ESP32-C3

技术参考手册

PRELIMINARY

预发布 v0.1
乐鑫信息科技
版权 © 2021

关于本手册

《ESP32-C3 技术参考手册》的目标读者群体是使用 ESP32-C3 芯片的应用开发工程师。本手册提供了关于 ESP32-C3 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-C3 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。

PRELIMINARY

目录

PRELIMINARY

PRELIMINARY

PRELIMINARY

PRELIMINARY

表格

PRELIMINARY

插图

PRELIMINARY

1 复位和时钟

1.1 复位

1.1.1 概述

ESP32-C3 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 1-1 展示了整个芯片系统的结构以及四种复位等级。

1.1.2 结构图

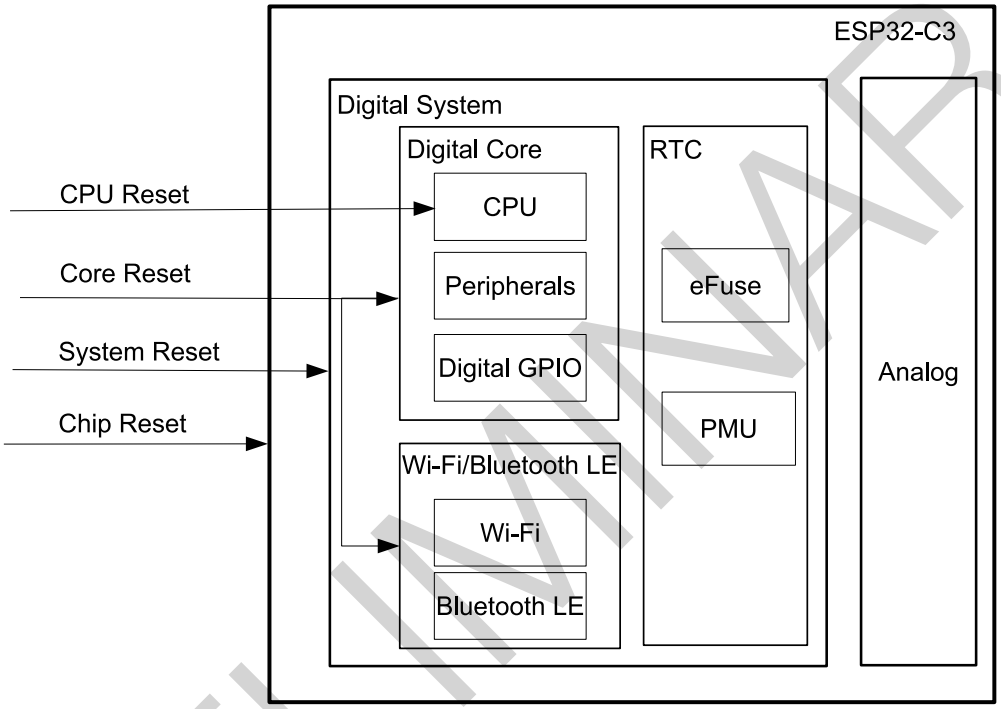


图 1-1. 四种复位等级

1.1.3 特性

- 支持四种复位等级：
  - CPU 复位：复位 CPU 核。复位释放后，程序将从 CPU Reset Vector 开始执行；
  - 内核复位：复位除 RTC 以外的其它数字系统，包括 CPU、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 配置相关寄存器可触发软件复位。
  - 硬件复位：硬件复位直接由硬件电路触发。

PRELIMINARY

说明：
如果 CPU 发生复位，则 SENSITIVE 寄存器也将复位。

1.1.4 功能描述

上述任一复位发生时，CPU 将立刻复位。复位释放后，CPU 可通过读取寄存器 RTC_CNTL_RESET_CAUSE _PROCPU 获取复位源。

表 1-1 列出了从上述寄存器中可能读出的复位源以及触发的复位方式。

表 1-1. 复位源

编码	复位源	复位方式	注释
0x01	芯片复位	芯片复位	见表下方说明 [1]
0x0F	欠压系统复位	芯片复位或系统复位	欠压检测器触发的系统复位，见表下方说明[2]
0x10	RWDT 系统复位	系统复位	
0x13	时钟毛刺复位	系统复位	
0x12	超级看门狗复位	系统复位	
0x03	软件系统复位	内核复位	配置 RTC_CNTL_SW_SYS_RST 寄存器触发
0x05	Deep-sleep 复位	内核复位	
0x07	MWDT0 内核复位	内核复位	
0x08	MWDT1 内核复位	内核复位	
0x09	RWDT 内核复位	内核复位	
0x14	eFuse 复位	内核复位	eFuse CRC 校验错误触发复位
0x17	电源毛刺复位	内核复位	电源毛刺触发复位
0x0B	MWDT0 CPU 复位	CPU 复位	
0x0C	软件 CPU 复位	CPU 复位	配置 RTC_CNTL_SW_PROCPU_RST 寄存器触发
0x0D	RWDT CPU 复位	CPU 复位	
0x11	MWDT1 CPU 复位	CPU 复位	

说明：

1. 芯片复位的触发源包括以下两项：
   - 芯片上电触发芯片复位
   - 欠压检测器触发芯片复位
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。

1.2 时钟

1.2.1 概述

ESP32-C3 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 1-2 为系统时钟结构。

PRELIMINARY

1.2.2 结构图

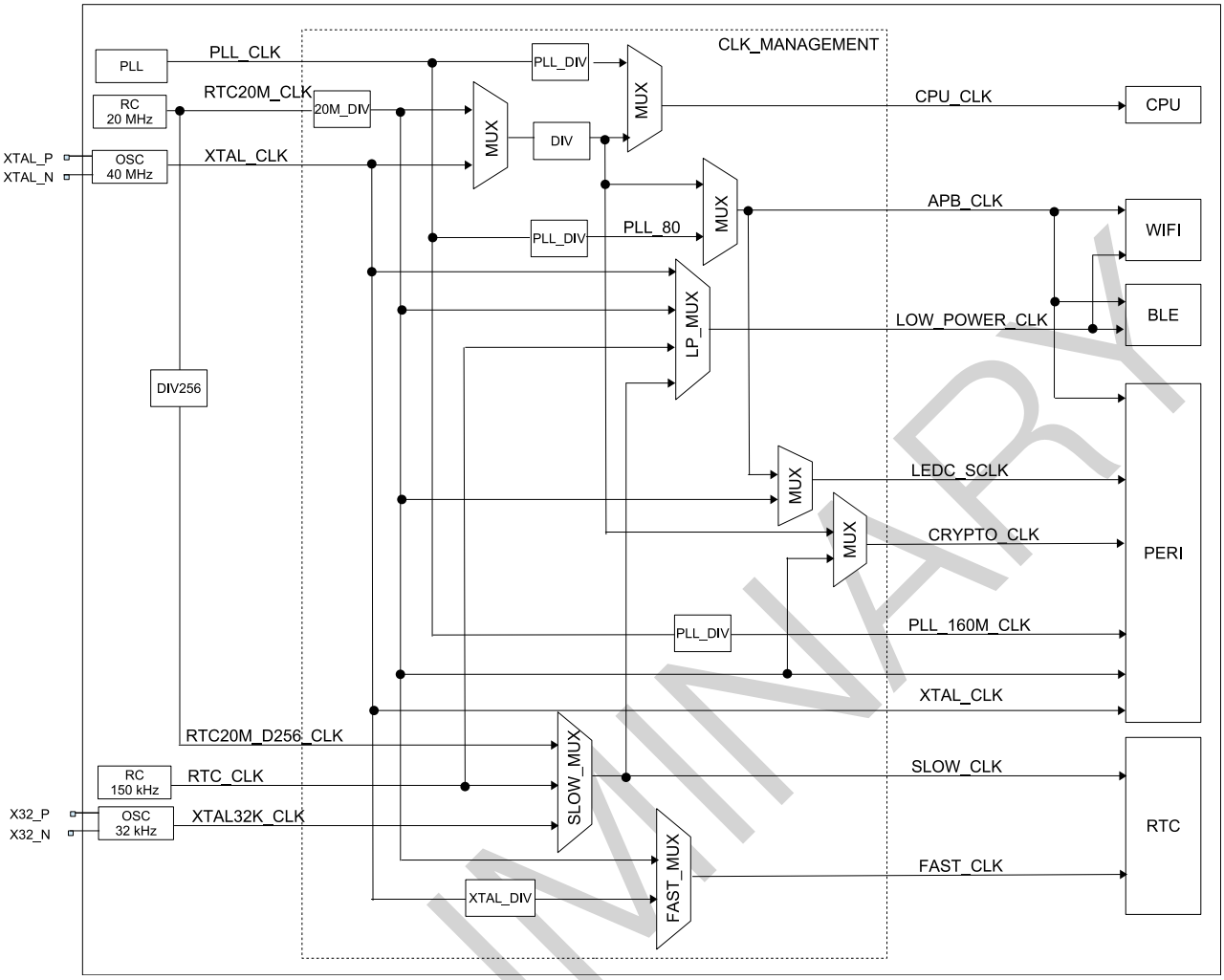


图 1-2. 系统时钟

1.2.3 特性

ESP32-C3 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
  - RTC20M_CLK：20 MHz 内部时钟，频率可调
  - RTC20M_D256_CLK：由 RTC20M_CLK 经 256 分频所得，频率为 RTC20M_CLK/256。当 RTC20M_CLK 的初始频率为 20 MHz 时，该时钟以 78.125 kHz 的频率运行
  - RTC_CLK：150 kHz 内部低功耗时钟，频率可调

PRELIMINARY

1.2.4 功能描述

1.2.4.1 CPU 时钟

如图 1-2 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 160 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、RTC20M_CLK 或 XTAL_CLK 作为 CPU_CLK 的时钟源。具体请参考表 1-2 和表 1-3。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 1-2. CPU_CLK 时钟源选择

SYSTEM_SOC_CLK_SEL 值	时钟源
0	XTAL_CLK
1	PLL_CLK
2	RTC20M_CLK

表 1-3. CPU_CLK 时钟频率

时钟源	SEL_0*	SEL_1*	SEL_2*	CPU 时钟频率
XTAL_CLK	0	-	-	CPU_CLK = XTAL_CLK/(SYSTEM_PRE_DIV_CNT + 1) SYSTEM_PRE_DIV_CNT 默认值为 1，范围 0 ~ 1023。
PLL_CLK (480 MHz)	1	1	0	CPU_CLK = PLL_CLK/6 CPU_CLK 频率为 80 MHz。
PLL_CLK (480 MHz)	1	1	1	CPU_CLK = PLL_CLK/3 CPU_CLK 频率为 160 MHz。
PLL_CLK (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。
RTC20M_CLK	2	-	-	CPU_CLK = RTC20M_CLK/(SYSTEM_PRE_DIV_CNT + 1) SYSTEM_PRE_DIV_CNT 默认值为 1，范围 0 ~ 1023。

* 寄存器 SYSTEM_SOC_CLK_SEL 的值；
* 寄存器 SYSTEM_PLL_FREQ_SEL 的值；
* 寄存器 SYSTEM_CPUPERIOD_SEL 的值。

1.2.4.2 外设时钟

外设所需要的时钟包括 APB_CLK、CRYPTO_CLK、PLL_160M_CLK、LEDC_SCLK、XTAL_CLK 和 RTC20M_CLK。表 1-4 列出了接入各个外设的时钟。

PRELIMINARY

表 1-4. 外设时钟

Peripheral	XTAL_CLK	APB_CLK	PLL_160M_CLK	(RTC) FAST_CLK	RTC20M_CLK	CRYPTO_CLK	LEDC_SCLK
TIMG	Y	Y					
I2S	Y		Y				
UHCI		Y					
UART	Y	Y			Y		
RMT	Y	Y			Y		
I2C	Y				Y		
SPI	Y	Y					
eFuse Controller				Y			
SARADC		Y					
Temperature Sensor	Y				Y		
USB		Y					
CRYPTO						Y	
TWAI Controller		Y					
LEDC	Y	Y	Y		Y		Y
SYS_TIMER	Y	Y					

PRELIMINARY

APB_CLK 时钟

如表 1-5 所示，APB_CLK 的频率由 CPU_CLK 的时钟源决定。

表 1-5. APB_CLK 时钟

CPU_CLK 时钟源	APB_CLK 频率
PLL_CLK	80 MHz
XTAL_CLK	CPU_CLK
RTC20M_CLK	CPU_CLK

CRYPTO_CLK 时钟

如表 1-6 所示，CRYPTO_CLK 的频率由 CPU_CLK 的时钟源决定。

表 1-6. CRYPTO_CLK 时钟

CPU_CLK 时钟源	CRYPTO_CLK 频率
PLL_CLK	160 MHz
XTAL_CLK	CPU_CLK
RTC20M_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。

LEDC_SCLK 时钟

LEDC 模块能将 RTC20M_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，其它外设都将停止工作（APB_CLK 关闭），但是 LEDC 仍然可以通过 RTC20M_CLK 来正常工作。

1.2.4.3 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、RTC20M_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。

1.2.4.4 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 RTC20M_D256_CLK，用于驱动功耗管理模块。
FAST_CLK 允许选择 XTAL_CLK 的分频时钟或 RTC20M_CLK 的分频时钟，用于驱动片上传感器模块。

PRELIMINARY

2 随机数发生器

2.1 概述

ESP32-C3 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。

2.2 主要特性

ESP32-C3 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样。

2.3 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**。

- **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。
- RTC20M_CLK 是一种**异步时钟源**，会产生电路亚稳态。这种亚稳态也可以作为随机数种子[2]，进入随机数生成器。

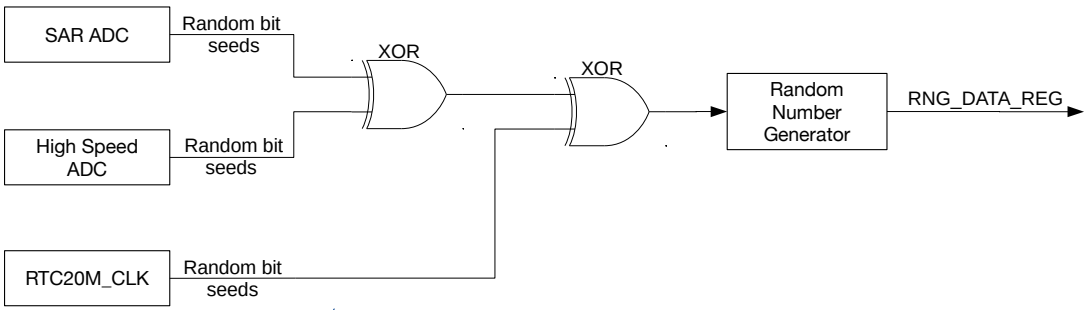


图 2-1. 噪声源

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 1 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 1 MHz。

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。

2.4 编程指南

在使用 ESP32-C3 的随机数生成器时，应该至少保证 SAR ADC 或高速 ADC[1] 或 RTC20M_CLK 处于使能状态[2]，否则可能会导致产生伪随机数，应注意避免。其中，

PRELIMINARY

- SAR ADC 受控于 DIG ADC 控制器。
- 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：

1. 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。
2. RTC20M_CLK 时钟仅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足够大的熵值，仍建议在使用随机数发生器时至少保证 SAR ADC 或高速 ADC 处于工作状态。

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 2.3 小节的介绍。

2.5 寄存器列表

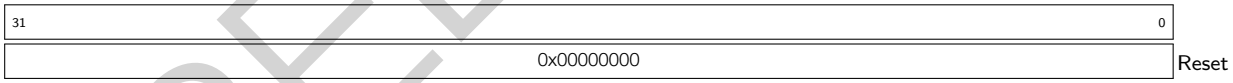
请注意，下表中的地址都是相对于随机数发生器基地址的地址偏移量（相对地址），详见章节 3 *系统和存储器* 中的表 3-4。

名称	描述	地址	访问
RNG_DATA_REG	随机数数据	0x00B0	只读

2.6 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 3 *系统和存储器* 中的表 3-4。

Register 2.1. RNG_DATA_REG (0x00B0)



RNG_DATA 随机数来源。（只读）

PRELIMINARY

3 系统和存储器

3.1 概述

ESP32-C3 是一个超低功耗和高度集成的系统，它集成了一颗 RISC-V 32 位单核处理器，四级流水线架构，主频高达 160 MHz。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。

3.2 主要特性

- **地址空间**
  - 792 KB 内部存储器指令地址空间
  - 552 KB 内部存储器数据地址空间
  - 836 KB 外设地址空间
  - 8 MB 外部存储器指令虚地址空间
  - 8 MB 外部存储器数据虚地址空间
  - 384 KB 内部 DMA 地址空间
- **内部存储器**
  - 384 KB 内部 ROM
  - 400 KB 内部 SRAM
  - 8 KB RTC 存储器
- **外部存储器**
  - 最大支持 16 MB 片外 flash
- 外设空间
  - 总计 35 个模块/外设
- **GDMA**
  - 7 个具有 GDMA 功能的模块/外设

图 3-1 描述了系统结构与地址映射结构。

PRELIMINARY

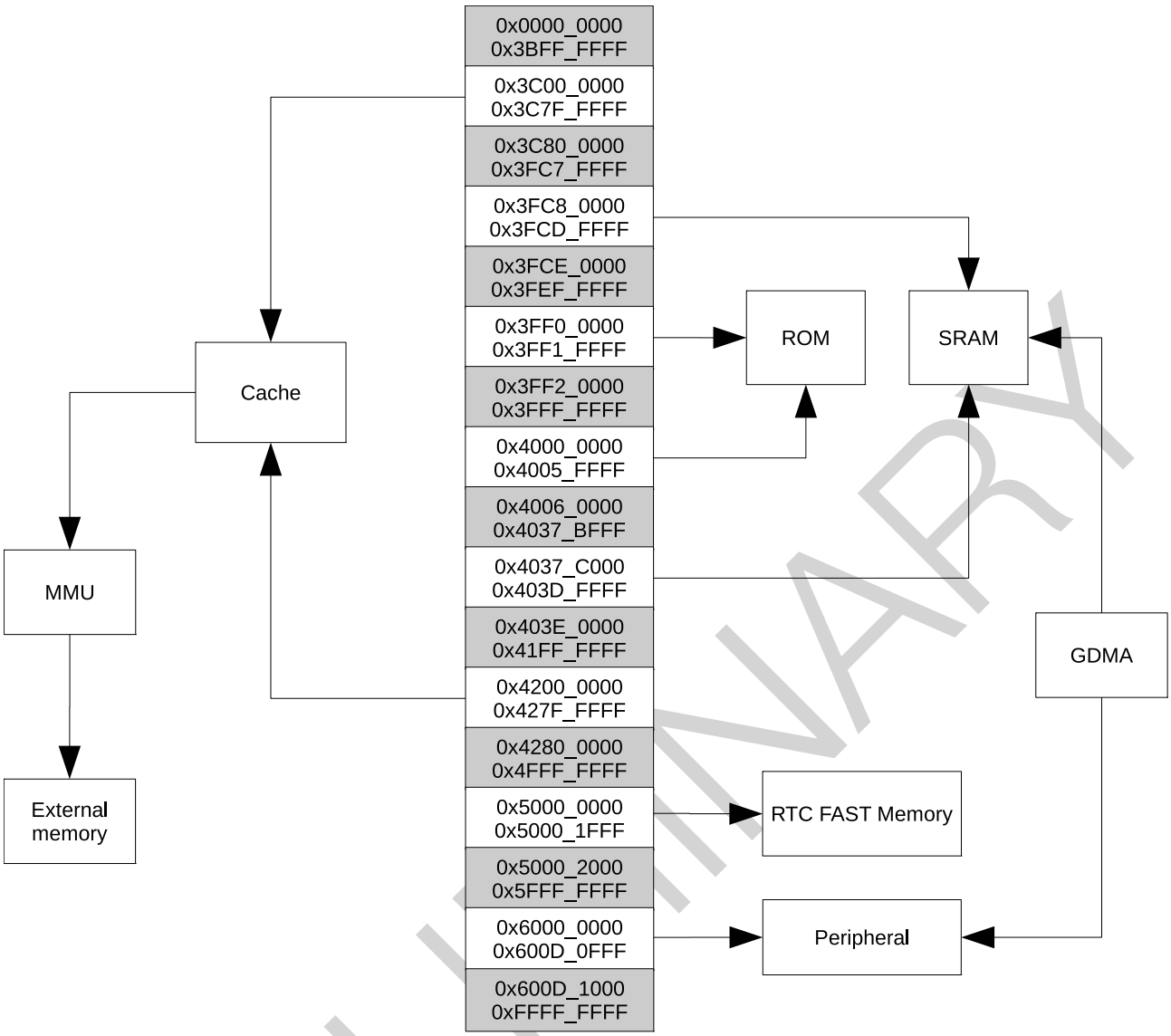


图 3-1. 系统结构与地址映射结构

说明：

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

3.3 功能描述

3.3.1 地址映射

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 字节的数据访问。CPU 也可以通过指令总线进行数据访问，但只能是 4 字节对齐的访问。

CPU 能够：

PRELIMINARY

- 通过数据总线与指令总线直接访问内部存储器；
- 通过 Cache 访问映射到虚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 3-1 描述了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 3-1. 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	0x3C7F_FFFF	8 MB	外部存储器
	0x3C80_0000	0x3FC7_FFFF		保留
数据	0x3FC8_0000	0x3FCD_FFFF	384 KB	内部存储器
	0x3FCE_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	0x4037_BFFF		保留
指令	0x4037_C000	0x403D_FFFF	400 KB	内部存储器
	0x403E_0000	0x41FF_FFFF		保留
指令	0x4200_0000	0x427F_FFFF	8 MB	外部存储器
	0x4280_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	0xFFFF_FFFF		保留

3.3.2 内部存储器

ESP32-C3 的内部存储器包含如下三种类型：

- 内部 ROM (384 KB)：内部 ROM 是只读存储器，不可编程。其中存放有一些系统底层软件的 ROM 代码（软件指令和一些只读数据）。
- 内部 SRAM (400 KB)：内部静态存储器（SRAM）是易失性存储器，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。
  - 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 存储器 (8 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。

PRELIMINARY

- RTC 快速存储器 (8 KB): RTC 快速存储器只可以被 CPU 访问，通常用来存放一些在休眠模式下仍需保持的程序指令和数据。

基于上述对三种类型的内部存储器的描述，ESP32-C3 的内部存储器可以被分为三个部分：内部 ROM (384 KB)、内部 SRAM (400 KB)、RTC 快速存储器 (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 通过数据总线访问），据此内部存储器可以被区分的更加细致。表 3-2 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 3-2. 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	内部 ROM 1
	0x3FC8_0000	0x3FCD_FFFF	384 KB	内部 SRAM 1
指令	0x4000_0000	0x4003_FFFF	256 KB	内部 ROM 0
	0x4004_0000	0x4005_FFFF	128 KB	内部 ROM 1
	0x4037_C000	0x4037_FFFF	16 KB	内部 SRAM 0
	0x4038_0000	0x403D_FFFF	384 KB	内部 SRAM 1
数据/指令	0x5000_0000	0x5000_1FFF	8 KB	RTC 快速存储器

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。

1. 内部 ROM 0

内部 ROM 0 的容量为 256 KB，只读。如表 3-2 所示，CPU 只可以通过指令总线地址段 0x4000_0000 ~ 0x4003_FFFF 访问这部分存储器。

2. 内部 ROM 1

内部 ROM 1 的容量为 128 KB，只读。如表 3-2 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内部 ROM 1 是指：地址 04004_0000 与 0x3FF0_0000 访问到相同的字，0x4004_0004 与 0x3FF0_0004 访问到相同的字，0x4004_0008 与 0x3FF0_0008 访问到相同的字，以此类推（下文的“同序访问”也参照此描述）。

3. 内部 SRAM 0

内部 SRAM 0 的容量为 16 KB，可读可写。如表 3-2 所示，CPU 只可以通过指令总线访问这部分存储器。

通过权限管理，这部分存储器可以被配置为指令缓存，用来缓存外部存储器的指令或只读数据。此时，已被配置为指令缓存的部分不可以被 CPU 访问。

4. 内部 SRAM 1

内部 SRAM 1 容量为 384 KB，可读可写。如表 3-2 所示，CPU 可以通过数据或指令总线同序访问。

5. RTC 快速存储器

PRELIMINARY

RTC 快速存储器容量为 8 KB，为可读可写的 SRAM。如表 3-2 所示，CPU 可以通过数据/指令总线的共用地址段 0x5000_0000 ~ 0x5000_1FFF 访问这部分存储器。

3.3.3 外部存储器

ESP32-C3 支持以 SPI、Dual SPI、Quad SPI、QPI 等接口形式连接片外 flash。ESP32-C3 还支持基于 XTS-AES 算法的硬件手动加密和自动解密功能，从而保护开发者片外 flash 中的程序和数据。

3.3.3.1 外部存储器地址映射

CPU 借助缓存（Cache）来访问外部存储器。Cache 将根据内存管理单元 (Memory Management Unit, MMU) 中的信息把 CPU 的地址映射为访问片外存储的实地址。经过地址映射，ESP32-C3 最大支持 16 MB 的片外 flash。

通过高速缓存，ESP32-C3 可支持以下地址空间映射。请注意，指令总线地址空间（8 MB）和数据总线地址空间 (8 MB) 是共用的。

- 8 MB 的指令总线地址空间以 64 KB 为单位映射到片外 flash。
- 8 MB 的数据总线（只读）地址空间以 64 KB 为单位映射到片外 flash。

表 3-3 列出了在访问外部存储器时 CPU 的数据总线与指令总线与 Cache 的对应关系。

表 3-3. 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（只读）	0x3C00_0000	0x3C7F_FFFF	8 MB	Uniform Cache
指令	0x4200_0000	0x427F_FFFF	8 MB	Uniform Cache

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。

3.3.3.2 高速缓存

如图 3-2 所示，ESP32-C3 采用一个只读的统一 cache，为 8 路组相联，容量为 16 KB，块大小为 32 字节。当 cache 处于工作状态时，将占用部分内部存储空间（参见第 3.3.2 节关于内部 SRAM 0 的描述）。

指令总线和数据总线可以同时访问该 cache，但此时 cache 只能对其中一个作出相应。当 cache 缺失时，cache 控制器会向外部存储器发起请求。

PRELIMINARY

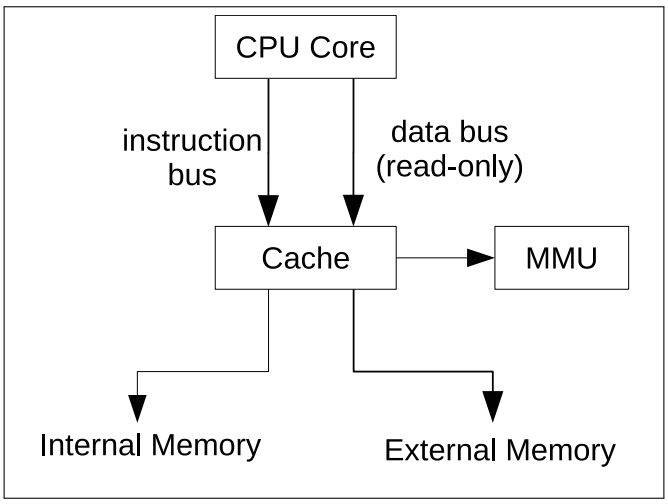


图 3-2. Cache 系统结构

3.3.3.3 Cache 操作

ESP32-C3 cache 支持如下几种操作：

1. **失效 (Invalidate)**：该操作用于删除 cache 中的有效数据。该操作完成后，删除的数据将仅存于外部存储器中。如果 CPU 接着去访问该数据，那么需要访问外部存储器。该操作包括两种类型：自动失效 (Auto-Invalidate) 和手动失效 (Manual-Invalidate)。手动失效仅对 cache 中落入指定区域的地址对应的数据做失效处理，而自动失效会对 cache 中的所有数据做失效处理。
2. **预取 (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)，手动预取是指硬件按软件指定的虚地址预取一段连续的数据；自动预取是指硬件根据当前命中/缺失（取决于配置）的地址，自动地预取一段连续的数据。
3. **锁定/解锁 (Lock/Unlock)**：该操作用于保护 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。

   请注意，手动失效操作只对未被锁定的数据起作用。如果想对已锁定的数据执行手动失效操作，请先解锁这些数据。

3.3.4 GDMA 地址空间

ESP32-C3 中的 GDMA (General Direct Memory Access) 外设可提供直接内存访问（Direct Memory Access, DMA) 服务，包括：

- 内部存储器中不同位置的数据搬运；
- 模块/外设和内部存储器之间的数据搬运。

GDMA 可以通过与数据总线完全相同的地址读写内部 SRAM 1，即 GDMA 通过地址访问内部 SRAM 1。请注意，GDMA 无法访问被 cache 占用的内部存储器。

ESP32-C3 中共有 7 个外设/模块可以和 GDMA 联合工作。如图 3-3 所示，其中的 7 根竖线依次对应这 7 个具有 GDMA 功能的外设/模块，横线表示 GDMA 的某一通道（可为任意通道），竖线与横线的交点表示对应外

PRELIMINARY

设/模块可以访问 GDMA 的某一通道。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。

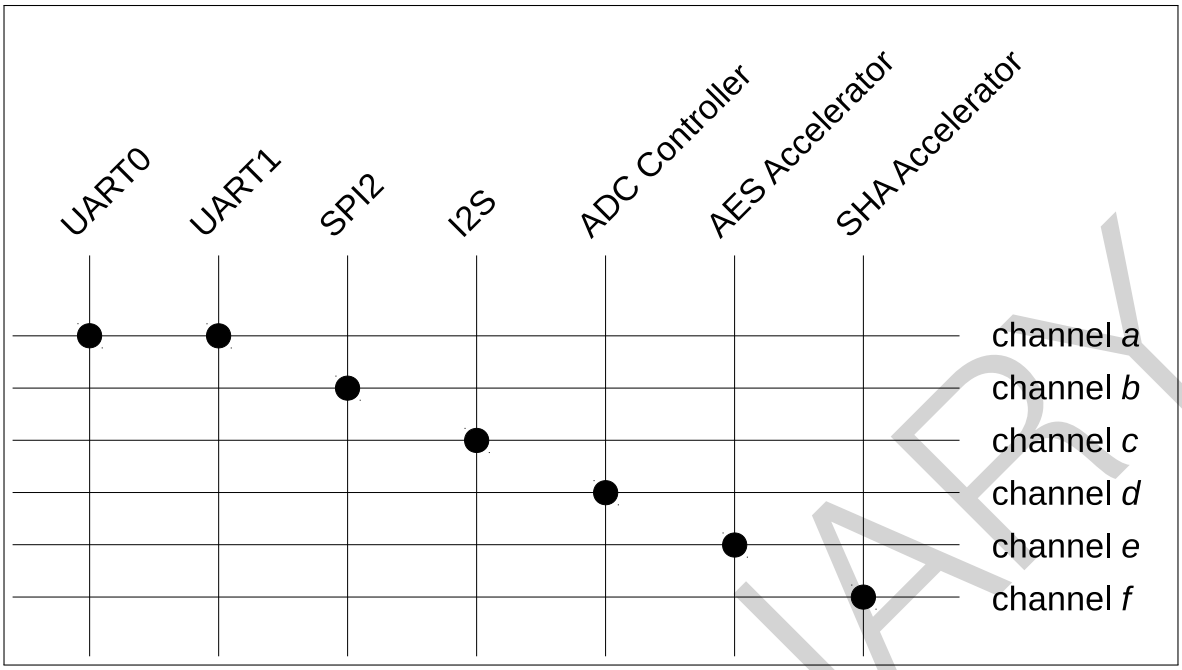


图 3-3. 具有 GDMA 功能的外设/模块

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。

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。

3.3.5 模块/外设

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

3.3.5.1 模块/外设地址空间映射

表 3-4 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”（包括低位地址和高位地址）栏中的两列数值共同决定了对应模块/外设的地址空间。

表 3-4. 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 Controller 0	0x6000_0000	0x6000_0FFF	4 KB	
Reserved	0x6000_1000	0x6000_1FFF		
SPI Controller 1	0x6000_2000	0x6000_2FFF	4 KB	
SPI Controller 0	0x6000_3000	0x6000_3FFF	4 KB	
GPIO	0x6000_4000	0x6000_4FFF	4 KB	
Reserved	0x6000_5000	0x6000_6FFF		
TIMER	0x6000_7000	0x6000_7FFF	4 KB	

PRELIMINARY

目标	边界地址		容量	说明
	低位地址	高位地址		
Low-Power Management	0x6000_8000	0x6000_8FFF	4 KB	
IO MUX	0x6000_9000	0x6000_9FFF	4 KB	
Reserved	0x6000_A000	0x6000_FFFF		
UART Controller 1	0x6001_0000	0x6001_0FFF	4 KB	
Reserved	0x6001_1000	0x6001_2FFF		
I2C Controller	0x6001_3000	0x6001_3FFF	4 KB	
UHCI0	0x6001_4000	0x6001_4FFF	4 KB	
Reserved	0x6001_5000	0x6001_5FFF		
Remote Control Peripheral	0x6001_6000	0x6001_6FFF	4 KB	
Reserved	0x6001_7000	0x6001_8FFF		
LED Control PWM	0x6001_9000	0x6001_9FFF	4 KB	
eFuse Controller	0x6001_A000	0x6001_AFFF	4 KB	
Reserved	0x6001_B000	0x6001_EFFF		
Timer Group 0	0x6001_F000	0x6001_FFFF	4 KB	
Timer Group 1	0x6002_0000	0x6002_0FFF	4 KB	
Reserved	0x6002_1000	0x6002_2FFF		
System Timer	0x6002_3000	0x6002_3FFF	4 KB	
SPI Controller 2	0x6002_4000	0x6002_4FFF	4 KB	
Reserved	0x6002_5000	0x6002_5FFF		
APB Controller	0x6002_6000	0x6002_6FFF	4 KB	
Reserved	0x6002_7000	0x6002_AFFF		
Two-wire Automotive Interface	0x6002_B000	0x6002_BFFF	4 KB	
Reserved	0x6002_C000	0x6002_CFFF		
I2S Controller	0x6002_D000	0x6002_DFFF	4 KB	
Reserved	0x6002_E000	0x6003_9FFF		
AES Accelerator	0x6003_A000	0x6003_AFFF	4 KB	
SHA Accelerator	0x6003_B000	0x6003_BFFF	4 KB	
RSA Accelerator	0x6003_C000	0x6003_CFFF	4 KB	
Digital Signature	0x6003_D000	0x6003_DFFF	4 KB	
HMAC Accelerator	0x6003_E000	0x6003_EFFF	4 KB	
GDMA Controller	0x6003_F000	0x6003_FFFF	4 KB	
ADC Controller	0x6004_0000	0x6004_0FFF	4 KB	
Reserved	0x6004_1000	0x6002_FFFF		
USB Serial/JTAG Controller	0x6004_3000	0x6004_3FFF	4 KB	
Reserved	0x6004_4000	0x600B_FFFF		
System Registers	0x600C_0000	0x600C_0FFF	4 KB	
Sensitive Register	0x600C_1000	0x600C_1FFF	4 KB	
Interrupt Matrix	0x600C_2000	0x600C_2FFF	4 KB	
Reserved	0x600C_3000	0x600C_3FFF		
Configure Cache	0x600C_4000	0x600C_BFFF	32 KB	
External Memory Encryption and Decryption	0x600C_C000	0x600C_CFFF	4 KB	

PRELIMINARY

目标	边界地址		容量	说明
	低位地址	高位地址		
Reserved	0x600C_D000	0x600C_DFFF		
Assist Debug	0x600C_E000	0x600C_EFFF	4 KB	
Reserved	0x600C_F000	0x600C_FFFF		
World Controller	0x600D_0000	0x600D_0FFF	4 KB	

PRELIMINARY

4 IO MUX 和 GPIO 交换矩阵 (GPIO, IO_MUX)

4.1 概述

ESP32-C3 芯片有 22 个物理通用输入输出管脚 (GPIO Pad)。每个管脚都可用作一个通用 IO，或连接一个内部的外设信号。利用 GPIO 交换矩阵和 IO MUX，可配置外设模块的输入信号来源于任何的 IO 管脚，并且外设模块的输出信号也可连接到任意 IO 管脚。这些模块共同组成了芯片的 IO 控制。

注意：这 22 个物理 GPIO 管脚的编号为：0 ~ 21。

4.2 主要特性

GPIO 交换矩阵主要特性

- 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。由 DRV、IE、OE、WPU、WPD 等信号控制；
- 49 个外设输入信号可以选择任意一个 GPIO 管脚的输入信息。由 SIG_IN_SEL 和 IE 等信号控制；
- 每个 GPIO 管脚的输出信号可以来自 125 个外设输出信号的任意一个。由 SIG_OUT_SEL 和 OE 等信号控制；
- 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 主要特性

- 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。

4.3 结构概览

本小节主要介绍 IO MUX 以及 GPIO 交换矩阵的架构，其中：

- 图 4-1 简要展示了 IO MUX 和 GPIO 交换矩阵的工作流程；
- 图 4-2 详细展示了 IO MUX 和 GPIO 交换矩阵将信号引入外设和引出至管脚的具体过程；
- 图 4-3 展示了 GPIO 管脚的接口逻辑。

PRELIMINARY

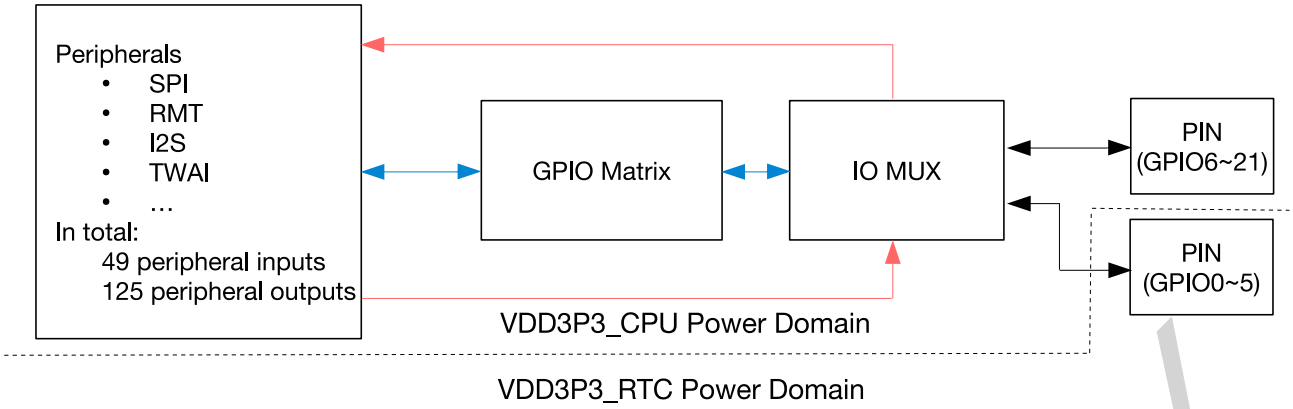


图 4-1. IO MUX 和 GPIO 交换矩阵框图（简图）

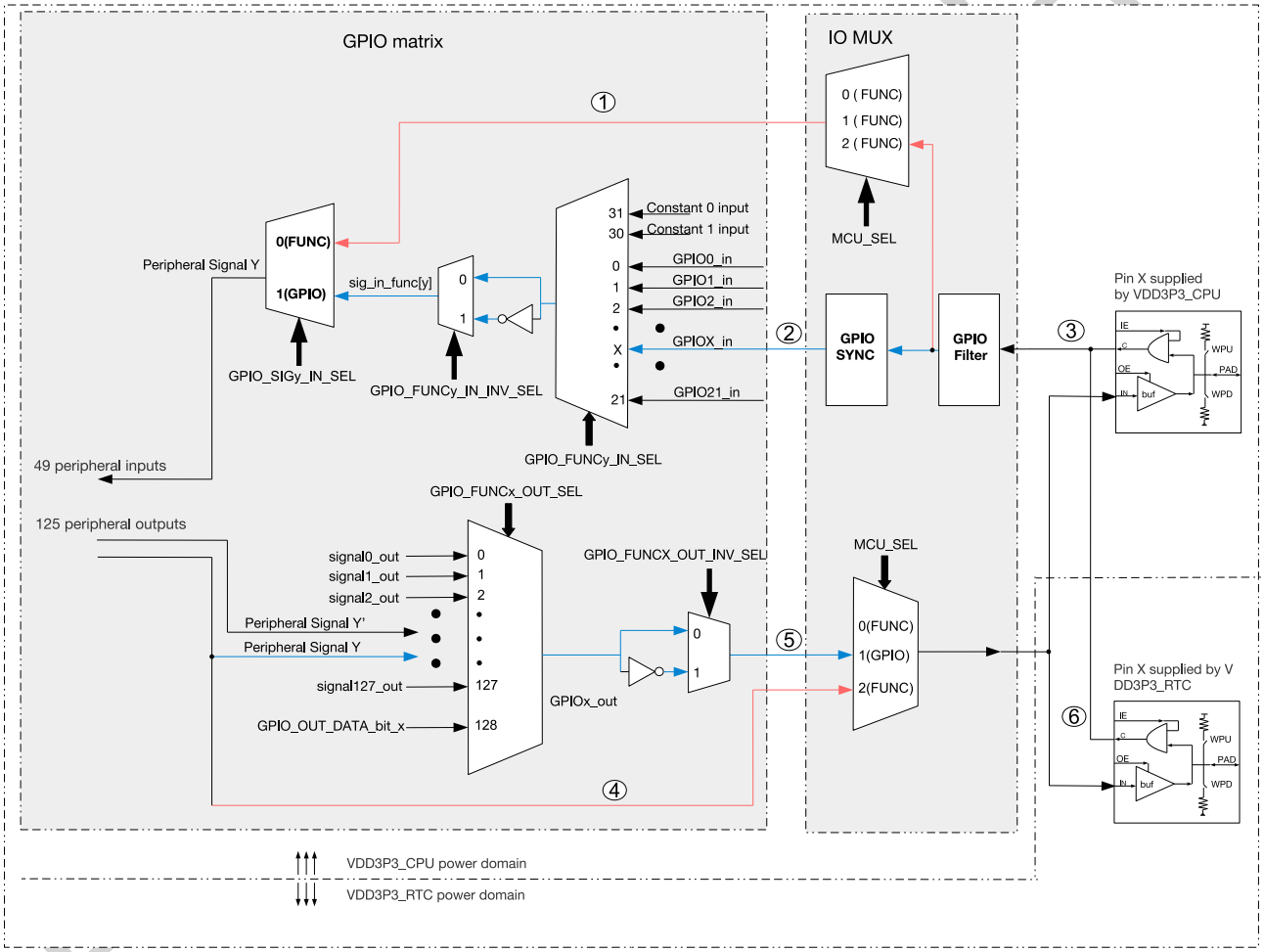


图 4-2. IO MUX 和 GPIO 交换矩阵框图（详图）

1. 注意，外设输入信号中，仅有索引号为 0 ~ 3、6 ~ 7、9 ~ 10、63 ~ 68 的输入信号可以直接通过 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-C3 共有 22 个 GPIO 管脚，因此从 GPIO SYNC 进入到 GPIO 交换矩阵的输入共用 22 个；
3. 位于 VDD3P3_CPU 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 ~ 59, 63~ 127) 可通过 IO MUX 直连管脚。可以实现直连功能的信号见表 4-1；

5. 从 GPIO 交换矩阵到 IO MUX 的输出共有 22 个，对应 GPIO X：0 ~ 21
6. 位于 VDD3P3_RTC 电源域的管脚由 IE、OE、WPU 和 WPD 控制。

图 4-3 展示了芯片 PAD 的内部结构，即芯片逻辑与 GPIO 管脚之间的电气接口。

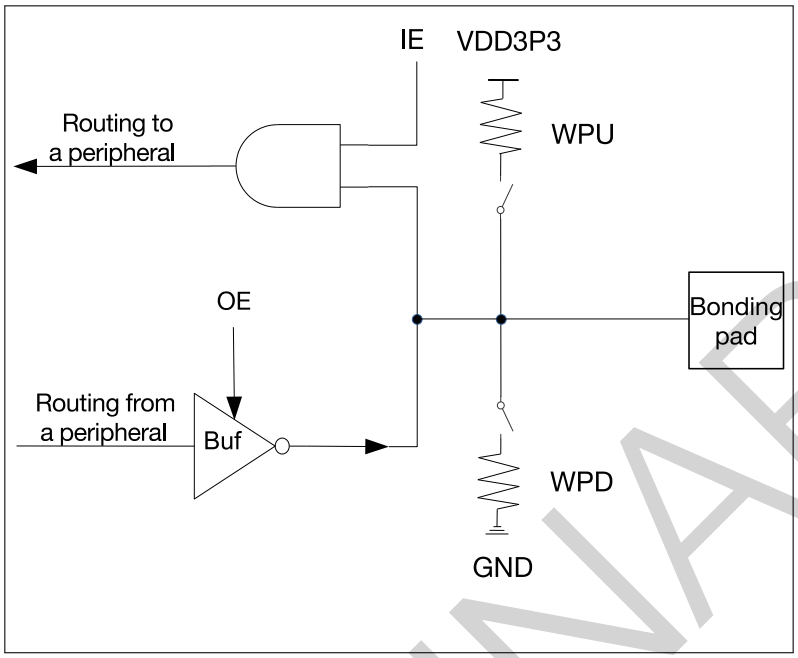


图 4-3. 管脚内部结构

- IE：输入使能
- OE：输出使能
- WPU：内部弱上拉
- WPD：内部弱下拉

4.4 通过 GPIO 交换矩阵的外设输入

4.4.1 概述

为实现通过 GPIO 交换矩阵接收外部输入信号，需要配置 GPIO 交换矩阵从 22 个 GPIO (0 ~ 21) 中获取外部输入信号，见交换矩阵表格 4-1。并需要配置外设输入选择通过 GPIO 交换矩阵接收输入信号。

4.4.2 信号同步

如图 4-2 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。

PRELIMINARY

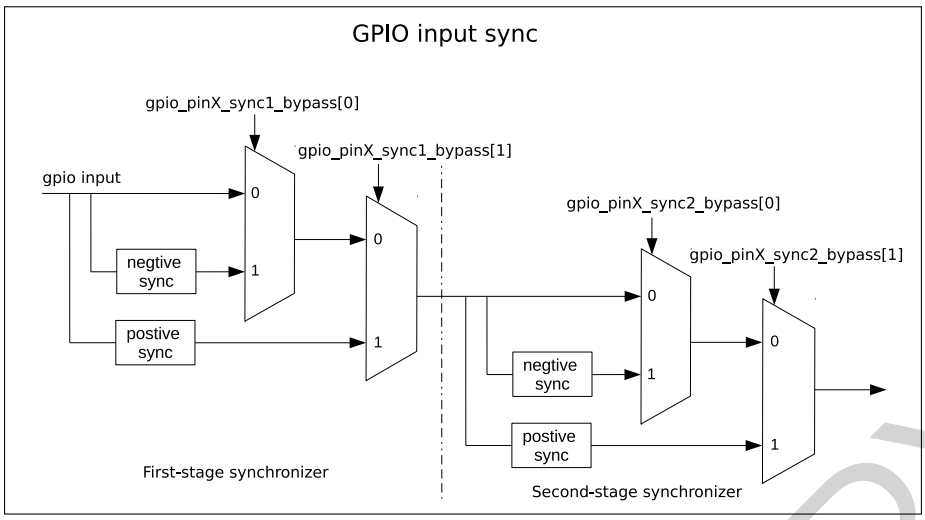


图 4-4. GPIO 输入经 APB 时钟上升沿或下降沿同步

GPIO SYNC 模块的功能如图 4-4 所示。其中，negative sync 为 GPIO 输入经过 APB 时钟的下降沿同步，positive sync 为 GPIO 输入经过 APB 时钟上升沿同步。

4.4.3 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 位，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_GPIO*n*_FILTER_EN 使能 GPIO 管脚的输入信号滤波功能，如图 4-5 所示。只有当输入信号的有效宽度大于两个时钟周期时，输入信号才会被采样。否则，输入信号将会被滤掉。

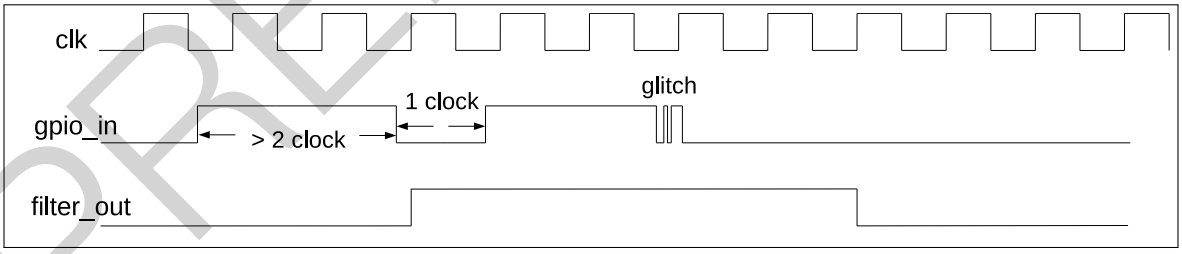


图 4-5. GPIO 输入信号滤波时序图

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 4-4 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 4-4 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_GPIO*x*_REG，过程如下：
   - 置位 IO_MUX_GPIO*x*_FUN_IE 使能输入[2]。

PRELIMINARY

- 置位或清零 IO_MUX_GPIO*x*_FUN_WPU 和 IO_MUX_GPIO*x*_FUN_WPD，使能或关闭内部上拉/下拉电阻。

例如，要把 I2S MCLK 输入信号 [3]（I2S_MCLK_in，信号索引号 12）绑定到 GPIO7，请按照以下步骤操作。注意，GPIO7 也叫做 MTDO 管脚。

1. 置位 GPIO_FUNC12_IN_SEL_CFG_REG 寄存器的 GPIO_SIG12_IN_SEL 位，使能通过 GPIO 交换矩阵接收外部输入信号；
2. 配置 GPIO_FUNC12_IN_SEL_CFG_REG 寄存器中的 GPIO_FUNC12_IN_SEL 位为 7，即选择管脚 GPIO7；
3. 置位 IO_MUX_GPIO7_REG 寄存器中 IO_MUX_GPIO7_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输入信号；
2. 置位 GPIO_FUNC*y*_IN_INV_SEL 可以把输入信号取反；
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：
   - 设置 GPIO_FUNC*y*_IN_SEL 为 0x1F，则输入信号恒为 0；
   - 设置 GPIO_FUNC*y*_IN_SEL 为 0x1E，则输入信号恒为 1。

4.4.4 简单 GPIO 输入

GPIO_IN_REG 寄存器存储着每个 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。但需要配置 GPIO 管脚 *X* 对应的 IO_MUX_GPIO*x*_REG 管脚中 IO_MUX_GPIO*x*_FUN
_IE 位以使能输入，如章节 4.4.2 所述。

4.5 通过 GPIO 交换矩阵的外设输出

4.5.1 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 ~ 59, 63~ 127) 的外设信号输出到 22 个 GPIO (0 ~ 21) 管脚。外设信号见表 4-1。

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 GPIO 信号就能连接到相应管脚。

说明：
输出索引号为 97 ~ 100 的外设信号，没有连接至外设，可配置为从一个 GPIO 管脚输出后，直接由另一个 GPIO 管脚输入（索引号：97 ~ 100）。

4.5.2 功能描述

如图 4-2 所示，对于信号输出，125 个输出信号 (0 ~ 59, 63~ 127) 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] 的步骤如下：

1. 在 GPIO 交换矩阵中配置 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 W1TS（写 1 置位）和 W1TC（写 1 清零）寄存器来更新 GPIO_ENABLE_REG 寄存器中的值：

- 设置 GPIO_FUNCx_OUT_SEL_CFG_REG 寄存器的 GPIO_FUNCx_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_FUNCx_OUT_SEL_CFG_REG 寄存器中 GPIO_FUNCx_OEN_SEL 字段置位；同时需要将 GPIO_ENABLE_W1TS_REG 中的相应位置位。或者，将 GPIO_FUNCx_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 4-1 中 “GPIO_FUNCn_OEN_SEL = 0 时输出信号的输出使能信号” 一栏的 SPIQ_oe 信号。
- 清零 GPIO_ENABLE_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_PINx_REG 寄存器中 GPIO_PINx_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 的 IO_MUX_GPIOx_REG 的过程如下：
   - 配置 GPIO 管脚 X 的 IO_MUX_GPIOx_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_GPIOx_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_GPIOx_FUN_WPU 和 IO_MUX_GPIOx_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_FUNCx_OUT_INV_SEL 可以把输出的信号取反。

4.5.3 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_FUNCn_OUT_SEL 寄存器为特定的外设索引值 128 (0x80)；
- 设置 GPIO_OUT_REG 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[25:22] 无效。
- 推荐使用相应的 W1TS 和 W1TC 寄存器，例如 GPIO_OUT_W1TS/GPIO_OUT_W1TC 来置位/清零 GPIO_OUT_REG。

PRELIMINARY

4.5.4 Sigma Delta 调制输出 (SDM)

4.5.4.1 功能描述

125 个外设输出信号中有四个信号（索引：55 ~ 58）支持 1-bit 二阶 SDM 调制输出。上述四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 SDM 调制的转换公式如下：

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

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 GPIOSD_FUNCTION_CLK_EN 使能调制器时钟；
- 配置 GPIOSD_SD*n*_PRESCALE 实现分频。*n* 取值范围为 0 ~ 3，对应四个信号通道。

分频后的时钟周期为调制器输出单位脉冲的周期。

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

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

PDM 信号占空比计算公式为：

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

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

4.5.4.2 配置方法

SDM 的配置方法如下：

- 将 SDM 输出经 GPIO 交换矩阵连接至相应管脚，见 4.5.2 章节；
- 置位 GPIOSD_FUNCTION_CLK_EN，使能 SDM 时钟；
- 配置 GPIOSD_SD*n*_PRESCALE 寄存器设置时钟分频系数；
- 配置 GPIOSD_SD*n*_IN 寄存器设置 SDM 输出信号的占空比。

4.6 IO MUX 的直接输入输出功能

4.6.1 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。

PRELIMINARY

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。

4.6.2 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_GPIO*n*_MCU_SEL 必须设置为相应的管脚功能，章节 4.11 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_GPIO*n*_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。

4.7 GPIO 管脚的模拟功能

ESP32-C3 部分 GPIO 管脚具有模拟功能。用于模拟功能时，请确保已按照下述方法关闭了上拉电阻和下拉电阻：

- 设置 IO_MUX_GPIO*n*_MCU_SEL 为 1，同时清零 IO_MUX_GPIO*n*_FUN_IE、IO_MUX_GPIO*n*_FUN_WPU、IO_MUX_GPIO*n*_FUN_WPD；
- 置位 GPIO_ENABLE_W1TC*[n]*，清除输出使能。

表 4-4 列出了 ESP32-C3 管脚的模拟功能。

4.8 管脚 Hold 特性

每个 GPIO 管脚（包括 RTC 管脚 GPIO0 ~ GPIO5）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 (GPIO6 ~ 21)，若要在 Deep-sleep 中保持管脚输入输出的状态值，需要在掉电之前将寄存器 RTC_CNTL_DIG_PAD_HOLD_REG 中的 RTC_CNTL_DIG_PAD_HOLD*[n]* 位置 1。在芯片被唤醒后，若要关闭 Hold 功能，可将寄存器 RTC_CNTL_DIG_PAD_HOLD*[n]* 设置为 0。
- 对于 RTC 管脚 (GPIO0 ~ 5)，管脚的输入输出值由寄存器 RTC_CNTL_RTC_PAD_HOLD_REG 中的相应位控制。用户可置位或清除相应位来实现 Hold 或 Unhold 管脚输入输出值。

4.9 GPIO 管脚供电和电源管理

PRELIMINARY

4.9.1 GPIO 管脚供电

GPIO 管脚供电请参考《ESP32-C3 规格书》中管脚定义章节。所有管脚均可用于将芯片从 Light-sleep 中唤醒，但仅有 VDD3P3_RTC 域中的管脚 (GPIO0 ~ GPIO5) 可用于将芯片从 Deep-sleep 唤醒。

4.9.2 电源管理

ESP32-C3 的管脚可分为如下两种不同的电源域。

- VDD3P3_RTC：RTC 和 CPU 的输入电源
- VDD3P3_CPU：CPU 的输入电源

4.10 外设信号列表

表 4-1 列出了所有经由 GPIO 交换矩阵的外设输入输出信号。

请注意 GPIO_FUNC*n*_OEN_SEL 位的配置：

- GPIO_FUNC*n*_OEN_SEL = 1，则寄存器 GPIO_ENABLE_REG 中的相应位 *n* 将用于控制信号输出使能。
  - GPIO_ENABLE_REG = 0：输出关闭；
  - GPIO_ENABLE_REG = 1：输出使能；
- GPIO_FUNC*n*_OEN_SEL = 0，则输出信号的使能由外设控制，例如表 4-1 中 “GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号” 一栏的 SPIQ_oe。注意，使能信号 SPIQ_oe 可设置为 1 (1’d1) 或 0 (1’d0)，具体由外设的配置决定。如果 “GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号” 一栏中为 1’d1，则表示寄存器 GPIO_FUNC*n*_OEN_SEL 已清零，输出信号默认始终使能。

说明：

信号连续编号，但并非所有信号均有效。

- 输入信号中，仅有索引号为 0 ~ 3、6 ~ 19、28 ~ 35、40 ~ 42、45、51 ~ 54、63 ~ 68、74、77 ~ 80 和 97 ~ 100 的输入信号有效。
- 输出信号中，仅有索引号为 0 ~ 59 和 63 ~ 127 的输出信号有效。

PRELIMINARY

表 4-1. 通过 GPIO 交换矩阵输入输出的外设信号列表

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	U0RXD_in	0	yes	U0TXD_out	1'd1	yes
7	U0CTS_in	0	yes	U0RTS_out	1'd1	no
8	U0DSR_in	0	no	U0DTR_out	1'd1	no
9	U1RXD_in	0	yes	U1TXD_out	1'd1	no
10	U1CTS_in	0	yes	U1RTS_out	1'd1	no
11	U1DSR_in	0	no	U1DTR_out	1'd1	no
12	I2S_MCLK_in	0	no	I2S_MCLK_out	1'd1	no
13	I2SO_BCK_in	0	no	I2SO_BCK_out	1'd1	no
13	I2SO_WS_in	0	no	I2SO_WS_out	1'd1	no
15	I2SI_SD_in	0	no	I2SO_SD_out	1'd1	no
16	I2SI_BCK_in	0	no	I2SI_BCK_out	1'd1	no
17	I2SI_WS_in	0	no	I2SI_WS_out	1'd1	no
18	gpio_bt_priority	0	no	gpio_wlan_prio	1'd1	no
19	gpio_bt_active	0	no	gpio_wlan_active	1'd1	no
20	-	-	-	cpu_test_bu0	1'd1	no
21	-	-	-	cpu_test_bu1	1'd1	no
22	-	-	-	cpu_test_bu2	1'd1	no
23	-	-	-	cpu_test_bu3	1'd1	no

PRELIMINARY

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
24	-	-	-	cpu_test_bu4	1'd1	no
25	-	-	-	cpu_test_bu5	1'd1	no
26	-	-	-	cpu_test_bu6	1'd1	no
27	-	-	-	cpu_test_bu7	1'd1	no
28	cpu_gpio_in0	0	no	cpu_gpio_out0	cpu_gpio_out_oen0	no
29	cpu_gpio_in1	0	no	cpu_gpio_out1	cpu_gpio_out_oen1	no
30	cpu_gpio_in2	0	no	cpu_gpio_out2	cpu_gpio_out_oen2	no
31	cpu_gpio_in3	0	no	cpu_gpio_out3	cpu_gpio_out_oen3	no
32	cpu_gpio_in4	0	no	cpu_gpio_out4	cpu_gpio_out_oen4	no
33	cpu_gpio_in5	0	no	cpu_gpio_out5	cpu_gpio_out_oen5	no
34	cpu_gpio_in6	0	no	cpu_gpio_out6	cpu_gpio_out_oen6	no
35	cpu_gpio_in7	0	no	cpu_gpio_out7	cpu_gpio_out_oen7	no
36	-	-	-	usb_jtag_tck	1'd1	no
37	-	-	-	usb_jtag_tms	1'd1	no
38	-	-	-	usb_jtag_tdi	1'd1	no
39	-	-	-	usb_jtag_tdo	1'd1	no
40	usb_extphy_vp	0	no	usb_extphy_oen	1'd1	no
41	usb_extphy_vm	0	no	usb_extphy_speed	1'd1	no
42	usb_extphy_rcv	0	no	usb_extphy_vpo	1'd1	no
43	-	-	-	usb_extphy_vmo	1'd1	no
44	-	-	-	usb_extphy_suspnd	1'd1	no
45	ext_adc_start	0	no	ledc_ls_sig_out0	1'd1	no
46	-	-	-	ledc_ls_sig_out1	1'd1	no
47	-	-	-	ledc_ls_sig_out2	1'd1	no
48	-	-	-	ledc_ls_sig_out3	1'd1	no
49	-	-	-	ledc_ls_sig_out4	1'd1	no

PRELIMINARY

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
50	-	-	-	ledc_ls_sig_out5	1'd1	no
51	rmt_sig_in0	0	no	rmt_sig_out0	1'd1	no
52	rmt_sig_in1	0	no	rmt_sig_out1	1'd1	no
53	I2CEXT0_SCL_in	1	no	I2CEXT0_SCL_out	I2CEXT0_SCL_oe	no
54	I2CEXT0_SDA_in	1	no	I2CEXT0_SDA_out	I2CEXT0_SDA_oe	no
55	-	-	-	gpio_sd0_out	1'd1	no
56	-	-	-	gpio_sd1_out	1'd1	no
57	-	-	-	gpio_sd2_out	1'd1	no
58	-	-	-	gpio_sd3_out	1'd1	no
59	-	-	-	I2SO_SD1_out	1'd1	no
60	-	-	-	-	1'd1	-
61	-	-	-	-	1'd1	-
62	-	-	-	-	1'd1	-
63	FSPICLK_in	0	yes	FSPICLK_out_mux	FSPICLK_oe	yes
64	FSPIQ_in	0	yes	FSPIQ_out	FSPIQ_oe	yes
65	FSPID_in	0	yes	FSPID_out	FSPID_oe	yes
66	FSPIHD_in	0	yes	FSPIHD_out	FSPIHD_oe	yes
67	FSPIWP_in	0	yes	FSPIWP_out	FSPIWP_oe	yes
68	FSPICS0_in	0	yes	FSPICS0_out	FSPICS0_oe	yes
69	-	-	-	FSPICS1_out	FSPICS1_oe	no
70	-	-	-	FSPICS2_out	FSPICS2_oe	no
71	-	-	-	FSPICS3_out	FSPICS3_oe	no
72	-	-	-	FSPICS4_out	FSPICS4_oe	no
73	-	-	-	FSPICS5_out	FSPICS5_oe	no
74	twai_rx	1	no	twai_tx	1'd1	no
75	-	-	-	twai_bus_off_on	1'd1	no

PRELIMINARY

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
76	-	-	-	twai_clkout	1'd1	no
77	pcmfsync_in	0	no	bt_audio0_irq	1'd1	no
78	pcmclk_in	0	no	bt_audio1_irq	1'd1	no
79	pcmdin	0	no	bt_audio2_irq	1'd1	no
80	rw_wakeup_req	0	no	ble_audio0_irq	1'd1	no
81	-	-	-	ble_audio1_irq	1'd1	no
82	-	-	-	ble_audio2_irq	1'd1	no
83	-	-	-	pcmfsync_out	pcmfsync_en	no
84	-	-	-	pcmclk_out	pcmclk_en	no
85	-	-	-	pcmdout	pcmdout_en	no
86	-	-	-	ble_audio_sync0_p	1'd1	no
87	-	-	-	ble_audio_sync1_p	1'd1	no
88	-	-	-	ble_audio_sync2_p	1'd1	no
89	-	-	-	ant_sel0	1'd1	no
90	-	-	-	ant_sel1	1'd1	no
91	-	-	-	ant_sel2	1'd1	no
92	-	-	-	ant_sel3	1'd1	no
93	-	-	-	ant_sel4	1'd1	no
94	-	-	-	ant_sel5	1'd1	no
95	-	-	-	ant_sel6	1'd1	no
96	-	-	-	ant_sel7	1'd1	no
97	sig_in_func_97	0	no	sig_in_func97	1'd1	no
98	sig_in_func_98	0	no	sig_in_func98	1'd1	no
99	sig_in_func_99	0	no	sig_in_func99	1'd1	no
100	sig_in_func_100	0	no	sig_in_func100	1'd1	no
101	-	-	-	syncerr	!efuse_dis_btlc_gpio1	no

PRELIMINARY

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
102	-	-	-	syncfound_flag	!efuse_dis_btlc_gpio1	no
103	-	-	-	evt_cntl_immediate_abort	!(efuse_dis_btlc_gpio1&efuse_dis_btlc_gpio0)	no
104	-	-	-	linklbl	!efuse_dis_btlc_gpio1&!efuse_dis_btlc_gpio0	no
105	-	-	-	data_en	!efuse_dis_btlc_gpio1&!efuse_dis_btlc_gpio0	no
106	-	-	-	data	!efuse_dis_btlc_gpio1&!efuse_dis_btlc_gpio0	no
107	-	-	-	pkt_tx_on	!efuse_dis_btlc_gpio1	no
108	-	-	-	pkt_rx_on	!efuse_dis_btlc_gpio1	no
109	-	-	-	rw_tx_on	!efuse_dis_btlc_gpio1	no
110	-	-	-	rw_rx_on	!efuse_dis_btlc_gpio1	no
111	-	-	-	evt_req_p	!(efuse_dis_btlc_gpio1&efuse_dis_btlc_gpio0)	no
112	-	-	-	evt_stop_p	!(efuse_dis_btlc_gpio1&efuse_dis_btlc_gpio0)	no
113	-	-	-	bt_mode_on	!(efuse_dis_btlc_gpio1&efuse_dis_btlc_gpio0)	no
114	-	-	-	gpio_lc_diag0	!efuse_dis_btlc_gpio1	no
115	-	-	-	gpio_lc_diag1	!efuse_dis_btlc_gpio1	no
116	-	-	-	gpio_lc_diag2	!efuse_dis_btlc_gpio1	no
117	-	-	-	ch_idx	!efuse_dis_btlc_gpio1&!efuse_dis_btlc_gpio0	no
118	-	-	-	rx_window	!efuse_dis_btlc_gpio1	no
119	-	-	-	update_rx	!efuse_dis_btlc_gpio1	no
120	-	-	-	rx_status	!efuse_dis_btlc_gpio1	no
121	-	-	-	clk_gpio	!efuse_dis_btlc_gpio1	no
122	-	-	-	nbt_ble	!(efuse_dis_btlc_gpio1&efuse_dis_btlc_gpio0)	no
123	-	-	-	CLK_OUT_out1	1'd1	no
124	-	-	-	CLK_OUT_out2	1'd1	no
125	-	-	-	CLK_OUT_out3	1'd1	no
126	-	-	-	SPICS1_out	1'd1	no
127	-	-	-	usb_jtag_trst	1'd1	no

PRELIMINARY

4.11 IO MUX 管脚功能列表

表 4-2 列出了所有 GPIO 管脚的 IO MUX 功能。

表 4-2. IO MUX 管脚功能

GPIO	管脚名称	功能 0	功能 1	功能 2	功能 3	功能 4	复位	说明
0	XTAL_32K_P	GPIO0	GPIO0	-	-	-	0	R
1	XTAL_32K_N	GPIO1	GPIO1	-	-	-	0	R
2	GPIO2	GPIO2	GPIO2	FSPIQ	-	-	1	R
3	GPIO3	GPIO3	GPIO3	-	-	-	1	R
4	MTMS	MTMS	GPIO4	FSPIHD	-	-	1	R
5	MTDI	MTDI	GPIO5	FSPIWP	-	-	1	R
6	MTCK	MTCK	GPIO6	FSPICLK	-	-	1*	G
7	MTDO	MTDO	GPIO7	FSPID	-	-	1	G
8	GPIO8	GPIO8	GPIO8	-	-	-	1	-
9	GPIO9	GPIO9	GPIO9	-	-	-	3	-
10	GPIO10	GPIO10	GPIO10	FSPICS0	-	-	1	G
11	VDD_SPI	GPIO11	GPIO11	-	-	-	0	-
12	SPIHD	SPIHD	GPIO12	-	-	-	3	-
13	SPIWP	SPIWP	GPIO13	-	-	-	3	-
14	SPICS0	SPICS0	GPIO14	-	-	-	3	-
15	SPICLK	SPICLK	GPIO15	-	-	-	3	-
16	SPID	SPID	GPIO16	-	-	-	3	-
17	SPIQ	SPIQ	GPIO17	-	-	-	3	-
18	GPIO18	GPIO18	GPIO18	-	-	-	0	USB, G
19	GPIO19	GPIO19	GPIO19	-	-	-	0*	USB
20	U0RXD	U0RXD	GPIO20	-	-	-	1	G
21	U0TXD	U0TXD	GPIO21	-	-	-	1	-

复位

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）
- **0*** - IE = 0，WPU = 0，GPIO19 的 USB 上拉默认值为 1，因此，其上拉电阻使能，具体见**说明**。
- **1*** - 如果 eFuse EFUSE_DIS_PAD_JTAG = 1，则 MTCK 管脚复位后浮空，即 IE = 1。如果 eFuse EFUSE_DIS_PAD_JTAG = 0，则 MTCK 管脚连接内部上拉电阻，即 IE = 1，WPU = 1。

说明

- **R** - 代表位于 VDD3P3_RTC 电源域的管脚，部分具有模拟功能，见表 4-4。

PRELIMINARY

- **USB** - GPIO18、GPIO19 为 USB 管脚。USB 管脚的上拉控制由管脚上拉和 USB 上拉共同控制。当其中任意一个为 1 时，对应管脚上拉电阻使能。USB 上拉值对应寄存器 USB_SERIAL_JTAG_DP_PULLUP。
- **G** - 管脚在芯片上电过程中有毛刺，具体见表 4-3。

表 4-3. 芯片上电过程中的管脚毛刺

管脚	毛刺类型	典型持续时间 (ns)
MTCK	低电平毛刺	5
MTDO	低电平毛刺	5
GPIO10	低电平毛刺	5
U0RXD	低电平毛刺	5
GPIO18	上拉	50000

4.12 IO MUX 管脚模拟功能列表

表 4-4 列出了具有模拟功能的 IO MUX 管脚。

表 4-4. IO MUX 管脚的模拟功能

GPIO 编号	管脚名称	模拟功能 0	模拟功能 1
0	XTAL_32K_P	XTAL_32K_P	ADC0
1	XTAL_32K_N	XTAL_32K_N	ADC1
2	GPIO2	-	ADC2
3	GPIO3	-	ADC3

说明

1.VDD_SPI 管脚可配置为电源，也可以配置成普通的 GPIO。

2.GPIO18 和 GPIO19 可配置为 USB 管脚。

4.13 寄存器列表

本小节的所有地址均为相对于 GPIO 交换矩阵、IO MUX 和 SDM 基地址的地址偏移量（相对地址），具体基地址请见章节 3 *系统和存储器* 中的表 3-4。

4.13.1 GPIO 交换矩阵寄存器列表

名称	描述	地址	访问
配置寄存器			
GPIO_BT_SELECT_REG	GPIO 位选择寄存器	0x0000	R/W
GPIO_OUT_REG	GPIO 输出寄存器	0x0004	R/W/SS
GPIO_OUT_W1TS_REG	GPIO 输出置位寄存器	0x0008	WT
GPIO_OUT_W1TC_REG	GPIO 输出清除寄存器	0x000C	WT
GPIO_ENABLE_REG	GPIO 输出使能寄存器	0x0020	R/W/SS
GPIO_ENABLE_W1TS_REG	GPIO 输出使能置位寄存器	0x0024	WT

PRELIMINARY

名称	描述	地址	访问
GPIO_ENABLE_W1TC_REG	GPIO 输出使能清除寄存器	0x0028	WT
GPIO_STRAP_REG	Strapping 管脚寄存器	0x0038	RO
GPIO_IN_REG	GPIO 输入寄存器	0x003C	RO
GPIO_STATUS_REG	GPIO 中断状态寄存器	0x0044	R/W/SS
GPIO_STATUS_W1TS_REG	GPIO 中断状态置位寄存器	0x0048	WT
GPIO_STATUS_W1TC_REG	GPIO 中断状态清除寄存器	0x004C	WT
GPIO_PCPU_INT_REG	GPIO PRO_CPU 中断状态寄存器	0x005C	RO
GPIO_PCPU_NMI_INT_REG	GPIO PRO_CPU 非屏蔽中断状态寄存器	0x0060	RO
GPIO_STATUS_NEXT_REG	GPIO 中断源寄存器	0x014C	RO
管脚配置寄存器			
GPIO_PIN0_REG	配置 GPIO0 管脚	0x0074	R/W
GPIO_PIN1_REG	配置 GPIO1 管脚	0x0078	R/W
GPIO_PIN2_REG	配置 GPIO2 管脚	0x007C	R/W
GPIO_PIN3_REG	配置 GPIO3 管脚	0x0080	R/W
GPIO_PIN4_REG	配置 GPIO4 管脚	0x0084	R/W
GPIO_PIN5_REG	配置 GPIO5 管脚	0x0088	R/W
GPIO_PIN6_REG	配置 GPIO6 管脚	0x008C	R/W
GPIO_PIN7_REG	配置 GPIO7 管脚	0x0090	R/W
GPIO_PIN8_REG	配置 GPIO8 管脚	0x0094	R/W
GPIO_PIN9_REG	配置 GPIO9 管脚	0x0098	R/W
GPIO_PIN10_REG	配置 GPIO10 管脚	0x009C	R/W
GPIO_PIN11_REG	配置 GPIO11 管脚	0x00A0	R/W
GPIO_PIN12_REG	配置 GPIO12 管脚	0x00A4	R/W
GPIO_PIN13_REG	配置 GPIO13 管脚	0x00A8	R/W
GPIO_PIN14_REG	配置 GPIO14 管脚	0x00AC	R/W
GPIO_PIN15_REG	配置 GPIO15 管脚	0x00B0	R/W
GPIO_PIN16_REG	配置 GPIO16 管脚	0x00B4	R/W
GPIO_PIN17_REG	配置 GPIO17 管脚	0x00B8	R/W
GPIO_PIN18_REG	配置 GPIO18 管脚	0x00BC	R/W
GPIO_PIN19_REG	配置 GPIO19 管脚	0x00C0	R/W
GPIO_PIN20_REG	配置 GPIO20 管脚	0x00C4	R/W
GPIO_PIN21_REG	配置 GPIO21 管脚	0x00C8	R/W
输入配置寄存器			
GPIO_FUNC0_IN_SEL_CFG_REG	外设输入信号 0 配置寄存器	0x0154	R/W
GPIO_FUNC1_IN_SEL_CFG_REG	外设输入信号 1 配置寄存器	0x0158	R/W
GPIO_FUNC2_IN_SEL_CFG_REG	外设输入信号 2 配置寄存器	0x015C	R/W
...	...	...	...
GPIO_FUNC125_IN_SEL_CFG_REG	外设输入信号 125 配置寄存器	0x0348	R/W
GPIO_FUNC126_IN_SEL_CFG_REG	外设输入信号 126 配置寄存器	0x034C	R/W
GPIO_FUNC127_IN_SEL_CFG_REG	外设输入信号 127 配置寄存器	0x0350	R/W
输出配置寄存器			
GPIO_FUNC0_OUT_SEL_CFG_REG	GPIO0 管脚的输出配置寄存器	0x0554	R/W
GPIO_FUNC1_OUT_SEL_CFG_REG	GPIO1 管脚的输出配置寄存器	0x0558	R/W

PRELIMINARY

名称	描述	地址	访问
GPIO_FUNC2_OUT_SEL_CFG_REG	GPIO2 管脚的输出配置寄存器	0x055C	R/W
GPIO_FUNC3_OUT_SEL_CFG_REG	GPIO3 管脚的输出配置寄存器	0x0560	R/W
GPIO_FUNC4_OUT_SEL_CFG_REG	GPIO4 管脚的输出配置寄存器	0x0564	R/W
GPIO_FUNC5_OUT_SEL_CFG_REG	GPIO5 管脚的输出配置寄存器	0x0568	R/W
GPIO_FUNC6_OUT_SEL_CFG_REG	GPIO6 管脚的输出配置寄存器	0x056C	R/W
GPIO_FUNC7_OUT_SEL_CFG_REG	GPIO7 管脚的输出配置寄存器	0x0570	R/W
GPIO_FUNC8_OUT_SEL_CFG_REG	GPIO8 管脚的输出配置寄存器	0x0574	R/W
GPIO_FUNC9_OUT_SEL_CFG_REG	GPIO9 管脚的输出配置寄存器	0x0578	R/W
GPIO_FUNC10_OUT_SEL_CFG_REG	GPIO10 管脚的输出配置寄存器	0x057C	R/W
GPIO_FUNC11_OUT_SEL_CFG_REG	GPIO11 管脚的输出配置寄存器	0x0580	R/W
GPIO_FUNC12_OUT_SEL_CFG_REG	GPIO12 管脚的输出配置寄存器	0x0584	R/W
GPIO_FUNC13_OUT_SEL_CFG_REG	GPIO13 管脚的输出配置寄存器	0x0588	R/W
GPIO_FUNC14_OUT_SEL_CFG_REG	GPIO14 管脚的输出配置寄存器	0x058C	R/W
GPIO_FUNC15_OUT_SEL_CFG_REG	GPIO15 管脚的输出配置寄存器	0x0590	R/W
GPIO_FUNC16_OUT_SEL_CFG_REG	GPIO16 管脚的输出配置寄存器	0x0594	R/W
GPIO_FUNC17_OUT_SEL_CFG_REG	GPIO17 管脚的输出配置寄存器	0x0598	R/W
GPIO_FUNC18_OUT_SEL_CFG_REG	GPIO18 管脚的输出配置寄存器	0x059C	R/W
GPIO_FUNC19_OUT_SEL_CFG_REG	GPIO19 管脚的输出配置寄存器	0x05A0	R/W
GPIO_FUNC20_OUT_SEL_CFG_REG	GPIO20 管脚的输出配置寄存器	0x05A4	R/W
GPIO_FUNC21_OUT_SEL_CFG_REG	GPIO21 管脚的输出配置寄存器	0x05A8	R/W
版本寄存器			
GPIO_DATE_REG	版本控制寄存器	0x06FC	R/W
时钟门控寄存器			
GPIO_CLOCK_GATE_REG	GPIO 时钟门控寄存器	0x062C	R/W

4.13.2 IO MUX 寄存器列表

名称	描述	地址	访问
配置寄存器			
IO_MUX_PIN_CTRL_REG	时钟输出配置寄存器	0x0000	R/W
IO_MUX_GPIO0_REG	XTAL_32K_P 的 IO MUX 管脚配置寄存器	0x0004	R/W
IO_MUX_GPIO1_REG	XTAL_32K_N 的 IO MUX 管脚配置寄存器	0x0008	R/W
IO_MUX_GPIO2_REG	GPIO2 的 IO MUX 管脚配置寄存器	0x000C	R/W
IO_MUX_GPIO3_REG	GPIO3 的 IO MUX 管脚配置寄存器	0x0010	R/W
IO_MUX_GPIO4_REG	MTMS 的 IO MUX 管脚配置寄存器	0x0014	R/W
IO_MUX_GPIO5_REG	MTDI 的 IO MUX 管脚配置寄存器	0x0018	R/W
IO_MUX_GPIO6_REG	MTCK 的 IO MUX 管脚配置寄存器	0x001C	R/W
IO_MUX_GPIO7_REG	MTDO 的 IO MUX 管脚配置寄存器	0x0020	R/W
IO_MUX_GPIO8_REG	GPIO8 的 IO MUX 管脚配置寄存器	0x0024	R/W
IO_MUX_GPIO9_REG	GPIO9 的 IO MUX 管脚配置寄存器	0x0028	R/W
IO_MUX_GPIO10_REG	GPIO10 的 IO MUX 管脚配置寄存器	0x002C	R/W
IO_MUX_GPIO11_REG	VDD_SPI 的 IO MUX 管脚配置寄存器	0x0030	R/W
IO_MUX_GPIO12_REG	SPIHD 的 IO MUX 管脚配置寄存器	0x0034	R/W

PRELIMINARY

名称	描述	地址	访问
IO_MUX_GPIO13_REG	SPIWP 的 IO MUX 管脚配置寄存器	0x0038	R/W
IO_MUX_GPIO14_REG	SPICS0 的 IO MUX 管脚配置寄存器	0x003C	R/W
IO_MUX_GPIO15_REG	SPICLK 的 IO MUX 管脚配置寄存器	0x0040	R/W
IO_MUX_GPIO16_REG	SPID 的 IO MUX 管脚配置寄存器	0x0044	R/W
IO_MUX_GPIO17_REG	SPIQ 的 IO MUX 管脚配置寄存器	0x0048	R/W
IO_MUX_GPIO18_REG	GPIO18 的 IO MUX 管脚配置寄存器	0x004C	R/W
IO_MUX_GPIO19_REG	GPIO19 的 IO MUX 管脚配置寄存器	0x0050	R/W
IO_MUX_GPIO20_REG	U0RXD 的 IO MUX 管脚配置寄存器	0x0054	R/W
IO_MUX_GPIO21_REG	U0TXD 的 IO MUX 管脚配置寄存器	0x0058	R/W
版本寄存器			
IO_MUX_DATE_REG	版本控制寄存器	0x00FC	R/W

4.13.3 SDM 寄存器列表

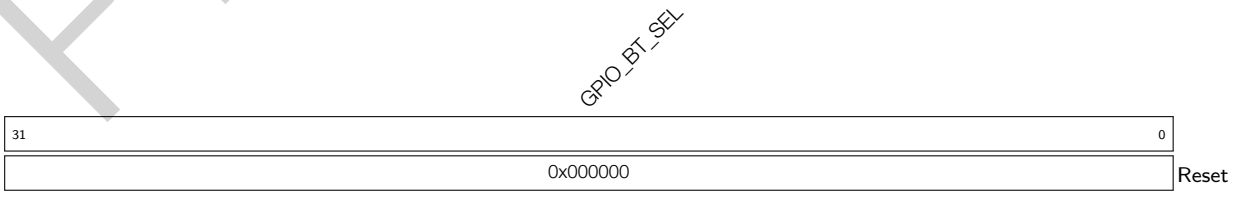
名称	描述	地址	访问
配置寄存器			
GPIOSD_SIGMADELTA0_REG	SDM0 占空比配置寄存器	0x0000	R/W
GPIOSD_SIGMADELTA1_REG	SDM1 占空比配置寄存器	0x0004	R/W
GPIOSD_SIGMADELTA2_REG	SDM2 占空比配置寄存器	0x0008	R/W
GPIOSD_SIGMADELTA3_REG	SDM3 占空比配置寄存器	0x000C	R/W
GPIOSD_SIGMADELTA_CG_REG	时钟门控配置寄存器	0x0020	R/W
GPIOSD_SIGMADELTA_MISC_REG	MISC 寄存器	0x0024	R/W
版本寄存器			
GPIOSD_SIGMADELTA_VERSION_REG	版本控制寄存器	0x0028	R/W

4.14 寄存器

本小节的所有地址均为相对于 GPIO 交换矩阵、IO MUX 和 SDM 基地址的地址偏移量（相对地址），具体基地址请见章节 3 *系统和存储器* 中的表 3-4。

4.14.1 GPIO 交换矩阵寄存器

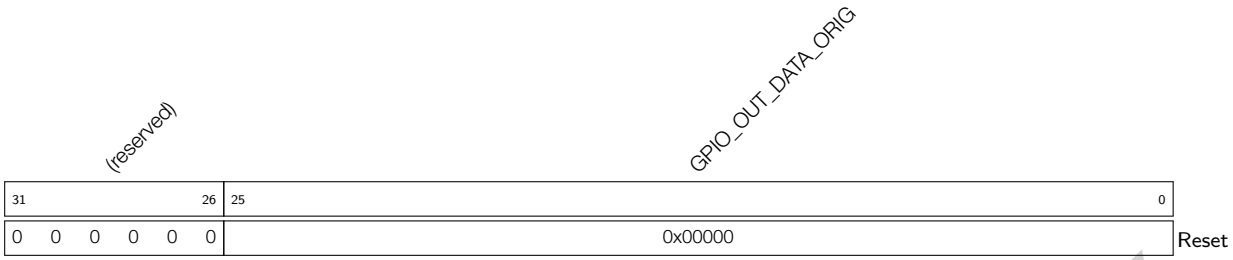
Register 4.1. GPIO_BT_SELECT_REG (0x0000)



GPIO_BT_SEL 保留 (R/W)

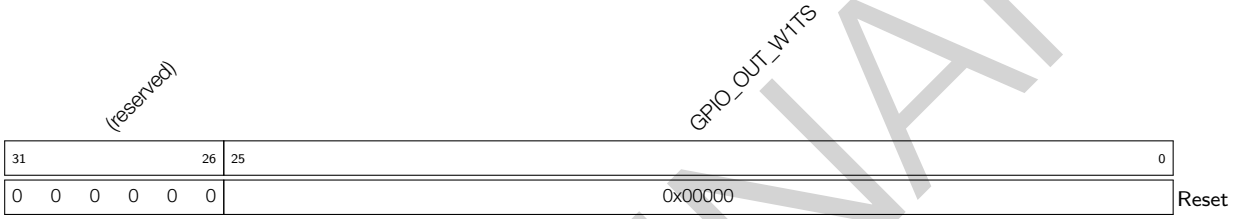
PRELIMINARY

Register 4.2. GPIO_OUT_REG (0x0004)



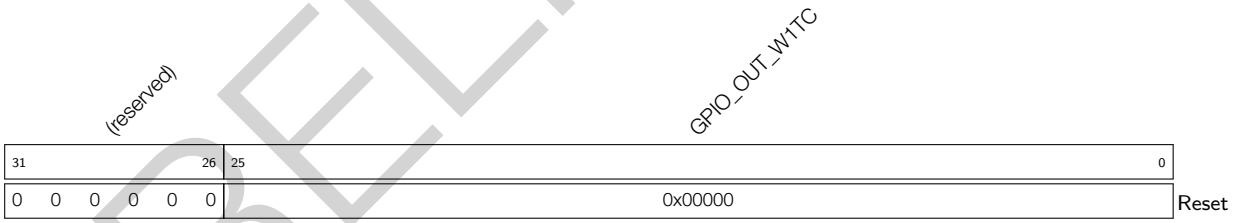
GPIO_OUT_DATA_ORIG 简单 GPIO 输出模式下 GPIO0 ~ 21 的输出值。bit0 ~ bit21 的值分别对应 GPIO0 ~ GPIO21 的值。bit22 ~ bit25 无效。(R/W/SS)

Register 4.3. GPIO_OUT_W1TS_REG (0x0008)



GPIO_OUT_W1TS GPIO0 ~ 21 输出置位寄存器，bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位置 1，则 GPIO_OUT_REG 中相应位也将置 1。注：推荐使用此寄存器来置位 GPIO_OUT_REG。(WT)

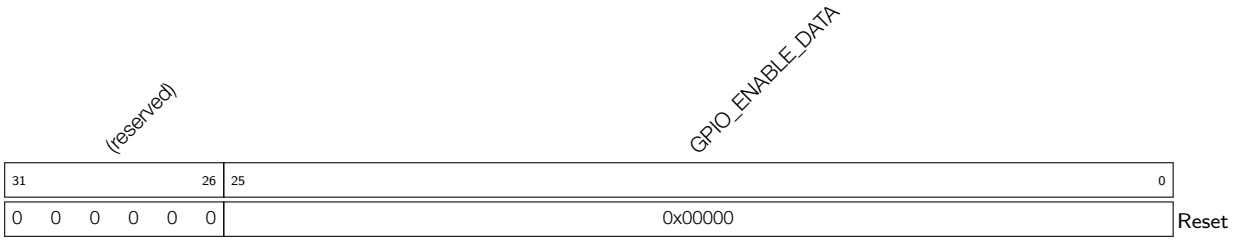
Register 4.4. GPIO_OUT_W1TC_REG (0x000C)



GPIO_OUT_W1TC GPIO0 ~ 21 输出清零寄存器，bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位置 1，则 GPIO_OUT_REG 中相应位会清零。注：推荐使用此寄存器来清零 GPIO_OUT_REG。(WT)

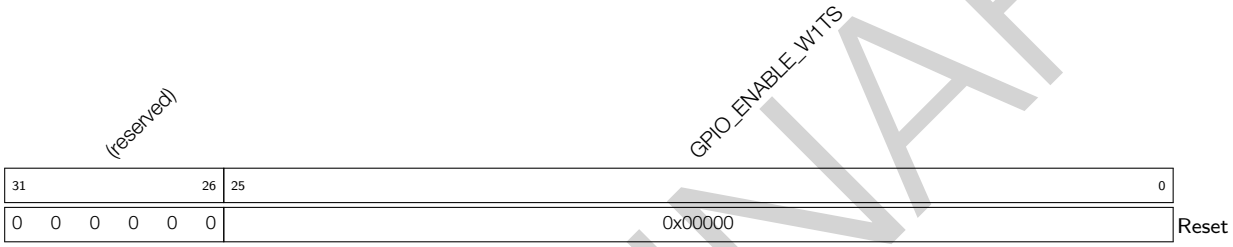
PRELIMINARY

Register 4.5. GPIO_ENABLE_REG (0x0020)



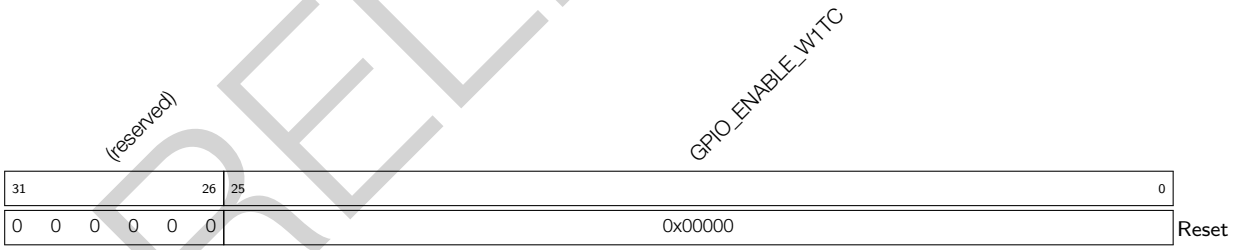
GPIO_ENABLE_DATA GPIO0 ~ 21 输出使能寄存器，bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。(R/W/SS)

Register 4.6. GPIO_ENABLE_W1TS_REG (0x0024)



GPIO_ENABLE_W1TS GPIO0 ~ 21 输出使能置位寄存器。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位置 1，则 GPIO_ENABLE_REG 中相应位也将置 1。注：推荐使用此寄存器来置位 GPIO_ENABLE_REG。(WT)

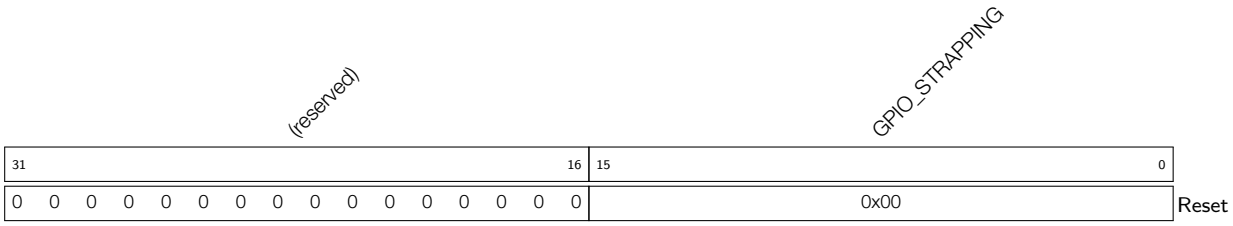
Register 4.7. GPIO_ENABLE_W1TC_REG (0x0028)



GPIO_ENABLE_W1TC GPIO0 ~ 21 输出使能清零寄存器。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位置 1，则 GPIO_ENABLE_REG 中相应位会清零。注：推荐使用此寄存器清零 GPIO_ENABLE_REG。(WT)

PRELIMINARY

Register 4.8. GPIO_STRAP_REG (0x0038)



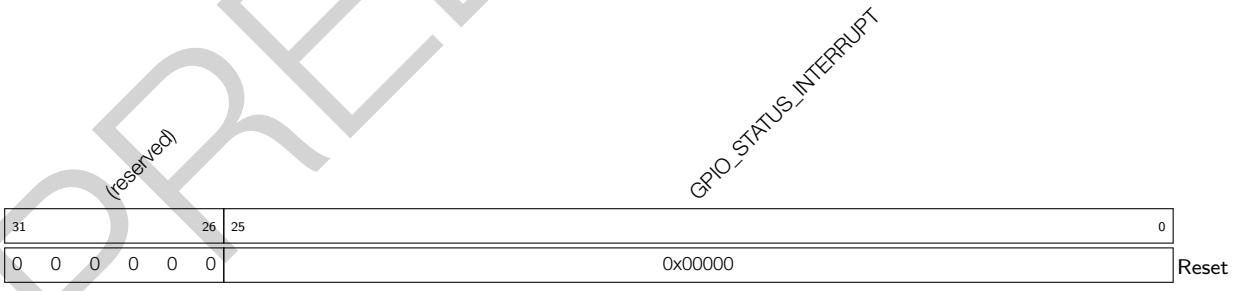
GPIO_STRAPPING GPIO Strapping 值。(RO)

- bit 0：对应 GPIO2
- bit 2：对应 GPIO8
- bit 3：对应 GPIO9

Register 4.9. GPIO_IN_REG (0x003C)

GPIO_IN_DATA_NEXT GPIO0 ~ 21 输入值。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位代表一个管脚的片外输入值，0 表示低电平，1 表示高电平。(RO)

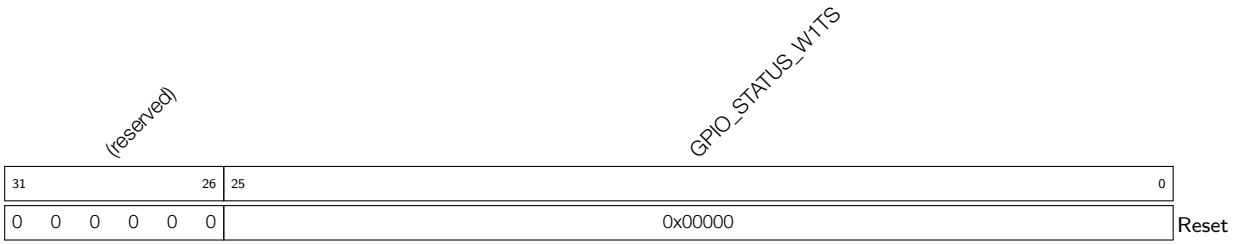
Register 4.10. GPIO_STATUS_REG (0x0044)



GPIO_STATUS_INTERRUPT GPIO0 ~ 21 中断状态寄存器。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。(R/W/SS)

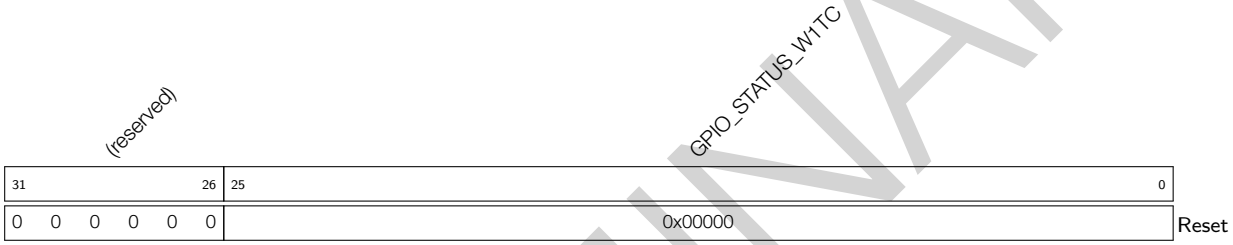
PRELIMINARY

Register 4.11. GPIO_STATUS_W1TS_REG (0x0048)



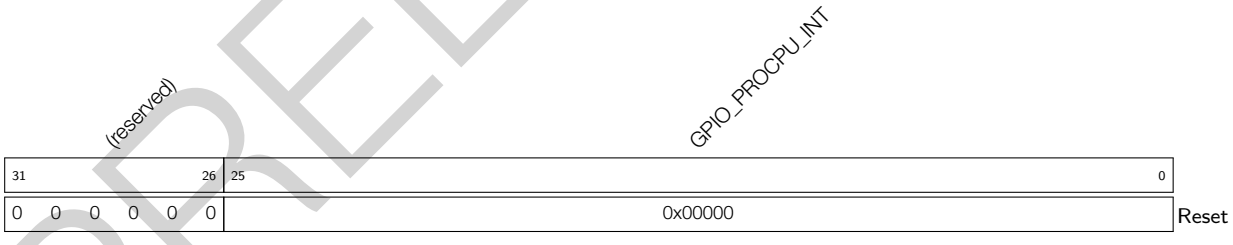
GPIO_STATUS_W1TS GPIO0 ~ 21 中断状态置位寄存器。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位置 1，则 GPIO_STATUS_INTERRUPT 中相应位也将置 1。注：推荐使用此寄存器来置位 GPIO_STATUS_INTERRUPT。(WT)

Register 4.12. GPIO_STATUS_W1TC_REG (0x004C)



GPIO_STATUS_W1TC GPIO0 ~ 21 中断状态清除寄存器。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位置 1，则 GPIO_STATUS_INTERRUPT 中相应位会清零。注：推荐使用此寄存器来清零 GPIO_STATUS_INTERRUPT。(WT)

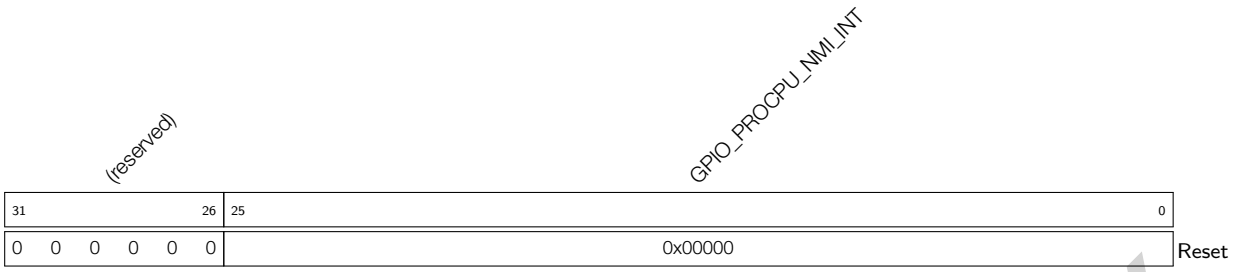
Register 4.13. GPIO_PCPU_INT_REG (0x005C)



GPIO_PROCPU_INT GPIO0 ~ 21 PRO_CPU 中断状态。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。如果 GPIO_PIN*n*_REG 中 bit13 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 中相应位的中断状态一致。(RO)

PRELIMINARY

Register 4.14. GPIO_PCPU_NMI_INT_REG (0x0060)



GPIO_PROCPU_NMI_INT GPIO0 ~ 21 PRO_CPU 非屏蔽中断状态寄存器。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。如果 GPIO_PIN*n*_REG 中 bit14 有效，即使能 CPU 非屏蔽中断，则该寄存器所示的中断状态应与 GPIO_STATUS_REG 中相应位的中断状态一致。(RO)

Register 4.15. GPIO_PIN*n*_REG (*n*: 0-21) (0x0074+4n*)**

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

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

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

GPIO_PIN*n*_INT_TYPE 中断类型选择。(R/W)

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

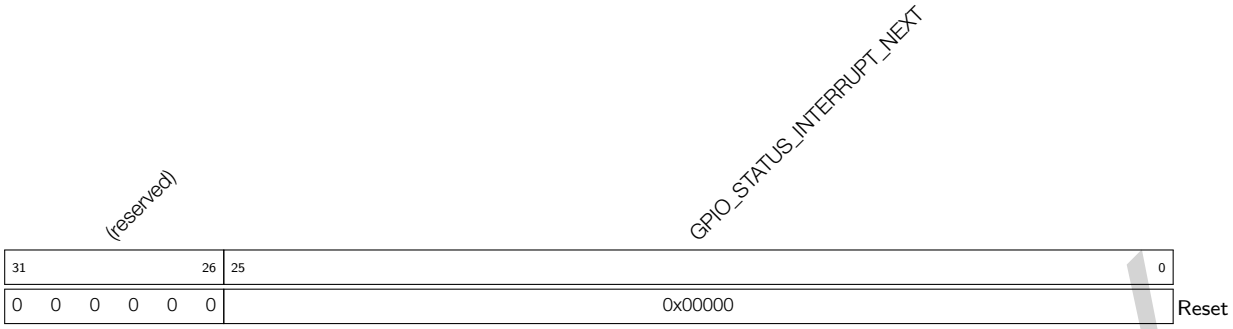
GPIO_PIN*n*_WAKEUP_ENABLE 使能 GPIO 唤醒，仅能将 CPU 从 Light-sleep 模式唤醒。(R/W)

GPIO_PIN*n*_CONFIG 保留。(R/W)

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

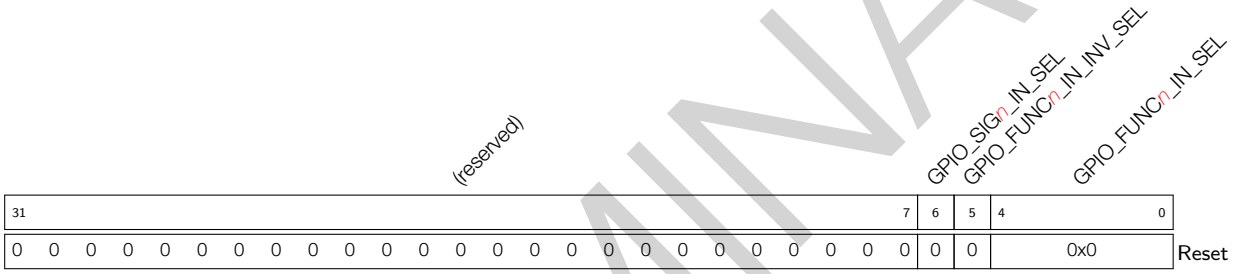
PRELIMINARY

Register 4.16. GPIO_STATUS_NEXT_REG (0x014C)



GPIO_STATUS_INTERRUPT_NEXT GPIO0 ~ 21 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。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。(RO)

Register 4.17. GPIO_FUNC*n*_IN_SEL_CFG_REG (*n*: 0-127) (0x0154+4n*)**



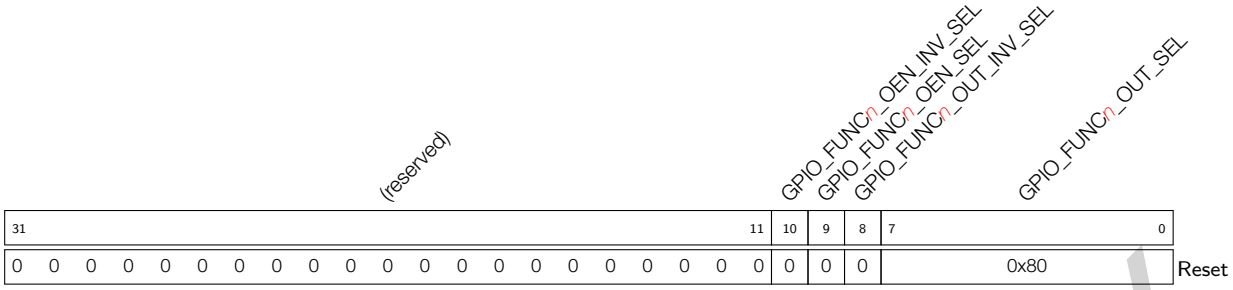
GPIO_FUNC*n*_IN_SEL 外设输入信号 *n* 的选择控制位。此位选择 1 个 GPIO 交换矩阵输入管脚与信号连接，或者选择 0x1E 与恒高电平输入信号连接，或者选择 0x1F 与恒低电平输入信号连接。(R/W)

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

GPIO_SIG*n*_IN_SEL 旁路 GPIO 交换矩阵。1：通过 GPIO 交换矩阵；0：直接通过 IO MUX 连接信号与外设。(R/W)

PRELIMINARY

Register 4.18. GPIO_FUNC*n*_OUT_SEL_CFG_REG (*n*: 0-21) (0x0554+4*n)



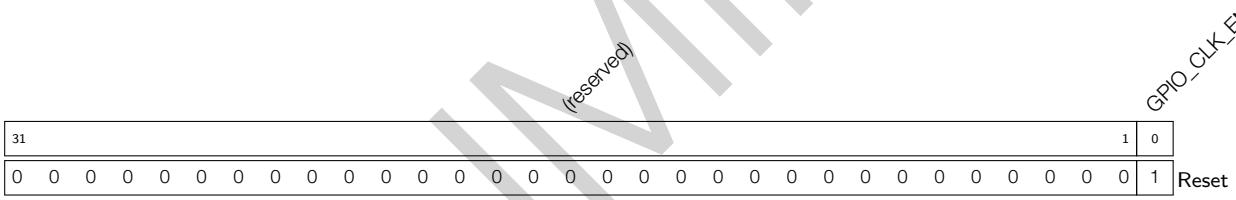
GPIO_FUNC*n*_OUT_SEL GPIO 管脚输出 *n* 的选择控制位。如果该字段设置为 *Y* (0<=Y<128)，则外设输出信号 *Y* 将连接至 GPIO *n* 输出。如果该字段设置为 128，则寄存器 GPIO_OUT_REG 和 GPIO_ENABLE_REG 中的 bit*n* 将用作输出值和输出使能。(R/W)

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

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

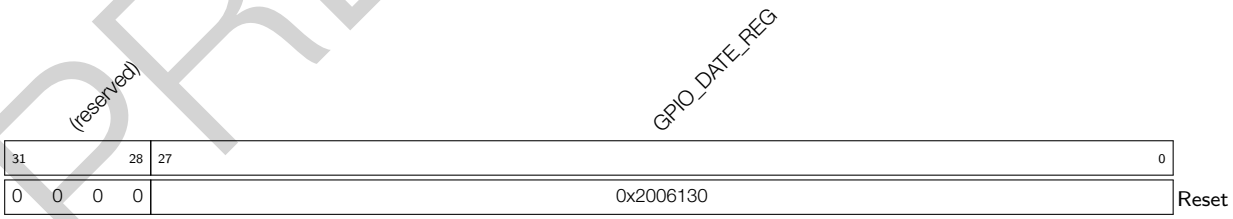
GPIO_FUNC*n*_OEN_INV_SEL 0：不反转输出使能信号；1：反转输出使能信号。(R/W)

Register 4.19. GPIO_CLOCK_GATE_REG (0x062C)



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

Register 4.20. GPIO_DATE_REG (0x06FC)

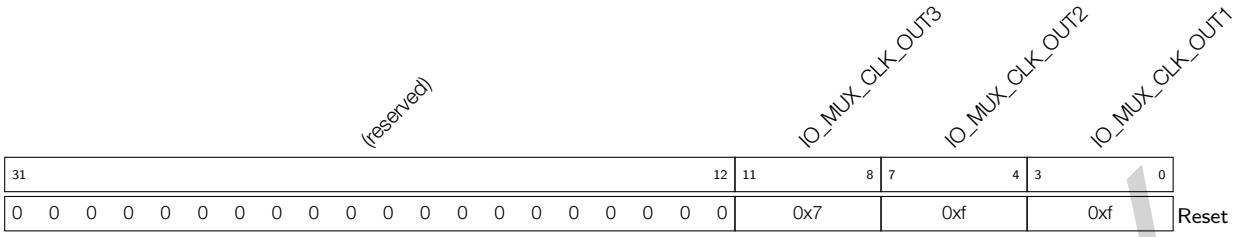


GPIO_DATE_REG 版本控制寄存器。(R/W)

PRELIMINARY

4.14.2 IO MUX 寄存器

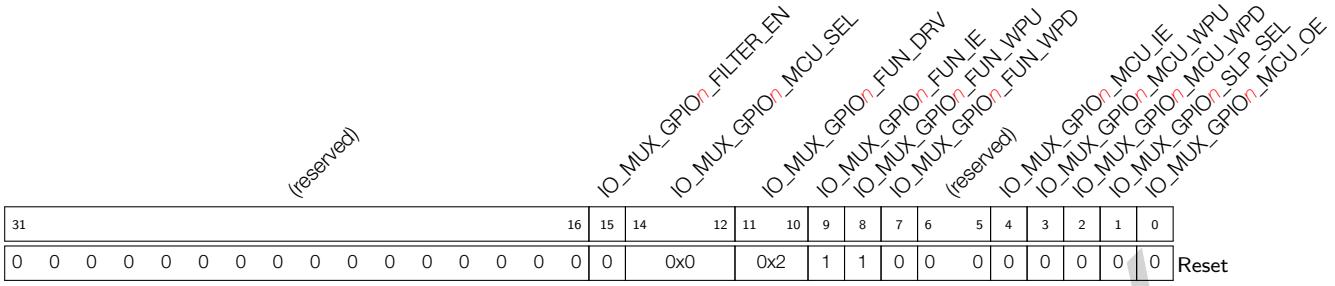
Register 4.21. IO_MUX_PIN_CTRL_REG (0x0000)



IO_MUX_CLK_OUT*x* 配置 I2S 外设时钟输出到 CLK_OUT_out*x*，需要设置 IO_MUX_CLK_OUT*x* 为 0x0。有关 CLK_OUT_out*x* 的信息，见表 4-1。(R/W)

PRELIMINARY

Register 4.22. IO_MUX_GPIO*n*_REG (*n*: 0-21) (0x0004+4n*)**



IO_MUX_GPIO*n*_MCU_OE 睡眠模式下，管脚的输出使能位。1：输出使能；0：输出关闭。(R/W)

IO_MUX_GPIO*n*_SLP_SEL 管脚睡眠模式选择。置 1 进入睡眠模式。(R/W)

IO_MUX_GPIO*n*_MCU_WPD 睡眠模式下，管脚的下拉电阻使能位。1：使能内部下拉电阻；0：关闭内部下拉电阻。(R/W)

IO_MUX_GPIO*n*_MCU_WPU 睡眠模式下，管脚的上拉电阻使能位。1：使能内部上拉电阻；0：关闭内部上拉电阻。(R/W)

IO_MUX_GPIO*n*_MCU_IE 睡眠模式下，管脚的输入使能位。1：使能输入；0：关闭输入。(R/W)

IO_MUX_GPIO*n*_FUN_WPD 管脚的下拉电阻使能位。1：使能内部下拉电阻；0：关闭内部下拉电阻。(R/W)

IO_MUX_GPIO*n*_FUN_WPU 管脚的上拉电阻使能位。1：使能内部上拉电阻；0：关闭内部上拉电阻。(R/W)

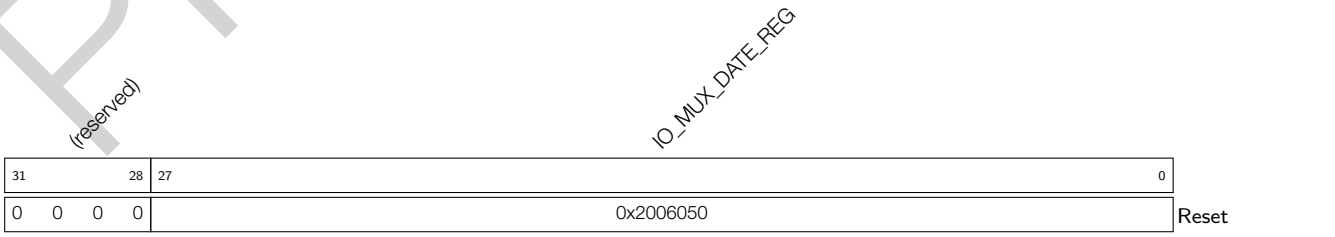
IO_MUX_GPIO*n*_FUN_IE 管脚的输入使能位。1：使能输入；0：关闭输入。(R/W)

IO_MUX_GPIO*n*_FUN_DRV 选择管脚驱动强度。0：~5 mA；1：~10mA；2：~20mA；3：~40mA。(R/W)

IO_MUX_GPIO*n*_MCU_SEL 选择管脚功能。0：选择 Function 0；1：选择 Function 1；以此类推。(R/W)

IO_MUX_GPIO*n*_FILTER_EN 使能管脚输入信号滤波。1：滤波使能；0：滤波关闭。(R/W)

Register 4.23. IO_MUX_DATE_REG (0x00FC)

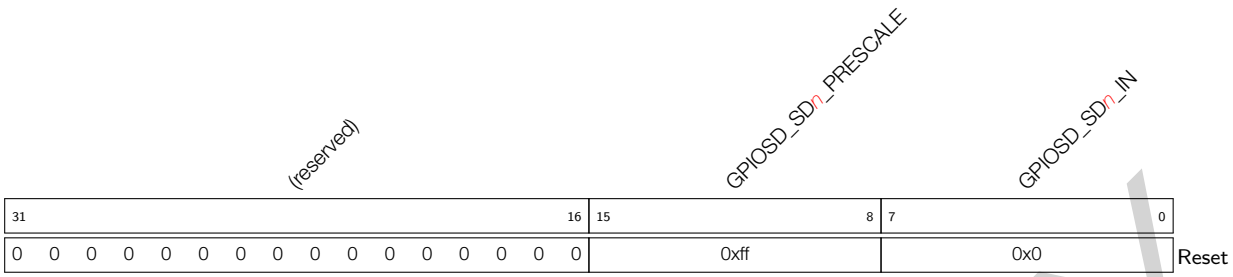


IO_MUX_DATE_REG 版本控制寄存器。(R/W)

PRELIMINARY

4.14.3 SDM 寄存器

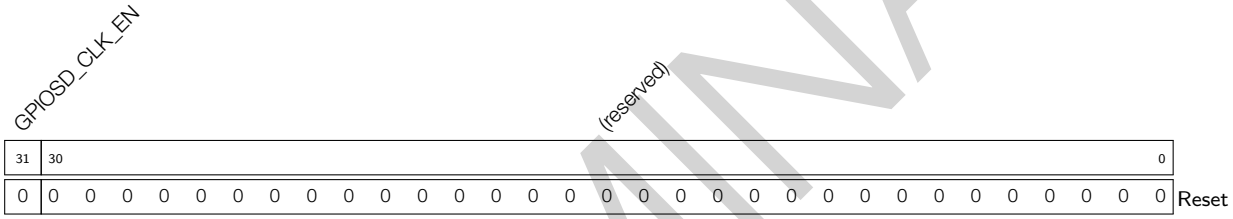
Register 4.24. GPIOSD_SIGMADELTA*n*_REG (*n*: 0-3) (0x0000+4n*)**



GPIOSD_SD*n*_IN 配置 SDM 输出信号的占空比。(R/W)

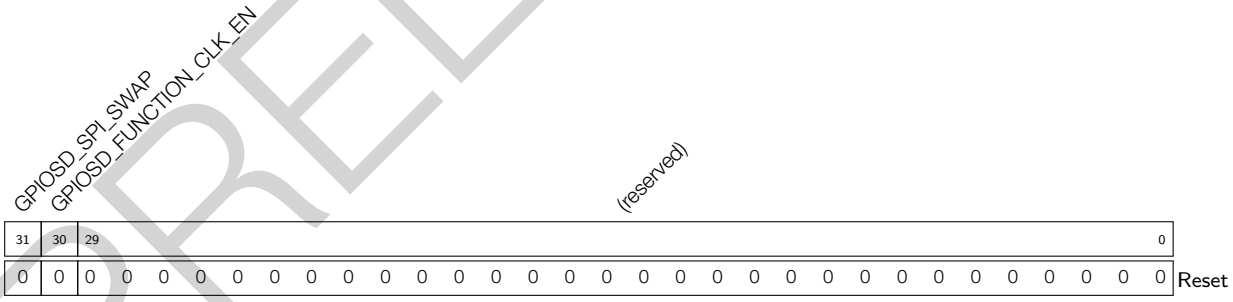
GPIOSD_SD*n*_PRESCALE 配置 APB_CLK 分频系数。(R/W)

Register 4.25. GPIOSD_SIGMADELTA_CG_REG (0x0020)



GPIOSD_CLK_EN 使能 SDM 配置寄存器的时钟。(R/W)

Register 4.26. GPIOSD_SIGMADELTA_MISC_REG (0x0024)



GPIOSD_FUNCTION_CLK_EN 使能 SDM 的时钟。(R/W)

GPIOSD_SPI_SWAP 保留。(R/W)

PRELIMINARY

Register 4.27. GPIOSD_SIGMADELTA_VERSION_REG (0x0028)

GPIOSD_DATE 版本控制寄存器。(R/W)

PRELIMINARY

5 SHA 加速器

5.1 概述

ESP32-C3 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。

5.2 主要特性

ESP32-C3 的 SHA 硬件加速器：

- 支持 FIPS PUB 180-4 规范中的以下运算标准
  - SHA-1 运算
  - SHA-224 运算
  - SHA-256 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）

5.3 工作模式简介

ESP32-C3 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 5-1。

表 5-1. 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

PRELIMINARY

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 5-2。

表 5-2. 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

5.4 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。

5.4.1 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。

5.4.1.1 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，则填充步骤见下：

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。

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

5.4.1.2 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 $M_{15}^{(i)}$。

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：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。

> **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。

5.4.1.3 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 $H^{(0)}$，其中 SHA-1、SHA-224 和 SHA-256 运算的哈希初始值为常量 C，且已经固定在硬件中，无需额外配置。

PRELIMINARY

5.4.2 哈希运算流程

在完成信息预处理后，ESP32-C3 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-C3 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。

5.4.2.1 Typical SHA 模式下的运算流程

通常情况下，ESP32-C3 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-C3 SHA 加速器还支持“interleaved”运算（Typical SHA 和 DMA-SHA 工作模式均支持），即在完成当前信息的所有运算前，允许插入其他运算任务。

- 在 Typical SHA 工作模式下，用户每计算完一个信息块后均可插入新的运算；
- 而在 DMA-SHA 工作模式下，用户必须等待本次 DMA 运算全部完成才可以插入新的运算。

具体来说，用户可以将存储在 SHA_H_*n*_REG 寄存器中的信息摘要暂时保存到其他地方，然后让 SHA 加速器来完成其他优先级更高的运算任务。当插入的运算结束后，用户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

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 5-2。
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。

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

PRELIMINARY

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-C3 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。

5.4.2.2 DMA-SHA 模式下的运算流程

ESP32-C3 SHA 加速器在 DMA-SHA 工作模式下不支持在完成每个 "信息块" 运算后插入新的运算，即用户必须在每次 DMA 运算（可能包括 1 个或多个信息块）全部结束后才能插入新的运算。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 运算。每次 DMA 运算之间允许插入其他运算标准的计算任务。

单次 DMA 运算最多可以处理 63 个数据块。

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。**DMA-SHA 的具体工作流程**

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 5-2。
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。

PRELIMINARY

6. 获取信息摘要

   - 从寄存器堆 SHA_H_*n*_REG 取出信息摘要。

5.4.3 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 ~ 7) 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 5-3：

表 5-3. 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

[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。

5.4.4 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。

PRELIMINARY

5.5 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 3 系统和存储器 中的表 3-4。

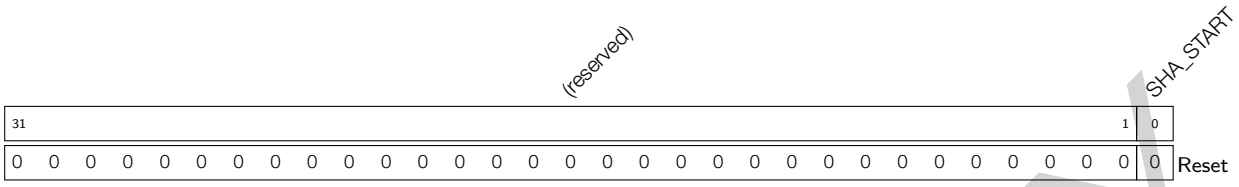
名称	描述	地址	权限
控制与状态寄存器			
SHA_CONTINUE_REG	继续 SHA 运算（仅用于 Typical SHA 模式）	0x0014	WO
SHA_BUSY_REG	指示 SHA 加速器是否处于“忙碌”状态	0x0018	RO
SHA_DMA_START_REG	启动 SHA 加速器的 DMA-SHA 模式	0x001C	WO
SHA_START_REG	启动 SHA 加速器的 Typical SHA 模式	0x0010	WO
SHA_DMA_CONTINUE_REG	继续 SHA 运算（仅用于 DMA-SHA 模式）	0x0020	WO
SHA_INT_CLEAR_REG	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			
SHA_MODE_REG	配置 SHA 加速器的运算标准	0x0000	R/W
数据寄存器			
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_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

PRELIMINARY

5.6 寄存器

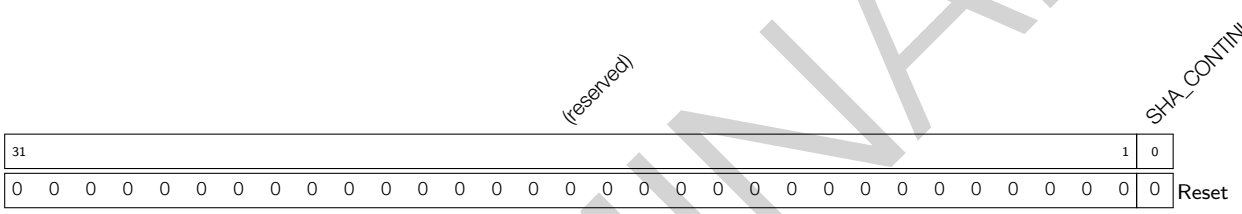
本小节的所有地址均为相对于 SHA 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

Register 5.1. SHA_START_REG (0x0010)



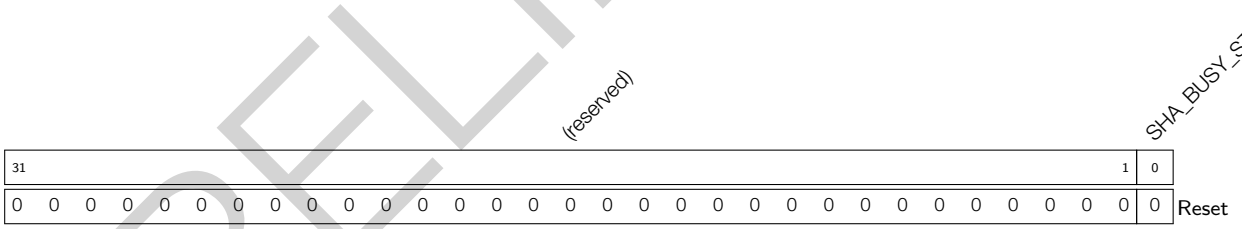
SHA_START 置 1 启动 SHA 加速器的 Typical SHA 模式。（只写）

Register 5.2. SHA_CONTINUE_REG (0x0014)



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

Register 5.3. SHA_BUSY_REG (0x0018)



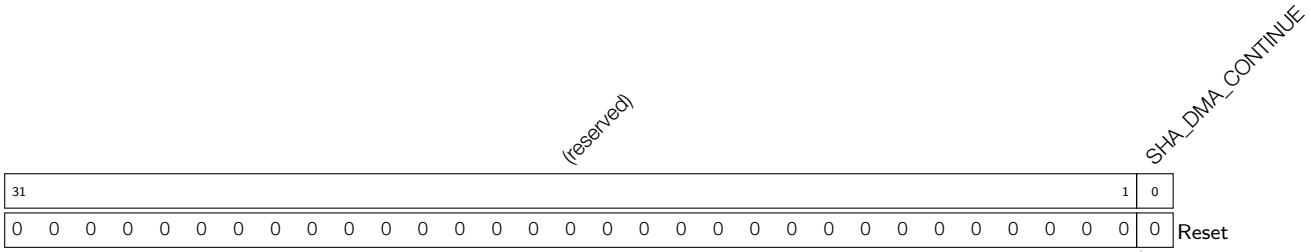
SHA_BUSY_STATE 指示 SHA 是否处于“忙碌”状态。（只读）1'h0: 空闲 1'h1: 忙碌

Register 5.4. SHA_DMA_START_REG (0x001C)

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

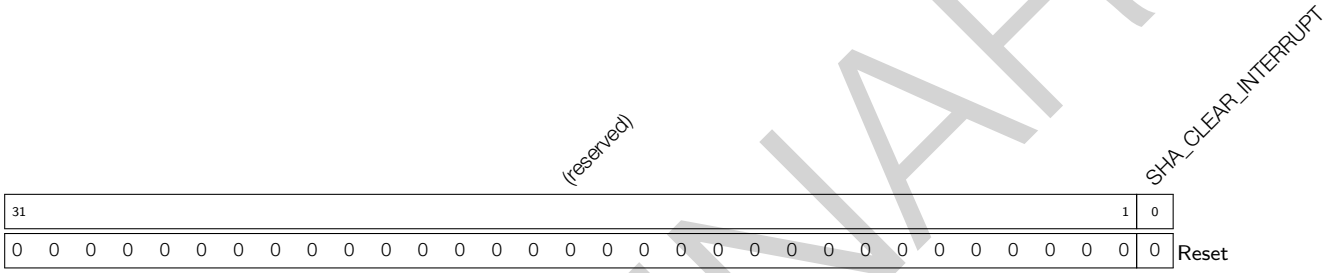
PRELIMINARY

Register 5.5. SHA_DMA_CONTINUE_REG (0x0020)



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

Register 5.6. SHA_INT_CLEAR_REG (0x0024)

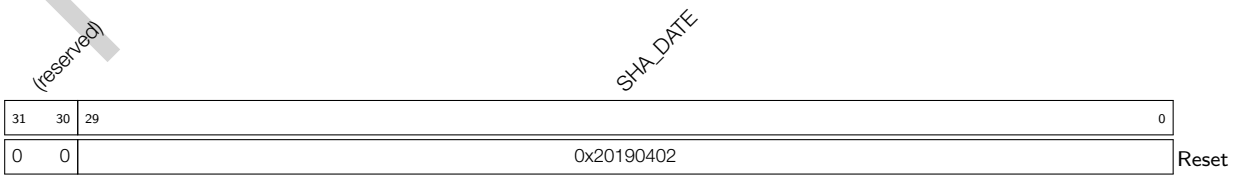


SHA_CLEAR_INTERRUPT 清除 DMA-SHA 中断。（只写）

Register 5.7. SHA_INT_ENA_REG (0x0028)

SHA_INTERRUPT_ENA 使能 DMA-SHA 中断。（读写）

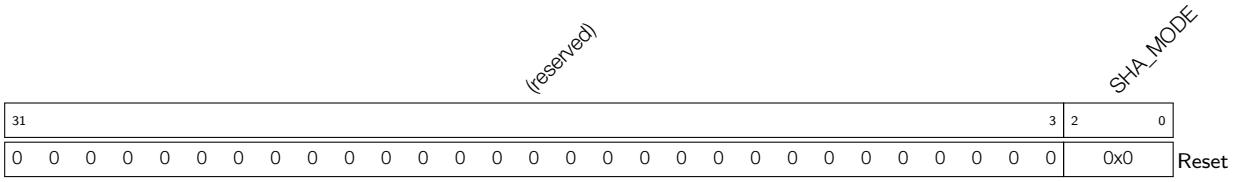
Register 5.8. SHA_DATE_REG (0x002C)



SHA_DATE 版本控制寄存器。（读写）

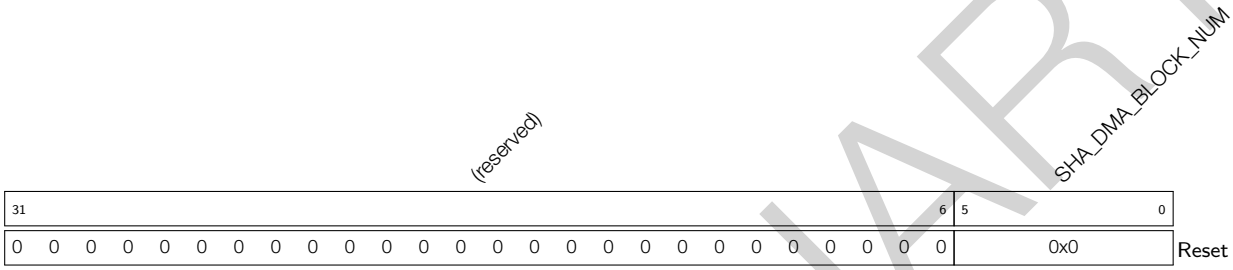
PRELIMINARY

Register 5.9. SHA_MODE_REG (0x0000)



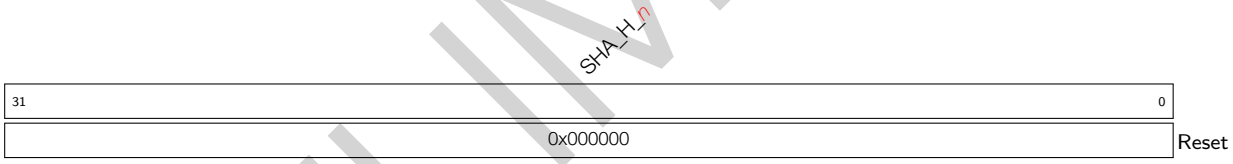
SHA_MODE 选择 SHA 加速器的运算标准，详见表 5-2。(R/W)

Register 5.10. SHA_DMA_BLOCK_NUM_REG (0x000C)



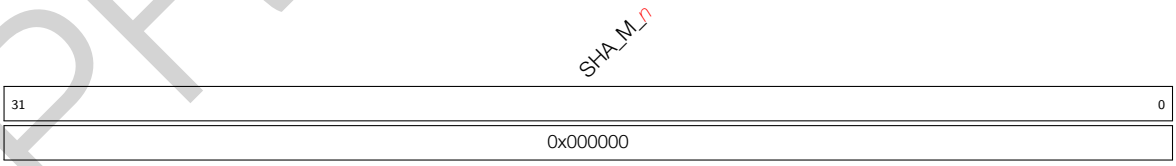
SHA_DMA_BLOCK_NUM 定义 DMA-SHA 工作模式下的信息块个数。（读写）

Register 5.11. SHA_H_*n*_REG (*n*: 0-7) (0x0040+4n*)**



SHA_H_*n* 存储第 *n* 个 32 位哈希值。（读写）

Register 5.12. SHA_M_*n*_REG (*n*: 0-15) (0x0080+4n*)**



SHA_M_*n* 存储第 *n* 个 32 位输入信息。（读写）

PRELIMINARY

6 AES 加速器

6.1 概述

ESP32-C3 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。

6.2 主要特性

ESP32-C3 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

6.3 工作模式简介

ESP32-C3 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 6-1。

PRELIMINARY

表 6-1. 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 6-2。

表 6-2. 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 6.4 章节和 6.5 章节。

6.4 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 6-3 所示：

表 6-3. 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

6.4.1 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。

PRELIMINARY

6.4.2 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 6-4 中定义的文本字节序。

表 6-4. 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 6-5 和表 6-6 中定义的文本字节序。

表 6-5. 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。

PRELIMINARY

表 6-6. 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。

PRELIMINARY

6.4.3 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。

PRELIMINARY

6.5 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 6-7。

表 6-7. 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 6-8 所示：

表 6-8. 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。

6.5.1 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 6-9 所示。

PRELIMINARY

表 6-9. 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**($X$) = $X_1 \| \| X_2 \| \| \cdots \| \| X_{n-1} \| \| X_n^*$ = $X \| \| 0^{128-t}$

6.5.2 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 6-10 所示。结果数据也遵从相同的存放规则，在此不多做介绍。

表 6-10. 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

6.5.3 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。

6.5.4 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

PRELIMINARY

密文（ciphertext）。该寄存器仅在 DMA-AES 工作模式下有意义。

6.5.5 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 6-10 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。

6.5.6 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。
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 6-7。
   - 初始化寄存器 AES_BLOCK_NUM_REG，请参照章节 6.5.4。
   - 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。
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。

PRELIMINARY

6.6 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 3 系统和存储器 中的表 3-4。

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

6.7 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 3 系统和存储器 中的表 3-4。

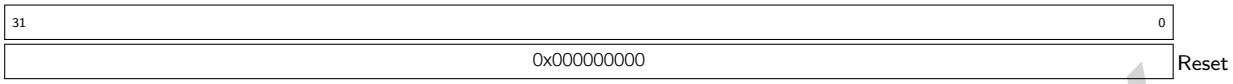
名称	描述	地址	访问
密钥寄存器			
AES_KEY_0_REG	AES 密钥资料寄存器 0	0x0000	读 / 写
AES_KEY_1_REG	AES 密钥资料寄存器 1	0x0004	读 / 写
AES_KEY_2_REG	AES 密钥资料寄存器 2	0x0008	读 / 写
AES_KEY_3_REG	AES 密钥资料寄存器 3	0x000C	读 / 写
AES_KEY_4_REG	AES 密钥资料寄存器 4	0x0010	读 / 写
AES_KEY_5_REG	AES 密钥资料寄存器 5	0x0014	读 / 写
AES_KEY_6_REG	AES 密钥资料寄存器 6	0x0018	读 / 写
AES_KEY_7_REG	AES 密钥资料寄存器 7	0x001C	读 / 写
TEXT_IN 寄存器			
AES_TEXT_IN_0_REG	源数据资料寄存器 0	0x0020	读 / 写
AES_TEXT_IN_1_REG	源数据资料寄存器 1	0x0024	读 / 写
AES_TEXT_IN_2_REG	源数据资料寄存器 2	0x0028	读 / 写
AES_TEXT_IN_3_REG	源数据资料寄存器 3	0x002C	读 / 写
TEXT_OUT 寄存器			
AES_TEXT_OUT_0_REG	结果数据资料寄存器 0	0x0030	只读
AES_TEXT_OUT_1_REG	结果数据资料寄存器 1	0x0034	只读
AES_TEXT_OUT_2_REG	结果数据资料寄存器 2	0x0038	只读
AES_TEXT_OUT_3_REG	结果数据资料寄存器 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

PRELIMINARY

6.8 寄存器

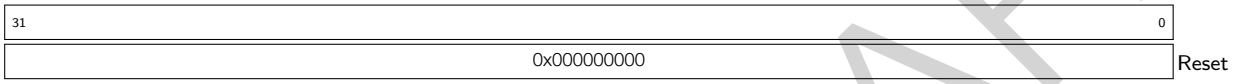
本小节的所有地址均为相对于 AES 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

Register 6.1. AES_KEY_*n*_REG (*n*: 0-7) (0x0000+4n*)**



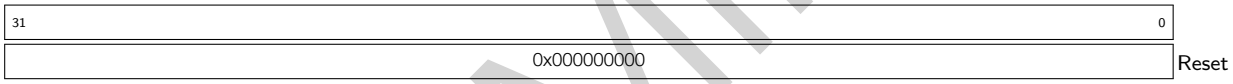
AES_KEY_*n*_REG (*n*: 0-7) AES 密钥资料寄存器。（读 / 写）

Register 6.2. AES_TEXT_IN_*m*_REG (*m*: 0-3) (0x0020+4m*)**



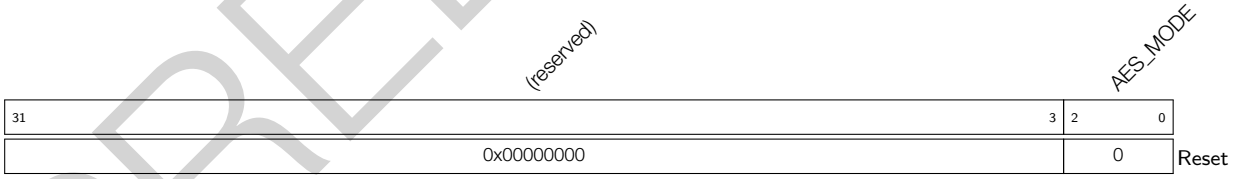
AES_TEXT_IN_*m*_REG (*m*: 0-3) Typical AES 文本输入寄存器。（读 / 写）

Register 6.3. 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）

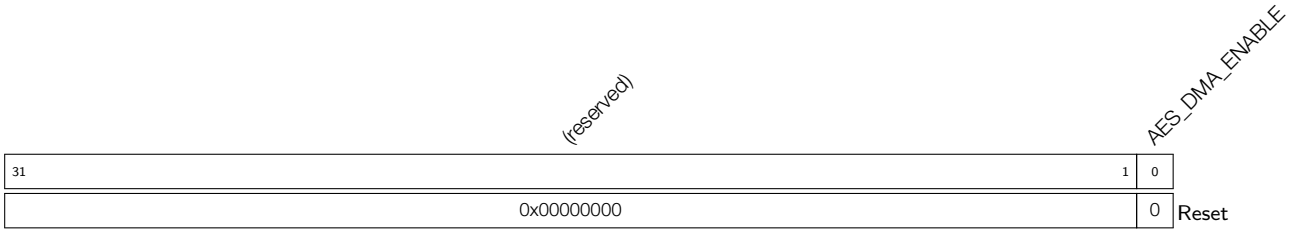
Register 6.4. AES_MODE_REG (0x0040)



AES_MODE 选择 AES 加速器的密钥长度和加解密方向，详情请见表 6-2。（读 / 写）

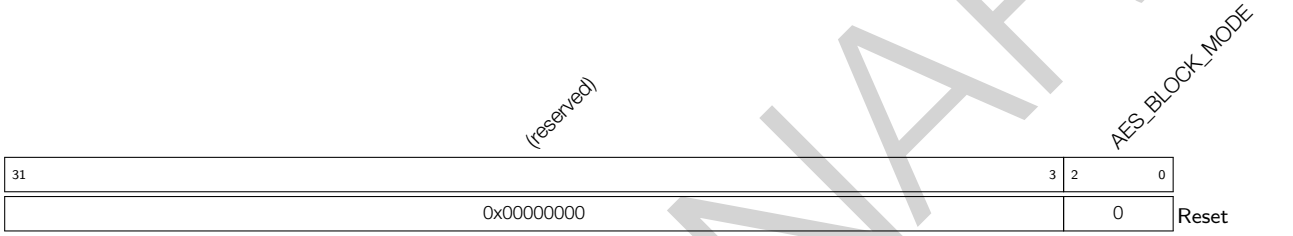
PRELIMINARY

Register 6.5. AES_DMA_ENABLE_REG (0x0090)



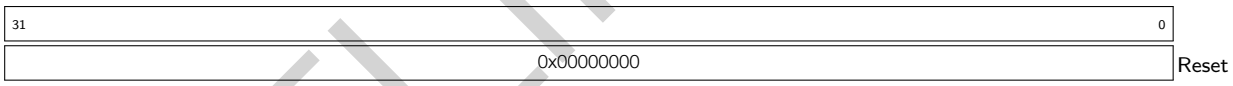
AES_DMA_ENABLE 选择 AES 加速器的工作模式。0：Typical AES，1：DMA-AES。详情请见表 6-1。(读 / 写)

Register 6.6. AES_BLOCK_MODE_REG (0x0094)



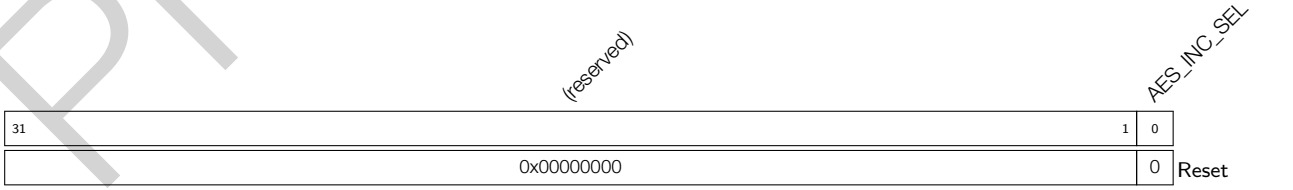
AES_BLOCK_MODE 选择 AES 加速器在 DMA-AES 工作模式下的块模式，详情请见表 6-7。(读 / 写)

Register 6.7. 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 6.5.4。(读 / 写)

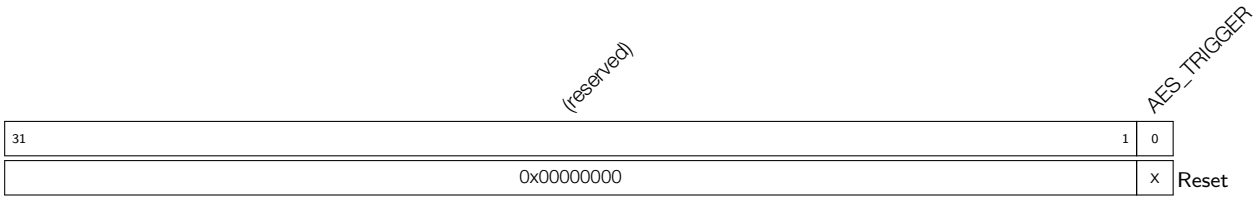
Register 6.8. AES_INC_SEL_REG (0x009C)



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)

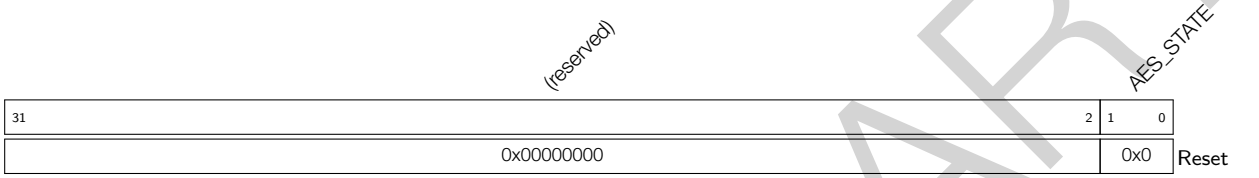
PRELIMINARY

Register 6.9. AES_TRIGGER_REG (0x0048)



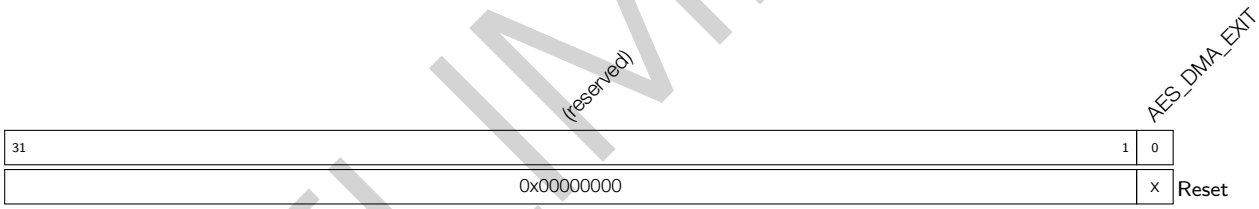
AES_TRIGGER 写入 1 使能 AES 运算。(只写)

Register 6.10. AES_STATE_REG (0x004C)



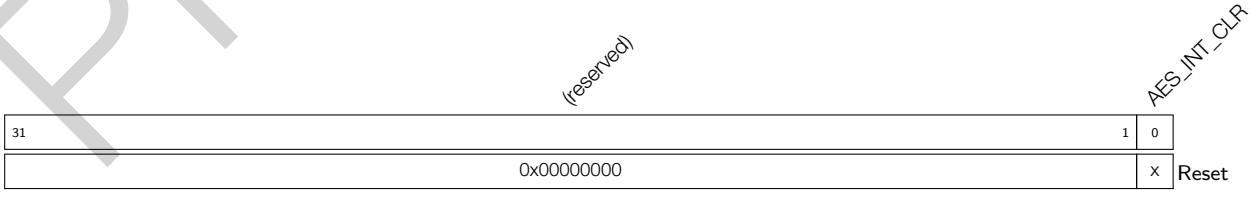
AES_STATE AES 状态寄存器。详见表 6-3 (Typical AES 工作模式) 和表 6-8 (DMA-AES 工作模式)。(只读)

Register 6.11. 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)

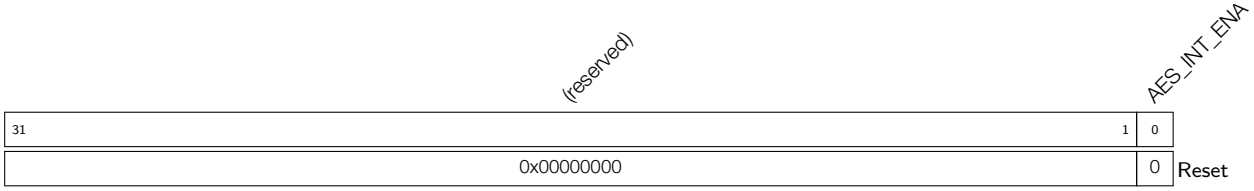
Register 6.12. AES_INT_CLR_REG (0x00AC)



AES_INT_CLR 写入 1 清除 AES 中断。(只写)

PRELIMINARY

Register 6.13. 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）

PRELIMINARY

7 RSA 加速器

7.1 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。

7.2 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

7.3 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。

7.3.1 大数模幂运算

大数模幂运算的算法是 $Z = X^Y \bmod M$，它是基于 Montgomery Multiplication（蒙哥马利乘法）实现的。因此，对于大数模幂运算，除了需要运算子 X、Y、M 外，还需要额外两个运算子，即参数 $\overline{r}$ 和 M'。这两个参数需要通过软件提前运算得到。

RSA 加速器支持运算子长度为 $N = 32 \times x$ ($x \in \{1, 2, 3, \ldots, 96\}$) 的大数模幂运算。$Z$、$X$、$Y$、$M$ 和 $\overline{r}$ 的位宽为这 96 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

$$b = 2^{32}$$

PRELIMINARY

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 7.3.4 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 96 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，只需要更新被覆盖的存储器即可。

PRELIMINARY

7.3.2 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 96 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 96 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，只需要更新被覆盖的存储器即可。

7.3.3 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 $N = 32 \times x$ $(x \in \{1, 2, 3, \ldots, 48\})$ 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。

PRELIMINARY

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。

7.3.4 控制加速

对于大数模幂运算，ESP32-C3 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 开启加速）
  - 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 开启加速）
  - 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 7-1 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。

PRELIMINARY

表 7-1. 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。

7.4 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 3 系统和存储器 中的表 3-4。

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	384	0x0000	0x017F	读 / 写
RSA_Z_MEM	存储器 Z	384	0x0200	0x037F	读 / 写
RSA_Y_MEM	存储器 Y	384	0x0400	0x057F	读 / 写
RSA_X_MEM	存储器 X	384	0x0600	0x077F	读 / 写

PRELIMINARY

7.5 寄存器列表

本小节的所有地址均为相对于 RSA 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 3 *系统和存储器* 中的表 3-4。

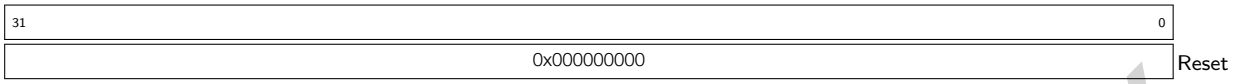
名称	描述	地址	访问
配置寄存器			
RSA_M_PRIME_REG	M' 存储器	0x0800	读 / 写
RSA_MODE_REG	RSA 长度模式	0x0804	读 / 写
RSA_CONSTANT_TIME_REG	固定时间选项	0x0820	读 / 写
RSA_SEARCH_ENABLE_REG	使能 search 加速选项	0x0824	读 / 写
RSA_SEARCH_POS_REG	search 起始位置	0x0828	读 / 写
状态/控制寄存器			
RSA_CLEAN_REG	RSA 清除寄存器	0x0808	只读
RSA_MODEXP_START_REG	模幂运算起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

PRELIMINARY

7.6 寄存器

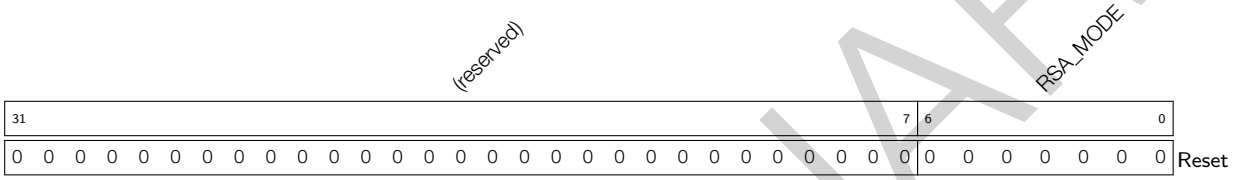
本小节的所有地址均为相对于 RSA 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

Register 7.1. RSA_M_PRIME_REG (0x0800)



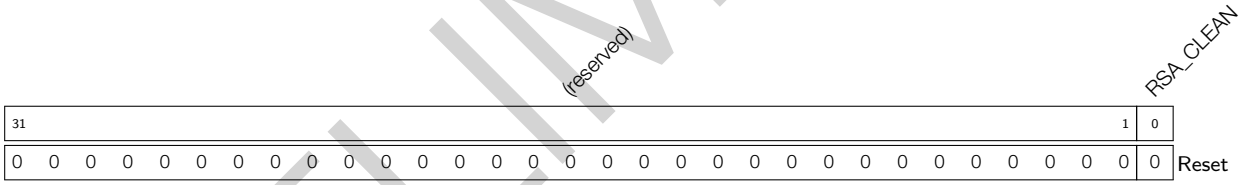
RSA_M_PRIME_REG 此寄存器存储 M'。(读 / 写)

Register 7.2. RSA_MODE_REG (0x0804)



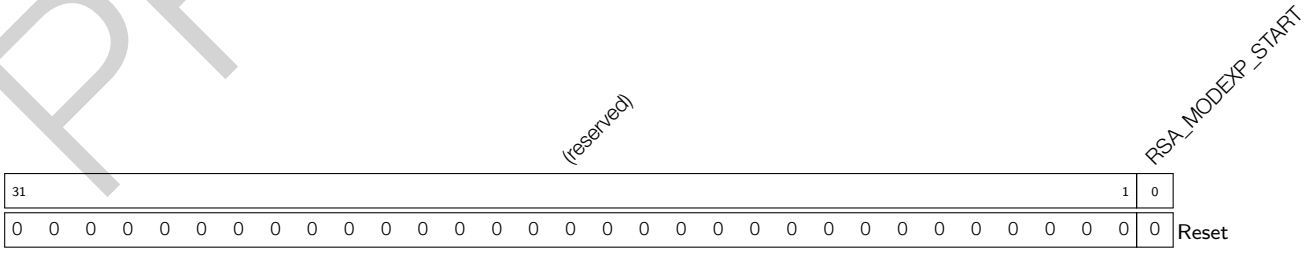
RSA_MODE 此寄存器存储模幂运算的模式。(读 / 写)

Register 7.3. RSA_CLEAN_REG (0x0808)



RSA_CLEAN 一旦存储器初始化结束，此位为 1。(只读)

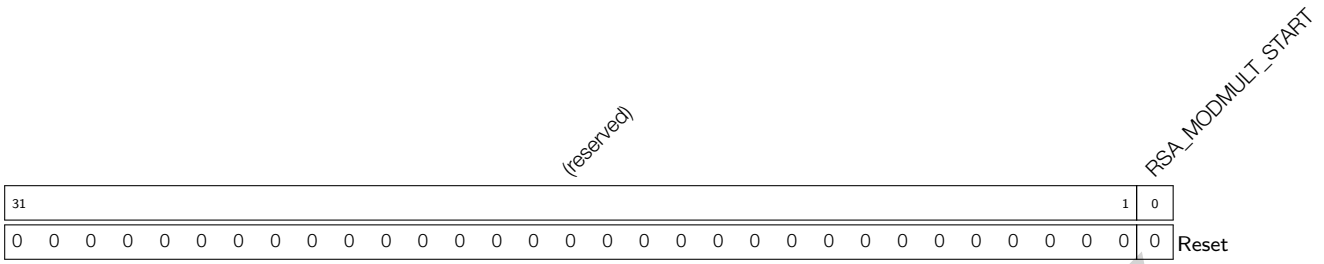
Register 7.4. RSA_MODEXP_START_REG (0x080C)



RSA_MODEXP_START 写入 1 以开始模幂运算。(只写)

PRELIMINARY

Register 7.5. RSA_MODMULT_START_REG (0x0810)



RSA_MODMULT_START 写入 1 以开始模乘运算。(只写)

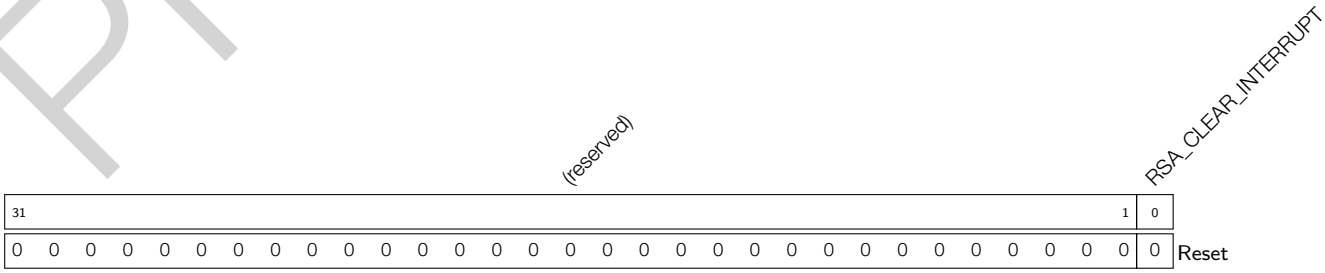
Register 7.6. RSA_MULT_START_REG (0x0814)

RSA_MULT_START 写入 1 以开始乘法运算。(只写)

Register 7.7. RSA_IDLE_REG (0x0818)

RSA_IDLE 当 RSA 空闲时，此位为 1。(只读)

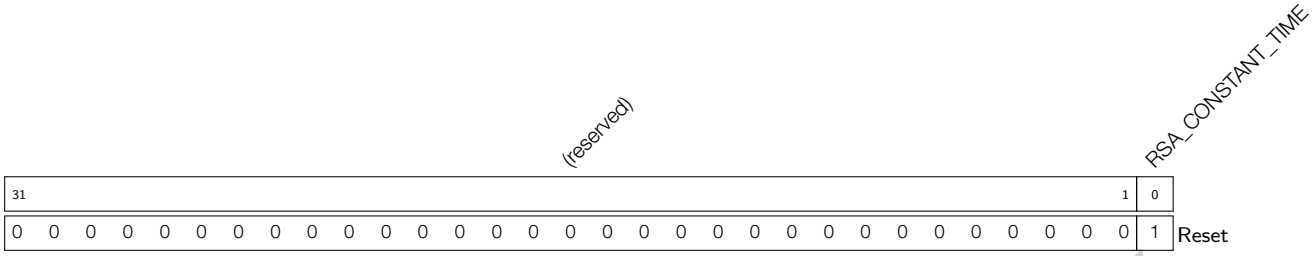
Register 7.8. RSA_CLEAR_INTERRUPT_REG (0x081C)



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

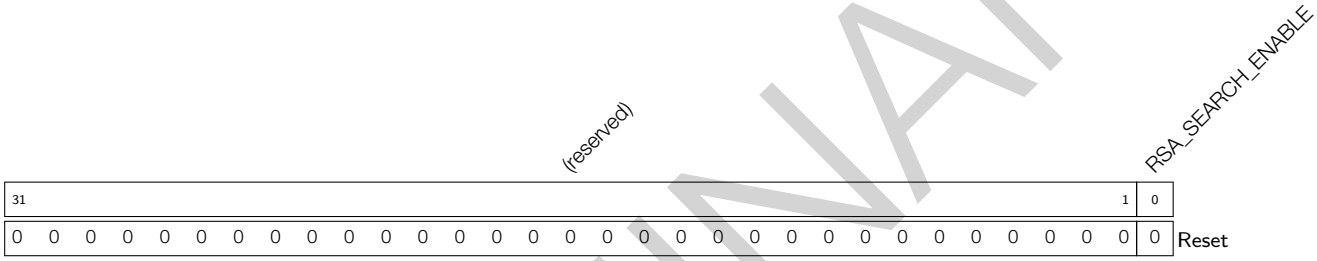
PRELIMINARY

Register 7.9. RSA_CONSTANT_TIME_REG (0x0820)



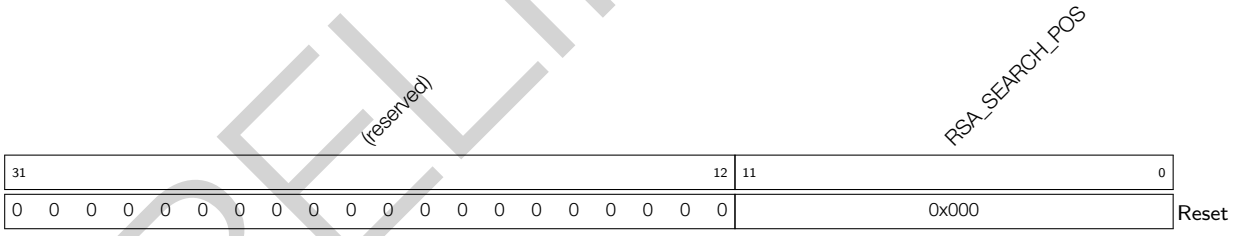
RSA_CONSTANT_TIME_REG 控制模幂运算中的 constant_time 选项。0：加速；1：不加速（默认）。（读 / 写）

Register 7.10. 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）

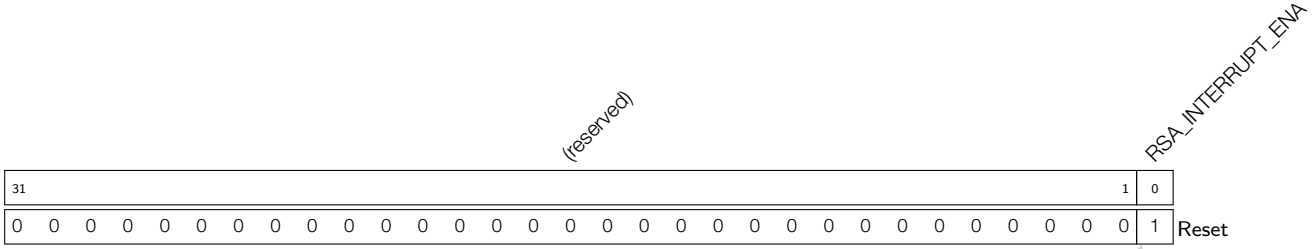
Register 7.11. RSA_SEARCH_POS_REG (0x0828)



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）。

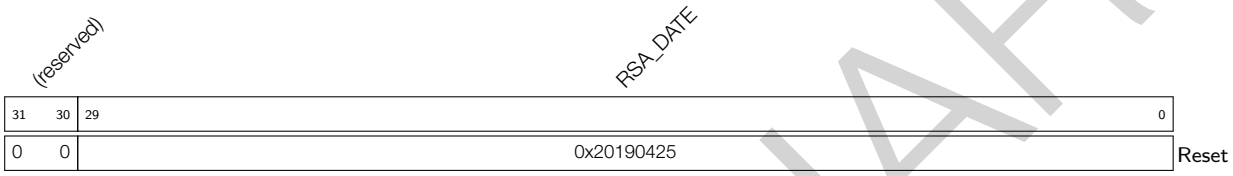
PRELIMINARY

Register 7.12. 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 7.13. RSA_DATE_REG (0x0830)



RSA_DATE 版本控制寄存器（读 / 写）。

PRELIMINARY

8 芯片 Boot 控制

8.1 概述

ESP32-C3 共有四个 Strapping 管脚：

- GPIO2
- GPIO8
- GPIO9
- GPIO10

Strapping 管脚可用于控制 ESP32-C3 芯片上电或硬件复位时的一些功能：

- 控制 Boot 模式
- 控制 ROM 代码信息打印到 UART
- 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 1 复位和时钟章节），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。GPIO2、GPIO8 和 GPIO9 锁存的状态可以通过软件从寄存器 GPIO_STRAPPING 中读取。

GPIO9 默认连接内部上拉电阻。如果这一管脚没有外部连接或者连接的外部线路处于高阻抗状态，内部弱上拉将决定这一管脚输入电平的默认值，如表 8-1 所示。

表 8-1. 管脚默认上拉/下拉

管脚	默认值
GPIO2	N/A
GPIO8	N/A
GPIO9	上拉
GPIO10	N/A

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-C3 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。

8.2 Boot 模式控制

复位释放后，GPIO2、GPIO8 和 GPIO9 共同控制 Boot 模式。

表 8-2. 系统启动模式

管脚	SPI Boot 模式	Download Boot 模式
GPIO2	1	1
GPIO8	x	1
GPIO9	1	0

表 8-2 列出了 GPIO2、GPIO8 和 GPIO9 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。

PRELIMINARY

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 UART0 或 USB 接口将代码下载至 SRAM 或 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。

8.3 ROM 代码打印控制

在系统启动早期阶段，GPIO8 与 eFuse UART_PRINT_CONTROL 一起控制 ROM 代码打印。

表 8-3. ROM 代码打印控制

eFuse[1]	GPIO8	ROM 代码打印
0	-	启动过程中，ROM 代码始终打印至 UART，此时不使用 GPIO8
1	0	启动过程中使能打印
	1	启动过程中关闭打印
2	0	启动过程中关闭打印
	1	启动过程中使能打印
3	-	启动过程中始终关闭打印，此时未使用 GPIO8

[1] eFuse: EFUSE_UART_PRINT_CONTROL

ROM 代码上电默认打印至 U0TXD，也可配置成打印到 USB Serial/JTAG 控制器，具体由 EFUSE_USB_PRINT
_CHANNEL 控制：

- 0：打印至 USB
- 1：打印至 UART

PRELIMINARY

注意：如果此 eFuse 设置为 0，即选择打印至 USB，但如果 USB Serial/JTAG 控制器已被禁用的话，则 ROM 代码将无法打印。

8.4 JTAG 信号源控制

在系统启动早期阶段，GPIO10 与 EFUSE_DIS_PAD_JTAG、EFUSE_DIS_USB_JTAG 和 EFUSE_JTAG_SEL_ENABLE 一起控制 JTAG 信号源，见表 8-4。

表 8-4. JTAG 信号源控制

eFuse 1[a]	eFuse 2[b]	eFuse 3[a]	GPIO 10	JTAG 信号源
0	0	0	-	JTAG 信号来自 USB Serial/JTAG 控制器，未使用 GPIO10
		1	0	JTAG 信号来自相应管脚
			1	JTAG 信号来自 USB Serial/JTAG 控制器
0	1	-	-	JTAG 信号来自相应管脚，未使用 EFUSE_JTAG_SEL_ENABLE 和 GPIO10
1	0	-	-	JTAG 信号来自 USB Serial/JTAG 控制器，未使用 EFUSE_JTAG_SEL_ENABLE 和 GPIO10
1	1	-	-	JTAG 被禁用，未使用 EFUSE_JTAG_SEL_ENABLE 和 GPIO10

[a] eFuse 1: EFUSE_DIS_PAD_JTAG
[b] eFuse 2: EFUSE_DIS_USB_JTAG
[c] eFuse 3: EFUSE_JTAG_SEL_ENABLE

8.5 USB Serial/JTAG 控制器

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。

PRELIMINARY

9 ESP-RISC-V CPU

9.1 概述

ESP-RISC-V CPU 是基于 RISC-V ISA 的 32 位内核，包括基本整数 (I)，乘法/除法 (M) 和压缩 (C) 标准扩展。ESP-RISC-V CPU 内核具有 4 级有序标量流水线，针对面积、功耗、性能等进行了优化。CPU 内核架构包含中断控制器 (INTC)、调试模块 (DM) 和用于访问存储器和外设的系统总线 (SYS BUS) 接口。

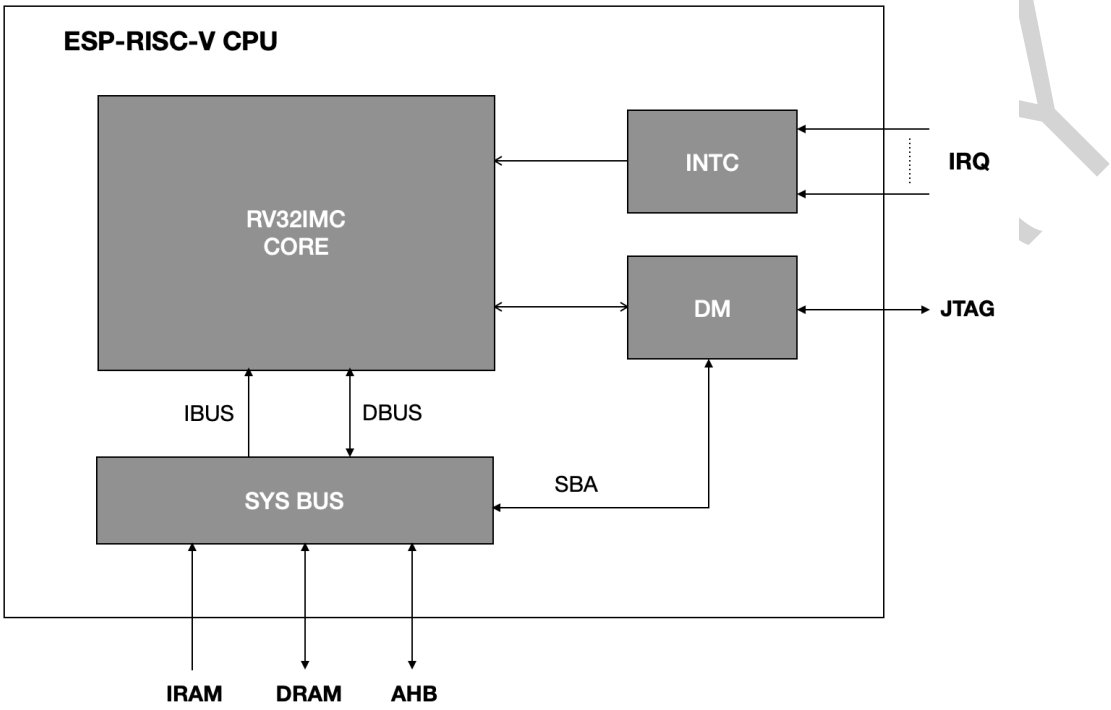


图 9-1. CPU 框图

9.2 特性

- 时钟工作频率高达 160 MHz
- 通过 IRAM/DRAM 接口零等待周期访问片上 SRAM 和缓存中的程序和数据
- 中断控制器 (INTC) 具有多达 31 个向量中断，可配置优先级和阈值级别
- 调试模块 (DM) 符合 RISC-V 调试规范 v0.13，支持通过行业标准的 JTAG/USB 端口连接外部调试器
- 调试器通过系统总线 (SBA) 直接访问存储器和外设
- 硬件触发器符合 RISC-V 调试规范 v0.13，具有多达 8 个断点/观察点
- 物理存储器保护 (PMP)，最多可配置 16 个区域
- 32 位 AHB 系统总线，用于访问外设
- 可配置的核心性能指标事件

PRELIMINARY

9.3 地址分布

下表列出了 CPU 可访问的指令地址空间，数据地址空间，调试地址空间和通过系统总线访问的外设地址空间。

表 9-1. CPU 地址分布

名称	描述	起始地址	结束地址	访问
IRAM	指令地址空间	0x4000_0000	0x47FF_FFFF	读/写
DRAM	数据地址空间	0x3800_0000	0x3FFF_FFFF	读/写
DM	调试地址空间	0x2000_0000	0x27FF_FFFF	读/写
AHB	AHB 地址空间	* 默认	* 默认	读/写

* 默认：IRAM、DRAM、DM 地址范围以外的地址空间通过 AHB 总线访问。

9.4 配置与状态寄存器 (CSR)

9.4.1 寄存器列表

下表为 CPU 可访问的 CSR 列表。除了自定义的性能计数器 CSR 外，所有已实现的 CSR 都遵循 RISC-V 指令集手册 V1.10 第二卷 “特权架构” (RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10) 中所述的位域标准映射。必须注意的是，受 CPU 中实现的功能子集的限制，即使在标准 CSR 中也并非实现了所有位域。有关详细的 CSR 寄存器描述，请参阅下一小节。

名称	描述	地址	访问
机器模式信息 CSR			
mvendorid	机器模式供应商编号寄存器	0xF11	只读
marchid	机器模式架构编号寄存器	0xF12	只读
mimpid	机器模式硬件实现编号寄存器	0xF13	只读
mhartid	机器模式硬件线程编号寄存器	0xF14	只读
机器模式异常设置 CSR			
mstatus	机器模式状态寄存器	0x300	读/写
misa [1]	机器模式 ISA 寄存器	0x301	读/写
mtvec [2]	机器模式异常向量寄存器	0x305	读/写
机器模式异常处理 CSR			
mscratch	机器模式暂存寄存器	0x340	读/写
mepc	机器模式异常程序计数器	0x341	读/写
mcause [3]	机器模式异常原因寄存器	0x342	读/写
mtval	机器模式异常值寄存器	0x343	读/写
物理存储器保护 (PMP) CSR			
pmpcfg0	物理存储器保护配置寄存器	0x3A0	读/写
pmpcfg1	物理存储器保护配置寄存器	0x3A1	读/写
pmpcfg2	物理存储器保护配置寄存器	0x3A2	读/写
pmpcfg3	物理存储器保护地址寄存器	0x3A3	读/写

[1] 尽管 misa 具有读/写属性，但由于它的域是硬连线的，所以写操作无效。在 RISC-V 术语中称为 WARL（写入任意数值读取合法数值）。
[2] mtvec 仅支持在向量模式下对异常处理进行配置，基地址为 256 字节对齐。
[3] mcause 中反映的外部中断 ID 也包括 RISC-V 标准为核心内部中断源预留的 ID。

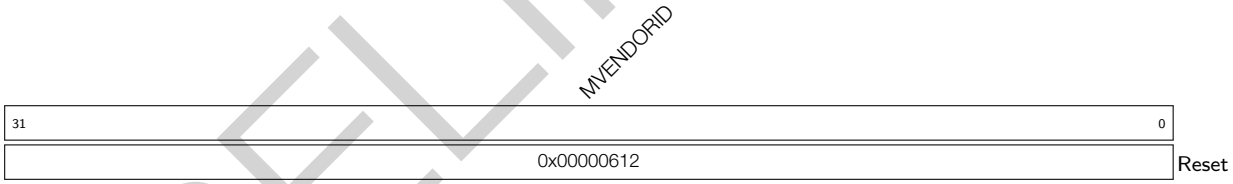
PRELIMINARY

名称	描述	地址	访问
pmpaddr0	物理存储器保护地址寄存器	0x3B0	读/写
pmpaddr1	物理存储器保护地址寄存器	0x3B1	读/写
....			
pmpaddr15	物理存储器保护地址寄存器	0x3BF	读/写
触发器模块 CSR（与调试模块共用）			
tselect	触发器选择寄存器	0x7A0	读/写
tdata1	触发器抽象数据寄存器 1	0x7A1	读/写
tdata2	触发器抽象数据寄存器 2	0x7A2	读/写
tcontrol	全局触发器控制寄存器	0x7A5	读/写
调试模式 CSR			
dcsr	调试模式控制与状态寄存器	0x7B0	读/写
dpc	调试模式 PC 寄存器	0x7B1	读/写
dscratch0	调试模式暂存寄存器 0	0x7B2	读/写
dscratch1	调试模式暂存寄存器 1	0x7B3	读/写
程序计数器 CSR（自定义）[4]			
mpcer	程序计数器事件寄存器	0x7E0	读/写
mpcmr	程序计数器模式寄存器	0x7E1	读/写
mpccr	程序计数器计数寄存器	0x7E2	读/写

请注意，如果对上表中只读属性的任何 CSR 尝试执行写入/置位/清除操作，CPU 将生成非法指令异常。

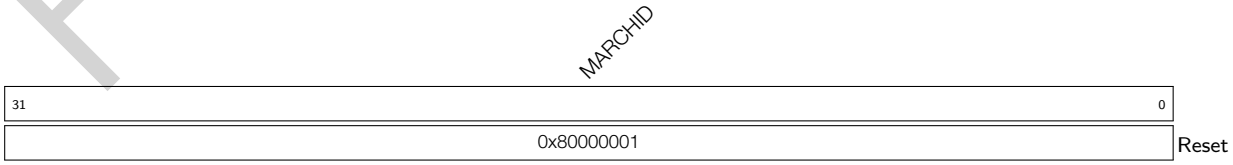
9.4.2 寄存器

Register 9.1. mvendorid (0xF11)



MVENDORID 供应商编号。（只读）

Register 9.2. marchid (0xF12)

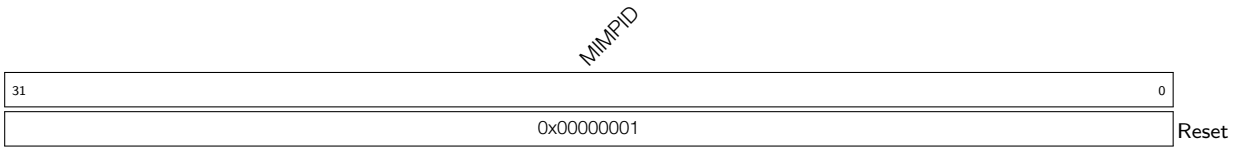


MARCHID 架构编号。（只读）

[4] 这些自定义机器模式 CSR 已经在 RISC-V 标准为用户保留的地址空间中实现。

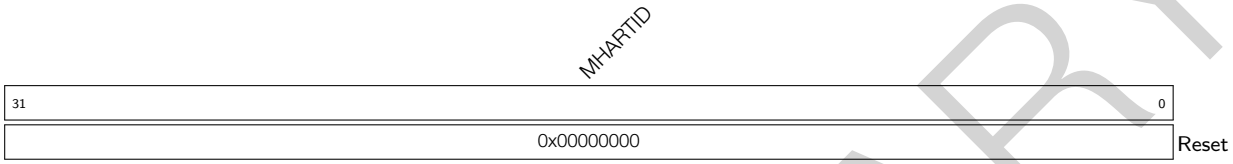
PRELIMINARY

Register 9.3. mimpid (0xF13)



MIMPID 架构编号。(只读)

Register 9.4. mhartid (0xF14)



MHARTID 硬件线程编号。(只读)

Register 9.5. mstatus (0x300)

MIE 全局机器模式中断使能。(读/写)

MPIE 之前的 MIE。(读/写)

MPP 机器之前的特权模式。(读/写)

可能的值:

- 0x0: 用户模式
- 0x3: 机器模式

说明: 仅低位可写。由于高位直接绑定低位，写入高位将被忽略。

TW 超时等待。(读/写)

如果该位置 1，用户模式下的 WFI（等待中断）指令将导致非法指令异常。

PRELIMINARY

Register 9.6. misa (0x301)

MXL	(reserved)	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
31 30	29 26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1	0x0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0

Reset

MXL 机器 XLEN = 1（32 位）。（只读）

Z 保留 = 0。（只读）

Y 保留 = 0。（只读）

X 非标准扩展 = 0。（只读）

W 保留 = 0。（只读）

V 保留 = 0。（只读）

U 实现用户模式 = 1。（只读）

T 保留 = 0。（只读）

S 实现监督模式 = 0。（只读）

R 保留 = 0。（只读）

Q 四精度浮点扩展 = 0。（只读）

P 保留 = 0。（只读）

O 保留 = 0。（只读）

N 支持用户级别中断 = 0。（只读）

M 整数乘除法标准扩展 = 1。（只读）

L 保留 = 0。（只读）

K 保留 = 0。（只读）

J 保留 = 0。（只读）

I RV32I 基本 ISA = 1。（只读）

H 虚拟机管理程序扩展 = 0。（只读）

G 其他标准扩展 = 0。（只读）

F 单精度浮点扩展 = 0。（只读）

E RV32E 基本 ISA = 0。（只读）

D 双精度浮点扩展 = 0。（只读）

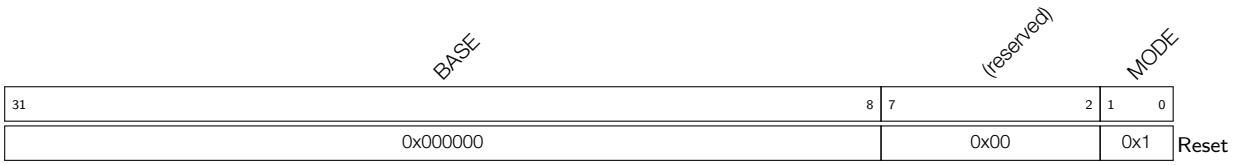
C 压缩标准扩展 = 1。（只读）

B 保留 = 0。（只读）

A 原子标准扩展 = 0。（只读）

PRELIMINARY

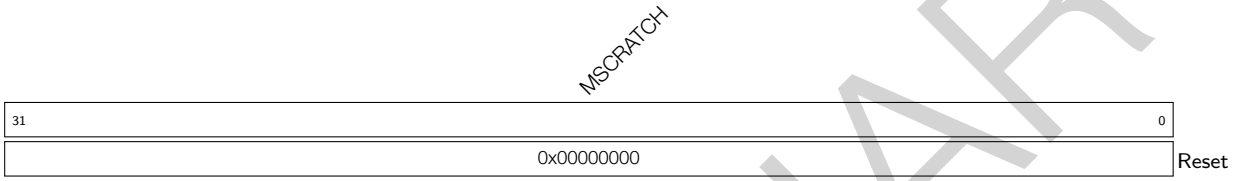
Register 9.7. mtvec (0x305)



MODE 仅支持向量模式 **0x1**。(只读)

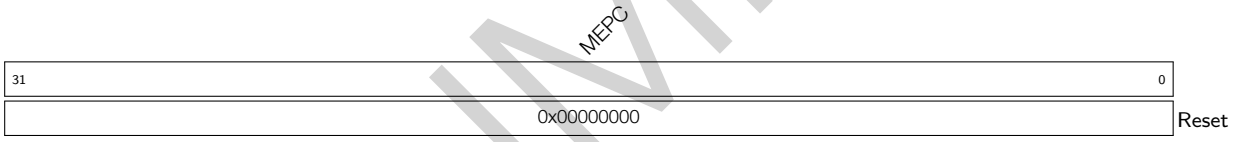
BASE 异常向量基址的高 24 位为 256 字节对齐。(读/写)

Register 9.8. mscratch (0x340)



MSCRATCH 用户自定义的机器暂存寄存器。(读/写)

Register 9.9. mepc (0x341)

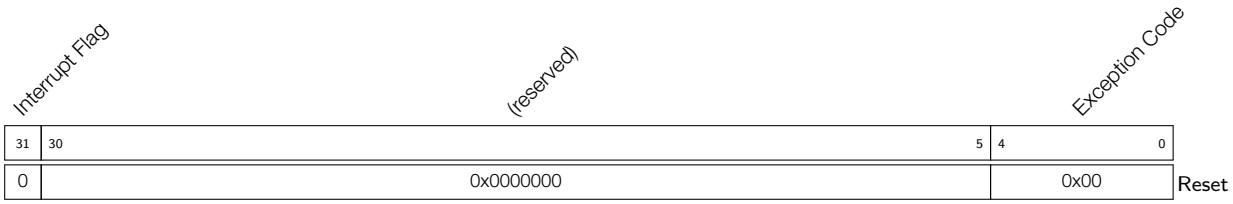


MEPC 机器陷阱/异常程序计数器。(读/写)

当 CPU 遇到异常时，此域将自动更新为 CPU 将要执行的指令的地址。

PRELIMINARY

Register 9.10. mcause (0x342)



Exception Code CPU 进入异常时，此域将自动更新为最近的异常或中断的唯一 ID。(读/写)

可能的异常 ID：

- 0x1：PMP 指令访问错误
- 0x2：非法指令
- 0x3：硬件断点/观察点或 EBREAK
- 0x5：PMP 读存储器访问错误
- 0x7：PMP 写存储器访问错误
- 0x8：用户模式环境调用 (ECALL)
- 0xb：机器模式环境调用

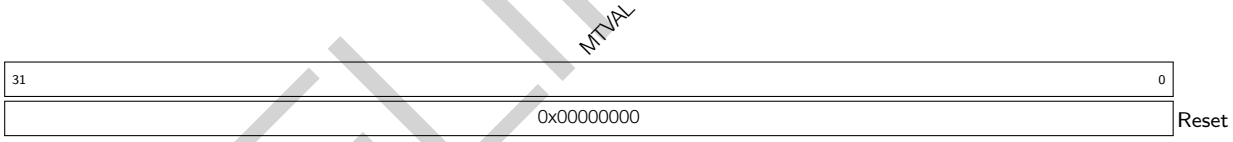
说明：异常 *ID 0x0*（指令地址非对齐）不存在，因为 *CPU* 在取指时始终屏蔽地址的最低位。

Interrupt Flag CPU 进入异常时，此标志位将自动更新。(读/写)

如果被置位，则表示最近的陷阱是由中断引起。在异常情况下保持为 0。

说明：中断控制器将中断编号 *1-31* 全部用于外部中断源，而 *RISC-V* 标准则为内核的内部中断源预留了编号 *0-15*。

Register 9.11. mtval (0x343)



MTVAL 机器模式异常值。(读/写)

将自动更新为与异常有关的数据，该数据可能有助于处理该异常。

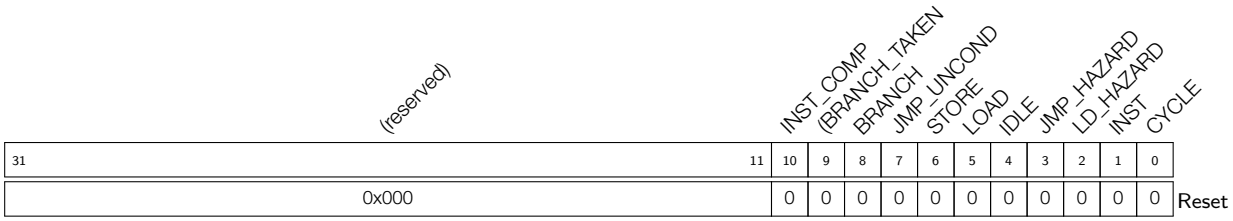
根据异常编号有以下解读：

- 0x1：指令虚拟地址错误
- 0x2：指令 opcode 错误
- 0x5：存储器读操作的数据地址错误
- 0x7：存储器写操作的数据地址错误

说明：该寄存器不支持其他异常 *ID* 和中断。

PRELIMINARY

Register 9.12. mpcer (0x7E0)



INST_COMP 计数压缩指令。(读/写)

BRANCH_TAKEN 采取分支跳转。(读/写)

BRANCH 计数分支。(读/写)

JMP_UNCOND 计数无条件跳转。(读/写)

STORE 计数存储器写操作。(读/写)

LOAD 计数存储器读操作。(读/写)

IDLE 计数 IDLE 周期。(读/写)

JMP_HAZARD 计数跳转冲突。(读/写)

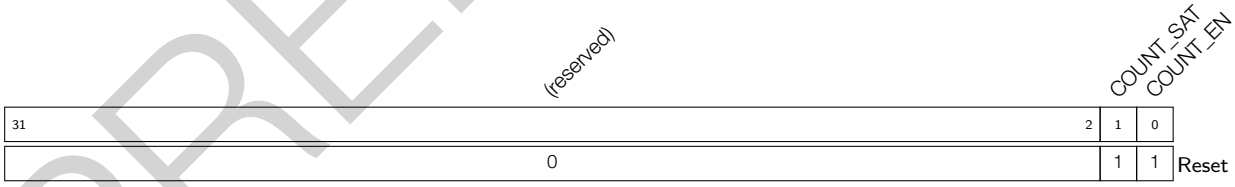
LD_HAZARD 计数存储器读操作冲突。(读/写)

INST 计数指令。(读/写)

CYCLE 计数时钟周期。(读/写)

注意：每个位选择一个特定事件由计数器递增计数。如果多个事件被选择且同时发生，则计数器只递增 *1*。

Register 9.13. mpcmr (0x7E1)



COUNT_SAT 计数器饱和控制。(读/写)

可能的值：

- 0：超出最大值上溢
- 1：超出最大值暂停

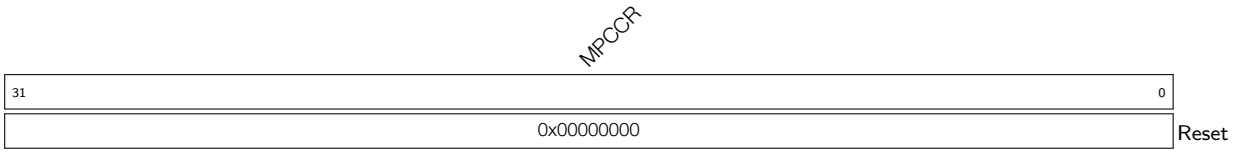
COUNT_EN 计数器使能控制。(读/写)

可能的值：

- 0：禁能
- 1：使能

PRELIMINARY

Register 9.14. mpccr (0x7E2)



MPCCR 程序计数器计数的值。(读/写)

PRELIMINARY

9.5 中断控制器

9.5.1 特性

中断控制器能够捕获、屏蔽来自 RISC-V CPU 外部的中断源，并对中断源的优先级进行动态仲裁。中断控制器具有以下特性：

- 多达 31 个具有唯一 ID (1-31) 的异步中断
- 支持通过读写存储器匹配寄存器进行配置
- 15 个优先级级别，可以分配给不同的中断
- 支持电平触发或边沿触发的中断源
- 可配置的全局阈值，用于屏蔽优先级较低的中断
- 与异常向量地址偏移量匹配的中断 ID

9.5.2 功能描述

每个中断 ID 都有 5 个属性：

1. 使能状态 (0-1)：
   - 决定是否允许由 CPU 捕获和处理中断。
   - 通过写入 INT_ENABLE_REG 相应的域进行配置。
2. 类型 (0-1)：
   - 在中断信号的上升沿使能闩锁状态。
   - 通过写入 INT_TYPE_REG 相应的域进行配置。
   - 类型保持为 0 的中断称为“电平”类型中断。
   - 类型保持为 1 的中断称为“边沿”类型中断。
3. 优先级 (1-15)：
   - 当有多个中断在等待时，决定 CPU 先处理哪一个中断。
   - 通过写入中断 ID *n* (1-31) 的 INT_PRIORITY_*n*_REG 进行配置。
   - 优先级为零或小于 INT_THRESH_REG 指定阈值的中断将被屏蔽。
   - 优先级最低为 1，最高为 15。
   - 具有相同优先级的中断通过其 ID 静态确定优先级，ID 越小，优先级越高。
4. 等待状态 (0-1)：
   - 反映已使能且未被屏蔽的中断信号被捕获时的状态。
   - 通过读取 INT_EIP_REG 中的相应位获得每个中断 ID 的等待状态。
   - 如果没有更高优先级的中断在等待，则当前在等待的中断将导致 CPU 进入异常。
   - 如果在等待的中断抢占 CPU 并导致其跳转到相应的异常向量地址，则称该中断为“已声明”。
   - 所有在等待的中断都为“未声明”。

PRELIMINARY

5. 清除状态 (0-1)：

   - 切换此属性将仅清除已声明的边沿类型中断的等待状态。
   - 通过先置位然后清零 INT_CLEAR_REG 中的相应位进行切换。
   - 电平类型的中断的等待状态不受切换操作的影响，而必须从中断源中清除。
   - 未声明的边沿类型中断的等待状态可以被清空，方法是先清零 INT_ENABLE_REG 中的相应位再置位 INT_CLEAR_REG 中的相同位。

当 CPU 处理在等待的中断时，会进行以下操作：

- 将当前未执行指令的地址保存在 mepc 中，以便之后恢复执行。
- 将 mcause 的值更新为正在处理的中断 ID。
- 将 MIE 的状态复制到 MPIE，然后清零 MIE，从而全局禁用中断。
- 通过跳转到 mtvec 中存储的地址的字对齐偏移量进入异常。

表 9-3 列出了每个中断 ID 及其对应的异常向量地址。简而言之，中断 $ID = i$ 的字对齐的异常地址 = $(mtvec + 4i)$.

说明：$ID = 0$ 不可用，不能用于捕获中断，这是因为相应的异常向量地址 *(mtvec + 0x00)* 已经预留给异常。

表 9-3. 中断 ID 与异常向量地址

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

在跳转到异常向量之后，执行流程取决于软件实现，但一般来说该中断将在某个中断服务程序 (ISR) 中被处理（并清除），然后在 CPU 遇到 MRET 指令后恢复正常程序流。

执行 MRET 指令后，CPU 将进行以下操作：

- 将 MPIE 的状态复制回 MIE，然后清零 MPIE。这意味着，如果之前置位了 MPIE，则执行 MRET 后 MIE 将被置位，进而全局使能中断。
- 跳转到 mepc 中存储的地址，然后恢复执行。

软件可以在 ISR 内部实现中断嵌套，具体请参考章节 9.5.3。

中断控制器具有以下行为特点：

- 仅当中断具有非零优先级、大于或等于阈值寄存器中的值时，它才会反映在 INT_EIP_REG 中。
- 如果一个中断反映在 INT_EIP_REG 中但是还未被处理，则可以通过降低它的优先级或提高全局阈值将其屏蔽（进而防止 CPU 对其进行处理）。

PRELIMINARY

- 如果一个中断反映在 INT_EIP_REG 中，要清除它（防止被处理），则必须将其禁用（如果是边沿属性的中断则需要清除）。

9.5.3 建议操作

9.5.3.1 延迟

配置中断控制器时应考虑延迟问题。

在稳态操作中，中断控制器的等待时间固定为 4 个周期。稳态操作的意思是最近没有对中断控制器寄存器作任何更改。这意味着一个中断从被中断控制器断言到被 CPU 开始处理刚好消耗 4 个周期。这也意味着，在抢占发生之前，CPU 最多可以执行 5 条指令。

当寄存器被修改时，中断控制器会进入临时状态，然后需要最多 4 个周期才能再次进入稳态。在临时状态期间，中断的顺序可能无法预测，因此，需要软件采取一些安全措施以避免任何同步问题。

还须注意的是，中断控制器的配置寄存器位于 APB 地址范围内，因此对这些寄存器的读写访问可能需要消耗几个周期。

考虑到上述特征，建议用户在修改中断控制器寄存器时遵循以下操作顺序：

1. 保存 MIE 的状态，然后将其清零
2. 通过“读-修改-写”的方式写中断控制器寄存器
3. 执行 FENCE 指令以等待所有未完成的写操作完成
4. 最后，恢复 MIE 的状态

如上述步骤显示，建议用户在配置中断控制器寄存器之前先全局禁用中断 (MIE=0)，然后立即恢复 MIE。

执行完上述操作后，中断控制器将恢复稳态操作。

9.5.3.2 配置流程

默认情况下，mstatus 里的 MIE 为 0，即全局禁用中断。在中断堆栈初始化（包括将 mtvec 设置为中断向量地址）完成之后，软件必须将 MIE 置为 1。

在正常情况下，如果要使能某个中断 *n*，可以遵循以下步骤：

1. 保存 MIE 的状态，然后将其清零
2. 根据中断的类型（边沿/电平），置位或取消置位 INT_TYPE_REG 中的第 *n* 个位
3. 通过写入 INT_PRIORITY_*n*_REG 指定优先级（最低为 1，最高为 15）
4. 置位 INT_ENABLE_REG 中的第 *n* 个位
5. 执行 FENCE 指令
6. 恢复 MIE 的状态

当一个或多个中断在等待时，CPU 将确认（声明）最高优先级的中断，然后跳转到与该中断 ID 相对应的异常向量地址。软件可以通过读取 mcause 来推断异常类型（mcause(31) 为 1 代表中断，为 0 代表异常）和中断 ID（mcause(4-0) 提供中断或异常的 ID）。如果异常向量中的每个表项都是指向不同异常处理程序的跳转指令，则软件无需做此推断。最后，异常处理程序会将程序指引到该中断相应的 ISR。

进入 ISR 后，如果中断为边沿类型，则软件必须切换 INT_CLEAR_REG 中的第 *n* 个位，如果是电平类型中断，则必须清除相应的中断源。

PRELIMINARY

软件还可以更新 INT_THRESH_REG 的值并置位 MIE 来让更高优先级的中断抢占当前 ISR（即嵌套），但是，在此之前，必须先保存所有状态 CSR（mepc、mstatus、mcause 等），这是由于发生嵌套时状态 CSR 的值会被覆盖。之后，在退出 ISR 时，再恢复这些 CSR 的值。

最后，程序从 ISR 返回到异常处理程序之后，可以执行 MRET 指令以恢复正常程序流。

如果不再需要中断 *n* 并且需要将其禁用，则可以遵循以下操作步骤：

1. 保存 MIE 的状态，然后将其清零
2. 读取 INT_EIP_REG 检查中断是否在等待
3. 置位/取消置位 INT_ENABLE_REG 中的第 *n* 个位
4. 如果中断属于边沿类型并且在等待，则必须切换 INT_CLEAR_REG 中的第 *n* 个位以清空它的等待状态
5. 执行 FENCE 指令
6. 恢复 MIE 的状态

以上只是建议的操作方案，实际操作由软件实现决定。

9.5.4 寄存器列表

本小节的所有地址均为相对于中断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 3 *系统和存储器* 中的表 3-4。

名称	描述	地址	访问
INT_ENABLE_REG	使能 CPU 处理中断	0x0104	读/写
INT_TYPE_REG	指定中断类型为电平/边沿类型	0x0108	读/写
INT_CLEAR_REG	写入清除“脉冲”类型中断	0x010C	读/写
INT_EIP_REG	外部中断的等待状态	0x0110	只读
INT_PRIORITY_1_REG	设置中断 ID=1 的优先级	0x0118	读/写
INT_PRIORITY_2_REG	设置中断 ID=2 的优先级	0x011C	读/写
INT_PRIORITY_3_REG	设置中断 ID=3 的优先级	0x0120	读/写
INT_PRIORITY_4_REG	设置中断 ID=4 的优先级	0x0124	读/写
INT_PRIORITY_5_REG	设置中断 ID=5 的优先级	0x0128	读/写
INT_PRIORITY_6_REG	设置中断 ID=6 的优先级	0x012C	读/写
INT_PRIORITY_7_REG	设置中断 ID=7 的优先级	0x0130	读/写
INT_PRIORITY_8_REG	设置中断 ID=8 的优先级	0x0134	读/写
INT_PRIORITY_9_REG	设置中断 ID=9 的优先级	0x0138	读/写
INT_PRIORITY_10_REG	设置中断 ID=10 的优先级	0x013C	读/写
INT_PRIORITY_11_REG	设置中断 ID=11 的优先级	0x0140	读/写
INT_PRIORITY_12_REG	设置中断 ID=12 的优先级	0x0144	读/写
INT_PRIORITY_13_REG	设置中断 ID=13 的优先级	0x0148	读/写
INT_PRIORITY_14_REG	设置中断 ID=14 的优先级	0x014C	读/写
INT_PRIORITY_15_REG	设置中断 ID=15 的优先级	0x0150	读/写
INT_PRIORITY_16_REG	设置中断 ID=16 的优先级	0x0154	读/写
INT_PRIORITY_17_REG	设置中断 ID=17 的优先级	0x0158	读/写
INT_PRIORITY_18_REG	设置中断 ID=18 的优先级	0x015C	读/写
INT_PRIORITY_19_REG	设置中断 ID=19 的优先级	0x0160	读/写

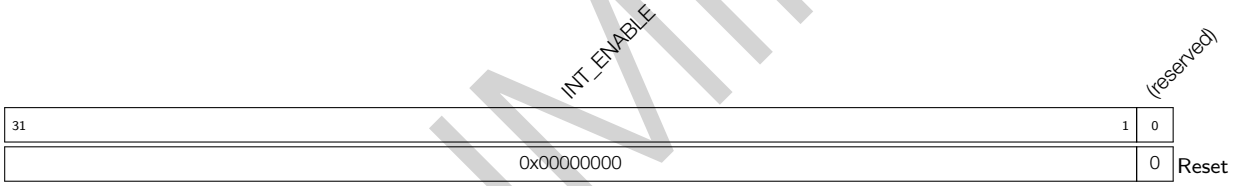
PRELIMINARY

名称	描述	地址	访问
INT_PRIORITY_20_REG	设置中断 ID=20 的优先级	0x0164	读/写
INT_PRIORITY_21_REG	设置中断 ID=21 的优先级	0x0168	读/写
INT_PRIORITY_22_REG	设置中断 ID=22 的优先级	0x016C	读/写
INT_PRIORITY_23_REG	设置中断 ID=23 的优先级	0x0170	读/写
INT_PRIORITY_24_REG	设置中断 ID=24 的优先级	0x0174	读/写
INT_PRIORITY_25_REG	设置中断 ID=25 的优先级	0x0178	读/写
INT_PRIORITY_26_REG	设置中断 ID=26 的优先级	0x017C	读/写
INT_PRIORITY_27_REG	设置中断 ID=27 的优先级	0x0180	读/写
INT_PRIORITY_28_REG	设置中断 ID=28 的优先级	0x0184	读/写
INT_PRIORITY_29_REG	设置中断 ID=29 的优先级	0x0188	读/写
INT_PRIORITY_30_REG	设置中断 ID=30 的优先级	0x018C	读/写
INT_PRIORITY_31_REG	设置中断 ID=31 的优先级	0x0190	读/写
INT_THRESH_REG	设置中断优先级阈值	0x0194	读/写

9.5.5 寄存器

本小节的所有地址均为相对于中断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 3 *系统和存储器* 中的表 3-4。

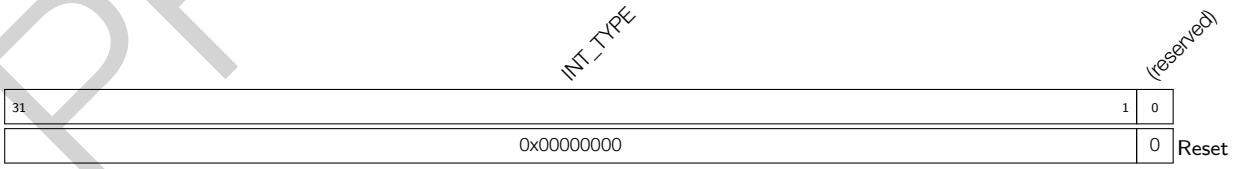
Register 9.15. INT_ENABLE_REG (0x0104)



INT_ENABLE[*n*] 置位第 *n* 个位使能中断 *n*。（读/写）

- 0：禁能
- 1：使能

Register 9.16. INT_TYPE_REG (0x0108)

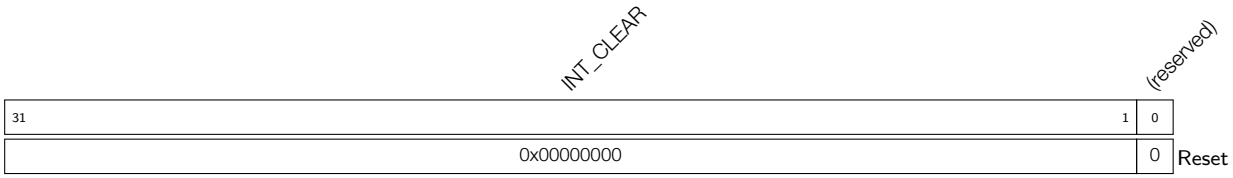


INT_TYPE[*n*] 置位第 *n* 个位捕获中断 *n* 的上升沿。（读/写）

- 0：电平类型（检测信号电平）
- 1：脉冲类型（检测上升沿）

PRELIMINARY

Register 9.17. INT_CLEAR_REG (0x010C)



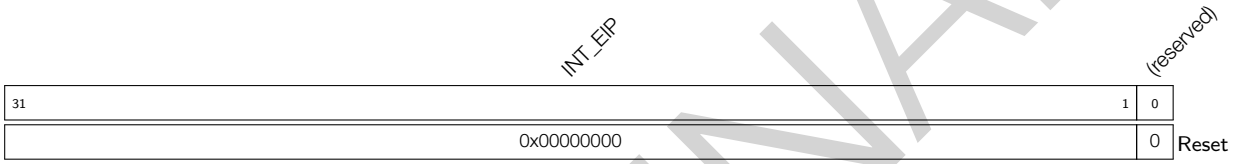
INT_CLEAR[*n*] 置位第 *n* 个位清除中断 *n* 的等待状态。(读/写)

仅对“脉冲”类型的中断有效，“电平”类型中断应在中断源进行清除。

注意置位后需要手动切换至 0。

- 0：无关项
- 1：清除等待状态

Register 9.18. INT_EIP_REG (0x0110)

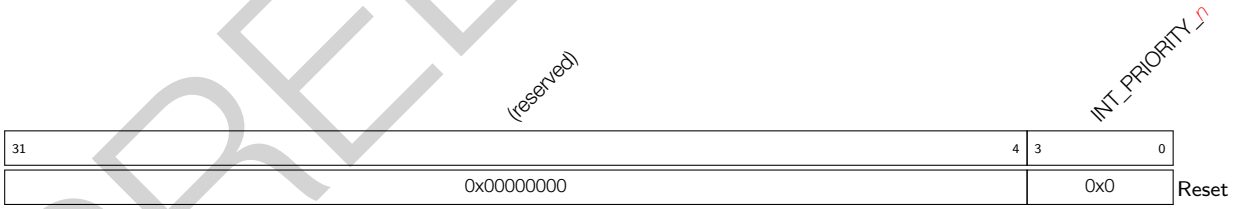


INT_EIP[*n*] 读取第 *n* 个位获得中断 *n* 的等待状态。(只读)

只有被使能的且阈值以上的中断才会反映在该域中。

- 0：不在等待
- 1：在等待

Register 9.19. INT_PRIORITY_*n*_REG (*n*: 1-31) (0x0114+4n*)**

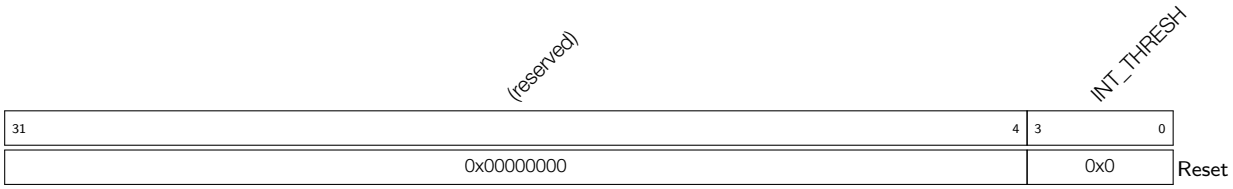


INT_PRIORITY_*n* 写入 4 位数值到第 *n* 个寄存器配置中断 *n* 的优先级。(读/写)

说明：不管阈值是多少，优先级为 *0* 的中断都会被屏蔽。

PRELIMINARY

Register 9.20. INT_THRESH_REG (0x0194)



INT_THRESH 写入 4 位的数值配置所有中断的全局优先级阈值。(读/写)

所有优先级低于阈值的中断都会被屏蔽。

说明：不管阈值是多少，优先级为 0 的中断都会被屏蔽。

PRELIMINARY

9.6 调试

9.6.1 概述

本节介绍如何调试和测试在 CPU 内核上运行的软件。调试功能由标准 JTAG 管脚提供，并符合 RISC-V 外部调试支持规范版本 0.13 (RISC-V External Debug Support Specification version 0.13)。

图 9-2 为外部调试系统架构图。

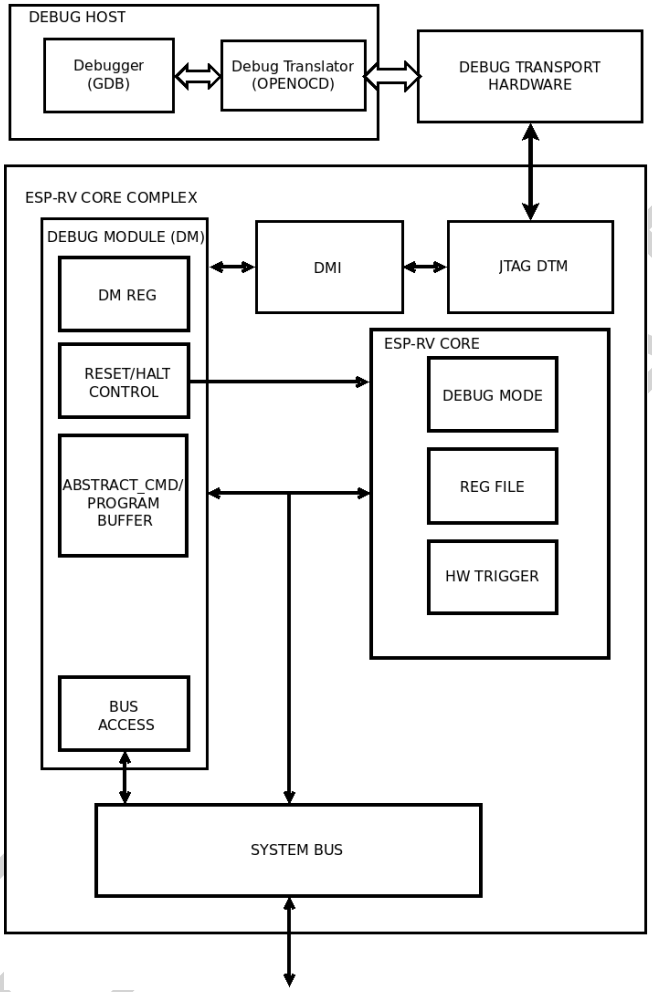


图 9-2. 调试系统架构

用户与运行调试器（Debugger，例如 GDB）的调试主机（DEBUG HOST，例如笔记本电脑）进行交互。调试器通过调试转换器（Debug Translator，可能包含硬件驱动，例如 OPENOCD）与调试传输硬件（DEBUG TRANSPORT，例如 Olimex USB-JTAG 适配器）进行通信。调试传输硬件通过标准 JTAG 接口将调试主机连接到 ESP-RV 内核的调试传输模块 (JTAG DTM)。JTAG DTM 使用调试模块接口 (DMI) 提供对调试模块 (DM) 的访问。

DM 允许调试器暂停内核。抽象命令提供对 GPR（通用寄存器）的访问。程序缓冲区允许调试器在内核上执行任意代码，从而读取 CPU 内核的其他运行状态。CPU 内核的其他运行状态也可以由其他抽象命令读取。ESP-RV 内核带有一个支持 8 个触发器的触发器模块。当满足触发条件时，内核将自发暂停并通知调试模块。

系统总线访问的 block 无需使用 RISC-V 内核即可访问存储器和外设寄存器。

PRELIMINARY

9.6.2 特性

基础调试功能具有以下特性：

- 向调试器提供有关实现的必要信息
- 支持暂停和恢复 CPU 内核
- CPU 内核寄存器（包括 CSR）可以由调试器读取/写入
- CPU 可以从复位后执行的第一条指令开始就被调试
- 可以通过调试器复位 CPU 内核
- 可以在软件断点（植入的断点指令）上暂停 CPU
- 硬件单步调试
- 通过程序缓冲区在暂停的 CPU 中执行任意指令。支持 16 字的程序缓冲区。
- 支持系统总线的字对齐地址访问
- 支持八个硬件触发器（可用作断点/观察点），具体见章节 9.7

9.6.3 功能描述

调试机制遵守 RISC-V 外部调试支持规范版本 0.13。有关调试功能的详细介绍，请参考 RISC-V 外部调试支持规范。

9.6.4 寄存器列表

下表列出了 ESP-RV 内核支持的调试 CSR。

名称	描述	地址	访问
dcsr	调试控制和状态寄存器	0x7B0	读/写
dpc	调试 PC 寄存器	0x7B1	读/写
dscratch0	调试暂存寄存器 0	0x7B2	读/写
dscratch1	调试暂存寄存器 1	0x7B3	读/写

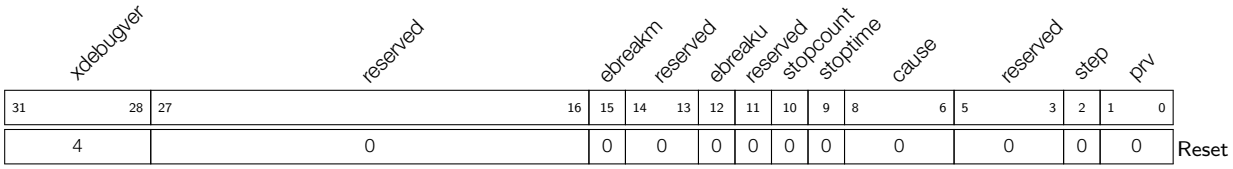
所有调试模块寄存器的实现均符合 RISC-V 外部调试支持规范版本 0.13。请参考 RISC-V 外部调试支持规范获取详细信息。

9.6.5 寄存器

以下是 ESP-RV 内核支持的调试 CSR 的详细描述。

PRELIMINARY

Register 9.21. dcsr (0x7B0)



xdebugver 调试版本。(只读)

- 4：存在外部调试支持

ebreakm 置位后，机器模式中的 ebreak 指令进入调试模式。(读/写)

ebreaku 置位后，用户/程序模式中的 ebreak 指令进入调试模式。(读/写)

stopcount 此域没有实现。调试器会始终读出 0。(只读)

stoptime 此性能没有实现。调试器会始终读出 0。(只读)

cause 说明进入调试模式的原因。当单个周期中有多个原因导致进入调试模式，会反映出具有最高优先级数值的那个原因。(只读)

1. 执行了一条 ebreak 指令（优先级 3）
2. 触发模块引起暂停（优先级 4）
3. haltreq 被置位（优先级 2）
4. step 被置位导致 CPU 单步执行（优先级 1）

其他值保留供以后使用。

step 当被置位且不处于调试模式时，内核将仅执行单个指令，然后进入调试模式。当该位置 1 时，中断被**使能** *。如果指令由于异常而未能完成，则内核将在执行异常处理程序之前立即进入调试模式，并置位相应的异常寄存器。(读/写)

prv 保存 CPU 进入调试模式时候的特权级别。退出调试模式时，调试器可以更改此值以改变内核的特权级别。仅支持 **0x3**（机器模式）和 **0x0**（用户模式）。

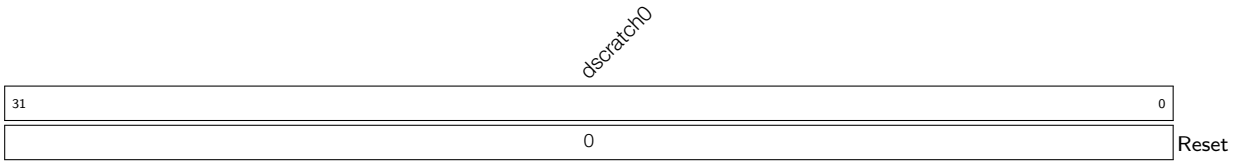
* **注意**：与 RISC-V 调试规格版本 0.13 不同。

Register 9.22. dpc (0x7B1)

dpc 进入调试模式后，dpc 将写入遇到异常的指令的虚拟地址。恢复执行时，CPU 内核的 PC 将更新为 dpc 保存的虚拟地址。调试器可以写入 dpc 配置 CPU 恢复执行的位置。(读/写)

PRELIMINARY

Register 9.23. dscratch0 (0x7B2)



dscratch0 供调试模块内部使用。(读/写)

Register 9.24. dscratch1 (0x7B3)

dscratch1 供调试模块内部使用。(读/写)

PRELIMINARY

9.7 硬件触发器

9.7.1 特性

硬件触发器模块提供了断点和观察点功能，供调试使用。硬件触发器具有以下特性：

- 8 个独立触发单元
- 每个单元都可以配置为匹配程序计数器的地址或存储器访问地址
- 可以通过引起断点异常来抢占执行
- 可以暂停执行并将控制权转移给调试器
- 支持 NAPOT（2 的幂次方对齐）地址编码

9.7.2 功能描述

硬件触发器模块提供了 4 个 CSR，见寄存器列表。其中，tdata1 和 tdata2 是抽象 CSR，也就是说它们是用于访问某个触发单元中的内部寄存器的影子寄存器，一次访问一个触发单元。

要选择特定的触发单元，需要将相应的编号 (0-7) 写入 tselect CSR。当写入有效数值时，抽象 CSR tdata1 和 tdata2 将自动匹配该触发单元的内部寄存器。每个触发单元都有两个内部寄存器，即 mcontrol 和 maddress，它们分别与 tdata1 和 tdata2 匹配。

向 tselect 写入超过最大编号的数值时会导致该数值被裁剪为最大的编号，此编号可以被读回。这个特性可用于枚举初始化期间或使用调试器时可用的触发器。

由于软件或调试器可能需要知道所选触发器的类型以便正确解读 tdata1 和 tdata2，因此 tdata1 的 4 个位 (31-28) 对所选触发器的类型进行了编码。此域为只读访问属性，并且值始终为 0x2，代表匹配类型触发器，因此，可以推断 tdata1 和 tdata2 会通过 mcontrol 和 maddress 被解读。RISC-V 调试规范 v0.13 提供了其他可能值的信息，但是该触发模块仅支持 0x2 类型。

一旦选定了触发单元，就可以通过置位 mcontrol CSR (tdata1) 中相应的域并将目标地址写入 maddress CSR (tdata2) 来对该触发单元进行配置。

通过写入 mcontrol 的 action 域，可以将每个触发单元配置为引起断点异常或进入调试模式。该域只能从调试器写入，因此默认情况下，触发器（如果启用）将引起断点异常。

每个触发单元的 mcontrol 都有一个 hit 域。在 CPU 暂停或进入异常后，通过读取该域可以查明是否是触发单元触发了。触发器触发后该域会立即被置位，但在恢复操作之前必须被手动清零，虽然不清零不会影响正常执行。

每个触发单元仅支持地址匹配，该地址可以是存储器访问地址，也可以是指令的虚拟地址。通过写入所选触发单元的 maddress (tdata2) CSR，可以指定区域的地址和大小。大于 1 个字节的区域大小通过 NAPOT 编码（见表 9-6）指定，并通过置位 mcontrol 中 match 域来使能。注意，根据定义，NAPOT 编码地址的起始地址与区域大小对齐（即，是区域大小的整数倍）。

表 9-6. NAPOT 编码的 maddress

maddress(31-0)	起始地址	大小（字节）
aaa...aaaaaaaaa0	aaa...aaaaaaaaa0	2
aaa...aaaaaaaa01	aaa...aaaaaaaa00	4
aaa...aaaaaaa011	aaa...aaaaaaa000	8
aaa...aaaaaa0111	aaa...aaaaaa0000	16

PRELIMINARY

....		
a01...1111111111	a00...0000000000	2^{31}

tcontrol CSR 对所有触发单元都是通用的。在机器模式下，当程序在异常处理程序中执行时，该寄存器可用于阻止触发器重复引起异常。默认情况下 ISR 内部的断点异常也被禁用，但是，出于调试目的，可以在进入 ISR 之前手动使能断点异常。如果将触发器配置为进入调试模式，则此 CSR 不相关。

9.7.3 触发执行流程

当触发器触发引起硬件线程暂停并进入调试模式时 (action = 1)：

- dpc 被设置为当前 PC（在解码阶段）
- dcsr 的 cause 域被设置为 2，表示暂停是由于触发器触发引起
- 与触发的触发器对应的 hit 域被置位

当触发器触发引起硬件线程进入异常时 (action = 0)：

- mepc 被设置为当前 PC（在解码阶段）
- mcause 被设置为 3，即断点异常
- mpte 被设置为异常发生之前的 mte 的值
- mte 被设置为 0
- 与触发的触发器对应的 hit 域被置位

说明：如果两个触发器同时触发，一个 *action = 0*，*action = 1*，则硬件线程会暂停并进入调试模式。

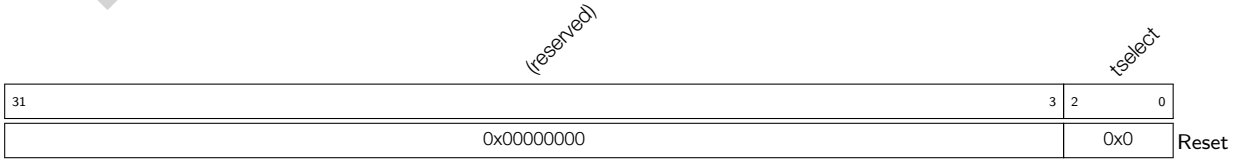
9.7.4 寄存器列表

下表列出了 CPU 可访问的的触发模块 CSR，只有在机器模式下才可以对它们进行读写。

名称	描述	地址	访问
tselect	触发器选择寄存器	0x7A0	读/写
tdata1	触发器抽象数据寄存器 1	0x7A1	读/写
tdata2	触发器抽象数据寄存器 2	0x7A2	读/写
tcontrol	全局触发器控制寄存器	0x7A5	读/写

9.7.5 寄存器

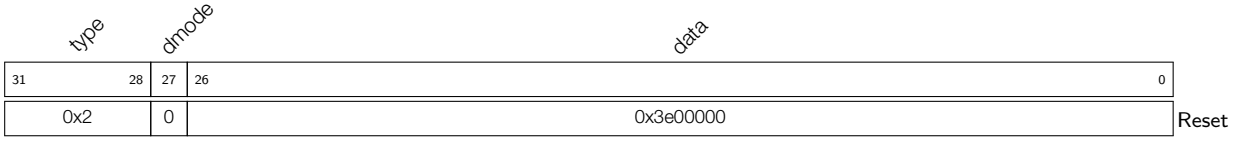
Register 9.25. tselect (0x7A0)



tselect 触发器单元编号 (0-7)。(读/写)

PRELIMINARY

Register 9.26. tdata1 (0x7A1)



type 触发器类型。(只读)

仅支持匹配类型 (0x2)，此域保留。

dmode 如果某触发器正在被调试器使用，则此域置为 1.(读/写 *)

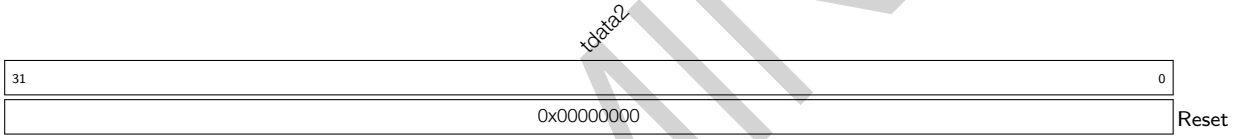
- 0：在调试模式和机器模式下都能写入 tdata1 和 tdata2
- 1：只有在调试模式下才能写入 tdata1 和 tdata2。其他模式下的写操作将被忽略。

* 说明：仅支持调试模式下的写操作。

data 保存抽象 tdata1 的内容。(读/写)

由于仅支持匹配类型 (0x2) 触发器，此域将始终被解读为 mcontrol 的域。

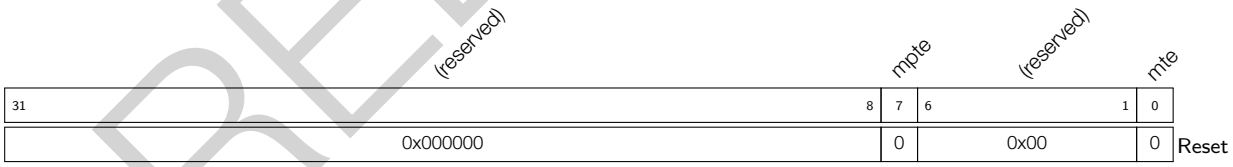
Register 9.27. tdata2 (0x7A2)



tdata2 保存抽象 tdata2 的内容。(读/写)

由于仅支持匹配类型 (0x2) 触发器，此域将始终被解读为 maddress。

Register 9.28. tcontrol (0x7A5)



mpte 机器模式下前一个触发器使能域。(读/写)

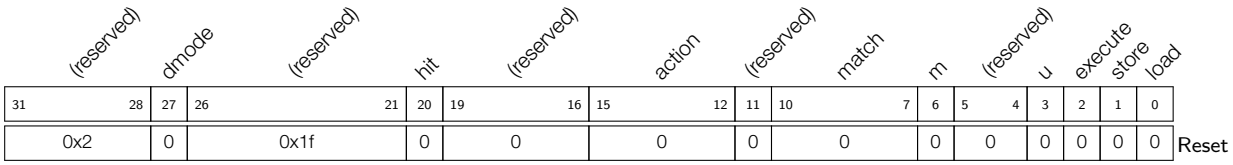
- 当 CPU 在机器模式下进入异常，mte 的值会自动写入此域。
- 当 CPU 执行 MRET，此域的值会返回 mte，此域变为 0。

mte 机器模式下触发器使能域。(读/写)

- 当 CPU 在机器模式下进入异常，此域的值会自动写入 mpte，然后此域变为 0，并且 action=0 的触发器被全局禁用。
- 当 CPU 执行 MRET，mpte 的值会自动返回此域。

PRELIMINARY

Register 9.29. mcontrol (0x7A1)



dmode 与 tdata1 的 dmode 一致。

hit 如果选定的触发器之前触发过，则此域为 1。(读/写)
此域必须手动清零。

action 配置选定的触发器在触发时进行以下操作。(读/写)
有效选项为：

- 0x0：引起断点异常
- 0x1：进入调试模式（仅当 dmode = 1 时有效）

说明：写入无效数值会导致此域变为默认值 *0x0*。

match 配置触发器进行数据/指令地址的下匹配操作。(读/写)
有效选项为：

- 0x0：严格字节匹配，即与访问中某个字节对应的地址必须严格匹配 maddress 的值。
- 0x1：NAPOT 匹配，即访问中至少有一个字节处于 maddress 中规定的 NAPOT 区域。

说明：写入超过最大值的数值会被裁剪为最大值 *0x1*。

m 置位使选定的触发器在机器模式下操作。(读/写)

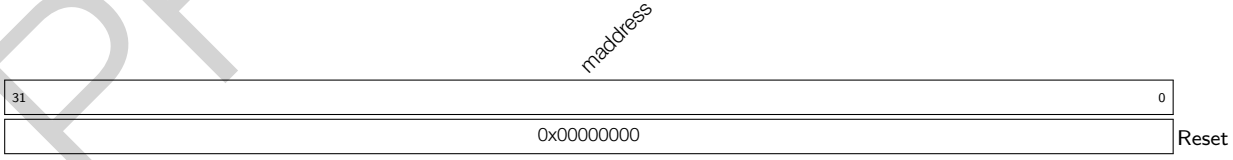
u 置位使选定的触发器在用户模式下操作。(读/写)

execute 置位使选定的触发器在 CPU 执行具有匹配的虚拟地址的指令之前触发。(读/写)

store 置位使选定的触发器在 CPU 执行具有匹配的数据地址的存储器写操作之前触发。(读/写)

load 置位使选定的触发器在 CPU 执行具有匹配的数据地址的存储器读操作之前触发。(读/写)

Register 9.30. maddress (0x7A2)



maddress 选定的触发器执行匹配操作时使用的地址。(读/写)
当 mcontrol 中的 match=1 时由 NAPOT 解码。

PRELIMINARY

9.8 存储器保护

9.8.1 概述

CPU 内核包含一个物理存储器保护单元，可以供软件设置存储器访问特权（读、写、执行权限）。该物理存储器保护单元不完全等同于 RISC-V 指令集手册 V1.10 第二卷“特权架构”中描述的 PMP，下文会详细描述不同之处。

如需了解 RISC-V PMP 的更多信息，请参考 RISC-V 指令集手册 V1.10 第二卷“特权架构”。

9.8.2 特性

PMP 单元用于控制对物理存储器的访问。它支持 16 个区域，可以对最小 4 个字节大小的区域进行权限设定。ESP32-C3 芯片的物理存储器保护单元与 RISC-V 规范中定义的 PMP 的不同之处在于：

- 不支持静态优先级，即不支持重叠区域
- 支持的最大 NAPOT 范围是 1 GB

根据 RISC-V 特权架构规范，PMP 表项是静态优先级排序的，并且，与存储器访问地址的任意字节匹配的最小编号的 PMP 表项将决定该访问是否成功。这意味着，当任意地址匹配多个 PMP 表项，即多个 PMP 表项的重叠区域时，编号最小的 PMP 表项将决定该访问是否成功。

但是，ESP32-C3 的 RISC-V CPU PMP 单元并未实现静态优先级。因此，软件应确保所有使能的 PMP 表项都有唯一的区域，即它们之间没有重叠区域。如果软件仍试图对具有重叠区域且权限相互矛盾的多个 PMP 表项进行配置，只要访问与其中一个 PMP 表项匹配，则访问成功。如果访问与所有使能的 PMP 表项都不匹配，则会产生一个异常。

9.8.3 功能描述

软件可以设置 PMP 单元的配置和地址寄存器，以保存错误并确保安全执行。PMP CSR 只能在机器模式下进行配置。写入、读取和执行权限检测一旦被使能，则将根据 16 个 pmpcfgX 和 pmpaddrX 寄存器（见寄存器列表）中的配置值作用于用户模式下的所有存储器访问。

默认情况下，PMP 允许机器模式下的所有存储器访问，而撤销用户模式下的所有访问。这意味着必须通过 pmpcfg 和 pmpaddr 寄存器（见寄存器列表）设置用户模式可以访问的地址范围和有效权限，以确保访问成功。但是在机器模式下没有此要求，因为机器模式下默认 PMP 允许所有访问。如果在机器模式下也需要 PMP 检测，则软件可以将所需 PMP 表项的锁定位置位来使能权限检测。锁定位一旦置位，就只能通过 CPU 复位被清零。

如果从存储器区域提取指令而没有执行权限，则会在处理器级别生成异常，并且在 mcause CSR 中异常原因被设置为指令访问错误。同样，任何没有有效读/写权限的读写访问都将生成异常，并且 mcause 会更新为读取存储器访问错误或写入存储器访问错误。如果发生存储器读写异常，则存储器访问地址会更新到 mtval CSR 中。

9.8.4 寄存器列表

下表列出了 CPU 可访问的的 PMP CSR，只有在机器模式下才可以对它们进行读写。

名称	描述	地址	访问
pmpcfg0	物理存储器保护配置寄存器	0x3A0	读/写
pmpcfg1	物理存储器保护配置寄存器	0x3A1	读/写

PRELIMINARY

名称	描述	地址	访问
pmpcfg2	物理存储器保护配置寄存器	0x3A2	读/写
pmpcfg3	物理存储器保护配置寄存器	0x3A3	读/写
pmpaddr0	物理存储器保护地址寄存器	0x3B0	读/写
pmpaddr1	物理存储器保护地址寄存器	0x3B1	读/写
pmpaddr2	物理存储器保护地址寄存器	0x3B2	读/写
pmpaddr3	物理存储器保护地址寄存器	0x3B3	读/写
pmpaddr4	物理存储器保护地址寄存器	0x3B4	读/写
pmpaddr5	物理存储器保护地址寄存器	0x3B5	读/写
pmpaddr6	物理存储器保护地址寄存器	0x3B6	读/写
pmpaddr7	物理存储器保护地址寄存器	0x3B7	读/写
pmpaddr8	物理存储器保护地址寄存器	0x3B8	读/写
pmpaddr9	物理存储器保护地址寄存器	0x3B9	读/写
pmpaddr10	物理存储器保护地址寄存器	0x3BA	读/写
pmpaddr11	物理存储器保护地址寄存器	0x3BB	读/写
pmpaddr12	物理存储器保护地址寄存器	0x3BC	读/写
pmpaddr13	物理存储器保护地址寄存器	0x3BD	读/写
pmpaddr14	物理存储器保护地址寄存器	0x3BE	读/写
pmpaddr15	物理存储器保护地址寄存器	0x3BF	读/写

9.8.5 寄存器

PMP 单元实现了 **RISC-V 指令集手册 V1.10 第二卷 “特权架构”** 中定义的所有 pmpcfg0-3 和 pmpaddr0-15 CSR。

PRELIMINARY

Glossary

Abbreviations for Peripherals

AES	AES (Advanced Encryption Standard) Accelerator
BOOTCTRL	Chip Boot Control
DS	Digital Signature
DMA	DMA (Direct Memory Access) Controller
eFuse	eFuse Controller
HMAC	HMAC (Hash-based Message Authentication Code) Accelerator
I2C	I2C (Inter-Integrated Circuit) Controller
I2S	I2S (Inter-IC Sound) Controller
LEDC	LED Control PWM (Pulse Width Modulation)
MCPWM	Motor Control PWM (Pulse Width Modulation)
PCNT	Pulse Count Controller
RMT	Remote Control Peripheral
RNG	Random Number Generator
RSA	RSA (Rivest Shamir Adleman) Accelerator
SDHOST	SD/MMC Host Controller
SHA	SHA (Secure Hash Algorithm) Accelerator
SPI	SPI (Serial Peripheral Interface) Controller
SYSTIMER	System Timer
TIMG	Timer Group
TWAI	Two-wire Automotive Interface
UART	UART (Universal Asynchronous Receiver-Transmitter) Controller
ULP Coprocessor	Ultra-low-power Coprocessor
USB OTG	USB On-The-Go
WDT	Watchdog Timers

Abbreviations for Registers

ISO	Isolation. When a module is power down, its output pins will be stuck in unknown state (some middle voltage). "ISO" registers will control to isolate its output pins to be a determined value, so it will not affect the status of other working modules which are not power down.
NMI	Non-maskable interrupt.
REG	Register.
R/W	Read/write. Software can read and write to these bits.
RO	Read-only. Software can only read these bits.
SYSREG	System Registers
WO	Write-only. Software can only write to these bits.

PRELIMINARY

修订历史

日期	版本	发布说明
2021-04-07	V0.1	Preliminary release

PRELIMINARY

PRELIMINARY

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。

版权归 © 2021 乐鑫信息科技（上海）股份有限公司。保留所有权利。