



32-Bit Arm® Cortex®-M33 BLE 单片机

HT32F67575 开发使用手册

版本: V1.00 日期: 2024-03-13

www.holtek.com

目录

1 电源管理控制单元 (PMU).....8

 简介 8

 特性 9

 功能描述 9

 电源管理 9

 低功耗管理 9

 PMU API 12

 PMU 中断 API 12

 PMU 电源 API 14

 PMU 时钟 API 20

 PMU 范例 27

 PMU 范例代码 27

2 通用 I/O (GPIO).....28

 简介 28

 特性 28

 功能描述 28

 输入模式 29

 输出模式 29

 驱动能力 29

 功能复用 29

 中断功能 29

 唤醒功能 30

 GPIO API 30

 GPIO ROM API 30

 SW 端口 API 函数 37

 IRAPI 函数 39

 DCXO 引脚 API 函数 41

 GPIO 范例 42

 输入范例代码 42

 输出范例代码 42

 GPIO 中断范例代码 42

 GPIO 唤醒范例代码 43

3 通用定时器 (GPTM).....44

 简介 44

 特性 44

 功能描述 44

 计数器模式 (CNT 模式) 44

 捕捉模式 46

 PWM 模式 47

GPTM API.....	48
定时器功能 API.....	48
PWM 功能 API.....	58
定时器捕捉功能	62
GPTM 范例.....	66
计数器模式范例代码	66
PWM 模式范例代码.....	67
4 系统节拍定时器 (STIM).....	68
简介	68
特性.....	68
功能描述.....	68
时钟精准度和预分频	69
计数器、节拍、溢出和比较	69
STIM API.....	70
STIM 中断 API.....	70
STIM 唤醒 & 状态 API.....	72
STIM 计数器 & 比较	73
STIM 范例代码	76
STIM 唤醒 & 中断范例代码	76
STIM 溢出中断范例代码.....	77
5 实时时钟 (RTC)	78
简介	78
特性.....	78
功能描述.....	78
RTC 电源结构.....	79
时钟精准度和预分频	79
计数器、节拍、溢出和比较	80
RTC API	80
RCT 配置 API.....	80
RTC 中断 API	82
RTC 状态 API.....	84
RTC 计数器 & 比较.....	85
RTC 范例代码	88
RTC 开始工作范例代码.....	88
RTC 中断范例代码.....	88
唤醒范例代码	89
6 通用异步收发器 (UART).....	90
简介	90
特性.....	90
功能描述	90

UART API	90
UART 配置 API	90
UART 中断 API	94
UART FIFO API	95
UART RX/TX API	97
UART 范例代码	98
使用 UART1 的应用范例	98
7 IR 模块	101
简介	101
特性	101
功能描述	101
IR 编码	101
IR 解码	102
IR 模块 API	102
IR 编码 HW API	102
IR 编码 HAL API	105
IR 解码 HW API	107
IR 解码 HAL API	109
IR 模块范例	111
IR NEC 编码范例	111
IR 解码范例	112
8 RTX RTOS	114
关于 RTX RTOS	114
什么是 RTX RTOS	114
通用 RTOS 接口	114
功能概述	116
RTX V5 实现	116
创建 RTX5 工程	117
工作原理	119
配置 RTX V5	121
RTX-RTOS 范例	126
启动 RTOS	128
创建线程	129
内存管理	129
多实例	130
可连接线程	130
时间管理	130
空闲线程	131
线程间通信	132
HT32F67575 中的 RTX-RTOS	139
启动线程	139
创建 LLC 线程	140

创建 LLC 消息队列.....	140
发送消息到 LLC 线程.....	141
9 2nd-Boot.....	142
存储器映射	142
2nd-Boot 代码.....	143
OTA 流程.....	143
初始化缓存读取.....	144
跳转到 SDK.....	145

表列表

表 1. LUT 表 11

表 2. 查找表事件 ID 11

表 3. GPTM 模式选择 45

表 4. 时间间隔 69

表 5. CMSIS 包目录 114

表 6. CMSIS RTOS 函数 116

表 7. Cortex®-M 中断 118

表 8. 系统配置 122

表 9. 线程配置 123

表 10. 定时器配置 124

表 11. 事件配置 124

表 12. 互斥锁配置 125

表 13. 信号量配置 125

表 14. 内存池配置 126

表 15. 消息队列配置 126

表 16. 线程块内容 128

表 17. 线程块内容 131

表 18. 线程标志选项 132

表 19. 互斥锁 137

表 20. 存储器映射 142

表
列
表

图列表

图 1. PMU 方框图 8

图 2. GPIO 基本结构图 28

图 3. GPTM 计数器寄存器 44

图 4. GPTM 预分频器 45

图 5. GPTM 工作模式时序图 45

图 6. STIM 参考设计原理图 68

图 7. 计数器触发溢出 69

图 8. RTC 参考设计原理图 78

图 9. RTC 电源参考设计原理图 79

图 10. 计数器工作流程 80

图 11. 计数器触发溢出 80

图 12. IR 编码 101

图 13. IR 解码 102

图 14. 引导码 111

图 15. Bit 说明 112

图 16. 重复码 112

图 17. CMSIS-RTOS API 结构 115

图 18. 创建 RTX5 工程 117

图 19. RTX5 工程文件结构 117

图 20. 添加模板文件 118

图 21. 线程调度和中断执行 120

图 22. 配置向导界面的 RTX_Config.h 121

图 23. RTX_Config.h: 系统配置 122

图 24. RTX_Config.h: 线程配置 122

图 25. RTX_Config.h: 定时器配置 124

图 26. RTX_Config.h: 事件标志配置 124

图 27. RTX_Config.h: 互斥锁配置 125

图 28. RTX_Config.h: 信号量配置 125

图 29. RTX_Config.h: 内存池配置 125

图 30. RTX_Config.h: 消息队列配置 126

图 31. RTOS 内核 127

图 32. 启动流程 142

图 33. 2nd-Boot 代码流程 143

图 34. Flash 存储器映射 143

图 35. 2nd-Boot 的 OTA 流程 144

图 36. 初始化缓存 144

1 电源管理控制单元 (PMU)

简介

HT32F67575 单片机具有完全集成的电源管理控制单元 (PMU)，包括一个具有一个输出的单电感单输出 (SISO) DC-DC 转换器，以及用于系统不同电源轨的多个 LDO。

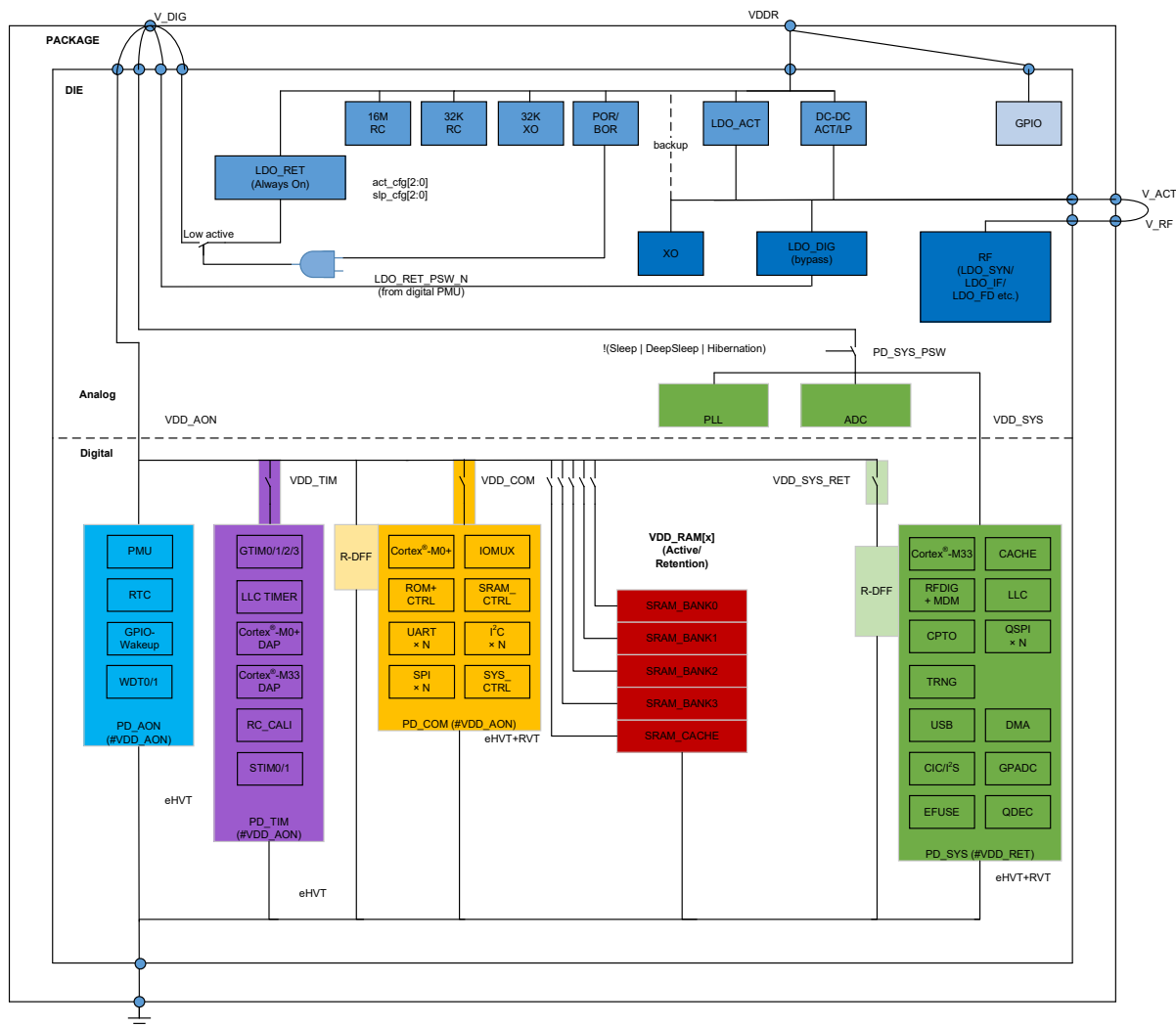


图 1. PMU 方框图

特性

- 支持多个 LDO：
 - LDO_ACT，可配置 0.95 V ~ 1.30 V
 - LDO_DIG，可配置 0.9 V ~ 1.25 V
 - LDO_RET，可配置 0.75 V ~ 1.1 V
 - LDO_1V8 输出到 GPIO 可选
- 支持 SISO DC-DC 转换器：
 - DCDC_ACT，可配置 0.95 V ~ 1.30 V
 - DCDC_RET，可配置 0.65 V ~ 1.05 V
 - DCDC_RET 时钟，可配置 1/2/4.../128 分频
- 多个时钟
 - 系统时钟：可配置 RC_HCLK (RC16M/RC48M)、DCXO16M、PLL64M
 - 低功耗时钟：可配置 RC32K、DXO32K
- 四种系统工作模式：
 - 活动模式：系统在正常模式下工作，可以配置不同的电源和系统时钟 (RC_HCLK/DCXO_HCLK/PLL)
 - 睡眠模式：系统时钟关闭，SRAM 保持保存功能
 - 深度睡眠模式：SRAM 关闭，低功耗时钟 (32K) 保持保存功能
 - 休眠模式：低功耗时钟 (32K) 关闭
- 支持在系统进入睡眠状态时可配置每个 SRAM 块保存功能
 - 支持 16 个查找表 (LUT) 用于唤醒
 - 支持电源监控单元、POR 和 BOR

功能描述

电源管理

该单片机支持两种电源模式：LDO 和 DC-DC。系统默认工作在 LDO 模式。电源模式切换只有在芯片唤醒后才会生效。DC-DC 模式需要外接电感，但系统功耗比 LDO 模式低。

低功耗管理

四种工作模式下各模块的工作情况如下：

	活动模式	睡眠模式	深度睡眠模式	休眠模式
CPU	活动	Off	Off	Off
FLASH	On	可用	Off	Off
SRAM	On	On	Off	Off
RADIO	On	Off	Off	Off
SRAM 保存	全部	部分	No	No
16M RC / DCXO	On	Off	Off	Off
32K RC / DCXO	On	On	On	Off
外设	可用	可用	Off	Off
从 RTC 唤醒	可用	可用	可用	Off
从 GPIO 唤醒	可用	可用	可用	可用

在活动模式下，应用程序 Arm® Cortex®-M33 或 Arm® Cortex®-M0+ CPU 正在主动执行代码。活动模式提供处理器和当前启用的所有外设的正常操作。系统时钟可以是任意可用的时钟源。

在睡眠模式下，可以对所有活动的外设进行时钟设置。但是应用程序的 CPU 核心和存储器没有时钟，也不执行任何代码。任何中断事件都会使处理器回到活动模式。

在深度睡眠模式下，只有始终开启的域处于活动状态。需要外部唤醒事件 RTC 事件才能将单片机恢复到活动模式。

在休眠模式下，单片机完全关闭，包括始终开启的域。I/O 与进入休眠模式之前的值一起锁定。任何 I/O 引脚上的状态变化(定义为从休眠引脚唤醒)都会唤醒单片机并用作复位触发器。

进入低功耗模式

由于 HT32F67575 是双核单片机，Cortex®-M33 和 Cortex®-M0+ 必须在芯片开始进入睡眠过程之前进入睡眠状态。Cortex®-M0+ 默认配置为低功耗模式。用户主要在 Cortex®-M33 中配置低功耗。

在 main.c 中，调用 lpwr_ctrl_init 函数进行低功耗初始化，如下所示：

```
lpwr_ctrl_init(enMode, lpwr_before_sleep, lpwr_after_wakeup);
```

enMode 指的是 EN_LPWR_MODE_SEL_T，如下所示：

```
typedef enum
{
    LPWR_MODE_ACTIVE = 0x00,
    LPWR_MODE_IDLE = 0x01,
    LPWR_MODE_SLEEP = 0x02, /* SRAM keep retention, system clock will power
    down. */
    LPWR_MODE_DEEPSLEEP = 0x03, /* SRAM will power down, LCLK clock will keep
    working (shutdown with 32K). */
    LPWR_MODE_HIBERNATION = 0x04, /* SRAM will power down, LCLK clock will turn
    off(shutdown without 32K). */
}EN_LPWR_MODE_SEL_T;
```

注：1. 如果 enMode 设置为 LPWR_MODE_ACTIVE，CPU 将保持活动，系统时钟不会关闭。

2. 如果 enMode 设置为 LPWR_MODE_IDLE，CPU 将保持空闲模式，系统时钟不会关闭。

■ lpwr_before_sleep 是一个回调函数，用于检查系统是否可以进入睡眠模式。用户可以在这里添加应用程序代码来确定系统是否可以进入睡眠模式。当返回 0 时，系统不会进入睡眠模式，否则系统将进入睡眠模式。

■ lpwr_after_wakeup 是一个回调函数，是系统从睡眠模式唤醒后第一个执行的函数。用户可以在这里添加应用程序代码来重新初始化外设。

■ 当 lpwr_ctrl_goto_sleep() 被执行后，系统将进入 enMode 定义的低功耗模式。

系统唤醒

该单片机的 PMU 模块具有 13-bit (1 word) 查找表 (LUT)。它最多可以定义 16 个事件，当事件发生时将系统从低功耗模式 (睡眠 / 深度睡眠或休眠模式) 唤醒。

查找表 (LUT) 由 Cortex®-M33 在冷启动时初始化。指示 PDC 根据触发源激活哪些数字电源域。

表 1. LUT 表

	位	功能			
触发器	0	如果是 0x00 则为 P0 GPIO	如果是 0x01 则为 P1 GPIO	如果是 0x02 则为 P2 GPIO	如果是 0x03 则为 外设
	1				
	2	Pin_ID	Pin_ID	Pin_ID	Periph_ID0
	3	Pin_ID	Pin_ID	Pin_ID	Periph_ID1
	4	Pin_ID	Pin_ID	Pin_ID	Periph_ID2
	5	Pin_ID	Pin_ID	Pin_ID	Periph_ID3
	6	Pin_ID	Pin_ID	Pin_ID	Periph_ID4
做什么	7	使能 PD_SYS			
	8	使能 DCXO16M			
	9	使能向 Cortex®-M0+ 发送中断			
	10	使能向 Cortex®-M33 发送中断			

LUT 的深度为 16 层，因此系统支持多达 16 种不同的唤醒配置，但可以通过应用软件动态更改。

当触发器根据每个 LUT 入口的 bit 0 和 bit 1 的值来触发时，会评估四种不同的触发类型：

- 如果 Bit[1:0] = 00，则是来自 P0 的 GPIO 翻转。Bit[6:2] 包含被触发的引脚编号；
- 如果 Bit[1:0] = 01，则是来自 P1 的 GPIO 翻转。Bit[6:2] 包含被触发的引脚编号；
- 如果 Bit[1:0] = 10，则是来自 P2 的 GPIO 翻转。Bit[6:2] 包含被触发的引脚编号；
- 如果 Bit[1:0] = 11，则是来某个外设的触发。Bit[6:2] 按以下方式定义外设：

表 2. 查找表事件 ID

ID (十进制)	外设
0	LUT_TRIG_ID_RTC_CH0
1	LUT_TRIG_ID_RTC_CH1
2	LUT_TRIG_ID_RTC_CH2
3	LUT_TRIG_ID_RTC_CH3
4	LUT_TRIG_ID_LLC
5	LUT_TRIG_ID_GTIM0
6	LUT_TRIG_ID_GTIM1
7	LUT_TRIG_ID_GTIM2
8	LUT_TRIG_ID_GTIM3
9	LUT_TRIG_ID_GPIO
10	LUT_TRIG_ID_CP_SWD
11	LUT_TRIG_ID_MP_SWD
12	LUT_TRIG_ID_USB
13	LUT_TRIG_ID_STIM0_CH0
14	LUT_TRIG_ID_STIM0_CH1
15	LUT_TRIG_ID_STIM0_CH2
16	LUT_TRIG_ID_STIM0_CH3
17	LUT_TRIG_ID_STIM1_CH0

ID (十进制)	外设
18	LUT_TRIG_ID_STIM1_CH1
19	LUT_TRIG_ID_STIM1_CH2
20	LUT_TRIG_ID_STIM1_CH3
21	LUT_TRIG_ID_WDT1

注：ID 9 表示所有 GPIO 唤醒都有去抖，只有在睡眠和深度睡眠模式下才可设置。
Bit[1:0] = 00/01/10 表示所有 GPIO 唤醒都不去抖。
Bit[10:7] 解释了如前所述的触发源在触发后需要执行的操作。更具体地说，触发源可能会有以下请求：

- 使能 DCXO16M。如果应用程序 (例如，蓝牙®低功耗操作，需要 PLL48 的 USB 操作，需要 PLL64 的高性能模式) 需要时钟精准度，则设置此位。
- 使能向 Cortex®-M0+ / Cortex®-M33 发送中断。外设 PMU 中断将发送到 Cortex®-M0+ 或 Cortex®-M33。

PMU API

PMU 中断 API

rom_hw_pmu_get_interrupt_flag

函数原型
EN_ERR_STA_T rom_hw_pmu_get_interrupt_flag (uint32_t* pu32IntMsk);

描述
通过读 PMU_LUT_INT_FLAG 寄存器来获取 PMU 中断标志 (状态)。

参数

参数	描述
pu32IntMsk	指示将读取哪个中断标志。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_clear_interrupt_flag

函数原型
EN_ERR_STA_T rom_hw_pmu_clear_interrupt_flag (uint32_t u32IntMsk);

描述
通过写 PMU_LUT_INT_CLR 寄存器来清除 PMU 中断标志 (状态)。

参数

参数	描述
u32IntMsk	指示将清除哪个标志。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_set_wakeup_source

函数原型

EN_ERR_STA_T rom_hw_pmu_set_wakeup_source (uint8_t u8LutIdx, PUM_LUT_TRIG_SEL_T enTrigSel, EN_LUT_TRIG_ID_T enLutTrigId, EN_LUT_ACT_T enLutAction);

描述

选择 PMU 唤醒源。

参数

参数	描述
u8LutIdx	LUT 索引，应小于 16。
enTrigSel	LUT 触发器选择，@ ref PUM_LUT_TRIG_SEL_T。
enLutTrigId	LUT 触发器 ID 选择，@ ref EN_LUT_TRIG_ID_T。
enLutAction	LUT 动作选择，@ ref EN_LUT_ACT_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_set_gpio_wakeup_source

函数原型

EN_ERR_STA_T rom_hw_pmu_set_gpio_wakeup_source (uint8_t u8LutIdx, PUM_LUT_TRIG_SEL_T enTrigSel, uint32_t u32Pin, EN_LUT_ACT_T enLutAction)

描述

设置 GPIO 从低功耗模式唤醒系统而不去抖。

参数

参数	描述
u8LutIdx	LUT 索引，需小于 16。
enTrigSel	LUT 触发器配置，必须选择 PMU_LUT_TRIG_GPIOA、PMU_LUT_TRIG_GPIOB、PMU_LUT_TRIG_GPIOC。
u32Pin	需要配置哪个引脚，一次只能配置一个引脚。
u8LutAction	LUT 动作配置，@ ref EN_LUT_ACT_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_set_sram_block_ret

函数原型
EN_ERR_STA_T rom_hw_pmu_set_sram_block_ret (uint32_t u32SramBlock);

描述
当系统进入低功耗模式时，配置 SRAM 存储器保持保存功能。

参数

参数	描述
u32SramBlock	将保持保存功能的 RAM 块。每个位对应一个 SRAM 块。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_clr_sram_block_ret

函数原型
EN_ERR_STA_T rom_hw_pmu_clr_sram_block_ret (uint32_t u32SramBlock);

描述
当系统进入睡眠模式时，清除指示的 SRAM 块保存功能。

参数

参数	描述
u32SramBlock	将关闭的 RAM 块。每个位对应一个 SRAM 块。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

PMU 电源 API

rom_hw_pmu_enable_ldo_act_output

函数原型
EN_ERR_STA_T rom_hw_pmu_enable_ldo_act_output (void);

描述
使能 LDO_ATC 输出。
当系统工作在活动模式时，LDO_ACT 输出为 CPU 供电。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_disable_ldo_act_output

EN_ERR_STA_T rom_hw_pmu_disable_ldo_act_output (void);

描述
除能 LDO_ATC 输出。
参见 rom_hw_pmu_enable_ldo_act_output。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_set_ldo_act_voltage

函数原型

EN_ERR_STA_T rom_hw_pmu_set_ldo_act_voltage (EN_LDO_ACT_VOLT_T enVolt);

描述

配置 LDO_ACT 输出电压，当系统工作在活动模式时，连接到 VDD_RF 为 RF 模块供电。

参数

参数	描述
enVolt	参考 EN_LDO_ACT_VOLT_T。 范围为 950 mV ~ 1300 mV，每阶 50 mV。 注：LDO_ACT 输出电压必须比 LDO_DIG 输出电压至少高 50 mV。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_enable_ldo_1v8_output

函数原型

EN_ERR_STA_T rom_hw_pmu_enable_ldo_1v8_output (void);

描述

使能 LDO_1V8 输出。

LDO_1V8 输出在 LDO 模式下为 GPIO 模块供电。在 DC-DC 模块下必须除能。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_disable_ldo_1v8_output

函数原型

EN_ERR_STA_T rom_hw_pmu_disable_ldo_1v8_output (void);

描述

除能 LDO_1V8 输出。

参见 rom_hw_pmu_enable_ldo_1v8_output。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_enable_ldo_dig_output

函数原型

EN_ERR_STA_T rom_hw_pmu_enable_ldo_dig_output(void);

描述

使能 LDO_DIG 输出。
当系统工作在活动模式时，LDO_DIG 输出为数字模块供电。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_disable_ldo_dig_output

函数原型

EN_ERR_STA_T rom_hw_pmu_disable_ldo_dig_output(void);

描述

除能 LDO_DIG 输出。
参见 rom_hw_pmu_enable_ldo_dig_output。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_set_ldo_dig_voltage

函数原型

EN_ERR_STA_T rom_hw_pmu_set_ldo_dig_voltage(EN_LDO_DIG_VOLT_T enVolt);

描述

配置 LDO_DIG 输出电压。
参见 rom_hw_pmu_enable_ldo_dig_output。

参数

参数	描述
enVolt	LDO_DIG 定义的电压。 范围为 900 mV ~ 1100 mV，每阶 50 mV。 注：1. 当 SYS_CLK 工作在 16 MHz 以下时，LDO_DIG 输出电压可设置为 900 mV。 2. SYS_CLK 工作在 128 MHz 时，LDO_DIG 输出电压必须大于 1100 mV。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_get_ldo_dig_voltage

函数原型

EN_ERR_STA_T rom_hw_pmu_get_ldo_dig_voltage (uint8_t* pu8Volt);

描述

获取当前 LDO_DIG 输出电压配置。

参数

参数	描述
enVolt	获取 LDO 输出电压, @ ref EN_LDO_DIG_VOLT_T。

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

rom_hw_pmu_set_ldo_dig_and_ret_act_voltage

函数原型

EN_ERR_STA_T rom_hw_pmu_set_ldo_dig_and_ret_act_voltage (EN_LDO_DIG_VOLT_T enVolt);

描述

同时配置 LDO_DIG、LDO_RET 和 LDC_ACT 输出电压。

参数

参数	描述
enVolt	LDO_DIG、LDO_RET 和 LDO_ACT 输出电压, 范围为 900 mV ~ 1100 mV, 每阶 50 mV。 更多详细信息参见 LDO_DIG、LDO_RET 和 LDO_ACT。

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

rom_hw_pmu_set_ldo_ret_act_voltage

函数原型

EN_ERR_STA_T rom_hw_pmu_set_ldo_ret_act_voltage (EN_LDO_RET_VOLT_T enVolt);

描述

配置系统工作在活动模式时的 LDO_RET 输出电压。

参数

参数	描述
enVolt	系统工作在活动模式时的 LDO_RET 输出电压。范围为 750 mV ~ 1100 mV, 每阶 50 mV。 注: 当系统工作在活动模式时, 请保持 LDO_RET 输出与 LDO_DIG 输出电压相同。

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

rom_hw_pmu_set_ldo_ret_sleep_voltage

函数原型

EN_ERR_STA_T rom_hw_pmu_set_ldo_ret_sleep_voltage (EN_LDO_RET_VOLT_T enVolt);

描述

配置系统工作在睡眠模式时的 LDO_RET 输出电压。

参数

参数	描述
enSleepVolt	配置系统工作在睡眠模式时的 LDO_RET 输出电压。 范围为 750 mV ~ 1100 mV，每阶 50 mV。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_dcdc_init

函数原型

EN_ERR_STA_T rom_hw_pmu_dcdc_init(void);

描述

初始化 DC-DC 模块。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_enable_dcdc_act_output

函数原型

EN_ERR_STA_T rom_hw_pmu_enable_dcdc_act_output(void);

描述

当系统工作在活动模式时，使能 DC-DC 输出。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_disable_dcdc_act_output

函数原型

EN_ERR_STA_T rom_hw_pmu_disable_dcdc_act_output(void);

描述

使能指示的 DMA 控制器。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_set_dcdc_act_voltage

函数原型

EN_ERR_STA_T rom_hw_pmu_set_dcdc_act_voltage(EN_DCDC_ACT_VOLT_T enVolt);

描述

当系统工作在活动模式时，配置 DC-DC 模块的输出电压。

DC-DC 模块在 DC-DC 模式下为 RF 模块和数字模块供电。

参数

参数	描述
enVolt	DC-DC 输出电压。范围为 950 mV ~ 1300 mV，每阶 50 mV。 注：DC-DC 输出电压至少要比 LDO_DIG 输出电压高 50 mV。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_enable_dcdc_ret_output

函数原型

EN_ERR_STA_T rom_hw_pmu_enable_dcdc_ret_output(void);

描述

当系统工作在低功耗模式 (包括睡眠、深度睡眠和休眠模式) 时，使能 DC-DC 模块输出。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_disable_dcdc_ret_output

函数原型

EN_ERR_STA_T rom_hw_pmu_disable_dcdc_ret_output(void);

描述

当系统工作在低功耗模式 (包括睡眠、深度睡眠和休眠模式) 时，除能 DC-DC 模块输出。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_set_dcdc_ret_voltage

函数原型
EN_ERR_STA_T rom_hw_pmu_set_dcdc_ret_voltage (EN_DCDC_LPWR_VOLT_T enLpwrVolt);

描述
当系统工作在低功耗模式 (包括睡眠、深度睡眠和休眠模式) 时, 配置 DC-DC 模块输出电压。

参数

参数	描述
enLpwrVolt	系统工作在低功耗模式时的 DC-DC 模块输出电压。范围为 650 mV ~ 1050 mV, 每阶 50 mV。

返回	
EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。

PMU 时钟 API

rom_hw_pmu_set_dcdc_ret_clk_divisor

函数原型
EN_ERR_STA_T rom_hw_pmu_set_dcdc_ret_clk_divisor (EN_DCDC_LPWR_CLK_T enDiv);

描述
配置 DC-DC 模块来自内部 RC_CLK 的时钟分频值。当系统工作在低功耗模式 (包括睡眠、深度睡眠和休眠模式) 时, DC-DC 模块时钟来自内部 RC_CLK。

参数

参数	描述
enDiv	来自内部 RC_CLK 的分频值。

返回	
EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。

rom_hw_pmu_set_low_power_mode

函数原型
EN_ERR_STA_T rom_hw_pmu_set_low_power_mode (EN_PWR_MODE_T enMode);

描述
设置系统进入指示的低功耗模式, 包括睡眠、深度睡眠和休眠模式。

参数

参数	描述
enMode	低功耗工作模式, @ ref EN_PWR_MODE_T。

返回	
EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。

rom_hw_pmu_rc_lclk_is_enable

函数原型

bool rom_hw_pmu_rc_lclk_is_enable (void);

描述

检查内部 RC_CLK 是否使能。

参数

无

返回

status	FALSE(0) - RC_LCLK 除能; TURE(1) - RC_LCLK 使能。
--------	---

rom_hw_pmu_dcxo_lclk_is_power_on

函数原型

bool rom_hw_pmu_dcxo_lclk_is_power_on (void);

描述

检查 DCXO 低速时钟 (DCXO_LCLK) 是开启还是关闭。

参数

无

返回

status	FALSE(0) - DCXO_LCLK 开启; TURE(1) - DCXO_LCLK 关闭。
--------	---

rom_hw_pmu_dcxo_lclk_is_clk_out

函数原型

bool rom_hw_pmu_dcxo_lclk_is_clk_out (void);

描述

检查 DCXO 低速时钟 (DCXO_LCLK) 输出是否使能。

参数

无

返回

status	FALSE(0) - DCXO_LCLK 输出除能; TRUE(1) - DCXO_LCLK 输出使能。
--------	---

rom_hw_pmu_rc_hclk_is_power_on

函数原型

bool rom_hw_pmu_rc_hclk_is_power_on (void);

描述

检查 RC 高速时钟 (RC_HCLK) 是开启还是关闭。

参数

无

返回

status	FALSE(0) - RC_HCLK 关闭; TRUE(1) - RC_HCLK 开启。
--------	---

rom_hw_pmu_dcxo_hclk_is_clk_out

函数原型

bool rom_hw_pmu_dcxo_hclk_is_clk_out(void);

描述

检查 DCXO 高速时钟 (DCXO_HCLK) 输出是使能还是除能。

参数

无

返回

status	FALSE(0) - DCXO_HCLK 输出除能; TURE(1) - DCXO_HCLK 输出使能。
--------	---

rom_hw_pmu_dcxo_hclk_is_power_on

函数原型

bool rom_hw_pmu_dcxo_hclk_is_power_on(void);

描述

检查 DCXO 高速时钟 (DCXO_HCLK) 是开启还是关闭。

参数

无

返回

status	FALSE(0) - DCXO_HCLK 关闭; TRUE (1) - DCXO_HCLK 开启。
--------	--

rom_hw_pmu_rc_hclk_is_clk_out

函数原型

bool rom_hw_pmu_rc_hclk_is_clk_out(void);

描述

检查 RC 高速时钟 (RC_HCLK) 输出是使能还是除能。

参数

无

返回

status	FALSE(0) - RC_HCLK 输出除能; TURE(1) - RC_HCLK 输出使能。
--------	---

rom_hw_pmu_pll_clk_is_power_on

函数原型

bool rom_hw_pmu_pll_clk_is_power_on(void);

描述

检查 PLL 时钟 (PLL_CLK) 是开启还是关闭。

参数

无

返回

status	FALSE(0) - PLL_CLK 关闭; TURE(1) - PLL_CLK 开启。
--------	---

rom_hw_pmu_pll_clk_is_clk_out

函数原型

bool rom_hw_pmu_pll_clk_is_clk_out(void);

描述

检查 PLL 时钟 (PLL_CLK) 输出是使能还是除能。

参数

无

返回

status	FALSE(0) - PLL_CLK 输出除能; TRUE (1) - PLL_CLK 输出使能。
--------	--

rom_hw_pmu_turn_clk_power_on

函数原型

EN_ERR_STA_T rom_hw_pmu_turn_clk_power_on (EN_CLK_POWER_CTRL_T enCLK,
uint32_t u32TimeUs);

描述

开启指示的时钟电源。

参数

参数	描述
enCLK	指示将开启哪个时钟电源，@ ref EN_CLK_POWER_CTRL_T。
u32TimeUs	时钟的设置时间，单位：μs。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_turn_clk_power_off

函数原型

EN_ERR_STA_T rom_hw_pmu_turn_clk_power_off (EN_CLK_POWER_CTRL_T enCLK);

描述

关闭指示的时钟电源。

参数

参数	描述
enCLK	指示将关闭哪个时钟电源，@ ref EN_CLK_POWER_CTRL_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_enable_clk_output

函数原型
EN_ERR_STA_T rom_hw_pmu_enable_clk_output(EN_CLK_OUT_CTRL_T enCLK);

描述
使能指示的时钟输出。

参数

参数	描述
enCLK	指示将使能哪个时钟输出，@ ref EN_CLK_OUT_CTRL_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_disable_clk_output

函数原型
EN_ERR_STA_T rom_hw_pmu_disable_clk_output(EN_CLK_OUT_CTRL_T enCLK);

描述
除能指示的时钟输出。

参数

参数	描述
enCLK	指示将除能哪个时钟输出，@ ref EN_CLK_OUT_CTRL_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_dcxo_lclk_cfg_is_valid

函数原型
bool rom_hw_pmu_dcxo_lclk_cfg_is_valid(stDCXOBuf_t* pstBuf, stDCXOParam_t* pstParam);

描述
检查 DCXO 低速时钟 (DCXO_LCLK) 配置是否有效。

参数

参数	描述
pstBuf	指向 stDCXOBuf_t 结构的指针。
pstParam	指向 stDCXOParam_t 结构的指针。

返回

status	FALSE(0) - DCXO_LCLK 配置无效； TURE(1) - DCXO_LCLK 配置有效。
--------	---

rom_hw_pmu_set_dcxo_lclk_param

函数原型
EN_ERR_STA_T rom_hw_pmu_set_dcxo_lclk_param (stDCXOBuf_t* pstBuf, stDCXOParam_t* pstParam);

描述
配置 DCXO 低速时钟 (DCXO_LCLK, 32 kHz) 参数，包括逆变器、电流和负载电容。

参数

参数	描述
pstBuf	指向 stDCXOBuf_t 结构的指针，此结构包含以下成员： u8PosBuf, u8NegBuf: 逆变器数量。取值范围为 [0:7]。
pstParam	指向 stDCXOParam_t 结构的指针，此结构包含以下成员： u8Ib: DCXO 电流。取值范围为 [0:7]。 u8Ngm: 与振荡器有源部分并联的逆变器数量。取值范围为 [0:7]。建议 cfg 为 2~4。 u8Cap: DCXO 负载电容。单位: 0.1 pF，范围为 [0:255]，即 3.0 pF ~ 28.5 pF。

返回	
EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

rom_hw_pmu_set_rc_lclk_tune

函数原型
EN_ERR_STA_T rom_hw_pmu_set_rc_lclk_tune (uint8_t u8Val);

描述
配置 RC_LCLK 校准值，对 RC_LCLK 进行校准。

参数

参数	描述
u8Val	RC_LCLK 调整值，范围为 [0:255]。

返回	
EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

rom_hw_pmu_set_rc_hclk_tune

函数原型
EN_ERR_STA_T rom_hw_pmu_set_rc_hclk_tune (uint8_t u8Val);

描述
配置校准值，对 RC_HCLK 进行校准。

参数

参数	描述
u8Val	RC_HCLK 调整值，范围为 [0:127]。

返回	
EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

rom_hw_pmu_set_rc_hclk_sync_out

函数原型

EN_ERR_STA_T rom_hw_pmu_set_rc_hclk_sync_out (EN_RC_HCLK_SYNC_OUT_MODE_T enMode);

描述

开启 RC_HCLK 时，使能或除能 RC_HCLK 时钟输出。

参数

参数	描述
enMode	RC_HCLK 时钟输出模式。 b0 – 开启 RC_HCLK 后延迟 0.5 ~ 1.5 个 32K 时钟周期再输出 RC_HCLK。 b1 – 开启 RC_HCLK 时立即输出 RC_HCLK。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_sel_dcxo_hclk_pwr

函数原型

EN_ERR_STA_T rom_hw_pmu_sel_dcxo_hclk_pwr (EN_DCXO_HCLK_PWR_T enPwr);

描述

配置 DCXO 高速时钟 (DCXO_HCLK) 电源为 DC-DC 或 VDDR。

参数

参数	描述
enPwr	DCXO_HCLK 电源选择，@ ref EN_DCXO_HCLK_PWR_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_set_dcxo_hclk_stable_time

函数原型

EN_ERR_STA_T rom_hw_pmu_set_dcxo_hclk_stable_time (uint8_t u8Time);

描述

配置 DCXO 高速时钟 (DCXO_HCLK) 唤醒后的稳定时间，单位为 31.25 μ s。

参数

参数	描述
u8Time	稳定时间，范围为 [00:255]，单位为 31.25 μ s。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_pmu_set_dcxo_hclk_param

函数原型

EN_ERR_STA_T rom_hw_pmu_set_dcxo_hclk_param (stDCXOParam_t* pstParam);

描述

配置 DCXO 高速时钟 (DCXO_HCLK) 电流和负载电容。

参数	
参数	描述
pstParam	指向 stDCXOParam_t 结构的指针，此结构包含以下成员： u8Ib: DCXO 电流。取值范围为 [0:7]。 u8Ngm: 与振荡器有源部分并联的逆变器数量。取值范围为 [0:7]。建议 cfg 为 2 ~ 4。 u8Cap: DCXO 负载电容。单位: 0.1 pF，范围为 [0:255]，即 3.0 pF ~ 28.5 pF。
返回	
EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

rom_hw_pmu_sel_usb_phy_iso

函数原型
EN_ERR_STA_T rom_hw_pmu_sel_usb_phy_iso (EN_USB_PHY_RX_ISO_CFG_T enIso);

描述
选择 USB PHY ISO。

参数	
参数	描述
enIso	USB PHY ISO, @ ref EN_USB_PHY_RX_ISO_CFG_T。
返回	
EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

PMU 范例

PMU 范例代码

```
/**
 * @ brief System power manage.
 * @ param enSel: Select DCDC or LDO, @ ref EN_PMU_POWER_SEL_T.
 */
static void system_power_init(EN_PMU_PWR_SEL_T enSel)
{
    // Set ldo_act voltage.
    rom_hw_pmu_set_ldo_act_voltage(EN_LDO_ACT_1200mV);
    // Init dcdc configuration and set dcdc_act voltage.
    patch_hw_pmu_dcdc_init();
    rom_hw_pmu_set_dcdc_act_voltage(EN_DCDC_ACT_VOLT_1200mV);
    // Set ldo_dig and ldo_ret voltage.
    rom_hw_pmu_set_ldo_dig_and_ret_act_voltage(EN_LDO_DIG_1100mV);
    rom_hw_pmu_set_ldo_ret_sleep_voltage(EN_LDO_RET_1100mV);
    // Power selection. It will be valid after the system
    // gets into sleep, default is ldo mode.
    rom_hal_pmu_sel_power_act_out_mode(enSel);
}
```

2 通用 I/O (GPIO)

简介

GPIO 是通用输入 / 输出端口 (General Purpose Input and Output port) 的简称，可以通过软件控制其输入和输出。

特性

- 支持多达 22 个 I/O 引脚
- 输出状态：浮空 + 上拉 / 下拉
- 输入状态：施密特 / CMOS + 上拉 / 下拉
- 每个 I/O 支持单独的高 / 低电平设置
- 每个 I/O 支持读取当前电平状态 / 输出状态

功能描述

该单片机中的任何 GPIO 都支持输入和输出功能的软件配置、其他外设接口的多路复用配置、外部中断功能的配置，并可配置 CPU 睡眠唤醒。

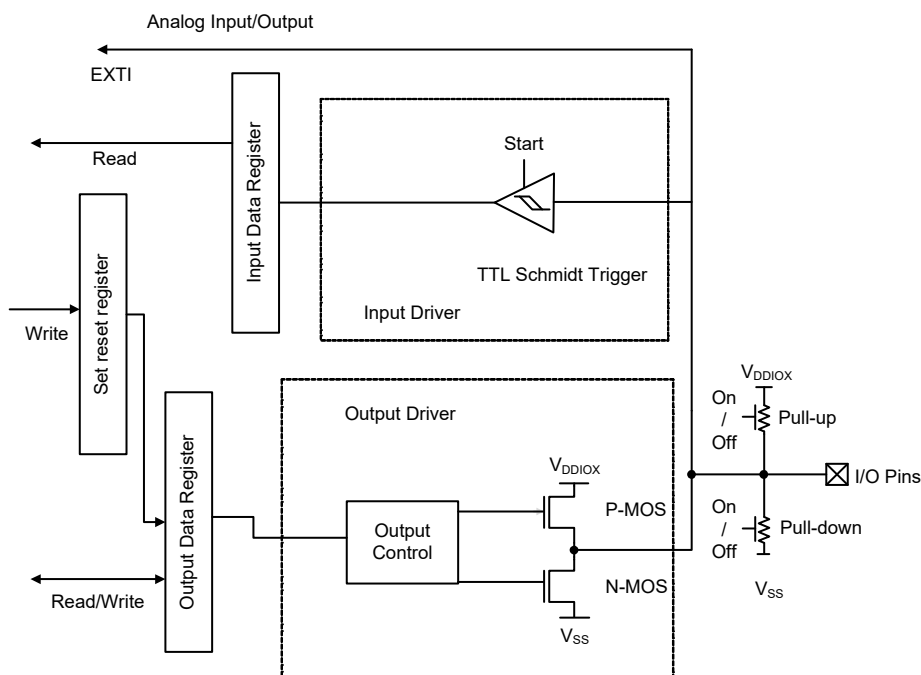


图 2. GPIO 基本结构图

如上图所示，该模块的功能主要是通过不同的输入和输出配置来实现的。该单片机的每个 I/O 口都可以通过软件配置相应的 I/O 模式，主要有以下几种配置模式：

输入	输入浮空	输入上拉	输入下拉	模拟输入	施密特 TTL 输入	施密特 CMOS 输入
输出	输出上拉	输出下拉				

输入模式

该单片机的每个 I/O 都可以用作输入，当 I/O 端口配置为输入时：

- 输出缓冲器关闭
- 施密特触发器开启
- 上拉电阻和下拉电阻根据寄存器状态开启或关闭
- 输入数据寄存器每 2 个 APB 时钟周期对 I/O 引脚上的数据进行采样
- 通过输入数据寄存器获取 I/O 状态

输出模式

该单片机的每个 I/O 输出不同的电平并读取输出状态，当 I/O 端口配置为输出时：

- 输出缓冲器开启
 - 当需要高输出电平时，PMOS 导通
 - 当需要低输出电平时，NMOS 导通
- 上拉电阻和下拉电阻根据寄存器状态开启或关闭
- 对输出状态寄存器进行读访问，以获取最后一次写入的值
- I/O 默认状态为高电阻下拉 (特殊要求的 I/O 除外，如 SWD、CLOCK 等)

驱动能力

单片机 GPIO 驱动能力的两个重要指标是拉电流 (源电流) 和填充电流 (灌电流)。此外，单个 I/O 的驱动能力和总 I/O 驱动能力也是衡量芯片驱动能力的重要指标。

- 拉电流 (源电流)：相对于单片机从外部输出电流的能力就是拉电流能力。例如，在单片机 I/O 和 GND 之间连接一个 1 K 负载电阻，单片机输出高电平并形成环路。电阻将电流从单片机内部拉出的过程就是拉电流过程。
- 灌电流：相对于单片机而言，主动吸收电流的能力就是灌电流能力。例如，在单片机 I/O 和 VCC 之间连接 1 K 负载电阻时时，单片机输出低电平并形成环路。外部电源将电流注入单片机内部的过程称为填充电流过程。

该单片机的每个 I/O 有四个等级的可配置驱动能力，Level 0 ~ 3，其中 Level 0 的驱动能力最弱，Level 3 驱动能力最强。

功能复用

GPIO 功能多路复用是将通用 I/O 用于其他功能，如需要 SCL 和 SDA 功能的 I²C I/O。对于该单片机，任何 GPIO 都可以被单片机支持的外设使用，如 SPI、UART、I²C、SPI、PS 等。单片机为所有外设定义了一个唯一的 ID，并且每个 I/O 可以在初始化时设置为外设的相应 ID。

中断功能

任何 GPIO 都可以产生外部中断，生成的外部中断由 CPU 处理并配置到中断组中。该单片机支持 3 组外部中断，分别为 GPIO_INT0 / GPIO_INT1 / GPIO_INT2 (IRQ_20、IRQ21、IRQ22)。单片机支持任意 GPIO 产生外部中断，触发方式可通过软件配置。触发方式包括：

- 高电平触发
- 低电平触发
- 上升沿触发
- 下降沿触发
- 双沿触发

唤醒功能

任何 GPIO 都可以将芯片从休眠或睡眠模式中唤醒。通过配置相应的寄存器，使能引脚触发唤醒请求。当引脚的唤醒功能使能时，可以通过配置相应的寄存器使能输入滤波器 (去抖功能) 进行去抖。去抖电路可避免毛刺引起的误唤醒。此外，还可以配置 GPIO 唤醒请求的极性。

GPIO API

GPIO ROM API

rom_hw_gpio_set_pin_cfg

函数原型

EN_ERR_STA_T rom_hw_gpio_set_pin_cfg(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, uint32_t u32Cfg);

描述

配置 GPIO 引脚。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚。
u32Cfg	引脚配置。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_pin_init

函数原型

EN_ERR_STA_T rom_hw_gpio_pin_init(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, uint32_t u32Cfg);

描述

初始化 GPIO。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚。
u32Cfg	引脚配置。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_set_pin_input_output

函数原型
EN_ERR_STA_T rom_hw_gpio_set_pin_input_output(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_PIN_MODE_T enMode);

描述
配置 GPIO 工作在输入和输出模式。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。
enMode	引脚输入或输出模式，@ ref EN_GPIO_PIN_MODE_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_set_pin_output_level

函数原型
EN_ERR_STA_T rom_hw_gpio_set_pin_output_level(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_LEVEL_T enLevel);

描述
设置或清除指示引脚的输出电平。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。
enLevel	输出电平状态，@ ref EN_GPIO_LEVEL_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_toggle_pin_output_level

函数原型
EN_ERR_STA_T rom_hw_gpio_toggle_pin_output_level(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin);

描述
翻转指示引脚的输出电平。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_get_pin_output_level

函数原型
EN_ERR_STA_T rom_hw_gpio_get_pin_output_level(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, uint32_t* pu32Level);

描述
获取指示引脚的输出状态。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。
pu32Level	引脚状态，@ ref EN_GPIO_LEVEL_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_get_pin_input_level

函数原型
EN_ERR_STA_T rom_hw_gpio_get_pin_input_level(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, uint32_t* pu32Level);

描述
获取指示引脚的输入状态。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。
pu32Level	引脚状态，@ ref EN_GPIO_LEVEL_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_set_pin_pid

函数原型
EN_ERR_STA_T rom_hw_gpio_set_pin_pid(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_PID_T enPID);

描述
设置指示引脚的外设功能。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。
enPID	引脚外设功能选择，@ ref EN_GPIO_PID_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_enable_qspi_pid

函数原型

EN_ERR_STA_T rom_hw_gpio_enable_qspi_pid(EN_QSPI_FIXED_GPIO_EN_T enCh);

描述

使能用于 QSPI 功能的 GPIO。

参数

参数	描述
enCh	QSPI 通道，@ ref EN_QSPI_FIXED_GPIO_EN_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_disable_qspi_pid

函数原型

EN_ERR_STA_T rom_hw_gpio_disable_qspi_pid(EN_QSPI_FIXED_GPIO_EN_T enCh);

描述

除能在 QSPI 功能中工作的指示引脚。

参数

参数	描述
enCh	QSPI 通道，@ ref EN_QSPI_FIXED_GPIO_EN_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_set_pin_pull_mode

函数原型

EN_ERR_STA_T rom_hw_gpio_set_pin_pull_mode(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_PULL_MODE_T enMode);

描述

配置指示引脚的上拉或下拉模式。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。
enMode	引脚上拉 / 下拉模式，@ ref EN_GPIO_PULL_MODE_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_set_pin_drive_strength

函数原型
EN_ERR_STA_T rom_hw_gpio_set_pin_drive_strength(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_DRV_STRENGTH_T enStrength);

描述
配置指示引脚的驱动强度。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。
enStrength	引脚驱动强度，@ ref EN_GPIO_DRIVE_STRENGTH_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_set_pin_interrupt_type

函数原型
EN_ERR_STA_T rom_hw_gpio_set_pin_interrupt_type(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_INT_CH_T enCh, EN_GPIO_INT_TYPE_T enType);

描述
设置指示引脚的中断类型。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。
enCh	中断句柄选择，@ ref EN_GPIO_INT_CH_T。
enType	中断类型，@ ref EN_GPIO_INT_TYPE_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_get_pin_interrupt_flag

函数原型
EN_ERR_STA_T rom_hw_gpio_get_pin_interrupt_flag(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, uint32_t* pu32Msk);

描述
获取指示引脚的中断标志 (状态)。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_clear_pin_interrupt_flag

函数原型

EN_ERR_STA_T rom_hw_gpio_clear_pin_interrupt_flag(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin);

描述

清除指示引脚的中断标志 (状态)。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_enable_pin_interrupt

函数原型

EN_ERR_STA_T rom_hw_gpio_enable_pin_interrupt(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin);

描述

使能指示引脚的中断。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_disable_pin_interrupt

函数原型

EN_ERR_STA_T rom_hw_gpio_disable_pin_interrupt(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin);

描述

除能指示引脚的中断。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_set_pin_wakeup_debounce

函数原型

EN_ERR_STA_T rom_hw_gpio_set_pin_wakeup_debounce(uint8_t u8Time, EN_GPIO_WAKEUP_DEBOUNCE_UNIT_T enUnit);

描述

设置所有引脚唤醒去抖，所有引脚共用一个唤醒去抖时间。

参数

参数	描述
u8Time	去抖时间，0 ~ 63。
enUnit	去抖单位，@ ref EN_GPIO_WAKEUP_DEBOUNCE_UNIT_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_enable_pin_wakeup

函数原型

EN_ERR_STA_T rom_hw_gpio_enable_pin_wakeup(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_WAKEUP_POL_T enPol);

描述

使能指示引脚的唤醒功能。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。
enPol	唤醒极性，@ ref EN_GPIO_WAKEUP_POL_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_disable_pin_wakeup

函数原型

EN_ERR_STA_T rom_hw_gpio_disable_pin_wakeup(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin);

描述

除能指示引脚的唤醒功能。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 GPIOA / GPIOB / GPIOC。
u32Pin	需要配置哪些引脚，@ ref EN_GPIO_PIN_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_clear_wakeup_flag

函数原型

EN_ERR_STA_T rom_hw_gpio_clear_wakeup_flag(void);

描述

清除指示引脚的唤醒标志。

参数

无

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

SW 端口 API 函数

rom_hw_gpio_enable_swd

函数原型

EN_ERR_STA_T rom_hw_gpio_enable_swd(stGPIO_SWD_Handle_t* pstGPIO);

描述

使能 Cortex®-M0+ 或 Cortex®-M33 SWD 端口。如果 SWD 端口配置为通用 I/O，需要先除能 SWD 端口功能。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 SWD_CP / SWD_MP。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_disable_swd

函数原型

EN_ERR_STA_T rom_hw_gpio_disable_swd(stGPIO_SWD_Handle_t* pstGPIO);

描述

除能 Cortex®-M0+ 或 Cortex®-M33 SWD 端口。如果 SWD 端口配置为通用 I/O，需要先除能 SWD 端口功能。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 SWD_CP / SWD_MP。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_set_sw_input_output

函数原型
EN_ERR_STA_T rom_hw_gpio_set_sw_input_output(stGPIO_SWD_Handle_t* pstGPIO, uint8_t u8Pin, EN_GPIO_PIN_MODE_T enMode);

描述
配置 SWD 接口引脚 (SWDIO 或 SWCLK) 的输入或输出模式。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 SWD_CP / SWD_MP。
u8Pin	要配置的引脚，@ ref EN_GPIO_SW_NUM_T。
enMode	引脚输入或输出模式，@ ref EN_GPIO_PIN_MODE_T。

返回	
HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

rom_hw_gpio_set_sw_output_level

函数原型
EN_ERR_STA_T rom_hw_gpio_set_sw_output_level(stGPIO_SWD_Handle_t* pstGPIO, uint8_t u8Pin, EN_GPIO_LEVEL_T enLevel);

描述
设置指示引脚 (SWDIO 或 SWCLK) 的输出状态。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 SWD_CP / SWD_MP。
u8Pin	要配置的引脚，@ ref EN_GPIO_SW_NUM_T。
enLevel	输出电平状态，@ ref EN_GPIO_LEVEL_T。

返回	
HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

rom_hw_gpio_get_sw_input_level

函数原型
EN_ERR_STA_T rom_hw_gpio_get_sw_input_level(stGPIO_SWD_Handle_t* pstGPIO, uint8_t u8Pin, uint8_t* pu8Level);

描述
获取指示引脚 (SWDIO 或 SWCLK) 的输入状态。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 SWD_CP / SWD_MP。
u8Pin	要配置的引脚，@ ref EN_GPIO_SW_NUM_T。

返回	
HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

rom_hw_gpio_set_sw_pull_mode

函数原型
EN_ERR_STA_T rom_hw_gpio_set_sw_pull_mode(stGPIO_SWD_Handle_t* pstGPIO, uint8_t u8Pin, EN_GPIO_PULL_MODE_T enMode);

描述
配置 SWD 接口引脚 (SWDIO 或 SWCLK) 的上拉模式或下拉模式。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 SWD_CP / SWD_MP。
u8Pin	要配置的引脚，@ ref EN_GPIO_SW_NUM_T。
enMode	引脚上拉 / 下拉模式，@ ref EN_GPIO_PULL_MODE_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_set_sw_drive_strength

函数原型
EN_ERR_STA_T rom_hw_gpio_set_sw_drive_strength(stGPIO_SWD_Handle_t* pstGPIO, uint8_t u8Pin, EN_GPIO_DRV_STRENGTH_T enStrength);

描述
配置 SWD 接口引脚 (SWDIO 或 SWCLK) 的驱动强度。

参数

参数	描述
pstGPIO	GPIO 外设句柄，应为 SWD_CP / SWD_MP。
u8Pin	要配置的引脚，@ ref EN_GPIO_SW_NUM_T。
enStrength	引脚驱动强度，@ ref EN_GPIO_DRV_STRENGTH_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

IR API 函数

rom_hw_gpio_enable_ir_tx_out

函数原型
EN_ERR_STA_T rom_hw_gpio_enable_ir_tx_out(void);

描述
使能 IR TX 输出。

参数
无

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_disable_ir_tx_out

函数原型
EN_ERR_STA_T rom_hw_gpio_disable_ir_tx_out(void);

描述
除能 IR TX 输出。

参数
无

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_enable_ir_rx_amp

函数原型
EN_ERR_STA_T rom_hw_gpio_enable_ir_rx_amp(void);

描述
使能 IR RX AMP。

参数
无

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_disable_ir_rx_amp

函数原型
EN_ERR_STA_T rom_hw_gpio_disable_ir_rx_amp(void);

描述
除能 IR RX AMP。

参数
无

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_set_ir_rx_rtune

函数原型
EN_ERR_STA_T rom_hw_gpio_set_ir_rx_rtune(uint8_t u8Rtune);

描述
设置 RC 滤波器中 R 的阻值。

参数

参数	描述
u8Rtune	$R = 50\text{ K} \times 2^{(u8Rtune)}$ 。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_get_ir_ctrl_value

函数原型

EN_ERR_STA_T rom_hw_gpio_get_ir_ctrl_value(uint32_t* pu32IrCtrlValue);

描述

获取 IR 控制寄存器值。

参数

参数	描述
pu32IrCtrlValue	指向 32-bit 缓冲器的指针，用于返回 IR CTRL 寄存器值。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

DCXO 引脚 API 函数

rom_hw_gpio_enable_dcxo32k_output

函数原型

EN_ERR_STA_T rom_hw_gpio_enable_dcxo32k_output(uint8_t u8Io);

描述

使能 PB07 (P39) 或 (和) PB10 (P42) 的 DCXO32K 时钟输出。

参数

参数	描述
u8Io	输出 I/O，@ ref EN_DCXO32K_OUT_CFG_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_gpio_disable_dcxo32k_output

函数原型

EN_ERR_STA_T rom_hw_gpio_disable_dcxo32k_output(uint8_t u8Io);

描述

除能 PB07 (P39) 或 (和) PB10 (P42) 的 DCXO32K 时钟输出。

参数

参数	描述
u8Io	输出 I/O，@ ref EN_DCXO32K_OUT_CFG_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

GPIO 范例

上一节阐述了 GPIO 的主要函数，本节将基于 GPIO 的主要函数开发 DEMO 工程。

输入范例代码

输入工程主要用于测试 GPIO 能否通过 I/O 口读取外部传输的信号。范例代码如下所示。

```
uint32_t i;
uint32_t u32Level = 0;
rom_hw_gpio_set_pin_pull_mode(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO, enPullMode);
rom_hw_gpio_set_pin_input_output(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO,
                                  GPIO_MODE_INPUT);

for (i = 0; i < 16; i++)
{
    rom_hw_gpio_get_pin_input_level(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO,
                                    &u32Level);
    PRINTF ("GPIO input level: %08X\n", u32Level);
    rom_delay_ms (1000);
}
```

输出范例代码

输出范例工程主要用于测试 GPIO 是否可以通过 I/O 口向外传输信号。范例代码如下所示。

```
uint32_t u32Level;
rom_hw_gpio_set_pin_input_output(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO,
                                  GPIO_MODE_OUTPUT);
rom_hw_gpio_set_pin_drive_strength(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO,
                                    enStrength);
rom_hw_gpio_set_pin_output_level(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO, 1);
rom_hw_gpio_get_pin_output_level(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO,
                                  &u32Level);
PRINTF ("(1) GPIO output level: %08X\n", u32Level);
rom_delay_ms (1000);
rom_hw_gpio_set_pin_output_level(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO, 0);
rom_hw_gpio_get_pin_output_level(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO,
                                  &u32Level);
PRINTF ("(2) GPIO output level: %08X\n", u32Level);
```

GPIO 中断范例代码

该范例工程主要用于测试是否能够响应通过 GPIO 产生的外部中断。范例工程如下所示。

中断处理程序

```
uint32_t u32Flag, u32Level;
/* Get interrupt status */
rom_hw_gpio_get_pin_interrupt_flag(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO,
                                    &u32Flag);
rom_hw_gpio_clear_pin_interrupt_flag(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO);
PRINTF ("io int flag: %08X\n", u32Flag);
rom_hw_gpio_get_pin_input_level(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO, &u32Level);
PRINTF ("io int level: %08X\n", u32Level);
```

中断 Demo

```
rom_hw_gpio_set_pin_input_output(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO, GPIO_
                                MODE_INPUT);

if (GPIO_INT_HIGH_LEVEL == enType)
{
    rom_hw_gpio_set_pin_pull_mode(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO, G PIO_
    PULL_DOWN);
}
else
{
    rom_hw_gpio_set_pin_pull_mode(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO, G PIO_
    PULL_UP);
}
/* Configure for gpio interrupt */
/* Registration interruption */
rom_hw_sys_ctrl_enable_peri_int (SYS_CTRL_MP, GPIO_IRQ0);
/* Enable gpiol interrupt */
NVIC_ClearPendingIRQ (GPIO_IRQ0);
NVIC_SetPriority (GPIO_IRQ0, 0x3);
NVIC_EnableIRQ (GPIO_IRQ0);
/* Disable gpio interrupt */
rom_hw_gpio_disable_pin_interrupt (GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO);
/* Initialize io to interrupt mode. */
rom_hw_gpio_set_pin_interrupt_type (GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO,
    GPIO_INT_CH0, enType);

/* Enable gpio interrupt */
rom_hw_gpio_enable_pin_interrupt (GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO);
```

GPIO 唤醒范例代码

该范例工程项目用于测试是否可以通过 GPIO 唤醒处于睡眠模式的 CPU。

```
#define WAKEUP_PIN GPIO_21
/* Configure wake-up source */
rom_hw_pmu_set_gpio_wakeup_source(n, PMU_LUT_TRIG_GPIOB, WAKEUP_PIN, LUT_ACT_PD_
    SYS_ON);

/* Configure debounce time */
rom_hw_gpio_set_pin_wakeup_debounce(60, GPIO_WAKEUP_DEB_UNIT_30US);
/* enable wakeup */
rom_hw_gpio_enable_pin_wakeup(GPIOB, WAKEUP_PIN, GPIO_WAKEUP_HIGH_LEVEL);
```

3 通用定时器 (GPTM)

简介

通用定时器 GPTM，也称为 GTIM，是一个 32-bit 定时器，配置了一个 4 通道比较器以使用定时器功能。GPTM 工作在 32K 时钟域，当单片机进入睡眠模式后，GPTM 仍然可以运行并唤醒系统。GPTM 引入了一种灵活的时钟方案，可提供所需的功能和性能，同时较大限度地降低功耗。

特性

- 可使用 16 MHz / 32 kHz 时钟或 GPIO 触发源作为时钟源
- 中断类型：cpu_cap / 溢出 / 匹配 / capture_coverd / 解码
- 2 路 PWM 和组合 32 位 PWM
- 1 路红外发射
- 2 路 GPIO / IR 触发捕捉功能
 - 支持 CNT_A / CNT_B 捕捉 (不支持同时)
 - 支持组合 32 位捕捉 (仅 CNT_A 使能)
- 2 路 GPIO / IR 触发解码功能
 - 支持 CNT_A / CNT_B 解码 (支持同时)
 - 支持组合 32 位解码 (仅 CNT_A 使能)
- 支持解码捕捉环回 (CNT_A 解码, CNT_B 捕捉)
- 可作为一个 32-bit 计数器或两个 16-bit 分频计数器使用
- 支持中断捕捉模式 (与 DMA 捕捉模式同时只能选择一种)
- 支持捕捉电平指示 (在 16-bit 模式下 capcnt[15] 表示 CNT_B 高电平 (为 0 时) 和低电平 (为 1 时) 的计数值, capcnt[31] 表示 CNT_A 电平的计数值, 在 32-bit 模式下 capcnt[31] 表示组合 32 位 cnt 电平的计数值)
- 支持多个 GPTM 同时启动和停止
- 比较值在下个周期生效

功能描述

该单片机主要具有计数、产生 PWM 波形和采样功能。

计数器模式 (CNT 模式)

当 GPTM 工作在计数器模式时，32-bit 计数器寄存器包含两个 16-bit 计数器，CNT_A 和 CNT_B，如下图所示：

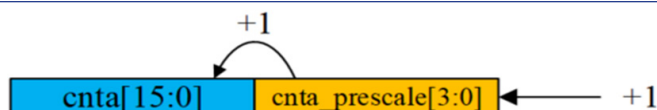


图 3. GPTM 计数器寄存器

如上图所示，当 CNT_A 计数输入有效时，cnta_precale 先计数，当 precale 计数达到设定值时，cnta 加 1。

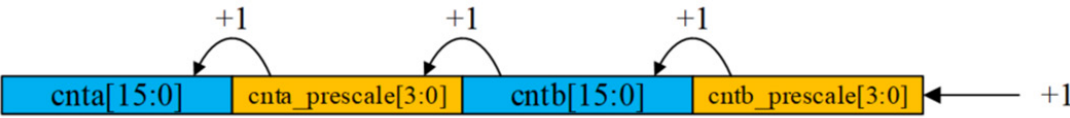


图 4. GPTM 预分频器

当两个 cnt 组合使用时，cnta_precale 的值应设为 0。cntb_precale 先计数，当计数达到设定值时，CNT_B 加 1。当 CNT_B 满时，将它送入 cnta_precale，当 cnta_precale 计数到设定值后，再将它送入 CNT_A。

计数以递增方式开始，寄存器控制变化点是比较点 (大于零的比较值被认为是有效值) 还是溢出点，以及到达变化点后的变化是复位 (递增方式复位为 0，递减方式复位为全 f) 还是渐变 (递增改为递减，递减改为递增)。比较值设置原理如下表所示。

表 3. GPTM 模式选择

rst_mode	rstval_mode	递增	递减
0	0	计数到 compare 后继续计数，当达到 16'hfff_fff 时复位为 0。	计数到 compare 后继续计数，当达到 0 时复位为 16'hfff_fff。
0	1	当达到 compare 时，计数复位为 0，并继续递增。	当达到 compare 时，计数复位为 16'hfff_fff，并继续递减。
1	0	计数到 compare 后继续计数，当达到 16'hfff_fff 时变为递减。	计数到 compare 后继续计数，当达到 0 时变为递增。
1	1	计数到 compare 后变为递减。	计数到 compare 后变为递增。

上表显示了不同模式下的工作原理。为了更形象地说明原理，下面将给出不同模式下的时序图。

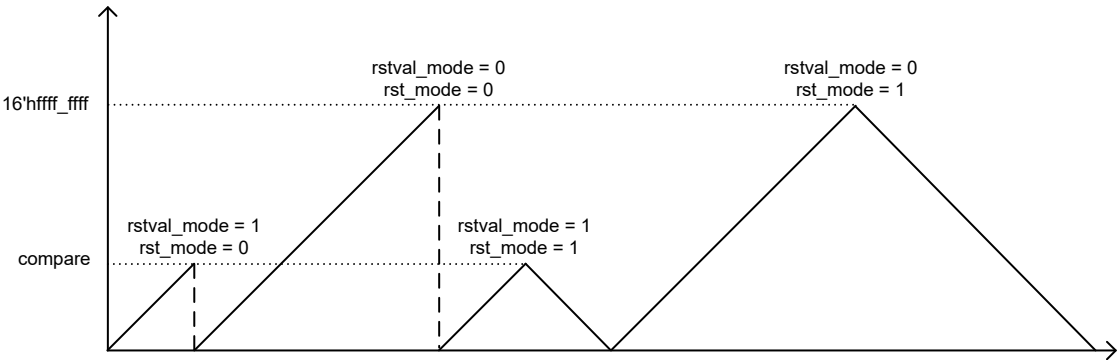
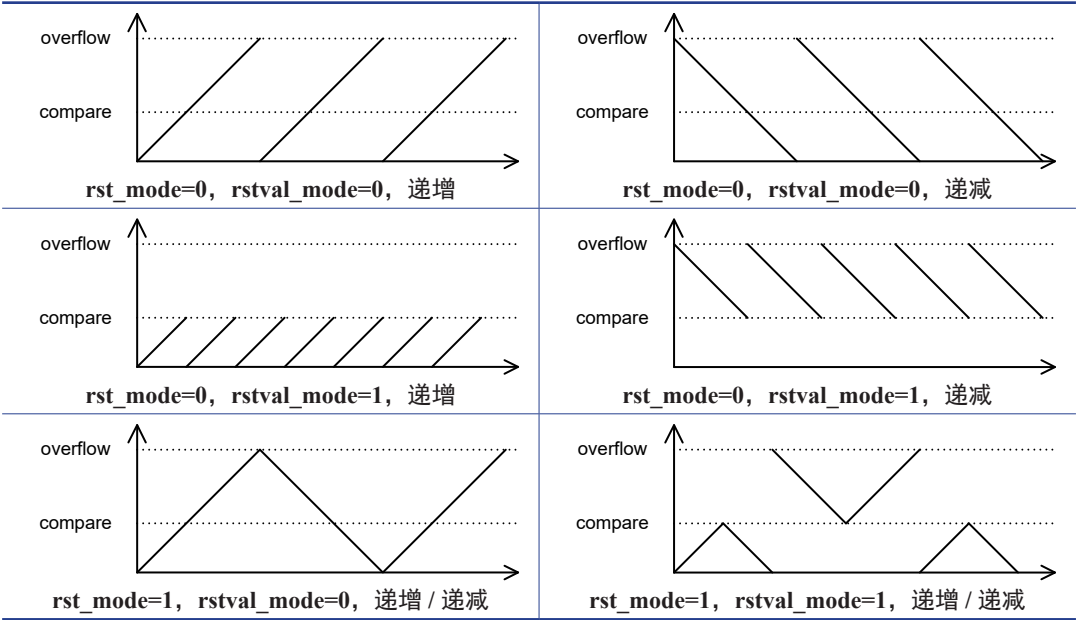


图 5. GPTM 工作模式时序图

通用功能时序说明

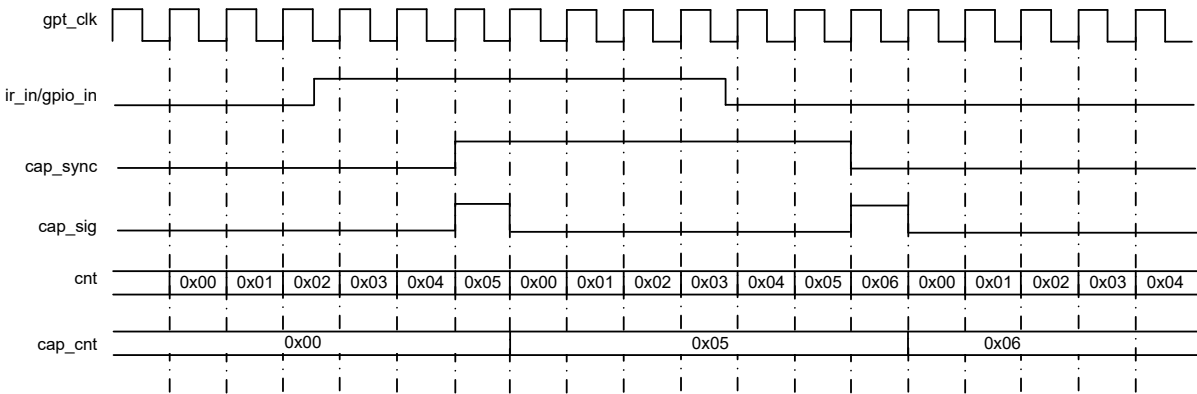


捕捉模式

捕捉功能用于检测 IR 输入信号或 GPIO 输入信号的边沿间隔。

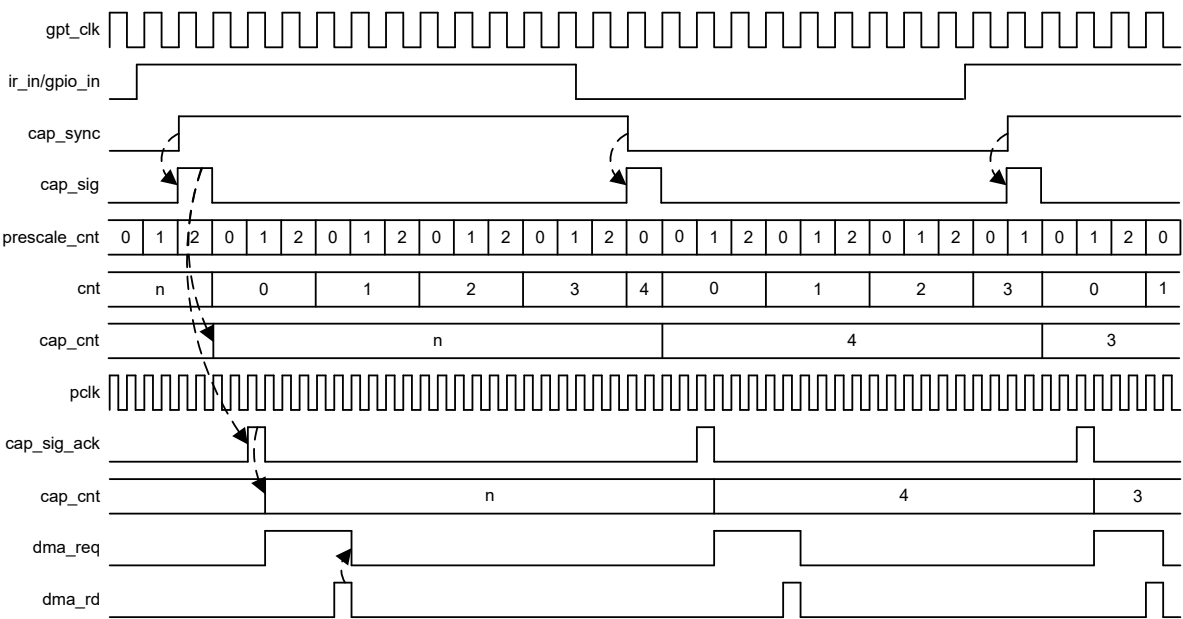
输入信号同步后，进行边沿检测，可选择上升沿、下降沿或双边沿采样。对于双边沿采样，可以使用电平指示功能将最高位作为电平指示位。每个采样点首先将 `cnt` 值保存到 `cnt_tmp`，并将 `cnt` 值清零，然后产生一个 dma 请求信号请求 dma 携带数据，或者使用中断捕捉方式产生中断，CPU 读取 `cnt` 数据。

如果在请求生成和数据处理之间的时间段内出现一个新的采样点，则忽略该采样点并报告 `capcov` 中断。需注意的是，采用点会将 `cnt` 清零，但不会覆盖 `cnt_tmp` 值。捕捉模式的基本功能如下图所示。



从上图可以看出，在 `cnt` 不除频的情况下，由于清零动作需要占用一个周期或者信号边沿本身占用一个周期，所以实际计数两个信号边沿从 `0x0` 到 `0x6` 共 7 个周期，此时可以直接使用 `cap_cnt + 1` 作为实际长度。

当有预分频频率时，如 3 分频，如下图所示，由于除法本身造成的误差，`cap_sig` 可能到达除法计数器 `prescale_cnt` 的 2、0 或 1，那么实际的计数周期 (`gpt_clk`) 应分别是 `cap_cnt × 3 + 3`、`cap_cnt × 3 + 1`、`cap_cnt × 3 + 2`，此时不能直接使用 `cap_cnt + 1` 作为实际的 `cnt` 计数值。



捕捉新的附加电平指示功能：由于双边沿采样难以区分上下边沿，因此占用 1-bit 最高位作为电平指示 (非双边沿采样请关闭此功能)，16-bit cnt 计数时，bit[15] 作为电平指示位，32-bit cnt 计数时，位 [31] 作为电平指示位，1 表示高电平。当使用这个功能时，compare 应配置为 15'h7fff (16-bit cnt) 或 31'h7fff_ffff 作为溢出中断，因为最高位被占用了。

PWM 模式

GPTM 模块也用于生成指定的波形。其工作原理描述如下。

比较值是计数周期，当计数值在 pwm_low 和 pwm_hi 之间时输出指定的电平 (高低电平由 pwm_pol 决定)。

pwm_low 和 pwm_hi 的值在配置完成触发器开始后第一次生效，其次在比较周期结束后生效。比较值可以在当前周期或下一个周期生效。

注：pwm_low ≤ cnt < pwm_hi 输出指定的电平，所以配置占空比为 0%: pwm_low 和 pwm_hi 都大于 compare 或 pwm_low ≥ pwm_hi; 配置占空比为 100%: pwm_low = 0, pwm_hi 大于 compare (compare 配置为全 f，不能产生 100% 占空比)。

GPTM API

定时器功能 API

rom_hw_timer_trig_cfg_valid

函数原型

EN_ERR_STA_T rom_hw_timer_trig_cfg_valid(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

使定时器的配置有效。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_work_mode

函数原型

EN_ERR_STA_T rom_hw_timer_set_work_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

通过写入具有指示位的 TIMER_MODE 寄存器来选择定时器工作模式。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_get_mode

函数原型

EN_ERR_STA_T rom_hw_timer_get_mode(stTIMER_Handle_t* pstTIMER, uint32_t* pu32Mode);

描述

获取定时器的模式。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
pu32Mode	指向定时器模式配置。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_clock_prescale

函数原型
EN_ERR_STA_T rom_hw_timer_set_clock_prescale(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint8_t u8Prescale);

描述
配置指示定时器的时钟预分频。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
u8Prescale	输入时钟的分频，u8precale 的范围为 0 ~ 0x0F。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_counter_mode

函数原型
EN_ERR_STA_T rom_hw_timer_set_counter_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_TIMER_CNT_MODE_T enMode);

描述
配置指示定时器工作在计数器模式。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
enMode	定时器计数器模式控制，参考 EN_TIMER_CNT_MODE_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_get_cfg

函数原型
EN_ERR_STA_T rom_hw_timer_get_cfg(stTIMER_Handle_t* pstTIMER, uint32_t* pu32Cfg);

描述
获取定时器配置。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
pu32Cfg	指向定时器配置。

返回	
HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

rom_hw_timer_start

函数原型

EN_ERR_STA_T rom_hw_timer_start(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

启动指示的定时器。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_stop

函数原型

EN_ERR_STA_T rom_hw_timer_stop(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

停止指示的定时器。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_get_counter

函数原型

EN_ERR_STA_T rom_hw_timer_get_counter(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t* pu32Cnt);

描述

获取指示的定时器当前计数器。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
pu32Cnt	当前定时器计数器。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_clear_counter

函数原型

EN_ERR_STA_T rom_hw_timer_clear_counter(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

清除指定的定时器计数器。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_compare

函数原型

EN_ERR_STA_T rom_hw_timer_set_compare(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32Value);

描述

配置指定的定时器比较值。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
u32Value	比较值。 16-bit 定时器：比较值范围为 0 ~ 0xFFFF。 32-bit 定时器：比较值范围为 0 ~ 0xFFFFFFFF。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_get_compare

函数原型

EN_ERR_STA_T rom_hw_timer_get_compare(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t* pu32Value);

描述

获取指示的定时器比较值。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
pu32Value	指向比较值。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_get_interrupt_flag

函数原型

EN_ERR_STA_T rom_hw_timer_get_interrupt_flag(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t* pu32Msk);

描述

通过读取 TIMER_INT_FLAG 寄存器来获取定时器中断标志 (状态)。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
pu32Msk	指示将读取哪个中断标志。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_clear_interrupt_flag

函数原型

EN_ERR_STA_T rom_hw_timer_clear_interrupt_flag(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32Msk);

描述

通过写入 TIMER_INT_CLR 寄存器来清除指示的定时器中断标志 (状态)。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
pu32Msk	指示将清除哪个标志。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_enable_interrupt

函数原型

EN_ERR_STA_T rom_hw_timer_enable_interrupt(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32IntEn);

描述

通过写入具有指示位的 TIMER_INT_EN 寄存器来使能指定的定时器中断。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
u32IntEn	指示将使能哪个中断。 Bit 为 1 表示使能 Bit 为 0 表示不影响 参考 EN_TIMER_INT_FLAG_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_disable_interrupt

函数原型

EN_ERR_STA_T rom_hw_timer_disable_interrupt(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32IntDis);

描述

通过写入具有指示位的 TIMER_INT_EN 寄存器来除能指定的定时器中断。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
u32IntDis	指示将除能哪个中断。 Bit 为 1 表示除能 Bit 为 0 表示不影响 参考 EN_TIMER_INT_FLAG_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_enable_wakeup

函数原型

EN_ERR_STA_T rom_hw_timer_enable_wakeup(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32En);

描述

使能定时器唤醒功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
u32En	指示将使能哪个唤醒源，参考 EN_TIMER_WAKEUP_SRC_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_disable_wakeup

函数原型

EN_ERR_STA_T rom_hw_timer_disable_wakeup(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32Dis);

描述

除能定时器唤醒功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
u32Dis	指示将除能哪个唤醒源，参考 EN_TIMER_WAKEUP_SRC_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_enable_sync_start

函数原型

EN_ERR_STA_T rom_hw_timer_enable_sync_start(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

使能指示定时器的开始同步工作功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_disable_sync_start

函数原型

EN_ERR_STA_T rom_hw_timer_disable_sync_start(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

除能指示定时器的开始同步工作功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_enable_sync_stop

函数原型

EN_ERR_STA_T rom_hw_timer_enable_sync_stop(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

使能指示定时器的停止同步工作功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_disable_sync_stop

函数原型

EN_ERR_STA_T rom_hw_timer_disable_sync_stop(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

除能指示定时器的停止同步工作功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_enable_compare_volid_delay

函数原型

EN_ERR_STA_T rom_hw_timer_enable_compare_volid_delay(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

配置指示的定时器比较值在下一个比较周期有效。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个定时器，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_disable_compare_valid_delay

函数原型

EN_ERR_STA_T rom_hw_timer_disable_compare_valid_delay(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

配置指示的定时器比较值立即有效。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个定时器，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_get_work_status

函数原型

EN_ERR_STA_T rom_hw_timer_get_work_status(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint8_t* pu8Status);

描述

获取指示的定时器工作状态。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示哪个定时器，参考 EN_TIMER_CH_T 枚举定义。
pu8Status	定时器的工作状态，参考 EN_TIMER_STATUS_T 枚举定义。 TIMER_STATUS_RUNNING：定时器正在运行； TIMER_STATUS_STOP：定时器除能或停止。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_get_counter_mode

函数原型

EN_ERR_STA_T rom_hw_timer_get_counter_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint8_t* pu8Mode);

描述

获取指示的定时器计数器模式。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	enCh：指示哪个定时器，参考 EN_TIMER_CH_T 枚举定义。
pu8Mode	定时器计数器模式，参考 EN_TIMER_COUNTER_MODE_T 枚举定义。 TIMER_COUNTER_MODE_DECREASE：定时器工作在递减模式； TIMER_COUNTER_MODE_INCREASE：定时器工作在递增模式。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_get_wakeup_status

函数原型

EN_ERR_STA_T rom_hw_timer_get_wakeup_status(stTIMER_Handle_t* pstTIMER, uint8_t* pu8State);

描述

获取定时器唤醒源的工作状态。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
pu8Status	指向保存定时器唤醒源工作状态。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hal_timer_enable_clk

函数原型

EN_ERR_STA_T rom_hal_timer_enable_clk(stTIMER_Handle_t* pstTIMER);

描述

使能定时器 clk。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hal_timer_disable_clk

函数原型

EN_ERR_STA_T rom_hal_timer_disable_clk(stTIMER_Handle_t* pstTIMER);

描述

除能定时器 clk。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hal_timer_init

函数原型
EN_ERR_STA_T rom_hal_timer_init(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, stTimerInit_t* pstTimerInit);

描述
初始化指示的定时器。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将使能哪个通道，请参考 EN_TIMER_CH_T 枚举定义。
pstTimerInit	初始化定时器参数，参考 stTimerInit_t 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hal_timer_set_compare

函数原型
EN_ERR_STA_T rom_hal_timer_set_compare(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32Value);

描述
配置指示的定时器比较值。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置给哪个通道比较值。
u32Value	比较值。 16-bit 定时器：比较值范围为 0 ~ 0xFFFF。 32-bit 定时器：比较值范围为 0 ~ 0xFFFFFFFF。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

PWM 功能 API

rom_hw_timer_enable_pwm

函数原型
EN_ERR_STA_T rom_hw_timer_enable_pwm(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述
使能 PWM 模块。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将使能哪个定时器的 PWM，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_disable_pwm

函数原型

EN_ERR_STA_T rom_hw_timer_disable_pwm(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

除能 PWM 模块。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将除能哪个定时器的 PWM，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_pwm_hi_low_cnt

函数原型

EN_ERR_STA_T rom_hw_timer_set_pwm_hi_low_cnt(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32HiCount, uint32_t u32LoCount);

描述

配置指示定时器的比较值高低计数器。

注：16-bit 定时器：计数器取值范围为 0 ~ 0xFFFF。

32-bit 定时器：计数器取值范围为 0 ~ 0xFFFFFFFF。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道。
u32HiCount	比较值高计数器。
u32LoCount	比较值低计数器。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_get_pwm_hi_low_cnt

函数原型

EN_ERR_STA_T rom_hw_timer_get_pwm_hi_low_cnt(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t* pu32HiCount, uint32_t* pu32LoCount);

描述

获取高低计数器的指定 PWM 通道。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道。
pu32HiCount	比较值 PWM 高计数器。
pu32LoCount	比较值 PWM 低计数器。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_pwm_polarity

函数原型

EN_ERR_STA_T rom_hw_timer_set_pwm_polarity(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_PWM_POL_T enPol);

描述

设置 PWM 输出极性。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道。
enPol	PWM 极性，参考 EN_PWM_POLARITY_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hal_timer_pwm_init

函数原型

EN_ERR_STA_T rom_hal_timer_pwm_init(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32Frequency, uint16_t u16Duty);

描述

配置指定的 PWM 频率和占空比。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置给哪个 PWM 频率和占空比，参考 EN_PWM_CH_T 枚举定义。
u32Frequency	PWM 输出频率，单位：Hz
u16Duty	PWM 占空比：0 ~ 10000 → (0 % ~ 100.00 %)， 精度：定时器 CLK / PWM 频率

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hal_timer_pwm_hi_lo_cnt

函数原型
EN_ERR_STA_T rom_hal_timer_set_pwm_hi_lo_cnt(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32HiCount, uint32_t u32LoCount);

描述
配置指示的 PWM 比较值和高、低计数器。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个 PWM，参考 EN_PWM_CH_T 枚举定义。
u32HiCount	PWM 高计数器。
u32LoCount	PWM 低计数器。

返回	
HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

rom_hal_timer_enable_pwm

函数原型
EN_ERR_STA_T rom_hal_timer_enable_pwm(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述
使能 PWM 模块。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将使能哪个 PWM，参考 EN_PWM_CH_T 枚举定义。

返回	
HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

rom_hal_timer_disable_pwm

函数原型
EN_ERR_STA_T rom_hal_timer_disable_pwm(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述
除能 PWM 模块。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将除能哪个 PWM，参考 EN_PWM_CH_T 枚举定义。

返回	
HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。

定时器捕捉功能

rom_hw_timer_enable_capture

函数原型

EN_ERR_STA_T rom_hw_timer_enable_capture(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

使能定时器捕捉功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将使能哪个通道，参考 EN_PWM_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_disable_capture

函数原型

EN_ERR_STA_T rom_hw_timer_disable_capture(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

除能定时器捕捉功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将除能哪个通道，参考 EN_PWM_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_capture_and_decode_src

函数原型

EN_ERR_STA_T rom_hw_timer_set_capture_and_decode_src(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_CAP_DECODE_SIGNAL_T enSignal);

描述

配置捕捉和解码信号源。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示哪个通道，参考 EN_PWM_CH_T 枚举定义。
enSignal	捕捉信号，参考 EN_CAP_DECODE_SIGNAL_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_capture_chb_src

函数原型
EN_ERR_STA_T rom_hw_timer_set_capture_chb_src(stTIMER_Handle_t* pstTIMER, EN_CAP_CHB_SRC_T enSrc);

描述
设置指定的定时器捕捉通道 B 信号源。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enSrc	捕捉通道 B 的信号源，参考 EN_CAP_CHB_SRC_T 枚举定义。 TIMER_CAP_CHB_SRC_GPIO_IR：捕捉通道 B 的信号源是 GPIO 或 IR TIMER_CAP_CHB_SRC_DECODE：捕捉通道 B 的信号源是通道 A 解码信号

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_capture_mode

函数原型
EN_ERR_STA_T rom_hw_timer_set_capture_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_CAP_MODE_T enMode);

描述
配置捕捉模式。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个定时器，参考 EN_TIMER_CH_T 枚举定义。
enMode	捕捉模式，参考 EN_CAP_MODE_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_get_capture_counter

函数原型
EN_ERR_STA_T rom_hw_timer_get_capture_counter(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t* pu32CapCnt);

描述
获取指示的定时器捕捉计数器。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示哪通道，参考 EN_TIMER_CH_T 枚举定义。
pu32CapCnt	指向捕捉计数器。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_capture_trig_mode

函数原型

EN_ERR_STA_T rom_hw_timer_set_capture_trig_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_CAP_TRIG_MODE_T enMode);

描述

配置捕捉触发方式。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示哪个通道，参考 EN_TIMER_CH_T 枚举定义。
enMode	捕捉触发方式，参考 EN_CAP_TRIG_MODE_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_timer_set_capture_counter_format

函数原型

EN_ERR_STA_T rom_hw_timer_set_capture_counter_format(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_CAP_CNT_FORMAT_T enFormat);

描述

配置捕捉计数器格式。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示哪个通道，参考 EN_TIMER_CH_T 枚举定义。
enFormat	捕捉计数器格式，参考 EN_CAP_CNT_FORMAT_T 枚举定义。 CAP_CNT_FORMAT_UNSIGNED：捕捉计数器为无符号格式。 CAP_CNT_FORMAT_SIGNED：捕捉计数器为有符号格式。 当捕捉工作在 16 位时，counter[15] = 0，表示输入信号为低电平； counter[15] = 1，表示输入信号为高电平。 当捕捉工作在 32 位时，counter[31] = 0，表示输入信号为低电平； counter[31] = 1，表示输入信号为高电平。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hal_timer_capture_init

函数原型

EN_ERR_STA_T rom_hal_timer_capture_init(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, stCapInit_t* pstCapInit);

描述

初始化指示的定时器捕捉功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个通道，参考 EN_TIMER_CH_T 枚举定义。
pstCapInit	捕捉初始结构，参考 stCapInit_t 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hal_timer_enable_capture

函数原型

EN_ERR_STA_T rom_hal_timer_enable_capture(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

使能定时器捕捉功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将使能哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hal_timer_disable_capture

函数原型

EN_ERR_STA_T rom_hal_timer_disable_capture(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

除能定时器捕捉功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将除能哪个通道，参考 EN_TIMER_CH_T 枚举定义。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

GPTM 范例

第一节详细介绍了 GPTM 的主要功能，本节在前两节基础上开发了相应的 DEMO。

计数器模式范例代码

本范例以 GPTM2 为例，在溢出和比较两种模式下实现 cnt 模块的功能，并在发生中断时进行电平翻转。

中断处理程序

```
static void gtim2_irq_handler(void)
{
    uint32_t u32IntFlag = 0;
    rom_hw_timer_get_interrupt_flag(TIMER2, TIMER_CHA, &u32IntFlag);
    rom_hw_timer_clear_interrupt_flag(TIMER2, TIMER_CHA, u32IntFlag);
    if(u32IntFlag & TIMER_INT_MATCH)
    {
        rom_hw_gpio_toggle_pin_output_level(GPIO_PORT_TIM2_CHA, GPIO_PIN_TIM2_CHA);
    }
    u32IntFlag = 0;
    rom_hw_timer_get_interrupt_flag(TIMER2, TIMER_CHB, &u32IntFlag);
    rom_hw_timer_clear_interrupt_flag(TIMER2, TIMER_CHB, u32IntFlag);
    if(u32IntFlag & TIMER_INT_OVERFLOW)
    {
        rom_hw_gpio_toggle_pin_output_level(GPIO_PORT_TIM2_CHB, GPIO_PIN_TIM2_CHB);
    }
}
```

GPTM 计数器模式范例代码

```
static void gtim2_mode_set(void)
{
    uint8_t u8Status = 0;
    stTimerInit_t pstTimerInit;
    pstTimerInit.u32Compare = 0;
    pstTimerInit.u8CounterMode = 0;
    pstTimerInit.u8Prescale = 0;
    stTimerInit_t pstTimerInit_A;
    pstTimerInit_A.u32Compare = 32767;
    pstTimerInit_A.u8CounterMode = 1;
    pstTimerInit_A.u8Prescale = 0;
    // Enable GTIM2 Clock
    rom_hal_timer_enable_clk(TIMER2);
    // Init GTIM2 CHB and CHA(Compare, Counter Mode and Prescale)
    rom_hal_timer_init(TIMER2, TIMER_CHB, &pstTimerInit);
    rom_hal_timer_init(TIMER2, TIMER_CHA, &pstTimerInit_A); 20.
    // Set GPTM2 CHB INT of Overflow and CHA INT of compare
    rom_hw_timer_enable_interrupt(TIMER2, TIMER_CHB, TIMER_INT_OVERFLOW);
    rom_hw_timer_enable_interrupt(TIMER2, TIMER_CHA, TIMER_INT_MATCH); 24.
    // Enable GPTM2 Peri INT
    rom_hw_sys_ctrl_enable_peri_int(SYS_CTRL_MP, TIMER2_IRQ); 27.
    // Register and Enable INT
    g_periIrqFuncTable[TIMER2_IRQ] = gtim2_irq_handler;
    NVIC_ClearPendingIRQ (TIMER2_IRQ);
    NVIC_SetPriority (TIMER2_IRQ, 0x3);
    NVIC_EnableIRQ (TIMER2_IRQ);
    __enable_irq();
    // START GPTM2 CHB and CHA
    rom_hw_timer_start(TIMER2, TIMER_CHB);
    rom_hw_timer_start(TIMER2, TIMER_CHA);
}
```

PWM 模式范例代码

此 DEMO 使用 GPTM2 在 CHA 通道和 CHB 通道上产生具有不同占空比的 PWM 波形。

中断处理程序

```
static void gtim3_irq_handler(void)
{
    uint32_t u32IntFlag = 0;
    rom_hw_timer_get_interrupt_flag(TIMER2, TIMER_CHA, &u32IntFlag);
    rom_hw_timer_clear_interrupt_flag(TIMER2, TIMER_CHA, u32IntFlag);
}
```

设置 PWM 模式

```
// GPIO PWM Init
rom_hw_gpio_set_pin_pid(GPIO_PORT_TIM3_CHB, GPIO_PIN_TIM3_CHB, PID_GTIM_PWM3_CHB);
rom_hw_gpio_set_pin_pid(GPIO_PORT_TIM3_CHA, GPIO_PIN_TIM3_CHA, PID_GTIM_PWM3_CHA);

// GTIM3 PWM Init
rom_hal_timer_pwm_init(TIMER3, TIMER_CHB, 16, 3000);
rom_hal_timer_pwm_init(TIMER3, TIMER_CHA, 16, 7000);
rom_hal_timer_set_pwm_polarity(TIMER3, TIMER_CHB, PWM_POL_RISING);
rom_hal_timer_set_pwm_polarity(TIMER3, TIMER_CHA, PWM_POL_RISING);
// Enable PWM Mode
rom_hal_timer_enable_pwm(TIMER3, TIMER_CHB);
rom_hal_timer_enable_pwm(TIMER3, TIMER_CHA);
rom_hw_sys_ctrl_enable_peri_int(SYS_CTRL_MP, TIMER3_IRQ);
g_periIrqFuncTable[TIMER3_IRQ] = gtim3_irq_handler;
NVIC_ClearPendingIRQ (TIMER3_IRQ);
NVIC_SetPriority (TIMER3_IRQ, 0x3);
NVIC_EnableIRQ (TIMER3_IRQ);
__enable_irq();
// START GPTM3
rom_hw_timer_start(TIMER3, TIMER_CHB);
rom_hw_timer_start(TIMER3, TIMER_CHA);
```

4 系统节拍定时器 (STIM)

简介

该单片机有两个 32-bit 系统节拍定时器 (STIM)。每个 32-bit 定时器都有一个 4 通道比较寄存器。

STIM 工作在低速时钟域，可以在睡眠模式下继续工作。STIM 引入了一种灵活的时钟方案，可提供所需的功能和性能，同时较大幅度地降低功耗。

特性

- 2 个独立的 STIM：STIM0 和 STIM1
- 支持低速 32 kHz 时钟作为时钟源
- 每个 STIM 支持 32-bit 计数器，12-bit 预分频
- 每个 STIM 支持 3 种类型的中断和唤醒：溢出 / 比较 / 节拍
- 每个 STIM 支持 4 通道比较中断和睡眠唤醒
- 独立支持 CPU 中断和 PMU 唤醒中断
- 支持配置成功标志检查

注：STIM 外设依靠低频时钟源提供频率。使用频率稳定的时钟源可以获得更好的时钟精度。

功能描述

STIM 外设由 STIM0 和 STIM1 组成，每个 STIM 的参考设计原理图如下所示。所有 STIM 有两个计数器、八个比较、两个节拍和溢出。

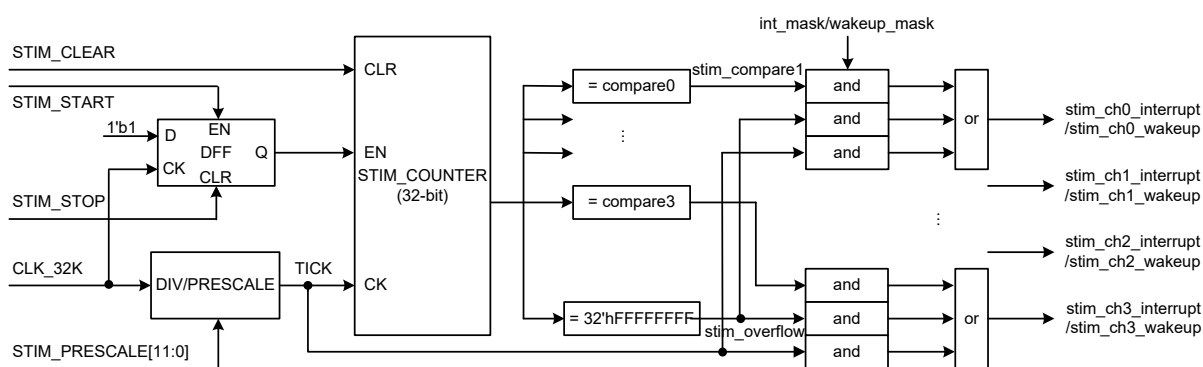


图 6. STIM 参考设计原理图

每个 STIM 是一个只能向上计数的 32-bit 计数器。它采用四个时钟源，分别是：

- RC 32K 低频时钟：RC_LCLK；
- DCXO 32K 低频时钟：DCXO_LCLK；
- RC 高频时钟分频：RC_HCLK_DIV_LCLK；
- DCXO 高频时钟分频：DCXO_HCLK_DIV_LCLK。

时钟精准度和预分频

该单片机的 STIM 使用 32768 Hz 时钟源，最小时间分辨率为 30.517 μs (一个节拍间隔)。下表是不同分频系数下每个节拍的时间间隔。溢出时间是计数器从 0 计数到 0xFFFF FFFF 所花费的时间。

表 4. 时间间隔

PRESCALE	节拍间隔	溢出时间
0	30.517 μs	2183.275 分钟
2 ⁸ - 1	7812.5 μs	9320.676 小时
2 ¹² - 1	125 ms	6213.784 天

设 PRESCALE 为分频系数，每个节拍的时间间隔可计算如下：

$$T_{RTC} = (\text{PRESCALE} + 1) / 32768, 0 \leq \text{PRESCALE} \leq 2^{12} - 1$$

注：该单片机的 PRESCALE 寄存器配置需要一个触发器 (Trig) 才能生效。

计数器、节拍、溢出和比较

该单片机的两个 STIM 外设的计数器长度均为 32-bit，可以在 STIM0 和 STIM1 中独立计数。每个 STIM 支持唤醒和中断配置：4 通道比较，溢出和节拍。当 STIM 工作时，计数器根据时钟信号上升沿的数量而增加并触发节拍中断。当计数器寄存器值达到比较 (n) 寄存器值时触发比较 (n) 中断；当计数器寄存器满 (0xFFFF FFFF) 时，溢出中断将被触发，计数器值将被清零，如下图所示。

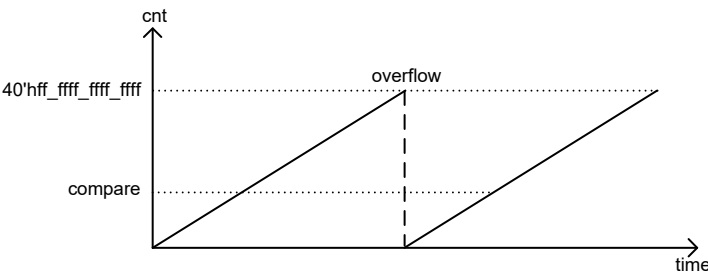


图 7. 计数器触发溢出

注：单片机默认除能节拍、溢出和比较 (n) 中断，使用时必须将其使能。

STIM API

STIM 中断 API

rom_hw_stim_get_interrupt_flag

函数原型

EN_ERR_STA_T rom_hw_stim_get_interrupt_flag (stSTIM_Handle_t* pstSTIM, uint16_t* pu16Flag);

描述

通过读取 STIM_INT_FLAG 寄存器来获取指示的定时器中断标志 (状态)。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
pu16Flag	指示将读取哪个中断标志。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_clear_interrupt_flag

函数原型

EN_ERR_STA_T rom_hw_stim_clear_interrupt_flag (stSTIM_Handle_t* pstSTIM, uint16_t u16Flag);

描述

通过写入 STIM_INT_CLR 寄存器来清除指示的 STIM 中断标志 (状态)。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
u16Flag	指示将清除哪个标志。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_enable_interrupt

函数原型

EN_ERR_STA_T rom_hw_stim_enable_interrupt (stSTIM_Handle_t* pstSTIM, uint16_t u16IntEn);

描述

通过写入具有指定位的 STIM_INT_EN 寄存器来使能指示的 STIM 中断。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
u16IntEn	指示将使能哪个中断，@ ref EN_STIM_INT_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_disable_interrupt

函数原型

EN_ERR_STA_T rom_hw_stim_disable_interrupt (stSTIM_Handle_t* pstSTIM, uint16_t u16IntDis);

描述

通过写入具有指定位的 STIM_INT_EN 寄存器来除能指示的 STIM 中断。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
u16IntDis	指示将除能哪个中断，@ ref EN_STIM_INT_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_enable_tick_overflow_interrupt

函数原型

EN_ERR_STA_T rom_hw_stim_enable_tick_overflow_interrupt (stSTIM_Handle_t* pstSTIM, uint16_t u16IntEn);

描述

使能指示的 STIM 节拍和溢出中断。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
u16IntEn	指示将使能哪个中断，@ ref EN_STIM_INT_TICK_OVFLW_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_disable_tick_overflow_interrupt

函数原型

EN_ERR_STA_T rom_hw_stim_disable_tick_overflow_interrupt (stSTIM_Handle_t* pstSTIM, uint16_t u16IntDis);

描述

除能指示的 STIM 节拍和溢出中断。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
u16IntDis	指示将除能哪个中断，@ ref EN_STIM_INT_TICK_OVFLW_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

STIM 唤醒 & 状态 API

rom_hw_stim_enable_wakeup

函数原型

EN_ERR_STA_T rom_hw_stim_enable_wakeup (stSTIM_Handle_t* pstSTIM, uint16_t u16WakeupEn);

描述

使能指示的 STIM 唤醒中断。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
u16WakeupEn	指示将使能哪个唤醒中断，@ ref EN_STIM_INT_WAKEUP_EN_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_disable_wakeup

函数原型

EN_ERR_STA_T rom_hw_stim_disable_wakeup (stSTIM_Handle_t* pstSTIM, uint16_t u16WakeupDis);

描述

除能指示的 STIM 唤醒中断。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
u16WakeupDis	指示将除能哪个唤醒中断，@ ref EN_STIM_INT_WAKEUP_EN_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_start

函数原型

EN_ERR_STA_T rom_hw_stim_start (stSTIM_Handle_t* pstSTIM);

描述

启动 STIM 计数器。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_stop

函数原型

EN_ERR_STA_T rom_hw_stim_stop (stSTIM_Handle_t* pstSTIM);

描述

停止 STIM 计数器。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_get_work_status

函数原型

EN_ERR_STA_T rom_hw_stim_get_work_status (stSTIM_Handle_t* pstSTIM, uint8_t* pu8Status);

描述

获取 STIM 工作状态。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
pu8Status	指向保存工作状态的指针。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

STIM 计数器 & 比较

rom_hw_stim_clear_count

函数原型

EN_ERR_STA_T rom_hw_stim_clear_count (stSTIM_Handle_t* pstSTIM);

描述

清除指示的 STIM 计数器寄存器。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_set_count_overflow

函数原型

EN_ERR_STA_T rom_hw_stim_set_count_overflow (stSTIM_Handle_t* pstSTIM);

描述

软件通过将 STIM_COUNTER_REG 寄存器设置为 0xFFFF FFF0 来生成溢出事件。当相应的中断使能时，将发生 STIM 溢出中断。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_get_count

函数原型

EN_ERR_STA_T rom_hw_stim_get_count (stSTIM_Handle_t* pstSTIM, uint32_t* pu32Count);

描述

获取指示的 STIM 计数器寄存器值。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
pu32Count	指向保存 STIM 当前计数器值 [31:0] 的指针。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_set_prescale

函数原型

EN_ERR_STA_T rom_hw_stim_set_prescale (stSTIM_Handle_t* pstSTIM, uint16_t u16Prescale);

描述

为指示的 STIM 设置预分频值。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
enCh	STIM 比较通道，@ ref EN_STIM_CH_T。
u32Comp	STIM 比较计数器值 [31:0]。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_set_compare

函数原型

EN_ERR_STA_T rom_hw_stim_set_compare (stSTIM_Handle_t* pstSTIM, EN_STIM_CH_T enCh, uint32_t u32Comp);

描述

为指示的 STIM 设置一个 32-bit 比较值。当 STIM 计数器递增到与比较值相同的值时，在中断使能时将发生中断。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
enCh	STIM 比较通道，@ ref EN_STIM_CH_T。
u32Comp	STIM 比较计数器值 [31:0]。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_stim_get_compare

函数原型

EN_ERR_STA_T rom_hw_stim_get_compare (stSTIM_Handle_t* pstSTIM, EN_STIM_CH_T enCh, uint32_t* pu32Comp);

描述

获取指示的 STIM 比较寄存器 STIM_COMP_REG 值。

参数

参数	描述
pstSTIM	STIM 句柄，应为 STIM0 / STIM1。
enCh	STIM 比较通道，@ ref EN_STIM_CH_T。
pu32Comp	指向保存 STIM 比较计数器值的指针。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

STIM 范例代码

STIM 外设的主要函数已经在前面一节描述过了，接下来的小节将演示 STIM 的主要函数。

STIM 唤醒 & 中断范例代码

范例代码工程

此工程演示了使用 STIM0 通道 0 唤醒 CPU 并进入中断处理程序。

```
uint8_t u8Status = 0;
/* 1. Enable stim0 clock gate first. */
hw_crg_enable_clk_gate (STIM0_CLK_GATE);
/* 2. Clear stim counter if needed. */
hw_stim_work_status (STIM0, &u8Status);
if (STIM_IS_WORKING == u8Status)
{
    hw_stim_stop (STIM0);
    hw_stim_clear_count (STIM0);
}
/* 3. Set stim prescale to 0(default). */
hw_stim_set_prescale (STIM0, 0);
/* 4. Configure STIM0 Channel0 Interrupt Handler*/
g_periIrqFuncTable[STIM0_IRQ0] = stim_stim0_ch0_handler;
NVIC_ClearPendingIRQ (STIM0_IRQ0);
NVIC_SetPriority (STIM0_IRQ0, 3);
NVIC_EnableIRQ (STIM0_IRQ0);
/* 5. Configure STIM0 interrupt and compare value. */
/* 5.1 Disable and clear interrupt flag*/
hw_stim_disable_wakeup (STIM0, STIM_INT_WAKEUP_EN_MASK);
hw_stim_disable_interrupt (STIM0, STIM_INT_MASK);
hw_stim_clear_interrupt_flag (STIM0, STIM_INT_TICK);
hw_stim_disable_tick_overflow_interrupt (STIM0, STIM_INT_TICK_OVFLW_MASK);
/* 5.2 Enable interrupt and wake up*/
hw_sys_ctrl_enable_peri_int (SYS_CTRL_MP, STIM0_IRQ0);
hw_stim_enable_wakeup (STIM0, STIM_CH0_INT_MATCH_WAKEUP);
hw_stim_enable_interrupt (STIM0, STIM_CH0_INT_MATCH);
/* 5.3 Set the compare0 channel0 to 200ms, the macro definition is used here to
    convert ms to a counter value */
hw_stim_set_compare(STIM0, STIM_CH0, STIM_MS_TO_CNT(LPWR_CLOCK_32K_HZ, 200));
/* 6. Configure stim0 channel0 wakeup source. */
hw_pmu_set_wakeup_source (0, LUT_TRIG_ID_OTHER, LUT_TRIG_ID_STIM0_CH0, LUT_
    STIM0_CH0_ACT);
/* 7. Start stim0 counter. */
hw_stim_start(STIM0);
```

STIM 中断处理程序范例代码

```
void stim_stim0_ch0_handler (void)
{
    uint32_t count = 0;
    uint16_t u16Flag;
    /* Get stim0 channel0 interrupt flag and clear it. */
    hw_stim_get_interrupt_flag (STIM0, &u16Flag);
    u16Flag &= (STIM_CH0_INT_MATCH | STIM_INT_TICK | STIM_INT_OVERFLOW);
    hw_stim_clear_interrupt_flag(STIM0, u16Flag);
    /* Get STIM0 Counter and print it. */
    hw_stim_get_count(STIM0, &count);
    PRINTF("STIM0 Counter is : %x", count);
}
```

STIM 溢出中断范例代码

此工程演示了如何使用 STIM0 溢出来唤醒 CPU 并进入中断处理程序。这里仍然使用前一节的中断处理程序。

范例代码工程

```
/* 1. Enable stim0 clock gate first. */  
hw_crg_enable_clk_gate (STIM0_CLK_GATE);  
/* 2. Configure Handler and Enable NVIC */  
g_periIrqFuncTable[STIM0_IRQ0] = stim_stim0_ch0_handler;  
NVIC_ClearPendingIRQ (STIM0_IRQ0);  
NVIC_SetPriority (STIM0_IRQ0, 3);  
NVIC_EnableIRQ (STIM0_IRQ0);  
/* 3. Configure wake up source */  
hw_pmu_set_wakeup_source (n, LUT_TRIG_ID_OTHER, LUT_TRIG_ID_STIM0_CH0, LUT_  
    STIM0_CH0_ACT);  
/* 4. Enable stim0 channel 0 overflow interrupt and wakeup system. */  
hw_stim_enable_wakeup (STIM0, STIM_INT_OVERFLOW_WAKEUP);  
hw_stim_enable_interrupt (STIM0, STIM_INT_OVERFLOW);  
hw_stim_enable_tick_overflow_interrupt (STIM0, STIM_CH0_INT_OVFLW);  
/* 5. Trigger stim counter to 0xFFFFFFFF0 */  
hw_stim_set_count_overflow(STIM0);  
/* 6. Start STIM. */  
hw_stim_start (STIM0);
```

5 实时时钟 (RTC)

简介

该单片机的实时时钟 (RTC) 外设依赖低频时钟源 (LCLK) 上提供了一个通用的低功耗定时器。当主电源正常运行时，主电源为 RTC 外设供电。当主电源断开时，RTC 外设可通过外部锂电池供电。在电源切换过程中，RTC 的数据始终保存在 RTC 的备份域中。

特性

- 支持低速 32 kHz 时钟作为时钟源
- 支持 40-bit 计数器，12-bit 预分频
- 支持 3 种类型的中断：溢出 / 比较 / 节拍
- 支持 4 通道中断和睡眠唤醒：
 - 通道 0：比较 0 / 节拍 / 溢出
 - 通道 1：比较 1 / 节拍 / 溢出
 - 通道 2：比较 2 / 节拍 / 溢出
 - 通道 3：比较 3 / 节拍 / 溢出
- 支持启动、停止、清除触发器 (Trig) 信号控制
- 独立支持 CPU 中断和 PMU 唤醒中断
- 支持配置成功标志检查

注：RTC 外设依赖低频时钟源提供频率。使用频率稳定的时钟源可以获得更好的时钟精度。

功能描述

该单片机 RTC 的定时器是一个只能向上计数的 40-bit 计数器。它采用三个时钟源，分别是：

- 128 个高速外部时钟：HSE / 128
- 内部时钟：LSI
- 低速外部时钟：LSE

主电源关闭会影响 HSE 分频和 LSI 时钟源，无法保证电源关闭时 RTC 的正常工作。因此，RTC 通常采用低速外部时钟源 LSE。下图为该单片机 RTC 参考设计原理图。

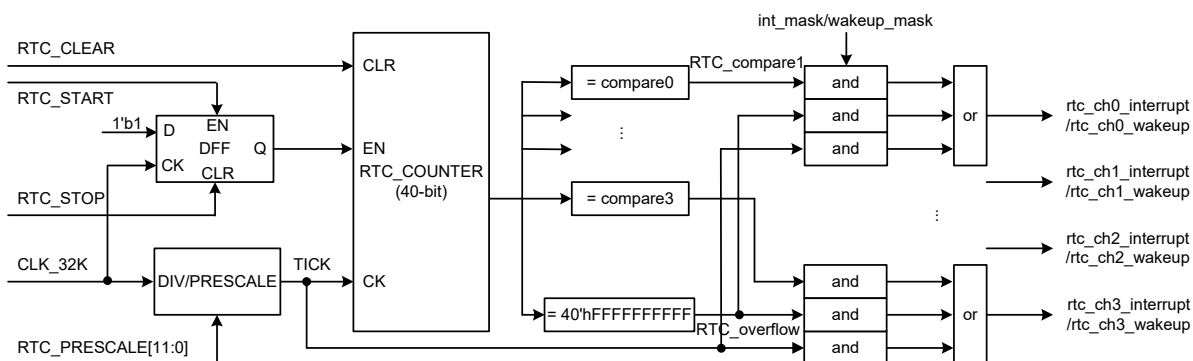


图 8. RTC 参考设计原理图

RTC 电源结构

为了保证掉电情况下的正常工作，该单片机 RTC 由主电源 VDDRTC 和备用电源 VDDRTC_BACKUP 供电。下图为该单片机 RTC 电源参考设计原理图。

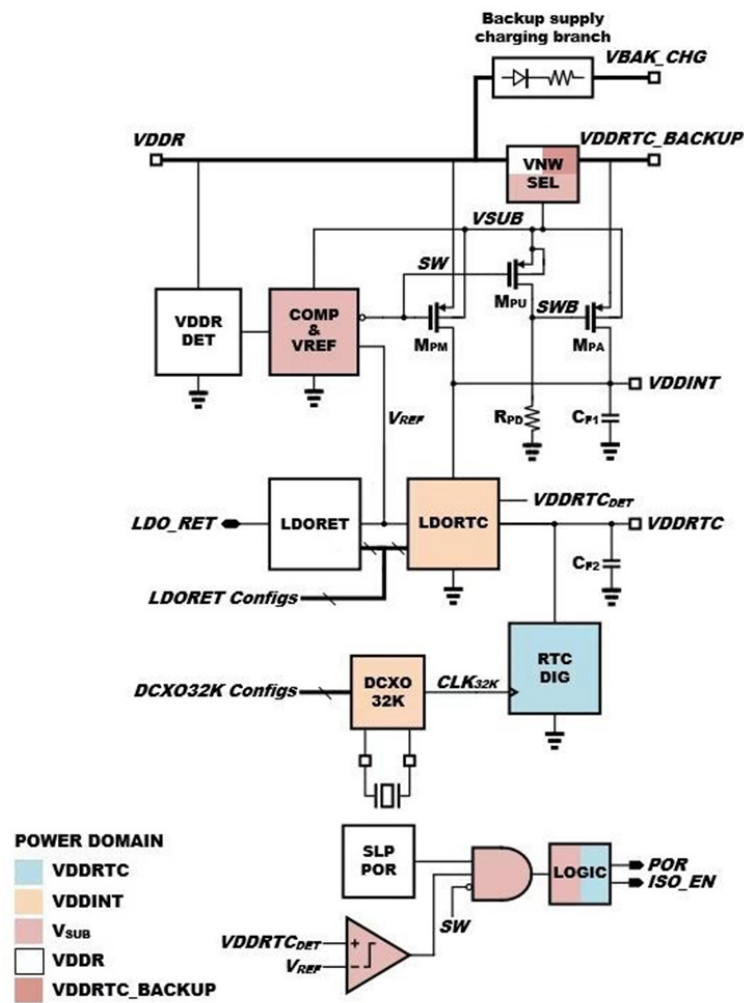


图 9. RTC 电源参考设计原理图

时钟精准度和预分频

该单片机的 RTC 外设使用 32768 Hz 时钟源，最小时间分辨率为 30.517 μ s (一个节拍间隔)。下表是不同分频系数下每个节拍的时间间隔。溢出时间是计数器从 0 计数到 0xFFFF FFFF 所花费的时间。

PRESCALE	节拍间隔	溢出时间
0	30.517 μ s	9315 小时
$2^8 - 1$	7812.5 μ s	99420 天
$2^{12} - 1$	125 ms	1590728 天

设 PRECALE 为分频系数，每个节拍的时间间隔可计算如下：

$$T_{RTC} = (\text{PRESCALER} + 1) / 32768, 0 \leq \text{PRESCALER} \leq 2^{12} - 1$$

注：该单片机的 PRECALE 寄存器配置需要一个触发器 (Trig) 才能生效。

计数器、节拍、溢出和比较

该单片机 RTC 外设的计数器寄存器长度为 40-bit，可同时配置比较、溢出和节拍中断或唤醒 4 个通道。下图显示了计数器工作流程。当 RTC 工作时，计数器根据时钟信号上升沿的数量而增加并触发节拍中断。当计数器寄存器值达到比较 (n) 寄存器值时触发比较 (n) 中断；当计数器寄存器满 (0xFF FFFF FFFF) 时，溢出中断将被触发，计数器值将被清零，如下图所示。

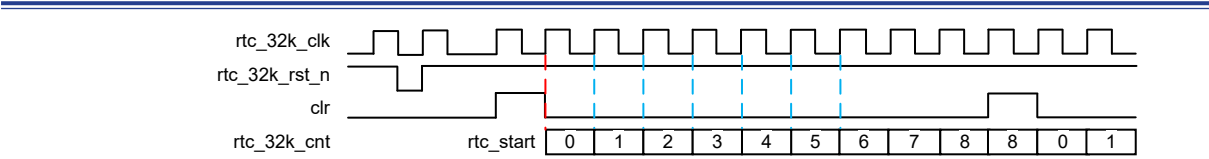


图 10. 计数器工作流程

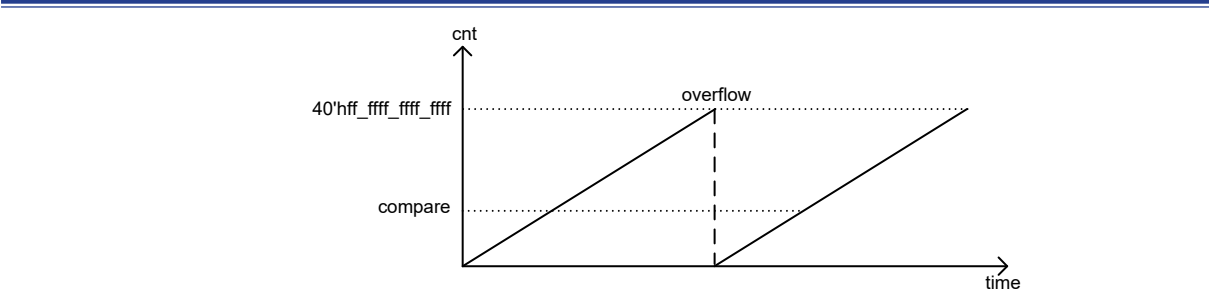


图 11. 计数器触发溢出

注：单片机默认除能节拍、溢出和比较 (n) 中断，使用时必须将其使能。

RTC API

RCT 配置 API

rom_hw_rtc_set_rtc_clk_src

函数原型

EN_ERR_STA_T rom_hw_rtc_set_rtc_clk_src (EN_RTC_CLK_SRC_T enSrc);

描述

选择 RTC_CLK 时钟源。

参数

参数	描述
enSrc	RTC 时钟源选择, @ ref EN_RTC_CLK_SRC_T。

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

rom_hw_rtc_set_prescale

函数原型

EN_ERR_STA_T rom_hw_rtc_set_prescale (uint16_t u16Prescale);

描述

设置 RTC 时钟的预分频值。

参数

参数	描述
u16Prescale	预分频值。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_set_ldo_ret_output_voltage

函数原型

EN_ERR_STA_T rom_hw_rtc_set_ldo_ret_output_voltage (EN_RTC_LDO_RET_VOLT_T enVolt);

描述

配置 VDD_RET 输出电压。当 CPU 进入睡眠模式时，保存 LDO 将工作并输出 VDD_RET 电压以保持系统处于睡眠模式。

参数

参数	描述
enVolt	配置 LDO_RET (VDDRTC) 输出电压，@ ref EN_RTC_LDO_RET_VOLT_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_pdw_reset

函数原型

EN_ERR_STA_T rom_hw_rtc_pdw_reset (void);

描述

复位 RTC 模块电源。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

RTC 中断 API

rom_hw_rtc_get_interrupt_flag

函数原型
EN_ERR_STA_T rom_hw_rtc_get_interrupt_flag (uint16_t *pul6Msk);

描述
通过读 RTC_INT_FLAG 寄存器来获取指示的 RTC 中断标志 (状态)。

参数

参数	描述
pu16Flag	指示将读取哪个中断标志。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_clear_interrupt_flag

函数原型
EN_ERR_STA_T rom_hw_rtc_clear_interrupt_flag (uint16_t u16Msk)

描述
通过写 RTC_INT_CLR 寄存器来清除指示的 RTC 中断标志 (状态)。

参数

参数	描述
u16Msk	指示将清除哪个标志。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_enable_interrupt

函数原型
EN_ERR_STA_T rom_hw_rtc_enable_interrupt (uint16_t u16IntEn);

描述
通过写具有指定位的 RTC_INT_EN 寄存器来使能指示的 RTC 中断。

参数

参数	描述
u16IntEn	指示将使能哪个中断。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_disable_interrupt

函数原型
EN_ERR_STA_T rom_hw_rtc_disable_interrupt (uint16_t u16IntDis);

描述
通过写具有指定位的 RTC_INT_EN 寄存器来除能指示的 RTC 中断。

参数

参数	描述
u16IntDis	指示将除能哪个中断。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_enable_tick_overflow_interrupt

函数原型

EN_ERR_STA_T rom_hw_rtc_enable_tick_overflow_interrupt (uint16_t u16IntEn);

描述

使能指示的 RTC 节拍和溢出中断。

参数

参数	描述
u16IntEn	指示将使能哪个中断，@ ref EN_RTC_INT_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_disable_tick_overflow_interrupt

函数原型

EN_ERR_STA_T rom_hw_rtc_disable_tick_overflow_interrupt (uint16_t u16IntDis);

描述

除能指示的 RTC 节拍和溢出中断。

参数

参数	描述
u16IntDis	指示将除能哪个中断，@ ref EN_RTC_INT_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_enable_wakeup

函数原型

EN_ERR_STA_T rom_hw_rtc_enable_wakeup (uint16_t u16WakeupEn);

描述

使能指示的 RTC 唤醒中断。在相应的中断使能后，CPU 将从睡眠模式中唤醒。

参数

参数	描述
u16WakeupEn	指示将使能哪个唤醒中断，@ ref EN_RTC_INT_WAKEUP_EN_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_disable_wakeup

函数原型
EN_ERR_STA_T rom_hw_rtc_disable_wakeup (uint16_t u16WakeupDis);

描述
除能指示的 RTC 唤醒中断。

参数

参数	描述
u16WakeupDis	指示将除能哪个唤醒中断, @ ref EN_RTC_INT_WAKEUP_EN_T。

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

RTC 状态 API

rom_hw_rtc_start

函数原型
EN_ERR_STA_T rom_hw_rtc_start (void);

描述
启动 RTC 计数器。

参数
无

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

rom_hw_rtc_stop

函数原型
EN_ERR_STA_T rom_hw_rtc_stop (void);

描述
停止 RTC 计数器。

参数
无

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

rom_hw_rtc_get_work_status

函数原型
EN_ERR_STA_T rom_hw_rtc_get_work_status (uint8_t* pu8Status);

描述
获取 RTC 工作状态。

参数

参数	描述
pu8Status	指向保存 RTC 工作状态的指针。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_get_wakeup_status

函数原型

EN_ERR_STA_T rom_hw_rtc_get_wakeup_status (uint8_t* pu8Status);

描述

获取 RTC 唤醒源工作状态，PMU 处于停止状态时才会进入低功耗模式。

参数

参数	描述
pu8Status	指向保存 RTC 唤醒源工作状态的指针，@ ref EN_RTC_WAKEUP_STATUS_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_set_sw_flag

函数原型

EN_ERR_STA_T rom_hw_rtc_set_sw_flag (uint32_t u32SwFlag);

描述

配置软件标志。检查 RTC 模块是否初始化。当 VDDR 掉电时，此标志将被保留。

参数

参数	描述
pu8Status	软件标志。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

RTC 计数器 & 比较

rom_hw_rtc_get_count

函数原型

EN_ERR_STA_T rom_hw_rtc_get_count (uint8_t* pu8Hi, uint32_t* pu32Lo);

描述

获取 RTC 当前计数器值。

参数

参数	描述
pu8Hi	指向保存 RTC 当前计数器值 [39:32]。
pu32Lo	指向保存 RTC 当前计数器值 [31:0]。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_get_count64

函数原型

EN_ERR_STA_T rom_hw_rtc_get_count64 (uint64_t* pu64Count);

描述

获取 RTC 当前计数器值。

参数

参数	描述
pu64Count	指向保存 RTC 当前计数器值 [39:0]。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_clear_count

函数原型

EN_ERR_STA_T rom_hw_rtc_clear_count (void);

描述

清除 RTC 计数器。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_set_count_overflow

函数原型

EN_ERR_STA_T rom_hw_rtc_set_count_overflow (void);

描述

将 RTC 计数器设置为 0xFFFFFFFFF0，可以快速触发溢出中断。

参数

无

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_rtc_set_compare

函数原型

EN_ERR_STA_T rom_hw_rtc_set_compare(EN_RTC_CH_T enCh, uint8_t u8Hi, uint32_t u32Lo);

描述

设置 RTC 比较计数器值。当 RTC 递增到与比较计数器值相同的值时，RTC 将在中断使能时产生中断。

参数

参数	描述
enCh	RTC 比较通道, @ ref EN_RTC_CH_T。
u8Hi	RTC 比较计数器值 [39:32]。
u32Lo	RTC 比较计数器值 [31:0]。

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

rom_hw_rtc_set_compare64

函数原型

EN_ERR_STA_T rom_hw_rtc_set_compare64 (EN_RTC_CH_T enCh, uint64_t u64Compare);

描述

设置 RTC 比较计数器值。当 RTC 递增到与比较计数器值相同的值时, RTC 将在中断使能时产生中断。

参数

参数	描述
enCh	RTC 比较通道, @ ref EN_RTC_CH_T。
u64Compare	RTC 比较计数器值 [39:0]。

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

rom_hw_rtc_get_compare

函数原型

EN_ERR_STA_T rom_hw_rtc_get_compare (EN_RTC_CH_T enCh, uint8_t* pu8Hi, uint32_t* pu32Lo);

描述

获取 RTC 比较计数器值。

参数

参数	描述
enCh	RTC 比较通道, @ ref EN_RTC_CH_T。
pu8Hi	指向保存 RTC 比较计数器值 [39:32]。
pu32Lo	指向保存 RTC 比较计数器值 [31:0]。

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

rom_hw_rtc_get_compare64

函数原型

EN_ERR_STA_T rom_hw_rtc_get_compare64 (EN_RTC_CH_T enCh, uint64_t* pu64Compare);

描述

获取 RTC 比较计数器值。

参数

参数	描述
enCh	RTC 比较通道, @ ref EN_RTC_CH_。
pu64Compare	指向保存 RTC 比较计数器值 [39:0]。

返回

EN_ERR_STA_T	函数返回状态, 参考 EN_ERR_STA_T 枚举定义。
--------------	-------------------------------

RTC 范例代码

RTC 外设的主要函数已经在上面的小节中描述过了, 接下来一节将演示 RTC 的主要函数。

RTC 开始工作范例代码

当使用 RTC 外设时, 需要设置 VDDRTC 电压并启动时钟。此工程演示了 RTC 正常工作时的时钟设置。

```
/* Open the system APB clock & RC clock */
rom_hw_crg_enable_clk_gate(CRG_RTC_APB_CLK_GATE);
rom_hw_crg_enable_clk_gate(CRG_RTC_RC_LCLK_GATE);
/* Set the voltage of the VDDRTC */
patch_hw_rtc_set_ldo_ret_output_voltage(EN_RTC_LDO_RET_950mV);
/* Set RC 32K as the RTC clock source */
patch_hw_rtc_set_rtc_clk_src(EN_RTC_CLK_SRC_RC_LCLK);
/* RTC start to work */
patch_hw_rtc_start();
```

RTC 中断范例代码

此范例工程旨在测试 RTC 是否可以通过获取当前计数器值并将比较 1 设置为当前计数器值 +1000 来响应比较中断。范例如下:

RTC 中断范例代码

```
void RTC_Interrupt (void)
{
    uint64_t now_counter = 0;
    uint64_t compare_val = 1000;
    /*Stop RTC counter*/
    patch_hw_rtc_stop();
    /*Set prescale value*/
    patch_hw_rtc_set_prescale(0);
    /*Enable RTC channel 1 interrupt*/
    patch_hw_rtc_enable_interrupt(RTC_CH1_INT_MATCH);
    /*Get counter value*/
    patch_hw_rtc_get_count64(&now_counter);
    /*Set compare1 value, Interrupt will be triggered after 1000 counters*/
    patch_hw_rtc_set_compare64(RTC_CH1, now_counter + compare_val);
    /*Configure for RTC interrupt handler*/
    g_periIrqFuncTable[RTC_CH1_IRQ] = RTC_Interrupt_Handler;
    /*Enable system interrupt*/
}
```



```

NVIC_ClearPendingIRQ (RTC_CH1_IRQ);
NVIC_SetPriority (RTC_CH1_IRQ, 3);
NVIC_EnableIRQ (RTC_CH1_IRQ);
__enable_irq();
/*RTC start to work*/
patch_hw_rtc_start();
}

```

RTC 中断处理程序范例代码

```

void RTC_Interrupt_Handler(void)
{
    uint16_t int_flag = 0;
    uint64_t compare_val = 0;
    uint64_t counter_val = 0;
    /*Clear RTC Interrupt flag*/
    patch_hw_rtc_get_interrupt_flag(&int_flag);
    patch_hw_rtc_clear_interrupt_flag(int_flag);
    /*Gets the value of the current counter*/
    patch_hw_rtc_get_count64(&counter_val);
    PRINTF("The counter value is: %llx \n", counter_val);
}

```

唤醒范例代码

下面的范例代码显示了 RTC 如何在睡眠模式下唤醒 CPU。

```

void RTC_Wake(void)
{
    uint64_t now_counter = 0;
    uint64_t compare_val = 1000;
    patch_hw_rtc_stop();
    patch_hw_rtc_set_prescale(0);
    patch_hw_rtc_enable_interrupt(RTC_CH1_INT_MATCH);
    patch_hw_rtc_get_count64(&now_counter);
    patch_hw_rtc_set_compare64(RTC_CH1, now_counter + compare_val);
    /*Set RTC CH1 as the wake source*/
    rom_hw_pmu_set_wakeup_source (9, LUT_TRIG_ID_OTHER, LUT_TRIG_ID_RTC_CH1,
    LUT_ACT_PD_SYS_ON | LUT_ACT_MP_IRQ_EN);
    /*Enable wakeup*/
    patch_hw_rtc_enable_wakeup(RTC_CH1_INT_MATCH_WAKEUP);
    patch_hw_rtc_start();
}

```

6 通用异步收发器 (UART)

简介

UART 符合 16550 工业标准，用于与外设数据集进行串行通信。数据通过 APB 总线从主机 (CPU / DMA) 写入到 UART，然后转换为串行形式并传输到目标设备。串行数据也由 UART 接收并存储，供主机 (CPU / DMA) 回读。这些 UART 都支持硬件流控制信号 (RTS 和 CTS)。

特性

- 8-byte 发送和接收 FIFO
- 硬件流控制支持 (CTS / RTS)
- 基于 16550 工业标准的功能
- 可编程字符属性，如每个字符的数据位数 (5 ~ 8) 可选
- 奇偶校验位 (奇 / 偶 / 粘性 / 不选择) 和停止位 (1、1.5 或 2)
- 线中止错误生成和检测
- 支持 RX 超时中断
- 可编程串行数据波特率

功能描述

该单片机 UART 功能主要包括：UART 初始化、UART 配置、串行数据接收 / 发送、中断配置等。

UART API

UART 配置 API

rom_hw_uart_deinit

函数原型

EN_ERR_STA_T rom_hw_uart_deinit(stUART_Handle_t* pstUART);

描述

初始化 UART 外设。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_init

函数原型
EN_ERR_STA_T rom_hw_uart_init(stUART_Handle_t* pstUART, stUartInit_t* pstInitType);

描述
根据 stUartInit_t 中指定的参数来初始化 UART 模式，并创建相关句柄。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
pstInitType	指向 stUartInit_t 结构的指针。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_set_baudrate

函数原型
EN_ERR_STA_T rom_hw_uart_set_baudrate(stUART_Handle_t* pstUART, uint32_t u32BaudRate);

描述
设置 UART 波特率。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
u32BaudRate	波特率。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_set_data_sizes

函数原型
EN_ERR_STA_T rom_hw_uart_set_data_sizes(stUART_Handle_t* pstUART, EN_UART_DATA_SIZE_T enSize);

描述
设置 UART 数据位。数据位向上兼容，高数据位可以正确接收低数据位的数据。databit5 最大值为 31，databit6 最大值为 63，databit7 最大值为 127，databit8 最大值为 255。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
enSize	UART 数据大小，@ ref EN_UART_DATA_SIZE_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_set_stop_sizes

函数原型

EN_ERR_STA_T rom_hw_uart_set_stop_sizes(stUART_Handle_t* pstUART, EN_UART_STOP_SIZE_T enStopbits);

描述

设置 UART 停止位。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
enStopbits	UART 停止位，@ ref EN_UART_STOP_SIZE_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_set_parity

函数原型

EN_ERR_STA_T rom_hw_uart_set_parity(stUART_Handle_t* pstUART, EN_UART_PARITY_BITS_T enParity);

描述

设置 UART 奇偶校验。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
enParity	UART 奇偶校验位，@ ref EN_UART_PARITY_BITS_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_set_mode

函数原型

EN_ERR_STA_T rom_hw_uart_set_mode(stUART_Handle_t* pstUART, EN_UART_ENDIAN_T enMode);

描述

设置 UART 大小端 (Endian) 模式。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
enMode	UART 大小端模式，@ ref EN_UART_ENDIAN_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_line_break_enable

函数原型

EN_ERR_STA_T rom_hw_uart_line_break_enable(stUART_Handle_t* pstUART, EN_UART_LB_EN_T enEn);

描述

配置 UART 发送线中止错误功能。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
enEn	使能或除能 TX 线中止错误功能。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_set_rx_timeout

函数原型

EN_ERR_STA_T rom_hw_uart_set_rx_timeout(stUART_Handle_t* pstUART, uint8_t u8SymbolNum);

描述

设置 UART RX 超时时间。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
u8SymbolNum	超时值，单位：ms

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_flow_enable

函数原型

EN_ERR_STA_T rom_hw_uart_flow_enable(stUART_Handle_t* pstUART, EN_UART_FLOW_EN_T enEn);

描述

UART 配置流功能。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
enEn	使能或除能流功能，@ ref EN_UART_FLOW_EN_T。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_set_rts_thld

函数原型
EN_ERR_STA_T rom_hw_uart_set_rts_thld(stUART_Handle_t* pstUART, uint8_t u8Thld);
描述
设置 RTS 阈值。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
u8Thld	UART RTS 阈值。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

UART 中断 API

rom_hw_uart_get_interrupt_flag

函数原型
EN_ERR_STA_T rom_hw_uart_get_interrupt_flag(stUART_Handle_t* pstUART, uint16_t* pul6Falg);
描述
获取 UART 中断标志 (状态)。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
pul6Falg	指示将读取哪个中断标志。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_enable_interrupt

函数原型
EN_ERR_STA_T rom_hw_uart_enable_interrupt(stUART_Handle_t* pstUART, uint16_t u16Msk);
描述
设置 UART 中断使能。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
u16Msk	指示将使能哪个中断。 Bit 为 1 表示使能 Bit 为 0 表示无影响

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_disable_interrupt

函数原型
EN_ERR_STA_T rom_hw_uart_disable_interrupt(stUART_Handle_t* pstUART, uint16_t u16Msk);
描述
设置 UART 中断除能。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
u16Msk	指示将除能哪个中断。 Bit 为 1 表示除能 Bit 为 0 表示无影响

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_clear_interrupt_flag

函数原型
EN_ERR_STA_T rom_hw_uart_clear_interrupt_flag(stUART_Handle_t* pstUART, uint16_t u16Msk);
描述
清除指示的 UART 中断标志。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
u16Msk	指示将清除哪个标志。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

UART FIFO API

rom_hw_uart_set_txfifo_thld

函数原型
EN_ERR_STA_T rom_hw_uart_set_txfifo_thld(stUART_Handle_t* pstUART, uint8_t u8Thld);
描述
设置 TX FIFO 低于阈值级别。当 TX FIFO 低于该阈值时将触发中断。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
u8Thld	UART TX FIFO 阈值。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_set_rxfifo_thld

函数原型
EN_ERR_STA_T rom_hw_uart_set_rxfifo_thld(stUART_Handle_t* pstUART, uint8_t u8Thld);
描述
设置 RX FIFO 超过阈值级别。当 RX FIFO 超过该阈值时触发中断。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
u8Thld	UART RX FIFO 阈值。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_clear_rxfifo

函数原型
EN_ERR_STA_T rom_hw_uart_clear_rxfifo(stUART_Handle_t* pstUART);
描述
配置 UART 清除 FIFO 功能。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_get_txfifo_cnt

函数原型
EN_ERR_STA_T rom_hw_uart_get_txfifo_cnt(stUART_Handle_t* pstUART, uint8_t* pu8Cnt);
描述
获取 UART TX FIFO 计数。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
pu8Cnt	TX FIFO 计数值。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_get_rxfifo_cnt

函数原型
EN_ERR_STA_T rom_hw_uart_get_rxfifo_cnt(stUART_Handle_t* pstUART, uint8_t* pu8Cnt);
描述
获取 UART RX FIFO 计数。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
pu8Cnt	RX FIFO 计数值。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

UART RX/TX API

rom_hw_uart_get_byte

函数原型

EN_ERR_STA_T rom_hw_uart_get_byte(stUART_Handle_t* pstUART, uint8_t* pu8Data);

描述

在非阻塞模式下获取一个字节的數據。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
pu8Data	指向数据缓冲器的指针。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_receive

函数原型

EN_ERR_STA_T rom_hw_uart_receive(stUART_Handle_t* pstUART, uint8_t* pu8Buf, uint8_t u8BufSize);

描述

在非阻塞模式下接收一定数量的数据。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
pu8Data	指向数据缓冲器的指针。
u16Len	要接收的数据量。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_send_byte

函数原型

EN_ERR_STA_T rom_hw_uart_send_byte(stUART_Handle_t* pstUART, uint8_t u8Data);

描述

当 TX FIFO 未滿时，通过 UART 传输单个数据，发送时阻塞。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
u8Data	要传输的数据。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

rom_hw_uart_transmit

函数原型

EN_ERR_STA_T rom_hw_uart_transmit(stUART_Handle_t* pstUART, uint8_t* pu8Buf, uint16_t u16Len);

描述

在阻塞模式下传输一定数量的数据。

参数

参数	描述
pstUART	UART 句柄，应为 UART0 / UART1 / UART2。
pu8Data	指向数据缓冲器的指针。
u16Len	要发送的数据量。

返回

HW status	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
-----------	------------------------------

UART 范例代码

使用 UART1 的应用范例

初始化串口引脚

要初始化串口引脚，步骤 1：定义串口的 GPIO，步骤 2：初始化串口时钟，步骤 3：设置串口 GPIO。代码如下：

```
#define GPIO_PIN_UART1_TX(GPIO_PIN_10)
#define GPIO_PIN_UART1_RX(GPIO_PIN_11)
static void uart_config(void)
{
    // Initialize the uart clock
    rom_hw_crg_enable_clk_gate(CRG_UART1_CLK_GATE);
    // Initialize the serial port pin
    patch_hw_gpio_set_pin_pid(GPIOA, GPIO_PIN_UART1_TX, PID_UART1_TXD);
    patch_hw_gpio_set_pin_pid(GPIOB, GPIO_PIN_UART1_RX, PID_UART1_RXD);
    // Initializing the serial port
    uart_init(UART1);
}
```

初始化 UART

要初始化 UART，步骤 1：初始化 UART 句柄，用于注册中断函数。步骤 2：初始化串口结构，设置波特率、FIFO、数据位和奇偶校验。步骤 3：设置中断接收模式。步骤 4：注册中断服务函数。步骤 5：使能超时中断。代码如下：

```
static void uart_init(pstUART_Handle_t UART_Handle)
{
    // Initialize the uart handle
    pstUART_Handle_t g_pu32UartHandle = UART_Handle;
    // Initialize the initial uart
    struct stUartInit_t stUartInit;
    // Set the communication baud rate
    stUartInit.u32UartBaudRate = UART_BAUDRATE_921600;
    // Set the default configuration of the serial port
    // TxFifoThld and RxFifoThld is 8, StopBit is 1, 8bit, no parity, little-
    // ending
    stUartInit.unUartCfg.u32UartCfg = UART_INIT_DEFAULT(UART_PARITY_NONE);
    // Write the default configuration of the serial port
    rom_hw_uart_init(g_pu32UartHandle, &stUartInit);
    // Receive data with a timeout interrupt
    rom_hw_uart_set_rx_timeout(g_pu32UartHandle, 0xFF);
    // Register the interrupt service function
    uart_nvic_init(UART_Handle);
    // Enable timeout interrupts
    rom_hw_uart_enable_interrupt(g_pu32UartHandle, UART_INT_RX_TIMEOUT);
}
```

注册中断服务函数

当接收到 UART 数据时，将进入 UART 中断回调函数。中断服务函数如下：

```
static void uart_nvic_init(pstUART_Handle_t UART_Handle)
{
    // Clear the interrupt status flag
    rom_hw_uart_clear_interrupt_flag(UART_Handle, 0x1fff);
    // Register the interrupt callback function.
    g_periIrqFuncTable[UART1_IRQ] = (PERI_IRQ_FUNC)UART1_IRQ_Handler;
    // Clear the interrupt
    NVIC_ClearPendingIRQ (UART1_IRQ);
    // Set interrupt priority
    NVIC_SetPriority (UART1_IRQ, 0x3);
    // enable interrupt
    NVIC_EnableIRQ (UART1_IRQ);
    // Here is divided into cortex_M33 and cortex_M0+
    #ifdef MAIN_PROCESSOR
    rom_hw_sys_ctrl_enable_peri_int(SYS_CTRL_MP, UART1_IRQ);
    #else
    rom_hw_sys_ctrl_enable_peri_int(SYS_CTRL_CP, UART1_IRQ);
    #endif
    __enable_irq();
}
```

接收数据

数据通常在 UART 中断回调函数中接收。当判断数据可用时，就可以接收数据。接收数据代码如下：

```
static void UART1_IRQ_Handler(void)
{
    static uint8_t u8Cnt = 0;
    // Determines if data is available.
    while(UART1->UART_RXFIFO_CNT & 0x1F)
    {
        // Save data
        g_PCMsg.Data[u8Cnt++] = UART1 → UART_RX_FIFO & 0xFF;
    }
    // Clear interruption flags.
    rom_hw_uart_clear_interrupt_flag(UART1, UART_INT_RX_TIMEOUT);
}
```

发送数据

此函数可以用于单字节发送。详细参数请参考前一节。

```
rom_hw_uart_send_byte (stUART_Handle_t* pstUART, uint8_t u8Data);
```

7 IR 模块

简介

该单片机可支持 IR 编码和 IR 解码，两者均由芯片的 GTIM 部分完成。

IR 编码：芯片通过 GTIM 实现 IR 编码，生成的 IR 信号通过连接的 IR 二极管发送。

IR 解码：芯片包含 IR 接收电路，可以接收 IR 信号，然后通过软件实现 IR 解码。

特性

- 1 通道内部 IR 传输电路
- 4 通道 IR 编码
 - IR 编码信号可通过 GPIO 输出
 - 通过在 GPIO 上连接 IR 传输电路，可实现 4 通道 IR 发射
- 4 通道 IR 触发解码功能
 - 支持 cnta/cntb 解码 (支持 cnta 和 cntb 同时解码)
 - 支持组合 32-bit 解码

功能描述

IR 编码

它用于产生 IR 波形。

IR 波形由 GTIM 的 cnta 和 cntb 产生的两个 PWM 波形组合而成。

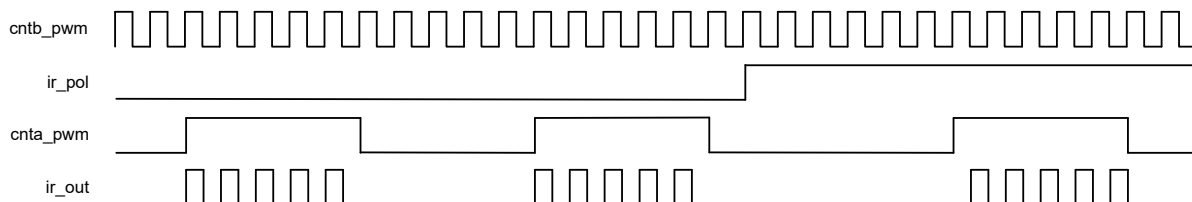


图 12. IR 编码

用 cntb 输出指定频率、占空比为 50 % 的 PWM 作为载波，用 cnta 输出指定 PWM 作为数据波形信号，然后用逻辑“与”将两个 PWM 信号合成所需的 IR 信号。

ir_pol 可用于对载波信号 cntb 波形进行反相，以满足特定需求。

注：该单片机有四个 IR 编码通道，但只有一个内部 IR 发送引脚。

单片机的 IR 编码信号可以通过 GPIO 输出，因此该单片机可以借助外部硬件设备实现四通道 IR 发射的效果。

IR 解码

它用于 IR 解码，将带载波的 IR 信号还原为电平信号。

每个 GTIM 包含两个通道 (通道 A 和通道 B)，每个通道中都有一个解码计数器。当波形变化时间在配置计数时间 (解码间隔) 内时，输出高电平。当波形变化时间超过配置时间时，输出低电平。(decode_gpout_pol 可以配置为反相输出)

两个通道也可以组合成一个 32-bit 通道使用。

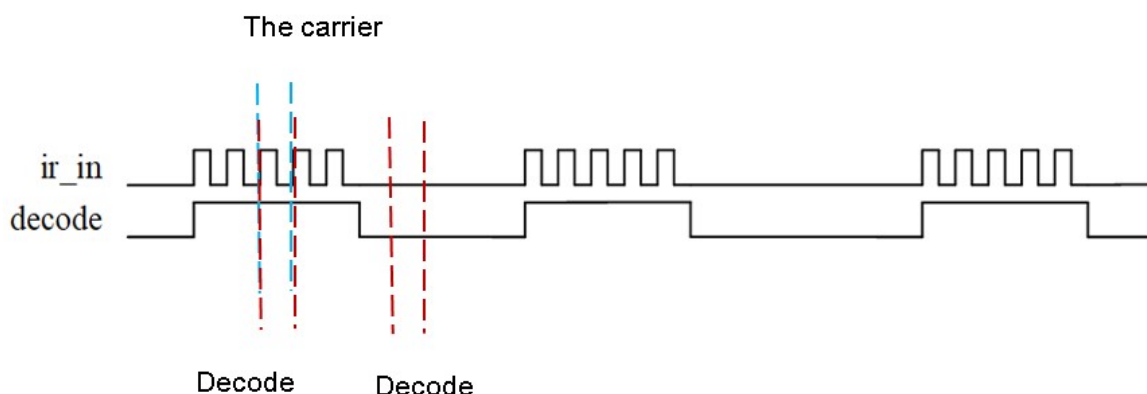


图 13. IR 解码

注：配置计数时间在代码配置中称为解码间隔。

当检测到输入信号为高电平时，解码信号将输出高电平，然后解码开始计数。如果在一段时间间隔内检测到输入 IR 信号的波形变化，则解码将继续输出高电平。如果在一段时间间隔内没有检测到波形变化，解码将立即拉低。

我们需要保证输入波形的一个变化周期时间在解码间隔内，并且解码间隔也尽可能接近输入波形的一个变化周期时间，以减少解码误差。

IR 模块 API

IR 编码 HW API

rom_hw_ir_set_send_path

函数原型

```
EN_ERR_STA_T rom_hw_ir_set_send_path (stTIMER_Handle_t* pstIR, EN_IR_SEND_PATH_T enPath);
```

描述

设置 IR 发送信号路径。

EN_IR_SEND_PATH_T 的定义如下：

```
typedef enum
{
    IR_SEND_DEFAULT_PATH = 0,
    IR_SEND_BY_IR_TX_PIN = 1,
    IR_SEND_BY_GPIO = 2,
    IR_SEND_BY_BOTH = 3,
} EN_IR_SEND_PATH_T;
```

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
enPath	IR 发送信号路径，参考 EN_IR_SEND_PATH_T 定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_ir_set_pwm_current_compare_and_polarity

函数原型

EN_ERR_STA_T rom_hw_ir_set_pwm_current_compare_and_polarity (stTIMER_Handle_t* pstIR, uint16_t u16Compare, EN_PWM_POL_T enPol)

描述

配置 IR TX 参数，包括电流、PWM 极性等。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
u16Compare	比较值。 范围为 0 ~ 65535。
enPol	PWM 极性，参考 EN_PWM_POL_T 定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_ir_pwm_set_next_compare

函数原型

EN_ERR_STA_T rom_hw_ir_pwm_set_next_compare (stTIMER_Handle_t* pstIR, uint16_t u16Compare, uint16_t u16LoCnt, uint16_t u16HiCnt);

描述

配置 IR TX 载波 PWM 周期和宽度。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
u16Compare	比较值。 范围为 0 ~ 65535。
u16HiCnt	比较高计数器。当计数器计数到高计数器值时，电平将翻转一次。 范围为 0 ~ 65535。
u16LoCnt	比较低计数器。当计数器计数到低计数器值时，电平将翻转一次。 范围为 0 ~ 65535。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_ir_enable

函数原型

EN_ERR_STA_T rom_hw_ir_enable (stTIMER_Handle_t* pstIR);

描述

使能 GPTM 工作在 IR 功能。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_ir_disable

函数原型

EN_ERR_STA_T rom_hw_ir_disable (stTIMER_Handle_t* pstIR);

描述

除能 GPTM IR 功能。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_ir_set_polarity

函数原型

EN_ERR_STA_T rom_hw_ir_set_polarity (stTIMER_Handle_t* pstIR, EN_IR_POL_T enPol);

描述

配置 IR TX 输出载波极性。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
enPol	IR 极性，参考 EN_PWM_POL_T 定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_ir_enable_channel

函数原型

EN_ERR_STA_T rom_hw_ir_enable_channel (stTIMER_Handle_t* pstIR, EN_IR_CH_T enCh);

描述

使能 IR 通道。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
enCh	IR 通道，参考 EN_IR_CH_T 枚举定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_ir_disable_channel

函数原型

EN_ERR_STA_T rom_hw_ir_disable_channel (stTIMER_Handle_t* pstIR, EN_IR_CH_T enCh);

描述

除能 IR 通道。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
enCh	IR 通道，参考 EN_IR_CH_T 枚举定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

IR 编码 HALAPI

rom_hal_ir_send_init

函数原型

EN_ERR_STA_T rom_hal_ir_send_init (stTIMER_Handle_t* pstIR, uint32_t u32Freq, uint16_t u16Duty, EN_IR_SEND_PATH_T enPath);

描述

初始化指示的 IR 定时器时钟。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
u32Frequency	载波的 PWM 输出频率，单位：Hz
u8Duty	载波的 PWM 占空比

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hal_ir_start_send

函数原型

EN_ERR_STA_T rom_hal_ir_start_send (stTIMER_Handle_t* pstIR);

描述

启动 IR 发送。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hal_ir_stop_send

函数原型

EN_ERR_STA_T rom_hal_ir_stop_send (stTIMER_Handle_t* pstIR);

描述

停止 IR 发送。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hal_ir_config_clock

函数原型

EN_ERR_STA_T rom_hal_ir_config_clock (stTIMER_Handle_t* pstIR, EN_IR_CH_T enCh, uint8_t u8Prescale);

描述

配置指示的通道预分频。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
enCh	IR 通道。
u8Prescale	输入时钟的分频，范围为 0 ~ 15。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hal_ir_send_next_signal_level

函数原型
EN_ERR_STA_T rom_hal_ir_send_next_signal_level (stTIMER_Handle_t* pstIR, unIR_SendSignalData_t* punIrData);

描述
配置 IR 下一周期比较值。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
punIrData	配置数据，参考 unIR_SendSignalData_t 枚举定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

IR 解码 HW API

rom_hw_timer_enable_decode

函数原型
EN_ERR_STA_T rom_hw_timer_enable_decode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述
使能定时器解码功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将使能哪个定时器，参考 EN_TIMER_CH_T 枚举定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_timer_disable_decode

函数原型
EN_ERR_STA_T rom_hw_timer_disable_decode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述
除能定时器解码功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将使能哪个定时器，参考 EN_TIMER_CH_T 枚举定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_timer_set_decode_pol

函数原型

EN_ERR_STA_T rom_hw_timer_set_decode_pol(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_DECODE_POL_T enPol);

描述

配置解码极性。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将使能哪个定时器，参考 EN_TIMER_CH_T 枚举定义。
enMode	解码极性，参考 EN_DECODE_POL_T 枚举定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_timer_set_decode_mode

函数原型

EN_ERR_STA_T rom_hw_timer_set_decode_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_DECODE_MODE_T enMode);

描述

配置解码模式。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将使能哪个定时器，参考 EN_TIMER_CH_T 枚举定义。
enMode	解码模式，参考 EN_DECODE_MODE_T 枚举定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hw_timer_set_decode_interval

函数原型

EN_ERR_STA_T rom_hw_timer_set_decode_interval(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint8_t u8Val);

描述

设置指示的定时器解码间隔。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将使能哪个定时器，参考 EN_TIMER_CH_T 枚举定义。
u8Val	解码间隔值。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

IR 解码 HAL API

rom_hal_ir_study_init

函数原型

EN_ERR_STA_T rom_hal_ir_study_init(stTIMER_Handle_t* pstIR, stDecodeInit_t* pstDecodeInit, stCapInit_t* pstCapInit);

描述

初始化 IR 学习模式。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
pstDecodeInit	解码初始化结构类型。 .u8Prescale: 输入时钟的分频，u8Prescale 的范围为 0 ~ 15。 .u8Mode: 解码模式，参考 EN_DECODE_MODE_T 枚举定义。 .u8Pol: 解码极性，参考 EN_DECODE_POL_T 枚举定义。 .u8Value: 解码间隔值。
pstCapInit	捕捉初始化结构类型。 .u8Prescale: 输入时钟的分频，u8Prescale 的范围为 0 ~ 15。 .u8Signal: 捕捉信号，@ ref EN_CAP_SIGNAL_T。 .u8Mode: 捕捉模式，@ ref EN_CAP_MODE_T。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hal_ir_start_study

函数原型

EN_ERR_STA_T rom_hal_ir_start_study(stTIMER_Handle_t* pstIR, uint8_t u8Rtune);

描述

启动 IR 学习模式。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。
u8Rtune	设置 RC 滤波器中 R 的阻值， $R = 50K \times 2^{(u8Rtune)}$ 。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hal_ir_stop_study

函数原型

EN_ERR_STA_T rom_hal_ir_stop_study(stTIMER_Handle_t* pstIR);

描述

停止 IR 学习模式。

参数

参数	描述
pstIR	IR 句柄，应为 IR0 / IR1 / IR2 / IR3。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hal_timer_decode_init

函数原型

EN_ERR_STA_T rom_hal_timer_decode_init(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, stDecodeInit_t* pstDecodeInit);

描述

初始化指示的定时器，配置解码。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个定时器，参考 EN_TIMER_DECODE_CH_T 枚举定义。
pstDecodeInit	解码初始化结构类型。 .u8Prescale: 输入时钟的分频。 16-bit 定时器: u8Prescale 范围为 [0:16)。 32-bit 定时器: u8Prescale 范围为 [0:256)。 .u8Mode: 解码模式，参考 EN_DECODE_MODE_T。 .u8Pol: 解码极性，参考 EN_DECODE_POL_T。 .u8Value: 解码间隔值。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hal_timer_enable_decode

函数原型

EN_ERR_STA_T rom_hal_timer_enable_decode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

使能定时器解码功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个定时器，参考 EN_TIMER_DECODE_CH_T 枚举定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

rom_hal_timer_disable_decode

函数原型

EN_ERR_STA_T rom_hal_timer_disable_decode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

描述

除能定时器解码功能。

参数

参数	描述
pstTIMER	TIMER 句柄，应为 TIMER0 / TIMER1 / TIMER2 / TIMER3。
enCh	指示将配置哪个定时器，参考 EN_TIMER_DECODE_CH_T 枚举定义。

返回

EN_ERR_STA_T	函数返回状态，参考 EN_ERR_STA_T 枚举定义。
--------------	------------------------------

IR 模块范例

IR NEC 编码范例

IR NEC 编码说明

该单片机的 IR 模块支持多种协议。本范例以 NEC IR 协议为例，介绍 IR 模块的使用方法。
在 NEC 协议中，载波频率为 38 kHz，引导码、Code 0、Code 1 和重复码的取值如下：

引导码

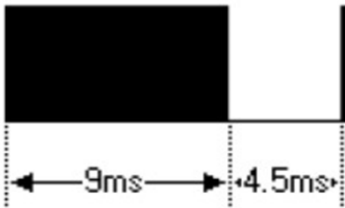


图 14. 引导码

Bit 说明

Bit 0: 脉冲时间 – 560 μ s，间隔时间 – 565 μ s，周期时间 – 1125 μ s

Bit 1: 脉冲时间 – 560 μ s，间隔时间 – 1690 μ s，周期时间 – 2250 μ s

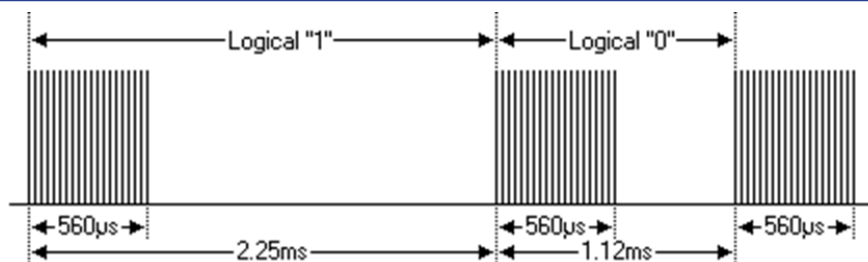


图 15. Bit 说明

重复码



图 16. 重复码

IR NEC 编码范例代码

```
static uint8_t ir_tx_data_signal_generate (void)
{
    // GPIO init
    patch_hw_gpio_set_pin_input_output(GPIO_PORT_IR_OUT, GPIO_PIN_0, GPIO_MODE_IMPEDANCE);
    patch_hw_gpio_set_pin_pull_mode(GPIO_PORT_IR_OUT, GPIO_PIN_0, GPIO_PULL_NONE);
    patch_hw_gpio_set_pin_pid(GPIO_PORT_IR_OUT, GPIO_PIN_7, PID_IR_OUT);
    patch_hw_gpio_set_pin_pid(GPIO_PORT_PWM_OUT, GPIO_PIN_9, PID_GTIM_PWM0_CHA);
    patch_hw_gpio_set_pin_pid(GPIO_PORT_IR_SIGNLE_OUT, GPIO_PIN_11, PID_GTIM_PWM0_CHB);
    // IR send init, config the leader code
    rom_hal_ir_send_init(IR0, 38000, 50);
    // Config data
    rom_hw_ir_set_pwm_current_compare_and_polarity(IR0, CLEAR_SIGN(u16Data[0]), PWM_POL_FALLING);
    // Start send
    rom_hal_start_send(IR0);
}
```

IR 解码范例

IR 解码范例代码

```
static uint8_t decode_func(void)
{
    // Decode Init struct type
    stDecodeInit_t g_stDecodeInit;
    g_stDecodeInit.ul6CompVal = u32Compare;
    g_stDecodeInit.u8Prescale = u8Prescale;
    g_stDecodeInit.u8Mode = u8TrigMode;
    g_stDecodeInit.u8Pol = u8OutPol;
    g_stDecodeInit.u8Interval = u8Interval;
```



```
g_stDecodeInit.u8Siganl = u8SignalSrc;
// Decode GPIO Init
patch_hw_gpio_set_pin_pull_mode(GPIOA, GPIO_PIN_0, GPIO_PULL_NONE);
patch_hw_gpio_set_pin_input_output(GPIOA, GPIO_PIN_0, GPIO_MODE_IMPEDANCE);
patch_hw_gpio_set_pin_pull_mode(GPIOA, GPIO_PIN_1, GPIO_PULL_NONE);
patch_hw_gpio_set_pin_pull_mode(GPIOA, GPIO_PIN_2, GPIO_PULL_NONE);
// Enable ir rx amp.
patch_hw_gpio_enable_ir_rx_amp();
// Set IR Amplifier amplification factor
patch_hw_gpio_set_ir_rx_rtune_amp(1);
// Set indicated pin peripheral function
patch_hw_gpio_set_pin_pid(GPIO_PORT_TIM0_CHB, GPIO_PIN_TIM0_CHB,
PID_GTIM_DECODE0_CHB);
// Ir function enable
rom_hw_ir_enable(stTimerTestCfg.stTIMER_Handle);
// Init a indicated timer, config decode.
rom_hal_timer_decode_init(g_stTimerTestCfg.stTIMER_Handle,
(EN_DECODE_CH_T)g_stTimerTestCfg.EN_TIMER_CH, &g_stDecodeInit);
// Enable timer decode function.
rom_hal_timer_enable_decode(TIMER0, (EN_DECODE_CH_T)TIMER_CHB);
}
```

8 RTX RTOS

关于 RTX RTOS

什么是 RTX RTOS

CMSIS-RTOS V2 (CMSIS-RTOS2) 为基于 Arm® Cortex® 处理器的单片机提供通用 RTOS 接口。它为需要 RTOS 功能的软件组件提供了一个标准化的 API，从而给用户和软件行业带来了重大的好处：

- CMSIS-RTOS2 提供了许多应用程序所需的基本功能。
- CMSIS-RTOS2 的统一特性集减少了学习的工作量并简化了软件组件的共用。
- 使用 CMSIS-RTOS2 的中间件组件与 RTOS 无关，并且更容易适应。
- CMSIS-RTOS2 的标准工程模板可以随免费提供的 CMSIS-RTOS2 实现一起提供。

CMSIS-RTOS2 管理单片机系统的资源，实现并发运行的并行线程的概念。

应用程序经常需要多个并发活动。CMSIS-RTOS2 可以在需要时管理多个并发活动。每个活动都有一个单独的线程来执行特定的任务，这简化了整个程序结构。CMSIS-RTOS2 系统具有可扩展性，可以在以后轻松添加其他的线程。线程具有优先级，允许更快地执行用户应用程序中对时间要求严格的部分。

CMSIS-RTOS2 提供了许多实时应用程序所需的服务，例如定时器功能的定时激活、存储器管理以及具有时间限制的线程之间的消息交换。

CMSIS-RTOS2 满足以下新要求：

- 动态对象创建不再需要静态内存，静态内存缓冲器现在是可选的。
- 支持 Armv8-M 架构，提供安全和非安全的代码执行状态。
- 多核系统中消息传递的规定。
- 全面支持 C++ 运行时环境。
- C 接口，在 ABI 兼容编译器中是二进制兼容的。

以下与 CMSIS-RTOS2 相关的文件存在于 ARM::CMSIS 包目录中：

表 5. CMSIS 包目录

目录	内容
CMSIS / Documentation / RTOS2	本文档
CMSIS / RTOS2 / Include	cmsis_os2.h 头文件
CMSIS / RTOS2 / RTX	基于 RTX 版本 5 的 CMSIS-RTOS2 参考实现
CMSIS / RTOS2 / Source	基于 OS Tick API 的各种处理器的通用 OS Tick 实现
CMSIS / RTOS2 / Template	CMSIS-RTOS V1 兼容层

通用 RTOS 接口

CMSIS-RTOS2 是一个通用的 API，它与底层的 RTOS 内核无关。程序员在用户代码中调用 CMSIS-RTOS2 API 函数，以确保从一个 RTOS 到另一个 RTOS 的最大可移植性。使用 CMSIS-RTOS2 API 的中间件可以避免不必要的移植工作。

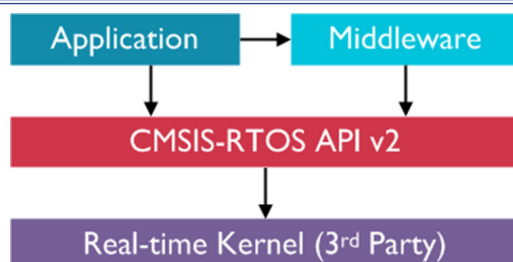


图 17. CMSIS-RTOS API 结构

典型的 CMSIS-RTOS2 API 实现与现有的实时内核接口。CMSIS-RTOS2 API 提供以下属性和功能：

- 函数名、标识符和参数是描述性的，易于理解。这些功能强大而灵活，减少了向用户公开的函数数量。
- 线程管理允许定义、创建和控制线程。
- 中断服务程序 (ISR) 可以调用一些 CMSIS-RTOS 函数。当不能从 ISR 上下文中调用 CMSIS-RTOS 函数时，它会拒绝调用并返回错误代码。
- 三种不同的事件类型支持多线程和 / 或 ISR 之间的通信：
 - 线程标志：可用于向线程指示特定条件。
 - 事件标志：可用于向线程或 ISR 指示事件。
 - 消息：可以发送到线程或 ISR。消息在队列中缓冲。
- 合并了互斥锁管理和信号量。
- CPU 时间可以通过以下功能进行调度：
 - 在许多 CMSIS-RTOS 函数中都包含了超时参数，以避免系统锁定。当指定超时时，系统会等待，直到资源可用或事件发生。在等待期间，将调度其他线程。
 - osDelay 和 osDelayUntil 函数将线程置于等待状态一段指定的时间。
 - osThreadYield 提供协作式线程切换，并将执行传递给具有相同优先级的另一个线程。
- 定时器管理功能用于触发函数的执行。

CMSIS-RTOS2 API 旨在通过 Cortex®-M 存储器保护单元 (MPU) 选择性地合并多处理器系统和 / 或访问保护。

在一些 RTOS 实现中，线程可能在不同的处理器上执行，因此消息队列可能驻留在共用的存储器资源中。

CMSIS-RTOS2 API 鼓励软件行业发展现有的 RTOS 实现。RTOS 实现可以与 Cortex®-M 处理器不同，并在各个方面进行了优化。可选功能可以是，例如：

- 支持 Cortex®-M 存储器保护单元 (MPU)
- 支持多处理器系统
- 支持 DMA 控制器
- 确定性上下文切换
- 循环上下文切换
- 避免死锁，例如优先级反转
- 通过使用 Armv7-M 指令 LDREX 和 STREX 来实现零中断延迟

功能概述

CMSIS-RTOS2 提供了以下 API 接口：

- CMSIS-RTOS C API V2 是支持动态对象创建和 Armv8-M (Arm® Cortex®-M33) 的 C 函数接口。
- OS Tick API 是内核系统定时器的接口。

函数引用：

表 6. CMSIS RTOS 函数

CMSIS-RTOS API V2	描述
内核信息和控制 (Kernel Information and Control)	提供版本 / 系统信息，并启动 / 控制 RTOS 内核。
线程管理 (Thread Management)	定义、创建和控制线程函数。
线程标志 (Thread Flags)	通过标志来同步线程。
事件标志 (Event Flags)	通过事件标志来同步线程。
通用等待函数 (Generic Wait Functions)	等待一段时间。
定时器管理 (Timer Management)	创建和控制定时器和定时器回调函数。
互斥锁管理 (Mutex Management)	通过互斥锁来同步资源访问。
信号量 (Semaphores)	从不同的线程同时访问共用资源。
内存池 (Memory Pool)	管理线程安全的固定大小的动态存储器区块。
消息队列 (Message Queue)	在类似 FIFO 的操作中在线程之间交换消息。

RTX V5 实现

Keil RTX 版本 5 (RTX5) 实现了 CMSIS-RTOS2 作为基于 Arm® Cortex®-M 处理器的单片机的本地 RTOS 接口。提供了到 CMSIS-RTOS API V1 的转换层。因此，RTX5 可以在之前基于 RTX 版本 4 和 CMSIS-RTOS 版本 1 的应用程序中使用。

以下各节提供了更多详细信息：

- 创建 RTX5 工程：介绍如何在 Keil MDK 中设置 RTX V5 工程。
- 工作原理：提供有关 CMSIS-RTOS RTX V5 操作的基本信息。
- 配置 RTX V5：描述 CMSIS-RTOS RTX V5 的配置参数。

创建 RTX5 工程

使用 RTX5 创建单片机应用程序的步骤如下：

- 创建一个新工程并选择一个单片机。
- 在“管理运行时环境 (Manage Run-Time Environment)”窗口中，选择“CMSIS::CORE”和“CMSIS::RTOS2 (API)::Keil RTX5”。可以选择将 RTX 添加为库 (Variant: Library) 或添加完整的源代码 (Variant: Source - 如果使用事件记录器，则为必需)。

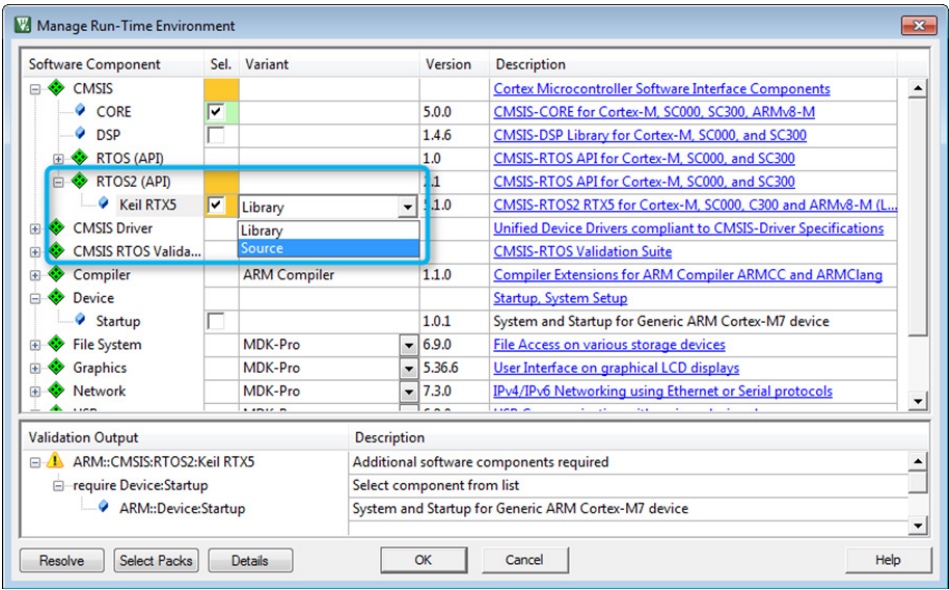


图 18. 创建 RTX5 工程

- 如果“验证输出 (Validation Output)”需要其他组件，请尝试使用“解决 (Resolve)”按钮。
- 点击“OK”。在“工程 (Project)”窗口中，将看到已经自动添加到工程中的文件，例如 RTX_Config.h, RTX_Config.c, 库或源代码文件，以及系统和启动文件。

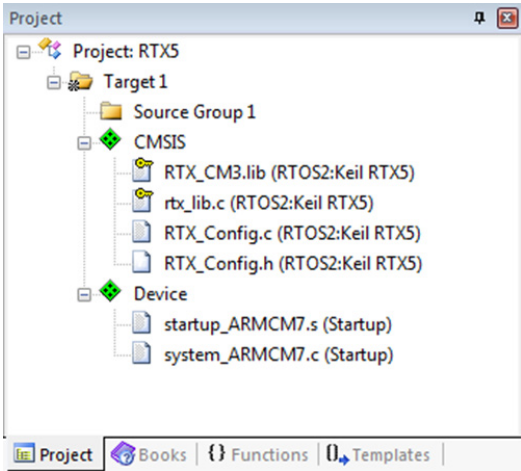


图 19. RTX5 工程文件结构

- 如果使用上述的“Variant: Source”，则必须确保至少使用 C99 编译器模式 (工程选项 → C/C++ → C99 模式)。
- 可以通过右键点击“Source Group 1”并选择“Add New Item”到“Source Group 1”来将模板文件添加到工程中。在新窗口中，点击“用户代码模板 (User Code Template)”。在右侧，将看到 CMSIS-RTOS RTX 的所有可用模板文件。

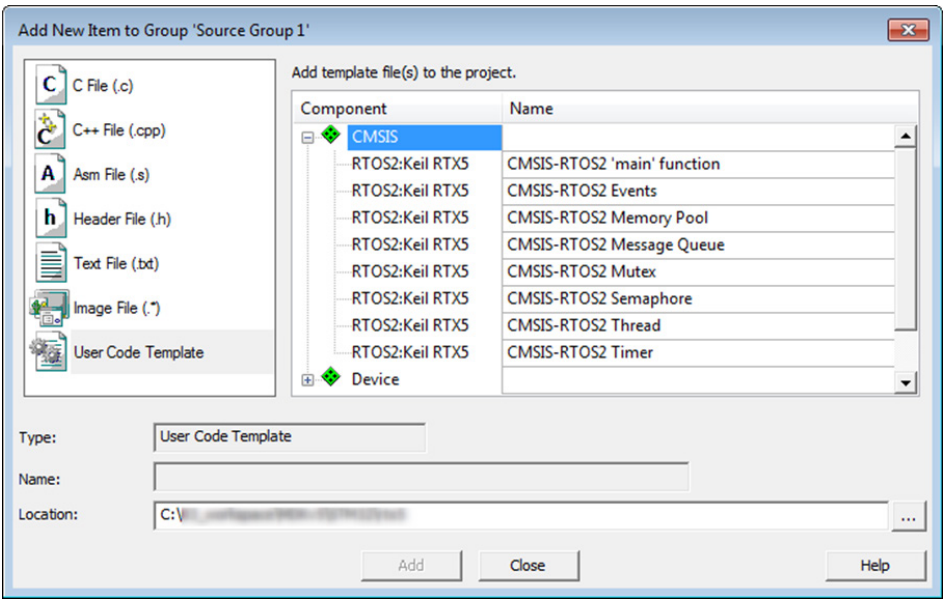


图 20. 添加模板文件

- 使用 RTX_Config.h 文件根据应用程序的需求配置 RTX5。

在 Cortex®-M 上使用中断

在 Cortex®-M 处理器上，RTX5 内核使用以下中断异常。下表还列出了必须分配给这些中断的优先级。

表 7. Cortex®-M 中断

处理程序	优先级	中断 / 异常
SysTick	最低	内核系统定时器中断，用于产生周期性定时器节拍
PendSV	最低	PendSV (请求系统级服务)，从处理程序 (Handler) 模式调用某些 RTX 函数时使用
SVC	最低 + 1	管理程序调用，用于从线程 (Thread) 模式进入 RTOS 内核

可以无限制地使用其他设备中断。对于 Arm® Cortex®-M33 处理器，RTX 内核永远不会除能中断。

中断优先级分组的使用

- 中断优先级分组应该在调用函数 osKernelStart() 之前使用 CMSIS-Core 函数 NVIC_SetPriorityGrouping 进行配置。RTX 内核使用优先级组值来设置 SysTick 和 PendSV 中断的优先级。
- RTX 内核为上表中列出的中断 / 异常设置优先级，并使用最低的两个优先级。
- 请勿更改 RTX 内核使用的优先级。如果无法避免，请确保 SysTick/PendSV 的抢占优先级低于 SVC。

- 允许的优先级组值为 0~6。优先级组值为 7 将导致 RTX 失败，因为只有一个优先级级别可用。
- 主堆栈用于运行 RTX 功能。因此，需要为 RTX 内核的执行配置足够的堆栈。

范例代码

```
osKernelInitialize();           // initialize RTX
NVIC_SetPriorityGrouping(3);    // setup priority grouping
tread_id = osThreadNew(tread_func, NULL, NULL); // create some threads
osKernelStart();
```

工作原理

内核的许多方面都是可配置的，并在适用的情况下提及配置选项。

系统启动

由于 main 不再是一个线程，因此 RTX5 在到达 main 之前不会干扰系统启动。一旦执行到达 main()，建议按照如下顺序来初始化硬件并启动内核。这也反映在随 RTX5 component.main() 提供的用户代码模板文件“CMSIS-RTOS2 ‘main’ function”中。

您的应用程序 main() 应该按照给定的顺序至少实现以下内容：

- 硬件的初始化和配置，包括外设、存储器、引脚、时钟和中断系统。
- 使用 CMSIS-Core (Cortex®-M) 的函数更新系统内核时钟。
- 使用 osKernelInitialize 初始化 CMSIS-RTOS 内核。
- (可选) 创建一个线程，例如 app_main，该线程将用作使用 osThreadNew 的主线程。一旦调度器运行该线程，它应负责创建和启动对象。或者，可以直接在 main() 中创建线程。
- 使用 osKernelStart 启动 RTOS 调度器，它还配置系统节拍定时器并初始化 RTOS 特定的中断。在成功执行的情况下，该函数不会返回。因此，osKernelStart 之后的任何应用程序代码都不会执行。

调度器

RTX5 实现了一个低延迟的抢占式调度器。RTX5 的主要部分在处理程序模式下执行，例如：

- systick_handler 用于基于时间的调度
- svc_handler 用于基于锁的调度
- pendsv_handler 用于基于中断的调度

为了在 ISR 执行方面具有低延迟，这些系统异常被配置为使用可用的最低优先级组。优先级的配置使得它们之间不会发生抢占。因此，不需要中断临界区（即中断锁）来保护调度器。

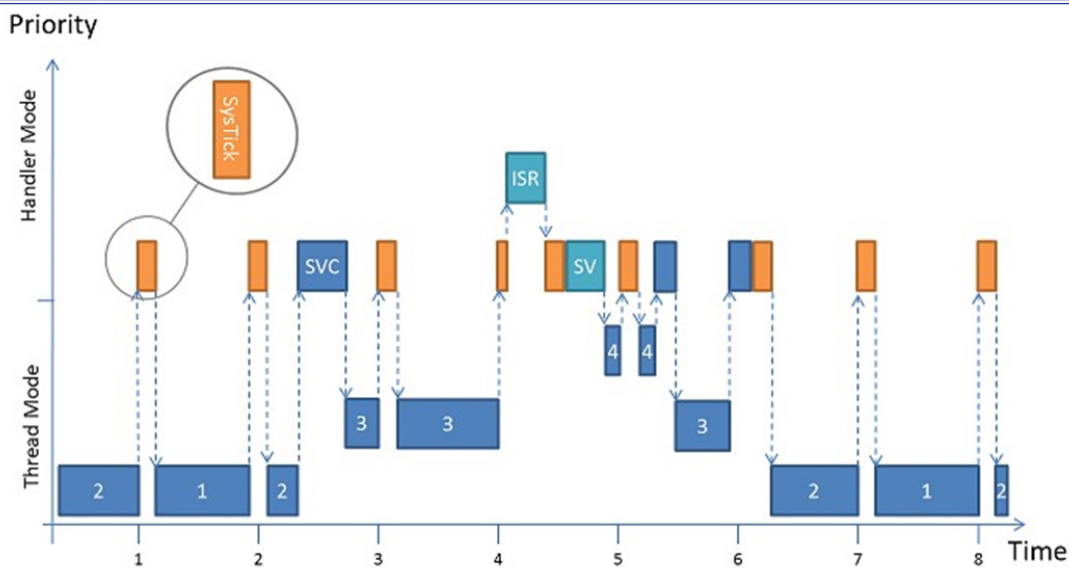


图 21. 线程调度和中断执行

调度器结合了优先级和基于循环的上下文切换。上图中描述的例子包含四个线程 (1、2、3 和 4)。线程 1 和线程 2 共用相同的优先级，线程 3 具有较高的优先级，线程 4 具有最高的优先级 (osThreadAttr_t::priority)。只要线程 3 和 4 被阻塞，调度器就会按时间片在线程 1 和线程 2 之间进行切换 (循环)。可以配置循环调度的时间片，请参阅系统配置中的循环超时。

在时间索引 2 时，线程 2 通过任意 RTOS 调用 (在 SVC 处理程序模式下执行) 解除对线程 3 的阻塞。由于线程 3 具有最高优先级，因此调度器立即切换到线程 3。线程 4 仍然被阻塞。

在时间索引 4 时，发生中断 (ISR) 并抢占 SysTick_Handler。RTX 不会给中断服务的执行增加任何延迟。ISR 程序使用 RTOS 调用来解除对线程 4 的阻塞。PendSV 标志被设置为延迟上下文切换，而不是立即切换到线程 4。PendSV_Handler 在 SysTick_Handler 返回之后立即执行，并执行到线程 4 的延迟上下文切换。一旦最高优先级的线程 4 通过使用阻塞 RTOS 调用执行再次阻塞，在时间索引 5 期间立即切换回线程 3。

在时间索引 5 时，线程 3 也使用阻塞 RTOS 调用。因此，调度器切换回线程 2 以获得时间索引 6。在时间索引 7 时，调度器使用循环机制切换到线程 1，依此类推。

内存分配

RTX5 对象 (线程、互斥锁、信号量、定时器、消息队列、线程和事件标志以及内存池) 需要专用的 RAM 存储器。可以使用 osObjectNew() 调用创建对象，并使用 osObjectDelete() 调用删除对象。相关的对象内存需要在对象的生命周期内可用。

RTX5 为对象提供了三种不同的内存分配方法：

- 全局内存池为所有对象使用单个全局内存池。它很容易配置，但是当创建和销毁具有不同大小的对象时，可能存在内存碎片的缺点。
- 对象特定内存池为每个对象类型使用固定大小的内存池。该方法具有时间确定性，避免了内存碎片。
- 静态对象内存存在编译期间保留内存，完全避免了系统内存不足的情况。这通常是一些重视安全的系统所必需的。

可以在同一个应用程序中混合使用所有内存分配方法。

配置 RTX V5

“RTX_Config.h”文件定义了 CMSIS-RTOS RTX 的配置参数，每一个使用 CMSIS-RTOS RTX 内核的工程都必须有这个文件。下面几节将详细解释各个配置选项：

- 系统配置 (System Configuration) 涵盖全局内存池、节拍频率、ISR 事件缓冲器和循环线程切换的系统范围设置。
- 线程配置 (Thread Configuration) 提供了几个参数来配置线程管理功能。
- 定时器配置 (Timer Configuration) 提供了几个参数来配置定时器管理功能。
- 事件标志配置 (Event Flags Configuration) 提供了几个参数来配置事件标志功能。
- 互斥锁配置 (Mutex Configuration) 提供了几个参数来配置互斥锁管理功能。
- 信号量配置 (Semaphore Configuration) 提供了几个参数来配置信号量功能。
- 内存池配置 (Memory Pool Configuration) 提供了几个参数来配置内存池功能。
- 消息队列配置 (Message Queue Configuration) 提供了几个参数来配置消息队列功能。
- 事件记录器配置 (Event Recorder Configuration) 提供了几个参数来配置 RTX 以与事件记录器一起使用。

文件“RTX_Config.c”包含函数 osRtxIdleThread 和 osRtxErrorNotify 的默认实现。通过将这两个函数重新定义为用户代码的一部分来简单地用自定义行为覆盖。

配置文件使用配置向导注释。更多详细信息，请参考“Pack - Configuration Wizard Annotations”。根据开发工具的不同，注释可能会带来更加用户友好的设置图形表示。下图显示了 MDK 中的 μ Vision 配置向导界面：

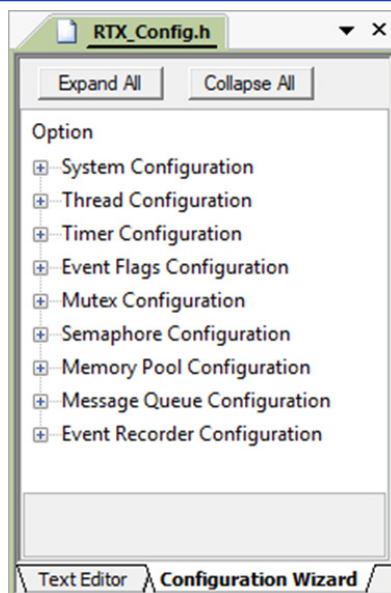


图 22. 配置向导界面的 RTX_Config.h

系统配置

系统配置涵盖全局内存池、节拍频率、ISR 事件缓冲器和循环线程切换的系统范围设置。

Option	Value
System Configuration	
Global Dynamic Memory size [bytes]	4096
Kernel Tick Frequency [Hz]	1000
Round-Robin Thread switching	<input checked="" type="checkbox"/>
ISR FIFO Queue	16 entries
Object Memory usage counters	<input type="checkbox"/>

图 23. RTX_Config.h: 系统配置

下表列出了用于系统配置的宏。

表 8. 系统配置

名称	#define	描述
Global Dynamic Memory size [bytes]	OS_DYNAMIC_MEM_SIZE	为全局内存池定义全局动态内存大小。默认值为 32768。取值范围为 [0-1073741824] 字节，是 8 字节的倍数。
Kernel Tick Frequency (Hz)	OS_TICK_FREQ	以 Hz 为单位定义延迟和超时的基准时间单位。默认值：1000Hz = 1ms 周期。
Round-Robin Thread switching	OS_ROBIN_ENABLE	使能循环线程切换。
Round-Robin Timeout	OS_ROBIN_TIMEOUT	定义线程切换之前执行线程的时间。默认值为 5。取值范围为 [1-1000]。
ISR FIFO Queue	OS_ISR_FIFO_QUEUE	从 ISR 调用的 RTOS 函数将请求存储到此缓冲器。默认值为 16 个条目。取值范围为 [4-256]，是 4 的倍数。
Object Memory usage counters	OS_OBJ_MEM_USAGE	使能对象内存使用计数器，用于单独评估每个 RTOS 对象类型的最大内存池需求。

线程配置

RTX5 提供了几个参数来配置线程管理功能。

Option	Value
Thread Configuration	
Object specific Memory allocation	<input checked="" type="checkbox"/>
Number of user Threads	1
Number of user Threads with default Stack size	0
Total Stack size [bytes] for user Threads with user-provided Stack size	0
Default Thread Stack size [bytes]	200
Idle Thread Stack size [bytes]	200
Stack overrun checking	<input checked="" type="checkbox"/>
Stack usage watermark	<input type="checkbox"/>
Processor mode for Thread execution	Privileged mode

图 24. RTX_Config.h: 线程配置

下表列出了用于线程配置的宏。

表 9. 线程配置

选项	#define	描述
Object specific Memory allocation	OS_THREAD_OBJ_MEM	使能对象特定内存分配。
Number of user Threads	OS_THREAD_NUM	定义可同时活动的最大用户线程数。适用于具有系统为控制块提供内存的用户线程。默认值为 1。取值范围为 [1-1000]。
Number of user Threads with default Stack size	OS_THREAD_DEF_STACK_NUM	定义具有默认堆栈大小的最大用户线程数，适用于指定堆栈大小为 0 的用户线程。取值范围为 [0-1000]。
Total Stack size [bytes] for user Threads with user-provided Stack size	OS_THREAD_USER_STACK_SIZE	定义具有用户提供的堆栈大小的用户线程的组合堆栈大小。默认值为 0。取值范围为 [0-1073741824] 字节，是 8 的倍数。
Default Thread Stack size [bytes]	OS_STACK_SIZE	定义指定堆栈大小为零的线程的堆栈大小。默认值为 3072。取值范围为 [96-1073741824] 字节，是 8 的倍数。
Idle Thread Stack size [bytes]	OS_IDLE_THREAD_STACK_SIZE	定义空闲线程的堆栈大小。默认值是 512。取值范围为 [72-1073741824] 字节，是 8 的倍数。
Idle Thread TrustZone Module ID	OS_IDLE_THREAD_TZ_MOD_ID	定义空闲线程将使用的 TrustZone 模块 ID。如果空闲线程需要调用安全函数，则需要将其设置为非零值。默认值为 0。
Stack overrun checking	OS_STACK_CHECK	在线程切换处使能堆栈溢出检查。
Stack usage watermark	OS_STACK_WATERMARK	使用水印模式初始化线程堆栈，以分析堆栈使用情况。使能此选项会显著增加线程创建的执行时间。
Processor mode for Thread execution	OS_PRIVILEGE_MODE	控制处理器模式。默认值为特权模式。取值范围为 [0 = 非特权； 1 = 特权] 模式。

线程数和堆栈空间的配置

RTX5 内核为每个线程使用一个单独的堆栈空间，并提供两种定义堆栈需求的方法：

- 静态分配：
当 `osThreadAttr_t::stack_mem` 和 `osThreadAttr_t::stack_size` 指定用于线程堆栈的内存区域时。注意：所提供的堆栈内存必须是 64 位对齐的，即通过使用 `uint64_t` 进行声明。
- 动态分配：
当 `osThreadAttr_t` 为 `NULL` 或 `osThreadAttr_t::stack_mem` 为 `NULL` 时，系统将从以下位置分配堆栈内存：
 - 当使能“对象特定内存分配”，并且“具有默认堆栈大小的用户线程数”不为 0 且 `osThreadAttr_t::stack_size` 为 0 (或 `osThreadAttr_t` 为 `NULL`) 时，则分配在对象特定内存池 (默认堆栈大小)。
 - 当使能“对象特定内存分配”，并且“用户...的总堆栈大小”不为 0 且 `osThreadAttr_t::stack_size` 不为 0 时，则分配在对象特定内存池 (用户提供的堆栈大小)。
 - 当除能“对象特定内存分配”，或 (`osThreadAttr_t::stack_size` 不为 0 且 “用户...的总堆栈大小”为 0) 或 (`osThreadAttr_t::stack_size` 为 0) 时，则分配在全局内存池。

`osThreadAttr_t` 是 `osThreadNew` 函数的一个参数。

定时器配置

RTX5 提供了几个参数来配置定时器管理功能。

Option	Value
Timer Configuration	
Object specific Memory allocation	<input type="checkbox"/>
Number of Timer objects	1
Timer Thread Priority	High
Timer Thread Stack size [bytes]	200
Timer Callback Queue entries	4

图 25. RTX_Config.h: 定时器配置

下表列出了用于定时器配置的宏。

表 10. 定时器配置

名称	#define	描述
Object specific Memory allocation	OS_TIMER_OBJ_MEM	使能对象特定内存分配。
Number of Timer objects	OS_TIMER_NUM	定义可以同时活动的最大对象数量。适用于具有系统为控制块提供内存的对象。取值范围为 [1-1000]。
Timer Thread Priority	OS_TIMER_THREAD_PRIO	定义定时器线程的优先级。默认值为 40。取值范围为 [8-48]，是 8 的倍数。这些数字的优先级相关如下：8 = 低；16 = 低于正常；24 = 正常；32 = 高于正常；40 = 高；48 = 实时
Timer Thread Stack size [bytes]	OS_TIMER_THREAD_STACK_SIZE	定义定时器线程的堆栈大小。当不使用定时器时，可以设置为 0。默认值是 512。取值范围为 [0-1073741824]，是 8 的倍数。
Timer Thread TrustZone Module ID	OS_TIMER_THREAD_TZ_MOD_ID	定义定时器线程将使用的 TrustZone 模块 ID。如果有任何定时器回调需要调用安全函数，则需要将其设置为非零值。默认值为 0。
Timer Callback Queue entries	OS_TIMER_CB_QUEUE	并发活动定时器回调函数的数量。当不使用定时器时，可以设置为 0。默认值为 4。取值范围为 [0-256]。

事件标志配置

RTX5 提供了几个参数来配置事件标志功能。

Option	Value
Event Flags Configuration	
Object specific Memory allocation	<input type="checkbox"/>
Number of Event Flags objects	1

图 26. RTX_Config.h: 事件标志配置

下表列出了用于事件事件配置的宏定义。

表 11. 事件配置

名称	#define	描述
Object specific Memory allocation	OS_EVFLAGS_OBJ_MEM	使能对象特定内存分配。
Number of Event Flags objects	OS_EVFLAGS_NUM	定义可以同时活动的最大对象数量。适用于具有系统为控制块提供内存的对象。取值范围为 [1-1000]。

互斥锁配置

RTX5 提供了几个参数来配置互斥锁管理功能。

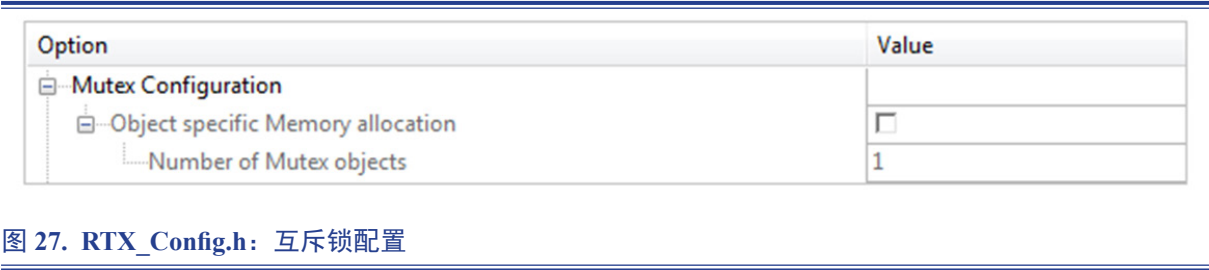


图 27. RTX_Config.h: 互斥锁配置

表 12. 互斥锁配置

名称	#define	描述
Object specific Memory allocation	OS_MUTEX_OBJ_MEM	使能对象特定内存分配。
Number of Mutex objects	OS_MUTEX_NUM	定义可以同时活动的最大对象数量。适用于具有系统为控制块提供内存的对象。取值范围为 [1-1000]。

信号量配置

RTX5 提供了几个参数来配置信号量功能。



图 28. RTX_Config.h: 信号量配置

下表列出了用于信号量配置的宏。

表 13. 信号量配置

名称	#define	描述
Object specific Memory allocation	OS_SEMAPHORE_OBJ_MEM	使能对象特定内存分配。
Number of Semaphore objects	OS_SEMAPHORE_NUM	定义可以同时活动的最大对象数量。适用于具有系统为控制块提供内存的对象。取值范围为 [1-1000]。

内存池配置

RTX5 提供了几个参数来配置内存池功能。

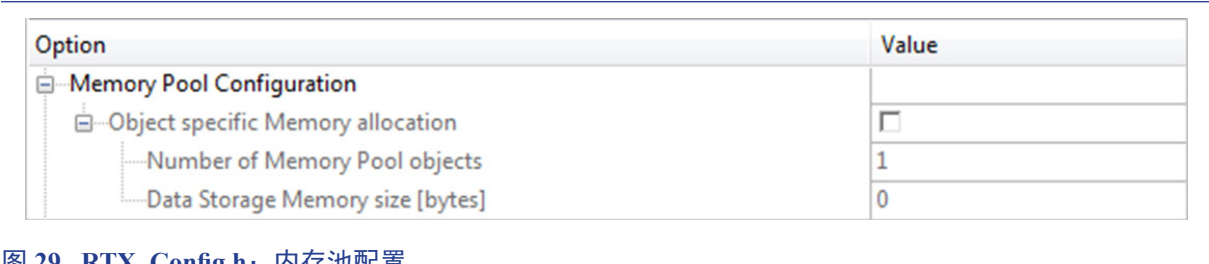


图 29. RTX_Config.h: 内存池配置

表 14. 内存池配置

名称	#define	描述
Object specific Memory allocation	OS_MEMPOOL_OBJ_MEM	使能对象特定内存分配。
Number of Memory Pool objects	OS_MEMPOOL_NUM	定义可以同时活动的最大对象数量。适用于具有系统为控制块提供内存的对象。取值范围为 [1-1000]。
Data Storage Memory size [bytes]	OS_MEMPOOL_DATA_SIZE	定义组合数据存储内存大小。适用于具有系统为数据存储提供内存的对象。默认值为 0。取值范围为 [0-1073741824]，是 8 的倍数。

消息队列配置

RTX5 提供了几个参数来配置消息队列功能。

Option	Value
Message Queue Configuration	
Object specific Memory allocation	<input type="checkbox"/>
Number of Message Queue objects	1
Data Storage Memory size [bytes]	0

图 30. RTX_Config.h: 消息队列配置

表 15. 消息队列配置

名称	#define	描述
Object specific Memory allocation	OS_MSGQUEUE_OBJ_MEM	使能对象特定内存分配。
Number of Message Queue objects	OS_MSGQUEUE_NUM	定义可以同时活动的最大对象数量。适用于具有系统为控制块提供内存的对象。取值范围为 [1-1000]。
Data Storage Memory size [bytes]	OS_MSGQUEUE_DATA_SIZE	定义组合数据存储内存大小。适用于具有系统为数据存储提供内存的对象。默认值为 0。取值范围为 [0-1073741824]，是 8 的倍数。

RTX-RTOS 范例

Keil RTX5 的第一步

RTOS 本身由一个调度器组成，该调度器支持程序线程的循环、抢占式和协作式多任务处理，以及时间和内存管理服务。线程间通信由额外的 RTOS 对象支持完成，包括信号线程和事件标志、信号量、互斥锁、消息传递和内存池系统。正如我们将看到的，中断处理也可以通过 RTOS 内核调度的优先线程来完成。

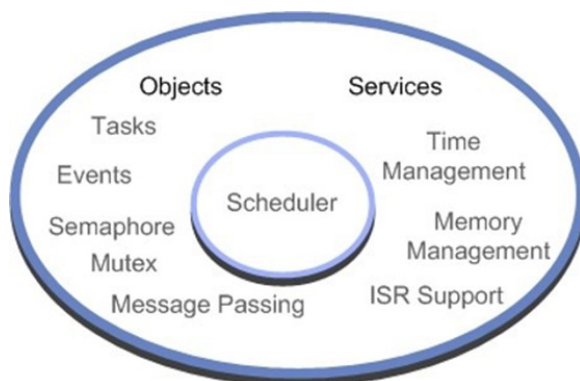


图 31. RTOS 内核

访问 CMSIS-RTOS2 API

要访问应用程序代码中的任何 CMSIS-RTOS2 功能，必须包含以下头文件。

```
#include <cmsis_os2.h>
```

该头文件由 Arm 作为 CMSIS-RTOS2 标准的一部分进行维护。对于 Keil RTX5，这是默认的 API。其他 RTOS 将有自己的专有 API，但可能会提供一个包装层来实现 CMSIS-RTOS2 API，因此可以在需要与 CMSIS 标准兼容的情况下使用。

线程

典型的 C 程序的构建块是函数，我们调用这些函数来执行特定的程序，然后返回到调用函数。在 CMSIS-RTOS2 中，执行的基本单位是“线程”。线程与 C 程序非常相似，但有一些非常根本的区别。

```
unsigned int procedure (void)
{
    return(ch);
}

void thread (void)
{
    while(1)
    {
    }
}

__NO_RETURN
void Thread1(void*argument)
{
    while(1)
    {
    }
}
```

C 函数总归会返回，而 RTOS 线程必须包含一个循环，一旦启动，永远不会终止，会一直运行。您可以将线程视为在 RTOS 中运行的小型自包含程序。在 Arm 编译器中，可以通过使用 __NO_RETURN 宏来优化线程。此属性可降低调用永远不会返回的函数的成本。

RTOS 程序由许多线程组成，这些线程由 RTOS 调度器控制。这个调度器使用 SysTick 定时器来生成一个周期性中断作为基准时间。调度器将为每个线程分配一定的执行时间。因此，线程 1 将运行 5 ms，然后重新调度使线程 2 运行类似的时间；线程 2 再让给线程 3，最后将控制权传递回线程 1。通过循环方式将这些运行时间分配给每个线程，表面看起来像所有三个线程彼此并行运行。

从概念上讲，我们可以将每个线程视为在所有线程同时运行的情况下执行程序的特定功能单元。这为我们带来了一种更加面向对象的设计，在这种设计中，每个功能块都可以单独编码和测试，然后集成到一个完全运行的程序中。这不仅为最终应用程序的设计带来了结构，而且还有助于调试，因为可以很容易地将特定的错误隔离到特定的线程中。它还有助于在以后的工程中重用代码。创建线程时，还会为其分配自己的线程 ID。这是一个变量，作为每个线程的句柄，当我们想要管理线程的活动时使用。

```
osThreadId_t id1, id2, id3;
```

为了实现线程切换，需要制定一个 CPU 硬件定时器用于提供 RTOS 时间参考，并消耗一定的 RTOS 代码开销。此外，每次切换正在运行的线程时，我们都必须将所有线程变量的状态保存到线程堆栈中。同时把将要运行的线程的所有信息存储在由 RTOS 内核管理的线程控制块中。因此，“上下文切换时间”，即保存当前线程状态并加载和启动下一个线程的时间，是一个关键数字，它将取决于 RTOS 内核和底层硬件的设计。

线程控制块包含有关线程状态的信息。部分信息是其运行状态。在给定的系统中，只有一个线程可以运行，其他线程都将挂起，但准备运行。RTOS 具有多种线程间通信的方法（信号、信号量、消息）。在这种情况下，一个线程可能会被挂起，等待另一个线程或中断发出信号，然后恢复其就绪状态，从而 RTOS 调度器可以将其置于运行状态。

表 16. 线程块内容

状态	描述
运行	当前正在运行的线程
就绪	准备运行的线程
等待	等待 OS 事件的阻塞线程

在任何给定时刻都可能有一个线程正在运行。其余线程将准备运行，并由内核调度。线程也可能正在等待挂起 OS 事件。当这种情况发生时，它们将返回就绪状态并由内核调度。

启动 RTOS

为了构建一个简单的 RTOS 程序，我们将每个线程声明为一个标准的 C 函数，并为每个函数声明一个线程 ID 变量。

```
void thread1 (void);  
void thread2 (void);  
osThreadId thrdID1, thrdID2;
```

一旦处理器离开复位向量，将像往常一样进入 main() 函数。在 main() 中，我们必须调用 osKernelInitialize() 来设置 RTOS。在 osKernelInitialize() 函数成功完成之前，无法调用任何 RTOS 函数。一旦 osKernelInitialize() 完成，我们就可以创建更多的线程和其他 RTOS 对象。这可以通过创建一个启动器线程来实现，在下面的例子中称为 app_main()。在 app_main() 线程中，我们创建了启动应用程序运行所需的所有 RTOS 线程和对象。正如我们稍后将看到的，还可以在应用程序运行时动态创建和销毁 RTOS 对象。接下来，我们可以调用 osKernelStart() 来启动 RTOS 和调度器任务切换。在启动 RTOS 之前，可以运行任何初始化代码来设置外设和初始化硬件。

```
void app_main(void *argument)  
{  
    T_led_ID1 = osThreadNew(led_Thread1, NULL, &ThreadAttr_LED1);  
    T_led_ID2 = osThreadNew(led_Thread2, NULL, &ThreadAttr_LED2);  
    osDelay(osWaitForever);  
    while (1);  
}  
void main(void)  
{  
    osKernelInitialize(); // Initialize the kernel  
    osThreadNew(app_main, NULL, NULL); // Create the app_main() launcher thread
```



```
    osKernelStart();          // Start the RTOS
}
```

创建线程时还会为它们分配优先级。如果有许多线程准备运行，并且它们都具有相同的优先级，则将以循环方式为其分配运行时间。但是，如果具有更高优先级的线程准备好运行，RTOS 调度器将取消当前正在运行的线程的调度，并启动高优先级线程运行。这称为基于优先级的抢占式调度。在分配优先级时，必须非常小心，因为高优先级线程将继续运行，直到它进入等待状态，或者直到具有相同或更高优先级的线程准备好运行。

创建线程

一旦 RTOS 开始运行，就会有許多系统调用来管理和控制活动线程。

正如我们在第一个例子中看到的，`app_main()` 线程用作启动线程来创建应用程序线程。这两个阶段完成。首先定义线程结构，这允许我们定义线程操作参数。

```
osThreadId thread1_id; // thread handle
static const osThreadAttr_t threadAttr_thread1 =
{
    "Name_String",          // Human readable Name for debugger
    Attribute_bits_Control_Block_Memory,
    Control_Block_Size,
    Stack_Memory,
    Stack_Size,
    Priority,
    TrustZone_ID,
    reserved
};
```

线程结构要求我们定义线程函数的名称、线程优先级、任何特殊属性位、TrustZone ID 及其内存分配。这需要介绍相当多的细节，但我们将在这个应用程序说明的最后介绍所有内容。一旦定义了线程结构，就可以使用 `osThreadNew()` API 调用来创建线程。然后从应用程序代码中创建线程，通常是在 `app_main()` 线程内，但是可以在任何线程内的任何点创建线程。

```
thread1_id = osThreadNew(name_Of_C_Function, argument,&threadAttr_thread1);
```

这将创建线程并使其开始运行。另外，可以在线程启动时将参数传递给线程。

```
uint32_t startupParameter = 0x23;
thread1_id = osThreadNew(name_Of_C_Function, (uint32_t)startupParameter,
&threadAttr_thread1);
```

内存管理

创建每个线程时，都会为其分配堆栈，用于在上下文切换期间存储数据。这不应与本机 Cortex®-M 处理器堆栈混淆；它实际上是分配给线程的一块内存。默认的堆栈大小是在 RTOS 配置文件中定义的（我们将在后面看到），除非我们覆盖它以分配自定义大小，否则此内存量将分配给每个线程。如果线程定义结构中的堆栈大小值设置为零，则会将默认堆栈大小分配给线程。如果需要，可以通过在线程结构中定义更大的堆栈大小来为线程提供额外的内存资源。Keil RTX5 支持多种内存模型来分配线程内存。默认模型是全局内存池。在此模型中，创建的每个 RTOS 对象（线程、消息队列、信号量等）都是从单个内存块中分配内存的。

如果对象被销毁，则已分配的内存将返回到内存池。

这具有内存重用的优点，但也有可能引入内存碎片的问题。

全局内存池的大小在配置文件中定义：

```
#define OS_DYNAMIC_MEM_SIZE 4096
```

每个线程的默认堆栈大小在线程部分中定义：

```
#define OS_STACK_SIZE 256
```

还可以为每种不同类型的 RTOS 对象定义对象特定内存池。在此模型中，您可以定义特定对象类型的最大数量及其内存需求。然后 RTOS 计算并保留所需的内存使用量。

通过使能配置文件中每个部分提供的“对象特定内存”选项，在 RTOS 配置文件中再次定义对象特定模型：

```
#define OS_SEMAPHORE_OBJ_MEM 1
#define OS_SEMAPHORE_NUM 1
```

对于需要固定内存分配的简单对象，我们只需要定义给定对象类型的最大数量。对于更复杂的对象，例如线程，我们需要定义所需的内存使用量：

```
#define OS_THREAD_OBJ_MEM 1
#define OS_THREAD_NUM 1
#define OS_THREAD_DEF_STACK_NUM 0
#define OS_THREAD_USER_STACK_SIZE 1024
```

要对线程使用对象特定内存分配模型，我们必须提供线程整体内存使用情况的详细信息。最后，可以静态地分配线程堆栈内存。这对于必须严格定义内存使用情况的安全相关系统非常重要。

多实例

RTOS 的一个有趣功能是可以为同一基本线程代码创建多个运行实例。例如，您可以编写一个线程来控制 UART，然后创建这个线程代码的两个运行实例。这里，UART 代码的每个实例都可以管理不同的 UART。然后，我们可以创建分配给不同线程句柄的两个线程实例。还传递了一个参数，允许每个实例识别它负责哪个 UART。

```
#define UART1 (void *) 1UL
#define UART2 (void *) 2UL
ThreadID_1_0 = osThreadNew (thread1, UART1, &ThreadAttr_Task1);
ThreadID_1_1 = osThreadNew (thread1, UART0, &ThreadAttr_Task1);
```

可连接线程

CMSIS-RTOS2 中的一个新特性是能够创建处于“可连接”状态的线程。这使得线程可以作为标准线程创建和执行。此外，第二个线程可以通过调用 `osThreadJoin()` 来加入它。这将会导致第二个线程重新调度并保持在等待状态，直到已加入的线程终止。这允许创建一个临时的可连接线程，它将从全局内存池中获取一块内存，这个线程可以执行一些处理，然后终止，将内存释放回内存池。可连接线程可以通过在线程属性结构中设置可连接属性位来创建，如下所示：

```
static const osThreadAttr_t ThreadAttr_worker =
{
    .attr_bits = osThreadJoinable
};
```

一旦创建了线程，它将按照与“正常”线程相同的规则执行。然后，任何其他线程都可以通过使用操作系统调用加入它：

```
osThreadJoin(<joinable_thread_handle>);
```

一旦 `osThreadJoin()` 被调用，线程将重新调度并进入等待状态，直到可连接的线程终止。

时间管理

除了将应用程序代码作为线程运行外，RTOS 还提供了一些可以通过 RTOS 系统调用访问的定时服务。

时间延迟

在所有的定时服务中，最基本的一个就是延时函数。在应用程序中是提供时间延迟的一种简单方法。尽管 RTOS 内核大小为 5 KB，但是在非 RTOS 应用程序中经常会用到延迟循环和简单调度循环等功能，无论如何都会消耗一定字节的代码，因此 RTOS 的开销可能比它立即显示的要少。

```
void osDelay (uint32_t ticks);
```

这个调用会使当前调用线程进入 **WAIT_DELAY** 状态并持续指定的时间 (毫秒)。调度器将把执行传递给下一个处于 **READY** 状态的线程。

当定时时间结束时，线程将离开 **WAIT_DELAY** 状态并进入 **READY** 状态。当调度器将线程移至 **RUNNING** 状态时，线程将恢复运行。如果线程在以后的执行过程中没有任何阻塞操作系统调用，它将在其时间片结束时重新调度，并置于 **READY** 状态，假设另一个具有相同优先级的线程准备运行。

绝对延时

除了 **osDelay()** 函数，它提供了从被调用的那一刻开始的相对时间延迟，还有一个延时函数，它使线程暂停到特定的时间点：

```
osStatus osDelayUntil (uint32_t ticks);
```

osDelayUntil() 函数将暂停线程，直到达到特定的内核定时器 Tick 计数值。有许多内核函数允许您读取当前 **SysTick** 计数值和内核 **Tick** 计数值。

表 17. 线程块内容

内核定时器函数
<code>uint64_t osKernelGetTickCount(void)</code>
<code>uint32_t osKernelGetTickFreq(void)</code>
<code>uint32_t osKernelGetSysTimerCount(void)</code>
<code>uint32_t osKernelGetSysTimerFreq(void)</code>

虚拟定时器

CMSIS-RTOS API 可以用来定义任意数量的虚拟定时器，作为向下计数定时器。当定时时间结束时，将运行一个用户回调函数来执行特定的操作。每个定时器可以配置为单次或重复定时器。通过首先定义一个定时器结构来创建虚拟定时器：

```
static const osThreadAttr_t ThreadAttr_app_main =
{
    const char * name // symbolic name of the timer
    uint32_t attr_bits // None
    void* cb_mem // pointer to memory for control block
    uint32_t cb_size // size of memory control block
}
```

这将定义定时器的名称和回调函数的名称。定时器必须由 RTOS 线程实例化：

```
osTimerId_t timer0_handle;
timer0_handle = osTimerNew(&callback, osTimerPeriodic, (void *)<parameter>,
&timerAttr_timer0);
```

这将创建定时器并将其定义为周期性定时器或单次定时器 (**osTimerOnce()**)。下一个形参在定时时间结束时传递一个实参给回调函数：

```
osTimerStart (timer0_handle, 0x100);
```

然后可以在线程中的任何点启动定时器，定时器启动函数通过其句柄调用定时器，并以内核节拍数定义计数周期。

空闲线程

RTOS 提供的最后一个定时服务并不是真正的定时器，但这里是适合讨论它的地方。如果在我们的 RTOS 程序中，没有线程正在运行，也没有线程准备运行 (例如，它们都在等待延时函数)，那么 RTOS 将开始运行空闲线程。这个线程在 RTOS 启动并以最低优先级运行时自动创建。空闲线程函数位于 **RTX_Config.c** 文件中：

```
__WEAK__ __NO_RETURN void osRtxIdleThread (void *argument)
{
    (void)argument;
    for (;;) {}
}
```

您可以向该线程添加任何代码，但它必须遵循与用户线程相同的规则。Idle Demon 最简单的用法是当单片机不做任何事情时将其置于低功耗模式。

```
__WEAK__ __NO_RETURN void osRtxIdleThread (void *argument)
{
    (void)argument;
    for (;;)
    {
        __WFE();
    }
}
```

接下来会发生什么取决于单片机中选择的电源模式。至少，CPU 将暂停，直到 SysTick 定时器产生中断，并恢复调度器的执行。如果有线程准备好运行，那么应用程序代码的执行将恢复。否则，Idle Demon 将重新进入，系统将返回睡眠状态。

线程间通信

到目前为止，我们已经了解了如何将应用程序代码定义为独立线程，以及如何访问 RTOS 提供的定时服务。在实际的应用程序中，我们需要能够在线程之间进行通信，以使应用程序有用。为此，典型的 RTOS 支持几种不同的通信对象，这些通信对象可以用来将线程链接在一起，形成一个有意义的程序。CMSIS-RTOS2 API 支持线程间通信，包括线程和事件标志、信号量、互斥锁、邮箱和消息队列。在第一节中，关键概念是并发。在本节中，关键概念是同步多个线程的活动。

线程标志

Keil RTX5 支持每个线程最多 32 个线程标志。这些线程标志存储在线程控制块中。可以暂停线程的执行，直到系统中的另一个线程设置了特定的线程标志或线程标志组。

osThreadFlagsWait() 系统调用将挂起线程的执行，并将其置于 wait_evnt 状态。在 osThreadFlagsWait() API 调用中设置的至少一个标志设置完成后，线程才会开始执行。还可以定义一个周期性超时，在该超时之后等待线程将移回就绪状态，以便在调度器选择时恢复执行。osWaitForever (0xFFFF) 的值定义了一个无限的超时周期。

```
osEvent osThreadFlagsWait (int32_t flags,int32_t options,uint32_t timeout);
```

线程标志选项如下：

表 18. 线程标志选项

选项	描述
osFlagsWaitAny	等待设置任意标志 (默认)
osFlagsWaitAll	等待设置所有标志
osFlagsNoClear	不要清除已指定等待的标志

如果指定了一个标志模式，则在设置任何一个指定标志 (逻辑或) 时，线程将恢复执行。如果使用了 osFlagsWaitAll 选项，那么必须设置模式中的所有标志 (逻辑与)。任何线程都可以在任何其他线程上设置标志，并且线程可以清除自己的标志：

```
int32_t osThredFlagsSet (osThreadId_t thread_id, int32_t flags);
int32_t osThreadFlagsClear (int32_t signals);
```

事件标志

事件标志的操作方式与线程标志类似，但必须创建，然后作为全局 RTOS 对象，供所有正在运行的线程使用。

首先，我们需要创建一组事件标志，这与创建线程的过程类似。我们定义一个事件标志属性结构。如果使用的是静态内存模型，则属性结构定义了 ASCII 名称字符串、属性位和内存保留。

```
osEventFlagsAttr_t
{
    const char *name;      ///< name of the event flags
    uint32_t attr_bits;    ///< attribute bits (none)
    void *cb_mem;          ///< memory for control block
    uint32_t cb_size;      ///< size of provided memory for control block
};
```

接下来我们需要一个句柄来控制对事件标志的访问：

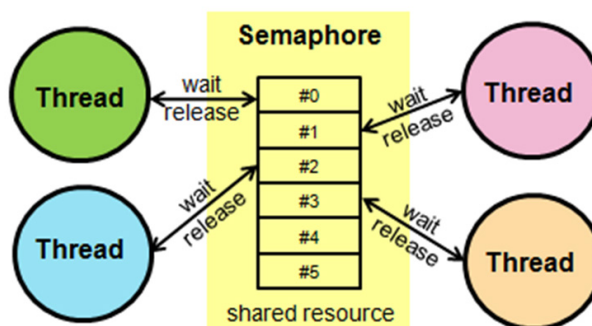
```
osEventFlagsId_t EventFlag_LED;
```

然后我们可以创建事件标志对象：

```
EventFlag_LED = osEventFlagsNew(&EventFlagAttr_LED);
```

信号量

与线程标志一样，信号量是一种在两个或多个线程之间同步活动的方法。简单地说，信号量是一个包含了许多令牌的容器。当线程执行时，它将到达 RTOS 调用以获取信号量令牌。如果信号量包含一个或多个令牌，则线程将继续执行，并且信号量中的令牌数量将减少 1。如果当前信号量中没有令牌，线程将处于等待状态，直到有令牌可用。在执行过程中的任何时候，线程都可以向信号量添加令牌，使其令牌计数增加 1。



上图说明了如何使用信号量来同步两个线程。首先，必须创建信号量并用初始令牌计数进行初始化。在这种情况下，信号量使用单个令牌初始化。两个线程都将运行并到达代码中的某个点，在那里它们将尝试从信号量获取令牌。到达该点的第一个线程将从信号量获取令牌并继续执行。第二个线程也将尝试获取令牌，但由于信号量为空，它将停止执行并进入等待状态，直到信号量令牌可用。

同时，执行线程可以将令牌释放回信号量。当这种情况发生时，等待线程将获得令牌并离开等待状态，进入就绪状态。一旦进入就绪状态，调度器将把线程置于运行状态，以便线程可以继续执行。虽然信号量有一组简单的操作系统调用，但它们可能是很难完全理解的操作系统对象之一。在本节中，我们将首先了解如何将信号量添加到 RTOS 程序中，然后继续研究更有用的信号量应用程序。

要在 CMSIS-RTOS 中使用信号量，必须先声明信号量属性：

```
osSemaphoreAttr_t
{
    const char *name;      ///< name of the semaphore
    uint32_t attr_bits;    ///< attribute bits (none)
};
```



```
void *cb_mem;          /// < memory for control block
uint32_t cb_size;      /// < size of provided memory for control block
};
```

接下来声明信号量句柄:

```
osSemaphoreId_t sem1;
```

然后, 在线程中, 可以使用多个令牌初始化信号量容器:

重要的是要了解, 信号量令牌也可能在线程运行时创建和销毁。例如, 可以用零令牌初始化一个信号量, 然后使用一个线程在信号量中创建令牌, 而另一个线程删除它们。这使得您可以将线程设计为生产者线程和消费者线程。

一旦信号量初始化, 就可以以与事件标志类似的方式获取令牌并发送给信号量。`osSemaphoreAcquire()` 调用用于阻塞线程, 直到信号量令牌可用。也可以指定超时周期为 `0xFFFF` 表示无限等待。

```
osStatus osSemaphoreAcquire(osSemaphoreId_t semaphore_id, uint32_t ticks);
```

一旦线程使用完信号量资源后, 可以向信号量容器发送令牌:

```
osStatus osSemaphoreRelease(osSemaphoreId_t semaphore_id);
```

发送信号 (Signaling)

同步两个线程的执行是信号量的最简单用法:

```
osSemaphoreId_t sem1;
static const osSemaphoreAttr_t semAttr_SEM1 =
{
    .name = "SEM1",
};
void thread1(void)
{
    sem1 = osSemaphoreNew(5, 0, &semAttr_SEM1);
    while (1)
    {
        FuncA();
        osSemaphoreRelease(sem1)
    }
}
void task2(void)
{
    while (1)
    {
        osSemaphoreAcquire(sem1, osWaitForever) FuncB();
    }
}
```

在这种情况下, 信号量用于确保 `FuncA()` 中的代码在 `FuncB()` 中的代码之前执行。

复用 (Multiplex)

复用是用来限制可以访问临界代码区的线程数。例如, 对于内存资源程序的访问, 只能支持有限数量的调用。

```
osSemaphoreId_t multiplex;
static const osSemaphoreAttr_t semAttr_Multiplex =
{
    .name = "SEM1",
};
void thread1(void)
{
    multiplex = osSemaphoreCreate(5, 5, &semAttr_Multiplex);
    while (1)
```

```
{
    osSemaphoreAcquire(multiplex, osWaitForever);
    processBuffer();
    osSemaphoreRelease(multiplex);
}
```

在本例中，我们给复用信号量初始化了五个令牌。当一个线程要调用 `processBuffer()` 函数时，就必须首先获取信号量令牌。一旦此函数结束，令牌就会被发送回信号量。如果超过五个线程试图调用 `processBuffer()`，第六个线程必须等待其中一个线程完成 `processBuffer()` 并归还令牌。因此，复用信号量确保了最多有五个线程可以“同时”调用 `processBuffer()` 函数。

交汇 (Rendezvous)

信号量信号的一种更通用的形式是交汇。交汇确保两个线程都到达某个执行点。在双方到达交汇点之前，任何一方都不可以继续运行。

```
osSemaphoreId_t arrived1, arrived2;
static const osSemaphoreAttr_t semAttr_Arrived1 =
{
    .name = "Arr1",
};
static const osSemaphoreAttr_t semAttr_Arrived2 =
{
    .name = "Arr2",
};
void thread1(void)
{
    arrived1 = osSemaphoreNew(2, 0);
    arrived1 = osSemaphoreNew(2, 0);
    while (1)
    {
        FuncA1();
        osSemaphoreRelease(arrived1);
        osSemaphoreAcquire(arrived2, osWaitForever);
        FuncA2();
    }
}
void thread2(void)
{
    while (1)
    {
        FuncB1();
        os_semaphore_Release(arrived2);
        os_semaphore_Acquire(arrived1, osWaitForever);
        FuncB2();
    }
}
```

在上面的例子中，两个信号量会确保两个线程发生交汇，然后继续执行 `FuncA2()` 和 `FuncB2()`。

屏障旋转栅门 (Barrier Turnstile)

尽管交汇对于同步代码的执行非常有用，但它只适用于两个函数。屏障是一种更通用的交汇形式，用于同步多个线程。

```
osSemaphoreId_t count, barrier;
static const osSemaphoreAttr_t semAttr_Counter =
{
    .name = "Counter",
};
static const osSemaphoreAttr_t semAttr_Barrier =
{
    .name = "Barrier",
};
unsigned int count;
void thread1(void)
{
    Turnstile_In = osSemaphoreNew(5, 0, &semAttr_SEM_In);
    Turnstile_Out = osSemaphoreNew(5, 1, &semAttr_SEM_Out);
    Mutex = osSemaphoreNew(1, 1, &semAttr_Mutex);
    while (1)
    {
        osSemaphoreAcquire(Mutex, osWaitForever); // Allow one task at a time to
        // access the first turnstile
        count = count + 1; // Increment count
        if (count == 5)
        {
            osSemaphoreAcquire(Turnstile_Out,
                               osWaitForever); // Lock the second turnstile
            osSemaphoreRelease(Turnstile_In); // Unlock the first turnstile
        }
        osSemaphoreRelease(Mutex); // Allow other tasks to access the turnstile
        osSemaphoreAcquire(Turnstile_In, osWaitForever); // Turnstile Gate
        osSemaphoreRelease(Turnstile_In);
        critical_Function();
    }
}
```

在这段代码中，我们使用一个全局变量来计算到达屏障的线程数。当每个函数到达屏障时，它将等待，直到可以从计数器信号量中获取令牌。一旦获得，**count** 变量将加 1。一旦 **count** 变量增加了，就会向计数器信号量发送一个令牌，以便其他等待的线程可以继续。接下来，屏障代码读取 **count** 变量。如果等于等待到达屏障的线程数，将向屏障信号量发送一个令牌。

在上面的例子中，我们同步了五个线程。前四个线程将递增 **count** 变量，然后在屏障信号量处等待。到达的第五个也是最后一个线程将递增 **count** 变量并向屏障信号量发送一个令牌。这使得它可以立即获得一个屏障信号量令牌并继续执行。通过屏障后，它立即向屏障信号量发送另一个令牌。从而使得其中一个等待线程恢复执行。这个线程在屏障信号量中放置另一个令牌，触发另一个等待线程，依此类推。屏障代码的最后一部分称为旋转门，因为它一次只允许一个线程通过屏障。在我们的并发执行模型中，这意味着每个线程都要在屏障处等待，直到最后一个线程到达，然后所有线程同时恢复。

信号量警告 (Semaphore Caveats)

信号量是任何 RTOS 的一个非常有用的特性。然而，信号量可能会被滥用。您必须始终记住，信号量中的令牌数量不是固定的。

在程序运行期间，可以创建和销毁信号量令牌。有时这很有用的，但是如果您的代码依赖于有固定数量的令牌可供信号量使用，那么必须非常小心，以便始终将令牌返回给它。您还应该排除意外创建额外新令牌的可能性。

互斥锁

Mutex (互斥锁) 是“Mutual Exclusion”的缩写。实际上,互斥锁是信号量的特殊版本。与信号量一样,互斥锁也是一个令牌容器。不同之处在于,互斥锁只能包含一个无法创建或销毁的令牌。互斥锁的主要用途是控制对芯片资源(如外设)的访问。因此,互斥锁令牌是二进制和有界的。除此之外,它的工作方式与信号量相同。首先,我们必须声明互斥锁容器并初始化互斥锁:

```
osMutexId_t uart_mutex;  
osMutexAttr_t  
{  
    const char *name;    ///< name of the mutex  
    uint32_t attr_bits;  ///< attribute bits  
    void *cb_mem;  
    ///< memory for control block  
    uint32_t cb_size;    ///< size of provided memory for control block  
};
```

创建互斥锁时,可以通过设置以下属性位来修改其功能:

表 19. 互斥锁

Bitmask	描述
osMutexRecursive	同一个线程可以多次使用互斥锁而不锁定自己。
osMutexPrioInherit	当一个线程拥有互斥锁时,不会被更高优先级的线程抢占。
osMutexRobust	通知获取互斥锁的线程之前的所有者已终止。

一旦声明了互斥锁,就必须在线程中创建。

```
uart_mutex = osMutexNew(&MutexAttr);
```

然后,任何需要访问外设的线程都必须先获取互斥锁令牌:

```
osMutexAcquire(osMutexId_t mutex_id,uint32_t ticks);
```

最后,当我们用完外设后,必须释放互斥锁:

```
osMutexRelease(osMutexId_t mutex_id);
```

使用互斥锁比使用信号量要严格得多,但是在控制对底层芯片寄存器的绝对访问时,它是一种更安全的机制。

数据交换

到目前为止,所有的线程间通信方法都只用于触发线程的执行;它们不支持线程之间交换程序数据。显然,在实际的程序中,我们需要在线程之间移动数据。这可以通过读取和写入全局声明的变量来实现。在除了非常简单的程序之外的任何程序中,试图保证数据完整性将非常困难,并且容易出现不可预见的错误。

线程之间的数据交换需要一种更正式的异步通信方法。

CMSIS-RTOS 提供了两种线程间数据传输的方法。第一种方法是一个消息队列,它在两个线程之间创建一个缓冲的数据“管道”。消息队列旨在传输整数值。

数据传输的第二种形式是邮件队列。这与消息队列非常相似,不同之处在于它传输的是数据块而不是单个整数。

消息队列和邮件队列都提供了在线程之间传输数据的方法。这允许您将设计视为由数据流相互连接的对象(线程)的集合。数据流由消息和邮件队列实现。这既提供了缓冲的数据传输,又在线程之间提供了定义良好的通信接口。从基于通过邮件和消息队列连接的线程的系统级设计开始,可以对工程的不同子系统进行编码,这在团队中工作时特别有用。此外,由于每个线程都有定义良好的输入和输出,因此很容易隔离以进行测试和代码重用。

消息队列

要设置消息队列，我们首先需要分配内存资源：

```
osMessag
eQId_t Q_LED;
osMessageQueueAttr_t
{
    const char *name;        ///< name of the message queue
    uint32_t attr_bits;      ///< attribute bits
    void *cb_mem;            ///< memory for control block
    uint32_t cb_size;        ///< size of provided memory for control block
    void *mq_mem;            ///< memory for data storage
    uint32_t mq_size;        ///< size of provided memory for data storage
};
```

一旦声明了消息队列句柄和属性，就可以在线程中创建消息队列：

```
Q_LED =
osMessageNew(DepthOfMessageQueue, WidthOfMessageQueue, &osMessageQueueAttr);
```

一旦创建了消息队列，就可以将数据从一个线程放入队列中：

```
osMessageQueuePut(Q_LED, &dataIn, messagePriority, osWaitForever);
```

然后从另一个线程的队列中读取：

```
result = osMessageQueueGet(Q_LED, &dataOut, messagePriority, osWaitForever);
```

扩展消息队列

在上一个范例中，我们定义了一个字的消息队列。如果需要发送大量数据，还可以定义一种消息队列，其中每个槽可以容纳更复杂的数据。首先，我们可以定义一个结构体来保存消息数据：

```
typedef struct
{
    uint32_t duration;
    uint32_t ledNumber;
    uint8_t priority;
} message_t;
```

然后，我们可以定义一个消息队列，格式化为接收以下类型的消息：

```
Q_LED = osMessageQueueNew(16, sizeof(message_t), &queueAttr_Q_LED);
```

内存池

我们可以设计一个消息队列来支持大量数据的传输。然而，此方法有一个开销，因为我们正在“移动”队列中的数据。在本节中，我们将着眼于设计一个更高效的“零拷贝”邮箱，其中数据保持静态。CMSIS-RTOS2 支持以固定块内存池的形式动态分配内存。首先，我们可以声明内存池属性：

```
osMemoryPoolAttr_t
{
    const char *name;        ///< name of the memory pool
    uint32_t attr_bits;      ///< attribute bits
    void *cb_mem;            ///< memory for control block
    uint32_t cb_size;        ///< size of provided memory for control block
    void *mp_mem;            ///< memory for data storage
    uint32_t mp_size;        ///< size of provided memory for data storage
} osMemoryPoolAttr_t;
```

以及内存池的句柄：

```
osMemoryPoolId_t mpool;
```

对于内存池本身，我们需要声明一个结构体，包含每个内存池槽中所需的内存元素：

```
typedef struct
{
    uint8_t LED0;
    uint8_t LED1;
    uint8_t LED2;
    uint8_t LED3;
} memory_block_t;
```

然后我们可以在应用程序代码中创建一个内存池：

```
mppool = osMemoryPoolNew(16, sizeof(message_t), &memorypoolAttr_mppool);
```

现在我们可以在线程中分配内存池槽：

```
memory_block_t *led_data;
led_data = (memory_block_t *) osMemoryPoolAlloc(mPool, osWaitForever);
```

然后用数据填充：

```
led_data->LED0 = 0;
led_data->LED1 = 1;
led_data->LED2 = 2;
led_data->LED3 = 3;
```

然后将指向内存块的指针放置在消息队列中：

```
osMessagePut(Q_LED, (uint32_t)led_data, osWaitForever);
```

数据现在可以被另一个任务访问：

```
osEvent event;
memory_block_t *received;
event = osMessageGet(Q_LED, osWaitForever);
received = (memory_block_t *)event.value.p;
led_on(received->LED0);
```

一旦内存块中的数据被使用，必须将该内存块释放回内存池以供重用。

```
osPoolFree(led_pool, received);
```

要创建零拷贝邮箱系统，我们可以将内存池与消息队列组合起来以存储数据，消息队列用于将指针传输到分配的内存池插槽。通过这种方式，消息数据将保持静态，并可以在线程之间传递指针。

HT32F67575 中的 RTX-RTOS

我们可以在 HT32F67575 SDK 中找到 RTOS 范例，例如：

SDK_Folder\projects\ble_examples\ble_peripheral\HT32F67575\project\cp\

启动线程

在 main.c 中，我们将初始化 RTOS 内核，创建应用程序主线程，然后开始线程的执行。

```
static osRtxThread_t appMainCB attribute ((aligned(4), section(".bss.os.thread.cb")));
static uint64_t appMainStack[64];
extern void app_main(void *argument);
int main(void)
{
    const osThreadAttr_t attr =
    {
        "app_main",
        osThreadJoinable,
        & appMainCB,
        sizeof(appMainCB),
    }
```

```

        & appMainStack[0],
        sizeof(appMainStack),
        osPriorityNormal,
        0
    };
    osKernelInitialize(); // Initialize CMSIS-RTOS
    osThreadNew(app_main, NULL, &attr); // Create application main thread
    osKernelStart(); // Start thread execution
    for (;;)
    return 0;
}

```

创建 LLC 线程

在 ble_task.c 中，我们将创建并启动一个 LLC 线程。

```

static osRtxThread_t appMainCB attribute ((aligned(4), section(".bss.os.thread.cb")));
static uint64_t appMainStack[64];
extern void app_main(void *argument);
int main(void)
{
    const osThreadAttr_t attr =
    {
        "app_main",
        osThreadJoinable,
        & appMainCB,
        sizeof(appMainCB),
        & appMainStack[0],
        sizeof(appMainStack),
        osPriorityNormal,
        0
    };
    osKernelInitialize(); // Initialize CMSIS-RTOS
    osThreadNew(app_main, NULL, &attr); // Create application main thread
    osKernelStart(); // Start thread execution
    for (;;)
    return 0;
}

```

创建 LLC 消息队列

下面的例子显示了如何创建 LLC 消息队列。

```

osMessageQueueId_t llcTaskMsgQueueId;
static uint32_t llcTaskMsgQueueCb[osRtxMessageQueueCbSize / 4U];
static uint32_t llcTaskMsgQueueMem[osRtxMessageQueueMemSize(LLC_TASK_MSG_COUNT,
sizeof(stLlcTaskMsg_t) / 4U)];
const osMessageQueueAttr_t llc_task_msg_queue_attr =
{
    .name = "llc_task_msg_queue",
    .attr_bits = 0,
    .cb_mem = llcTaskMsgQueueCb,
    .cb_size = sizeof(llcTaskMsgQueueCb),
    .mq_mem = llcTaskMsgQueueMem,
    .mq_size = sizeof(llcTaskMsgQueueMem),
};
llcTaskMsgQueueId = osMessageQueueNew(LLC_TASK_MSG_COUNT, sizeof(stLlcTaskMsg_t), &llc_task_msg_queue_attr);

```

发送消息到 LLC 线程

下面的例子显示了如何向 LLC 线程发送消息。

```
static bool ble_task_send_msg_to_llc_task(stLlcTaskMsg_t stLlcTaskMsg)
{
    uint32_t u32Timeout = osWaitForever;
    if(__get_IPSR())
    {
        u32Timeout = 0; // in interrupt
    }
    osStatus_t status = osMessageQueuePut(llcTaskMsgQueueId, &stLlcTaskMsg, 0,
                                          u32Timeout);
}
```

9 2nd-Boot

该单片机有三种类型的存储器：ROM、Flash 和 SRAM。ROM 用于存储不可修改的代码和数据；Flash 用于存储用户可修改的代码和数据。SRAM 用于存储运行代码和运行数据。该单片机存储器映射如下：

表 20. 存储器映射

存储器类型	起始地址	结束地址
SRAM	0x20000000	0x2003FFFF
Flash	0x10000000	0x1007FFFF
ROM	0x00000000	0x0003FFFF

存储器映射

单片机上电后，CPU 默认起始地址为 0x00000000 (ROM 中)。由于 ROM 代码无法修改，我们设计了用于加载 ROM 中的启动代码的代码，以帮助 CPU 正确跳转到 SDK。这个启动代码称为 2nd-Boot 代码。

2nd-Boot 代码存储在 Flash 中，并在运行时加载到 SRAM 中。2nd-Boot 代码是 ROM 代码和 SDK 之间的链接。单片机代码运行流程如下：

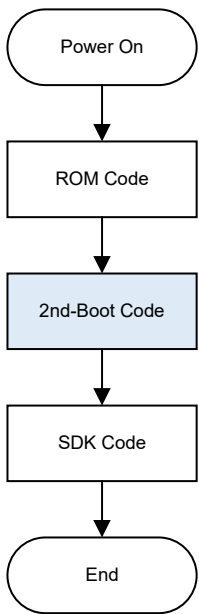


图 32. 启动流程

2nd-Boot 代码

2nd-Boot 代码有两个主要功能。首先，确定 OTA 代码区域中是否有新版本的应用层代码，并进行后续处理。其次是初始化并使能缓存，以便在 Flash 中运行的代码可以加载到缓存中。2nd-Boot 代码运行流程如下：

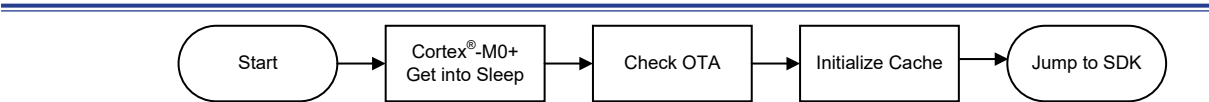


图 33. 2nd-Boot 代码流程

OTA 流程

我们将 Flash 分为四个区域，分别是：2nd-Boot 代码区、OTA 代码信息区、应用代码区和 OTA 代码区（每个区域的大小可由用户修改），具体如下：

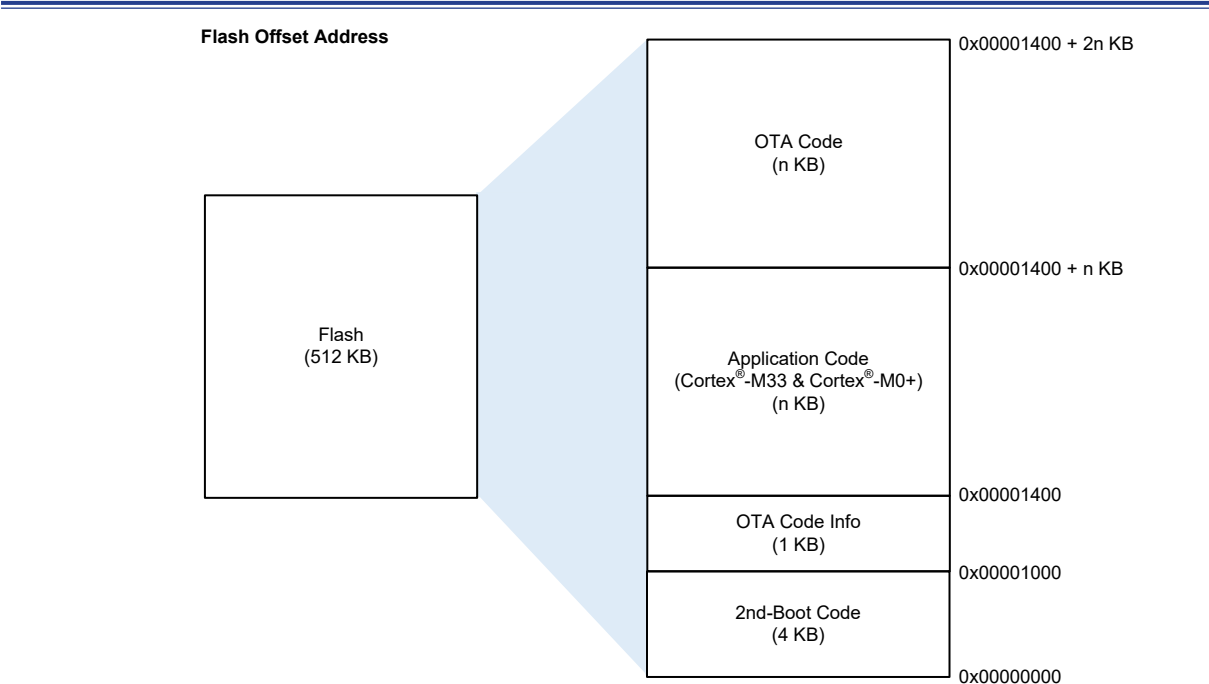


图 34. Flash 存储器映射

2nd-Boot 代码包括 2nd-Boot 代码；OTA 代码信息包括运行代码信息、OTA 代码大小和 CRC；应用代码包括 Cortex®-M33 和 Cortex®-M0+ 应用代码；OTA 代码包含需要升级的代码。2nd-Boot 的 OTA 流程包括以下 5 个主要步骤：

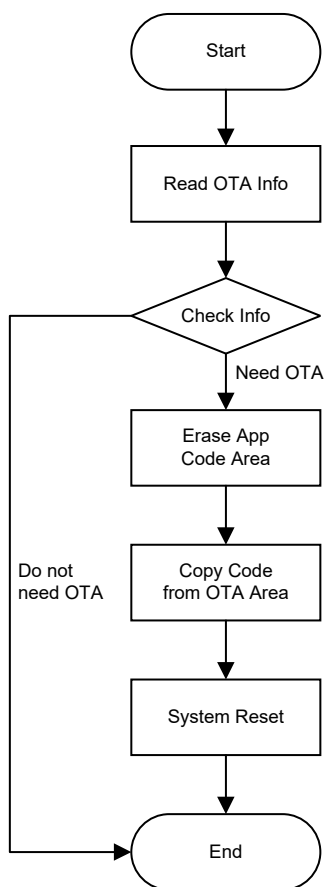


图 35. 2nd-Boot 的 OTA 流程

步骤 1. 从 Flash 中读取 OTA 信息，以检查是否需要升级。

如果不是，执行下一步，以初始化缓存；

如果是，我们将逐步检查 ROM 和 2nd-Boot 代码版本，OTA 代码 CRC。

步骤 2. 擦除应用代码，以准备将 OTA 代码复制到应用代码区。

步骤 3. 在 Flash 中从 OTA 代码区读取 OTA 代码，然后将其写入应用代码区。

步骤 4. 检查 OTA 代码和新应用代码的 CRC。

步骤 5. 如果两者相等，则修改 OTA 信息并复位系统。如果不相等，则重新执行步骤 2。

初始化缓存读取

初始化缓存读取包括设置缓存读取模式为 QSPI，和使能缓存读取。

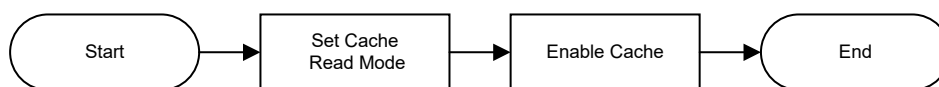


图 36. 初始化缓存

跳转到 SDK

在 2nd-Boot 代码执行完成后，代码将跳转到 SDK。我们将修改 PC 指针来复位 SDK 的处理程序。

Copyright© 2024 by HOLTEK SEMICONDUCTOR INC. All Rights Reserved.

本文件出版时 HOLTEK 已针对所载信息为合理注意，但不保证信息准确无误。文中提到的信息仅是提供作为参考，且可能被更新取代。HOLTEK 不担保任何明示、默示或法定的，包括但不限于适合商品化、令人满意的质量、规格、特性、功能与特定用途、不侵害第三方权利等保证责任。HOLTEK 就文中提到的信息及该信息之应用，不承担任何法律责任。此外，HOLTEK 并不推荐将 HOLTEK 的产品使用在会由于故障或其他原因而可能会对人身安全造成危害的地方。HOLTEK 特此声明，不授权将产品使用于救生、维生或安全关键零部件。在救生 / 维生或安全应用中使用 HOLTEK 产品的风险完全由买方承担，如因该等使用导致 HOLTEK 遭受损害、索赔、诉讼或产生费用，买方同意出面进行辩护、赔偿并使 HOLTEK 免受损害。HOLTEK (及其授权方，如适用) 拥有本文件所提供信息 (包括但不限于内容、数据、示例、材料、图形、商标) 的知识产权，且该信息受著作权法和其他知识产权法的保护。HOLTEK 在此并未明示或暗示授予任何知识产权。HOLTEK 拥有不事先通知而修改本文件所载信息的权利。如欲取得最新的信息，请与我们联系。