_INT_ST 存储温度传感器中断的中断状态。(只读)

RTC_CNTL_RTC_COCPU_INT_ST 存储 ULP-RISCV 中断的中断状态。(只读)

RTC_CNTL_RTC_SARADC2_INT_ST 存储 SAR ADC 2 中断的中断状态。(只读)

RTC_CNTL_RTC_SWD_INT_ST 存储超级看门狗中断的中断状态。(只读)

RTC_CNTL_RTC_XTAL32K_DEAD_INT_ST 存储 32 kHz 掉电中断的中断状态。(只读)

接下页...

Register 28.30: RTC_CNTL_INT_ST_RTC_REG (0x0048)

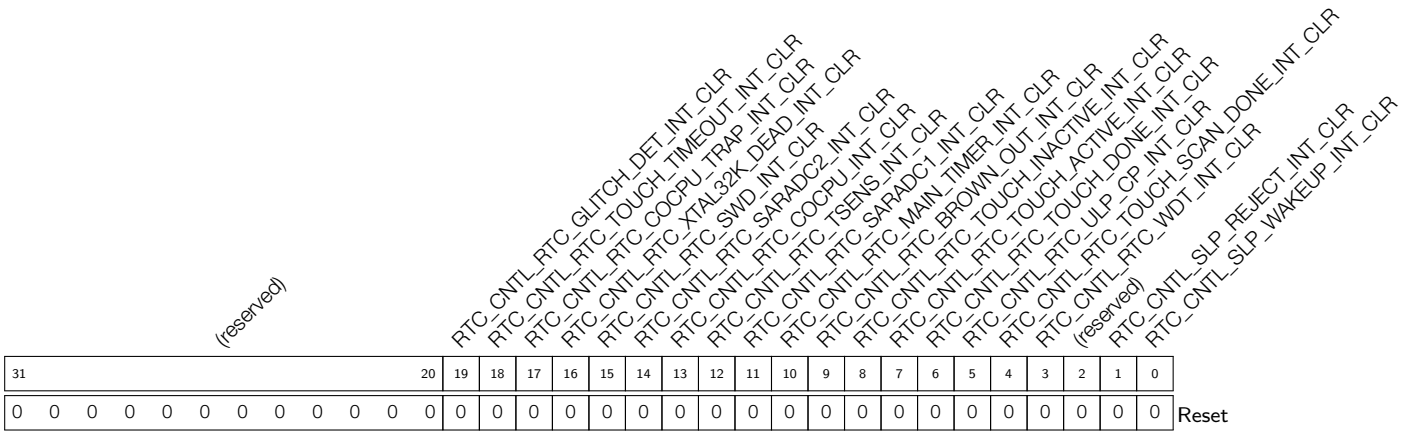
接上页...

RTC_CNTL_RTC_COCPU_TRAP_INT_ST 存储 ULP-RISCV 受困时中断的中断状态。(只读)

RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_ST 存储触摸传感器超时时中断的中断状态。(只读)

RTC_CNTL_RTC_GLITCH_DET_INT_ST 存储检测到脉冲毛刺时中断的中断状态。(只读)

Register 28.31: 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。(只写)

RTC_CNTL_SLP_REJECT_INT_CLR 清除芯片“拒绝入睡”时的中断。(只写)

RTC_CNTL_RTC_WDT_INT_CLR 清除 RTC 看门狗中断。(只写)

RTC_CNTL_RTC_TOUCH_SCAN_DONE_INT_CLR 清除触摸完成时的中断。(只写)

RTC_CNTL_RTC_ULP_CP_INT_CLR 清除 ULP 协处理器中断。(只写)

RTC_CNTL_RTC_TOUCH_DONE_INT_CLR 清除单次触摸完成时的中断。(只写)

RTC_CNTL_RTC_TOUCH_ACTIVE_INT_CLR 清除检测到触摸时的终端。(只写)

RTC_CNTL_RTC_TOUCH_INACTIVE_INT_CLR 清除触摸释放时的终端。(只写)

RTC_CNTL_RTC_BROWN_OUT_INT_CLR 清除欠压检测中断。(只写)

RTC_CNTL_RTC_MAIN_TIMER_INT_CLR 清除 RTC 主定时器中断。(只写)

RTC_CNTL_RTC_SARADC1_INT_CLR 清除 SAR ADC 1 中断。(只写)

RTC_CNTL_RTC_TSENS_INT_CLR 清除温度传感器中断。(只写)

RTC_CNTL_RTC_COCPU_INT_CLR 清除 ULP-RISCV 中断。(只写)

RTC_CNTL_RTC_SARADC2_INT_CLR 清除 SAR ADC 2 中断。(只写)

RTC_CNTL_RTC_SWD_INT_CLR 清除超级看门狗中断。(只写)

RTC_CNTL_RTC_XTAL32K_DEAD_INT_CLR 清除 32 kHz 晶振掉电中断。(只写)

接下页...

Register 28.31: RTC_CNTL_INT_CLR_RTC_REG (0x004C)

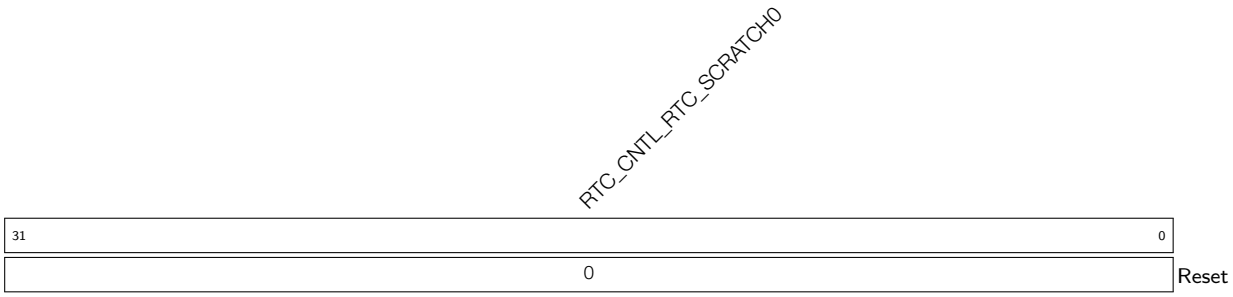
接上页...

RTC_CNTL_RTC_COCPU_TRAP_INT_CLR 清除 ULP-RISCV 受困时的中断。（只写）

RTC_CNTL_RTC_TOUCH_TIMEOUT_INT_CLR 清除触摸传感器超时时的中断。（只写）

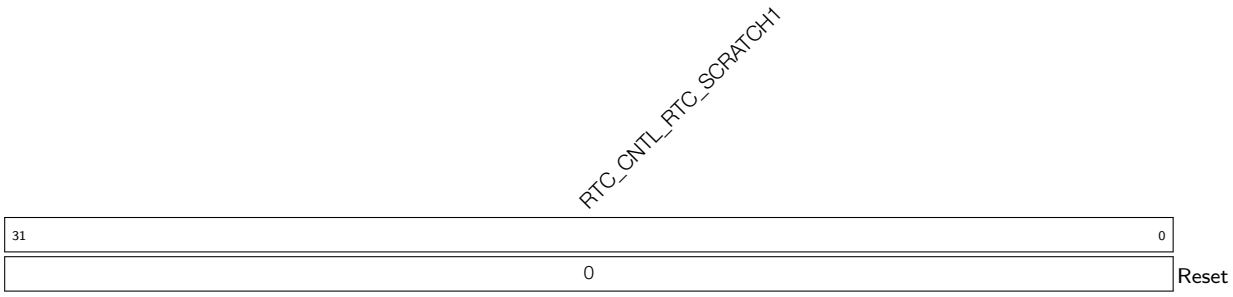
RTC_CNTL_RTC_GLITCH_DET_INT_CLR 清除检测到脉冲毛刺时的中断。（只写）

Register 28.32: RTC_CNTL_STORE0_REG (0x0050)



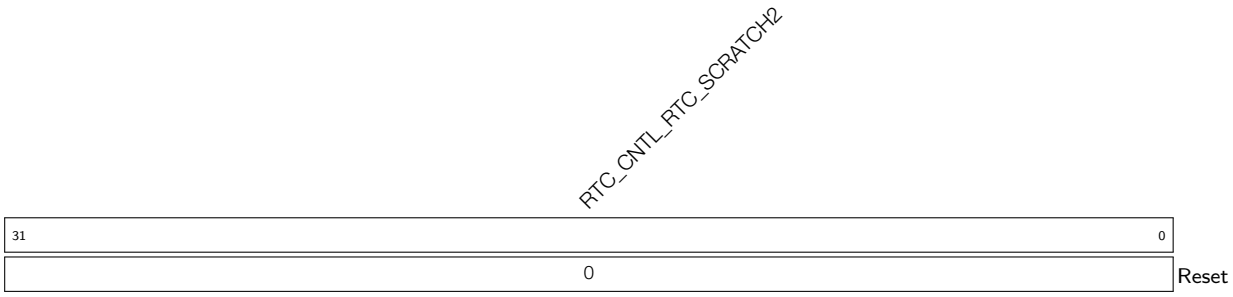
RTC_CNTL_RTC_SCRATCH0 保留寄存器 0。（读/写）

Register 28.33: RTC_CNTL_STORE1_REG (0x0054)



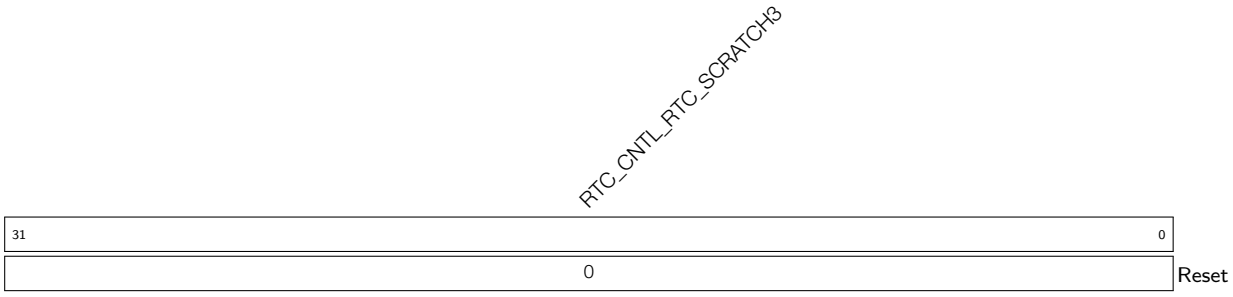
RTC_CNTL_RTC_SCRATCH1 保留寄存器 1。（读/写）

Register 28.34: RTC_CNTL_STORE2_REG (0x0058)



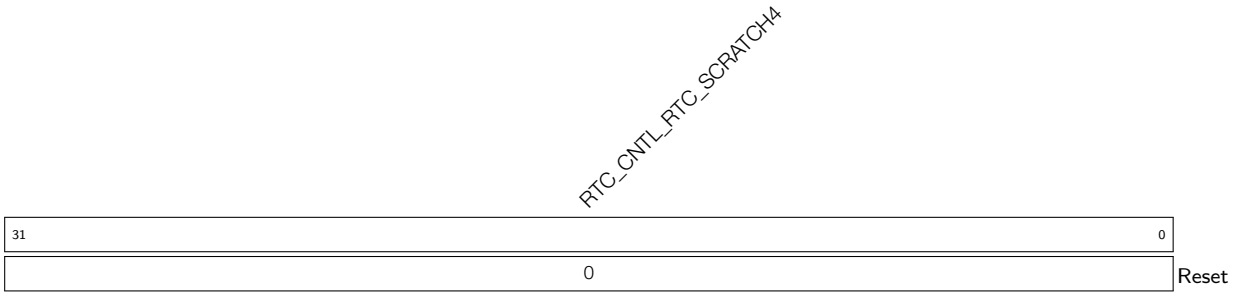
RTC_CNTL_RTC_SCRATCH2 保留寄存器 2。（读/写）

Register 28.35: RTC_CNTL_STORE3_REG (0x005C)



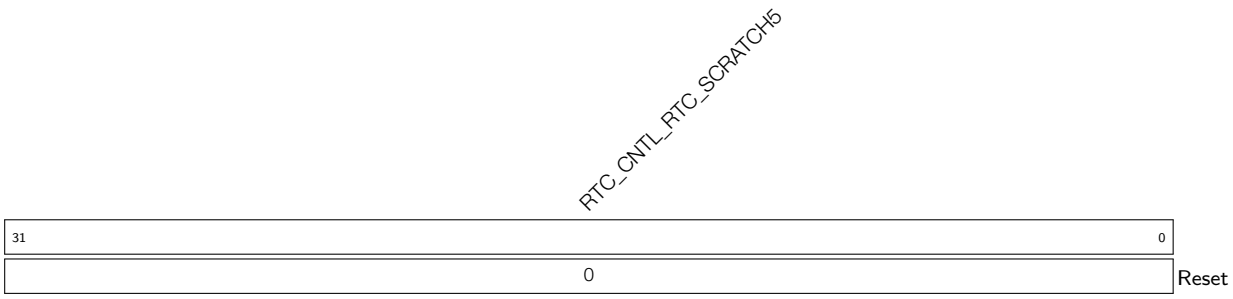
RTC_CNTL_RTC_SCRATCH3 保留寄存器 3。（读/写）

Register 28.36: RTC_CNTL_STORE4_REG (0x00BC)



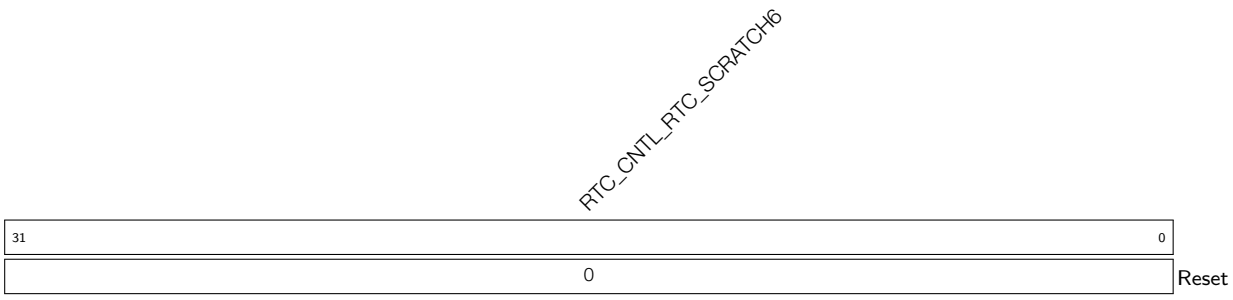
RTC_CNTL_RTC_SCRATCH4 保留寄存器 4。（读/写）

Register 28.37: RTC_CNTL_STORE5_REG (0x00C0)



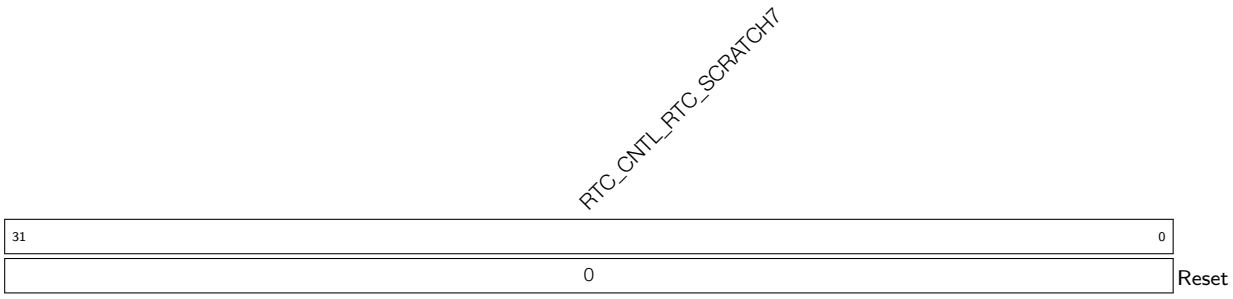
RTC_CNTL_RTC_SCRATCH5 保留寄存器 5。(读/写)

Register 28.38: RTC_CNTL_STORE6_REG (0x00C4)



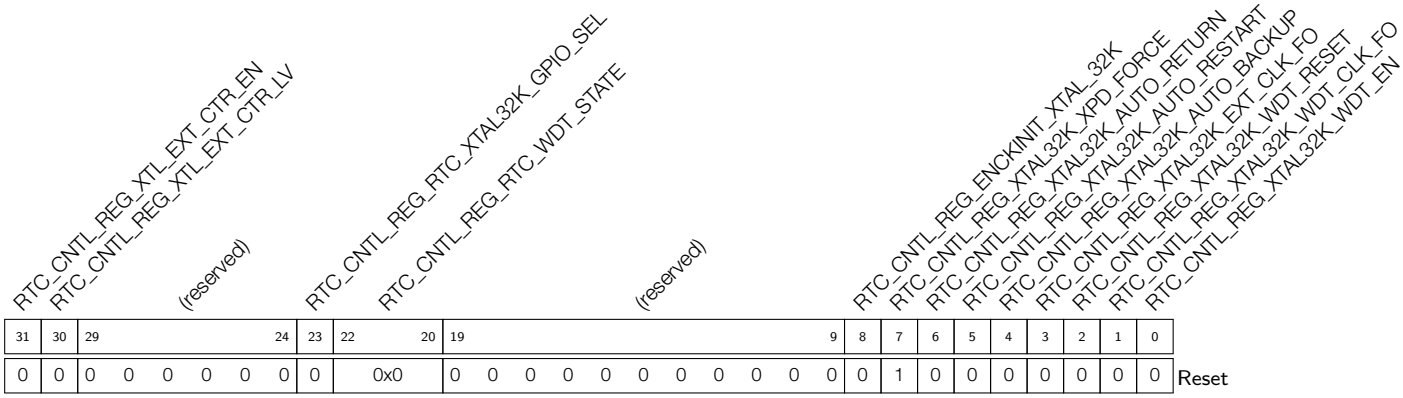
RTC_CNTL_RTC_SCRATCH6 保留寄存器 6。(读/写)

Register 28.39: RTC_CNTL_STORE7_REG (0x00C8)



RTC_CNTL_RTC_SCRATCH7 保留寄存器 7。(读/写)

Register 28.40: RTC_CNTL_EXT_XTL_CONF_REG (0x0060)



RTC_CNTL_REG_XTAL32K_WDT_EN 置 1 使能 32 晶振看门狗。(读/写)

RTC_CNTL_REG_XTAL32K_WDT_CLK_FO 置 1 强制打开 32 kHz 晶振看门狗时钟。(读/写)

RTC_CNTL_REG_XTAL32K_WDT_RESET 置 1 软件复位 32 kHz 晶振看门狗。(读/写)

RTC_CNTL_REG_XTAL32K_EXT_CLK_FO 置 1 强制打开 32 kHz 晶振的外部时钟。(读/写)

RTC_CNTL_REG_XTAL32K_AUTO_BACKUP 置 1 在 32 kHz 晶振掉电时切换至后备时钟。(读/写)

RTC_CNTL_REG_XTAL32K_AUTO_RESTART 置 1 在 32 kHz 晶振掉电时自动重启 32 kHz 晶振。(读/写)

RTC_CNTL_REG_XTAL32K_AUTO_RETURN 置 1 在 32 kHz 晶振重启时自动切换回 32 kHz 晶振。(读/写)

RTC_CNTL_REG_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。(读/写)

RTC_CNTL_REG_ENCKINIT_XTAL_32K 置 1 使用内部时钟协助 32 kHz 晶振重启。(读/写)

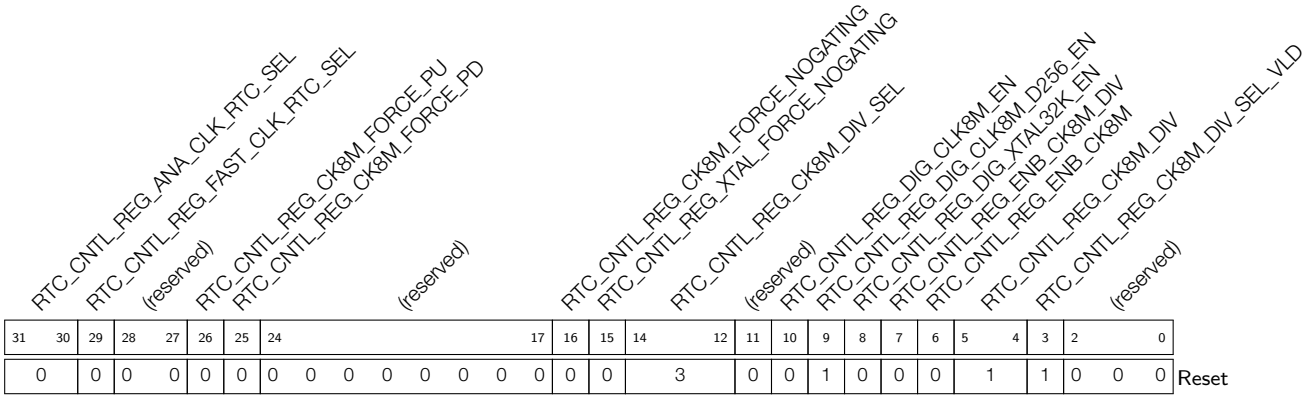
RTC_CNTL_REG_RTC_WDT_STATE 存储 32 kHz 看门狗的状态。(只读)

RTC_CNTL_REG_RTC_XTAL32K_GPIO_SEL 选择 32 kHz 晶振时钟。置 0 选择外部 32 kHz 时钟，置 1 选择来自 RTC GPIO 管脚 X32P_C 的时钟。(读/写)

RTC_CNTL_REG_XTL_EXT_CTR_LV 置 1 设置 XTAL 低电平掉电，置 0 设置 XTAL 高电平掉电。(读/写)

RTC_CNTL_REG_XTL_EXT_CTR_EN 置 1 使能 GPIO 关闭晶振。(读/写)

Register 28.41: RTC_CNTL_CLK_CONF_REG (0x0074)



RTC_CNTL_REG_CK8M_DIV_SEL_VLD 同步 reg_ck8m_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。(读/写)

RTC_CNTL_REG_CK8M_DIV 设置 CK8M_D256_OUT 分频器。00: 128 分频，01：256 分频，10：512 分频，11：1024 分频。(读/写)

RTC_CNTL_REG_ENB_CK8M 置 1 禁用 CK8M 和 CK8M_D256_OUT。(读/写)

RTC_CNTL_REG_ENB_CK8M_DIV 选择 CK8M_D256_OUT 时钟。置 1 选择 CK8M，置 0 选择 CK8M 的 256 分频。(读/写)

RTC_CNTL_REG_DIG_XTAL32K_EN 置 1 使能数字内核的 CK_XTAL_32K 时钟。(读/写)

RTC_CNTL_REG_DIG_CLK8M_D256_EN 置 1 使能数字内核的 CK8M_D256_OUT 时钟。(读/写)

RTC_CNTL_REG_DIG_CLK8M_EN 置 1 使能数字内核的 8 MHz 时钟。(读/写)

RTC_CNTL_REG_CK8M_DIV_SEL 存储 8 MHz 分频器，即 reg_ck8m_div_sel + 1。(读/写)

RTC_CNTL_REG_XTAL_FORCE_NOGATING 置 1 强制关闭 Sleep 状态下的晶振门控。(读/写)

RTC_CNTL_REG_CK8M_FORCE_NOGATING 置 1 强制关闭 Sleep 状态下的 8 MHz 晶振门控。(读/写)

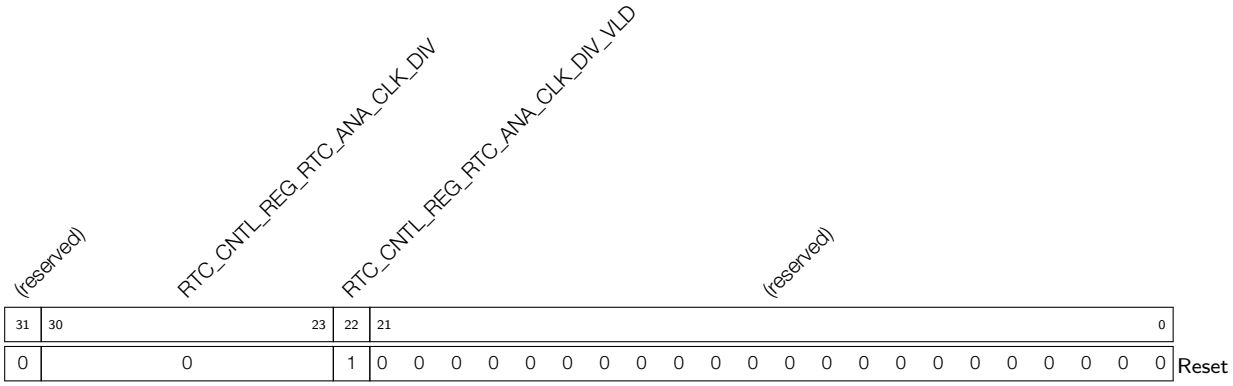
RTC_CNTL_REG_CK8M_FORCE_PD 置 1 强制关闭 8 MHz 时钟。(读/写)

RTC_CNTL_REG_CK8M_FORCE_PU 置 1 强制打开 8 MHz 时钟。(读/写)

RTC_CNTL_REG_FAST_CLK_RTC_SEL 选择 RTC 快速时钟。置 1 选择 XTAL 的四分频，置 0 选择 CK8M。(读/写)

RTC_CNTL_REG_ANA_CLK_RTC_SEL 选择 RTC 慢速时钟。0：90K rtc_clk，1：32k XTAL，2：8md256。(读/写)

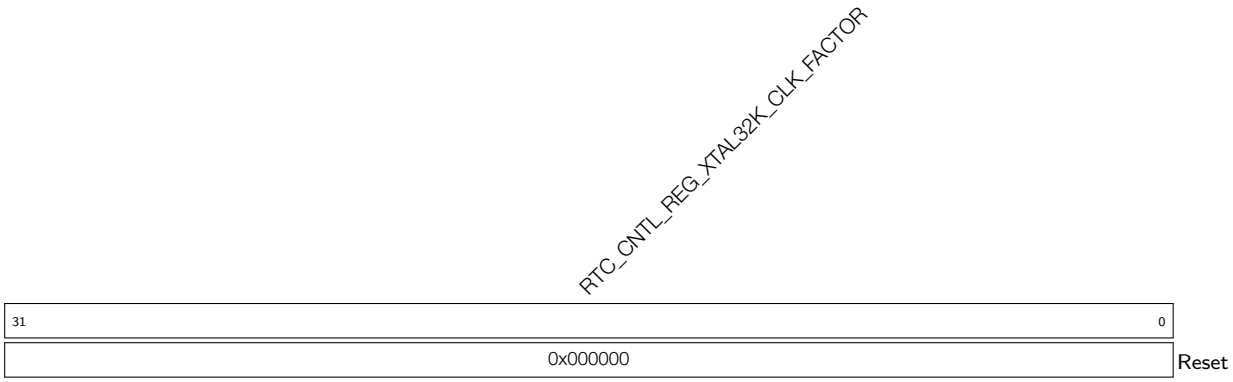
Register 28.42: RTC_CNTL_SLOW_CLK_CONF_REG (0x0078)



RTC_CNTL_REG_RTC_ANA_CLK_DIV_VLD 同步 reg_rtc_ana_clk_div 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。

RTC_CNTL_REG_RTC_ANA_CLK_DIV 设置 RTC 时钟分频器。（读/写）

Register 28.43: RTC_CNTL_XTAL32K_CLK_FACTOR_REG (0x00F0)



RTC_CNTL_REG_XTAL32K_CLK_FACTOR 设置 32 kHz 晶振的分频器系数。（读/写）

Register 28.44: RTC_CNTL_XTAL32K_CONF_REG (0x00F4)

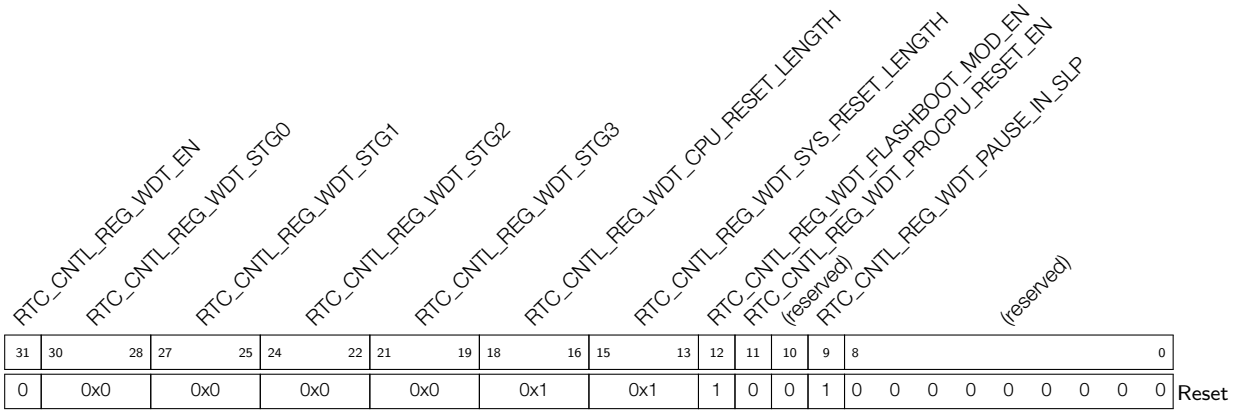
RTC_CNTL_REG_XTAL32K_RETURN_WAIT 设置切换回 32 kHz 晶振之前的等待周期。(读/写)

RTC_CNTL_REG_XTAL32K_RESTART_WAIT 设置重启 32 kHz 晶振之前的等待周期。(读/写)

RTC_CNTL_REG_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。(读/写)

RTC_CNTL_REG_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。(读/写)

Register 28.45: RTC_CNTL_WDTCONFIG0_REG (0x0094)



RTC_CNTL_REG_WDT_PAUSE_IN_SLP 置 1 设置”睡眠中看门狗暂停使用“。(读/写)

RTC_CNTL_REG_WDT_PROCPU_RESET_EN 置 1 允许看门狗重启 CPU。(读/写)

RTC_CNTL_REG_WDT_FLASHBOOT_MOD_EN 置 1 在芯片从 flash 重启时使能看门狗。(读/写)

RTC_CNTL_REG_WDT_SYS_RESET_LENGTH 设置系统复位计数器的长度。(读/写)

RTC_CNTL_REG_WDT_CPU_RESET_LENGTH 设置 CPU 复位计数器的长度。(读/写)

RTC_CNTL_REG_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。(读/写)

RTC_CNTL_REG_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。(读/写)

RTC_CNTL_REG_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。(读/写)

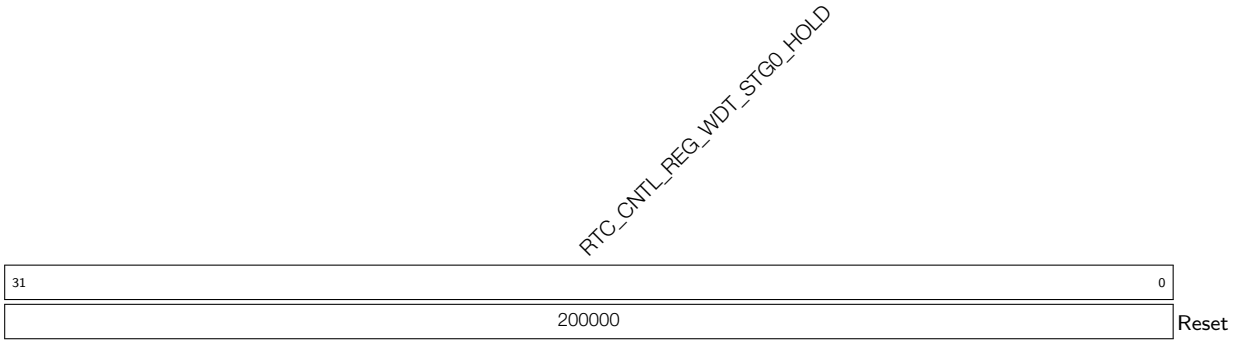
RTC_CNTL_REG_WDT_STG0 1：在中断阶段使能，2：在 CPU 阶段使能，3：在系统阶段使能，4：在系统和 RTC 阶段使能。(读/写)

RTC_CNTL_REG_WDT_EN 置 1 使能 RTC 看门狗。(读/写)

说明：

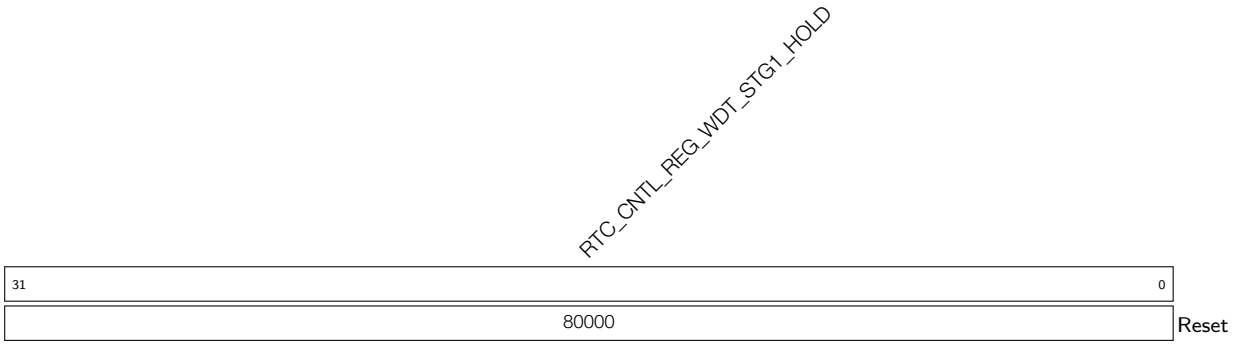
更多详情，请见章节 13 看门狗定时器。

Register 28.46: RTC_CNTL_WDTCONFIG1_REG (0x0098)



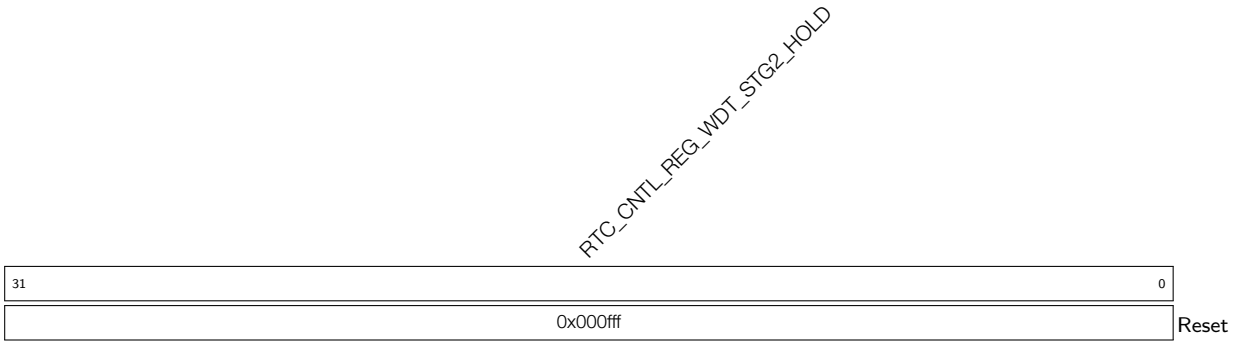
RTC_CNTL_REG_WDT_STG0_HOLD 配置阶段 1 的 RTC 看门狗保持时间。（读/写）

Register 28.47: RTC_CNTL_WDTCONFIG2_REG (0x009C)



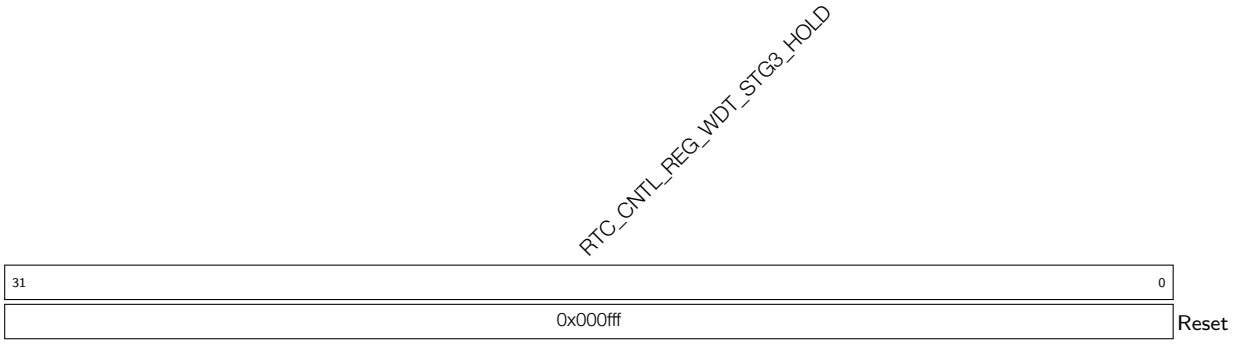
RTC_CNTL_REG_WDT_STG1_HOLD 配置阶段 2 的 RTC 看门狗保持时间。（读/写）

Register 28.48: RTC_CNTL_WDTCONFIG3_REG (0x00A0)



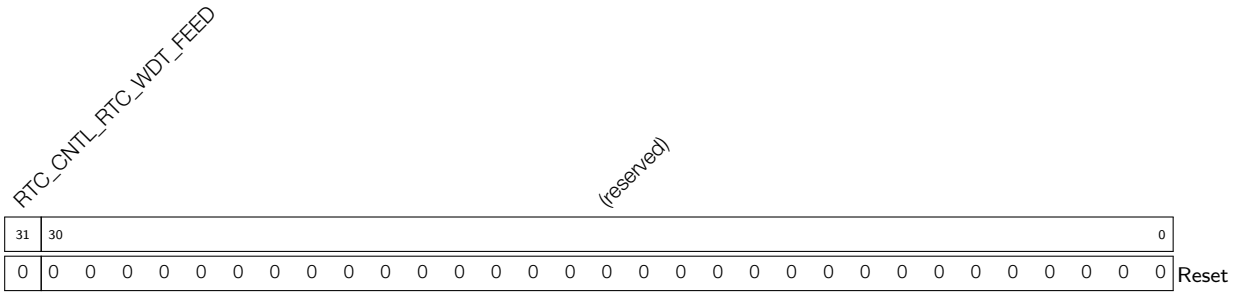
RTC_CNTL_REG_WDT_STG2_HOLD 配置阶段 3 的 RTC 看门狗保持时间。（读/写）

Register 28.49: RTC_CNTL_WDTCONFIG4_REG (0x00A4)



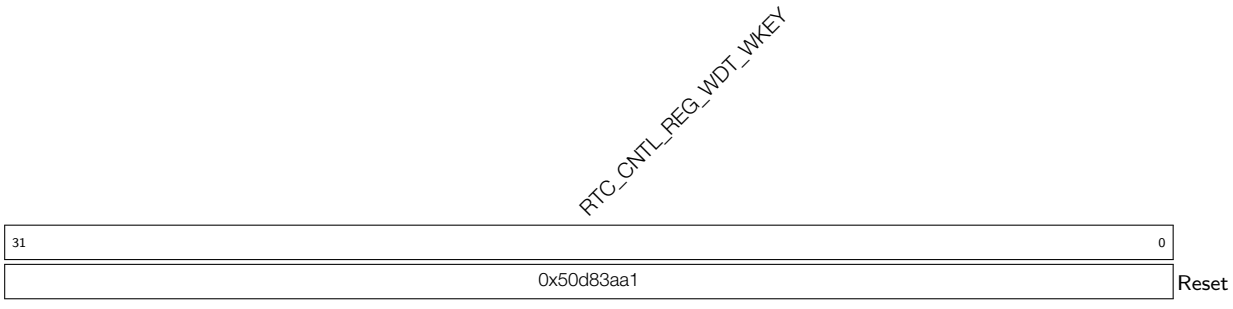
RTC_CNTL_REG_WDT_STG3_HOLD 配置阶段 4 的 RTC 看门狗保持时间。(读/写)

Register 28.50: RTC_CNTL_WDTFEED_REG (0x00A8)



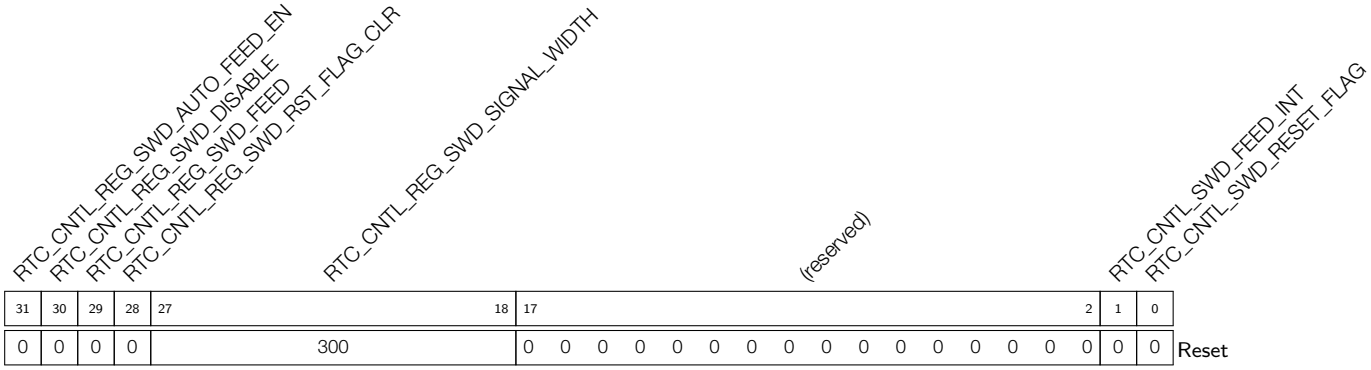
RTC_CNTL_RTC_WDT_FEED 置 1 开始喂狗。(只写)

Register 28.51: RTC_CNTL_WDTWPROTECT_REG (0x00AC)



RTC_CNTL_REG_WDT_WKEY 置 1 设置看门狗的写保护。(读/写)

Register 28.52: RTC_CNTL_SWD_CONF_REG (0x00B0)



RTC_CNTL_SWD_RESET_FLAG 代表超级看门狗的复位旗帜。(只读)

RTC_CNTL_SWD_FEED_INT 该中断产生则通过软件进行超级看门狗喂狗。(只读)

RTC_CNTL_REG_SWD_SIGNAL_WIDTH 调节传递给超级看门狗的信号宽度。(读/写)

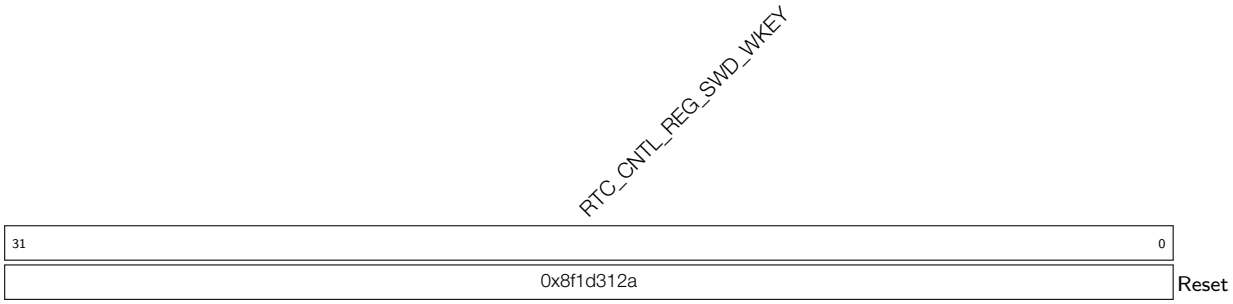
RTC_CNTL_REG_SWD_RST_FLAG_CLR 置 1 清除超级看门狗复位旗帜。(只写)

RTC_CNTL_REG_SWD_FEED 置 1 开始软件喂超级看门狗。(只写)

RTC_CNTL_REG_SWD_DISABLE 置 1 禁用超级看门狗。(读/写)

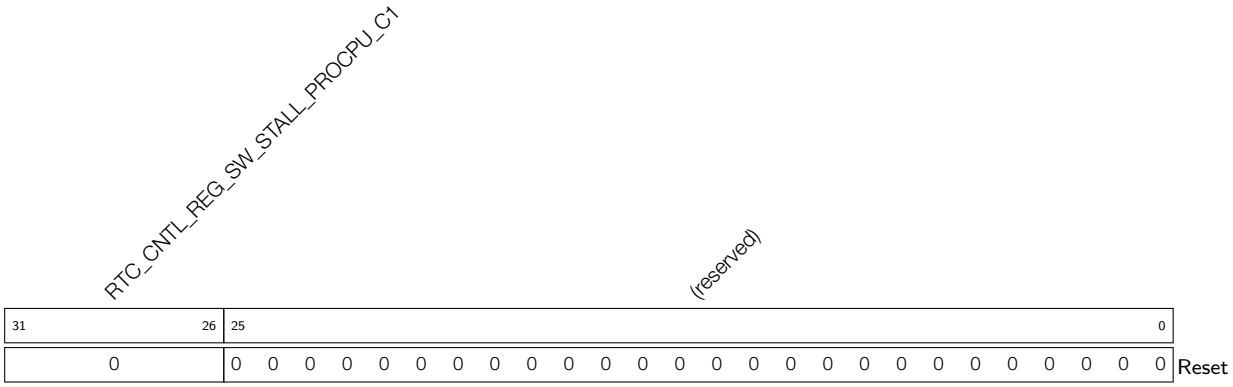
RTC_CNTL_REG_SWD_AUTO_FEED_EN 置 1 使能在发生中断时自动喂狗。(读/写)

Register 28.53: RTC_CNTL_SWD_WPROTECT_REG (0x00B4)



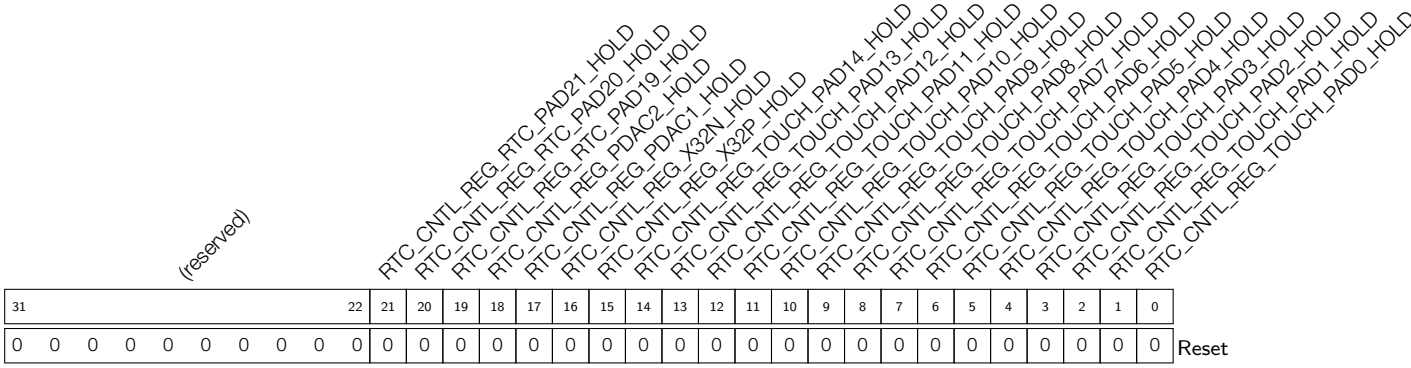
RTC_CNTL_REG_SWD_WKEY 设置超级看门狗的写保护密钥。(读/写)

Register 28.54: RTC_CNTL_SW_CPU_STALL_REG (0x00B8)



RTC_CNTL_REG_SW_STALL_PROCPU_C1 当 RTC_CNTL_REG_SW_STALL_PROCPU_C0 配置为 0x2 时，设置该位为 0x21 将软件使 CPU 进入 stall 状态。(读/写)

Register 28.55: RTC_CNTL_PAD_HOLD_REG (0x00D4)



RTC_CNTL_REG_TOUCH_PAD0_HOLD 置 1 使触摸 GPIO 0 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD1_HOLD 置 1 使触摸 GPIO 1 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD2_HOLD 置 1 使触摸 GPIO 2 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD3_HOLD 置 1 使触摸 GPIO 3 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD4_HOLD 置 1 使触摸 GPIO 4 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD5_HOLD 置 1 使触摸 GPIO 5 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD6_HOLD 置 1 使触摸 GPIO 6 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD7_HOLD 置 1 使触摸 GPIO 7 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD8_HOLD 置 1 使触摸 GPIO 8 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD9_HOLD 置 1 使触摸 GPIO 9 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD10_HOLD 置 1 使触摸 GPIO 10 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD11_HOLD 置 1 使触摸 GPIO 11 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD12_HOLD 置 1 使触摸 GPIO 12 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD13_HOLD 置 1 使触摸 GPIO 13 进入 hold 状态。(读/写)

RTC_CNTL_REG_TOUCH_PAD14_HOLD 置 1 使触摸 GPIO 14 进入 hold 状态。(读/写)

RTC_CNTL_REG_X32P_HOLD 置 1 使 x32p 进入 hold 状态。(读/写)

RTC_CNTL_REG_X32N_HOLD 置 1 使 x32n 进入 hold 状态。(读/写)

RTC_CNTL_REG_PDAC1_HOLD 置 1 使 pdac1 进入 hold 状态。(读/写)

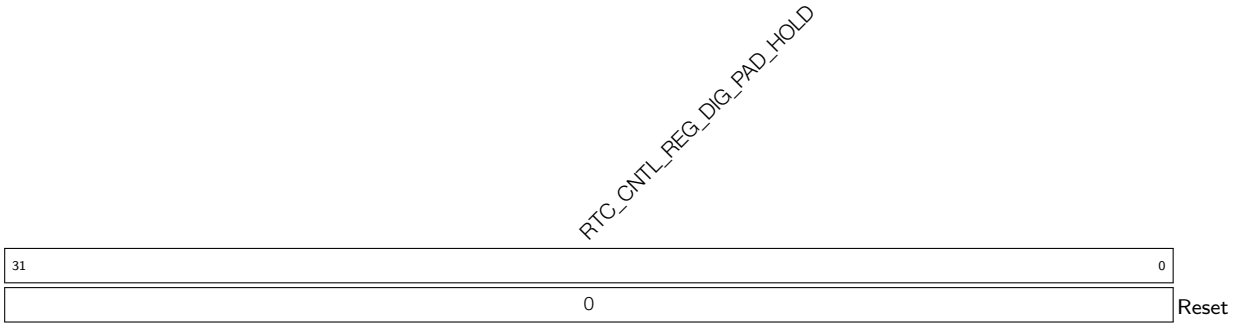
RTC_CNTL_REG_PDAC2_HOLD 置 1 使 pdac2 进入 hold 状态。(读/写)

RTC_CNTL_REG_RTC_PAD19_HOLD 置 1 使触摸 GPIO 19 进入 hold 状态。(读/写)

RTC_CNTL_REG_RTC_PAD20_HOLD 置 1 使触摸 GPIO 20 进入 hold 状态。(读/写)

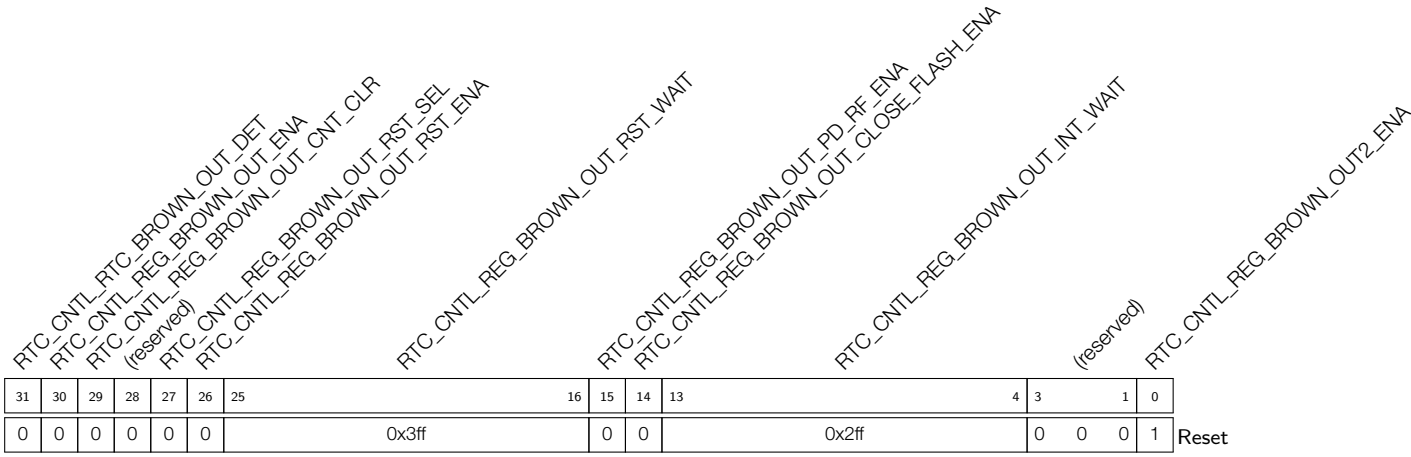
RTC_CNTL_REG_RTC_PAD21_HOLD 置 1 使触摸 GPIO 21 进入 hold 状态。(读/写)

Register 28.56: RTC_CNTL_DIG_PAD_HOLD_REG (0x00D8)



RTC_CNTL_REG_DIG_PAD_HOLD 置 1 使 GPIO 21 到 GPIO 45 进入 hold 状态。其中，GPIO 的位置可见芯片位图。(读/写)

Register 28.57: RTC_CNTL_BROWN_OUT_REG (0x00E4)



RTC_CNTL_REG_BROWN_OUT2_ENA 置 1 使能欠压掉电触发芯片复位。(读/写)

RTC_CNTL_REG_BROWN_OUT_INT_WAIT 配置发送欠压掉电中断前的等待周期。(读/写)

RTC_CNTL_REG_BROWN_OUT_CLOSE_FLASH_ENA 置 1 使能在发生欠压掉电时强制关闭 flash。(读/写)

RTC_CNTL_REG_BROWN_OUT_PD_RF_ENA 置 1 使能在发生欠压掉电前强制关闭 RF 电路。(读/写)

RTC_CNTL_REG_BROWN_OUT_RST_WAIT 配置欠压掉电后芯片重启之前的等待周期。(读/写)

RTC_CNTL_REG_BROWN_OUT_RST_ENA 置 1 使能欠压掉电复位。(读/写)

RTC_CNTL_REG_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。(读/写)

RTC_CNTL_REG_BROWN_OUT_CNT_CLR 清除欠压检测计数器。(只写)

RTC_CNTL_REG_BROWN_OUT_ENA 置 1 使能欠压检测。(读/写)

RTC_CNTL_RTC_BROWN_OUT_DET 指示欠压掉电信号状态。(只读)

修订历史

日期	版本	发布说明
2020-05-21	V0.4	增加以下章节： • 第 18 章 *TWAI* • 第 28 章 *低功耗管理* 更新以下章节： • 在章节 5 *IO MUX 和 GPIO 交换矩阵* 中新增 5.4 *专用 GPIO* • 在章节 16 *eFuse 控制器*中将 EFUSE_PGM_DATA1_REG[16] 位从供硬件使用，对软件不可用的表单中删除；增加 EFUSE_RPT4_RESERVED5 和 EFUSE_RPT4_RESERVED5_ERR 位的描述 • 更新第 22 章 *随机数发生器* 中的基地址
2020-04-01	V0.3	增加以下章节： • 第 14 章 *XTAL32K 看门狗* • 第 15 章 *系统定时器* • 第 24 章 *权限控制* • 第 26 章 *HMAC 模块* • 第 27 章 *超低功耗协处理器* 更新 RTC_CLK 频率。
2020-01-20	V0.2	增加以下章节： • 第 5 章 *IO MUX 和 GPIO 交换矩阵* • 第 7 章 *DMA 控制器* • 第 8 章 *通用异步收发传输器 (UART)* 更新 eFuse 烧写时序、读取时序参数配置
2019-11-27	V0.1	预发布。