ESP32-S2

技术参考手册

预发布 V0.1

乐鑫信息科技

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。

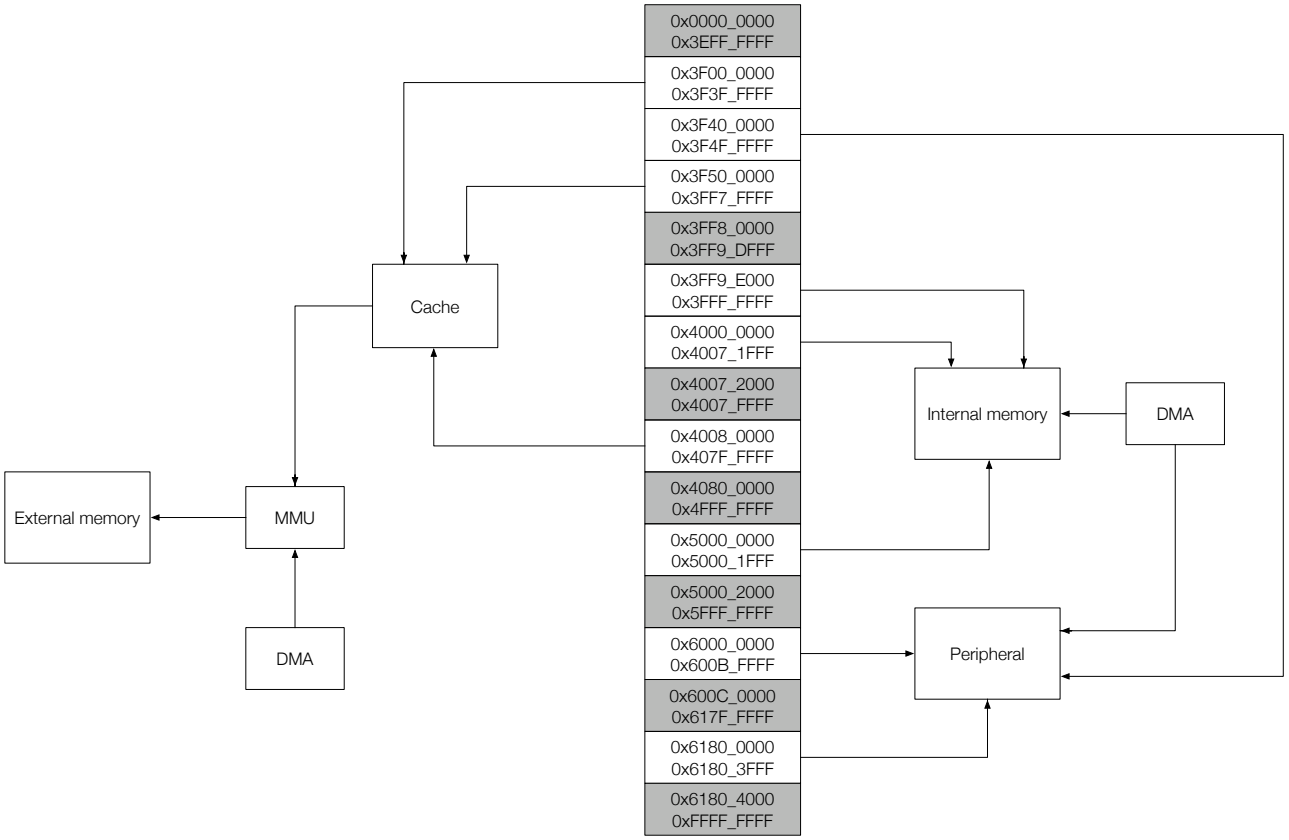


图 1-1. 系统结构与地址映射结构

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 1-1 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 1-1. 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 1-2 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 1-2. 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 1-2 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 1-2 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 1-2 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 1-2 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 1-2 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 1-2 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 1-2 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 1-3 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 1-3. 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 1-3 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。

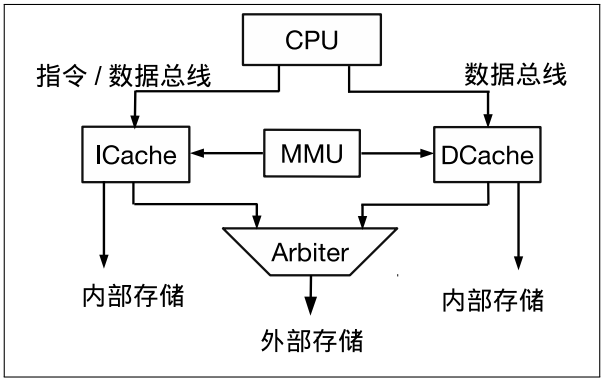


图 1-2. Cache 系统框图

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 1-4 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 1-4. 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 1-5 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 1-5. 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

目标	边界地址		Size	说明
	低位地址	高位地址		
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		
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 1-6 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 1-6. 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

目标外设	受限地址值 / 区间
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。

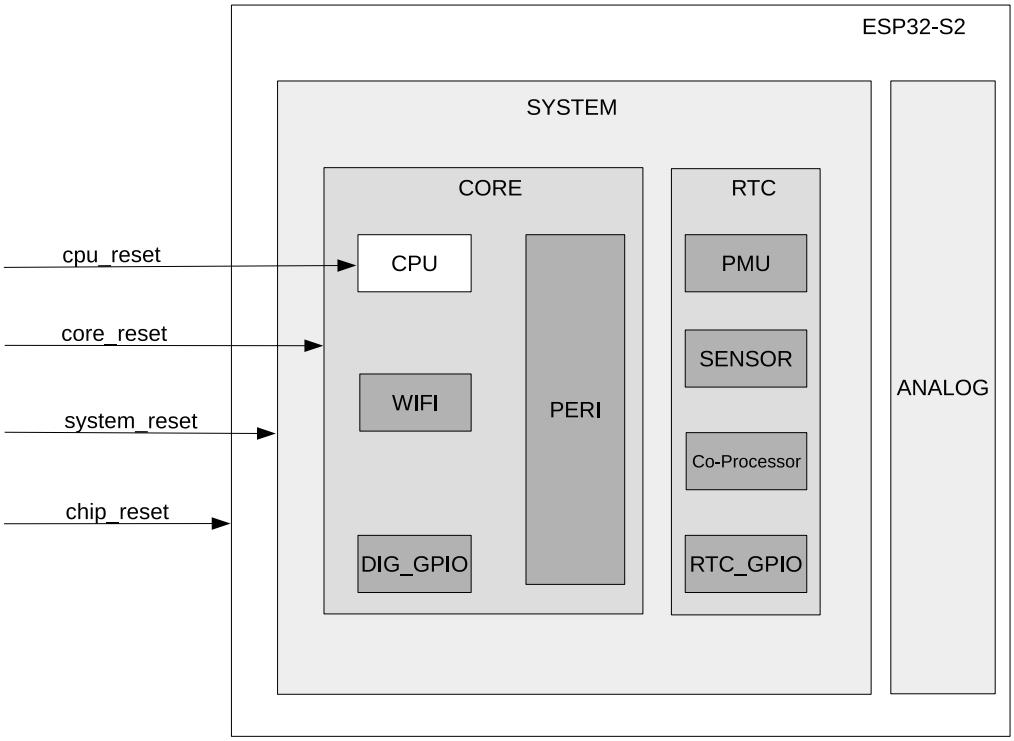


图 2-1. 四种复位等级

- 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 2-1 列出了从这一寄存器中可能读出的复位源。

表 2-1. 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
0x10	RWDT 系统复位	系统复位	详见章节 10 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	-
0x07	MWDT0 全局复位	内核复位	详见章节 10 看门狗定时器
0x08	MWDT1 全局复位	内核复位	详见章节 10 看门狗定时器
0x09	RWDT 内核复位	内核复位	详见章节 10 看门狗定时器
0x0B	MWDT0 CPU 复位	CPU 复位	详见章节10 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 10 看门狗定时器
0x11	MWDT1 CPU 复位	CPU 复位	详见章节 10 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
  - 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。

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。

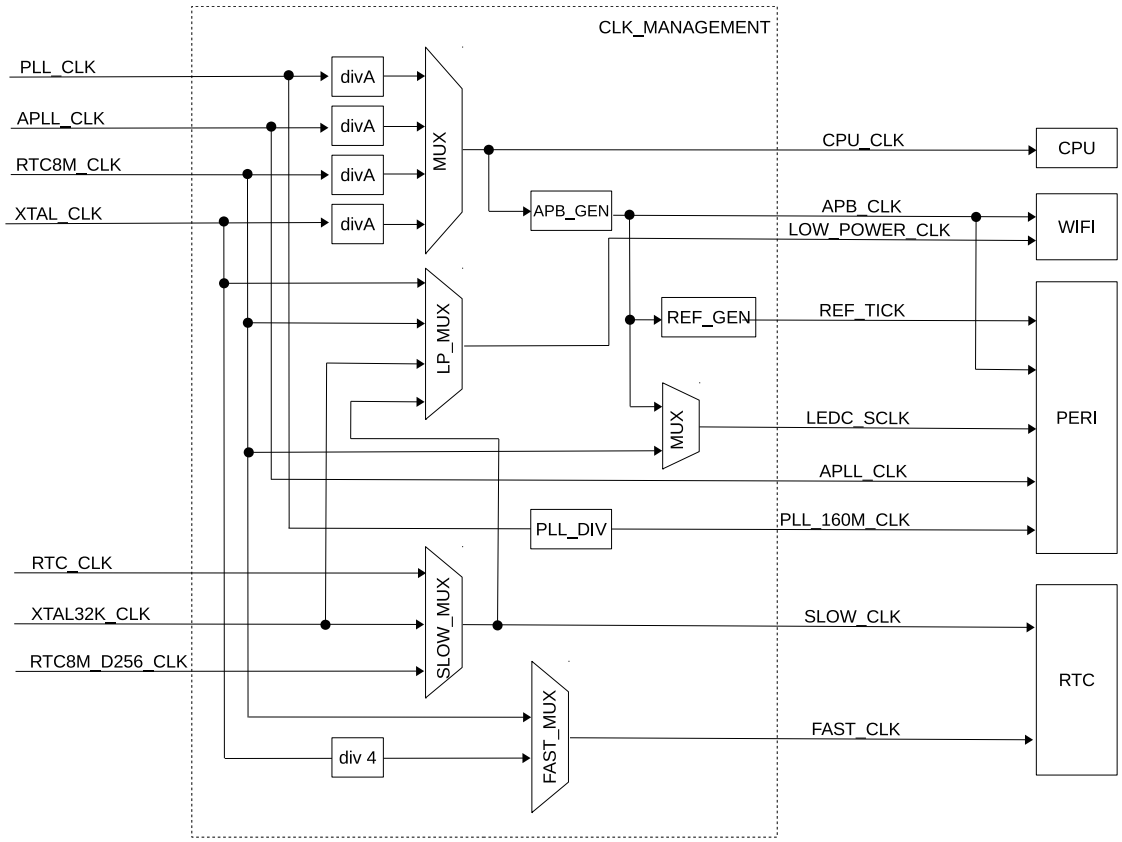


图 2-2. 系统时钟

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：150 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 2-2 和表 2-3。

表 2-2. CPU_CLK 源

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

表 2-3. 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 2-4 为接入各个外设的时钟。

表 2-4. 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 2-5 所示，APB_CLK 的频率由 CPU_CLK 的时钟源决定。

表 2-5. 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 2-6 所示：

表 2-6. 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 2-7 所示。

表 2-7. 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 2-4。

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 3-1 所示。

表 3-1. 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 3-2. 系统启动模式

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

表 3-2 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_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_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 UART_PRINT_CONTROL 一起控制 ROM Code 打印。

表 3-3. 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 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_FORC 设置为 1 时，可关闭上述功能。此时 VDD_SPI 电压由 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。

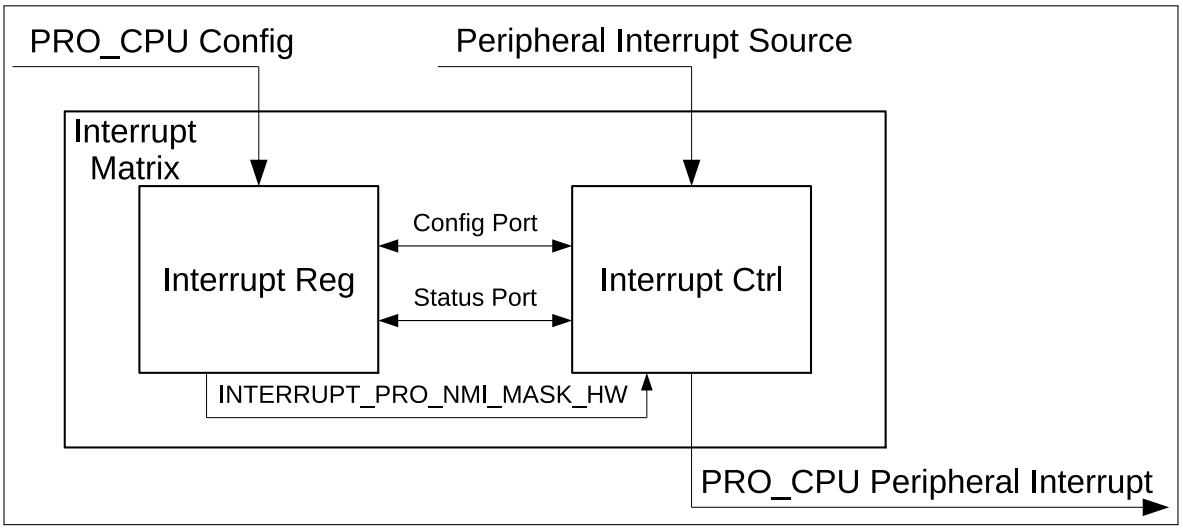


图 4-1. 中断矩阵结构图

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 4-1 列出了所有外部中断源，以及对应的中断配置寄存器与中断状态寄存器。上述 95 个外部中断源均可分配至 CPU 外部中断。

表 4-1. CPU 外部中断配置寄存器、外部中断状态寄存器、外部中断源

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

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

No.	中断源	配置寄存器	状态寄存器	
			位	名称
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 4-2 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 4-2. 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 4-1。
- 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 4-1 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 4-2。
- 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 4-2。

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 4-1 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 4-3 所示。更多信息，请访问章节 1 *系统和存储器*。

表 4-3. 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
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

名称	描述	地址	访问
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
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

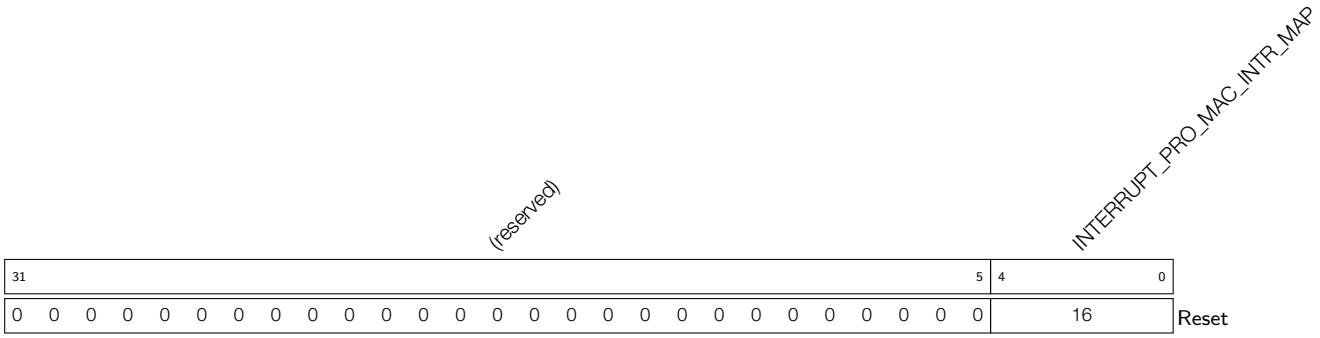
名称	描述	地址	访问
INTERRUPT_PRO_TIMER_INT2_MAP_REG	TIMER_INT2 中断配置寄存器	0x00F4	读/写
INTERRUPT_PRO_TG_T0_EDGE_INT_MAP_REG	TG_T0_EDGE_INT 中断配置寄存器	0x00F8	读/写
INTERRUPT_PRO_TG_T1_EDGE_INT_MAP_REG	TG_T1_EDGE_INT 中断配置寄存器	0x00FC	读/写
INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP_REG	TG_WDT_EDGE_INT 中断配置寄存器	0x0100	读/写
INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP_REG	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
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

名称	描述	地址	访问
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

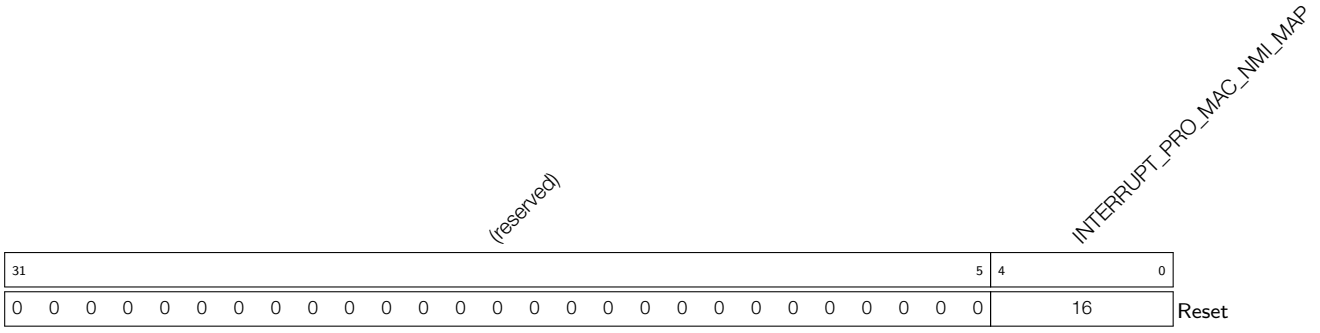
请注意，下表中的地址都是相对于中断矩阵基地址的地址偏移量（相对地址）。更多有关中断矩阵基地址的信息，请前往第 4.4 节。

Register 4.1: INTERRUPT_PRO_MAC_INTR_MAP_REG (0x0000)



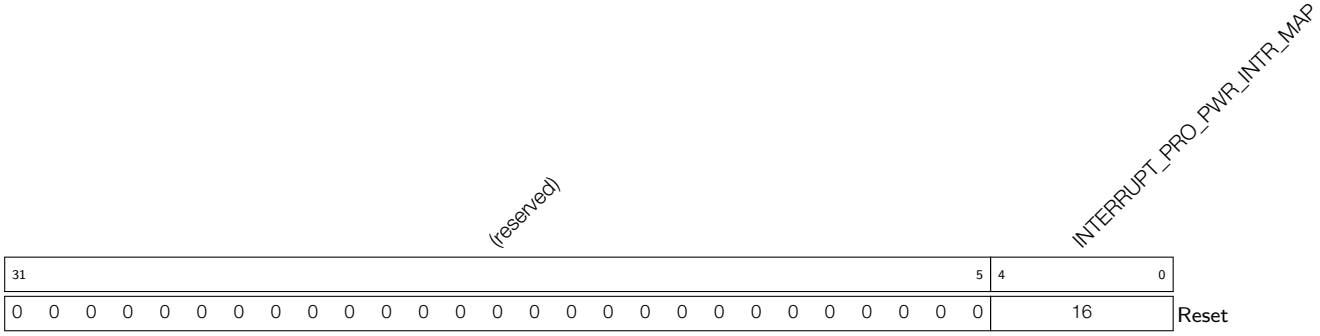
INTERRUPT_PRO_MAC_INTR_MAP 用于将 MAC_INTR 中断信号映射至 CPU 中断。（读/写）

Register 4.2: INTERRUPT_PRO_MAC_NMI_MAP_REG (0x0004)



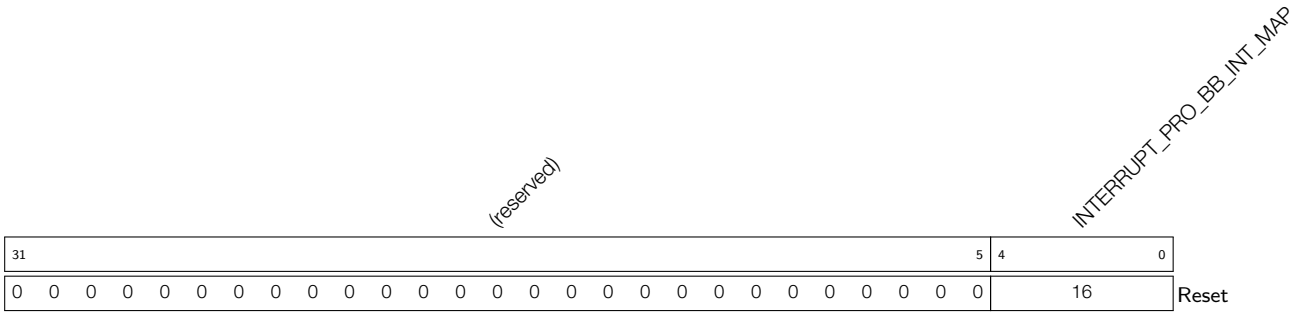
INTERRUPT_PRO_MAC_NMI_MAP 用于将 MAC_NMI 中断信号映射至 CPU 中断。（读/写）

Register 4.3: INTERRUPT_PRO_PWR_INTR_MAP_REG (0x0008)



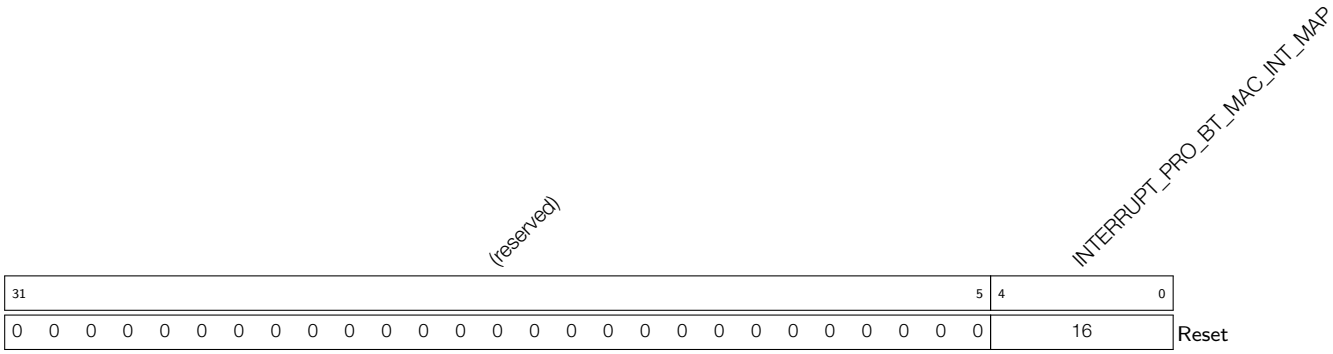
INTERRUPT_PRO_PWR_INTR_MAP 用于将 PWR_INTR 中断信号映射至 CPU 中断。（读/写）

Register 4.4: INTERRUPT_PRO_BB_INT_MAP_REG (0x000C)



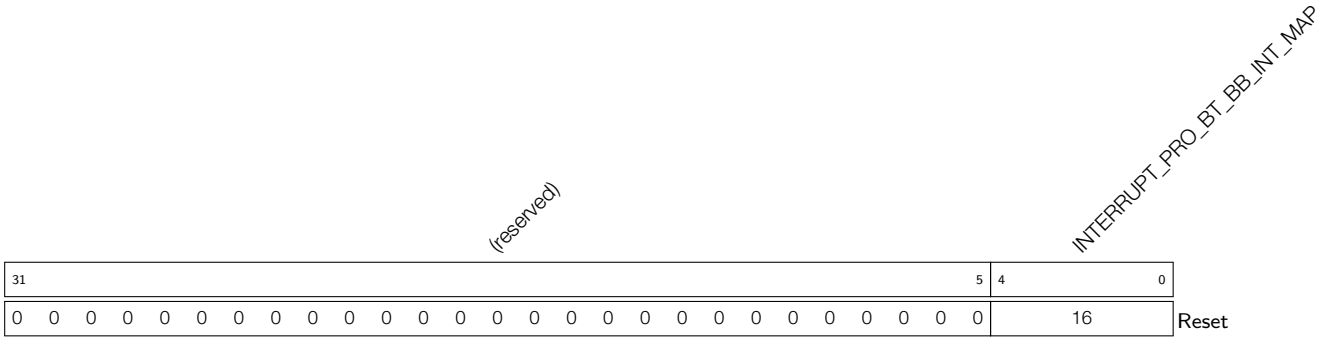
INTERRUPT_PRO_BB_INT_MAP 用于将 BB_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.5: INTERRUPT_PRO_BT_MAC_INT_MAP_REG (0x0010)



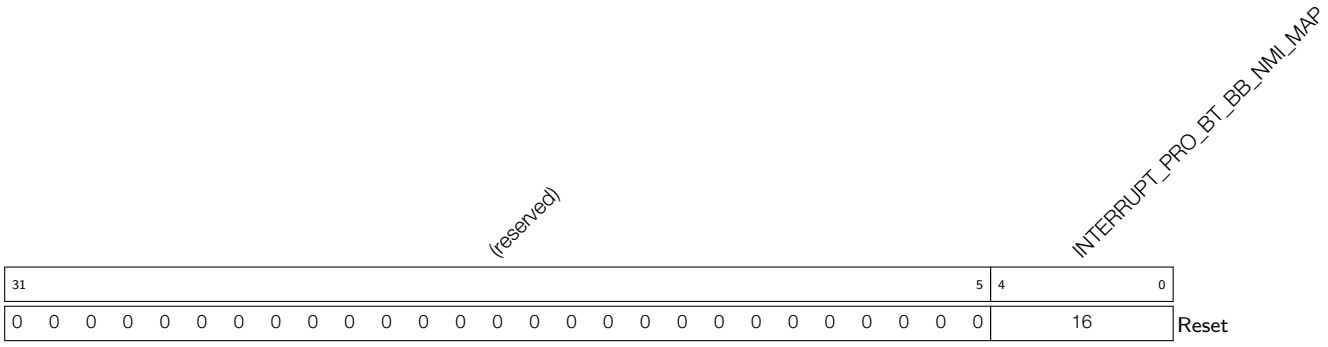
INTERRUPT_PRO_BT_MAC_INT_MAP 用于将 BT_MAC_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.6: INTERRUPT_PRO_BT_BB_INT_MAP_REG (0x0014)



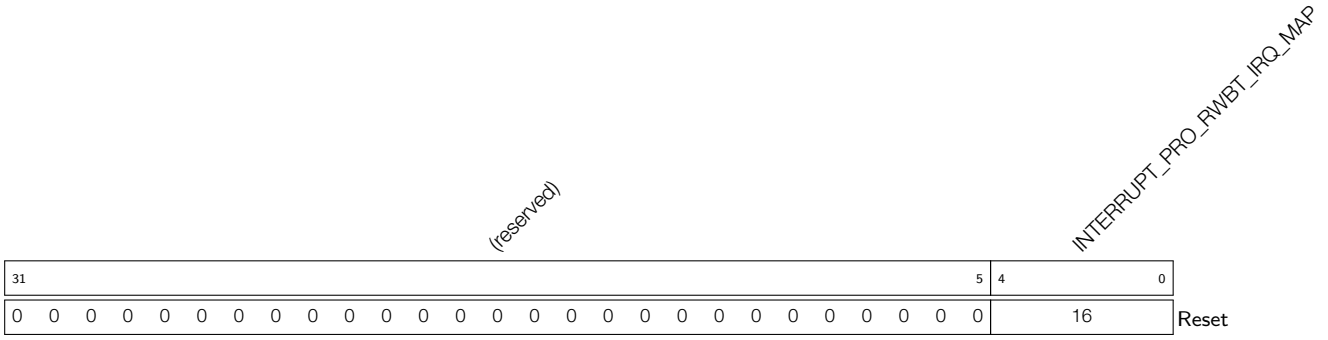
INTERRUPT_PRO_BT_BB_INT_MAP 用于将 BT_BB_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.7: INTERRUPT_PRO_BT_BB_NMI_MAP_REG (0x0018)



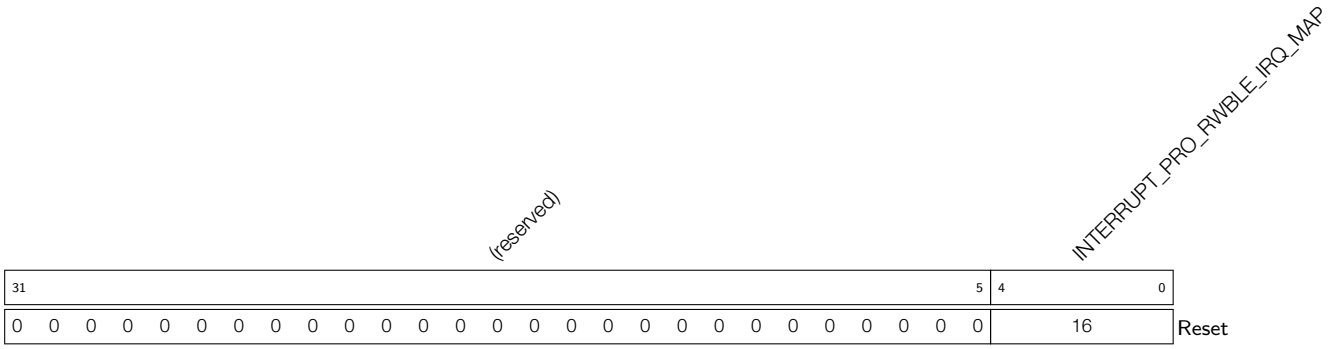
INTERRUPT_PRO_BT_BB_NMI_MAP 用于将 BT_BB_NMI 中断信号映射至 CPU 中断。(读/写)

Register 4.8: INTERRUPT_PRO_RWBT_IRQ_MAP_REG (0x001C)



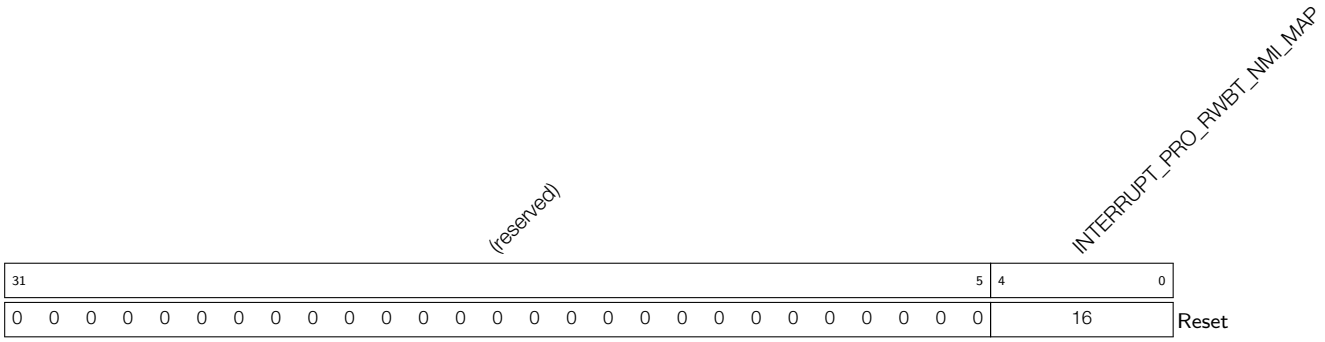
INTERRUPT_PRO_RWBT_IRQ_MAP 用于将 RWBT_IRQ 中断信号映射至 CPU 中断。(读/写)

Register 4.9: INTERRUPT_PRO_RWBLE_IRQ_MAP_REG (0x0020)



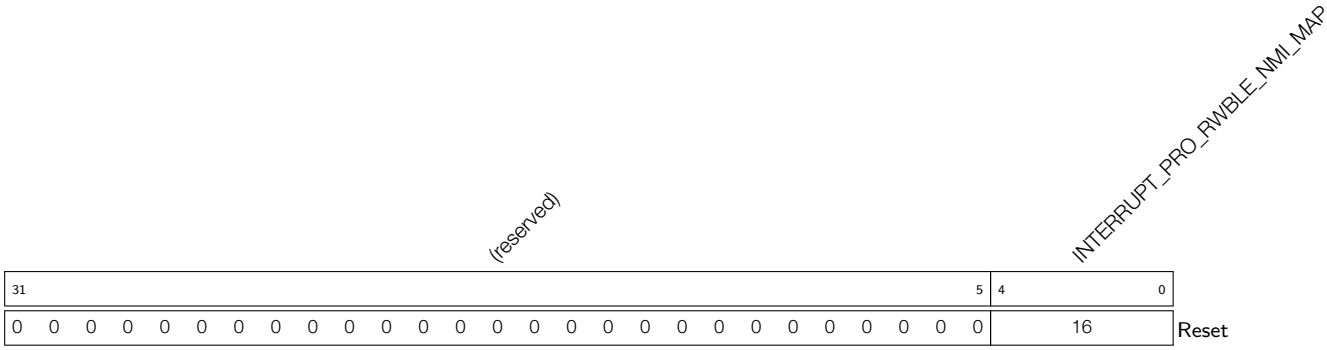
INTERRUPT_PRO_RWBLE_IRQ_MAP 用于将 RWBLE_IRQ 中断信号映射至 CPU 中断。(读/写)

Register 4.10: INTERRUPT_PRO_RWBT_NMI_MAP_REG (0x0024)



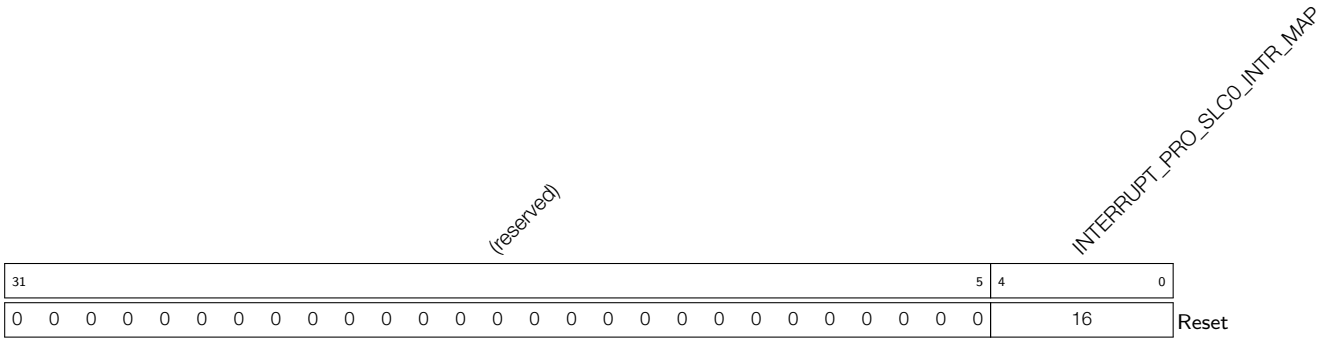
INTERRUPT_PRO_RWBT_NMI_MAP 用于将 RWBT_NMI 中断信号映射至 CPU 中断。(读/写)

Register 4.11: INTERRUPT_PRO_RWBLE_NMI_MAP_REG (0x0028)



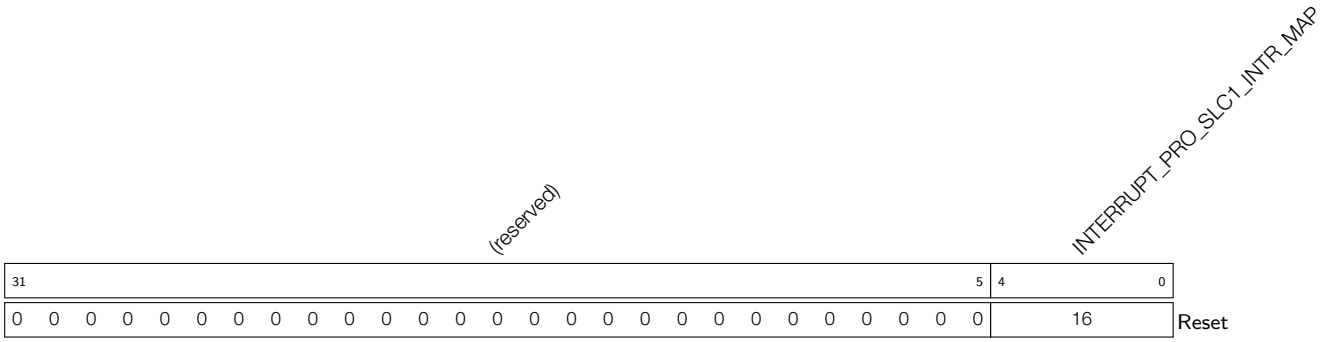
INTERRUPT_PRO_RWBLE_NMI_MAP 用于将 RWBLE_NMI 中断信号映射至 CPU 中断。(读/写)

Register 4.12: INTERRUPT_PRO_SLC0_INTR_MAP_REG (0x002C)



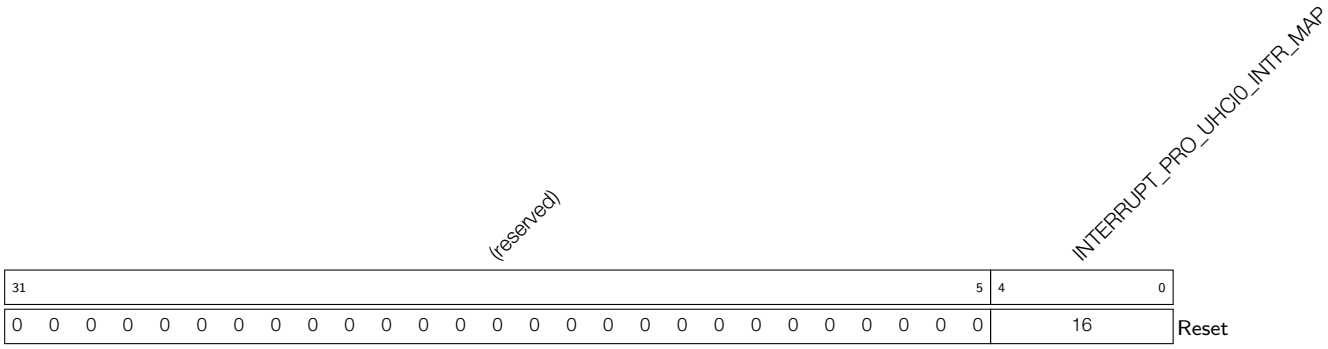
INTERRUPT_PRO_SLC0_INTR_MAP 用于将 SLC0_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.13: INTERRUPT_PRO_SLC1_INTR_MAP_REG (0x0030)



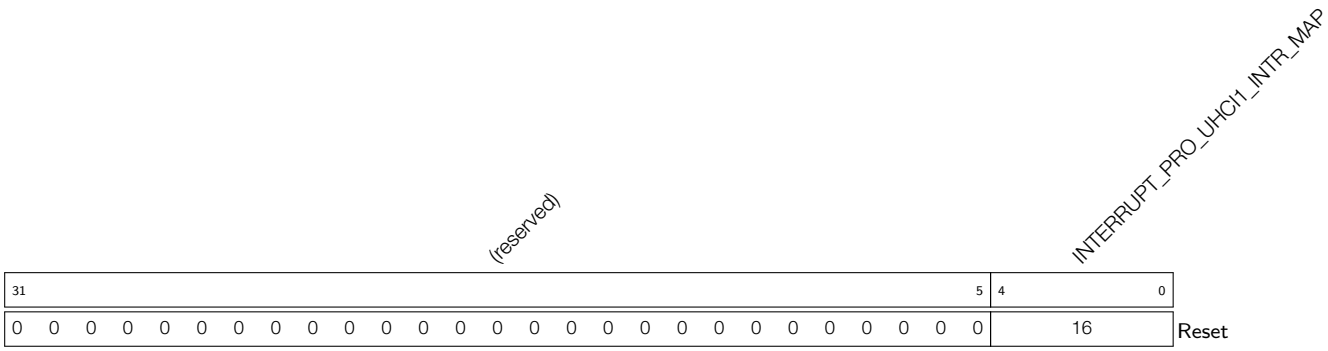
INTERRUPT_PRO_SLC1_INTR_MAP 用于将 SLC1_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.14: INTERRUPT_PRO_UHCI0_INTR_MAP_REG (0x0034)



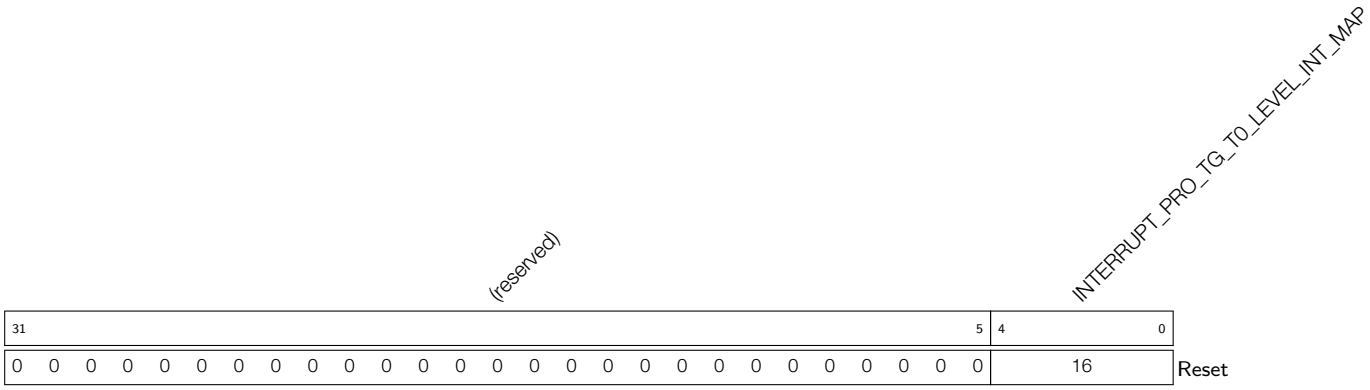
INTERRUPT_PRO_UHCI0_INTR_MAP 用于将 UHCI0_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.15: INTERRUPT_PRO_UHCI1_INTR_MAP_REG (0x0038)



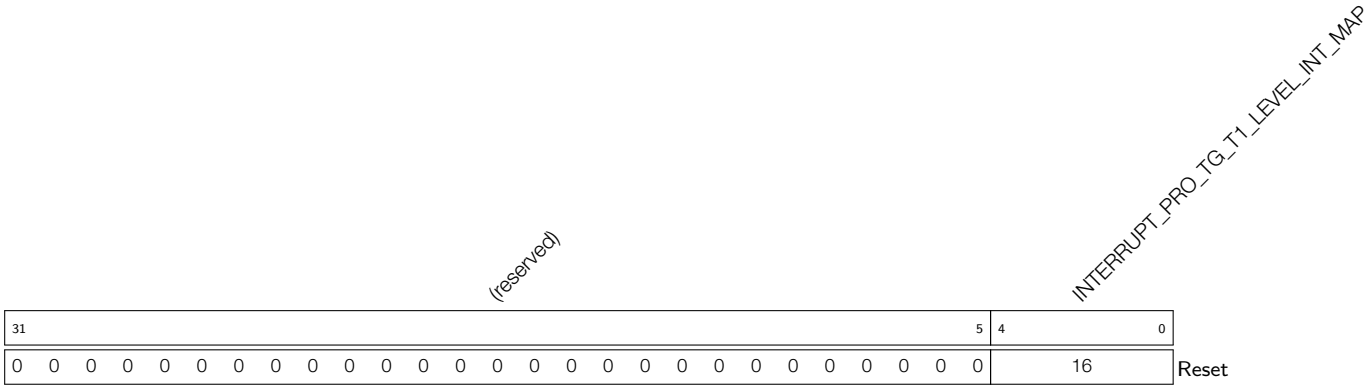
INTERRUPT_PRO_UHCI1_INTR_MAP 用于将 UHCI1_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.16: INTERRUPT_PRO_TG_T0_LEVEL_INT_MAP_REG (0x003C)



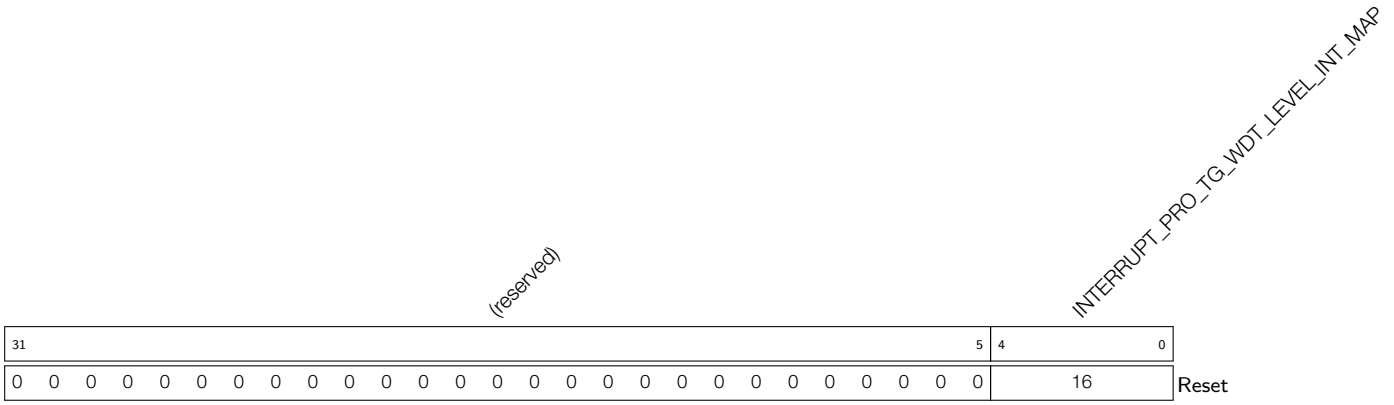
INTERRUPT_PRO_TG_T0_LEVEL_INT_MAP 用于将 TG_T0_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.17: INTERRUPT_PRO_TG_T1_LEVEL_INT_MAP_REG (0x0040)



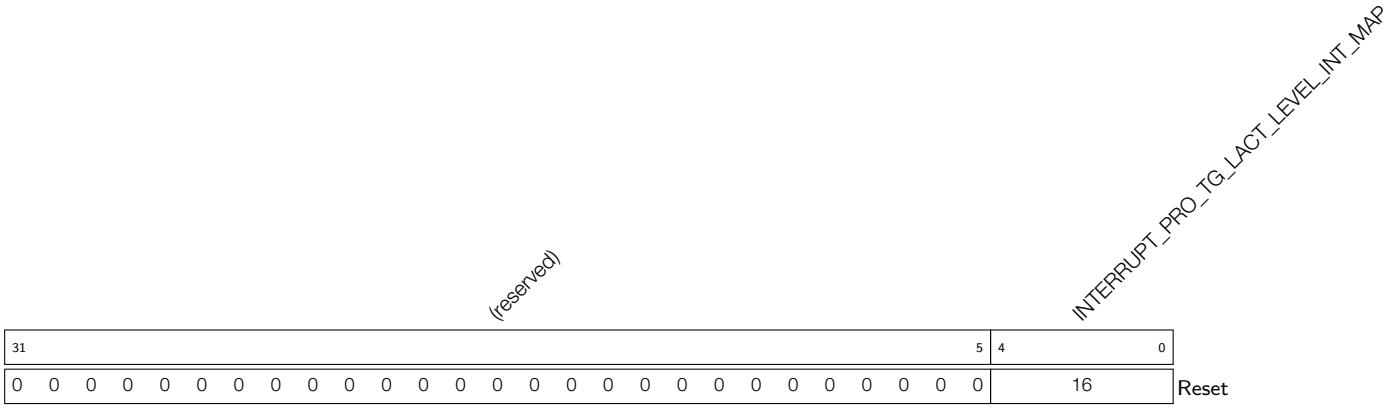
INTERRUPT_PRO_TG_T1_LEVEL_INT_MAP 用于将 TG_T1_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.18: INTERRUPT_PRO_TG_WDT_LEVEL_INT_MAP_REG (0x0044)



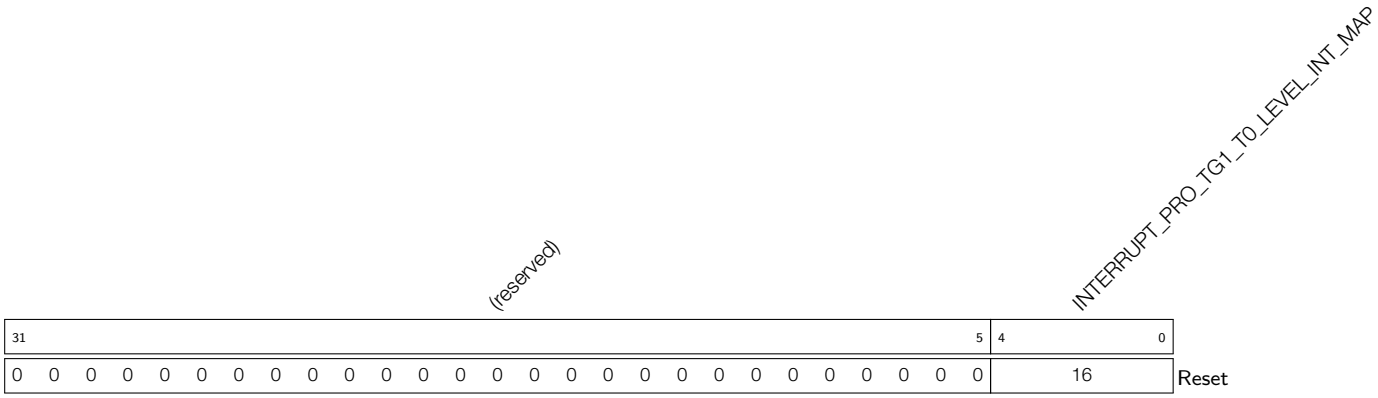
INTERRUPT_PRO_TG_WDT_LEVEL_INT_MAP 用于将 TG_WDT_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.19: INTERRUPT_PRO_TG_LACT_LEVEL_INT_MAP_REG (0x0048)



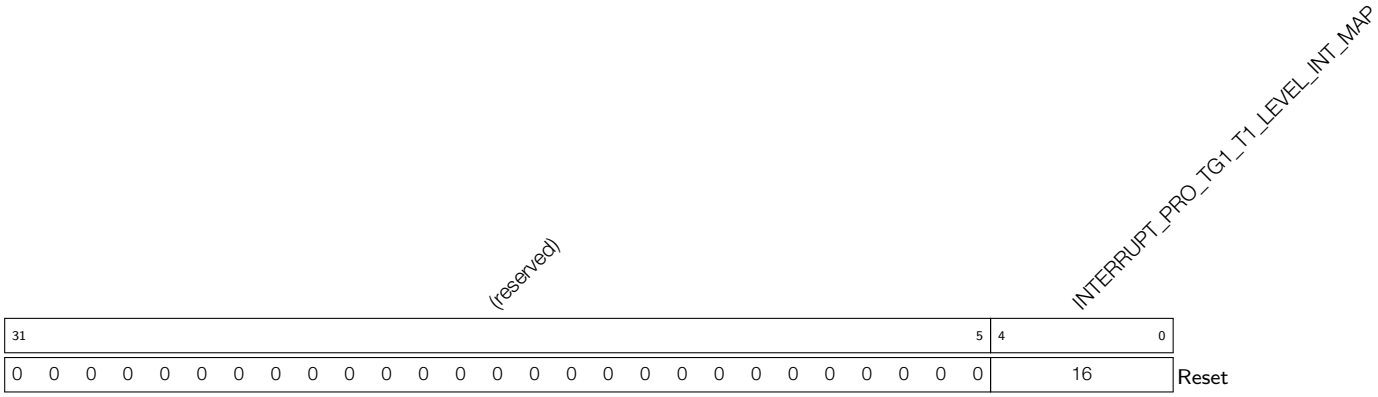
INTERRUPT_PRO_TG_LACT_LEVEL_INT_MAP 用于将 TG_LACT_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.20: INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP_REG (0x004C)



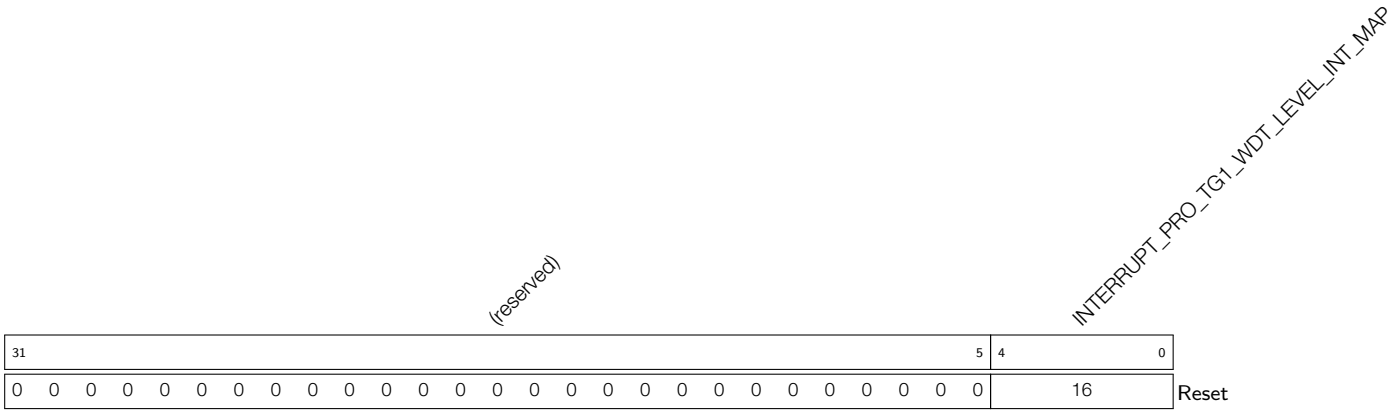
INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP 用于将 TG1_T0_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.21: INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP_REG (0x0050)



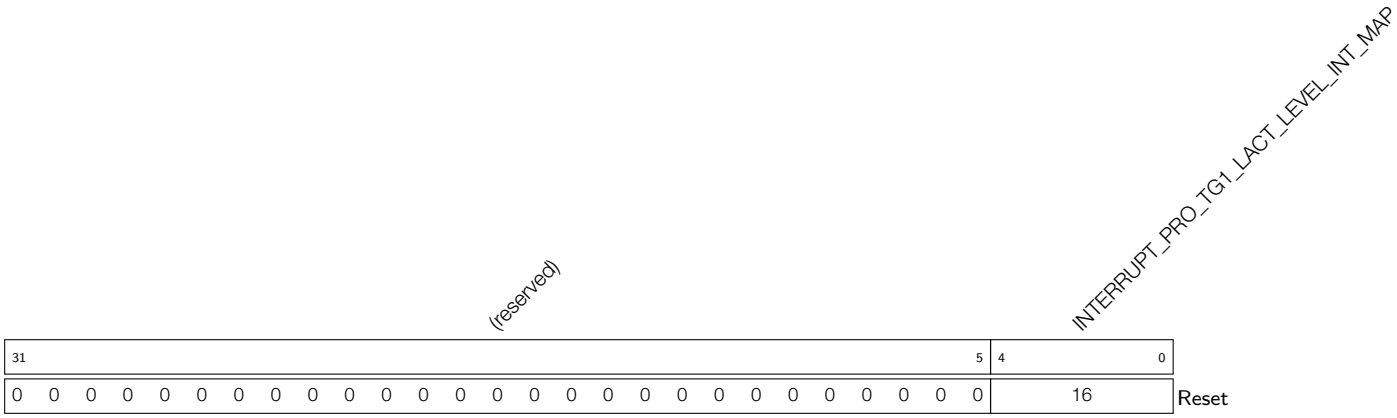
INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP 用于将 TG1_T1_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.22: INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP_REG (0x0054)



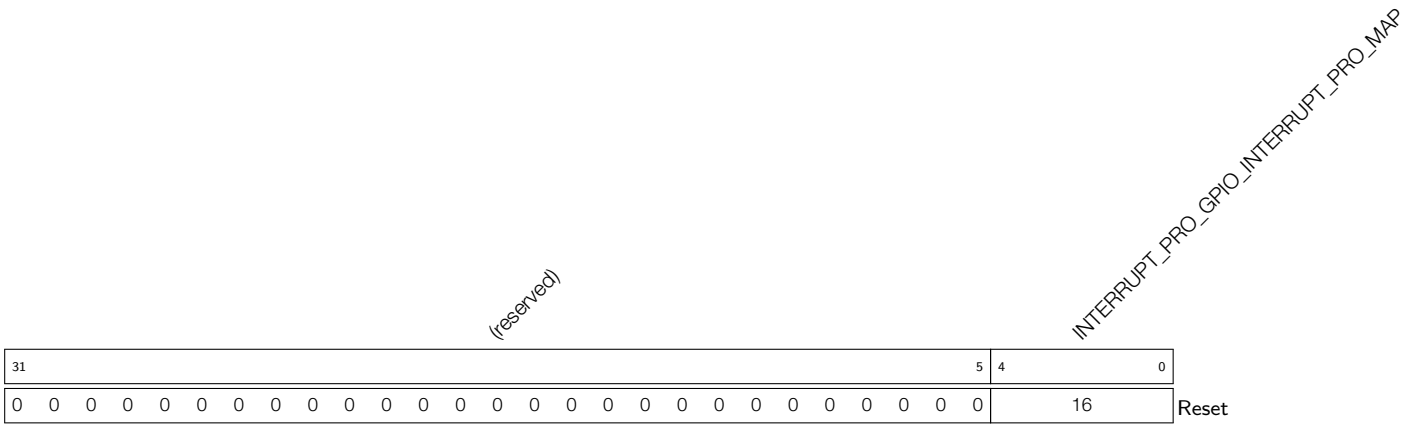
INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP 用于将 TG1_WDT_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.23: INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP_REG (0x0058)



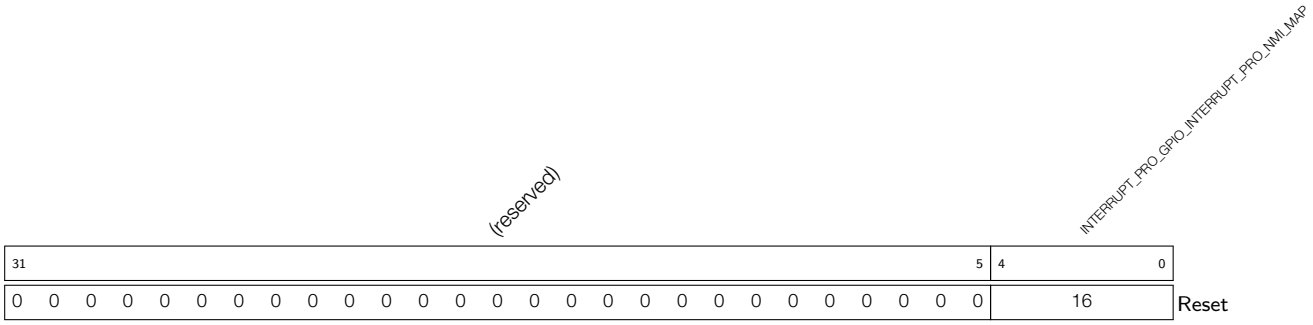
INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP 用于将 TG1_LACT_LEVEL_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.24: INTERRUPT_PRO_GPIO_INTERRUPT_PRO_MAP_REG (0x005C)



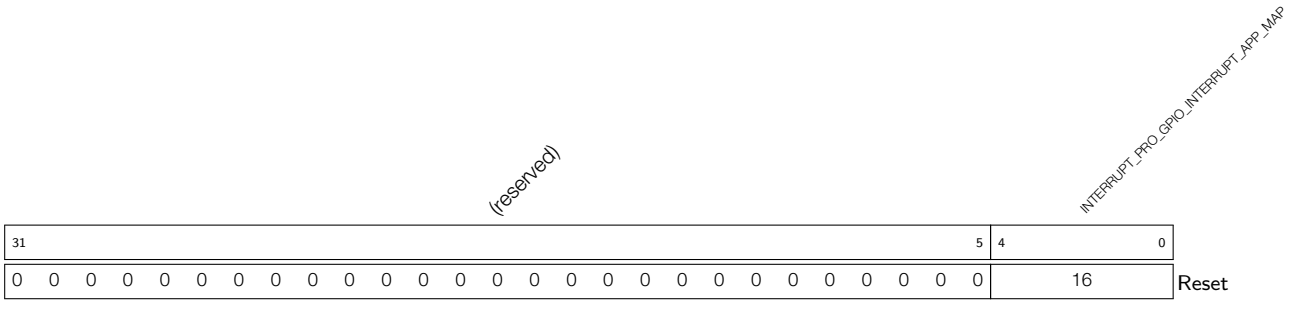
INTERRUPT_PRO_GPIO_INTERRUPT_PRO_MAP 用于将 GPIO_INTERRUPT_PRO 中断信号映射至 CPU 中断。(读/写)

Register 4.25: INTERRUPT_PRO_GPIO_INTERRUPT_PRO_NMI_MAP_REG (0x0060)



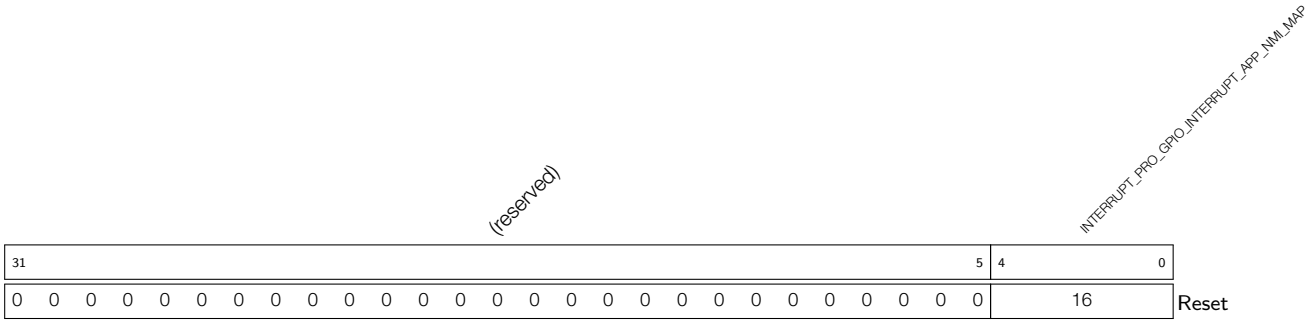
INTERRUPT_PRO_GPIO_INTERRUPT_PRO_NMI_MAP 用于将 GPIO_INTERRUPT_PRO_NMI 中断信号映射至 CPU 中断。(读/写)

Register 4.26: INTERRUPT_PRO_GPIO_INTERRUPT_APP_MAP_REG (0x0064)



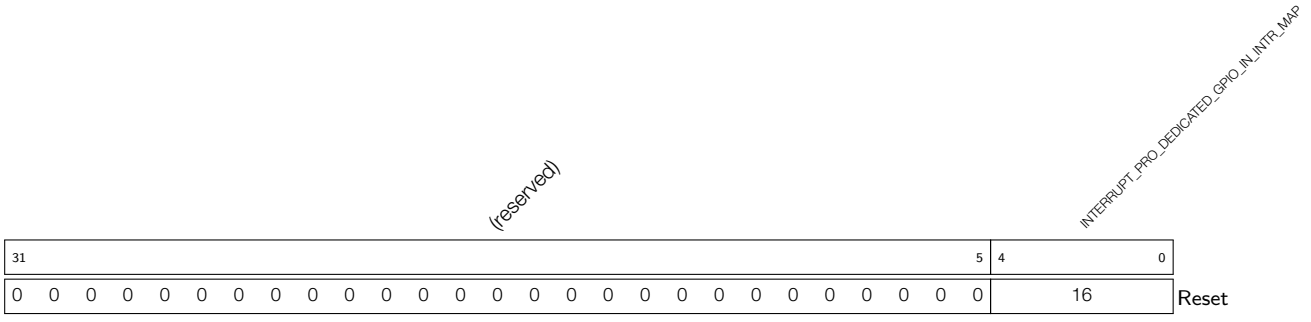
INTERRUPT_PRO_GPIO_INTERRUPT_APP_MAP 用于将 GPIO_INTERRUPT_APP 中断信号映射至 CPU 中断。(读/写)

Register 4.27: INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP_REG (0x0068)



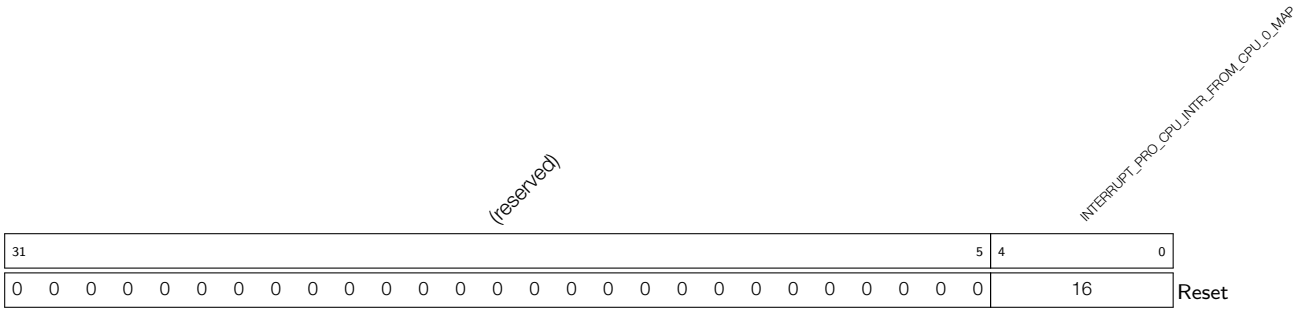
INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP 用于将 GPIO_INTERRUPT_APP 中断信号映射至 CPU 中断。(读/写)

Register 4.28: INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP_REG (0x006C)



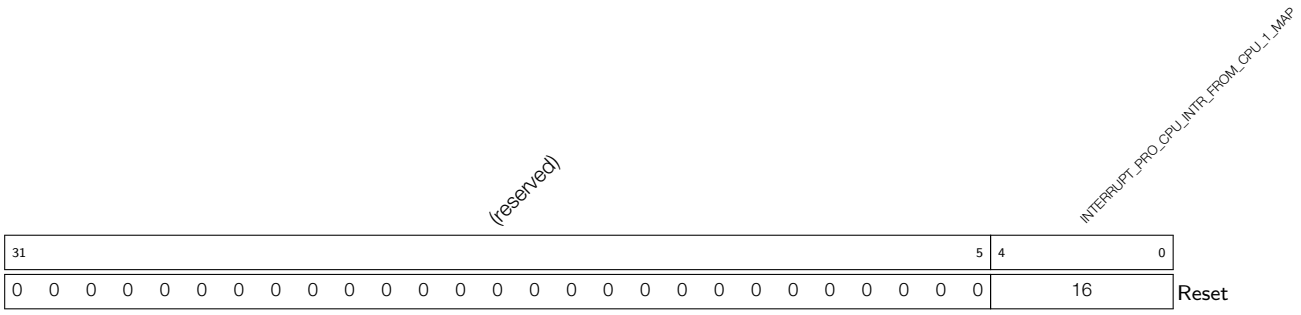
INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP 用于将 DEDICATED_GPIO_IN_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.29: INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP_REG (0x0070)



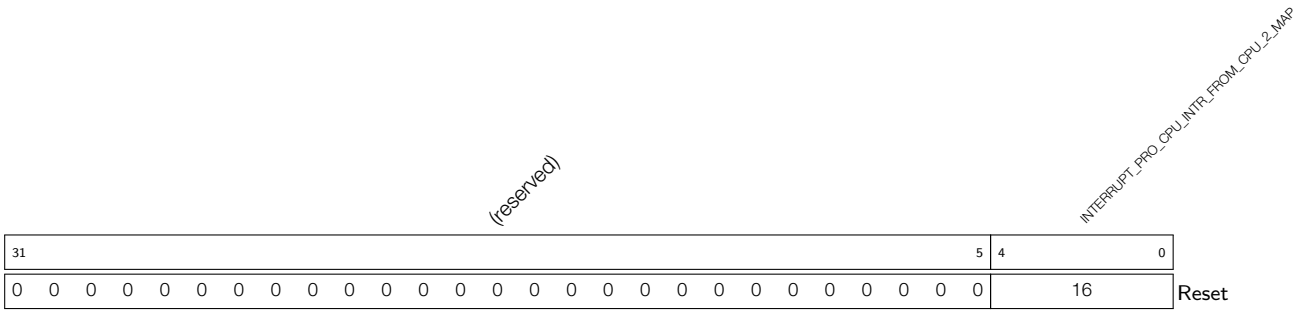
INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP 用于将 CPU_INTR_FROM_CPU_0 中断信号映射至 CPU 中断。(读/写)

Register 4.30: INTERRUPT_PRO_CPU_INTR_FROM_CPU_1_MAP_REG (0x0074)



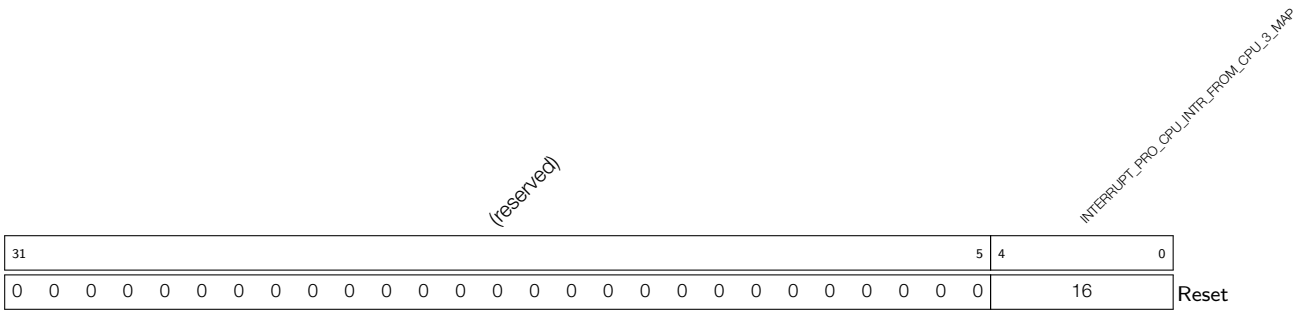
INTERRUPT_PRO_CPU_INTR_FROM_CPU_1_MAP 用于将 CPU_INTR_FROM_CPU_1 中断信号映射至 CPU 中断。(读/写)

Register 4.31: INTERRUPT_PRO_CPU_INTR_FROM_CPU_2_MAP_REG (0x0078)



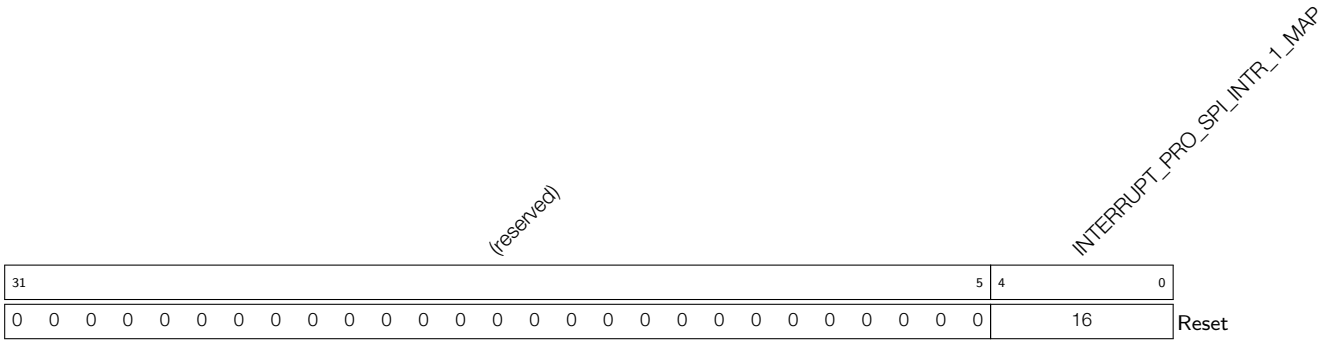
INTERRUPT_PRO_CPU_INTR_FROM_CPU_2_MAP 用于将 CPU_INTR_FROM_CPU_2 中断信号映射至 CPU 中断。(读/写)

Register 4.32: INTERRUPT_PRO_CPU_INTR_FROM_CPU_3_MAP_REG (0x007C)



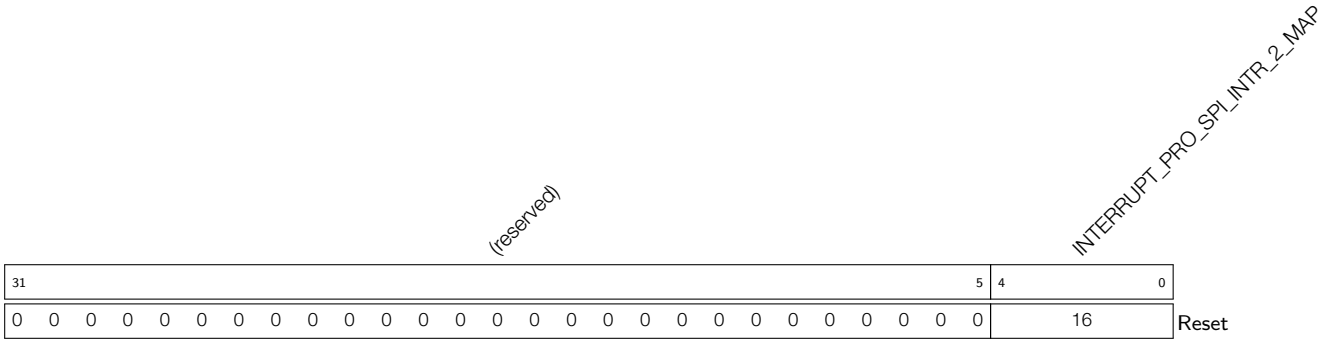
INTERRUPT_PRO_CPU_INTR_FROM_CPU_3_MAP 用于将 CPU_INTR_FROM_CPU_3 中断信号映射至 CPU 中断。(读/写)

Register 4.33: INTERRUPT_PRO_SPI_INTR_1_MAP_REG (0x0080)



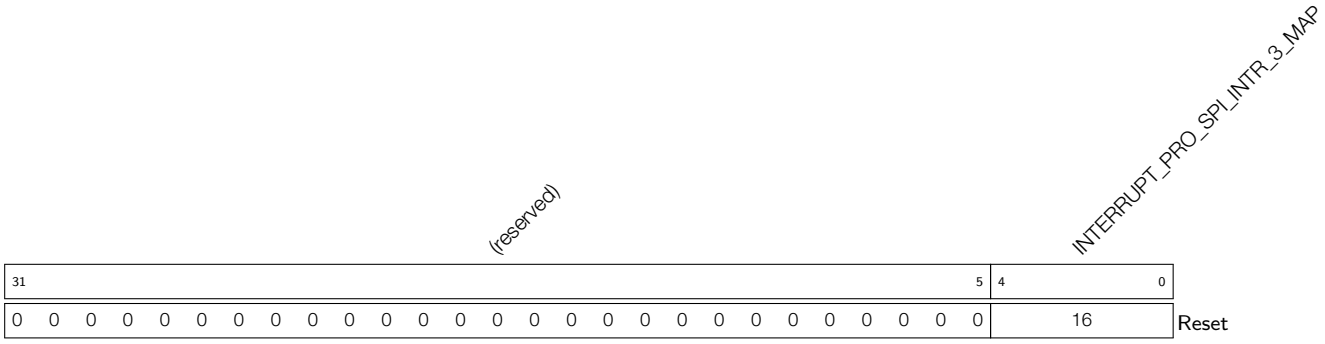
INTERRUPT_PRO_SPI_INTR_1_MAP 用于将 SPI_INTR_1 中断信号映射至 CPU 中断。(读/写)

Register 4.34: INTERRUPT_PRO_SPI_INTR_2_MAP_REG (0x0084)



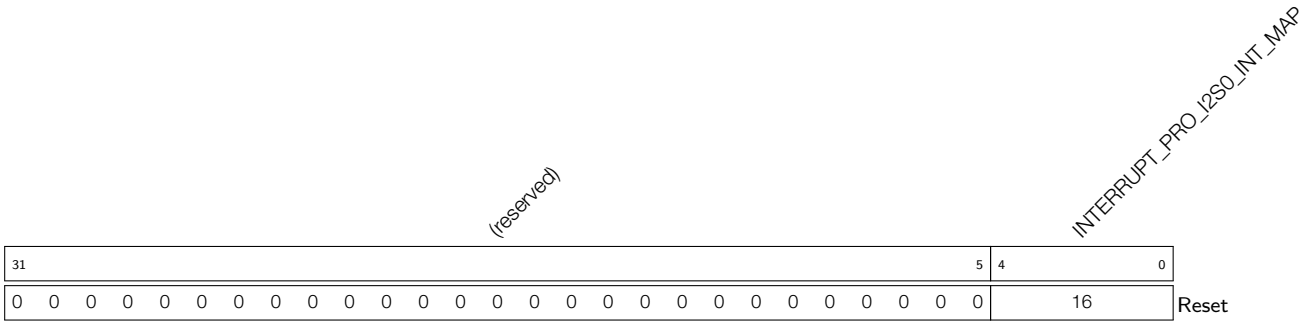
INTERRUPT_PRO_SPI_INTR_2_MAP 用于将 SPI_INTR_2 中断信号映射至 CPU 中断。(读/写)

Register 4.35: INTERRUPT_PRO_SPI_INTR_3_MAP_REG (0x0088)



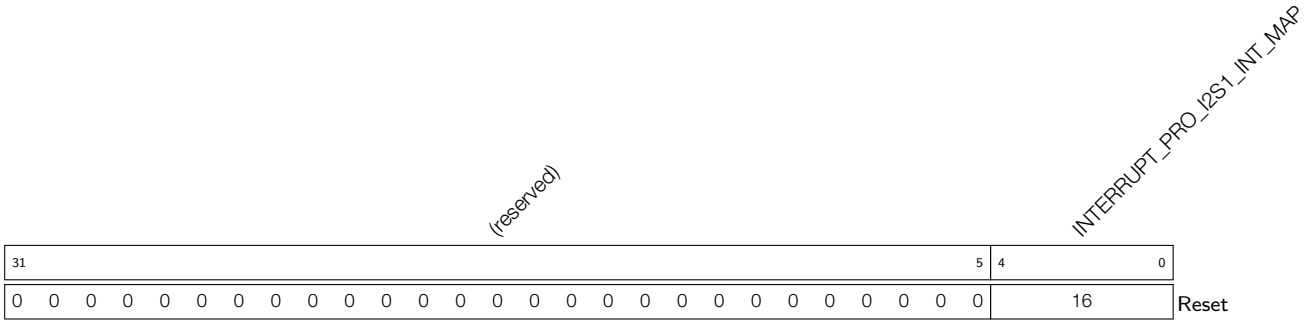
INTERRUPT_PRO_SPI_INTR_3_MAP 用于将 SPI_INTR_3 中断信号映射至 CPU 中断。(读/写)

Register 4.36: INTERRUPT_PRO_I2S0_INT_MAP_REG (0x008C)



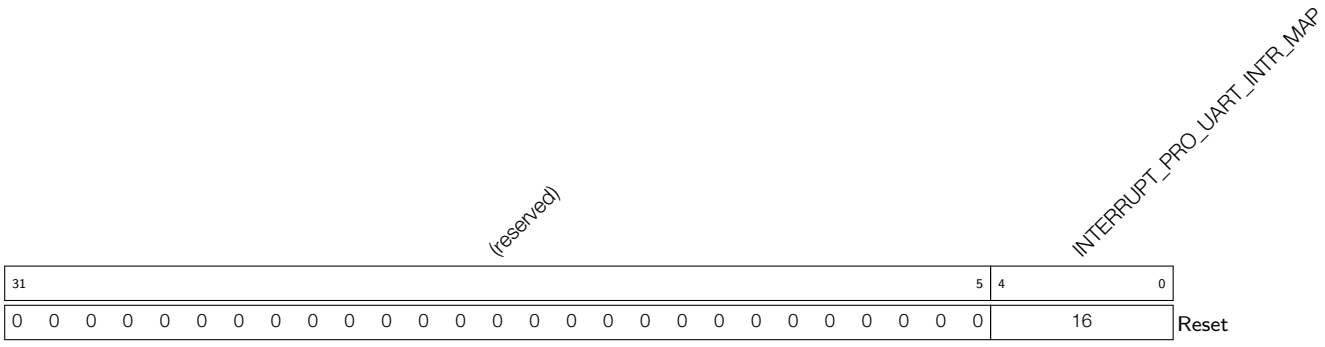
INTERRUPT_PRO_I2S0_INT_MAP 用于将 I2S0_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.37: INTERRUPT_PRO_I2S1_INT_MAP_REG (0x0090)



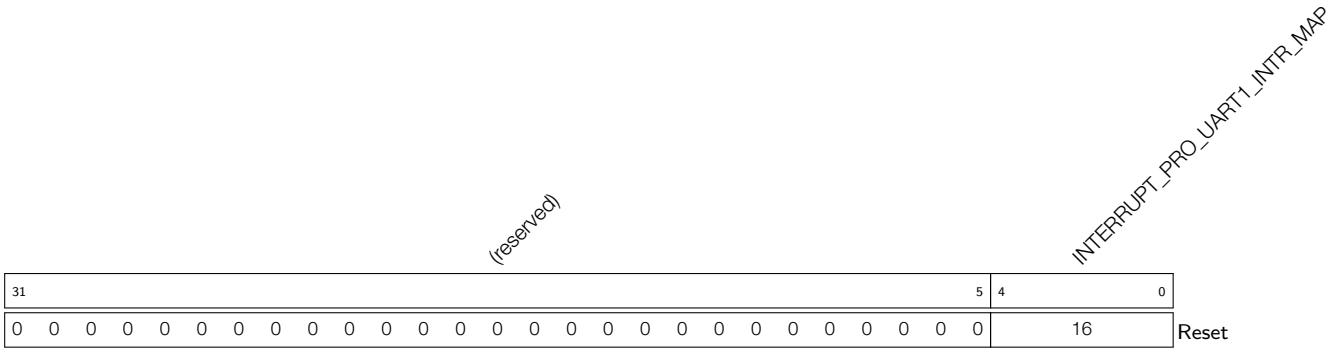
INTERRUPT_PRO_I2S1_INT_MAP 用于将 I2S1_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.38: INTERRUPT_PRO_UART_INTR_MAP_REG (0x0094)



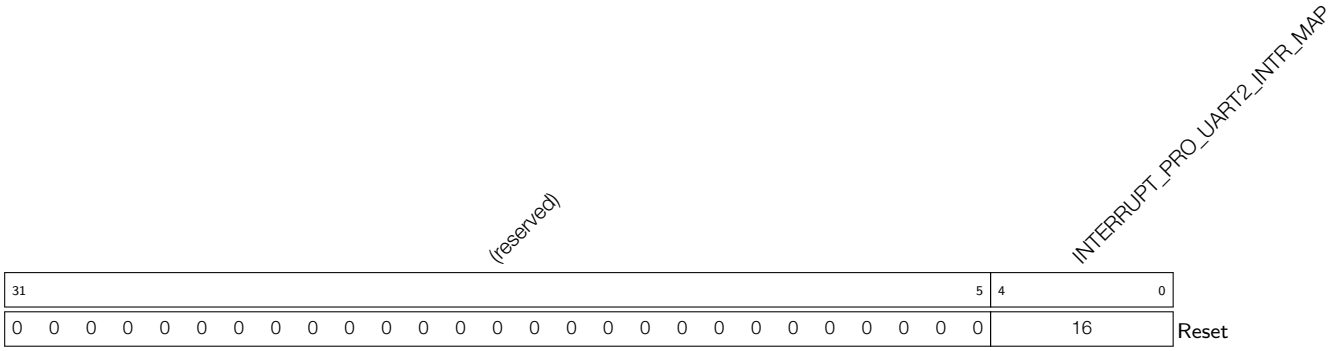
INTERRUPT_PRO_UART_INTR_MAP 用于将 UART_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.39: INTERRUPT_PRO_UART1_INTR_MAP_REG (0x0098)



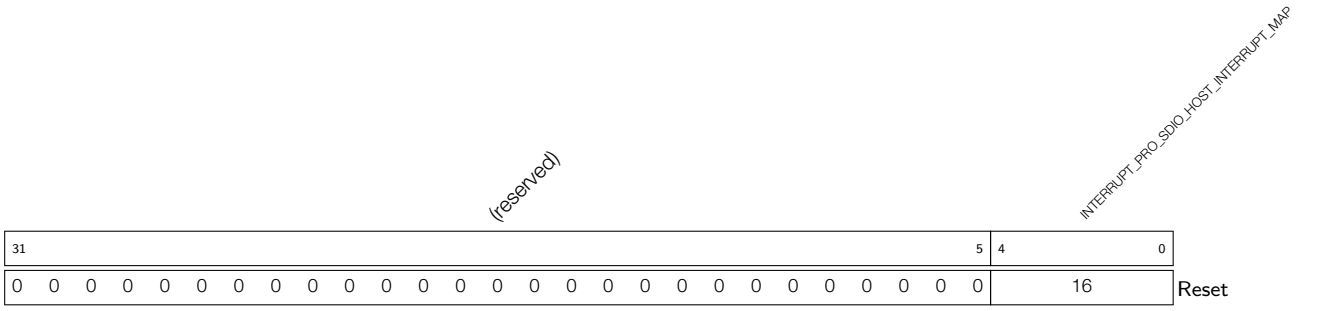
INTERRUPT_PRO_UART1_INTR_MAP 用于将 UART1_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.40: INTERRUPT_PRO_UART2_INTR_MAP_REG (0x009C)



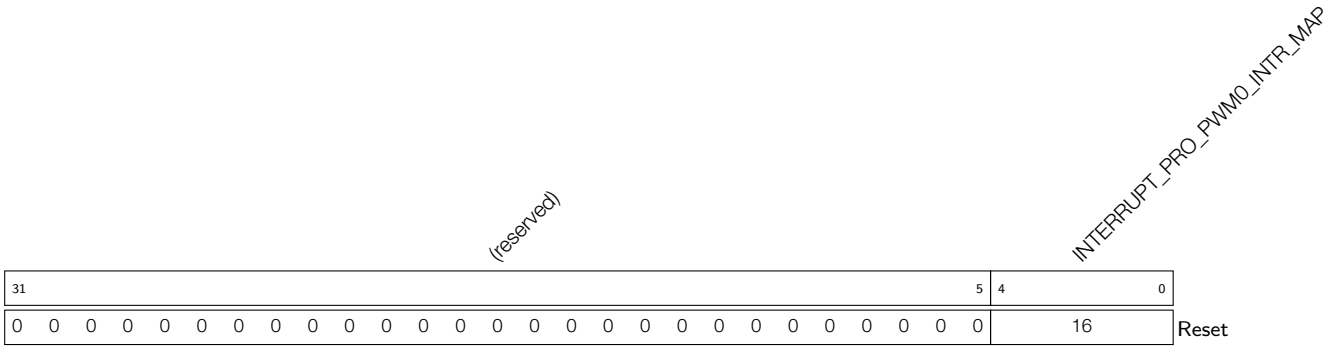
INTERRUPT_PRO_UART2_INTR_MAP 用于将 UART2_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.41: INTERRUPT_PRO_SDIO_HOST_INTERRUPT_MAP_REG (0x00A0)



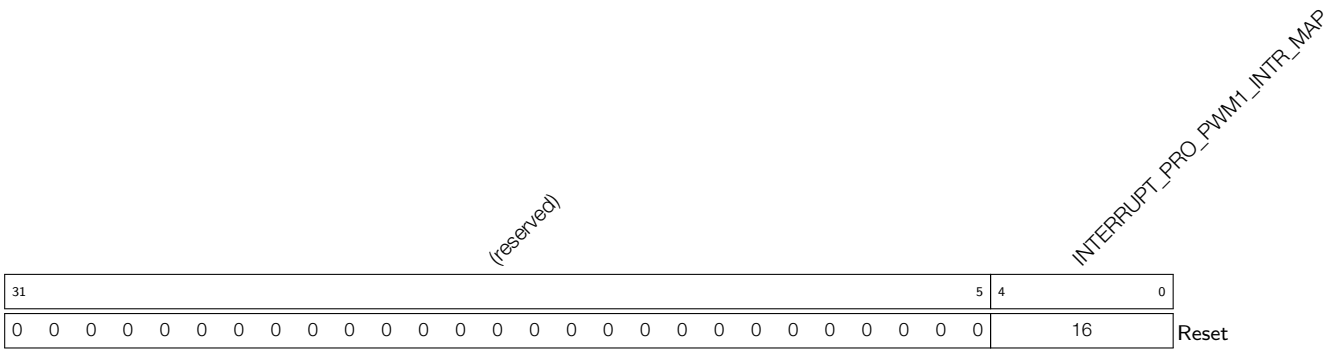
INTERRUPT_PRO_SDIO_HOST_INTERRUPT_MAP 用于将 SDIO_HOST 中断信号映射至 CPU 中断。(读/写)

Register 4.42: INTERRUPT_PRO_PWM0_INTR_MAP_REG (0x00A4)



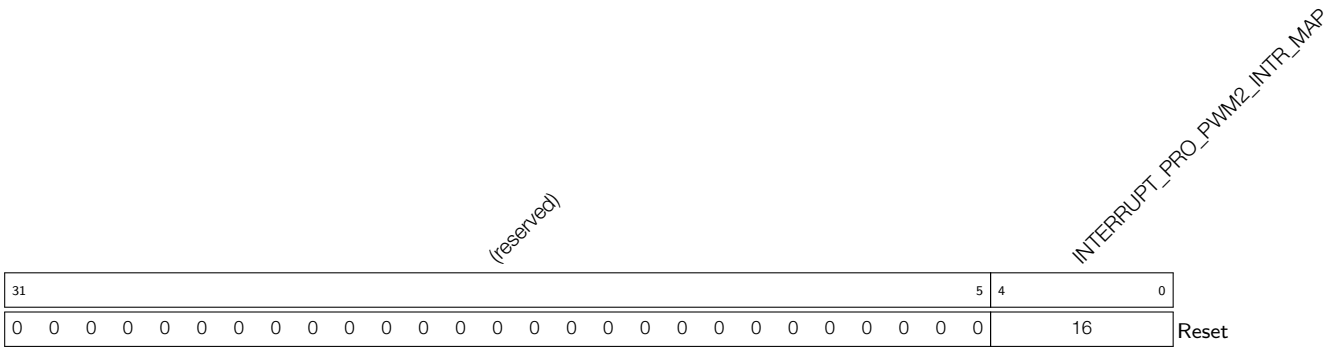
INTERRUPT_PRO_PWM0_INTR_MAP 用于将 PWM0_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.43: INTERRUPT_PRO_PWM1_INTR_MAP_REG (0x00A8)



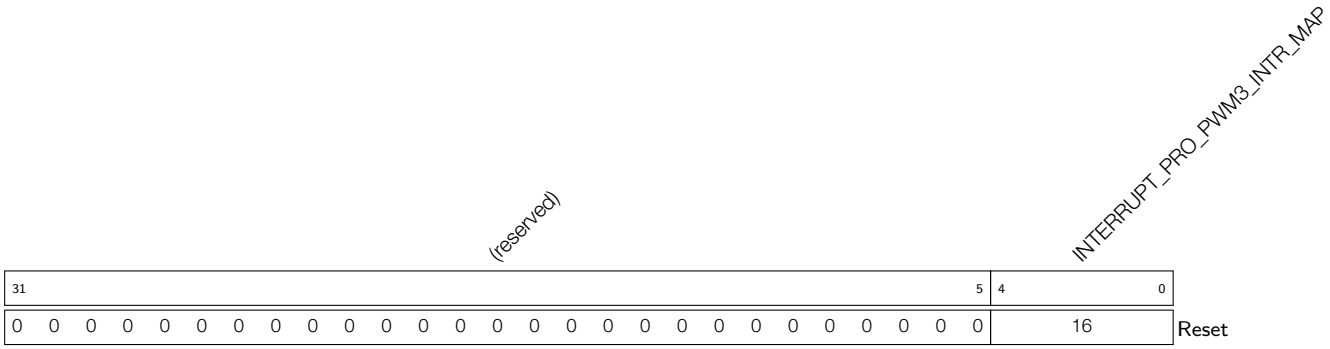
INTERRUPT_PRO_PWM1_INTR_MAP 用于将 PWM1_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.44: INTERRUPT_PRO_PWM2_INTR_MAP_REG (0x00AC)



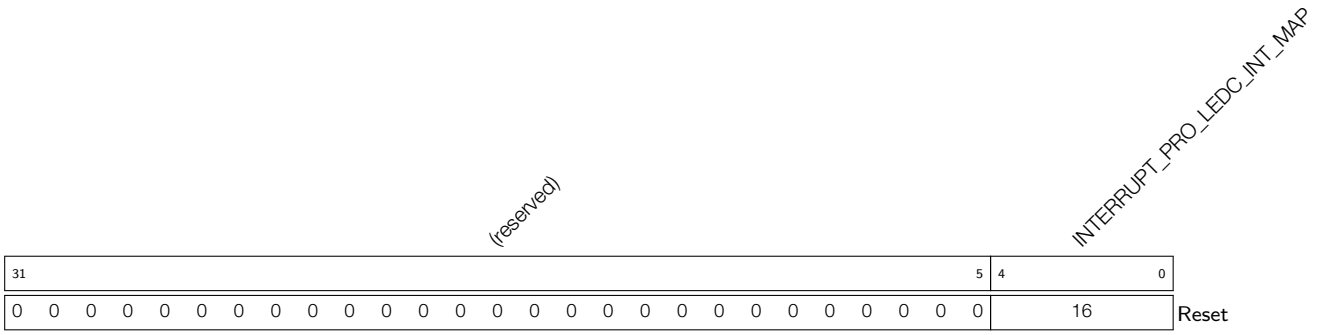
INTERRUPT_PRO_PWM2_INTR_MAP 用于将 PWM2_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.45: INTERRUPT_PRO_PWM3_INTR_MAP_REG (0x00B0)



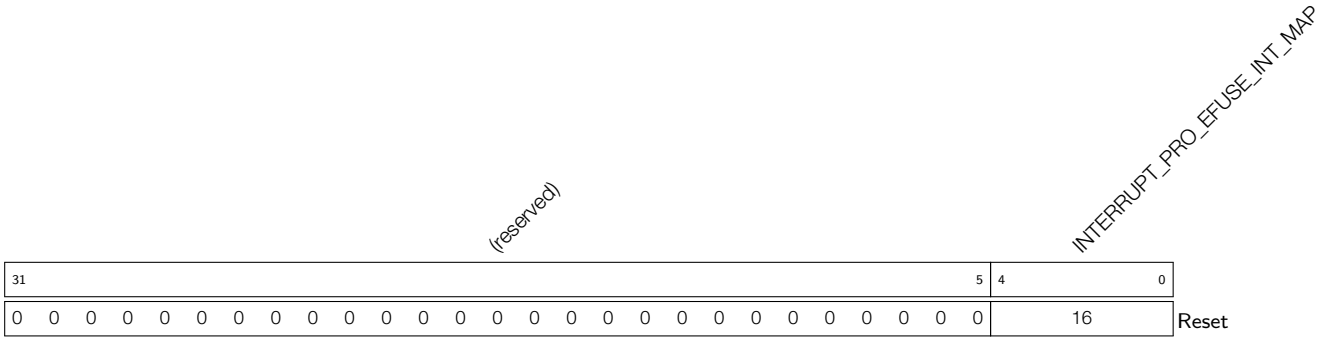
INTERRUPT_PRO_PWM3_INTR_MAP 用于将 PWM3_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.46: INTERRUPT_PRO_LEDC_INT_MAP_REG (0x00B4)



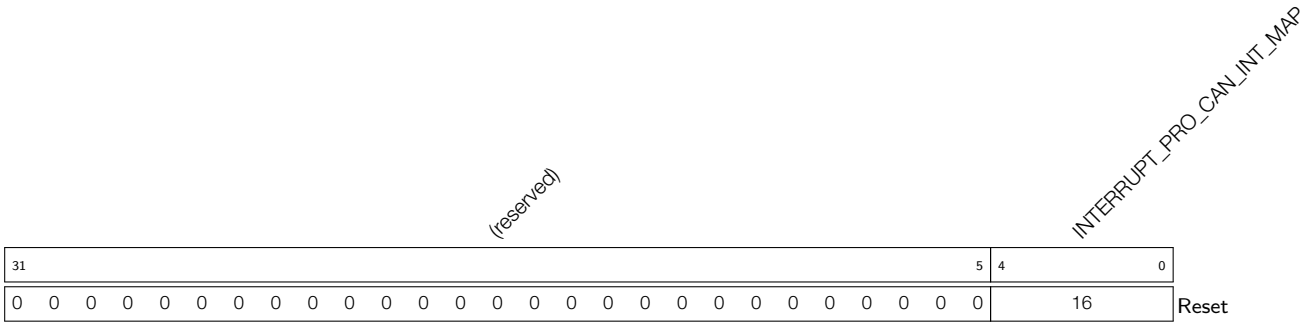
INTERRUPT_PRO_LEDC_INT_MAP 用于将 LEDC_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.47: INTERRUPT_PRO_EFUSE_INT_MAP_REG (0x00B8)



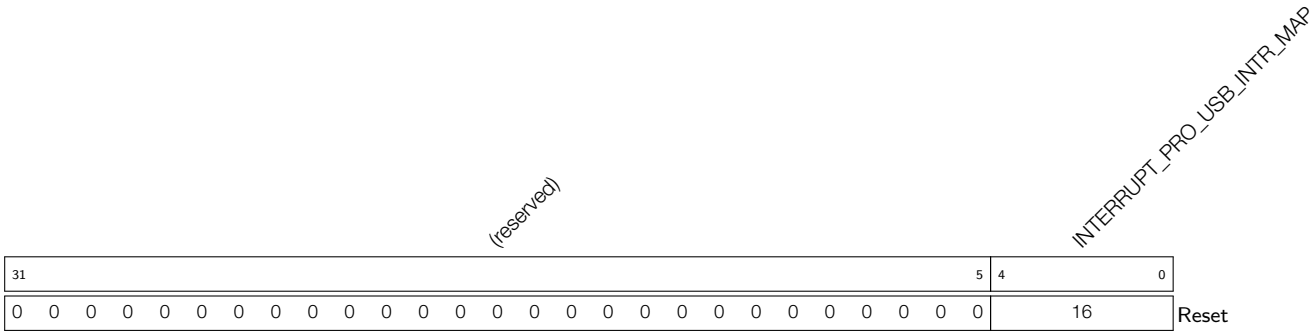
INTERRUPT_PRO_EFUSE_INT_MAP 用于将 EFUSE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.48: INTERRUPT_PRO_CAN_INT_MAP_REG (0x00BC)



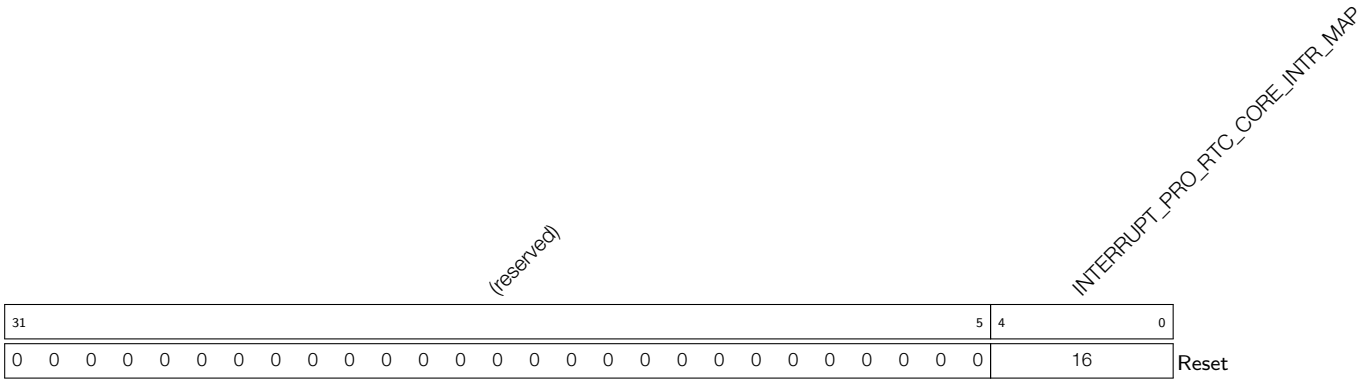
INTERRUPT_PRO_CAN_INT_MAP 用于将 CAN_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.49: INTERRUPT_PRO_USB_INTR_MAP_REG (0x00C0)



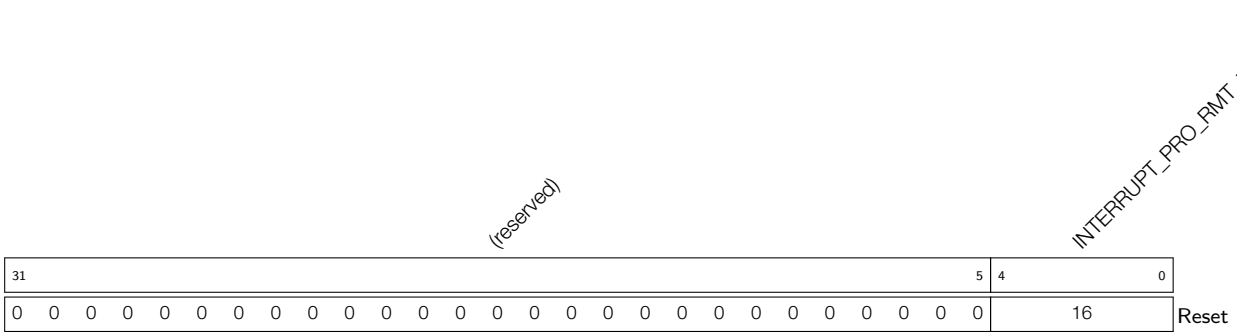
INTERRUPT_PRO_USB_INTR_MAP 用于将 USB_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.50: INTERRUPT_PRO_RTC_CORE_INTR_MAP_REG (0x00C4)



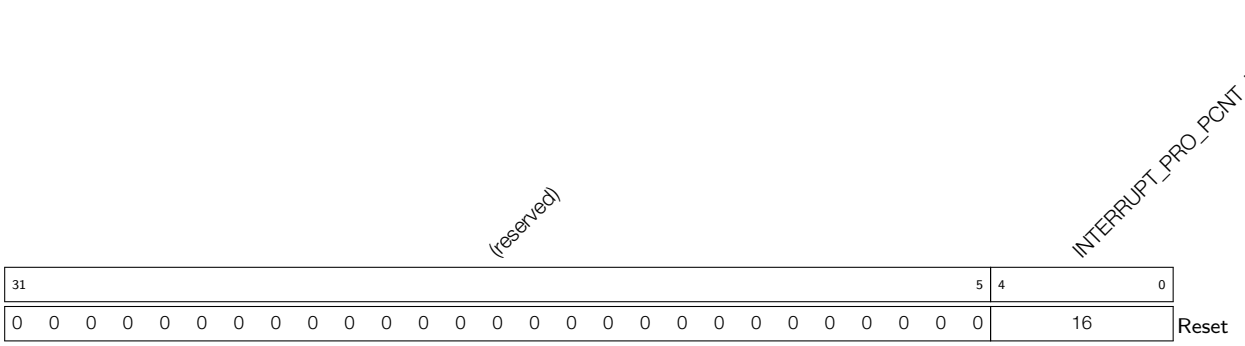
INTERRUPT_PRO_RTC_CORE_INTR_MAP 用于将 RTC_CORE_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.51: INTERRUPT_PRO_RMT_INTR_MAP_REG (0x00C8)



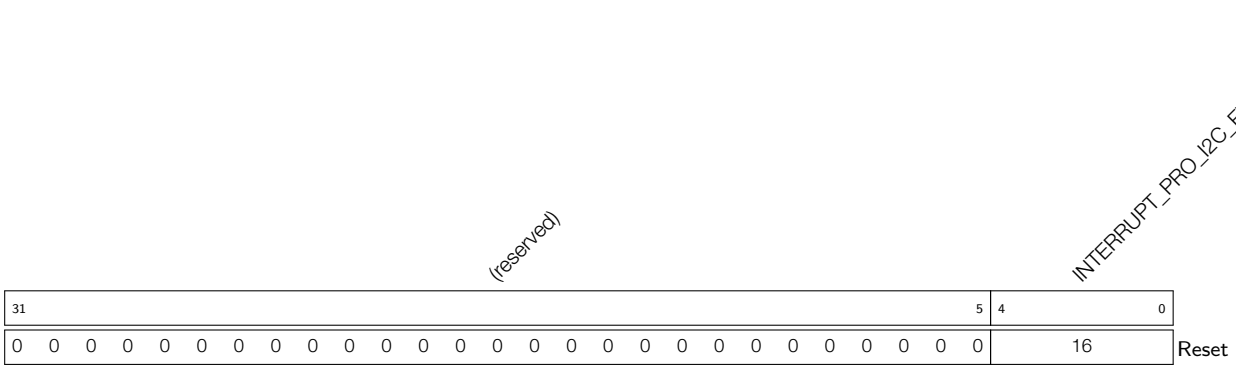
INTERRUPT_PRO_RMT_INTR_MAP 用于将 RMT_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.52: INTERRUPT_PRO_PCNT_INTR_MAP_REG (0x00CC)



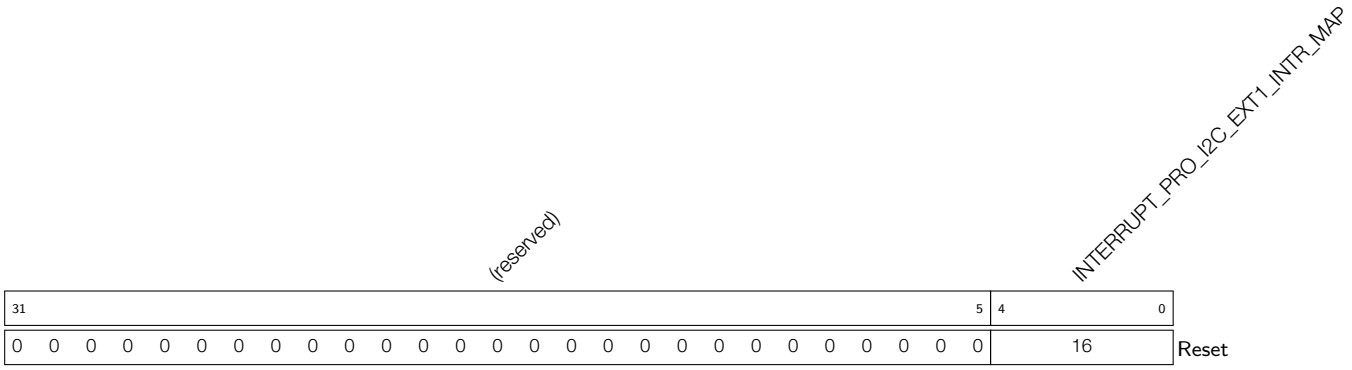
INTERRUPT_PRO_PCNT_INTR_MAP 用于将 PCNT_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.53: INTERRUPT_PRO_I2C_EXT0_INTR_MAP_REG (0x00D0)



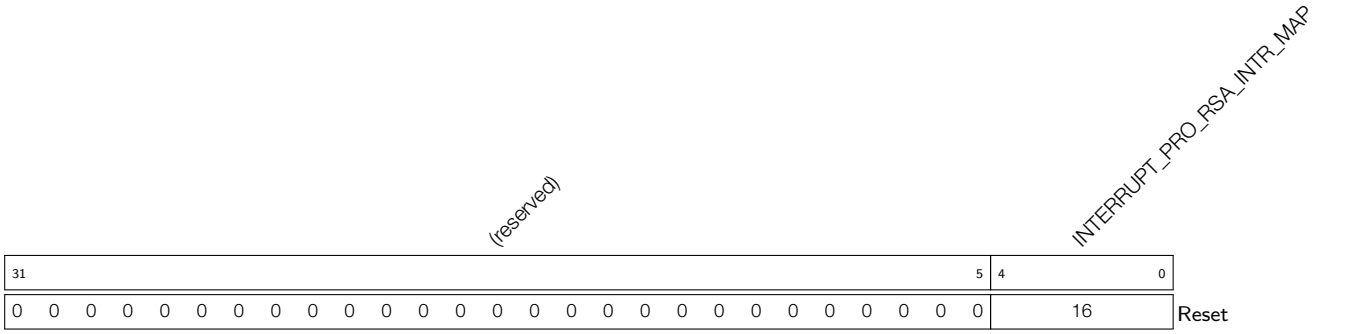
INTERRUPT_PRO_I2C_EXT0_INTR_MAP 用于将 I2C_EXT0_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.54: INTERRUPT_PRO_I2C_EXT1_INTR_MAP_REG (0x00D4)



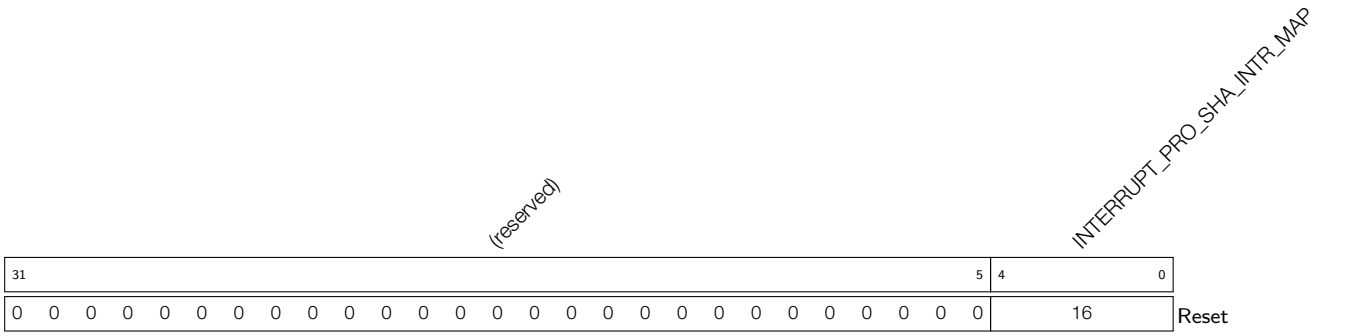
INTERRUPT_PRO_I2C_EXT1_INTR_MAP 用于将 I2C_EXT1_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.55: INTERRUPT_PRO_RSA_INTR_MAP_REG (0x00D8)



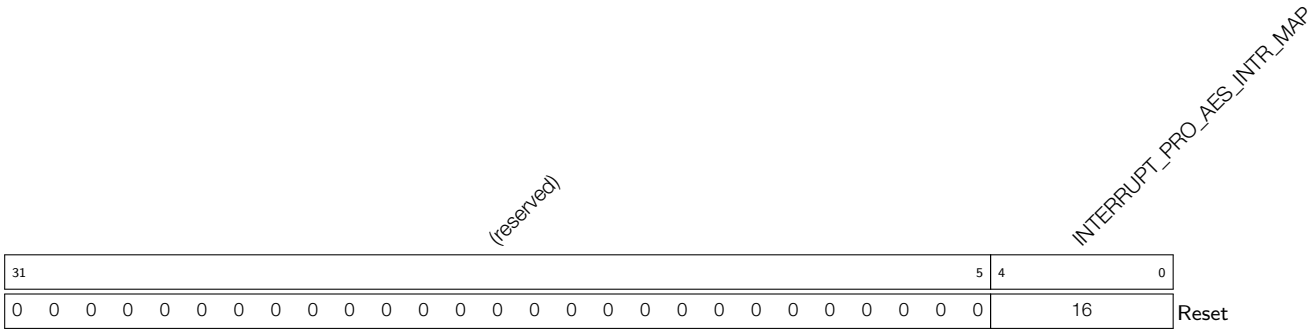
INTERRUPT_PRO_RSA_INTR_MAP 用于将 RSA_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.56: INTERRUPT_PRO_SHA_INTR_MAP_REG (0x00DC)



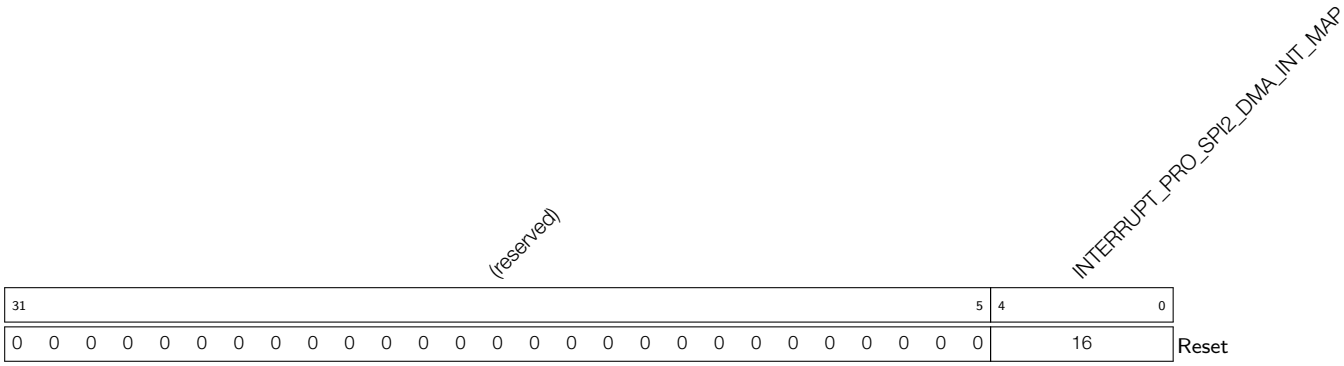
INTERRUPT_PRO_SHA_INTR_MAP 用于将 SHA_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.57: INTERRUPT_PRO_AES_INTR_MAP_REG (0x00E0)



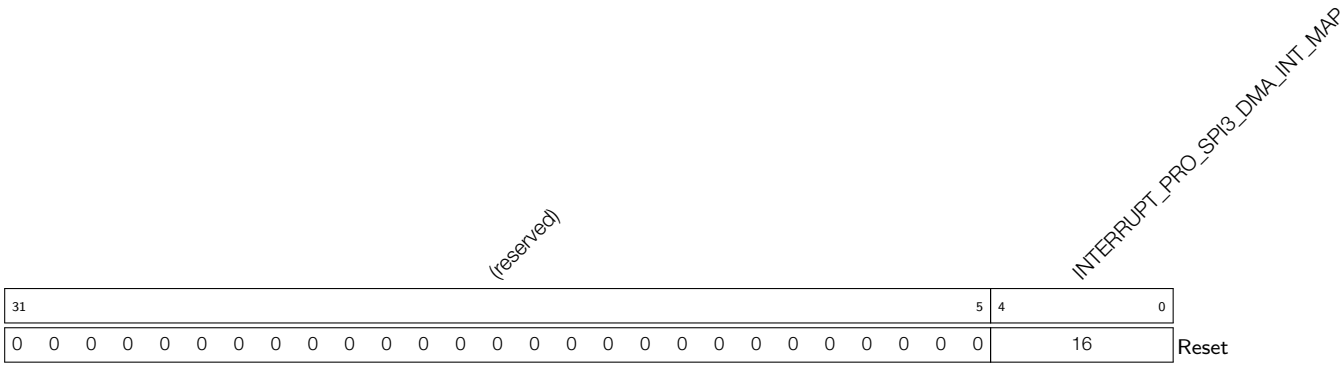
INTERRUPT_PRO_AES_INTR_MAP 用于将 AES_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.58: INTERRUPT_PRO_SPI2_DMA_INT_MAP_REG (0x00E4)



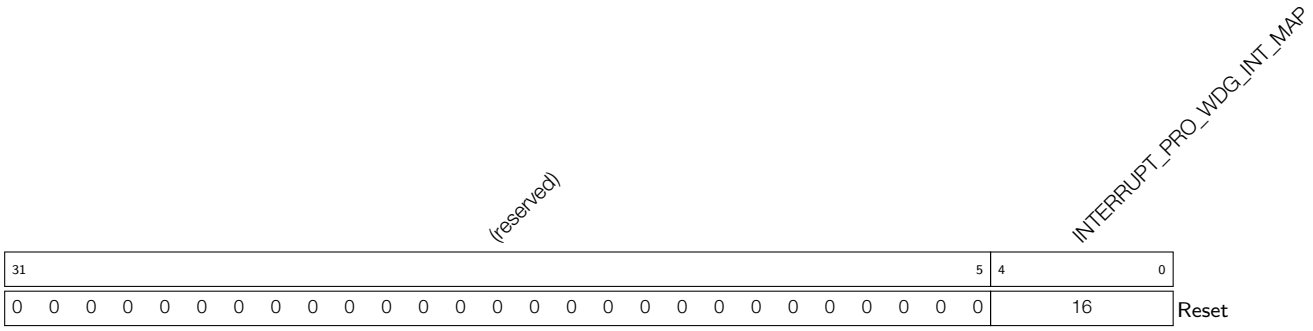
INTERRUPT_PRO_SPI2_DMA_INT_MAP 用于将 SPI2_DMA_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.59: INTERRUPT_PRO_SPI3_DMA_INT_MAP_REG (0x00E8)



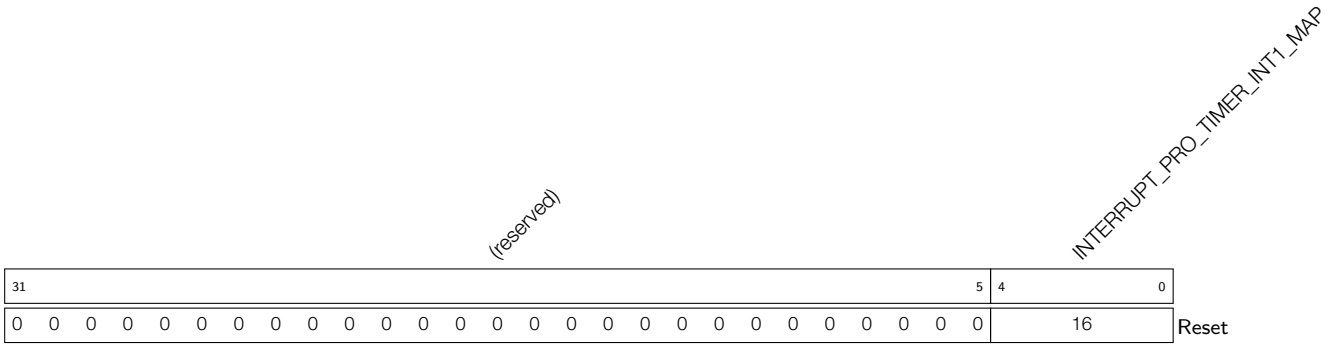
INTERRUPT_PRO_SPI3_DMA_INT_MAP 用于将 SPI3_DMA_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.60: INTERRUPT_PRO_WDG_INT_MAP_REG (0x00EC)



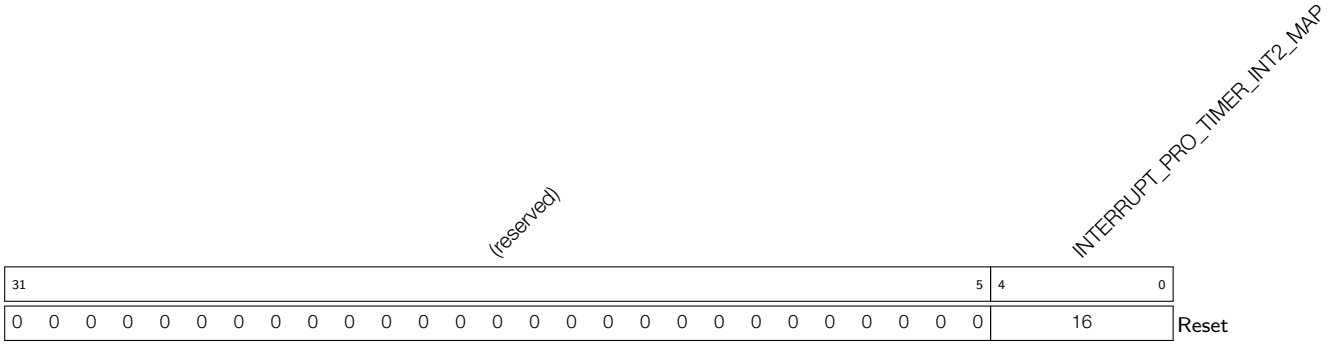
INTERRUPT_PRO_WDG_INT_MAP 用于将 WDG_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.61: INTERRUPT_PRO_TIMER_INT1_MAP_REG (0x00F0)



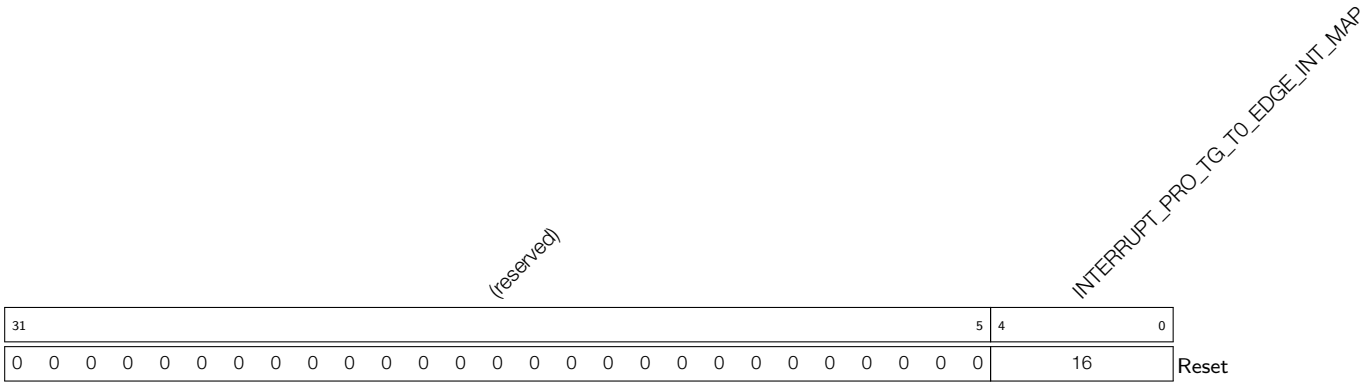
INTERRUPT_PRO_TIMER_INT1_MAP 用于将 TIMER_INT1 中断信号映射至 CPU 中断。(读/写)

Register 4.62: INTERRUPT_PRO_TIMER_INT2_MAP_REG (0x00F4)



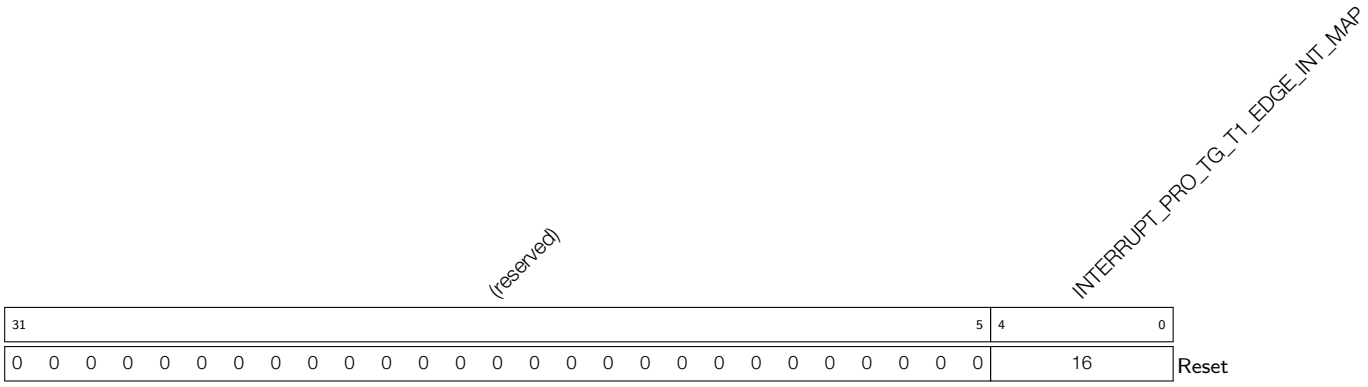
INTERRUPT_PRO_TIMER_INT2_MAP 用于将 TIMER_INT2 中断信号映射至 CPU 中断。(读/写)

Register 4.63: INTERRUPT_PRO_TG_T0_EDGE_INT_MAP_REG (0x00F8)



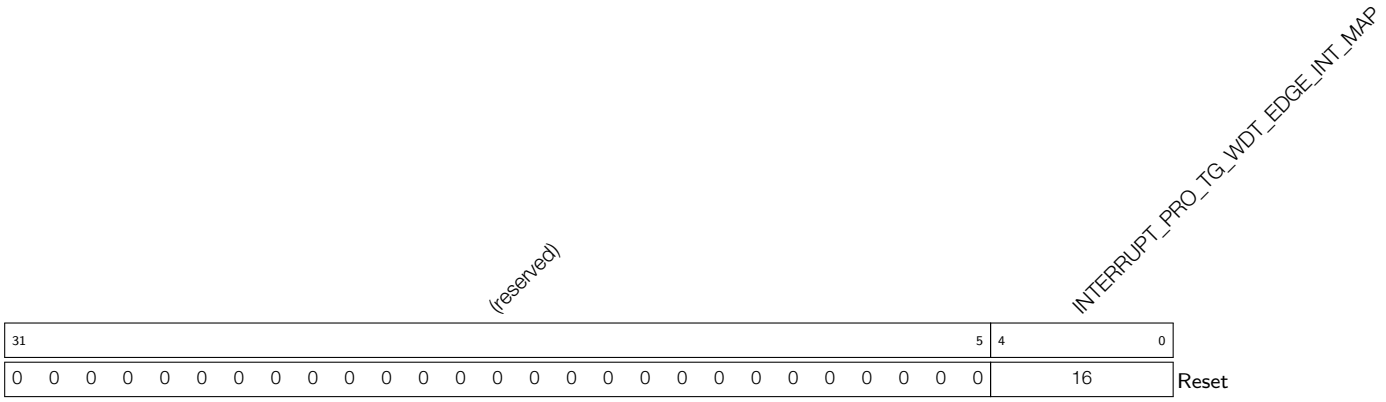
INTERRUPT_PRO_TG_T0_EDGE_INT_MAP 用于将 TG_T0_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.64: INTERRUPT_PRO_TG_T1_EDGE_INT_MAP_REG (0x00FC)



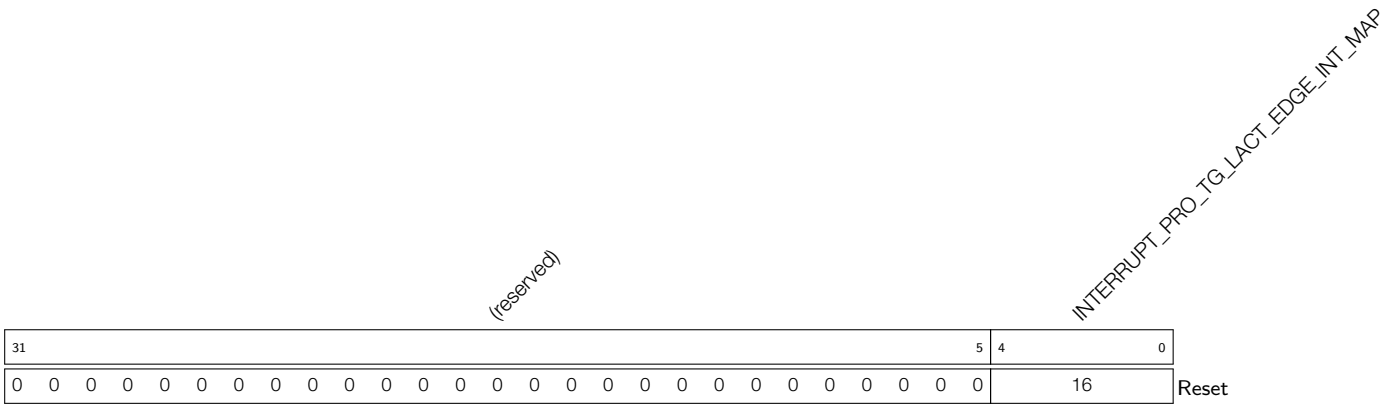
INTERRUPT_PRO_TG_T1_EDGE_INT_MAP 用于将 TG_T1_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.65: INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP_REG (0x0100)



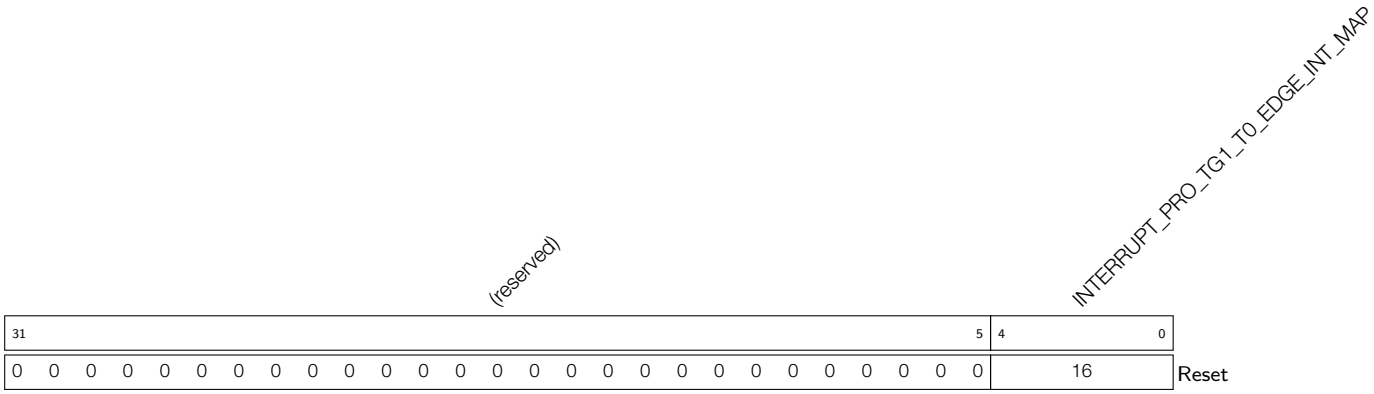
INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP 用于将 TG_WDT_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.66: INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP_REG (0x0104)



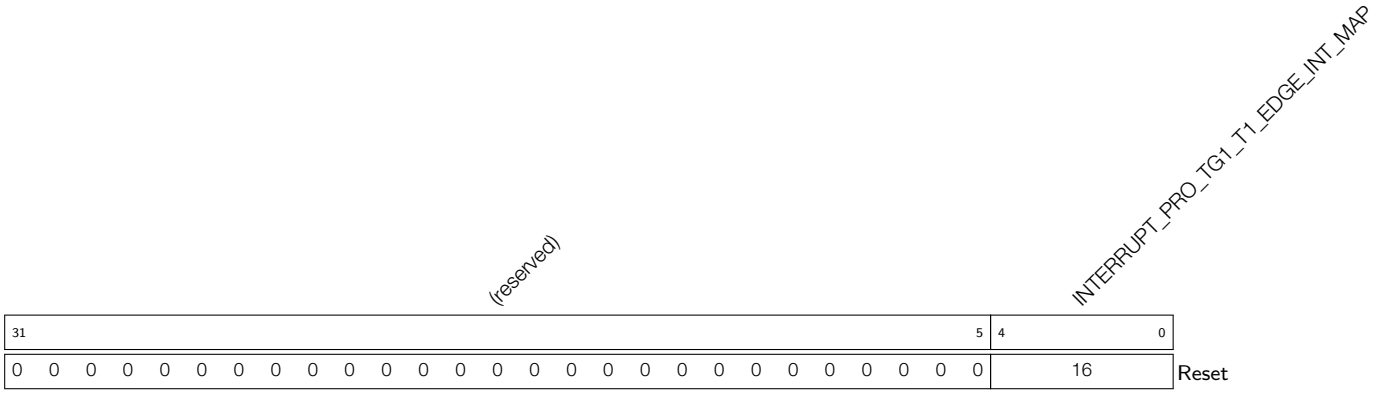
INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP 用于将 TG_LACT_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.67: INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP_REG (0x0108)



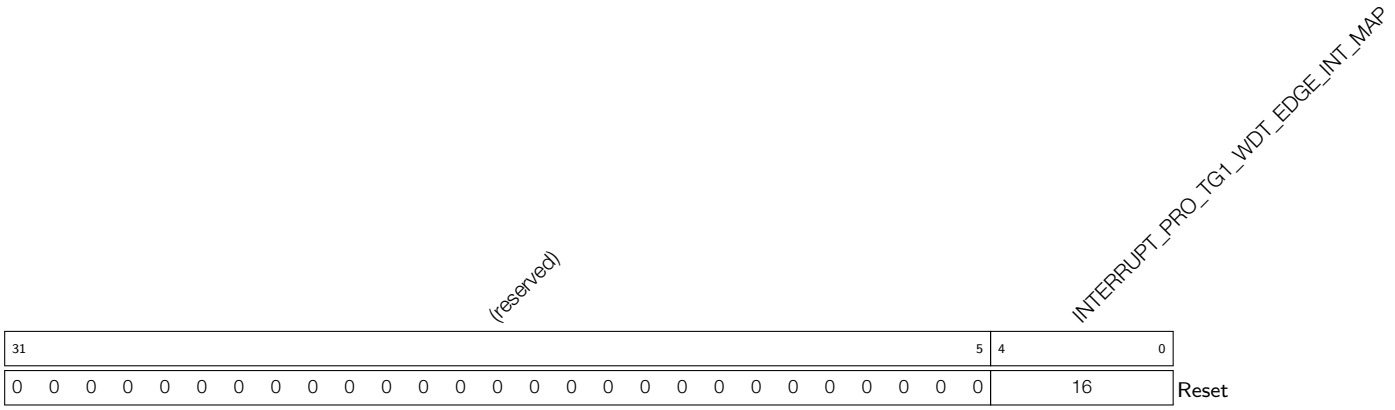
INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP 用于将 TG1_T0_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.68: INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP_REG (0x010C)



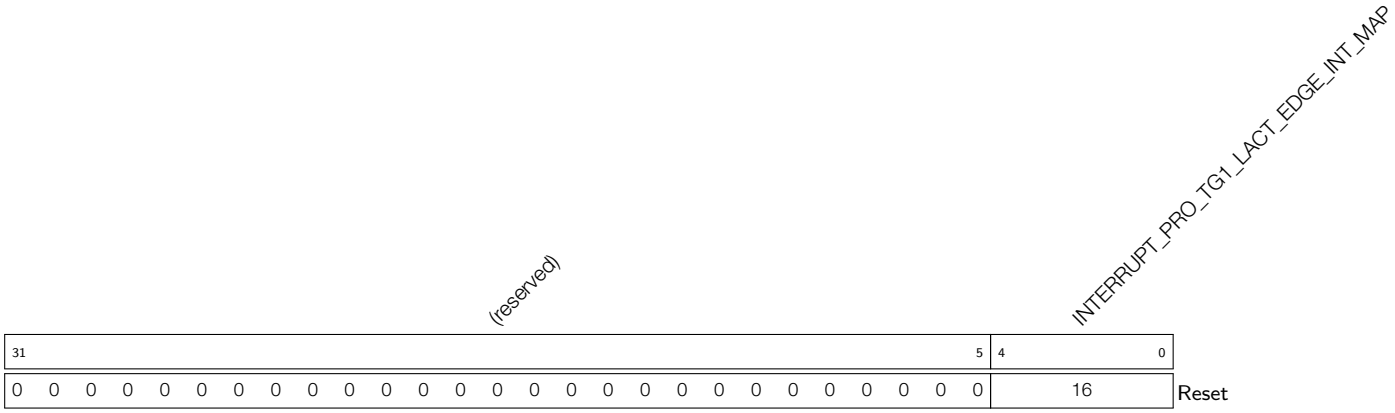
INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP 用于将 TG1_T1_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.69: INTERRUPT_PRO_TG1_WDT_EDGE_INT_MAP_REG (0x0110)



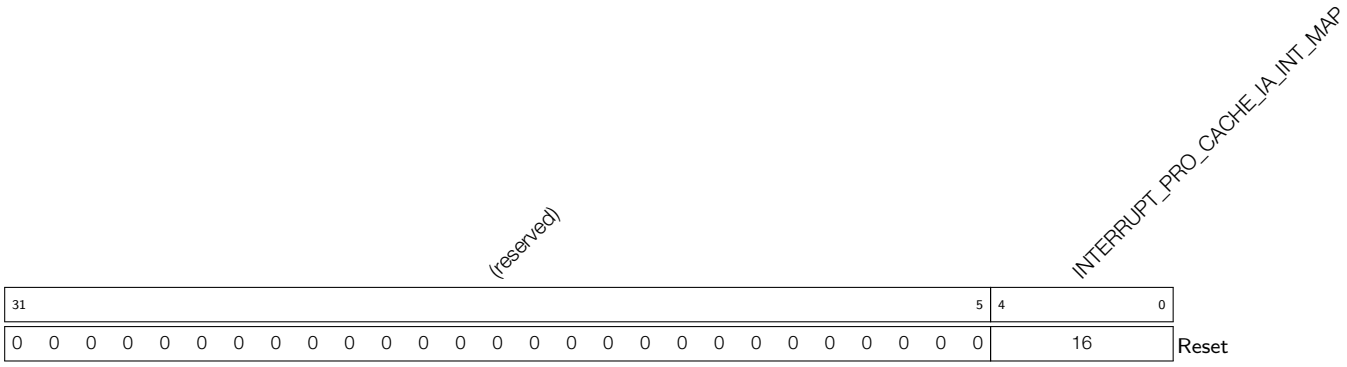
INTERRUPT_PRO_TG1_WDT_EDGE_INT_MAP 用于将 TG1_WDT_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.70: INTERRUPT_PRO_TG1_LACT_EDGE_INT_MAP_REG (0x0114)



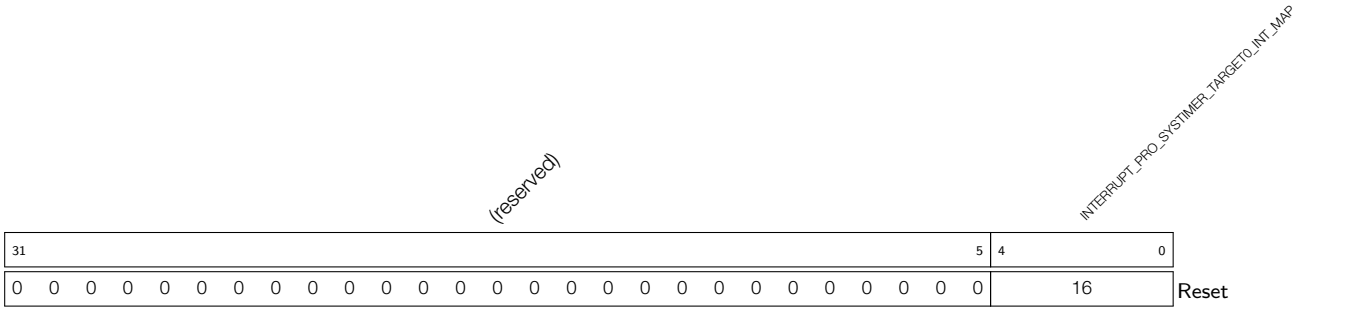
INTERRUPT_PRO_TG1_LACT_EDGE_INT_MAP 用于将 TG1_LACT_EDGE_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.71: INTERRUPT_PRO_CACHE_IA_INT_MAP_REG (0x0118)



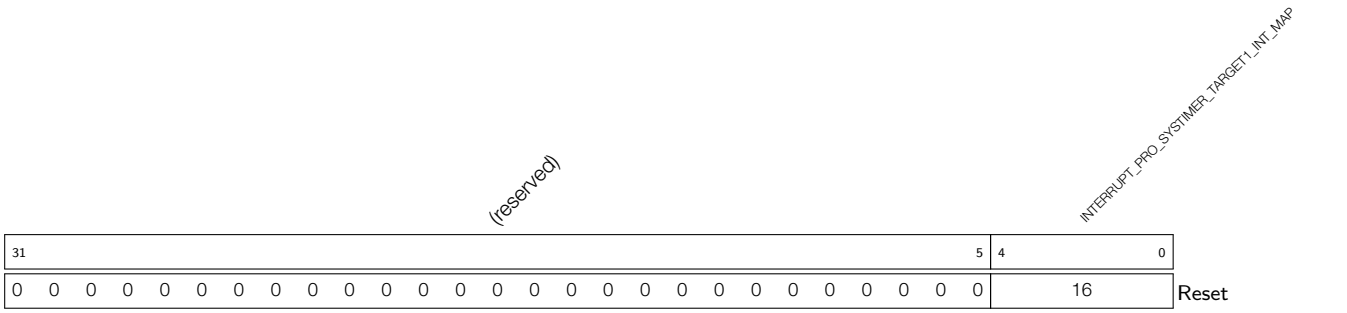
INTERRUPT_PRO_CACHE_IA_INT_MAP 用于将 CACHE_IA_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.72: INTERRUPT_PRO_SYSTIMER_TARGET0_INT_MAP_REG (0x011C)



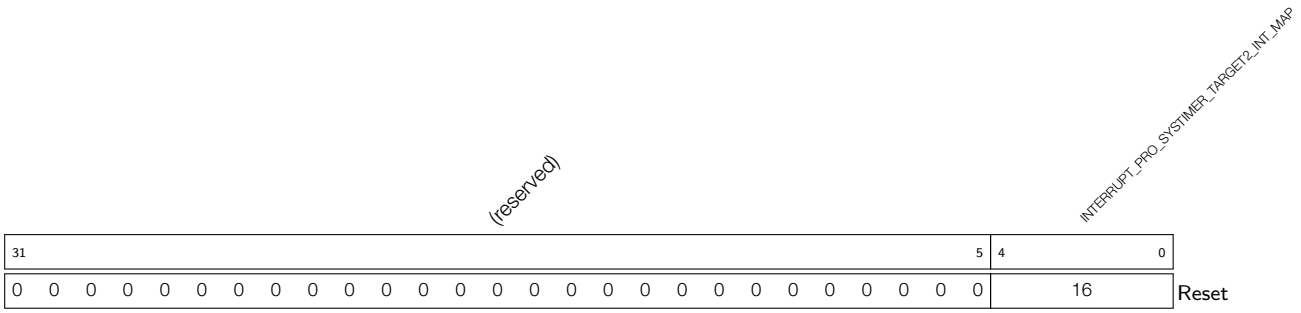
INTERRUPT_PRO_SYSTIMER_TARGET0_INT_MAP 用于将 SYSTIMER_TARGET0 中断信号映射至 CPU 中断。(读/写)

Register 4.73: INTERRUPT_PRO_SYSTIMER_TARGET1_INT_MAP_REG (0x0120)



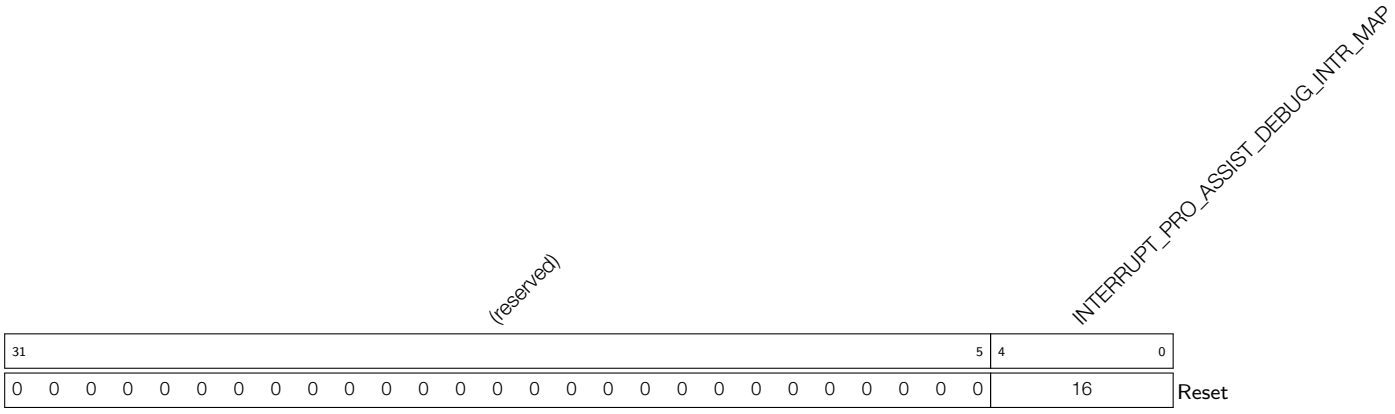
INTERRUPT_PRO_SYSTIMER_TARGET1_INT_MAP 用于将 SYSTIMER_TARGET1 中断信号映射至 CPU 中断。(读/写)

Register 4.74: INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP_REG (0x0124)



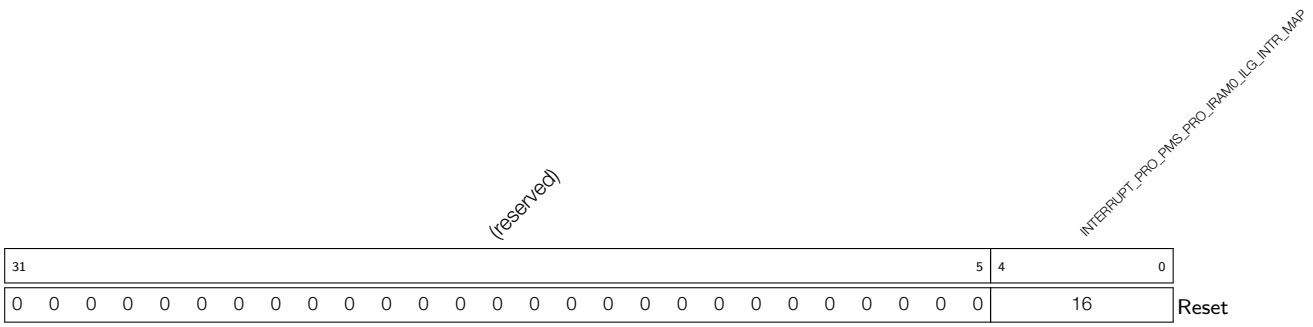
INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP 用于将 SYSTIMER_TARGET2 中断信号映射至 CPU 中断。(读/写)

Register 4.75: INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP_REG (0x0128)



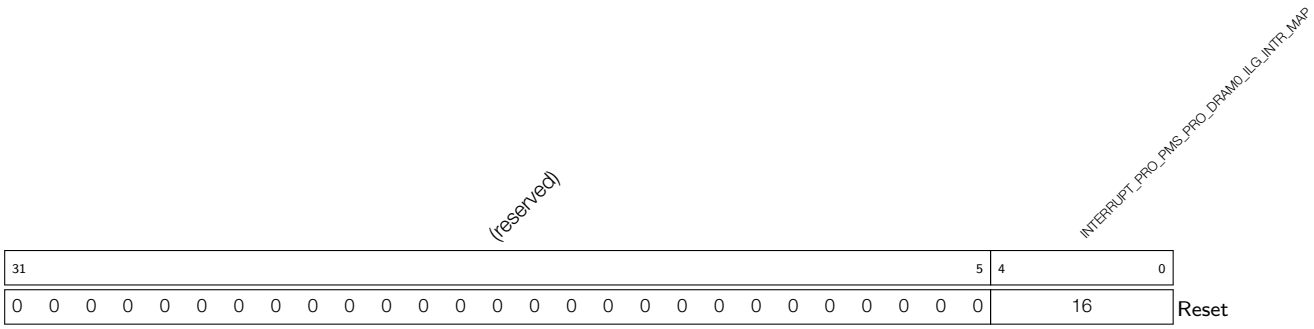
INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP 用于将 ASSIST_DEBUG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.76: INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP_REG (0x012C)



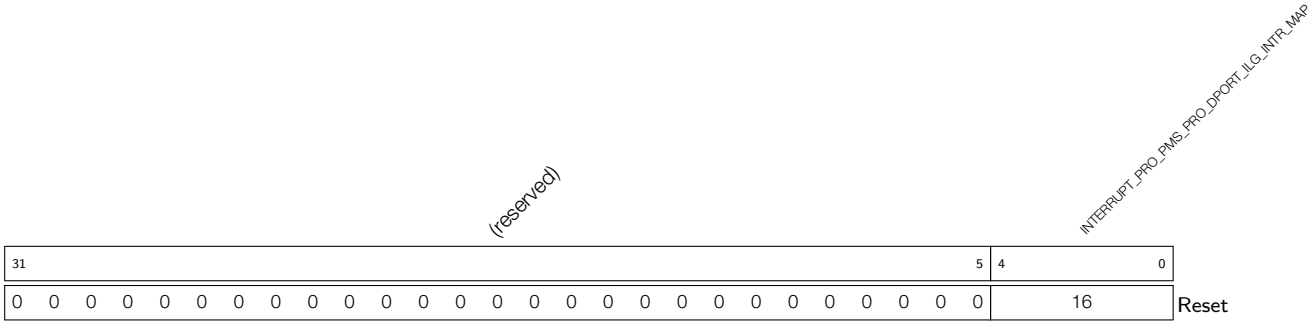
INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP 用于将 PMS_PRO_IRAM0_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.77: INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP_REG (0x0130)



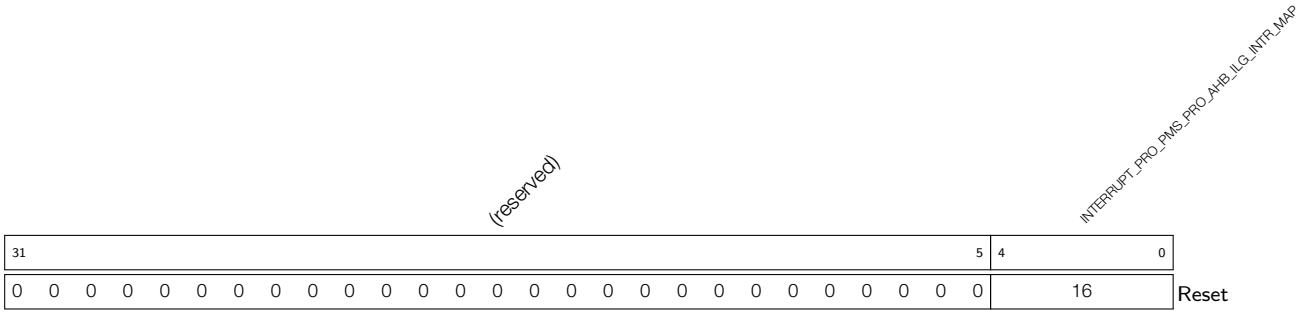
INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP 用于将 PMS_PRO_DRAM0_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.78: INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP_REG (0x0134)



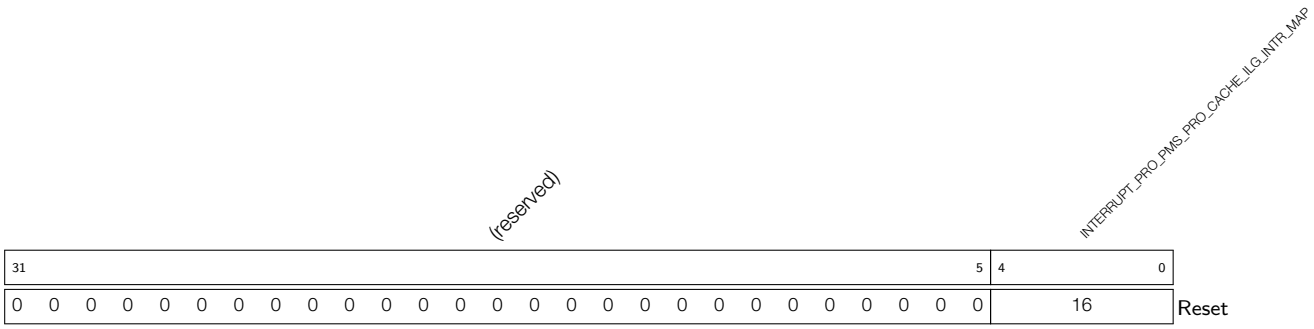
INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP 用于将 PMS_PRO_DPORT_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.79: INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP_REG (0x0138)



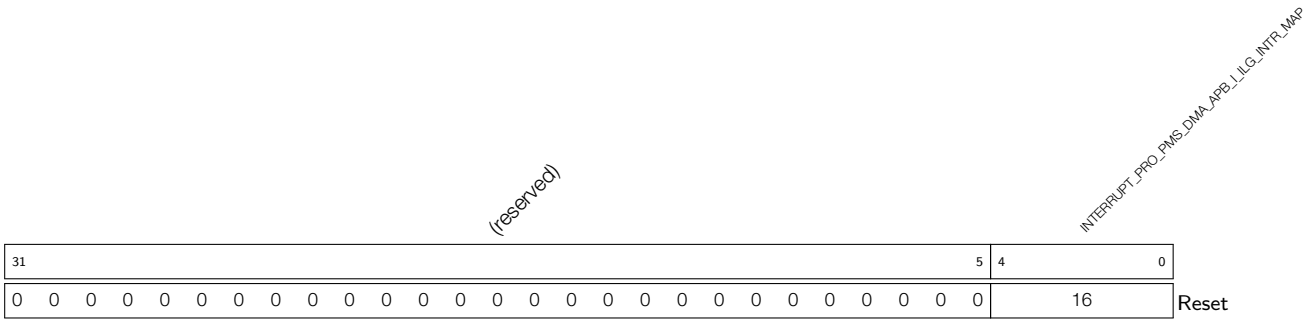
INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP 用于将 PMS_PRO_AHB_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.80: INTERRUPT_PRO_PMS_PRO_CACHE_ILG_INTR_MAP_REG (0x013C)



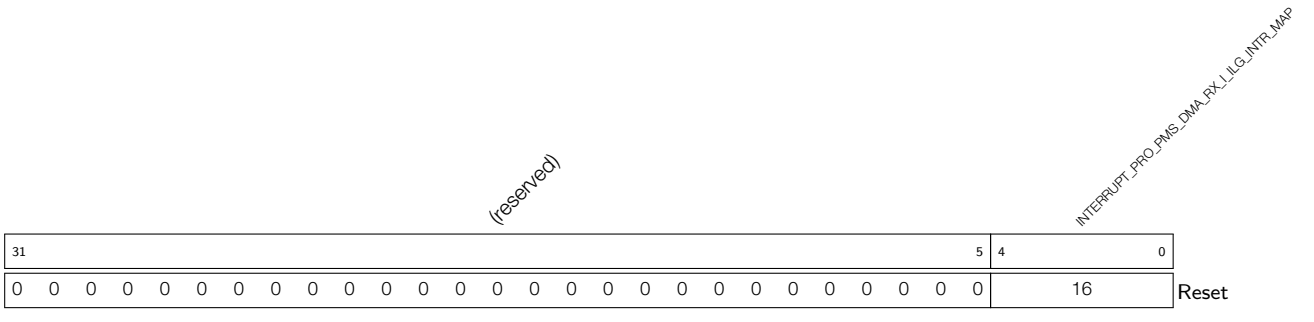
INTERRUPT_PRO_PMS_PRO_CACHE_ILG_INTR_MAP 用于将 PMS_PRO_CACHE_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.81: INTERRUPT_PRO_PMS_DMA_APB_I_ILG_INTR_MAP_REG (0x0140)



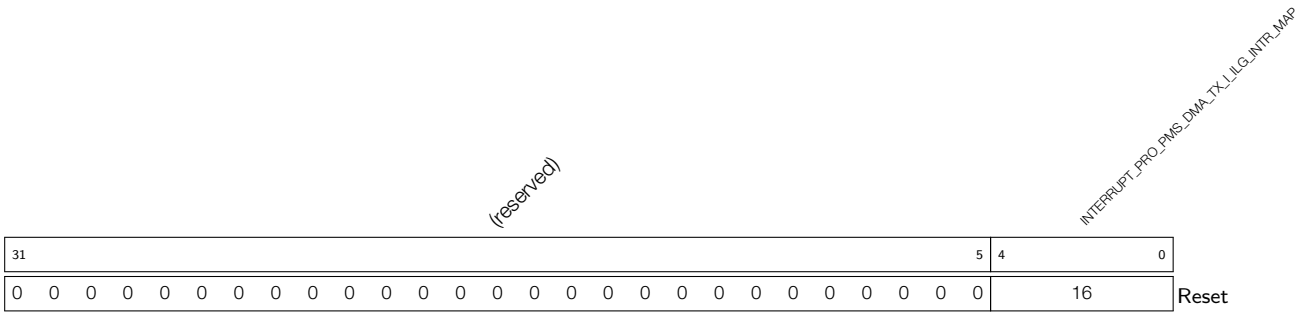
INTERRUPT_PRO_PMS_DMA_APB_I_ILG_INTR_MAP 用于将 PMS_DMA_APB_I_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.82: INTERRUPT_PRO_PMS_DMA_RX_I_ILG_INTR_MAP_REG (0x0144)



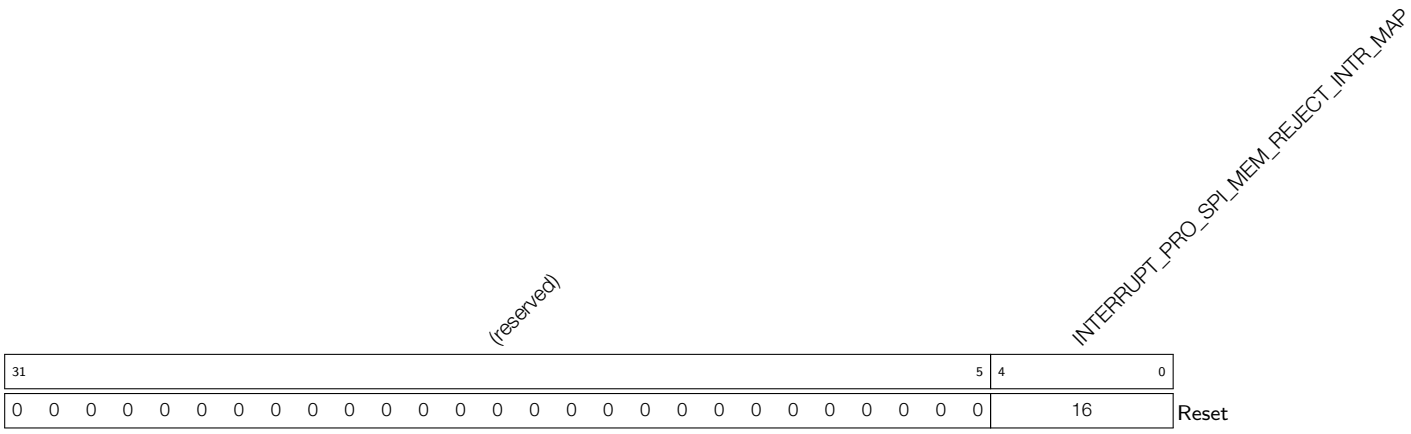
INTERRUPT_PRO_PMS_DMA_RX_I_ILG_INTR_MAP 用于将 PMS_DMA_RX_I_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.83: INTERRUPT_PRO_PMS_DMA_TX_I_ILG_INTR_MAP_REG (0x0148)



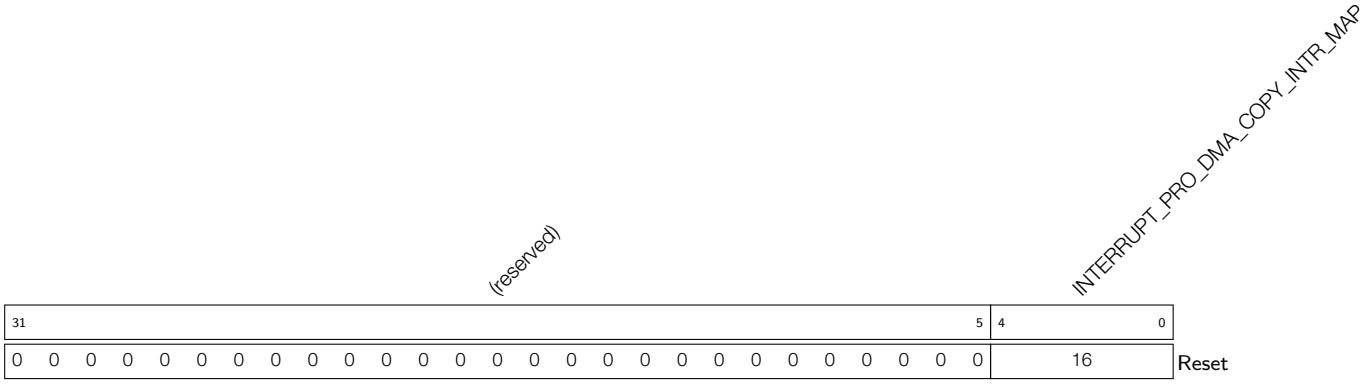
INTERRUPT_PRO_PMS_DMA_TX_I_ILG_INTR_MAP 用于将 PMS_DMA_TX_I_ILG_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.84: INTERRUPT_PRO_SPI_MEM_REJECT_INTR_MAP_REG (0x014C)



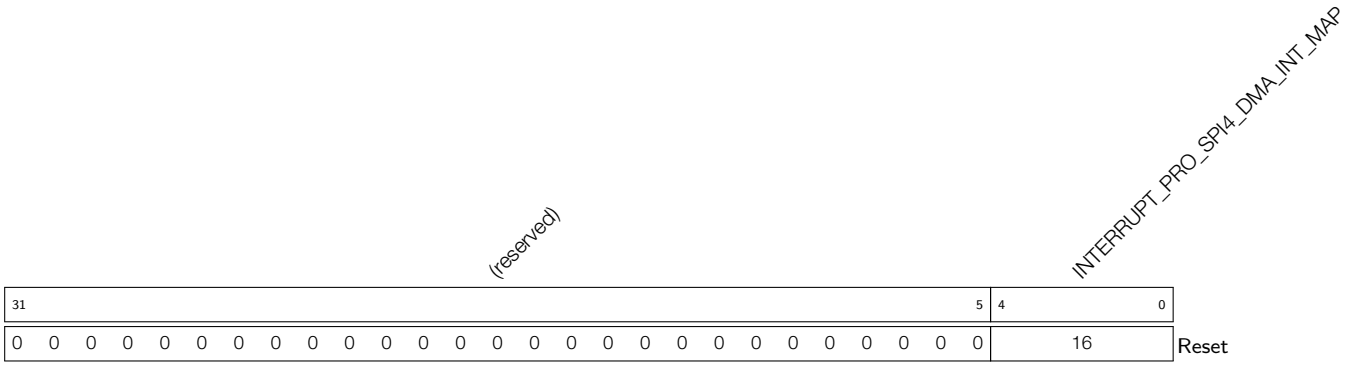
INTERRUPT_PRO_SPI_MEM_REJECT_INTR_MAP 用于将 SPI_MEM_REJECT_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.85: INTERRUPT_PRO_DMA_COPY_INTR_MAP_REG (0x0150)



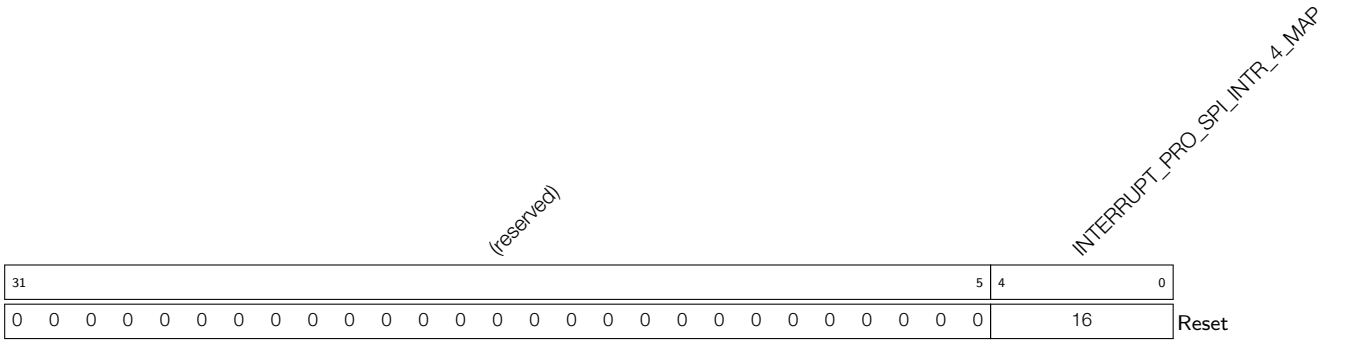
INTERRUPT_PRO_DMA_COPY_INTR_MAP 用于将 DMA_COPY_INTR 中断信号映射至 CPU 中断。(读/写)

Register 4.86: INTERRUPT_PRO_SPI4_DMA_INT_MAP_REG (0x0154)



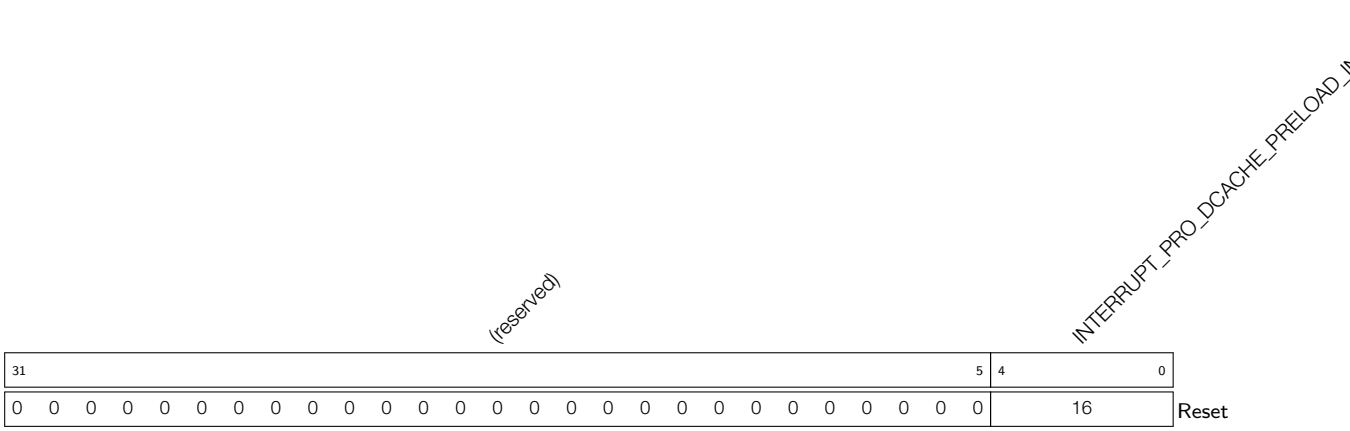
INTERRUPT_PRO_SPI4_DMA_INT_MAP 用于将 SPI4_DMA_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.87: INTERRUPT_PRO_SPI_INTR_4_MAP_REG (0x0158)



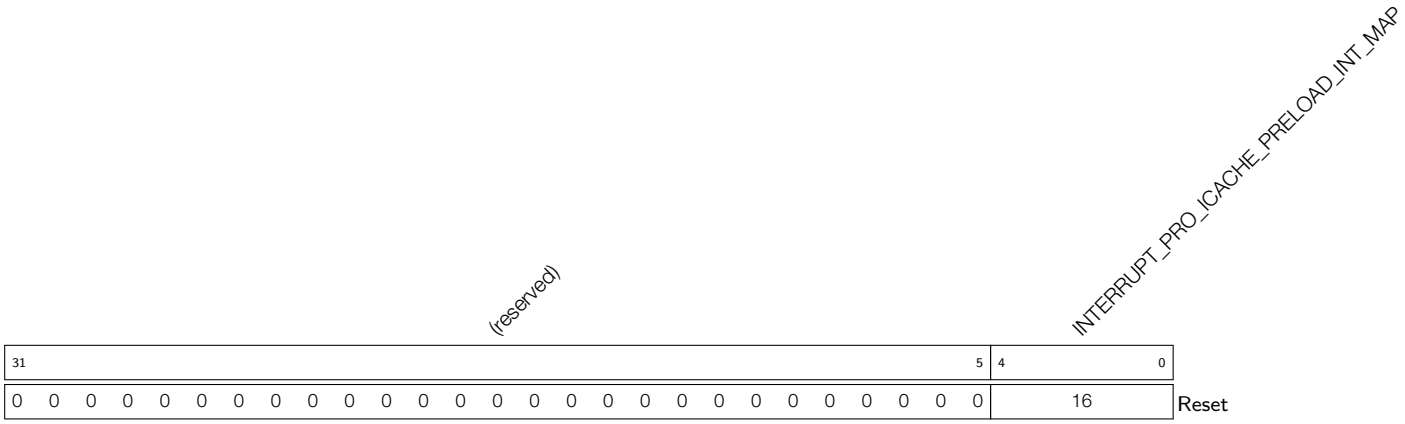
INTERRUPT_PRO_SPI_INTR_4_MAP 用于将 SPI_INTR_4 中断信号映射至 CPU 中断。(读/写)

Register 4.88: INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP_REG (0x015C)



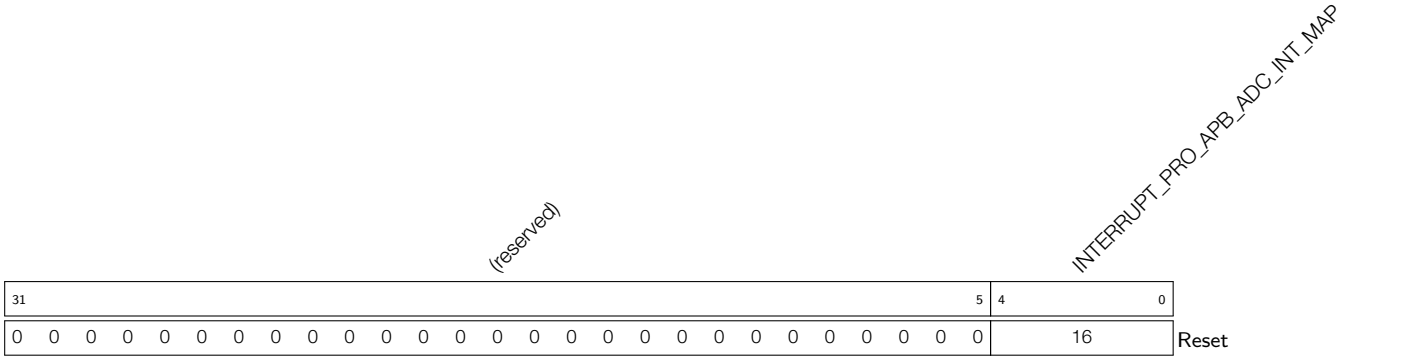
INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP 用于将 DCACHE_PRELOAD_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.89: INTERRUPT_PRO_ICACHE_PRELOAD_INT_MAP_REG (0x0160)



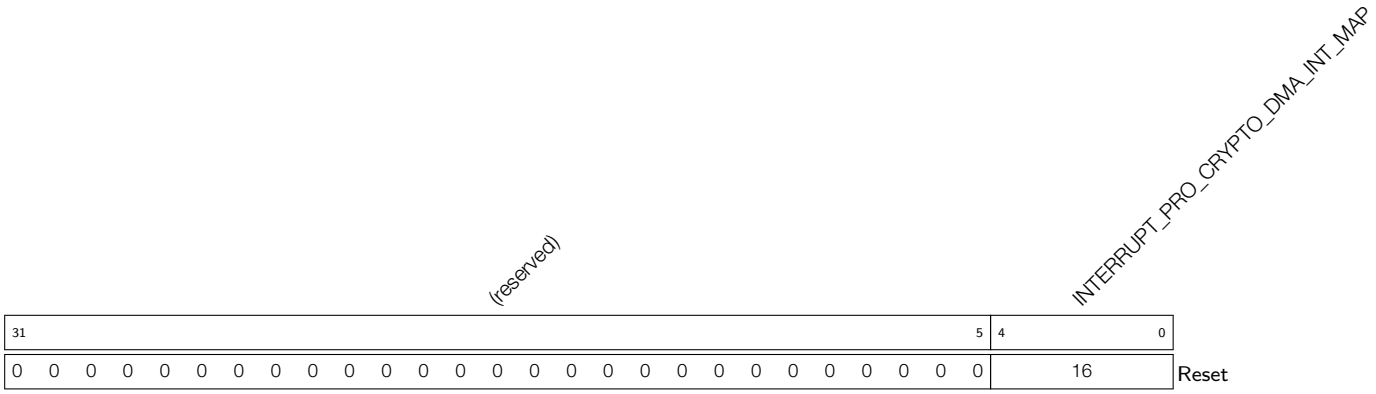
INTERRUPT_PRO_ICACHE_PRELOAD_INT_MAP 用于将 ICACHE_PRELOAD_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.90: INTERRUPT_PRO_APB_ADC_INT_MAP_REG (0x0164)



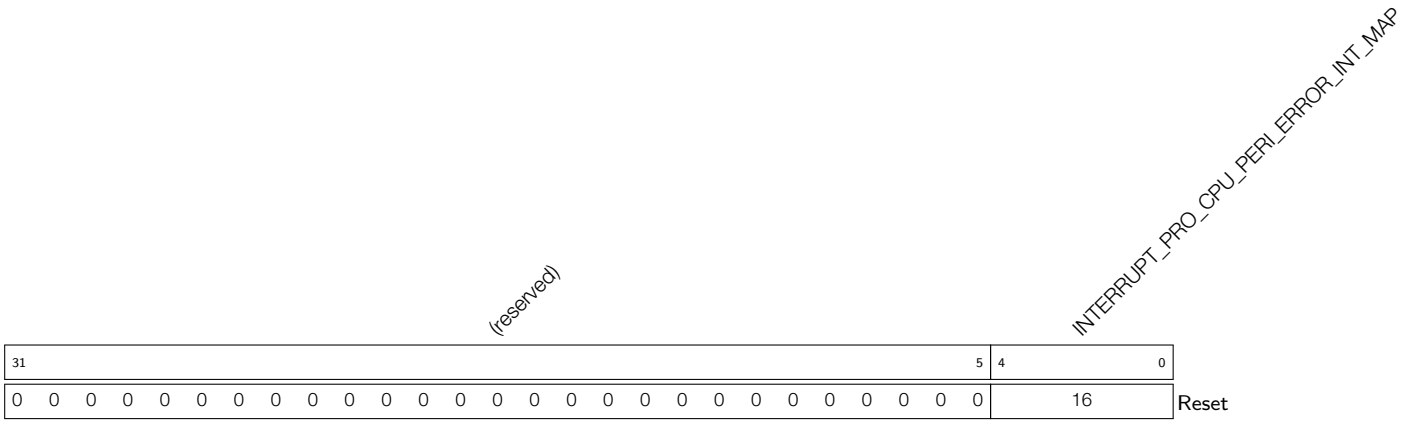
INTERRUPT_PRO_APB_ADC_INT_MAP 用于将 APB_ADC_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.91: INTERRUPT_PRO_CRYPTO_DMA_INT_MAP_REG (0x0168)



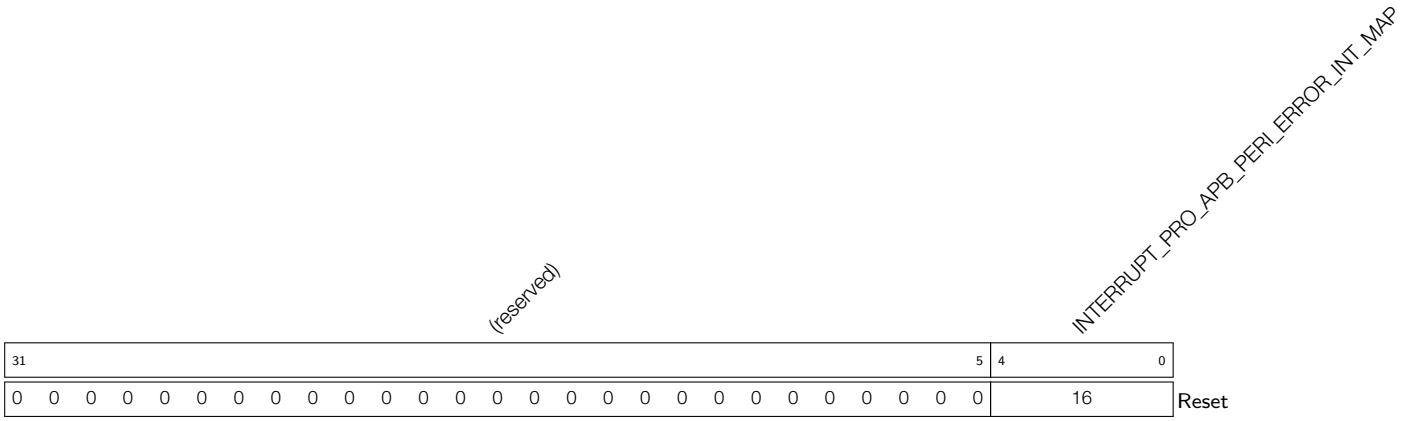
INTERRUPT_PRO_CRYPTO_DMA_INT_MAP 用于将 CRYPTO_DMA_INT 中断信号映射至 CPU 中断。（读/写）

Register 4.92: INTERRUPT_PRO_CPU_PERI_ERROR_INT_MAP_REG (0x016C)



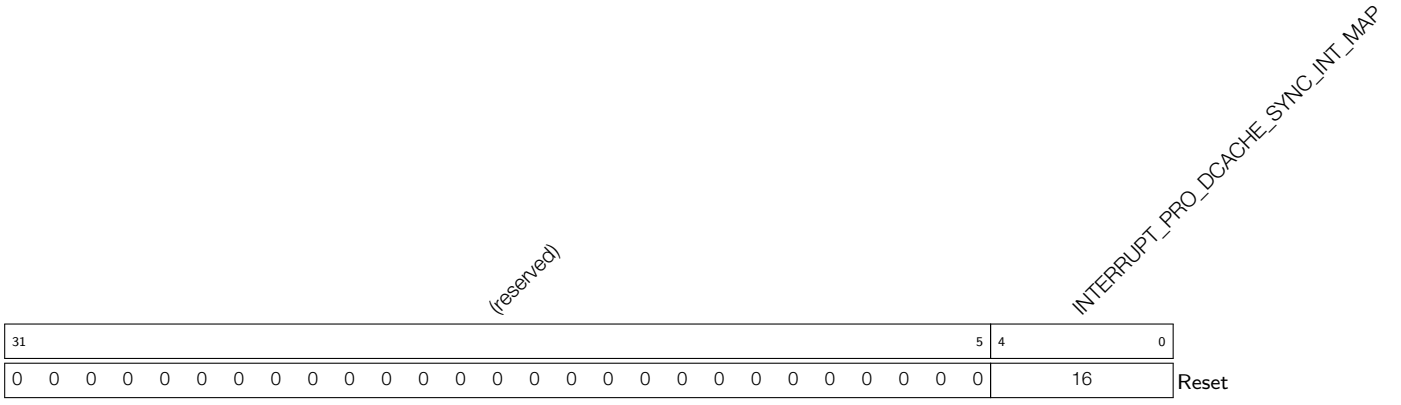
INTERRUPT_PRO_CPU_PERI_ERROR_INT_MAP 用于将 CPU_PERI_ERROR_INT 中断信号映射至 CPU 中断。（读/写）

Register 4.93: INTERRUPT_PRO_APB_PERI_ERROR_INT_MAP_REG (0x0170)



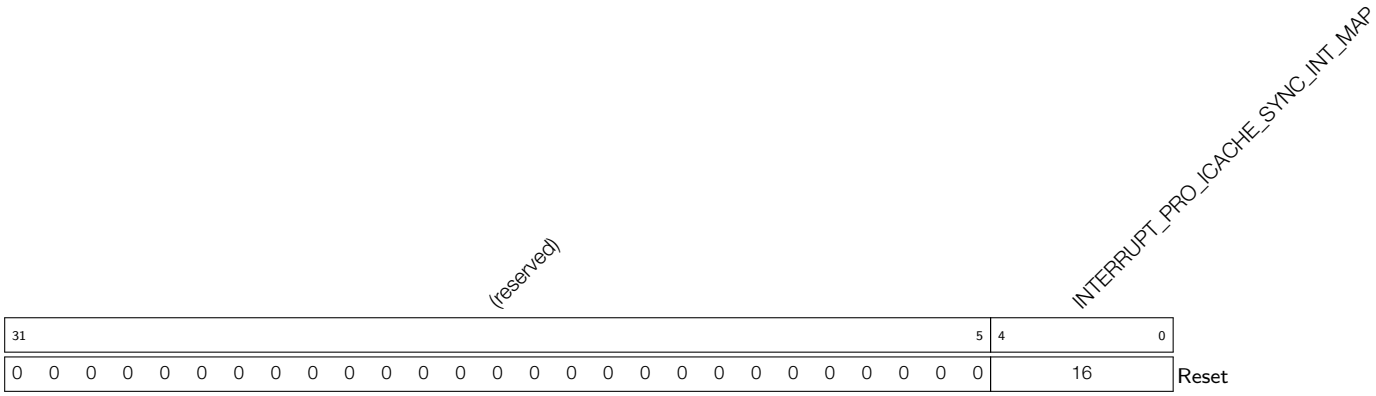
INTERRUPT_PRO_APB_PERI_ERROR_INT_MAP 用于将 APB_PERI_ERROR_INT 中断信号映射至 CPU 中断。(读/写)

Register 4.94: INTERRUPT_PRO_DCACHE_SYNC_INT_MAP_REG (0x0174)



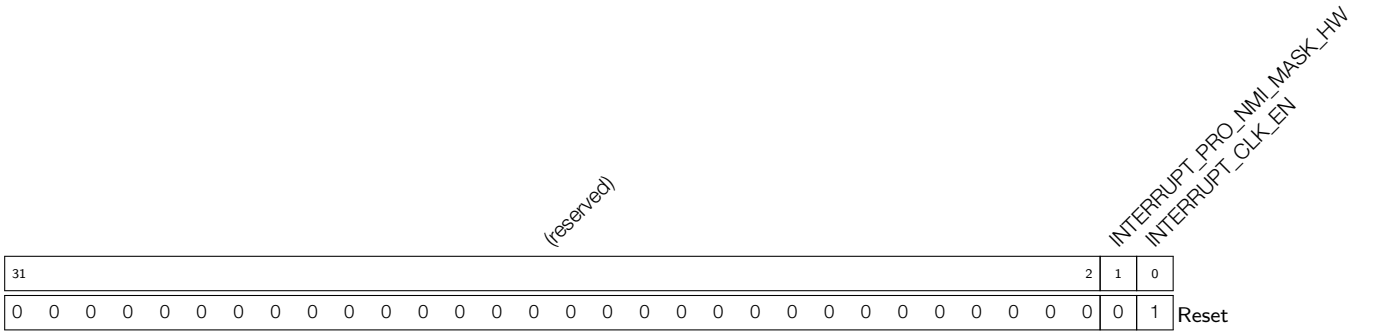
INTERRUPT_PRO_DCACHE_SYNC_INT_MAP 用于将 DCACHE_SYNC_INT 中断信号映射至 CPU 中断。(读/写)

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)

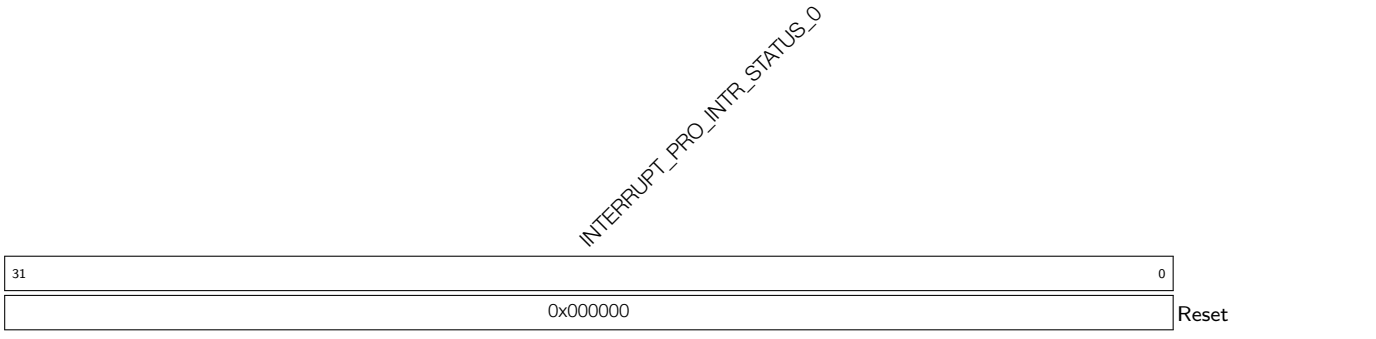
Register 4.96: INTERRUPT_CLOCK_GATE_REG (0x0188)



INTERRUPT_CLK_EN 使能或关闭中断矩阵的时钟。1：使能时钟；2：关闭时钟。(读/写)

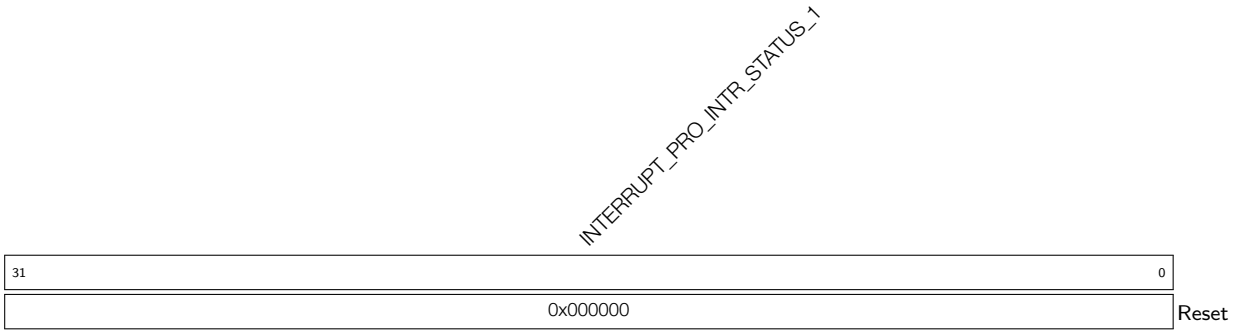
INTERRUPT_PRO_NMI_MASK_HW 屏蔽所有 NMI 中断信号映射至 CPU。(读/写)

Register 4.97: INTERRUPT_PRO_INTR_STATUS_REG_0_REG (0x017C)



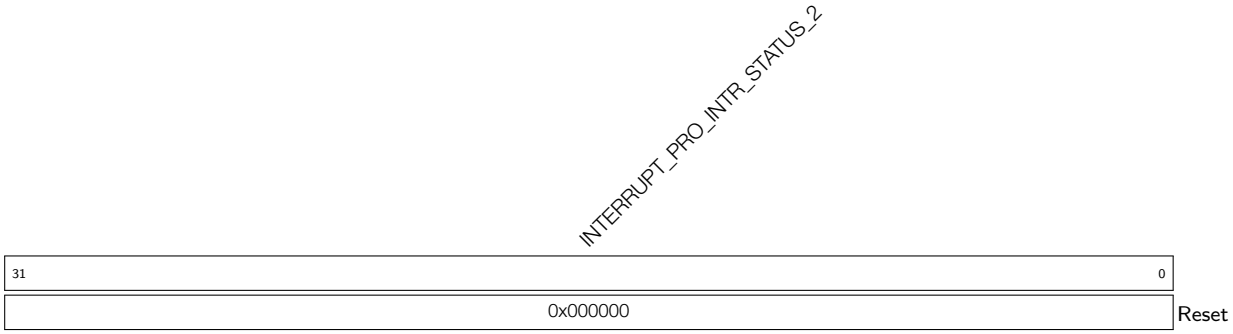
INTERRUPT_PRO_INTR_STATUS_0 用于存储前 32 个中断源的状态。(只读)

Register 4.98: INTERRUPT_PRO_INTR_STATUS_REG_1_REG (0x0180)



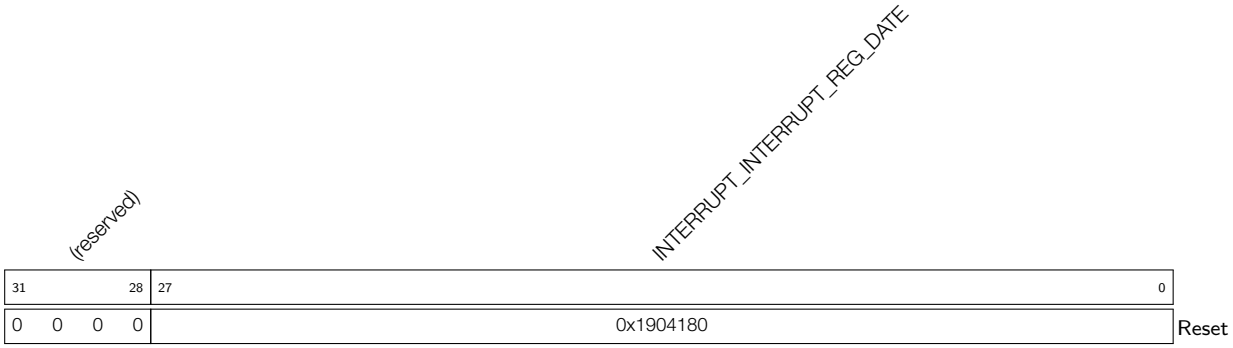
INTERRUPT_PRO_INTR_STATUS_1 用于存储中间 32 个中断源的状态。(只读)

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. 系统寄存器

5.1 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。

5.2 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

5.3 功能描述

5.3.1 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 5-1。

表 5-1. 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 5-2。

表 5-2. 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

5.3.2 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

5.3.3 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

5.3.4 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

5.3.5 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

5.3.6 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 5-3。

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

表 5-3. 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_SPI2_DMA_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。

5.4 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 5-4 所示。更多信息，请访问章节 1 系统和存储器。

表 5-4. 系统寄存器基地址

访问总线	基地址
PeriBUS1	0x3F4C0000

5.5 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 5.4 获取有关系统寄存器基地址的信息。

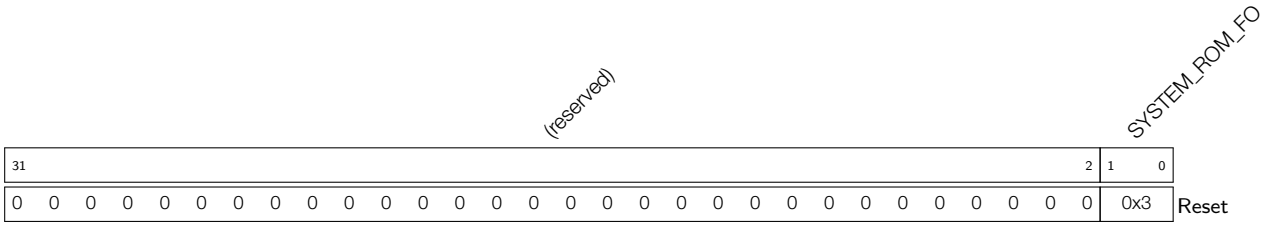
名称	描述	地址	访问权限
系统和存储器寄存器			
SYSTEM_ROM_CTRL_0_REG	系统 ROM 配置寄存器 0	0x0000	读写
SYSTEM_ROM_CTRL_1_REG	系统 ROM 配置寄存器 1	0x0004	读写
SYSTEM_SRAM_CTRL_0_REG	系统 SRAM 配置寄存器 0	0x0008	读写
SYSTEM_SRAM_CTRL_1_REG	系统 SRAM 配置寄存器 1	0x000C	读写
SYSTEM_SRAM_CTRL_2_REG	系统 SRAM 配置寄存器 2	0x0088	读写
SYSTEM_RSA_PD_CTRL_REG	RSA 存储掉电寄存器	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

名称	描述	地址	访问权限
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
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

5.6 寄存器

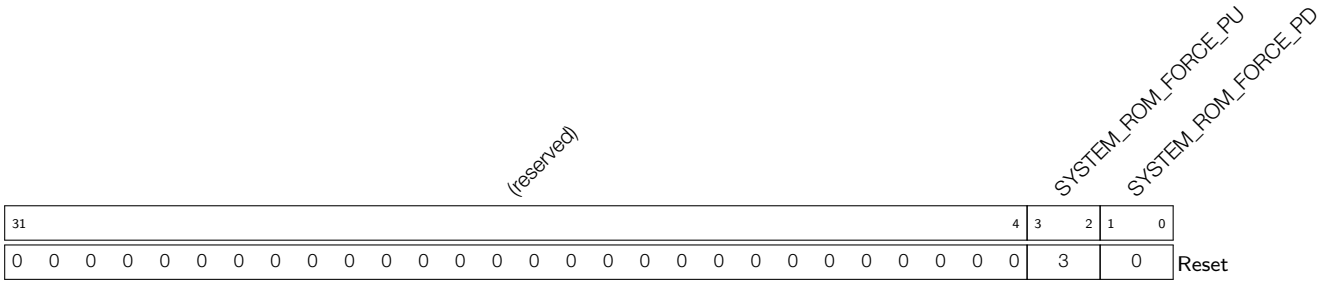
请注意，这里的地址是相对于系统寄存器基地址的地址偏移量（相对地址）。请参阅章节 5.4 获取有关系统寄存器基地址的信息。

Register 5.1: SYSTEM_ROM_CTRL_0_REG (0x0000)



SYSTEM_ROM_FO 强制打开内部 ROM 的时钟门控。详见表 5-1。（读写）

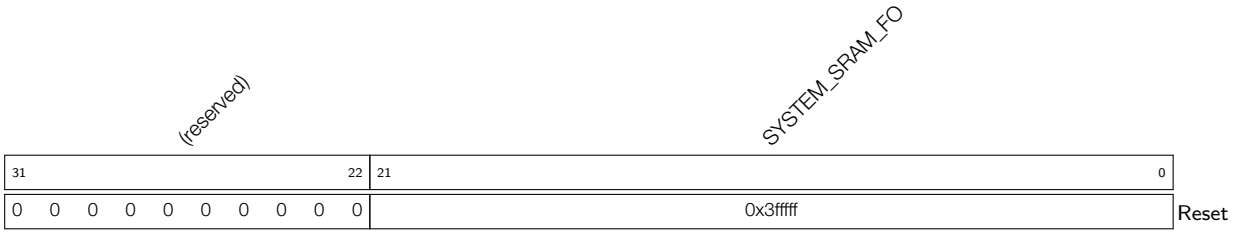
Register 5.2: SYSTEM_ROM_CTRL_1_REG (0x0004)



SYSTEM_ROM_FORCE_PD 控制内部 ROM 的掉电。详见表 5-1。(读写)

SYSTEM_ROM_FORCE_PU 控制内部 ROM 的上电。详见表 5-1。(读写)

Register 5.3: SYSTEM_SRAM_CTRL_0_REG (0x0008)

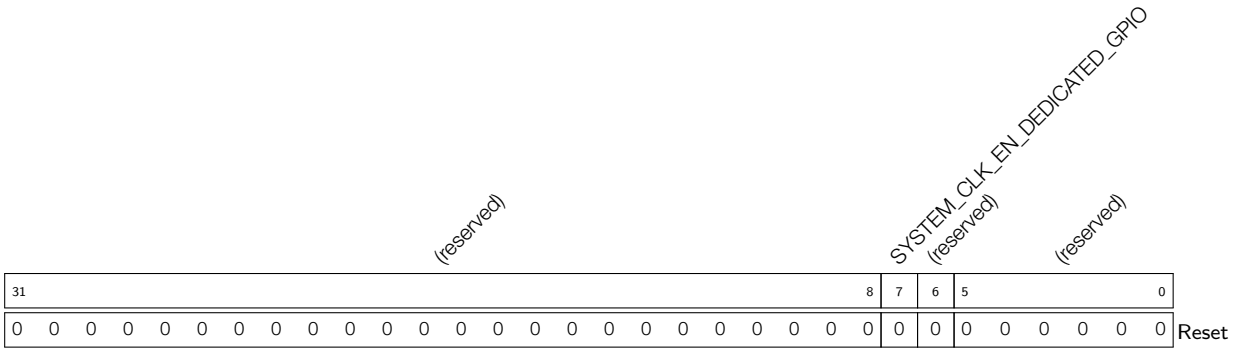


SYSTEM_SRAM_FO 强制打开内部 SRAM 的时钟门控。详见表 5-2。(读写)

Register 5.4: SYSTEM_SRAM_CTRL_1_REG (0x000C)

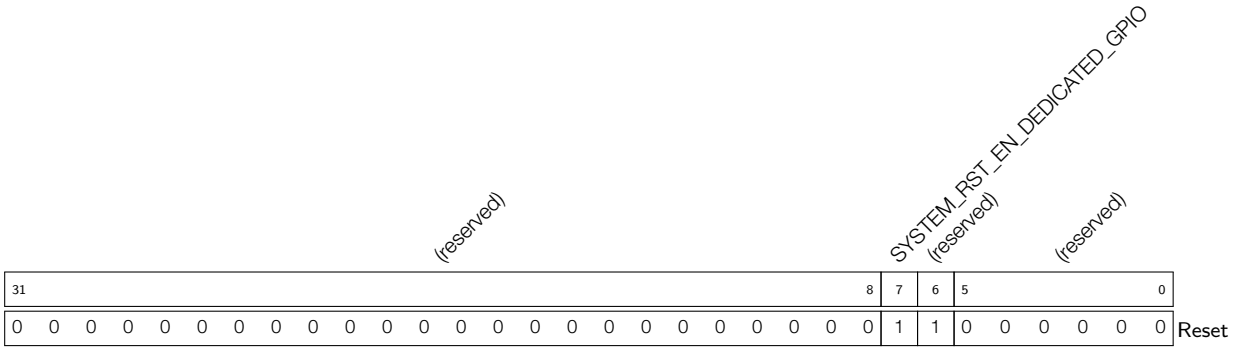
SYSTEM_SRAM_FORCE_PD 控制内部 SRAM 的掉电。详见表 5-2。(读写)

Register 5.5: SYSTEM_CPU_PERI_CLK_EN_REG (0x0010)



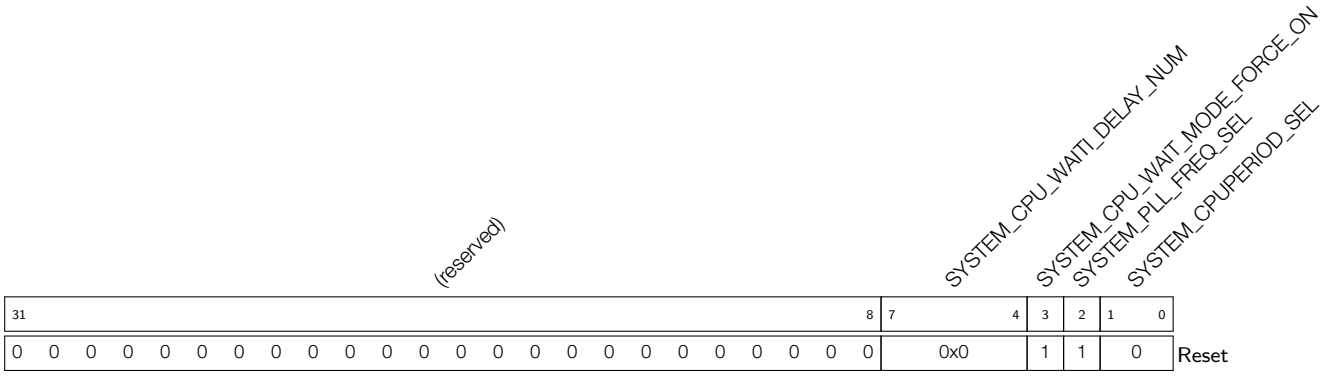
SYSTEM_CLK_EN_DEDICATED_GPIO 置 1 打开 DEDICATED GPIO 时钟。（读写）

Register 5.6: SYSTEM_CPU_PERI_RST_EN_REG (0x0014)



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

Register 5.7: SYSTEM_CPU_PER_CONF_REG (0x0018)



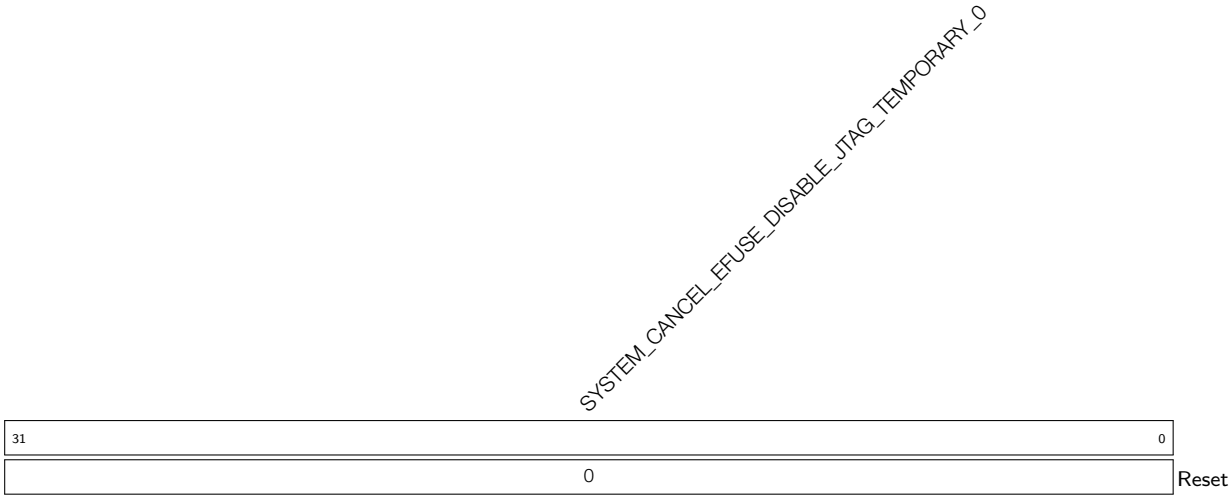
SYSTEM_CPUPERIOD_SEL 选择 CPU 时钟周期或时钟频率。详见章节 2 复位和时钟中表 2-2。(读写)

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)

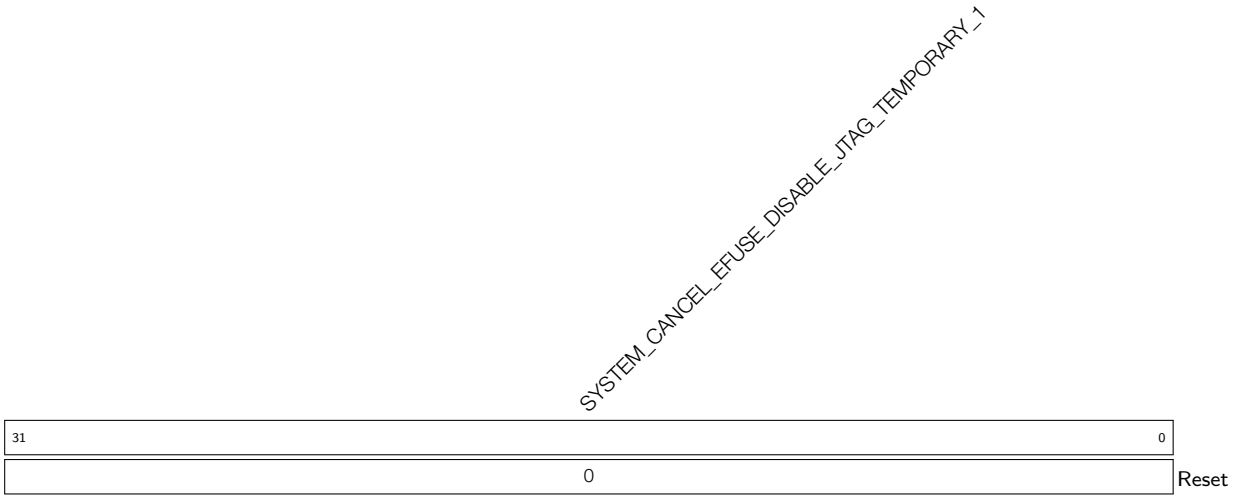
SYSTEM_CPU_WAITI_DELAY_NUM 设置 CPU 在收到 WAITI 指令后进入 CPU 等待模式前的等待周期。(读写)

Register 5.8: SYSTEM_JTAG_CTRL_0_REG (0x001C)



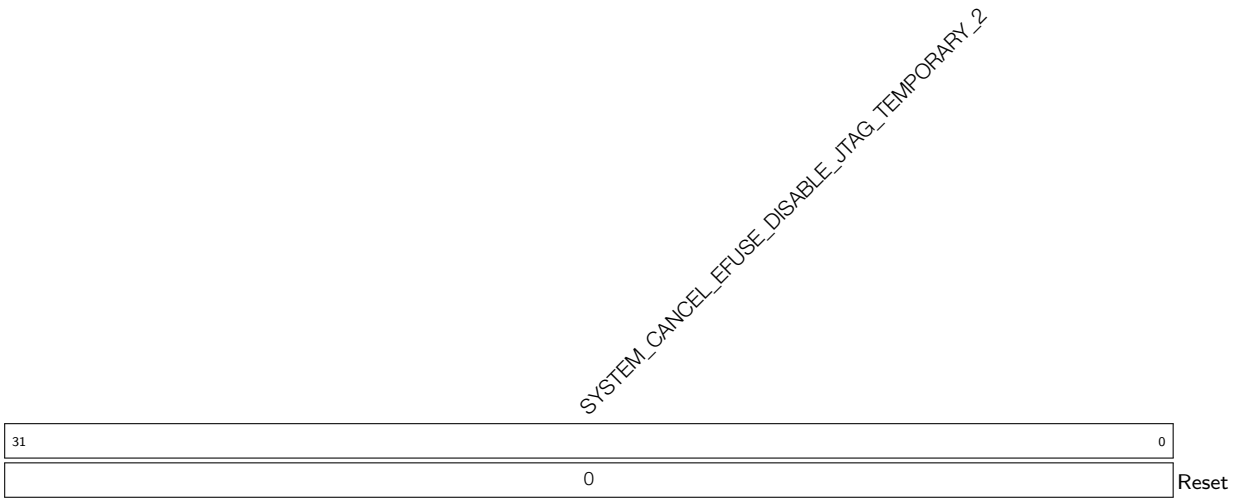
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_0 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 0 到 31 位。（只写）

Register 5.9: SYSTEM_JTAG_CTRL_1_REG (0x0020)



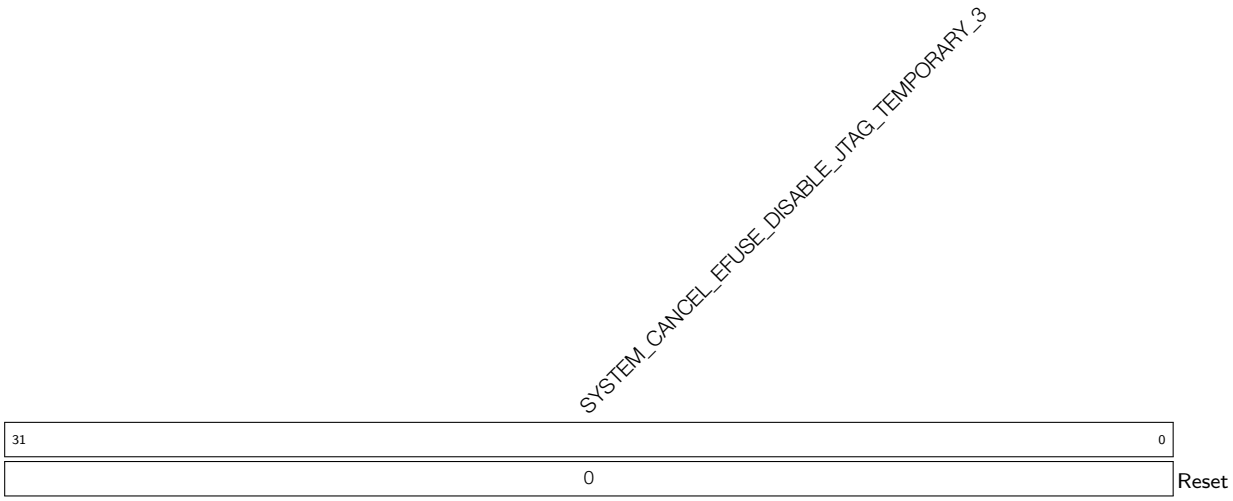
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_1 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 32 到 63 位。（只写）

Register 5.10: SYSTEM_JTAG_CTRL_2_REG (0x0024)



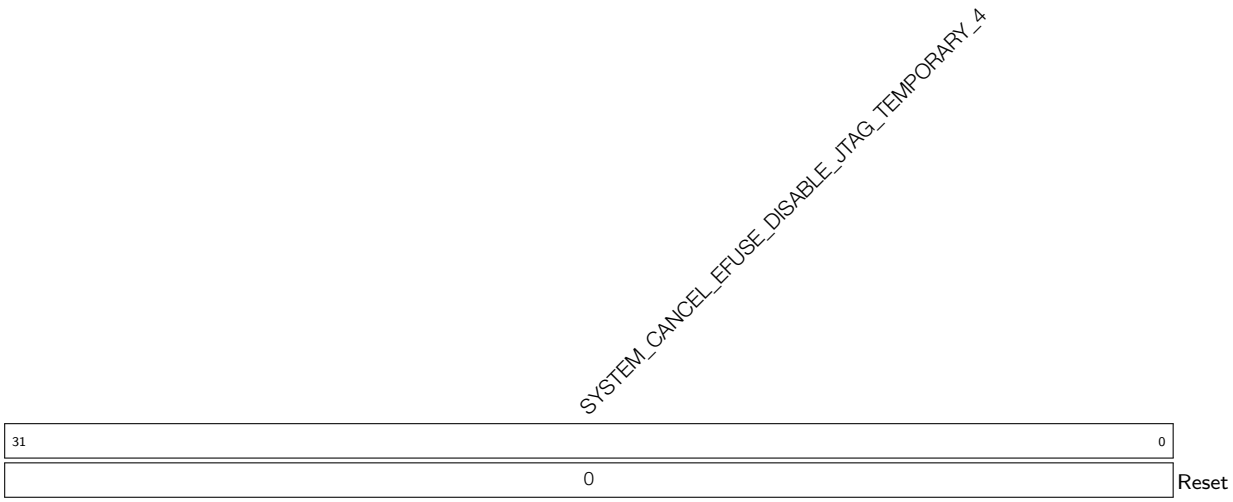
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_2 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 64 到 95 位。（只写）

Register 5.11: SYSTEM_JTAG_CTRL_3_REG (0x0028)



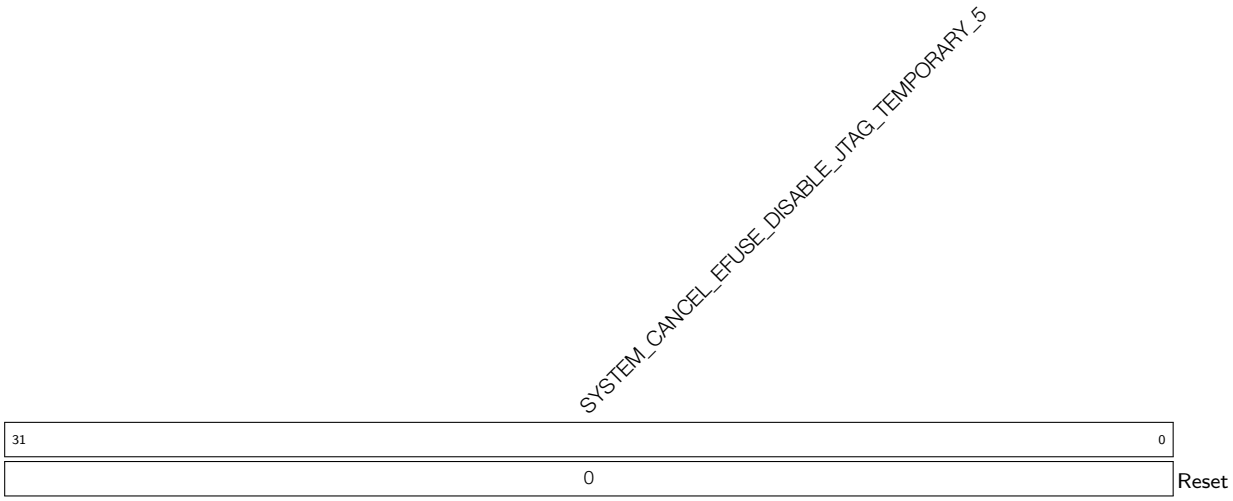
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_3 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 96 到 127 位。（只写）

Register 5.12: SYSTEM_JTAG_CTRL_4_REG (0x002C)



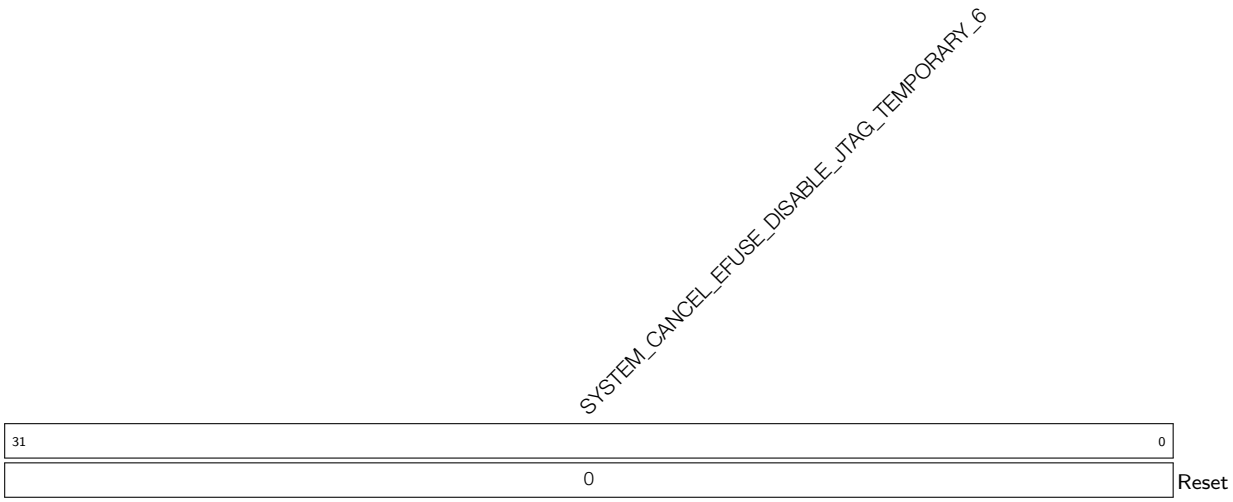
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_4 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 128 到 159 位。（只写）

Register 5.13: SYSTEM_JTAG_CTRL_5_REG (0x0030)



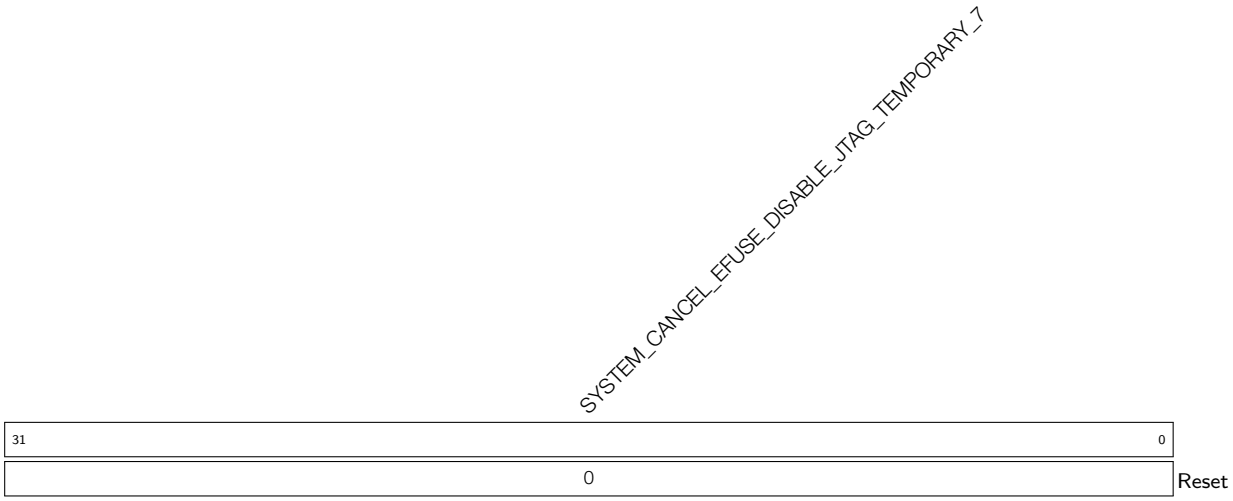
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_5 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 160 到 191 位。（只写）

Register 5.14: SYSTEM_JTAG_CTRL_6_REG (0x0034)



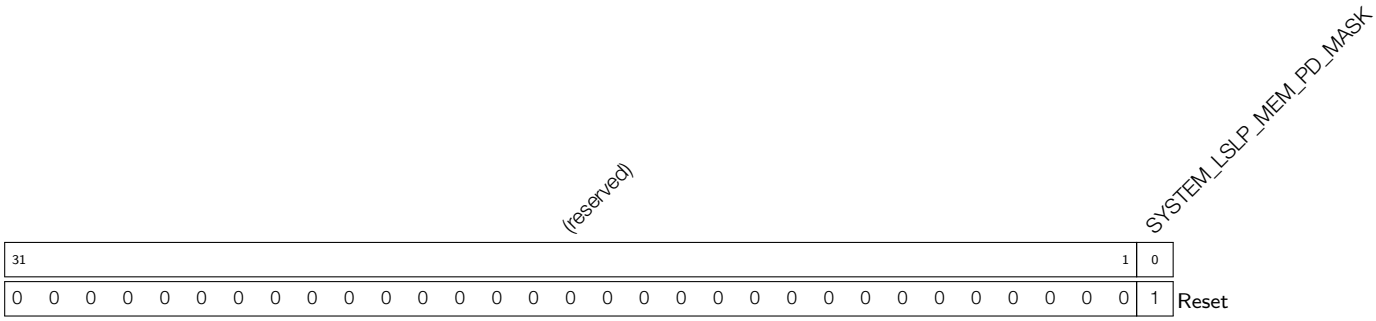
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_6 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 192 到 223 位。（只写）

Register 5.15: SYSTEM_JTAG_CTRL_7_REG (0x0038)



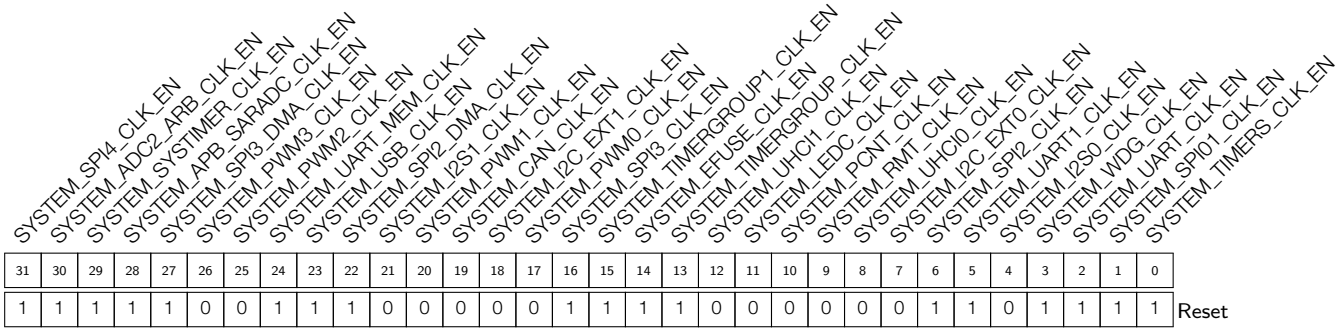
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_7 本组寄存器（共 256 位）用于撤销 eFuse 对 JTAG 的临时关闭，本寄存器为第 224 到 255 位。（只写）

Register 5.16: SYSTEM_MEM_PD_MASK_REG (0x003C)



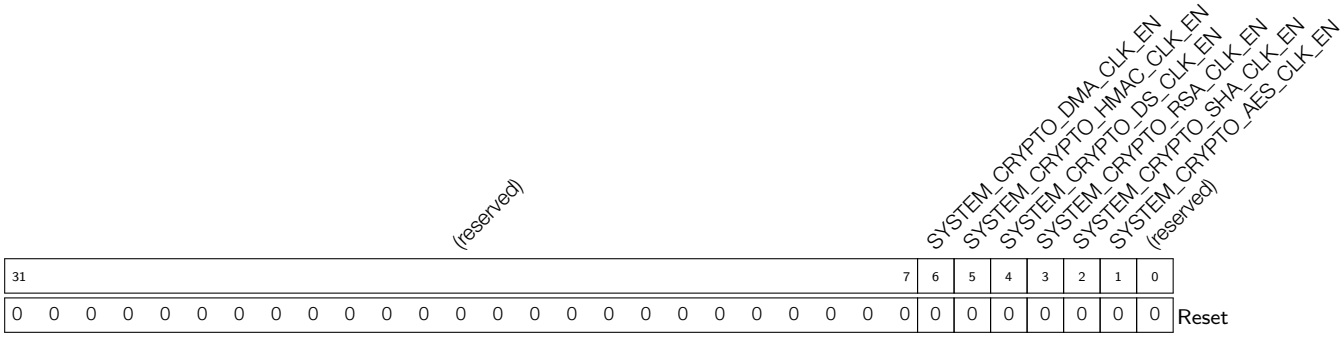
SYSTEM_LSLP_MEM_PD_MASK 置 1 允许存储器在 light-sleep 模式下仍正常工作。（读写）

Register 5.17: SYSTEM_PERIP_CLK_EN0_REG (0x0040)



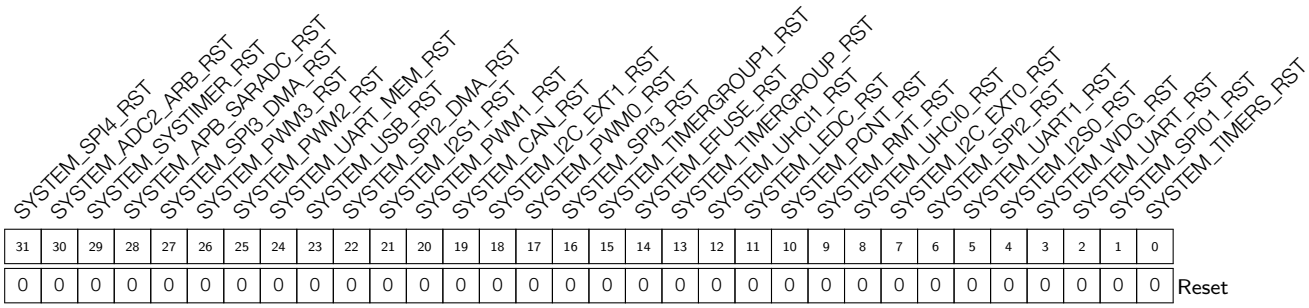
SYSTEM_CPU_PERI_CLK_EN0_REG 使能不同的外设时钟，详见表 5-3。

Register 5.18: SYSTEM_PERIP_CLK_EN1_REG (0x0044)



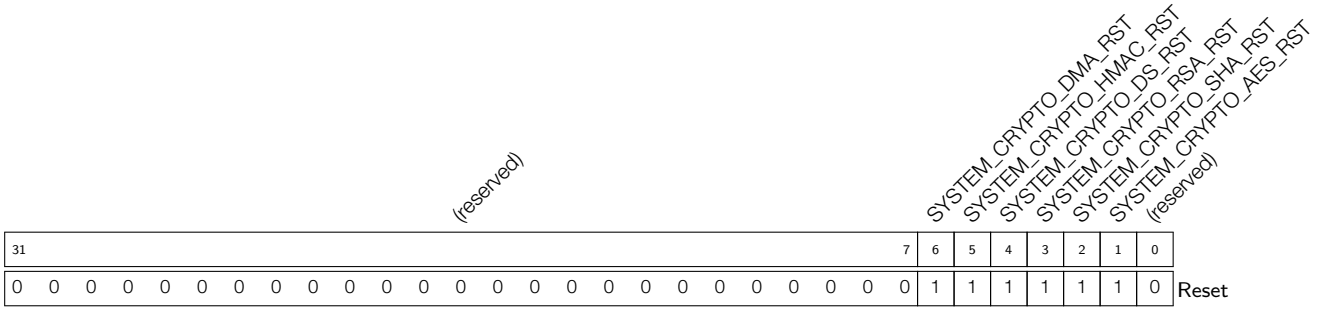
SYSTEM_PERIP_CLK_EN1_REG 使能不同的加速器时钟，详见表 5-3。

Register 5.19: SYSTEM_PERIP_RST_EN0_REG (0x0048)



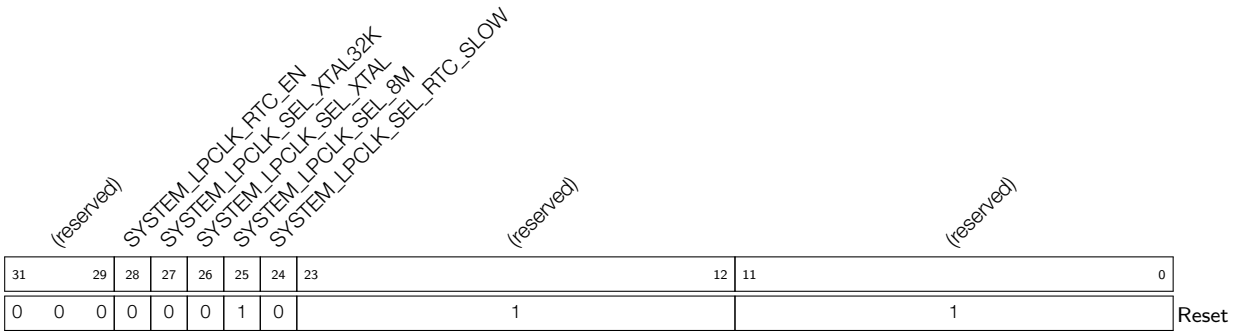
SYSTEM_PERIP_RST_EN0_REG 复位不同外设，详见表 5-3。

Register 5.20: SYSTEM_PERIP_RST_EN1_REG (0x004C)



SYSTEM_PERIP_RST_EN1_REG 复位不同加速器，详见表 5-3。

Register 5.21: 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)

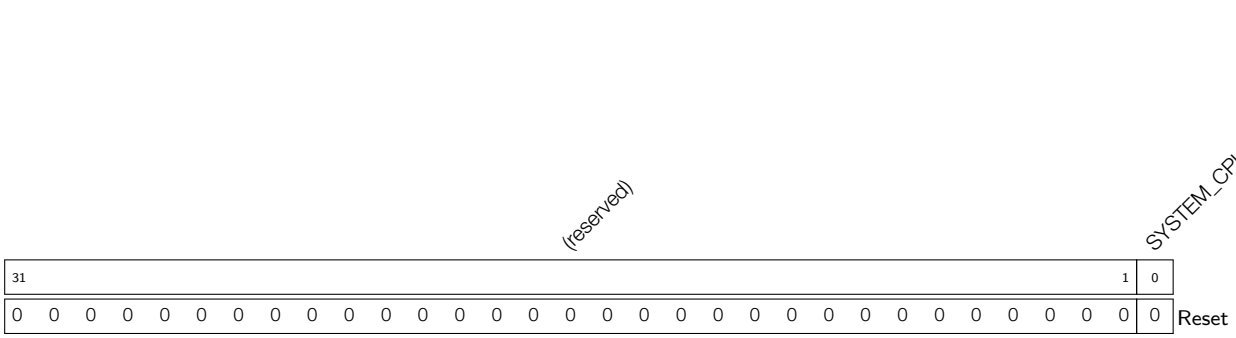
SYSTEM_LPCLK_SEL_8M 置 1 选择 8m 时钟为低功耗时钟。(读写)

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)

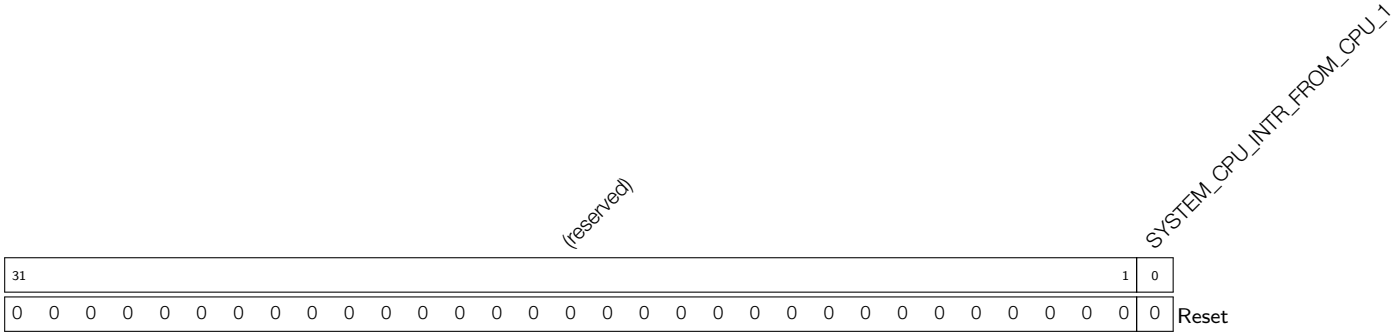
SYSTEM_LPCLK_RTC_EN 置 1 使能 RTC 低功耗时钟。(读写)

Register 5.22: SYSTEM_CPU_INTR_FROM_CPU_0_REG (0x0058)



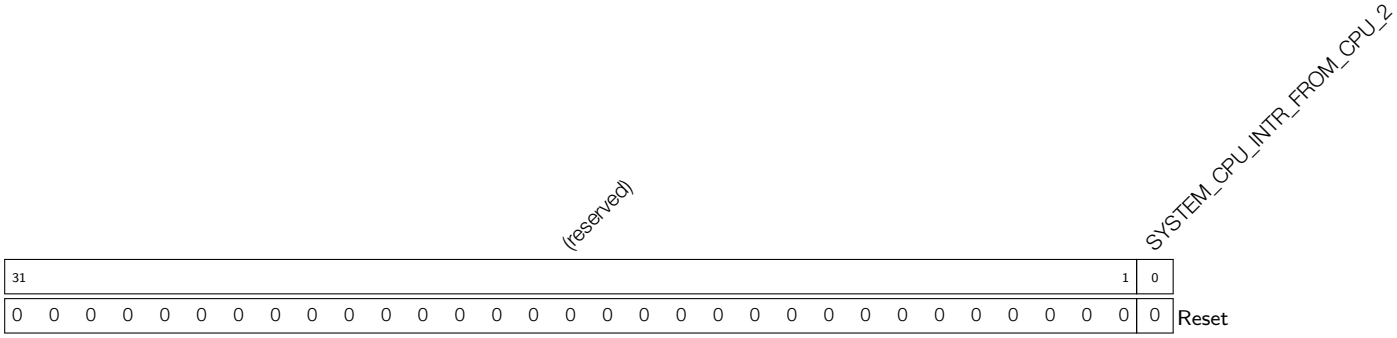
SYSTEM_CPU_INTR_FROM_CPU_0 置 1 生成 CPU 中断 0。该位需在 ISR 过程中由软件清 0。(读写)

Register 5.23: SYSTEM_CPU_INTR_FROM_CPU_1_REG (0x005C)



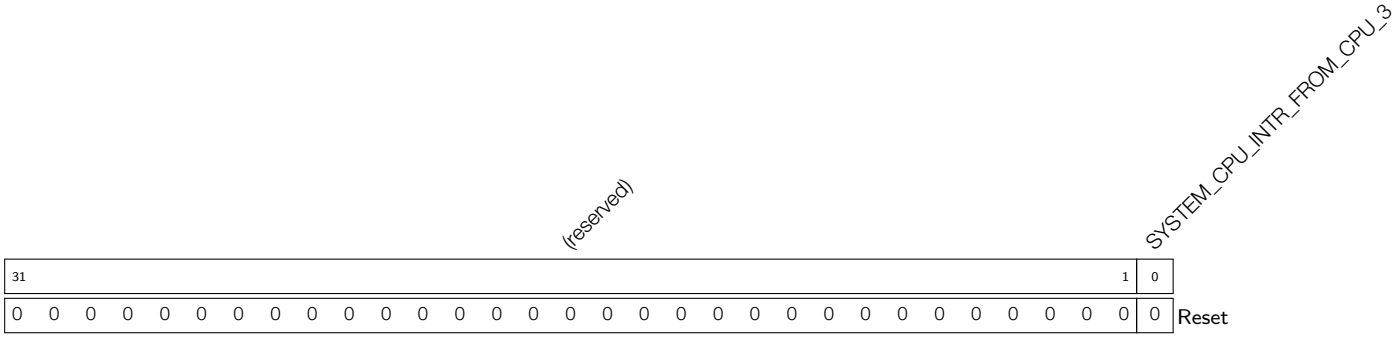
SYSTEM_CPU_INTR_FROM_CPU_1 置 1 生成 CPU 中断 1。该位需在 ISR 过程中由软件清 0。(读写)

Register 5.24: SYSTEM_CPU_INTR_FROM_CPU_2_REG (0x0060)



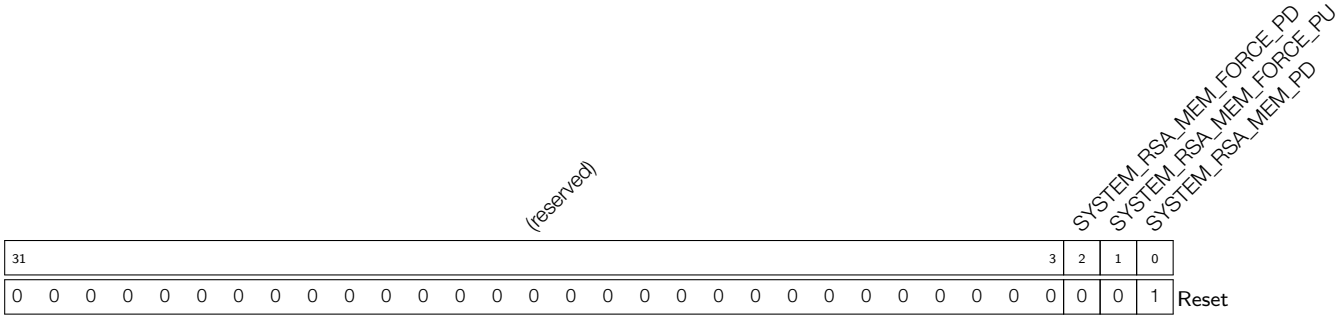
SYSTEM_CPU_INTR_FROM_CPU_2 置 1 生成 CPU 中断 2。该位需在 ISR 过程中由软件清 0。(读写)

Register 5.25: 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 5.26: SYSTEM_RSA_PD_CTRL_REG (0x0068)

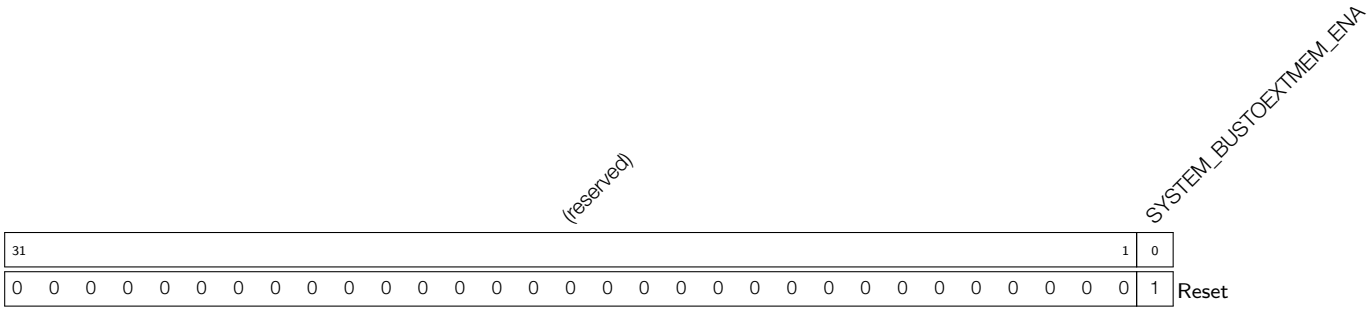


SYSTEM_RSA_MEM_PD 置 1 关闭 RSA 内存。该位优先级最低。当数字签名占用 RSA 加速器时，该位无效。（读写）

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

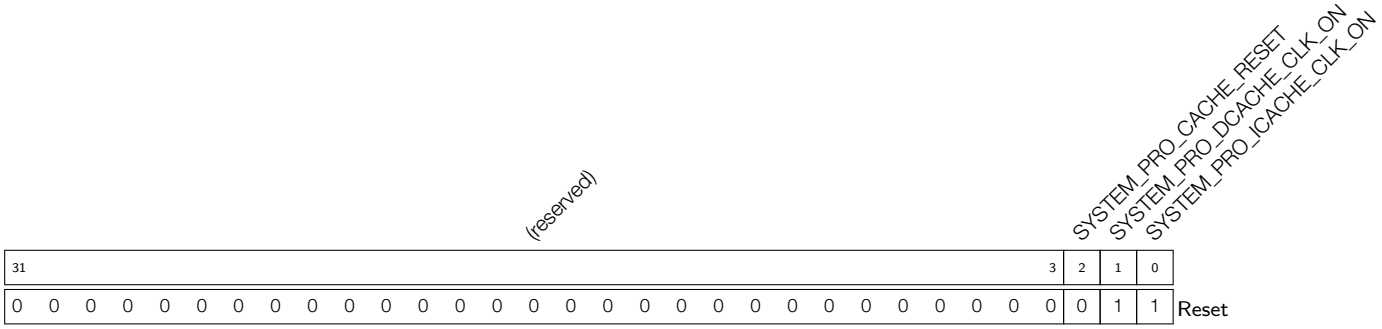
SYSTEM_RSA_MEM_FORCE_PD 置 1 关闭 RSA 内存。该位优先级最高。（读写）

Register 5.27: SYSTEM_BUSTOEXTMEM_ENA_REG (0x006C)



SYSTEM_BUSTOEXTMEM_ENA 置 1 使能 EDMA 总线。（读写）

Register 5.28: SYSTEM_CACHE_CONTROL_REG (0x0070)

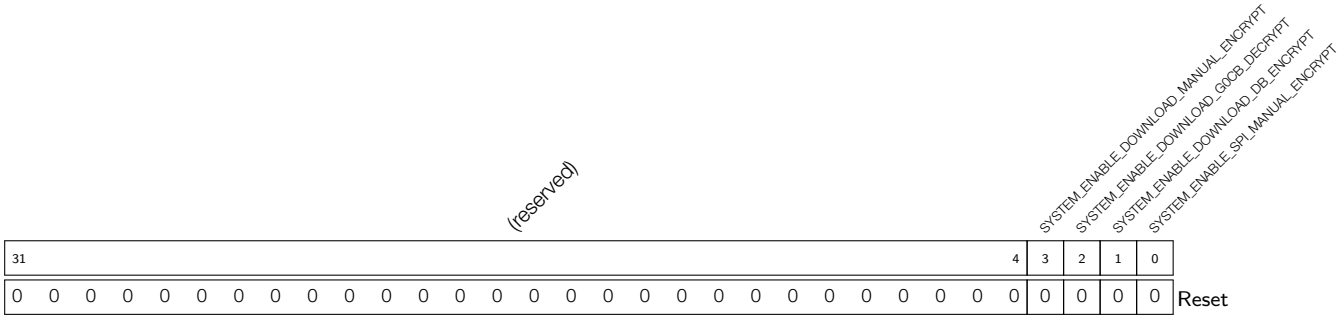


SYSTEM_PRO_ICACHE_CLK_ON 置 1 使能 i-cache 时钟。（读写）

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 5.29: SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG (0x0074)



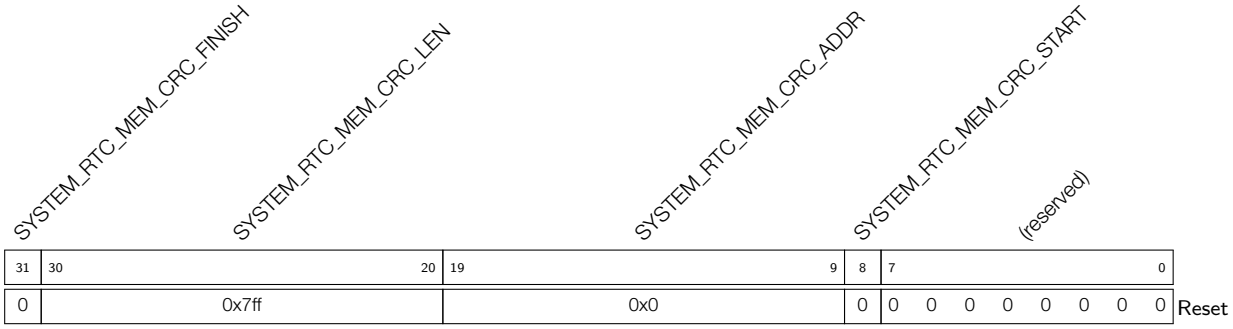
SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT 置 1 在 SPI Boot 模式下使能手动加密 (Manual Encryption)。(读写)

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 5.30: SYSTEM_RTC_FASTMEM_CONFIG_REG (0x0078)



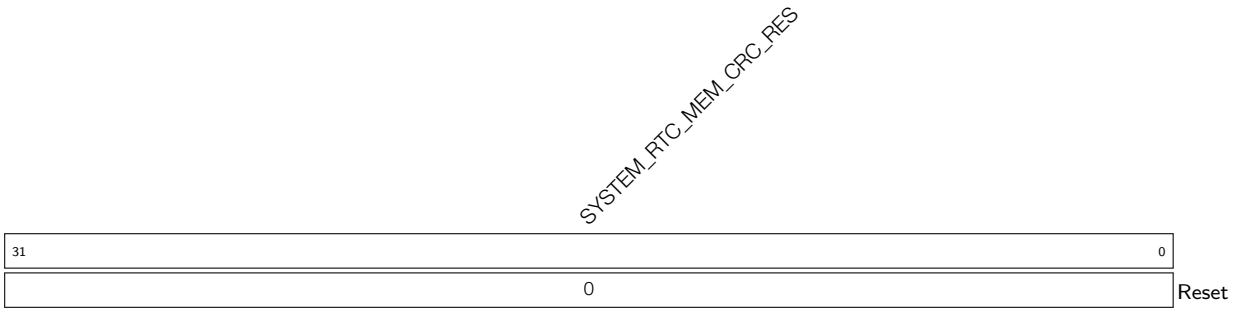
SYSTEM_RTC_MEM_CRC_START 置 1 启动 RTC 存储的 CRC 校验。(读写)

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)

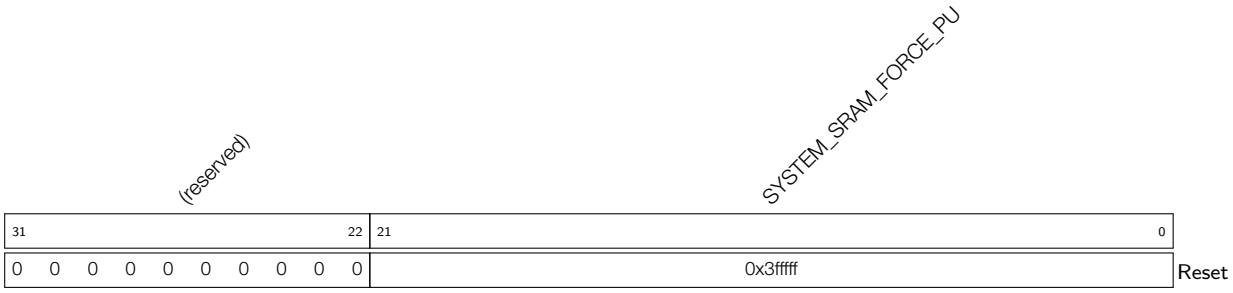
SYSTEM_RTC_MEM_CRC_FINISH 储存 RTC 存储 CRC 校验状态。高电平表示校验完成，低电平表示校验未完成。(只读)

Register 5.31: SYSTEM_RTC_FASTMEM_CRC_REG (0x007C)



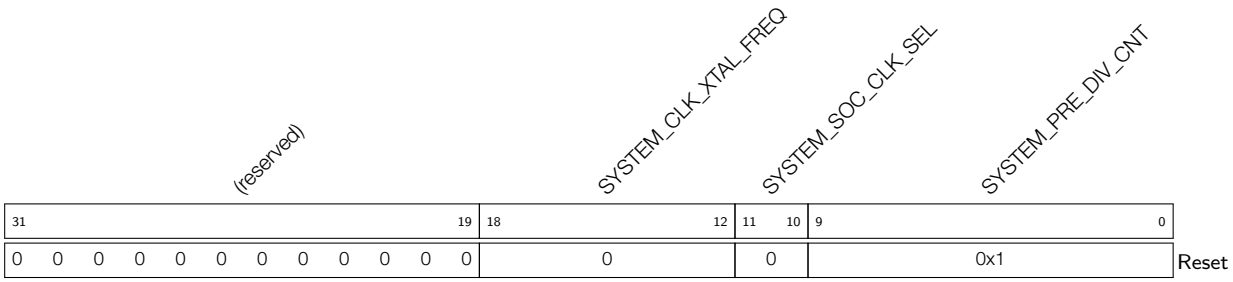
SYSTEM_RTC_MEM_CRC_RES 储存 RTC 存储的 CRC 校验结果。（只读）

Register 5.32: SYSTEM_SRAM_CTRL_2_REG (0x0088)



SYSTEM_SRAM_FORCE_PU 控制内部 SRAM 的上电。详见表 5-2。（读写）

Register 5.33: SYSTEM_SYSCLK_CONF_REG (0x008C)

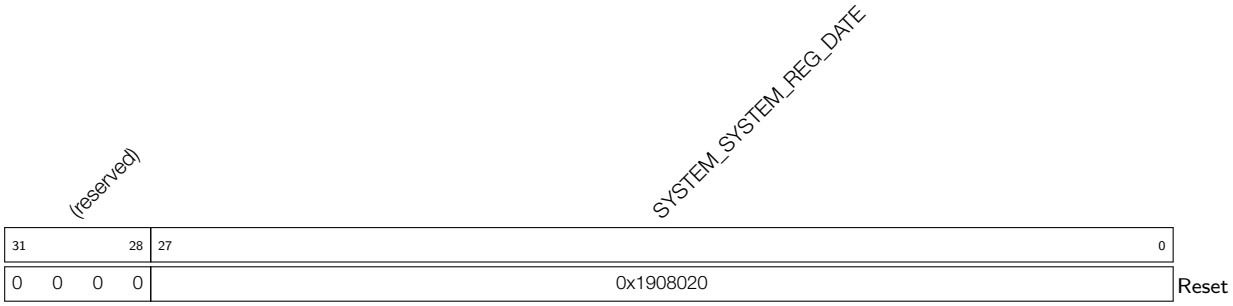


SYSTEM_PRE_DIV_CNT 设置预分频器计数器。具体配置，请见章节 2 复位和时钟中表 2-4。（读写）

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 2-2。（读写）

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

Register 5.34: SYSTEM_REG_DATE_REG (0x0FFC)



SYSTEM_DATE 版本控制寄存器。（读写）

6. LED PWM 控制器

6.1 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。

6.2 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*。

6.3 功能描述

6.3.1 架构

图 6-1为 LED PWM 控制器的架构。图 6-2 为一个 PWM 生成器及其选取的定时器和计数器。

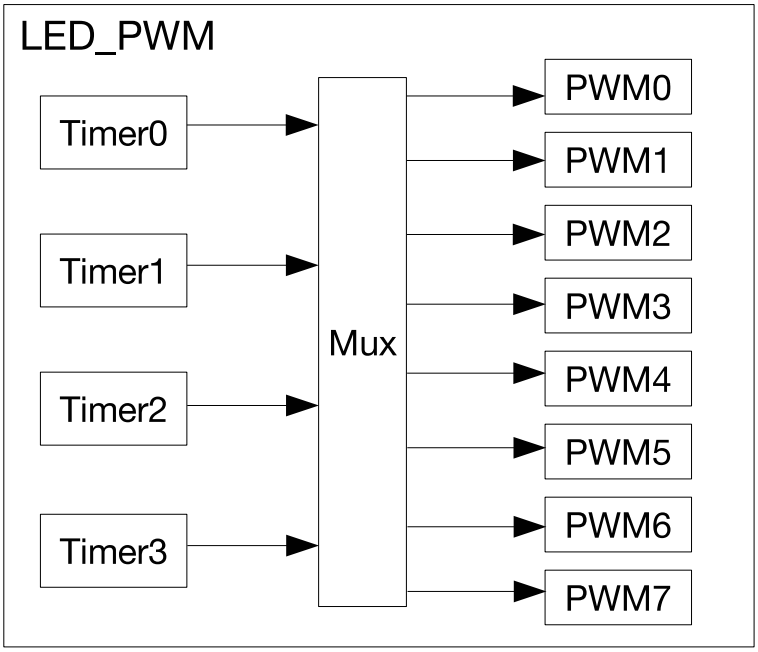


图 6-1. LED_PWM 架构

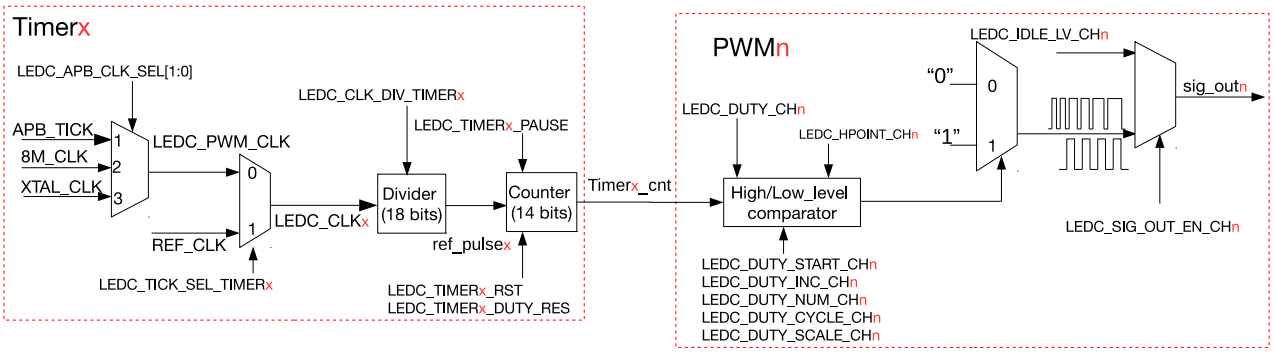


图 6-2. LED_PWM 生成器图

6.3.2 定时器

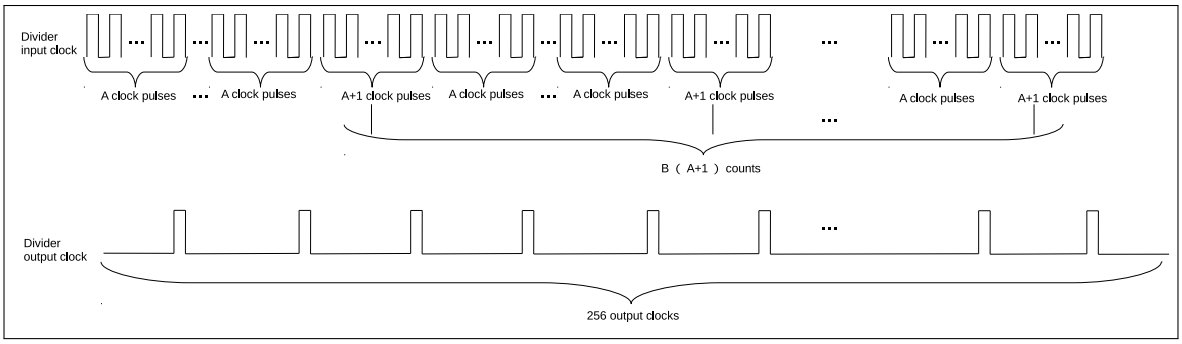


图 6-3. LED_PWM 分频器

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 6-3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* 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. 配置溢出次数 LEDC_OVF_NUM_CH*n*
3. 置位LEDC_OVF_CNT_CH*n*_INT_ENA

4. 置位 LEDC_TIMERx_DUTY_RES 启动定时器，等待 LEDC_OVF_CNT_CHn_INT 中断

PWM 生成器输出信号 sig_outn 的频率取决于分频器的分频系数以及计数器的计数范围：

$$f_{sig\_outn} = \frac{f_{LEDC\_CLKx}}{LEDC\_CLK\_DIVx \cdot 2^{LEDC\_TIMERx\_DUTY\_RES}}$$

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_TIMERx 和 LEDC_TIMERx_DUTY_RES，然后置位 LEDC_TIMERx_PARA_UP，否则配置无效。新的配置在计数器溢出时更新。

6.3.3 PWM 生成器

如图 6-2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 Hpointn 和 Lpointn 比较，来控制 PWM 信号的电平。

- 如果 Timerx_cnt == Hpointn，则 sig_outn 为 1。
- 如果 Timerx_cnt == Lpointn，则 sig_outn 为 0。

Hpointn 由 LEDC_HPOINT_CHn 在计数器溢出时更新。Lpointn 的初始值为计数器溢出时 LEDC_DUTY_CHn[18..4] 和 LEDC_HPOINT_CHn 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 6-4 为占空比固定时的 PWM 波形。

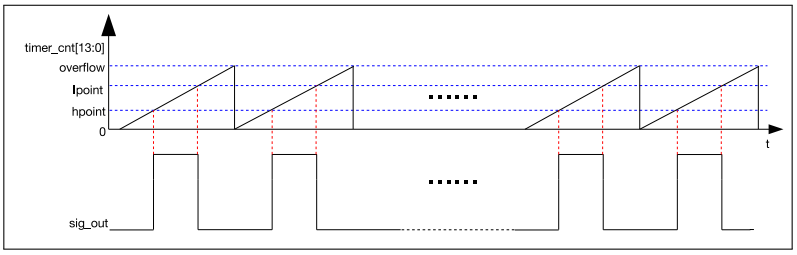


图 6-4. LED_PWM 输出信号图

LEDC_DUTY_CHn 是一个具有四位小数的定点寄存器。LEDC_DUTY_CHn[18..4] 为整数部分，可直接用于 PWM 计算。LEDC_DUTY_CHn[3..0] 为小数部分，可用于调整输出信号。如 LEDC_DUTY_CHn[3..0] 不为 0，那么 sig_outn 每 16 个周期中，有 LEDC_DUTY_CHn[3..0] 个周期的 PWM 脉冲宽度要比 (16 - LEDC_DUTY_CHn[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。

6.3.4 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_CHn、LEDC_DUTY_START_CHn、LEDC_DUTY_INC_CHn 、LEDC_DUTY_NUM_CHn 和 LEDC_DUTY_SCALE_CHn 配置。

LEDC_DUTY_START_CHn 用于更新 Lpointn 的值。如置位该位，计数器溢出时 Lpointn 会自动递增或递减。
LEDC_DUTY_INC_CHn 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 6-5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*。

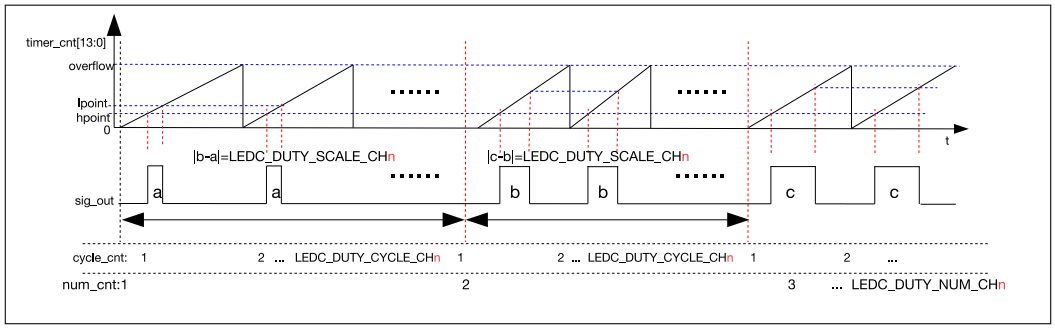


图 6-5. 渐变占空比输出信号图

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。

6.3.5 中断

- LEDC_OVF_CNT_CH*n*_INT：定时器计数器溢出 LEDC_OVF_NUM_CH*n* 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。

6.4 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 6-1 所示。更多信息，请访问章节 1 *系统和存储器*。

表 6-1. LED_PWM 基地址

访问总线	基地址
PeriBUS1	0x3F419000
PeriBUS2	0x60019000

6.5 寄存器列表

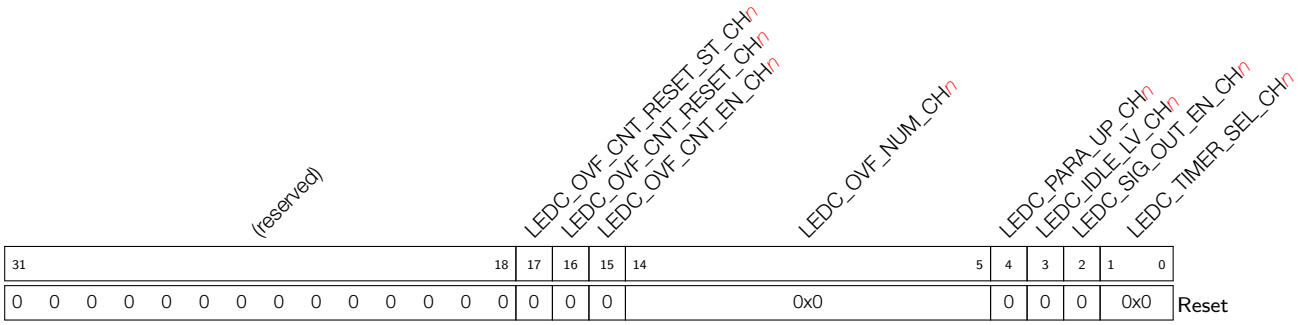
请注意，下表的地址指相对于 LED PWM 基地址的偏移量（相对地址）。请参阅章节 6.4 获取有关 LED PWM 基地址的信息。请注意，下表中的地址都是相对于 LED PWM 基地址的地址偏移量（相对地址）。更多有关 LED PWM 基地址的信息，请前往 6.4 章节。

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

名称	描述	地址	访问
LEDC_CH4_DUTY_R_REG	通道 4 的当前占空比	0x0060	只读
LEDC_CH5_DUTY_REG	通道 5 的初始占空比	0x006C	读/写
LEDC_CH5_DUTY_R_REG	通道 5 的当前占空比	0x0074	只读
LEDC_CH6_DUTY_REG	通道 6 的初始占空比	0x0080	读/写
LEDC_CH6_DUTY_R_REG	通道 6 的当前占空比	0x0088	只读
LEDC_CH7_DUTY_REG	通道 7 的初始占空比	0x0094	读/写
LEDC_CH7_DUTY_R_REG	通道 7 的当前占空比	0x009C	只读
定时器寄存器			
LEDC_TIMER0_CONF_REG	定时器 0 配置	0x00A0	不定
LEDC_TIMER0_VALUE_REG	定时器 0 的当前计数器值	0x00A4	只读
LEDC_TIMER1_CONF_REG	定时器 1 配置	0x00A8	不定
LEDC_TIMER1_VALUE_REG	定时器 1 的当前计数器值	0x00AC	只读
LEDC_TIMER2_CONF_REG	定时器 2 配置	0x00B0	不定
LEDC_TIMER2_VALUE_REG	定时器 2 的当前计数器值	0x00B4	只读
LEDC_TIMER3_CONF_REG	定时器 3 配置	0x00B8	不定
LEDC_TIMER3_VALUE_REG	定时器 3 的当前计数器值	0x00BC	只读
中断寄存器			
LEDC_INT_RAW_REG	原始中断状态	0x00C0	只读
LEDC_INT_ST_REG	屏蔽中断状态	0x00C4	只读
LEDC_INT_ENA_REG	中断使能位	0x00C8	读/写
LEDC_INT_CLR_REG	中断清除位	0x00CC	只写
版本寄存器			
LEDC_DATE_REG	版本控制寄存器	0x00FC	读/写

6.6 寄存器

Register 6.1: 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)

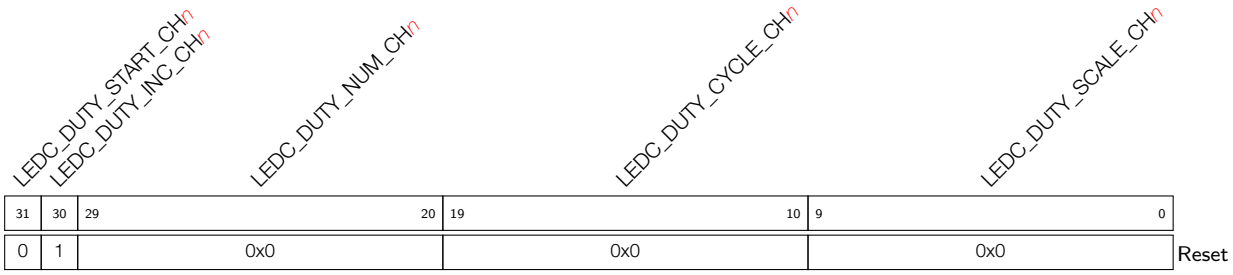
LEDC_OVF_NUM_CH*n* 用于配置 ovf_cnt 的最大值。通道 *n* 的 ovf_cnt 达到 LEDC_OVF_CNT_CH*n*_INT 寄存器指定值时触发 LEDC_OVF_CNT_CH*n*_INT 中断。(读/写)

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 6.2: 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)

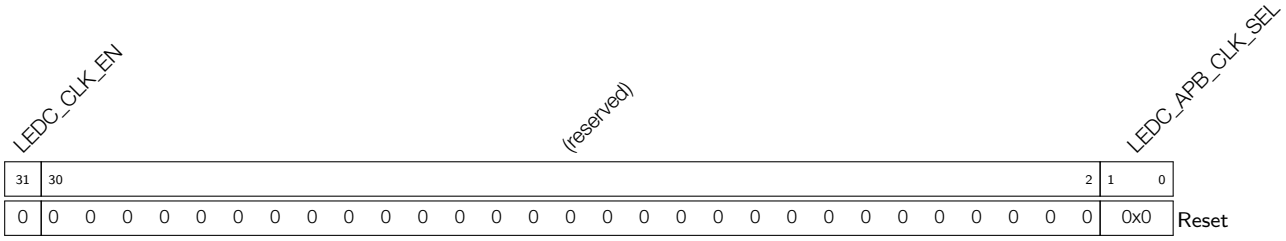
LEDC_DUTY_CYCLE_CH*n* 通道 *n* 占空比每隔 LEDC_DUTY_CYCLE_CH*n* 变化一次。(读/写)

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)

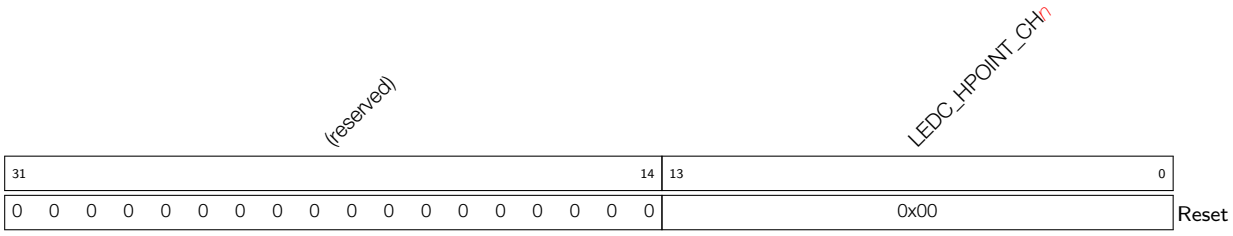
Register 6.3: LEDC_CONF_REG (0x00D0)



LEDC_APB_CLK_SEL 用于设置 4 个定时器 SLOW_CLK 的频率。1：APB_CLK；2：RTC8M_CLK；3：XTAL_CLK。(读/写)

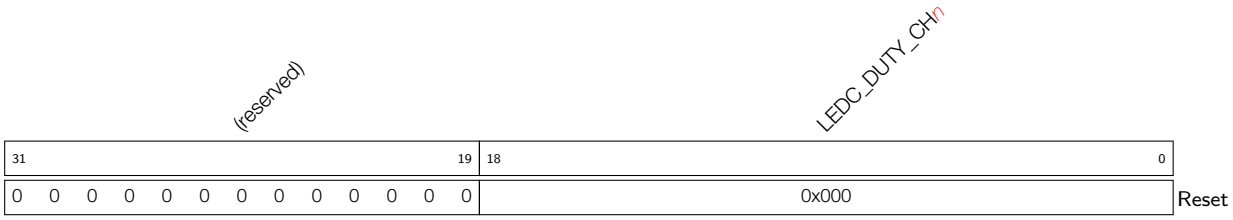
LEDC_CLK_EN 用于控制时钟。1'b1：强制开启寄存器时钟。1'b0：仅在应用写寄存器时支持时钟。(读/写)

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



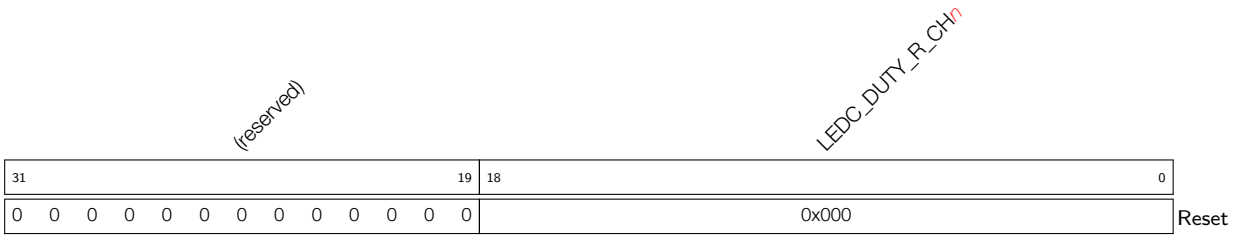
LEDC_HPOINT_CH*n* 所选定时器计数值达到该值时，输出信号翻转为高电平。(读/写)

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



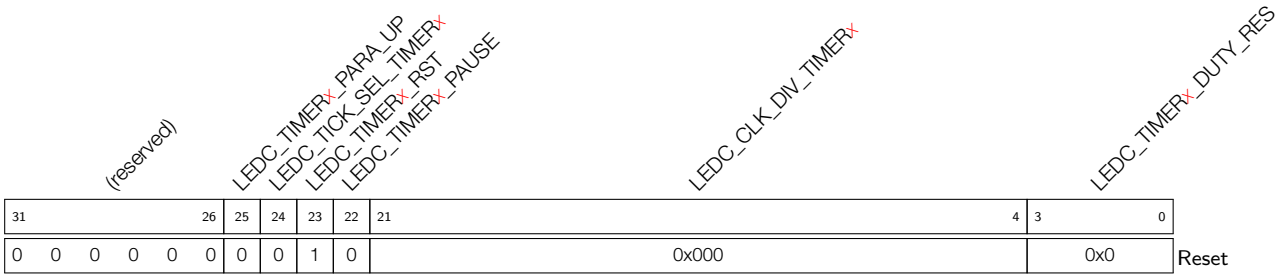
LEDC_DUTY_CH*n* 通过控制低位点改变输出信号占空比。所选定时器达到低位点时，输出信号翻转为低电平。(读/写)

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



LEDC_DUTY_R_CH*n* 存储通道 *n* 输出信号的当前占空比。(只读)

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



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)

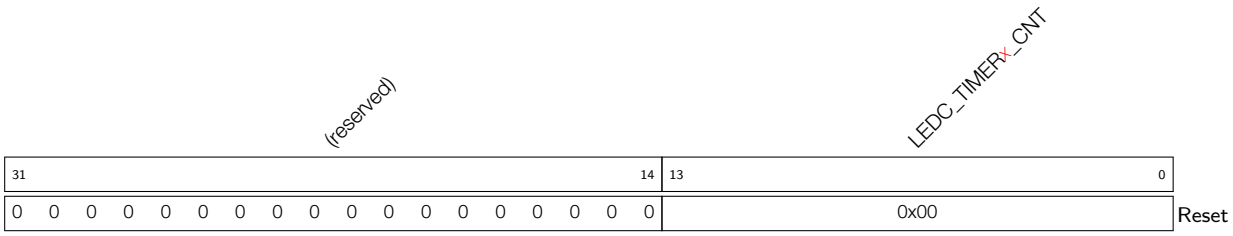
LEDC_TIMER*x*_PAUSE 用于暂停定时器 *x* 的计数器。(读/写)

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)

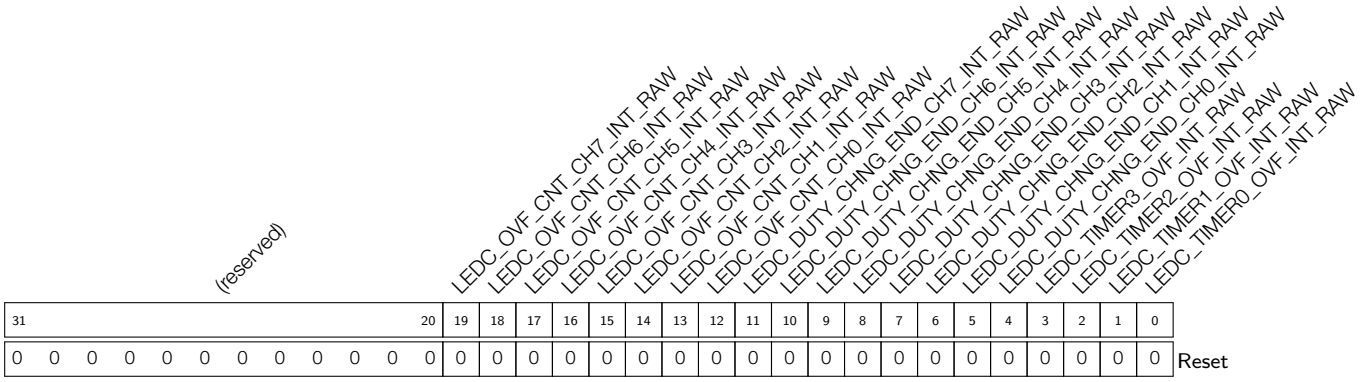
LEDC_TIMER*x*_PARA_UP 置位此位，更新 LEDC_CLK_DIV_TIMER*x* 和 LEDC_TIMER*x*_DUTY_RES。(只写)

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



LEDC_TIMER*x*_CNT 存储定时器 *x* 的当前计数器值 (只读)

Register 6.9: LEDC_INT_RAW_REG (0x00C0)

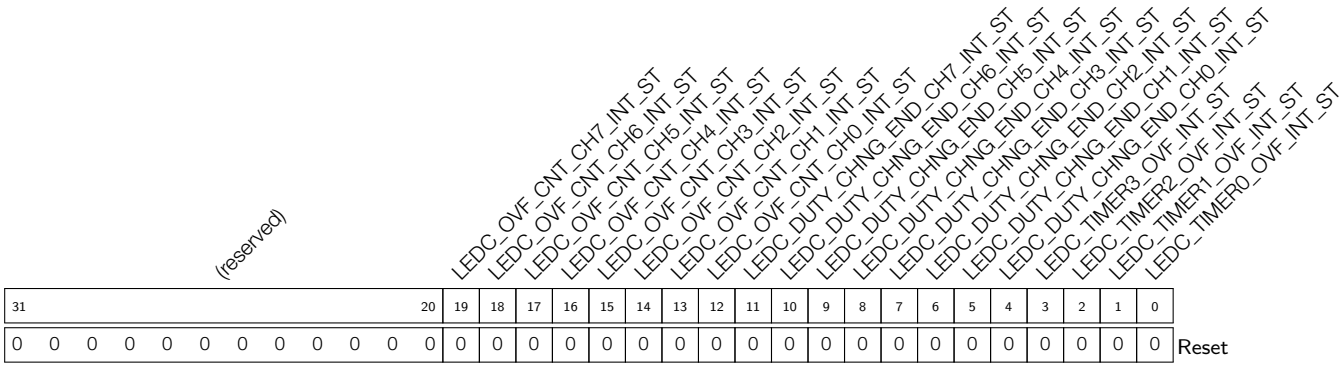


LEDC_TIMER*x*_OVF_INT_RAW 定时器 *x* 达到最大计数值时触发中断。(只读)

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 6.10: LEDC_INT_ST_REG (0x00C4)

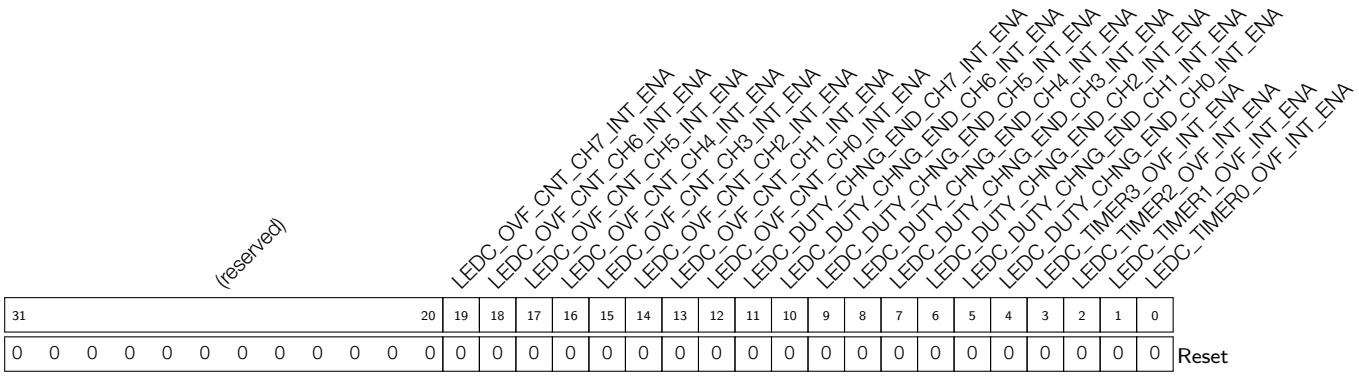


LEDC_TIMER*x*_OVF_INT_ST LEDC_TIMER*x*_OVF_INT_ENA 置 1 时，LEDC_TIMER*x*_OVF_INT 中断的屏蔽中断状态位。(只读)

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 6.11: LEDC_INT_ENA_REG (0x00C8)

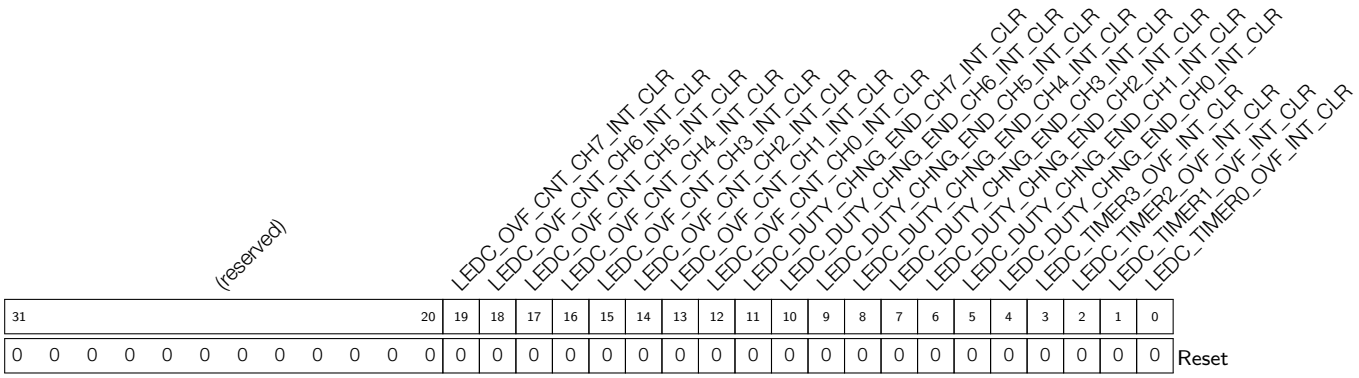


LEDC_TIMER*x*_OVF_INT_ENA LEDC_TIMER*x*_OVF_INT 中断的使能位。(读/写)

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 6.12: LEDC_INT_CLR_REG (0x00CC)

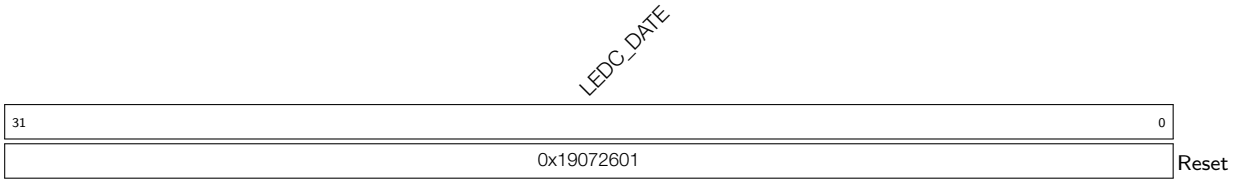


LEDC_TIMER*x*_OVF_INT_CLR 置位此位，清除 LEDC_TIMER*x*_OVF_INT 中断。(只写)

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 6.13: LEDC_DATE_REG (0x00FC)



LEDC_DATE 版本控制寄存器。(读/写)

7. 红外遥控

7.1 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。

7.2 功能描述

7.2.1 RMT 架构

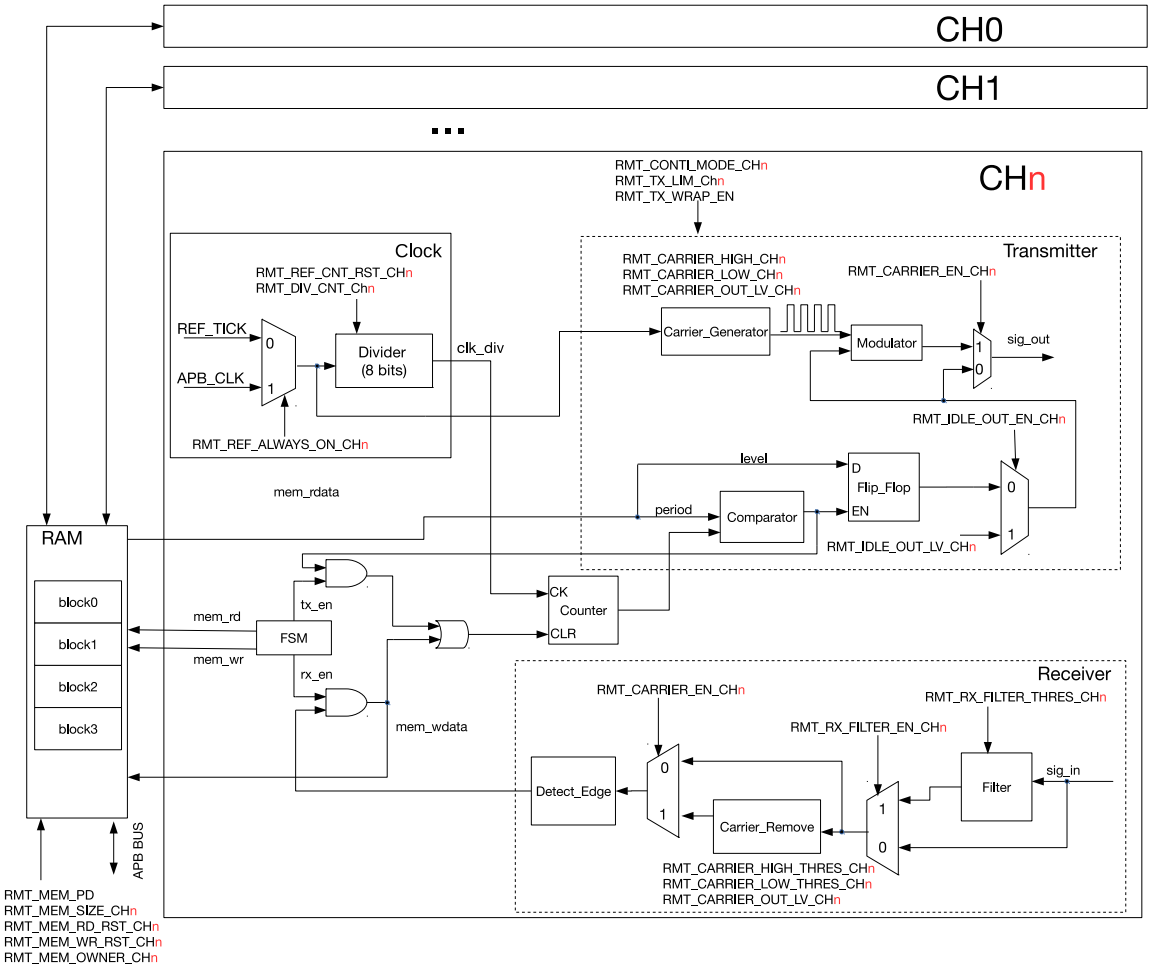


图 7-1. RMT 结构框图

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。

7.2.2 RMT RAM

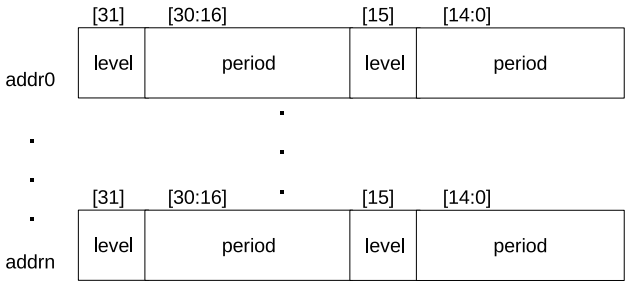


图 7-2. RAM 中脉冲编码结构

RAM 中脉冲编码结构如图 7-2 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7-1 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。

7.2.3 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。

7.2.4 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。

7.2.5 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。

7.2.6 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。

7.3 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 7-1 所示。更多信息，请访问章节 1 系统和存储器。

表 7-1. RMT 基地址

访问总线	基地址
PeriBUS1	0x3F416000
PeriBUS2	0x60016000

7.4 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 7.3 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

名称	描述	地址	访问
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
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

7.5 寄存器

Register 7.1: 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)

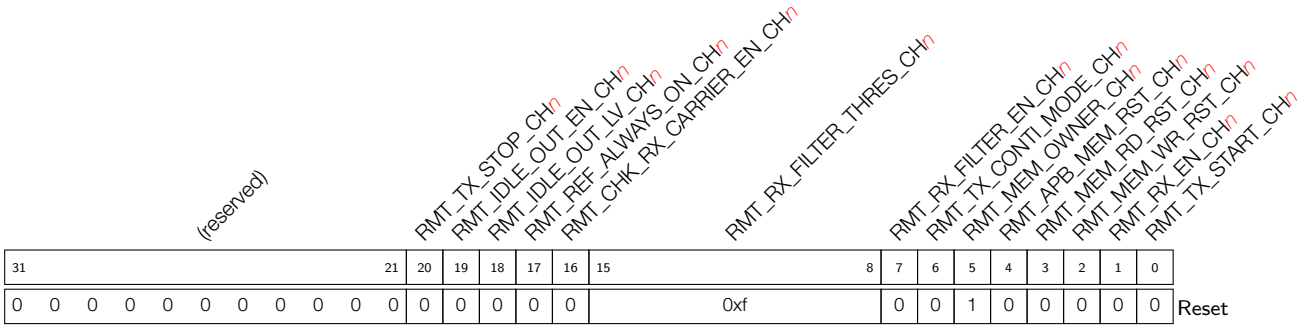
RMT_MEM_SIZE_CH*n* 配置通道 *n* 可以使用的 block 数量，范围为 1 ~ 4-*n*。(读/写)

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 7.2: 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)

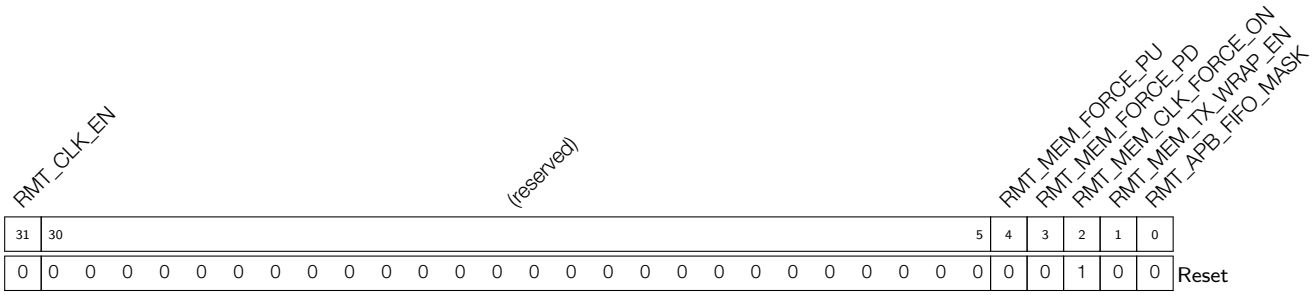
RMT_REF_ALWAYS_ON_CH*n* 用于选择通道 *n* 的基础时钟。1'h1：APB_CLK；1'h0：REF_TICK (读/写)

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 7.3: 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）

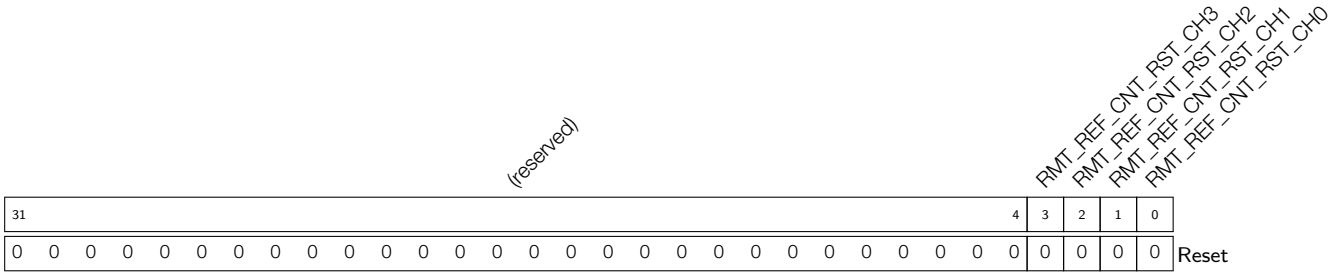
RMT_MEM_CLK_FORCE_ON RMT 模块工作时，使能 RAM 的时钟；RMT 模块不工作时，关闭 RAM 的时钟，从而实现低功耗。（读/写）

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 7.4: RMT_REF_CNT_RST_REG (0x0088)



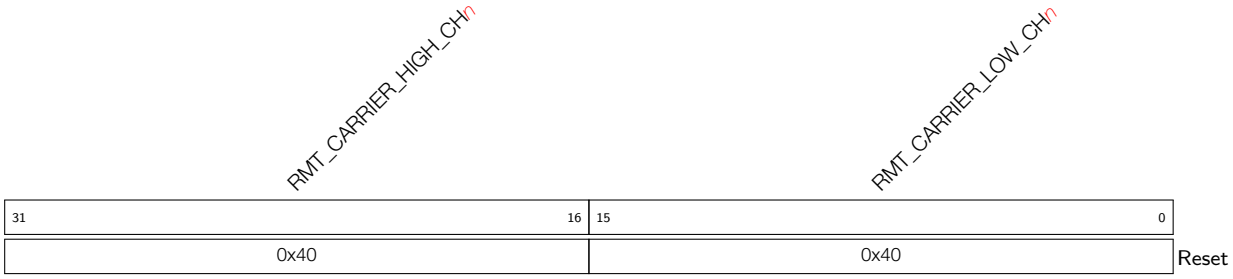
RMT_REF_CNT_RST_CH*n* 复位通道 *n* 的时钟分频器。（读/写）

Register 7.5: 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)

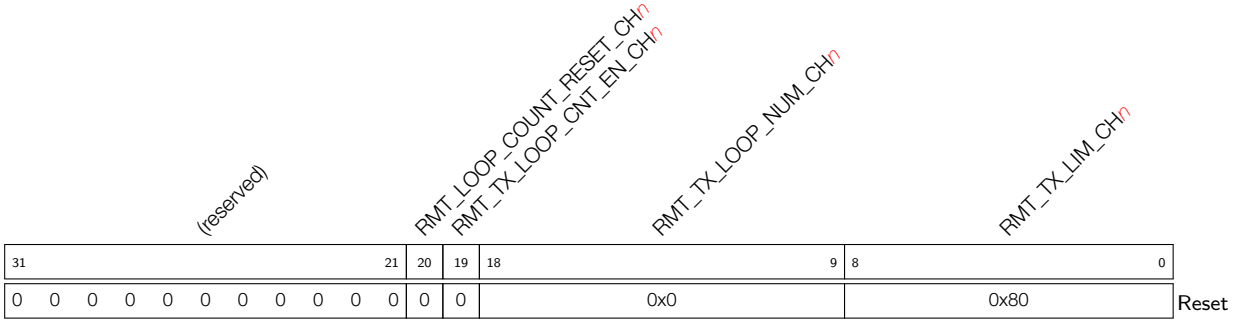
Register 7.6: RMT_CH*n*CARRIER_DUTY_REG (*n*: 0-3) (0x0060+4*n)



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



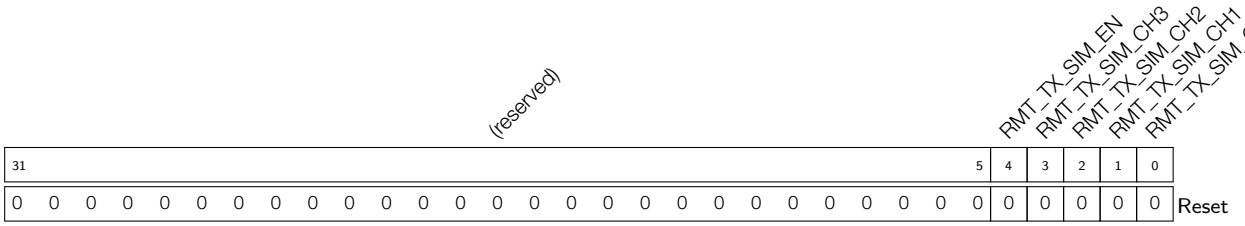
RMT_TX_LIM_CH*n* 用于配置通道 *n* 发送脉冲编码数量的上限值。当 RMT_MEM_SIZE_CH*n* = 1 时，此寄存器可设置为 0 ~ 128 (64 * 32 /16 = 128) 之间的值；当 RMT_MEM_SIZE_CH*n* > 1 时，此寄存器可设置为 (0 ~ 128) * RMT_MEM_SIZE_CH*n* 之间的值。（读/写）

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）

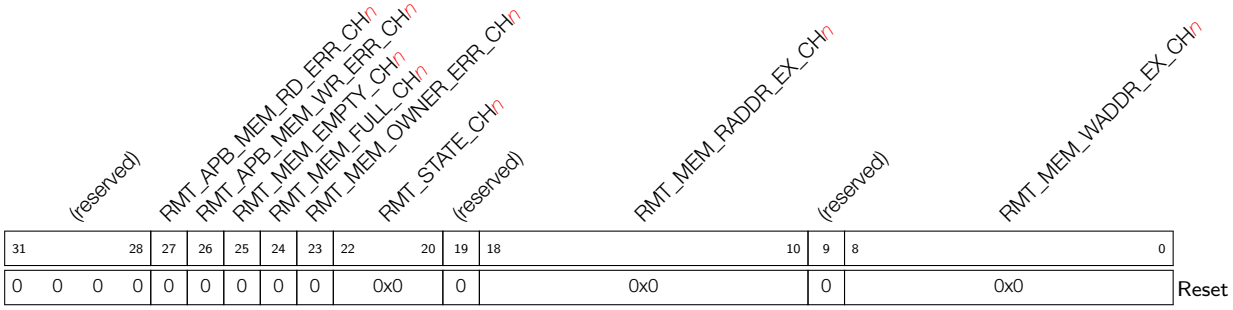
Register 7.8: RMT_TX_SIM_REG (0x0084)



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 7.9: 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)

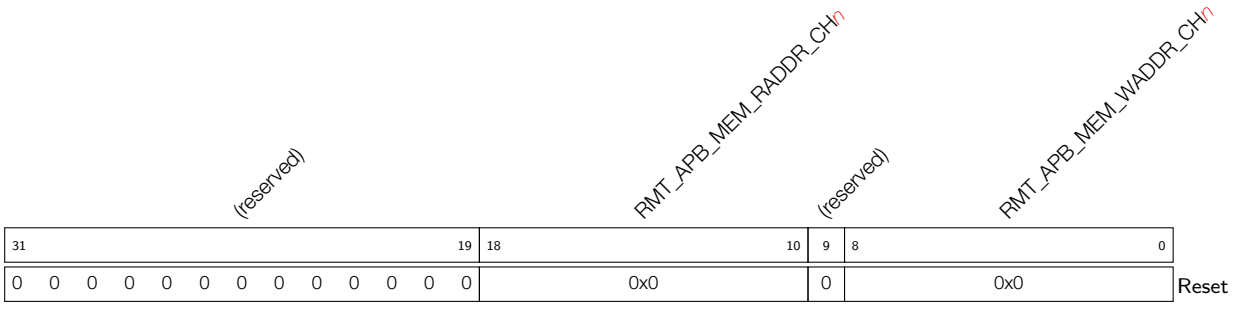
RMT_MEM_FULL_CH*n* 接收器接收的数据量大于 RAM block 时，将触发此状态位。(只读)

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)

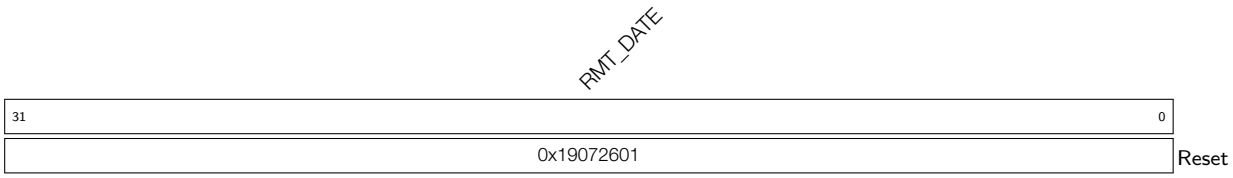
Register 7.10: RMT_CH*n*ADDR_REG (*n*: 0-3) (0x0040+4n*)**



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)

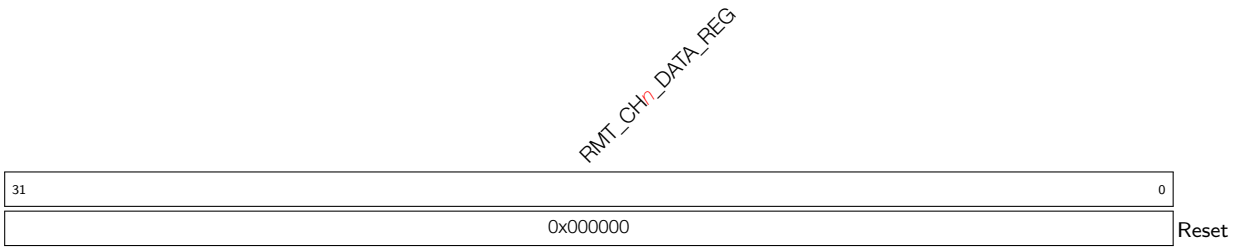
RMT_APB_MEM_RADDR_CH*n* 记录通过 APB 总线对 RAM 执行读操作时，地址的偏移量。(只读)

Register 7.11: RMT_DATE_REG (0x00FC)



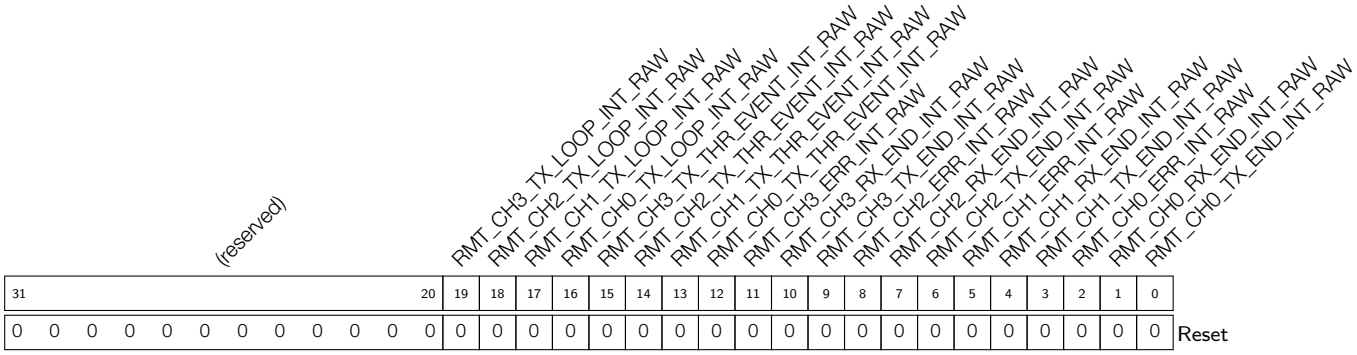
RMT_DATE 版本控制寄存器（读/写）

Register 7.12: 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 7.13: 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）

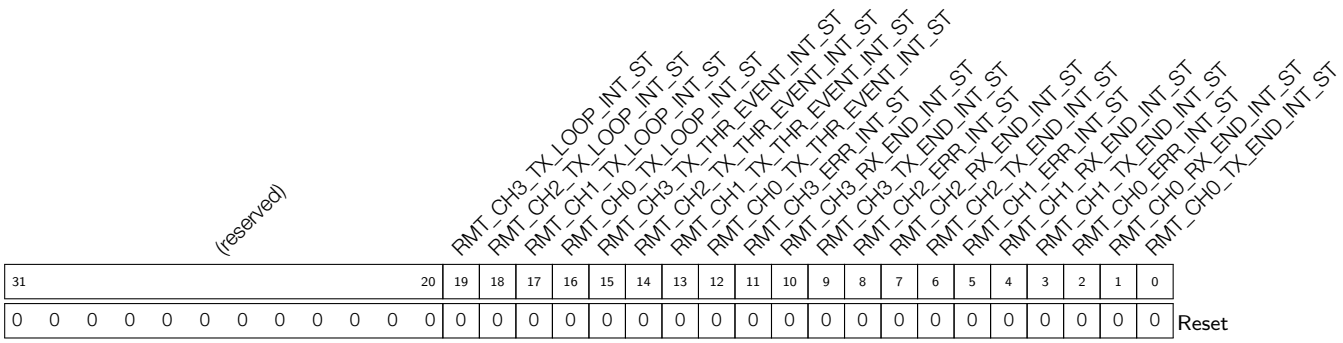
RMT_CH*n*_RX_END_INT_RAW 通道 *n* 的原始中断状态位。接收结束即触发该中断。（只读）

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 7.14: 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)

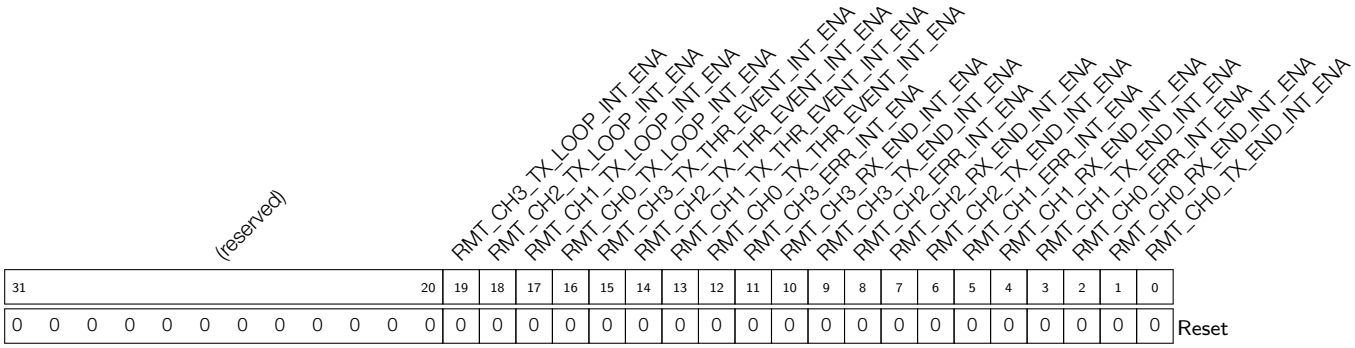
RMT_CH*n*_RX_END_INT_ST RMT_CH*n*_RX_END_INT 的隐蔽中断状态位。(只读)

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 7.15: 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)

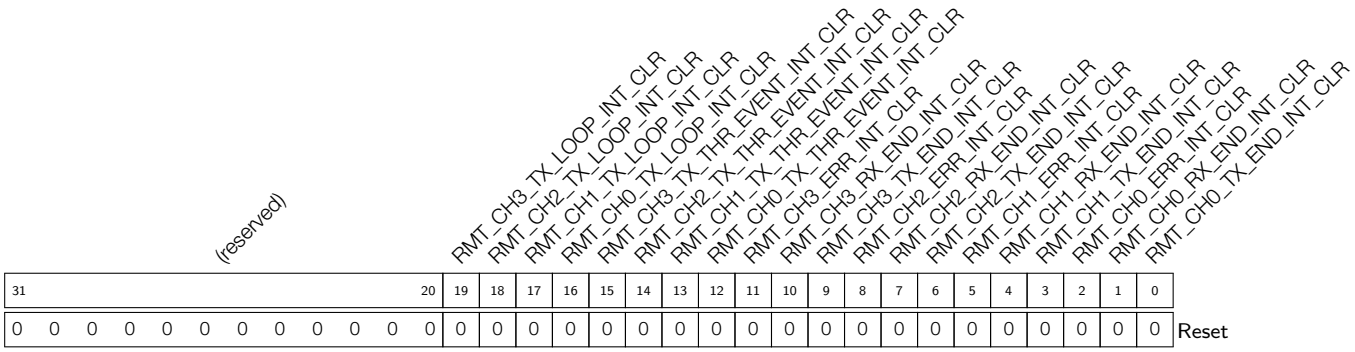
RMT_CH*n*_RX_END_INT_ENA RMT_CH*n*_RX_END_INT 中断使能位。(读/写)

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 7.16: 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)

8. 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 8-1 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）

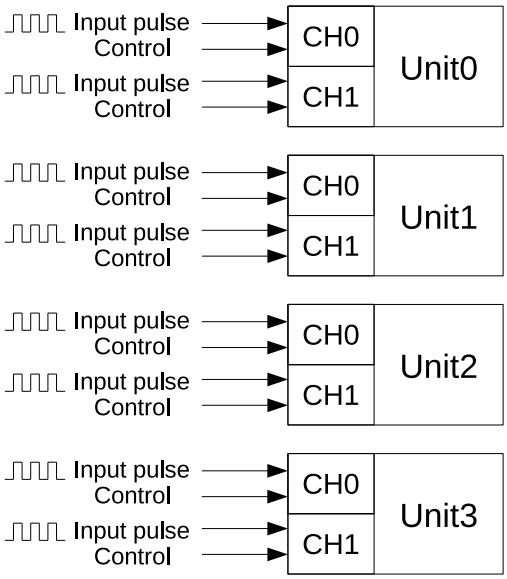


图 8-1. PCNT 框图

8.1 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。

8.2 功能描述

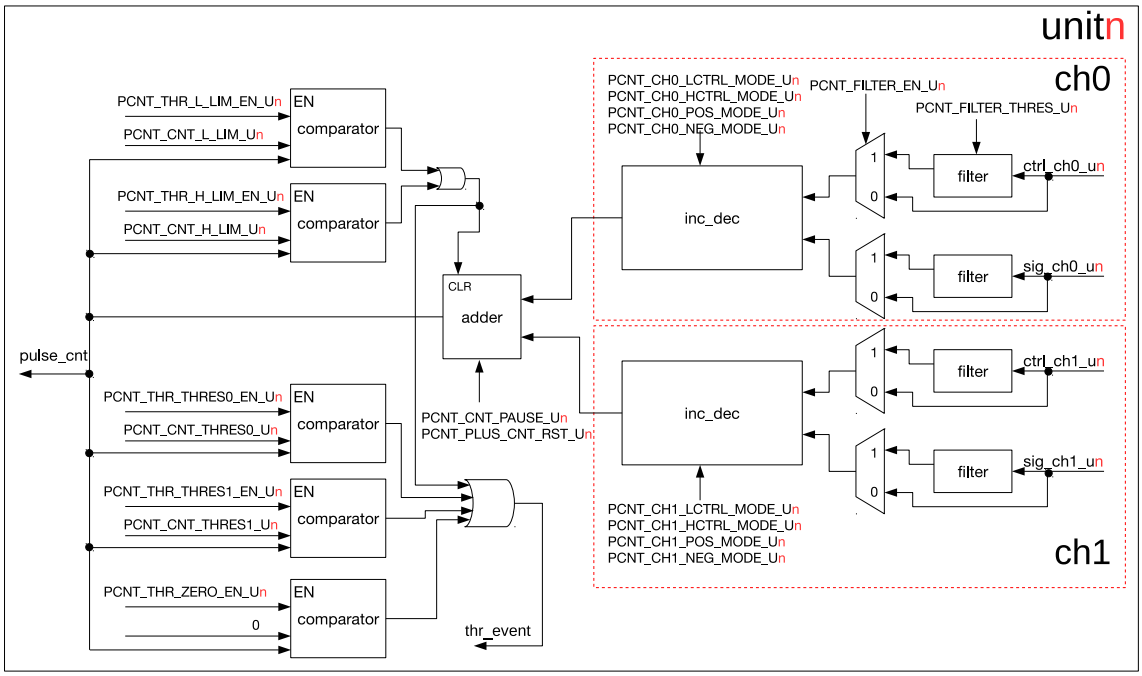


图 8-2. PCNT 单元基本架构图

图 8-2 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 8-1 至表 8-4 说明了如何配置通道 0 的计数模式。

表 8-1. 控制信号为低电平时输入脉冲信号上升沿的计数模式

PCNT_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 8-2. 控制信号为高电平时输入脉冲信号上升沿的计数模式

PCNT_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 8-3. 控制信号为低电平时输入脉冲信号下降沿的计数模式

PCNT_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 8-4. 控制信号为高电平时输入脉冲信号下降沿的计数模式

PCNT_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。

8.3 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。

8.3.1 通道 0 独自递增计数

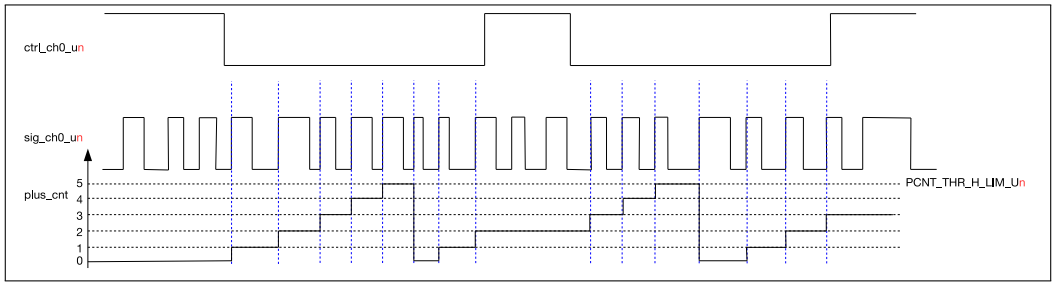


图 8-3. 通道 0 递增计数图

图 8-3 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 8.2 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。

8.3.2 通道 0 独自递减计数

图 8-4 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 8-3 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。

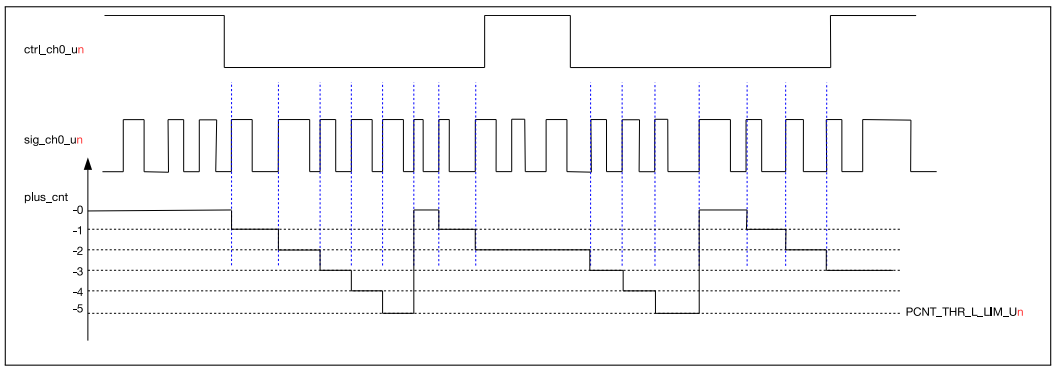


图 8-4. 通道 0 递减计数图

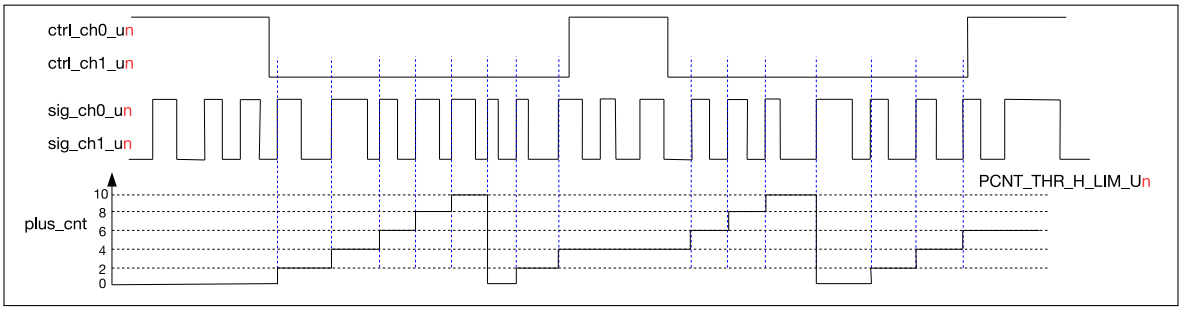


图 8-5. 双通道递增计数图

8.3.3 通道 0 和通道 1 同时递增计数

图 8-5 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 8-5 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。

8.4 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 8-5 所示。更多信息，请访问章节 1 系统和存储器。

表 8-5. PCNT 基地址

访问总线	基地址
PeriBUS1	0x3F417000
PeriBUS2	0x60017000

8.5 寄存器列表

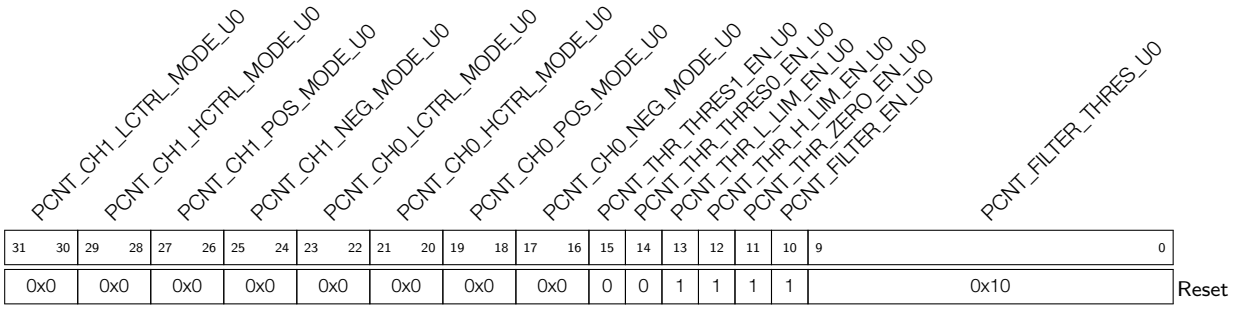
请注意，下表中的地址都是相对于 PCNT 基地址的地址偏移量（相对地址）。更多有关 PCNT 基地址的信息，请前往 8.4 章节。

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

名称	描述	地址	访问
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

8.6 寄存器

Register 8.1: 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)

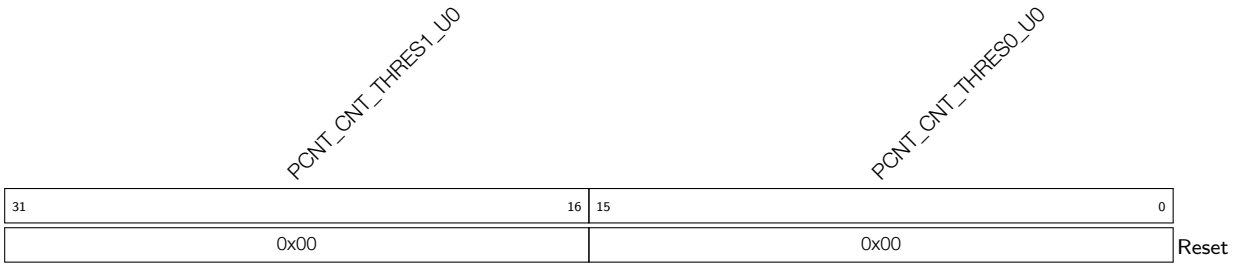
PCNT_CH1_NEG_MODE_U*n* 用于设置通道 1 输入信号检测下降沿的工作模式。1：计数器递增；2：计数器递减；0、3：对计数器无任何影响。(读/写)

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)

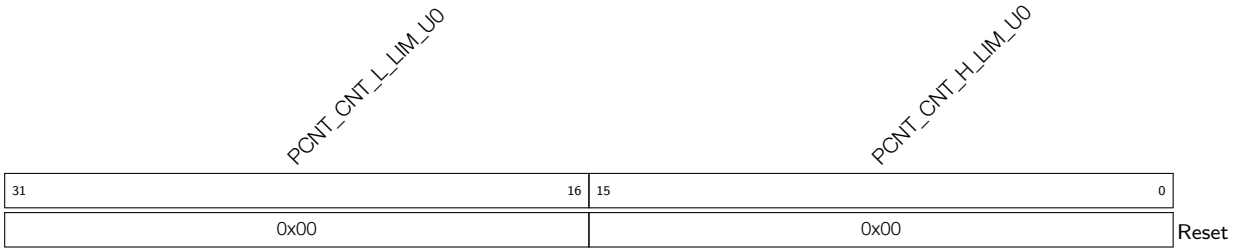
Register 8.2: PCNT_U*n*_CONF1_REG (*n*: 0-3) (0x0004+12**n*)



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)

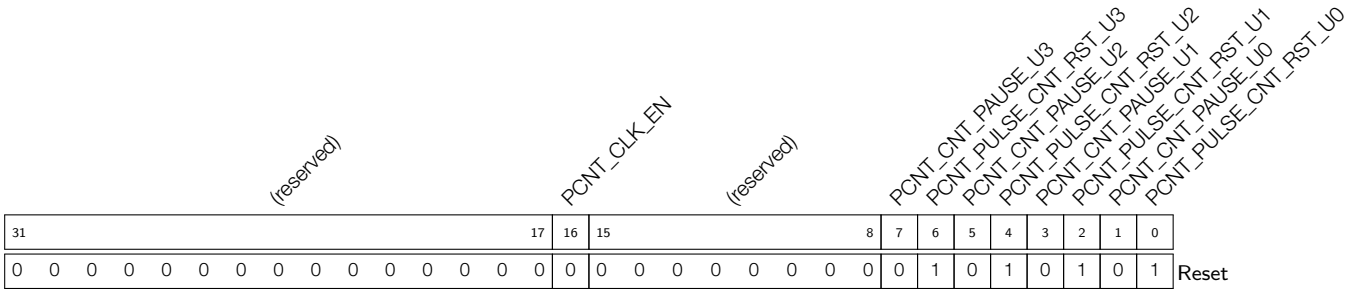
Register 8.3: PCNT_U*n*_CONF2_REG (*n*: 0-3) (0x0008+12**n*)



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 8.4: PCNT_CTRL_REG (0x0060)

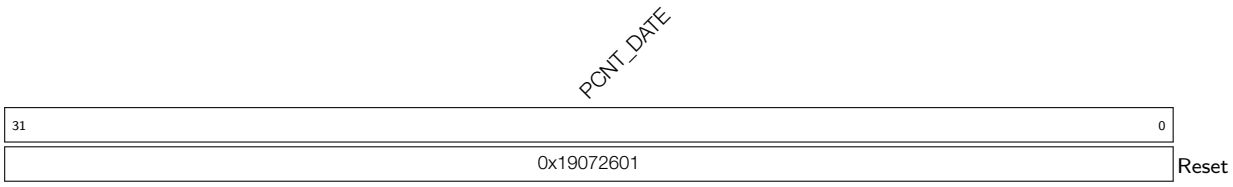


PCNT_PULSE_CNT_RST_U*n* 置位此位，清零单元 *n* 的计数器。(读/写)

PCNT_CNT_PAUSE_U*n* 置位此位，暂停单元 *n* 的计数器。(读/写)

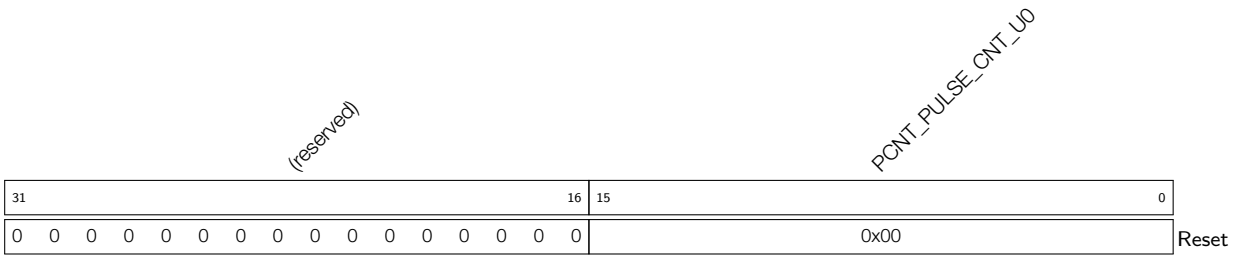
PCNT_CLK_EN 脉冲计数器模块寄存器时钟门控的使能信号 1：寄存器可通过应用读取、写值。0：寄存器无法通过应用读取、写值。(读/写)

Register 8.5: PCNT_DATE_REG (0x00FC)



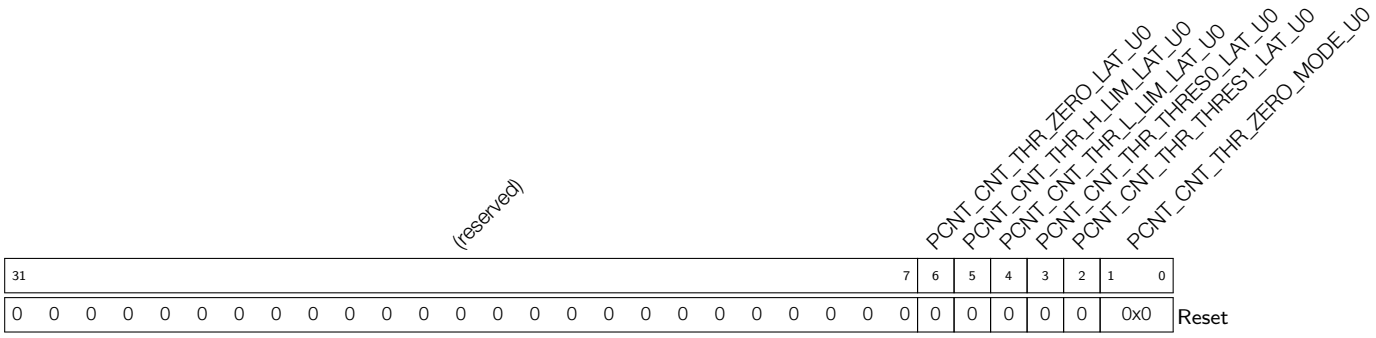
PCNT_DATE 脉冲计数器的版本控制寄存器。(读/写)

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



PCNT_PULSE_CNT_U*n* 存储单元 *n* 脉冲计数器的当前值。(只读)

Register 8.7: 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)

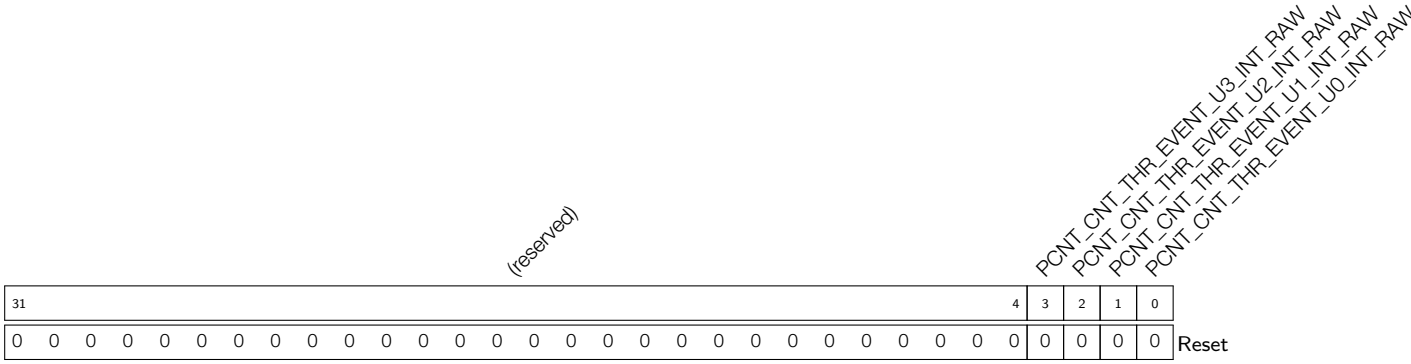
PCNT_CNT_THR_THRES0_LAT_U*n* 阈值中断有效时，PCNT_U*n* 阈值 0 的锁存值。1：脉冲计数器的当前值与阈值 0 相等，阈值 0 有效。0：其他。(只读)

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)

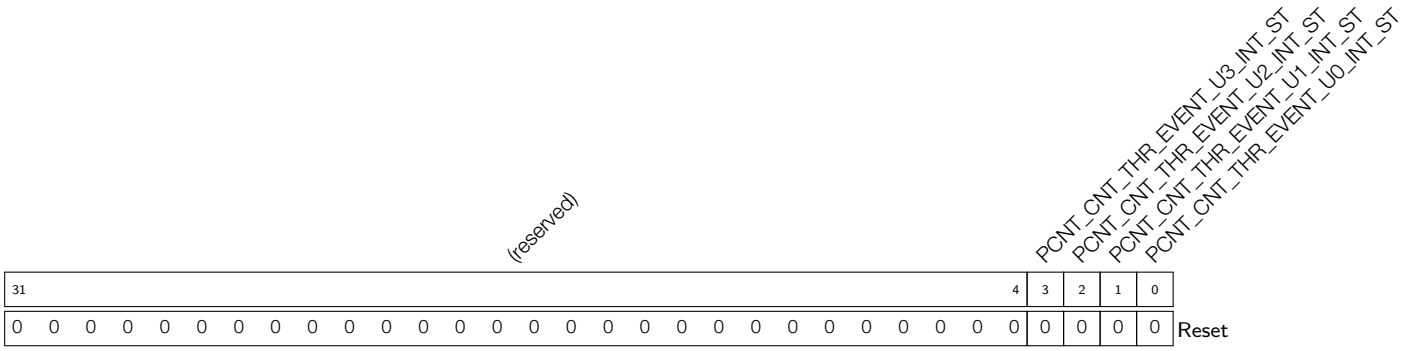
PCNT_CNT_THR_ZERO_LAT_U*n* 阈值中断有效时，PCNT_U*n* 阈值 0 的锁存值。1：脉冲计数器的当前值为 0，阈值 0 有效。0：其他。(只读)

Register 8.8: PCNT_INT_RAW_REG (0x0040)



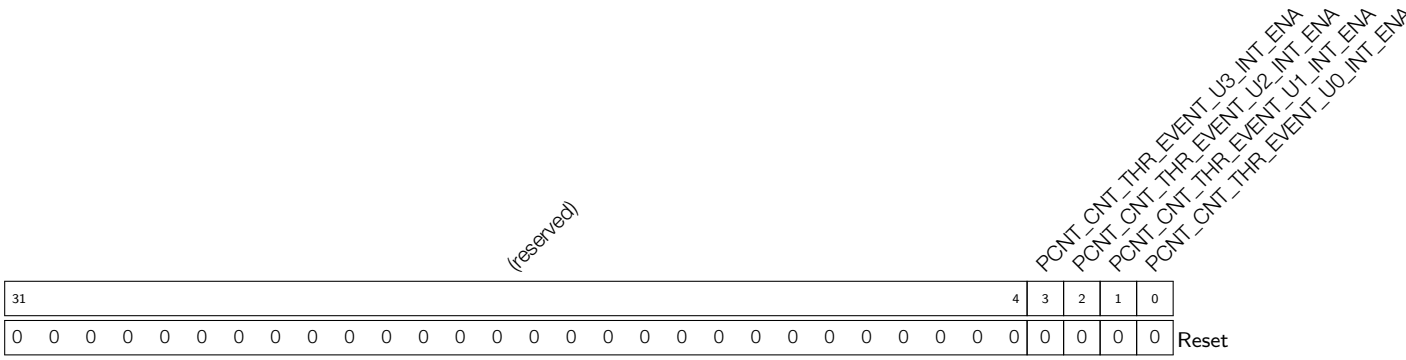
PCNT_CNT_THR_EVENT_U*n*_INT_RAW 单元 *n* 事件中断的原始中断状态位。(只读)

Register 8.9: PCNT_INT_ST_REG (0x0044)



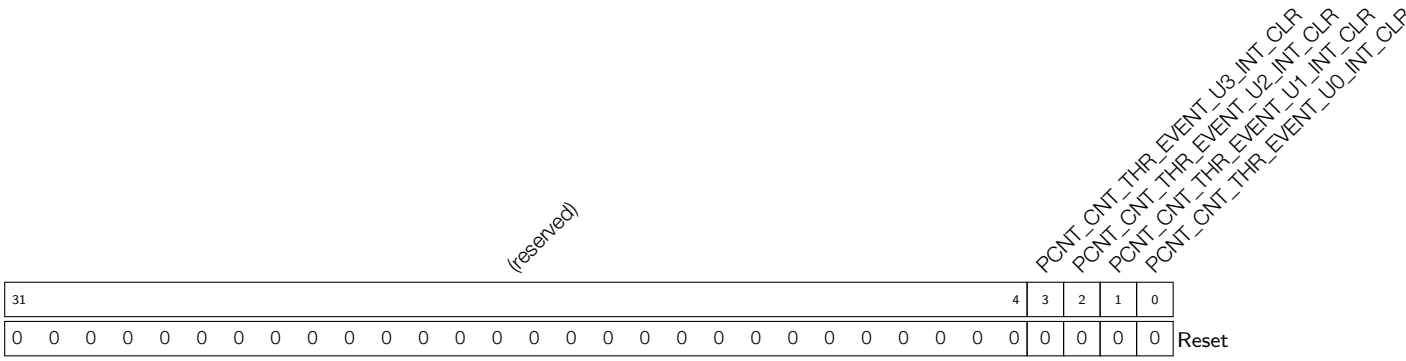
PCNT_CNT_THR_EVENT_U*n*_INT_ST 单元 *n* 事件中断的屏蔽中断状态位。(只读)

Register 8.10: 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 8.11: 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)

9. 64 位定时器

9.1 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9-1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。

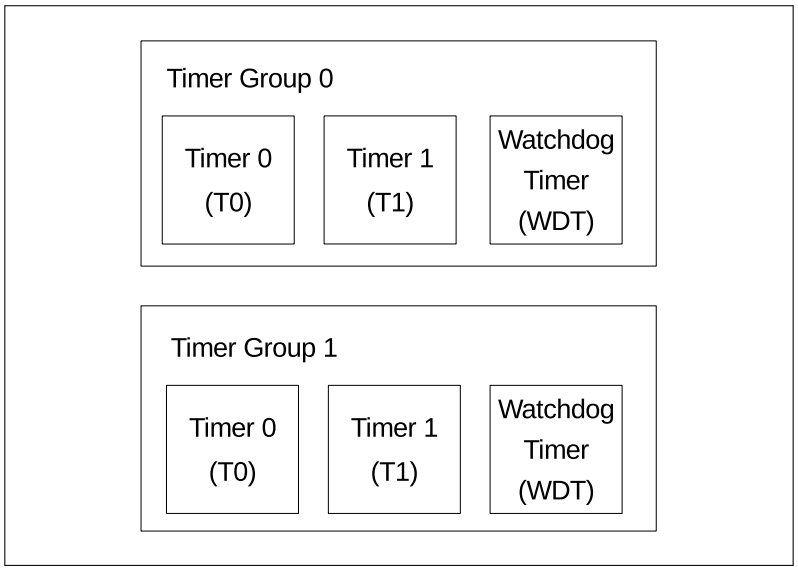


图 9-1. 定时器组

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 10 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

9.2 功能描述

9.2.1 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。

9.2.2 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。

9.2.3 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 9.2.4 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。

9.2.4 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。

9.2.5 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。

9.3 配置与使用

9.3.1 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。

9.3.2 定时器用于一次性报警

1. 按照第 9.3.1 节的第 1 步配置时基计数器。
2. 配置报警。
    - 置位 TIMG_T*x*ALARMLO_REG 和 TIMG_T*x*ALARMHI_REG 配置报警值。
    - 置位 TIMG_T*x*_LEVEL_INT_EN 或 TIMG_T*x*_EDGE_INT_EN 分别使能电平中断和边沿中断。
3. 清零 TIMG_T*x*_AUTORELOAD 关闭自动重新加载。
4. 置位 TIMG_T*x*_EN 开启定时器。
5. 处理报警中断。
    - 置位定时器对应的 TIMG_T*x*_INT_CLR 位清除中断。
    - 清零 TIMG_T*x*_EN 关闭定时器。

9.3.3 定时器用于周期性报警

1. 按照第 9.3.1 节的第 1 步配置时基计数器。
2. 按照第 9.3.2 节的第 2 步配置报警。
3. 置位TIMG_T*x*_AUTORELOAD 使能自动重新加载，将重新加载值写入 TIMG_T*x*_LOAD_LO 和 TIMG_T*x*_LOAD_HI。
4. 置位 TIMG_T*x*_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_T*x*_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_T*x*ALARMLO_REG、TIMG_T*x*ALARMHI_REG、TIMG_T*x*_LOAD_LO 和 TIMG_T*x*_LOAD_HI 。否则，上述寄存器应保持不变。
    - 置位 TIMG_T*x*_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_T*x*_INT_CLR 位清除中断。
    - 清零 TIMG_T*x*_EN 关闭定时器。

9.4 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 9-1 所示。更多信息，请访问章节 1 系统和存储器。

表 9-1. 64 位定时器基地址

	访问总线	基地址
TIMG0	PeriBUS1	0x3F41F000
	PeriBUS2	0x6001F000
TIMG1	PeriBUS1	0x3F420000
	PeriBUS2	0x60020000

9.5 寄存器列表

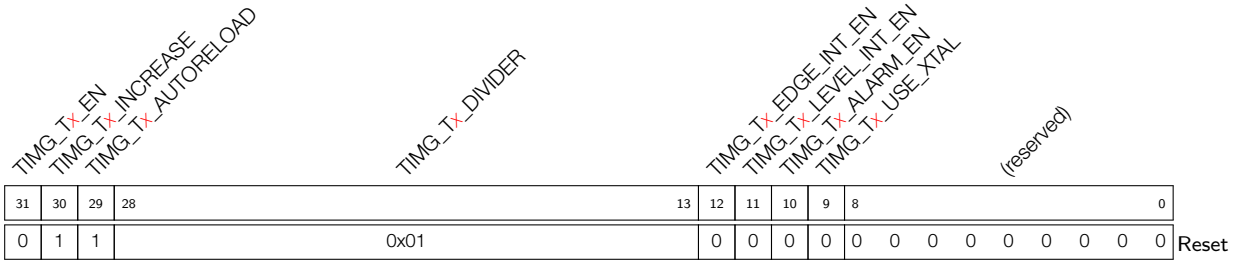
请注意，下表的地址指相对于 64 位定时器基地址的偏移量（相对地址）。请参阅第 9.4 节获取有关 64 位定时器基地址的信息。

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

名称	描述	地址	访问
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
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

9.6 寄存器

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



TIMG_Tx_USE_XTAL 2：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)

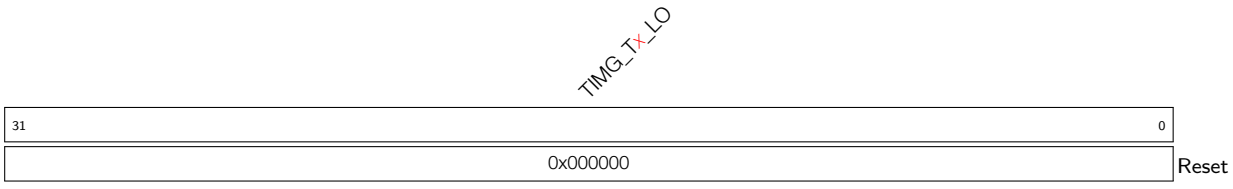
TIMG_Tx_DIVIDER 定时器 *x* 时钟 (Tx_clk) 的预分频器值。(读/写)

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)

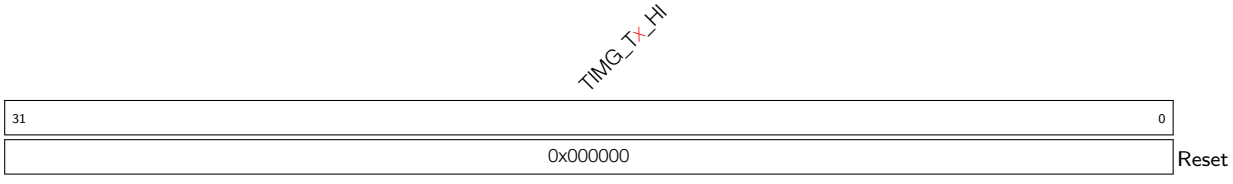
TIMG_Tx_EN 置 1 后，定时器 *x* 时基计数器使能。(读/写)

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



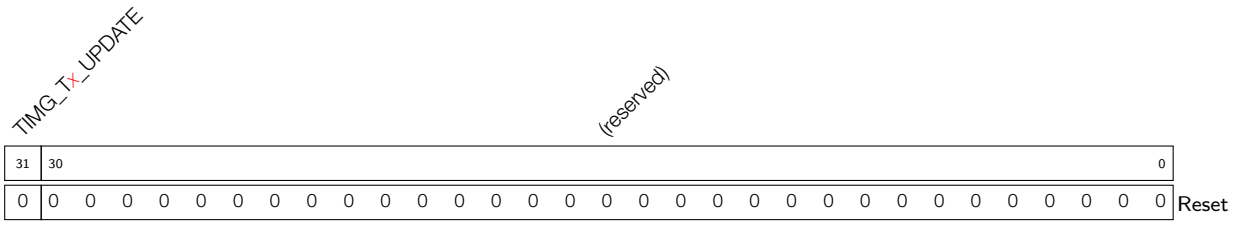
TIMG_Tx_LO 在 TIMG_TxUPDATE_REG 上写值后，可读取定时器 *x* 时基计数器的低 32 位。(只读)

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



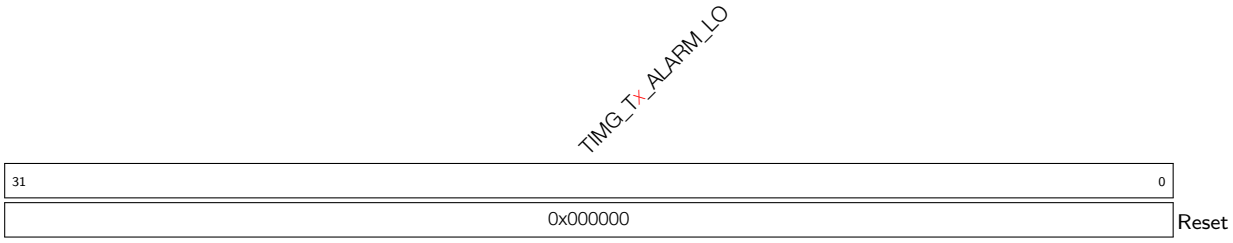
TIMG_Tx_HI 在 TIMG_TxUPDATE_REG 上写值后，可读取定时器 *x* 时基计数器的高 32 位。(只读)

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



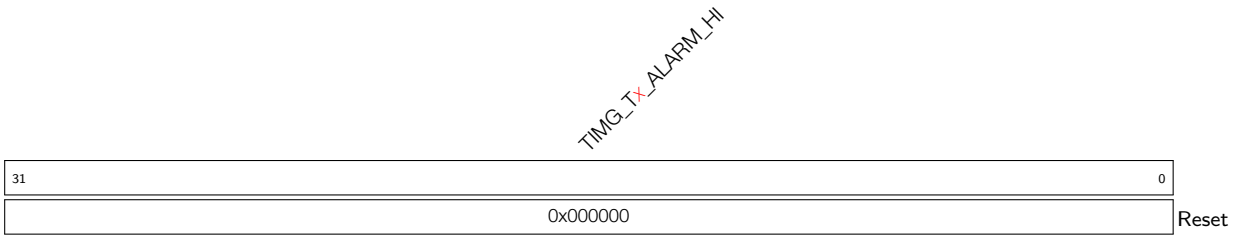
TIMG_Tx_UPDATE 在 TIMG_TxUPDATE_REG 上写 0 或 1，计数器的值被锁存 (读/写)

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



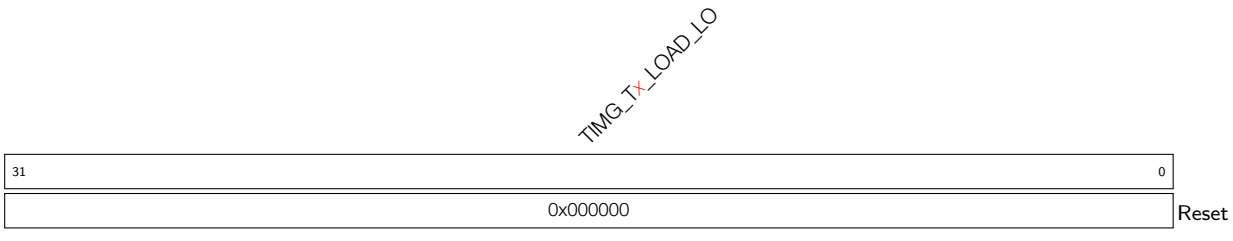
TIMG_Tx_ALARM_LO 定时器 x 时基计数器低 32 位的警报触发值。(读/写)

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



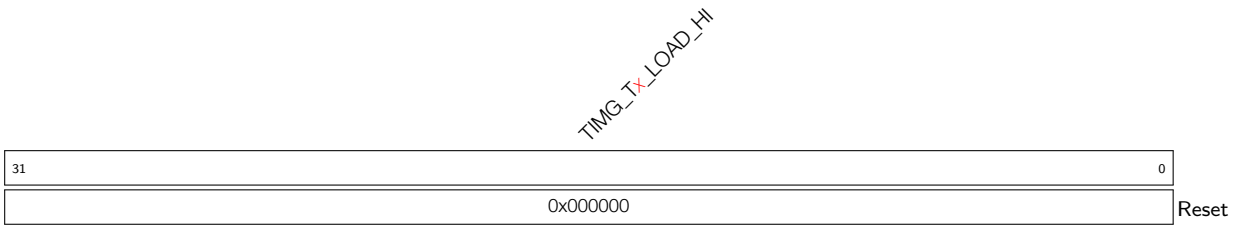
TIMG_Tx_ALARM_HI 定时器 x 时基计数器高 32 位的警报触发值。(读/写)

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



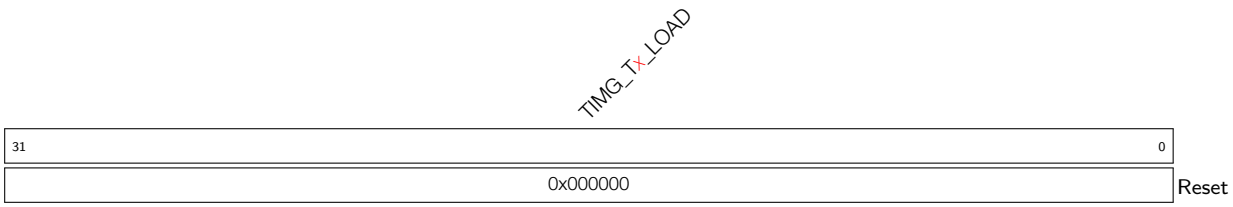
TIMG_Tx_LOAD_LO 定时器 x 时基计数器低 32 位的重新加载值。(读/写)

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



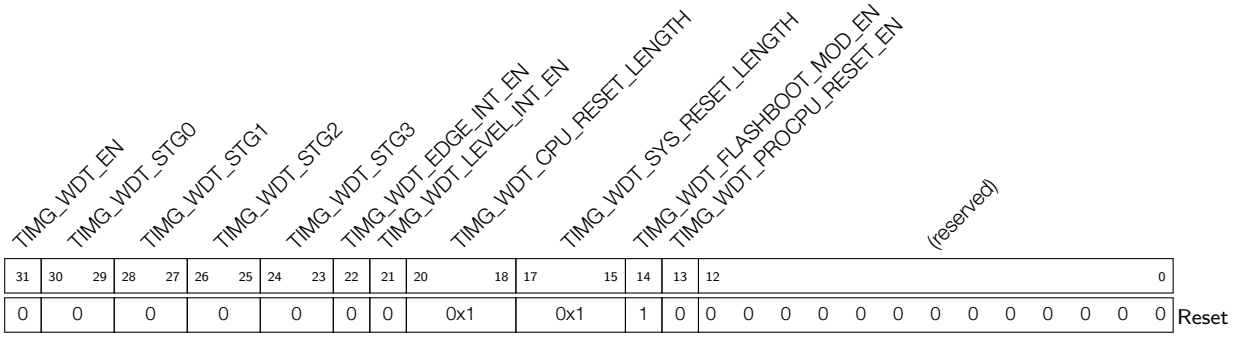
TIMG_Tx_LOAD_HI 定时器 x 时基计数器高 32 位的重新加载值。(读/写)

Register 9.9: 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 9.10: 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)

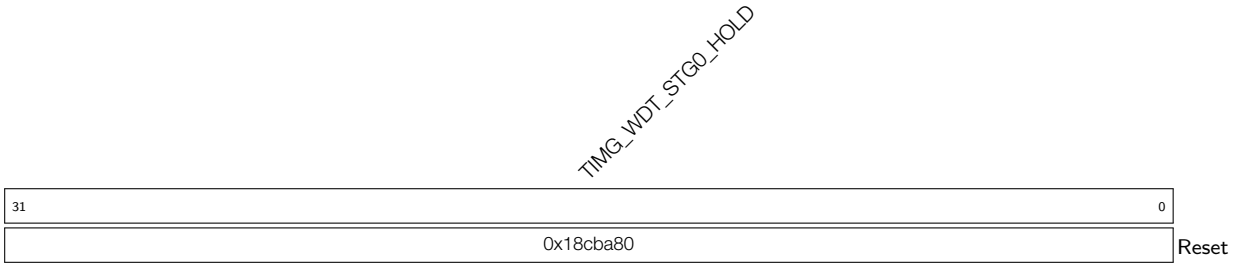
TIMG_WDT_STG0 阶段 0 配置。0：关闭，1：中断，2：复位 CPU，3：复位系统。(读/写)

TIMG_WDT_EN 置 1 后，MWDT 使能。(读/写)

Register 9.11: TIMG_WDTCONFIG1_REG (0x004C)

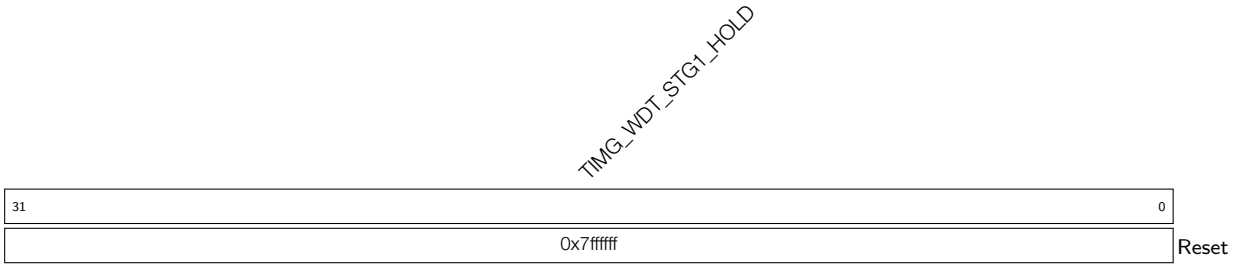
TIMG_WDT_CLK_PRESCALER MWDT 时钟预分频器值。MWDT 时钟长度 = 12.5 ns * TIMGn_WDT_CLK_PRESCALE。(读/写)

Register 9.12: TIMG_WDTCONFIG2_REG (0x0050)



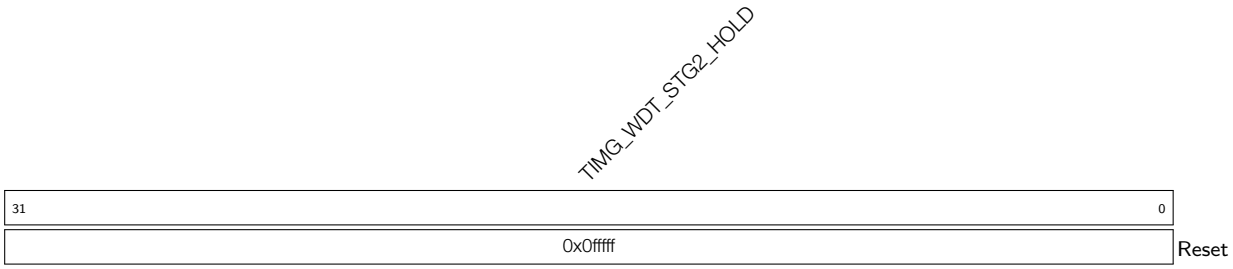
TIMG_WDT_STG0_HOLD MWDT 时钟周期中阶段 0 超时时间。(读/写)

Register 9.13: TIMG_WDTCONFIG3_REG (0x0054)



TIMG_WDT_STG1_HOLD MWDT 时钟周期中阶段 1 超时时间。(读/写)

Register 9.14: TIMG_WDTCONFIG4_REG (0x0058)

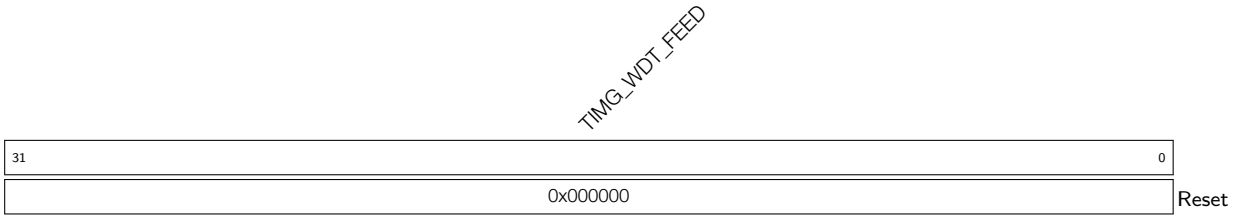


TIMG_WDT_STG2_HOLD MWDT 时钟周期中阶段 2 超时时间。(读/写)

Register 9.15: TIMG_WDTCONFIG5_REG (0x005C)

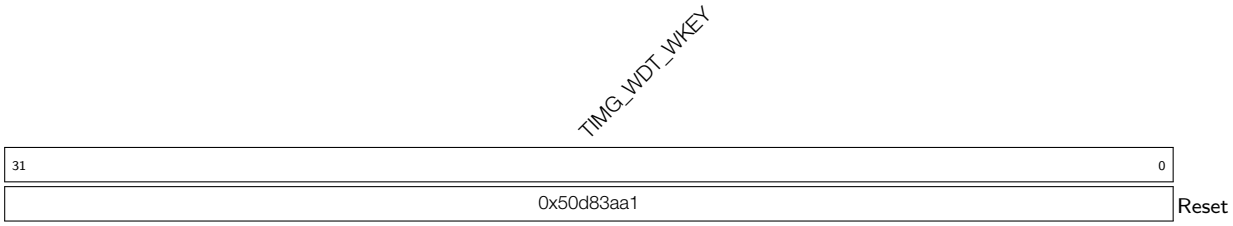
TIMG_WDT_STG3_HOLD MWDT 时钟周期中阶段 3 超时时间。(读/写)

Register 9.16: TIMG_WDTFEED_REG (0x0060)



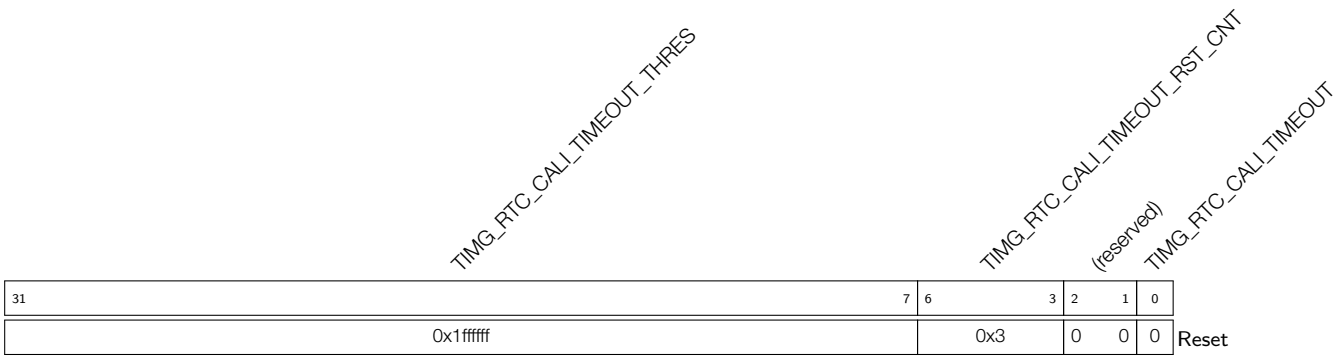
TIMG_WDT_FEED 写任意值驱动 MWDT。(只写)

Register 9.17: 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 9.18: TIMG_RTCCALICFG2_REG (0x00A8)

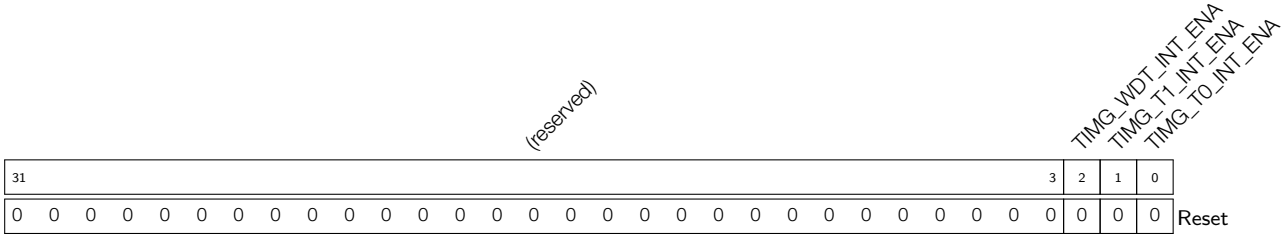


TIMG_RTC_CALI_TIMEOUT RTC 校准超时指示器 (只读)

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)

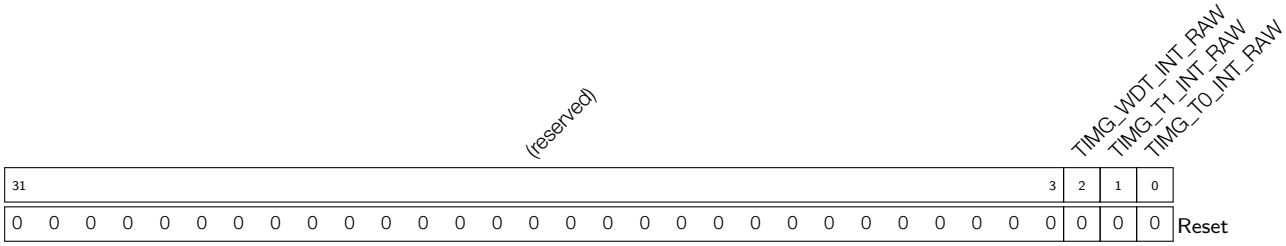
Register 9.19: TIMG_INT_ENA_TIMERS_REG (0x0098)



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)

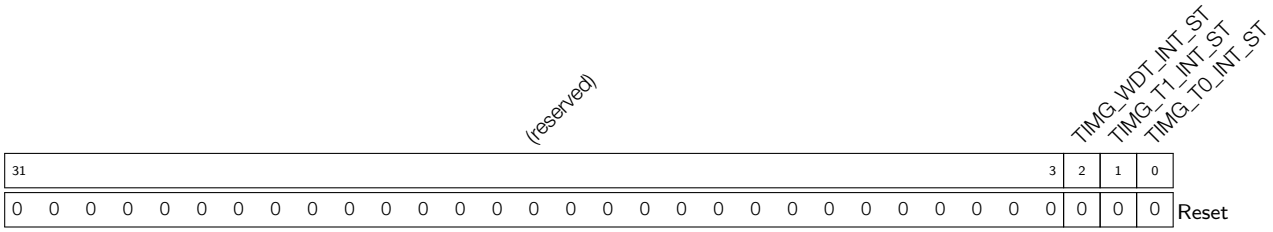
Register 9.20: TIMG_INT_RAW_TIMERS_REG (0x009C)



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)

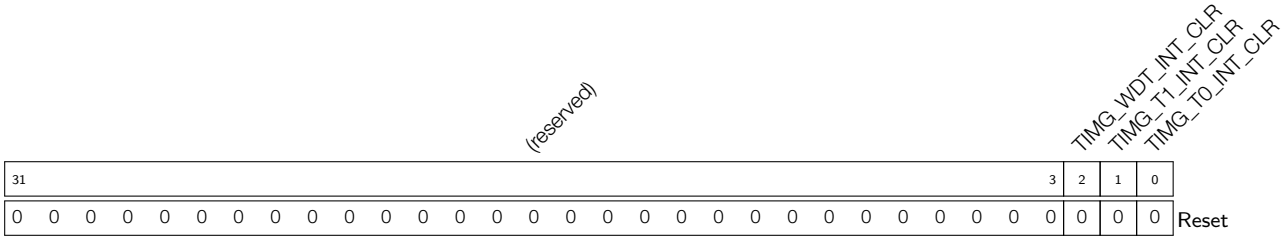
Register 9.21: TIMG_INT_ST_TIMERS_REG (0x00A0)



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)

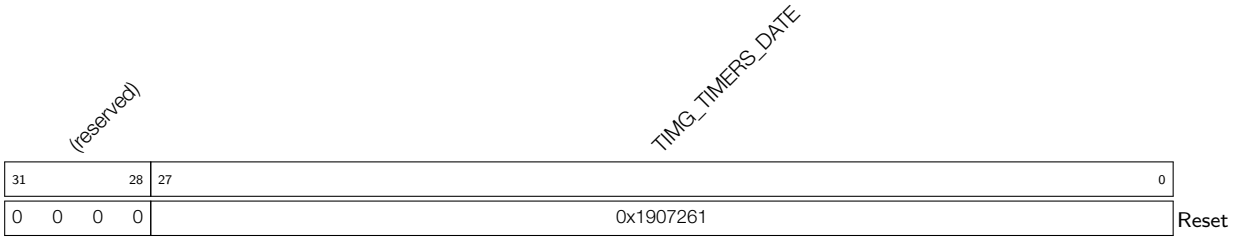
Register 9.22: TIMG_INT_CLR_TIMERS_REG (0x00A4)



TIMG_T*x*_INT_CLR 置位此位，清除 TIMG_T*x*_INT 中断。(只写)

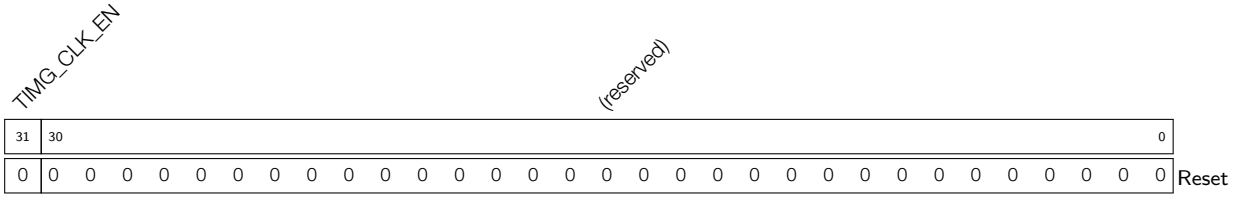
TIMG_WDT_INT_CLR 置位此位，清除 TIMG_WDT_INT 中断。(只写)

Register 9.23: TIMG_TIMERS_DATE_REG (0x00F8)



TIMG_TIMERS_DATE 版本控制寄存器。(读/写)

Register 9.24: 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)

10. 看门狗定时器

10.1 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 9 章：*64 位定时器*。

10.2 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。

10.3 功能描述

10.3.1 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。

10.3.2 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。

10.3.3 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。

10.3.4 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。

10.4 寄存器

MWDT “寄存器” 是定时器模块的一部分，在第 9 章：*64 位定时器*定时器寄存器中有详细描述。

11. eFuse 控制器

11.1 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。

11.2 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

11.3 功能描述

11.3.1 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 25 位供硬件使用，软件不可见；还有 37 位处于保留状态，留作未来使用。

表 11-1 列出了 BLOCK0 中的参数名称、偏移地址、位宽、是否可供硬件使用、烧写保护，以及描述信息。

在这些参数中，**EFUSE_WR_DIS** 用于控制其他参数的烧写，**EFUSE_RD_DIS** 用于控制软件读取 BLOCK4 ~ BLOCK10。更多关于这两个参数的信息请见章节 11.3.1.1、11.3.1.2。

表 11-1. 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

参数	偏移	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	描述
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
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 11-2
EFUSE_KEY_PURPOSE_1	92	4	Y	9	Key1 用途，见表 11-2
EFUSE_KEY_PURPOSE_2	96	4	Y	10	Key2 用途，见表 11-2
EFUSE_KEY_PURPOSE_3	100	4	Y	11	Key3 用途，见表 11-2
EFUSE_KEY_PURPOSE_4	104	4	Y	12	Key4 用途，见表 11-2
EFUSE_KEY_PURPOSE_5	108	4	Y	13	Key5 用途，见表 11-2
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

参数	偏移	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	描述
EFUSE_DIS_LEGACY_SPI_BOOT	129	1	N	18	关闭 Legacy SPI boot 模式
EFUSE_UART_PRINT_CONTROL	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
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 11-2 为密钥用途各个数值对应的含义。通过配置参数 EFUSE_KEY_PURPOSE_*n* 来声明 KEY*n* 用途 (*n*: 0 ~ 5)。

表 11-2. 密钥用途数值对应的含义

密钥用途数值	含义
0	指定为用户使用（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 11-3 列出了 BLOCK1 ~ BLOCK10 中存储的参数的信息。

表 11-3. BLOCK1-10 参数

块	参数	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	EFUSE_RD_DIS 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)
		[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
		[54:53]	N	20	N/A	D6
		[60:53]	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 11-2）。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 11.3.1.3：*数据存储方式*，和章节 11.3.2：*软件烧写参数*。

11.3.1.1 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 11.3.3 中：*更新 eFuse 读寄存器*）。

表 11-1 以及表 11-3 中的“EFUSE_WR_DIS 烧写保护位”列描述了各参数的烧写保护状态具体由 EFUSE_WR_DIS 的哪个位决定。

当某个参数对应的烧写保护位为 0 时，表示此参数未处于烧写保护状态，可以烧写该参数，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。

11.3.1.2 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 11-3 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 11.3.3 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。

11.3.1.3 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 11-1 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。

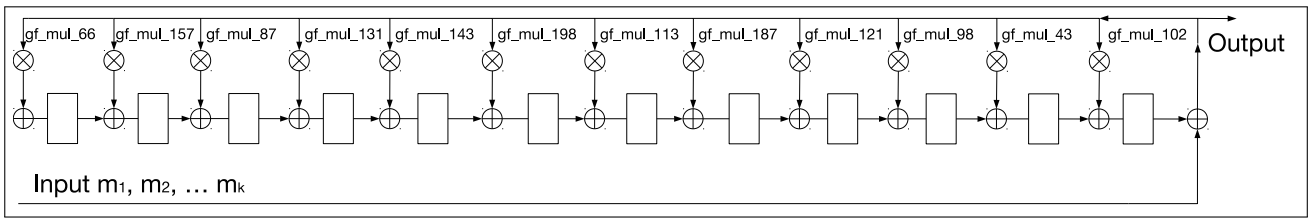


图 11-1. 移位寄存器电路图

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。

11.3.2 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 25 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_DATA1_REG[16]
- 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 11.3.4.1：*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 11.3.4.2：*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 11.3.3 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 11.3.3：*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。

11.3.3 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
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 11.3.4.3：*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。

11.3.4 时序

11.3.4.1 eFuse 烧写时序

图 11-2 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

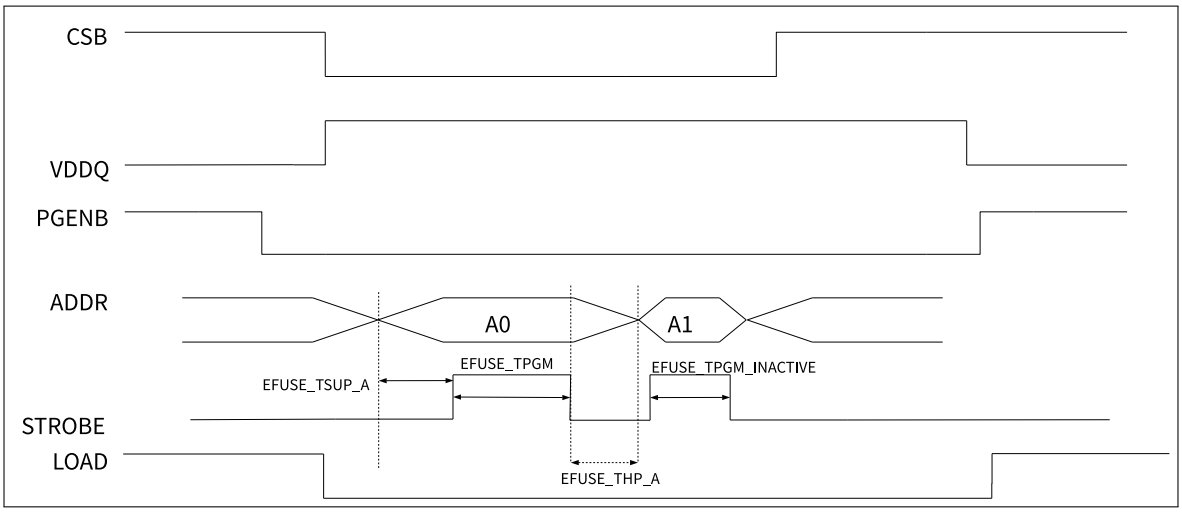


图 11-2. eFuse 烧写时序图

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 11-5。复位后，eFuse 默认的参数配置对应的是 20 MHz 时钟频率。

表 11-5. 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)

APB Frequency	EFUSE_TSUP_A (> 6.669 ns)	EFUSE_TPGM (9-11 μs, usually 10 μs)	EFUSE_THP_A (> 6.166 ns)	EFUSE_TPGM_INACTIVE (> 35.96 ns)
80 MHz	0x1	0x320	0x1	0x3
40 MHz	0x1	0x190	0x1	0x2
20 MHz	0x1	0xC8	0x1	0x1

图 11-2 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。

11.3.4.2 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 11-6. 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

11.3.4.3 eFuse 读取时序

图 11-3 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。

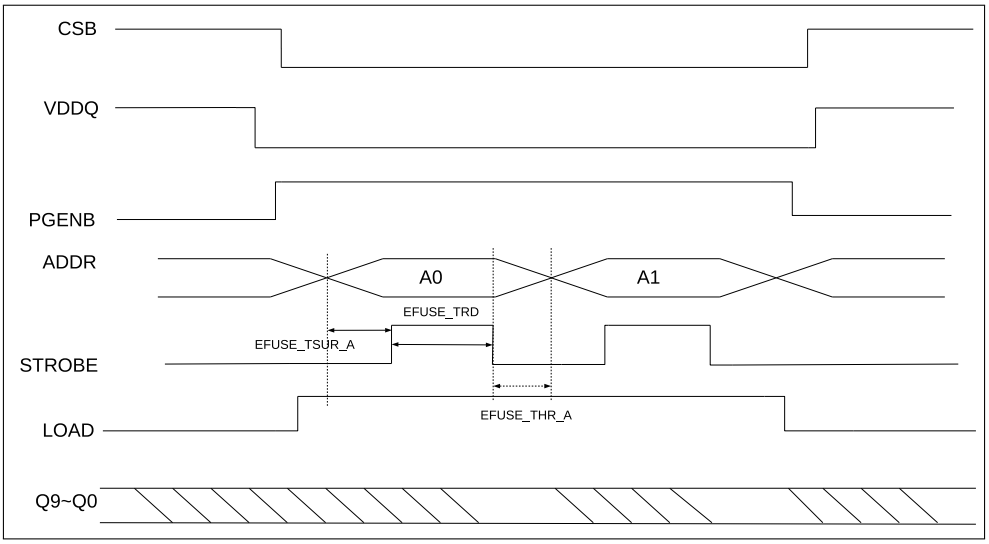


图 11-3. eFuse 读取时序图

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 11-7。

表 11-7. 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	0x1	0x3	0x1
40 MHz	0x1	0x2	0x1
20 MHz	0x1	0x1	0x1

11.3.5 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 11-1 和 11-3 “硬件使用”一栏中标记为“Y”的参数。

11.3.6 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。

11.4 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 11-8 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 11-8. eFuse 控制器基地址

访问外设总线	地址值
PeriBUS1	0x6001A000
PeriBUS2	0x3FC1A000

11.5 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 11.4 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

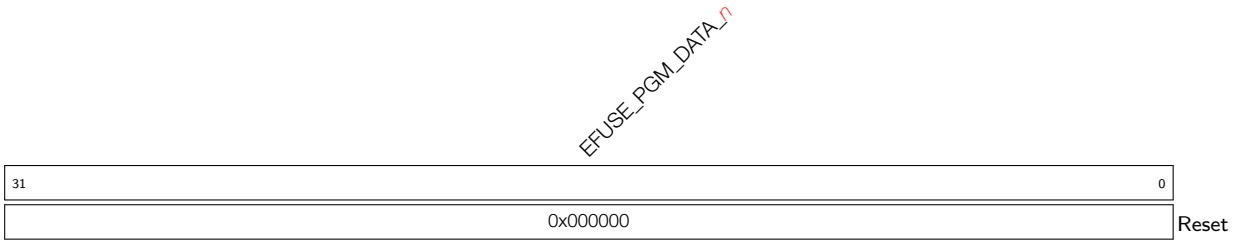
名称	描述	地址	访问
EFUSE_PGM_DATA6_REG	存放待烧写数据的第 6 个寄存器内容。	0x0018	读/写
EFUSE_PGM_DATA7_REG	存放待烧写数据的第 7 个寄存器内容。	0x001C	读/写
EFUSE_PGM_CHECK_VALUE0_REG	存放待烧写 RS 代码的第 0 个寄存器数据。	0x0020	读/写
EFUSE_PGM_CHECK_VALUE1_REG	存放待烧写 RS 代码的第 1 个寄存器数据。	0x0024	读/写
EFUSE_PGM_CHECK_VALUE2_REG	存放待烧写 RS 代码的第 2 个寄存器数据。	0x0028	读/写
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
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

名称	描述	地址	访问
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
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

名称	描述	地址	访问
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			
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

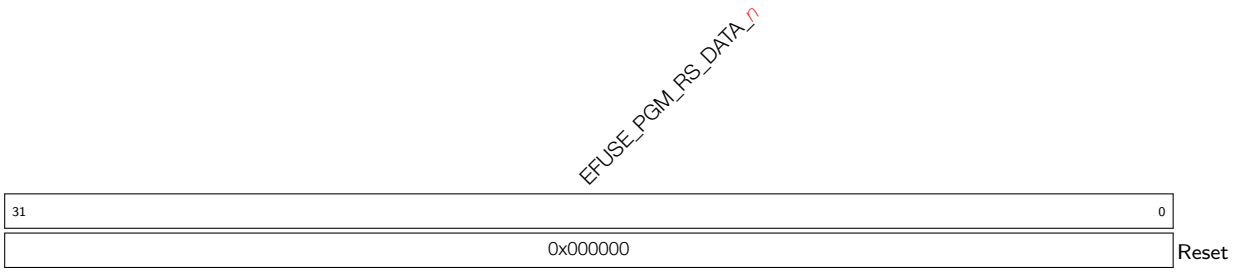
11.6 寄存器

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



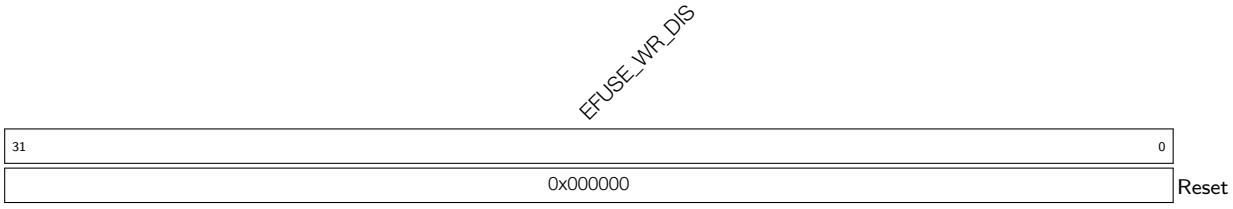
EFUSE_PGM_DATA_*n* 存放待烧写数据的第 *n* 个 32 位数据内容。(读/写)

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



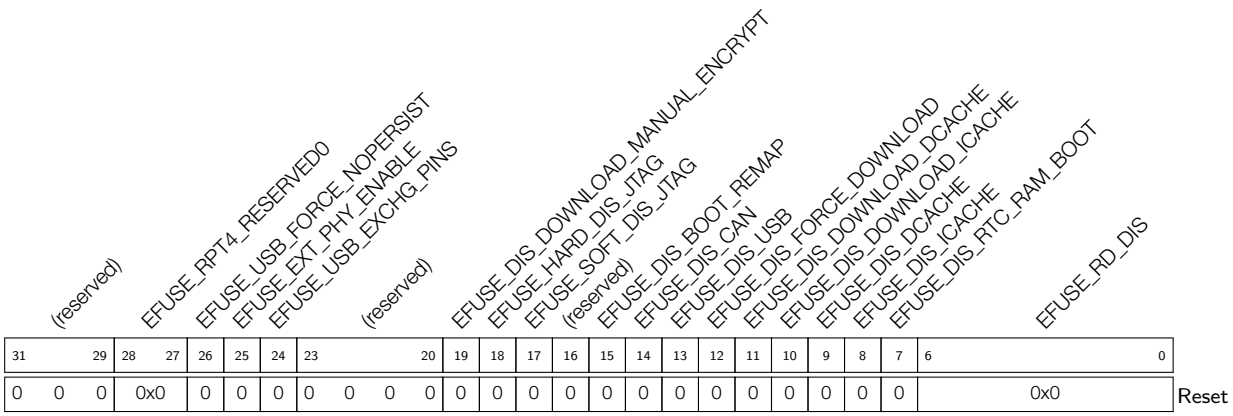
EFUSE_PGM_RS_DATA_*n* 存放待烧写 RS 代码的第 *n* 个 32 位数据内容。(读/写)

Register 11.3: EFUSE_RD_WR_DIS_REG (0x002C)



EFUSE_WR_DIS 置位禁用 eFuse 烧写。(只读)

Register 11.4: 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_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 11.5: 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)

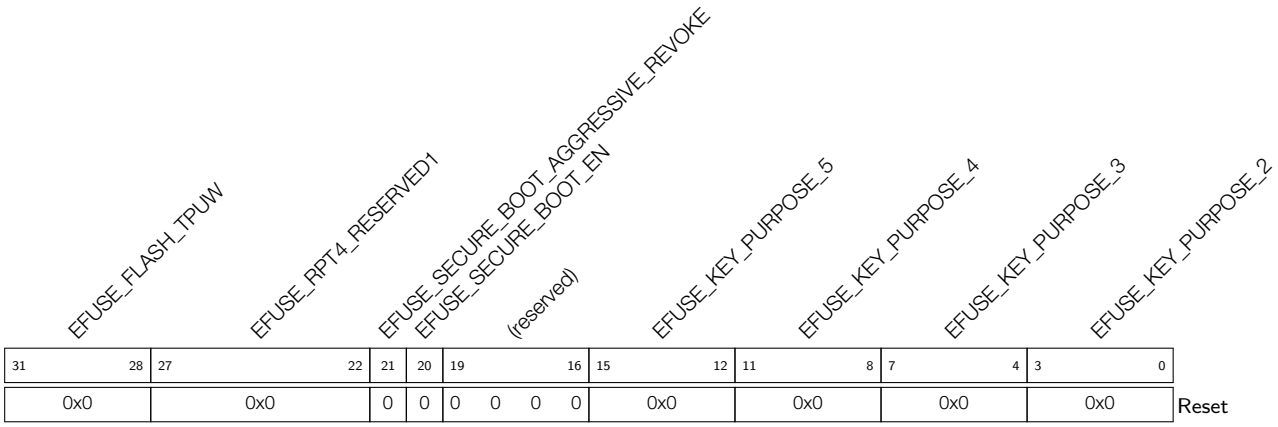
EFUSE_SECURE_BOOT_KEY_REVOKE1 置位使能撤销第二个安全启动密钥。(只读)

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 11-2：密钥用途数值对应的含义。(只读)

EFUSE_KEY_PURPOSE_1 KEY1 purpose，详见表 11-2：密钥用途数值对应的含义。(只读)

Register 11.6: EFUSE_RD_REPEAT_DATA2_REG (0x0038)



EFUSE_KEY_PURPOSE_2 KEY2 purpose，详见表 11-2：密钥用途数值对应的含义。(只读)

EFUSE_KEY_PURPOSE_3 KEY3 purpose，详见表 11-2：密钥用途数值对应的含义。(只读)

EFUSE_KEY_PURPOSE_4 KEY4 purpose，详见表 11-2：密钥用途数值对应的含义。(只读)

EFUSE_KEY_PURPOSE_5 KEY5 purpose，详见表 11-2：密钥用途数值对应的含义。(只读)

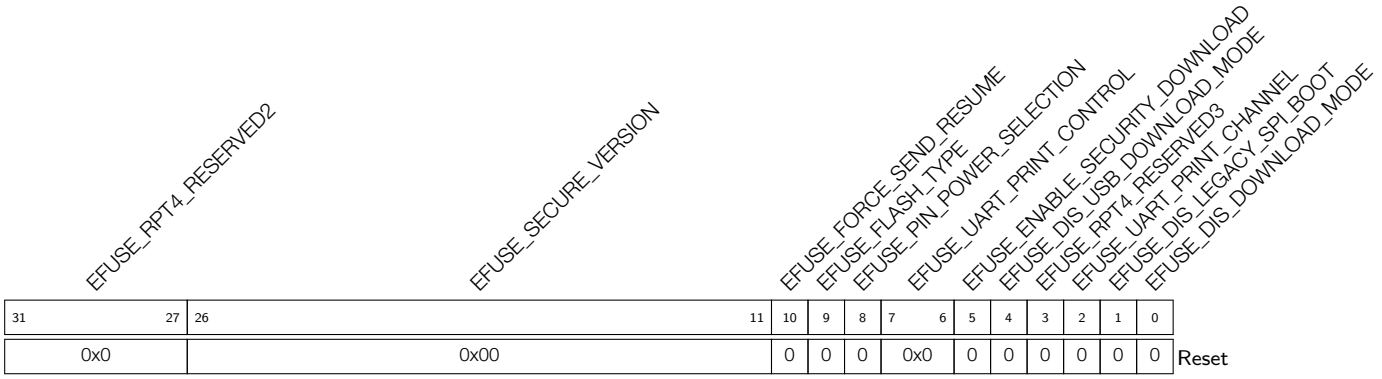
EFUSE_SECURE_BOOT_EN 置位使能安全启动。(只读)

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 11.7: 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)

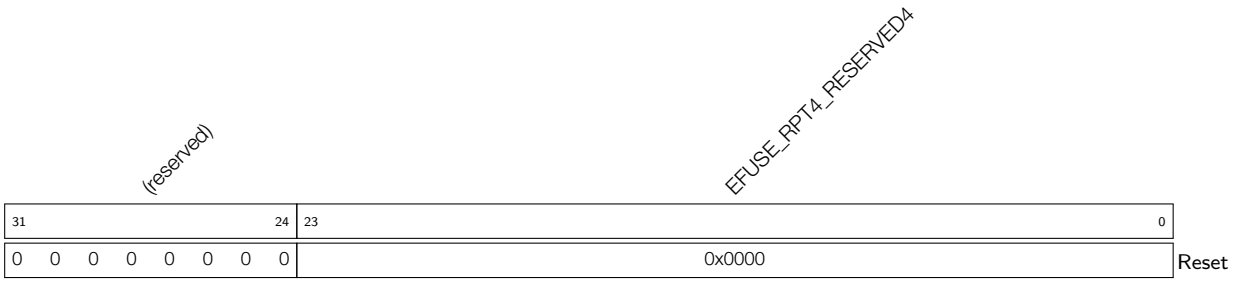
EFUSE_FLASH_TYPE Flash 类型。0：4 根数据线；1：8 根数据线。(只读)

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)

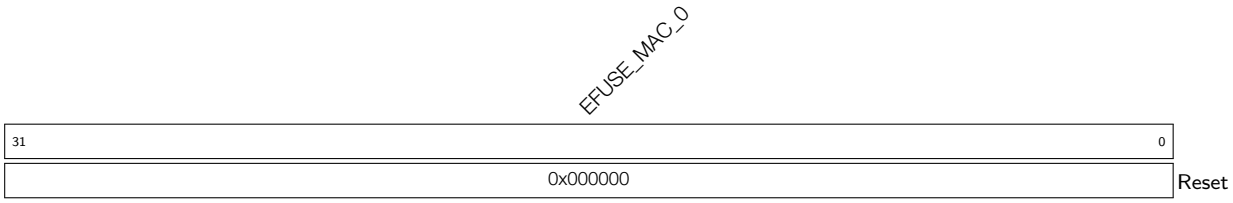
EFUSE_RPT4_RESERVED2 保留（采用 4 备份编码)。(只读)

Register 11.8: EFUSE_RD_REPEAT_DATA4_REG (0x0040)



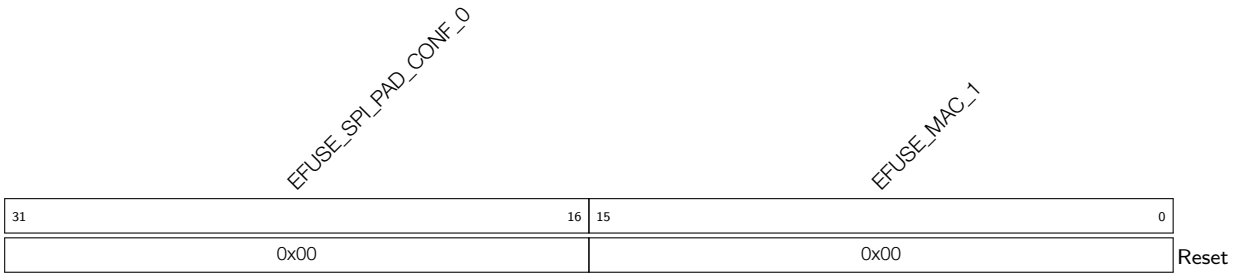
EFUSE_RPT4_RESERVED4 保留（采用 4 备份编码）。（只读）

Register 11.9: EFUSE_RD_MAC_SPI_SYS_0_REG (0x0044)



EFUSE_MAC_0 存储 MAC 地址低 32 位的内容。（只读）

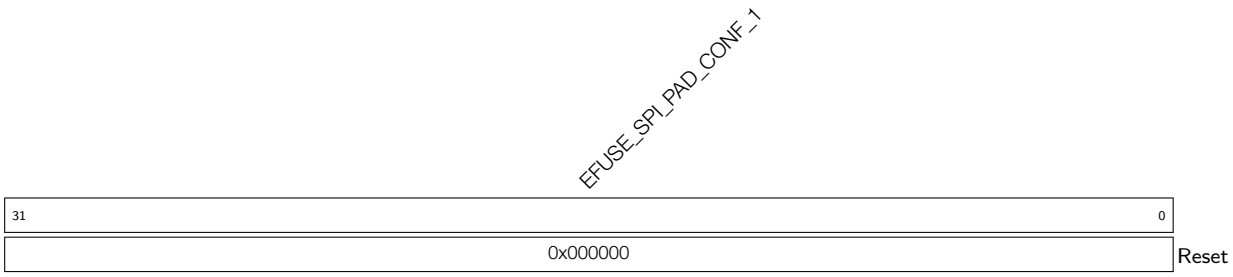
Register 11.10: EFUSE_RD_MAC_SPI_SYS_1_REG (0x0048)



EFUSE_MAC_1 存储 MAC 地址高 16 位的内容。（只读）

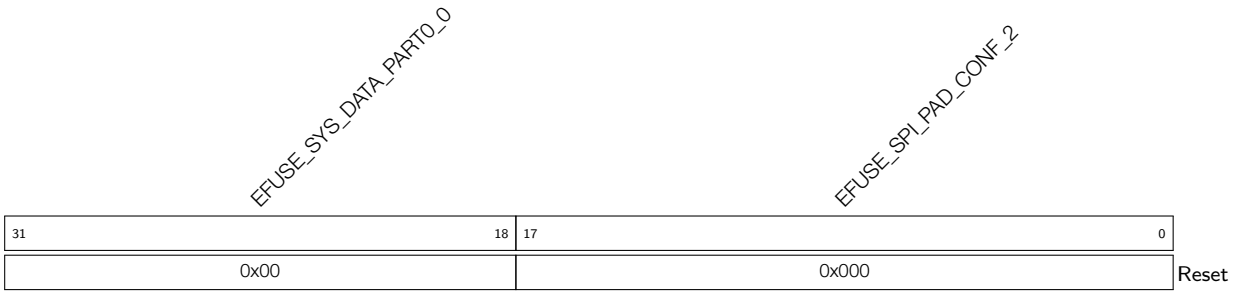
EFUSE_SPI_PAD_CONF_0 存储 SPI_PAD_CONF 第 0 部分的内容。（只读）

Register 11.11: 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）

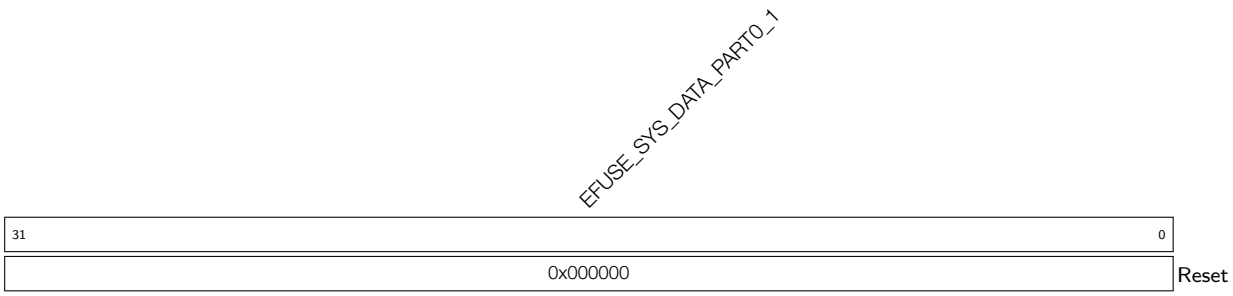
Register 11.12: EFUSE_RD_MAC_SPI_SYS_3_REG (0x0050)



EFUSE_SPI_PAD_CONF_2 存储 SPI_PAD_CONF 第 2 部分的内容。（只读）

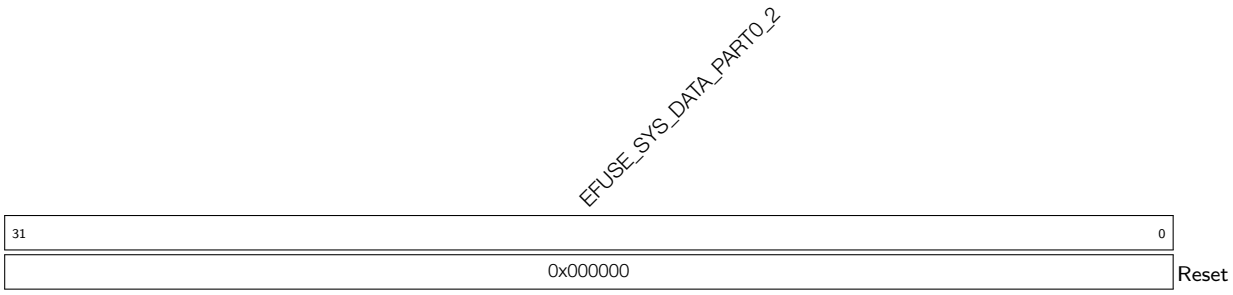
EFUSE_SYS_DATA_PART0_0 存储系统数据第 0 部分的第 0 部分的内容。（只读）

Register 11.13: EFUSE_RD_MAC_SPI_SYS_4_REG (0x0054)



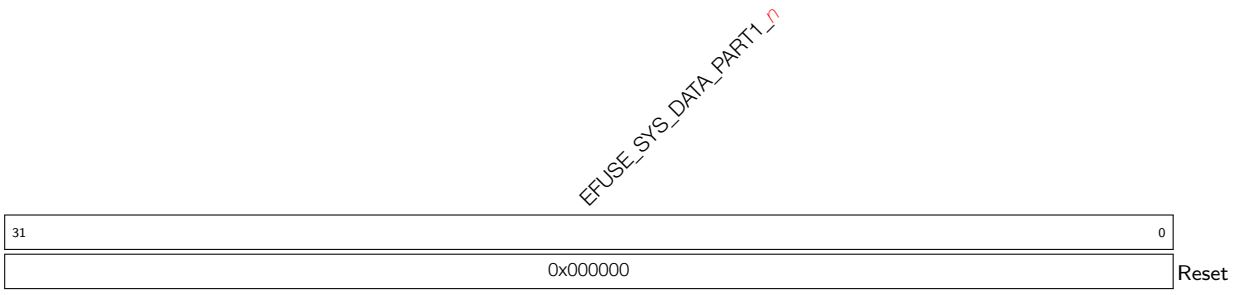
EFUSE_SYS_DATA_PART0_1 存储系统数据第 0 部分的第 1 部分的内容。（只读）

Register 11.14: EFUSE_RD_MAC_SPI_SYS_5_REG (0x0058)



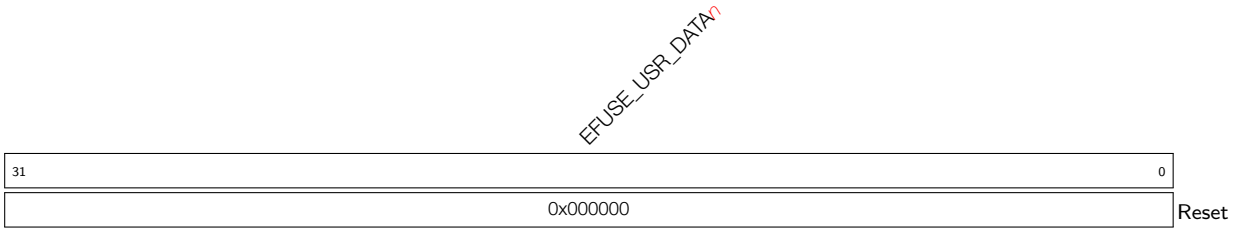
EFUSE_SYS_DATA_PART0_2 存储系统数据第 0 部分的第 2 部分的内容。(只读)

Register 11.15: EFUSE_RD_SYS_DATA_PART1_*n*_REG (*n*: 0-7) (0x005C+4**n*)



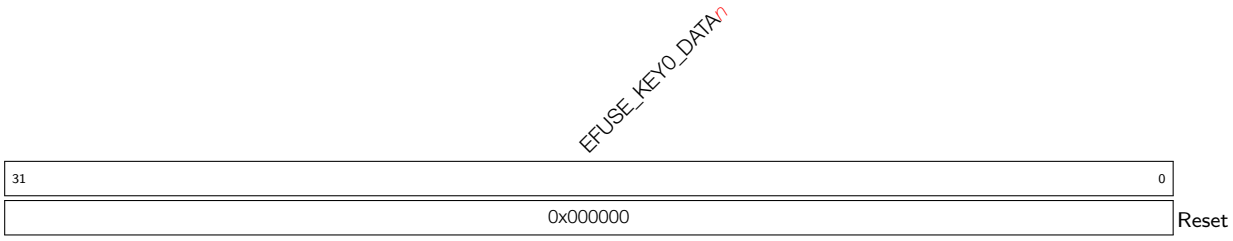
EFUSE_SYS_DATA_PART1_*n* 存储系统数据第 1 部分的第 *n* 个 32 位的内容。(只读)

Register 11.16: EFUSE_RD_USR_DATA*n*_REG (*n*: 0-7) (0x007C+4**n*)



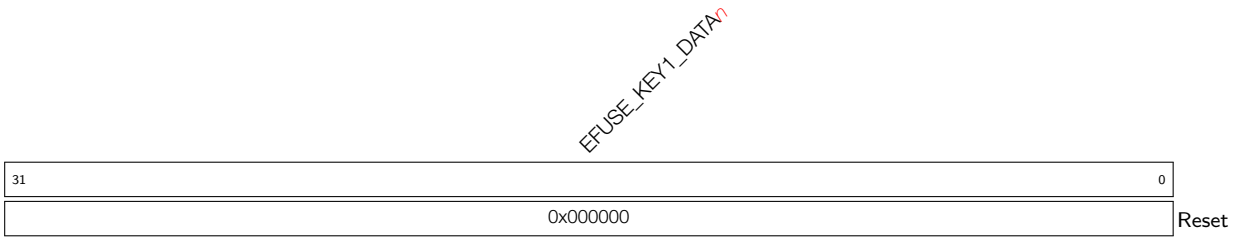
EFUSE_USR_DATA*n* 存储 BLOCK3 (user) 的第 *n* 个 32 位的内容。(只读)

Register 11.17: EFUSE_RD_KEY0_DATA*n*_REG (*n*: 0-7) (0x009C+4**n*)



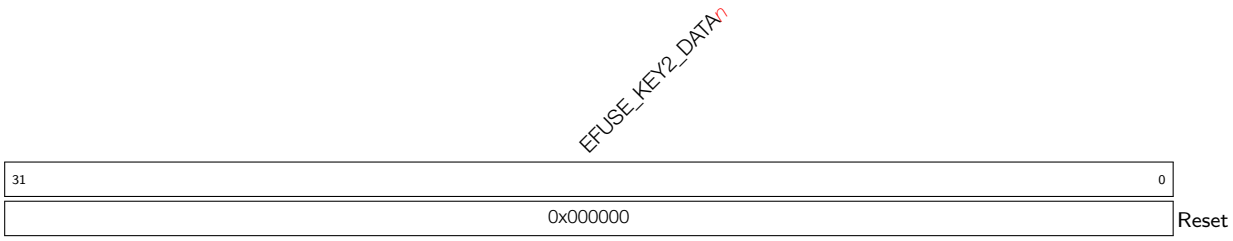
EFUSE_KEY0_DATA*n* 存储 KEY0 的第 *n* 个 32 位的内容。（只读）

Register 11.18: EFUSE_RD_KEY1_DATA*n*_REG (*n*: 0-7) (0x00BC+4**n*)



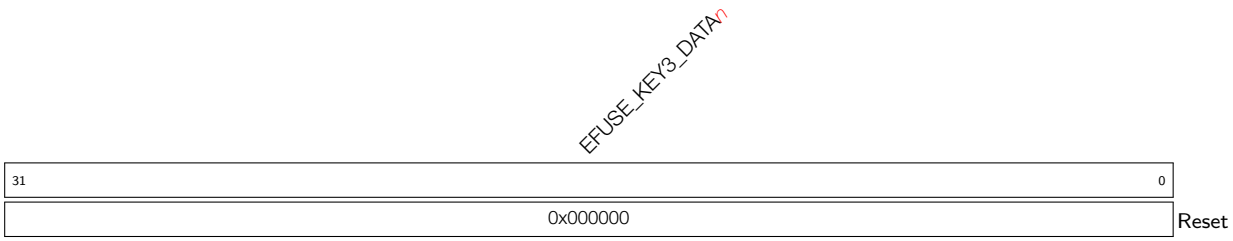
EFUSE_KEY1_DATA*n* 存储 KEY1 的第 *n* 个 32 位的内容。（只读）

Register 11.19: EFUSE_RD_KEY2_DATA*n*_REG (*n*: 0-7) (0x00DC+4**n*)



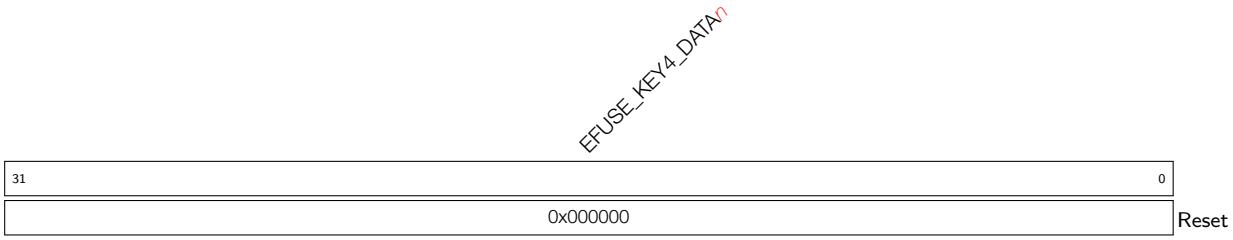
EFUSE_KEY2_DATA*n* 存储 KEY2 的第 *n* 个 32 位的内容。（只读）

Register 11.20: EFUSE_RD_KEY3_DATA*n*_REG (*n*: 0-7) (0x00FC+4**n*)



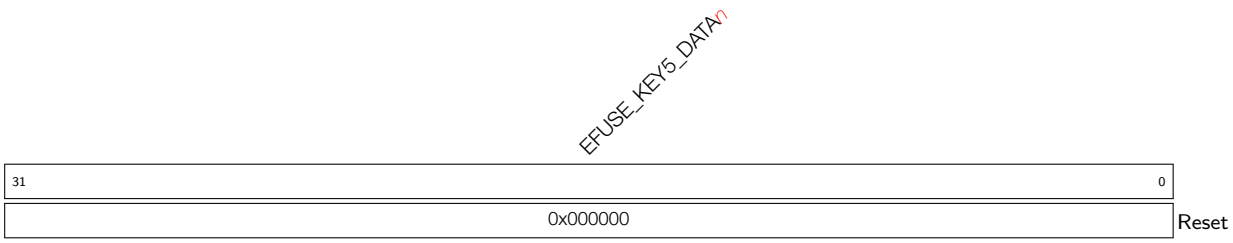
EFUSE_KEY3_DATA*n* 存储 KEY3 的第 *n* 个 32 位的内容。（只读）

Register 11.21: EFUSE_RD_KEY4_DATA*n*_REG (*n*: 0-7) (0x011C+4**n*)



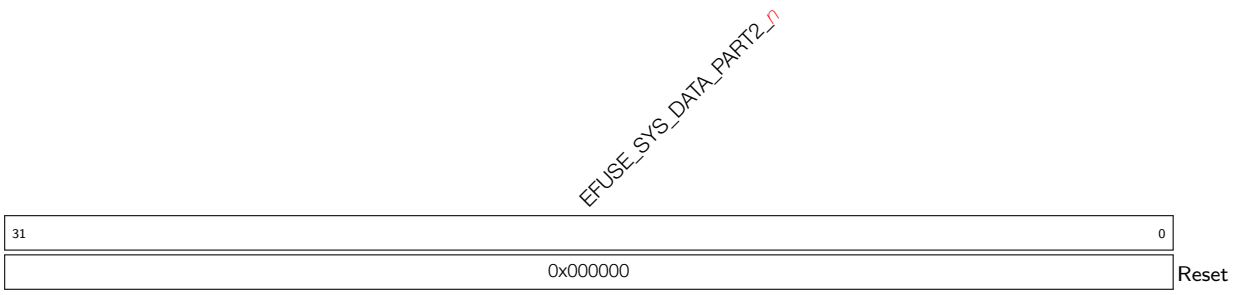
EFUSE_KEY4_DATA*n* 存储 KEY4 的第 *n* 个 32 位的内容。（只读）

Register 11.22: EFUSE_RD_KEY5_DATA*n*_REG (*n*: 0-7) (0x013C+4**n*)



EFUSE_KEY5_DATA*n* 存储 KEY5 的第 *n* 个 32 位的内容。（只读）

Register 11.23: EFUSE_RD_SYS_DATA_PART2_*n*_REG (*n*: 0-7) (0x015C+4**n*)



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 11.24: 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_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 11.24: 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）

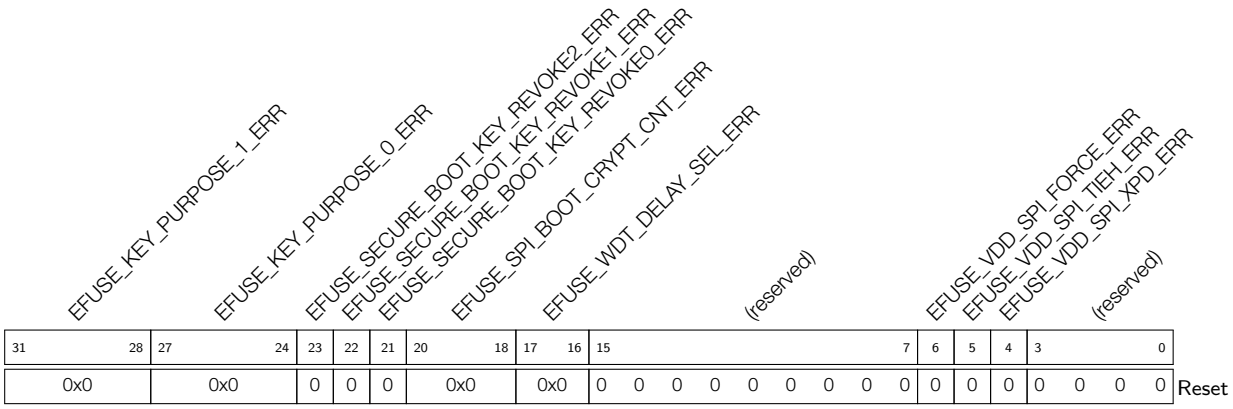
EFUSE_USB_EXCHG_PINS_ERR 若该参数中任意位为 1，表明对应 EFUSE_USB_EXCHG_PINS 中该位的四备份不一致，参数不可靠。(只读)

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 11.25: 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)

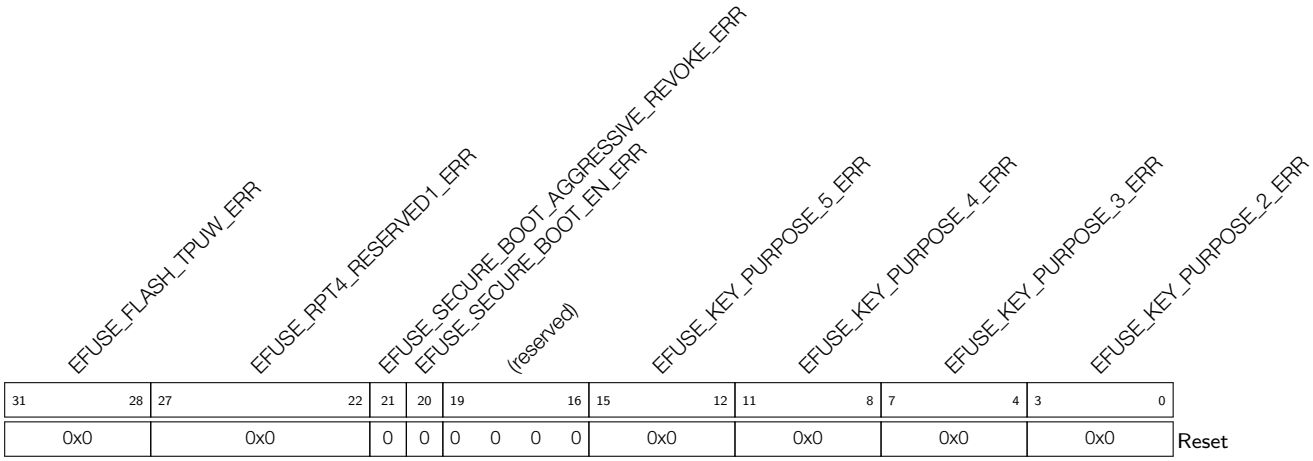
EFUSE_SECURE_BOOT_KEY_REVOKE1_ERR 若该参数中任意位为 1，表明对应 EFUSE_SECURE_BOOT_KEY_REVOKE1 中该位的四备份不一致，参数不可靠。(只读)

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 11.26: 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)

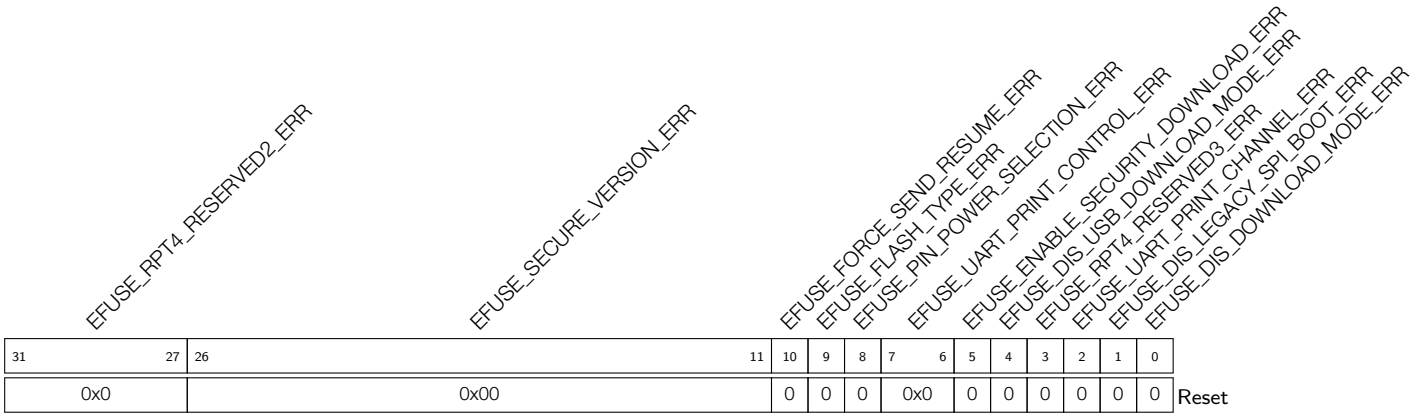
EFUSE_SECURE_BOOT_EN_ERR 若该参数中任意位为 1，表明对应 EFUSE_SECURE_BOOT_EN 中该位的四备份不一致，参数不可靠。(只读)

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 11.27: 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)

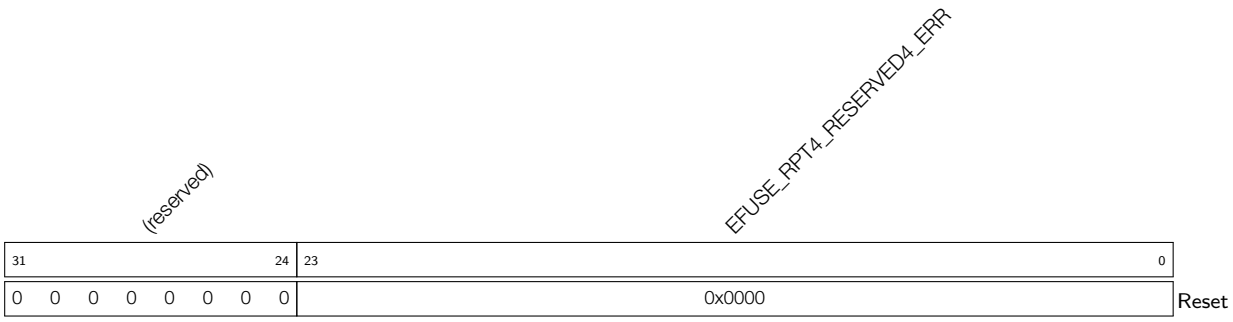
EFUSE_FLASH_TYPE_ERR 若该参数中任意位为 1，表明对应 EFUSE_FLASH_TYPE 中该位的四备份不一致，参数不可靠。(只读)

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)

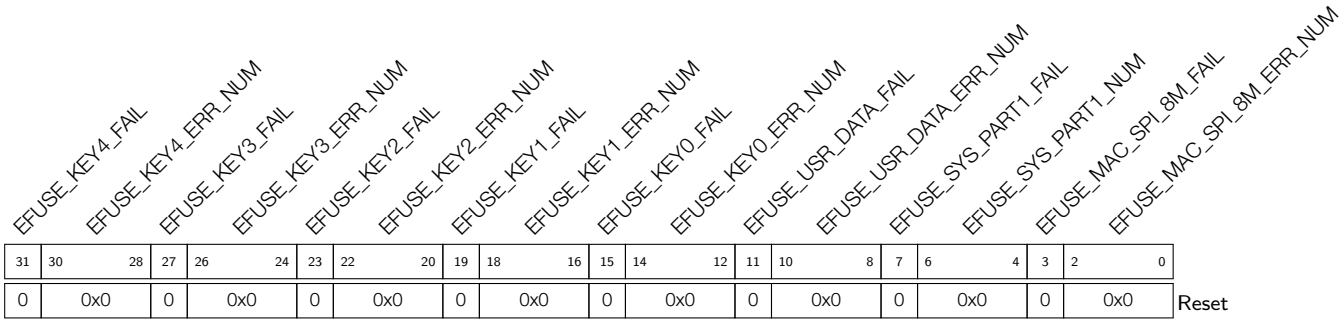
EFUSE_RPT4_RESERVED2_ERR 若该参数中任意位为 1，表明对应 EFUSE_RPT4_RESERVED2 中该位的四备份不一致，参数不可靠。(只读)

Register 11.28: 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 11.29: 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)

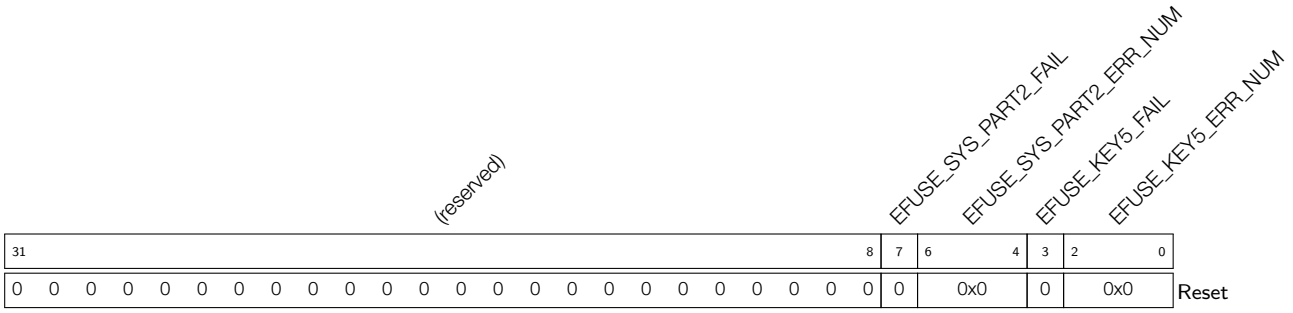
EFUSE_USR_DATA_ERR_NUM 指示 BLOCK3 中的错误字节的个数。(只读)

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 11.30: EFUSE_RD_RS_ERR1_REG (0x01C4)



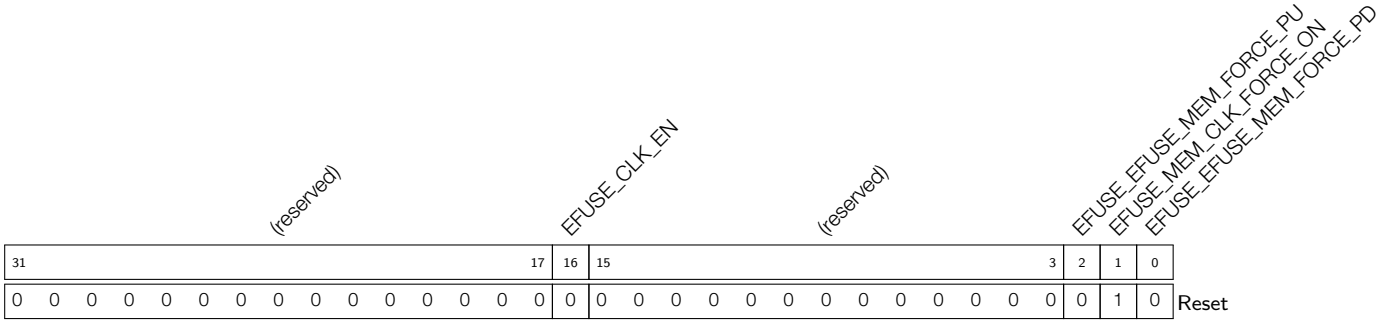
EFUSE_KEY5_ERR_NUM 指示 KEY5 中的错误字节的个数。(只读)

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 11.31: EFUSE_CLK_REG (0x01C8)



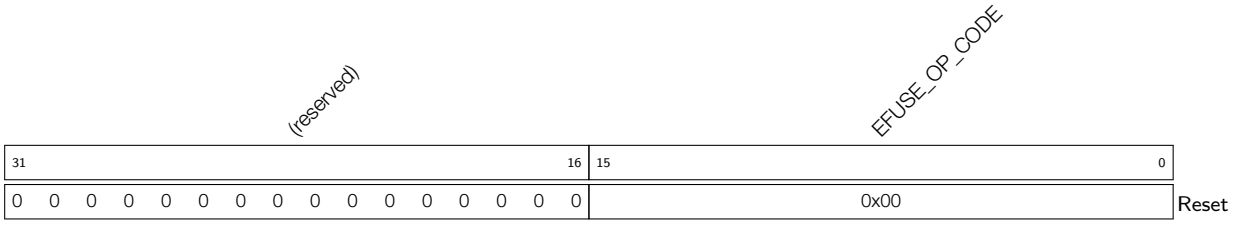
EFUSE_EFUSE_MEM_FORCE_PD 置位强制使 eFuse SRAM 进入低功耗模式。(读/写)

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)

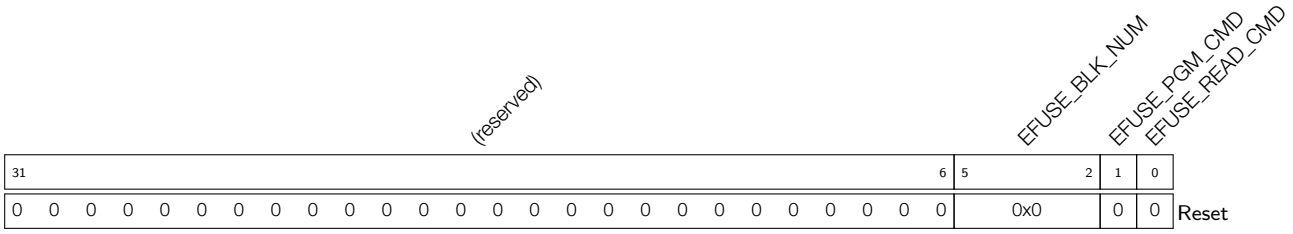
EFUSE_CLK_EN 置位强制使能 eFuse memory 的时钟信号。(读/写)

Register 11.32: EFUSE_CONF_REG (0x01CC)



EFUSE_OP_CODE 0x5A5A：运行烧写指令；0x5AA5：运行读取指令。(读/写)

Register 11.33: EFUSE_CMD_REG (0x01D4)

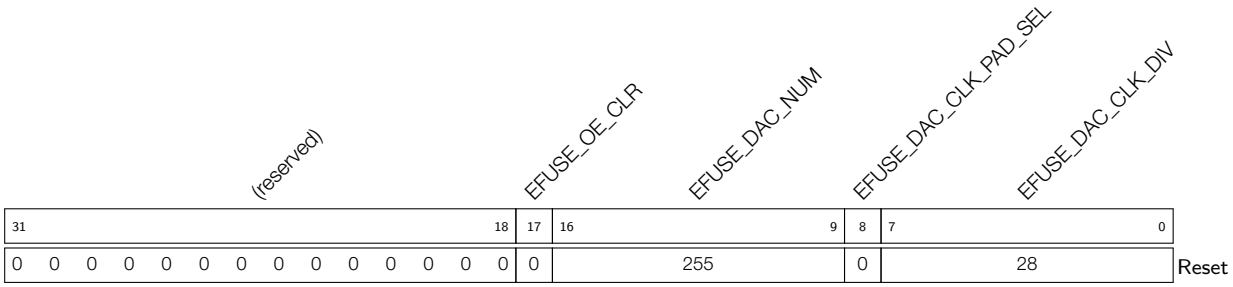


EFUSE_READ_CMD 置位发送读取指令。(读/写)

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 11.34: EFUSE_DAC_CONF_REG (0x01E8)



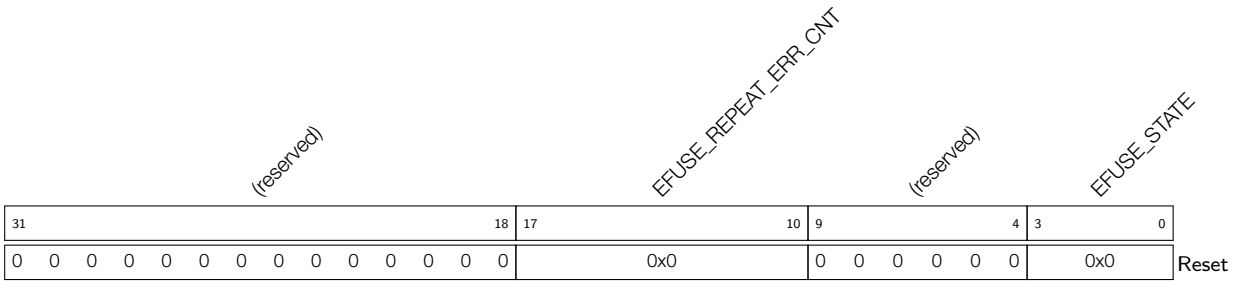
EFUSE_DAC_CLK_DIV 烧写电压的爬升时钟分频系数。(读/写)

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)

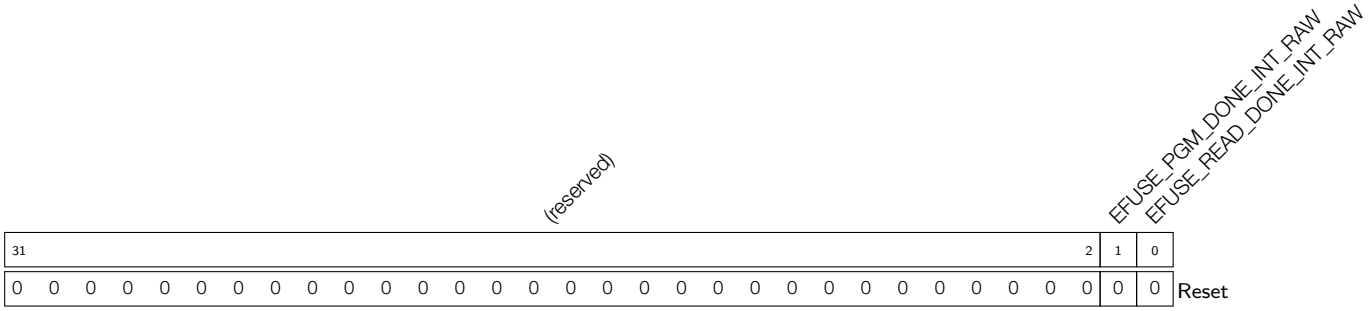
Register 11.35: EFUSE_STATUS_REG (0x01D0)



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)

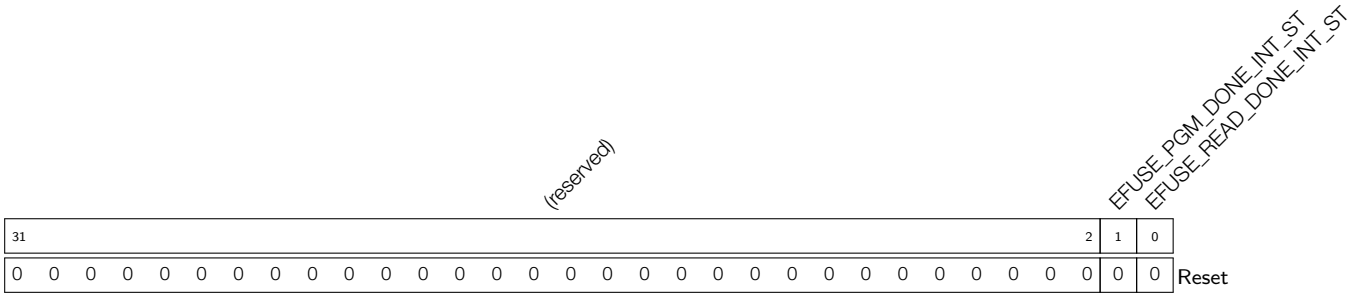
Register 11.36: EFUSE_INT_RAW_REG (0x01D8)



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)

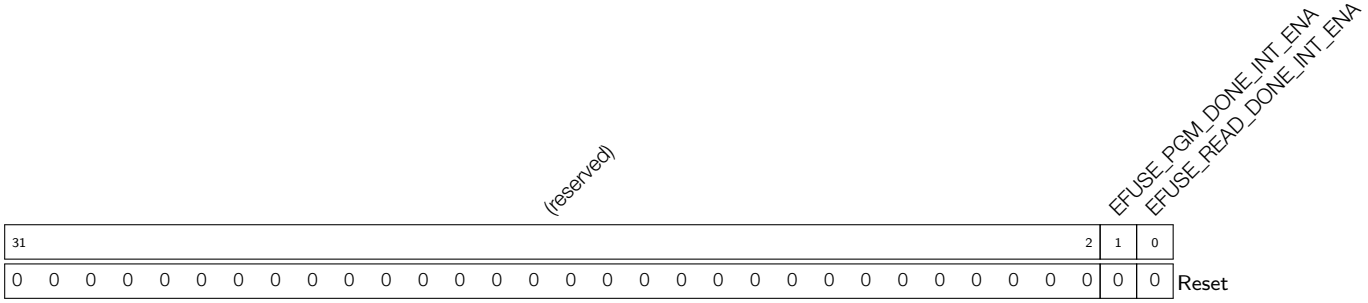
Register 11.37: EFUSE_INT_ST_REG (0x01DC)



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)

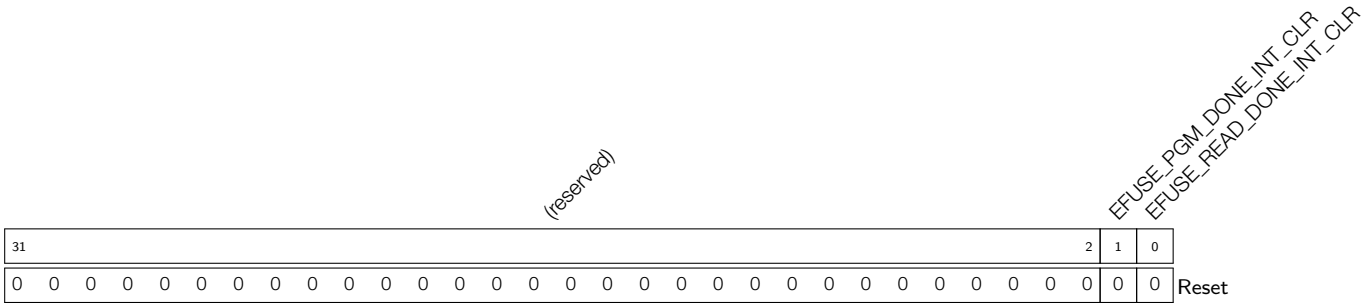
Register 11.38: EFUSE_INT_ENA_REG (0x01E0)



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)

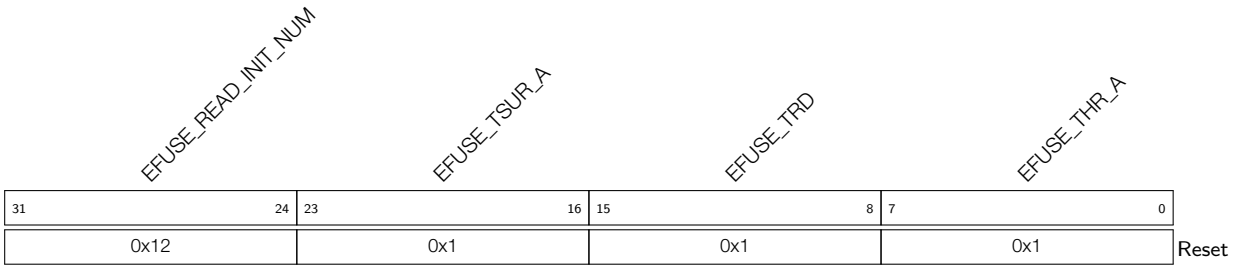
Register 11.39: EFUSE_INT_CLR_REG (0x01E4)



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 11.40: EFUSE_RD_TIM_CONF_REG (0x01EC)



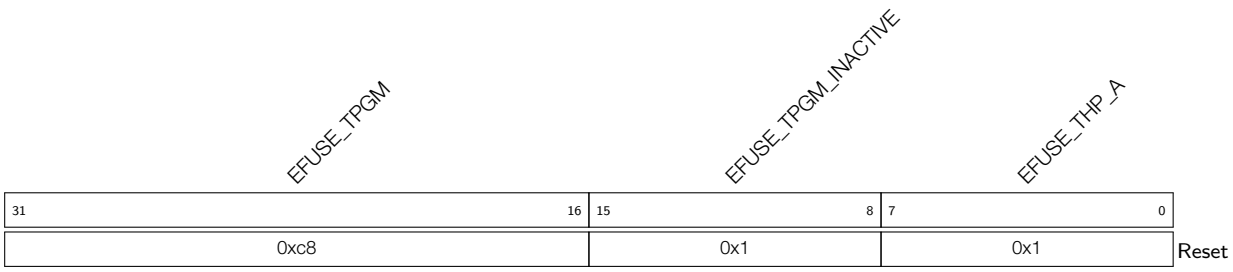
EFUSE_THR_A 配置读取操作的保持时间。(读/写)

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 11.41: EFUSE_WR_TIM_CONF0_REG (0x01F0)

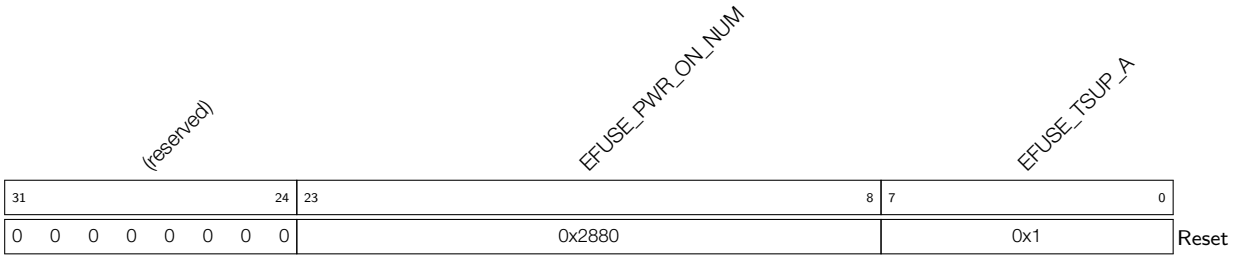


EFUSE_THP_A 配置烧写操作的保持时间。(读/写)

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)

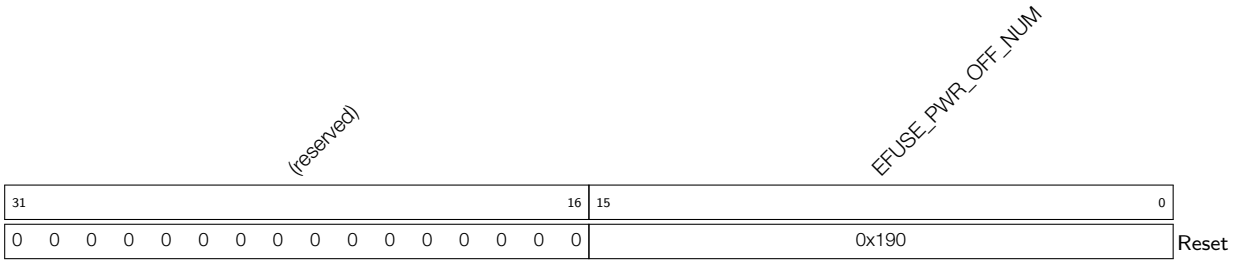
Register 11.42: EFUSE_WR_TIM_CONF1_REG (0x01F4)



EFUSE_TSUP_A 配置烧写操作的建立时间。(读/写)

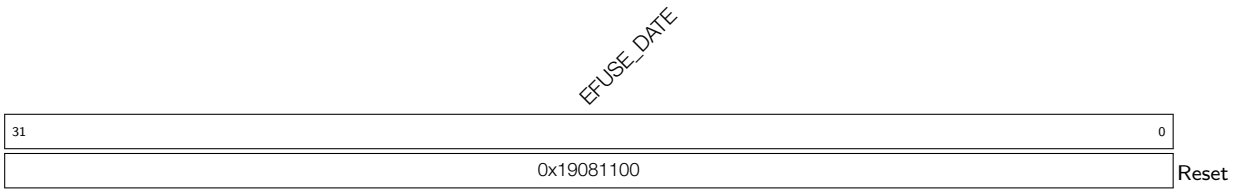
EFUSE_PWR_ON_NUM 配置烧写电压 VDDQ 的上升时间。(读/写)

Register 11.43: 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 11.44: EFUSE_DATE_REG (0x01FC)



EFUSE_DATE 版本控制寄存器。(读/写)

12. I²C 控制器

12.1 概述

I²C (Inter-Integrated Circuit) 总线用于使 ESP32 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。

12.2 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

12.3 I²C 功能描述

12.3.1 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。

12.3.2 I²C 架构

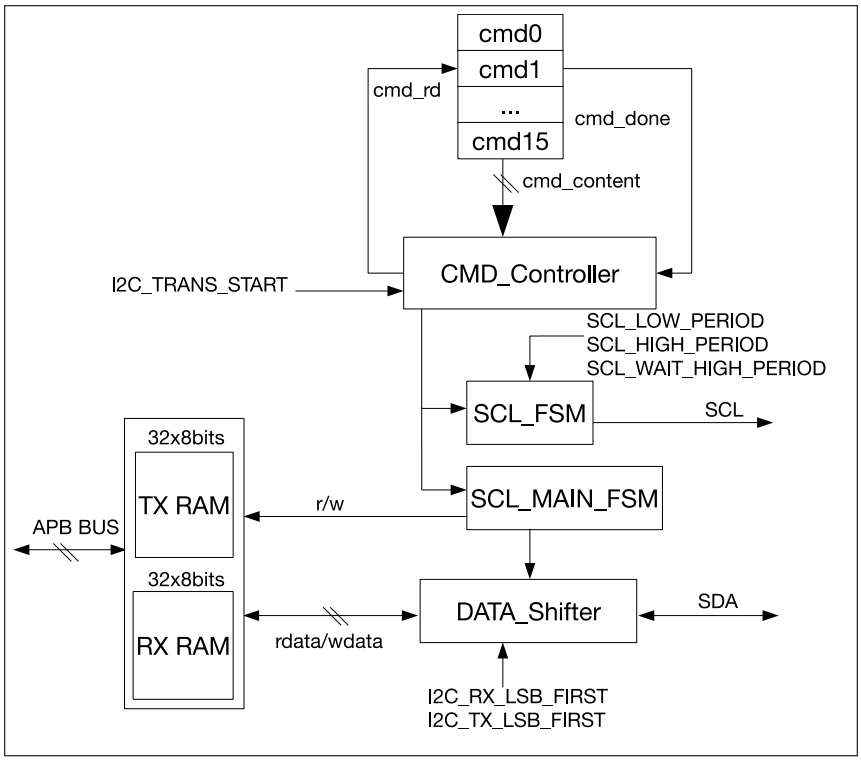


图 12-1. I²C Master 基本架构

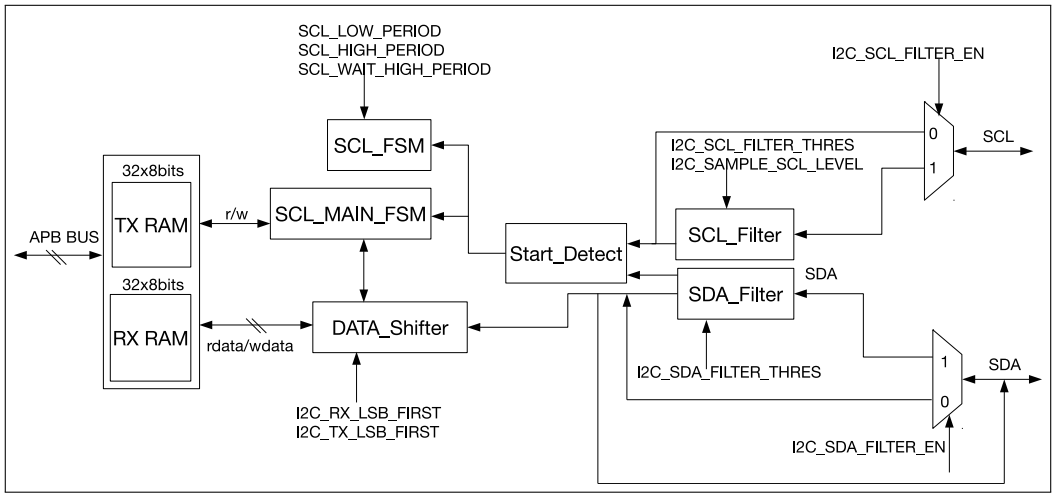


图 12-2. I²C Slave 基本架构

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 12-1 为 I²C Master 基本架构图，图 12-2 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 。

12.3.2.1 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 中的数据并串形输出到 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。

12.3.2.2 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。

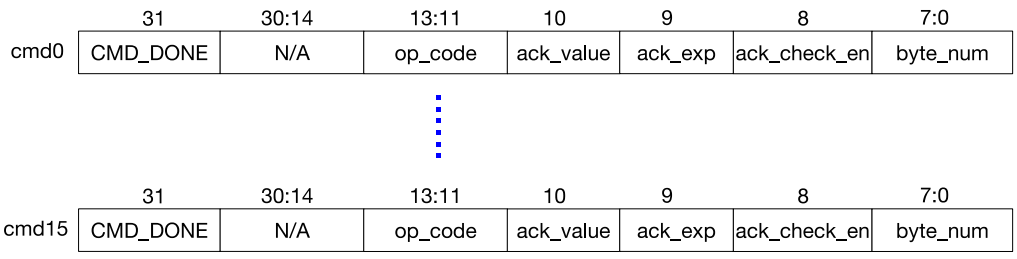


图 12-3. I²C 命令寄存器结构

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 12-3 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。

12.3.2.3 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。

12.3.2.4 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。

12.3.2.5 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。

12.3.2.6 SCL_Filter 和 SDA_Filter

SCL_Filter 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。

12.3.3 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。

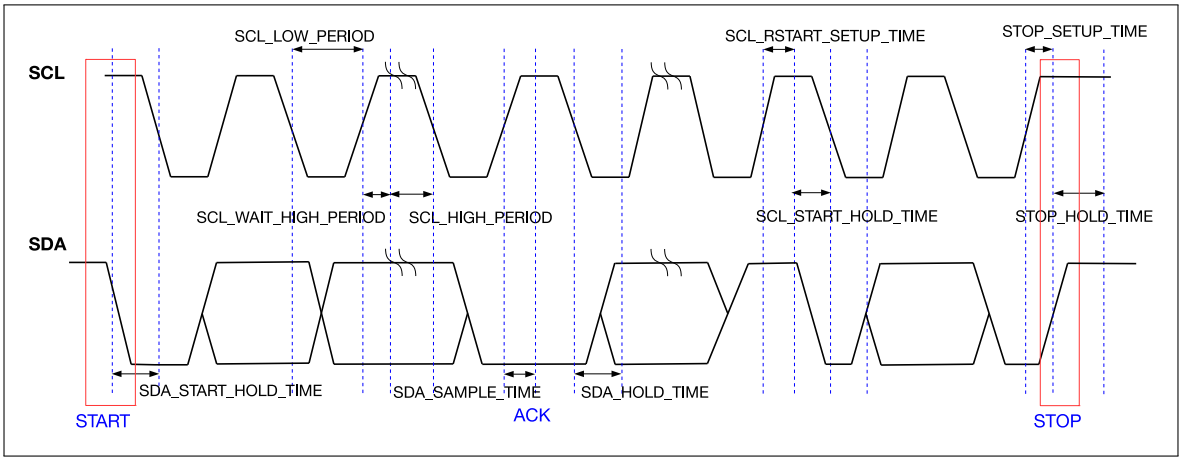


图 12-4. I²C 时序图

图 12-4 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 12-4 中所示的寄存器进行配置。如图12-4 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. 2C_EXT_SCL_HIGH_PERIOD SCL 线拉高后维持高电平的模块时钟周期数。仅控制器工作在主机模式时有意义。当 SCL 线在 I2C_SCL_WAIT_HIGH_PERIOD + 1 个模块时钟内完成上拉，则 SCL 线的频率为：

$$f_{scl} = \frac{f_{\text{I2C\_CLK}}}{\text{I2C\_SCL\_LOW\_PERIOD}+1+\text{I2C\_SCL\_HIGH\_PERIOD}+\text{I2C\_SCL\_WAIT\_HIGH\_PERIOD}}$$

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。

12.4 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。

12.4.1 I²C 主机写入从机，7-bit 寻址，单次命令序列

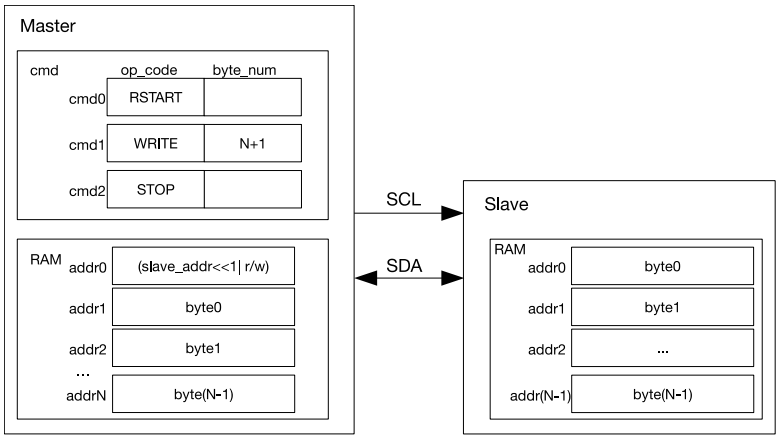


图 12-5. I²C Master 写 7-bit 寻址的 Slave

图 12-5 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 12-5 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。

12.4.2 I²C 主机写入从机，10-bit 寻址，单次命令序列

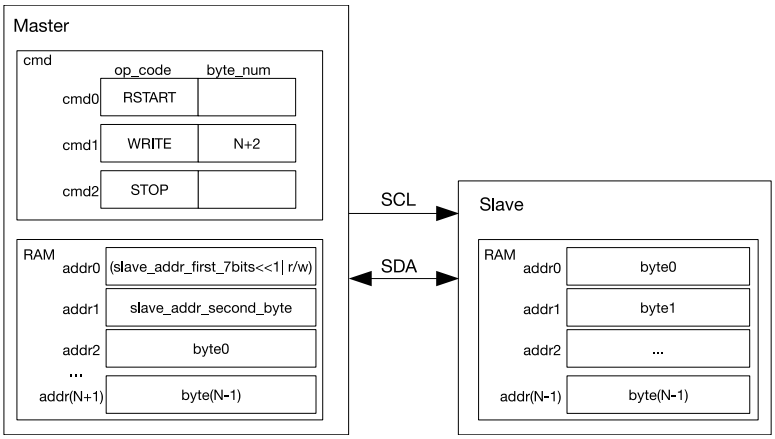


图 12-6. I²C Master 写 10-bit 寻址的 Slave

假设从机地址为 SLV_ADDR 。ESP32 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 12-6 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。

12.4.3 I²C 主机写入从机，7-bit 双地址寻址，单次命令序列

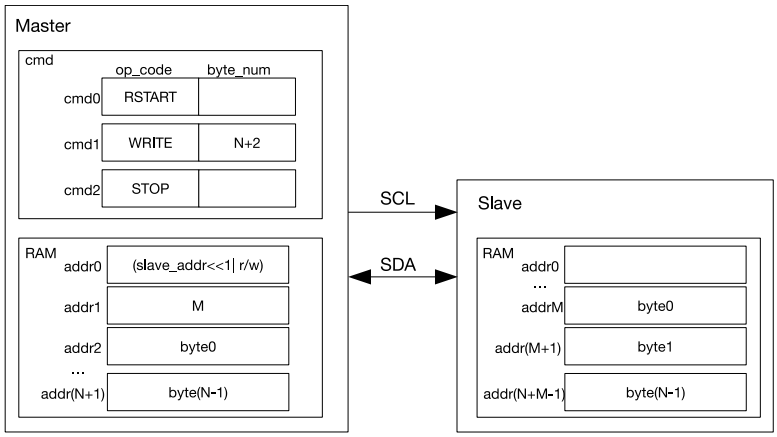


图 12-7. I²C Master 写 7-bit 寻址 Slave 的 M 地址 RAM

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 12-7 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。

12.4.4 I²C 主机写入从机，7-bit 寻址，多次命令序列

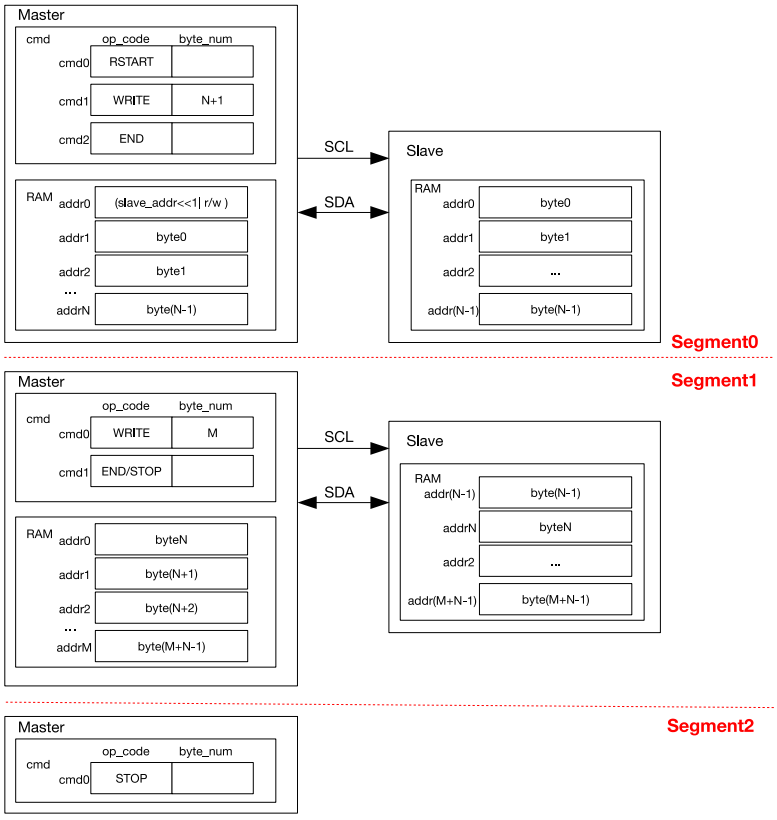


图 12-8. I²C Master 分段写 7-bit 寻址的 Slave

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 12-8 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 I²C 外设有所不同，具体请参考各个 I²C 设备的技术规格书。

12.4.5 I²C 主机读取从机，7-bit 寻址，单次命令序列

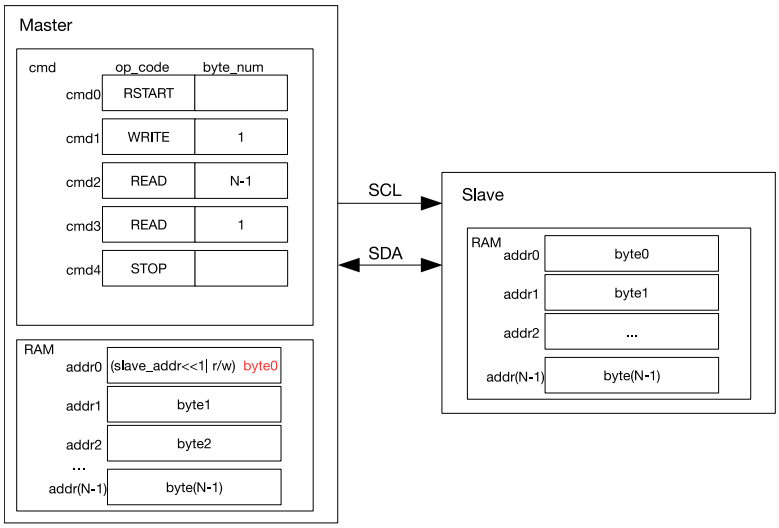


图 12-9. I²C Master 读 7-bit 寻址的 Slave

图 12-9 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 12-9 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）。

12.4.6 I²C 主机读取从机，10-bit 寻址，单次命令序列

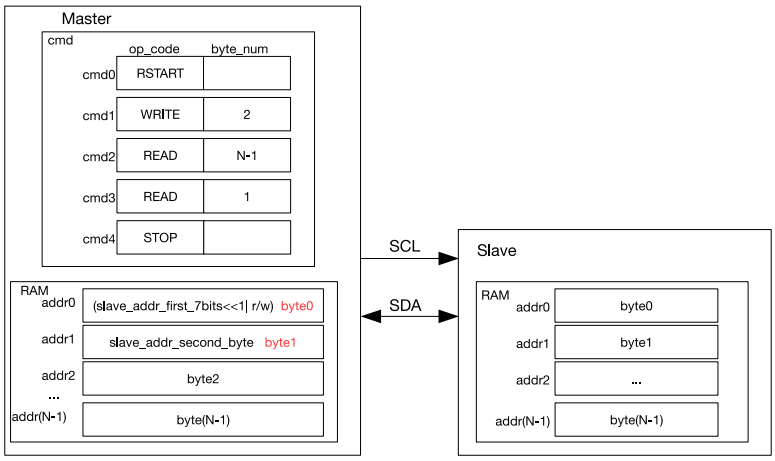


图 12-10. I²C Master 读 10-bit 寻址的 Slave

图 12-10 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 12.4.2 小节的相同。

12.4.7 I²C 主机读取从机，7-bit 双寻址，单次命令序列

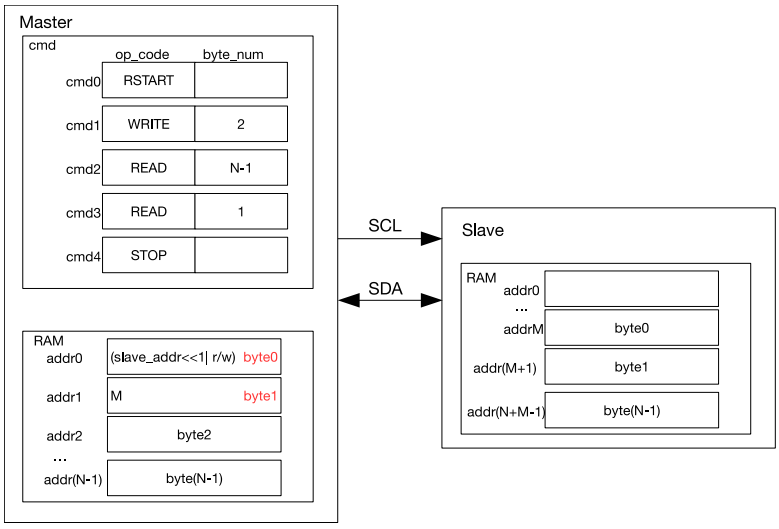


图 12-11. I²C Master 从 7-bit 寻址 Slave 的 M 地址读取 N 个数据

图 12-11 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。

12.4.8 I²C 主机读取从机，7-bit 寻址，多次命令序列

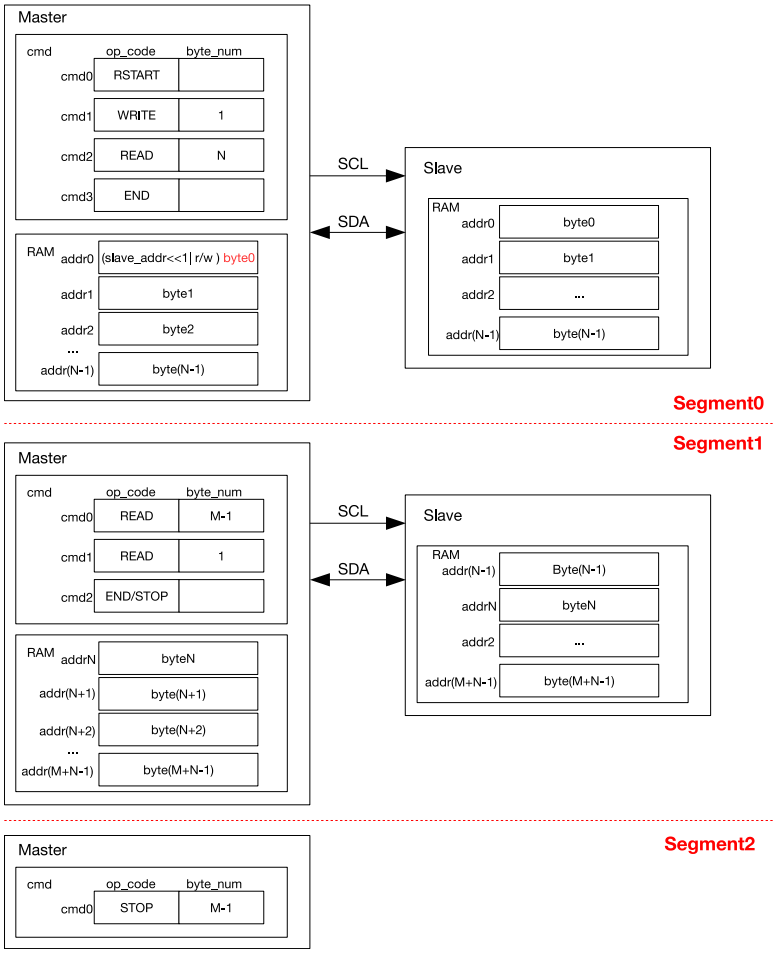


图 12-12. I²C Master 分段读 7-bit 寻址的 Slave

图 12-12 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。

12.5 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。

12.6 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。

12.7 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 12-1 所示。更多信息，请访问章节 1 系统和存储器。

表 12-1. I²C 控制器基地址

	访问总线	基地址
I²C0	PeriBUS1	0x3F413000
	PeriBUS2	0x60013000
I²C1	PeriBUS1	0x3F427000
	PeriBUS2	0x60027000

12.8 寄存器列表

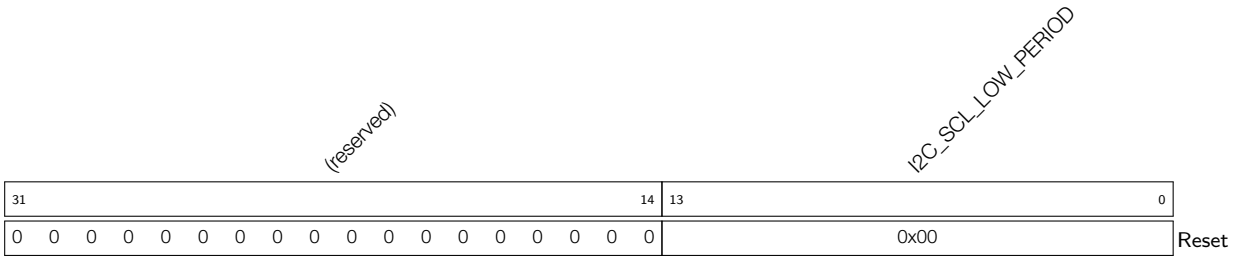
请注意，下表的地址指相对于 I²C 控制器基地址的偏移量（相对地址）。更多有关 I²C 控制器基地址的信息，请前往 12.7 章节。

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

名称	描述	地址	访问
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
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

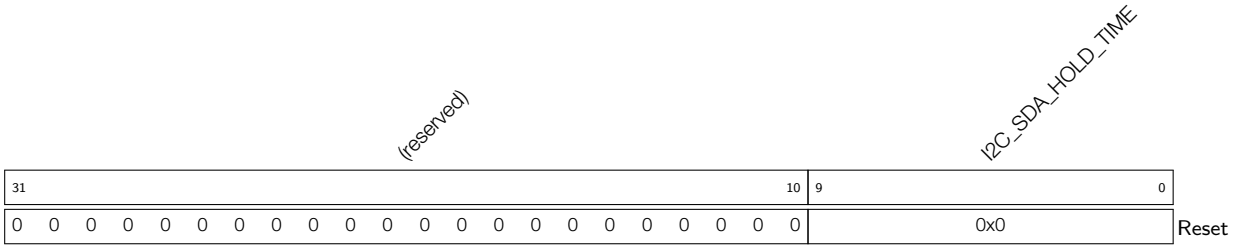
12.9 寄存器

Register 12.1: I2C_SCL_LOW_PERIOD_REG (0x0000)



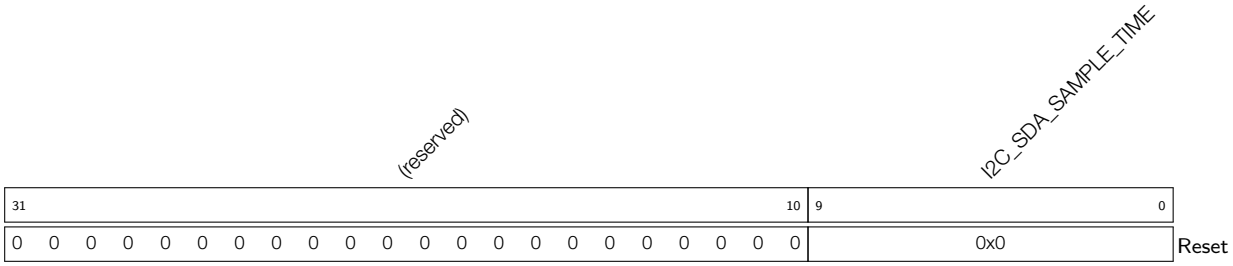
I2C_SCL_LOW_PERIOD 用于配置 SCL 低电平的保持时间，以 I2C 模块时钟周期数为单位。(读/写)

Register 12.2: I2C_SDA_HOLD_REG (0x0030)



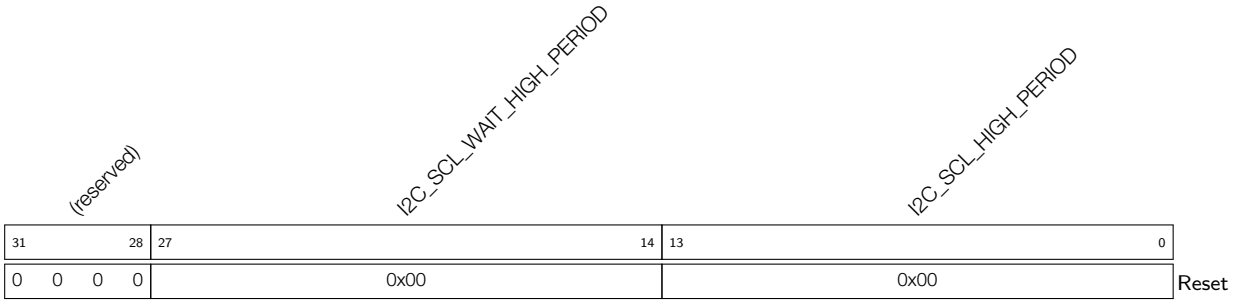
I2C_SDA_HOLD_TIME 用于配置 SCL 下降沿后的数据保持时间，以 I2C 模块时钟周期数为单位。(读/写)

Register 12.3: 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)

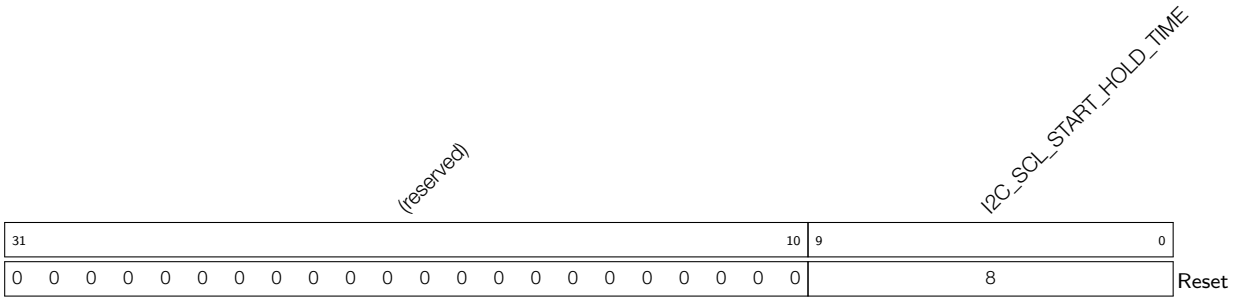
Register 12.4: I2C_SCL_HIGH_PERIOD_REG (0x0038)



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)

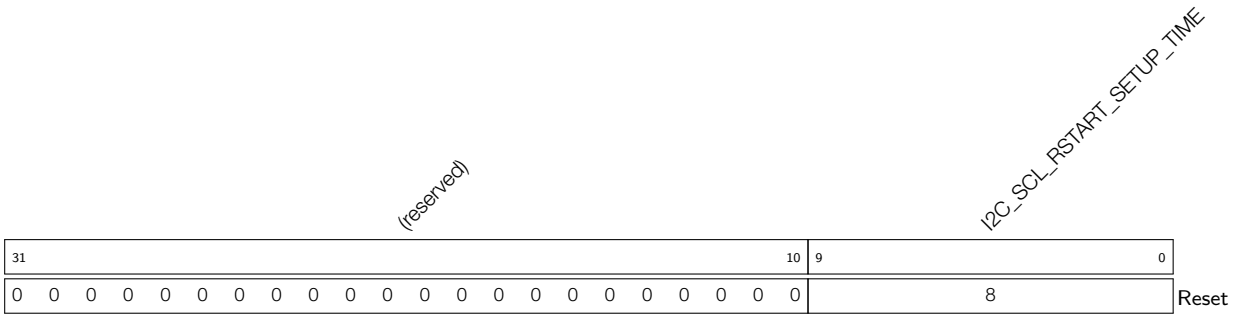
I2C_SCL_WAIT_HIGH_PERIOD 用于配置 SCL_FSM 等待 SCL 在主机模式下翻转至高电平的时间，以 I2C 模块时钟周期数为单位。(读/写)

Register 12.5: I2C_SCL_START_HOLD_REG (0x0040)



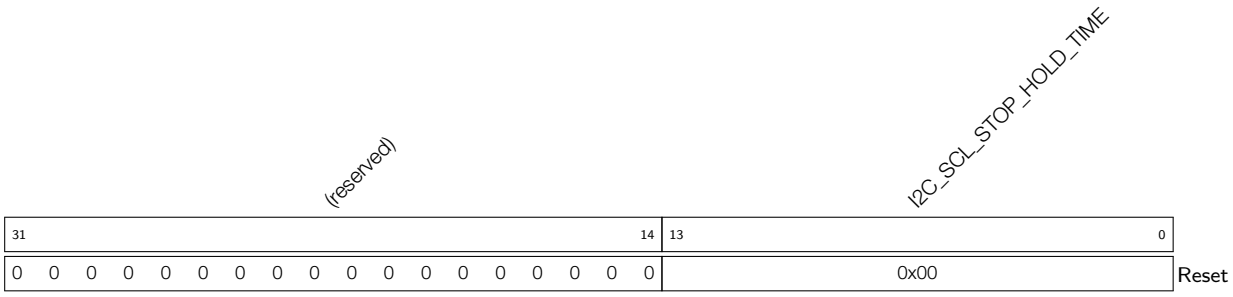
I2C_SCL_START_HOLD_TIME 配置 START 命令产生时 SDA 下降沿和 SCL 下降沿的间隔时间，以 I2C 模块时钟周期数为单位。(读/写)

Register 12.6: I2C_SCL_RSTART_SETUP_REG (0x0044)



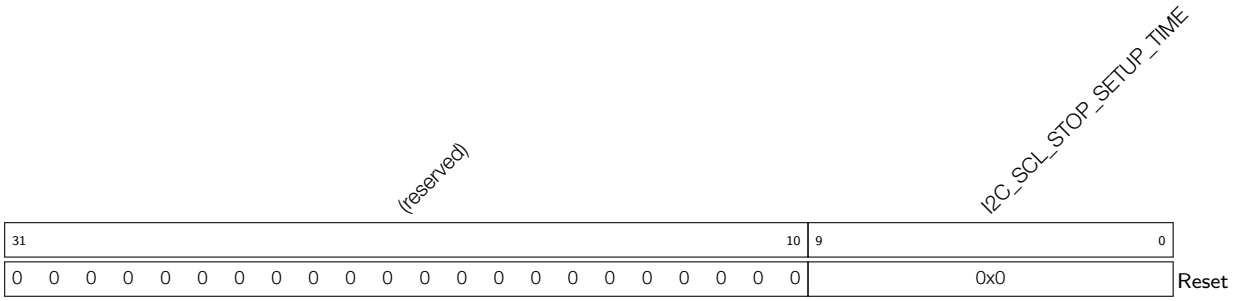
I2C_SCL_RSTART_SETUP_TIME 配置 RESTART 命令产生时 SCL 上升沿和 SDA 下降沿的间隔时间，以 I2C 模块的时钟周期数为单位。(读/写)

Register 12.7: I2C_SCL_STOP_HOLD_REG (0x0048)



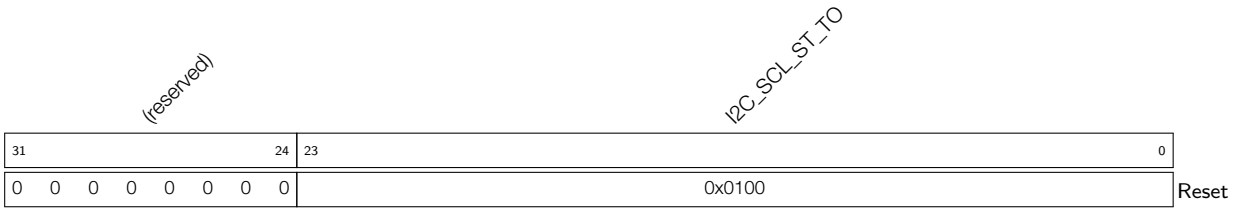
I2C_SCL_STOP_HOLD_TIME 配置 STOP 命令后的延迟，以 I2C 模块时钟周期数为单位。(读/写)

Register 12.8: I2C_SCL_STOP_SETUP_REG (0x004C)



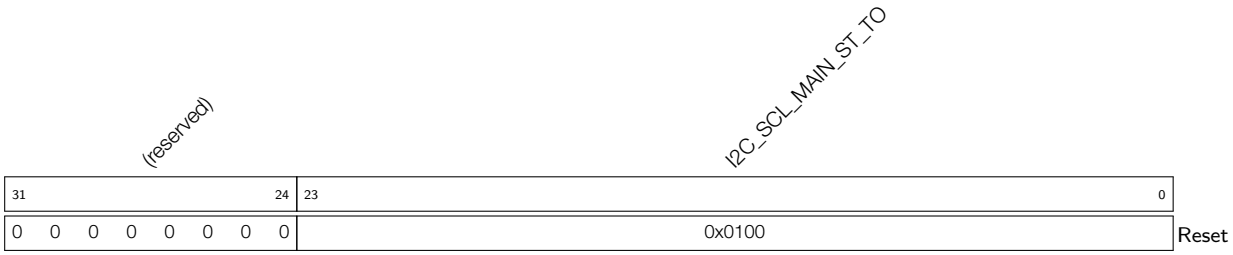
I2C_SCL_STOP_SETUP_TIME 配置 SCL 上升沿和 SDA 上升沿的间隔时间，以 I2C 模块时钟周期数为单位。(读/写)

Register 12.9: I2C_SCL_ST_TIME_OUT_REG (0x0098)



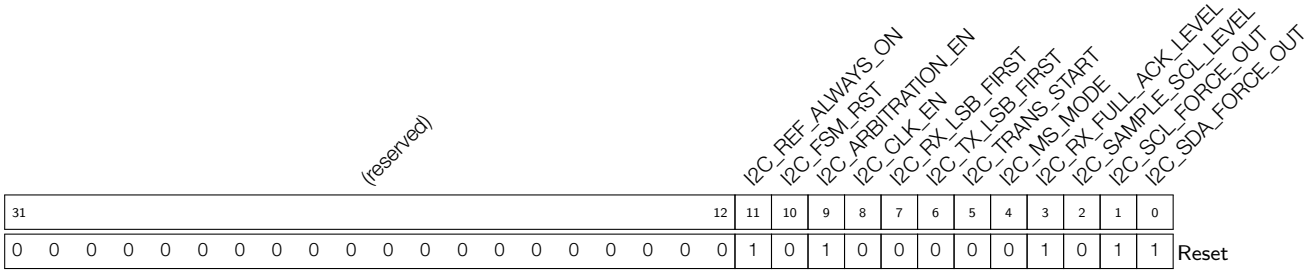
I2C_SCL_ST_TO SCL_FSM 状态不变的阈值。(读/写)

Register 12.10: 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 12.11: 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)

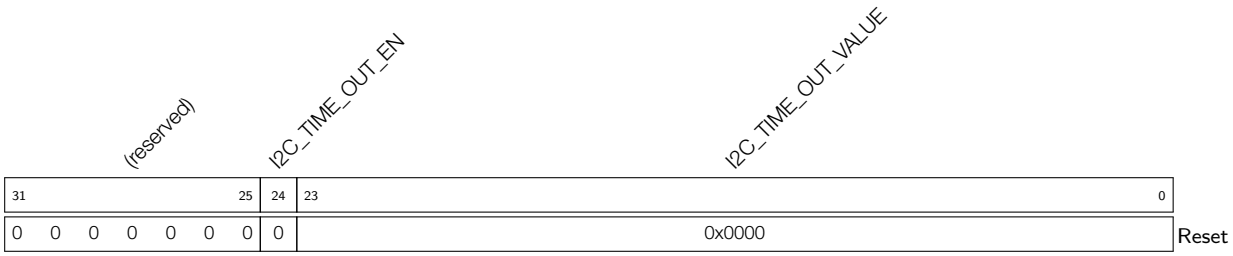
I2C_CLK_EN 保留 (读/写)

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)

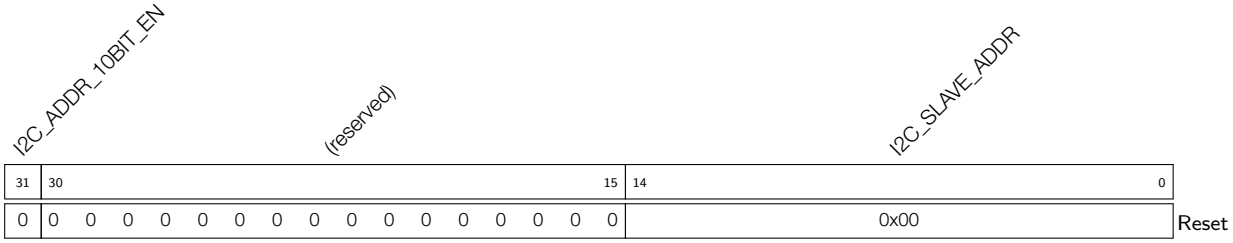
Register 12.12: I2C_TO_REG (0x000C)



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)

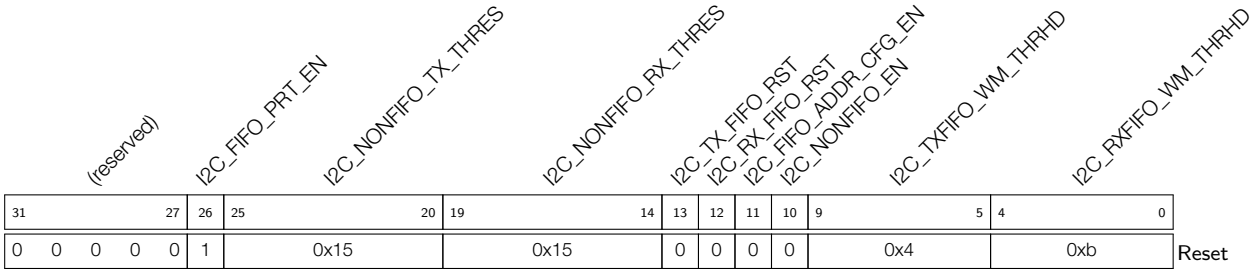
Register 12.13: I2C_SLAVE_ADDR_REG (0x0010)



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 12.14: 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)

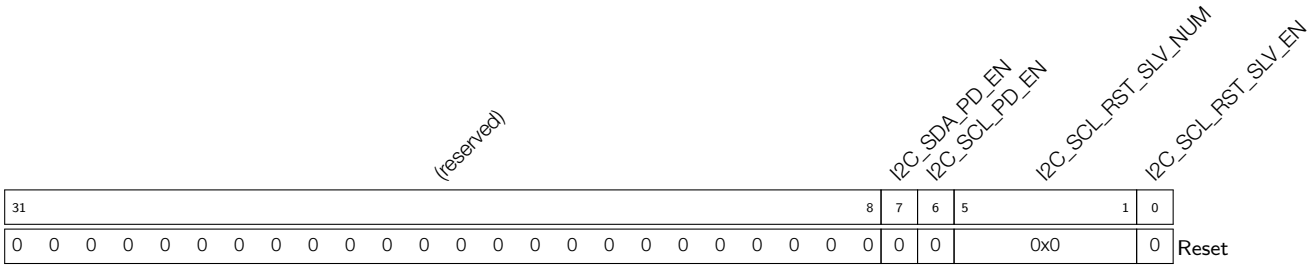
I2C_TX_FIFO_RST 置位此位，复位 TX FIFO 。(读/写)

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 12.15: I2C_SCL_SP_CONF_REG (0x00A0)



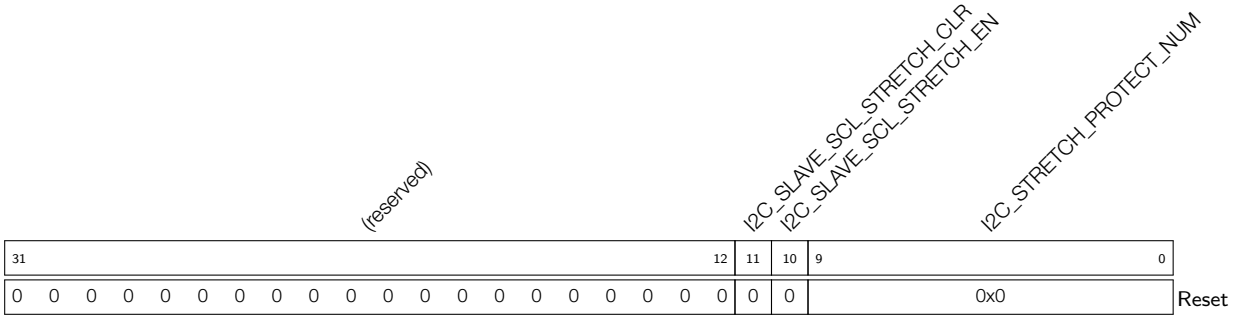
I2C_SCL_RST_SLV_EN I2C 主机处于空闲状态时，置位此位发送 SCL 脉冲。脉冲数量为 I2C_SCL_RST_SLV_NUM[4:0]。(读/写)

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 12.16: I2C_SCL_STRETCH_CONF_REG (0x00A4)

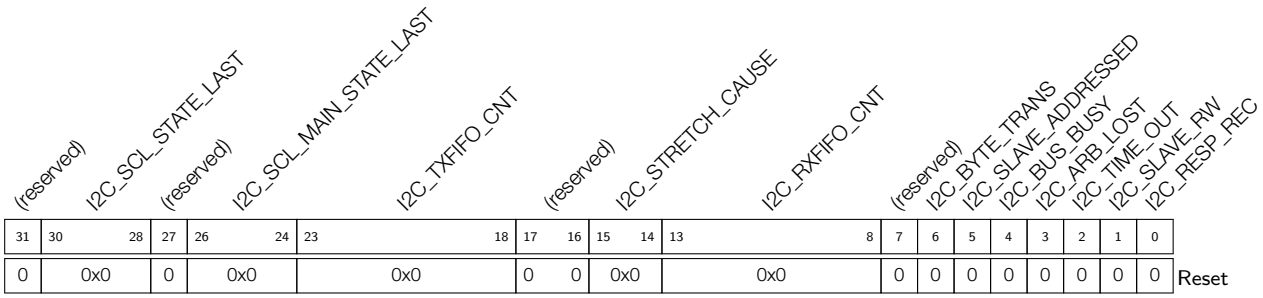


I2C_STRETCH_PROTECT_NUM 配置 I2C 从机延长 SCL 低电平时间，以时钟周期为单位。(读/写)

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 12.17: 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)

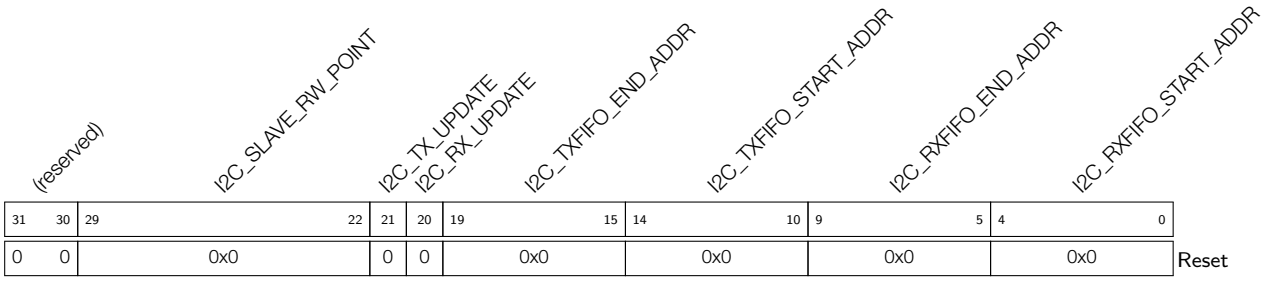
I2C_STRETCH_CAUSE 从机模式下延长 SCL 低电平时间的原因。0：I2C 开始读取数据时延长 SCL 的低电平时间。1：从机模式下 TX FIFO 为空时延长 SCL 的低电平时间。2：从机模式下 RX FIFO 为满时延长 SCL 的低电平时间。(只读)

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 12.18: 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)

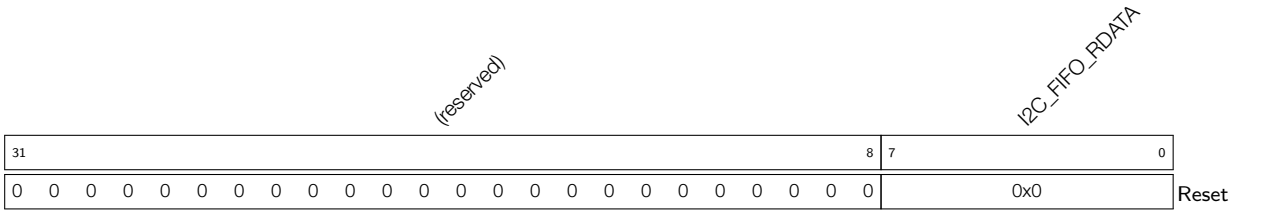
I2C_TXFIFO_END_ADDR 最后发送数据的偏移地址，如寄存器 I2C_NONFIFO_TX_THRES 所述。该值在 I2C_TX_SEND_EMPTY_INT 中断或 I2C_TRANS_COMPLETE_INT 中断产生时更新。(只读)

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)

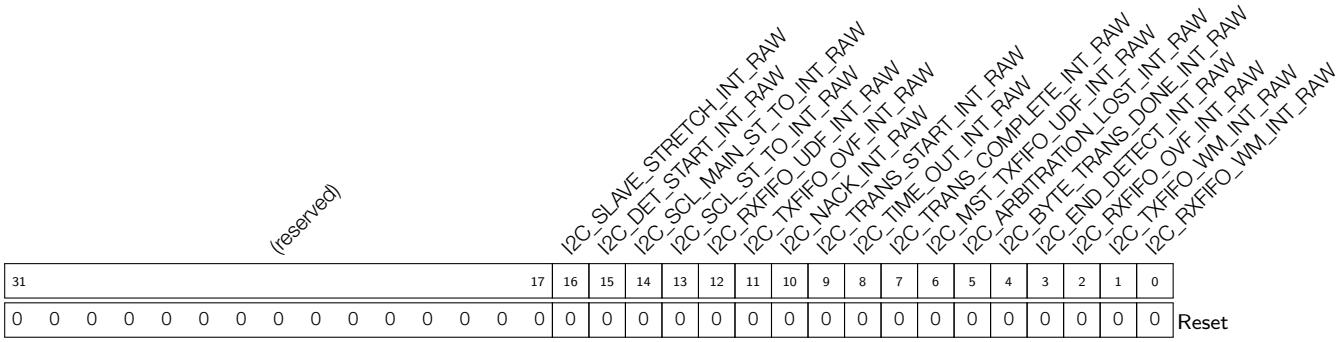
I2C_SLAVE_RW_POINT 从机模式下接收的数据。(只读)

Register 12.19: I2C_DATA_REG (0x001C)



I2C_FIFO_RDATA RX FIFO 读取数据的值。(只读)

Register 12.20: 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)

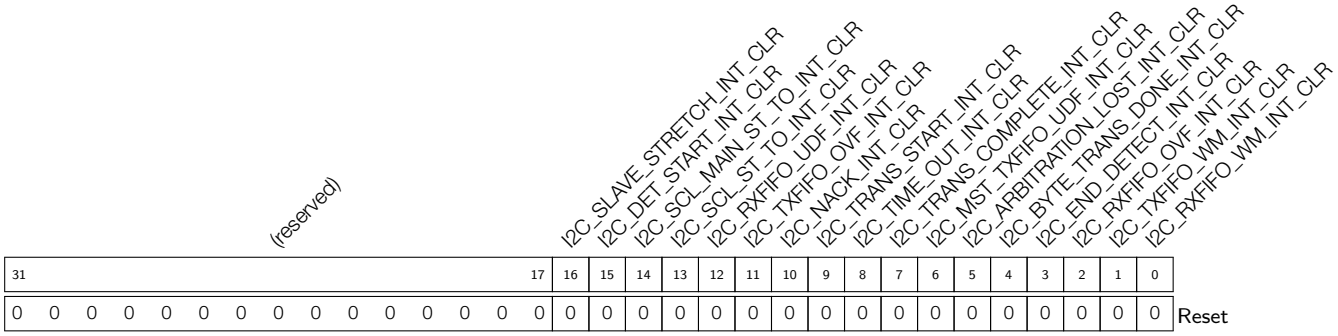
I2C_SCL_ST_TO_INT_RAW I2C_SCL_ST_TO_INT 的原始中断位。(只读)

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 12.21: 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)

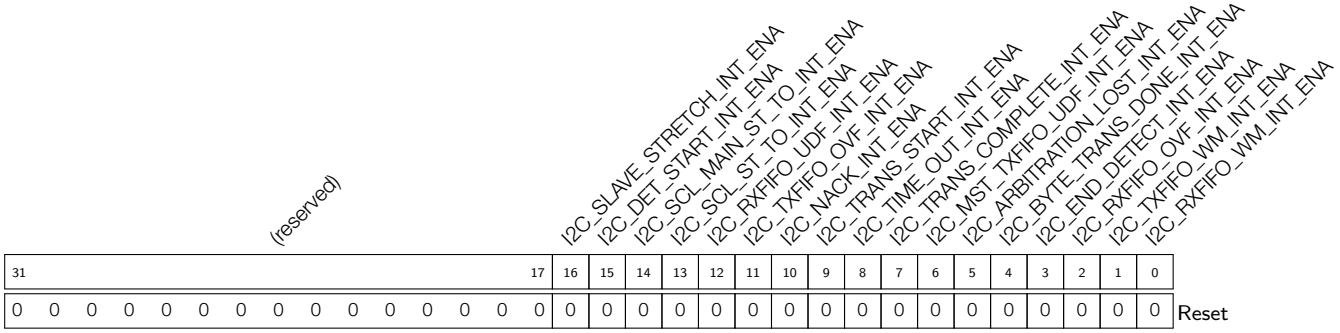
I2C_SCL_ST_TO_INT_CLR 置位此位，清除 I2C_SCL_ST_TO_INT 中断。(只写)

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 12.22: 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)

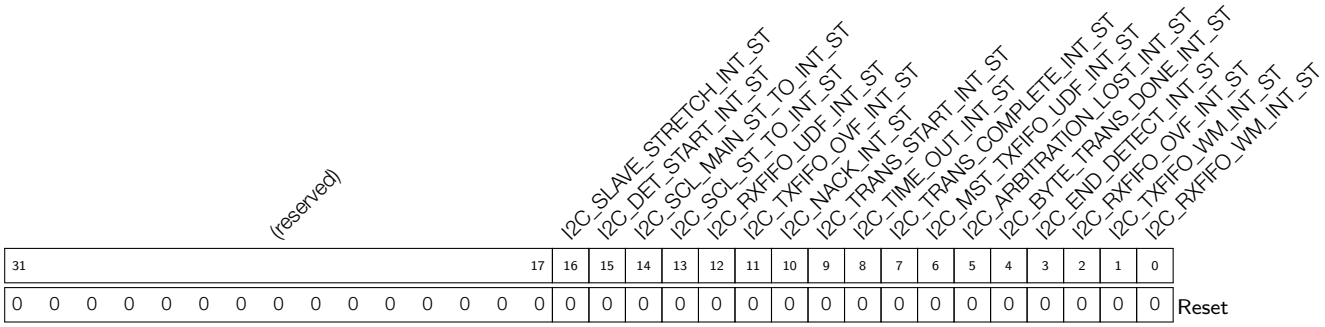
I2C_SCL_ST_TO_INT_ENA I2C_SCL_ST_TO_INT 的原始中断位。(读/写)

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 12.23: 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)

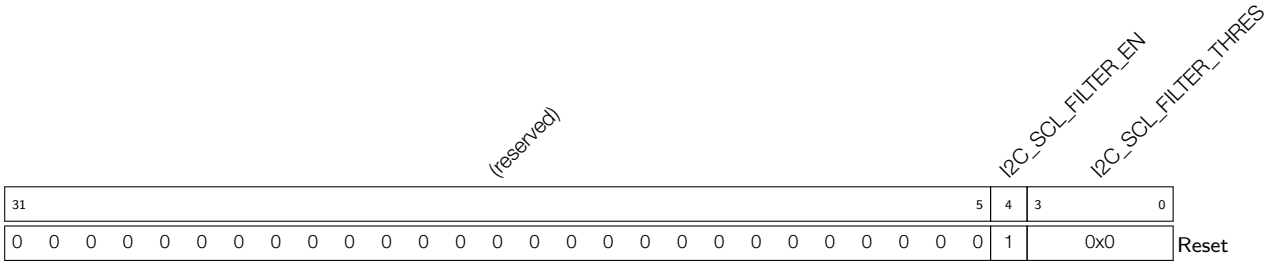
I2C_SCL_ST_TO_INT_ST I2C_SCL_ST_TO_INT 的屏蔽中断状态位。(只读)

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)

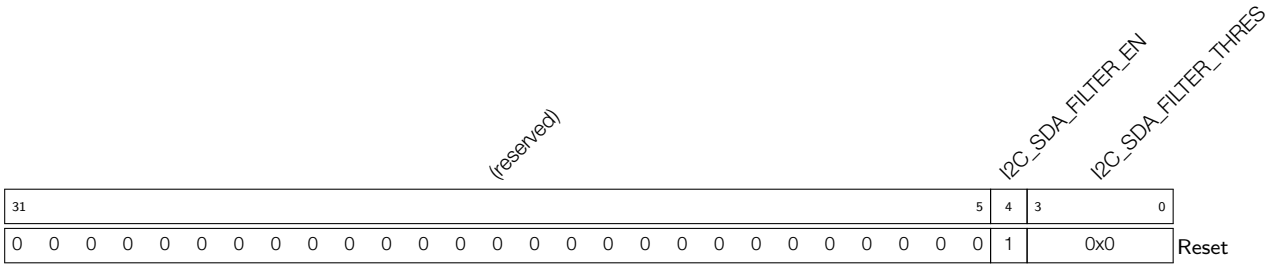
Register 12.24: I2C_SCL_FILTER_CFG_REG (0x0050)



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)

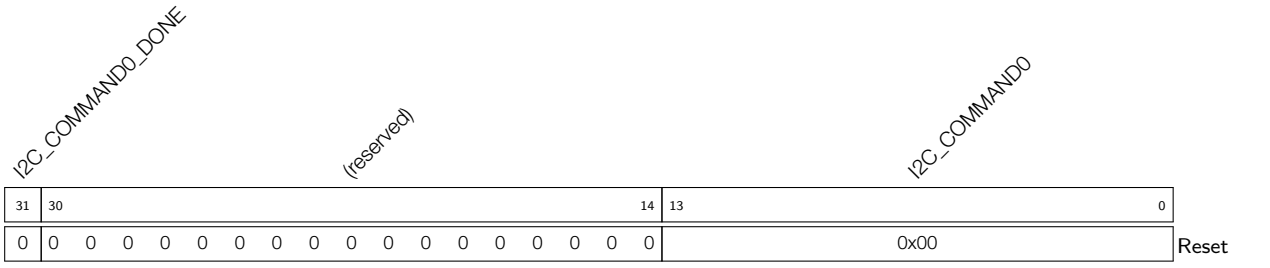
Register 12.25: I2C_SDA_FILTER_CFG_REG (0x0054)



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)

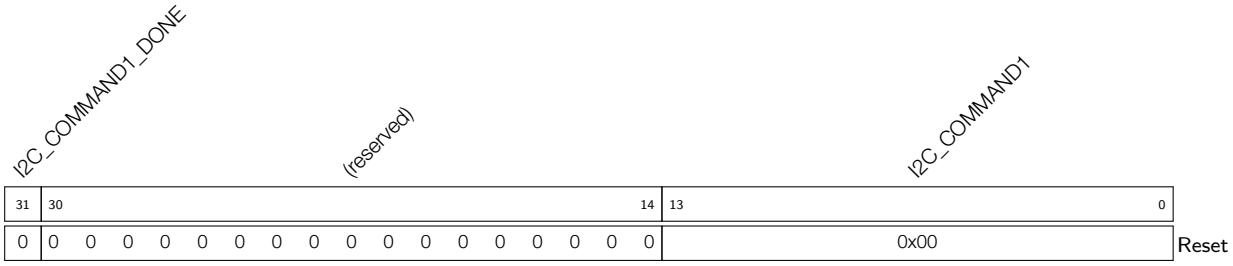
Register 12.26: I2C_COMD0_REG (0x0058)



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)

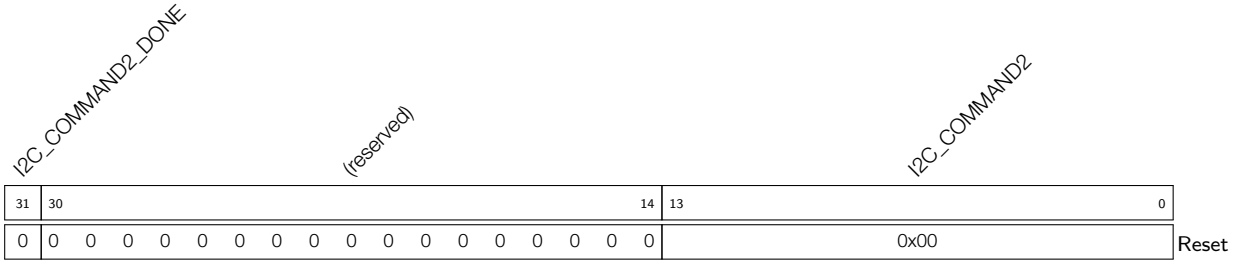
Register 12.27: I2C_COMD1_REG (0x005C)



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)

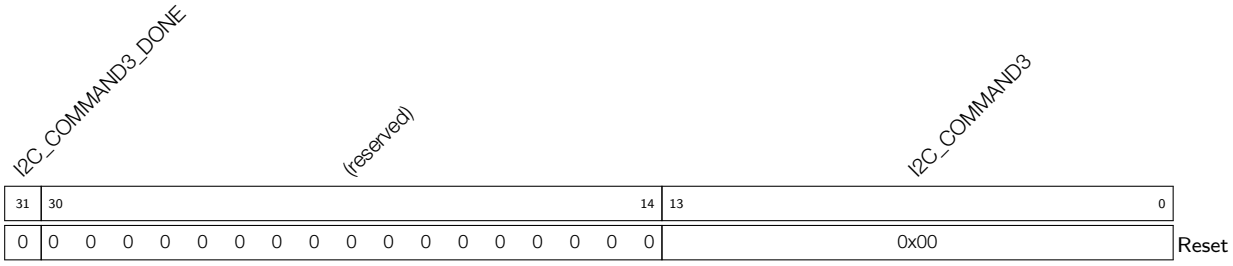
Register 12.28: I2C_COMD2_REG (0x0060)



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)

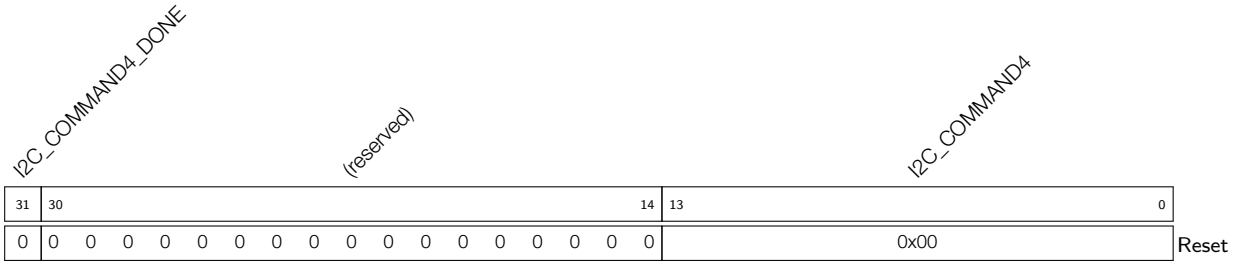
Register 12.29: I2C_COMD3_REG (0x0064)



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)

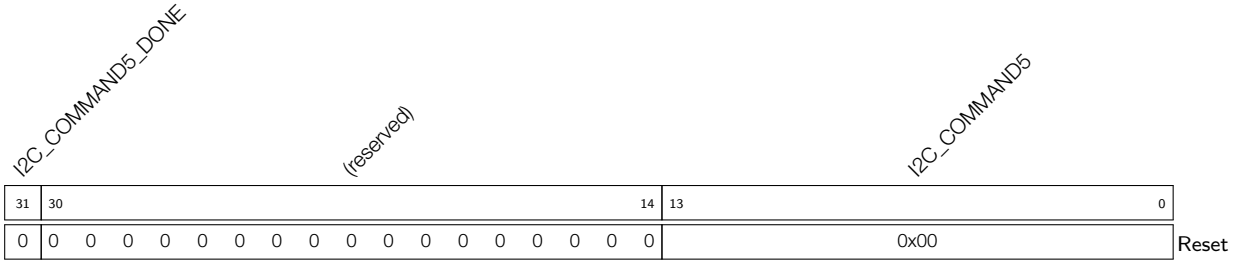
Register 12.30: I2C_COMD4_REG (0x0068)



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)

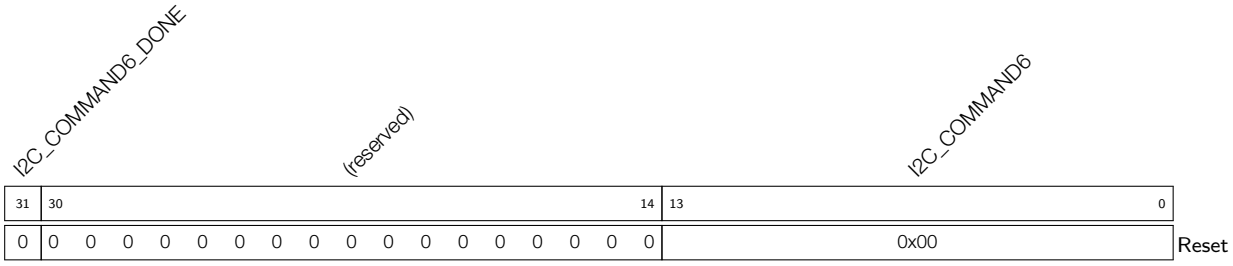
Register 12.31: I2C_COMD5_REG (0x006C)



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)

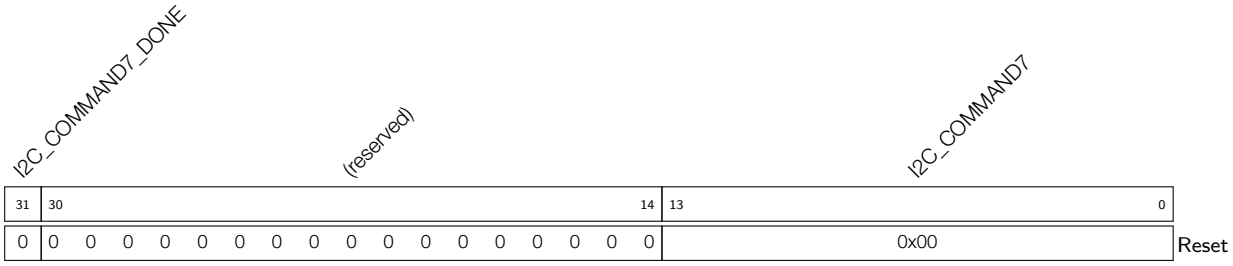
Register 12.32: I2C_COMD6_REG (0x0070)



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)

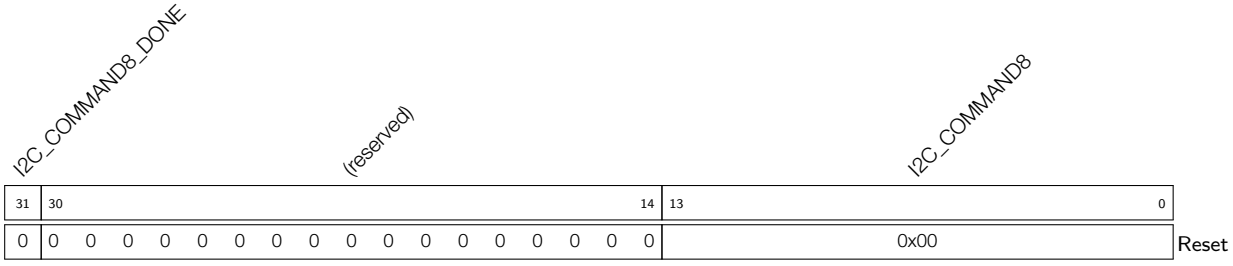
Register 12.33: I2C_COMD7_REG (0x0074)



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)

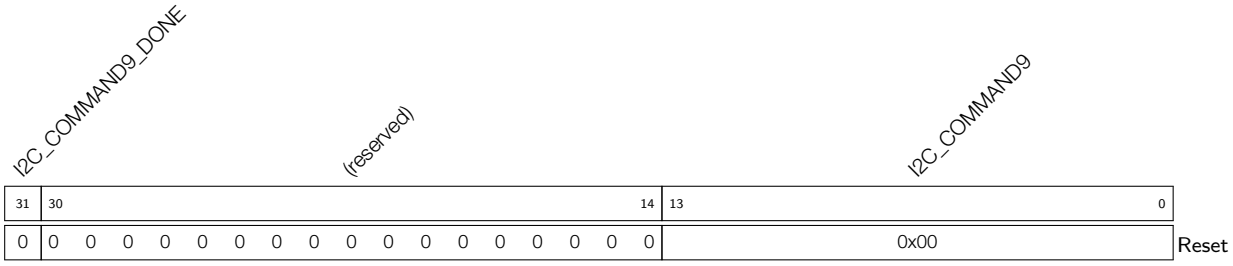
Register 12.34: I2C_COMD8_REG (0x0078)



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)

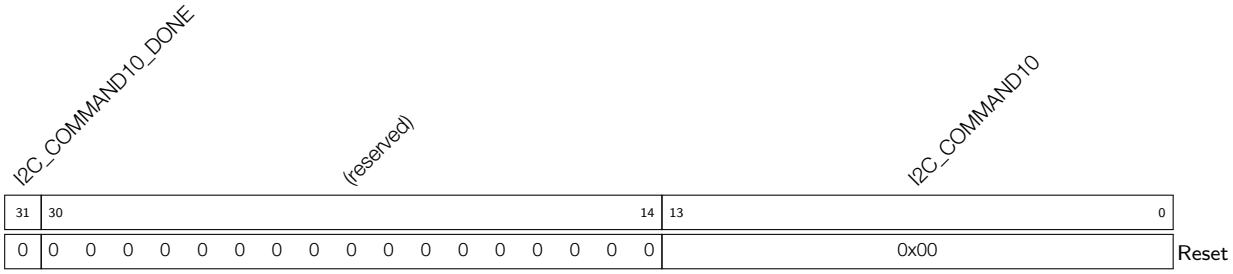
Register 12.35: I2C_COMD9_REG (0x007C)



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)

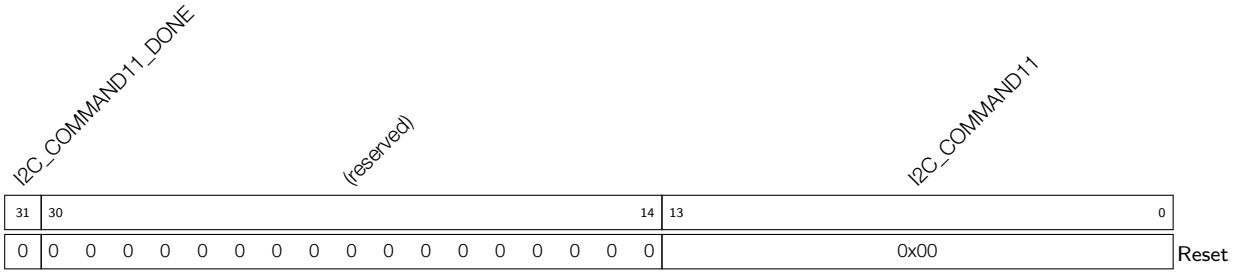
Register 12.36: I2C_COMD10_REG (0x0080)



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)

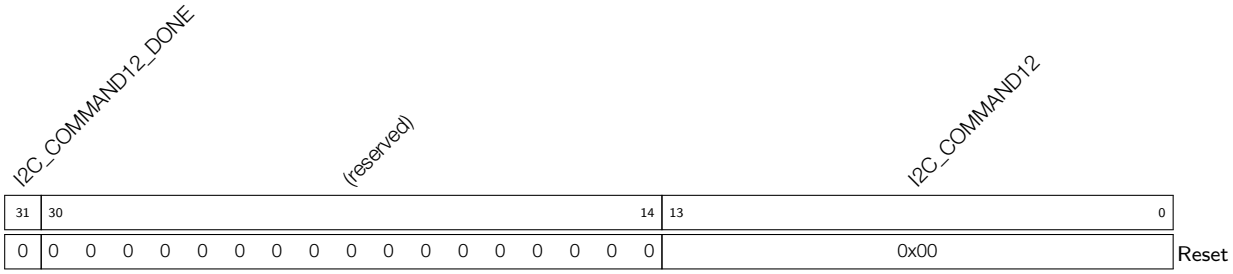
Register 12.37: I2C_COMD11_REG (0x0084)



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)

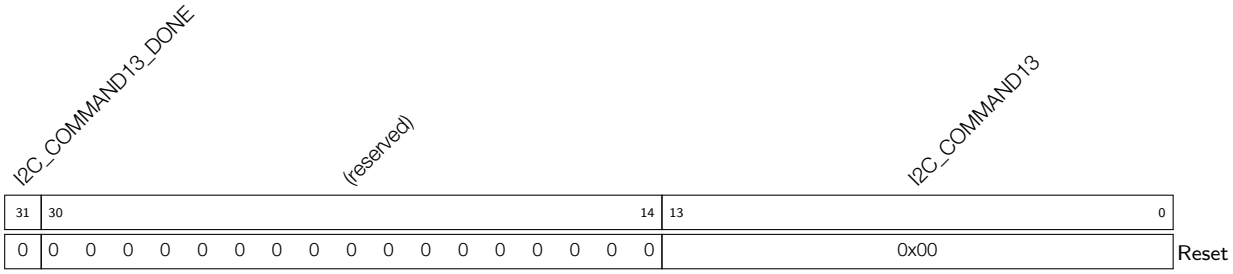
Register 12.38: I2C_COMD12_REG (0x0088)



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)

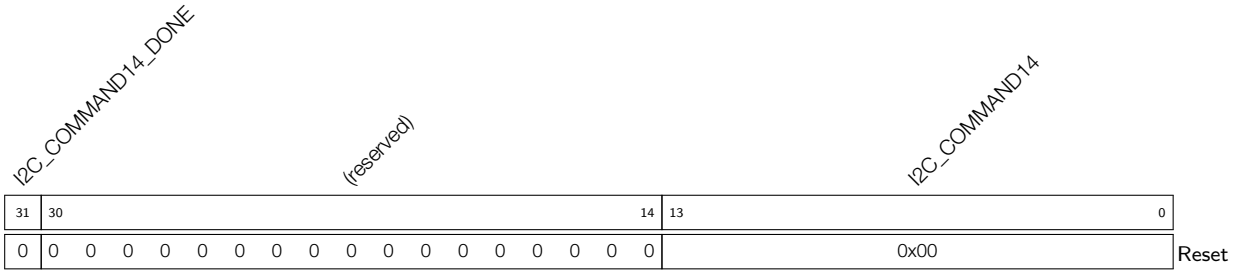
Register 12.39: I2C_COMD13_REG (0x008C)



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)

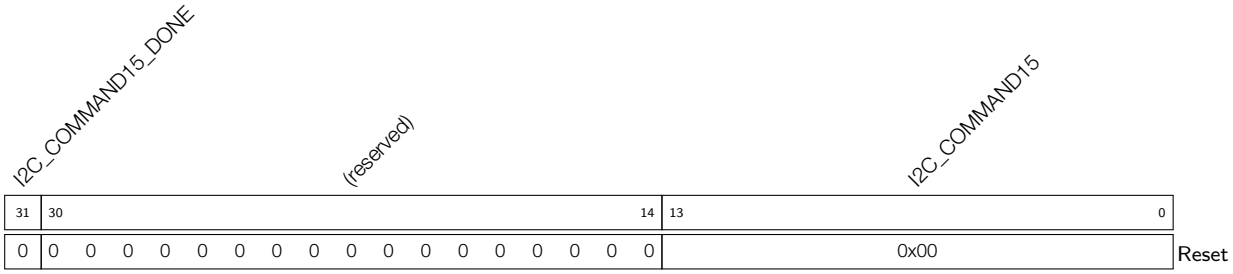
Register 12.40: I2C_COMD14_REG (0x0090)



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)

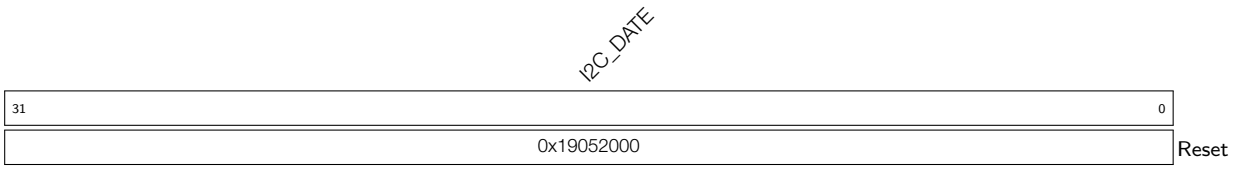
Register 12.41: I2C_COMD15_REG (0x0094)



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 12.42: I2C_DATE_REG (0x00F8)



I2C_DATE 版本控制寄存器。(读/写)

13. AES 加速器

13.1 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。

13.2 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

13.3 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 13-1。

表 13-1. 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 13.4 章节和 13.5 章节。

注意：

ESP32-S2 的数字签名和片外存储器加密与解密模块也会调用 AES 加速器。此时，用户无法正常访问 AES 加速器。

13.4 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 13-2。

表 13-2. 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 13-3 所示：

表 13-3. 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。

13.4.1 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。

13.4.2 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 13-4 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 13-4. 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 13-5、表 13-6、表 13-7 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 13-5. 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 13-6. 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 13-7. 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。

13.4.3 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。

13.5 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 13-8。

表 13-8. 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 13-9 所示：

表 13-9. 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。

13.5.1 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 13-10 所示。

表 13-10. TEXT-PADDING

Function : TEXT-PADDING( )
Input : X, bit string. **Output** : Y = **TEXT-PADDING**(X), whose length is the nearest integral multiples of 128 bits.
Steps Let us assume that X is a data-stream that can be split into n parts as following: $X = X_1 \| \| X_2 \| \| \cdots \| \| X_{n-1} \| \| X_n$ Here, the lengths of $X_1, X_2, \cdots, X_{n-1}$ all 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**($X$) = $X_1 \| \| X_2 \| \| \cdots \| \| X_{n-1} \| \| X_n^*$ = $X \| \| 0^{128-t}$

13.5.2 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 13-11 所示。结果数据也遵从相同的存放规则，在此不多做介绍。

表 13-11. 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。

13.5.3 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。

13.5.4 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。

13.5.5 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 13-11 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。

13.5.6 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 13-8。
   - 初始化寄存器 AES_BLOCK_NUM_REG，请参照章节 13.5.4。
   - 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。

13.5.7 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 13.5.4。
   - 初始化寄存器 AES_AAD_BLOCK_NUM_REG，请参照章节 13.6.4。
   - 初始化寄存器 AES_REMAINDER_BIT_NUM_REG，请参照章节 13.6.5。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 13-9。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 13-9。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。

13.6 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 13-1 显示了 ESP32-S2 内置 AES 加速器实现 GCM 加密算法的流程。

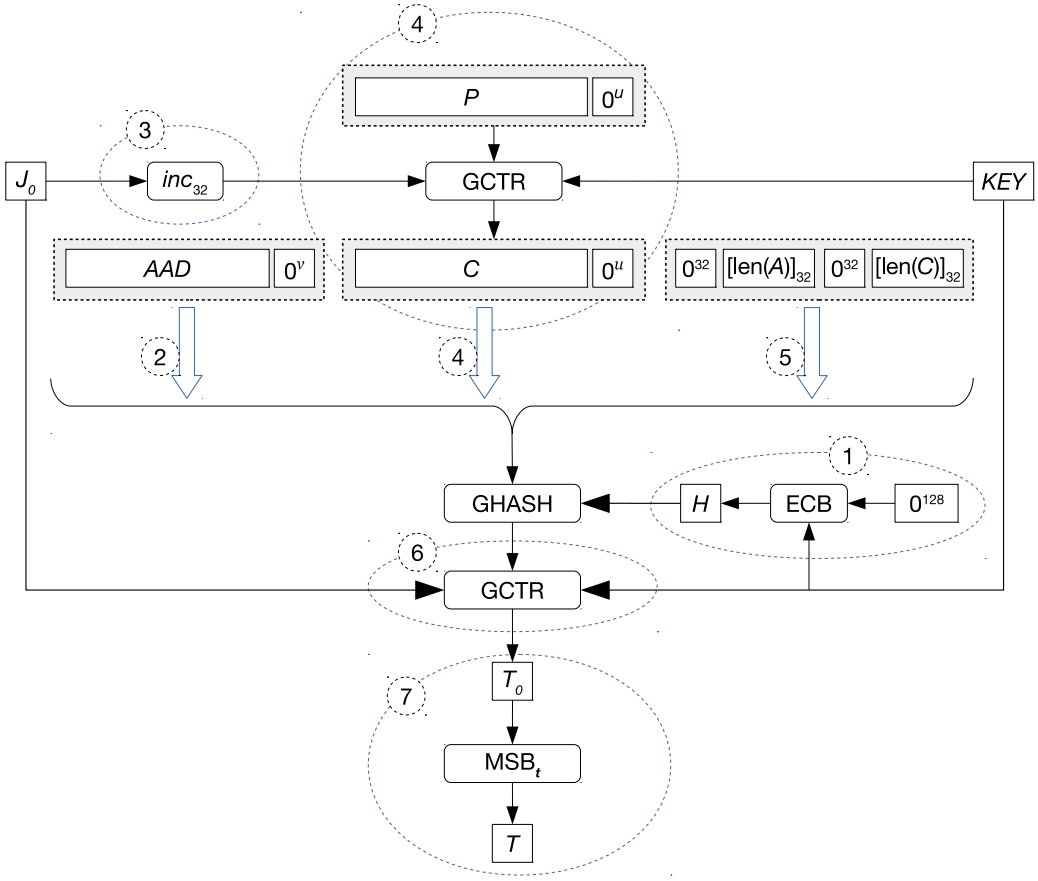


图 13-1. GCM 加密操作流程图

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 13-1 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。

13.6.1 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 13-1 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 13-11 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。

13.6.2 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 13-1 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 13-11 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。

13.6.3 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 13-1 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 13-11 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。

13.6.4 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。

13.6.5 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。

13.7 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 13-12 所示。更多信息，请前往章节 1 系统和存储器。

表 13-12. AES 基地址

访问总线	基地址
PeriBUS1	0x3F43A000
PeriBUS2	0x6003A000

13.8 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 13.7 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

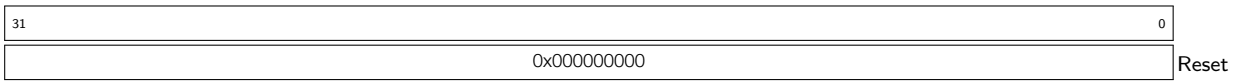
13.9 寄存器列表

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 13.7 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

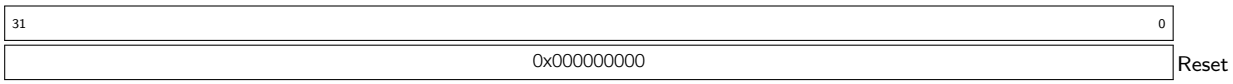
13.10 寄存器

Register 13.1: AES_KEY_*n*_REG (*n*: 0-7) (0x0000+4**n*)



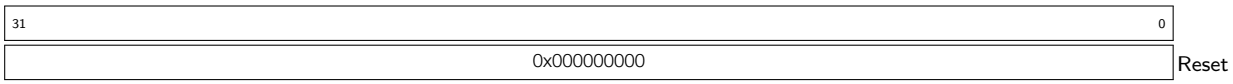
AES_KEY_*n*_REG (*n*: 0-7) AES 密钥寄存器。（读 / 写）

Register 13.2: AES_TEXT_IN_*m*_REG (*m*: 0-3) (0x0020+4**m*)



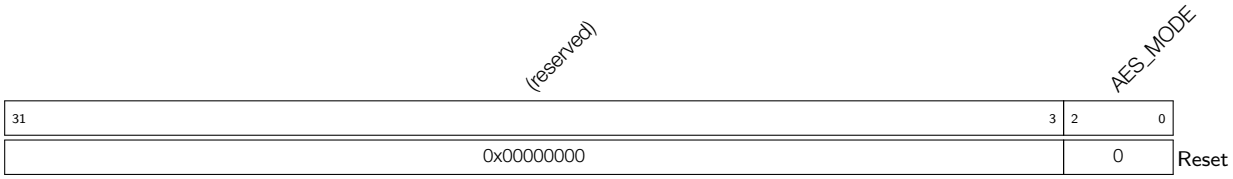
AES_TEXT_IN_*m*_REG (*m*: 0-3) Typical AES 文本输入寄存器。（读 / 写）

Register 13.3: AES_TEXT_OUT_*m*_REG (*m*: 0-3) (0x0030+4**m*)



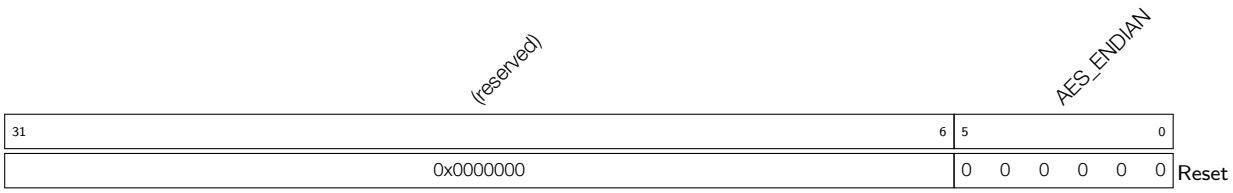
AES_TEXT_OUT_*m*_REG (*m*: 0-3) Typical AES 文本输出寄存器。（只读）

Register 13.4: AES_MODE_REG (0x0040)



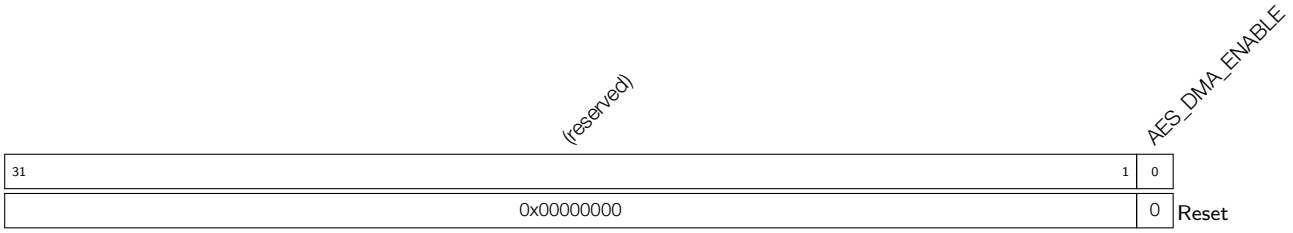
AES_MODE 选择 AES 加速器的工作模式。详情请见表 13-2。（读 / 写）

Register 13.5: AES_ENDIAN_REG (0x0044)



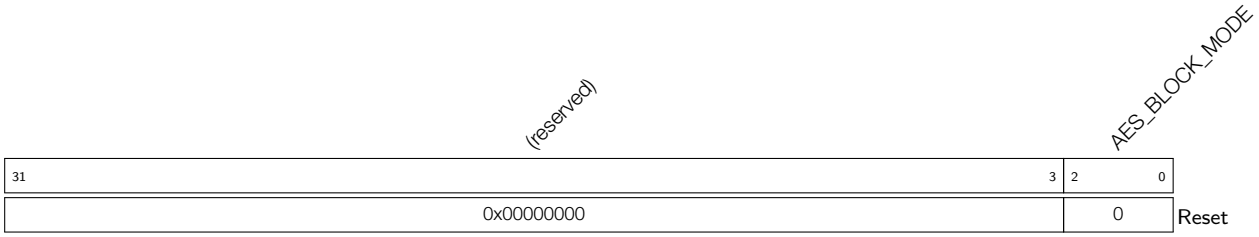
AES_ENDIAN 字节序选择寄存器。详情请见表 13-4。（读 / 写）

Register 13.6: AES_DMA_ENABLE_REG (0x0090)



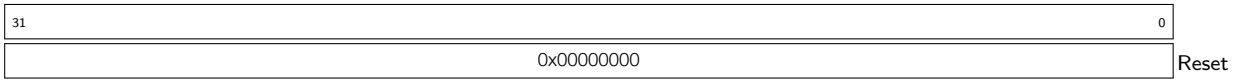
AES_DMA_ENABLE 选择工作模式。详情请见表 13-1。（读 / 写）

Register 13.7: AES_BLOCK_MODE_REG (0x0094)



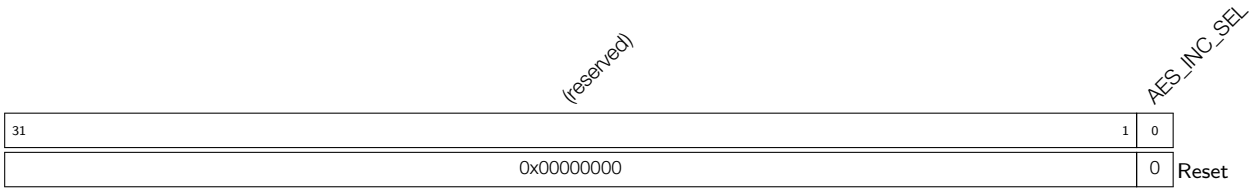
AES_BLOCK_MODE 选择 DMA-AES 使用的块模式。详情请见表 13-8。（读 / 写）

Register 13.8: AES_BLOCK_NUM_REG (0x0098)



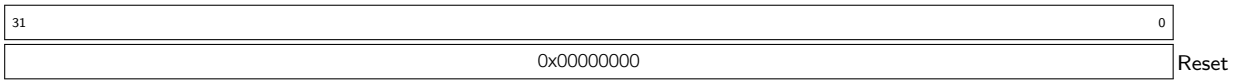
AES_BLOCK_NUM 在 DMA-AES 运算中待加解密的文本块数。详情请见章节 13.5.4。（读 / 写）

Register 13.9: AES_INC_SEL_REG (0x009C)



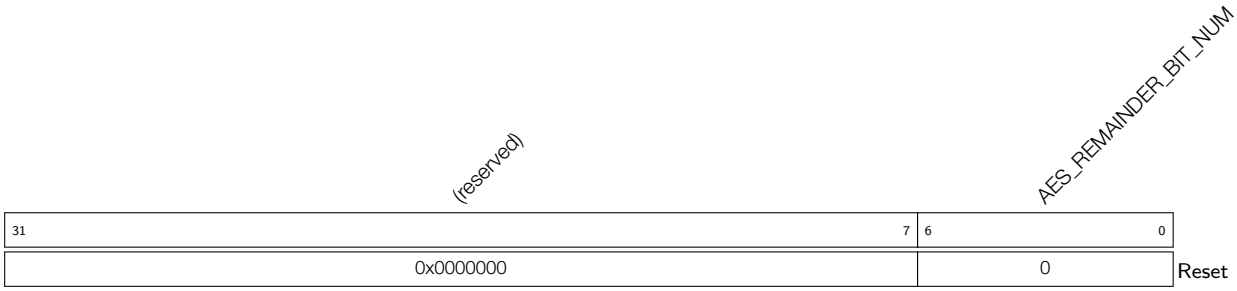
AES_INC_SEL 选择 CTR 块模式使用的标准增量函数。置 0 选择 INC_{32} 标准增量函数，置 1 选择 INC_{128} 标准增量函数。（读 / 写）

Register 13.10: AES_AAD_BLOCK_NUM_REG (0x00A0)



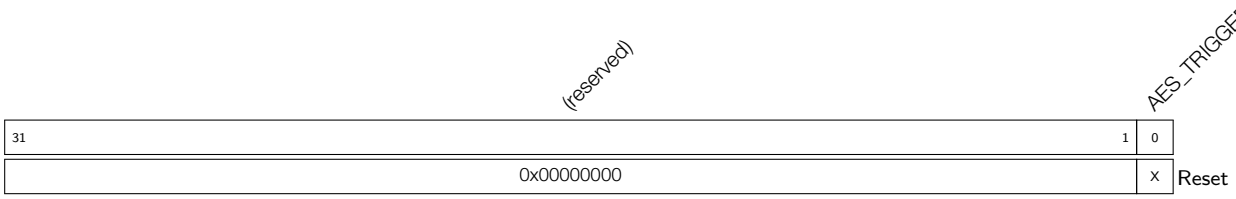
AES_AAD_BLOCK_NUM 在 GCM 运算中 AAD 的块数。详情请见章节 13.6.4。(读 / 写)

Register 13.11: AES_REMAINDER_BIT_NUM_REG (0x00A4)



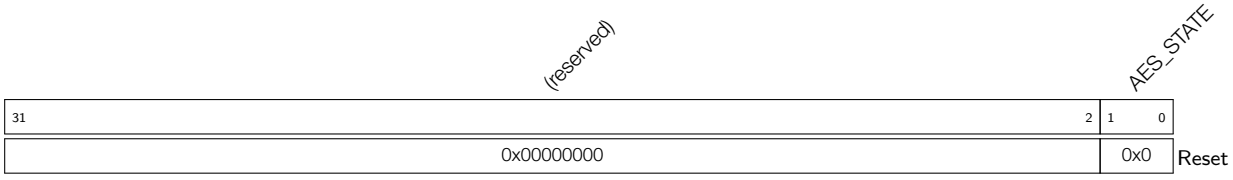
AES_REMAINDER_BIT_NUM 在 GCM 运算中输入文本的不完整块的有效比特数。详情请见章节 13.6.5。(读 / 写)

Register 13.12: AES_TRIGGER_REG (0x0048)



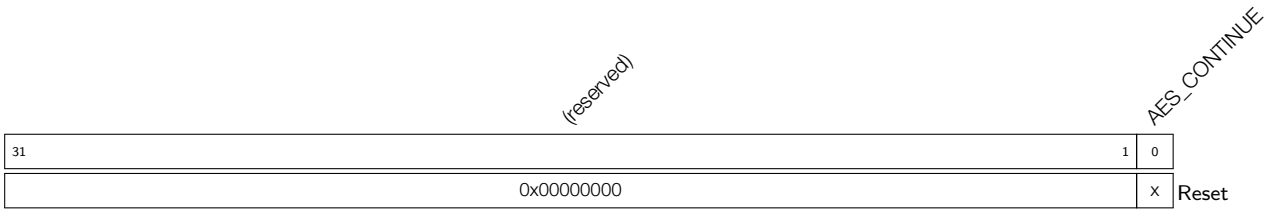
AES_TRIGGER 写入 1 使能 AES 运算。(只写)

Register 13.13: AES_STATE_REG (0x004C)



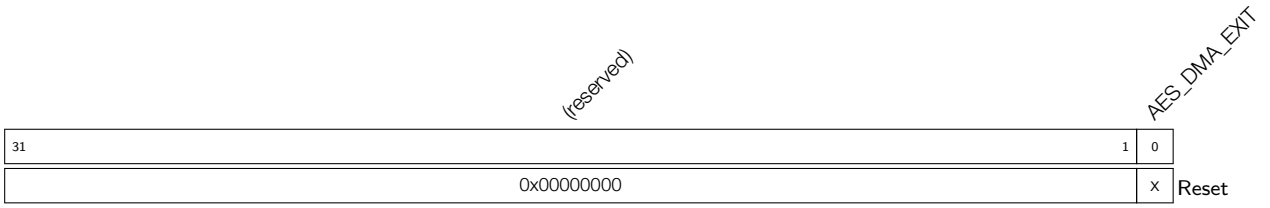
AES_STATE AES 状态寄存器。详见表 13-3 (Typical AES 工作模式) 和表 13-9 (DMA-AES 工作模式)。(只读)

Register 13.14: AES_CONTINUE_REG (0x00A8)



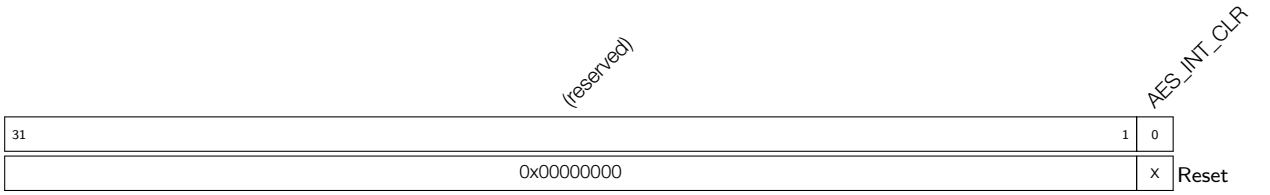
AES_CONTINUE 在 GCM 运算中，写入 1 继续运算。（只写）

Register 13.15: AES_DMA_EXIT_REG (0x00B8)



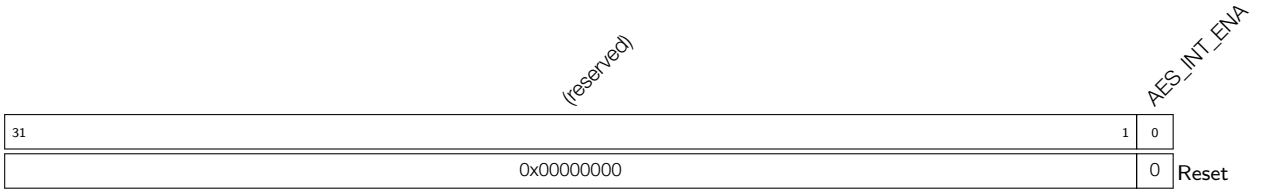
AES_DMA_EXIT 在 DMA-AES 运算完成后，在下一次配置 AES 任何寄存器之前，写入 1 使 AES 回到空闲状态。（只写）

Register 13.16: AES_INT_CLR_REG (0x00AC)



AES_INT_CLR 写入 1 清除 AES 中断。（只写）

Register 13.17: 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）

14. SHA 加速器

14.1 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。

14.2 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）

14.3 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 14-1。

表 14-1. 工作模式选择

工作模式	选择方式
Typical SHA	SHA_START_REG 置 1
DMA-SHA	SHA_DMA_START_REG 置 1

用户可通过配置 SHA_MODE_REG 寄存器选择 SHA 加速器的运算标准，具体请见表 14-2。

表 14-2. 运算标准选择

哈希运算标准	SHA_MODE_REG 的配置
SHA-1	0
SHA-224	1
SHA-256	2
SHA-384	3
SHA-512	4
SHA-512/224	5
SHA-512/256	6
SHA-512/*t*	7

注意：

ESP32-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。

14.4 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。

14.4.1 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。

14.4.1.1 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。

14.4.1.2 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。

14.4.1.3 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”。

14.4.2 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。

14.4.2.1 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 14-2。
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 14.4.1.3 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。

14.4.2.2 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 14.4.1.2 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 14-2。
2. 选择是否启用中断。请将 SHA_INT_ENA_REG 寄存器配置为 1 以启动中断。
3. 配置块个数。
   - 将待加密数据的总块数 *M* 写入 SHA_DMA_BLOCK_NUM_REG 寄存器。
4. 开始 DMA-SHA 运算。
   - 如果当前 DMA-SHA 运算为接着另一次 DMA-SHA 的运算，需要提前将另一次计算得到的信息摘要写入寄存器堆 SHA_H_*n*_REG 中，随后将 1 写入寄存器 SHA_DMA_CONTINUE_REG;
   - 否则，只需要将 1 写入寄存器 SHA_DMA_START_REG。

5. 等待 DMA-SHA 运算结束。判断 DMA-SHA 运算结束有以下两种方法：
   - 轮询寄存器 SHA_BUSY_REG 结果为 0。
   - 等待中断信号产生。此时，应及时通过软件将 SHA_INT_CLEAR_REG 寄存器置为 1 以清除中断。
6. 获取信息摘要
   - 从寄存器堆 SHA_H_*n*_REG 取出信息摘要。

DMA-SHA 的具体工作流程（SHA-512/*t*）

1. 选择运算标准。
   - 配置 SHA_MODE_REG 寄存器为 7 选择 SHA-512/*t* 运算标准。
2. 选择是否启用中断。请将 SHA_INT_ENA_REG 寄存器配置为 1 以启动中断。
3. 计算哈希初始值。
   (a) 配置 t_string 和 t_length，并将其写入 SHA_T_STRING_REG 和 SHA_T_LENGTH_REG 寄存器。具体请见 14.4.1.3 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。

14.4.3 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 14-6：

表 14-6. 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。

14.4.4 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。

14.5 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 14-7 所示。更多信息，请前往章节 1 *系统和存储器*。

表 14-7. SHA 基地址

基地址	地址值
PeriBUS1	0x3F43B000
PeriBUS2	0x6003B000

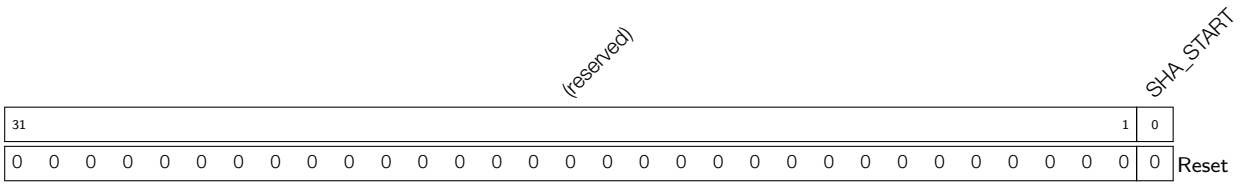
14.6 寄存器列表

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

名称	描述	地址	访问权限
SHA_M_5_REG	输入信息	0x0094	R/W
SHA_M_6_REG	输入信息	0x0098	R/W
SHA_M_7_REG	输入信息	0x009C	R/W
SHA_M_8_REG	输入信息	0x00A0	R/W
SHA_M_9_REG	输入信息	0x00A4	R/W
SHA_M_10_REG	输入信息	0x00A8	R/W
SHA_M_11_REG	输入信息	0x00AC	R/W
SHA_M_12_REG	输入信息	0x00B0	R/W
SHA_M_13_REG	输入信息	0x00B4	R/W
SHA_M_14_REG	输入信息	0x00B8	R/W
SHA_M_15_REG	输入信息	0x00BC	R/W
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

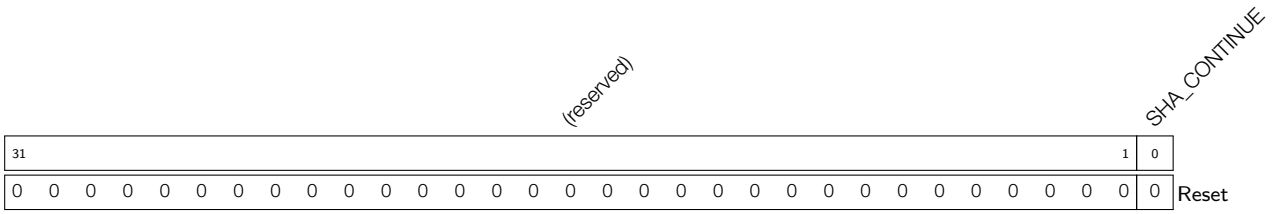
14.7 寄存器

Register 14.1: SHA_START_REG (0x0010)



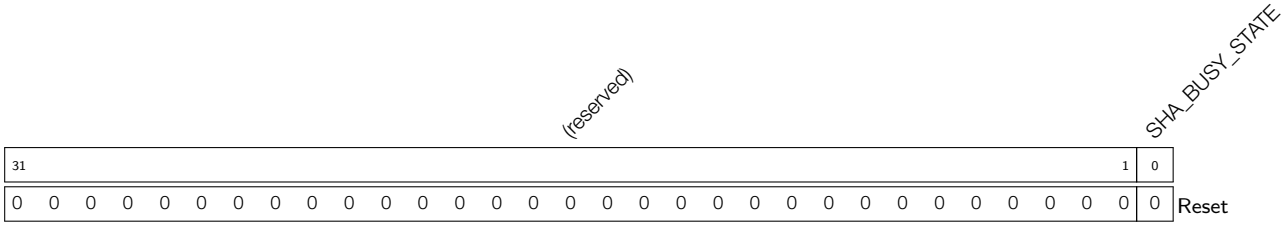
SHA_START 置 1 启动 SHA 加速器的 Typical SHA 模式。（只写）

Register 14.2: SHA_CONTINUE_REG (0x0014)



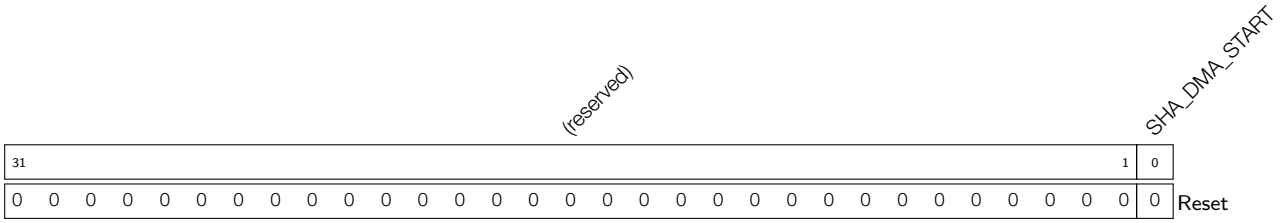
SHA_CONTINUE 置 1 继续 SHA 加速器的 Typical SHA 运算。（只写）

Register 14.3: SHA_BUSY_REG (0x0018)



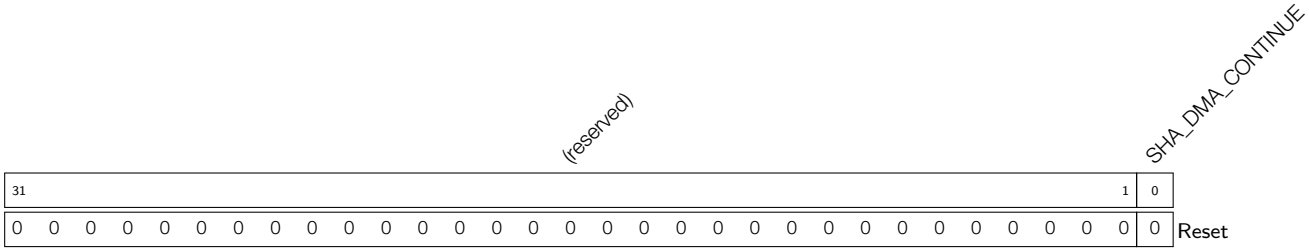
SHA_BUSY_STATE 指示 SHA 是否处于“忙碌”状态。（只读）1'h0: 空闲 1'h1: 忙碌

Register 14.4: SHA_DMA_START_REG (0x001C)



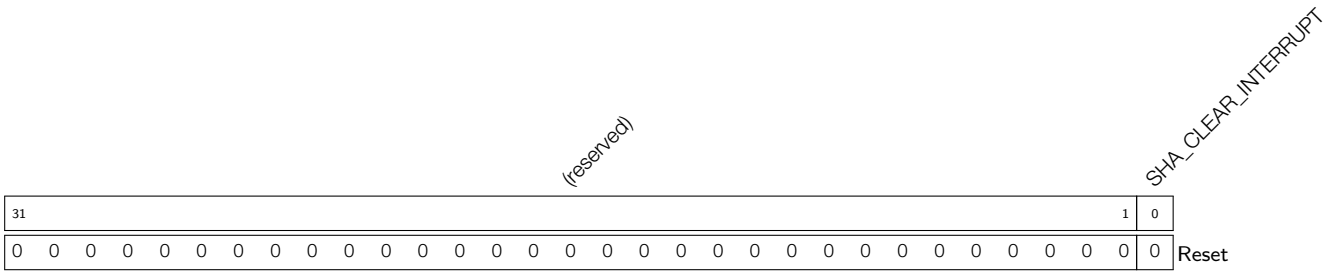
SHA_DMA_START 置 1 启动 SHA 加速器的 DMA-SHA 模式。（只写）

Register 14.5: SHA_DMA_CONTINUE_REG (0x0020)



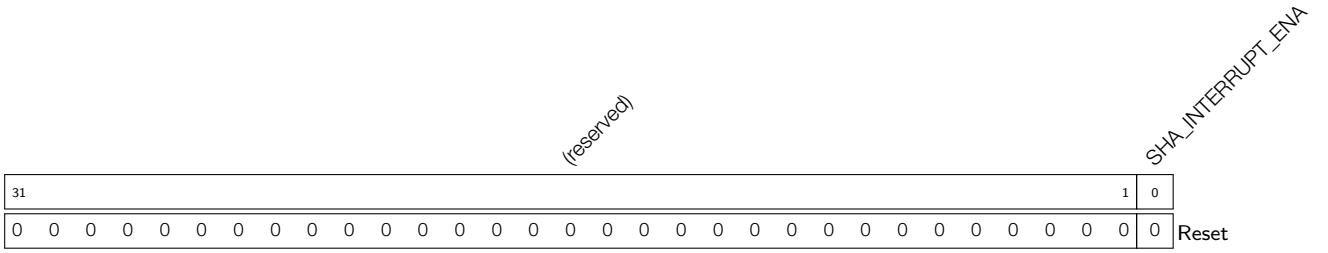
SHA_DMA_CONTINUE 置 1 继续 SHA 加速器的 DMA-SHA 运算。（只写）

Register 14.6: SHA_INT_CLEAR_REG (0x0024)



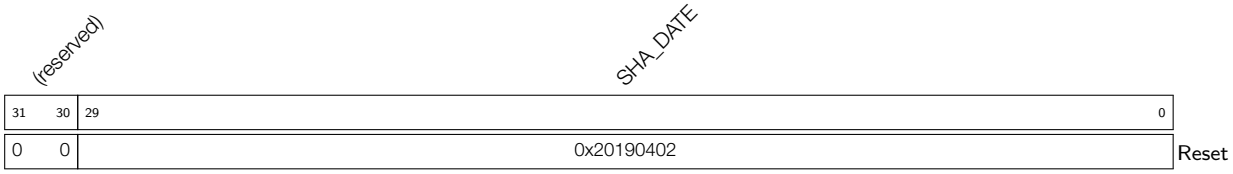
SHA_CLEAR_INTERRUPT 清除 DMA-SHA 中断。(只写)

Register 14.7: SHA_INT_ENA_REG (0x0028)



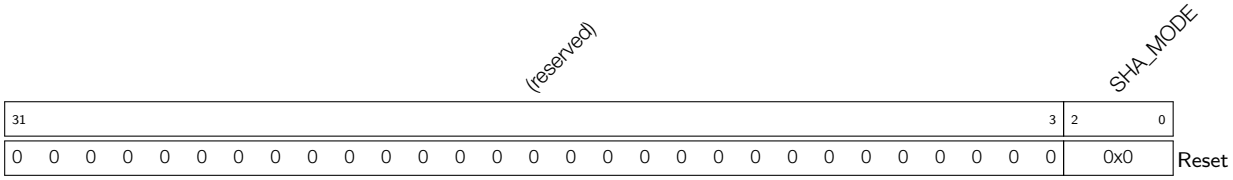
SHA_INTERRUPT_ENA 使能 DMA-SHA 中断。(读写)

Register 14.8: SHA_DATE_REG (0x002C)



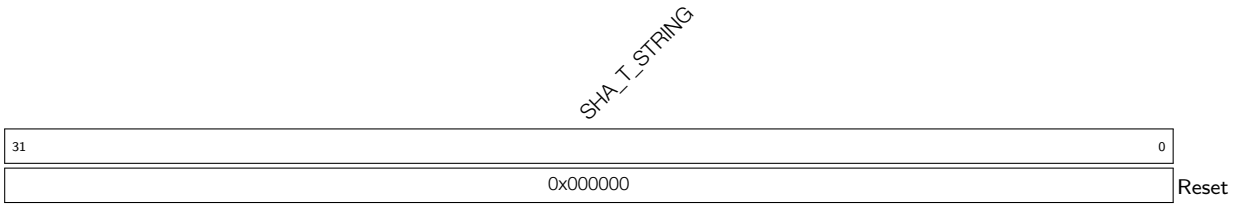
SHA_DATE 版本控制寄存器。(读写)

Register 14.9: SHA_MODE_REG (0x0000)



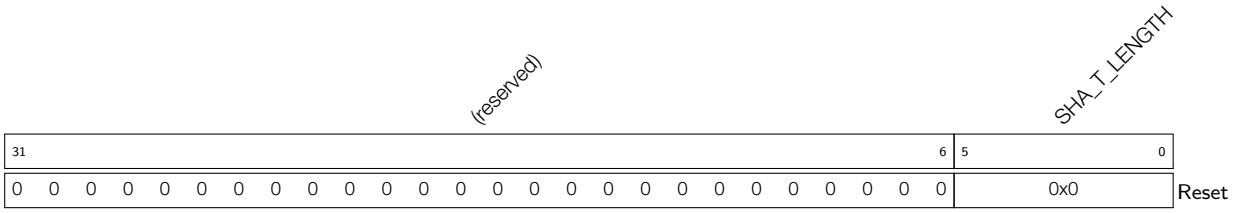
SHA_MODE 选择 SHA 加速器的运算标准，详见表 14-2。(R/W)

Register 14.10: SHA_T_STRING_REG (0x0004)



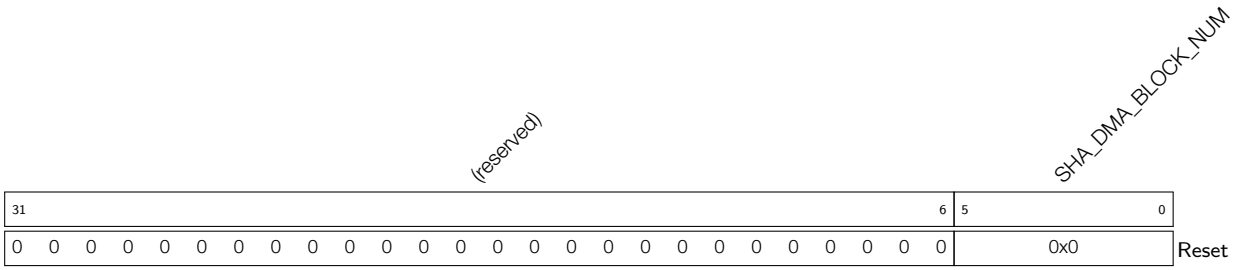
SHA_T_STRING 存储哈希字符串内容（仅用于计算 SHA-512/t 的哈希初始值）。（读写）

Register 14.11: SHA_T_LENGTH_REG (0x0008)



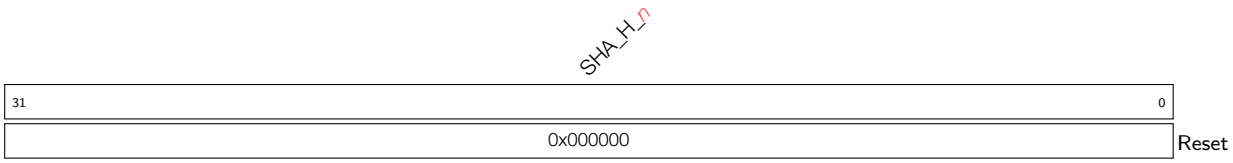
SHA_T_LENGTH 存储哈希字符串长度（仅用于计算 SHA-512/t 的哈希初始值）。（读写）

Register 14.12: SHA_DMA_BLOCK_NUM_REG (0x000C)



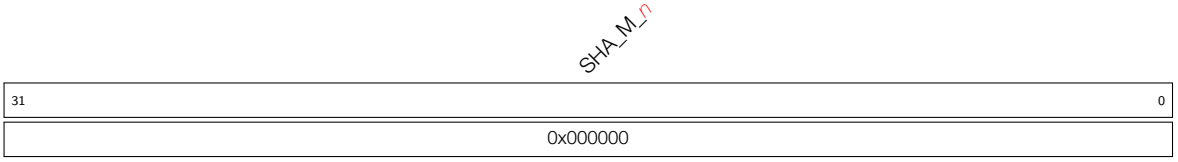
SHA_DMA_BLOCK_NUM 定义 DMA-SHA 工作模式下的信息块个数。（读写）

Register 14.13: 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 14.14: 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)

15. RSA 加速器

15.1 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。

15.2 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

15.3 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。

15.3.1 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 15.3.4 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 $\overline{r}_i$ 都已经被覆盖。所以当需要连续运算时，只需要更新必需的寄存器与存储器即可。

15.3.2 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 $\overline{r}$ 和 M'。

RSA 加速器也支持 128 种运算子长度的大数模乘运算。

大数模乘运算的软件流程为：

1. 对寄存器 RSA_INTERRUPT_ENA_REG 写 1 / 0 以开启 / 关闭中断。

2. 配置相关寄存器。

   (a) 对寄存器 RSA_MODE_REG 写入 $(\frac{N}{32} - 1)$。

   (b) 对寄存器 RSA_M_PRIME_REG 写入 M'。

3. 将 X_i、Y_i、M_i、$\overline{r}_i$ $(i \in \{0, 1, \ldots, n\})$ 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 $\overline{r}_i$ 已经被覆盖。所以当需要连续运算时，只需要更新必需的寄存器与存储器即可。

15.3.3 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。

15.3.4 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，
- $\widetilde{Y}_{N-1}$, $\widetilde{Y}_{N-2}$, ..., $\widetilde{Y}_{t+1}$ 的值均为 0，

- 且 $\widetilde{Y}_{t-1}, \widetilde{Y}_{t-2}, ..., \widetilde{Y}_0$ 中包括 m 个 0，其余 t-m 全部为 1，即 $\widetilde{Y}_{t-1}\widetilde{Y}_{t-2}, \cdots, \widetilde{Y}_0$ 的汉明重量 (Hamming weight) 为 $t-m$。

此时，当启动加速选项时：

- 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，$\widetilde{Y}_{N-1}, \widetilde{Y}_{N-2}, ..., \widetilde{Y}_{t+1}$ 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 15-1 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 15-1. 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%

15.4 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 15-2 所示。更多有关总线的信息，请见章节 *1 系统和存储器*。

表 15-2. RSA 基地址

外设访问方式	地址值
PeriBUS1	0x3F43C000
PeriBUS2	0x6003C000

15.5 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 15.4 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

15.6 寄存器列表

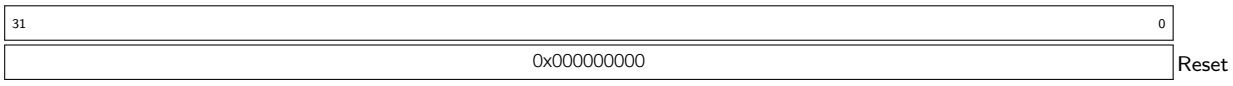
请注意，这里的地址是相对于 RSA 基地址的地址偏移量（相对地址）。请参阅章节 15.4 获取有关 RSA 基地址的信息。

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

15.7 寄存器

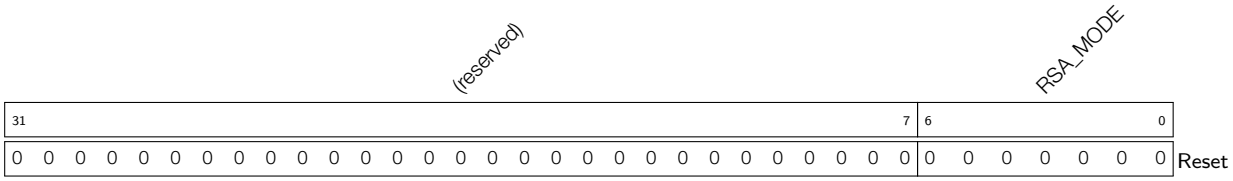
请注意，本章节的地址是相对于 RSA 基地址的地址偏移量（相对地址）。请参阅章节 15.4 获取有关 RSA 基地址的信息。

Register 15.1: RSA_M_PRIME_REG (0x0800)



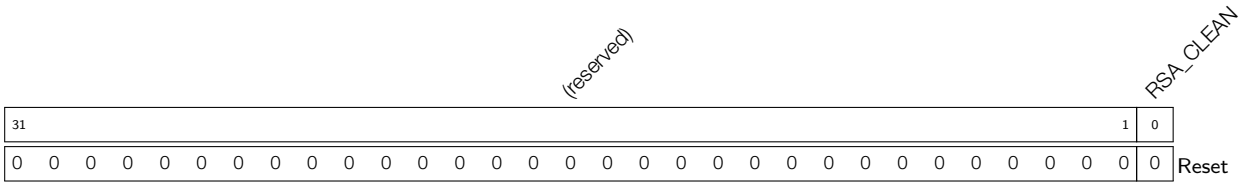
RSA_M_PRIME_REG 此寄存器存储 M'。(读 / 写)

Register 15.2: RSA_MODE_REG (0x0804)



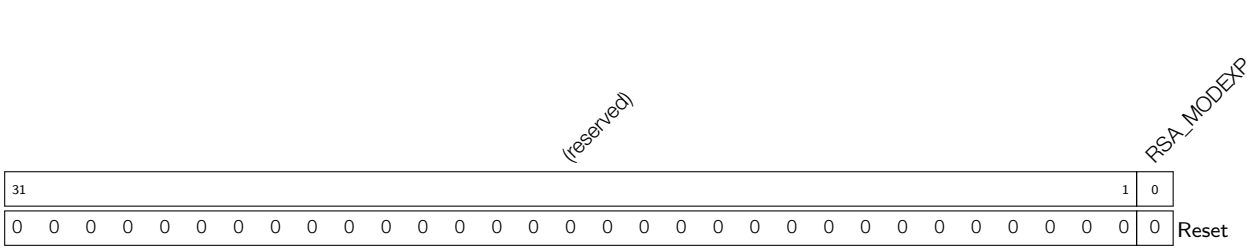
RSA_MODE 此寄存器存储模幂运算的模式。(读 / 写)

Register 15.3: RSA_CLEAN_REG (0x0808)



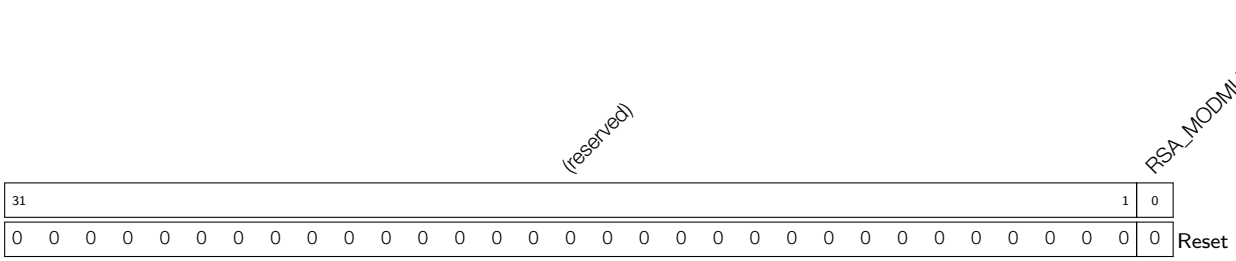
RSA_CLEAN 一旦存储器初始化结束，此位为 1。(只读)

Register 15.4: RSA_MODEXP_START_REG (0x080C)



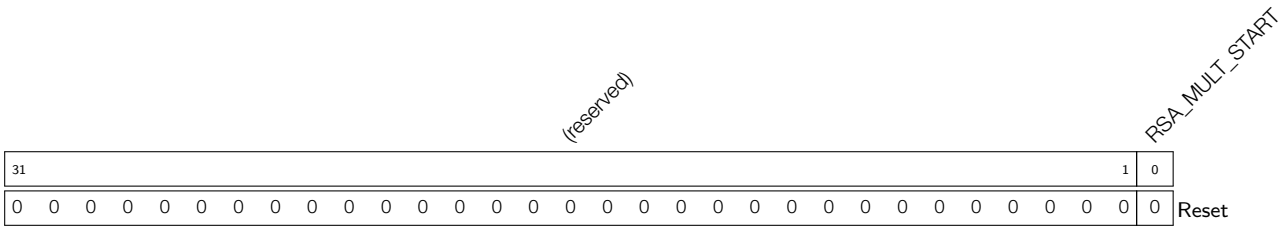
RSA_MODEXP_START 写入 1 以开始模幂运算。(只写)

Register 15.5: RSA_MODMULT_START_REG (0x0810)



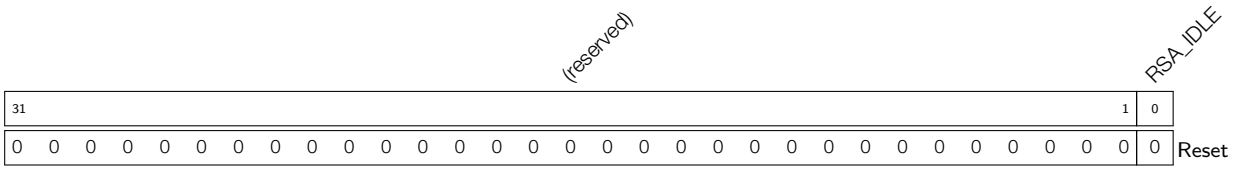
RSA_MODMULT_START 写入 1 以开始模乘运算。(只写)

Register 15.6: RSA_MULT_START_REG (0x0814)



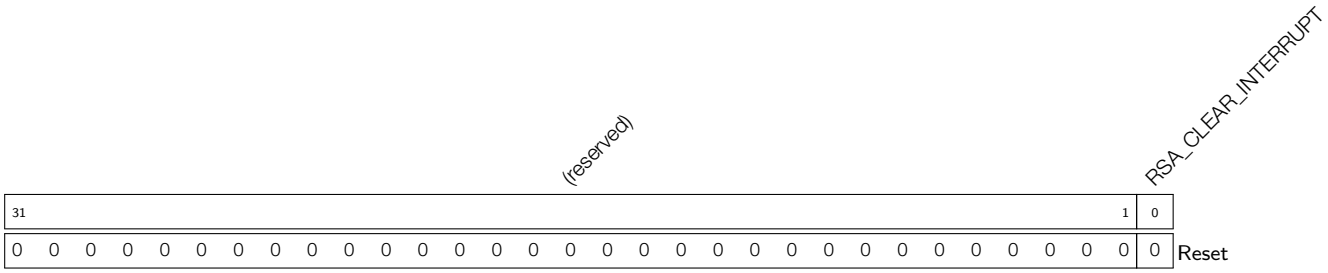
RSA_MULT_START 写入 1 以开始乘法运算。(只写)

Register 15.7: RSA_IDLE_REG (0x0818)



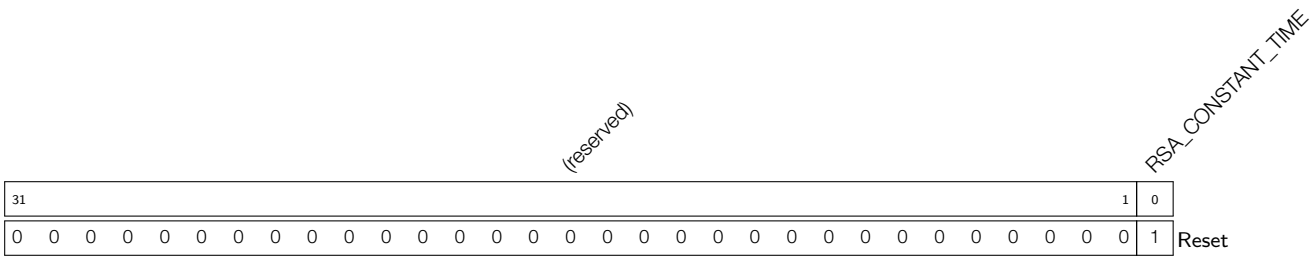
RSA_IDLE 当 RSA 空闲时，此位为 1。(只读)

Register 15.8: RSA_CLEAR_INTERRUPT_REG (0x081C)



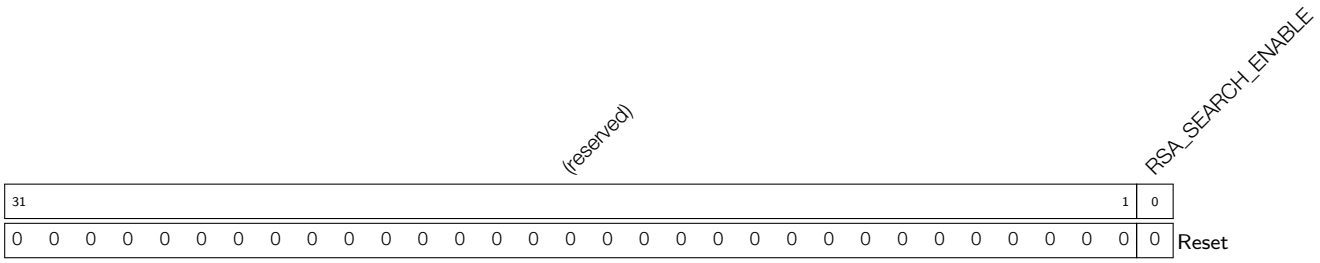
RSA_CLEAR_INTERRUPT RSA 中断清除寄存器。写入 1 清除中断。(只写)

Register 15.9: RSA_CONSTANT_TIME_REG (0x0820)



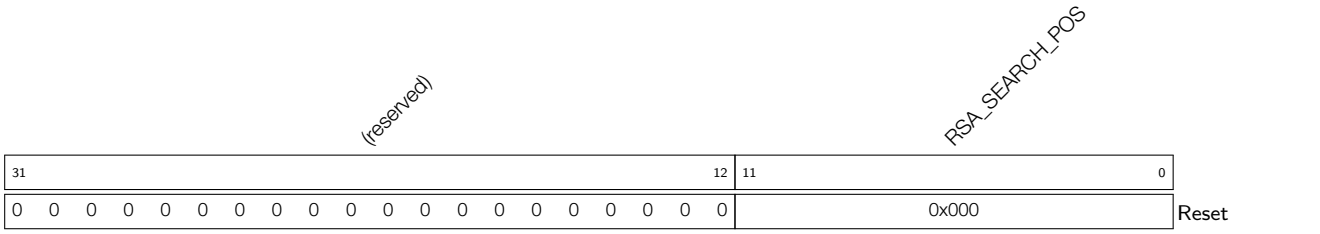
RSA_CONSTANT_TIME_REG 模幂运算中的 constant_time 加速选项。0: 开启; 1: 关闭 (默认)。(读 / 写)

Register 15.10: RSA_SEARCH_ENABLE_REG (0x0824)



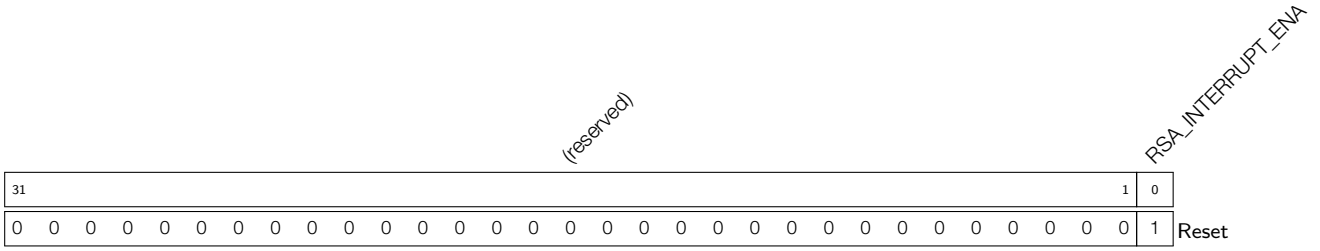
RSA_SEARCH_ENABLE 模幂运算中的 search 加速选项。1：开启；0：关闭（默认）。（读 / 写）

Register 15.11: RSA_SEARCH_POS_REG (0x0828)



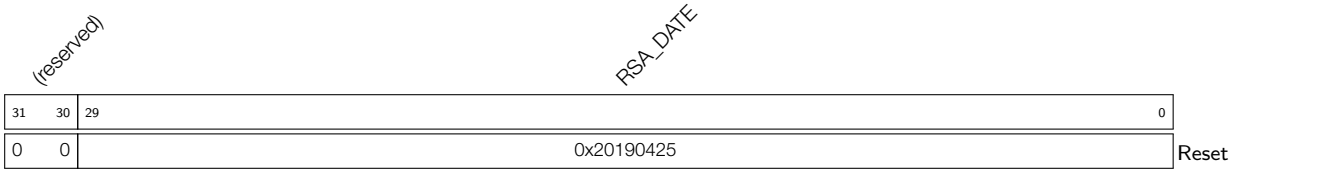
RSA_SEARCH_POS 模幂运算中的 search 加速选项。用于配置 search 的起始位置（读 / 写）。

Register 15.12: 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 15.13: RSA_DATE_REG (0x0830)



RSA_DATE 版本控制寄存器（读 / 写）。

16. 随机数发生器

16.1 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。

16.2 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。

16.3 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。

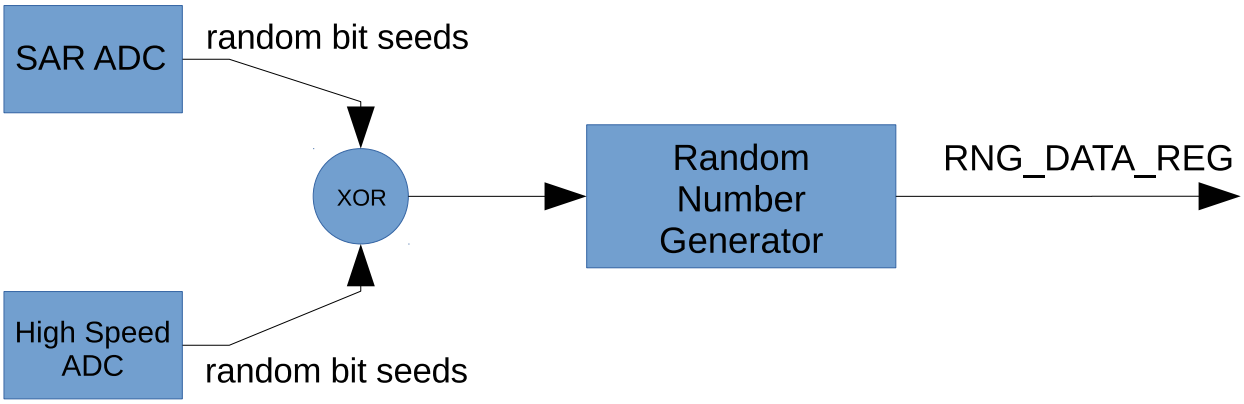


图 16-1. 噪声源

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。

16.4 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 16-1 所示。更多信息，请访问章节 1 系统和存储器。

表 16-1. 随机数发生器基地址

访问总线	基地址
PeriBUS1	0x3FF75000
PeriBUS2	0x60035000

16.5 寄存器列表

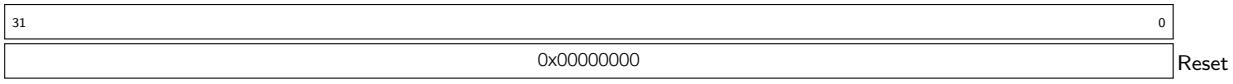
请注意，下表中的地址都是相对于随机数发生器基地址的地址偏移量（相对地址）。更多有关随机数发生器基地址的信息，请前往 16.4 章节。

名称	描述	地址	访问
RNG_DATA_REG	随机数数据	0x0110	只读

16.6 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 16.4 章节。

Register 16.1: RNG_DATA_REG (0x0110)



RNG_DATA 随机数来源。(只读)

17. 片外存储器加密与解密

17.1 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。

17.2 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

17.3 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 17-1 所示。

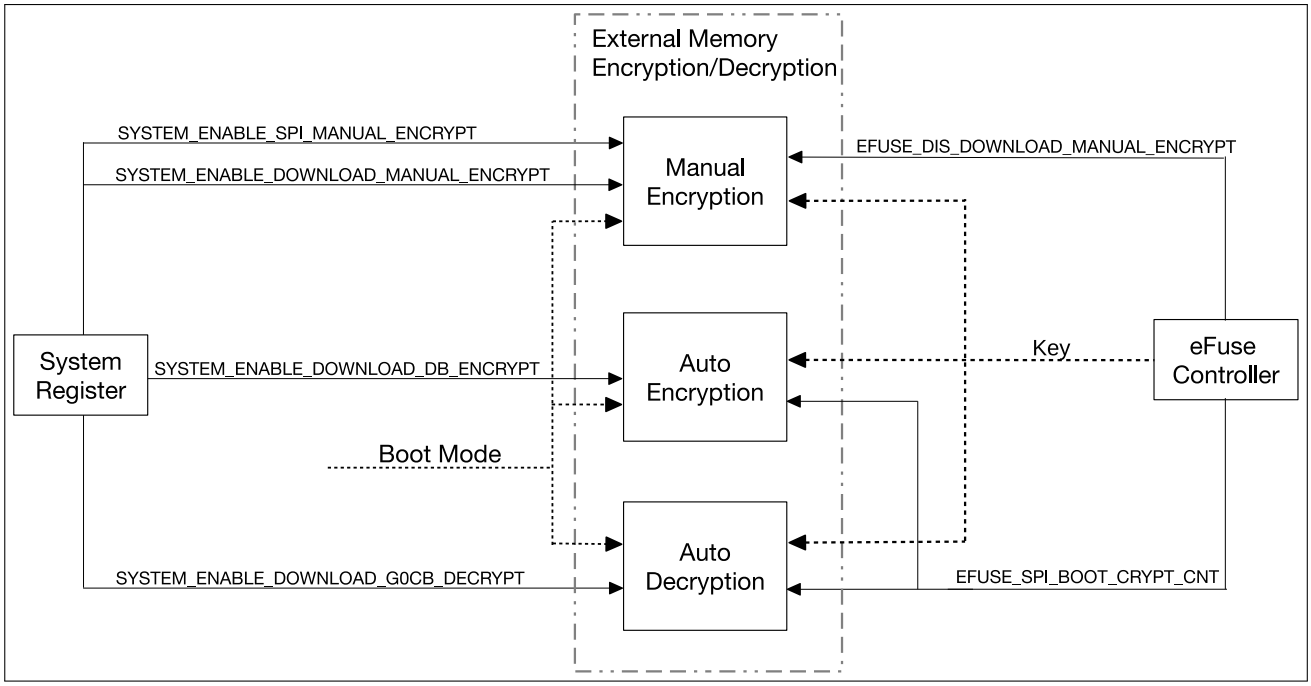


图 17-1. 片外存储器加解密模块架构

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。

17.3.1 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。

17.3.2 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 17-1 所示。

表 17-1. 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

$Block_A$	$Block_B$	$Block_C$	Key 值	Key 长度（位）
No	No	No	0^{256}	256

注：表 17-1 中的 “Yes” 指存在，“No” 指不存在，“0^{256}” 表示 256 个位”0” 组成的位串，“||” 表示将两个比特串按照前后顺序合成一个更长的新位串。

更多有关密钥用途的信息，请参考章节 11 *eFuse 控制器*。

17.3.3 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。

17.3.4 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 17-2 所示。

表 17-2. 目标空间与寄存器堆的映射关系

$offset$	寄存器	$offset$	寄存器
0x00	XTS_AES_PLAIN_0_REG	0x20	XTS_AES_PLAIN_8_REG
0x04	XTS_AES_PLAIN_1_REG	0x24	XTS_AES_PLAIN_9_REG
0x08	XTS_AES_PLAIN_2_REG	0x28	XTS_AES_PLAIN_10_REG
0x0C	XTS_AES_PLAIN_3_REG	0x2C	XTS_AES_PLAIN_11_REG
0x10	XTS_AES_PLAIN_4_REG	0x30	XTS_AES_PLAIN_12_REG
0x14	XTS_AES_PLAIN_5_REG	0x34	XTS_AES_PLAIN_13_REG
0x18	XTS_AES_PLAIN_6_REG	0x38	XTS_AES_PLAIN_14_REG
0x1C	XTS_AES_PLAIN_7_REG	0x3C	XTS_AES_PLAIN_15_REG

17.3.5 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 17.3.3。
2. 用明文填充寄存器堆 XTS_AES_PLAIN_n_REG (n: 0-15)。请参考章节 17.3.4 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。

17.3.6 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。

17.3.7 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。

17.4 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 17-3 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 17-3. 手动加密模块基地址

访问外设总线	地址值
PeriBUS1	0x3F43A000
PeriBUS2	0x6003A000

17.5 寄存器列表

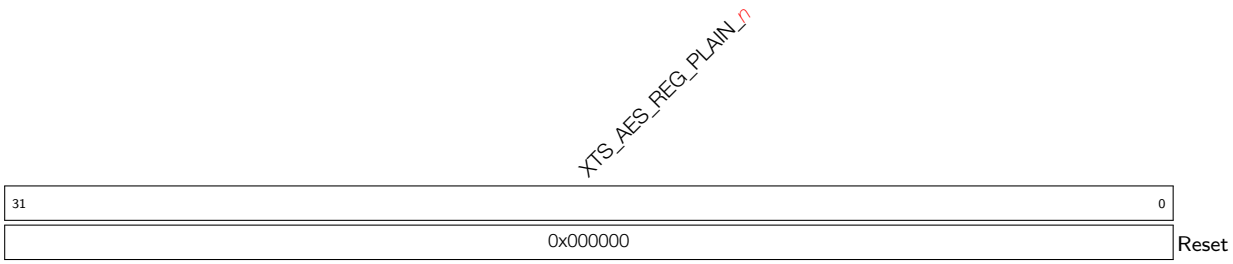
请注意，这里的地址是相对于手动加密模块基地址的地址偏移量（相对地址）。请参阅章节 17.4 获取有关手动加密模块的基地址的信息。

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

名称	描述	地址	访问
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
版本寄存器			
XTS_AES_DATE_REG	版本控制寄存器	0x015C	只读

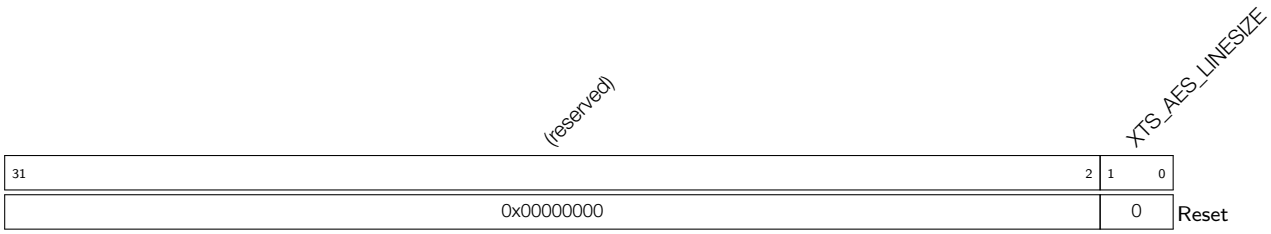
17.6 寄存器

Register 17.1: 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)

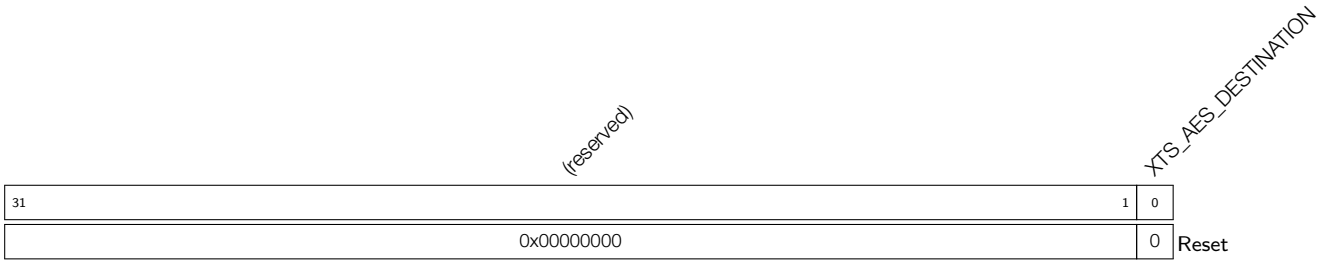
Register 17.2: XTS_AES_LINESIZE_REG (0x0140)



XTS_AES_LINESIZE 块大小寄存器，决定单次加密的数据量。(读/写)

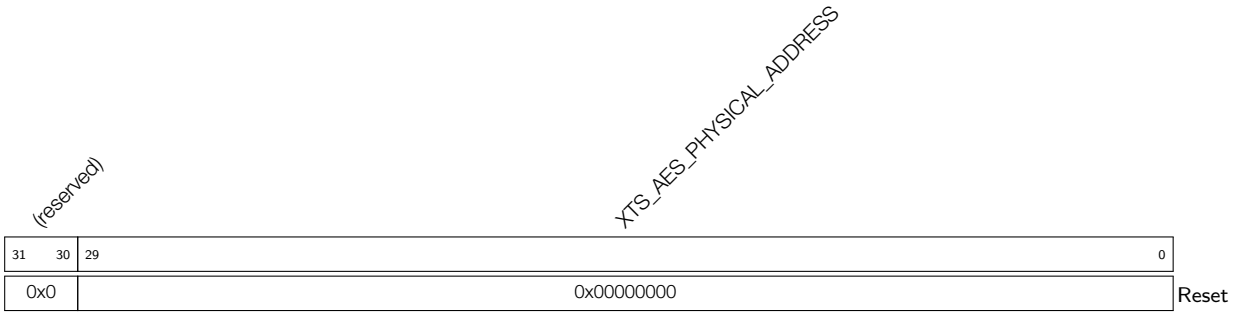
- 0：加密 128 位；
- 1：加密 256 位；
- 2：加密 512 位。

Register 17.3: XTS_AES_DESTINATION_REG (0x0144)



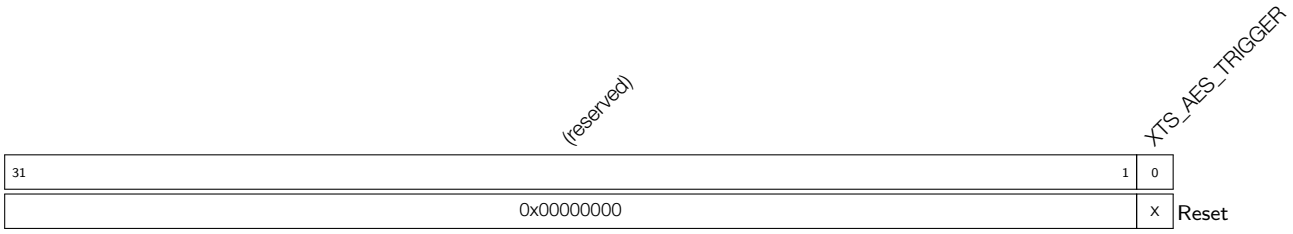
XTS_AES_DESTINATION 决定手动加密类型，目前只能手动加密 flash，所以只能为 0。用户不能写入 1，否则将发生错误。0：加密 flash；1：加密片外 RAM。（读/写）

Register 17.4: XTS_AES_PHYSICAL_ADDRESS_REG (0x0148)



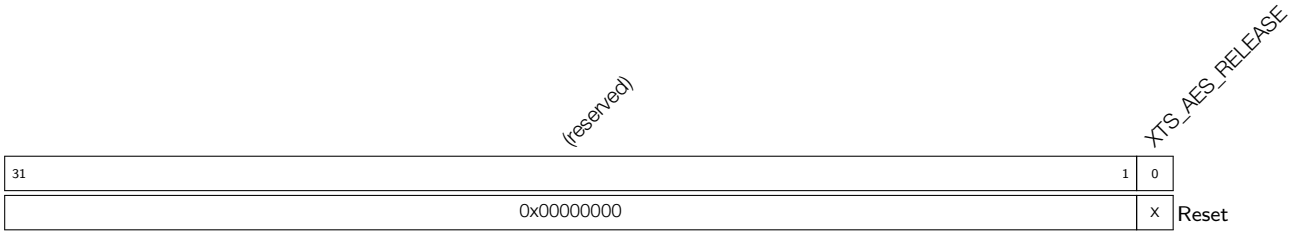
XTS_AES_PHYSICAL_ADDRESS 物理地址。（读/写）

Register 17.5: XTS_AES_TRIGGER_REG (0x014C)



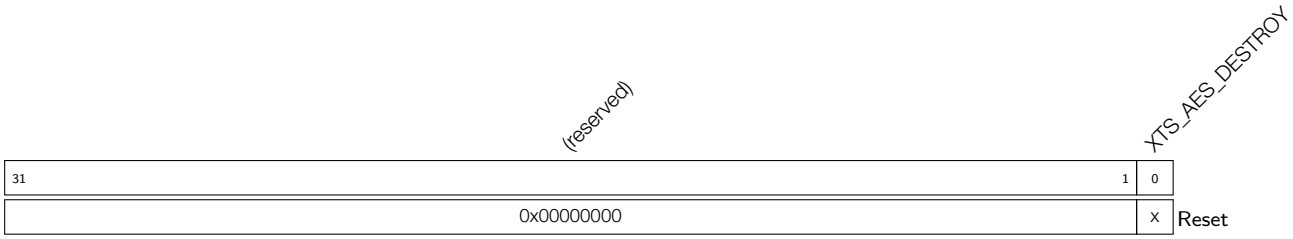
XTS_AES_TRIGGER 写入 1 使能手动加密运算。（只写）

Register 17.6: XTS_AES_RELEASE_REG (0x0150)



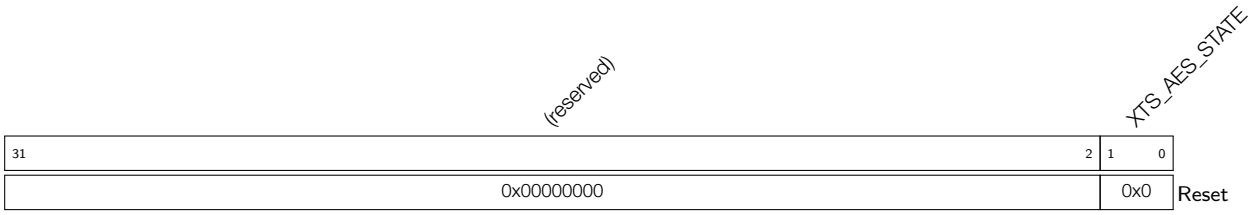
XTS_AES_RELEASE 写入 1 使加密结果对 SPI1 可见，因而 SPI1 可以获取到加密结果。（只写）

Register 17.7: XTS_AES_DESTROY_REG (0x0154)



XTS_AES_DESTROY 写入 1 销毁加密结果。（只写）

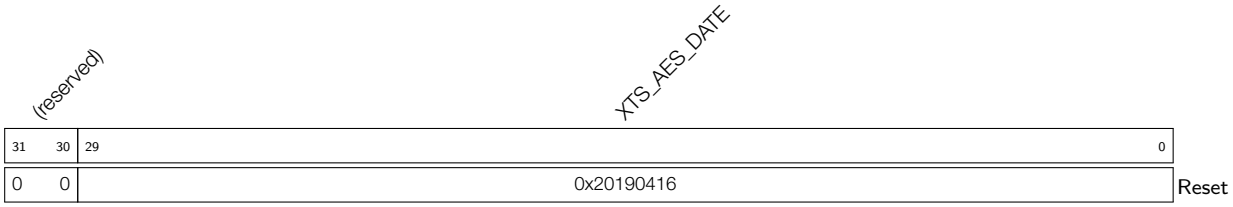
Register 17.8: XTS_AES_STATE_REG (0x0158)



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 17.9: XTS_AES_DATE_REG (0x015C)



XTS_AES_DATE 版本控制寄存器。(只读)

18. 数字签名

18.1 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。

18.2 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

18.3 功能描述

18.3.1 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。

18.3.2 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 18.3.4。

DS 模块计算 RSA 加密操作 $Z = X^Y \bmod M$，更多信息请参考章节 15 *RSA 加速器*中第 15.3.1 节*大数模幂运算*。

18.3.3 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。

18.3.4 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 18.3.2 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。

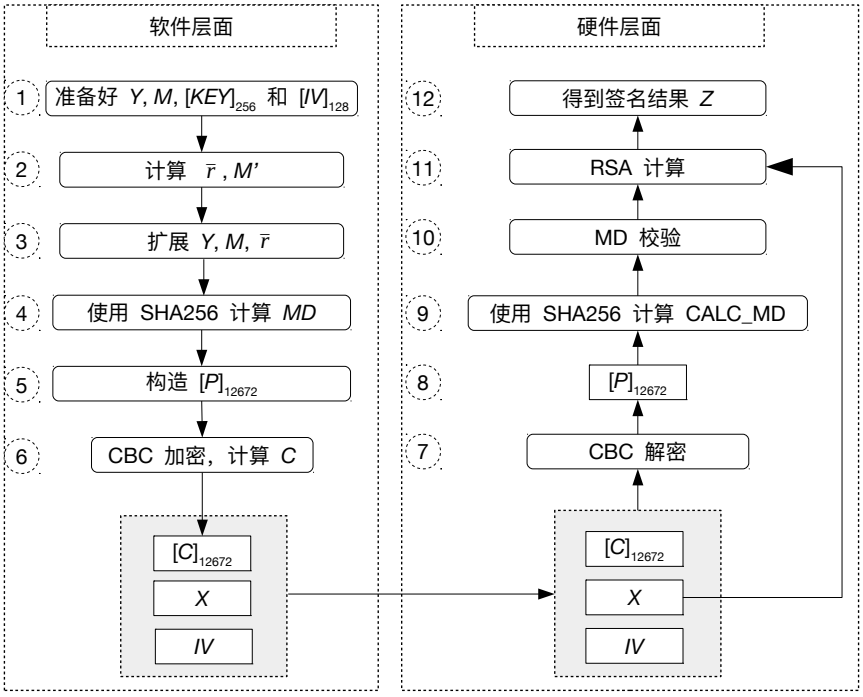


图 18-1. 软件准备工作与硬件工作流程

图 18-1 中左半边给出了计算 C 的过程。用户需要依照图 18-1 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 13 *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 18.3.4 所述，DS_KEY 由 eFuse 中的 $HMAC\_KEY$ 得出。

18.3.5 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 18.3.4 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 18-1 中的步骤 7 和步骤 8**

   解析过程是图 18-1 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 18-1 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 18-1 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 18-1 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 $X^Y \bmod M$ 的所有必要输入参数。大数模幂运算的运算长度由 L 的值唯一指定。DS 模块调用 RSA 加速器完成大数模幂运算 $Z = X^Y \bmod M$，Z 为签名结果。

18.3.6 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 18.3.5 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 13 *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）。

18.4 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 18-1 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 18-1. 基地址

访问外设总线	地址值
PeriBUS1	0x3F43D000
PeriBUS2	0x6003D000

18.5 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 18-1 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

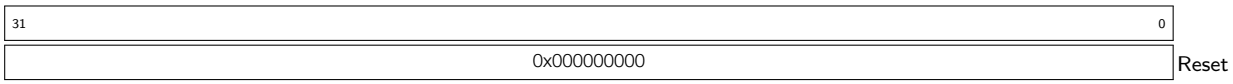
18.6 寄存器列表

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 18-1 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

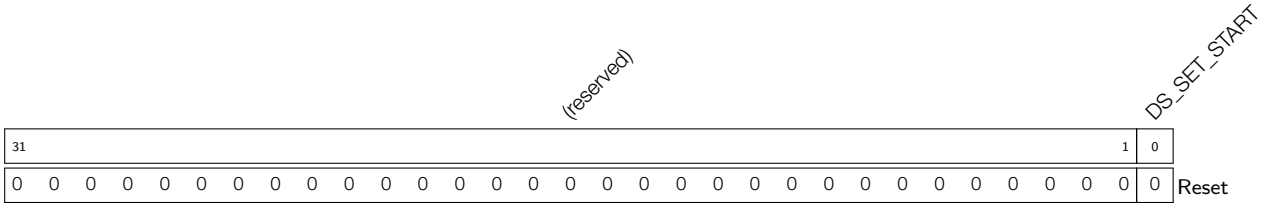
18.7 寄存器

Register 18.1: DS_IV_*m*_REG (*m*: 0-3) (0x0630+4**m*)



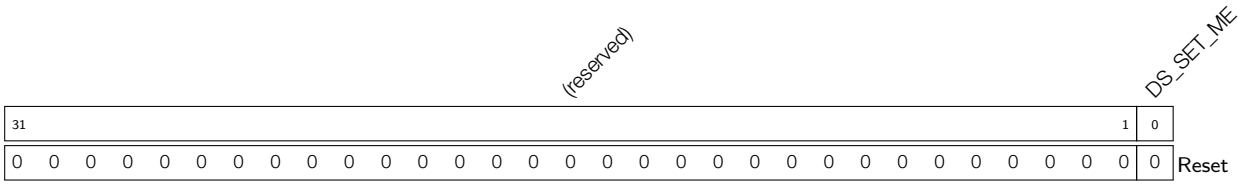
DS_IV_*m*_REG (*m*: 0-3) *IV* block 数据。(只写)

Register 18.2: DS_SET_START_REG (0x0E00)



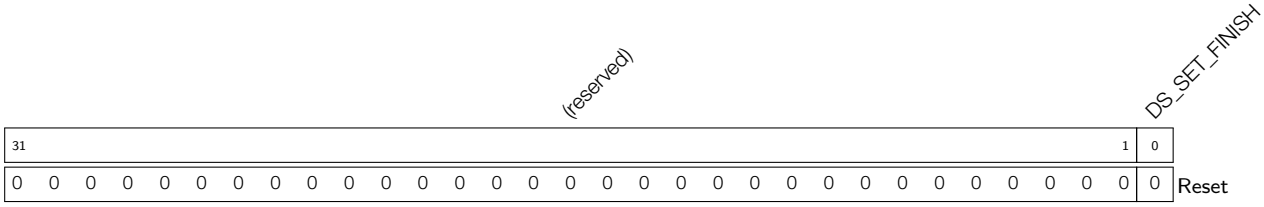
DS_SET_START 写入 1 启动 DS 模块。(只写)

Register 18.3: DS_SET_ME_REG (0x0E04)



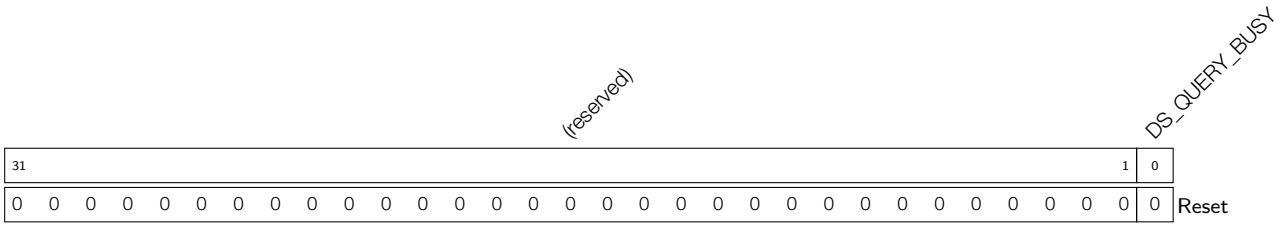
DS_SET_ME 写入 1 以开始计算。(只写)

Register 18.4: DS_SET_FINISH_REG (0x0E08)



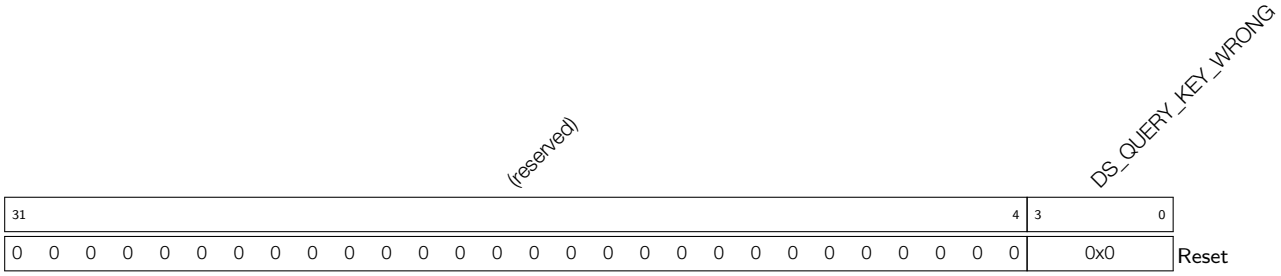
DS_SET_FINISH 写入 1 以结束运算。(只写)

Register 18.5: DS_QUERY_BUSY_REG (0x0E0C)



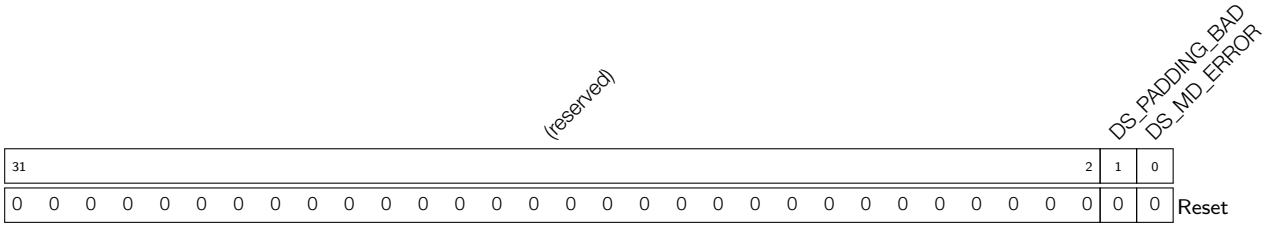
DS_QUERY_BUSY 1：DS 模块正在忙；0：DS 模块空闲。（只读）

Register 18.6: 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）

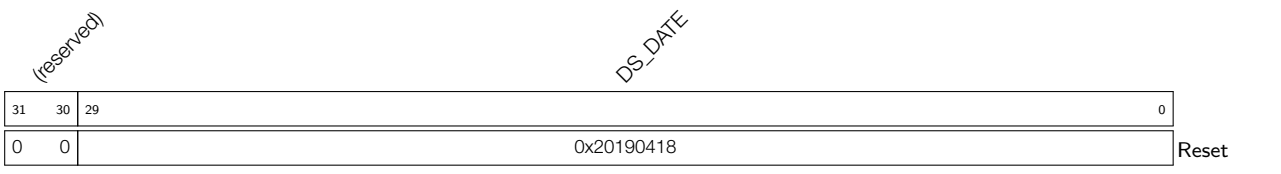
Register 18.7: DS_QUERY_CHECK_REG (0x0E14)



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 18.8: DS_DATE_REG (0x0E20)



DS_DATE 版本控制寄存器（读/写）。

修订历史

日期	版本	发布说明
2019.11	V0.1	预发布。