AMetal-AML165-Core 用户手册

类别	内容
关键词	AML165-Core、ZML165、功能介绍
摘要	本文档简述了 AML165-Core 硬件资源，详细介绍了 AMetal-AML165-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	2018/4/16	创建文档

目 录

1. 开发平台简介

AML165-Core 开发平台主要用于 ZML165 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 AML165-Core 200723 Rev.C P/N: 13.01.04993。用户在使用 SDK 前请确认 SDK 支持自己手中的开发平台。

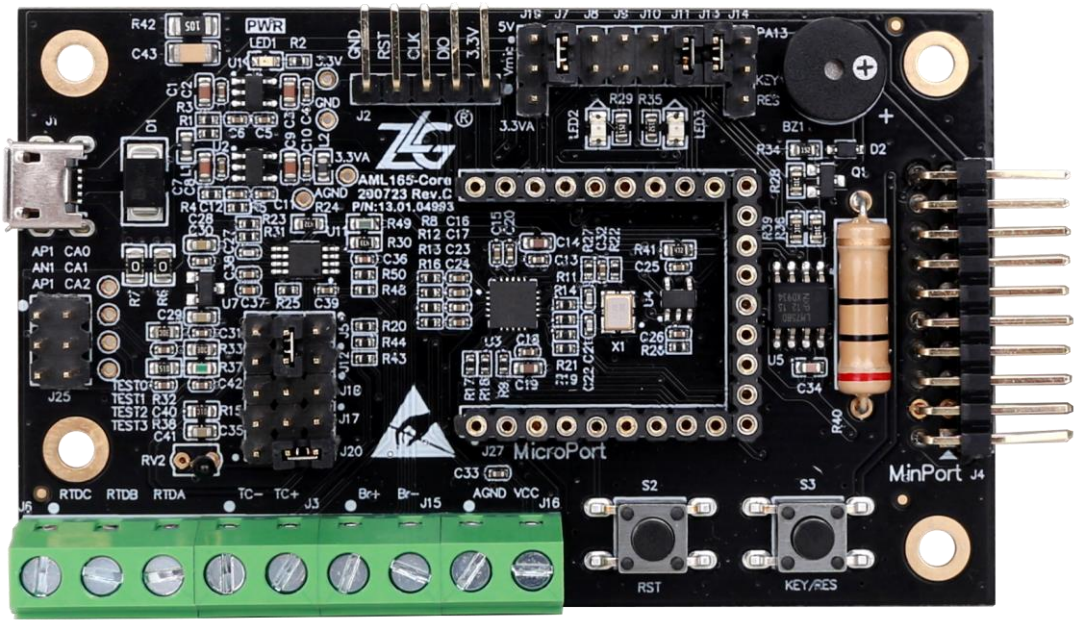

图 1.1　AML165-Core 开发平台

AML165-Core 评估板是提供给客户快速上手 ZML165N20A 芯片的硬件平台，ZML165N20A 是一款内置 24 位 ADC 的 Cortex M0 的混合信号微控制器，其外形小巧、结构简单、片上资源设计合理。名片大小的电路板上包含了 1 路 MiniPort 接口、1 路 MicroPort 接口。这些接口不仅把单片机的大部分 I/O 资源引出，还可以借助 MiniPort 接口和 MicroPort 接口外扩多种模块。

1.1　ZML165

- 工作电压 3.0V ~ 5.5V；
- ARM Cortex-M0 32 位内核，主频可达 48MHz；
- 32KB Flash，4KB SRAM；
- 1 个 12 位 ADC、1 个 24 位的 ADC；
- 多达 9 个定时器；
- 1 个 UART 接口、1 个 I2C 接口和 1 个 SPI 接口；
- 5 通道 DMA 控制器；
- 睡眠、停机和待机模式；
- 96 位的芯片唯一 ID（UID）。

1.2 AML165-Core

- 5V MicroUSB 供电；
- 2 个 LED 发光二极管；
- 1 个无源蜂鸣器；
- 1 个加热电阻；
- 1 个 LM75B 测温芯片；
- 1 个多功能按键（可用跳线帽选择用作加热按键或是独立按键）；
- 1 个复位按键；

2. aml165_core 软件包

软件包名为 aml165_core（路径：ametal\board\aml165_core）为叙述方便，下文简称 ametal 为 SDK，使用 **{SDK}** 表示软件包的路径。

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。

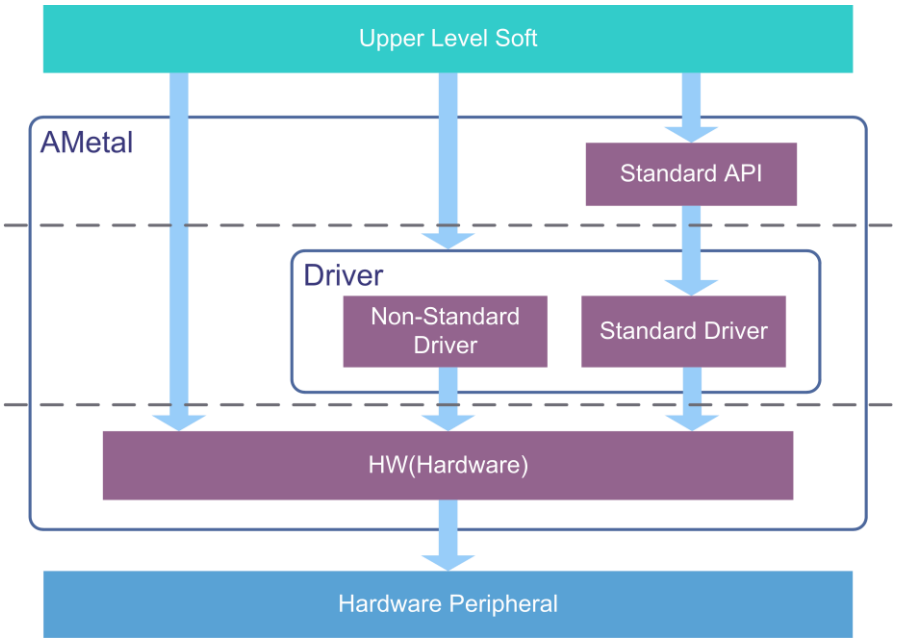

图 2.1 AMetal 框架

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_zml165、AMHW_ZML165。

参见：更多的硬件层接口定义及示例请参考 {SDK}\documents\《AMetal API 参考手册.chm》或者 {SDK}\soc\zlg\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 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_zml165、AM_ZML165。

参见:更多的驱动层接口定义及示例请参考 {SDK}\documents\《AMetal API 参考手册.chm》

或者 {SDK}\soc\zlg\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 {SDK}\documents\《AMetal API 参考手册.chm》或者 {SDK}\interface 文件夹中的相关文件。

2.2 目录结构

{SDK}下一般有 9 个文件夹，如图 2.2 所示。

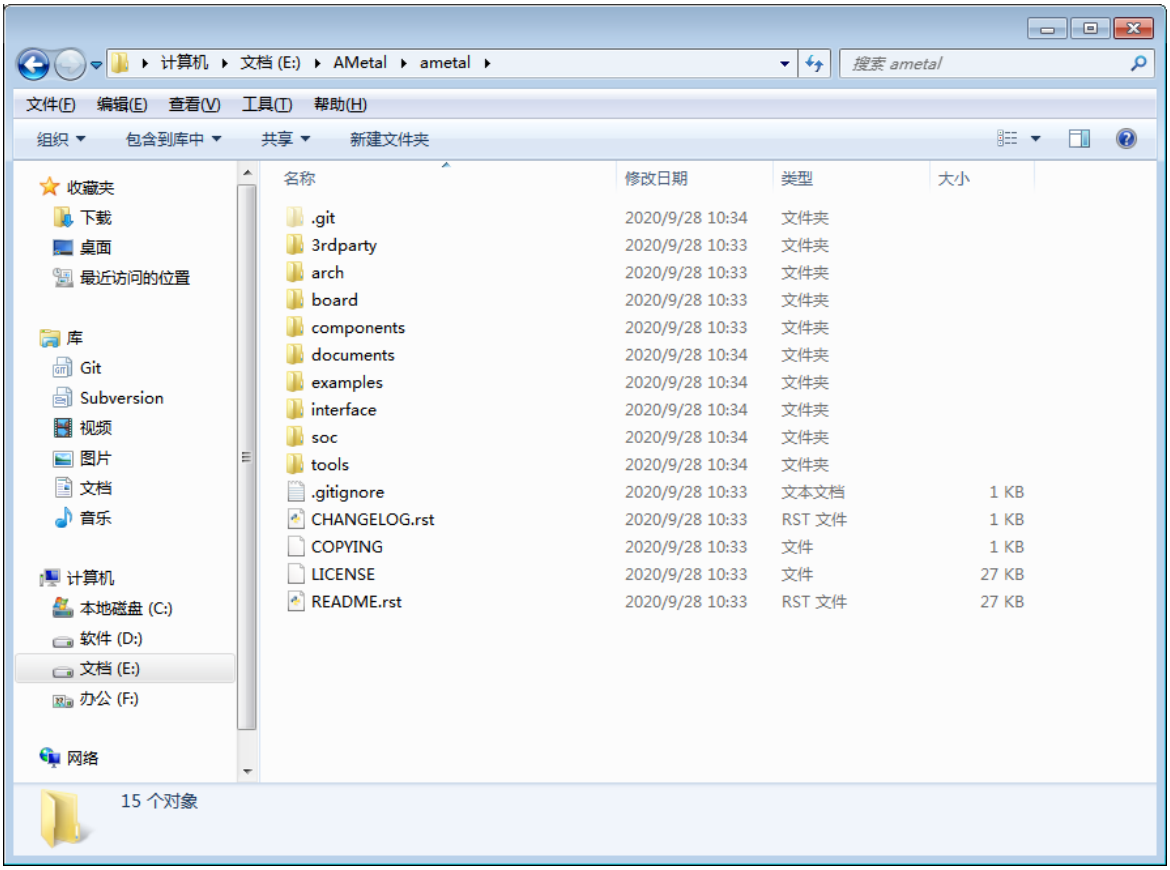

图 2.2 软件包目录结构

- 3rdparty 存放第三方软件包；
- arch 存放内核相关文件；
- board 存放板级相关文件；
- components 存放组件文件；
- documents 存放各类文档；
- examples 存放各类例程；
- interface 存放 AMetal 标准接口文件；
- soc 存放片上系统（MCU）相关文件；
- tools 存放一些工具包，如 keil 的 PACK 包；
- CHANGELOG 版本修改记录文件。

2.2.1 ametal 目录

(1) 3rdparty

3rdparty 文件夹用于放置第三方软件包，这些软件不是由广州致远电子有限公司开发，当用户需要用到某些第三方软件时，可以把它们存放到这个文件夹内。例如 CMSIS 就是一个常用的第三方软件包，它是 Cortex-M 系列处理器的微控制器软件接口标准。

(2) arch

arch 文件夹用于存放与架构相关的通用文件，如 ARM、 X86 等。该目录下按不同的架构分成不同的文件夹，如 arm 文件夹。

(3) board

board 文件夹包含了与开发板相关的文件，如启动文件及与开发板相关的设置和初始化函数等。 board 文件夹内分为板级通用文件夹“bsp_common”和若干某一型号开发板的专用文件夹，如“aml165_core”。分别用于存放板级通用文件和对应型号开发板的专用板级文件。其目录结构如图 2.3 所示。

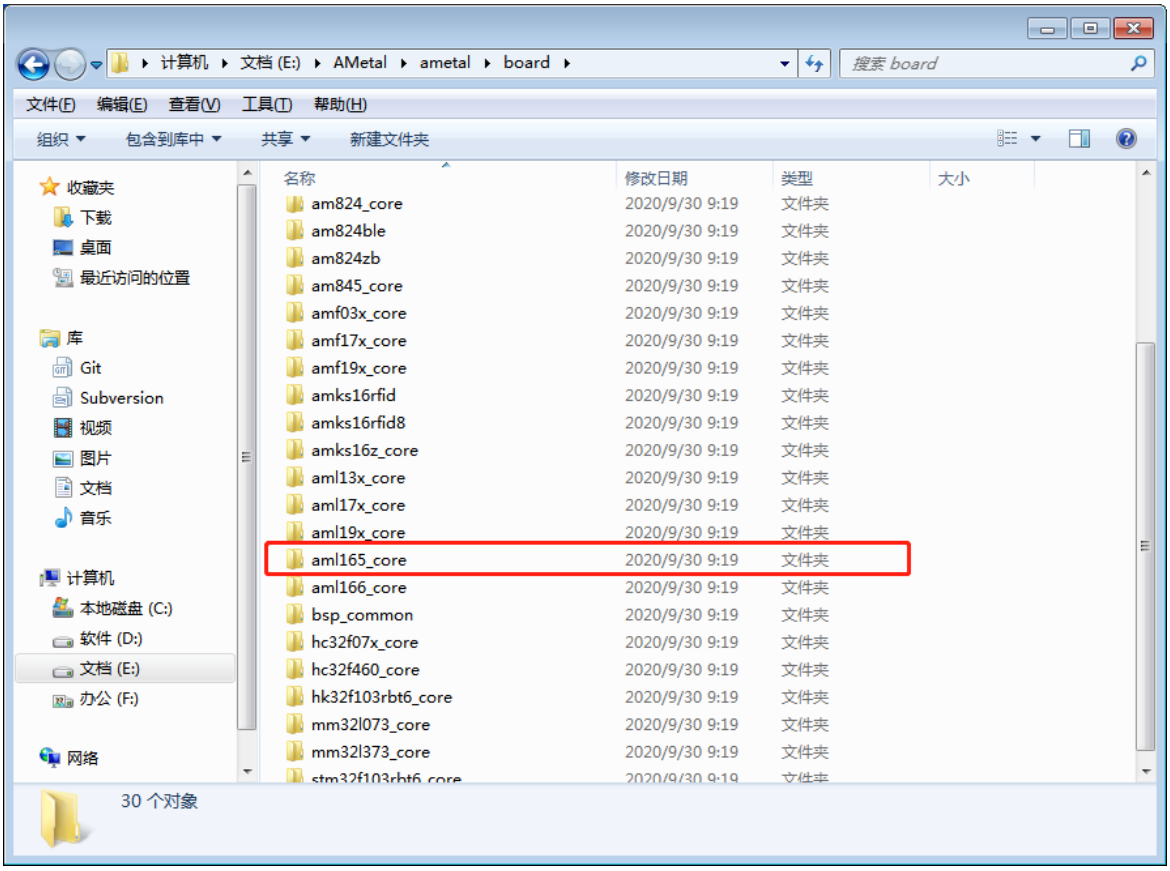

图 2.3 board 目录结构

aml165_core 存放的是 aml165_core 开发平台相关的板级文件。该目录下一般包含一个模板工程文件夹“project_template”和一个示例工程文件夹“project_example”。模板工程文件夹一般用于创建工程，示例工程文件夹一般用于运行各个 demo 程序。两个文件夹的目录结构相同，本文仅以模板工程文件夹为例进行介绍。 aml165_core 文件夹目录结构如图 2.4 所示。

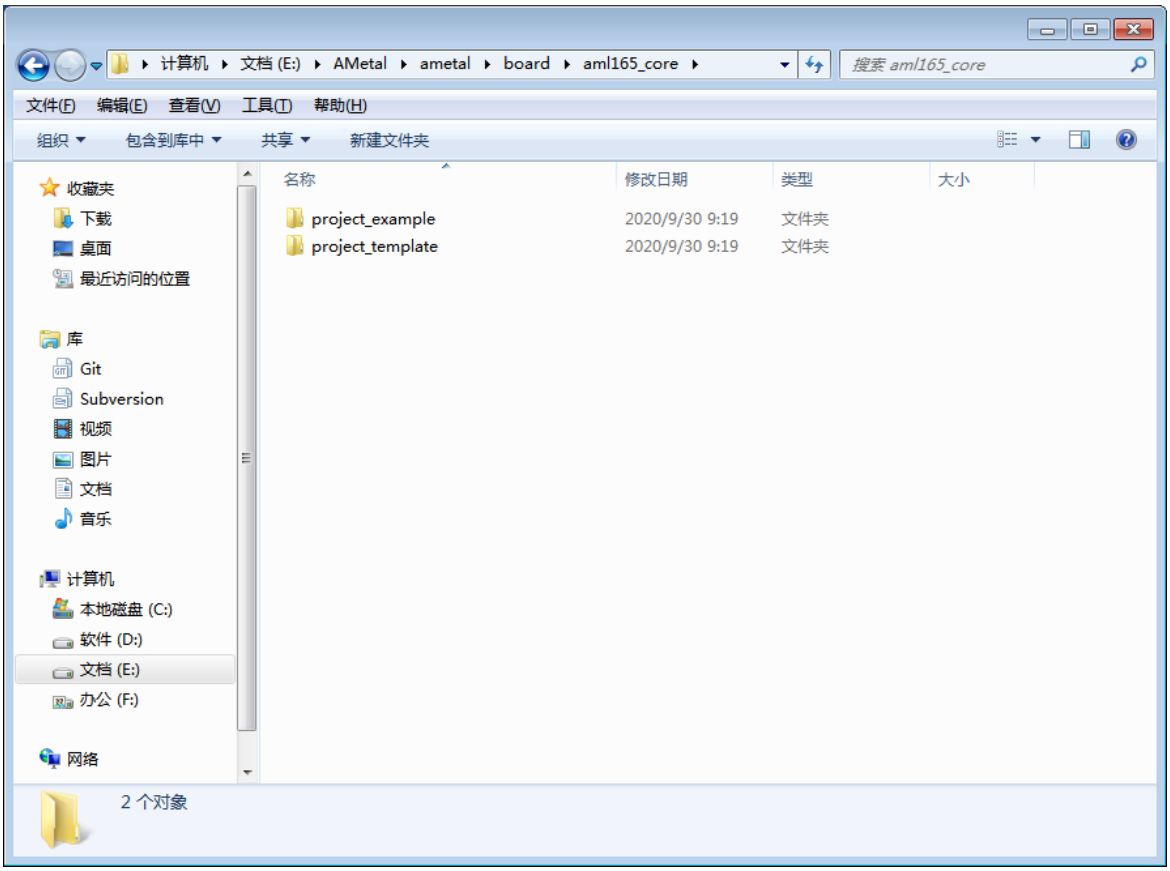

图 2.4　board/aml165_core 目录结构

模板工程文件夹“project_template”存放了 aml165_core 开发板的板级文件，包括各个 IDE 的工程文件（如 eclipse 和 Keil5 的工程文件）启动文件、用户代码和用户配置文件等。其目录结构如图 2.5 所示。

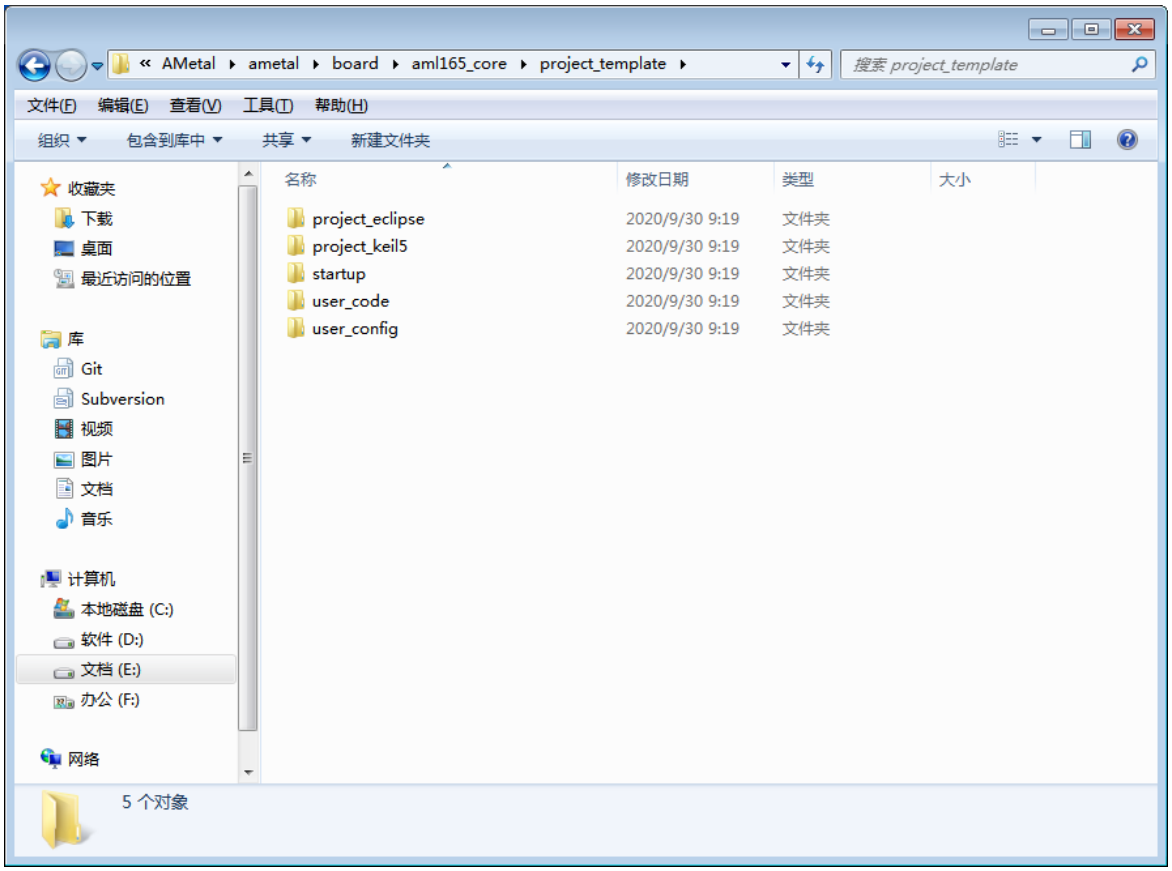

图 2.5　board/aml165_core/project_template 目录结构

- project_eclipse：存放 eclipse 工程文件；
- project_keil5：存放 Keil5 工程文件夹；
- startup：存放启动文件；
- user_code：存放用户代码；
- user_config：存放用户配置文件。

(4)　components

components 文件夹用于存放 AMetal 的一些组件。比如 AMetal 通用服务组件 service，其内部包含了一些通用外设的抽象定义，如蜂鸣器、数码管等以及它们的标准接口函数定义等，用户可通过 AMetal 标准接口调用。

(5)　documents

documents 用于存放 SDK 相关文档，通常包含《快速入门手册.pdf》、《AMetal-AML165-Core 用户手册.pdf》、《API 参考手册.chm》及《引脚配置及查询.xlsm》等。

1. 《快速入门手册.pdf》

快速入门手册介绍了获取到 SDK 后，如何快速的搭建好开发环境，成功运行、调试第一个程序。建议首先阅读。

2. 《AMetal-AML165-Core 用户手册.pdf》

用户手册详细介绍了 AMetal 架构、目录结构、平台资源以及通用外设常见的配置方法。

3. 《API 参考手册.chm》

API 参考手册详细描述了 SDK 各层中每个 API 函数的使用方法，往往还提供了 API 函数的使用范例。在使用 API 之前，应该通过该文档详细了解 API 的使用方法和注意事项。

4. 《引脚配置及查询.xlsm》

引脚配置及查询表可以用于查询引脚的上下拉模式，以及引脚是否有高驱动能力或者具有滤波功能，里面还详细介绍了各个引脚可以用于哪些外设，并提供了可以快速生成对应于外设的引脚配置代码。

(6) examples

examples 文件夹主要包含各级示例程序，包括硬件层 demo、驱动层 demo、板级 demo 及组件 demo 等。路径：{SDK}\examples\board\am165_core 主要包含 aml165_core 开发板各个外设及板载器件的例程源文件，am165_core 目录中包含了各个例程的.c 文件，同时还包含了一个.h 文件(demo_aml165_core_entries.h 文件)，此.h 文件中声明了所有例程的入口函数，用户使用例程时，一般都要包含这一个.h 头文件。这些例程对应的工程位于{SDK}\project_keil5 或{SDK}\project_eclipse 目录下。目录视图如图 2.6 所示。

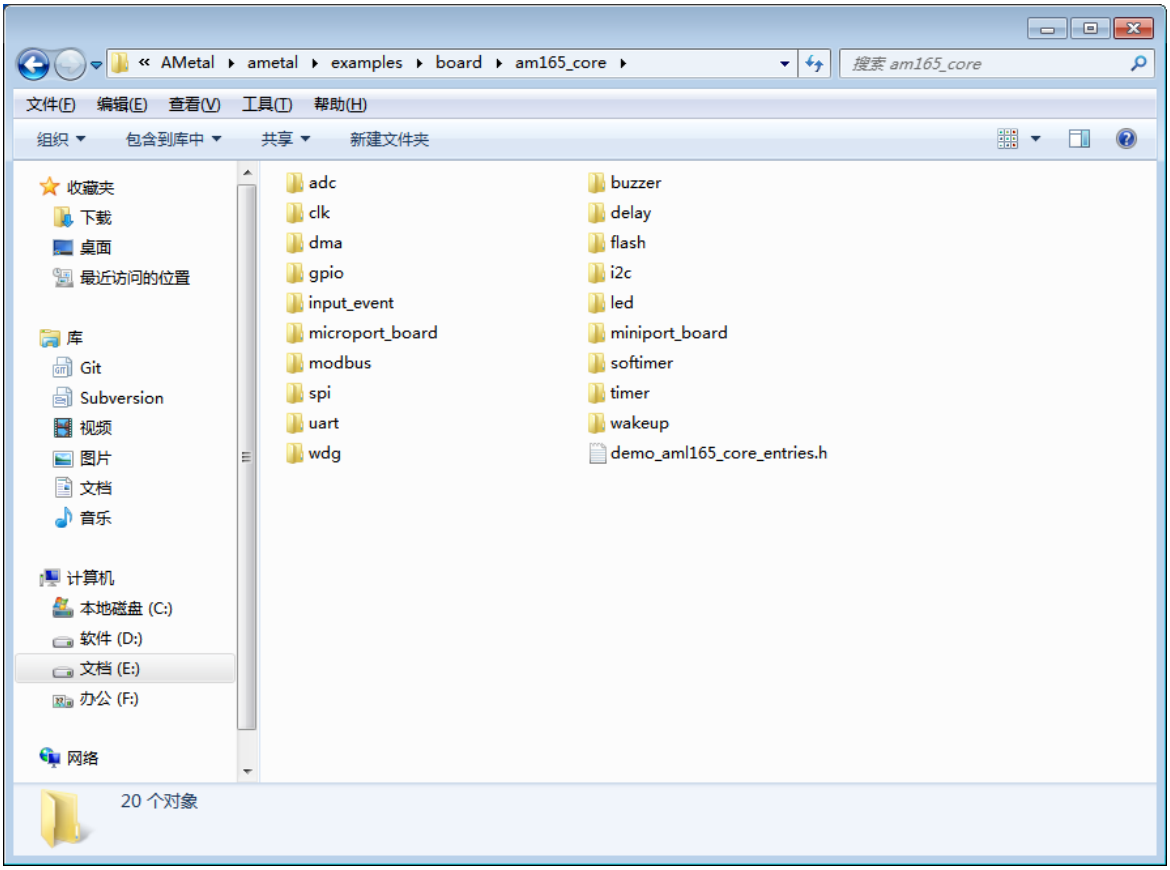

图 2.6 例程源文件目录视图

以 uart 相关外设为例，打开 uart 目录，如图 2.7 所示，可以看到该目录下提供了 6 个 dem 源程序。

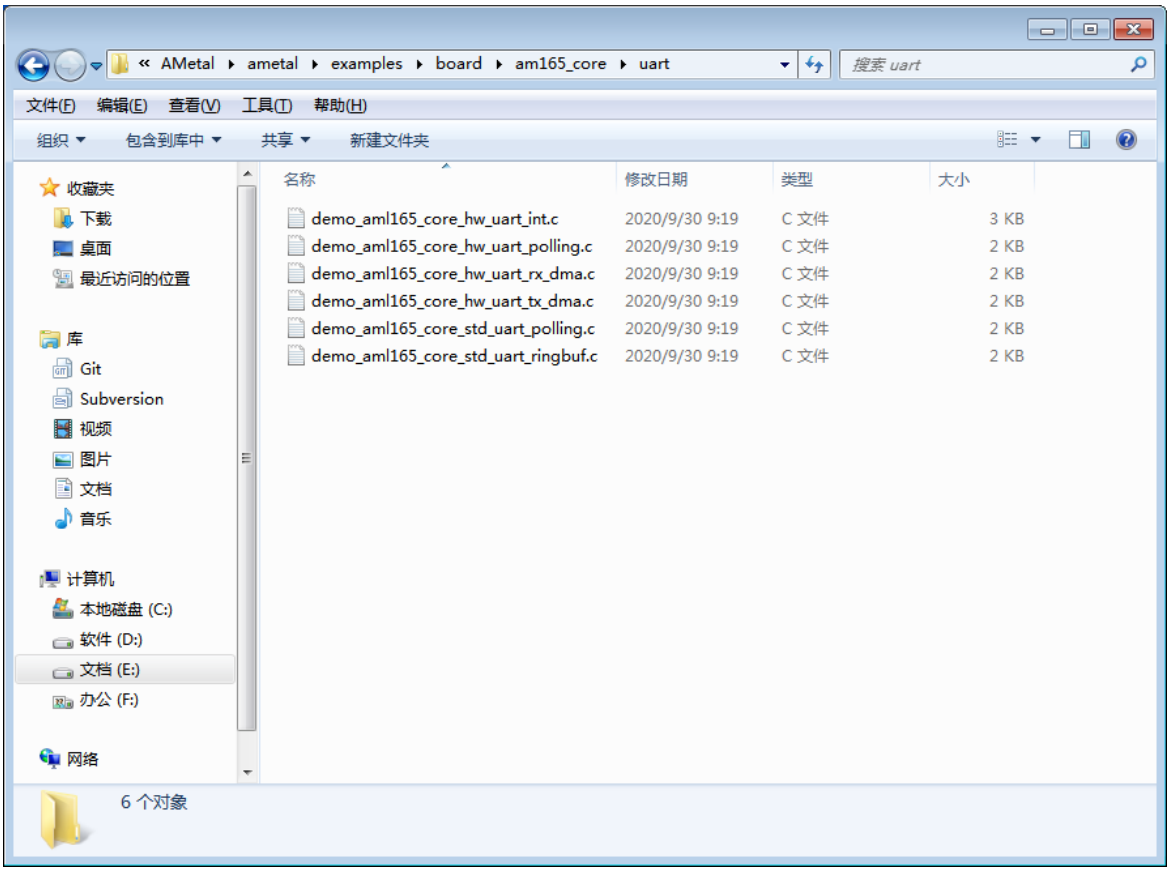

图 2.7　uart 外设所有 demo 源程序

所有芯片外设的 demo 源程序命名为：demo_aml165_std(hw/drv)_{外设名}_{示例功能}.c。所有 demo 程序的入口函数名为 {文件名}_entry()的函数，需要在 am_main()函数中调用相应的 demo 入口函数才能看到相应的实验现象。

- demo_aml165_hw_* 表示该例程展示的是 HW 层接口的使用范例，demo 入口函数一般无参数，直接调用即可；
- demo_aml165_drv_* 表示该例程展示的是驱动层接口的使用范例，demo 入口函数一般无参数，直接调用即可；
- demo_aml165_std_* 表示该例程展示的是标准接口层接口的使用范例，demo 入口函数一般无参数，直接调用即可。

(7)　interface

interface 文件夹下包含 AMetal 提供的通用文件，包括标准接口文件和一些工具文件，这些标准接口与具体芯片无关，只与外设的功能相关，屏蔽了不同芯片底层的差异性，使不同厂商、型号的 MCU 都能以通用接口进行操作。

(8)　soc

soc，片上系统文件夹，主要包含了与 MCU 密切相关的文件，包括硬件层和驱动层文件。如图 2.8 所示

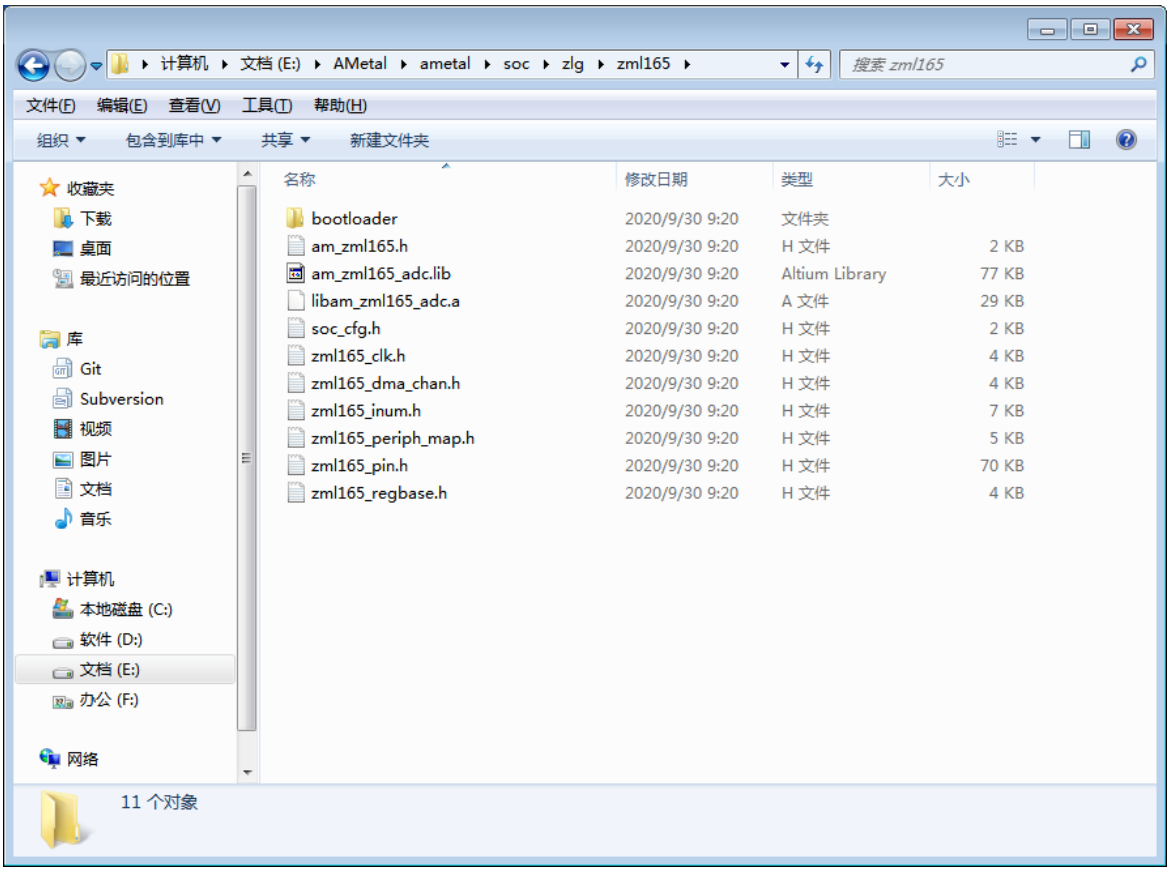

图 2.8　soc 目录

{SDK}\soc\zlg\zml165 目录中的还有几个.h 文件，主要定义了该芯片通用的一些内容，如引脚号、中断号、DMA 通道号等。各文件内容简介如表 2.1 所示。

表 2.1　ZML165 芯片各公共文件内容简介

文件名	内容简介
am_zml165.h	包含了其它 ZML165 公共部分头文件，只要使用 ZML165 的代码，建议默认均包含此头文件
zml165_clk.h	时钟 ID 号，如 CLK_ADC1、CLK_SPI1 等
zml165_dma_chan.h	DMA 通道号相关定义，如 DMA_CHAN_1，DMA_CHAN_2 等
zml165_inum.h	中断号定义，如 INUM_DMA1_1，INUM_I2C1 等
zml165_pin.h	IO 引脚号定义，包含 PIOA、PIOB、PIOC、PIOD 四个端口的引脚
zml165_regbase.h	ZML165 各外设寄存器基地址定义

注解：由于这几个文件很特殊，属于芯片的一些公共定义，并不能指定其属于哪一层。因此，这些文件仅以“芯片名”作为这些文件的命名空间。特别地，将这些公共文件统一包含到了 am_zml165.h 文件中，实际应用程序在使用时，只需要简单的包含 am_zml165.h 即可。

(9)　tools

tools 目录下存放 SDK 相关工具，如 Keil 的 PACK 包。

打开 tools 目录，如图 2.9 所示。

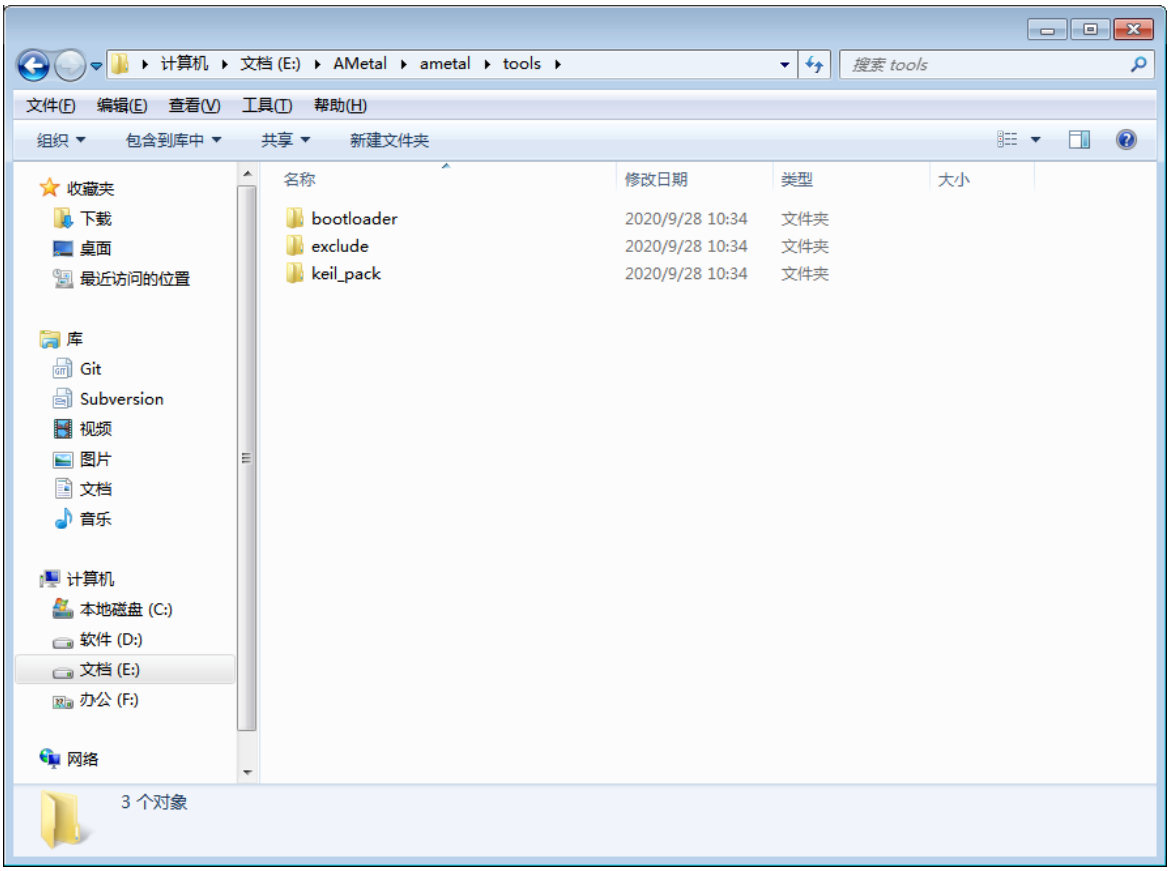

图 2.9　tools 目录视图

2.3　工程结构

2.3.1 Keil 工程结构

找到路径：{SDK}\board\aml165_core\project_template\projects_keil5。

该目录下主要包含 Keil5 的相关工程文件，如链接脚本文件、工程文件以及调试配置文件。其中，“template_aml165_core.uvprojx”是 Keil5 的工程文件，“template_aml165_core.sct”文件是 Keil5 工程的链接脚本文件，类似于 eclipse 工程中的“template_aml165_core.ld”文件。JlinkSettings.ini 是 Keil 工程采用 J-Link 调试时的一些配置信息。“template_aml165_core.uvoptx”是编译时产生的临时文件。其目录结构如图 2.10 所示。

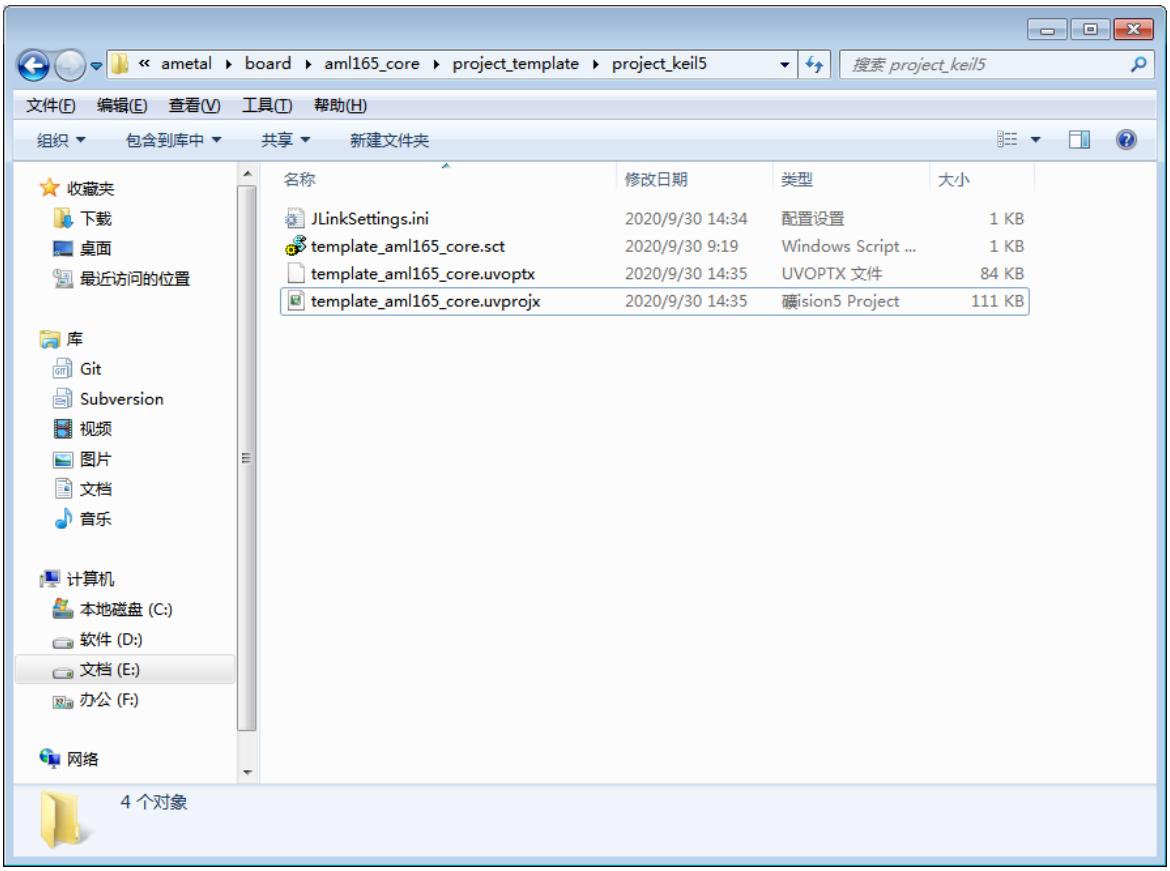

图 2.10　board/aml165_core/project_template/project_keil5 目录结构

2.3.2 Eclipse 工程结构

找到路径：{SDK}\board\aml165_core\project_template\projects_eclipse。

该目录下包含 eclipse 的相关工程文件，如链接脚本文件“template_aml165_core.ld”和调试配置信息文件“template_aml165_core Debug.launch”等。其目录结构如图 2.11 所示。

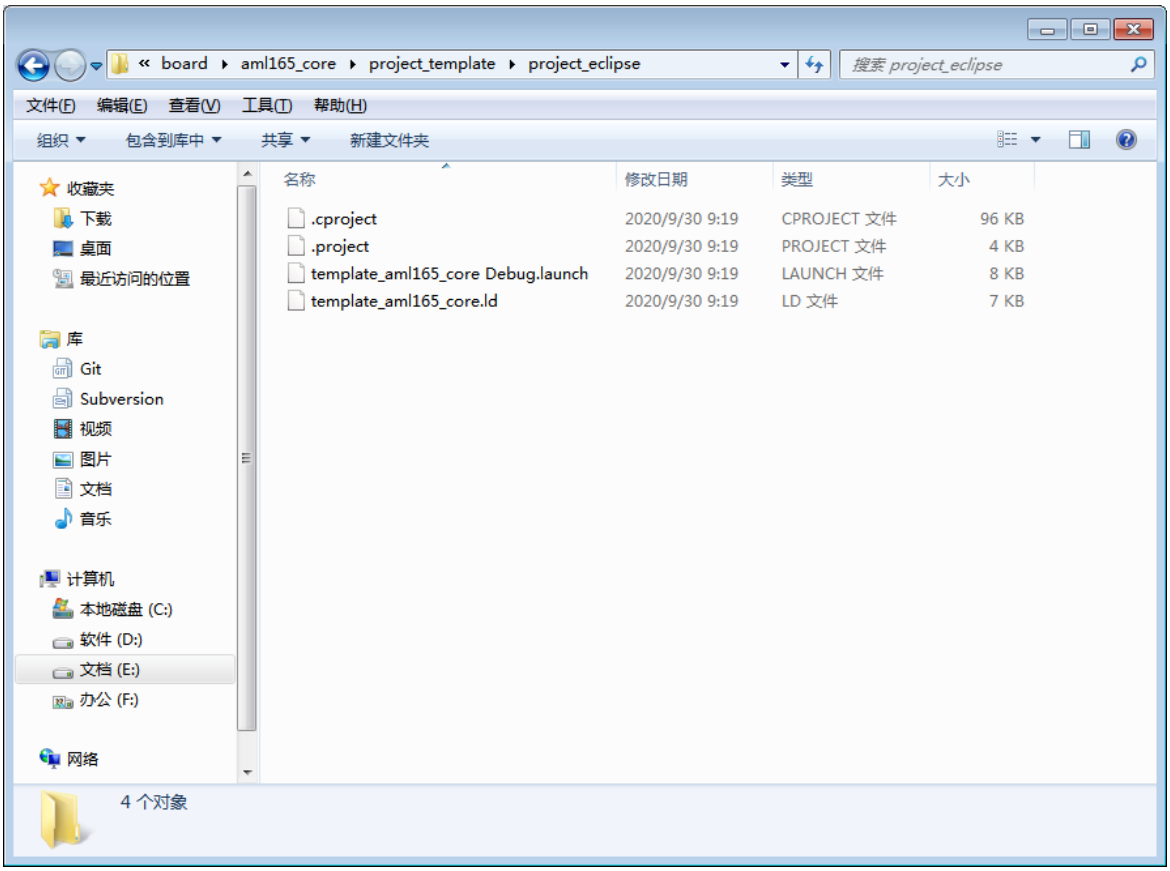

图 2.11　board/aml165_core/project_template/project_eclipse 目录结构

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。（为叙述方便，下文使用 {PROJECT} 表示 ametal\board\aml165_core\project_template 的路径。）

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，详细定义见程序清单 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。

程序清单 3.1 GPIO 自动初始化使能/禁能配置

```
/** \brief 为 1，初始化 GPIO 的相关功能 */
#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、蜂鸣器、按键、调试串口、延时、系统滴答、软件定时器、标准库、中断延时和温度传感器 LM75 等。除 LM75 外（用户使用 LM75 时需自行完成初始化操作，详见第 5.2.6 章），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，详细定义见程序清单 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。

程序清单 3.2 LED 自动初始化使能/禁能配置

```
/**
 * \brief 如果为 1，则初始化 led 的相关功能，板上默认有两个 LED
 *
 * ID: 0 --- PIOB.4 （需要短接跳线帽 J9）
 * ID: 1 --- PIOB.3 （需要短接跳线帽 J10）
 */
#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_BUZZER_ENABLE	蜂鸣器，使能后才能正常使用蜂鸣器
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，默认使用滴答定时器，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{SDK}\board\bsp_common\source\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

ZML165 包含了众多的外设资源，只要 SDK 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_zml165_*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{HWCONFIG}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_zml165_adc.c am_hwconf_zml165_24adc.c
2	时钟	am_hwconf_zml165_clk.c
3	循环冗余校验	am_hwconf_zml165_crc.c
4	DMA	am_hwconf_zml165_dma.c
5	GPIO	am_hwconf_zml165_gpio.c
6	I²C	am_hwconf_zml165_i2c.c
7	I²C 从机	am_hwconf_zml165_i2c_slv.c
8	独立看门狗	am_hwconf_zml165_iwdg.c
9	电源管理	am_hwconf_zml165_pwr.c
10	SPI(DMA 方式)	am_hwconf_zml165_spi_dma.c
11	SPI(中断方式)	am_hwconf_zml165_spi_int.c
12	滴答定时器	am_hwconf_zml165_systick.c
13	标准定时器的捕获功能	am_hwconf_zml165_tim_cap.c
14	标准定时器的 PWM 功能	am_hwconf_zml165_tim_pwm.c
15	标准定时器的定时功能	am_hwconf_zml165_tim_timing.c
16	UART	am_hwconf_zml165_uart.c
17	窗口看门狗	am_hwconf_zml165_wwdg.c
18	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_zml165_gpio.c，可以看到设备实例已经定义好。详见程序清单 4.1。

程序清单 4.1　定义设备实例

```
/** \brief GPIO 设备实例 */
am_zml165_gpio_dev_t __g_gpio_dev;
```

这里使用 am_zml165_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{SDK}\soc\zlg\drivers\include\gpio\am_zml165_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_zml165_gpio.c，可以看到定义的设备信息如程序清单 4.2 所示。

程序清单 4.2　GPIO 设备信息定义

```
/** \brief GPIO 设备信息 */
const am_zml165_gpio_devinfo_t __g_gpio_devinfo = {
    ZML165_GPIO_BASE,           /**< \brief GPIO 控制器寄存器块基址 */
    ZML165_SYSCFG_BASE,         /**< \brief SYSCFG 配置寄存器块基址 */
    ZML165_EXTI_BASE,           /**< \brief 外部事件控制器寄存器块基址 */

    {
        INUM_EXTI0_1,           /**< \brief 外部中断线 0 与线 1 */
        INUM_EXTI2_3,           /**< \brief 外部中断线 2 与线 3 */
        INUM_EXTI4_15           /**< \brief 外部中断线 4 与线 15 */
    },

    PIN_INT_MAX,                /**< \brief GPIO 支持的引脚中断号数量 */
    __g_gpio_infomap,           /**< \brief 引脚触发信息映射 */
    __g_gpio_triginfos,         /**< \brief 引脚触发信息内存 */
    __zml165_plfm_gpio_init,
    __zml165_plfm_gpio_deinit
};
```

这里使用 am_zml165_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{SDK}\soc\zlg\drivers\include\gpio\am_zml165_gpio.h 文件中定义详见程序清单 4.3。

程序清单 4.3　GPIO 设备信息结构体类型定义

```
/**
 * \brief GPIO 设备信息
 */
typedef struct am_zml165_gpio_devinfo {

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

    /** \brief 系统配置 SYSCFG 寄存器块基址 */
    uint32_t        syscfg_regbase;

    /** \brief 外部事件 EXTI 控制寄存器块基址 */
    uint32_t        exti_regbase;

    /** \brief GPIO 引脚中断号列表 */
    const int8_t inum_pin[3];

    /** \brief GPIO 支持的引脚中断号数量 */
    size_t          pint_count;

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

    /** \brief 指向引脚触发信息的指针 */
    struct am_zml165_gpio_trigger_info *p_triginfo;

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

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

} am_zml165_gpio_devinfo_t;
```

可见，共计有 9 个成员，看似很多，这是由于 GPIO 关联的外设较多，GPIO、SYSCFG、EXTI 等都统一归到 GPIO 下管理，仅寄存器基地址就有三个成员。无论怎样，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。

一般来讲，外设关联的寄存器基地址都只有一个，而 GPIO 属于较为特殊的外设，它统一管理了 GPIO、SYSCFG、EXTI 共计 3 个部分的外设，因此，在 GPIO 的设备信息中，需要三个基地址，对应三个成员变量，分别为：gpio_regbase、syscfg_regbase 和 exti_regbase。

寄存器基地址已经在{SDK}\soc\zlg\zml165\zml165_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，详见程序清单 4.4。

程序清单 4.4 外设寄存器基地址定义

```
/** \brief GPIO 基地址 */
#define ZML165_GPIO_BASE                (0x48000000UL)
/** \brief SYSCFG 基地址 */
#define ZML165_SYSCFG_BASE              (0x40010000UL)
/** \brief 外部中断(事件)控制器 EXTI 基地址 */
#define ZML165_EXTI_BASE                (0x40010400UL)
```

可见，列程序清单 4.2 中，设备信息前三个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 3 路中断服务入口给 EXTI，因此，EXTI 共计有 3 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 3 的数组。详见程序清单 4.5。

程序清单 4.5 GPIO 设备信息结构体类型——中断号成员定义

```
/** \brief GPIO 引脚中断号列表 */
const int8_t inum_pin[3];
```

所有中断号已经在 {SDK}\soc\zlg\zml165\zml165_inum.h 文件中定义好了，与 EXTI 相关的中断号定义详见程序清单 4.6。

程序清单 4.6 EXTI 各个中断号定义

```
#define INUM_EXTI0_1            5     /**< \brief EXTI 线[1：0]中断 */
#define INUM_EXTI2_3            6     /**< \brief EXTI 线[3：2]中断 */
#define INUM_EXTI4_15           7     /**< \brief EXTI 线[15：4]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。可见程序清单 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 {SDK}\soc\zlg\zml165\zml165_clk.h 文件中定义好了，详见程序清单 4.7。

程序清单 4.7 时钟 ID 号

```
/* APB1 外设时钟 */
#define CLK_TIM2        (0x01ul << 8 | 0ul)     /**< \brief TIM2 定时器 时钟 */
#define CLK_TIM3        (0x01ul << 8 | 1ul)     /**< \brief TIM3 定时器 时钟 */
```

```
#define CLK_WWDG      (0x01ul << 8 | 11ul)   /**< \brief WWDG 窗口看门狗 时钟 */
#define CLK_SPI2      (0x01ul << 8 | 14ul)   /**< \brief SPI2 时钟 */
#define CLK_UART2     (0x01ul << 8 | 17ul)   /**< \brief UART2 时钟 */
#define CLK_I2C1      (0x01ul << 8 | 21ul)   /**< \brief I2C1 时钟 */
#define CLK_USB       (0x01ul << 8 | 23ul)   /**< \brief USB 时钟 */
#define CLK_CAN       (0x01ul << 8 | 25ul)   /**< \brief CAN 时钟 */
#define CLK_CRS       (0x01ul << 8 | 27ul)   /**< \brief CRS 时钟 */
#define CLK_PWR       (0x01ul << 8 | 28ul)   /**< \brief 电源接口 时钟 */

/* APB2 外设时钟 */
#define CLK_SYSCFG    (0x02ul << 8 | 0ul)    /**< \brief 系统配置寄存器 时钟 */
#define CLK_ADC1      (0x02ul << 8 | 9ul)    /**< \brief ADC1 接口 时钟 */
#define CLK_TIM1      (0x02ul << 8 | 11ul) /**< \brief TIM1 定时器 时钟 */
#define CLK_SPI1      (0x02ul << 8 | 12ul) /**< \brief SPI1 时钟 */
#define CLK_UART1     (0x02ul << 8 | 14ul) /**< \brief UART1 时钟 */
#define CLK_CPT       (0x02ul << 8 | 15ul) /**< \brief 比较器 时钟 */
#define CLK_TIM14     (0x02ul << 8 | 16ul) /**< \brief TIM14 时钟 */
#define CLK_TIM16     (0x02ul << 8 | 17ul) /**< \brief TIM16 时钟 */
#define CLK_TIM17     (0x02ul << 8 | 18ul) /**< \brief TIM17 时钟 */
#define CLK_DBGMCU    (0x02ul << 8 | 22ul) /**< \brief DBGMCU 时钟 */

/* AHB 外设时钟 */
#define CLK_DMA       (0x03ul << 8 | 0ul)     /**< \brief DMA 时钟 */
#define CLK_SRAM      (0x03ul << 8 | 2ul)     /**< \brief SRAM 时钟 */
#define CLK_FLITF     (0x03ul << 8 | 4ul)     /**< \brief FLITF 时钟 */
#define CLK_AES       (0x03ul << 8 | 7ul)     /**< \brief AES 时钟 */
#define CLK_GPIOA     (0x03ul << 8 | 17ul)    /**< \brief GPIOA 时钟 */
#define CLK_GPIOB     (0x03ul << 8 | 18ul)    /**< \brief GPIOB 时钟 */
#define CLK_GPIOC     (0x03ul << 8 | 19ul)    /**< \brief GPIOC 时钟 */
#define CLK_GPIOD     (0x03ul << 8 | 20ul)    /**< \brief GPIOD 时钟 */

/* 其他时钟 */
#define CLK_PLLIN     (0x04ul << 8 | 0ul)    /**< \brief PLL 输入 时钟 */
#define CLK_PLLOUT (0x04ul << 8 | 1ul)    /**< \brief PLL 输出 时钟 */
#define CLK_AHB       (0x04ul << 8 | 2ul)    /**< \brief AHB 时钟 */
#define CLK_APB1      (0x04ul << 8 | 3ul)    /**< \brief APB1 时钟 */
#define CLK_APB2      (0x04ul << 8 | 4ul)    /**< \brief APB2 时钟 */
#define CLK_HSEOSC (0x04ul << 8 | 5ul)    /**< \brief 外部晶振 时钟 */
#define CLK_LSI       (0x04ul << 8 | 6ul)    /**< \brief LSI 时钟 */
#define CLK_HSI       (0x04ul << 8 | 7ul)    /**< \brief HSI 时钟 */
```

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，详见程序清单 4.8。

程序清单 4.8　串口外设时钟 ID 号

```
/** \brief 串口 1 设备信息 */
static const am_zlg_uart_devinfo_t __g_uart1_devinfo = {
    ZML165_UART1_BASE,                  /**< \brief 串口 1 */
    INUM_UART1,                         /**< \brief 串口 1 的中断编号 */
    CLK_UART1,                          /**< \brief 串口 1 的时钟 */

    AMHW_ZLG_UART_DATA_8BIT    |  /**< \brief 8 位数据 */
    AMHW_ZLG_UART_PARITY_NO    |  /**< \brief 无极性 */
    AMHW_ZLG_UART_STOP_1BIT,      /**< \brief 1 个停止位 */

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

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

    NULL,                               /**< \brief USART1 使用 RS485 */
    __zlg_plfm_uart1_init,              /**< \brief USART1 的平台初始化 */
    __zlg_plfm_uart1_deinit,            /**< \brief USART1 的平台去初始化 */
};
```

4. 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、I2C 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，详见程序清单 4.9。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。

程序清单 4.9 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{HWCON-FIG}\am_hwconf_zml165_gpio.c 文件中定义，详见程序清单 4.10。

程序清单 4.10 GPIO 平台初始化函数

```
/** \brief GPIO 平台初始化 */
void __zml165_plfm_gpio_init (void)
{

    /* 使能 GPIO 相关外设时钟 */

    /* 开启 GPIO 各个端口时钟 */
    am_clk_enable(CLK_GPIOA);
    am_clk_enable(CLK_GPIOB);
    am_clk_enable(CLK_GPIOC);
```

```
    am_clk_enable(CLK_GPIOD);

    /* 系统配置时钟使能(等价于 AFIO 时钟) */
    am_clk_enable(CLK_SYSCFG);

    /* 复位 GPIO 相关外设 */
    am_zml165_clk_reset(CLK_GPIOA);
    am_zml165_clk_reset(CLK_GPIOB);
    am_zml165_clk_reset(CLK_GPIOC);
    am_zml165_clk_reset(CLK_GPIOD);
    am_zml165_clk_reset(CLK_SYSCFG);
}
```

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_zml165_clk_reset() 函数用于复位一个外设，在{SDK}\soc\zlg\drivers\source\clk\am_zml165_clk.c 文件中定义。函数原型详见程序清单 4.11。

程序清单 4.11 am_zml165_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_zml165_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{SDK}\soc\zlg\zml165\zml165_clk.h 文件中定义。可见程序清单 4.7。

在平台初始化函数中，参数 CLK_GPIOA 表示复位 GPIOA 外设。

am_clk_enable() 函数用于使能一个外设的时钟，在{SDK}\soc\zlg\drivers\source\clk\am_zml165_clk.c 文件中定义。函数原型详见程序清单 4.12。

程序清单 4.12 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
*    \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{SDK}\soc\zlg\zml165\zml165_clk.h 文件中定义。可见程序清单 4.7。

平台初始化函数中，参数 CLK_GPIOA 和 CLK_SYSCFG 分别使能了 GPIOA 和 SYSCFG 的时钟。

在设备信息结构体赋值时，详见程序清单 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。

5. 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，详见程序清单 4.13。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。

程序清单 4.13 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{HWCON-FIG}\am_hwconf_zml165_gpio.c 文件中定义，详见程序清单 4.14。

程序清单 4.14 GPIO 平台解初始化函数

```
/** \brief GPIO 平台去初始化 */
void __zml165_plfm_gpio_deinit (void)
{

    /* 复位 GPIO 相关外设 */
    am_zml165_clk_reset(CLK_GPIOA);
    am_zml165_clk_reset(CLK_GPIOB);
    am_zml165_clk_reset(CLK_GPIOC);
    am_zml165_clk_reset(CLK_GPIOD);
    am_zml165_clk_reset(CLK_SYSCFG);

    /* 禁能 GPIO 相关外设时钟 */

    /* 禁能 GPIO 各个端口时钟 */
    am_clk_disable(CLK_GPIOA);
    am_clk_disable(CLK_GPIOB);
    am_clk_disable(CLK_GPIOC);
    am_clk_disable(CLK_GPIOD);

    /* 系统配置时钟禁能(等价于 AFIO 时钟) */
    am_clk_disable(CLK_SYSCFG);
```

```
}
```

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{SDK}\soc\zlg\drivers\source\clk\am_zml165_clk.c 文件中定义。函数原型详见程序清单 4.15。

程序清单 4.15 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
*/
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{SDK}\soc\zlg\zml165\zml165_clk.h 文件中定义。可见程序清单 4.7。

在设备信息结构体赋值时，详见程序清单 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 5 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 9 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。

除了常见的 GPIO 设备信息中的这 5 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{SDK}\soc\zlg\drivers\include\gpio\am_zml165_gpio.h 中声明。详见程序清单 4.16。

程序清单 4.16 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_zml165_gpio_init (am_zml165_gpio_dev_t          *p_dev,
                         const am_zml165_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，详见程序清单 4.17。

程序清单 4.17　完成 GPIO 初始化

```
am_zml165_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_zml165_gpio.c 文件中，详见程序清单 4.18。

程序清单 4.18　GPIO 实例初始化函数

```
/** \brief GPIO 实例初始化 */
int am_zml165_gpio_inst_init (void)
{
    return am_zml165_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{SDK}\soc\zlg\drivers\include\gpio\am_zml165_gpio.h 中声明。详见程序清单 4.19。

程序清单 4.19 GPIO 解初始化函数

```
/**
*     \brief GPIO 解初始化
*
*     \param[in] 无
*
*     \return 无
*/
void am_zml165_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，详见程序清单 4.20。

程序清单 4.20 完成 GPIO 解初始化

```
am_zml165_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_zml165_gpio.c 文件中，详见程序清单 4.21。

程序清单 4.21 GPIO 实例解初始化函数

```
/** \brief GPIO 实例解初始化 */
void am_zml165_gpio_inst_deinit (void)
{
    am_zml165_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

ZML165 内置 24 位 ADC，有两个差分输入通道，模拟输入通道信号增益可软件配置，支持宽动态范围信号输入。

下面以 24 位 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_zml165_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、增益倍数选择、输出速率选择、通道设置。

1. ADC 参考电压

ADC 参考电压由具体的电路决定，ADC 参考电压默认为 2.5V，即 2500mV。设备信息中，默认的参考电压即为 2500（单位:mV）。详见程序清单 4.22。

程序清单 4.22 ADC 参考电压默认配置

```
/* 定义 ZML165_ADC 实例信息 */
const am_zml165_adc_devinfo_t __g_zml165_adc_info = {
    {
        AM_ZML165_ADC_PGA_1,
        AM_ZML165_ADC_SPEED_10HZ,
        AM_ZML165_ADC_CHANNEL_A,
        AM_ZML165_ADC_VOUT_ENABLE
    },
    2500
};
```

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 PGA 选择

ADC 增益倍数选择，详见表 4.2。

表 4.2 ADC PGA 选择宏

增益倍数	功能宏
1	AM_ZML165_ADC_PGA_1
2	AM_ZML165_ADC_PGA_2
64	AM_ZML165_ADC_PGA_64
128	AM_ZML165_ADC_PGA_128

3. ADC 输出速率选择

ADC 输出速率选择，详见表 4.3。

表 4.3 ADC 输出速率选择宏

输出速率(Hz)	引脚
10	AM_ZML165_ADC_SPEED_10HZ
40	AM_ZML165_ADC_SPEED_40HZ
640	AM_ZML165_ADC_SPEED_640HZ
1280	AM_ZML165_ADC_SPEED_1280HZ

4. ADC 通道配置

AML165_Core ADC 通道设置，详见表 4.4。

表 4.4 ADC 通道宏

序号	通道	功能宏
1	通道 A	AM_ZML165_ADC_CHANNEL_A
2	通道 B	AM_ZML165_ADC_CHANNEL_B
3	温度	AM_ZML165_ADC_CHANNEL_TEMP
4	内短	AM_ZML165_ADC_CHANNEL_SHORT

4.2.2 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。设备信息详见程序清单 4.23。

程序清单 4.23 时钟默认设备信息

```
/**
 * \brief CLK 设备信息
 */
static const am_zml165_clk_devinfo_t __g_clk_devinfo =
{

    /**
     * \brief 时钟输入频率
     *      若 SYS 时钟为 HSE   此项配置值为用户外部晶振的实际频率（2000000-24000000）
     *      若 SYS 时钟为 HSI   则此配置项作为内部晶振频率选择项，有两种频率可供选择：
     *                          AM_ZML165_HSI_72M、 AM_ZML165_HSI_48M
     */
     AM_ZML165_HSI_48M,

    /** \brief
     *      SYS 时钟源选择
```

```
     *      -# AMHW_ZML165_SYSCLK_HSI_DIV6 : HSI 振荡器  6 分频作为系统时钟
     *      -# AMHW_ZML165_SYSCLK_HSE      : HSE 作为系统时钟
     *      -# AMHW_ZML165_SYSCLK_HSI      : HSI 作为系统时钟
     *      -# AMHW_ZML165_SYSCLK_LSI      : LSI 作为系统时钟
     */
     AMHW_ZML165_SYSCLK_HSI,

    /**
     * \brief AHB 分频系数，AHBCLK = PLLOUT / DIV,AHB 最大频率为 48Mhz
     *
     *      ahb_div |   DIV
     *    ----------------------
     *            0-7 |    1
     *            8   |    2
     *            9   |    4
     *            10  |    8
     *            11  |    16
     *            12  |    64
     *            13  |   128
     *            14  |   256
     *            15  |   512
     */
    0,

    /**
     * \brief APB1 分频系数，APB1CLK = AHBCLK / (2 ^ apb1_div)
     *            APB1 最大频率为 24Mhz
     */
    1,

    /**
     * \brief APB2 分频系数，APB2CLK = AHBCLK / (2 ^ apb2_div)
     *            APB2 最大频率为 48Mhz
     */
    0,

    /* 平台初始化函数，配置时钟引脚等操作 */
    NULL,

    /* CLK 无平台去初始化函数 */
    NULL
};
```

可见，由于默认配置中，主时钟源选择 HSI。分频系数为 1，因此系统时钟频率为

48MHz。

4.2.3 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。

4.2.4 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。

4.2.5 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。

4.2.6 I2C

平台有 1 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_zml165_i2c.c 中的 I²C1 设备信息中 4 个参数配置，详见程序清单 4.24。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

程序清单 4.24 I²C1 速率配置

```
/**
 * \brief I2C1 设备信息
 */
static const am_zlg_i2c_devinfo_t __g_i2c1_devinfo = {

    ZML165_I2C1_BASE,                   /**< \brief I2C1 寄存器块基址 */
    CLK_I2C1,                           /**< \brief 时钟 ID 值 */
    INUM_I2C1,                          /**< \brief I2C1 中断编号 */

    100000,                             /**< \brief I2C 速率 */
    10,                                 /**< \brief 超时值 10 */
    __zlg_i2c1_bus_clean,               /**< \brief 总线恢复函数 */
    __zlg_i2c1_plfm_init,               /**< \brief 平台初始化 */
    __zlg_i2c1_plfm_deinit              /**< \brief 平台去初始化 */
};
```

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{HWCON-FIG}\am_hwconf_zml165_i2c.c 设备信息中第 5 个参数设置，详见程序清单 4.24 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 4.31 。

程序清单 4.25 I²C1 平台初始化函数——引脚配置

```
/** \brief I2C1 平台初始化函数 */
static void __zlg_i2c1_plfm_init (void)
{

    /**
     * PIOA_5 ~ I2C1_SCL, PIOA_4 ~ I2C1_SDA
     */
    am_gpio_pin_cfg(PIOA_5, PIOA_5_I2C_SCL | PIOA_5_AF_OD | PIOA_5_SPEED_20MHz);
    am_gpio_pin_cfg(PIOA_4, PIOA_4_I2C_SDA | PIOA_4_AF_OD | PIOA_4_SPEED_20MHz);

    am_clk_enable(CLK_I2C1);
    am_zml165_clk_reset(CLK_I2C1);
}
```

可见，程序中，将 PIOA_5 配置为 I2C1 的时钟线，PIOA_4 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 6 ，其中 SCL 和 SDA 引脚必须成对使用，I2C1_REMAP0 和 I2C1_REMAP1 虽然都是 I2C1 引脚，但不可交叉使用。

表 4.5 I²C 引脚选择

I2C 总线接口	SCL 引脚	SDA 引脚
I2C1_REMAP0	PIOA_5	PIOA_4
I2C1_REMAP1	PIOD_1	PIOD_0

4.2.7 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 4.2.6。

4.2.8 IWDG

独立看门狗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。

4.2.9 PWR

电源管理，主要是引脚触发信息内存的引脚设置，详见程序清单 4.26。

程序清单 4.26 引脚触发信息内存

```
/** \brief 引脚触发信息内存 */
static struct am_zml165_pwr_mode_init __g_pwr_mode_init[3] = {
```

```
    {AM_ZML165_PWR_MODE_SLEEP,    PIOA_8},
    {AM_ZML165_PWR_MODE_STOP,     PIOA_8},
    {AM_ZML165_PWR_MODE_STANBY,   PIOA_0},
};
```

可见电源的三种模式都储存在这个数组中，设置的相关引脚为 PIOA_8 和 PIOA_0。若想设置为其他引脚，需要在这里修改。

注意： AM_ZML165_PWR_MODE_STANBY 模式只能用 PIOA_0 唤醒。

4.2.10 SPI

平台有 1 个 SPI 总线接口，定义为 SPI1。SPI 可以选择中断方式和 DMA 传输方式，这两种方式的配置略有不同，因此，平台分别提供了这两种方式的配置文件。一般地，只需要配置 SPI 相关引脚即可。

1. 中断方式

以 SPI1 为例，如果使用 SPI 的中断方式，需要在{HWCON-FIG}\am_hwconf_zml165_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：{SDK}\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程序清单 4.27

程序清单 4.27 SPI1 平台初始化函数

```
/** \brief SPI1 平台初始化 */
static void __zlg_plfm_spi1_int_init (void)
{
    am_gpio_pin_cfg(PIOB_3, PIOB_3_SPI1_SCK  | PIOB_3_AF_PP);
    am_gpio_pin_cfg(PIOB_4, PIOB_4_SPI1_MISO | PIOB_4_INPUT_FLOAT);
    am_gpio_pin_cfg(PIOB_5, PIOB_5_SPI1_MOSI | PIOB_5_AF_PP);

    am_clk_enable(CLK_SPI1);
}
```

可见这里 PIOB_3 设置为 SCK，PIOB_4 设置为 MISO，PIOB_5 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 4.6

表 4.6 SPI 可选引脚

SPI 总线接口	SCK	MISO	MOSI
SPI1_REMAP0	PIOB_3	PIOB_4	PIOB_5
SPI1_REMAP1	PIOA_5	PIOB_4	PIOB_5

2. DMA 传输方式

如果需要使用 SPI 的 DMA 方式，需要在{HWCONFIG}\am_hwconf_zml165_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。

4.2.11 Timer

ZML165 有 1 个 16 位通用定时器、1 个 32 位通用定时器、3 个 16 位基本定时器、1 个

16 位高级定时器。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 TIM3 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_zml165_tim_cap.c，TIM3 支持 2 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。详见程序清单 4.28

程序清单 4.28 TIM3 用于捕获功能的设备信息

```
/** \brief TIM3 用于捕获功能的设备信息 */
const am_zlg_tim_cap_devinfo_t __g_tim3_cap_devinfo = {
    ZML165_TIM3_BASE,                   /**< \brief TIM3 寄存器块的基地址 */
    INUM_TIM3,                          /**< \brief TIM3 中断编号 */
    CLK_TIM3,                           /**< \brief TIM3 时钟 ID */
    4,                                  /**< \brief 4 个捕获通道 */
    &__g_tim3_cap_ioinfo_list[0],       /**< \brief 引脚配置信息列表 */
    AMHW_ZLG_TIM_TYPE1,                 /**< \brief 定时器类型 */
    __zlg_plfm_tim3_cap_init,           /**< \brief 平台初始化函数 */
    __zlg_plfm_tim3_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_tim3_cap_ioinfo_list 数组定义，详见程序清单 4.29

程序清单 4.29 TIM3_CAP 各捕获通道相关引脚设置

```
/** \brief TIM3 用于捕获功能的引脚配置信息列表 */
am_zlg_tim_cap_ioinfo_t __g_tim3_cap_ioinfo_list[] = {

    /** \brief 通道 1 */
    {PIOB_4,  PIOB_4_TIM3_CH1 | PIOB_4_INPUT_FLOAT, PIOB_4_GPIO | PIOB_4_INPUT_FLOAT},

    /** \brief 通道 2 */
    {PIOB_5,  PIOB_5_TIM3_CH2 | PIOB_5_INPUT_FLOAT, PIOB_5_GPIO | PIOB_5_INPUT_FLOAT},

};
```

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{SDK}\soc\zlg\drivers\include\tim\am_zlg_tim_cap.h 中定义详见程序清单 4.30

程序清单 4.30 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_zlg_tim_cap_ioinfo {
    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_zlg_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{HWCON-FIG}\am_hwconf_zml165_tim_pwm.c，TIM3 用于 PWM 的设备信息详见程序清单 4.31

程序清单 4.31 TIM3 用于 PWM 功能的设备信息

```
/** \brief TIM3 用于 PWM 设备信息 */
static const am_zlg_tim_pwm_devinfo_t __g_tim3_pwm_devinfo = {
    ZML165_TIM3_BASE,                           /**< \brief TIM3 寄存器块的基地址 */
    CLK_TIM3,                                   /**< \brief TIM3 时钟 ID */
    AM_NELEMENTS(__g_tim3_pwm_chaninfo_list),   /**< \brief 配置输出通道个数 */
    AMHW_ZLG_TIM_PWM_MODE2,                     /**< \brief PWM 输出模式 2 */
    0,                                          /**< \brief PWM 输出高电平有效 */
    &__g_tim3_pwm_chaninfo_list[0],             /**< \brief 通道配置信息列表 */
    AMHW_ZLG_TIM_TYPE1,                         /**< \brief 定时器类型 */
    __zlg_plfm_tim3_pwm_init,                   /**< \brief 平台初始化函数 */
    __zlg_plfm_tim3_pwm_deinit                  /**< \brief 平台解初始化函数 */
};
```

注意：PWM 功能和 CAP 功能在配置引脚时必须按照重映像整组选用，例如，不可以 CH1 选用重映像 0 引脚，CH2 选用重映像 1 引脚。

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_zml165_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。

4.2.12 UART

平台有 1 个 UART 接口，定义为 UART1。在 SDK 提供的驱动中实现 UART 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 UART1 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{SDK}\interface\am_uart.h。

1. 引脚配置

UART1 相关的引脚在配置文件 {HWCONFIG}\am_hwconf_zml165_uart.c 文件中配置，详见程序清单 4.32

程序清单 4.32 UART 引脚初始化函数

```
/** \brief 串口 1 平台初始化 */
static void __zlg_plfm_uart1_init (void)
{
    /* 引脚初始化 PIOB_3_UART1_TX   PIOB_4_UART1_RX */
    am_gpio_pin_cfg(PIOB_3, PIOB_3_UART1_TX | PIOB_3_AF_PP);
    am_gpio_pin_cfg(PIOB_4, PIOB_4_UART1_RX | PIOB_4_INPUT_FLOAT);
}
```

程序中将 PIOB_3 作为 UART1 的发送引脚，PIOB_4 作为 UART1 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 4.7，引脚选用必须成对选用，不可交叉选用。

表 4.7 UART 引脚选择示例

UART 接口	TXD 功能引脚	RXD 功能引脚
UART1_REMAP0	PIOB_3	PIOB_4

2. 配置波特率

在 UART1 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，详见程序清单 4.33。

程序清单 4.33 UART 设备信息结构体类型

```
/** \brief 串口 1 设备信息 */
static const am_zlg_uart_devinfo_t __g_uart1_devinfo = {
    ZML165_UART1_BASE,                  /**< \brief 串口 1 */
    INUM_UART1,                         /**< \brief 串口 1 的中断编号 */
    CLK_UART1,                          /**< \brief 串口 1 的时钟 */

    AMHW_ZLG_UART_DATA_8BIT    |    /**< \brief 8 位数据 */
    AMHW_ZLG_UART_PARITY_NO    |    /**< \brief 无极性 */
    AMHW_ZLG_UART_STOP_1BIT,        /**< \brief 1 个停止位 */

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

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

    NULL,                           /**< \brief USART1 使用 RS485 */
    __zlg_plfm_uart1_init,          /**< \brief USART1 的平台初始化 */
    __zlg_plfm_uart1_deinit,        /**< \brief USART1 的平台去初始化 */
};
```

4.2.13 WWDG

窗口看门狗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。

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

4.3.2 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_aml165_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_aml165_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 4.8

表 4.8 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_zml165_nvic_inst_init (void);
2	ADC	am_zml165_adc_handle_t am_zml165_24adc_inst_init (void);
3	CLK	int am_zml165_clk_inst_init (void);
4	CRC	am_crc_handle_t am_zml165_crc_inst_init (void);
5	DMA	int am_zml165_dma_inst_init (void);
6	GPIO	int am_zml165_gpio_inst_init (void);
7	I2C1	am_i2c_handle_t am_zml165_i2c1_inst_init (void);
8	I2C1_SLV	am_i2c_slv_handle_t am_zml165_i2c1_slv_inst_init (void);
9	IWDG	am_wdt_handle_t am_zml165_iwdg_inst_init (void);
10	PWR	am_zml165_pwr_handle_t am_zml165_pwr_inst_init (void);
11	SPI1(DMA 方式)	am_spi_handle_t am_zml165_spi1_dma_inst_init (void);

续上表

序号	外设	实例初始化函数原型
12	SPI1(中断方式)	am_spi_handle_t am_zml165_spi1_int_inst_init (void);
13	SYSTICK	am_timer_handle_t am_zml165_systick_inst_init (void);
14	TIM1_CAP	am_cap_handle_t am_zml165_tim1_cap_inst_init (void);
15	TIM2_CAP	am_cap_handle_t am_zml165_tim2_cap_inst_init (void);
16	TIM3_CAP	am_cap_handle_t am_zml165_tim3_cap_inst_init (void);

17	TIM14_CAP	am_cap_handle_t am_zml165_tim14_cap_inst_init (void);
18	TIM1_PWM	am_pwm_handle_t am_zml165_tim1_pwm_inst_init (void);
19	TIM2_PWM	am_pwm_handle_t am_zml165_tim2_pwm_inst_init (void);
20	TIM3_PWM	am_pwm_handle_t am_zml165_tim3_pwm_inst_init (void);
21	TIM14_PWM	am_pwm_handle_t am_zml165_tim14_pwm_inst_init (void);
22	TIM1_TIMING	am_timer_handle_t am_zml165_tim1_timing_inst_init (void);
23	TIM2_TIMING	am_timer_handle_t am_zml165_tim2_timing_inst_init (void);
24	TIM3_TIMING	am_timer_handle_t am_zml165_tim3_timing_inst_init (void);
25	TIM14_TIMING	am_timer_handle_t am_zml165_tim14_timing_inst_init (void);
26	TIM16_TIMING	am_timer_handle_t am_zml165_tim16_timing_inst_init (void);
27	TIM17_TIMING	am_timer_handle_t am_zml165_tim17_timing_inst_init (void);
28	UART1	am_uart_handle_t am_zml165_uart1_inst_init (void);
29	WWDG	am_wdt_handle_t am_zml165_wwdg_inst_init (void);

4.3.3 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_zml165_*_handle_t ，类型由驱动头文件自定义），如 am_zml165_pwr_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 4.9。

表 4.9 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_zml165_clk_inst_init (void);
2	DMA	int am_zml165_dma_inst_init (void);
3	GPIO	int am_zml165_gpio_inst_init (void);
4	NVIC 中断	int am_zml165_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{SDK}\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。简单示例如程序清单 4.34。

程序清单 4.34 GPIO 标准接口使用范例

```
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 {SDK}\documents\《AMetal API 参考手册.chm》 或者 {SDK}\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 {SDK}\soc\zlg\drivers\include\dma\am_zlg_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 3.2。

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 4.10。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 4.10 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_zml165_adc_handle_t am_zml165_24adc_inst_init (void);
2	CRC	am_crc_handle_t am_zml165_crc_inst_init (void);
3	I2C1	am_i2c_handle_t am_zml165_i2c1_inst_init (void);
4	I2C1_SLV	am_i2c_slv_handle_t am_zml165_i2c1_slv_inst_init (void);
5	IWDG	am_wdt_handle_t am_zml165_iwdg_inst_init (void);
6	SPI1(DMA 方式)	am_spi_handle_t am_zml165_spi1_dma_inst_init (void);
7	SPI1(中断方式)	am_spi_handle_t am_zml165_spi1_int_inst_init (void);
8	SYSTICK	am_timer_handle_t am_zml165_systick_inst_init (void);
9	TIM1_CAP	am_cap_handle_t am_zml165_tim1_cap_inst_init (void);

续上表

序号	外设	实例初始化函数原型
10	TIM2_CAP	am_cap_handle_t am_zml165_tim2_cap_inst_init (void);
11	TIM3_CAP	am_cap_handle_t am_zml165_tim3_cap_inst_init (void);
12	TIM14_CAP	am_cap_handle_t am_zml165_tim14_cap_inst_init (void);
13	TIM1_PWM	am_pwm_handle_t am_zml165_tim1_pwm_inst_init (void);
14	TIM2_PWM	am_pwm_handle_t am_zml165_tim2_pwm_inst_init (void);
15	TIM3_PWM	am_pwm_handle_t am_zml165_tim3_pwm_inst_init (void);
16	TIM14_PWM	am_pwm_handle_t am_zml165_tim14_pwm_inst_init (void);
17	TIM1_TIMING	am_timer_handle_t am_zml165_tim1_timing_inst_init (void);
18	TIM2_TIMING	am_timer_handle_t am_zml165_tim2_timing_inst_init (void);
19	TIM3_TIMING	am_timer_handle_t am_zml165_tim3_timing_inst_init (void);
20	TIM14_TIMING	am_timer_handle_t am_zml165_tim14_timing_inst_init (void);
21	TIM16_TIMING	am_timer_handle_t am_zml165_tim16_timing_inst_init (void);
22	TIM17_TIMING	am_timer_handle_t am_zml165_tim17_timing_inst_init (void);
23	UART1	am_uart_handle_t am_zml165_uart1_inst_init (void);
24	WWDG	am_wdt_handle_t am_zml165_wwdg_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。详见程序清单 4.35。

程序清单 4.35　ADC 简单操作示例

```
#include "ametal.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_zml165_adc.h"
#include "demo_zlg_entries.h"

/**
 * \brief ZML165 ADC 电压测试例程
 */

int am_main (void)
{
    am_zml165_adc_handle_t handle = am_zml165_24adc_inst_init();
    uint8_t    gpa[4] = {1, 2, 64, 128};
    am_zml165_adc_handle_t    handle = (am_zml165_adc_handle_t)p_handle;
    am_zml165_adc_config_t    config;

    config.pga = AM_ZML165_ADC_PGA_1;
    config.speed = AM_ZML165_ADC_SPEED_1280HZ;
    config.channel = AM_ZML165_ADC_CHANNEL_A;
    config.refo_off = AM_ZML165_ADC_VOUT_DISABLE;

    am_zml165_adc_config_load(handle, &config);

    while(1){
        uint8_t i = 0;
        int32_t adc_val[10];
        float     vol = 0;
        volatile uint32_t time = am_sys_tick_get();
        am_adc_read(&handle->adc_serve, 0, (void *)adc_val, AM_NELEMENTS(adc_val));
        time = am_sys_tick_get() - time;
        /* 丢弃前四个数据以保证设置生效后数据建立时间 */
        for(i = 4 ; i < AM_NELEMENTS(adc_val); i++){
            if(adc_val[i] >= 0x800000)    {
                adc_val[i] &= 0x7fffff;
                adc_val[i] |= 0xff800000;
```

```
            }
            vol += (adc_val[i] / ((double)AM_NELEMENTS(adc_val) - 4));
        }

        vol = (double)((double)(vol    / ((1 << 24) -1)) * handle->p_devinfo->vref);
        vol *= 10000;

        vol /= gpa[config.pga];

        if(vol > 0){
            am_kprintf("Voltage is %d.%04d mV\r\n", (int32_t)vol/10000, (int32_t)vol%10000);
        }else {
            vol *= -1;
            am_kprintf("Voltage is    -%d.%04d mV\r\n", (int32_t)vol/10000, (int32_t)vol%10000);
        }
        am_mdelay(500);
    }
}
```

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 4.11。可见，仅 PWR 外设提供了一个该类型的实例初始化函数。

表 4.11 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	PWR	am_zml165_pwr_handle_t am_zml165_pwr_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。

4.3.4 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_aml165_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_aml165_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 4.12

表 4.12 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_zml165_nvic_inst_deinit (void);
2	ADC	void am_zml165_24adc_inst_deinit (am_zml165_adc_handle_t handle);
3	CRC	void am_zml165_crc_inst_deinit (am_crc_handle_t handle);
4	DMA	void am_zml165_dma_inst_deinit (void);
5	GPIO	void am_zml165_gpio_inst_deinit (void);
6	I2C1	void am_zml165_i2c1_inst_deinit (am_i2c_handle_t handle);

7	I2C1_SLV	void am_zml165_i2c1_slv_inst_deinit (am_i2c_slv_handle_t handle);
8	IWDG	void am_zml165_iwdg_inst_deinit (am_wdt_handle_t handle);

序号	外设	实例初始化函数原型
9	PWR	void am_zml165_pwr_inst_deinit (void);
10	SPI1(DMA 方式)	void am_zml165_spi1_dma_inst_deinit (am_spi_handle_t handle);
11	SPI1(中断方式)	void am_zml165_spi1_int_inst_deinit (am_spi_handle_t handle);
12	SYSTICK	void am_zml165_systick_inst_deinit (am_timer_handle_t handle);
13	TIM1_CAP	void am_zml165_tim1_cap_inst_deinit (am_cap_handle_t handle);
14	TIM2_CAP	void am_zml165_tim2_cap_inst_deinit (am_cap_handle_t handle);
15	TIM3_CAP	void am_zml165_tim3_cap_inst_deinit (am_cap_handle_t handle);
16	TIM14_CAP	void am_zml165_tim14_cap_inst_deinit (am_cap_handle_t handle);
17	TIM1_PWM	void am_zml165_tim1_pwm_inst_deinit (am_pwm_handle_t handle);
18	TIM2_PWM	void am_zml165_tim2_pwm_inst_deinit (am_pwm_handle_t handle);
19	TIM3_PWM	void am_zml165_tim3_pwm_inst_deinit (am_pwm_handle_t handle);
20	TIM14_PWM	void am_zml165_tim14_pwm_inst_deinit (am_pwm_handle_t handle);
21	TIM1_TIMING	void am_zml165_tim1_timing_inst_deinit (am_timer_handle_t handle);
22	TIM2_TIMING	void am_zml165_tim2_timing_inst_deinit (am_timer_handle_t handle);
23	TIM3_TIMING	void am_zml165_tim3_timing_inst_deinit (am_timer_handle_t handle);
24	TIM14_TIMING	void am_zml165_tim14_timing_inst_deinit (am_timer_handle_t handle);
25	TIM16_TIMING	void am_zml165_tim16_timing_inst_deinit (am_timer_handle_t handle);
26	TIM17_TIMING	void am_zml165_tim17_timing_inst_deinit (am_timer_handle_t handle);
27	UART1	void am_zml165_uart1_inst_deinit (am_uart_handle_t handle);
28	WWDG	void am_zml165_wwdg_inst_deinit (am_wdt_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。

4.3.5 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{SDK}\soc\zlg\drivers\include\spi\hw\amhw_zlg_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，详见程序清单 4.36。

程序清单 4.36 SPI 硬件层操作函数

```
/**
 * \brief RX enable
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag     : TRUE or FALSE
 * \return none
 */
am_static_inline
void amhw_zlg_spi_rx_enable (amhw_zlg_spi_t *p_hw_spi, am_bool_t flag)
{
    p_hw_spi->gctl = (p_hw_spi->gctl & (~(1u << 4))) | (flag << 4);
}

/**
 * \brief TX enable
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag     : TRUE or FALSE
 *
 * \return none
 */
am_static_inline
void amhw_zlg_spi_tx_enable (amhw_zlg_spi_t *p_hw_spi, am_bool_t flag)
{
    p_hw_spi->gctl = (p_hw_spi->gctl & (~(1u << 3))) | (flag << 3);
}
```

其它一些函数读者可自行打开{SDK}**{SDK}\soc\zlg\drivers\include\spi\hw\amhw_zlg_spi.h 文件查看。这些函数均是以 amhw_zlg_spi_t 类型作为第一个参数。amhw_zlg_spi_t 类型在{SDK}\soc\zlg\drivers\include\spi\hw\amhw_zlg_spi.h 文件中定义，用于定义出 SPI 外设的各个寄存器。详见程序清单 4.37。

程序清单 4.37 SPI 寄存器结构体定义

```
/**
 * \brief SPI structure of register
 */
typedef struct amhw_zlg_spi {
    __IO uint32_t   txreg;          /**< \brief SPI TX data register */
    __I  uint32_t   rxreg;          /**< \brief SPI RX data register */
    __I  uint32_t   cstat;          /**< \brief SPI current status register */
    __I  uint32_t   intstat;        /**< \brief SPI interrupt status register */
    __IO uint32_t   inten;          /**< \brief SPI interrupt enable register */
    __O  uint32_t   intclr;         /**< \brief SPI interrupt clear register */
    __IO uint32_t   gctl;           /**< \brief SPI global control */
```

```
    __IO uint32_t    cctl;          /**< \brief SPI common control */
    __IO uint32_t    spbrg;         /**< \brief SPI baud rate register */
    __IO uint32_t    rxdnr;         /**< \brief SPI RX data number */
    __IO uint32_t    nssr;          /**< \brief SPI slave selection register */
    __IO uint32_t    extctl;        /**< \brief SPI data control */
} amhw_zlg_spi_t;
```

该类型的指针已经在{SDK}\soc\zlg\zml165\zml165_periph_map.h 文件中定义，应用程序可以直接使用。详见程序清单 4.38

程序清单 4.38　SPI 寄存器结构体指针定义

```
/** \brief SPI1 寄存器块指针 */
#define ZML165_SPI1      ((amhw_zlg_spi_t                *)ZML165_SPI1_BASE)
```

注解：其中的 ZML165_SPI1_BASE 是 SPI1 外设寄存器的基地址，在{SDK}\soc\zlg\zml165\zml165_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 ZML165_SPI1 ，则表示操作的是 SPI1。如程序清单 4.39 所示，用于使能 SPI1。

程序清单 4.39　使能 SPI1

```
amhw_zlg_spi_module_enable(ZML165_SPI1,TRUE);
```

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，要使能 SPI1，也可以直接设置寄存器的值，详见程序清单 4.40

程序清单 4.40　直接设置寄存器的值使能 SPI1

```
/*
* 设置 CR1 寄存器的 bit6 为 1，使能 SPI1
*/
ZML165_SPI1->gctl |= (1u << 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 {SDK}\soc\zlg\zml165\zml165_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程序清单 4.41。

程序清单 4.41　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	PWR	ZML165_PWR
2	BKP	ZML165_BKP
3	SYSCFG	ZML165_SYSCFG
4	GPIOA	ZML165_GPIOA
5	GPIOB	ZML165_GPIOB

6	GPIOC	ZML165_GPIOC
7	GPIOD	ZML165_GPIOD
8	GPIO	ZML165_GPIO
9	TIM1	ZML165_TIM1
10	TIM2	ZML165_TIM2
11	TIM3	ZML165_TIM3
12	TIM14	ZML165_TIM14
13	TIM16	ZML165_TIM16
14	TIM17	ZML165_TIM17
15	I2C	ZML165_I2C
16	IWDG	ZML165_IWDG
17	WWDG	ZML165_WWDG
18	UART	ZML165_UART1
19	ADC	ZML165_ADC
20	SPI	ZML165_SPI1
21	DMA	ZML165_DMA
22	COMP	ZML165_CMP
23	FLASH	ZML165_FLASH
24	RCC	ZML165_RCC
25	EXTI	ZML165_EXTI

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、蜂鸣器、按键、调试串口、温度传感器 LM75、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	蜂鸣器	am_hwconf_buzzer.c
4	温度传感器 (LM75)	am_hwconf_lm75.c
5	调试串口	am_hwconf_debug_uart.c
6	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}\user_config\am_board.c 中自动完成初始化。以 LED 为例，初始化代码详见程序清单 5.1

程序清单 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

AML165_Core 板上有两个 LED 灯， 默认引脚分别为 PIOB_4 和 PIOB_3， 使用时，需要使用跳线帽短接 AML165_Core 板上的 J9 和 J10。LED 相关信息定义在

{SDK}\user_config\am_hwconf_usrcfg\am_hwconf_led_gpio.c 文件中，详见程序清单 5.2。

程序清单 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[] = {PIOB_4, PIOB_3};

/* 定义 GPIO 按键实例信息 */
static const am_led_gpio_info_t __g_led_gpio_info = {
    {
        0,                                  /* 起始编号 0 */
        1                                   /* 结束编号 1，共计 2 个 LED */
    },
    __g_led_pins,
    AM_TRUE
};
```

其中，am_led_gpio_info_t 类型在{SDK}\components\drivers\include\am_led_gpio.h 文件中定义，详见程序清单 5.3。

程序清单 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 zml165_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 PIOB_4 和 PIOB_3 ，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{SDK}\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 PIOB_4 和 PIOB_3，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 蜂鸣器配置

板载蜂鸣器为无源蜂鸣器，需要使用 PWM 驱动才能实现发声。默认使用 TIM2 的输

出通道1（TIM2_CH1）输出PWM。可以通过{PROJECT}\user_config\am_hwconf_usrcfg\am_hwconf_buzzer.c 文件中的两个相关宏来配置PWM 的频率和占空比，相应宏名及含义详见表 5.2。

表 5.2 蜂鸣器配置相关宏

宏名	含义
____BUZZER_PWM_FREQ	PWM 的频率，默认为 2.5KHz
____BUZZER_PWM_DUTY	PWM 的占空比，默认为50（即50%）

注解：由于蜂鸣器使用了 TIM2 的 PWM 功能，若应用程序需要使用 TIM2，建议通过使能/禁能宏禁止蜂鸣器的使用，以免冲突。

5.2.3 按键

AML165_Core 有两个板载按键 KEY/RES 和 RST，KEY/ RES 的默认引脚为 PIOA_13，使用时， 需要使用跳线帽短接 AML165_Core 板上的 J14 和 J8。其中 RST 为复位按键， 可供使用的按键只有 KEY/RES。KEY 相关信息定义在{SDK}\user_config\am_hwconf_usrcfg\am_hwconf_key_gpio.c 文件中， 详见程序清单 5.4 KEY 相关配置信息。

程序清单 5.4 KEY 相关配置信息

```
static const int __g_key_pins[]   = {PIOA_13};
static const int __g_key_codes[] = {KEY_KP0};

/* 定义 GPIO 按键实例信息 */
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}\components\drivers\include\am_key_gpio.h 文件中定义，详见程序清单 5.5。

程序清单 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 zml165_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，KEY/RES 对应 PIOA_13 ，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 PIOA_13，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。

5.2.4 调试串口配置

AML165_Core 具有 1 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 5.3。

表 5.3 调试串口相关配置

宏名	含义
____DEBUG_UART	串口号，1-UART1
____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）。

5.2.5 系统滴答和软件定时器配置

系统滴答需要 SYSTICK 定时器为其提供一个周期性的定时中断，默认使用 SYSTICK 定时器。 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。详细定义见程序清单 5.6。

程序清单 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_softimer.c 文件中的 SYSTEM_TICK_RATE 宏来设置运行频率，默认 1KHz。详细定义见程序清单 5.6。

注解： 使用软件定时器时必须开启系统滴答。

5.2.6 温度传感器 LM75

AML165_Core 自带一个 LM75B 测温芯片， 使用 LM75 传感器需要配置{PROJECT}\user_config\am_hwconf_usrcfg\am_hwconf_lm75.c 文件中 LM75 的实例信息__g_temp_lm75_info ， __g_temp_lm75_info 存放的是 I2C 从机地址，详细定义见程序清单 5.7。

程序清单 5.7 LM75 地址配置

```
/** \brief 设备信息 */
static const am_temp_lm75_info_t __g_temp_lm75_info = {
    0x48
};
```

LM75 没有相应的使能/禁能宏，配置完成后，用户需要自行调用实例初始化函数获得温度标准服务操作句柄，通过标准句柄获取温度值。

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 3.1，使用方法可以参考 4.3.2

表 5.4 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	蜂鸣器	am_pwm_handle_t am_buzzer_inst_init (void);
4	温度传感器 (LM75)	am_temp_handle_t am_temp_lm75_inst_init (void);
5	调试串口	am_uart_handle_t am_debug_uart_inst_init (void);
6	系统滴答	am_timer_handle_t am_system_tick_inst_init (void);
7	系统滴答和软件定时器	am_timer_handle_t am_system_tick_softimer_inst_init (void);

6. MicroPort 系列扩展板

为了便于扩展开发板功能，ZLG 制定了 MicroPort 接口标准，MicroPort 是一种专门用于扩展功能模块的硬件接口，其有效地解决了器件与 MCU 之间的连接和扩展。其主要功能特点如下：

- 具有标准的接口定义；
- 接口包括丰富的外设资源，支持 UART、I2C、SPI、PWM、ADC 和 DAC 功能；
- 配套功能模块将会越来越丰富；
- 支持上下堆叠扩展。

ZML165 板载 1 路带扩展的 MicroPort 接口，如图 6.1 所示。用户可以依据需求，选择或开发功能多样的 MicroPort 模块，快速灵活地搭建原型机。由于 ZML165 片上资源有限，还有极少部分 MicroPort 接口定义的引脚功能不支持，其相应的引脚可以当做普通 I/O 使用。

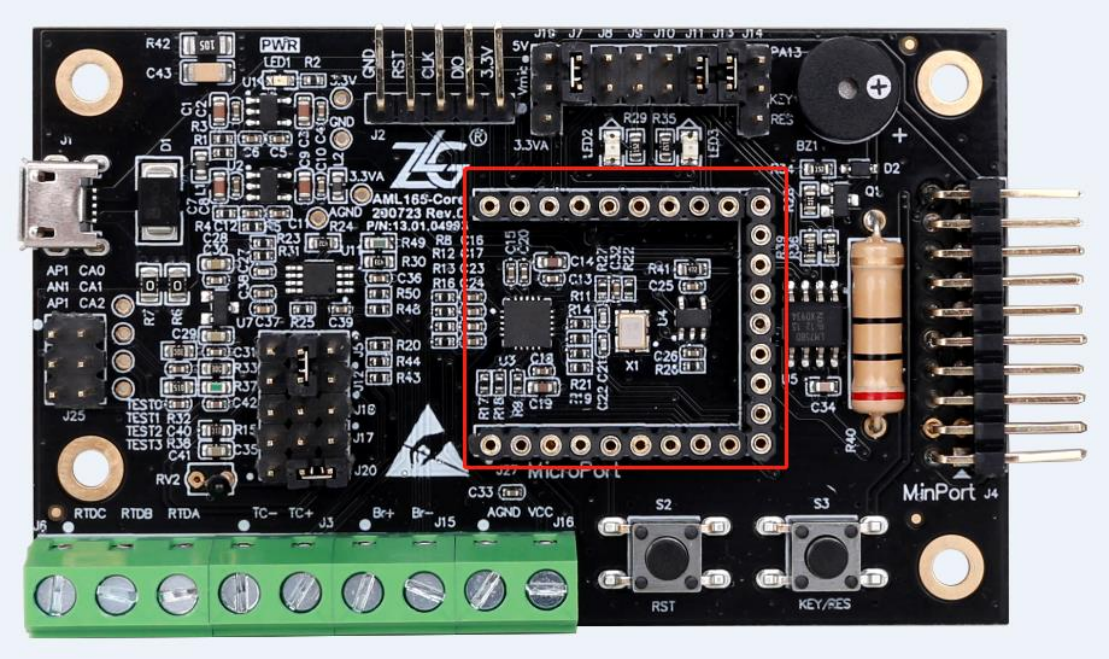

图 6.1 ZML165 MicroPort

6.1 配置文件结构

当前可用的 MicroPort 扩展板有：MicroPort-DS1302、MicroPort-EEPROM、MicroPort-FLASH、MicroPort-RS485、MicroPort-RTC 和 MicroPort-RX8025T，与 MicroPort 相关的配置由{PROJECT}\user_config\am_hwconf_usrcfg 下的一组 am_hwconf_microport_开头的.c 文件完成的，通常情况下不需要用户自己修改，详见表 6.1。MicroPort 扩展板的配置文件与片上外设配置文件结构基本类似。但是，MicroPort 扩展板的配置文件中不提供实例解初始化函数。

表 6.1 MicroPort 对应的配置文件

序号	外设	配置文件

1	MicroPort-DS1302	am_hwconf_microport_ds1302.c
2	MicroPort-EEPROM	am_hwconf_microport_eeprom.c
3	MicroPort-FLASH	am_hwconf_microport_flash.c
4	MicroPort-RS485	am_hwconf_microport_rs485.c
5	MicroPort-RTC	am_hwconf_microport_rtc.c
6	MicroPort-RX8025T	am_hwconf_microport_rx8025t.c

6.2 使用方法

MicroPort 扩展板对应的实例初始化函数的原型详见表 6.2。使用方法可以参考 4.3.2，也可以参考 {SDK}\examples\board\am165_core\microport_board 目录下的例程。

表 6.2 MicroPort 扩展板实例初始化函数

序号	外设	实例初始化函数原型
1	MicroPort-DS1302(使用芯片特殊功能)	am_ds1302_handle_t am_microport_ds1302_inst_init (void);
2	MicroPort-DS1302(使用通用的 RTC 功能)	am_rtc_handle_t am_microport_ds1302_rtc_inst_init (void);
3	MicroPort-DS1302(用作系统时间)	int am_microport_ds1302_time_inst_init (void);
4	MicroPort-EEPROM(使用 EP24CXX 标准接口)	am_ep24cxx_handle_t am_microport_eeprom_inst_init (void);
5	MicroPort-EEPROM(用作标准的 NVRAM 器件)	int am_microport_eeprom_nvram_inst_init (void);
6	MicroPort-FLASH(使用 MX25XX 标准接口)	am_mx25xx_handle_t am_microport_flash_inst_init (void);
7	MicroPort-FLASH(使用 MTD 标准接口)	am_mtd_handle_t am_microport_flash_mtd_inst_init (void);
8	MicroPort-FLASH(使用 FTL 标准接口)	am_ftl_handle_t am_microport_flash_ftl_inst_init (void);
9	MicroPort-RS485	am_uart_handle_t am_microport_rs485_inst_init (void);
10	MicroPort-RTC(使用芯片特殊功能)	am_pcf85063_handle_t am_microport_rtc_inst_init (void);
11	MicroPort-RTC(使用通用的 RTC 功能)	am_rtc_handle_t am_microport_rtc_rtc_inst_init (void);
12	MicroPort-RTC(使用通用的闹钟功能)	am_alarm_clk_handle_t am_microport_rtc_alarm_clk_inst_init (void);
13	MicroPort-RTC(用作系统时间)	int am_microport_rtc_time_inst_init (void);
14	MicroPort-RX8025T(使用芯片特殊功能)	am_rx8025t_handle_t am_microport_rx8025t_inst_init (void);
15	MicroPort-RX8025T(使用通用的 RTC 功能)	am_rtc_handle_t am_microport_rx8025t_rtc_inst_init (void);
16	MicroPort-RX8025T(使用通用的闹钟功能)	am_alarm_clk_handle_t am_microport_rx8025t_alarm_clk_inst_init (void);
17	MicroPort-RX8025T(用作系统时间)	int am_microport_rx8025t_time_inst_init (void);

7. MiniPort 系列扩展板

MiniPort 接口是一个通用板载标准硬件接口，通过该接口可以与配套的标准模块相连，便于进一步简化硬件设计和扩展。其特点如下：

- 采用标准的接口定义，采用 2x10 间距 2.54mm 的 90° 弯针；
- 可同时连接多个扩展接口模块；
- 具有 16 个通用 I/O 端口；
- 支持 1 路 SPI 接口；
- 支持 1 路 I2C 接口；
- 支持 1 路 UART 接口；
- 支持 1 路 3.3V 和 1 路 5V 电源接口。

ZML165 开发板搭载了 1 路 MiniPort，接口标号 J4，如图 7.1 所示。

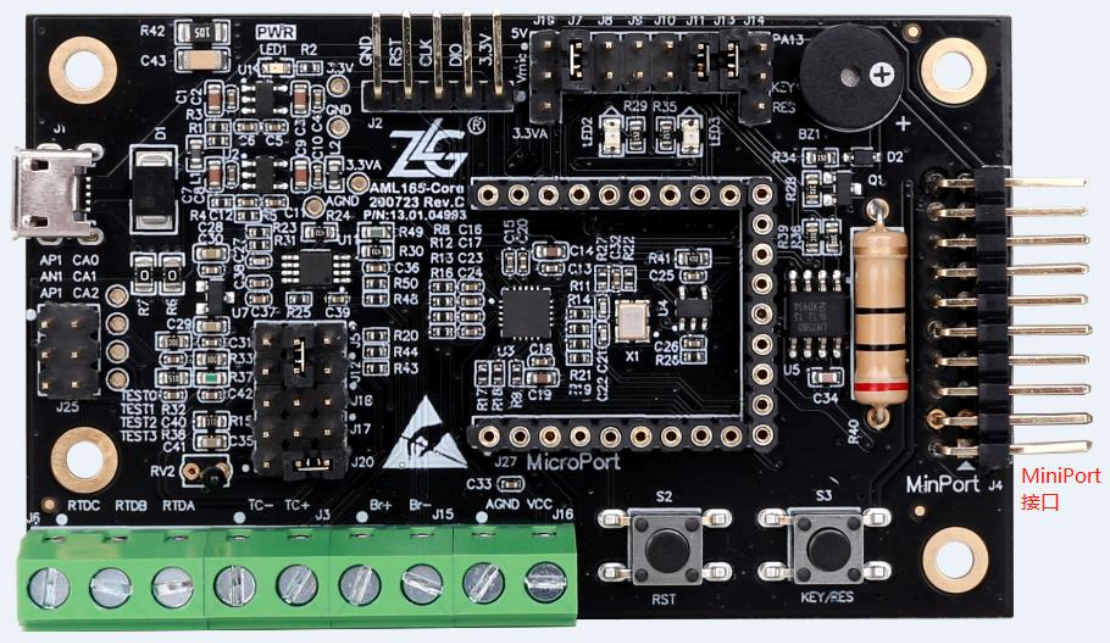

图 7.1 ZML165 MiniPort

7.1 配置文件结构

当前可用的 MiniPort 扩展板有：MiniPort-595、MiniPort-KEY、MiniPort-LED、MiniPort-VIEW 和 MiniPort-ZLG72128，与 MiniPort 相关的配置由{PROJECT}\user_config\am_hwconf_usrcfg 下的一组 am_hwconf_miniport_开头的.c 文件完成的，通常情况下不需要用户自己修改，详见表 7.1。MiniPort 扩展板的配置文件与片上外设配置文件结构基本类似。但是，MiniPort 扩展板的配置文件中不提供实例解初始化函数。

表 7.1 MiniPort 对应的配置文件

序号	外设	配置文件
1	MiniPort-595	am_hwconf_miniport_595.c

2	MiniPort-KEY	am_hwconf_miniport_key.c
3	MiniPort-LED	am_hwconf_miniport_led.c
4	MiniPort-VIEW	am_hwconf_miniport_view.c
5	MiniPort-VIEW KEY	am_hwconf_miniport_view_key.c
6	MiniPort-ZLG72128	am_hwconf_miniport_zlg72128.c

7.2 使用方法

MiniPort 扩展板对应的实例初始化函数的原型详见表 7.2。使用方法可以参考 4.3.2，也可以参考{SDK}\examples\board\am165_core\miniport_board 目录下的例程。

表 7.2 MiniPort 配板实例初始化函数

序号	外设	实例初始化函数原型
1	MiniPort-595	am_hc595_handle_t am_miniport_595_inst_init (void);
2	MiniPort-KEY	int am_miniport_key_inst_init (void);
3	MiniPort-LED	int am_miniport_led_inst_init (void);
4	MiniPort-LED(LED 595 联合初始化)	int am_miniport_led_595_inst_init (void);
5	MiniPort-View	int am_miniport_view_inst_init (void);
6	MiniPort-View(View 595 联合初始化)	int am_miniport_view_595_inst_init (void);
7	MiniPort-View KEY(View KEY 联合初始化)	int am_miniport_view_key_inst_init (void);
8	MiniPort-View KEY(View 595 KEY 联合初始化)	int am_miniport_view_key_595_inst_init (void);
9	MiniPort-ZLG72128	int am_miniport_zlg72128_inst_init (void);

MiniPort 扩展板通过排母与 ZML165 开发板相连，同时采用排针将所有引脚引出，实现模块的横向堆叠。例如实例初始化函数 int am_miniport_view_595_inst_init (void); 可以初始化 MiniPort-View 和 MiniPort-595，初始化成功之后能够通过 MiniPort-595 驱动 MiniPort-View。

8. 免责声明

本着为用户提供更好服务的原则，广州致远微电子有限公司（下称“致远微电子”）在本手册中将尽可能地为用户呈现详实、准确的产品信息。但介于本手册的内容具有一定的时效性，致远微电子不能完全保证该文档在任何时段的时效性与适用性。致远微电子有权在没有通知的情况下对本手册上的内容进行更新，恕不另行通知。为了得到最新版本的信息，请尊敬的用户定时访问立功科技官方网站或者与致远微电子工作人员联系。感谢您的包容与支持！

专业 . 专注成就梦想

Dreams come true with professionalism and dedication.

广州致远微电子有限公司

更多详情请访问
ww.zlgmcu.com

欢迎拨打全国服务热线
400-888-2705