User Manual

AMetal-AMF07X-Core 用户手册

AMetal

V1.0.0 Date:2020/08/17

产品用户手册

类别	内容
关键词	AMF07X-Core、功能介绍
摘　要	本文档简述了 AMF07X-Core 硬件资源，详细介绍了 ametal-amf07x-core 软件包的结构、配置方法等。

修订历史

版本	日期	原因
发布 1.0.0	2020/8/17	创建文档

目 录

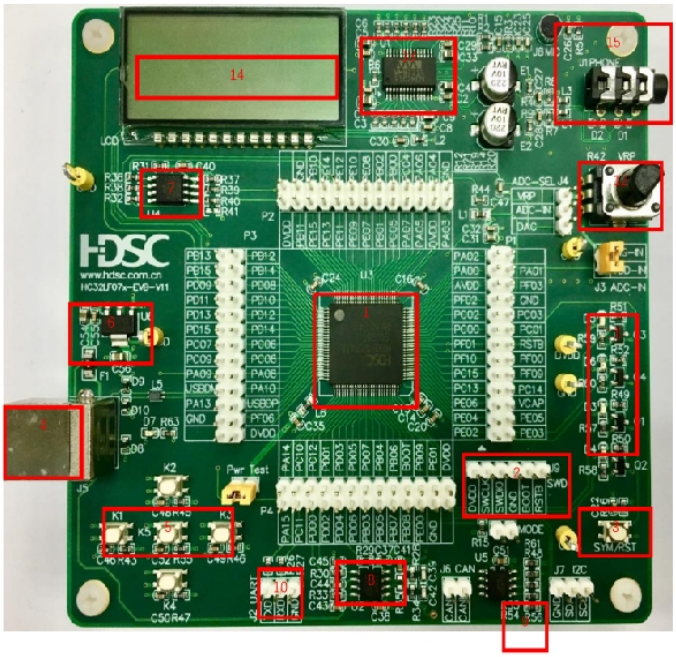
1. 开发平台简介 …… 1
1.1 HC32F07X …… 1
1.2 AMF07X-Core …… 2
2. AMetal 软件包 …… 2
2.1 AMetal 架构 …… 2
2.1.1 硬件层 …… 3
2.1.2 驱动层 …… 3
2.1.3 标准接口层 …… 3
2.2 目录结构 …… 3
2.3 工程结构 …… 3
2.3.1 Keil 工程结构 …… 3
2.3.2 Eclipse 工程结构 …… 4
3. 工程配置 …… 5
3.1 部分外设初始化使能/禁能 …… 5
3.2 板级资源初始化使能/禁能 …… 6
4. 外设资源及典型配置 …… 7
4.1 配置文件结构 …… 8
4.1.1 设备实例 …… 8
4.1.2 设备信息 …… 9
4.1.3 实例初始化函数 …… 16
4.1.4 实例解初始化函数 …… 17
4.2 典型配置 …… 18
4.2.1 ADC …… 18
4.2.2 AES …… 21
4.2.3 CLK …… 21
4.2.4 CRC …… 22
4.2.5 DAC …… 22
4.2.6 DMA …… 23
4.2.7 GPIO …… 23
4.2.8 I²C …… 23
4.2.9 I²C 从机 …… 24
4.2.10 OPA …… 25
4.2.11 RTC …… 25
4.2.12 LVD …… 26
4.2.13 SPI …… 27

4.2.14 Timer ····· 29
4.2.15 TRNG ····· 30
4.2.16 USART ····· 30
4.2.17 WDT ····· 32
4.2.18 CAN ····· 32
4.2.19 USB ····· 33
4.3 使用方法 ····· 34
4.3.1 使用 AMetal 软件包提供的驱动 ····· 34
4.3.1.1 初始化 ····· 34
4.3.1.2 操作外设 ····· 35
4.3.1.3 解初始化 ····· 39
4.3.2 直接使用硬件层函数 ····· 41
5. 板级资源 ····· 44
5.1 配置文件结构 ····· 44
5.2 典型配置 ····· 45
5.2.1 LED 配置 ····· 45
5.2.2 按键 ····· 46
5.2.3 调试串口配置 ····· 46
5.2.4 系统滴答和软件定时器配置 ····· 47
5.3 使用方法 ····· 47
6. 免责声明 ····· 48

1. 开发平台简介

AMF07X-Core 开发平台主要用于 HC32F07X 系列微控制器的学习和开发，配套 AMetal 软件包，提供了各个外设的驱动程序、丰富的例程和详尽的资料，是工程师进行项目开发的首选。该平台也可用于教学、毕业设计及电子竞赛等，开发平台如 图 1.1 所示。

注意： 当前提供的 SDK 支持的开发平台版本为 **HDSC HC32F07X-EVB-V11** ，版本号可以在开发平台的正面找到。用户在使用 SDK 前请确认 SDK 支持自己手中的开发平台。



1	HC32F072
2	SWD/ISP 调试编程端口
3	复位按键（RST）
4	USB-FS 接口
5	用户按键（K1~K5）
6	LDO
7	SPI FLASH
8	CAN PHY
9	I2C EEPROM/IF
10	UART IF
11	LED（4）
12	AD/DA IF
13	CODEC
14	液晶显示（LCD）
15	Audio IF

图 1.1 AMF07X-Core 开发平台

AMF07X-Core 开发板基于 HDSC 的 HC32F07X 微控制器，功能强大，支持外设丰富，除了常见外设，还支持 Can，USB 等。开发板已经将单片机的大部分 I/O 资源引出，帮助用户快速测试。

1.1 HC32F07X

- 工作电压 1.8V ~ 5.5V；
- 低功耗；
- ARM Cortex-M0+ 32 位内核，主频可达 48MHz；
- 128KB Flash，16KB SRAM；
- 1 个 12 位 ADC，1μS 转换时间；
- 多达 7 个定时器；
- 4 个 USART 接口、2 个 I^2C 接口、2 个 SPI 接口和 1 个 CAN 接口；
- 2 个 LPUART 接口、2 个 ^{2} 接口、2 个 DAC、1 个 USB 接口

- 2 通道 DMA 控制器；
- 多达 86 个快速 I/O 端口；
- 采用 LQFP100 封装。

1.2 AMF07X-Core

- 5V USB-Device 供电；
- 4 个 LED 发光二极管；
- 1 个 LCD；
- 5 个独立按键；
- 1 个复位按键；

基于这些资源，可以完成多种基础实验。

2. AMetal 软件包

AMetal 由广州致远电子有限公司开发，是完全开源的，相关资源共享在 github 平台上。（链接：https://github.com/zlgopen/ametal）。目前 AMetal 软件包内包含众多的开发板驱动，AMF07X-Core 也是其中之一。

2.1 AMetal 架构

如 图 2.1 所示，AMetal 共分为 3 层，硬件层、驱动层和标准接口层。

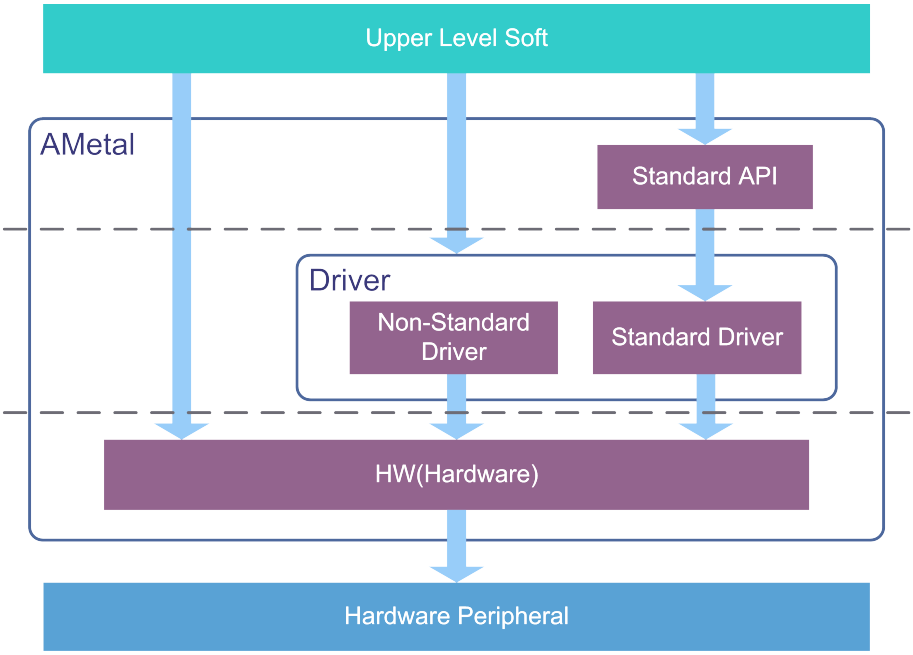


图 2.1 AMetal 框架

根据实际需求，这三层对应的接口均可被应用程序使用。对于 AWorks 平台或者其他操作系统，它们可以使用 AMetal 的标准接口层接口开发相关外设的驱动。这样，AWorks 或者其它操作系统在以后的使用过程中，针对提供相同标准服务的不同外设，不需要再额外开发相对应的驱动。

2.1.1 硬件层

硬件层对 SOC 做最原始封装，其提供的 API 基本上是直接操作寄存器的内联函数，效率最高。当需要操作外设的特殊功能，或者对效率、特殊使用等有需求时，可以调用硬件层 API。硬件层等价于传统 SOC 原厂的裸机包。硬件层接口使用 **amhw_/AMHW_ + 芯片名**作为命名空间，如 **amhw_hc32f07x** 、**AMHW_HC32F07X** 。

参见:

更多的硬件层接口定义及示例请参考 **ametal\documents\《AMetal API 参考手册 V1.10.chm》** 或者 **\ametal\soc\hdsc\drivers** 文件夹中的相关文件。

注解: 本文使用 SOC(System On Chip) 泛指将 CPU 和外设封装在一起的 MCU、DSP 等微型计算机系统。

2.1.2 驱动层

虽然硬件层对外设做了封装，但其通常与外设寄存器的联系比较紧密，用起来比较繁琐。为了方便使用，驱动层在硬件层的基础上做了进一步封装，进一步简化对外设的操作。

根据是否实现了标准层接口可以划分为标准驱动和非标准驱动，前者实现了标准层的接口，例如 GPIO、UART、SPI 等常见的外设；后者因为某些外设的特殊性，并未实现标准层接口，需要自定义接口，例如 DMA 等。驱动层接口使用 **am_/AM_ + 芯片名**作为命名空间，如 **am_hc32f07x** 、**AM_HC32F07X** 。

参见:

更多的驱动层接口定义及示例请参考 **ametal\documents\《AMetal API 参考手册 V1.10.chm》** 或者 **\ametal\soc\hdsc\drivers** 文件夹中的相关文件。

2.1.3 标准接口层

标准接口层对常见外设的操作进行了抽象，提取出了一套标准 API 接口，可以保证在不同的硬件上，标准 API 的行为都是一样的。标准层接口使用 **am_/AM_** 作为命名空间。

参见:

更多的标准接口定义及示例请参考 **ametal\documents\《AMetal API 参考手册 V1.10.chm》** 或者 **\ametal\interface** 文件夹中的相关文件。

2.2 目录结构

关于软件包内的目录结构信息可参考软件包中 **documents** 文件夹下的**《AMetal 目录结构.PDF》** 文档。

2.3 工程结构

2.3.1 Keil 工程结构

打开 Keil 版本的 template_amf07x_core 模板工程，其工程结构如 图 2.2 所示。

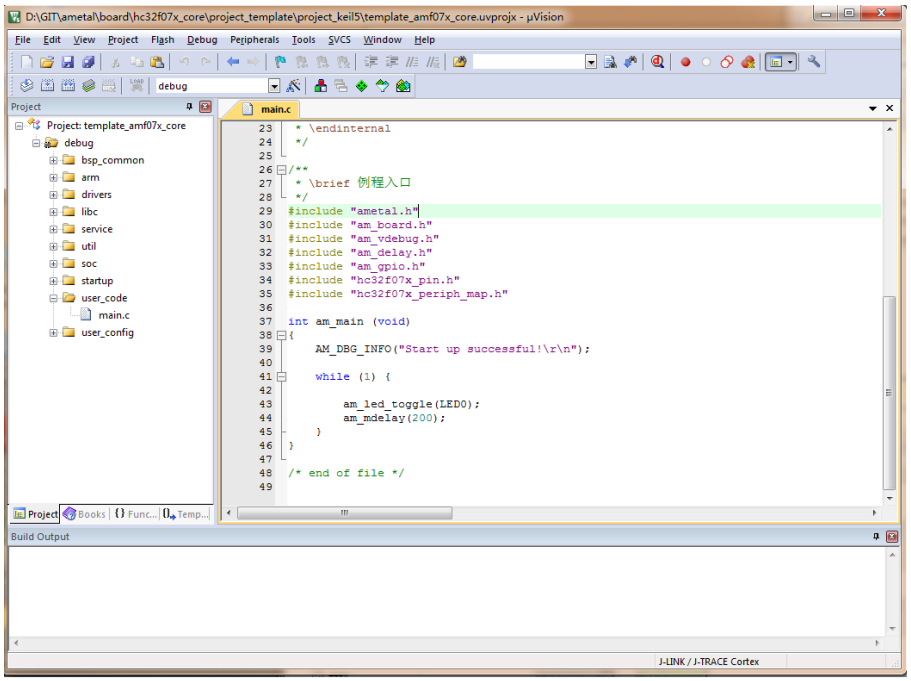


图 2.2 Keil 版本工程模板结构

在工程结构中，包含了 bsp_common、arm、drivers、libc、service、util、soc、startup、user_code、user_config 共 10 个节点。

- **bsp_common** 存放板级通用文件，如系统堆的适配文件，C 库的板级适配文件等。
- **arm** 文件夹存放了与内核相关的通用文件，如 NVIC、Systick 等。
- **drivers** 文件夹包含通用外设及模块的标准驱动文件，如按键、矩阵键盘、温度传感器模块、ZigBee 模块等。
- **libc** 文件夹下存放的是 C 库适配文件，如 armlib_adpater，主要用于适配相应的 C 库。
- **service** 文件夹主要存放通用的服务组件的驱动头文件，如 ADC、蜂鸣器等，。
- **util** 文件夹主要包含 AMetal 通用的辅助工具文件，如堆管理、软件定时器以及打印输出函数等。
- **soc** 文件夹主要包含了与芯片密切相关的文件，主要是硬件层和驱动层文件。
- **startup** 文件夹下包含启动相关的文件。
- **user_code** 下为用户程序，相关文件位于 {PROJECT}**user_code** 文件夹下（为叙述方便，下文均以 {PROJECT} 表示工程所在路径），每个工程对应的应用程序均存放在该目录下，该目录下默认有一个 main.c 文件，其中包含了 AMetal 软件包的应用程序入口函数 am_main()。用户开发的其它程序源文件均应存放在 user_code 目录下。
- **user_config** 下为配置文件，相关文件位于 {PROJECT}**user_config** 文件夹下，不同工程可以有不同的配置。

2.3.2 Eclipse 工程结构

建立 Eclipse 工作空间并导入 template_amf07x_core 模板工程，其工程结构如 图 2.3 所示。

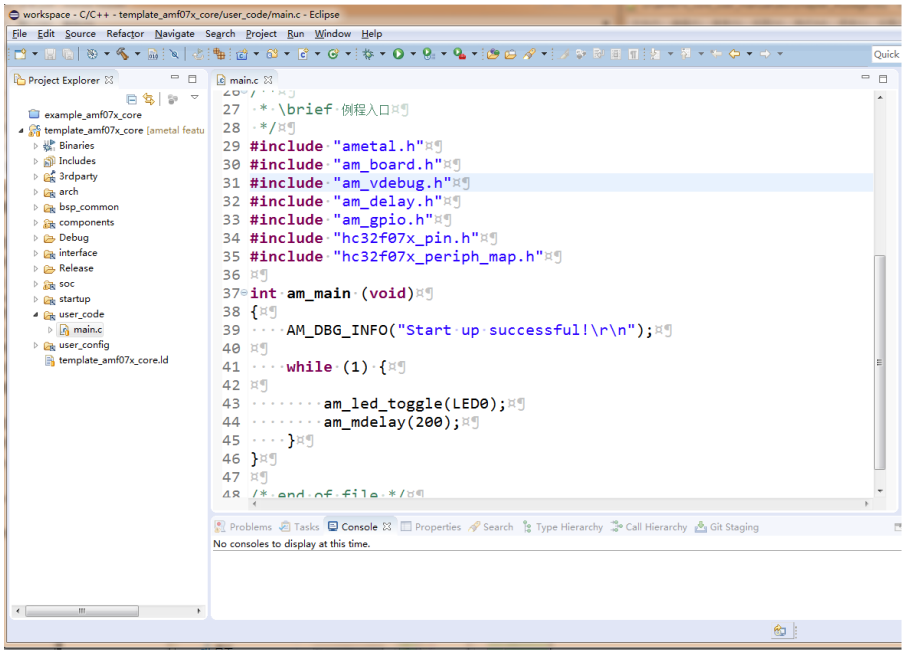


图 2.3 Eclipse 版本工程模板结构

在工程结构中，包含了 3rdparty、arch、bsp_common、components、interface、soc、startup、user_code、user_config 几个节点。详细介绍见 2.2 章节和 keil 工程结构小节。

3. 工程配置

由于系统正常工作时，往往需要初始化一些必要的外设，如 GPIO、中断和时钟等。同时，板上的资源也需要初始化后才能正常使用。为了操作方便，默认情况下，这些资源都在系统启动时自动完成初始化，在进入用户入口函数 **am_main()** 后，这些资源就可以直接使用，非常方便。

但是，一些特殊的应用场合，可能不希望在系统启动时自动初始化一些特定的资源。这时，就可以使用工程配置文件 {PROJECT}**\user_config\am_prj_config.h** 文件禁能一些外设或资源的自动初始化。

3.1 部分外设初始化使能/禁能

一些全局外设，如 CLK、GPIO、DMA、INT 和 NVRAM，由于需要在全局使用，因此在系统启动时已默认初始化，在应用程序需要使用时，无需再重复初始化，直接使用即可。相关的宏在工程配置文件 {PROJECT}**\user_config\am_prj_config.h** 中定义。

以 GPIO 为例，其对应的使能宏为：**AM_CFG_GPIO_ENABLE**，详细定义见 列表 3.1 。宏值默认为 1，即 GPIO 外设在系统启动时自动初始化，如果确定系统不使用 GPIO 资源或希望由应用程序自行完成初始化操作，则可以将该宏的宏值修改为 0。

列表 3.1 GPIO 自动初始化使能/禁能配置

```
1 /** \brief 为 1，初始化 GPIO 的相关功能 */
2 #define AM_CFG_GPIO_ENABLE              1
```

其它一些外设初始化使能/禁能宏定义详见 表 3.1 。配置方式与 GPIO 相同，将宏值修改为 0 即可禁止在系统启动时自动完成初始化。

表 3.1 其它一些外设初始化使能/禁能宏

宏名	对应的外设
AM_CFG_CLK_ENABLE	系统时钟
AM_CFG_INT_ENABLE	中断
AM_CFG_DMA_ENABLE	DMA
AM_CFG_NVRAM_ENABLE	NVRAM

注解： 有的资源除使能外，可能还需要其它一些参数的配置，关于外设参数的配置，可以详见 4.2 节。

3.2 板级资源初始化使能/禁能

与板级相关的资源有 LED、按键、调试串口、延时、系统滴答、软件定时器、标准库和中断延时等。其他板级资源都可以通过配置对应的使能/禁能宏来决定系统启动时是否自动完成初始化操作。相关的宏在工程配置文件 {PROJECT}**\user_config\am_prj_config.h** 中定义。

以 LED 为例，其对应的使能宏为：**AM_CFG_LED_ENABLE** ，详细定义见 列表 3.2 。宏值默认为 1，即 LED 在系统启动时自动完成初始化，如果确定系统不使用 LED 资源或希望由应用程序自行完成初始化操作，则可以将该宏的宏值修改为 0。

列表 3.2 LED 自动初始化使能/禁能配置

```
1  /**
2   * \brief 如果为 1，则初始化 led 的相关功能，板上默认有 4 个 LED
3   *
4   * ID: 0 --- PIOE_0
5   * ID: 1 --- PIOE_1
6   * ID: 2 --- PIOE_2
7   * ID: 3 --- PIOE_3
8   */
9  #define AM_CFG_LED_ENABLE            1
```

其它一些板级资源初始化使能/禁能宏定义详见 表 3.2 。配置方式与 LED 相同，将宏值修改为 0 即可禁止在系统启动时自动完成初始化。

表 3.2 其它一些板级资源初始化使能/禁能宏

宏名	对应资源
AM_CFG_KEY_GPIO_ENABLE	板载按键，使能后才能正常使用板载按键
AM_CFG_DELAY_ENABLE	延时，使能初始化后才能在应用中直接使用 am_udelay() 和 am_mdelay()
AM_CFG_SYSTEM_TICK_ENABLE	系统滴答，默认使用 TIM2 定时器，使能后才能正常使用系统滴答相关功能
AM_CFG_SOFTIMER_ENABLE	软件定时器，使能后才能正常使用软件定时器的相关功能
AM_CFG_DEBUG_ENABLE	串口调试，使能调试输出，使能后，则可以使用 AM_DBG_INFO() 通过串口输出调试信息
AM_CFG_KEY_ENABLE	按键系统，使能后方可管理按键事件
AM_CFG_ISR_DEFER_ENABLE	中断延时，使能后，ISR DEFER 板级初始化，将中断延迟任务放在 PENDSV 中处理
AM_CFG_STDLIB_ENABLE	标准库，使能标准库，则系统会自动适配标准库，用户即可使用 printf()、malloc()、free() 等标准库函数

注解：有的资源除使能外，可能还需要其它一些参数的配置，关于这参数的配置，可以详见第 5 章。

对于延时，每个硬件平台可能具有不同的实现方法，默认实现详见 **ametal\board\bsp_common\am_bsp_delay_timer.c** ，应用可以根据具体需求修改（例如：应用程序不需要精确延时，完全可以使用 for 循环去做一个大概的延时即可，无需再额外耗费一个定时器），因此，将延时部分归类到板级资源下。

对于调试输出，即使用一路串口来输出调试信息，打印出一些关键信息以及变量的值等等，非常方便。

对于软件定时器，需要一个硬件定时器为其提供一个的周期性的定时中断。不同的硬件平台可以有不同的提供方式，因此，同样将软件定时器的初始化部分归类到板级资源下。

4. 外设资源及典型配置

HC32F07X 包含了众多的外设资源，只要 AMetal 软件包提供了对应外设的驱动，就一定会提供一套相应的默认配置信息。所有片上外设的配置由 {PROJECT}**user_config\am_hwconf_usrcfg** (为叙述简便，下文统一使用 {HWCONFIG} 表示该路径) 下的一组 am_hwconf_hc32f07x_* 开头的 **.c** 文件完成的。

注解：为方便介绍本文将与 ARM 内核相关的文件（NVIC 和 Systick）与片上外设资源放在一起，其中 NVIC 中断的配置文件位于 {HCONFIG} 路径下，以 am_hwconf_arm_* 开头。

片上外设及其对应的配置文件如 表 4.1 所示。

表 4.1 片上外设及对应的配置文件

序号	外设	配置文件
1	ADC	am_hwconf_hc32f07x_adc.c
2	时钟	am_hwconf_hc32f07x_clk.c
3	循环冗余校验	am_hwconf_hc32f07x_crc.c
4	DMA	am_hwconf_hc32f07x_dma.c
5	GPIO	am_hwconf_hc32f07x_gpio.c
6	I^2C	am_hwconf_hc32f07x_i2c.c
7	I^2C 从机	am_hwconf_hc32f07x_i2c_slv.c
8	看门狗	am_hwconf_hc32f07x_wdt.c
9	OPA	am_hwconf_hc32f07x_opa.c
10	实时时钟	am_hwconf_hc32f07x_rtc.c
11	SPI(DMA 方式)	am_hwconf_hc32f07x_spi_dma.c
12	SPI(中断方式)	am_hwconf_hc32f07x_spi_int.c
13	SPI(轮询方式)	am_hwconf_hc32f07x_spi_poll.c
14	滴答定时器	am_hwconf_hc32f07x_systick.c
15	标准定时器的捕获功能	am_hwconf_hc32f07x_tim_cap.c
16	标准定时器的 PWM 功能	am_hwconf_hc32f07x_tim_pwm.c
17	标准定时器的定时功能	am_hwconf_hc32f07x_tim_timing.c
18	USART	am_hwconf_hc32f07x_usart.c
19	USB mouse	am_hwconf_hc32f07x_usb_mouse.c
20	CAN	am_hwconf_hc32f07x_can.c
21	NVIC 中断	am_hwconf_arm_nvic.c

每个外设都提供了对应的配置文件，使得看起来配置文件的数量非常之多。但实际上，所有配置文件的结构和配置方法都非常类似，同时，由于所有的配置文件已经是一种常用的默认配置，因此，用户在实际配置时，需要配置的项目非常之少，往往只需要配置外设相关的几个引脚号就可以了。

4.1 配置文件结构

配置文件的核心是定义一个设备实例和设备信息结构体，并提供封装好的实例初始化函数和实例解初始化函数。下面以 GPIO 为例，详述整个配置文件的结构。

4.1.1 设备实例

设备实例为整个外设驱动提供必要的内存空间，设备实例实际上就是使用相应的设备结构体类型定义的一个结构体变量，无需用户赋值。因此，用户完全不需要关心设备结构体类型的具体成员变量，只需要使用设备结构体类型定义一个变量即可。在配置文件中，设备实例均已定义。打开 {HWCONFIG}**am_hwconf_hc32f07x_gpio.c**，可以看到设备实例已经定义好。详见 列表 4.1 。

列表 4.1 定义设备实例

```
/** \brief GPIO 设备实例 */
am_hc32_gpio_dev_t __g_gpio_dev;
```

这里使用 **am_hc32_gpio_dev_t** 类型定义了一个 GPIO 设备实例。设备结构体类型在相对应的驱动头文件中定义。对于通用输入输出 GPIO 外设，该类型即在 **\ametal\soc\hdsc\drivers\include\gpio\am_hc32f07x_gpio.h** 文件中定义。

4.1.2 设备信息

设备信息用于在初始化一个设备时，传递给驱动一些外设相关的信息，如常见的该外设对应的寄存器基地址、使用的中断号等等。设备信息实际上就是使用相应的设备信息结构体类型定义的一个结构体变量，与设备实例不同的是，该变量需要用户赋初值。同时，由于设备信息无需在运行过程中修改，因此往往将设备信息定义为 **const** 变量。

打开 {HWCONFIG}**\am_hwconf_hc32f07x_gpio.c**，可以看到定义的设备信息如 列表 4.2 所示。

列表 4.2 GPIO 设备信息定义

```
/** \brief GPIO 设备信息 */
const am_hc32_gpio_devinfo_t __g_gpio_devinfo = {
    HC32_PORT0_BASE,              /**< \brief GPIO 控制器寄存器块基址 */

    {
    INUM_PORTA,
    INUM_PORTB,
    INUM_PORTC_E,
    INUM_PORTD_F,
    },

    PIN_INT_MAX,                   /**< \brief GPIO 使用的最大外部中断线编号 +1 */

    &__g_gpio_infomap[0],          /**< \brief GPIO 引脚外部事件信息 */
    &__g_gpio_triginfos[0],        /**< \brief GPIO PIN 触发信息 */

    __hc32_plfm_gpio_init,        /**< \brief GPIO 平台初始化 */
    __hc32_plfm_gpio_deinit       /**< \brief GPIO 平台去初始化 */
};
```

这里使用 **am_hc32_gpio_devinfo_t** 类型定义了一个 GPIO 设备信息结构体。设备信息结构体类型在相应的驱动头文件中定义。对于 GPIO，该类型在 **\ametal\soc\hdsc\drivers\include\gpio\am_hc32_gpio.h** 文件中定义。详见 列表 4.3 。

列表 4.3 GPIO 设备信息结构体类型定义

```
/**
 * \brief GPIO 设备信息
 */
typedef struct am_hc32_gpio_devinfo {

    /** \brief GPIO 寄存器块基址 */
    uint32_t                          gpio_regbase;

    /** \brief 引脚中断号列表 */
    const int8_t                      inum_pin[4];

    /** \brief 支持外部中断的 GPIO 最大引脚编号 */
```

```
    const uint8_t                       exti_num_max;

    /** \brief 触发方式 */
    uint8_t                            *p_trigger;

    /** \brief 指向引脚触发信息的指针 */
    struct am_hc32_gpio_trigger_info   *p_triginfo;

    /** \brief 平台初始化函数 */
    void                              (*pfn_plfm_init) (void);

    /** \brief 平台去初始化函数 */
    void                              (*pfn_plfm_deinit) (void);

} am_hc32_gpio_devinfo_t;
```

可见，共计有 7 个成员。设备信息一般仅由 5 部分构成：寄存器基地址、中断号、需要用户根据实际情况分配的内存、平台初始化函数和平台解初始化函数。下面一一解释各个部分的含义。

1. 寄存器基地址

每个片上外设都有对应的寄存器，这些寄存器有一个起始地址（基地址），只要根据这个起始地址，就能够操作到所有寄存器。因此，设备信息需要提供外设的基地址。

寄存器基地址已经在 **\ametal\soc\hdsc\hc32f07x\hc32f07x_regbase.h** 文件中使用宏定义好了，用户直接使用即可。对于 GPIO 相关的寄存器基地址，详见 列表 4.4 。

列表 4.4 外设寄存器基地址定义

```
/** \brief GPIO 基地址 */
#define HC32F07X_PORT0_BASE             (0x40020C00UL)
```

可见，列表 4.2 中，设备信息成员的赋值均来自于此。

2. 中断号

中断号对应了外设的中断服务入口，需要将该中断号传递给驱动，以便驱动使用相应的中断资源。

对于绝大部分外设，中断入口只有一个，因此中断号也只有一个，一些特殊的外设，中断号可能存在多个，如 GPIO，中断的产生来源于 EXTI，EXTI 最高可提供 16 路中断，为了快速响应，与之对应地，系统提供了 4 路中断服务入口给 EXTI，因此，EXTI 共计有 4 个中断号，在设备信息结构体类型中，为了方便提供所有的中断号，使用了一个大小为 4 的数组。详见 列表 4.5 。

列表 4.5 GPIO 设备信息结构体类型——中断号成员定义

```
/** \brief EXIT 中断号列表 */
const int8_t  inum_pin[4];
```

所有中断号已经在 **\ametal\soc\hdsc\hc32f07x\hc32f07x_inum.h** 文件中定义好了，与 EXTI 相关的中断号定义详见 列表 4.6 。

列表 4.6 EXTI 各个中断号定义

```
#define INUM_PORTA                  0    /**< \brief PORTA 中断 */
#define INUM_PORTB                  1    /**< \brief PORTB 中断 */
#define INUM_PORTC_E                2    /**< \brief PORTC_E 中断 */
```

```
#define INUM_PORTD_F                3    /**< \brief PORTD_F 中断 */
```

实际为结构体信息的中断号成员赋值时，只需要使用定义好的宏为相应的设备信息结构体赋值即可。可见，列表 4.2 中，设备信息中断号成员的赋值均来自于此。

3. 时钟 ID 号

时钟 ID 号对应了外设的时钟来源，需要将该时钟 ID 号传递给驱动，以便驱动中可以获取外设的频率及使能该外设的相关时钟。所有时钟 ID 号已经在 **\ametal\soc\hdsc\hc32f07x\hc32f07x_clk.h** 文件中定义好了，详见 列表 4.7 。

列表 4.7 时钟 ID 号

```
/* 系统初始相关 */
#define CLK_XTHOSC    (0ul)  /**< \brief 外部高速时钟 */
#define CLK_XTLOSC    (1ul)  /**< \brief 外部低速时钟 */
#define CLK_RCH       (2ul)  /**< \brief 内部高速时钟 */
#define CLK_RCL       (3ul)  /**< \brief 内部低速时钟 */
#define CLK_PLLIN     (4ul)  /**< \brief PLL 输入时钟 */
#define CLK_PLLOUT    (5ul)  /**< \brief PLL 输出时钟 */
#define CLK_SYSCLK    (6ul)  /**< \brief SYS 时钟 */
#define CLK_HCLK      (7ul)  /**< \brief 主时钟 */
#define CLK_PCLK      (8ul)  /**< \brief 外设时钟 */

/* hc32 所有外设时钟均挂载在 PCLK 时钟下 */
#define CLK_FLASH     ((0x1ul << CLK_PCLK) | 31ul)
#define CLK_HDIV      ((0x1ul << CLK_PCLK) | 30ul)
#define CLK_DMA       ((0x1ul << CLK_PCLK) | 29ul)
#define CLK_GPIO      ((0x1ul << CLK_PCLK) | 28ul)
#define CLK_AES       ((0x1ul << CLK_PCLK) | 27ul)
#define CLK_CRC       ((0x1ul << CLK_PCLK) | 26ul)
#define CLK_SWD       ((0x1ul << CLK_PCLK) | 25ul)
#define CLK_TICK      ((0x1ul << CLK_PCLK) | 24ul)
#define CLK_LCD       ((0x1ul << CLK_PCLK) | 22ul)
#define CLK_CLOCKTRIM ((0x1ul << CLK_PCLK) | 21ul)
#define CLK_RTC       ((0x1ul << CLK_PCLK) | 20ul)
#define CLK_PCNT      ((0x1ul << CLK_PCLK) | 19ul)
#define CLK_RNG       ((0x1ul << CLK_PCLK) | 18ul)
#define CLK_VC_LVD    ((0x1ul << CLK_PCLK) | 17ul)
#define CLK_ADC_BGR   ((0x1ul << CLK_PCLK) | 16ul)
#define CLK_WDT       ((0x1ul << CLK_PCLK) | 15ul)
#define CLK_PCA       ((0x1ul << CLK_PCLK) | 14ul)
#define CLK_OPA       ((0x1ul << CLK_PCLK) | 13ul)
#define CLK_TIM3      ((0x1ul << CLK_PCLK) | 11ul)
#define CLK_TIM456    ((0x1ul << CLK_PCLK) | 10ul)
#define CLK_LPTIM0    ((0x1ul << CLK_PCLK) | 9ul)
#define CLK_TIM012    ((0x1ul << CLK_PCLK) | 8ul)
#define CLK_SPI1      ((0x1ul << CLK_PCLK) | 7ul)
#define CLK_SPI0      ((0x1ul << CLK_PCLK) | 6ul)
#define CLK_I2C1      ((0x1ul << CLK_PCLK) | 5ul)
#define CLK_I2C0      ((0x1ul << CLK_PCLK) | 4ul)
#define CLK_LPUART1   ((0x1ul << CLK_PCLK) | 3ul)
#define CLK_LPUART0   ((0x1ul << CLK_PCLK) | 2ul)
#define CLK_UART1     ((0x1ul << CLK_PCLK) | 1ul)
#define CLK_UART0     ((0x1ul << CLK_PCLK) | 0ul)
#define CLK_UART3     ((0x1ul << CLK_PCLK) | 41ul)
#define CLK_UART2     ((0x1ul << CLK_PCLK) | 40ul)
#define CLK_I2S1      ((0x1ul << CLK_PCLK) | 38ul)
#define CLK_I2S0      ((0x1ul << CLK_PCLK) | 37ul)
#define CLK_LPTIM1    ((0x1ul << CLK_PCLK) | 36ul)
#define CLK_DAC       ((0x1ul << CLK_PCLK) | 35ul)
```

```
#define CLK_CTS       ((0x1ul << CLK_PCLK) | 34ul)
#define CLK_CAN       ((0x1ul << CLK_PCLK) | 33ul)
#define CLK_USB       ((0x1ul << CLK_PCLK) | 32ul)
```

在 GPIO 设备信息当中，没有使用时钟 ID 号，故不需要配置。在很大一部分外设，需要使用时钟 ID 号，如串口外设，详见 列表 4.8 。

列表 4.8 串口外设时钟 ID 号

```
/** \brief 串口 1 设备信息 */
static const am_hc32_uart_devinfo_t __g_uart1_devinfo = {

    HC32_UART1_BASE,                /**< \brief 串口 1 */
    INUM_UART1_3,                   /**< \brief 串口 1 的中断编号 */
    CLK_UART1,                      /**< \brief 串口时钟 ID */

    AMHW_HC32_UART_PARITY_NO |      /**< \brief 无极性 */
    AMHW_HC32_UART_STOP_1_0_BIT,    /**< \brief 1 个停止位 */

    AM_FALSE,                       /**< \brief 不使用异步半双工（单线）模式 */

    115200,                         /**< \brief 设置的波特率 */

    0,                              /**< \brief 无其他中断 */

    NULL,                           /**< \brief 使用 RS485 */
    __hc32_plfm_uart1_init,         /**< \brief UART1 的平台初始化 */
    __hc32_plfm_uart1_deinit,       /**< \brief UART1 的平台去初始化 */
};
```

4. 需要用户根据实际情况分配的内存

前文已经提到，设备实例是用来为外设驱动分配内存的，为什么在设备信息中还需要分配内存呢？

这是因为系统有的资源提供得比较多，而用户实际使用数量可能远远小于系统提供的资源数，如果按照默认都使用的操作方式，将会造成不必要的资源浪费。

以 EXTI 为例，系统提供了 16 个外部中断线连接 4 路中断信号，最多可以将 16 个 GPIO 用作触发模式。每一路 GPIO 触发模式就需要内存来保存用户设定的触发回调函数。如果按照默认，假定用户可能会使用到所有的 16 路 GPIO 触发，则就需要 16 份用于保存相关信息的内存空间。而实际上，用户可能只使用 1 路，这就导致了不必要的空间浪费。

基于此，某些可根据用户实际情况增减的内存由用户通过设备信息提供。以实现资源的最优化利用。

在设备信息结构体类型中，相关的成员有 3 个，详见 列表 4.9 。

列表 4.9 GPIO 设备信息结构体类型——内存分配相关成员定义

```
/** \brief GPIO 使用的最大外部中断线编号 +1 */
const uint8_t exti_num_max;

/** \brief 触发信息映射 */
uint8_t      *p_trigger;

......

/** \brief 指向引脚触发信息的指针 */
struct am_hc32_gpio_trigger_info   *p_triginfo;
```

- 成员 **exti_num_max** 使用的最大中断线加 1，例如仅使用 EXTI 线 0（对应引脚为 PIOA0/PIOB0/PIOC0/PIOD0），则该变量应为 1。
- 成员 **p_trigger** 用于保存触发信息的映射关系，对应内存的大小应该与 **exti_num_max** 一致。
- 成员 **p_triginfo** 用于保存触发信息，主要包括触发回调函数和回调函数对应的参数，对应内存的大小应该与 **exti_num_max** 一致。类型 **am_hc32_gpio_trigger_info** 同样在 GPIO 驱动头文件 **\ametal\soc\hdsc\drivers\include\gpio\am_hc32_gpio.h** 中定义，详见 列表 4.10 。

列表 4.10 GPIO 触发信息结构体类型定义

```
/**
 * \brief 引脚的触发信息
 */
struct am_hc32_gpio_trigger_info {

    /** \brief 触发回调函数 */
    am_pfnvoid_t                        pfn_callback;

    /** \brief 回调函数的参数 */
    void                               *p_arg;
};
```

可见，虽然有三个成员，但实际上可配置的核心就是 **exti_num_max** ，另外有两个成员的实际的内存大小应该与该参数一致，为了方便用户根据实际情况配置，用户配置文件中，提供了默认的这两个成员的值定义，详见 列表 4.11 。

列表 4.11 需要用户根据实际情况分配的内存定义

```
/** \brief 引脚触发信息内存 */
static struct am_hc32_gpio_trigger_info __g_gpio_triginfos[PIN_INT_MAX];

/** \brief 引脚触发信息映射 */
static uint8_t __g_gpio_infomap[PIN_INT_MAX];
```

设备信息结构体的赋值详见 列表 4.2，其中，使用 **PIN_INT_MAX** 宏作为 **exti_num_max** 的值；使用 **&__g_gpio_infomap[0]** 作为 **p_infomap** 的值；使用 **&__g_gpio_triginfos[0]** 作为 **p_triginfo** 的参数。

默认情况下，**PIN_INT_MAX** 的值即为硬件支持的最大 EXTI 通道数目。

注解： 实际中，有的外设可能不需要根据实际分配内存。那么，设备信息结构体中将不包含该部分内容。

---

5. 平台初始化函数

平台初始化函数主要用于初始化与该外设相关的平台资源，如使能该外设的时钟，初始化与该外设相关的引脚等。一些通信接口，都需要配置引脚，如 UART、SPI、I^2C 等，这些引脚的初始化都需要在平台初始化函数中完成。

在设备信息结构体类型中，均有一个用于存放平台初始化函数的指针，以指向平台初始化函数，详见 列表 4.12 。当驱动程序初始化相应外设前，将首先调用设备信息中提供的平台初始化函数。

列表 4.12 GPIO 设备信息结构体类型——平台初始化函数指针定义

```
void (*pfn_plfm_init)(void);   /**< \brief 平台初始化函数 */
```

平台初始化函数均在设备配置文件中定义，GPIO 的平台初始化函数在 {HWCONFIG}**am_hwconf_hc32f07x_gpio.c** 文件中定义，详见 列表 4.13 。

列表 4.13 GPIO 平台初始化函数

```
/** \brief GPIO 平台初始化 */
void __hc32_plfm_gpio_init (void)
{
    am_clk_enable(CLK_GPIO);

    am_hc32f07x_clk_reset(CLK_GPIO);
}
```

平台初始化函数中，使能了与 GPIO 相关外设 PORT 端口的门控时钟。

am_hc32f07x_clk_reset() 函数用于复位一个外设，在 **\ametal\soc\hdsc\drivers\source\clk\am_hc32f07x_clk.c** 文件中定义。函数原型详见 列表 4.14 。

列表 4.14 am_hc32f07x_clk_reset() 函数原型

```
/**
 * \brief CLK 外设复位
 *
 * \param[in] clk_id  时钟 ID (由平台定义)
 *
 * \retval AM_OK : 操作成功
 */
int am_hc32f07x_clk_reset (am_clk_id_t clk_id);
```

参数为 **am_clk_id_t** 类型，用于指定需要复位的外设时钟门控，在 **\ametal\soc\hdsc\hc32f07x\hc32f07x_clk.h** 文件中定义。可见 列表 4.7 。

在平台初始化函数中，参数 CLK_IOPA 表示复位 GPIO PORTA 外设。

am_clk_enable() 函数用于使能一个外设的时钟，在 **\ametal\soc\hdsc\drivers\source\clk\am_hc32f07x_clk.c** 文件中定义。函数原型详见 列表 4.15 。

列表 4.15 am_clk_enable() 函数原型

```
/**
 * \brief 使能时钟
 *
 * \param[in]   clk_id  时钟 ID (由平台定义)
 *
 * \retval      AM_OK       成功
 * \retval     -AM_ENXIO    时钟频率 ID 不存在
 * \retval     -AM_EIO      使能失败
 */
int am_clk_enable (am_clk_id_t clk_id);
```

参数为 **am_clk_id_t** 类型，用于指定需要使能时钟的外设，在 **\ametal\soc\hdsc\hc32f07x\hc32f07x_clk.h** 文件中定义。可见 列表 4.7 。

在设备信息结构体赋值时，详见 列表 4.2 ，直接以该函数的函数名作为平台初始化函数

指针成员的值。

某些特殊的外设，很可能不需要平台初始化函数，这时，只需要将平台初始化函数指针成员赋值为 NULL 即可。

6. 平台解初始化函数

平台解初始化函数与平台初始化函数对应，平台初始化函数打开了的时钟等，就可以通过平台解初始化函数关闭。

在设备信息结构体类型中，均有一个用于存放平台解初始化函数的指针，以指向平台解初始化函数，详见 列表 4.16 。当不再需要使用某个外设时，驱动在解初始化相应外设后，将调用设备信息中提供的平台解初始化函数，以释放掉平台提供的相关资源。

列表 4.16 GPIO 设备信息结构体类型——平台解初始化函数指针定义

```
void (*pfn_plfm_deinit)(void); /**< \brief 平台去初始化函数 */
```

平台解初始化函数均在设备配置文件中定义，GPIO 的平台解初始化函数在 {HWCONFIG}**am_hwconf_hc32f07x_gpio.c** 文件中定义，详见 列表 4.17 。

列表 4.17 GPIO 平台解初始化函数

```
/** \brief GPIO 平台解初始化 */
void __hc32_plfm_gpio_deinit (void)
{
    am_hc32f07x_clk_reset(CLK_GPIO);

    am_clk_disable(CLK_GPIO);
}
```

平台解初始化函数中，复位了 GPIO，并关闭了各个相关外设的时钟。

am_clk_disable() 函数与 **am_clk_enable()** 对应，用于关闭相关外设的时钟，该函数在 **\ametal\soc\hdsc\drivers\source\clk\am_hc32f07x_clk.c** 文件中定义。函数原型详见 列表 4.18 。

列表 4.18 am_clk_disable() 函数原型

```
/**
 * \brief 禁能时钟
 *
 * \param[in]   clk_id  时钟 ID (由平台定义), 参见 \ref grp_clk_id
 *
 * \retval      AM_OK     成功
 * \retval     -AM_ENXIO  时钟频率 ID 不存在
 * \retval     -AM_EIO       禁能失败
 */
int am_clk_disable (am_clk_id_t clk_id);
```

参数为 **am_clk_id_t** 类型，用于指定需要关闭时钟的外设，在 **\ametal\soc\hdsc\hc32f07x\hc32f07x_clk.h** 文件中定义。可见 列表 4.7 。

在设备信息结构体赋值时，详见 列表 4.2 ，直接以该函数的函数名作为平台解初始化函数指针成员的值。

某些特殊的外设，很可能不需要平台解初始化函数，这时，只需要将平台解初始化函数指针成员赋值为 NULL 即可。

综上，以 GPIO 为例，讲述了设备信息结构体中的 6 个部分，这些均只需要了解即可，

在查看其它外设的设备信息结构体时，只要按照这个结构，就可以很清晰的理解各个部分的用途。

实际上，设备信息结构体中所有的成员，均已提供一种默认的配置。需要用户手动配置的地方少之又少。从 GPIO 的设备配置信息可以看出，虽然设备信息包含了 7 个成员，而真正需要用户根据实际情况配置的内容，仅仅只有一个宏 **PIN_INT_MAX**（详见 列表 4.11 ）。

除了常见的 GPIO 设备信息中的这 6 个部分外，还可能包含一些需要简单配置的值，如 ADC 中的参考电压等，这些配置内容，从意义上很好理解，就不再赘述。

4.1.3 实例初始化函数

任何外设使用前，都需要初始化。通过前文的讲述，设备配置文件中已经定义好了设备实例和设备信息结构体。至此，只需要再调用相应的驱动提供的外设初始化函数，传入对应的设备实例地址和设备信息的地址，即可完成该外设的初始化。

以 GPIO 为例，GPIO 的驱动初始化函数在 GPIO 驱动头文件 **\ametal\soc\hdsc\drivers\include\gpio\am_hc32_gpio.h** 中声明。详见 列表 4.19 。

列表 4.19 GPIO 初始化函数

```
/**
 * \brief GPIO 初始化
 *
 * \param[in] p_dev     : 指向 GPIO 设备的指针
 * \param[in] p_devinfo : 指向 GPIO 设备信息的指针
 *
 * \retval AM_OK : 操作成功
 */
int am_hc32_gpio_init (am_hc32_gpio_dev_t           *p_dev,
                       const am_hc32_gpio_devinfo_t *p_devinfo);
```

因此，要完成 GPIO 的初始化，只需要调用一下该函数即可，详见 列表 4.20 。

列表 4.20 完成 GPIO 初始化

```
am_hc_gpio_init(&__g_gpio_dev, &__g_gpio_devinfo);
```

__g_gpio_dev 和 **__g_gpio_devinfo** 分别为前面在设备配置文件中定义的设备实例和设备信息。

可见，该初始化动作行为很单一，仅仅是调用一下外设初始化函数，并传递已经定义好的设备实例地址和设备信息地址。

为了进一步减少用户的工作，设备配置文件中，将该初始化动作封装为一个函数，该函数即为实例初始化函数，用于初始化一个外设。

以 GPIO 为例，实例初始化函数定义在 {HWCONFIG}**\am_hwconf_hc32f07x_gpio.c** 文件中，详见 列表 4.21 。

列表 4.21 GPIO 实例初始化函数

```
/** \brief GPIO 实例初始化 */
int am_hc32_gpio_inst_init (void)
{
    return am_hc32_gpio_init(&__g_gpio_dev, &__g_gpio_devinfo);
}
```

这样，要初始化一个外设，用户只需要调用对应的实例初始化函数即可。实例初始化函

数无任何参数，使用起来非常方便。

关于实例初始化函数的返回值，往往与对应的驱动初始化函数返回值一致。根据驱动初始化函数的不同，可能有三种不同的返回值。

(1) 返回值为 int 类型

一些资源全局统一管理的设备，返回值就是一个 int 值。**AM_OK** 即表示初始化成功；其它值表明初始化失败。

(2) 返回值为标准服务句柄

绝大部分外设驱动初始化函数均是返回一个标准的服务句柄（handle），以提供标准服务。值为 NULL 表明初始化失败；其它值表明初始化成功。若初始化成功，则可以使用获取到的 handle 作为标准接口层相关函数的参数，操作对应的外设。

(3) 返回值为驱动自定义服务句柄

一些较为特殊的外设，功能还没有被标准接口层标准化。此时，为了方便用户使用一些特殊功能，相应驱动初始化函数就直接返回一个驱动自定义的服务句柄（handle），值为 NULL 表明初始化失败；其它值表明初始化成功。若初始化成功，则可以使用该 handle 作为该外设驱动提供的相关服务函数的参数，用来使用一些标准接口未抽象的功能或该外设的一些较为特殊的功能。特别地，如果一个外设在提供特殊功能的同时，还可以提供标准服务，那么该外设对应的驱动还会提供一个标准服务 handle 获取函数，通过自定义服务句柄获取到标准服务句柄。

4.1.4 实例解初始化函数

每个外设驱动都提供了对应的驱动解初始化函数，以便当应用不再使用某个外设时，释放掉相关资源。以 GPIO 为例，GPIO 的驱动解初始化函数在 GPIO 驱动头文件 **\ametal\soc\hdsc\drivers\include\gpio\am_hc32_gpio.h** 中声明。详见 列表 4.22 。

列表 4.22 GPIO 解初始化函数

```
/**
 * \brief GPIO 解初始化
 *
 * \param[in] 无
 *
 * \return 无
 */
void am_hc32_gpio_deinit (void);
```

当应用不再使用该外设时，只需要调用一下该函数即可，详见 列表 4.23 。

列表 4.23 完成 GPIO 解初始化

```
am_hc32_gpio_deinit();
```

为了方便用户理解，使用户使用起来更简单，与实例初始化函数相对应，每个设备配置文件同样提供了一个实例解初始化函数。用于当不再使用一个外设时，解初始化该外设，释放掉相关资源。

这样，用户需要使用一个外设时，完全不用关心驱动解初始化函数，只需要调用用户配置文件提供的实例解初始化函数解初始化外设即可。

以 GPIO 为例，实例解初始化函数定义在 {HWCONFIG}**\am_hwconf_hc32f07x_gpio.c**

文件中，详见 列表 4.24 。

列表 4.24 GPIO 实例解初始化函数

```
/** \brief GPIO 实例解初始化 */
void am_hc32_gpio_inst_deinit (void)
{
    am_hc32_gpio_deinit();
}
```

所有实例解初始化函数均无返回值。解初始化后，该外设即不再可用。如需再次使用，需要重新调用实例初始化函数。

根据设备的不同，实例解初始化函数的参数会有不同。若实例初始化函数返回值为 int 类型，则解初始化时，无需传入任何参数；若实例初始化函数返回了一个 handle，则解初始化时，应该传入通过实例初始化函数获取到的 handle 作为参数。

4.2 典型配置

在上一节中，以 GPIO 为例，详细讲解了设备配置文件的结构以及各个部分的含义。虽然设备配置文件内容较多，但是对于用户来讲，需要自行配置的项目却非常少，往往只需要配置极少的内容，然后使用设备配置文件提供的实例初始化函数即可完成一个设备的初始化。

由于所有配置文件的结构非常相似，下文就不再一一完整地列出所有外设的设备配置信息内容。仅仅将各个外设在使用过程中，实际需要用户配置的内容列出，告知用户该如何配置。

4.2.1 ADC

HC32F07X 有 1 个 12 位 ADC，它支持可配置的参考电压和精度、可选的硬件转换触发等功能。

下面以 ADC 标准转换功能 (中断方式) 为示例，讲解 ADC 设备信息结构体 **am_hc32_adc_devinfo_t** ，一些用户根据具体的情况可能需要配置。一般来说，仅仅需配置 ADC 的参考电压、转换精度。

1. ADC 参考电压

ADC 参考电压由引脚 VREFP 和 VREFN 之间的电压差值决定，ADC 参考电压默认为 3.3V，即 3300mV。设备信息中，默认的参考电压即为 3300（单位:mV）。详见 列表 4.25 。

列表 4.25 ADC 参考电压默认配置

```
/** \brief 设备信息 */
static const am_hc32_adc_devinfo_t __g_adc_devinfo = {
    HC32_ADC_BASE,                      /**< \brief ADC */
    INUM_ADC_DAC,                       /**< \brief ADC 的中断编号 */
    CLK_ADC_BGR,                        /**< \brief ADC 时钟号 */
    AMHW_HC32_ADC_REFVOL_AVCC,          /**< \brief 参考电压选择 */
    3300,                               /**< \brief 参考电压 (mv)*/
    0,                                  /**< \brief ADC 通道 28 内部温度传感器开启使能
                                         *          1：开启， 0：关闭
                                         */
    12,                                 /**< \brief 转换精度，hc32 精度只能为 12 位 */
```

```
    &__g_adc_ioinfo_list[15],           /**< \brief 引脚信息列表 */
    __hc32_plfm_adc_init,               /**< \brief ADC1 的平台初始化 */
    __hc32_plfm_adc_deinit,             /**< \brief ADC1 的平台去初始化 */

};
```

如需修改，只需要将配置信息中的参考电压值修改为实际的值即可。

2. ADC 通道引脚配置

hc32f07x ADC 最高可多达 41 个通道，0-23 通道对应的引脚详 表 4.2 所示。

表 4.2 ADC 通道对应的引脚号

序号	通道	引脚
1	ADC_IN0	PIOA_0
2	ADC_IN1	PIOA_1
3	ADC_IN2	PIOA_2
4	ADC_IN3	PIOA_3
5	ADC_IN4	PIOA_4
6	ADC_IN5	PIOA_5
7	ADC_IN6	PIOA_6
8	ADC_IN7	PIOA_7
9	ADC_IN8	PIOB_0
10	ADC_IN9	PIOB_1
11	ADC_IN10	PIOC_0
12	ADC_IN11	PIOC_1
13	ADC_IN12	PIOC_2
14	ADC_IN13	PIOC_3
15	ADC_IN14	PIOC_4
16	ADC_IN15	PIOC_5
17	ADC_IN16	PIOB_2
18	ADC_IN17	PIOB_10
19	ADC_IN18	PIOB_11
20	ADC_IN19	PIOB_12
21	ADC_IN20	PIOB_13
22	ADC_IN21	PIOB_14
23	ADC_IN22	PIOB_15
24	ADC_IN23	PIOE_15

如需使用一个通道，需要先配置一个与该通道对应的引脚。引脚的配置在平台初始化中完成。默认平台初始化函数详见程序清单 列表 4.26 。

列表 4.26 ADC 平台初始化

```
/**
 *  \brief  ADC 的引脚配置信息列表
```

```
 *
 *  \note1 每个成员代表每个 ADC 通道的引脚配置，例如：AMHW_HC32_CHAN_AIN0_PA0，代表 TIM0 捕获
 *         ADC 通道 0 的引脚配置
 *
 *  \note2 使用 AMHW_HC32_CHAN_INSIDE_TEMP_SENSOR 时，要使能内部温度传感器
 *
 */
am_hc32_adc_ioinfo_t __g_adc_ioinfo_list[] = {
    {AMHW_HC32_CHAN_AIN0_PA0 ,  PIOA_0 , PIOA_0_GPIO  | PIOA_0_AIN , PIOA_0_GPIO  | PIOA_0_INPUT_PU },
    {AMHW_HC32_CHAN_AIN1_PA1 ,  PIOA_1 , PIOA_1_GPIO  | PIOA_1_AIN , PIOA_1_GPIO  | PIOA_1_INPUT_PU },
    {AMHW_HC32_CHAN_AIN2_PA2 ,  PIOA_2 , PIOA_2_GPIO  | PIOA_2_AIN , PIOA_2_GPIO  | PIOA_2_INPUT_PU },
    {AMHW_HC32_CHAN_AIN3_PA3 ,  PIOA_3 , PIOA_3_GPIO  | PIOA_3_AIN , PIOA_3_GPIO  | PIOA_3_INPUT_PU },
    {AMHW_HC32_CHAN_AIN4_PA4 ,  PIOA_4 , PIOA_4_GPIO  | PIOA_4_AIN , PIOA_4_GPIO  | PIOA_4_INPUT_PU },
    {AMHW_HC32_CHAN_AIN5_PA5 ,  PIOA_5 , PIOA_5_GPIO  | PIOA_5_AIN , PIOA_5_GPIO  | PIOA_5_INPUT_PU },
    {AMHW_HC32_CHAN_AIN6_PA6 ,  PIOA_6 , PIOA_6_GPIO  | PIOA_6_AIN , PIOA_6_GPIO  | PIOA_6_INPUT_PU },
    {AMHW_HC32_CHAN_AIN7_PA7 ,  PIOA_7 , PIOA_7_GPIO  | PIOA_7_AIN , PIOA_7_GPIO  | PIOA_7_INPUT_PU },
    {AMHW_HC32_CHAN_AIN8_PB0 ,  PIOB_0 , PIOB_0_GPIO  | PIOB_0_AIN , PIOB_0_GPIO  | PIOB_0_INPUT_PU },
    {AMHW_HC32_CHAN_AIN9_PB1 ,  PIOB_1 , PIOB_1_GPIO  | PIOB_1_AIN , PIOB_1_GPIO  | PIOB_1_INPUT_PU },
    {AMHW_HC32_CHAN_AIN10_PC0,  PIOC_0 , PIOC_0_GPIO  | PIOC_0_AIN , PIOC_0_GPIO  | PIOC_0_INPUT_PU },
    {AMHW_HC32_CHAN_AIN11_PC1,  PIOC_1 , PIOC_1_GPIO  | PIOC_1_AIN , PIOC_1_GPIO  | PIOC_1_INPUT_PU },
    {AMHW_HC32_CHAN_AIN12_PC2,  PIOC_2 , PIOC_2_GPIO  | PIOC_2_AIN , PIOC_2_GPIO  | PIOC_2_INPUT_PU },
    {AMHW_HC32_CHAN_AIN13_PC3,  PIOC_3 , PIOC_3_GPIO  | PIOC_3_AIN , PIOC_3_GPIO  | PIOC_3_INPUT_PU },
    {AMHW_HC32_CHAN_AIN14_PC4,  PIOC_4 , PIOC_4_GPIO  | PIOC_4_AIN , PIOC_4_GPIO  | PIOC_4_INPUT_PU },
    {AMHW_HC32_CHAN_AIN15_PC5,  PIOC_5 , PIOC_5_GPIO  | PIOC_5_AIN , PIOC_5_GPIO  | PIOC_5_INPUT_PU },
    {AMHW_HC32_CHAN_AIN16_PB2,  PIOB_2 , PIOB_2_GPIO  | PIOB_2_AIN , PIOB_2_GPIO  | PIOB_2_INPUT_PU },
    {AMHW_HC32_CHAN_AIN17_PB10, PIOB_10, PIOB_10_GPIO | PIOB_10_AIN, PIOB_10_GPIO | PIOB_10_INPUT_PU},
    {AMHW_HC32_CHAN_AIN18_PB11, PIOB_11, PIOB_11_GPIO | PIOB_11_AIN, PIOB_11_GPIO | PIOB_11_INPUT_PU},
    {AMHW_HC32_CHAN_AIN19_PB12, PIOB_12, PIOB_12_GPIO | PIOB_12_AIN, PIOB_12_GPIO | PIOB_12_INPUT_PU},
    {AMHW_HC32_CHAN_AIN20_PB13, PIOB_13, PIOB_13_GPIO | PIOB_13_AIN, PIOB_13_GPIO | PIOB_13_INPUT_PU},
    {AMHW_HC32_CHAN_AIN21_PB14, PIOB_14, PIOB_14_GPIO | PIOB_14_AIN, PIOB_14_GPIO | PIOB_14_INPUT_PU},
    {AMHW_HC32_CHAN_AIN22_PB15, PIOB_15, PIOB_15_GPIO | PIOB_15_AIN, PIOB_15_GPIO | PIOB_15_INPUT_PU},
    {AMHW_HC32_CHAN_AIN23_PE15, PIOE_15, PIOE_15_GPIO | PIOE_15_AIN, PIOE_15_GPIO | PIOE_15_INPUT_PU},
    {AMHW_HC32_CHAN_DAC_OUT   , 0                                   , 0                    },
    {AMHW_HC32_CHAN_VCC_DIV3  , 0                                   , 0                    },
    {AMHW_HC32_CHAN_INSIDE_TEMP_SENSOR, 0                           , 0                    },
```

```
    {AMHW_HC32_CHAN_INSIDE_REFVCC_12,   0                          , 0          ␣
↪                   },
};
```

4.2.2 AES

AES 没有自定义参数需要配置，同时，也没有外部相关的引脚。因此，该外设完全不需要用户参与配置，实际需要使用时，调用其对应的实例初始化函数即可。

4.2.3 CLK

时钟配置，主要是配置时钟源以及时钟源的倍频，分频系数。设备信息详见 列表 4.27 。

列表 4.27 时钟默认设备信息

```
/*
 * PLL 可以选择 XTH 晶振生成的时钟、XTH 从管脚 PF00 输入的时钟、RCH 时钟作为 PLL 时钟源
 * 还可以设置 PLL 的倍频系数，输出得到一个频率，从而可以用作系统时钟。
 *
 * 系统时钟源的选择主要有五种，   RCH  ---  内部高速时钟
 *                        XTH  ---  外部高速时钟
 *                        RCL  ---  内部低速时钟
 *                        XTL  ---  外部低速时钟
 *                        PLL  ---  内部 PLL 输出时钟  = PLL 输入时钟 * PLL 倍频系数
 *
 * 系统时钟源选定，则得到以下关系
 *
 * SYSCLK = 被选择的系统时钟源
 * HCLK   = SYSCLK / (2 ^ hclk_div)
 * PCLK   = HCLK   / (2 ^ pclk_div)
 *
 * \note 系统进入 DeepSleep 模式，高速时钟（RCH、XTH）自动关闭，
 *       需手动切换到低速时钟（RCL、XTL）作为系统时钟源。
 */
/** \brief CLK 设备信息 */
static const am_hc32_clk_devinfo_t __g_clk_devinfo =
{
    /**
     * \brief XTH 外部高速时钟晶振频率（Hz）
     */
    8000000,

    /**
     * \brief XTL 外部低速时钟晶振频率（Hz）
     */
    32768,

    /**
     * \brief RCH 内部高速时钟晶振频率（Hz）
     *
     * \note 频率选择    24M、 22.12M、16M、 8M 或 4M
     *       设置其他频率，将默认使用 4M
     */
    AMHW_HC32_RCH_FRE_4MHz,

    /**
     * \brief RCL 内部低速时钟晶振频率（Hz）
     *
     * \note 频率选择    38.4k  或      32.768K
     *       设置其他频率，将默认使用 38.4k
```

```
     */
    AMHW_HC32_RCl_FRE_38400Hz,

    /** \brief
     *    PLL 时钟源选择
     *    -# AMHW_HC32_PLL_INPUT_FRE_SRC_XTH_XTAL : XTH 晶振生成的时钟
     *    -# AMHW_HC32_PLL_INPUT_FRE_SRC_XTH_PF00 : XTH 从管脚 PF00 输入的时钟
     *    -# AMHW_HC32_PLL_INPUT_FRE_SRC_RCH      : RCH 时钟
     */
   AMHW_HC32_PLL_INPUT_FRE_SRC_XTH_XTAL,

    /**
     * \brief PLL 倍频系数，允许范围 2 ~ 12
     *        PLLOUT = PLLIN * pll_mul
     */
    6,

    /** \brief
     *    系统 时钟源选择
     *    -# AMHW_HC32_SYSCLK_RCH : 内部高速时钟作为系统时钟
     *    -# AMHW_HC32_SYSCLK_XTH : 外部高速时钟作为系统时钟
     *    -# AMHW_HC32_SYSCLK_RCL : 内部低速时钟作为系统时钟
     *    -# AMHW_HC32_SYSCLK_XTL :   外部低速时钟作为系统时钟
     *    -# AMHW_HC32_SYSCLK_PLL :  内部 PLL 作为系统时钟
     */
    AMHW_HC32_SYSCLK_PLL,

    /**
     * \brief HCLK 分频系数，HCLK = SYSCLK / (2 ^ hclk_div)
     */
    0,

    /**
     * \brief PCLK 分频系数，PCLK = HCLK / (2 ^ pclk_div)
     */
    0,

    /** \brief 平台初始化函数，配置引脚等工作 */
    __hc32_clk_plfm_init,

    /** \brief 平台解初始化函数 */
    __hc32_clk_plfm_deinit,
};
```

可见，由于默认配置中，主时钟源选择 PLL，PLL 选择外部高速时钟。PLL 倍频系数为 6，因此系统时钟频率为 48MHz。

4.2.4 CRC

CRC 没有自定义参数需要配置，同时，也没有外部相关的引脚。因此，该外设完全不需要用户参与配置，实际需要使用时，调用其对应的实例初始化函数即可。

4.2.5 DAC

平台有两路 DAC，支持可配置的参考电压和精度、参考电压源等功能。以 DAC0 为例，其用户配置信息，详见 列表 4.28

列表 4.28 dac 设备配置信息

```
/** \brief DAC0 设备信息 */
static const am_hc32_dac_devinfo_t __g_dac0_devinfo = {
    HC32_DAC_BASE, /**< \brief 指向 DAC 寄存器块的指针 */

    8, /**< \brief DAC 转换精度 */

    AMHW_HC32F07X_DAC_CHAN_MASK_AVCC_VOLT, /**< \brief DAC 参考电压源 */

    3300, /**< \brief DAC 参考电压, 单位: mv*/

    AM_HC32_DAC_ALIGN_WAY_8_RIGHT, /**< \brief DAC 数据对齐方式 */

    __hc32_plfm_dac0_init, /**< \brief DAC 平台初始化函数 */

    __hc32_plfm_dac0_deinit, /**< \brief DAC 平台解初始化函数  */
};
```

用户可以根据实际情况修改转换精度、参考电压源值即可。其他值可以使用默认值。

注解：这里默认接外部参考电压源 3.3V，用户需要将外部参考电压接入 DAC 参考电压输入端。如用户修改为其他参考源需要修改正确的参考电压值。

4.2.6 DMA

DMA 没有自定义参数需要配置，同时，也没有外部相关的引脚。因此，该外设完全不需要用户参与配置，实际需要使用时，调用其对应的实例初始化函数即可。

4.2.7 GPIO

没有自定义参数需要配置，因此，该外设完全不需要用户参与配置，实际需要使用时，调用其对应的实例初始化函数即可。

4.2.8 I²C

平台有 2 个 I²C 总线接口，下面以 I²C1 为例讲述其配置内容。一般地，只需要配置 I²C 总线速率、超时时间和对应的 I²C 引脚即可。

1. I²C 总线速率

I²C 总线速率由设备配置文件 {HWCONFIG}**am_hwconf_hc32f07x_i2c.c** 中的 I²C1 设备信息中第 4 个参数配置，详见 列表 4.29 。默认为标准 I²C 速率，即 100KHz，如需修改为其它频率，直接修改对应的值即可

列表 4.29 I2C1 速率配置

```
/** \brief I2C1 设备信息 */
am_local am_const am_hc32_i2c_devinfo_t __g_hc32f07x_i2c1_devinfo = {
    HC32_I2C1_BASE,           /* I2C1 寄存器块基址 */
    CLK_I2C1,                   /* I2C1 时钟号 */
    INUM_I2C1,                  /* I2C1 中断号 */
    __BUS_SPEED_I2C1,           /* I2C1 总线速率 */
    20,                         /* 超时时间 */
    __hc32f07x_i2c1_bus_clean,    /* 总线恢复函数 */
```

```
    __hc32f07x_i2c1_plfm_init,    /* 平台初始化函数 */
    __hc32f07x_i2c1_plfm_deinit   /* 平台解初始化函数 */
};
```

2. 超时时间

由于 I²C 总线的特殊性，I²C 总线可能由于某种异常情况进入“死机”状态，为了避免该现象，I²C 驱动可以由用户提供一个超时时间，若 I²C 总线无任何响应的持续时间超过了超时时间，则 I²C 自动复位内部逻辑，以恢复正常状态。I²C 超时时间在 {HWCONFIG}**am_hwconf_hc32f07x_i2c.c** 设备信息中第 5 个参数设置，详见 列表 4.29 默认值为 10，即超时时间为 10ms。若有需要，可以将该值修改为其它值。例如：将其修改为 5，表示将超时时间设置为 5ms。

3. I²C 引脚

每个 I²C 都需要配置相应的引脚，包括时钟线 SCL 和数据线 SDA。I²C 引脚在平台初始化函数中完成。以 I²C1 为例，详见 列表 4.30 。

列表 4.30 I2C1 平台初始化函数——引脚配置

```
/**
 * \brief I2C1 平台初始化函数
 */
am_local void __hc32f07x_i2c1_plfm_init (void)
{
    am_gpio_pin_cfg(PIOB_10, PIOB_10_I2C1_SCL | PIOB_10_OUT_OD);
    am_gpio_pin_cfg(PIOB_11, PIOB_11_I2C1_SDA | PIOB_11_OUT_OD);

    am_clk_enable(CLK_I2C1);
}
```

可见，程序中，将 PIOB_10 配置为 I²C1 的时钟线，PIOB_11 配置为 I²C1 的数据线。可以直接使用 **am_gpio_pin_cfg()** 函数将一个引脚配置为相应的 I²C 功能。如果想使用其他引脚配置为 I²C 功能，则需要在这里修改。各 I²C 可使用的引脚详见表 表 4.3 ，其中 SCL 和 SDA 引脚必须成对使用，不可交叉使用。

表 4.3 I2C 引脚选择

I2C 总线接口	SCL 引脚	SDA 引脚
I2C0	PIOF_1	PIOF_0
I2C0	PIOA_9	PIOA_10
I2C0	PIOB_6	PIOB_7
I2C0	PIOB_8	PIOB_9
I2C1	PIOB_10	PIOB_11
I2C1	PIOB_13	PIOB_14
I2C1	PIOF_6	PIOF_7

4.2.9 I²C 从机

平台有的 2 个 I²C 总线接口也可以配置为从机使用。一般地，只需要配置对应的 I²C 引脚即可，配置方法可以参考 4.2.8 。

4.2.10 OPA

一般采用默认配置即可，如需修改其用户参数参考**\soc\hdsc\drivers\include\opa\hw\amhw_hc32f07x_opa.h** 文件中的参数定义。

4.2.11 RTC

RTC 作为一种实时时钟，可以提供一种实时时间服务 (包括时间、日期)，需要配置的仅有时钟源配置，可选时钟源有 6 个，详见 表 4.4 。

表 4.4 RTC 可选时钟源

RTC 时钟来相关枚举	含义
AMHW_HC32_RTC_CLK_SRC_XTL_32768Hz	外部低速晶振 XTL
AMHW_HC32_RTC_CLK_SRC_RCL_32KHz	内部低速时钟
AMHW_HC32_RTC_CLK_SRC_XTH_4MHz_DIV128	外部 4M 晶振
AMHW_HC32_RTC_CLK_SRC_XTH_8MHz_DIV256	外部 8M 晶振
AMHW_HC32_RTC_CLK_SRC_XTH_16MHz_DIV512	外部 16M 晶振
AMHW_HC32_RTC_CLK_SRC_XTH_32MHz_DIV1024	外部 32M 晶振

RTC 设备参数配置，如 列表 4.31 所示。

列表 4.31 RTC 设备信息

```
/** \brief RTC 设备信息 */
const struct am_hc32_rtc_devinfo __g_rtc_devinfo = {

    /** \brief RTC 设备基地址 */
    HC32_RTC_BASE,

    /**< \brief RTC 中断号 */
    INUM_RTC,

    /** \brief 从 1970-1-1 0:0:0 开始计算 */
    1970,

    /** \brief 平台初始化函数 */
    __hc32_plfm_rtc_init,

    /** \brief 平台去初始化函数 */
    __hc32_plfm_rtc_deinit
};
```

在 **__hc32_plfm_rtc_init()** 平台初始化函数中, 会完成时钟始能等操作，如 列表 4.32 所示。

列表 4.32 RTC 平台初始化

```
/** \brief RTC 平台初始化 */
void __hc32_plfm_rtc_init()
{
    /* 外部晶振驱动能力选择 */
    amhw_hc32f07x_rcc_xtl_xtal_driver_set(AMHW_HC32F07X_XTL_XTAL_DRIVER_
↪DEFAULT);

    /* XTL 晶振振荡幅度的调整 */
    amhw_hc32f07x_rcc_xtl_xtal_amplitude_set(
```

```
    AMHW_HC32F07X_XTL_XTAL_AMPLITUDE_BIG_DEFAULT);

    /* 外部高速时钟 XTL 稳定时间选择 */
    amhw_hc32f07x_rcc_xtl_waittime_set(AMHW_HC32F07X_XTL_WAITTIME_16384);

    amhw_hc32f07x_rcc_set_start(0x5A5A);
    amhw_hc32f07x_rcc_set_start(0xA5A5);
    amhw_hc32f07x_rcc_xtl_enable();

    /* 等待稳定 */
    while(amhw_hc32f07x_rcc_xtl_state_get() == AM_FALSE);

    am_clk_enable(CLK_RTC);

    /* RTC 时钟源选择 */
    amhw_hc32_rtc_clk_src_sel(HC32_RTC, AMHW_HC32_RTC_CLK_SRC_XTL_32768Hz);
};
```

4.2.12 LVD

HC32F07X 提供了 LVD 低压检测模块可以用于监测芯片管脚电压，其特性有：多路监测源、16 阶阈值电压、8 种触发条件、2 种触发结果等等。所以用户需要对这些参数进行配置。其用户配置文件详见 列表 4.33

列表 4.33 LVD 设备配置信息

```
/** \brief LVD 设备信息 */
static const am_hc32_lvd_devinfo_t __g_lvd_devinfo =
{
    /**< \brief 指向 LVD 寄存器块的指针 */
    HC32_LVD_BASE,

    /**< \brief LVD 中断编号 */
    INUM_LVD,

    /**
     * \brief LVD 触发条件
     *
     * \note 八种触发条件，如需要选择多个触发条件，
     *       例：AMHW_HC32_LVD_FTEN | AMHW_HC32_LVD_RTEN。
     */
    AMHW_HC32_LVD_RTEN,

    /**
     * \brief LVD 数字滤波时间，用户配置值参考枚举定义：LVD 数字滤波时间
     */
    AMHW_HC32_LVD_DEB_TIME_7P2_MS,

    /**
     * \brief LVD 阈值电压，用户配置值参考枚举定义：LVD 阈值电压
     */
    AMHW_HC32_LVD_VIDS_2P5_V,

    /**
     * \brief LVD 监测来源，用户配置值参考宏定义：LVD 监测来源选择掩码
     */
    AMHW_HC32_LVD_SRC_AVCC,

    /**
     * \brief LVD 触发动作，用户配置值参考宏定义：LVD 触发动作选择掩码
     *
```

```
     * \note 此配置位只能选择为中断或者复位触发动作
     */
    AMHW_HC32_LVD_TRI_ACT_SYSTEM_RESET,

    /**< \brief LVD 平台初始化函数 */
    __hc32_plfm_lvd_init,

    /**< \brief LVD 平台解初始化函数  */
    __hc32_plfm_lvd_deinit,
};
```

LVD 用于监测电压输入源低压与阈值电压的大小，所以首先用户需要配置 LVD 监测来源然后选择阈值电压。LVD 数字滤波时间可以采用默认配置即可，最后需要设置 LVD 的动作方式。LVD 提供了两种：复位和中断，可以根据程序设计具体情况进行设置，这里需要用户参考配置触发条件。

4.2.13 SPI

平台有 2 个 SPI 总线接口，定义为 SPI0、SPI1。SPI 可以选择中断方式、DMA 传输方式和轮询方式，这三种种方式的配置略有不同，同时 SPI 可以配置成主机或者从机模式。因此，平台分别提供了不同方式的配置文件。

1. 主机中断方式

以 SPI1 为例，如果使用 SPI 的中断方式，需要在 {HWCONFIG}**\am_hwconf_hc32f07x_spi_int.c** 文件中进行 SPI1 相关引脚的设置，需要设置的引脚仅有 SCK、MOSI 和 MISO，片选引脚无需设置，因为在使用 SPI 标准接口层函数（参见：**\ametal\interface\am_spi.h**）时，片选引脚可以作为参数任意设置。SPI 相关引脚的设置详见 列表 4.34 。

列表 4.34 SPI1 平台初始化函数

```
/** \brief SPI1 平台初始化 */
static void __hc32_plfm_spi1_int_init (void)
{
    am_gpio_pin_cfg(PIOB_10, PIOB_10_SPI1_SCK  | PIOB_10_OUT_PP);
    am_gpio_pin_cfg(PIOB_15, PIOB_15_SPI1_MOSI | PIOB_15_OUT_PP);
    am_gpio_pin_cfg(PIOC_2,  PIOC_2_SPI1_MISO  | PIOC_2_INPUT_PU);

    am_clk_enable(CLK_SPI1);
}
```

可见这里 PIOB_10 设置为 SCK，PIOC_2 设置为 MISO，PIOB_10 设置为 MOSI。如果你需要设置别的引脚的话，需要在这里更改，可用引脚配置详见 表 4.5 。

表 4.5 SPI 可选引脚

SPI 总线接口	SCK	MISO	MOSI
SPI0	PIOA_5	PIOA_6	PIOA_7
SPI0	PIOE_13	PIOE_14	PIOE_15
SPI0	PIOB_3	PIOB_4	PIOB_5
SPI1	PIOB_10	PIOC_2	PIOC_3
SPI1	PIOB_13	PIOB_14	PIOB_15
SPI1	PIOD_1	PIOD_3	PIOD_4

设备信息结构体主要配置 MOSI 引脚，详见 列表 4.35 。

列表 4.35 SPI1 主机中断传输方式设备信息结构体定义

```
/** \brief SPI1 设备信息 */
const  struct am_hc32_spi_int_devinfo  __g_spi1_int_devinfo = {
    HC32_SPI1_BASE,                          /**< \brief SPI1 寄存器指针 */
    CLK_SPI1,                                /**< \brief 时钟 ID 号 */
    INUM_SPI1_I2S1,                          /**< \brief SPI1 中断号 */
    PIOB_15_SPI1_MOSI | PIOB_15_OUT_PP,      /**< \brief SPI1 配置标识 */
    PIOB_15,                                 /**< \brief MOSI 引脚号 */
    __hc32_plfm_spi1_int_init,               /**< \brief SPI1 平台初始化函数 */
    __hc32_plfm_spi1_int_deinit              /**< \brief SPI1 平台解初始化函数 */
};
```

2. 主机 DMA 传输方式

如果需要使用 SPI 的 DMA 方式，需要在 {HWCONFIG}**\am_hwconf_hc32f07x_spi_dma.c** 文件中进行 SPI 引脚的配置，引脚配置方法与中断方式配置引脚的方法完全一样，在此不再赘述。

而设备信息有不同，需要增加配置 DMA 的发送和接收通道，DMA 的通道信息可参考 **\ametal\soc\hdsc\hc32f07x\hc32f07x_dma_chan.h** ，设备配置详见 列表 4.36 。

列表 4.36 SPI1 主机 DMA 传输方式设备信息结构体定义

```
/** \brief SPI1 设备信息 */
static const  struct am_hc32_spi_dma_devinfo  __g_spi1_dma_devinfo = {
    HC32_SPI1_BASE,                         /**< \brief SPI1 寄存器指针 */
    CLK_SPI1,                                 /**< \brief 时钟 ID 号 */
    INUM_SPI1_I2S1,                                /**< \brief SPI1 中断号 */
    DMA_CHAN_1,
    DMA_CHAN_2,
    __hc32_plfm_spi1_dma_init,              /**< \brief SPI1 平台初始化函数 */
    __hc32_plfm_spi1_dma_deinit             /**< \brief SPI1 平台解初始化函数 */
};
```

3. 主机轮询传输方式

如果需要使用 SPI 的 poll 方式，需要在 {HWCONFIG}**\am_hwconf_hc32f07x_spi_poll.c** 文件中进行 SPI 引脚的配置，引脚配置方法与中断方式配置引脚的方法完全一样，设备信息配置也和中断方式配置完全一样，在此不再赘述。

4. 从机 DMA 传输方式

如果需要使用 SPI 的从机 DMA 方式，需要在 {HWCONFIG}**\am_hwconf_hc32f07x_spi_slv_dma.c** 文件中进行 SPI 引脚的配置，引脚配置方法与中断方式配置引脚的方法完全一样，在此不再赘述。

同主机类似从机的 DMA 传输方式需要配置设备信息中的 DMA 发送和接收通道，同时需要配置 NSS 引脚。从机 DMA 传输方式设备信息详见 列表 4.37 。

列表 4.37 SPI1 从机 DMA 方式传输设备信息结构体定义

```
/**
 * \brief SPI1 设备信息
 */
const  struct am_hc32_spi_poll_devinfo  __g_spi1_poll_devinfo = {
    HC32_SPI1_BASE,                        /**< \brief SPI1 寄存器指针 */
    CLK_SPI1,                                /**< \brief 时钟 ID 号 */
    __hc32_plfm_spi1_poll_init,            /**< \brief SPI1 平台初始化函数 */
```

```
    __hc32_plfm_spi1_poll_deinit          /**< \brief SPI1 平台解初始化函数 */
};
```

4.2.14 Timer

hc32f07x 有 4 个 16 位通用定时器、3 个高级 PWM 定时器。可以实现定时、捕获、和 PWM 输出功能。用户可以直接使用 AMetal 抽象好的 PWM、CAP、Timing 标准接口服务，也可以直接调用驱动层提供的相关接口操作 Timer 的一些功能。由于 Timer 抽象出来的标准服务功能各不相同，在初始化时，就需要确定将 Timer 用于何种功能，不同功能对应的设备信息存在不同，这就使得 Timer 部分对应了几套设备配置文件，使用 Timer 的何种功能，就使用对应的配置文件。下面以 TIM1 为例，讲解其配置内容。

1. 使用标准捕获 (CAP) 服务

在该模式下，对应的配置文件为 {HWCONFIG}**am_hwconf_hc32f07x_tim_cap.c** ，TIM0、TIM1、TIM2 、TIM4、TIM5 和 TIM6 都支持 2 路捕获通道，TIM3 支持 6 路捕获通道。实际使用到的通道数目可以在配置文件中的设备信息中修改。详见 列表 4.38。

列表 4.38 TIM1 用于捕获功能的设备信息

```
/** \brief TIM1 用于捕获功能的设备信息 */
const am_hc32_tim_cap_devinfo_t  __g_tim1_cap_devinfo = {
    HC32_TIM1_BASE,                     /**< \brief TIM1 寄存器块的基地址 */
    INUM_TIM1,                          /**< \brief TIM1 中断编号 */
    AMHW_HC32_TIM_CLK_DIV1,             /**< \brief 时钟分频系数 */
    2,                                  /**< \brief 2 个捕获通道 */
    AMHW_HC32_TIM_TYPE_TIM1,            /**< \brief 定时器选择 */

    &__g_tim1_cap_ioinfo_list[0],
    __hc32f07x_plfm_tim1_cap_init,      /**< \brief 平台初始化函数 */
    __hc32f07x_plfm_tim1_cap_deinit     /**< \brief 平台解初始化函数 */
};
```

使用捕获功能时，每个捕获通道都需要设置一个对应的引脚，相关引脚信息由设备信息文件中的 **__g_tim1_cap_ioinfo_list** 数组定义，详见 列表 4.39 。

列表 4.39 TIM1_CAP 各捕获通道相关引脚设置

```
/** \brief TIM1 用于捕获功能的引脚配置信息列表 */
am_hc32_tim_cap_ioinfo_t __g_tim1_cap_ioinfo_list[] = {
    /**< \brief 通道 0 */
    {
        AM_HC32_TIM_CAP_CH0A,
        PIOA_0,
        PIOA_0_TIM1_CHA  | PIOA_0_INPUT_FLOAT,
        PIOA_0_GPIO  | PIOA_0_INPUT_PU
    },

    /**< \brief 通道 1 */
    {
        AM_HC32_TIM_CAP_CH0B,
        PIOA_1,
        PIOA_1_TIM1_CHB  | PIOA_1_INPUT_FLOAT,
        PIOA_1_GPIO  | PIOA_1_INPUT_PU
    },
};
```

每个数组元素对应了一个捕获通道，0 号元素对应通道 0，1 号元素对应通道 1，

以此类推。数组元素的类型为 **am_zlg_tim_cap_ioinfo_t** ，该类型在对应驱动头文件 **\ametal\soc\hdsc\drivers\include\tim\am_hc32_tim_cap.h** 中定义详见 列表 4.40 。

列表 4.40 TIM_CAP 通道引脚信息结构体类型

```
/**
 * \brief TIM 捕获功能相关的 GPIO 信息
 */
typedef struct am_hc32_tim_cap_ioinfo {
    int8_t   channel;       /**< \brief CAP 所使用的通道标识符 */
    uint32_t gpio;          /**< \brief 对应的 GPIO 管脚 */
    uint32_t func;          /**< \brief 为捕获功能时的 GPIO 功能设置 */
    uint32_t dfunc;         /**< \brief 禁能管脚捕获功能时的默认 GPIO 功能设置 */
} am_hc32_tim_cap_ioinfo_t;
```

2. 使用标准 PWM 服务

在该模式下，设置与上述基本相同，此处不再赘述，此处要注意的是 PWM 的模式与输出电平，用户可以根据自己的需求设置。对应的配置文件为 {HWCONFIG}**\am_hwconf_hc32f07x_tim_pwm.c** ，TIM1 用于 PWM 的设备信息详见 列表 4.41 。

列表 4.41 TIM1 用于 PWM 功能的设备信息

```
/** \brief TIM1 用于 PWM 设备信息 */
const am_hc32_tim_pwm_devinfo_t  __g_tim1_pwm_devinfo = {
    HC32_TIM1_BASE,                             /**< \brief TIM1 寄存器块的基地址␣
↪*/
    AM_NELEMENTS(__g_tim1_pwm_chaninfo_list), /**< \brief 配置输出通道个数 */
    0,                                          /**< \brief 互补 PWM 选择
                                                 *          1: 互补 PWM
                                                 *          0: 独立 PWM
                                                 */
    AM_HC32_TIM_PWM_OCPOLARITY_HIGH,            /**< \brief 脉宽极性 */
    &__g_tim1_pwm_chaninfo_list[0],             /**< \brief 通道配置信息列表  */
    AMHW_HC32_TIM_TYPE_TIM1,                    /**< \brief 定时器类型 */
    __hc32f07x_plfm_tim1_pwm_init,              /**< \brief 平台初始化函数 */
    __hc32f07x_plfm_tim1_pwm_deinit             /**< \brief 平台解初始化函数 */
};
```

3. 使用标准定时器服务

在该模式下，对应的配置文件为 {HWCONFIG}**\am_hwconf_hc32f07x_tim_timing.c** ，由于用作定时器时不需要配置定时器位数，也无外部相关引脚，所有不需要用户配置该配置文件，实际需要使用时，调用其对应的实例初始化函数即可。

4.2.15 TRNG

TRNG 没有自定义参数需要配置，同时，也没有外部相关的引脚。因此，该外设完全不需要用户参与配置，实际需要使用时，调用其对应的实例初始化函数即可。

4.2.16 USART

平台有 4 个 USART 接口，定义为 USART0、USART1、USART2、USART3。在 AMetal 软件包提供的驱动中实现了 USART 功能。对于用户来讲，一般只需要配置串口的相关引脚即可。特别地，可能需要配置串口的输入时钟频率。下面以 USART0 为例，讲解其配置内容。

注意：串口波特率、数据位、停止位、校验位等的设置应直接使用 UART 标准接口层相关函数配置，详见 UART 标准接口文件 **\ametal\interface\am_uart.h** 。

1. 引脚配置

USART0 相关的引脚在配置文件 {HWCONFIG}**\am_hwconf_hc32f07x_usart.c** 文件中配置，详见 列表 4.42 。

列表 4.42 USART 引脚初始化函数

```
/** \brief 串口 0 平台初始化 */
static void __hc32_plfm_uart0_init (void)
{
    am_clk_enable(CLK_UART0);

    am_gpio_pin_cfg(PIOA_9,  PIOA_9_UART0_TXD  | PIOA_9_OUT_PP );
    am_gpio_pin_cfg(PIOA_10, PIOA_10_UART0_RXD | PIOA_10_INPUT_FLOAT);
}
```

程序中将 PIOA_9 作为 USART0 的发送引脚，PIOA_10 作为 USART0 的接收引脚。用户也可以选择其他引脚作为串口的发送，接受引脚，可用串口引脚详见 表 4.6 ，引脚选用必须成对选用，不可交叉选用。

表 4.6 USART 引脚选择示例

USART 接口	TXD 功能引脚	RXD 功能引脚
USART0	PIOA_9	PIOA_10
USART1	PIOA_2	PIOA_3
USART2	PIOC_3	PIOC_2
USART3	PIOC_7	PIOC_6

2. 配置波特率

在 USART0 设备配置信息中可以设置串口位数、奇偶校验、停止位、波特率等信息，串口默认设置为 8 位数据，无奇偶校验，1 个停止位，波特率为 115200，详见 列表 4.43 。

列表 4.43 USART 设备信息结构体类型

```
/** \brief 串口 0 设备信息 */
static const am_hc32_uart_devinfo_t __g_uart0_devinfo = {

    HC32_UART0_BASE,                /**< \brief 串口 0 */
    INUM_UART0_2,                   /**< \brief 串口 0 的中断编号 */
    CLK_UART0,                      /**< \brief 串口时钟 ID */

    AMHW_HC32_UART_PARITY_NO |      /**< \brief 无极性 */
    AMHW_HC32_UART_STOP_1_0_BIT,    /**< \brief 1 个停止位 */

    AM_FALSE,                       /**< \brief 不使用异步半双工（单线）模式 */

    115200,                         /**< \brief 设置的波特率 */

    0,                              /**< \brief 无其他中断 */

    NULL,                           /**< \brief 使用 RS485 */
    __hc32_plfm_uart0_init,         /**< \brief UART0 的平台初始化 */
```

```
    __hc32_plfm_uart0_deinit,       /**< \brief UART0 的平台去初始化 */
};
```

4.2.17 WDT

WDT 没有自定义参数需要配置，同时，也没有外部相关的引脚。因此，该外设完全不需要用户参与配置，实际需要使用时，调用其对应的实例初始化函数即可。

4.2.18 CAN

此芯片只有一个 CAN 总线接口，所以无需修改引脚和中断相关的信息。主要的配置信息为 can 初始配置信息、滤波器参数配置、标识符设置和波特率设置。

1. CAN 初始配置信息

初始配置信息中包括 CAN 工作模式设置，时间触发通信模式、缓冲器上溢、缓冲器将满设定值、错误警告限定值和 FIFO 的相关设置。详见 列表 4.44

列表 4.44 CAN 初始化配置信息结构体类型

```
/** \brief CAN 初始化配置信息 */
am_hc32f07x_can_init_info_t __g_can_initfo = {

    /** \brief 0 普通模式   1 存储所有 */
    AMHW_HC32F07X_CAN_RXBUF_NORMAL,

    /** \brief 时间触发通信模式  0 非触发  1 触发 */
     0,

    /** \brief 接收缓冲器上溢模式
     *   0 最早接收的数据被覆盖
     *   1 最新接收到的数据不被存储
     */
    AMHW_HC32F07X_CAN_RXBUF_NOTSTORED,

    /** \brief 缓冲器发送模式选择
     *   0 FIFO 模式
     *   1 优先级模式
     */
    AMHW_HC32F07X_CAN_STB_MODE_FIFO,

    /** \brief 接收缓冲器将满设定值 */
    10,
    /** \brief 错误警告限定值 */
    15,
};
```

2. 设备配置信息

CAN 设备配置信息详见 列表 4.45

列表 4.45 CAN 设备配置信息结构体类型

```
/** \brief 设备信息 */
static const am_hc32f07x_can_devinfo_t __g_can_devinfo = {

    HC32F07X_CAN_BASE,                  /**< \brief CAN */
    INUM_CAN,              /**< \brief CAN 的中断编号  此处不可修改 */
    AMHW_HC32F07X_CAN_PTB,
    __g_can_intcb_info,              /**< \brief 回调信息 */
```

```
    MAX,                                /**< \brief 回调信息内存大小 */
    __hc32f072_plfm_can_init,           /**< \brief CAN 的平台初始化 */
    __hc32f072_plfm_can_deinit,         /**< \brief CAN 的平台去初始化 */
    &__g_can_initfo

};
```

3. 波特率设置

此平台 CAN 总线最大支持 1Mbps 的波特率。波特率设置时，用户通过修改相关宏参数，直接调用接口即可。详见 {EXAMPLES} **\board\amf07x_core\can\demo_hc32f07x_std_can.c** 中。

4.2.19 USB

平台有一个 USB 接口，支持设备模式，支持 USB2.0 全速传输。

1. 设备配置信息

USB 设备配置信息详见 列表 4.46

列表 4.46 USB 设备配置信息结构体类型

```
/**< \brief 定义 USB 设备信息 */
static const am_hc32f07x_usbd_devinfo_t  __g_hc32f07x_usbd_mouse_info = {
    HC32F07X_USB_BASE,              /**< \brief 寄存器基地址 */
    INUM_USB,                       /**< \brief 中断号 */
    __usb_mouse_init,               /**< \brief 平台初始化 */
    __usb_mouse_deinit,             /**< \brief 平台去初始化 */
    &__g_usbd_mouse_info,
};
```

2. 初始化信息

平台初始化函数需要对模块时钟进行配置。详见 列表 4.47

列表 4.47 USB 平台初始化

```
/**
 * \brief 平台初始化
 */
static void __usb_mouse_init (void) {
    /* 使能时钟 */
    amhw_hc32f07x_rcc_set_start(0x5A5A);
    amhw_hc32f07x_rcc_set_start(0xA5A5);
    amhw_hc32f07x_rcc_usbclk_adjust_set (AMHW_HC32F07X_USBCLK_ADJUST_PLL);
    am_clk_enable(CLK_USB);
}
```

USBFS 所使用的时钟有 PHY 时钟和系统，其中时钟和系统，其中 PHY 时钟频率需要配置为 48MHz，该 48MHz 时钟可由内部 PLL 电路产生，也可由内部高速 48M RC 时钟产生，使用 USBFS 模块前，需要在系统控制器内配置好 USBFS 时钟。

AMetal 平台里面现在除了上面的外设资源配置文件外，在 {HWCONFIG} 配置文件夹有可能还会有 **am_hwconf_microport_ds1302.c** 、**am_hwconf_miniport_view_key.c** 等这些配置文件，这些配置文件用法可以参考该源文件实现，里面有详细的注释。它们属于 AMetal 拓展进阶部分，如果想进一步了解这一部分的实现及用法，用户请参考《面向 AMetal 框架与接口的编程》一书，现在该书正在我司官方淘宝店火热销售中。

4.3 使用方法

使用外设资源的方法有两种，一种是使用软件包提供的驱动，一种是不使用驱动，自行使用硬件层提供的函数完成相关操作。

4.3.1 使用 AMetal 软件包提供的驱动

一般来讲，除非必要，一般都会优先选择使用经过测试验证的驱动完成相关的操作。使用外设的操作顺序一般是初始化、使用相应的接口函数操作该外设、解初始化

4.3.1.1 初始化

无论何种外设，在使用前均需初始化。所有外设的初始化操作均只需调用用户配置文件中提供的设备实例初始化函数即可。

所有外设的实例初始化函数均在 {PROJECT}**\user_config\am_hc32f07x_inst_init.h** 文件中声明。使用实例初始化函数前，应确保已包含 **am_hc32f07x_inst_init.h** 头文件。片上外设对应的设备实例初始化函数的原型详见 表 4.7 。

表 4.7 片上外设及对应的实例初始化函数

序号	外设	实例初始化函数原型
1	NVIC 中断	int am_hc32_nvic_inst_init (void);
2	ADC	am_adc_handle_t am_hc32_adc_inst_init (void);
3	CLK	int am_hc32_clk_inst_init (void);
4	CRC	am_crc_handle_t am_hc32_crc_inst_init (void);
5	DAC1	am_dac_handle_t am_hc32_dac0_inst_init (void);
6	DAC2	am_dac_handle_t am_hc32_dac1_inst_init (void);
7	DMA	int am_hc32_dma_inst_init (void);
8	GPIO	int am_hc32_gpio_inst_init (void);
9	I2C0	am_i2c_handle_t am_hc32_i2c0_inst_init (void);
10	I2C1	am_i2c_handle_t am_hc32_i2c1_inst_init (void);
11	I2C0_SLV	am_i2c_slv_handle_t am_hc32_i2c0_slv_inst_init (void);
12	I2C1_SLV	am_i2c_slv_handle_t am_hc32_i2c1_slv_inst_init (void);
13	WDT	am_wdt_handle_t am_hc32_wdt_inst_init (void);
15	RTC	am_rtc_handle_t am_hc32_rtc_inst_init (void);
16	SPI0(DMA 方式)	am_spi_handle_t am_hc32_spi0_dma_inst_init (void);
17	SPI1(DMA 方式)	am_spi_handle_t am_hc32_spi1_dma_inst_init (void);
18	SPI0(中断方式)	am_spi_handle_t am_hc32_spi0_int_inst_init (void);
19	SPI1(中断方式)	am_spi_handle_t am_hc32_spi1_int_inst_init (void);
20	SPI0(轮询方式)	am_spi_handle_t am_hc32_spi0_poll_inst_init (void);
21	SPI1(轮询方式)	am_spi_handle_t am_hc32_spi1_poll_inst_init (void);
22	SYSTICK	am_timer_handle_t am_hc32_systick_inst_init (void);
23	TIM0_CAP	am_cap_handle_t am_hc32_tim0_cap_inst_init (void);

下页继续

表 4.7 – 续上页

序号	外设	实例初始化函数原型
24	TIM1_CAP	am_cap_handle_t am_hc32_tim1_cap_inst_init (void);
25	TIM2_CAP	am_cap_handle_t am_hc32_tim2_cap_inst_init (void);
26	TIM3_CAP	am_cap_handle_t am_hc32_tim3_cap_inst_init (void);
27	TIM4_CAP	am_cap_handle_t am_hc32_tim4_cap_inst_init (void);
28	TIM5_CAP	am_cap_handle_t am_hc32_tim5_cap_inst_init (void);
29	TIM6_CAP	am_cap_handle_t am_hc32_tim6_cap_inst_init (void);
30	TIM0_PWM	am_pwm_handle_t am_hc32_tim0_pwm_inst_init (void);
31	TIM1_PWM	am_pwm_handle_t am_hc32_tim1_pwm_inst_init (void);
32	TIM2_PWM	am_pwm_handle_t am_hc32_tim2_pwm_inst_init (void);
33	TIM3_PWM	am_pwm_handle_t am_hc32_tim3_pwm_inst_init (void);
34	TIM4_PWM	am_pwm_handle_t am_hc32_tim4_pwm_inst_init (void);
35	TIM5_PWM	am_pwm_handle_t am_hc32_tim5_pwm_inst_init (void);
36	TIM6_PWM	am_pwm_handle_t am_hc32_tim6_pwm_inst_init (void);
37	TIM0_TIMING	am_timer_handle_t am_hc32_tim0_timing_inst_init (void);
38	TIM1_TIMING	am_timer_handle_t am_hc32_tim1_timing_inst_init (void);
39	TIM2_TIMING	am_timer_handle_t am_hc32_tim2_timing_inst_init (void);
40	TIM3_TIMING	am_timer_handle_t am_hc32_tim3_timing_inst_init (void);
41	TIM4_TIMING	am_timer_handle_t am_hc32_tim4_timing_inst_init (void);
42	TIM5_TIMING	am_timer_handle_t am_hc32_tim5_timing_inst_init (void);
43	TIM6_TIMING	am_timer_handle_t am_hc32_tim6_timing_inst_init (void);
44	USART0	am_uart_handle_t am_hc32_uart0_inst_init (void);
45	USART1	am_uart_handle_t am_hc32_uart1_inst_init (void);
46	USART2	am_uart_handle_t am_hc32_uart2_inst_init (void);
47	USART3	am_uart_handle_t am_hc32_uart3_inst_init (void);
48	USB	am_usbd_dev_t am_hc32f07x_usbd_mouse_inst_init (void);
49	CAN	am_can_handle_t am_hc32f07x_can_inst_init (void);
50	LPUART0	am_uart_handle_t am_hc32_lpuart0_inst_init (void);
51	LPUART1	am_uart_handle_t am_hc32_lpuart1_inst_init (void);
52	PCA	am_hc32_pca_handle_t am_hc32_pca_inst_init (void);

4.3.1.2 操作外设

根据实例初始化函数的返回值类型，可以判断后续该如何继续操作该外设。实例初始化函数的返回值可能有以下三类：

- int 型；
- 标准服务 handle 类型（**am_*_handle_t**，类型由标准接口层定义），如 **am_adc_handle_t**；
- 驱动自定义 handle 类型（**am_hc32_*_handle_t**，类型由驱动头文件自定义），如

am_hc32_pca_handle_t 。

下面分别介绍这三种不同返回值的含义以及实例初始化后，该如何继续使用该外设。

1. 返回值为 int 型

常见的全局资源外设对应的实例初始化函数的返回值均为 int 类型。相关外设详见 表 4.8 。

表 4.8 返回值为 int 类型的实例初始化函数

序号	外设	实例初始化函数原型
1	CLK	int am_hc32_clk_inst_init (void);
2	DMA	int am_hc32_dma_inst_init (void);
3	GPIO	int am_hc32_gpio_inst_init (void);
4	NVIC 中断	int am_hc32_nvic_inst_init (void);

若返回值为 **AM_OK** ，表明实例初始化成功；否则，表明实例初始化失败，需要检查设备相关的配置信息。

后续操作该类外设直接使用相关的接口操作即可，根据接口是否标准化，可以将操作该外设的接口分为两类。

接口已标准化，如 GPIO 提供了标准接口，在 **\ametal\interface\am_gpio.h** 文件中声明。则可以查看相关接口说明和示例，以使用 GPIO。简单示例如 列表 4.48 。

列表 4.48 GPIO 标准接口使用范例

```
1  am_gpio_pin_cfg(PIOA_0, AM_GPIO_OUTPUT_INIT_HIGH);   /* 将 GPIO 配置为输出模式并且为高电平 */
```

参见:

接口原型及详细的使用方法请参考 **ametal\documents\《AMetal API 参考手册 V1.10.chm》** 或者 **\ametal\interface\am_gpio.h** 文件。

接口未标准化，则相关接口由驱动头文件自行提供，如 DMA，相关接口在 **\ametal\soc\hdsc\drivers\include\dma\am_hc32_dma.h** 文件中声明。

注意： 无论是标准接口还是非标准接口，使用前，均需要包含对应的接口头文件。需要特别注意的是，这些全局资源相关的外设设备，一般在系统启动时已默认完成初始化，无需用户再自行初始化。详见 表 3.1 。

2. 返回值为标准服务句柄

有些外设实例初始化函数后返回的是标准服务句柄，相关外设详见 表 4.9 。可以看到，绝大部分外设实例初始化函数，均是返回标准的服务句柄。若返回值不为 **NULL** ，表明初始化成功；否则，初始化失败，需要检查设备相关的配置信息。

表 4.9 返回值为标准服务句柄的实例初始化函数

序号	外设	实例初始化函数原型
1	ADC	am_adc_handle_t am_hc32_adc_inst_init (void);
2	CRC	am_crc_handle_t am_hc32_crc_inst_init (void);
3	DAC0	am_dac_handle_t am_hc32_dac0_inst_init (void);
4	DAC1	am_dac_handle_t am_hc32_dac1_inst_init (void);
5	I2C0	am_i2c_handle_t am_hc32_i2c0_inst_init (void);
6	I2C1	am_i2c_handle_t am_hc32_i2c1_inst_init (void);
7	I2C0_SLV	am_i2c_slv_handle_t am_hc32_i2c0_slv_inst_init (void);
8	I2C1_SLV	am_i2c_slv_handle_t am_hc32_i2c1_slv_inst_init (void);
9	WDT	am_wdt_handle_t am_hc32_wdt_inst_init (void);
10	RTC	am_rtc_handle_t am_hc32_rtc_inst_init (void);
11	SPI0(DMA 方式)	am_spi_handle_t am_hc32_spi0_dma_inst_init (void);
12	SPI1(DMA 方式)	am_spi_handle_t am_hc32_spi1_dma_inst_init (void);
13	SPI0(中断方式)	am_spi_handle_t am_hc32_spi0_int_inst_init (void);
14	SPI1(中断方式)	am_spi_handle_t am_hc32_spi1_int_inst_init (void);
15	SPI0(轮询方式)	am_spi_handle_t am_hc32_spi0_poll_inst_init (void);
16	SPI1(轮询方式)	am_spi_handle_t am_hc32_spi1_poll_inst_init (void);
17	SYSTICK	am_timer_handle_t am_hc32_systick_inst_init (void);
18	TIM0_CAP	am_cap_handle_t am_hc32_tim0_cap_inst_init (void);
19	TIM1_CAP	am_cap_handle_t am_hc32_tim1_cap_inst_init (void);
20	TIM2_CAP	am_cap_handle_t am_hc32_tim2_cap_inst_init (void);
21	TIM3_CAP	am_cap_handle_t am_hc32_tim3_cap_inst_init (void);
22	TIM4_CAP	am_cap_handle_t am_hc32_tim4_cap_inst_init (void);
23	TIM5_CAP	am_cap_handle_t am_hc32_tim5_cap_inst_init (void);
24	TIM6_CAP	am_cap_handle_t am_hc32_tim6_cap_inst_init (void);
25	TIM0_PWM	am_pwm_handle_t am_hc32_tim0_pwm_inst_init (void);
26	TIM1_PWM	am_pwm_handle_t am_hc32_tim1_pwm_inst_init (void);
27	TIM2_PWM	am_pwm_handle_t am_hc32_tim2_pwm_inst_init (void);
28	TIM3_PWM	am_pwm_handle_t am_hc32_tim3_pwm_inst_init (void);
29	TIM4_PWM	am_pwm_handle_t am_hc32_tim4_pwm_inst_init (void);
30	TIM5_PWM	am_pwm_handle_t am_hc32_tim5_pwm_inst_init (void);
31	TIM6_PWM	am_pwm_handle_t am_hc32_tim6_pwm_inst_init (void);
32	TIM0_TIMING	am_timer_handle_t am_hc32_tim0_timing_inst_init (void);
33	TIM1_TIMING	am_timer_handle_t am_hc32_tim1_timing_inst_init (void);
34	TIM2_TIMING	am_timer_handle_t am_hc32_tim2_timing_inst_init (void);
35	TIM3_TIMING	am_timer_handle_t am_hc32_tim3_timing_inst_init (void);
36	TIM4_TIMING	am_timer_handle_t am_hc32_tim4_timing_inst_init (void);

下页继续

表 4.9 – 续上页

序号	外设	实例初始化函数原型
37	TIM5_TIMING	am_timer_handle_t am_hc32_tim5_timing_inst_init (void);
38	TIM6_TIMING	am_timer_handle_t am_hc32_tim6_timing_inst_init (void);
39	USART0	am_uart_handle_t am_hc32_uart0_inst_init (void);
40	USART1	am_uart_handle_t am_hc32_uart1_inst_init (void);
41	USART2	am_uart_handle_t am_hc32_uart2_inst_init (void);
42	USART3	am_uart_handle_t am_hc32_uart3_inst_init (void);
43	USB	am_usbd_dev_t am_hc32f07x_usbd_mouse_inst_init (void);
44	CAN	am_can_handle_t am_hc32f07x_can_inst_init (void);
45	LPUART0	am_uart_handle_t am_hc32_lpuart0_inst_init (void);
46	LPUART1	am_uart_handle_t am_hc32_lpuart1_inst_init (void);

对于这些外设，后续可以利用返回的 handle 来使用相应的标准接口层函数。使用标准接口层函数的相关代码是可跨平台复用的！

例如，ADC 设备的实例初始化函数的返回值类型为 **am_adc_handle_t**，为了方便后续使用，可以定义一个变量保存下该返回值，后续就可以使用该 handle 完成电压的采集了。详见 列表 4.49 。

列表 4.49 ADC 简单操作示例

```
#include "ametal.h"
#include "am_vdebug.h"
#include "am_board.h"
#include "am_hc32f07x.h"
#include "am_hc32_adc.h"
#include "am_hc32f07x_inst_init.h"

/**
 * \brief ADC 例程，通过标准接口实现
 */
int am_main (void)
{
    int             i;
    int             chan = 0;     /* 通道 0 */
    uint32_t        adc_mv[5];    /* 采样电压 */
    am_adc_handle_t adc_handle;  /* ADC 标准服务操作句柄 */

    adc_handle = am_hc32_adc_inst_init (); /* ADC 实例初始化并获取句柄值 */

    am_kprintf("The ADC STD HT Demo\r\n");

    while (1) {

        /* 获取 ADC 采集电压，采样完成才返回 */
        am_adc_read_mv(adc_handle, chan, adc_mv, 5);
        for (i = 1; i < 5; i++) {
            adc_mv[0] += adc_mv[i];
        }
        adc_mv[0] /= 5;
        am_kprintf("Vol: %d mv\r\n", adc_mv[0]);
    }
}
```

3. 返回值为组件定义服务句柄

这类外设功能较为特殊，AMetal 未对其进行标准服务的抽象，这时，使用该外设对应的实例初始化函数返回的即是驱动自定义的服务句柄，相关外设详见 表 4.10 。可见，仅 PCA 外设提供了一个该类型的实例初始化函数。

表 4.10 返回值为驱动自定义服务句柄的实例初始化函数

序号	外设	实例初始化函数原型
1	PCA	am_hc32_pca_handle_t am_hc32_pca_inst_init (void);

若返回值不为 **NULL** ，表明初始化成功；否则，初始化失败，需要检查设备相关的配置信息。这类外设初始化后，即可利用返回的 handle 去使用驱动提供的相关函数。

4.3.1.3 解初始化

外设使用完毕后，应该调用相应设备配置文件提供的设备实例解初始化函数，以释放相关资源。所有外设的实例解初始化函数均在 {PROJECT}**user_config\am_hc32f07x_inst_init.h** 文件中声明。使用实例解初始化函数前，应确保已包含 **am_hc32f07x_inst_init.h** 头文件。各个外设对应的设备实例解初始化函数的原型详见 表 4.11 。

表 4.11 片上外设及对应的实例解初始化函数

序号	外设	实例初始化函数原型
1	NVIC 中断	void am_hc32_nvic_inst_deinit (void);
2	ADC	void am_hc32_adc_inst_deinit (am_adc_handle_t handle);
3	CRC	void am_hc32_crc_inst_deinit (am_crc_handle_t handle);
5	DAC0	void am_hc32_dac0_inst_deinit (am_dac_handle_t handle);
6	DAC1	void am_hc32_dac1_inst_deinit (am_dac_handle_t handle);
7	DMA	void am_hc32_dma_inst_deinit (void);
8	GPIO	void am_hc32_gpio_inst_deinit (void);
9	I2C0	void am_hc32_i2c0_inst_deinit (am_i2c_handle_t handle);
10	I2C1	void am_hc32_i2c1_inst_deinit (am_i2c_handle_t handle);
11	I2C0_SLV	void am_hc32_i2c0_slv_inst_deinit (am_i2c_slv_handle_t handle);
12	I2C1_SLV	void am_hc32_i2c1_slv_inst_deinit (am_i2c_slv_handle_t handle);
13	WDT	void am_hc32_wdt_inst_deinit (am_wdt_handle_t handle);
14	PCA	void am_hc32_pca_inst_deinit (am_hc32_pca_handle_t handle);
15	RTC	void am_hc32_rtc_inst_deinit (am_rtc_handle_t handle);
16	SPI0(DMA 方式)	void am_hc32_spi0_dma_inst_deinit (am_spi_handle_t handle);
17	SPI1(DMA 方式)	void am_hc32_spi1_dma_inst_deinit (am_spi_handle_t handle);
18	SPI0(中断方式)	void am_hc32_spi0_int_inst_deinit (am_spi_handle_t handle);
19	SPI1(中断方式)	void am_hc32_spi1_int_inst_deinit (am_spi_handle_t handle);
20	SPI0(轮询方式)	void am_hc32_spi0_poll_inst_deinit (am_spi_handle_t handle);

下页继续

表 4.11 – 续上页

序号	外设	实例初始化函数原型
21	SPI1(轮询方式)	void am_hc32_spi1_poll_inst_deinit (am_spi_handle_t handle);
22	SYSTICK	void am_hc32_systick_inst_deinit (am_timer_handle_t handle);
23	TIM0_CAP	void am_hc32_tim0_cap_inst_deinit (am_cap_handle_t handle);
24	TIM1_CAP	void am_hc32_tim1_cap_inst_deinit (am_cap_handle_t handle);
25	TIM2_CAP	void am_hc32_tim2_cap_inst_deinit (am_cap_handle_t handle);
26	TIM3_CAP	void am_hc32_tim3_cap_inst_deinit (am_cap_handle_t handle);
27	TIM4_CAP	void am_hc32_tim4_cap_inst_deinit (am_cap_handle_t handle);
28	TIM5_CAP	void am_hc32_tim5_cap_inst_deinit (am_cap_handle_t handle);
29	TIM6_CAP	void am_hc32_tim6_cap_inst_deinit (am_cap_handle_t handle);
30	TIM0_PWM	void am_hc32_tim0_pwm_inst_deinit (am_pwm_handle_t handle);
31	TIM1_PWM	void am_hc32_tim1_pwm_inst_deinit (am_pwm_handle_t handle);
32	TIM2_PWM	void am_hc32_tim2_pwm_inst_deinit (am_pwm_handle_t handle);
33	TIM3_PWM	void am_hc32_tim3_pwm_inst_deinit (am_pwm_handle_t handle);
34	TIM4_PWM	void am_hc32_tim4_pwm_inst_deinit (am_pwm_handle_t handle);
35	TIM5_PWM	void am_hc32_tim5_pwm_inst_deinit (am_pwm_handle_t handle);
36	TIM6_PWM	void am_hc32_tim6_pwm_inst_deinit (am_pwm_handle_t handle);
37	TIM0_TIMING	void am_hc32_tim0_timing_inst_deinit (am_timer_handle_t handle);
38	TIM1_TIMING	void am_hc32_tim1_timing_inst_deinit (am_timer_handle_t handle);
39	TIM2_TIMING	void am_hc32_tim2_timing_inst_deinit (am_timer_handle_t handle);
40	TIM3_TIMING	void am_hc32_tim3_timing_inst_deinit (am_timer_handle_t handle);
41	TIM4_TIMING	void am_hc32_tim4_timing_inst_deinit (am_timer_handle_t handle);
42	TIM5_TIMING	void am_hc32_tim5_timing_inst_deinit (am_timer_handle_t handle);
43	TIM6_TIMING	void am_hc32_tim6_timing_inst_deinit (am_timer_handle_t handle);
44	USART0	void am_hc32_uart0_inst_deinit (am_uart_handle_t handle);
45	USART1	void am_hc32_uart1_inst_deinit (am_uart_handle_t handle);
46	USART2	void am_hc32_uart2_inst_deinit (am_uart_handle_t handle);
47	USART3	void am_hc32_uart3_inst_deinit (am_uart_handle_t handle);
48	USB	void am_hc32f07x_usbd_mouse_inst_deinit (void);
49	CAN	void am_hc32f07x_can_inst_deinit (am_can_handle_t handle);
50	LPUART0	void am_hc32_lpuart0_inst_deinit (am_uart_handle_t handle);
51	LPUART1	void am_hc32_lpuart1_inst_deinit (am_uart_handle_t handle);

注意： 时钟部分不能被解初始化。

外设实例解初始化函数相对简单，所有实例解初始化函数均无返回值。

关于解初始化函数的参数，若实例初始化时返回值为 **int** 类型，则实例解初始化时无需传入任何参数；若实例初始化函数返回了一个服务句柄，则实例解初始化时应该传入实例初

始化函数获得的服务句柄。

4.3.2 直接使用硬件层函数

一般情况下，使用设备实例初始化函数返回的 handle，再利用标准接口层或驱动层提供的函数。已经能满足绝大部分应用场合。若在一些效率要求很高或功能要求很特殊的场合，可能需要直接操作硬件。此时，则可以直接使用 HW 层提供的相关接口。

通常，HW 层的接口函数都是以外设寄存器结构体指针为参数。

以 SPI 为例，所有硬件层函数均在 **\ametal\soc\hdsc\drivers\include\spi\hw\amhw_hc32_spi.h** 文件中声明（一些简单的内联函数直接在该文件中定义）。简单列举几个函数，详见 列表 4.50 。

列表 4.50 SPI 硬件层操作函数

```
/**
 * \brief 标志有效性判断
 * \param[in] p_hw_spi : 指向 SPI 寄存器结构体的指针
 * \param[in] flag     : 标志宏定义
 *
 * \return AM_TRUE : 标志有效
 *         AM_FALSE: 标志无效
 */
am_static_inline
am_bool_t amhw_hc32_spi_flag_check (amhw_hc32_spi_t *p_hw_spi,
                                    uint8_t          flag)
{
    return ((p_hw_spi->stat & flag)) ? AM_TRUE : AM_FALSE;
}

/**
 * \brief  SPI 中断清除
 * \param[in] p_hw_spi : 指向 SPI 寄存器结构体的指针
 *
 * \return none
 */
am_static_inline
void amhw_hc32_spi_flag_clr (amhw_hc32_spi_t *p_hw_spi)
{
    p_hw_spi->iclr &= ~(0x1ul << 0);
}
```

其它一些函数读者可自行打开 **\ametal\soc\hdsc\drivers\include\spi\hw\amhw_hc32_spi.h** 文件查看。这些函数均是以 amhw_hc32_spi_t * amhw_hc32_spi_t 类型在 **\ametal\soc\hdsc\drivers\include\spi\hw\amhw_hc32_spi.h** 文件中定义，用于定义出 SPI 外设的各个寄存器。详见 列表 4.51 。

列表 4.51 SPI 寄存器结构体定义

```
/**
 * \brief SPI structure of register
 */
typedef struct amhw_hc32_spi {
    __IO uint32_t  cr;       /**< \brief SPI 配置寄存器 */
    __IO uint32_t  ssn;      /**< \brief SPI 片选配置寄存器 */
    __I  uint32_t  stat;     /**< \brief SPI 状态寄存器 */
    __IO uint32_t  data;     /**< \brief SPI 数据寄存器 */
    __IO uint32_t  cr2;      /**< \brief SPI 配置寄存器 2 */
    __O  uint32_t  iclr;     /**< \brief SPI 中断清除寄存器 */
```

```
} amhw_hc32_spi_t;
```

该类型的指针已经在 **\ametal\soc\hdsc\hc32f07x\hc32f07x_periph_map.h** 文件中定义，应用程序可以直接使用。详见 列表 4.52 。

列表 4.52 SPI 寄存器结构体指针定义

```
/** \brief SPI1 控制器寄存器块指针 */
#define HC32_SPI0 ((amhw_hc32_spi_t *)HC32F07X_SPI0_BASE)

/** \brief SPI2 控制器寄存器块指针 */
#define HC32_SPI1 ((amhw_hc32_spi_t *)HC32F07X_SPI1_BASE)
```

注解： 其中的 **HC32F07X_SPI0_BASE** 和 **HC32F07X_SPI1_BASE** 是 SPI0 和 SPI1 外设寄存器的基地址，在 **\ametal\soc\hdsc\hc32f07x\hc32f07x_regbase.h** 文件中定义，其他所有外设的基地址均在该文件中定义。

有了这两个 SPI 寄存器结构体指针宏后，就可以直接使用 SPI 硬件层的相关函数了。使用硬件层函数时，若传入参数为 **HC32_SPI0** ，则表示操作的是 SPI0；若传入参数为 **HC32_SPI1** ，则表示操作的是 SPI1。如 列表 4.53 和 列表 4.54 所示，分别用于 SPI0 和 SPI1 的波特率获取。

列表 4.53 波特率获取 SPI0

```
amhw_hc32_spi_baud_rate_get (HC32_SPI0);
```

列表 4.54 波特率获取 SPI1

```
amhw_hc32_spi_baud_rate_get (HC32_SPI1);
```

特别地，可能想要操作的功能，硬件层也没有提供出相关接口，此时，可以基于各个外设指向寄存器结构体的指针，直接操作寄存器实现，例如，要清除 SPI0 和 SPI1 中断标志，也可以直接直接设置寄存器的值，详见 列表 4.55 和 列表 4.56 。

列表 4.55 直接设置寄存器的值清除中断标志 SPI0

```
/*
 * 设置 iclr 寄存器的 bit0 为 0，中断标志清除
 */
HC32_SPI0->iclr &= ~(0x1ul << 0);
```

列表 4.56 直接设置寄存器的值清除中断标志 SPI1

```
/*
 * 设置 iclr 寄存器的 bit0 为 0，中断标志清除
 */
HC32_SPI1->iclr &= ~(0x1ul << 0);
```

注解： 一般情况下，均无需这样操作。若特殊情况下需要以这种方式操作寄存器，应详细了解该寄存器各个位的含义，谨防出错。

所有外设均在 **\ametal\soc\hdsc\hc32f07x\include\hc32f07x_periph_map.h** 文件中定义了

指向外设寄存器的结构体指针，与各外设对应的指向该外设寄存器的结构体指针宏详见 表 4.12 。

表 4.12 指向各片上外设寄存器结构体的指针宏

序号	外设	指向该外设寄存器结构体的指针宏
1	GPIO	HC32_GPIO0
2	TIM0	HC32_TIM0
3	TIM1	HC32_TIM1
4	TIM2	HC32_TIM2
5	TIM3	HC32_TIM3
6	TIM4	HC32_TIM4
7	TIM5	HC32_TIM5
8	TIM6	HC32_TIM6
9	RTC	HC32_RTC
10	WDT	HC32_WDT
11	USART0	HC32_UART0
12	USART1	HC32_UART1
13	USART2	HC32_UART2
14	USART3	HC32_UART3
15	ADC	HC32_ADC
16	DAC	HC32_DAC
17	DMA	HC32_DMA
18	RCC	HC32_RCC
19	SPI0	HC32_SPI0
20	SPI1	HC32_SPI1
21	I2C0	HC32_I2C0
22	I2C1	HC32_I2C1
23	PCA	HC32_PCA
24	TRNG	HC32_TRNG
25	CAN	HC32_CAN
26	USB	HC32_USB
27	FLASH	HC32_FLASH
28	CRC	HC32_CRC
29	RAM	HC32_RAM
30	LPUART0	HC32_LPUART0
31	LPUART1	HC32_LPUART1
32	AES	HC32_AES
33	CLKTRIM	HC32_CLKTRIM
34	LVD	HC32_LVD

下页继续

表 4.12 – 续上页

序号	外设	指向该外设寄存器结构体的指针宏
35	OPA	HC32_OPA
36	VC	HC32_VC
37	CTS	HC32_CTS
38	I2S0	HC32_I2S0
39	I2S1	HC32_I2S1

5. 板级资源

与板级相关的资源默认情况下使能即可使用。特殊情况下，LED、按键、调试串口、系统滴答和软件定时器可能需要进行一些配置。所有资源的配置由 {HWCONFIG} 下的一组 am_hwconf_* 开头的 **.c** 文件完成的。

各资源及其对应的配置文件如 表 5.1 所示。

表 5.1 板级资源的配置文件

序号	外设	配置文件
1	按键	am_hwconf_key_gpio.c
2	LED	am_hwconf_led_gpio.c
3	调试串口	am_hwconf_debug_uart.c
4	系统滴答和软件定时器	am_hwconf_system_tick_softimer.c

5.1 配置文件结构

板级资源的配置文件与片上外设配置文件结构基本类似。一般来说，板级资源配置只需要设备信息和实例初始化函数即可。而且实例初始化函数通常情况下不需要用户手动调用，也不需要用户自己修改。只需要在工程配置文件 {PROJECT}**\user_config\am_prj_config.h** 中打开或禁用相应的宏，相关资源会在系统启动时在 {PROJECT}**\am_board.c** 中自动完成初始化。以 LED 为例，初始化代码详见 列表 5.1 。

列表 5.1 LED 实例初始化函数调用

```
/**
 * \brief 板级初始化
 */
void am_board_init (void)
{
    // ……
#if (AM_CFG_LED_ENABLE == 1)
    am_led_gpio_inst_init();
#endif /* (AM_CFG_LED_ENABLE == 1) */
    // ……
}
```

5.2 典型配置

5.2.1 LED 配置

AMF07X_Core 板上有 4 个 LED 灯，默认引脚分别为 PIOE_0, PIOE_1, PIOE_2 和 PIOE_3。LED 相关信息定义在 {PROJECT}**user_config\am_hwconf_usrcfg\am_hwconf_led_gpio.c** 文件中，详见 列表 5.2 。

列表 5.2 LED 相关配置信息

```
/** \brief 定义 LED 相关的 GPIO 管脚信息 */
static const int __g_led_pins[] = {PIOE_0, PIOE_1, PIOE_2, PIOE_3};

/** \brief 设备信息 */
static const am_led_gpio_info_t __g_led_gpio_info = {
    {
        0,                              /* 起始编号 */
        AM_NELEMENTS(__g_led_pins) - 1 /* 结束编号 */
    },
    __g_led_pins,
    AM_TRUE
};
```

其中，am_led_gpio_info_t 类型在 **\ametal\components\drivers\include\am_led_gpio.h** 文件中定义，详见 列表 5.3 。

列表 5.3 LED 引脚配置信息类型定义

```
typedef struct am_led_gpio_info {

    /** \brief LED 基础服务信息 ，包含起始编号和结束编号 */
    am_led_servinfo_t  serv_info;

    /** \brief 使用的 GPIO 引脚，引脚数目应该为 （结束编号 - 起始编号 + 1）*/
    const int         *p_pins;

    /** \brief LED 是否是低电平点亮  */
    am_bool_t          active_low;

} am_led_gpio_info_t;
```

其中，serv_info 为 LED 的基础服务信息，包含 LED 的起始编号和介绍编号，p_pins 指向存放 LED 引脚的数组首地址，在本平台可选择的管脚在 hc32f07x_pin.h 文件中定义，active_low 参数用于确定其点亮电平，若是低电平点亮，则该值为 AM_TRUE，否则，该值为 AM_FALSE。

可见，在 LED 配置信息中，LED0 和 LED1 分别对应 PIOE_0 和 PIOE_1 ，均为低电平点亮。如需添加更多的 LED，只需在该配置信息数组中继续添加即可。

可使用 LED 标准接口操作这些 LED，详见 **\ametal\interface\am_led.h** 。led_id 参数与该数组对应的索引号一致。

注解： 由于 LED 使用了 PIOE_0 和 PIOE_1，若应用程序需要使用这两个引脚，建议通过使能/禁能宏禁止 LED 资源的使用。

5.2.2 按键

AMF07X_Core 有 5 个板载按键 KEY1、KEY2、KEY3、KEY4 和 KEY5，对应引脚分别为 PIOE_0、PIOE_1、PIOE_2、PIOE_3。KEY 相关信息定义在 {PROJECT}**user_config\am_hwconf_usrcfg\am_hwconf_key_gpio.c** 文件中，详见 列表 5.4 。

列表 5.4 KEY 相关配置信息

```
static const int __g_key_pins[] = {PIOE_0, PIOE_1,PIOE_2, PIOE_3};
static const int __g_key_codes[] = {KEY_KP0, KEY_KP1, KEY_KP2, KEY_KP3};

/** \brief 设备信息 */
static const am_key_gpio_info_t __g_key_gpio_info = {
    __g_key_pins,
    __g_key_codes,
    AM_NELEMENTS(__g_key_pins),
    AM_TRUE,
    10
};
```

其中，KEY_KP0 为默认按键编号；AM_NELEMENTS 是计算按键个数的宏函数；am_key_gpio_info_t 类型在 {SDK}**ametal\components\drivers\include\am_key_gpio.h** 文件中定义，详见 列表 5.5 。

列表 5.5 KEY 引脚配置信息类型定义

```
/**
 * \brief 按键信息
 */
typedef struct am_key_gpio_info {
    const int  *p_pins;              /**< \brief 使用的引脚号                  */
    const int  *p_codes;             /**< \brief 各个按键对应的编码（上报） */
    int         pin_num;             /**< \brief 按键数目 */
    am_bool_t   active_low;          /**< \brief 是否低电平激活（按下为低电平）
↪*/
    int         scan_interval_ms;    /**< \brief 按键扫描时间间隔，一般 10ms */
} am_key_gpio_info_t;
```

p_pins 指向存放 KEY 引脚的数组首地址，在本平台可选择的管脚在 hc32f07x_pin.h 文件中定义；p_codes 指向存放按键对应编码的数组首地址；pin_num 为按键数目；active_low 参数用于确定其点亮电平，若是低电平点亮，则该值为 AM_TRUE，否则，该值为 AM_FALSE；scan_interval_ms 按键扫描时间，一般为 10ms 。

可见，在 KEY 配置信息中，低电平有效。如需添加更多的 KEY，只需在 **__g_key_pins** 和 **__g_key_codes** 数组中继续添加按键对应的管脚和编码即可。

注解： 由于 KEY/RES 使用了引脚 PIOE_0、PIOE_1、PIOE_2、PIOE_3，若应用程序需要使用这个引脚，建议通过使能/禁能宏禁止 KEY 资源的使用。

5.2.3 调试串口配置

AMF07X_Core 具有 4 个串口，可以选择使用其中一个串口来输出调试信息。使用 {PROJECT}**user_config\am_hwconf_usrcfg\am_hwconf_debug_uart.c** 文件中的两个相关宏

用来配置使用的串口号和波特率，相应宏名及含义详见 表 5.2 。

表 5.2 调试串口相关配置

宏名	含义
__DEBUG_UART	0-UART0，1-UART1，2-UART2，3-UART3
__DEBUG_BAUDRATE	使用的波特率，默认 115200

注解： 每个串口还可能需要引脚的配置，这些配置属于具体外设资源的配置，详见第 4 章中的相关内容。若应用程序需要使用串口，应确保调试串口与应用程序使用的串口不同，以免冲突。调试串口的其它配置固定为：8-N-1（8 位数据位，无奇偶校验，1 位停止位）。

5.2.4 系统滴答和软件定时器配置

系统滴答需要 TIM2 定时器为其提供一个周期性的定时中断，默认使用 TIM2 定时器的通道 0。其配置还需要使用 {PROJECT}**\user_config\am_hwconf_usrcfg\am_hwconf_system_tick_softimer.c** 文件中的 **__SYSTEM_TICK_RATE** 宏来设置系统滴答的频率，默认为 1KHz。详细定义见 列表 5.6 。

列表 5.6 系统 TICK 频率配置

```
/**
 * \brief 设置系统滴答的频率，默认 1KHz
 *
 * 系统滴答的使用详见 am_system.h
 */
#define __SYSTEM_TICK_RATE      1000
```

软件定时器基于系统滴答实现。它的配置也需要使用 {PROJECT}**\user_config\am_hwconf_usrcfg\am_hwconf_system_tick.c** 文件中的 **__SYSTEM_TICK_RATE** 宏来设置运行频率，默认 1KHz。详细定义见 列表 5.6 。

注解： 使用软件定时器时必须开启系统滴答。

5.3 使用方法

板级资源对应的设备实例初始化函数的原型详见 表 5.3 ，使用方法可以参考 4.3.1.2 。

表 5.3 板级资源及对应的实例初始化函数

序号	板级资源	实例初始化函数原型
1	按键	int am_key_gpio_inst_init (void);
2	LED	int am_led_gpio_inst_init (void);
3	调试串口	am_uart_handle_t am_debug_uart_inst_init (void);
4	系统滴答	am_timer_handle_t am_system_tick_inst_init (void);
5	系统滴答和软件定时器	am_timer_handle_t am_system_tick_softimer_inst_init (void);

6. 免责声明

应用信息：本应用信息适用于嵌入式产品的开发设计。客户在开发产品前，必须根据其产品特性给予修改并验证。

修改文档的权利：本手册所陈述的产品文本及相关软件版权均属广州周立功单片机科技有限公司所有，其产权受国家法律绝对保护，未经本公司授权，其它公司、单位、代理商及个人不得非法使用和拷贝，否则将受到国家法律的严厉制裁。广州周立功单片机科技有限公司保留在任何时候修订本用户手册且不需通知的权利。您若需要我公司产品及相关信息，请及时与我们联系，我们将热情接待。

销售与服务网络

广州立功科技股份有限公司

地址：广州市天河区龙怡路 117 号银汇大厦 16 楼
邮编：510630
网址：www.zlgmcu.com

全国服务热线电话：400-888-2705

华南地区

广州总部

广州市天河区龙怡路 117 号银汇大厦 16 楼

华南汽车

深圳市坪山区比亚迪路大万文化广场 A 座 1705

厦门办事处

厦门市思明区厦禾路 855 号英才商厦 618 室

深圳分公司

深圳市福田区深南中路 2072 号电子大厦 1203 室

华东地区

上海分公司

上海市黄浦区北京东路 668 号科技京城东座 12E 室

苏州办事处

江苏省苏州市广济南路 258 号（百脑汇科技中心 1301 室）

南京分公司

南京市秦淮区汉中路 27 号友谊广场 17 层 F、G 区

合肥办事处

安徽省合肥市蜀山区黄山路 665 号汇峰大厦 1607

杭州分公司

杭州市西湖区紫荆花路 2 号杭州联合大厦 A 座 4 单元 508

宁波办事处

浙江省宁波市高新区星海南路 16 号轿辰大厦 1003

华北、东北地区

北京分公司

北京市海淀区紫金数码园 3 号楼（东华合创大厦）8 层 0802 室

天津办事处

天津市河东区十一经路与津塘公路交口鼎泰大厦 1004 室

山东办事处

山东省青岛市李沧区青山路 689 号宝龙公寓 3 号楼 701

沈阳办事处

沈阳市浑南新区营盘西街 17 号万达广场 A4 座 2722 室

华中地区

武汉分公司

武汉市武昌区武珞路 282 号思特大厦 807 室

西安办事处

西安市高新区科技二路 41 号高新水晶城 C 座 616 室

郑州办事处

河南郑州市中原区百花路与建设路东南角锦绣华庭 A 座 1502 室

长沙办事处

湖南省长沙市岳麓区奥克斯广场国际公寓 A 栋 2309 房

西南地区

重庆办事处

重庆市渝北区龙溪街道新溉大道 18 号山顶国宾城 11 幢 4-14

成都办事处

成都市一环路南二段 1 号数码科技大厦 403 室

请您用以上方式联系我们，我们会为您安排样机现场演示，感谢您对我公司产品的关注！