ESP32-C3

技术参考手册

预发布 v0.2
乐鑫信息科技

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。

目录

表格

插图

1 ESP-RISC-V CPU

1.1 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。

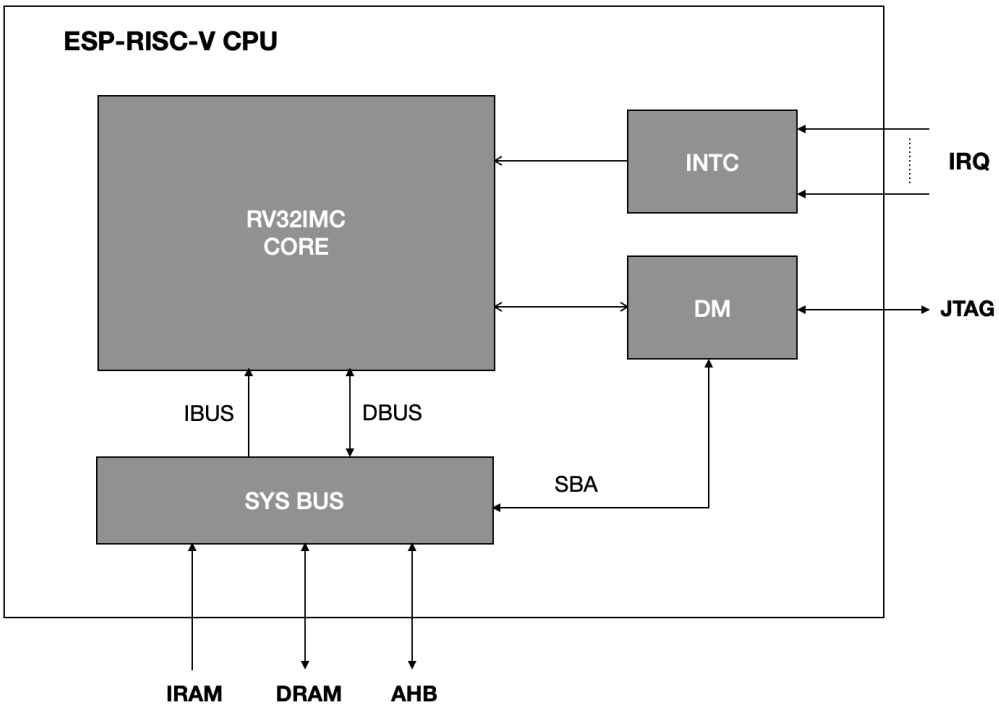


图 1-1. CPU 框图

1.2 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

1.3 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 1-1. 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。

1.4 配置与状态寄存器 (CSR)

1.4.1 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

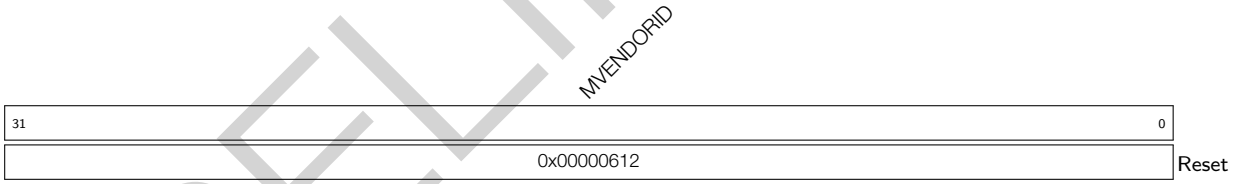
[1] 尽管 misa 具有读/写属性，但由于它的域是硬连线的，所以写操作无效。在 RISC-V 术语中称为 WARL（写入任意数值读取合法数值）。
[2] mtvec 仅支持在向量模式下对异常处理进行配置，基地址为 256 字节对齐。
[3] mcause 中反映的外部中断 ID 也包括 RISC-V 标准为核心内部中断源预留的 ID。

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。

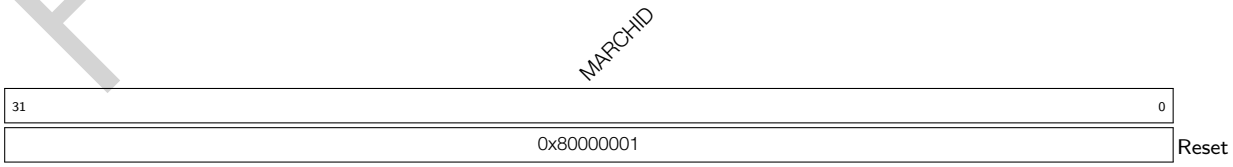
1.4.2 寄存器

Register 1.1. mvendorid (0xF11)



MVENDORID 供应商编号。（只读）

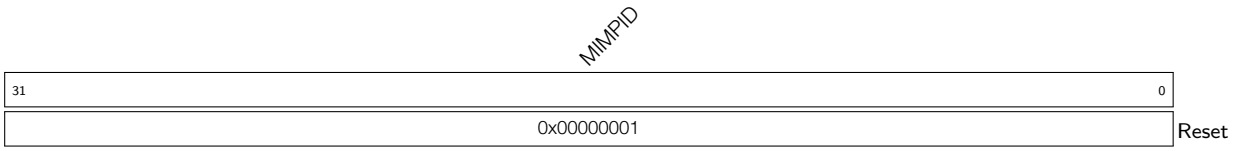
Register 1.2. marchid (0xF12)



MARCHID 架构编号。（只读）

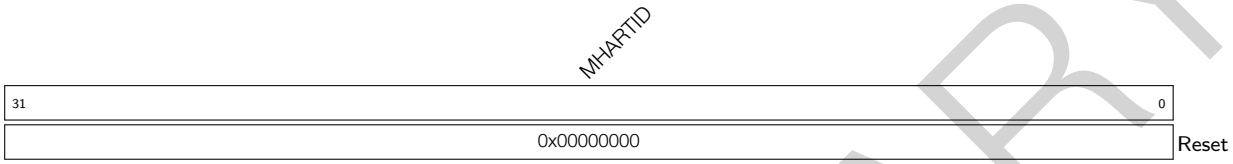
[4] 这些自定义机器模式 CSR 已经在 RISC-V 标准为用户保留的地址空间中实现。

Register 1.3. mimpid (0xF13)



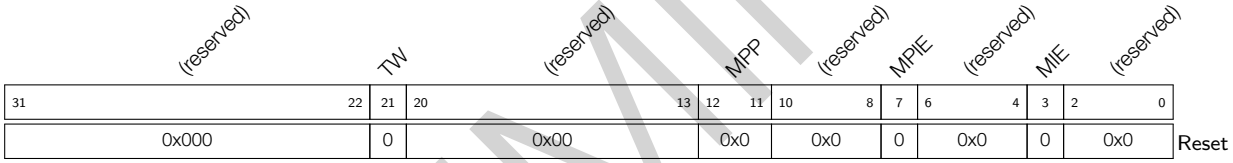
MIMPID 架构编号。(只读)

Register 1.4. mhartid (0xF14)



MHARTID 硬件线程编号。(只读)

Register 1.5. 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。

Register 1.6. 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）

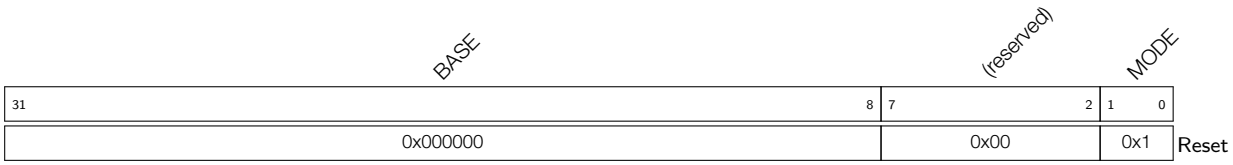
D 双精度浮点扩展 = 0。（只读）

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）

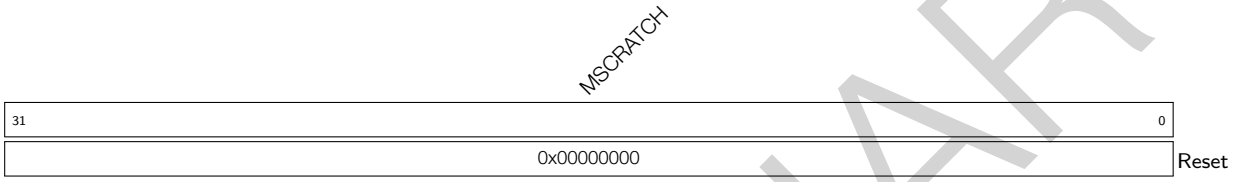
Register 1.7. mtvec (0x305)



MODE 仅支持向量模式 **0x1**。(只读)

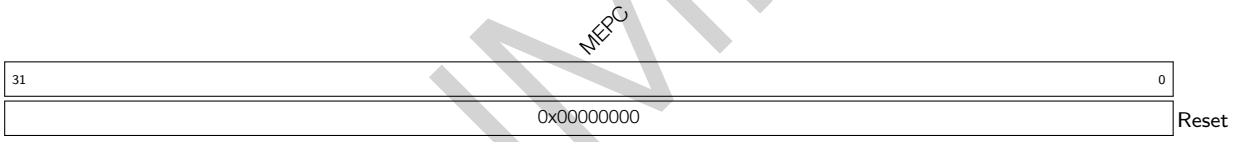
BASE 异常向量基址的高 24 位为 256 字节对齐。(读/写)

Register 1.8. mscratch (0x340)



MSCRATCH 用户自定义的机器暂存寄存器。(读/写)

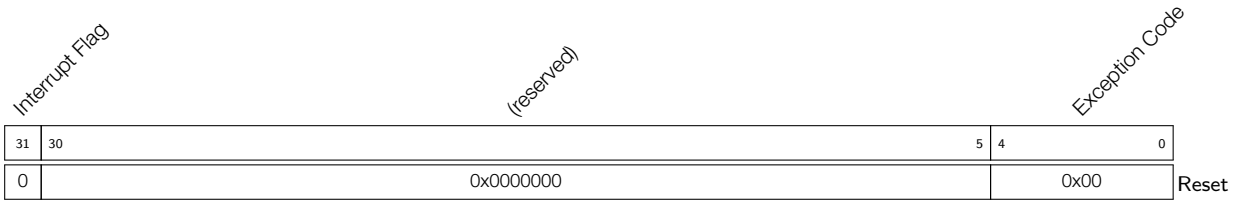
Register 1.9. mepc (0x341)



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。

Register 1.10. 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

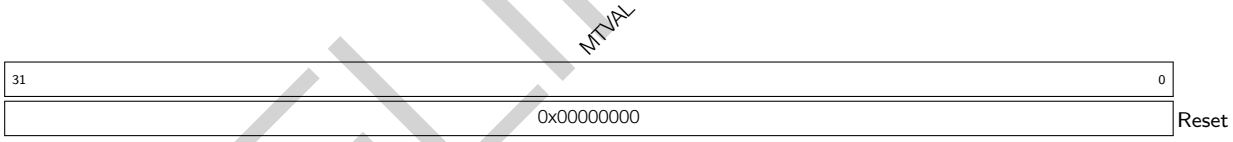
说明：异常 *ID 0x0*（指令地址非对齐）不存在，因为 *CPU* 在取指时始终屏蔽地址的最低位。

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 1.11. mtval (0x343)



MTVAL 机器模式异常值。(读/写)

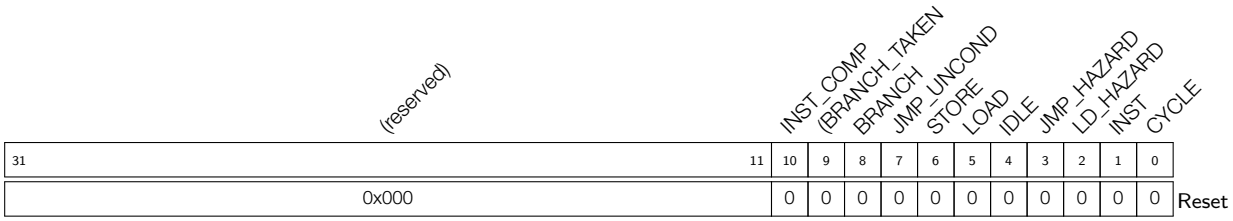
将自动更新为与异常有关的数据，该数据可能有助于处理该异常。

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。

Register 1.12. 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)

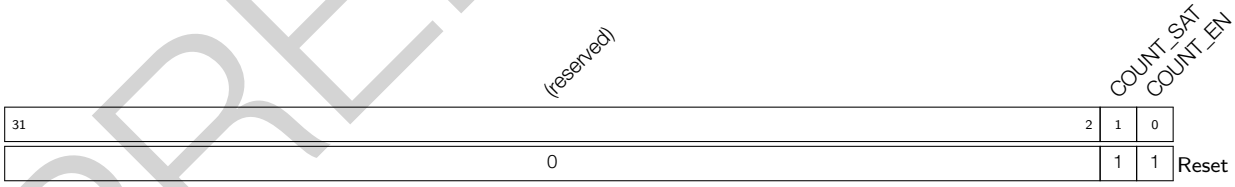
LD_HAZARD 计数存储器读操作冲突。(读/写)

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 1.13. mpcmr (0x7E1)



COUNT_SAT 计数器饱和控制。(读/写)

可能的值：

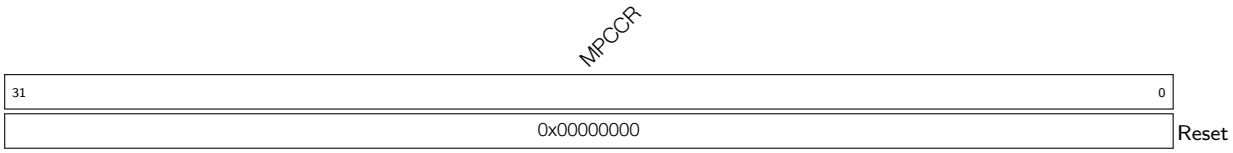
- 0：超出最大值上溢
- 1：超出最大值暂停

COUNT_EN 计数器使能控制。(读/写)

可能的值：

- 0：禁能
- 1：使能

Register 1.14. mpccr (0x7E2)



MPCCR 程序计数器计数的值。(读/写)

1.5 中断控制器

1.5.1 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

1.5.2 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”。

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 1-3 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 1-3. 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 1.5.3。

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）。

- 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）。

1.5.3 建议操作

1.5.3.1 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。

1.5.3.2 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。

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。

1.5.4 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。

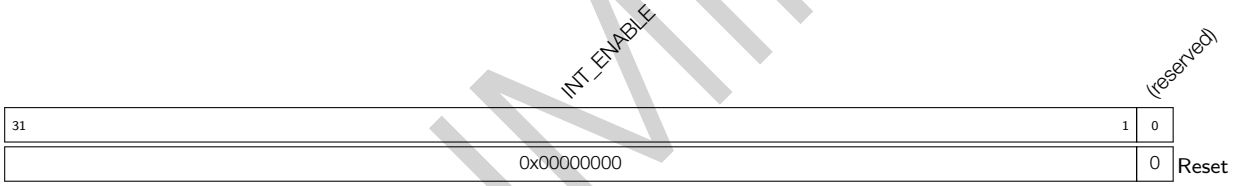
名称	描述	地址	访问
INT_ENABLE_REG	使能 CPU 处理中断	0x0104	读/写
INT_TYPE_REG	指定中断类型为电平/边沿类型	0x0108	读/写
INT_CLEAR_REG	写入清除“脉冲”类型中断	0x010C	读/写
INT_EIP_REG	外部中断的等待状态	0x0110	只读
INT_PRIORITY_1_REG	设置中断 ID=1 的优先级	0x0118	读/写
INT_PRIORITY_2_REG	设置中断 ID=2 的优先级	0x011C	读/写
INT_PRIORITY_3_REG	设置中断 ID=3 的优先级	0x0120	读/写
INT_PRIORITY_4_REG	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

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

1.5.5 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。

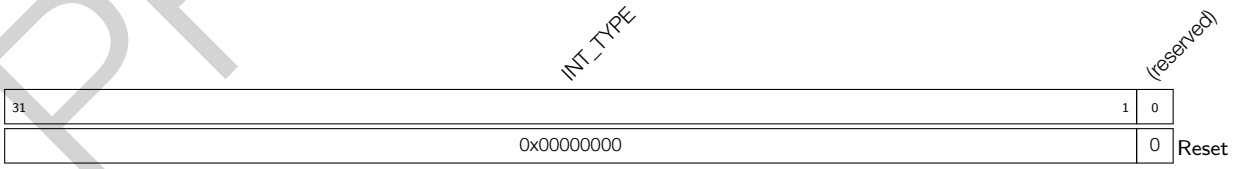
Register 1.15. INT_ENABLE_REG (0x0104)



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

- 0：禁能
- 1：使能

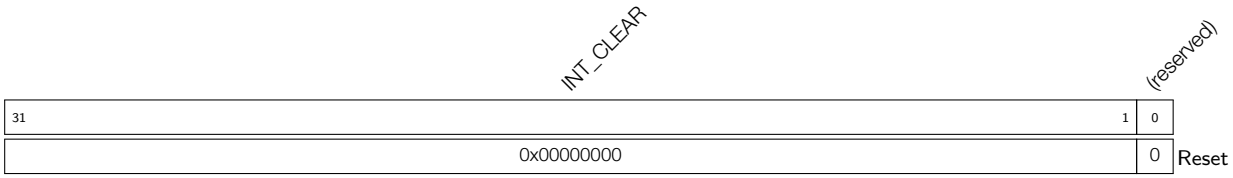
Register 1.16. INT_TYPE_REG (0x0108)



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）

Register 1.17. INT_CLEAR_REG (0x010C)



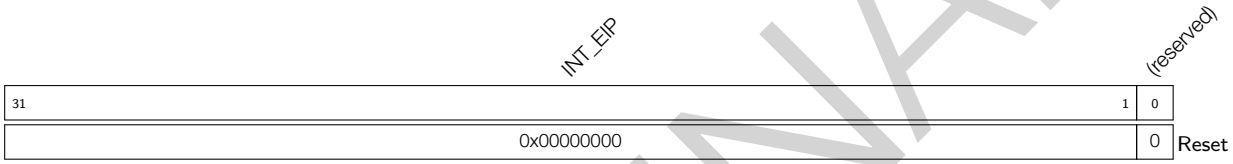
INT_CLEAR[*n*] 置位第 *n* 个位清除中断 *n* 的等待状态。(读/写)

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 1.18. INT_EIP_REG (0x0110)

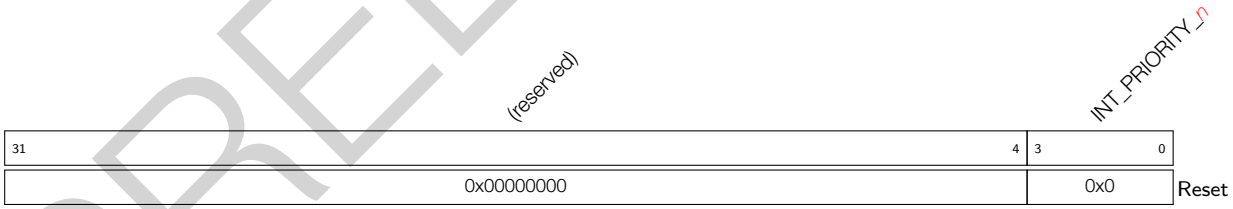


INT_EIP[*n*] 读取第 *n* 个位获得中断 *n* 的等待状态。(只读)

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

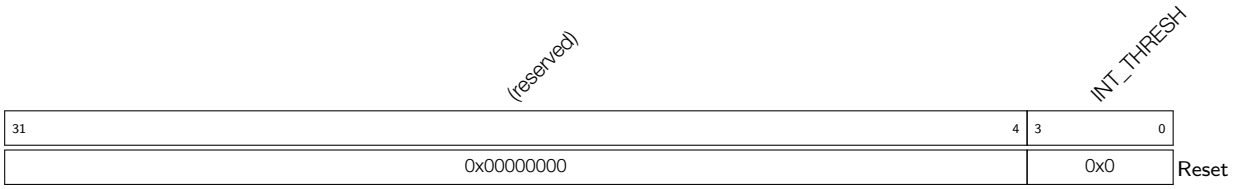
Register 1.19. INT_PRIORITY_*n*_REG (*n*: 1-31) (0x0114+4n*)**



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。

Register 1.20. 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。

1.6 调试

1.6.1 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 1-2 为外部调试系统架构图。

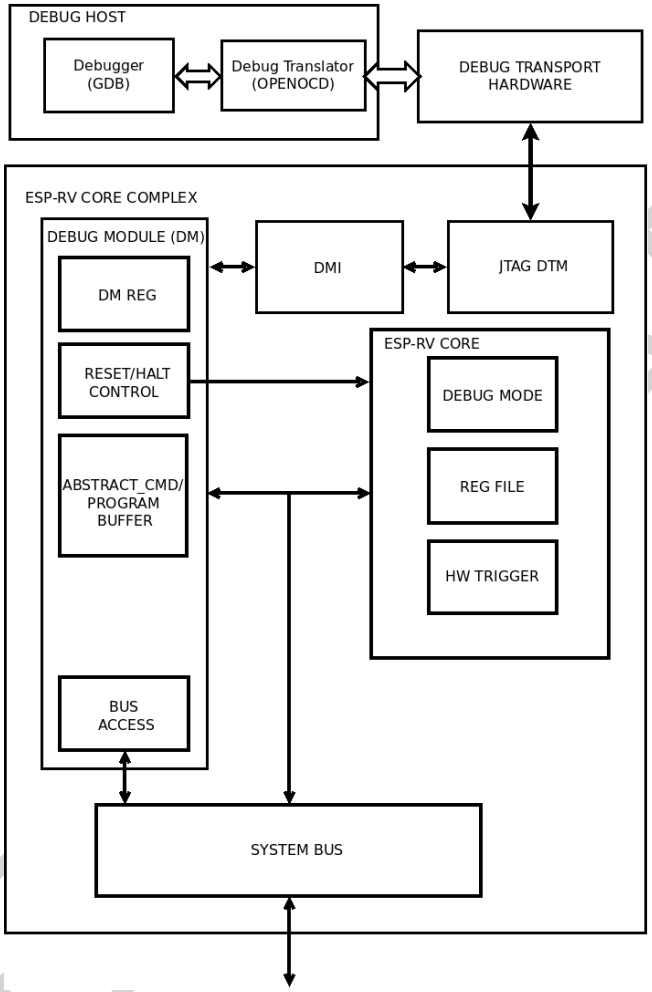


图 1-2. 调试系统架构

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。

1.6.2 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 1.7

1.6.3 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。

1.6.4 寄存器列表

下表列出了 ESP-RV 内核支持的调试 CSR。

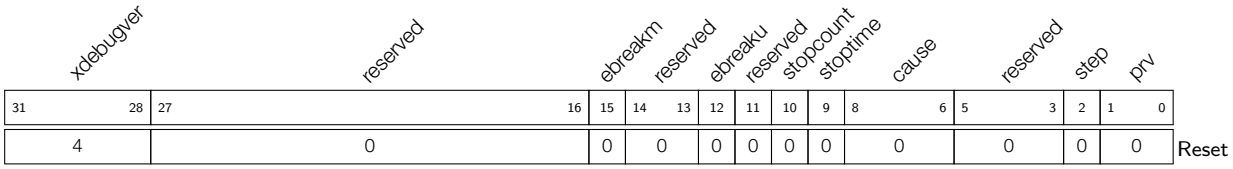
名称	描述	地址	访问
dcsr	调试控制和状态寄存器	0x7B0	读/写
dpc	调试 PC 寄存器	0x7B1	读/写
dscratch0	调试暂存寄存器 0	0x7B2	读/写
dscratch1	调试暂存寄存器 1	0x7B3	读/写

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。

1.6.5 寄存器

以下是 ESP-RV 内核支持的调试 CSR 的详细描述。

Register 1.21. 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)

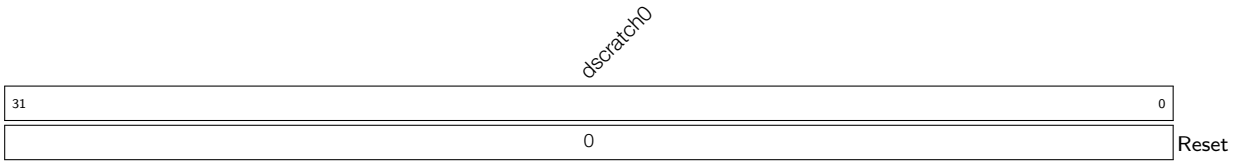
prv 保存 CPU 进入调试模式时候的特权级别。退出调试模式时，调试器可以更改此值以改变内核的特权级别。仅支持 **0x3**（机器模式）和 **0x0**（用户模式）。

* **注意**：与 RISC-V 调试规格版本 0.13 不同。

Register 1.22. 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)

Register 1.23. dscratch0 (0x7B2)



dscratch0 供调试模块内部使用。(读/写)

Register 1.24. dscratch1 (0x7B3)

dscratch1 供调试模块内部使用。(读/写)

1.7 硬件触发器

1.7.1 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

1.7.2 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 1-6）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 1-6. 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

....		
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。

1.7.3 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。

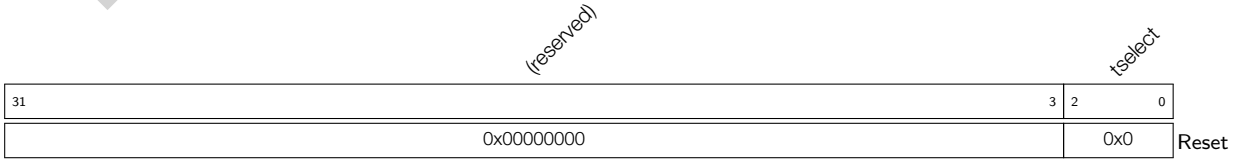
1.7.4 寄存器列表

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

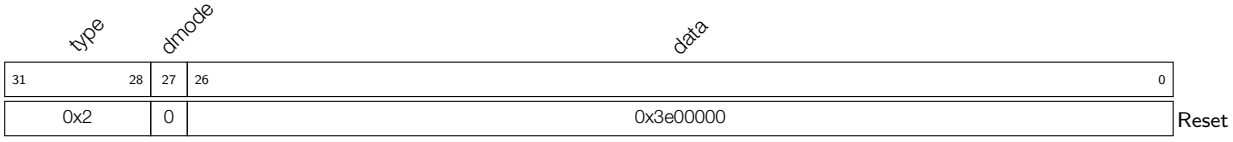
1.7.5 寄存器

Register 1.25. tselect (0x7A0)



tselect 触发器单元编号 (0-7)。(读/写)

Register 1.26. 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 *)

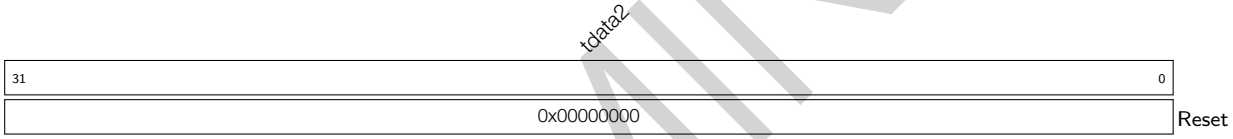
- 0：在调试模式和机器模式下都能写入 tdata1 和 tdata2
- 1：只有在调试模式下才能写入 tdata1 和 tdata2。其他模式下的写操作将被忽略。

* 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。

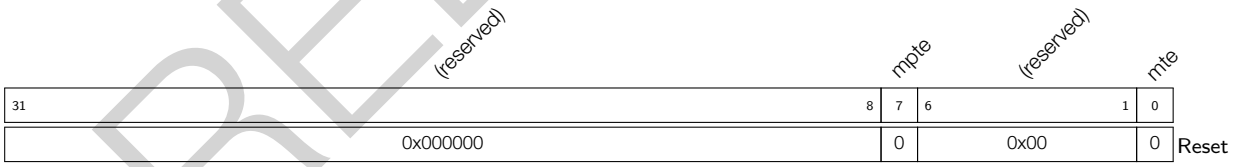
Register 1.27. tdata2 (0x7A2)



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 1.28. tcontrol (0x7A5)



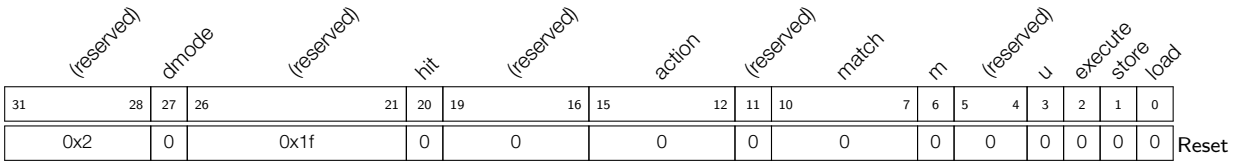
mpte 机器模式下前一个触发器使能域。(读/写)

- 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。

Register 1.29. 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)

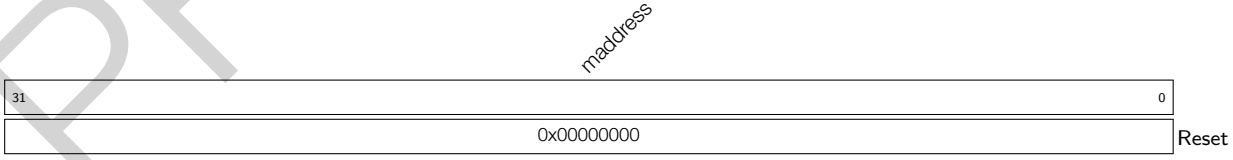
u 置位使选定的触发器在用户模式下操作。(读/写)

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 1.30. 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。

1.8 存储器保护

1.8.1 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”。

1.8.2 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。

1.8.3 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。

1.8.4 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
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

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
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

1.8.5 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。

2 通用 DMA 控制器 (GDMA)

2.1 概述

通用直接内存访问 (General Direct Memory Access, GDMA) 用于在外设与存储器之间以及存储器与存储器之间提供高速数据传输。软件可以在无需 CPU 干预的情况下通过 GDMA 快速搬移数据，从而降低了 CPU 的工作负载，提高了效率。

ESP32-C3 GDMA 共有 6 个独立的通道，其中包括 3 个发送通道和 3 个接收通道。这 6 个通道被支持 GDMA 功能的外设所共享，用户可以将通道分配给任何支持 DMA 功能的外设。这些外设包括：SPI2，UHCI0 (UART0/UART1)，I2S，AES，SHA 和 ADC。UART0 与 UART1 共用一个 UHCI0 接口。

GDMA 支持通道间固定优先级及轮询仲裁以管理外设不同的带宽需求。

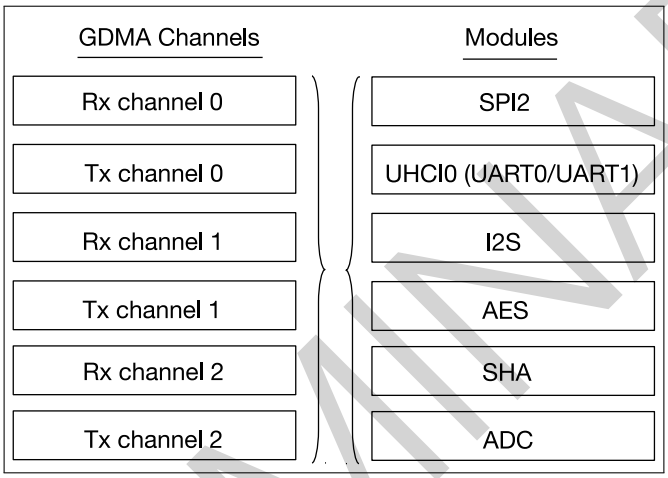


图 2-1. 具有 GDMA 功能的模块和 GDMA 通道

2.2 特性

GDMA 控制器具有以下几个特点：

- AHB 总线架构
- 数据传输以字节为单位，传输数据量可软件编程
- 支持链表
- 访问内部 RAM 时，支持 INCR burst 传输
- GDMA 能够访问的内部 RAM 最大地址空间为 384 KB
- 包含 3 个 TX、3 个 RX 通道
- 任一通道支持可配置的外设选择
- 通道间固定优先级及轮询仲裁

2.3 架构

ESP32-C3 中所有需要进行高速数据传输的模块都具有 GDMA 功能。GDMA 控制器与 CPU 的数据总线使用相同的地址空间访问内部 RAM。图 2-2 为 GDMA 引擎基本架构图。

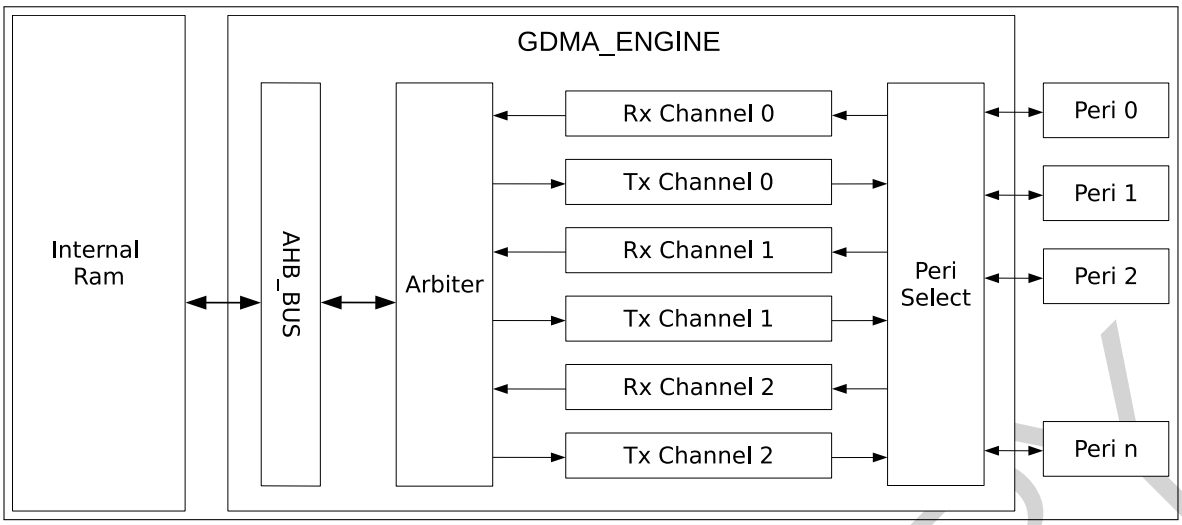


图 2-2. GDMA 引擎的架构

GDMA 引擎共有 6 个独立的通道，其中包括 3 个发送通道和 3 个接收通道。每个通道可选择与不同的外设相连，从而实现通道资源被外设共享。GDMA 引擎通过 AHB_BUS 将数据存入内部 RAM 或者将数据从内部 RAM 取出。内部 RAM 的具体使用范围详见章节 3 *系统和存储器*。软件可以通过挂载链表的方式来使用 GDMA 引擎。链表本身须存储在片内 RAM 中，包括 outlink*n* 与 inlink*n*，本文以*n* 来表示通道号，*n*为 0 ~ 2。GDMA 从片内 RAM 中取得链表，然后根据 outlink*n* 中的内容将相应 RAM 中的数据发送出去，也可根据 inlink*n* 中的内容将接收的数据存入指定 RAM 地址空间。

2.4 功能描述

2.4.1 链表

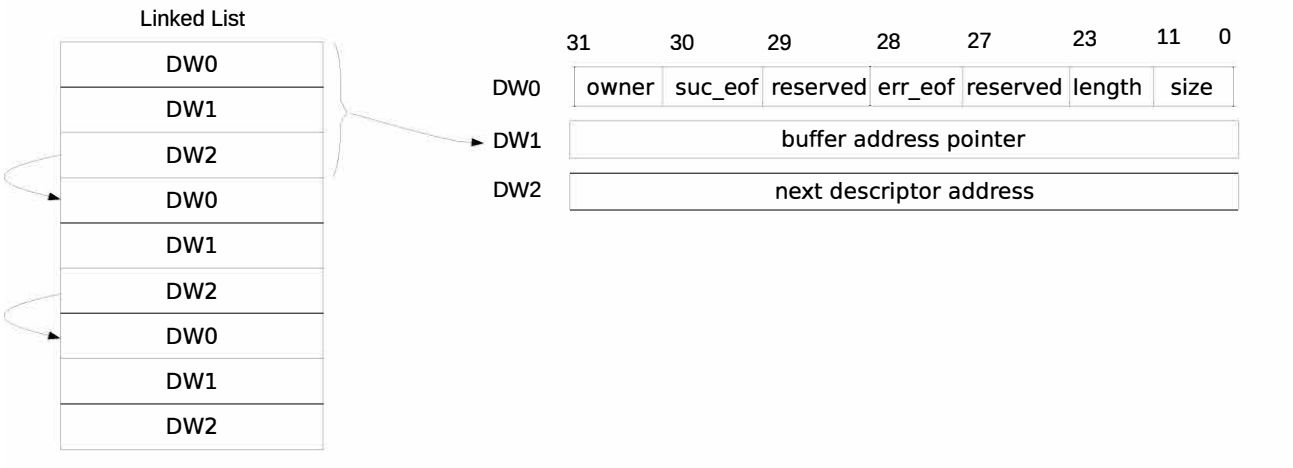


图 2-3. 链表结构图

图 2-3 所示为链表的结构图。发送链表与接收链表结构相同。每个链表由一个或者若干个描述符构成，一个描述符由 3 个字组成。链表应存放在内部 RAM 中，供 GDMA 引擎使用。描述符每一字段的意义如下：

- owner (DW0) [31]：表示当前描述符对应的 buffer 允许的操作者。
  1'b0：允许的操作者为 CPU；

1'b1：允许的操作者为 GDMA 控制器。
在 GDMA 使用完该描述符对应的 buffer 后，对于接收描述符，硬件默认会自动将该 bit 清零；对于发送描述符，需要将 GDMA_OUT_AUTO_WRBACK_CH*n* 置 1，硬件才会自动将该 bit 清零。软件也可通过置位 GDMA_OUT_LOOP_TEST_CH*n* 或 GDMA_IN_LOOP_TEST_CH*n* 来关闭硬件自动清零的功能。软件在挂载链表时需要将该 bit 置 1。

注意：本文以 GDMA_OUT 开头的寄存器对应 TX 通道寄存器，以 GDMA_IN 开头的寄存器对应 RX 通道寄存器。

- suc_eof (DW0) [30]：表示结束标志。
  1'b0：当前描述符不是链表中的最后一个描述符；
  1'b1：当前描述符为数据包的最后一个描述符。
  对于接收描述符，需要软件将该 bit 写 0，硬件会在收完 1 个包后将该 bit 置 1。对于发送描述符，需要软件在包的最后一个描述符中的该 bit 置 1。
- Reserved (DW0) [29]：保留。此位为无关项。
- err_eof (DW0) [28]：表示接收结束错误标志。
  该 bit 只用于 UHCI0 利用 GDMA 接收数据。对于接收描述符，硬件在收完 1 个包并检测到接收数据错误会将该 bit 置 1。
- Reserved (DW0) [27:24]：保留。
- length (DW0) [23:12]：表示当前描述符对应的 buffer 中的有效字节数。对于发送描述符，该段由软件填写，表示从 buffer 中读取数据时需要读取的字节数；对于接收描述符，该段由硬件使用完该 buffer 后或者接收到最后一个数据时自动填写，表示 buffer 中存储的有效字节数。
- size (DW0) [11:0]：表示当前描述符对应的 buffer 容量的字节数。
- buffer address pointer (DW1)：buffer 的地址。
- next descriptor address (DW2)：下一个描述符的地址。如果当前描述符为链表中最后一个描述符时 (即 suc_eof = 1)，该值可以为 0。该地址必须指向片内 RAM 的地址空间。

用 GDMA 接收数据时，如果接收数据的长度小于当前描述符指定的 buffer 长度，那么下一个描述符对应的接收数据不会占用该 buffer 的剩余空间。

2.4.2 外设到存储及存储到外设的数据传输

GDMA 支持存储到外设及外设到存储的数据传输，分别对应 TX 及 RX 功能。TX 通道通过 outlink*n* 实现将指定存储区域中的数据搬运到外设的发送端；RX 通道通过 inlink*n* 实现将外设接收到的数据搬运到指定的存储区域。

每个 RX/TX 通道均可以被配置连接到任意一个支持 GDMA 功能的外设，表2-1 所示为配置寄存器与其对应外设的关系。当其中一个通道已经与某一个外设连接时，其他通道将不能配置为与该外设连接。

表 2-1. 配置寄存器与外设选择关系表

GDMA_IN_PERI_SEL_CH*n* GDMA_OUT_PERI_SEL_CH*n*	外设
0	SPI2
1	Reserved
2	UHCI0
3	I2S

4	Reserved
5	Reserved
6	AES
7	SHA
8	ADC

2.4.3 存储到存储数据传输

GDMA 支持存储到存储的数据传输。置位 GDMA_MEM_TRANS_EN_CH*n*，TX 通道*n*的输出将与 RX 通道*n*的输入相连，从而使能存储到存储的数据传输功能。需要注意的是，一个 TX 通道只与其编号对应的 RX 通道相连而实现存储到存储的数据传输。

2.4.4 启动 DMA

软件通过挂载链表的方式来使用 GDMA。对于接收数据，软件挂载好接收链表并准备好接收数据，配置 GDMA_INLINK_ADDR_C 字段指向第一个接收链表描述符。置位 GDMA_INLINK_START_CH*n* 位启动 GDMA。对于发送数据，软件挂载好发送链表并准备好发送数据，配置 GDMA_OUTLINK_ADDR_CH*n* 字段指向第一个发送链表描述符。置位 GDMA_OUTLINK_START_CH*n* 位启动 GDMA。GDMA_INLINK_START_CH*n* 与 GDMA_OUTLINK_START_CH*n* 位由硬件自动清零。

有时您可能想要在 DMA 数据传输已经开始后追加更多描述符。要挂载更多描述符，原本看似只需清空已挂载链表最后一个描述符的 EOF 位，并将该描述符的 next descriptor address (DW2) 字段配置为新链表第一个描述符。但如果 DMA 数据传输已经或马上就要结束，这个方法便行不通了。GDMA 引擎有专门的逻辑来确保数据传输继续或重启：如果数据传输仍在进行，GDMA 引擎会确保顾及到新追加的描述符；如果数据传输已经结束，GDMA 引擎会重启数据传输，传输新追加的描述符。这个逻辑由 Restart 功能实现。

软件使用 Restart 功能时，需要重写已挂载链表的最后一个描述符，使其第三个 word 中的内容（即 DW2）指向新链表的首地址；然后置位GDMA_INLINK_RESTART_CH*n* 或者 GDMA_OUTLINK_RESTART_CH*n*（这两个位由硬件自动清零），如图 2-4所示，硬件会在读取已挂载链表的最后一个描述符时，获取新挂载链表的地址，从而继续处理新挂载的链表。

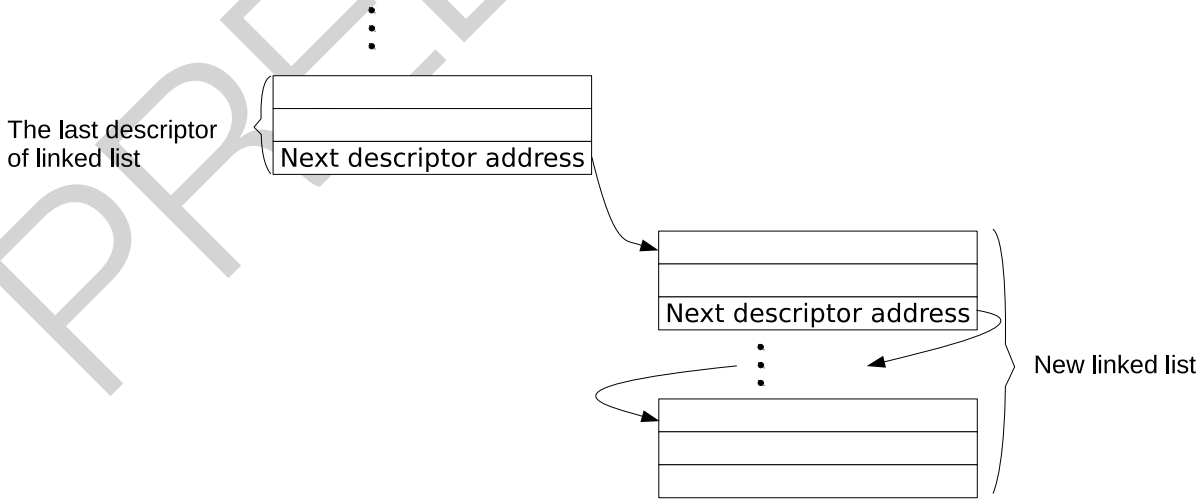


图 2-4. 链表关系图

2.4.5 读链表

软件在配置并启动 GDMA 后，GDMA 会从内部 RAM 读取链表。GDMA 会检查读入的链表描述符是否正确。只有当链表描述符通过检查时，GDMA 对应的通道才会开始搬运数据。当链表描述符没有通过检查，硬件将触发描述符错误中断（GDMA_IN_DSCR_ERR_CH*n*_INT 或者 GDMA_OUT_DSCR_ERR_CH*n*_INT），同时该通道将会处于阻塞状态，停止工作。

描述符检查项包括：

- GDMA_IN_CHECK_OWNER_CH*n* 或者 GDMA_OUT_CHECK_OWNER_CH*n* 置 1 时，检查描述符的 owner 比特。如果该比特为 0，表示当前操作者为 CPU，则不通过检查。GDMA_IN_CHECK_OWNER_CH*n* 或者 GDMA_OUT_CHECK_OWNER_CH*n* 置 0 时，不检查描述符的 owner 比特；
- 检查描述符中第二个 word 指示的地址是否在 0x3FC80000 ~ 0x3FCDFFFF 时（请参见本节 2.4.7）。如果在该范围内，则通过检查。否则，不通过检查。

软件在检查到通道描述符错误中断后，需要复位对应的通道，并置位 GDMA_OUTLINK_START_CH*n* 或者 GDMA_INLINK_START_CH*n*位启动 GDMA。

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

2.4.6 数据传输结束标志

GDMA 通过 EOF 来指示数据传输结束。

发送数据时，置位 GDMA_OUT_TOTAL_EOF_CH*n*_INT_ENA 位使能 GDMA_OUT_TOTAL_EOF_CH*n*_INT 中断，当带有 EOF 标志的描述符对应 buffer 的数据传输完成后，GDMA 会产生该中断。

接收数据时，置位 GDMA_IN_SUC_EOF_CH*n*_INT_ENA 位使能 GDMA_IN_SUC_EOF_CH*n*_INT 中断，表示数据接收完成。GDMA 还支持中断 GDMA_IN_ERR_CH*n*_EOF_INT，置位 GDMA_IN_ERR_EOF_CH*n*_INT_ENA 使能该中断，表示接收完成但接收数据有错误。需要注意的是，只有当通道连接的外设为 UHCI0 时，才支持该中断。

软件在检测到 GDMA_OUT_TOTAL_EOF_CH*n*_INT 或 GDMA_IN_SUC_EOF_CH*n*_INT 中断时，可以记录 GDMA_OUT_EOF_DE
或 GDMA_IN_SUC_EOF_DES_ADDR_CH*n* 字段的值，即最后一个描述符的地址。这样，软件可以知道哪些描述符已经被使用并根据需要回收描述符。

注意：本章中提到发送链表描述符的 EOF 为 suc_eof，接收链表描述符的 EOF 可以为 suc_eof 和 err_eof。

2.4.7 访问片内 RAM

GDMA 任意 RX/TX 通道均可以访问片内 RAM，其可访问的片内地址空间为 0x3FC80000 ~ 0x3FCDFFFF。为加速数据传输速率，支持突发传输模式。置位 GDMA_IN_DATA_BURST_EN_CH*n* 使能 RX 通道突发传输模式；置位 GDMA_OUT_DATA_BURST_EN_CH*n* 使能 TX 通道突发传输模式。默认情况下，突发传输没有使能。

表 2-2. 链表描述符参数对齐要求

inlink/outlink	burst mode	size	length	buffer address pointer
inlink	0	无对齐要求	无对齐要求	无对齐要求
	1	字对齐	无对齐要求	字对齐

outlink	0	无对齐要求	无对齐要求	无对齐要求
	1	无对齐要求	无对齐要求	无对齐要求

如表2-2所示为访问片内时，链表描述符参数配置对齐要求。

当突发模式没有被使能时，无论是发送链表描述符还是接收链表描述符，其参数 size，length 及 buffer address pointer 均没有字对齐的要求。也就是说，对于一个描述符，在可访问的片内地址空间，GDMA 可以从任意起始地址，读出配置长度的数据，长度取值范围为 1 ~ 4095；或者，将接收到的数据长度（1 ~ 4095）写入任意起始地址开始的连续地址。

当突发模式使能时，对于发送链表描述符，参数 size，length 及 buffer address pointer 均没有字对齐的要求。而对于接收链表描述符，除了参数 length，参数 size 和 buffer address pointer 均需要保持字对齐。

2.4.8 仲裁

为了确保及时响应高速低延迟的外设请求，比如 SPI 等，GDMA 在通道仲裁机制中引入固定优先级，即每个通道的优先级可配置。GDMA 支持 10（0 ~ 9）个等级的优先级。其数值越大，对应的优先级越高，请求响应越及时。当若干个通道配置为相同的优先级时，这几个通道间对请求的响应将采用轮询仲裁机制。

需要注意的是，所有外设总的吞吐率之和不能超过 GDMA 能支持的最大有效带宽，从而保证低优先级的外设请求也能获得响应。

2.4.9 带宽

GDMA 作为 AHB 主机，通过发起 AHB 传输来访问存储空间。在不考虑其他 AHB 主机（WIFI）与 GDMA 竞争的情况下，GDMA 能支持的总带宽计算公式如下：
每个通道均使能突发传输模式：8/5*fhclk MB/s；
每个通道均不使能突发传输模式：4/3*fhclk MB/s；

其中 fhclk 为 AHB 时钟频率，固定为 80 MHz。根据上述计算公式，GDMA 的总带宽如表2-3所示：

表 2-3. GDMA 支持的总带宽

fpclk 突发传输模式	各通道均不使能	各通道均使能
80 MHz	106.6 MB/s	128 MB/s

需要注意的是，由于 GDMA 通过描述符来控制数据的传输，总的数据传输量包含读描述符自身的字节数。一个描述符对应的传输效率为：length/(length + 12)

其中 length 为描述符中参数，12 对应描述符的 3 个字。因此，在多链表描述符的应用中，应尽可能的提高单个描述符的 length 来提高传输效率，其最大值为 99.7%。

软件在分配带宽给外设时，可以通过公式：T*(length + 12)/length 来预估该外设占用的 GDMA 带宽。其中 T 为外设的吞吐率。

2.5 GDMA 中断

- GDMA_OUT_TOTAL_EOF_CH*n*_INT：对于发送通道*n*，当一个链表（可包含多个链表描述符）对应的所有数据都已发送完成时触发此中断。

- GDMA_IN_DSCR_EMPTY_CH*n*_INT：对于接收通道*n*，当接收链表描述符指向的 buffer 大小小于待接收数据长度时触发此中断。
- GDMA_OUT_DSCR_ERR_CH*n*_INT：对于发送通道*n*，当发送链表描述符里有错误时触发此中断。
- GDMA_IN_DSCR_ERR_CH*n*_INT：对于接收通道*n*，当接收链表描述符里有错误时触发此中断。
- GDMA_OUT_EOF_CH*n*_INT：对于发送通道*n*，当发送描述符的 EOF 位为 1，并且该描述符对应的数据发送完成时触发此中断。当GDMA_OUT_EOF_MODE_CH*n*为 0 时，该描述符对应的最后一个数据进入到 GDMA TX 通道时，该中断触发；当GDMA_OUT_EOF_MODE_CH*n*为 1 时，该描述符对应的最后一个数据从 GDMA TX 通道取出时，该中断触发。
- GDMA_OUT_DONE_CH*n*_INT：对于发送通道*n*，当一个发送链表描述符对应的数据发送完成时触发此中断。
- GDMA_IN_ERR_EOF_CH*n*_INT：对于接收通道*n*，当接收的一个数据包中有错误发生时触发此中断。(该中断只用于外设选择 UHCI0（UART0/UART1）时)。
- GDMA_IN_SUC_EOF_CH*n*_INT：对于接收通道*n*，当一个数据包接收完成时触发此中断。
- GDMA_IN_DONE_CH*n*_INT：对于接收通道*n*，当一个接收链表描述符对应的数据接收完成时触发此中断。

2.6 编程流程

2.6.1 GDMA TX 通道配置流程

利用 GDMA 发送数据时，GDMA TX 通道的软件配置流程如下：

1. 对寄存器 GDMA_OUT_RST_CH*n* 置 1 然后置 0，复位 GDMA TX 通道状态机和 FIFO 指针；
2. 挂载好发送链表，配置寄存器 GDMA_OUTLINK_ADDR_CH*n* 指向第一个发送链表描述符；
3. 配置 GDMA_PERI_OUT_SEL_CH*n* 为对应的外设号，见表2-1；
4. 置位 GDMA_OUTLINK_START_CH*n* 启动 GDMA TX 通道发送数据；
5. 配置对应的外设（SPI2、UHCI0（UART0/UART1）、I2S、AES、SHA、ADC），并启动该外设，具体配置请参考对应的外设章节；
6. 等待 GDMA_OUT_EOF_CH*n*_INT 中断，即数据传输完成。

2.6.2 GDMA RX 通道配置流程

利用 GDMA 接收数据时，GDMA RX 通道的软件配置流程如下：

1. 对寄存器 GDMA_IN_RST_CH*n* 置 1 然后置 0，复位 GDMA RX 通道状态机和 FIFO 指针；
2. 挂载好接收链表，配置寄存器 GDMA_INLINK_ADDR_CH*n* 指向第一个接收链表描述符；
3. 配置 GDMA_PERI_IN_SEL_CH*n* 为对应的外设号，见表2-1；
4. 置位 GDMA_INLINK_START_CH*n* 启动 GDMA RX 通道发送数据；
5. 配置对应的外设（SPI2、UHCI0（UART0/UART1）、I2S、AES、ADC），并启动该外设，具体配置请参考对应的外设章节；
6. 等待 GDMA_IN_SUC_EOF_CH*n*_INT 中断，即一个数据包接收完成。

2.6.3 GDMA 存储器到存储器配置流程

利用 GDMA 从存储到存储搬运数据时配置流程如下：

1. 对寄存器 GDMA_OUT_RST_CH*n* 置 1 然后置 0，复位 GDMA TX 通道状态机和 FIFO 指针；
2. 对寄存器 GDMA_IN_RST_CH*n* 置 1 然后置 0，复位 GDMA RX 通道状态机和 FIFO 指针；
3. 挂载好发送链表，配置寄存器 GDMA_OUTLINK_ADDR_CH*n* 指向第一个发送链表描述符；
4. 挂载好接收链表，配置寄存器 GDMA_INLINK_ADDR_CH*n* 指向第一个接收链表描述符；
5. 置位 GDMA_MEM_TRANS_EN_CH*n* 使能 memory-to-memory 传输功能；
6. 置位 GDMA_OUTLINK_START_CH*n* 启动 GDMA TX 通道发送数据；
7. 置位 GDMA_INLINK_START_CH*n* 启动 GDMA RX 通道发送数据；
8. 等待 GDMA_IN_SUC_EOF_CH*n*_INT 中断，即一次数据搬运完成。

2.7 寄存器列表

本小节的所有地址均为相对于通用 DMA 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
中段寄存器			
GDMA_INT_RAW_CH0_REG	接收通道 0 的原始中断状态	0x0000	R/WTC/SS
GDMA_INT_ST_CH0_REG	接收通道 0 的屏蔽中断	0x0004	RO
GDMA_INT_ENA_CH0_REG	接收通道 0 的中断使能位	0x0008	R/W
GDMA_INT_CLR_CH0_REG	接收通道 0 的中断清除位	0x000C	WT
GDMA_INT_RAW_CH1_REG	接收通道 1 的原始中断状态	0x0010	R/WTC/SS
GDMA_INT_ST_CH1_REG	接收通道 1 的屏蔽中断	0x0014	RO
GDMA_INT_ENA_CH1_REG	接收通道 1 的中断使能位	0x0018	R/W
GDMA_INT_CLR_CH1_REG	接收通道 1 的中断清除位	0x001C	WT
GDMA_INT_RAW_CH2_REG	接收通道 2 的原始中断状态	0x0020	R/WTC/SS
GDMA_INT_ST_CH2_REG	接收通道 2 的屏蔽中断	0x0024	RO
GDMA_INT_ENA_CH2_REG	接收通道 2 的中断使能位	0x0028	R/W
GDMA_INT_CLR_CH2_REG	接收通道 2 的中断清除位	0x002C	WT
配置寄存器			
GDMA_MISC_CONF_REG	MISC 寄存器	0x0044	R/W
版本寄存器			
GDMA_DATE_REG	版本控制寄存器	0x0048	R/W
配置寄存器			
GDMA_IN_CONF0_CH0_REG	接收通道 0 的配置寄存器 0	0x0070	R/W
GDMA_IN_CONF1_CH0_REG	接收通道 0 的配置寄存器 1	0x0074	R/W
GDMA_IN_POP_CH0_REG	接收通道 0 的数据弹出控制寄存器	0x007C	varies
GDMA_IN_LINK_CH0_REG	接收通道 0 的链表配置和控制寄存器	0x0080	varies
GDMA_OUT_CONF0_CH0_REG	发送通道 0 的配置寄存器 0	0x00D0	R/W
GDMA_OUT_CONF1_CH0_REG	发送通道 0 的配置寄存器 1	0x00D4	R/W
GDMA_OUT_PUSH_CH0_REG	发送通道 0 的数据推送控制寄存器	0x00DC	varies
GDMA_OUT_LINK_CH0_REG	发送通道 0 的链表配置和控制寄存器	0x00E0	varies
GDMA_IN_CONF0_CH1_REG	接收通道 1 的配置寄存器 0	0x0130	R/W
GDMA_IN_CONF1_CH1_REG	接收通道 1 的配置寄存器 1	0x0134	R/W
GDMA_IN_POP_CH1_REG	接收通道 1 的数据弹出控制寄存器	0x013C	varies
GDMA_IN_LINK_CH1_REG	接收通道 1 的链表配置和控制寄存器	0x0140	varies
GDMA_OUT_CONF0_CH1_REG	发送通道 1 的配置寄存器 0	0x0190	R/W
GDMA_OUT_CONF1_CH1_REG	发送通道 1 的配置寄存器 1	0x0194	R/W
GDMA_OUT_PUSH_CH1_REG	发送通道 1 的数据推送控制寄存器	0x019C	varies
GDMA_OUT_LINK_CH1_REG	发送通道 1 的链表配置和控制寄存器	0x01A0	varies
GDMA_IN_CONF0_CH2_REG	接收通道 2 的配置寄存器 0	0x01F0	R/W
GDMA_IN_CONF1_CH2_REG	接收通道 2 的配置寄存器 1	0x01F4	R/W

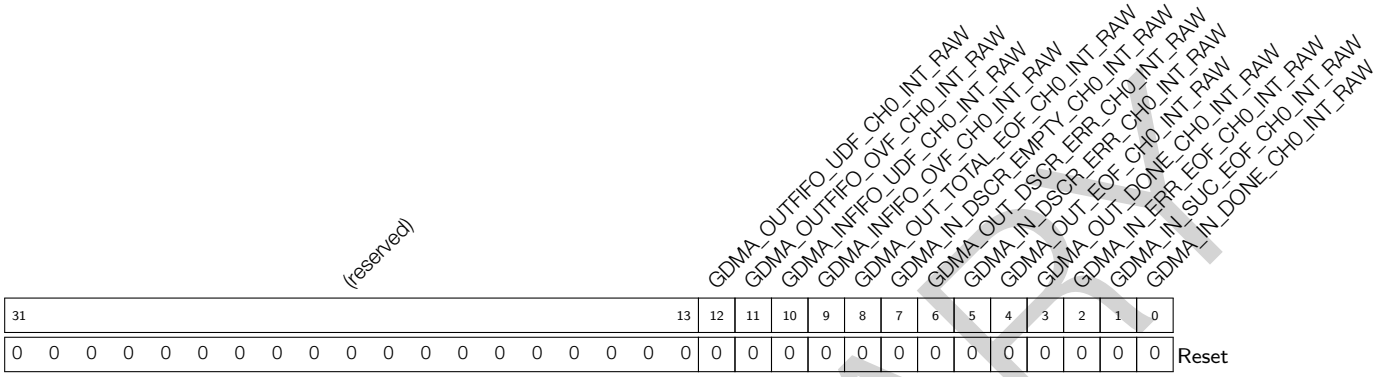
名称	描述	地址	访问
GDMA_IN_POP_CH2_REG	接收通道 2 的数据弹出控制寄存器	0x01FC	varies
GDMA_IN_LINK_CH2_REG	接收通道 2 的链表配置和控制寄存器	0x0200	varies
GDMA_OUT_CONF0_CH2_REG	发送通道 2 的配置寄存器 0	0x0250	R/W
GDMA_OUT_CONF1_CH2_REG	发送通道 2 的配置寄存器 1	0x0254	R/W
GDMA_OUT_PUSH_CH2_REG	发送通道 2 的数据推送控制寄存器	0x025C	varies
GDMA_OUT_LINK_CH2_REG	发送通道 2 的链表配置和控制寄存器	0x0260	varies
状态寄存器			
GDMA_INFIFO_STATUS_CH0_REG	接收通道 0 的 RX FIFO 状态	0x0078	RO
GDMA_IN_STATE_CH0_REG	接收通道 0 的接收状态	0x0084	RO
GDMA_IN_SUC_EOF_DES_ADDR_CH0_REG	接收通道 0 传输完成时的接收链表描述符地址	0x0088	RO
GDMA_IN_ERR_EOF_DES_ADDR_CH0_REG	接收通道 0 发生错误时的接收链表描述符地址	0x008C	RO
GDMA_IN_DSCR_CH0_REG	接收通道 0 当前的接收链表描述符地址	0x0090	RO
GDMA_IN_DSCR_BF0_CH0_REG	接收通道 0 最后一个接收链表描述符地址	0x0094	RO
GDMA_IN_DSCR_BF1_CH0_REG	接收通道 0 倒数第二个接收链表描述符地址	0x0098	RO
GDMA_OUTFIFO_STATUS_CH0_REG	发送通道 0 的 TX FIFO 状态	0x00D8	RO
GDMA_OUT_STATE_CH0_REG	发送通道 0 的发送状态	0x00E4	RO
GDMA_OUT_EOF_DES_ADDR_CH0_REG	发送通道 0 传输完成时的发送链表描述符地址	0x00E8	RO
GDMA_OUT_EOF_BFR_DES_ADDR_CH0_REG	发送通道 0 传输完成时的最后一个发送链表描述符地址	0x00EC	RO
GDMA_OUT_DSCR_CH0_REG	发送通道 0 当前的发送链表描述符地址	0x00F0	RO
GDMA_OUT_DSCR_BF0_CH0_REG	发送通道 0 最后一个发送链表描述符地址	0x00F4	RO
GDMA_OUT_DSCR_BF1_CH0_REG	发送通道 0 倒数第二个发送链表描述符地址	0x00F8	RO
GDMA_INFIFO_STATUS_CH1_REG	接收通道 1 的 RX FIFO 状态	0x0138	RO
GDMA_IN_STATE_CH1_REG	接收通道 1 的接收状态	0x0144	RO
GDMA_IN_SUC_EOF_DES_ADDR_CH1_REG	接收通道 1 传输完成时的接收链表描述符地址	0x0148	RO
GDMA_IN_ERR_EOF_DES_ADDR_CH1_REG	接收通道 1 发生错误时的接收链表描述符地址	0x014C	RO
GDMA_IN_DSCR_CH1_REG	接收通道 1 当前的接收链表描述符地址	0x0150	RO
GDMA_IN_DSCR_BF0_CH1_REG	接收通道 1 最后一个接收链表描述符地址	0x0154	RO
GDMA_IN_DSCR_BF1_CH1_REG	接收通道 1 倒数第二个接收链表描述符地址	0x0158	RO
GDMA_OUTFIFO_STATUS_CH1_REG	发送通道 1 的 TX FIFO 状态	0x0198	RO
GDMA_OUT_STATE_CH1_REG	发送通道 1 的发送状态	0x01A4	RO
GDMA_OUT_EOF_DES_ADDR_CH1_REG	发送通道 1 传输完成时的发送链表描述符地址	0x01A8	RO
GDMA_OUT_EOF_BFR_DES_ADDR_CH1_REG	发送通道 1 传输完成时的最后一个发送链表描述符地址	0x01AC	RO
GDMA_OUT_DSCR_CH1_REG	发送通道 1 当前的发送链表描述符地址	0x01B0	RO
GDMA_OUT_DSCR_BF0_CH1_REG	发送通道 1 最后一个发送链表描述符地址	0x01B4	RO
GDMA_OUT_DSCR_BF1_CH1_REG	发送通道 1 倒数第二个发送链表描述符地址	0x01B8	RO

名称	描述	地址	访问
GDMA_INFIFO_STATUS_CH2_REG	接收通道 2 的 RX FIFO 状态	0x01F8	RO
GDMA_IN_STATE_CH2_REG	接收通道 2 的接收状态	0x0204	RO
GDMA_IN_SUC_EOF_DES_ADDR_CH2_REG	接收通道 2 传输完成时的接收链表描述符地址	0x0208	RO
GDMA_IN_ERR_EOF_DES_ADDR_CH2_REG	接收通道 2 发生错误时的接收链表描述符地址	0x020C	RO
GDMA_IN_DSCR_CH2_REG	接收通道 2 当前的接收链表描述符地址	0x0210	RO
GDMA_IN_DSCR_BF0_CH2_REG	接收通道 2 最后一个接收链表描述符地址	0x0214	RO
GDMA_IN_DSCR_BF1_CH2_REG	接收通道 2 倒数第二个接收链表描述符地址	0x0218	RO
GDMA_OUTFIFO_STATUS_CH2_REG	发送通道 2 的 TX FIFO 状态	0x0258	RO
GDMA_OUT_STATE_CH2_REG	发送通道 2 的发送状态	0x0264	RO
GDMA_OUT_EOF_DES_ADDR_CH2_REG	发送通道 2 传输完成时的发送链表描述符地址	0x0268	RO
GDMA_OUT_EOF_BFR_DES_ADDR_CH2_REG	发送通道 2 传输完成时的最后一个发送链表描述符地址	0x026C	RO
GDMA_OUT_DSCR_CH2_REG	发送通道 2 当前的发送链表描述符地址	0x0270	RO
GDMA_OUT_DSCR_BF0_CH2_REG	发送通道 2 最后一个发送链表描述符地址	0x0274	RO
GDMA_OUT_DSCR_BF1_CH2_REG	发送通道 2 倒数第二个发送链表描述符地址	0x0278	RO
优先级寄存器			
GDMA_IN_PRI_CH0_REG	接收通道 0 的优先级寄存器	0x009C	R/W
GDMA_OUT_PRI_CH0_REG	发送通道 0 的优先级寄存器	0x00FC	R/W
GDMA_IN_PRI_CH1_REG	接收通道 1 的优先级寄存器	0x015C	R/W
GDMA_OUT_PRI_CH1_REG	发送通道 1 的优先级寄存器	0x01BC	R/W
GDMA_IN_PRI_CH2_REG	接收通道 2 的优先级寄存器	0x021C	R/W
GDMA_OUT_PRI_CH2_REG	发送通道 2 的优先级寄存器	0x027C	R/W
外设选择寄存器			
GDMA_IN_PERI_SEL_CH0_REG	接收通道 0 的外设选择	0x00A0	R/W
GDMA_OUT_PERI_SEL_CH0_REG	发送通道 0 的外设选择	0x0100	R/W
GDMA_IN_PERI_SEL_CH1_REG	接收通道 1 的外设选择	0x0160	R/W
GDMA_OUT_PERI_SEL_CH1_REG	发送通道 1 的外设选择	0x01C0	R/W
GDMA_IN_PERI_SEL_CH2_REG	接收通道 2 的外设选择	0x0220	R/W
GDMA_OUT_PERI_SEL_CH2_REG	发送通道 2 的外设选择	0x0280	R/W

2.8 寄存器

本小节的所有地址均为相对于通用 DMA 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 2.1. GDMA_INT_RAW_CH*n*_REG (*n*: 0-2) (0x0000+16n*)**



GDMA_IN_DONE_CH*n*_INT_RAW 接收通道 0 接收到接收链表描述符指向的最后一个字节数据时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_IN_SUC_EOF_CH*n*_INT_RAW 接收通道 0 接收到接收链表描述符指向的最后一个字节数据时，该原始中断位翻转至高电平。接收通道 0 连接 UHCI0，接收到接收链表描述符指向的最后一个字节数据且没有检测到数据错误时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_IN_ERR_EOF_CH*n*_INT_RAW 仅在接收通道 0 连接 UHCI0，且检测到数据错误时，该原始中断位翻转至高电平。如通道连接其他外设，该原始中断保留。(R/WTC/SS)

GDMA_OUT_DONE_CH*n*_INT_RAW 发送通道 0 将发送链表描述符指向的最后一个字节数据发送给外设时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_OUT_EOF_CH*n*_INT_RAW 发送通道 0 从存储器读取了发送链表描述符指向的最后一个字节数据时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_IN_DSCR_ERR_CH*n*_INT_RAW 接收通道 0 检测到接收链表描述符错误时，包括 owner 位错误、接收链表描述符的第二个字和第三个字错误，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_OUT_DSCR_ERR_CH*n*_INT_RAW 发送通道 0 检测到发送链表描述符错误时，包括 owner 位错误、发送链表描述符的第二个字和第三个字错误，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_IN_DSCR_EMPTY_CH*n*_INT_RAW 接收通道 0 接收链表指向的 RX FIFO 已满，数据接收未完成，但没有更多接收链表时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_OUT_TOTAL_EOF_CH*n*_INT_RAW 发送通道 0 发送了发送链表（包括一个或几个发送链表描述符）对应的数据时，该原始中断位翻转至高电平。(R/WTC/SS)

见下页...

Register 2.1. GDMA_INT_RAW_CH*n*_REG (*n*: 0-2) (0x0000+16n*)**

接上页...

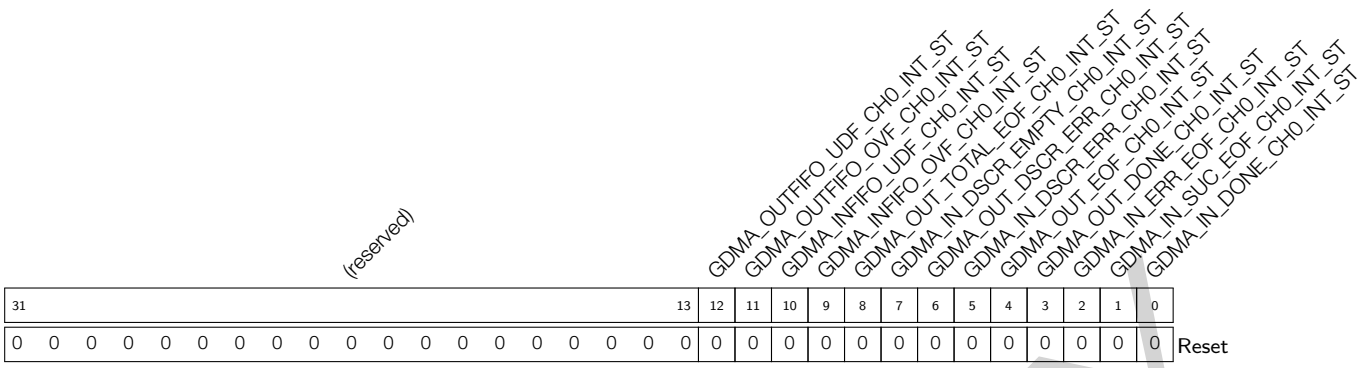
GDMA_INFIFO_OVF_CH*n*_INT_RAW 接收通道 0 的 L1 FIFO 上溢时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_INFIFO_UDF_CH*n*_INT_RAW 接收通道 0 的 L1 FIFO 下溢时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_OUTFIFO_OVF_CH*n*_INT_RAW 发送通道 0 的 L1 FIFO 上溢时，该原始中断位翻转至高电平。(R/WTC/SS)

GDMA_OUTFIFO_UDF_CH*n*_INT_RAW 发送通道 0 的 L1 FIFO 下溢时，该原始中断位翻转至高电平。(R/WTC/SS)

Register 2.2. GDMA_INT_ST_CH*n*_REG (*n*: 0-2) (0x0004+16n*)**



GDMA_IN_DONE_CH*n*_INT_ST IN_DONE_CH_INT 中断的原始状态位。(RO)

GDMA_IN_SUC_EOF_CH*n*_INT_ST IN_SUC_EOF_CH_INT 中断的原始状态位。(RO)

GDMA_IN_ERR_EOF_CH*n*_INT_ST IN_ERR_EOF_CH_INT 中断的原始状态位。(RO)

GDMA_OUT_DONE_CH*n*_INT_ST OUT_DONE_CH_INT 中断的原始状态位。(RO)

GDMA_OUT_EOF_CH*n*_INT_ST OUT_EOF_CH_INT 中断的原始状态位。(RO)

GDMA_IN_DSCR_ERR_CH*n*_INT_ST IN_DSCR_ERR_CH_INT 中断的原始状态位。(RO)

GDMA_OUT_DSCR_ERR_CH*n*_INT_ST OUT_DSCR_ERR_CH_INT 中断的原始状态位。(RO)

GDMA_IN_DSCR_EMPTY_CH*n*_INT_ST IN_DSCR_EMPTY_CH_INT 中断的原始状态位。(RO)

GDMA_OUT_TOTAL_EOF_CH*n*_INT_ST OUT_TOTAL_EOF_CH_INT 中断的原始状态位。(RO)

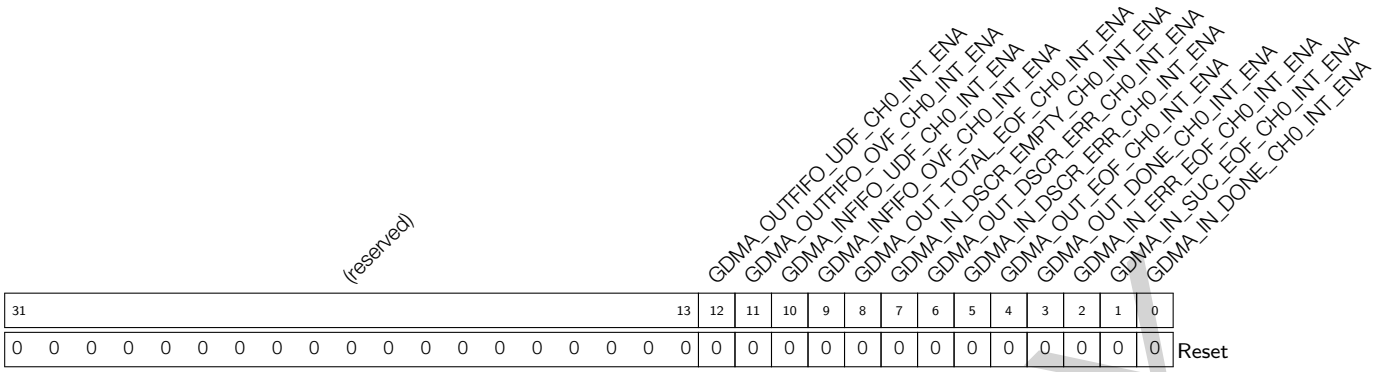
GDMA_INFIFO_OVF_CH*n*_INT_ST INFIFO_OVF_L1_CH_INT 中断的原始状态位。(RO)

GDMA_INFIFO_UDF_CH*n*_INT_ST INFIFO_UDF_L1_CH_INT 中断的原始状态位。(RO)

GDMA_OUTFIFO_OVF_CH*n*_INT_ST OUTFIFO_OVF_L1_CH_INT 中断的原始状态位。(RO)

GDMA_OUTFIFO_UDF_CH*n*_INT_ST OUTFIFO_UDF_L1_CH_INT 中断的原始状态位。(RO)

Register 2.3. GDMA_INT_ENA_CH*n*_REG (*n*: 0-2) (0x0008+16n*)**



GDMA_IN_DONE_CH*n*_INT_ENA IN_DONE_CH_INT 中断的使能位。(R/W)

GDMA_IN_SUC_EOF_CH*n*_INT_ENA IN_SUC_EOF_CH_INT 中断的使能位。(R/W)

GDMA_IN_ERR_EOF_CH*n*_INT_ENA IN_ERR_EOF_CH_INT 中断的使能位。(R/W)

GDMA_OUT_DONE_CH*n*_INT_ENA OUT_DONE_CH_INT 中断的使能位。(R/W)

GDMA_OUT_EOF_CH*n*_INT_ENA OUT_DONE_CH_INT 中断的使能位。(R/W)

GDMA_IN_DSCR_ERR_CH*n*_INT_ENA IN_DSCR_ERR_CH_INT 中断的使能位。(R/W)

GDMA_OUT_DSCR_ERR_CH*n*_INT_ENA OUT_DSCR_ERR_CH_INT 中断的使能位。(R/W)

GDMA_IN_DSCR_EMPTY_CH*n*_INT_ENA IN_DSCR_EMPTY_CH_INT 中断的使能位。(R/W)

GDMA_OUT_TOTAL_EOF_CH*n*_INT_ENA OUT_TOTAL_EOF_CH_INT 中断的使能位。(R/W)

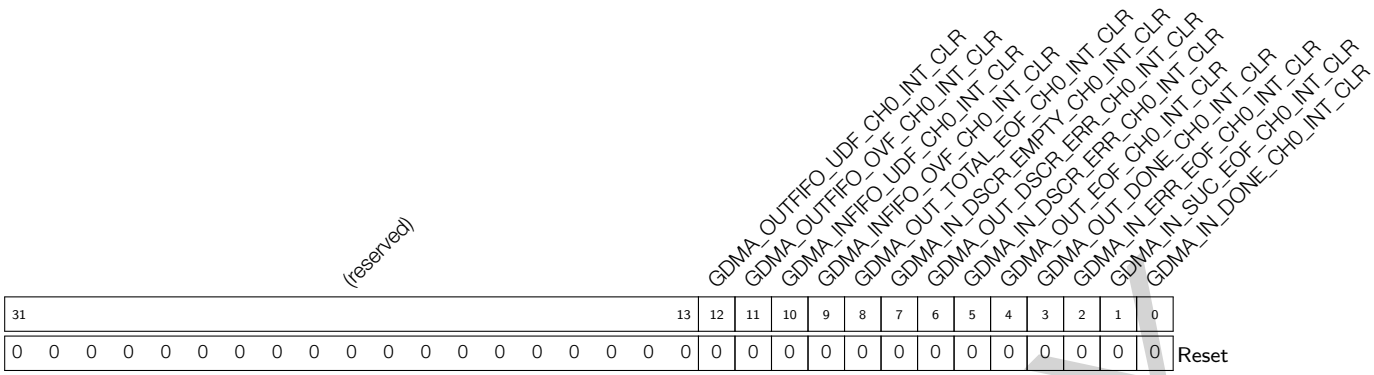
GDMA_INFIFO_OVF_CH*n*_INT_ENA INFIFO_OVF_L1_CH_INT 中断的使能位。(R/W)

GDMA_INFIFO_UDF_CH*n*_INT_ENA INFIFO_UDF_L1_CH_INT 中断的使能位。(R/W)

GDMA_OUTFIFO_OVF_CH*n*_INT_ENA OUTFIFO_OVF_L1_CH_INT 中断的使能位。(R/W)

GDMA_OUTFIFO_UDF_CH*n*_INT_ENA OUTFIFO_UDF_L1_CH_INT 中断的使能位。(R/W)

Register 2.4. GDMA_INT_CLR_CH*n*_REG (*n*: 0-2) (0x000C+16n*)**



GDMA_IN_DONE_CH*n*_INT_CLR 置位此位，清除 IN_DONE_CH_INT 中断。(WT)

GDMA_IN_SUC_EOF_CH*n*_INT_CLR 置位此位，清除 IN_SUC_EOF_CH_INT 中断。(WT)

GDMA_IN_ERR_EOF_CH*n*_INT_CLR 置位此位，清除 IN_ERR_EOF_CH_INT 中断。(WT)

GDMA_OUT_DONE_CH*n*_INT_CLR 置位此位，清除 OUT_DONE_CH_INT 中断。(WT)

GDMA_OUT_EOF_CH*n*_INT_CLR 置位此位，清除 OUT_EOF_CH_INT 中断。(WT)

GDMA_IN_DSCR_ERR_CH*n*_INT_CLR 置位此位，清除 IN_DSCR_ERR_CH_INT 中断。(WT)

GDMA_OUT_DSCR_ERR_CH*n*_INT_CLR 置位此位，清除 OUT_DSCR_ERR_CH_INT 中断。(WT)

GDMA_IN_DSCR_EMPTY_CH*n*_INT_CLR 置位此位，清除 IN_DSCR_EMPTY_CH_INT 中断。(WT)

GDMA_OUT_TOTAL_EOF_CH*n*_INT_CLR 置位此位，清除 OUT_TOTAL_EOF_CH_INT 中断。(WT)

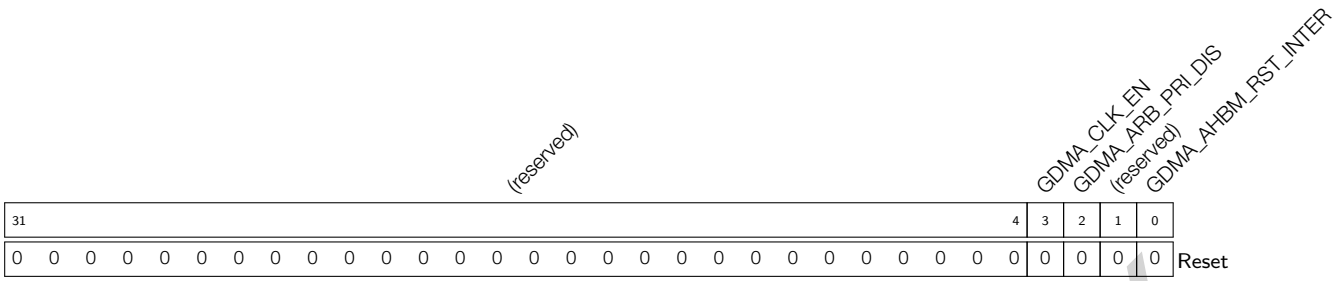
GDMA_INFIFO_OVF_CH*n*_INT_CLR 置位此位，清除 INFIFO_OVF_L1_CH_INT 中断。(WT)

GDMA_INFIFO_UDF_CH*n*_INT_CLR 置位此位，清除 INFIFO_UDF_L1_CH_INT 中断。(WT)

GDMA_OUTFIFO_OVF_CH*n*_INT_CLR 置位此位，清除 OUTFIFO_OVF_L1_CH_INT 中断。(WT)

GDMA_OUTFIFO_UDF_CH*n*_INT_CLR 置位此位，清除 OUTFIFO_UDF_L1_CH_INT 中断。(WT)

Register 2.5. GDMA_MISC_CONF_REG (0x0044)

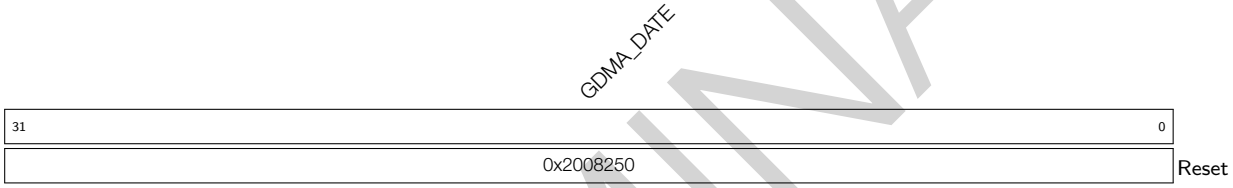


GDMA_AHBM_RST_INTER 置位此位，然后清零此位，重置内部 AHB 状态机。(R/W)

GDMA_ARB_PRI_DIS 置位此位，关闭优先级仲裁功能。(R/W)

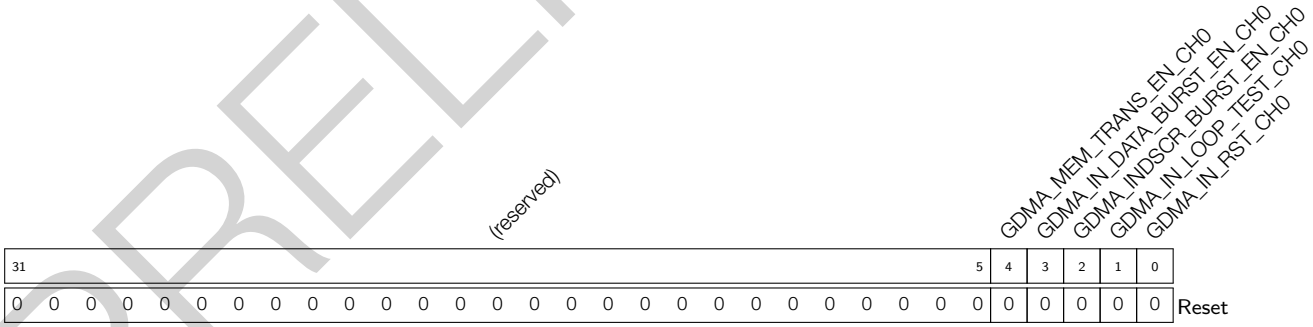
GDMA_CLK_EN reg_clk_en (R/W)

Register 2.6. GDMA_DATE_REG (0x0048)



GDMA_DATE 寄存器版本。(R/W)

Register 2.7. GDMA_IN_CONF0_CH*n*_REG (*n*: 0-2) (0x0070+192n*)**



GDMA_IN_RST_CH*n* 用于复位 GDMA 通道 0 的 RX 状态机和 RX FIFO 指针。(R/W)

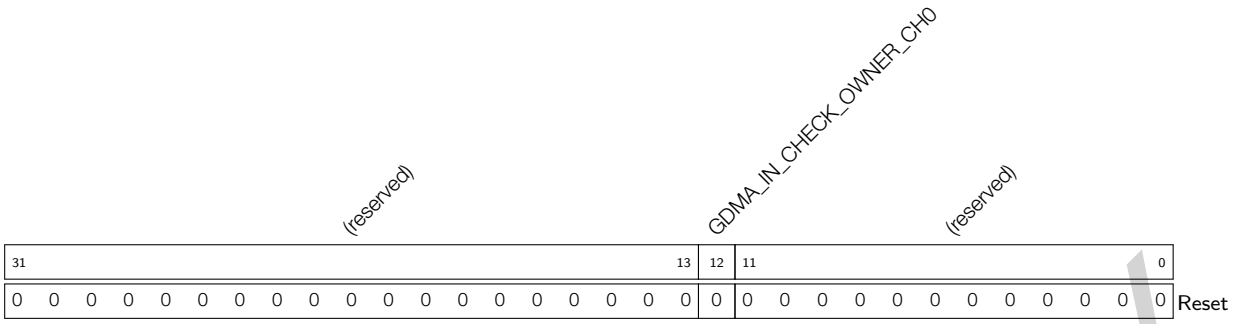
GDMA_IN_LOOP_TEST_CH*n* 用于硬件填充接收链表描述符的 owner 位。(R/W)

GDMA_INDSCR_BURST_EN_CH*n* 将此位置 1，在接收通道 0 访问内部 SRAM 读取接收链表描述符时使能 INCR 突发传输。(R/W)

GDMA_IN_DATA_BURST_EN_CH*n* 将此位置 1，在接收通道 0 访问内部 SRAM 接收数据时使能 INCR 突发传输。(R/W)

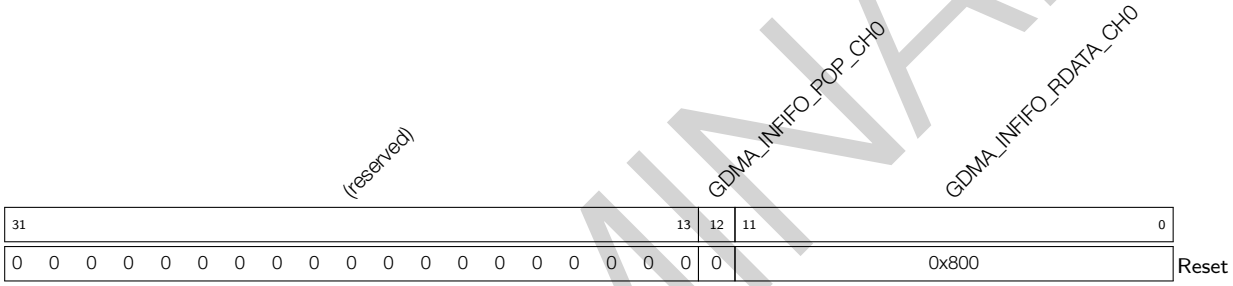
GDMA_MEM_TRANS_EN_CH*n* 将此位置 1，使能 GDMA 存储器到存储器自动传输。(R/W)

Register 2.8. GDMA_IN_CONF1_CH*n*_REG (*n*: 0-2) (0x0074+192n*)**



GDMA_IN_CHECK_OWNER_CH*n* Set this bit to enable checking the owner attribute of the link descriptor. (R/W)

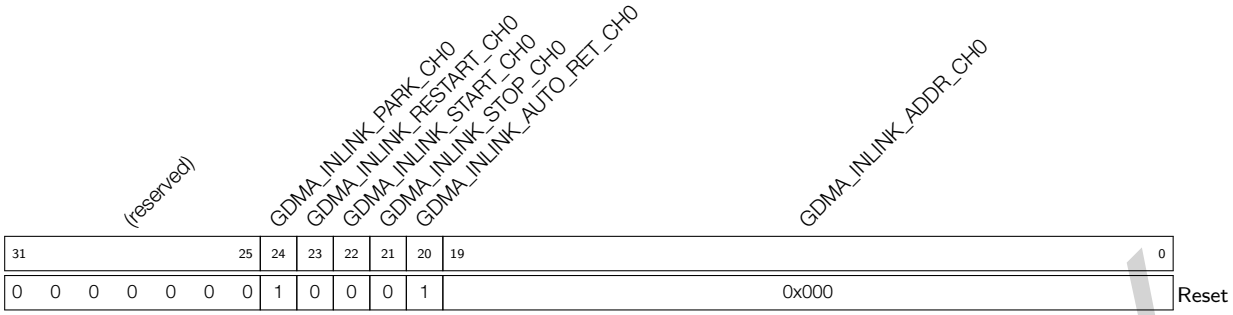
Register 2.9. GDMA_IN_POP_CH*n*_REG (*n*: 0-2) (0x007C+192n*)**



GDMA_INFIFO_RDATA_CH*n* 存储从 GDMA FIFO 中弹出的数据。(RO)

GDMA_INFIFO_POP_CH*n* 置位此位，从 GDMA FIFO 中弹出数据。(R/W/SC)

Register 2.10. GDMA_IN_LINK_CH*n*_REG (*n*: 0-2) (0x0080+192n*)**



GDMA_INLINK_ADDR_CH*n* 存储第一个接收链表描述符地址的低 20 位。(R/W)

GDMA_INLINK_AUTO_RET_CH*n* 当前接收数据有错误时，置位此位返回当前接收链表描述符的地址。(R/W)

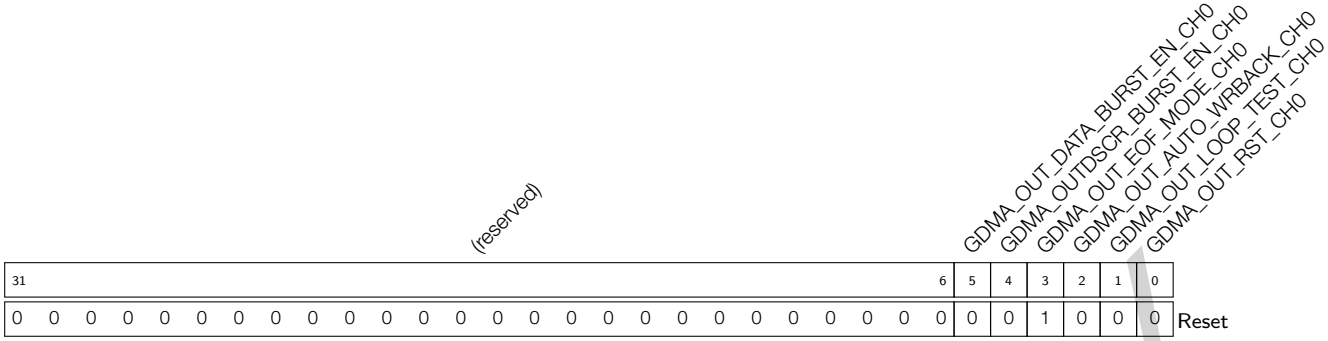
GDMA_INLINK_STOP_CH*n* 置位此位，停止处理接收链表描述符。(R/W/SC)

GDMA_INLINK_START_CH*n* 置位此位，开始处理接收链表描述符。(R/W/SC)

GDMA_INLINK_RESTART_CH*n* 置位此位，挂载新的接收链表描述符。(R/W/SC)

GDMA_INLINK_PARK_CH*n* 1：接收链表描述符的状态机空闲。0：接收链表描述符的状态机工作中。(RO)

Register 2.11. GDMA_OUT_CONF0_CH*n*_REG (*n*: 0-2) (0x00D0+192n*)**



GDMA_OUT_RST_CH*n* 用于复位 GDMA 通道 0 的 TX 状态机和 TX FIFO 指针。(R/W)

GDMA_OUT_LOOP_TEST_CH*n* 保留 (R/W)

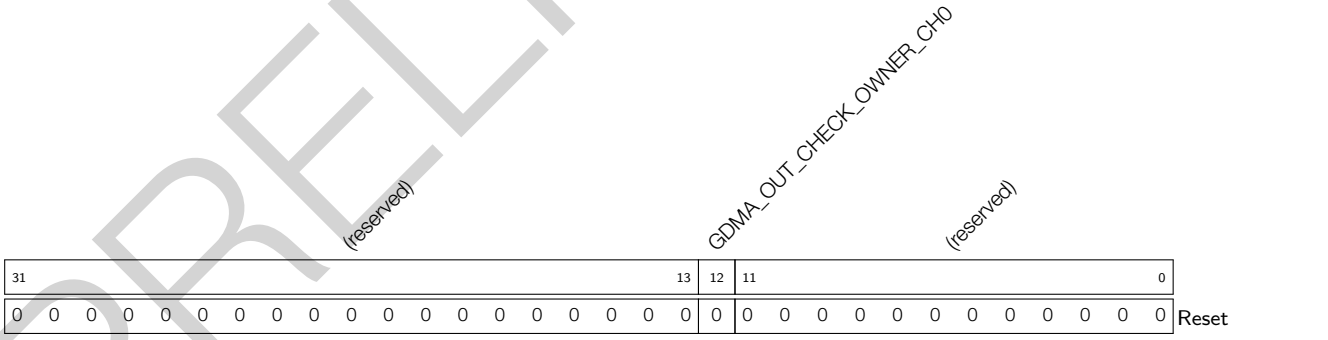
GDMA_OUT_AUTO_WRBACK_CH*n* 置位此位，在 TX FIFO 中所有数据发送出去后自动回写发送链表。(R/W)

GDMA_OUT_EOF_MODE_CH*n* 发送数据时生成 EOF 标志位。1：需要发送的数据已从 GDMA FIFO 中弹出时，发送通道 0 的 EOF 标志生成。(R/W)

GDMA_OUTDSCR_BURST_EN_CH*n* 将此位置 1，在发送通道 0 访问内部 SRAM 读取发送链表描述符时使能 INCR 突发传输。(R/W)

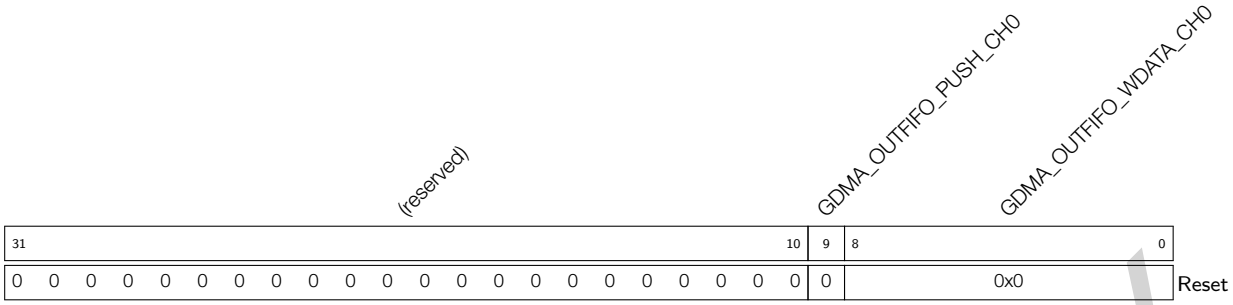
GDMA_OUT_DATA_BURST_EN_CH*n* 将此位置 1，在发送通道 0 访问内部 SRAM 发送数据时使能 INCR 突发传输。(R/W)

Register 2.12. GDMA_OUT_CONF1_CH*n*_REG (*n*: 0-2) (0x00D4+192n*)**



GDMA_OUT_CHECK_OWNER_CH*n* 置位此位，使能链表描述符的 owner 位检查。(R/W)

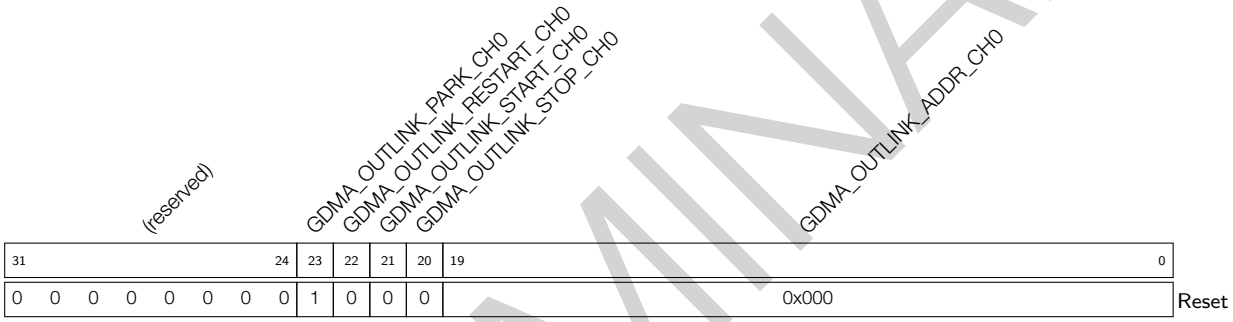
Register 2.13. GDMA_OUT_PUSH_CH*n*_REG (*n*: 0-2) (0x00DC+192n*)**



GDMA_OUTFIFO_WDATA_CH*n* 存储需推送至 GDMA FIFO 的数据。(R/W)

GDMA_OUTFIFO_PUSH_CH*n* 置位此位，将数据推送至 GDMA FIFO 中。(R/W/SC)

Register 2.14. GDMA_OUT_LINK_CH*n*_REG (*n*: 0-2) (0x00E0+192n*)**



GDMA_OUTLINK_ADDR_CH*n* 存储第一个发送链表描述符地址的低 20 位。(R/W)

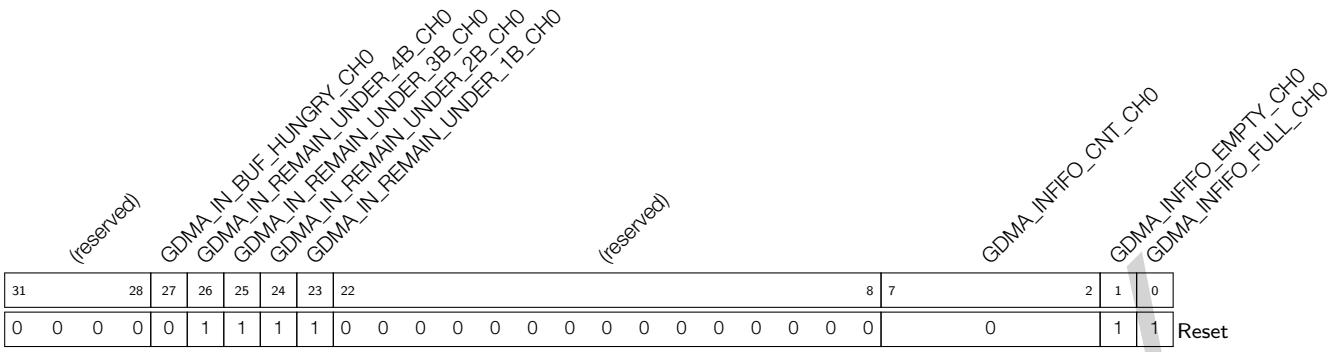
GDMA_OUTLINK_STOP_CH*n* 置位此位，停止处理发送链表描述符。(R/W/SC)

GDMA_OUTLINK_START_CH*n* 置位此位，开始处理发送链表描述符。(R/W/SC)

GDMA_OUTLINK_RESTART_CH*n* 置位此位，重新最后一个地址挂载新的发送链表。(R/W/SC)

GDMA_OUTLINK_PARK_CH*n* 1：发送链表描述符的状态机空闲。0：发送链表描述符的状态机工作中。(RO)

Register 2.15. GDMA_INFIFO_STATUS_CH*n*_REG (*n*: 0-2) (0x0078+192n*)**



GDMA_INFIFO_FULL_CH*n* 接收通道 0 的 L1 RX FIFO 已满。(RO)

GDMA_INFIFO_EMPTY_CH*n* 接收通道 0 的 L1 RX FIFO 为空。(RO)

GDMA_INFIFO_CNT_CH*n* 接收通道 0 的 L1 RX FIFO 存储的字节数。(RO)

GDMA_IN_REMAIN_UNDER_1B_CH*n* 保留 (RO)

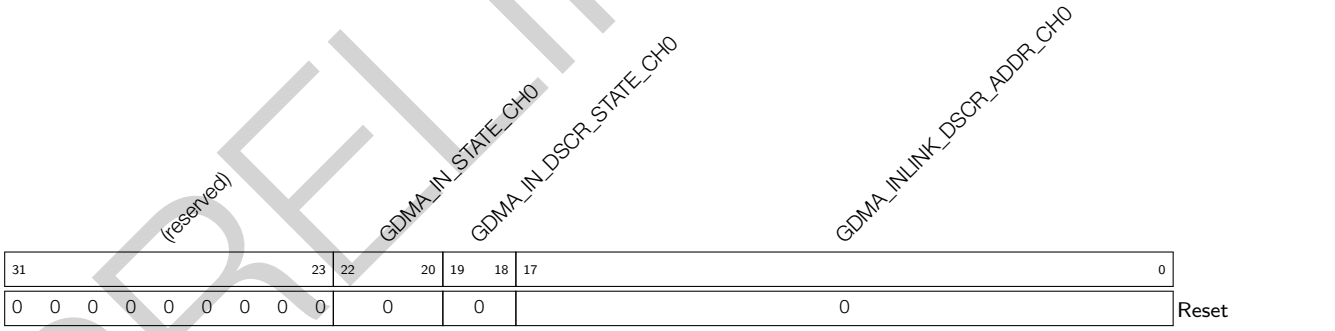
GDMA_IN_REMAIN_UNDER_2B_CH*n* 保留 (RO)

GDMA_IN_REMAIN_UNDER_3B_CH*n* 保留 (RO)

GDMA_IN_REMAIN_UNDER_4B_CH*n* 保留 (RO)

GDMA_IN_BUF_HUNGRY_CH*n* 保留 (RO)

Register 2.16. GDMA_IN_STATE_CH*n*_REG (*n*: 0-2) (0x0084+192n*)**

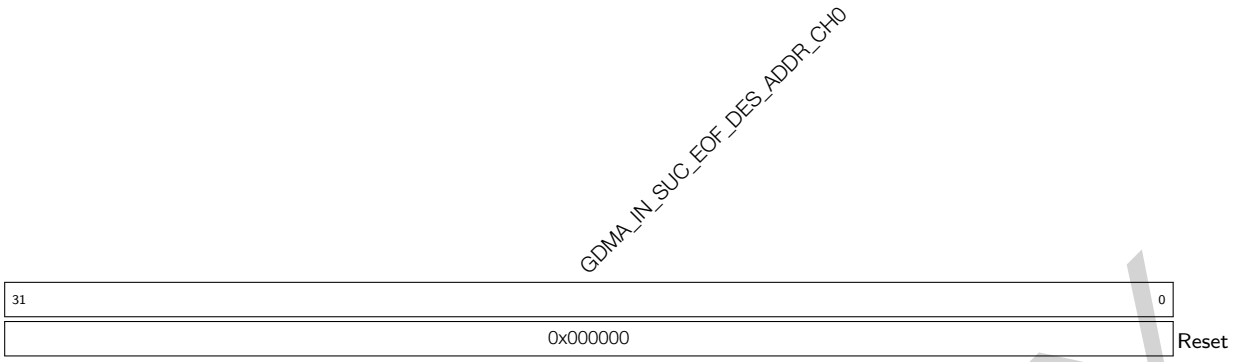


GDMA_INLINK_DSCR_ADDR_CH*n* 当前接收链表描述符的地址。(RO)

GDMA_IN_DSCR_STATE_CH*n* 保留 (RO)

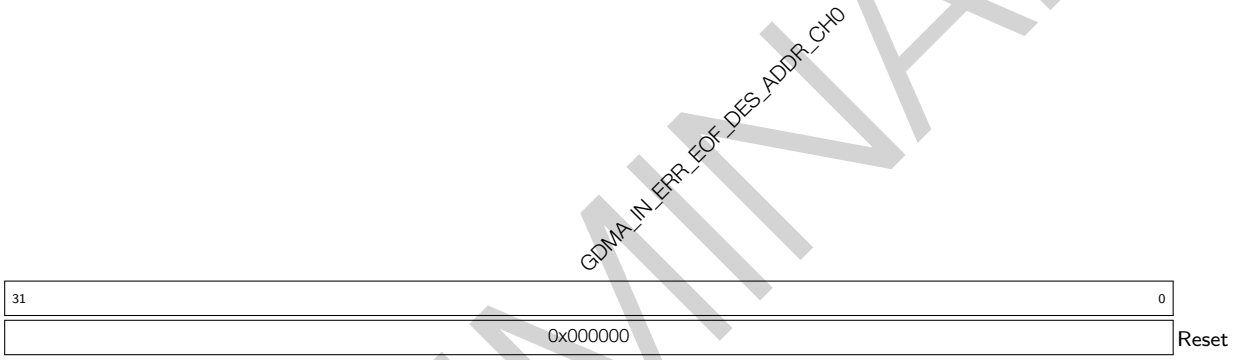
GDMA_IN_STATE_CH*n* 保留 (RO)

Register 2.17. GDMA_IN_SUC_EOF_DES_ADDR_CH*n*_REG (*n*: 0-2) (0x0088+192n*)**



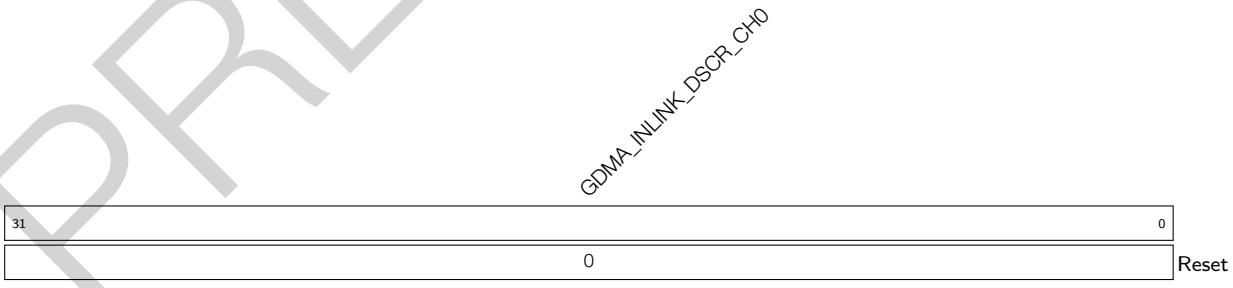
GDMA_IN_SUC_EOF_DES_ADDR_CH*n* 接收链表描述符的 EOF 为 1 时，该描述符的地址。(RO)

Register 2.18. GDMA_IN_ERR_EOF_DES_ADDR_CH*n*_REG (*n*: 0-2) (0x008C+192n*)**



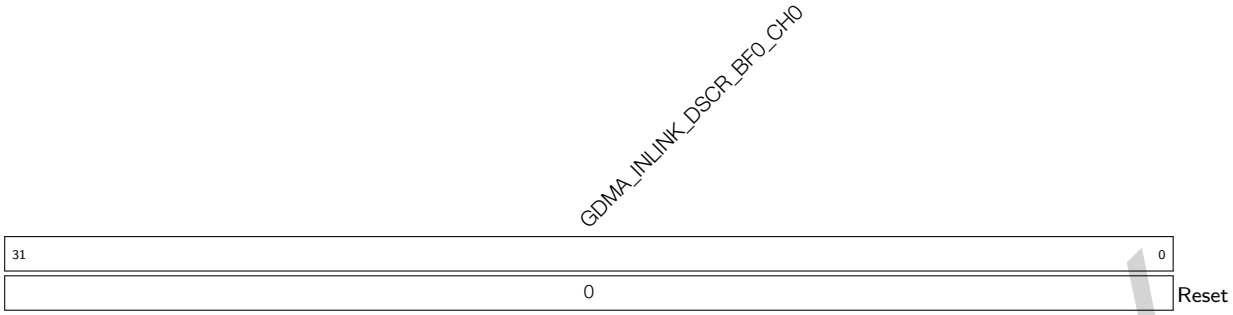
GDMA_IN_ERR_EOF_DES_ADDR_CH*n* 当前接收数据有错误时，接收链表描述符的地址。仅在连接外设为 UHCI 0 时使用。(RO)

Register 2.19. GDMA_IN_DSCR_CH*n*_REG (*n*: 0-2) (0x0090+192n*)**



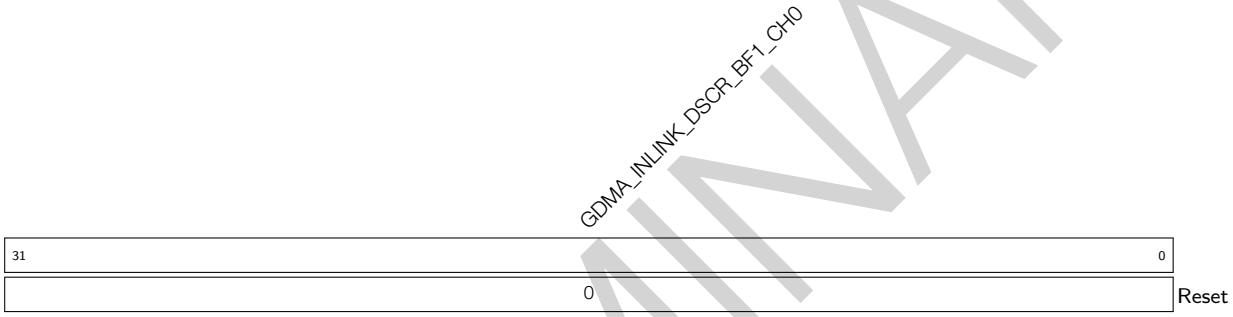
GDMA_INLINK_DSCR_CH*n* 当前接收链表描述符 x 的地址。(RO)

Register 2.20. GDMA_IN_DSCR_BF0_CH*n*_REG (*n*: 0-2) (0x0094+192n*)**



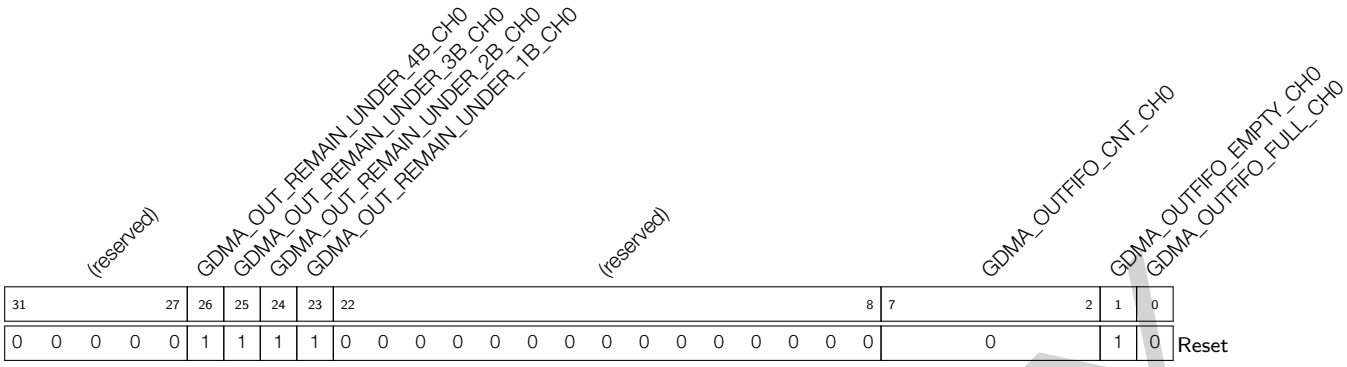
GDMA_INLINK_DSCR_BF0_CH*n* 最后一个接收链表描述符 x-1 的地址。(RO)

Register 2.21. GDMA_IN_DSCR_BF1_CH*n*_REG (*n*: 0-2) (0x0098+192n*)**



GDMA_INLINK_DSCR_BF1_CH*n* 倒数第二个接收链表描述符 x-2 的地址。(RO)

Register 2.22. GDMA_OUTFIFO_STATUS_CH*n*_REG (*n*: 0-2) (0x00D8+192n*)**



GDMA_OUTFIFO_FULL_CH*n* 发送通道 0 的 L1 TX FIFO 已满。(RO)

GDMA_OUTFIFO_EMPTY_CH*n* 发送通道 0 的 L1 TX FIFO 为空。(RO)

GDMA_OUTFIFO_CNT_CH*n* 发送通道 0 的 L1 TX FIFO 存储的字节数。(RO)

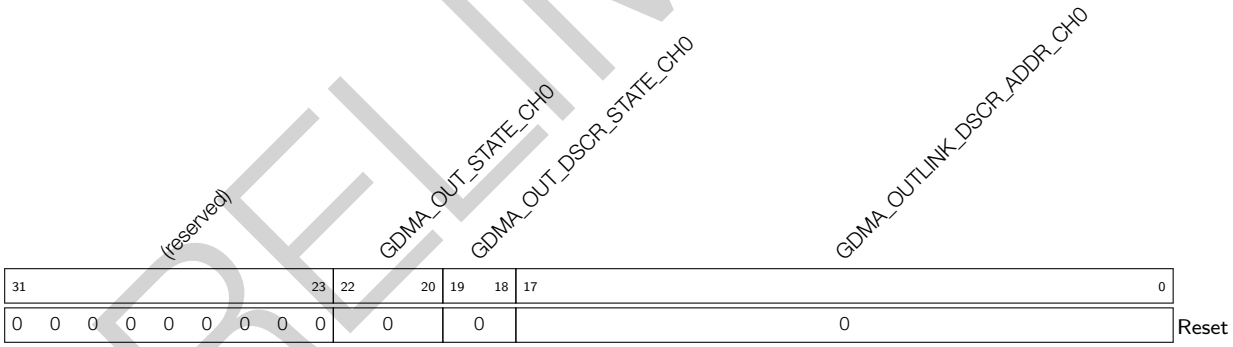
GDMA_OUT_REMAIN_UNDER_1B_CH*n* 保留 (RO)

GDMA_OUT_REMAIN_UNDER_2B_CH*n* 保留 (RO)

GDMA_OUT_REMAIN_UNDER_3B_CH*n* 保留 (RO)

GDMA_OUT_REMAIN_UNDER_4B_CH*n* 保留 (RO)

Register 2.23. GDMA_OUT_STATE_CH*n*_REG (*n*: 0-2) (0x00E4+192n*)**

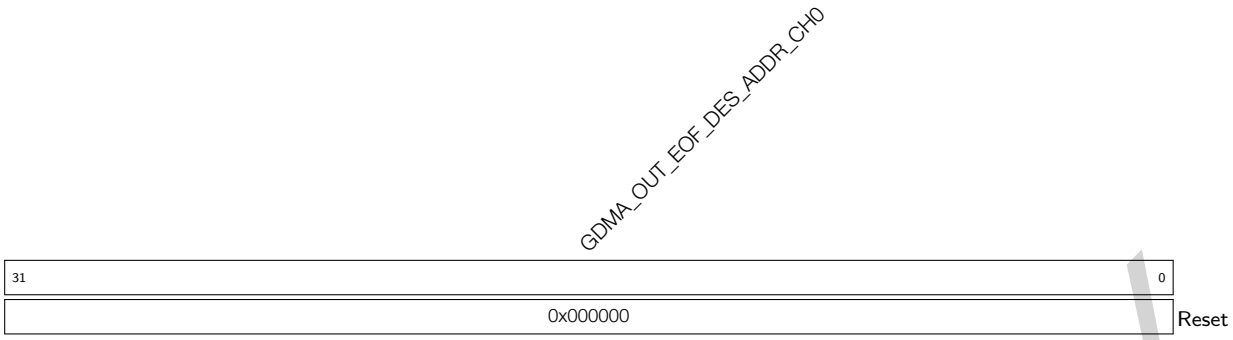


GDMA_OUTLINK_DSCR_ADDR_CH*n* 当前发送链表描述符的地址。(RO)

GDMA_OUT_DSCR_STATE_CH*n* 保留 (RO)

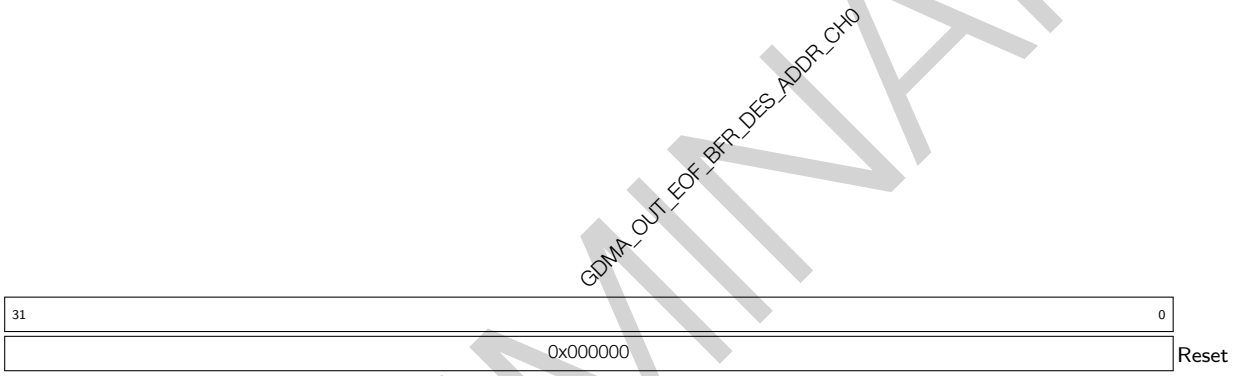
GDMA_OUT_STATE_CH*n* 保留 (RO)

Register 2.24. GDMA_OUT_EOF_DES_ADDR_CH*n*_REG (*n*: 0-2) (0x00E8+192n*)**



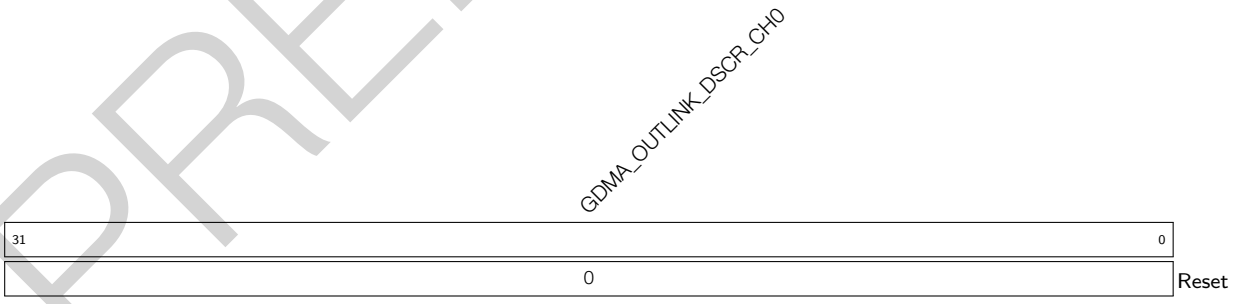
GDMA_OUT_EOF_DES_ADDR_CH*n* 发送链表描述符的 EOF 为 1 时，该描述符的地址。(RO)

Register 2.25. GDMA_OUT_EOF_BFR_DES_ADDR_CH*n*_REG (*n*: 0-2) (0x00EC+192n*)**



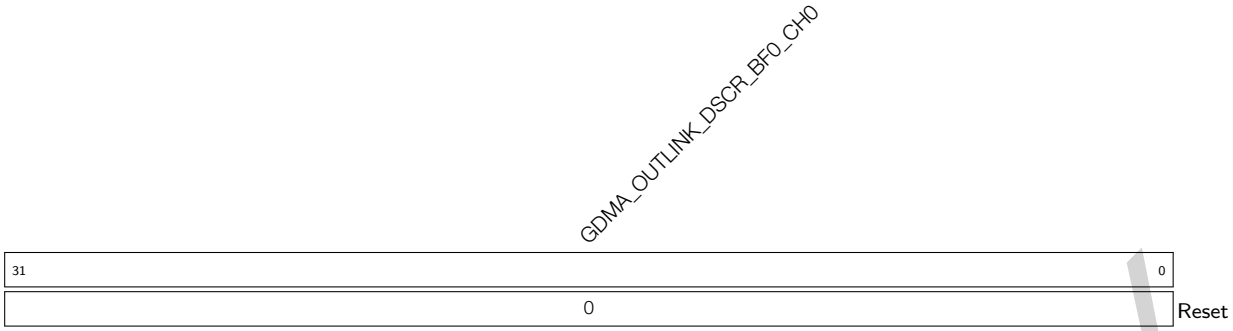
GDMA_OUT_EOF_BFR_DES_ADDR_CH*n* 倒数第二个发送链表描述符的地址。(RO)

Register 2.26. GDMA_OUT_DSCR_CH*n*_REG (*n*: 0-2) (0x00F0+192n*)**



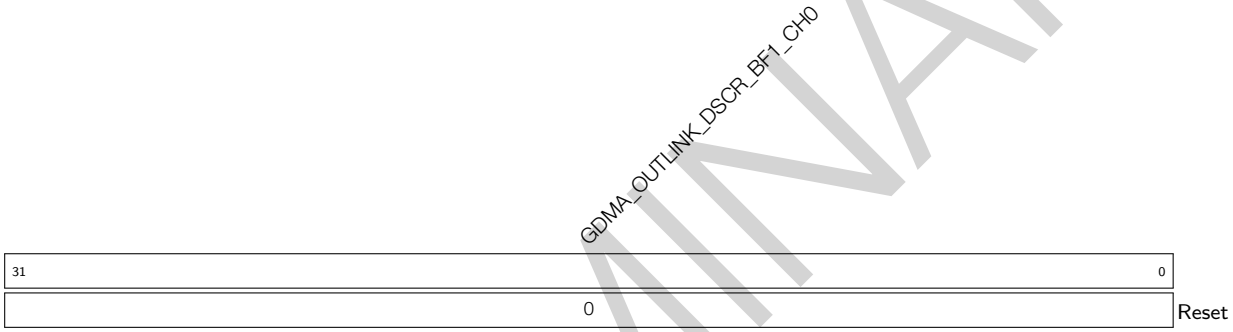
GDMA_OUTLINK_DSCR_CH*n* 当前链表描述符 y 的地址。(RO)

Register 2.27. GDMA_OUT_DSCR_BF0_CH*n*_REG (*n*: 0-2) (0x00F4+192n*)**



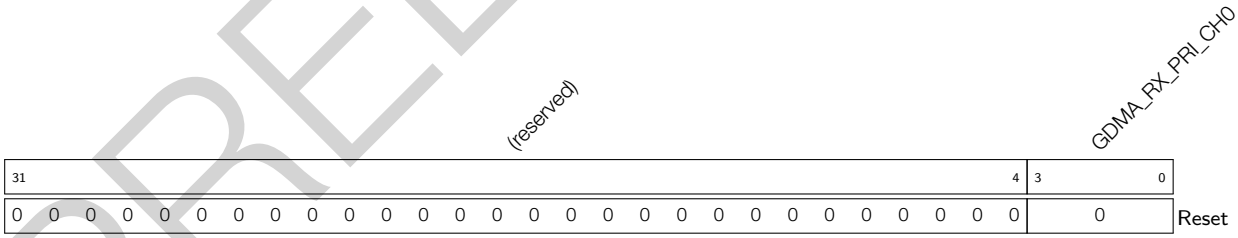
GDMA_OUTLINK_DSCR_BF0_CH*n* 最后一个发送链表描述符 y-1 的地址。(RO)

Register 2.28. GDMA_OUT_DSCR_BF1_CH*n*_REG (*n*: 0-2) (0x00F8+192n*)**



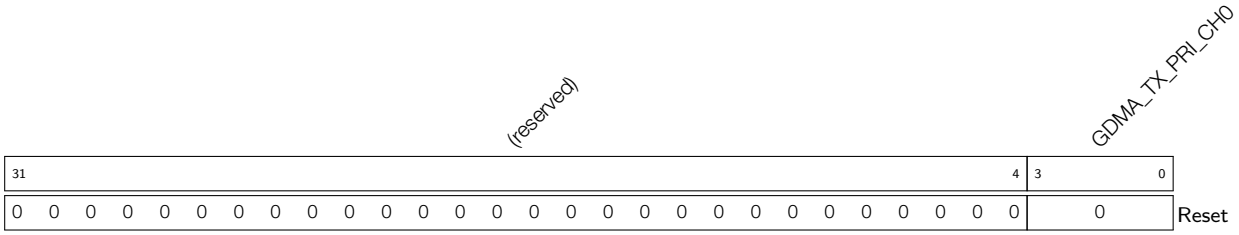
GDMA_OUTLINK_DSCR_BF1_CH*n* 倒数第二个接收链表描述符 x-2 的地址。(RO)

Register 2.29. GDMA_IN_PRI_CH*n*_REG (*n*: 0-2) (0x009C+192n*)**



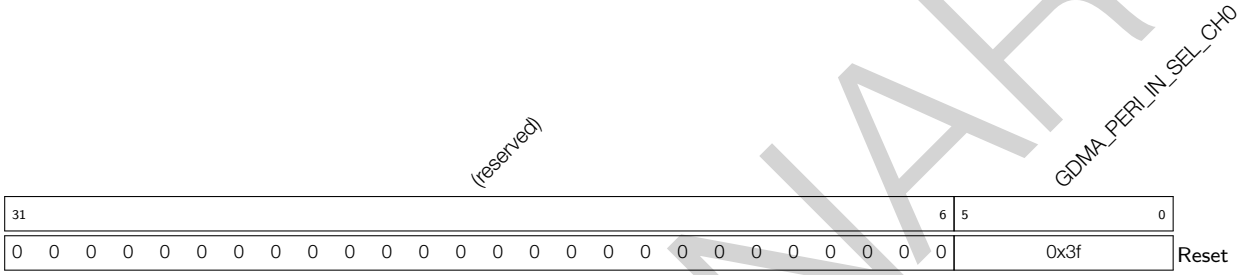
GDMA_RX_PRI_CH*n* 接收通道 0 的优先级。该值越大，优先级越高。(R/W)

Register 2.30. GDMA_OUT_PRI_CH*n*_REG (*n*: 0-2) (0x00FC+192n*)**



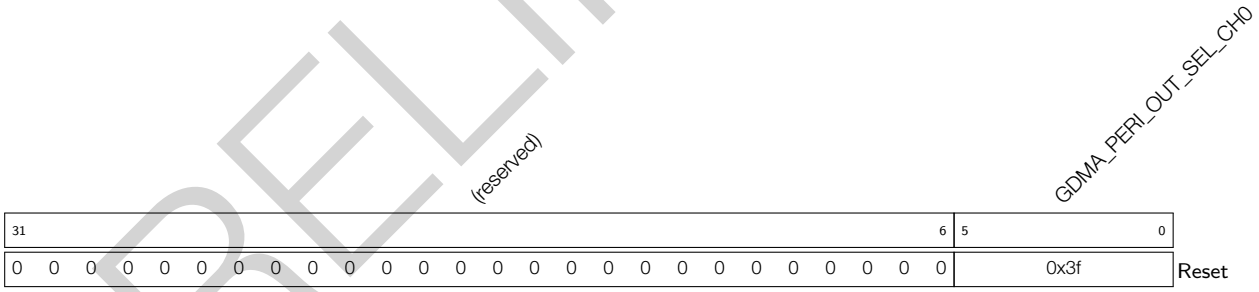
GDMA_TX_PRI_CH*n* 发送通道 0 的优先级。该值越大，优先级越高。(R/W)

Register 2.31. GDMA_IN_PERI_SEL_CH*n*_REG (*n*: 0-2) (0x00A0+192n*)**



GDMA_PERI_IN_SEL_CH*n* 用于选择接收通道 0 连接的外设。0：SPI2；1：保留；2：UHCI0；3：I2S；4：保留；5：保留；6：AES；7：SHA；8：ADC。(R/W)

Register 2.32. GDMA_OUT_PERI_SEL_CH*n*_REG (*n*: 0-2) (0x0100+192n*)**



GDMA_PERI_OUT_SEL_CH*n* 用于选择发送通道 0 连接的外设。0：SPI2；1：保留；2：UHCI0；3：I2S；4：保留；5：保留；6：AES；7：SHA；8：ADC。(R/W)

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。

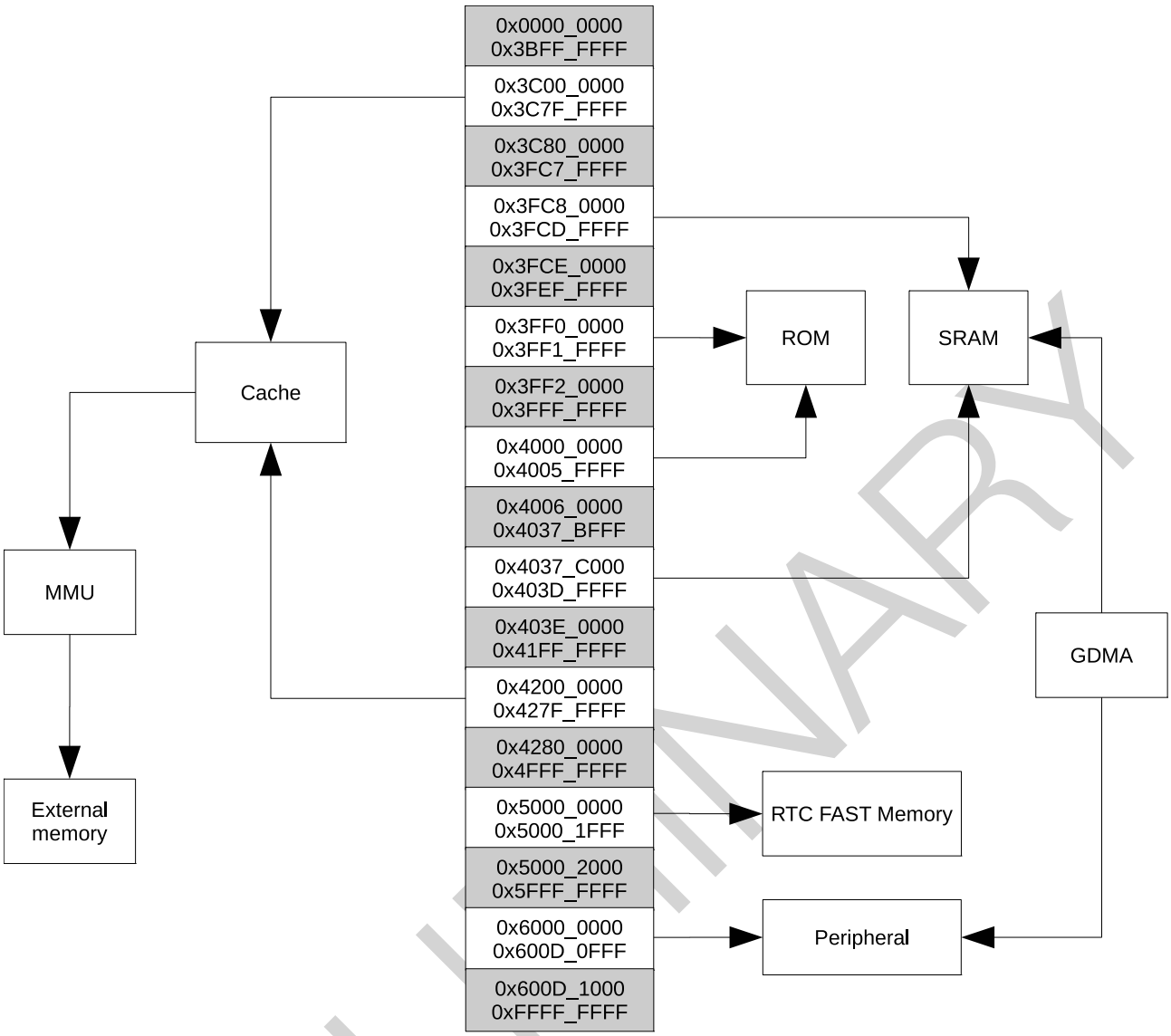


图 3-1. 系统结构与地址映射结构

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：

- 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。

– 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

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。

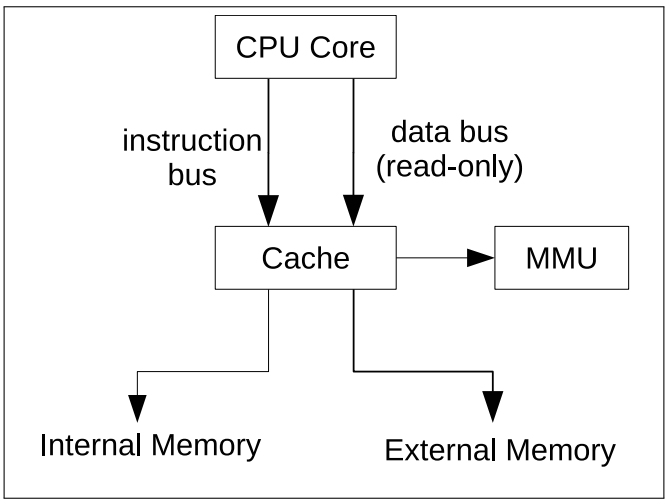


图 3-2. Cache 系统结构

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。

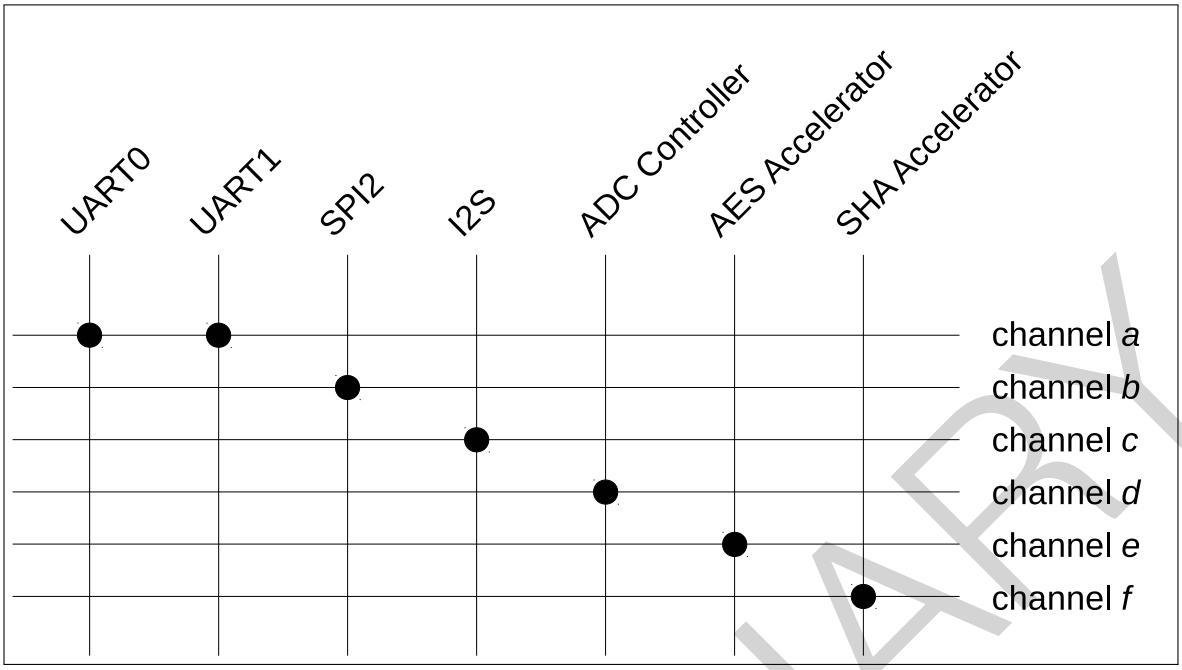


图 3-3. 具有 GDMA 功能的外设/模块

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	
Low-Power Management	0x6000_8000	0x6000_8FFF	4 KB	

目标	边界地址		容量	说明
	低位地址	高位地址		
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	
Reserved	0x600C_D000	0x600C_DFFF		

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

4 eFuse 控制器 (EFUSE)

4.1 概述

ESP32-C3 系统中有一块 4096 位的 eFuse，其中存储着参数内容。eFuse 的各个位一旦被烧写为 1，则不能再恢复为 0。eFuse 控制器按照软件配置完成对 eFuse 中各参数中的各个位的烧写。这些参数有些可以通过 eFuse 控制器被软件读取，有些直接由硬件模块使用。

4.2 主要特性

- 一次性可编程存储
- 烧写保护可配置
- 软件读取保护可配置
- 使用多种硬件编码方式保护参数内容

4.3 功能描述

4.3.1 结构

eFuse 从结构上分成 11 个块 (BLOCK0 ~ BLOCK10)。

BLOCK0 存储大部分参数，其中 9 位供硬件使用，软件不可见；还有 61 位处于保留状态，留作未来使用。

表 4-1 列出了 BLOCK0 中的参数名称、偏移地址、位宽、是否可供硬件使用、烧写保护，以及描述信息。

在这些参数中，**EFUSE_WR_DIS** 用于控制其他参数的烧写，**EFUSE_RD_DIS** 用于控制软件读取 BLOCK4 ~ BLOCK10。更多关于这两个参数的信息请见章节 4.3.1.1、4.3.1.2。

表 4-1. BLOCK0 参数

参数	偏移	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	描述
EFUSE_WR_DIS	0	32	Y	N/A	禁止 eFuse 烧写
EFUSE_RD_DIS	32	7	Y	0	禁止软件读取 eFuse BLOCK4 ~ 10 的内容
EFUSE_DIS_ICACHE	40	1	Y	2	关闭 ICache
EFUSE_DIS_USB_JTAG	41	1	Y	2	关闭 usb 转 jtag 功能
EFUSE_DIS_DOWNLOAD_ICACHE	42	1	Y	2	在 Download 模式下关闭 ICache
EFUSE_DIS_USB_SERIAL_JTAG	43	1	Y	2	禁用 usb_serial_jtag 模块
EFUSE_DIS_FORCE_DOWNLOAD	44	1	Y	2	禁止强制芯片进入 Download 模式
EFUSE_DIS_TWAI	46	1	Y	2	关闭 TWAI 控制器功能
EFUSE_JTAG_SEL_ENABLE	47	1	Y	2	置 1 启用 strap 引脚（GPIO 10）来选择 usb 转 jtag 功能或直接使用 jtag（GPIO 10 为 1 表明选择 usb 转 jtag 功能，为 0 表明直接使用 jtag）。

参数	偏移	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	描述
EFUSE_SOFT_DIS_JTAG	48	3	Y	31	烧写奇数个 1 表示禁用 JTAG，可通过 HMAC 重新启动
EFUSE_DIS_PAD_JTAG	51	1	Y	2	硬件永远禁用 JTAG
EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT	52	1	Y	2	在 download boot 模式下禁用 flash 加密功能
EFUSE_USB_EXCHG_PINS	57	1	Y	30	交换 USB D+/D- 管脚
EFUSE_VDD_SPI_AS_GPIO	58	1	N	30	VDD SPI 引脚和普通 gpio 选择信号，置 1 表明作为普通 gpio 引脚。
EFUSE_WDT_DELAY_SEL	80	2	Y	3	选择 RTC WDT 超时阈值
EFUSE_SPI_BOOT_CRYPT_CNT	82	3	Y	4	使能 SPI boot 加解密，奇数个 1：使能；偶数个 1：关闭
EFUSE_SECURE_BOOT_KEY_REVOKE0	85	1	N	5	使能撤销第一个 secure boot（安全启动）密钥
EFUSE_SECURE_BOOT_KEY_REVOKE1	86	1	N	6	使能撤销第二个 secure boot 密钥
EFUSE_SECURE_BOOT_KEY_REVOKE2	87	1	N	7	使能撤销第三个 secure boot 密钥
EFUSE_KEY_PURPOSE_0	88	4	Y	8	Key0 用途 (purpose)，见表 4-2
EFUSE_KEY_PURPOSE_1	92	4	Y	9	Key1 用途，见表 4-2
EFUSE_KEY_PURPOSE_2	96	4	Y	10	Key2 用途，见表 4-2
EFUSE_KEY_PURPOSE_3	100	4	Y	11	Key3 用途，见表 4-2
EFUSE_KEY_PURPOSE_4	104	4	Y	12	Key4 用途，见表 4-2
EFUSE_KEY_PURPOSE_5	108	4	Y	13	Key5 用途，见表 4-2
EFUSE_SECURE_BOOT_EN	116	1	N	15	使能 secure boot
EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE	117	1	N	16	Secure boot 的撤销采用激进策略
EFUSE_FLASH_TPUW	124	4	N	18	上电后 flash 等待时间，单位为 ms/2，值为 15 时，等待时间为 7.5 ms
EFUSE_DIS_DOWNLOAD_MODE	128	1	N	18	关闭所有 download boot 模式
EFUSE_UART_PRINT_CHANNEL	130	1	N	18	置 1 禁止 usb 打印功能
EFUSE_DIS_USB_DOWNLOAD_MODE	132	1	N	18	在 UART download boot 模式下关闭 USB OTG 下载功能
EFUSE_ENABLE_SECURITY_DOWNLOAD	133	1	N	18	使能 UART 安全下载模式（仅支持读写 flash）
EFUSE_UART_PRINT_CONTROL	134	2	N	18	控制 UART boot 信息输出模式。2'b00：强制打印；2'b01：由 GPIO8 控制，低电平打印；2'b10：由 GPIO 8 控制，高电平打印；2'b11：强制关闭打印
EFUSE_FORCE_SEND_RESUME	141	1	N	18	强制 ROM 代码在 SPI 启动过程中发送 SPI flash 继续指令
EFUSE_SECURE_VERSION	142	16	N	18	安全版本（用于 ESP-IDF 的防回滚功能）

表 4-2 为密钥用途各个数值对应的含义。通过配置参数 EFUSE_KEY_PURPOSE_*n* 来声明 KEY*n* 用途 (*n*: 0 ~ 5)。

表 4-2. 密钥用途数值对应的含义

密钥用途数值	含义
0	指定为用户使用（仅为软件使用）
1	保留
2	指定为 XTS_AES_256_KEY_1 使用（用于 flash/SRAM 加解密）
3	指定为 XTS_AES_256_KEY_2 使用（用于 flash/SRAM 加解密）
4	指定为 XTS_AES_128_KEY 使用（用于 flash/SRAM 加解密）
5	指定为 HMAC Downstream（下行）模式（JTAG 和数字签名）使用
6	指定为 HMAC Downstream 模式下的 JTAG 使用
7	指定为 HMAC Downstream 模式下的数字签名使用
8	指定为 HMAC Upstream（上行）模式使用
9	指定为 SECURE_BOOT_DIGEST0 使用（secure boot 密钥摘要）
10	指定为 SECURE_BOOT_DIGEST1 使用（secure boot 密钥摘要）
11	指定为 SECURE_BOOT_DIGEST2 使用（secure boot 密钥摘要）

表 4-3 列出了 BLOCK1 ~ BLOCK10 中存储的参数的信息。

表 4-3. BLOCK1-10 参数

块	参数	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	EFUSE_RD_DIS 软件读取保护位	描述
BLOCK1	EFUSE_MAC	48	N	20	N/A	MAC 地址
	EFUSE_SPI_PAD_CONFIGURE	[0:5]	N	20	N/A	CLK
		[6:11]	N	20	N/A	Q (D1)
		[12:17]	N	20	N/A	D (D0)
		[18:23]	N	20	N/A	CS
		[24:29]	N	20	N/A	HD (D3)
		[30:35]	N	20	N/A	WP (D2)
		[36:41]	N	20	N/A	DQS
		[42:47]	N	20	N/A	D4
		[48:53]	N	20	N/A	D5
		[54:59]	N	20	N/A	D6
		[60:65]	N	20	N/A	D7
	EFUSE_SYS_DATA_PART0	78	N	20	N/A	系统数据
BLOCK2	EFUSE_SYS_DATA_PART1	256	N	21	N/A	系统数据
BLOCK3	EFUSE_USR_DATA	256	N	22	N/A	用户数据
BLOCK4	EFUSE_KEY0_DATA	256	Y	23	0	KEY0 或用户数据
BLOCK5	EFUSE_KEY1_DATA	256	Y	24	1	KEY1 或用户数据
BLOCK6	EFUSE_KEY2_DATA	256	Y	25	2	KEY2 或用户数据
BLOCK7	EFUSE_KEY3_DATA	256	Y	26	3	KEY3 或用户数据
BLOCK8	EFUSE_KEY4_DATA	256	Y	27	4	KEY4 或用户数据
BLOCK9	EFUSE_KEY5_DATA	256	Y	28	5	KEY5 或用户数据

块	参数	位宽	硬件使用	EFUSE_WR_DIS 烧写保护位	EFUSE_RD_DIS 软件读取保护位	描述
BLOCK10	EFUSE_SYS_DATA_PART2	256	N	29	6	系统数据

其中，BLOCK4 ~ 9 分别存储 KEY0 ~ 5，表示 eFuse 中至多可以烧写 6 个 256 位的密钥。每烧写一个密钥，还需要烧写该密钥用途的数值（见表 4-2）。例如，软件将用于 HMAC Downstream 模式下的 JTAG 功能的密钥烧写到 KEY3（即 BLOCK7），还需要将密钥用途的数值 6 烧写到 EFUSE_KEY_PURPOSE_3。

BLOCK1 ~ BLOCK10 均采用 RS 编码方式，因此参数烧写受到一定的限制，具体请参考章节 4.3.1.3 和章节 4.3.2。

4.3.1.1 EFUSE_WR_DIS

参数 EFUSE_WR_DIS 决定了 eFuse 中所有的参数是否处于烧写保护状态。烧写完 EFUSE_WR_DIS 参数后，需要更新 eFuse 读寄存器才能生效。

表 4-1 以及表 4-3 中的 “EFUSE_WR_DIS 烧写保护位” 列描述了各参数的烧写保护状态具体由 EFUSE_WR_DIS 的哪个位决定。

当某个参数对应的烧写保护位为 0 时，表示此参数未处于烧写保护状态，可以烧写该参数，但已经被烧写的参数不能被重复烧写。

当某个参数对应的烧写保护位为 1 时，表示此参数处于烧写保护状态，此参数的每一个位都无法被更改，未被烧写的位永远为 0，已经被烧写的位永远为 1。所以如果某个参数已经处于烧写保护状态了，则会一直处在该状态，无法再更改。

4.3.1.2 EFUSE_RD_DIS

所有参数中，只有 BLCOK4 ~ BLOCK10 的参数受软件读取保护状态的约束，即表 4-3 中 “EFUSE_RD_DIS 软件读取保护” 列非 “N/A” 的参数。烧写完 EFUSE_RD_DIS 参数后，需要更新 eFuse 读寄存器才能生效。

参数 EFUSE_RD_DIS 中的某个位为 0，表示此位管理的参数未处于软件读取保护状态；某个位为 1，表示此位管理的参数处于软件读取保护状态。

除 BLOCK4 ~ BLOCK10 之外，其他参数不受软件读取保护状态的约束，均可被软件读取。

BLOCK4 ~ BLOCK10 即使被配置处于读取保护状态，仍然可以通过设置 EFUSE_KEY_PURPOSE_*n* 被硬件使用。

4.3.1.3 数据存储方式

eFuse 使用硬件编码机制保护数据，对用户不可见。

BLOCK0 使用 4 备份方式存储参数，即 BLOCK0 中的所有参数（除了 EFUSE_WR_DIS ）均在 eFuse 中存储了 4 份。4 备份机制对软件不可见。

BLOCK1 ~ BLOCK10 使用 RS (44, 32) 编码方式，最多支持自动校正 5 个字节。本文 RS (44, 32) 使用的本源多项式为 $p(x) = x^8 + x^4 + x^3 + x^2 + 1$。

产生校验码的移位寄存器电路如图 4-1 所示，其中 gf_mul_n（n 为一个整数）为 $GF(2^8)$ 域中某一字节数据与元素 α^n 相乘的结果。

软件需要先对 32 字节参数进行 RS (44, 32) 编码得到 12 字节验证码，然后将参数及验证码一起烧入 eFuse。eFuse 控制器会在读 eFuse 的过程中自动完成解码和自动校正。

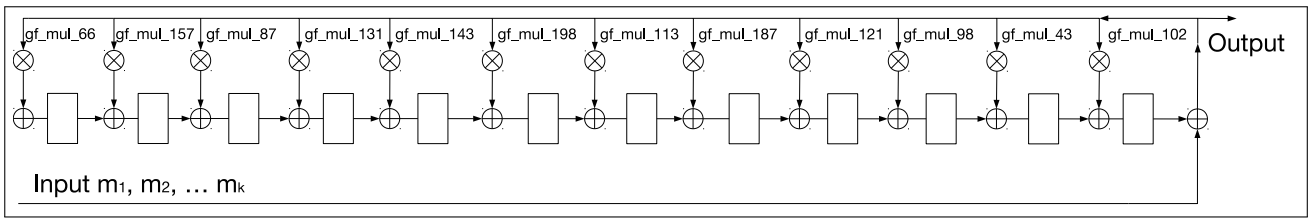


图 4-1. 移位寄存器电路图

由于 RS 校验码是在整个 256 位的 eFuse block 上生成的，因此每个 block 只能写入一次。

4.3.2 软件烧写参数

烧写 eFuse 参数时，需要按块烧写。BLOCK0 ~ BLOCK10 共用同一段地址来存储即将烧写的参数。通过配置 EFUSE_BLK_NUM 参数表明当前需要烧写的是哪一个块。

烧写 BLOCK0

当 EFUSE_BLK_NUM = 0 时，烧写 BLOCK0。EFUSE_PGM_DATA0_REG 寄存器存储着 EFUSE_WR_DIS。EFUSE_PGM_DATA1_REG ~ EFUSE_PGM_DATA5_REG 用来存储即将烧写的参数的有效信息，其中 9 位为硬件可用、软件不可见的有效信息，必须写入 0，对应位置为：

- EFUSE_PGM_DATA1_REG[24:21]
- EFUSE_PGM_DATA1_REG[31:27]

EFUSE_PGM_DATA6_REG ~ EFUSE_PGM_DATA7_REG 以及 EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG 中的数据不影响 BLOCK0 的烧写。

烧写 BLOCK1

当 EFUSE_BLK_NUM = 1 时，EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA5_REG 存储着 BLOCK1 即将烧写的参数，EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG 中存储着对应的 RS 校验码。EFUSE_PGM_DATA6_REG ~ EFUSE_PGM_DATA7_REG 中的数据不影响 BLOCK1 的烧写。软件计算 BLOCK1 的 RS 校验码时，应该视这 8 个字节的数据为 0。

烧写 BLOCK2 ~ 10

当 EFUSE_BLK_NUM = 2 ~ 10 时，EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG 存储着即将烧写的参数，EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG 中存储着对应的 RS 校验码。

烧写流程

烧写参数的流程如下：

1. 配置 EFUSE_BLK_NUM 参数，决定烧写哪一个块。
2. 将需要烧写的参数填写到寄存器 EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG 和 EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG 中。
3. 确保 eFuse 烧写电压 VDDQ 的配置正确，具体请参考章节 4.3.4。
4. 配置寄存器 EFUSE_CONF_REG 的 EFUSE_OP_CODE 位域为 0x5A5A。
5. 配置寄存器 EFUSE_CMD_REG 的 EFUSE_PGM_CMD 位域为 1。

6. 轮询寄存器 EFUSE_CMD_REG 直到其为 0x0，或者等待烧写完成中断产生。识别烧写/读取完成中断产生的方法详见章节 4.3.3 最后的说明。
7. 将 EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG 和 EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG 中写入的参数清零。
8. 执行更新 eFuse 读寄存器操作使写入的新值生效，具体请参考章节 4.3.3。

限制

BLOCK0 中不同的参数，甚至对于同一个参数中的不同位可以在多次烧写中分别完成。但是并不推荐这样做，而是建议尽量减少烧写次数。我们建议对于某个参数中的所有需要烧写的位都在一次烧写中完成。并且当 EFUSE_WR_DIS 的某个位管理的所有参数都烧写之后，就立即烧写 EFUSE_WR_DIS 的这个位。甚至可以在同一次烧写中既烧写 EFUSE_WR_DIS 的某个位管理的所有参数，同时也烧写 EFUSE_WR_DIS 的这个位。另外严禁对已经烧写了的位重复烧写，否则将发生烧写错误。

BLOCK1 中数据信息在出厂时已经烧写完毕，不允许再次烧写。

BLOCK2 ~ 10 中每一个 BLOCK 都只能烧写一次，不允许重复烧写。

4.3.3 软件读取参数

软件不能直接读取 eFuse 中烧写的信息内容。eFuse 控制器能够将烧写的数据信息读取到对应的地址段的寄存器内，软件再通过读取以 EFUSE_RD_ 开始的寄存器来获取 eFuse 信息。下表 4-4 列出了读取数据的寄存器名称以及对应烧写时的烧写寄存器名称。

表 4-4. 软件读取寄存器信息

BLOCK	读寄存器	烧写寄存器
0	EFUSE_RD_WR_DIS_REG	EFUSE_PGM_DATA0_REG
0	EFUSE_RD_REPEAT_DATA0 ~ 4_REG	EFUSE_PGM_DATA1 ~ 5_REG
1	EFUSE_RD_MAC_SPI_SYS_0 ~ 5_REG	EFUSE_PGM_DATA0 ~ 5_REG
2	EFUSE_RD_SYS_PART1_0 ~ 7_REG	EFUSE_PGM_DATA0 ~ 7_REG
3	EFUSE_RD_USR_DATA0 ~ 7_REG	EFUSE_PGM_DATA0 ~ 7_REG
4-9	EFUSE_RD_KEY*n*_DATA0 ~ 7_REG (*n*: 0 ~ 5)	EFUSE_PGM_DATA0 ~ 7_REG
10	EFUSE_RD_SYS_PART2_0 ~ 7_REG	EFUSE_PGM_DATA0 ~ 7_REG

更新 eFuse 读寄存器

eFuse 控制器读取内部 eFuse 来更新相应寄存器的数据。读取操作在系统复位时进行，也可以根据需要由软件手动触发（例如在需要读取新烧写 eFuse 中的数据内容时）。软件触发 eFuse 读取操作的流程如下：

1. 配置寄存器 EFUSE_CONF_REG 的 EFUSE_OP_CODE 位域为 0x5AA5。
2. 配置寄存器 EFUSE_CMD_REG 的 EFUSE_READ_CMD 位域为 1。
3. 轮询寄存器 EFUSE_CMD_REG 直到其为 0x0，或者等待 read_done interrupt（读取完成中断）产生，识别烧写/读取完成中断产生的方法详见下方说明。
4. 软件从 eFuse 存储器中读取参数的值。

eFuse 读寄存器中的数值将一直保持到下一次执行更新 eFuse 读操作。

烧写错误检测

烧写错误记录寄存器允许软件检测 eFuse 参数的备份是否有不一致的错误。

EFUSE_RD_REPEAT_ERR0 ~ 3_REG 寄存器用于指示 BLOCK0 中除了 EFUSE_WR_DIS 外的其他参数的烧写是否出错（对应位为 1 代表烧写出错，此位作废；为 0 代表烧写正确）。

EFUSE_RD_RS_ERR0 ~ 1_REG 寄存器记录 eFuse 读 BLOCK1 ~ BLOCK10 过程中，纠错的字节数目以及 RS 解码是否失败的信息。

每次更新 eFuse 读寄存器操作完成之后，上述寄存器内的数值都会被更新。

识别烧写/读取操作完成

识别烧写/读取操作完成的方法如下。位 1 对应烧写操作，位 0 对应读取操作。

- 方法 1：
  1. 轮询寄存器 EFUSE_INT_RAW_REG 的位 1/0，直到位 1/0 为 1，表示烧写/读取操作完成。
- 方法 2:
  1. 将寄存器 EFUSE_INT_ENA_REG 的位 1/0 置 1，使 eFuse 控制器能够产生烧写/读取完成中断。
  2. 配置中断矩阵使 CPU 能够响应 eFuse 的中断信号，可参见 *8 中断矩阵 (INTERRUPT) [to be added later]*。
  3. 等待烧写/读取完成中断产生。
  4. 对寄存器 EFUSE_INT_CLR_REG 的位 1/0 置 1 以清除烧写/读取完成中断。

4.3.4 eFuse VDDQ 时序

eFuse 控制器工作在 20 MHz 时钟频率下，其烧写电压 VDDQ 的配置参数需要满足以下条件：

- EFUSE_DAC_NUM（烧写电压上升周期数），默认烧写电压为 2.5 V，每个上升周期增加 0.01 V，该参数对应的默认值为 255；
- EFUSE_DAC_CLK_DIV（烧写电压时钟分频系数），要求烧写电压时钟周期大于 1 μs；
- EFUSE_PWR_ON_NUM （eFuse 烧写电压上电等待时间），要求该等待时间结束后烧写电压已稳定，即要求配置数值大于 EFUSE_DAC_CLK_DIV * EFUSE_DAC_NUM；
- EFUSE_PWR_OFF_NUM（烧写电压掉电等待时间），要求该时间大于 10 μs；

表 4-5. VDDQ 默认时序参数配置

EFUSE_DAC_NUM	EFUSE_DAC_CLK_DIV	EFUSE_PWR_ON_NUM	EFUSE_PWR_OFF_NUM
0xFF	0x28	0x3000	0x190

4.3.5 硬件模块使用参数

硬件模块使用参数是通过电路连接实现的，软件无法干预这个过程。硬件使用的参数为表 4-1 和 4-3 “硬件使用”一栏中标记为“Y”的参数。

4.3.6 中断

- 烧写完成中断：当 eFuse 烧写完成后，此中断被触发。如果要启动该中断信号，需将寄存器 EFUSE_INT_ENA_REG的 EFUSE_PGM_DONE_INT_ENA 域置 1。

- 读取完成中断：当 eFuse 读取完成后，此中断被触发。如果要启动该中断信号，需将寄存器 EFUSE_INT_ENA_REG 的 EFUSE_READ_DONE_INT_ENA 域置 1。

4.4 寄存器列表

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

07

名称	描述	地址	访问
烧写数据寄存器			
EFUSE_PGM_DATA0_REG	存放待烧写数据的第 0 个寄存器内容	0x0000	R/W
EFUSE_PGM_DATA1_REG	存放待烧写数据的第 1 个寄存器内容	0x0004	R/W
EFUSE_PGM_DATA2_REG	存放待烧写数据的第 2 个寄存器内容	0x0008	R/W
EFUSE_PGM_DATA3_REG	存放待烧写数据的第 3 个寄存器内容	0x000C	R/W
EFUSE_PGM_DATA4_REG	存放待烧写数据的第 4 个寄存器内容	0x0010	R/W
EFUSE_PGM_DATA5_REG	存放待烧写数据的第 5 个寄存器内容	0x0014	R/W
EFUSE_PGM_DATA6_REG	存放待烧写数据的第 6 个寄存器内容	0x0018	R/W
EFUSE_PGM_DATA7_REG	存放待烧写数据的第 7 个寄存器内容	0x001C	R/W
EFUSE_PGM_CHECK_VALUE0_REG	存放待烧写 RS 代码的第 0 个寄存器数据	0x0020	R/W
EFUSE_PGM_CHECK_VALUE1_REG	存放待烧写 RS 代码的第 1 个寄存器数据	0x0024	R/W
EFUSE_PGM_CHECK_VALUE2_REG	存放待烧写 RS 代码的第 2 个寄存器数据	0x0028	R/W
读取数据寄存器			
EFUSE_RD_WR_DIS_REG	BLOCK0 的第 0 个寄存器内容	0x002C	RO
EFUSE_RD_REPEAT_DATA0_REG	BLOCK0 的第 1 个寄存器内容	0x0030	RO
EFUSE_RD_REPEAT_DATA1_REG	BLOCK0 的第 2 个寄存器内容	0x0034	RO
EFUSE_RD_REPEAT_DATA2_REG	BLOCK0 的第 3 个寄存器内容	0x0038	RO
EFUSE_RD_REPEAT_DATA3_REG	BLOCK0 的第 4 个寄存器内容	0x003C	RO
EFUSE_RD_REPEAT_DATA4_REG	BLOCK0 的第 5 个寄存器内容	0x0040	RO
EFUSE_RD_MAC_SPI_SYS_0_REG	BLOCK1 的第 0 个寄存器内容	0x0044	RO
EFUSE_RD_MAC_SPI_SYS_1_REG	BLOCK1 的第 1 个寄存器内容	0x0048	RO
EFUSE_RD_MAC_SPI_SYS_2_REG	BLOCK1 的第 2 个寄存器内容	0x004C	RO
EFUSE_RD_MAC_SPI_SYS_3_REG	BLOCK1 的第 3 个寄存器内容	0x0050	RO
EFUSE_RD_MAC_SPI_SYS_4_REG	BLOCK1 的第 4 个寄存器内容	0x0054	RO
EFUSE_RD_MAC_SPI_SYS_5_REG	BLOCK1 的第 5 个寄存器内容	0x0058	RO
EFUSE_RD_SYS_PART1_DATA0_REG	BLOCK2 (system) 的第 0 个寄存器内容	0x005C	RO
EFUSE_RD_SYS_PART1_DATA1_REG	BLOCK2 (system) 的第 1 个寄存器内容	0x0060	RO
EFUSE_RD_SYS_PART1_DATA2_REG	BLOCK2 (system) 的第 2 个寄存器内容	0x0064	RO
EFUSE_RD_SYS_PART1_DATA3_REG	BLOCK2 (system) 的第 3 个寄存器内容	0x0068	RO
EFUSE_RD_SYS_PART1_DATA4_REG	BLOCK2 (system) 的第 4 个寄存器内容	0x006C	RO
EFUSE_RD_SYS_PART1_DATA5_REG	BLOCK2 (system) 的第 5 个寄存器内容	0x0070	RO
EFUSE_RD_SYS_PART1_DATA6_REG	BLOCK2 (system) 的第 6 个寄存器内容	0x0074	RO
EFUSE_RD_SYS_PART1_DATA7_REG	BLOCK2 (system) 的第 7 个寄存器内容	0x0078	RO
EFUSE_RD_USR_DATA0_REG	BLOCK3 (user) 的第 0 个寄存器内容	0x007C	RO
EFUSE_RD_USR_DATA1_REG	BLOCK3 (user) 的第 1 个寄存器内容	0x0080	RO
EFUSE_RD_USR_DATA2_REG	BLOCK3 (user) 的第 2 个寄存器内容	0x0084	RO
EFUSE_RD_USR_DATA3_REG	BLOCK3 (user) 的第 3 个寄存器内容	0x0088	RO
EFUSE_RD_USR_DATA4_REG	BLOCK3 (user) 的第 4 个寄存器内容	0x008C	RO

名称	描述	地址	访问
EFUSE_RD_USR_DATA5_REG	BLOCK3 (user) 的第 5 个寄存器内容	0x0090	RO
EFUSE_RD_USR_DATA6_REG	BLOCK3 (user) 的第 6 个寄存器内容	0x0094	RO
EFUSE_RD_USR_DATA7_REG	BLOCK3 (user) 的第 7 个寄存器内容	0x0098	RO
EFUSE_RD_KEY0_DATA0_REG	BLOCK4 (KEY0) 的第 0 个寄存器内容	0x009C	RO
EFUSE_RD_KEY0_DATA1_REG	BLOCK4 (KEY0) 的第 1 个寄存器内容	0x00A0	RO
EFUSE_RD_KEY0_DATA2_REG	BLOCK4 (KEY0) 的第 2 个寄存器内容	0x00A4	RO
EFUSE_RD_KEY0_DATA3_REG	BLOCK4 (KEY0) 的第 3 个寄存器内容	0x00A8	RO
EFUSE_RD_KEY0_DATA4_REG	BLOCK4 (KEY0) 的第 4 个寄存器内容	0x00AC	RO
EFUSE_RD_KEY0_DATA5_REG	BLOCK4 (KEY0) 的第 5 个寄存器内容	0x00B0	RO
EFUSE_RD_KEY0_DATA6_REG	BLOCK4 (KEY0) 的第 6 个寄存器内容	0x00B4	RO
EFUSE_RD_KEY0_DATA7_REG	BLOCK4 (KEY0) 的第 7 个寄存器内容	0x00B8	RO
EFUSE_RD_KEY1_DATA0_REG	BLOCK5 (KEY1) 的第 0 个寄存器内容	0x00BC	RO
EFUSE_RD_KEY1_DATA1_REG	BLOCK5 (KEY1) 的第 1 个寄存器内容	0x00C0	RO
EFUSE_RD_KEY1_DATA2_REG	BLOCK5 (KEY1) 的第 2 个寄存器内容	0x00C4	RO
EFUSE_RD_KEY1_DATA3_REG	BLOCK5 (KEY1) 的第 3 个寄存器内容	0x00C8	RO
EFUSE_RD_KEY1_DATA4_REG	BLOCK5 (KEY1) 的第 4 个寄存器内容	0x00CC	RO
EFUSE_RD_KEY1_DATA5_REG	BLOCK5 (KEY1) 的第 5 个寄存器内容	0x00D0	RO
EFUSE_RD_KEY1_DATA6_REG	BLOCK5 (KEY1) 的第 6 个寄存器内容	0x00D4	RO
EFUSE_RD_KEY1_DATA7_REG	BLOCK5 (KEY1) 的第 7 个寄存器内容	0x00D8	RO
EFUSE_RD_KEY2_DATA0_REG	BLOCK6 (KEY2) 的第 0 个寄存器内容	0x00DC	RO
EFUSE_RD_KEY2_DATA1_REG	BLOCK6 (KEY2) 的第 1 个寄存器内容	0x00E0	RO
EFUSE_RD_KEY2_DATA2_REG	BLOCK6 (KEY2) 的第 2 个寄存器内容	0x00E4	RO
EFUSE_RD_KEY2_DATA3_REG	BLOCK6 (KEY2) 的第 3 个寄存器内容	0x00E8	RO
EFUSE_RD_KEY2_DATA4_REG	BLOCK6 (KEY2) 的第 4 个寄存器内容	0x00EC	RO
EFUSE_RD_KEY2_DATA5_REG	BLOCK6 (KEY2) 的第 5 个寄存器内容	0x00F0	RO
EFUSE_RD_KEY2_DATA6_REG	BLOCK6 (KEY2) 的第 6 个寄存器内容	0x00F4	RO
EFUSE_RD_KEY2_DATA7_REG	BLOCK6 (KEY2) 的第 7 个寄存器内容	0x00F8	RO
EFUSE_RD_KEY3_DATA0_REG	BLOCK7 (KEY3) 的第 0 个寄存器内容	0x00FC	RO
EFUSE_RD_KEY3_DATA1_REG	BLOCK7 (KEY3) 的第 1 个寄存器内容	0x0100	RO
EFUSE_RD_KEY3_DATA2_REG	BLOCK7 (KEY3) 的第 2 个寄存器内容	0x0104	RO
EFUSE_RD_KEY3_DATA3_REG	BLOCK7 (KEY3) 的第 3 个寄存器内容	0x0108	RO
EFUSE_RD_KEY3_DATA4_REG	BLOCK7 (KEY3) 的第 4 个寄存器内容	0x010C	RO
EFUSE_RD_KEY3_DATA5_REG	BLOCK7 (KEY3) 的第 5 个寄存器内容	0x0110	RO
EFUSE_RD_KEY3_DATA6_REG	BLOCK7 (KEY3) 的第 6 个寄存器内容	0x0114	RO
EFUSE_RD_KEY3_DATA7_REG	BLOCK7 (KEY3) 的第 7 个寄存器内容	0x0118	RO
EFUSE_RD_KEY4_DATA0_REG	BLOCK8 (KEY4) 的第 0 个寄存器内容	0x011C	RO
EFUSE_RD_KEY4_DATA1_REG	BLOCK8 (KEY4) 的第 1 个寄存器内容	0x0120	RO
EFUSE_RD_KEY4_DATA2_REG	BLOCK8 (KEY4) 的第 2 个寄存器内容	0x0124	RO
EFUSE_RD_KEY4_DATA3_REG	BLOCK8 (KEY4) 的第 3 个寄存器内容	0x0128	RO
EFUSE_RD_KEY4_DATA4_REG	BLOCK8 (KEY4) 的第 4 个寄存器内容	0x012C	RO
EFUSE_RD_KEY4_DATA5_REG	BLOCK8 (KEY4) 的第 5 个寄存器内容	0x0130	RO
EFUSE_RD_KEY4_DATA6_REG	BLOCK8 (KEY4) 的第 6 个寄存器内容	0x0134	RO
EFUSE_RD_KEY4_DATA7_REG	BLOCK8 (KEY4) 的第 7 个寄存器内容	0x0138	RO

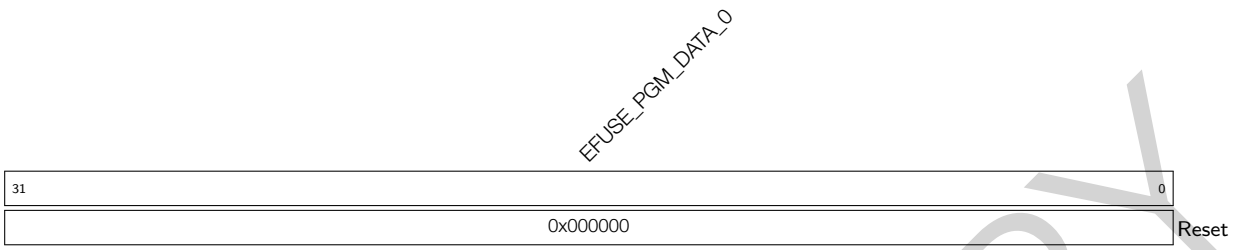
名称	描述	地址	访问
EFUSE_RD_KEY5_DATA0_REG	BLOCK9 (KEY5) 的第 0 个寄存器内容	0x013C	RO
EFUSE_RD_KEY5_DATA1_REG	BLOCK9 (KEY5) 的第 1 个寄存器内容	0x0140	RO
EFUSE_RD_KEY5_DATA2_REG	BLOCK9 (KEY5) 的第 2 个寄存器内容	0x0144	RO
EFUSE_RD_KEY5_DATA3_REG	BLOCK9 (KEY5) 的第 3 个寄存器内容	0x0148	RO
EFUSE_RD_KEY5_DATA4_REG	BLOCK9 (KEY5) 的第 4 个寄存器内容	0x014C	RO
EFUSE_RD_KEY5_DATA5_REG	BLOCK9 (KEY5) 的第 5 个寄存器内容	0x0150	RO
EFUSE_RD_KEY5_DATA6_REG	BLOCK9 (KEY5) 的第 6 个寄存器内容	0x0154	RO
EFUSE_RD_KEY5_DATA7_REG	BLOCK9 (KEY5) 的第 7 个寄存器内容	0x0158	RO
EFUSE_RD_SYS_PART2_DATA0_REG	BLOCK10 (system) 的第 0 个寄存器内容	0x015C	RO
EFUSE_RD_SYS_PART2_DATA1_REG	BLOCK10 (system) 的第 1 个寄存器内容	0x0160	RO
EFUSE_RD_SYS_PART2_DATA2_REG	BLOCK10 (system) 的第 2 个寄存器内容	0x0164	RO
EFUSE_RD_SYS_PART2_DATA3_REG	BLOCK10 (system) 的第 3 个寄存器内容	0x0168	RO
EFUSE_RD_SYS_PART2_DATA4_REG	BLOCK10 (system) 的第 4 个寄存器内容	0x016C	RO
EFUSE_RD_SYS_PART2_DATA5_REG	BLOCK10 (system) 的第 5 个寄存器内容	0x0170	RO
EFUSE_RD_SYS_PART2_DATA6_REG	BLOCK10 (system) 的第 6 个寄存器内容	0x0174	RO
EFUSE_RD_SYS_PART2_DATA7_REG	BLOCK10 (system) 的第 7 个寄存器内容	0x0178	RO
报告寄存器			
EFUSE_RD_REPEAT_ERR0_REG	BLOCK0 参数烧写错误记录第 0 个寄存器	0x017C	RO
EFUSE_RD_REPEAT_ERR1_REG	BLOCK0 参数烧写错误记录第 1 个寄存器	0x0180	RO
EFUSE_RD_REPEAT_ERR2_REG	BLOCK0 参数烧写错误记录第 2 个寄存器	0x0184	RO
EFUSE_RD_REPEAT_ERR3_REG	BLOCK0 参数烧写错误记录第 3 个寄存器	0x0188	RO
EFUSE_RD_REPEAT_ERR4_REG	BLOCK0 参数烧写错误记录第 4 个寄存器	0x0190	RO
EFUSE_RD_RS_ERR0_REG	记录 BLOCK1 ~ 10 参数烧写错误信息的第 0 个寄存器	0x01C0	RO
EFUSE_RD_RS_ERR1_REG	记录 BLOCK1 ~ 10 参数烧写错误信息的第 1 个寄存器	0x01C4	RO
配置寄存器			
EFUSE_CLK_REG	eFuse 时钟配置寄存器	0x01C8	R/W
EFUSE_CONF_REG	eFuse 运行模式配置寄存器	0x01CC	R/W
EFUSE_CMD_REG	eFuse 指令寄存器	0x01D4	varies
EFUSE_DAC_CONF_REG	eFuse 烧写电压控制寄存器	0x01E8	R/W
EFUSE_RD_TIM_CONF_REG	eFuse 读取时序参数配置寄存器	0x01EC	R/W
EFUSE_WR_TIM_CONF1_REG	eFuse 烧写时序参数第 1 个配置寄存器	0x01F4	R/W
EFUSE_WR_TIM_CONF2_REG	eFuse 烧写时序参数第 2 个配置寄存器	0x01F8	R/W
状态寄存器			
EFUSE_STATUS_REG	eFuse 状态寄存器	0x01D0	RO
中断寄存器			
EFUSE_INT_RAW_REG	eFuse 原始中断寄存器	0x01D8	R/WC/SS
EFUSE_INT_ST_REG	eFuse 中断状态寄存器	0x01DC	RO
EFUSE_INT_ENA_REG	eFuse 中断使能寄存器	0x01E0	R/W
EFUSE_INT_CLR_REG	eFuse 中断清除寄存器	0x01E4	WO

名称	描述	地址	访问
版本寄存器			
EFUSE_DATE_REG	版本控制寄存器	0x01FC	R/W

4.5 寄存器

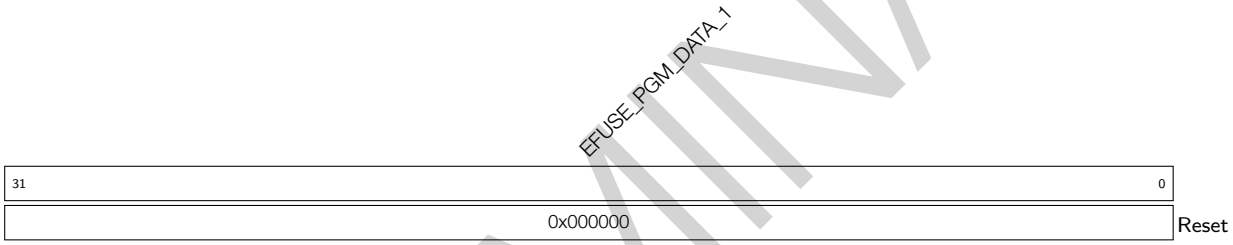
本小节的所有地址均为相对于 eFuse 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 3 *系统和存储器* 中的表 3-4。

Register 4.1. EFUSE_PGM_DATA0_REG (0x0000)



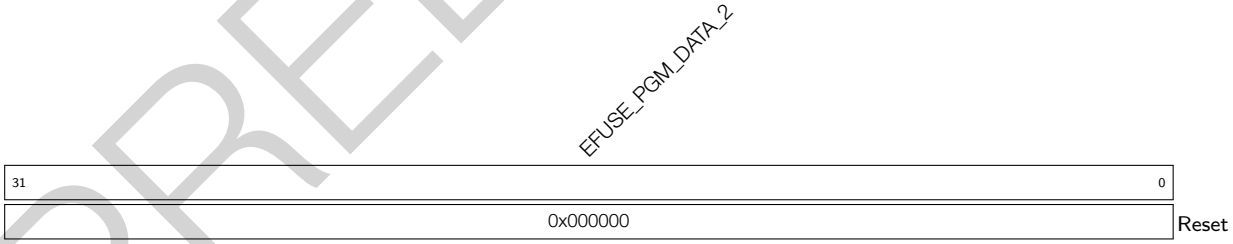
EFUSE_PGM_DATA_0 存放待烧写数据的第 0 个 32 位数据内容。(读/写)

Register 4.2. EFUSE_PGM_DATA1_REG (0x0004)



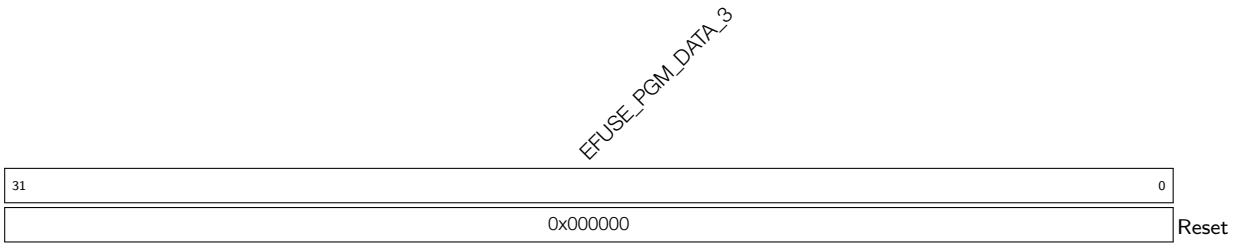
EFUSE_PGM_DATA_1 存放待烧写数据的第 1 个 32 位数据内容。(读/写)

Register 4.3. EFUSE_PGM_DATA2_REG (0x0008)



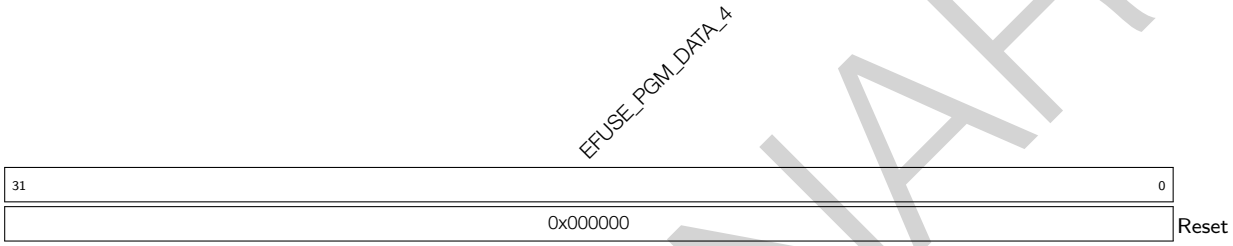
EFUSE_PGM_DATA_2 存放待烧写数据的第 2 个 32 位数据内容。(读/写)

Register 4.4. EFUSE_PGM_DATA3_REG (0x000C)



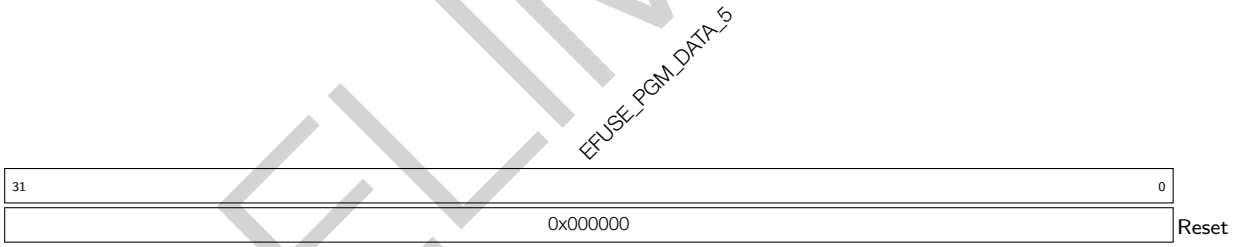
EFUSE_PGM_DATA_3 存放待烧写数据的第 3 个 32 位数据内容。(读/写)

Register 4.5. EFUSE_PGM_DATA4_REG (0x0010)



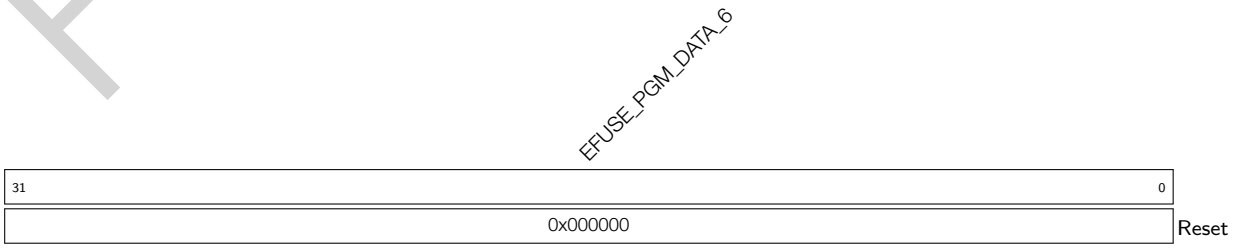
EFUSE_PGM_DATA_4 存放待烧写数据的第 4 个 32 位数据内容。(读/写)

Register 4.6. EFUSE_PGM_DATA5_REG (0x0014)



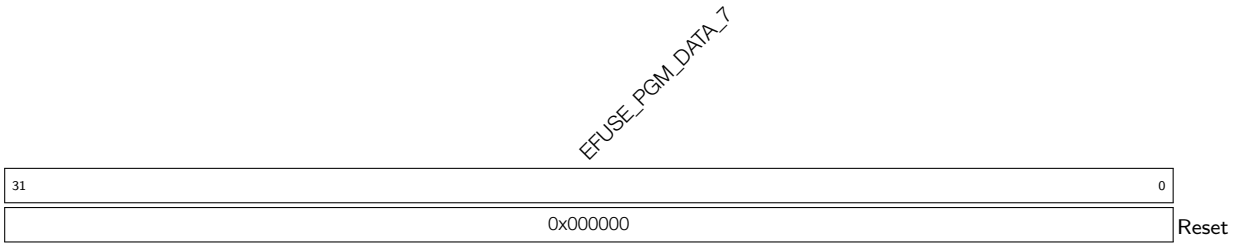
EFUSE_PGM_DATA_5 存放待烧写数据的第 5 个 32 位数据内容。(读/写)

Register 4.7. EFUSE_PGM_DATA6_REG (0x0018)



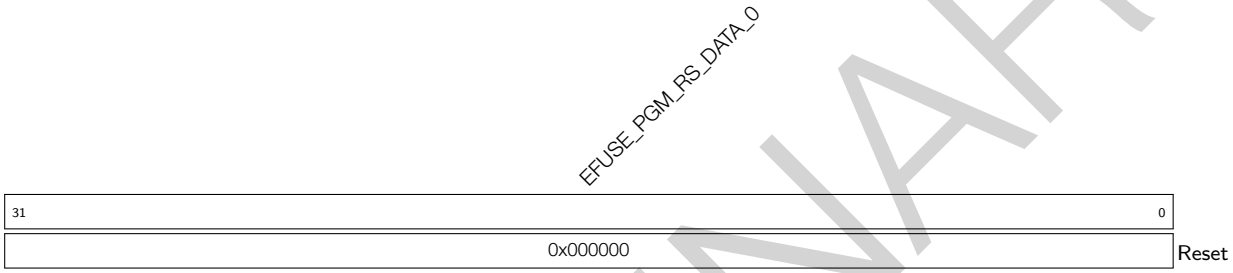
EFUSE_PGM_DATA_6 存放待烧写数据的第 6 个 32 位数据内容。(读/写)

Register 4.8. EFUSE_PGM_DATA7_REG (0x001C)



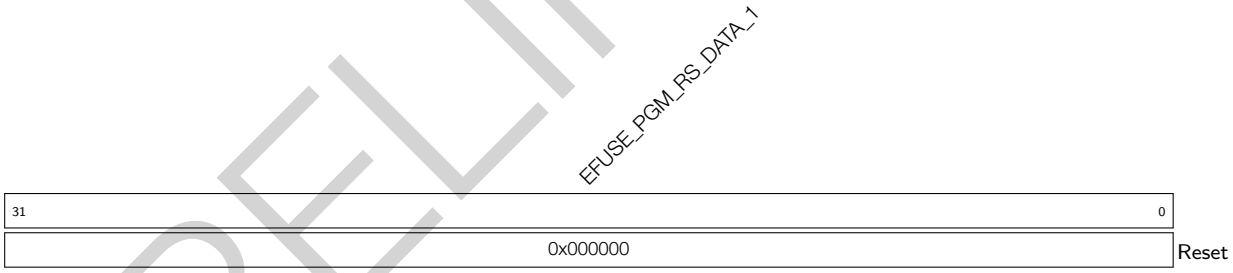
EFUSE_PGM_DATA_7 存放待烧写数据的第 7 个 32 位数据内容。(读/写)

Register 4.9. EFUSE_PGM_CHECK_VALUE0_REG (0x0020)



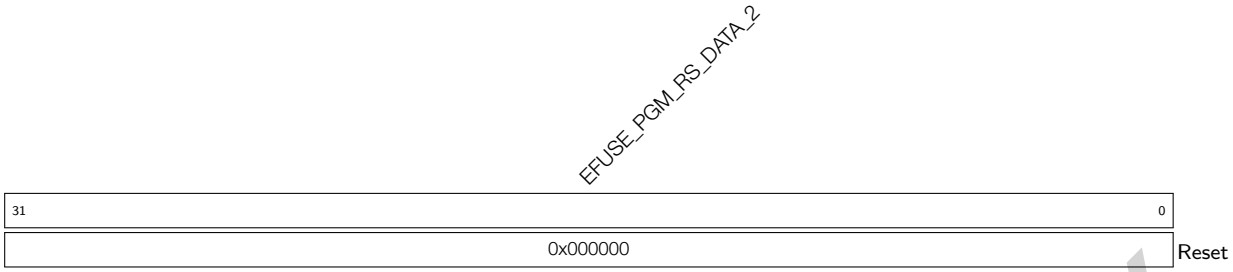
EFUSE_PGM_RS_DATA_0 存放待烧写 RS 代码的第 0 个 32 位数据内容。(读/写)

Register 4.10. EFUSE_PGM_CHECK_VALUE1_REG (0x0024)



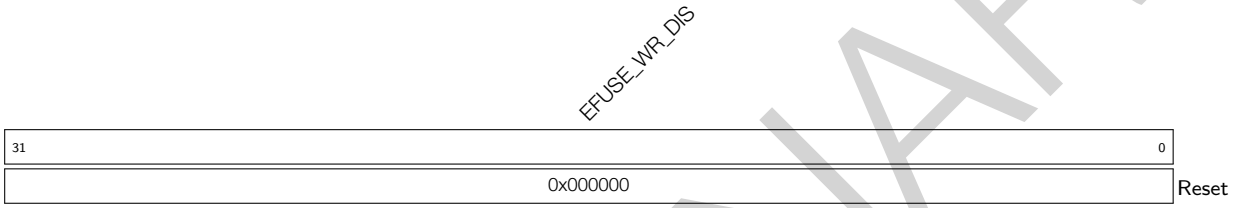
EFUSE_PGM_RS_DATA_1 存放待烧写 RS 代码的第 1 个 32 位数据内容。(读/写)

Register 4.11. EFUSE_PGM_CHECK_VALUE2_REG (0x0028)



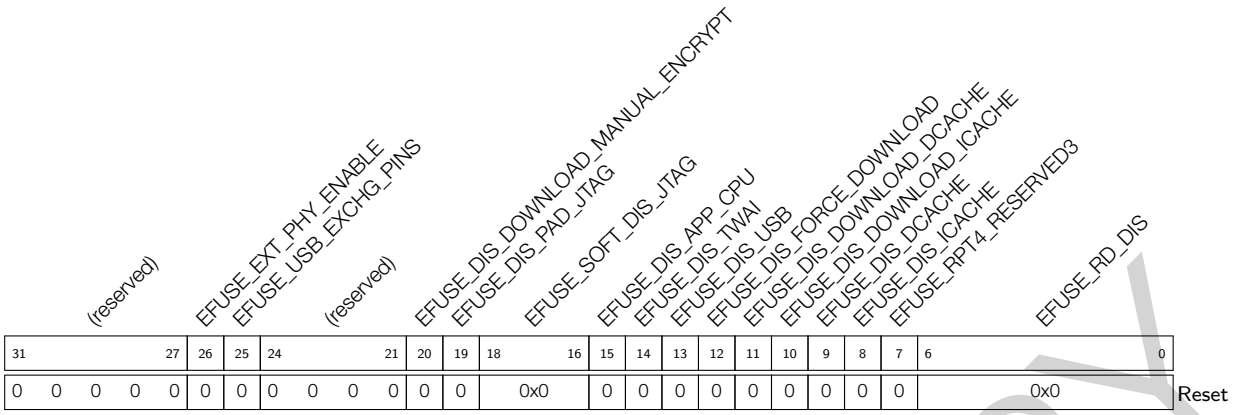
EFUSE_PGM_RS_DATA_2 存放待烧写 RS 代码的第 2 个 32 位数据内容。(读/写)

Register 4.12. EFUSE_RD_WR_DIS_REG (0x002C)



EFUSE_WR_DIS 置位禁用 eFuse 烧写。(只读)

Register 4.13. EFUSE_RD_REPEAT_DATA0_REG (0x0030)



EFUSE_RD_DIS 置位禁止软件读取 eFuse Block4 ~ 10 的内容。(只读)

EFUSE_RPT4_RESERVED3 保留（采用 4 备份编码）。(只读)

EFUSE_DIS_ICACHE 置位禁用 Icache。(只读)

EFUSE_DIS_DCACHE 置位禁用 Dcache。(只读)

EFUSE_DIS_DOWNLOAD_ICACHE 置位在下载模式下关闭 Icache（boot_mode[3:0] 为 0, 1, 2, 3, 6, 7）。(只读)

EFUSE_DIS_DOWNLOAD_DCACHE 置位在下载模式下关闭 Dcache（boot_mode[3:0] 为 0, 1, 2, 3, 6, 7）。(只读)

EFUSE_DIS_FORCE_DOWNLOAD 置位禁止强制芯片进入下载模式。(只读)

EFUSE_DIS_USB 置位关闭 USB 功能。(只读)

EFUSE_DIS_TWAI 置位关闭 TWAI 功能。(只读)

EFUSE_DIS_APP_CPU 置位禁止启用 app cpu。(只读)

EFUSE_SOFT_DIS_JTAG 软关断 JTAG 功能 (奇数个比特值为 1 表示关断)，用户还可以通过 HAMC 模块再次打开 JTAG。(只读)

EFUSE_DIS_PAD_JTAG 硬关断 JTAG 功能，永久关断。(只读)

EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT 置位在 download boot 模式下关闭 flash 加密功能。(只读)

EFUSE_USB_EXCHG_PINS 置位交换 USB D+ 和 D- 管脚。(只读)

EFUSE_EXT_PHY_ENABLE 置位使能外部 USB PHY。(只读)

Register 4.14. EFUSE_RD_REPEAT_DATA1_REG (0x0034)

EFUSE_VDD_SPI_XPD 置位控制 SPI 调节器上电。(只读)

EFUSE_VDD_SPI_TIEH 置位 SPI 调节器短接至 VDD3P3_RTC_IO。(只读)

EFUSE_VDD_SPI_FORCE 置位强制使用 eFuse 配置 VDD_SPI。(只读)

EFUSE_WDT_DELAY_SEL 选择 RTC 看门狗超时阈值，单位为慢速时钟周期。00: 40,000 个慢速时钟周期；01: 80,000 个慢速时钟周期；10: 160,000 个慢速时钟周期；11: 320,000 个慢速时钟周期。(只读)

EFUSE_SPI_BOOT_CRYPT_CNT 置位使能 SPI boot 加解密。奇数个 1: 使能；偶数个 1: 禁用。(只读)

EFUSE_SECURE_BOOT_KEY_REVOKE0 置位使能撤销第一个安全启动密钥。(只读)

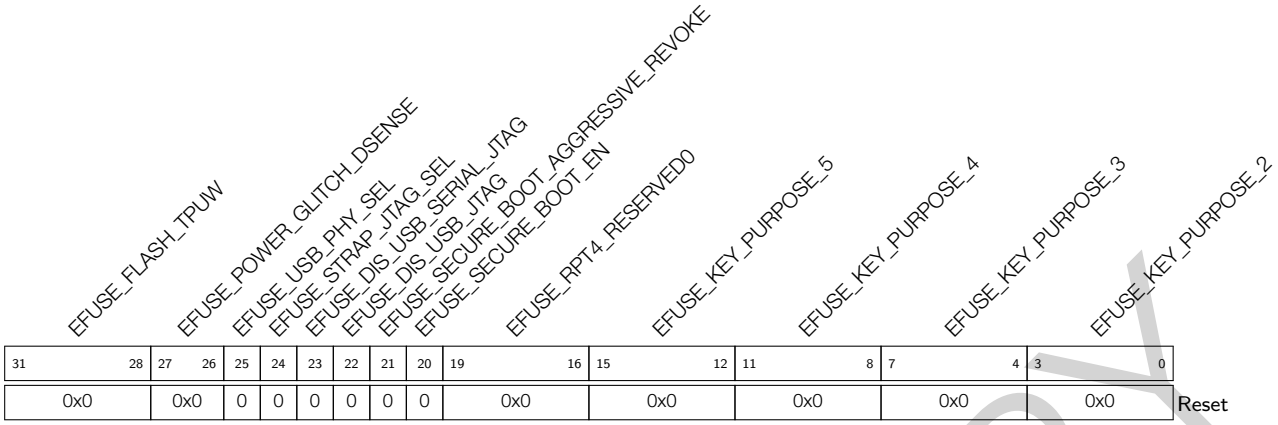
EFUSE_SECURE_BOOT_KEY_REVOKE1 置位使能撤销第二个安全启动密钥。(只读)

EFUSE_SECURE_BOOT_KEY_REVOKE2 置位使能撤销第三个安全启动密钥。(只读)

EFUSE_KEY_PURPOSE_0 Key0 用途。(只读)

EFUSE_KEY_PURPOSE_1 Key1 用途。(只读)

Register 4.15. EFUSE_RD_REPEAT_DATA2_REG (0x0038)



EFUSE_KEY_PURPOSE_2 Key2 用途。(只读)

EFUSE_KEY_PURPOSE_3 Key3 用途。(只读)

EFUSE_KEY_PURPOSE_4 Key4 用途。(只读)

EFUSE_KEY_PURPOSE_5 Key5 用途。(只读)

EFUSE_RPT4_RESERVED0 保留（采用 4 备份编码)。(只读)

EFUSE_SECURE_BOOT_EN 置位使能安全启动。(只读)

EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE 置位使能密钥失效的激进策略。(只读)

EFUSE_DIS_USB_JTAG 置位禁用 usb_serial_jtag 模块的 usb 转 jtag 功能。(只读)

EFUSE_DIS_USB_SETIAL_JTAG 置位禁能 usb_serial_jtag 模块。(只读)

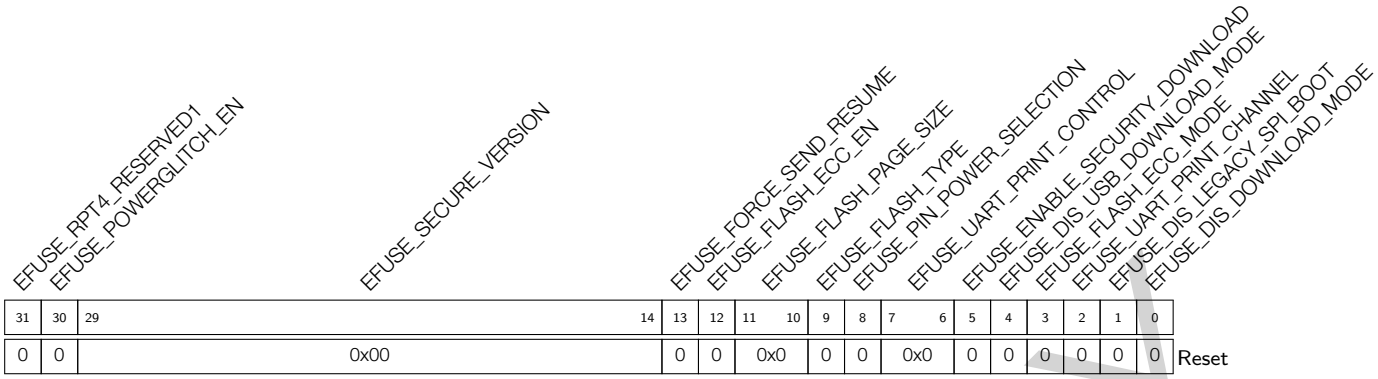
EFUSE_STRAP_JTAG_SEL 当 reg_dis_usb_jtag 和 reg_dis_pad_jtag 都为 0 时，置位使能使用 strapping gpio10 选择 usb_to_jtag 或 pad_to_jtag 的功能。(只读)

EFUSE_USB_PHY_SEL 切换 USB OTG 和 USB 设备使用内部 PHY 还是外部 PHY。0: USB 设备使用内部 PHY，USB OTG 使用外部 PHY；1: USB OTG 使用内部 PHY，USB 设备使用外部 PHY。(只读)

EFUSE_POWER_GLITCH_DSENSE 配置电压毛刺的采样延迟。(只读)

EFUSE_FLASH_TPUW 配置上电后 flash 等待时间，单位为 ms。该值小于 15 时，等待时间为配置的值；该值大于等于 15 时，等待时间为配置的时间的 2 倍。(只读)

Register 4.16. EFUSE_RD_REPEAT_DATA3_REG (0x003C)



EFUSE_DIS_DOWNLOAD_MODE 置位关闭下载模式 (boot_mode[3:0] = 0, 1, 2, 3, 6, 7)。(只读)

EFUSE_DIS_LEGACY_SPI_BOOT 置位关闭 Legacy SPI boot 模式 (boot_mode[3:0] = 4)。(只读)

EFUSE_UART_PRINT_CHANNEL 选择打印 boot 信息的 UART 通道。0: UART0; 1: UART1。(只读)

EFUSE_FLASH_ECC_MODE 置位配置 ROM 中 flash ECC 模式。0: 使能 16-to-18 字节模式; 1: 使能 16-to-17 字节模式。(只读)

EFUSE_DIS_USB_DOWNLOAD_MODE 置位在 UART download boot 模式下关闭 USB 功能。(只读)

EFUSE_ENABLE_SECURITY_DOWNLOAD 置位使能安全 UART 下载模式。(只读)

EFUSE_UART_PRINT_CONTROL 控制 UART boot 信息的默认打印方式。00: 使能打印; 01: GPIO8 低电平复位时，使能打印; 10: GPIO8 高电平复位时，使能打印; 11: 关闭打印。(只读)

EFUSE_PIN_POWER_SELECTION ROM 模式时选择 GPIO33 ~ GPIO37 的电源。0: VDD3P3_CPU; 1: VDD_SPI. (只读)

EFUSE_FLASH_TYPE 配置 SPI flash 的最大行数。0: 4 行; 1: 8 行。(只读)

EFUSE_FLASH_PAGE_SIZE 配置 flash 的页大小。(只读)

EFUSE_FLASH_ECC_EN 置位在 flash boot 中启用 ECC 功能。(只读)

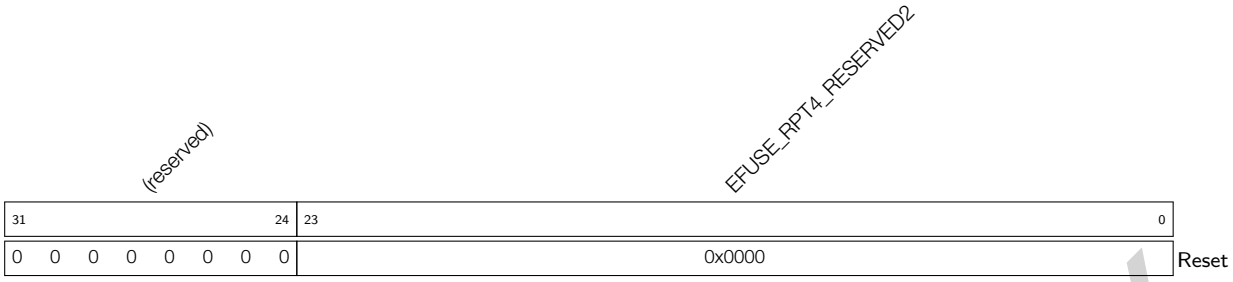
EFUSE_FORCE_SEND_RESUME 置位强制 ROM 代码在 SPI 启动过程中发送恢复指令。(只读)

EFUSE_SECURE_VERSION 表明 IDF 安全版本 (用于 ESP-IDF 的防回滚功能)。(只读)

EFUSE_POWERGLITCH_EN 置位使能电压毛刺功能。(只读)

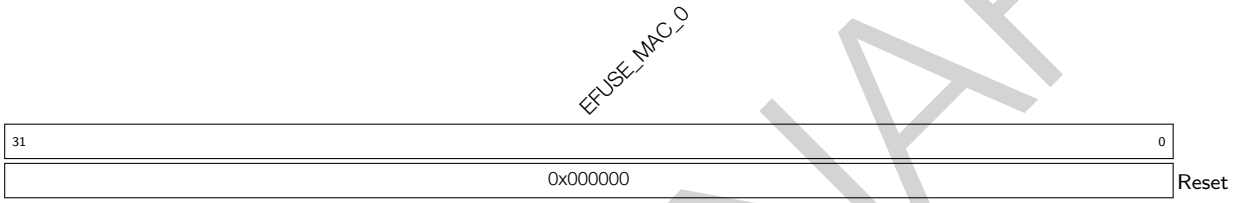
EFUSE_RPT4_RESERVED1 保留 (采用 4 备份编码)。(只读)

Register 4.17. EFUSE_RD_REPEAT_DATA4_REG (0x0040)



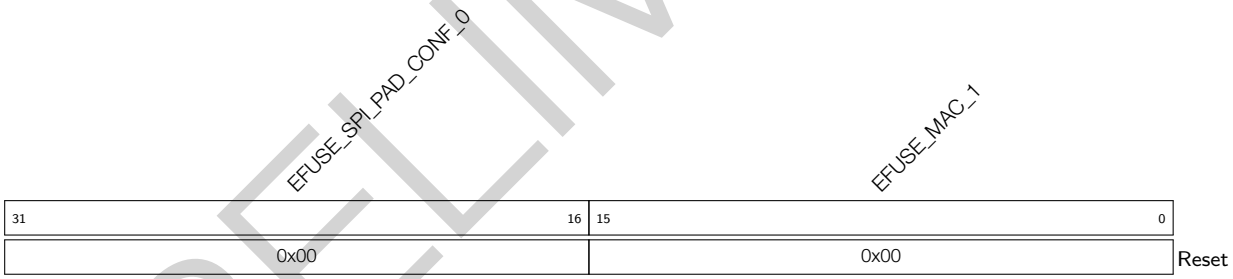
EFUSE_RPT4_RESERVED2 保留（采用 4 备份编码）。(只读)

Register 4.18. EFUSE_RD_MAC_SPI_SYS_0_REG (0x0044)



EFUSE_MAC_0 存储 MAC 地址低 32 位的内容。(只读)

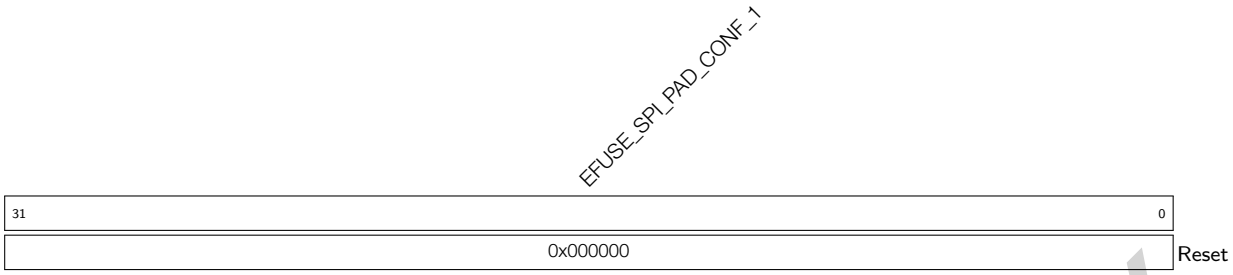
Register 4.19. EFUSE_RD_MAC_SPI_SYS_1_REG (0x0048)



EFUSE_MAC_1 存储 MAC 地址高 16 位的内容。(只读)

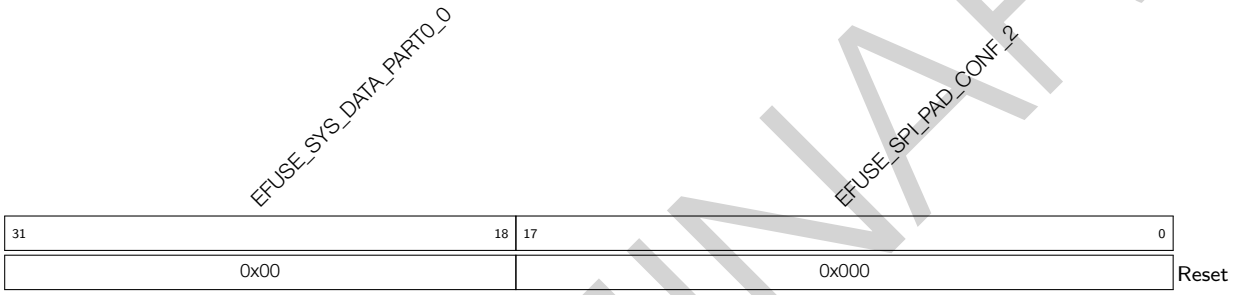
EFUSE_SPI_PAD_CONF_0 存储 SPI_PAD_CONF 第 0 部分的内容。(只读)

Register 4.20. EFUSE_RD_MAC_SPI_SYS_2_REG (0x004C)



EFUSE_SPI_PAD_CONF_1 存储 SPI_PAD_CONF 第 1 部分的内容。(只读)

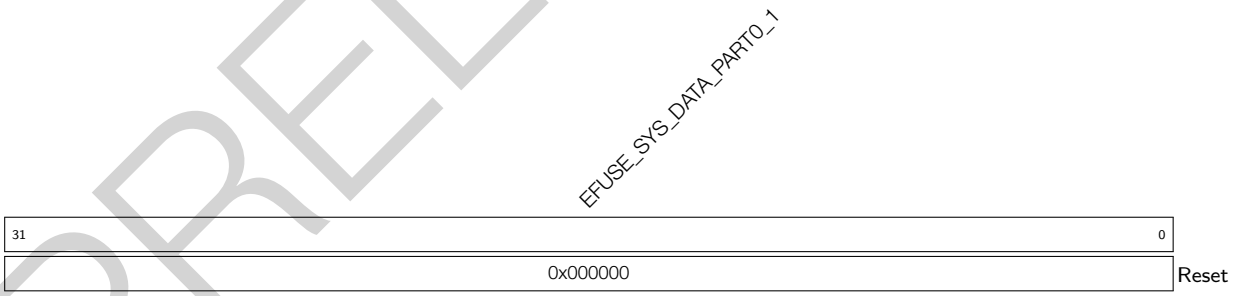
Register 4.21. EFUSE_RD_MAC_SPI_SYS_3_REG (0x0050)



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

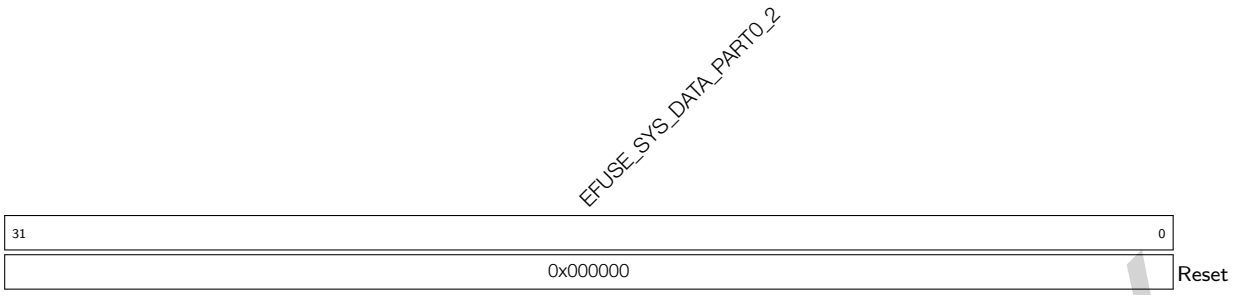
EFUSE_SYS_DATA_PART0_0 存储系统数据第 0 部分的第 1 个 14 位内容。(只读)

Register 4.22. EFUSE_RD_MAC_SPI_SYS_4_REG (0x0054)



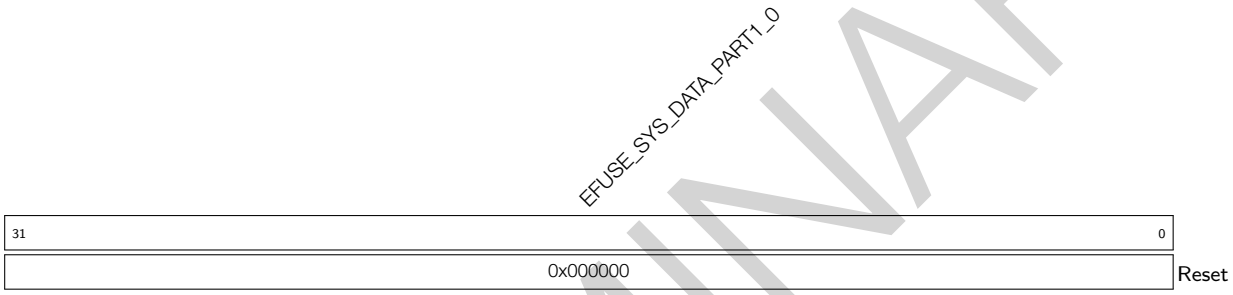
EFUSE_SYS_DATA_PART0_1 存储系统数据第 0 部分的第 1 个 32 位内容。(只读)

Register 4.23. EFUSE_RD_MAC_SPI_SYS_5_REG (0x0058)



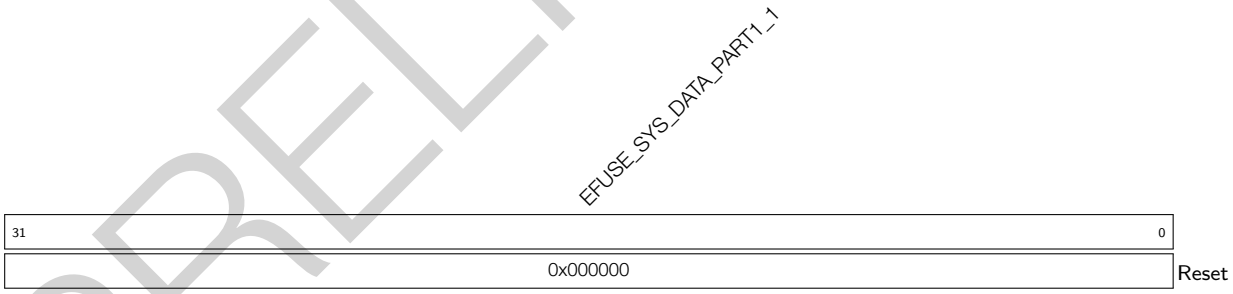
EFUSE_SYS_DATA_PART0_2 存储系统数据第 0 部分的第 2 个 32 位内容。(只读)

Register 4.24. EFUSE_RD_SYS_PART1_DATA0_REG (0x005C)



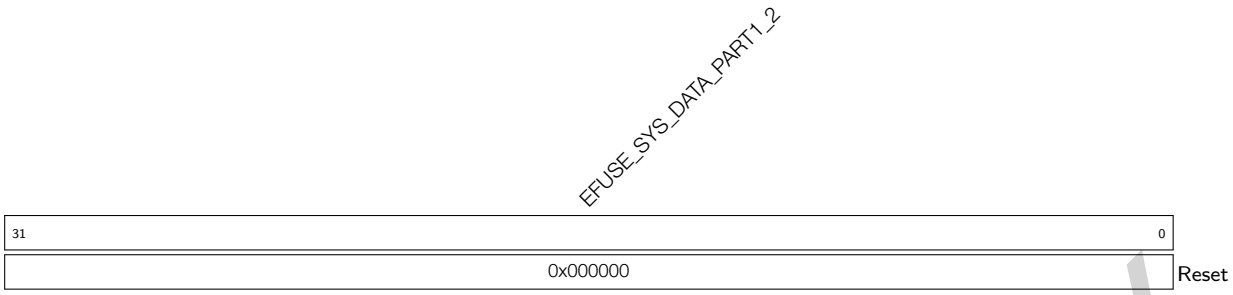
EFUSE_SYS_DATA_PART1_0 存储系统数据第 1 部分的第 0 个 32 位的内容。(只读)

Register 4.25. EFUSE_RD_SYS_PART1_DATA1_REG (0x0060)



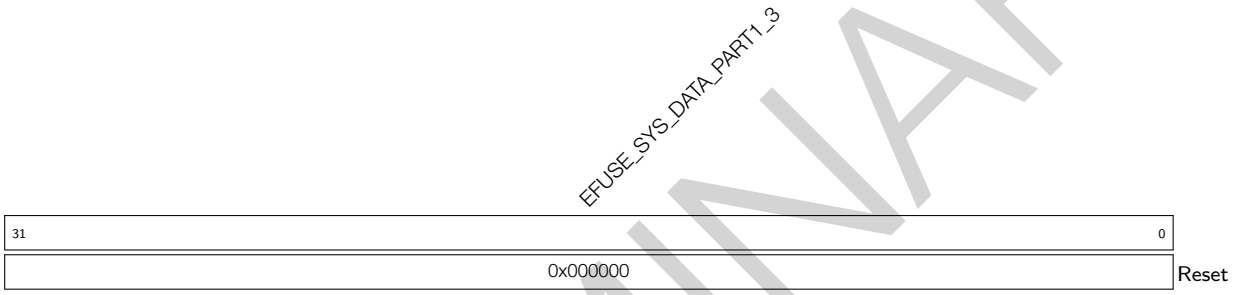
EFUSE_SYS_DATA_PART1_1 存储系统数据第 1 部分的第 1 个 32 位内容。(只读)

Register 4.26. EFUSE_RD_SYS_PART1_DATA2_REG (0x0064)



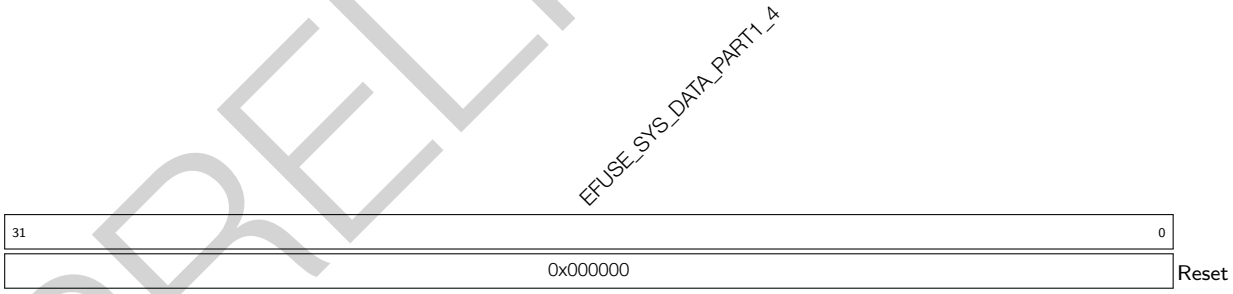
EFUSE_SYS_DATA_PART1_2 存储系统数据第 1 部分的第 2 个 32 位内容。(只读)

Register 4.27. EFUSE_RD_SYS_PART1_DATA3_REG (0x0068)



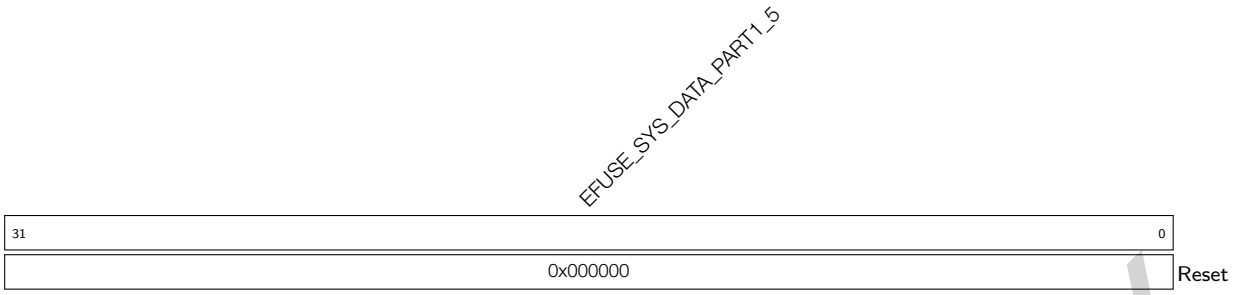
EFUSE_SYS_DATA_PART1_3 存储系统数据第 1 部分的第 3 个 32 位内容。(只读)

Register 4.28. EFUSE_RD_SYS_PART1_DATA4_REG (0x006C)



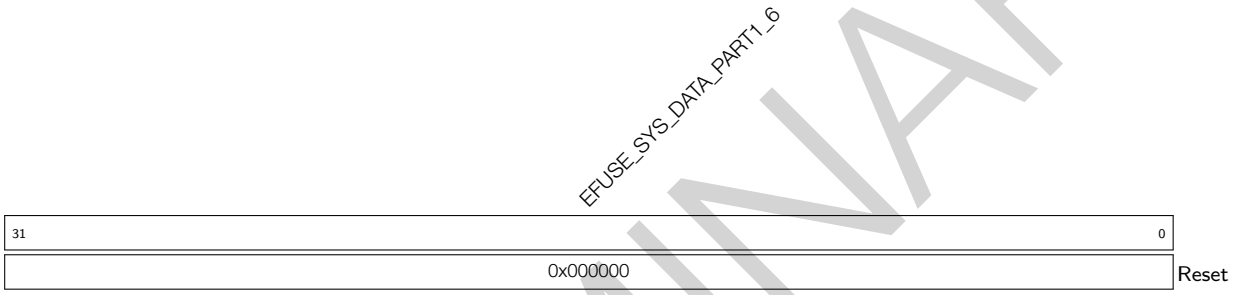
EFUSE_SYS_DATA_PART1_4 存储系统数据第 1 部分的第 4 个 32 位内容。(只读)

Register 4.29. EFUSE_RD_SYS_PART1_DATA5_REG (0x0070)



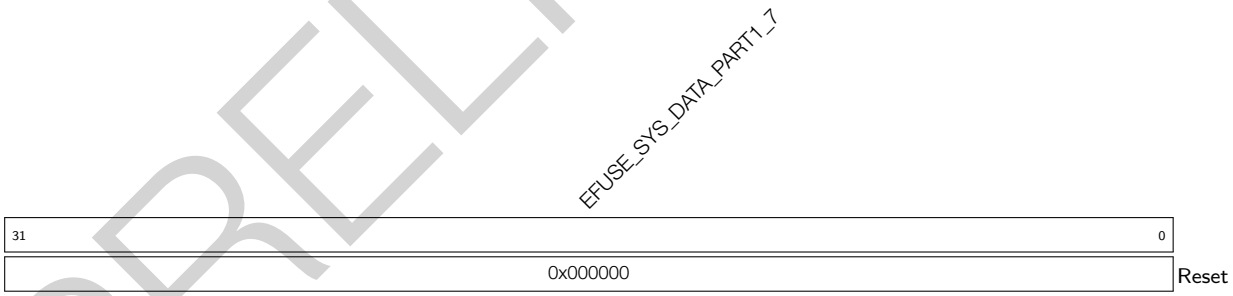
EFUSE_SYS_DATA_PART1_5 存储系统数据第 1 部分的第 5 个 32 位内容。(只读)

Register 4.30. EFUSE_RD_SYS_PART1_DATA6_REG (0x0074)



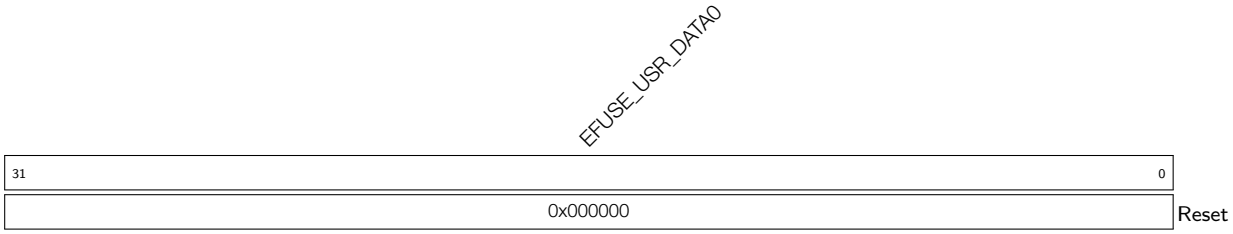
EFUSE_SYS_DATA_PART1_6 存储系统数据第 1 部分的第 6 个 32 位内容。(只读)

Register 4.31. EFUSE_RD_SYS_PART1_DATA7_REG (0x0078)



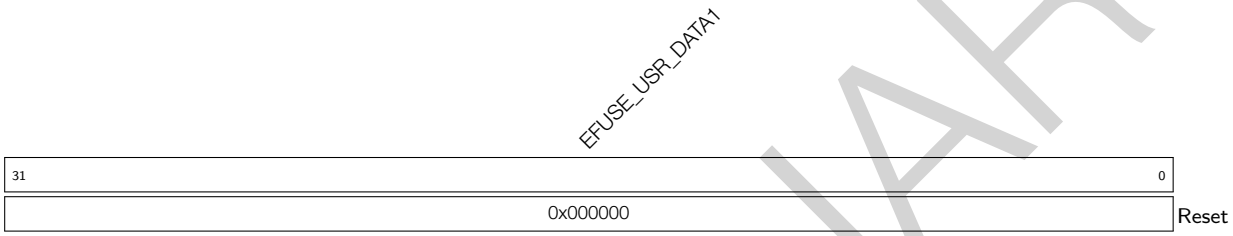
EFUSE_SYS_DATA_PART1_7 存储系统数据第 1 部分的第 7 个 32 位内容。(只读)

Register 4.32. EFUSE_RD_USR_DATA0_REG (0x007C)



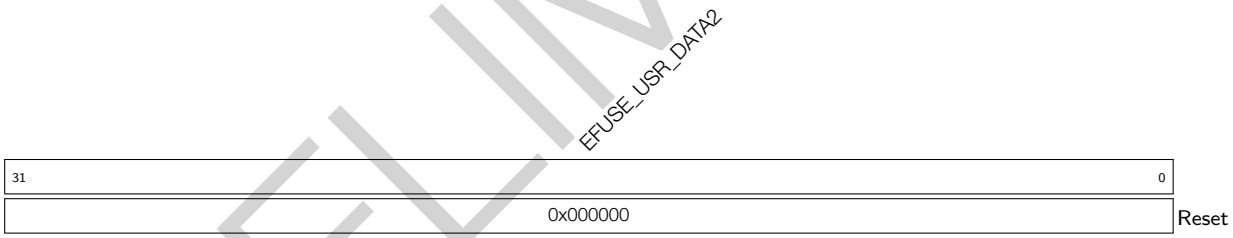
EFUSE_USR_DATA0 存储 BLOCK3 (user) 第 0 个 32 位内容。(只读)

Register 4.33. EFUSE_RD_USR_DATA1_REG (0x0080)



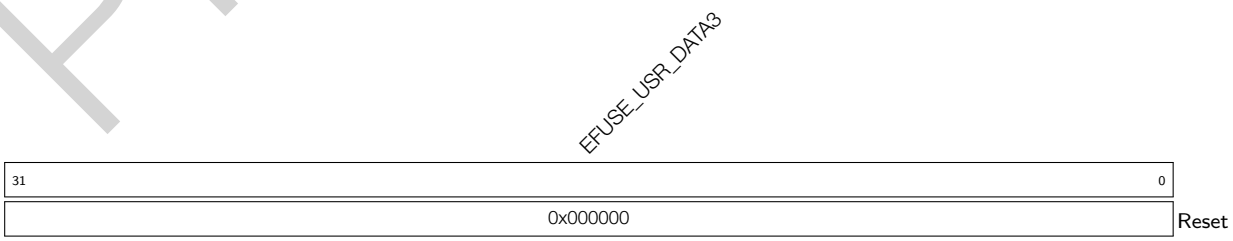
EFUSE_USR_DATA1 存储 BLOCK3 (user) 第 1 个 32 位内容。(只读)

Register 4.34. EFUSE_RD_USR_DATA2_REG (0x0084)



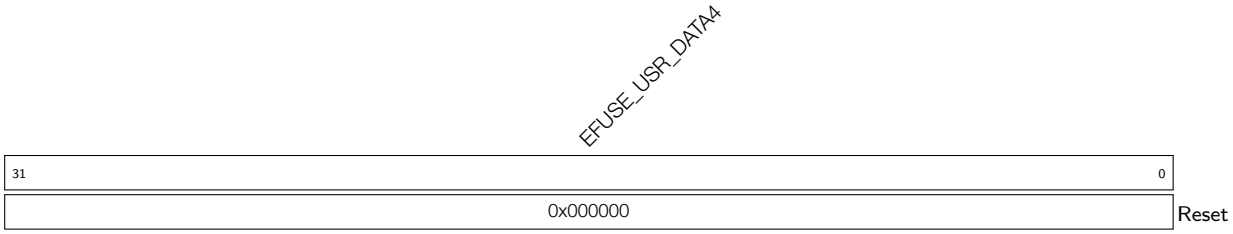
EFUSE_USR_DATA2 存储 BLOCK3 (user) 第 2 个 32 位内容。(只读)

Register 4.35. EFUSE_RD_USR_DATA3_REG (0x0088)



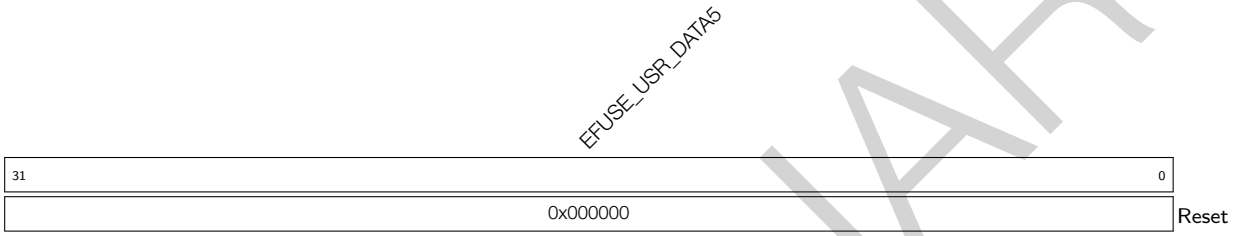
EFUSE_USR_DATA3 存储 BLOCK3 (user) 第 3 个 32 位内容。(只读)

Register 4.36. EFUSE_RD_USR_DATA4_REG (0x008C)



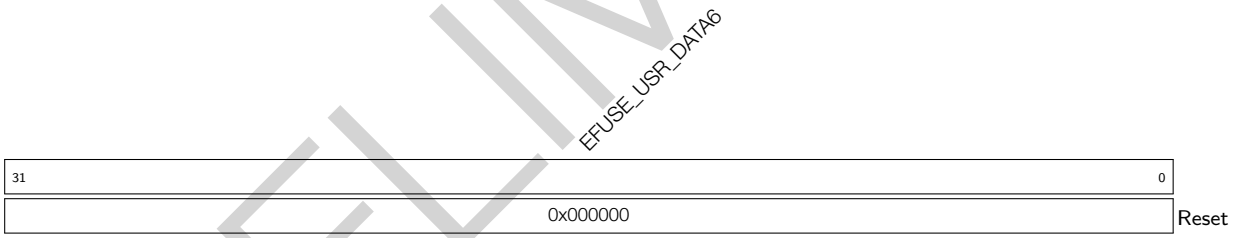
EFUSE_USR_DATA4 存储 BLOCK3 (user) 第 4 个 32 位内容。(只读)

Register 4.37. EFUSE_RD_USR_DATA5_REG (0x0090)



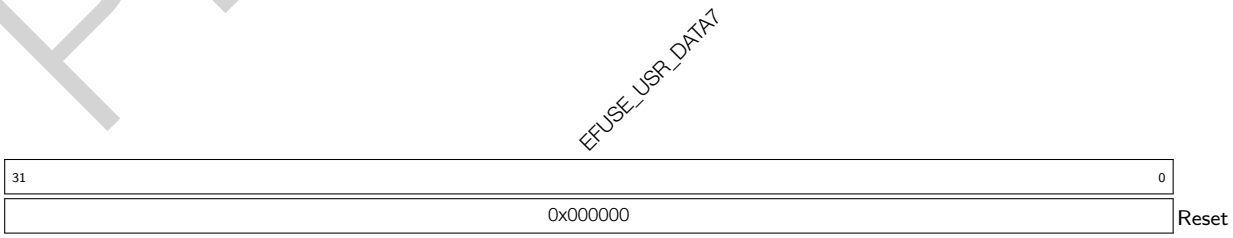
EFUSE_USR_DATA5 存储 BLOCK3 (user) 第 5 个 32 位内容。(只读)

Register 4.38. EFUSE_RD_USR_DATA6_REG (0x0094)



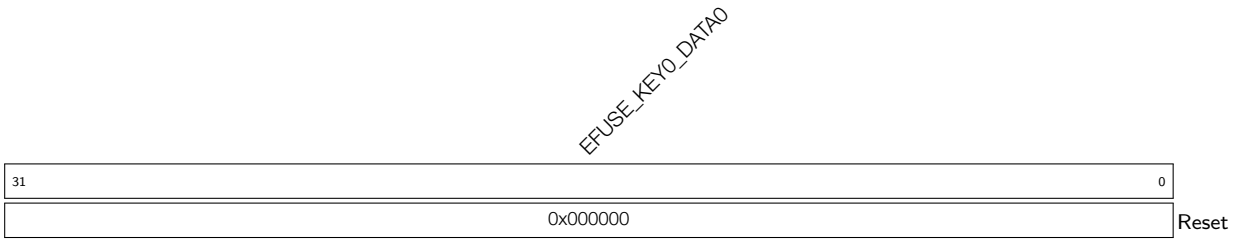
EFUSE_USR_DATA6 存储 BLOCK3 (user) 第 6 个 32 位内容。(只读)

Register 4.39. EFUSE_RD_USR_DATA7_REG (0x0098)



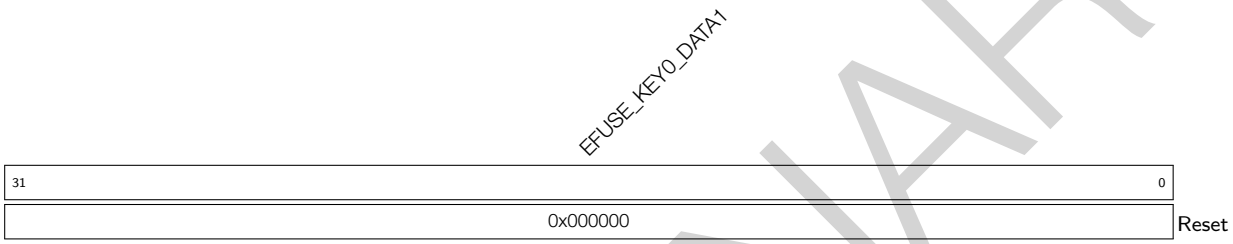
EFUSE_USR_DATA7 存储 BLOCK3 (user) 第 7 个 32 位内容。(只读)

Register 4.40. EFUSE_RD_KEY0_DATA0_REG (0x009C)



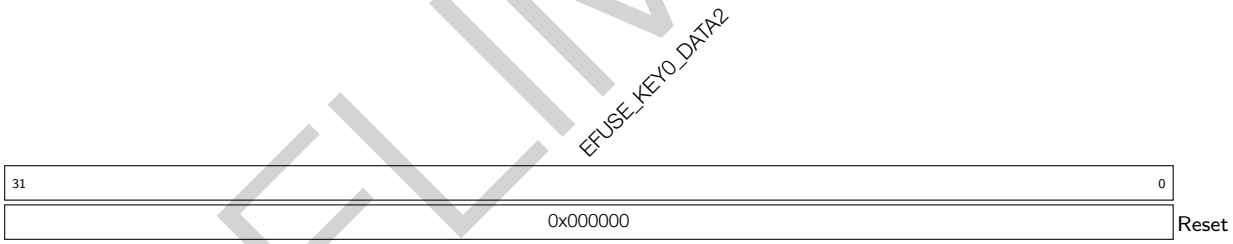
EFUSE_KEY0_DATA0 存储 KEY0 第 0 个 32 位内容。(只读)

Register 4.41. EFUSE_RD_KEY0_DATA1_REG (0x00A0)



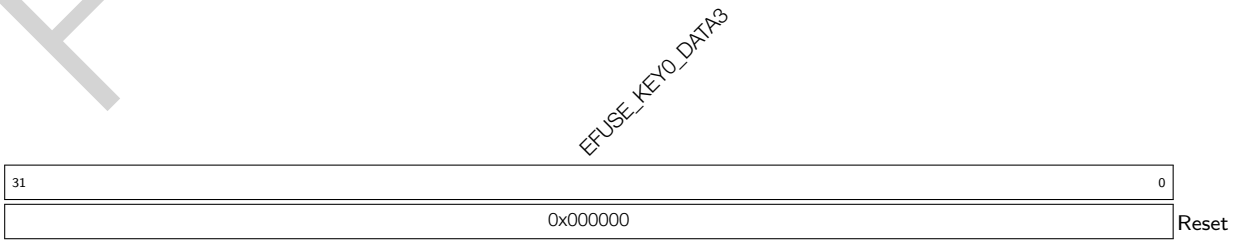
EFUSE_KEY0_DATA1 存储 KEY0 第 1 个 32 位内容。(只读)

Register 4.42. EFUSE_RD_KEY0_DATA2_REG (0x00A4)



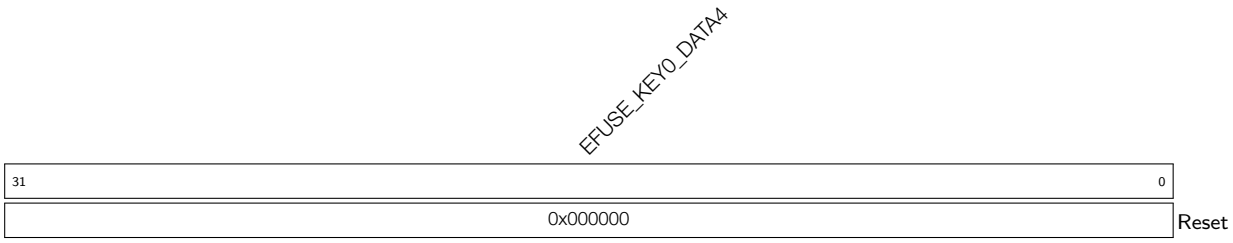
EFUSE_KEY0_DATA2 存储 KEY0 第 2 个 32 位内容。(只读)

Register 4.43. EFUSE_RD_KEY0_DATA3_REG (0x00A8)



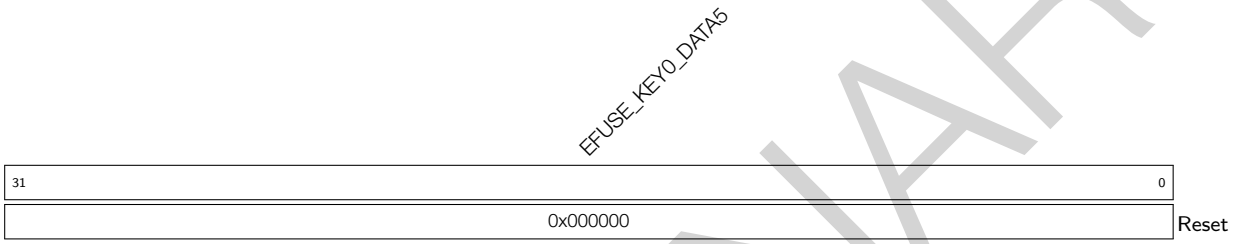
EFUSE_KEY0_DATA3 存储 KEY0 第 3 个 32 位内容。(只读)

Register 4.44. EFUSE_RD_KEY0_DATA4_REG (0x00AC)



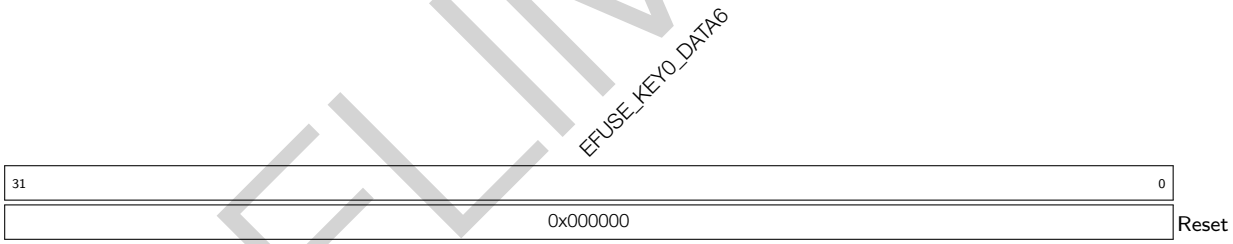
EFUSE_KEY0_DATA4 存储 KEY0 第 4 个 32 位内容。(只读)

Register 4.45. EFUSE_RD_KEY0_DATA5_REG (0x00B0)



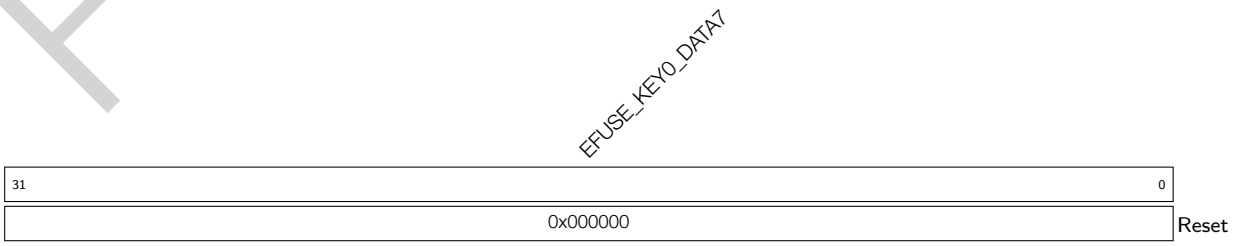
EFUSE_KEY0_DATA5 存储 KEY0 第 5 个 32 位内容。(只读)

Register 4.46. EFUSE_RD_KEY0_DATA6_REG (0x00B4)



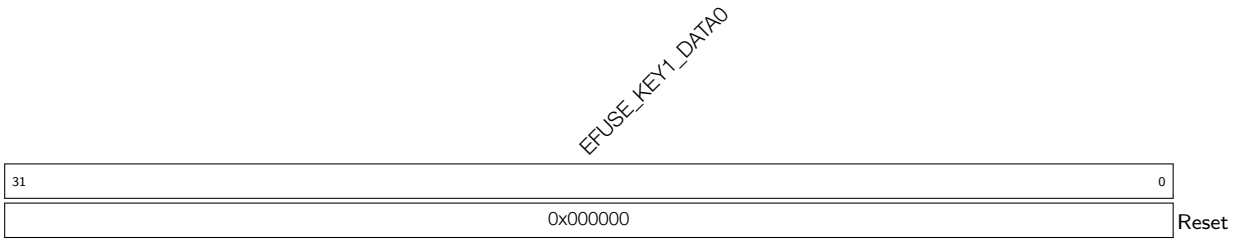
EFUSE_KEY0_DATA6 存储 KEY0 第 6 个 32 位内容。(只读)

Register 4.47. EFUSE_RD_KEY0_DATA7_REG (0x00B8)



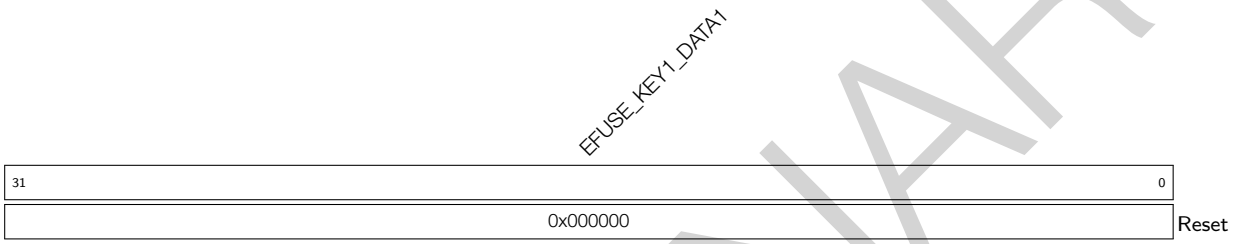
EFUSE_KEY0_DATA7 存储 KEY0 第 7 个 32 位内容。(只读)

Register 4.48. EFUSE_RD_KEY1_DATA0_REG (0x00BC)



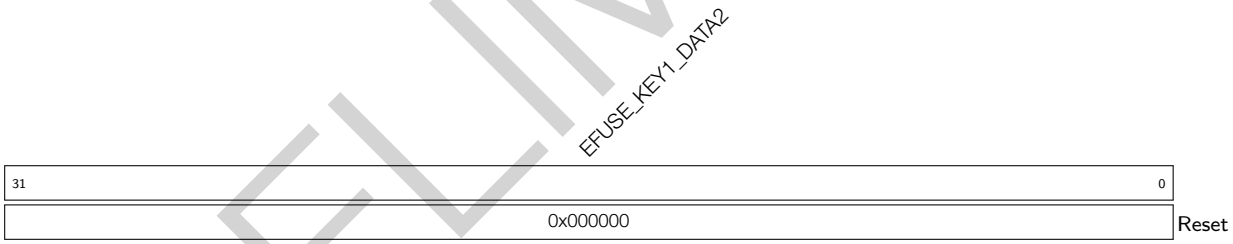
EFUSE_KEY1_DATA0 存储 KEY1 第 0 个 32 位内容。(只读)

Register 4.49. EFUSE_RD_KEY1_DATA1_REG (0x00C0)



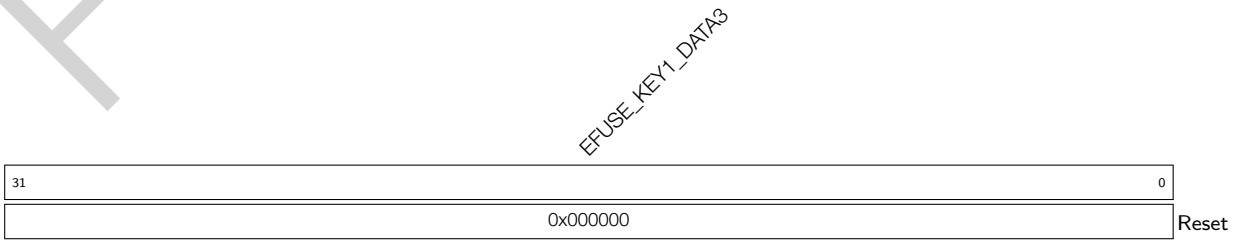
EFUSE_KEY1_DATA1 存储 KEY1 第 1 个 32 位内容。(只读)

Register 4.50. EFUSE_RD_KEY1_DATA2_REG (0x00C4)



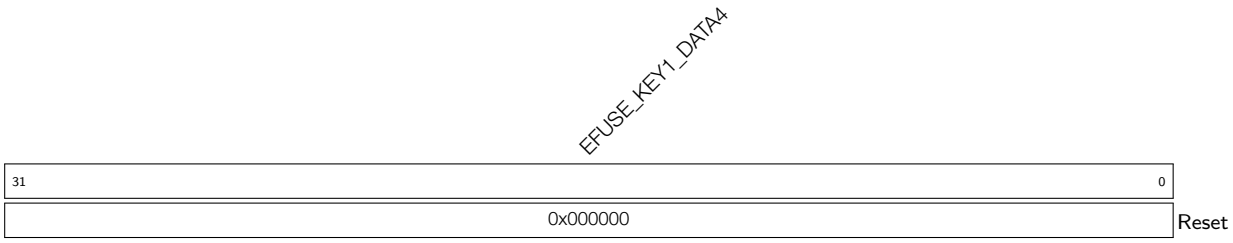
EFUSE_KEY1_DATA2 存储 KEY1 第 2 个 32 位内容。(只读)

Register 4.51. EFUSE_RD_KEY1_DATA3_REG (0x00C8)



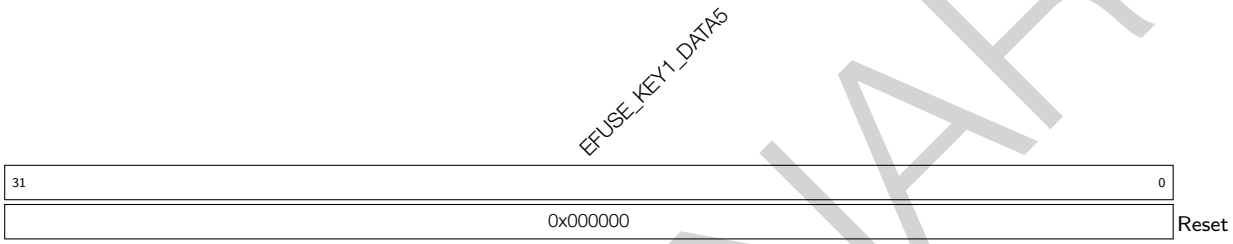
EFUSE_KEY1_DATA3 存储 KEY1 第 3 个 32 位内容。(只读)

Register 4.52. EFUSE_RD_KEY1_DATA4_REG (0x00CC)



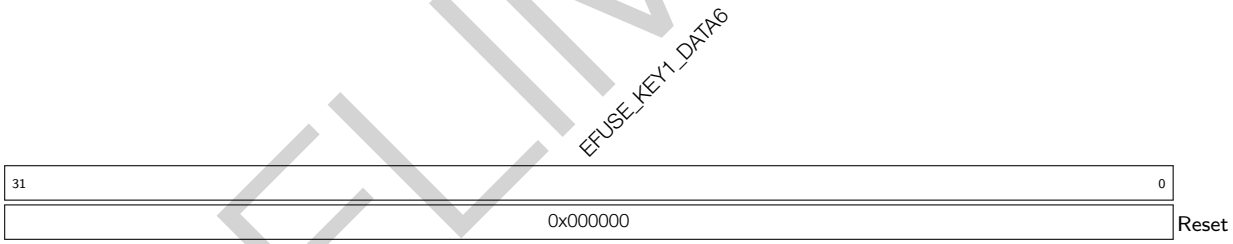
EFUSE_KEY1_DATA4 存储 KEY1 第 4 个 32 位内容。(只读)

Register 4.53. EFUSE_RD_KEY1_DATA5_REG (0x00D0)



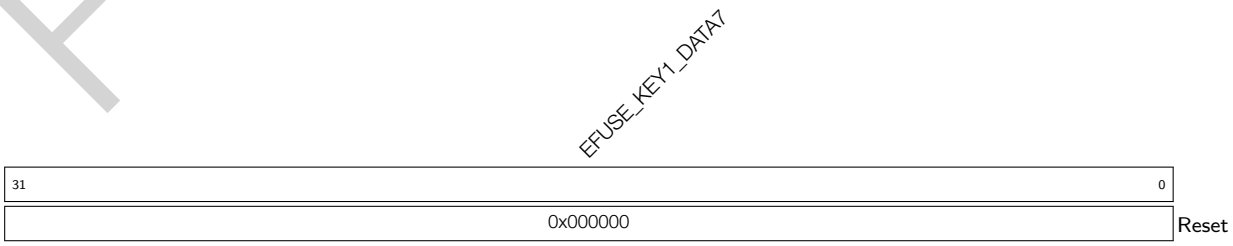
EFUSE_KEY1_DATA5 存储 KEY1 第 5 个 32 位内容。(只读)

Register 4.54. EFUSE_RD_KEY1_DATA6_REG (0x00D4)



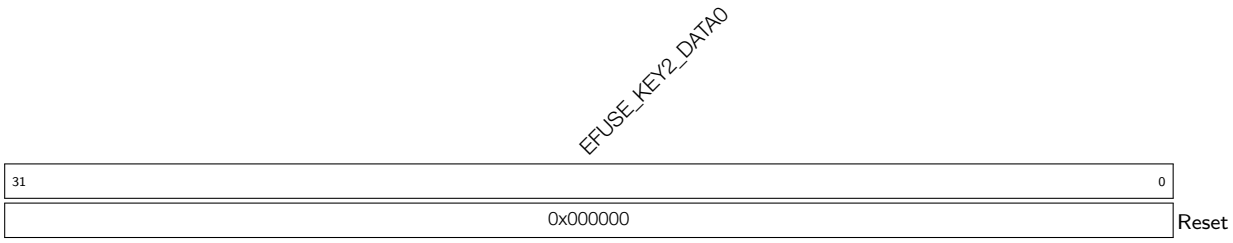
EFUSE_KEY1_DATA6 存储 KEY1 第 6 个 32 位内容。(只读)

Register 4.55. EFUSE_RD_KEY1_DATA7_REG (0x00D8)



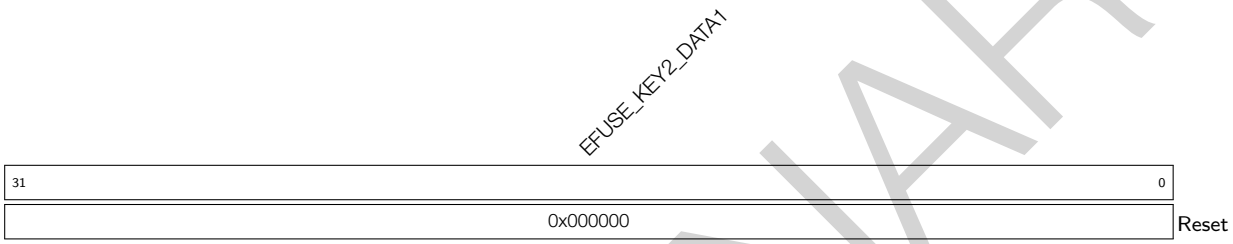
EFUSE_KEY1_DATA7 存储 KEY1 第 7 个 32 位内容。(只读)

Register 4.56. EFUSE_RD_KEY2_DATA0_REG (0x00DC)



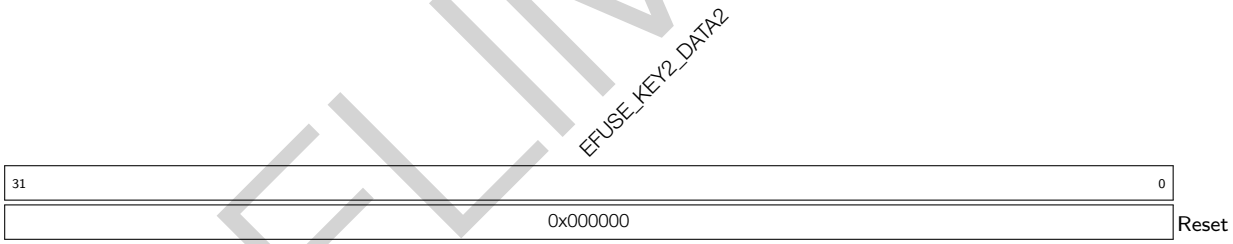
EFUSE_KEY2_DATA0 存储 KEY2 第 0 个 32 位内容。(只读)

Register 4.57. EFUSE_RD_KEY2_DATA1_REG (0x00E0)



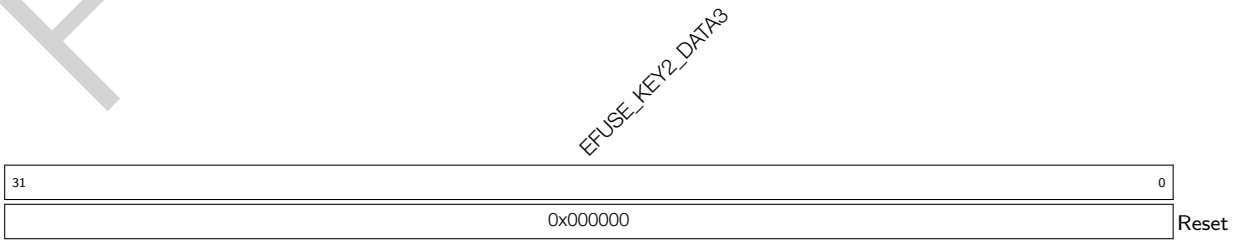
EFUSE_KEY2_DATA1 存储 KEY2 第 1 个 32 位内容。(只读)

Register 4.58. EFUSE_RD_KEY2_DATA2_REG (0x00E4)



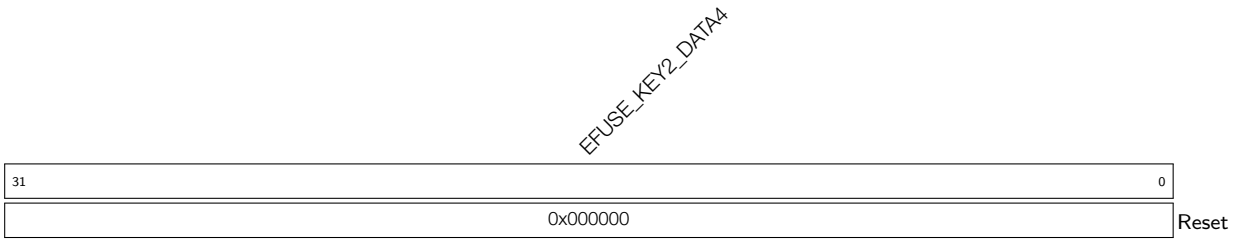
EFUSE_KEY2_DATA2 存储 KEY2 第 2 个 32 位内容。(只读)

Register 4.59. EFUSE_RD_KEY2_DATA3_REG (0x00E8)



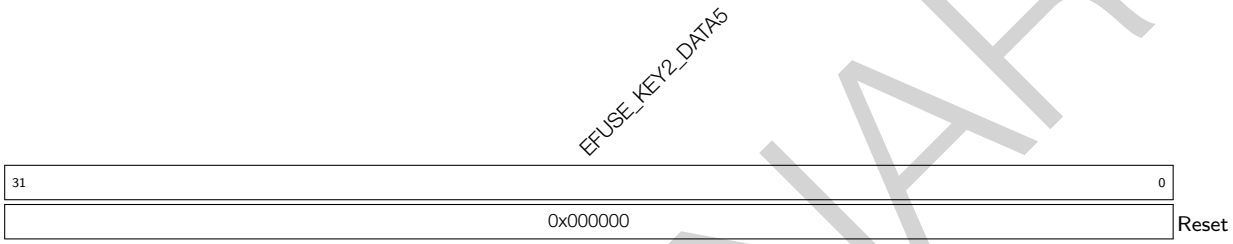
EFUSE_KEY2_DATA3 存储 KEY2 第 3 个 32 位内容。(只读)

Register 4.60. EFUSE_RD_KEY2_DATA4_REG (0x00EC)



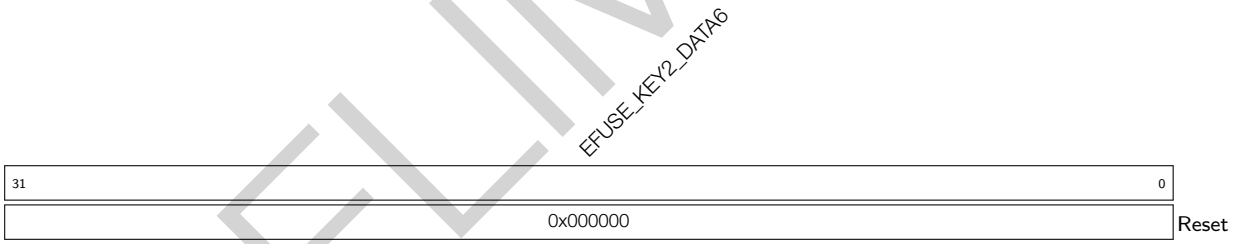
EFUSE_KEY2_DATA4 存储 KEY2 第 4 个 32 位内容。(只读)

Register 4.61. EFUSE_RD_KEY2_DATA5_REG (0x00F0)



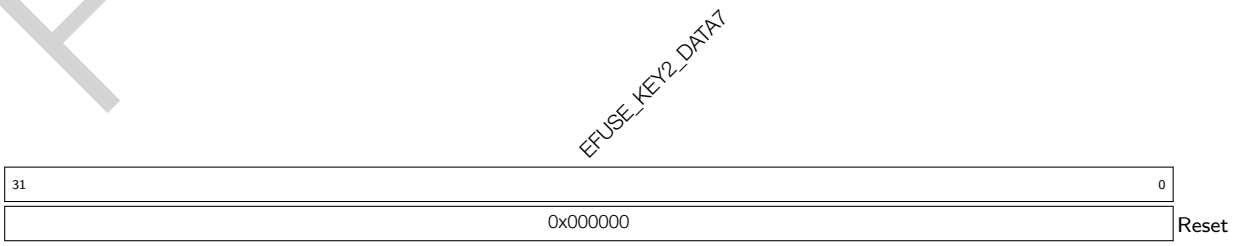
EFUSE_KEY2_DATA5 存储 KEY2 第 5 个 32 位内容。(只读)

Register 4.62. EFUSE_RD_KEY2_DATA6_REG (0x00F4)



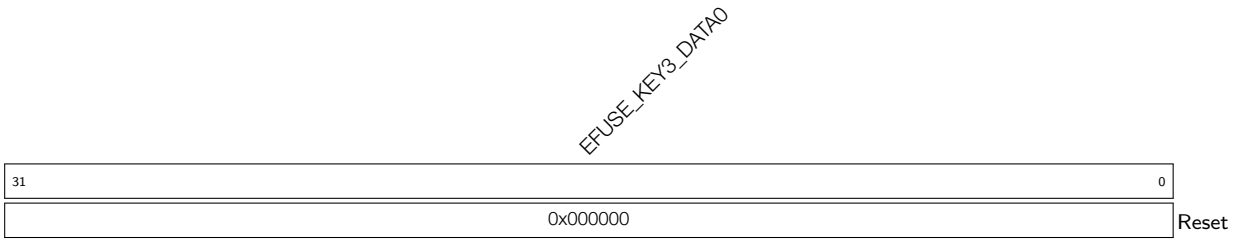
EFUSE_KEY2_DATA6 存储 KEY2 第 6 个 32 位内容。(只读)

Register 4.63. EFUSE_RD_KEY2_DATA7_REG (0x00F8)



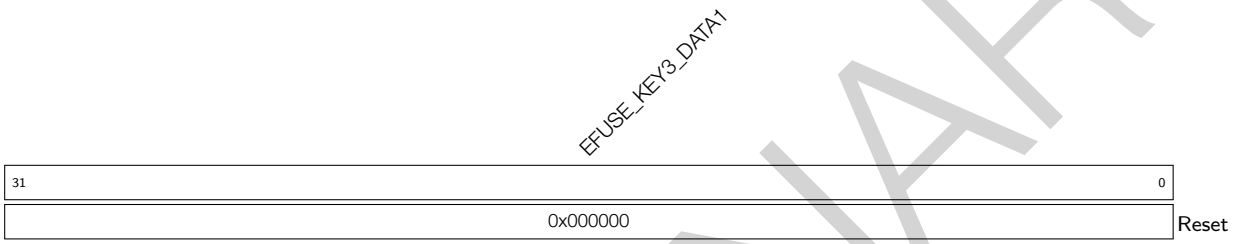
EFUSE_KEY2_DATA7 存储 KEY2 第 7 个 32 位内容。(只读)

Register 4.64. EFUSE_RD_KEY3_DATA0_REG (0x00FC)



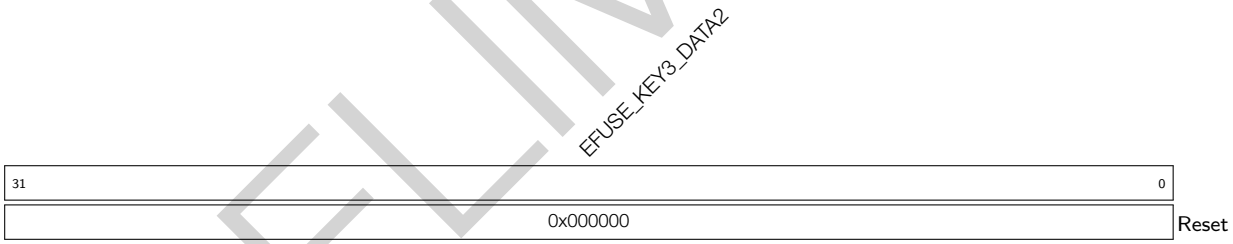
EFUSE_KEY3_DATA0 存储 KEY3 第 0 个 32 位内容。(只读)

Register 4.65. EFUSE_RD_KEY3_DATA1_REG (0x0100)



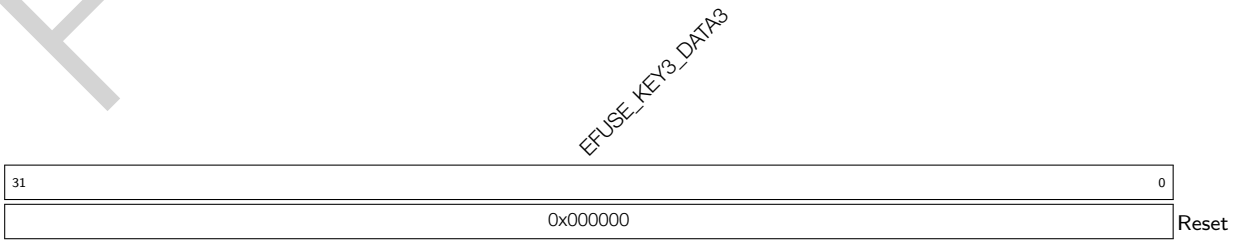
EFUSE_KEY3_DATA1 存储 KEY3 第 1 个 32 位内容。(只读)

Register 4.66. EFUSE_RD_KEY3_DATA2_REG (0x0104)



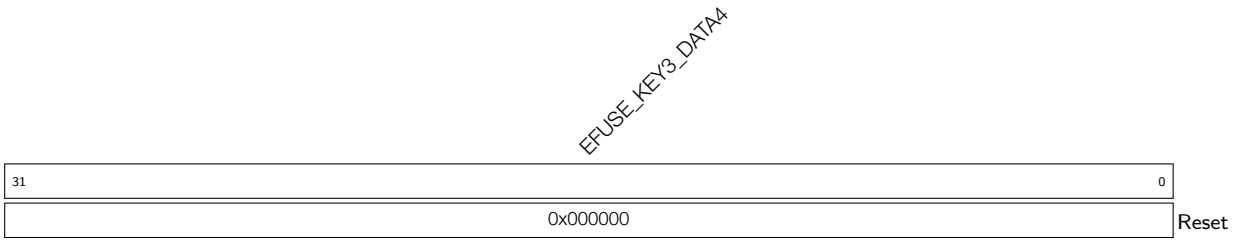
EFUSE_KEY3_DATA2 存储 KEY3 第 2 个 32 位内容。(只读)

Register 4.67. EFUSE_RD_KEY3_DATA3_REG (0x0108)



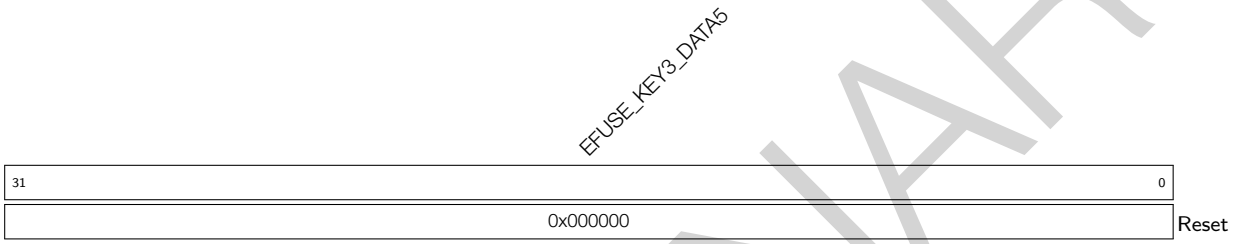
EFUSE_KEY3_DATA3 存储 KEY3 第 3 个 32 位内容。(只读)

Register 4.68. EFUSE_RD_KEY3_DATA4_REG (0x010C)



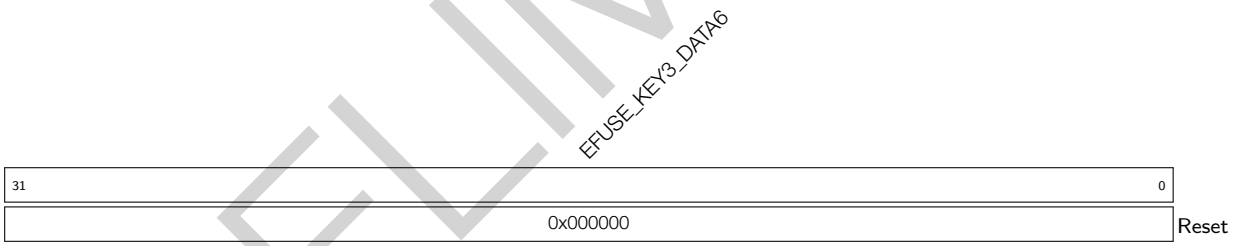
EFUSE_KEY3_DATA4 存储 KEY3 第 4 个 32 位内容。(只读)

Register 4.69. EFUSE_RD_KEY3_DATA5_REG (0x0110)



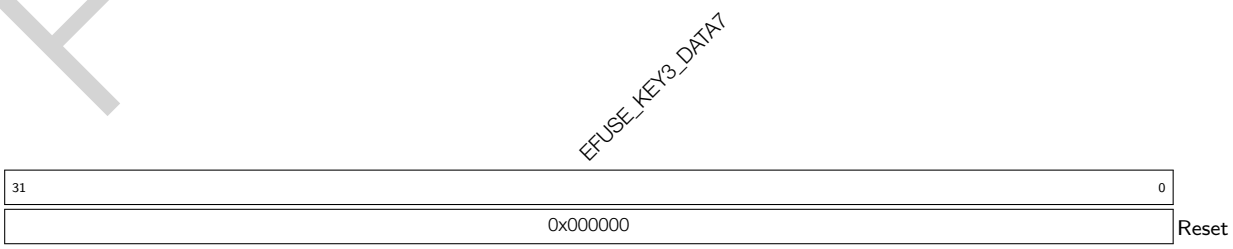
EFUSE_KEY3_DATA5 存储 KEY3 第 5 个 32 位内容。(只读)

Register 4.70. EFUSE_RD_KEY3_DATA6_REG (0x0114)



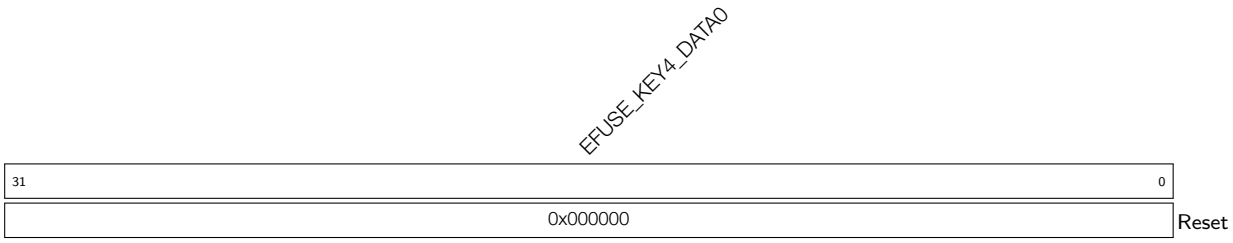
EFUSE_KEY3_DATA6 存储 KEY3 第 6 个 32 位内容。(只读)

Register 4.71. EFUSE_RD_KEY3_DATA7_REG (0x0118)



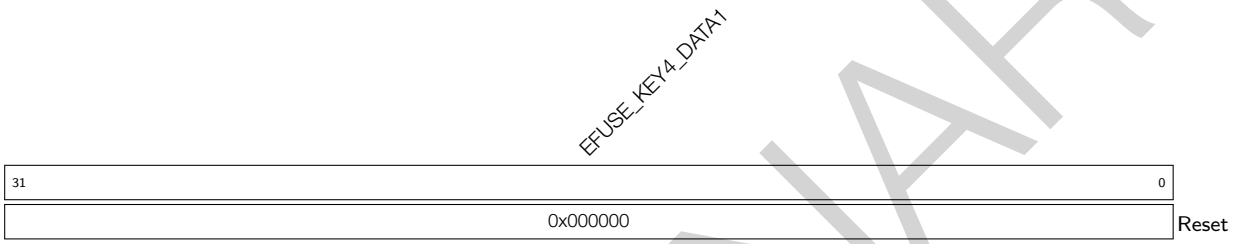
EFUSE_KEY3_DATA7 存储 KEY3 第 7 个 32 位内容。(只读)

Register 4.72. EFUSE_RD_KEY4_DATA0_REG (0x011C)



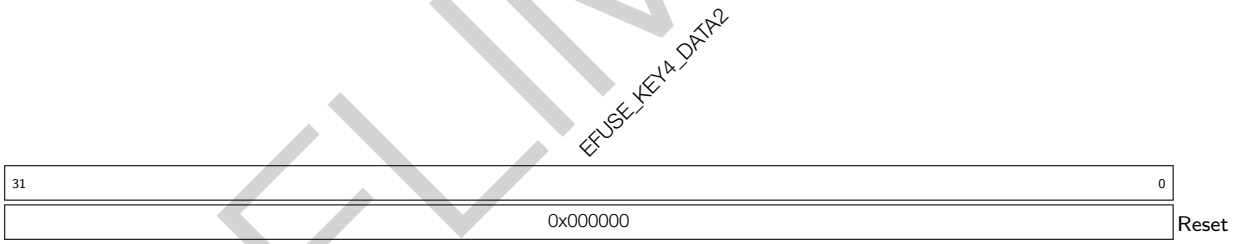
EFUSE_KEY4_DATA0 存储 KEY4 第 0 个 32 位内容。(只读)

Register 4.73. EFUSE_RD_KEY4_DATA1_REG (0x0120)



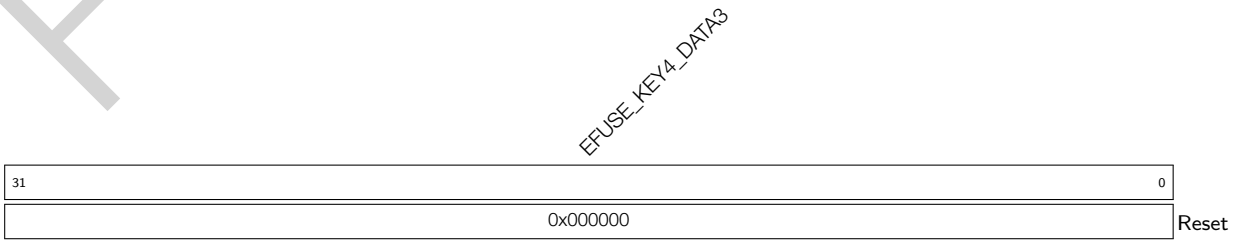
EFUSE_KEY4_DATA1 存储 KEY4 第 1 个 32 位内容。(只读)

Register 4.74. EFUSE_RD_KEY4_DATA2_REG (0x0124)



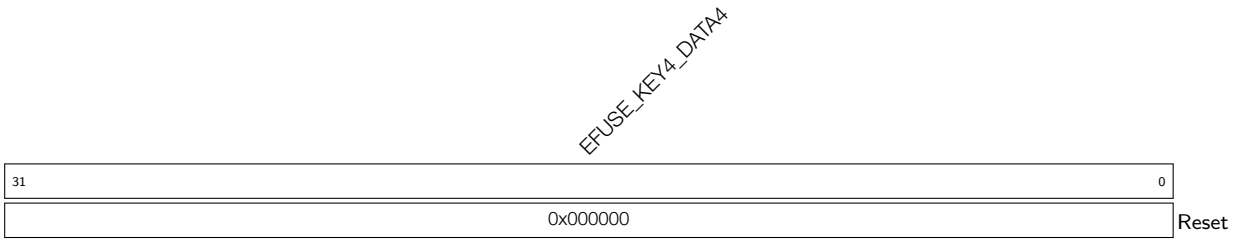
EFUSE_KEY4_DATA2 存储 KEY4 第 2 个 32 位内容。(只读)

Register 4.75. EFUSE_RD_KEY4_DATA3_REG (0x0128)



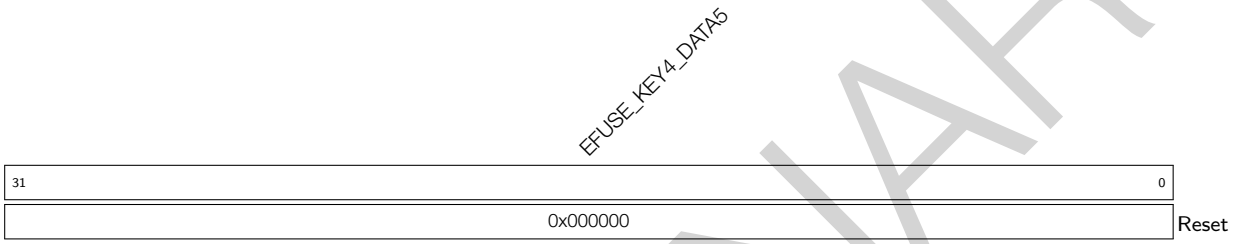
EFUSE_KEY4_DATA3 存储 KEY4 第 3 个 32 位内容。(只读)

Register 4.76. EFUSE_RD_KEY4_DATA4_REG (0x012C)



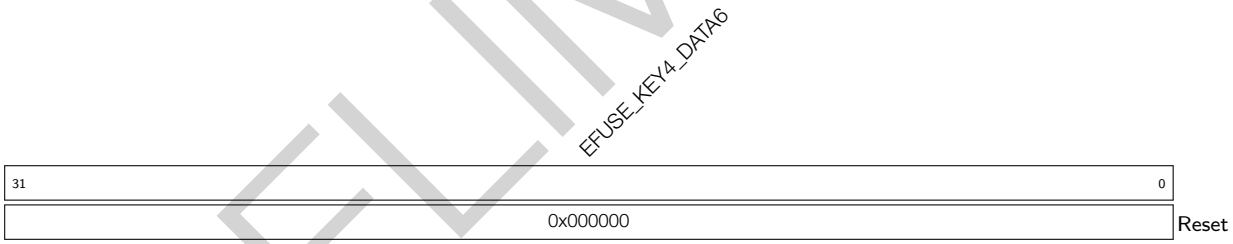
EFUSE_KEY4_DATA4 存储 KEY4 第 4 个 32 位内容。(只读)

Register 4.77. EFUSE_RD_KEY4_DATA5_REG (0x0130)



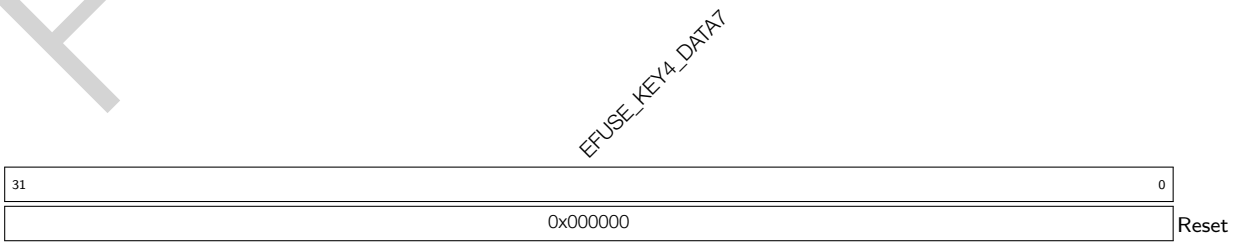
EFUSE_KEY4_DATA5 存储 KEY4 第 5 个 32 位内容。(只读)

Register 4.78. EFUSE_RD_KEY4_DATA6_REG (0x0134)



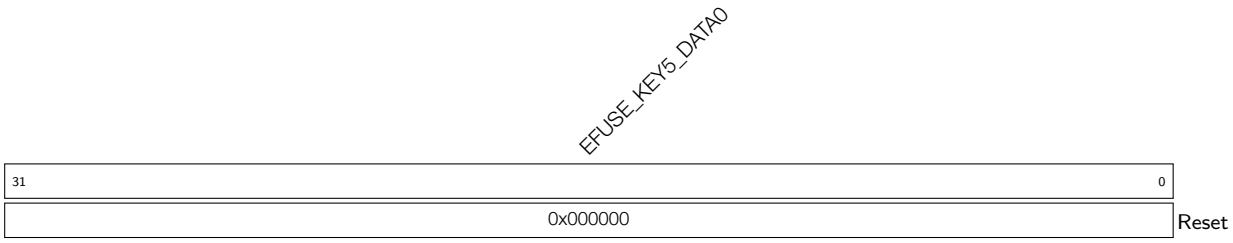
EFUSE_KEY4_DATA6 存储 KEY4 第 6 个 32 位内容。(只读)

Register 4.79. EFUSE_RD_KEY4_DATA7_REG (0x0138)



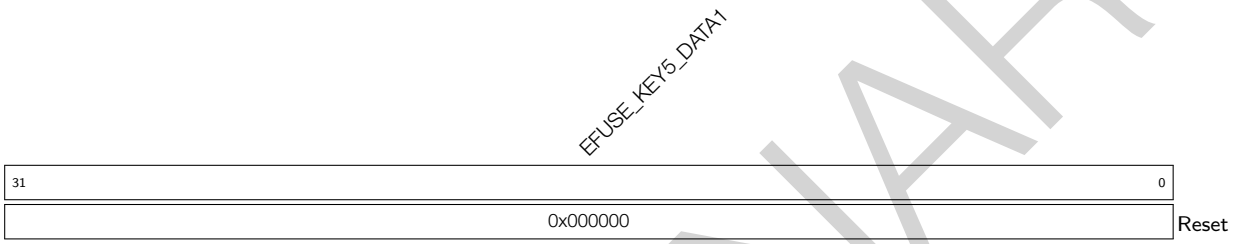
EFUSE_KEY4_DATA7 存储 KEY4 第 7 个 32 位内容。(只读)

Register 4.80. EFUSE_RD_KEY5_DATA0_REG (0x013C)



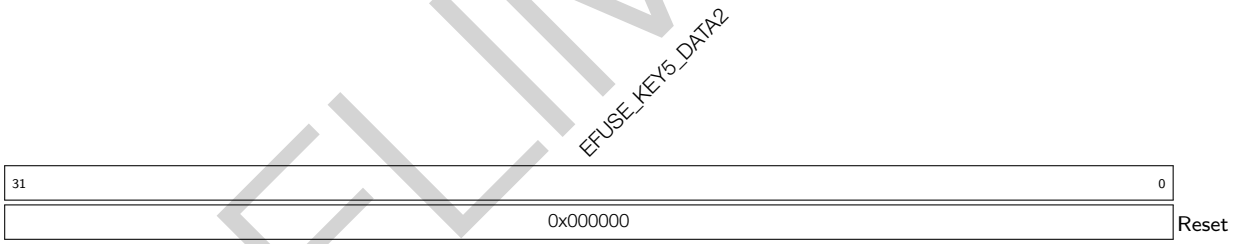
EFUSE_KEY5_DATA0 存储 KEY5 第 0 个 32 位内容。(只读)

Register 4.81. EFUSE_RD_KEY5_DATA1_REG (0x0140)



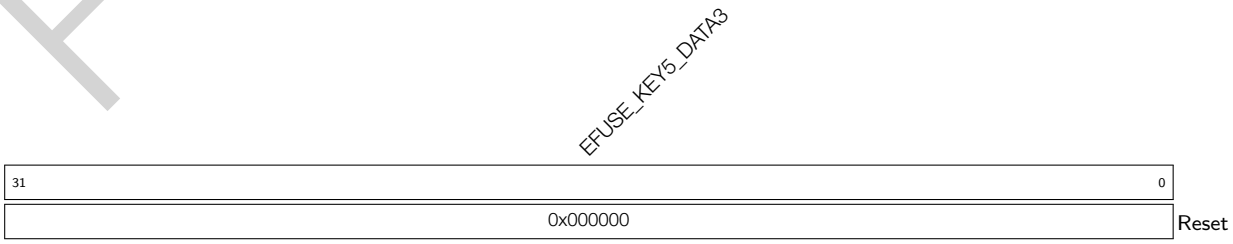
EFUSE_KEY5_DATA1 存储 KEY5 第 1 个 32 位内容。(只读)

Register 4.82. EFUSE_RD_KEY5_DATA2_REG (0x0144)



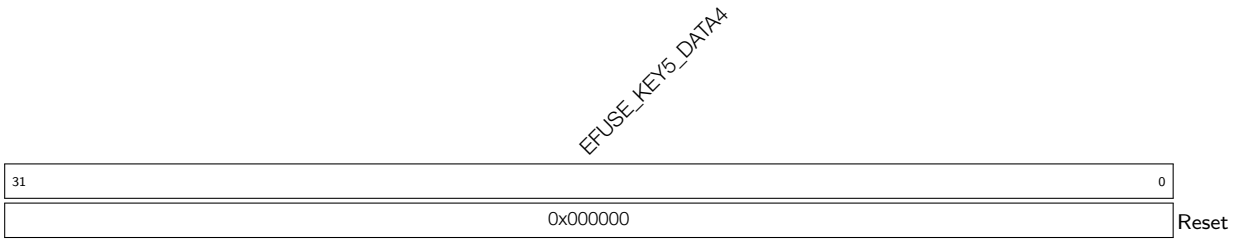
EFUSE_KEY5_DATA2 存储 KEY5 第 2 个 32 位内容。(只读)

Register 4.83. EFUSE_RD_KEY5_DATA3_REG (0x0148)



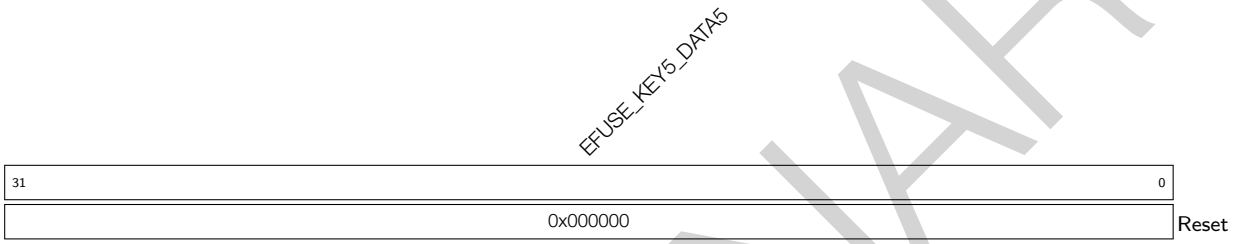
EFUSE_KEY5_DATA3 存储 KEY5 第 3 个 32 位内容。(只读)

Register 4.84. EFUSE_RD_KEY5_DATA4_REG (0x014C)



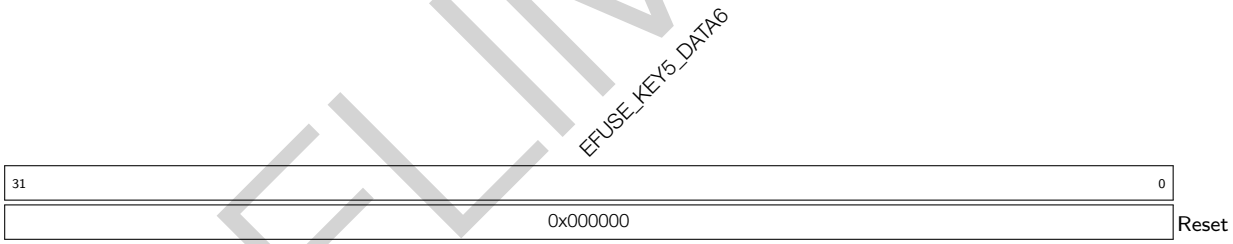
EFUSE_KEY5_DATA4 存储 KEY5 第 4 个 32 位内容。(只读)

Register 4.85. EFUSE_RD_KEY5_DATA5_REG (0x0150)



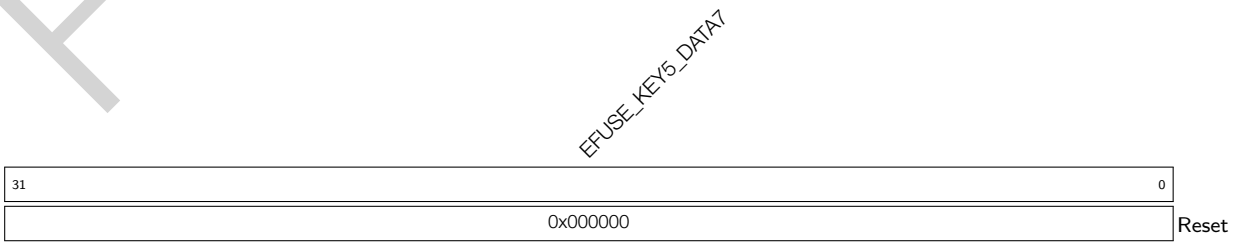
EFUSE_KEY5_DATA5 存储 KEY5 第 5 个 32 位内容。(只读)

Register 4.86. EFUSE_RD_KEY5_DATA6_REG (0x0154)



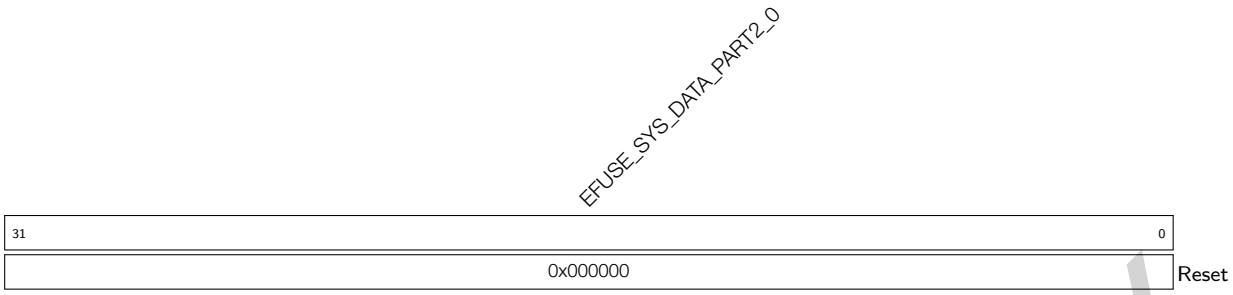
EFUSE_KEY5_DATA6 存储 KEY5 第 6 个 32 位内容。(只读)

Register 4.87. EFUSE_RD_KEY5_DATA7_REG (0x0158)



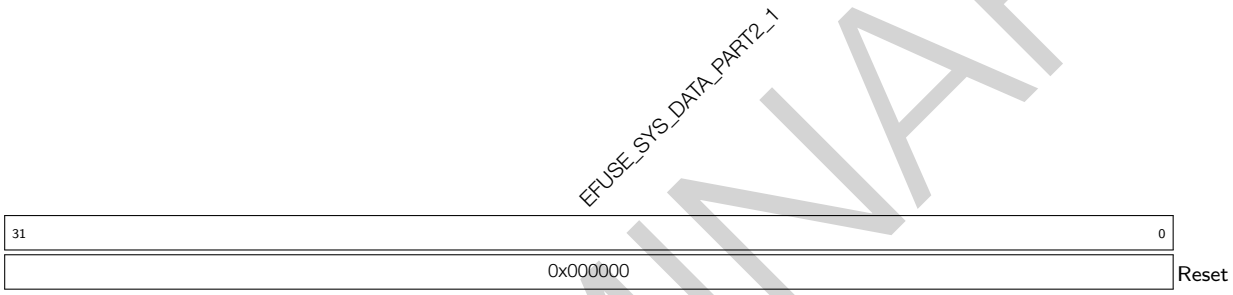
EFUSE_KEY5_DATA7 存储 KEY5 第 7 个 32 位内容。(只读)

Register 4.88. EFUSE_RD_SYS_PART2_DATA0_REG (0x015C)



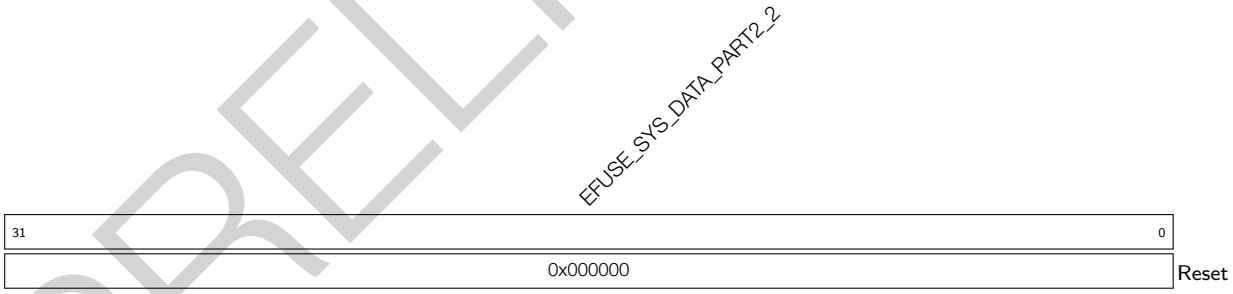
EFUSE_SYS_DATA_PART2_0 存储系统数据第 2 部分的第 0 个 32 位内容。(只读)

Register 4.89. EFUSE_RD_SYS_PART2_DATA1_REG (0x0160)



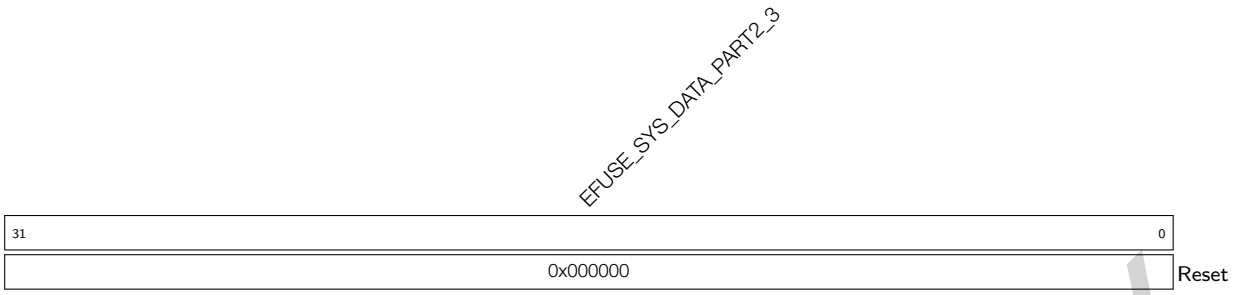
EFUSE_SYS_DATA_PART2_1 存储系统数据第 2 部分的第 1 个 32 位内容。(只读)

Register 4.90. EFUSE_RD_SYS_PART2_DATA2_REG (0x0164)



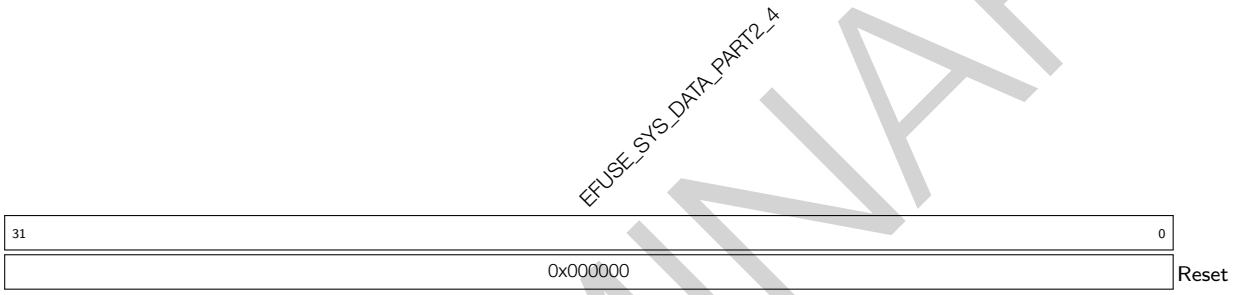
EFUSE_SYS_DATA_PART2_2 存储系统数据第 2 部分的第 2 个 32 位内容。(只读)

Register 4.91. EFUSE_RD_SYS_PART2_DATA3_REG (0x0168)



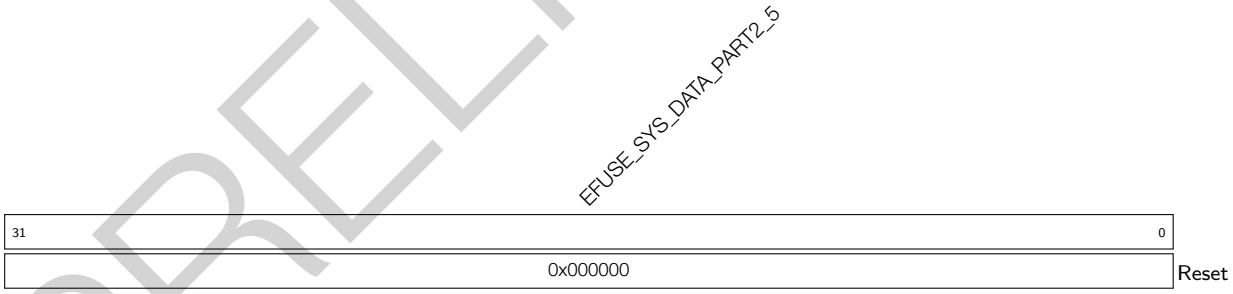
EFUSE_SYS_DATA_PART2_3 存储系统数据第 2 部分的第 3 个 32 位内容。(只读)

Register 4.92. EFUSE_RD_SYS_PART2_DATA4_REG (0x016C)



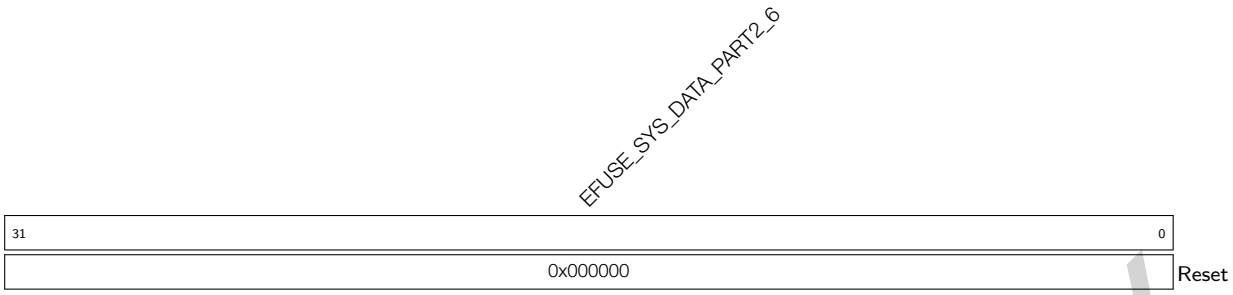
EFUSE_SYS_DATA_PART2_4 存储系统数据第 2 部分的第 4 个 32 位内容。(只读)

Register 4.93. EFUSE_RD_SYS_PART2_DATA5_REG (0x0170)



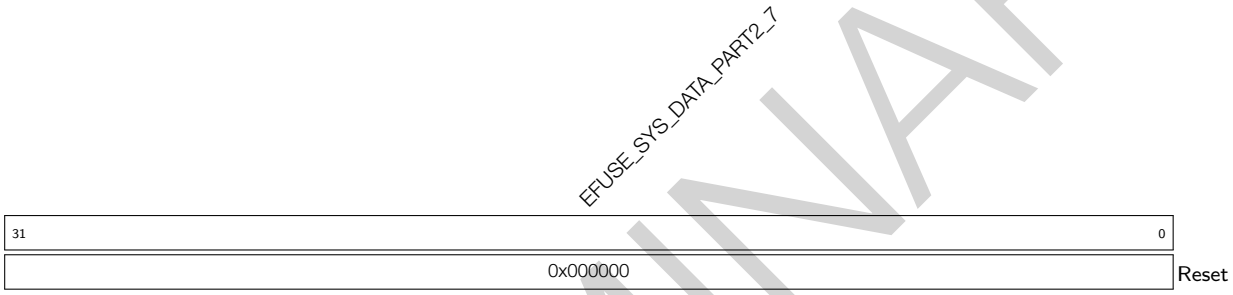
EFUSE_SYS_DATA_PART2_5 存储系统数据第 2 部分的第 5 个 32 位内容。(只读)

Register 4.94. EFUSE_RD_SYS_PART2_DATA6_REG (0x0174)



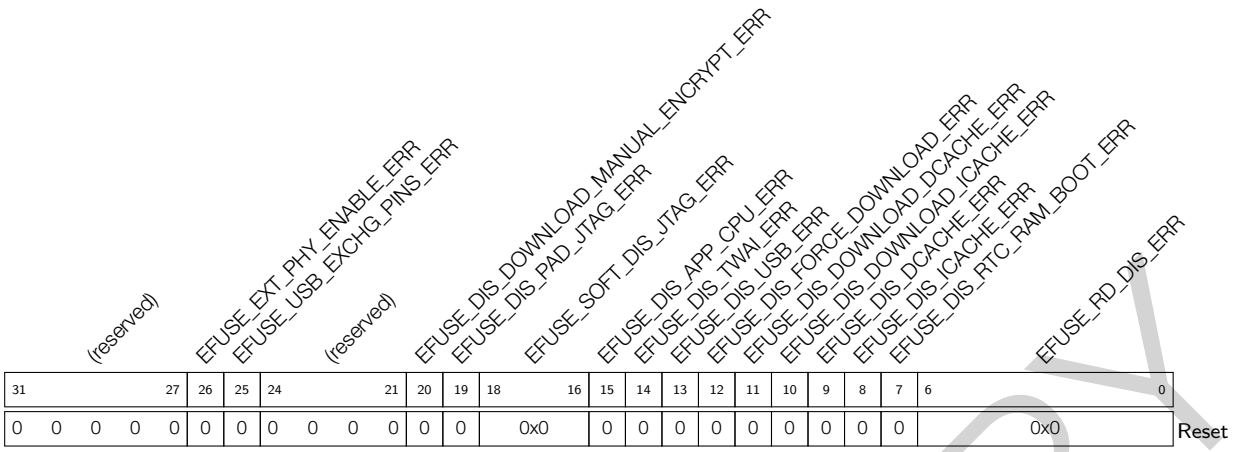
EFUSE_SYS_DATA_PART2_6 存储系统数据第 2 部分的第 6 个 32 位内容。(只读)

Register 4.95. EFUSE_RD_SYS_PART2_DATA7_REG (0x0178)



EFUSE_SYS_DATA_PART2_7 存储系统数据第 2 部分的第 7 个 32 位内容。(只读)

Register 4.96. EFUSE_RD_REPEAT_ERR0_REG (0x017C)



EFUSE_RD_DIS_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_RTC_RAM_BOOT_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_ICACHE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_DCACHE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_DOWNLOAD_ICACHE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_DOWNLOAD_DCACHE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_FORCE_DOWNLOAD_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_USB_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_TWAI_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_APP_CPU_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_SOFT_DIS_JTAG_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

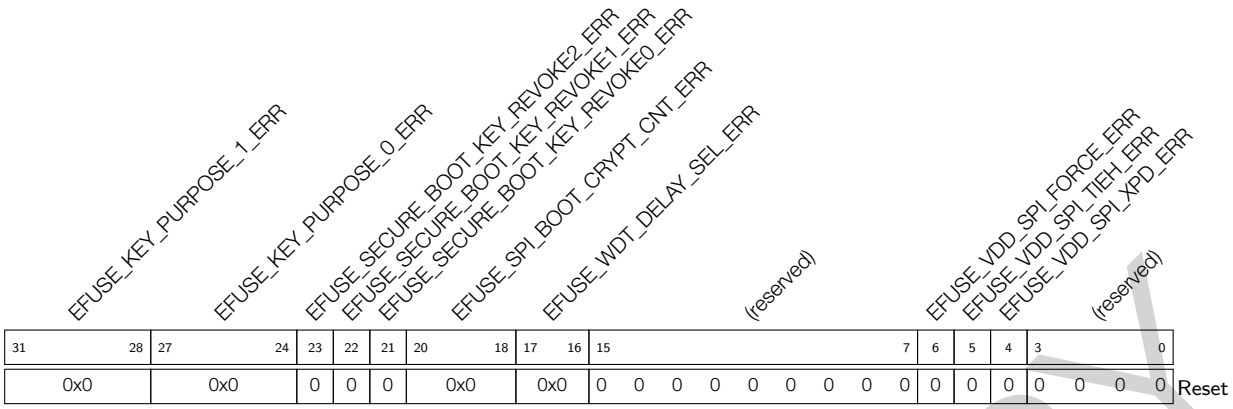
EFUSE_DIS_PAD_JTAG_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_USB_EXCHG_PINS_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_EXT_PHY_ENABLE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

Register 4.97. EFUSE_RD_REPEAT_ERR1_REG (0x0180)



EFUSE_VDD_SPI_XPD_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_VDD_SPI_TIEH_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_VDD_SPI_FORCE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_WDT_DELAY_SEL_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_SPI_BOOT_CRYPT_CNT_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_SECURE_BOOT_KEY_REVOKE0_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

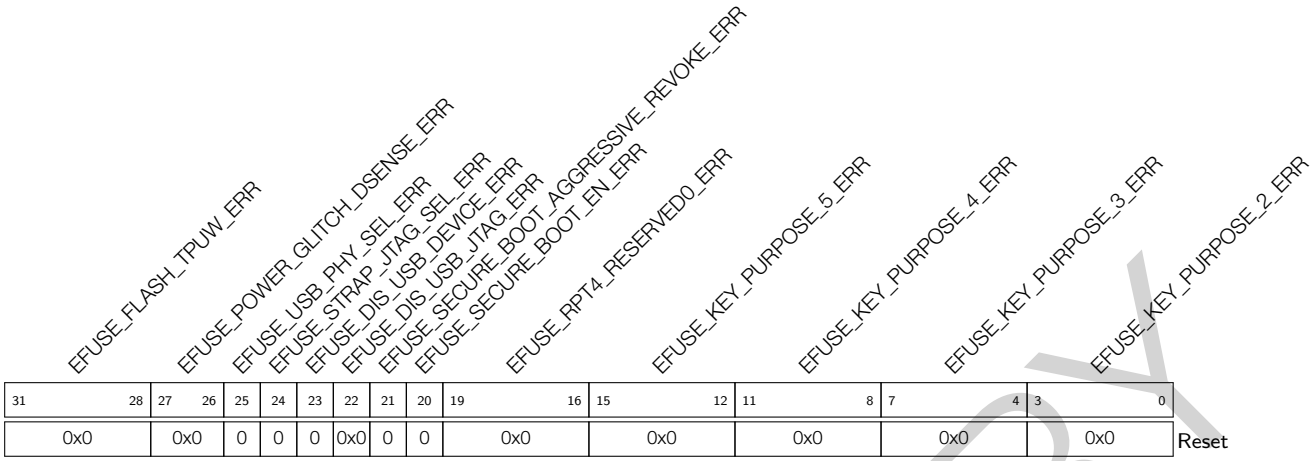
EFUSE_SECURE_BOOT_KEY_REVOKE1_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_SECURE_BOOT_KEY_REVOKE2_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_KEY_PURPOSE_0_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_KEY_PURPOSE_1_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

Register 4.98. EFUSE_RD_REPEAT_ERR2_REG (0x0184)



EFUSE_KEY_PURPOSE_2_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_KEY_PURPOSE_3_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_KEY_PURPOSE_4_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_KEY_PURPOSE_5_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_RPT4_RESERVED0_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_SECURE_BOOT_EN_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_USB_JTAG_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_USB_DEVICE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

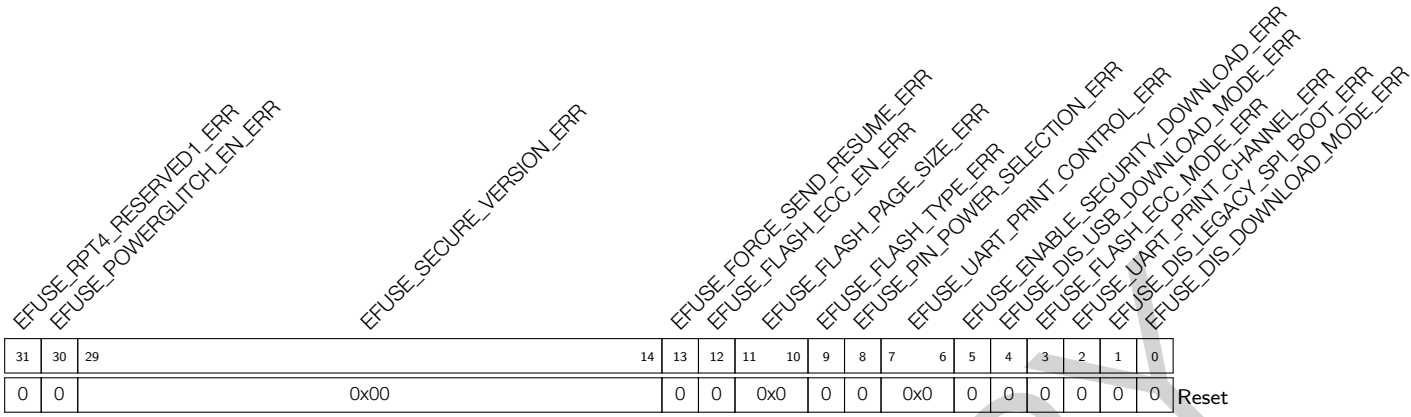
EFUSE_STRAP_JTAG_SEL_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_USB_PHY_SEL_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_POWER_GLITCH_DSENSE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_FLASH_TPUW_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

Register 4.99. EFUSE_RD_REPEAT_ERR3_REG (0x0188)



EFUSE_DIS_DOWNLOAD_MODE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_LEGACY_SPI_BOOT_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_UART_PRINT_CHANNEL_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_FLASH_ECC_MODE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_DIS_USB_DOWNLOAD_MODE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_ENABLE_SECURITY_DOWNLOAD_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_UART_PRINT_CONTROL_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_PIN_POWER_SELECTION_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_FLASH_TYPE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_FLASH_PAGE_SIZE_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_FLASH_ECC_EN_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

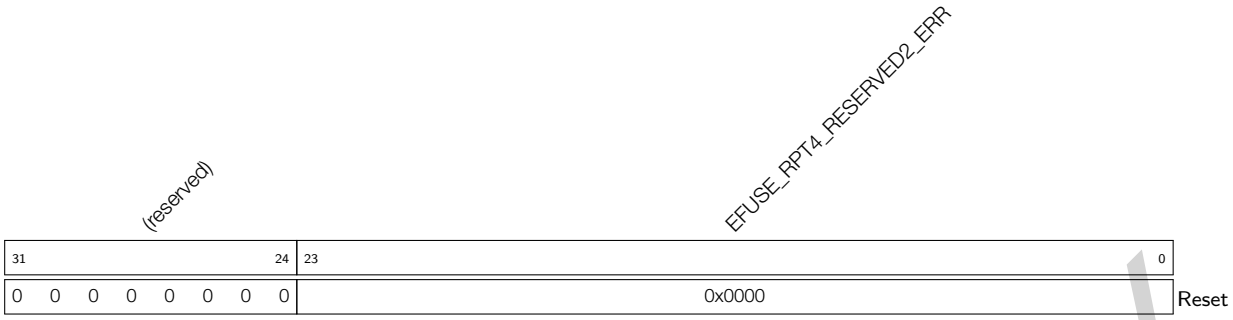
EFUSE_FORCE_SEND_RESUME_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_SECURE_VERSION_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

EFUSE_POWERGLITCH_EN_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

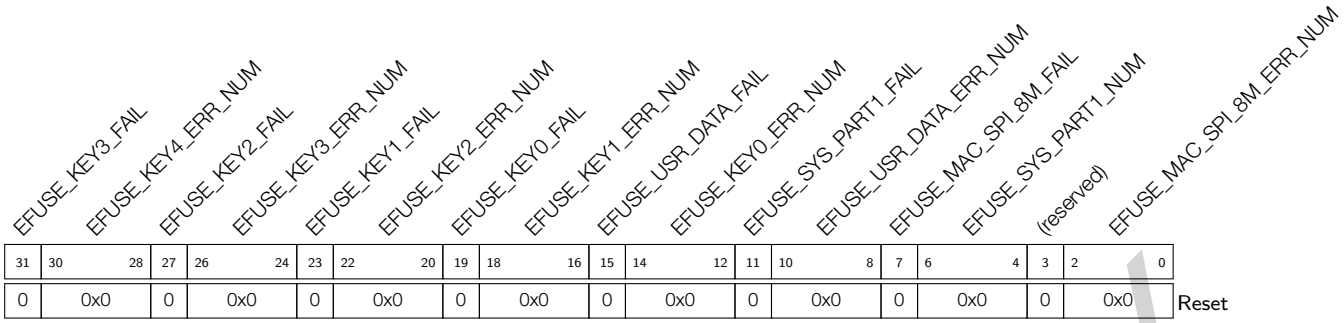
EFUSE_RPT4_RESERVED1_ERR 保留。(只读)

Register 4.100. EFUSE_RD_REPEAT_ERR4_REG (0x0190)



EFUSE_RPT4_RESERVED2_ERR 若该参数中任意比特为 1，表明出现烧写错误。(只读)

Register 4.101. EFUSE_RD_RS_ERR0_REG (0x01C0)



EFUSE_MAC_SPI_8M_ERR_NUM 指示用户数据的错误字节个数。(只读)

EFUSE_SYS_PART1_NUM 指示第 1 部分系统数据的错误字节个数。(只读)

EFUSE_MAC_SPI_8M_FAIL 0: 无烧写错误，MAC_SPI_8M 的数据是可靠的；1: 烧写数据失败，错误字节数超过 6。(只读)

EFUSE_USR_DATA_ERR_NUM 指示用户数据的错误字节个数。(只读)

EFUSE_SYS_PART1_FAIL 0: 无烧写错误，第 1 部分的系统数据是可靠的；1: 烧写数据失败，错误字节数超过 6。(只读)

EFUSE_KEY0_ERR_NUM 指示 KEY0 的错误字节个数。(只读)

EFUSE_USR_DATA_FAIL 0: 无烧写错误，用户数据是可靠的；1: 烧写数据失败，错误字节数超过 6。(只读)

EFUSE_KEY1_ERR_NUM 指示 KEY1 的错误字节个数。(只读)

EFUSE_KEY0_FAIL 0: 无烧写错误，KEY0 数据是可靠的；1: KEY0 烧写失败，错误字节数超过 6。(只读)

EFUSE_KEY2_ERR_NUM 指示 KEY2 的错误字节个数。(只读)

EFUSE_KEY1_FAIL 0: 无烧写错误，KEY1 数据是可靠的；1: KEY1 烧写失败，错误字节数超过 6。(只读)

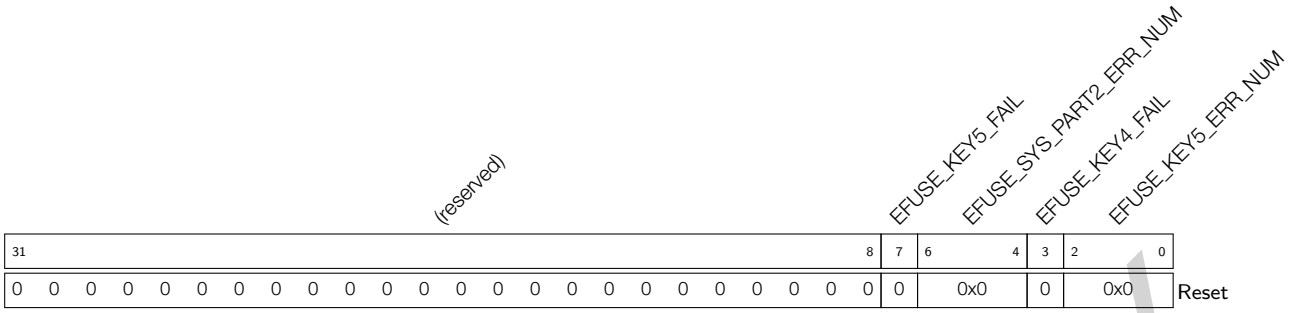
EFUSE_KEY3_ERR_NUM 指示 KEY3 的错误字节个数。(只读)

EFUSE_KEY2_FAIL 0: 无烧写错误，KEY2 数据是可靠的；1: KEY2 烧写失败，错误字节数超过 6。(只读)

EFUSE_KEY4_ERR_NUM 指示 KEY4 的错误字节个数。(只读)

EFUSE_KEY3_FAIL 0: 无烧写错误，KEY3 数据是可靠的；1: KEY3 烧写失败，错误字节数超过 6。(只读)

Register 4.102. EFUSE_RD_RS_ERR1_REG (0x01C4)



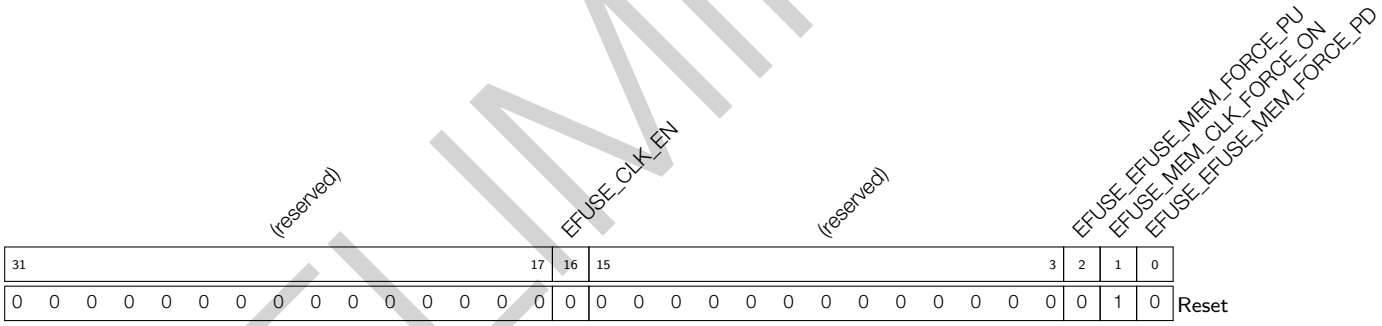
EFUSE_KEY5_ERR_NUM 指示 KEY5 的错误字节个数。(只读)

EFUSE_KEY5_FAIL 0: 无烧写错误，KEY5 数据是可靠的；1: KEY4 数据烧写失败，错误字节数超过 6。(只读)

EFUSE_SYS_PART2_ERR_NUM 指示第 2 部分系统数据的错误字节个数。(只读)

EFUSE_KEY5_FAIL 0: 无烧写错误, KEY5 数据是可靠的; 1: 数据烧写失败, 错误字节数超过 6。(只读)

Register 4.103. EFUSE_CLK_REG (0x01C8)



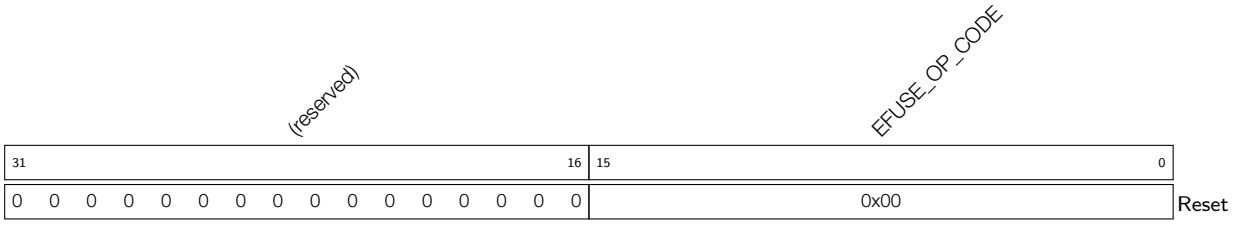
EFUSE_EFUSE_MEM_FORCE_PD 置位强制使 eFuse SRAM 进入低功耗模式。(读/写)

EFUSE_MEM_CLK_FORCE_ON 置位强制激活 eFuse SRAM 的时钟信号。(读/写)

EFUSE_EFUSE_MEM_FORCE_PU 置位强制使 eFuse SRAM 进入工作模式。(读/写)

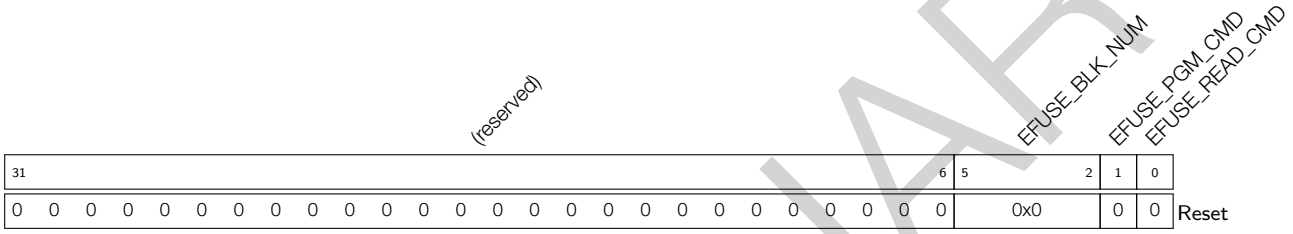
EFUSE_CLK_EN 置位强制使能 eFuse 存储器的时钟信号。(读/写)

Register 4.104. EFUSE_CONF_REG (0x01CC)



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

Register 4.105. EFUSE_CMD_REG (0x01D4)

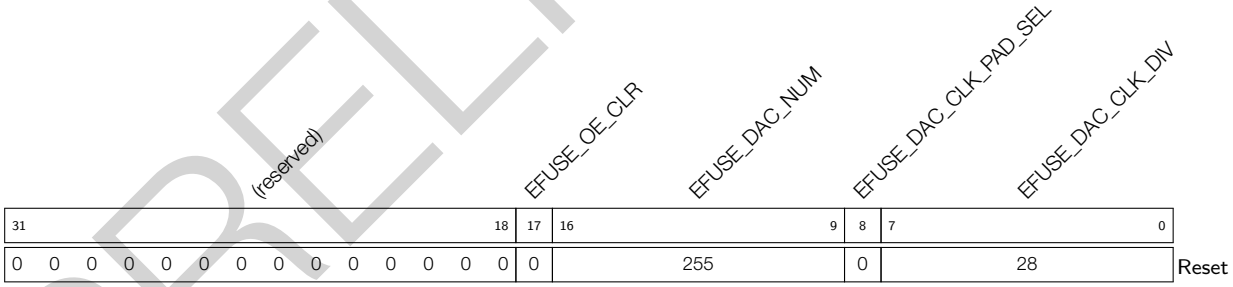


EFUSE_READ_CMD 置位发送读取指令。(R/WS/SC)

EFUSE_PGM_CMD 置位发送烧写指令。(R/WS/SC)

EFUSE_BLK_NUM 表明烧写哪个块，值 0 ~ 10 分别对应 BLOCK0 ~ 10。(读/写)

Register 4.106. EFUSE_DAC_CONF_REG (0x01E8)



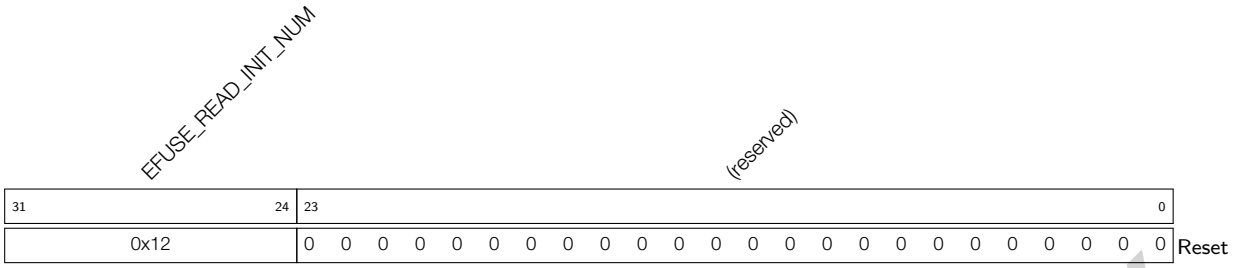
EFUSE_DAC_CLK_DIV 控制烧写电压的爬升时钟分频系数。(读/写)

EFUSE_DAC_CLK_PAD_SEL 无关项。(读/写)

EFUSE_DAC_NUM 烧写供电的上升周期。(读/写)

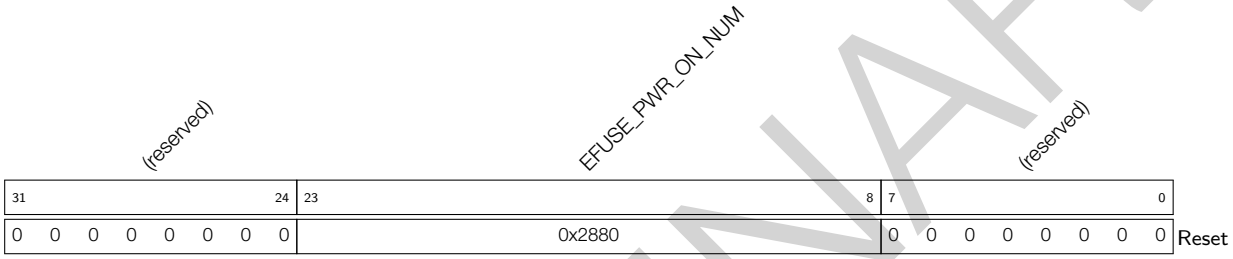
EFUSE_OE_CLR 降低烧写电压的供电能力。(读/写)

Register 4.107. EFUSE_RD_TIM_CONF_REG (0x01EC)



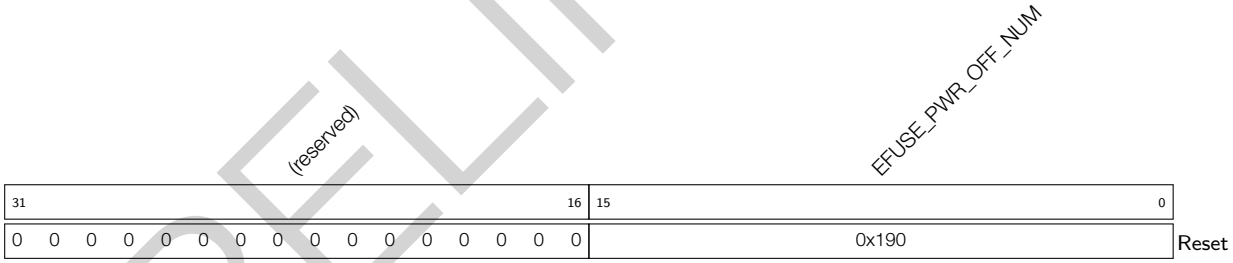
EFUSE_READ_INIT_NUM 配置 eFuse 的首次读取时间。(读/写)

Register 4.108. EFUSE_WR_TIM_CONF1_REG (0x01F4)



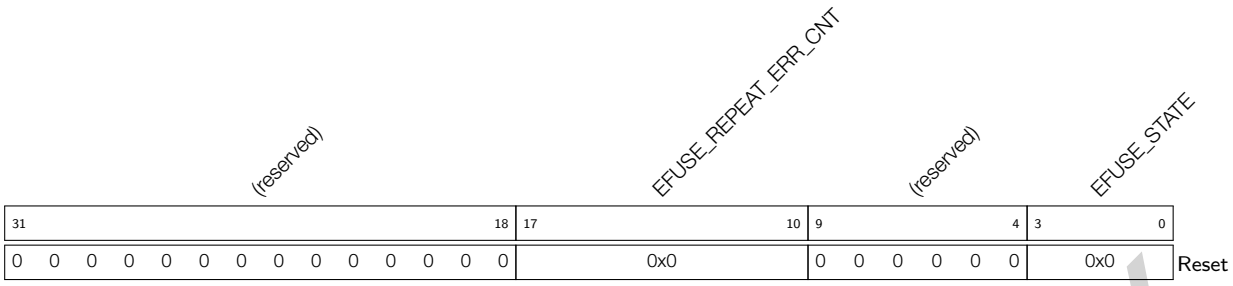
EFUSE_PWR_ON_NUM 配置 VDDQ 的上电时间。(读/写)

Register 4.109. EFUSE_WR_TIM_CONF2_REG (0x01F8)



EFUSE_PWR_OFF_NUM 配置 VDDQ 的掉电时间。(读/写)

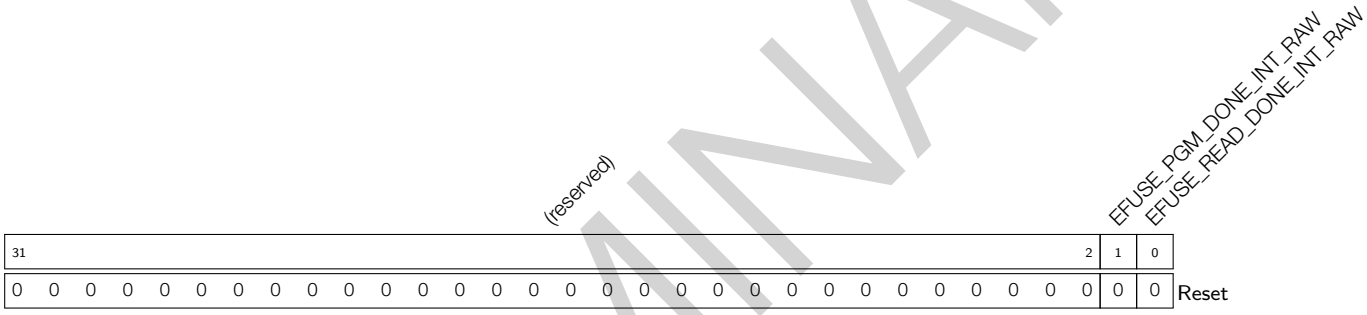
Register 4.110. EFUSE_STATUS_REG (0x01D0)



EFUSE_STATE 表明 eFuse 控制器所处的状态。(只读)

EFUSE_REPEAT_ERR_CNT 表明烧写 BLOCK0 时的错误位的个数。(只读)

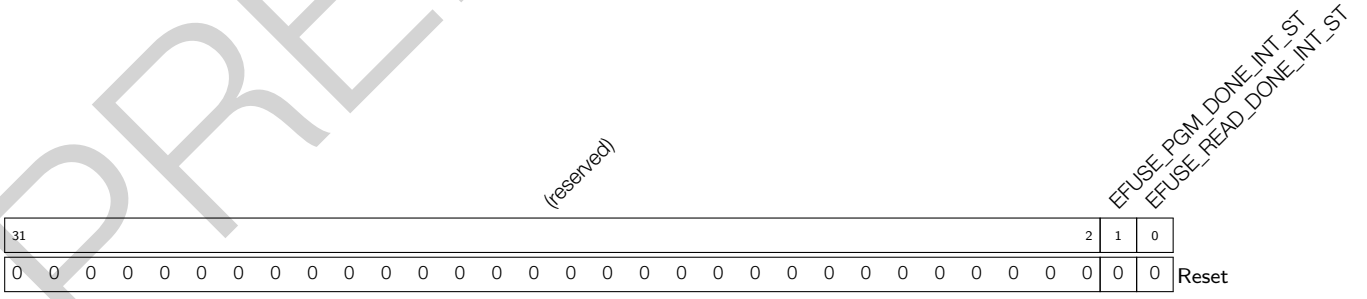
Register 4.111. EFUSE_INT_RAW_REG (0x01D8)



EFUSE_READ_DONE_INT_RAW 读取完成中断的原始中断状态位。(R/WC/SS)

EFUSE_PGM_DONE_INT_RAW 烧写完成中断的原始中断状态位。(R/WC/SS)

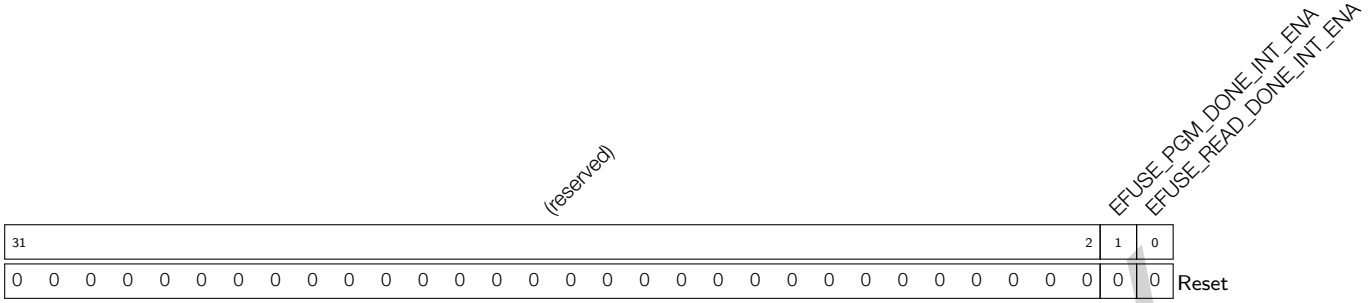
Register 4.112. EFUSE_INT_ST_REG (0x01DC)



EFUSE_READ_DONE_INT_ST 读取完成中断的状态位。(只读)

EFUSE_PGM_DONE_INT_ST 烧写完成中断的状态位。(只读)

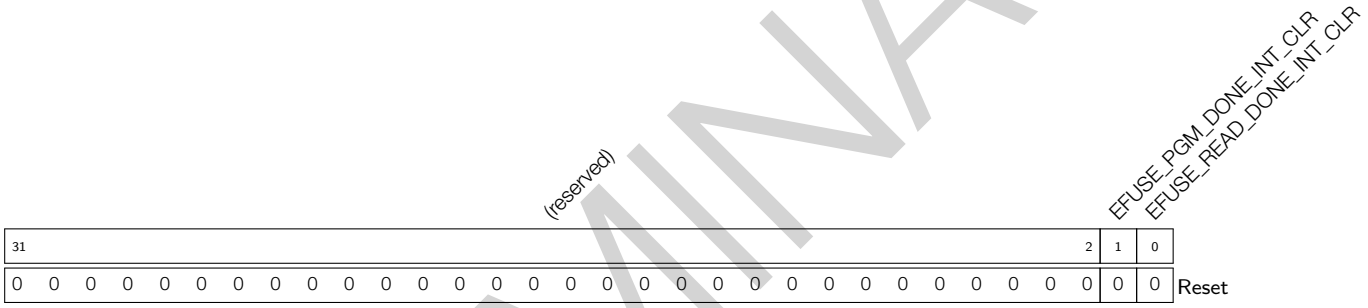
Register 4.113. EFUSE_INT_ENA_REG (0x01E0)



EFUSE_READ_DONE_INT_ENA 读取完成中断的使能位。(读/写)

EFUSE_PGM_DONE_INT_ENA 烧写完成中断的使能位。(读/写)

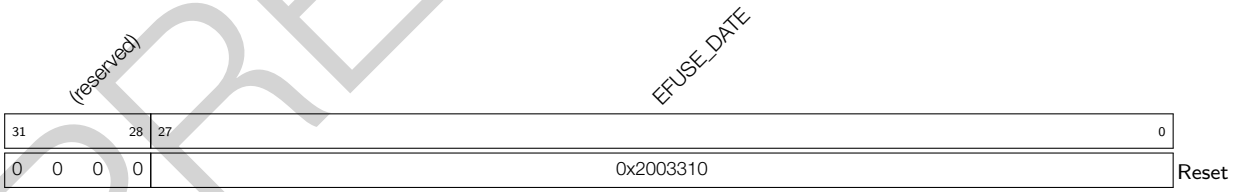
Register 4.114. EFUSE_INT_CLR_REG (0x01E4)



EFUSE_READ_DONE_INT_CLR 读取完成中断的清除位。(只写)

EFUSE_PGM_DONE_INT_CLR 烧写完成中断的清除位。(只写)

Register 4.115. EFUSE_DATE_REG (0x01FC)



EFUSE_DATE 版本控制寄存器。(读/写)

5 IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)

5.1 概述

ESP32-C3 芯片有 22 个物理通用输入输出管脚 (GPIO Pin)。每个管脚都可用作一个通用 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。

5.2 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 管脚的输入信号。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。

5.3 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 5-1 简要展示了 IO MUX 和 GPIO 交换矩阵的工作流程；
- 图 5-2 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 5-3 展示了 GPIO 管脚的接口逻辑。

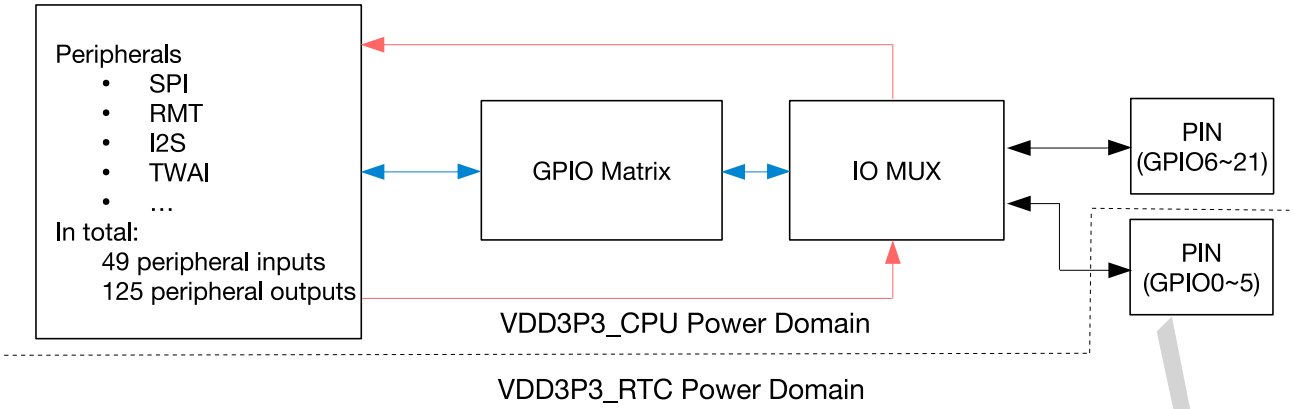


图 5-1. IO MUX 和 GPIO 交换矩阵框图（简图）

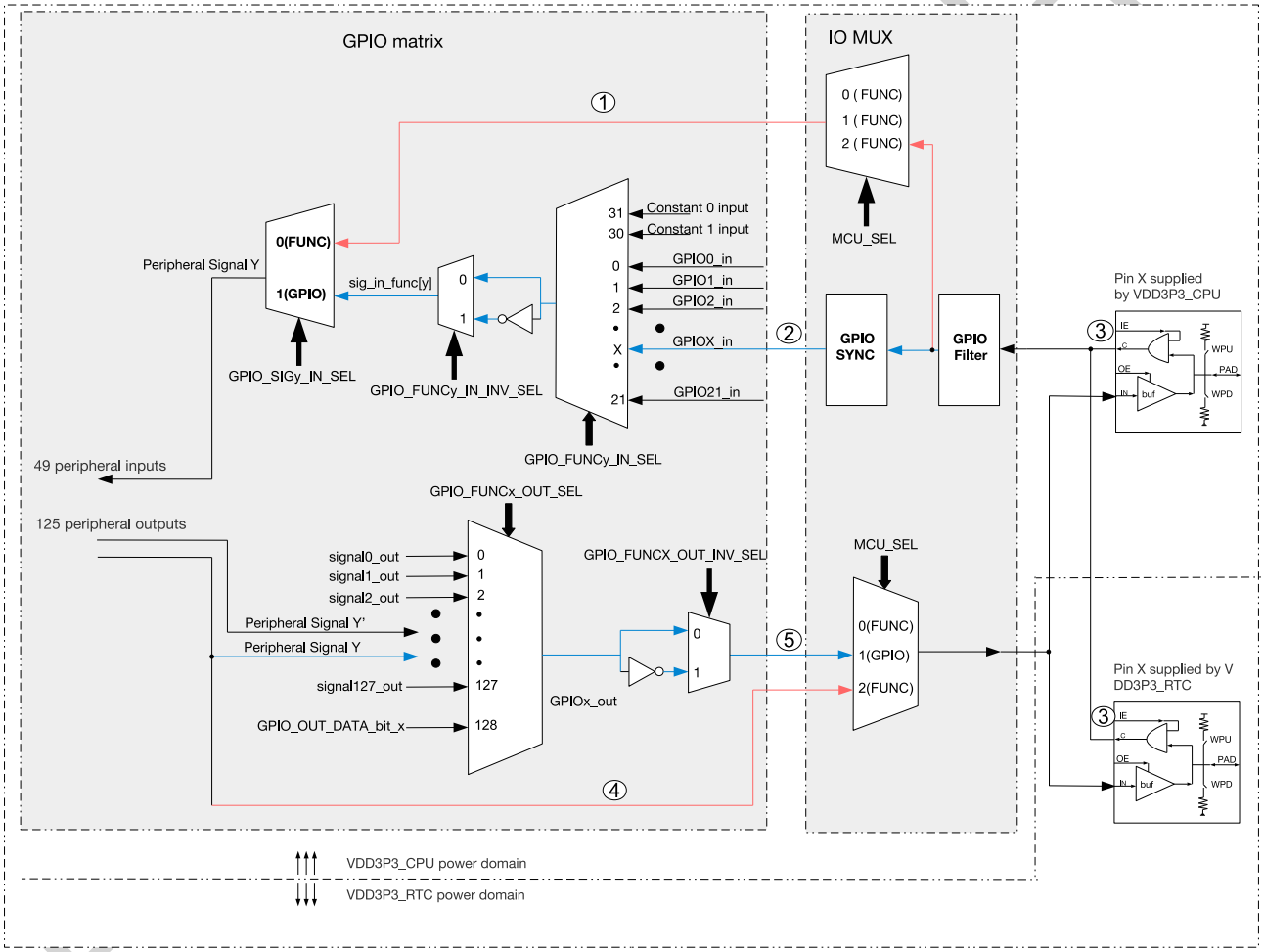


图 5-2. IO MUX 和 GPIO 交换矩阵框图（详图）

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 交换矩阵的输入共有 22 个；
3. 位于 VDD3P3_CPU 电源域和 VDD3P3_RTC 电源域的管脚由 IE、OE、WPU 和 WPD 信号控制；
4. 仅有部分外设输出信号 (0 ~ 6, 63~ 68) 可通过 IO MUX 直连管脚。可以实现直连功能的信号见表 5-1；

5. 从 GPIO 交换矩阵到 IO MUX 的输出共有 22 个，对应 GPIO X：0 ~ 21

图 5-3 展示了芯片焊盘的内部结构，即芯片逻辑与 GPIO 管脚之间的电气接口。22 个 GPIO 管脚均采用这一结构，且由 IE、OE、WPU 和 WPD 信号控制。

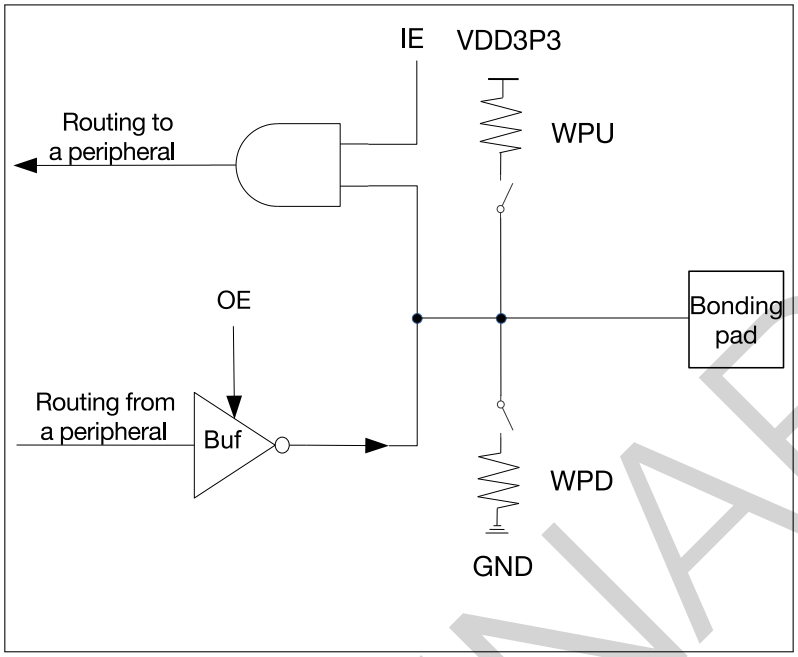


图 5-3. 焊盘内部结构

说明：

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

5.4 通过 GPIO 交换矩阵的外设输入

5.4.1 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 5-1。并需要配置外设输入选择通过 GPIO 交换矩阵接收输入信号。

5.4.2 信号同步

如图 5-2 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。

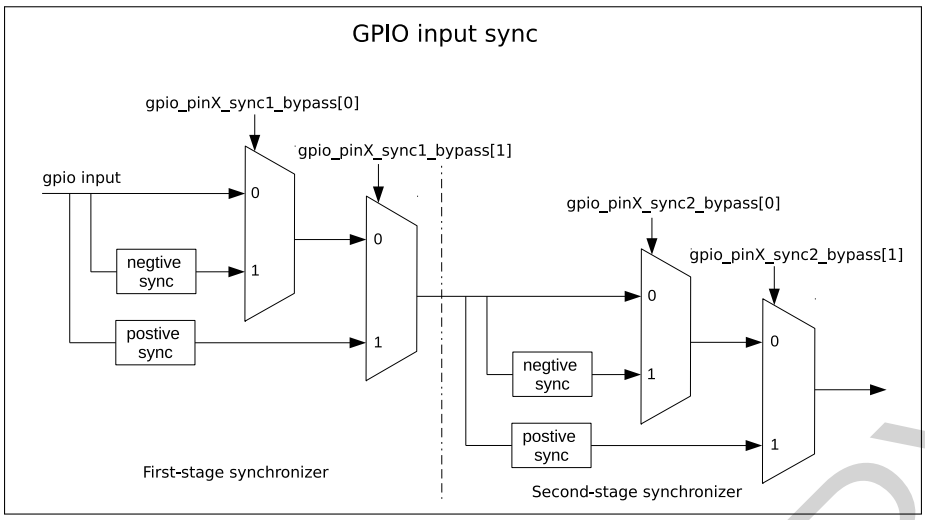


图 5-4. GPIO 输入经 APB 时钟上升沿或下降沿同步

GPIO SYNC 模块的功能如图 5-4 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。

5.4.3 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 5-5 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。

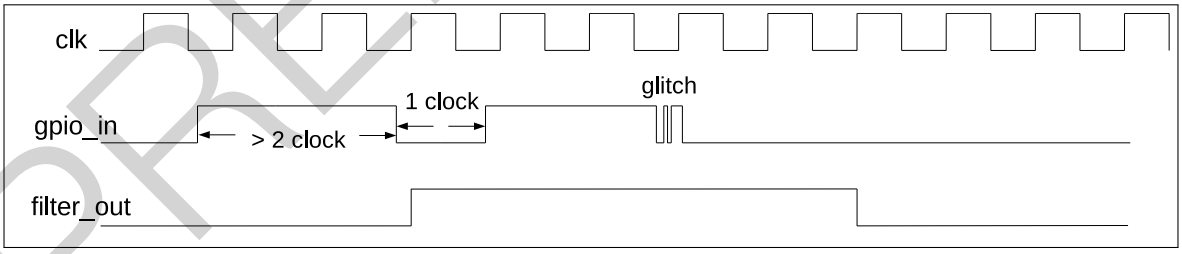


图 5-5. GPIO 输入信号滤波时序图

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 5-4 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 5-4 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]。

- 置位或清零 IO_MUX_GPIOx_FUN_WPU 和 IO_MUX_GPIOx_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 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_FUNCy_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_FUNCy_IN_SEL 输入值而不是一个 GPIO 序号：
   - 设置 GPIO_FUNCy_IN_SEL 为 0x1F，则输入信号恒为 0；
   - 设置 GPIO_FUNCy_IN_SEL 为 0x1E，则输入信号恒为 1。

5.4.4 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_GPIOx_REG 中 IO_MUX_GPIOx_FUN_IE 位以使能输入，如章节 5.4.2 所述。

5.5 通过 GPIO 交换矩阵的外设输出

5.5.1 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 5-1。

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）。

5.5.2 功能描述

如图 5-2 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_FUNCx_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_FUNC*x*_OUT_SEL_CFG_REG 寄存器的 GPIO_FUNC*x*_OUT_SEL 字段为外设输出信号 Y 的索引号 (Y)。
- 要将信号强制使能为输出模式，需要将 GPIO 管脚 *X* 对应的 GPIO_FUNC*x*_OUT_SEL_CFG_REG 寄存器中 GPIO_FUNC*x*_OEN_SEL 字段置位；同时需要将 GPIO_ENABLE_W1TS_REG 中的相应位置位。或者，将 GPIO_FUNC*x*_OEN_SEL 清零，即选择采用外设的输出使能信号，此时输出使能信号由内部逻辑功能决定。比如，表 5-1 中 “GPIO_FUNC*n*_OEN_SEL = 0 时输出信号的输出使能信号” 一栏的 SPIQ_oe 信号。
- 置位 GPIO_ENABLE_W1TC_REG 中相应位可以关闭 GPIO 管脚的输出。

2. 要选择以开漏方式输出，可以设置 GPIO 管脚 *X* 的 GPIO_PIN*x*_REG 寄存器中 GPIO_PIN*x*_PAD_DRIVER 位。
3. 配置 IO MUX 寄存器来选择经由 GPIO 交换矩阵输出信号。配置 GPIO 管脚 *X* 的 IO_MUX_GPIO*x*_REG 的过程如下：
   - 配置 GPIO 管脚 *X* 的 IO_MUX_GPIO*x*_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_GPIO*x*_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_GPIO*x*_FUN_WPU 和 IO_MUX_GPIO*x*_FUN_WPD 使能或关闭上拉/下拉电阻。

说明：

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

5.5.3 简单 GPIO 输出

GPIO 交换矩阵也可用于简单 GPIO 输出，具体配置如下：

- 设置 GPIO 交换矩阵 GPIO_FUNC*n*_OUT_SEL 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。

5.5.4 Sigma Delta 调制输出 (SDM)

5.5.4.1 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。

5.5.4.2 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 5.5.2 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。

5.6 IO MUX 的直接输入输出功能

5.6.1 概述

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

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

5.6.2 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 5.11 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。

5.7 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 5-4 列出了 ESP32-C3 管脚的模拟功能。

5.8 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。

5.9 GPIO 管脚供电和电源管理

5.9.1 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。

5.9.2 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

5.10 外设信号列表

表 5-1 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 5-1 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、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 ~ 39、45 ~ 59 和 63 ~ 127 的输出信号有效。

表 5-1. 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

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	-	-	-	-	1'd1	no
41	-	-	-	-	1'd1	no
42	-	-	-	-	1'd1	no
43	-	-	-	-	1'd1	no
44	-	-	-	-	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

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

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

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

5.11 IO MUX 管脚功能列表

表 5-2 列出了所有 GPIO 管脚的 IO MUX 功能。

表 5-2. IO MUX 管脚功能

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

驱动强度

“驱动强度” 一栏所示为每个管脚复位后的默认驱动强度。

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

复位

“复位” 一栏所示为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_DIS_PAD_JTAG = 1，则 MTCK 管脚复位后浮空，即 IE = 1。如果 EFUSE_DIS_PAD_JTAG = 0，则 MTCK 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 5-4。
- **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 5-3。

表 5-3. 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

5.12 IO MUX 管脚模拟功能列表

表 5-4 列出了具有模拟功能的 IO MUX 管脚。

表 5-4. 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	ADC1_CH0
1	XTAL_32K_N	XTAL_32K_N	ADC1_CH1
2	GPIO2	-	ADC1_CH2
3	GPIO3	-	ADC1_CH3
4	MTMS	-	ADC1_CH4

说明

1.VDD_SPI 管脚可配置为电源，也可以配置成普通的 GPIO。具体配置可参考章节 7 *芯片 Boot 控制*。

2.GPIO18 和 GPIO19 可配置为 USB 管脚。具体配置，可参考章节 5 *USB Serial/JTAG 控制器 (USB_SERIAL_JTAG) [to be added later]*。

5.13 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。

5.13.1 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
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

名称	描述	地址	访问
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
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

5.13.2 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

名称	描述	地址	访问
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
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

5.13.3 SDM 寄存器列表

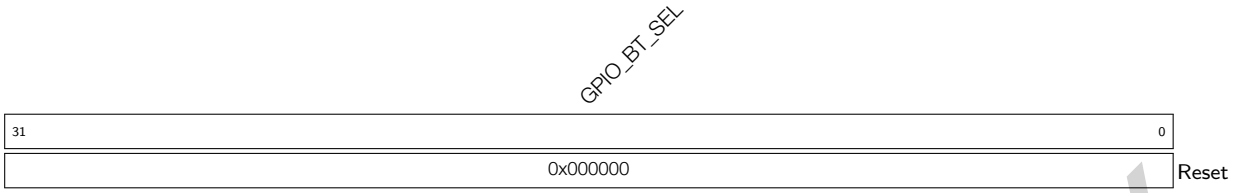
名称	描述	地址	访问
配置寄存器			
GPIOSD_SIGMADELTA0_REG	SDM0 占空比配置寄存器	0x0000	R/W
GPIOSD_SIGMADELTA1_REG	SDM1 占空比配置寄存器	0x0004	R/W
GPIOSD_SIGMADELTA2_REG	SDM2 占空比配置寄存器	0x0008	R/W
GPIOSD_SIGMADELTA3_REG	SDM3 占空比配置寄存器	0x000C	R/W
GPIOSD_SIGMADELTA_CG_REG	时钟门控配置寄存器	0x0020	R/W
GPIOSD_SIGMADELTA_MISC_REG	MISC 寄存器	0x0024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

5.14 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。

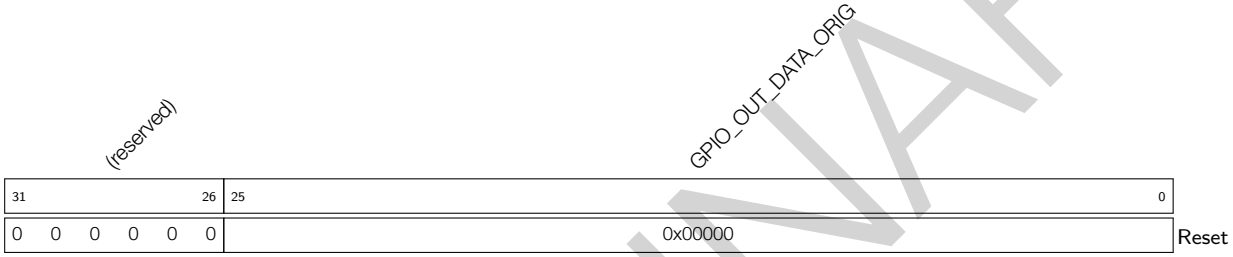
5.14.1 GPIO 交换矩阵寄存器

Register 5.1. GPIO_BT_SELECT_REG (0x0000)



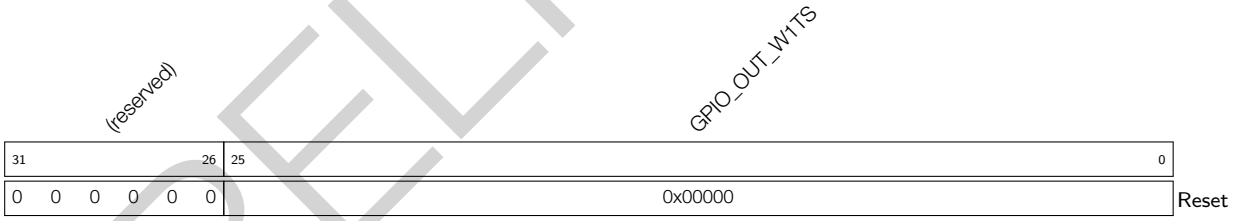
GPIO_BT_SEL 保留 (R/W)

Register 5.2. GPIO_OUT_REG (0x0004)



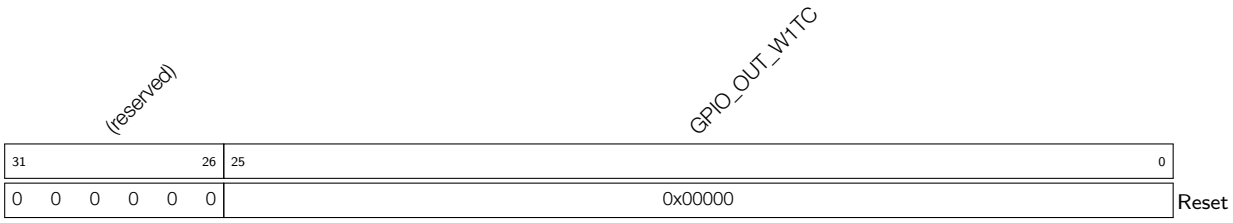
GPIO_OUT_DATA_ORIG 简单 GPIO 输出模式下 GPIO0 ~ 21 的输出值。bit0 ~ bit21 的值分别对应 GPIO0 ~ GPIO21 的值。bit22 ~ bit25 无效。(R/W/SS)

Register 5.3. GPIO_OUT_W1TS_REG (0x0008)



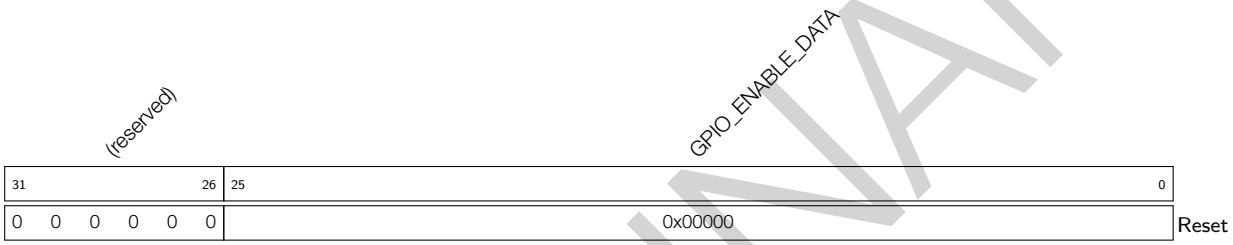
GPIO_OUT_W1TS GPIO0 ~ 21 输出置位寄存器，bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位置 1，则 GPIO_OUT_REG 中相应位也将置 1。注：推荐使用此寄存器来置位 GPIO_OUT_REG。(WT)

Register 5.4. GPIO_OUT_W1TC_REG (0x000C)



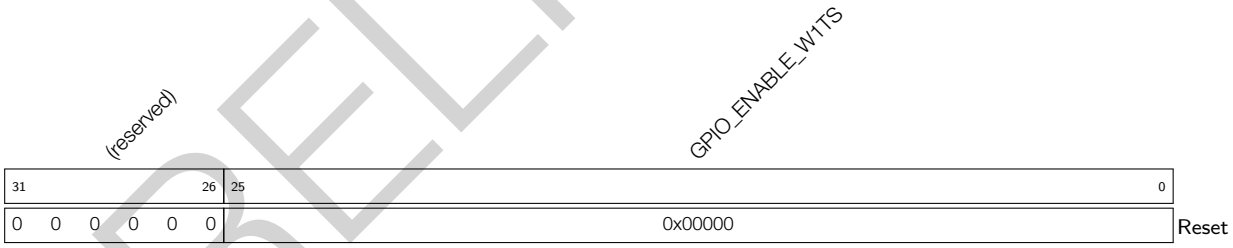
GPIO_OUT_W1TC GPIO0～21 输出清零寄存器，bit0～bit21 对应 GPIO0～21，bit22～bit25 无效。每一位置 1，则 GPIO_OUT_REG 中相应位会清零。注：推荐使用此寄存器来清零 GPIO_OUT_REG。(WT)

Register 5.5. GPIO_ENABLE_REG (0x0020)



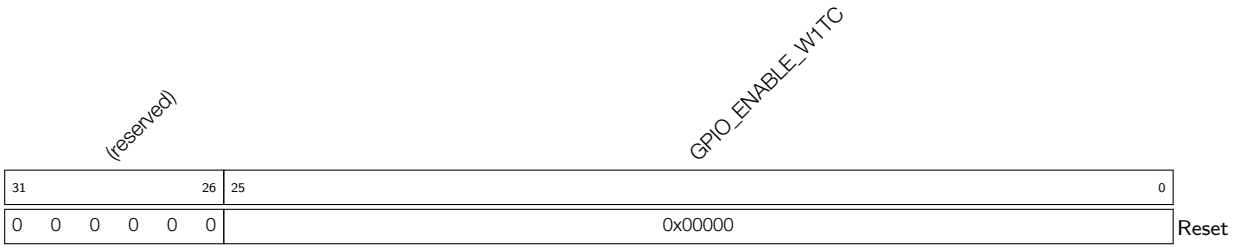
GPIO_ENABLE_DATA GPIO0 ~ 21 输出使能寄存器，bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。(R/W/SS)

Register 5.6. GPIO_ENABLE_W1TS_REG (0x0024)



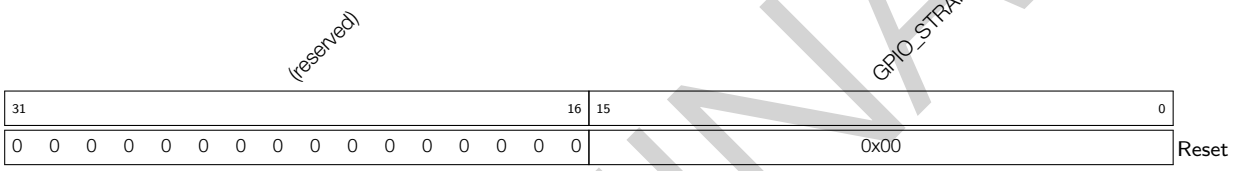
GPIO_ENABLE_W1TS GPIO0 ~ 21 输出使能置位寄存器。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位置 1，则 GPIO_ENABLE_REG 中相应位也将置 1。注：推荐使用此寄存器来置位 GPIO_ENABLE_REG。(WT)

Register 5.7. 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)

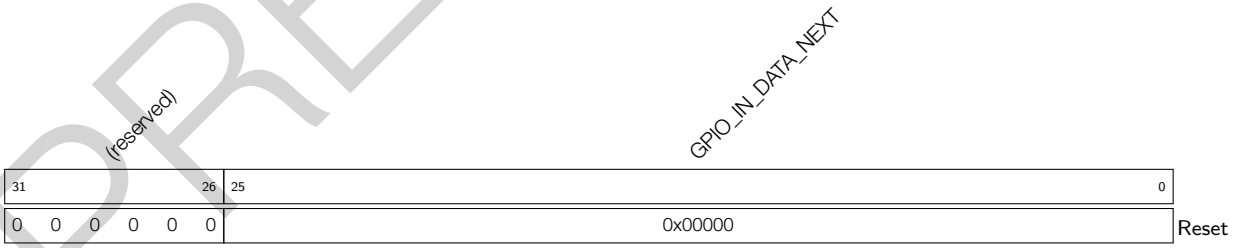
Register 5.8. GPIO_STRAP_REG (0x0038)



GPIO_STRAPPING GPIO Strapping 值。(RO)

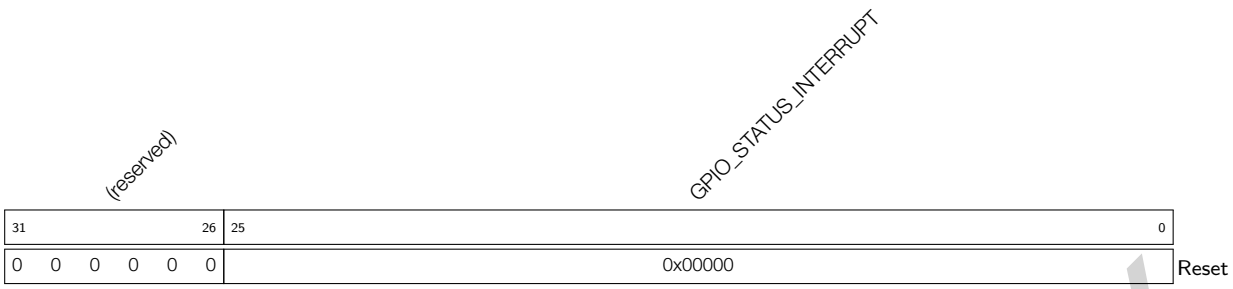
- bit 0：对应 GPIO2
- bit 2：对应 GPIO8
- bit 3：对应 GPIO9

Register 5.9. GPIO_IN_REG (0x003C)



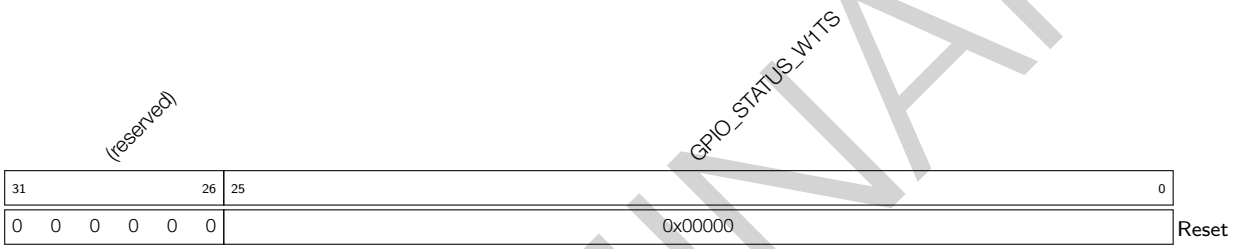
GPIO_IN_DATA_NEXT GPIO0 ~ 21 输入值。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位代表一个管脚的片外输入值，0 表示低电平，1 表示高电平。(RO)

Register 5.10. GPIO_STATUS_REG (0x0044)



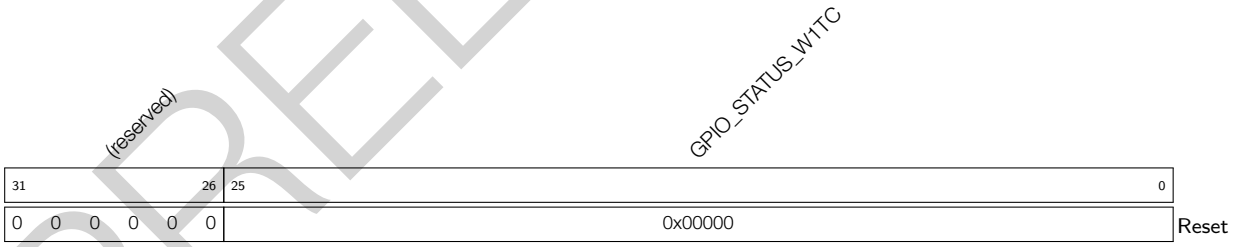
GPIO_STATUS_INTERRUPT GPIO0 ~ 21 中断状态寄存器。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。(R/W/SS)

Register 5.11. GPIO_STATUS_W1TS_REG (0x0048)



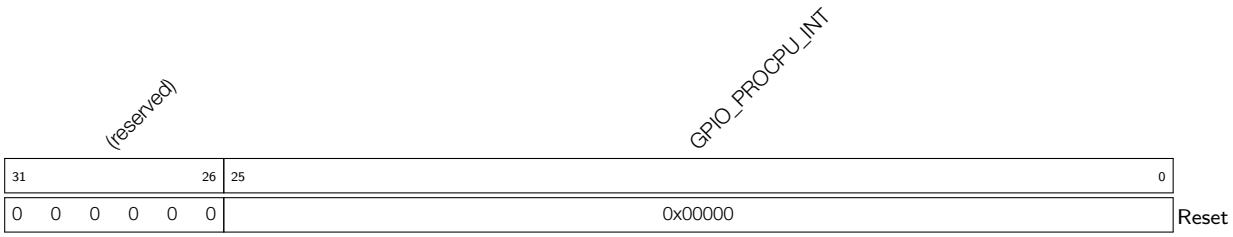
GPIO_STATUS_W1TS GPIO0 ~ 21 中断状态置位寄存器。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位置 1，则 GPIO_STATUS_INTERRUPT 中相应位也将置 1。注：推荐使用此寄存器来置位 GPIO_STATUS_INTERRUPT。(WT)

Register 5.12. GPIO_STATUS_W1TC_REG (0x004C)



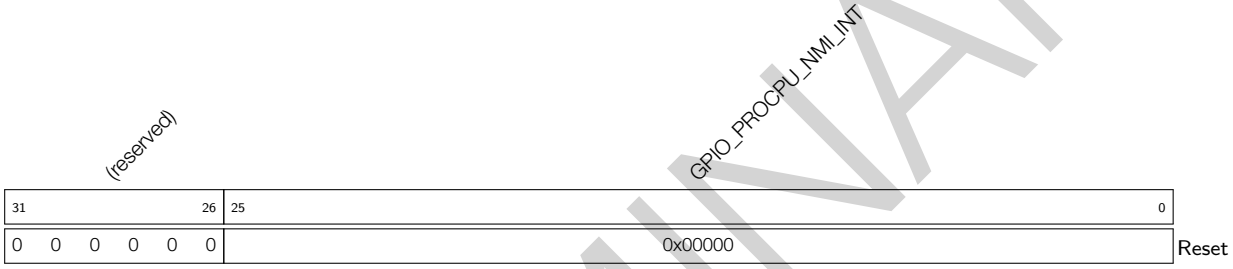
GPIO_STATUS_W1TC GPIO0 ~ 21 中断状态清除寄存器。bit0 ~ bit21 对应 GPIO0 ~ 21，bit22 ~ bit25 无效。每一位置 1，则 GPIO_STATUS_INTERRUPT 中相应位会清零。注：推荐使用此寄存器来清零 GPIO_STATUS_INTERRUPT。(WT)

Register 5.13. 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)

Register 5.14. 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 5.15. 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)

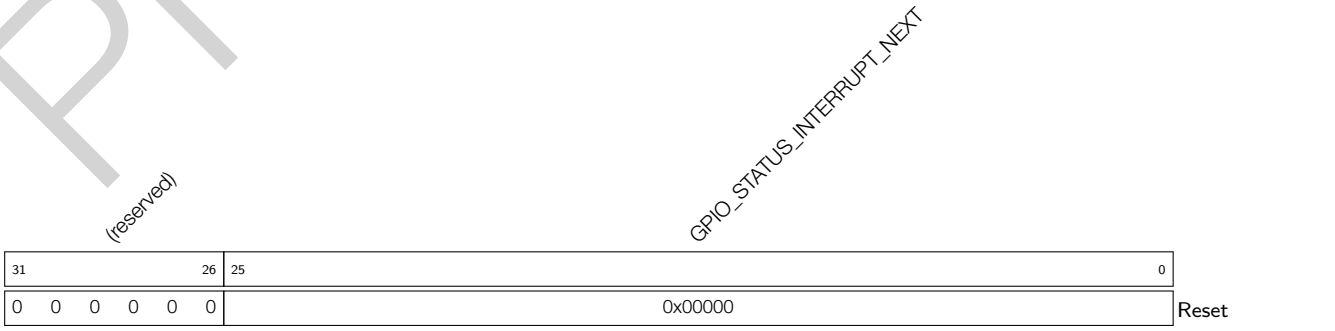
- 0：禁用 GPIO 中断
- 1：上升沿触发
- 2：下降沿触发
- 3：任一沿触发
- 4：低电平触发
- 5：高电平触发

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)

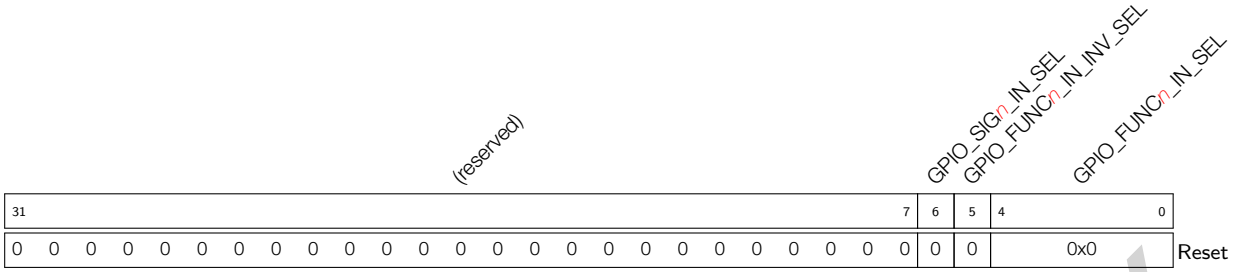
GPIO_PIN*n*_INT_ENA 中断使能位。bit13：使能 CPU 中断；bit14：使能 CPU 非屏蔽中断。(R/W)

Register 5.16. 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 5.17. GPIO_FUNC*n*_IN_SEL_CFG_REG (*n*: 0-127) (0x0154+4n*)**

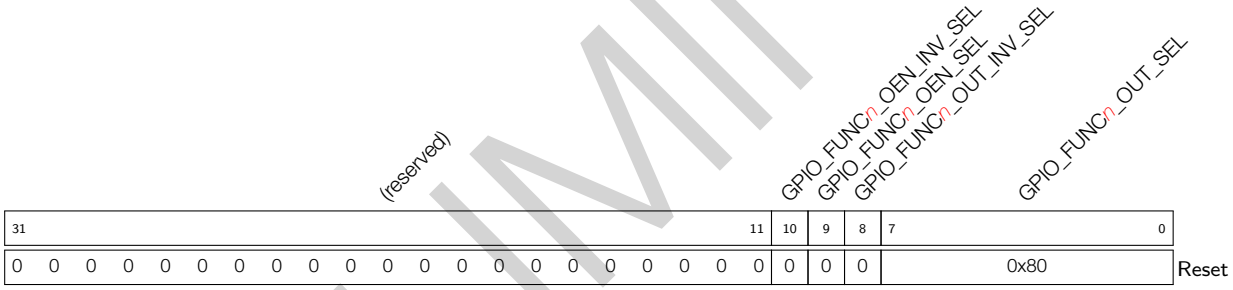


GPIO_FUNC*n*_IN_SEL 外设输入信号 *n* 的选择控制位。此位选择 1 个 GPIO 交换矩阵输入管脚与信号连接，或者选择 0x1E 与恒高电平输入信号连接，或者选择 0x1F 与恒低电平输入信号连接。(R/W)

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)

Register 5.18. GPIO_FUNC*n*_OUT_SEL_CFG_REG (*n*: 0-21) (0x0554+4n*)**



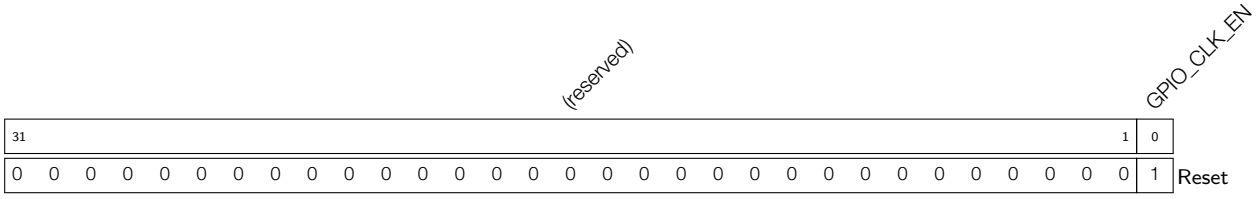
GPIO_FUNC*n*_OUT_SEL GPIO 管脚输出 *n* 的选择控制位。如果该字段设置为 *Y* (0<=Y<128)，则外设输出信号 *Y* 将连接至 GPIO *n* 输出。如果该字段设置为 128，则寄存器 GPIO_OUT_REG 和 GPIO_ENABLE_REG 中的 bit*n* 将用作输出值和输出使能。(R/W)

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)

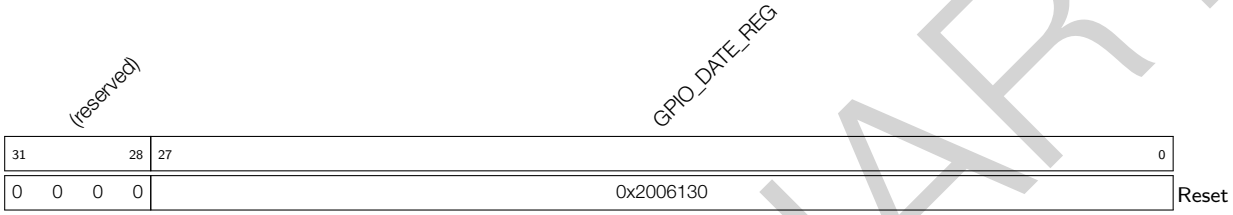
GPIO_FUNC*n*_OEN_INV_SEL 0：不反转输出使能信号；1：反转输出使能信号。(R/W)

Register 5.19. 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)

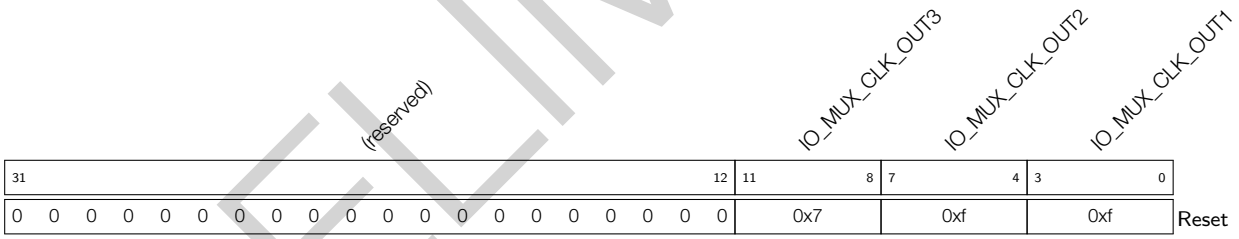
Register 5.20. GPIO_DATE_REG (0x06FC)



GPIO_DATE_REG 版本控制寄存器。(R/W)

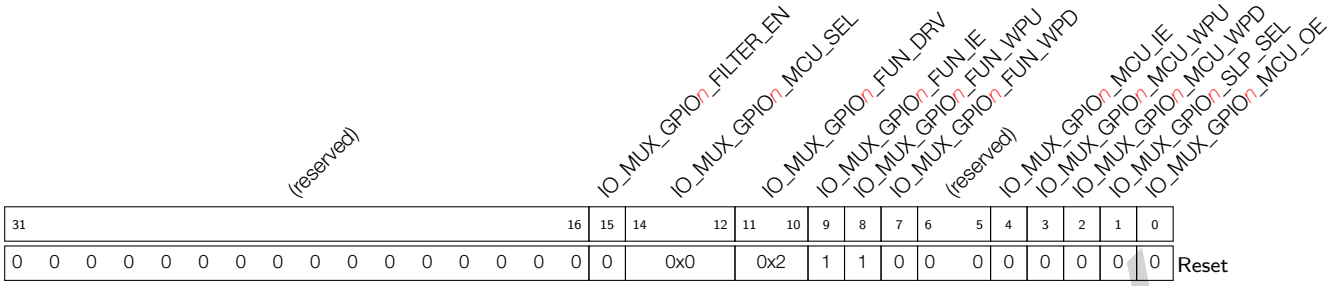
5.14.2 IO MUX 寄存器

Register 5.21. 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 5-1。(R/W)

Register 5.22. IO_MUX_GPIO*n*_REG (*n*: 0-21) (0x0004+4*n*)



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)

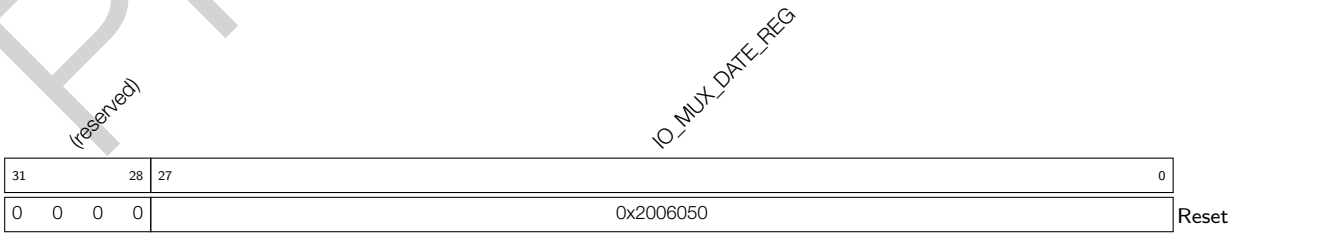
IO_MUX_GPIO*n*_FUN_IE 管脚的输入使能位。1：使能输入；0：关闭输入。(R/W)

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)

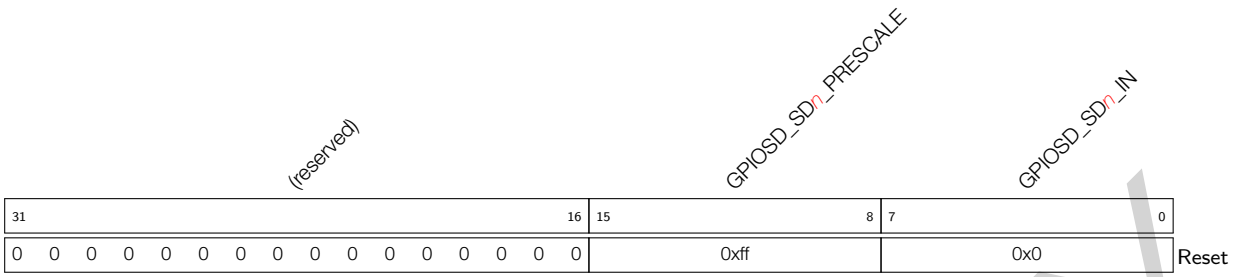
Register 5.23. IO_MUX_DATE_REG (0x00FC)



IO_MUX_DATE_REG 版本控制寄存器。(R/W)

5.14.3 SDM 寄存器

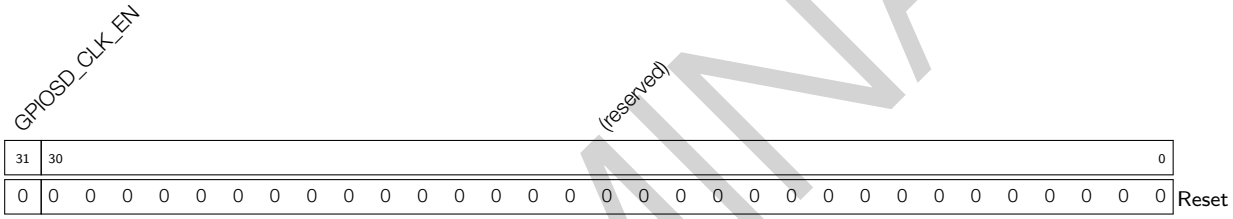
Register 5.24. GPIOSD_SIGMADELTA*n*_REG (*n*: 0-3) (0x0000+4n*)**



GPIOSD_SD*n*_IN 配置 SDM 输出信号的占空比。(R/W)

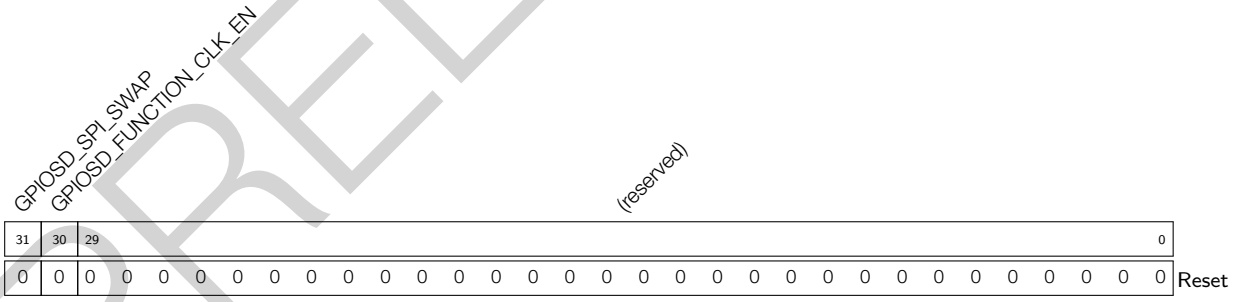
GPIOSD_SD*n*_PRESCALE 配置 APB_CLK 分频系数。(R/W)

Register 5.25. GPIOSD_SIGMADELTA_CG_REG (0x0020)



GPIOSD_CLK_EN 使能 SDM 配置寄存器的时钟。(R/W)

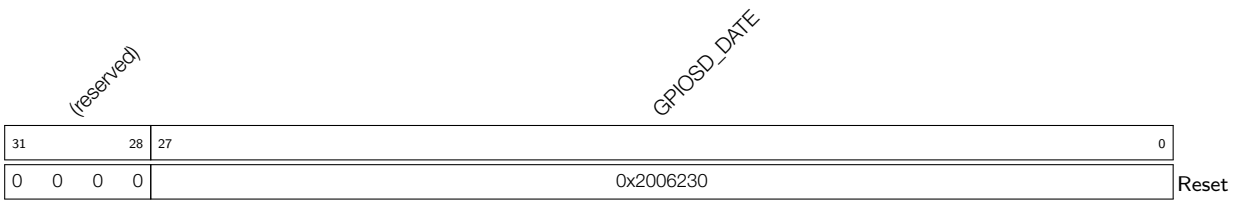
Register 5.26. GPIOSD_SIGMADELTA_MISC_REG (0x0024)



GPIOSD_FUNCTION_CLK_EN 使能 SDM 的时钟。(R/W)

GPIOSD_SPI_SWAP 保留。(R/W)

Register 5.27. GPIOSD_SIGMADELTA_VERSION_REG (0x0028)



GPIOSD_DATE 版本控制寄存器。(R/W)

6 复位和时钟

6.1 复位

6.1.1 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 6-1 展示了整个芯片系统的结构以及四种复位等级。

6.1.2 结构图

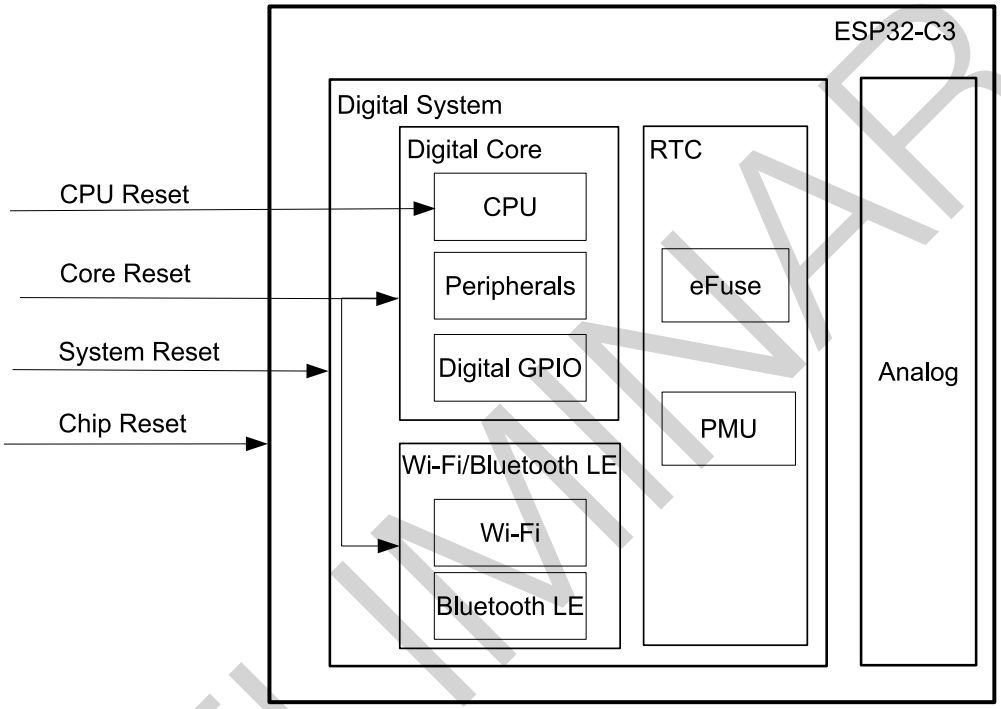


图 6-1. 四种复位等级

6.1.3 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，见章节 12 低功耗管理 *(RTC_CNTL) [to be added later]*；
  - 硬件复位：硬件复位直接由硬件电路触发。

说明：
如果 CPU 发生复位，则 SENSITIVE 寄存器也将复位。

6.1.4 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 6-1 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 6-1. 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	详见章节 7 看门狗定时器 (WDT) [to be added later]
0x13	时钟毛刺复位	系统复位	详见章节 16 时钟毛刺检测 [to be added later]
0x12	超级看门狗复位	系统复位	详见章节 7 看门狗定时器 (WDT) [to be added later]
0x03	软件系统复位	内核复位	配置 RTC_CNTL_SW_SYS_RST 寄存器触发
0x05	Deep-sleep 复位	内核复位	详见章节 12 低功耗管理 (RTC_CNTL) [to be added later]
0x07	MWDT0 内核复位	内核复位	详见章节 7 看门狗定时器 (WDT) [to be added later]
0x08	MWDT1 内核复位	内核复位	详见章节 7 看门狗定时器 (WDT) [to be added later]
0x09	RWDT 内核复位	内核复位	详见章节 7 看门狗定时器 (WDT) [to be added later]
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	详见章节 7 看门狗定时器 (WDT) [to be added later]
0x0C	软件 CPU 复位	CPU 复位	配置 RTC_CNTL_SW_PROCPU_RST 寄存器触发
0x0D	RWDT CPU 复位	CPU 复位	详见章节 7 看门狗定时器 (WDT) [to be added later]
0x11	MWDT1 CPU 复位	CPU 复位	详见章节 7 看门狗定时器 (WDT) [to be added later]

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。详见章节 12 低功耗管理 (RTC_CNTL) [to be added later]。

6.2 时钟

6.2.1 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 6-2 为系统时钟结构。

6.2.2 结构图

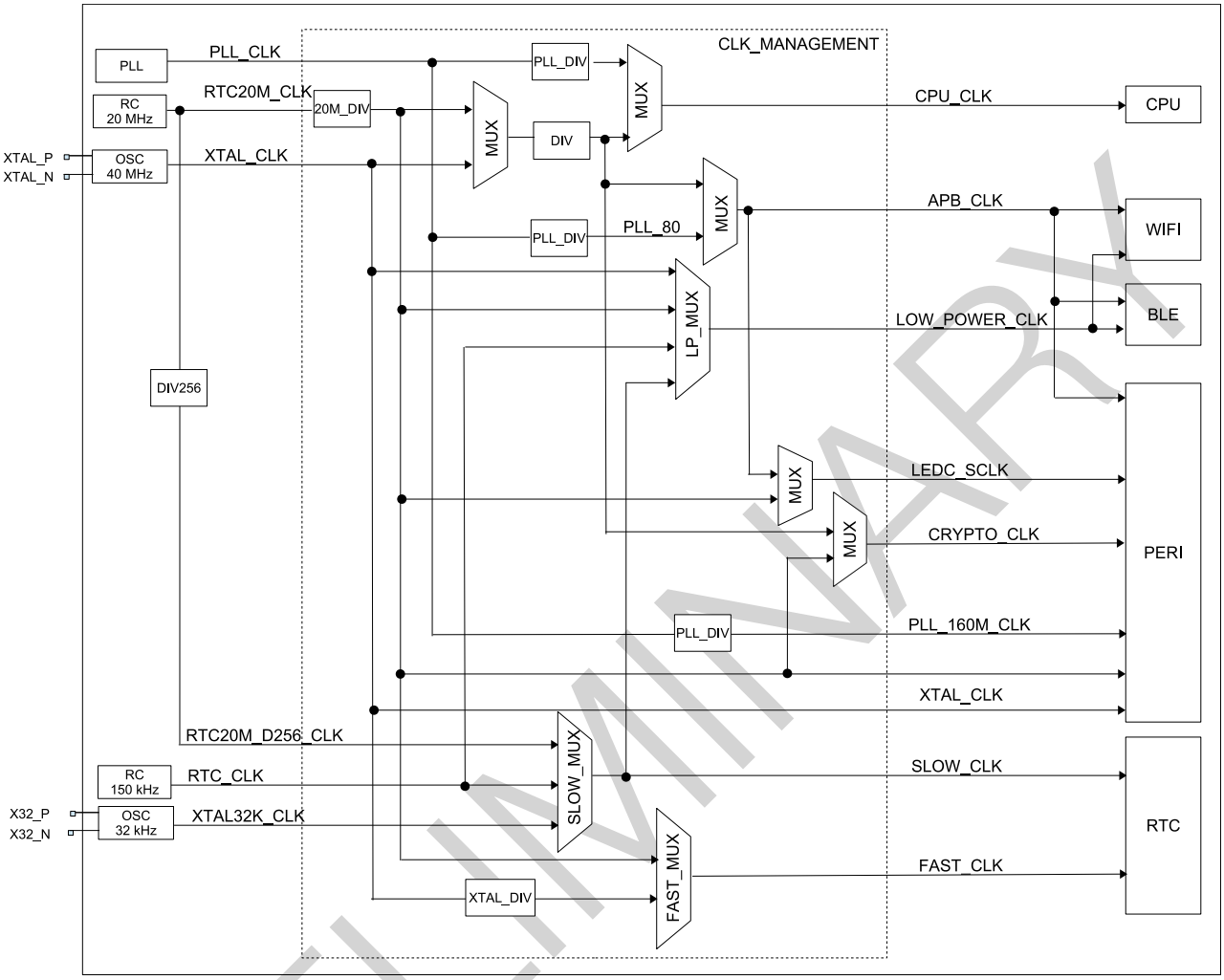


图 6-2. 系统时钟

6.2.3 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

6.2.4 功能描述

6.2.4.1 CPU 时钟

如图 6-2 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 6-2 和表 6-3。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 6-2. 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 6-3. 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。

6.2.4.2 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 6-4 列出了接入各个外设的时钟。

表 6-4. 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					

APB_CLK 时钟

如表 6-5 所示，APB_CLK 的频率由 CPU_CLK 的时钟源决定。

表 6-5. 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 6-6 所示，CRYPTO_CLK 的频率由 CPU_CLK 的时钟源决定。

表 6-6. 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。

6.2.4.3 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。

6.2.4.4 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。

7 芯片 Boot 控制

7.1 概述

ESP32-C3 共有三个 Strapping 管脚：

- GPIO2
- GPIO8
- GPIO9

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 code 日志打印到 UART

在系统复位过程中，包括上电复位、欠压复位和模拟超级看门狗复位，（请参考章节 6 复位和时钟），硬件将采样 Strapping 管脚电平存储到锁存器中，并一直保持到芯片掉电或关闭。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 7-1 所示。

表 7-1. 管脚默认上拉/下拉

管脚	默认值
GPIO2	N/A
GPIO8	N/A
GPIO9	上拉

如需改变 Strapping 管脚的默认值，用户可以应用外部下拉/上拉电阻，或者应用主机 MCU 的 GPIO 来控制 ESP32-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。

7.2 Boot 模式控制

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

表 7-2. 系统启动模式

启动模式	GPIO2	GPIO8	GPIO9
SPI Boot 模式	1	x	1
Download Boot 模式	1	1	0

表 7-2 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。

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 flash 中，或将程序加载到 SRAM 并在 SRAM 中运行程序。

下面几个 eFuse 可用于控制启动模式的具体行为：

- EFUSE_DIS_FORCE_DOWNLOAD

  如果此 eFuse 设置为 0（默认），软件可通过设置 RTC_CNTL_FORCE_DOWNLOAD_BOOT，触发 CPU 复位，将芯片启动模式强制从 SPI Boot 模式切换至 Download Boot 模式；如果此 eFuse 设置为 1，则禁用 RTC_CNTL_FORCE_DOWNLOAD_BOOT。

- EFUSE_DIS_DOWNLOAD_MODE

  如果此 eFuse 设置为 1，则禁用 Download Boot 模式。

- EFUSE_ENABLE_SECURITY_DOWNLOAD

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

USB Serial/JTAG 控制器可将芯片从 SPI Boot 模式强制切换到 Download Boot 模式，或从 Download Boot 模式强制切换到 SPI Boot 模式。更多信息，请参考 *5 USB Serial/JTAG 控制器 (USB_SERIAL_JTAG) [to be added later]* 章节。

7.3 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 7-3. ROM 代码打印控制

eFuse[1]	GPIO8	ROM 代码打印
0	x	启动过程中，ROM 代码始终打印至 UART，此时 GPIO8 的值被忽略
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	x	启动过程中始终关闭打印，此时 GPIO8 的值被忽略

[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

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 定时器组 (TIMG)

8.1 概述

通用定时器可用于准确设定时间间隔、在一定间隔后触发（周期或非周期的）中断或充当硬件时钟。如图 8-1 所示，ESP32-C3 包含两个定时器组，即定时器组 0 和定时器组 1。每个定时器组有一个通用定时器（下文用 T0 表示）和一个主系统看门狗定时器。所有通用定时器均基于 16 位预分频器和 54 位可自动重新加载的可逆计数器。

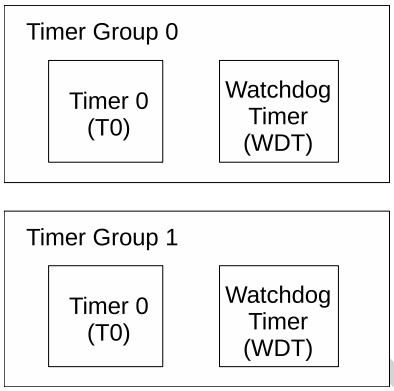


图 8-1. 定时器组

本章包含主系统看门狗定时器的寄存器描述，其功能描述请参阅章节 *7 看门狗定时器 (WDT) [to be added later]*。本章中“定时器”指代通用定时器。

定时器具有如下功能：

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

8.2 功能描述

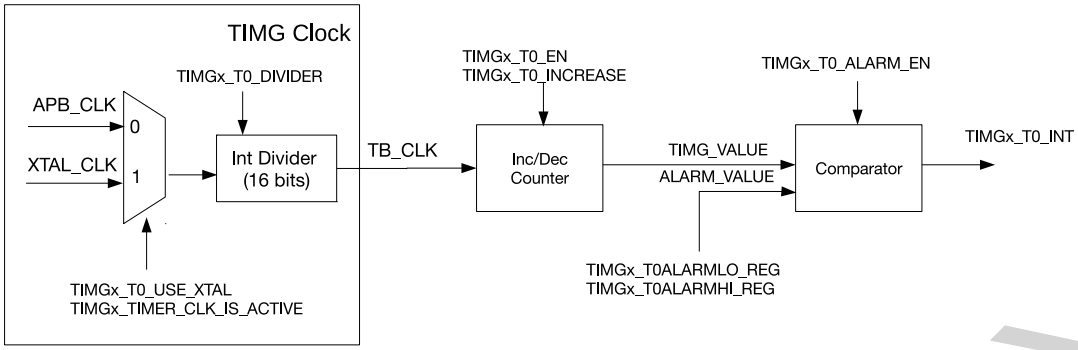


图 8-2. 定时器组架构

图 8-2 为定时器组的 T0。T0 包含一个时钟选择器、一个 16 位整数预分频器、一个时基计数器和一个用于产生警报的比较器。

8.2.1 16 位预分频器与时钟选择器

每个定时器可通过配置寄存器 TIMG_T0CONFIG_REG 的 TIMG_T0_USE_XTAL 字段，选择 APB 时钟 (APB_CLK) 或外部时钟 (XTAL_CLK) 作为时钟源。要开启所选时钟，需置位 TIMG_REGCLK_REG 寄存器的 TIMG_TIMER_CLK_IS_ACTIVE 字段，要关闭时钟需清零该字段。所选时钟经 16 位预分频器分频，产生时基计数器使用的时基计数器时钟 (TB_CLK)。16 位预分频器的分频系数可通过 TIMG_T0_DIVIDER 字段配置，选取从 2 到 65536 之间的任意值。注意，将 TIMG_T0_DIVIDER 置 0 后，分频系数会变为 65536。TIMG_T0_DIVIDER 置 1 时，实际分频系数为 2，计数器的值为实际时间的一半。

要更改 16 位预分频器，需先重新配置TIMG_T0_DIVIDER 字段，再将TIMG_T0_DIVIDER_RST 置 1，同时需要关闭定时器（即 TIMG_T0_EN 必须清零），否则会造成不可预知的结果。

8.2.2 54 位时基计数器

54 位时基计数器基于 TB_CLK，可通过 TIMG_T0_INCREASE 字段配置为递增或递减。时基计数器可通过置位或清零 TIMG_T0_EN 字段使能或关闭。使能时，时基计数器的值会在每个 TB_CLK 周期递增或递减。关闭时，时基计数器暂停计数。注意，TIMG_T0_EN 置位后，TIMG_T0_INCREASE 字段还可以更改，时基计数器可立即改变计数方向。

时基计数器 54 位定时器的当前值必须被锁入两个寄存器，才能被 CPU 读取（因为 CPU 为 32 位）。在 TIMG_T0UPDATE_REG 上写任意值，54 位定时器的值可立即锁入寄存器 TIMG_T0LO_REG 和 TIMG_T0HI_REG，两个寄存器分别锁存低 32 位和高 22 位。在 TIMG_T0UPDATE_REG 被写入新值之前，寄存器 TIMG_T0LO_REG 和 TIMG_T0HI_REG 的值保持不变，以便 CPU 读值。

8.2.3 报警产生

定时器可配置为在当前值与报警值相同时触发报警。报警会产生中断，（可选择）让定时器的当前值自动重新加载（详见第 8.2.4节）。

54 位报警值可在 TIMG_T0ALARMLO_REG 和 TIMG_T0ALARMHI_REG 配置，两者分别代表报警值的低 32 位和高 22 位。但是，只有置位 TIMG_T0_ALARM_EN 字段使能报警功能后，配置的报警值才会生效。为解决报警使能“过晚”（即报警使能时，定时器的值已过报警值），可逆计数器向上计数时，若定时器的当前值高于报警值

(在一定范围内)，或可逆计数器向下计数时，定时器的当前值低于报警值 (在一定范围内)，硬件都会立即触发报警。表 8-1 和表 8-2 说明了定时器当前值、报警值与报警触发的关系。假设定时器当前值和报警值如下：

- TIMG_VALUE = {TIMG_T0HI_REG, TIMG_T0LO_REG}
- ALARM_VALUE = {TIMG_T0ALARMHI_REG, TIMG_T0ALARMLO_REG}

表 8-1. 可逆计数器向上计数时的报警触发条件

Scenario	Range	Alarm
1	ALARM_VALUE − TIMG_VALUE $> 2^{53}$	触发
2	$0 <$ ALARM_VALUE − TIMG_VALUE $\leq 2^{53}$	可逆计数器向上计数，TIMG_VALUE 达到 ALARM_VALUE 时报警
3	$0 \leq$ TIMG_VALUE − ALARM_VALUE $< 2^{53}$	触发
4	TIMG_VALUE − ALARM_VALUE $\geq 2^{53}$	可逆计数器向上计数达到最大值时，重新开始从 0 向上计数，TIMG_VALUE 达到 ALARM_VALUE 时触发报警

表 8-2. 可逆计数器向下计数时的报警触发情景

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

报警时，TIMG_T0_ALARM_EN 字段自动清零，在下次置位 TIMG_T0_ALARM_EN 前不会再次报警。

8.2.4 定时器重新加载

定时器重新加载指将定时器的低 32 位和高 22 位分别更新为寄存器 TIMG_T0_LOAD_LO 和 TIMG_T0_LOAD_HI 存储的重新加载值。但是，把重新加载值写入 TIMG_T0_LOAD_LO 和 TIMG_T0_LOAD_HI 寄存器不会改变定时器的当前值。写入的重新加载值会被定时器忽视，直到重新加载事件被触发。重新加载事件可由软件即时重新加载或报警时自动重新加载触发。

CPU 在寄存器 TIMG_T0LOAD_REG 写任意值会触发软件即时重新加载，定时器的当前值会立即改变。如置位 TIMG_T0_EN ，定时器会继续从新数值开始递增或递减计数。如清零 TIMG_T0_EN ，定时器将保持当前值，直至计数重新使能。

报警时自动重新加载功能可让定时器在报警时重新加载，从重新加载值开始继续递增或递减计数。该功能通常用于周期性报警时重置定时器的值。要使能报警时自动重新加载，需将 TIMG_T0_AUTORELOAD 字段置 1。如未使能该功能，报警后定时器的值会在过报警值后继续递增或递减。

8.2.5 低功耗时钟 (SLOW_CLK) 频率计算

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

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

8.2.6 中断

每个定时器都有一根连接至 CPU 的中断线。因此，每个定时器组有两根中断线。定时器每次产生的电平中断必须由 CPU 清除。

电平中断在报警后（或看门狗定时器阶段超时）触发。报警（或阶段超时）后，电平中断会一直被拉高，直至手动清除中断。要使能定时器的中断，TIMG_T0_INT_ENA 需置 1。

每个定时器组的中断受一组寄存器控制。每个定时器在下列寄存器中都有对应的位：

- TIMG_T0_INT_RAW：报警时置 1。该位在写值到对应的 TIMG_T0_INT_CLR 位后才会被清零。
- TIMG_WDT_INT_RAW：阶段超时时置 1。该位在写值到对应的 TIMG_WDT_INT_CLR 位后才会被清零。
- TIMG_T0_INT_ST：反映每个定时器中断的状态，通过用 TIMG_T0_INT_ENA 屏蔽 TIMG_T0_INT_RAW 位来生成。
- TIMG_WDT_INT_ST：反映每个看门狗定时器中断的状态，通过用 TIMG_WDT_INT_ENA 屏蔽 TIMG_WDT_INT_RAW 位来生成。
- TIMG_T0_INT_ENA：用于使能或屏蔽组内定时器的中断状态位。
- TIMG_WDT_INT_ENA：用于使能或屏蔽组内看门狗定时器的中断状态位。
- TIMG_T0_INT_CLR：置 1 此位清除定时器中断，定时器在 TIMG_T0_INT_RAW 和 TIMG_T0_INT_ST 的对应位会清零。注意，下一个中断产生前，必须清除定时器中断。
- TIMG_WDT_INT_CLR：置 1 此位清除定时器中断，看门狗定时器在 TIMG_WDT_INT_RAW 和 TIMG_WDT_INT_ST 的对应位会清零。注意，下一个中断产生前，必须清除看门狗定时器中断。

8.3 配置与使用

8.3.1 定时器用作简单时钟

1. 配置时基计数器。
   - 置位或清除 TIMG_T0_USE_XTAL 字段选择时钟源。
   - 置位 TIMG_T0_DIVIDER 配置 16 位预分频器。
   - 置位或清除 TIMG_T0_INCREASE 配置定时器方向。
   - 在 TIMG_T0_LOAD_LO 和 TIMG_T0_LOAD_HI 上写初始值设置定时器的初始值，然后在 TIMG_T0LOAD_REG 上写任意值将初始值重新加载进定时器。
2. 置位 TIMG_T0_EN 开启定时器。
3. 获得定时器的当前值。
   - 在 TIMG_T0UPDATE_REG 上写任意值锁存定时器的当前值。
   - 从 TIMG_T0LO_REG 和 TIMG_T0HI_REG 读取锁存的定时器值。

8.3.2 定时器用于单次报警

1. 按照第 8.3.1 节的第 1 步配置时基计数器。
2. 配置报警。
   - 置位 TIMG_T0ALARMLO_REG 和 TIMG_T0ALARMHI_REG 配置报警值。
   - 置位 TIMG_T0_INT_ENA 使能中断。
3. 清零 TIMG_T0_AUTORELOAD 关闭自动重新加载。
4. 置位 TIMG_T0_ALARM_EN 开启报警。
5. 处理报警中断。
   - 置位定时器在 TIMG_T0_INT_CLR 的对应位清除中断。
   - 清零 TIMG_T0_EN 关闭定时器。

8.3.3 定时器用于周期性报警

1. 按照第 8.3.1 节的第 1 步配置时基计数器。
2. 按照第 8.3.2节的第 2 步配置报警。
3. 置位 TIMG_T0_AUTORELOAD 使能自动重新加载，将重新加载值写入 TIMG_T0_LOAD_LO 和 TIMG_T0_LOAD_HI。
4. 置位 TIMG_T0_ALARM_EN 开启报警。
5. 处理报警中断（每次报警时重复）。
   - 置位定时器在 TIMG_T0_INT_CLR 的对应位清除中断。
   - 如下一次报警需要新的报警值和重新加载值（即每次都有不同的报警间隔），则应根据需要重新配置 TIMG_T0ALARMLO_REG、TIMG_T0ALARMHI_REG、TIMG_T0_LOAD_LO 和 TIMG_T0_LOAD_HI 。否则，上述寄存器应保持不变。
   - 置位 TIMG_T0_ALARM_EN 重新使能报警。
6. （最后一次报警时）关闭定时器。
   - 置位定时器在 TIMG_T0_INT_CLR 的对应位清除中断。
   - 清零 TIMG_T0_EN 关闭定时器。

8.3.4 SLOW_CLK 频率计算

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

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

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

8.4 寄存器列表

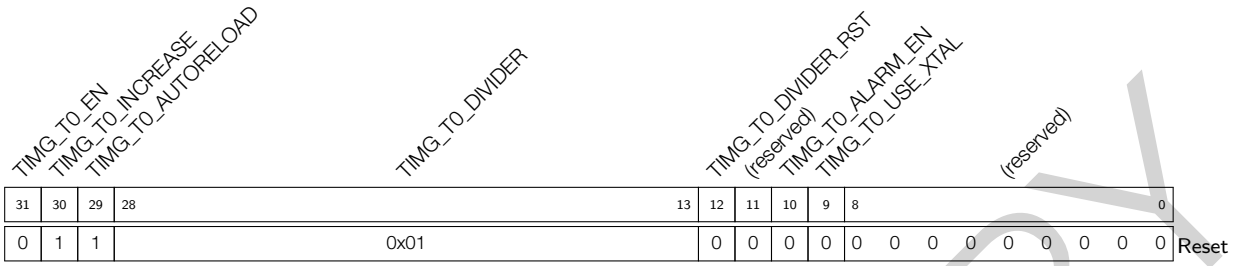
本小节的所有地址均为相对于 定时器组 基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

名称	描述	地址	访问
定时器 0 控制和配置寄存器			
TIMG_T0CONFIG_REG	定时器 0 配置寄存器	0x0000	varies
TIMG_T0LO_REG	定时器 0 的当前值，低 32 位	0x0004	RO
TIMG_T0HI_REG	定时器 0 的当前值，高 22 位	0x0008	RO
TIMG_T0UPDATE_REG	写值将当前定时器的值复制到 TIMG_T0LO_REG 或 TIMG_T0HI_REG	0x000C	R/ W/ SC
TIMG_T0ALARMLO_REG	定时器 0 的报警值，低 32 位	0x0010	R/W
TIMG_T0ALARMHI_REG	定时器 0 的报警值，高位	0x0014	R/W
TIMG_T0LOADLO_REG	定时器 0 的重新加载值，低 32 位	0x0018	R/W
TIMG_T0LOADHI_REG	定时器 0 的重新加载值，高 22 位	0x001C	R/W
TIMG_T0LOAD_REG	写值从 TIMG_T0LOADLO_REG 或 TIMG_T0LOADHI_REG 上加载定时器	0x0020	WT
看门狗定时器控制和配置寄存器			
TIMG_WDTCONFIG0_REG	看门狗定时器配置寄存器	0x0048	varies
TIMG_WDTCONFIG1_REG	看门狗定时器预分频器寄存器	0x004C	varies
TIMG_WDTCONFIG2_REG	看门狗定时器阶段 0 超时值	0x0050	R/W
TIMG_WDTCONFIG3_REG	看门狗定时器阶段 1 超时值	0x0054	R/W
TIMG_WDTCONFIG4_REG	看门狗定时器阶段 2 超时值	0x0058	R/W
TIMG_WDTCONFIG5_REG	看门狗定时器阶段 3 超时值	0x005C	R/W
TIMG_WDTFEED_REG	写值喂看门狗定时器	0x0060	WT
TIMG_WDTWPROTECT_REG	看门狗写保护寄存器	0x0064	R/W
RTC 频率计算控制和配置寄存器			
TIMG_RTCCALICFG_REG	RTC 频率计算配置寄存器 0	0x0068	varies
TIMG_RTCCALICFG1_REG	RTC 频率计算配置寄存器 1	0x006C	RO
TIMG_RTCCALICFG2_REG	RTC 频率计算配置寄存器 2	0x0080	varies
中断寄存器			
TIMG_INT_ENA_TIMERS_REG	中断使能位	0x0070	R/W
TIMG_INT_RAW_TIMERS_REG	原始中断状态	0x0074	R/ SS/ WTC
TIMG_INT_ST_TIMERS_REG	屏蔽中断状态	0x0078	RO
TIMG_INT_CLR_TIMERS_REG	中断清除位	0x007C	WT
版本寄存器			
TIMG_NTIMERS_DATE_REG	版本控制寄存器	0x00F8	R/W
时钟配置寄存器			
TIMG_REGCLK_REG	定时器组时钟门控寄存器	0x00FC	R/W

8.5 寄存器

本小节的所有地址均为相对于 定时器组 基地址的地址偏移量（定时器组 0 和定时器组 1 各自的相对地址），具体基地址请见章节 3 *系统和存储器* 中的表 3-4。

Register 8.1. TIMG_T0CONFIG_REG (0x0000)



TIMG_T0_USE_XTAL 0：使用 APB_CLK 作为定时器组的源时钟；1：使用 XTAL_CLK 作为定时器组的源时钟。(R/W)

TIMG_T0_ALARM_EN 置 1 后，报警使能。报警时，此位自动清零。(R/W/SC)

TIMG_T0_DIVIDER_RST 置 1 后，复位定时器 *x* 时钟分频器的计数器。(WT)

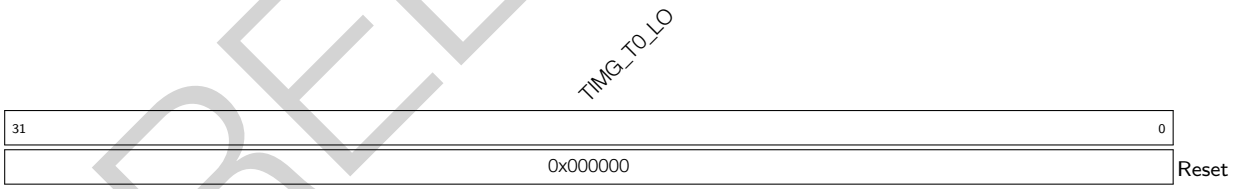
TIMG_T0_DIVIDER 定时器 *x* 时钟 (T0_clk) 的预分频器值。(R/W)

TIMG_T0_AUTORELOAD 置 1 后，定时器 *x* 报警时自动重新加载使能。(R/W)

TIMG_T0_INCREASE 置 1 后，定时器 *x* 的时基计数器会在每个时钟周期后递增。清零后，定时器 *x* 的时基计数器会递减。(R/W)

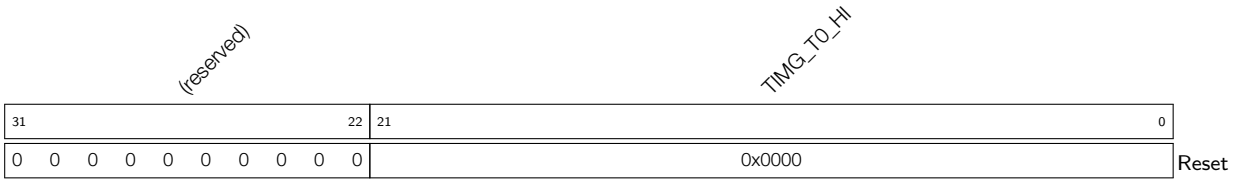
TIMG_T0_EN 置 1 后，定时器 *x* 时基计数器使能。(R/W)

Register 8.2. TIMG_T0LO_REG (0x0004)



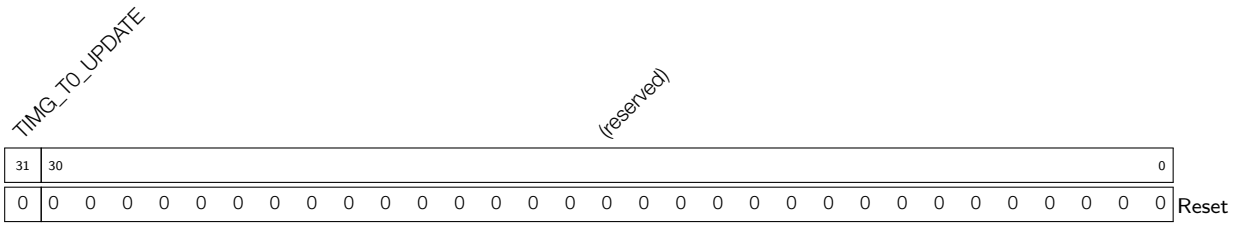
TIMG_T0_LO 在 TIMG_T0UPDATE_REG 上写值后，可读取定时器 *x* 时基计数器的低 32 位。(RO)

Register 8.3. TIMG_T0HI_REG (0x0008)



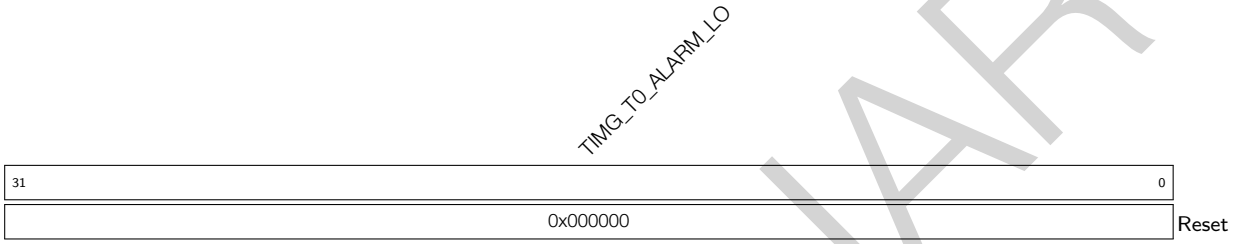
TIMG_T0_HI 在 TIMG_T0UPDATE_REG 上写值后，可读取定时器 *x* 时基计数器的高 22 位。(RO)

Register 8.4. TIMG_T0UPDATE_REG (0x000C)



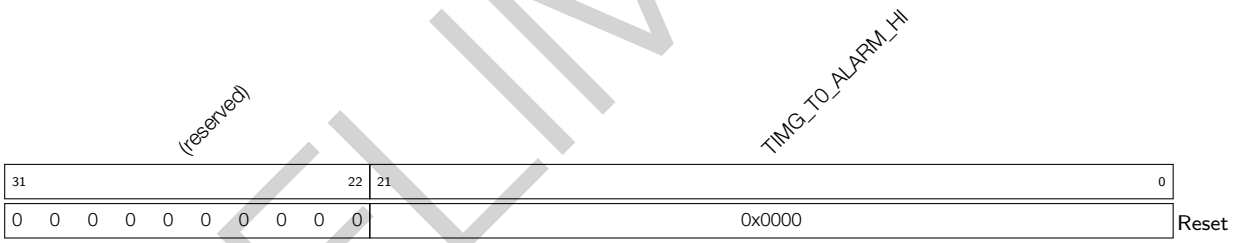
TIMG_T0_UPDATE 在 TIMG_T0UPDATE_REG 上写 0 或 1，计数器的值被锁住。(R/W/SC)

Register 8.5. TIMG_T0ALARMLO_REG (0x0010)



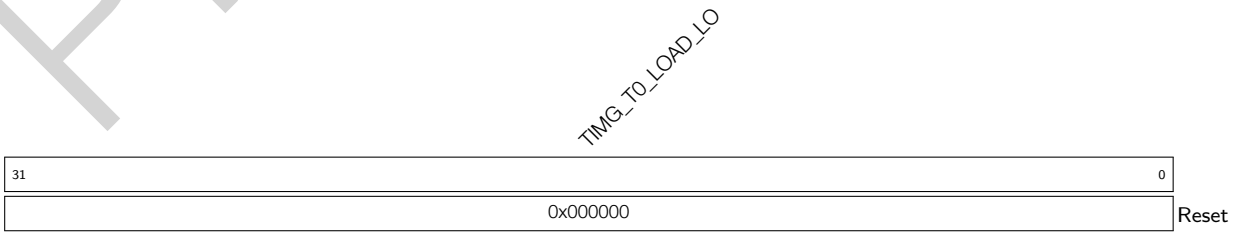
TIMG_T0_ALARM_LO 定时器 *x* 时基计数器触发警报值的低 32 位。(R/W)

Register 8.6. TIMG_T0ALARMHI_REG (0x0014)



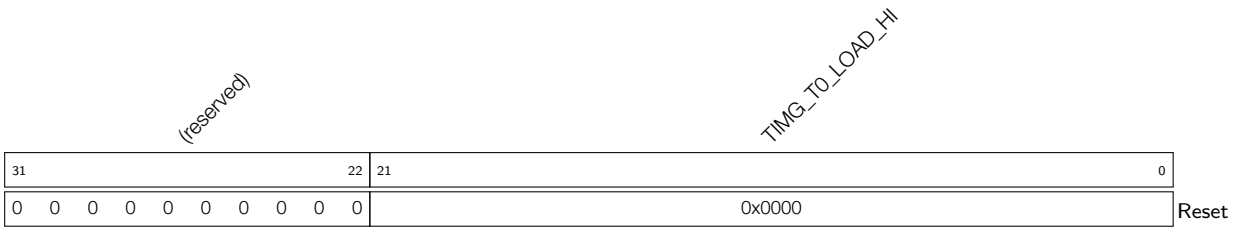
TIMG_T0_ALARM_HI 定时器 *x* 时基计数器触发警报值的高 22 位。(R/W)

Register 8.7. TIMG_T0LOADLO_REG (0x0018)



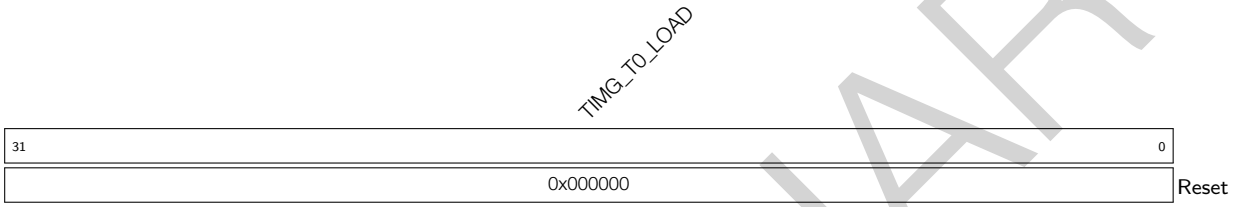
TIMG_T0_LOAD_LO 定时器 *x* 时基计数器重新加载的低 32 位值。(R/W)

Register 8.8. TIMG_T0LOADHI_REG (0x001C)



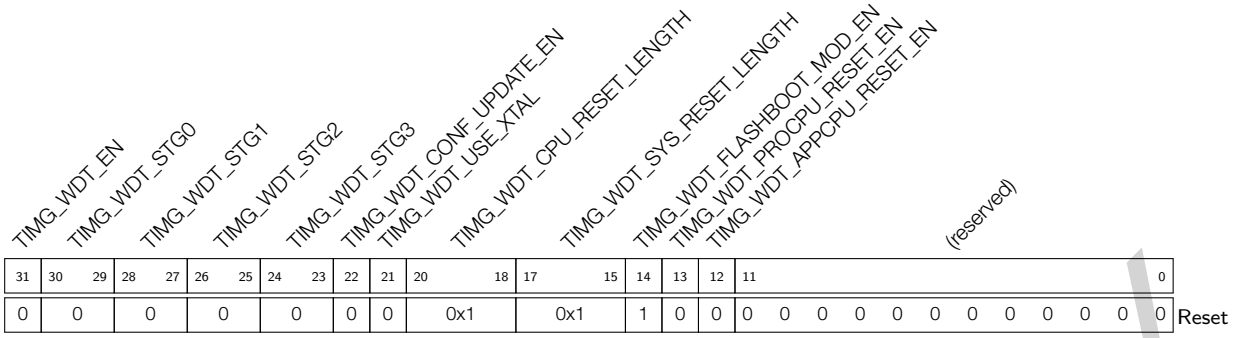
TIMG_T0_LOAD_HI 定时器 *x* 时基计数器重新加载的高 22 位值。(R/W)

Register 8.9. TIMG_T0LOAD_REG (0x0020)



TIMG_T0_LOAD 写任意值触发定时器 *x* 时基计数器重新加载。(WT)

Register 8.10. TIMG_WDTCONFIG0_REG (0x0048)



TIMG_WDT_APPCPU_RESET_EN WDT 复位 CPU 使能。(R/W)

TIMG_WDT_PROCPU_RESET_EN WDT 复位 CPU 使能。(R/W)

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

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

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

TIMG_WDT_USE_XTAL 选择看门狗定时器的时钟。0：APB_CLK；1：XTAL_CLK。(R/W)

TIMG_WDT_CONF_UPDATE_EN 更新看门狗定时器配置寄存器。(WT)

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

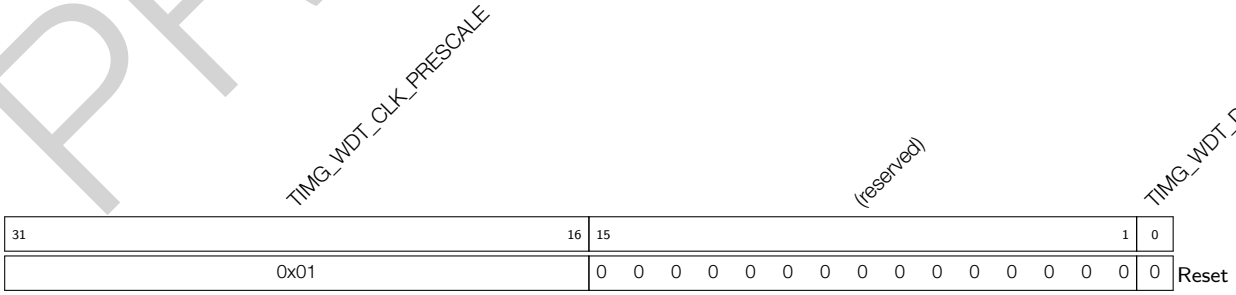
TIMG_WDT_STG2 阶段 2 配置。0：关闭；1：中断；2：复位 CPU；3：复位系统。(R/W)

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

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

TIMG_WDT_EN 置 1 后，MWDT 使能。(R/W)

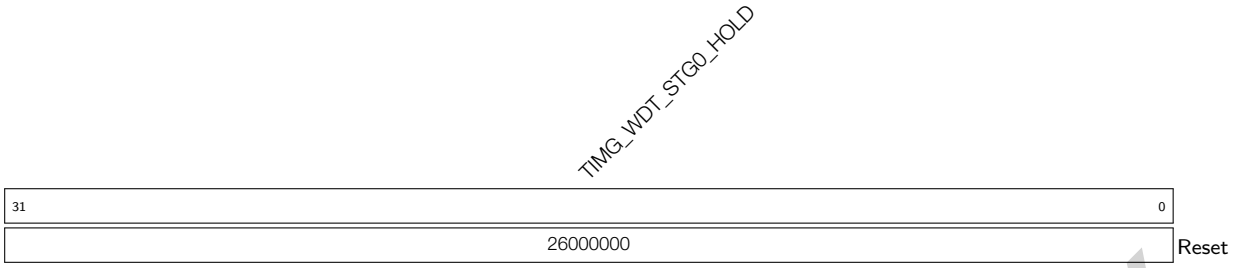
Register 8.11. TIMG_WDTCONFIG1_REG (0x004C)



TIMG_WDT_DIVCNT_RST 置 1 后，复位看门狗定时器时钟分频器的计数器。(WT)

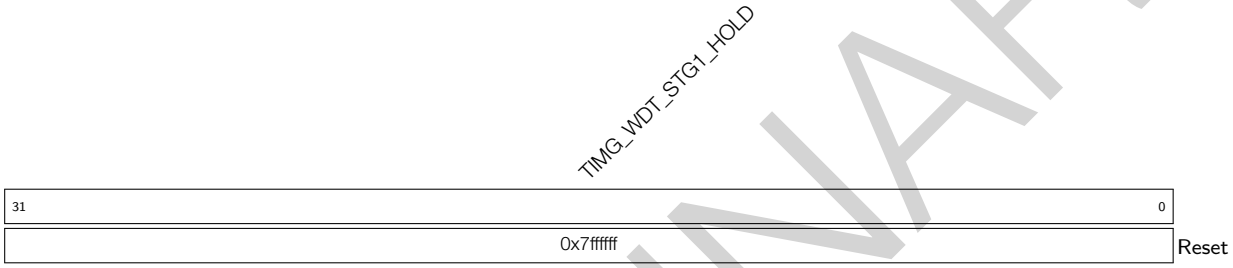
TIMG_WDT_CLK_PRESCALE MWDT 时钟预分频器值。MWDT 时钟长度 = 12.5 ns * TIMG_WDT_CLK_PRESCALE。(R/W)

Register 8.12. TIMG_WDTCONFIG2_REG (0x0050)



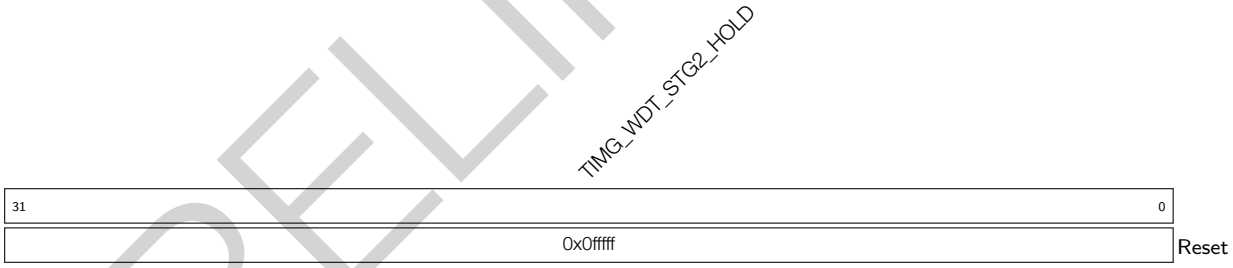
TIMG_WDT_STG0_HOLD 阶段 0 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 8.13. TIMG_WDTCONFIG3_REG (0x0054)



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

Register 8.14. TIMG_WDTCONFIG4_REG (0x0058)

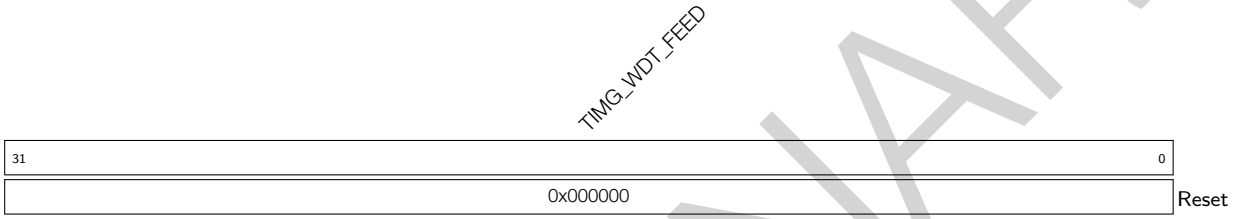


TIMG_WDT_STG2_HOLD 阶段 2 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 8.15. TIMG_WDTCONFIG5_REG (0x005C)

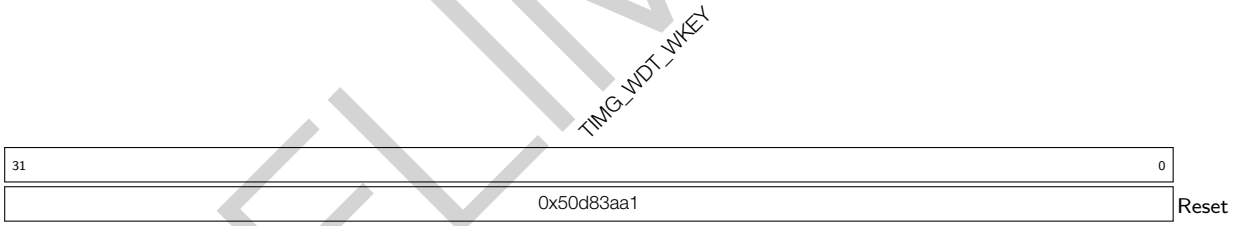
TIMG_WDT_STG3_HOLD 阶段 3 超时时间，单位是 MWDT 时钟周期。(R/W)

Register 8.16. TIMG_WDTFEED_REG (0x0060)



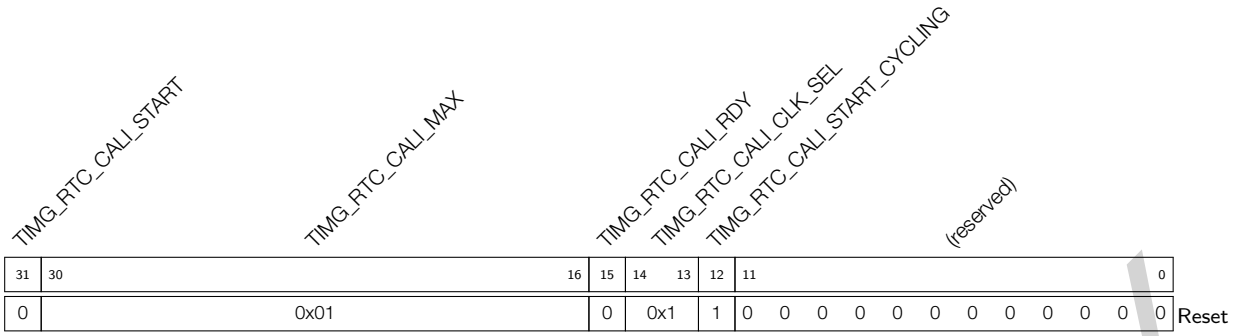
TIMG_WDT_FEED 写任意值喂 MWDT。(WT)

Register 8.17. TIMG_WDTWPROTECT_REG (0x0064)



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

Register 8.18. TIMG_RTCCALICFG_REG (0x0068)



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

TIMG_RTC_CALI_CLK_SEL 0：RTC_CLK；1：RTC20M_D256_CLK；2：XTAL32K_CLK。(R/W)

TIMG_RTC_CALI_RDY 标记频率计算完成。(RO)

TIMG_RTC_CALI_MAX 配置频率计算时间。(R/W)

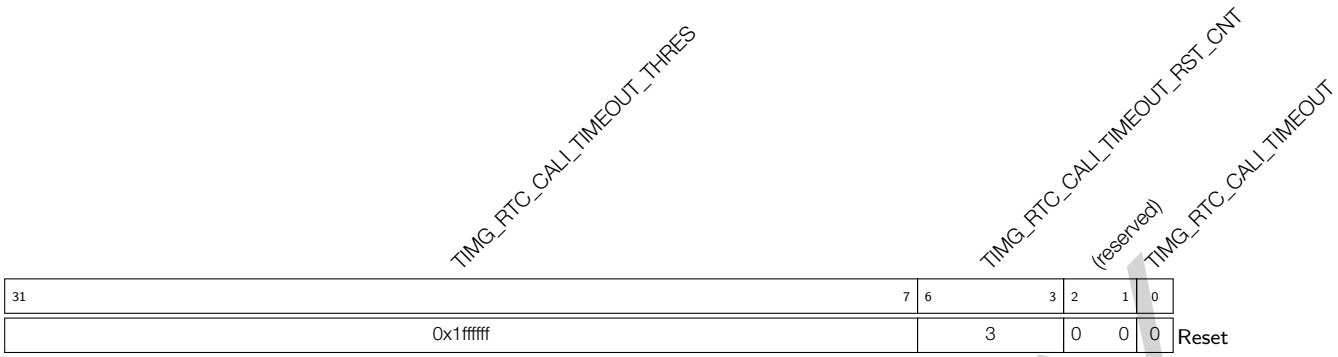
TIMG_RTC_CALI_START 使能单次频率计算。(R/W)

Register 8.19. TIMG_RTCCALICFG1_REG (0x006C)

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

TIMG_RTC_CALI_VALUE 频率计算结果。(RO)

Register 8.20. TIMG_RTCCALICFG2_REG (0x0080)

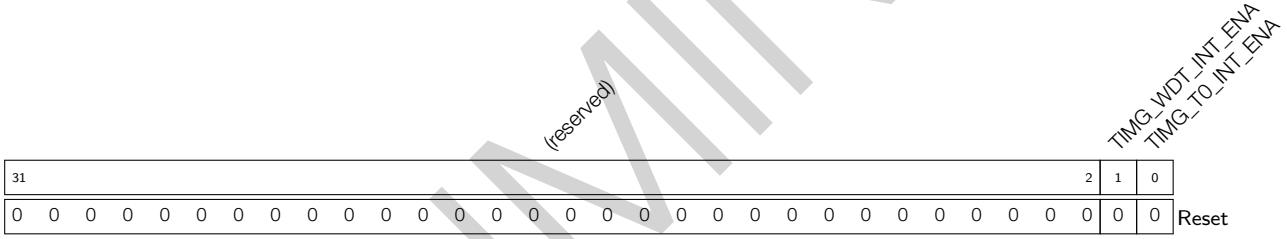


TIMG_RTC_CALI_TIMEOUT 提示时钟频率计算超时。(RO)

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

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

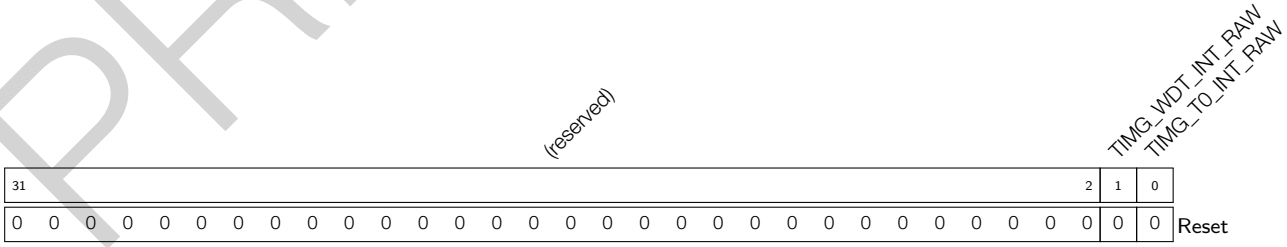
Register 8.21. TIMG_INT_ENA_TIMERS_REG (0x0070)



TIMG_T0_INT_ENA TIMG_T0_INT 中断的使能位。(R/W)

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

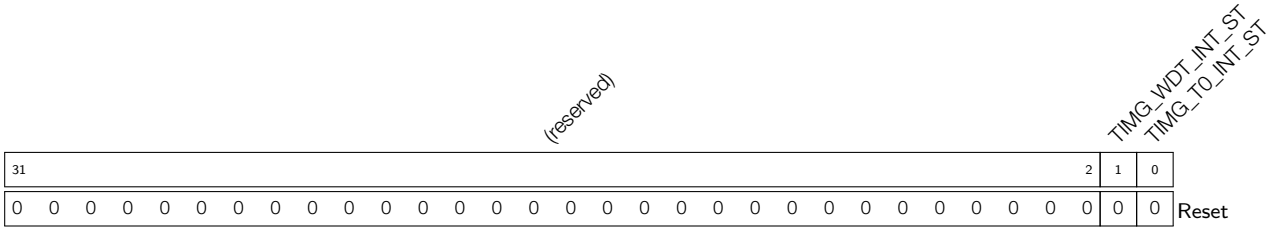
Register 8.22. TIMG_INT_RAW_TIMERS_REG (0x0074)



TIMG_T0_INT_RAW TIMG_T0_INT 中断的原始中断状态位。(R/SS/WTC)

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

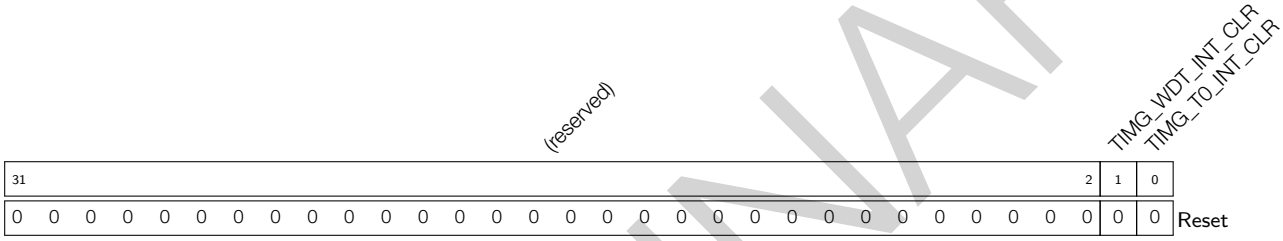
Register 8.23. TIMG_INT_ST_TIMERS_REG (0x0078)



TIMG_T0_INT_ST TIMG_T0_INT 中断的屏蔽中断状态位。(RO)

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

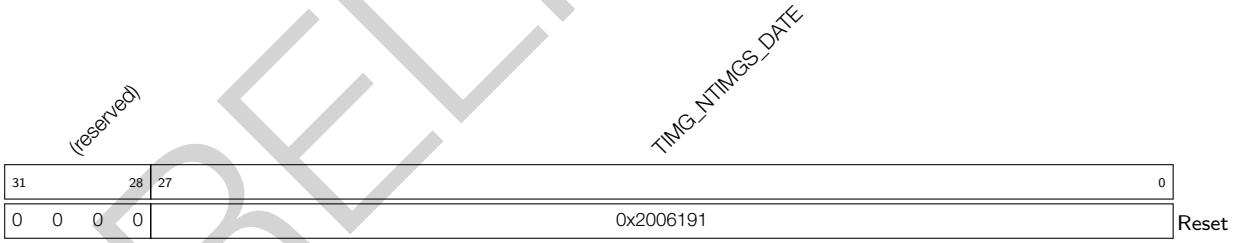
Register 8.24. TIMG_INT_CLR_TIMERS_REG (0x007C)



TIMG_T0_INT_CLR 置位此位，清除 TIMG_T0_INT 中断。(WT)

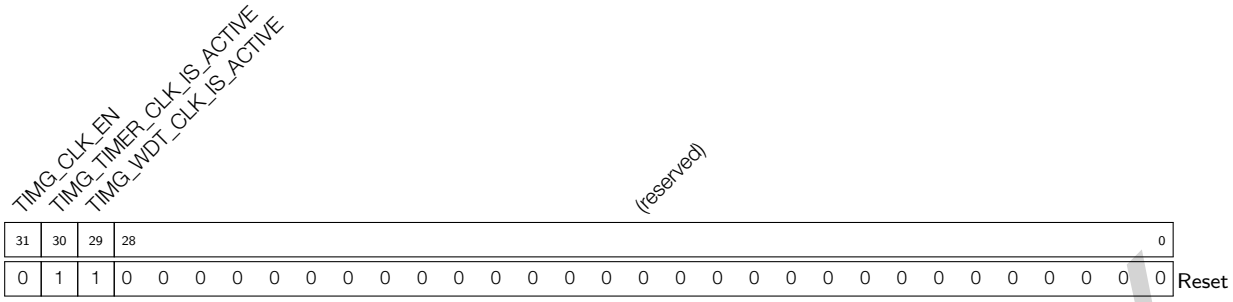
TIMG_WDT_INT_CLR 置位此位，清除 TIMG_WDT_INT 中断。(WT)

Register 8.25. TIMG_NTIMERS_DATE_REG (0x00F8)



TIMG_NTIMGS_DATE 版本控制寄存器。(R/W)

Register 8.26. TIMG_REGCLK_REG (0x00FC)



TIMG_WDT_CLK_IS_ACTIVE 使能看门狗定时器的时钟。(R/W)

TIMG_TIMER_CLK_IS_ACTIVE 使能定时器 *x* 的时钟。(R/W)

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

9 SHA 加速器 (SHA)

9.1 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。

9.2 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）

9.3 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 9-1。

表 9-1. 工作模式选择

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

用户可通过配置 SHA_MODE_REG 寄存器选择 SHA 加速器的运算标准，具体请见表 9-2。

表 9-2. 运算标准选择

哈希运算标准	SHA_MODE_REG 的配置
SHA-1	0
SHA-224	1
SHA-256	2

注意：

ESP32-C3 的数字签名 (DS) [to be added later] 和 HMAC 模块也会调用 SHA 加速器。此时，用户无法正常访问 SHA 加速器。

9.4 功能描述

SHA 加速器可以提取信息摘要 (message digest)，其主要工作流程分为两步：信息预处理和哈希运算。

9.4.1 信息预处理

信息预处理分为三个主要步骤：附加填充比特、信息解析和设置初始哈希值。

9.4.1.1 附加填充比特

SHA 加速器仅能处理长度为 512 位及其整倍数的信息。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。

9.4.1.2 信息解析

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

9.4.1.3 哈希初始值 (Initial Hash Value)

在进行哈希运算前，首先必须设置哈希初始值 $H^{(0)}$，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。

9.4.2 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。

9.4.2.1 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 9-2。
2. 处理当前信息块。
   - 将当前信息块写入 SHA_M_*n*_REG 寄存器。
3. 启动 SHA 加速器 [1]。
   - 如果为首次运算，则对 SHA_START_REG 寄存器置 1，启动 SHA 加速器的运算。此时，SHA 加速器按照步骤 1 中选定的运算标准，使用硬件中固定的哈希初始值进行运算；
   - 如果非首次运算[2]，则对 SHA_CONTINUE_REG 寄存器置 1，启动 SHA 加速器的运算。此时，SHA 加速器使用 SHA_H_*n*_REG 寄存器中的值作为哈希初始值进行运算。
4. 查询当前信息块的处理进度。
   - 轮询寄存器 SHA_BUSY_REG 一直到读回的值为 0，代表 SHA 硬件加速器已完成对当前信息块的计算，进入 “空闲” 状态[3]。
5. 选择是否有后续的待处理信息块。
   - 如果存在后续待处理信息块，则跳回执行步骤 2。
   - 否则，继续执行。
6. 获取信息摘要：

- 从寄存器堆 SHA_H_*n*_REG 取出信息摘要。

说明：

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

如上文所述，ESP32-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。

9.4.2.2 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。具体配置可见章节 2 *通用 DMA 控制器 (GDMA)*。

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 9-2。
2. 选择是否启用中断。请将 SHA_INT_ENA_REG 寄存器配置为 1 以启动中断。
3. 配置块个数。
   - 将待加密数据的总块数 *M* 写入 SHA_DMA_BLOCK_NUM_REG 寄存器。
4. 开始 DMA-SHA 运算。
   - 如果当前 DMA-SHA 运算为接着另一次 DMA-SHA 的运算，需要提前将另一次计算得到的信息摘要写入寄存器堆 SHA_H_*n*_REG 中，随后将 1 写入寄存器 SHA_DMA_CONTINUE_REG;

- 否则，只需要将 1 写入寄存器 SHA_DMA_START_REG。

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

9.4.3 信息摘要存储

哈希运算完成之后，计算得到的信息摘要被 SHA 加速器更新至对应的 SHA_H_*n*_REG (*n*: 0 ~ 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 9-3：

表 9-3. 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。

9.4.4 中断

SHA 加速器在 DMA-SHA 工作模式下允许中断发生。用户可通过将 SHA_INT_ENA_REG 寄存器配置为 1 开启中断。如开启中断功能，SHA 加速器在完成运算时，中断发生。注意，该中断必须由软件将 SHA_INT_CLEAR_REG 寄存器置为 1 进行清除。由于 SHA 加速器在 Typical SHA 工作模式下的时间开销较小，因此不支持中断功能。

9.5 寄存器列表

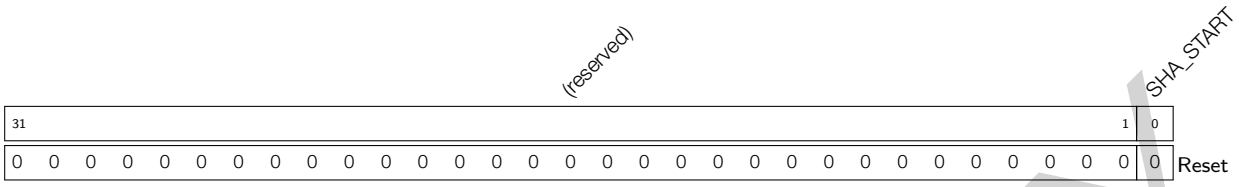
本小节的所有地址均为相对于 SHA 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

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

9.6 寄存器

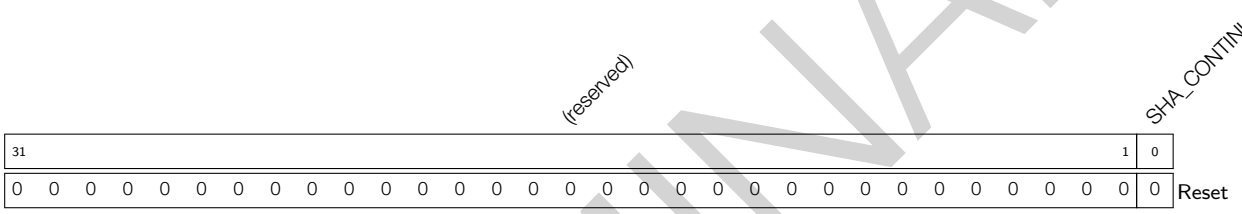
本小节的所有地址均为相对于 SHA 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

Register 9.1. SHA_START_REG (0x0010)



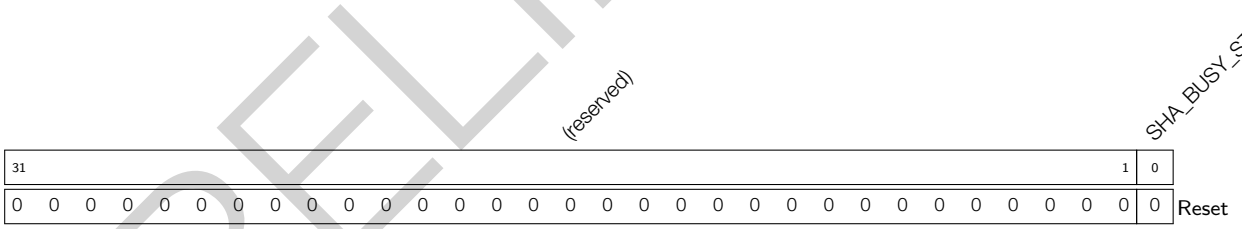
SHA_START 置 1 启动 SHA 加速器的 Typical SHA 模式。（只写）

Register 9.2. SHA_CONTINUE_REG (0x0014)



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

Register 9.3. SHA_BUSY_REG (0x0018)

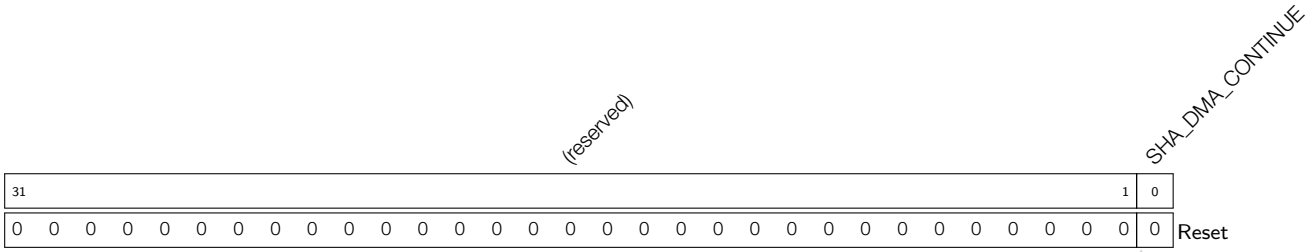


SHA_BUSY_STATE 指示 SHA 是否处于“忙碌”状态。（只读）1'h0: 空闲 1'h1: 忙碌

Register 9.4. SHA_DMA_START_REG (0x001C)

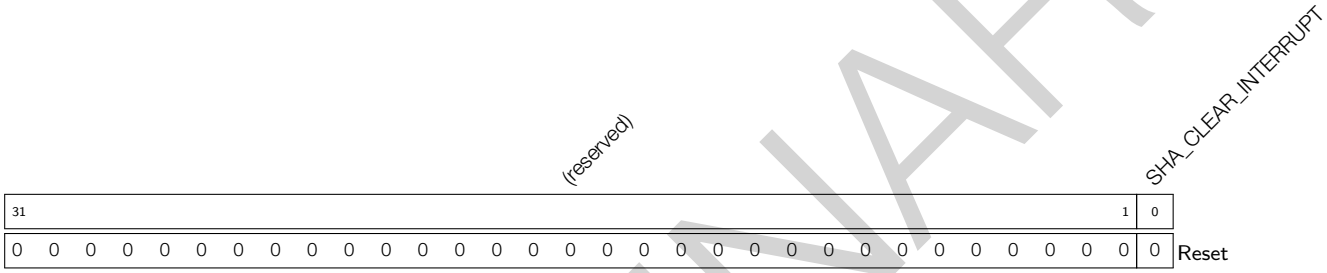
SHA_DMA_START 置 1 启动 SHA 加速器的 DMA-SHA 模式。（只写）

Register 9.5. SHA_DMA_CONTINUE_REG (0x0020)



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

Register 9.6. SHA_INT_CLEAR_REG (0x0024)

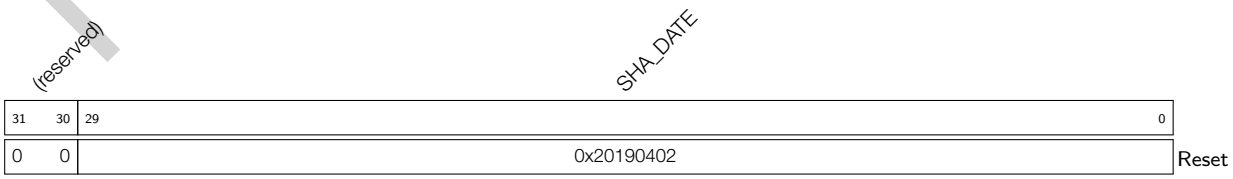


SHA_CLEAR_INTERRUPT 清除 DMA-SHA 中断。（只写）

Register 9.7. SHA_INT_ENA_REG (0x0028)

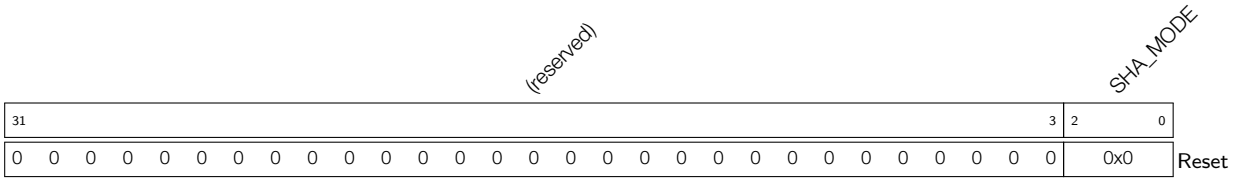
SHA_INTERRUPT_ENA 使能 DMA-SHA 中断。（读写）

Register 9.8. SHA_DATE_REG (0x002C)



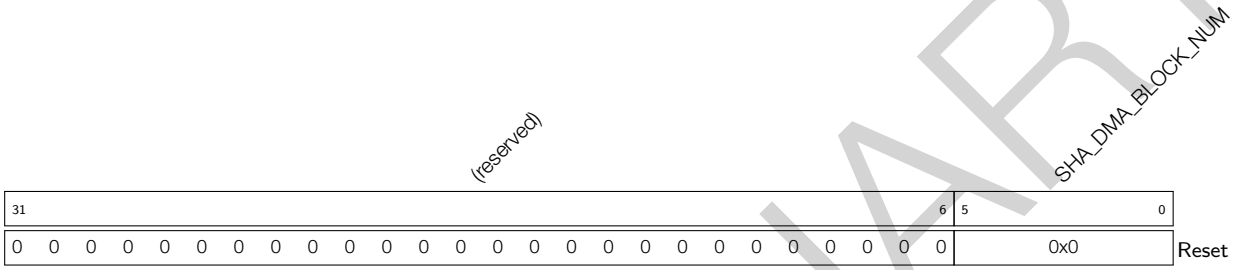
SHA_DATE 版本控制寄存器。（读写）

Register 9.9. SHA_MODE_REG (0x0000)



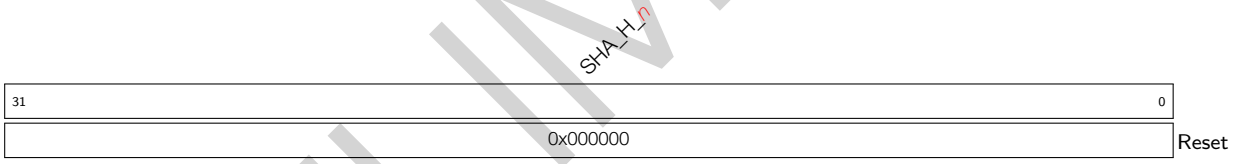
SHA_MODE 选择 SHA 加速器的运算标准，详见表 9-2。(R/W)

Register 9.10. SHA_DMA_BLOCK_NUM_REG (0x000C)



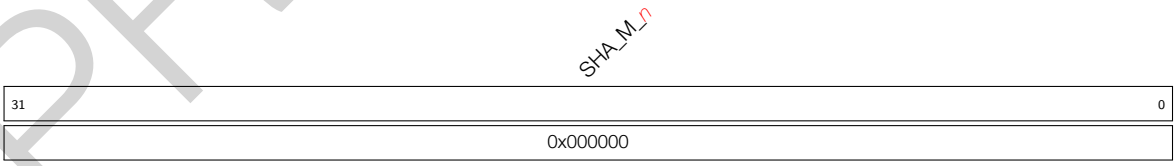
SHA_DMA_BLOCK_NUM 定义 DMA-SHA 工作模式下的信息块个数。（读写）

Register 9.11. 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 9.12. 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）

10 AES 加速器 (AES)

10.1 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。

10.2 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

10.3 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 10-1。

表 10-1. 工作模式

AES_DMA_ENABLE_REG	工作模式
0	Typical AES
1	DMA-AES

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

表 10-2. 密钥长度和加解密方向

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

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

注意：

ESP32-C3 的数字签名 (DS) [to be added later]模块也会调用 AES 加速器。此时，用户无法正常访问 AES 加速器。

10.4 Typical AES 工作模式

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

表 10-3. 状态返回值

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

10.4.1 密钥、明文、密文

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

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

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

- 如果为 AES-128/265 加密运算，则运算开始之前用明文初始化寄存器 AES_TEXT_IN_*m*_REG。运算完成

之后，AES 加速器将把密文更新入寄存器 AES_TEXT_OUT_*m*_REG。

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

10.4.2 字节序

文本字节序

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

表 10-4. Typical AES 文本字节序

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

[1] 有关 “State（以及 c 和 r）” 的详细定义，请参考 NIST FIPS 197 中 “3.4 The State” 章节。
[2] 其中，*x* = IN 或 OUT。

密钥字节序

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

表 10-5. AES-128 密钥字节序

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

[1] Bit 列代表 w[0] ~ w[3] 每个 word 中的各个字节。
[2] w[0] ~ w[3] 符合标准 NIST FIPS 197 中 “5.2 Key Expansion” 章节中对 “the first Nk words of the expanded key” 的描述。

表 10-6. AES-256 密钥字节序

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

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

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

10.4.3 Typical AES 工作模式的流程

单次运算

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

连续运算

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

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

10.5 DMA-AES 工作模式

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

表 10-7. 块模式选择

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

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

表 10-8. 状态返回值

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

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

10.5.1 密钥、明文、密文

块运算模式

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

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

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

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

表 10-9. TEXT-PADDING

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

10.5.2 字节序

在 DMA-AES 工作模式下，源数据和结果数据的传输完全由 DMA 完成，因此不支持字节序的控制调节，但要求它们在 memory 中以一定的方式来存放，且要求数据量必须是 block 的整数倍。

举例说明，假设 DMA 需要搬运 2 个 block 大小的源数据：

- 十六进制：0102030405060708090A0B0C0D0E0F10111213141516171819 1A1B1C1D1E1F20

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

表 10-10. DMA AES 存储字节序

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

10.5.3 标准增量函数

AES 加速器在进行 CTR 块运算时，还可提供两种标准增量函数供用户选择：INC_{32} 和 INC_{128} 。用户可通过将寄存器 AES_INC_SEL_REG 置为 0 或 1 选择 INC_{32} 或 INC_{128} 标准增量函数。更多有关标准增量函数的内容，请见 NIST SP 800-38A 标准中的 “B.1 The Standard Incrementing Function” 章节。

10.5.4 块个数

寄存器 AES_BLOCK_NUM_REG 存放明文或密文的块个数 (Block Number)，其值等于 length(**TEXT-PADDING**(P))/128，也等于 length(**TEXT-PADDING**(C))/128。这里的 P 指明文（plaintext），C 指密文（ciphertext）。该寄存器

仅在 DMA-AES 工作模式下有意义。

10.5.5 初始向量

存储器 AES_IV_MEM 的空间大小为 16 字节，仅在块运算模式下有效。对于 CBC/OFB/CFB8/CFB128 等操作，AES_IV_MEM 用于存放初始向量 (Initialization Vector, IV) 的值。对于 CTR 操作，AES_IV_MEM 存放初始计数器 (Initial Counter Block, ICB) 的值。

IV 和 ICB 都是 128-bit 长的比特串，从左向右被分割成 16 个字节 (Byte0, Byte1, Byte2, ⋯, Byte15)，构成一个字节序列，在 AES_IV_MEM 中存放时需要遵循表 10-10 中的字节序规则，即 Byte0 存放在 AES_IV_MEM 中的最低地址中，Byte15 存放在 AES_IV_MEM 中的最高地址中。

更多有关 IV 和 ICB 的信息，请参考 NIST SP 800-38A 标准。

10.5.6 DMA-AES 工作模式的流程

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

10.6 存储器列表

本小节的所有地址均为相对于 AES 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 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

10.7 寄存器列表

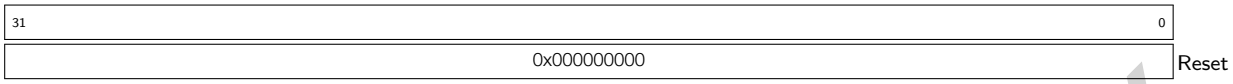
本小节的所有地址均为相对于 AES 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

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

10.8 寄存器

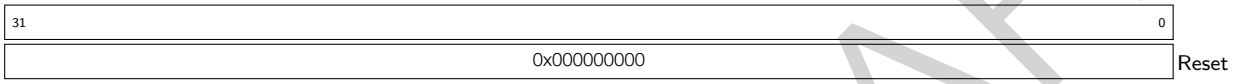
本小节的所有地址均为相对于 AES 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

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



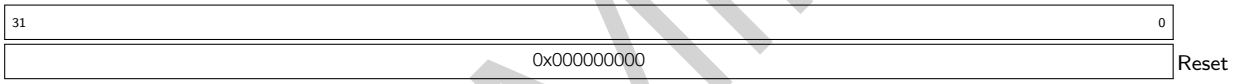
AES_KEY_*n*_REG (*n*: 0-7) AES 密钥资料寄存器。（读 / 写）

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



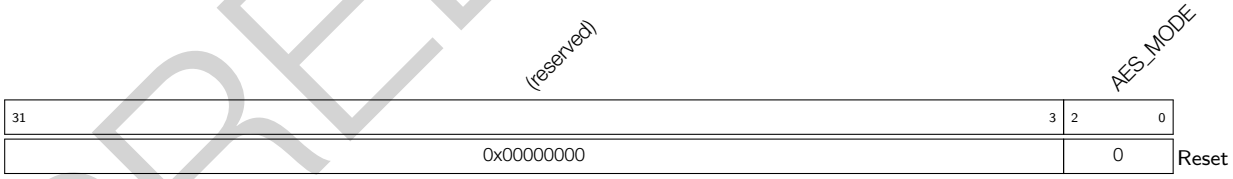
AES_TEXT_IN_*m*_REG (*m*: 0-3) Typical AES 文本输入寄存器。（读 / 写）

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



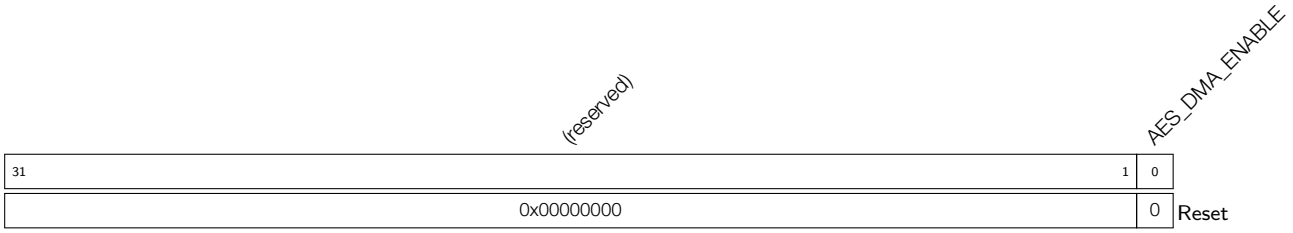
AES_TEXT_OUT_*m*_REG (*m*: 0-3) Typical AES 文本输出寄存器。（只读）

Register 10.4. AES_MODE_REG (0x0040)



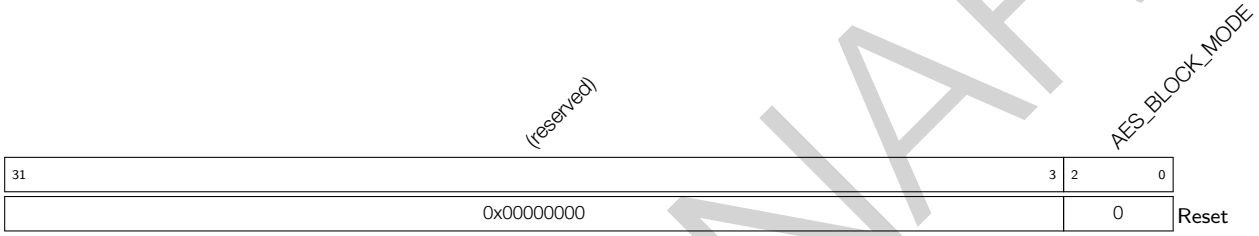
AES_MODE 选择 AES 加速器的密钥长度和加解密方向，详情请见表 10-2。（读 / 写）

Register 10.5. AES_DMA_ENABLE_REG (0x0090)



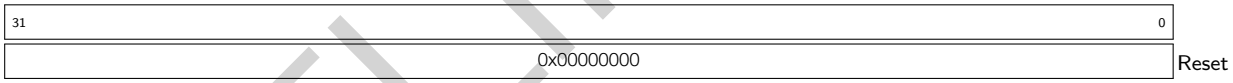
AES_DMA_ENABLE 选择 AES 加速器的工作模式。0：Typical AES，1：DMA-AES。详情请见表 10-1。（读 / 写）

Register 10.6. AES_BLOCK_MODE_REG (0x0094)



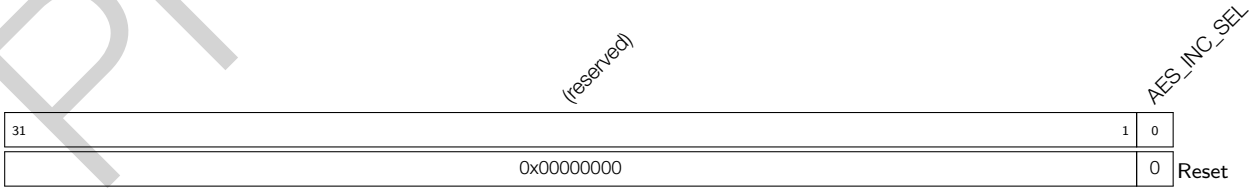
AES_BLOCK_MODE 选择 AES 加速器在 DMA-AES 工作模式下的块模式，详情请见表 10-7。（读 / 写）

Register 10.7. AES_BLOCK_NUM_REG (0x0098)



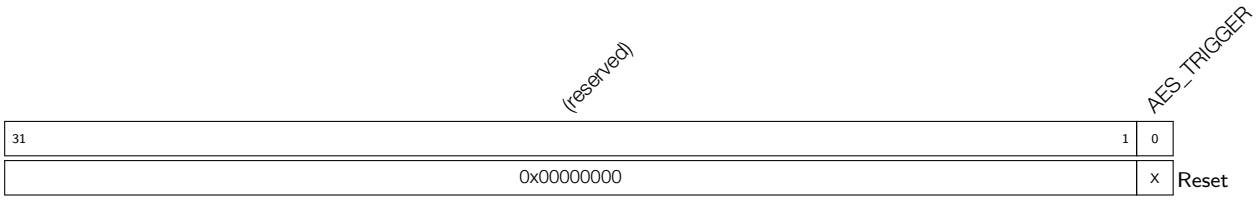
AES_BLOCK_NUM 在 DMA-AES 运算中待加解密的文本块数。详情请见章节 10.5.4。（读 / 写）

Register 10.8. AES_INC_SEL_REG (0x009C)



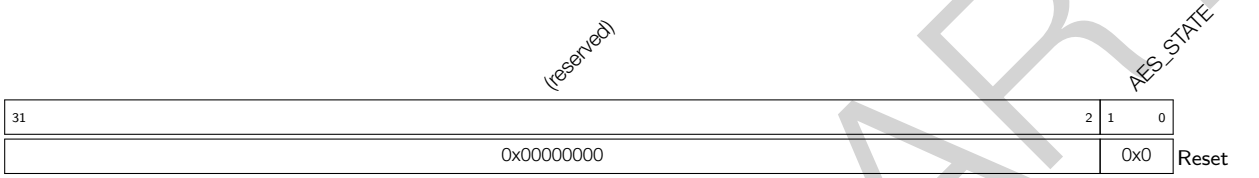
AES_INC_SEL 选择 CTR 块模式使用的标准增量函数。置 0 选择 INC_{32} 标准增量函数，置 1 选择 INC_{128} 标准增量函数。（读 / 写）

Register 10.9. AES_TRIGGER_REG (0x0048)



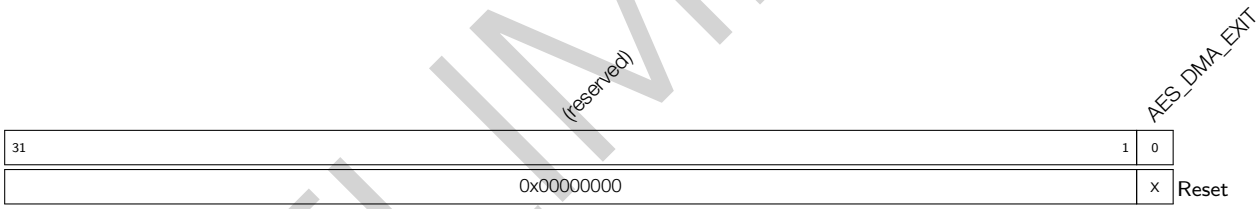
AES_TRIGGER 写入 1 使能 AES 运算。(只写)

Register 10.10. AES_STATE_REG (0x004C)



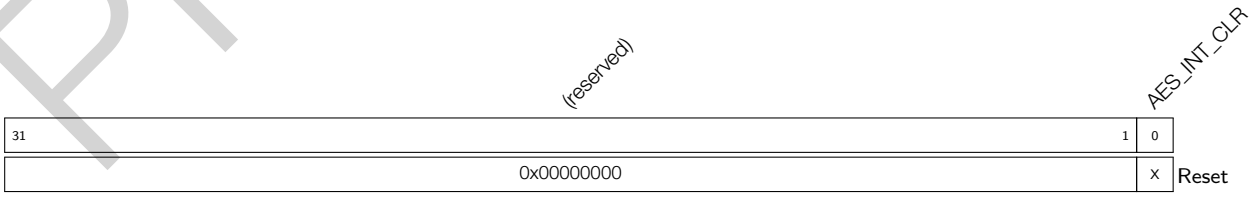
AES_STATE AES 状态寄存器。详见表 10-3 (Typical AES 工作模式) 和表 10-8 (DMA-AES 工作模式)。(只读)

Register 10.11. AES_DMA_EXIT_REG (0x00B8)



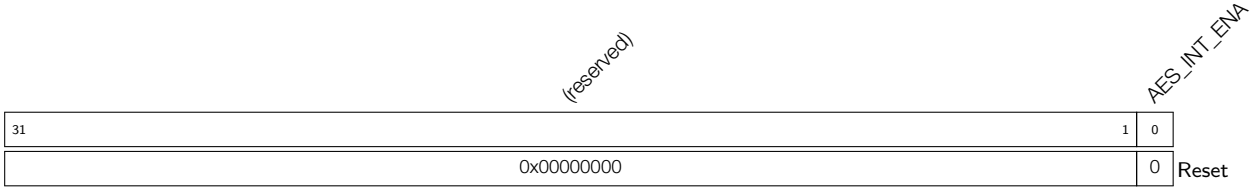
AES_DMA_EXIT 在 DMA-AES 运算完成后，在下一次配置 AES 任何寄存器之前，写入 1 使 AES 回到空闲状态。(只写)

Register 10.12. AES_INT_CLR_REG (0x00AC)



AES_INT_CLR 写入 1 清除 AES 中断。(只写)

Register 10.13. AES_INT_ENA_REG (0x00B0)



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

11 RSA 加速器 (RSA)

11.1 概述

RSA 加速器可为多种运用于“RSA 非对称式加密演算法”的高精度计算提供硬件支持，能够极大地降低此类运算的软件复杂度，且支持多种“运算子长度”，具有很高的运算效率。

11.2 主要特性

RSA 加速器支持以下功能：

- 大数模幂运算（支持两个加速选项）
- 大数模乘运算
- 大数乘法运算
- 多种运算子长度
- 中断功能

11.3 功能描述

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

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

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

> **注意：**
> ESP32-C3 的数字签名 (DS) [to be added later]模块也会调用 RSA 加速器。此时，用户无法正常访问 RSA 加速器。

11.3.1 大数模幂运算

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

$$b = 2^{32}$$

则运算子可以由若干个 b 进制数来表示：

$$n = \frac{N}{32}$$
$$Z = (Z_{n-1}Z_{n-2}\cdots Z_0)_b$$
$$X = (X_{n-1}X_{n-2}\cdots X_0)_b$$
$$Y = (Y_{n-1}Y_{n-2}\cdots Y_0)_b$$
$$M = (M_{n-1}M_{n-2}\cdots M_0)_b$$
$$\overline{r} = (\overline{r}_{n-1}\overline{r}_{n-2}\cdots \overline{r}_0)_b$$

其中 $Z_{n-1}\cdots Z_0$、$X_{n-1}\cdots X_0$、$Y_{n-1}\cdots Y_0$、$M_{n-1}\cdots M_0$、$\overline{r}_{n-1}\cdots \overline{r}_0$ 分别表示一个 b 进制数，位宽皆为 32。且 Z_{n-1}、X_{n-1}、Y_{n-1}、M_{n-1}、$\overline{r}_{n-1}$ 分别为 Z、X、Y、M、$\overline{r}$ 最高位的 b 进制数，而 Z_0、X_0、Y_0、M_0、$\overline{r}_0$ 分别为 Z、X、Y、M、$\overline{r}$ 最低位的 b 进制数。

另设 $R = b^n$，则计算得参数 $\overline{r} = R^2 \bmod M$。

M' 可使用下方公式计算：

$$M^{-1} \times M + 1 = R \times R^{-1}$$
$$M' = M^{-1} \bmod b$$

注意，上方公式适用于使用扩展二进制 GCD 算法的运算。

大数模幂运算的软件流程为：

1. 对寄存器 RSA_INTERRUPT_ENA_REG 写 1 / 0 以开启 / 关闭中断。
2. 配置相关寄存器。
   (a) 对寄存器 RSA_MODE_REG 写入 $(\frac{N}{32} - 1)$。
   (b) 对寄存器 RSA_M_PRIME_REG 写入 M'。
   (c) 根据需要配置加速选项相关寄存器。请参照章节 11.3.4 获取详细信息。
3. 将 X_i、Y_i、M_i、$\overline{r}_i (i \in \{0, 1, \ldots, n-1\})$ 分别写入存储器 RSA_X_MEM、RSA_Y_MEM、RSA_M_MEM、RSA_Z_MEM。每块存储器的容量都是 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。

11.3.2 大数模乘运算

大数模乘运算 $Z = X \times Y \bmod M$ 也是基于 Montgomery Multiplication 实现的。因此，与大数模幂运算类似，也需要预先通过软件计算额外的两个运算子 $\overline{r}$ 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、$\overline{r}_i$ $(i \in \{0, 1, \ldots, n-1\})$ 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_MODMULT_START_REG 写入 1。
5. 等待运算结束。轮询寄存器 RSA_IDLE_REG 直到读到 1，或者等待 RSA 中断产生。
6. 从存储器 RSA_Z_MEM 读出运算结果 Z_i $(i \in \{0, 1, \ldots, n-1\})$。
7. 若中断功能已开启，对寄存器 RSA_CLEAR_INTERRUPT_REG 写入 1 以清除中断。

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

11.3.3 大数乘法运算

大数乘法运算实现了 $Z = X \times Y$。其中 Z 的长度是运算子 X、Y 长度的两倍。所以 RSA 加速器只支持运算子 X、Y 长度为 $N = 32 \times x$ $(x \in \{1, 2, 3, \ldots, 48\})$ 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。
4. 对寄存器 RSA_MULT_START_REG 写入 1。

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

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

11.3.4 控制加速

对于大数模幂运算，ESP32-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 11-1 展示了 4 种选项组合对应的时间开销。注意，这里 SEARCH 选项开启时设定 α 为 16。

表 11-1. 加速效果

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

可以看到：

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

11.4 存储器列表

请注意，这里的地址都是相对于 RSA 加速器基地址的地址偏移量（相对地址），详见章节 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

11.5 寄存器列表

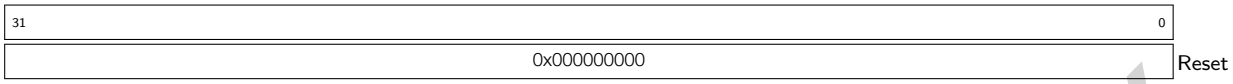
本小节的所有地址均为相对于 RSA 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

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

11.6 寄存器

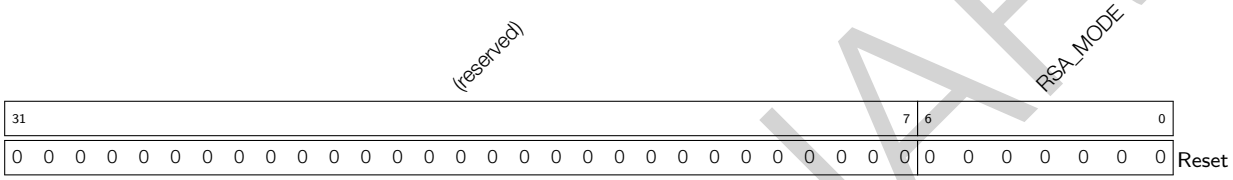
本小节的所有地址均为相对于 RSA 加速器基地址的地址偏移量（相对地址），具体基地址请见章节 3 *系统和存储器* 中的表 3-4。

Register 11.1. RSA_M_PRIME_REG (0x0800)



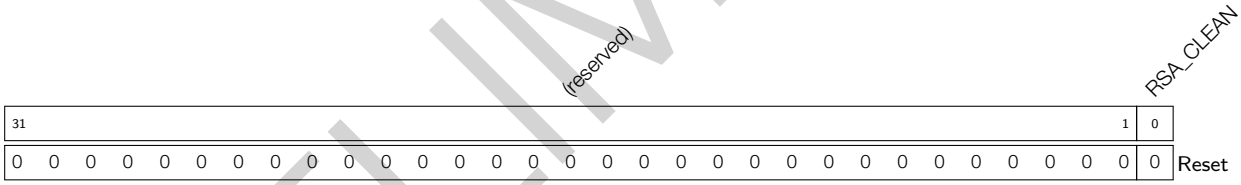
RSA_M_PRIME_REG 此寄存器存储 M'。（读 / 写）

Register 11.2. RSA_MODE_REG (0x0804)



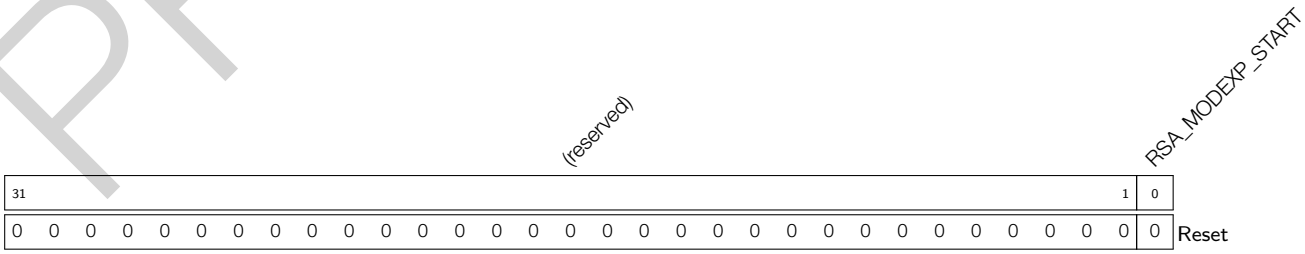
RSA_MODE 此寄存器存储模幂运算的模式。（读 / 写）

Register 11.3. RSA_CLEAN_REG (0x0808)



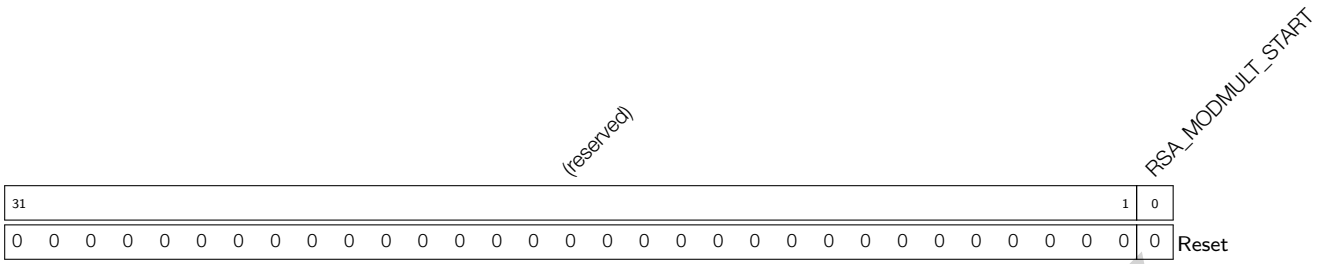
RSA_CLEAN 一旦存储器初始化结束，此位为 1。（只读）

Register 11.4. RSA_MODEXP_START_REG (0x080C)



RSA_MODEXP_START 写入 1 以开始模幂运算。（只写）

Register 11.5. RSA_MODMULT_START_REG (0x0810)



RSA_MODMULT_START 写入 1 以开始模乘运算。(只写)

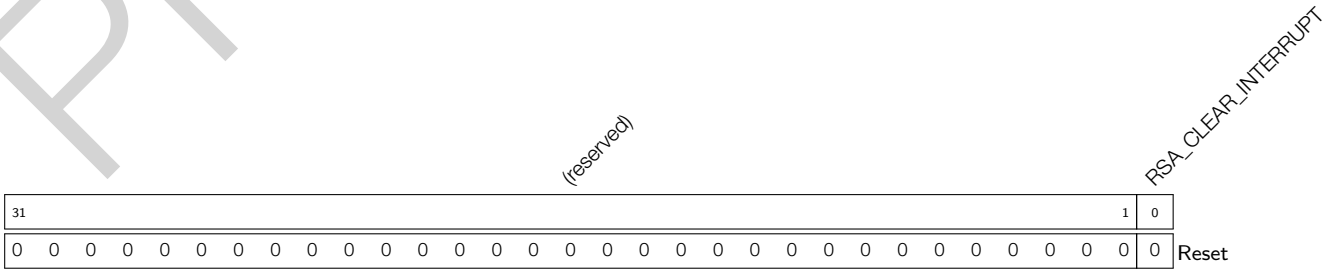
Register 11.6. RSA_MULT_START_REG (0x0814)

RSA_MULT_START 写入 1 以开始乘法运算。(只写)

Register 11.7. RSA_IDLE_REG (0x0818)

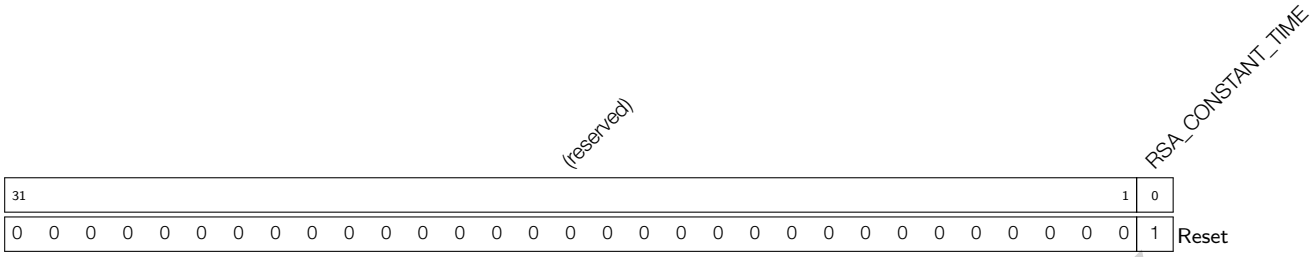
RSA_IDLE 当 RSA 空闲时，此位为 1。(只读)

Register 11.8. RSA_CLEAR_INTERRUPT_REG (0x081C)



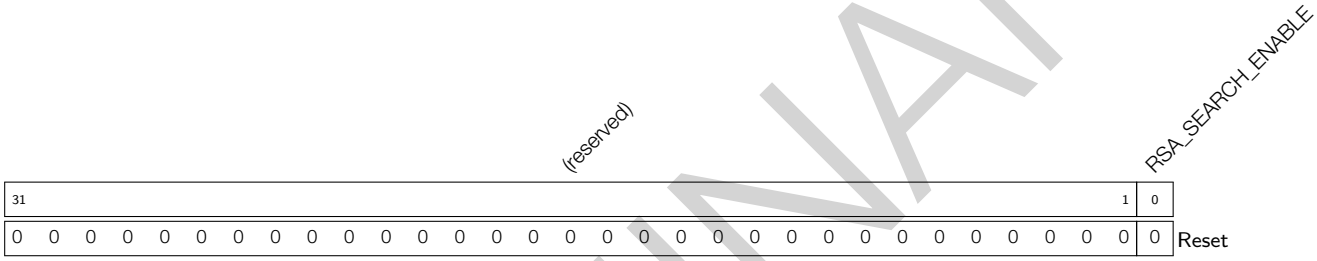
RSA_CLEAR_INTERRUPT RSA 中断清除寄存器。写入 1 清除中断。(只写)

Register 11.9. RSA_CONSTANT_TIME_REG (0x0820)



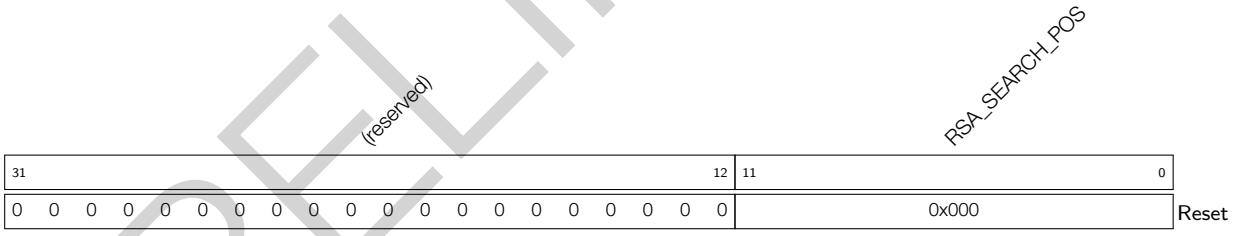
RSA_CONSTANT_TIME_REG 控制模幂运算中的 constant_time 选项。0：加速；1：不加速（默认）。（读 / 写）

Register 11.10. RSA_SEARCH_ENABLE_REG (0x0824)



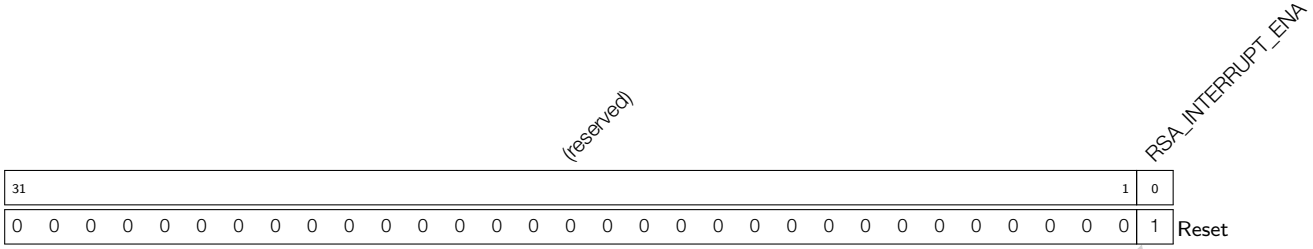
RSA_SEARCH_ENABLE 控制模幂运算中的 search 选项。1：加速；0：不加速（默认）。（读 / 写）

Register 11.11. RSA_SEARCH_POS_REG (0x0828)



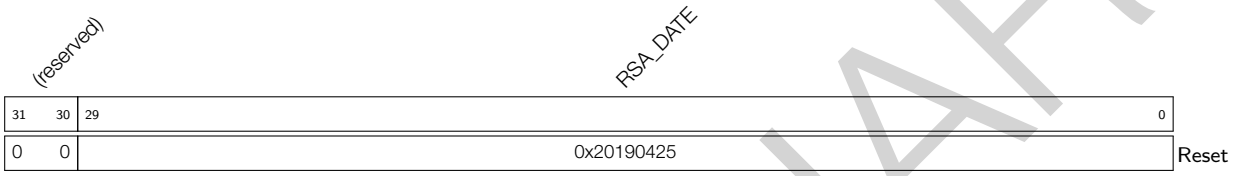
RSA_SEARCH_POS 模幂运算中的 search 加速选项。用于配置 search 的起始位置（读 / 写）。

Register 11.12. RSA_INTERRUPT_ENA_REG (0x082C)



RSA_INTERRUPT_ENA RSA 中断使能寄存器。写入 1 开启中断，默认开启。（读 / 写）

Register 11.13. RSA_DATE_REG (0x0830)



RSA_DATE 版本控制寄存器（读 / 写）。

12 随机数发生器 (RNG)

12.1 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。

12.2 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。

12.3 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。

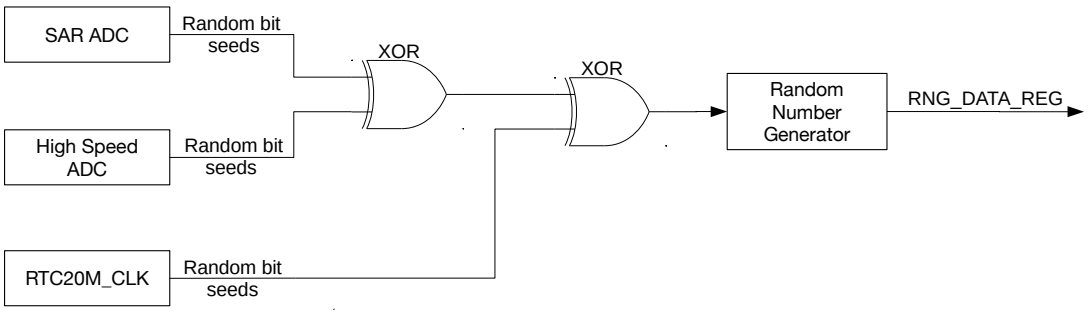


图 12-1. 噪声源

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 6 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。

12.4 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，

- SAR ADC 受控于 DIG ADC 控制器。详见 11 片上传感器与模拟信号处理 *[to be added later]* 章节。
- 高速 ADC 在 Wi-Fi 或蓝牙开启时自动打开。
- RTC20M_CLK 可通过设置 RTC_CNTL_CLK_CONF_REG 寄存器中的 RTC_CNTL_DIG_CLK20M_EN 位使能。

说明：

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 12.3 小节的介绍。

12.5 寄存器列表

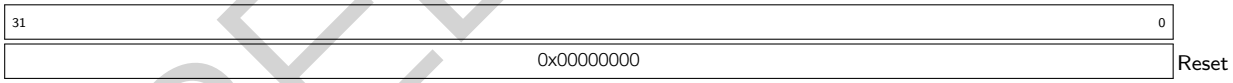
请注意，下表中的地址都是相对于随机数发生器基地址的地址偏移量（相对地址），详见章节 3 系统和存储器 中的表 3-4。

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

12.6 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 12.1. RNG_DATA_REG (0x00B0)



RNG_DATA 随机数来源。（只读）

13 UART 控制器 (UART)

13.1 概述

嵌入式应用通常要求一个简单的并且占用系统资源少的方法来传输数据。通用异步收发传输器 (UART) 即可以满足这些要求，它能够灵活地与外部设备进行全双工数据交换。芯片中有两个 UART 控制器可供使用，并且兼容不同的 UART 设备。另外，UART 还可以用作红外数据交换 (IrDA) 或 RS485 调制解调器。

两个 UART 控制器分别有一组功能相同的寄存器。本文以 UART*n* 指代两个 UART 控制器，*n* 为 0、1。

UART 是一种以字符为导向的通用数据链，可以实现设备间的通信。异步传输的意思是不需要在发送数据上添加时钟信息。这也要求发送端和接收端的速率、停止位、奇偶校验位等都要相同，通信才能成功。

一个典型的 UART 帧开始于一个起始位，紧接着是有效数据，然后是奇偶校验位（可有可无），最后是停止位。芯片上的 UART 控制器支持多种字符长度和停止位。另外，控制器还支持软硬件流控和 GDMA，可以实现无缝高速的数据传输。开发者可以使用多个 UART 端口，同时又能保证很少的软件开销。

13.2 主要特性

UART 控制器具有如下特性：

- 支持三个可预分频的时钟源
- 可编程收发波特率
- 两个 UART 的发送 FIFO 以及接收 FIFO 共享 512 x 8-bit RAM
- 全双工异步通信
- 支持输入信号波特率自检功能
- 支持 5/6/7/8 位数据长度
- 支持 1/1.5/2/3 个停止位
- 支持奇偶校验位
- 支持 AT_CMD 特殊字符检测
- 支持 RS485 协议
- 支持 IrDA 协议
- 支持 GDMA 高速数据通信
- 支持 UART 唤醒模式
- 支持软件流控和硬件流控

13.3 UART 架构

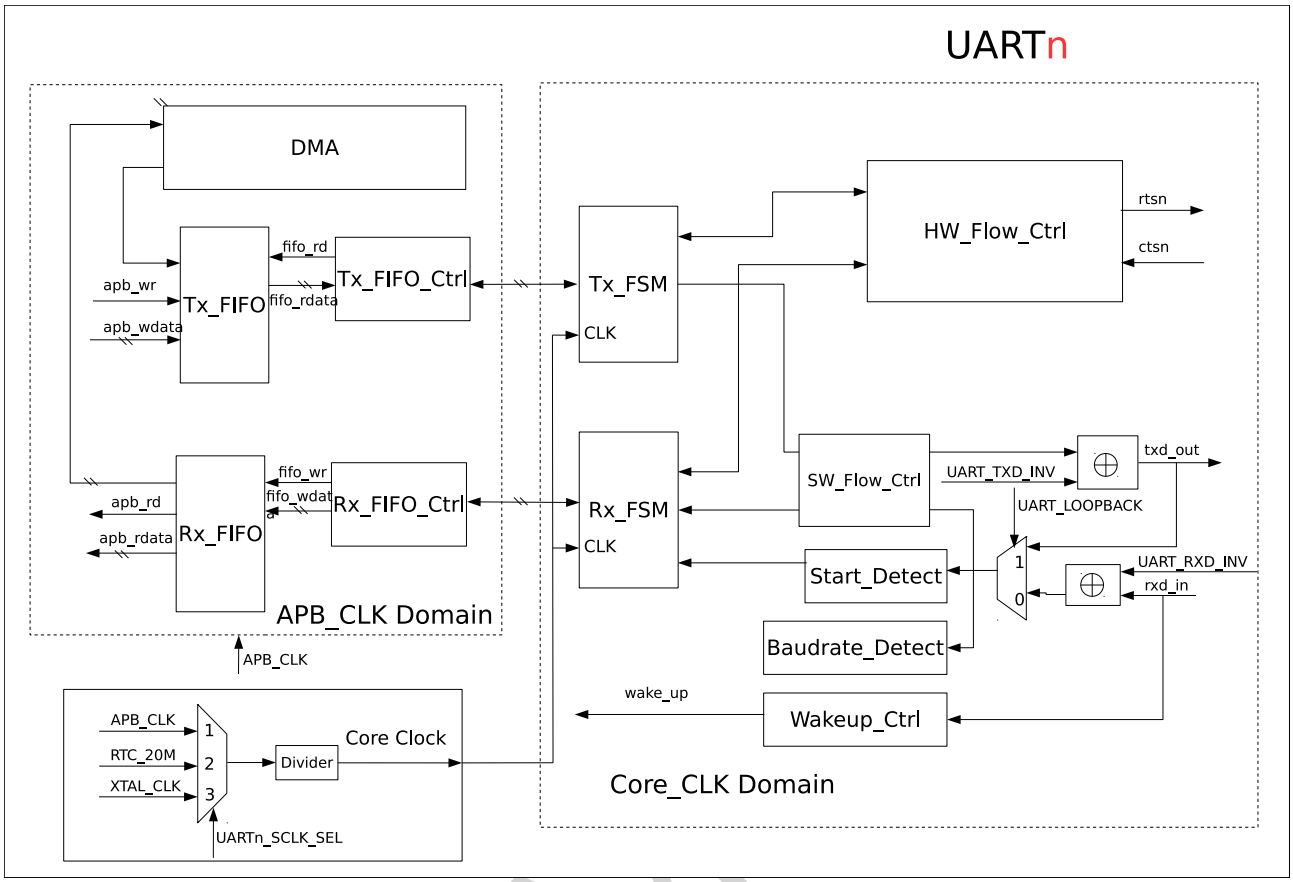


图 13-1. UART 基本架构图

图 13-1 为 UART 基本架构图。UART 模块工作在两个时钟域：APB_CLK 时钟域和 Core 时钟域。UART Core 有三个时钟源：80-MHz APB_CLK、RTC20M_CLK 以及晶振时钟 XTAL_CLK（详情请参考章节 6 复位和时钟）。可以通过配置 UART_SCLK_SEL 来选择时钟源。分频器用于对时钟源进行分频，然后产生时钟信号来驱动 UART Core 模块。UART_CLKDIV_REG 将分频系数分成两个部分：UART_CLKDIV 用于配置整数部分，UART_CLKDIV_FRAG 用于配置小数部分。

UART 控制器可以分为两个功能块：发送块和接收块。

发送块包含一个发送 FIFO 用于缓存待发送的数据。软件可以通过 APB 总线向 Tx_FIFO 写数据，也可以通过 GDMA 将数据搬入 Tx_FIFO。Tx_FIFO_Ctrl 用于控制 Tx_FIFO 的读写过程，当 Tx_FIFO 非空时，Tx_FSM 通过 Tx_FIFO_Ctrl 读取数据，并将数据按照配置的帧格式转化成比特流。比特流输出信号 txd_out 可以通过配置 UART_TXD_INV 寄存器实现取反功能。

接收块包含一个接收 FIFO 用于缓存待处理的数据。输入比特流 rxd_in 可以输入到 UART 控制器。可以通过 UART_RXD_INV 寄存器实现取反。Baudrate_Detect 通过检测最小比特流输入信号的脉宽来测量输入信号的波特率。Start_Detect 用于检测数据的 START 位，当检测到 START 位之后，Rx_FSM 通过 Rx_FIFO_Ctrl 将帧解析后的数据存入 Rx_FIFO 中。软件可以通过 APB 总线读取 Rx_FIFO 中的数据也可以使用 GDMA 进行数据接收。

HW_Flow_Ctrl 通过标准 UART RTS 和 CTS（rtsn_out 和 ctsn_in）流控信号来控制 rxd_in 和 txd_out 的数据流。SW_Flow_Ctrl 通过在发送数据流中插入特殊字符以及在接收数据流中检测特殊字符来进行数据流的控制。当 UART 处于 Light-sleep（详情请参考章节 12 低功耗管理 *(RTC_CNTL) [to be added later]*）状态时，Wakeup_Ctrl

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

13.4 功能描述

13.4.1 时钟与复位

UART 为异步外设。其寄存器配置模块与 TX/RX FIFO 工作在 APB_CLK 时钟域，而控制 UART 发送与接收的 Core 模块工作在 UART Core 时钟域。UART Core 有三个时钟源：APB_CLK，RTC20M_CLK 以及晶振时钟 XTAL_CLK，可通过配置 UART_SCLK_SEL 字段来选择时钟源。选择后的时钟源通过预分频器分频后进入 UART Core 模块。该预分频器支持小数分频，UART_SCLK_DIV_NUM 字段为整数部分，UART_SCLK_DIV_B 字段为小数部分的分子，UART_SCLK_DIV_A 为小数部分的分母。支持的分频范围为：1 ~ 256。

在 UART 波特率满足需求的情况下，通过预分频使 UART Core 模块工作在较小的时钟频率，从而减小 UART 外设的功耗。通常情况下，UART Core 模块时钟小于 APB_CLK 时钟，并且在满足 UART 波特率的情况下，UART Core 时钟分频系数可以配置到最大值。UART 也支持 UART Core 模块时钟大于 APB_CLK 时钟，此时，UART Core 模块时钟最大为 APB_CLK 的 3 倍。另外，UART TX/RX 的 Core 时钟可以被单独控制。置位 UART_TX_SCLK_EN 使能 UART TX 的 Core 时钟；置位 UART_RX_SCLK_EN 使能 UART RX 的 Core 时钟。

为确保配置寄存器的值成功从 APB_CLK 时钟域同步到 UART Core 时钟域，寄存器配置需要遵循一定的流程，详情情参考本节13.5。

对整个 UART 的复位，需要遵循如下配置流程：

- 将 SYSTEM_UART_MEM_CLK_EN 置 1 打开 UART RAM 时钟；
- 将 SYSTEM_UART*n*_CLK_EN 置 1 打开 UART*n* APB_CLK；
- 将 SYSTEM_UART*n*_RST 位清 0；
- 向寄存器 UART_RST_CORE 写 1；
- 向寄存器 SYSTEM_UART*n*_RST 写 1；
- 将寄存器 SYSTEM_UART*n*_RST 清 0；
- 向寄存器 UART_RST_CORE 清 0。

注：不推荐单独复位 UART APB 模块或者 UART Core 模块。

13.4.2 UART RAM

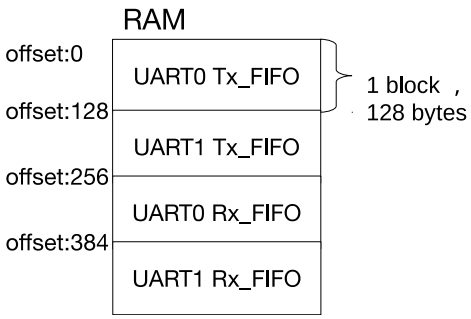


图 13-2. UART 共享 RAM 图

芯片中两个 UART 控制器共用 512x8-bit RAM 空间。如图 13-2 所示，RAM 以 block 为单位进行分配，1 block 为 128x8 bits。图 13-2 所示为默认情况下两个 UART 控制器的 Tx_FIFO 和 Rx_FIFO 占用 RAM 的情况。通过配置 UART_TX_SIZE 可以对 UART*n* 的 Tx_FIFO 进行扩展，通过配置 UART_RX_SIZE 可以对 UART*n* 的 Rx_FIFO 进行扩展，UART0 Tx_FIFO 可以从地址 0 扩展到整个 RAM 空间，UART1 Tx_FIFO 可以从地址 128 扩展到 RAM 的尾地址，UART0 Rx_FIFO 可以从地址 256 扩展到 RAM 的尾地址，UART1 Rx_FIFO 则不支持地址空间扩展，以 1 block 为单位进行扩展。需要注意的是当扩展某一个 UART 的 FIFO 空间时可能会占用其他 UART 的 FIFO 空间。比如，设置 UART0 的 UART_TX_SIZE 为 2, 则 UART0 Tx_FIFO 的地址从 0 扩展到 255。这时，UART1 Tx_FIFO 的默认空间被占用，这时将不能使用 UART1 发送器功能。

当两个 UART 控制器都不工作时，可以通过置位 UART_MEM_FORCE_PD 来使 RAM 进入低功耗状态。

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

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

配置 UART_TXFIFO_EMPTY_THRHD 可以设置 Tx_FIFO 空信号阈值，当存储在 Tx_FIFO 中的数据量小于 UART_TXFIFO_EMPTY_THRHD 时会产生中断 UART_TXFIFO_EMPTY_INT；配置 UART_RXFIFO_FULL_THRHD 可以设置 Rx_FIFO 满信号阈值，当储存在 Rx_FIFO 中的数据量大于 UART_RXFIFO_FULL_THRHD 会产生中断 UART_RXFIFO_FULL_INT。另外，当 Rx_FIFO 中储存的数据量超过其能存储的最大值时，会产生 UART_RXFIFO_OVF_INT 中断。

UART*n* 可以通过寄存器 UART_FIFO_REG 访问 FIFO。您可以写 UART_RXFIFO_RD_BYTE 将数据存入 TX FIFO，也可以读 UART_RXFIFO_RD_BYTE 获取 RX FIFO 中的数据。

13.4.3 波特率产生与检测

13.4.3.1 波特率产生

在 UART 发送或接收数据之前，需要配置寄存器来设置波特率。波特率发生器主要通过对输入时钟源的分频来实现，支持小数分频。UART_CLKDIV_REG 将分频系数分成两个部分：UART_CLKDIV 用于配置整数部分，UART_CLKDIV_FRAG 用于配置小数部分。在输入时钟为 80 MHz 的情况下，UART 能支持的最大波特率为 5 MBaud。

波特率分频器系数由 UART_CLKDIV + (UART_CLKDIV_FRAG/16) 构成。也就是说，最终波特率为 INPUT_FREQ/(UART_CLKDIV + (UART_CLKDIV_FRAG/16))。例如，若 UART_CLKDIV = 694，UART_CLKDIV_FRAG = 7，则分频系数为 (694 + 7/16) = 694.4375。需要注意的是 INPUT_FREQ 为 UART Core 时钟。

UART_CLKDIV_FRAG 为 0 时，分频器为整数分频，每 UART_CLKDIV 个输入脉冲都会产生一个输出脉冲。

UART_CLKDIV_FRAG 不为 0 时，分频器为小数分频，输出波特率脉冲不完全统一。如图 13-3 所示，每 16 个输出脉冲，波特率发生器分频 (UART_CLKDIV + 1) 个输入脉冲或 UART_CLKDIV 个输入脉冲。分频 (UART_CLKDIV + 1) 个输入脉冲产生 UART_CLKDIV_FRAG 个输出脉冲，分频 UART_CLKDIV 个输入脉冲产生剩余的 (16 - UART_CLKDIV_FRAG) 个输出脉冲。

如图 13-3 所示，输出脉冲相互交错，使得输出时序更加统一。

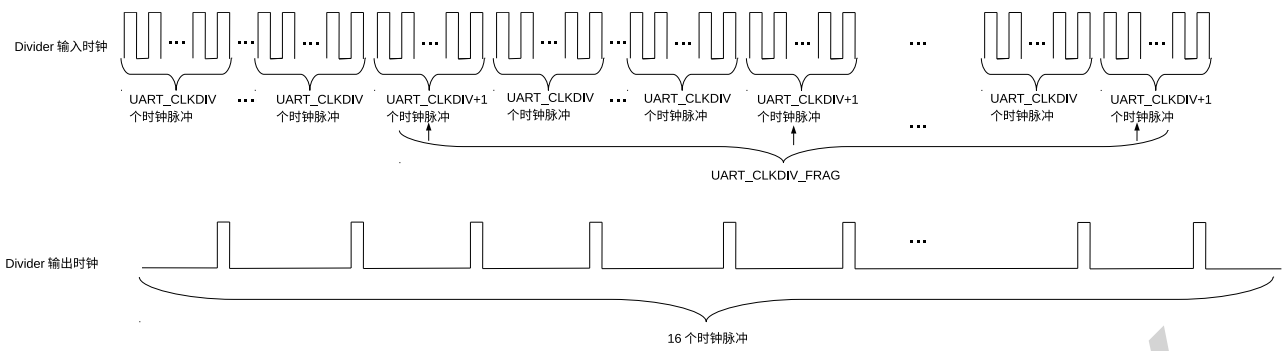


图 13-3. UART 控制器分频

为了支持 IrDA （详情见本节 13.4.6 *IrDA*），IrDA 小数分频器会产生 16 × UART_CLKDIV_REG 分频的时钟用于 IrDA 数据传输。产生 IrDA 数据传输时钟的小数分频器原理与上述小数分频器一样，取UART_CLKDIV/16 作为分频值的整数部分，取 UART_CLKDIV 的低 4 比特作为小数部分。

13.4.3.2 波特率检测

置位 UART_AUTOBAUD_EN 可以开启 UART 波特率自检测功能。图 13-1 中的 Baudrate_Detect 可以滤除信号脉宽小于 UART_GLITCH_FILT 的噪声。

在 UART 双方进行通信之前可以通过发送几个随机数据让具有波特率检测功能的数据接收方进行波特率分析。UART_LOWPULSE_MIN_CNT 存储了最小低电平脉冲宽度，UART_HIGHPULSE_MIN_CNT 存储了最小高电平脉冲宽度，UART_POSEDGE_MIN_CNT 存储了两个上升沿之间的最小脉冲宽度，UART_NEGEDGE_MIN_CNT 存储了两个下降沿之间最小的脉冲宽度。软件可以通过读取这四个寄存器获取发送方的波特率。

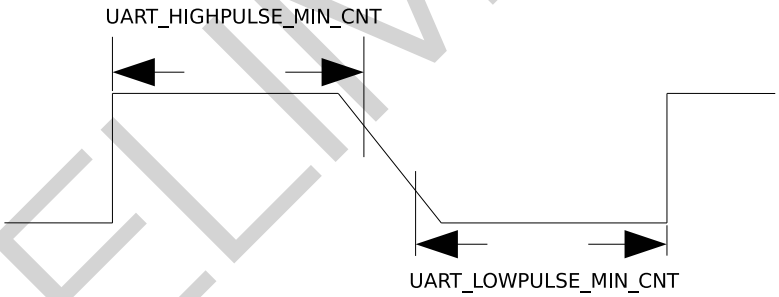


图 13-4. UART 信号下降沿较差时序图

波特率的计算分为三种情况：

1. 正常情况下，为防止因亚稳态在上升沿或下降沿附近采样数据错误而导致 UART_LOWPULSE_MIN_CNT 或者 UART_HIGHPULSE_MIN_CNT 不准确，单比特脉冲宽度可以通过将这两个值相加取平均消除误差。计算公式如下：

$$B_{\text{uart}} = \frac{f_{\text{clk}}}{(\text{UART\_LOWPULSE\_MIN\_CNT} + \text{UART\_HIGHPULSE\_MIN\_CNT} + 2)/2}$$

2. 对于 UART 信号的下降沿信号比较差的情况，如图13-4所示，这时通过取 UART_LOWPULSE_MIN_CNT 与 UART_HIGHPULSE_MIN_CNT 的和平均得到的值不准确，可以通过 UART_POSEDGE_MIN_CNT 获取发送方波特率。计算公式如下：

$$B_{\text{uart}} = \frac{f_{\text{clk}}}{(\text{UART\_POSEDGE\_MIN\_CNT} + 1)/2}$$

3. 对于 UART 信号的上升沿信号比较差的情况，可以通过 UART_NEGEDGE_MIN_CNT 获取发送方波特率。计算公式如下：

$$B_{uart} = \frac{f_{clk}}{(UART\_NEGEDGE\_MIN\_CNT + 1)/2}$$

13.4.4 UART 数据帧

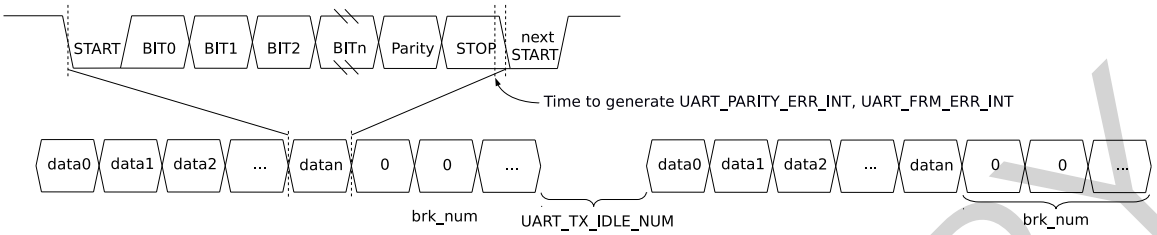


图 13-5. UART 数据帧结构

图 13-5 所示为基本数据帧格式，数据帧从 START 位开始以 STOP 位结束。START 占用 1 bit，STOP 位可以通过配置 UART_STOP_BIT_NUM、UART_DL1_EN 和 UART_DL0_EN 实现 1、1.5、2、3 位宽。START 为低电平，STOP 为高电平。

数据位宽 (BIT0 ~ BITn) 为 5 ~ 8 bit，可以通过 UART_BIT_NUM 进行配置。当置位 UART_PARITY_EN 时，数据帧会在数据之后添加一位奇偶校验位。UART_PARITY 用于选择奇校验或是偶校验。当接收器检测到输入数据的校验位错误时会产生 UART_PARITY_ERR_INT 中断，输入数据仍会存入 Rx_FIFO。当接收器检测到数据数据帧格式错误时会产生 UART_FRM_ERR_INT 中断，默认情况下，输入数据会被存入 Rx_FIFO。

Tx_FIFO 中数据都发送完成后会产生 UART_TX_DONE_INT 中断。置位 UART_TXD_BRK 时，Tx_FIFO 中数据发送完成后发送端会继续发送几个连续的特殊数据帧 NULL，在 NULL 数据帧，TX 数据线输出为低电平。NULL 数据帧的数量可由 UART_TX_BRK_NUM 进行配置。发送器发送完所有的 NULL 数据帧之后会产生 UART_TX_BRK_DONE_INT 中断。数据帧之间可以通过配置 UART_TX_IDLE_NUM 保持最小间隔时间。当一帧数据之后的空闲时间大于等于 UART_TX_IDLE_NUM 寄存器的配置值时则产生 UART_TX_BRK_IDLE_DONE_INT 中断。

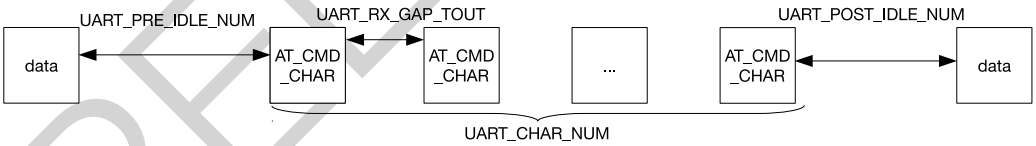


图 13-6. AT_CMD 字符格式

图 13-6 为一种特殊的 AT_CMD 字符格式。当接收器连续收到 AT_CMD_CHAR 字符且字符之间满足如下条件时将会产生 UART_AT_CMD_CHAR_DET_INT 中断。

- 接收到的第一个 AT_CMD_CHAR 与上一个非 AT_CMD_CHAR 之间间隔至少 UART_PRE_IDLE_NUM 个波特率周期。
- AT_CMD_CHAR 字符之间间隔小于 UART_RX_GAP_TOUT 个波特率周期。
- 接收的 AT_CMD_CHAR 字符个数必须大于等于 UART_CHAR_NUM。
- 接收到的最后一个 AT_CMD_CHAR 字符与下一个非 AT_CMD_CHAR 之间间隔至少 UART_POST_IDLE_NUM 个波特率周期。

13.4.5 RS485

UART 支持 RS485 协议，RS485 因使用差分信号传输数据，相比于 RS232 具有更远的传输距离及更高的传输速率。RS485 有两线半双工及四线全双工模式，UART 模块采用两线半双工模式，并支持侦听总线的功能。RS485 两线 multidrop 模式，最大可支持 32 个 slave。

13.4.5.1 驱动控制

如图 13-7 所示，RS485 两线 multidrop 系统中，需要一个外部 RS485 传输器实现单端信号与差分信号的转换。RS485 传输器包括一个驱动器与一个接收器。当 UART 不作为发送器时，通过关闭驱动器来断开与差分传输线的连接。DE 为 1 时，使能驱动器；DE 为 0 关闭驱动器。

UART 接收端通过外部接收器将差分信号转为单端信号。RE 作为接收器的使能控制信号，RE 为 0，使能接收器；RE 为 1，关闭接收器。如果 RE 被配置为 0，从而允许 UART 保持侦听总线上的数据，包括 UART 发送的数据。

DE 信号的控制分为软件控制和硬件控制两种方法。为减少软件的开销，DE 信号采用硬件来控制。图 13-7 所示，DE 与 UART 的 dtrn_out 相连（详见 13.4.8.1 小节）。

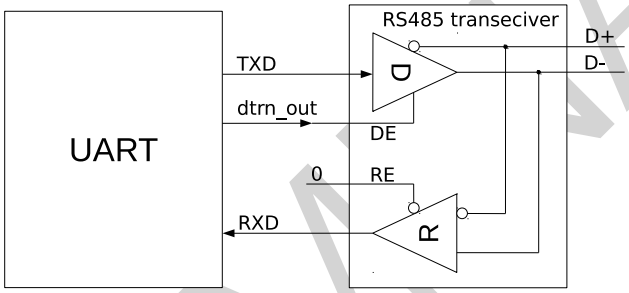


图 13-7. RS485 模式驱动控制结构图

13.4.5.2 转换延时

默认情况下，UART 处于接收状态。当从发送转为接收状态时，为保证发送数据被稳定接收，RS485 协议推荐在发送停止位之后增加一个波特率的转换延时。UART 发送模块支持在 Start 位之前或在停止位之后增加一个波特率的延时。置位UART_DL0_EN，在 Start 位之前增加一个波特率周期延时；置位 UART_DL1_EN，在停止位之后增加一个波特率周期延时。

13.4.5.3 总线侦听

RS485 两线 multidrop 系统中，当外部 RS485 传输器的 RE 被配置为 0 时，UART 支持侦听总线。默认情况下，不允许 UART 在发送数据时接收数据。置位UART_RS485TX_RX_EN，允许在发送数据时接收数据，配合外部 RS485 传输器的配置，UART 保持侦听传输总线。另外，默认情况下，不允许 UART 在接收数据时发送数据。置位 UART_RS485RXBY_TX_EN，允许在接收数据时发送数据。

UART 支持侦听 UART 发送的数据。UART 处于发送状态下，当侦听到 UART 发送的数据与 UART 接收的数据不同时，触发 UART_RS485_CLASH_INT 中断；侦听到发送的数据帧错误时，触发 UART_RS485_FRM_ERR_INT 中断；侦听到发送数据极性错误时，触发 UART_RS485_PARITY_ERR_INT 中断。

13.4.6 IrDA

IrDA 数据协议由物理层，链路接入层和链路管理层三个基本层协议组成。UART 实现了其物理层协议。在 IrDA 编码模式下，支持最大信号速率到 115.2 Kbit/s，即 SIR 模式。如图 13-8 所示，IrDA 编码器将来自 UART 的非归零编码（NRZ）信号采用反向归零编码（RZI）并输出给外部驱动和红外 LED，用 3/16 Bit Time 的脉宽调制信号表示逻辑 “0”，用低电平表示逻辑 “1”。IrDA 解码器接收来自红外接收器的信号并输出为 UART 的 NRZ 编码。一般情况下，接收端信号空闲时为高电平，编码器输出极性与解码器输入极性相反。当检测到低脉冲表示接收到开始信号。

IrDA 使能时，一个比特被划分为 16 个时钟周期，在其第 9、10、11 个时钟周期中，当需要发送的比特为 0 时，IrDA 输出为高。

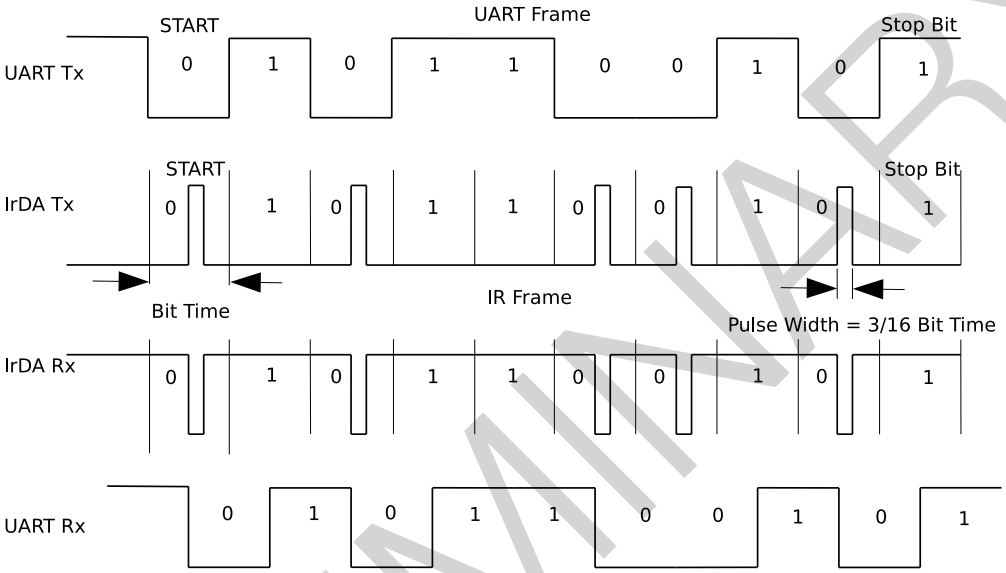


图 13-8. SIR 模式编解码时序图

IrDA 为半双工传输协议，不允许同时进行收发。如图13-9所示，置位 UART_IRDA_EN 使能 IrDA 功能。置位 UART_IRDA_TX_EN（拉高）使能 IrDA 发送数据，这时不允许 IrDA 接收数据；复位 UART_IRDA_TX_EN（拉低）使能 IrDA 接收数据，这时不允许 IrDA 发送数据。

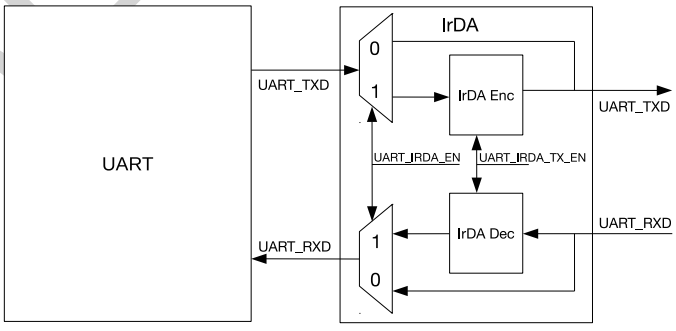


图 13-9. IrDA 编解码结构图

13.4.7 唤醒

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

芯片。

13.4.8 流控

UART 控制器有两种数据流控方式：硬件流控和软件流控。硬件流控主要通过输出信号 rtsn_out 以及输入信号 dsrn_in 进行数据流控制。软件流控主要通过在发送数据流中插入特殊字符以及在接收数据流中检测特殊字符来实现数据流控功能。

13.4.8.1 硬件流控

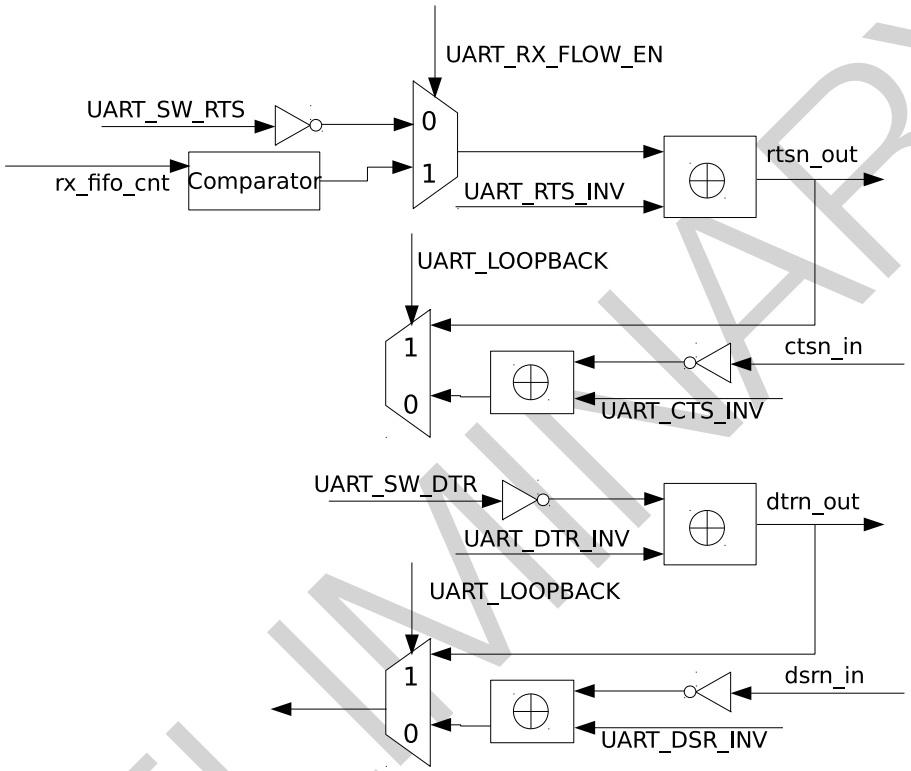


图 13-10. 硬件流控图

图 13-10 为 UART 硬件流控图。硬件流控的控制信号为输出信号 rtsn_out 及输入信号 ctsn_in。图 13-11 为两个 UART 之间硬件流控信号连接图。记 ESP32-C3 UART 为 IU0，External UART 为 EU0，下文将使用这两个标记来区分两个 UART。输出信号 rtsn_out （IU0）为低电平表示允许对方（EU0）发送数据，rtsn_out（IU0）为高电平表示通知对方（EU0）中止数据发送直到 rtsn_out （IU0）恢复低电平。rtsn_out 输出信号的控制有两种方式。

- 软件控制：将 UART_RX_FLOW_EN 置 0 进入该模式。该模式下通过软件配置 UART_SW_RTS 改变 rtsn_out 的电平。
- 硬件控制：将 UART_RX_FLOW_EN 置 1 进入该模式。该模式下硬件会当 Rx_FIFO 中的数据大于 UART_RX_FLOW_THRHD 时拉高 rtsn_out 的电平。

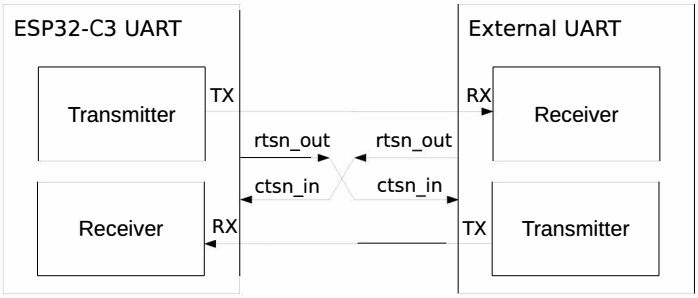


图 13-11. 硬件流控信号连接图

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

UART 发送设备 (IU0) 输出信号 dtrn_out 为高电平表示发送数据已经准备完毕，处于可用状态。dtrn_out 通过配置寄存器 UART_SW_DTR 产生。UART 接收设备 (IU0) 在检测到输入信号 dsrn_in 的沿变化时会产生 UART_DSR_CHG_INT 中断。软件在检测到中断后，通过读取 UART_DSRN 可以获取 dsrn_in 的输入信号电平，UART_DSRN 为高电平时，表示对方设备（EU0）处于可用状态。

对于 RS485 两线 multidrop 系统，使用 dtrn_out 来收发转换。置位 UART_RS485_EN 使能 RS485 功能，dtrn_out 由硬件产生。数据开始发送时，dtrn_out 拉高，使能外部驱动器；数据最后一位发送完成后，dtrn_out 拉低，关闭外部驱动器。注意，当使能停止位之后增加一个波特率延时时，dtrn_out 会在延时结束后才拉低。

置位 UART_LOOPBACK 即开启 UART 的回环测试功能。此时 UART 的输出信号 txd_out 和其输入信号 rxd_in 相连，rtsn_out 和 ctsn_in 相连，dtrn_out 和 dsrn_out 相连。当接收的数据与发送的数据相同时表明 UART 能够正常发送和接收数据。

13.4.8.2 软件流控

软件流控不使用硬件的 CTS/RTS 控制线，而是在发送数据流中嵌入 XON/XOFF 字符来通知对方是否可以使用数据发送来实现流控。将 UART_SW_FLOW_CON_EN 置 1 使能软件流控。

在使用软件流控后，硬件会自动检测接收数据流中是否有 XON/XOFF 字符，在检测到相应的字符后会产生 UART_SW_XOFF_IN 或 UART_SW_XON_INT 中断。在检测到接收数据流中有 XOFF 字符后，发送器将会在发送完当前数据后停止发送；在检测到接收数据流中有 XON 字符后，将会使能发送器发送数据。另外，软件可以通过置位 UART_FORCE_XOFF 来强制发送器停止发送数据，发送器会在发送完当前字节后停止发送；也可以通过置位 UART_FORCE_XON 来使能发送器发送数据。

软件可以根据 Rx_FIFO 中剩余空间大小决定流控字符的发送。置位 UART_SEND_XOFF，发送器会在发送完当前数据之后插入一个 XOFF 字符，该字符通过寄存器 UART_XOFF_CHAR 配置；置位 UART_SEND_XON，发送器会在发送完当前数据之后插入一个 XON 字符，该字符通过寄存器 UART_XON_CHAR 配置。另外，当 UART 接收 FIFO 中的数据量超过 UART_XOFF_THRESHOLD 时，硬件会置位 UART_SEND_XOFF，UART 发送器会在发送完当前数据之后插入一个 XOFF 字符，该字符通过寄存器 UART_XOFF_CHAR 配置。当 UART 接收 FIFO 中的数据量小于 UART_XON_THRESHOLD 时，硬件会置位 UART_SEND_XON，UART 发送器会在发送完当前数据之后插入一个 XON 字符，该字符通过寄存器 UART_XON_CHAR 配置。

13.4.9 GDMA 模式

ESP32-C3 中的两个 UART 接口通过通用主机控制器接口（UHCI）共用 1 组 GDMA TX/RX 通道。在 GDMA 模式下，支持对 HCI 协议数据包的解析（decoder）及数据包封装（encoder）。UHCI_UART*n*_CE 字段用于选择哪

个串口占用 GDMA 通道。

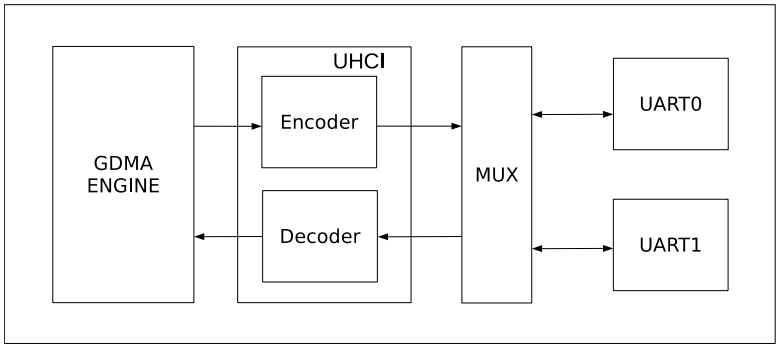


图 13-12. GDMA 模式数据传输

图 13-12 为 GDMA 方式数据传输图。在 GDMA Rx 通道接收数据前，软件将接收链表准备好。GDMA_INLINK_ADDR_CH*n* 用于指向第一个接收链表描述符。置位 GDMA_INLINK_START_CH*n* 之后，通用主机控制器接口 (UHCI) 会将 UART 接收到的数据传送给 Decoder。经过 Decoder 解析之后的数据在 GDMA 通道的控制下存入接收链表指定的 RAM 空间。

在 GDMA Tx 通道发送数据前，软件需要将发送链表和发送数据准备好，GDMA_OUTLINK_ADDR_CH*n* 用于指向第一个发送链表描述符。置位 GDMA_OUTLINK_START_CH*n* 之后，GDMA 引擎即从链表中指定的 RAM 地址读取数据，并通过 Encoder 进行数据包封装，然后经 UART 的发送模块串行发送出去。

HCI 的数据包格式为（分隔符 + 数据 + 分隔符）。Encoder 用于在数据前后加上分隔符，并将数据中和分隔符一样的数据用特殊字符替换。Decoder 用于去除数据包前后分隔符，并将数据中的特殊字符进行替换为分隔符。数据前后的分隔符可以有连续多个。分隔符可由 UHCI_SEPER_CHAR 进行配置，默认值为 0xC0。数据中与分隔符一样的数据可以用 UHCI_ESC_SEQ0_CHAR0（默认为 0xDB）和 UHCI_ESC_SEQ0_CHAR1（默认为 0xDD）进行替换。当数据全部发送完成后，会产生 GDMA_OUT_TOTAL_EOF_CH*n*_INT 中断。当数据接收完成后，会产生 GDMA_IN_SUC_EOF_CH*n*_INT 中断。

13.4.10 UART 中断

- UART_AT_CMD_CHAR_DET_INT：当接收器检测到 AT_CMD 字符时触发此中断。
- UART_RS485_CLASH_INT：在 RS485 模式下检测到发送器和接收器之间的冲突时触发此中断。
- UART_RS485_FRM_ERR_INT：在 RS485 模式下检测到发送块发送的数据帧错误时触发此中断。
- UART_RS485_PARITY_ERR_INT：在 RS485 模式下检测到发送块发送的数据校验位错误时触发此中断。
- UART_TX_DONE_INT：当发送器发送完 FIFO 中的所有数据时触发此中断。
- UART_TX_BRK_IDLE_DONE_INT：当发送器在最后一个数据发送后保持了最短的间隔时间时触发此中断。
- UART_TX_BRK_DONE_INT：当发送 FIFO 中的数据发送完之后发送器完成了发送 NULL 则触发此中断。
- UART_GLITCH_DET_INT：当接收器在起始位的中点处检测到 glitch 时触发此中断。
- UART_SW_XOFF_INT：UART_SW_FLOW_CON_EN 置位时，当接收器接收到 Xoff 字符时触发此中断。
- UART_SW_XON_INT：UART_SW_FLOW_CON_EN 置位时，当接收器接收到 Xon 字符时触发此中断。
- UART_RXFIFO_TOUT_INT：当接收器接收一个字节的时间大于 UART_RX_TOUT_THRHD 时触发此中断。
- UART_BRK_DET_INT：当接收器在停止位之后检测到 NULL 时触发此中断。

- UART_CTS_CHG_INT：当接收器检测到 CTSn 信号的沿变化时触发此中断。
- UART_DSR_CHG_INT：当接收器检测到 DSRn 信号的沿变化时触发此中断。
- UART_RXFIFO_OVF_INT：当接收器接收到的数据量多于 FIFO 的存储量时触发此中断。
- UART_FRM_ERR_INT：当接收器检测到数据帧错误时触发此中断。
- UART_PARITY_ERR_INT：当接收器检测到校验位错误时触发此中断。
- UART_TXFIFO_EMPTY_INT：当发送 FIFO 中的数据量少于 UART_TXFIFO_EMPTY_THRHD 所指定的值时触发此中断。
- UART_RXFIFO_FULL_INT：当接收器接收到的数据多于 UART_RXFIFO_FULL_THRHD 所指定的值时触发此中断。
- UART_WAKEUP_INT：UART 被唤醒时产生此中断。

13.4.11 UCHI 中断

- UHCI_APP_CTRL1_INT：软件置位 UHCI_APP_CTRL1_INT_RAW 时触发此中断。
- UHCI_APP_CTRL0_INT：软件置位 UHCI_APP_CTRL0_INT_RAW 时触发此中断。
- UHCI_OUTLINK_EOF_ERR_INT：当检测到发送链表描述符中的 EOF 有错误时触发此中断。
- UHCI_SEND_A_REG_Q_INT：当使用 always_send 发送一串短包，UHCI 发送了短包后触发此中断。
- UHCI_SEND_S_REG_Q_INT：当使用 single_send 发送一串短包，UHCI 发送了短包后触发此中断。
- UHCI_TX_HUNG_INT：当 UHCI 利用 GDMA Tx 通道从 RAM 中读取数据的时间过长时触发此中断。
- UHCI_RX_HUNG_INT：当 UHCI 利用 GDMA Rx 通道接收数据的时间过长时触发此中断。
- UHCI_TX_START_INT：当检测到分隔符时触发此中断。
- UHCI_RX_START_INT：当分隔符已发送时触发此中断。

13.5 编程流程

13.5.1 寄存器类型

UART 的所有寄存器都处于 APB_CLK 时钟域。对于软件可配置的寄存器，根据其作用的时钟域及同步处理，将其分为三类：立即寄存器，同步寄存器及静态寄存器。立即寄存器作用于 APB_CLK 时钟域，通过 APB 总线配置后立即生效；同步寄存器作用于 Core 时钟域，这些寄存器需要经过同步之后才能生效；静态寄存器也作用于 Core 时钟域，但这些寄存器不会在 UART 工作过程中动态修改。静态寄存器没有同步处理，软件可以通过开关 UART TX/RX Core 时钟的方式保证 UART Core 时钟域采样到正确的配置信息。

13.5.1.1 同步寄存器

为了确保作用于 UART Core 时钟域的寄存器被正确采样，他们中大多数都做了跨时钟域处理，这部分即为同步寄存器。同步寄存器如表13-1所示。对这些寄存器的配置流程如下：

- 将 UART_UPDATE_CTRL 清 0 使能寄存器同步功能；
- 等待 UART_REG_UPDATE 为 0，确保上一次同步已经完成；
- 配置同步寄存器；

- 向 UART_REG_UPDATE 写 1，将配置的值同步到 Core 时钟域。

表 13-1. UART*n*同步寄存器

寄存器	域名
UART_CLKDIV_REG	UART_CLKDIV_FRAG[3:0]
	UART_CLKDIV[11:0]
UART_CONF0_REG	UART_AUTOBAUD_EN
	UART_ERR_WR_MASK
	UART_TXD_INV
	UART_RXD_INV
	UART_IRDA_EN
	UART_TX_FLOW_EN
	UART_LOOPBACK
	UART_IRDA_RX_INV
	UART_IRDA_TX_EN
	UART_IRDA_WCTL
	UART_IRDA_TX_EN
	UART_IRDA_DPLX
	UART_STOP_BIT_NUM
	UART_BIT_NUM
	UART_PARITY_EN
	UART_PARITY
UART_FLOW_CONF_REG	UART_SEND_XOFF
	UART_SEND_XON
	UART_FORCE_XOFF
	UART_FORCE_XON
	UART_XONOFF_DEL
	UART_SW_FLOW_CON_EN
UART_TXBRK_CONF_REG	UART_RS485_TX_DLY_NUM[3:0]
	UART_RS485_RX_DLY_NUM
	UART_RS485RXBY_TX_EN
	UART_RS485TX_RX_EN
	UART_DL1_EN
	UART_DL0_EN
	UART_RS485_EN

13.5.1.2 静态寄存器

在作用于 UART Core 时钟域的寄存器中，有一部分寄存器不会在 UART 工作过程中动态修改，被认为是静态的，称为静态寄存器。静态寄存器没有做跨时钟域处理。静态寄存器的配置一定是 UART TX/RX 停止工作阶段，因此可以通过关闭 UART TX/RX 时钟的方式，保证配置寄存器的亚稳态不会被采样到。当打开 UART TX/RX 时钟打开时，软件配置的值已经稳定，从而确保配置的值被正确采样。表 13-2 列出了这些寄存器。对这些寄存器的配置流程如下：

- 根据当前停止工作的模块为 UART TX 还是 RX，将 UART_TX_SCLK_EN 或 UART_RX_SCLK_EN 清 0 关闭 UART Tx 或 RX 时钟；

- 配置静态寄存器；
- 向 UART_TX_SCLK_EN 或 UART_RX_SCLK_EN 写 1 打开 UART Tx 或 RX 时钟。

表 13-2. UARTn静态寄存器

寄存器	域名
UART_RX_FILT_REG	UART_GLITCH_FILT_EN
	UART_GLITCH_FILT[7:0]
UART_SLEEP_CONF_REG	UART_ACTIVE_THRESHOLD[9:0]
UART_SWFC_CONF0_REG	UART_XOFF_CHAR[7:0]
UART_SWFC_CONF1_REG	UART_XON_CHAR[7:0]
UART_IDLE_CONF_REG	UART_TX_IDLE_NUM[9:0]
UART_AT_CMD_PRECNT_REG	UART_PRE_IDLE_NUM[15:0]
UART_AT_CMD_POSTCNT_REG	UART_POST_IDLE_NUM[15:0]
UART_AT_CMD_GAPTOUT_REG	UART_RX_GAP_TOUT[15:0]
UART_AT_CMD_CHAR_REG	UART_CHAR_NUM[7:0]
	UART_AT_CMD_CHAR[7:0]

13.5.1.3 立即寄存器

除表13-1与13-2 外的所有软件可配置寄存器作用于 APB_CLK 时钟域，即为立即寄存器，例如，中断及 FIFO 配置寄存器等。

13.5.2 具体步骤

图13-13 显示了 UART 模块的编程流程。主要包括：初始化、寄存器配置、启动 UART TX/RX 和数据传输结束。

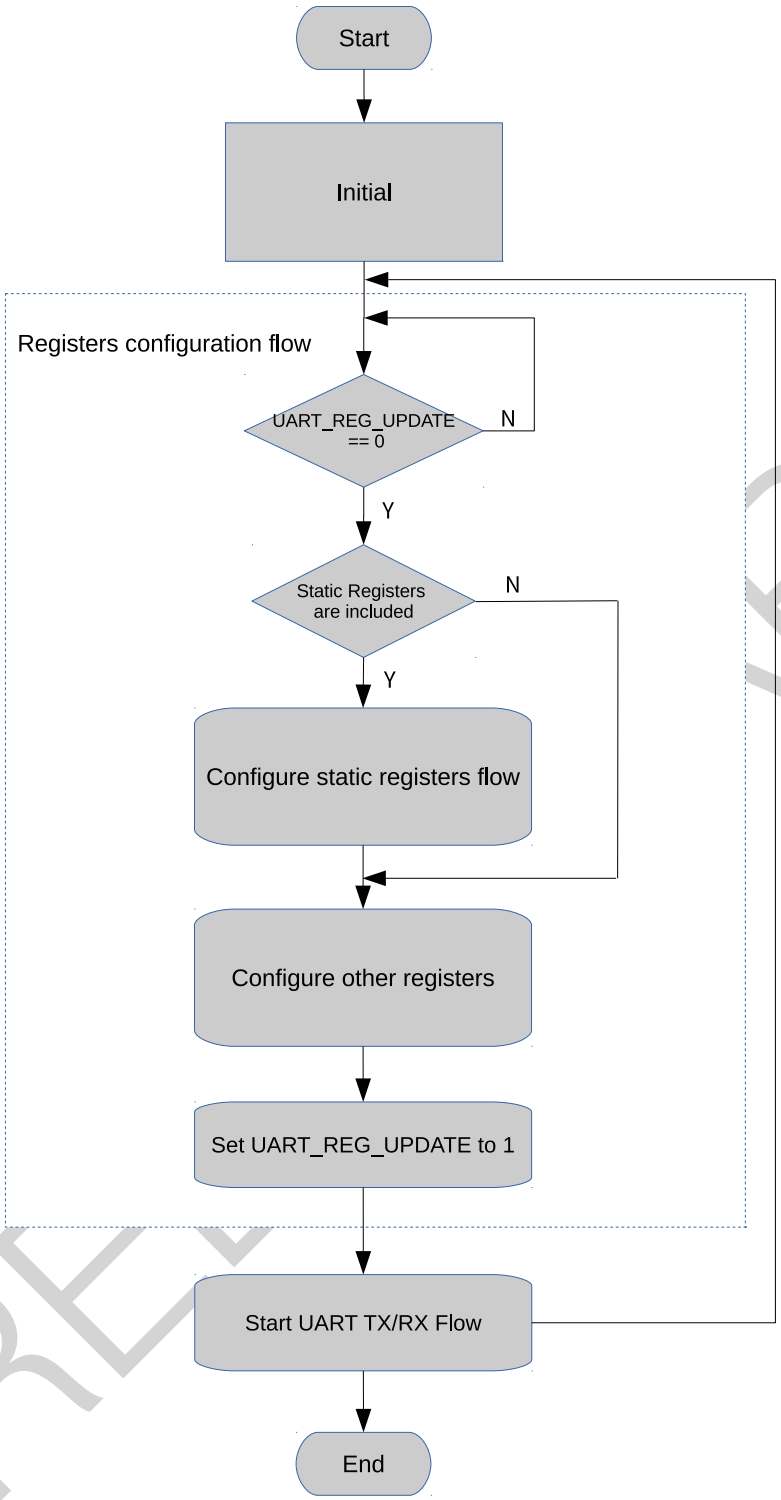


图 13-13. UART 编程流程

13.5.2.1 URAT*n* 模块初始化

URAT*n* 模块初始化流程如下：

- 将 SYSTEM_UART_MEM_CLK_EN 置 1 打开 UART RAM 时钟；
- 将 SYSTEM_UART*n*_CLK_EN 置 1 打开 UART*n* APB_CLK；
- 将寄存器 SYSTEM_UART*n*_RST 清 0；

- 向寄存器 UART_RST_CORE 写 1；
- 向寄存器 SYSTEM_UART*n*_RST 写 1；
- 将寄存器 SYSTEM_UART*n*_RST 清 0；
- 将寄存器 UART_RST_CORE 清 0；
- 将 UART_UPDATE_CTRL 清 0 使能寄存器同步功能。

13.5.2.2 URAT*n* 通信配置

URAT*n* 通信配置流程如下：

- 等待 UART_REG_UPDATE 为 0，确保上一次同步已经完成；
- 如果配置寄存器中包含静态寄存器，配置流程参考13.5.1.2完成配置；
- 配置 UART_SCLK_SEL 选择时钟源；
- 配置 UART_SCLK_DIV_NUM、UART_SCLK_DIV_A、UART_SCLK_DIV_B 设置预分频器系数；
- 配置 UART_CLKDIV、UART_CLKDIV_FRAG 设置发送波特率；
- 配置 UART_BIT_NUM 设置数据长度；
- 配置 UART_PARITY_EN、UART_PARITY 设置奇偶校验；
- 可选步骤，根据应用不同存在差异...
- 向 UART_REG_UPDATE 写 1，将配置的值同步到 Core 时钟域。

13.5.2.3 启动 URAT*n*

启动 UART*n* TX 发送数据：

- 配置 UART_TXFIFO_EMPTY_THRHD，设置 TXFIFO 空阈值；
- 对 UART_TXFIFO_EMPTY_INT_ENA 置 0，关闭 UART_TXFIFO_EMPTY_INT 中断；
- 向 UART_RXFIFO_RD_BYTE 写入需要发送的数据；
- 置位 UART_TXFIFO_EMPTY_INT_CLR，清除 UART_TXFIFO_EMPTY_INT 中断；
- 置位 UART_TXFIFO_EMPTY_INT_ENA，使能 UART_TXFIFO_EMPTY_INT 中断；
- 检测 UART_TXFIFO_EMPTY_INT，等待发送数据结束。

启动 UART*n* RX 数据接收：

- 配置 UART_RXFIFO_FULL_THRHD，设置 RXFIFO 满阈值；
- 置位 UART_RXFIFO_FULL_INT_ENA，使能 UART_RXFIFO_FULL_INT 中断；
- 检测 UART_TXFIFO_FULL_INT，等待 RXFIFO 接收数据满；
- 通过读 UART_RXFIFO_RD_BYTE，从 RXFIFO 中读出数据，并可通过 UART_RXFIFO_CNT 获得当前 RXFIFO 中的接收数据量。

13.6 寄存器列表

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

名称	描述	地址	访问
FIFO 配置			
UART_FIFO_REG	FIFO 数据寄存器	0x0000	RO
UART_MEM_CONF_REG	UART 阈值和分配配置	0x0060	R/W
中断寄存器			
UART_INT_RAW_REG	原始中断状态	0x0004	R/ WTC/ SS
UART_INT_ST_REG	屏蔽中断状态	0x0008	RO
UART_INT_ENA_REG	中断使能位	0x000C	R/W
UART_INT_CLR_REG	中断清除位	0x0010	WT
配置寄存器			
UART_CLKDIV_REG	时钟分频配置	0x0014	R/W
UART_RX_FILT_REG	RX 滤波器配置	0x0018	R/W
UART_CONF0_REG	配置寄存器 0	0x0020	R/W
UART_CONF1_REG	配置寄存器 1	0x0024	R/W
UART_FLOW_CONF_REG	软件流控配置	0x0034	varies
UART_SLEEP_CONF_REG	睡眠模式配置	0x0038	R/W
UART_SWFC_CONF0_REG	软件流控字符配置	0x003C	R/W
UART_SWFC_CONF1_REG	软件流控字符配置	0x0040	R/W
UART_TXBRK_CONF_REG	帧结束空闲配置	0x0044	R/W
UART_IDLE_CONF_REG	帧结束空闲配置	0x0048	R/W
UART_RS485_CONF_REG	RS485 模式配置	0x004C	R/W
UART_CLK_CONF_REG	UART core 时钟配置	0x0078	R/W
状态寄存器			
UART_STATUS_REG	UART 状态寄存器	0x001C	RO
UART_MEM_TX_STATUS_REG	TX FIFO 写入、读取偏移地址	0x0064	RO
UART_MEM_RX_STATUS_REG	RX FIFO 写入、读取偏移地址	0x0068	RO
UART_FSM_STATUS_REG	UART 发送和接收状态	0x006C	RO
自动波特率检测寄存器			
UART_LOWPULSE_REG	自动波特率检测最短低电平脉冲持续时间寄存器	0x0028	RO
UART_HIGHPULSE_REG	自动波特率检测最短高电平脉冲持续时间寄存器	0x002C	RO
UART_RXD_CNT_REG	自动波特率检测沿变化计数寄存器	0x0030	RO
UART_POSPULSE_REG	自动波特率检测高电平脉冲寄存器	0x0070	RO
UART_NEGPULSE_REG	自动波特率检测低电平脉冲寄存器	0x0074	RO
AT 转义序列检测配置			
UART_AT_CMD_PRECNT_REG	序列发送前的时序配置	0x0050	R/W
UART_AT_CMD_POSTCNT_REG	序列发送后的时序配置	0x0054	R/W
UART_AT_CMD_GAPTOUT_REG	超时配置	0x0058	R/W

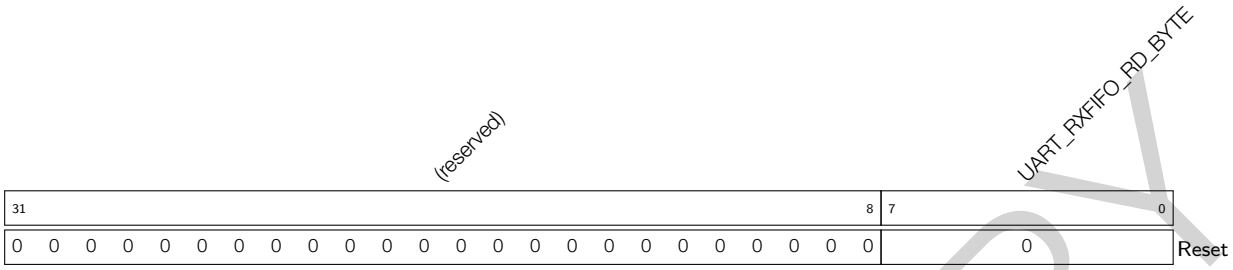
名称	描述	地址	访问
UART_AT_CMD_CHAR_REG	AT 转义序列检测配置	0x005C	R/W
版本寄存器			
UART_DATE_REG	UART 版本控制寄存器	0x007C	R/W
UART_ID_REG	UART ID 寄存器	0x0080	varies

名称	描述	地址	访问
配置寄存器			
UHCI_CONF0_REG	UHCI 配置寄存器	0x0000	R/W
UHCI_CONF1_REG	UHCI 配置寄存器	0x0014	varies
UHCI_ESCAPE_CONF_REG	转义符配置	0x0020	R/W
UHCI_HUNG_CONF_REG	超时配置	0x0024	R/W
UHCI_ACK_NUM_REG	配置 UHCI ACK 值	0x0028	varies
UHCI_QUICK_SENT_REG	UHCI 快速发送配置寄存器	0x0030	varies
UHCI_REG_Q0_WORD0_REG	Q0_WORD0 快速发送寄存器	0x0034	R/W
UHCI_REG_Q0_WORD1_REG	Q0_WORD1 快速发送寄存器	0x0038	R/W
UHCI_REG_Q1_WORD0_REG	Q1_WORD0 快速发送寄存器	0x003C	R/W
UHCI_REG_Q1_WORD1_REG	Q1_WORD1 快速发送寄存器	0x0040	R/W
UHCI_REG_Q2_WORD0_REG	Q2_WORD0 快速发送寄存器	0x0044	R/W
UHCI_REG_Q2_WORD1_REG	Q2_WORD1 快速发送寄存器	0x0048	R/W
UHCI_REG_Q3_WORD0_REG	Q3_WORD0 快速发送寄存器	0x004C	R/W
UHCI_REG_Q3_WORD1_REG	Q3_WORD1 快速发送寄存器	0x0050	R/W
UHCI_REG_Q4_WORD0_REG	Q4_WORD0 快速发送寄存器	0x0054	R/W
UHCI_REG_Q4_WORD1_REG	Q4_WORD1 快速发送寄存器	0x0058	R/W
UHCI_REG_Q5_WORD0_REG	Q5_WORD0 快速发送寄存器	0x005C	R/W
UHCI_REG_Q5_WORD1_REG	Q5_WORD1 快速发送寄存器	0x0060	R/W
UHCI_REG_Q6_WORD0_REG	Q6_WORD0 快速发送寄存器	0x0064	R/W
UHCI_REG_Q6_WORD1_REG	Q6_WORD1 快速发送寄存器	0x0068	R/W
UHCI_ESC_CONF0_REG	转义序列配置寄存器 0	0x006C	R/W
UHCI_ESC_CONF1_REG	转义序列配置寄存器 1	0x0070	R/W
UHCI_ESC_CONF2_REG	转义序列配置寄存器 2	0x0074	R/W
UHCI_ESC_CONF3_REG	转义序列配置寄存器 3	0x0078	R/W
UHCI_PKT_THRES_REG	包长度配置寄存器	0x007C	R/W
中断寄存器			
UHCI_INT_RAW_REG	原始中断状态	0x0004	varies
UHCI_INT_ST_REG	屏蔽中断状态	0x0008	RO
UHCI_INT_ENA_REG	中断使能位	0x000C	R/W
UHCI_INT_CLR_REG	中断清除位	0x0010	WT
UHCI 状态寄存器			
UHCI_STATE0_REG	UHCI 接收状态	0x0018	RO
UHCI_STATE1_REG	UHCI transmit status	0x001C	RO
UHCI_RX_HEAD_REG	UHCI 包报头寄存器	0x002C	RO
版本寄存器			
UHCI_DATE_REG	UHCI 版本控制寄存器	0x0080	R/W

13.7 寄存器

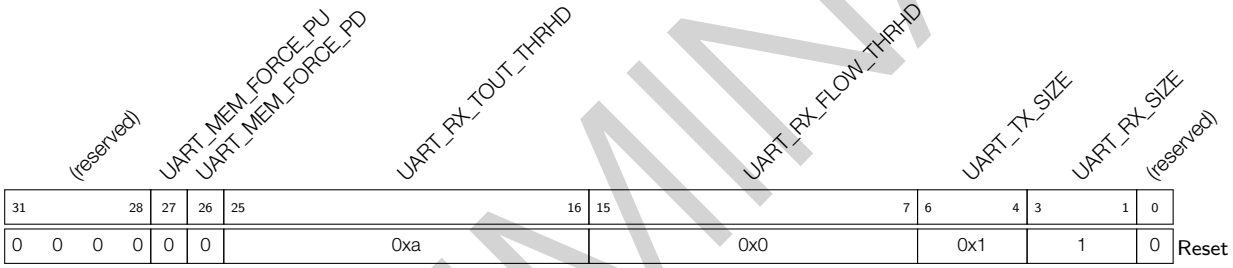
本小节的所有地址均为相对于 UART 控制器基地址的地址偏移量（相对地址），具体基地址请见章节 3 *系统和存储器* 中的表 3-4。

Register 13.1. UART_FIFO_REG (0x0000)



UART_RXFIFO_RD_BYTE UART*n* 通过此寄存器访问 FIFO。(RO)

Register 13.2. UART_MEM_CONF_REG (0x0060)



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

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

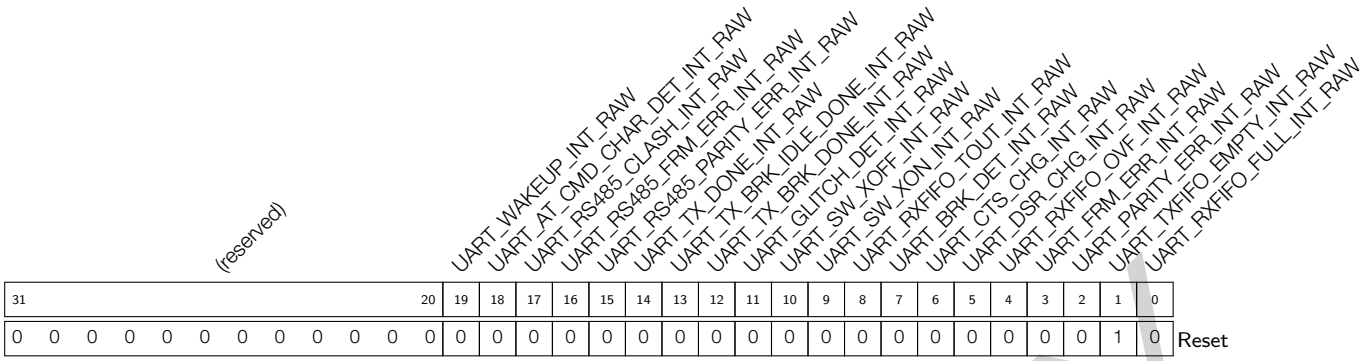
UART_RX_FLOW_THRHD 配置使用硬件流控时接收数据的最大值。(R/W)

UART_RX_TOUT_THRHD 配置接收器接收一个字节所需时间的阈值，单位是比特时间（即传输一个比特所需的时间）。接收器接收一个字节所需时间超过阈值且 UART_RX_TOUT_EN 置 1 时触发 UART_RXFIFO_TOUT_INT 中断。(R/W)

UART_MEM_FORCE_PD 置位此位强制关闭 UART 存储器。(R/W)

UART_MEM_FORCE_PU 置位此位强制开启 UART 存储器。(R/W)

Register 13.3. UART_INT_RAW_REG (0x0004)



UART_RXFIFO_FULL_INT_RAW 接收器接收数据多于 UART_RXFIFO_FULL_THRHD 的值时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_TXFIFO_EMPTY_INT_RAW TX FIFO 中的数据少于 UART_TXFIFO_EMPTY_THRHD 的值时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_PARITY_ERR_INT_RAW 接收器检测到数据奇偶检验位错误时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_FRM_ERR_INT_RAW 接收器检测到数据帧错误时，该原始中断位翻转至高电平。(R/WTC/SS)

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

UART_DSR_CHG_INT_RAW 接收器检测到 DSRn 信号的沿变化时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_CTS_CHG_INT_RAW 接收器检测到 CTSn 信号的沿变化时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_BRK_DET_INT_RAW 接收器在停止位后检测到 0 时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_RXFIFO_TOUT_INT_RAW 接收器接收一个字节所需时间超过 UART_RX_TOUT_THRHD 时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_SW_XON_INT_RAW 接收器接收到 XON 字符且 UART_SW_FLOW_CON_EN 置 1 时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_SW_XOFF_INT_RAW 接收器接收到 XOFF 字符且 UART_SW_FLOW_CON_EN 置 1 时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_GLITCH_DET_INT_RAW 接收器在起始位的中点处检测到毛刺时，该原始中断位翻转至高电平。(R/WTC/SS)

见下页...

Register 13.3. UART_INT_RAW_REG (0x0004)

接上页...

UART_TX_BRK_DONE_INT_RAW 发送器在发送完 TX FIFO 中所有数据后完成 NULL 字符的发送时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_TX_BRK_IDLE_DONE_INT_RAW 发送器发送完最后一个数据后的间隔时间达到阈值时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_TX_DONE_INT_RAW 发送器发完 FIFO 中的所有数据后，该原始中断位翻转至高电平。(R/WTC/SS)

UART_RS485_PARITY_ERR_INT_RAW RS485 模式下接收器检测到发送器回音的数据检验位错误时，该原始中断位翻转至高电平。(R/WTC/SS)

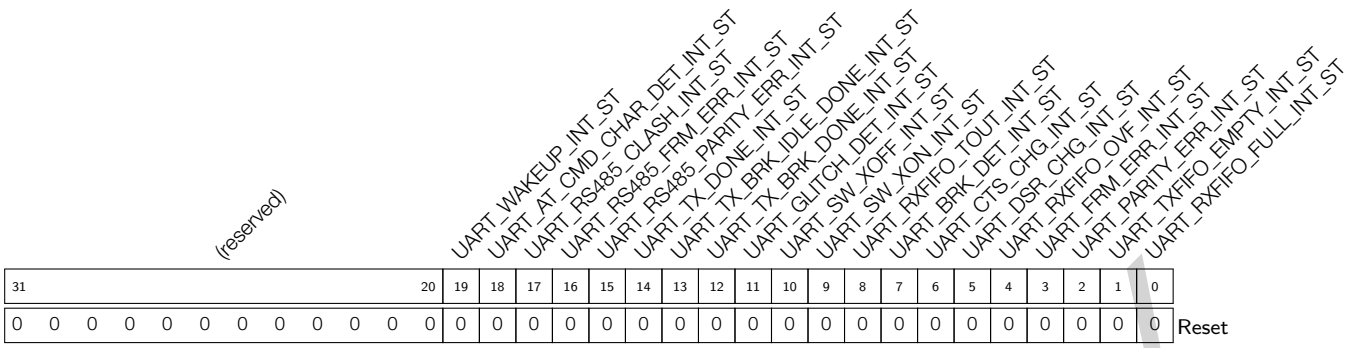
UART_RS485_FRM_ERR_INT_RAW RS485 模式下接收器检测到发送器回音的数据帧错误时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_RS485_CLASH_INT_RAW RS485 模式下检测到发送器与接收器冲突时，该原始中断位翻转至高电平。(R/WTC/SS)

UART_AT_CMD_CHAR_DET_INT_RAW 接收器检测到配置的 UART_AT_CMD CHAR 时，该原始中断位翻转至高电平。(R/WTC/SS)

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

Register 13.4. UART_INT_ST_REG (0x0008)



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

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

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

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

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

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

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

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

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

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

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

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

见下页...

Register 13.4. UART_INT_ST_REG (0x0008)

接上页...

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

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

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

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

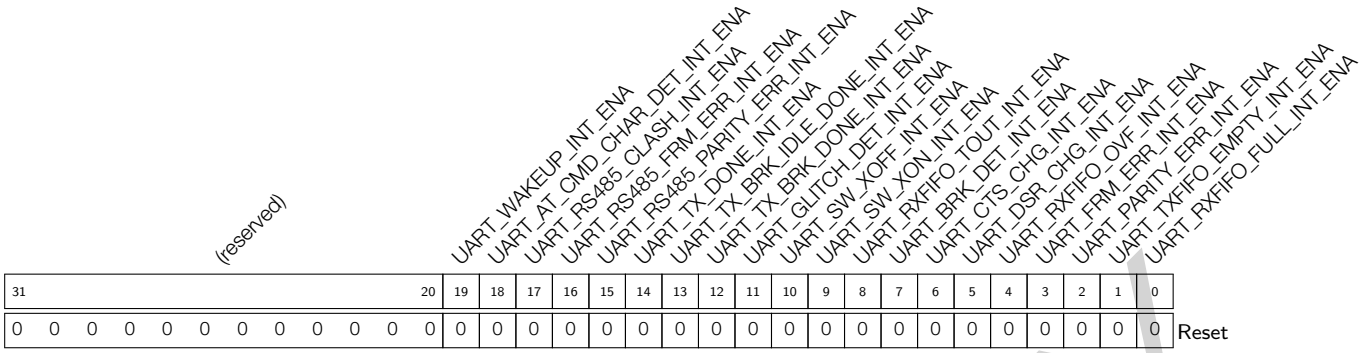
UART_RS485_FRM_ERR_INT_ST UART_RS485_FM_ERR_INT_ENA 置 1 时 UART_RS485_FRM_ERR_INT_RAW 的状态位。(RO)

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

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

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

Register 13.5. UART_INT_ENA_REG (0x000C)



UART_RXFIFO_FULL_INT_ENA UART_RXFIFO_FULL_INT_ST 寄存器的使能位。(R/W)

UART_TXFIFO_EMPTY_INT_ENA UART_TXFIFO_EMPTY_INT_ST 寄存器的使能位。(R/W)

UART_PARITY_ERR_INT_ENA UART_PARITY_ERR_INT_ST 寄存器的使能位。(R/W)

UART_FRM_ERR_INT_ENA UART_FRM_ERR_INT_ST 寄存器的使能位。(R/W)

UART_RXFIFO_OVF_INT_ENA UART_RXFIFO_OVF_INT_ST 寄存器的使能位。(R/W)

UART_DSR_CHG_INT_ENA UART_DSR_CHG_INT_ST 寄存器的使能位。(R/W)

UART_CTS_CHG_INT_ENA UART_CTS_CHG_INT_ST 寄存器的使能位。(R/W)

UART_BRK_DET_INT_ENA UART_BRK_DET_INT_ST 寄存器的使能位。(R/W)

UART_RXFIFO_TOUT_INT_ENA UART_RXFIFO_TOUT_INT_ST 寄存器的使能位。(R/W)

UART_SW_XON_INT_ENA UART_SW_XON_INT_ST 寄存器的使能位。(R/W)

UART_SW_XOFF_INT_ENA UART_SW_XOFF_INT_ST 寄存器的使能位。(R/W)

UART_GLITCH_DET_INT_ENA UART_GLITCH_DET_INT_ST 寄存器的使能位。(R/W)

UART_TX_BRK_DONE_INT_ENA UART_TX_BRK_DONE_INT_ST 寄存器的使能位。(R/W)

UART_TX_BRK_IDLE_DONE_INT_ENA UART_TX_BRK_IDLE_DONE_INT_ST 寄存器的使能位。(R/W)

UART_TX_DONE_INT_ENA UART_TX_DONE_INT_ST 寄存器的使能位。(R/W)

UART_RS485_PARITY_ERR_INT_ENA UART_RS485_PARITY_ERR_INT_ST 寄存器的使能位。(R/W)

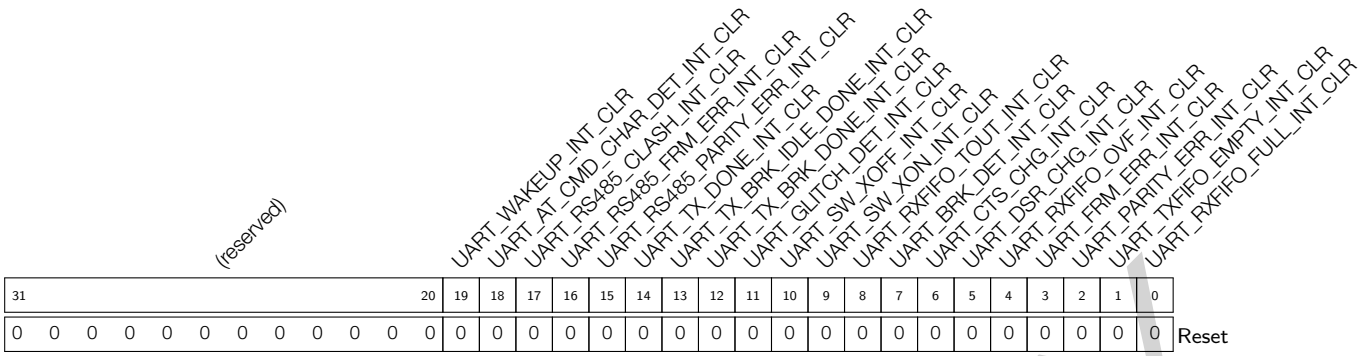
UART_RS485_FRM_ERR_INT_ENA UART_RS485_PARITY_ERR_INT_ST 寄存器的使能位。(R/W)

UART_RS485_CLASH_INT_ENA UART_RS485_CLASH_INT_ST 寄存器的使能位。(R/W)

UART_AT_CMD_CHAR_DET_INT_ENA UART_AT_CMD_CHAR_DET_INT_ST 寄存器的使能位。(R/W)

UART_WAKEUP_INT_ENA UART_WAKEUP_INT_ST 寄存器的使能位。(R/W)

Register 13.6. UART_INT_CLR_REG (0x0010)



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

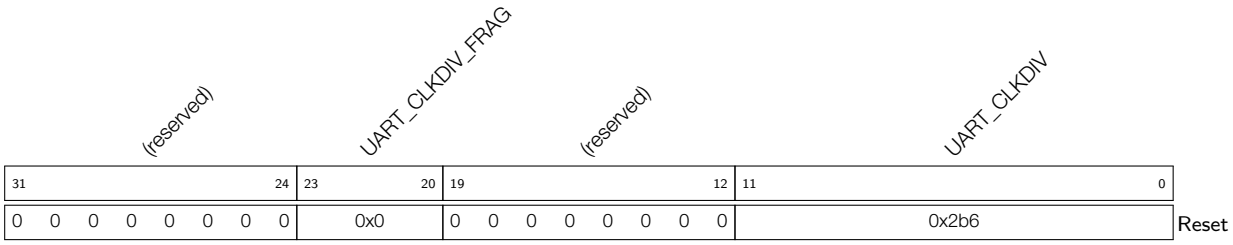
UART_RS485_FRM_ERR_INT_CLR 置位此位清除 UART_RS485_FRM_ERR_INT_RAW 中断。(WT)

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

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

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

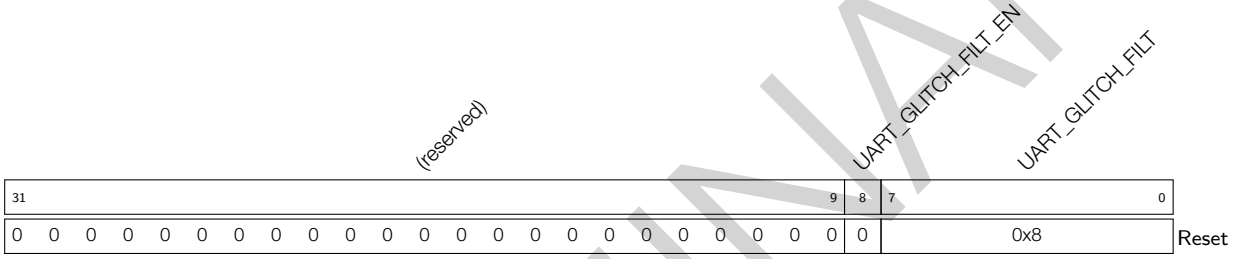
Register 13.7. UART_CLKDIV_REG (0x0014)



UART_CLKDIV 分频系数的整数部分。(R/W)

UART_CLKDIV_FRAG 分频系数的小数部分。(R/W)

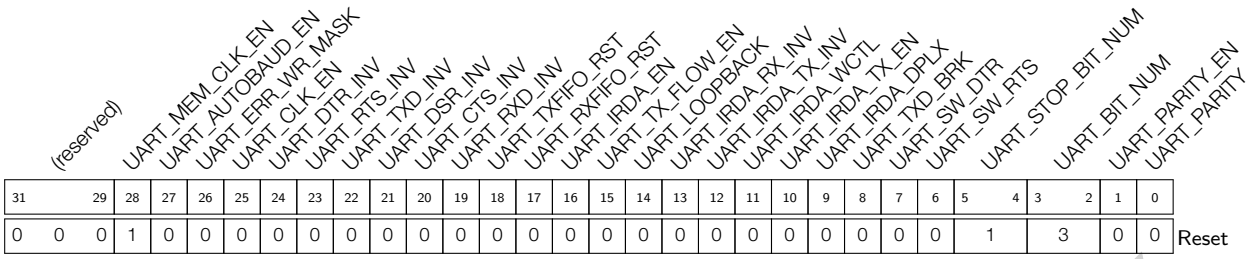
Register 13.8. UART_RX_FILT_REG (0x0018)



UART_GLITCH_FILT 宽度小于该寄存器值的输入脉冲会被忽略。(R/W)

UART_GLITCH_FILT_EN 置位此位，使能 RX 信号滤波器。(R/W)

Register 13.9. UART_CONF0_REG (0x0020)



UART_PARITY 配置奇偶检验方式。(R/W)

UART_PARITY_EN 置位此位使能 UART 奇偶检验。(R/W)

UART_BIT_NUM 设置数据长度。(R/W)

UART_STOP_BIT_NUM 设置停止位的长度。(R/W)

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

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

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

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

UART_IRDA_TX_EN IrDA 发送器的启动使能位。(R/W)

UART_IRDA_WCTL 1'h1：IrDA 发送器的第 11 位与第 10 位相同。1'h0：将 IrDA 发送器的第 11 位置 0。(R/W)

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

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

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

UART_TX_FLOW_EN 置位此位使能发送器的流控功能。(R/W)

UART_IRDA_EN 置位此位使能 IrDA 协议。(R/W)

UART_RXFIFO_RST 置位此位复位 UART RX FIFO。(R/W)

UART_TXFIFO_RST 置位此位复位 UART TX FIFO。(R/W)

UART_RXD_INV 置位此位翻转 UART RXD 信号电平。(R/W)

UART_CTS_INV 置位此位翻转 UART CTS 信号电平。(R/W)

UART_DSR_INV 置位此位翻转 UART DSR 信号电平。(R/W)

UART_TXD_INV 置位此位翻转 UART TXD 信号电平。(R/W)

UART_RTS_INV 置位此位翻转 UART RTS 信号电平。(R/W)

UART_DTR_INV 置位此位翻转 UART DTR 信号电平。(R/W)

见下页...

Register 13.9. UART_CONF0_REG (0x0020)

接上页...

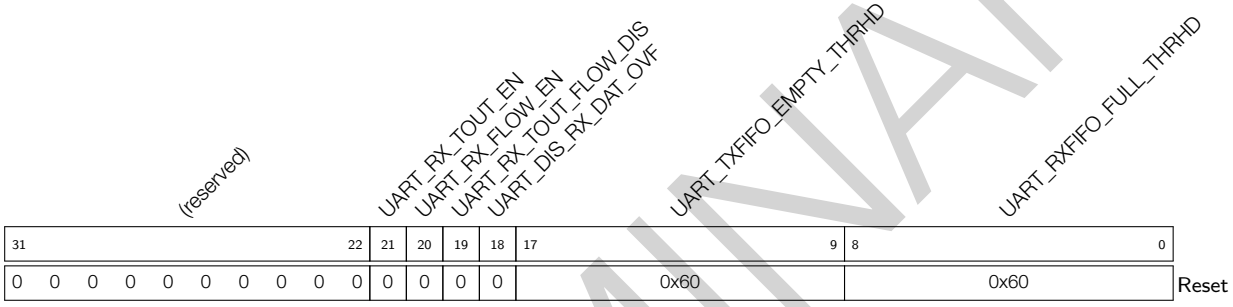
UART_CLK_EN 1'h1：强制为寄存器开启时钟。1' h0：仅在应用写寄存器时支持时钟。(R/W)

UART_ERR_WR_MASK 1'h1：若数据错误，接收器不再将数据存入 FIFO。1' h0：若数据错误，接收器仍存储。(R/W)

UART_AUTOBAUD_EN 波特率检测的使能信号。(R/W)

UART_MEM_CLK_EN UART 存储器门控使能信号。(R/W)

Register 13.10. UART_CONF1_REG (0x0024)



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

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

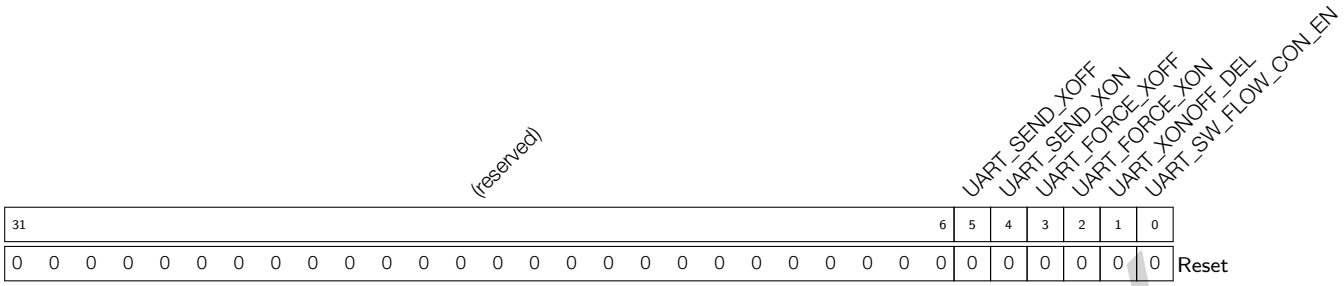
UART_DIS_RX_DAT_OVF 关闭 UART RX 数据溢出检测。(R/W)

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

UART_RX_FLOW_EN UART 接收器流控功能的使能位。(R/W)

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

Register 13.11. UART_FLOW_CONF_REG (0x0034)



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

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

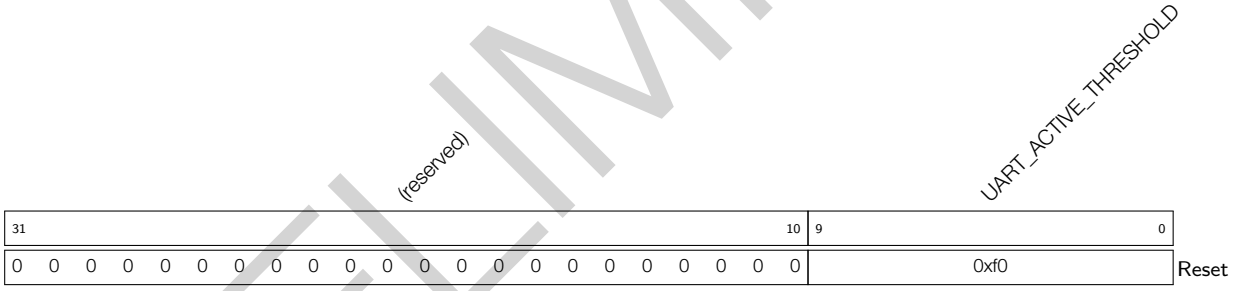
UART_FORCE_XON 置位此位让发送器继续发送数据。(R/W)

UART_FORCE_XOFF 置位此位让发送器停止发送数据。(R/W)

UART_SEND_XON 置位此位发送 XON 字符。此位由硬件自动清除。(R/W/SS/SC)

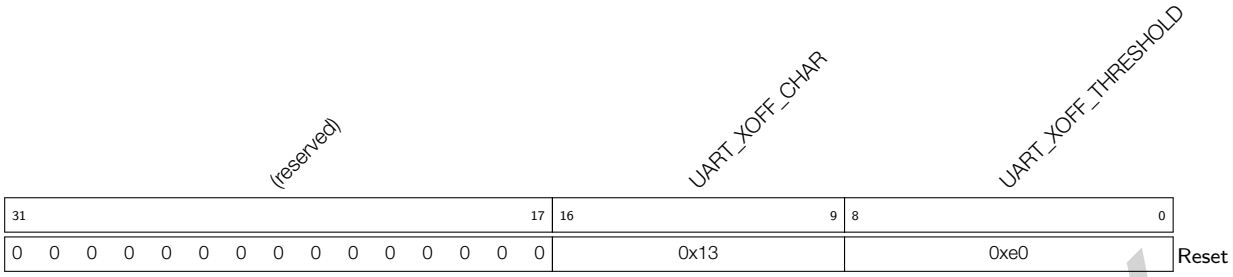
UART_SEND_XOFF 置位此位发送 XOFF 字符。此位由硬件自动清除。(R/W/SS/SC)

Register 13.12. UART_SLEEP_CONF_REG (0x0038)



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

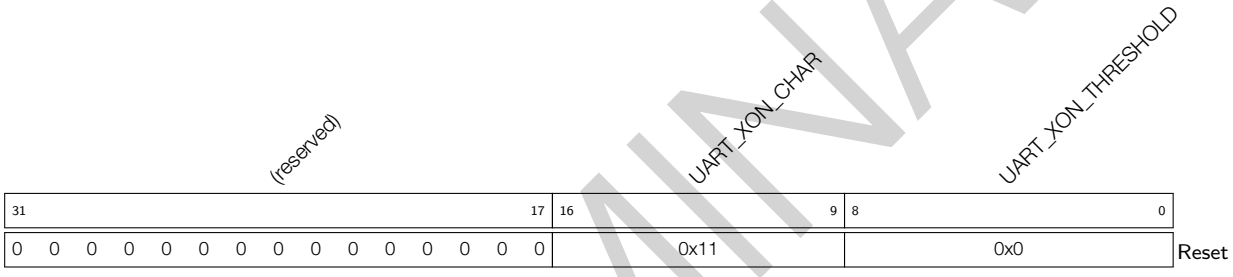
Register 13.13. UART_SWFC_CONF0_REG (0x003C)



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

UART_XOFF_CHAR 存储 XOFF 流控字符。(R/W)

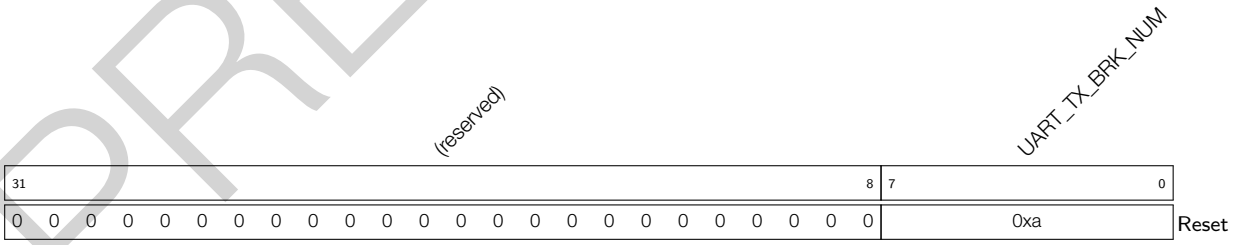
Register 13.14. UART_SWFC_CONF1_REG (0x0040)



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

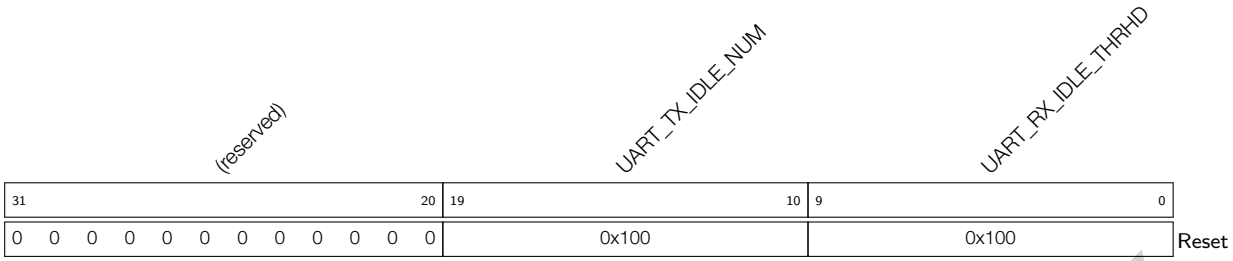
UART_XON_CHAR 存储 XON 流控字符。(R/W)

Register 13.15. UART_TXBRK_CONF_REG (0x0044)



UART_TX_BRK_NUM 配置数据发完后待发 NULL 字符的数量。UART_TXD_BRK 置 1 时有意义。(R/W)

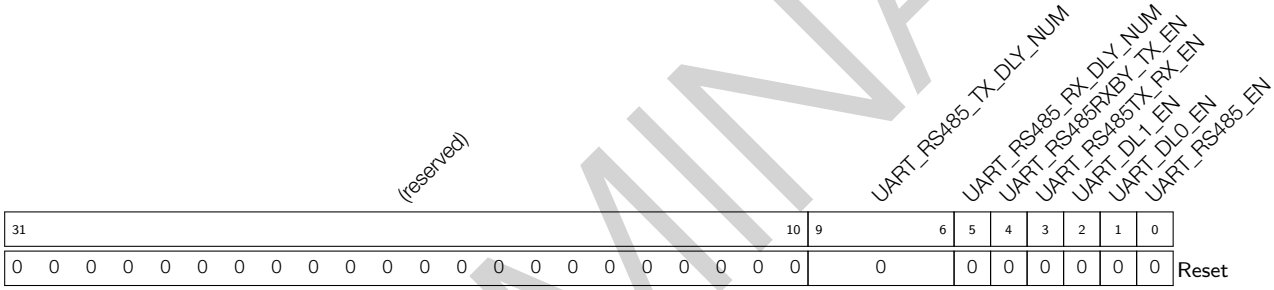
Register 13.16. UART_IDLE_CONF_REG (0x0048)



UART_RX_IDLE_THRHD 接收器接收一字节数据所需时间超过该寄存器的值时产生帧结束信号，单位是比特时间（即传输一个比特所需的时间）。(R/W)

UART_TX_IDLE_NUM 配置两次数据传输的间隔时间，单位是比特时间（即传输一个比特所需的时间）。(R/W)

Register 13.17. UART_RS485_CONF_REG (0x004C)



UART_RS485_EN 置位此位选择 RS485 模式。(R/W)

UART_DL0_EN 置位此位，延迟停止位 1 位。(R/W)

UART_DL1_EN 置位此位，延迟停止位 1 位。(R/W)

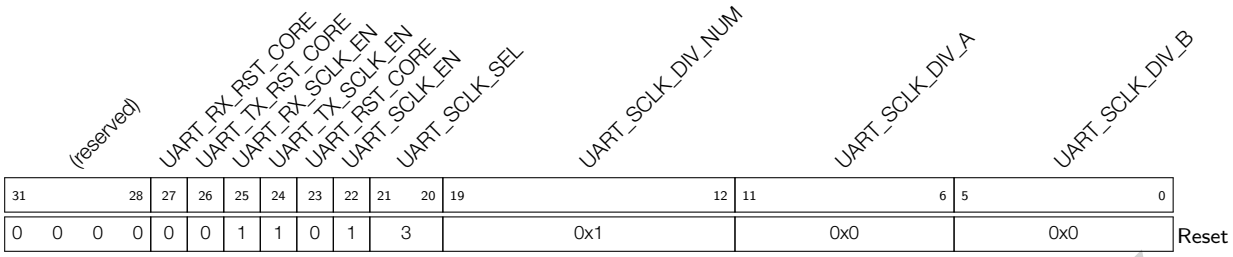
UART_RS485TX_RX_EN 发送器在 RS485 模式下发送数据时，置位此位使能接收器接收数据。(R/W)

UART_RS485RXBY_TX_EN 1'h1：RS485 接收器线路繁忙时使能 RS485 发送器发送数据。(R/W)

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

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

Register 13.18. UART_CLK_CONF_REG (0x0078)



UART_SCLK_DIV_B 分频系数的分母。(R/W)

UART_SCLK_DIV_A 分频系数的分子。(R/W)

UART_SCLK_DIV_NUM 分频系数的整数部分。(R/W)

UART_SCLK_SEL 选择 UART 时钟源。1：APB_CLK；2：RTC20M_CLK；3：XTAL_CLK。(R/W)

UART_SCLK_EN 置位此位，使能 UART TX/RX 使能。(R/W)

UART_RST_CORE 向此位先写 1 后写 0，复位 UART TX/RX。(R/W)

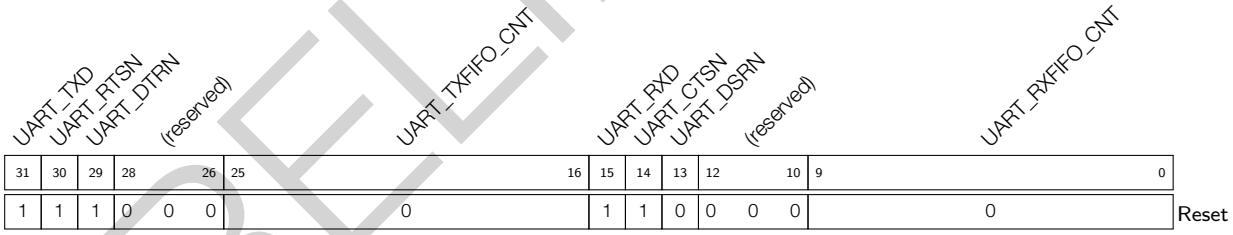
UART_TX_SCLK_EN 置位此位，使能 UART TX 时钟。(R/W)

UART_RX_SCLK_EN 置位此位，使能 UART RX 时钟。(R/W)

UART_TX_RST_CORE 向此位先写 1 后写 0，复位 UART TX。(R/W)

UART_RX_RST_CORE 向此位先写 1 后写 0，复位 UART RX。(R/W)

Register 13.19. UART_STATUS_REG (0x001C)



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

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

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

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

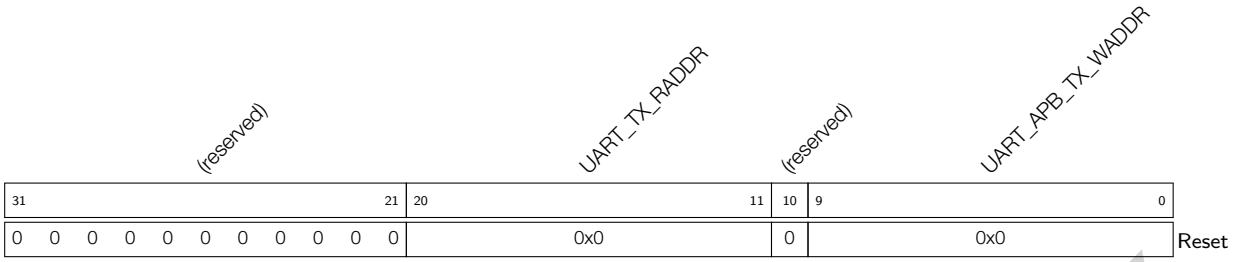
UART_TXFIFO_CNT 存储 TX FIFO 中数据的字节数。(RO)

UART_DTRN 此位表示内部 UART DTR 信号的电平。(RO)

UART_RTSN 此位表示内部 UART RTS 信号的电平。(RO)

UART_TXD 此位表示内部 UART TXD 信号的电平。(RO)

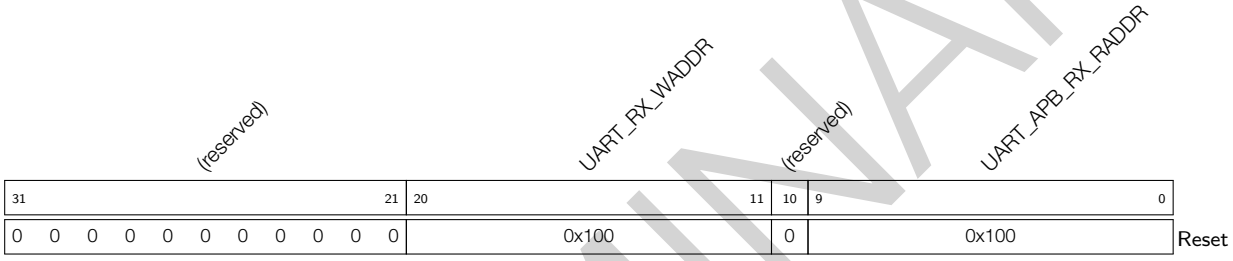
Register 13.20. UART_MEM_TX_STATUS_REG (0x0064)



UART_APB_TX_WADDR 在软件通过 APB 总线写 TX FIFO 时存储 TX FIFO 的偏移地址。(RO)

UART_TX_RADDR 在 TX FSM 通过 Tx_FIFO_Ctrl 读取数据时存储 TX FIFO 的偏移地址。(RO)

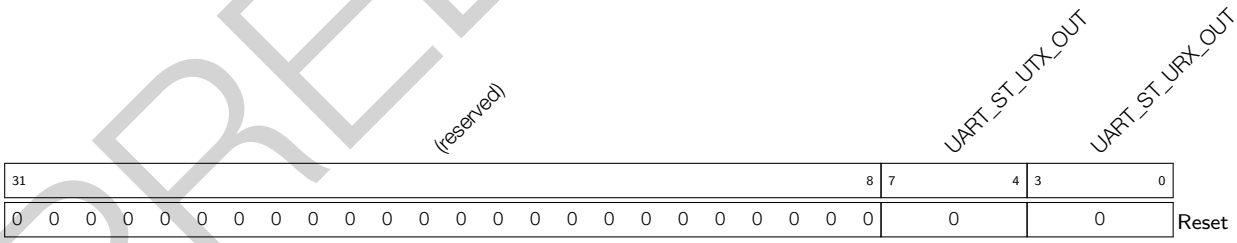
Register 13.21. UART_MEM_RX_STATUS_REG (0x0068)



UART_APB_RX_RADDR 在软件通过 APB 总线读取 RX FIFO 数据时存储 RX FIFO 的偏移地址。UART0 为 10'h100，UART1 为 10'h180。(RO)

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

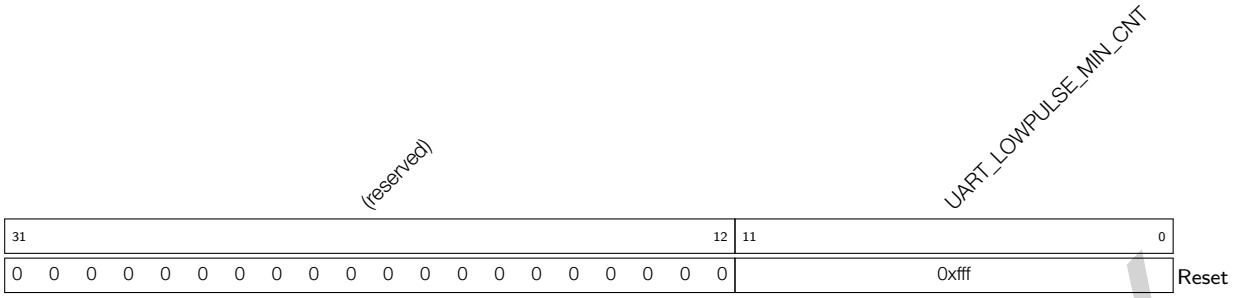
Register 13.22. UART_FSM_STATUS_REG (0x006C)



UART_ST_URX_OUT 接收器的状态寄存器。(RO)

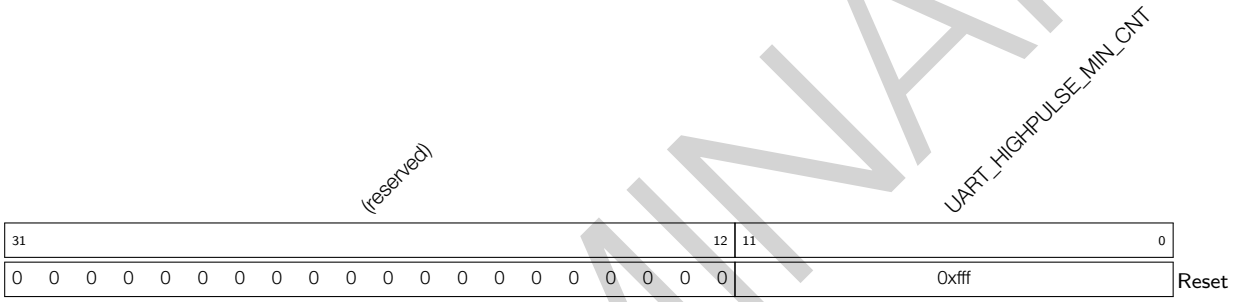
UART_ST_UTX_OUT 发送器的状态寄存器。(RO)

Register 13.23. UART_LOWPULSE_REG (0x0028)



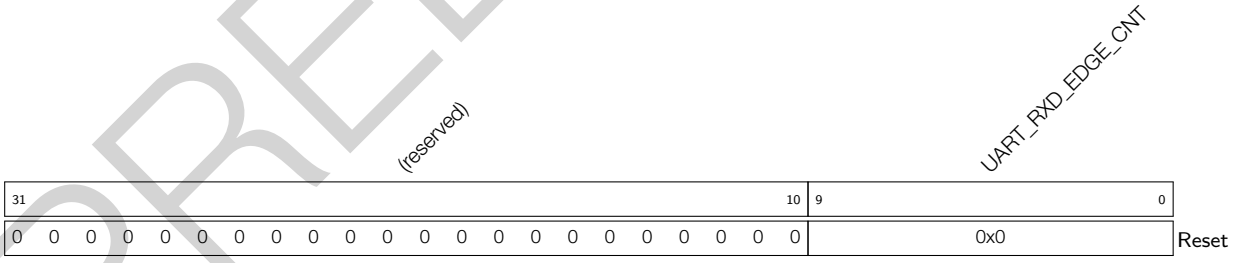
UART_LOWPULSE_MIN_CNT 存储低电平脉冲的最短持续时间，用于波特率检测，单位是 APB_CLK 时钟周期。(RO)

Register 13.24. UART_HIGHPULSE_REG (0x002C)



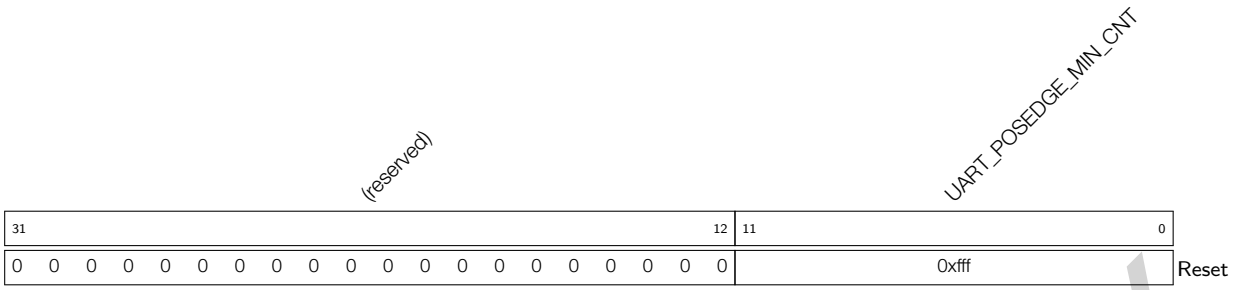
UART_HIGHPULSE_MIN_CNT 存储最长高电平脉冲持续时间。用于波特率检测，单位是 APB_CLK 时钟周期。(RO)

Register 13.25. UART_RXD_CNT_REG (0x0030)



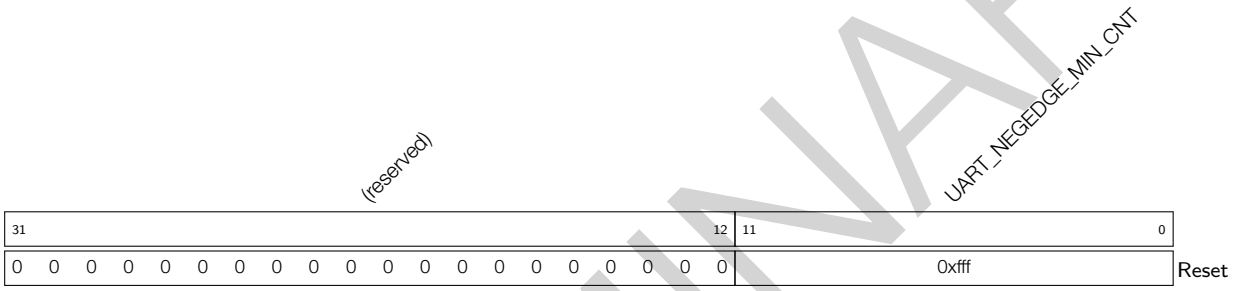
UART_RXD_EDGE_CNT 存储 RXD 沿变化的次数。用于波特率检测。(RO)

Register 13.26. UART_POSPULSE_REG (0x0070)



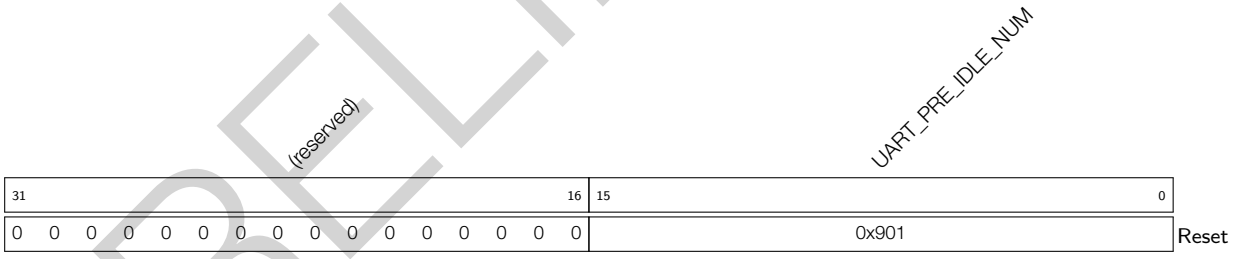
UART_POSEDGE_MIN_CNT 存储两个上升沿之间的最小输入时钟计数值。用于波特率检测。(RO)

Register 13.27. UART_NEGPULSE_REG (0x0074)



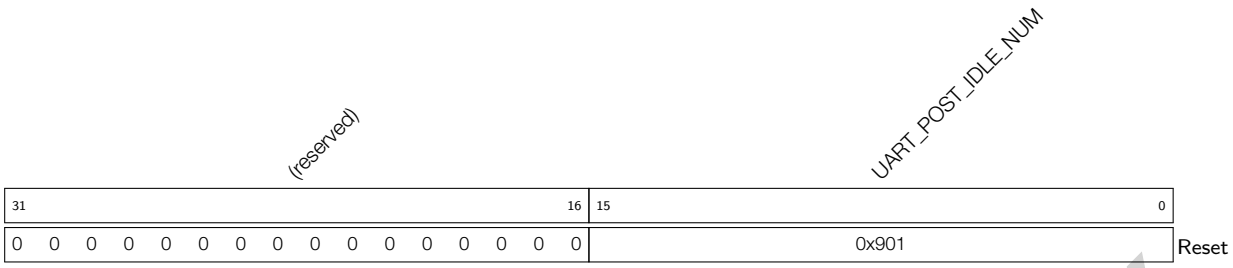
UART_NEGEDGE_MIN_CNT 存储两个下降沿之间的最小输入时钟计数值。用于波特率检测。(RO)

Register 13.28. UART_AT_CMD_PRECNT_REG (0x0050)



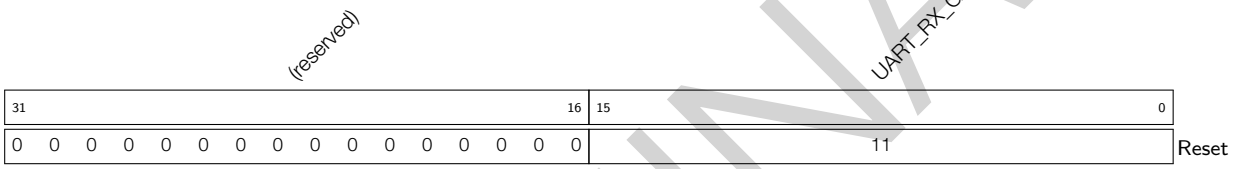
UART_PRE_IDLE_NUM 配置接收器接收第一个 AT_CMD 字符前的空闲时间，单位是比特时间（即传输一个比特所需的时间）。(R/W)

Register 13.29. UART_AT_CMD_POSTCNT_REG (0x0054)



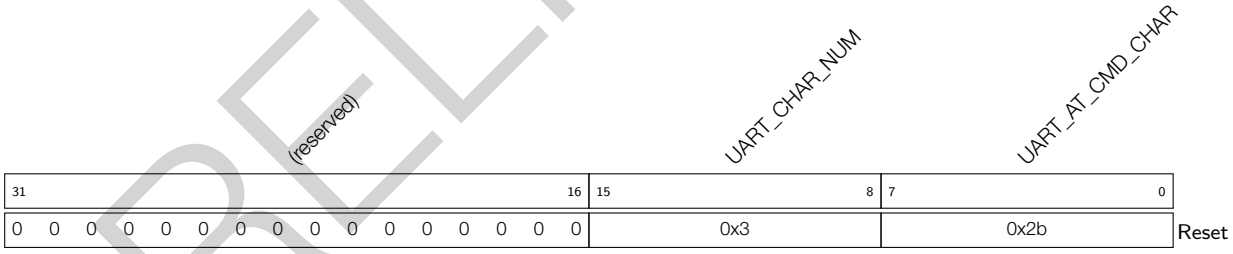
UART_POST_IDLE_NUM 配置最后一个 AT_CMD 字符和后续数据的间隔时间，单位是比特时间（即传输一个比特所需的时间）。(R/W)

Register 13.30. UART_AT_CMD_GAPTOUT_REG (0x0058)



UART_RX_GAP_TOUT 配置 AT_CMD 字符的间隔时间，单位是比特时间（即传输一个比特所需的时间）。(R/W)

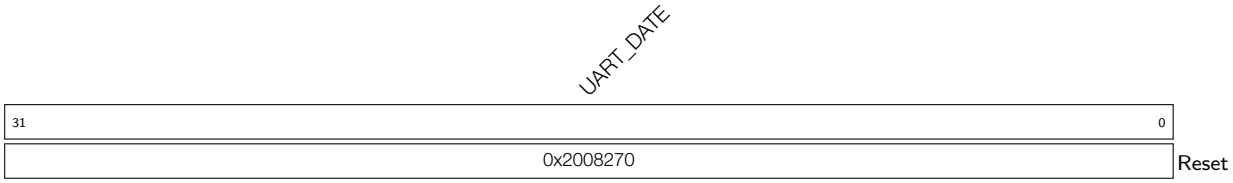
Register 13.31. UART_AT_CMD_CHAR_REG (0x005C)



UART_AT_CMD_CHAR 配置 AT_CMD 字符的内容。(R/W)

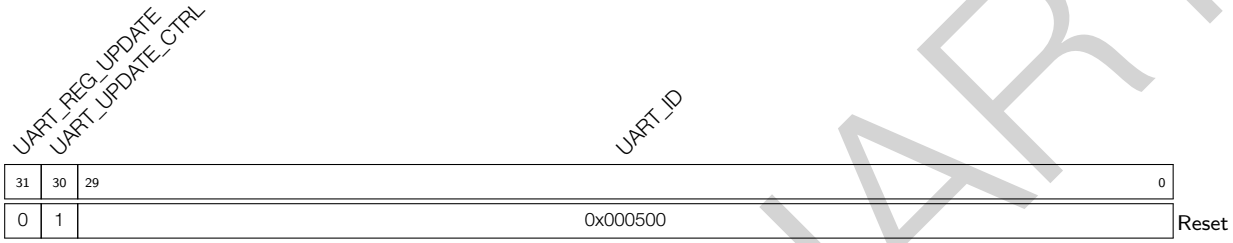
UART_CHAR_NUM 配置接收器接收连续 AT_CMD 字符的个数。(R/W)

Register 13.32. UART_DATE_REG (0x007C)



UART_DATE 版本控制寄存器。(R/W)

Register 13.33. UART_ID_REG (0x0080)



UART_ID 配置 UART_ID。(R/W)

UART_UPDATE_CTRL 用于控制同步模式。在向域 UART_REG_UPDATE 写 1 同步配置寄存器至 UART Core 时钟域之前，该域必须配置为 0。(R/W)

UART_REG_UPDATE 软件向该字段写 1，将寄存器值同步到 UART Core 时钟域。该字段在同步完成后由硬件自清。(R/W/SC)

Register 13.34. UHCI_CONF0_REG (0x0000)

UHCI_TX_RST 向此位先写 1 再写 0 复位解码状态机。(R/W)

UHCI_RX_RST 向此位先写 1 再写 0 复位编码状态机。(R/W)

UHCI_UART0_CE 置位此位，将 HCI 和 UART0 相连。(R/W)

UHCI_UART1_CE 置位此位，将 HCI 和 UART1 相连。(R/W)

UHCI_SEPER_EN 置位此位，使用特殊字符分隔数据帧。(R/W)

UHCI_HEAD_EN 置位此位，用格式报头编码数据包。(R/W)

UHCI_CRC_REC_EN 置位此位，使能 UHCI 接收 16 位 CRC。(R/W)

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

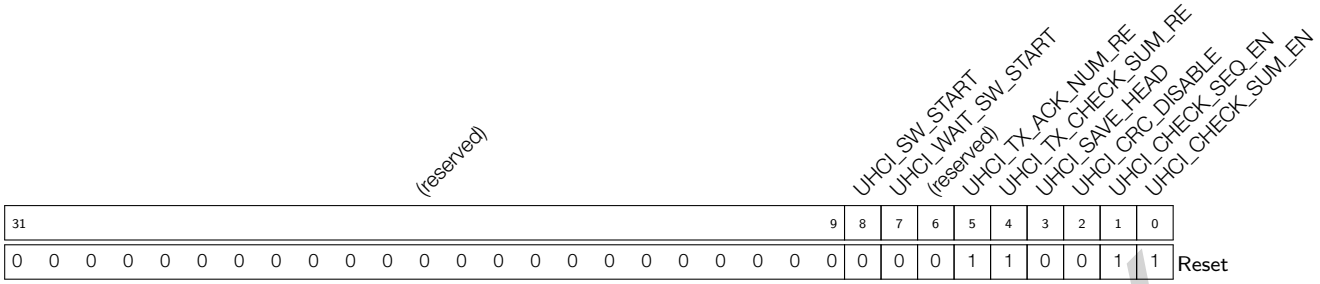
UHCI_LEN_EOF_EN 若此位置 1，UHCI 解码器接收字节数达到指定值时停止接收有效载荷数据。UHCI_HEAD_EN 为 1 时，该值是 UCHI 数据包报头明确的有效负载长度；UHCI_HEAD_EN 为 0 时，该值为配置值。若此位置 0，UHCI 解码器在接收到 0xc0 后停止接收有效载荷数据。(R/W)

UHCI_ENCODE_CRC_EN 置位此位，在有效载荷末尾加 16 位 CCITT-CRC 开始数据完整性检测。(R/W)

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

UHCI_UART_RX_BRK_EOF_EN 若此位置 1，UART 收到 NULL 帧后 UHCI 会停止接收有效载荷。(R/W)

Register 13.35. UHCI_CONF1_REG (0x0014)



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

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

UHCI_CRC_DISABLE 置位此位，支持 CRC 计算。UHCI 包中的数据完整性检测位应为 1。(R/W)

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

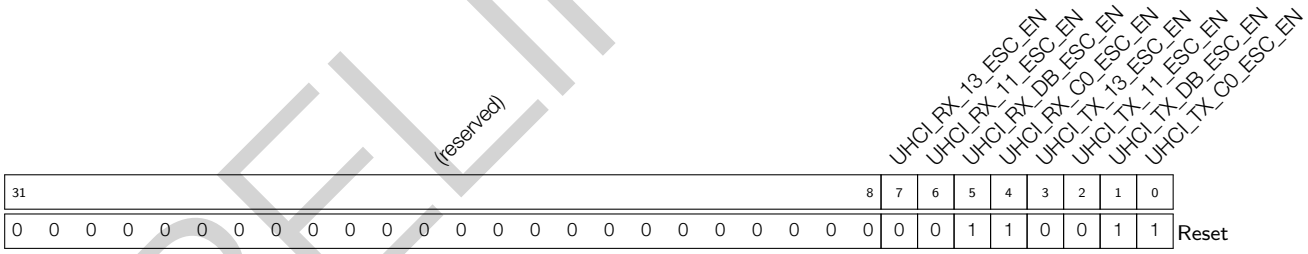
UHCI_TX_CHECK_SUM_RE 置位此位，用校验和编码数据包。(R/W)

UHCI_TX_ACK_NUM_RE 准备发送可靠数据包时，置位此位用 ACK 编码该数据包。(R/W)

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

UHCI_SW_START 若当前 UHCI_ENCODE_STATE 为 ST_SW_WAIT 状态，此位置 1 时 UHCI 开始发送数据包。(R/W/SC)

Register 13.36. UHCI_ESCAPE_CONF_REG (0x0020)



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

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

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

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

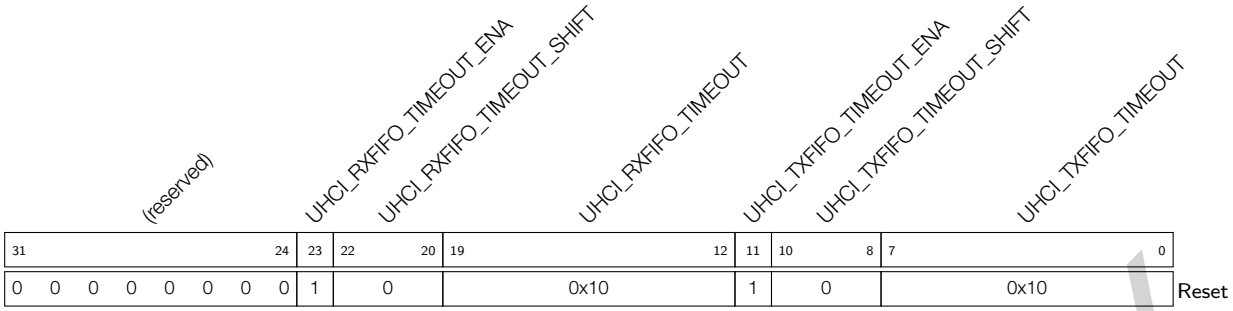
UHCI_RX_C0_ESC_EN 置位此位，在 DMA 发送数据时用特殊字符替换字符 0xc0。(R/W)

UHCI_RX_DB_ESC_EN 置位此位，在 DMA 发送数据时用特殊字符替换字符 0xdb。(R/W)

UHCI_RX_11_ESC_EN 置位此位，在 DMA 发送数据时用特殊字符替换流控字符 0x11。(R/W)

UHCI_RX_13_ESC_EN 置位此位，在 DMA 发送数据时用特殊字符替换流控字符 0x13。(R/W)

Register 13.37. UHCI_HUNG_CONF_REG (0x0024)



UHCI_TXFIFO_TIMEOUT . 存储超时值。DMA 接收数据超时时产生 UHCI_TX_HUNG_INT 中断。(R/W)

UHCI_TXFIFO_TIMEOUT_SHIFT 用于配置计数最大值。(R/W)

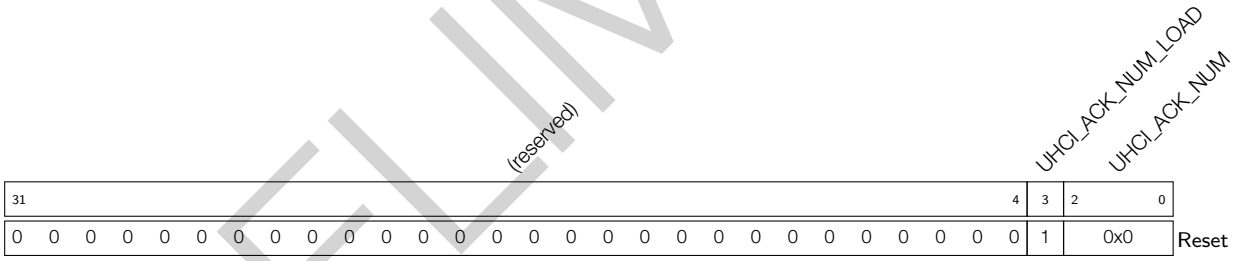
UHCI_TXFIFO_TIMEOUT_ENA TX FIFO 接收数据超时的使能位。(R/W)

UHCI_RXFIFO_TIMEOUT 存储超时值。DMA 读取 RAM 数据超时时产生 UHCI_RX_HUNG_INT 中断。(R/W)

UHCI_RXFIFO_TIMEOUT_SHIFT 用于配置计数最大值。(R/W)

UHCI_RXFIFO_TIMEOUT_ENA DMA 发送数据超时的使能位。(R/W)

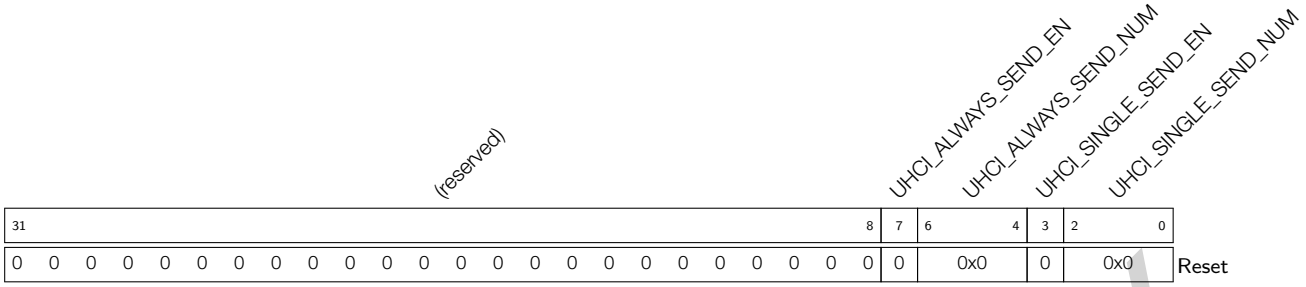
Register 13.38. UHCI_ACK_NUM_REG (0x0028)



UHCI_ACK_NUM 软件流控中使用的 ACK 值。(R/W)

UHCI_ACK_NUM_LOAD 置位此位，加载 UHCI_ACK_NUM 配置的值。(WT)

Register 13.39. UHCI_QUICK_SENT_REG (0x0030)



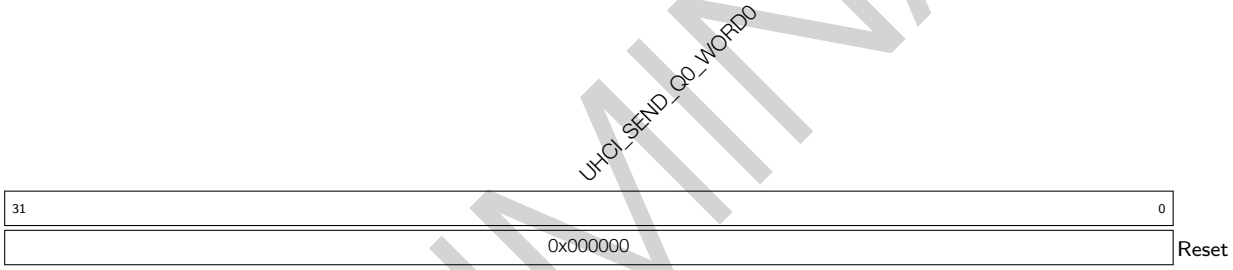
UHCI_SINGLE_SEND_NUM 设定 single_send 寄存器。(R/W)

UHCI_SINGLE_SEND_EN 置位此位使能 single_send 模式发送短包。(R/W/SC)

UHCI_ALWAYS_SEND_NUM 设定 always_send 寄存器。(R/W)

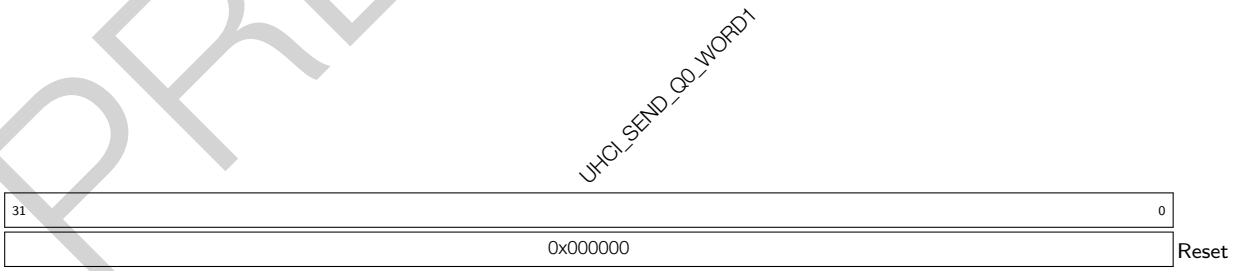
UHCI_ALWAYS_SEND_EN 置位此位使能 always_send 模式发送短包。(R/W)

Register 13.40. UHCI_REG_Q0_WORD0_REG (0x0034)



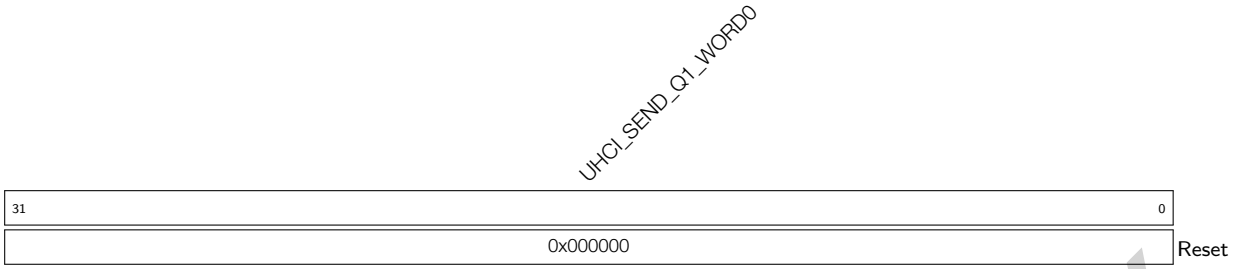
UHCI_SEND_Q0_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.41. UHCI_REG_Q0_WORD1_REG (0x0038)



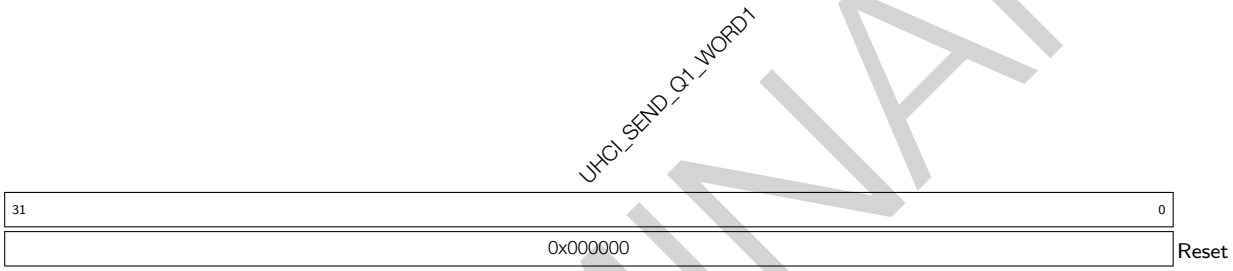
UHCI_SEND_Q0_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.42. UHCI_REG_Q1_WORD0_REG (0x003C)



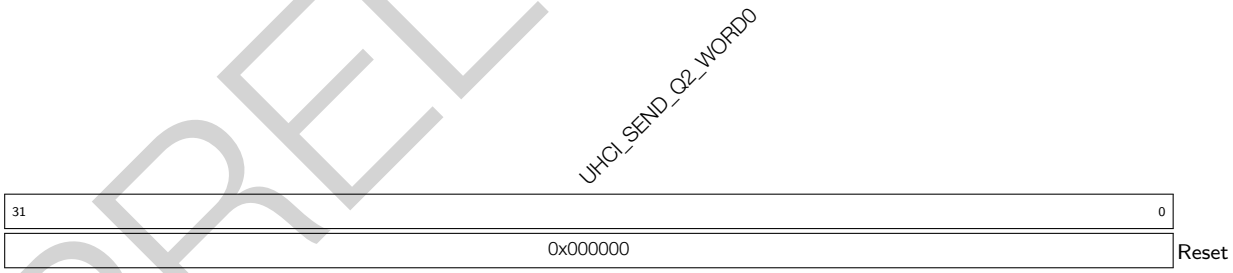
UHCI_SEND_Q1_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.43. UHCI_REG_Q1_WORD1_REG (0x0040)



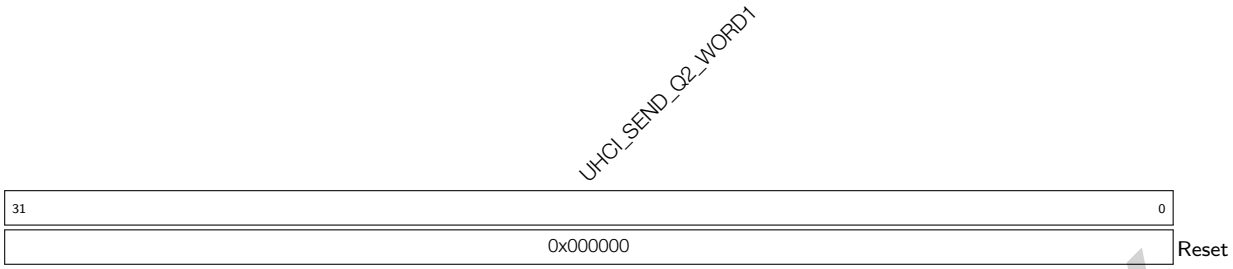
UHCI_SEND_Q1_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.44. UHCI_REG_Q2_WORD0_REG (0x0044)



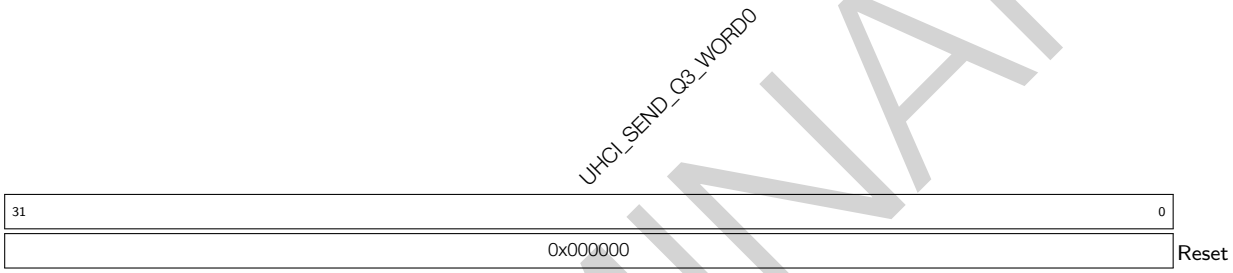
UHCI_SEND_Q2_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.45. UHCI_REG_Q2_WORD1_REG (0x0048)



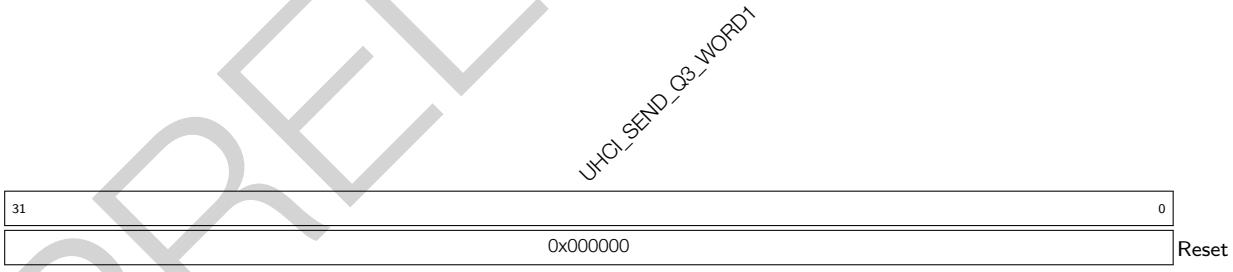
UHCI_SEND_Q2_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.46. UHCI_REG_Q3_WORD0_REG (0x004C)



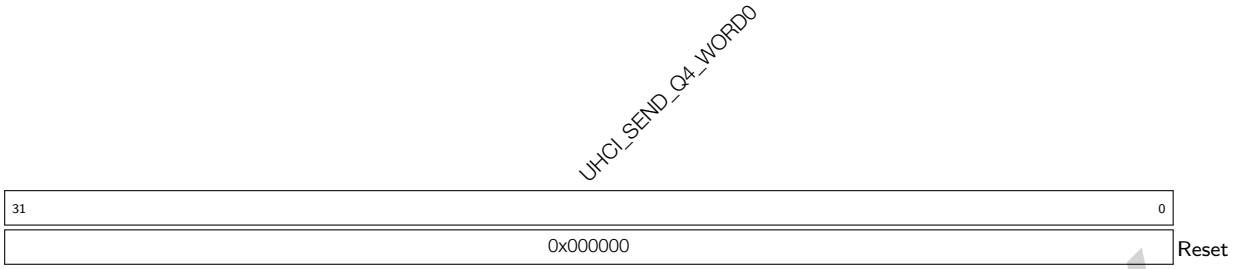
UHCI_SEND_Q3_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.47. UHCI_REG_Q3_WORD1_REG (0x0050)



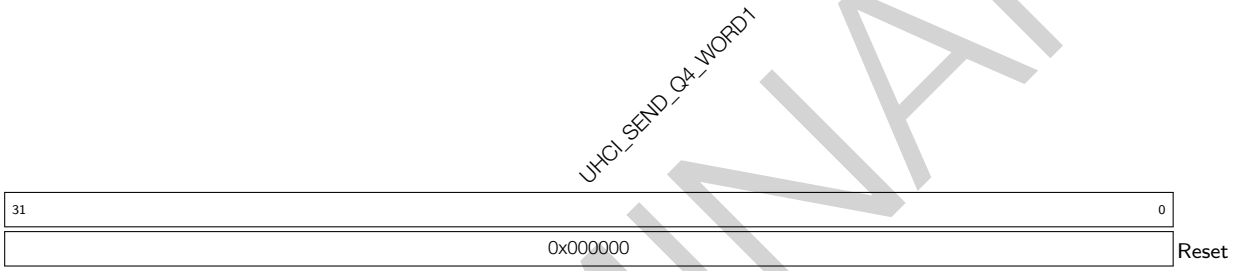
UHCI_SEND_Q3_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.48. UHCI_REG_Q4_WORD0_REG (0x0054)



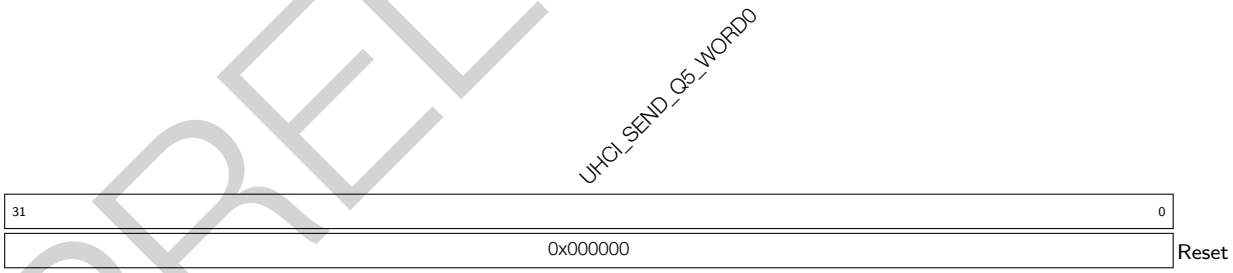
UHCI_SEND_Q4_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.49. UHCI_REG_Q4_WORD1_REG (0x0058)



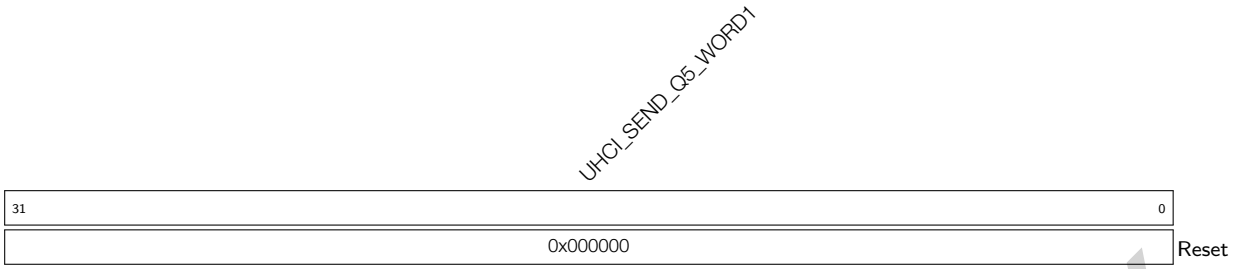
UHCI_SEND_Q4_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.50. UHCI_REG_Q5_WORD0_REG (0x005C)



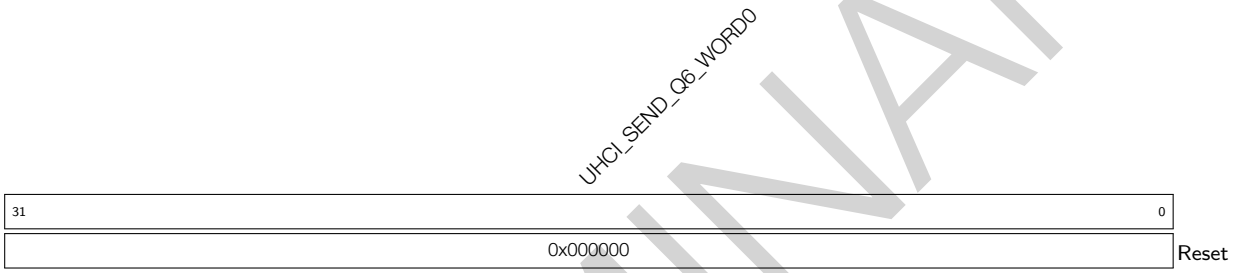
UHCI_SEND_Q5_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.51. UHCI_REG_Q5_WORD1_REG (0x0060)



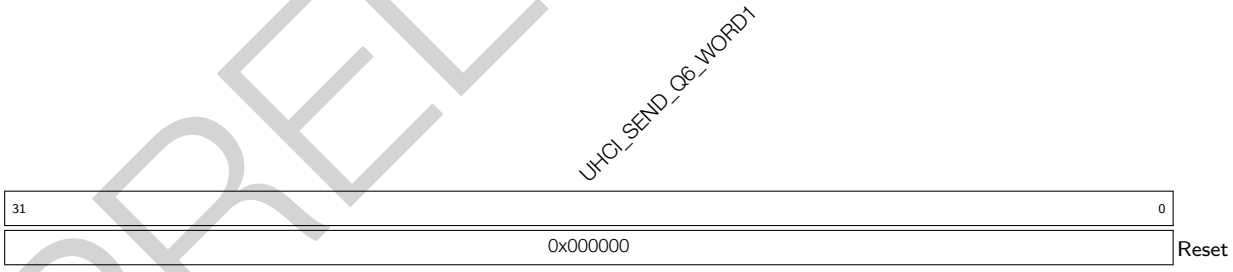
UHCI_SEND_Q5_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.52. UHCI_REG_Q6_WORD0_REG (0x0064)



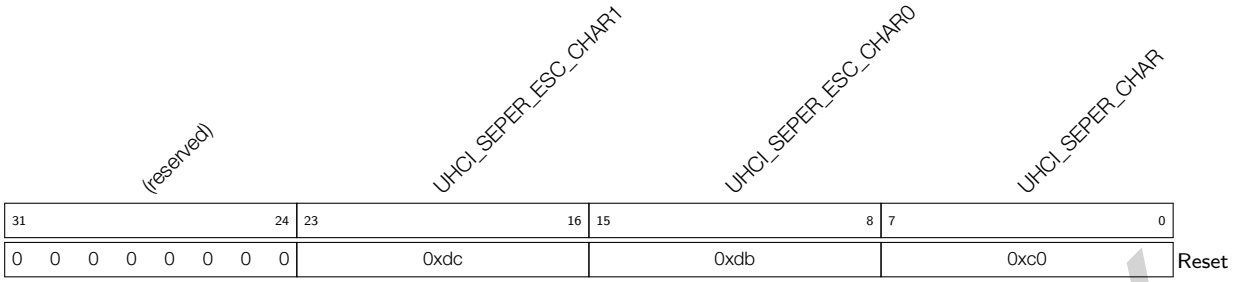
UHCI_SEND_Q6_WORD0 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.53. UHCI_REG_Q6_WORD1_REG (0x0068)



UHCI_SEND_Q6_WORD1 在 UHCI_ALWAYS_SEND_NUM 或 UHCI_SINGLE_SEND_NUM 指定时用作快速发送寄存器。(R/W)

Register 13.54. UHCI_ESC_CONF0_REG (0x006C)

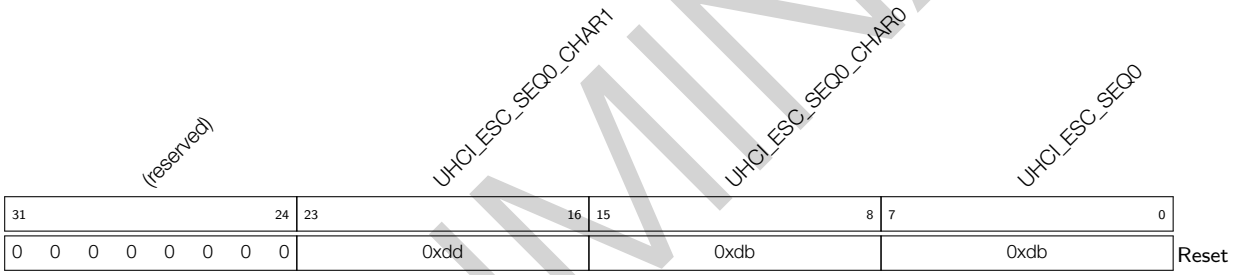


UHCI_SEPER_CHAR 定义需编码的分隔符，默认为 0xc0。(R/W)

UHCI_SEPER_ESC_CHAR0 编码分隔符时定义 Slip 转义序列的第一个字符，分隔符默认为 0xdb。(R/W)

UHCI_SEPER_ESC_CHAR1 编码分隔符时定义 Slip 转义序列的第二个字符，分隔符默认为 0xdc。(R/W)

Register 13.55. UHCI_ESC_CONF1_REG (0x0070)

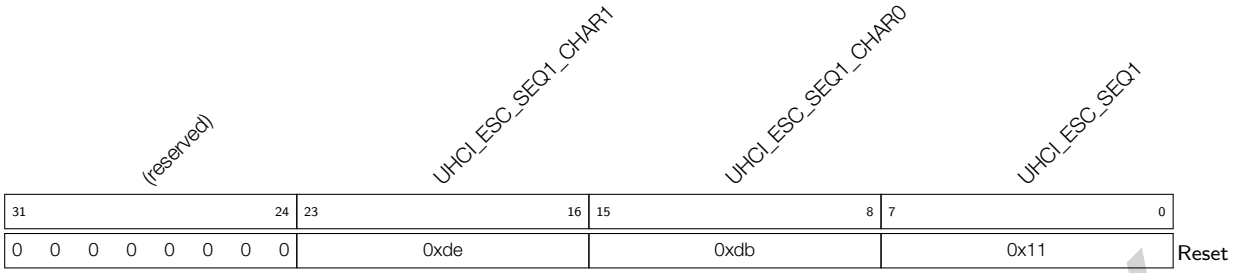


UHCI_ESC_SEQ0 定义需编码的字符，默认为用作 Slip 转义序列第一个字符的 0xdb。(R/W)

UHCI_ESC_SEQ0_CHAR0 编码 UHCI_ESC_SEQ0 时定义 Slip 转义序列的第一个字符，UHCI_ESC_SEQ0 默认为 0xdb。(R/W)

UHCI_ESC_SEQ0_CHAR1 编码 UHCI_ESC_SEQ0 时定义 Slip 转义序列的第二个字符，UHCI_ESC_SEQ0 默认为 0xdd。(R/W)

Register 13.56. UHCI_ESC_CONF2_REG (0x0074)

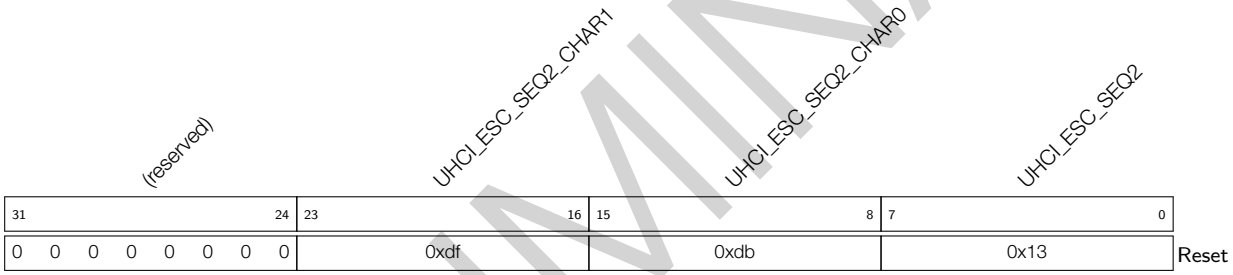


UHCI_ESC_SEQ1 定义需编码的字符，默认为用作流控字符的 0x11。(R/W)

UHCI_ESC_SEQ1_CHAR0 编码 UHCI_ESC_SEQ1 时定义 Slip 转义序列的第一个字符，UHCI_ESC_SEQ1 默认为 0xdb。(R/W)

UHCI_ESC_SEQ1_CHAR1 编码 UHCI_ESC_SEQ1 时定义 Slip 转义序列的第二个字符，UHCI_ESC_SEQ1 默认为 0xde。(R/W)

Register 13.57. UHCI_ESC_CONF3_REG (0x0078)

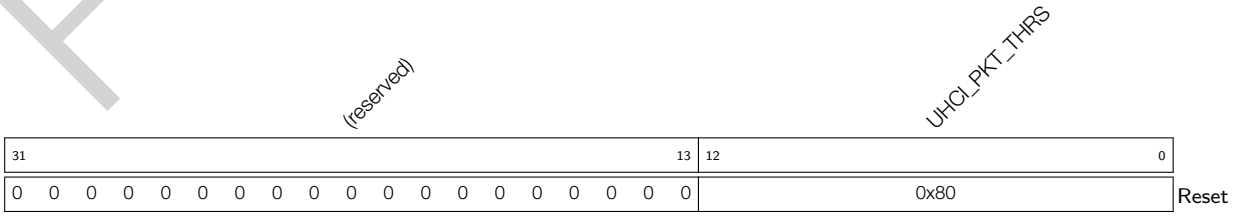


UHCI_ESC_SEQ2 定义需编码的字符，默认为用作流控字符的 0x13。(R/W)

UHCI_ESC_SEQ2_CHAR0 编码 UHCI_ESC_SEQ2 时定义 Slip 转义序列的第一个字符，UHCI_ESC_SEQ2 默认为 0xdb。(R/W)

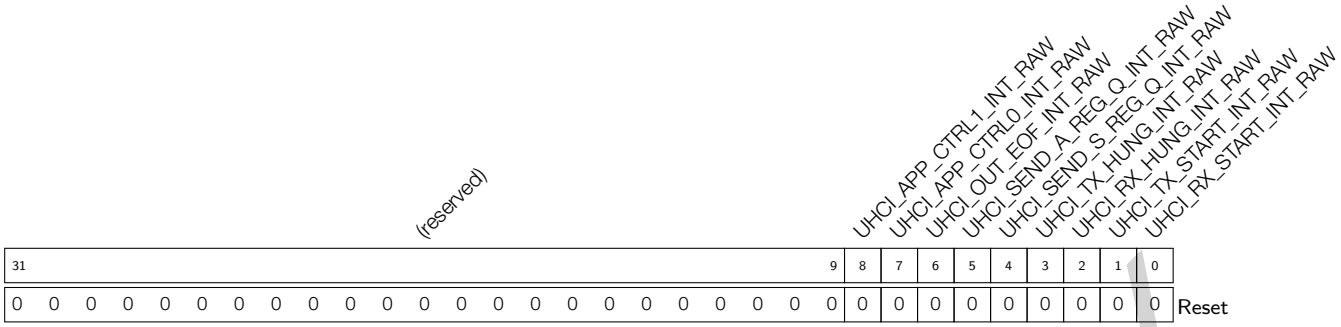
UHCI_ESC_SEQ2_CHAR1 编码 UHCI_ESC_SEQ2 时定义 Slip 转义序列的第二个字符，UHCI_ESC_SEQ2 默认为 0xdf。(R/W)

Register 13.58. UHCI_PKT_THRES_REG (0x007C)



UHCI_PKT_THRS UHCI_HEAD_EN 为 0 时配置包长度的最大值。(R/W)

Register 13.59. UHCI_INT_RAW_REG (0x0004)



UHCI_RX_START_INT_RAW 原始中断位。分隔符成功发送时触发此中断。(R/WTC/SS)

UHCI_TX_START_INT_RAW 原始中断位。DMA 检测到分隔符时触发此中断。(R/WTC/SS)

UHCI_RX_HUNG_INT_RAW 原始中断位。DMA 接收数据所需时间超过配置值时触发此中断。(R/WTC/SS)

UHCI_TX_HUNG_INT_RAW 原始中断位。DMA 读取 RAM 数据所需时间超过配置值时触发此中断。(R/WTC/SS)

UHCI_SEND_S_REG_Q_INT_RAW 原始中断位。UHCI 使用 single_send 寄存器成功发送短包时触发此中断。(R/WTC/SS)

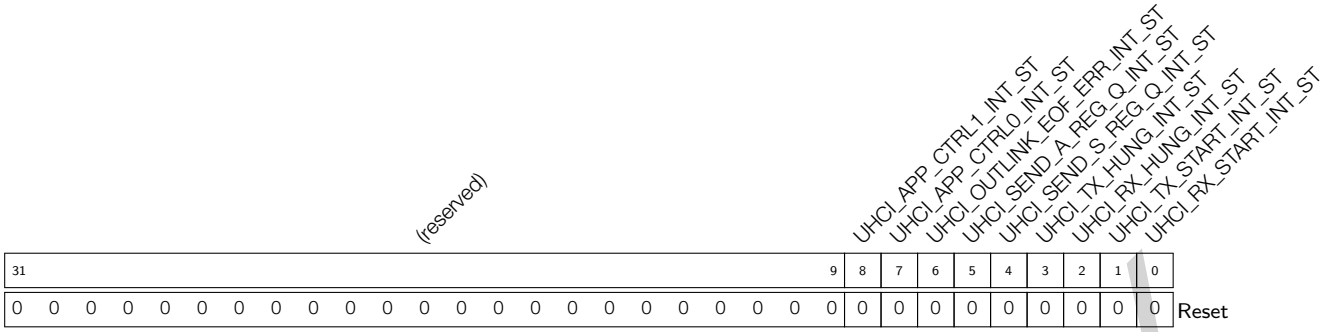
UHCI_SEND_A_REG_Q_INT_RAW 原始中断位。UHCI 使用 always_send 寄存器成功发送短包时触发此中断。(R/WTC/SS)

UHCI_OUT_EOF_INT_RAW 原始中断位。接收数据的 EOF 有错误时触发此中断。(R/WTC/SS)

UHCI_APP_CTRL0_INT_RAW 原始中断位，此位置 1 时触发中断，清零时清除中断。(R/W)

UHCI_APP_CTRL1_INT_RAW 原始中断位，此位置 1 时触发中断，清零时清除中断。(R/W)

Register 13.60. UHCI_INT_ST_REG (0x0008)



UHCI_RX_START_INT_ST UHCI_RX_START_INT_ENA 置 1 时 UHCI_RX_START_INT 中断的屏蔽中断位。(RO)

UHCI_TX_START_INT_ST UHCI_TX_START_INT_ENA 置 1 时 UHCI_TX_START_INT 中断的屏蔽中断位。(RO)

UHCI_RX_HUNG_INT_ST UHCI_RX_HUNG_INT_ENA 置 1 时 UHCI_RX_HUNG_INT 中断的屏蔽中断位。(RO)

UHCI_TX_HUNG_INT_ST UHCI_TX_HUNG_INT_ENA 置 1 时 UHCI_TX_HUNG_INT 中断的屏蔽中断位。(RO)

UHCI_SEND_S_REG_Q_INT_ST UHCI_SEND_S_REG_Q_INT_ENA 置 1 时 UHCI_SEND_S_REG_Q_INT 中断的屏蔽中断位。(RO)

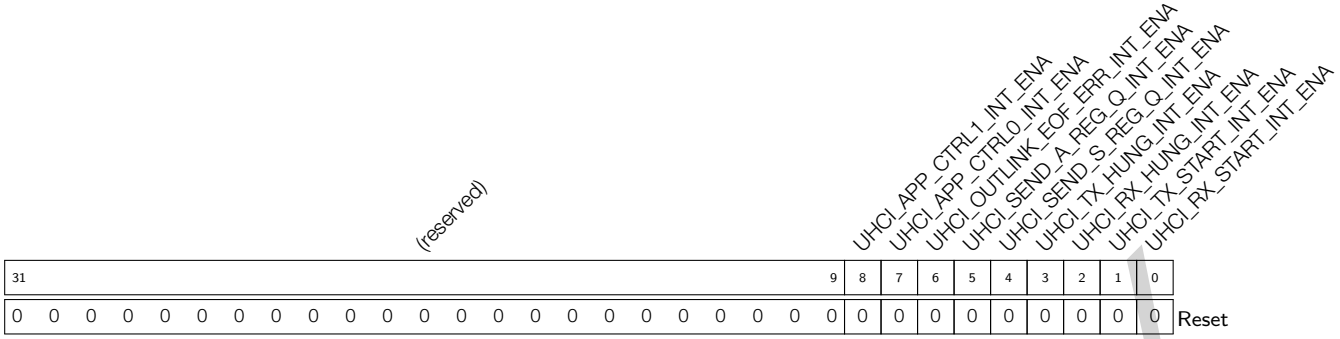
UHCI_SEND_A_REG_Q_INT_ST UHCI_SEND_A_REG_Q_INT_ENA 置 1 时 UHCI_SEND_A_REG_Q_INT 中断的屏蔽中断位。(RO)

UHCI_OUTLINK_EOF_ERR_INT_ST UHCI_OUTLINK_EOF_ERR_INT_ENA 置 1 时 UHCI_OUTLINK_EOF_ERR_INT 中断的屏蔽中断位。(RO)

UHCI_APP_CTRL0_INT_ST UHCI_APP_CTRL0_INT_ENA 置 1 时 UHCI_APP_CTRL0_INT 中断的屏蔽中断位。(RO)

UHCI_APP_CTRL1_INT_ST UHCI_APP_CTRL1_INT_ENA 置 1 时 UHCI_APP_CTRL1_INT 中断的屏蔽中断位。(RO)

Register 13.61. UHCI_INT_ENA_REG (0x000C)



UHCI_RX_START_INT_ENA UHCI_RX_START_INT 的中断使能位。(R/W)

UHCI_TX_START_INT_ENA UHCI_TX_START_INT 的中断使能位。(R/W)

UHCI_RX_HUNG_INT_ENA UHCI_RX_HUNG_INT 的中断使能位。(R/W)

UHCI_TX_HUNG_INT_ENA UHCI_TX_HUNG_INT 的中断使能位。(R/W)

UHCI_SEND_S_REG_Q_INT_ENA UHCI_SEND_S_REG_Q_INT 的中断使能位。(R/W)

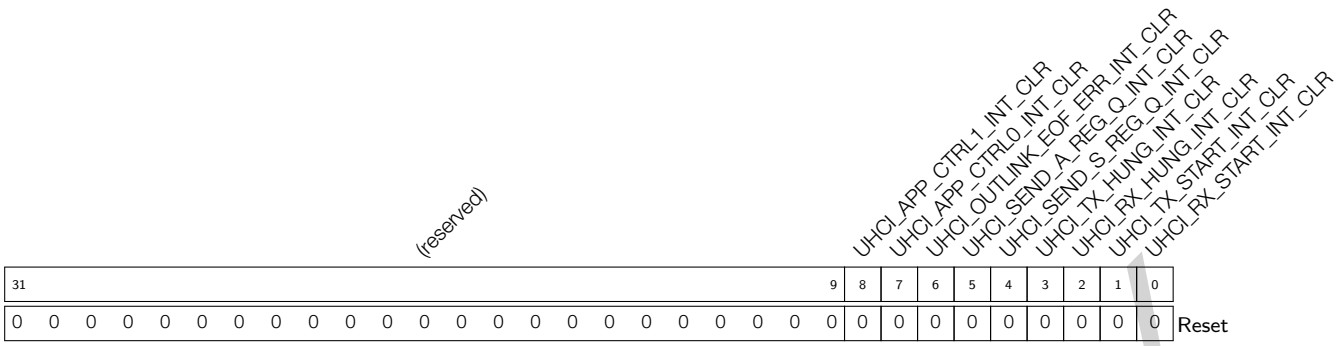
UHCI_SEND_A_REG_Q_INT_ENA UHCI_SEND_A_REG_Q_INT 的中断使能位。(R/W)

UHCI_OUTLINK_EOF_ERR_INT_ENA UHCI_OUTLINK_EOF_ERR_INT 的中断使能位。(R/W)

UHCI_APP_CTRL0_INT_ENA UHCI_APP_CTRL0_INT 的中断使能位。(R/W)

UHCI_APP_CTRL1_INT_ENA UHCI_APP_CTRL1_INT 的中断使能位。(R/W)

Register 13.62. UHCI_INT_CLR_REG (0x0010)



UHCI_RX_START_INT_CLR 置位此位清除 UHCI_RX_START_INT 中断。(WT)

UHCI_TX_START_INT_CLR 置位此位清除 UHCI_TX_START_INT 中断。(WT)

UHCI_RX_HUNG_INT_CLR 置位此位清除 UHCI_RX_HUNG_INT 中断。(WT)

UHCI_TX_HUNG_INT_CLR 置位此位清除 UHCI_TX_HUNG_INT 中断。(WT)

UHCI_SEND_S_REG_Q_INT_CLR 置位此位清除 UHCI_SEND_S_REQ_Q_INT 中断。(WT)

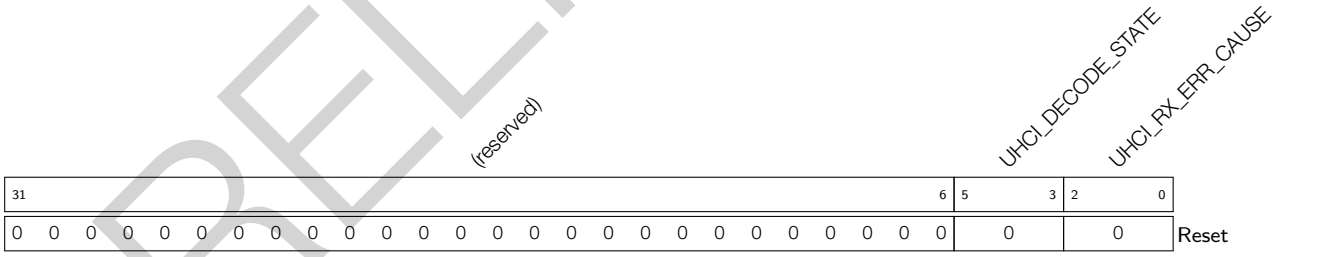
UHCI_SEND_A_REG_Q_INT_CLR 置位此位清除 UHCI_SEND_A_REQ_Q_INT 中断。(WT)

UHCI_OUTLINK_EOF_ERR_INT_CLR 置位此位清除 UHCI_OUTLINK_EOF_ERR_INT 中断。(WT)

UHCI_APP_CTRL0_INT_CLR 置位此位清除 UHCI_APP_CTRL0_INT 中断。(WT)

UHCI_APP_CTRL1_INT_CLR 置位此位清除 UHCI_APP_CTRL1_INT 中断。(WT)

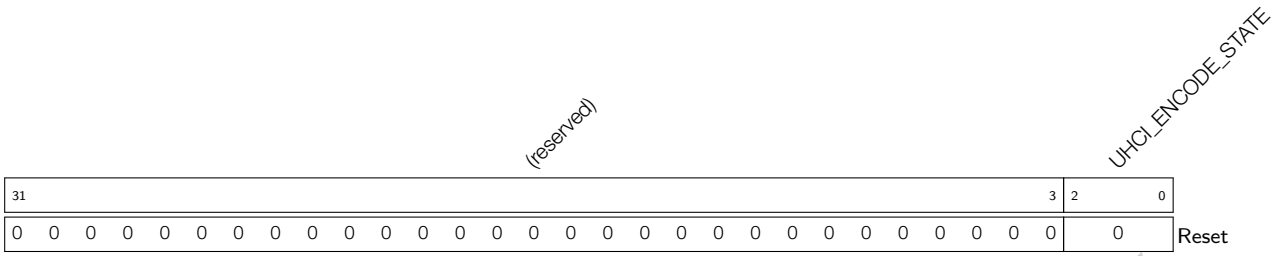
Register 13.63. UHCI_STATE0_REG (0x0018)



UHCI_RX_ERR_CAUSE 在 DMA 接收到错误帧时表示错误类型。3'b001：HCI 包校验和错误；3'b010：HCI 包序列号错误。3'b011：HCI 包 CRC 位错误；3'b100：找到 0xc0 但接收的 HCI 包不完整；3'b101：未找到 0xc0 但接收的 HCI 包完整；3'b110：CRC 检测错误。(RO)

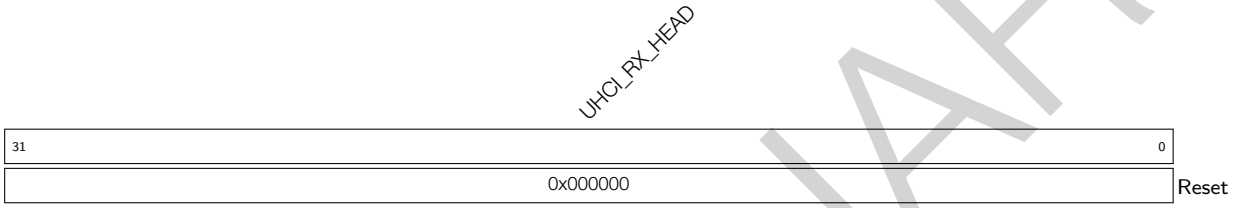
UHCI_DECODE_STATE UHCI 解码器状态。(RO)

Register 13.64. UHCI_STATE1_REG (0x001C)



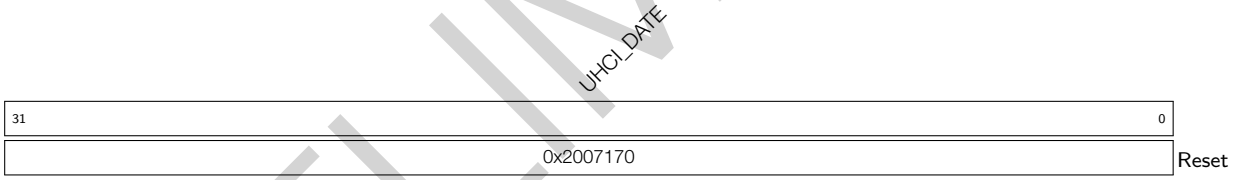
UHCI_ENCODE_STATE UHCI encoder status. (RO)

Register 13.65. UHCI_RX_HEAD_REG (0x002C)



UHCI_RX_HEAD 存储当前接收包的报头 (RO)

Register 13.66. UHCI_DATE_REG (0x0080)



UHCI_DATE 版本控制寄存器。(R/W)

14 双线汽车接口 (TWAI)

双线车载串口 (Two-wire Automotive Interface, TWAI®) 协议是一种多主机、多播的通信协议，具有检测错误、发送错误信号以及内置报文优先仲裁等功能。TWAI 协议适用于汽车和工业应用（可参见第 14.2 章）。

ESP32-C3 包含一个 TWAI 控制器，可通过外部收发器连接到 TWAI 总线。TWAI 控制器包含一系列先进的功能，用途广泛，可用于如汽车产品、工业自动化控制、楼宇自动化等。

14.1 主要特性

ESP32-C3 TWAI 控制器具有以下特性：

- 兼容 ISO 11898-1 协议
- 支持标准格式（11-bit 标识符）和扩展格式（29-bit 标识符）两种帧格式
- 支持 1 Kbit/s ~ 1 Mbit/s 位速率
- 支持多种操作模式
  - 正常模式
  - 只听模式（不影响总线）
  - 自测模式（发送数据时不需应答）
- 64-byte 接收 FIFO
- 特殊发送
  - 单次发送（发生错误时不会自动重新发送）
  - 自发自收（TWAI 控制器同时发送和接收报文）
- 接收滤波器（支持单滤波器和双滤波器模式）
- 错误检测与处理
  - 错误计数
  - 错误报警限制可配置
  - 错误代码捕捉
  - 仲裁丢失捕捉

14.2 功能性协议

14.2.1 TWAI 性能

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

单通道通信与不归零编码： TWAI 总线只有一根传输线进行单通道通信，因此为半双工通信。同步调整也在单通道中进行，因此不需其他通道（如时钟通道和使能通道）。TWAI 上报文的位流采用不归零编码 (NRZ) 方式。

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

位填充：TWAI 报文的某些域已经过位填充。发送器在发送连续五个位的相同值（如显性数值或隐性数值）后，需自动插入一个互补位。同理，接收到 5 个连续位的接收器应将下一个位视为填充位。位填充应用于以下域：SOF、仲裁域、控制域、数据域和 CRC 序列（可参见第 14.2.2 章）。

多播：当各节点连接到同个总线上时，这些节点都将接收到相同的位。各节点上的数据将保持一致，除非发生总线错误（可参见第 14.2.3 章）。

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

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

错误检测：各节点将积极检测总线上的错误，并通过向 TWAI 总线发送错误帧来广播检测到的错误。

故障限制：各节点都维护有错误计数器，该错误计数器依据 TWAI 协议规则增加或减少。当错误计数超过一定阈值时，对应节点将自动关闭并退出网络。

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

发送器与接收器：不论何时，任意 TWAI 节点都可作为发送器和接收器。

- 产生报文的节点为发送器。且该节点将一直作为发送器，直到总线空闲或该节点失去仲裁。请注意，仲裁期间有多个节点作为发送器。
- 所有不属于发送器的节点都将成为接收器。

14.2.2 TWAI 报文

TWAI 节点使用报文发送数据，并在监测到总线上存在错误时向其他节点发送错误信号。报文有多的帧类型，不同的帧类型具有不同的帧格式。

TWAI 协议有以下帧类型：

- 数据帧
- 远程帧
- 错误帧
- 过载帧
- 帧间距

TWAI 协议有以下帧格式：

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

14.2.2.1 数据帧和远程帧

节点使用数据帧向其他节点发送数据，可负载 0 ~ 8 字节数据。节点使用远程帧向其他节点请求具有相同标识符的数据帧，因此远程帧中不包含任何数据字节。但是，数据帧和远程帧中包含许多相同域。下图 14-1 所示为不

同帧类型和不同帧格式中包含的域和子域。

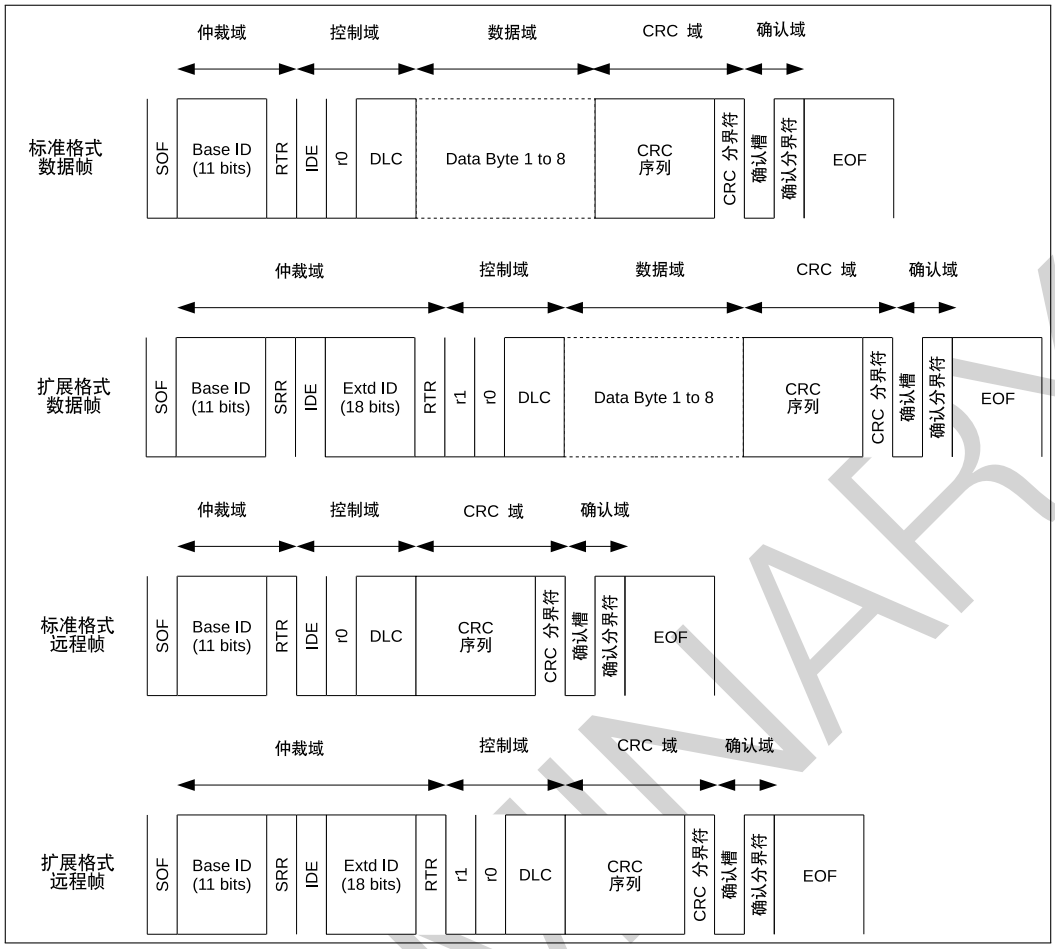


图 14-1. 数据帧和远程帧中的位域

仲裁域

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

仲裁域主要由获得最高优先发送权的帧标识符的有效位组成。根据显性位代表的逻辑值为 0，隐性位代表的逻辑值为 1，有以下规律:

- ID 值最小的帧将总是获得仲裁（逻辑 0 是显性位值）。
- 如果 ID 数值相同，由于数据帧的 RTR 位为显性位，数据帧将优先于远程帧。
- 如果 ID 的前 11 位相同，由于扩展帧的 SRR 位是隐性，因而标准格式帧将总优先于扩展格式帧。

控制域

控制域主要由数据长度代码 (DLC) 组成，DLC 表示一个数据帧中的负载的数据字节长度，或一个远程帧请求的数据字节长度。DLC 优先发送长度数值的最高有效位。

数据域

数据域中包含一个数据帧真实负载的数据字节。远程帧中不包含数据域。

CRC 域

CRC 域主要由 CRC 序列组成。CRC 序列是一个 15-bit 的循环冗余校验编码，根据数据帧或远程帧中位填充前的内容（从 SOF 到数据域末尾的所有内容）计算而来。

确认域

确认 (ACK) 域由确认槽和确认分界符组成，主要功能为：接收器向发送器报告已正确接收到有效报文。

表 14-1. 不同帧类型、帧格式下的域及子域信息

数据/远程帧	描述
SOF	帧起始 (SOF) 是一个用于同步总线上节点的单个显性位。
Base ID	基标识符 (ID.28 ~ ID.18) 是 SFF 的 11-bit 标识符，或者是 EFF 中 29-bit 标识符的前 11-bit。
RTR	远程发送请求位 (RTR) 显示当前报文是数据帧（显性）还是远程帧（隐性）。这意味着，当某个数据帧和一个远程帧有相同标识符时，数据帧始终优先于远程帧仲裁。
SRR	在 EFF 中发送替代远程请求位 (SRR)，以替代 SFF 中相同位置的 RTR 位。
IDE	标识符扩展位 (IED) 显示当前报文是 SFF （显性）还是 EFF（隐性）。这意味着，当某 SFF 帧和 EFF 帧有相同基标识符时，SFF 帧将始终优先于 EFF 帧仲裁。
Extd ID	扩展标识符 (ID.17 ~ ID.0) 是 EFF 中 29-bit 标识符的剩余 18-bit。
r1	r1（保留位 1）始终是显性位。
r0	r0（保留位 0）始终是显性位。
DLC	数据长度代码 (DLC) 为 4-bits，且为 0 ~ 8 中任一数值。数据帧使用 DLC 表示自身包含的数据字节长度。远程帧使用 DLC 表示从其他节点请求的数据字节长度。
数据字节	表示数据帧的数据负载量。该字节长度应与 DLC 的值匹配。首先发送数据字节的最高有效位（MSB）。
CRC 序列	CRC 序列是一个 15-bit 的循环冗余校验编码。
CRC 分界符	CRC 分界符是紧随 CRC 序列的 1-bit 隐性位。
确认槽	确认槽用于接收器节点表示是否已成功接收数据帧或远程帧。发送器节点将在确认槽中发送一个隐性位，如果接收到的帧没有错误，则接收器节点将发送 1-bit 显性位替代隐性位。
确认分界符	确认分界符是紧随确认槽的 1-bit 隐性位。
EOF	帧结束 (EOF) 标志着数据帧或远程帧的结束，由七个隐性位组成。

14.2.2.2 错误帧和过载帧

错误帧

当某节点检测到总线错误时，将发送一个错误帧。错误帧由一个特殊的错误标志构成，该标志由某相同值的六个连续位组成，因而违反了位填充的规则。所以，当某节点检测到总线错误并发送错误帧时，其余节点也将相应地检测到一个填充错误并各自发送错误帧。也就是说，当发生总线错误时，通过上述过程可将该报文传递至总线上的所有节点。

当某节点检测到总线错误时，该节点将于下一个位发送错误帧。特例：如果总线错误类型为 CRC 错误，那么错误

帧将从确认分界符的下一个位开始（可参见第 14.2.3 章）。下图 14-2 所示为一个错误帧所包含的不同域：

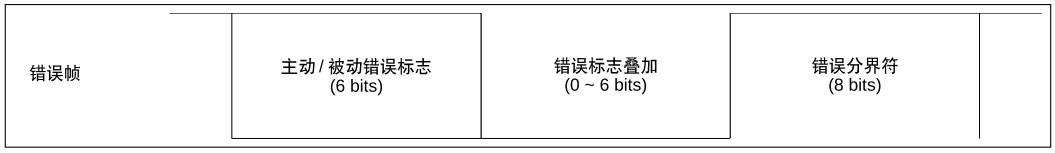


图 14-2. 错误帧中的位域

表 14-2. 错误帧中的位域信息

错误帧	描述
错误标志	错误标志包括两种形式: 主动错误标志和被动错误标志，主动错误标志由 6 个显性位组成，被动错误标志由 6 个隐性位组成（被其他节点的显性位优先仲裁时除外）。处于主动错误状态的节点发送主动错误标志，处于被动错误状态的节点发送被动错误标志。
错误标志叠加	错误标志叠加域的主要目的是允许总线上的其他节点发送各自的主动错误标志。叠加域的范围是 0 ~ 6 位，结束标志是检测到第一个隐性位（如检测到分界符上的第一个位时）。
错误分界符	分界符域标志着错误/过载帧结束，由 8 个隐性位构成。

过载帧

过载帧与包含主动错误标志的错误帧有着相同的位信息。二者区别在于触发发送过载帧的条件。下图 14-3 所示为过载帧中包含的位域：

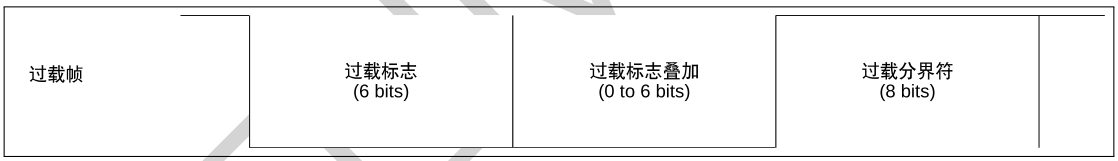


图 14-3. 过载帧中的位域

表 14-3. 过载帧中的位域信息

过载帧	描述
过载标志	由 6 个显性位构成。与主动错误标志相同。
过载标志叠加	允许其他节点发送过载标志的叠加，与错误标志叠加相似。
过载分界符	由 8 个隐性位构成。与错误分界符相同。

下列情况将触发发送过载帧：

1. 接收器内部要求延迟发送下一个数据帧或远程帧。
2. 在间歇域中的首个和第二个位上检测到显性位。
3. 如果在错误分界符的第八个（最后一个）位上检测到显性位。请注意，在这种情况下 TEC 和 REC 的值将不会增加（可参见第 14.2.3 章）。

由于上述情况发送过载帧时，须满足以下规定：

- 第 1 条情况下发送的过载帧只能从间歇域后的第一个位开始。
- 第 2、3 条情况下发送的过载帧须从检测到显性位的后一个位开始。
- 针对第 1 条情况，最多可生成两个过载帧。

14.2.2.3 帧间距

帧间距充当各帧之间的分隔符。数据帧和远程帧必须与前一帧用一个帧间距分隔开，不论前面的帧是何类型（数据帧、远程帧、错误帧、过载帧）。但是，错误帧和过载帧则无需与前一个帧分隔开。

下图 14-4 所示为帧间距中包含的域：

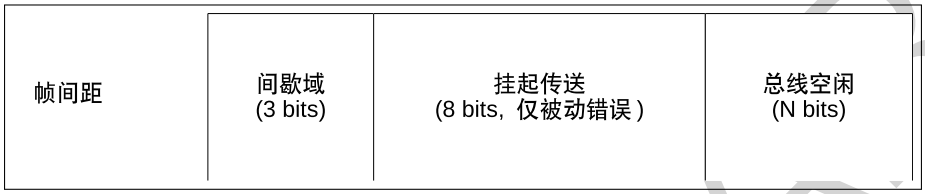


图 14-4. 帧间距中的域

表 14-4. 帧间距中的域信息

帧间距	描述
间歇域	间歇域由 3 个隐性位构成。
挂起传送	被动错误节点发送报文后，必须在帧间距中包含一个挂起传送域，由 8 个隐性位构成。主动错误节点中不含这个域。
总线空闲	总线空闲域长度任意。发送 SOF 时，总线空闲结束。若节点中有挂起传送，则 SOF 应在间歇域后的第一位发送。

14.2.3 TWAI 错误

14.2.3.1 错误类型

TWAI 中的总线错误包括以下类型：

位错误

当节点发送一个位值（显性位或隐性位）但检测到相反的位时（如，发送显性位时检测到了隐性位），就会发生位错误。但是，如果发送的位是隐性位，且位于仲裁域或确认槽或被动错误标志中，那么此时检测到显性位的话不会认定为位错误。

填充错误

当检测到相同值的 6 个连续位时（违反位填充的编码规则），发生填充错误。

CRC 错误

接收器将根据接收到的数据帧和远程帧的有效位（无填充位流）计算 CRC 值。当接收器计算的值与接收到的数据帧和远程帧中的 CRC 序列不匹配时，会发生 CRC 错误。

格式错误

当某个报文中的固定格式位中包含非法位时，可检测到格式错误。比如，r1 和 r0 域必须固定为显性。

确认错误
当发送器无法在确认槽中检测到显性位时，将发生确认错误。

14.2.3.2 错误状态

每个节点 TWAI 控制器通过维护两个错误计数器来实现故障界定，计数数值决定错误状态。这两个错误计数器分别为：发送错误计数 (TEC) 和接收错误计数 (REC)。TWAI 包含以下错误状态。

主动错误
处于主动错误状态的节点可参与到总线交互中，且在检测到错误时可以发送主动错误标志。

被动错误
处于被动错误状态的节点可参与到总线交互中，但在检测到错误时只能发送一次被动错误标志。被动错误节点发送数据帧或远程帧后，需在后续的帧间距中增加挂起传送域。

离线
禁止处于离线状态的节点以任意方式干扰总线（如，不允许其进行数据传输）。

14.2.3.3 错误计数

TEC 和 REC 根据以下规则递增/递减。**请注意，一条报文传输中可应用多个规则**。

1. 当接收器检测到错误时，REC 数值将增加 1。当检测到的错误为发送主动错误标志或过载标志期间的位错误除外。
2. 发送错误标志后，当接收器第一个检测到的位是显性位时，REC 数值将增加 8。
3. 当发送器发送错误标志时，TEC 数值增加 8。但是，以下情况不适用于该规则：
   - 发送器为被动错误状态，因为在应答槽未检测到显性位而产生应答错误，且在发送被动错误标志时检测到显性位时，则 TEC 数值不应增加。
   - 发送器在仲裁期间因填充错误而发送错误标志，且填充位本该是隐性位但是检测到显性位，则 TEC 数值不应增加。
4. 若发送器在发送主动错误标志和过载标志时检测到位错误，则 TEC 数值增加 8。
5. 若接收器在发送主动错误标志和过载标志时检测到位错误，则 REC 数值增加 8。
6. 任意节点在发送主动/被动错误标志或过载标志后，节点仅能承载最多 7 个连续显性位。在（发送主动错误标志或过载标志时）检测到第 14 个连续显性位，或在被动错误标志后检测到第 8 个连续显性位后，发送器将使其 TEC 数值增加 8，而接收器将使其 REC 数值增加 8。每增加 8 个连续显性位的同时，（发送器的）TEC 和（接收器的）REC 数值也将增加 8。
7. 每当发送器成功发送报文后（接收到 ACK，且直到 EOF 完成未发生错误），TEC 数值将减小 1，除非 TEC 的数值已经为 0。
8. 当接收器成功接收报文后（确认槽前未检测到错误，且成功发送 ACK），则 REC 数值将相应减小。
   - 若 REC 数值位于 1 ~ 127 之间，则其值减小 1。
   - 若 REC 数值大于 127，则其值减小到 127。
   - 若 REC 数值为 0，则仍保持为 0。

9. 当一个节点的 TEC 和/或 REC 数值大于等于 128 时，该节点变为被动错误节点。导致节点发生上述状态切换的错误，该节点仍发送主动错误标志。请注意，一旦 REC 数值到达 128，后续任何增加该值的动作都是无效的，直到 REC 数值返回到 128 以下。
10. 当某节点的 TEC 数值大于等于 256 时，该节点将进入离线状态。
11. 当某被动错误节点的 TEC 和 REC 数值都小于等于 127，则该节点将变为主动错误节点。
12. 当离线节点在总线上检测到 128 次 11 个连续隐性位后，该节点可变为主动错误节点（TEC 和 REC 数值都重设为 0）。

14.2.4 TWAI 位时序

14.2.4.1 标称位

TWAI 协议允许 TWAI 总线以特定的位速率运行。但是，总线内的所有节点必须以统一位速率运行。

- **标称位速率**为每秒发送比特数量。
- **标称位时间**为 **1/标称位速率**。

每个标称位时间中含多个段，每段由多个时间定额 (Time Quanta) 组成。**时间定额**为最小时间单位，作为一种预分频时钟信号应用于各个节点中。下图 14-5 所示为一个标称位时间内所包含的段。

TWAI 控制器将在以一个时间定额的步长进行操作，每个时间定额中都会分析 TWAI 的总线状态。如果两个连续的时间定额中总线状态不同（隐性-显性，或反之），意味着有边沿产生。PBS1 和 PBS2 的交点将被视为采样点，采样的总线状态即为这个位的数值。

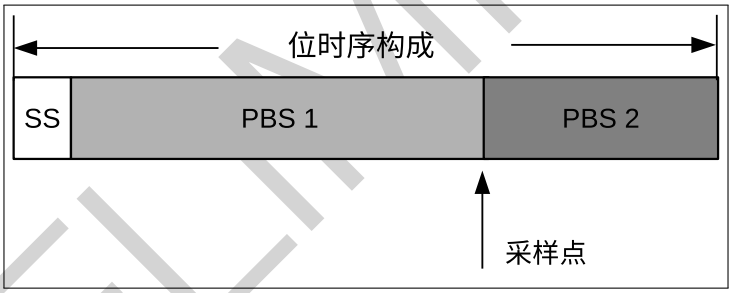


图 14-5. 标称位构成

表 14-5. 名义位时序中包含的段

段	描述
同步段 (SS)	SS（同步段）的长度为 1 个时间定额。若所有节点都同步，则位边沿发生时应位于该段内。
缓冲时期段 1 (PBS1)	PBS1 的长度可为 1 ~ 16 个时间定额，用于补偿网络中的物理延迟时间。可通过增加 PBS1 的长度来实现同步。
缓冲时期段 2 (PBS2)	PBS2 的长度可为 1 ~ 8 个时间定额，用于补偿节点中的信息处理时间。可通过缩短 PBS2 的长度来实现同步。

14.2.4.2 硬同步与再同步

由于时钟偏移和抖动，同一总线上节点的位时序可能会脱离相位段。因而，位边沿可能会偏移到同步段的前后。针对上述位边沿偏移的问题 TWAI 提供多种同步方式。设发生相位错误时位边沿偏移的 TQ（时间定额）数量为

“e”，该值与 SS 相关。

- 主动相位错误 (e > 0)：位边沿位于同步段之后采样点之前（即，边沿向后偏移）。
- 被动相位错误 (e < 0)：位边沿位于前个位的采样点之后同步段之前（即，边沿向前偏移）。

为解决相位错误，有两种同步方式，**硬同步**与**再同步**。**硬同步**与**再同步**遵守以下规则：

- 单个位时序中仅可执行一次同步。
- 同步仅可发生在隐性位到显性位的边沿上。

硬同步

总线空闲期间，硬同步发生在隐性位到显性位的变化边沿上（如总线空闲后的第一个 SOF 位）。此时，所有节点都将重启其内部位时序，从而使该变化边沿位于重启位时序的同步段内。

再同步

非总线空闲期间，再同步发生在隐性位到显性位的变化边沿上。如果边沿上有主动相位错误 (e > 0)，则增加 PBS1 段的长度。如果边沿上有被动相位错误 (e < 0)，则减小 PBS2 段的长度。

PBS1/PBS2 具体增加和减小的时间定额取决于相位错误的绝对值，同时也受可配置的同步跳宽 (SJW) 数值限制。

- 当相位错误的绝对值小于等于 SJW 数值时，PBS1/PBS2 将增加/减小 **e** 个时间定额。该过程与硬同步具有相同效果。
- 当相位错误的绝对值大于 SJW 数值时，PBS1/PBS2 将增加/减小与 SJW 相同数值的时间定额。这意味着，在完全解决相位错误之前，可能需要执行多次位的再同步。

14.3 结构概述

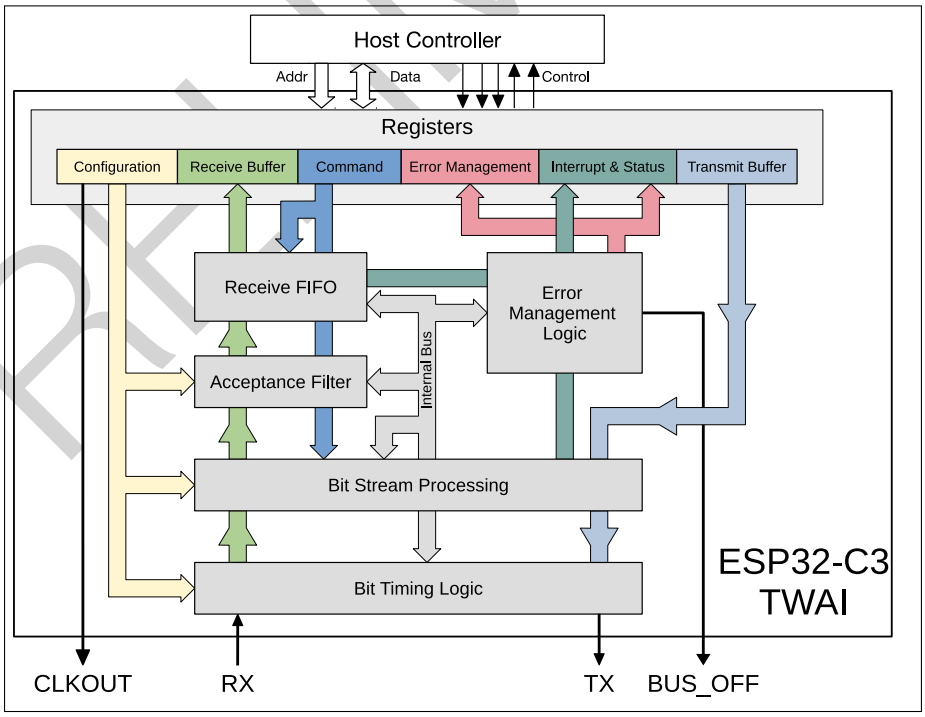


图 14-6. TWAI 概略图

TWAI 控制器的主要功能模块如图 14-6。

14.3.1 寄存器模块

ESP32-C3 的 CPU 使用 32-bit 字对齐地址访问外设。但对于 TWAI 控制器中的大部分寄存器，仅最低有效字节 (bits [7:0]) 数据有效。在这些寄存器中，bits [31:8] 数值在写入时被忽略，在读取时返回 0。

配置寄存器

配置寄存器存储 TWAI 控制器的各配置项，如位速率、操作模式、接收滤波器等。只有在 TWAI 控制器处于复位模式时，才可修改配置寄存器（可参见第 14.4.1 章）。

指令寄存器

CPU 通过指令寄存器驱动 TWAI 控制器执行任务，如发送报文或清除接收缓冲器。只有在 TWAI 控制器处于操作模式时，才可修改指令寄存器（可参见第 14.4.1 章）。

中断 & 状态寄存器

中断寄存器显示 TWAI 控制器中发生的事件（每个事件由一个单独的位表示）。状态寄存器显示 TWAI 控制器的当前状态。

错误管理寄存器

错误管理寄存器包括错误计数和捕捉寄存器。错误计数寄存器表示 TEC 和 REC 的数值。捕捉寄存器负责记录相关信息，如 TWAI 控制器在何处检测到总线错误，或何时丢失仲裁。

发送缓冲寄存器

发送缓冲器大小为 13 字节，用于存储 TWAI 的待发送报文。

接收缓冲寄存器

接收缓冲器大小为 13 字节，用于存储单个报文。接收缓冲器是读取接收 FIFO 的窗口，接收 FIFO 中的第一个报文将被映射到接收缓冲器中。

请注意，发送缓冲寄存器、接收缓冲寄存器和接收滤波配置寄存器的地址范围重叠，地址偏移涉及 0x0040 ~ 0x0070。上述地址范围内寄存器遵循以下规则：

- 当 TWAI 控制器处于复位模式时，该地址范围被映射到接收滤波配置寄存器。
- TWAI 控制器处于操作模式时：
  - 对地址范围的所有读取都映射到接收缓冲寄存器中。
  - 对地址范围的所有写入都映射到发送缓冲寄存器中。

14.3.2 位流处理器

位流处理 (BSP) 模块负责对发送缓冲器的数据进行帧处理 (如，位填充和附加 CRC 域) 并为位时序逻辑 (BTL) 模块生成位流。同时，BSP 模块还负责处理从 BTL 模块中接收的位流（如，去填充和验证 CRC），并将处理后报文写入接收 FIFO。BSP 还负责检测 TWAI 总线上的错误并将此类错误报告给错误管理逻辑 (EML)。

14.3.3 错误管理逻辑

错误管理逻辑 (EML) 模块负责更新 TEC 和 REC 数值，记录错误信息（如，错误类型和错误位置），更新控制器的错误状态，确保 BSP 模块发送正确的错误标志。此外，该模块还负责记录 TWAI 控制器丢失仲裁时的 bit 位置。

14.3.4 位时序逻辑

位时序逻辑 (BTL) 模块负责以预先配置的位速率发送和接受报文。BTL 模块还负责同步位时序，确保数据传输的稳定性。位速率由多个可编程的段组成，且用户可设置每个段的 TQ（时间定额）长度，来调整传播延迟、控

制器处理时间等因素。

14.3.5 接收滤波器

接收滤波器是一个可编程的报文过滤单元，允许 TWAI 控制器根据报文的标识符域接收或拒绝该报文。通过接收滤波器的报文才能被存储到接收 FIFO 中。用户可配置接收滤波器的模式：单滤波器、双滤波器。

14.3.6 接收 FIFO

接收 FIFO 是大小为 64-byte 的缓冲器（位于 TWAI 控制器内部），负责存储通过接收滤波器的接收报文。接收 FIFO 中存储的报文大小可以不同（范围在 3 ~ 13 byte 之间）。当接收 FIFO 为满时（或剩余的空间不足以完全存储下一个接收报文），将触发溢出中断，后续的接收报文将丢失，直到接收 FIFO 中清除出足够的存储空间。接收 FIFO 中的第一条报文将被映射到 13-byte 的接收缓冲器中，直到该报文被清除（通过释放接收缓冲器指令）。清除后，接收 FIFO 将释放上一条已清除报文的空间，同时将窗口映射到接收 FIFO 中的下一条报文。

14.4 功能描述

14.4.1 模式

ESP32-C3 TWAI 控制器有两种工作模式：复位模式和操作模式。将 TWAI_RESET_MODE 位置 1，进入复位模式；置 0，进入操作模式。

14.4.1.1 复位模式

要修改 TWAI 控制器的各种配置寄存器，需进入复位模式。进入复位模式时，TWAI 控制器彻底与 TWAI 总线断开连接。复位模式下，TWAI 控制器将无法发送任何报文（包括错误信号）。任何正在进行的报文传输将立即被终止。同样的，TWAI 控制器在该模式下也将无法接收任何报文。

14.4.1.2 操作模式

进入操作模式后，TWAI 控制器与总线相连，并且写保护各配置寄存器，以确保控制器的配置在运行期间保持一致。操作模式下，TWAI 控制器可以发送和接收报文（包括错误信号），但具体取决于 TWAI 控制器配置于哪种运行子模式。TWAI 控制器支持以下三种子模式：

- **正常模式：** TWAI 控制器可以发送和接收包含错误信号在内的报文（如，错误帧和过载帧）。
- **自测模式：** 与正常模式相同，但在该模式下，TWAI 控制器发送报文时，即使在 CRC 域之后没有接收到应答信号，也不会产生应答错误。通常在单个 TWAI 控制器自测时使用该模式。
- **只听模式：** TWAI 控制器可以接收报文，但在 TWAI 总线上保持完全被动。因此，TWAI 控制器将无法发送任何报文、应答或错误信号。错误计数将保持冻结状态。该模式用于 TWAI 总线监控。

请注意，退出复位模式后（如，进入操作模式时），TWAI 控制器需等待 11 个连续隐性位出现，才能完全连接上 TWAI 总线（即，可以发送或接收报文）。

14.4.2 位时序

TWAI 控制器的工作位速率必须在控制器处于复位模式时进行配置。在寄存器 TWAI_BUS_TIMING_0_REG 和 TWAI_BUS_TIMING_1_REG 中配置位速率，这两个寄存器包含以下域：

下表 14-6 所示为 TWAI_BUS_TIMING_0_REG 包含的位域。

表 14-6. TWAI_BUS_TIMING_0_REG 的 bit 信息 (0x18)

Bit 31-16	Bit 15	Bit 14	Bit 13	Bit 12	.....	Bit 1	Bit 0
保留	SJW.1	SJW.0	保留	BRP.12	.....	BRP.1	BRP.0

说明：

- 预分频值 (BRP)：TWAI 时间定额时钟由 APB 时钟分频得到，APB 时钟通常为 80 MHz。可通过以下公式计算分频数值，其中 t_{Tq} 为时间定额的时钟周期，t_{CLK} 为 APB 时钟周期：

  $t_{Tq} = 2 \times t_{CLK} \times (2^{12} \times BRP.12 + 2^{11} \times BRP.11 + ... + 2^{1} \times BRP.1 + 2^{0} \times BRP.0 + 1)$

- 同步跳宽 (SJW)：SJW 数值在 SJW.0 和 SJW.1 中配置，计算公式为：SJW = (2 x SJW.1 + SJW.0 + 1)。

下表 14-7 所示为 TWAI_BUS_TIMING_1_REG 包含的位域。

表 14-7. TWAI_BUS_TIMING_1_REG 的 bit (0x1c)

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	SAM	PBS2.2	PBS2.1	PBS2.0	PBS1.3	PBS1.2	PBS1.1	PBS1.0

说明：

- PBS1: 根据以下公式计算缓冲时期段 1 中的时间定额数量：(8 x PBS1.3 + 4 x PBS1.2 + 2 x PBS1.1 + PBS1.0 + 1)。
- PBS2: 根据以下公式计算缓冲时期段 2 中的时间定额数量：(4 x PBS2.2 + 2 x PBS2.1 + PBS2.0 + 1)。
- SAM: 该值置 1 启动三点采样。用于低/中速总线，有利于过滤总线上的尖峰信号。

14.4.3 中断管理

ESP32-C3 TWAI 控制器提供了八种中断，每种中断由寄存器 TWAI_INT_RAW_REG 中的一个位表示。要触发某个特定的中断，须设置 TWAI_INT ENA_REG 中相应的使能位。

TWAI 控制器提供了以下八种中断：

- 接收中断
- 发送中断
- 错误报警中断
- 数据溢出中断
- 被动错误中断
- 仲裁丢失中断
- 总线错误中断
- 总线状态中断

只要在 TWAI_INT_RAW_REG 一个或多个中断位为 1，TWAI 控制器中的中断信号即为有效，当 TWAI_INT_RAW_REG 中的所有位都被清除时，TWAI 控制器中的中断信号则失效。读操作访问寄存器 TWAI_INT_RAW_REG 后，其中

的大多数中断位将自动清除。但是，只有通过 TWAI_RELEASE_BUF 指令位清除所有接收报文后，接收中断位才能被清除。

14.4.3.1 接收中断 (RXI)

当 TWAI 接收 FIFO 中有待读取报文时(TWAI_RX_MESSAGE_CNT_REG > 0),都会触发 RXI。TWAI_RX_MESSAGE_CNT_REG 中记录的报文数量包括接收 FIFO 中的有效报文和溢出报文。直到通过 TWAI_RELEASE_BUF 指令位清除所有挂起接收报文后，RXI 才会失效。

14.4.3.2 发送中断 (TXI)

当发送缓冲器由锁定状态变为为空闲状态，将会触发 TXI。此时软件可以将其他报文加载到发送缓冲器中等待发送。以下几种情况都会触发该中断：

- 报文发送已成功完成（如，应答未发现错误）。任何发送失败将自动重发。
- 单次发送已完成（TWAI_TX_COMPLETE 位指示发送成功与否）。
- 使用 TWAI_ABORT_TX 指令位终止报文发送。

14.4.3.3 错误报警中断 (EWI)

每当寄存器 TWAI_STATUS_REG 中 TWAI_ERR_ST 和 TWAI_BUS_OFF_ST 的位值改变时（如，从 0 变为 1 或反之），都会触发 EWI。根据 EWI 触发时 TWAI_ERR_ST 和 TWAI_BUS_OFF_ST 的值分成以下几种情况：

- 如果 TWAI_ERR_ST = 0 且 TWAI_BUS_OFF_ST = 0：
  - 如果 TWAI 控制器处于主动错误状态，则表示 TEC 和 REC 的值都返回到了 TWAI_ERR_WARNING_LIMIT_REG 所设的阈值之下。
  - 如果 TWAI 控制器此前正处于总线恢复状态，则表示此时总线恢复已成功完成。
- 如果 TWAI_ERR_ST = 1 且 TWAI_BUS_OFF_ST = 0：表示 TEC 或 REC 数值已超过 TWAI_ERR_WARNING_LIMIT_REG 所设的阈值。
- 如果 TWAI_ERR_ST = 1 且 TWAI_BUS_OFF_ST = 1：表示 TWAI 控制器已进入 BUS_OFF 状态（因 TEC >= 256）。
- 如果 TWAI_ERR_ST = 0 且 TWAI_BUS_OFF_ST = 1：表示 BUS_OFF 恢复期间，TWAI 控制器的 TEC 数值已低于 TWAI_ERR_WARNING_LIMIT_REG 所设的阈值。

14.4.3.4 数据溢出中断 (DOI)

每当接收 FIFO 中有溢出发生时，都会触发 DOI。DOI 表示接收 FIFO 已满且应立即进行读取，以防出现更多溢出报文。

只有导致接收 FIFO 溢出的第一条报文可触发 DOI（如，当接收 FIFO 从未满变为开始溢出时）。任意后续的溢出报文将不会再次重复触发 DOI。只有当所有接收的（有效报文或溢出）报文都被读取后，才能再次触发 DOI。

14.4.3.5 被动错误中断 (TXI)

每当 TWAI 控制器从主动错误变为被动错误，或反之时，都会触发 EPI。

14.4.3.6 仲裁丢失中断 (ALI)

每当 TWAI 控制器尝试发送报文且丢失仲裁时，都会触发 ALI。TWAI 控制器丢失仲裁的 bit 位置将自动记录在仲裁丢失捕捉寄存器 (TWAI_ARB LOST CAP_REG) 中。仲裁丢失捕捉寄存器被清除（通过 CPU 读取该寄存器）之前，将不会再记录新发生的仲裁失败时的 bit 位置。

14.4.3.7 总线错误中断 (BEI)

每当 TWAI 控制器在 TWAI 总线上检测到错误时，都会触发 BEI。发生总线错误时，总线错误的类型和发生错误时的 bit 位置都将自动记录在错误捕捉寄存器 (TWAI_ERR_CODE_CAP_REG) 中。错误捕捉寄存器被清除（通过 CPU 的读取）之前，将不会再记录新的总线错误信息。

14.4.3.8 总线状态中断 (BSI)

每当 TWAI 控制器在收发总线数据状态与空闲状态之间切换时，都会触发 BSI。当该中断发生时，可通过读取 TWAI_STATUS_REG 寄存器 TWAI_RX_ST 和 TWAI_TX_ST 两个域来判断 TWAI 控制器当前的状态。

14.4.4 发送缓冲器与接收缓冲器

14.4.4.1 缓冲器概述

表 14-8. SFF 与 EFF 的缓冲器布局

标准格式 (SFF)		扩展格式 (EFF)	
TWAI 地址	内容	TWAI 地址	内容
0x40	TX/RX 帧信息	0x40	TX/RX 帧信息
0x44	TX/RX identifier 1	0x44	TX/RX identifier 1
0x48	TX/RX identifier 2	0x48	TX/RX identifier 2
0x4c	TX/RX data byte 1	0x4c	TX/RX identifier 3
0x50	TX/RX data byte 2	0x50	TX/RX identifier 4
0x54	TX/RX data byte 3	0x54	TX/RX data byte 1
0x58	TX/RX data byte 4	0x58	TX/RX data byte 2
0x5c	TX/RX data byte 5	0x5c	TX/RX data byte 3
0x60	TX/RX data byte 6	0x60	TX/RX data byte 4
0x64	TX/RX data byte 7	0x64	TX/RX data byte 5
0x68	TX/RX data byte 8	0x68	TX/RX data byte 6
0x6c	保留	0x6c	TX/RX data byte 7
0x70	保留	0x70	TX/RX data byte 8

表 14-8 所示为发送缓冲器和接收缓冲器的寄存器布局。发送和接收缓冲寄存器的访问地址范围相同，且只有当 TWAI 控制器处于操作模式时才可访问。CPU 的写入操作将访问发送缓冲寄存器，CPU 的读取操作将访问接收缓冲寄存器。发送缓冲器和接受缓冲器中存储报文（接收报文或待发送报文）的寄存器布局和域完全一致。

发送缓冲寄存器用于配置 TWAI 的待发送报文。CPU 可以在发送缓冲寄存器进行写入操作，指定报文的帧类型、帧格式、帧 ID 和帧数据（有效载荷）。一旦发送缓冲器配置完成后，CPU 可以将 TWAI_CMD_REG 中的 TWAI_TX_REQ 位置 1，以开始报文发送。

- 若是自发自收请求，变更为将 TWAI_SELF_RX_REQ 置 1。

- 若是单次发送，需要同时将 TWAI_TX_REQ 和 TWAI_ABORT_TX 置 1。

接收缓冲寄存器将映射接收 FIFO 中的第一条报文。CPU 可以对接收缓冲寄存器执行读取操作，获取第一条报文的帧类型、帧格式、帧 ID 和帧数据（有效载荷）。读取完接收缓冲寄存器中的报文后，CPU 通过将 TWAI_CMD_REG 中的 TWAI_RELEASE_BUF 位置 1 来清除接收缓冲寄存器，若接收 FIFO 中仍有待处理的报文，按照接收报文的先后次序依次将接收到的报文映射到接收缓冲寄存器。

14.4.4.2 帧信息

帧信息的长度为 1-byte，主要用于明确报文的帧类型、帧格式以及数据长度。下表 14-9 所示为帧信息域。

表 14-9. TX/RX 帧信息 (SFF/EFF)；TWAI 地址 0x40

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	FF	RTR	X	X	DLC.3	DLC.2	DLC.1	DLC.0

说明：

1. FF: 主要明确某报文属于 EFF 还是 SFF。当 FF 位为 1 时，该报文为 EFF，当 FF 位为 0 时，该报文为 SFF。
2. RTR: 主要明确某报文是数据帧还是远程帧。当 RTR 位为 1 时，该报文为远程帧，当 RTR 位为 0 时，该报文为数据帧。
3. X: 无关 bit，可以是任意值。
4. DLC: 主要明确数据帧中的数据字节数量，或从远程帧中请求的数据字节数量。TWAI 数据帧的最大载荷为 8 个数据字节，因此 DLC 的数值范围应是 0 ~ 8。

14.4.4.3 帧标识符

若报文为 SFF，则对应的帧标识符域为 2-bytes (11-bits)；若报文为 EFF，则对应的帧标识符域为 4-bytes (29-bits)。

下表 Table 14-10 ~ 14-11 所示为 SFF (11-bits) 报文的帧标识符域。

表 14-10. TX/RX 标识符 1 (SFF); TWAI 地址 0x44

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3

表 14-11. TX/RX 标识符 2 (SFF); TWAI 地址 0x48

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.2	ID.1	ID.0	X[1]	X[2]	X[2]	X[2]	X[2]

说明：

1. 无关项。建议设置为与接收缓冲器兼容（设为 RTR），以防需使用自接收功能（或与自接收功能一起使用）。
2. 无关项。建议设置为与接收缓冲器兼容（设为 0），以防需使用自接收功能（或与自接收功能一起使用）。

下表 14-12 ~ 14-15 所示为 EFF (29-bits) 报文的帧标识符域。

表 14-12. TX/RX 标识符 1 (EFF); TWAI 地址 0x44

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.28	ID.27	ID.26	ID.25	ID.24	ID.23	ID.22	ID.21

表 14-13. TX/RX 标识符 2 (EFF); TWAI 地址 0x48

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.20	ID.19	ID.18	ID.17	ID.16	ID.15	ID.14	ID.13

表 14-14. TX/RX 标识符 3 (EFF); TWAI 地址 0x4c

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.12	ID.11	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5

表 14-15. TX/RX 标识符 4 (EFF); TWAI 地址 0x50

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ID.4	ID.3	ID.2	ID.1	ID.0	X[1]	X[2]	X[2]

说明：

1. 无关项。建议设置为与接收缓冲器兼容（设为 RTR），以防需使用自接收功能（或与自接收功能一起使用）。
2. 无关项。建议设置为与接收缓冲器兼容（设为 0），以防需使用自接收功能（或与自接收功能一起使用）。

14.4.4.4 帧数据

帧数据域包含发送或接收的数据帧，范围为 0 ~ 8 bytes。其中的有效字节数应与 DLC 相同。但是，如果 DLC 数值大于 8，则帧数据域的有效字节数仍为 8。远程帧中不包含数据载荷，因此不存在帧数据域。

比如，当发送 5 个字节的数据帧时，CPU 应在 DLC 域中写入数值 5，并将数据写入数据域 1 ~ 5 字节对应的寄存器。同样，当接收 DLC 为 5 的数据帧时，只有 1 ~ 5 数据字节中包含 CPU 可以读取的有效载荷数据。

14.4.5 接收 FIFO 和数据溢出

接收 FIFO 是一个 64-byte 的内部缓冲器，用于以先进先出的原则存储接收到的报文。一条接收报文可在接收 FIFO 中占 3 ~ 13 bytes 空间，且其中字节序与接收缓冲器的寄存器地址顺序相同。接收缓冲寄存器将被映射到接收 FIFO 中第一条报文。

当 TWAI 控制器接收到一条报文时，TWAI_RX_MESSAGE_COUNTER 的值将增加 1，最大值为 64。如果接收 FIFO 中有足够的剩余空间，报文内容将被写入到接收 FIFO 中。读取接收缓冲器中的消息后，通过将 TWAI_RELEASE_BUF 的位置 1，释放接收 FIFO 第一条报文所占的空间，TWAI_RX_MESSAGE_COUNTER 的值也将减小 1。然后，接收缓冲器将映射接收 FIFO 中的下一条报文。

当 TWAI 控制器接收到一条报文，但接收 FIFO 没有足够空间完整地存储这条接收报文时（不论是因为报文内容大小大于接收 FIFO 中的空闲空间，还是因为接收 FIFO 已满），便会发生数据溢出。

数据溢出发生时：

- 接收 FIFO 中剩余的空间将填满溢出报文的内容。如果接收 FIFO 已满，则无法存储溢出报文的任何内容。
- 接收 FIFO 首次发生数据溢出时，将触发数据溢出中断。
- 溢出报文仍将增加 TWAI_RX_MESSAGE_COUNTER 的值到最大值 64。
- 接收 FIFO 将在内部将溢出报文标记为无效。可使用 TWAI_MISS_ST 位，确认目前接收缓冲器映射的报文是有效报文还是溢出报文。

为了清除接收 FIFO 中的溢出报文，应重复调用 TWAI_RELEASE_BUF，直到 TWAI_RX_MESSAGE_COUNTER 为 0。这样可以读取接收 FIFO 中的所有有效报文，并清除所有溢出报文。

14.4.6 接收滤波器

接收滤波器允许 TWAI 控制器根据报文 ID 过滤接收报文（有时可以过滤报文的第一个数据字节和帧类型）。只有通过过滤的报文才能存储到接收 FIFO 中。接收滤波器的使用可以一定程度地减轻 TWAI 控制器的运行负荷（如，可减少使用接收 FIFO 和发生接收中断的次数），因为 TWAI 控制器将只需要操作过滤后的一小部分报文。

接收滤波的配置寄存器和发送/接收缓冲寄存器使用相同的访问地址空间，只有当 TWAI 控制器处于复位模式时，该地址段才被映射到接收滤波器的配置寄存器。

接收滤波器的配置寄存器由 32-bit 的 Code 值和 32-bit 的 Mask 值组成。Code 值将指定一种位排列模式，每条过滤报文中的位都必须匹配该模式，才能使该报文通过过滤。Mask 值可屏蔽 Code 值中的某些位（屏蔽位将被设置为“不相关”位）。如图 14-7 所示，为了使报文通过过滤，每条过滤报文的 ID 位都必须匹配 Code 值所设模式或者被 Mask 值屏蔽。

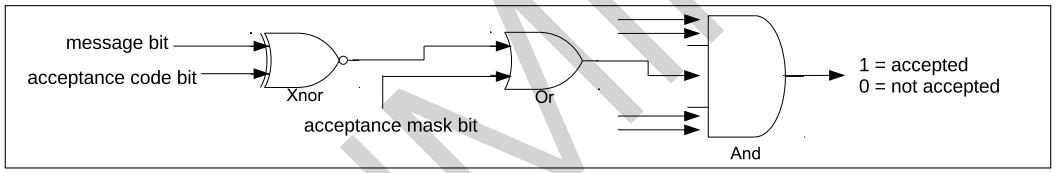


图 14-7. 接收滤波器

TWAI 控制器的接收滤波器允许 32-bit 的 Code 值和 Mask 值定义单个滤波器（单滤波模式），或两个滤波器（双滤波模式）。接收滤波器如何解析 32-bit 的 code 值和 mask 值，取决于滤波模式以及接收报文的格式（如，SFF 还是 EFF）。

14.4.6.1 单滤波模式

将 TWAI_RX_FILTER_MODE 的位置 1，可启动单滤波模式。此后，32-bit code/mask 的值将定义单个滤波器。

单个滤波器可过滤数据帧和远程帧中的以下位：

- SFF
  - 11-bit ID 整体
  - RTR bit
  - 数据字节 1 和数据字节 2
- EFF
  - 29-bit ID 整体
  - RTR bit

下图 14-8 所示为单滤波模式下如何解析 32-bit code/mask 的值。

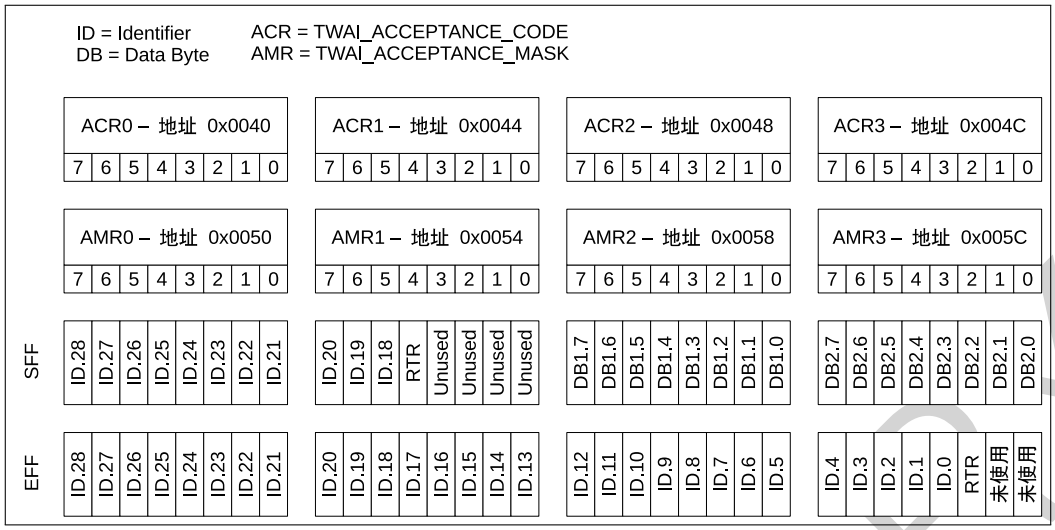


图 14-8. 单滤波模式

14.4.6.2 双滤波模式

将 TWAI_RX_FILTER_MODE 的位置 0，可启动双滤波模式。此后，32-bit code/mask 的值将定义两个滤波器之一，即滤波器 1 或滤波器 2。双滤波模式下，如果报文通过这两个滤波器中的至少一个，则表示该报文已成功通过过滤。

这两个滤波器可以过滤数据帧和远程帧中的以下位：

- SFF
  - 11-bit ID 整体
  - RTR bit
  - 数据字节 1 (仅适用于滤波器 1)
- EFF
  - 29-bit ID 的前 16-bit

下图 14-9 所示为双滤波模式下如何解析 32-bit code/mask 的值。

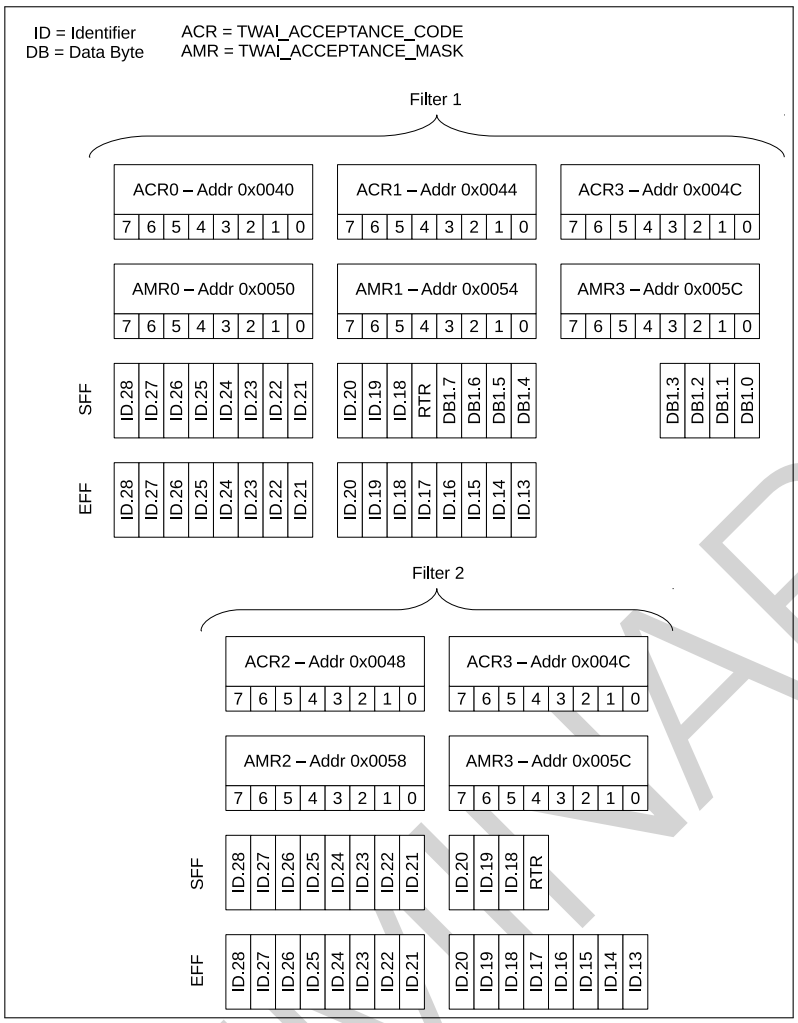


图 14-9. 双滤波模式

14.4.7 错误管理

TWAI 协议要求每个 TWAI 节点中都包含发送错误计数器 (TEC) 和接收错误计数器 (REC)。这两个错误计数的数值决定了 TWAI 控制器当前的错误状态（如，主动错误、被动错误、离线）。TWAI 控制器将 TEC 和 REC 的数值分别存储在 TWAI_TX_ERR_CNT_REG 和 TWAI_RX_ERR_CNT_REG 中，CPU 可随时进行读取。除了错误状态之外，TWAI 控制器还提供错误报警限制 (EWL) 的功能，这个功能可在 TWAI 控制器进入被动错误状态之前，提醒用户当前发生的严重总线错误。

TWAI 控制器的当前错误状态通过以下各数值和状态位体现，即：TEC、REC、TWAI_ERR_ST 和 TWAI_BUS_OFF_ST。这些数值和状态位的变化也将触发中断，从而提醒用户当前的错误状态变化（可参见第 14.4.3 章）。下图 14-10 所示为错误状态、上述数值和状态位以及错误状态相关中断之间的关系。

14.4.7.1 错误报警限制

错误报警限制 (EWL) 为 TEC 和 REC 的可配置阈值，若错误计数数值超过该阈值，将触发 EWI 中断。EWL 将作为一个报警功能提示当前发生的严重 TWAI 总线错误，且在 TWAI 控制器进入被动错误状态之前被触发。EWL 数值应在寄存器 TWAI_ERR_WARNING_LIMIT_REG 中进行配置，配置同时 TWAI 控制器必须处于复位模式下。TWAI_ERR_WARNING_LIMIT_REG 默认数值为 96。

当 TEC 和/或 REC 数值大于等于 EWL 数值时，TWAI_ERR_ST 位将立即被置 1。同理，当 TEC 和 REC 数值都

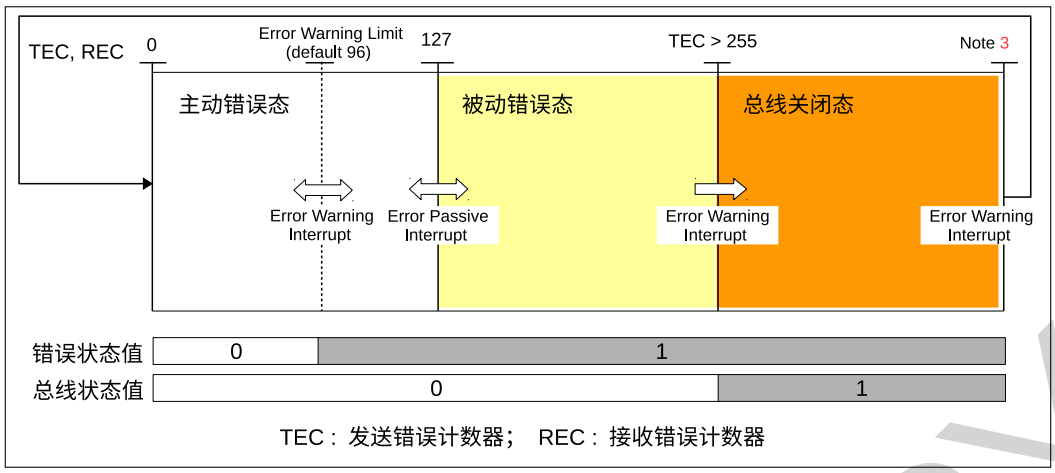


图 14-10. 错误状态变化

小于 EWL 数值时，TWAI_ERR_ST 位将立即复位为 0。只要 TWAI_ERR_ST（或 TWAI_BUS_OFF_ST）位值发生变化，便会触发错误报警中断。

14.4.7.2 被动错误

当 TEC 或 REC 数值大于 127 时，TWAI 控制器处于被动错误状态。同理，当 TEC 和 REC 数值都小于等于 127 时，TWAI 控制器进入主动错误状态。每当 TWAI 控制器从主动错误状态变为被动错误状态，或反之时，都将触发被动错误中断。

14.4.7.3 离线状态与离线恢复

当 TEC 数值大于 255 时，TWAI 控制器将进入离线状态。进入离线状态后，TWAI 控制器将自动进行以下动作：

- REC 数值置为 0
- TEC 数值置为 127
- TWAI_BUS_OFF_ST 位置 1
- 进入复位模式

每当 TWAI_BUS_OFF_ST 位（或 TWAI_ERR_ST 位）数值发生变化时，都将触发错误报警中断。

为了返回主动错误状态，TWAI 控制器必须进行离线恢复。要启动离线恢复，首先需要退出复位模式，进入操作模式。然后要求 TWAI 控制器在总线上检测到 128 次 11 个连续隐性位。

每一次 TWAI 控制器检测到 11 个连续隐性位时，TEC 数值都将减小，以追踪离线恢复进程。当离线恢复完成后（TEC 数值从 127 减小到 0），TWAI_BUS_OFF_ST 位将自动复位为 0，从而触发错误报警中断。

14.4.8 错误捕捉

错误捕捉 (ECC) 功能允许 TWAI 控制器以错误代码的形式记录 TWAI 总线错误的错误类型和 bit 位置。当检测到一个 TWAI 总线错误时，总线错误中断将被触发，相应的错误代码将记录在 TWAI_ERR_CODE_CAP_REG 中。寄存器 TWAI_ERR_CODE_CAP_REG 中存储的当前错误代码被读取之前，后续的总线错误中断触发时，将不会再记录错误代码。

下表 14-16 所示为寄存器 TWAI_ERR_CODE_CAP_REG 中的域：

表 14-16. TWAI_ERR_CODE_CAP_REG 中的位信息 (0x30)

Bit 31-8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	ERRC.1	ERRC.0	DIR	SEG.4	SEG.3	SEG.2	SEG.1	SEG.0

说明：

- 错误代码 (ERRC)：表示总线错误的类型。00 代表位错误，01 代表格式错误，10 代表填充错误，11 代表其他错误类型。
- 传输方向 (DIR)：表示总线错误发生时，TWAI 控制器处于发送器状态还是接收器状态。0 代表发送器，1 代表接收器。
- 错误段 (SEG)：表示总线错误发生在 TWAI 报文的哪个段。

下表 14-17 所示为 SEG.0 ~ SEG.4 的位信息。

表 14-17. SEG.4 - SEG.0 的位信息

Bit SEG.4	Bit SEG.3	Bit SEG.2	Bit SEG.1	Bit SEG.0	描述
0	0	0	1	1	帧起始
0	0	0	1	0	ID.28 ~ ID.21
0	0	1	1	0	ID.20 ~ ID.18
0	0	1	0	0	bit SRTR
0	0	1	0	1	bit IDE
0	0	1	1	1	ID.17 ~ ID.13
0	1	1	1	1	ID.12 ~ ID.5
0	1	1	1	0	ID.4 ~ ID.0
0	1	1	0	0	bit RTR
0	1	1	0	1	保留位 1
0	1	0	0	1	保留位 0
0	1	0	1	1	数据长度代码
0	1	0	1	0	数据域
0	1	0	0	0	CRC 序列
1	1	0	0	0	CRC 分界符
1	1	0	0	1	确认槽
1	1	0	1	1	确认分界符
1	1	0	1	0	帧结束
1	0	0	1	0	间歇域
1	0	0	0	1	主动错误标志
1	0	1	1	0	被动错误标志
1	0	0	1	1	兼容显性位
1	0	1	1	1	错误分界符
1	1	1	0	0	过载标志

说明：

- Bit SRTR: 标准格式 RTR bit。

- Bit IDE: 标识符扩展位。0 表示标准格式。

14.4.9 仲裁丢失捕捉

仲裁丢失捕捉 (ALC) 功能允许 TWAI 控制器记录丢失仲裁的 bit 位置。当 TWAI 控制器丢失仲裁时，bit 位置将被记录在寄存器 TWAI_ARB LOST CAP_REG 中，同时触发仲裁丢失中断。

后续的仲裁丢失中断触发时，bit 位置将不会被记录在 TWAI_ARB LOST CAP_REG 中，直到 TWAI_ERR_CODE_CAP_REG 中的当前仲裁丢失捕捉被读取。

下表 14-18 所示为 TWAI_ERR_CODE_CAP_REG 中的位域；下图 14-11 所示为一条 TWAI 报文的 bit 位置。

表 14-18. TWAI_ARB LOST CAP_REG 中的位信息 (0x2c)

Bit 31-5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
保留	BITNO.4	BITNO.3	BITNO.2	BITNO.1	BITNO.0

说明：

- 位号 (BITNO)：表示丢失仲裁的 TWAI 报文的第 n 个位。

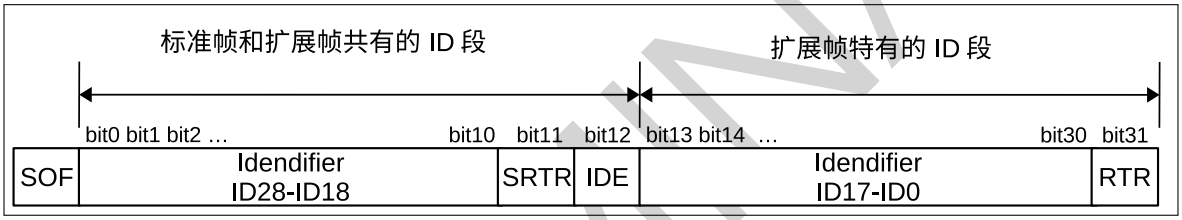


图 14-11. 丢失仲裁的 bit 位置

14.5 寄存器列表

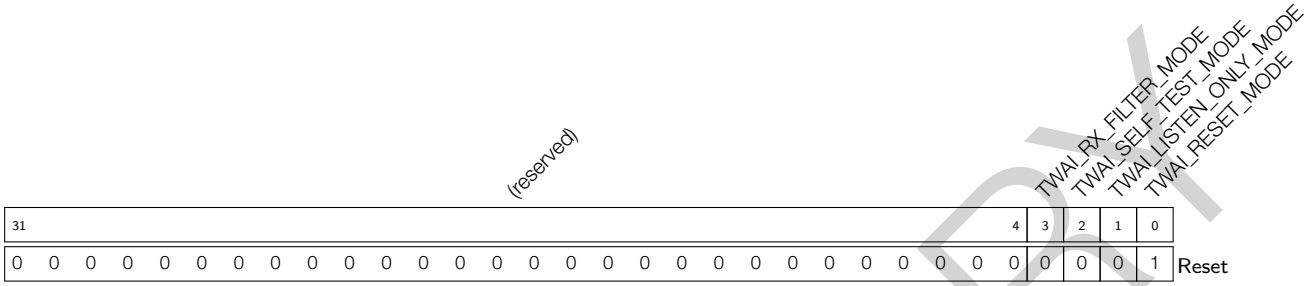
请注意，“访问权限”一栏中，“|”用于区分第 14.4.1 中描述的工作模式，其中左侧为操作模式下的访问权限，右侧标红字体为复位模式下的访问权限。本小节的所有地址均为相对于 Two-wire Automotive Interface 基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

名称	描述	地址	访问权限
配置寄存器			
TWAI_MODE_REG	模式寄存器	0x0000	读/写
TWAI_BUS_TIMING_0_REG	时序配置寄存器 0	0x0018	只读 | 读/写
TWAI_BUS_TIMING_1_REG	时序配置寄存器 1	0x001C	只读 | 读/写
TWAI_ERR_WARNING_LIMIT_REG	错误寄存器	0x0034	只读 | 读/写
TWAI_DATA_0_REG	数据寄存器 0	0x0040	只写 | 读/写
TWAI_DATA_1_REG	数据寄存器 1	0x0044	只写 | 读/写
TWAI_DATA_2_REG	数据寄存器 2	0x0048	只写 | 读/写
TWAI_DATA_3_REG	数据寄存器 3	0x004C	只写 | 读/写
TWAI_DATA_4_REG	数据寄存器 4	0x0050	只写 | 读/写
TWAI_DATA_5_REG	数据寄存器 5	0x0054	只写 | 读/写
TWAI_DATA_6_REG	数据寄存器 6	0x0058	只写 | 读/写
TWAI_DATA_7_REG	数据寄存器 7	0x005C	只写 | 读/写
TWAI_DATA_8_REG	数据寄存器 8	0x0060	只写 | 只读
TWAI_DATA_9_REG	数据寄存器 9	0x0064	只写 | 只读
TWAI_DATA_10_REG	数据寄存器 10	0x0068	只写 | 只读
TWAI_DATA_11_REG	数据寄存器 11	0x006C	只写 | 只读
TWAI_DATA_12_REG	数据寄存器 12	0x0070	只写 | 只读
TWAI_CLOCK_DIVIDER_REG	时钟分频寄存器	0x007C	不定
控制寄存器			
TWAI_CMD_REG	指令寄存器	0x0004	只写
状态寄存器			
TWAI_STATUS_REG	状态寄存器	0x0008	只读
TWAI_ARB LOST CAP_REG	仲裁丢失寄存器	0x002C	只读
TWAI_ERR_CODE_CAP_REG	错误捕获寄存器	0x0030	只读
TWAI_RX_ERR_CNT_REG	接收错误寄存器	0x0038	只读 | 读/写
TWAI_TX_ERR_CNT_REG	发送错误寄存器	0x003C	只读 | 读/写
TWAI_RX_MESSAGE_CNT_REG	接收数据寄存器	0x0074	只读
中断寄存器			
TWAI_INT_RAW_REG	中断寄存器	0x000C	只读
TWAI_INT ENA_REG	中断使能寄存器	0x0010	读/写

14.6 寄存器

请注意“访问权限”一栏中，“|”左侧为操作模式下的访问权限，右侧标红字体为复位模式下的访问权限。本小节的所有地址均为相对于 Two-wire Automotive Interface 基地址的地址偏移量（相对地址），具体基地址请见章节 3 系统和存储器 中的表 3-4。

Register 14.1. TWAI_MODE_REG (0x0000)



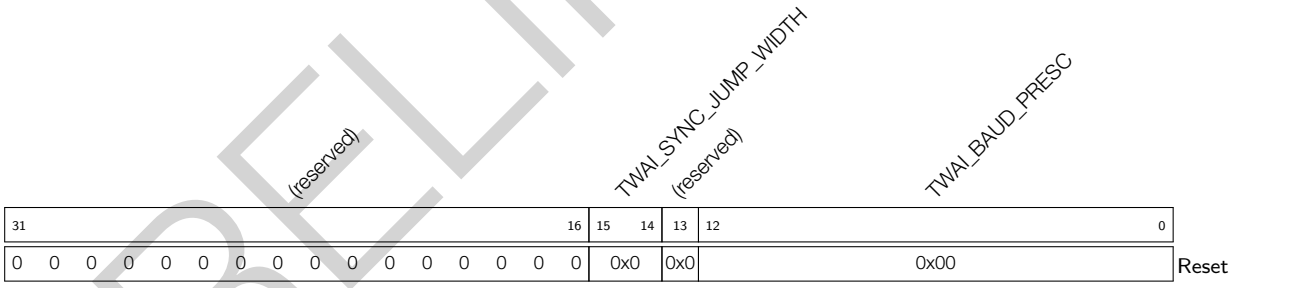
TWAI_RESET_MODE 配置 TWAI 控制器操作模式。1：复位模式；0：操作模式（读/写）

TWAI_LISTEN_ONLY_MODE 置 1 进入只听模式，处于该模式下的节点只接收总线上数据，不产生应答信号，也不更新接收错误计数。(读/写)

TWAI_SELF_TEST_MODE 置 1 启动自测模式，此模式下发送节点发送完数据后无需应答信号反馈。该模式常配合自接自收指令测试某个节点。(读/写)

TWAI_RX_FILTER_MODE 配置滤波模式。0: 双滤波模式；1: 单滤波模式（读/写）

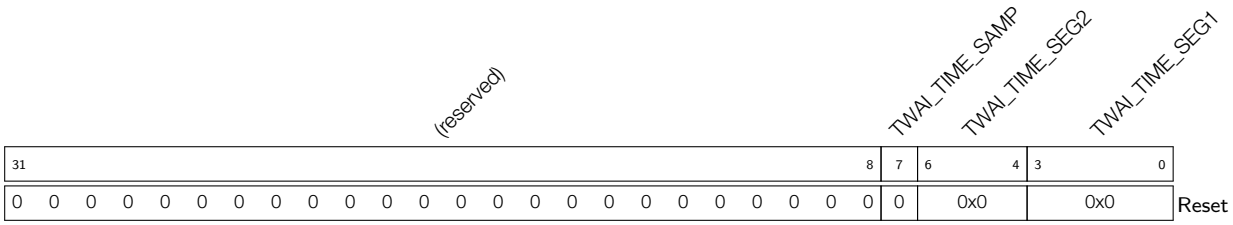
Register 14.2. TWAI_BUS_TIMING_0_REG (0x0018)



TWAI_BAUD_PRESC 预分频值，决定分频比例。(只读 | 读/写)

TWAI_SYNC_JUMP_WIDTH 同步跳宽 (SJW)，范围为 1 ~ 4 个时间定额。(只读 | 读/写)

Register 14.3. TWAI_BUS_TIMING_1_REG (0x001C)

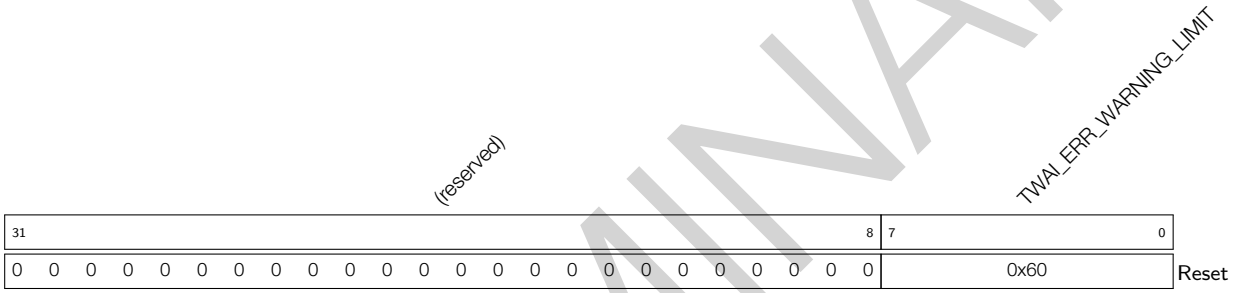


TWAI_TIME_SEG1 缓冲时期段 1 的宽度。(只读 | 读/写)

TWAI_TIME_SEG2 缓冲时期段 2 的宽度。(只读 | 读/写)

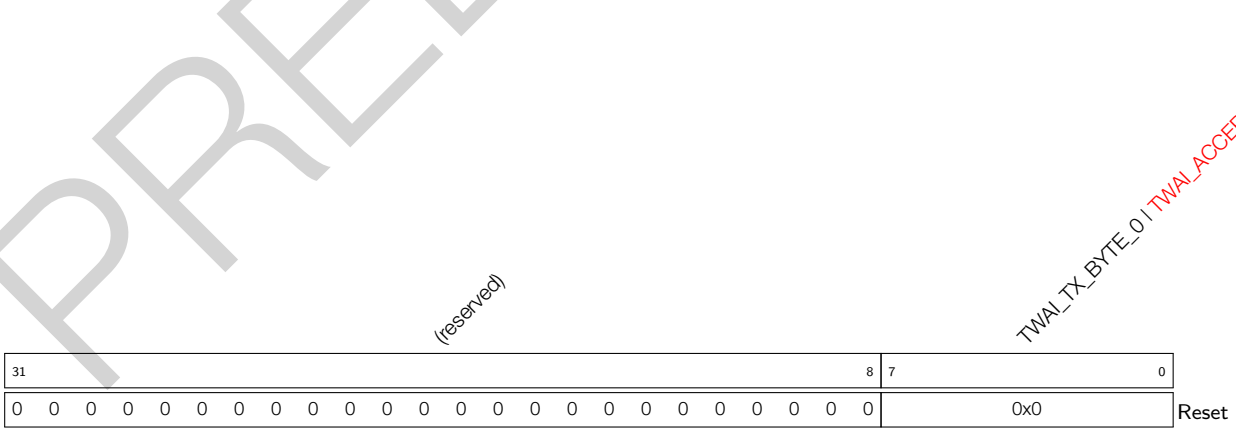
TWAI_TIME_SAMP 采样点数目。0：采样 1 次；1：采样三次（只读 | 读/写）

Register 14.4. TWAI_ERR_WARNING_LIMIT_REG (0x0034)



TWAI_ERR_WARNING_LIMIT 错误报警阈值，当任一错误计数数值超过该阈值或者所有错误计数数值都小于该阈值时，将触发错误报警中断（使能信号有效情况下）。(只读 | 读/写)

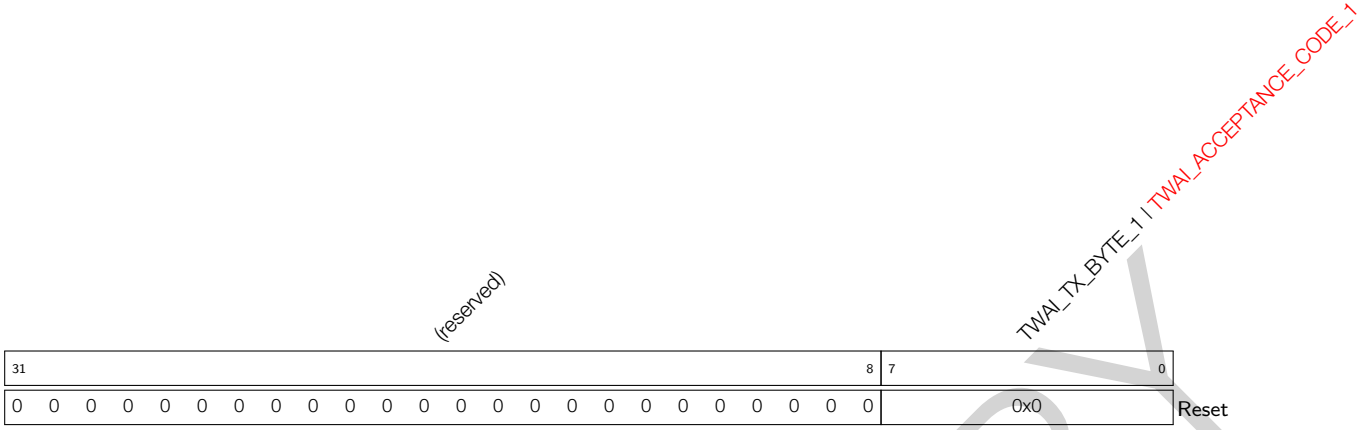
Register 14.5. TWAI_DATA_0_REG (0x0040)



TWAI_TX_BYTE_0 操作模式下，存储着待发送数据的第 0 个字节内容。(只写)

TWAI_ACCEPTANCE_CODE_0 复位模式下，存储着滤波编码的第 0 个字节。(读/写)

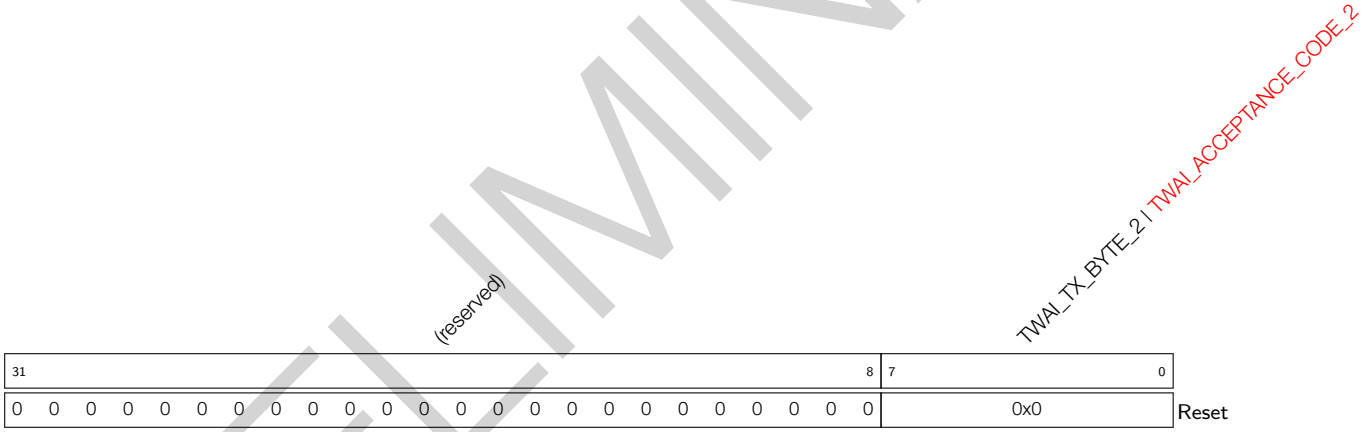
Register 14.6. TWAI_DATA_1_REG (0x0044)



TWAI_TX_BYTE_1 操作模式下，存储着待发送数据的第 1 个字节内容。(只写)

TWAI_ACCEPTANCE_CODE_1 复位模式下，存储着滤波编码的第 1 个字节。(读/写)

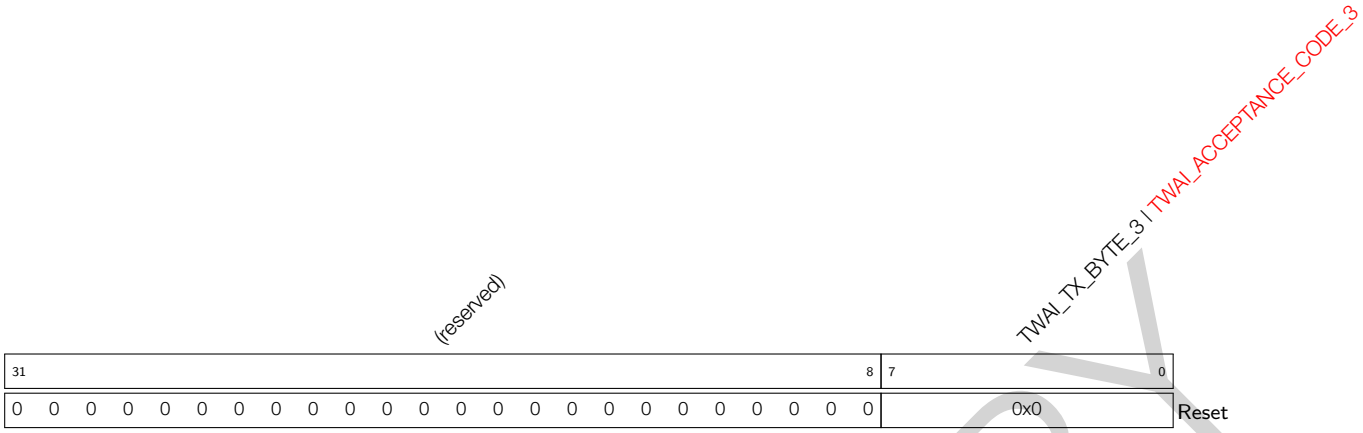
Register 14.7. TWAI_DATA_2_REG (0x0048)



TWAI_TX_BYTE_2 操作模式下，存储着待发送数据的第 2 个字节内容。(只写)

TWAI_ACCEPTANCE_CODE_2 复位模式下，存储着滤波编码的第 2 个字节。(读/写)

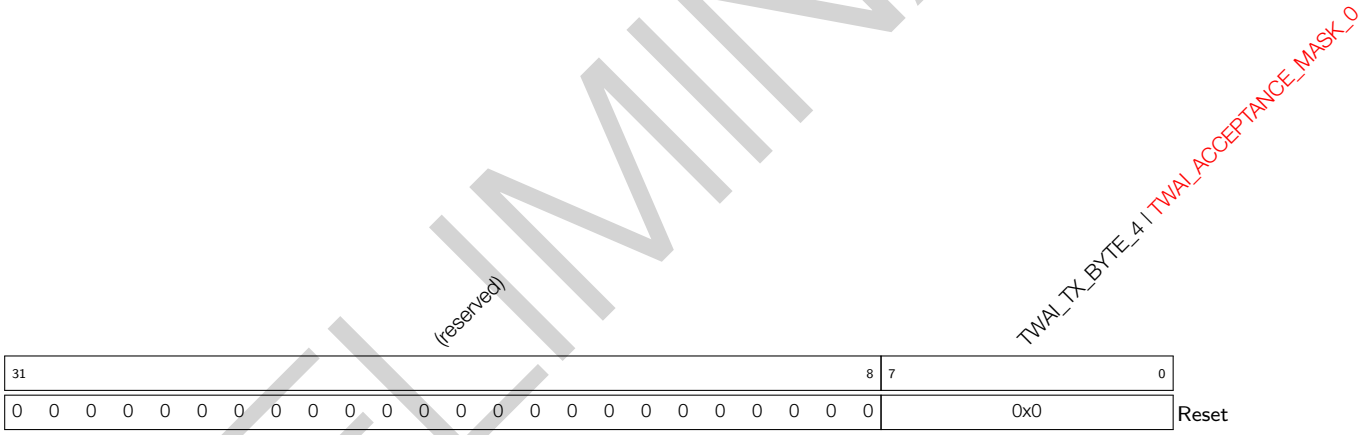
Register 14.8. TWAI_DATA_3_REG (0x004C)



TWAI_TX_BYTE_3 操作模式下，存储着待发送数据的第 3 个字节内容。(只写)

TWAI_ACCEPTANCE_CODE_3 复位模式下，存储着滤波编码的第 3 个字节。(读/写)

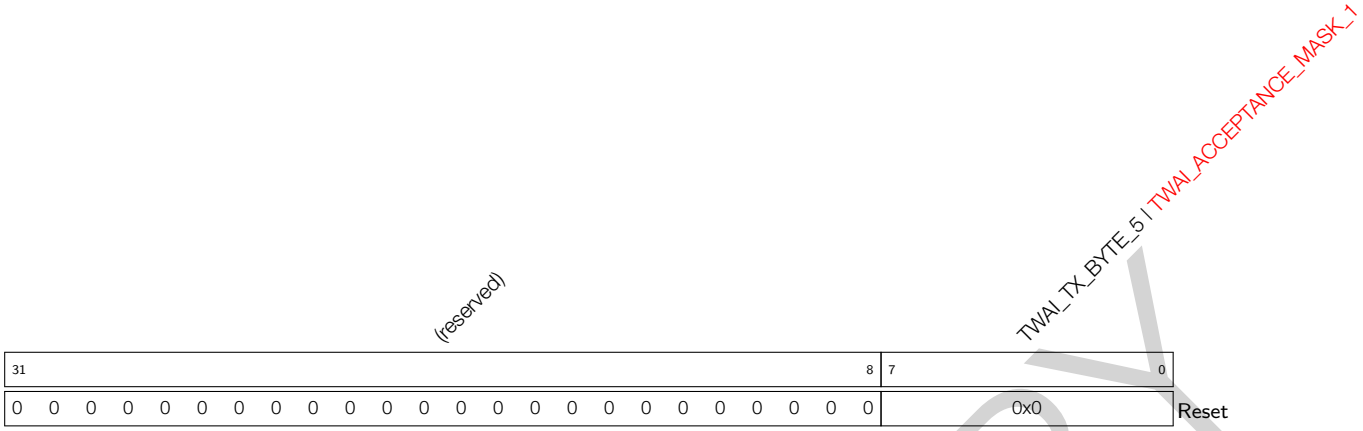
Register 14.9. TWAI_DATA_4_REG (0x0050)



TWAI_TX_BYTE_4 操作模式下，存储着待发送数据的第 4 个字节内容。(只写)

TWAI_ACCEPTANCE_MASK_0 复位模式下，存储着滤波编码的第 0 个字节。(读/写)

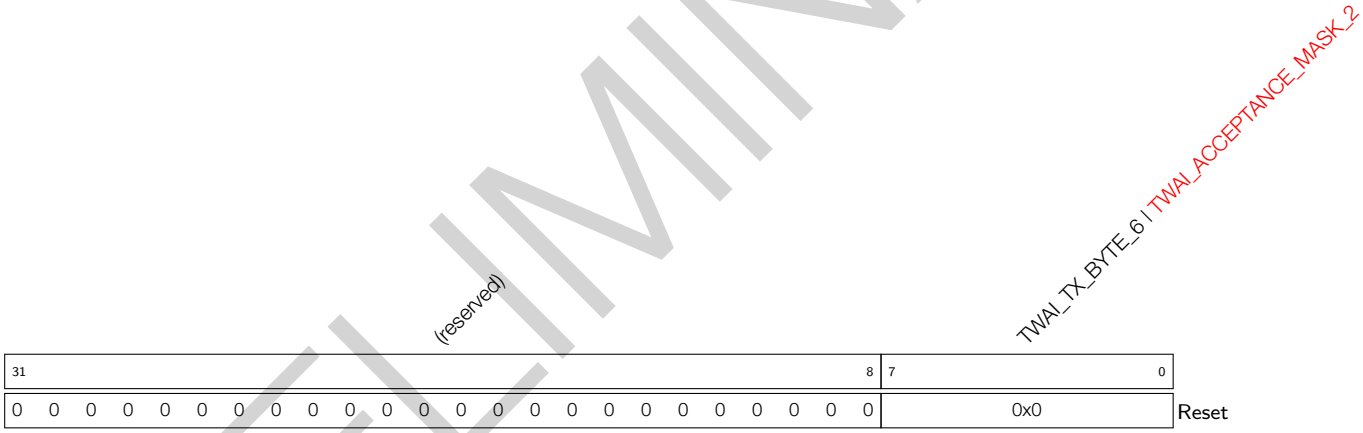
Register 14.10. TWAI_DATA_5_REG (0x0054)



TWAI_TX_BYTE_5 操作模式下，存储着待发送数据的第 5 个字节内容。(只写)

TWAI_ACCEPTANCE_MASK_1 复位模式下，存储着滤波编码的第 1 个字节。(读/写)

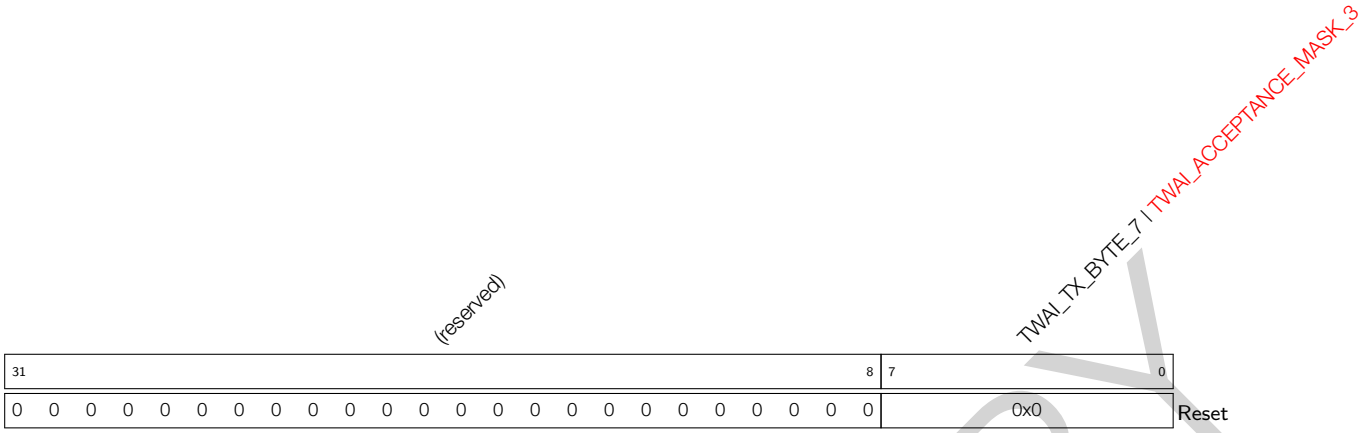
Register 14.11. TWAI_DATA_6_REG (0x0058)



TWAI_TX_BYTE_6 操作模式下，存储着待发送数据的第 6 个字节内容。(只写)

TWAI_ACCEPTANCE_MASK_2 复位模式下，存储着滤波编码的第 2 个字节。(读/写)

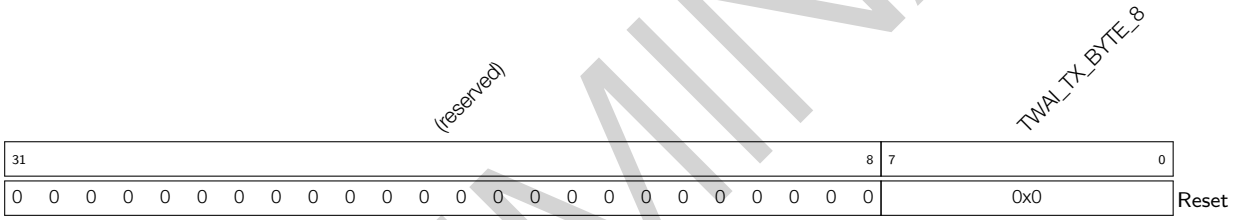
Register 14.12. TWAI_DATA_7_REG (0x005C)



TWAI_TX_BYTE_7 操作模式下，存储着待发送数据的第 7 个字节内容。(只写)

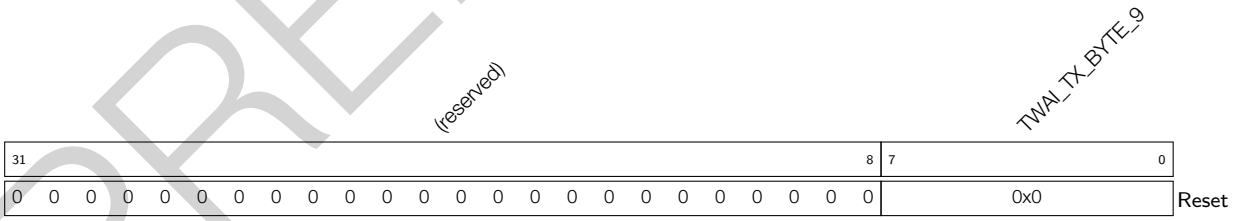
TWAI_ACCEPTANCE_MASK_3 复位模式下，存储着滤波编码的第 3 个字节。(读/写)

Register 14.13. TWAI_DATA_8_REG (0x0060)



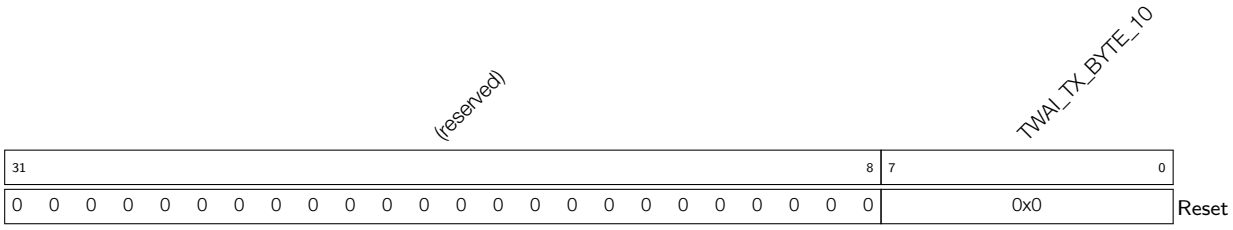
TWAI_TX_BYTE_8 操作模式下，存储着待发送数据的第 8 个字节内容。(只写)

Register 14.14. TWAI_DATA_9_REG (0x0064)



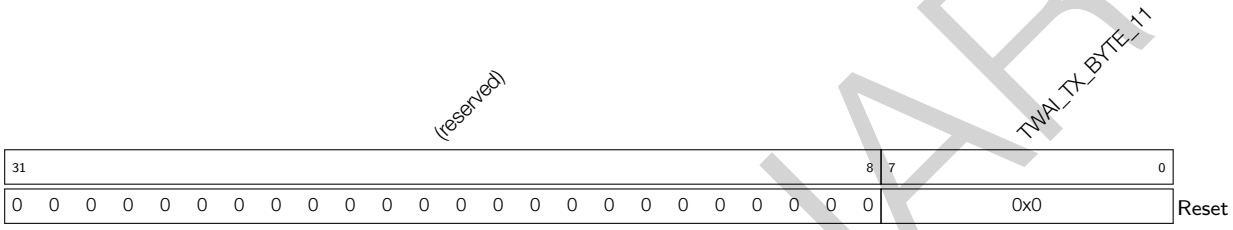
TWAI_TX_BYTE_9 操作模式下，存储着待发送数据的第 9 个字节内容。(只写)

Register 14.15. TWAI_DATA_10_REG (0x0068)



TWAI_TX_BYTE_10 操作模式下，存储着待发送数据的第 10 个字节内容。(只写)

Register 14.16. TWAI_DATA_11_REG (0x006C)

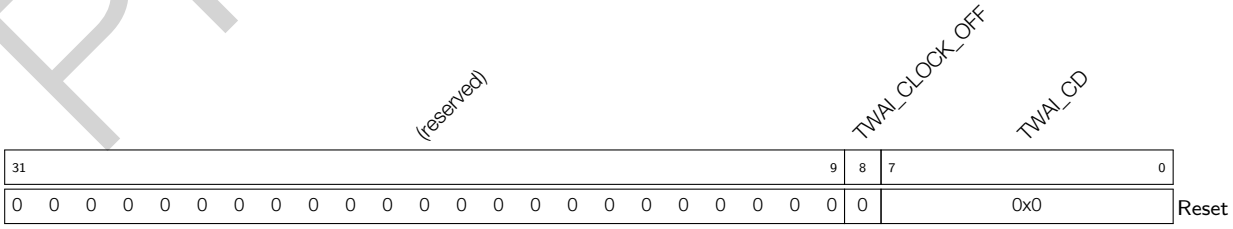


TWAI_TX_BYTE_11 操作模式下，存储着待发送数据的第 11 个字节内容。(只写)

Register 14.17. TWAI_DATA_12_REG (0x0070)

TWAI_TX_BYTE_12 操作模式下，存储着待发送数据的第 12 个字节内容。(只写)

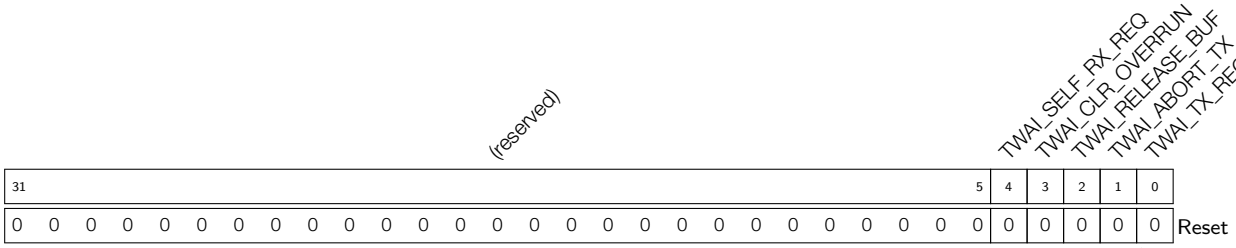
Register 14.18. TWAI_CLOCK_DIVIDER_REG (0x007C)



TWAI_CD 配置输出时钟 CLKOUT 的分频系数。(读/写)

TWAI_CLOCK_OFF 复位模式下可配。1：关闭输出的 CLKOUT 时钟；0：打开 CLKOUT 时钟（只读 | 读/写）

Register 14.19. TWAI_CMD_REG (0x0004)



TWAI_TX_REQ 置 1 驱动节点开始发送数据任务。(只写)

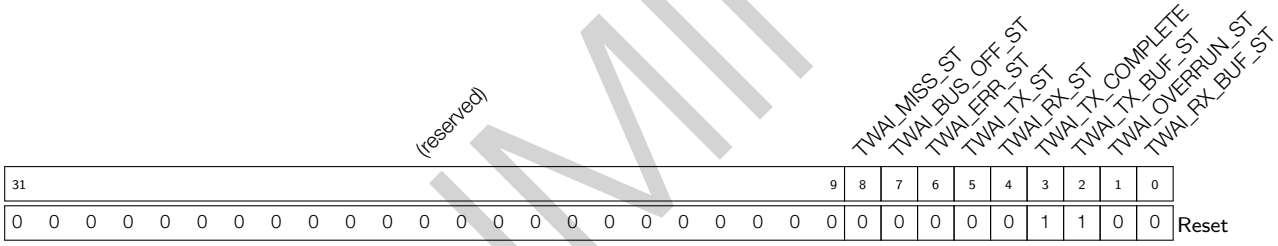
TWAI_ABORT_TX 置 1 取消当前还未开始的发送任务。(只写)

TWAI_RELEASE_BUF 置 1 释放接收缓冲器。(只写)

TWAI_CLR_OVERRUN 置 1 清除数据溢出状态。(只写)

TWAI_SELF_RX_REQ 自接自收命令。置 1 允许发送节点发送数据的同时接收总线上的数据。(只写)

Register 14.20. TWAI_STATUS_REG (0x0008)



TWAI_RX_BUF_ST 若值为 1，表明接收缓冲器中数据不为空，至少有一个已经接收到的数据包。(只读)

TWAI_OVERRUN_ST 若值为 1，表明接收 FIFO 中存储的数据已满，产生了溢出。(只读)

TWAI_TX_BUF_ST 若值为 1，表明发送缓冲器为空，允许写入待发送数据。(只读)

TWAI_TX_COMPLETE 若值为 1，表明成功从总线上接收到一个数据包。(只读)

TWAI_RX_ST 若值为 1，表明节点正在从总线上接收数据。(只读)

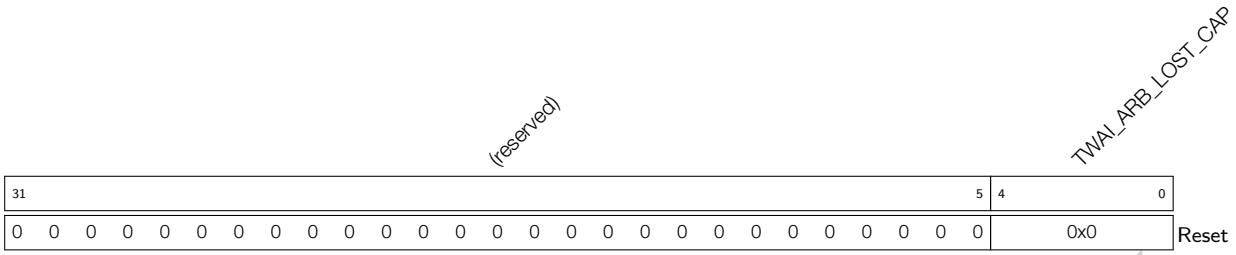
TWAI_TX_ST 若值为 1，表明节点正在往总线上发送数据。(只读)

TWAI_ERR_ST 若值为 1，表明接收错误计数和发送错误计数中至少有一个数值大于等于寄存器 TWAI_ERR_WARNING_LIMIT_REG 中配置的数值 (只读)

TWAI_BUS_OFF_ST 若值为 1，表明节点处于离线状态，不再响应总线上的数据传输。(只读)

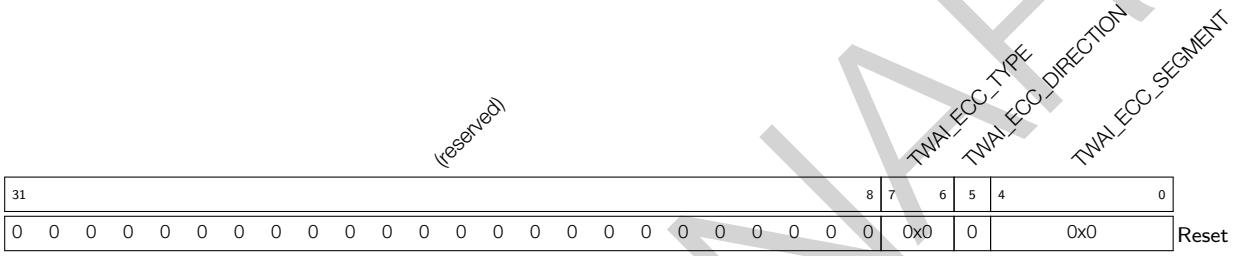
TWAI_MISS_ST 反映了从接收 FIFO 中取出数据包的完整状态。1: 当前数据包是缺失的；0：当前数据包是完整的 (只读)

Register 14.21. TWAI_ARB LOST CAP_REG (0x002C)



TWAI_ARB_LOST_CAP 记录着发送节点仲裁丢失的 bit 位置。（只读）

Register 14.22. TWAI_ERR_CODE_CAP_REG (0x0030)

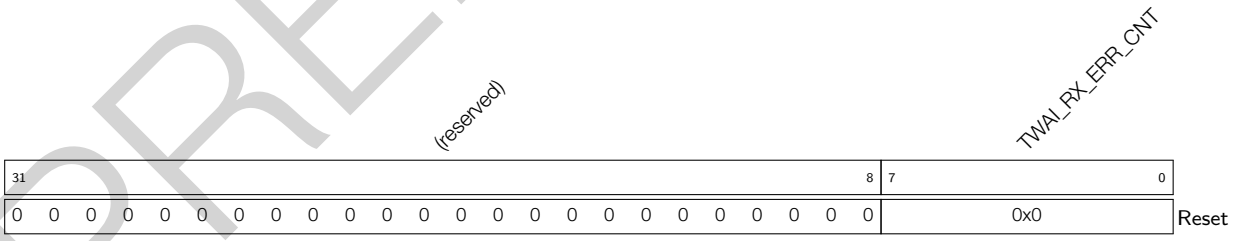


TWAI_ECC_SEGMENT 记录错误发生的位置，详见表 14-16。（只读）

TWAI_ECC_DIRECTION 记录错误时节点的数据传输方向。1：接收数据时发生错误；0：发送数据时发生错误（只读）

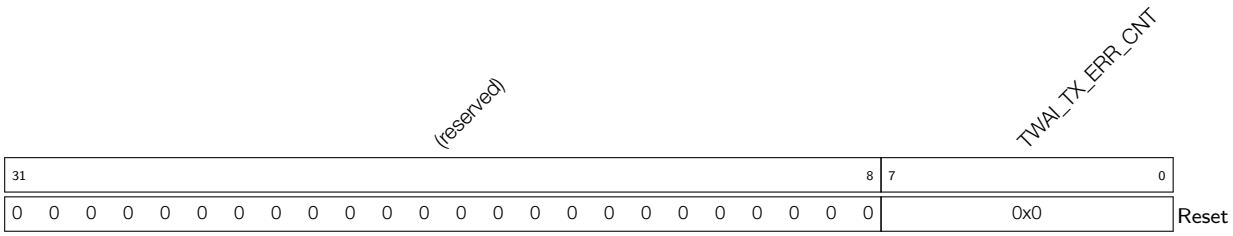
TWAI_ECC_TYPE 记录错误类别：00：位错误；01：格式错误；10：填充错误；11：其他错误（只读）

Register 14.23. TWAI_RX_ERR_CNT_REG (0x0038)



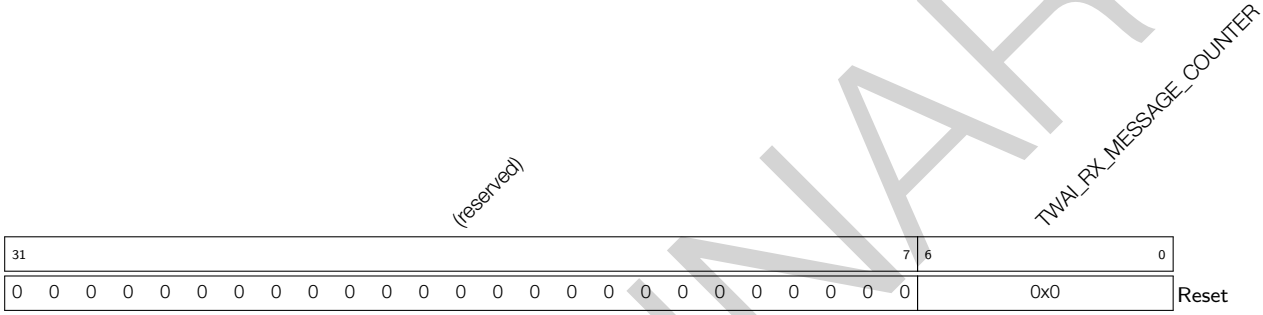
TWAI_RX_ERR_CNT 接收错误计数，数值变化发生在接收状态下。（只读 | 读/写）

Register 14.24. TWAI_TX_ERR_CNT_REG (0x003C)



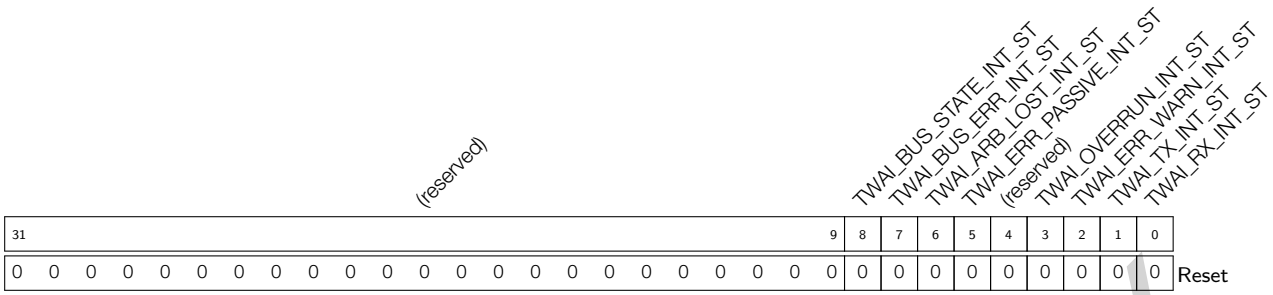
TWAI_TX_ERR_CNT 发送错误计数，数值变化发生在发送状态下。(只读 | 读/写)

Register 14.25. TWAI_RX_MESSAGE_CNT_REG (0x0074)



TWAI_RX_MESSAGE_COUNTER 存储着接收 FIFO 中数据包的个数。(只读)

Register 14.26. TWAI_INT_RAW_REG (0x000C)



TWAI_RX_INT_ST 接收中断。若值为 1，表明接收 FIFO 不为空，有接收数据待处理。(只读)

TWAI_TX_INT_ST 发送中断。若值为 1，表明节点数据发送任务结束，可以执行新的数据发送任务。(只读)

TWAI_ERR_WARN_INT_ST 错误报警中断。若值为 1，表明状态寄存器中错误状态信号和离线信号发生变化（0 变为 1 或 1 变为 0)。(只读)

TWAI_OVERRUN_INT_ST 数据溢出中断。若值为 1，表明节点的接收 FIFO 数据溢出。(只读)

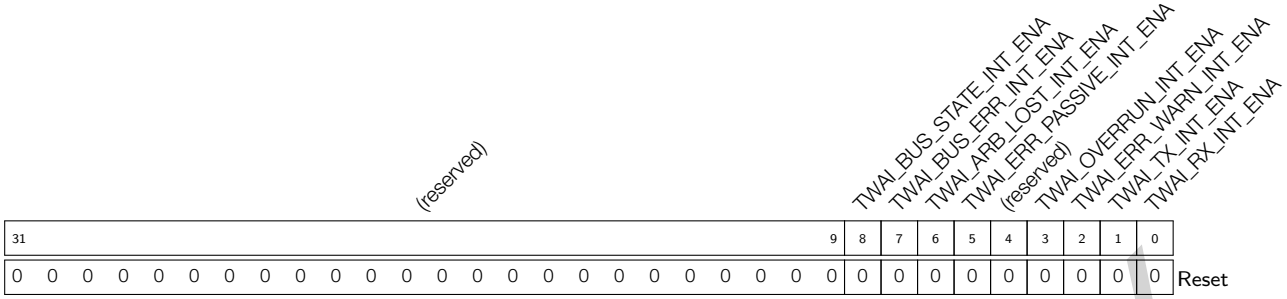
TWAI_ERR_PASSIVE_INT_ST 被动错误中断。若值为 1，表明节点由于错误计数数值的变化，在主动错误状态与被动错误状态间发生了切换。(只读)

TWAI_ARB_LOST_INT_ST 仲裁丢失中断。若值为 1，表明发送节点丢失仲裁。(只读)

TWAI_BUS_ERR_INT_ST 错误中断。若值为 1，表明节点检测到总线上发生了错误。(只读)

TWAI_BUS_STATE_INT_ST 总线状态中断。若值为 1，表明控制器状态发生了变化。(只读)

Register 14.27. TWAI_INT ENA_REG (0x0010)



TWAI_RX_INT_ENA 置 1 使能接收中断。(读/写)

TWAI_TX_INT_ENA 置 1 使能发送中断。(读/写)

TWAI_ERR_WARN_INT_ENA 置 1 使能错误报警中断。(读/写)

TWAI_OVERRUN_INT_ENA 置 1 使能数据溢出中断。(读/写)

TWAI_ERR_PASSIVE_INT_ENA 置 1 使能被动错误中断。(读/写)

TWAI_ARB_LOST_INT_ENA 置 1 使能仲裁丢失中断。(读/写)

TWAI_BUS_ERR_INT_ENA 置 1 使能总线错误中断。(读/写)

TWAI_BUS_STATE_INT_ENA 置 1 使能总线状态中断。(读/写)

15 LED PWM 控制器 (LEDC)

15.1 概述

LED PWM 控制器用于生成控制 LED 的脉冲宽度调制信号 (PWM)，具有占空比自动渐变等专门功能。该外设也可成生成 PWM 信号用作其他用途。

15.2 特性

LED PWM 控制器具有如下特性:

- 六个独立的 PWM 生成器（即六个通道）
- 四个独立定时器，可实现小数分频
- 占空比自动渐变（即 PWM 信号占空比可逐渐增加或减小，无须处理器干预），渐变完成时产生中断
- PWM 输出信号相位可调节
- 低功耗模式 (Light-sleep mode) 下可输出 PWM 信号
- PWM 最大精度为 14 位

四个定时器具有相同的功能和运行方式，下文将四个定时器统称为定时器*x*（*x* 的范围是 0 到 3）。六个 PWM 生成器的功能和运行方式也相同，下文将统称为 PWM *n*（n 的范围是 0 到 5）。

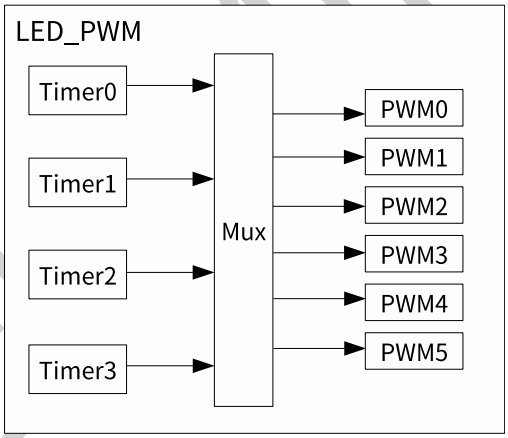


图 15-1. LED PWM 控制器架构

15.3 功能描述

15.3.1 架构

图 15-1为 LED PWM 控制器的架构。

四个定时器可独立配置（可配置时钟分频器和计数器最大值），每个定时器内部有一个时基计数器（即基于基准时钟周期计数的计数器）。每个 PWM 生成器在四个定时器中择一，以该定时器的计数值为基准生成 PWM 信号。

图 15-2 为定时器和 PWM 生成器的主要功能块。

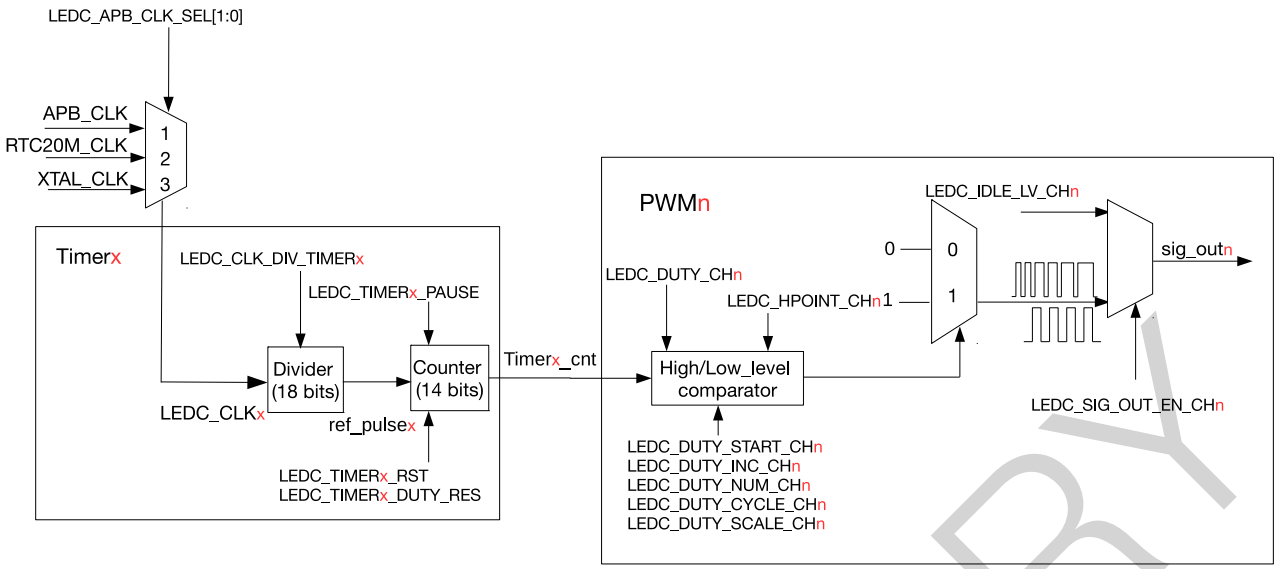


图 15-2. 定时器和 PWM 生成器功能块

15.3.2 定时器

LED PWM 控制器的每个定时器内部都有一个时基计数器。图 15-2 中时基计数器使用的时钟信号称为 ref_pulse*x*。所有定时器使用同一个时钟源信号 LEDC_CLK*x*，该时钟源信号经分频器分频后产生 ref_pulse*x* 供计数器使用。

15.3.2.1 时钟源

软件配置 LED PWM 寄存器由 APB_CLK 驱动。更多关于 APB_CLK 的信息，详见章节 6 *复位和时钟*。要使用 LED PWM 控制器，需使能 LED PWM 的 APB_CLK 时钟信号，该时钟信号可通过置位 SYSTEM_PERIP_CLK_EN0_REG 寄存器的 SYSTEM_LEDC_CLK_EN 使能，通过软件置位 SYSTEM_PERIP_RST_EN0_REG 寄存器的 SYSTEM_LEDC_RST 位复位。更多信息，请参阅章节 9 *系统寄存器 (SYSREG) [to be added later]* 的表 18。

LED PWM 控制器的定时器有三个时钟源信号可以选择：APB_CLK、RTC20M_CLK 和 XTAL_CLK（更多有关时钟源的信息详见章节 6 *复位和时钟*）。为 LEDC_CLK*x* 选择时钟源信号的配置如下：

- APB_CLK：将 LEDC_APB_CLK_SEL[1:0] 置 1
- RTC20M_CLK：将 LEDC_APB_CLK_SEL[1:0] 置 2
- XTAL_CLK：将 LEDC_APB_CLK_SEL[1:0] 置 3

之后，LEDC_CLK*x* 信号会进入时钟分频器。

15.3.2.2 时钟分频器配置

LEDC_CLK*x* 信号传输到时钟分频器，产生 ref_pulse*x* 信号供计数器使用。ref_pulse*x* 的频率等于 LEDC_CLK*x* 的频率经 LEDC_CLK_DIV_TIMER*x*分频系数分频后的结果（见图 15-2）。

LEDC_CLK_DIV_TIMER*x* 分频系数为小数分频，因此其值可为非整数。分频系数 LEDC_CLK_DIV_TIMER*x*可根据下列等式通过 LEDC_CLK_DIV_TIMER*x* 字段配置：

$$LEDC\_CLK\_DIV\_TIMERx = A + \frac{B}{256}$$

- 整数部分 *A* 为 LEDC_CLK_DIV_TIMER*x* 字段的高 10 位（即 LEDC_TIMER*x*_CONF_REG[21:12]）
- 小数部分 *B* 为 LEDC_CLK_DIV_TIMER*x* 字段的低 8 位（即 LEDC_TIMER*x*_CONF_REG[11:4]）

小数部分 *B* 为 0 时，LEDC_CLK_DIV_TIMER*x*的值为整数（整数分频）。也就是说，每 *A* 个 LEDC_CLK*x* 时钟周期产生一个 ref_pulse*x* 时钟脉冲。

小数部分 *B* 不为 0 时，LEDC_CLK_DIV_TIMER*x*的值非整数。时钟分频器按照 *A* 个 LEDC_CLK*x* 时钟周期和 (*A*+1) 个 LEDC_CLK*x* 时钟周期轮流进行非整数分频。这样一来，ref_pulse*x* 时钟脉冲的平均频率便会是理想值（非整数分频的频率）。每 256 个 ref_pulse*x* 时钟脉冲中：

- 有 *B* 个以 (*A*+1) 个 LEDC_CLK*x* 时钟周期分频
- 有 (256-*B*) 个以 *A* 个 LEDC_CLK*x* 时钟周期分频
- 以 (*A*+1) 个 LEDC_CLK*x* 时钟周期分频的时钟脉冲均匀分布在以 *A* 分频的时钟脉冲中

图 15-3 展示了 LEDC_CLK_DIV_TIMER*x*分频系数非整数时，LEDC_CLK*x* 时钟周期和 ref_pulse*x* 时钟脉冲的关系。

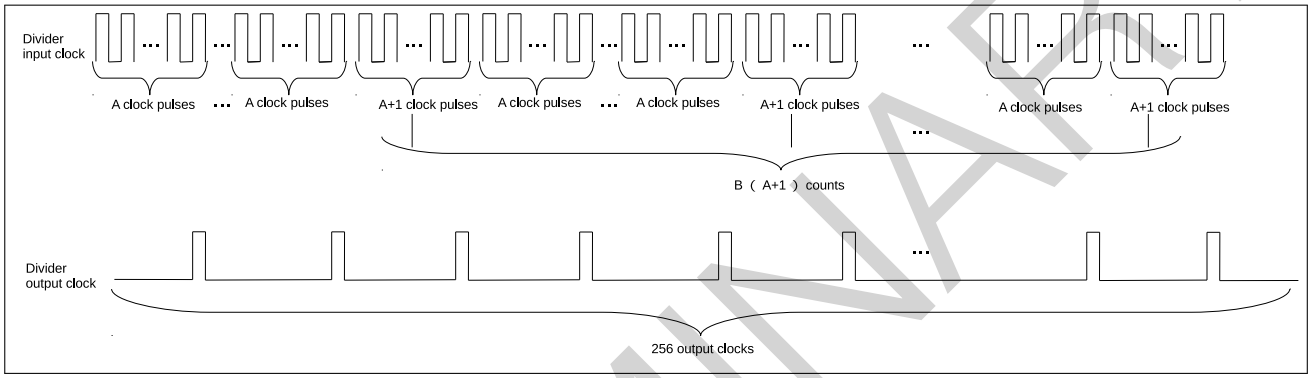


图 15-3. LEDC_CLK_DIV_TIMER*x* 非整数时的分频

在运行时改变定时器时钟的分频系数，需先置位 LEDC_CLK_DIV_TIMER*x* 字段，然后置位 LEDC_TIMER*x*_PARA_UP 字段应用新配置。新配置会在计数器下次溢出时生效。LEDC_TIMER*x*_PARA_UP 字段由硬件自动清除。

15.3.2.3 14 位计数器

每个定时器有一个以 ref_pulse*x* 为基准时钟的 14 位时基计数器（见图 15-2）。LEDC_TIMER*x*_DUTY_RES 字段用于配置 14 位计数器的最大值。因此，PWM 信号的最大精确度为 14 位。计数器最大可计数至 $2^{LEDC\_TIMERx\_DUTY\_RES}$ − 1，然后溢出重新从 0 开始计数。软件可以读取、复位、暂停计数器。

计数器可在每次溢出时触发 (LEDC_TIMER*x*_OVF_INT) 中断，这个中断为硬件自动产生，不需要配置。计数器也可配置为在溢出 $(LEDC\_OVF\_NUM\_CHn + 1)$ 次时触发 LEDC_OVF_CNT_CH*n*_INT 中断，该中断配置步骤如下：

1. 配置 LEDC_TIMER_SEL_CH*n* 为 PWM 生成器选择该计数器
2. 置位 LEDC_OVF_CNT_EN_CH*n* 使能计数器
3. 把 LEDC_OVF_NUM_CH*n* 的值设为计数器触发中断的溢出次数减 1
4. 置位 LEDC_OVF_CNT_CH*n*_INT_ENA 使能溢出中断
5. 置位 LEDC_TIMER*x*_DUTY_RES 使能定时器，等待 LEDC_OVF_CNT_CH*n*_INT 中断产生

如图 15-2 所示，PWM 生成器输出信号 sig_out*n* 的频率取决于定时器的时钟源 LEDC_CLK*x*、时钟分频系数 LEDC_CLK_DIV_TIMER*x* 以及计数器的计数范围 LEDC_TIMER*x*_DUTY_RES：

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

在运行时改变计数器的最大值，需先置位 LEDC_TIMERx_DUTY_RES 字段，然后置位 LEDC_TIMERx_PARA_UP 字段。新的配置在计数器下一次溢出时生效。如果重新配置 LEDC_OVF_CNT_EN_CHn 字段，也需置位 LEDC_PARA_UP_CHn 应用新配置。总之，更改配置时都需置位 LEDC_PARA_UP_CHn 应用新配置。LEDC_TIMERx_PARA_UP 字段由硬件自动清除。

15.3.3 PWM 生成器

要生成 PWM 信号，PWM 生成器 (PWMn) 需选择一个定时器 (Timerx)。每个 PWM 生成器均可通过置位 LEDC_TIMER_SEL_CHn 单独配置，在四个定时器中选择一个输出 PWM 信号。

如图 15-2所示，每个 PWM 生成器主要包括一个高低电平比较器和两个选择器。PWM 生成器将定时器的 14 位计数值 (Timerx_cnt) 与高低电平比较器的值 Hpointn 和 Lpointn 比较。如果定时器的计数值等于 Hpointn 或 Lpointn，PWM 信号可以输出高低电平：

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

图 15-4 展示了如何使用 Hpointn 和 Lpointn 生成占空比固定的 PWM 信号。

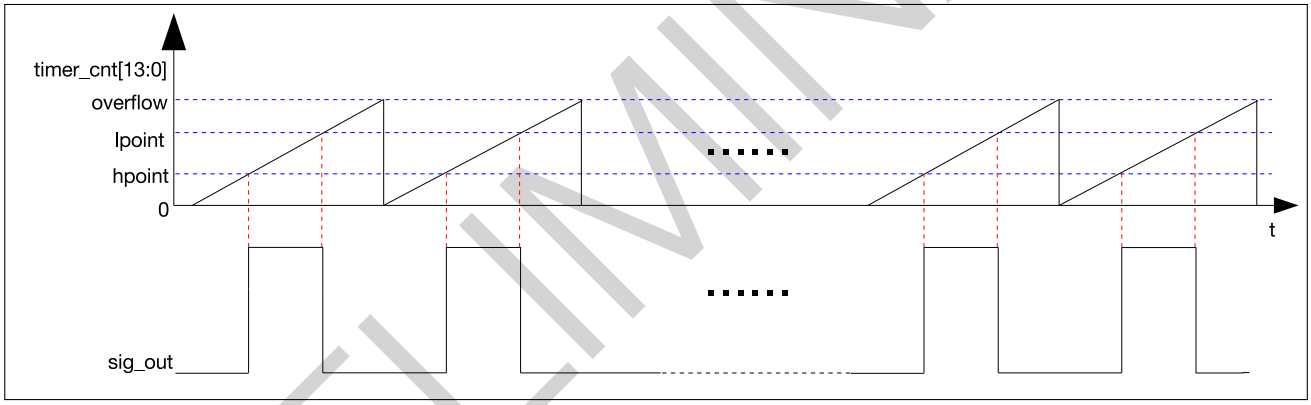


图 15-4. LED_PWM Output Signal Diagram

每当所选定时器的计数器溢出时，PWM 生成器 (PWMn) 的 Hpointn 值更新为 LEDC_HPOINT_CHn。Lpointn 的值同样在计数器每次溢出时更新，为 LEDC_DUTY_CHn[18:4] 和 LEDC_HPOINT_CHn 的和。通过配置 LEDC_DUTY_CHn[18:4] 和 LEDC_HPOINT_CHn 两个字段，可设置 PWM 输出的相对相位和占空比。

置位 LEDC_SIG_OUT_EN_CHn，开启 PWM 信号（sig_outn）输出；清除 LEDC_SIG_OUT_EN_CHn，关闭 PWM 信号输出，输出信号 sig_outn 输出恒定电平，电平值为 LEDC_IDLE_LV_CHn.

LEDC_DUTY_CHn[3:0] 通过周期性改变 PWM 输出信号 sig_outn 的占空比实现微调。如 LEDC_DUTY_CHn[3:0] 不为 0，那么 sig_outn 每 16 个周期中，有 LEDC_DUTY_CHn[3:0] 个周期的 PWM 脉冲占空比要比 (16 - LEDC_DUTY_CHn[3:0]) 个周期的脉冲占空比多一个定时器的计数周期。比如，如果 LEDC_DUTY_CHn[18:4] 设为 10，LEDC_DUTY_CHn[3:0] 设为 5，则 16 个周期中，有 5 个周期的 PWM 脉冲占空比为 11，剩余 11 个周期的 PWM 脉冲占空比为 10。16 个周期的平均占空比为 10.3125。

如果重新配置 LEDC_TIMER_SEL_CHn、LEDC_HPOINT_CHn、LEDC_DUTY_CHn[18:4] 和 LEDC_SIG_OUT_EN_CHn 字段，需置位 LEDC_PARA_UP_CHn 应用新配置。新配置在计数器下次溢出时生效。LEDC_TIMERx_PARA_UP 字段由硬件自动清除。

15.3.4 占空比渐变

PWM 生成器可以渐变 PWM 输出信号的占空比，即由一种占空比逐渐变为为另一种占空比。如果开启占空比渐变功能，Lpoint*n* 的值会在计数器溢出固定次数后递增或递减。图 15-5 展示了占空比渐变功能。

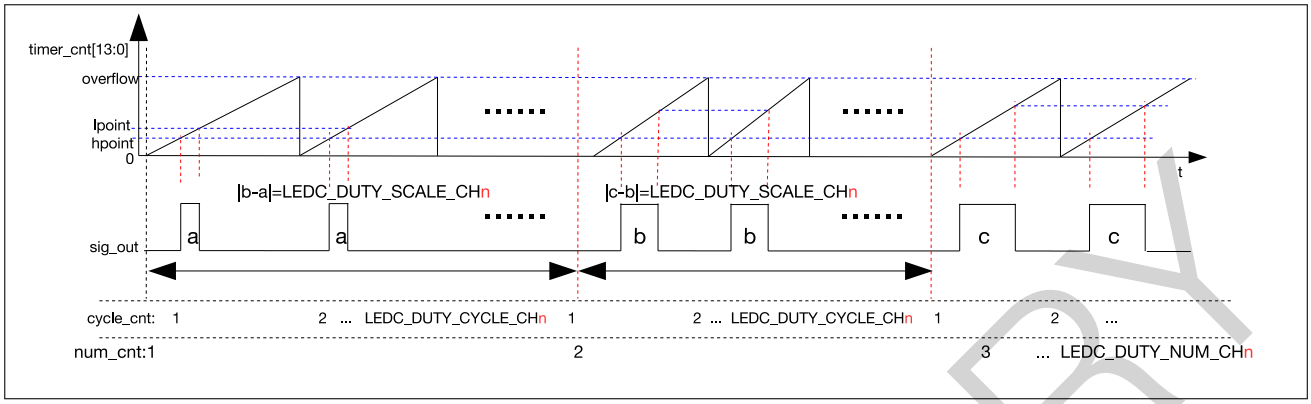


图 15-5. 输出信号占空比渐变图

占空比渐变功能可通过以下寄存器字段配置：

- LEDC_DUTY_CH*n* 用于设置 Lpoint*n* 的初始值。
- LEDC_DUTY_START_CH*n* 置 1 或清零，使能或关闭占空比渐变功能。
- LEDC_DUTY_CYCLE_CH*n* 用于设置 Lpoint*n* 在计数器溢出多少次时递增或递减。也就是说，Lpoint*n* 会在计数器溢出 LEDC_DUTY_CYCLE_CH*n* 次时递增或递减。
- LEDC_DUTY_INC_CH*n* 置 1 或清零，Lpoint*n* 递增或递减。
- LEDC_DUTY_SCALE_CH*n* 用于设置 Lpoint*n* 递增或递减的值。
- LEDC_DUTY_NUM_CH*n* 用于设置占空比渐变停止前，Lpoint*n* 递增或递减的最大次数。

如果重新配置 LEDC_DUTY_CH*n*、LEDC_DUTY_START_CH*n*、LEDC_DUTY_CYCLE_CH*n*、LEDC_DUTY_INC_CH*n*、LEDC_DUTY_SCALE_CH*n* 和 LEDC_DUTY_NUM_CH*n*字段，需置位 LEDC_PARA_UP_CH*n* 应用新配置。LEDC_PARA_UP_CH*n* 置位后，新配置立即生效。LEDC_TIMER*x*_PARA_UP 字段由硬件自动清除。

15.3.5 中断

- LEDC_OVF_CNT_CH*n*_INT：定时器计数器溢出 (LEDC_OVF_NUM_CH*n* + 1) 次且寄存器 LEDC_OVF_CNT_EN_CH*n* 置 1 时触发中断。
- LEDC_DUTY_CHNG_END_CH*n*_INT：PWM 生成器渐变完成后触发中断。
- LEDC_TIMER*x*_OVF_INT：定时器达到最大计数值时触发中断。

15.4 寄存器列表

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

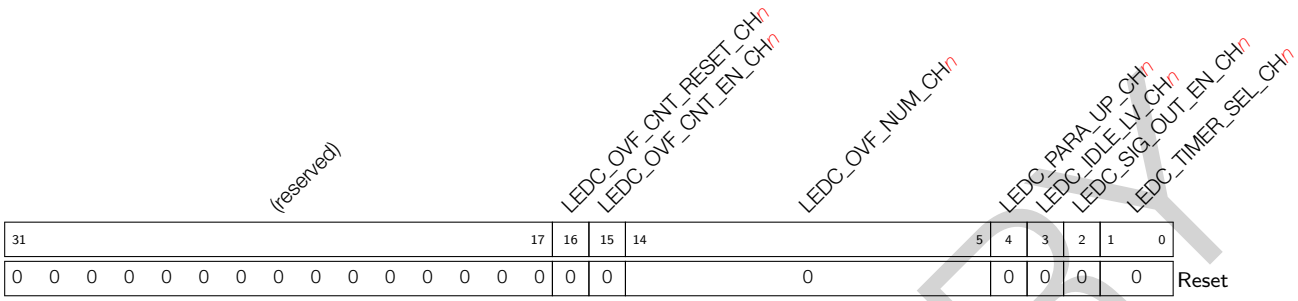
名称	描述	地址	访问
配置寄存器			
LEDC_CH0_CONF0_REG	通道 0 的配置寄存器 0	0x0000	varies
LEDC_CH0_CONF1_REG	通道 0 的配置寄存器 1	0x000C	varies
LEDC_CH1_CONF0_REG	通道 1 的配置寄存器 0	0x0014	varies
LEDC_CH1_CONF1_REG	通道 1 的配置寄存器 1	0x0020	varies
LEDC_CH2_CONF0_REG	通道 2 的配置寄存器 0	0x0028	varies
LEDC_CH2_CONF1_REG	通道 2 的配置寄存器 1	0x0034	varies
LEDC_CH3_CONF0_REG	通道 3 的配置寄存器 0	0x003C	varies
LEDC_CH3_CONF1_REG	通道 3 的配置寄存器 1	0x0048	varies
LEDC_CH4_CONF0_REG	通道 4 的配置寄存器 0	0x0050	varies
LEDC_CH4_CONF1_REG	通道 4 的配置寄存器 1	0x005C	varies
LEDC_CH5_CONF0_REG	通道 5 的配置寄存器 0	0x0064	varies
LEDC_CH5_CONF1_REG	通道 5 的配置寄存器 1	0x0070	varies
LEDC_CONF_REG	LEDC 全局配置寄存器	0x00D0	R/W
高位点寄存器			
LEDC_CH0_HPOINT_REG	通道 0 的高位点寄存器	0x0004	R/W
LEDC_CH1_HPOINT_REG	通道 1 的高位点寄存器	0x0018	R/W
LEDC_CH2_HPOINT_REG	通道 2 的高位点寄存器	0x002C	R/W
LEDC_CH3_HPOINT_REG	通道 3 的高位点寄存器	0x0040	R/W
LEDC_CH4_HPOINT_REG	通道 4 的高位点寄存器	0x0054	R/W
LEDC_CH5_HPOINT_REG	通道 5 的高位点寄存器	0x0068	R/W
占空比寄存器			
LEDC_CH0_DUTY_REG	通道 0 的初始占空比	0x0008	R/W
LEDC_CH0_DUTY_R_REG	通道 0 的当前占空比	0x0010	RO
LEDC_CH1_DUTY_REG	通道 1 的初始占空比	0x001C	R/W
LEDC_CH1_DUTY_R_REG	通道 1 的当前占空比	0x0024	RO
LEDC_CH2_DUTY_REG	通道 2 的初始占空比	0x0030	R/W
LEDC_CH2_DUTY_R_REG	通道 2 的当前占空比	0x0038	RO
LEDC_CH3_DUTY_REG	通道 3 的初始占空比	0x0044	R/W
LEDC_CH3_DUTY_R_REG	通道 3 的当前占空比	0x004C	RO
LEDC_CH4_DUTY_REG	通道 4 的初始占空比	0x0058	R/W
LEDC_CH4_DUTY_R_REG	通道 4 的当前占空比	0x0060	RO
LEDC_CH5_DUTY_REG	通道 5 的初始占空比	0x006C	R/W
LEDC_CH5_DUTY_R_REG	通道 5 的当前占空比	0x0074	RO
定时器寄存器			
LEDC_TIMER0_CONF_REG	定时器 0 配置	0x00A0	varies
LEDC_TIMER0_VALUE_REG	定时器 0 的当前计数器值	0x00A4	RO
LEDC_TIMER1_CONF_REG	定时器 1 配置	0x00A8	varies
LEDC_TIMER1_VALUE_REG	定时器 1 的当前计数器值	0x00AC	RO

名称	描述	地址	访问
LEDC_TIMER2_CONF_REG	定时器 2 配置	0x00B0	varies
LEDC_TIMER2_VALUE_REG	定时器 2 的当前计数器值	0x00B4	RO
LEDC_TIMER3_CONF_REG	定时器 3 配置	0x00B8	varies
LEDC_TIMER3_VALUE_REG	定时器 3 的当前计数器值	0x00BC	RO
中断寄存器			
LEDC_INT_RAW_REG	原始中断状态	0x00C0	R/WTC/SS
LEDC_INT_ST_REG	屏蔽中断状态	0x00C4	RO
LEDC_INT_ENA_REG	中断使能位	0x00C8	R/W
LEDC_INT_CLR_REG	中断清除位	0x00CC	WT
版本寄存器			
LEDC_DATE_REG	版本控制寄存器	0x00FC	R/W

15.5 寄存器

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

Register 15.1. LEDC_CH*n*_CONF0_REG (*n*: 0-5) (0x0000+20n*)**



LEDC_TIMER_SEL_CH*n* 用于选择通道 *n* 的定时器。

0：选择定时器 0

1：选择定时器 1

2：选择定时器 2

3：选择定时器 3 (R/W)

LEDC_SIG_OUT_EN_CH*n* 置位此位，使能通道 *n* 的信号输出。(R/W)

LEDC_IDLE_LV_CH*n* 控制通道 *n* 不工作时（LEDC_SIG_OUT_EN_CH*n* 为 0 时）的输出电平。(R/W)

LEDC_PARA_UP_CH*n* 用于更新通道 *n* 的下列字段，由硬件自动清除。(WT)

- LEDC_HPOINT_CH*n*
- LEDC_DUTY_START_CH*n*
- LEDC_SIG_OUT_EN_CH*n*
- LEDC_TIMER_SEL_CH*n*
- LEDC_DUTY_NUM_CH*n*
- LEDC_DUTY_CYCLE_CH*n*
- LEDC_DUTY_SCALE_CH*n*
- LEDC_DUTY_INC_CH*n*
- LEDC_OVF_CNT_EN_CH*n*

见下页...

Register 15.1. LEDC_CH*n*_CONF0_REG (*n*: 0-5) (0x0000+20n*)**

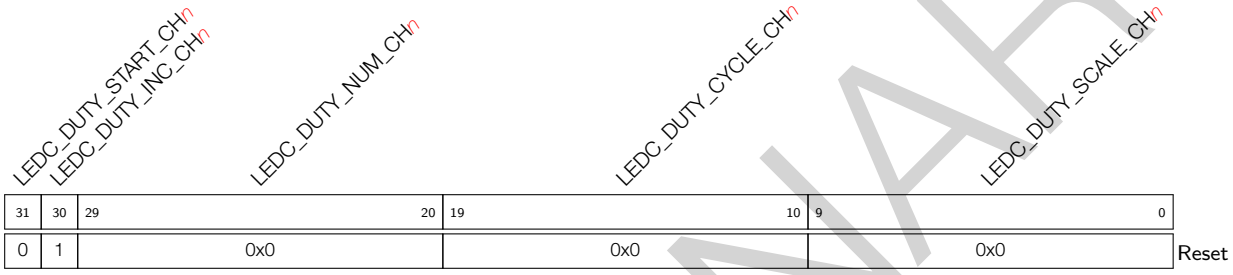
接上页...

LEDC_OVF_NUM_CH*n* 用于配置定时器溢出次数的最大值减 1。通道 *n* 的定时器溢出次数达到 (LEDC_OVF_NUM_CH*n* +1) 次时，触发 LEDC_OVF_CNT_CH*n*_INT 中断。(R/W)

LEDC_OVF_CNT_EN_CH*n* 用于计算通道 *n* 选择的定时器溢出的次数。(R/W)

LEDC_OVF_CNT_RESET_CH*n* 置位此位，复位通道 *n* 的定时器溢出计数器。(WT)

Register 15.2. LEDC_CH*n*_CONF1_REG (*n*: 0-5) (0x000C+20n*)**



LEDC_DUTY_SCALE_CH*n* 用于配置渐变时占空比的步长变化。(R/W)

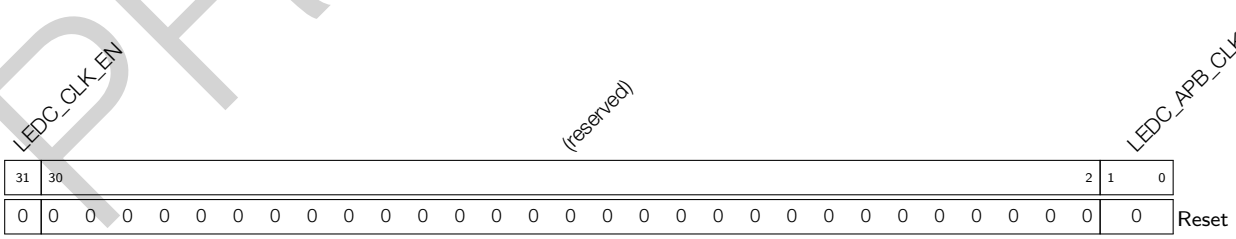
LEDC_DUTY_CYCLE_CH*n* 通道 *n* 占空比每隔 LEDC_DUTY_CYCLE_CH*n* 周期变化一次。(R/W)

LEDC_DUTY_NUM_CH*n* 用于控制占空比变化的次数。(R/W)

LEDC_DUTY_INC_CH*n* 决定了通道 *n* 输出信号的占空比是递增还是递减。1：递增；0：递减。(R/W)

LEDC_DUTY_START_CH*n* 此位置 1 时，LEDC_CH*n*_CONF1_REG 中的其他字段在定时器下次溢出时生效。(R/W/SC)

Register 15.3. LEDC_CONF_REG (0x00D0)

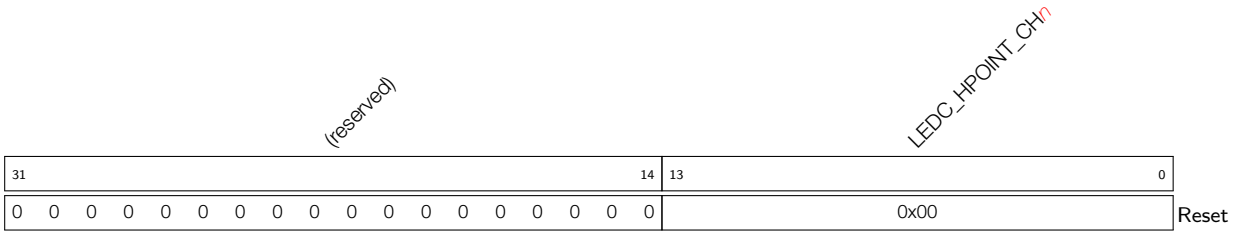


LEDC_APB_CLK_SEL 用于设置 4 个定时器共同的时钟源。1：APB_CLK；2：RTC20M_CLK；3：XTAL_CLK。(R/W)

LEDC_CLK_EN 用于控制时钟。

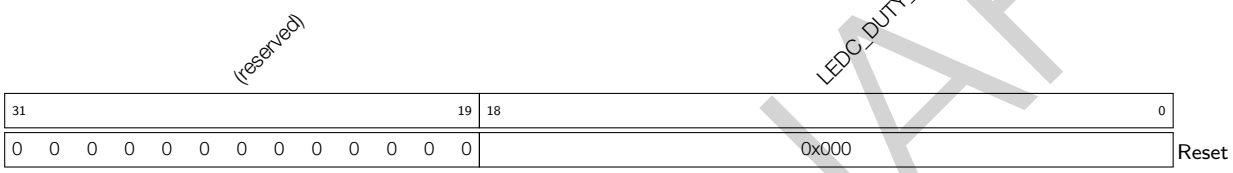
1：强制开启寄存器时钟。0：仅在应用写寄存器时支持时钟。(R/W)

Register 15.4. LEDC_CH*n*_HPOINT_REG (*n*: 0-5) (0x0004+20n*)**



LEDC_HPOINT_CH*n* 该通道所选定时器计数值达到该字段的值时，输出信号翻转为高电平。(R/W)

Register 15.5. LEDC_CH*n*_DUTY_REG (*n*: 0-5) (0x0008+20n*)**



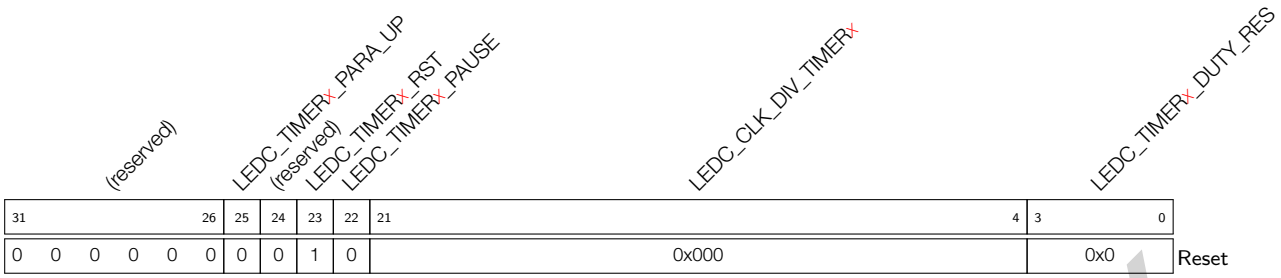
LEDC_DUTY_CH*n* 通过控制低位点改变输出信号占空比。

该通道所选定时器达到低位点时，输出信号翻转为低电平。(R/W)

Register 15.6. LEDC_CH*n*_DUTY_R_REG (*n*: 0-5) (0x0010+20n*)**

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

Register 15.7. LEDC_TIMERx_CONF_REG (x: 0-3) (0x00A0+8*x)



LEDC_TIMERx_DUTY_RES 用于控制定时器 x 计数器的计数范围。(R/W)

LEDC_CLK_DIV_TIMERx 用于配置定时器 x 分频器的分频系数。

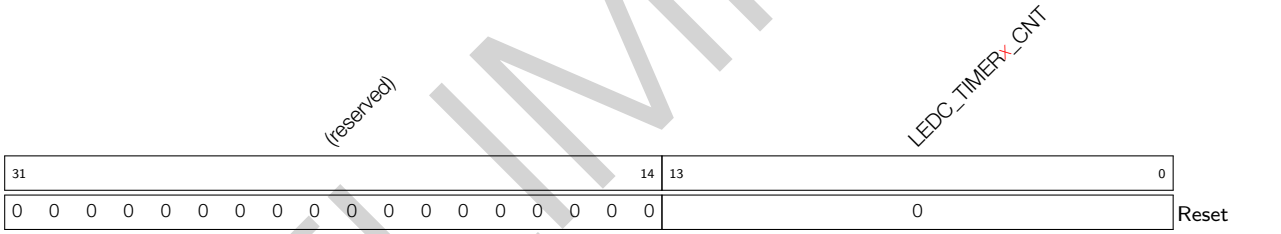
低 8 位为小数部分。(R/W)

LEDC_TIMERx_PAUSE 用于暂停定时器 x 的计数器。(R/W)

LEDC_TIMERx_RST 用于复位定时器 x。复位后计数器为 0。(R/W)

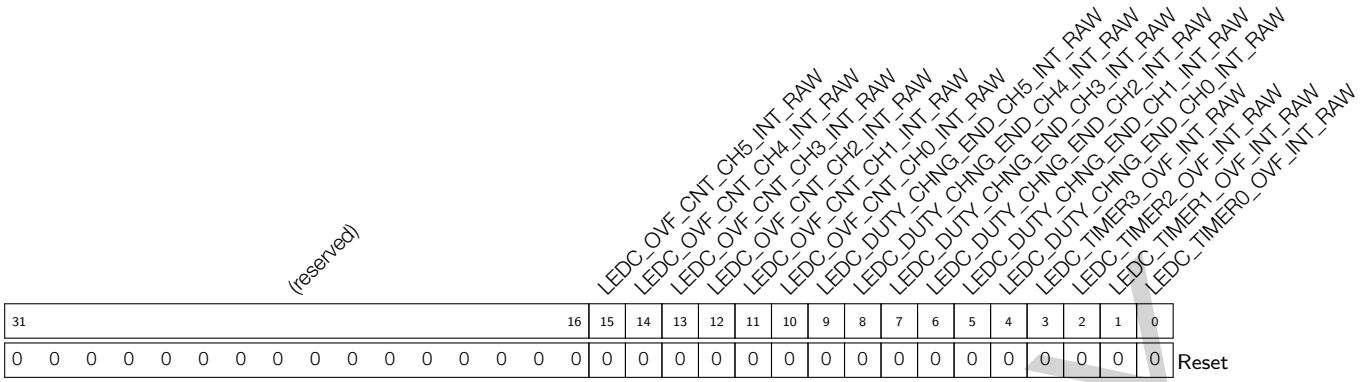
LEDC_TIMERx_PARA_UP 置位此位，更新 LEDC_CLK_DIV_TIMERx 和 LEDC_TIMERx_DUTY_RES。(WT)

Register 15.8. LEDC_TIMERx_VALUE_REG (x: 0-3) (0x00A4+8*x)



LEDC_TIMERx_CNT 存储定时器 x 的当前计数器值 (RO)

Register 15.9. LEDC_INT_RAW_REG (0x00C0)

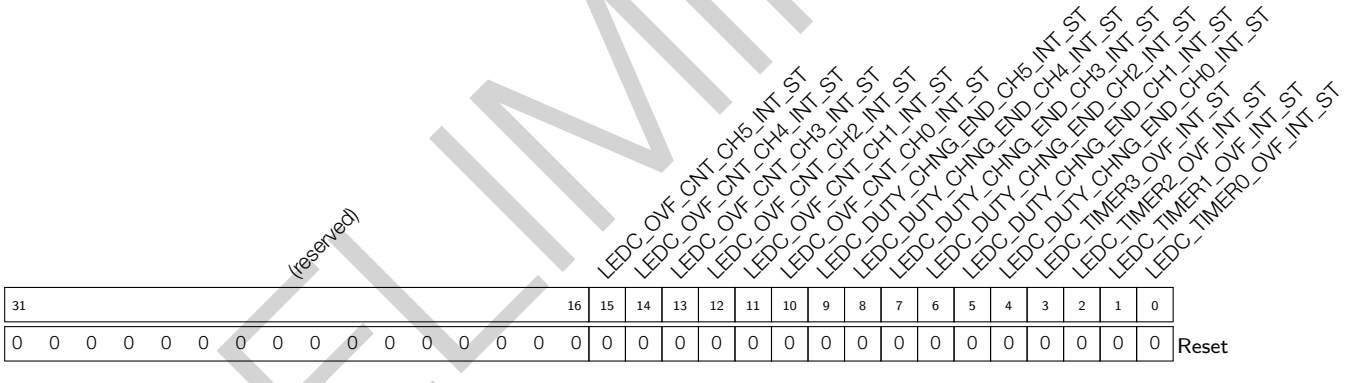


LEDC_TIMER*x*_OVF_INT_RAW 定时器 *x* 达到最大计数值时触发中断。(R/WTC/SS)

LEDC_DUTY_CHNG_END_CH*n*_INT_RAW 通道 *n* 的原始中断位。占空比渐变结束时触发。(R/WTC/SS)

LEDC_OVF_CNT_CH*n*_INT_RAW 通道 *n* 的原始中断位。ovf_cnt 达到 LEDC_OVF_NUM_CH*n* 的值时触发。(R/WTC/SS)

Register 15.10. LEDC_INT_ST_REG (0x00C4)

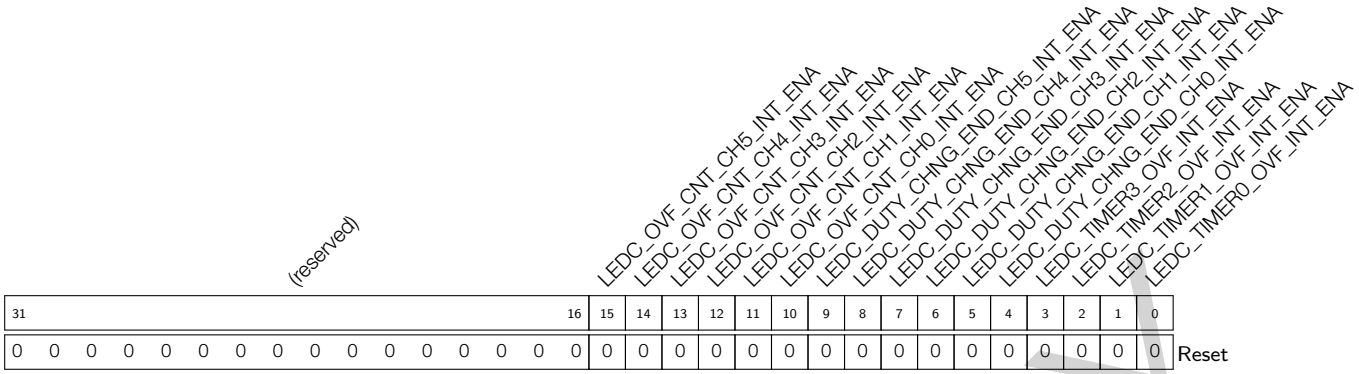


LEDC_TIMER*x*_OVF_INT_ST LEDC_TIMER*x*_OVF_INT_ENA 置 1 时，LEDC_TIMER*x*_OVF_INT 中断的屏蔽中断状态位。(RO)

LEDC_DUTY_CHNG_END_CH*n*_INT_ST LEDC_DUTY_CHNG_END_CH*n*_INT_ENA 置 1 时，LEDC_DUTY_CHNG_END_CH*n*_INT 中断的屏蔽中断状态位。(RO)

LEDC_OVF_CNT_CH*n*_INT_ST LEDC_OVF_CNT_CH*n*_INT_ENA 置 1 时，LEDC_OVF_CNT_CH*n*_INT 中断的屏蔽中断状态位。(RO)

Register 15.11. LEDC_INT_ENA_REG (0x00C8)

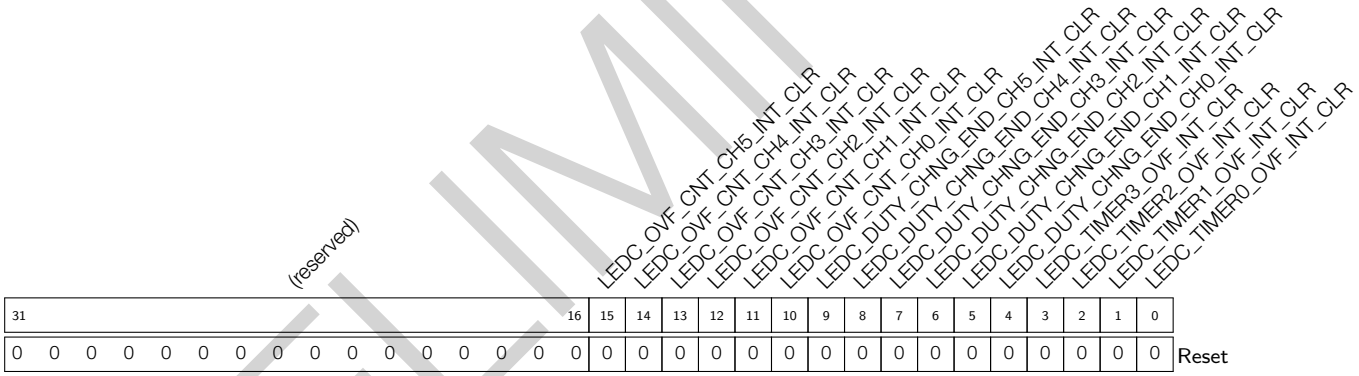


LEDC_TIMER*x*_OVF_INT_ENA LEDC_TIMER*x*_OVF_INT 中断的使能位。(R/W)

LEDC_DUTY_CHNG_END_CH*n*_INT_ENA LEDC_DUTY_CHNG_END_CH*n*_INT 中断的使能位。(R/W)

LEDC_OVF_CNT_CH*n*_INT_ENA LEDC_OVF_CNT_CH*n*_INT 中断的使能位。(R/W)

Register 15.12. LEDC_INT_CLR_REG (0x00CC)

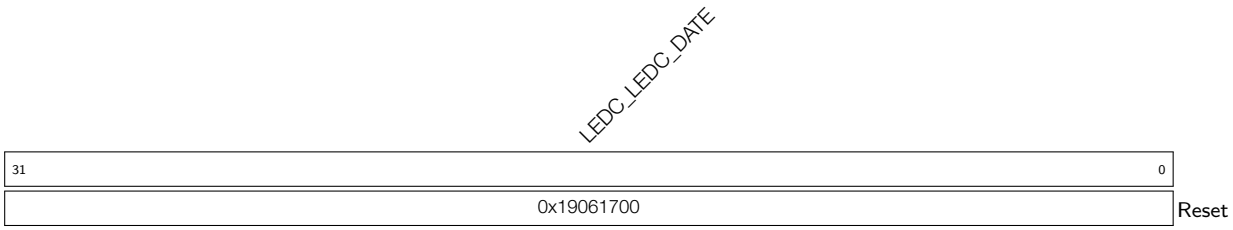


LEDC_TIMER*x*_OVF_INT_CLR 置位此位，清除 LEDC_TIMER*x*_OVF_INT 中断。(WT)

LEDC_DUTY_CHNG_END_CH*n*_INT_CLR 置位此位，清除 LEDC_DUTY_CHNG_END_CH*n*_INT 中断。(WT)

LEDC_OVF_CNT_CH*n*_INT_CLR 置位此位，清除 LEDC_OVF_CNT_CH*n*_INT 中断。(WT)

Register 15.13. LEDC_DATE_REG (0x00FC)



LEDC_LEDC_DATE 版本控制寄存器。(R/W)

词汇列表

外设相关词汇

AES	AES 加速器
BOOTCTRL	芯片 Boot 控制
DS	数字签名
DMA	DMA 控制器
eFuse	eFuse 控制器
HMAC	HMAC 加速器
I2C	I2C 控制器
I2S	I2S 控制器
LEDC	LED 控制 PWM
MCPWM	电机控制 PWM
PCNT	脉冲计数器控制器
RMT	红外遥控
RNG	随机数生成器
RSA	RSA 加速器
SDHOST	SD/MMC 主机控制器
SHA	SHA 加速器
SPI	SPI 控制器
SYSTIMER	系统定时器
TIMG	定时器组
TWAI	双线汽车接口
UART	UART 控制器
ULP 协处理器	超低功耗协处理器
USB OTG	USB On-The-Go
WDT	看门狗定时器

寄存器相关词汇

ISO	隔离。当模块断电时，其输出的引脚将处于未知状态（某些中间电压）。“ISO”寄存器将使其输出引脚隔离在一个确定的电压，从而不会影响其他未掉电的工作模块的状态。
NMI	不可屏蔽中断。
REG	寄存器。
R/W	读/写，软件可读写这些位。
RO	只读，软件只能读这些位。
SYSREG	系统寄存器。
WO	只写，软件只能写这些位。

修订历史

日期	版本	发布说明
2021-04-07	V0.1	首次发布
2021-05-27	V0.2	新增以下章节： • 章节 4 *eFuse 控制器 (EFUSE)* • 章节 13 *UART 控制器 (UART)* • 章节 8 *定时器组 (TIMG)* • 章节 2 *通用 DMA 控制器 (GDMA)* • 章节 15 *LED PWM 控制器 (LEDC)* 更新章节 5 *IO MUX 和 GPIO 交换矩阵 (GPIO, IO MUX)* 调整章节顺序

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。