ESP32-S3

技术参考手册

预发布 v0.1
乐鑫信息科技

关于本手册

《ESP32-S3 技术参考手册》的目标读者群体是使用 ESP32-S3 芯片的应用开发工程师。本手册提供了关于 ESP32-S3 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-S3 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。

目录

表格

插图

1 系统和存储器

1.1 概述

ESP32-S3 采用哈佛结构 Xtensa® LX7 CPU 构成双核系统。所有的内部存储器、外部存储器以及外设都分布在 CPU 的总线上。

1.2 主要特性

- **地址空间**
  - 848 KB 内部存储器指令地址空间
  - 560 KB 内部存储器数据地址空间
  - 836 KB 外设地址空间
  - 32 MB 外部存储器指令虚地址空间
  - 32 MB 外部存储器数据虚地址空间
  - 480 KB 内部 DMA 地址空间
  - 32 MB 外部 DMA 地址空间
- **内部存储器**
  - 384 KB 内部 ROM
  - 512 KB 内部 SRAM
  - 8 KB RTC 快速存储器
  - 8 KB RTC 慢速存储器
- **外部存储器**
  - 最大支持 1 GB 片外 flash
  - 最大支持 1 GB 片外 RAM
- **外设空间**
  - 总计 45 个模块/外设
- **GDMA**
  - 11 个具有 GDMA 功能的模块/外设

图 1-1 描述了系统结构与地址映射结构。

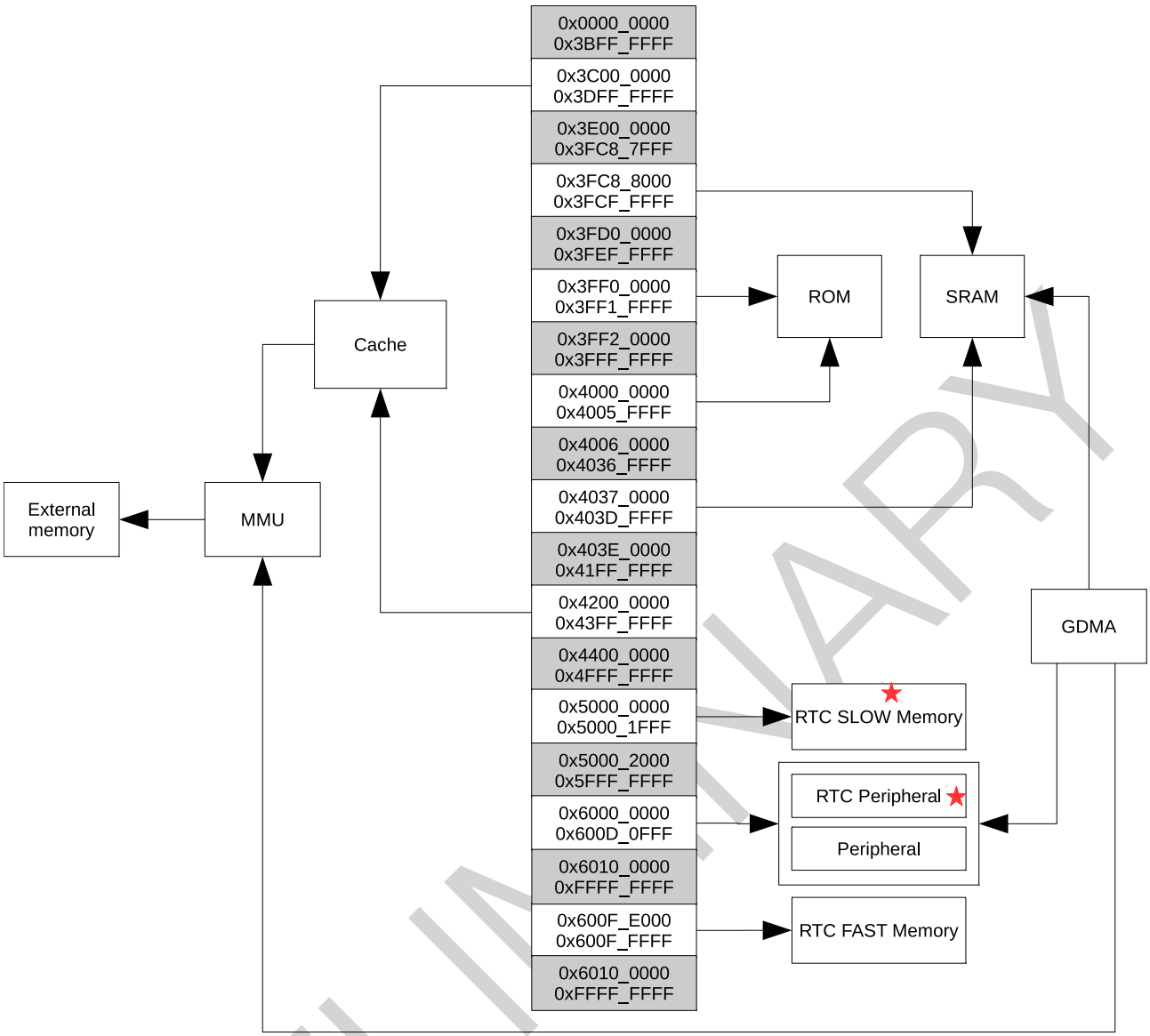


图 1-1. 系统结构与地址映射结构

说明：

- 图中灰色背景标注的地址空间不可用。
- 图中红色五角星表示对应存储器或外设可以被协处理器访问。
- 地址空间中可用的地址范围可能大于实际可用的内存。

1.3 功能描述

1.3.1 地址映射

系统由两个哈佛结构 Xtensa® LX7 CPU 构成，这两个 CPU 能够访问的地址空间范围完全一致。

地址 0x4000_0000 以下的部分属于数据总线的地址范围，地址 0x4000_0000 ~ 0x4FFF_FFFF 部分为指令总线的地址范围，地址 0x5000_0000 及以上的部分是数据总线与指令总线共用的地址范围。

CPU 的数据总线与指令总线都为小端序。CPU 可以通过数据总线进行单字节、双字节、4 字节、16 字节的数据访问。CPU 也可以通过指令总线进行数据访问，但必须是 4 字节对齐方式；非对齐数据访问会导致 CPU 工作异常。

CPU 能够：

- 通过数据总线与指令总线直接访问内部存储器；
- 通过 Cache 直接访问映射到地址空间的外部存储器；
- 通过数据总线直接访问模块/外设。

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	0x3BFF_FFFF		保留
数据	0x3C00_0000	0x3DFF_FFFF	32 MB	外部存储器
	0x3E00_0000	0x3FC8_7FFF		保留
数据	0x3FC8_8000	0x3FCF_FFFF	480 KB	内部存储器
	0x3FD0_0000	0x3FEF_FFFF		保留
数据	0x3FF0_0000	0x3FF1_FFFF	128 KB	内部存储器
	0x3FF2_0000	0x3FFF_FFFF		保留
指令	0x4000_0000	0x4005_FFFF	384 KB	内部存储器
	0x4006_0000	0x4036_FFFF		保留
指令	0x4037_0000	0x403D_FFFF	448 KB	内部存储器
	0x403E_0000	0x41FF_FFFF		保留
指令	0x4200_0000	0x43FF_FFFF	32 MB	外部存储器
	0x4400_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	0x600D_0FFF	836 KB	外设
	0x600D_1000	0x600F_DFFF		保留
	0x600F_E000	0x600F_FFFF	8 KB	内部存储器
	0x6010_0000	0xFFFF_FFFF		保留

1.3.2 内部存储器

ESP32-S3 的内部存储器包含如下三种类型：

- Internal ROM (384 KB)：Internal ROM 是只读存储器，不可编程。Internal ROM 中存放有一些系统底层软件的 ROM 代码（程序指令和一些只读数据）。
- Internal SRAM (512 KB)：内部静态存储器（SRAM）是易失性（volatile）存储器，可以快速响应 CPU 的访问请求（通常一个 CPU 时钟周期）。
  - SRAM 中的一部分可以被配置成外部存储器访问的缓存（Cache），在这种情况下无法被 CPU 访问。

  - SRAM 中的某些部分只可以被 CPU 的指令总线访问。
  - SRAM 中的某些部分只可以被 CPU 的数据总线访问。
  - SRAM 中的某些部分既可以被 CPU 的指令总线访问，又可以被 CPU 的数据总线访问。
- RTC Memory (16 KB)：RTC 存储器以静态 RAM (SRAM) 方式实现，因此也是易失性存储器。但是，在 deep sleep 模式下，存放在 RTC 存储器中的数据不会丢失。
  - RTC FAST Memory (8 KB)：RTC FAST memory 只可以被 CPU 访问，不可以被协处理器访问，通常用来存放一些在 Deep Sleep 模式下仍需保持的程序指令和数据。
  - RTC SLOW Memory (8 KB)：RTC SLOW memory 既可以被 CPU 访问，又可以被协处理器访问，因此通常用来存放一些 CPU 和协处理器需要共享的程序指令和数据。

基于上述对几种类型的内部存储器的描述，ESP32-S3 的内部存储器可以被分为四个部分：Internal ROM (384 KB)、Internal SRAM (512 KB)、RTC FAST Memory (8 KB)、RTC SLOW Memory (8 KB)。CPU 通过不同的总线访问这几部分内部存储器时会有些许限制（如某些部分只允许 CPU 通过指令总线访问），据此内部存储器可以被区分的更加细致。表 1-2 列出了所有内部存储器以及可以访问内部存储器的数据总线与指令总线地址段。

表 1-2. 内部存储器地址映射

总线类型	边界地址		容量	目标
	低位地址	高位地址		
数据	0x3FF0_0000	0x3FF1_FFFF	128 KB	Internal ROM 1
	0x3FC8_0000	0x3FCE_FFFF	416 KB	Internal SRAM 1
	0x3FCF_0000	0x3FCF_FFFF	64 KB	Internal SRAM 2
指令	0x4000_0000	0x4003_FFFF	256 KB	Internal ROM 0
	0x4004_0000	0x4005_FFFF	128 KB	Internal ROM 1
	0x4037_0000	0x4037_7FFF	32 KB	Internal SRAM 0
	0x4037_8000	0x403D_FFFF	416 KB	Internal SRAM 1
数据/指令	0x5000_0000	0x5000_1FFF	8 KB	RTC SLOW Memory
	0x600F_E000	0x600F_FFFF	8 KB	RTC FAST Memory

说明：

所有的内部存储器都接受权限管理。只有获取到访问内部存储器的访问权限，才可以执行正常的访问操作，CPU 访问内部存储器时才可以被响应。关于权限管理的更多信息，请参考章节 7 权限控制 *(PMS) [to be added later]*。

1. Internal ROM 0

Internal ROM 0 的容量为 256 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。

2. Internal ROM 1

Internal ROM 1 的容量为 128 KB，只读。如表 1-2 所示，CPU 可以通过指令总线地址段 0x4004_0000 ~ 0x4005_FFFF 或数据总线地址段 0x3FF0_0000 ~ 0x3FF1_FFFF 同序访问这部分存储器。

这两段地址同序访问 Internal ROM 1 是指：地址 0x4005_0000 与 0x3FF0_0000 访问到相同的字，0x4005_0004 与 0x3FF0_0004 访问到相同的字，0x4005_0008 与 0x3FF0_0008 访问到相同的字，以此类推（下同）。

3. Internal SRAM 0

Internal SRAM 0 的容量为 32 KB，可读可写。如表 1-2 所示，CPU 只可以通过指令总线访问这部分存储器。

通过配置，这部分存储器中的 16 KB、或全部 32 KB 可以被 instruction Cache (ICache) 占用，用来缓存外部存储器的指令或只读数据。被 ICache 占用的部分不可以被 CPU 访问，未被 ICache 占用的部分仍然可以被 CPU 访问。

4. Internal SRAM 1

Internal SRAM 1 容量为 416 KB，可读可写。如表 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 416 KB 地址空间由多个容量为 8 KB 和 16 KB 的小存储器（子存储器）组成。可以从中选取至多 16 KB 的存储空间作为 Trace Memory 用于 CPU 的 Trace 功能，并且 Trace Memory 仍可被 CPU 访问。

5. Internal SRAM 2

Internal SRAM 2 的容量为 64 KB，可读可写。如表 1-2 所示，CPU 只可以通过数据总线访问这部分存储器。

通过配置，这部分存储器中的 32 KB、或全部 64 KB 可以被 data Cache (DCache) 占用，用来缓存外部存储器的数据。被 DCache 占用的部分不可以被 CPU 访问，未被 DCache 占用的部分仍然可以被 CPU 访问。

6. RTC FAST Memory

RTC FAST Memory 容量为 8 KB，可读可写。如表 1-2 所示，CPU 可以通过数据/指令总线的共用地址段 0x600F_E000 ~ 0x600F_FFFF 访问这部分存储器。

7. RTC SLOW Memory

RTC SLOW Memory 容量为 8 KB，可读可写。如表 1-2 所示，CPU 可以通过数据/指令总线的共用地址段 0x5000_E000 ~ 0x5001_FFFF 访问这部分存储器。

RTC SLOW Memory 也可以被当作一个外设来使用，此时 CPU 可以通过地址段 0x6002_1000 ~ 0x6002_2FFF 来访问它。

1.3.3 外部存储器

ESP32-S3 支持以 SPI、Dual SPI、Quad SPI、Octal SPI、QPI、OPI 等接口形式连接 flash 和片外 RAM。ESP32-S3 还支持基于 XTS-AES 算法的硬件加解密功能，从而保护开发者片外 flash 和片外 RAM 中的程序和数据。

1.3.3.1 外部存储器地址映射

CPU 借助高速缓存 (Cache) 来访问外部存储器。Cache 将根据内存管理单元 (MMU) 中的信息把 CPU 指令总线或数据总线的地址变换为访问片外 flash 与片外 RAM 的实地址。经过变换的实地址所组成的实地址空间最大支持 1 GB 的片外 flash 与 1 GB 的片外 RAM。

通过高速缓存，ESP32-S3 一次最多可以同时有：

- 32 MB 的指令总线地址空间，通过指令缓存 (ICache) 以 64 KB 为单位映射到片外 flash 或片外 RAM，支持 4 字节对齐的读访问或取指访问。
- 32 MB 的数据总线地址空间，通过数据缓存 (DCache) 以 64 KB 为单位映射到片外 RAM，支持单字节、双字节、4 字节、16 字节的读写访问。这部分地址空间也可以用作只读数据空间，映射到片外 flash 或片外 RAM。

表 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
	低位地址	高位地址		
数据	0x3C00_0000	0x3DFF_FFFF	32 MB	DCache
指令	0x4200_0000	0x43FF_FFFF	32 MB	ICache

说明：

只有获取到外部存储器的访问权限，CPU 访问外部存储器时才可以被响应。关于权限管理的更多信息，请参考章节 7 权限控制 *(PMS) [to be added later]*。

1.3.3.2 高速缓存

如图 1-2 所示，ESP32-S3 采用双核共享 ICache 和 DCache 结构，以便当 CPU 的指令总线和数据总线同时发起请求时，也可以迅速响应。Cache 的存储空间与内部存储空间可以复用（详见章节 1.3.2 中内部 SRAM 0 与内部 SRAM 2）。

当两个核的指令总线同时访问 ICache 时，由仲裁器决定谁先获得访问 ICache 的权限；当两个核的数据总线同时访问 DCache 时，由仲裁器决定谁先获得访问 DCache 的权限。当 Cache 缺失时，Cache 控制器会向外部存储器发起请求，当 ICache 和 DCache 同时发起外部存储器请求时，由仲裁器决定谁先获得外部存储器的使用权。ICache 的缓存大小可配置为 16 KB 或 32 KB，块大小可以配置为 16 B 或 32 B，当 ICache 缓存大小配置为 32 KB 时禁用 16 B 块大小模式。DCache 的缓存大小可配置为 32 KB 或 64 KB，块大小可以配置为 16 B、32 B 或 64 B，当 DCache 缓存大小配置为 64 KB 时禁用 16 B 块大小模式。

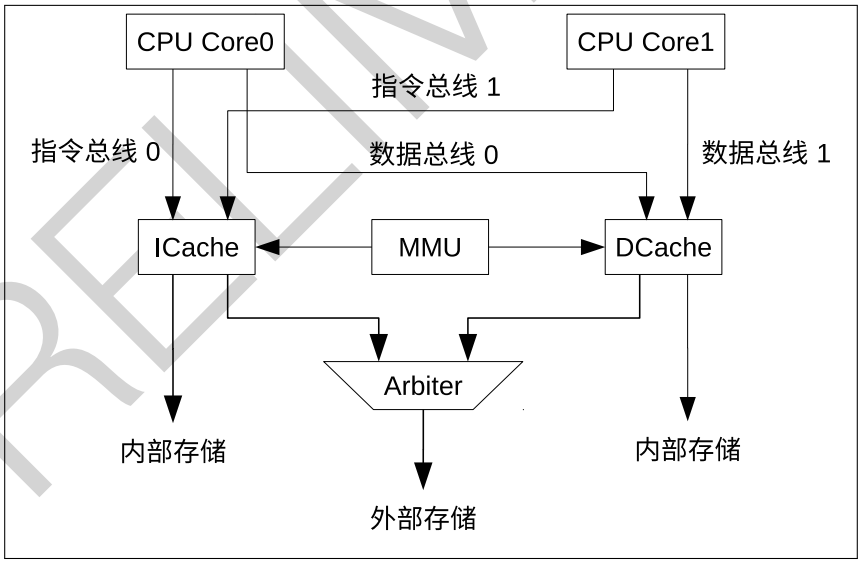


图 1-2. Cache 系统框图

1.3.3.3 Cache 操作

ESP32-S3 Cache 共有如下几种操作：

1. **Write-Back**：Write-Back 操作用于将 Cache 中的“脏块”的“脏”标记（Dirty bit）抹除，并将数据同步到外部存储器。Write-Back 操作结束后，外部存储器和 Cache 中存放的都是新数据。如果 CPU 接着去访问该数据，那么可以直接从 Cache 中访问到。只有 DCache 具有此功能。

   所谓“脏块”是指，如果 Cache 中的数据比外部存储器中的数据要新，则 Cache 中的新数据被称为“脏块”，Cache 通过“脏”标记来追踪这些“脏块”数据。如果抹除掉某个新数据的“脏”标记，Cache 将认为这个数据不是新的。

2. **Clean**：Clean 操作用于将 Cache 中的“脏块”的“脏”标记（Dirty bit）抹除，但不将数据同步到外部存储器。Clean 操作结束后，外部存储器中仍是旧数据，Cache 中存放的是新数据，但 Cache 以为不是新的。如果 CPU 接着去访问该数据，那么直接可以从 Cache 中访问到。只有 DCache 具有此功能。

3. **Invalidate**：Invalidate 操作用于删除 Cache 中的有效数据，即使该数据是“脏块”，也不会将脏块同步到外部存储器。对于非脏块数据，Invalidate 操作结束后，该数据仅存在于外部存储器中。如果 CPU 接着去访问该数据，那么需要访问外部存储器。对于脏块数据，invalidate 结束后，外部存储器中存在的是旧数据，新数据将彻底丢失。Invalidate 分为自动失效 (Auto-Invalidate) 和手动失效 (Manual-Invalidate)。Manual-Invalidate 仅对 Cache 中落入指定区域的地址对应的数据做失效处理，而 Auto-Invalidate 会对 Cache 中的所有数据做失效处理。ICache 和 DCache 均具有此功能。

4. **预取** (Preload)：Preload 功能用于将指令和数据提前加载到 Cache 中。预取操作的最小单位为 1 个块。预取分为手动预取 (Manual-Preload) 和自动预取 (Auto-Preload)，Manual-Preload 是指硬件按软件指定的虚地址预取一段连续的数据；Auto-Preload 是指硬件根据当前命中/缺失（取决于配置）的地址，自动地预取一段连续的数据。ICache 和 DCache 均具有此功能。

5. **锁定/解锁** (Lock/Unlock)：Lock 操作用于保护 Cache 中的数据不被替换掉。锁定分为预锁定和手动锁定。预锁定开启时，Cache 在填充缺失数据到 Cache 时，如果该数据落在指定区域，则将该数据锁定，未落入指定区域的数据不会被锁定。手动锁定开启时，Cache 检查 Cache 中的数据，并将落在指定区域的数据锁定，未落入指定区域的数据不会被锁定。当缺失发生时，Cache 会优先替换掉未被锁定的那一路（way）的数据，因此锁定区域的数据会一直保存在 Cache 中。但当所有路都被锁定时，Cache 将进行正常替换，就像所有路都没有被锁定一样。解锁是锁定的逆操作，但解锁只有手动解锁。ICache 和 DCache 均具有此功能。

   需要注意的是，Cache 的 Clean、Write-Back 和 Manual-Invalidate 操作均只对未被锁定的数据起作用。如果想对被锁定的数据执行这些操作，请先解锁这些数据。

1.3.4 GDMA 地址空间

ESP32-S3 中的通用直接存储访问（General Direct Memory Access, GDMA）外设可以提供直接存储访问（Direct Memory Access, DMA）服务，包括：

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

GDMA 可以通过与 CPU 数据总线完全相同的地址访问 Internal SRAM 1 与 Internal SRAM 2，即通过地址 0x3FC8_8000 ~ 0x3FCE_FFFF 访问 Internal SRAM 1，通过地址 0x3FCF_0000 ~ 0x3FCF_FFFF 访问 Internal SRAM 2。但 GDMA 无法访问被 Cache 占用的内部存储器。

GDMA 可以通过与 CPU 访问 DCache 相同的地址 (0x3C00_0000 ~ 0x3DFF_FFFF) 来访问外部存储器，但只可以访问片外 RAM。当 GDMA 与 DCache 同时访问外部存储器时，数据一致性问题需要由软件来保证。

另外，ESP32-S3 中的某些外设/模块可以和 GDMA 联合工作，此时 GDMA 可以为这些外设提供如下服务：

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

ESP32-S3 中有 11 个外设/模块可以和 GDMA 联合工作，如图 1-3 所示。其中的 11 根竖线依次对应这 11 个具有 GDMA 功能的外设/模块，横线表示 GDMA 的某一个通道（可以是任意一个通道），竖线与横线的交点表示对应外设/模块可以访问 GDMA 的某一个通道。同一行上有多个交点则表示这几个外设/模块不可以同时开启 GDMA 功能。

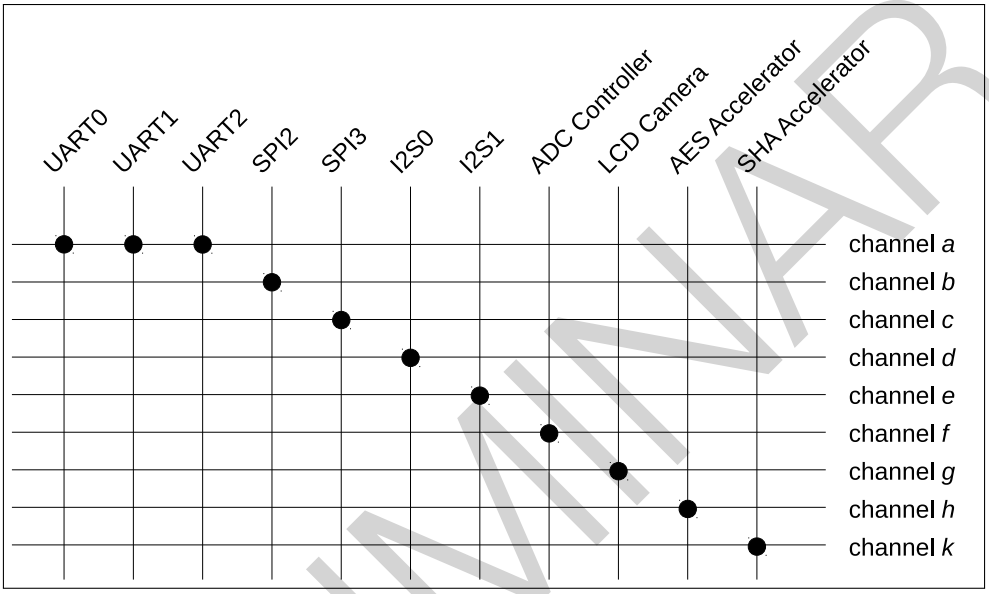


图 1-3. 具有 GDMA 功能的外设

具有 GDMA 功能的模块/外设通过 GDMA 可以访问任何 GDMA 可以访问到的存储器。更多关于 GDMA 的信息，请参考章节 9 *通用 DMA 控制器 (DMA) [to be added later]*。

说明：

当使用 GDMA 访问任何存储器时，都需要获取对应的访问权限，否则访问将会失败。关于权限管理的更多信息，请参考章节 7 *权限控制 (PMS) [to be added later]*。

1.3.5 模块/外设地址空间

CPU 可以通过数据/指令总线的共用地址段 0x6000_0000 ~ 0x600D_0FFF 来访问模块/外设。

1.3.5.1 模块/外设地址空间列表

表 1-4 详细列出了模块/外设地址空间的各段地址与其能访问到的模块/外设的映射关系。其中，“边界地址”栏中的两列数值共同决定了对应模块/外设的地址空间。

表 1-4. 模块/外设地址空间映射表

目标	边界地址		容量	说明
	低位地址	高位地址		
UART 控制器 0	0x6000_0000	0x6000_0FFF	4 KB	
保留	0x6000_1000	0x6000_1FFF		
SPI 控制器 1	0x6000_2000	0x6000_2FFF	4 KB	
SPI 控制器 0	0x6000_3000	0x6000_3FFF	4 KB	
GPIO	0x6000_4000	0x6000_4FFF	4 KB	
保留	0x6000_5000	0x6000_6FFF		
eFuse 控制器	0x6000_7000	0x6000_7FFF	4 KB	
低功耗管理	0x6000_8000	0x6000_8FFF	4 KB	
IO MUX	0x6000_9000	0x6000_9FFF	4 KB	
保留	0x6000_A000	0x6000_EFFF		
I2S 控制器 0	0x6000_F000	0x6000_FFFF	4 KB	
UART 控制器 1	0x6001_0000	0x6001_0FFF	4 KB	
保留	0x6001_1000	0x6001_2FFF		
I2C 控制器 0	0x6001_3000	0x6001_3FFF	4 KB	
UHCI0	0x6001_4000	0x6001_4FFF	4 KB	
保留	0x6001_5000	0x6001_5FFF		
红外遥控	0x6001_6000	0x6001_6FFF	4 KB	
脉冲计数控制器	0x6001_7000	0x6001_7FFF	4 KB	
保留	0x6001_8000	0x6001_8FFF		
LED PWM 控制器	0x6001_9000	0x6001_9FFF	4 KB	
保留	0x6001_A000	0x6001_DFFF		
电机控制器 0	0x6001_E000	0x6001_EFFF	4 KB	
定时器组 0	0x6001_F000	0x6001_FFFF	4 KB	
定时器组 1	0x6002_0000	0x6002_0FFF	4 KB	
RTC SLOW Memory	0x6002_1000	0x6002_2FFF	8 KB	
系统定时器	0x6002_3000	0x6002_3FFF	4 KB	
SPI 控制器 2	0x6002_4000	0x6002_4FFF	4 KB	
SPI 控制器 3	0x6002_5000	0x6002_5FFF	4 KB	
APB 控制器	0x6002_6000	0x6002_6FFF	4 KB	
I2C 控制器 1	0x6002_7000	0x6002_7FFF	4 KB	
SD/MMC 主机控制器	0x6002_8000	0x6002_8FFF	4 KB	
保留	0x6002_9000	0x6002_AFFF		
双线汽车接口	0x6002_B000	0x6002_BFFF	4 KB	
电机控制器 1	0x6002_C000	0x6002_CFFF	4 KB	
I2S 控制器 1	0x6002_D000	0x6002_DFFF	4 KB	
UART 控制器 2	0x6002_E000	0x6002_EFFF	4 KB	
保留	0x6002_F000	0x6003_7FFF		
USB Serial/JTAG 控制器	0x6003_8000	0x6003_8FFF	4 KB	
USB 外部控制寄存器	0x6003_9000	0x6003_9FFF	4 KB	1
AES 加速器	0x6003_A000	0x6003_AFFF	4 KB	
SHA 加速器	0x6003_B000	0x6003_BFFF	4 KB	

目标	边界地址		容量	说明
	低位地址	高位地址		
RSA 加速器	0x6003_C000	0x6003_CFFF	4 KB	
数字签名	0x6003_D000	0x6003_DFFF	4 KB	
HMAC 加速器	0x6003_E000	0x6003_EFFF	4 KB	
通用 DMA 控制器	0x6003_F000	0x6003_FFFF	4 KB	
ADC 控制器	0x6004_0000	0x6004_0FFF	4 KB	
Camera 与 LCD 控制器	0x6004_1000	0x6004_1FFF	4 KB	
保留	0x6004_2000	0x6007_FFFF		
USB 内核寄存器	0x6008_0000	0x600B_FFFF	256 KB	1
系统寄存器	0x600C_0000	0x600C_0FFF	4 KB	
Sensitive Register	0x600C_1000	0x600C_1FFF	4 KB	
中断矩阵	0x600C_2000	0x600C_2FFF	4 KB	
保留	0x600C_3000	0x600C_3FFF		
Configure Cache	0x600C_4000	0x600C_BFFF	32 KB	
片外存储器加密与解密	0x600C_C000	0x600C_CFFF	4 KB	
保留	0x600C_D000	0x600C_DFFF		
辅助调试	0x600C_E000	0x600C_EFFF	4 KB	
保留	0x600C_F000	0x600C_FFFF		
World 控制器	0x600D_0000	0x600D_0FFF	4 KB	

说明：

1. 该模块/外设的地址空间不连续。
2. CPU 要想访问某一个模块/外设，需要先获取该模块/外设的访问权限，否则访问将不会被响应。关于权限管理的更多信息，请参考章节 7 权限控制 *(PMS) [to be added later]*。

2 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)

2.1 概述

ESP32-S3 芯片有 45 个物理通用输入输出管脚 (GPIO Pin)。每个管脚都可用作一个通用输入输出，或连接一个内部外设信号。利用 GPIO 交换矩阵、IO MUX 和 RTC IO MUX，可配置外设模块的输入信号来源于任何的 GPIO 管脚，并且外设模块的输出信号也可连接到任意 GPIO 管脚。这些模块共同组成了芯片的输入输出控制。

注意：这 45 个物理 GPIO 管脚的编号为：0 ~ 21、26 ~ 48。这些管脚既可作为输入又可作为输出管脚。

2.2 特性

GPIO 交换矩阵特性

- GPIO 交换矩阵是外设输入输出信号和 GPIO 管脚之间的全交换矩阵。
- 175 个数字外设输入信号可以选择任意一个 GPIO 管脚的输入信号。
- 每个 GPIO 管脚的输出信号可以来自 184 个数字外设输出信号的任意一个。制；
- 支持输入信号经 GPIO SYNC 模块同步至 APB 时钟总线；
- 支持输入信号滤波；
- 支持 Sigma Delta 调制输出 (SDM)；
- 支持 GPIO 简单输入输出；

IO MUX 特性

- 为每个 GPIO 管脚提供一个寄存器 IO_MUX_GPIO*n*_REG，每个管脚可配置成：
  - GPIO 功能，连接 GPIO 交换矩阵；
  - 直连功能，旁路 GPIO 交换矩阵。
- 支持快速信号如 SPI、JTAG、UART 等可以旁路 GPIO 交换矩阵以实现更好的高频数字特性。所以高速信号会直接通过 IO MUX 输入和输出。

RTC IO MUX 特性

- 控制 22 个 RTC GPIO 管脚的低功耗特性；
- 控制 22 个 RTC GPIO 管脚的模拟功能；
- 将 22 个 RTC 输入输出信号引入 RTC 系统。

2.3 结构概览

图 2-1 所示为 GPIO 交换矩阵、IO MUX 和 RTC IO MUX 将信号引入外设和引出至管脚的具体过程。

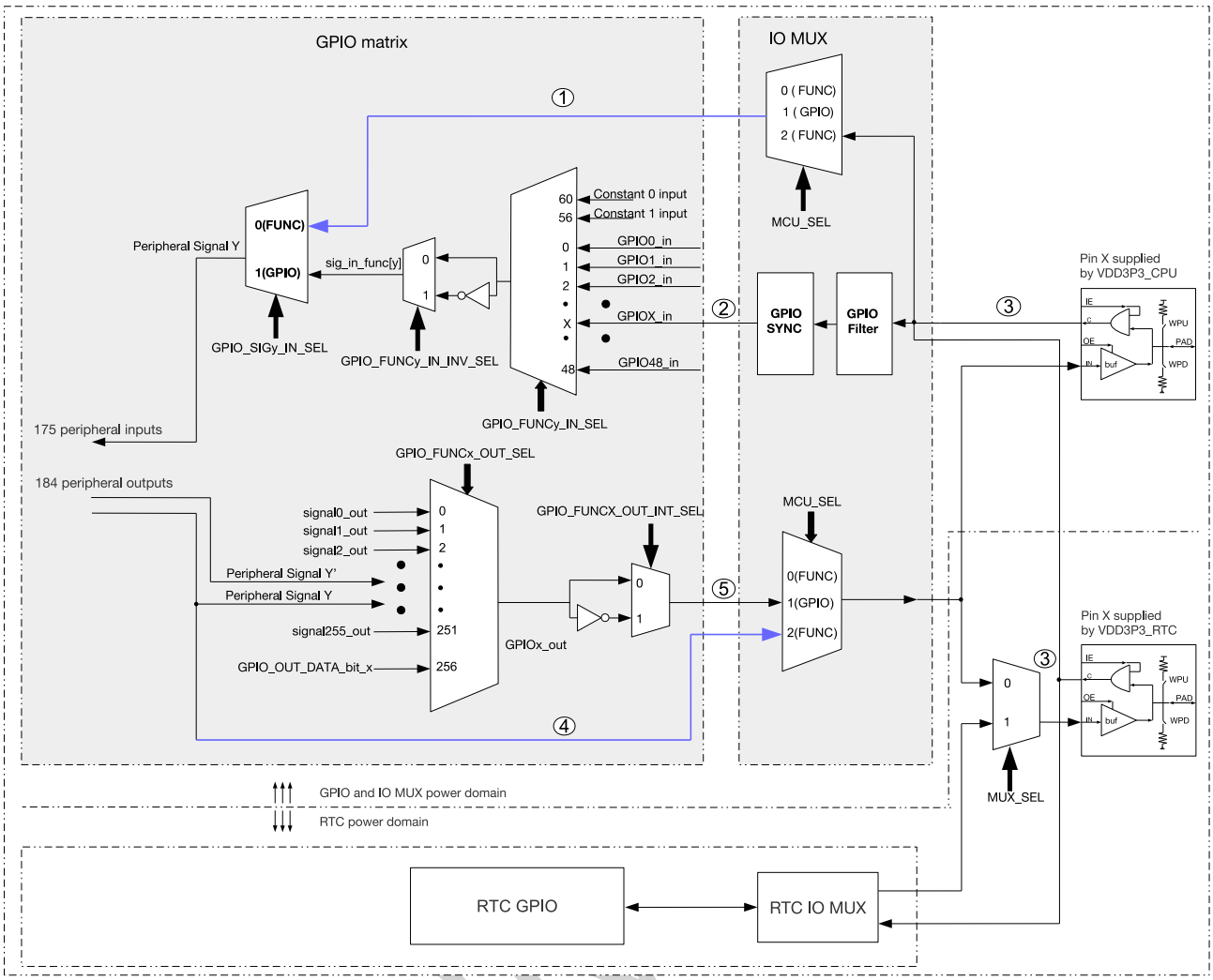


图 2-1. IO MUX、RTC IO MUX 和 GPIO 交换矩阵结构框图

1. 注意，外设输入信号中，仅有索引号为 0 ~ 3、7 ~ 13、15 ~ 16、101 ~ 110、120 ~ 123、155 ~ 159 的输入信号可以通过 IO MUX 直连外设。剩余其它信号只能通过 GPIO 交换矩阵连接至外设；
2. ESP32-S3 共有 45 个 GPIO 管脚，因此从 GPIO SYNC 进入到 GPIO 交换矩阵的输入共有 45 个；
3. 位于 VDD3P3_CPU 电源域和 VDD3P3_RTC 电源域的管脚由 IE、OE、WPU 和 WPD 信号控制；
4. 仅有部分外设输出信号 (0 ~ 13、15 ~ 16、101 ~ 110、120 ~ 126) 可通过 IO MUX 直连管脚。可以实现直连功能的信号见表 2-2；
5. 从 GPIO 交换矩阵到 IO MUX 的输出共有 45 个，对应 GPIO X：0 ~ 21、26 ~ 48。

图 2-2 展示了芯片焊盘 (PAD) 的内部结构，即芯片逻辑与 GPIO 管脚之间的电气接口。45 个 GPIO 管脚均采用这一结构，且由 IE、OE、WPU 和 WPD 信号控制。

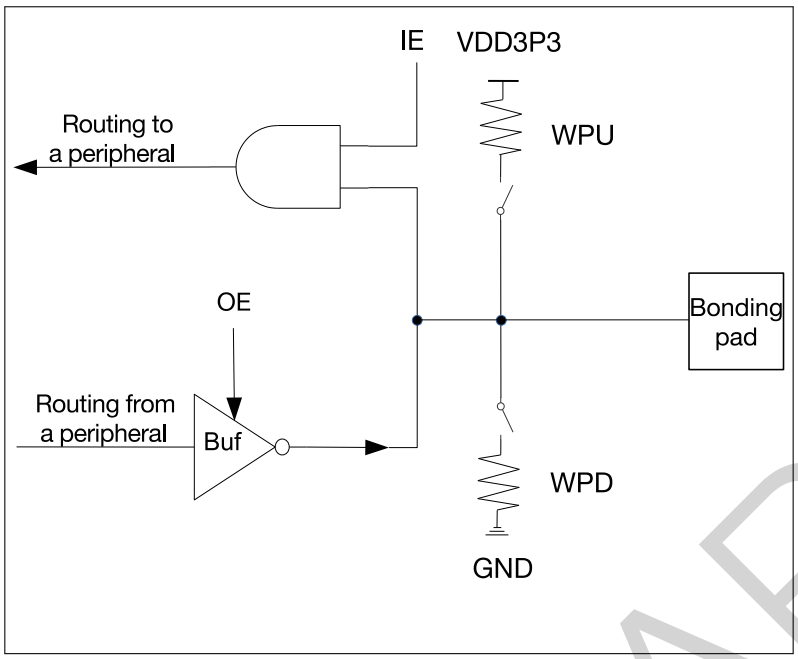


图 2-2. 焊盘内部结构

说明：

- IE：输入使能
- OE：输出使能
- WPU：内部弱上拉
- WPD：内部弱下拉
- Bonding pad：接合焊盘，芯片逻辑的结点，实现芯片封装内晶片与 GPIO 管脚之间的物理连接。

2.4 通过 GPIO 交换矩阵的外设输入

2.4.1 概述

为实现通过 GPIO 交换矩阵接收外设输入信号，需要配置 GPIO 交换矩阵从 45 个 GPIO（0 ~ 21、26 ~ 48）中获取外设输入信号，见交换矩阵表格 2-2。并需要配置外设输入选择通过 GPIO 交换矩阵接收输入信号。

2.4.2 信号同步

如图 2-1 所示，对于信号输入，外部输入信号从 GPIO 管脚输入，经 GPIO SYNC 模块同步至 APB 总线时钟后进入 GPIO 交换矩阵。外部输入信号也可以通过 IO MUX 直接进入外设，但信号无法经由 GPIO SYNC 模块同步。

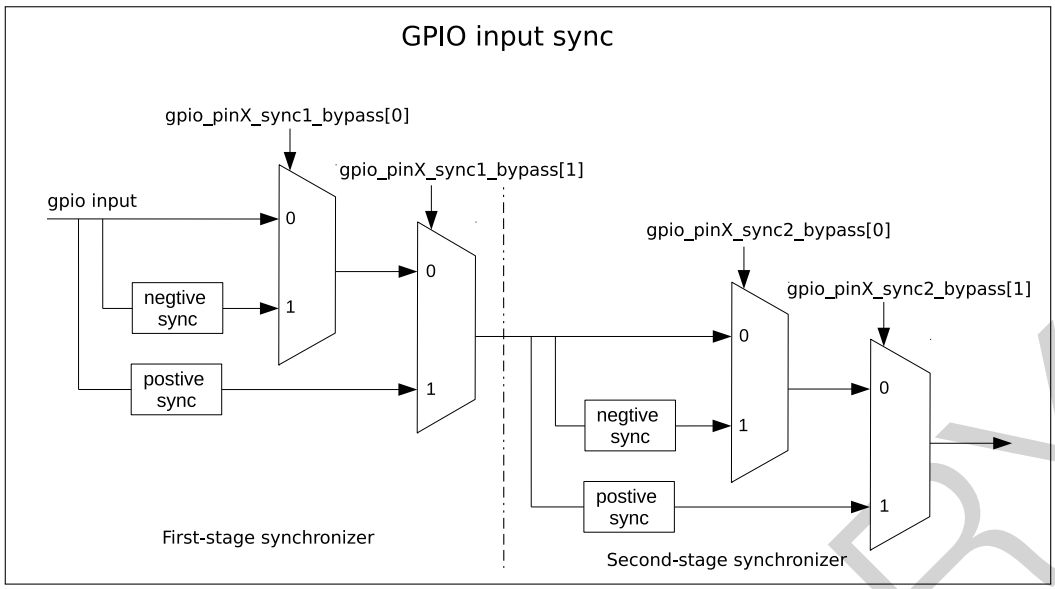


图 2-3. GPIO 输入经 APB 时钟上升沿或下降沿同步

GPIO SYNC 模块的功能如图 2-3 所示。其中，negative sync 表示 GPIO 输入信号经 APB 时钟的下降沿同步，positive sync 表示 GPIO 输入信号经 APB 时钟上升沿同步。

2.4.3 功能描述

把某个外设输入信号 *Y* 绑定到某个 GPIO 管脚 *X* [1] 的配置过程如下：

1. 在 GPIO 交换矩阵中配置外设信号 *Y* 的 GPIO_FUNC*y*_IN_SEL_CFG_REG 寄存器：
   - 置位 GPIO_SIG*y*_IN_SEL 选择通过 GPIO 交换矩阵接收外部输入信号；
   - 设置 GPIO_FUNC*y*_IN_SEL 为需要的 GPIO 管脚编号，此处应为 *X*。

   注意： 并不是所有外设信号都有有效的 GPIO_SIG*y*_IN_SEL，即有些外设信号只能通过 GPIO 交换矩阵接收外部输入信号。
2. 可选：置位 IO_MUX_FILTER_EN 使能 GPIO 管脚的输入信号滤波功能，如图 2-4 所示。只有当输入信号的有效宽度大于两个 APB 时钟周期时，输入信号才会被采样。否则，输入信号将会被滤掉。

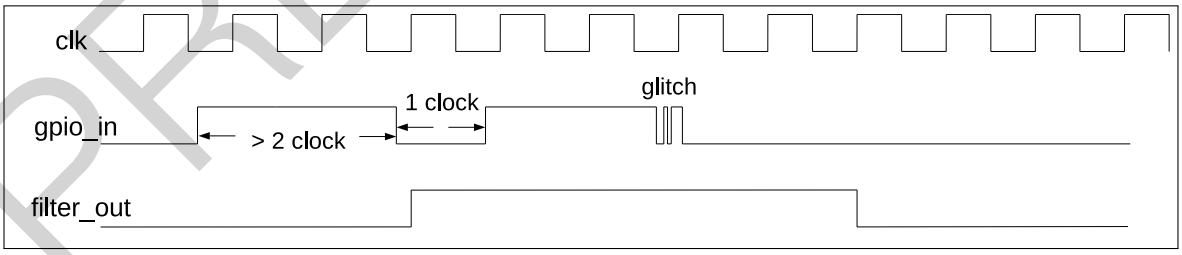


图 2-4. GPIO 输入信号滤波时序图

3. 同步 GPIO 输入信号。配置 GPIO 管脚 *X* 的 GPIO_PIN*x*_REG 来同步 GPIO 输入信号，过程如下：
   - 如图 2-3 所示，配置 GPIO_PIN*x*_SYNC1_BYPASS 使能输入信号第一拍为上升沿或下降沿同步；
   - 如图 2-3 所示，配置 GPIO_PIN*x*_SYNC2_BYPASS 使能输入信号第二拍为上升沿或下降沿同步。
4. 配置 IO MUX 寄存器使能 GPIO 管脚的输入功能。配置 GPIO 管脚 *X* 的 IO_MUX_*x*_REG，过程如下：

- 置位 IO_MUX_FUN_IE 使能输入 [2]；
- 置位或清零 IO_MUX_FUN_WPU 和 IO_MUX_FUN_WPD，使能或关闭内部上拉/下拉电阻。

例如，要把 RMT 外设通道 0 的输入信号 rmt_sig_in0[3]（信号索引号 81）绑定到 GPIO40，请按照以下步骤操作。注意，GPIO40 也叫做 MTDO 管脚：

1. 置位 GPIO_FUNC81_IN_SEL_CFG_REG 寄存器中 GPIO_SIG81_IN_SEL 位，使能通过 GPIO 交换矩阵接收外部输入信号；
2. 将 GPIO_FUNC81_IN_SEL_CFG_REG 寄存器中 GPIO_FUNC81_IN_SEL 字段设置为 40，即选择管脚 GPIO40；
3. 置位 IO_MUX_GPIO40_REG 寄存器中 IO_MUX_FUN_IE 位使能管脚输入。

说明：

1. 同一个输入管脚可以同时绑定多个内部输入信号；
2. 置位 GPIO_FUNC*y*_IN_INV_SEL 可以把输入的信号取反；
3. 无需将输入信号绑定到一个 GPIO 管脚也可以使外设读取恒低或恒高电平的输入值。实现方式为选择特定的 GPIO_FUNC*y*_IN_SEL 输入值而不是一个 GPIO 序号：
   - 当 GPIO_FUNC*y*_IN_SEL 是 0x3C 时，输入信号始终为 0；
   - 当 GPIO_FUNC*y*_IN_SEL 是 0x38 时，输入信号始终为 1。

2.4.4 简单 GPIO 输入

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

任意 GPIO 管脚的输入值都可以随时读取而无需为某一个外设信号配置 GPIO 交换矩阵。但是需要配置 GPIO 管脚 *X* 对应的 IO_MUX_*x*_REG 寄存器中 IO_MUX_FUN_IE 位以使能输入，如章节 2.4.2 所述。

2.5 通过 GPIO 交换矩阵的外设输出

2.5.1 概述

为实现通过 GPIO 交换矩阵输出外设信号，需要配置 GPIO 交换矩阵将输出索引号为（0 ~ 32、54、60 ~ 84、89 ~ 187、208 ~ 228、251 ~ 250）的外设信号输出到 45 个 GPIO 管脚（0 ~ 21、26 ~ 48）。具体输出索引号见交换矩阵表格 2-2。

输出信号从外设输出到 GPIO 交换矩阵，然后到达 IO MUX。IO MUX 必须设置相应管脚为 GPIO 功能，这样输出信号就能连接到相应管脚。

说明：

输出索引号为 208 ~ 212 的输出信号，与输入索引号为 208 ~ 212 的输入信号对应相连。可配置从一个 GPIO 管脚输入，直接由另一个 GPIO 管脚输出。

2.5.2 功能描述

如图 2-1 所示，对于信号输出，256 个输出信号中的某一个信号通过 GPIO 交换矩阵到达 IO MUX，然后连接到某个 GPIO 管脚。

输出外设信号 *Y* 到某一 GPIO 管脚 *X*[1，2] 的步骤为：

1. 在 GPIO 交换矩阵里配置 GPIO 管脚 *X* 的 GPIO_FUNC*x*_OUT_SEL_CFG_REG 寄存器和 GPIO_ENABLE_REG[*x*] 字段。推荐使用相应 GPIO_ENABLE_W1TS_REG (write 1 to set) 或 GPIO_ENABLE_W1TC_REG (write 1 to clear) 寄存器来更新 GPIO_ENABLE_REG 中的值：
   - 设置 GPIO_FUNC*x*_OUT_SEL_CFG_REG 寄存器的 GPIO_FUNC*x*_OUT_SEL 字段为外设输出信号 *Y* 的索引号 (*Y*)。
   - 要将信号强制使能为输出模式，需要将 GPIO 管脚 *X* 对应的 GPIO_FUNC*x*_OUT_SEL_CFG_REG 寄存器中 GPIO_FUNC*x*_OEN_SEL 字段置位；同时需要将 GPIO_ENABLE_W1TS_REG 或 GPIO_ENABLE1_W1TS_REG 中相应字段置位。或者，将 GPIO_FUNC*x*_OEN_SEL 清零，即选择采用外设的输出使能信号，此时输出使能信号由内部逻辑功能决定。比如，表 2-2 中 “GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号” 一栏的 SPIQ_oe 信号。
   - 置位 GPIO_ENABLE_W1TC_REG 或 GPIO_ENABLE1_W1TC_REG 中相应位可以关闭 GPIO 管脚的输出。
2. 要选择以开漏方式输出，可以设置 GPIO 管脚 *X* 的 GPIO_PIN*x*_REG 寄存器中 GPIO_PIN*x*_PAD_DRIVER 位。
3. 配置 IO MUX 寄存器来选择经由 GPIO 交换矩阵输出信号。配置 GPIO 管脚 *X* 相应寄存器 IO_MUX_*x*_REG 的过程如下：
   - 配置 GPIO 管脚 *X* 的 IO_MUX_MCU_SEL 为所需的管脚功能。此处选择数值 1，即 Function 1（GPIO 功能），适用于所有管脚。
   - 设置 IO_MUX_FUN_DRV 字段为特定的输出强度值 (0 ~ 3)，值越大，输出驱动能力越强：
     - 0: ~5 mA
     - 1: ~10 mA
     - 2: ~20 mA（默认值）
     - 3: ~40 mA
   - 在开漏模式下，通过置位/清零 IO_MUX_FUN_WPU 和 IO_MUX_FUN_WPD 使能或关闭上拉/下拉电阻。

说明：

1. 某一个外设的输出信号可以同时从多个管脚输出；
2. 置位 GPIO_FUNC*x*_OUT_INV_SEL 可以把输出的信号取反。

2.5.3 简单 GPIO 输出

GPIO 交换矩阵也可用于简单 GPIO 输出，具体配置如下：

- 设置 GPIO 交换矩阵 GPIO_FUNC*n*_OUT_SEL 为特定的外设索引值 256 (0x100)；
- 设置 GPIO_OUT_REG[31:0] 或者 GPIO_OUT1_REG[21:0] 寄存器中相应位的值为期望 GPIO 输出的值。

说明：

- GPIO_OUT_REG[0] ~ GPIO_OUT_REG[21] 对应 GPIO0 ~ GPIO21；GPIO_OUT_REG[26] ~ GPIO_OUT_REG[31] 对应 GPIO26 ~ GPIO31。GPIO_OUT_REG[25:22] 无效。

- GPIO_OUT1_REG[0] ~ GPIO_OUT1_REG[16] 对应 GPIO32 ~ GPIO48，GPIO_OUT1_REG[21:17] 无效。
- 推荐使用相应的 W1TS (write 1 to set) 和 W1TC (write 1 to clear) 寄存器，例如：GPIO_OUT_W1TS/GPIO_OUT_W1TC 来置位/清零 GPIO_OUT_REG 或 GPIO_OUT1_REG。

2.5.4 Sigma Delta 调制输出 (SDM)

2.5.4.1 功能描述

256 个数字外设输出中有 8 个信号（索引：93 ~ 100）支持 1-bit 二阶 Sigma Delta 调制输出。上述 8 个信号通道默认输出使能。Sigma Delta 调制器可实现输出可配占空比的 PDM（脉冲密度调制）信号。二阶 Sigma Delta 调制器传输函数为：

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

E(z) 为量化误差，X(z) 为输入。
Sigma Delta 调制器内部支持对 APB_CLK 的 1 ~ 256 倍分频：

- 置位 GPIO_FUNCTION_CLK_EN 使能调制器时钟；
- 配置寄存器 GPIO_SD*n*_PRESCALE 实现分频。*n* 为 0 ~ 7，对应 8 个通道。

分频后的时钟周期为调制器输出单位脉冲的周期。

GPIO_SD*n*_IN 为有符号数，范围为 [-128, 127]，配置此寄存器控制输出 PDM 信号的占空比 [1]。

- GPIO_SD*n*_IN = -128，调制器输出信号占空比为 0%；
- GPIO_SD*n*_IN = 0，调制器输出信号占空比接近 50%；
- GPIO_SD*n*_IN = 127，调制器输出信号占空比接近 100%。

PDM 信号占空比计算公式为：

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

说明：
对 PDM 信号来说，占空比是指在若干脉冲周期内（比如 256 个脉冲周期），高电平占整个统计周期的比值。

2.5.4.2 配置方法

SDM 的配置方法如下：

- 将 SDM 输出经 GPIO 交换矩阵连接至管脚，见章节 2.5.2；
- 置位 GPIO_FUNCTION_CLK_EN，使能 SDM 时钟；
- 配置 GPIO_SD*n*_PRESCALE 设置时钟分频系数；
- 配置 GPIO_SD*n*_IN 设置 SDM 输出信号的占空比。

2.6 IO MUX 的直接输入输出功能

2.6.1 概述

快速信号如 SPI、JTAG 等会旁路 GPIO 交换矩阵以实现更好的高频数字特性。所以高速信号会直接通过 IO MUX 输入和输出。

这样比使用 GPIO 交换矩阵的灵活度要低，即每个 GPIO 管脚的 IO MUX 寄存器只有较少的功能选择，但可以实现更好的高频数字特性。

2.6.2 功能描述

对于外设输入信号，旁路 GPIO 交换矩阵必须配置两个寄存器：

1. GPIO 管脚的 IO_MUX_MCU_SEL 必须设置为相应的管脚功能，章节 2.12 列出了管脚功能。
2. 清零 GPIO_SIG*n*_IN_SEL，直接将输入信号连接到外设。

对于外设输出信号，旁路 GPIO 交换矩阵只需将 GPIO 管脚的 IO_MUX_MCU_SEL 配置为相应的管脚功能即可。

说明：
并非所有外设输入/输出信号均可直接通过 IO MUX 连接到外设，某些输入/输出信号只能通过 GPIO 交换矩阵连接到外设。

2.7 RTC IO MUX 的低功耗性能和模拟输入输出功能

2.7.1 概述

ESP32-S3 中有 22 个 GPIO 管脚具有低功耗 (RTC) 性能和模拟功能，由 RTC 子系统控制。这些功能不使用 IO MUX 和 GPIO 交换矩阵，而是使用 RTC IO MUX 将 22 个 RTC 输入输出信号引入 RTC 子系统。

当这些管脚被配置为 RTC GPIO 管脚，作为输出管脚时仍然能够在芯片处于 Deep-sleep 模式下保持输出电平值或者作为输入管脚使用时可以将芯片从 Deep-sleep 中唤醒。

2.7.2 低功耗性能描述

每个管脚的 RTC 功能是由 RTC_IO_TOUCH/RTC_PAD*n*_REG 寄存器中的 RTC_IO_TOUCH/RTC_PAD*n*_MUX_SEL 位控制的。此位默认置为 0，通过 IO MUX 子系统输入输出信号，如前文所述。

如果置位 RTC_IO_TOUCH/RTC_PAD*n*_MUX_SEL，则输入输出信号会经过 RTC 子系统。在这种模式下，RTC_IO_TOUCH/RTC_PAD*n*_REG 寄存器用于控制 RTC 低功耗 GPIO 管脚。表 2-4 列出了 GPIO 管脚的 RTC 功能。请注意 RTC_IO_TOUCH/RTC_PAD*n*_REG 寄存器使用的是 RTC GPIO 管脚的序号，不是 GPIO 管脚的序号。

2.7.3 模拟功能描述

当使用管脚的模拟功能时，需要确保该管脚处于悬空状态，此时外部模拟信号通过 GPIO 管脚直接与内部的模拟信号相连。通常利用 RTC_IO_TOUCH/RTC_PAD*n*_REG 寄存器控制使管脚处于浮空状态。相关配置如下：

- 置位 RTC_IO_TOUCH/RTC_PAD*n*_MUX_SEL 选择通过 RTC IO MUX 输入输出信号；
- 同时清零 RTC_IO_TOUCH/RTC_PAD*n*_FUN_IE、RTC_IO_TOUCH/RTC_PAD*n*_FUN_RUE、RTC_IO_TOUCH/RTC_PAD*n*_FUN_RDE 将管脚设置为浮空状态；

- 配置 RTC_IO_TOUCH/RTC_PAD*n*_FUN_SEL 为 0，即选择模拟功能 0；
- 向 RTC_IO_GPIO_ENABLE_W1TC 中对应位写 1，清零输出使能。

表 2-5 列出了 GPIO 管脚的模拟功能。

2.8 Light-sleep 模式管脚功能

当 ESP32-S3 处于 Light-sleep 模式时管脚可以有不同的功能。如果某一 GPIO 管脚的 IO_MUX_*n*_REG 寄存器中 IO_MUX_SLP_SEL 位置为 1，芯片处于 Light-sleep 模式下将由另一组不同的寄存器控制管脚。

表 2-1. IO MUX Light-sleep 管脚功能控制寄存器

IO MUX 功能	正常工作模式 OR IO_MUX_SLP_SEL = 0	Light-sleep 模式 AND IO_MUX_SLP_SEL = 1
输出驱动强度	IO_MUX_FUN_DRV	IO_MUX_FUN_DRV
上拉电阻	IO_MUX_FUN_WPU	IO_MUX_MCU_WPU
下拉电阻	IO_MUX_FUN_WPD	IO_MUX_MCU_WPD
输出使能	由 GPIO 交换矩阵的 OEN_SEL 位控制 *	IO_MUX_MCU_OE

说明：

如果 IO_MUX_SLP_SEL 置为 0，则芯片在正常工作和 Light-sleep 模式下，管脚的功能一样。此时，具体的输出使能配置请参考 2.5.2 章节。

2.9 管脚 Hold 特性

每个 GPIO 管脚（包括 RTC 管脚）都有单独的 Hold 功能，由 RTC 寄存器控制。管脚的 Hold 功能被置上后，管脚在置上 Hold 那一刻的状态被强制保持，无论内部信号如何变化，修改 IO MUX 配置或者 GPIO 配置，都不会改变管脚的状态。应用如果希望在看门狗超时触发内核复位和系统复位时或者 Deep-sleep 时管脚的状态不被改变，就需要提前把 Hold 置上。

说明：

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

2.10 GPIO 管脚供电和电源管理

2.10.1 GPIO 管脚供电

GPIO 管脚供电请参考 《ESP32-S3 技术规格书》中管脚定义章节。

2.10.2 电源管理

ESP32-S3 的管脚可分为如下三种不同的电源域。

- VDD3P3_RTC：RTC 和 CPU 的输入电源
- VDD3P3_CPU：CPU 的输入电源
- VDD_SPI：可配置为输入电源或输出电源

VDD_SPI 可配置使用一个内置 LDO，该内置 LDO 的输入和输出均为 1.8 V。如未使能 LDO，VDD_SPI 可以与 VDD3P3_RTC 连接在相同的电源上。

VDD_SPI 的具体配置由 GPIO45 的 Strapping 值决定，用户可通过 eFuse 或寄存器修改 VDD_SPI 的配置。请参考《ESP32-S3 技术规格书》中的电源管理章节和 Strapping 管脚章节查看更多信息。

其中，GPIO33 ~ GPIO37 管脚既可以由 VDD_SPI 供电，也可以由 VDD3P3_CPU 供电。

2.11 GPIO 交换矩阵外设信号列表

表 2-2 列出了所有经由 GPIO 交换矩阵的外设输入输出信号。

请注意 GPIO_FUNC*n*_OEN_SEL 位的配置：

- GPIO_FUNC*n*_OEN_SEL = 1，则寄存器 GPIO_ENABLE_REG 中的相应位 *n* 将用于控制信号输出使能。
  - GPIO_ENABLE_REG = 0：输出关闭；
  - GPIO_ENABLE_REG = 1：输出使能；
- GPIO_FUNC*n*_OEN_SEL = 0，则输出信号的使能由外设控制，例如表 2-2 中 “GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号” 一栏的 SPIQ_oe。注意，使能信号 SPIQ_oe 可设置为 1 (1’d1) 或 0 (1’d0)，具体由外设的配置决定。如果 “GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号” 一栏中为 1’d1，则表示寄存器 GPIO_FUNC*n*_OEN_SEL 已清零，输出信号默认始终使能。

说明：

信号连续编号，但并非所有信号均有效。

- 输入信号中，仅有索引号为 0 ~ 3、7 ~ 48、51 ~ 54、58 ~ 62、66 ~ 71、73、81 ~ 84、89 ~ 92、101 ~ 110、116、120 ~ 123、129 ~ 131、133 ~ 152、155 ~ 187、192 ~ 199、208 ~ 228 和 251 ~ 255 的输入信号有效。
- 输出信号中，仅有索引号为 0 ~ 32、54、60 ~ 84、89 ~ 187、208 ~ 228 和 251 ~ 250 的输出信号有效。

表 2-2. GPIO 交换矩阵外设信号

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
0	SPIQ_in	0	yes	SPIQ_out	SPIQ_oe	yes
1	SPID_in	0	yes	SPID_out	SPID_oe	yes
2	SPIHD_in	0	yes	SPIHD_out	SPIHD_oe	yes
3	SPIWP_in	0	yes	SPIWP_out	SPIWP_oe	yes
4	-	-	-	SPICLK_out_mux	SPICLK_oe	yes
5	-	-	-	SPICS0_out	SPICS0_oe	yes
6	-	-	-	SPICS1_out	SPICS1_oe	yes
7	SPID4_in	0	yes	SPID4_out	SPID4_oe	yes
8	SPID5_in	0	yes	SPID5_out	SPID5_oe	yes
9	SPID6_in	0	yes	SPID6_out	SPID6_oe	yes
10	SPID7_in	0	yes	SPID7_out	SPID7_oe	yes
11	SPIDQS_in	0	yes	SPIDQS_out	SPIDQS_oe	yes
12	U0RXD_in	0	yes	U0TXD_out	1'd1	yes
13	U0CTS_in	0	yes	U0RTS_out	1'd1	yes
14	U0DSR_in	0	no	U0DTR_out	1'd1	no
15	U1RXD_in	0	yes	U1TXD_out	1'd1	yes
16	U1CTS_in	0	yes	U1RTS_out	1'd1	yes
17	U1DSR_in	0	no	U1DTR_out	1'd1	no
18	U2RXD_in	0	no	U2TXD_out	1'd1	no
19	U2CTS_in	0	no	U2RTS_out	1'd1	no
20	U2DSR_in	0	no	U2DTR_out	1'd1	no
21	I2S1_MCLK_in	0	no	I2S1_MCLK_out	1'd1	no
22	I2S0O_BCK_in	0	no	I2S0O_BCK_out	1'd1	no
23	I2S0_MCLK_in	0	no	I2S0_MCLK_out	1'd1	no
24	I2S0O_WS_in	0	no	I2S0O_WS_out	1'd1	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
25	I2S0I_SD_in	0	no	I2S0O_SD_out	1'd1	no
26	I2S0I_BCK_in	0	no	I2S0I_BCK_out	1'd1	no
27	I2S0I_WS_in	0	no	I2S0I_WS_out	1'd1	no
28	I2S1O_BCK_in	0	no	I2S1O_BCK_out	1'd1	no
29	I2S1O_WS_in	0	no	I2S1O_WS_out	1'd1	no
30	I2S1I_SD_in	0	no	I2S1O_SD_out	1'd1	no
31	I2S1I_BCK_in	0	no	I2S1I_BCK_out	1'd1	no
32	I2S1I_WS_in	0	no	I2S1I_WS_out	1'd1	no
33	pcnt_sig_ch0_in0	0	no	-	1'd1	no
34	pcnt_sig_ch1_in0	0	no	-	1'd1	no
35	pcnt_ctrl_ch0_in0	0	no	-	1'd1	-
36	pcnt_ctrl_ch1_in0	0	no	-	1'd1	-
37	pcnt_sig_ch0_in1	0	no	-	1'd1	-
38	pcnt_sig_ch1_in1	0	no	-	1'd1	-
39	pcnt_ctrl_ch0_in1	0	no	-	1'd1	-
40	pcnt_ctrl_ch1_in1	0	no	-	1'd1	-
41	pcnt_sig_ch0_in2	0	no	-	1'd1	-
42	pcnt_sig_ch1_in2	0	no	-	1'd1	-
43	pcnt_ctrl_ch0_in2	0	no	-	1'd1	-
44	pcnt_ctrl_ch1_in2	0	no	-	1'd1	-
45	pcnt_sig_ch0_in3	0	no	-	1'd1	-
46	pcnt_sig_ch1_in3	0	no	-	1'd1	-
47	pcnt_ctrl_ch0_in3	0	no	-	1'd1	-
48	pcnt_ctrl_ch1_in3	0	no	-	1'd1	-
49	-	-	-	-	1'd1	-
50	-	-	-	-	1'd1	-
51	I2S0I_SD1_in	0	no	-	1'd1	-

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
52	I2S0I_SD2_in	0	no	-	1'd1	-
53	I2S0I_SD3_in	0	no	-	1'd1	-
54	Core1_gpio_in7	0	no	Core1_gpio_out7	1'd1	no
55	-	-	-	-	1'd1	-
56	-	-	-	-	1'd1	-
57	-	-	-	-	1'd1	-
58	usb_otg_iddig_in	0	no	-	1'd1	-
59	usb_otg_avalid_in	0	no	-	1'd1	-
60	usb_srp_bvalid_in	0	no	usb_otg_idpullup	1'd1	no
61	usb_otg_vbusvalid_in	0	no	usb_otg_dppulldown	1'd1	no
62	usb_srp_sessend_in	0	no	usb_otg_dmpulldown	1'd1	no
63	-	-	-	usb_otg_drvvbus	1'd1	no
64	-	-	-	usb_srp_chrgvbus	1'd1	no
65	-	-	-	usb_srp_dischrgvbus	1'd1	no
66	SPI3_CLK_in	0	no	SPI3_CLK_out_mux	SPI3_CLK_oe	no
67	SPI3_Q_in	0	no	SPI3_Q_out	SPI3_Q_oe	no
68	SPI3_D_in	0	no	SPI3_D_out	SPI3_D_oe	no
69	SPI3_HD_in	0	no	SPI3_HD_out	SPI3_HD_oe	no
70	SPI3_WP_in	0	no	SPI3_WP_out	SPI3_WP_oe	no
71	SPI3_CS0_in	0	no	SPI3_CS0_out	SPI3_CS0_oe	no
72	-	-	-	SPI3_CS1_out	SPI3_CS1_oe	no
73	ext_adc_start	0	no	ledc_ls_sig_out0	1'd1	no
74	-	-	-	ledc_ls_sig_out1	1'd1	no
75	-	-	-	ledc_ls_sig_out2	1'd1	no
76	-	-	-	ledc_ls_sig_out3	1'd1	no
77	-	-	-	ledc_ls_sig_out4	1'd1	no
78	-	-	-	ledc_ls_sig_out5	1'd1	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
79	-	-	-	ledc_ls_sig_out6	1'd1	no
80	-	-	-	ledc_ls_sig_out7	1'd1	no
81	rmt_sig_in0	0	no	rmt_sig_out0	1'd1	no
82	rmt_sig_in1	0	no	rmt_sig_out1	1'd1	no
83	rmt_sig_in2	0	no	rmt_sig_out2	1'd1	no
84	rmt_sig_in3	0	no	rmt_sig_out3	1'd1	no
85	-	-	-	-	1'd1	-
86	-	-	-	-	1'd1	-
87	-	-	-	-	1'd1	-
88	-	-	-	-	1'd1	-
89	I2CEXT0_SCL_in	1	no	I2CEXT0_SCL_out	I2CEXT0_SCL_oe	no
90	I2CEXT0_SDA_in	1	no	I2CEXT0_SDA_out	I2CEXT0_SDA_oe	no
91	I2CEXT1_SCL_in	1	no	I2CEXT1_SCL_out	I2CEXT1_SCL_oe	no
92	I2CEXT1_SDA_in	1	no	I2CEXT1_SDA_out	I2CEXT1_SDA_oe	no
93	-	-	-	gpio_sd0_out	1'd1	no
94	-	-	-	gpio_sd1_out	1'd1	no
95	-	-	-	gpio_sd2_out	1'd1	no
96	-	-	-	gpio_sd3_out	1'd1	no
97	-	-	-	gpio_sd4_out	1'd1	no
98	-	-	-	gpio_sd5_out	1'd1	no
99	-	-	-	gpio_sd6_out	1'd1	no
100	-	-	-	gpio_sd7_out	1'd1	no
101	FSPICLK_in	0	yes	FSPICLK_out_mux	FSPICLK_oe	yes
102	FSPIQ_in	0	yes	FSPIQ_out	FSPIQ_oe	yes
103	FSPID_in	0	yes	FSPID_out	FSPID_oe	yes
104	FSPIHD_in	0	yes	FSPIHD_out	FSPIHD_oe	yes
105	FSPIWP_in	0	yes	FSPIWP_out	FSPIWP_oe	yes

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
106	FSPIIO4_in	0	yes	FSPIIO4_out	FSPIIO4_oe	yes
107	FSPIIO5_in	0	yes	FSPIIO5_out	FSPIIO5_oe	yes
108	FSPIIO6_in	0	yes	FSPIIO6_out	FSPIIO6_oe	yes
109	FSPIIO7_in	0	yes	FSPIIO7_out	FSPIIO7_oe	yes
110	FSPICS0_in	0	yes	FSPICS0_out	FSPICS0_oe	yes
111	-	-	-	FSPICS1_out	FSPICS1_oe	no
112	-	-	-	FSPICS2_out	FSPICS2_oe	no
113	-	-	-	FSPICS3_out	FSPICS3_oe	no
114	-	-	-	FSPICS4_out	FSPICS4_oe	no
115	-	-	-	FSPICS5_out	FSPICS5_oe	no
116	twai_rx	1	no	twai_tx	1'd1	no
117	-	-	-	twai_bus_off_on	1'd1	no
118	-	-	-	twai_clkout	1'd1	no
119	-	-	-	SUBSPICLK_out_mux	SUBSPICLK_oe	no
120	SUBSPIQ_in	0	yes	SUBSPIQ_out	SUBSPIQ_oe	yes
121	SUBSPID_in	0	yes	SUBSPID_out	SUBSPID_oe	yes
122	SUBSPIHD_in	0	yes	SUBSPIHD_out	SUBSPIHD_oe	yes
123	SUBSPIWP_in	0	yes	SUBSPIWP_out	SUBSPIWP_oe	yes
124	-	-	-	SUBSPICS0_out	SUBSPICS0_oe	yes
125	-	-	-	SUBSPICS1_out	SUBSPICS1_oe	yes
126	-	-	-	FSPIDQS_out	FSPIDQS_oe	yes
127	-	-	-	SPI3_CS2_out	SPI3_CS2_oe	no
128	-	-	-	I2S0O_SD1_out	1'd1	no
129	Core1_gpio_in0	0	no	Core1_gpio_out0	1'd1	no
130	Core1_gpio_in1	0	no	Core1_gpio_out1	1'd1	no
131	Core1_gpio_in2	0	no	Core1_gpio_out2	1'd1	no
132	-	-	-	LCD_CS	1'd1	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
133	CAM_DATA_in0	0	no	LCD_DATA_out0	1'd1	no
134	CAM_DATA_in1	0	no	LCD_DATA_out1	1'd1	no
135	CAM_DATA_in2	0	no	LCD_DATA_out2	1'd1	no
136	CAM_DATA_in3	0	no	LCD_DATA_out3	1'd1	no
137	CAM_DATA_in4	0	no	LCD_DATA_out4	1'd1	no
138	CAM_DATA_in5	0	no	LCD_DATA_out5	1'd1	no
139	CAM_DATA_in6	0	no	LCD_DATA_out6	1'd1	no
140	CAM_DATA_in7	0	no	LCD_DATA_out7	1'd1	no
141	CAM_DATA_in8	0	no	LCD_DATA_out8	1'd1	no
142	CAM_DATA_in9	0	no	LCD_DATA_out9	1'd1	no
143	CAM_DATA_in10	0	no	LCD_DATA_out10	1'd1	no
144	CAM_DATA_in11	0	no	LCD_DATA_out11	1'd1	no
145	CAM_DATA_in12	0	no	LCD_DATA_out12	1'd1	no
146	CAM_DATA_in13	0	no	LCD_DATA_out13	1'd1	no
147	CAM_DATA_in14	0	no	LCD_DATA_out14	1'd1	no
148	CAM_DATA_in15	0	no	LCD_DATA_out15	1'd1	no
149	CAM_PCLK	0	no	CAM_CLK	1'd1	no
150	CAM_H_ENABLE	0	no	LCD_H_ENABLE	1'd1	no
151	CAM_H_SYNC	0	no	LCD_H_SYNC	1'd1	no
152	CAM_V_SYNC	0	no	LCD_V_SYNC	1'd1	no
153	-	-	-	LCD_DC	1'd1	no
154	-	-	-	LCD_PCLK	1'd1	no
155	SUBSPID4_in	0	yes	SUBSPID4_out	SUBSPID4_oe	no
156	SUBSPID5_in	0	yes	SUBSPID5_out	SUBSPID5_oe	no
157	SUBSPID6_in	0	yes	SUBSPID6_out	SUBSPID6_oe	no
158	SUBSPID7_in	0	yes	SUBSPID7_out	SUBSPID7_oe	no
159	SUBSPIDQS_in	0	yes	SUBSPIDQS_out	SUBSPIDQS_oe	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
160	pwm0_sync0_in	0	no	pwm0_out0a	1'd1	no
161	pwm0_sync1_in	0	no	pwm0_out0b	1'd1	no
162	pwm0_sync2_in	0	no	pwm0_out1a	1'd1	no
163	pwm0_f0_in	0	no	pwm0_out1b	1'd1	no
164	pwm0_f1_in	0	no	pwm0_out2a	1'd1	no
165	pwm0_f2_in	0	no	pwm0_out2b	1'd1	no
166	pwm0_cap0_in	0	no	pwm1_out0a	1'd1	no
167	pwm0_cap1_in	0	no	pwm1_out0b	1'd1	no
168	pwm0_cap2_in	0	no	pwm1_out1a	1'd1	no
169	pwm1_sync0_in	0	no	pwm1_out1b	1'd1	no
170	pwm1_sync1_in	0	no	pwm1_out2a	1'd1	no
171	pwm1_sync2_in	0	no	pwm1_out2b	1'd1	no
172	pwm1_f0_in	0	no	sdhost_cclk_out_1	1'd1	no
173	pwm1_f1_in	0	no	sdhost_cclk_out_2	1'd1	no
174	pwm1_f2_in	0	no	sdhost_rst_n_1	1'd1	no
175	pwm1_cap0_in	0	no	sdhost_rst_n_2	1'd1	no
176	pwm1_cap1_in	0	no	sdhost_ccmd_od_pullup_en_n	1'd1	no
177	pwm1_cap2_in	0	no	sdio_tohost_int_out	1'd1	no
178	sdhost_ccmd_in_1	1	no	sdhost_ccmd_out_1	sdhost_ccmd_out_en_1	no
179	sdhost_ccmd_in_2	1	no	sdhost_ccmd_out_2	sdhost_ccmd_out_en_2	no
180	sdhost_cdata_in_10	1	no	sdhost_cdata_out_10	sdhost_cdata_out_en_10	no
181	sdhost_cdata_in_11	1	no	sdhost_cdata_out_11	sdhost_cdata_out_en_11	no
182	sdhost_cdata_in_12	1	no	sdhost_cdata_out_12	sdhost_cdata_out_en_12	no
183	sdhost_cdata_in_13	1	no	sdhost_cdata_out_13	sdhost_cdata_out_en_13	no
184	sdhost_cdata_in_14	1	no	sdhost_cdata_out_14	sdhost_cdata_out_en_14	no
185	sdhost_cdata_in_15	1	no	sdhost_cdata_out_15	sdhost_cdata_out_en_15	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
186	sdhost_cdata_in_16	1	no	sdhost_cdata_out_16	sdhost_cdata_out_en_16	no
187	sdhost_cdata_in_17	1	no	sdhost_cdata_out_17	sdhost_cdata_out_en_17	no
188	-	-	-	-	1'd1	-
189	-	-	-	-	1'd1	-
190	-	-	-	-	1'd1	-
191	-	-	-	-	1'd1	-
192	sdhost_data_strobe_1	0	no	-	1'd1	-
193	sdhost_data_strobe_2	0	no	-	1'd1	-
194	sdhost_card_detect_n_1	0	no	-	1'd1	-
195	sdhost_card_detect_n_2	0	no	-	1'd1	-
196	sdhost_card_write_prt_1	0	no	-	1'd1	-
197	sdhost_card_write_prt_2	0	no	-	1'd1	-
198	sdhost_card_int_n_1	0	no	-	1'd1	-
199	sdhost_card_int_n_2	0	no	-	1'd1	-
200	-	-	-	-	1'd1	no
201	-	-	-	-	1'd1	no
202	-	-	-	-	1'd1	no
203	-	-	-	-	1'd1	no
204	-	-	-	-	1'd1	no
205	-	-	-	-	1'd1	no
206	-	-	-	-	1'd1	no
207	-	-	-	-	1'd1	no
208	sig_in_func_208	0	no	sig_in_func208	1'd1	no
209	sig_in_func_209	0	no	sig_in_func209	1'd1	no
210	sig_in_func_210	0	no	sig_in_func210	1'd1	no
211	sig_in_func_211	0	no	sig_in_func211	1'd1	no
212	sig_in_func_212	0	no	sig_in_func212	1'd1	no

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
213	sdhost_cdata_in_20	1	no	sdhost_cdata_out_20	sdhost_cdata_out_en_20	no
214	sdhost_cdata_in_21	1	no	sdhost_cdata_out_21	sdhost_cdata_out_en_21	no
215	sdhost_cdata_in_22	1	no	sdhost_cdata_out_22	sdhost_cdata_out_en_22	no
216	sdhost_cdata_in_23	1	no	sdhost_cdata_out_23	sdhost_cdata_out_en_23	no
217	sdhost_cdata_in_24	1	no	sdhost_cdata_out_24	sdhost_cdata_out_en_24	no
218	sdhost_cdata_in_25	1	no	sdhost_cdata_out_25	sdhost_cdata_out_en_25	no
219	sdhost_cdata_in_26	1	no	sdhost_cdata_out_26	sdhost_cdata_out_en_26	no
220	sdhost_cdata_in_27	1	no	sdhost_cdata_out_27	sdhost_cdata_out_en_27	no
221	pro_alonegpio_in0	0	no	pro_alonegpio_out0	1'd1	no
222	pro_alonegpio_in1	0	no	pro_alonegpio_out1	1'd1	no
223	pro_alonegpio_in2	0	no	pro_alonegpio_out2	1'd1	no
224	pro_alonegpio_in3	0	no	pro_alonegpio_out3	1'd1	no
225	pro_alonegpio_in4	0	no	pro_alonegpio_out4	1'd1	no
226	pro_alonegpio_in5	0	no	pro_alonegpio_out5	1'd1	no
227	pro_alonegpio_in6	0	no	pro_alonegpio_out6	1'd1	no
228	pro_alonegpio_in7	0	no	pro_alonegpio_out7	1'd1	no
229	-	-	-	-	1'd1	-
230	-	-	-	-	1'd1	-
231	-	-	-	-	1'd1	-
232	-	-	-	-	1'd1	-
233	-	-	-	-	1'd1	-
234	-	-	-	-	1'd1	-
235	-	-	-	-	1'd1	-
236	-	-	-	-	1'd1	-
237	-	-	-	-	1'd1	-
238	-	-	-	-	1'd1	-
239	-	-	-	-	1'd1	-

信号索引	输入信号	默认值	信号可经由 IO MUX 直接输入	输出信号	GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号	信号可经由 IO MUX 直接输出
240	-	-	-	-	1'd1	-
241	-	-	-	-	1'd1	-
242	-	-	-	-	1'd1	-
243	-	-	-	-	1'd1	-
244	-	-	-	-	1'd1	-
245	-	-	-	-	1'd1	-
246	-	-	-	-	1'd1	-
247	-	-	-	-	1'd1	-
248	-	-	-	-	1'd1	-
249	-	-	-	-	1'd1	-
250	-	-	-	-	1'd1	-
251	usb_jtag_tdo_bridge	0	no	usb_jtag_trst	1'd1	no
252	Core1_gpio_in3	0	no	Core1_gpio_out3	1'd1	no
253	Core1_gpio_in4	0	no	Core1_gpio_out4	1'd1	no
254	Core1_gpio_in5	0	no	Core1_gpio_out5	1'd1	no
255	Core1_gpio_in6	0	no	Core1_gpio_out6	1'd1	no

2.12 IO MUX 管脚功能列表

表 2-3 列出了所有 GPIO 管脚的 IO MUX 功能。

表 2-3. IO MUX 管脚功能

GPIO	管脚	功能 0	功能 1	功能 2	功能 3	功能 4	DRV	RST	说明
0	GPIO0	GPIO0	GPIO0	-	-	-	2	3	R
1	GPIO1	GPIO1	GPIO1	-	-	-	2	1	R
2	GPIO2	GPIO2	GPIO2	-	-	-	2	1	R
3	GPIO3	GPIO3	GPIO3	-	-	-	2	1	R
4	GPIO4	GPIO4	GPIO4	-	-	-	2	0	R
5	GPIO5	GPIO5	GPIO5	-	-	-	2	0	R
6	GPIO6	GPIO6	GPIO6	-	-	-	2	0	R
7	GPIO7	GPIO7	GPIO7	-	-	-	2	0	R
8	GPIO8	GPIO8	GPIO8	-	SUBSPICS1	-	2	0	R
9	GPIO9	GPIO9	GPIO9	-	SUBSPIHD	FSPIHD	2	1	R
10	GPIO10	GPIO10	GPIO10	FSPIIO4	SUBSPICS0	FSPICS0	2	1	R
11	GPIO11	GPIO11	GPIO11	FSPIIO5	SUBSPID	FSPID	2	1	R
12	GPIO12	GPIO12	GPIO12	FSPIIO6	SUBSPICLK	FSPICLK	2	1	R
13	GPIO13	GPIO13	GPIO13	FSPIIO7	SUBSPIQ	FSPIQ	2	1	R
14	GPIO14	GPIO14	GPIO14	FSPIDQS	SUBSPIWP	FSPIWP	2	1	R
15	XTAL_32K_P	GPIO15	GPIO15	U0RTS	-	-	2	0	R
16	XTAL_32K_N	GPIO16	GPIO16	U0CTS	-	-	2	0	R
17	GPIO17	GPIO17	GPIO17	U1TXD	-	-	2	1	R
18	GPIO18	GPIO18	GPIO18	U1RXD	CLK_OUT3	-	2	1	R
19	GPIO19	GPIO19	GPIO19	U1RTS	CLK_OUT2	-	2	0	R
20	GPIO20	GPIO20	GPIO20	U1CTS	CLK_OUT1	-	2	0	R
21	GPIO21	GPIO21	GPIO21	-	-	-	2	0	R
26	SPICS1	SPICS1	GPIO26	-	-	-	2	3	-
27	SPIHD	SPIHD	GPIO27	-	-	-	3	3	-
28	SPIWP	SPIWP	GPIO28	-	-	-	3	3	-
29	SPICS0	SPICS0	GPIO29	-	-	-	3	3	-
30	SPICLK	SPICLK	GPIO30	-	-	-	3	3	-
31	SPIQ	SPIQ	GPIO31	-	-	-	3	3	-
32	SPID	SPID	GPIO32	-	-	-	3	3	-
33	GPIO33	GPIO33	GPIO33	FSPIHD	SUBSPIHD	SPIIO4	2	1	-
34	GPIO34	GPIO34	GPIO34	FSPICS0	SUBSPICS0	SPIIO5	2	1	-
35	GPIO35	GPIO35	GPIO35	FSPID	SUBSPID	SPIIO6	2	1	-
36	GPIO36	GPIO36	GPIO36	FSPICLK	SUBSPICLK	SPIIO7	2	1	-
37	GPIO37	GPIO37	GPIO37	FSPIQ	SUBSPIQ	SPIDQS	2	1	-
38	GPIO38	GPIO38	GPIO38	FSPIWP	SUBSPIWP	-	2	1	-
39	MTCK	MTCK	GPIO39	CLK_OUT3	SUBSPICS1	-	2	1*	-
40	MTDO	MTDO	GPIO40	CLK_OUT2	-	-	2	1	-
41	MTDI	MTDI	GPIO41	CLK_OUT1	-	-	2	1	-
42	MTMS	MTMS	GPIO42	-	-	-	2	1	-

GPIO	管脚	功能 0	功能 1	功能 2	功能 3	功能 4	DRV	RST	说明
43	U0TXD	U0TXD	GPIO43	CLK_OUT1	-	-	2	4	-
44	U0RXD	U0RXD	GPIO44	CLK_OUT2	-	-	2	3	-
45	GPIO45	GPIO45	GPIO45	-	-	-	2	2	-
46	GPIO46	GPIO46	GPIO46	-	-	-	2	2	-
47	SPICLK_P	SPICLK_DIFF	GPIO47	SUBSPICLK_P_DIFF	-	-	2	1	-
48	SPICLK_N	SPICLK_DIFF	GPIO48	SUBSPICLK_N_DIFF	-	-	2	1	-

驱动强度

“DRV” 一栏所示为每个管脚复位后的默认驱动强度。

- **0** - 驱动电流 = ~5 mA
- **1** - 驱动电流 = ~10 mA
- **2** - 驱动电流 = ~20 mA
- **3** - 驱动电流 = ~40 mA

复位配置

“RST” 一栏是每个管脚复位后的默认配置。

- **0** - IE = 0（输入关闭）
- **1** - IE = 1（输入使能）
- **2** - IE = 1，WPD = 1（输入使能，下拉电阻使能）
- **3** - IE = 1，WPU = 1（输入使能，上拉电阻使能）
- **4** - OE = 1，WPU = 1（输出使能，上拉电阻使能）
- **1*** - 如果 EFUSE_DIS_JTAG = 1，则 MTCK 管脚复位后浮空，即 IE = 1。如果 EFUSE_DIS_JTAG = 0，则 MTCK 复位之后连接内部上拉电阻，即 IE = 1，WPU = 1。

说明

- **R** - 管脚通过 RTC IO MUX 具有 RTC/模拟功能。

2.13 RTC IO MUX 管脚功能列表

表 2-4 列出了 RTC 管脚和对应 GPIO 管脚及 RTC 功能。

表 2-4. RTC IO MUX 管脚的 RTC 功能

RTC GPIO No.	GPIO No.	管脚	RTC 功能			
			0	1	2	3
0	0	GPIO0	RTC_GPIO0	-	-	sar_i2c_scl_0[a]
1	1	GPIO1	RTC_GPIO1	-	-	sar_i2c_sda_0[a]
2	2	GPIO2	RTC_GPIO2	-	-	sar_i2c_scl_1[a]
3	3	GPIO3	RTC_GPIO3	-	-	sar_i2c_sda_1[a]
4	4	GPIO4	RTC_GPIO4	-	-	-

见下页

表 2-4 – 接上页

RTC GPIO No.	GPIO No.	管脚	RTC 功能			
			0	1	2	3
5	5	GPIO5	RTC_GPIO5	-	-	-
6	6	GPIO6	RTC_GPIO6	-	-	-
7	7	GPIO7	RTC_GPIO7	-	-	-
8	8	GPIO8	RTC_GPIO8	-	-	-
9	9	GPIO9	RTC_GPIO9	-	-	-
10	10	GPIO10	RTC_GPIO10	-	-	-
11	11	GPIO11	RTC_GPIO11	-	-	-
12	12	GPIO12	RTC_GPIO12	-	-	-
13	13	GPIO13	RTC_GPIO13	-	-	-
14	14	GPIO14	RTC_GPIO14	-	-	-
15	15	XTAL_32K_P	RTC_GPIO15	-	-	-
16	16	XTAL_32K_N	RTC_GPIO16	-	-	-
17	17	GPIO17	RTC_GPIO17	-	-	-
18	18	GPIO18	RTC_GPIO18	-	-	-
19	19	GPIO19	RTC_GPIO19	-	-	-
20	20	GPIO20	RTC_GPIO20	-	-	-
21	21	GPIO21	RTC_GPIO21	-	-	-

[a] 有关 sar_i2c_xx 的配置信息，请参考章节 13 超低功耗协处理器 *(ULP-FSM, ULP-RISC-V) [to be added later]*：RTC I2C 控制器。

表 2-5 列出了 RTC 管脚和对应 GPIO 管脚及模拟功能。

表 2-5. RTC IO MUX 管脚模拟功能

RTC GPIO No.	GPIO No.	管脚	模拟功能	
			0	1
0	0	GPIO0	-	-
1	1	GPIO1	TOUCH1	ADC1_CH0
2	2	GPIO2	TOUCH2	ADC1_CH1
3	3	GPIO3	TOUCH3	ADC1_CH2
4	4	GPIO4	TOUCH4	ADC1_CH3
5	5	GPIO5	TOUCH5	ADC1_CH4
6	6	GPIO6	TOUCH6	ADC1_CH5
7	7	GPIO7	TOUCH7	ADC1_CH6
8	8	GPIO8	TOUCH8	ADC1_CH7
9	9	GPIO9	TOUCH9	ADC1_CH8
10	10	GPIO10	TOUCH10	ADC1_CH9
11	11	GPIO11	TOUCH11	ADC2_CH0
12	12	GPIO12	TOUCH12	ADC2_CH1
13	13	GPIO13	TOUCH13	ADC2_CH2
14	14	GPIO14	TOUCH14	ADC2_CH3
15	15	XTAL_32K_P	XTAL_32K_P	ADC2_CH4

RTC GPIO No.	GPIO No.	管脚	模拟功能	
			0	1
16	16	XTAL_32K_N	XTAL_32K_N	ADC2_CH5
17	17	GPIO17	-	ADC2_CH6
18	18	GPIO18	-	ADC2_CH7
19	19	GPIO19	USB_D-	ADC2_CH8
20	20	GPIO20	USB_D+	ADC2_CH9
21	21	GPIO21	-	-

2.14 寄存器列表

2.14.1 GPIO 交换矩阵寄存器列表

本小节的所有地址均为相对于 GPIO 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

名称	描述	地址	访问
GPIO 配置寄存器			
GPIO_BT_SELECT_REG	GPIO 位选择寄存器	0x0000	读/写
GPIO_OUT_REG	GPIO0 ~ 31 输出寄存器	0x0004	读/写
GPIO_OUT_W1TS_REG	GPIO0 ~ 31 输出置位寄存器	0x0008	只写
GPIO_OUT_W1TC_REG	GPIO0 ~ 31 输出清零寄存器	0x000C	只写
GPIO_OUT1_REG	GPIO32 ~ 48 输出寄存器	0x0010	读/写
GPIO_OUT1_W1TS_REG	GPIO32 ~ 48 输出置位寄存器	0x0014	只写
GPIO_OUT1_W1TC_REG	GPIO32 ~ 48 输出清零寄存器	0x0018	只写
GPIO_SDIO_SELECT_REG	GPIO SDIO 选择寄存器	0x001C	读/写
GPIO_ENABLE_REG	GPIO0 ~ 31 输出使能寄存器	0x0020	读/写
GPIO_ENABLE_W1TS_REG	GPIO0 ~ 31 输出使能置位寄存器	0x0024	只写
GPIO_ENABLE_W1TC_REG	GPIO0 ~ 31 输出使能清零寄存器	0x0028	只写
GPIO_ENABLE1_REG	GPIO32 ~ 48 输出使能寄存器	0x002C	读/写
GPIO_ENABLE1_W1TS_REG	GPIO32 ~ 48 输出使能置位寄存器	0x0030	只写
GPIO_ENABLE1_W1TC_REG	GPIO32 ~ 48 输出使能清零寄存器	0x0034	只写
GPIO_STRAP_REG	Strapping 管脚寄存器	0x0038	只读
GPIO_IN_REG	GPIO0 ~ 31 输入寄存器	0x003C	只读
GPIO_IN1_REG	GPIO32 ~ 48 输入寄存器	0x0040	只读
GPIO_PIN0_REG	配置 GPIO pin 0	0x0074	读/写
GPIO_PIN1_REG	配置 GPIO pin 1	0x0078	读/写
GPIO_PIN2_REG	配置 GPIO pin 2	0x007C	读/写
...	...	...	...
GPIO_PIN46_REG	配置 GPIO pin 46	0x012C	读/写
GPIO_PIN47_REG	配置 GPIO pin 47	0x0130	读/写
GPIO_PIN48_REG	配置 GPIO pin 48	0x0134	读/写
GPIO_FUNC0_IN_SEL_CFG_REG	外设信号 0 的输入选择寄存器	0x0154	读/写
GPIO_FUNC1_IN_SEL_CFG_REG	外设信号 1 的输入选择寄存器	0x0158	读/写
GPIO_FUNC2_IN_SEL_CFG_REG	外设信号 2 的输入选择寄存器	0x015C	读/写

名称	描述	地址	访问
...	...	...	...
GPIO_FUNC253_IN_SEL_CFG_REG	外设信号 253 的输入选择寄存器	0x0548	读/写
GPIO_FUNC254_IN_SEL_CFG_REG	外设信号 254 的输入选择寄存器	0x054C	读/写
GPIO_FUNC255_IN_SEL_CFG_REG	外设信号 255 的输入选择寄存器	0x0550	读/写
GPIO_FUNC0_OUT_SEL_CFG_REG	GPIO0 的外设输出信号选择寄存器	0x0554	读/写
GPIO_FUNC1_OUT_SEL_CFG_REG	GPIO1 的外设输出信号选择寄存器	0x0558	读/写
GPIO_FUNC2_OUT_SEL_CFG_REG	GPIO2 的外设输出信号选择寄存器	0x055C	读/写
...	...	...	...
GPIO_FUNC47_OUT_SEL_CFG_REG	GPIO47 的外设输出信号选择寄存器	0x0610	读/写
GPIO_FUNC48_OUT_SEL_CFG_REG	GPIO48 的外设输出信号选择寄存器	0x0614	读/写
GPIO_CLOCK_GATE_REG	GPIO 时钟门控寄存器	0x062C	读/写
中断状态寄存器			
GPIO_STATUS_REG	GPIO0 ~ 31 中断状态寄存器	0x0044	读/写
GPIO_STATUS1_REG	GPIO32 ~ 48 中断状态寄存器	0x0050	读/写
GPIO_PCPU_INT_REG	GPIO0 ~ 31 PRO_CPU 中断状态寄存器	0x005C	只读
GPIO_PCPU_NMI_INT_REG	GPIO0 ~ 31 PRO_CPU 非屏蔽中断状态寄存器	0x0060	只读
GPIO_PCPU_INT1_REG	GPIO32 ~ 48 PRO_CPU 中断状态寄存器	0x0068	只读
GPIO_PCPU_NMI_INT1_REG	GPIO32 ~ 48 PRO_CPU 非屏蔽状态寄存器	0x006C	只读
中断配置寄存器			
GPIO_STATUS_W1TS_REG	GPIO0 ~ 31 中断状态置位寄存器	0x0048	只写
GPIO_STATUS_W1TC_REG	GPIO0 ~ 31 中断状态清零寄存器	0x004C	只写
GPIO_STATUS1_W1TS_REG	GPIO32 ~ 48 中断状态置位寄存器	0x0054	只写
GPIO_STATUS1_W1TC_REG	GPIO32 ~ 48 中断状态清零寄存器	0x0058	只写
GPIO 中断源寄存器			
GPIO_STATUS_NEXT_REG	GPIO0 ~ 31 中断源寄存器	0x014C	只读
GPIO_STATUS_NEXT1_REG	GPIO32 ~ 48 中断源寄存器	0x0150	只读
版本寄存器			
GPIO_DATE_REG	版本控制寄存器	0x06FC	读/写

2.14.2 IO MUX 寄存器列表

本小节的所有地址均为相对于 IO MUX 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

名称	描述	地址	访问
IO_MUX_PIN_CTRL	时钟输出配置寄存器	0x0000	读/写
IO_MUX_GPIO0_REG	GPIO0 配置寄存器	0x0004	读/写
IO_MUX_GPIO1_REG	GPIO1 配置寄存器	0x0008	读/写
IO_MUX_GPIO2_REG	GPIO2 配置寄存器	0x000C	读/写
IO_MUX_GPIO3_REG	GPIO3 配置寄存器	0x0010	读/写
IO_MUX_GPIO4_REG	GPIO4 配置寄存器	0x0014	读/写
IO_MUX_GPIO5_REG	GPIO5 配置寄存器	0x0018	读/写
IO_MUX_GPIO6_REG	GPIO6 配置寄存器	0x001C	读/写
IO_MUX_GPIO7_REG	GPIO7 配置寄存器	0x0020	读/写

名称	描述	地址	访问
IO_MUX_GPIO8_REG	GPIO8 配置寄存器	0x0024	读/写
IO_MUX_GPIO9_REG	GPIO9 配置寄存器	0x0028	读/写
IO_MUX_GPIO10_REG	GPIO10 配置寄存器	0x002C	读/写
IO_MUX_GPIO11_REG	GPIO11 配置寄存器	0x0030	读/写
IO_MUX_GPIO12_REG	GPIO12 配置寄存器	0x0034	读/写
IO_MUX_GPIO13_REG	GPIO13 配置寄存器	0x0038	读/写
IO_MUX_GPIO14_REG	GPIO14 配置寄存器	0x003C	读/写
IO_MUX_GPIO15_REG	XTAL_32K_P 配置寄存器	0x0040	读/写
IO_MUX_GPIO16_REG	XTAL_32K_N 配置寄存器	0x0044	读/写
IO_MUX_GPIO17_REG	GPIO17 配置寄存器	0x0048	读/写
IO_MUX_GPIO18_REG	GPIO18 配置寄存器	0x004C	读/写
IO_MUX_GPIO19_REG	GPIO19 配置寄存器	0x0050	读/写
IO_MUX_GPIO20_REG	GPIO20 配置寄存器	0x0054	读/写
IO_MUX_GPIO21_REG	GPIO21 配置寄存器	0x0058	读/写
IO_MUX_GPIO26_REG	SPICS1 配置寄存器	0x006C	读/写
IO_MUX_GPIO27_REG	SPIHD 配置寄存器	0x0070	读/写
IO_MUX_GPIO28_REG	SPIWP 配置寄存器	0x0074	读/写
IO_MUX_GPIO29_REG	SPICS0 配置寄存器	0x0078	读/写
IO_MUX_GPIO30_REG	SPICLK 配置寄存器	0x007C	读/写
IO_MUX_GPIO31_REG	SPIQ 配置寄存器	0x0080	读/写
IO_MUX_GPIO32_REG	SPID 配置寄存器	0x0084	读/写
IO_MUX_GPIO33_REG	GPIO33 配置寄存器	0x0088	读/写
IO_MUX_GPIO34_REG	GPIO34 配置寄存器	0x008C	读/写
IO_MUX_GPIO35_REG	GPIO35 配置寄存器	0x0090	读/写
IO_MUX_GPIO36_REG	GPIO36 配置寄存器	0x0094	读/写
IO_MUX_GPIO37_REG	GPIO37 配置寄存器	0x0098	读/写
IO_MUX_GPIO38_REG	GPIO38 配置寄存器	0x009C	读/写
IO_MUX_GPIO39_REG	MTCK 配置寄存器	0x00A0	读/写
IO_MUX_GPIO40_REG	MTDO 配置寄存器	0x00A4	读/写
IO_MUX_GPIO41_REG	MTDI 配置寄存器	0x00A8	读/写
IO_MUX_GPIO42_REG	MTMS 配置寄存器	0x00AC	读/写
IO_MUX_GPIO43_REG	U0TXD 配置寄存器	0x00B0	读/写
IO_MUX_GPIO44_REG	U0RXD 配置寄存器	0x00B4	读/写
IO_MUX_GPIO45_REG	GPIO45 配置寄存器	0x00B8	读/写
IO_MUX_GPIO46_REG	GPIO46 配置寄存器	0x00BC	读/写
IO_MUX_GPIO47_REG	GPIO47 配置寄存器	0x00C0	读/写
IO_MUX_GPIO48_REG	GPIO48 配置寄存器	0x00C4	读/写

2.14.3 SDM 寄存器列表

本小节的所有地址均为相对于 GPIO 基地址 + 0x0F00 的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

名称	描述	地址	权限
配置寄存器			
GPIO_SIGMADELTA0_REG	SDM0 占空比配置寄存器	0x0000	读/写
GPIO_SIGMADELTA1_REG	SDM1 占空比配置寄存器	0x0004	读/写
GPIO_SIGMADELTA2_REG	SDM2 占空比配置寄存器	0x0008	读/写
GPIO_SIGMADELTA3_REG	SDM3 占空比配置寄存器	0x000C	读/写
GPIO_SIGMADELTA4_REG	SDM4 占空比配置寄存器	0x0010	读/写
GPIO_SIGMADELTA5_REG	SDM5 占空比配置寄存器	0x0014	读/写
GPIO_SIGMADELTA6_REG	SDM6 占空比配置寄存器	0x0018	读/写
GPIO_SIGMADELTA7_REG	SDM7 占空比配置寄存器	0x001C	读/写
GPIO_SIGMADELTA_CG_REG	时钟门控配置寄存器	0x0020	读/写
GPIO_SIGMADELTA_MISC_REG	MISC 寄存器	0x0024	读/写
GPIO_SIGMADELTA_VERSION_REG	版本控制寄存器	0x0028	读/写

2.14.4 RTC IO MUX 寄存器列表

本小节的所有地址均为相对于低功耗管理模块基地址 + 0x0400 的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

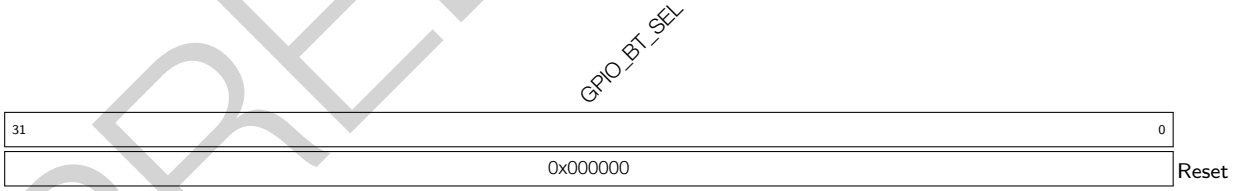
名称	描述	地址	权限
GPIO 配置/数据寄存器			
RTC_GPIO_OUT_REG	RTC GPIO 输出寄存器	0x0000	读/写
RTC_GPIO_OUT_W1TS_REG	RTC GPIO 输出置位寄存器	0x0004	只写
RTC_GPIO_OUT_W1TC_REG	RTC GPIO 输出置位清零寄存器	0x0008	只写
RTC_GPIO_ENABLE_REG	RTC GPIO 输出使能寄存器	0x000C	读/写
RTC_GPIO_ENABLE_W1TS_REG	RTC GPIO 输出使能置位寄存器	0x0010	只写
RTC_GPIO_ENABLE_W1TC_REG	RTC GPIO 输出使能清零寄存器	0x0014	只写
RTC_GPIO_STATUS_REG	RTC GPIO 中断状态寄存器	0x0018	读/写
RTC_GPIO_STATUS_W1TS_REG	RTC GPIO 中断状态置位寄存器	0x001C	只写
RTC_GPIO_STATUS_W1TC_REG	RTC GPIO 中断状态清零寄存器	0x0020	只写
RTC_GPIO_IN_REG	RTC GPIO 输入寄存器	0x0024	只读
RTC_GPIO_PIN0_REG	Pin0 RTC 配置	0x0028	读/写
RTC_GPIO_PIN1_REG	Pin1 RTC 配置	0x002C	读/写
RTC_GPIO_PIN2_REG	Pin2 RTC 配置	0x0030	读/写
RTC_GPIO_PIN3_REG	Pin3 RTC 配置	0x0034	读/写
...	...	...	...
RTC_GPIO_PIN19_REG	Pin19 RTC 配置	0x0074	读/写
RTC_GPIO_PIN20_REG	Pin20 RTC 配置	0x0078	读/写
RTC_GPIO_PIN21_REG	Pin21 RTC 配置	0x007C	读/写
GPIO RTC 功能配置寄存器			
RTC_IO_TOUCH_PAD0_REG	Touch pad 0 配置寄存器	0x0084	读/写
RTC_IO_TOUCH_PAD1_REG	Touch pad 1 配置寄存器	0x0088	读/写
RTC_IO_TOUCH_PAD2_REG	Touch pad 2 配置寄存器	0x008C	读/写
RTC_IO_TOUCH_PAD3_REG	Touch pad 3 配置寄存器	0x0090	读/写
RTC_IO_TOUCH_PAD4_REG	Touch pad 4 配置寄存器	0x0094	读/写

名称	描述	地址	权限
RTC_IO_TOUCH_PAD5_REG	Touch pad 5 配置寄存器	0x0098	读/写
RTC_IO_TOUCH_PAD6_REG	Touch pad 6 配置寄存器	0x009C	读/写
RTC_IO_TOUCH_PAD7_REG	Touch pad 7 配置寄存器	0x00A0	读/写
RTC_IO_TOUCH_PAD8_REG	Touch pad 8 配置寄存器	0x00A4	读/写
RTC_IO_TOUCH_PAD9_REG	Touch pad 9 配置寄存器	0x00A8	读/写
RTC_IO_TOUCH_PAD10_REG	Touch pad 10 配置寄存器	0x00AC	读/写
RTC_IO_TOUCH_PAD11_REG	Touch pad 11 配置寄存器	0x00B0	读/写
RTC_IO_TOUCH_PAD12_REG	Touch pad 12 配置寄存器	0x00B4	读/写
RTC_IO_TOUCH_PAD13_REG	Touch pad 13 配置寄存器	0x00B8	读/写
RTC_IO_TOUCH_PAD14_REG	Touch pad 14 配置寄存器	0x00BC	读/写
RTC_IO_XTAL_32P_PAD_REG	32KHz crystal P-pad 配置寄存器	0x00C0	读/写
RTC_IO_XTAL_32N_PAD_REG	32KHz crystal N-pad 配置寄存器	0x00C4	读/写
RTC_IO_RTC_PAD17_REG	管脚 17 的配置寄存器	0x00C8	读/写
RTC_IO_RTC_PAD18_REG	管脚 17 的配置寄存器	0x00CC	读/写
RTC_IO_RTC_PAD19_REG	管脚 19 的配置寄存器	0x00D0	读/写
RTC_IO_RTC_PAD20_REG	管脚 20 的配置寄存器	0x00D4	读/写
RTC_IO_RTC_PAD21_REG	管脚 21 的配置寄存器	0x00D8	读/写
RTC_IO_XTL_EXT_CTR_REG	晶振断电 GPIO 使能源	0x00E0	读/写
RTC_IO_SAR_I2C_IO_REG	RTC I2C Pad 选择寄存器	0x00E4	读/写
版本寄存器			
RTC_IO_DATE_REG	版本控制寄存器	0x01FC	读/写

2.15 寄存器

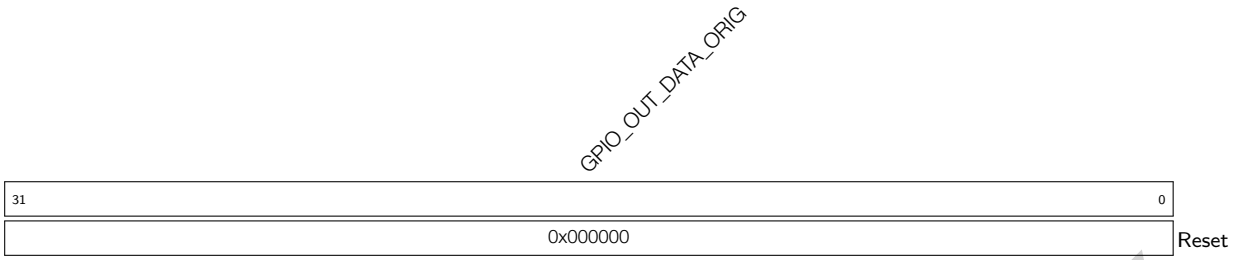
2.15.1 GPIO 交换矩阵寄存器

Register 2.1. GPIO_BT_SELECT_REG (0x0000)



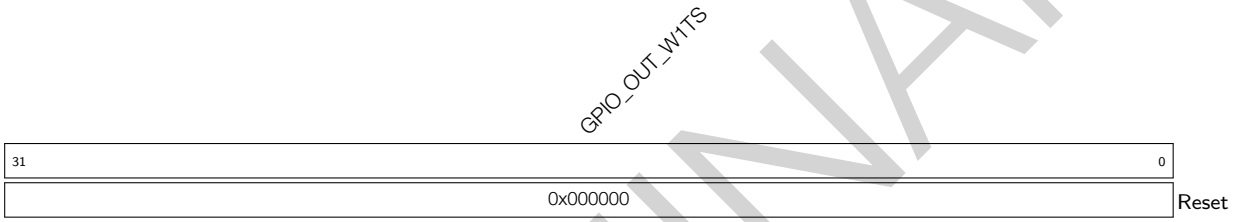
GPIO_BT_SEL 保留（读/写）

Register 2.2. GPIO_OUT_REG (0x0004)



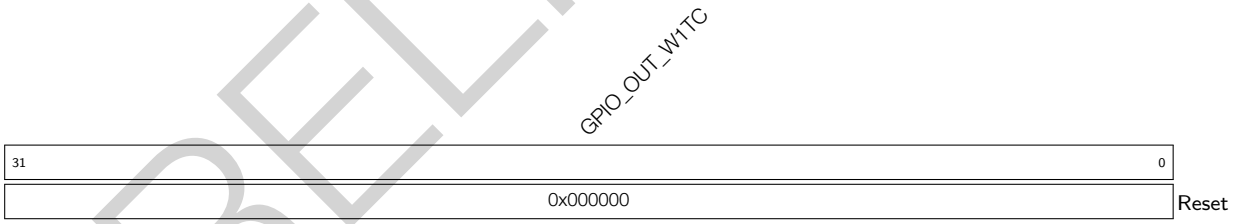
GPIO_OUT_DATA_ORIG 简单 GPIO 输出模式下，GPIO0 ~ 21 和 GPIO26 ~31 的输出值。bit0 ~ bit21 的值分别对应 GPIO0 ~ GPIO21 的输出值；bit26 ~ bit31 的值分别对应 GPIO26 ~ GPIO31 的输出值。bit22 ~ bit25 无效。（读/写）

Register 2.3. GPIO_OUT_W1TS_REG (0x0008)



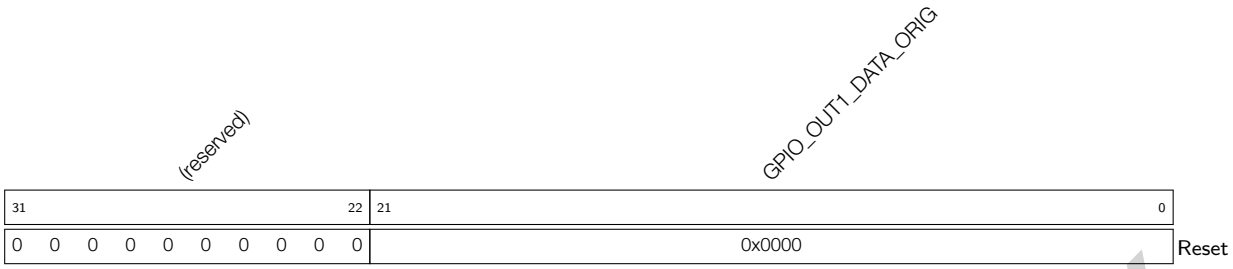
GPIO_OUT_W1TS GPIO0 ~ 31 输出置位寄存器。每一位置 1，GPIO_OUT_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_OUT_REG。（只写）

Register 2.4. GPIO_OUT_W1TC_REG (0x000C)



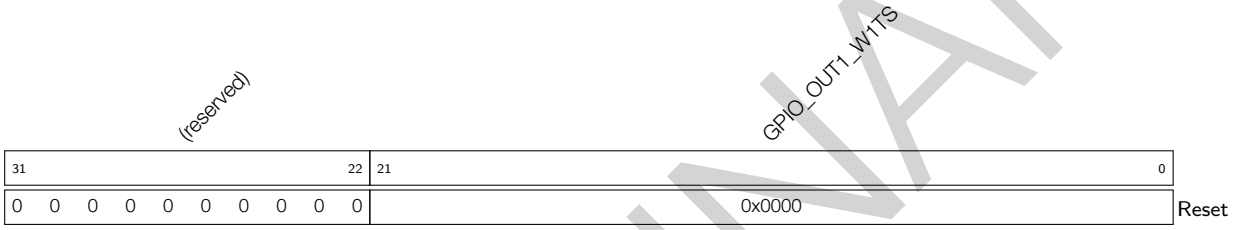
GPIO_OUT_W1TC GPIO0 ~ 31 输出清零寄存器。每一位置 1，则 GPIO_OUT_REG 中的相应位会清零。注：推荐使用此寄存器来清零 GPIO_OUT_REG。（只写）

Register 2.5. GPIO_OUT1_REG (0x0010)



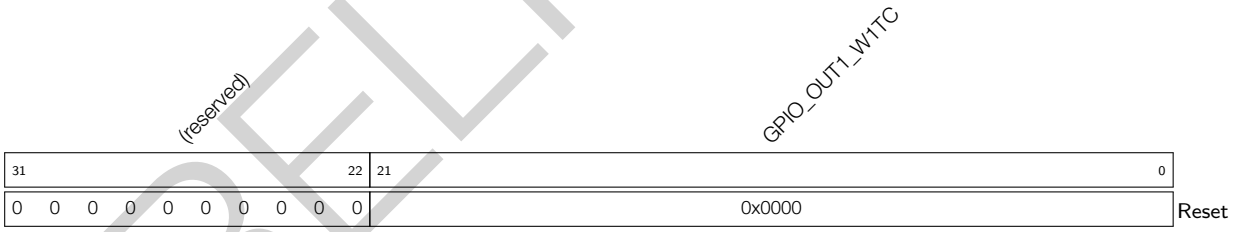
GPIO_OUT1_DATA_ORIG 简单 GPIO 输出模式下，GPIO32 ~ 48 的输出值。bit0 ~ bit16 的值分别对应 GPIO32 ~ GPIO48 的输出值。bit17 ~ bit21 无效。(读/写)

Register 2.6. GPIO_OUT1_W1TS_REG (0x0014)



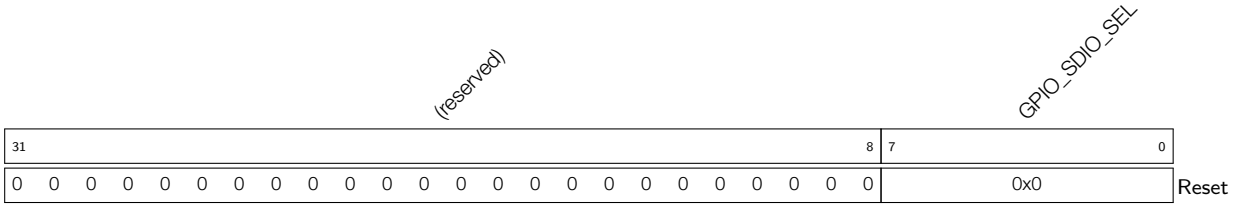
GPIO_OUT1_W1TS GPIO32 ~ 48 输出置位寄存器。每一位置 1，则 GPIO_OUT1_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_OUT1_REG。(只写)

Register 2.7. GPIO_OUT1_W1TC_REG (0x0018)



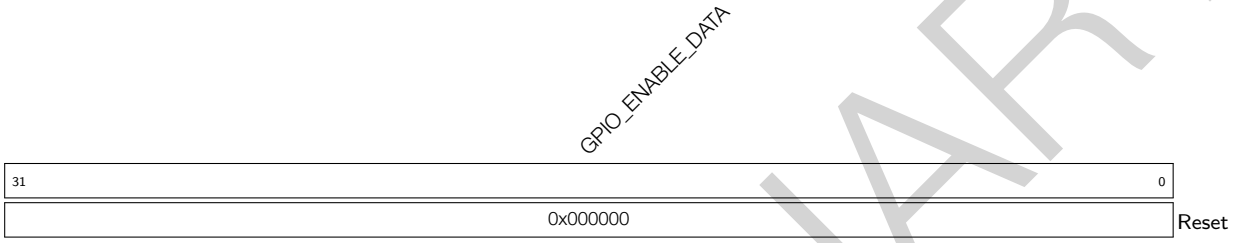
GPIO_OUT1_W1TC GPIO32 ~ 48 输出清零寄存器。每一位置 1，则 GPIO_OUT1_REG 中的相应位会清零。注：推荐使用此寄存器来清零 GPIO_OUT1_REG。(只写)

Register 2.8. GPIO_SDIO_SELECT_REG (0x001C)



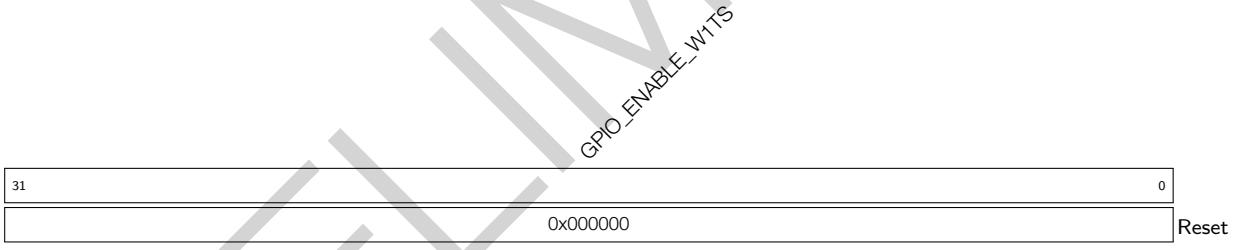
GPIO_SDIO_SEL 保留（读/写）

Register 2.9. GPIO_ENABLE_REG (0x0020)



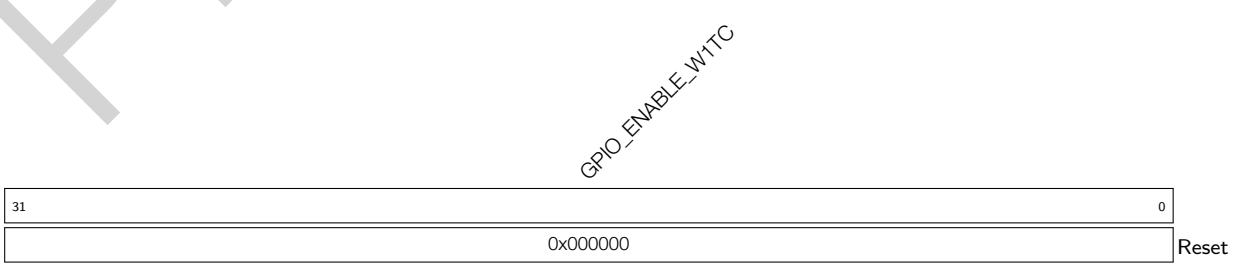
GPIO_ENABLE_DATA GPIO0 ~ 31 输出使能寄存器。（读/写）

Register 2.10. GPIO_ENABLE_W1TS_REG (0x0024)



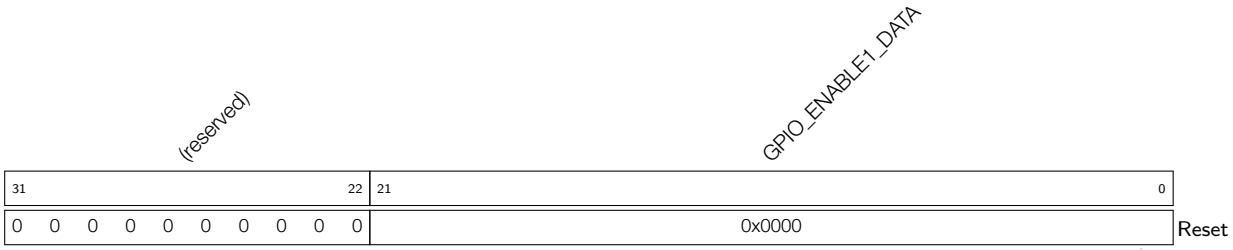
GPIO_ENABLE_W1TS GPIO0 ~ 31 输出使能置位寄存器。每一位置 1，则 GPIO_ENABLE_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_ENABLE_REG。（只写）

Register 2.11. GPIO_ENABLE_W1TC_REG (0x0028)



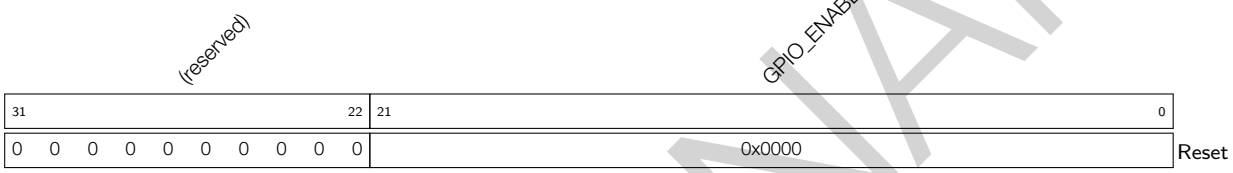
GPIO_ENABLE_W1TC GPIO0 ~ 31 输出使能清零寄存器。每一位置 1，则 GPIO_ENABLE_REG 中的相应位会清零。注：推荐使用此寄存器清零 GPIO_ENABLE_REG。（只写）

Register 2.12. GPIO_ENABLE1_REG (0x002C)



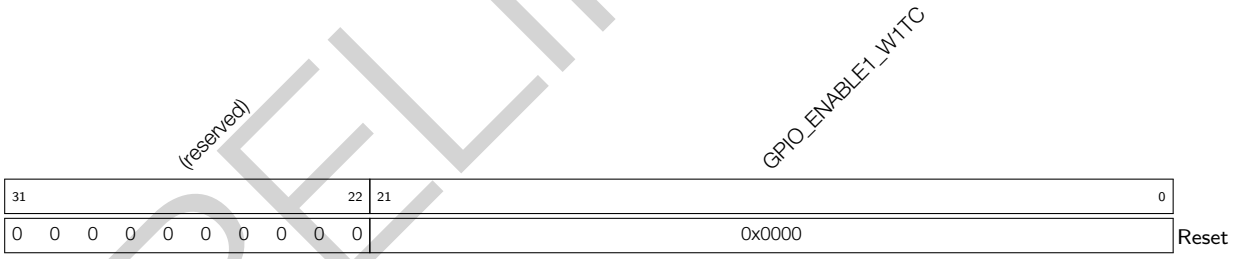
GPIO_ENABLE1_DATA GPIO32 ~ 48 输出使能寄存器。(读/写)

Register 2.13. GPIO_ENABLE1_W1TS_REG (0x0030)



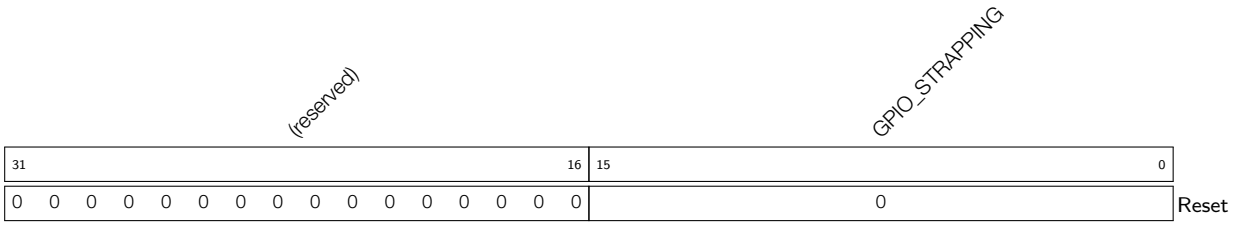
GPIO_ENABLE1_W1TS GPIO32 ~ 48 输出使能置位寄存器。每一位置 1，则 GPIO_ENABLE1_REG 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_ENABLE1_REG。(只写)

Register 2.14. GPIO_ENABLE1_W1TC_REG (0x0034)



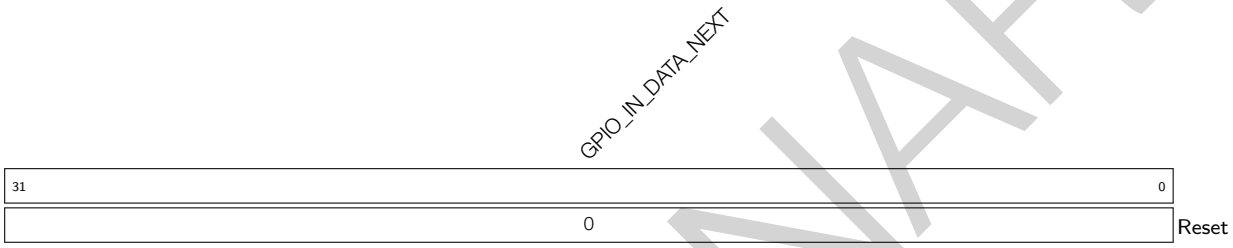
GPIO_ENABLE1_W1TC GPIO32 ~ 48 输出使能清零寄存器。每一位置 1，则 GPIO_ENABLE1_REG 中的相应位会清零。注：推荐使用此寄存器清零 GPIO_ENABLE1_REG。(只写)

Register 2.15. GPIO_STRAP_REG (0x0038)



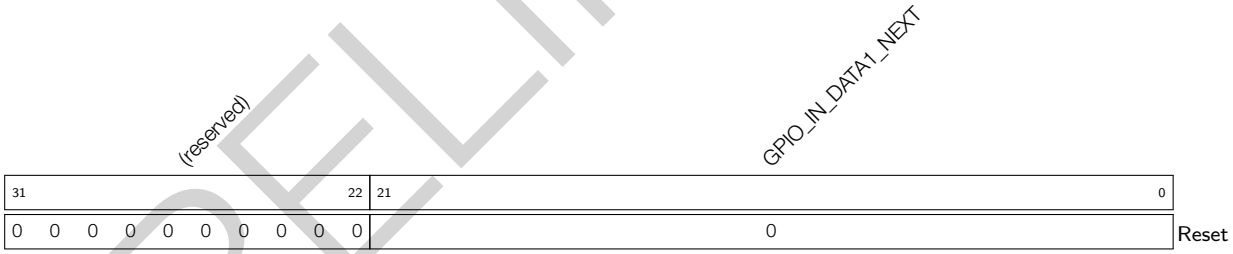
GPIO_STRAPPING GPIO Strapping 值：bit5 ~ bit2 分别对应 GPIO3、GPIO45、GPIO0 和 GPIO46。(只读)

Register 2.16. GPIO_IN_REG (0x003C)



GPIO_IN_DATA_NEXT GPIO0 ~ 31 输入值。每一位代表一个管脚的片外输入值。比如片外引脚为高电平，则此位应为 1；片外引脚为低电平，此位应为 0。(只读)

Register 2.17. GPIO_IN1_REG (0x0040)



GPIO_IN_DATA1_NEXT GPIO32 ~ 48 输入值。每一位代表一个管脚的片外输入值。(只读)

Register 2.18. GPIO_PIN*n*_REG (*n*: 0-48) (0x0074+0x4n*)**

GPIO_PIN*n*_SYNC2_BYPASS 使能 GPIO 输入信号第二拍为 APB 时钟上升沿或下降沿同步。0：关闭同步；1：下降沿同步；2 或 3：上升沿同步。(读/写)

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

GPIO_PIN*n*_SYNC1_BYPASS 使能 GPIO 输入信号第一拍为 APB 时钟上升沿或下降沿同步。0：关闭同步；1：下降沿同步；2 或 3：上升沿同步。(读/写)

GPIO_PIN*n*_INT_TYPE 中断类型选择。(读/写)

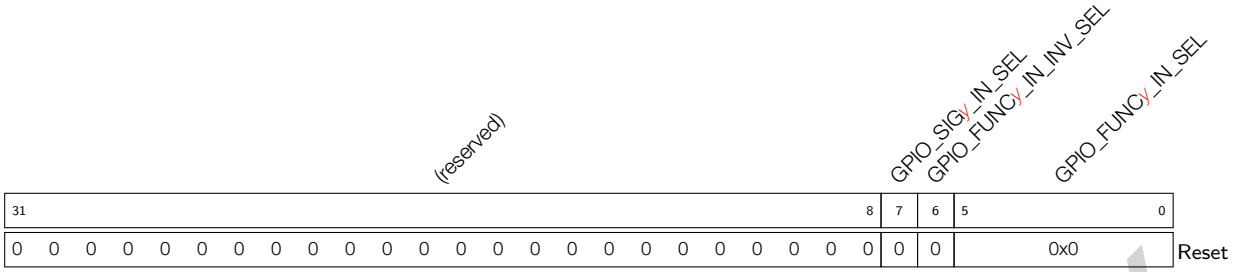
0：禁用 GPIO 中断
1：上升沿触发
2：下降沿触发
3：任一沿触发
4：低电平触发
5：高电平触发

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

GPIO_PIN*n*_CONFIG 保留。(读/写)

GPIO_PIN*n*_INT_ENA 中断使能位。bit13：使能 CPU 中断；bit14：使能 CPU 非屏蔽中断。(读/写)

Register 2.19. GPIO_FUNCy_IN_SEL_CFG_REG (y: 0-255) (0x0154+0x4*y)

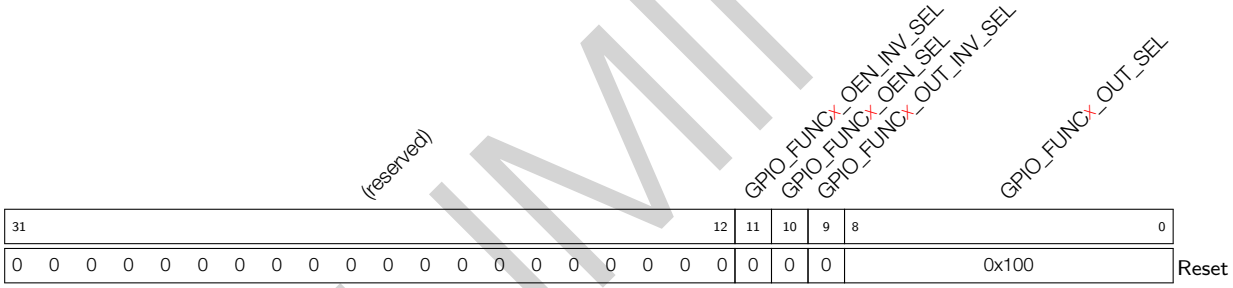


GPIO_FUNCy_IN_SEL 外设输入信号 Y 的选择控制位。此位选择 1 个 GPIO 交换矩阵输入管脚与信号连接；或者选择 0x38，则输入信号恒为高电平；或者选择 0x3C，则输入信号恒为低电平。(读/写)

GPIO_FUNCy_IN_INV_SEL 反转输入值。1：反转；0：不反转。(读/写)

GPIO_SIGy_IN_SEL 旁路 GPIO 交换矩阵。1：通过 GPIO 交换矩阵；0：直接通过 IO MUX 连接信号与外设。(读/写)

Register 2.20. GPIO_FUNCx_OUT_SEL_CFG_REG (x: 0-48) (0x0554+0x4*x)



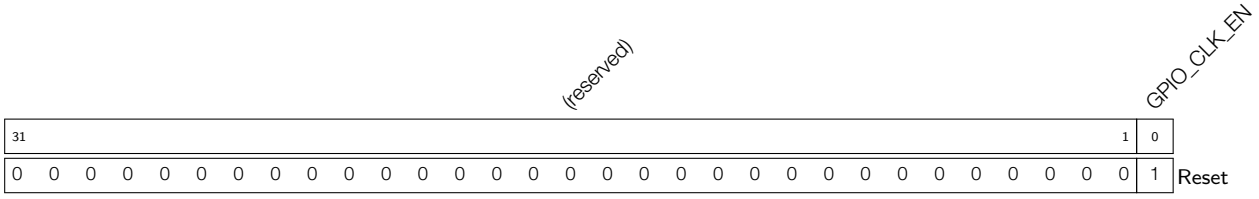
GPIO_FUNCx_OUT_SEL GPIO 管脚 X 的输出信号选择控制位。值为 y (0<=y<256) 连接外设输出 y 与 GPIO 输出 x。值为 256 选择 GPIO_OUT_REG/GPIO_OUT1_REG[x] 和 GPIO_ENABLE_REG/GPIO_ENABLE1_REG [x] 作为输出值和输出使能。(读/写)

GPIO_FUNCx_OUT_INV_SEL 0：不反转输出值；1：反转输出值。(读/写)

GPIO_FUNCx_OEN_SEL 0：采用外设的输出使能信号；1：强制使用 GPIO_ENABLE_REG[x] 用作输出使能信号。(读/写)

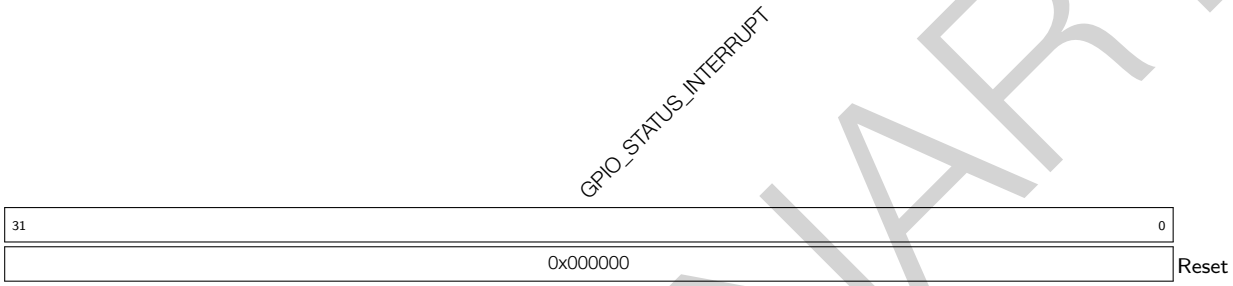
GPIO_FUNCx_OEN_INV_SEL 0：不反转输出使能信号；1：反转输出使能信号。(读/写)

Register 2.21. GPIO_CLOCK_GATE_REG (0x062C)



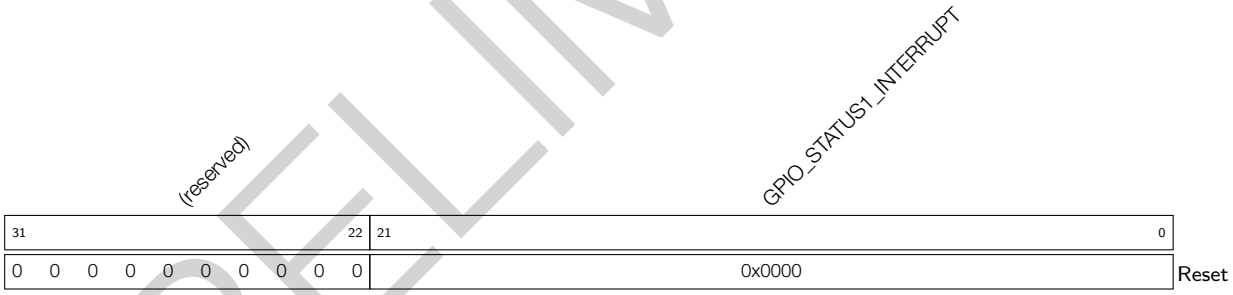
GPIO_CLK_EN 时钟门控使能。此位置 1，则时钟自由运转。(读/写)

Register 2.22. GPIO_STATUS_REG (0x0044)



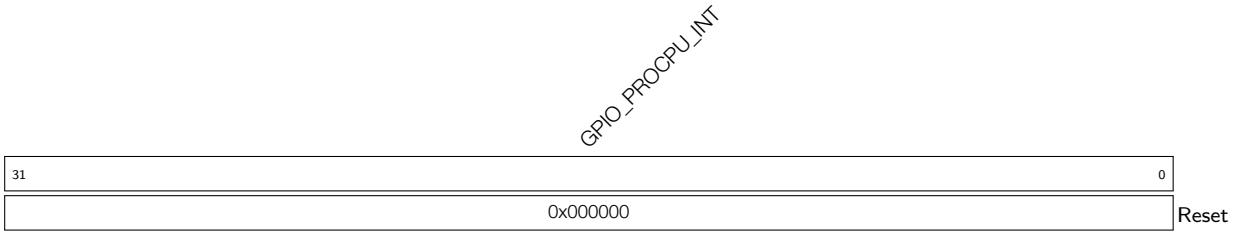
GPIO_STATUS_INTERRUPT GPIO0 ~ 31 中断状态寄存器。(读/写)

Register 2.23. GPIO_STATUS1_REG (0x0050)



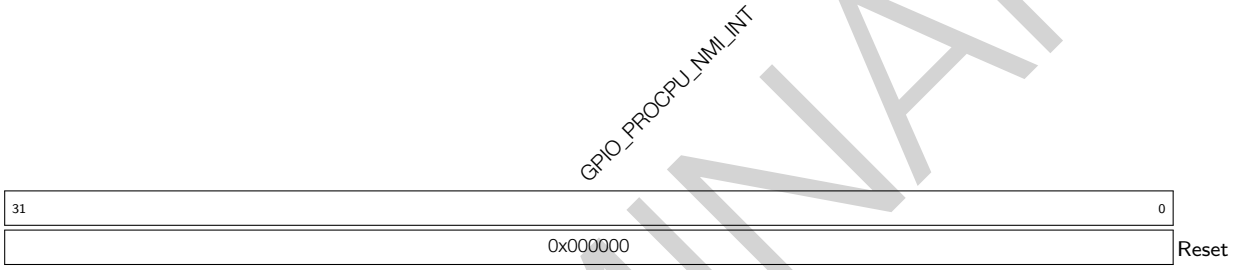
GPIO_STATUS1_INTERRUPT GPIO32 ~ 48 中断状态寄存器。(读/写)

Register 2.24. GPIO_PCPU_INT_REG (0x005C)



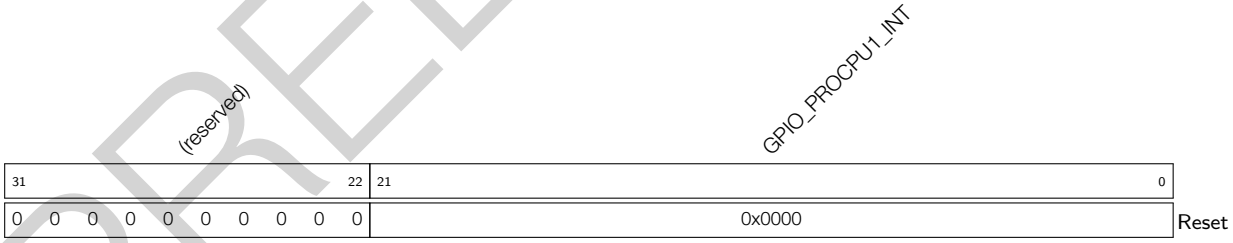
GPIO_PROCPU_INT GPIO0 ~ 31 PRO_CPU 中断状态。如果 GPIO_PIN*n*_REG 中 bit13 高电平有效，即使能 CPU 中断，则此寄存器所示的中断状态应与 GPIO_STATUS_REG 中相应 bit 的中断状态一致。(只读)

Register 2.25. GPIO_PCPU_NMI_INT_REG (0x0060)



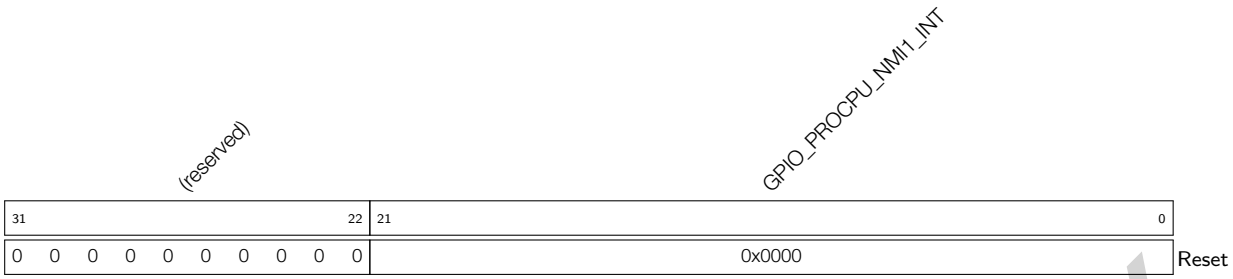
GPIO_PROCPU_NMI_INT GPIO0 ~ 31 PRO_CPU 非屏蔽中断状态寄存器。如果 GPIO_PIN*n*_REG 中 bit14 高电平有效，即使能 CPU 非屏蔽中断，则此寄存器所示的中断状态应与 GPIO_STATUS_REG 中相应 bit 的中断状态一致。(只读)

Register 2.26. GPIO_PCPU_INT1_REG (0x0068)



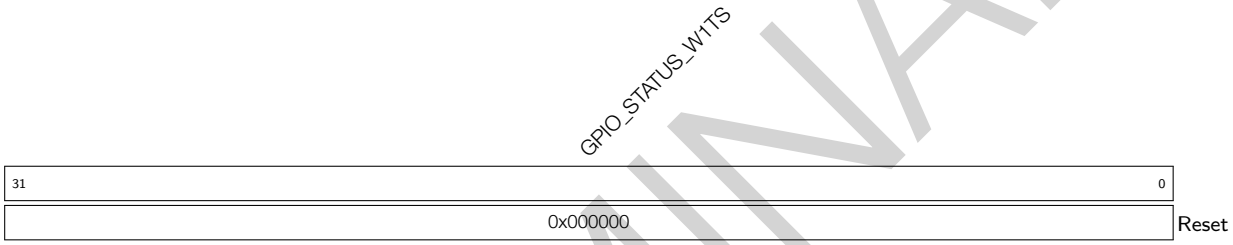
GPIO_PROCPU1_INT GPIO32 ~ 48 PRO_CPU 中断状态寄存器。如果 GPIO_PIN*n*_REG 中 bit13 高电平有效，即使能 CPU 中断，则此寄存器所示的中断状态应与 GPIO_STATUS1_REG 中相应 bit 的中断状态一致。(只读)

Register 2.27. GPIO_PCPU_NMI_INT1_REG (0x006C)



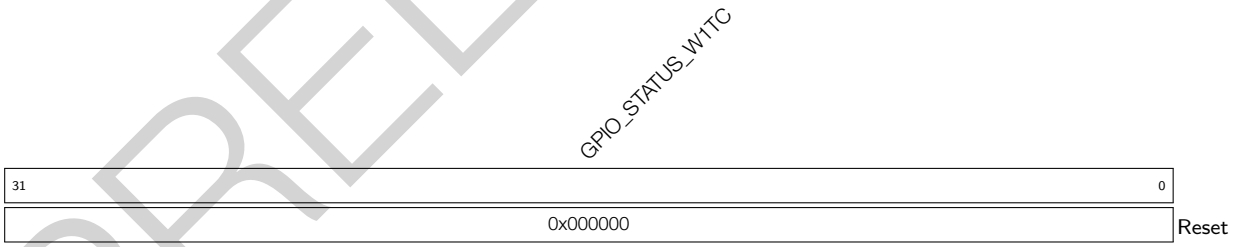
GPIO_PROCPU_NMI1_INT GPIO32 ~ 48 PRO_CPU 非屏蔽中断状态寄存器。如果 GPIO_PIN*n*_REG 中 bit14 高电平有效，即使能 CPU 非屏蔽中断，则此寄存器所示的中断状态应与 GPIO_STATUS1_REG 中相应 bit 的中断状态一致。(只读)

Register 2.28. GPIO_STATUS_W1TS_REG (0x0048)



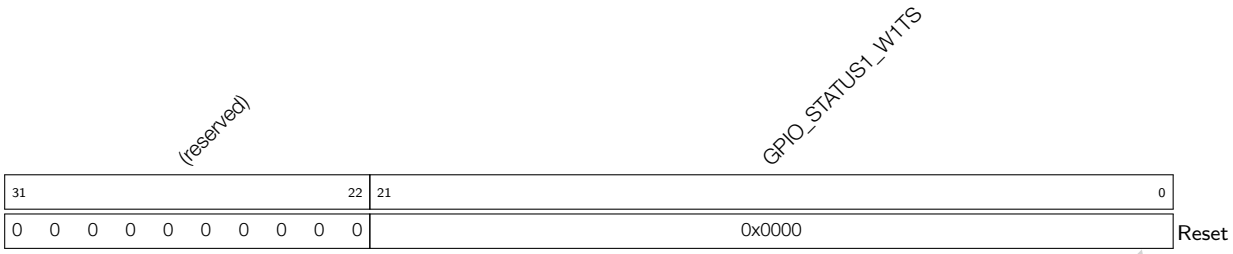
GPIO_STATUS_W1TS GPIO0 ~ 31 中断状态置位寄存器。每位置 1，则 GPIO_STATUS_INTERRUPT 中的相应位也置 1。注：推荐使用此寄存器来置位 GPIO_STATUS_INTERRUPT。(只写)

Register 2.29. GPIO_STATUS_W1TC_REG (0x004C)



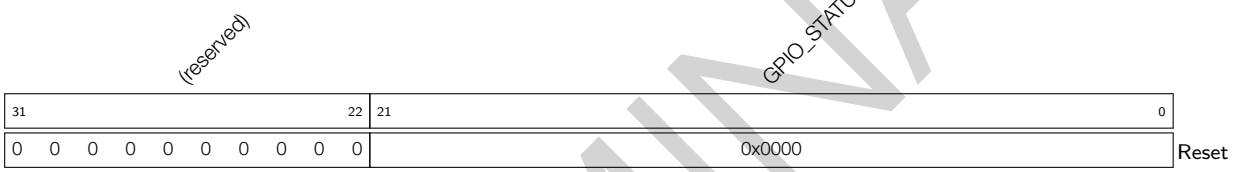
GPIO_STATUS_W1TC GPIO0 ~ 31 中断状态清除寄存器。每一位置 1，则 GPIO_STATUS_INTERRUPT 中的相应位也会清零。注：推荐使用此寄存器来清零 GPIO_STATUS_INTERRUPT。(只写)

Register 2.30. GPIO_STATUS1_W1TS_REG (0x0054)



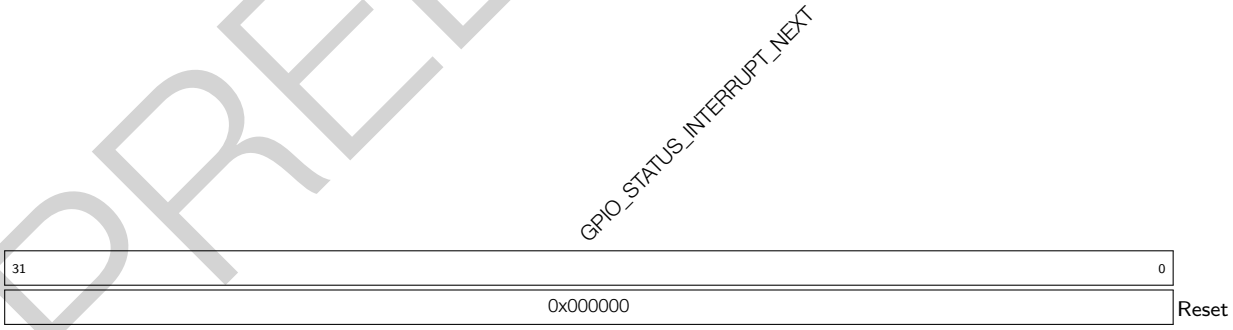
GPIO_STATUS1_W1TS GPIO32 ~ 48 中断状态置位寄存器。每一位置 1，GPIO_STATUS_INTERRUPT1 中相应位也置 1。注：推荐使用此寄存器来置位 GPIO_STATUS_INTERRUPT1。(只写)

Register 2.31. GPIO_STATUS1_W1TC_REG (0x0058)



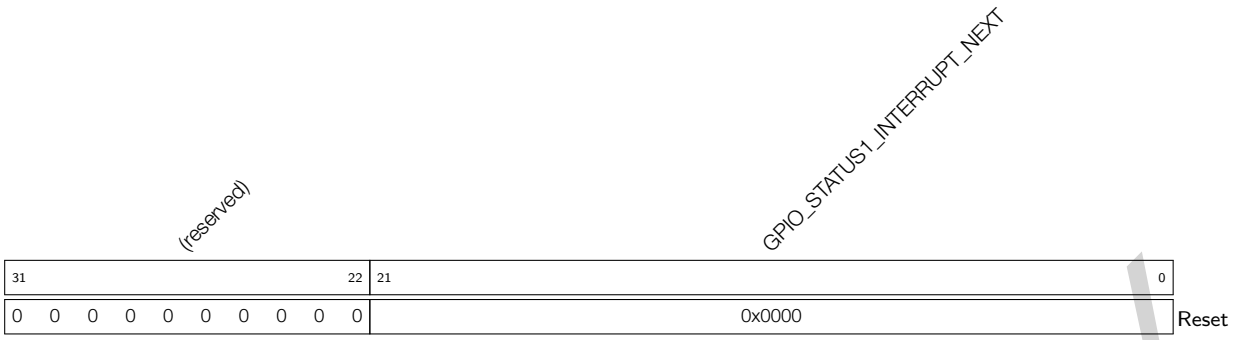
GPIO_STATUS1_W1TC GPIO32 ~ 48 中断状态清除寄存器。每一位置 1，则 GPIO_STATUS_INTERRUPT1 中的相应位也将清零。注：推荐使用此寄存器来清零 GPIO_STATUS_INTERRUPT1。(只写)

Register 2.32. GPIO_STATUS_NEXT_REG (0x014C)



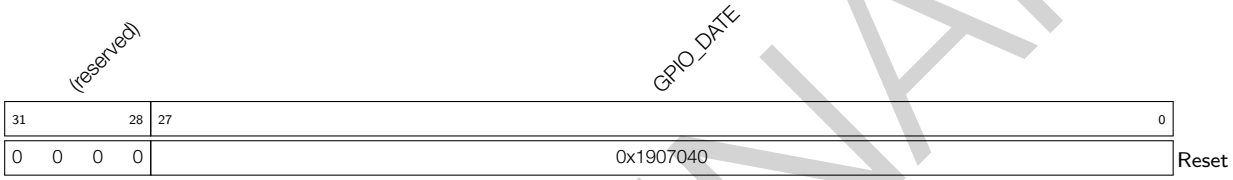
GPIO_STATUS_INTERRUPT_NEXT GPIO0 ~ 31 中断源信号，可以设置为上升沿中断、下降沿中断、电平敏感中断或任一沿中断。(只读)

Register 2.33. GPIO_STATUS_NEXT1_REG (0x0150)



GPIO_STATUS1_INTERRUPT_NEXT GPIO32 ~ 48 的中断源信号。(只读)

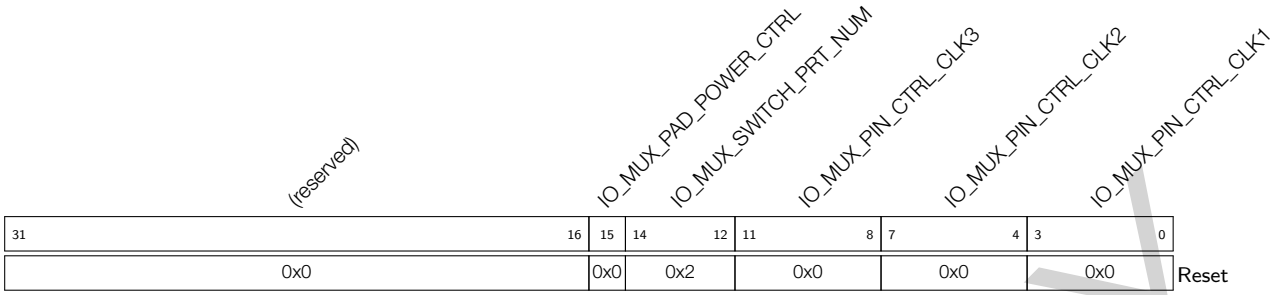
Register 2.34. GPIO_REG_DATE_REG (0x06FC)



GPIO_DATE 版本控制寄存器。(读/写)

2.15.2 IO MUX 寄存器

Register 2.35. IO_MUX_PIN_CTRL (0x0000)



IO_MUX_PIN_CTRL_CLK*x* 配置 I2S0 外设时钟输出到：

CLK_OUT1，配置 IO_MUX_PIN_CTRL_CLK1 = 0x0；

CLK_OUT2，配置 IO_MUX_PIN_CTRL_CLK2 = 0x0；

CLK_OUT3，配置 IO_MUX_PIN_CTRL_CLK3 = 0x0。

配置 I2S1 外设时钟输出到：

CLK_OUT1，配置 IO_MUX_PIN_CTRL_CLK1 = 0xF；

CLK_OUT2，配置 IO_MUX_PIN_CTRL_CLK2 = 0xF；

CLK_OUT3，配置 IO_MUX_PIN_CTRL_CLK3 = 0xF。

说明：

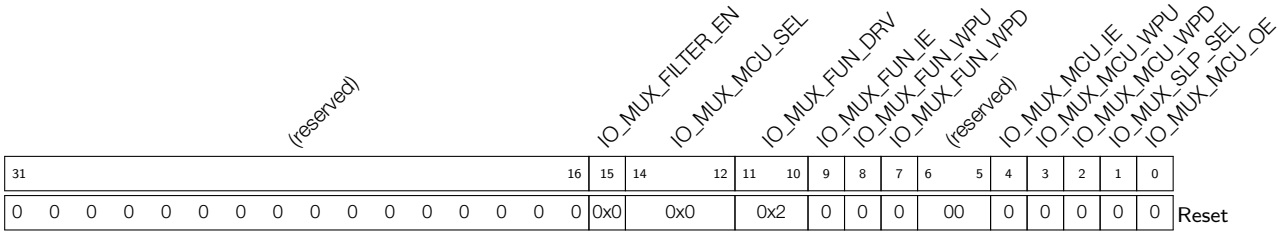
只能有上述配置组合。

CLK_OUT1 ~ 3 可在 IO MUX 管脚功能列表中查询。

IO_MUX_SWITCH_PRT_NUM GPIO 管脚电源切换延时，延时单位为一个 APB 时钟周期。

IO_MUX_PAD_POWER_CTRL 选择 GPIO33 ~ 37 的电源电压。1：选择 VDD_SPI 1.8 V 供电；0：选择 VDD3P3_CPU 3.3 V 供电。

Register 2.36. IO_MUX_*n*_REG (*n*: GPIO0-GPIO21, GPIO26-GPIO48) (0x0010+4n*)**



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

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

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

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

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

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

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

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

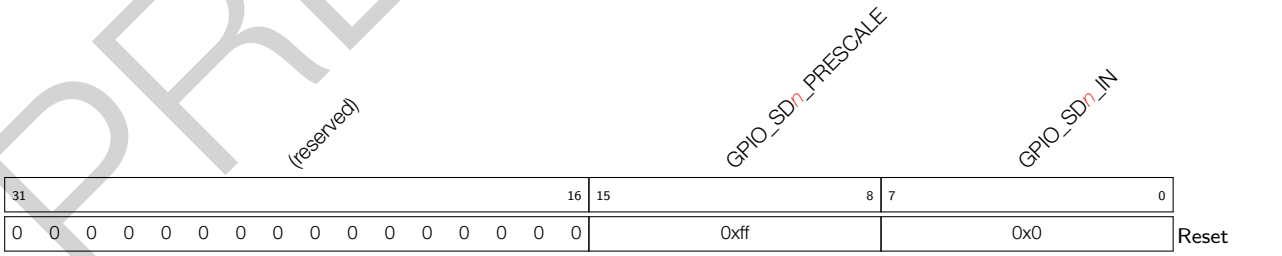
IO_MUX_FUN_DRV 选择管脚驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

IO_MUX_MCU_SEL 为信号选择 IO MUX 功能。0：选择 Function 0；1：选择 Function 1；以此类推。(读/写)

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

2.15.3 SDM 寄存器

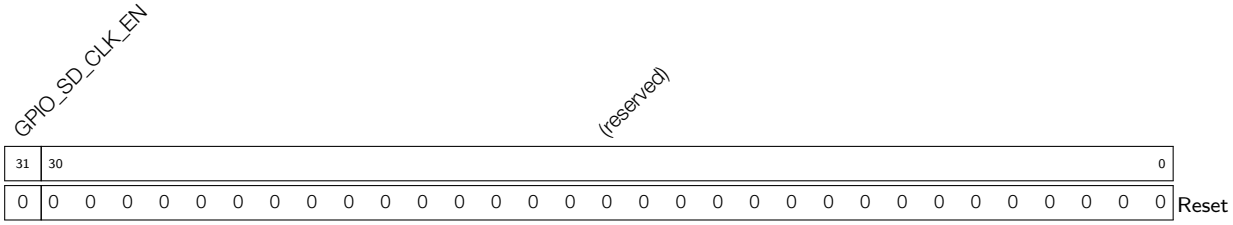
Register 2.37. GPIO_SIGMADELTA*n*_REG (*n*: 0-7) (0x0000+4n*)**



GPIO_SD*n*_IN 配置 SDM 输出信号的占空比。(读/写)

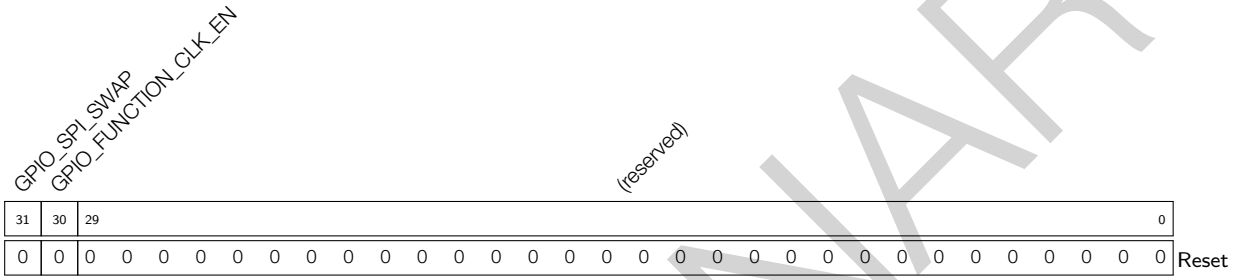
GPIO_SD*n*_PRESCALE 配置 APB_CLK 分频系数。(读/写)

Register 2.38. GPIO_SIGMADELTA_CG_REG (0x0020)



GPIO_SD_CLK_EN 使能 SDM 配置寄存器的时钟。(读/写)

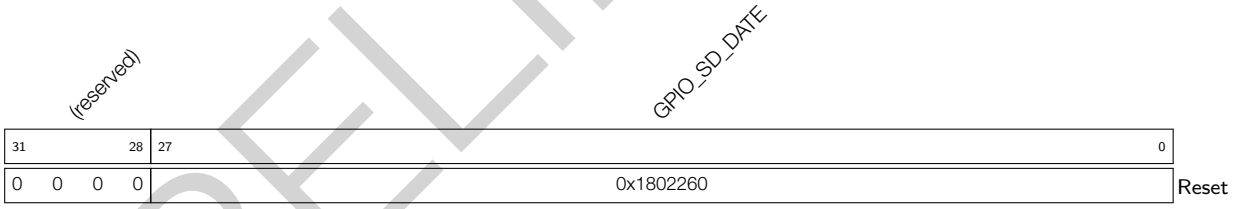
Register 2.39. GPIO_SIGMADELTA_MISC_REG (0x0024)



GPIO_FUNCTION_CLK_EN 使能 SDM 的时钟。(读/写)

GPIO_SPI_SWAP 保留。(读/写)

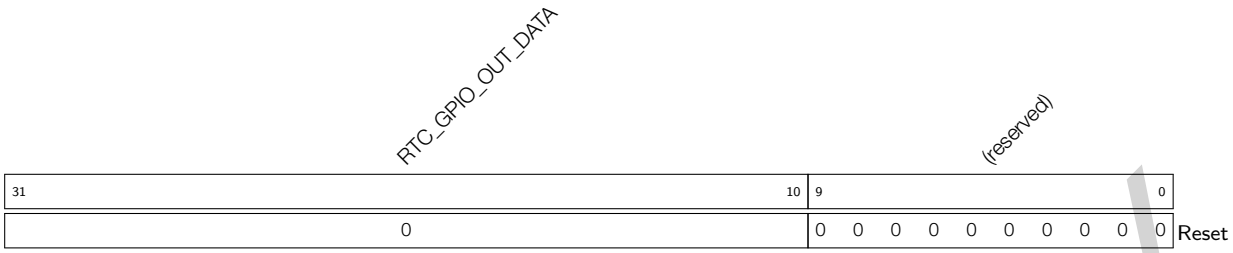
Register 2.40. GPIO_SIGMADELTA_VERSION_REG (0x0028)



GPIO_SD_DATE 版本控制寄存器。(读/写)

2.15.4 RTC IO MUX 寄存器

Register 2.41. RTC_GPIO_OUT_REG (0x0000)

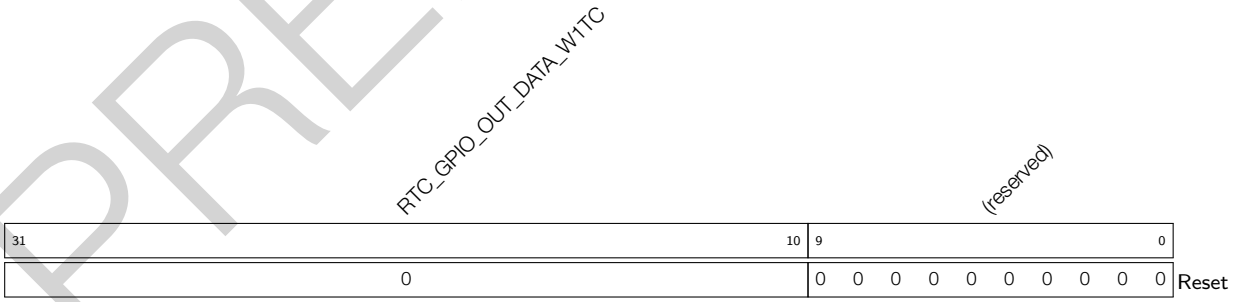


RTC_GPIO_OUT_DATA GPIO0 ~ 21 输出寄存器。bit10 对应 GPIO0，bit11 对应 GPIO1，以此类推。(读/写)

Register 2.42. RTC_GPIO_OUT_W1TS_REG (0x0004)

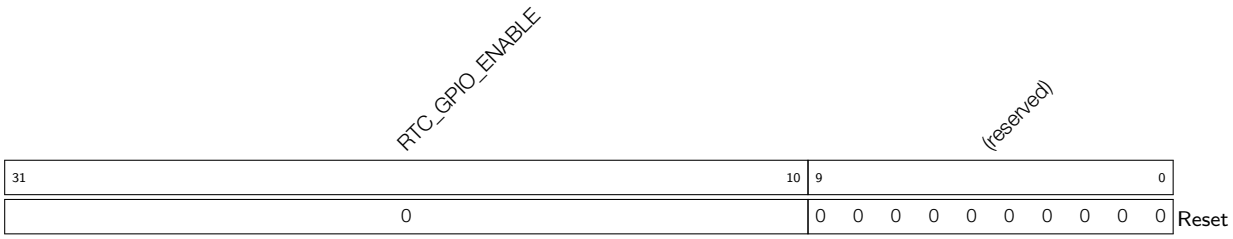
RTC_GPIO_OUT_DATA_W1TS GPIO0 ~ 21 输出置位寄存器。每一位置 1，RTC_GPIO_OUT_REG 中相应位也置 1。注：推荐使用此寄存器来置位 RTC_GPIO_OUT_REG。(只写)

Register 2.43. RTC_GPIO_OUT_W1TC_REG (0x0008)



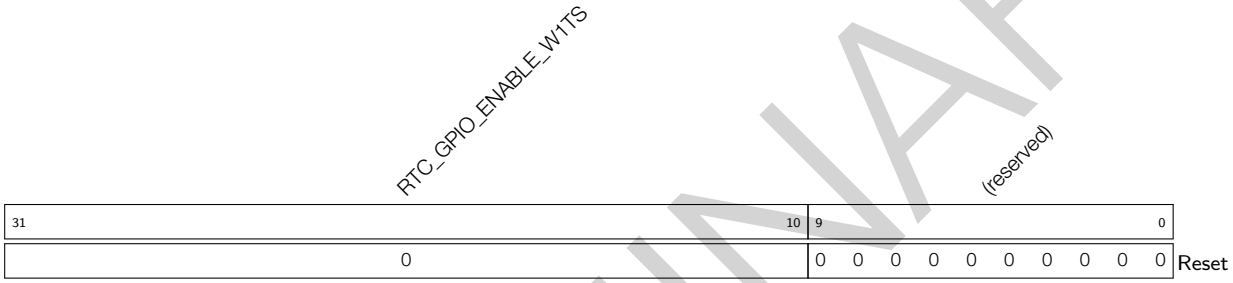
RTC_GPIO_OUT_DATA_W1TC GPIO0 ~ 21 输出清零寄存器。每一位置 1，则 RTC_GPIO_OUT_REG 中相应位将被清零。注：推荐使用此寄存器来清零 RTC_GPIO_OUT_REG。(只写)

Register 2.44. RTC_GPIO_ENABLE_REG (0x000C)



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

Register 2.45. RTC_GPIO_ENABLE_W1TS_REG (0x0010)

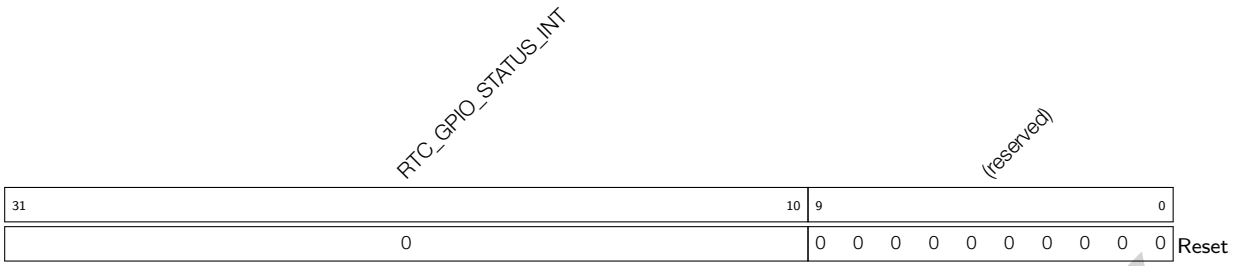


RTC_GPIO_ENABLE_W1TS GPIO0 ~ 21 输出使能置位寄存器。每一位置 1，则 RTC_GPIO_ENABLE_REG 中相应位也将置 1。注：推荐使用此寄存器来置位 RTC_GPIO_ENABLE_REG。(只写)

Register 2.46. RTC_GPIO_ENABLE_W1TC_REG (0x0014)

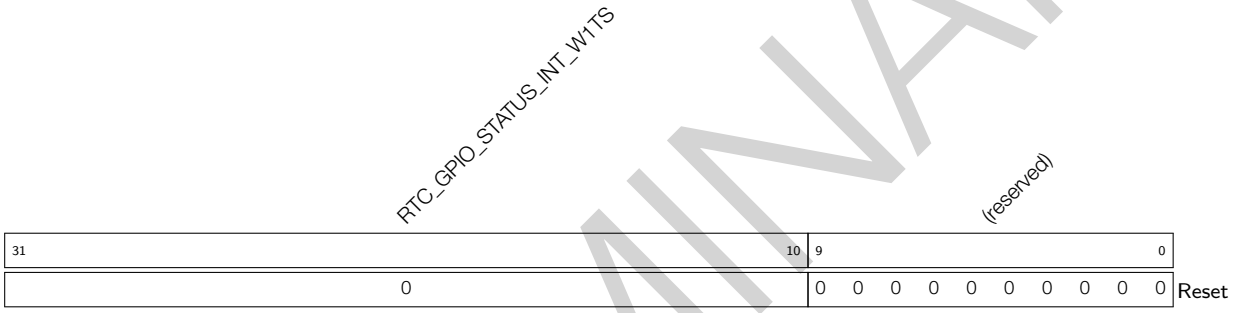
RTC_GPIO_ENABLE_W1TC GPIO0 ~ 21 输出使能清零寄存器。每一位置 1，则 RTC_GPIO_ENABLE_REG 中相应位将被清零。注：推荐使用此寄存器来清零 RTC_GPIO_ENABLE_REG。(只写)

Register 2.47. RTC_GPIO_STATUS_REG (0x0018)



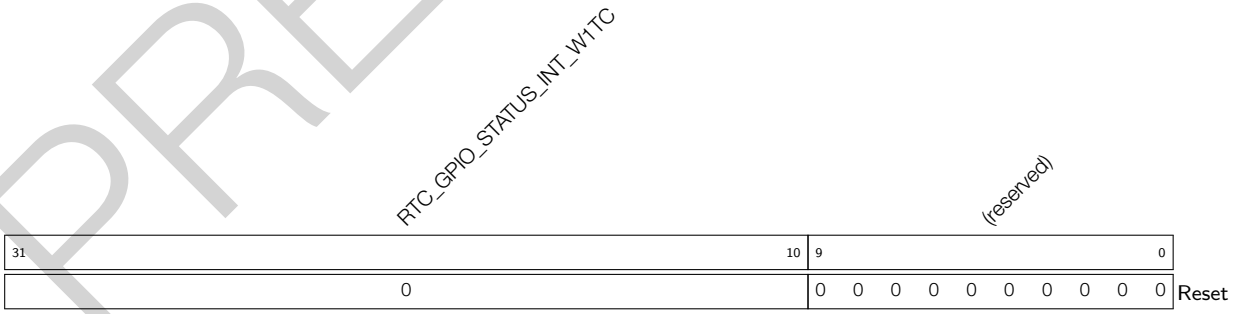
RTC_GPIO_STATUS_INT GPIO0 ~ 21 中断状态寄存器。bit10 对应 GPIO0，bit11 对应 GPIO1，以此类推。此寄存器应同时与 RTC_GPIO_PIN*n*_REG 寄存器中的 RTC_GPIO_PIN*n*_INT_TYPE 中断类型配合使用。0：代表没有中断；1：代表有相应中断。(读/写)

Register 2.48. RTC_GPIO_STATUS_W1TS_REG (0x001C)



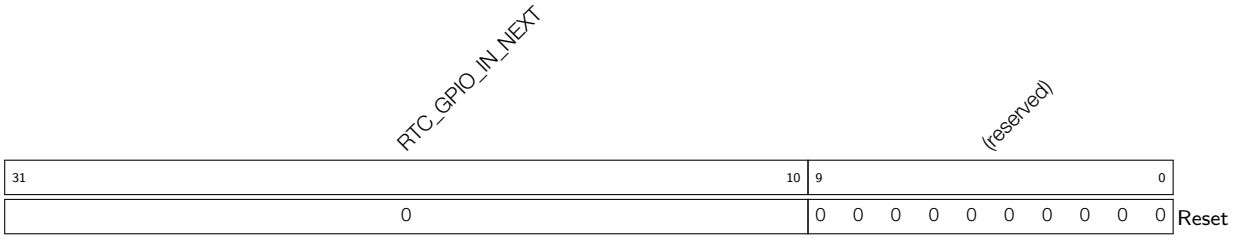
RTC_GPIO_STATUS_INT_W1TS GPIO0 ~ 21 中断状态置位寄存器。每一位置 1，则 RTC_GPIO_STATUS_INT 中相应位也将置 1。注：推荐使用此寄存器来置位 RTC_GPIO_STATUS_INT。(只写)

Register 2.49. RTC_GPIO_STATUS_W1TC_REG (0x0020)



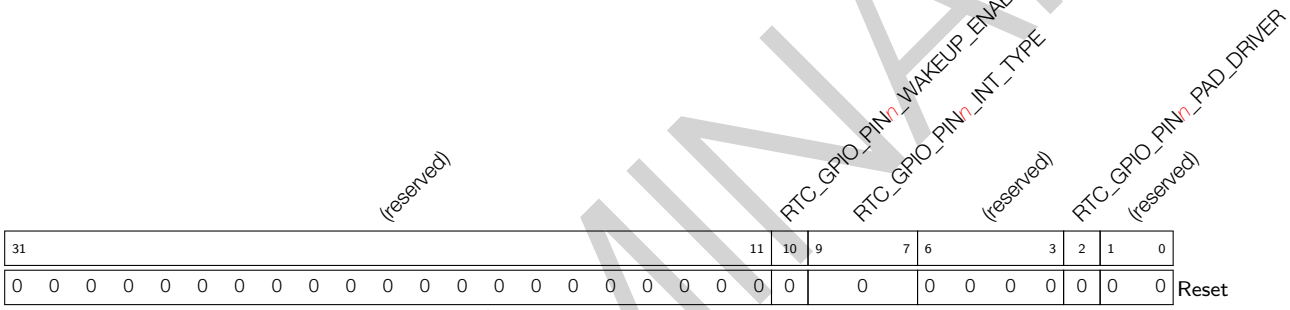
RTC_GPIO_STATUS_INT_W1TC GPIO0 ~ 21 中断状态清零寄存器。每一位置 1，则 RTC_GPIO_STATUS_INT 中的相应位也将清零。注：推荐使用此寄存器来清零 RTC_GPIO_STATUS_INT。(只写)

Register 2.50. RTC_GPIO_IN_REG (0x0024)



RTC_GPIO_IN_NEXT GPIO0 ~ 21 输入值。bit10 对应 GPIO0，bit11 对应 GPIO1，以此类推。每个 bit 代表 pad 的片外输入值，比如片外引脚为高电平，此 bit 值应为 1，片外引脚为低电平，此 bit 值应为 0。（只读）

Register 2.51. RTC_GPIO_PINn_REG (n: 0-21) (0x0028+0x4*n)



RTC_GPIO_PINn_PAD_DRIVER 管脚驱动选择寄存器。0：正常输出；1：开漏输出。（读/写）

RTC_GPIO_PINn_INT_TYPE GPIO 中断类型选择寄存器。（读/写）

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

RTC_GPIO_PINn_WAKEUP_ENABLE GPIO 唤醒使能。只能将芯片从 Light-sleep 中唤醒。（读/写）

Register 2.52. RTC_IO_TOUCH_PAD*n*_REG (*n*: 0-14) (0x0084+0x4**n*)

Bits	31	30–29	28	27	26–23	22	21	20	19	18–17	16	15	14	13	12–0
Field	(reserved)	RTC_IO_TOUCH_PADn_DRV	RTC_IO_TOUCH_PADn_RDE	RTC_IO_TOUCH_PADn_RUE	(reserved)	RTC_IO_TOUCH_PADn_START	RTC_IO_TOUCH_PADn_TIE_OPT	RTC_IO_TOUCH_PADn_XPD	RTC_IO_TOUCH_PADn_MUX_SEL	RTC_IO_TOUCH_PADn_FUN_SEL	RTC_IO_TOUCH_PADn_SLP_SEL	RTC_IO_TOUCH_PADn_SLP_IE	RTC_IO_TOUCH_PADn_SLP_OE	RTC_IO_TOUCH_PADn_FUN_IE	(reserved)
Reset	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0

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

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

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

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

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

RTC_IO_TOUCH_PAD*n*_MUX_SEL 选择连接 RTC 管脚输入或数字管脚输入,可以置 0,即选择数字管脚输入。(读/写)

RTC_IO_TOUCH_PAD*n*_XPD 触摸传感器上电。(读/写)

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

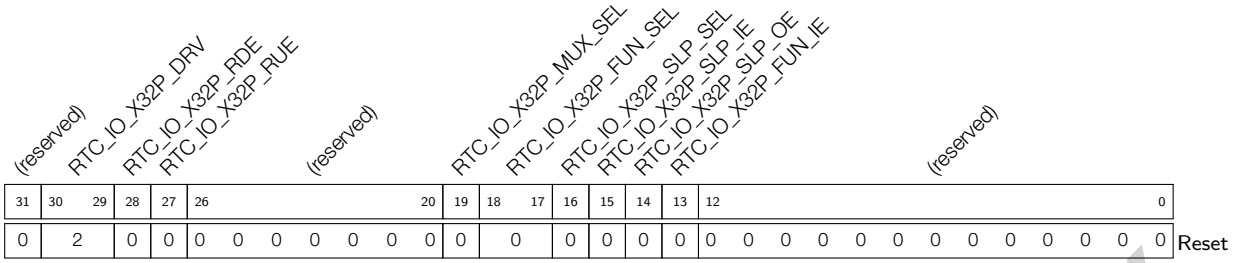
RTC_IO_TOUCH_PAD*n*_START 启动触摸传感器。(读/写)

RTC_IO_TOUCH_PAD*n*_RUE 使能管脚上拉。1:内部上拉使能;0:内部上拉关闭。(读/写)

RTC_IO_TOUCH_PAD*n*_RDE 使能管脚下拉。1:内部下拉使能;0:内部下拉关闭。(读/写)

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

Register 2.53. RTC_IO_XTAL_32P_PAD_REG (0x00C0)



RTC_IO_X32P_FUN_IE 使能工作模式下管脚输入。(读/写)

RTC_IO_X32P_SLP_OE 使能睡眠模式下管脚输出。(读/写)

RTC_IO_X32P_SLP_IE 使能睡眠模式下管脚输入。(读/写)

RTC_IO_X32P_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(读/写)

RTC_IO_X32P_FUN_SEL Function 选择。(读/写)

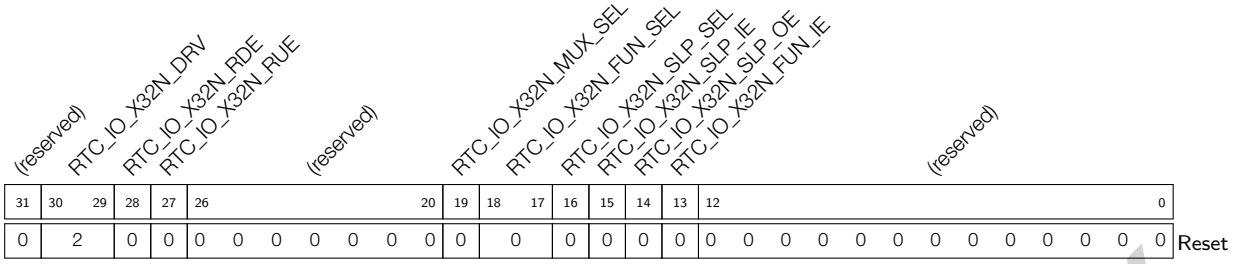
RTC_IO_X32P_MUX_SEL 1：选择使用 RTC GPIO；0：选择使用数字 GPIO。(读/写)

RTC_IO_X32P_RUE 管脚上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

RTC_IO_X32P_RDE 管脚下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

RTC_IO_X32P_DRV 选择管脚的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

Register 2.54. RTC_IO_XTAL_32N_PAD_REG (0x00C4)



RTC_IO_X32N_FUN_IE 工作模式下输入使能。(读/写)

RTC_IO_X32N_SLP_OE 睡眠模式下输出使能。(读/写)

RTC_IO_X32N_SLP_IE 睡眠模式下输入使能。(读/写)

RTC_IO_X32N_SLP_SEL 1：使能睡眠模式；0：关闭睡眠模式。(读/写)

RTC_IO_X32N_FUN_SEL Function 选择。(读/写)

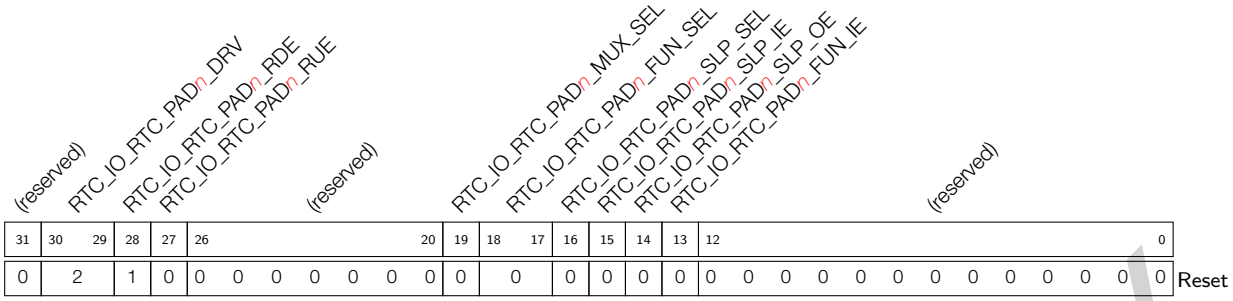
RTC_IO_X32N_MUX_SEL 1：选择使用 RTC GPIO；0：选择使用数字 GPIO。(读/写)

RTC_IO_X32N_RUE Pad 上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

RTC_IO_X32N_RDE Pad 下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

RTC_IO_X32N_DRV 选择 pad 的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

Register 2.55. RTC_IO_RTC_PADn_REG (n: 17-21) (0x00C8, 0x00CC, 0x00D0, 0x00D4, 0x00D8)



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

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

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

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

RTC_IO_RTC_PADn_FUN_SEL Function 选择。(读/写)

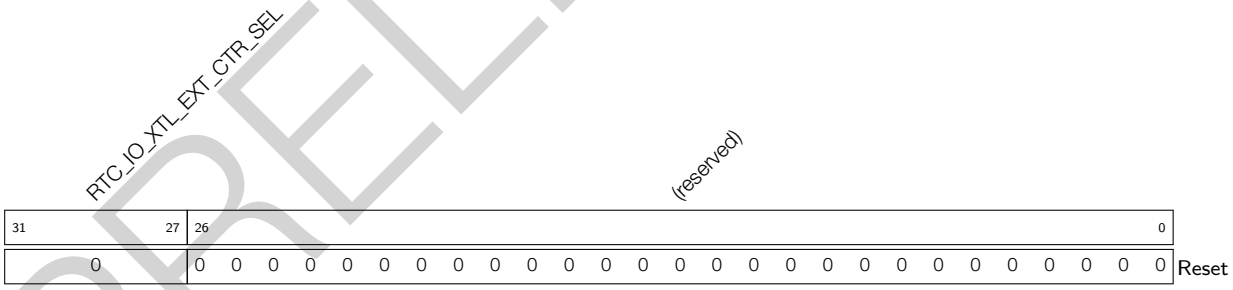
RTC_IO_RTC_PADn_MUX_SEL 1：选择使用 RTC GPIO；2：选择使用数字 GPIO。(读/写)

RTC_IO_RTC_PADn_RUE 管脚上拉使能。1：内部上拉使能；0：内部上拉关闭。(读/写)

RTC_IO_RTC_PADn_RDE 管脚下拉使能。1：内部下拉使能；0：内部下拉关闭。(读/写)

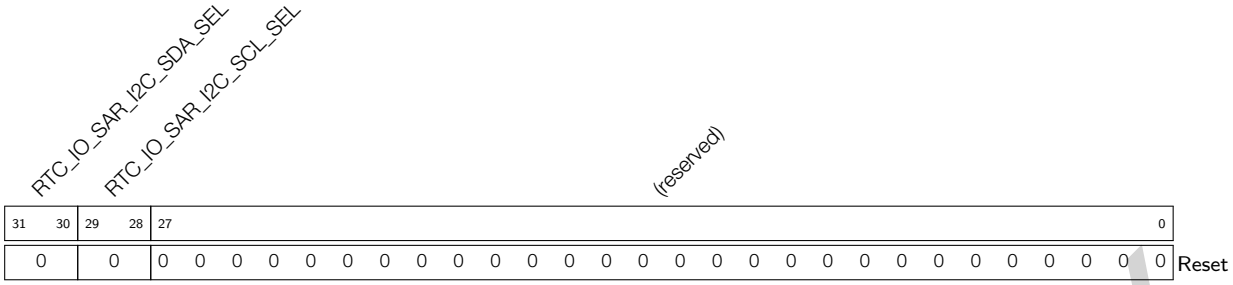
RTC_IO_RTC_PADn_DRV 选择管脚的驱动强度。0：~5 mA；1：~10 mA；2：~20 mA；3：~40 mA。(读/写)

Register 2.56. RTC_IO_XTL_EXT_CTR_REG (0x00E0)



RTC_IO_XTL_EXT_CTR_SEL 选择睡眠模式下外部晶振断电使能源。0：选择 GPIO0；1：选择 GPIO1，以此类推。被选择管脚的值异或 RTC_CNTL_EXT_XTL_CONF_REG[30] 上的逻辑值时晶振断电使能信号。(读/写)

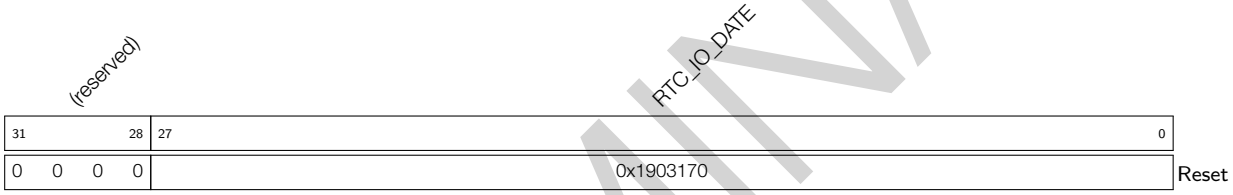
Register 2.57. RTC_IO_SAR_I2C_IO_REG (0x00E4)



RTC_IO_SAR_I2C_SCL_SEL 选择 RTC I2C SCL 信号连接的管脚。0：选择 RTC GPIO0；1：选择 RTC GPIO2。（读/写）

RTC_IO_SAR_I2C_SDA_SEL 选择 RTC I2C SDA 信号连接的管脚。0：选择 RTC GPIO1；1：选择 RTC GPIO3。（读/写）

Register 2.58. RTC_IO_DATE_REG (0x01FC)



RTC_IO_DATE 版本控制寄存器（读/写）

3 复位和时钟

3.1 复位

3.1.1 概述

ESP32-S3 提供四种级别的复位方式，分别是 CPU 复位、内核复位、系统复位和芯片复位。除芯片复位外其它复位方式不影响片上内存存储的数据。图 3-1 展示了整个芯片系统的结构以及四种复位等级。

3.1.2 结构图

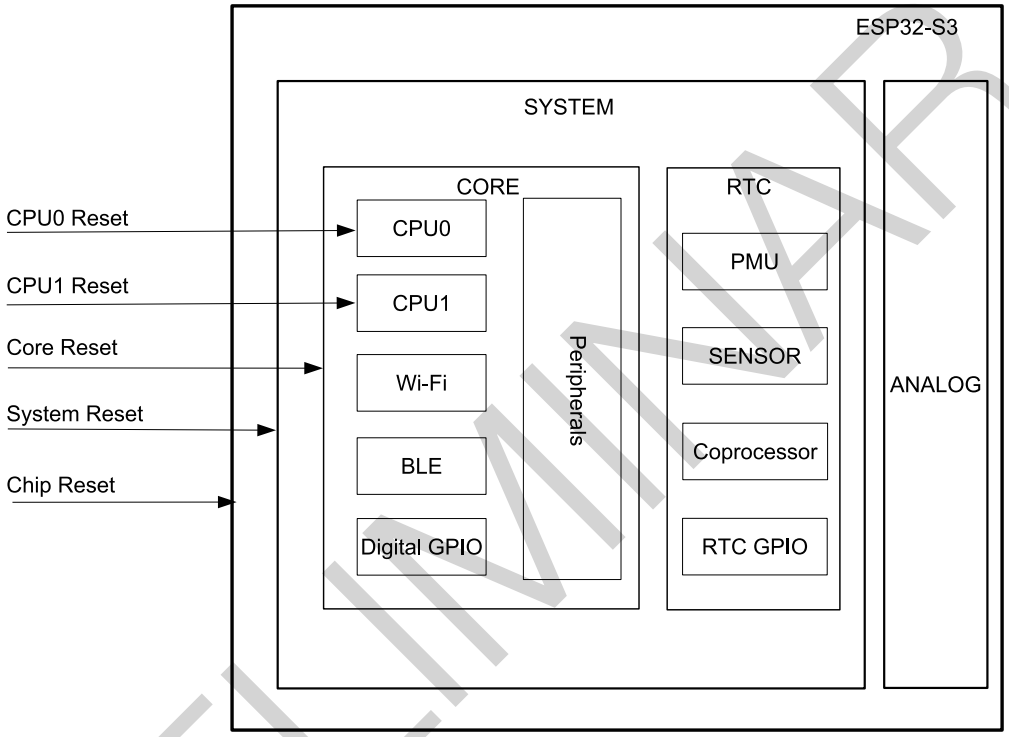


图 3-1. 四种复位等级

3.1.3 特性

- 支持四种复位等级：
  - CPU 复位：只复位 CPU*x* 核。这里的 CPU*x* 代表 CPU0 或 CPU1。复位释放后，程序将从 CPU*x* Reset Vector 开始执行。每个 CPU 核拥有独立的复位逻辑。
  - 内核复位：复位除 RTC 以外的其它数字系统，包括 CPU0、CPU1、外设、Wi-Fi、Bluetooth® LE 及数字 GPIO；
  - 系统复位：复位包括 RTC 在内的整个数字系统；
  - 芯片复位：复位整个芯片。
- 支持软件复位和硬件复位：
  - 软件复位：CPU*x* 配置相关寄存器可触发软件复位，见章节 14 *低功耗管理 (RTC_CNTL) [to be added later]*；

– 硬件复位：硬件复位直接由硬件电路触发。

说明：
如果 CPU 复位来自 CPU0，则 SENSITIVE 寄存器也将复位。

3.1.4 功能描述

上述任一复位源产生时，CPU0 和 CPU1 均将立刻复位。复位释放后，CPU0 和 CPU1 可分别通过读取寄存器 RTC_CNTL_RESET_CAUSE_PROCPU 和 RTC_CNTL_RESET_CAUSE_APPCPU 获取复位源。这两个寄存器记录的复位源除了复位级别为 CPU 复位的复位源分别对应自身的 CPU*x* 以外，其余的复位源保持一致。

表 3-1 列出了从上述两个寄存器中可能读出的复位源。

表 3-1. 复位源

编码	复位源	复位等级	说明
0x01	芯片复位[1]	芯片复位	—
0x0F	欠压系统复位	系统复位或芯片复位	欠压检测器触发的系统复位[2]
0x10	RWDT 系统复位	系统复位	详见章节 7 看门狗定时器
0x13	GLITCH 复位	系统复位	详见章节 17 时钟毛刺检测 *[to be added later]*
0x03	软件系统复位	内核复位	配置 RTC_CNTL_SW_SYS_RST 寄存器触发
0x05	Deep-sleep 复位	内核复位	详见章节 14 低功耗管理 *(RTC_CNTL) [to be added later]*
0x07	MWDT0 内核复位	内核复位	详见章节 7 看门狗定时器
0x08	MWDT1 内核复位	内核复位	详见章节 7 看门狗定时器
0x09	RWDT 内核复位	内核复位	详见章节 7 看门狗定时器
0x0B	MWDT0 CPU*x* 复位	CPU 复位	详见章节 7 看门狗定时器
0x0C	软件 CPU*x* 复位	CPU 复位	配置 RTC_CNTL_SW_PROCPU_RST 寄存器触发
0x0D	RWDT CPU*x* 复位	CPU 复位	详见章节 7 看门狗定时器
0x11	MWDT1 CPU*x* 复位	CPU 复位	详见章节 7 看门狗定时器
0x12	Super Watchdog 复位	系统复位	详见章节 7 看门狗定时器
0x14	eFuse 复位	内核复位	eFuse CRC 校验错误触发复位

[1] 芯片复位的触发源包括以下三项：
- 芯片上电触发芯片复位
- 欠压检测器触发芯片复位
- 超级看门狗 (SWD) 触发芯片复位

[2] 欠压检测器在检测到欠压状态时，将根据寄存器配置，选择触发系统复位或者芯片复位。详见章节 14 低功耗管理 *(RTC_CNTL) [to be added later]*。

3.2 时钟

3.2.1 概述

ESP32-S3 的时钟主要来源于振荡器 (oscillator, OSC)、RC 振荡电路和 PLL 时钟生成电路。上述时钟源产生的时钟经时钟分频器或时钟选择器等时钟模块的处理，使得大部分功能模块可以根据不同功耗和性能需求来获取及选择对应频率的工作时钟。图 3-2 为系统时钟结构。

3.2.2 结构图

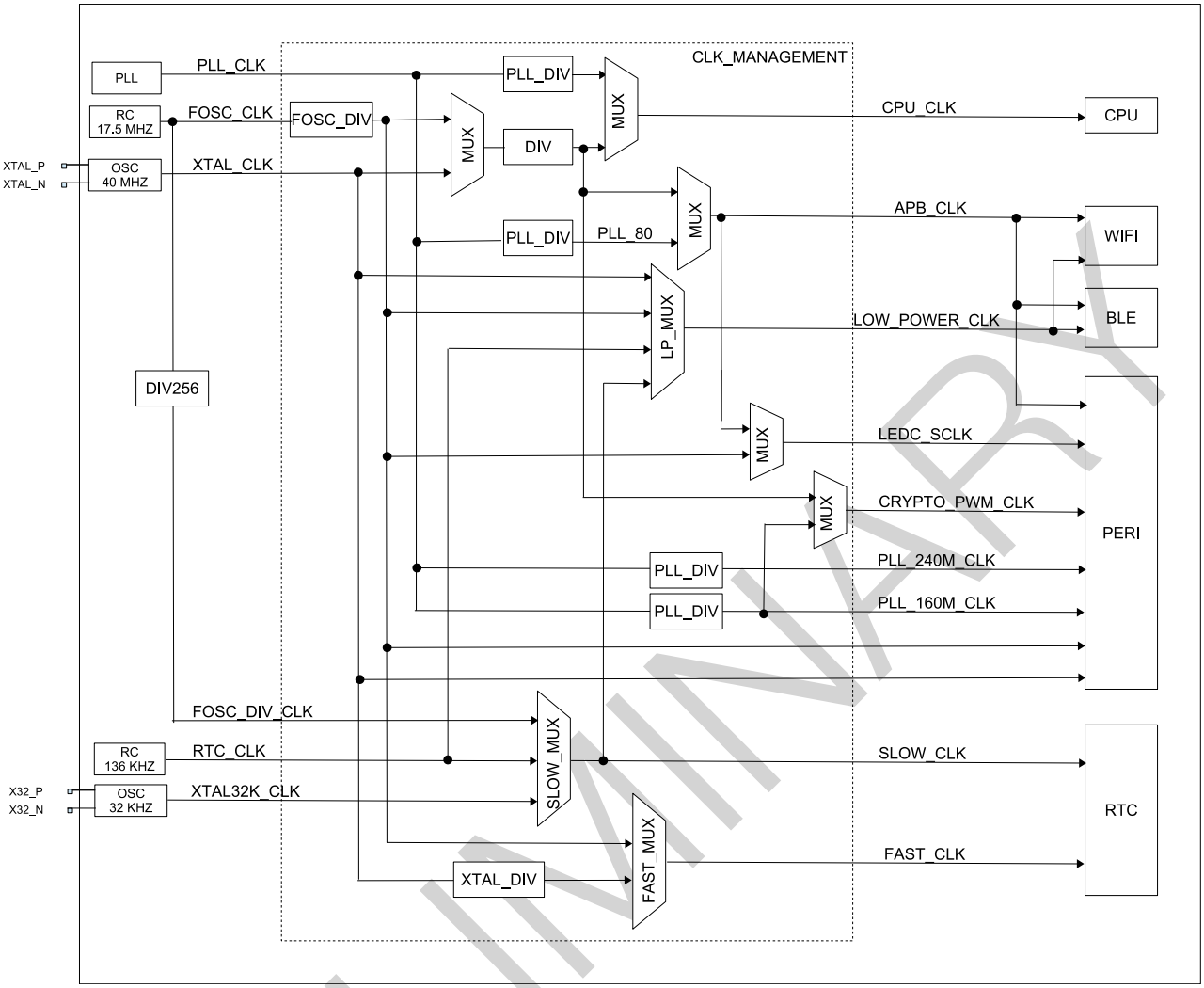


图 3-2. 系统时钟

3.2.3 特性

ESP32-S3 的时钟根据频率不同，可分为：

- 高性能时钟，主要为 CPU 和数字外设提供工作时钟
  - PLL_CLK：320 MHz 或 480 MHz 内部 PLL 时钟
  - XTAL_CLK：40 MHz 外部晶振时钟
- 低功耗时钟，主要为 RTC 模块以及部分处于低功耗模式的外设提供工作时钟
  - XTAL32K_CLK：32 kHz 外部晶振时钟
  - FOSC_CLK：内置快速 RC 振荡器时钟，频率可调节（通常为 17.5 MHz）
  - FOSC_DIV_CLK：内置快速 RC 振荡器分频时钟，由内置快速 RC 振荡器时钟经 256 分频生成
  - RTC_CLK：内置慢速 RC 振荡器，频率可调节（通常为 136 kHz）

3.2.4 功能描述

3.2.4.1 CPU 时钟

如图 3-2 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、FOSC_CLK 或 XTAL_CLK 作为 CPU_CLK 的时钟源。具体请参考表 3-2 和表 3-3。默认状态下，CPU 的时钟为 XTAL_CLK，且分频系数为 2 分频，即 20 MHz。

表 3-2. CPU_CLK 时钟源选择

SYSTEM_SOC_CLK_SEL 值	时钟源
0	XTAL_CLK
1	PLL_CLK
2	FOSC_CLK

表 3-3. CPU_CLK 时钟频率

时钟源	SEL_0[a]	SEL_2[b]	SEL_3[c]	CPU 时钟频率
XTAL_CLK	0	-	-	CPU_CLK = XTAL_CLK/(SYSTEM_PRE_DIV_CNT + 1) SYSTEM_PRE_DIV_CNT 默认值为 1，范围 0 ~ 1023。
PLL_CLK (480 MHz)	1	1	0	CPU_CLK = PLL_CLK/6 CPU_CLK 频率为 80 MHz。
PLL_CLK (480 MHz)	1	1	1	CPU_CLK = PLL_CLK/3 CPU_CLK 频率为 160 MHz。
PLL_CLK (480 MHz)	1	1	2	CPU_CLK = PLL_CLK/2 CPU_CLK 频率为 240 MHz。
PLL_CLK (320 MHz)	1	0	0	CPU_CLK = PLL_CLK/4 CPU_CLK 频率为 80 MHz。
PLL_CLK (320 MHz)	1	0	1	CPU_CLK = PLL_CLK/2 CPU_CLK 频率为 160 MHz。
FOSC_CLK	2	-	-	CPU_CLK = FOSC_CLK/(SYSTEM_PRE_DIV_CNT + 1) SYSTEM_PRE_DIV_CNT 默认值为 1，范围 0 ~ 1023。

[a] 寄存器 SYSTEM_SOC_CLK_SEL 的值
[b] 寄存器 SYSTEM_PLL_FREQ_SEL 的值
[c] 寄存器 SYSTEM_CPUPERIOD_SEL 的值

3.2.4.2 外设时钟

外设所需要的时钟包括 APB_CLK、CRYPTO_PWM_CLK、PLL_160M_CLK、PLL_240M_CLK、LEDC_CLK、XTAL_CLK 和 FOSC_CLK。表 3-4 列出了接入各个外设的时钟。

表 3-4. 外设时钟

外设	XTAL_CLK	APB_CLK	PLL_160M_CLK	PLL_240M_CLK	FOSC_CLK	CRYPTO_PWM_CLK	LEDC_CLK
TIMG	Y	Y					
I2S	Y		Y	Y			
UHCI		Y					
UART	Y	Y			Y		
RMT	Y	Y			Y		
PWM						Y	
I2C	Y				Y		
SPI	Y	Y					
PCNT		Y					
eFuse Controller		Y					
SARADC		Y		Y			
USB		Y					
CRYPTO						Y	
TWAI Controller		Y					
SDIO HOST	Y		Y				
LEDC	Y	Y			Y		Y
LCD_CAM	Y		Y	Y			
SYS_TIMER	Y	Y					

APB_CLK 时钟

如表 3-5 所示，APB_CLK 的频率由 CPU_CLK 的时钟源决定。

表 3-5. APB_CLK 时钟

CPU_CLK 时钟源	APB_CLK 频率
PLL_CLK	80 MHz
XTAL_CLK	CPU_CLK
FOSC_CLK	CPU_CLK

CRYPTO_PWM_CLK 时钟

如表 3-6 所示，CRYPTO_PWM_CLK 的频率由 CPU_CLK 的时钟源决定。

表 3-6. CRYPTO_PWM_CLK 时钟

CPU_CLK 时钟源	CRYPTO_PWM_CLK 频率
PLL_CLK	160 MHz
XTAL_CLK	CPU_CLK
FOSC_CLK	CPU_CLK

PLL_160M_CLK 时钟

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

PLL_240M_CLK 时钟

PLL_240M_CLK 是 PLL_CLK 根据当前 PLL 的频率分频所得。

LEDC_CLK 时钟

LEDC 模块能将 FOSC_CLK 作为时钟源使用，即在 APB_CLK 关闭的时候，LEDC 也可工作。换而言之，当系统处于低功耗模式时，其他外设都将停止工作（APB_CLK 关闭），但是 LEDC 仍然可以通过 FOSC_CLK 来正常工作。

3.2.4.3 Wi-Fi 和 Bluetooth LE 时钟

Wi-Fi 和 Bluetooth LE 必须在 CPU_CLK 时钟源选择 PLL_CLK 下才能工作。只有当 Wi-Fi 和 Bluetooth LE 进入低功耗模式时，才能暂时关闭 PLL_CLK。

LOW_POWER_CLK 允许选择 XTAL32K_CLK、XTAL_CLK、FOSC_CLK、SLOW_CLK（RTC 当前所选的慢速时钟）用于 Wi-Fi 和 Bluetooth LE 的低功耗模式。

3.2.4.4 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 FOSC_DIV_CLK，用于驱动功耗管理模块。FAST_CLK 允许选择 XTAL_CLK 或 FOSC_CLK 的分频时钟，用于驱动片上传感器模块。

4 芯片 Boot 控制

4.1 概述

ESP32-S3 共有四个 Strapping 管脚：

- GPIO0
- GPIO3
- GPIO45
- GPIO46

Strapping 管脚用于控制 ESP32-S3 芯片上电或硬件复位时的一些功能：

- 控制 Boot 模式
- 控制 ROM 代码日志打印到 UART
- 设置 VDD_SPI 电压
- 控制 JTAG 信号源

在系统复位过程中，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 3 复位和时钟），硬件将采样 Strapping 管脚电平存储到锁存器中，并一直保持到芯片掉电或关闭。GPIO0、GPIO3、GPIO45 和 GPIO46 锁存的状态可以通过软件从寄存器 GPIO_STRAPPING 中读取。

GPIO0、GPIO45 和 GPIO46 默认连接内部上拉/下拉。如果这些管脚没有外部连接或者连接的外部线路处于高阻抗状态，内部弱上拉/下拉将决定这几个管脚输入电平的默认值，如表 4-1 所示。

表 4-1. Strapping 管脚默认上拉/下拉

管脚	默认值
GPIO0	上拉
GPIO3	N/A
GPIO45	下拉
GPIO46	下拉

如需改变 Strapping 管脚的默认值，用户可以应用外部下拉/上拉电阻，或者应用主机 MCU 的 GPIO 来控制 ESP32-S3 上电复位时的 Strapping 管脚电平。复位释放后，Strapping 管脚和普通管脚功能相同。

4.2 Boot 模式控制

复位释放后，GPIO0 和 GPIO46 共同控制 Boot 模式。

表 4-2. 系统启动模式

启动模式	GPIO0	GPIO46
SPI Boot 模式	1	x
Download Boot 模式	0	0

表 4-2 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-

S3 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。SPI Boot 模式可进一步细分为以下两种启动方式：

- 常规 flash 启动方式：支持安全启动，程序运行在 RAM 中；
- 直接启动方式：不支持安全启动，程序直接运行在 flash 中。如需使能这一启动方式，请确保下载至 flash 的 bin 文件其前两个字（地址：0x42000000）为 0xaebd041d。

在 Download Boot 模式下，用户可通过 UART 或 USB 接口将代码下载至 flash 中，或将程序加载到 SRAM 并在 SRAM 中运行程序。

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

- EFUSE_DIS_DOWNLOAD_MODE

  如果此 eFuse 设置为 1，则禁用 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。

USB Serial/JTAG 控制器可将芯片从 SPI Boot 模式强制切换到 Download Boot 模式，或从 Download Boot 模式强制切换到 SPI Boot 模式。更多信息，请参考章节 19 *USB Serial/JTAG 控制器 (USB_SERIAL_JTAG) [to be added later]*。

4.3 ROM 代码日志打印控制

在系统启动过程中，当 EFUSE_DIS_USB_DEVICE 和 EFUSE_DIS_USB 同时为 0 时，ROM 代码日志打印至 USB Serial/JTAG 控制器。否则 ROM 代码日志打印至 UART，此时 GPIO46 与 EFUSE_UART_PRINT_CONTROL 一起控制 ROM 代码日志打印。

表 4-3. ROM 代码日志打印控制

eFuse[1]	GPIO46	ROM 代码日志打印
0	x	ROM 代码日志打印至 UART，此时 GPIO46 的值被忽略
1	0	系统启动过程中使能打印
	1	系统启动过程中关闭打印
2	0	系统启动过程中关闭打印
	1	系统启动过程中使能打印
3	x	系统启动过程中始终关闭打印，此时 GPIO46 的值被忽略

[1] eFuse: EFUSE_UART_PRINT_CONTROL

ROM 代码日志打印至 UART 时，默认打印至 U0TXD，也可配置成打印到 U1TXD，具体由 EFUSE_UART_PRINT_CHANNEL 控制：

- 0：打印至 U0TXD
- 1：打印至 U1TXD

4.4 VDD_SPI 电压控制

芯片复位时，GPIO45 可用于选择 VDD_SPI 电压：

- GPIO45 = 0 时，VDD_SPI 由 VDD3P3_RTC 通过电阻 R_{SPI} 后供电（电压典型值为 3.3 V）；更多信息见《ESP32-S3 技术规格书》中图 4：ESP32-S3 电源管理。
- GPIO45 = 1 时，VDD_SPI 可选择由内置 LDO 供电（电压为 1.8 V）。

EFUSE_VDD_SPI_FORCE 设置为 1 时，可关闭上述功能。此时 VDD_SPI 电压由 EFUSE_VDD_SPI_TIEH 的值决定：

- EFUSE_VDD_SPI_TIEH = 0 时，VDD_SPI 连接 1.8 V LDO；
- EFUSE_VDD_SPI_TIEH = 1 时，VDD_SPI 连接 VDD3P3_RTC。

4.5 JTAG 信号源控制

在系统启动早期阶段，GPIO3 与 EFUSE_DIS_PAD_JTAG、EFUSE_DIS_USB_JTAG 和 EFUSE_STRAP_JTAG_SEL 一起控制 JTAG 信号源，见表 4-4。

表 4-4. JTAG 信号源控制

eFuse 1[a]	eFuse 2[b]	eFuse 3[c]	GPIO3	JTAG 信号源
0	0	0	x	JTAG 信号来自 USB Serial/JTAG 控制器，GPIO3 的值被忽略
		1	0	JTAG 信号来自相应管脚[d]
			1	JTAG 信号来自 USB Serial/JTAG 控制器
0	1	x	x	JTAG 信号来自相应管脚[d]，EFUSE_STRAP_JTAG_SEL 和 GPIO3 的值被忽略
1	0	x	x	JTAG 信号来自 USB Serial/ JTAG 控制器，EFUSE_STRAP_JTAG_SEL 和 GPIO3 的值被忽略
1	1	x	x	JTAG 被禁用，EFUSE_STRAP_JTAG_SEL 和 GPIO3 的值被忽略

[a] eFuse 1: EFUSE_DIS_PAD_JTAG
[b] eFuse 2: EFUSE_DIS_USB_JTAG
[c] eFuse 3: EFUSE_STRAP_JTAG_SEL
[d] JTAG 管脚：MTDI、MTCK、MTMS 和 MTDO

5 中断矩阵 (INTERRUPT)

5.1 概述

ESP32-S3 中断矩阵将任一外部中断源单独分配到双核 CPU 的任一外部中断上，以便在外设中断信号产生后，及时通知 CPU0 或 CPU1 进行处理。

外部中断源必须经中断矩阵分配至 CPU0/CPU1 外部中断，主要是因为：

- ESP32-S3 有 99 个外部中断源，但每个 CPU 只有 32 个中断。将这些外部中断源映射至 CPU0 中断或 CPU1 中断需要使用中断矩阵。
- 通过中断矩阵，可以根据应用需要，将一个外部中断源映射至多个 CPU0 中断或 CPU1 中断。

5.2 主要特性

- 接收 99 个外部中断源作为输入
- 生成 26 个 CPU0 的外部中断和 26 个 CPU1 的外部中断作为输出。

  注意，CPU0 剩余的 6 个中断和 CPU1 剩余的 6 个中断均为内部中断
- 支持屏蔽 CPU 的 NMI 类型中断
- 支持查询外部中断源当前的中断状态

中断矩阵的结构如图 5-1 所示。

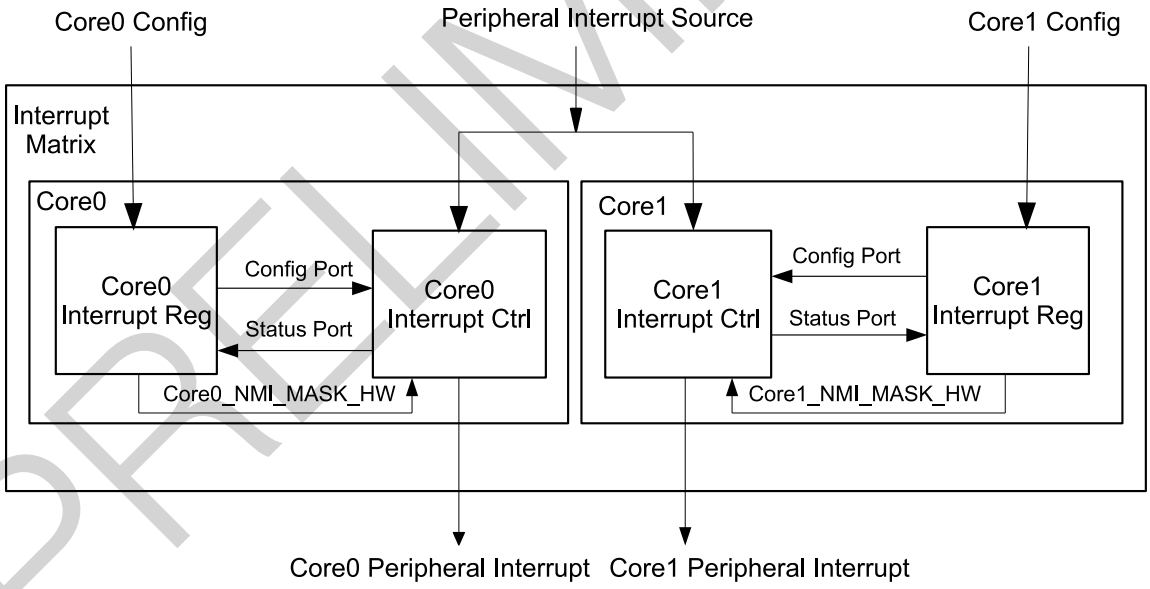


图 5-1. 中断矩阵结构图

所有外部中断源产生的中断信号均可由 CPU0 或 CPU1 进行处理。配置 CPU0 中断寄存器（图 5-1 Core0 Interrupt Reg 模块）可将外部中断源分配给 CPU0 的外部中断，此时外部中断源产生的中断信号会由 CPU0 响应处理。同理配置 CPU1 中断寄存器（图 5-1 Core1 Interrupt Reg 模块）可将中断信号交由 CPU1 响应处理。也可将外部中断源同时分配给 CPU0 和 CPU1，此时 CPU0 和 CPU1 都会接收到中断信号。

5.3 功能描述

5.3.1 外部中断源

ESP32-S3 共有 99 个外部中断源。表 5-1 列出了所有外部中断源，以及对应的中断配置寄存器与中断状态寄存器。

- “No.”：表示外部中断源序号，范围：0 ~ 98
- “中断源”：表示所有外部中断源
- “配置寄存器”：用于将外部中断源分配至 CPU0/CPU1 外部中断
- “状态寄存器”：用于读取中断源的中断状态
  - “状态寄存器 - 位”：表示在状态寄存器中的比特位置
  - “状态寄存器 - 名称”：表示状态寄存器的名称

中断源与同一行中断配置寄存器和中断状态寄存器位一一对应，例如：中断源 MAC_INTR 对应的中断配置寄存器为 INTERRUPT_CORE*x*_MAC_INTR_MAP_REG，对应的中断状态寄存器位为 INTERRUPT_CORE*x*_INTR_STATUS_0_REG 的 0 比特。

注意，表格中 CORE*x* 可以是 CORE0 (CPU0) 或 CORE1 (CPU1)。

表 5-1. CPU 外部中断配置寄存器、外部中断状态寄存器、外部中断源

No.	中断源	配置寄存器	状态寄存器	
			位	名称
0	MAC_INTR	INTERRUPT_COREx_MAC_INTR_MAP_REG	0	INTERRUPT_COREx_INTR_STATUS_0_REG
1	MAC_NMI	INTERRUPT_COREx_MAC_NMI_MAP_REG	1	
2	PWR_INTR	INTERRUPT_COREx_PWR_INTR_MAP_REG	2	
3	BB_INT	INTERRUPT_COREx_BB_INT_MAP_REG	3	
4	BT_MAC_INT	INTERRUPT_COREx_BT_MAC_INT_MAP_REG	4	
5	BT_BB_INT	INTERRUPT_COREx_BT_BB_INT_MAP_REG	5	
6	BT_BB_NMI	INTERRUPT_COREx_BT_BB_NMI_MAP_REG	6	
7	RWBT_IRQ	INTERRUPT_COREx_RWBT_IRQ_MAP_REG	7	
8	RWBLE_IRQ	INTERRUPT_COREx_RWBLE_IRQ_MAP_REG	8	
9	RWBT_NMI	INTERRUPT_COREx_RWBT_NMI_MAP_REG	9	
10	RWBLE_NMI	INTERRUPT_COREx_RWBLE_NMI_MAP_REG	10	
11	I2C_MST_INT	INTERRUPT_COREx_I2C_MST_INT_MAP_REG	11	
12	保留	保留	12	
13	保留	保留	13	
14	UHCI0_INTR	INTERRUPT_COREx_UHCI0_INTR_MAP_REG	14	
15	保留	保留	15	
16	GPIO_INTERRUPT_PRO	INTERRUPT_COREx_GPIO_INTERRUPT_PRO_MAP_REG	16	
17	GPIO_INTERRUPT_PRO_NMI	INTERRUPT_COREx_GPIO_INTERRUPT_PRO_NMI_MAP_REG	17	
18	保留	保留	18	
19	保留	保留	19	
20	SPI_INTR_1	INTERRUPT_COREx_SPI_INTR_1_MAP_REG	20	
21	SPI_INTR_2	INTERRUPT_COREx_SPI_INTR_2_MAP_REG	21	
22	SPI_INTR_3	INTERRUPT_COREx_SPI_INTR_3_MAP_REG	22	
23	保留	保留	23	
24	LCD_CAM_INT	INTERRUPT_COREx_LCD_CAM_INT_MAP_REG	24	
25	I2S0_INT	INTERRUPT_COREx_I2S0_INT_MAP_REG	25	
26	I2S1_INT	INTERRUPT_COREx_I2S1_INT_MAP_REG	26	
27	UART_INTR	INTERRUPT_COREx_UART_INTR_MAP_REG	27	
28	UART1_INTR	INTERRUPT_COREx_UART1_INTR_MAP_REG	28	
29	UART2_INTR	INTERRUPT_COREx_UART2_INTR_MAP_REG	29	
30	SDIO_HOST_INTERRUPT	INTERRUPT_COREx_SDIO_HOST_INTERRUPT_MAP_REG	30	
31	PWM0_INTR	INTERRUPT_COREx_PWM0_INTR_MAP_REG	31	
32	PWM1_INTR	INTERRUPT_COREx_PWM1_INTR_MAP_REG	0	INTERRUPT_COREx_INTR_STATUS_1_REG
33	保留	保留	1	
34	保留	保留	2	

No.	中断源	配置寄存器	状态寄存器	
			位	名称
35	LEDC_INT	INTERRUPT_COREx_LEDC_INT_MAP_REG	3	
36	EFUSE_INT	INTERRUPT_COREx_EFUSE_INT_MAP_REG	4	
37	CAN_INT	INTERRUPT_COREx_CAN_INT_MAP_REG	5	
38	USB_INTR	INTERRUPT_COREx_USB_INTR_MAP_REG	6	
39	RTC_CORE_INTR	INTERRUPT_COREx_RTC_CORE_INTR_MAP_REG	7	
40	RMT_INTR	INTERRUPT_COREx_RMT_INTR_MAP_REG	8	
41	PCNT_INTR	INTERRUPT_COREx_PCNT_INTR_MAP_REG	9	
42	I2C_EXT0_INTR	INTERRUPT_COREx_I2C_EXT0_INTR_MAP_REG	10	
43	I2C_EXT1_INTR	INTERRUPT_COREx_I2C_EXT1_INTR_MAP_REG	11	
44	保留	保留	12	
45	保留	保留	13	
46	保留	保留	14	
47	保留	保留	15	
48	保留	保留	16	
49	保留	保留	17	
50	TG_T0_INT	INTERRUPT_COREx_TG_T0_INT_MAP_REG	18	
51	TG_T1_INT	INTERRUPT_COREx_TG_T1_INT_MAP_REG	19	INTERRUPT_COREx_INTR_STATUS_1_REG
52	TG_WDT_INT	INTERRUPT_COREx_TG_WDT_INT_MAP_REG	20	
53	TG1_T0_INT	INTERRUPT_COREx_TG1_T0_INT_MAP_REG	21	
54	TG1_T1_INT	INTERRUPT_COREx_TG1_T1_INT_MAP_REG	22	
55	TG1_WDT_INT	INTERRUPT_COREx_TG1_WDT_INT_MAP_REG	23	
56	CACHE_IA_INT	INTERRUPT_COREx_CACHE_IA_INT_MAP_REG	24	
57	SYSTIMER_TARGET0_INT	INTERRUPT_COREx_SYSTIMER_TARGET0_INT_MAP_REG	25	
58	SYSTIMER_TARGET1_INT	INTERRUPT_COREx_SYSTIMER_TARGET1_INT_MAP_REG	26	
59	SYSTIMER_TARGET2_INT	INTERRUPT_COREx_SYSTIMER_TARGET2_INT_MAP_REG	27	
60	SPI_MEM_REJECT_INTR	INTERRUPT_COREx_SPI_MEM_REJECT_INTR_MAP_REG	28	
61	DCACHE_PRELOAD_INT	INTERRUPT_COREx_DCACHE_PRELOAD_INT_MAP_REG	29	
62	ICACHE_PRELOAD_INT	INTERRUPT_COREx_ICACHE_PRELOAD_INT_MAP_REG	30	
63	DCACHE_SYNC_INT	INTERRUPT_COREx_DCACHE_SYNC_INT_MAP_REG	31	
64	ICACHE_SYNC_INT	INTERRUPT_COREx_ICACHE_SYNC_INT_MAP_REG	0	
65	APB_ADC_INT	INTERRUPT_COREX_APB_ADC_INT_MAP_REG	1	
66	DMA_IN_CH0_INT	INTERRUPT_COREX_DMA_IN_CH0_INT_MAP_REG	2	
67	DMA_IN_CH1_INT	INTERRUPT_COREx_DMA_IN_CH1_INT_MAP_REG	3	
68	DMA_IN_CH2_INT	INTERRUPT_COREx_DMA_IN_CH2_INT_MAP_REG	4	INTERRUPT_COREx_INTR_STATUS_2_REG
69	DMA_IN_CH3_INT	INTERRUPT_COREx_DMA_IN_CH3_INT_MAP_REG	5	
70	DMA_IN_CH4_INT	INTERRUPT_COREx_DMA_IN_CH4_INT_MAP_REG	6	
71	DMA_OUT_CH0_INT	INTERRUPT_COREX_DMA_OUT_CH0_INT_MAP_REG	7	

No.	中断源	配置寄存器	状态寄存器 位	状态寄存器 名称
72	DMA_OUT_CH1_INT	INTERRUPT_COREx_DMA_OUT_CH1_INT_MAP_REG	8	
73	DMA_OUT_CH2_INT	INTERRUPT_COREx_DMA_OUT_CH2_INT_MAP_REG	9	
74	DMA_OUT_CH3_INT	INTERRUPT_COREx_DMA_OUT_CH3_INT_MAP_REG	10	
75	DMA_OUT_CH4_INT	INTERRUPT_COREx_DMA_OUT_CH4_INT_MAP_REG	11	
76	RSA_INTR	INTERRUPT_COREx_RSA_INTR_MAP_REG	12	
77	AES_INTR	INTERRUPT_COREx_AES_INTR_MAP_REG	13	
78	SHA_INTR	INTERRUPT_COREx_SHA_INTR_MAP_REG	14	
79	CPU_INTR_FROM_CPU_0	INTERRUPT_COREx_CPU_INTR_FROM_CPU_0_MAP_REG	15	
80	CPU_INTR_FROM_CPU_1	INTERRUPT_COREx_CPU_INTR_FROM_CPU_1_MAP_REG	16	
81	CPU_INTR_FROM_CPU_2	INTERRUPT_COREx_CPU_INTR_FROM_CPU_2_MAP_REG	17	
82	CPU_INTR_FROM_CPU_3	INTERRUPT_COREx_CPU_INTR_FROM_CPU_3_MAP_REG	18	
83	ASSIST_DEBUG_INTR	INTERRUPT_COREx_ASSIST_DEBUG_INTR_MAP_REG	19	INTERRUPT_COREx_INTR_STATUS_2_REG
84	DMA_APB_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_DMA_APB_PMS_MONITOR_VIOLATE_INTR_MAP_REG	20	
85	CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	21	
86	CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	22	
87	CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	23	
88	CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR	INTERRUPT_COREx_CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	24	
89	CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	25	
90	CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	26	
91	CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR	INTERRUPT_COREx_CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	27	
92	CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR	INTERRUPT_COREx_CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	28	
93	BACKUP_PMS_VIOLATE_INT	INTERRUPT_COREx_BACKUP_PMS_VIOLATE_INTR_MAP_REG	29	
94	CACHE_CORE0_ACS_INT	INTERRUPT_COREx_CACHE_CORE0_ACS_INT_MAP_REG	30	
95	CACHE_CORE1_ACS_INT	INTERRUPT_COREx_CACHE_CORE1_ACS_INT_MAP_REG	31	
96	USB_DEVICE_INT	INTERRUPT_COREx_USB_DEVICE_INT_MAP_REG	0	
97	PERI_BACKUP_INT	INTERRUPT_COREx_PERI_BACKUP_INT_MAP_REG	1	INTERRUPT_COREx_INTR_STATUS_3_REG
98	DMA_EXTMEM_REJECT_INT	INTERRUPT_COREx_DMA_EXTMEM_REJECT_INT_MAP_REG	2	

5.3.2 CPU 中断

每个 CPU 都有 32 个中断号 (0 ~ 31)，其中包括 26 个外部中断，6 个内部中断。

- 外部中断为外部中断源引发的中断，包括下面三种类型：
  - 电平触发类型中断：高电平触发，要求保持中断的电平状态直到 CPU*x* 响应；
  - 边沿触发类型中断：上升沿触发，此中断一旦产生，CPU*x* 即可响应；
  - NMI 中断：软件不可使用 CPU*x* 内部特殊寄存器屏蔽此类中断，World Controller 模块提供了屏蔽此中断的机制。更多信息，见章节 World Controller。
- 内部中断为 CPU*x* 内部自己产生的中断，包括下面三种类型：
  - 定时器中断：由内部定时器触发，可用于产生周期性的中断；
  - 软件中断：软件写特殊寄存器时将触发此中断；
  - 解析中断：用于性能监测和分析。

上述电平类型和边沿类型中断指 CPU 接收中断信号的方式。对于电平类型中断，在 CPU 响应此中断之前该中断信号的电平需要一直保持，如果电平提前掉下去 CPU 会丢失此次中断。对于边沿类型中断，CPU 会检测中断信号的边沿，当检测到边沿之后 CPU 会记录此次中断，此时中断信号可以提前拉低。

通过中断矩阵将外部中断源映射到任意一个外部中断，即可让 CPU 接收到中断源的中断信号。表 5-2 列出了所有的中断号对应的类型和优先级。

ESP32-S3 支持六级中断，同时支持中断嵌套，即低优先级中断可以被高优先级中断打断。在下表优先级一栏中，数字越大代表优先级越高。其中，NMI 中断拥有最高优先级，此类中断一经触发，CPU 必须处理。

表 5-2. CPU 中断

中断号	类别	种类	优先级
0	外部中断	电平触发	1
1	外部中断	电平触发	1
2	外部中断	电平触发	1
3	外部中断	电平触发	1
4	外部中断	电平触发	1
5	外部中断	电平触发	1
6	内部中断	定时器 0	1
7	内部中断	软件	1
8	外部中断	电平触发	1
9	外部中断	电平触发	1
10	外部中断	边沿触发	1
11	内部中断	解析	3
12	外部中断	电平触发	1
13	外部中断	电平触发	1
14	外部中断	NMI	NMI
15	内部中断	定时器 1	3
16	内部中断	定时器 2	5
17	外部中断	电平触发	1
18	外部中断	电平触发	1

中断号	类别	种类	优先级
19	外部中断	电平触发	2
20	外部中断	电平触发	2
21	外部中断	电平触发	2
22	外部中断	边沿触发	3
23	外部中断	电平触发	3
24	外部中断	电平触发	4
25	外部中断	电平触发	4
26	外部中断	电平触发	5
27	外部中断	电平触发	3
28	外部中断	边沿触发	4
29	内部中断	软件	3
30	外部中断	边沿触发	4
31	外部中断	电平触发	5

5.3.3 分配外部中断源至 CPU*x* 外部中断

在本小节中，我们将使用以下术语描述中断矩阵相关操作：

- Source_*Y*：代表某个外部中断源，其中 Y 为中断源编号，详见表 5-1。
- INTERRUPT_CORE*x*_SOURCE_*Y*_MAP_REG：CPU*x* 外部中断源 (Source_*Y*) 的中断配置寄存器。
- Interrupt_P：表示中断号为 Num_P 的外部中断，Num_P 的取值范围为 0 ~ 5、8 ~ 10、12 ~ 14、17 ~ 28、30 ~ 31，详见表 5-2。
- Interrupt_I：表示中断号为 Num_I 的内部中断，Num_I 的取值范围为 6、7、11、15、16、29，详见表 5-2。

5.3.3.1 分配一个外部中断源 Source_Y 至 CPU*x* 外部中断

将外部中断源 Source_Y 对应的寄存器 INTERRUPT_CORE*x*_SOURCE_*Y*_MAP_REG 配成 Num_P，即可将该中断源分配至序号为 Num_P 的外部中断 (Interrupt_P)。Num_P 可以取 CPU*x* 的任一外部中断号，包括 0 ~ 5、8 ~ 10、12 ~ 14、17 ~ 28、30 ~ 31。每个 CPU 中断可被多个外设共享。

5.3.3.2 分配多个外部中断源 Source_Y*n* 至 CPU*x* 外部中断

将各个中断源对应的寄存器 INTERRUPT_CORE*x*_SOURCE_Y*n*_MAP_REG 均配置成相同的 Num_P，即可将多个中断源 Source_Y*n* 分配至同一 CPU*x* 外部中断 Interrupt_P。上述任一外设中断均会触发 CPU*x* 外部中断 Interrupt_P。待中断触发后，CPU*x* 需查询中断状态寄存器，判断产生中断的外设。

5.3.3.3 关闭 CPU*x* 外部中断源 Source_Y

将中断源对应的寄存器 INTERRUPT_CORE*x*_SOURCE_*Y*_MAP_REG 配置成任意 Num_I，即可关闭外部中断源。这是因为任何被配置成 Num_I 的外部中断均无法连接至 CPU*x*，而且选择任一内部中断号 (6、7、11、15、16、29) 不会造成其他影响，可用于关闭外部中断。

5.3.4 关闭 CPU*x* 的 NMI 类型中断

表 5-2 中的 32 个中断，除 NMI 类型中断，其余中断均可通过软件配置 CPU 特殊寄存器 (INTENABLE) 进行屏蔽和使能。ESP32-S3 另外提供两种屏蔽 NMI 的机制：

- 断开外部中断源与 NMI 中断的连接，即将所有分配给 NMI 中断的外部中断源分配给其它中断；
- 不断开外部中断源与 NMI 中断的连接，但使用 World Controller 模块的 NMI 屏蔽功能进行屏蔽。更多信息，请参考 World Controller 章节。

5.3.5 查询外部中断源当前的中断状态

读取寄存器 INTERRUPT_CORE*x*_INTR_STATUS_*n*_REG（只读）中特定位的值可以获取 CPU*x* 外部中断源当前的中断状态。寄存器 INTERRUPT_CORE*x*_INTR_STATUS_*n*_REG 与外部中断源的对应关系如表 5-1 所示。

5.4 寄存器列表

本小节的所有地址均为相对于中断矩阵基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

5.4.1 CPU0 中断寄存器列表

名称	描述	地址	访问
配置寄存器			
INTERRUPT_CORE0_MAC_INTR_MAP_REG	MAC 中断配置寄存器	0x0000	R/W
INTERRUPT_CORE0_MAC_NMI_MAP_REG	MAC_NMI 中断配置寄存器	0x0004	R/W
INTERRUPT_CORE0_PWR_INTR_MAP_REG	PWR 中断配置寄存器	0x0008	R/W
INTERRUPT_CORE0_BB_INT_MAP_REG	BB 中断配置寄存器	0x000C	R/W
INTERRUPT_CORE0_BT_MAC_INT_MAP_REG	BB_MAC 中断配置寄存器	0x0010	R/W
INTERRUPT_CORE0_BT_BB_INT_MAP_REG	BT_BB 中断配置寄存器	0x0014	R/W
INTERRUPT_CORE0_BT_BB_NMI_MAP_REG	BT_BB_NMI 中断配置寄存器	0x0018	R/W
INTERRUPT_CORE0_RWBT_IRQ_MAP_REG	RWBT_IRQ 中断配置寄存器	0x001C	R/W
INTERRUPT_CORE0_RWBLE_IRQ_MAP_REG	RWBLE_IRQ 中断配置寄存器	0x0020	R/W
INTERRUPT_CORE0_RWBT_NMI_MAP_REG	RWBT_NMI 中断配置寄存器	0x0024	R/W
INTERRUPT_CORE0_RWBLE_NMI_MAP_REG	RWBLE_NMI 中断配置寄存器	0x0028	R/W
INTERRUPT_CORE0_I2C_MST_INT_MAP_REG	I2C_MST 中断配置寄存器	0x002C	R/W
INTERRUPT_CORE0_UHCI0_INTR_MAP_REG	UHCI0 中断配置寄存器	0x0038	R/W
INTERRUPT_CORE0_GPIO_INTERRUPT_PRO_MAP_REG	GPIO_INTERRUPT_PRO 中断配置寄存器	0x0040	R/W
INTERRUPT_CORE0_GPIO_INTERRUPT_PRO_NMI_MAP_REG	GPIO_INTERRUPT_PRO_NMI 中断配置寄存器	0x0044	R/W
INTERRUPT_CORE0_SPI_INTR_1_MAP_REG	SPI_INTR_1 中断配置寄存器	0x0050	R/W
INTERRUPT_CORE0_SPI_INTR_2_MAP_REG	SPI_INTR_2 中断配置寄存器	0x0054	R/W
INTERRUPT_CORE0_SPI_INTR_3_MAP_REG	SPI_INTR_3 中断配置寄存器	0x0058	R/W
INTERRUPT_CORE0_LCD_CAM_INT_MAP_REG	LCD_CAM 中断配置寄存器	0x0060	R/W
INTERRUPT_CORE0_I2S0_INT_MAP_REG	I2S0 中断配置寄存器	0x0064	R/W
INTERRUPT_CORE0_I2S1_INT_MAP_REG	I2S1 中断配置寄存器	0x0068	R/W
INTERRUPT_CORE0_UART_INTR_MAP_REG	UART 中断配置寄存器	0x006C	R/W
INTERRUPT_CORE0_UART1_INTR_MAP_REG	UART1 中断配置寄存器	0x0070	R/W
INTERRUPT_CORE0_UART2_INTR_MAP_REG	UART2 中断配置寄存器	0x0074	R/W
INTERRUPT_CORE0_SDIO_HOST_INTERRUPT_MAP_REG	SDIO_HOST 中断配置寄存器	0x0078	R/W
INTERRUPT_CORE0_PWM0_INTR_MAP_REG	PWM0 中断配置寄存器	0x007C	R/W

名称	描述	地址	访问
INTERRUPT_CORE0_PWM1_INTR_MAP_REG	PWM1 中断配置寄存器	0x0080	R/W
INTERRUPT_CORE0_LEDC_INT_MAP_REG	LEDC 中断配置寄存器	0x008C	R/W
INTERRUPT_CORE0_EFUSE_INT_MAP_REG	EFUSE 中断配置寄存器	0x0090	R/W
INTERRUPT_CORE0_CAN_INT_MAP_REG	CAN 中断配置寄存器	0x0094	R/W
INTERRUPT_CORE0_USB_INTR_MAP_REG	USB 中断配置寄存器	0x0098	R/W
INTERRUPT_CORE0_RTC_CORE_INTR_MAP_REG	RTC_CORE 中断配置寄存器	0x009C	R/W
INTERRUPT_CORE0_RMT_INTR_MAP_REG	RMT 中断配置寄存器	0x00A0	R/W
INTERRUPT_CORE0_PCNT_INTR_MAP_REG	PCNT 中断配置寄存器	0x00A4	R/W
INTERRUPT_CORE0_I2C_EXT0_INTR_MAP_REG	I2C_EXT0 中断配置寄存器	0x00A8	R/W
INTERRUPT_CORE0_I2C_EXT1_INTR_MAP_REG	I2C_EXT1 中断配置寄存器	0x00AC	R/W
INTERRUPT_CORE0_TG_T0_INT_MAP_REG	TG_T0 中断配置寄存器	0x00C8	R/W
INTERRUPT_CORE0_TG_T1_INT_MAP_REG	TG_T1 中断配置寄存器	0x00CC	R/W
INTERRUPT_CORE0_TG_WDT_INT_MAP_REG	TG_WDT 中断配置寄存器	0x00D0	R/W
INTERRUPT_CORE0_TG1_T0_INT_MAP_REG	TG1_T0 中断配置寄存器	0x00D4	R/W
INTERRUPT_CORE0_TG1_T1_INT_MAP_REG	TG1_T1 中断配置寄存器	0x00D8	R/W
INTERRUPT_CORE0_TG1_WDT_INT_MAP_REG	TG1_WDT 中断配置寄存器	0x00DC	R/W
INTERRUPT_CORE0_CACHE_IA_INT_MAP_REG	CACHE_IA 中断配置寄存器	0x00E0	R/W
INTERRUPT_CORE0_SYSTIMER_TARGET0_INT_MAP_REG	SYSTIMER_TARGET0 中断配置寄存器	0x00E4	R/W
INTERRUPT_CORE0_SYSTIMER_TARGET1_INT_MAP_REG	SYSTIMER_TARGET1 中断配置寄存器	0x00E8	R/W
INTERRUPT_CORE0_SYSTIMER_TARGET2_INT_MAP_REG	SYSTIMER_TARGET2 中断配置寄存器	0x00EC	R/W
INTERRUPT_CORE0_SPI_MEM_REJECT_INTR_MAP_REG	SPI_MEM_REJECT 中断配置寄存器	0x00F0	R/W
INTERRUPT_CORE0_DCACHE_PRELOAD_INT_MAP_REG	DCACHE_PRELAOD 中断配置寄存器	0x00F4	R/W
INTERRUPT_CORE0_ICACHE_PRELOAD_INT_MAP_REG	ICACHE_PRELOAD 中断配置寄存器	0x00F8	R/W
INTERRUPT_CORE0_DCACHE_SYNC_INT_MAP_REG	DCACHE_SYNC 中断配置寄存器	0x00FC	R/W
INTERRUPT_CORE0_ICACHE_SYNC_INT_MAP_REG	ICACHE_SYNC 中断配置寄存器	0x0100	R/W
INTERRUPT_CORE0_APB_ADC_INT_MAP_REG	APB_ADC 中断配置寄存器	0x0104	R/W
INTERRUPT_CORE0_DMA_IN_CH0_INT_MAP_REG	DMA_IN_CH0 中断配置寄存器	0x0108	R/W
INTERRUPT_CORE0_DMA_IN_CH1_INT_MAP_REG	DMA_IN_CH1 中断配置寄存器	0x010C	R/W
INTERRUPT_CORE0_DMA_IN_CH2_INT_MAP_REG	DMA_IN_CH2 中断配置寄存器	0x0110	R/W

名称	描述	地址	访问
INTERRUPT_CORE0_DMA_IN_CH3_INT_MAP_REG	DMA_IN_CH3 中断配置寄存器	0x0114	R/W
INTERRUPT_CORE0_DMA_IN_CH4_INT_MAP_REG	DMA_IN_CH4 中断配置寄存器	0x0118	R/W
INTERRUPT_CORE0_DMA_OUT_CH0_INT_MAP_REG	DMA_OUT_CH0 中断配置寄存器	0x011C	R/W
INTERRUPT_CORE0_DMA_OUT_CH1_INT_MAP_REG	DMA_OUT_CH1 中断配置寄存器	0x0120	R/W
INTERRUPT_CORE0_DMA_OUT_CH2_INT_MAP_REG	DMA_OUT_CH2 中断配置寄存器	0x0124	R/W
INTERRUPT_CORE0_DMA_OUT_CH3_INT_MAP_REG	DMA_OUT_CH3 中断配置寄存器	0x0128	R/W
INTERRUPT_CORE0_DMA_OUT_CH4_INT_MAP_REG	DMA_OUT_CH4 中断配置寄存器	0x012C	R/W
INTERRUPT_CORE0_RSA_INT_MAP_REG	RSA 中断配置寄存器	0x0130	R/W
INTERRUPT_CORE0_AES_INT_MAP_REG	AES 中断配置寄存器	0x0134	R/W
INTERRUPT_CORE0_SHA_INT_MAP_REG	SHA 中断配置寄存器	0x0138	R/W
INTERRUPT_CORE0_CPU_INTR_FROM_CPU_0_MAP_REG	CPU_INTR_FROM_CPU_0 中断配置寄存器	0x013C	R/W
INTERRUPT_CORE0_CPU_INTR_FROM_CPU_1_MAP_REG	CPU_INTR_FROM_CPU_1 中断配置寄存器	0x0140	R/W
INTERRUPT_CORE0_CPU_INTR_FROM_CPU_2_MAP_REG	CPU_INTR_FROM_CPU_2 中断配置寄存器	0x0144	R/W
INTERRUPT_CORE0_CPU_INTR_FROM_CPU_3_MAP_REG	CPU_INTR_FROM_CPU_3 中断配置寄存器	0x0148	R/W
INTERRUPT_CORE0_ASSIST_DEBUG_INTR_MAP_REG	ASSIST_DEBUG 中断配置寄存器	0x014C	R/W
INTERRUPT_CORE0_DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR_MAP_REG	dma_pms_monitor_violatile 中断配置寄存器	0x0150	R/W
INTERRUPT_CORE0_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_IRam0_pms_monitor_violatile 中断配置寄存器	0x0154	R/W
INTERRUPT_CORE0_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_DRam0_pms_monitor_violatile 中断配置寄存器	0x0158	R/W
INTERRUPT_CORE0_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_PIF_pms_monitor_violatile 中断配置寄存器	0x015C	R/W
INTERRUPT_CORE0_CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	core0_PIF_pms_monitor_violatile_size 中断配置寄存器	0x0160	R/W
INTERRUPT_CORE0_CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_IRam0_pms_monitor_violatile 中断配置寄存器	0x0164	R/W
INTERRUPT_CORE0_CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_DRam0_pms_monitor_violatile 中断配置寄存器	0x0168	R/W

名称	描述	地址	访问
INTERRUPT_CORE0_CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_PIF_pms_monitor_violatile 中断配置寄存器	0x016C	R/W
INTERRUPT_CORE0_CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	core1_PIF_pms_monitor_violatile_size 中断配置寄存器	0x0170	R/W
INTERRUPT_CORE0_BACKUP_PMS_VIOLATE_INTR_MAP_REG	BACKUP_PMS_MONITOR_VIOLATILE 中断配置寄存器	0x0174	R/W
INTERRUPT_CORE0_CACHE_CORE0_ACS_INT_MAP_REG	CACHE_CORE0_ACS 中断配置寄存器	0x0178	R/W
INTERRUPT_CORE0_CACHE_CORE1_ACS_INT_MAP_REG	CACHE_CORE1_ACS 中断配置寄存器	0x017C	R/W
INTERRUPT_CORE0_USB_DEVICE_INT_MAP_REG	USB_DEVICE 中断配置寄存器	0x0180	R/W
INTERRUPT_CORE0_PERI_BACKUP_INT_MAP_REG	PERI_BACKUP 中断配置寄存器	0x0184	R/W
INTERRUPT_CORE0_DMA_EXTMEM_REJECT_INT_MAP_REG	DMA_EXTMEM_REJECT 中断配置寄存器	0x0188	R/W
状态寄存器			
INTERRUPT_CORE0_INTR_STATUS_0_REG	中断状态寄存器 0	0x018C	RO
INTERRUPT_CORE0_INTR_STATUS_1_REG	中断状态寄存器 1	0x0190	RO
INTERRUPT_CORE0_INTR_STATUS_2_REG	中断状态寄存器 2	0x0194	RO
INTERRUPT_CORE0_INTR_STATUS_3_REG	中断状态寄存器 3	0x0198	RO
时钟寄存器			
INTERRUPT_CORE0_CLOCK_GATE_REG	时钟门控寄存器	0x019C	R/W
版本寄存器			
INTERRUPT_CORE0_DATE_REG	版本控制寄存器	0x07FC	R/W

5.4.2 CPU1 中断寄存器列表

名称	描述	地址	访问
配置寄存器			
INTERRUPT_CORE1_MAC_INTR_MAP_REG	MAC 中断配置寄存器	0x0800	R/W
INTERRUPT_CORE1_MAC_NMI_MAP_REG	MAC_NMI 中断配置寄存器	0x0804	R/W
INTERRUPT_CORE1_PWR_INTR_MAP_REG	PWR 中断配置寄存器	0x0808	R/W
INTERRUPT_CORE1_BB_INT_MAP_REG	BB 中断配置寄存器	0x080C	R/W
INTERRUPT_CORE1_BT_MAC_INT_MAP_REG	BB_MAC 中断配置寄存器	0x0810	R/W

名称	描述	地址	访问
INTERRUPT_CORE1_BT_BB_INT_MAP_REG	BT_BB 中断配置寄存器	0x0814	R/W
INTERRUPT_CORE1_BT_BB_NMI_MAP_REG	BT_BB_NMI 中断配置寄存器	0x0818	R/W
INTERRUPT_CORE1_RWBT_IRQ_MAP_REG	RWBT_IRQ 中断配置寄存器	0x081C	R/W
INTERRUPT_CORE1_RWBLE_IRQ_MAP_REG	RWBLE_IRQ 中断配置寄存器	0x0820	R/W
INTERRUPT_CORE1_RWBT_NMI_MAP_REG	RWBT_NMI 中断配置寄存器	0x0824	R/W
INTERRUPT_CORE1_RWBLE_NMI_MAP_REG	RWBLE_NMI 中断配置寄存器	0x0828	R/W
INTERRUPT_CORE1_I2C_MST_INT_MAP_REG	I2C_MST 中断配置寄存器	0x082C	R/W
INTERRUPT_CORE1_UHCI0_INTR_MAP_REG	UHCI0 中断配置寄存器	0x0838	R/W
INTERRUPT_CORE1_GPIO_INTERRUPT_PRO_MAP_REG	GPIO_INTERRUPT_PRO 中断配置寄存器	0x0840	R/W
INTERRUPT_CORE1_GPIO_INTERRUPT_PRO_NMI_MAP_REG	GPIO_INTERRUPT_PRO_NMI 中断配置寄存器	0x0844	R/W
INTERRUPT_CORE1_SPI_INTR_1_MAP_REG	SPI_INTR_1 中断配置寄存器	0x0850	R/W
INTERRUPT_CORE1_SPI_INTR_2_MAP_REG	SPI_INTR_2 中断配置寄存器	0x0854	R/W
INTERRUPT_CORE1_SPI_INTR_3_MAP_REG	SPI_INTR_3 中断配置寄存器	0x0858	R/W
INTERRUPT_CORE1_LCD_CAM_INT_MAP_REG	LCD_CAM 中断配置寄存器	0x0860	R/W
INTERRUPT_CORE1_I2S0_INT_MAP_REG	I2S0 中断配置寄存器	0x0864	R/W
INTERRUPT_CORE1_I2S1_INT_MAP_REG	I2S1 中断配置寄存器	0x0868	R/W
INTERRUPT_CORE1_UART_INTR_MAP_REG	UART 中断配置寄存器	0x086C	R/W
INTERRUPT_CORE1_UART1_INTR_MAP_REG	UART1 中断配置寄存器	0x0870	R/W
INTERRUPT_CORE1_UART2_INTR_MAP_REG	UART2 中断配置寄存器	0x0874	R/W
INTERRUPT_CORE1_SDIO_HOST_INTERRUPT_MAP_REG	SDIO_HOST 中断配置寄存器	0x0878	R/W
INTERRUPT_CORE1_PWM0_INTR_MAP_REG	PWM0 中断配置寄存器	0x087C	R/W
INTERRUPT_CORE1_PWM1_INTR_MAP_REG	PWM1 中断配置寄存器	0x0880	R/W
INTERRUPT_CORE1_LEDC_INT_MAP_REG	LEDC 中断配置寄存器	0x088C	R/W
INTERRUPT_CORE1_EFUSE_INT_MAP_REG	EFUSE 中断配置寄存器	0x0890	R/W
INTERRUPT_CORE1_CAN_INT_MAP_REG	CAN 中断配置寄存器	0x0894	R/W
INTERRUPT_CORE1_USB_INTR_MAP_REG	USB 中断配置寄存器	0x0898	R/W
INTERRUPT_CORE1_RTC_CORE_INTR_MAP_REG	RTC_CORE 中断配置寄存器	0x089C	R/W
INTERRUPT_CORE1_RMT_INTR_MAP_REG	RMT 中断配置寄存器	0x08A0	R/W
INTERRUPT_CORE1_PCNT_INTR_MAP_REG	PCNT 中断配置寄存器	0x08A4	R/W

名称	描述	地址	访问
INTERRUPT_CORE1_I2C_EXT0_INTR_MAP_REG	I2C_EXT0 中断配置寄存器	0x08A8	R/W
INTERRUPT_CORE1_I2C_EXT1_INTR_MAP_REG	I2C_EXT1 中断配置寄存器	0x08AC	R/W
INTERRUPT_CORE1_TG_T0_INT_MAP_REG	TG_T0 中断配置寄存器	0x08C8	R/W
INTERRUPT_CORE1_TG_T1_INT_MAP_REG	TG_T1 中断配置寄存器	0x08CC	R/W
INTERRUPT_CORE1_TG_WDT_INT_MAP_REG	TG_WDT 中断配置寄存器	0x08D0	R/W
INTERRUPT_CORE1_TG1_T0_INT_MAP_REG	TG1_T0 中断配置寄存器	0x08D4	R/W
INTERRUPT_CORE1_TG1_T1_INT_MAP_REG	TG1_T1 中断配置寄存器	0x08D8	R/W
INTERRUPT_CORE1_TG1_WDT_INT_MAP_REG	TG1_WDT 中断配置寄存器	0x08DC	R/W
INTERRUPT_CORE1_CACHE_IA_INT_MAP_REG	CACHE_IA 中断配置寄存器	0x08E0	R/W
INTERRUPT_CORE1_SYSTIMER_TARGET0_INT_MAP_REG	SYSTIMER_TARGET0 中断配置寄存器	0x08E4	R/W
INTERRUPT_CORE1_SYSTIMER_TARGET1_INT_MAP_REG	SYSTIMER_TARGET1 中断配置寄存器	0x08E8	R/W
INTERRUPT_CORE1_SYSTIMER_TARGET2_INT_MAP_REG	SYSTIMER_TARGET2 中断配置寄存器	0x08EC	R/W
INTERRUPT_CORE1_SPI_MEM_REJECT_INTR_MAP_REG	SPI_MEM_REJECT 中断配置寄存器	0x08F0	R/W
INTERRUPT_CORE1_DCACHE_PRELOAD_INT_MAP_REG	DCACHE_PRELAOD 中断配置寄存器	0x08F4	R/W
INTERRUPT_CORE1_ICACHE_PRELOAD_INT_MAP_REG	ICACHE_PRELOAD 中断配置寄存器	0x08F8	R/W
INTERRUPT_CORE1_DCACHE_SYNC_INT_MAP_REG	DCACHE_SYNC 中断配置寄存器	0x08FC	R/W
INTERRUPT_CORE1_ICACHE_SYNC_INT_MAP_REG	ICACHE_SYNC 中断配置寄存器	0x0900	R/W
INTERRUPT_CORE1_APB_ADC_INT_MAP_REG	APB_ADC 中断配置寄存器	0x0904	R/W
INTERRUPT_CORE1_DMA_IN_CH0_INT_MAP_REG	DMA_IN_CH0 中断配置寄存器	0x0908	R/W
INTERRUPT_CORE1_DMA_IN_CH1_INT_MAP_REG	DMA_IN_CH1 中断配置寄存器	0x090C	R/W
INTERRUPT_CORE1_DMA_IN_CH2_INT_MAP_REG	DMA_IN_CH2 中断配置寄存器	0x0910	R/W
INTERRUPT_CORE1_DMA_IN_CH3_INT_MAP_REG	DMA_IN_CH3 中断配置寄存器	0x0914	R/W
INTERRUPT_CORE1_DMA_IN_CH4_INT_MAP_REG	DMA_IN_CH4 中断配置寄存器	0x0918	R/W
INTERRUPT_CORE1_DMA_OUT_CH0_INT_MAP_REG	DMA_OUT_CH0 中断配置寄存器	0x091C	R/W
INTERRUPT_CORE1_DMA_OUT_CH1_INT_MAP_REG	DMA_OUT_CH1 中断配置寄存器	0x0920	R/W
INTERRUPT_CORE1_DMA_OUT_CH2_INT_MAP_REG	DMA_OUT_CH2 中断配置寄存器	0x0924	R/W
INTERRUPT_CORE1_DMA_OUT_CH3_INT_MAP_REG	DMA_OUT_CH3 中断配置寄存器	0x0928	R/W
INTERRUPT_CORE1_DMA_OUT_CH4_INT_MAP_REG	DMA_OUT_CH4 中断配置寄存器	0x092C	R/W
INTERRUPT_CORE1_RSA_INT_MAP_REG	RSA 中断配置寄存器	0x0930	R/W

名称	描述	地址	访问
INTERRUPT_CORE1_AES_INT_MAP_REG	AES 中断配置寄存器	0x0934	R/W
INTERRUPT_CORE1_SHA_INT_MAP_REG	SHA 中断配置寄存器	0x0938	R/W
INTERRUPT_CORE1_CPU_INTR_FROM_CPU_0_MAP_REG	CPU_INTR_FROM_CPU_0 中断配置寄存器	0x093C	R/W
INTERRUPT_CORE1_CPU_INTR_FROM_CPU_1_MAP_REG	CPU_INTR_FROM_CPU_1 中断配置寄存器	0x0940	R/W
INTERRUPT_CORE1_CPU_INTR_FROM_CPU_2_MAP_REG	CPU_INTR_FROM_CPU_2 中断配置寄存器	0x0944	R/W
INTERRUPT_CORE1_CPU_INTR_FROM_CPU_3_MAP_REG	CPU_INTR_FROM_CPU_3 中断配置寄存器	0x0948	R/W
INTERRUPT_CORE1_ASSIST_DEBUG_INTR_MAP_REG	ASSIST_DEBUG 中断配置寄存器	0x094C	R/W
INTERRUPT_CORE1_DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR_MAP_REG	dma_pms_monitor_violatile 中断配置寄存器	0x0950	R/W
INTERRUPT_CORE1_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_IRam0_pms_monitor_violatile 中断配置寄存器	0x0954	R/W
INTERRUPT_CORE1_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_DRam0_pms_monitor_violatile 中断配置寄存器	0x0958	R/W
INTERRUPT_CORE1_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core0_PIF_pms_monitor_violatile 中断配置寄存器	0x095C	R/W
INTERRUPT_CORE1_CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	core0_PIF_pms_monitor_violatile_size 中断配置寄存器	0x0960	R/W
INTERRUPT_CORE1_CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_IRam0_pms_monitor_violatile 中断配置寄存器	0x0964	R/W
INTERRUPT_CORE1_CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_DRam0_pms_monitor_violatile 中断配置寄存器	0x0968	R/W
INTERRUPT_CORE1_CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG	core1_PIF_pms_monitor_violatile 中断配置寄存器	0x096C	R/W
INTERRUPT_CORE1_CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG	core1_PIF_pms_monitor_violatile_size 中断配置寄存器	0x0970	R/W
INTERRUPT_CORE1_BACKUP_PMS_VIOLATE_INTR_MAP_REG	BACKUP_PMS_MONITOR_VIOLATILE 中断配置寄存器	0x0974	R/W
INTERRUPT_CORE1_CACHE_CORE0_ACS_INT_MAP_REG	CACHE_CORE0_ACS 中断配置寄存器 REG	0x0978	R/W
INTERRUPT_CORE1_CACHE_CORE1_ACS_INT_MAP_REG	CACHE_CORE1_ACS 中断配置寄存器 REG	0x097C	R/W
INTERRUPT_CORE1_USB_DEVICE_INT_MAP_REG	USB_DEVICE 中断配置寄存器	0x0980	R/W

名称	描述	地址	访问
INTERRUPT_CORE1_PERI_BACKUP_INT_MAP_REG	PERI_BACKUP 中断配置寄存器	0x0984	R/W
INTERRUPT_CORE1_DMA_EXTMEM_REJECT_INT_MAP_REG	DMA_EXTMEM_REJECT 中断配置寄存器	0x0988	R/W
状态寄存器			
INTERRUPT_CORE1_INTR_STATUS_0_REG	中断状态寄存器 0	0x098C	RO
INTERRUPT_CORE1_INTR_STATUS_1_REG	中断状态寄存器 1	0x0990	RO
INTERRUPT_CORE1_INTR_STATUS_2_REG	中断状态寄存器 2	0x0994	RO
INTERRUPT_CORE1_INTR_STATUS_3_REG	中断状态寄存器 3	0x0998	RO
时钟寄存器			
INTERRUPT_CORE1_CLOCK_GATE_REG	时钟门控寄存器	0x099C	R/W
版本寄存器			
INTERRUPT_CORE1_DATE_REG	版本控制寄存器	0x0FFC	R/W

5.5 寄存器

本小节的所有地址均为相对于中断矩阵基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

5.5.1 CPU0 中断寄存器

Register 5.1. INTERRUPT_CORE0_*MAC_INTR*_MAP_REG (0x0000)

Register 5.2. INTERRUPT_CORE0_*MAC_NMI*_MAP_REG (0x0004)

Register 5.3. INTERRUPT_CORE0_*PWR_INTR*_MAP_REG (0x0008)

Register 5.4. INTERRUPT_CORE0_*BB_INT*_MAP_REG (0x000C)

Register 5.5. INTERRUPT_CORE0_*BT_MAC_INT*_MAP_REG (0x0010)

Register 5.6. INTERRUPT_CORE0_*BT_BB_INT*_MAP_REG (0x0014)

Register 5.7. INTERRUPT_CORE0_*BT_BB_NMI*_MAP_REG (0x0018)

Register 5.8. INTERRUPT_CORE0_*RWBT_IRQ*_MAP_REG (0x001C)

Register 5.9. INTERRUPT_CORE0_*RWBLE_IRQ*_MAP_REG (0x0020)

Register 5.10. INTERRUPT_CORE0_*RWBT_NMI*_MAP_REG (0x0024)

Register 5.11. INTERRUPT_CORE0_*RWBLE_NMI*_MAP_REG (0x0028)

Register 5.12. INTERRUPT_CORE0_*I2C_MST_INT*_MAP_REG (0x002C)

Register 5.13. INTERRUPT_CORE0_*UHCI0_INTR*_MAP_REG (0x0038)

Register 5.14. INTERRUPT_CORE0_*GPIO_INTERRUPT_PRO*_MAP_REG (0x0040)

Register 5.15. INTERRUPT_CORE0_*GPIO_INTERRUPT_PRO_NMI*_MAP_REG (0x0044)

Register 5.16. INTERRUPT_CORE0_*SPI_INTR_1*_MAP_REG (0x0050)

Register 5.17. INTERRUPT_CORE0_*SPI_INTR_2*_MAP_REG (0x0054)

Register 5.18. INTERRUPT_CORE0_*SPI_INTR_3*_MAP_REG (0x0058)

Register 5.19. INTERRUPT_CORE0_*LCD_CAM_INT*_MAP_REG (0x0060)

Register 5.20. INTERRUPT_CORE0_*I2S0_INT*_MAP_REG (0x0064)

Register 5.21. INTERRUPT_CORE0_*I2S1_INT*_MAP_REG (0x0068)

Register 5.22. INTERRUPT_CORE0_*UART_INTR*_MAP_REG (0x006C)

Register 5.23. INTERRUPT_CORE0_*UART1_INTR*_MAP_REG (0x0070)

Register 5.24. INTERRUPT_CORE0_*UART2_INTR*_MAP_REG (0x0074)

Register 5.25. INTERRUPT_CORE0_*SDIO_HOST_INTERRUPT*_MAP_REG (0x0078)

Register 5.26. INTERRUPT_CORE0_*PWM0_INTR*_MAP_REG (0x007C)

Register 5.27. INTERRUPT_CORE0_*PWM1_INTR*_MAP_REG (0x0080)

Register 5.28. INTERRUPT_CORE0_*LEDC_INT*_MAP_REG (0x008C)

Register 5.29. INTERRUPT_CORE0_*EFUSE_INT*_MAP_REG (0x0090)

Register 5.30. INTERRUPT_CORE0_*CAN_INT*_MAP_REG (0x0094)

Register 5.66. INTERRUPT_CORE0_*CPU_INTR_FROM_CPU_0*_MAP_REG (0x013C)

Register 5.67. INTERRUPT_CORE0_*CPU_INTR_FROM_CPU_1*_MAP_REG (0x0140)

Register 5.68. INTERRUPT_CORE0_*CPU_INTR_FROM_CPU_2*_MAP_REG (0x0144)

Register 5.69. INTERRUPT_CORE0_*CPU_INTR_FROM_CPU_3*_MAP_REG (0x0148)

Register 5.70. INTERRUPT_CORE0_*ASSIST_DEBUG_INTR*_MAP_REG (0x014C)

Register 5.71. INTERRUPT_CORE0_*DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0150)

Register 5.72. INTERRUPT_CORE0_*CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0154)

Register 5.73. INTERRUPT_CORE0_*CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0158)

Register 5.74. INTERRUPT_CORE0_*CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x015C)

Register 5.75. INTERRUPT_CORE0_*CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR*_MAP_REG (0x0160)

Register 5.76. INTERRUPT_CORE0_*CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0164)

Register 5.77. INTERRUPT_CORE0_*CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0168)

Register 5.78. INTERRUPT_CORE0_*CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x016C)

Register 5.79. INTERRUPT_CORE0_*CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR*_MAP_REG (0x0170)

Register 5.80. INTERRUPT_CORE0_*BACKUP_PMS_VIOLATE_INTR*_MAP_REG (0x0174)

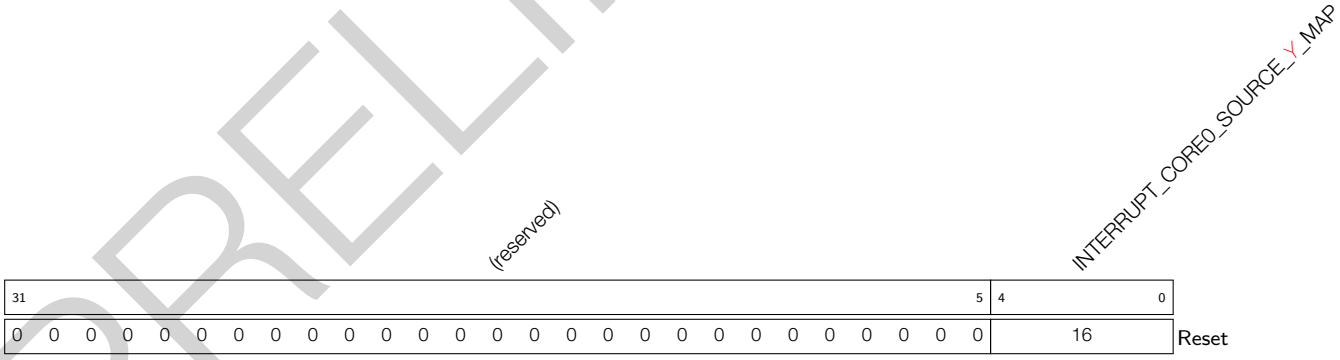
Register 5.81. INTERRUPT_CORE0_*CACHE_CORE0_ACS_INT*_MAP_REG (0x0178)

Register 5.82. INTERRUPT_CORE0_*CACHE_CORE1_ACS_INT*_MAP_REG (0x017C)

Register 5.83. INTERRUPT_CORE0_*USB_DEVICE_INT*_MAP_REG (0x0180)

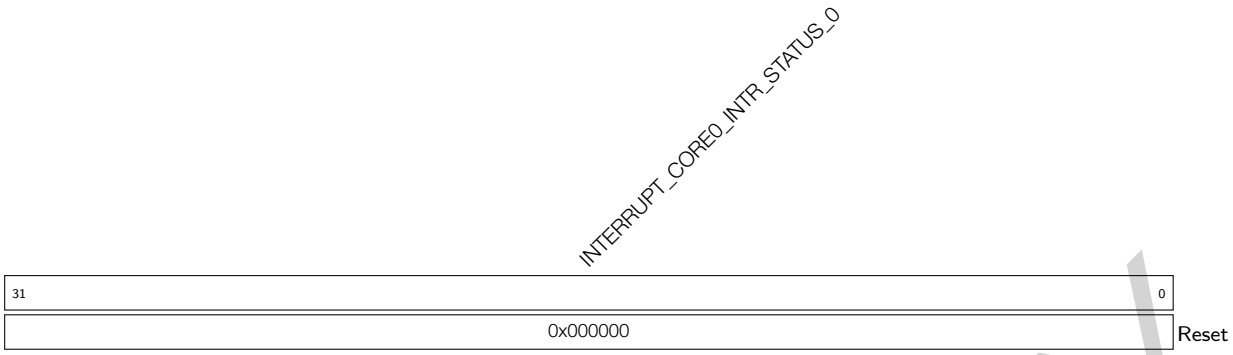
Register 5.84. INTERRUPT_CORE0_*PERI_BACKUP_INT*_MAP_REG (0x0184)

Register 5.85. INTERRUPT_CORE0_*DMA_EXTMEM_REJECT_INT*_MAP_REG (0x0188)



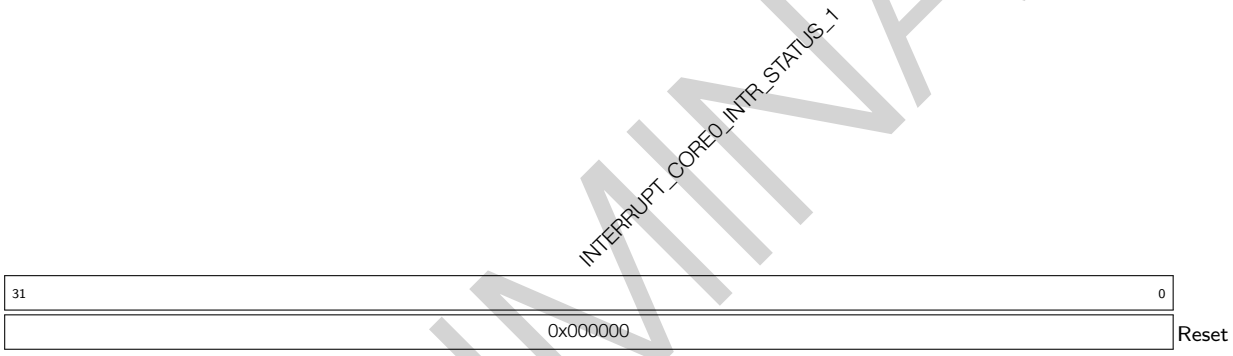
INTERRUPT_CORE0_SOURCE_*Y*_MAP 将中断源 Source_*Y* 的中断信号映射至 CPU0 外部中断，可配置为 0～5、8～10、12～14、17～28 和 30～31，其它值无效。中断源 Source_*Y* 见表 5-1。(R/W)

Register 5.86. INTERRUPT_CORE0_INTR_STATUS_0_REG (0x018C)



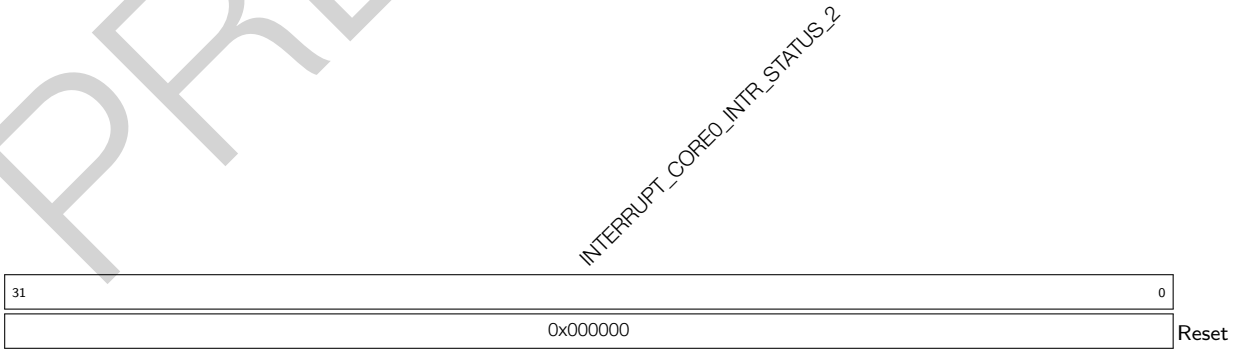
INTERRUPT_CORE0_INTR_STATUS_0 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：0 ~ 31。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 5.87. INTERRUPT_CORE0_INTR_STATUS_1_REG (0x0190)



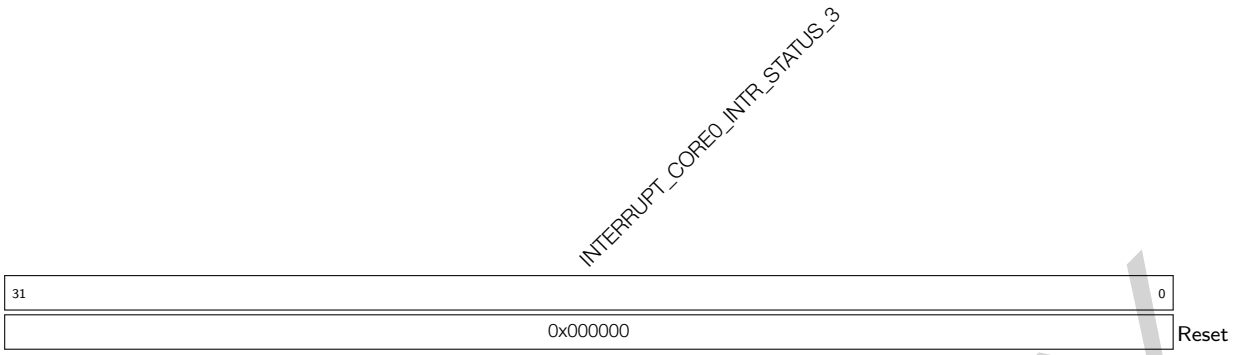
INTERRUPT_CORE0_INTR_STATUS_1 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：32 ~ 63。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 5.88. INTERRUPT_CORE0_INTR_STATUS_2_REG (0x0194)



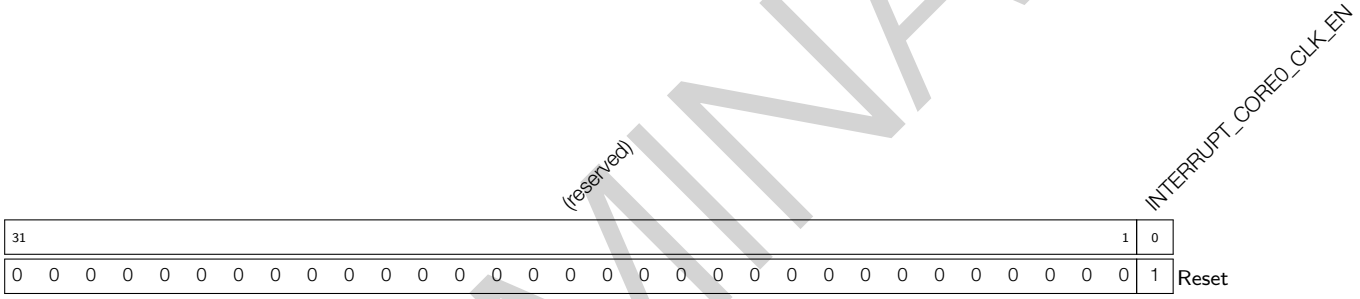
INTERRUPT_CORE0_INTR_STATUS_2 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：64 ~ 95。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 5.89. INTERRUPT_CORE0_INTR_STATUS_3_REG (0x0198)



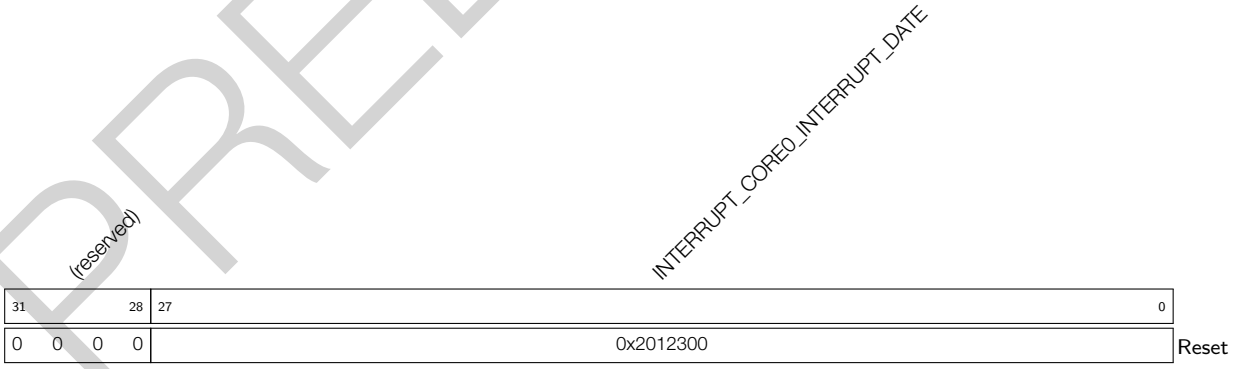
INTERRUPT_CORE0_INTR_STATUS_3 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：96 ~ 98。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 5.90. INTERRUPT_CORE0_CLOCK_GATE_REG (0x019C)



INTERRUPT_CORE0_CLK_EN 中断矩阵的时钟门控控制位。(R/W)

Register 5.91. INTERRUPT_CORE0_DATE_REG (0x07FC)



INTERRUPT_CORE0_INTERRUPT_DATE 版本控制寄存器。(R/W)

5.5.2 CPU1 中断寄存器

Register 5.92. INTERRUPT_CORE1_*MAC_INTR*_MAP_REG (0x0800)

Register 5.163. INTERRUPT_CORE1_*CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0954)

Register 5.164. INTERRUPT_CORE1_*CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0958)

Register 5.165. INTERRUPT_CORE1_*CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x095C)

Register 5.166. INTERRUPT_CORE1_*CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR*_MAP_REG (0x0960)

Register 5.167. INTERRUPT_CORE1_*CORE_1_IRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0964)

Register 5.168. INTERRUPT_CORE1_*CORE_1_DRAM0_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x0968)

Register 5.169. INTERRUPT_CORE1_*CORE_1_PIF_PMS_MONITOR_VIOLATE_INTR*_MAP_REG (0x096C)

Register 5.170. INTERRUPT_CORE1_*CORE_1_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR*_MAP_REG (0x0970)

Register 5.171. INTERRUPT_CORE1_*BACKUP_PMS_VIOLATE_INTR*_MAP_REG (0x0974)

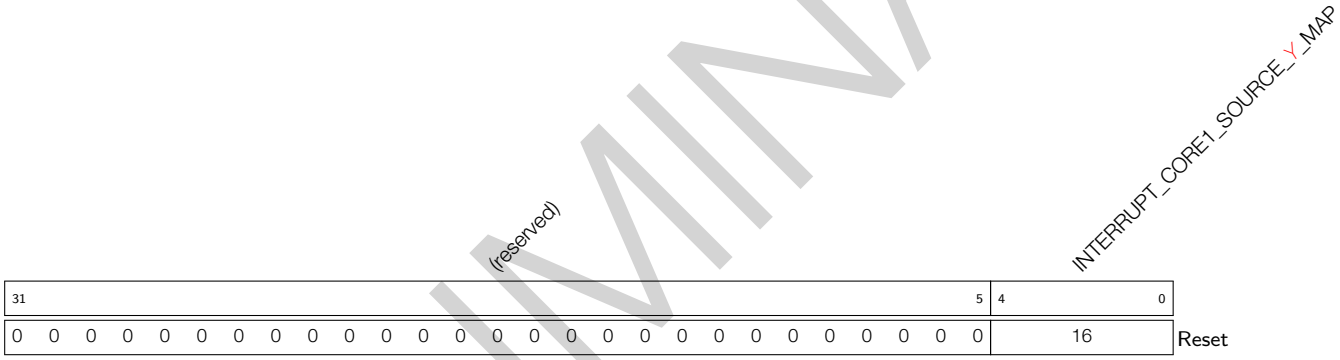
Register 5.172. INTERRUPT_CORE1_*CACHE_CORE0_ACS_INT*_MAP_REG (0x0978)

Register 5.173. INTERRUPT_CORE1_*CACHE_CORE1_ACS_INT*_MAP_REG (0x097C)

Register 5.174. INTERRUPT_CORE1_*USB_DEVICE_INT*_MAP_REG (0x0980)

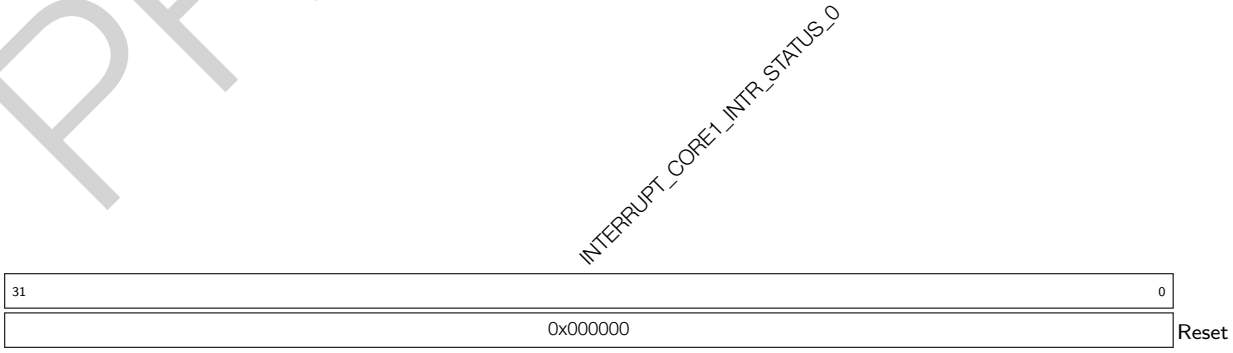
Register 5.175. INTERRUPT_CORE1_*PERI_BACKUP_INT*_MAP_REG (0x0984)

Register 5.176. INTERRUPT_CORE1_*DMA_EXTMEM_REJECT_INT*_MAP_REG (0x0988)



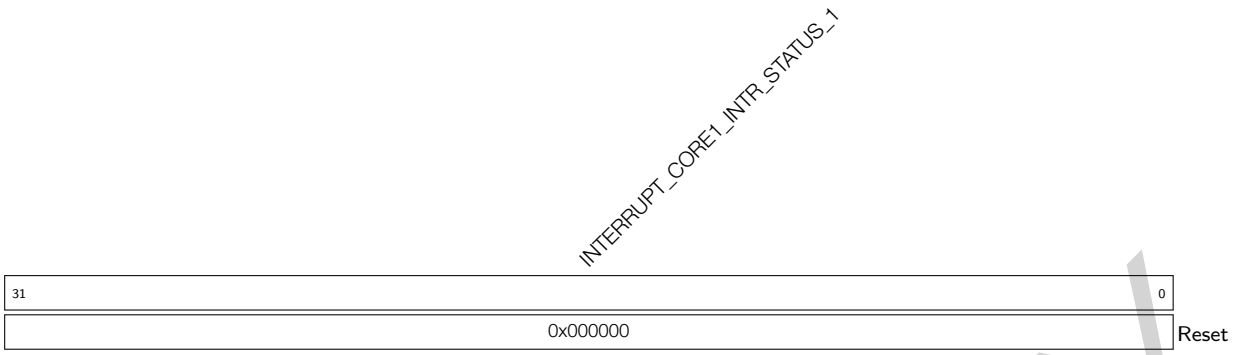
INTERRUPT_CORE1_SOURCE_*Y*_MAP 将中断源 Source_*Y* 的中断信号映射至 CPU1 外部中断，可配置为 0 ~ 5、8 ~ 10、12 ~ 14、17 ~ 28 和 30 ~ 31，其它值无效。中断源 Source_*Y* 见表 5-1。(R/W)

Register 5.177. INTERRUPT_CORE1_INTR_STATUS_0_REG (0x098C)



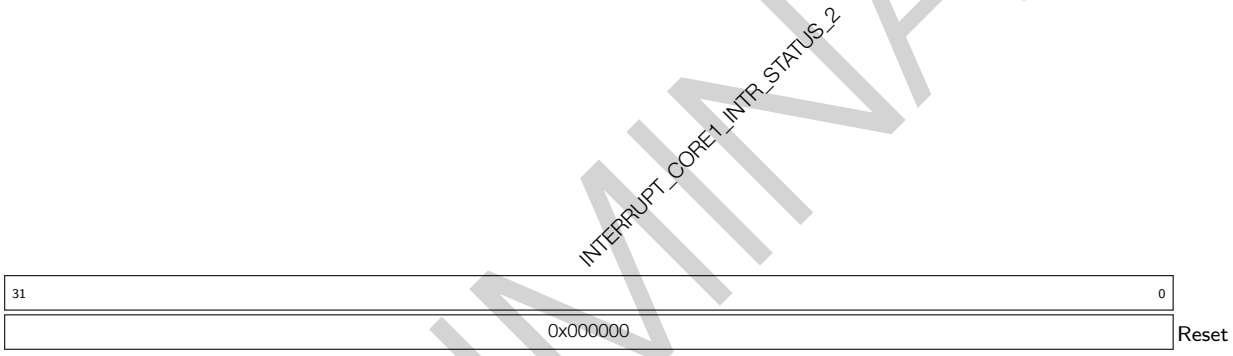
INTERRUPT_CORE1_INTR_STATUS_0 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：0 ~ 31。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 5.178. INTERRUPT_CORE1_INTR_STATUS_1_REG (0x0990)



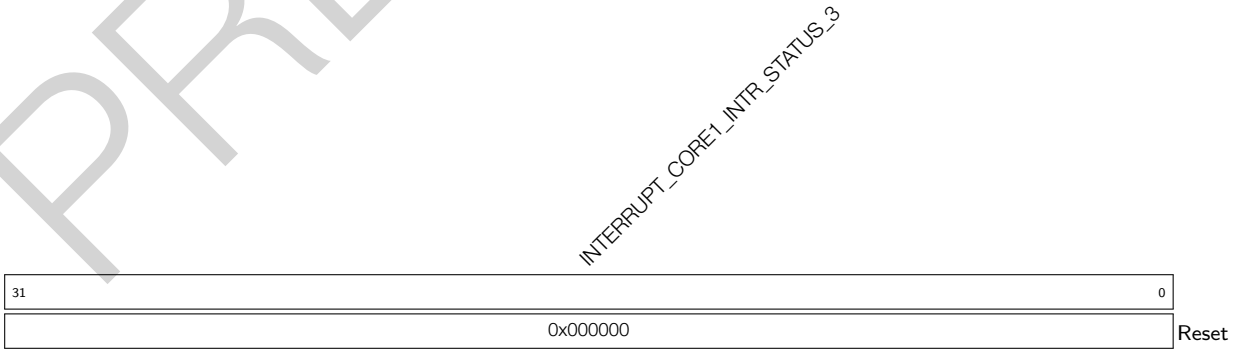
INTERRUPT_CORE1_INTR_STATUS_1 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：32 ~ 63。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 5.179. INTERRUPT_CORE1_INTR_STATUS_2_REG (0x0994)



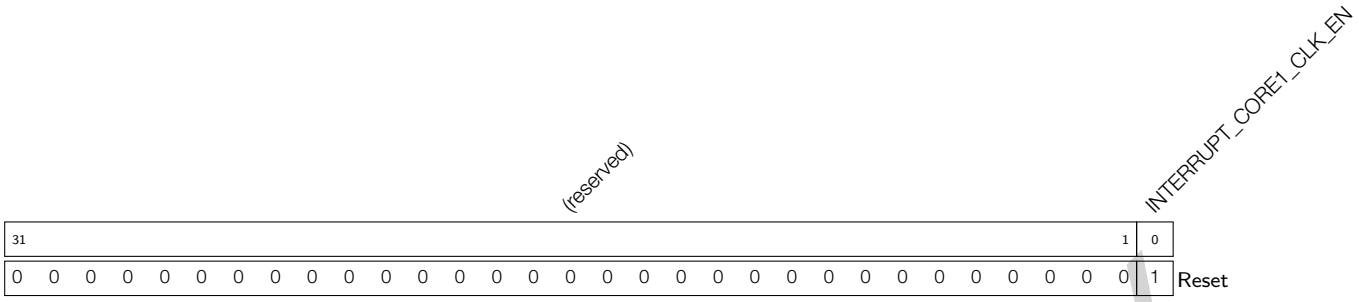
INTERRUPT_CORE1_INTR_STATUS_2 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：64 ~ 95。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 5.180. INTERRUPT_CORE1_INTR_STATUS_3_REG (0x0998)



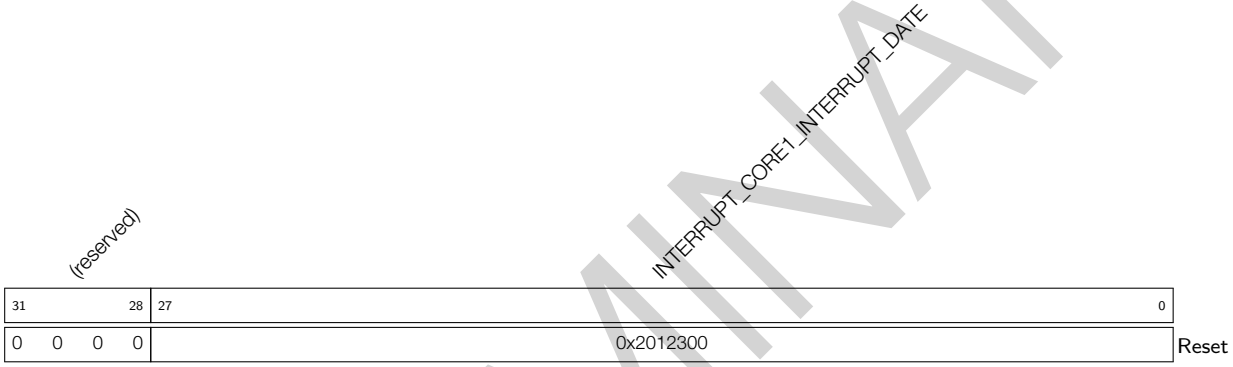
INTERRUPT_CORE1_INTR_STATUS_3 用于存储外部中断源的状态，每一位均代表一个外部中断源的状态，对应中断编号源：96 ~ 98。如果对应的位为 1，则表示该中断源触发了中断。(RO)

Register 5.181. INTERRUPT_CORE1_CLOCK_GATE_REG (0x099C)



INTERRUPT_CORE1_CLK_EN 中断矩阵的时钟门控控制位。(R/W)

Register 5.182. INTERRUPT_CORE1_DATE_REG (0x0FFC)



INTERRUPT_CORE1_INTERRUPT_DATE 版本控制寄存器。(R/W)

6 定时器组 (TIMG)

6.1 概述

通用定时器可用于准确设定时间间隔、在一定间隔后触发（周期或非周期的）中断或充当硬件时钟。如图 6-1 所示，ESP32-S3 包含两个定时器组，即定时器组 0 和定时器组 1。每个定时器组有两个通用定时器（下文用 T*x* 表示，*x* 为 0 或 1）和一个主系统看门狗定时器。所有通用定时器均基于 16 位预分频器和 54 位可自动重新加载向上 / 向下计数器。

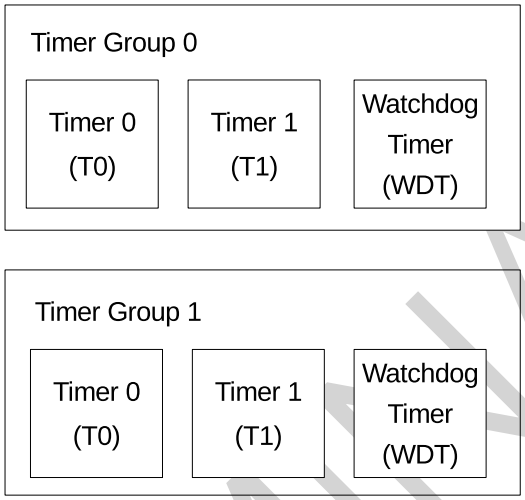


图 6-1. 定时器组

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 7 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 2 到 65536
- 54 位时基计数器可配置成递增或递减
- 可读取时基计数器的实时值
- 暂停和恢复时基计数器
- 可配置的报警产生机制
- 计数器值重新加载（报警时自动重新加载或软件控制的即时重新加载）
- 电平触发中断

6.2 功能描述

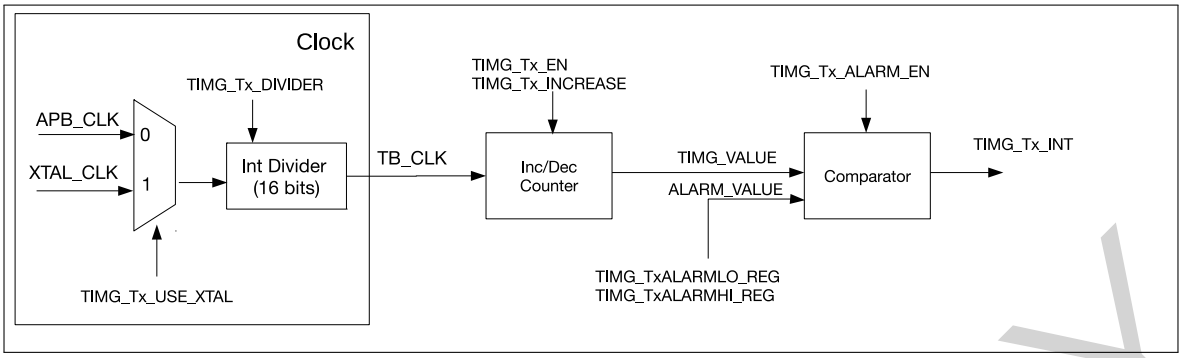


图 6-2. 定时器组架构

图 6-2 为定时器组的 T*x*。T*x* 包含时钟选择器、一个 16 位整数预分频器、一个时基计数器和一个用于产生警报的比较器。

6.2.1 16 位预分频器与时钟选择器

每个定时器可通过配置寄存器 TIMG_T*x*CONFIG_REG 的 TIMG_T*x*_USE_XTAL 字段，选择 APB 时钟 (APB_CLK) 或外部时钟 (XTAL_CLK) 作为时钟源。时钟源经 16 位预分频器分频，产生时基计数器使用的时基计数器时钟 (TB_CLK)。16 位预分频器的分频系数可通过 TIMG_T*x*_DIVIDER 字段配置，选取从 2 到 65536 之间的任意值。注意，将 TIMG_T*x*_DIVIDER 置 0 后，分频系数会变为 65536。TIMG_T*x*_DIVIDER 置 1 时，实际分频系数为 2，计数器的值为实际时间的一半。

定时器必须关闭（即 TIMG_T*x*_EN 必须清零），才能更改 16 位预分频器。在定时器使能时更改 16 位预分频器会造成不可预知的结果。

6.2.2 54 位时基计数器

54 位时基计数器基于 TB_CLK，可通过 TIMG_T*x*_INCREASE 字段配置为递增或递减。时基计数器可通过置位或清零 TIMG_T*x*_EN 字段使能或关闭。使能时，时基计数器的值会在每个 TB_CLK 周期递增或递减。关闭时，时基计数器暂停计数。注意，TIMG_T*x*_EN 置位后，TIMG_T*x*_INCREASE 字段还可以更改，时基计数器可立即改变计数方向。

时基计数器 54 位定时器的当前值必须被锁入两个寄存器，才能被 CPU 读取（因为 CPU 为 32 位）。在 TIMG_T*x*UPDATE_REG 上写任意值，54 位定时器的值可立即锁入寄存器 TIMG_T*x*LO_REG 和 TIMG_T*x*HI_REG，两个寄存器分别锁存低 32 位和高 22 位。在 TIMG_T*x*UPDATE_REG 被写入新值之前，寄存器 TIMG_T*x*LO_REG 和 TIMG_T*x*HI_REG 的值保持不变，以便 CPU 读值。

6.2.3 报警产生

定时器可配置为在当前值与报警值相同时触发报警。报警会产生中断，（可选择）让定时器的当前值自动重新加载（详见第 6.2.4节）。

54 位报警值可在 TIMG_T*x*ALARMLO_REG 和 TIMG_T*x*ALARMHI_REG 配置，两者分别代表报警值的低 32 位和高 22 位。但是，只有置位 TIMG_T*x*_ALARM_EN 字段使能报警功能后，配置的报警值才会生效。为解决报警使能“过晚”（即报警使能时，定时器的值已过报警值），可逆计数器向上计数时，若定时器的当前值高于报警值（在一定范围内），或可逆计数器向下计数时，定时器的当前值低于报警值（在一定范围内），硬件都会立即触发报警。表 6-1 和表 6-2 说明了定时器当前值、报警值与报警触发的关系。假设定时器当前值和报警值如下：

- TIMG_VALUE = {TIMG_TxHI_REG, TIMG_TxLO_REG}
- ALARM_VALUE = {TIMG_TxALARMHI_REG, TIMG_TxALARMLO_REG}

表 6-1. 可逆计数器向上计数时的报警触发场景

场景	范围	报警
1	ALARM_VALUE − TIMG_VALUE $> 2^{53}$	触发
2	$0 <$ ALARM_VALUE − TIMG_VALUE $\leq 2^{53}$	定时器计数器向上计数，TIMG_VALUE 达到 ALARM_VALUE 时报警
3	$0 \leq$ TIMG_VALUE − ALARM_VALUE $< 2^{53}$	触发
4	TIMG_VALUE − ALARM_VALUE $\geq 2^{53}$	定时器计数器向上计数达到最大值时，重新开始从 0 向上计数，TIMG_VALUE 达到 ALARM_VALUE 时触发报警

表 6-2. 可逆计数器向下计数时的报警触发场景

场景	范围	报警
5	TIMG_VALUE − ALARM_VALUE $> 2^{53}$	触发
6	$0 <$ TIMG_VALUE − ALARM_VALUE $\leq 2^{53}$	定时器计数器向下计数，TIMG_VALUE 达到 ALARM_VALUE 时报警
7	$0 \leq$ ALARM_VALUE − TIMG_VALUE $< 2^{53}$	触发
8	ALARM_VALUE − TIMG_VALUE $\geq 2^{53}$	定时器计数器向下计数达到最小值时，重新开始从最大值向下计数，TIMG_VALUE 达到 ALARM_VALUE 时触发报警

报警时，TIMG_Tx_ALARM_EN 字段自动清零，在下次置位 TIMG_Tx_ALARM_EN 前不会再次报警。

6.2.4 定时器重新加载

定时器重新加载指将定时器的低 32 位和高 22 位分别更新为寄存器 TIMG_Tx_LOAD_LO 和 TIMG_Tx_LOAD_HI 存储的重新加载值。但是，把重新加载值写入 TIMG_Tx_LOAD_LO 和 TIMG_Tx_LOAD_HI 寄存器不会改变定时器的当前值。写入的重新加载值会被定时器忽视，直到重新加载事件被触发。重新加载事件可由软件即时重新加载或报警时自动重新加载触发。

CPU 在寄存器 TIMG_TxLOAD_REG 写任意值会触发软件即时重新加载，定时器的当前值会立即改变。如置位 TIMG_Tx_EN，定时器会继续从新数值开始递增或递减计数。如清零 TIMG_Tx_EN，定时器将保持当前值，直至计数重新使能。

报警时自动重新加载功能可让定时器在报警时重新加载，从重新加载值开始继续递增或递减计数。该功能通常用于周期性报警时重置定时器的值。TIMG_Tx_AUTORELOAD 字段置 1 可以使能报警时自动重新加载。如未使能该功能，报警后定时器的值会在过报警值后继续递增或递减。

6.2.5 低功耗时钟 (SLOW_CLK) 频率计算

定时器组可以通过使用 XTAL_CLK 计算低功耗时钟的三个慢速时钟源 RTC_CLK、RTC20M_D256_CLK 和 XTAL32K_CLK 的实际频率。计算方式如下：

1. 通过周期性或单次计算的方式启动频率计算模块；

2. 在接收到计算开始的信号后，两个分别工作在 XTAL_CLK 以及 SLOW_CLK 的计数器同时开始计数，当 SLOW_CLK 的计数器达到设定的计算周期 C0 时，同时停止两个计数器；
3. 通过 XTAL_CLK 的计数器值 C1 即可计算 SLOW_CLK 的时钟频率：$f\_rtc = \frac{C0 \times f\_XTAL\_CLK}{C1}$

6.2.6 中断

每个定时器都有一根连接至 CPU 的中断线。因此，每个定时器组有三根中断线。定时器每次产生的电平中断必须由 CPU 清除。

电平中断在报警后（或看门狗定时器阶段超时）触发。报警（或阶段超时）后，电平中断会一直被拉高，直至手动清除中断。要使能定时器的中断，TIMG_Tx_INT_ENA 需置 1。

每个定时器组的中断受一组寄存器控制。每个定时器在下列寄存器中都有对应的位：

- TIMG_Tx_INT_RAW：报警时置 1。该位在写值到对应的 TIMG_Tx_INT_CLR 位后才会被清零。
- TIMG_WDT_INT_RAW：阶段超时时置 1。该位在写值到对应的 TIMG_WDT_INT_CLR 位后才会被清零。
- TIMG_Tx_INT_ST：反映每个定时器中断的状态，通过用 TIMG_Tx_INT_ENA 屏蔽 TIMG_Tx_INT_RAW 位来生成。
- TIMG_WDT_INT_ST：反映每个看门狗定时器中断的状态，通过用 TIMG_WDT_INT_ENA 屏蔽 TIMG_WDT_INT_RAW 位来生成。
- TIMG_Tx_INT_ENA：用于使能或屏蔽组内定时器的中断状态位。
- TIMG_WDT_INT_ENA：用于使能或屏蔽组内看门狗定时器的中断状态位。
- TIMG_Tx_INT_CLR：置 1 此位清除定时器中断，定时器在 TIMG_Tx_INT_RAW 和 TIMG_Tx_INT_ST 的对应位会清零。注意，下一个中断产生前，必须清除定时器中断。
- TIMG_WDT_INT_CLR：置 1 此位清除定时器中断，看门狗定时器在 TIMG_WDT_INT_RAW 和 TIMG_WDT_INT_ST 的对应位会清零。注意，下一个中断产生前，必须清除看门狗定时器中断。

6.3 配置与使用

6.3.1 定时器用作简单时钟

1. 配置时基计数器。
   - 置位或清除 TIMG_Tx_USE_XTAL 字段选择时钟源。
   - 置位 TIMG_Tx_DIVIDER 配置 16 位预分频器。
   - 置位或清除 TIMG_Tx_INCREASE 配置定时器方向。
   - 在 TIMG_Tx_LOAD_LO 和 TIMG_Tx_LOAD_HI 上写初始值设置定时器的初始值，然后在 TIMG_TxLOAD_REG 上写任意值将初始值重新加载进定时器。
2. 置位 TIMG_Tx_EN 开启定时器。
3. 获得定时器的当前值。
   - 在 TIMG_TxUPDATE_REG 上写任意值锁存定时器的当前值。
   - 从 TIMG_TxLO_REG 和 TIMG_TxHI_REG 读取锁存的定时器值。

6.3.2 定时器用于一次性报警

1. 按照第 6.3.1 节的第 1 步配置时基计数器。
2. 配置报警。
   - 置位 TIMG_TxALARMLO_REG 和 TIMG_TxALARMHI_REG 配置报警值。
   - 置位 TIMG_Tx_INT_ENA 使能中断。
3. 清零 TIMG_Tx_AUTORELOAD 关闭自动重新加载。
4. 置位 TIMG_Tx_ALARM_EN 开启报警。
5. 处理报警中断。
   - 置位定时器在 TIMG_Tx_INT_CLR 的对应位清除中断。
   - 清零 TIMG_Tx_EN 关闭定时器。

6.3.3 定时器用于周期性报警

1. 按照第 6.3.1 节的第 1 步配置时基计数器。
2. 按照第 6.3.2节的第 2 步配置报警。
3. 置位 TIMG_Tx_AUTORELOAD 使能自动重新加载，将重新加载值写入 TIMG_Tx_LOAD_LO 和 TIMG_Tx_LOAD_HI。
4. 置位 TIMG_Tx_ALARM_EN 开启报警。
5. 处理报警中断（每次报警时重复）。
   - 置位定时器在 TIMG_Tx_INT_CLR 的对应位清除中断。
   - 如下一次报警需要新的报警值和重新加载值（即每次都有不同的报警间隔），则应根据需要重新配置 TIMG_TxALARMLO_REG、TIMG_TxALARMHI_REG、TIMG_Tx_LOAD_LO 和 TIMG_Tx_LOAD_HI 。否则，上述寄存器应保持不变。
   - 置位 TIMG_Tx_ALARM_EN 重新使能报警。
6. （最后一次报警时）关闭定时器。
   - 置位定时器在 TIMG_Tx_INT_CLR 的对应位清除中断。
   - 清零 TIMG_Tx_EN 关闭定时器。

6.3.4 SLOW_CLK 频率计算

1. 单次计算
   - 设置 TIMG_RTC_CALI_CLK_SEL 选择需要计算频率的时钟（SLOW_CLK 的时钟源），设置 TIMG_RTC_CALI_MAX 配置频率计算时间。
   - 清空 TIMG_RTC_CALI_START_CYCLING 选择单次校准模式，然后配置 TIMG_RTC_CALI_START 开启两个计数器。
   - 等待 TIMG_RTC_CALI_RDY 的值变为 1 ，读取TIMG_RTC_CALI_VALUE 获取 XTAL_CLK 计数器值，计算 SLOW_CLK 频率。
2. 周期性计算

- 设置 TIMG_RTC_CALI_CLK_SEL 选择需要计算频率的时钟(SLOW_CLK 的时钟源),设置 TIMG_RTC_CALI_MAX 配置频率计算时间。
- 使能 TIMG_RTC_CALI_START_CYCLING，硬件将不间断进行频率计算过程。
- 只要TIMG_RTC_CALI_CYCLING_DATA_VLD 为 1，即表示TIMG_RTC_CALI_VALUE 有效。

3. 超时
   如果 SLOW_CLK 的计数器没有在TIMG_RTC_CALI_TIMEOUT_RST_CNT 的 XTAL_CLK 计数器内完成计数，将置位 TIMG_RTC_CALI_TIMEOUT 标记计算超时。

6.4 寄存器列表

本小节的所有地址均为相对于 定时器组 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

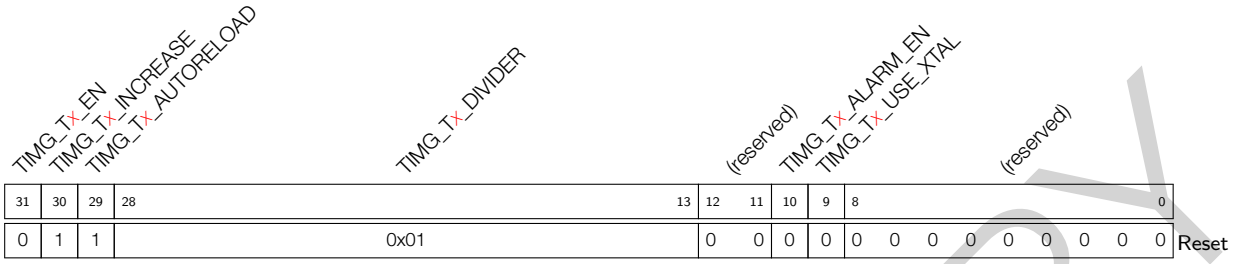
名称	描述	地址	访问
定时器 0 配置和控制寄存器			
TIMG_T0CONFIG_REG	定时器 0 配置寄存器	0x0000	varies
TIMG_T0LO_REG	定时器 0 的当前值，低 32 位	0x0004	RO
TIMG_T0HI_REG	定时器 0 的当前值，高 22 位	0x0008	RO
TIMG_T0UPDATE_REG	写值将当前定时器的值复制到 TIMG_T0LO_REG 或 TIMG_T0HI_REG	0x000C	R/ W/ SC
TIMG_T0ALARMLO_REG	定时器 0 的报警值，低 32 位	0x0010	R/W
TIMG_T0ALARMHI_REG	定时器 0 的报警值，高位	0x0014	R/W
TIMG_T0LOADLO_REG	定时器 0 的重新加载值，低 32 位	0x0018	R/W
TIMG_T0LOADHI_REG	定时器 0 的重新加载值，高 22 位	0x001C	R/W
TIMG_T0LOAD_REG	写值从 TIMG_T0LOADLO_REG 或 TIMG_T0LOADHI_REG 上加载定时器	0x0020	WT
定时器 1 配置和控制寄存器			
TIMG_T1CONFIG_REG	定时器 1 配置寄存器	0x0024	varies
TIMG_T1LO_REG	定时器 1 的当前值，低 32 位	0x0028	RO
TIMG_T1HI_REG	定时器 1 的当前值，高 22 位	0x002C	RO
TIMG_T1UPDATE_REG	写值将当前定时器的值复制到 TIMG_T1LO_REG 或 TIMG_T1HI_REG	0x0030	R/ W/ SC
TIMG_T1ALARMLO_REG	定时器 1 的报警值，低 32 位	0x0034	R/W
TIMG_T1ALARMHI_REG	定时器 1 的报警值，高位	0x0038	R/W
TIMG_T1LOADLO_REG	定时器 1 的重新加载值，低 32 位	0x003C	R/W
TIMG_T1LOADHI_REG	定时器 1 的重新加载值，高 22 位	0x0040	R/W
TIMG_T1LOAD_REG	写值从 TIMG_T1LOADLO_REG 或 TIMG_T1LOADHI_REG 上加载定时器	0x0044	WT
看门狗定时器配置和控制寄存器			
TIMG_WDTCONFIG0_REG	看门狗定时器配置寄存器	0x0048	R/W
TIMG_WDTCONFIG1_REG	看门狗定时器预分频器寄存器	0x004C	R/W
TIMG_WDTCONFIG2_REG	看门狗定时器阶段 0 超时值	0x0050	R/W
TIMG_WDTCONFIG3_REG	看门狗定时器阶段 1 超时值	0x0054	R/W
TIMG_WDTCONFIG4_REG	看门狗定时器阶段 2 超时值	0x0058	R/W
TIMG_WDTCONFIG5_REG	看门狗定时器阶段 3 超时值	0x005C	R/W
TIMG_WDTFEED_REG	写值喂看门狗定时器	0x0060	WT
TIMG_WDTWPROTECT_REG	看门狗写保护寄存器	0x0064	R/W
RTC 频率计算配置和控制寄存器			
TIMG_RTCCALICFG_REG	RTC 频率计算配置寄存器 0	0x0068	varies
TIMG_RTCCALICFG1_REG	RTC 频率计算配置寄存器 1	0x006C	RO
TIMG_RTCCALICFG2_REG	RTC 频率计算配置寄存器 2	0x0080	varies
中断寄存器			
TIMG_INT_ENA_TIMERS_REG	中断使能位	0x0070	R/W

名称	描述	地址	访问
TIMG_INT_RAW_TIMERS_REG	原始中断状态	0x0074	R/WTC/SS
TIMG_INT_ST_TIMERS_REG	屏蔽中断状态	0x0078	RO
TIMG_INT_CLR_TIMERS_REG	中断清除位	0x007C	WT
版本寄存器			
TIMG_NTIMERS_DATE_REG	版本控制寄存器	0x00F8	R/W
定时器组配置寄存器			
TIMG_REGCLK_REG	定时器组时钟门控寄存器	0x00FC	R/W

6.5 寄存器

本小节的所有地址均为相对于 定时器组 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

Register 6.1. TIMG_TxCONFIG_REG (x: 0-1) (0x0000+0x24*x)



TIMG_Tx_USE_XTAL 0：使用 APB_CLK 作为定时器组的源时钟；1：使用 XTAL_CLK 作为定时器组的源时钟。(R/W)

TIMG_Tx_ALARM_EN 置 1 后，报警使能。报警时，此位自动清零。(R/W/SC)

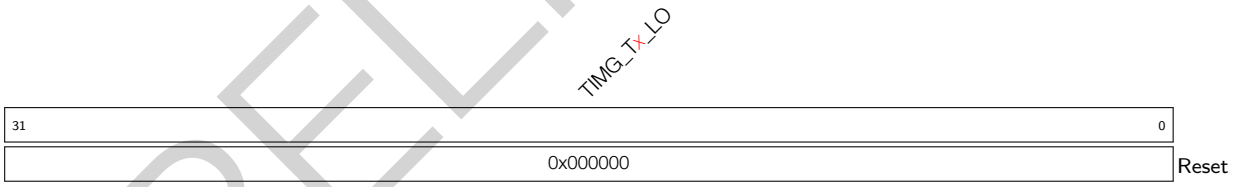
TIMG_Tx_DIVIDER 定时器 x 时钟 (Tx_clk) 的预分频器值。(R/W)

TIMG_Tx_AUTORELOAD 置 1 后，定时器 x 报警时自动重新加载使能。(R/W)

TIMG_Tx_INCREASE 置 1 后，定时器 x 的时基计数器会在每个时钟周期后递增。清零后，定时器 x 的时基计数器会递减。(R/W)

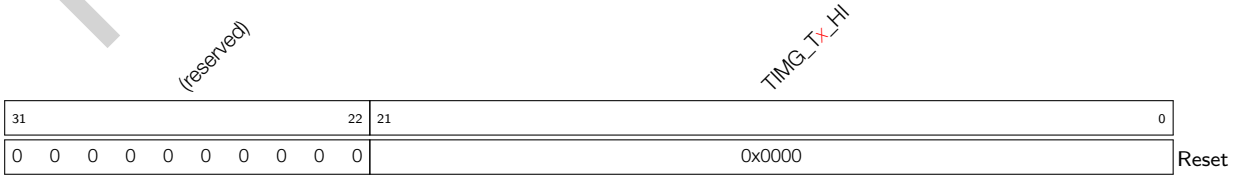
TIMG_Tx_EN 置 1 后，定时器 x 时基计数器使能。(R/W)

Register 6.2. TIMG_TxLO_REG (x: 0-1) (0x0004+0x24*x)



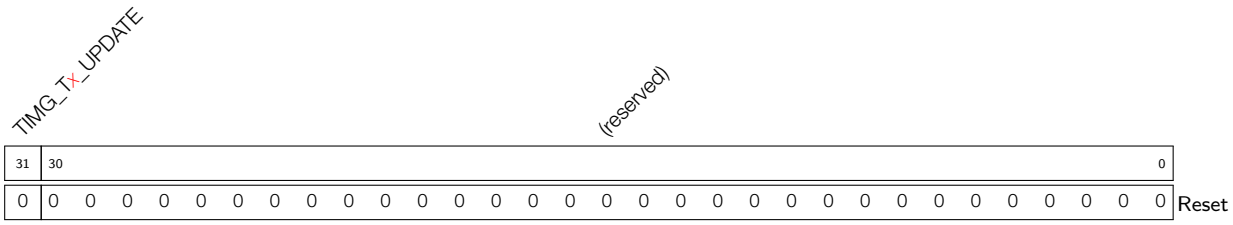
TIMG_Tx_LO 在 TIMG_TxUPDATE_REG 上写值后，可读取定时器 x 时基计数器的低 32 位。(RO)

Register 6.3. TIMG_TxHI_REG (x: 0-1) (0x0008+0x24*x)



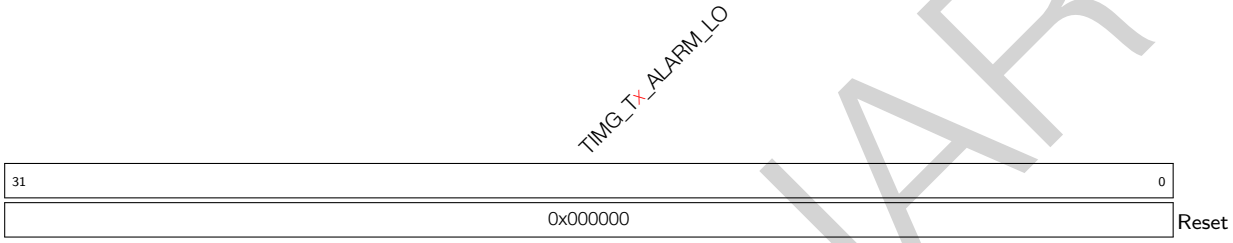
TIMG_Tx_HI 在 TIMG_TxUPDATE_REG 上写值后，可读取定时器 x 时基计数器的高 22 位。(RO)

Register 6.4. TIMG_TxUPDATE_REG (x: 0-1) (0x000C+0x24*x)



TIMG_Tx_UPDATE 在 TIMG_TxUPDATE_REG 上写 0 或 1，计数器的值被锁住。(R/W/SC)

Register 6.5. TIMG_TxALARMLO_REG (x: 0-1) (0x0010+0x24*x)

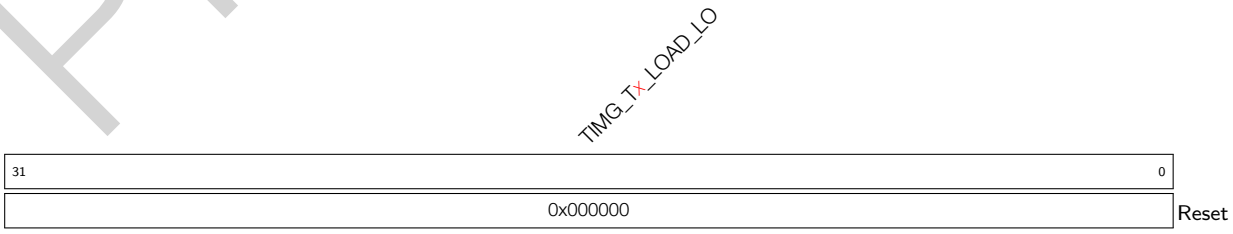


TIMG_Tx_ALARM_LO 定时器 x 时基计数器触发警报值的低 32 位。(R/W)

Register 6.6. TIMG_TxALARMHI_REG (x: 0-1) (0x0014+0x24*x)

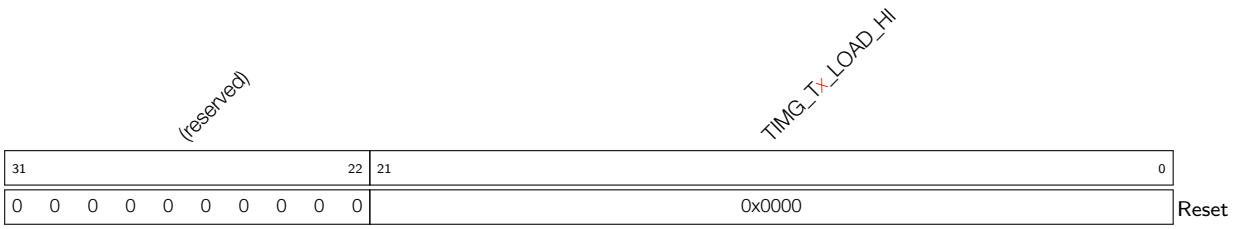
TIMG_Tx_ALARM_HI 定时器 x 时基计数器触发警报值的高 22 位。(R/W)

Register 6.7. TIMG_TxLOADLO_REG (x: 0-1) (0x0018+0x24*x)



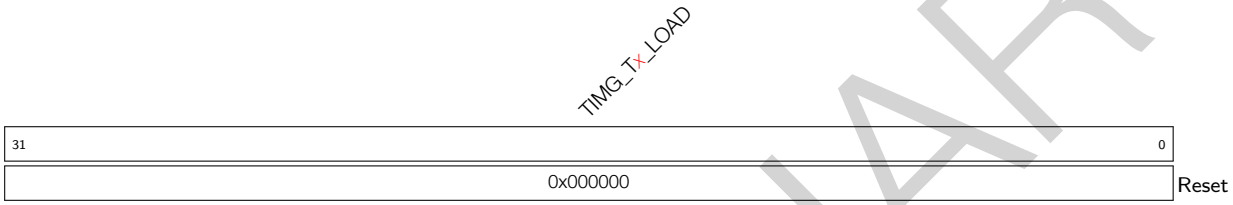
TIMG_Tx_LOAD_LO 定时器 x 时基计数器重新加载的低 32 位值。(R/W)

Register 6.8. TIMG_TxLOADHI_REG (x: 0-1) (0x001C+0x24*x)



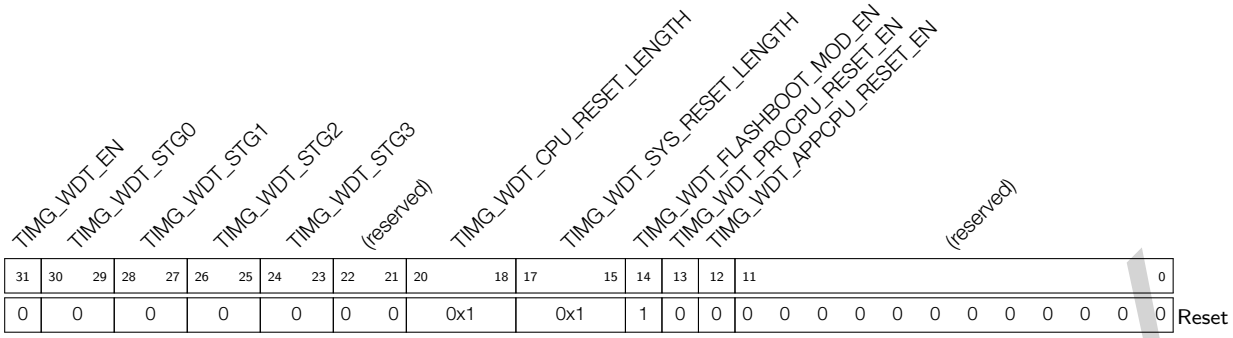
TIMG_Tx_LOAD_HI 定时器 x 时基计数器重新加载的高 22 位值。(R/W)

Register 6.9. TIMG_TxLOAD_REG (x: 0-1) (0x0020+0x24*x)



TIMG_Tx_LOAD 写任意值触发定时器 x 时基计数器重新加载。(WT)

Register 6.10. TIMG_WDTCONFIG0_REG (0x0048)



TIMG_WDT_APPCPU_RESET_EN 保留。(R/W)

TIMG_WDT_PROCPU_RESET_EN WDT 复位 CPU 使能。(R/W)

TIMG_WDT_FLASHBOOT_MOD_EN 置 1 后, flash 启动保护使能。(R/W)

TIMG_WDT_SYS_RESET_LENGTH 系统复位信号长度选择。0：100 ns；1：200 ns；2：300 ns；3：400 ns；4：500 ns；5：800 ns；6：1.6 μs；7：3.2 μs。(R/W)

TIMG_WDT_CPU_RESET_LENGTH CPU 复位信号长度选择。0：100 ns；1：200 ns；2：300 ns；3：400 ns；4：500 ns；5：800 ns；6：1.6 μs；7：3.2 μs。(R/W)

TIMG_WDT_STG3 阶段 3 配置。0：关闭；1：中断；2：复位 CPU；3：复位系统。(R/W)

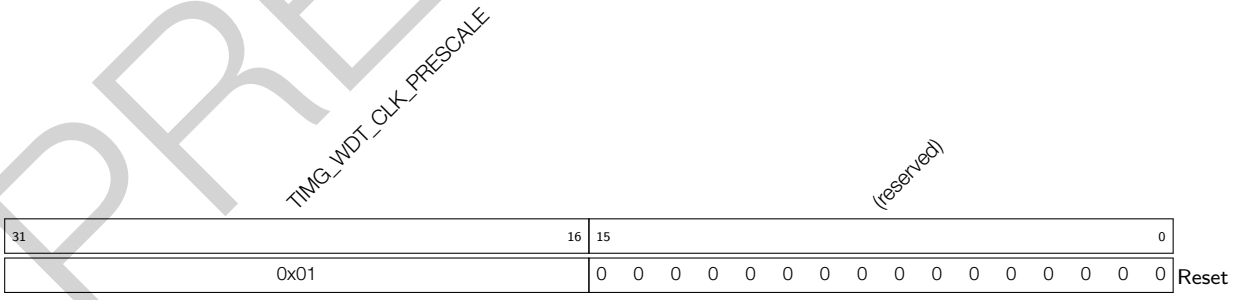
TIMG_WDT_STG2 阶段 2 配置。0：关闭；1：中断；2：复位 CPU；3：复位系统。(R/W)

TIMG_WDT_STG1 阶段 1 配置。0：关闭；1：中断；2：复位 CPU；3：复位系统。(R/W)

TIMG_WDT_STG0 阶段 0 配置。0：关闭；1：中断；2：复位 CPU；3：复位系统。(R/W)

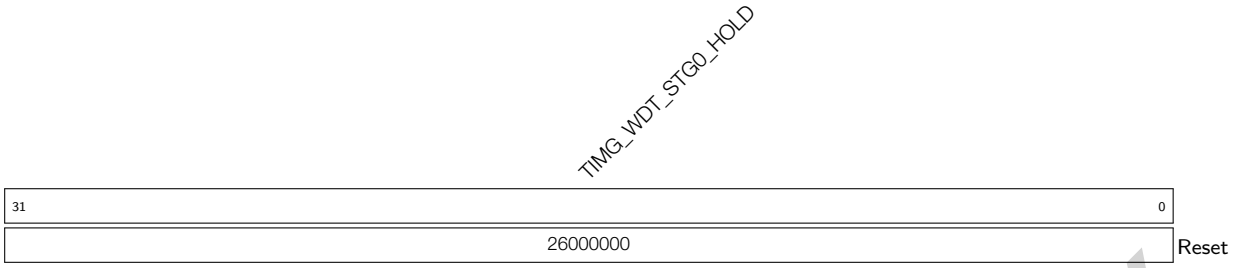
TIMG_WDT_EN 置 1 后，MWDT 使能。(R/W)

Register 6.11. TIMG_WDTCONFIG1_REG (0x004C)



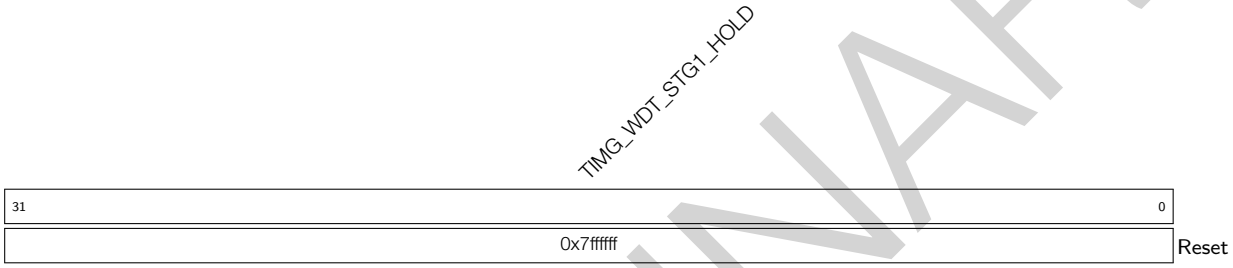
TIMG_WDT_CLK_PRESCALE MWDT 时钟预分频器值。MWDT 时钟长度 = 12.5 ns * TIMG_WDT_CLK_PRESCALE。(R/W)

Register 6.12. TIMG_WDTCONFIG2_REG (0x0050)



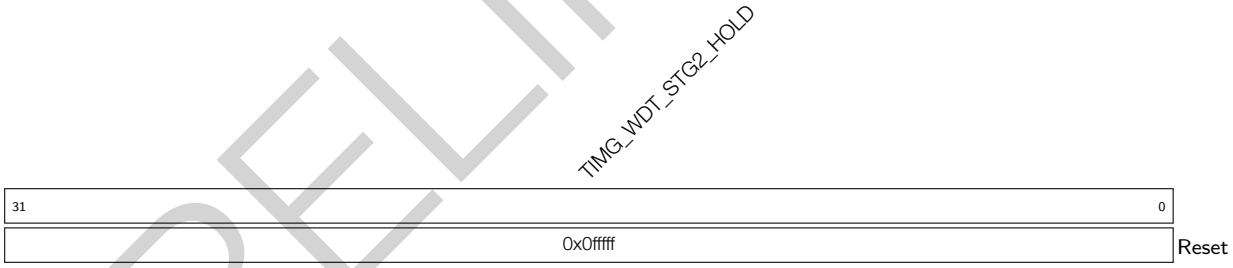
TIMG_WDT_STG0_HOLD 阶段 0 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 6.13. TIMG_WDTCONFIG3_REG (0x0054)



TIMG_WDT_STG1_HOLD 阶段 1 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 6.14. TIMG_WDTCONFIG4_REG (0x0058)

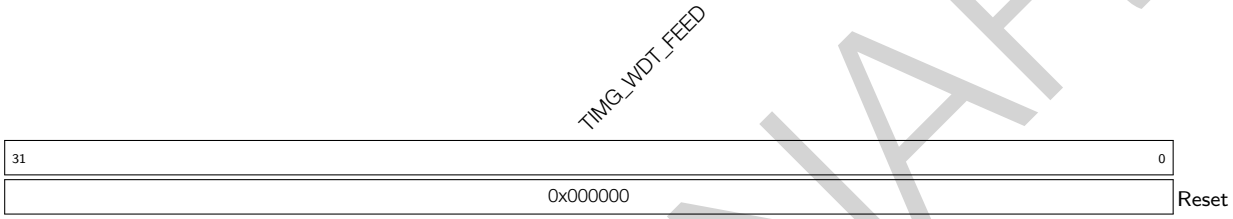


TIMG_WDT_STG2_HOLD 阶段 2 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 6.15. TIMG_WDTCONFIG5_REG (0x005C)

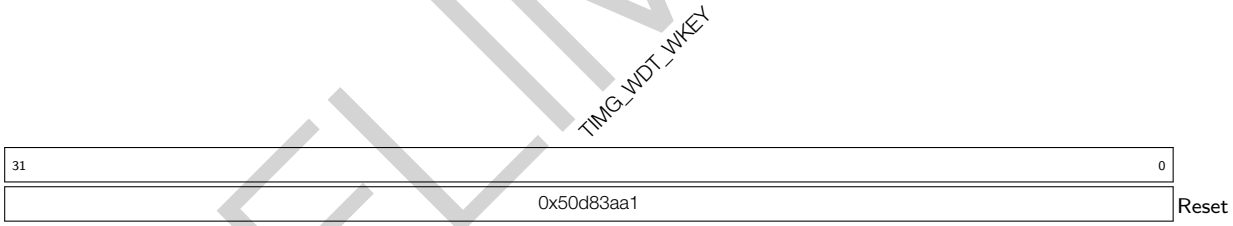
TIMG_WDT_STG3_HOLD 阶段 3 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 6.16. TIMG_WDTFEED_REG (0x0060)



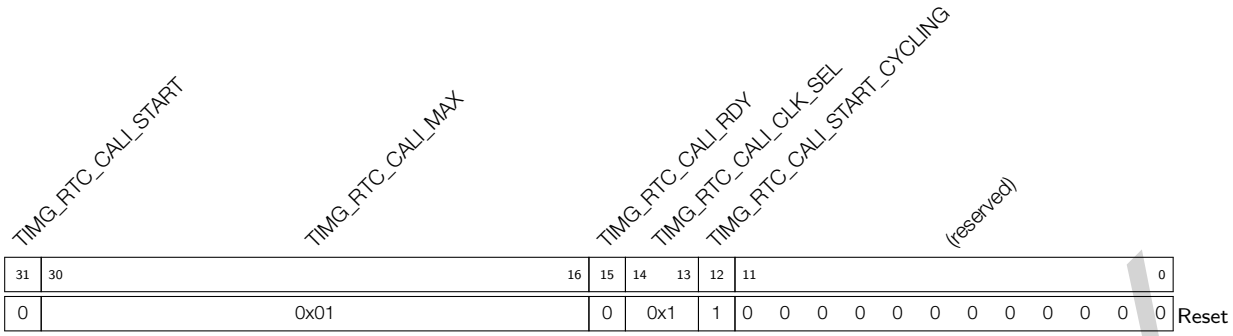
TIMG_WDT_FEED 写任意值喂 MWDT。(WT)

Register 6.17. TIMG_WDTWPROTECT_REG (0x0064)



TIMG_WDT_WKEY 如果寄存器的值与复位值不同，写保护使能。(R/W)

Register 6.18. TIMG_RTCCALICFG_REG (0x0068)



TIMG_RTC_CALI_START_CYCLING 使能周期性频率计算。(R/W)

TIMG_RTC_CALI_CLK_SEL 选择待校准时钟。0：RTC_CLK；1：RTC20M_D256_CLK；2：XTAL32K_CLK。(R/W)

TIMG_RTC_CALI_RDY 标记频率计算完成。(RO)

TIMG_RTC_CALI_MAX 配置频率计算时间。(R/W)

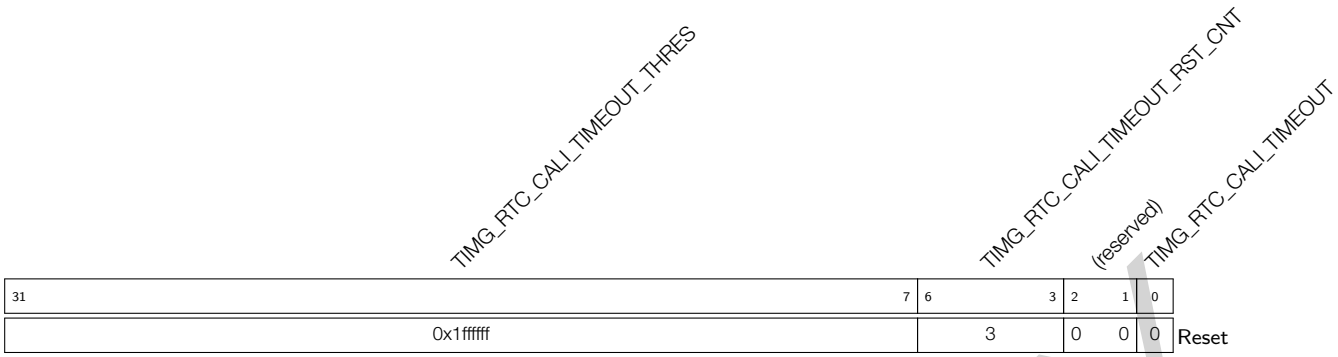
TIMG_RTC_CALI_START 使能单次频率计算。(R/W)

Register 6.19. TIMG_RTCCALICFG1_REG (0x006C)

TIMG_RTC_CALI_CYCLING_DATA_VLD 周期性频率计算结束标志。(RO)

TIMG_RTC_CALI_VALUE 频率计算结果。(RO)

Register 6.20. TIMG_RTCCALICFG2_REG (0x0080)

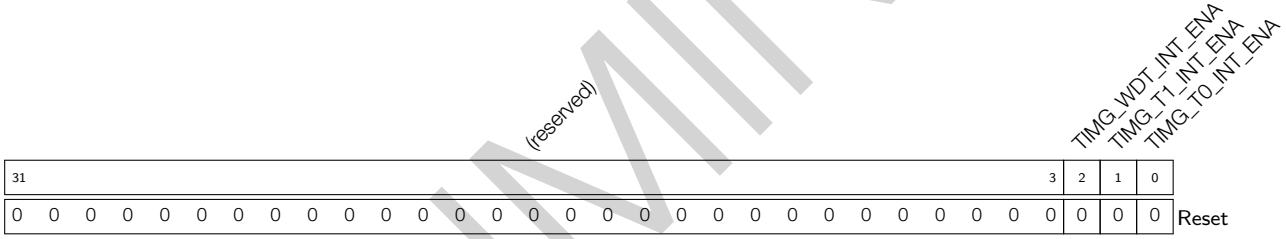


TIMG_RTC_CALI_TIMEOUT 提示时钟频率计算超时。(RO)

TIMG_RTC_CALI_TIMEOUT_RST_CNT 频率计算超时复位周期。(R/W)

TIMG_RTC_CALI_TIMEOUT_THRES RTC 频率计算定时器的阈值。频率计算定时器的值超过此值时触发超时。(R/W)

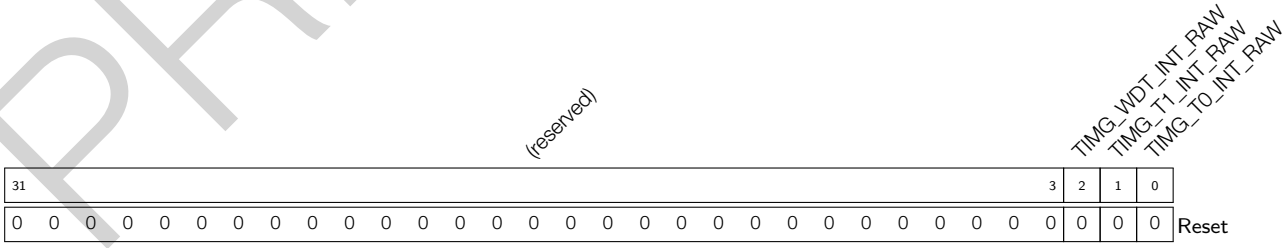
Register 6.21. TIMG_INT_ENA_TIMERS_REG (0x0070)



TIMG_T*x*_INT_ENA TIMG_T*x*_INT 中断的使能位。(R/W)

TIMG_WDT_INT_ENA TIMG_WDT_INT 中断的使能位。(R/W)

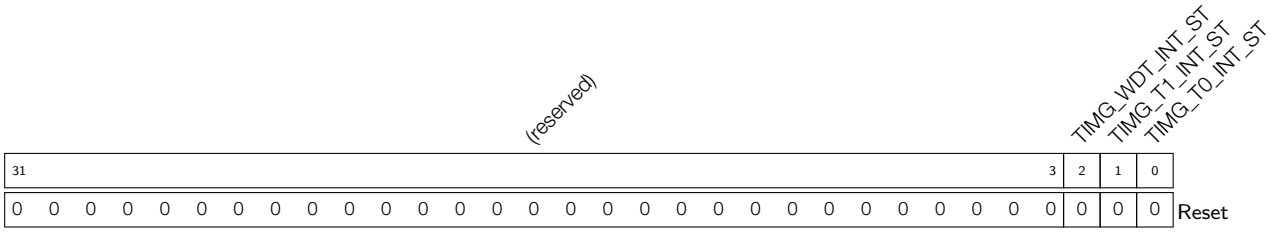
Register 6.22. TIMG_INT_RAW_TIMERS_REG (0x0074)



TIMG_T*x*_INT_RAW TIMG_T*x*_INT 中断的原始中断状态位。(R/WTC/SS)

TIMG_WDT_INT_RAW TIMG_WDT_INT 中断的原始中断状态位。(R/WTC/SS)

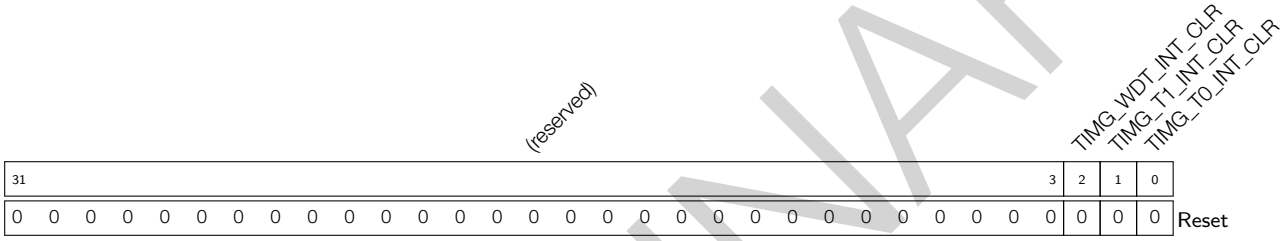
Register 6.23. TIMG_INT_ST_TIMERS_REG (0x0078)



TIMG_T*x*_INT_ST TIMG_T*x*_INT 中断的屏蔽中断状态位。(RO)

TIMG_WDT_INT_ST TIMG_WDT_INT 中断的屏蔽中断状态位。(RO)

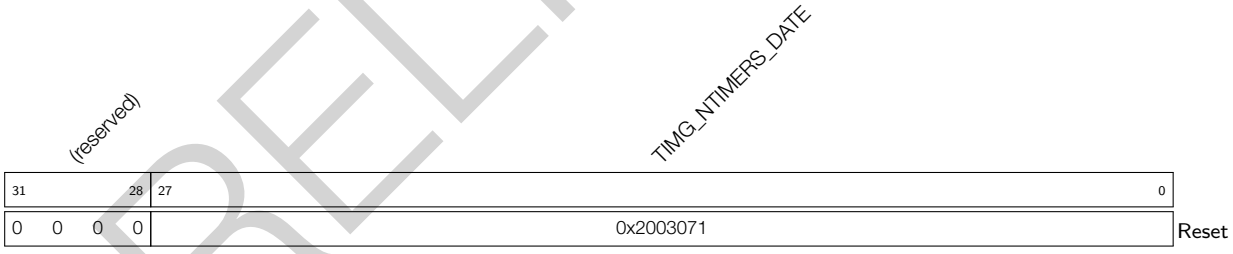
Register 6.24. TIMG_INT_CLR_TIMERS_REG (0x007C)



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

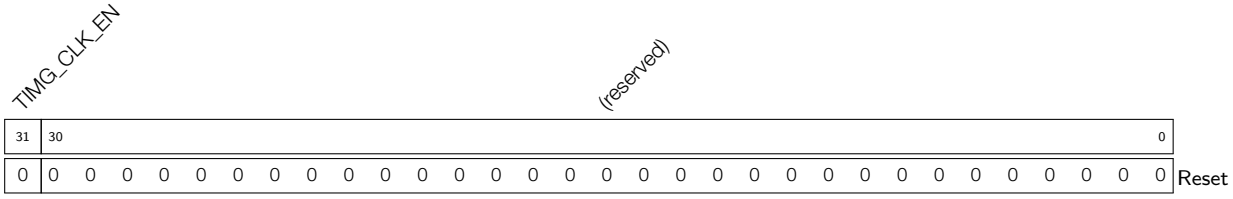
TIMG_WDT_INT_CLR 置位此位，清除 TIMG_WDT_INT 中断。(WT)

Register 6.25. TIMG_NTIMERS_DATE_REG (0x00F8)



TIMG_NTIMERS_DATE 版本控制寄存器。(R/W)

Register 6.26. TIMG_REGCLK_REG (0x00FC)



TIMG_CLK_EN 寄存器时钟门控信号。0：仅在软件运行时打开读写寄存器所需的时钟；1：一直开启软件读写寄存器所需时钟。(R/W)

7 看门狗定时器

7.1 概述

看门狗定时器是一种硬件定时器，用于检测和修复故障。软件必须定期喂狗（复位），以防超时。系统或软件若出现不可预知的问题（比如软件卡在某个循环或逾期事件中）将无法按时喂狗，造成看门狗超时。因此，看门狗定时器有助于检测、处理系统或软件的错误行为。

如图 7-1 所示，ESP32-S3 中有三个数字看门狗定时器：章节 6 定时器组 *(TIMG)* 描述的两个定时器组中各有一个（称作主系统看门狗定时器，缩写为 MWDT），RTC 模块中有一个（称作 RTC 看门狗定时器，缩写为 RWDT）。数字看门狗在运行期间会经历四个阶段（除非看门狗按时喂狗或者处于关闭状态），每个阶段均可配置单独的超时时间和超时动作，其中 MWDT 支持中断、CPU 复位和内核复位三种超时动作，RWDT 支持中断、CPU 复位、内核复位和系统复位四种超时动作（详见章节 7.2.2.2 阶段与超时动作）。每个阶段的超时时间都可单独设置。

在 flash 引导模式下，RWDT 和定时器组 0 的 MWDT 会默认使能，以检测引导过程中发生的错误，并恢复运行。

ESP32-S3 中还有一个模拟看门狗定时器——超级看门狗 (SWD)。超级看门狗是模拟域的超低功耗电路，可以防止系统在数字电路异常状态下运行，并在必要时复位系统。

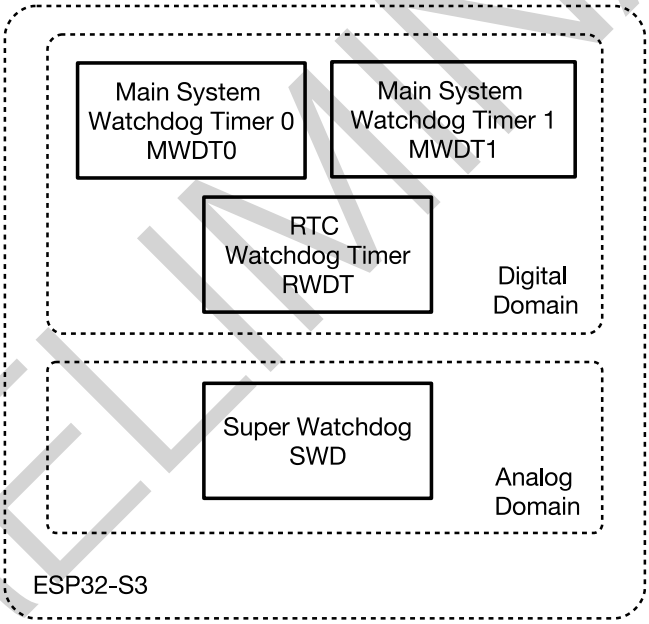


图 7-1. 看门狗定时器概览

请注意，本章节仅包含看门狗定时器的功能描述，其寄存器部分详见章节 6 定时器组 *(TIMG)* 和章节 14 低功耗管理 *(RTC_CNTL) [to be added later]*。

7.2 数字看门狗定时器

7.2.1 主要特性

看门狗定时器具有如下特性：

- 四个阶段，每个阶段都可配置超时时间。每阶段都可单独配置、使能和关闭

- 如在某个阶段发生超时，MWDT 会采取中断、CPU 复位和内核复位中的一种超时动作，RWDT 则会采取中断、CPU 复位、内核复位和系统复位中的一种超时动作
- 32 位超时计数器
- 写保护，防止 RWDT 和 MWDT 配置误改动
- Flash 启动保护
  如果在预定时间内 SPI flash 的引导过程没有完成，看门狗会重启整个主系统

7.2.2 功能描述

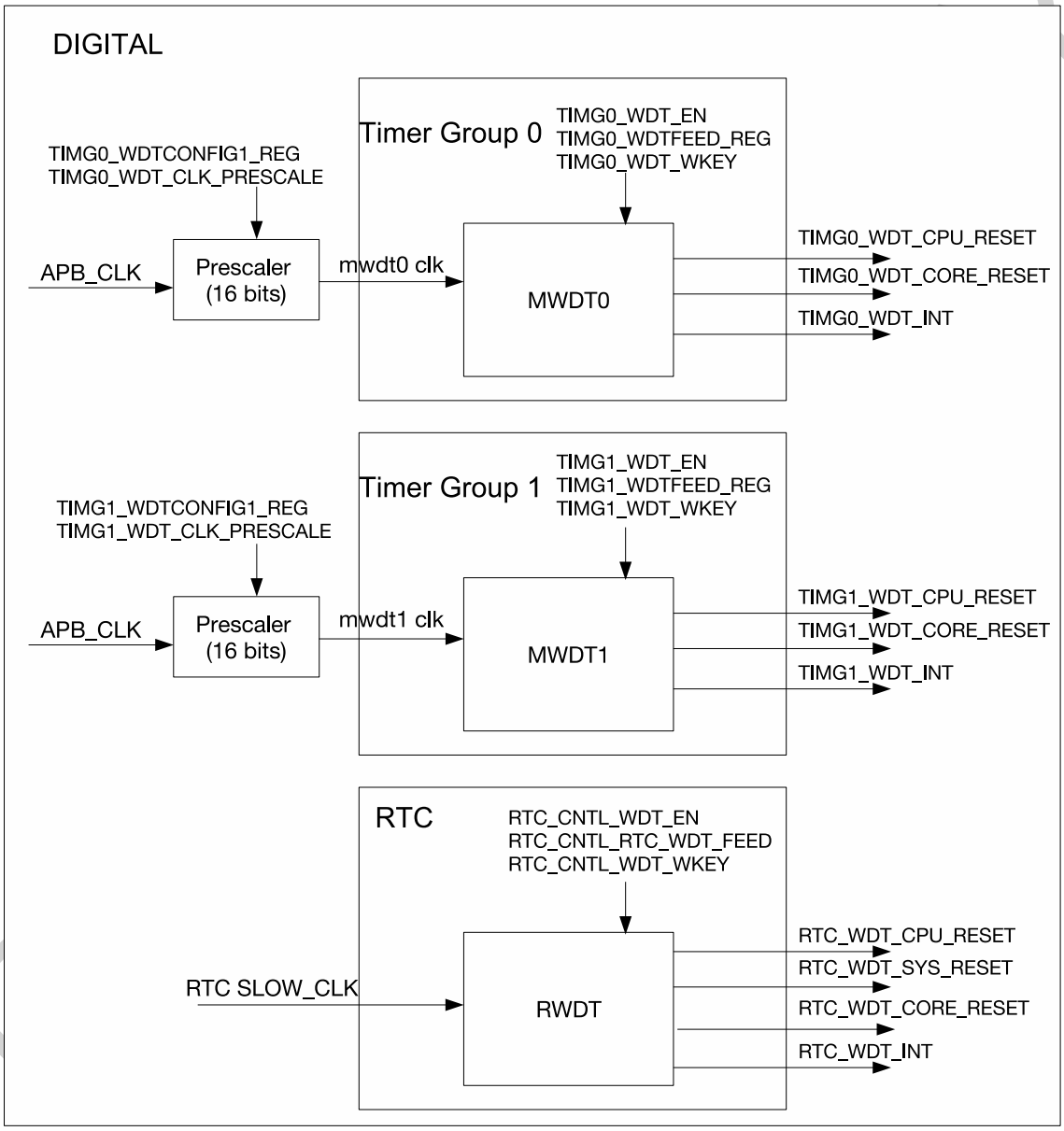


图 7-2. ESP32-S3 的看门狗定时器

图 7-2 为 ESP32-S3 数字系统中的三个看门狗定时器。

7.2.2.1 时钟源与 32 位计数器

每个看门狗定时器的核心是一个 32 位计数器。APB 时钟经过可配置的 16 位预分频器后会得到 MWDT 的时钟源。RWDT 的时钟源则直接取自于 RTC 慢速时钟（RTC 慢速时钟源详见章节 3 复位和时钟）。MWDT 的 16 位预分频器可通过 TIMG_WDTCONFIG1_REG 寄存器的 TIMG_WDT_CLK_PRESCALE 字段配置。

MWDT 和 RWDT 看门狗可分别通过设置 TIMG_WDT_EN 和 RTC_CNTL_WDT_EN 字段使能。看门狗使能后，其内部 32 位计数器的值会在每个时钟源周期内累加 1，直到达到该阶段的超时时间（即在该阶段发生超时）。如发生超时，计数器的值会重置为 0，同时看门狗进入下一阶段。如果软件在规定的时间内成功喂狗，看门狗定时器会回到阶段 0，并将计数器的值重置为 0。软件向 TIMG_WDTFEED_REG 和 RTC_CNTL_RTC_WDT_FEED 寄存器内写入任意值，便可分别为 MDWT 和 RWDT 喂狗。

7.2.2.2 阶段与超时动作

定时器在各阶段可以配置不同的超时时间和对应的超时动作。某一阶段超时会触发对应的超时动作，同时计数器的值被重置为 0，看门狗进入下一阶段。MWDT 和 RWDT 有四个阶段（称为阶段 0 至阶段 3）。看门狗定时器会循环工作（即从阶段 0 至阶段 3，再回到阶段 0）。

MWDT 每个阶段的超时时间可用 TIMG_WDTCONFIG*i*_REG（i 的范围是 2 到 5）寄存器配置，RWDT 的超时时间可用 RTC_CNTL_WDT_STG*j*_HOLD（j 的范围是 0 到 3）字段配置。

值得注意的是，RWDT 在阶段 0 的超时时间 (T_{hold0}) 受 eFuse 寄存器 EFUSE_RD_REPEAT_DATA1_REG 的 EFUSE_WDT_DELAY_SEL 字段和 RTC_CNTL_WDT_STG0_HOLD 字段共同影响，关系如下：

$$T_{hold0} = RTC\_CNTL\_WDT\_STG0\_HOLD << (EFUSE\_WDT\_DELAY\_SEL + 1)$$

其中，<< 为左移运算符。

如某个阶段超时，下列超时动作之一将会执行：

- 触发中断
  如阶段超时，中断被触发。
- CPU 复位 – 复位 CPU 核心
  如阶段超时，复位 CPU 核心。
- 内核复位 – 复位主系统
  如阶段超时，主系统（包括 MWDT、CPU 和所有外设）复位。功耗管理单元和 RTC 外设不会复位。
- 系统复位 – 复位主系统、功耗管理单元和 RTC 外设
  如阶段超时，主系统、功耗管理单元和 RTC 外设（详见章节 14 低功耗管理 *(RTC_CNTL) [to be added later]*）同时复位。此动作仅可在 RWDT 中实现。
- 关闭
  该阶段对系统不产生影响。

MWDT 所有阶段的超时动作均在 TIMG_WDTCONFIG0_REG 寄存器中配置。RWDT 的超时动作可在 RTC_CNTL_WDTCONFIG0_REG 寄存器配置。

7.2.2.3 写保护

看门狗定时器对于检测和处理系统或软件错误而言至关重要，不应轻易关闭（例如，因写寄存器位置错误而误将看门狗关闭）。因此，MWDT 和 RWDT 引入写保护机制，防止看门狗因无意的写操作而被关闭或篡改。

写保护机制通过每个看门狗定时器的写密钥字段运行（MWDT 看门狗使用 TIMG_WDT_WKEY，RWDT 看门狗使用 RTC_CNTL_WDT_WKEY）。必须向看门狗定时器的写密钥字段写入 0x50D83AA1，才能修改其它看门狗寄存器。如果写密钥字段的值不是 0x50D83AA1，任何试图向看门狗定时器寄存器（除了向写密钥字段本身）写值的操作都会被忽略。推荐按以下步骤访问看门狗定时器：

1. 将 0x50D83AA1 写入看门狗定时器的写密钥字段，关闭写保护。
2. 根据需要修改看门狗，如喂狗或改变配置。
3. 向看门狗定时器的写密钥字段上写入除 0x50D83AA1 以外的任意值，重新使能写保护。

7.2.2.4 Flash 引导保护

在 flash 引导模式下，定时器组 0（见图 6-1 定时器组）的 MWDT 和 RWDT 会默认使能。MWDT 的阶段 0 的默认超时动作为系统复位。RWDT 的阶段 0 超时动作也是系统复位（复位主系统和 RTC）。引导后，应将 TIMG_WDT_FLASHBOOT_MOD_EN 和 RTC_CNTL_WDT_FLASHBOOT_MOD_EN 位清零，分别关闭 MWDT 和 RWDT 的 flash 引导保护。然后，软件可以配置 MWDT 和 RWDT。

7.3 模拟看门狗定时器

超级看门狗 (SWD) 是模拟域的超低功耗电路，可以防止系统在数字电路异常状态下运行，并在必要时复位系统。SWD 包含一个看门狗电路，需在每个超时阶段（约不足一秒）至少喂狗一次。该电路会在看门狗超时时间约 100 ms 之前发送 WD_INTR 信号提醒系统喂狗。

如果系统不回应 SWD 的喂狗请求，看门狗超时，SWD 会产生系统电平信号 SWD_RSTB，复位芯片上的整个数字电路。

7.3.1 主要特性

SWD 具有如下特性：

- 超低功耗
- 用中断提醒 SWD 即将超时
- 软件有多种专用的方法喂 SWD，让 SWD 监控整个操作系统的工作状态

7.3.2 SWD 控制器

7.3.2.1 结构

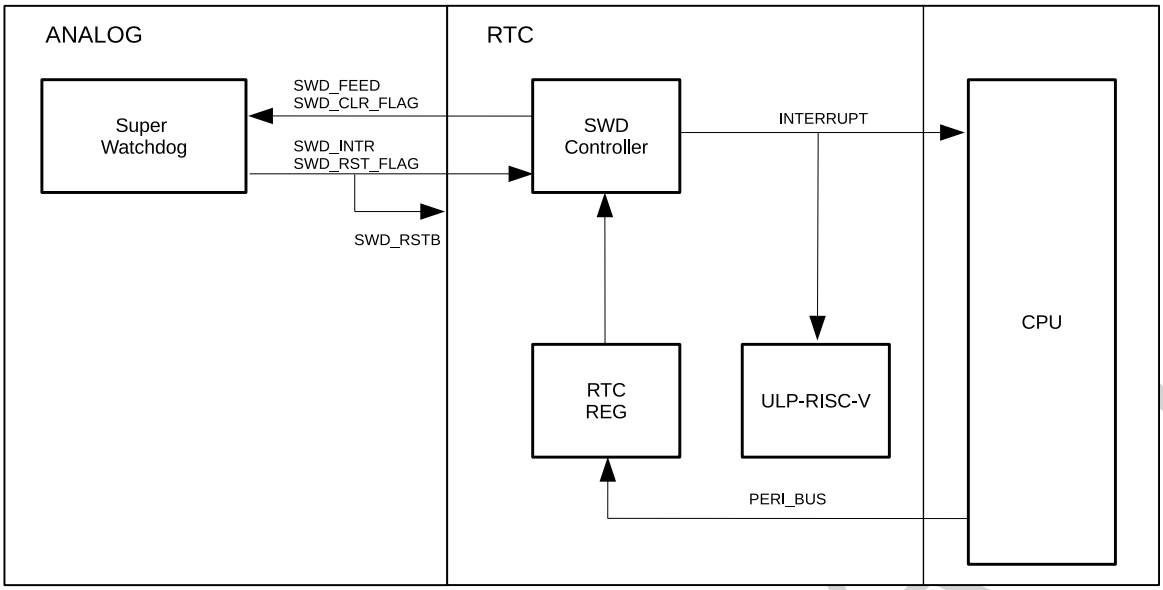


图 7-3. SWD 控制器结构

7.3.2.2 工作流程

正常状态下：

- SWD 控制器收到 SWD 的喂狗请求。
- SWD 控制器可以向主 CPU 或 ULP-RISC-V 发送中断。
- 主 CPU 可以决定是通过置位 RTC_CNTL_SWD_FEED 直接喂狗，还是发送中断让 ULP-RISC-V 置位该字段喂狗。
- CPU 或 ULP-RISC-V 喂狗时，需要先向 RTC_CNTL_SWD_WKEY 写 0x8F1D312A 关闭 SWD 控制器的写保护。这样做可以防止系统在数字电路异常状态下运行时误喂 SWD。
- 如将 RTC_CNTL_SWD_AUTO_FEED_EN 置 1，SWD 控制器也可配置为在不需要 CPU 或 ULP-RISC-V 干预的情况下喂 SWD。

复位后：

- 可查看 RTC_CNTL_RESET_CAUSE_PROCPU[5:0] 获知 CPU 复位原因。
  如 RTC_CNTL_RESET_CAUSE_PROCPU[5:0] == 0x12，则表示上一次复位的原因是 SWD 复位。
- 置位 RTC_CNTL_SWD_RST_FLAG_CLR 清除 SWD 复位标志。

7.4 中断

看门狗定时器中断，请前往章节 6 定时器组 *(TIMG)* 的第 6.2.6 节 中断 查看。

7.5 寄存器

MWDT 寄存器是定时器组模块的一部分，在章节 6 定时器组 *(TIMG)* 的第 6.4 节 寄存器列表 中有详细描述。
RWDT 和 SWD 寄存器是 RTC 模块的一部分，在章节 14 低功耗管理 *(RTC_CNTL) [to be added later]* 的第 20 节

寄存器列表 中有详细描述。

8 XTAL32K 看门狗定时器 (XTWDT)

ESP32-S3 的 XTAL32K 看门狗定时器是用于检测 XTAL32K_CLK 时钟的工作状态，有 XTAL32K_CLK 停振监测，切换 RTC 时钟源等功能。当外部晶振 XTAL32K_CLK 作为 RTC 的 SLOW_CLK 源（时钟描述详见章节 3 复位和时钟），若 XTAL32K_CLK 时钟停振，XTAL32K 看门狗定时器会将 XTAL32K_CLK 替换为 RTC_CLK 的分频时钟 BACKUP32K_CLK 并发送中断 (若芯片处于 Light-sleep 和 Deep-sleep 状态则唤醒 CPU)，由软件重启 XTAL32K_CLK，并切回。

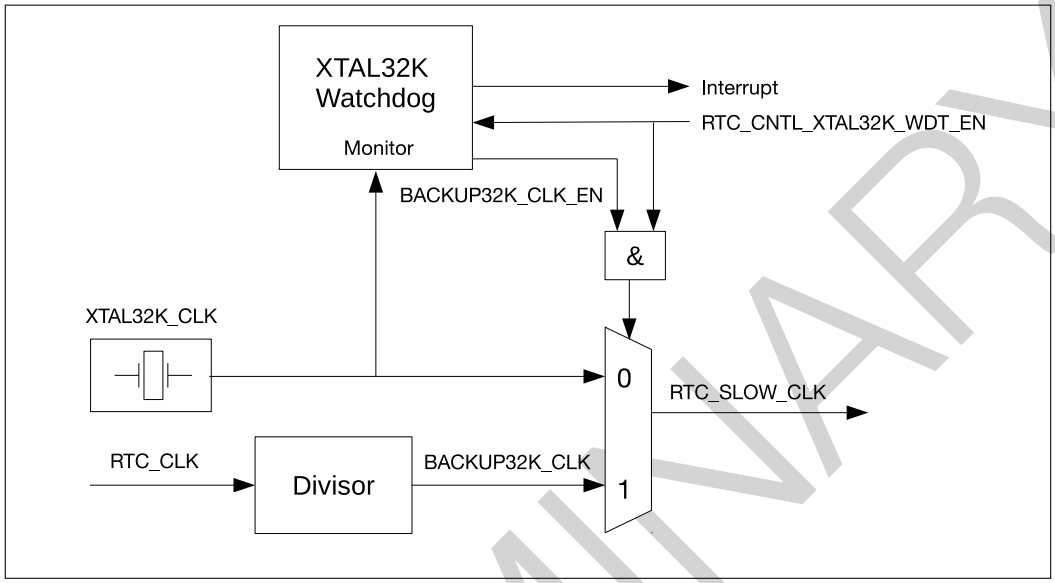


图 8-1. XTAL32K 看门狗定时器

8.1 主要特性

8.1.1 XTAL32K 看门狗定时器的中断及唤醒

XTAL32K 看门狗定时器监控到 XTAL32K_CLK 停振时，将发起停振中断 RTC_XTAL32K_DEAD_INT（中断描述详见章节 14 低功耗管理 *(RTC_CNTL) [to be added later]*)，，如果 CPU 处于 Light-sleep 和 Deep-sleep 状态，将唤醒 CPU。

8.1.2 BACKUP32K_CLK

XTAL32K 看门狗定时器监控到 XTAL32K_CLK 停振后，将使用 RTC_CLK 的分频时钟 BACKUP32K_CLK (频率约为 32 kHz) 替代 XTAL32K_CLK 作为 RTC 的 SLOW_CLK 维持系统继续正常工作。

8.2 功能描述

8.2.1 工作流程

1. 使能 RTC_CNTL_XTAL32K_WDT_EN，XTAL32K 看门狗定时器将由空闲状态转入计数状态，看门狗的计数器（工作时钟为 RTC_CLK）在检测到 XTAL_32K 的时钟上升沿时将被清零，否则将持续计数；当计数器的值达到 RTC_CNTL_XTAL32K_WDT_TIMEOUT 时，发出中断/唤醒信号，随后计数器复位。
2. 如果 RTC_CNTL_XTAL32K_AUTO_BACKUP 已置 1 且步骤 1 已完成，XTAL32K 看门狗定时器会自动开启 BACKUP32K_CLK，替换 RTC 的慢速时钟源 RTC SLOW_CLK，保证系统能够正常运行，以及工作在

RTC 慢速时钟 RTC SLOW_CLK 的定时器 (如 RTC_TIMER 等) 能够保持计时准确性。时钟频率的配置参见 8.2.2。

3. 软件通过 RTC_CNTL_XPD_XTAL_32K 位开关 XTAL32K_CLK 的 XPD（no power-down 的缩写，意为不关闭电源）信号来重启 XTAL32K_CLK，然后通过将 RTC_CNTL_XTAL32K_WDT_EN 位设置 0（BACKUP32K_CLK_EN 会随之自动清零），RTC 的 SLOW_CLK 时钟源将从 BACKUP32K_CLK 切回到 XTAL32K_CLK。若是芯片处于 Light-sleep 和 Deep-sleep 状态，则 XTAL32K 看门狗定时器将唤醒 CPU，完成上述操作。

8.2.2 BACKUP32K_CLK 实现原理

由于 RTC_CLK 的时钟频率存在芯片差异，所以为保证 BACKUP32K_CLK 生效期间，RTC_TIMER 等使用 RTC 的 SLOW_CLK 工作的定时器依然能够准确计时，需要根据 RTC_CLK（详见章节 *14 低功耗管理 (RTC_CNTL) [to be added later]*）的实际频率，可通过配置 RTC_CNTL_XTAL32K_CLK_FACTOR_REG 调整 BACKUP32K_CLK 的分频系数。该寄存器的每个字节对应一个分频因子 (x_0 ~x_7)。BACKUP32K_CLK 的分频器是一个分母恒为 4 的小数分频器，算法如下:

$$f\_back\_clk/4 = f\_rtc\_clk/S$$

$$S = x_0 + x_1 + ... + x_7$$

其中 f_back_clk 为分频后的 BACKUP32K_CLK 目标频率为 32.768 kHz；f_rtc_clk 为当前 RTC_CLK 的实际频率；x_0 ~x_7 分别对应四个 BACKUP32K 时钟信号的高低电平的脉宽，单位为 RTC_CLK 的周期.

8.2.3 BACKUP32K_CLK 分频因子配置方法

根据 8.2.2 小节的分频原理描述，可以通过以下步骤计算并完成分频因子的配置:

- 根据 RTC_CLK 的频率以及 BACKUP32K 的目标分频频率计算出分频因子的总和 S;
- 计算出分频器的整数部分，$N = f\_rtc\_clk/f\_back\_clk$;
- 因为 BACKUP32K 的分频因子是单个脉宽 (高或低电平)，所以需要将分频系数的整数部分分成两份，计算分频因子的整数部分，$M = N/2$;
- 根据 M 和 S 确定 xn = M 以及 xn = M + 1 的个数，M + 1 即为分频因子的小数部分。

例如，RTC_CLK 的时钟频率为 163 kHz，则 $f\_rtc\_clk = 163000$，$f\_back\_clk = 32768$，$S = 20$，$M = 2$，所以满足条件的 $\{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\} = \{2, 3, 2, 3, 2, 3, 2, 3\}$，BACKUP32K_CLK 的时钟频率为 32.6 kHz。

9 SHA 加速器 (SHA)

9.1 概述

ESP32-S3 内置 SHA（安全哈希算法）硬件加速器可完成 SHA 运算，具有 Typical SHA 和 DMA-SHA 两种工作模式。整体而言，相比基于纯软件的 SHA 运算，SHA 硬件加速器能够极大地提高运算速度。

9.2 主要特性

ESP32-S3 的 SHA 硬件加速器：

- 支持 FIPS PUB 180-4 规范的全部运算标准
  - SHA-1 运算
  - SHA-224 运算
  - SHA-256 运算
  - SHA-384 运算
  - SHA-512 运算
  - SHA-512/224 运算
  - SHA-512/256 运算
  - SHA-512/*t* 运算
- 提供两种工作模式
  - Typical SHA 工作模式
  - DMA-SHA 工作模式
- 允许插入 (interleaved) 功能（仅限 Typical SHA 工作模式）
- 允许中断功能（仅限 DMA-SHA 工作模式）

9.3 工作模式简介

ESP32-S3 内置的 SHA 加速器支持两种工作模式。

- Typical SHA 工作模式：所有数据读写统一通过 CPU 访问完成。
- DMA-SHA 工作模式：所有读数据通过硬件上的 DMA 完成。具体来说，用户可配置 DMA 控制器，由 DMA 控制器提供 SHA 运算过程中所需的数据信息。因此，可以释放 CPU 执行其他任务。

用户可通过配置 SHA_START_REG 或 SHA_DMA_START_REG 选择 SHA 加速器的工作模式，先配置的工作模式生效，具体请见表 9-1。

表 9-1. 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 9-2。

表 9-2. 运算标准选择

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

注意：

ESP32-S3 的数字签名 (DS) 和 HMAC 模块也会调用 SHA 加速器。此时，用户无法正常访问 SHA 加速器。

9.4 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。

9.4.1 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。

9.4.1.1 附加填充比特

SHA 加速器仅能处理长度为 512 位及其整倍数或 1024 位及其整倍数的信息。因此，在将信息送至 SHA 加速器进行运算前，应先通过软件操作将信息填充为符合要求的长度。

假设待处理信息 M 的长度为 m 位，则针对不同运算标准的填充步骤见下：

- **SHA-1、SHA-224 和 SHA-256**
  1. 首先，在待处理信息后填充 1 个 “1”；
  2. 随后，再填充 k 个 “0”。其中，k 为满足 $m + 1 + k \equiv 448 \ mod \ 512$ 的最小非负数解；
  3. 最后，在末尾填充一个 64 位的信息块。该信息块的内容为用二进制表示的待处理信息的长度，即 m 的值。
- **SHA-384、SHA-512、SHA-512/224、SHA-512/256 和 SHA-512/*t***
  1. 首先，在待处理信息后填充 1 个 “1”；
  2. 随后，再填充 k 个 “0”。其中，k 为满足 $m + 1 + k \equiv 896 \ mod \ 1024$ 的最小非负数解；
  3. 最后，在末尾填充一个 128 位的信息块。该信息块的内容为用二进制表示的待处理信息的长度，即 m 的值。

更多详情，请参考 FIPS PUB 180-4 规范 中的 “5.1 Padding the Message” 章节。

9.4.1.2 信息解析

在完成信息填充后，我们还需将待处理信息（及其填充）解析为 N 个 512 位或 1024 位的信息块。

- 对于 **SHA-1**、**SHA-224** 和 **SHA-256**：待处理信息（及其填充）应解析为 N 个 512 位的信息块，即 $M^{(1)}$、$M^{(2)}$、...、$M^{(N)}$。一个 512 位信息块包括 16 个 32 位的字 (word)，则第 i 个信息块的第一个 32 位字表示为 $M_0^{(i)}$，第二个 32 位字表示为 $M_1^{(i)}$，...，第 16 个 32 位字表示为 $M_{15}^{(i)}$。
- 对于 **SHA-384**、**SHA-512**、**SHA-512/224**、**SHA-512/256** 和 **SHA-512/t**：待处理信息（及其填充）应解析为 N 个 1024 位的信息块，即 $M^{(1)}$、$M^{(2)}$、...、$M^{(N)}$。一个 1024 位信息块包括 16 个 64 位的字 (word)，则第 i 个信息块的第一个 64 位字表示为 $M_0^{(i)}$，第二个 64 位字表示为 $M_1^{(i)}$，...，第 16 个 64 位字表示为 $M_{15}^{(i)}$。

SHA 加速器在工作时，每次处理的信息块数据均将按照如下规则写入相应的寄存器中：

- **SHA-1**、**SHA-224**、**SHA-256** 将 $M_0^{(i)}$ 存放在 SHA_M_0_REG 中，$M_1^{(i)}$ 存放在 SHA_M_1_REG，...，$M_{15}^{(i)}$ 存放在 SHA_M_15_REG 中。
- **SHA-384**、**SHA-512**、**SHA-512/224**、**SHA-512/256** 将 $M_0^{(i)}$ 的高 32 位存放在 SHA_M_0_REG 中，低 32 位存放在 SHA_M_1_REG，$M_1^{(i)}$ 的高 32 位存放在 SHA_M_2_REG 中，低 32 位存放在 SHA_M_3_REG，...，$M_{15}^{(i)}$ 的高 32 位存放在 SHA_M_30_REG 中，低 32 位存放在 SHA_M_31_REG。

说明：
有关“信息块”及相关概念的描述，请参考 FIPS PUB 180-4 规范 中“2.1 Glossary of Terms and Acronyms”章节。

9.4.1.3 哈希初始值 (Initial Hash Value)

在进行哈希运算前，首先必须设置哈希初始值 $H^{(0)}$。不同运算标准的哈希初始值设置要求不同，其中 SHA-1、SHA-224、SHA-256、SHA-384、SHA-512、SHA-512/224、SHA-512/256 等运算的哈希初始值为常量 C，且已经固定在硬件中，无需专门计算。

然而，SHA-512/t 对于不同的 t 均需要一个不同的哈希初始值。简单来说，SHA-512/t 是一种基于 SHA-512 的 t 位运算标准，其运算结果将截断至 t 位。其中，运算标准对 t 值的要求为“大于 0 小于 512 且不等于 384 的正整数”。对于不同 t 值的 SHA-512/t 运算，其哈希初始值可通过对“SHA-512/t”字符串的十六进制表示进行 SHA-512 运算获得。不难看出，对于 t 取值不同的 SHA-512/t 运算标准，其不同之处仅在于 t 值不同。

因此，为了简化 SHA-512/t 的哈希初始值计算，我们特别提出了以下方法：

1. **计算 t_string 和 t_length**：其中，t_string 为 t 的字符串信息，长度为 32-bit。t_length 指明字符串长度信息，长度为 7-bit。根据 t 的取值范围不同，t_string 和 t_length 的计算过程如下：
   - 如果 $1 <= t <= 9$，则 $t\_length = 7'h48$，t_string 需要按照如下格式封装：

$8'h30 + 8'ht_0$	$1'b1$	$23'b0$

     其中，$t_0 = t$。

     举例，如果 t = 8，则 t_0 = 8，$t\_string = 32'h38800000$。
   - 如果 $10 <= t <= 99$，则 $t\_length = 7'h50$，t_string 需要按照如下格式封装：

$8'h30 + 8'ht_1$	$8'h30 + 8'ht_0$	$1'b1$	$15'b0$

其中，$t_0 = t\%10$，$t_1 = t/10$。

举例，如果 t = 56，则 t_0 = 6，t_1 = 5，t_string = $32'h35368000$。

- 如果 $100 <= t < 512$，则 $t\_length = 7'h58$，t_string 需要按照如下格式封装:

$8'h30 + 8'ht_2$	$8'h30 + 8'ht_1$	$8'h30 + 8'ht_0$	$1'b1$	$7'b0$

其中，$t_0 = t\%10$，$t_1 = (t/10)\%10$，$t_2 = t/100$。

举例，如果 t = 231，则 t_0 = 1，t_1 = 3，t_2 = 2，t_string = $32'h32333180$。

2. **配置计算哈希初始值所需的寄存器**：用 t_string 和 t_length 初始化文本寄存器 SHA_T_STRING_REG 和 SHA_T_LENGTH_REG。
3. **计算得到哈希初始值**：对 SHA_MODE_REG 寄存器置 7 选择 SHA-512/t 运算，并对 SHA_START_REG 寄存器置 1，启动 SHA 加速器的运算即可。最后，轮询寄存器 SHA_BUSY_REG 结果为 0，则哈希初始值已计算完毕。

此外，您也可以按照 FIPS PUB 180-4 Spec 中 “5.3.6 SHA-512/t” 章节的描述计算 SHA-512/t 的哈希初始值，也就是对 “SHA-512/t” 字符串的十六进制表示进行一次 “特殊” 的 SHA-512 运算，其运算得到的信息摘要即为所需的哈希初始值。这里的 “特殊” 指本次 SHA-512 运算的哈希初始值为 “SHA-512 运算标准的初始值常量 C 与 0xa5 每 8 位进行一次异或位运算后得到的结果”。

9.4.2 哈希运算流程

在完成信息预处理后，ESP32-S3 SHA 加速器将正式开始哈希运算，最终根据不同运算标准得到不同长度的信息摘要。正如上文所述，ESP32-S3 SHA 加速器支持 Typical SHA 和 DMA-SHA 两种工作模式，下面将对这两种工作模式的具体流程进行介绍。

9.4.2.1 Typical SHA 模式下的运算流程

通常情况下，ESP32-S3 的 SHA 会处理完当前信息的所有信息块并生成该信息的信息摘要，之后再开始计算新的信息摘要。不过，ESP32-S3 SHA 加速器在 Typical SHA 工作模式下还支持 “interleaved” 运算，即在每次运算全部完成前，允许插入其他运算任务。具体来说，在计算完每一个信息块后，用户都可以将存储在 SHA_H_*n*_REG 寄存器中的信息摘要暂存起来，然后插入优先级更高的运算任务，包括 Typical SHA 运算和 DMA-SHA 运算。当临时任务结束后，再将之前暂存的信息摘要重新写入 SHA_H_*n*_REG 中，并继续完成之前中断的计算。

Typical SHA 的具体运算流程（SHA-512/t 除外）

1. 选择运算标准。
   - 配置 SHA_MODE_REG 寄存器，设置运算标准。具体配置，请参考表 9-2。
2. 处理当前信息块。
   - 将当前信息块写入 SHA_M_*n*_REG 寄存器。
3. 启动 SHA 加速器 [1]。

- 如果为首次运算，则对 SHA_START_REG 寄存器置 1，启动 SHA 加速器的运算。此时，SHA 加速器按照步骤 1 中选定的运算标准，使用硬件中固定的哈希初始值进行运算；
- 如果非首次运算[2]，则对 SHA_CONTINUE_REG 寄存器置 1，启动 SHA 加速器的运算。此时，SHA 加速器使用 SHA_H_*n*_REG 寄存器中的值作为哈希初始值进行运算。

4. 查询当前信息块的处理进度。
   - 轮询寄存器 SHA_BUSY_REG 一直到读回的值为 0，代表 SHA 硬件加速器已完成对当前信息块的计算，进入“空闲”状态[3]。
5. 选择是否有后续的待处理信息块。
   - 如果存在后续待处理信息块，则跳回执行步骤 2。
   - 否则，继续执行。
6. 获取信息摘要：
   - 从寄存器堆 SHA_H_*n*_REG 取出信息摘要。

Typical SHA 的具体运算流程（SHA-512/*t*）

1. 选择运算标准。
   - 配置 SHA_MODE_REG 寄存器为 7 选择 SHA-512/*t* 运算标准。
2. 计算哈希初始值。
   (a) 配置 t_string 和 t_length，并将其写入 SHA_T_STRING_REG 和 SHA_T_LENGTH_REG 寄存器。具体请见 9.4.1.3 章节。
   (b) 对 SHA_START_REG 寄存器置 1，启动 SHA 加速器的运算。
   (c) 轮询寄存器 SHA_BUSY_REG 一直到读回的值为 0，代表 SHA 硬件加速器已完成哈希初始值的计算。
3. 处理当前信息块[1]。
   - 将当前信息块写入 SHA_M_*n*_REG 寄存器。
4. 启动 SHA 加速器。
   - 对 SHA_CONTINUE_REG 寄存器置 1，启动 SHA 加速器的运算。此时，SHA 加速器使用 SHA_H_*n*_REG 寄存器中的值作为哈希初始值进行运算。
5. 查询当前信息块的处理进度。
   - 轮询寄存器 SHA_BUSY_REG 一直到读回的值为 0，代表 SHA 硬件加速器已完成对当前信息块的计算，进入“空闲”状态[3]。
6. 选择是否有后续的待处理信息块。
   - 如果存在后续待处理信息块，则跳回执行步骤 3。
   - 否则，继续执行。
7. 获取信息摘要：
   - 从寄存器堆 SHA_H_*n*_REG 取出信息摘要。

说明：

1. 这里，在 SHA 加速器进行硬件运算时，如果存在后续待处理信息块，软件还可以同时将后续信息块写入 SHA_M_*n*_REG 寄存器，以节省时间。
2. 比如重新启动 SHA 加速器完成之前暂停任务的情况。
3. 这里，你可以选择是否需要插入其他任务。如需插入，请前往 插入任务工作流程 具体查看。

如上文所述，ESP32-S3 SHA 加速器**支持在 Typical SHA 模式下 "插入" 任务**。

具体工作流程如下。

1. 保存插入前任务的以下数据，准备将 SHA 加速器的使用权移交给插入的任务。
   - 读取并保存寄存器 SHA_MODE_REG 中的运算标准类型。
   - 读取并保存寄存器堆 SHA_H_*n*_REG 中的信息摘要。
2. 执行插入的任务。具体按照插入运行类型的不同，请见 Typical SHA 或 DMA-SHA 工作流程。
3. 恢复插入前任务的以下数据，准备将 SHA 加速器的使用权交还给插入前的任务。
   - 将获得使用权前保存的运算标准类型重新写入寄存器 SHA_MODE_REG;
   - 将获得使用权前保存的信息摘要写入寄存器堆 SHA_H_*n*_REG。
4. 将之前任务的下一个待处理信息块写入 SHA_M_*n*_REG 寄存器，并对 SHA_CONTINUE_REG 寄存器置 1，重新启动 SHA 加速器，完成之前暂停的任务。

9.4.2.2 DMA-SHA 模式下的运算流程

ESP32-S3 SHA 加速器在 DMA-SHA 工作模式下不支持 "interleaved" 运算方法，即在每次运算全部完成前，不允许插入其他运算任务。这种情况下，用户如有插入运算需求，可将较大信息块进行拆分，并进行多次 DMA-SHA 运算。每次 DMA-SHA 运算之间允许插入其他运算标准的计算任务。

与 Typical SHA 不同，SHA 在 DMA-SHA 工作模式下，运算过程中的数据搬运过程均由硬件完成，具体配置可见章节 9 *通用 DMA 控制器 (DMA) [to be added later]*。

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 9-2。
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 9.4.1.3 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。

9.4.3 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 9-6：

表 9-6. 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

[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。

9.4.4 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。

9.5 寄存器列表

本小节的所有地址均为相对于 SHA 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

名称	描述	地址	权限
控制与状态寄存器			
SHA_CONTINUE_REG	继续 SHA 运算（仅用于 Typical SHA 模式）	0x0014	WO
SHA_BUSY_REG	指示 SHA 加速器是否处于“忙碌”状态	0x0018	RO
SHA_DMA_START_REG	启动 SHA 加速器的 DMA-SHA 模式	0x001C	WO
SHA_START_REG	启动 SHA 加速器的 Typical SHA 模式	0x0010	WO
SHA_DMA_CONTINUE_REG	继续 SHA 运算（仅用于 DMA-SHA 模式）	0x0020	WO
SHA_INT_CLEAR_REG	DMA-SHA 中断清除寄存器	0x0024	WO
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			

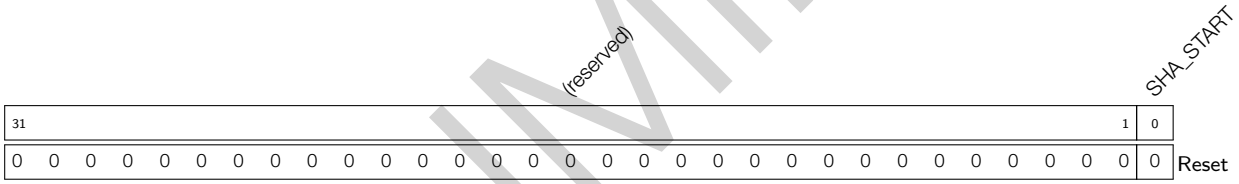
名称	描述	地址	权限
SHA_MODE_REG	配置 SHA 加速器的运算标准	0x0000	R/W
SHA_T_STRING_REG	哈希字符串内容寄存器（仅用于计算 SHA-512/t 的哈希初始值）	0x0004	R/W
SHA_T_LENGTH_REG	哈希字符串长度寄存器（仅用于计算 SHA-512/t 的哈希初始值）	0x0008	R/W
存储器			
SHA_DMA_BLOCK_NUM_REG	信息块个数寄存器（仅用于 DMA-SHA 工作模式）	0x000C	R/W
SHA_H_0_REG	哈希值	0x0040	R/W
SHA_H_1_REG	哈希值	0x0044	R/W
SHA_H_2_REG	哈希值	0x0048	R/W
SHA_H_3_REG	哈希值	0x004C	R/W
SHA_H_4_REG	哈希值	0x0050	R/W
SHA_H_5_REG	哈希值	0x0054	R/W
SHA_H_6_REG	哈希值	0x0058	R/W
SHA_H_7_REG	哈希值	0x005C	R/W
SHA_H_8_REG	哈希值	0x0060	R/W
SHA_H_9_REG	哈希值	0x0064	R/W
SHA_H_10_REG	哈希值	0x0068	R/W
SHA_H_11_REG	哈希值	0x006C	R/W
SHA_H_12_REG	哈希值	0x0070	R/W
SHA_H_13_REG	哈希值	0x0074	R/W
SHA_H_14_REG	哈希值	0x0078	R/W
SHA_H_15_REG	哈希值	0x007C	R/W
SHA_M_0_REG	输入信息	0x0080	R/W
SHA_M_1_REG	输入信息	0x0084	R/W
SHA_M_2_REG	输入信息	0x0088	R/W
SHA_M_3_REG	输入信息	0x008C	R/W
SHA_M_4_REG	输入信息	0x0090	R/W
SHA_M_5_REG	输入信息	0x0094	R/W
SHA_M_6_REG	输入信息	0x0098	R/W
SHA_M_7_REG	输入信息	0x009C	R/W
SHA_M_8_REG	输入信息	0x00A0	R/W
SHA_M_9_REG	输入信息	0x00A4	R/W
SHA_M_10_REG	输入信息	0x00A8	R/W
SHA_M_11_REG	输入信息	0x00AC	R/W
SHA_M_12_REG	输入信息	0x00B0	R/W
SHA_M_13_REG	输入信息	0x00B4	R/W
SHA_M_14_REG	输入信息	0x00B8	R/W
SHA_M_15_REG	输入信息	0x00BC	R/W
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

名称	描述	地址	权限
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

9.6 寄存器

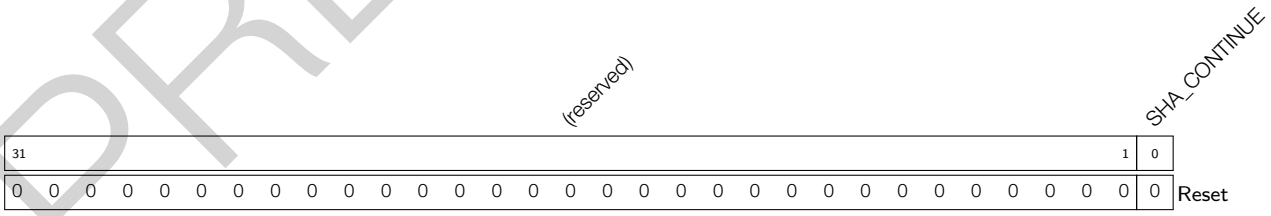
本小节的所有地址均为相对于 SHA 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

Register 9.1. SHA_START_REG (0x0010)



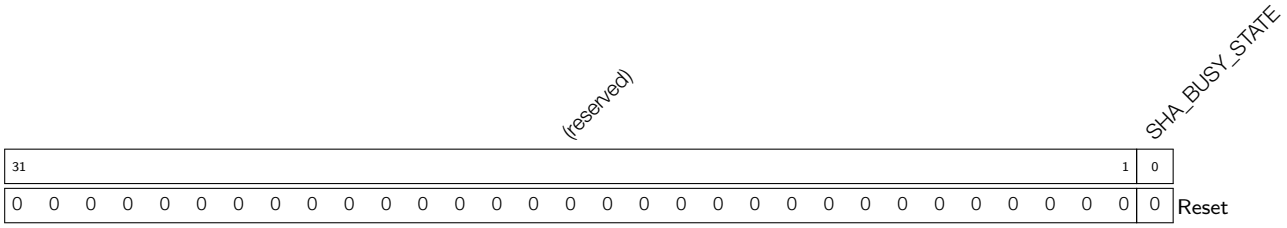
SHA_START 置 1 启动 SHA 加速器的 Typical SHA 模式。（只写）

Register 9.2. SHA_CONTINUE_REG (0x0014)



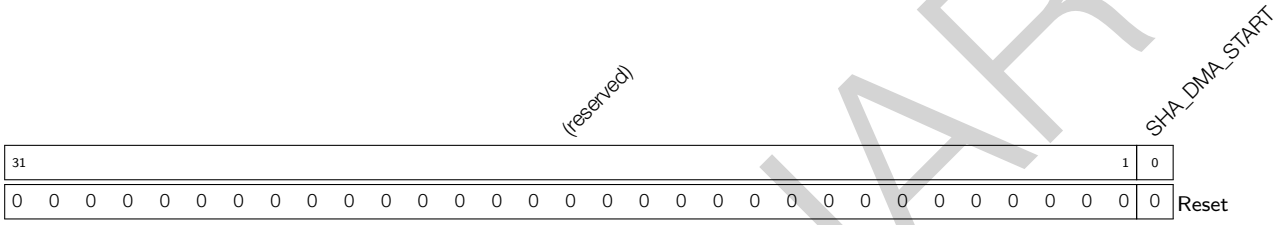
SHA_CONTINUE 置 1 继续 SHA 加速器的 Typical SHA 运算。（只写）

Register 9.3. SHA_BUSY_REG (0x0018)



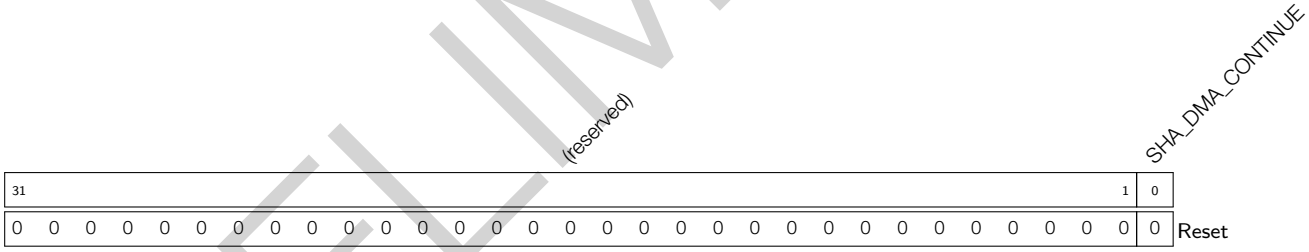
SHA_BUSY_STATE 指示 SHA 是否处于“忙碌”状态。(只读) 1'h0: 空闲 1'h1: 忙碌

Register 9.4. SHA_DMA_START_REG (0x001C)



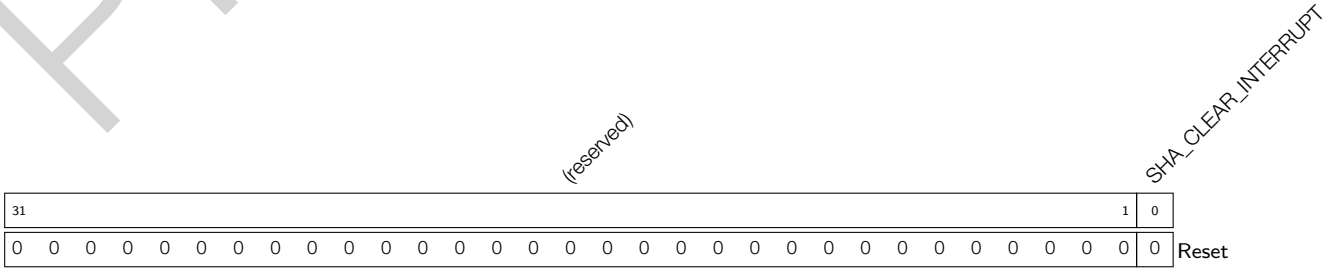
SHA_DMA_START 置 1 启动 SHA 加速器的 DMA-SHA 模式。(只写)

Register 9.5. SHA_DMA_CONTINUE_REG (0x0020)



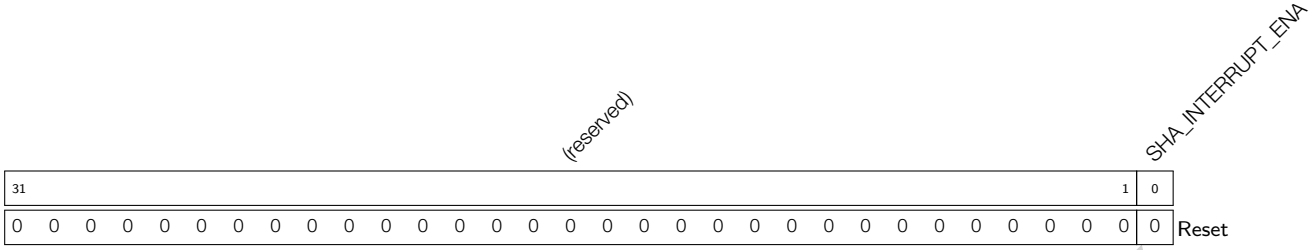
SHA_DMA_CONTINUE 置 1 继续 SHA 加速器的 DMA-SHA 运算。(只写)

Register 9.6. SHA_INT_CLEAR_REG (0x0024)



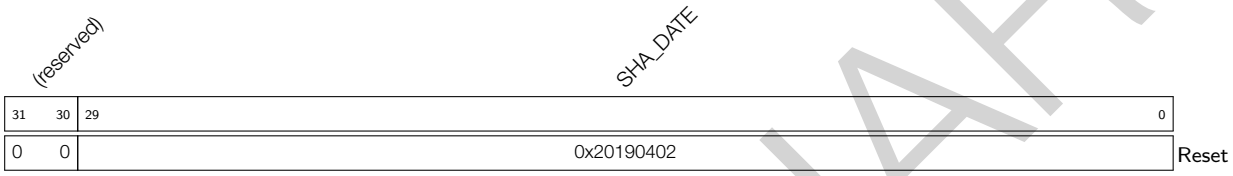
SHA_CLEAR_INTERRUPT 清除 DMA-SHA 中断。(只写)

Register 9.7. SHA_INT_ENA_REG (0x0028)



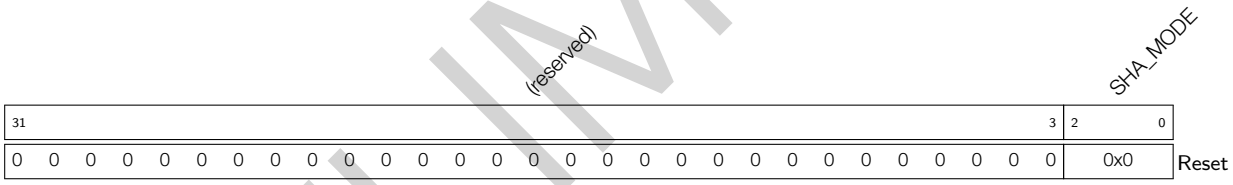
SHA_INTERRUPT_ENA 使能 DMA-SHA 中断。(读写)

Register 9.8. SHA_DATE_REG (0x002C)



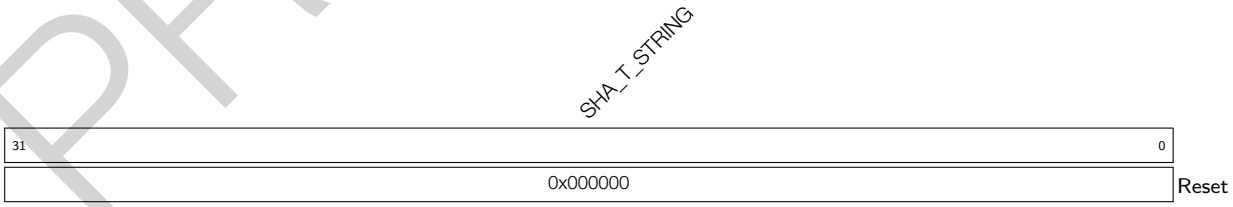
SHA_DATE 版本控制寄存器。(读写)

Register 9.9. SHA_MODE_REG (0x0000)



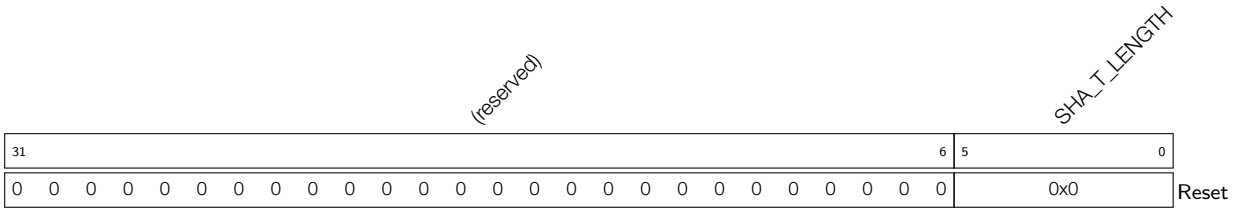
SHA_MODE 选择 SHA 加速器的运算标准，详见表 9-2。(R/W)

Register 9.10. SHA_T_STRING_REG (0x0004)



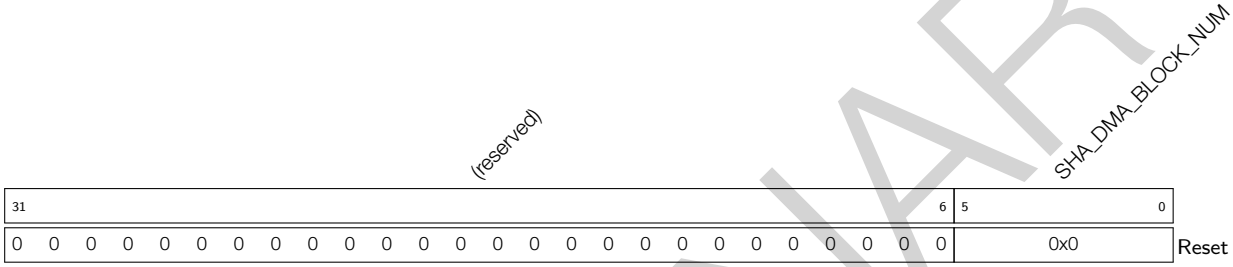
SHA_T_STRING 存储哈希字符串内容（仅用于计算 SHA-512/t 的哈希初始值）。(读写)

Register 9.11. SHA_T_LENGTH_REG (0x0008)



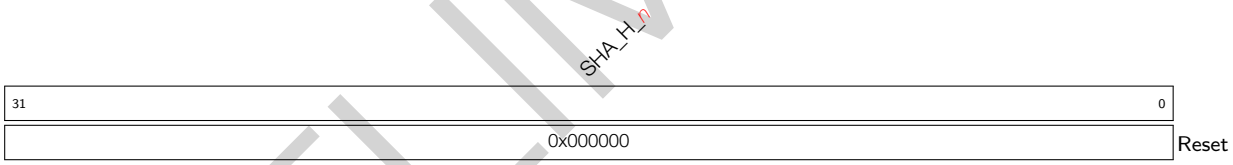
SHA_T_LENGTH 存储哈希字符串长度（仅用于计算 SHA-512/t 的哈希初始值）。（读写）

Register 9.12. SHA_DMA_BLOCK_NUM_REG (0x000C)



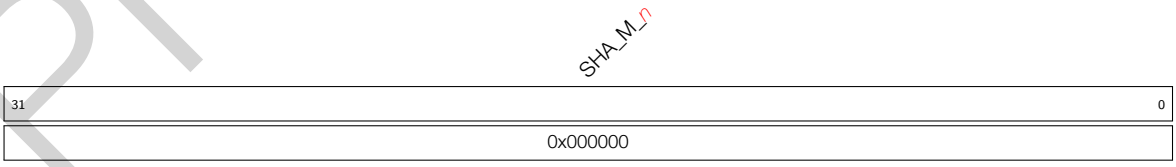
SHA_DMA_BLOCK_NUM 定义 DMA-SHA 工作模式下的信息块个数。（读写）

Register 9.13. SHA_H_*n*_REG (*n*: 0-15) (0x0040+4**n*)



SHA_H_*n* 存储第 *n* 个 32 位哈希值。（读写）

Register 9.14. SHA_M_*n*_REG (*n*: 0-31) (0x0080+4**n*)



SHA_M_*n* 存储第 *n* 个 32 位输入信息。（读写）

10 AES 加速器 (AES)

10.1 概述

ESP32-S3 内置 AES（高级加密标准）硬件加速器可使用 AES 算法，完成数据的加解密运算，具有 Typical AES 和 DMA-AES 两种工作模式。整体而言，相比基于纯软件的 AES 运算，AES 硬件加速器能够极大地提高运算速度。

10.2 主要特性

ESP32-S3 支持以下特性：

- Typical AES 工作模式
  - AES-128/AES-256 加解密运算
- DMA-AES 工作模式
  - AES-128/AES-256 加解密运算
  - 块（加密）模式
    - ECB (Electronic Codebook)
    - CBC (Cipher Block Chaining)
    - OFB (Output Feedback)
    - CTR (Counter)
    - CFB8 (8-bit Cipher Feedback)
    - CFB128 (128-bit Cipher Feedback)
  - 中断发生

10.3 工作模式简介

ESP32-S3 内置的 AES 加速器支持 Typical AES 和 DMA-AES 两种工作模式。

- Typical AES 工作模式：
  - 支持使用 128 位或 256 位密钥进行加密与解密运算，即 NIST FIPS 197 标准中的 AES-128 和 AES-256 加解密运算。

  这种情况下，明文/密文的读/写操作统一通过 CPU 访问完成。
- DMA-AES 工作模式：
  - 支持使用 128 位或 256 位密钥进行加密与解密运算，即 NIST FIPS 197 标准中的 AES-128 和 AES-256 加解密运算；
  - 还支持 NIST SP 800-38A 标准中的 ECB/CBC/OFB/CTR/CFB8/CFB128 等块加密模式运算。

  在这种情况下，明文/密文的传输通过硬件上的 DMA 完成，计算完成时会有中断发生。

用户可通过配置 AES_DMA_ENABLE_REG 选择 AES 加速器的工作模式，具体参考表 10-1。

表 10-1. 工作模式

AES_DMA_ENABLE_REG	工作模式
0	Typical AES
1	DMA-AES

用户可通过配置 AES_MODE_REG 寄存器选择密钥长度和加解密方向，具体可参考表 10-2。

表 10-2. 密钥长度和加解密方向

AES_MODE_REG[2:0]	密钥长度和加解密方向
0	AES-128 加密
1	保留
2	AES-256 加密
3	保留
4	AES-128 解密
5	保留
6	AES-256 解密
7	保留

有关 Typical AES 和 DMA-AES 两种工作模式的具体介绍，请见下方 10.4 章节和 10.5 章节。

注意：

ESP32-S3 的数字签名 (DS)模块也会调用 AES 加速器。此时，用户无法正常访问 AES 加速器。

10.4 Typical AES 工作模式

在 Typical AES 工作模式下，AES 加速器的状态值可查看寄存器 AES_STATE_REG，具体见表 10-3 所示：

表 10-3. 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

10.4.1 密钥、明文、密文

寄存器 AES_KEY_*n*_REG 用于存放密钥，由 8 个 32 位寄存器组成。

- 如果为 AES-128 加解密运算，则 128 位密钥在寄存器 AES_KEY_0_REG ~ AES_KEY_3_REG 中。
- 如果为 AES-256 加解密运算，则 256 位密钥在寄存器 AES_KEY_0_REG ~ AES_KEY_7_REG 中。

寄存器 AES_TEXT_IN_*m*_REG 和 AES_TEXT_OUT_*m*_REG 用于存放明文和密文，各由 4 个 32 位寄存器组成。

- 如果为 AES-128/265 加密运算，则运算开始之前用明文初始化寄存器 AES_TEXT_IN_*m*_REG。运算完成之后，AES 加速器将把密文更新入寄存器 AES_TEXT_OUT_*m*_REG。

- 如果为 AES-128/256 解密运算，则运算开始之前用密文初始化寄存器 AES_TEXT_IN_*m*_REG。运算完成之后，AES 加速器将把明文更新入寄存器 AES_TEXT_OUT_*m*_REG。

10.4.2 字节序

文本字节序

在 Typical AES 工作模式下，AES 加速器可以使用密钥对 128 位的 block 进行加解密。在操作寄存器 AES_TEXT_IN_*m*_REG 和 AES_TEXT_OUT_*m*_REG 中的数据时，用户应遵循表 10-4 中定义的文本字节序。

表 10-4. Typical AES 文本字节序

明文/密文					
State[1]		c[2]			
		0	1	2	3
r	0	AES_TEXT_*x*_0_REG[7:0]	AES_TEXT_*x*_1_REG[7:0]	AES_TEXT_*x*_2_REG[7:0]	AES_TEXT_*x*_3_REG[7:0]
	1	AES_TEXT_*x*_0_REG[15:8]	AES_TEXT_*x*_1_REG[15:8]	AES_TEXT_*x*_2_REG[15:8]	AES_TEXT_*x*_3_REG[15:8]
	2	AES_TEXT_*x*_0_REG[23:16]	AES_TEXT_*x*_1_REG[23:16]	AES_TEXT_*x*_2_REG[23:16]	AES_TEXT_*x*_3_REG[23:16]
	3	AES_TEXT_*x*_0_REG[31:24]	AES_TEXT_*x*_1_REG[31:24]	AES_TEXT_*x*_2_REG[31:24]	AES_TEXT_*x*_3_REG[31:24]

[1] 有关 “State（以及 c 和 r）” 的详细定义，请参考 NIST FIPS 197 中 “3.4 The State” 章节。

[2] 其中，*x* = IN 或 OUT。

密钥字节序

在 Typical AES 工作模式下，在向寄存器 AES_KEY_*n*_REG 中填入数据时，用户应遵循表 10-5 和表 10-6 中定义的文本字节序。

表 10-5. AES-128 密钥字节序

Bit[1]	w[0]	w[1]	w[2]	w[3][2]
[31:24]	AES_KEY_0_REG[7:0]	AES_KEY_1_REG[7:0]	AES_KEY_2_REG[7:0]	AES_KEY_3_REG[7:0]
[23:16]	AES_KEY_0_REG[15:8]	AES_KEY_1_REG[15:8]	AES_KEY_2_REG[15:8]	AES_KEY_3_REG[15:8]
[15:8]	AES_KEY_0_REG[23:16]	AES_KEY_1_REG[23:16]	AES_KEY_2_REG[23:16]	AES_KEY_3_REG[23:16]
[7:0]	AES_KEY_0_REG[31:24]	AES_KEY_1_REG[31:24]	AES_KEY_2_REG[31:24]	AES_KEY_3_REG[31:24]

[1] Bit 列代表 w[0] ~ w[3] 每个 word 中的各个字节。

[2] w[0] ~ w[3] 符合标准 NIST FIPS 197 中 “5.2 Key Expansion” 章节中对 “the first Nk words of the expanded key” 的描述。

表 10-6. AES-256 密钥字节序

Bit[1]	w[0]	w[1]	w[2]	w[3]	w[4]	w[5]	w[6]	w[7][2]
[31:24]	AES_KEY_0_REG[7:0]	AES_KEY_1_REG[7:0]	AES_KEY_2_REG[7:0]	AES_KEY_3_REG[7:0]	AES_KEY_4_REG[7:0]	AES_KEY_5_REG[7:0]	AES_KEY_6_REG[7:0]	AES_KEY_7_REG[7:0]
[23:16]	AES_KEY_0_REG[15:8]	AES_KEY_1_REG[15:8]	AES_KEY_2_REG[15:8]	AES_KEY_3_REG[15:8]	AES_KEY_4_REG[15:8]	AES_KEY_5_REG[15:8]	AES_KEY_6_REG[15:8]	AES_KEY_7_REG[15:8]
[15:8]	AES_KEY_0_REG[23:16]	AES_KEY_1_REG[23:16]	AES_KEY_2_REG[23:16]	AES_KEY_3_REG[23:16]	AES_KEY_4_REG[23:16]	AES_KEY_5_REG[23:16]	AES_KEY_6_REG[23:16]	AES_KEY_7_REG[23:16]
[7:0]	AES_KEY_0_REG[31:24]	AES_KEY_1_REG[31:24]	AES_KEY_2_REG[31:24]	AES_KEY_3_REG[31:24]	AES_KEY_4_REG[31:24]	AES_KEY_5_REG[31:24]	AES_KEY_6_REG[31:24]	AES_KEY_7_REG[31:24]

[1] Bit 列代表 w[0] ~ w[7] 每个 word 中的各个字节。

[2] w[0] ~ w[7] 符合标准 NIST FIPS 197 中 “5.2 Key Expansion” 章节中对 “the first Nk words of the expanded key” 的描述。

10.4.3 Typical AES 工作模式的流程

单次运算

1. 对寄存器 AES_DMA_ENABLE_REG 写入 0。
2. 初始化寄存器 AES_MODE_REG、AES_KEY_*n*_REG、AES_TEXT_IN_*m*_REG。
3. 启动运算。对寄存器 AES_TRIGGER_REG 写入 1。
4. 等待运算完成。轮询寄存器 AES_STATE_REG，直到读到 0。
5. 从寄存器 AES_TEXT_OUT_*m*_REG 读取结果。

连续运算

在连续运算过程中，每次运算完成之后，只有寄存器 AES_TEXT_IN_*m*_REG 和 AES_TEXT_OUT_*m*_REG (*m*: 0-3) 会被 AES 加速器更新，而 AES_DMA_ENABLE_REG、AES_MODE_REG、AES_KEY_*n*_REG 等寄存器中的内容不会变化。所以进行连续运算时可以简化初始化操作。

1. 第一次运算之前对寄存器 AES_DMA_ENABLE_REG 写入 0。
2. 第一次运算之前初始化寄存器 AES_MODE_REG 和 AES_KEY_*n*_REG。
3. 更新寄存器 AES_TEXT_IN_*m*_REG。
4. 启动运算。对寄存器 AES_TRIGGER_REG 写入 1。
5. 等待运算完成。轮询寄存器 AES_STATE_REG，直到读到 0。
6. 从寄存器 AES_TEXT_OUT_*m*_REG 读取结果 。返回步骤 3，进行下一轮运算。

10.5 DMA-AES 工作模式

在 DMA-AES 工作模式下，AES 加速器可支持 ECB/CBC/OFB/CTR/CFB8/CFB128 等 6 种块模式运算。用户可通过配置 AES_BLOCK_MODE_REG 寄存器选择具体运算类型，具体可参考表 10-7。

表 10-7. 块模式选择

AES_BLOCK_MODE_REG[2:0]	块模式
0	ECB (Electronic Code Book)
1	CBC (Cipher Block Chaining)
2	OFB (Output FeedBack)
3	CTR (Counter)
4	CFB8 (8-bit Cipher FeedBack)
5	CFB128 (128-bit Cipher FeedBack)
6	保留
7	保留

AES 加速器的状态值可查看寄存器 AES_STATE_REG，具体见表 10-8 所示：

表 10-8. 状态返回值

返回值	描述	状态说明
0	IDLE	加速器空闲
1	WORK	加速器忙于计算
2	DONE	加速器计算完成

AES 加速器在 DMA-AES 工作模式下允许中断发生，软件清零。中断功能默认关闭，用户可通过将 AES_INT_ENA_REG 寄存器配置为 1 开启中断。如开启中断功能，AES 加速器在完成计算时，中断发生。

10.5.1 密钥、明文、密文

块运算模式

在块运算模式下，AES 加速器的源数据来自 DMA，结果数据也将被写入 DMA。

- 如果为加密运算，则 DMA 从 memory 中读取明文数据流并将其传给 AES。AES 计算出密文后将密文写入 DMA。DMA 再将密文写入 memory。
- 如果为解密运算，则 DMA 从 memory 中读取密文数据流并将其传给 AES。AES 计算出明文后将明文写入 DMA。DMA 再将明文写入 memory。

AES 加速器在进行块运算时，结果数据与源数据的大小保持一致。此时，DMA 的数据搬运过程和 AES 的计算过程有所交叠，因此总工作时间有所减少。

值得注意的是，AES 加速器在 DMA-AES 工作模式下要求源数据的大小必须是 128 位的整数倍，否则需要将原始明文封装为 128 位的整数倍，即在原比特串 (bit string) 尾部尽可能少的补 “0”，具体过程见表 10-9 所示。

表 10-9. TEXT-PADDING

Function : **TEXT-PADDING**( )
Input : X, bit string. **Output** : Y = **TEXT-PADDING**(X), whose length is the nearest integral multiples of 128 bits.
Steps Let us assume that X is a data-stream that can be split into n parts as following: $X = X_1 \| \| X_2 \| \| \cdots \| \| X_{n-1} \| \| X_n$ Here, the lengths of $X_1, X_2, \cdots, X_{n-1}$ all equal to 128 bits, and the length of X_n is t (0<=t<=127). If $t = 0$, then **TEXT-PADDING**(X) = X; If $0 < t <= 127$, define a 128-bit block, X_n^*, and let $X_n^* = X_n \| \| 0^{128-t}$, then **TEXT-PADDING**($X$) = $X_1 \| \| X_2 \| \| \cdots \| \| X_{n-1} \| \| X_n^*$ = $X \| \| 0^{128-t}$

10.5.2 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：0102030405060708090A0B0C0D0E0F10111213141516171819 1A1B1C1D1E1F20

假设起始地址为 0x0280，则源数据在 memory 中的存放位置如表 10-10 所示。结果数据也遵从相同的存放规则，在此不多做介绍。

表 10-10. DMA AES 存储字节序

地址	字节	地址	字节	地址	字节	地址	字节
0x0280	0x01	0x0281	0x02	0x0282	0x03	0x0283	0x04
0x0284	0x05	0x0285	0x06	0x0286	0x07	0x0287	0x08
0x0288	0x09	0x0289	0x0A	0x028A	0x0B	0x028B	0x0C
0x028C	0x0D	0x028D	0x0E	0x028E	0x0F	0x028F	0x10
0x0290	0x11	0x0291	0x12	0x0292	0x13	0x0293	0x14
0x0294	0x15	0x0295	0x16	0x0296	0x17	0x0297	0x18
0x0298	0x19	0x0299	0x1A	0x029A	0x1B	0x029B	0x1C
0x029C	0x1D	0x029D	0x1E	0x029E	0x1F	0x029F	0x20

另外，值得注意的是，DMA 既可以访问片内存储空间，又可以访问片外 PSRAM。当访问片外 PSRAM 时，基地址必须满足 DMA 对地址的相关要求，但当访问片内存储器空间时则没有限制。详情请见章节 9 *通用 DMA 控制器 (DMA) [to be added later]*。

10.5.3 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。

10.5.4 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。

10.5.5 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 10-10 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。

10.5.6 DMA-AES 工作模式的流程

1. 选择一条 DMA 通道与 AES 加速器连接，配置 DMA 链表，而后启动 DMA。详情请见章节 9 *通用 DMA 控制器 (DMA) [to be added later]*。
2. 配置 AES：
   - 对寄存器 AES_DMA_ENABLE_REG 写入 1。
   - 选择是否开启中断。根据需要设置寄存器 AES_INT_ENA_REG 的值。
   - 初始化 AES_MODE_REG 和 AES_KEY_*n*_REG 寄存器。
   - 配置 AES_BLOCK_MODE_REG 寄存器，选择具体块加密模式。详见表 10-7。
   - 初始化寄存器 AES_BLOCK_NUM_REG，请参照章节 10.5.4。
   - 初始化寄存器 AES_INC_SEL_REG（仅在 CTR 块模式下使用）。
   - 初始化存储器 AES_IV_MEM（在 ECB 块模式下不使用）。
3. 启动运算。对寄存器 AES_TRIGGER_REG 写入 1。
4. 等待运算完成。轮询寄存器 AES_STATE_REG，直到读到 2。如果开启了中断功能，也可以等待 AES_INT 中断产生。
5. 确认 DMA 完成从 AES 到内存的数据传输。此时，结果数据已经被 DMA 写入 memory，可以直接从中读取。详情请参考章节 9 *通用 DMA 控制器 (DMA) [to be added later]*。
6. 如果开启了中断，当处理中断程序完成后，请及时对寄存器 AES_INT_CLR_REG 写 1 以清除中断。
7. 对寄存器 AES_DMA_EXIT_REG 写入 1 释放 AES 加速器。之后如果再读取寄存器 AES_STATE_REG 将读到 0。该步操作可以提前完成，但必须在步骤 4 之后。

10.6 存储器列表

本小节的所有地址均为相对于 AES 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

名称	描述	大小（比特）	起始地址	结束地址	访问权限
AES_IV_MEM	存储器 IV	16 字节	0x0050	0x005F	读 / 写

10.7 寄存器列表

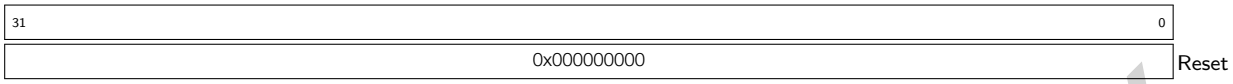
本小节的所有地址均为相对于 AES 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

名称	描述	地址	访问
密钥寄存器			
AES_KEY_0_REG	AES 密钥寄存器 0	0x0000	读 / 写
AES_KEY_1_REG	AES 密钥寄存器 1	0x0004	读 / 写
AES_KEY_2_REG	AES 密钥寄存器 2	0x0008	读 / 写
AES_KEY_3_REG	AES 密钥寄存器 3	0x000C	读 / 写
AES_KEY_4_REG	AES 密钥寄存器 4	0x0010	读 / 写
AES_KEY_5_REG	AES 密钥寄存器 5	0x0014	读 / 写
AES_KEY_6_REG	AES 密钥寄存器 6	0x0018	读 / 写
AES_KEY_7_REG	AES 密钥寄存器 7	0x001C	读 / 写
TEXT_IN 寄存器			
AES_TEXT_IN_0_REG	源数据寄存器 0	0x0020	读 / 写
AES_TEXT_IN_1_REG	源数据寄存器 1	0x0024	读 / 写
AES_TEXT_IN_2_REG	源数据寄存器 2	0x0028	读 / 写
AES_TEXT_IN_3_REG	源数据寄存器 3	0x002C	读 / 写
TEXT_OUT 寄存器			
AES_TEXT_OUT_0_REG	结果数据寄存器 0	0x0030	只读
AES_TEXT_OUT_1_REG	结果数据寄存器 1	0x0034	只读
AES_TEXT_OUT_2_REG	结果数据寄存器 2	0x0038	只读
AES_TEXT_OUT_3_REG	结果数据寄存器 3	0x003C	只读
配置寄存器			
AES_MODE_REG	选择密钥长度和加解密方向	0x0040	读 / 写
AES_DMA_ENABLE_REG	选择 AES 加速器工作模式	0x0090	读 / 写
AES_BLOCK_MODE_REG	选择 DMA-AES 下的块运算模式	0x0094	读 / 写
AES_BLOCK_NUM_REG	块数量配置寄存器	0x0098	读 / 写
AES_INC_SEL_REG	标准增量函数选择寄存器	0x009C	读 / 写
控制 / 状态寄存器			
AES_TRIGGER_REG	开始运算寄存器	0x0048	只写
AES_STATE_REG	运算状态寄存器	0x004C	只读
AES_DMA_EXIT_REG	退出运算寄存器	0x00B8	只写
中断寄存器			
AES_INT_CLR_REG	DMA-AES 中断清除	0x00AC	只写
AES_INT_ENA_REG	DMA-AES 中断使能寄存器	0x00B0	读 / 写

10.8 寄存器

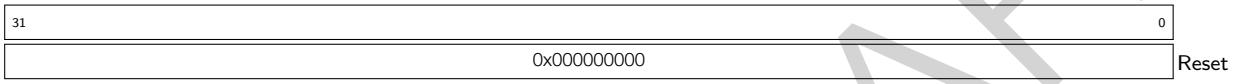
本小节的所有地址均为相对于 AES 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

Register 10.1. AES_KEY_*n*_REG (*n*: 0-7) (0x0000+4n*)**



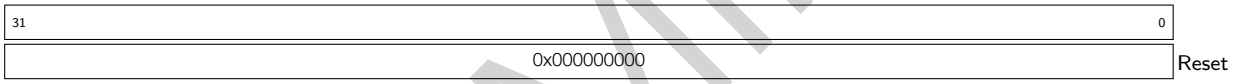
AES_KEY_*n*_REG (*n*: 0-7) AES 密钥寄存器。（读 / 写）

Register 10.2. AES_TEXT_IN_*m*_REG (*m*: 0-3) (0x0020+4m*)**



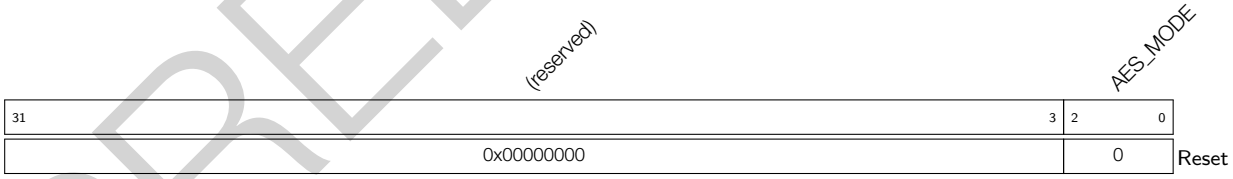
AES_TEXT_IN_*m*_REG (*m*: 0-3) Typical AES 文本输入寄存器。（读 / 写）

Register 10.3. AES_TEXT_OUT_*m*_REG (*m*: 0-3) (0x0030+4m*)**



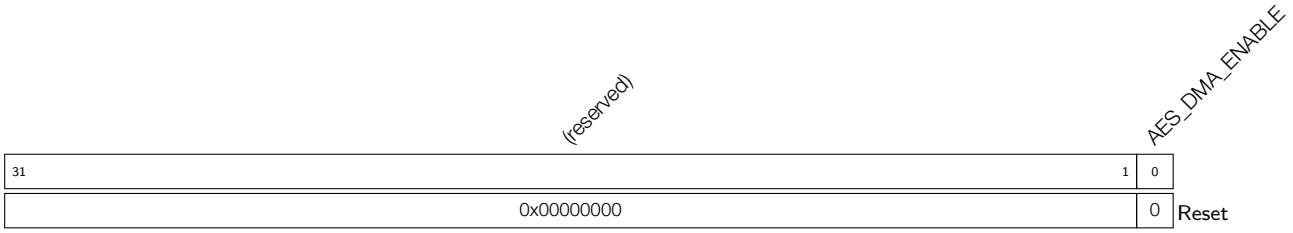
AES_TEXT_OUT_*m*_REG (*m*: 0-3) Typical AES 文本输出寄存器。（只读）

Register 10.4. AES_MODE_REG (0x0040)



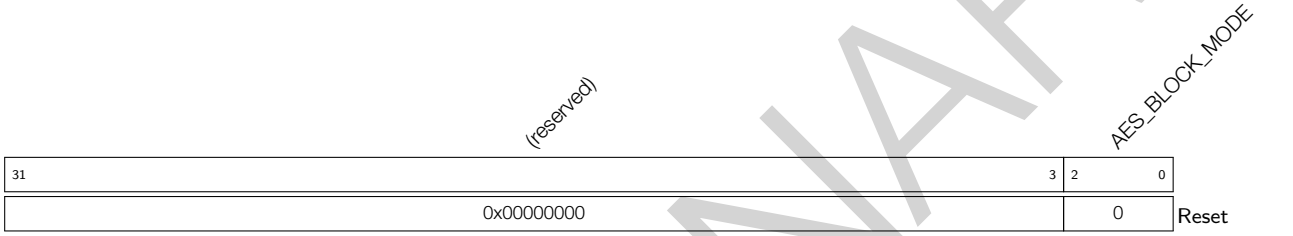
AES_MODE 选择 AES 加速器的密钥长度和加解密方向，详情请见表 10-2。（读 / 写）

Register 10.5. AES_DMA_ENABLE_REG (0x0090)



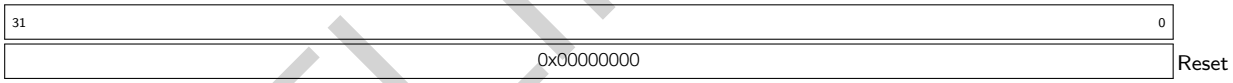
AES_DMA_ENABLE 选择 AES 加速器的工作模式。0：Typical AES，1：DMA-AES。详情请见表 10-1。（读 / 写）

Register 10.6. AES_BLOCK_MODE_REG (0x0094)



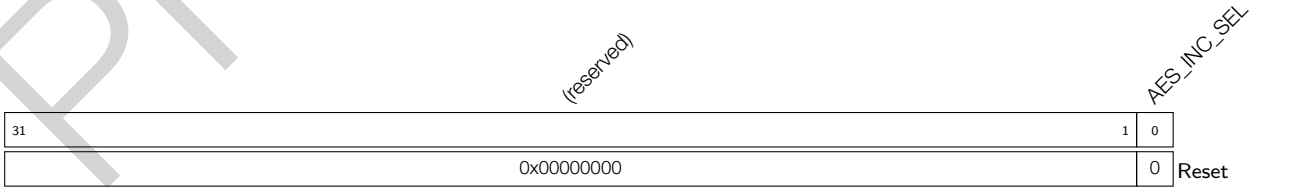
AES_BLOCK_MODE 选择 AES 加速器在 DMA-AES 工作模式下的块模式，详情请见表 10-7。（读 / 写）

Register 10.7. AES_BLOCK_NUM_REG (0x0098)



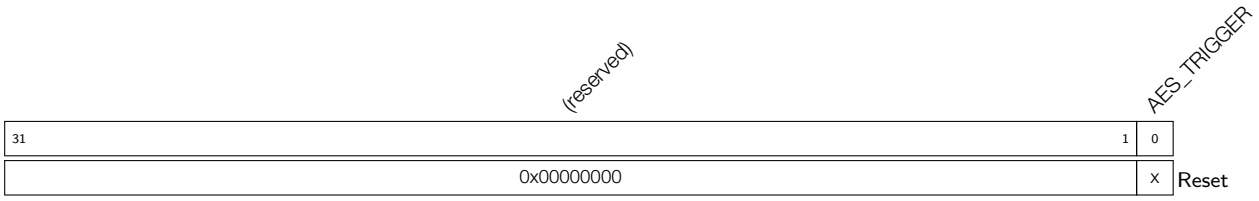
AES_BLOCK_NUM 在 DMA-AES 运算中待加解密的文本块数。详情请见章节 10.5.4。（读 / 写）

Register 10.8. AES_INC_SEL_REG (0x009C)



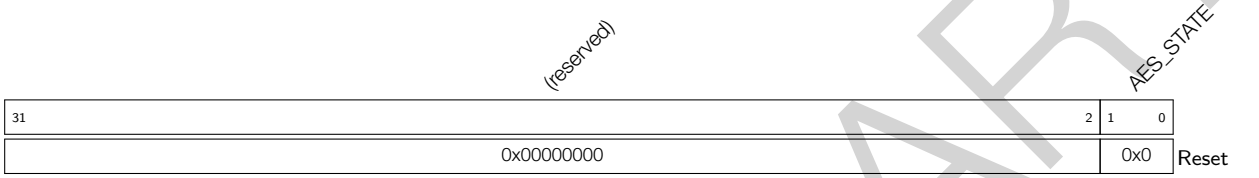
AES_INC_SEL 选择 CTR 块模式使用的标准增量函数。置 0 选择 INC_{32} 标准增量函数，置 1 选择 INC_{128} 标准增量函数。（读 / 写）

Register 10.9. AES_TRIGGER_REG (0x0048)



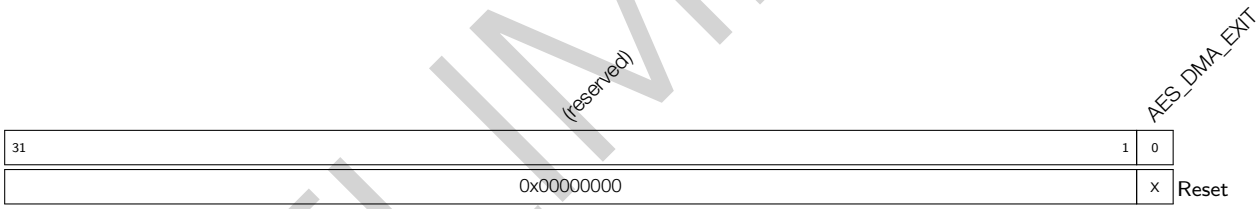
AES_TRIGGER 写入 1 使能 AES 运算。(只写)

Register 10.10. AES_STATE_REG (0x004C)



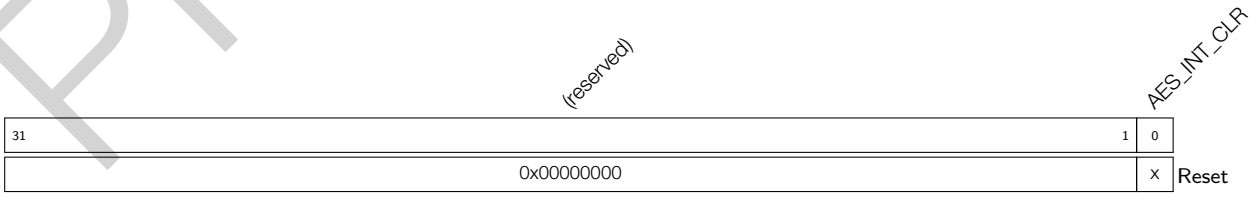
AES_STATE AES 状态寄存器。详见表 10-3 (Typical AES 工作模式) 和表 10-8 (DMA-AES 工作模式)。(只读)

Register 10.11. AES_DMA_EXIT_REG (0x00B8)



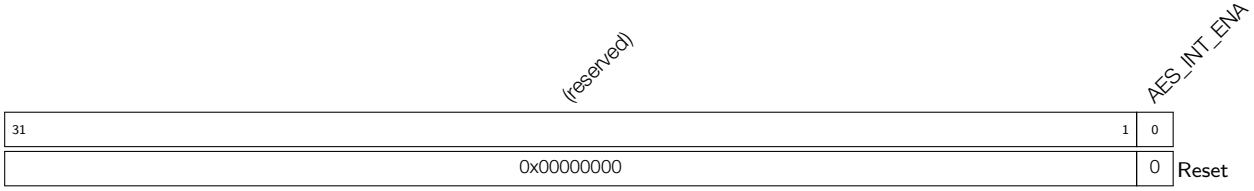
AES_DMA_EXIT 在 DMA-AES 运算完成后，在下一次配置 AES 任何寄存器之前，写入 1 使 AES 回到空闲状态。(只写)

Register 10.12. AES_INT_CLR_REG (0x00AC)



AES_INT_CLR 写入 1 清除 AES 中断。(只写)

Register 10.13. AES_INT_ENA_REG (0x00B0)



AES_INT_ENA 写入 1 使能 AES 中断功能，写入 0 关闭 AES 中断功能。(读 / 写)

11 RSA 加速器 (RSA)

11.1 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。

11.2 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

11.3 功能描述

RSA 加速器的激活仅需使能 SYSTEM_PERIP_CLK_EN1_REG 外围时钟的 SYSTEM_CRYPTO_RSA_CLK_EN 位，并同时清零 SYSTEM_RSA_PD_CTRL_REG 寄存器中的SYSTEM_RSA_MEM_PD 位。

不过，RSA 加速器激活后还须等待RSA 相关存储器初始化完成后才能开始工作。具体来说，寄存器 RSA_CLEAN_REG 读 0 时初始化开始，读 1 时初始化完成。因此，在复位后首次使用 RSA 加速器时，软件需要先查询寄存器 RSA_CLEAN_REG 的值是否为 1，以确保 RSA 加速器可正常工作。

此外，RSA 加速器支持中断功能，可对寄存器 RSA_INTERRUPT_ENA_REG 写 1 / 0 以开启 / 关闭中断。RSA 加速器的中断功能默认开启。

> **注意：**
> ESP32-S3 的数字签名 (DS)模块也会调用 RSA 加速器。此时，用户无法正常访问 RSA 加速器。

11.3.1 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 $\bar{r}$ 和 M'。这两个参数需要通过软件提前运算得到。

RSA 加速器支持运算子长度为 $N = 32 \times x$ ($x \in \{1, 2, 3, \ldots, 128\}$) 的大数模幂运算。$Z$、$X$、$Y$、$M$ 和 $\bar{r}$ 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 11.3.4 获取详细信息。
3. 将 X_i、Y_i、M_i、$\overline{r}_i (i \in \{0, 1, \ldots, n-1\})$ 分别写入存储器 RSA_X_MEM、RSA_Y_MEM、RSA_M_MEM、RSA_Z_MEM。每块存储器的容量都是 128 字 (word)。每块存储器的每一个字刚好存放一个 b 进制数。这些存储器都是低地址存放运算子的低位进制数，高地址存放运算子的高位进制数。

   只需要根据运算子长度，将各个运算子中有效的数据写入存储器，没有使用到的存储器可以是任意值。
4. 对寄存器 RSA_MODEXP_START_REG 写入 1 启动计算。
5. 等待运算结束。轮询寄存器 RSA_IDLE_REG 直到读到 1，或者等待 RSA 中断产生。
6. 从存储器 RSA_Z_MEM 读出运算结果 $Z_i (i \in \{0, 1, \ldots, n-1\})$。
7. 若中断功能已开启，对寄存器 RSA_CLEAR_INTERRUPT_REG 写入 1 以清除中断。

运算结束后，寄存器 RSA_MODE_REG 中存储的运算子长度信息以及存储器 RSA_Y_MEM 中的 Y_i、存储器 RSA_M_MEM 中的 M_i、寄存器 RSA_M_PRIME_REG 中的 M' 都不会变化。但是，存储器 RSA_X_MEM 中的 X_i 与存储器 RSA_Z_MEM 中的 $\overline{r}_i$ 都已经被覆盖。所以当需要连续运算时，只需要更新必需的寄存器与存储器即可。

11.3.2 大数模乘运算

大数模乘运算 $Z = X \times Y \bmod M$ 也是基于 Montgomery Multiplication 实现的。因此，与大数模幂运算类似，也需要预先通过软件计算额外的两个运算子 $\bar{r}$ 和 M'。

RSA 加速器也支持 128 种运算子长度的大数模乘运算。

大数模乘运算的软件流程为：

1. 对寄存器 RSA_INTERRUPT_ENA_REG 写 1 / 0 以开启 / 关闭中断。
2. 配置相关寄存器。
   (a) 对寄存器 RSA_MODE_REG 写入 $(\frac{N}{32} - 1)$。
   (b) 对寄存器 RSA_M_PRIME_REG 写入 M'。
3. 将 X_i、Y_i、M_i、$\bar{r}_i$ $(i \in \{0, 1, \ldots, n-1\})$ 分别写入存储器 RSA_X_MEM、RSA_Y_MEM、RSA_M_MEM、RSA_Z_MEM。每块存储器的容量都是 128 字 (word)。

   每块存储器的每一个字刚好存放一个 b 进制数。这些存储器都是低地址存放运算子的低位进制数，高地址存放运算子的高位进制数。

   只需要根据运算子长度，将各个运算子中有效的数据写入存储器，没有使用到的存储器可以是任意值。
4. 对寄存器 RSA_MODMULT_START_REG 写入 1。
5. 等待运算结束。轮询寄存器 RSA_IDLE_REG 直到读到 1，或者等待 RSA 中断产生。
6. 从存储器 RSA_Z_MEM 读出运算结果 Z_i $(i \in \{0, 1, \ldots, n-1\})$。
7. 若中断功能已开启，对寄存器 RSA_CLEAR_INTERRUPT_REG 写入 1 以清除中断。

运算结束后，寄存器 RSA_MODE_REG 中存储的运算子长度信息以及存储器 RSA_X_MEM 中的 X_i、存储器 RSA_Y_MEM 中的 Y_i、存储器 RSA_M_MEM 中的 M_i、寄存器 RSA_M_PRIME_REG 中的 M' 都不会变化。但是，存储器 RSA_Z_MEM 中的 $\bar{r}_i$ 已经被覆盖。所以当需要连续运算时，只需要更新必需的寄存器与存储器即可。

11.3.3 大数乘法运算

大数乘法运算实现了 $Z = X \times Y$。其中 Z 的长度是运算子 X、Y 长度的两倍。所以 RSA 加速器只支持运算子 X、Y 长度为 $N = 32 \times x$ $(x \in \{1, 2, 3, \ldots, 64\})$ 的大数乘法运算。运算子 Z 的长度 $\hat{N}$ 为 $2 \times N$。

大数乘法运算的软件流程为：

1. 对寄存器 RSA_INTERRUPT_ENA_REG 写 1 / 0 以开启 / 关闭中断。
2. 配置相关寄存器。对寄存器 RSA_MODE_REG 写入 $(\frac{\hat{N}}{32} - 1)$，即 $(\frac{N}{16} - 1)$。
3. 将 X_i、Y_i $(i \in \{0, 1, \ldots, n-1\})$ 分别写入存储器 RSA_X_MEM、RSA_Z_MEM。每块存储器的容量都是 64 字。每块存储器的每一个字刚好存放一个 b 进制数。这些存储器都是低地址存放运算子的低位进制数，高地址存放运算子的高位进制数。n 为 $\frac{N}{32}$。

   X_i $(i \in \{0, 1, \ldots, n-1\})$ 要填充到存储器 RSA_X_MEM 中的第 i 个字对应的地址中，但需要注意的是，Y_i $(i \in \{0, 1, \ldots, n-1\})$ 并不是要填充到存储器 RSA_Z_MEM 中的第 i 个字对应的地址中，而是需要填充到存储器 RSA_Z_MEM 中的第 $n+i$ 个字对应的地址中，即存储器 RSA_Z_MEM 的基地址加上偏移量 $4 \times (n+i)$。

   只需要根据运算子长度，将这两个运算子中有效的数据写入存储器，没有使用到的存储器可以是任意值。
4. 对寄存器 RSA_MULT_START_REG 写入 1。

5. 等待运算结束。轮询寄存器 RSA_IDLE_REG 直到读到 1，或者等待 RSA 中断产生。
6. 从存储器 RSA_Z_MEM 读出运算结果 Z_i ($i \in \{0, 1, \ldots, \hat{n}-1\}$)。$\hat{n}$ 为 $2 \times n$ 。
7. 若中断功能已开启，对寄存器 RSA_CLEAR_INTERRUPT_REG 写入 1 以清除中断。

运算结束后，寄存器 RSA_MODE_REG 中存储的运算子长度信息以及存储器 RSA_X_MEM 中的 X_i 都不会变化。但是，存储器 RSA_Z_MEM 中的 Y_i 已经被覆盖。所以当需要连续运算时，只需要更新必需的寄存器与存储器即可。

11.3.4 控制加速

对于大数模幂运算，ESP32-S3 的 RSA 加速器还特别提供 SEARCH 和 CONSTANT_TIME 两个选项，可提高运算速度。默认情况下，这两个选项均处于不加速状态，可以单独使用，也可以同时使用。

具体来说，当这两个选项均处于不加速状态时，求解 $Z = X^Y \bmod M$ 的时间开销完全由运算子长度决定。否则，只要有某个选项携带有加速效果，那么运算的时间开销还与 Y 的 0/1 分布有关。

为了更清楚地说明问题，首先假设 Y 的二进制表示为：

$$Y = (\widetilde{Y}_{N-1}\widetilde{Y}_{N-2}\cdots\widetilde{Y}_{t+1}\widetilde{Y}_t\widetilde{Y}_{t-1}\cdots\widetilde{Y}_0)_2$$

其中，

- N 代表 Y 的长度，
- $\widetilde{Y}_t$ 的值为 1，
- $\widetilde{Y}_{N-1}, \widetilde{Y}_{N-2}, \ldots, \widetilde{Y}_{t+1}$ 的值均为 0，
- 且 $\widetilde{Y}_{t-1}, \widetilde{Y}_{t-2}, \ldots, \widetilde{Y}_0$ 中包括 m 个 0，其余 t-m 全部为 1，即 $\widetilde{Y}_{t-1}\widetilde{Y}_{t-2}, \cdots, \widetilde{Y}_0$ 的汉明重量 (Hamming weight) 为 $t-m$。

此时，当启动任一选项时：

- SEARCH 选项（RSA_SEARCH_ENABLE 置 1 开始加速）
  - RSA 加速器将忽略所有 $\widetilde{Y}_i$ ($i > \alpha$) 位。其中，加速位置 α 可通过 RSA_SEARCH_POS_REG 寄存器配置。α 的最大值不能超过 N-1，否则相当于没有加速；且不建议小于 t，否则无法正确求解 $Z = X^Y \bmod M$。当设置 α 为 t 时，加速效果最佳。此时，$\widetilde{Y}_{N-1}, \widetilde{Y}_{N-2}, \ldots, \widetilde{Y}_{t+1}$ 中的 0 位将在运算中全部被忽略。
- CONSTANT_TIME 选项（RSA_CONSTANT_TIME_REG 置 0 开始加速）
  - RSA 加速器在运算过程中将简化对 Y 中 0 位的处理。因此不难想象，Y 中的 0 越多，加速效果越明显。

为了直观地展示这两个选项带来的加速效果，下面通过一个典型实例加以说明。在 $Z = X^Y \bmod M$ 中，N 等于 3072，Y 等于 65537。表 11-1 展示了 4 种选项组合对应的时间开销。注意，这里 SEARCH 选项开启时设定 α 为 16。

表 11-1. 加速效果

SEARCH 选项	CONSTANT_TIME 选项	时间开销
不加速	不加速	376.405 ms
加速	不加速	2.260 ms
不加速	加速	1.203 ms
加速	加速	1.165 ms

可以看到：

- 当两个选项均处于不加速状态时，时间开销最大。
- 当两个选项均处于加速状态时，时间开销最小。
- 相比于不加速状态，任一选项处于加速状态时的时间开销明显大幅度降低。

11.4 存储器列表

请注意，这里的地址都是相对于 RSA 加速器基地址的地址偏移量（相对地址），详见章节 1 系统和存储器 中的表 1-4。

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

11.5 寄存器列表

请注意，这里的地址都是相对于 RSA 加速器基地址的地址偏移量（相对地址），详见章节 1 系统和存储器 中的表 1-4。

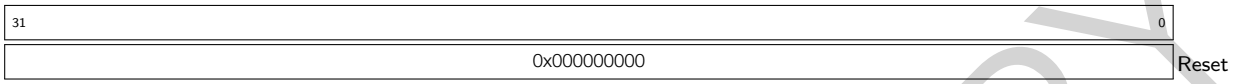
名称	描述	地址	访问
配置寄存器			
RSA_M_PRIME_REG	M' 存储器	0x0800	读 / 写
RSA_MODE_REG	RSA 长度模式	0x0804	读 / 写
RSA_CONSTANT_TIME_REG	固定时间选项	0x0820	读 / 写
RSA_SEARCH_ENABLE_REG	使能 search 加速选项	0x0824	读 / 写
RSA_SEARCH_POS_REG	search 起始位置	0x0828	读 / 写
状态/控制寄存器			
RSA_CLEAN_REG	RSA 清除寄存器	0x0808	只读
RSA_MODEXP_START_REG	模幂运算起始位	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

11.6 寄存器

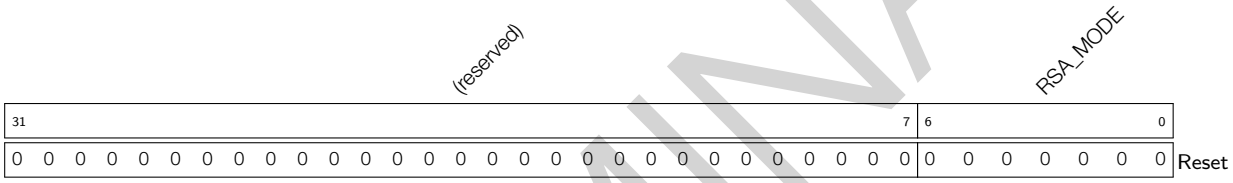
请注意，这里的地址都是相对于 RSA 加速器基地址的地址偏移量（相对地址），详见章节 1 系统和存储器 中的表 1-4。

Register 11.1. RSA_M_PRIME_REG (0x0800)



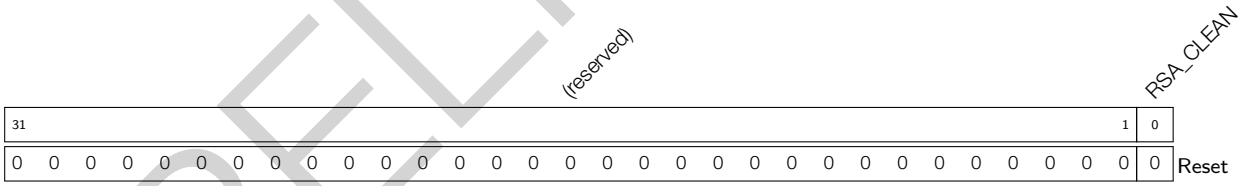
RSA_M_PRIME_REG 此寄存器存储 M'。(读 / 写)

Register 11.2. RSA_MODE_REG (0x0804)



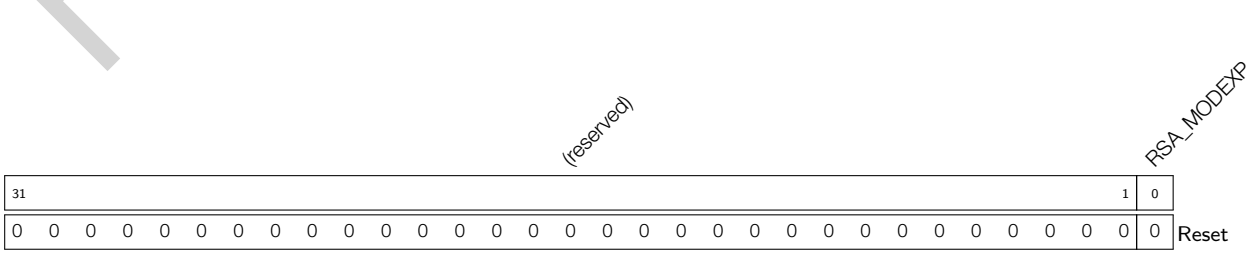
RSA_MODE 此寄存器存储模幂运算的模式。(读 / 写)

Register 11.3. RSA_CLEAN_REG (0x0808)



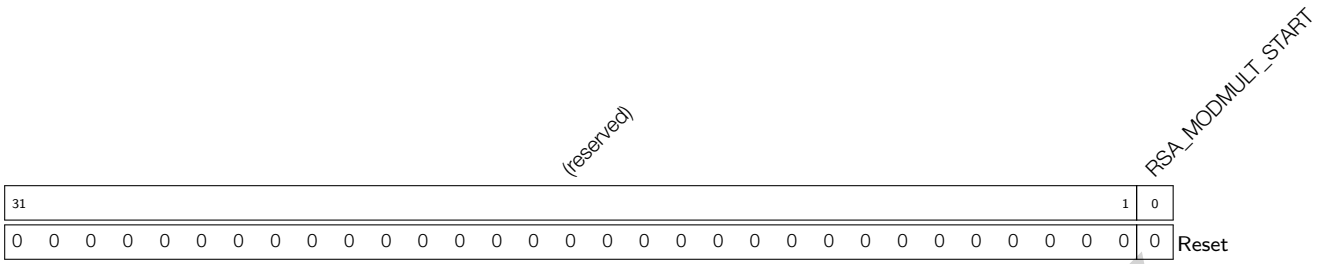
RSA_CLEAN 一旦存储器初始化结束，此位为 1。(只读)

Register 11.4. RSA_MODEXP_START_REG (0x080C)



RSA_MODEXP_START 写入 1 以开始模幂运算。(只写)

Register 11.5. RSA_MODMULT_START_REG (0x0810)



RSA_MODMULT_START 写入 1 以开始模乘运算。(只写)

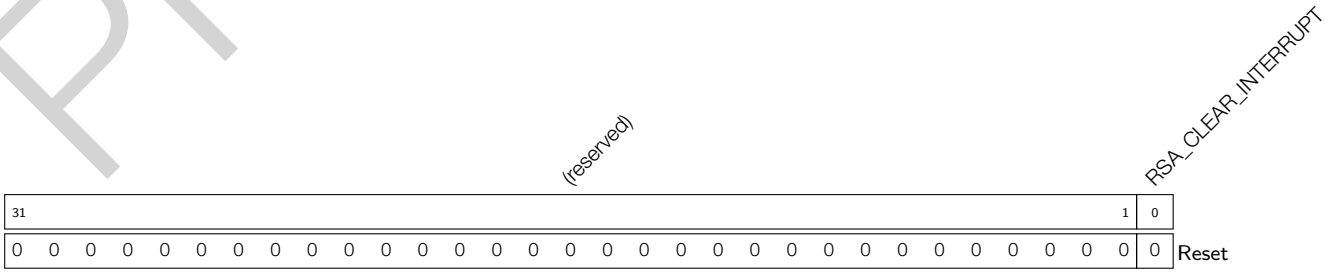
Register 11.6. RSA_MULT_START_REG (0x0814)

RSA_MULT_START 写入 1 以开始乘法运算。(只写)

Register 11.7. RSA_IDLE_REG (0x0818)

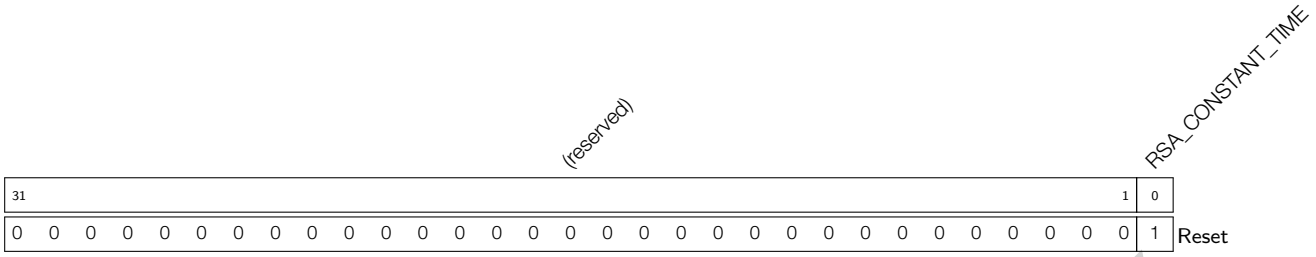
RSA_IDLE 当 RSA 空闲时，此位为 1。(只读)

Register 11.8. RSA_CLEAR_INTERRUPT_REG (0x081C)



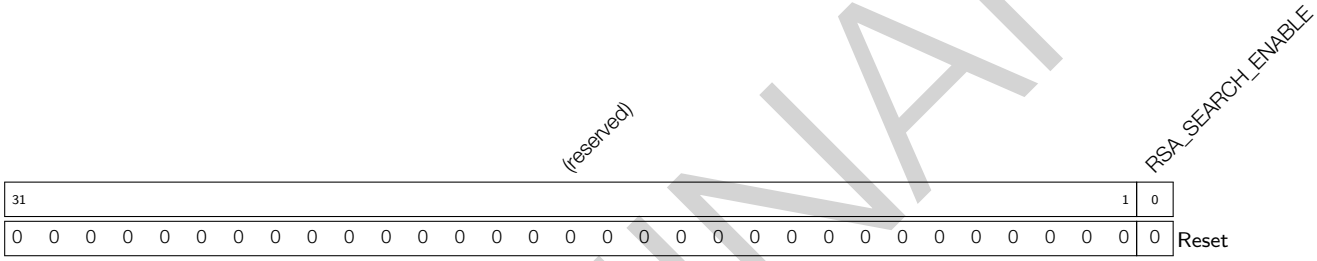
RSA_CLEAR_INTERRUPT RSA 中断清除寄存器。写入 1 清除中断。(只写)

Register 11.9. RSA_CONSTANT_TIME_REG (0x0820)



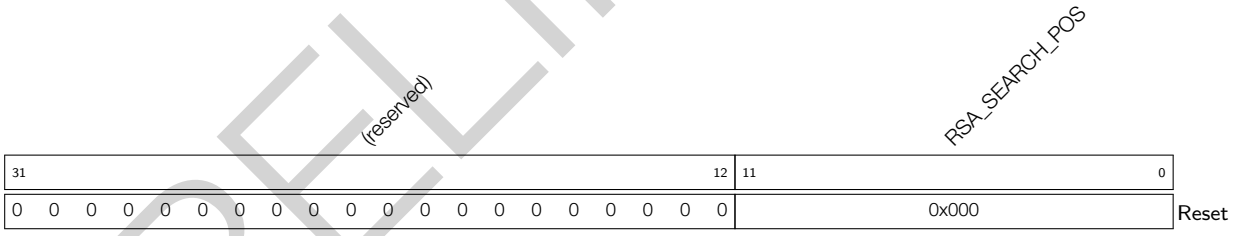
RSA_CONSTANT_TIME_REG 控制模幂运算中的 constant_time 选项。0：加速；1：不加速（默认）。（读 / 写）

Register 11.10. RSA_SEARCH_ENABLE_REG (0x0824)



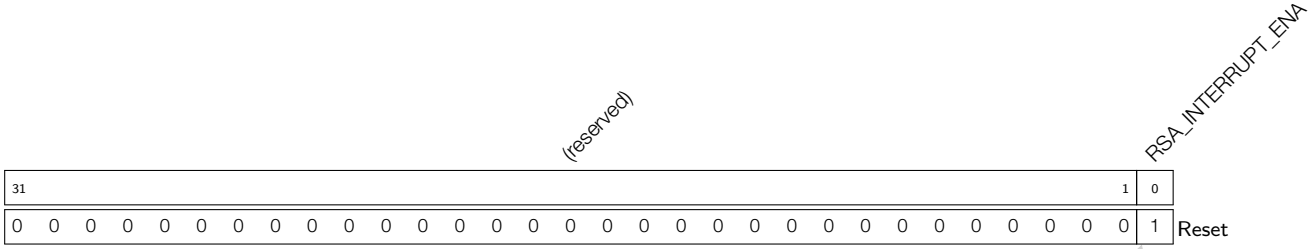
RSA_SEARCH_ENABLE 控制模幂运算中的 search 选项。1：加速；0：不加速（默认）。（读 / 写）

Register 11.11. RSA_SEARCH_POS_REG (0x0828)



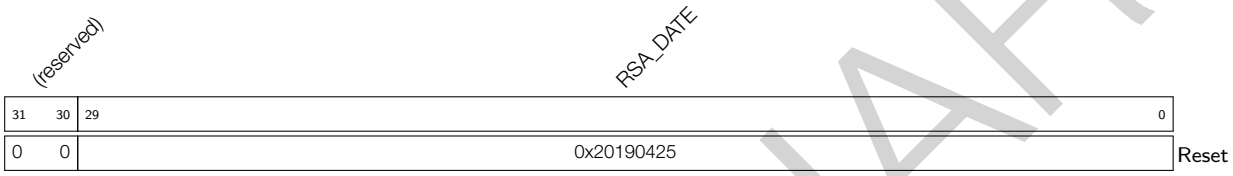
RSA_SEARCH_POS 模幂运算中的 search 加速选项。用于配置 search 的起始位置（读 / 写）。

Register 11.12. 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 11.13. RSA_DATE_REG (0x0830)



RSA_DATE 版本控制寄存器（读 / 写）。

12 数字签名 (DS)

12.1 概述

数字签名技术在密码学算法层面上用于验证消息的真实性和完整性。这可用于向服务器验证设备自身身份，或检查消息是否未被篡改。

ESP32-S3 包含一个数字签名 (Digital Signature, DS) 模块，可基于 RSA 高效生成数字签名。数字签名模块使用预先加密的参数计算出签名，HMAC 作为密钥导出函数加密这些参数。反过来，HMAC 则使用 eFuse 作为输入密钥。上述整个过程都发生在硬件层面，因此在计算过程中，不论是解密 RSA 参数的密钥还是用于 HMAC 密钥导出函数的输入密钥都对软件不可见。

12.2 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

12.3 功能描述

12.3.1 概述

DS 模块计算 RSA 签名操作 $Z = X^Y \bmod M$，其中 Z 是签名，X 是输入消息，Y 和 M 是 RSA 私钥参数。

私钥参数以密文形式存储在 flash 或其他存储器中。对其解密而使用的密钥（DS_KEY）只能由 DS 模块通过 HMAC 模块获取，并且 HMAC 模块求解该密钥所需的一切输入信息（$HMAC\_KEY$）只存放在 eFuse 中且只允许被 HMAC 模块访问。这意味着只有 DS 硬件才能解密私钥密文，软件绝对不会获取私钥明文。关于 eFuse 和 HMAC 的相关细节请参照章节 5 *eFuse 控制器 (eFuse) [to be added later]* 和章节 15 *HMAC 加速器 (HMAC) [to be added later]*。

需要签名时，软件直接将输入消息 X 发送到 DS 外设。RSA 签名操作完成后，软件将读取签名结果 Z。

为方便描述，这里约定几个符号，它们的作用域局限于本章。

- 1^s　　表示一个完全由“1”组成的长度为 s 位的位串。
- $[x]_s$　　一个长度为 s 位的位串，要求 s 为 8 的整数倍。如果 x 是一个数 ($x < 2^s$)，那么其在位串中遵循小端字节序。x 可以是一个变量，例如 $[Y]_{4096}$，或一个十六进制的常数，比如 $[0x0C]_8$。根据需要，$[x]_t$ 右边可以加上$(s-t)$ 个 0，使字串长度扩展成 s 位，得到 $[x]_s$。例如：$[0x05]_8 = 00000101$，$[0x05]_{16} = 0000010100000000$，$[0x0005]_{16} = 0000000000000101$，$[0x13]_8 = 00010011$，$[0x13]_{16} = 0001001100000000$，$[0x0013]_{16} = 0000000000010011$。
- $||$　　表示位串粘接操作符，用于将两个位串前后粘成一个较长的位串。

12.3.2 私钥运算子

私钥运算子 Y（私钥指数）和 M（密钥模数）由用户生成。它们具有特定的 RSA 密钥长度（最大为 4096 位）。RSA 签名操作还额外需要两个运算子，即参数 $\overline{r}$ 和 M'，这两个参数需要软件由 Y 和 M 运算得到。

运算子 Y、M、$\overline{r}$ 和 M' 与验证摘要一起由用户加密并以密文 C 的形式存储。密文 C 输入到 DS 模块之后先由硬件解密，然后参与 RSA 签名运算。如何生成 C 请参考第 12.3.3 节。

DS 模块支持运算子长度为 $N = 32 \times x$ ($x \in \{1, 2, 3, \ldots, 128\}$) 的 RSA 签名运算 $Z = X^Y \bmod M$，需要满足 RSA 计算的运算子长度要求，即 Z、X、Y、M 和 $\overline{r}$ 的位宽必须相同，但必须为这 128 种中的其中一种，而 M' 的位宽始终是 32。更多 RSA 计算相关信息请参考章节 11 *RSA 加速器 (RSA)* 中的 11.3.1 大数模幂运算 部分。

12.3.3 软件需要做的准备工作

如果用户想使用 DS 模块进行数字签名，软件和硬件必须紧密配合才可以顺利完成，并且软件需要做一系列准备工作，如图 12-1 所示。图中左半边给出了在硬件开始 RSA 签名计算之前，软件需要做哪些准备工作。图中右半边展示了硬件在整个签名计算的过程中具体会做些什么。

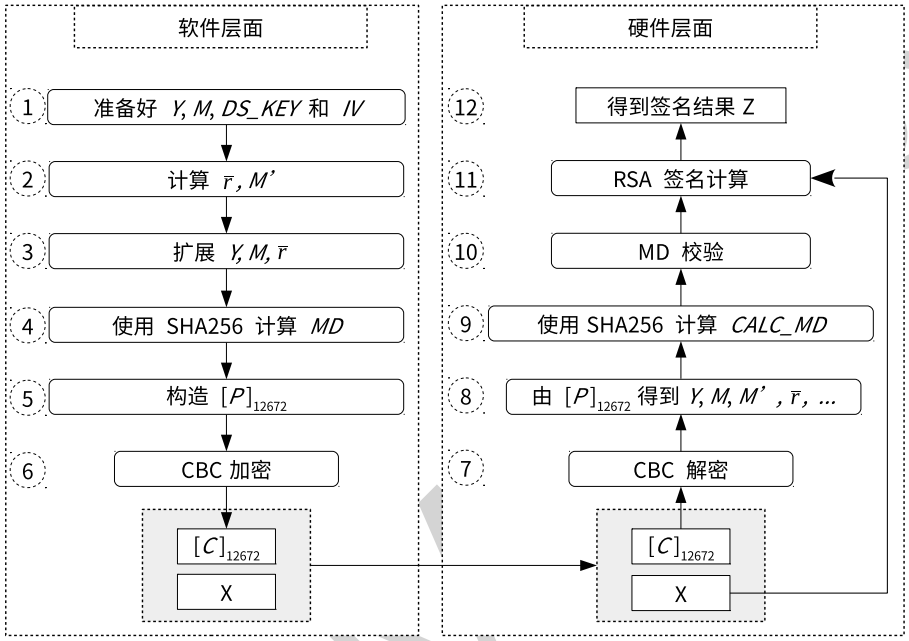


图 12-1. 软件准备工作与硬件工作流程

说明：

1. 对于多次签名计算，软件准备工作（图 12-1 左侧）只需要进行一次，但对于每一次签名计算，硬件运算（图 1-1 右侧）都需要重新进行。

用户需要依照图 12-1 指定的步骤来计算 C。详细的过程描述如下：

- **步骤 1**：按照第 12.3.2 节所述准备好 RSA 私钥运算子 Y 和 M，它们应符合运算子的长度要求。记 $[L]_{32} = \frac{N}{32}$（比如，对于 RSA 4096，$[L]_{32}$ == [0x80]$_{32}$）。还需要准备好 $[HMAC\_KEY]_{256}$，并计算出 $[DS\_KEY]_{256}$，即 DS_KEY = HMAC-SHA256($[HMAC\_KEY]_{256}$, 1^{256})。另外还需要引入一个随机的 $[IV]_{128}$，它需要符合 AES-CBC 块加密算法的要求。有关 AES 更多信息，请参考章节 10 *AES 加速器 (AES)*。
- **步骤 2**：根据 M 求解 $\overline{r}$ 和 M'。
- **步骤 3**：扩展 Y、M 和 $\overline{r}$，得到 $[Y]_{4096}$、$[M]_{4096}$ 和 $[\overline{r}]_{4096}$。由于 Y、M 和 $\overline{r}$ 的最大位宽为 4096，运算子位宽小于 4096 需要扩展为 4096，位宽等于 4096 则不需要扩展。
- **步骤 4**：使用 SHA-256 计算 MD 校验码：$[MD]_{256}$ = SHA256 ($[Y]_{4096}||[M]_{4096}||[\overline{r}]_{4096}||[M']_{32}||[L]_{32}||[IV]_{128}$)
- **步骤 5**：构造 $[P]_{12672}$ = ($[Y]_{4096}||[M]_{4096}||[\overline{r}]_{4096}||[MD]_{256}||[M']_{32}||[L]_{32}||[\beta]_{64}$)，其中 $[\beta]_{64}$ 是符合 PKCS#7 封装方式的追加码，由 8 个字节码 0x80 组成的一个 64 位的位串 $[0x0808080808080808]_{64}$，目的在于使 P 的长度为 128 位的整数倍。

- **步骤 6**：计算密文形式的私钥参数 $C = [C]_{12672}$ = AES-CBC-ENC ($[P]_{12672}$, $[DS\_KEY]_{256}$, $[IV]_{128}$)，长度为 1584 字节。

12.3.4 硬件工作流程

每次需要计算数字签名时，都会触发硬件操作。硬件需要三个输入信息：预先生成的私钥密文 C、唯一的消息 X、IV。

DS 模块的工作流程可以分为如下三个阶段：

1. **解析阶段，即图 12-1 中的步骤 7 和步骤 8**

   解析过程是图 12-1 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 12-1 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 12-1 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 12-1 中的步骤 11 和步骤 12**

   DS 模块将把用户输入的 X，以及解析得到的 Y、M 和 $\overline{r}$ 都视为大数，结合解析得到的 M'，构成了大数模幂运算 X^Y mod M 的所有必要输入参数。大数模幂运算的运算长度由 L 的值唯一指定。DS 模块调用 RSA 加速器完成大数模幂运算 $Z = X^Y$ mod M，Z 为签名结果。

12.3.5 软件工作流程

每次需要计算数字签名时，都应执行以下软件操作。输入消息是预先生成的私钥密文 C、唯一的消息 X、IV。这些软件步骤触发章节 12.3.4 中描述的硬件工作流程。下述流程基于一个假设：软件已经调用了 HMAC 外设，硬件上 HMAC 已经根据 $HMAC\_KEY$ 计算出了 DS_KEY。

1. **准备工作**：准备好 C、X、IV。具体方法请参考章节 12.3.3 。
2. **启动 DS**：对寄存器 DS_SET_START_REG 写 1。
3. **检查 DS_KEY 是否已经准备好**：轮询寄存器 DS_QUERY_BUSY_REG 直到读到 0。

   如果 DS_QUERY_BUSY_REG 超过 1 ms 还没读到 0，则说明 HMAC 未被调用。此时，软件应当读寄存器 DS_QUERY_KEY_WRONG_REG，根据返回值判断具体是哪一种情况。

   - 如果读到零值，说明 HMAC 未被调用。

- 如果读到非零值 (1 ~ 15)，则说明 HMAC 被调用过，但是 DS 模块没有拿到 DS_KEY，原因可能是有其他程序的干扰。

4. **配置寄存器**：将 IV block 写入寄存器 DS_IV_*m*_REG (*m*: 0-3)。有关 IV block 的更多信息，请参考章节 10 *AES 加速器 (AES)*。
5. **将 X 写入存储器 DS_X_MEM**：将 X_i ($i \in \{0, 1, \ldots, n-1\}$) 写入存储器 DS_X_MEM，容量为 128 个字 (word)，其中 $n = \frac{N}{32}$。每一个字刚好存放一个 b 进制数。存储器的低地址存放运算子的低位进制数，高地址存放运算子的高位进制数。当 X 的长度小于 128 个字时，存储器 DS_X_MEM 中有一部分空间未使用，该部分空间中的数据可以是任意值。
6. **将 C 写入存储器 DS_C_MEM**：将 C_i ($i \in \{0, 1, \ldots, 395\}$) 写入存储器 DS_C_MEM，容量为 396 个字。每一个字刚好存放一个 b 进制数。
7. **启动计算**：对寄存器 DS_SET_ME_REG 写入 1。
8. **等待运算结束**：轮询寄存器 DS_QUERY_BUSY_REG 直到读到 0。
9. **检查校验结果**：读寄存器 DS_QUERY_CHECK_REG，根据返回值决定后续操作。
   - 如果返回值为 0，则说明填充校验通过，MD 校验通过，可以继续读取 Z 结果值。
   - 如果返回值为 1，则说明填充校验通过，但 MD 校验失败。Z 结果值全零无效，跳至步骤 11。
   - 如果返回值为 2，则说明填充校验通过失败，但 MD 校验通过，用户可以继续读取 Z 结果值。但仍需注意的是，数据填充不符合 PKCS#7 封装方式，这可能不是你想要的。
   - 如果返回值为 3，则说明填充填充校验失败，且 MD 校验失败。这种情况说明有致命错误发生。Z 结果值全零无效。跳至步骤 11。
10. **读出运算结果**：从存储器 DS_Z_MEM 读出运算结果 Z_i ($i \in \{0, 1, \ldots, n-1\}$)，其中 $n = \frac{N}{32}$。Z 以小端字节序存储在存储器中。
11. **退出计算环境**：对寄存器 DS_SET_FINISH_REG 写入 1，然后轮询寄存器 DS_QUERY_BUSY_REG 直到读到 0。

DS 退出计算环境后，所有输入/输出寄存器和存储器中的数据都已经被抹除（清零）。

12.4 存储器列表

请注意，这里的起始地址和结束地址都是相对于 Digital Signature 基地址的地址偏移量（相对地址）。请参阅章节 1 系统和存储器 中的表 1-4 获取 DS 模块的基地址。

名称	描述	大小（字节）	起始地址	结束地址	访问
DS_C_MEM	存储器 C	1584	0x0000	0x062F	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

12.5 寄存器列表

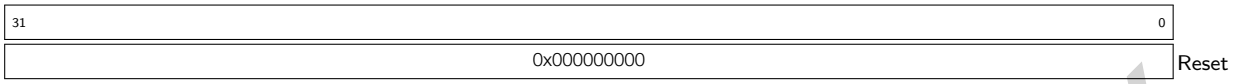
本小节的所有地址均为相对于 Digital Signature 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

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

12.6 寄存器

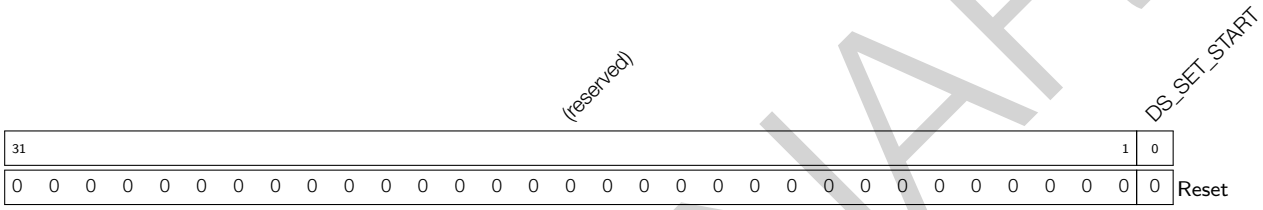
本小节的所有地址均为相对于 Digital Signature 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

Register 12.1. DS_IV_*m*_REG (*m*: 0-3) (0x0630+4m*)**



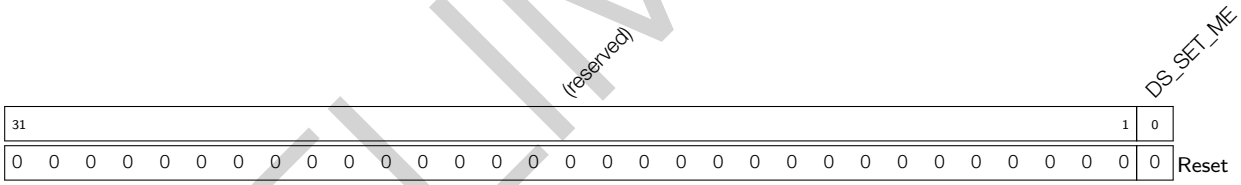
DS_IV_*m*_REG (*m*: 0-3) *IV* block 数据。（只写）

Register 12.2. DS_SET_START_REG (0x0E00)



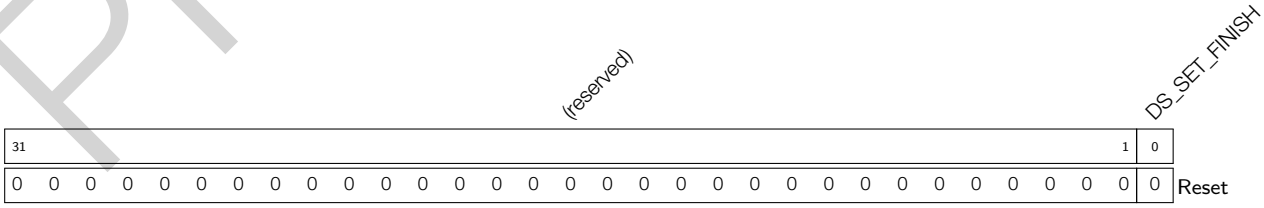
DS_SET_START 写入 1 启动 DS 模块。（只写）

Register 12.3. DS_SET_ME_REG (0x0E04)



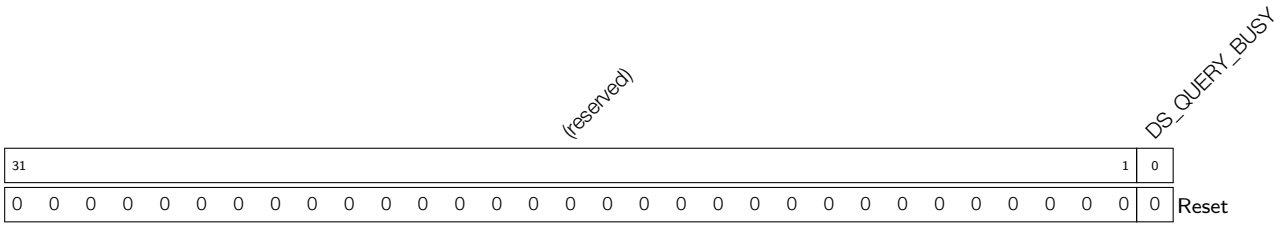
DS_SET_ME 写入 1 以开始计算。（只写）

Register 12.4. DS_SET_FINISH_REG (0x0E08)



DS_SET_FINISH 写入 1 以结束运算。（只写）

Register 12.5. DS_QUERY_BUSY_REG (0x0E0C)

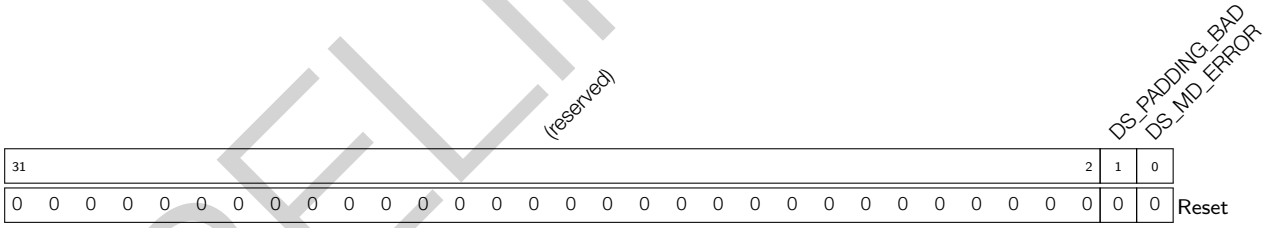


DS_QUERY_BUSY 1：DS 模块正在忙；0：DS 模块空闲。（只读）

Register 12.6. 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）

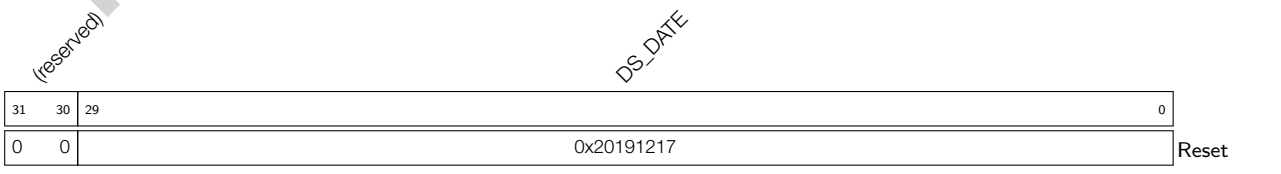
Register 12.7. DS_QUERY_CHECK_REG (0x0E14)



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 12.8. DS_DATE_REG (0x0E20)



DS_DATE 版本控制寄存器。（读/写）

13 片外存储器加密与解密 (XTS_AES)

13.1 概述

ESP32-S3 芯片集成了片外存储器加密与解密模块，采用符合 IEEE Std 1619-2007 指定的 XTS-AES 标准的算法，为用户存放在片外存储器（片外 flash 与片外 RAM）的应用代码和数据提供了安全保障。用户可以将专有软件、敏感的用户数据（如用来访问私有网络的凭据）存放在片外 flash 中，或将通用数据存放在片外 RAM 中。

13.2 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

13.3 模块结构

片外存储器加解密模块包含三个部分：手动加密 (Manual Encryption) 模块、自动加密 (Auto Encryption) 模块、自动解密 (Auto Decryption) 模块。结构图如图 13-1 所示。

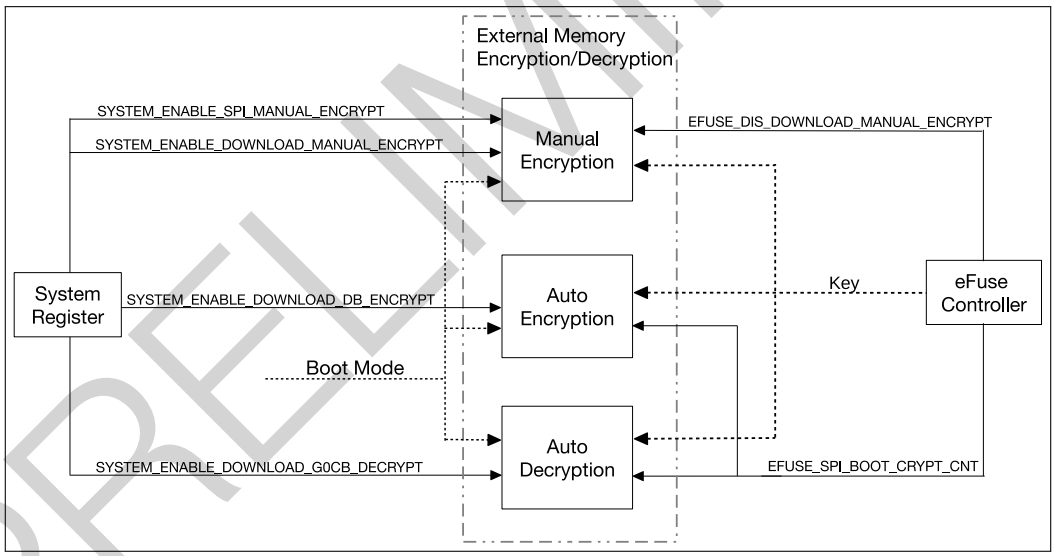


图 13-1. 片外存储器加解密工作配置

手动加密模块能够对指令/数据进行加密，指令/数据将以密文状态通过 SPI1 被写入片外 flash。

当 CPU 通过 cache 写片外 RAM 时，自动加密模块会先对数据自动进行加密，数据将以密文状态被写入片外 RAM。

当 CPU 通过 cache 读片外 flash 或片外 RAM 时，自动解密模块将对读取到的密文自动进行解密以恢复指令和数据。

系统寄存器 (SYSREG) 外设中 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 内的以下 4 个位与片外存储器加解密相关：

- SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT
- SYSTEM_ENABLE_DOWNLOAD_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。

13.4 功能描述

13.4.1 XTS 算法

不论是手动加密，还是自动加/解密，使用的是同一种算法——XTS 算法。根据算法特征，在具体实现中，使用 1024 位为一个数据单元 (data unit)，此处 “data unit” 由 *XTS-AES Tweakable Block Cipher* 标准中的章节 *XTS-AES encryption procedure* 定义。更多关于 XTS-AES 算法的信息，请参考 IEEE Std 1619-2007。

13.4.2 密钥 Key

在执行 XTS 运算时，手动加密模块、自动加密模块和自动解密模块使用完全相同的密钥 Key 。密钥 Key 来自硬件 eFuse，且无法被用户访问获取。

密钥 Key 支持两种长度：256 位、512 位。Key 的值和长度完全由 eFuse 参数信息决定。为方便阐述如何通过 eFuse 参数信息推导出 Key 的值，现作如下约定：

- $Block_A$：指 BLOCK4 ~ BLOCK9 中 key purpose（密钥用途）的值等于 EFUSE_KEY_PURPOSE_XTS_AES_256_KEY_1 的 BLOCK。如果 $Block_A$ 存在，那么 $Block_A$ 中存放着 256 位的 Key_A。
- $Block_B$：指 BLOCK4 ~ BLOCK9 中密钥用途的值等于 EFUSE_KEY_PURPOSE_XTS_AES_256_KEY_2 的 BLOCK。如果 $Block_B$ 存在，那么 $Block_B$ 中存放 256 位的 Key_B。
- $Block_C$：指 BLOCK4 ~ BLOCK9 中密钥用途的值等于 EFUSE_KEY_PURPOSE_XTS_AES_128_KEY 的 BLOCK。如果 $Block_C$ 存在，那么 $Block_C$ 中存放 256 位的 Key_C。

根据 $Block_A$、$Block_B$ 和 $Block_C$ 是否存在，可以获得 5 种组合。在不同组合情况下，Key 值可以由 Key_A、Key_B、Key_C 的值惟一确定，如表 13-1 所示。

表 13-1. 根据 Key_A、Key_B、Key_C 生成 Key 值

$Block_A$	$Block_B$	$Block_C$	生成 Key 值	Key 长度（位）
Yes	Yes	无关项	$Key_A \| \| Key_B$	512
Yes	No	无关项	$Key_A \| \| 0^{256}$	512
No	Yes	无关项	$0^{256} \| \| Key_B$	512
No	No	Yes	Key_C	256
No	No	No	0^{256}	256

Notes:

表 13-1 中的 “Yes” 指存在，“No” 指不存在，“0^{256}” 表示 256 个位”0” 组成的位串，“||” 表示将两个比特串按照前后顺序合成一个更长的新位串。

更多有关密钥用途的信息，请参考章节 5 *eFuse 控制器 (eFuse) [to be added later]* 中的表 *密钥 Key*。

13.4.3 目标空间

目标空间是指单次加密后的密文将要存放在片外存储器中的哪一段连续的地址空间中。目标空间可以由目标类型、目标尺寸、目标基地址这三个参数唯一确定。这三个参数的定义如下：

- 目标类型：目标空间的类型 ($type$)，片外 flash 或片外 RAM。目标类型的值为 0 时指片外 flash，值为 1 时指片外 RAM。
- 目标尺寸：目标空间的大小 ($size$)，以字节为单位，即单次对多少数据进行加密。只有 16、32 和 64 可选。
- 目标基地址：目标空间的基地址 ($base\_addr$)，这是一个 30-bit 的物理地址，取值范围为 0x0000_0000 ~ 0x3FFF_FFFF，但要求以目标尺寸为单位对齐，即 $base\_addr\%size == 0$。

如某一次加密操作，要将 16 字节的指令数据加密后存放在片外 flash 中的地址段 0x130 ~ 0x13F 中，则目标空间为 0x130 ~ 0x13F，目标类型为 0 (片外 flash)，目标尺寸为 16 (字节)，目标基地址为 0x130。

对于任意长度（必须是 16 字节的整数倍）的明文指令/数据的加密，可以将整个加密过程拆分成多次进行，每次都有各自的目标空间参数。

对于自动加/解密模块，目标空间等参数由硬件自动调节。对于手动加密模块，目标空间等参数需要用户主动配置。

说明：

IEEE Std 1619-2007 中的章节 *5.1 Data units and tweaks* 中定义的 “tweak” 是一个 128-bit 的非负整数 ($tweak$)，其值可以通过公式求出：$tweak = type * 2^{30} + (base\_addr$ & 0x3FFFFF80)。显然，$tweak$ 中最低的 7 个比特和最高的 97 个比特恒为零。

13.4.4 数据填充

对于自动加/解密模块，数据的填充由硬件自动完成。对于手动加密模块，数据的填充需要用户主动配置。手动加密模块中由 16 个寄存器 XTS_AES_PLAIN_n_REG (n: 0-15) 构成的寄存器块专用于数据填充，一次可以存放最多 512 位明文指令/数据。

实际上，手动加密模块不在乎明文来自什么地方，只注重密文将要存放在什么地方。考虑到明文和密文之间呈严格的对应关系，为了更好地描述明文如何封装在寄存器块中，现假设明文从一开始就放在目标空间中，并在加密完成后被密文替换。因此，接下来的描述不再出现 “明文” 这个概念，而用 “目标空间” 来代替。但请注意，在真正使用时，明文可以来自任何地方，但用户必须清晰知道明文如何封装在寄存器块中。

目标空间映射到寄存器块的方法为：

假设目标空间中某一 word 的存放地址为 $address$，记 $offset = address\%64$，$n = \frac{offset}{4}$，那么该 word 将被存放在编号为 n 的寄存器 XTS_AES_PLAIN_n_REG 中。

例如，当目标尺寸为 64 时，寄存器块中的所有寄存器都将被使用，目标空间中的地址与寄存器块之间的填充映射关系如表 13-2 所示。

表 13-2. 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

$offset$	寄存器	$offset$	寄存器
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

13.4.5 手动加密模块

手动加密模块是一个外设模块，自身带有寄存器，可以被 CPU 直接访问。模块内的寄存器、系统寄存器 (SYSREG) 外设、eFuse 参数、boot 模式共同配置并使用这一模块。请注意，手动加密模块只能加密片外 flash。

当且仅当手动加密模块拥有工作权限时，才允许手动加密。手动加密模块是否拥有工作权限取决于：

- SPI Boot 模式下

  当寄存器 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 的 SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT 位为 1 时，手动加密模块拥有工作权限，否则无法工作。

- Download Boot 模式下

  当寄存器 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 的 SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT 位为 1，且 eFuse 参数 EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT 为 0 时，手动加密模块拥有工作权限，否则无法工作。

说明：

- 虽然 CPU 可以不通过 cache 而直接读片外存储器从而得到加密指令/数据，但软件还是绝对无法获取到密钥 Key。

13.4.6 自动加密模块

自动加密并非外设模块，自身不带寄存器，不能被 CPU 直接访问。系统寄存器 (SYSREG) 外设、eFuse 参数、boot 模式共同配置并使用这一模块。

当且仅当自动加密模块拥有工作权限时，才允许自动加密。自动加密模块是否拥有工作权限取决于：

- SPI Boot 模式下

  当参数 SPI_BOOT_CRYPT_CNT（3 位宽）中有奇数个位为 1 时，自动加密模块拥有工作权限，否则无法工作。

- Download Boot 模式下

  当寄存器 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 的 SYSTEM_ENABLE_DOWNLOAD_DB_ENCRYPT 位为 1 时，自动加密模块拥有工作权限，否则无法工作。

说明：

- 当自动加密模块拥有工作权限时，如果 CPU 通过 cache 写访问片外 RAM，自动加密模块将自动对数据进行加密，然后将加密结果写到片外 RAM。加密的整个过程无需软件参与并且对 cache 是透明的。加密算法过程中密钥 Key 绝对无法被软件获取。
- 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。

13.4.7 自动解密模块

自动解密并非外设模块，自身不带寄存器，不能被 CPU 直接访问。系统寄存器 (SYSREG) 外设、eFuse 参数、boot 模式共同配置并使用这一模块。

当且仅当自动解密模块拥有工作权限时，才允许自动解密。自动解密模块是否拥有工作权限取决于：

- SPI Boot 模式下

  当参数 SPI_BOOT_CRYPT_CNT（3 位宽）中有奇数个位为 1 时，自动解密模块拥有工作权限，否则无法工作。

- Download Boot 模式下

  当寄存器 SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG 的 SYSTEM_ENABLE_DOWNLOAD_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。解密算法过程中密钥 Key 绝对无法被软件获取。
- 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。

13.5 软件流程

手动加密模块工作时需要软件参与，软件流程为：

1. 配置 XTS_AES：
   - 将寄存器 XTS_AES_DESTINATION_REG 的值设置为 $type$ = 0。
   - 将寄存器 XTS_AES_PHYSICAL_ADDRESS_REG 的值设置为 $base\_addr$。
   - 将寄存器 XTS_AES_LINESIZE_REG 的值设置为 $\frac{size}{32}$。

   关于 $type$、$base\_addr$、$size$ 的定义，请参考章节 13.4.3。
2. 用明文填充寄存器块 XTS_AES_PLAIN_*n*_REG (*n*: 0-15)。请参考章节 13.4.4 获取相关信息。
   只需要根据需要填充寄存器，不需要使用的寄存器可以是任意值。
3. 轮询寄存器 XTS_AES_STATE_REG 直到读到 0，确保手动加密模块是空闲的。
4. 启动计算。对寄存器 XTS_AES_TRIGGER_REG 写入 1。
5. 等待加密完成。轮询寄存器 XTS_AES_STATE_REG，直到读到 2。
   步骤 1 至 5 操作手动加密模块对明文指令进行加密。加密算法使用的密钥就是 Key 。
6. 下放密文访问权限给 SPI1。对寄存器 XTS_AES_RELEASE_REG 写入 1，使得加密结果允许被 SPI1 获取。之后如果读取寄存器 XTS_AES_STATE_REG 将读到 3。

7. 调用 SPI1，将加密结果写入片外 flash（请参阅章节 11 *SPI 控制器 (SPI) [to be added later]*）。
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。

13.6 寄存器列表

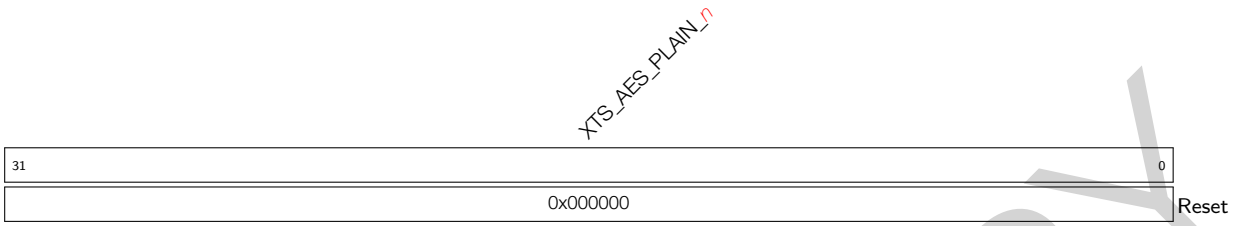
本小节的所有地址均为相对于 External Memory Encryption and Decryption 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

名称	描述	地址	访问
明文寄存器堆			
XTS_AES_PLAIN_0_REG	明文寄存器 0	0x0000	读/写
XTS_AES_PLAIN_1_REG	明文寄存器 1	0x0004	读/写
XTS_AES_PLAIN_2_REG	明文寄存器 2	0x0008	读/写
XTS_AES_PLAIN_3_REG	明文寄存器 3	0x000C	读/写
XTS_AES_PLAIN_4_REG	明文寄存器 4	0x0010	读/写
XTS_AES_PLAIN_5_REG	明文寄存器 5	0x0014	读/写
XTS_AES_PLAIN_6_REG	明文寄存器 6	0x0018	读/写
XTS_AES_PLAIN_7_REG	明文寄存器 7	0x001C	读/写
XTS_AES_PLAIN_8_REG	明文寄存器 8	0x0020	读/写
XTS_AES_PLAIN_9_REG	明文寄存器 9	0x0024	读/写
XTS_AES_PLAIN_10_REG	明文寄存器 10	0x0028	读/写
XTS_AES_PLAIN_11_REG	明文寄存器 11	0x002C	读/写
XTS_AES_PLAIN_12_REG	明文寄存器 12	0x0030	读/写
XTS_AES_PLAIN_13_REG	明文寄存器 13	0x0034	读/写
XTS_AES_PLAIN_14_REG	明文寄存器 14	0x0038	读/写
XTS_AES_PLAIN_15_REG	明文寄存器 15	0x003C	读/写
配置寄存器			
XTS_AES_LINESIZE_REG	加密块大小寄存器	0x0040	读/写
XTS_AES_DESTINATION_REG	加密类型寄存器	0x0044	读/写
XTS_AES_PHYSICAL_ADDRESS_REG	物理地址寄存器	0x0048	读/写
控制/状态寄存器			
XTS_AES_TRIGGER_REG	启动运算	0x004C	只写
XTS_AES_RELEASE_REG	释放控制	0x0050	只写
XTS_AES_DESTROY_REG	销毁控制	0x0054	只写
XTS_AES_STATE_REG	状态寄存器	0x0058	只读
版本寄存器			
XTS_AES_DATE_REG	版本控制寄存器	0x005C	只读

13.7 寄存器

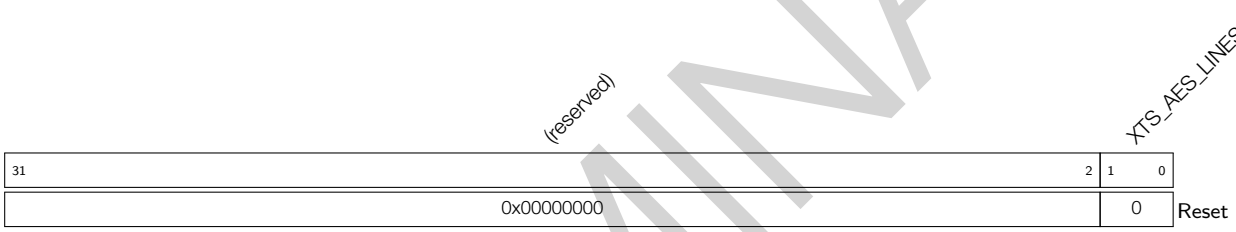
本小节的所有地址均为相对于 External Memory Encryption and Decryption 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

Register 13.1. XTS_AES_PLAIN_*n*_REG (*n*: 0-15) (0x0000+4n*)**



XTS_AES_PLAIN_*n* 存储明文的第 *n* 个 32 位部分。（读/写）

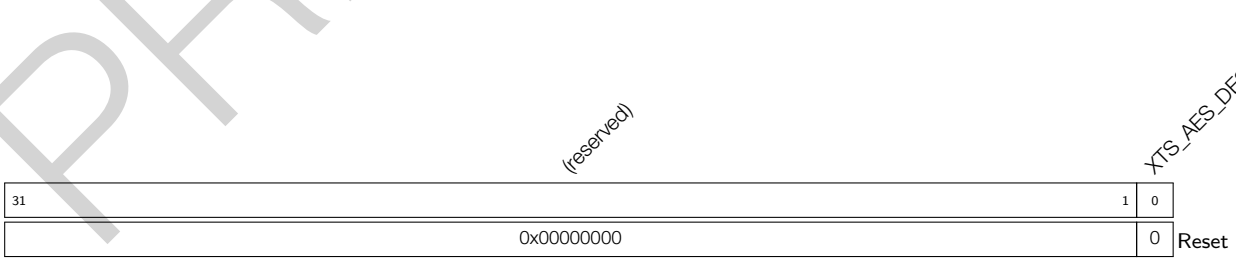
Register 13.2. XTS_AES_LINESIZE_REG (0x0040)



XTS_AES_LINESIZE 块大小寄存器，决定单次加密的数据量。

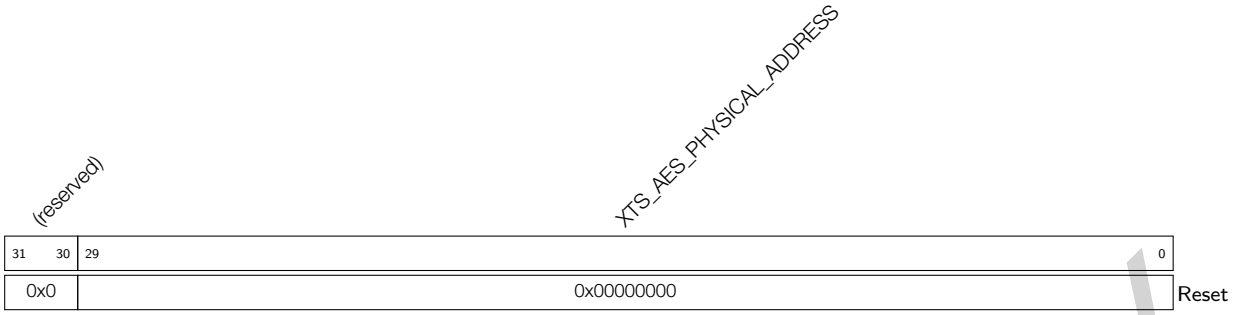
- 0：加密 16 bytes；
- 1：加密 32 bytes；
- 2：加密 64 bytes。（读/写）

Register 13.3. XTS_AES_DESTINATION_REG (0x0044)



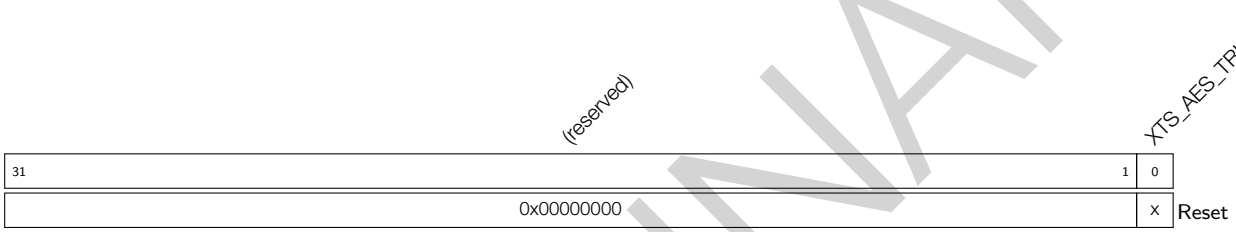
XTS_AES_DESTINATION 决定手动加密类型，目前只能手动加密 flash，所以只能为 0。用户不能写入 1，否则将发生错误。0：加密 flash；1：加密片外 RAM。（读/写）

Register 13.4. XTS_AES_PHYSICAL_ADDRESS_REG (0x0048)



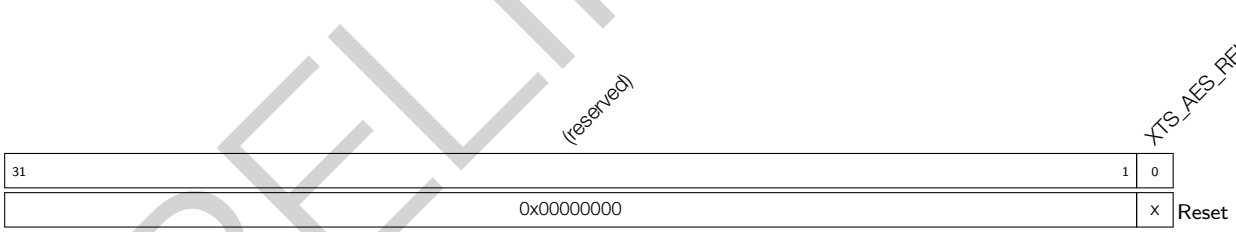
XTS_AES_PHYSICAL_ADDRESS 物理地址。(读/写)

Register 13.5. XTS_AES_TRIGGER_REG (0x004C)



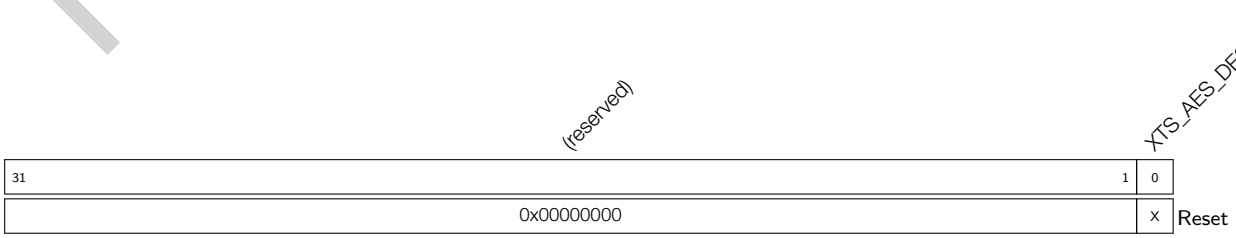
XTS_AES_TRIGGER 写入 1 使能手动加密运算。(只写)

Register 13.6. XTS_AES_RELEASE_REG (0x0050)



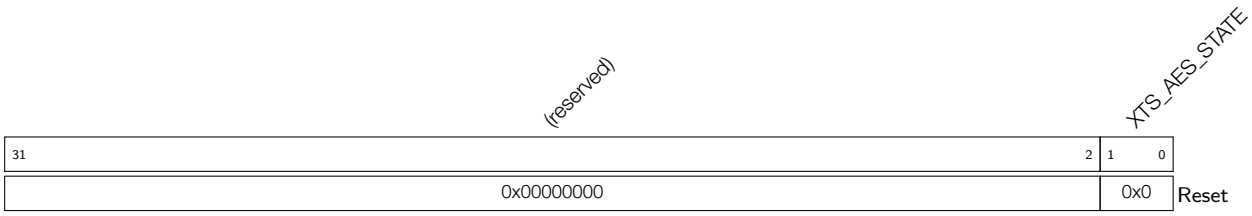
XTS_AES_RELEASE 写入 1 使加密结果对 SPI1 可见，因而 SPI1 可以获取到加密结果。(只写)

Register 13.7. XTS_AES_DESTROY_REG (0x0054)



XTS_AES_DESTROY 写入 1 销毁加密结果。(只写)

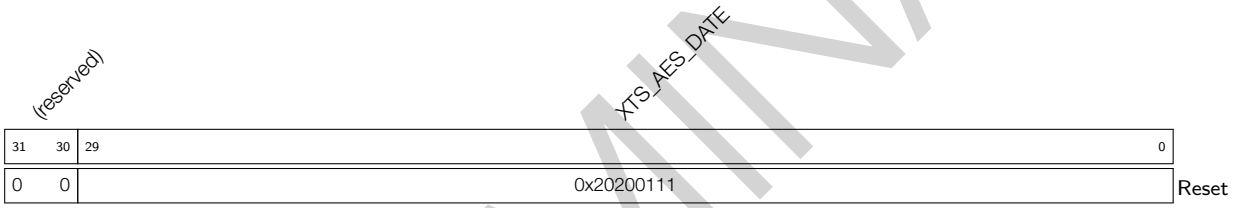
Register 13.8. XTS_AES_STATE_REG (0x0058)



XTS_AES_STATE 手动加密模块状态寄存器。

- 0x0 (XTS_AES_IDLE)：空闲；
- 0x1 (XTS_AES_BUSY)：忙于计算；
- 0x2 (XTS_AES_DONE)：计算完成，但手动加密结果数据对 SPI 不可见；
- 0x3 (XTS_AES_RELEASE)：手动加密结果对 SPI 可见。（只读）

Register 13.9. XTS_AES_DATE_REG (0x005C)



XTS_AES_DATE 版本控制寄存器。（读/写）

14 随机数发生器 (RNG)

14.1 概述

ESP32-S3 内置一个真随机数发生器，其生成的 32 位随机数可作为加密等操作的基础。

14.2 主要特性

ESP32-S3 的随机数发生器可通过物理过程而非算法生成真随机数，所有生成的随机数在特定范围内出现的概率完全一致。

14.3 功能描述

系统可以从随机数发生器的寄存器 RNG_DATA_REG 中读取随机数，每个读到的 32 位随机数都是真随机数，噪声源为系统中的热噪声和异步时钟。

具体来说，这些热噪声可以来自 SAR ADC 或高速 ADC 或两者兼有。当芯片的 SAR ADC 或高速 ADC 工作时，就会产生比特流，并通过异或 (XOR) 逻辑运算作为随机数种子进入随机数生成器。

当为数字内核使能 RTC20M_CLK 时钟时，随机数发生器也会对 RTC20M_CLK (20 MHz) 进行采样，作为随机数种子。RTC20M_CLK 是一种异步时钟源，由于存在电路亚稳态，因此可以提高随机数发生器的熵值。然而，为了保证随机数发生器可以获得最大熵值，仍建议在使用随机数发生器时至少使能一路 ADC 作为随机数种子。

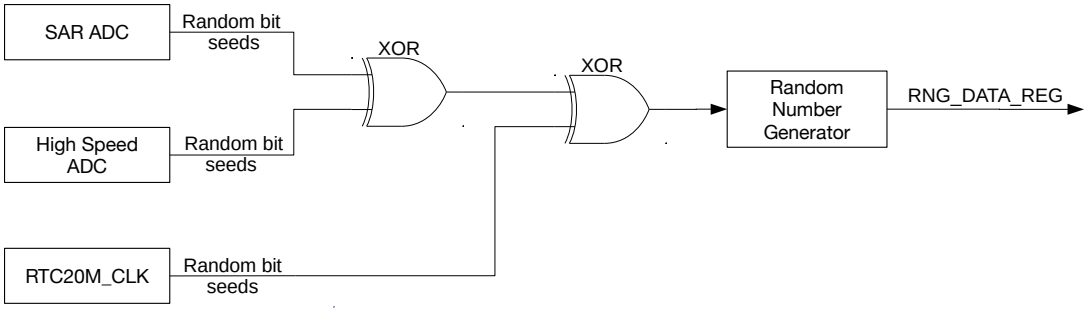


图 14-1. 噪声源

当 SAR ADC 打开时，每个 RTC20M_CLK （20 MHz）时钟周期内（来自内部 RC 振荡器，详见 3 复位和时钟 章节），随机数发生器将获得 2 位的熵。因此，为了获得最大的熵值，建议读取 RNG_DATA_REG 寄存器时的速率不超过 500 kHz。

当高速 ADC 打开时，每个 APB 时钟周期（通常为 80 MHz）内，随机数发生器将获得 2 位的熵。因此，为了获得最大的熵值，建议读取 RNG_DATA_REG 寄存器时的速率不超过 5 MHz。

我们在仅打开高速 ADC 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。

14.4 编程指南

在使用 ESP32-S3 的随机数生成器时，应该至少打开 SAR ADC，高速 ADC，或 RTC20M_CLK，否则可能会导致产生伪随机数，应注意避免。其中，

- SAR ADC 可通过 DIG ADC 控制器打开，详见 12 片上传感器与模拟信号处理 *[to be added later]* 章节。
- 高速 ADC 在 Wi-Fi 或蓝牙开启时自动打开。
- RTC20M_CLK 可通过设置 RTC_CNTL_CLK_CONF_REG 寄存器中的 RTC_CNTL_DIG_CLK20M_EN 位使能。

说明：

注意，在 Wi-Fi 开启时，极端情况下高速 ADC 有读值饱和的可能，这会降低熵值。因此，建议在 Wi-Fi 开启时，同时通过 DIG ADC1 控制器打开 SAR ADC 产生随机数。

在使用随机数生成器时，请多次读取 RNG_DATA_REG 寄存器的值，直至获得足够多的随机数。在读取寄存器时，注意控制速率不要超过上方第 14.3 小节的介绍。

14.5 寄存器列表

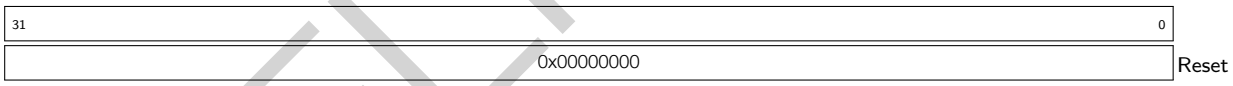
请注意，下表中的地址都是相对于随机数发生器基地址的地址偏移量（相对地址），详见章节 1 系统和存储器 中的表 1-4。

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

14.6 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），相见章节 1 系统和存储器 中的表 1-4。

Register 14.1. RNG_DATA_REG (0x0110)



RNG_DATA 随机数来源。(只读)

15 双线汽车接口 (TWAI®)

15.1 概述

双线车载串口 (Two-wire Automotive Interface, TWAI®) 协议是一种多主机、多播的通信协议，具有检测错误、发送错误信号以及内置报文优先仲裁等功能。TWAI 协议适用于汽车和工业应用（可参见第 15.3 章）。

ESP32-S3 包含一个 TWAI 控制器，可通过外部收发器连接到 TWAI 总线。TWAI 控制器包含一系列先进的功能，用途广泛，可用于如汽车产品、工业自动化控制、楼宇自动化等。

15.2 主要特性

ESP32-S3 TWAI 控制器具有以下特性：

- 兼容 ISO 11898-1 协议
- 支持标准格式（11-bit 标识符）和扩展格式（29-bit 标识符）
- 支持 1 Kbit/s ~ 1 Mbit/s 位速率
- 支持多种操作模式
  - 正常模式
  - 只听模式（不影响总线）
  - 自测模式（发送数据时不需应答）
- 64-byte 接收 FIFO
- 特殊发送
  - 单次发送（发生错误时不会自动重新发送）
  - 自发自收（TWAI 控制器同时发送和接收报文）
- 接收滤波器（支持单滤波器和双滤波器模式）
- 错误检测与处理
  - 错误计数
  - 错误报警限制可配置
  - 错误代码捕捉
  - 仲裁丢失捕捉

15.3 功能性协议

15.3.1 TWAI 性能

TWAI 协议连接网络中的两个或多个节点，并允许各节点以延迟限制的形式进行报文交互。TWAI 总线具有以下性能：

单通道通信与不归零编码： TWAI 总线由承载着位的单通道组成，因此为半双工通信。同步调整也在单通道中进行，因此不需其他通道（如时钟通道和使能通道）。TWAI 上报文的位流采用不归零编码 (NRZ) 方式。

位值：单通道可处于显性状态或隐性状态，显性状态的逻辑值为 0，隐性状态的逻辑值为 1。发送显性状态数据的节点总是比发送隐性状态数据的节点优先级高。总线上的其他物理功能（如，差分电平、单线）由其各自应用实现。

位填充：TWAI 报文的某些域已经过位填充。每发送某个相同值（如显性数值或隐性数值）的连续五个位后，需自动插入一个互补位。同理，接收到 5 个连续位的接收器应将下一个位视为填充位。位填充应用于以下域：SOF、仲裁域、控制域、数据域和 CRC 序列（可参见第 15.3.2 章）。

多播：当各节点连接到同个总线上时，这些节点将接收相同的位。各节点上的数据将保持一致，除非发生总线错误（可参见第 15.3.3 章）。

多主机：任意节点都可发起数据传输。如果当前已有正在进行的数据传输，则节点将等待当前传输结束后再发起其数据传输。

报文优先级与仲裁：若两个或多个节点同时发起数据传输，则 TWAI 协议将确保其中一个节点获得总线的优先仲裁权。各节点所发送报文的仲裁域决定哪个节点可以获得优先仲裁。

错误检测与通报：各节点将积极检测总线上的错误，并通过发送错误帧来通报检测到的错误。

故障限制：若一组错误计数依据规定增加/减少时，各节点将维护该组错误计数。当错误计数超过一定阈值时，对应节点将自动关闭以退出网络。

可配置位速率：单个 TWAI 总线的位速率是可配置的。但是，同个总线中的所有节点须以相同位速率工作。

发送器与接收器：不论何时，任意 TWAI 节点都可作为发送器和接收器。

- 产生报文的节点为发送器。且该节点将一直作为发送器，直到总线空闲或该节点失去仲裁。请注意，未丢失仲裁的多个节点都可作为发送器。
- 所有非发送器的节点都将作为接收器。

15.3.2 TWAI 报文

TWAI 节点使用报文发送数据，并在监测到总线上存在错误时向其他节点发送错误信号。报文分为不同的帧类型，某些帧类型将具有不同的帧格式。

TWAI 协议有以下帧类型：

- 数据帧
- 远程帧
- 错误帧
- 过载帧
- 帧间距

TWAI 协议有以下帧格式：

- 标准格式 (SFF) 由 11-bit 标识符组成
- 扩展格式 (EFF) 由 29-bit 标识符组成

15.3.2.1 数据帧和远程帧

节点使用数据帧向其他节点发送数据，可负载 0 ~ 8 字节数据。节点使用远程帧向其他节点请求具有相同标识符的数据帧，因此远程帧中不包含任何数据字节。但是，数据帧和远程帧中包含许多相同域。下图 15-1 所示为不

同帧类型和不同帧格式中包含的域和子域。

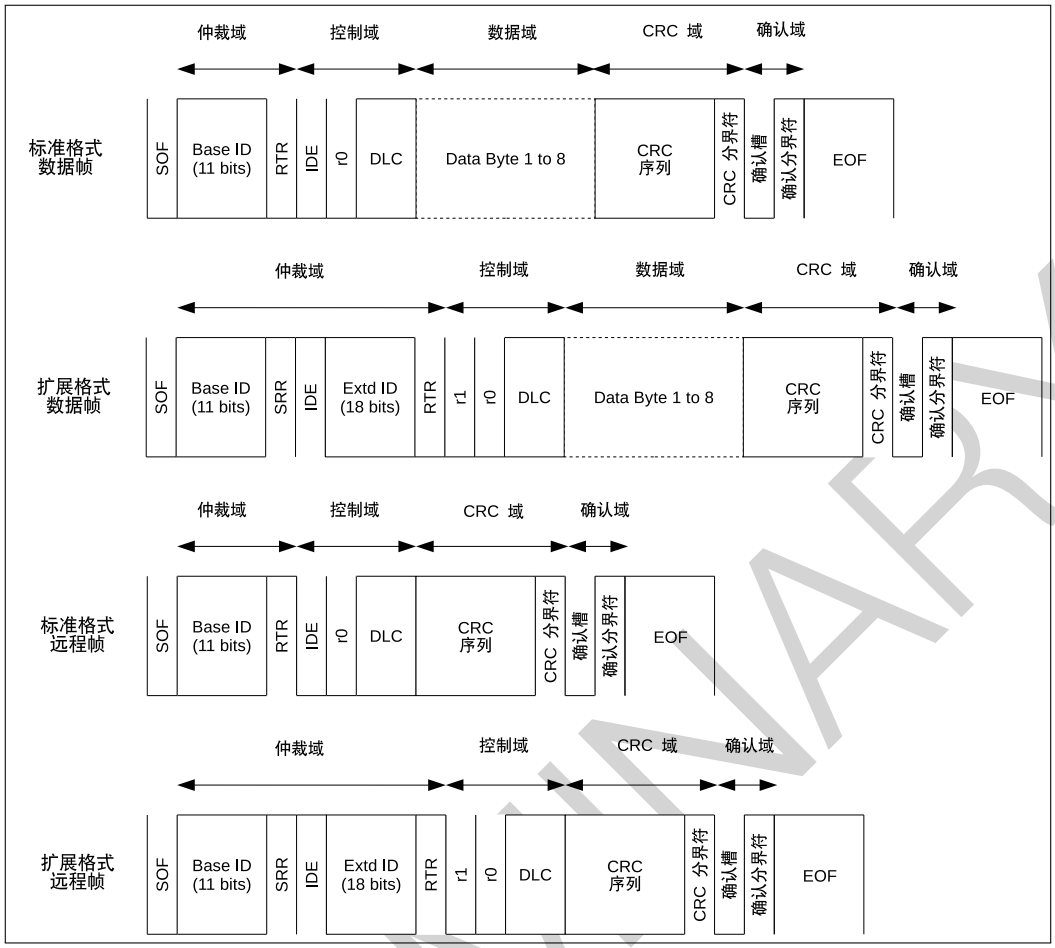


图 15-1. 数据帧和远程帧中的位域

仲裁域

当两个或多个节点同时发送数据帧和远程帧时，将根据仲裁域的位信息来决定总线上获得优先仲裁的节点。在仲裁域作用时，如果一个节点在发送隐性位的同时检测到了一个显性位，这表示有其他节点优先于了这个隐性位。那么，这个发送隐性位的节点已丢失总线仲裁，应立即转为接收器。

仲裁域主要由优先发送的最高有效位的帧标识符组成。根据显性位代表的逻辑值为 0，隐性位代表的逻辑值为 1，有以下规律:

- ID 值最小的帧将总是获得仲裁。
- 如果 ID 和格式相同，由于数据帧的 RTR 位为显性位，数据帧将优先于远程帧。
- 如果 ID 的前 11 位相同，由于扩展帧的 SRR 位是隐性，因而标准格式帧将总优先于扩展格式帧。

控制域

控制域主要由数据长度代码 (DLC) 组成，DLC 表示一个数据帧中的负载的数据字节数量，或一个远程帧请求的数据字节数量。DLC 优先发送最高有效位。

数据域

数据域中包含一个数据帧真实负载的数据字节。远程帧中不包含数据域。

CRC 域

CRC 域主要由 CRC 序列组成。CRC 序列是一个 15-bit 的循环冗余校验编码，根据数据帧或远程帧中的未填充内容（从 SOF 到数据域末尾的所有内容）中计算而来。

确认域

确认 (ACK) 域由确认槽和确认分界符组成，主要功能为：接收器向发送器报告已正确接收到有效报文。

表 15-1. SFF 和 EFF 中的数据帧和远程帧

数据/远程帧	描述
SOF	帧起始 (SOF) 是一个用于同步总线上节点的单个显性位。
Base ID	基标识符 (ID.28 ~ ID.18) 是 SFF 的 11-bit 标识符，或者是 EFF 中 29-bit 标识符的前 11-bit。
RTR	远程发送请求位 (RTR) 显示当前报文是数据帧（显性）还是远程帧（隐性）。这意味着，当某个数据帧和一个远程帧有相同标识符时，数据帧始终优先于远程帧仲裁。
SRR	在 EFF 中发送替代远程请求位 (SRR)，以替代 SFF 中相同位置的 RTR 位。
IDE	标识符扩展位 (IED) 显示当前报文是 SFF （显性）还是 EFF（隐性）。这意味着，当某 SFF 帧和 EFF 帧有相同基标识符时，SFF 帧将始终优先于 EFF 帧仲裁。
Extd ID	扩展标识符 (ID.17 ~ ID.0) 是 EFF 中 29-bit 标识符的剩余 18-bit。
r1	r1（保留位 1）始终是显性位。
r0	r0（保留位 0）始终是显性位。
DLC	数据长度代码 (DLC) 为 4-bits，且应包含 0 ~ 8 中任一数值。数据帧使用 DLC 表示自身包含的数据字节数量。远程帧使用 DLC 表示从其他节点请求的数据字节数量。
数据字节	表示数据帧的数据负载量。该字节数量应与 DLC 的值匹配。首先发送数据字节 0，各数据字节优先发送最高有效位。
CRC 序列	CRC 序列是一个 15-bit 的循环冗余校验编码。
CRC 分界符	CRC 分界符是遵循 CRC 序列的单一隐性位。
确认槽	确认槽用于接收器节点，表示是否已成功接收数据帧或远程帧。发送器节点将在确认槽中发送一个隐性位，如果接收到的帧没有错误，则接收器节点应用一个显性位替代确认槽。
确认分界符	确认分界符是一个单一的隐性位。
EOF	帧结束 (EOF) 标志着数据帧或远程帧的结束，由七个隐性位组成。

15.3.2.2 错误帧和过载帧

错误帧

当某节点检测到总线错误时，将发送一个错误帧。错误帧由一个特殊的错误标志构成，该标志由某相同值的六个连续位组成，因而违反了位填充的规则。所以，当某节点检测到总线错误并发送错误帧时，其余节点也将相应地检测到一个填充错误并各自发送错误帧。也就是说，当发生总线错误时，通过上述过程可将该报文传递至总线上的所有节点。

当某节点检测到总线错误时，该节点将于下一个位发送错误帧。特例：如果总线错误类型为 CRC 错误，那么错误帧将从确认分界符的下一个位开始（可参见第 15.3.3 章）。下图 15-2 所示为一个错误帧所包含的不同域：

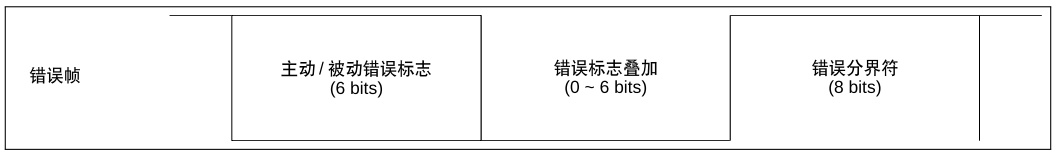


图 15-2. 错误帧中的位域

表 15-2. 错误帧

错误帧	描述
错误标志	错误标志包括两种形式: 主动错误标志和被动错误标志，主动错误标志由 6 个显性位组成，被动错误标志由 6 个隐性位组成（被其他节点的显性位优先仲裁时除外）。主动错误节点发送主动错误标志，被动错误节点发送被动错误标志。
错误标志叠加	错误标志叠加域的主要目的是允许总线上的其他节点发送各自的主动错误标志。叠加域的范围可以是 0 ~ 6 位，在检测到第一个隐性位时结束（如检测到分界符上的第一个位时）。
错误分界符	分界符域标志着错误/过载帧结束，由 8 个隐性位构成。

过载帧

过载帧与包含主动错误标志的错误帧有着相同的位域。二者主要区别在于触发发送过载帧的条件。下图 15-3 所示为过载帧中包含的位域:

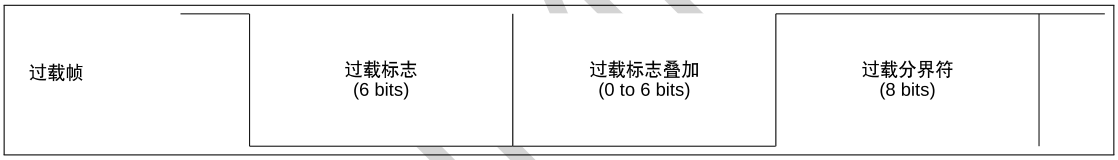


图 15-3. 过载帧中的位域

表 15-3. 过载帧

过载帧	描述
过载标志	由 6 个显性位构成。与主动错误标志相同。
过载标志叠加	允许其他节点发送过载标志的叠加，与错误标志叠加相似。
过载分界符	由 8 个隐性位构成。与错误分界符相同。

下列情况将触发发送过载帧:

1. 接收器内部要求延迟发送下一个数据帧或远程帧。
2. 在间歇域后的首个和第二个位上检测到显性位。
3. 如果在错误分界符的第八个（最后一个）位上检测到显性位。请注意，在这种情况下 TEC 和 REC 的值将不会增加（可参见第 15.3.3 章）。

由于上述情况发送过载帧时，须满足以下规定:

- 第 1 条情况下发送的过载帧只能从间歇域后的第一个位开始。
- 第 2、3 条情况下发送的过载帧须从检测到显性位的后一个位开始。
- 要延迟发送下一个数据帧或远程帧，最多可生成两个过载帧。

15.3.2.3 帧间距

帧间距充当各帧之间的分隔符。数据帧和远程帧必须与前一帧用一个帧间距分隔开，不论前面的帧是何类型（数据帧、远程帧、错误帧、过载帧）。但是，错误帧和过载帧则无需与前一个帧分隔开。

下图 15-4 所示为帧间距中包含的域：

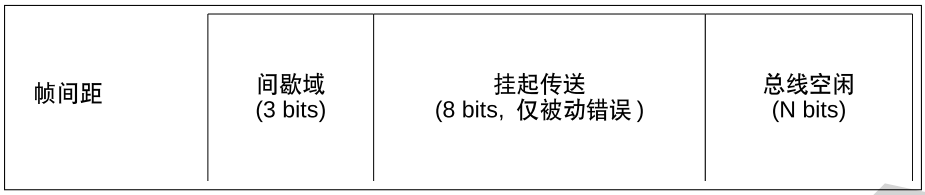


图 15-4. 帧间距中的域

表 15-4. 帧间距

帧间距	描述
间歇域	间歇域由 3 个隐性位构成。
挂起传送	被动错误节点发送报文后，节点中须包含一个挂起传送域，由 8 个隐性位构成。主动错误节点中不含这个域。
总线空闲	总线空闲域长度任意。发送 SOF 时，总线空闲结束。若节点中有挂起传送，则 SOF 应在间歇域后的第一位发送。

15.3.3 TWAI 错误

15.3.3.1 错误类型

TWAI 中的总线错误包括以下类型：

位错误

当节点发送一个位值（显性位或隐性位）但检测到相反的位时（如，发送显性位时检测到了隐性位），就会发生位错误。但是，如果发送的位是隐性位，且位于仲裁域或确认槽或被动错误标志中，那么此时检测到显性位的话也不会认定为位错误。

填充错误

当检测到相同值的 6 个连续位时（违反位填充的编码规则），发生填充错误。

CRC 错误

数据帧和远程帧的接收器将根据接收到的位计算 CRC 值。当接收器计算的值与接收到的数据帧和远程帧中的 CRC 序列不匹配时，会发生 CRC 错误。

格式错误

当某个报文中的固定格式位中包含非法位时，可检测到格式错误。比如，r1 和 r0 域必须固定为显性。

确认错误

当发送器无法在确认槽中检测到显性位时，将发生确认错误。

15.3.3.2 错误状态

TWAI 通过每个节点维护两个错误计数来实现故障界定，计数数值决定错误状态。这两个错误计数分别为：发送错误计数 (TEC) 和接收错误计数 (REC)。TWAI 包含以下错误状态。

主动错误

主动错误节点可参与到总线交互中，且在检测到错误时可以发送主动错误标志。

被动错误

被动错误节点可参与到总线交互中，但在检测到错误时只能发送一次被动错误标志。被动错误节点发送数据帧或远程帧后，须在后续的帧间距中设置挂起传送域。

离线

禁止离线节点以任意方式干扰总线（如，不允许其进行数据传输）。

15.3.3.3 错误计数

TEC 和 REC 根据以下规则递增/递减。**请注意，一条报文传输中可应用多个规则**。

1. 当接收器检测到错误时，REC 数值将增加 1。当检测到的错误为发送主动错误标志或过载标志期间的位错误除外。
2. 发送错误标志后，当接收器第一个检测到的位是显性位时，REC 数值将增加 8。
3. 当发送器发送错误标志时，TEC 数值增加 8。但是，以下情况不适用于该规则：
   - 发送器为被动错误状态，因为在应答槽未检测到显性位而产生应答错误，且在发送被动错误标志时检测到显性位时，则 TEC 数值不应增加。
   - 发送器在仲裁期间因填充错误而发送错误标志，且填充位本该是隐性位但是检测到显性位，则 TEC 数值不应增加。
4. 若发送器在发送主动错误标志和过载标志时检测到位错误，则 TEC 数值增加 8。
5. 若接收器在发送主动错误标志和过载标志时检测到位错误，则 REC 数值增加 8。
6. 任意节点在发送主动/被动错误标志或过载标志后，节点仅能承载最多 7 个连续显性位。在（发送主动错误标志或过载标志时）检测到第 14 个连续显性位，或在被动错误标志后检测到第 8 个连续显性位后，发送器将使其 TEC 数值增加 8，而接收器将使其 REC 数值增加 8。每增加 8 个连续显性位的同时，（发送器的）TEC 和（接收器的）REC 数值也将增加 8。
7. 每当发送器成功发送报文后（接收到 ACK，且直到 EOF 完成未发生错误），TEC 数值将减小 1，除非 TEC 的数值已经为 0。
8. 当接收器成功接收报文后（确认槽前未检测到错误，且成功发送 ACK），则 REC 数值将相应减小。
   - 若 REC 数值位于 1 ~ 127 之间，则其值减小 1。
   - 若 REC 数值大于 127，则其值减小到 127。
   - 若 REC 数值为 0，则仍保持为 0。
9. 当一个节点的 TEC 和/或 REC 数值大于等于 128 时，该节点变为被动错误节点。导致节点发生上述状态切换的错误，该节点仍发送主动错误标志。请注意，一旦 REC 数值到达 128，后续任何增加该值的动作都是无效的，直到 REC 数值返回到 128 以下。
10. 当某节点的 TEC 数值大于等于 256 时，该节点将变为离线节点。
11. 当某被动错误节点的 TEC 和 REC 数值都小于等于 127，则该节点将变为主动错误节点。
12. 当离线节点在总线上检测到 128 次 11 个连续隐性位后，该节点可变为主动错误节点（TEC 和 REC 数值都重设为 0）。

15.3.4 TWAI 位时序

15.3.4.1 名义位

TWAI 协议允许 TWAI 总线以特定的位速率运行。但是，总线内的所有节点必须以统一位速率运行。

- **名义位速率**为每秒发送比特数量。
- **名义位时间**为 **1/名义位速率**。

每个名义位时间中含多个段，每段由多个时间定额 (Time Quanta) 组成。**时间定额**为最小时间单位，作为一种预分频时钟信号应用于各个节点中。下图 15-5 所示为一个名义位时间内所包含的段。

TWAI 控制器将在一个时间定额的时间步长中进行操作，每个时间定额中都会分析 TWAI 的总线状态。如果两个连续的时间定额中总线状态不同（隐性-显性，或反之），意味着有边沿产生。PBS1 和 PBS2 的交点将被视为采样点，且采样的总线数值即为这个位的数值。

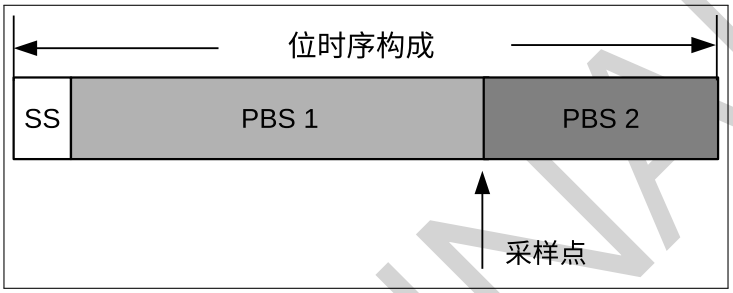


图 15-5. 位时序构成

表 15-5. 名义位时序中包含的段

段	描述
同步段 (SS)	SS（同步段）的长度为 1 个时间定额。若所有节点都同步正常，则位边沿应位于该段内。
缓冲时期段 1 (PBS1)	PBS1 的长度可为 1 ~ 16 个时间定额，用于补偿网络中的物理延迟时间。可增加 PBS1 的长度，从而更好地实现同步。
缓冲时期段 2 (PBS2)	PBS2 的长度可为 1 ~ 8 个时间定额，用于补偿节点中的信息处理时间。可缩短 PBS2 的长度，从而更好地实现同步。

15.3.4.2 硬同步与再同步

由于时钟偏移和抖动，同一总线上节点的位时序可能会脱离相位段。因而，位边沿可能会偏移到同步段的前后。针对上述位边沿偏移的问题 TWAI 提供多种同步方式。设位边沿偏移的 TQ（时间定额）数量为**相位错误 “e”**，该值与 SS 相关。

- 主动相位错误 (e > 0)：位边沿位于同步段之后采样点之前（即，边沿向后偏移）。
- 被动相位错误 (e < 0)：位边沿位于前个位的采样点之后同步段之前（即，边沿向前偏移）。

为解决相位错误，可进行两种同步方式，即**硬同步**与**再同步**。**硬同步**与**再同步**遵守以下规则：

- 单个位时序中仅可发生一次同步。
- 同步仅可发生在隐性位到显性位的边沿上。

硬同步

总线空闲期间，硬同步发生在隐性位到显性位的变化边沿上（如总线空闲后的第一个 SOF 位）。此时，所有节点都将重启其内部位时序，从而使该变化边沿位于重启位时序的同步段内。

再同步

非总线空闲期间，再同步发生在隐性位到显性位的变化边沿上。如果边沿上有主动相位错误 (e > 0)，则 PBS1 长度将增加。如果边沿上有被动相位错误 (e < 0)，则 PBS2 长度将减小。

PBS1/PBS2 具体增加和减小的时间定额取决于相位错误的绝对值，同时也受可配置的同步跳宽 (SJW) 数值限制。

- 当相位错误的绝对值小于等于 SJW 数值时，PBS1/PBS2 将增加/减小 **e** 个时间定额。该过程与硬同步具有相同效果。
- 当相位错误的绝对值大于 SJW 数值时，PBS1/PBS2 将增加/减小与 SJW 相同数值的时间定额。这意味着，在完全解决相位错误之前，可能需要多个同步位。

15.4 结构概述

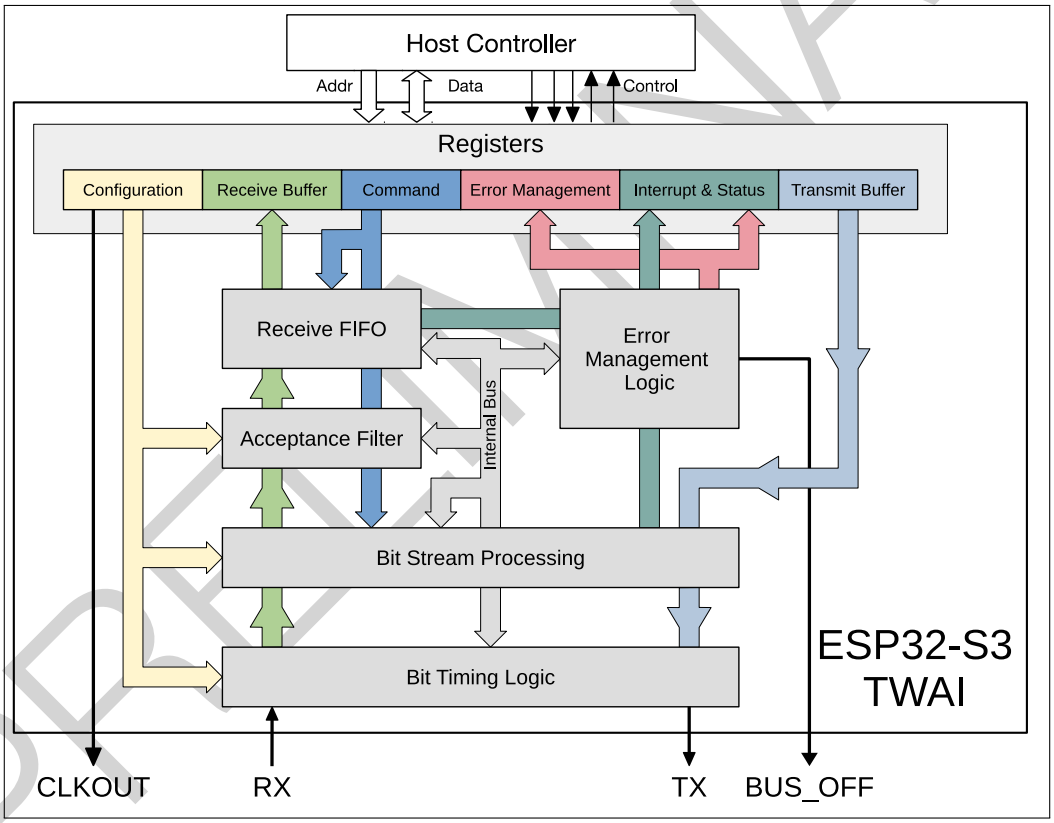


图 15-6. TWAI 概略图

TWAI 控制器的主要功能模块如图 15-6。

15.4.1 寄存器模块

ESP32-S3 的 CPU 使用 32-bit 对齐字访问外设。但是，TWAI 控制器中的大部分寄存器仅存储最低有效字节 (bits [7:0]) 上的有用数据。因此在这些寄存器中，bits [31:8] 在写入时被忽略，在读取时返回 0。

配置寄存器

配置寄存器存储 TWAI 控制器的各配置项，如位速率、操作模式、接收滤波器等。只有在 TWAI 控制器处于复位模式时，才可修改配置寄存器（可参见第 15.5.1 章）。

指令寄存器

CPU 通过指令寄存器驱动 TWAI 控制器执行任务，如发送报文或清除接收缓冲器。只有在 TWAI 控制器处于操作模式时，才可修改指令寄存器（可参见第 15.5.1 章）。

中断 & 状态寄存器

中断寄存器显示 TWAI 控制器中发生的事件（每个事件由一个单独的位表示）。状态寄存器显示 TWAI 控制器的当前状态。

错误管理寄存器

错误管理寄存器包括错误计数和捕捉寄存器。错误计数寄存器表示 TEC 和 REC 的数值。捕捉寄存器负责记录相关信息，如 TWAI 控制器在何处检测到总线错误，或何时丢失仲裁。

发送缓冲寄存器

发送缓冲器大小为 13 字节，用于存储 TWAI 的待发送报文。

接收缓冲寄存器

接收缓冲器大小为 13 字节，用于存储单个报文。接收缓冲器是进入接收 FIFO 的窗口，接收 FIFO 中的第一个报文将被映射到接收缓冲器中。

请注意，发送缓冲寄存器、接收缓冲寄存器和接收滤波寄存器的地址范围相同（地址偏移包含 0x0040 ~ 0x0070）。这些寄存器的访问权限遵循以下规则：

- 当 TWAI 控制器处于复位模式时，该地址范围被映射到接收滤波寄存器中。
- TWAI 控制器处于操作模式时：
  - 对地址范围的所有读取都映射于接收缓冲寄存器中。
  - 对地址范围的所有写入都映射于发送缓冲寄存器中。

15.4.2 位流处理器

位流处理 (BSP) 模块负责对发送缓冲器的数据进行帧处理 (如，位填充和附加 CRC 域) 并为位时序逻辑 (BTL) 模块生成位流。同时，BSP 模块还负责处理从 BTL 模块中接收的位流（如，去填充和验证 CRC），并将处理报文置于接收 FIFO。BSP 还负责检测 TWAI 总线上的错误并将此类错误报告给错误管理逻辑 (EML)。

15.4.3 错误管理逻辑

错误管理逻辑 (EML) 模块负责更新 TEC 和 REC 数值，记录错误信息（如，错误类型和错误位置），更新控制器的错误状态，确保 BSP 模块发送正确的错误标志。此外，该模块还负责记录 TWAI 控制器丢失仲裁时的 bit 位置。

15.4.4 位时序逻辑

位时序逻辑 (BTL) 模块负责以预先配置的位速率发送和接受报文。BTL 模块还负责同步位时序，确保数据传输的稳定性。位速率由多个可编程的段组成，且用户可设置每个段的 TQ（时间定额）长度，来调整传播延迟、控

制器处理时间等因素。

15.4.5 接收滤波器

接收滤波器是一个可编程的报文过滤单元，允许 TWAI 控制器根据报文的标识符域接收或拒绝该报文。通过接收滤波器的报文才能被存储到接收 FIFO 中。用户可配置接收滤波器的模式：单滤波器、双滤波器。

15.4.6 接收 FIFO

接收 FIFO 是大小为 64-byte 的缓冲器（位于 TWAI 控制器内部），负责存储通过接收滤波器的接收报文。接收 FIFO 中存储的报文大小可以不同（3 ~ 13 byte 范围之间）。当接收 FIFO 为满时（或剩余的空间不足以完全存储下一个接收报文），将触发溢出中断，后续的接收报文将丢失，直到接收 FIFO 中清除出足够的存储空间。接收 FIFO 中的第一条报文将被映射到 13-byte 的接收缓冲器中，直到该报文被清除（通过释放接收缓冲器指令）。清除后，接收缓冲器将继续映射接收 FIFO 中的下一条报文，接收 FIFO 中上一条已清除报文的空间将被释放。

15.5 功能描述

15.5.1 模式

ESP32-S3 TWAI 控制器有两种工作模式：复位模式和操作模式。将 TWAI_RESET_MODE 位置 1，进入复位模式；置 0，进入操作模式。

15.5.1.1 复位模式

要修改 TWAI 控制器的各种配置寄存器，需进入复位模式。进入复位模式时，TWAI 控制器彻底与 TWAI 总线断开连接。复位模式下，TWAI 控制器将无法发送任何报文（包括错误信号）。任何正在进行的报文传输将立即被终止。同样的，TWAI 控制器在该模式下也将无法接收任何报文。

15.5.1.2 操作模式

进入操作模式后，TWAI 控制器与总线相连，并且写保护各配置寄存器，以确保控制器的配置在运行期间保持一致。操作模式下，TWAI 控制器可以发送和接收报文（包括错误信号），但具体取决于 TWAI 控制器配置于哪种运行子模式。TWAI 控制器支持以下三种子模式：

- **正常模式：** TWAI 控制器可以发送和接收包含错误信号在内的报文（如，错误帧和过载帧）。
- **自测模式：** 与正常模式相同，但在该模式下，TWAI 控制器发送报文时，即使在 CRC 域之后没有接收到应答信号，也不会产生应答错误。通常在 TWAI 控制器自测时使用该模式。
- **只听模式：** TWAI 控制器可以接收报文，但在 TWAI 总线上保持完全被动。因此，TWAI 控制器将无法发送任何报文、应答或错误信号。错误计数将保持冻结状态。该模式用于 TWAI 总线监控。

请注意，退出复位模式后（如，进入操作模式时），TWAI 控制器需等待 11 个连续隐性位出现，才能完全连接上 TWAI 总线（即，可以发送或接收报文）。

15.5.2 位时序

TWAI 控制器的工作位速率必须在控制器处于复位模式时进行配置。在寄存器 TWAI_BUS_TIMING_0_REG 和 TWAI_BUS_TIMING_1_REG 中配置位速率，这两个寄存器包含以下域：

下表 15-6 所示为 TWAI_BUS_TIMING_0_REG 包含的位域。

表 15-6. TWAI_BUS_TIMING_0_REG 的 bit 信息 (0x18)

Bit 31-16	Bit 15	Bit 14	Bit 13	Bit 12	.....	Bit 1	Bit 0
保留	SJW.1	SJW.0	保留	BRP.12	.....	BRP.1	BRP.0

说明：

- 预分频值 (BRP)：TWAI 时间定额时钟由 APB 时钟分频得到，APB 时钟通常为 80 MHz。可通过以下公式计算分频数值，其中 t_{Tq} 为时间定额的时钟周期，t_{CLK} 为 APB 时钟周期：

  $t_{Tq} = 2 \times t_{CLK} \times (2^{12} \times BRP.12 + 2^{11} \times BRP.11 + ... + 2^{1} \times BRP.1 + 2^{0} \times BRP.0 + 1)$

- 同步跳宽 (SJW)：SJW 数值在 SJW.0 和 SJW.1 中配置，计算公式为：SJW = (2 x SJW.1 + SJW.0 + 1)。

下表 15-7 所示为 TWAI_BUS_TIMING_1_REG 包含的位域。

表 15-7. TWAI_BUS_TIMING_1_REG 的 bit (0x1c)

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	SAM	PBS2.2	PBS2.1	PBS2.0	PBS1.3	PBS1.2	PBS1.1	PBS1.0

说明：

- PBS1: 根据以下公式计算缓冲时期段 1 中的时间定额数量：(8 x PBS1.3 + 4 x PBS1.2 + 2 x PBS1.1 + PBS1.0 + 1)。
- PBS2: 根据以下公式计算缓冲时期段 2 中的时间定额数量：(4 x PBS2.2 + 2 x PBS2.1 + PBS2.0 + 1)。
- SAM: 该值置 1 启动三点采样。用于低/中速总线，有利于过滤总线上的尖峰信号。

15.5.3 中断管理

ESP32-S3 TWAI 控制器提供了八种中断，每种中断由寄存器 TWAI_INT_RAW_REG 中的一个位表示。要触发某个特定的中断，须设置 TWAI_INT ENA_REG 中相应的使能位。

TWAI 控制器提供了以下八种中断：

- 接收中断
- 发送中断
- 错误报警中断
- 数据溢出中断
- 被动错误中断
- 仲裁丢失中断
- 总线错误中断
- 总线状态中断

只要在 TWAI_INT_RAW_REG 一个或多个中断位为 1，TWAI 控制器中的中断信号即为有效，当 TWAI_INT_RAW_REG 中的所有位都被清除时，TWAI 控制器中的中断信号则失效。寄存器 TWAI_INT_RAW_REG 被读取后，其中的大

多数中断位将自动清除。但是，只有通过 TWAI_RELEASE_BUF 指令位清除所有接收报文后，接收中断位才能被清除。

15.5.3.1 接收中断 (RXI)

当 TWAI 接收 FIFO 中有待读取报文时(TWAI_RX_MESSAGE_CNT_REG > 0),都会触发 RXI。TWAI_RX_MESSAGE_CNT_REG 中记录的报文数量包括接收 FIFO 中的有效报文和溢出报文。直到通过 TWAI_RELEASE_BUF 指令位清除所有挂起接收报文后，RXI 才会失效。

15.5.3.2 发送中断 (TXI)

每当发送缓冲器空闲，将其他报文加载到发送缓冲器中等待发送时，都会触发 TXI。以下情况下，发送缓冲器将变为空闲，同时 TXI 将失效：

- 报文发送已成功完成（如，应答未发现错误）。任何发送失败将自动重发。
- 单次发送已完成（TWAI_TX_COMPLETE 位指示发送成功与否）。
- 使用 TWAI_ABORT_TX 指令位终止报文发送。

15.5.3.3 错误报警中断 (EWI)

每当寄存器 TWAI_STATUS_REG 中 TWAI_ERR_ST 和 TWAI_BUS_OFF_ST 的位值改变时（如，从 0 变为 1 或反之），都会触发 EWI。根据 EWI 触发时 TWAI_ERR_ST 和 TWAI_BUS_OFF_ST 的值分成以下几种情况：

- 如果 TWAI_ERR_ST = 0 且 TWAI_BUS_OFF_ST = 0：
  - 如果 TWAI 控制器处于主动错误状态，则表示 TEC 和 REC 的值都返回到了 TWAI_ERR_WARNING_LIMIT_REG 所设的阈值之下。
  - 如果 TWAI 控制器此前正处于总线恢复状态，则表示此时总线恢复已成功完成。
- 如果 TWAI_ERR_ST = 1 且 TWAI_BUS_OFF_ST = 0：表示 TEC 或 REC 数值已超过 TWAI_ERR_WARNING_LIMIT_REG 所设的阈值。
- 如果 TWAI_ERR_ST = 1 且 TWAI_BUS_OFF_ST = 1：表示 TWAI 控制器已进入 BUS_OFF 状态（因 TEC >= 256）。
- 如果 TWAI_ERR_ST = 0 且 TWAI_BUS_OFF_ST = 1：表示 BUS_OFF 恢复期间，TWAI 控制器的 TEC 数值已低于 TWAI_ERR_WARNING_LIMIT_REG 所设的阈值。

15.5.3.4 数据溢出中断 (DOI)

每当接收 FIFO 中有溢出发生时，都会触发 DOI。DOI 表示接收 FIFO 已满且应立即进行读取，以防出现更多溢出报文。

只有导致接收 FIFO 溢出的第一条报文可触发 DOI（如，当接收 FIFO 从未满变为开始溢出时）。任意后续的溢出报文将不会再次重复触发 DOI。只有当所有接收的（有效报文或溢出）报文都被读取后，才能再次触发 DOI。

15.5.3.5 被动错误中断 (TXI)

每当 TWAI 控制器从主动错误变为被动错误，或反之时，都会触发 EPI。

15.5.3.6 仲裁丢失中断 (ALI)

每当 TWAI 控制器尝试发送报文且丢失仲裁时，都会触发 ALI。TWAI 控制器丢失仲裁的 bit 位置将自动记录在仲裁丢失捕捉寄存器 (TWAI_ARB LOST CAP_REG) 中。仲裁丢失捕捉寄存器被清除（通过 CPU 读取该寄存器）之前，将不会再记录新发生的仲裁失败时的 bit 位置。

15.5.3.7 总线错误中断 (BEI)

每当 TWAI 控制器在 TWAI 总线上检测到错误时，都会触发 BEI。发生总线错误时，总线错误的类型和发生错误时的 bit 位置都将自动记录在错误捕捉寄存器 (TWAI_ERR_CODE_CAP_REG) 中。错误捕捉寄存器被清除（通过 CPU 的读取）之前，将不会再记录新的总线错误信息。

15.5.3.8 总线状态中断 (BSI)

每当 TWAI 控制器在收发总线数据状态与空闲状态之间切换时，都会触发 BSI。当该中断发生时，可通过读取 TWAI_STATUS_REG 寄存器 TWAI_RX_ST 和 TWAI_TX_ST 两个域来判断 TWAI 控制器当前的状态。

15.5.4 发送缓冲器与接收缓冲器

15.5.4.1 缓冲器概述

表 15-8. SFF 与 EFF 的缓冲器布局

标准格式 (SFF)		扩展格式 (EFF)	
TWAI 地址	内容	TWAI 地址	内容
0x40	TX/RX 帧信息	0x40	TX/RX 帧信息
0x44	TX/RX identifier 1	0x44	TX/RX identifier 1
0x48	TX/RX identifier 2	0x48	TX/RX identifier 2
0x4c	TX/RX data byte 1	0x4c	TX/RX identifier 3
0x50	TX/RX data byte 2	0x50	TX/RX identifier 4
0x54	TX/RX data byte 3	0x54	TX/RX data byte 1
0x58	TX/RX data byte 4	0x58	TX/RX data byte 2
0x5c	TX/RX data byte 5	0x5c	TX/RX data byte 3
0x60	TX/RX data byte 6	0x60	TX/RX data byte 4
0x64	TX/RX data byte 7	0x64	TX/RX data byte 5
0x68	TX/RX data byte 8	0x68	TX/RX data byte 6
0x6c	保留	0x6c	TX/RX data byte 7
0x70	保留	0x70	TX/RX data byte 8

表 15-8 所示为发送缓冲器和接收缓冲器的寄存器布局。发送和接收缓冲寄存器的访问地址范围相同，且只有当 TWAI 控制器处于操作模式时才可访问。CPU 的写入操作将访问发送缓冲寄存器，CPU 的读取操作将访问接收缓冲寄存器。发送缓冲器和接受缓冲器中存储报文（接收报文或待发送报文）的寄存器布局和域完全一致。

发送缓冲寄存器用于配置 TWAI 的待发送报文。CPU 会在发送缓冲寄存器进行写入操作，指定报文的帧类型、帧格式、帧 ID 和帧数据（有效载荷）。一旦发送缓冲器配置完成后，CPU 会将 TWAI_CMD_REG 中的 TWAI_TX_REQ 位置 1，以开始报文发送。

- 若是自发自收请求，变更为将 TWAI_SELF_RX_REQ 置 1。

- 若是单次发送，需要同时将 TWAI_TX_REQ 和 TWAI_ABORT_TX 置 1。

接收缓冲寄存器将映射接收 FIFO 中的第一条报文。CPU 会在接收缓冲寄存器中进行读取操作，获取第一条报文的帧类型、帧格式、帧 ID 和帧数据（有效载荷）。读取完接收缓冲寄存器中的报文后，CPU 通过将 TWAI_CMD_REG 中的 TWAI_RELEASE_BUF 位置 1 来来清除接收缓冲寄存器，若接收 FIFO 中仍有待处理的报文，按照接收报文的先后次序将最早接收到的报文映射到接收缓冲寄存器。

15.5.4.2 帧信息

帧信息的长度为 1-byte，主要用于明确报文的帧类型、帧格式以及数据长度。下表 15-9 所示为帧信息域。

表 15-9. TX/RX 帧信息 (SFF/EFF)；TWAI 地址 0x40

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	FF	RTR	X	X	DLC.3	DLC.2	DLC.1	DLC.0

说明：

1. FF: 主要明确某报文属于 EFF 还是 SFF。当 FF 位为 1 时，该报文为 EFF，当 FF 位为 0 时，该报文为 SFF。
2. RTR: 主要明确某报文是数据帧还是远程帧。当 RTR 位为 1 时，该报文为远程帧，当 RTR 位为 0 时，该报文为数据帧。
3. X: 无关 bit，可以是任意值。
4. DLC: 主要明确数据帧中的数据字节数量，或从远程帧中请求的数据字节数量。TWAI 数据帧的最大载荷为 8 个数据字节，因此 DLC 的数值范围应是 0 ~ 8。

15.5.4.3 帧标识符

若报文为 SFF，则对应的帧标识符域为 2-bytes (11-bits)；若报文为 EFF，则对应的帧标识符域为 4-bytes (29-bits)。

下表 Table 15-10-15-11 所示为 SFF (11-bits) 报文的帧标识符域。

表 15-10. TX/RX 标识符 1 (SFF); TWAI 地址 0x44

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3

表 15-11. TX/RX 标识符 2 (SFF); TWAI 地址 0x48

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.2	ID.1	ID.0	X[1]	X[2]	X[2]	X[2]	X[2]

说明：

1. 无关项。建议设置为与接收缓冲器兼容（设为 RTR），以防需使用自接收功能（或与自接收功能一起使用）。
2. 无关项。建议设置为与接收缓冲器兼容（设为 0），以防需使用自接收功能（或与自接收功能一起使用）。

下表 15-12-15-15 所示为 EFF (29-bits) 报文的帧标识符域。

表 15-12. TX/RX 标识符 1 (EFF); TWAI 地址 0x44

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.28	ID.27	ID.26	ID.25	ID.24	ID.23	ID.22	ID.21

表 15-13. TX/RX 标识符 2 (EFF); TWAI 地址 0x48

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.20	ID.19	ID.18	ID.17	ID.16	ID.15	ID.14	ID.13

表 15-14. TX/RX 标识符 3 (EFF); TWAI 地址 0x4c

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.12	ID.11	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5

表 15-15. TX/RX 标识符 4 (EFF); TWAI 地址 0x50

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.4	ID.3	ID.2	ID.1	ID.0	X[1]	X[2]	X[2]

说明：

1. 无关项。建议设置为与接收缓冲器兼容（设为 RTR），以防需使用自接收功能（或与自接收功能一起使用）。
2. 无关项。建议设置为与接收缓冲器兼容（设为 0），以防需使用自接收功能（或与自接收功能一起使用）。

15.5.4.4 帧数据

帧数据域包含发送或接收的数据帧，范围为 0 ~ 8 bytes。其中的有效字节数应与 DLC 相同。但是，如果 DLC 数值大于 8，则帧数据域的有效字节数仍为 8。远程帧中不包含数据载荷，因此不存在帧数据域。

比如，当发送 5 个字节的数据帧时，CPU 应在 DLC 域中写入数值 5，并将数据写入数据域 1 ~ 5 字节对应的寄存器。同样，当接收 DLC 为 5 的数据帧时，只有 1 ~ 5 数据字节中包含 CPU 可以读取的有效载荷数据。

15.5.5 接收 FIFO 和数据溢出

接收 FIFO 是一个 64-byte 的内部缓冲器，用于以先进先出的原则存储接收到的报文。一条接收报文可在接收 FIFO 中占 3 ~ 13 bytes 空间，且其中字节序与接收缓冲器的寄存器地址顺序相同。接收缓冲寄存器将被映射到接收 FIFO 中第一条报文。

当 TWAI 控制器接收到一条报文时，TWAI_RX_MESSAGE_COUNTER 的值将增加 1，最大值为 64。如果接收 FIFO 中有足够的剩余空间，报文内容将被写入到接收 FIFO 中。读取接收缓冲器中的消息后，通过将 TWAI_RELEASE_BUF 的位置 1，释放接收 FIFO 第一条报文所占的空间，TWAI_RX_MESSAGE_COUNTER 的值也将减小 1。然后，接收缓冲器将映射接收 FIFO 中的下一条报文。

当 TWAI 控制器接收到一条报文，但接收 FIFO 没有足够空间完整地存储这条接收报文时（不论是因为报文内容大小大于接收 FIFO 中的空闲空间，还是因为接收 FIFO 已满），便会发生数据溢出。

数据溢出发生时：

- 接收 FIFO 中剩余的空间将填满溢出报文的内容。如果接收 FIFO 已满，则无法存储溢出报文的任何内容。
- 接收 FIFO 首次发生数据溢出时，将触发数据溢出中断。
- 溢出报文仍将增加 TWAI_RX_MESSAGE_COUNTER 的值到最大值 64。
- 接收 FIFO 将在内部将溢出报文标记为无效。可使用 TWAI_MISS_ST 位，确认目前接收缓冲器映射的报文是有效报文还是溢出报文。

为了清除接收 FIFO 中的溢出报文，应重复调用 TWAI_RELEASE_BUF，直到 TWAI_RX_MESSAGE_COUNTER 为 0。这样可以读取接收 FIFO 中的所有有效报文，并清除所有溢出报文。

15.5.6 接收滤波器

接收滤波器允许 TWAI 控制器根据报文 ID 过滤接收报文（有时可以过滤报文的第一个数据字节和帧类型）。只有通过过滤的报文才能存储到接收 FIFO 中。接收滤波器的使用可以一定程度地减轻 TWAI 控制器的运行负荷（如，可减少使用接收 FIFO 和发生接收中断的次数），因为 TWAI 控制器将只需要操作一小部分过滤后的报文。

只有当 TWAI 控制器处于复位模式时，才可以访问接收滤波器的配置寄存器，因为这些配置寄存器和发送/接收缓冲寄存器的地址空间相同。

接收滤波器的配置寄存器由 32-bit 的 Code 值和 32-bit 的 Mask 值组成。Code 值将指定一种位排列模式，每条过滤报文中的位都必须匹配该模式，才能使该报文通过过滤。Mask 值可屏蔽 Code 值中的某些位（将屏蔽位设置为“不相关”的位）。如图 15-7 所示，为了使报文通过过滤，每条过滤报文的 ID 都必须匹配 Code 值所设模式或者被 Mask 值屏蔽。

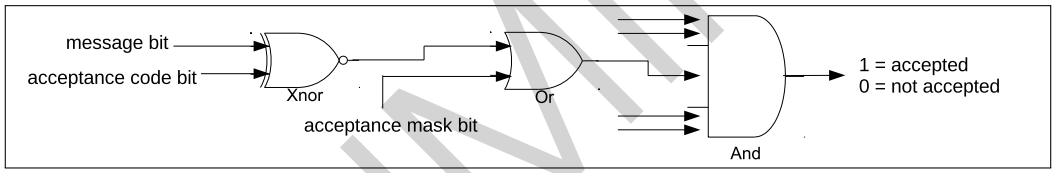


图 15-7. 接收滤波器

TWAI 控制器的接收滤波器允许 32-bit 的 Code 值和 Mask 值定义单个滤波器（单滤波模式），或两个滤波器（双滤波模式）。接收滤波器如何解析 32-bit 的 code 值和 mask 值，取决于滤波模式以及接收报文的格式（如，SFF 还是 EFF）。

15.5.6.1 单滤波模式

将 TWAI_RX_FILTER_MODE 的位置 1，可启动单滤波模式。此后，32-bit code/mask 的值将定义单个滤波器。

单个滤波器可过滤数据帧和远程帧中的以下位：

- SFF
  - 11-bit ID 整体
  - RTR bit
  - 数据字节 1 和数据字节 2
- EFF
  - 29-bit ID 整体
  - RTR bit

下图 15-8 所示为单滤波模式下如何解析 32-bit code/mask 的值。

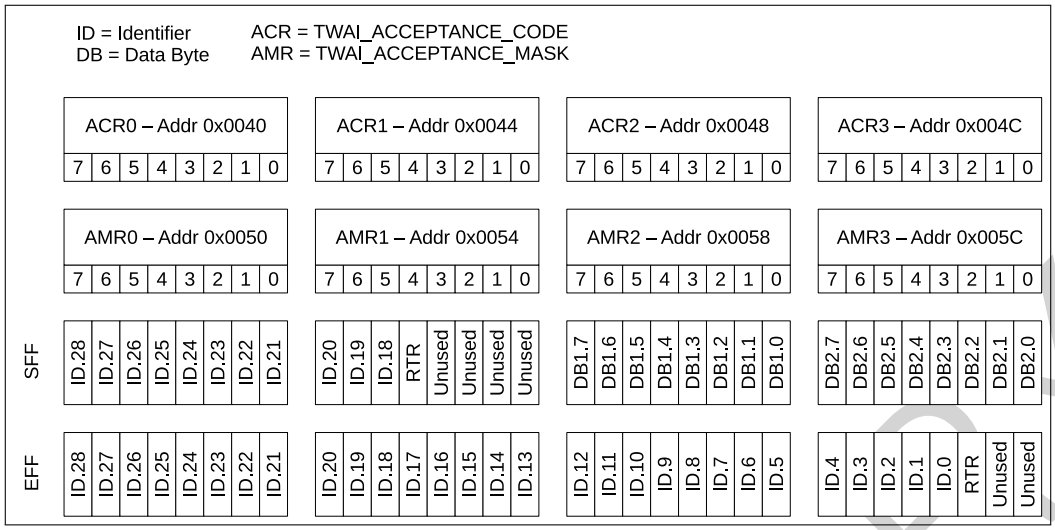


图 15-8. 单滤波模式

15.5.6.2 双滤波模式

将 TWAI_RX_FILTER_MODE 的位置 0，可启动双滤波模式。此后，32-bit code/mask 的值将定义两个滤波器之一，即滤波器 1 或滤波器 2。双滤波模式下，如果报文通过这两个滤波器中的至少一个，则表示该报文已成功通过过滤。

这两个滤波器可以过滤数据帧和远程帧中的以下位：

- SFF
  - 11-bit ID 整体
  - RTR bit
  - 数据字节 1 (仅适用于滤波器 1)
- EFF
  - 29-bit ID 的前 16-bit

下图 15-9 所示为双滤波模式下如何解析 32-bit code/mask 的值。

15.5.7 错误管理

TWAI 协议要求每个 TWAI 节点中都包含发送错误计数 (TEC) 和接收错误计数 (REC)。这两个错误计数的数值决定了 TWAI 控制器当前的错误状态（如，主动错误、被动错误、离线）。TWAI 控制器将 TEC 和 REC 的数值分别存储在 TWAI_TX_ERR_CNT_REG 和 TWAI_RX_ERR_CNT_REG 中，CPU 可随时进行读取。除了错误状态之外，TWAI 控制器还提供错误报警限制 (EWL) 的功能，这个功能可在 TWAI 控制器进入被动错误状态之前，提醒用户当前发生的严重总线错误。

TWAI 控制器的当前错误状态通过以下各数值和状态位体现，即：TEC、REC、TWAI_ERR_ST 和 TWAI_BUS_OFF_ST。这些数值和状态位的变化也将触发中断，从而提醒用户当前的错误状态变化（可参见第 15.5.3 章）。下图 15-10 所示为错误状态、上述数值和状态位以及错误状态相关中断之间的关系。

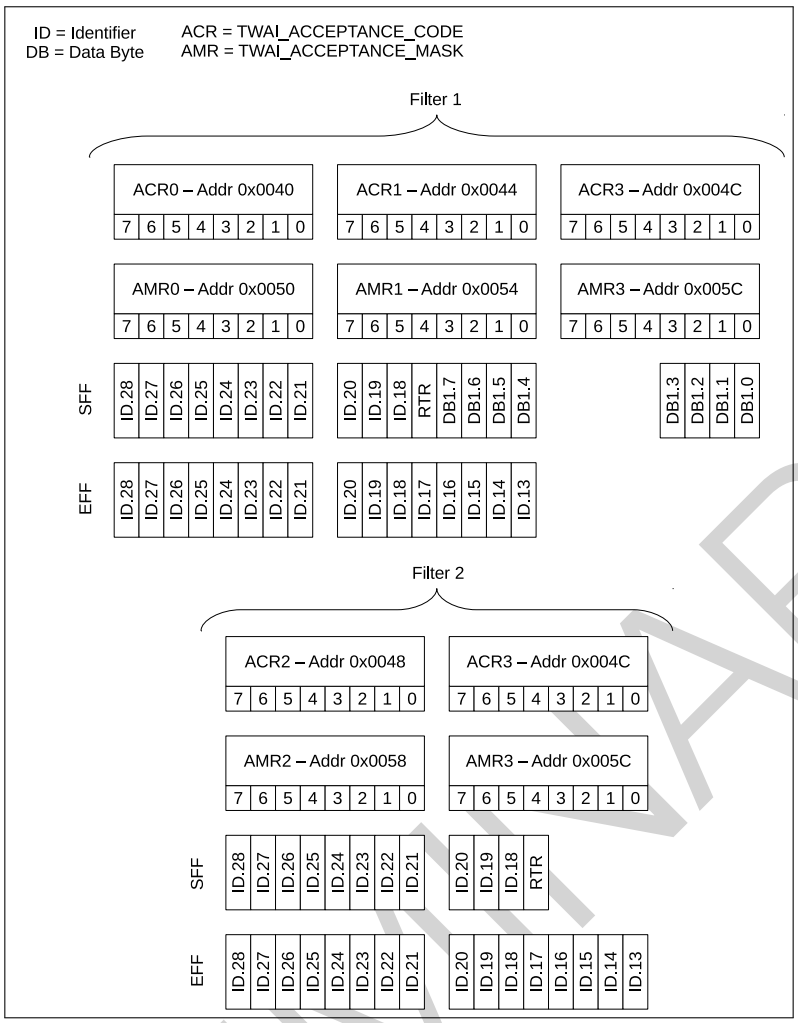


图 15-9. 双滤波模式

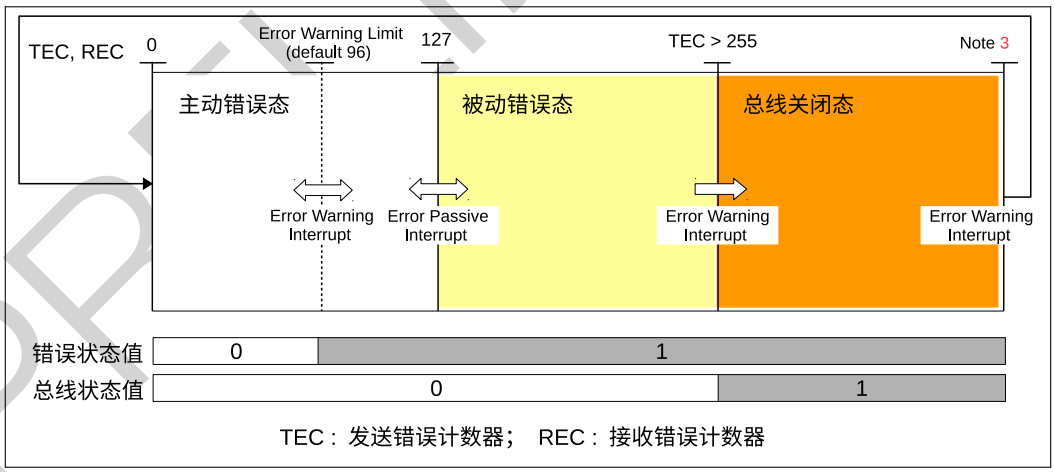


图 15-10. 错误状态变化

15.5.7.1 错误报警限制

错误报警限制 (EWL) 为 TEC 和 REC 的可配置阈值，若错误计数数值超过该阈值，将触发 EWI 中断。EWL 将作为一个报警功能提示当前发生的严重 TWAI 总线错误，且在 TWAI 控制器进入被动错误状态之前被触发。EWL 数值应在寄存器 TWAI_ERR_WARNING_LIMIT_REG 中进行配置，配置同时 TWAI 控制器必须处于复位模式下。

TWAI_ERR_WARNING_LIMIT_REG 默认数值为 96。

当 TEC 和/或 REC 数值大于等于 EWL 数值时，TWAI_ERR_ST 位将立即被置 1。同理，当 TEC 和 REC 数值都小于 EWL 数值时，TWAI_ERR_ST 位将立即复位为 0。只要 TWAI_ERR_ST（或 TWAI_BUS_OFF_ST）位值发生变化，便会触发错误报警中断。

15.5.7.2 被动错误

当 TEC 或 REC 数值大于 127 时，TWAI 控制器处于被动错误状态。同理，当 TEC 和 REC 数值都小于等于 127 时，TWAI 控制器进入主动错误状态。每当 TWAI 控制器从主动错误状态变为被动错误状态，或反之时，都将触发被动错误中断。

15.5.7.3 离线状态与离线恢复

当 TEC 数值大于 255 时，TWAI 控制器将进入离线状态。进入离线状态后，TWAI 控制器将自动进行以下动作：

- REC 数值置为 0
- TEC 数值置为 127
- TWAI_BUS_OFF_ST 位置 1
- 进入复位模式

每当 TWAI_BUS_OFF_ST 位（或 TWAI_ERR_ST 位）数值发生变化时，都将触发错误报警中断。

为了返回主动错误状态，TWAI 控制器必须进行离线恢复。要启动离线恢复，首先需要退出复位模式，进入操作模式。然后要求 TWAI 控制器在总线上检测到 128 次 11 个连续隐性位。

每一次 TWAI 控制器检测到 11 个连续隐性位时，TEC 数值都将减小，以追踪离线恢复进程。当离线恢复完成后（TEC 数值从 127 减小到 0），TWAI_BUS_OFF_ST 位将自动复位为 0，从而触发错误报警中断。

15.5.8 错误捕捉

错误捕捉 (ECC) 功能允许 TWAI 控制器以错误代码的形式记录 TWAI 总线错误的错误类型和 bit 位置。当检测到一个 TWAI 总线错误时，总线错误中断将被触发，相应的错误代码将记录在 TWAI_ERR_CODE_CAP_REG 中。寄存器 TWAI_ERR_CODE_CAP_REG 中存储的当前错误代码被读取之前，后续的总线错误中断触发时，将不会再记录错误代码。

下表 15-16 所示为寄存器 TWAI_ERR_CODE_CAP_REG 中的域：

表 15-16. TWAI_ERR_CODE_CAP_REG 中的位信息 (0x30)

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ERRC.1	ERRC.0	DIR	SEG.4	SEG.3	SEG.2	SEG.1	SEG.0

说明：

- 错误代码 (ERRC)：表示总线错误的类型。00 代表位错误，01 代表格式错误，10 代表填充错误，11 代表其他错误类型。
- 传输方向 (DIR)：表示总线错误发生时，TWAI 控制器处于发送器状态还是接收器状态。0 代表发送器，1 代表接收器。

- 错误段 (SEG)：表示总线错误发生在 TWAI 报文的哪个段。

下表 15-17 所示为 SEG.0 ~ SEG.4 的位信息。

表 15-17. SEG.4 - SEG.0 的位信息

Bit SEG.4	Bit SEG.3	Bit SEG.2	Bit SEG.1	Bit SEG.0	描述
0	0	0	1	1	帧起始
0	0	0	1	0	ID.28 ~ ID.21
0	0	1	1	0	ID.20 ~ ID.18
0	0	1	0	0	bit SRTR
0	0	1	0	1	bit IDE
0	0	1	1	1	ID.17 ~ ID.13
0	1	1	1	1	ID.12 ~ ID.5
0	1	1	1	0	ID.4 ~ ID.0
0	1	1	0	0	bit RTR
0	1	1	0	1	保留位 1
0	1	0	0	1	保留位 0
0	1	0	1	1	数据长度代码
0	1	0	1	0	数据域
0	1	0	0	0	CRC 序列
1	1	0	0	0	CRC 分界符
1	1	0	0	1	确认槽
1	1	0	1	1	确认分界符
1	1	0	1	0	帧结束
1	0	0	1	0	间歇域
1	0	0	0	1	主动错误标志
1	0	1	1	0	被动错误标志
1	0	0	1	1	兼容显性位
1	0	1	1	1	错误分界符
1	1	1	0	0	过载标志

说明：

- Bit SRTR: 标准格式 RTR bit。
- Bit IDE: 标识符扩展位。0 表示标准格式。

15.5.9 仲裁丢失捕捉

仲裁丢失捕捉 (ALC) 功能允许 TWAI 控制器记录丢失仲裁的 bit 位置。当 TWAI 控制器丢失仲裁时，bit 位置将被记录在寄存器 TWAI_ARB LOST CAP_REG 中，同时触发仲裁丢失中断。

后续的仲裁丢失中断触发时，bit 位置将不会被记录在 TWAI_ARB LOST CAP_REG 中，直到 TWAI_ERR_CODE_CAP_REG 中的当前仲裁丢失捕捉被读取。

下表 15-18 所示为 TWAI_ERR_CODE_CAP_REG 中的位域；下图 15-11 所示为一条 TWAI 报文的 bit 位置。

表 15-18. TWAI_ARB LOST CAP_REG 中的位信息 (0x2c)

Bit 31-5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	BITNO.4	BITNO.3	BITNO.2	BITNO.1	BITNO.0

说明：

- 位号 (BITNO)：表示丢失仲裁的 TWAI 报文的第 n 个位。

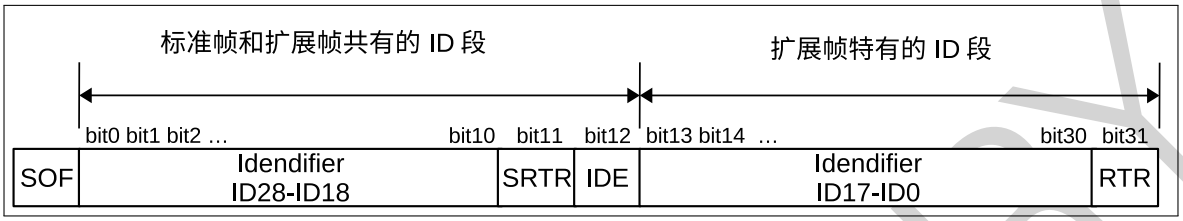


图 15-11. 丢失仲裁的 bit 位置

15.6 寄存器列表

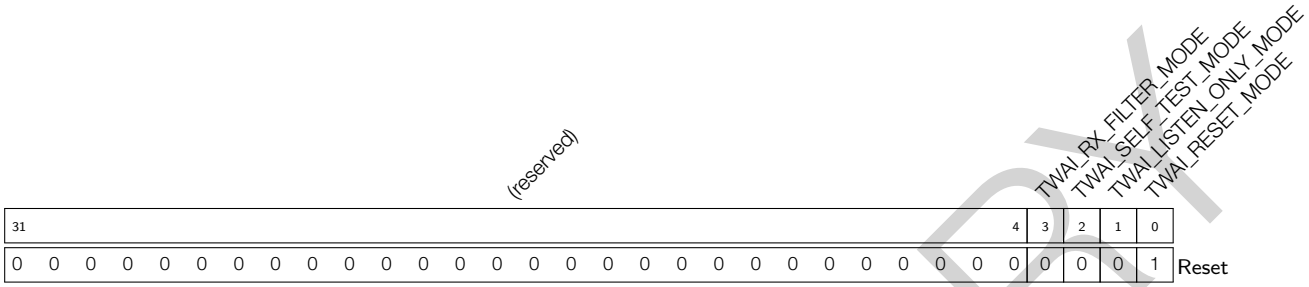
请注意，“访问权限”一栏中，“|”用于区分第 15.5.1 中描述的工作模式，其中左侧为操作模式下的访问权限，右侧标红字体为复位模式下的访问权限。本小节的所有地址均为相对于 Two-wire Automotive Interface 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

名称	描述	地址	访问权限
配置寄存器			
TWAI_MODE_REG	模式寄存器	0x0000	读/写
TWAI_BUS_TIMING_0_REG	时序配置寄存器 0	0x0018	只读 | | 读/写
TWAI_BUS_TIMING_1_REG	时序配置寄存器 1	0x001C	只读 | | 读/写
TWAI_ERR_WARNING_LIMIT_REG	错误寄存器	0x0034	只读 | | 读/写
TWAI_DATA_0_REG	数据寄存器 0	0x0040	只写 | 读/写
TWAI_DATA_1_REG	数据寄存器 1	0x0044	只写 | 读/写
TWAI_DATA_2_REG	数据寄存器 2	0x0048	只写 | 读/写
TWAI_DATA_3_REG	数据寄存器 3	0x004C	只写 | 读/写
TWAI_DATA_4_REG	数据寄存器 4	0x0050	只写 | 读/写
TWAI_DATA_5_REG	数据寄存器 5	0x0054	只写 | 读/写
TWAI_DATA_6_REG	数据寄存器 6	0x0058	只写 | 读/写
TWAI_DATA_7_REG	数据寄存器 7	0x005C	只写 | 读/写
TWAI_DATA_8_REG	数据寄存器 8	0x0060	只写 | 只读
TWAI_DATA_9_REG	数据寄存器 9	0x0064	只写 | 只读
TWAI_DATA_10_REG	数据寄存器 10	0x0068	只写 | 只读
TWAI_DATA_11_REG	数据寄存器 11	0x006C	只写 | 只读
TWAI_DATA_12_REG	数据寄存器 12	0x0070	只写 | 只读
TWAI_CLOCK_DIVIDER_REG	时钟分频寄存器	0x007C	不定
控制寄存器			
TWAI_CMD_REG	指令寄存器	0x0004	只写
状态寄存器			
TWAI_STATUS_REG	状态寄存器	0x0008	只读
TWAI_ARB LOST CAP_REG	仲裁丢失寄存器	0x002C	只读
TWAI_ERR_CODE_CAP_REG	错误捕获寄存器	0x0030	只读
TWAI_RX_ERR_CNT_REG	接收错误寄存器	0x0038	只读 | 读/写
TWAI_TX_ERR_CNT_REG	发送错误寄存器	0x003C	只读 | 读/写
TWAI_RX_MESSAGE_CNT_REG	接收数据寄存器	0x0074	只读
中断寄存器			
TWAI_INT_RAW_REG	中断寄存器	0x000C	只读
TWAI_INT ENA_REG	中断使能寄存器	0x0010	读/写

15.7 寄存器

请注意“访问权限”一栏中，“I”左侧为操作模式下的访问权限，右侧标红字体为复位模式下的访问权限。本小节的所有地址均为相对于 Two-wire Automotive Interface 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

Register 15.1. TWAI_MODE_REG (0x0000)



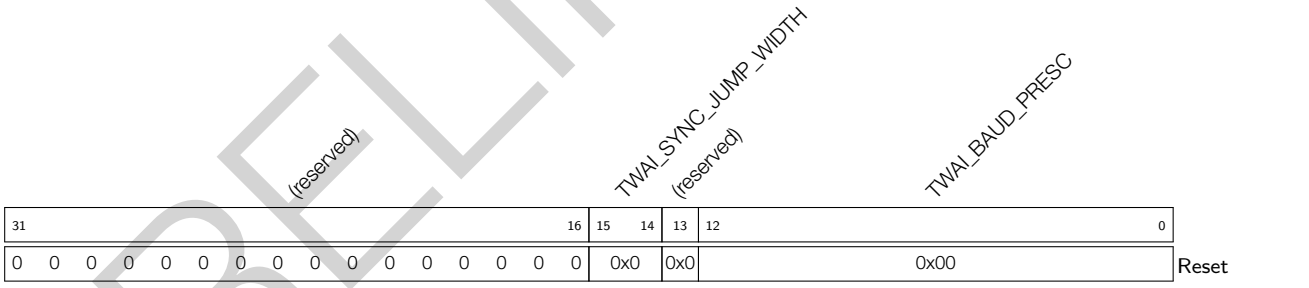
TWAI_RESET_MODE 配置 TWAI 控制器操作模式。1：复位模式；0：操作模式（读/写）

TWAI_LISTEN_ONLY_MODE 置 1 进入只听模式，处于该模式下的节点只接收总线上数据，不产生应答信号，也不更新接收错误计数。（读/写）

TWAI_SELF_TEST_MODE 置 1 启动自测模式，此模式下发送节点发送完数据后无需应答信号反馈。该模式常配合自接自收指令测试某个节点。（读/写）

TWAI_RX_FILTER_MODE 配置滤波模式。0: 双滤波模式；1: 单滤波模式（读/写）

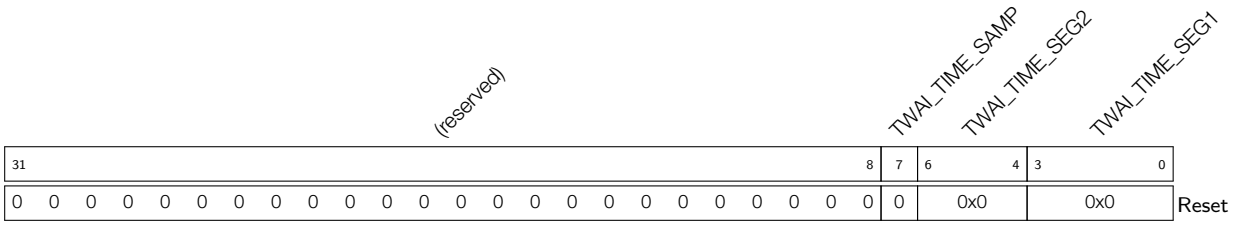
Register 15.2. TWAI_BUS_TIMING_0_REG (0x0018)



TWAI_BAUD_PRESC 预分频值，决定分频比例。（只读 | 读/写）

TWAI_SYNC_JUMP_WIDTH 同步跳宽 (SJW)，范围为 1 ~ 4 个时间定额。（只读 | 读/写）

Register 15.3. TWAI_BUS_TIMING_1_REG (0x001C)

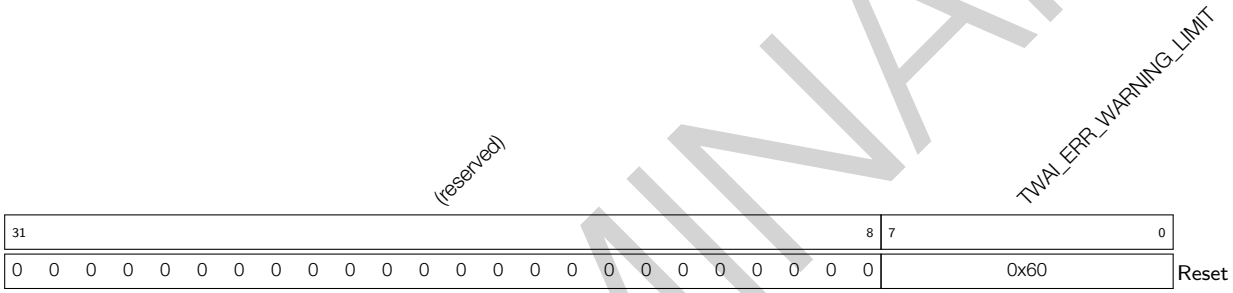


TWAI_TIME_SEG1 缓冲时期段 1 的宽度。(只读 | 读/写)

TWAI_TIME_SEG2 缓冲时期段 2 的宽度。(只读 | 读/写)

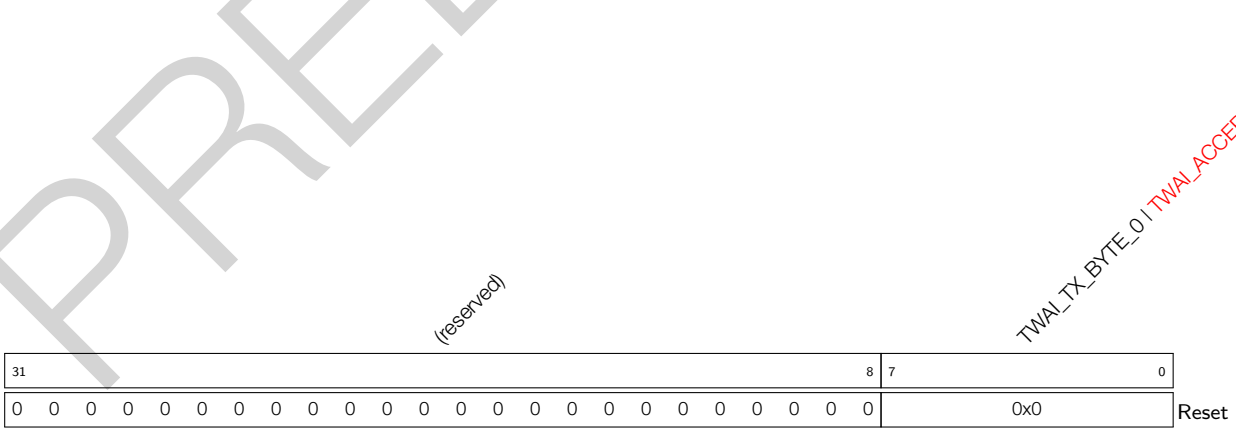
TWAI_TIME_SAMP 采样点数目。0：采样 1 次；1：采样三次（只读 | 读/写）

Register 15.4. TWAI_ERR_WARNING_LIMIT_REG (0x0034)



TWAI_ERR_WARNING_LIMIT 错误报警阈值，当任一错误计数数值超过该阈值或者所有错误计数数值都小于该阈值时，将触发错误报警中断（使能信号有效情况下）。(只读 | 读/写)

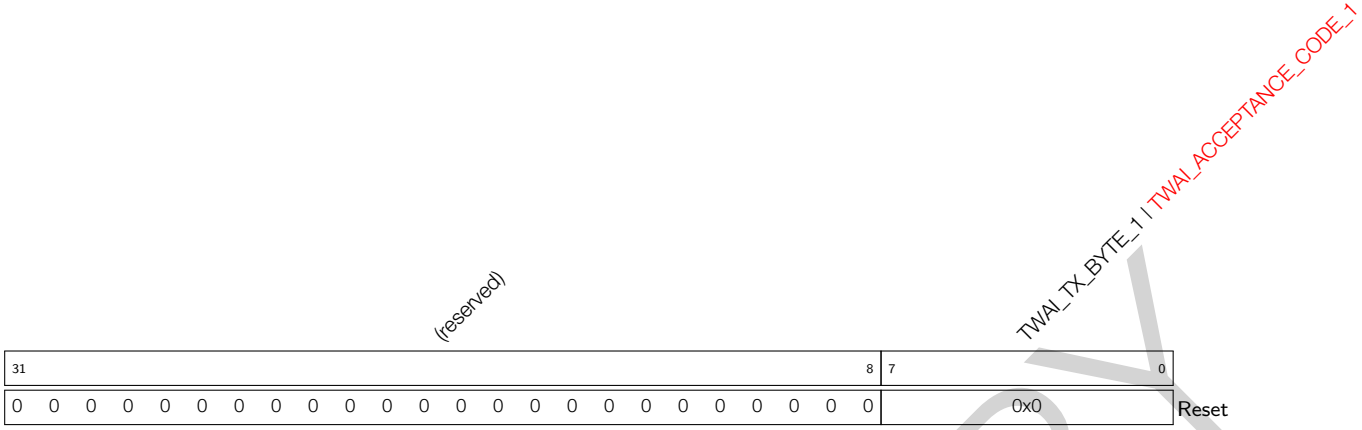
Register 15.5. TWAI_DATA_0_REG (0x0040)



TWAI_TX_BYTE_0 操作模式下，存储着待发送数据的第 0 个字节内容。(只写)

TWAI_ACCEPTANCE_CODE_0 复位模式下，存储着滤波编码的第 0 个字节。(读/写)

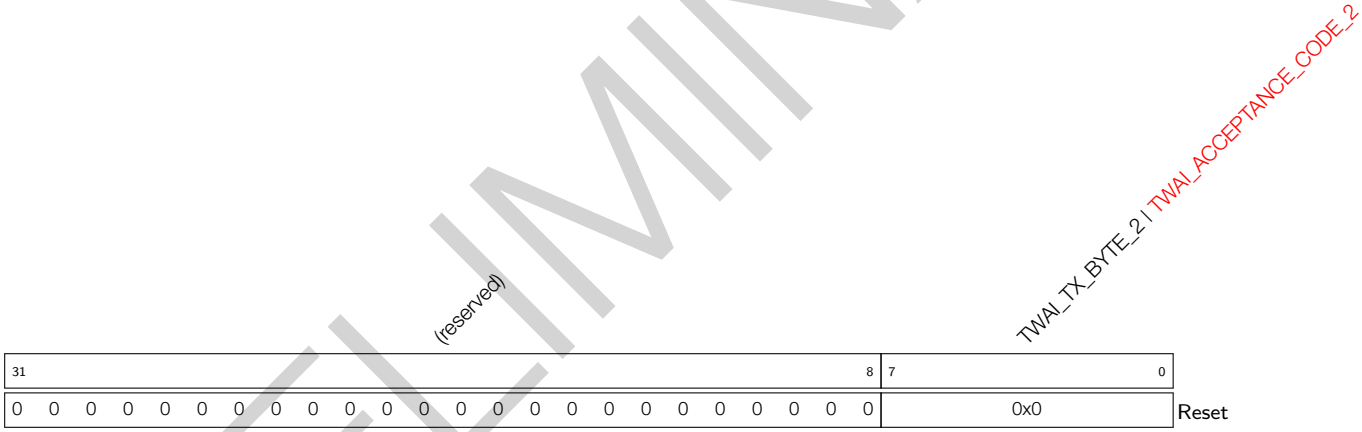
Register 15.6. TWAI_DATA_1_REG (0x0044)



TWAI_TX_BYTE_1 操作模式下，存储着待发送数据的第 1 个字节内容。(只写)

TWAI_ACCEPTANCE_CODE_1 复位模式下，存储着滤波编码的第 1 个字节。(读/写)

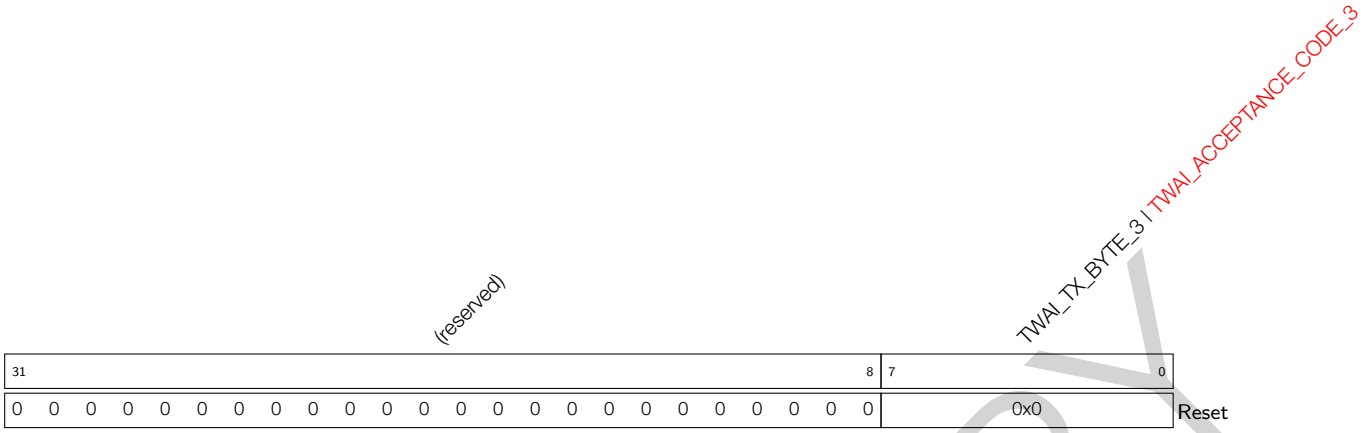
Register 15.7. TWAI_DATA_2_REG (0x0048)



TWAI_TX_BYTE_2 操作模式下，存储着待发送数据的第 2 个字节内容。(只写)

TWAI_ACCEPTANCE_CODE_2 复位模式下，存储着滤波编码的第 2 个字节。(读/写)

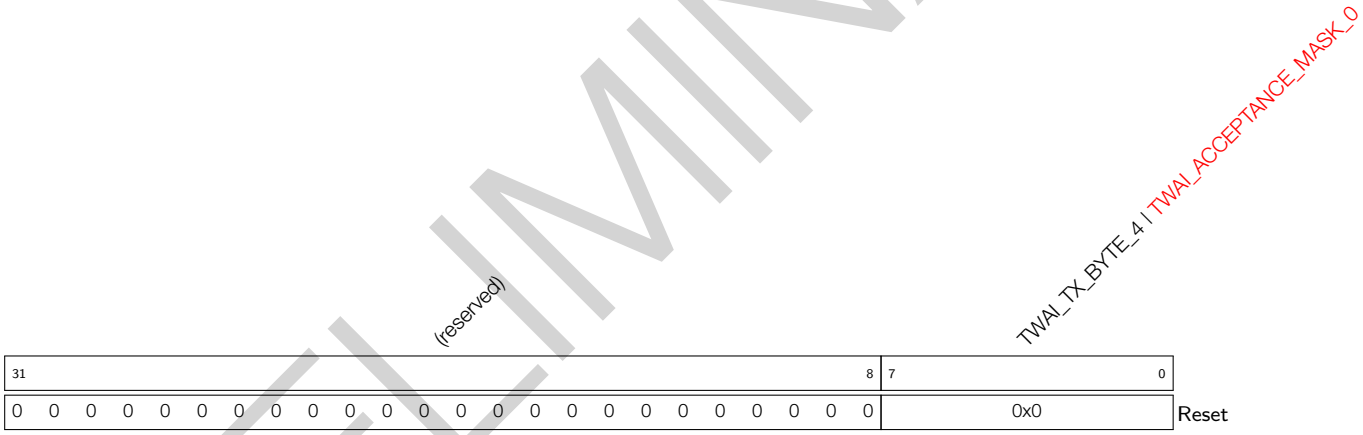
Register 15.8. TWAI_DATA_3_REG (0x004C)



TWAI_TX_BYTE_3 操作模式下，存储着待发送数据的第 3 个字节内容。(只写)

TWAI_ACCEPTANCE_CODE_3 复位模式下，存储着滤波编码的第 3 个字节。(读/写)

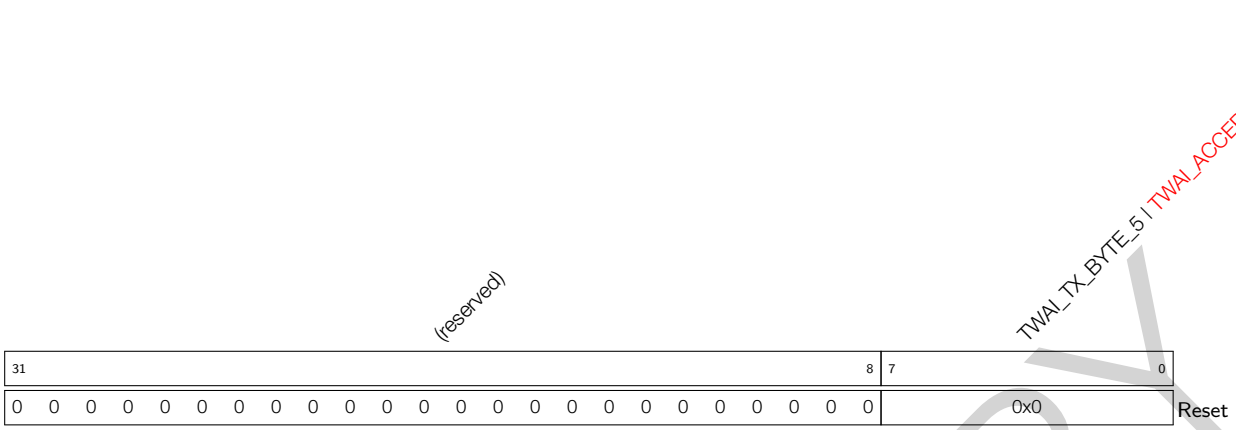
Register 15.9. TWAI_DATA_4_REG (0x0050)



TWAI_TX_BYTE_4 操作模式下，存储着待发送数据的第 4 个字节内容。(只写)

TWAI_ACCEPTANCE_MASK_0 复位模式下，存储着滤波编码的第 0 个字节。(读/写)

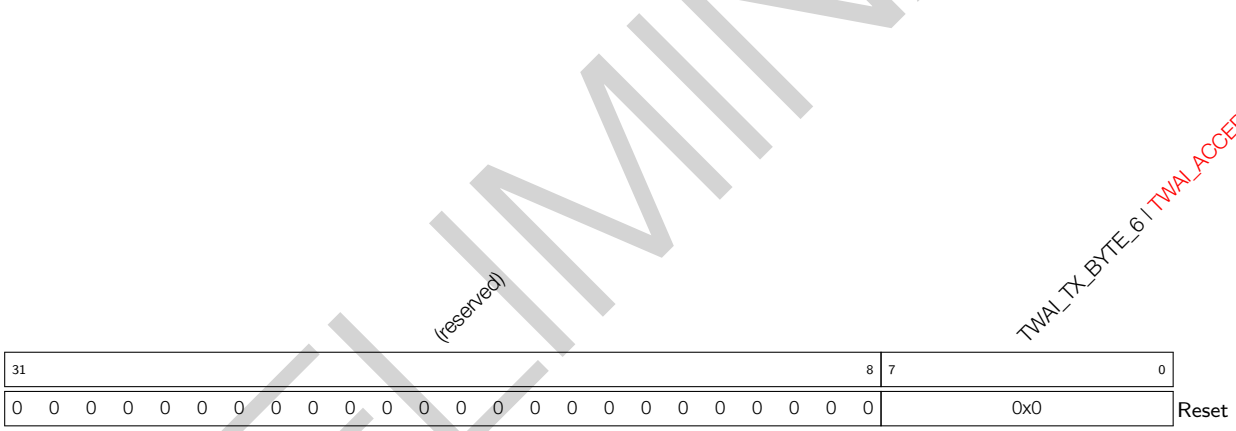
Register 15.10. TWAI_DATA_5_REG (0x0054)



TWAI_TX_BYTE_5 操作模式下，存储着待发送数据的第 5 个字节内容。(只写)

TWAI_ACCEPTANCE_MASK_1 复位模式下，存储着滤波编码的第 1 个字节。(读/写)

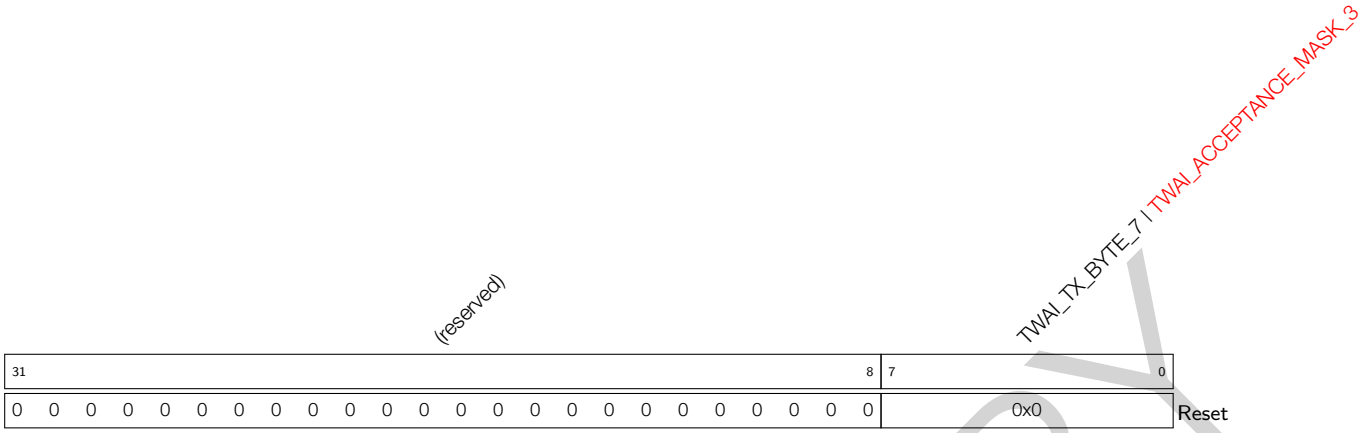
Register 15.11. TWAI_DATA_6_REG (0x0058)



TWAI_TX_BYTE_6 操作模式下，存储着待发送数据的第 6 个字节内容。(只写)

TWAI_ACCEPTANCE_MASK_2 复位模式下，存储着滤波编码的第 2 个字节。(读/写)

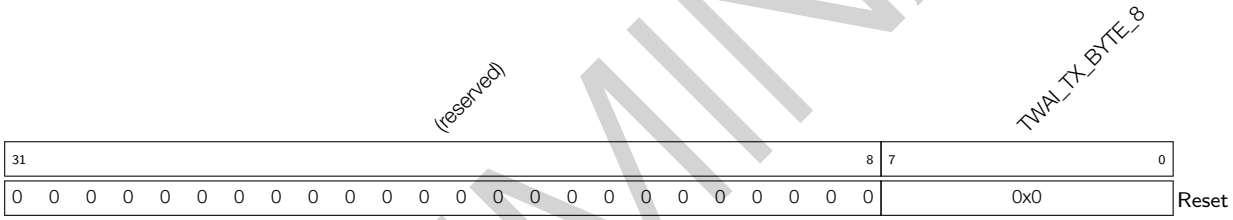
Register 15.12. TWAI_DATA_7_REG (0x005C)



TWAI_TX_BYTE_7 操作模式下，存储着待发送数据的第 7 个字节内容。(只写)

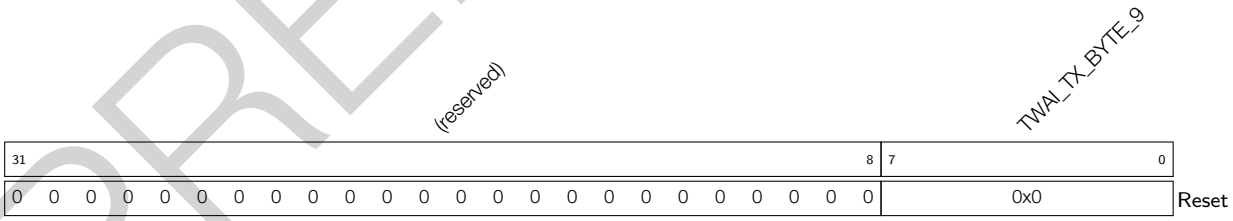
TWAI_ACCEPTANCE_MASK_3 复位模式下，存储着滤波编码的第 3 个字节。(读/写)

Register 15.13. TWAI_DATA_8_REG (0x0060)



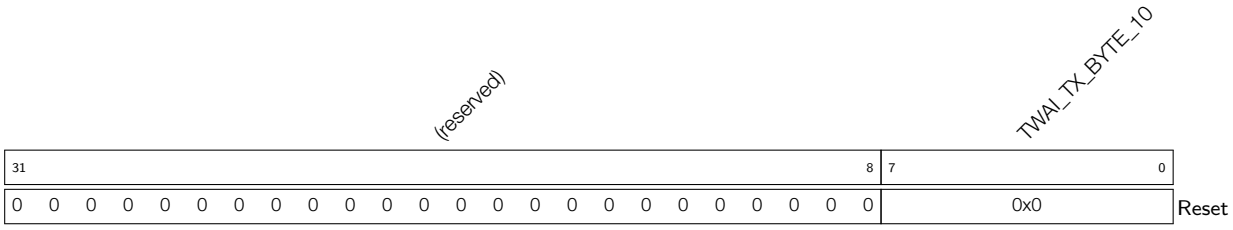
TWAI_TX_BYTE_8 操作模式下，存储着待发送数据的第 8 个字节内容。(只写)

Register 15.14. TWAI_DATA_9_REG (0x0064)



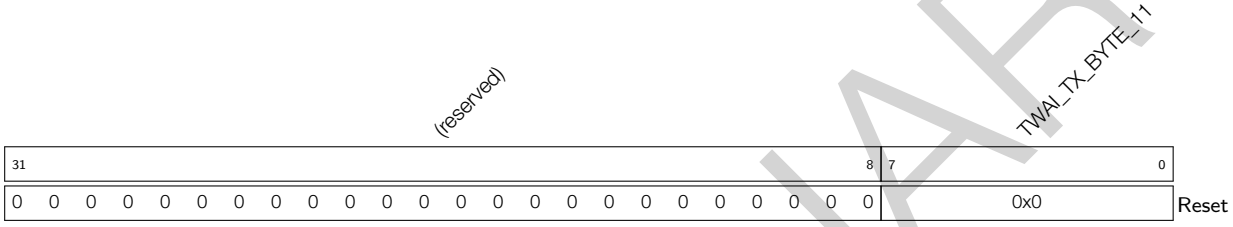
TWAI_TX_BYTE_9 操作模式下，存储着待发送数据的第 9 个字节内容。(只写)

Register 15.15. TWAI_DATA_10_REG (0x0068)



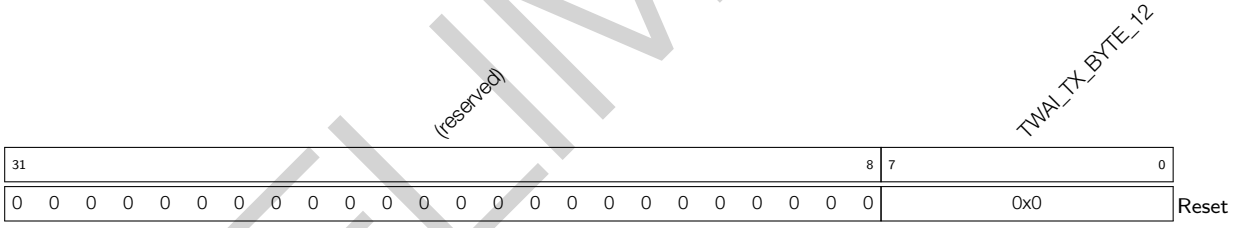
TWAI_TX_BYTE_10 操作模式下，存储着待发送数据的第 10 个字节内容。（只写）

Register 15.16. TWAI_DATA_11_REG (0x006C)



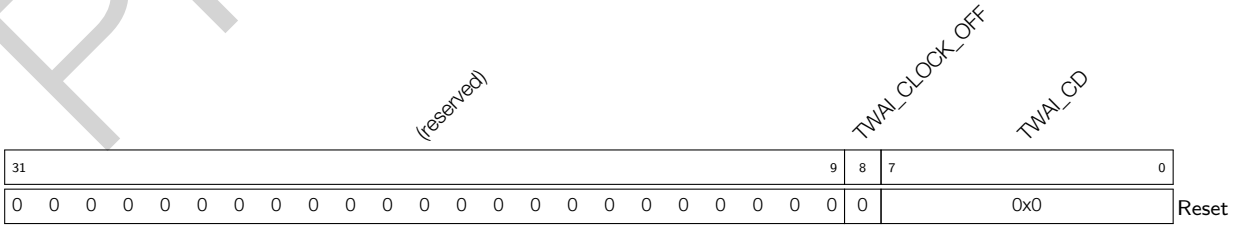
TWAI_TX_BYTE_11 操作模式下，存储着待发送数据的第 11 个字节内容。（只写）

Register 15.17. TWAI_DATA_12_REG (0x0070)



TWAI_TX_BYTE_12 操作模式下，存储着待发送数据的第 12 个字节内容。（只写）

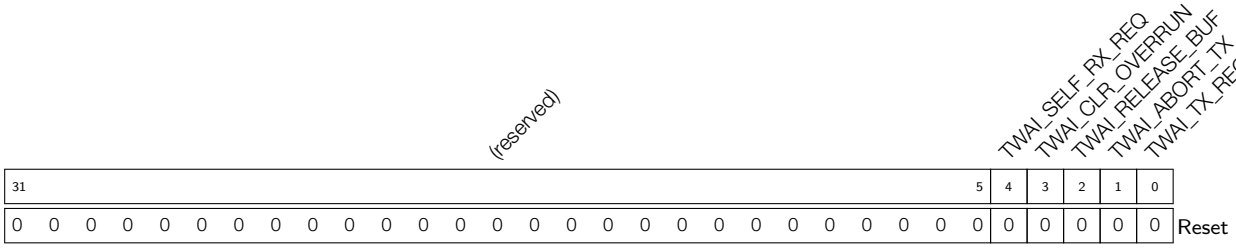
Register 15.18. TWAI_CLOCK_DIVIDER_REG (0x007C)



TWAI_CD 配置输出时钟 CLKOUT 的分频系数。（读/写）

TWAI_CLOCK_OFF 复位模式下可配。1：关闭输出的 CLKOUT 时钟；0：打开 CLKOUT 时钟（只读 | 读/写）

Register 15.19. TWAI_CMD_REG (0x0004)



TWAI_TX_REQ 置 1 驱动节点开始发送数据任务。(只写)

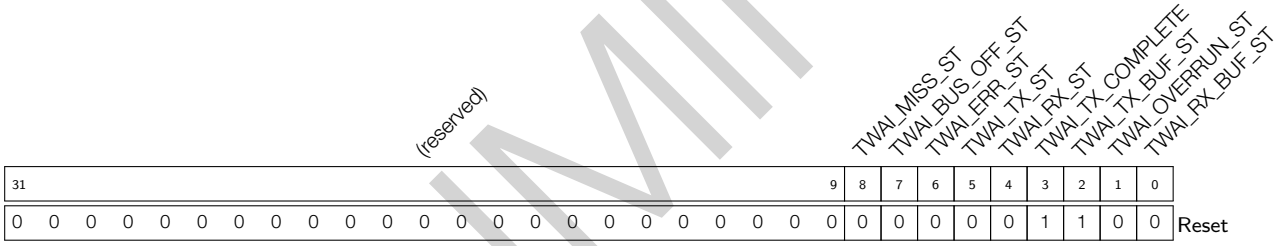
TWAI_ABORT_TX 置 1 取消当前还未开始的发送任务。(只写)

TWAI_RELEASE_BUF 置 1 释放接收缓冲器。(只写)

TWAI_CLR_OVERRUN 置 1 清除数据溢出状态。(只写)

TWAI_SELF_RX_REQ 自接自收命令。置 1 允许发送节点发送数据的同时接收总线上的数据。(只写)

Register 15.20. TWAI_STATUS_REG (0x0008)



TWAI_RX_BUF_ST 若值为 1，表明接收缓冲器中数据不为空，至少有一个已经接收到的数据包。(只读)

TWAI_OVERRUN_ST 若值为 1，表明接收 FIFO 中存储的数据已满，产生了溢出。(只读)

TWAI_TX_BUF_ST 若值为 1，表明发送缓冲器为空，允许写入待发送数据。(只读)

TWAI_TX_COMPLETE 若值为 1，表明成功从总线上接收到一个数据包。(只读)

TWAI_RX_ST 若值为 1，表明节点正在从总线上接收数据。(只读)

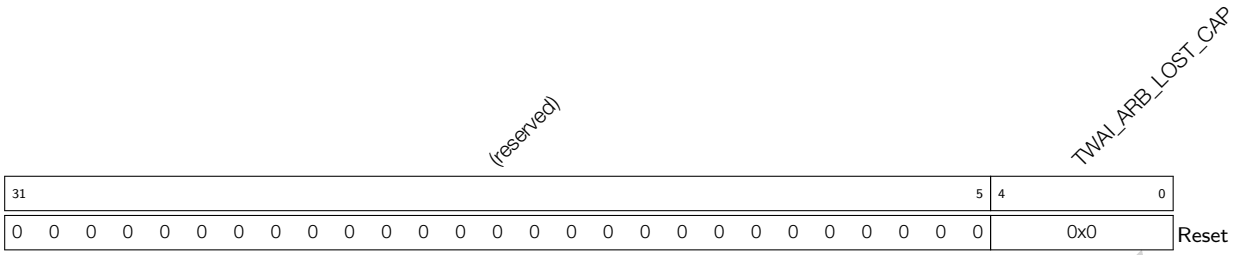
TWAI_TX_ST 若值为 1，表明节点正在往总线上发送数据。(只读)

TWAI_ERR_ST 若值为 1，表明接收错误计数和发送错误计数中至少有一个数值大于等于寄存器 TWAI_ERR_WARNING_LIMIT_REG 中配置的数值 (只读)

TWAI_BUS_OFF_ST 若值为 1，表明节点处于离线状态，不再响应总线上的数据传输。(只读)

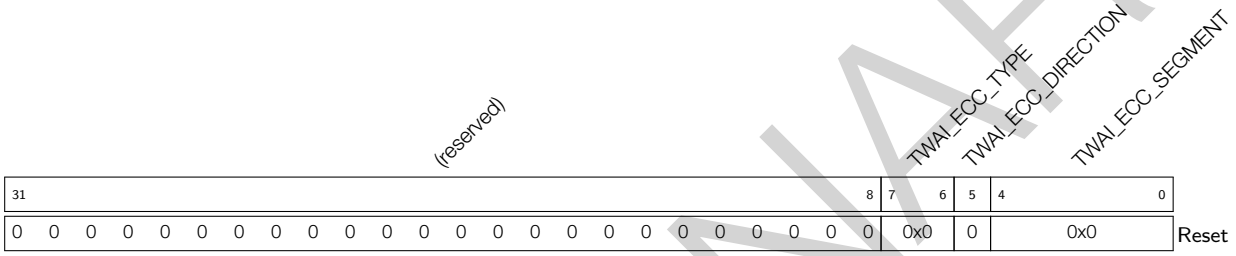
TWAI_MISS_ST 反映了从接收 FIFO 中取出数据包的完整状态。1: 当前数据包是缺失的；0：当前数据包是完整的 (只读)

Register 15.21. TWAI_ARB LOST CAP_REG (0x002C)



TWAI_ARB_LOST_CAP 记录着发送节点仲裁丢失的 bit 位置。（只读）

Register 15.22. TWAI_ERR_CODE_CAP_REG (0x0030)

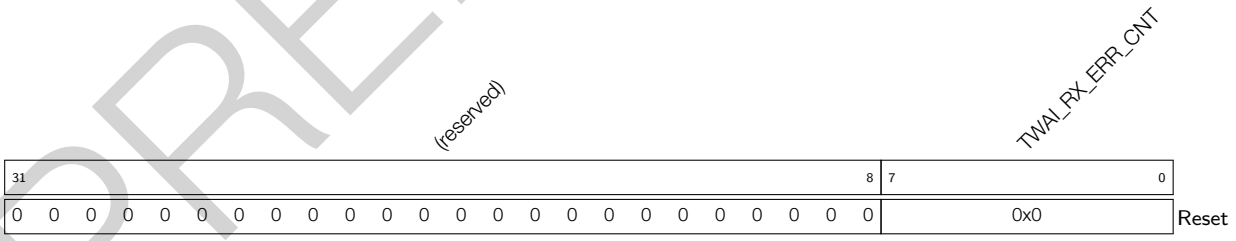


TWAI_ECC_SEGMENT 记录错误发生的位置，详见表 15-16。（只读）

TWAI_ECC_DIRECTION 记录错误时节点的数据传输方向。1：接收数据时发生错误；0：发送数据时发生错误（只读）

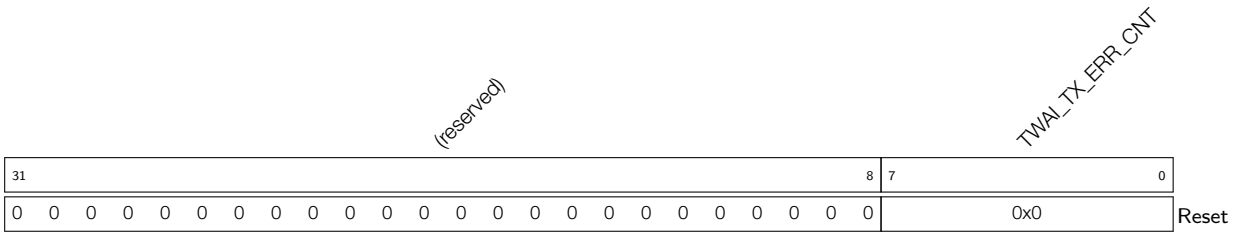
TWAI_ECC_TYPE 记录错误类别：00：位错误；01：格式错误；10：填充错误；11：其他错误（只读）

Register 15.23. TWAI_RX_ERR_CNT_REG (0x0038)



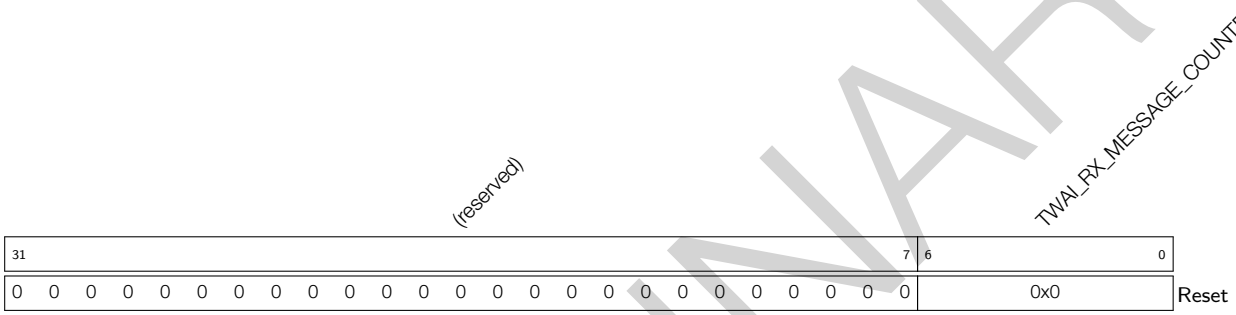
TWAI_RX_ERR_CNT 接收错误计数，数值变化发生在接收状态下。（只读 | 读/写）

Register 15.24. TWAI_TX_ERR_CNT_REG (0x003C)



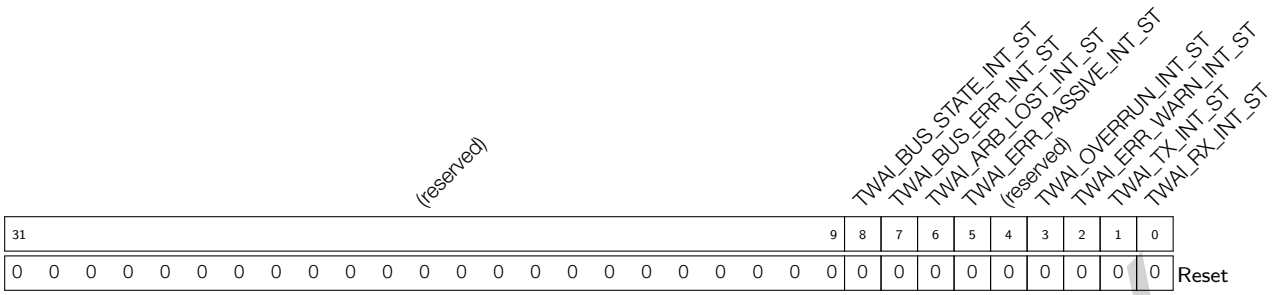
TWAI_TX_ERR_CNT 发送错误计数，数值变化发生在发送状态下。(只读 | 读/写)

Register 15.25. TWAI_RX_MESSAGE_CNT_REG (0x0074)



TWAI_RX_MESSAGE_COUNTER 存储着接收 FIFO 中数据包的个数。(只读)

Register 15.26. TWAI_INT_RAW_REG (0x000C)



TWAI_RX_INT_ST 接收中断。若值为 1，表明接收 FIFO 不为空，有接收数据待处理。(只读)

TWAI_TX_INT_ST 发送中断。若值为 1，表明节点数据发送任务结束，可以执行新的数据发送任务。(只读)

TWAI_ERR_WARN_INT_ST 错误报警中断。若值为 1，表明状态寄存器中错误状态信号和离线信号发生变化（0 变为 1 或 1 变为 0)。(只读)

TWAI_OVERRUN_INT_ST 数据溢出中断。若值为 1，表明节点的接收 FIFO 数据溢出。(只读)

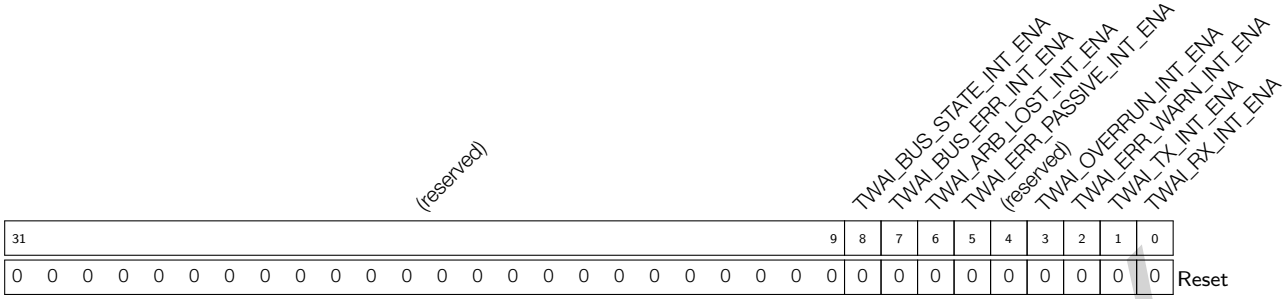
TWAI_ERR_PASSIVE_INT_ST 被动错误中断。若值为 1，表明节点由于错误计数数值的变化，在主动错误状态与被动错误状态间发生了切换。(只读)

TWAI_ARB_LOST_INT_ST 仲裁丢失中断。若值为 1，表明发送节点丢失仲裁。(只读)

TWAI_BUS_ERR_INT_ST 错误中断。若值为 1，表明节点检测到总线上发生了错误。(只读)

TWAI_BUS_STATE_INT_ST 总线状态中断。若值为 1，表明控制器状态发生了变化。(只读)

Register 15.27. TWAI_INT ENA_REG (0x0010)



TWAI_RX_INT_ENA 置 1 使能接收中断。(读/写)

TWAI_TX_INT_ENA 置 1 使能发送中断。(读/写)

TWAI_ERR_WARN_INT_ENA 置 1 使能错误报警中断。(读/写)

TWAI_OVERRUN_INT_ENA 置 1 使能数据溢出中断。(读/写)

TWAI_ERR_PASSIVE_INT_ENA 置 1 使能被动错误中断。(读/写)

TWAI_ARB_LOST_INT_ENA 置 1 使能仲裁丢失中断。(读/写)

TWAI_BUS_ERR_INT_ENA 置 1 使能总线错误中断。(读/写)

TWAI_BUS_STATE_INT_ENA 置 1 使能总线状态中断。(读/写)

16 USB OTG (USB)

16.1 概述

ESP32-S3 带有一个集成了收发器的 USB On-The-Go（下文将称为 OTG_FS）外设。该 OTG_FS 外设可配置成主机模式 (Host mode) 或设备模式 (Device mode)，完全符合 USB1.1 协议规范。它支持传输速率为 12 Mbit/s 的全速模式 (Full-Speed, FS) 和传输速率为 1.5 Mbit/s 的低速模式 (Low-Speed, LS)，还支持主机协商协议 (Host Negotiation Protocol, HNP) 和会话请求协议 (Session Request Protocol, SRP)。

16.2 特性

16.2.1 通用特性

- 支持全速和低速速率
- 主机协商协议 (HNP) 和会话请求协议 (SRP)，均可作为 A 或 B 设备
- 动态 FIFO (DFIFO) 大小
- 支持多种存储器访问模式
  - Scatter/Gather DMA 模式
  - 缓冲 (Buffer) DMA 模式
  - Slave 模式
- 可选择集成收发器或外部收发器
- 当仅使用集成收发器时，可通过时分复用技术，和 USB 串行/JTAG 控制器共用集成收发器
- 当集成收发器和外部收发器同时投入使用时，支持 USB OTG 和 USB 串行/JTAG 两外设各自挑选不同的收发器使用
- 可作为 Light-sleep 唤醒源

16.2.2 设备模式 (Device mode) 特性

- 端点 0 永远存在（双向控制，由 EP0 IN 和 EP0 OUT 组成）
- 6 个附加端点 (1 ~ 6)，可配置为 IN 或 OUT
- 最多 5 个 IN 端点同时工作（包括 EP0 IN）
- 所有 OUT 端点共享一个 RX FIFO
- 每个 IN 端点都有专用的 TX FIFO

16.2.3 主机模式 (Host mode) 特性

- 8 个通道（管道）
  - 由 IN 与 OUT 两个通道组成的一个控制管道，因为 IN 和 OUT 必须分开处理。仅支持控制传输类型。
  - 其余 7 个管道可被配置为 IN 或 OUT，支持批量、同步、中断中的任意传输类型。
- 所有通道共用一个 RX FIFO、一个非周期性 TX FIFO、和一个周期性 TX FIFO。每个 FIFO 大小可配置。

16.3 功能描述

16.3.1 控制器内核与接口

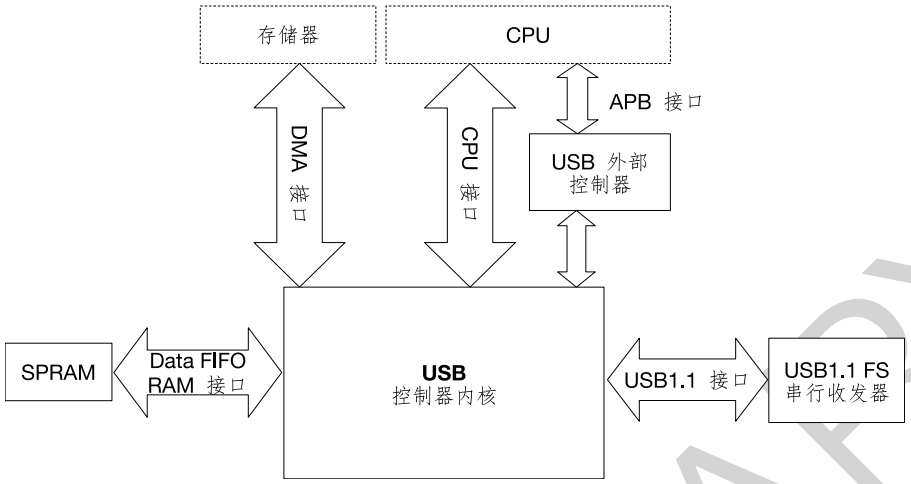


图 16-1. OTG_FS 系统架构

OTG_FS 外设的核心称为 USB 控制器内核。如图 16-1 所示，控制器内核有以下 4 个接口：

- **CPU 接口**
  CPU 可以通过该接口读写控制器内核的多个寄存器和 FIFO。该接口在内部实现为 AHB 从机接口。通过该接口访问 FIFO 的方式称为 Slave 模式。
- **APB 接口**
  CPU 可以通过 USB 外部控制器 (USB external controller) 来控制 USB 控制器内核的接口。
- **DMA 接口**
  控制器内核的内部 DMA 可以通过该接口读写系统存储器（例如在 DMA 模式下获取和写入有效数据）。该接口在内部实现为 AHB 主机接口。
- **USB1.1 接口**
  控制器内核通过该接口连接 USB1.1 全速串行收发器。除 USB OTG 之外，ESP32-S3 还内置一个 USB 串行/JTAG 控制器（请参阅章节 19 *USB Serial/JTAG 控制器 (USB_SERIAL_JTAG) [to be added later]*）。这两个 USB 控制器可通过时分复用使用内部集成收发器，或者一个控制器连接内部收发器，另一个控制器连接外部收发器。

  当只使用内部收发器时，USB OTG 和 USB 串行/JTAG 外设共用这个收发器。默认情况下，内部收发器与 USB 串行/JTAG 外设相连。当 RTC_CNTL_SW _HW_USB_PHY_SEL_CFG 为 0 时，eFuse 中的 EFUSE_USB_PHY_SEL 位决定内部收发器与哪个外设相连。若该位为 0，内部收发器与 USB 串行/JTAG 外设相连；若该位为 1，内部收发器与 USB OTG 外设相连。当 RTC_CNTL_SW _HW_USB_PHY_SEL_CFG 为 1 时，由 RTC_CNTL_SW _USB_PHY_SEL_CFG 控制内部收发器与哪个外设相连（与 EFUSE_USB_PHY_SEL 位的使用方式相同）。

  当内部和外部收发器都被使用时，一个 USB 控制器选择其中一个收发器使用，另一个 USB 控制器则使用剩余的收发器。具体的地址映射信息，请参阅章节 19 *USB Serial/JTAG 控制器 (USB_SERIAL_JTAG) [to be added later]*。
- **USB 外部控制器**
  USB 外部控制器主要用于将 USB 1.1 全速串行接口连接到内部或外部收发器。USB 外部控制器还能实现

省电模式，具体做法是控制器内核时钟（AHB 时钟）采用门控时钟，或关闭所连接的 SPRAM 时钟。需要注意的是此节能模式与通过 SRP 实现的节能方式有所不同。

- **数据 FIFO RAM 接口**
  控制器内核使用的多个 FIFO 实际上并不位于控制器内核内部，而是位于 SPRAM（单端口 RAM）上。FIFO 的大小可动态配置，因此在运行时在 SPRAM 中进行分配。CPU、DMA 或控制器通过该接口读写 FIFO。

16.3.2 存储器布局

图 16-2 显示了用于配置和控制 USB 控制器内核的寄存器布局。请注意，USB 外部控制器使用另外一组寄存器（称为 wrap 寄存器）。

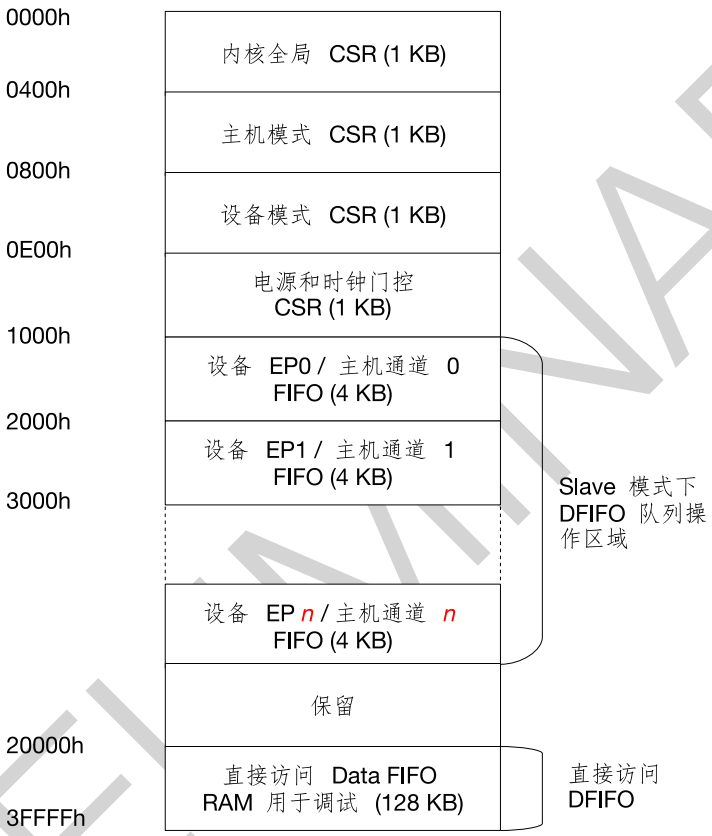


图 16-2. 内核地址映射

16.3.2.1 控制 & 状态寄存器 (CSR)

- **内核全局 CSR**
  内核全局寄存器用于配置/控制 OTG_FS 的通用功能（即主机和设备模式共同的功能），并表示其内部状态。通用功能包括 OTG 控制（HNP，SRP 和 A/B 设备检测），USB 配置（选择主机或设备模式，PHY 选择），以及系统级中断。在主机和设备模式下，软件均可以对内核全局 CSR 进行访问。

- **主机模式 CSR**
  主机模式寄存器用于主机模式下的配置/控制/状态表示，只能在主机模式下被访问。每个通道各自有一组主机模式寄存器。

- **设备模式 CSR**
  设备模式寄存器用于设备模式下的配置/控制/状态表示，只能在设备模式下被访问。每个端点各自有一组

设备模式寄存器。

- **电源和时钟门控寄存器**
  此单一寄存器用于控制模块电源和门控时钟。

16.3.2.2 FIFO 访问

OTG_FS 利用多个 FIFO 缓冲发送或接收的有效数据。FIFO 的数量和类型取决于主机或设备模式，以及所使用的通道或端点的数量（参考章节 16.3.3）。FIFO 访问有两种方式：DMA 模式和 Slave 模式。当使用 Slave 模式时，CPU 需要通过读写 DFIFO 的推入/弹出操作 (push/pop) 区域或读写调试区域来访问这些 FIFO。FIFO 访问遵循以下规则：

- 对 4 KB 推入/弹出区域中任何地址的读访问将在共享 RX FIFO 中产生一个弹出 (pop) 操作。
- 对特定 4 KB 推入/弹出区域的写访问将写入相应的端点或通道的 TX FIFO（前提是该端点是 IN 端点，或者该通道是 OUT 通道）。
  - 在设备模式下，数据写入相应的 IN 端点的专用 TX FIFO。
  - 在主机模式下，根据通道是非周期性通道还是周期性通道，数据写入非周期性 TX FIFO 或周期性 TX FIFO。
- 访问 128 KB 读写调试区域将直接读/写，而不是进行推入/弹出操作。此种访问通常仅用于调试目的。

请注意，仅在 Slave 模式下，才需要由 CPU 直接向 FIFO 进行数据操作。在 DMA 模式下，内部 DMA 将处理 TX FIFO 和 RX FIFO 的数据推入/弹出操作。

16.3.3 FIFO 和队列组织

OTG_FS 中的 FIFO 主要用于保存有效数据（USB 数据包的数据字段）。TX FIFO 用于存储将由主机模式下的 OUT 事务或设备模式下的 IN 事务发送的有效数据。RX FIFO 用于存储主机模式下 IN 事务或设备模式下 OUT 事务的已接收的有效数据。除了存储有效数据之外，RX FIFO 还存储每个有效数据的**状态条目**。状态条目中包含关于有效数据的信息，如：通道编号、字节数、是否有效等。在 Slave 模式下，状态条目还用于指示各类通道事件。

可用于 FIFO 分配的 SPRAM 大小为 256×35 位（35 位包括 32 个数据位加 3 个控制位）。各个通道（在主机模式下）或端点（在设备模式下）使用的多个 FIFO 被分配到 SPRAM 中，并且可以动态调整大小。

16.3.3.1 主机模式 FIFO 和队列

如图 16-3 所示，主机模式使用以下 FIFO：

- **非周期性 TX FIFO**：存储所有通道的批量和控制类型的 OUT 事务的有效数据。
- **周期性 TX FIFO**：存储所有通道的中断或同步类型的 OUT 事务的有效数据。
- **RX FIFO**：存储所有 IN 事务的有效数据，以及用于指示有效数据大小和事务/通道事件（例如传输完成或通道暂停）的状态条目。

除 FIFO 外，主机模式还包含两个请求队列，用于存储来自多个通道的事务请求。请求队列中的每个条目都包含 IN/OUT 通道编号以及执行事务的其他信息（例如事务类型）。请求队列也用于存储其他请求类型，例如通道暂停请求。

与 FIFO 不同，请求队列的大小是固定的，且不能由软件直接访问。相反，一旦启用了通道，主机内核会自动将请求写入请求队列。请求写入队列的顺序决定了 USB 事务的处理顺序。

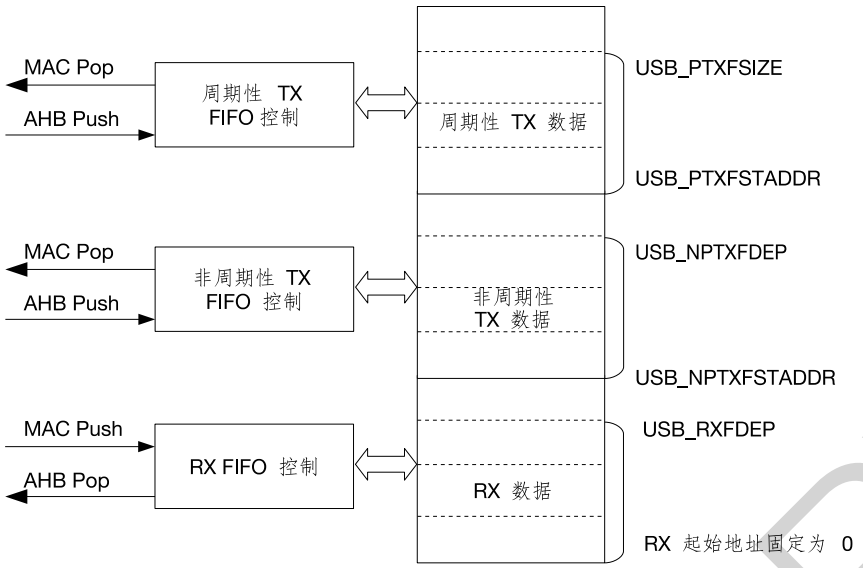


图 16-3. 主机模式 FIFO

主机模式包含以下请求队列：

- **非周期性请求队列**：针对非周期性事务（批量和控制）的请求队列，最多可以存储 4 个条目。
- **周期性请求队列**：针对周期性事务（中断和同步）的请求队列，最多可以存储 8 个条目。

调度事务时，硬件将首先执行周期性请求队列上的所有请求，再执行非周期性请求队列上的请求。

16.3.3.2 设备模式 FIFO

如图 16-4 所示，设备模式使用以下 FIFO：

- **RX FIFO**：存储数据包内接收的有效数据和状态条目（用于指示有效数据的大小）。
- **专用 TX FIFO**：每个使能的 IN 端点都有一个专用的 TX FIFO，用于存储该端点的所有 IN 有效数据，而无论事务类型（周期性或非周期性 IN 事务）。

由于有专用的 FIFO，设备模式不使用任何请求队列。IN 事务的顺序由主机确定。

16.3.4 中断层次结构

OTG_FS 有一条中断线，可以通过中断矩阵连接到一个 CPU。可以通过置位 USB_GLBLINTRMSK 来显示中断信号。OTG_FS 中断是 USB_GINTSTS_REG 寄存器中所有位的或 (OR)，且置位 USB_GINTMSK_REG 寄存器中的相应位可以使能 USB_GINTSTS_REG 中的位。USB_GINTSTS_REG 包含系统级中断，还包含主机或设备模式专有的中断位以及 OTG 有关中断。OTG_FS 中断源的层次结构如图 16-5 所示。

USB_GINTSTS_REG 寄存器的以下位指示较低层级的中断源：

- **USB_PRTINT** 表示主机端口有未处理的中断。USB_HPRT_REG 寄存器指示中断源。
- **USB_HCHINT** 表示一个或多个主机通道有未处理的中断。通过读取 USB_HAINT_REG 寄存器可以确定哪些通道有未处理的中断，然后查询该通道的 USB_HCINT*n*_REG 寄存器以确定中断源。
- **USB_OEPINT** 表示一个或多个 OUT 端点有未处理的中断。通过读取 USB_DAINT_REG 寄存器可以确定哪些 OUT 端点有未处理的中断，然后查询 OUT 端点的 USB_DOEPINT*n*_REG 寄存器以确定中断源。

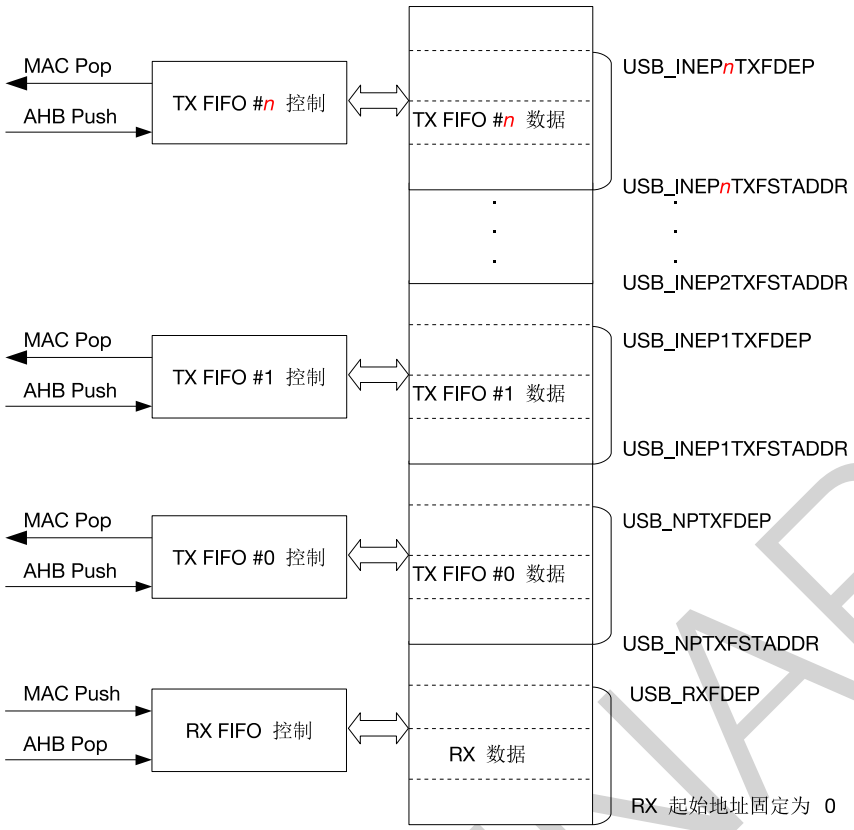


图 16-4. 设备模式 FIFO

- **USB_IEPINT** 表示一个或多个 IN 端点有未处理的中断。通过读取 USB_DAINT_REG 寄存器可以确定哪些 IN 端点有未处理的中断，然后查询 IN 端点的 USB_DIEPINT*n*_REG 寄存器以确定中断源。
- **USB_OTGINT** 表示 OTG 事件已触发中断。查询 USB_GOTGINT_REG 寄存器以确定哪些 OTG 事件触发了中断。

16.3.5 DMA 模式和 Slave 模式

USB OTG 支持 3 种存储器访问方式：Scatter/Gather DMA 模式、缓冲 DMA 模式，和 Slave 模式。

16.3.5.1 Slave 模式

在 Slave 模式下，所有有效数据放入 FIFO 或从 FIFO 中取出都必须通过 CPU 进行。

- 使用 IN 端点或 OUT 通道传输数据包时，必须将有效数据放入相应的端点或通道的 TX FIFO 中。
- 接收到数据包时，必须先通过读取 USB_GRXSTSP_REG 从 RX FIFO 中取出数据包的状态条目，以确定数据包中有效数据的长度（以字节为单位）。然后必须由 CPU 手动从 RX FIFO 中取出相应的字节数（通过读取 RX FIFO 中的存储区域）。

16.3.5.2 缓冲 DMA 模式

缓冲模式类似于 Slave 模式，不同之处在于该模式为利用内部 DMA 将有效数据放入 FIFO 或从 FIFO 中取出。

- 使用 IN 端点或 OUT 通道传输数据包时，应将有效数据的存储地址写入 USB_HCDMA*n*_REG（主机模式）或 USB_DOEPDMA*n*_REG（设备模式）寄存器。启用端点或通道后，内部 DMA 会将有效数据从存储器中

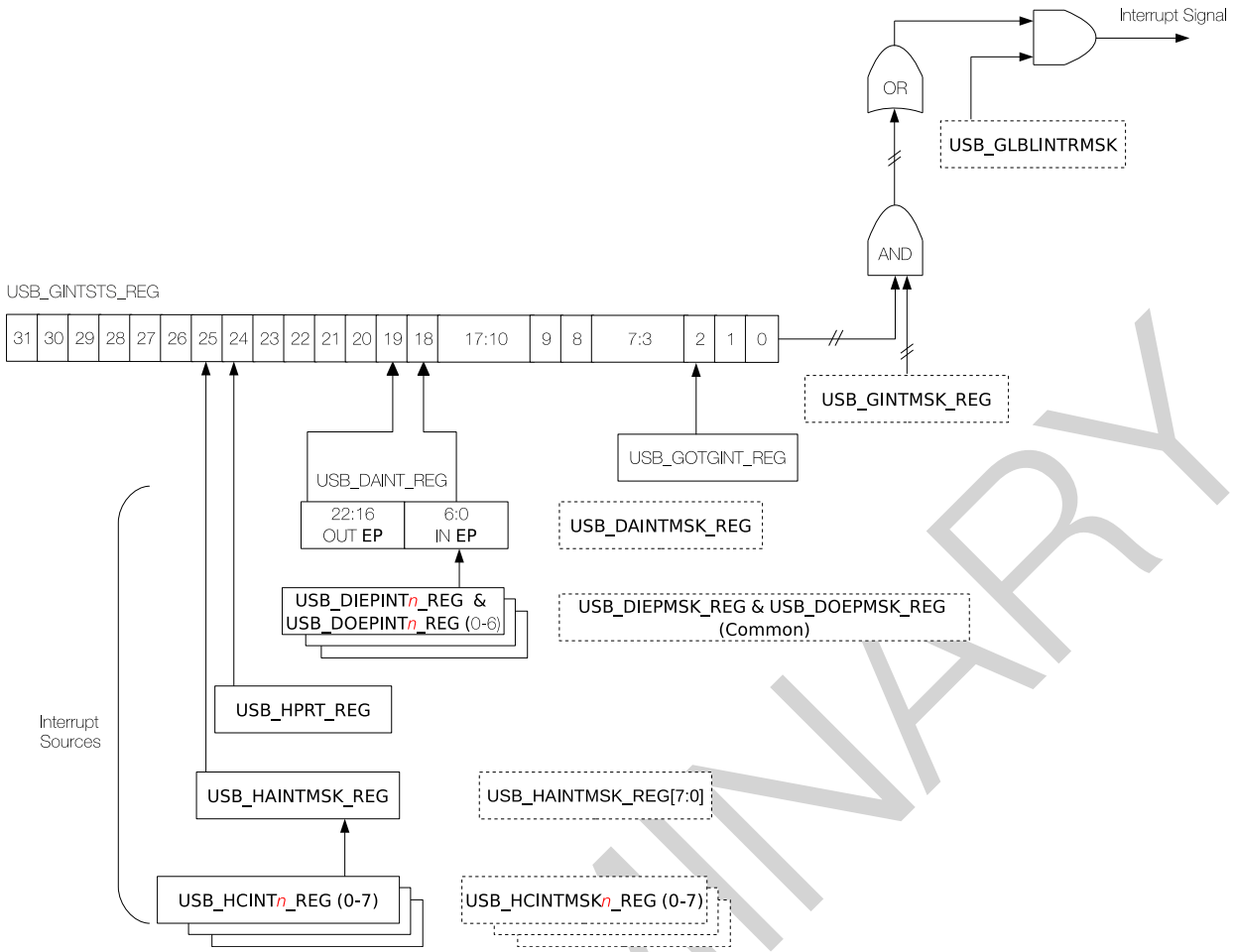


图 16-5. OTG_FS 中断层次结构图

推送到通道或端点的 TX FIFO 中。

- 使用 OUT 端点或 IN 通道接收数据包时，应将存储器中空缓冲区的地址写入 USB_HCDMA*n*_REG（主机模式）或 USB_DOEPDMA*n*_REG（设备模式）寄存器。启用端点或通道后，内部 DMA 将把有效数据从 RX FIFO 弹出到相应缓冲区中。

16.3.5.3 Scatter/Gather DMA 模式

在 Scatter/Gather DMA 模式下，包含有效数据的接收缓冲区可能分散在整个存储器各处。每个端点或通道都有一个连续的 DMA 描述符列表。每个描述符包含一个指向有效数据或接收缓冲区的 32 位指针和一个 32 位的缓冲区描述符 (BufferStatus Quadlet)。有效数据和接收缓冲区可以对应单个事务（即 < 1 MPS 字节，MPS：maximum packet size）或整个传输（即 > 1 MPS 字节）。该列表的实现为环形缓冲区，意味着 DMA 在遇到列表中的最后一个条目时将返回至第一个条目。

- 当使用 IN 端点或 OUT 通道发送传输/事务时，DMA 将从多个缓冲区收集有效数据并将其放入 TX FIFO。
- 当使用 OUT 端点或 IN 通道接收传输/事务时，DMA 将取出来自 RX FIFO 的接收有效数据，并将它们分别存放到 DMA 列表条目所指向的多个缓冲区中。

16.3.6 事务和传输级操作

在主机或设备模式下，通信可以在事务级或传输级进行。

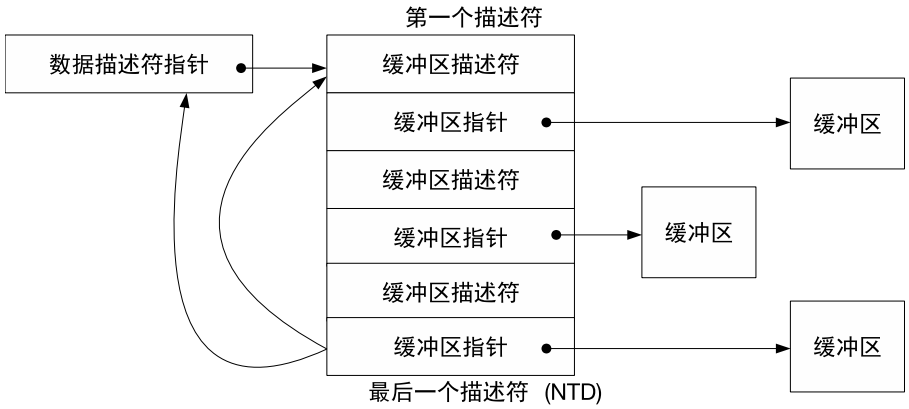


图 16-6. Scatter/Gather DMA 链表结构

16.3.6.1 DMA 模式下的事务和传输级操作

在 DMA 模式下的传输级操作，只有当一个传输通道被终止时，中断才会发生。以下三种情况下发生传输通道终止：传输通道中所有要求传送的数据全部成功传送、接收到 STALL 或出现连续事物级错误（如，3 个连续的事务级错误）。在 DMA 设备模式下操作时，所有错误均由控制器内核进行处理。

在 DMA 模式下的事务级操作，传输大小是一个数据包的大小（最大数据包大小或短数据包大小）。

16.3.6.2 Slave 模式下的事务和传输级操作

在 Slave 模式下的事务级操作，一次只能处理一个事务。每组有效数据应对应一个数据包，软件必须根据 USB 接收到的握手应答（例如 ACK 或 NAK）来确定是否需要重启事务。

下表描述了 Slave 模式下进行 IN 和 OUT 事务级操作的方法。

表 16-1. Slave 模式下的 IN 和 OUT 事务级操作

主机模式	设备模式
OUT 事务	
1. 软件配置 USB_HCTSIZ*n*_REG 寄存器，指定数据包的大小和数量（1 个），使能该通道，然后将数据包的有效数据复制到 TX FIFO 中。 2. 软件在写完每个数据包的最后一个 DWORD 之后，控制器内核将自动把请求条目写入相应的请求队列。 3. 如果该事务成功，将生成 USB_XFERCOMPL 中断。如果该事务失败，则会发生错误中断（例如 USB_H_NACK*n*）。	1. 软件配置 USB_DIEPTSIZ*n*_REG 寄存器，指定数据包的大小 (1 MPS) 和数量（1 个）。端点使能后，将等待主机向其发送数据包。 2. 接收到的数据包将与数据包状态条目一起放入 RX FIFO。 3. 如果该事务失败（例如，由于 RX FIFO 已满），则端点将回传 NAK。
IN 事务	
1. 软件配置 USB_HCTSIZ*n*_REG 寄存器，指定数据包的大小和数量（1 个），然后使能该通道。 2. 控制器内核自动将请求条目写入相应的请求队列。 3. 如果该事务成功，接收数据以及状态条目将写入 RX FIFO。否则，会产生错误中断（例如 USB_H_NACK*n*）。	1. 软件配置 USB_DIEPTSIZ*n*_REG 寄存器，指定数据包的大小和数量（1 个）。端点使能后，将等待主机从其读取数据包。 2. 数据包传输完成后，将产生 USB_XFERCOMPL 中断。

在 Slave 模式下进行传输级操作时，可以在队列中一次性排入一个或多个事务级操作，达到类似于 DMA 模式下的传输级操作的效果。在同一次触发的传输中，多个事务的数据包均可以从 FIFO 中读写，这样就无需以数据包为单位触发中断。

Slave 模式下进行传输级操作的方法类似于事务级操作，不同之处在于，需要配置 USB_HCTSIZ*n*_REG 或 USB_DOEPTSIZ*n*_REG 寄存器以指定整次传输的大小和数据包个数。使能通道或端点后，应分别向 TX FIFO 或 RX FIFO 写入或读取对应于多个数据包的有效数据（假设有足够的空间或足够的数据量）。

16.4 OTG

USB OTG 允许 OTG 设备作为 USB 主机或 USB 设备。因此，OTG 设备上一般都有一个 Mini-AB 或 Micro-AB 接口，可用于连接 A-plug 或 B-plug。当 OTG 设备连接上 A-plug/B-plug 时，其将成为 A 设备/B 设备。

- A 设备默认为主机模式（A 主机），B 设备默认为设备模式（B 外设）。
- 通过使用主机协商协议 (NHP)，A、B 设备可互相交换角色，即变更为 A 外设和 B 主机。
- A 设备可关闭 Vbus 省电。然后，B 设备可通过请求 A 设备启动 Vbus 并发起一个新的会话来唤醒 A 设备。该机制称为会话请求协议 (SRP)。
- Vbus 只能由 A 设备供电，即使 A 设备为外设模式。

OTG 设备可通过接头的 ID 管脚确定其连接的是 A-plug 还是 B-plug。A-plug 中的 ID 管脚为接地，B-plug 中的 ID 管脚则为悬空。

16.4.1 OTG 接口

OTG_FS 支持 OTG Revision 1.3 规范的 SRP 和 HNP 协议。OTG_FS 控制器内核通过 UTMI+ OTG 接口与收发器（内部或外部）连接。UTMI+ OTG 接口允许控制器内核操作收发器（比如启用/禁用 HNP 中的上拉和下拉）以实现 OTG 的功能，并且还允许收发器指示与 OTG 相关的事件。如果改用外部收发器，那么 UTMI+ OTG 将通过 GPIO 交换矩阵连接到 ESP32-S3 的 GPIO，请参阅章节 *2 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)*。表 16-2 描述了 UTMI+ OTG 接口信号。

表 16-2. UTMI OTG 接口

接口信号	I/O	描述
usb_otg_iddig_in	I	迷你 A/B 插头指示器。指示所连接的插头是 mini-A 还是 mini-B。仅在 usb_otg_idpullup 被采样断言时有效。 1'b0：连接 mini-A 1'b1：连接 mini-B
usb_otg_avalid_in	I	A 类外设会话有效。指示 Vbus 电压是否在 A 类外设会话的有效电平上。比较器阈值为： 1'b0: Vbus <0.8 V 1'b1: Vbus = 0.2 V ~ 2.0 V
usb_otg_bvalid_in	I	B 类外设会话有效。指示 Vbus 电压是否在 B 类外设会话的有效电平上。比较器阈值为： 1'b0: Vbus <0.8 V 1'b1: Vbus = 0.8 V ~ 4 V
usb_otg_vbusvalid_in	I	Vbus 有效。指示 Vbus 电压是否在 A/B 设备/外设操作的有效电平上。比较器阈值为： 1'b0: Vbus <4.4 V 1'b1: Vbus >4.75 V
usb_srp_sessend_in	I	B 设备会话结束。指示 Vbus 电压是否在 B 设备会话结束的阈值以下。比较器阈值为： 1'b0: Vbus >0.8 V 1'b1: Vbus <0.2 V
usb_otg_idpullup	O	模拟 ID 输入采样使能。使能采样模拟 ID 线。 1'b0: ID 管脚采样禁能 1'b1: ID 管脚采样使能
usb_otg_dppulldown	O	D+ 下拉电阻使能。使能 D+ 线上的 15 kΩ 下拉电阻。
usb_otg_dmpulldown	O	D- 下拉电阻使能。使能 D- 线上的 15 kΩ 下拉电阻。
usb_otg_drvvbus	O	驱动 Vbus。驱动 Vbus 到 5 V。 1'b0：不驱动 Vbus 1'b1：驱动 Vbus
usb_srp_chrgvbus	O	Vbus 输入充电使能。指示 PHY 为 Vbus 充电。 1'b0：不通过电阻为 Vbus 充电 1'b1：通过电阻为 Vbus 充电（需激活至少 30 ms）

接口信号	I/O	描述
usb_srp_dischrgvbus	O	Vbus 输入放电使能。指示 PHY 为 Vbus 放电。 1'b0：不通过电阻为 Vbus 放电 1'b1：通过电阻为 Vbus 放电（需激活至少 50 ms）

16.4.2 ID 管脚检测

寄存器 USB_GOTGCTL_REG 中的 USB_CONIDSTS 位指示 OTG 控制器为 A 设备 (1'b0) 还是 B 设备 (1'b1)。当 USB_CONIDSTS 发生改变（即连接或断开插头时），会产生 USB_CONIDSTSCHNG 中断。

16.4.3 会话请求协议 (SRP)

16.4.3.1 A 设备 SRP

图 16-7 说明了 OTG_FS 充当 A 设备（即默认主机并为 Vbus 供电）时的 SRP 流程。

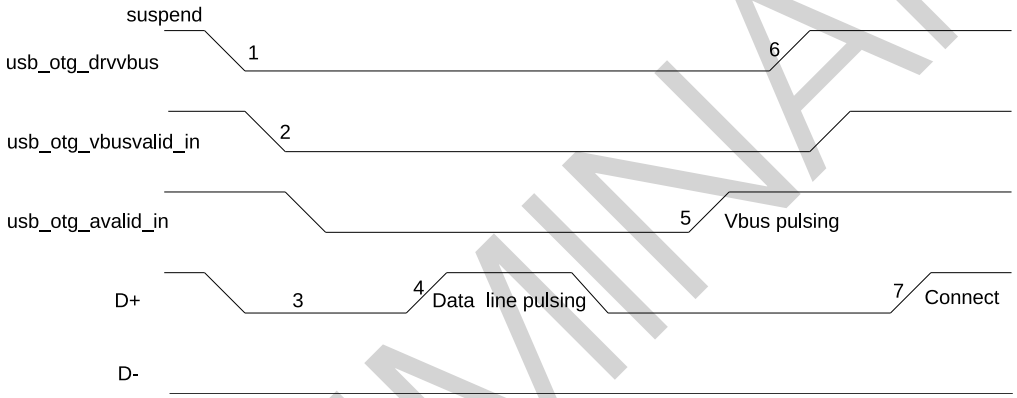


图 16-7. A 设备 SRP

1. 为了节省电能，当总线空闲时，应用程序将挂起并关闭端口电源，方法是写入主机端口控制和状态寄存器中的端口挂起位（USB_PRTSUSP 置为 1'b0）和端口电源位（USB_PRTPWR 置为 1'b0）。
2. PHY 通过使 usb_otg_vbusvalid_in 信号无效来指示端口断电。
3. 当 Vbus 电源关闭时，A 设备必须检测到 SE0 至少 2 ms 才能启动 SRP。
4. 要启动 SRP，B 设备会打开其数据线上拉电阻 5 到 10 ms。OTG_FS 内核将检测数据线脉冲。
5. 设备将 Vbus 驱动到 A 设备会话有效阈值之上（至少 2.0 V）以执行 Vbus 脉冲。OTF_FS 内核在检测 SRP 时中断应用程序。全局中断状态寄存器中的会话请求检测位 (USB_SESSREQINT) 将被置位。
6. 应用程序必须处理会话请求检测中断，并通过写入主机端口控制和状态寄存器中的端口电源位来打开端口电源位。PHY 通过确认 usb_otg_vbusvalid_in 信号来指示端口上电。
7. 当 USB 上电时，B 设备连接，完成 SRP 过程。

16.4.3.2 B 设备 SRP

图 16-8 说明了 OTG_FS 充当 B 设备（即不为 Vbus 供电）时的 SRP 流程。

1. 为了节省电能，当总线空闲时，主机（A 设备）将挂起并关闭端口电源。PHY 通过使 usb_otg_vbusvalid_in 信号无效来指示端口掉电。在检测到 3 ms 总线空闲后，OTG_FS 内核将内核中断寄存器中的早期挂起位

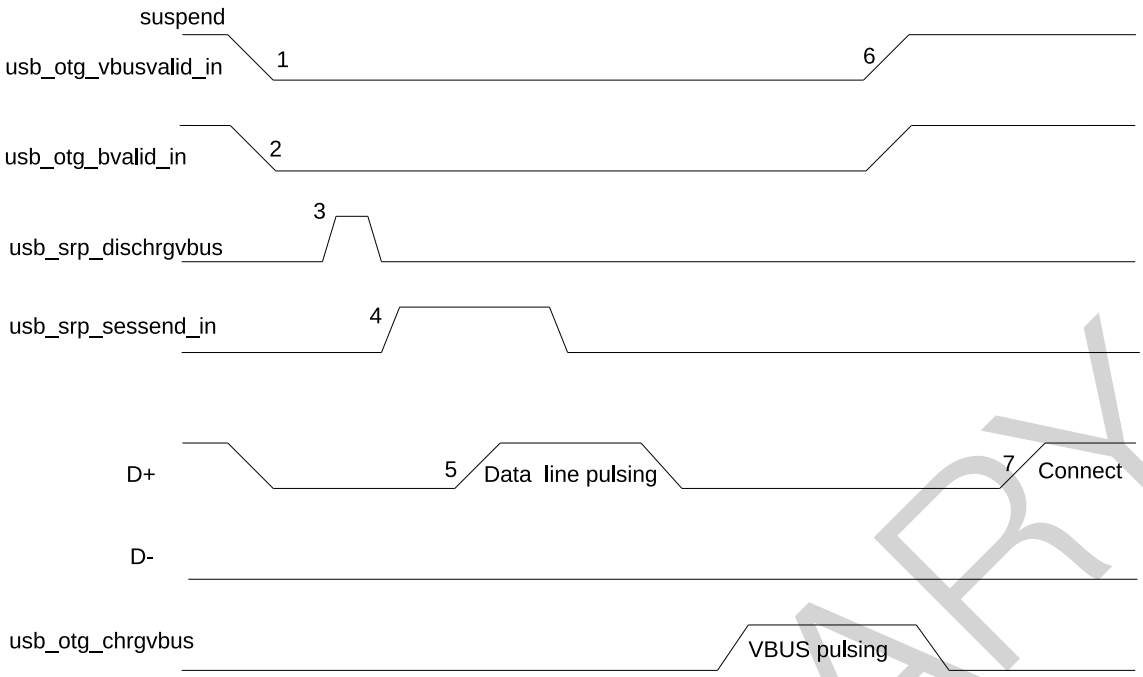


图 16-8. B 设备 SRP

（USB_ERLYSUSP 中断）置 1。之后，OTF_FS 内核将内核中断寄存器中的 USB 挂起位 (USB_USBSUSP) 置 1。PHY 通过使 usb_otg_bvalid_in 信号无效来指示 B 设备会话结束。

2. OTG_FS 内核确认 usb_otg_dischrgvbus 信号，指示 PHY 加快 Vbus 放电。
3. PHY 通过确认 usb_otg_sessend_in 信号来指示会话结束。这是 SRP 的初始条件。OTG_FS 内核在启动 SRP 之前需要检测到 SE0 2 ms。对于 USB 1.1 全速串行收发器，在 USB_BSESVLD 无效后，应用程序必须等待 Vbus 放电至 0.2 V。
4. 应用程序等待 1.5 秒（TB_SE0_SRP 时间），然后写入 OTG 控制和状态寄存器中的会话请求位 (USB_SESREQ) 并启动 SRP。OTG_FS 内核执行数据线脉冲，然后执行 Vbus 脉冲。
5. 主机（A 设备）从数据线或 Vbus 脉冲检测到 SRP，然后打开 Vbus。PHY 确认 usb_otg_vbusvalid_in 信号指示 Vbus 上电。
6. OTG_FS 内核确认 usb_srp_chrgvbus 并执行 Vbus 脉冲。主机（A 设备）打开 Vbus，启动新会话，指示 SRP 成功。OTG_FS 内核通过置位 OTG 中断状态寄存器中的会话请求成功状态改变位 (USB_SESREQSC) 来中断应用程序。应用程序读取 OTG 控制和状态寄存器中的会话请求成功位。
7. 当 USB 通电时，OTG_FS 内核连接，从而完成 SRP 过程。

16.4.4 主机协商协议 (HNP)

16.4.4.1 A 设备 HNP

图 16-9 说明了 OTG_FS 充当 A 设备时的 HNP 流程。

1. OTG_FS 内核向 B 设备发送 SetFeature b_hnp_enable 描述符以启用 HNP 支持。B 设备回复 ACK 则表明其支持 HNP。应用程序必须置位 OTG 控制和状态寄存器中的主机设置 HNP 使能位 (USB_HSTSETHNPEN) 向 OTG_FS 内核说明 B 设备支持 HNP。
2. 使用完总线后，应用程序写入主机端口控制和状态寄存器中的端口挂起位 (USB_PRTSUSP) 进入挂起状态。

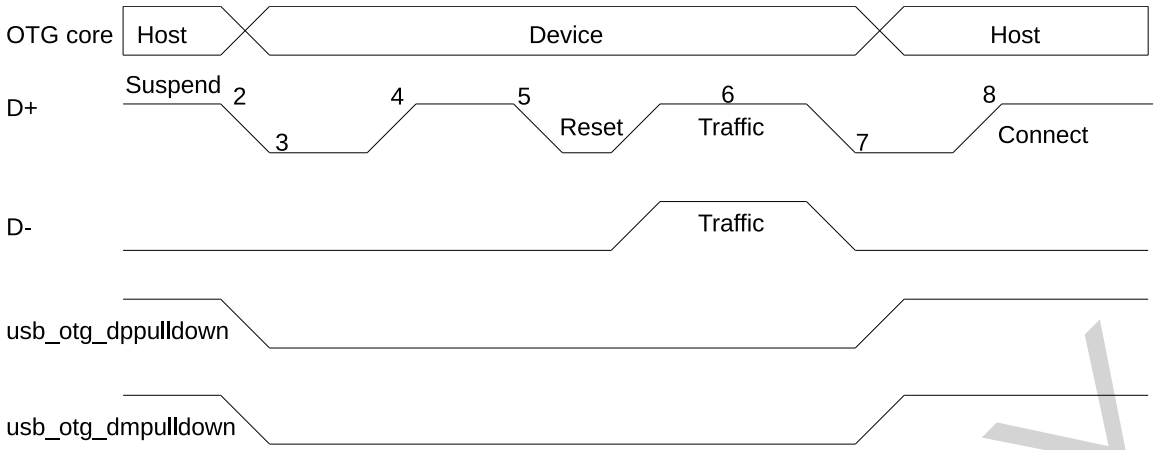


图 16-9. A 设备 HNP

3. 当 B 设备观察到 USB 挂起时，它将断开连接，表明 HNP 的初始状态。B 设备仅在必须切换到主机角色时才启动 HNP；否则，总线将继续挂起。OTG_FS 内核在 OTG 中断状态寄存器中置位主机协商中断位 (USB_HSTNEGDET)，指示 HNP 的开始。OTG_FS 内核将 usb_otg_dppulldown 和 usb_otg_dmpulldown 信号置为无效，指示设备角色。PHY 启用 D+ 上拉电阻，指示 B 设备的连接。应用程序必须读取 OTG 控制和状态寄存器中的当前模式位 (USB_CURMOD_INT) 以确定设备模式。
4. B 设备检测到连接，发出 USB 复位，对 OTG_FS 内核进行枚举以开始数据通信。
5. B 设备继续充当主机角色，启动数据通信，并在完成后挂起总线。OTG_FS 内核在检测到 3 ms 总线空闲之后，将内核中断寄存器中的早期挂起位 (USB_ERLYSUSP) 置 1。之后，OTG_FS 内核将内核中断寄存器中的 USB 挂起位 (USB_USBSUSP) 置 1。
6. 在协商模式下，OTG_FS 内核检测到挂起，断开连接并切换回主机角色。OTG_FS 内核确认 usb_otg_dppulldown 和 usb_otg_dmpulldown 信号，以表明其承担了主机角色。
7. OTG_FS 内核将 OTG 中断状态寄存器中的连接器 ID 状态改变中断 (USB_CONIDSTS) 置位。应用程序必须读取 OTG 控制和状态寄存器中的连接器 ID 状态，以确定 OTG_FS 内核作为 A 设备。这表明该应用程序已完成 HNP。应用程序必须读取 OTG 控制和状态寄存器中的当前模式位，以确定主机模式操作。
8. B 设备连接，完成 HNP 过程。

16.4.4.2 B 设备 HNP

图 16-10 说明了 OTG_FS 充当 B 设备时的 HNP 流程。

1. A 设备发送 SetFeature b_hnp_enable 描述符以启用 HNP 支持。OTG_FS 内核回复 ACK 响应以表明其支持 HNP。应用程序必须将 OTG 控制和状态寄存器中的设备 HNP 使能位 (USB_DEVHNPEN) 置 1，以表明支持 HNP。应用程序将 OTG 控制和状态寄存器中的 HNP 请求位 (USB_DEVHNPEN) 置 1，以指示 OTG_FS 内核启动 HNP。
2. A 设备使用完总线后，将挂起总线。
   (a) OTG_FS 内核在总线空闲 3 ms 之后将内核中断寄存器中的早期挂起位 (USB_ERLYSUSP) 置 1。之后，OTG_FS 内核将内核中断寄存器中的 USB 挂起位 (USB_USBSUSP) 置 1。OTG_FS 内核断开连接，并且 A 设备检测到总线上的 SE0，指示 HNP。
   (b) OTG_FS 内核确认 usb_otg_dppulldown 和 usb_otg_dmpulldown 信号，以表明其承担了主机角色。
   (c) A 设备通过在检测到 SE0 的 3 ms 内激活 D+ 上拉电阻来做出响应。OTG_FS 内核将检测到连接。

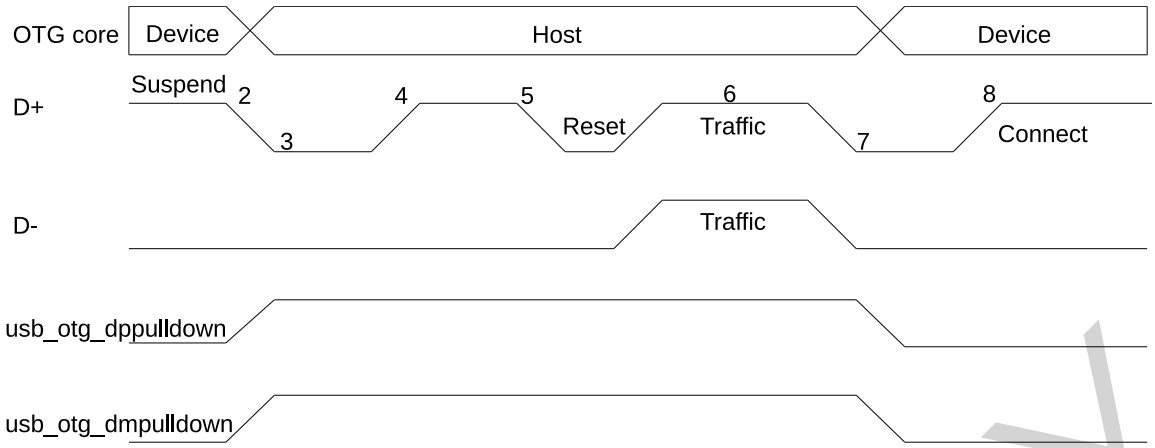


图 16-10. B 设备 HNP

(d) OTG_FS 内核将 OTG 中断状态寄存器中的主机协商成功状态改变位 (USB_CONIDSTS) 置 1，以指示 HNP 状态。应用程序必须读取 OTG 控制和状态寄存器中的主机协商成功位 (USB_HSTNEGSCS)，才能确定主机协商成功。应用程序必须读取内核中断寄存器中的当前模式位 (USB_CURMOD_INT)，才能确定主机模式操作。

3. 将 USB_PRTPWR 位设置为 1'b1，以驱动 USB 上的 Vbus。
4. 等待 USB_PRTCONNDET 中断。这表明设备已连接到端口。
5. 应用程序将重置位 (USB_PRTRST) 置为 1，OTG_FS 内核将发出 USB 重置，对 A 设备进行枚举并开始数据通信。
6. 等待 USB_PRTENCHNG 中断。
7. OTG_FS 内核继续充当启动数据通信的主机角色，完成后，通过写入主机端口控制和状态寄存器中的端口挂起位 (USB_PRTSUSP) 将总线挂起。
8. 在协商模式下，当 A 设备检测到挂起时，它将断开连接并切换回主机角色。OTG_FS 内核将 usb_otg_dppulldown 和 usb_otg_dmpulldown 信号置为无效，以指示承担设备角色。
9. 应用程序必须读取内核中断寄存器中的当前模式位 (USB_CURMOD_INT)，以确定主机模式操作。
10. OTG_FS 内核连接，完成 HNP 过程。

17 SD/MMC 主机控制器 (SDHOST)

17.1 概述

ESP32-S3 存储卡接口控制器提供了一个访问安全数字输入输出卡 (SDIO)、MMC 卡和 CE-ATA 设备的硬件接口，用于连接高级外围设备总线 (APB) 和外部存储设备。该控制器支持两个外部卡（卡 0 和卡 1）。所有 SD/MMC 模块接口信号都须通过 GPIO 矩阵传输至 GPIO pad。

17.2 主要特性

该模块支持以下特性：

- 支持两个外部卡
- 支持 3.0、3.01 版本 SD 存储卡标准
- 支持 4.41、4.5、4.51 版本 MMC
- 支持 1.1 版本 CE-ATA
- 支持 1-bit、4-bit 和 8-bit 位宽模式

SD/MMC 控制器连接的拓扑结构如图 17-1 所示。该控制器支持两组外设工作，但不支持同时工作。

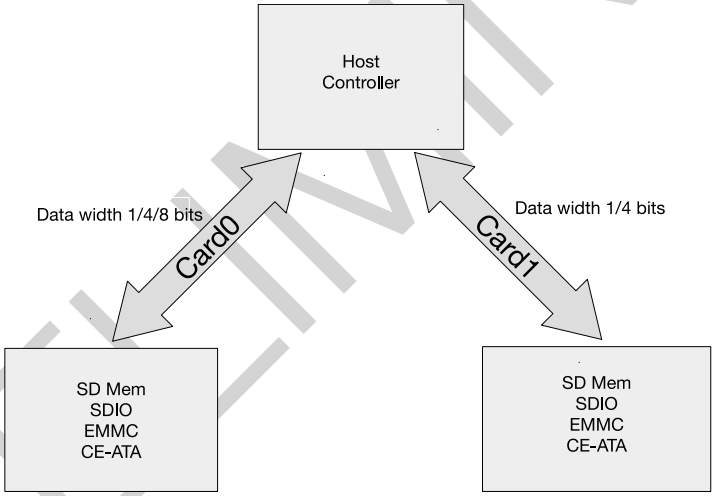


图 17-1. SD/MMC 控制器连接的拓扑结构

17.3 SD/MMC 外部接口信号

SD/MMC 的外部接口信号主要为时钟信号 (sdhost_cclk_out_1.eg:card1)、命令信号 (sdhost_ccmd_out_1)、数据信号 (sdhost_cdata_in_1[7:0]/sdhost_cdata_out_1[7:0])，SD/MMC 控制器可通过这些外部接口信号与外部设备通信。其它信号还包括卡中断信号、卡检测信号和写保护信号等。各个信号的方向如图 17-2 所示。每个管脚的方向和描述如表 17-1 所示。

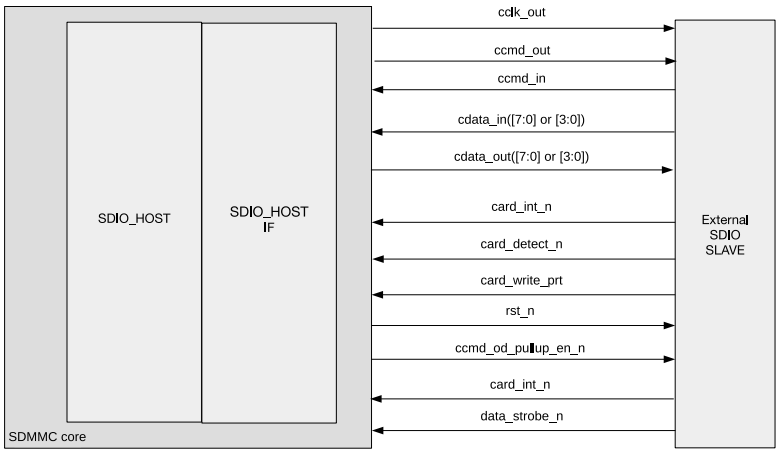


图 17-2. SD/MMC 控制器外部接口信号

表 17-1. SD/MMC 信号描述

管脚	方向	描述
sdhost_cclk_out	输出	向从机发送时钟信号
sdhost_ccmd	双向	双向命令/响应线
sdhost_cdata	双向	双向数据读/写线
sdhost_card_detect_n	输入	卡检测输入线
sdhost_card_write_prt	输入	卡写保护状态输入

17.4 功能描述

17.4.1 SD/MMC 主机控制器结构

SD/MMC 主机控制器主要由两大功能块组成，如图 17-3 所示，分别为：

- 总线接口单元 (BIU)：提供 APB 总线访问寄存器的接口、RAM 数据访问方式和 DMA 数据读写操作
- 卡接口单元 (CIU)：处理外部存储卡的接口协议，控制时钟。

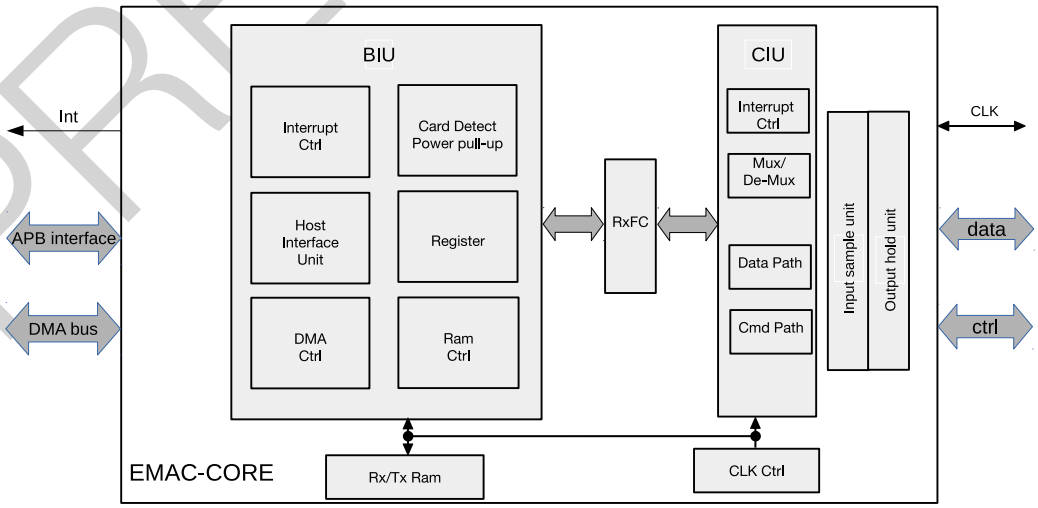


图 17-3. SDIO 主机结构框图

17.4.1.1 总线接口单元 (BIU)

该模块提供了通过主机接口单元 (HIU) 访问寄存器和 RAM 数据的方式。除此之外，它通过 DMA 接口提供独立的 RAM 数据访问。它通过 DMA 接口访问 Memory 中的数据。图 17-3 描述了该模块的内部结构。图 17-9 描述了时钟相位选择。

BIU 支持以下功能：

- 主机接口
- DMA 接口
- 中断控制
- 寄存器访问
- FIFO 访问
- 上电/上拉控制和卡检测

17.4.1.2 卡接口单元 (CIU)

该模块实现卡专用接口协议。在 CIU 中，命令通路 (Cmd Path) 控制单元和数据通路 (Data Path) 控制单元将控制器分别连接到 SD/MMC/CE-ATA 卡的命令接口和数据接口。CIU 还提供时钟控制。图 17-3 描述了 CIU 的内部结构。

CIU 由以下主要功能模块组成：

- 命令通路
- 数据通路
- SDIO 中断控制
- 时钟控制
- Mux/De-Mux 单元

17.4.2 命令通路

命令通路具有以下功能：

- 配置时钟参数
- 配置卡命令参数
- 向卡总线发送命令 (sdhost_ccmd_out)
- 从卡总线接收响应 (sdhost_ccmd_in)
- 向 BIU 发送响应
- 在命令线上发送 P-bit

命令通路状态机如图 17-4 所示。

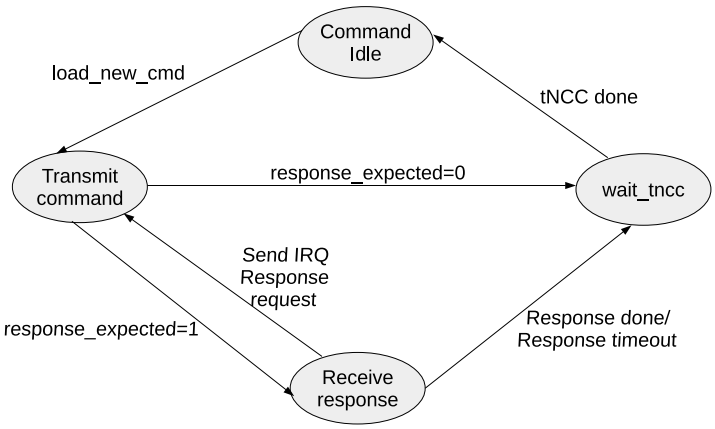


图 17-4. 命令通路状态机

17.4.3 数据通路

发送时，数据通路模块会取出 RAM 中的数据，并将其发送至 sdhost_cdata_out；接收时，数据通路会接收 sdhost_cdata_in 上的数据，并将其导入 RAM 中。数据发送命令未运行时，数据通路将加载新的数据参数，即希望发送的数据、读/写数据发送、流/块发送、块大小、字节数、卡类型、超时寄存器等。

如果在命令寄存器 (SDHOST_CMD_REG) 中置了 SDHOST_DATA_EXPECTED 位，则新命令为数据传输命令，数据通路将执行以下操作之一：

- 若 SDHOST_READ_WRITE 位为 1，发送数据
- 若 SDHOST_READ_WRITE 位为 0，接收数据

17.4.3.1 数据发送

接收到数据写入命令后，该模块将在两个时钟周期开始发送数据。即使命令通路在响应信息中检测到响应错误或循环冗余检查 (CRC) 错误，数据发送也会正常进行。但是，如果直到响应超时仍没有从卡接收到响应，则不发送数据。根据 SDHOST_CMD_REG 寄存器中 SDHOST_TRANSFER_MODE 位的值，数据发送状态机将数据以流或块的形式加至卡数据总线上。数据发送状态机如图 17-5 所示。

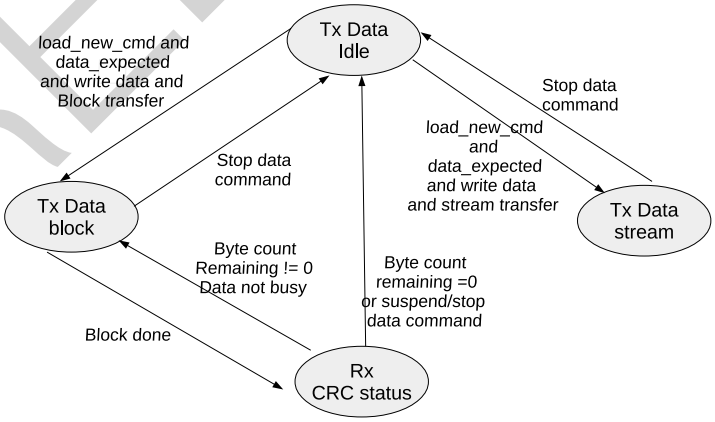


图 17-5. 数据发送状态机

17.4.3.2 数据接收

数据读取命令完成两个时钟周期后，该模块将开始接收数据。即使命令通路检测到响应错误或 CRC 错误，数据发送也会正常进行。如果一直未从卡接收到响应而发生了响应超时，则 BIU 无法接收到 CIU 数据发送完成的信号。如果 CIU 发送的命令是卡所禁止的非法操作，那么将无法开始读取从卡发送的数，且 BIU 也不会接收到 CIU 数据发送完成的信号。

若在数据超时前未接收到数据，数据通路将向 BIU 发出数据超时信号并结束数据传输。根据 SDHOST_CMD_REG 寄存器中的 SDHOST_TRANSFER_MODE 位的值，数据接收状态机以流或块的形式从卡数据总线获取数据。数据接收状态机如图 17-6 所示。

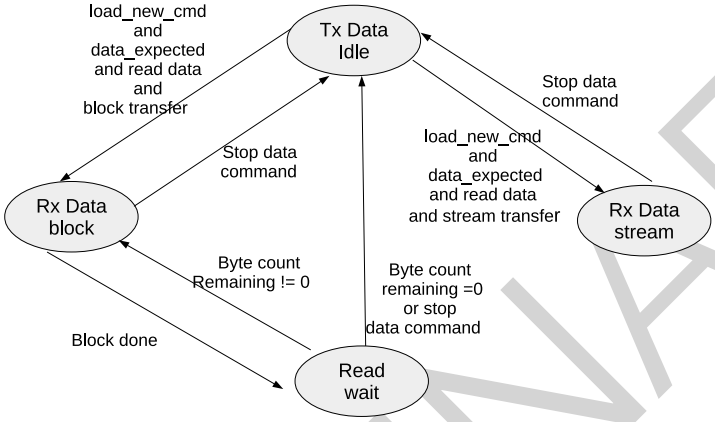


图 17-6. 数据接收状态机

17.5 CIU 操作的软件限制

- 一次只能选择一个卡执行命令或发送数据。例如，当向某张卡发送数据或从这张卡接收数据时，不可以给另一张卡发送其它新的命令。但是，可以将新的命令发送到同一张卡上，使其读取设备状态或停止数据传输。
- 一次只能发出一个数据传输命令。
- 在开放式写卡操作期间，如果由于 RAM 为空导致卡时钟停止，那么软件必须先将数据填充到 RAM 中并启动卡时钟，然后才可以向卡发出一个停止/中止命令。
- 在 SDIO/Combo 卡数据传输期间，如果卡功能暂停，并且软件想要恢复所暂停的数据传输，则其必须先重置 RAM（置位 SDHOST_FIFO_RESET）并启动恢复命令，这和启动一个新的数据传输命令相似。
- 要在进行卡数据传输中发出卡复位命令(CMD0、CMD15 或 CMD52_reset)，软件必须在 SDHOST_CMD_REG 寄存器上置位 SDHOST_STOP_ABORT_CMD，保证 CIU 可以在发出卡复位命令后停止数据传输。
- 当在 SDHOST_RINTSTS_REG 寄存器中设置数据结束位错误时，CIU 无法控制 SDIO 中断。因此在该情况下，软件应忽略 SDIO 中断，并向卡发出停止/中止命令使其停止发送数据。
- 在一次读卡过程中，如果由于 RAM 已满而导致卡时钟停止，那么软件将至少读取两个 RAM 地址来重启卡时钟。
- 在一次命令/数据传输中只能选取一个 CE-ATA 设备。例如，当一个 CE-ATA 设备已经在传输数据，则其它 CE-ATA 设备不可传输新命令。
- 使能 CE-ATA 设备的中断 (nIEN=0) 后，如果该设备已经有正在执行的 SDHODT_RW_BLK 命令，则不可以再向这一设备发送新的 SDHODT_RW_BLK 命令。只有在等待 CCS 时可以发送 CCSD。

- 但是，如果一个 CE-ATA (nIEN=1) 设备未使能，则可以向同一设备发送新的命令来读取状态信息。
- CE-ATA 设备不支持开放式数据传输。
- CE-ATA 数据传输不支持 sdhost_send_auto_stop 信号（禁止软件置位 SDHOST_SEND_AUTO_STOP）。

置位命令起始位后，在发送出命令之前以下寄存器的值不可改变：

- CMD — 命令
- CMDARG — 命令参数
- BYTCNT — 字节计数
- BLKSIZ — 块大小
- CLKDIV — 时钟分频器
- CKLENA — 时钟使能
- CLKSRC — 时钟源
- TMOUT — 超时
- CTYPE — 卡类型

17.6 收发数据 RAM

RAM 子模块是一个收发数据缓冲区，分为接收和发送两个单元。也可通过 CPU 和 DMA 实现数据的收发，从而进行读写操作，可参阅章节 17.8。

17.6.1 TX RAM 模块

可通过两种方式使能写入操作：DMA 和 CPU 读写。

如果使能 SDIO 发送，那么可以通过 APB 接口将数据写入到 TX RAM 模块中。此时，数据将通过 APB 接口直接从 CPU 写入到 SDHOST_BUFFIFO_REG 寄存器中。

除此之外，还可以通过 DMA 实现数据发送。

17.6.2 RX RAM 模块

可通过两种方式使能读取操作：DMA 和 CPU 读写。

当数据通路接收到数据时，该数据将被写入 RX RAM 中。可在读取端通过 APB 读取这些数据，APB 可直接读取寄存器 SDHOST_BUFFIFO_REG。

除此之外，还可以通过 DMA 实现数据接收。

17.7 DMA 链表环

链表环中的每个链表模块都由两部分组成：链表和一个数据缓冲区。也就是说，每一个链表都指向一个独特的数据缓冲区，同时还连接至下一个链表模块。链表环结构如图 17-7 所示。

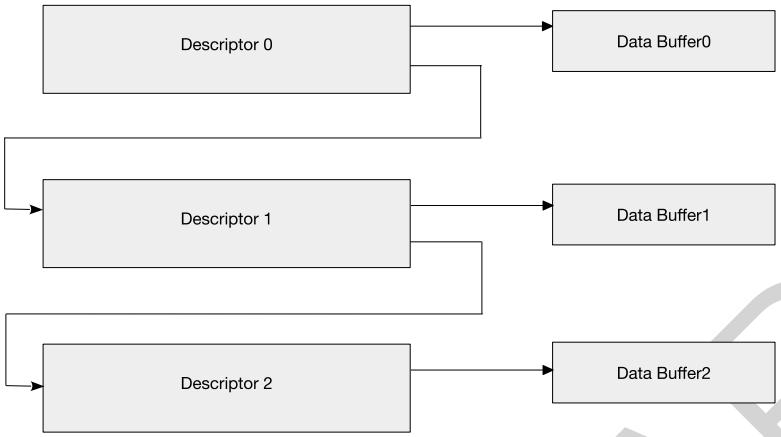


图 17-7. 链表环结构

17.8 DMA 链表结构

每个链表由 4 个字组成，如图 17-8 所示。表 17-2、表 17-3、表 17-4、表 17-5 为这 4 个字的描述。

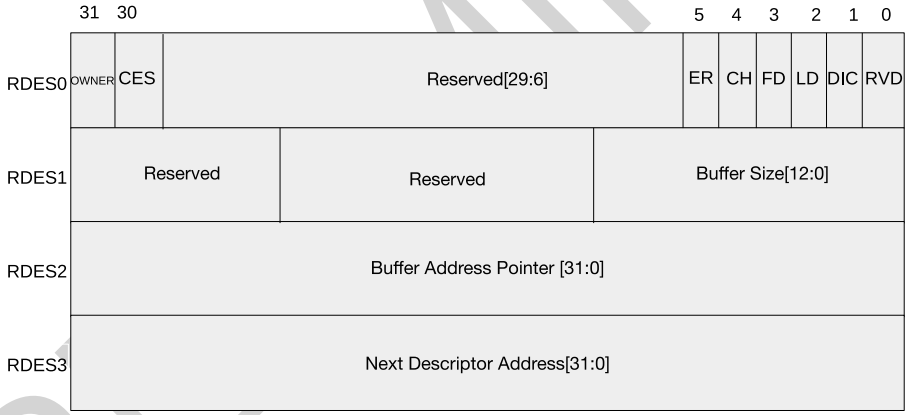


图 17-8. 链表结构

DES0 单元包含控制和状态信息。

表 17-2. DES0 单元描述

位	名称	描述
31	OWNER	置位时，表明该描述符允许的操作者为 DMA 控制器。该位被重置时，表明该描述符允许的操作者为主机。DMA 控制器在完成数据传输后清除该位。
30	CES (Card Error Summary)	这些错误位显示卡的读/写状态。 以下各位也存储于 SDHOST_RINTSTS_REG 中，为或运算： • EBE：结束位错误 • RTO：响应超时 • RCRC：响应 CRC • SBE：起始位错误 • DRTO：读取数据超时 • DCRC：对接收数据进行循环冗余校验 • RE：响应错误
29:6	保留	保留
5	ER (End of Ring)	置位时，表示当前描述符为数据包的最后一个描述符。此时，DMA 控制器返回到链表的基地址，创建一个链表环。
4	CH (Second Address Chained)	置位时，表示该描述符中的第二个地址为下一描述符的地址。此时，BS2 (DES1[25:13]) 应全部归零。
3	FD (First Descriptor)	置位时，表示该描述符内包含了数据的第一个缓冲区。若该缓冲区大小为 0，则下一个描述符内为数据的开始。
2	LD (Last Descriptor)	该位与 DMA 传输的最后一个数据块相关。置位时，表示该描述符指向的缓冲区是数据的最后一个缓冲区。在该描述符完成之后，剩余字节数为 0。也就是说，带有被置位的 LD 位的描述符完成之后，剩余字节数应为 0。
1	DIC (Disable Interrupt on Completion)	置位时，为了保留在该描述符指向的缓冲区中结束的数据，该位将阻止置位 DMA 状态寄存器 (IDSTS) 上 TI/RI 位。
0	保留	保留

DES1 单元包含缓冲区大小。

表 17-3. DES1 单元描述

位	名称	描述
31:26	保留	保留
25:13	保留	保留
12:0	BS（缓冲区大小）	表示数据缓冲区的字节大小。该数值必须为 4 的倍数。否则，其大小将无法被定义成功。该字段不可设置为 0。

DES2 单元包含指向数据缓冲区的地址指针。

表 17-4. DES2 单元描述

位	名称	描述
31:0	Buffer Address Pointer （数据缓冲区地址指针）	表示数据缓冲区的物理地址，须按字对齐。

如果当前的描述符不是链表环中的最后一个，则 DES3 单元包含指向下一个描述符的地址指针。

表 17-5. DES3 单元描述

位	名称	描述
31:0	Next Descriptor Address （下一个链表地址）	如果置位 CH (DES0[4])，则该位中有指向包含下一个描述符位置的物理内存的指针。 如果该描述符不是链表环结构中的最后一个描述符，则下一个描述符的地址指针必须满足 DES3[1:0] = 0。

17.9 初始化

17.9.1 DMA 控制器初始化

DMA 控制器初始化过程如下：

1. 向 DMA 控制器的总线模式寄存器 (SDHOST_BMOD_REG) 写入数据，设置主机总线访问参数。
2. 向 DMA 控制器的中断使能寄存器 (SDHOST_IDINTEN_REG) 写入数据，屏蔽不必要的中断类型。
3. 软件驱动器创建发送链表或接收链表。然后向 DMA 控制器链表基地址寄存器 (SDHOST_DBADDR_REG) 中写入链表的起始地址。
4. DMA 控制器引擎尝试从链表中获取描述符。

17.9.2 DMA 控制器数据发送初始化

DMA 控制器发送数据的过程如下：

1. 主机设置发送数据的描述符单元 (DES0 ~ DES3)，置位 OWNER 位 (DES0[31])，同时准备数据缓冲区。
2. 主机在 BIU 的命令 (CMD) 寄存器中写入数据命令。
3. 主机设置所需的数据发送阈值（在 SDHOST_FIFOTH_REG 寄存器中的 SDHOST_TX_WMARK 字段进行）。
4. DMA 控制器引擎读取描述符并检查 OWNER 位。如果 OWNER 位未置位，表明该描述符所允许的操作者为主机。此时，DMA 控制器进入挂起状态，并在 SDHOST_IDSTS_REG 寄存器中产生禁能描述符中断。然后，主机需在 SDHOST_PLDMND_REG 中写入任意值来释放 DMA 控制器资源。
5. DMA 控制器等待 DHOST_RINTSTS_REG 寄存器中的命令完成 (CD) 位，如果 BIU 无报错，则表明数据发送已完成。
6. 然后，DMA 控制器引擎等待 BIU 发送的 DMA 总线请求，该请求将基于配置的数据发送阈值生成。对于使用突发传输不能访问的最后一个字节，则在 AHB 总线上执行单次发送。

7. DMA 控制器从主机存储器的数据缓冲区获取发送数据，并通过 RAM 将其发送至卡中。
8. 当数据跨越多个描述符时，DMA 控制器将获取下一个描述符，并继续使用后面的描述符进行后续操作。最后一个描述符位将显示该数据是否跨越多个描述符。
9. 当数据发送完成且 SDHOST_IDSTS_TI 位已使能，将通过置位 SDHOST_IDSTS_TI 将状态信息更新至寄存器 SDHOST_IDSTS_REG。同时，DMA 控制器将通过对 DES0 执行写操作来清除 OWNER 位。

17.9.3 DMA 控制器数据接收初始化

DMA 控制器接收数据的过程如下：

1. 主机设置接收数据的描述符单元 (DES0 ~ DES3)，置位 OWNER 位 (DES0[31])。
2. 主机在 BIU 的命令寄存器中写入读数据命令。
3. 主机设置所需的数据接收阈值（在 SDHOST_FIFOTH_REG 寄存器中的 SDHOST_RX_WMARK 字段进行）。
4. DMA 控制器引擎读取描述符并检查 OWNER 位。如果 OWNER 位未置位，表明该描述符所允许的操作者为主机。此时，DMA 控制器进入挂起状态，并在 SDHOST_IDSTS_REG 寄存器中产生禁能描述符中断。然后，软件需在 SDHOST_PLDMND_REG 中写入任意值释放 DMA 控制器资源。
5. DMA 控制器等待命令完成位 (CD)，如果 BIU 无报错，则表明数据接收已完成。
6. DMA 控制器引擎等待 BIU 发送的 DMA 总线请求。该请求将基于配置的数据接收阈值生成。对于使用突发传输不能访问的最后一个字节，则在 AHB 总线上执行单次传输。
7. DMA 控制器从 RAM 获取数据，并将其发送至主机存储器。
8. 当数据跨越多个描述符时，DMA 控制器将获取下一个描述符，并继续使用后面的描述符进行后续操作。最后一个描述符位将显示该数据是否跨越多个描述符。
9. 当数据接收完成且 SDHOST_IDSTS_RI 位已使能，将通过置位 SDHOST_IDSTS_RI 将状态信息更新至寄存器 SDHOST_IDSTS_REG。同时，DMA 控制器将通过对 DES0 执行写操作来清除 OWNER 位。

17.10 时钟相位选择

如果输入数据信号或输出数据信号的建立时间的时序不符合要求，用户可以参照下图 17-9 改变时钟相位。

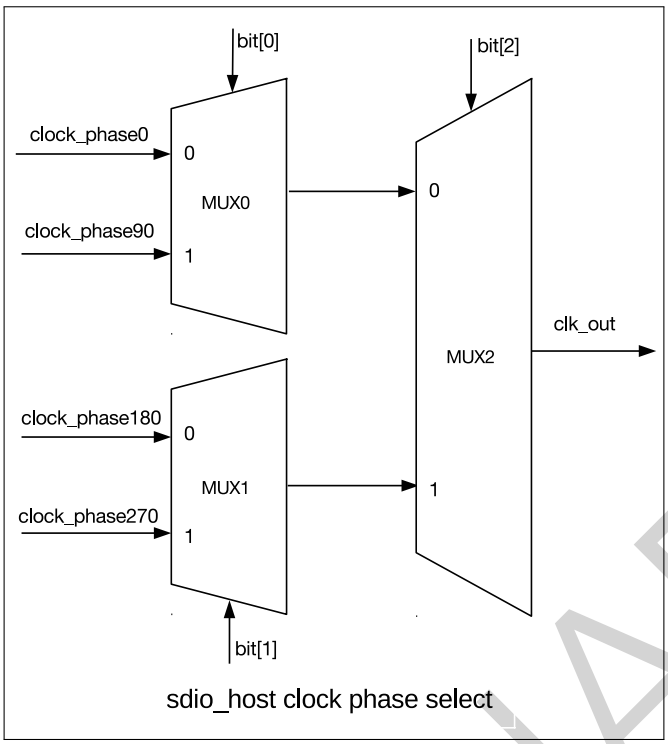


图 17-9. 时钟相位选择

可通过设置寄存器 SDHOST_CLK_DIV_EDGE_REG 来解决时序问题。例如，清除 CCLKIN_EDGE_DRV_SEL 位发送相位 0 中的输出数据，然后置位 CCLKIN_EDGE_SAM_SEL 选择 90 度相位采样 SDIO 从机中的数据。如果此时仍有时序问题，还可通过向 CCLKIN_EDGE_SAM_SEL 写入 4 或 6 分别选择 180 度或 270 度相位对 SDIO 从机进行数据采样。

有关时钟相位选择寄存器 SDHOST_CLK_DIV_EDGE_REG 的说明，请见寄存器章节 17.13。

表 17-6. SDHOST 时钟相位选择

时钟相位	phase_select 数值
0	0
90	1
180	4
270	6

17.11 中断

中断可在多个事件中产生。SDHOST_IDSTS_REG 寄存器中包含所有可能导致中断的位。SDHOST_IDINTEN_REG 寄存器中包含所有可导致中断的事件的使能位。

SDHOST_IDSTS_REG 寄存器中有两组中断汇总：正常中断汇总 (第 8 位：SDHOST_IDSTS_NIS) 和异常中断汇总 (第 9 位：SDHOST_IDSTS_AIS)。置位相应的位，可以清除中断。当某组中的所有使能中断都被清除，相应的汇总位将被清零。当两组的汇总位都被清零，连接 CPU 的中断信号将撤销（停止发送信号）。

中断不排序，如果中断在驱动程序响应之前发生，则不会产生其他中断。例如，SDHOST_IDSTS_RI 表示一个或多个数据已发送至主机缓冲区。

并发事件只会产生一个中断。驱动程序须扫描 SDHOST_IDSTS_REG 寄存器来查找导致中断的原因。

17.12 寄存器列表

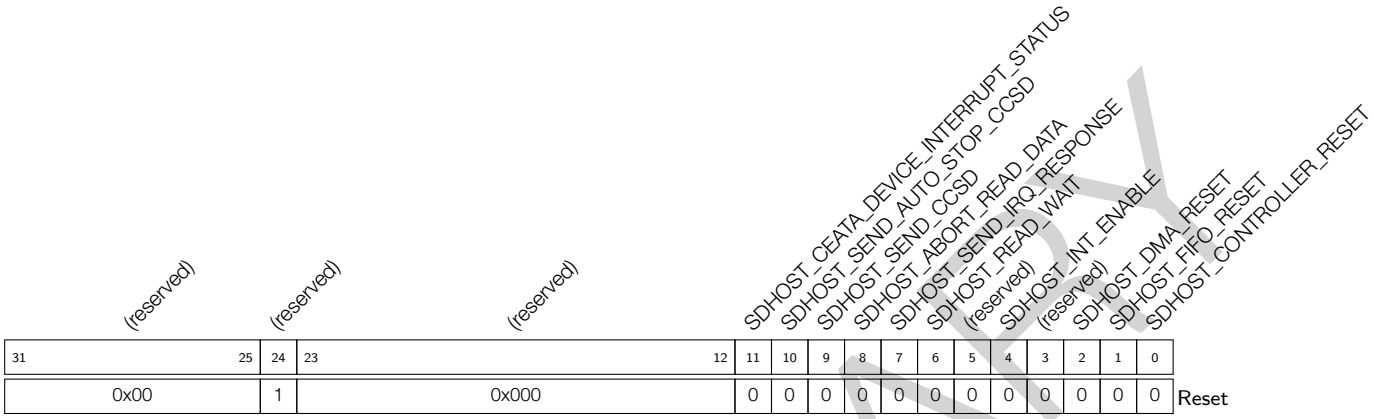
本小节的所有地址均为相对于 SD/MMC Host Controller 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

名称	描述	地址	访问
SDHOST_CTRL_REG	控制寄存器	0x0000	读/写
SDHOST_CLKDIV_REG	时钟分频器配置寄存器	0x0008	读/写
SDHOST_CLKSRC_REG	时钟源选择寄存器	0x000C	读/写
SDHOST_CLKENA_REG	时钟使能寄存器	0x0010	读/写
SDHOST_TMOUT_REG	数据和响应超时配置寄存器	0x0014	读/写
SDHOST_CTYPE_REG	卡总线宽度配置寄存器	0x0018	读/写
SDHOST_BLKSIZ_REG	卡数据块大小配置寄存器	0x001C	读/写
SDHOST_BYTCNT_REG	数据传输长度配置寄存器	0x0020	读/写
SDHOST_INTMASK_REG	SDIO 中断屏蔽寄存器	0x0024	读/写
SDHOST_CMDARG_REG	指令参数数据寄存器	0x0028	读/写
SDHOST_CMD_REG	指令和启动配置寄存器	0x002C	读/写
SDHOST_RESP0_REG	响应数据寄存器	0x0030	只读
SDHOST_RESP1_REG	长响应数据寄存器 0	0x0034	只读
SDHOST_RESP2_REG	长响应数据寄存器 1	0x0038	只读
SDHOST_RESP3_REG	长响应数据寄存器 2	0x003C	只读
SDHOST_MINTSTS_REG	屏蔽的中断状态寄存器	0x0040	只读
SDHOST_RINTSTS_REG	原始中断状态寄存器	0x0044	读/写
SDHOST_STATUS_REG	SD/MMC 状态寄存器	0x0048	只读
SDHOST_FIFOTH_REG	FIFO 配置寄存器	0x004C	读/写
SDHOST_CDETECT_REG	卡检测寄存器	0x0050	只读
SDHOST_WRTPRT_REG	卡写保护状态寄存器	0x0054	只读
SDHOST_TCBCNT_REG	传输字节计数寄存器	0x005C	只读
SDHOST_TBBCNT_REG	传输字节计数寄存器	0x0060	只读
SDHOST_DEBNCE_REG	去抖过滤器时间配置寄存器	0x0064	读/写
SDHOST_USRID_REG	用户 ID (scratchpad) 寄存器	0x0068	读/写
SDHOST_RST_N_REG	卡重置寄存器	0x0078	读/写
SDHOST_BMOD_REG	突发模式传输配置寄存器	0x0080	读/写
SDHOST_PLDMND_REG	轮询命令配置寄存器	0x0084	只写
SDHOST_DBADDR_REG	链表基地址寄存器	0x0088	读/写
SDHOST_IDSTS_REG	IDMAC 状态寄存器	0x008C	读/写
SDHOST_IDINTEN_REG	IDMAC 中断使能寄存器	0x0090	读/写
SDHOST_DSCADDR_REG	主机描述符地址指针寄存器	0x0094	只读
SDHOST_BUFADDR_REG	主机缓冲区地址指针寄存器	0x0098	只读
SDHOST_BUFFIFO_REG	CPU 通过 FIFO 读写发送数据	0x0200	读/写
SDHOST_CLK_DIV_EDGE_REG	时钟相位选择寄存器	0x0800	读/写

17.13 寄存器

本小节的所有地址均为相对于 SD/MMC Host Controller 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

Register 17.1. SDHOST_CTRL_REG (0x0000)



SDHOST_CEATA_DEVICE_INTERRUPT_STATUS 软件应在 CE-ATA 设备上电复位或其他复位后写此位。CE-ATA 设备的中断通常被禁能 (nIEN = 1)。如果主机使能中断，则软件应将此位置为 1。(读/写)

SDHOST_SEND_AUTO_STOP_CCSD 始终将 SDHOST_SEND_AUTO_STOP_CCSD 和 SDHOST_SEND_CCSD 一起置位，不能分开单独置位。置位时，SD/MMC 自动发送内部生成的 STOP 命令 (CMD12) 给 CE-ATA 设备。发送 STOP 命令后，SDHOST_RINTSTS_REG 里的 Auto Command Done (ACD) 位被置为 1，如果 ACD 中断没有屏蔽，则用于主机的中断将会生成。发送 Command Completion Signal Disable (CCSD) 后，SD/MMC 自动清零 SDHOST_SEND_AUTO_STOP_CCSD 位。(读/写)

SDHOST_SEND_CCSD 置位时，SD/MMC 发送 CCSD 给 CE-ATA 设备。只有在当前命令等待 CCS 并且 CE-ATA 设备的中断使能时软件才会将此位置位。一旦 CCSD 模式发送给设备，SD/MMC 就会自动清零 SDHOST_SEND_CCSD 位。如果 ACD 中断没有屏蔽，则寄存器 SDHOST_RINTSTS_REG 里的 Command Done (CD) 位会被置为 1，且主机的中断将会生成。
说明：一旦 SDHOST_SEND_CCSD 被置位，需要两个卡时钟周期来驱动 CMD 线上的 CCSD。因此，即使设备已经发送 CCS 信号，在边界条件内 CCSD 信号可能会发送给 CE-ATA 设备。(读/写)

SDHOST_ABORT_READ_DATA 在读操作期间，挂起命令发送后，软件会轮询卡并找出挂起事件是从什么时候开始的。一旦挂起事件开始，软件会复位数据状态机，此时状态机正在等待下一个数据块。当数据状态机复位为空闲状态时，此位自动清零。(读/写)

SDHOST_SEND_IRQ_RESPONSE 响应发送后此位自动清零。为了等待 MMC 卡发送的中断，主机发送 CMD40，然后等待 MMC 卡的中断响应。同时，如果主机想要 SD/MMC 退出等待中断的状态，则会将此位置 1，此时 SD/MMC 命令状态机发送 CMD40 响应并返回空闲状态。(读/写)

见下页...

Register 17.1. SDHOST_CTRL_REG (0x0000)

接上页...

SDHOST_READ_WAIT 发送读操作等待给 SDIO 卡。(读/写)

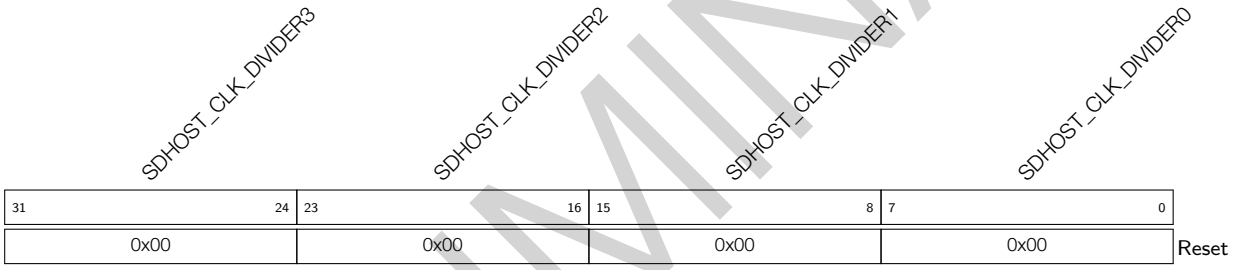
SDHOST_INT_ENABLE 全局中断使能/禁能位。0：禁能；1：使能。(读/写)

SDHOST_DMA_RESET 要复位 DMA 接口，软件应将此位置为 1。两个 AHB 时钟后此位自动清零。(读/写)

SDHOST_FIFO_RESET 要复位 FIFO，软件应将此位置为 1。复位操作结束后自动清零。
说明：清零后，再过两个系统时钟周期和同步延迟（两个卡时钟周期），FIFO 指针将会退出复位。(读/写)

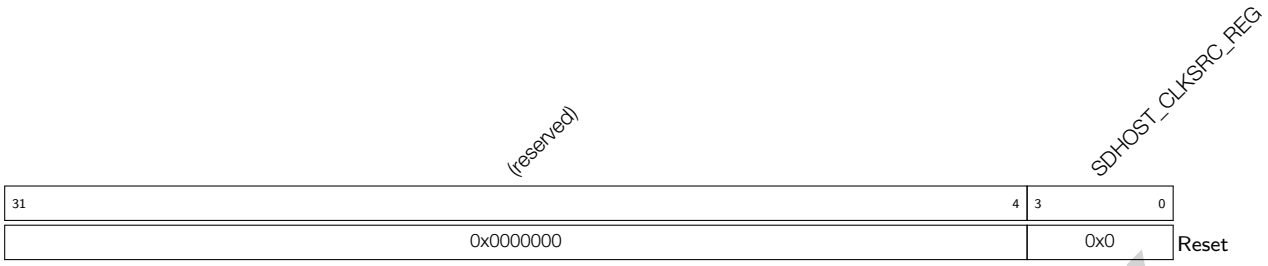
SDHOST_CONTROLLER_RESET 要复位控制器，软件应将此位置为 1。两个 AHB 时钟周期和两个 sdhost_cclk_in 时钟周期后此位自动清零。(读/写)

Register 17.2. SDHOST_CLKDIV_REG (0x0008)



SDHOST_CLK_DIVIDER*m* Clock divider-*m* 的值。时钟分频系数为 2**n*，*n*=0 旁路分频器（分频系数为 1）。例如，值为 1 代表分频系数为 2*1 = 2，值为 0xFF 代表分频系数为 2*255 = 510，以此类推。m 的取值范围为 0 至 3。(读/写)

Register 17.3. SDHOST_CLKSRC_REG (0x000C)



SDHOST_CLKSRC_REG 时钟分频源可以支持 2 个 SD 卡。每个卡分配有两个位。例如，bit[1:0] 分配给卡 0，bit[3:2] 分配给卡 1。卡 0 根据位值将时钟分频器 [0:3] 的输出信号传输给 cclk_out[1:0] 管脚。(读/写)

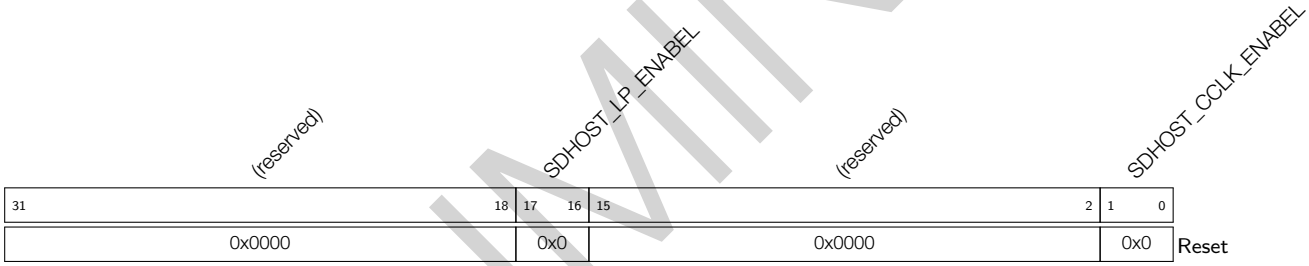
00：时钟分频器 0；

01：时钟分频器 1；

10：时钟分频器 2；

11：时钟分频器 3。

Register 17.4. SDHOST_CLKENA_REG (0x0010)



SDHOST_LP_ENABLE 当卡处于 IDLE 状态时，把时钟停掉。每个卡分配有一个位。(读/写)

0：时钟禁用；

1：时钟使能。

SDHOST_CCLK_ENABLE 时钟使能控制可支持 2 个 SD 卡时钟和一个 MMC 卡时钟。每个卡分配有一个位。(读/写)

0：时钟禁用；

1：时钟使能。

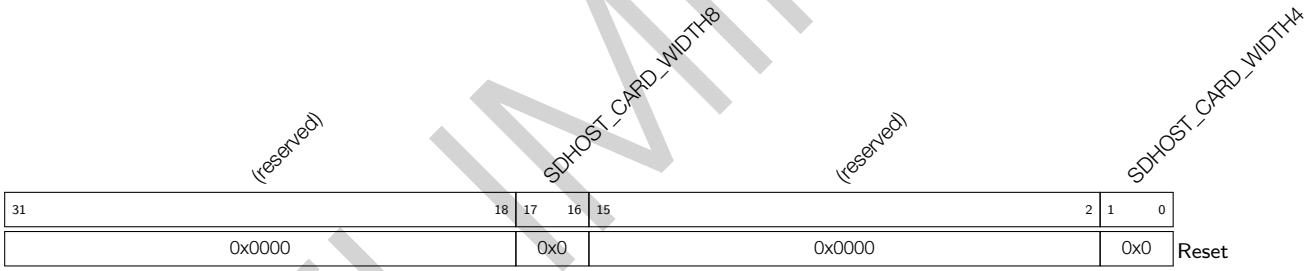
Register 17.5. SDHOST_TMOUT_REG (0x0014)

SDHOST_DATA_TIMEOUT 此位用于设置卡数据读取超时的值，还可以用来设置主机超时的定时器的值。只有当卡时钟停止后超时计数器才开始启动。此位的值以卡输出时钟数来表示，即被选取卡的 sdhost_cclk_out 时钟。(读/写)

说明：如果超时值是 100 ms 左右，则应该使用软件定时器。这种情况下，读数据超时中断应该被禁能。

SDHOST_RESPONSE_TIMEOUT 此位用于设置响应超时的值，以卡输出时钟数来表示，即 sdhost_cclk_out 时钟。(读/写)

Register 17.6. SDHOST_CTYPE_REG (0x0018)



SDHOST_CARD_WIDTH8 每个卡一个位，表明卡是否处于 8-bit 模式。(读/写)

0：非 8-bit 模式；

1：8-bit 模式。

Bit[17:16] 分别对应卡 [1:0]。

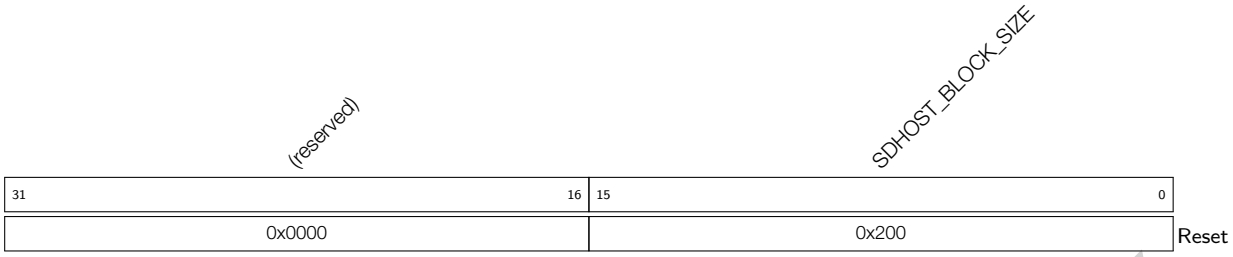
SDHOST_CARD_WIDTH4 每个卡一个位，表明卡处于 1-bit 模式还是 4-bit 模式。(读/写)

0：1-bit 模式；

1：4-bit 模式。

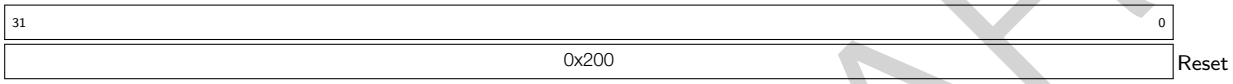
Bit[1:0] 分别对应卡 [1:0]。

Register 17.7. SDHOST_BLKSIZ_REG (0x001C)



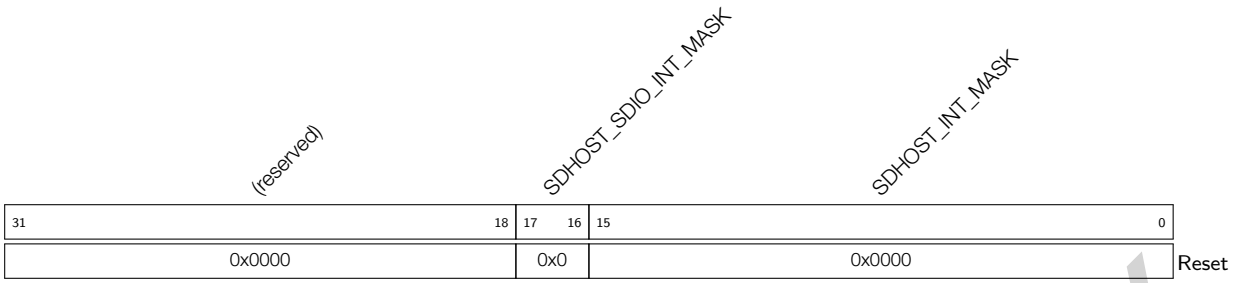
SDHOST_BLOCK_SIZE 块大小。(读/写)

Register 17.8. SDHOST_BYTCNT_REG (0x0020)



SDHOST_BYTCNT_REG 表明传输的字节数。对于块的传输，值应为块大小的整数倍。对于未定义字节长度的数据传输，字节计数应该设置为 0。当字节计数为 0 时，主机应当明确发送停止/终止命令来停止数据传输。(读/写)

Register 17.9. SDHOST_INTMASK_REG (0x0024)

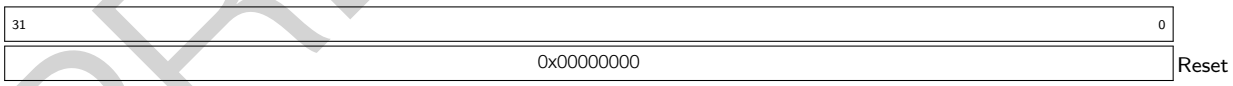


SDHOST_SDIO_INT_MASK SDIO 中断的屏蔽位，每个卡一个位。Bit[17:16] 分别对应卡 [1:0]。当被屏蔽时，SDIO 中断的检测被禁能。0 屏蔽中断，1 使能中断。(读/写)

SDHOST_INT_MASK 这些位用于屏蔽不想要的中断。0 屏蔽中断，1 使能中断。(读/写)

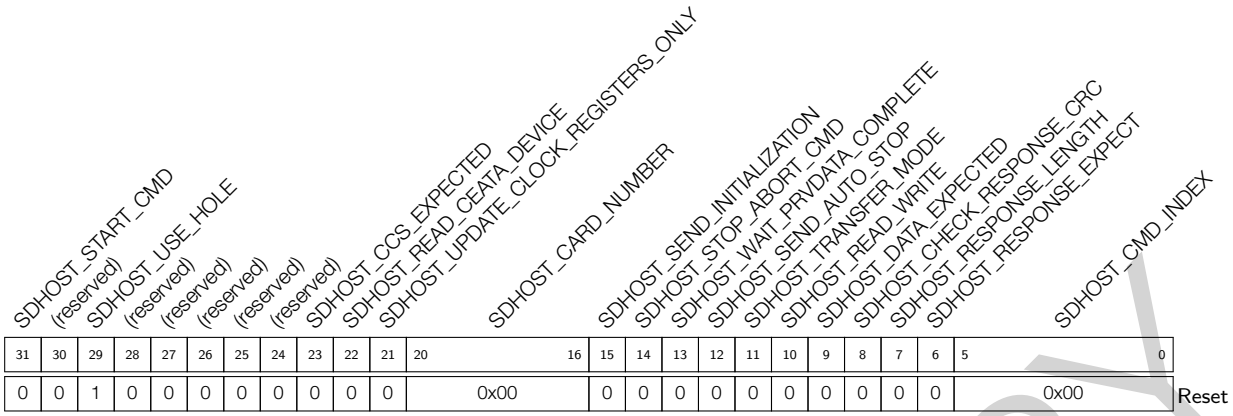
Bit 15 (EBE): 结束位错误/无 CRC 错误；
Bit 14 (ACD): 自动命令结束；
Bit 13 (SBE/BCI): 接收启动位错误；
Bit 12 (HLE): 硬件锁定写入错误
Bit 11 (FRUN): FIFO 空/满错误；
Bit 10 (HTO): 主机填充数据超时；
Bit 9 (DRTO): 数据读取超时；
Bit 8 (RTO): 响应超时；
Bit 7 (DCRC): 数据 CRC 错误；
Bit 6 (RCRC): 响应 CRC 错误；
Bit 5 (RXDR): 接收 FIFO 数据请求；
Bit 4 (TXDR): 发送 FIFO 数据请求；
Bit 3 (DTO): 数据传输结束；
Bit 2 (CD): 命令执行完毕；
Bit 1 (RE): 响应错误；
Bit 0 (CD): 卡检测。

Register 17.10. SDHOST_CMDARG_REG (0x0028)



SDHOST_CMDARG_REG 传递给卡的命令参数。(读/写)

Register 17.11. SDHOST_CMD_REG (0x002C)



SDHOST_START_CMD 开始发送命令。一旦 CIU 响应命令，此位自动清零。当此位置为 1 时，主机不应尝试写任何命令寄存器。如果尝试写寄存器，原始中断寄存器的硬件锁定错误位将会被置为 1。一旦命令被发送并且接收到 SD_MMC_CEATA 卡的响应，则原始中断寄存器的 Command Done 位将会被置为 1。(读/写)

SDHOST_USE_HOLE Use Hold 寄存器。(读/写)

0：发送给卡的 CMD 和 DATA 旁路 Hold 寄存器；

1：发送给卡的 CMD 和 DATA 经过 Hold 寄存器。

SDHOST_CCS_EXPECTED 预期命令完成信号 (CCS) 的配置。(读/写)

0：CE-ATA 设备的中断不使能 (ATA 控制寄存器中 nIEN = 1)；或者命令不等待设备端的 CCS 信号；

1：CE-ATA 设备的中断使能 (nIEN = 0)，并且 RW_BLK 命令等待 CE-ATA 设备的 CCS 信号。

如果命令等待 CE-ATA 设备的 CCS 信号，软件应该将此位置为 1。SD/MMC 置位 RINTSTS 寄存器里的 Data Transfer Over (DTO) 位并且如果 DTO 中断没有被屏蔽，会产生对主机的中断。

SDHOST_READ_CEATA_DEVICE 读操作标志位。(读/写)

0：主机不进行对于 CE-ATA 设备的读操作 (RW_REG 或 RW_BLK)；

1：主机进行对于 CE-ATA 设备的读操作 (RW_REG 或 RW_BLK)。

软件应将此位置为 1 来表明 CE-ATA 设备正在被访问用于读传输。此位用于在执行 CE-ATA 读传输时禁能读数据超时指示。I/O 传输延迟的最大值至少为 10 秒。SD/MMC 在等待来自 CE-ATA 设备的数据时不应指示读取数据超时。

见下页...

Register 17.11. SDHOST_CMD_REG (0x002C)

接上页...

SDHOST_UPDATE_CLOCK_REGISTERS_ONLY 0：正常命令序列；1：不发送命令，仅更新时钟寄存器的值到卡时钟域内。(读/写)

以下寄存器的值被传输到卡时钟域内：CLKDIV、CLRSRC 和 CLKENA。可改变卡时钟（改变时钟频率、设置是否截断、以及设置低频模式）。这是为了改变时钟频率或停止时钟，而不必发送命令给卡。

在正常命令序列下，当 SDHOST_UPDATE_CLOCK_REGISTERS_ONLY = 0，以下控制寄存器从 BIU 传输到 CIU：CMD、CMDARG、TMOUT、CTYPE、BLKSIZ 和 BYTCNT。CIU 为新的命令序列使用新的寄存器的值。当此位置为 1 时，由于没有命令被发送给 SD_MMC_CEATA 卡，所以没有 Command Done 中断。

SDHOST_CARD_NUMBER 使用中的卡号，表示正在访问的卡的物理插槽编号。在仅 SD 模式下，支持 2 张卡。(读/写)

SDHOST_SEND_INITIALIZATION 0：在发送命令前不发送初始化序列（80 个时钟周期的 1）；1：在发送命令前发送初始化序列。(读/写)

上电后，发送任何命令到卡之前，必须向卡发送 80 个时钟进行初始化。在向卡发送第一个命令时应该将此位置为 1，以便控制器在向卡发送命令之前初始化时钟。

SDHOST_STOP_ABORT_CMD 0：停止或中止命令，停止命令和中止命令都不会停止当前的数据传输。如果中止命令发送到当前选择的功能号或不在数据传输模式，则该位应设置为 0；1：停止或中止命令，用于停止当前的数据传输。(读/写)

当开放式预定义数据传输正在进行时，并且主机发出停止或中止命令以停止数据传输时，应将此位置为 1，以使 CIU 的命令/数据状态机可以正确返回到空闲状态。

SDHOST_WAIT_PRVDATA_COMPLETE 0：即使以前的数据传输尚未完成，也立即发送命令；1：等待前面的数据传输完成后再发送命令。(读/写)

SDHOST_WAIT_PRVDATA_COMPLETE = 0 选项通常用来在数据传输时询问卡的状态或停止当前的数据传输。SDHOST_CARD_NUMBER 应该与上一个命令相同。

SDHOST_SEND_AUTO_STOP 0：在数据传输结束时不发送停止命令；1：在数据传送结束时发送停止命令。(读/写)

见下页...

Register 17.11. SDHOST_CMD_REG (0x002C)

接上页...

SDHOST_TRANSFER_MODE 0：模块数据传输命令；1：流数据传输命令。(读/写)
如果不等待数据则为无关项。

SDHOST_READ_WRITE 0：读卡；1：写卡。(读/写)
如果不等待数据则为无关项。

SDHOST_DATA_EXPECTED 0：不等待数据传输；1：等待数据传输。(读/写)

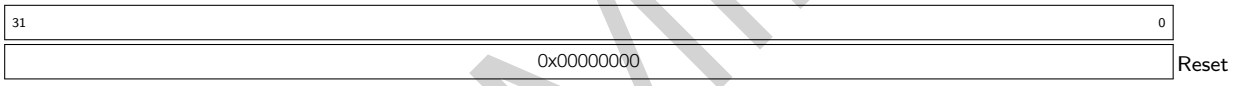
SDHOST_CHECK_RESPONSE_CRC 0：不检查；1：检查响应 CRC。(读/写)
有些命令响应不会返回有效的 CRC 位。软件应禁能对于这些命令的 CRC 检查以便禁能控制器进行 CRC 检查。

SDHOST_RESPONSE_LENGTH 0：等待卡的短响应；1：等待卡的长响应。(读/写)

SDHOST_RESPONSE_EXPECT 0：不等待卡的响应；1：等待卡的响应。(读/写)

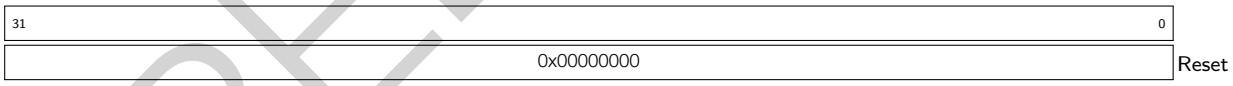
SDHOST_CMD_INDEX 命令指数。(读/写)

Register 17.12. SDHOST_RESP0_REG (0x0030)



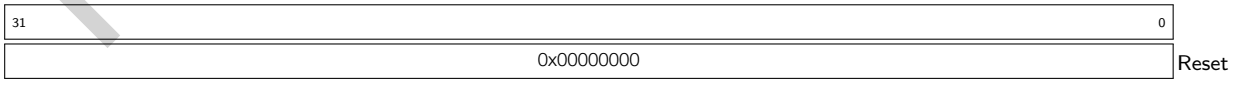
SDHOST_RESP0_REG 响应的 bit[31:0]。(只读)

Register 17.13. SDHOST_RESP1_REG (0x0034)



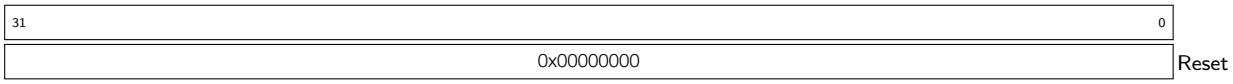
SDHOST_RESP1_REG 长响应的 bit[63:32]。(只读)

Register 17.14. SDHOST_RESP2_REG (0x0038)



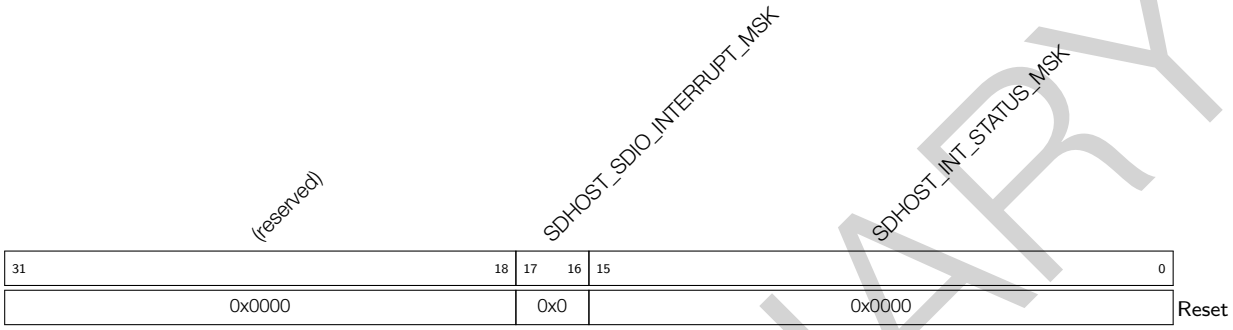
SDHOST_RESP2_REG 长响应的 bit[95:64]。(只读)

Register 17.15. SDHOST_RESP3_REG (0x003C)



SDHOST_RESP3_REG 长响应的 bit[127:96]。(只读)

Register 17.16. SDHOST_MINTSTS_REG (0x0040)

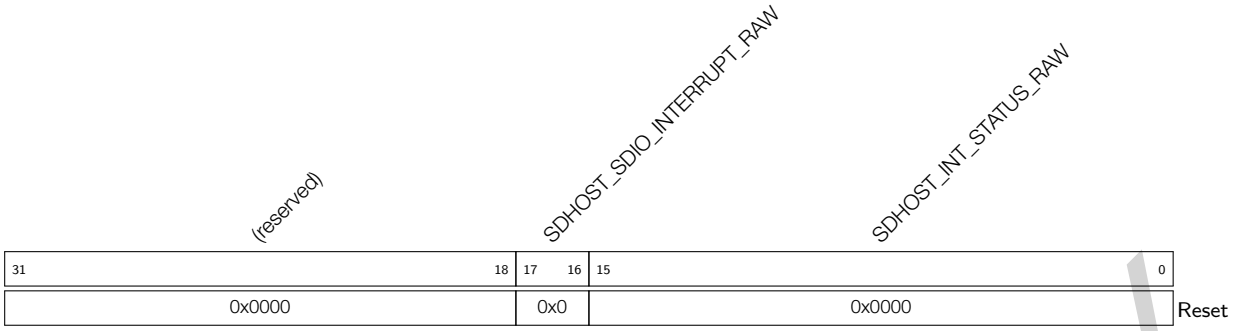


SDHOST_SDIO_INTERRUPT_MSK SDIO 中断的屏蔽位，每个卡占一个位。Bit[17:16] 分别对应卡 1 和卡 0。只有对应的 sdhost_sdio_int_mask 位被置为 1 时，SDIO 中断才会使能（置位屏蔽位使能中断）。(只读)

SDHOST_INT_STATUS_MSK 只有当中断屏蔽寄存器中的对应位被置为 1 时，中断才会使能。(只读)

Bit 15 (EBE): 结束位错误 / 无 CRC 错误;
Bit 14 (ACD): 自动命令结束;
Bit 13 (SBE/BCI): 接收启动位错误;
Bit 12 (HLE): 硬件锁定写入错误
Bit 11 (FRUN): FIFO 空/满错误;
Bit 10 (HTO): 主机填充数据超时;
Bit 9 (DRTO): 数据读取超时;
Bit 8 (RTO): 响应超时;
Bit 7 (DCRC): 数据 CRC 错误;
Bit 6 (RCRC): 响应 CRC 错误;
Bit 5 (RXDR): 接收 FIFO 数据请求;
Bit 4 (TXDR): 发送 FIFO 数据请求;
Bit 3 (DTO): 数据传输结束;
Bit 2 (CD): 命令执行完毕;
Bit 1 (RE): 响应错误;
Bit 0 (CD): 卡检测。

Register 17.17. SDHOST_RINTSTS_REG (0x0044)



SDHOST_SDIO_INTERRUPT_RAW 来自 SDIO 卡的中断，一个卡占一个位。Bit[17:16] 分别对应卡 1 和卡 0。置位某位就把相应的中断位清零，写 0 无效。(读/写)

0：没有来自卡的 SDIO 中断；

1：有来自卡的 SDIO 中断。

SDHOST_INT_STATUS_RAW 置位某位就把相应的中断位清零，写 0 无效。无论中断屏蔽状态如何，这些中断位都会被记录。(读/写)

Bit 15 (EBE): 结束位错误 / 无 CRC 错误；

Bit 14 (ACD): 自动命令结束；

Bit 13 (SBE/BCI): 接收启动为错误；

Bit 12 (HLE): 硬件锁定写入错误

Bit 11 (FRUN): FIFO 空/满错误；

Bit 10 (HTO): 主机填充数据超时；

Bit 9 (DRTO): 数据读取超时；

Bit 8 (RTO): 响应超时；

Bit 7 (DCRC): 数据 CRC 错误；

Bit 6 (RCRC): 响应 CRC 错误；

Bit 5 (RXDR): 接收 FIFO 数据请求；

Bit 4 (TXDR): 发送 FIFO 数据请求；

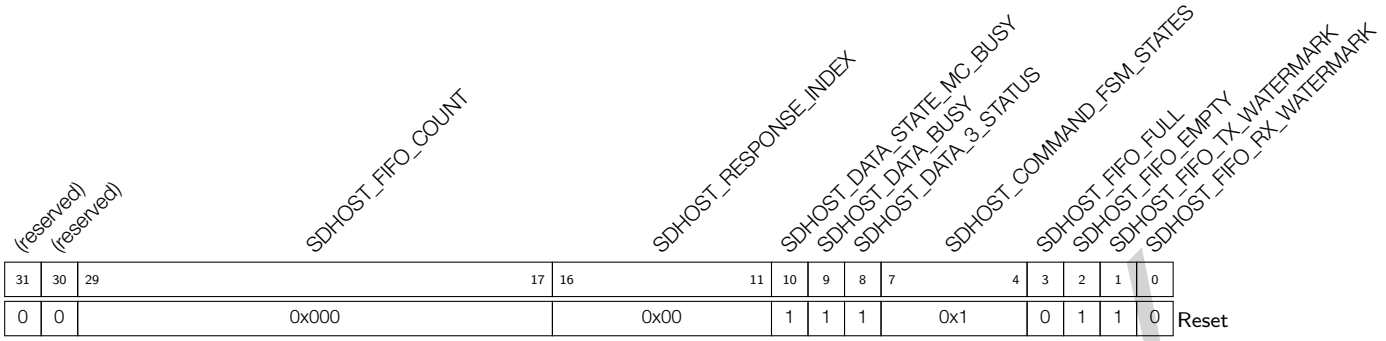
Bit 3 (DTO): 数据传输结束；

Bit 2 (CD): 命令执行完毕；

Bit 1 (RE): 响应错误；

Bit 0 (CD): 卡检测。

Register 17.18. SDHOST_STATUS_REG (0x0048)



SDHOST_FIFO_COUNT FIFO 计数器，FIFO 中被填充的地址的数量。(只读)

SDHOST_RESPONSE_INDEX 前一个响应的指数，包括内核发送的任何自动停止的响应。(只读)

SDHOST_DATA_STATE_MC_BUSY 数据发送或接收状态机忙。(只读)

SDHOST_DATA_BUSY 数据线 sdhost_card_data[0] 的值取反。(只读)

0：卡数据不忙；
1：卡数据忙。

SDHOST_DATA_3_STATUS 数据线 sdhost_card_data[3] 上的值，检查卡是否存在。(只读)

0：卡不存在；
1：卡存在。

SDHOST_COMMAND_FSM_STATES 命令状态机状态。(只读)

0: 空闲；
1: 发送初始序列；
2: 发送命令开始位；
3: 发送命令发送位；
4: 发送命令指数和参数；
5: 发送命令 CRC7；
6: 发送命令结束位；
7: 接收响应开始位；
8: 接收响应 IRQ 响应；
9: 接收响应发送位；
10: 接收响应命令指数；
11: 接收响应数据；
12: 接收响应 CRC7；
13: 接收响应结束位；
14: 命令路径等待 NCC；
15: 等待，命令-响应回转。

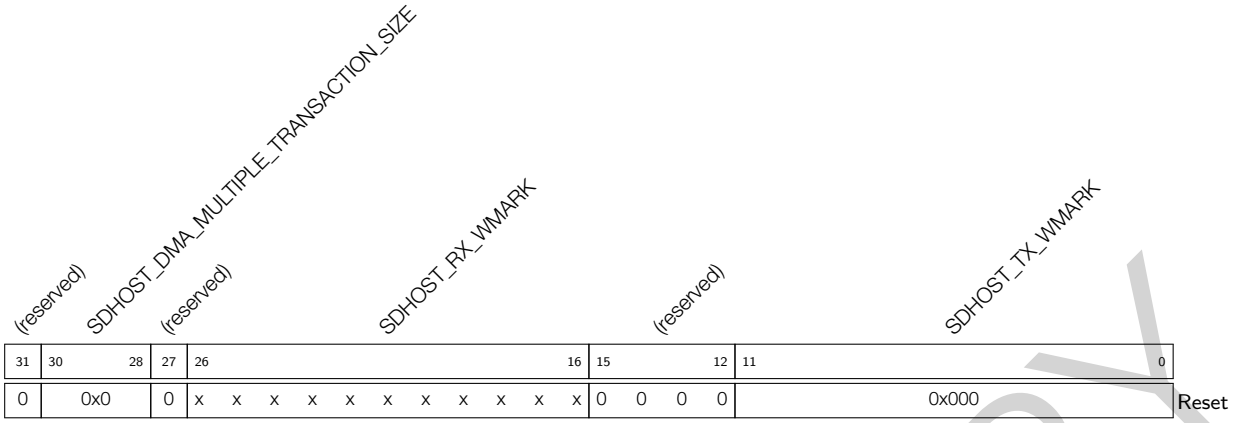
SDHOST_FIFO_FULL FIFO 为满的状态。(只读)

SDHOST_FIFO_EMPTY FIFO 为空的状态。(只读)

SDHOST_FIFO_TX_WATERMARK FIFO 达到发送阈值，不是数据传输的必要条件。(只读)

SDHOST_FIFO_RX_WATERMARK FIFO 达到接收阈值，不是数据传输的必要条件。(只读)

Register 17.19. SDHOST_FIFOTH_REG (0x004C)

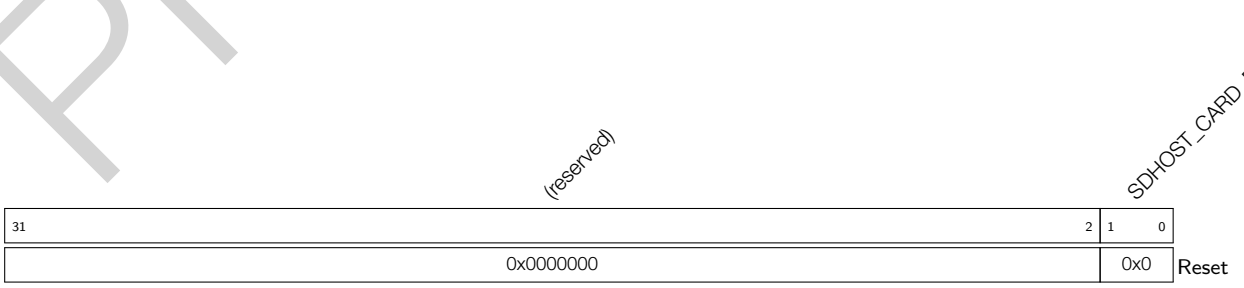


SDHOST_DMA_MULTIPLE_TRANSACTION_SIZE 突发传输的字节数，配置的值应与 DMA 控制器多个传输大小 SDHOST_SRC/DEST_MSIZE 相同。000: 1 字节传输；001: 4 字节传输；010: 8 字节传输；011: 16 字节传输；100: 32 字节传输；101: 64 字节传输；110: 128 字节传输；111: 256 字节传输。(读/写)

SDHOST_RX_WMARK 当接收数据给卡时 FIFO 的阈值。当 FIFO 数据计数大于该数值时，DMA/FIFO 请求被提出。在数据包结束期间，无论阈值大小如何，都会生成请求，以完成剩余的数据传输。在非 DMA 模式下，当接收 FIFO 阈值 (RXDR) 中断使能时，则会产生中断，而不是 DMA 请求。如果阈值大于任何剩余数据，则不产生中断。当主机看见数据发送结束中断时，主机应该读取剩下的数据。在 DMA 模式下，在数据包结束时，即使剩余的字节数少于阈值，DMA 请求也会在数据传输结束中断设置之前进行单次传输以清除所有剩余的字节。(读/写)

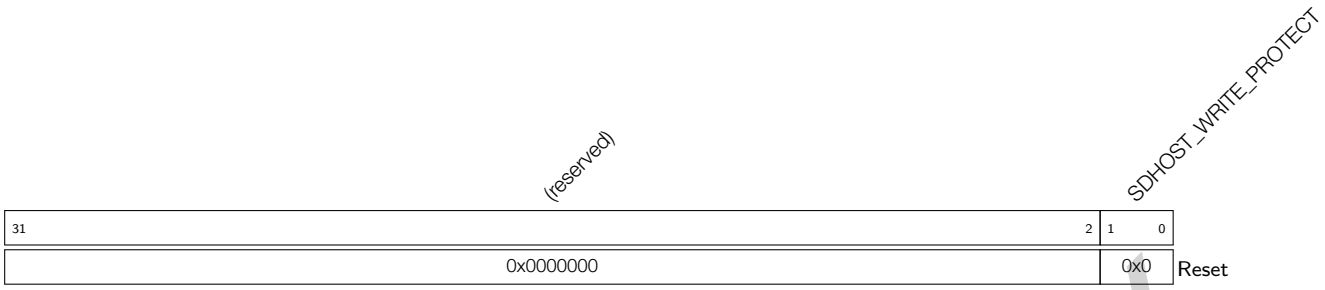
SDHOST_TX_WMARK 当发送数据给卡时 FIFO 的阈值。当 FIFO 数据计数小于等于该数值时，DMA/FIFO 请求被提出。如果使能中断，则中断发生。在数据包结束期间，无论阈值大小如何，都会生成请求。在非 DMA 模式下，当发送 FIFO 阈值 (TXDR) 中断使能时，则会产生中断，而不是 DMA 请求。在数据包结束期间，在最后一个中断时，主机负责仅用所需的剩余字节填充 FIFO（不是在 FIFO 满之前或在 CIU 完成数据传输之后，因为 FIFO 可能不为空）。在 DMA 模式下，在数据包结束时，如果最后一次传输比突发传输小，DMA 控制器将执行单个周期，直到传输所需的字节。(读/写)

Register 17.20. SDHOST_CDETECT_REG (0x0050)



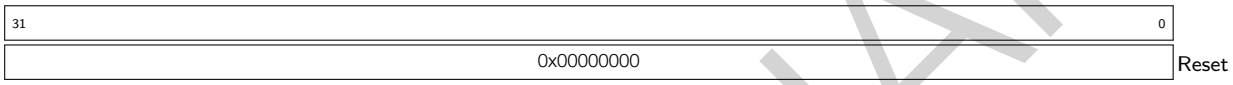
SDHOST_CARD_DETECT_N 信号线 sdhost_card_detect_n 输入端口（每个卡一个位）的值。0 代表卡存在。只有相应位被执行。(只读)

Register 17.21. SDHOST_WRTPRT_REG (0x0054)



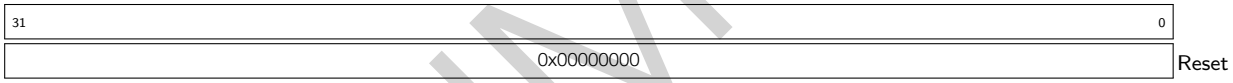
SDHOST_WRITE_PROTECT 信号线 sdhost_card_write_prt 输入端口（每个卡一个位）的值。1 表示写保护。只有的相应位被执行。（只读）

Register 17.22. SDHOST_TCBCNT_REG (0x005C)



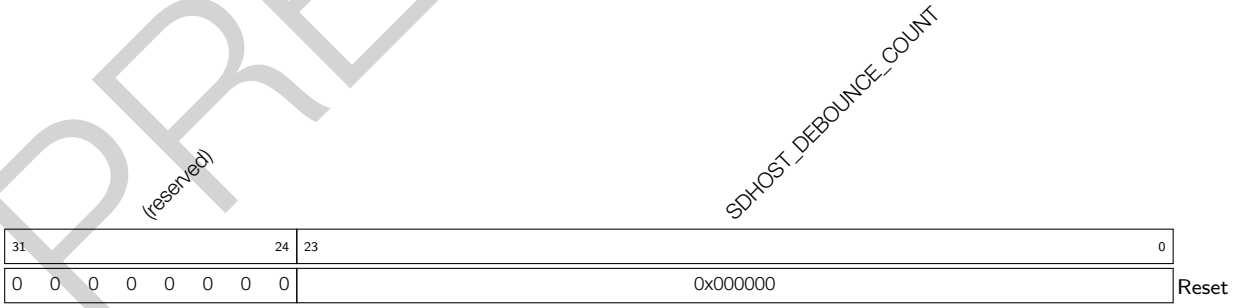
SDHOST_TCBCNT_REG CIU 模块以及传输给卡的字节数。（只读）

Register 17.23. SDHOST_TBBCNT_REG (0x0060)



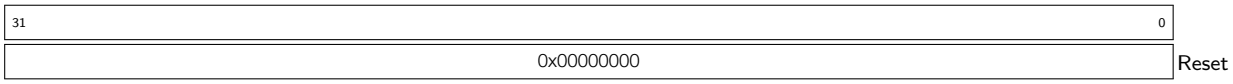
SDHOST_TBBCNT_REG 主机/DMA 和 BIU FIFO 之间传输的字节数。（只读）

Register 17.24. SDHOST_DEBNCE_REG (0x0064)



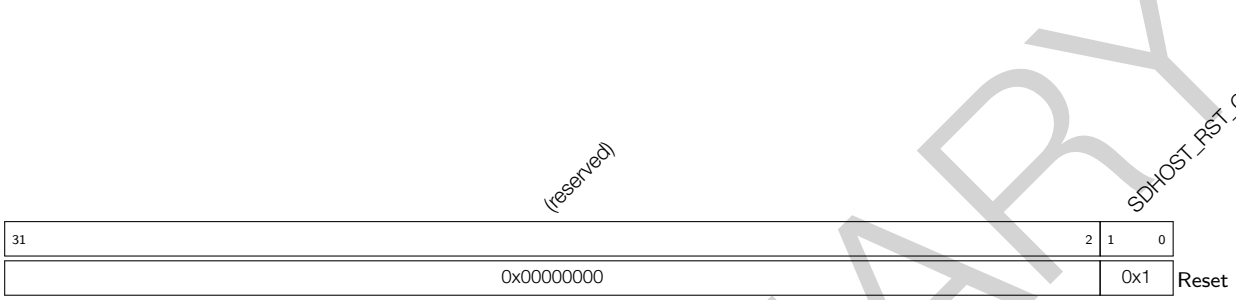
SDHOST_DEBOUNCE_COUNT 抖动消除滤波器逻辑使用的主机时钟数。典型的去抖动时间为 5 ~ 25 ms，防止插卡或移除卡的时候的不稳定性。（读/写）

Register 17.25. SDHOST_USRID_REG (0x0068)



SDHOST_USRID_REG 用户识别寄存器。此寄存器也可以作为用户的寄存器使用。(读/写)

Register 17.26. SDHOST_RST_N_REG (0x0078)



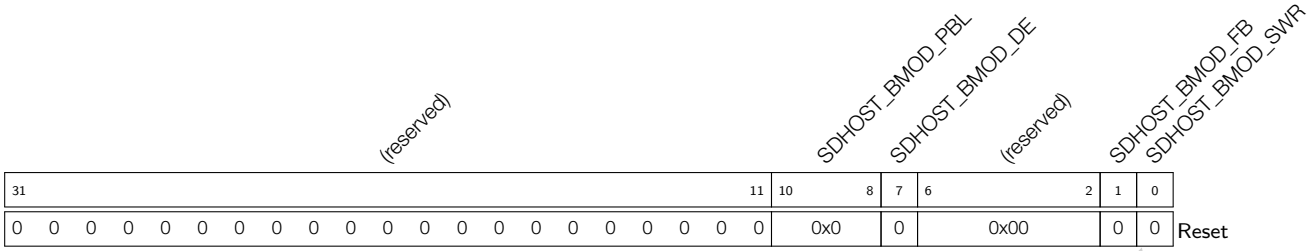
SDHOST_RST_CARD_RESET 硬件复位。(读/写)

1：工作模式；

0：复位。

这些位让卡进入空闲状态，主机工作时需要被重新初始化。SDHOST_RST_CARD_RESET[0] 应被置为 1'b0 来复位卡 0。SDHOST_RST_CARD_RESET[1] 应被置为 1'b0 来复位卡 1。

Register 17.27. SDHOST_BMOD_REG (0x0080)



SDHOST_BMOD_PBL 可编程突发长度。这些位指示在一个 IDMAC（Internal DMA Control）传输中要执行的最大节拍数。IDMAC 每次在主机总线上开始突发传输时将总是尝试在 PBL 中指定的突发值。允许的值为 1、4、8、16、32、64、128 和 256。该值是 FIFOTH 寄存器的 MSIZE 的镜像。要修改此值，将所需的值写入 FIFOTH 寄存器。这是一个编码值，如下所示：（只读）

000: 1 字节传输；
001: 4 字节传输；
010: 8 字节传输；
011: 16 字节传输；
100: 32 字节传输；
101: 64 字节传输；
110: 128 字节传输；
111: 256 字节传输。

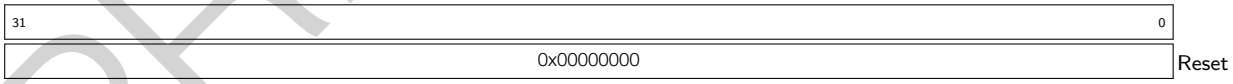
PBL 是只读值，并且仅适用于数据访问，不适用于链表访问。

SDHOST_BMOD_DE IDMAC 使能位。置位后 IDMAC 使能。（只读）

SDHOST_BMOD_FB 固定突发。控制 AHB 主接口是否执行固定突发传输。置位时，AHB 将在正常突发传输开始期间仅使用 SINGLE、INCR4、INCR8 或 INCR16。当复位时，AHB 将使用 SINGLE 和 INCR 突发传输操作。（读/写）

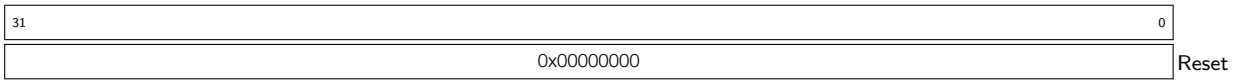
SDHOST_BMOD_SWR 软件复位。当置位时，DMA 控制器复位所有内部寄存器。一个时钟周期后自动清零。（读/写）

Register 17.28. SDHOST_PLDMND_REG (0x0080)



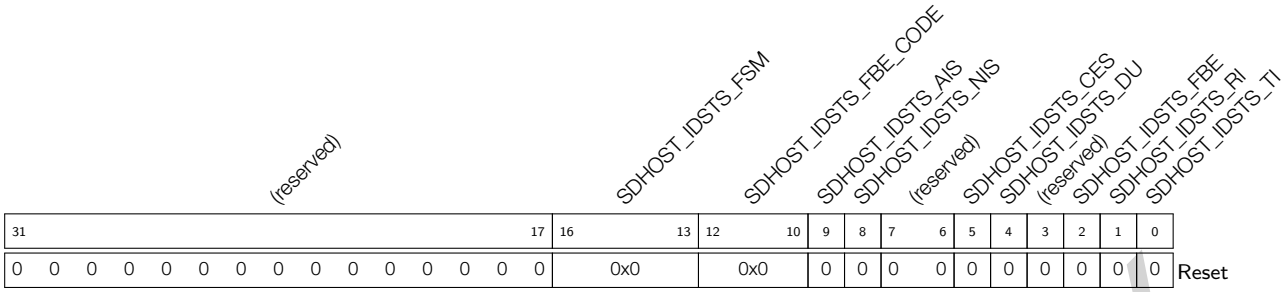
SDHOST_PLDMND_REG 轮询需求。如果链表的 OWNER 位未置位，则状态机进入挂起状态。主机需要对这个寄存器写入任意值，以使 IDMAC 状态机恢复正常读取链表的操作。这是一个只写寄存器。（只写）

Register 17.29. SDHOST_DBADDR_REG (0x0088)



SDHOST_DBADDR_REG 链表的开始。包含第一个链表的基址。最低效位 (LSB bit) [1:0] 被忽略，并由 IDMAC 内部全部取为零，因此这些 LSB 位可被视为只读。（读/写）

Register 17.30. SDHOST_IDSTS_REG (0x008C)



SDHOST_IDSTS_FSM DMAC 状态机当前状态。(只读)

0: DMA_IDLE (状态机空闲状态);

1: DMA_SUSPEND (状态机暂停状态);

2: DESC_RD (读链表状态);

3: DESC_CHK (检查链表状态);

4: DMA_RD_REQ_WAIT (状态机读数据请求等待状态);

5: DMA_WR_REQ_WAIT (状态机写数据请求等待状态);

6: DMA_RD (状态机读数据状态);

7: DMA_WR (状态机写数据状态);

8: DESC_CLOSE (当前链表使用完关闭状态)。

SDHOST_IDSTS_FBE_CODE 致命总线错误代码。表明导致总线错误的错误类型。仅当设置致命总线错误位 IDSTS[2] 被置位时有效。此字段不生成中断。(只读)

001: 发送期间接收到主机中止发送;

010: 接收期间接收到主机中止接收;

其他: 保留。

SDHOST_IDSTS_AIS 异常中断汇总。以下各项的逻辑或: (读/写)

IDSTS[2]: 致命总线中断;

IDSTS[4]: SDHOST_IDSTS_DU 位中断。

只有未屏蔽的位影响该位。这是一个粘滞位，必须在每次清零可以引起 AIS 置 1 的相应位时清零。写 1 清零该位。

SDHOST_IDSTS_NIS 正常中断汇总。以下各项的逻辑或: (读/写)

IDSTS[0]: 发送中断;

IDSTS[1]: 接收中断。

只有未屏蔽的位影响该位。这是一个粘滞位，必须在每次清零可以引起 NIS 置 1 的相应位时清零。写 1 清零该位。

见下页...

Register 17.30. SDHOST_IDSTS_REG (0x008C)

接上页...

SDHOST_IDSTS_CES 卡错误汇总。指示发送/接收卡的传输状态，也出现在 RINTSTS 中。表示以下位的逻辑或：（读/写）

EBE：结束位错误；

RTO：响应超时/引导确认超时错误；

RCRC：响应 CRC 错误；

SBE：启动位错误；

DRTO：数据读取超时/ BDS 超时错误；

DCRC：接收的数据 CRC 错误；

RE：响应错误。

写 1 清零该位。IDMAC 的中止条件取决于此 CES 位的配置。如果 CES 位被使能，则 IDMAC 在响应错误时中止。

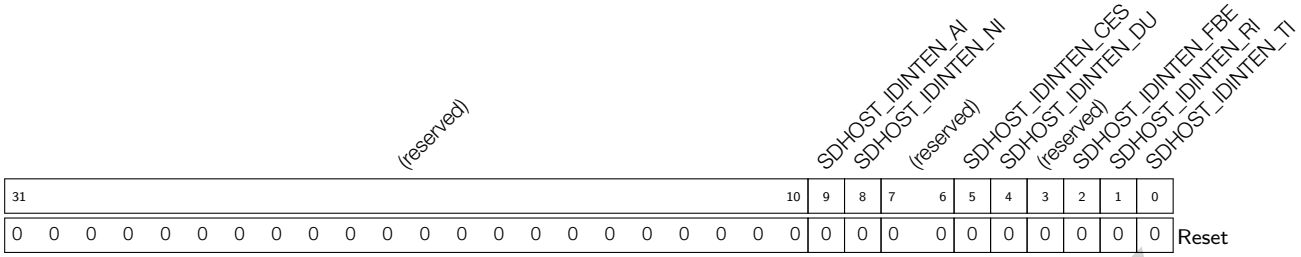
SDHOST_IDSTS_DU 链表不可用中断。当链表由于 OWNER 位 = 0 (DES0 [31] = 0) 而不可用时，该位置 1。写 1 清零该位。（读/写）

SDHOST_IDSTS_FBE 致命总线错误中断。表示发生总线错误 (IDSTS[12:10])。当该位置 1 时，DMA 禁止所有总线访问。写 1 清零该位。（读/写）

SDHOST_IDSTS_RI 接收中断。表示链表的数据接收完成。写 1 清零该位。（读/写）

SDHOST_IDSTS_TI 发送中断。表示链表的数据发送完成。写 1 清零该位。（读/写）

Register 17.31. SDHOST_IDINTEN_REG (0x0090)



SDHOST_IDINTEN_AI 异常中断汇总使能位。置 1 时，使能异常中断。该位使能以下位：（读/写）

IDINTEN [2]：致命总线错误中断；

IDINTEN [4]：DU 中断。

SDHOST_IDINTEN_NI 正常中断汇总使能位。置 1 时，使能正常中断。重置时，禁能正常中断。该位使能以下位：（读/写）

IDINTEN[0]：发送中断；

IDINTEN[1]：接收中断。

SDHOST_IDINTEN_CES 卡错误汇总中断使能位。置 1 时使能卡中断汇总。（读/写）

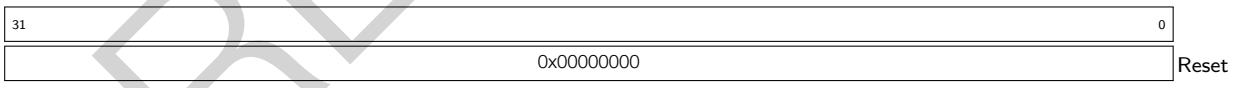
SDHOST_IDINTEN_DU 链表不可用中断。当与异常中断汇总使能位一起设置时，将使能 DU 中断。（读/写）

SDHOST_IDINTEN_FBE 致命总线错误使能位。当与异常中断汇总使能位一起设置时，将使能致命总线错误中断。复位时，致命总线错误使能中断被禁能。（读/写）

SDHOST_IDINTEN_RI 接收中断使能位。当与正常中断汇总使能位一起设置时，将使能接收中断。复位时，接收中断被禁能。（读/写）

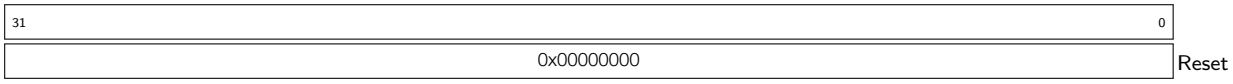
SDHOST_IDINTEN_TI 发送中断使能位。当与正常中断汇总使能位一起设置时，将使能发送中断。复位时，发送中断被禁能。（读/写）

Register 17.32. SDHOST_DSCADDR_REG (0x0094)



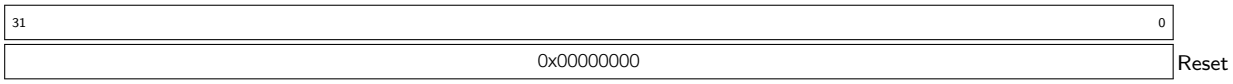
SDHOST_DSCADDR_REG 主机链表地址指针，在操作期间由 IDMAC 更新，并在复位时清零。该寄存器指向由 IDMAC 读取的当前链表的起始地址。（只读）

Register 17.33. SDHOST_BUFADDR_REG (0x0098)



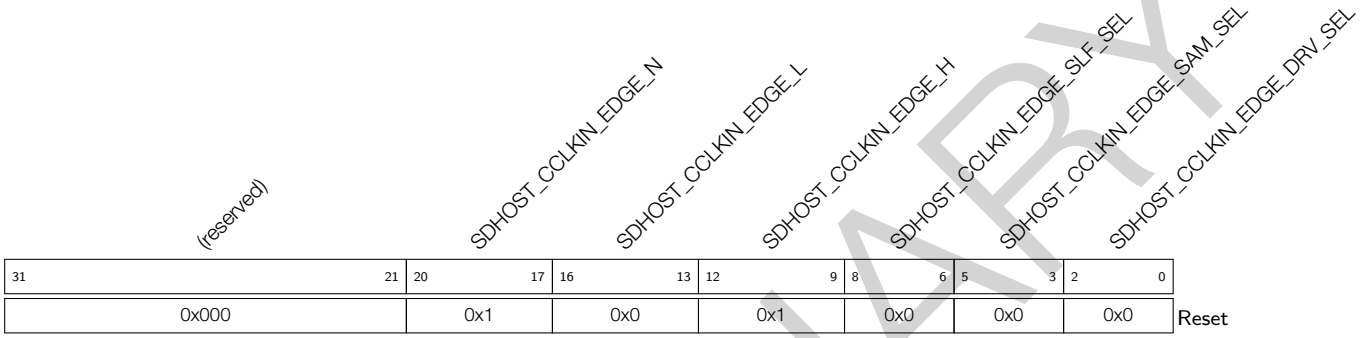
SDHOST_BUFADDR_REG 主机缓冲区地址指针，在操作期间由 IDMAC 更新，并在复位时清零。该寄存器指向由 IDMAC 访问的当前数据缓冲区地址。（只读）

Register 17.34. SDHOST_BUFFIFO_REG (0x0200)



SDHOST_BUFFIFO_REG CPU 通过 FIFO 读写发送的数据。该寄存器指向当前数据 FIFO。(只读)

Register 17.35. SDHOST_CLK_DIV_EDGE_REG (0x0800)



CCLKIN_EDGE_N 值应与 CCLKIN_EDGE_L 相同。(读/写)

CCLKIN_EDGE_L 分频时钟的低电平，值应比 CCLKIN_EDGE_H 大。(读/写)

CCLKIN_EDGE_H 分频时钟的高电平，值应比 CCLKIN_EDGE_L 小。(读/写)

CCLKIN_EDGE_SLF_SEL 用于选择内部时钟信号的相位，90 度相位、180 度相位或 270 度相位。(读/写)

CCLKIN_EDGE_SAM_SEL 用于选择输入时钟信号的相位，90 度相位、180 度相位或 270 度相位。(读/写)

CCLKIN_EDGE_DRV_SEL 用于选择输出时钟信号的相位，90 度相位、180 度相位或 270 度相位。(读/写)

说明：SD/MMC 使用该寄存器分频 160M 时钟 (CCLKIN_EDGE_H/CCLKIN_EDGE_L)。输出时钟通过该寄存器和寄存器 SDHOST_CLKDIV_REG 连接 SDIO 从机分频器，使用 SDHOST_CLKSRC_REG 时可选择 4 个时钟源。

18 LED PWM 控制器 (LEDC)

LED PWM 控制器用于生成控制 LED 的脉冲宽度调制信号 (PWM)，具有占空比自动渐变等专门功能。该外设也可成生成 PWM 信号用作其他用途。

18.1 主要特性

LED PWM 控制器具有如下特性:

- 八个独立的 PWM 生成器（即八个通道）
- 四个独立定时器，可实现小数分频
- 占空比自动渐变（即 PWM 信号占空比可逐渐增加或减小，无须处理器干预），渐变完成时产生中断
- 输出 PWM 信号相位可调
- 低功耗模式 (Light-sleep mode) 下可输出 PWM 信号
- PWM 最大精度为 14 位

四个定时器具有相同的功能和运行方式，下文将四个定时器统称为定时器*x*（*x* 的范围是 0 到 3）。八个 PWM 生成器的功能和运行方式也相同，下文将统称为 PWM *n*（n 的范围是 0 到 7）。

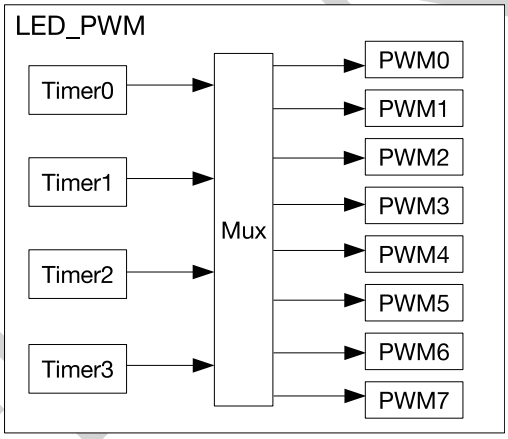


图 18-1. LED PWM 控制器架构

18.2 功能描述

18.2.1 架构

图 18-1为 LED PWM 控制器的架构。四个定时器可独立配置（时钟分频器和计数器最大值），每个定时器内部有一个时基计数器（即基于基准时钟周期计数的计数器）。每个 PWM 生成器会在四个定时器中择一，以该定时器的计数值为基准生成 PWM 信号。

图 18-2 为定时器和 PWM 生成器的主要功能块。

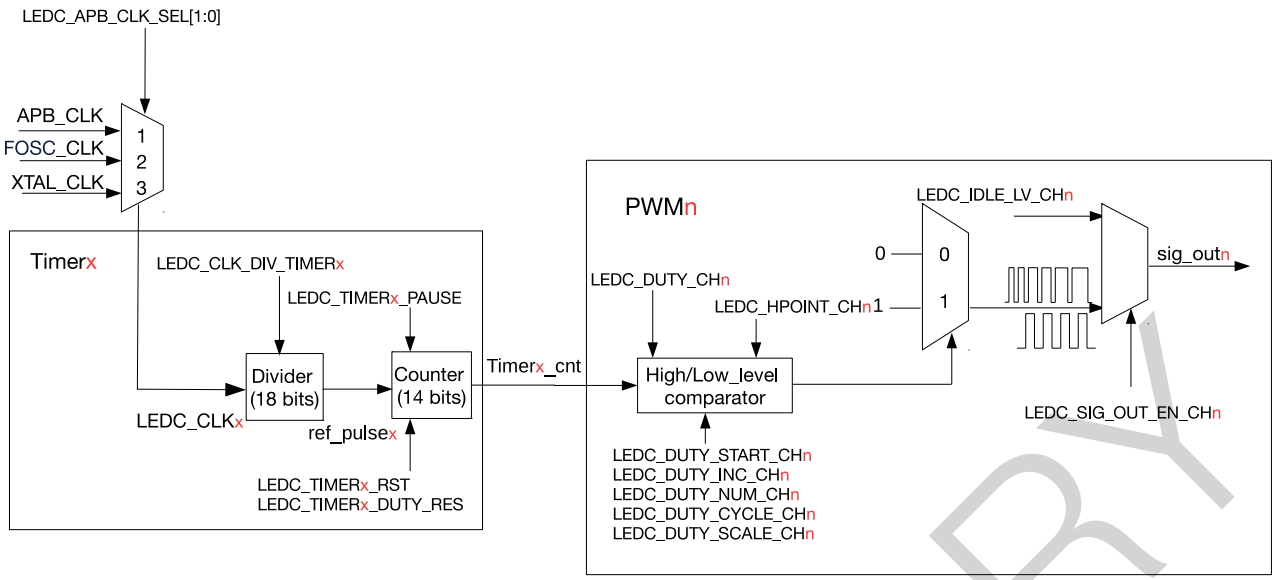


图 18-2. 定时器和 PWM 生成器功能块

18.2.2 定时器

LED PWM 控制器的每个定时器内部都有一个时基计数器，该计数器在时钟信号的每个周期加 1。图 18-2 中时基计数器使用的时钟信号称为 ref_pulse*x*。所有定时器使用同一个时钟源信号（LEDC_CLK*x*），该时钟源信号经分频器分频后产生 ref_pulse*x* 供计数器使用。

18.2.2.1 时钟源

软件配置 LED PWM 寄存器由 APB_CLK 驱动。更多关于 APB_CLK 的信息，详见章节 3 *复位和时钟*。要使用 LED PWM 控制器，需使能 LED PWM 的 APB_CLK 时钟信号，该时钟信号可通过置位 SYSTEM_PERIP_CLK_EN0_REG 寄存器的 SYSTEM_LEDC_CLK_EN 使能，通过软件置位 SYSTEM_PERIP_RST_EN0_REG 寄存器的 SYSTEM_LEDC_RST 位复位。更多信息，请参阅章节 21 *系统寄存器 (TBA)* 的表 21

LED PWM 控制器的定时器有三个时钟源信号可以选择：APB_CLK、FOSC_CLK 和 XTAL_CLK（更多有关时钟源的信息详见章节 3 *复位和时钟*）。为 LEDC_CLK*x* 选择时钟源信号的配置如下：

- APB_CLK：将 LEDC_APB_CLK_SEL[1:0] 置 1
- FOSC_CLK：将 LEDC_APB_CLK_SEL[1:0] 置 2
- XTAL_CLK：将 LEDC_APB_CLK_SEL[1:0] 置 3

之后，LEDC_CLK*x* 信号会进入时钟分频器。

18.2.2.2 时钟分频器配置

LEDC_CLK*x* 信号传输到时钟分频器，产生 ref_pulse*x* 信号供计数器使用。ref_pulse*x* 的频率等于 LEDC_CLK*x* 的频率经 LEDC_CLK_DIV_TIMER*x*分频系数分频后的结果（见图 18-2）。

LEDC_CLK_DIV_TIMER*x* 分频系数为小数分频，因此其值可为非整数。分频系数 LEDC_CLK_DIV_TIMER*x*可根据下列等式通过 LEDC_CLK_DIV_TIMER*x* 字段配置：

$$LEDC\_CLK\_DIV\_TIMERx = A + \frac{B}{256}$$

- 整数部分 *A* 为 LEDC_CLK_DIV_TIMER*x* 字段的高 10 位（即 LEDC_TIMER*x*_CONF_REG[21:12]）

- 小数部分 *B* 为 LEDC_CLK_DIV_TIMER*x* 字段的低 8 位（即 LEDC_TIMER*x*_CONF_REG[11:4]）

小数部分 *B* 为 0 时，LEDC_CLK_DIV_TIMER*x*的值为整数（整数分频）。也就是说，每 *A* 个 LEDC_CLK*x* 时钟周期产生一个 ref_pulse*x* 时钟脉冲。

小数部分 *B* 不为 0 时，LEDC_CLK_DIV_TIMER*x*的值非整数。时钟分频器按照 *A* 个 LEDC_CLK*x* 时钟周期和 (*A*+1) 个 LEDC_CLK*x* 时钟周期轮流进行非整数分频。这样一来，ref_pulse*x* 时钟脉冲的平均频率便会是理想值（非整数分频的频率）。每 256 个 ref_pulse*x* 时钟脉冲中：

- 有 *B* 个以 (*A*+1) 个 LEDC_CLK*x* 时钟周期分频
- 有 (256-*B*) 个以 *A* 个 LEDC_CLK*x* 时钟周期分频
- 以 (*A*+1) 个 LEDC_CLK*x* 时钟周期分频的时钟脉冲均匀分布在以 *A* 分频的时钟脉冲中

图 18-3 展示了 LEDC_CLK_DIV_TIMER*x*分频系数非整数时，LEDC_CLK*x* 时钟周期和 ref_pulse*x* 时钟脉冲的关系。

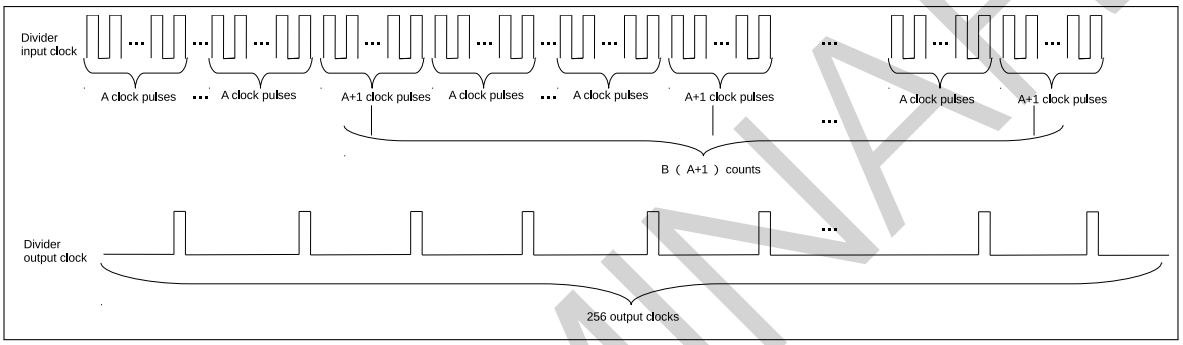


图 18-3. LEDC_CLK_DIV_TIMER*x*非整数时的分频

重新设置分频器的分频系数最大值，需先置位 LEDC_CLK_DIV_TIMER*x* 字段，然后置位 LEDC_TIMER*x*_PARA_UP 字段应用新配置。新配置会在计数器下次溢出时生效。LEDC_TIMER*x*_PARA_UP 字段由硬件自动清除。

18.2.2.3 14 位计数器

每个定时器有一个以 ref_pulse*x* 为基准时钟的 14 位时基计数器（见图 18-2）。LEDC_TIMER*x*_DUTY_RES 字段用于配置 14 位计数器的最大值。因此，PWM 信号的最大精确度为 14 位。计数器最大可计数至 $2^{LEDC\_TIMERx\_DUTY\_RES}$ − 1，然后溢出重新从 0 开始计数。软件可以读取、复位、暂停计数器。

计数器可在每次溢出时触发 (LEDC_TIMER*x*_OVF_INT) 中断，这个中断为硬件自动产生，不需要配置。计数器也可配置为在溢出 $(LEDC\_OVF\_NUM\_CHn + 1)$ 次时触发 LEDC_OVF_CNT_CH*n*_INT 中断，该中断配置步骤如下：

1. 配置LEDC_TIMER_SEL_CH*n*为 PWM 生成器选择该计数器
2. 置位 LEDC_OVF_CNT_EN_CH*n* 使能计数器
3. 把 LEDC_OVF_NUM_CH*n* 的值设为计数器触发中断的溢出次数减 1
4. 置位 LEDC_OVF_CNT_CH*n*_INT_ENA 使能溢出中断
5. 置位 LEDC_TIMER*x*_DUTY_RES 使能定时器，等待 LEDC_OVF_CNT_CH*n*_INT 中断产生

如图 18-2 所示，PWM 生成器输出信号 sig_out*n* 的频率取决于定时器的时钟源 LEDC_CLK*x*、分频器的分频系数 LEDC_CLK_DIV_TIMER*x*以及计数器的计数范围 LEDC_TIMER*x*_DUTY_RES：

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

在运行时改变计数器的最大值，需先置位 LEDC_TIMERx_DUTY_RES 字段，然后置位 LEDC_TIMERx_PARA_UP 字段。新的配置在计数器下一次溢出时生效。如果重新配置 LEDC_OVF_CNT_EN_CHn 字段，也需置位 LEDC_PARA_UP_CHn 应用新配置。总之，更改配置时都需置位 LEDC_PARA_UP_CHn 应用新配置。LEDC_TIMERx_PARA_UP 字段由硬件自动清除。

18.2.3 PWM 生成器

要生成 PWM 信号，PWM 生成器 (PWMn) 需选择一个定时器 (Timerx)。每个 PWM 生成器均可通过置位 LEDC_TIMER_SEL_CHn 单独配置，在四个定时器中选择一个输出 PWM 信号。

如图 18-2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 生成器将定时器的 14 位计数值 (Timerx_cnt) 与高低电平比较器的值 Hpointn 和 Lpointn 比较。如果定时器的计数值等于 Hpointn 或 Lpointn，PWM 信号可以输出高低电平：

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

图 18-4 展示了如何使用 Hpointn 和 Lpointn 生成占空比固定的 PWM 信号。

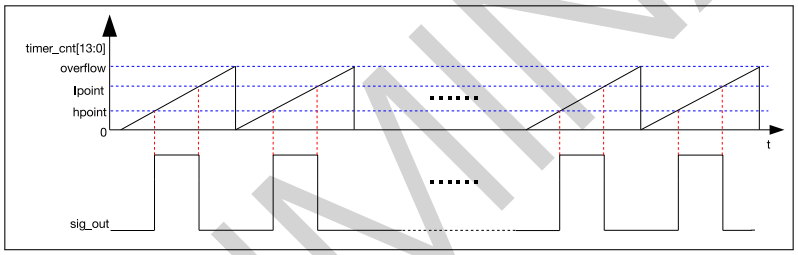


图 18-4. LED PWM 输出信号图

每当所选定时器的计数器溢出时，PWM 生成器 (PWMn) 的 Hpointn 值更新为 LEDC_HPOINT_CHn。Lpointn 的值同样在计数器每次溢出时更新，为 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。

置位 LEDC_SIG_OUT_EN_CHn，开启 PWM 信号（sig_outn）输出；清除 LEDC_SIG_OUT_EN_CHn，关闭 PWM 信号输出，输出信号 sig_outn 输出恒定电平，电平值为 LEDC_IDLE_LV_CHn.

LEDC_DUTY_CHn[3:0] 通过周期性改变 PWM 输出信号 sig_outn 的占空比实现微调。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 脉冲占空比要比 (16 - LEDC_DUTY_CHn[3:0]) 个周期的脉冲占空比多一个定时器的计数周期。比如，如果 LEDC_DUTY_CHn[18:4] 设为 10，LEDC_DUTY_CHn[3:0] 设为 5，则 16 个周期中，有 5 个周期的 PWM 脉冲占空比为 11，剩余 11 个周期的 PWM 脉冲占空比为 10。16 个周期的平均占空比为 10.3125。

如果重新配置 LEDC_TIMER_SEL_CHn、LEDC_HPOINT_CHn、LEDC_DUTY_CHn[18:4] 和 LEDC_SIG_OUT_EN_CHn 字段，需置位 LEDC_PARA_UP_CHn 应用新配置。新配置在计数器下次溢出时生效。LEDC_TIMERx_PARA_UP 字段由硬件自动清除。

18.2.4 占空比渐变

PWM 生成器可以渐变 PWM 输出信号的占空比，即由一种占空比逐渐变为为另一种占空比。如果开启占空比渐变功能，Lpointn 的值会在计数器溢出固定次数后递增或递减。图 18-5 展示了占空比渐变功能。

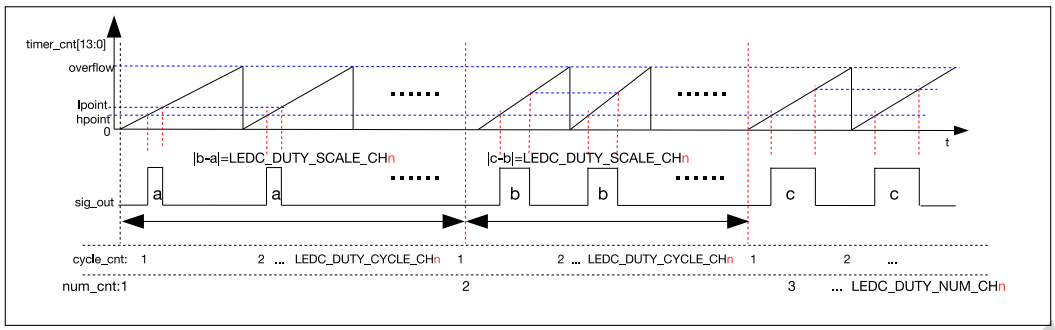


图 18-5. 输出信号占空比渐变图

占空比渐变功能可通过以下寄存器字段配置：

- LEDC_DUTY_CH*n* 用于设置 Lpoint*n* 的初始值。
- LEDC_DUTY_START_CH*n* 置 1 或清零，使能或关闭占空比渐变功能。
- LEDC_DUTY_CYCLE_CH*n* 用于设置 Lpoint*n* 在计数器溢出多少次时递增或递减。也就是说，Lpoint*n* 会在计数器溢出 LEDC_DUTY_CYCLE_CH*n* 次时递增或递减。
- LEDC_DUTY_INC_CH*n* 置 1 或清零，Lpoint*n* 递增或递减。
- LEDC_DUTY_SCALE_CH*n* 用于设置 Lpoint*n* 递增或递减的值。
- LEDC_DUTY_NUM_CH*n* 用于设置占空比渐变停止前，Lpoint*n* 递增或递减的最大次数。

如果重新配置 LEDC_DUTY_CH*n*、LEDC_DUTY_START_CH*n*、LEDC_DUTY_CYCLE_CH*n*、LEDC_DUTY_INC_CH*n*、LEDC_DUTY_SCALE_CH*n* 和 LEDC_DUTY_NUM_CH*n*字段，需置位 LEDC_PARA_UP_CH*n* 应用新配置。LEDC_PARA_UP_CH*n* 置位后，新配置立即生效。LEDC_TIMER*x*_PARA_UP 字段由硬件自动清除。

18.2.5 中断

- LEDC_OVF_CNT_CH*n*_INT：定时器计数器溢出 (LEDC_OVF_NUM_CH*n* + 1) 次且寄存器 LEDC_OVF_CNT_EN_CH*n* 置 1 时触发中断。
- LEDC_DUTY_CHNG_END_CH*n*_INT：PWM 生成器渐变完成后触发中断。
- LEDC_TIMER*x*_OVF_INT：定时器达到最大计数值时触发中断。

18.3 寄存器列表

本小节的所有地址均为相对于 LED PWM 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

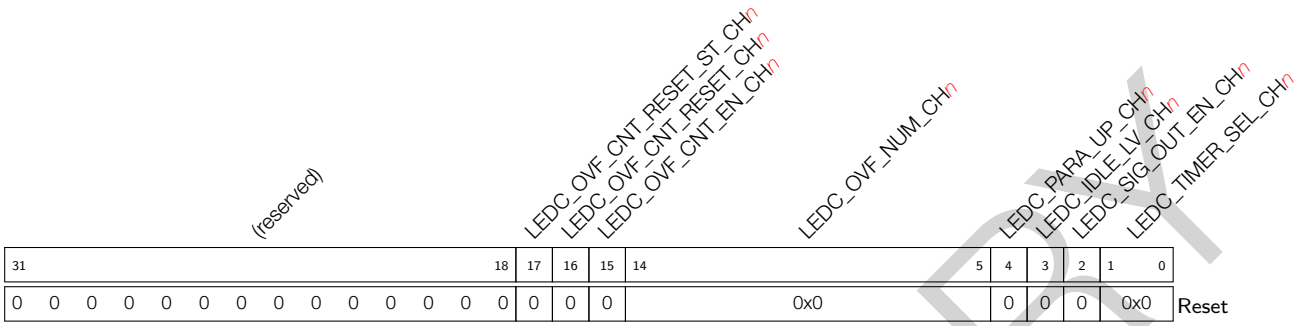
名称	描述	地址	访问
配置寄存器			
LEDC_CH0_CONF0_REG	通道 0 的配置寄存器 0	0x0000	不定
LEDC_CH0_CONF1_REG	通道 0 的配置寄存器 1	0x000C	读/写
LEDC_CH1_CONF0_REG	通道 1 的配置寄存器 0	0x0014	不定
LEDC_CH1_CONF1_REG	通道 1 的配置寄存器 1	0x0020	读/写
LEDC_CH2_CONF0_REG	通道 2 的配置寄存器 0	0x0028	不定
LEDC_CH2_CONF1_REG	通道 2 的配置寄存器 1	0x0034	读/写
LEDC_CH3_CONF0_REG	通道 3 的配置寄存器 0	0x003C	不定
LEDC_CH3_CONF1_REG	通道 3 的配置寄存器 1	0x0048	读/写
LEDC_CH4_CONF0_REG	通道 4 的配置寄存器 0	0x0050	不定
LEDC_CH4_CONF1_REG	通道 4 的配置寄存器 1	0x005C	读/写
LEDC_CH5_CONF0_REG	通道 5 的配置寄存器 0	0x0064	不定
LEDC_CH5_CONF1_REG	通道 5 的配置寄存器 1	0x0070	读/写
LEDC_CH6_CONF0_REG	通道 6 的配置寄存器 0	0x0078	不定
LEDC_CH6_CONF1_REG	通道 6 的配置寄存器 1	0x0084	读/写
LEDC_CH7_CONF0_REG	通道 7 的配置寄存器 0	0x008C	不定
LEDC_CH7_CONF1_REG	通道 7 的配置寄存器 1	0x0098	读/写
LEDC_CONF_REG	LEDC 全局配置寄存器	0x00D0	读/写
高位点寄存器			
LEDC_CH0_HPOINT_REG	通道 0 的高位点寄存器	0x0004	读/写
LEDC_CH1_HPOINT_REG	通道 1 的高位点寄存器	0x0018	读/写
LEDC_CH2_HPOINT_REG	通道 2 的高位点寄存器	0x002C	读/写
LEDC_CH3_HPOINT_REG	通道 3 的高位点寄存器	0x0040	读/写
LEDC_CH4_HPOINT_REG	通道 4 的高位点寄存器	0x0054	读/写
LEDC_CH5_HPOINT_REG	通道 5 的高位点寄存器	0x0068	读/写
LEDC_CH6_HPOINT_REG	通道 6 的高位点寄存器	0x007C	读/写
LEDC_CH7_HPOINT_REG	通道 7 的高位点寄存器	0x0090	读/写
占空比寄存器			
LEDC_CH0_DUTY_REG	通道 0 的初始占空比	0x0008	读/写
LEDC_CH0_DUTY_R_REG	通道 0 的当前占空比	0x0010	只读
LEDC_CH1_DUTY_REG	通道 1 的初始占空比	0x001C	读/写
LEDC_CH1_DUTY_R_REG	通道 1 的当前占空比	0x0024	只读
LEDC_CH2_DUTY_REG	通道 2 的初始占空比	0x0030	读/写
LEDC_CH2_DUTY_R_REG	通道 2 的当前占空比	0x0038	只读
LEDC_CH3_DUTY_REG	通道 3 的初始占空比	0x0044	读/写
LEDC_CH3_DUTY_R_REG	通道 3 的当前占空比	0x004C	只读
LEDC_CH4_DUTY_REG	通道 4 的初始占空比	0x0058	读/写
LEDC_CH4_DUTY_R_REG	通道 4 的当前占空比	0x0060	只读
LEDC_CH5_DUTY_REG	通道 5 的初始占空比	0x006C	读/写

名称	描述	地址	访问
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

18.4 寄存器

本小节的所有地址均为相对于 LED PWM 控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

Register 18.1. LEDC_CH*n*_CONF0_REG (*n*: 0-7) (0x0000+0x14n*)**



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 (LEDC_SIG_OUT_EN_CH*n* 为 0 时) 的输出电平。(读/写)

LEDC_PARA_UP_CH*n* 用于更新通道 *n* 的的下列字段，由硬件自动清除。(只写)

- LEDC_HPOINT_CH*n*
- LEDC_DUTY_START_CH*n*
- LEDC_SIG_OUT_EN_CH*n*
- LEDC_TIMER_SEL_CH*n*
- LEDC_DUTY_NUM_CH*n*
- LEDC_DUTY_CYCLE_CH*n*
- LEDC_DUTY_SCALE_CH*n*
- LEDC_DUTY_INC_CH*n*
- LEDC_OVF_CNT_EN_CH*n*

见下页...

Register 18.1. LEDC_CH*n*_CONF0_REG (*n*: 0-7) (0x0000+0x14n*)**

接上页...

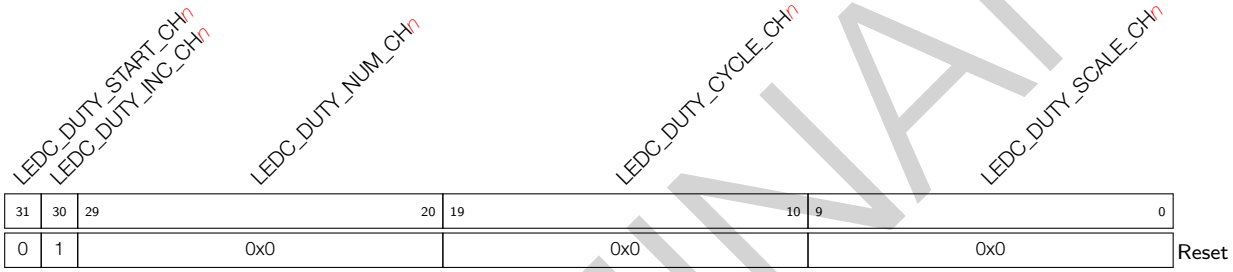
LEDC_OVF_NUM_CH*n* 用于配置定时器溢出次数的最大值减 1。通道 *n* 的定时器溢出次数达到 (LEDC_OVF_NUM_CH*n* +1) 次时，触发 LEDC_OVF_CNT_CH*n*_INT 中断。(读/写)

LEDC_OVF_CNT_EN_CH*n* 用于计算通道 *n* 选择的定时器溢出的次数。(读/写)

LEDC_OVF_CNT_RESET_CH*n* 置位此位，复位通道 *n* 的定时器溢出计数器。(只写)

LEDC_OVF_CNT_RESET_ST_CH*n* LEDC_OVF_CNT_RESET_CH*n* 的状态位。(只读)

Register 18.2. LEDC_CH*n*_CONF1_REG (*n*: 0-7) (0x000C+0x14n*)**



LEDC_DUTY_SCALE_CH*n* 用于配置通道 *n* 占空比的变化步长。(读/写)

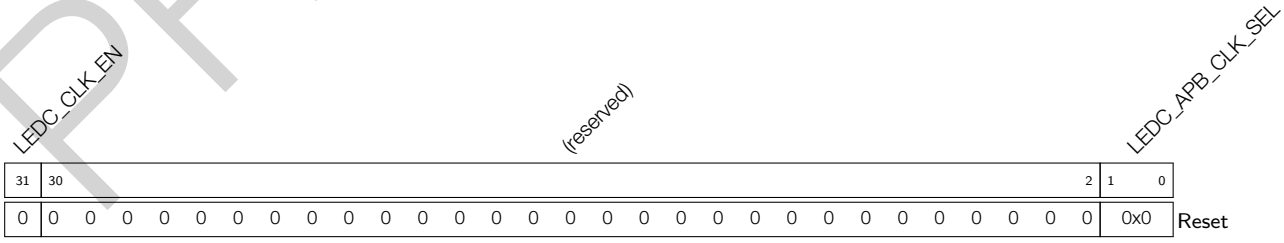
LEDC_DUTY_CYCLE_CH*n* 通道 *n* 占空比每隔 LEDC_DUTY_CYCLE_CH*n* 变化一次。(读/写)

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 中的其他字段在定时器下次溢出时生效。(读/写)

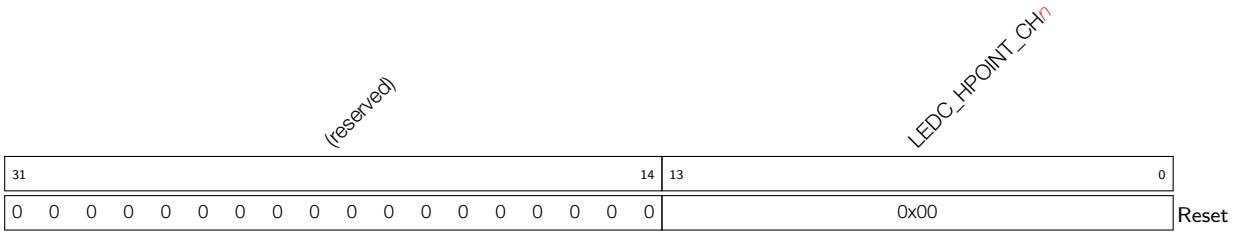
Register 18.3. LEDC_CONF_REG (0x00D0)



LEDC_APB_CLK_SEL 用于设置 4 个定时器的共同时钟源。1：APB_CLK；2：FOSC_CLK；3：XTAL_CLK。(读/写)

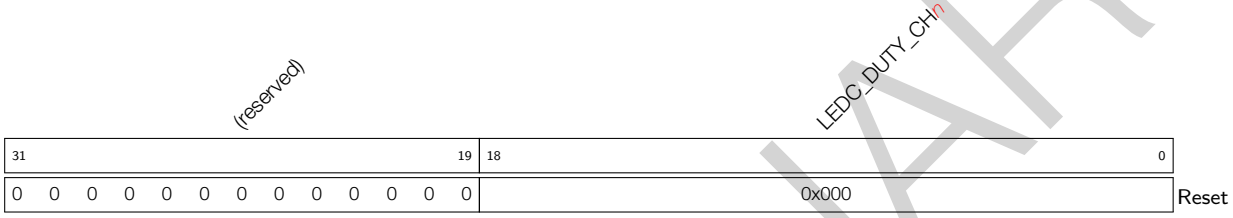
LEDC_CLK_EN 用于控制时钟。1：强制开启寄存器时钟。1：仅在应用写寄存器时支持时钟。(读/写)

Register 18.4. LEDC_CH*n*_HPOINT_REG (*n*: 0-7) (0x0004+0x14n*)**



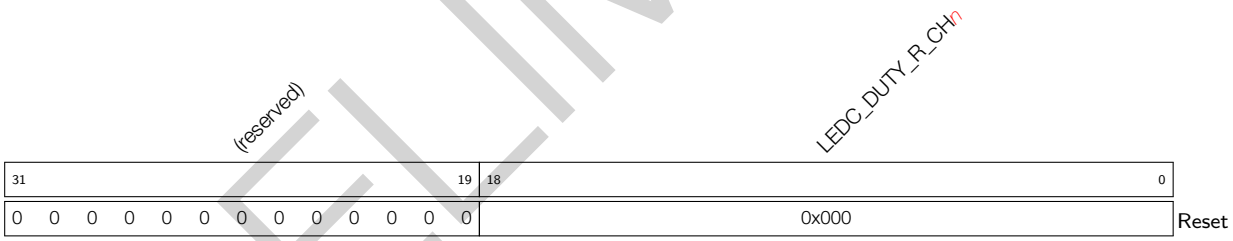
LEDC_HPOINT_CH*n* 所选定时器计数值达到该值时，输出信号翻转为高电平。(读/写)

Register 18.5. LEDC_CH*n*_DUTY_REG (*n*: 0-7) (0x0008+0x14n*)**



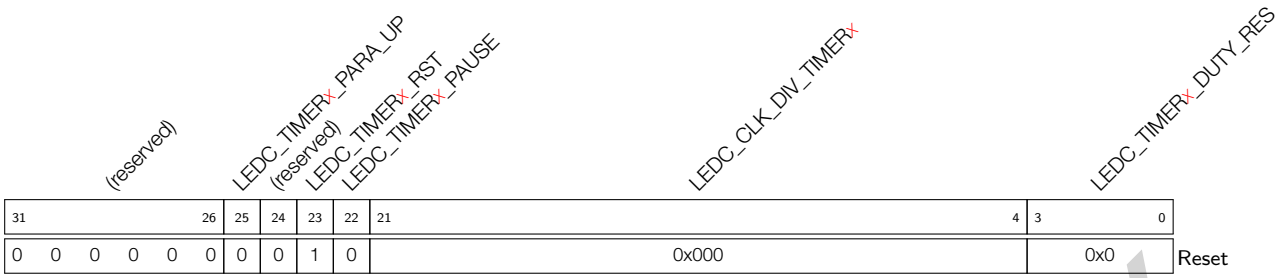
LEDC_DUTY_CH*n* 通过控制低位点改变输出信号占空比。所选定时器达到低位点时，输出信号翻转为低电平。(读/写)

Register 18.6. LEDC_CH*n*_DUTY_R_REG (*n*: 0-7) (0x0010+0x14n*)**



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

Register 18.7. LEDC_TIMERx_CONF_REG (x: 0-3) (0x00A0+0x8*x)



LEDC_TIMERx_DUTY_RES 用于控制定时器 x 计数器的计数范围。(读/写)

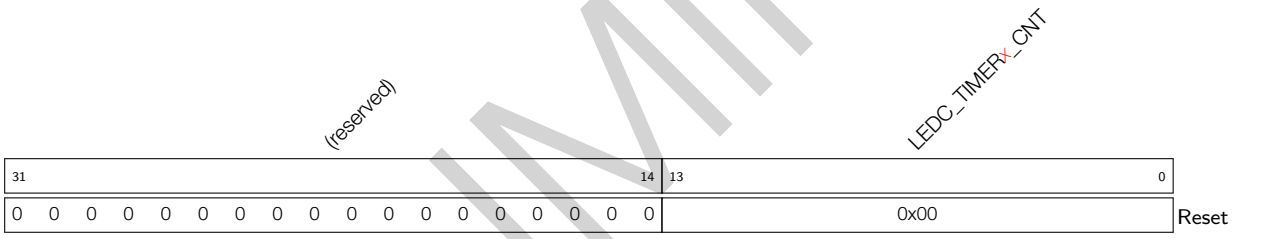
LEDC_CLK_DIV_TIMERx 用于配置定时器 x 分频器的分频系数。低 8 位为小数部分。(读/写)

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

LEDC_TIMERx_RST 用于复位定时器 x。复位后计数器为 0。(读/写)

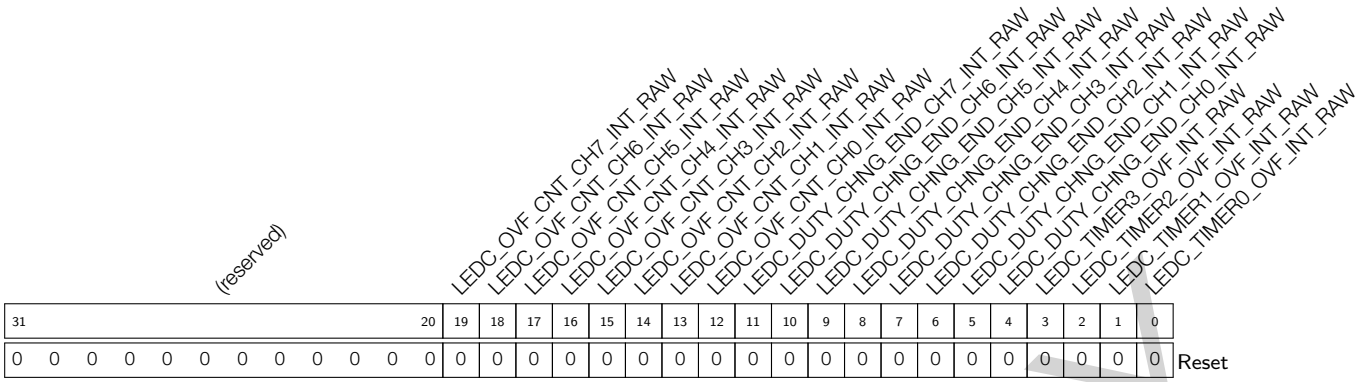
LEDC_TIMERx_PARA_UP 置位此位，更新 LEDC_CLK_DIV_TIMERx 和 LEDC_TIMERx_DUTY_RES。(只写)

Register 18.8. LEDC_TIMERx_VALUE_REG (x: 0-3) (0x00A4+0x8*x)



LEDC_TIMERx_CNT 存储定时器 x 的当前计数器值 (只读)

Register 18.9. LEDC_INT_RAW_REG (0x00C0)

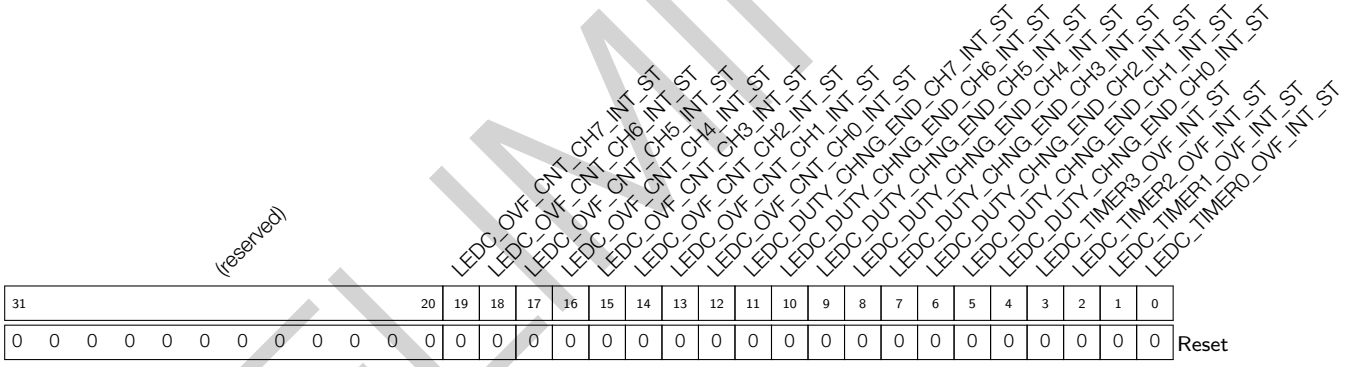


LEDC_TIMER*x*_OVF_INT_RAW 定时器 *x* 达到最大计数值时触发中断。(只读)

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 18.10. LEDC_INT_ST_REG (0x00C4)

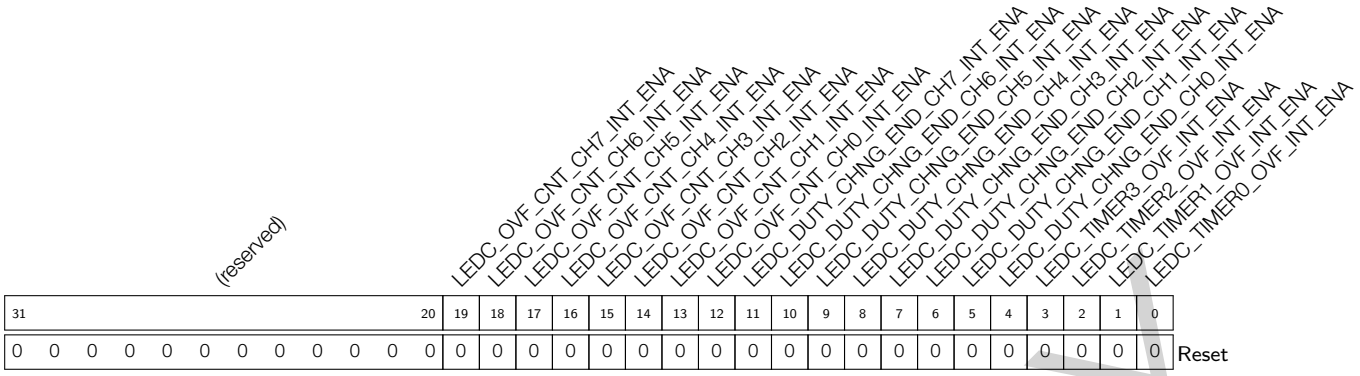


LEDC_TIMER*x*_OVF_INT_ST LEDC_TIMER*x*_OVF_INT_ENA 置 1 时，LEDC_TIMER*x*_OVF_INT 中断的屏蔽中断状态位。(只读)

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 18.11. LEDC_INT_ENA_REG (0x00C8)

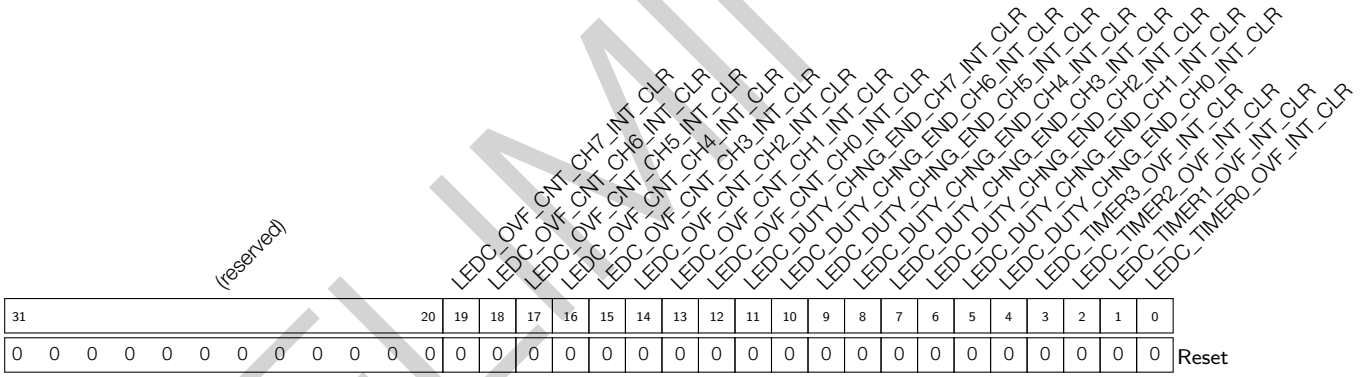


LEDC_TIMER*x*_OVF_INT_ENA LEDC_TIMER*x*_OVF_INT 中断的使能位。(读/写)

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 18.12. LEDC_INT_CLR_REG (0x00CC)

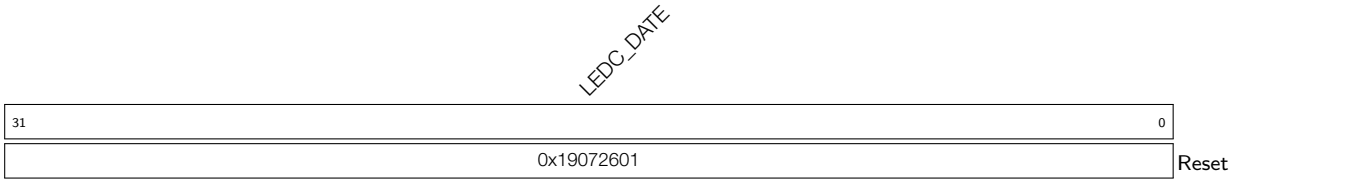


LEDC_TIMER*x*_OVF_INT_CLR 置位此位，清除 LEDC_TIMER*x*_OVF_INT 中断。(只写)

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 18.13. LEDC_DATE_REG (0x00FC)



LEDC_DATE 版本控制寄存器。(读/写)

19 脉冲计数控制器 (PCNT)

脉冲计数控制器 (Pulse Count Controller, PCNT) 用于对输入脉冲计数，通过记录输入脉冲信号的上升沿或下降沿进行递增或递减计数。PCNT 有四个称为“单元”的独立脉冲计数控制器，这些单元拥有自己的寄存器。PCNT 模块仅有一个时钟，为 APB_CLK。下文描述中 *n* 表示单元编号 0 ~ 3。

每个单元有两个通道（ch0 和 ch1），可以独立配置为递增或递减计数。两个通道功能相同，下文以通道 0 (ch0) 为例进行介绍。

如图 19-1 所示，每个通道有两个输入信号：

1. 一个脉冲输入信号（如 sig_ch0_u*n* 为单元 *n* ch0 的脉冲输入信号）
2. 一个控制信号（如 ctrl_ch0_u*n* 为单元 *n* ch0 的控制信号）

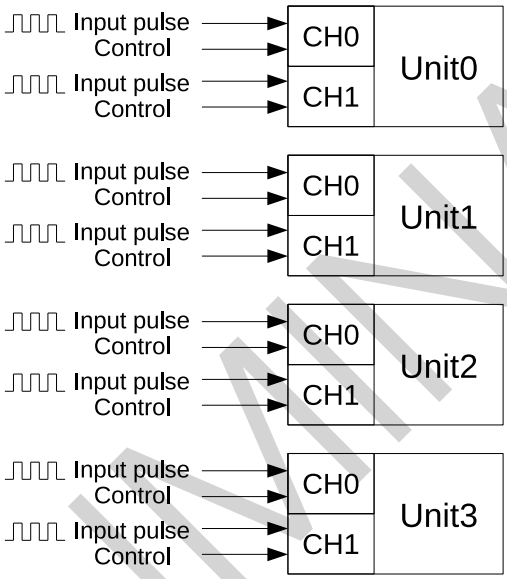


图 19-1. PCNT 框图

19.1 主要特性

PCNT 有如下特性：

- 四个脉冲计数控制器（单元），各自独立工作，计数范围是 1 ~ 65535
- 每个单元有两个独立的通道，共用一个脉冲计数控制器
- 所有通道均有输入脉冲信号（如 sig_ch0_u*n*）和相应的控制信号（如 ctrl_ch0_u*n*）
- 滤波器独立工作，过滤每个单元输入脉冲信号（sig_ch0_u*n* 和 sig_ch1_u*n*）控制信号（ctrl_ch0_u*n* 和 ctrl_ch1_u*n*）的毛刺
- 每个通道参数如下：
  1. 选择在输入脉冲信号的上升沿或下降沿计数
  2. 在控制信号为高电平或低电平时可将计数模式配置为递增、递减或停止计数

19.2 功能描述

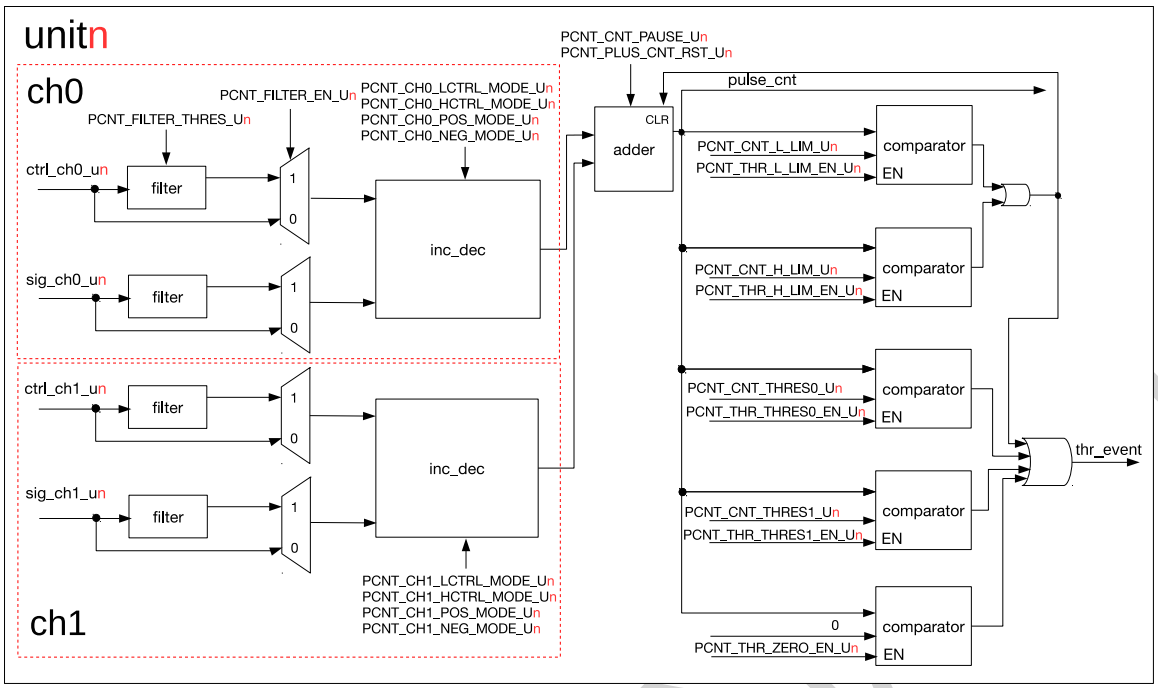


图 19-2. PCNT 单元基本架构图

图 19-2 为 PCNT 单元的基本架构图。如上所述，ctrl_ch0_u*n* 为单元 *n* ch0 的控制信号，控制信号 ctrl_ch0_u*n* 为高电平或低电平时可配置不同的计数模式，在输入脉冲信号 sig_ch0_u*n* 的上升沿和下降沿计数。可选计数模式如下：

- 递增模式：通道检测到 sig_ch0_u*n* 的有效边沿（软件可配）时，计数器的值 pulse_cnt 加 1。pulse_cnt 的值达到 PCNT_CNT_H_LIM_U*n* 时被清零。如果在 pulse_cnt 达到 PCNT_CNT_H_LIM_U*n* 前，该通道的计数模式改变或 PCNT_CNT_PAUSE_U*n* 置 1，则 pulse_cnt 停止计数，计数模式改变。
- 递减模式：通道检测到 sig_ch0_u*n* 的有效边沿（软件可配）时，计数器的值 pulse_cnt 减 1。pulse_cnt 的值达到 PCNT_CNT_L_LIM_U*n* 时被清零。如果在 pulse_cnt 达到 PCNT_CNT_H_LIM_U*n* 前，该通道的计数模式改变或 PCNT_CNT_PAUSE_U*n* 置 1，则 pulse_cnt 停止计数，计数模式改变。
- 停止计数：计数停止，计数器的值 pulse_cnt 保持不变。

表 19-1 至表 19-4 说明了如何配置通道 0 的计数模式。

表 19-1. 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 19-2. 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 19-3. 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 19-4. 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加法器模块，该模块有一个带符号位的 16 位宽寄存器。软件可以通过置位 PCNT_CNT_PAUSE_U*n* 暂停加法器，也可以通过置位 PCNT_PULSE_CNT_RST_U*n* 清零加法器。

PCNT 可以设置五个观察点，五个观察点共用一个中断，可以通过每个观察点各自的中断使能信号开启或屏蔽中断。

- 最大计数值：当 pulse_cnt 大于等于 PCNT_CNT_H_LIM_U*n* 时，产生上限中断，同时 PCNT_CNT_THR_H_LIM_LAT_U*n* 为高。

- 最小计数值：当 pulse_cnt 小于等于 PCNT_CNT_L_LIM_U*n* 时，产生下限中断，同时 PCNT_CNT_THR_L_LIM_LAT_U*n* 为高。
- 两个中间阈值：当 pulse_cnt 等于 PCNT_CNT_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。

19.3 应用实例

每个单元的通道 0 和通道 1 可配置为独立工作或一起工作。下文详细说明了通道 0 独自递增计数、通道 0 独自递减计数和两个通道一起递增计数的应用实例。本节中未详述的通道工作模式（如通道 1 独自递减或递减、双通道一增一减），可参考这三种模式。

19.3.1 通道 0 独自递增计数

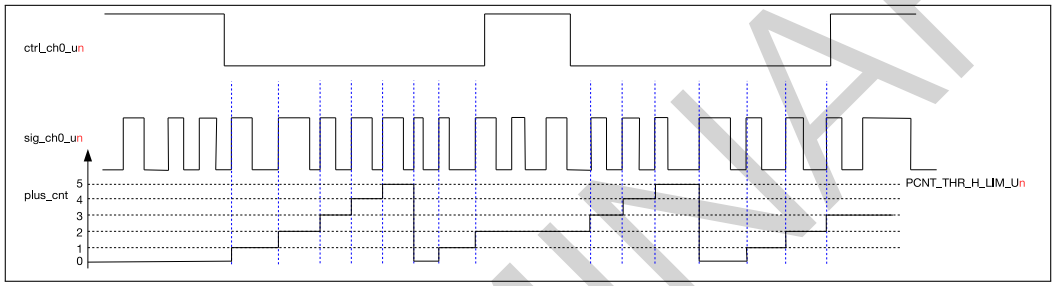


图 19-3. 通道 0 递增计数图

图 19-3 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 19.2 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。

19.3.2 通道 0 独自递减计数

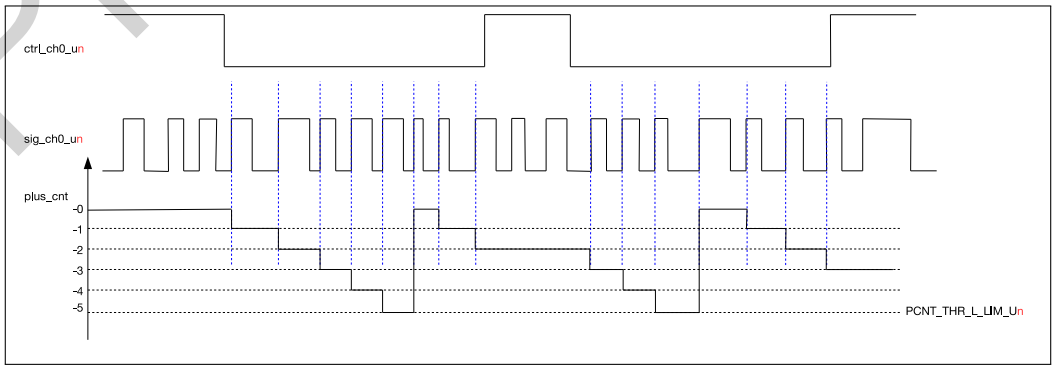


图 19-4. 通道 0 递减计数图

图 19-4 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 19-3 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。

19.3.3 通道 0 和通道 1 同时递增计数

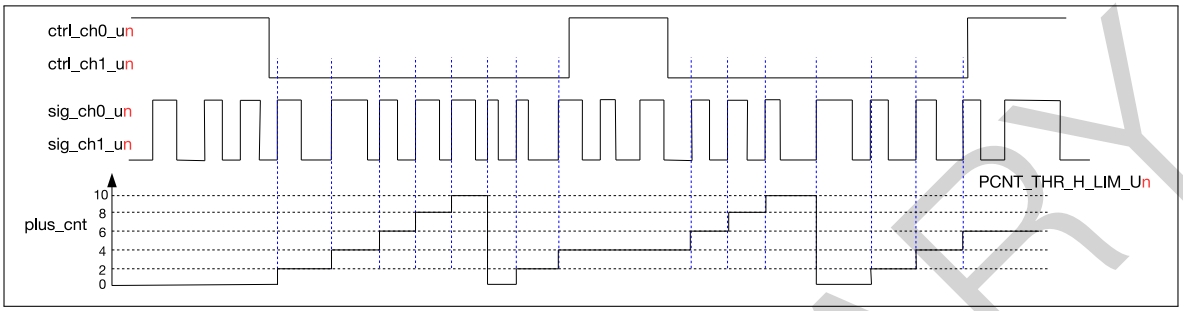


图 19-5. 双通道递增计数图

图 19-5 为通道 0 和通道 1 分别在 sig_ch0_u*n* 和 sig_ch1_u*n* 上升沿同时递增计数的示意图。如图 19-5 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。

19.4 寄存器列表

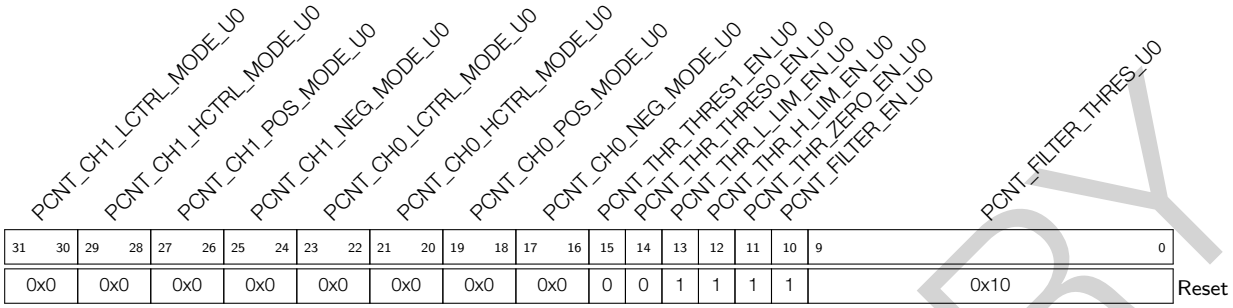
本小节的所有地址均为相对于 脉冲计数控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

名称	描述	地址	访问
配置寄存器			
PCNT_U0_CONF0_REG	单元 0 的配置寄存器 0	0x0000	读/写
PCNT_U0_CONF1_REG	单元 0 的配置寄存器 1	0x0004	读/写
PCNT_U0_CONF2_REG	单元 0 的配置寄存器 2	0x0008	读/写
PCNT_U1_CONF0_REG	单元 1 的配置寄存器 0	0x000C	读/写
PCNT_U1_CONF1_REG	单元 1 的配置寄存器 1	0x0010	读/写
PCNT_U1_CONF2_REG	单元 1 的配置寄存器 2	0x0014	读/写
PCNT_U2_CONF0_REG	单元 2 的配置寄存器 0	0x0018	读/写
PCNT_U2_CONF1_REG	单元 2 的配置寄存器 1	0x001C	读/写
PCNT_U2_CONF2_REG	单元 2 的配置寄存器 2	0x0020	读/写
PCNT_U3_CONF0_REG	单元 3 的配置寄存器 0	0x0024	读/写
PCNT_U3_CONF1_REG	单元 3 的配置寄存器 1	0x0028	读/写
PCNT_U3_CONF2_REG	单元 3 的配置寄存器 2	0x002C	读/写
PCNT_CTRL_REG	所有计数器的控制寄存器	0x0060	读/写
状态寄存器			
PCNT_U0_CNT_REG	单元 0 的计数器值	0x0030	只读
PCNT_U1_CNT_REG	单元 1 的计数器值	0x0034	只读
PCNT_U2_CNT_REG	单元 2 的计数器值	0x0038	只读
PCNT_U3_CNT_REG	单元 3 的计数器值	0x003C	只读
PCNT_U0_STATUS_REG	脉冲计数器单元 0 的状态寄存器	0x0050	只读
PCNT_U1_STATUS_REG	脉冲计数器单元 1 的状态寄存器	0x0054	只读
PCNT_U2_STATUS_REG	脉冲计数器单元 2 的状态寄存器	0x0058	只读
PCNT_U3_STATUS_REG	脉冲计数器单元 3 的状态寄存器	0x005C	只读
中断寄存器			
PCNT_INT_RAW_REG	原始中断状态寄存器	0x0040	只读
PCNT_INT_ST_REG	中断状态寄存器	0x0044	只读
PCNT_INT_ENA_REG	中断使能寄存器	0x0048	读/写
PCNT_INT_CLR_REG	中断清除寄存器	0x004C	只写
版本寄存器			
PCNT_DATE_REG	脉冲计数器的版本控制寄存器	0x00FC	读/写

19.5 寄存器

本小节的所有地址均为相对于 脉冲计数控制器 基地址的地址偏移量（相对地址），具体基地址请见章节 1 系统和存储器 中的表 1-4。

Register 19.1. PCNT_U*n*_CONF0_REG (*n*: 0-3) (0x0000+0xCn*)**



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)

见下页...

Register 19.1. PCNT_U*n*_CONF0_REG (*n*: 0-3) (0x0000+0xCn*)**

接上页...

PCNT_CH1_POS_MODE_U*n* 用于设置通道 1 输入信号检测上升沿的工作模式。

1：计数器递增；2：计数器递减；0、3：对计数器无任何影响。(读/写)

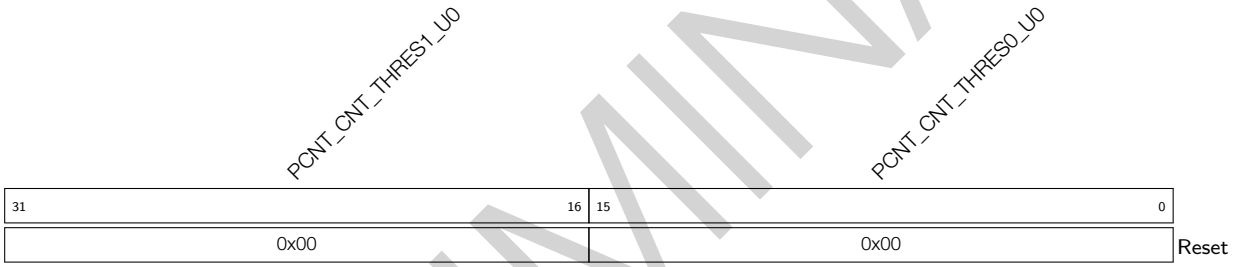
PCNT_CH1_HCTRL_MODE_U*n* 控制信号为高电平时，用于改变 CH*n*_POS_MODE 和 CH*n*_NEG_MODE 的设置。

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)

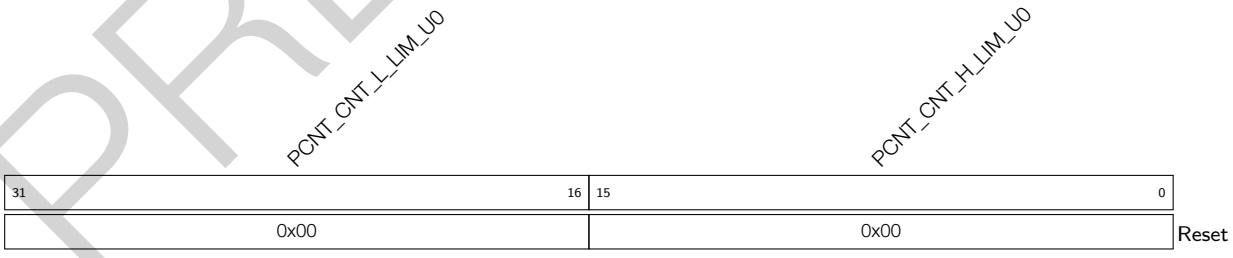
Register 19.2. PCNT_U*n*_CONF1_REG (*n*: 0-3) (0x0004+0xCn*)**



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)

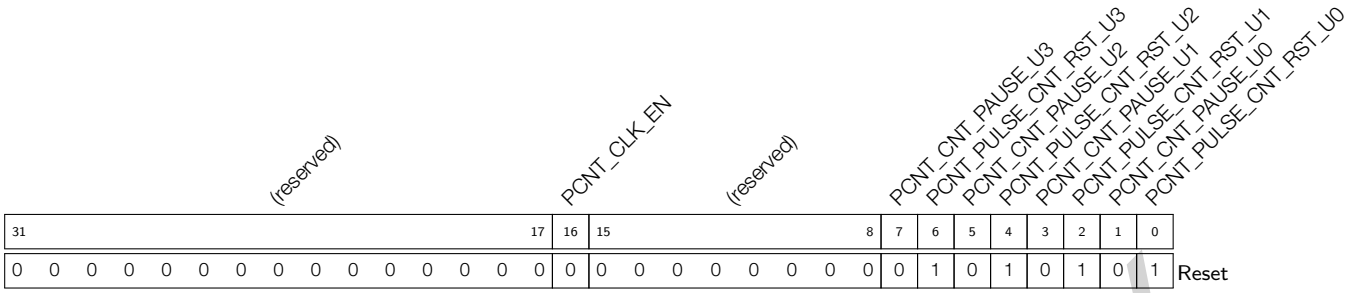
Register 19.3. PCNT_U*n*_CONF2_REG (*n*: 0-3) (0x0008+0xCn*)**



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 19.4. PCNT_CTRL_REG (0x0060)

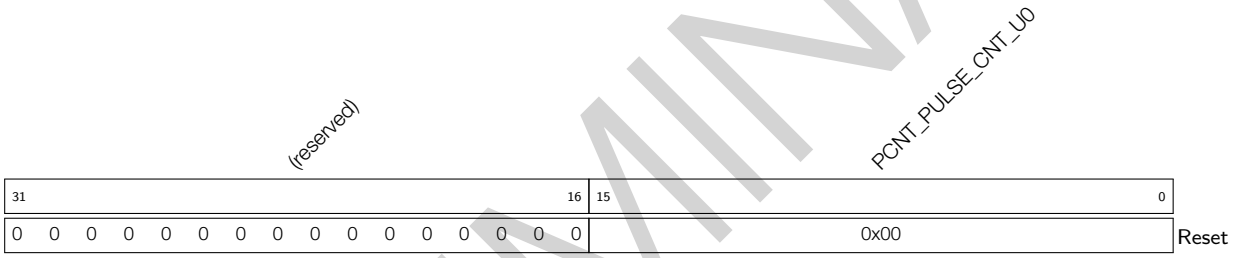


PCNT_PULSE_CNT_RST_U*n* 置位此位，清零单元 *n* 的计数器。(读/写)

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

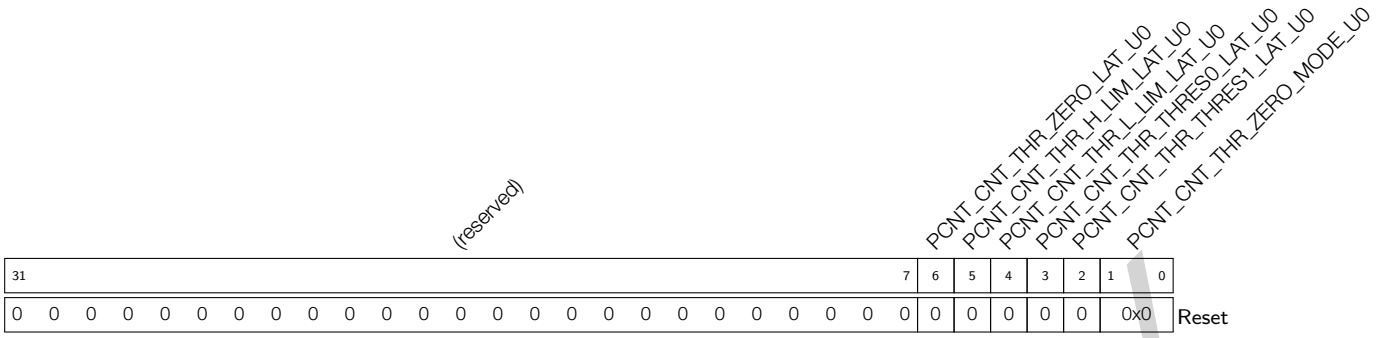
PCNT_CLK_EN 脉冲计数器模块寄存器时钟门控的使能信号 1：寄存器可通过应用读取、写值。0：寄存器无法通过应用读取、写值。(读/写)

Register 19.5. PCNT_U*n*_CNT_REG (*n*: 0-3) (0x0030+0x4n*)**



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

Register 19.6. PCNT_U*n*_STATUS_REG (*n*: 0-3) (0x0050+0x4n*)**



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)

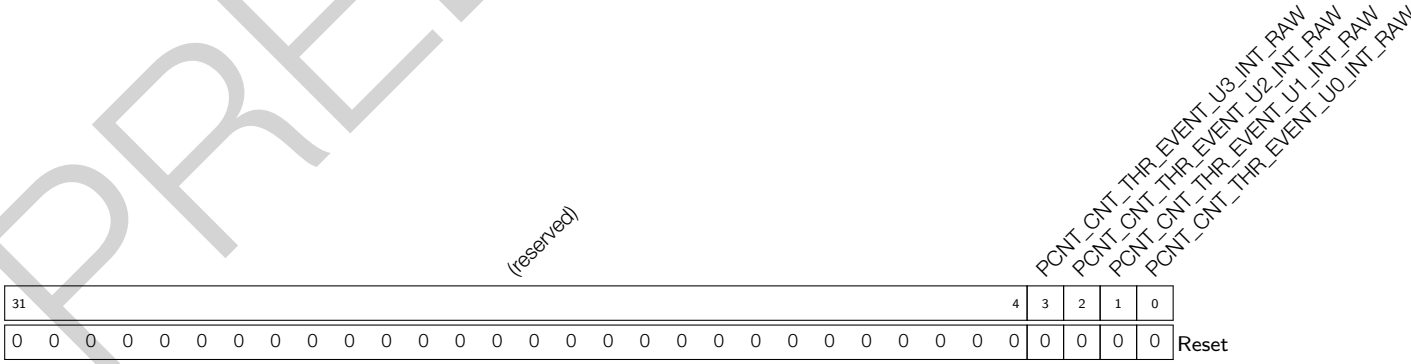
PCNT_CNT_THR_THRES0_LAT_U*n* 阈值中断有效时，PCNT_U*n* 阈值 0 的锁存值。1：脉冲计数器的当前值与阈值 0 相等，阈值 0 有效。0：其他。(只读)

PCNT_CNT_THR_L_LIM_LAT_U*n* 下限中断有效时，PCNT_U*n* 下限的锁存值。1：脉冲计数器的当前值与下限阈值相等，下限有效。0：其他。(只读)

PCNT_CNT_THR_H_LIM_LAT_U*n* 上限中断有效时，PCNT_U*n* 上限的锁存值。1：脉冲计数器的当前值与上限阈值相等，上限有效。0：其他。(只读)

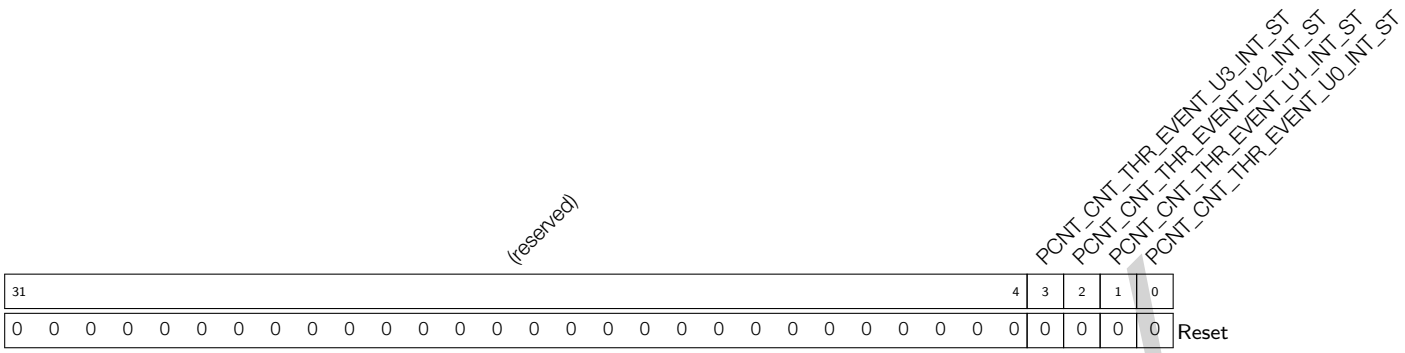
PCNT_CNT_THR_ZERO_LAT_U*n* 阈值中断有效时，PCNT_U*n* 阈值 0 的锁存值。1：脉冲计数器的当前值为 0，阈值 0 有效。0：其他。(只读)

Register 19.7. PCNT_INT_RAW_REG (0x0040)



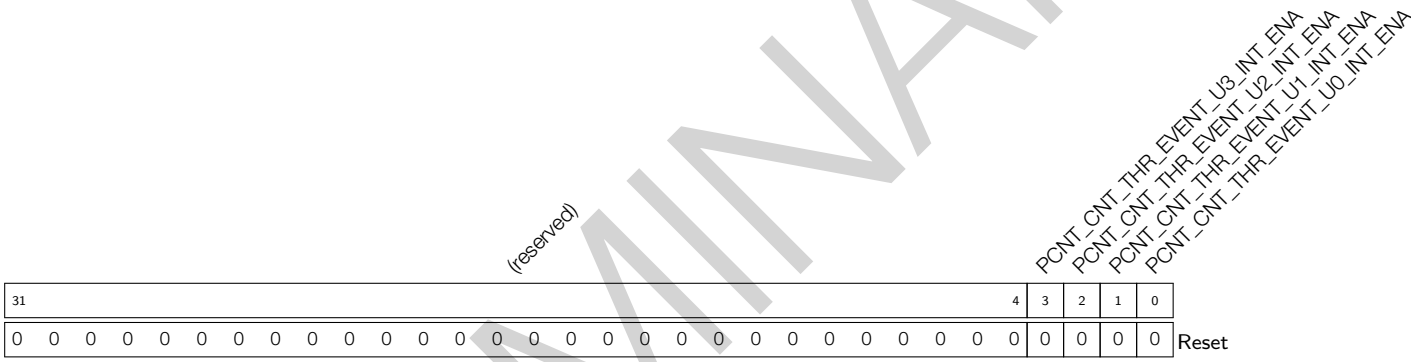
PCNT_CNT_THR_EVENT_U*n*_INT_RAW 单元 *n* 事件中断的原始中断状态位。(只读)

Register 19.8. PCNT_INT_ST_REG (0x0044)



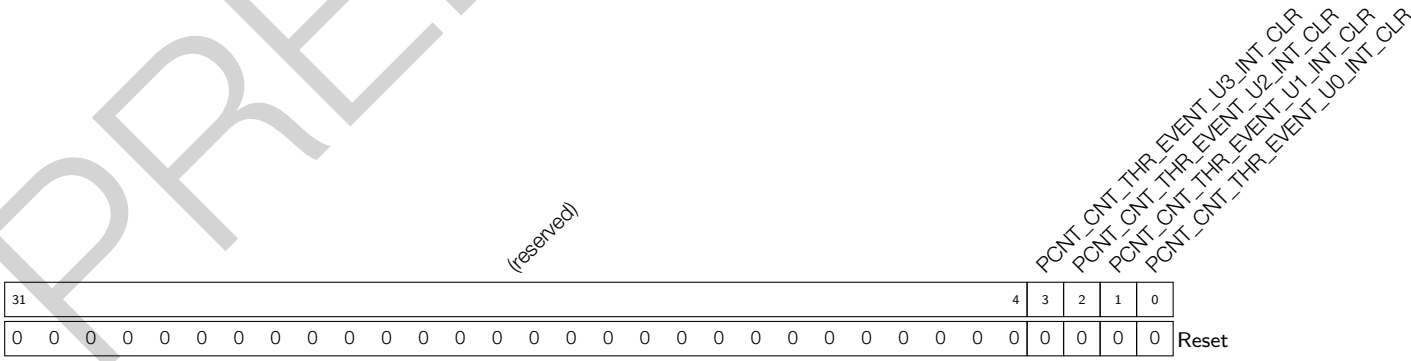
PCNT_CNT_THR_EVENT_U*n*_INT_ST 单元 *n* 事件中断的屏蔽中断状态位。(只读)

Register 19.9. PCNT_INT_ENA_REG (0x0048)



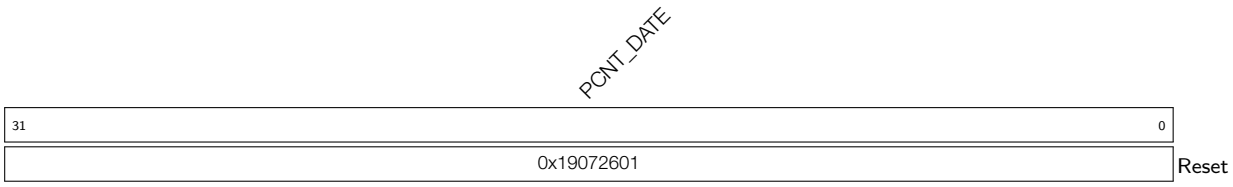
PCNT_CNT_THR_EVENT_U*n*_INT_ENA 单元 *n* 事件中断的中断使能位。(读/写)

Register 19.10. PCNT_INT_CLR_REG (0x004C)



PCNT_CNT_THR_EVENT_U*n*_INT_CLR 置位此位，清除单元 *n* 事件中断。(只写)

Register 19.11. PCNT_DATE_REG (0x00FC)



PCNT_DATE 脉冲计数器的版本控制寄存器。(读/写)

词汇列表

外设相关词汇

AES	AES 加速器
BOOTCTRL	芯片 Boot 控制
DS	数字签名
DMA	DMA 控制器
eFuse	eFuse 控制器
HMAC	HMAC 加速器
I2C	I2C 控制器
I2S	I2S 控制器
LEDC	LED 控制 PWM
MCPWM	电机控制 PWM
PCNT	脉冲计数器控制器
RMT	红外遥控
RNG	随机数生成器
RSA	RSA 加速器
SDHOST	SD/MMC 主机控制器
SHA	SHA 加速器
SPI	SPI 控制器
SYSTIMER	系统定时器
TIMG	定时器组
TWAI	双线汽车接口
UART	UART 控制器
ULP 协处理器	超低功耗协处理器
USB OTG	USB On-The-Go
WDT	看门狗定时器

寄存器相关词汇

ISO	隔离。当模块断电时，其输出的引脚将处于未知状态（某些中间电压）。“ISO”寄存器将使其输出引脚隔离在一个确定的电压，从而不会影响其他未掉电的工作模块的状态。
NMI	不可屏蔽中断。
REG	寄存器。
R/W	读/写，软件可读写这些位。
RO	只读，软件只能读这些位。
SYSREG	系统寄存器。
WO	只写，软件只能写这些位。

修订历史

日期	版本	发布说明
2021-07-09	V0.1	Preliminary release

www.espressif.com

免责声明和版权公告

本文档中的信息，包括供参考的 URL 地址，如有变更，恕不另行通知。

本文档可能引用了第三方的信息，所有引用的信息均为“按现状”提供，乐鑫不对信息的准确性、真实性做任何保证。

乐鑫不对本文档的内容做任何保证，包括内容的适销性、是否适用于特定用途，也不提供任何其他乐鑫提案、规格书或样品在他处提到的任何保证。

乐鑫不对本文档是否侵犯第三方权利做任何保证，也不对使用本文档内信息导致的任何侵犯知识产权的行为负责。本文档在此未以禁止反言或其他方式授予任何知识产权许可，不管是明示许可还是暗示许可。

Wi-Fi 联盟成员标志归 Wi-Fi 联盟所有。蓝牙标志是 Bluetooth SIG 的注册商标。

文档中提到的所有商标名称、商标和注册商标均属其各自所有者的财产，特此声明。