

# **RTL8762D SDK User Guide**

**V1.0**

**2020/06/05**

## 修订历史 (Revision History)

日期	版本	修改	作者	Reviewer
2020/04/28	Draft v0.1	初稿	Lory	
2020/06/05	V1.0	正式版	Lory	

Realtek Confidential

# 目录

修订历史 (Revision History)	2
图目录	6
表目录	7
1 概述	8
2 用户指南	9
2.1 开发环境	9
2.1.1 Keil	9
2.1.2 J-Link	9
2.1.3 SDK	10
2.1.4 EVB Kit	10
2.2 平台设置	10
2.3 EVB 安装	11
2.4 IDE 安装	11
2.4.1 Keil 工程	11
2.4.2 Keil 设置	11
3 硬件架构	15
4 软件架构	16
4.1 系统架构	16
4.2 操作系统	16
4.3 OS 接口	17
4.4 任务和优先级	17
4.4.1 任务	17
4.4.2 优先级	19
5 应用程序	20
5.1 SDK 目录	20
5.2 示例工程	21
5.3 流程图	22
5.4 消息和事件处理流程	23
5.5 IO 消息	24

5.5.1	消息格式 .....	24
5.5.2	消息类型定义 .....	24
5.5.3	消息子类型定义 .....	24
5.5.4	用户自定义消息 .....	25
5.6	Pin 设置.....	25
5.7	DLPS 设置 .....	25
6	存储 .....	27
6.1	存储映射 .....	27
6.2	ROM.....	27
6.3	RAM.....	27
6.3.1	Data Ram .....	27
6.3.2	Buffer Ram .....	28
6.4	Cache.....	29
6.5	Flash .....	29
6.5.1	Flash APIs.....	29
6.5.2	FTL .....	30
6.6	eFuse .....	30
7	中断 .....	31
7.1	Nested Vectored Interrupt Controller (NVIC) .....	31
7.2	中断向量表.....	31
7.3	中断优先级.....	34
8	电量管理 .....	35
9	烧录 .....	37
9.1	Images 相关 .....	37
9.2	Image 处理工具.....	37
9.2.1	fromelf .....	37
9.2.2	prepend_header.....	37
9.2.3	md5 .....	39
9.3	烧录方式.....	39
10	调试.....	39

10.1 Log 机制 .....	39
10.1.1 Debug analyzer .....	40
10.1.2 Log 打印基本接口 .....	42
10.1.3 Log 打印封装接口 .....	42
10.1.4 辅助打印接口 .....	43
10.1.5 Log 打印示例 .....	43
10.1.6 Log 控制接口 .....	44
10.1.7 DBG_DIRECT .....	45
10.2 SWD 调试 .....	45
10.2.1 SWD 调试接口 .....	46
10.2.2 DWT 监视接口 .....	47

## 图目录

图 10-1 Debug Analyzer 主页面.....	40
图 10-2 Debug Analyzer 设置页面.....	41

Realtek Confidential

## 表目录

表 5-1 APP 初始化 .....	23
表 7-1 中断向量表 .....	31
表 7-2 Timer 4~7 ISR .....	32
表 7-3 Peripheral ISR .....	33
表 7-4 GPIO Group3 ISR .....	33
表 7-5 GPIO Group2 ISR .....	33
表 7-6 GPIO Group1 ISR .....	34
表 7-7 GPIO Group0 ISR .....	34
表 7-8 中断优先级 .....	34
表 9-1 Images 可用烧录方式 .....	39
表 10-1 调试等级 .....	42

# 1 概述

本文将说明如何使用 RTL8762D 软件开发包（SDK）来开发蓝牙低功耗应用。

Realtek Confidential



## 2 用户指南

### 2.1 开发环境

以下是在开发过程中需要的软件和硬件：

1. Keil MDK-ARM Lite V5 或者更新版本
2. J-Link Software v6.32 或者更新版本
3. RTL8762D SDK
4. EVB Kit

#### 2.1.1 Keil

SDK 中所有的应用能够通过 Keil Microcontroller Development Kit(MDK)编译以及使用。所以在进行软件开发之前，要先取得并且安装 Keil。有关 Keil 的更多信息请访问 [www.keil.com](http://www.keil.com)。

Realtek 使用的工具链版本信息如下所示。为避免产生 ROM 可执行程序和应用之间的兼容性问题，建议使用以下版本或者更新的版本。

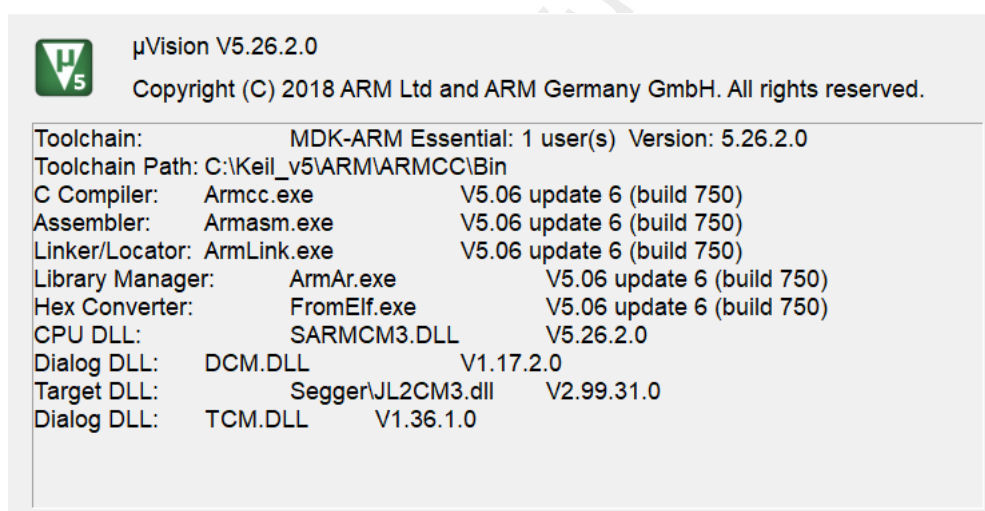


图 2-1 Keil

#### 2.1.2 J-Link

如果想要有比打印 log 机制更全面的调试方法，J-link 是另一个必须要具备的工具。访问 [www.segger.com](http://www.segger.com) 来下载相关的软件和文档，安装 SEGGER J-Link 软件。设备的驱动应该被正确地安装，才能搭配 Keil MDK 使用 J-Link 调试器。

## 2.1.3 SDK

SDK 包括示例工程，文档以及必要的工具。

## 2.1.4 EVB Kit

EVB 是针对 RTL8762D 的硬件评估，开发和应用调试设计的。

## 2.2 平台设置

图 2-1 展示了硬件部分和软件组件之间的关系。通过 Keil 开发的应用程序能够通过 J-Link 下载到 RTL8762D，并且通过 SWD 接口调试。

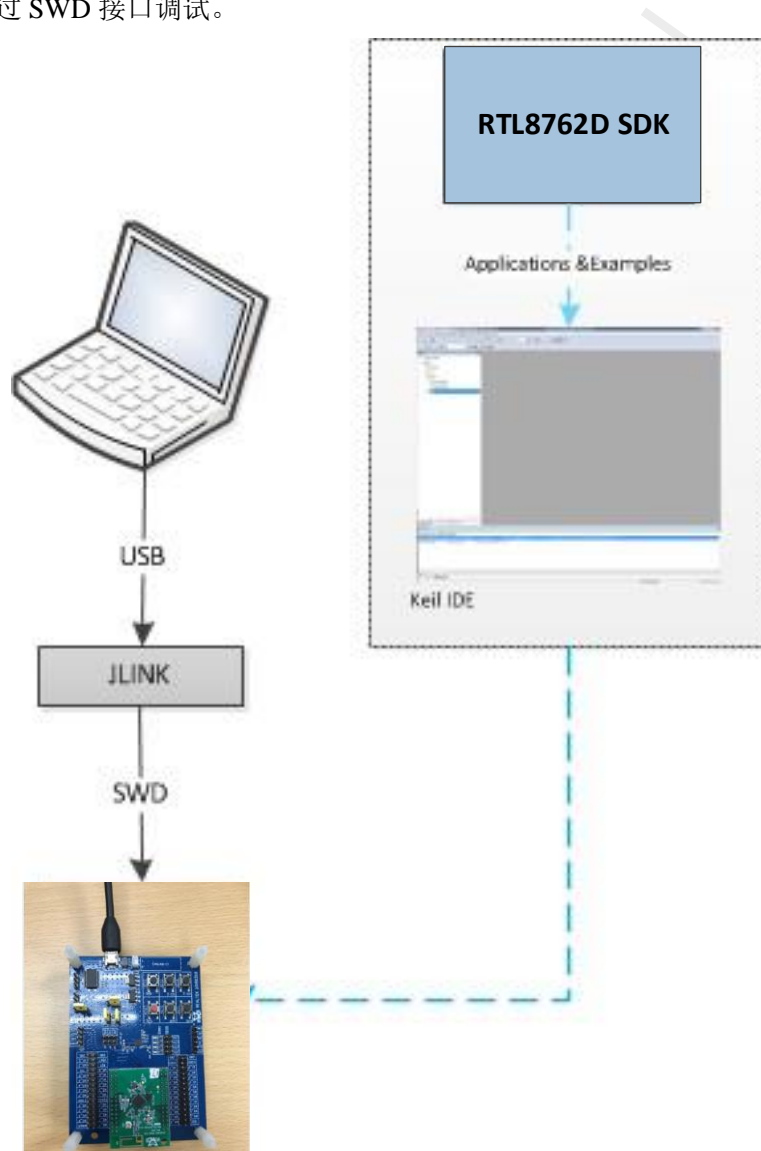


图 2-1 平台设置

## 2.3 EVB 安装

EVB 套件由一块母板和一块子板组成，详细请参考 RTL8762D Evaluation Board User Guide。

## 2.4 IDE 安装

### 2.4.1 Keil 工程

一般来说不建议新建工程开发，而是直接打开现有的示例工程，然后在此基础上增加功能代码。

如果安装的是 Keil V5 或者更新的版本，以 PXP 工程为例（工程目录在 s-bee2\_sdk\_xxxx\board\evb\pxp），打开方式如图 2-2 所示。

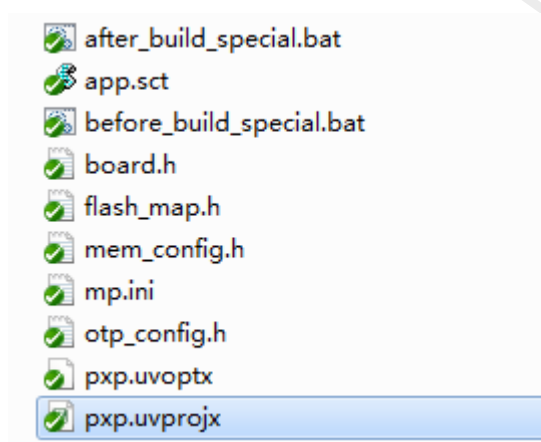


图 2-2 Keil 工程

### 2.4.2 Keil 设置

下面是对 Debugger 设定和 Keil 使用的 Flash 下载算法的描述：

1. 把 sdk\tool\flash 目录下的 RTL876x\_SPI\_FLASH.FLM 和 RTL876x\_LOG\_TRACE.FLM 文件拷贝到 Keil 安装目录：Keil\_installed\_dir\ARM\Flash\，如下所示：

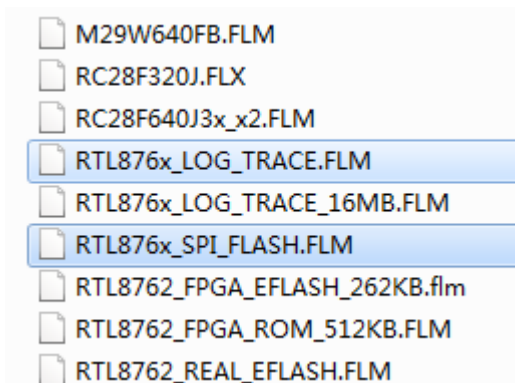



图 2-3 Keil Flash 算法

2. 点击工具栏上的, 或者在菜单中进入 Project > Options, 点击 Debug 页面。选择 J-LINK/J-TRACE Cortex, 然后点击“Settings”:

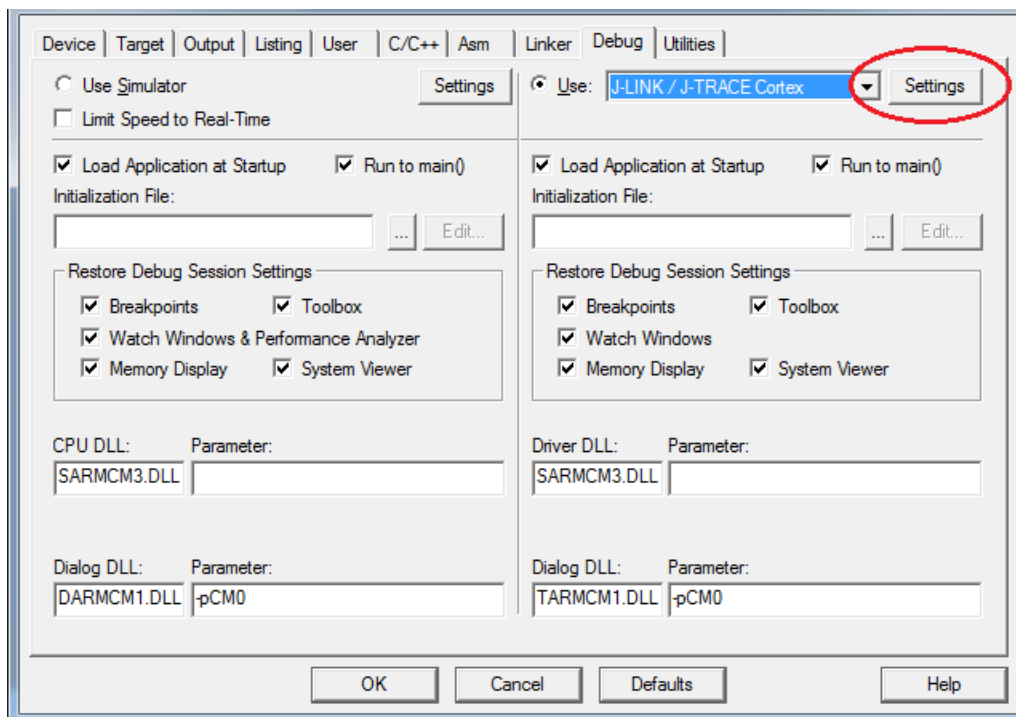


图 2-4 Keil 设置

3. J-Link Port 选择 SW。如果硬件连接是正确的, CPU 就能在 SW Device 列表中识别出来。

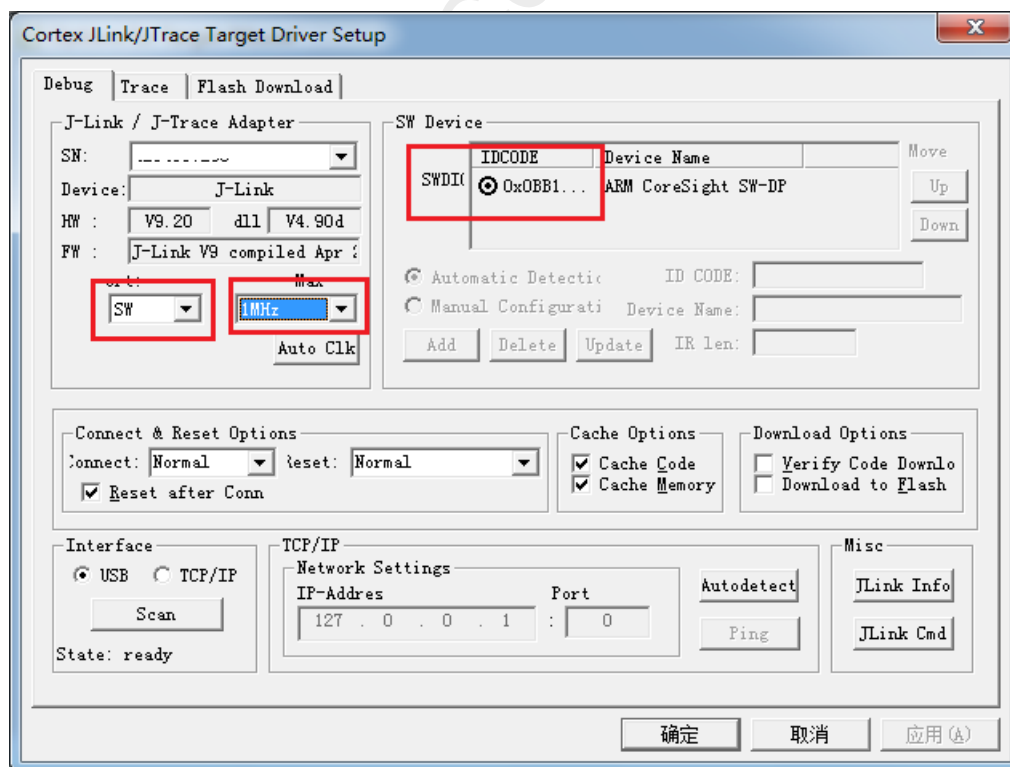


图 2-5 Keil 调试设置

- 选择“Flash Download”页面,如果已经有下载算法就先删除。点击“Add”添加 RTL876x\_SPI\_FLASH 和 RTL876x\_LOG\_TRACE。修改“RAM for Algorithm” 的起始地址为 0x00200000,大小为 0x4000。

Cortex JLink/JTrace Target Driver Setup

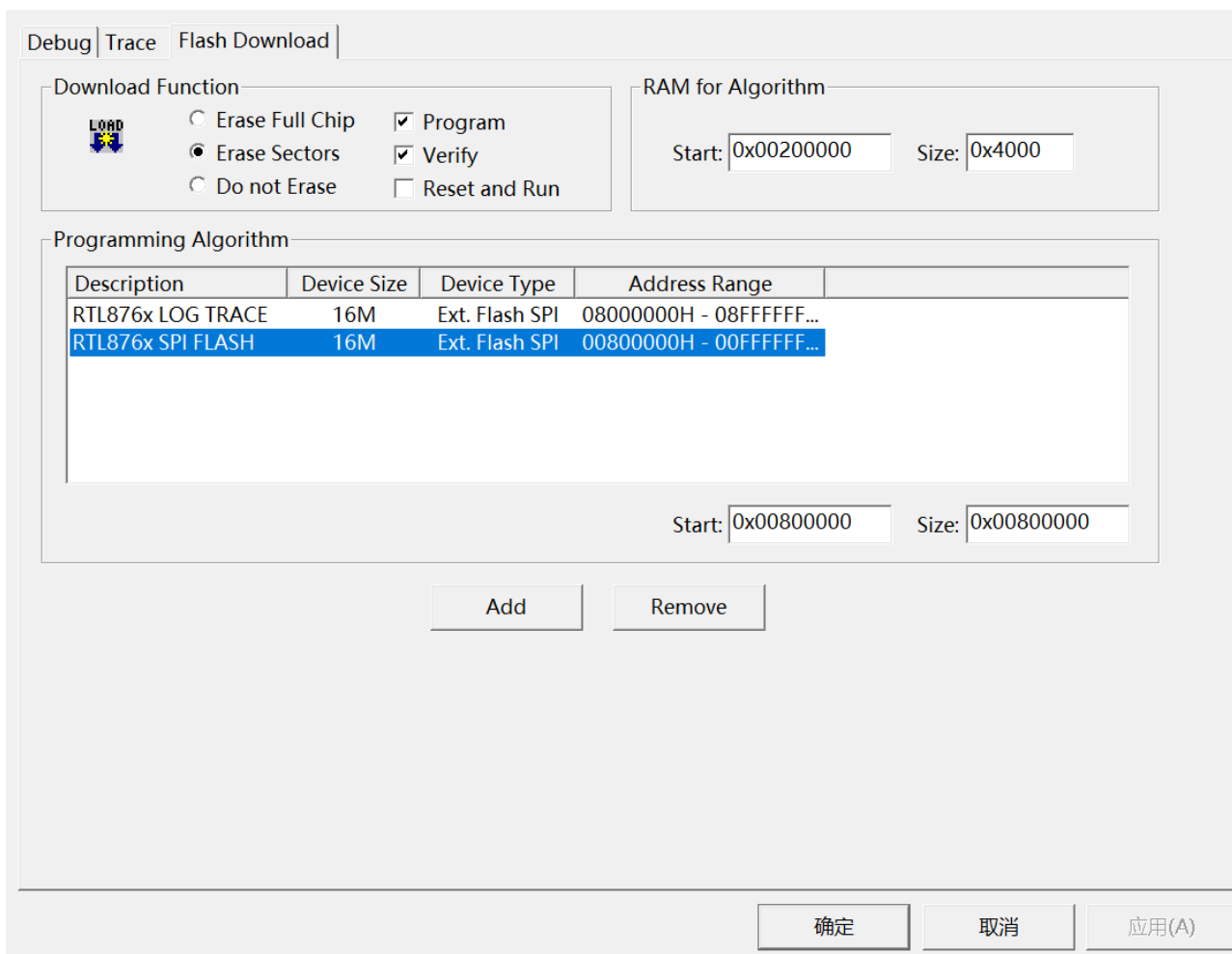




图 2-6 Flash 下载算法

- 点击图标  编译工程。如果有任何编译或者链接的错误,在解决完后就可以把应用程序下载到 RTL8762D 来做进一步的验证和调试。
- 如果 Flash 算法有成功的安装,就可以通过点击下载图标  来进行下载。在下载过程中,预期是不会产生任何出错的信息。当应用程序被成功下载到 RTL8762D 后,重启 RTL8762D 来运行程序。可以通过 Debug Analyzer 工具上的日志信息来检查应用程序是否按照预期运行。

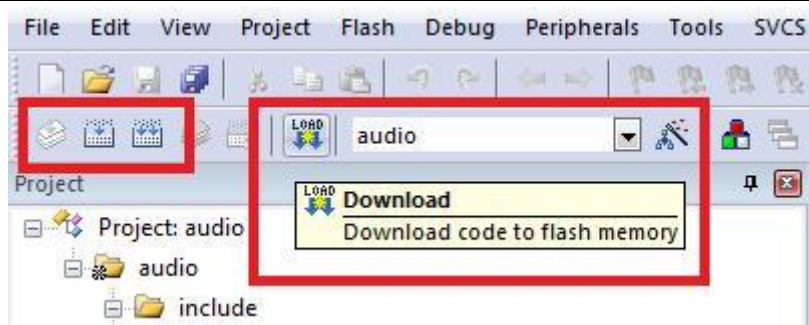



图 2-7 编译和运行

7. 通过点击调试图标，利用 J-Link 来调试和追踪应用程序。

**注意：**如果 DLPS 模式是打开的，那么一旦系统进入 DLPS 模式，调试就不能继续进行。所以想要在开发早期利用 Keil 来调试程序时，要先关闭 DLPS 模式。DLPS 打开和关闭方式：

打开：lps\_mode\_set (LPM\_DLPS\_MODE)

关闭：lps\_mode\_set (LPM\_ACTIVE\_MODE)

### 3 硬件架构

RTL8762D 的硬件模块如图 3-1 所示：

- 丰富的外设
- 灵活的 RAM 配置
- 电源管理单元
- 时钟管理单元
- 蓝牙模块

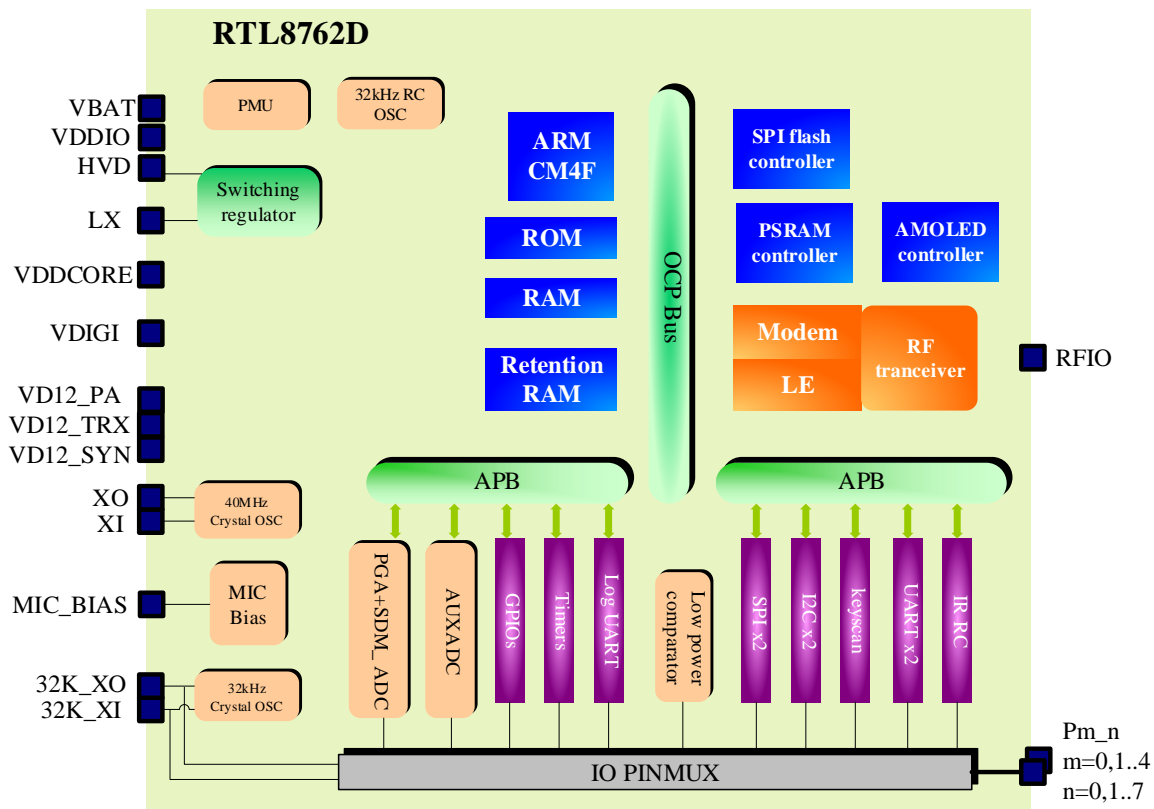


图 3-1 RTL8762D 硬件架构

## 4 软件架构

### 4.1 系统架构

如图 4-1 所示，软件架构包括几个主要的部分：

- GAP: 应用程序和 BLE 协议栈交互的抽象层。GAP 接口的详细信息见 GAP Interfaces User Manual;
- Platform: 包括 OTA, flash, FTL 等;
- IO Drivers: 提供应用层访问 RTL8762D 外设的接口，无需直接访问寄存器;
- OSIF: 实时操作系统的抽象层。

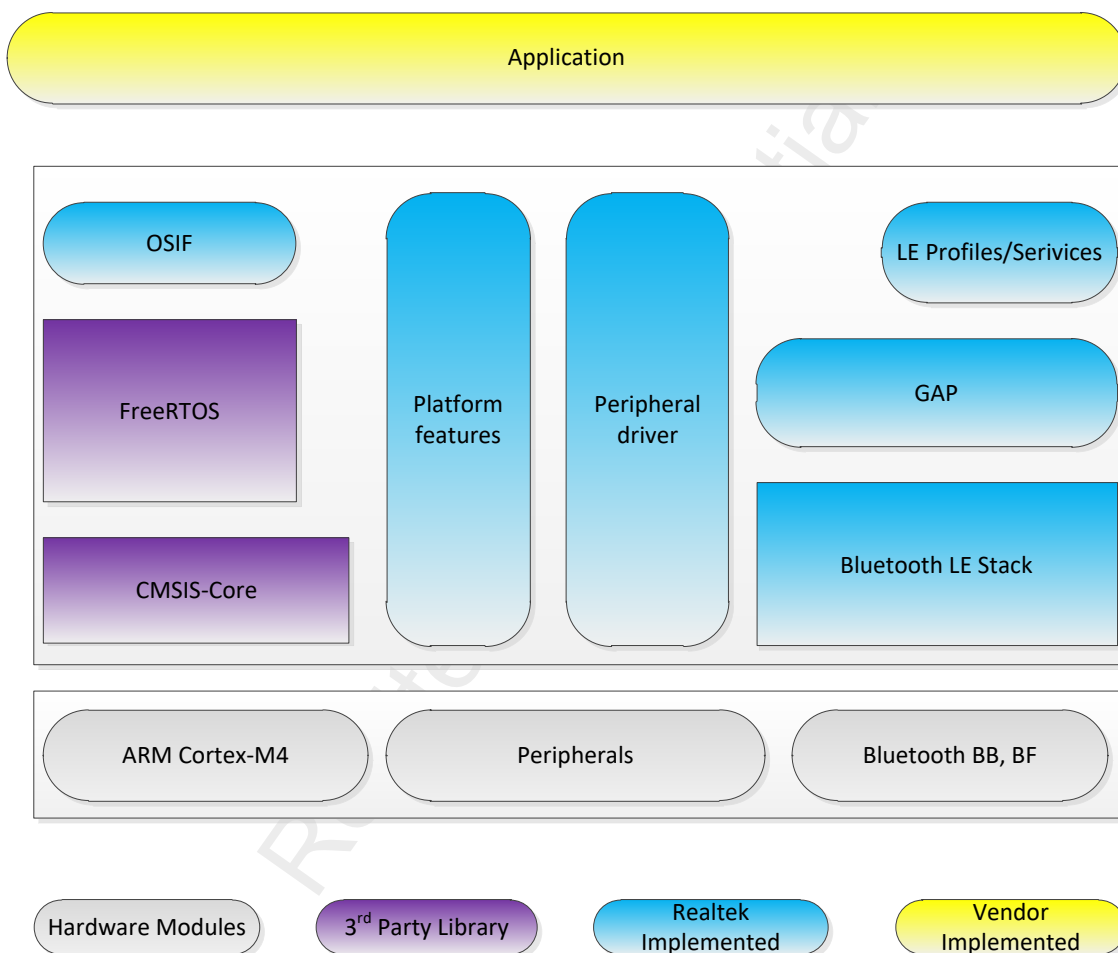


图 4-1 RTL8762D 软件架构

### 4.2 操作系统

RTL8762D 上使用的是操作系统是 FreeRTOS V10.2，该系统是运行在 ROM 上，并且包括以下几个模块：

1. 任务管理



2. 队列管理
3. 中断管理
4. 资源管理
5. 内存管理
6. 时间管理

## 4.3 OS 接口

OSIF 是实现在 RealBlue™ 工程的。因为项目需求可能会更新到特定的 RTOS 版本，甚至换成一个不同的 RTOS，所以需要提供一个 OSIF 层来清理不同 RTOS 间特定接口带来的不兼容问题。

如图 4-2 所示，OSIF 层通过封装特定的 RTOS 接口来提供一层统一的接口。使用 OSIF 的软件组件能够在 RealBlue™ 工程的演进中更好地移植。开发者可以基于 OSIF 层提供自定义的 RTOS 实现，而不需要在上层的软件做任何修改。所以，强烈推荐在软件开发中使用 OSIF 接口，而不是直接访问特定 RTOS 的接口。

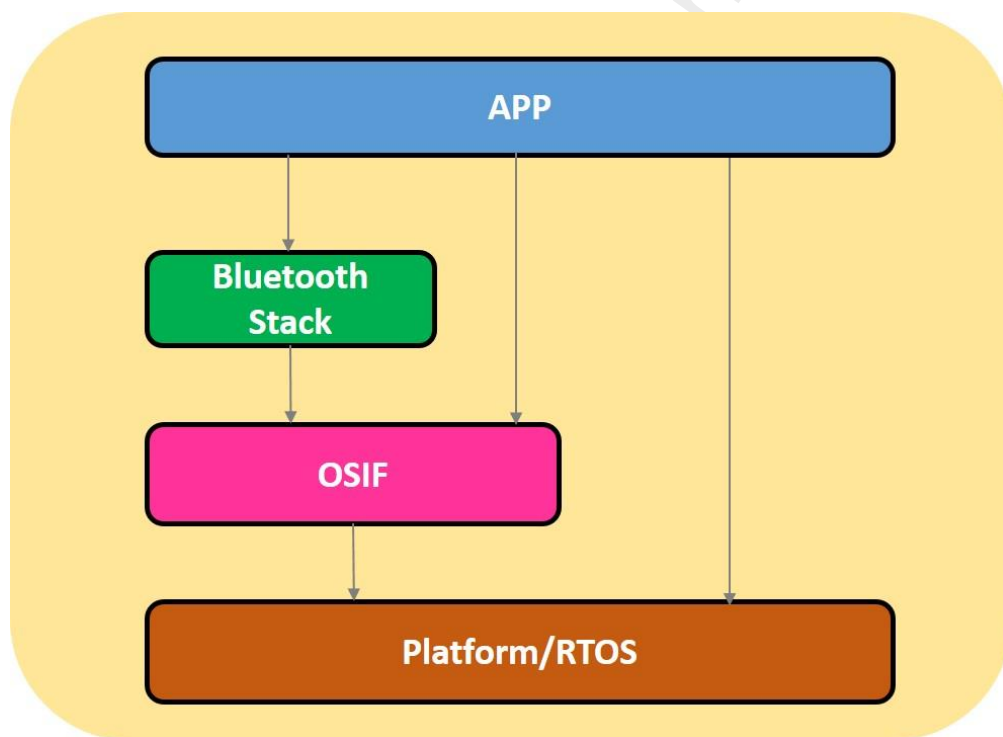


图 4-2 OSIF 架构

## 4.4 任务和优先级

### 4.4.1 任务

如图 4-3 所示，应用程序创建了 5 个任务：

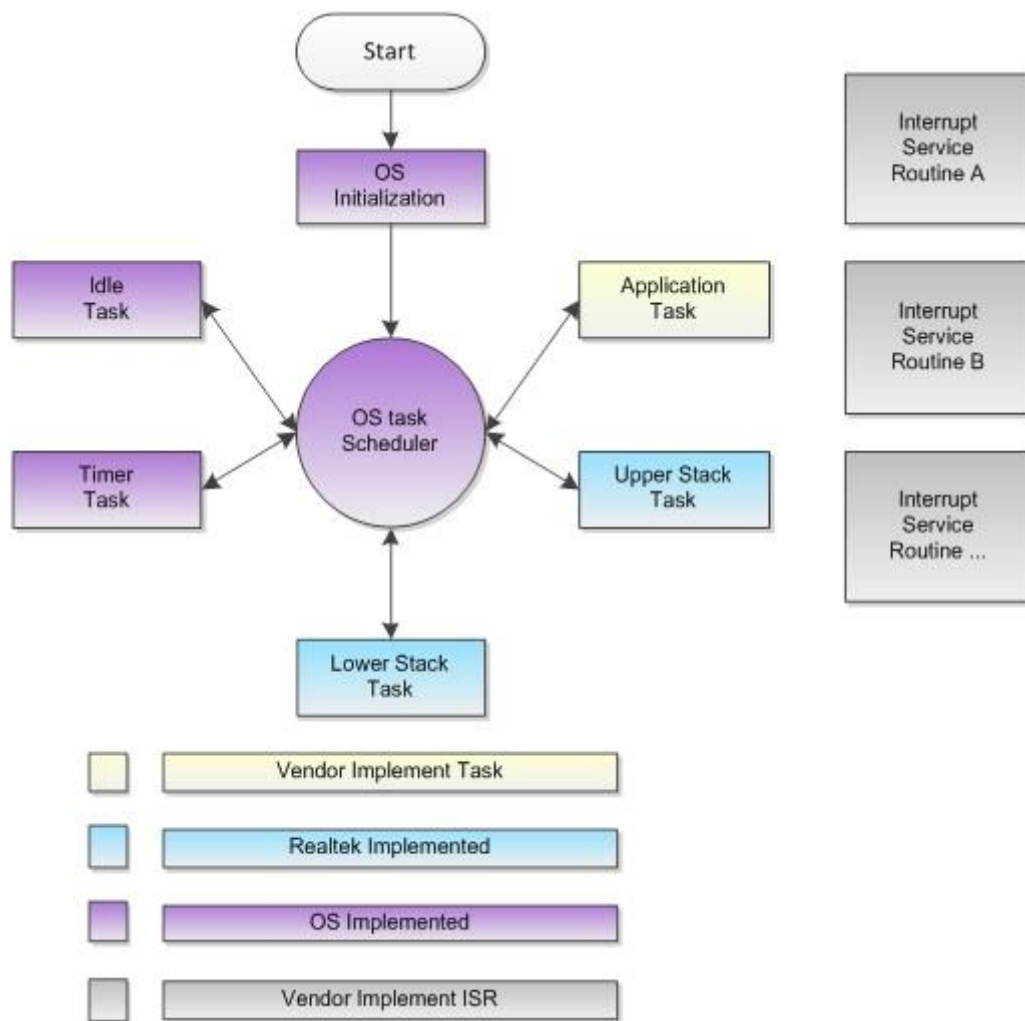


图 4-3 RTL8762D 任务

Task	Description	Priority
timer	FreeRTOS 实现的软件定时器	6
lower stack	蓝牙 HCI 层以下的协议栈实现	6
upper stack	蓝牙 HCI 层以上的协议栈实现	5
app	应用层功能的实现	1~4
Idle	后台空闲任务，包括低功耗的处理	0

说明：

- 可以创建多个 APP 任务，同时内存资源也会被相应地分配；
- Idle 任务和 Timer 任务是 FreeRTOS 创建的；
- 任务是配置成根据优先级抢占式的；
- 硬件中断服务例程（ISR）是由 Vendor 来实现的。

## 4.4.2 优先级

任务分 7 种优先级：Priority<sub>SW Timer</sub> = Priority<sub>lower stack</sub> > Priority<sub>upper stack</sub> > Priority<sub>app</sub> > Priority<sub>idle</sub>。APP 优先级可从 1~4 中选择。

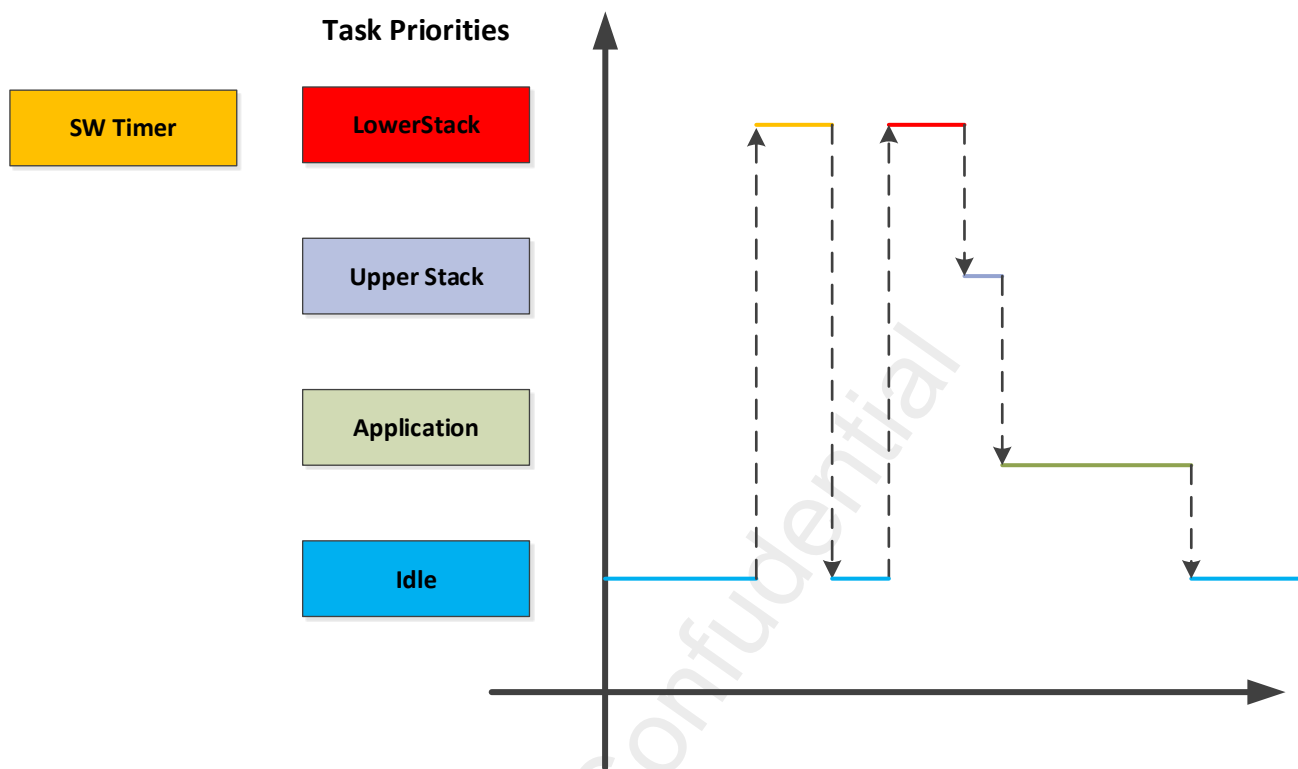


图 4-4 任务优先级

## 5 应用程序

### 5.1 SDK 目录

SDK 目录如下所示:

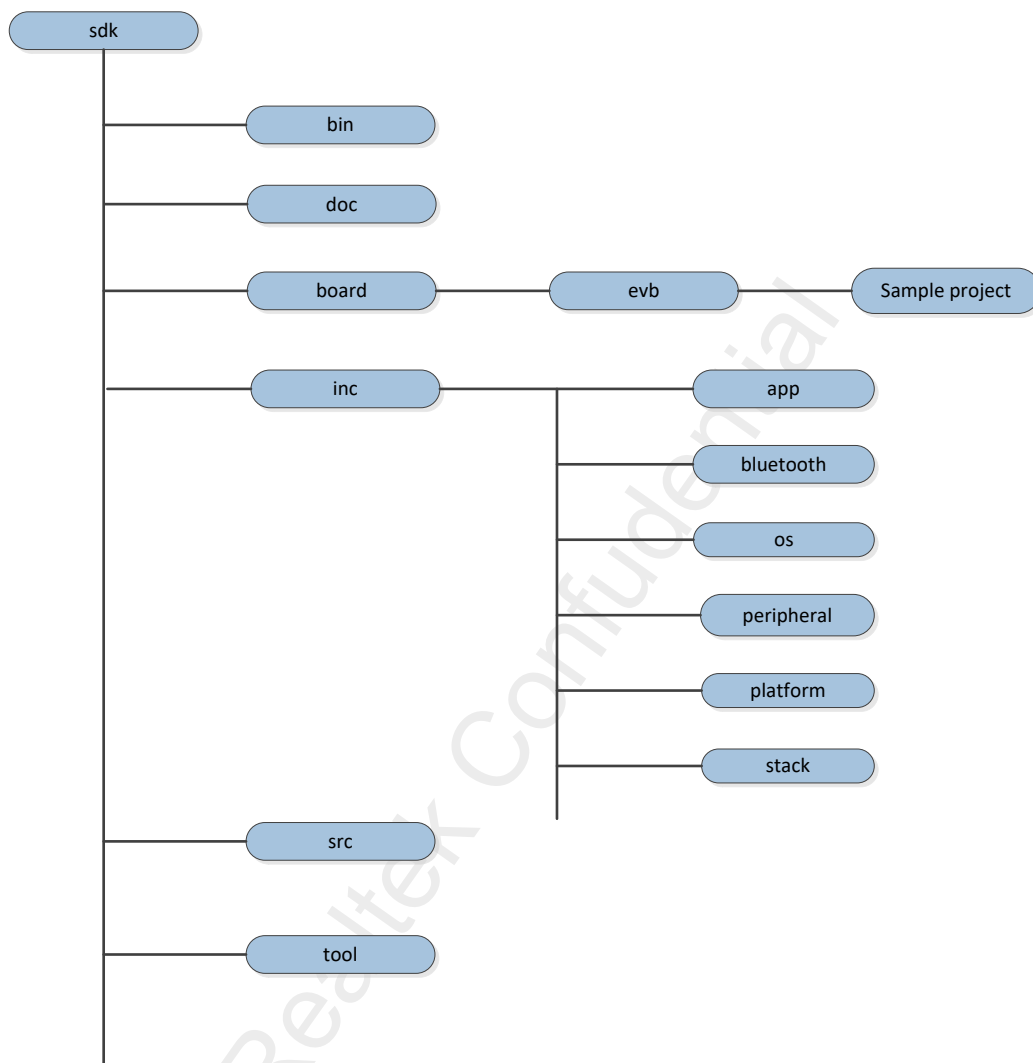


图 5-1 SDK 目录

Directory	Description
board	配置好可直接使用的 Keil 示例工程
doc	文档
inc	提供除了 ROM 符号以外的接口定义的头文件
bin	应用程序链接的二进制文件
src	应用程序的源码文件

## 5.2 示例工程

为了帮助创建应用程序，SDK 中已经创建好了许多示例工程，例如 PXP 以及一些 BLE 相关的示例工程。通过学习示例工程，开发者可以很容易地熟悉 SDK。所有的示例工程已经配置好，并且根据 RTL8762D SOC 划分好内存。

下面以 PXP 为例，展示如何通过示例工程来开发客制化应用程序。下面的图摘自当前 SDK，实际可能会随着 SDK 的更新而变化。

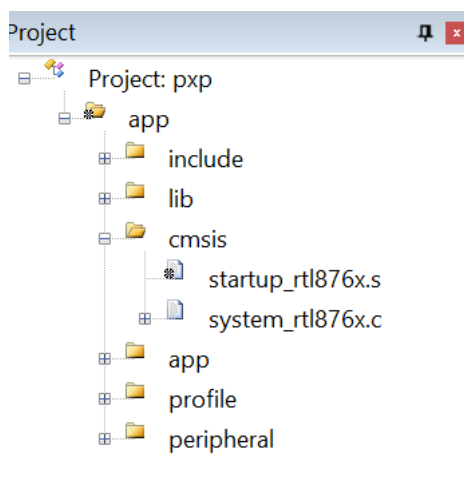


图 5-2 PXP 工程示例

PXP 工程中的文件分为以下几个类别：

- Include 目录主要是用来导出 UUID，而 UUID 是任何在 RTL8762D 运行的应用所必须的。应用开发者不能修改此目录；
- Lib 目录包括应用程序使用的所有二进制文件；
- CMSIS 目录存放启动代码；
- APP 目录包括 PXP 应用的实现；
- Profile 目录包括应用使用的 BLE profiles 或者服务；
- Peripheral 目录包括应用使用的所有外设驱动和模块代码。

应用中的一些公共文件说明如下：

File name	Description
rom_uuid.h	UUID 头文件是 SDK 用来识别 ROM 的，无需更改
ROM.lib	ROM 符号库，是应用程序链接 ROM 中的符号的
gap_utils.lib	实现 BLE 功能的 Gap 库
startup_rtl876x.s	RTL8762D 应用程序启动的汇编文件
system_rtl876x.c	RTL8762D 应用程序启动的 C 文件

board.h	配置引脚和 DLPS 的头文件
flash_map.h	Flash layout 文件，此文件由 FlashMapGenerateTool 工具生成
mem_config.h	Memory 配置相关的文件

示例工程可能会随着 SDK 一起更新。为了更好地使用最新的示例代码，建议把新增的用户代码组织起来模块化。

每个示例工程的详细信息请参考对应示例工程的使用手册。

## 5.3 流程图

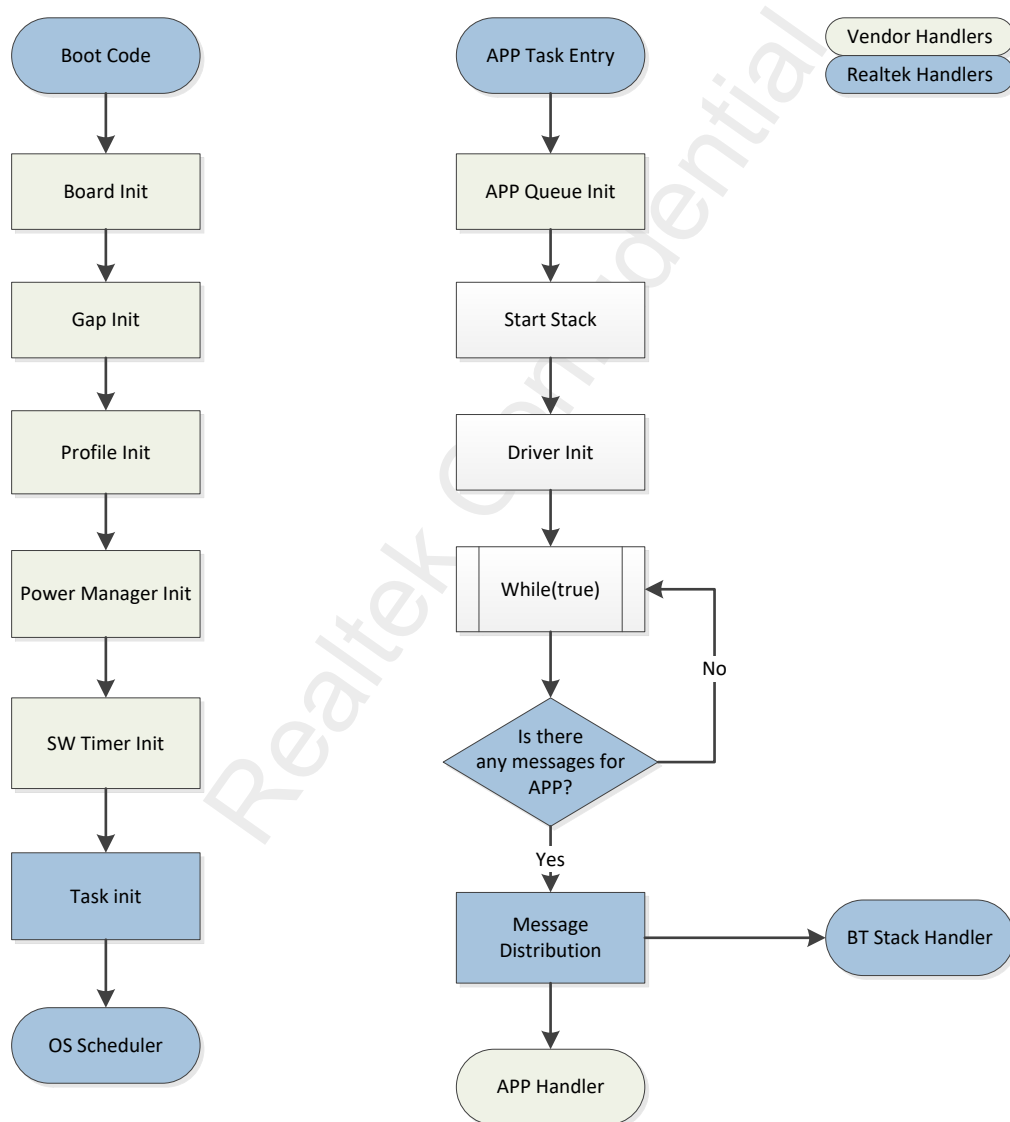


图 5-3 APP 流程

表 5-1 APP 初始化

Action	Description
board init	PINMUX 和 PAD 初始化设定
gap init	GAP 相关参数的初始化
profile init	BLE Profiles 的初始化
power manager init	电源管理相关的初始化
SW timer init	软件定时器的初始化
app queue init	APP Queue 的初始化
driver init	外设的初始化

系统是在 main() 函数中初始化，包括：Board、Peripherals、BT Stack、Profile、Power 和 Task 等。

BT Stack，Profiles 和外设驱动是在应用层任务中初始化，并且实现了 IO 消息机制。所有的功能都被封装成 IO 事件，这些事件是在相关的消息处理器中处理。

BT Stack 消息也是被封装成 BT IO 事件，跟外设事件处理的方式一样。

蓝色区块为用户要实现的部分

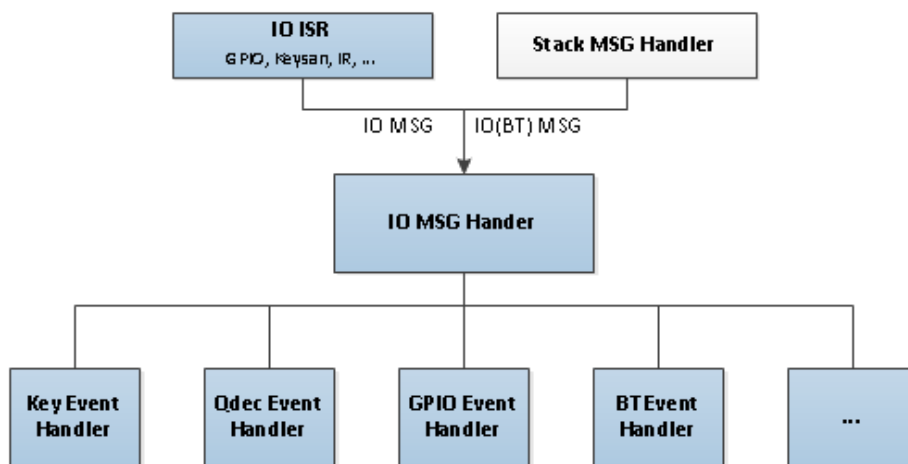


图 5-4 IO 消息处理流程

## 5.4 消息和事件处理流程

原始 MSG 的来源有通用 Peripherals 的 ISR 和 BT Stack，处理流程如下：

1. 来自通用 Peripherals 的 MSG 会被 MSG 分发器转发给 IOMSG Handler 处理；
2. 来自蓝牙协议栈的 MSG 会被 MSG 分发器转发给 BT 状态机，BT 状态机处理完 MSG 后，会封装成 BT IOMSG 发出，MSG 分发器收到 BT IO MSG 后再转发给 IO MSG Handler 处理；
3. IO MSG Handler 收到 MSG 后再判断 Event 类别，进而调用对应的 Event Handler 处理。

用户程序负责：

1. 实现 Peripheral ISR，在 ISR 中完成初步处理，如果需要进一步处理则封装 MSG 发给 APP；
2. 维护 IO MSG Handler，以便接收和处理用户定义的 ISR MSG；

### 3. 实现应用层相关的 Event Handler。

底层通知上层是通过 MSG 和 Event 机制，而上层可以直接调用底层的 APIs。

## 5.5 IO 消息

### 5.5.1 消息格式

```
typedef struct
{
    uint16_t type;
    uint16_t subtype;
    union
    {
        uint32_t param;
        void *buf;
    } u;
} T_IO_MSG;
```

### 5.5.2 消息类型定义

```
typedef enum
{
    IO_MSG_TYPE_BT_STATUS,
    IO_MSG_TYPE_KEYSCAN,
    IO_MSG_TYPE_QDECODE,
    IO_MSG_TYPE_UART,
    IO_MSG_TYPE_KEYPAD,
    IO_MSG_TYPE_IR,
    IO_MSG_TYPE_GDMA,
    IO_MSG_TYPE_ADC,
    ...
} T_IO_MSG_TYPE;
```

### 5.5.3 消息子类型定义

以 T\_IO\_MSG\_UART 为例，定义 UART 的子类型：



```
typedef enum
```

```
{
    IO_MSG_UART_RX                = 1,
    IO_MSG_UART_RX_TIMEOUT        = 2,
    IO_MSG_UART_RX_OVERFLOW       = 3,
    IO_MSG_UART_RX_TIMEOUT_OVERFLOW = 4,
    IO_MSG_UART_RX_EMPTY          = 5,
} T_IO_MSG_UART;
```

### 5.5.4 用户自定义消息

APP 开发者可以根据需求扩充定义 Message Type 以及自定义 Subtype Message。

## 5.6 Pin 设置

引脚的定义在 board.h 中定义：

```
#define KEY_0    P4_0
#define BEEP     P4_1
#define LED_0    P2_1
#define LED_1    P2_4
```

## 5.7 DLPS 设置

DLPS 的配置是在 board.h 中，如下所示：

1. DLPS 功能开关：

```
#define DLPS_EN    1
```

2. 用户自定义的 DLPS CB，在进入 DLPS 或者退出 DLPS 时调用这两个 CB。

```
#define USE_USER_DEFINE_DLPS_EXIT_CB    1
```

```
#define USE_USER_DEFINE_DLPS_ENTER_CB    1
```

3. 外设功能开关，需要使用的外设必须对应打开，这样在进入 DLPS 的时候才会保存外设的寄存器设定，退出 DLPS 的时候恢复外设的寄存器设定。没有使用的外设请关闭（0 表示关闭，1 表示打开）

```
#define USE_I2C0_DLPS    0
```

```
#define USE_I2C1_DLPS    0
```

```
#define USE_TIM_DLPS     0
```

```
#define USE_QDECODER_DLPS 0
```

```
...
```

Realtek Confidential

## 6 存储

### 6.1 存储映射

RTL8762D 内存包括 ROM, RAM, external SPI Flash 和 eFuse。Flash Cache 可以配置成 Cache 来使用,也可以配置成 Data RAM 来使用,如 Figure 6-1 所示。细节请参考文档 RTL8762D Memory User Guide。

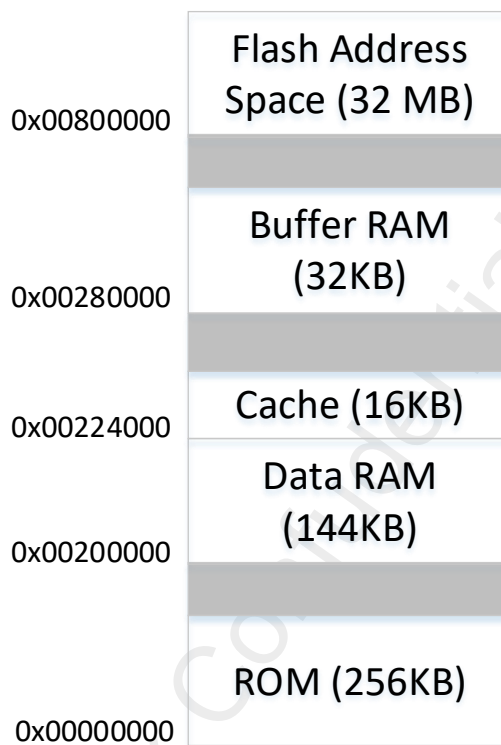


Figure 6-1 Memory Map

### 6.2 ROM

ROM 代码位于地址[0x0, 0x40000), 其中包括 Boot loader、RTOS、BT Stack、Flash Driver 以及别的平台模块。

### 6.3 RAM

RTL8762D 有两块 RAM: Data RAM 的位置在[0x00200000, 0x00224000), 而 Buffer RAM 的位置在[0x00280000, 0x00288000)。Cache 可以被配置成 Data RAM, 它的地址是在 [0x00224000, 0x00228000)。

#### 6.3.1 Data Ram

SDK 中 Data RAM 默认是分成了 6 部分, 如 Figure 6-2 所示。**Upper Stack RAM** 默认设定是 4KB, 不

同版本的 Size 是不同的，比如为了支持更高的性能，需要占用更多的 size；**APP RAM** 和 **Data RAM Heap** 是由 APP 决定的；这三者的总大小是 115K，不可更改，但是每个模块的大小是可以通过 mem\_config.h 调整的。

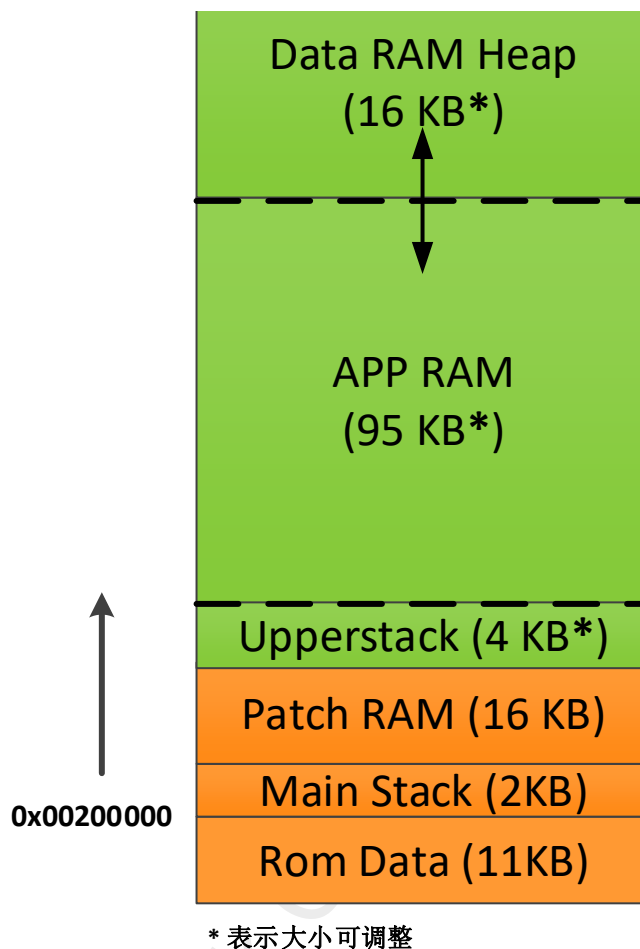


Figure 6-2 Data ram layout

### 6.3.2 Buffer Ram

SDK 中 Buffer RAM 默认是分成了 2 部分，如 Figure 6-3 所示。

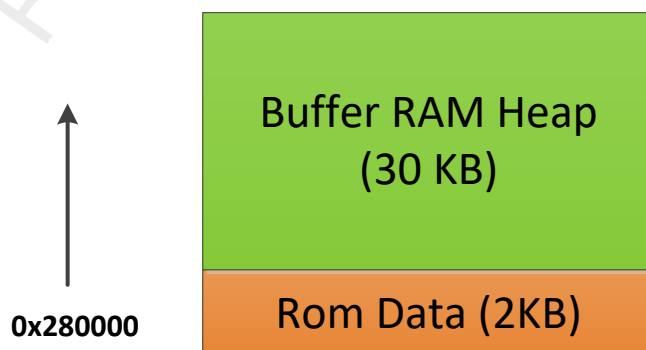


Figure 6-3 Buffer RAM Layout

## 6.4 Cache

RTL8762D 有一块 16K 字节的 cache，它是和 SPIC (SPI Flash Controller)一起工作来加快 SPI Flash 读操作。Cache 也可以当成 Data RAM 使用，如果 Cache 被配置成 Data RAM，它可以用作数据存储也可以用作代码执行，它的地址范围是[0x00224000, 0x00228000)，该地址范围是紧跟在 Data RAM 后面。

## 6.5 Flash

RTL8762D 支持最大 32M 字节大小的 external SPI Flash。在 SDK 中，external Flash 默认是分成 7 份，如图 6-4 所示：

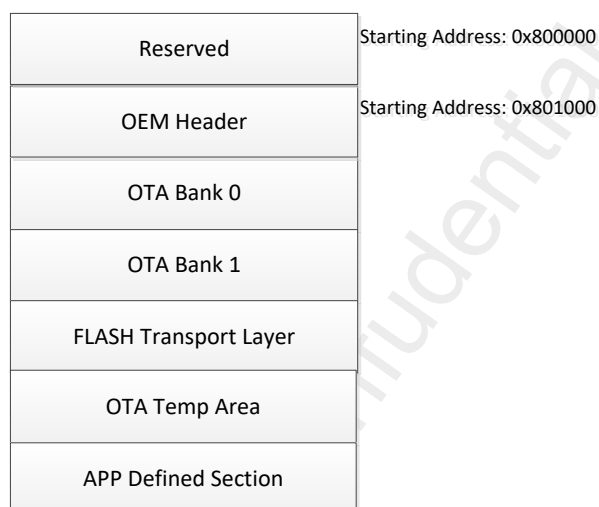


图 6-4 External Flash Layout

### 6.5.1 Flash APIs

Flash 操作的 APIs 如下所列，详细请查阅 SBee2-SDK.chm:

```
bool flash_read_locked(uint32_t start_addr, uint32_t data_len, uint8_t *data);

bool flash_auto_read_locked(uint32_t addr, uint32_t *data);

bool flash_split_read_locked(uint32_t start_addr, uint32_t data_len, uint8_t *data, uint32_t *counter);

bool flash_write_locked(uint32_t start_addr, uint32_t data_len, uint8_t *data);

bool flash_auto_write_locked(uint32_t start_addr, uint32_t data);

bool flash_auto_write_buffer_locked(uint32_t start_addr, uint32_t *data, uint32_t len);

bool flash_erase_locked(T_ERASE_TYPE type, uint32_t addr);
```

## 6.5.2 FTL

FTL (flash transport layer) 是 BT Stack 和用户程序读写 flash 数据的抽象层。FTL 的逻辑地址范围是[0, 0x17f0)。

### 1. BT 存储空间

- 地址范围: [0x0000, 0x0C00);
- 此范围是用于存储 BT 信息, 例如设备地址, link key 等;
- 详细请参考 RTL8762D BLE Stack User Manual。

### 2. APP 存储空间

- 地址范围: [0x0C00, 0x17f0);
- 此范围是用于存储 APP 的信息;
- 可以用下面的 APIs 在此范围内读/写数据, 详细请参考 SBee2-SDK.chm:

```
uint32_t ftl_save(void * p_data, uint16_t offset, uint16_t size)
uint32_t ftl_load(void * p_data, uint16_t offset, uint16_t size)
```

## 6.6 eFuse

eFuse 是一次编程的存储体, 用来存储重要且固定的信息, 例如: UUID, security key 以及其它的只会配置一次的信息。eFuse 中每个 bit 不能从 0 改成 1, 并且也没有擦除的操作, 所以更新 eFuse 要十分小心。Realtek 提供 MP Tool 更新 eFuse。

## 7 中断

### 7.1 Nested Vectored Interrupt Controller (NVIC)

NVIC 特性:

- 16 个 Cortex-M4 异常，32 条可屏蔽的中断通道
- 8 个可编程的中断优先级
- 支持向量表的重映射
- 低延迟异常和中断处理
- 系统控制寄存器（System Control Registers）的实现

NVIC 和处理器内核接口紧密耦合，使得低延迟中断处理和高效处理晚抵达中断成为可能。所有的中断包括内核异常都由 NVIC 管理。

### 7.2 中断向量表

表 7-1 中断向量表

Exception Number	NVIC Number	Exception Type	Description
1		Reset	Reset
2		NMI	Nonmaskable interrupt. The WDG is linked to the NMI vector
3		Hard Fault	All fault conditions if the corresponding fault handler is not enabled
4		MemManage	Memory management fault; Memory Protection Unit (MPU) violation or access to illegal locations
5		BusFault	Bus error;
6		Usage fault	Exceptions resulting from program error
7 ~ 10		RSVD	
11		SVC	Supervisor Call
12		Debug Monitor	Debug monitor (breakpoints, watch points, or external debug requests)
13		RSVD	
14		PendSV	Pendable Service Call
15		SYSTICK	System Tick Timer

16	[0]	System_ISR	System interrupt
17	[1]	WDG	Watch dog global interrupt
18	[2]	BTMAC_ISR	BT MAC interrupt
19	[3]	Timer3	Timer3 global interrupt
20	[4]	Timer2	Timer2 global interrupt
21	[5]	Platform	Platform interrupt
22	[6]	I2S_0 TX	I2S_0 TX interrupt
23	[7]	I2S_0 RX	I2S_0 RX interrupt
24*	[8]	Timer[4:7]	Timer4-7 interrupt(refer to 错误!未找到引用源。)
25	[9]	GPIO4	GPIO interrupt, P0_4
26	[10]	GPIO5	GPIO interrupt, P0_5
27	[11]	RTK_UART1	RTK_UART1 interrupt
28	[12]	RTK_UART0	RTK_UART0 interrupt
29	[13]	RTC	Real time counter interrupt
30	[14]	SPI_0	SPI_0 interrupt
31	[15]	SPI_1	SPI_1 interrupt
32	[16]	I2C_0	I2C_0 interrupt
33	[17]	I2C_1	I2C_1 interrupt
34	[18]	ADC	ADC global interrupt
35*	[19]	Peripheral_ISR	Peripheral interrupt(refer to 错误!未找到引用源。)
36	[20]	GDMA0_Channel0	RTK-DMA0 channel 0 global interrupt
37	[21]	GDMA0_Channel1	RTK-DMA0 channel 1 global interrupt
38	[22]	GDMA0_Channel2	RTK-DMA0 channel 2 global interrupt
39	[23]	GDMA0_Channel3	RTK-DMA0 channel 3 global interrupt
40	[24]	GDMA0_Channel4	RTK-DMA0 channel 4 global interrupt
41	[25]	GDMA0_Channel5	RTK-DMA0 channel 5 global interrupt
42	[26]	GPIO_Group3	GPIO group3 interrupt(refer to 错误!未找到引用源。)
43	[27]	GPIO_Group2	GPIO group2 interrupt(refer to 错误!未找到引用源。)
44	[28]	IR	IR module global interrupt
45	[29]	GPIO_Group1	GPIO group1 interrupt(refer to 错误!未找到引用源。)
46	[30]	GPIO_Group0	GPIO group0 interrupt(refer to 错误!未找到引用源。)
47	[31]	RTK_UART2	RTK_UART2 interrupt

表 7-2 Timer 4~7 ISR

Exception Number	NVIC Number	Exception Type	Description
------------------	-------------	----------------	-------------



48	[8]	Timer4	Timer4 interrupt
49	[8]	Timer5	Timer5 interrupt
50	[8]	Timer6	Timer6 interrupt
51	[8]	Timer7	Timer7 interrupt

**表 7-3 Peripheral ISR**

Exception Number	NVIC Number	Exception Type	Description
52	[19]	SPIC0_Flash	SPIC0_Flash interrupt
53	[19]	qdecode	qdecode global interrupt
54	[19]	keyscan	keyscan global interrupt
55	[19]	2-wire SPI	2-wire/3-wire SPI interrupt
56	[19]	analog comparator	analog comparator interrupt
57	[19]	MailBox	MailBox interrupt
58	[19]	I2S_1 TX	I2S_1 TX interrupt
59	[19]	I2S_1 RX	I2S_1 RX interrupt
60	[19]	LCD	LCD interrupt
61	[19]	TRNG	true random generator interrupt
62	[19]	SPIC1	SPIC1_Flash interrupt
63	[19]	SPIC2	SPIC2_Flash interrupt

**表 7-4 GPIO Group3 ISR**

NVIC Number	Exception Type	Description
[26]	GPIO3	GPIO3 interrupt
[26]	GPIO7	GPIO7 interrupt
[26]	GPIO11	GPIO11 interrupt
[26]	GPIO15	GPIO15 interrupt
[26]	GPIO19	GPIO19 interrupt
[26]	GPIO23	GPIO23 interrupt
[26]	GPIO27	GPIO27 interrupt
[26]	GPIO31	GPIO31 interrupt

**表 7-5 GPIO Group2 ISR**

NVIC Number	Exception Type	Description
[27]	GPIO2	GPIO2 interrupt

[27]	GPIO6	GPIO6 interrupt
[27]	GPIO10	GPIO10 interrupt
[27]	GPIO14	GPIO14 interrupt
[27]	GPIO18	GPIO18 interrupt
[27]	GPIO22	GPIO22 interrupt
[27]	GPIO26	GPIO26 interrupt
[27]	GPIO30	GPIO30 interrupt

表 7-6 GPIO Group1 ISR

NVIC Number	Exception Type	Description
[29]	GPIO1	GPIO1 interrupt
[29]	GPIO9	GPIO9 interrupt
[29]	GPIO13	GPIO13 interrupt
[29]	GPIO17	GPIO17 interrupt
[29]	GPIO21	GPIO21 interrupt
[29]	GPIO25	GPIO25 interrupt
[29]	GPIO29	GPIO29 interrupt

表 7-7 GPIO Group0 ISR

NVIC Number	Exception Type	Description
[30]	GPIO0	GPIO0 interrupt
[30]	GPIO8	GPIO8 interrupt
[30]	GPIO12	GPIO12 interrupt
[30]	GPIO16	GPIO16 interrupt
[30]	GPIO20	GPIO20 interrupt
[30]	GPIO24	GPIO24 interrupt
[30]	GPIO28	GPIO28 interrupt

## 7.3 中断优先级

RTL8762D 中断支持 3 个固定最高优先级和 8 个可编程优先级。

表 7-8 中断优先级

Priority	Usage
-3	Reset Handler

-2	NMI
-1	Hard Fault
0	用于实时性要求非常高的中断。
1	此优先级的中断不会被 FreeRTOS 的临界区屏蔽，但是中断处理函数内不能调用任何 FreeRTOS 的 APIs。
2	BT 相关的中断
3	通用的中断
4	
5	
6	
7	SysTick 和 PendSV

## 8 电量管理

RTL8762D 会在某些条件满足时进入 DLPS，在需要正常工作时唤醒。详细请参考文档 RTL8762D Deep Low Power State。

以下是要进入 DLPS 满足的条件：

1. Idle task 在执行，即其余的 task 都处于 block 状态或者 suspend 状态，且没有 ISR 执行；
2. 所有 OS 的消息已经被处理；
3. BT Stack, peripherals 和 application 等注册的 Check Callback 执行返回 true；
4. 软件定时器超时时间大于或等于 20ms；
5. BT 在如下的某个状态，并且参数满足要求：
  - 1) Standby State
  - 2) BT Advertising State, Advertising Interval\*0.625ms >= 20ms
  - 3) BT Scan State, (Scan Interval – Scan Window)\*0.625ms >= 15ms
  - 4) BT connection as Master role, Connection interval \* 1.25ms >= 12.5ms
  - 5) BT connection as Slave role, Connection interval\* (1+ Slave latency)\*1.25ms >= 12.5ms

RTL8762D 可以由以下事件从 DLPS 唤醒：

### 1. PAD 唤醒信号

### 2. 下列场景下产生的 BT 中断

- 1) 处于 BT advertising, advertising anchor 到来；
- 2) 处于 BT connection, connection interval anchor 到来；
- 3) 有 BT 事件发生，如远端连接请求或者收到数据。

### 3. RTC 中断

该功能需要调用以下 API 使能：

```
void RTC_EnableSystemWakeup(void)
```

#### 4. SW Timer 超时

##### Application 中 DLPS 模式的示例代码

1. 注册 **CHECK Callback 函数**：系统会调用 Callback 函数来决定是否能进 DLPS。

```
if (false == dlps_check_cb_reg(DLPS_PxpCheck))
{
    DBG_DIRECT("Error: dlps_check_cb_reg(DLPS_PxpCheck) failed!\n");
}
```

2. 实现 **CHECK Callback 函数**：如果返回 True，允许进入 DLPS；如果返回 False，不允许进入 DLPS。以 PXP 应用为例：

```
bool DLPS_PxpCheck(void)
{
    return allowedPxpEnterDlps;
}
```

3. 注册 **Callback 函数来保存和恢复外设寄存器**：

外设在进入 DLPS 时会掉电，所以在进 DLPS 时要保存外设寄存器状态，在退出 DLPS 时恢复外设寄存器状态。外设寄存器值应该在 DLPS\_IO\_EnterDlpsCb 函数中保存，在 DLPS\_IO\_ExitDlpsCb 函数中恢复。

```
dlps_hw_control_cb_reg(DLPS_IO_EnterDlpsCb, DLPS_ENTER);
dlps_hw_control_cb_reg(DLPS_IO_ExitDlpsCb, DLPS_EXIT4_BT_READY);
```

4. 使能 **DLPS 模式**：

```
lps_mode_set(LPM_DLPS_MODE);
```

## 9 烧录

### 9.1 Images 相关

RTL8762D BLE 应用需要 5 个 images: Patch Image、App Image、System Configuration File、OTA Header File 和 Secure Boot Loader Image。

1. Patch Image: Realtek 发布, ROM code 中保留了 Patch 函数入口, 通过 Patch 可以修改 ROM code 原有行为、扩充 ROM code 功能等;
2. Configuration file 记录 IC 硬件配置和蓝牙配置等信息, 例如: 配置 BT 地址, 修改 Link 数目等。可通过 MP Tool 配置生成;
3. App Image: 开发者开发的 BLE 应用程序。由 Keil 工程编译生成, 并经过 fromelf 等工具处理;
4. OTA Header File: 定义 Flash Bank layout 的文件, 由 MPPackTool 生成;
5. Secure Boot Loader Image: Realtek 发布, 优化 Boot Flow。如果 OTA 方式是不切 bank 的, 那么一定要烧录 Secure Boot Loader Image。

### 9.2 Image 处理工具

Keil 编译直接生成的 application image 是 ELF 格式的, 为了便于烧录以及支持 OTA 功能, 需要将 ELF 格式的 application image 转换为二进制格式的 bin 文件, 并作相应处理。

Demo 工程中使用了 3 个 Command 来处理 application image, 分别是 fromelf, prepend\_header 以及 md5。

#### 9.2.1 fromelf

Fromelf 来自 Keil (默认安装路径: C:\Keil\_v5\ARM\ARMCC\bin\), 具体用法可参考 Keil 帮助文档中的《Using the fromelf Image Converter》一节。

使用该工具产生烧录的 bin 文件以及反汇编文件。

#### 9.2.2 prepend\_header

1. 用法: prepend\_header.exe [options] ...
2. 作用: 为支持 OTA 功能, 需要在 bin 文件添加校验以及长度信息。

对于 Application Image, 其最开始 1024 字节是 image Header, 详细请查看 system\_rtl876x.c 中定义的 T\_IMG\_HEADER\_FORMAT 结构。

Application Image 的 Checksum 可以是 crc16 或者 sha256, 分别记录在 T\_IMG\_HEADER\_FORMAT 结构体中的 crc16 和 sha256 成员, payload 长度记录在 payload\_len 成员。

```
typedef struct _IMG_CTRL_HEADER_FORMAT
```

```
{
    uint8_t ic_type;
    uint8_t secure_version;
    union
    {
        uint16_t value;
        struct
        {
            uint16_t xip: 1; // payload is executed on flash
            uint16_t enc: 1; // all the payload is encrypted
            uint16_t load_when_boot: 1; // load image when boot
            uint16_t enc_load: 1; // encrypt load part or not
            uint16_t enc_key_select: 3; // referenced to ENC_KEY_SELECT
            uint16_t not_ready : 1; //for copy image in ota
            uint16_t not_obsolete : 1; //for copy image in ota
            uint16_t integrity_check_en_in_boot : 1; // enable image integrity check in boot flow
            uint16_t rsvd: 6;
        };
    } ctrl_flag;
    uint16_t image_id;
    uint16_t crc16;
    uint32_t payload_len;
} T_IMG_CTRL_HEADER_FORMAT;
```

```
typedef struct _IMG_HEADER_FORMAT
```

```
{
    T_IMG_CTRL_HEADER_FORMAT ctrl_header;
    uint8_t uuid[16];
    uint32_t exe_base;
    uint32_t load_base;
    uint32_t load_len;
    uint8_t rsvd0[8];
    uint32_t magic_pattern;
    uint8_t dec_key[16];
}
```

```
uint8_t rsvd1[28];

T_VERSION_FORMAT git_ver;

RSA_PUBLIC_KEY rsaPubKey;

uint8_t sha256[32];

uint8_t rsvd2[76];

} T_IMG_HEADER_FORMAT;
```

## 9.2.3 md5

1. 用法: md5.exe [Input Binary Filename];

2. 作用: 计算 bin 文件的 MD5, 并且加到 bin 文件中, 用于 MP Tool 烧录时验证。后缀形式为: [Original File Name]-[MD5].bin;

MP Tool 采用 MD5 作为烧录文件的校验值, 并要求将 MD5 值加入到待烧录文件的文件名后缀中。比如原始文件名为 App\_ver1.1.0 .bin, 经 md5.exe 处理后, 带后缀的文件名为: App\_ver1.1.0-d77dd83cb2848d3e9ac04c7dd9367e69.bin。

## 9.3 烧录方式

Patch Image, Config File 和 App Image 分别支持的下载的方式如表格 9-1 所示:

表 9-1 Images 可用烧录方式

	Patch Image	Config File	APP Image
Keil	×	×	√
SWDTool	√	√	√
MPTool	√	√	√

## 10 调试

有两种方式可以调试应用程序:

1. 使用 log 机制跟踪代码的执行和数据;
2. 使用 Keil MDK 和 SWD 进行 running control, 增加/删除 breakpoints 以及访问/追踪 memory 等。

### 10.1 Log 机制

Debug analyzer 是用来通过打印 log 的方式来追踪应用程序的执行情况, 以及抓取 BT 的数据并进一步通过第三方 BT 分析工具来分析。参考 Debug analyzer user manual 来获取详细的信息。

P0\_3 是默认作为 Log UART 的输出引脚（也可以将 Log 功能重新配置到其它引脚），经 UART-to-USB 连接到 PC，在 PC 端 COM 口接收 Log 数据。

## 10.1.1 Debug analyzer

使用 Debug analyzer 的步骤：

1. 打开 DebugAnalyzer.exe，点击“Settings”，之后点击“Start”即可。

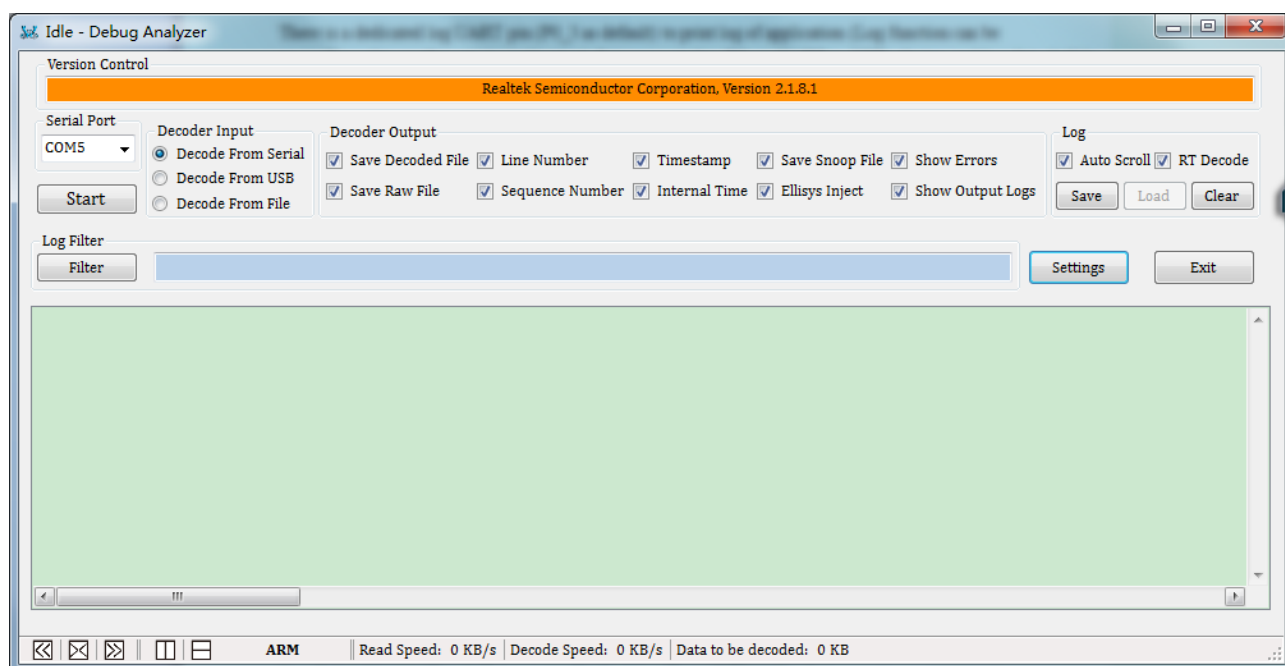


图 10-1 Debug Analyzer 主页面

2. “Settings”界面中，“Serial Port”要选择 LOG 对应的 COM 口；
3. “App Trace File”要选择对应的 App.trace。 App.trace 是在 APP 程序编译后和 App.bin 生成在同一个目录位置。



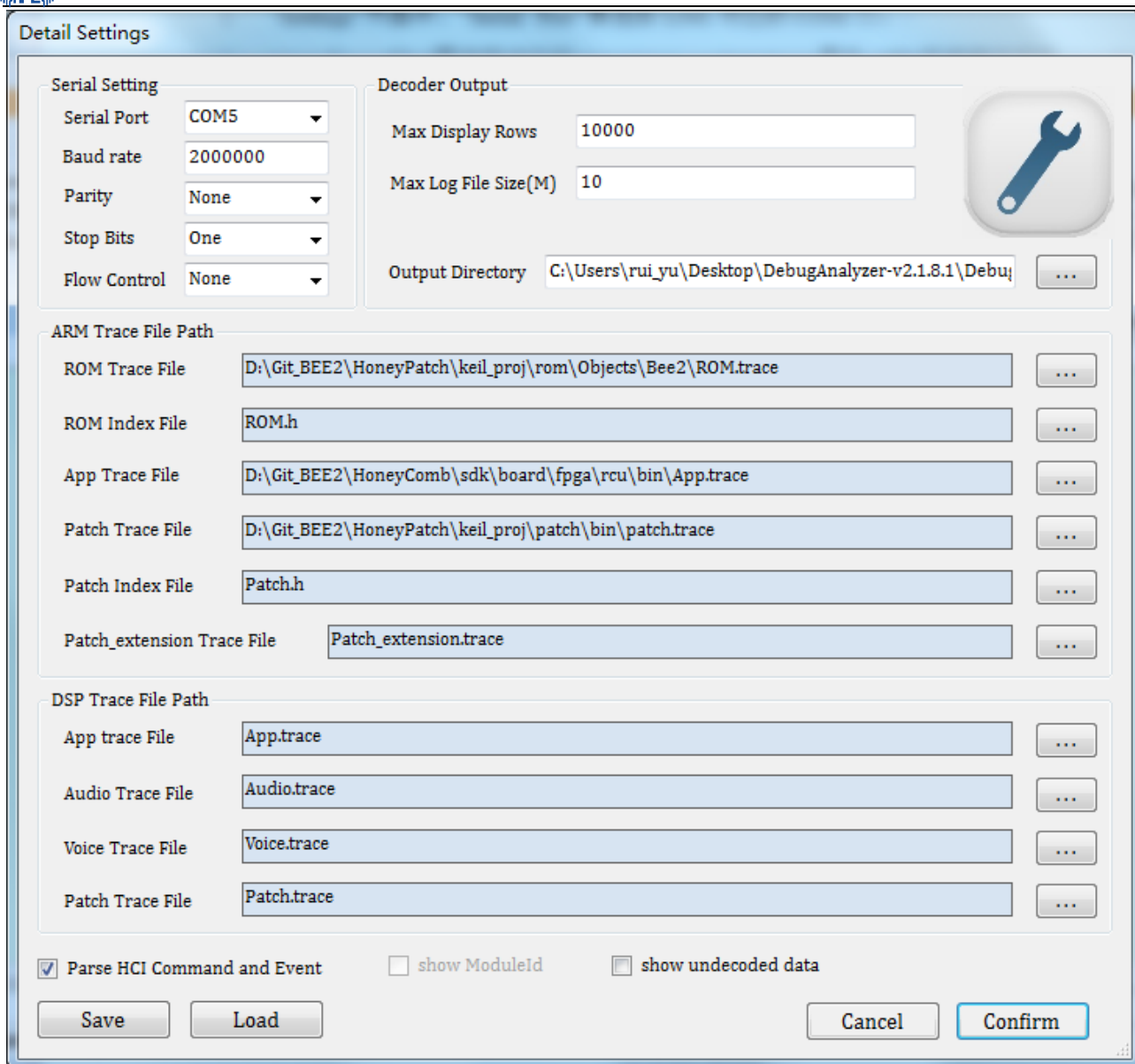


图 10-2 Debug Analyzer 设置页面

Debug analyzer 提供的一些高级功能：

1. **“Save Raw File”** (建议勾选)：保存 Log 到 xxx.bin 文件，此文件可以提供给 Realtek 做进一步的分析；
2. **“Log Filter”**：过滤显示。弹出新窗口，只显示满足过滤条件的 Log；
3. **“keyword search”**：在 Log 打印窗口高亮显示关键字；
4. **“Clear Trace”**：清空显示框中的 Log。

Debug analyzer 会自动保存 Log 文件，并按照端口号和生成时间命名，如 COM 5\_2015-06-12\_18-05-00.log。

## 10.1.2 Log 打印基本接口

硬件有实现特定的 GDMA 通道来打印 Log，log 接口声明：

```
DBG_BUFFER(T_LOG_TYPE type, T_LOG_SUBTYPE sub_type, T_MODULE_ID module, uint8_t level,
char* fmt, uint8_t param_num,...).
```

参数 type 固定是 TYPE\_BEE2，sub\_type 固定是 SUBTYPE\_FORMAT。只要关心参数 module 和 level。

T\_MODULE\_ID 类型中有预定义好几种 module，这些 module 用于将 log 分类别。Debug Analyzer 能够识别这些 module，在每条打印的 log 前自动加到 module 名称。

还有一个概念是 Debug Level，表示 log 要打印的级别。定义了 4 种类别：

表 10-1 调试等级

Debug Level	Usage Scenario
LEVEL_ERROR	Fatal, Procedure Cannot Advance (Log Token !!!)
LEVE_WARN	Abnormal Condition Occurred, But Procedure Can Advance (Log Token !!*)
LEVEL_INFO	Important Notification (Log Token !**)
LEVEL_TRACE	Verbose Debug

DBG\_BUFFER() 接口虽然灵活、功能丰富，但是也会有一点不容易使用。RTL8762D SDK 有包装此接口，在 trace.h 中提供了一些易读性更强的 APIs。

## 10.1.3 Log 打印封装接口

有提供一些封装的 APIs 来打印 Log，这些 APIs 有一个通用的语法：

```
{MODULE}_PRINT_{LEVEL}_{PARAMNUM}(...)
```

- {MODULE} 可以用定义在 trace.h 中的 module 名称代替，例如 APP/GAP/USB/FLASH...
- {LEVEL} 可以用这四种 level 代替：ERROR/WARN/INFO/TRACE。
- {PARAMNUM} 可以用数字 0 到 8 代替，意味着这条 log 会打印参数的个数。

例如，如果想要打印一条含 2 个参数的 log，可以像这样写：

```
APP_PRINT_WARN2("Test app: ID = %d, data = 0x%x", id, data);
```

Debug Analyzer 会如下显示这条 Log：

```
00494 10-13#17:06:45.994 087 02145 [APP] !!*Test app: ID = 3, data = 0xF0
```

说明：

1. 不要超过 8 个参数 (如果使用 DBG\_BUFFER()接口，最多允许 20 个参数)；
2. 单条 log 中不要超过 128 个字符串；
3. 如果可能，尽量把所有的参数放到一条打印语句中。

## 10.1.4 辅助打印接口

DBG\_BUFFER()接口只能打印诸如 (d, i, u, o, x) 的整型、字符和指针类型，但是有时候可能需要打印 string, binary array 或者 BT 地址，那么需要用到几个辅助的打印接口：

### 1. TRACE\_STRING(char\* data)

直接打印字符串，转换符%s。

### 2. TRACE\_BINARY(uint16\_t length, uint8\_t\* data)

以 16 进制格式打印二进制字符串，转换符%b。

### 3. TRACE\_BDADDR(char\* bd\_addr)

以 BT 地址格式打印二进制字符串，例如：

Hex Array: 0xaa 0xbb 0xcc 0xdd 0xee 0xff -> Literal String: FF::EE::DD::CC::BB::AA

说明：单条 Log 中最多能有 4 个地址，转换符%s。

## 10.1.5 Log 打印示例

下面的例子展示了 log APIs 的一般用法，以及在 Debug Analyzer 上对应的输出：

Log 打印代码：

```
uint32_t n = 77777;
uint8_t m = 0x5A;
uint8_t bd1[6] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55};
uint8_t bd2[6] = {0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};
char c1[10] = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '0'};
char c2[8] = {'a', 'b', 'C', 'd', 'E', 'F', 'g', 'H'};
char *s1 = "Hello world!";
char *s2 = "Log Test";
ADC_PRINT_TRACE1("ADC value is %d", n);
UART_PRINT_INFO3("Serial data: 0x%x, c1[%b], s1[%s]", m, TRACE_BINARY(10, c1),
TRACE_STRING(s1));
GAP_PRINT_WARN6("n[%d], m[%c] bd1[%s], bd2[%s], c2[%b], s2[%s]", n, m, TRACE_BDADDR(bd1),
TRACE_BDADDR(bd2), TRACE_BINARY(8, c2), TRACE_STRING(s2));
APP_PRINT_ERROR0("APP ERROR OCCURED...");
```

Debug Analyzer 上对应的输出结果：

```
00252 10-25#17:12:02.021 132 10241 [ADC] ADC value is 77777
00253 10-25#17:12:02.021 133 10241 [UART] ***Serial data: 0x5a, c1[31-32-33-34-35-36-37-38-39-30],
s1[Hello world!]
00254 10-25#17:12:02.022 134 10241 [GAP] !!*n[77777], m[Z] bd1[55::44::33::22::11::00],
bd2[FF::EE::DD::CC::BB::AA], c2[61-62-43-64-45-46-67-48], s2[Log Test]
00255 10-25#17:12:02.022 135 10241 [APP] !!!APP ERROR OCCURED...
```

## 10.1.6 Log 控制接口

有时候可能需要控制某些 log 的打开和关闭。一种方式是通过设定 MP Tool 中的 Config File, 再将 Config File 烧录到 RTL8762D, 然而如果想要频繁改变 log 打印 level, 那么这种方式可能就不够灵活了, 因为要每次重烧 Config File 也不方便。所以 SDK 中有提供 3 个 APIs 来控制特定 module 的特定 level。

1. log\_module\_trace\_init()
2. log\_module\_trace\_set()
3. log\_module\_bitmap\_trace\_set()

查阅 SDK API 文档获取详细的参数信息。

这里通过一些场景来帮助你理解如何在应用程序中使用 log 控制的 APIs。假设 log 打印功能已经打开, 并且 Config File 中将所有的 log trace mask 置成 1, 意味着所有 module 的所有 level 的 log 都会打印出来。

**场景 1:** 关闭 APP module 的所有 trace 和 info 级别的 log。

```
int main(void)
{
    log_module_trace_set(MODULE_APP, LEVEL_INFO, false);
    log_module_trace_set(MODULE_APP, LEVEL_TRACE, false);
    ...
}
```

**场景 2:** 只打开 PROFILE module 的 log。

```
int main(void)
{
    uint64_t mask[LEVEL_NUM];
    memset(mask, 0, sizeof(mask));
    log_module_trace_init(mask);

    log_module_trace_set(MODULE_PROFILE, LEVEL_ERROR, true);
    log_module_trace_set(MODULE_PROFILE, LEVEL_WARN, true);
    log_module_trace_set(MODULE_PROFILE, LEVEL_INFO, true);
    log_module_trace_set(MODULE_PROFILE, LEVEL_TRACE, true);
    ...
}
```

**场景 3:** 关闭 PROFILE/PROTOCOL/GAP/APP module 的 trace 级别 log。

```
int main(void)
{
    log_module_bitmap_trace_set(MODULE_BIT_PROFILE | MODULE_BIT_PROTOCOL |
                                MODULE_BIT_GAP | MODULE_BIT_APP, LEVEL_TRACE, false);
    ...
}
```

**场景 4:** 关闭除了 APP module 以外的所有 trace 和 info 级别的 log, 包括关闭 BT Snoop log。

```
int main(void)
```

```
{
    for (uint8_t i = 0; i < MODULE_NUM; i++)
    {
        log_module_trace_set((T_MODULE_ID)i, LEVEL_TRACE, false);
        log_module_trace_set((T_MODULE_ID)i, LEVEL_INFO, false);
    }
    log_module_trace_set(MODULE_APP, LEVEL_INFO, true);
    log_module_trace_set(MODULE_APP, LEVEL_TRACE, true);
    log_module_trace_set(MODULE_SNOOP, LEVEL_ERROR, false);
    ...
}
```

说明：如果 MODULE\_SNOOP 的 LEVEL\_ERROR log 打开，那么 Debug Analyzer 能产生 BT Snoop 的 log 文件 (\*.cfa)。换句话说，如果关闭 MODULE\_SNOOP 的 log，那么就不会产生 BT Snoop log，如上面场景 2 和场景 4。

### 10.1.7 DBG\_DIRECT

通过调用基于 DBG\_BUFFER 接口的函数打印 log 对系统性能的影响比较小，但是打印实时性不高，因为通过这些 APIs 打印的 log 会被缓存到 buffer 中，在系统空闲的时候发送给 Log Uart。

DBG\_DIRECT 用来打印实时 log。这个 API 打印 log 对系统的影响比较大，因为会直接发送数据给 Log Uart，在 Log 打印完之前系统不能继续执行，所以强烈建议只在以下一些特定的情况下使用该 API：

1. 在系统重启前
2. 实时性要求高的场景
3. 在 DLPS exit callback 中

打印 Log 的示例代码：

```
DBG_DIRECT("ROM version: %s %s", __DATE__, __TIME__);
```

Debug Analyzer 上对应的输出结果：

```
00002 12-22#19:19:32.573 004 00000 ROM version: Dec 22 2017 14:54:04
```

## 10.2 SWD 调试

如图 10-1 所示，RTL8762D 实现了以下 3 个基本的调试接口：run control, breakpoint 和 memory access，还有一个 trace 口：DWT。RTL8762D 不支持 JTAG，支持 SWD。

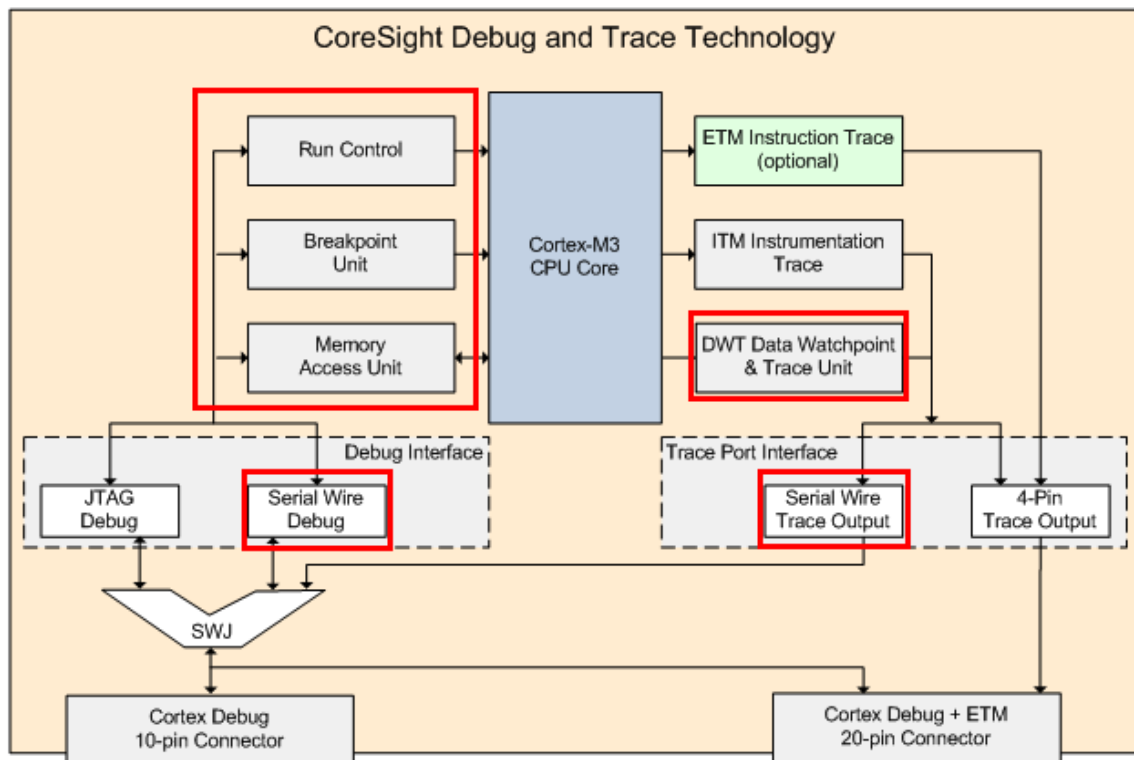


图 10-1 RTL8762D 调试接口

### 10.2.1 SWD 调试接口

在使用 SWD 之前，应该确保已经安装和配置好 Keil MDK-ARM 和 SWD debugger。

RTL8762D 支持的主要的调试特性如 UI 上展示的：

- Running control 按钮：Running/Reset/Step
- Breakpoints 按钮：Add/Delete/Conditional
- 一些调试的功能窗口：
  1. Core registers 窗口：监视/修改 MCU 寄存器值；
  2. Disassembly 窗口：查看反汇编代码，支持断点和 mixed 模式（显示 C 代码）；
  3. Source code 窗口：查看 C 代码，支持断点以及变量的实时显示；
  4. Variable watch window：查看加入监视的变量值；
  5. Memory window：显示/修改待查的内存，支持直接的地址输入和可变的地址输入；
  6. Call stack and local variable window：显示当前的调用栈，以及 local 变量（即 stack 上的变量）。

有时候可能会遇到 image 无法成功下载到 flash 上，或者即使有正确和 J-link 连接，但是依旧找不到 SWD，这时检查下如下的配置可能会有帮助：

- 更改 debug clock 到一个较小的值（例如，从 2MHz 降到 1 MHz），或者换用更短的 SWD 调试线；
- IC 可能已经进入了 DLPS 模式，reset IC，这样系统会保持处于 active 状态 5 秒钟。

## 10.2.2DWT 监视接口

RTL8762D 提供一种使用 DWT 追踪 memory 的方式。一组共 4 个 API 用来监视最多 4 块 memory 区域：

```
watch_point_0_setting()
watch_point_1_setting()
watch_point_2_setting()
watch_point_3_setting()
```

更多细节请查阅 debug\_monitor.h。

下面的示例展示了如何使用 debug monitor API:

```
void debug_monitor_enable(void)
{
    //set debug monitor priority
    NVIC_SetPriority(DebugMonitor_IRQn, 3);

    //enable exception and monitor control register
    CoreDebug->DEMCR |= CoreDebug_DEMCR_MON_EN_Msk | CoreDebug_DEMCR_TRCENA_Msk;

    //comment: set appropriate parameter and enable watch point to specific address if desired
    //set DWT compare registers (max 4 comparators)
    watch_point_0_setting(0x1000180C, DWT_DATAVSIZ_WORD, DWT_FUNCTION_WRITE);
    watch_point_1_setting(0x10000004, DWT_DATAVSIZ_WORD, DWT_FUNCTION_READ_OR_WRITE);
    watch_point_2_setting(0x10000008, DWT_DATAVSIZ_WORD, DWT_FUNCTION_READ_OR_WRITE);
    watch_point_3_setting(0x1000000C, DWT_DATAVSIZ_WORD, DWT_FUNCTION_READ_OR_WRITE);

    //enable DWT control register
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
    return;
}
```

如果被监视的内存被访问了，那么会通过 Log Uart 打印出来，就能通过 Debug Analyzer 来分析了。