User Manual

# AMetal-AM116-Core 用户手册



| 类别 | 内容 |
| --- | --- |
| 关键词 | AM116-Core、功能介绍 |
| 摘要 | 本文档简述了 AM116-Core 硬件资源，详细介绍了 ametal-am116-core 软件包的结构、配置方法等 |

修订历史

| 版本 | 日期 | 原因 |
| --- | --- | --- |
| 2.0.10 | 2018/04/16 | 创建文档 |
| | | |

# 目 录

# 1. 开发平台简介

AM116-Core 开发平台主要用于 ZLG116 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 AM116-CORE 170814 Rev.A P/N: 1.14.24.0156 ，版本号可以在开发平台的背面找到。用户在使用 SDK 前请确认 SDK 支持自己手中的开发平台。

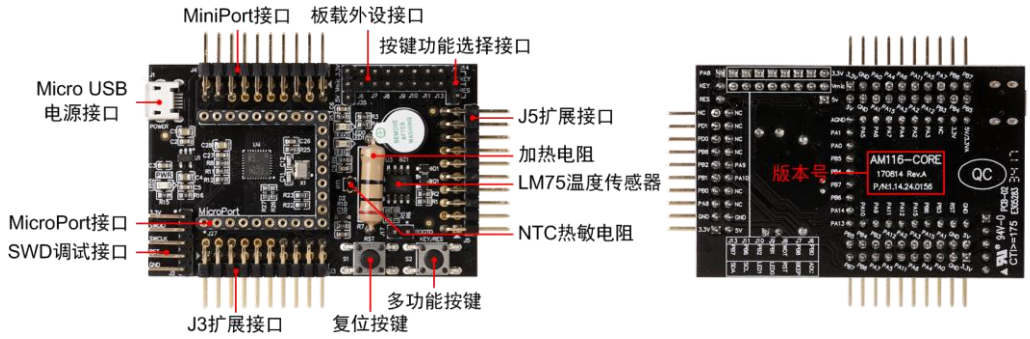


图 1.1 AM116-Core 开发平台

AM116-Core 开发板基于 ZLG 的 ZLG116 微控制器，其外形小巧、结构简单、片上资源设计合理。不到名片大小的电路板上包含了 1 路 MiniPort 接口、1 路 MicroPort 接口和 2 路 2×10 扩展接口。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 ZLG116

- 工作电压 2.0V ~ 5.5V；
- ARM Cortex-M0 32 位内核，主频可达 48MHz；
- 64KB Flash，8KB SRAM；
- 1 个 12 位 ADC、4 个 16 位通用定时器、1 个 32 位通用定时器、1 个高级 PWM 定时器；
- 2 个 UART 接口、1 个 $I^2C$ 接口和 2 个 SPI 接口；
- 2 个比较器、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 AM116-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 个热敏电阻；
- 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。

基于这些资源，可以完成多种基础实验。例如：加热电阻配合 LM75B 和热敏电阻可分别实现数字和模拟测温。

# 2. ametal_am116_core 软件包

软件包名为 ametal_am116_core_2.0.1（不同版本，版本号会有区别，请以实际获取的 AMetal 包版本为准）。为叙述方便，下文简称软件包为 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。

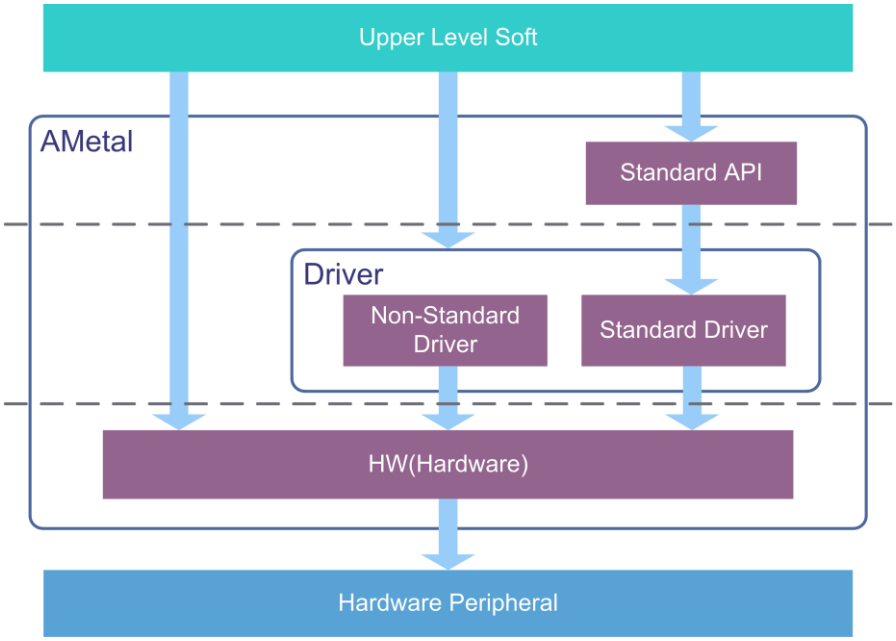


图 2.1 AMetal 框架

根据实际需求，这三层对应的接口均可被应用程序使用。对 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_zlg116、AMHW_ZLG116。

参见:

更多的硬件层接口定义及示例请参考{SDK}\documents\《AMetal-AM116-Core API 参考手册.chm》或者{SDK}\ametal\soc\zlg\zlg116\hw 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_zlg116 、AM_ZLG116 。

参见:

更多的驱动层接口定义及示例请参考{SDK}\documents\《AMetal-AM116-Core API 参考手册.chm》 或者 {SDK}\ametal\soc\zlg\zlg116\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-AM116-Core API 参考手册.chm》 或者 {SDK}\ametal\common\interface 文件夹中的相关文件。

## 2.2 目录结构

{SDK} 下一般有 5 个文件夹，如图 2.2 所示。

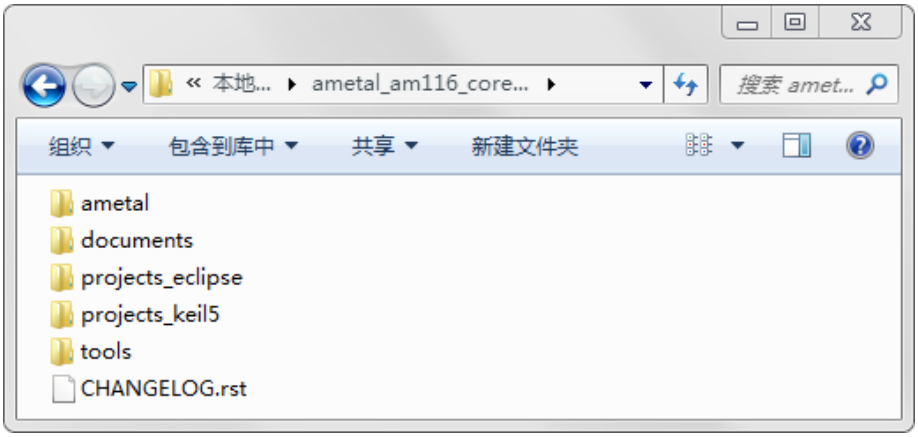


图 2.2 软件包目录结构

- ametal 目录存放软件包相关文件；
- documents 目录存放相关说明文档；
- projects_keil5 存放 Keil5 的工程模板和例程；
- projects_eclipse 存放 Eclipse 的工程模板和例程；
- tools 目录存放 SDK 相关的工具。

注意: projects_eclipse 与 projects_keil5 并不是所有的{SDK}都会提供，会根据用户的需要，选择提供。如果缺少了某一个，用户可以跳过其相关的内容介绍。

### 2.2.1 ametal 目录

ametal 目录结构如图 2.3 所示。

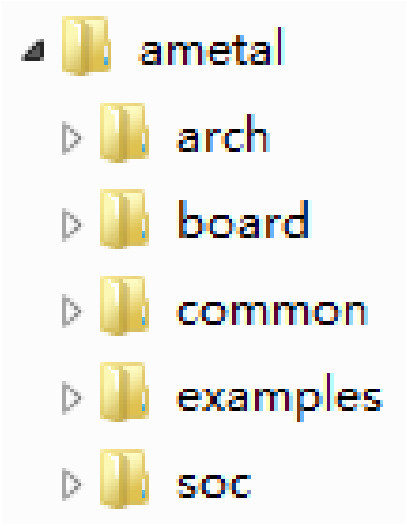


图 2.3 ametal 目录结构

1. arch

arch 文件夹存放了与内核相关的通用文件，如 NVIC、Systick 等，支持的内核有：Cortex-M0、Cortex-M0+、Cortex-M3、Cortex-M4。

2. board

board 文件夹包含了一些与芯片相关的启动文件及与开发板相关的设置和初始化函数，如板上 LED、蜂鸣器等，不同开发板，可能对应不同的 board 文件。

3. common

common 目录结构如图 2.4 所示。

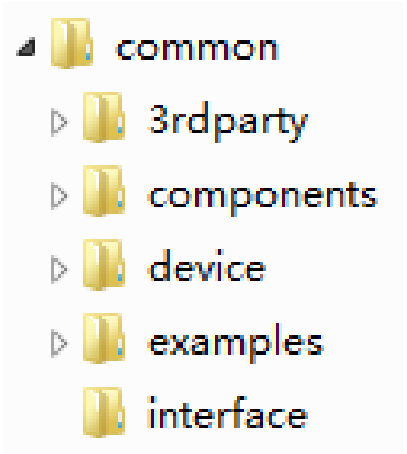


图 2.4 common 目录结构

common 文件夹下相关文件是 AMetal 提供的公共文件，包括一些通用的工具文件，外围设备的驱动文件，整合的第三方文件和标准接口文件。这些文件与具体芯片无关，下面介绍 common 目录下的各文件夹的作用。

(1) 3rdparty

3rdpart 用于存放一些完全由第三方提供的软件包，比如 CMSIS 软件包。CMSIS（Cortex Microcontroller Software Interface Standard）是 ARM Cortex 微控制器软件接口标准，是 Cortex-M 处理器系列的与供应商无关的硬件抽象层。

(2) 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。

(3) device

device 文件夹下存放的是某些外围设备的接口文件及相关的外围设备的驱动源文件，主要包括 NVRAM 模块(EP24Cxx)、BLE 模块(ZLG9021)、FLASH 模块(MX25xx)、RTC 模块(PCF85063) 等。

(4) examples

examples 文件夹下存放的是所有使用标准接口层实现的例程，仅供参考。

(5) interface

interface 文件夹下相关文件是 AMetal 提供的公共文件，包括一些通用的工具文件和标准接口文件。这些文件与具体芯片无关。

### 4. examples

examples 文件夹主要包含 am116_core 开发板各个外设及板载器件的例程源文件，目录视图如图 2.5 所示。

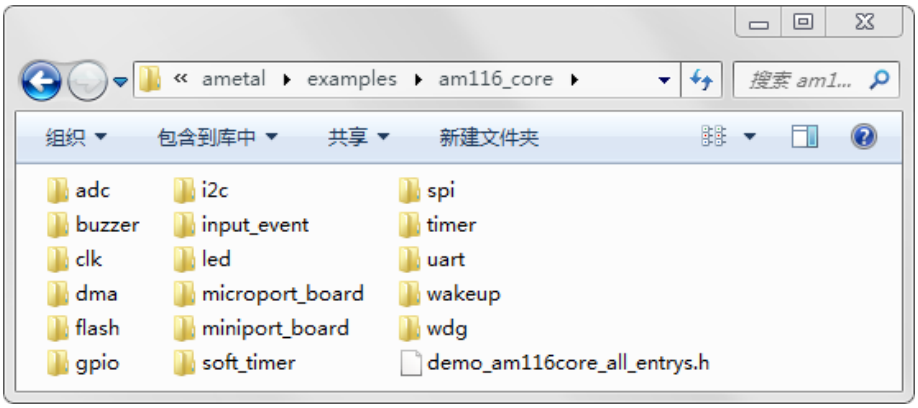


图 2.5 例程源文件目录视图

在 {SDK}\ametal\examples\am116_core 文件夹目录包含了各个外设的例程源文件，am116_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_am116_core_all_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。

以 uart 相关外设为例，打开 uart 目录，如 图 2.6 所示，可以看到该目录下提供了 6 个 demo 源程序。

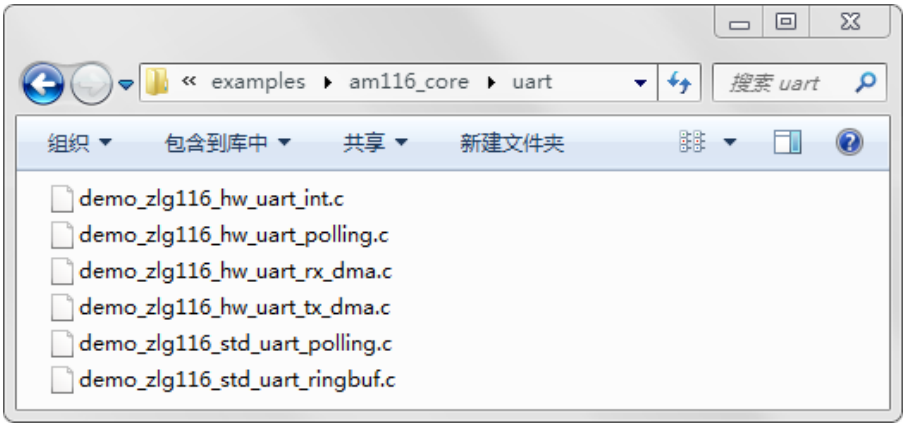


图 2.6 uart 外设所有 demo 源程序

所有芯片外设的 demo 源程序命名为：demo_zlg116_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_zlg116_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_zlg116_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_zlg116_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。

5. soc

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。如图 2.7 所示。

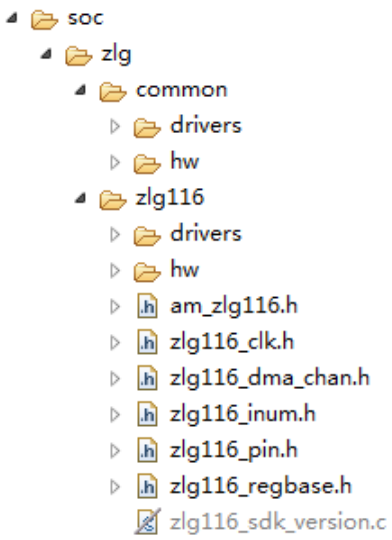


图 2.7 soc 目录结构

在图 2.7 中，drivers 文件夹存放了驱动层相关文件，hw 文件夹存放了硬件层相关文件。{SDK}\ametal\soc\zlg\zlg116 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 ZLG116 芯片各公共文件内容简介

| 文件名 | 内容简介 |
| --- | --- |
| am_zlg116.h | 包含了其它 ZLG116 公共部分头文件，只要使用 ZLG116 的代码，建议默认均包含此头文件 |
| zlg116_clk.h | 时钟 ID 号，如 CLK_ADC1、CLK_SPI1 等 |
| zlg116_dma_chan.h | DMA 通道号相关定义，如 DMA_CHAN_0，DMA_CHAN_1 等 |
| zlg116_inum.h | 中断号定义，如 INUM_DMA，INUM_I2C1 等 |
| zlg116_pin.h | IO 引脚号定义，包含 PIOA、PIOB、PIOC、PIOD 四个端口的引脚 |
| zlg116_regbase.h | ZLG116 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_zlg116.h 文件中，实际应用程序在使用时，只需要简单的包含 am_zlg116.h 即可。

### 2.2.2 documents 目录

documents 目录中存放了 4 个文件，分别为《AMetal-AM116-Core 快速入门手册(Keil)》、《AMetal-AM116-Core API 参考手册.chm》、《AMetal-AM116-Core 用户手册.pdf》、《AMetal ZLG116 引脚配置及查询.xlsm》。

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. 《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和引脚速率，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。

### 2.2.3 projects_keil5 目录

为了便于 Keil 工程的统一管理，Keil 版本工程模板和例程工程统一放置在了 projects_keil5 目录下。projects_keil5 表明该目录下的所有工程应该使用 Keil5 软件打开。打开 projects_keil5 目录，如图 2.8 所示。

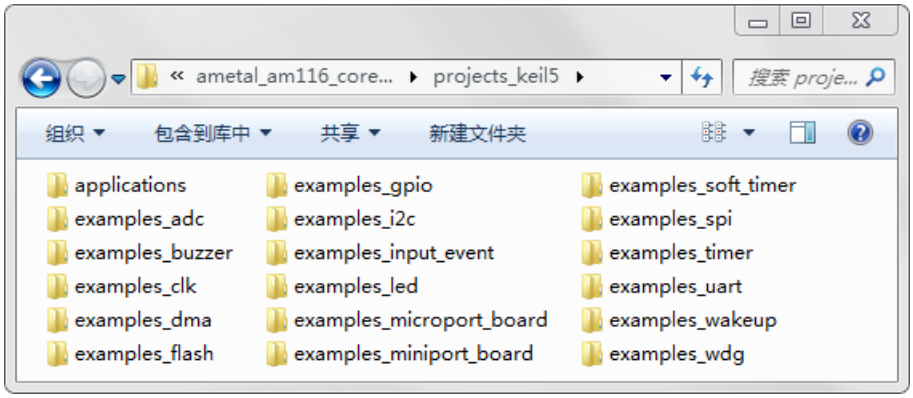


图 2.8 projects_keil5 目录视图

1. applications

该文件夹下存放所有的应用程序。用户开发应用程序时，工程文件夹均应存放在该目录下。初始时，该文件夹下仅包含工程模板，如图 2.9 所示。

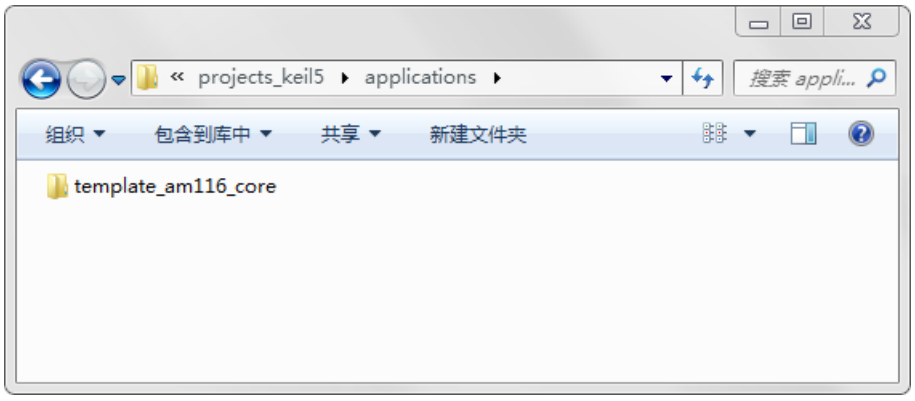


图 2.9 applications 目录视图

如需新建工程，只需要拷贝一份 template_am116_core 模板工程文件夹，并重命名为自己期望的文件名即可。如图 2.10 所示，通过拷贝的方式新建了一个命名为 led 的工程。

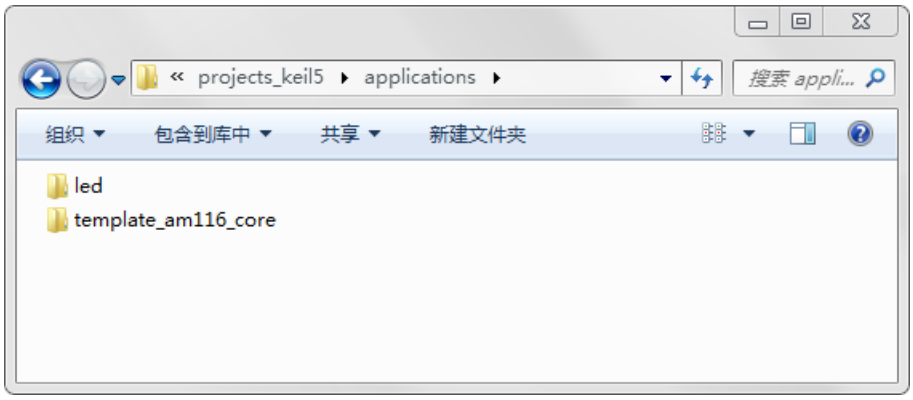


图 2.10 通过拷贝新建一个 LED 工程目录

注解：在拷贝工程时，不可任意拷贝至其它目录，只能与 template_am116_core 处于同一目录，即拷贝后的工程也应位于：{SDK}\projects_keil5\applications\ 目录下。否则工程文件相对路径发生变化，在编译时会产生找不到文件的错误。

打开 led 工程文件夹，如图 2.11 所示。template_am116_core.uvprojx 即为 Keil5 的工程文件，为了与工程目录的文件名一致，建议将 template_am116_core.uvprojx 重命名为 led.uvprojx。重命名后如图 2.12 所示。

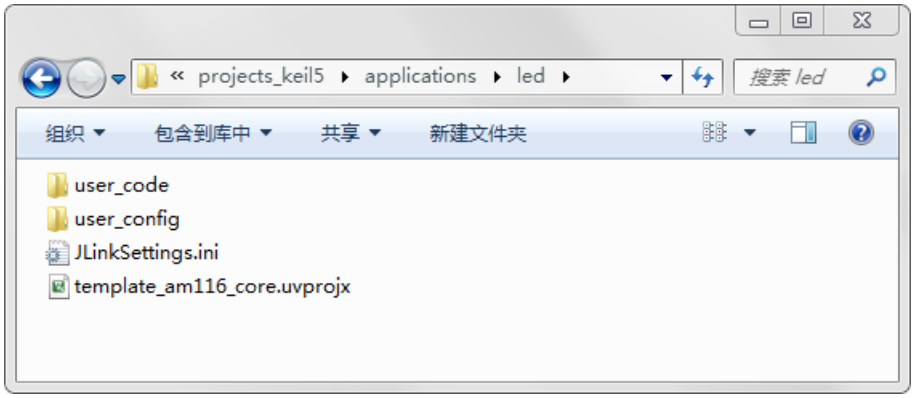


图 2.11 工程初始视图

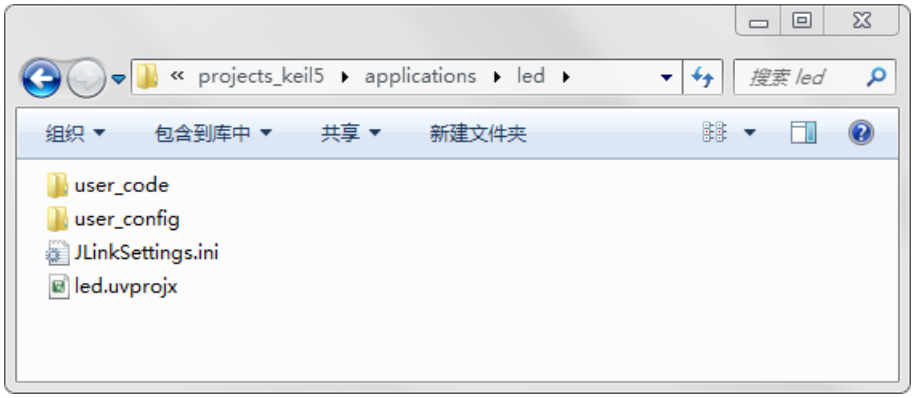


图 2.12 重命名 Keil 工程文件

可见，创建一个新的工程非常简单。所有工程（包括例程工程）均是基于工程模板创建的。在所有工程目录下，除工程文件 *.uvprojx 外，还有 user_code 和 user_config 两个文件夹及 JLinkSettings.ini 文件。

user_code 包含了用户主程序文件 main.c，用户程序的入口 am_main()函数即在该文件中。user_config 包含了工程配置文件以及各个外设的配置文件，提供了默认配置，用户可根据需要自行修改。

JLinkSettings.ini 是 Keil 工程采用 J-Link 调试时的一些配置信息。

2. 其它示例工程

除 application 目录外，其它工程均为示例工程。与例程源文件目录结构保持一致，如 UART 外设，打开 examples_uart，如图 2.13 所示，可以看到有 6 个工程文件夹，文件夹名与对应例程源文件的文件名保持一致。

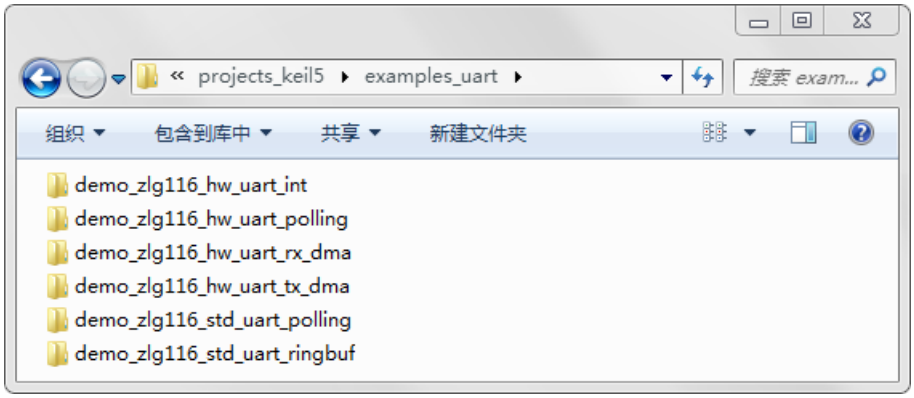


图 2.13 UART 外设所有例程的工程

这些工程均可以直接打开，编译后即可下载到开发板上运行，查看相对应的现象。以 demo_zlg116_std_uart_polling 为例，打开该文件夹，如图 2.14 所示，双击 Keil5 工程文件 demo_zlg116_std_uart_polling.uvprojx 文件即可打开工程。

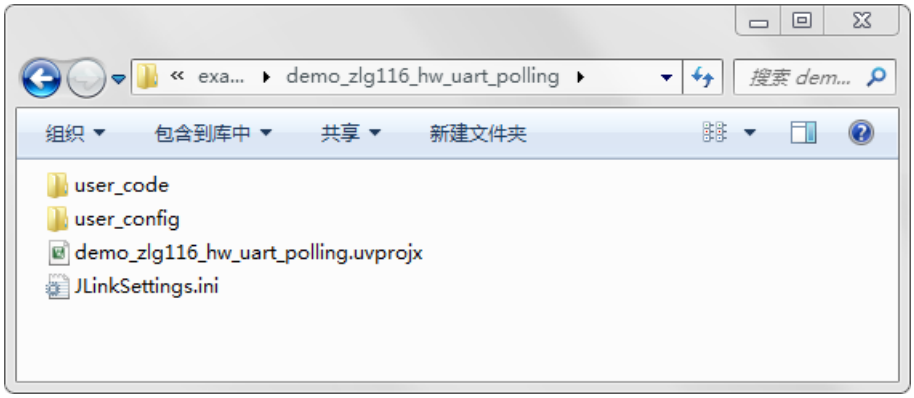


图 2.14 demo_zlg116_std_uart_polling 工程文件夹

注解:具体如何编译工程以及下载程序，请参考《AMetal-AM116-Core 快速入门手册(Keil).pdf》。

### 2.2.4 projects_eclipse 目录

为了便于 Eclipse 工程的统一管理， 我们将工程模板和例程工程统一放置在 projects_eclipse 目录下。projects_eclipse 表明该目录下的所有工程应该使用 Eclipse 软件打开。

打开 projects_eclipse 目录，如图 2.15 所示。

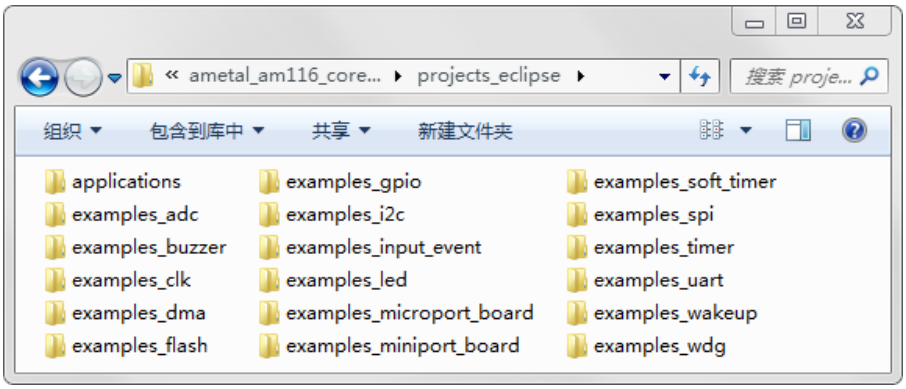


图 2.15 projects_eclipse 目录视图

1. applications

该文件夹下存放所有的应用程序。用户开发应用程序时，工程文件夹均应存放在该目录下。初始时，该文件夹下仅包含工程模板，如图 2.16 所示。

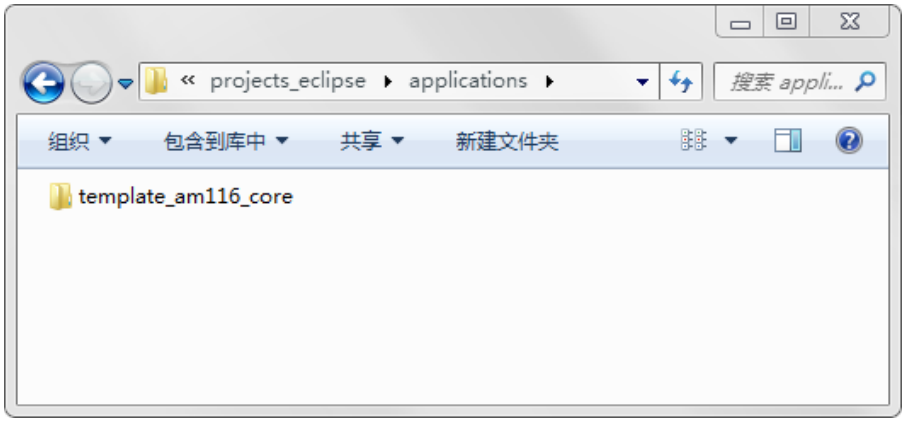


图 2.16 applications 目录视图

在模板工程目录下，除工程文件 .cproject 及 .project 外，还有 user_code 和 user_config 这个两个文件夹以及 zlg116_flash.ld 和 template_am116_core Debug.launch 和 tem- plate_am116_core Release.launch 这三个文件，如图 2.17 所示。

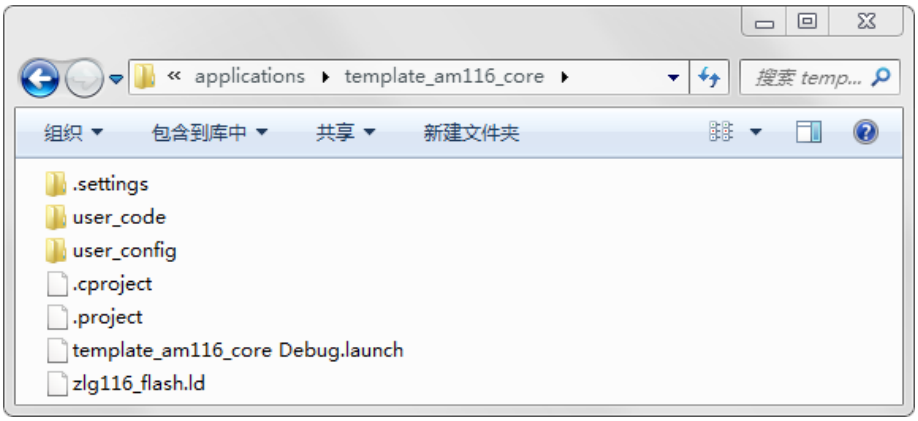


图 2.17 eclipse 模板工程目录结构

- user_code 包含了用户主程序文件 main.c ，用户程序的入口 am_main() 函数即在该文

件中。

- user_config 包含了工程配置文件以及各个外设的配置文件，提供了默认配置，用户可根据需要自行修改。
- zlg116_flash.ld 为工程使用到的链接脚本文件，用户不可更改里面的内容。
- template_am116_core Debug.launch 和 template_am116_core Release.launch 文件是用于生成调试配置信息的文件，有了这个文件，在调试配置时就不用手动填入调试配置信息，比如说设备名，调试所用的接口(JTAG 或者 SWD) 等，用户不用更改里面的内容。

2. 其它示例工程

除 application 目录外，其它工程均为示例工程。与例程源文件目录结构保持一致，如 UART 外设，打开 examples_uart，如图 2.18 所示，可以看到有 6 个工程文件夹，文件名与对应源文件的文件名保持一致。

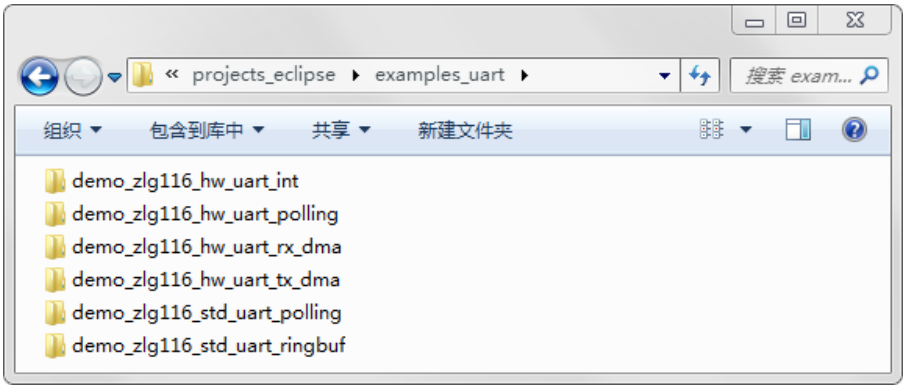


图 2.18 UART 外设所有例程的工程

这些工程均可以直接导入 Eclipse 工作空间中。把工程导入 Eclipse 工作空间后，进行编译后即可下载到开发板上运行，查看相对应的现象。

注解： 具体如何建立 Eclipse 工作空间并导入工程、新建工程、编译程序代码以及下载调试程序，请参考《AMetal-AM116-Core 快速入门手册 (Keil)》。

### 2.2.5 tools 目录

本目录下存放 SDK 相关的工具，例如 Keil 的 PACK 包。打开 tools 目录，如图 2.19 所示。

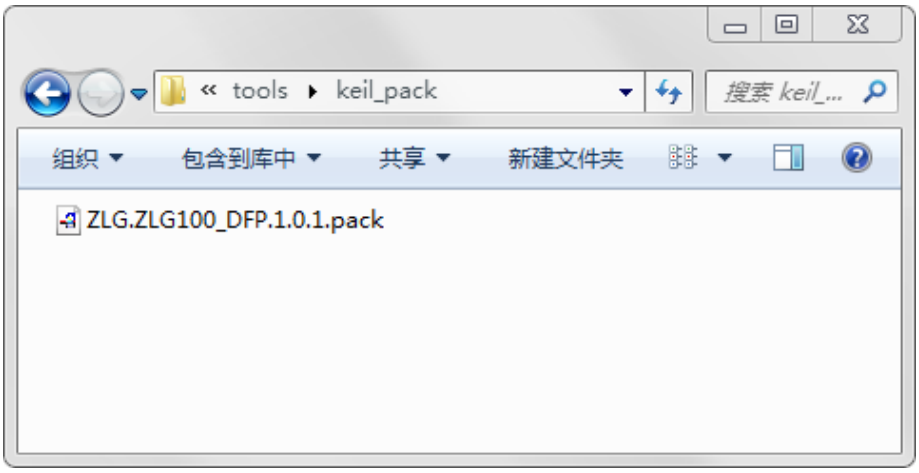


图 2.19 tools 目录视图

## 2.3 工程结构

### 2.3.1 Keil 工程结构

打开 Keil 版本的 template_am116_core 模板工程，其工程结构如图 2.20 所示。

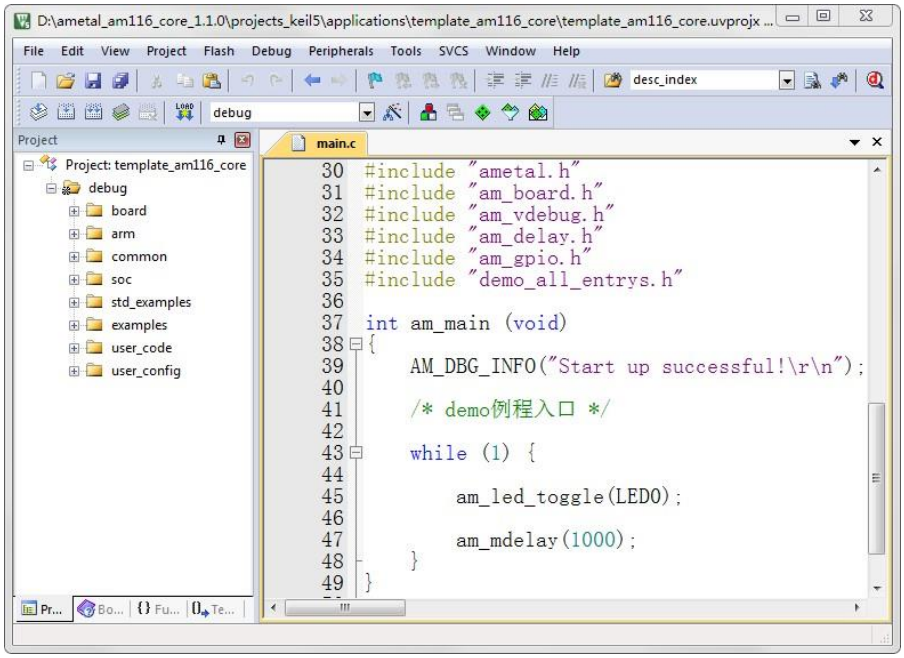


图 2.20 Keil 版本工程模板结构

在工程结构中，包含了 board、arm、common、soc、std_examples、examples、user_code、user_config 共 8 个节点。

- board 文件夹包含了一些与芯片相关的启动文件及与开发板相关的设置和初始化函数，如板上 LED、蜂鸣器等，不同开发板，可能对应不同的 board 文件，board 文件夹位于 {SDK}\ametal 下。
- arm 文件夹存放了与内核相关的通用文件，如 NVIC、Systick 等。
- common 文件夹包含一些通用的工具文件，外围设备的驱动文件，整合的第三方文件和

标准接口文件。

- 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。
- std_examples 文件夹下存放的是所有使用标准接口层实现的例程，仅供参考。
- examples 文件夹主要包含 am116_core 开发板各个外设及板载器件的例程源文件，ex-
- amples 文件夹位于 {SDK}\ametal 下。
- 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，对于 template_am116_core 工程，{PROJECT} 等价于{SDK}\projects_keil5\applications\template_am116_core ），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_am116_core 模板工程，其工程结构如图 2.21 所示。

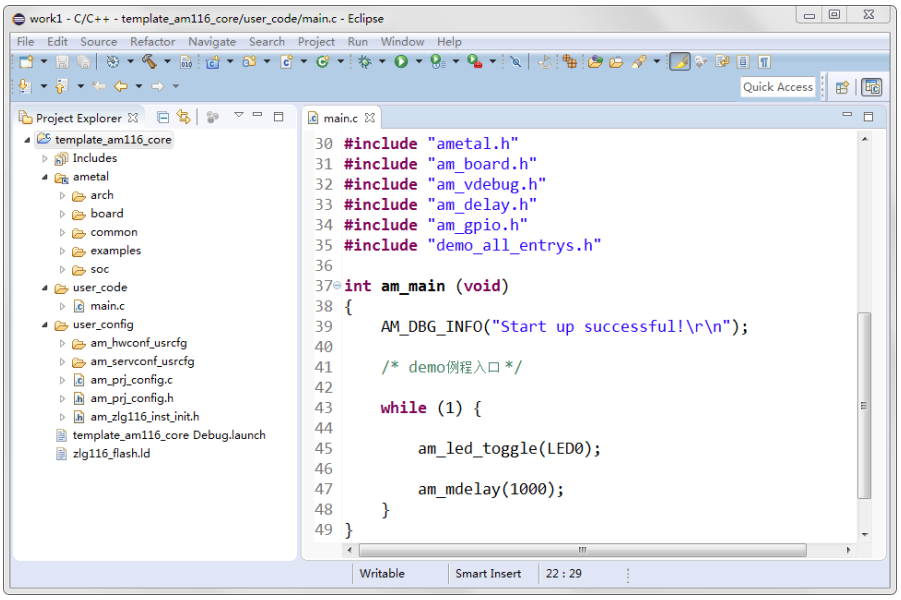


图 2.21 Eclipse 版本工程模板结构

在工程结构中，包含了 ametal、user_config、user_code 这几个节点。

- ametal 节点下包含了 arch、board、common、examples、lib 和 soc 几个文件夹，详细介绍见 2.2.1。
- user_code 下为用户程序，相关文件位于{PROJECT}\user_code 文件夹下(对于 template_am116_core 模板工程，{PROJECT} 等价于

  {SDK}\projects_eclipses\applications\template_am116_core )， 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 user_config 文件。

# 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_ENAB | 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_1 （需要短接跳线帽 J9）
*     ID: 1 --- PIOB_2 （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 TIM14 定时器，使能后才能正常使用定时器相关功能 |
| 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，使能后方可管理系统输入事件 |
| 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 |
| 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 |

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{PROJECT}\board\am_delay.，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. 片上外设资源

ZLG116 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_zlg116_* 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 | INT(中断) | am_hwconf_arm_nvic.c |
| 2 | ADC | am_hwconf_zlg116_adc.c |
| 3 | 时钟部分 (CLK) | am_hwconf_zlg116_clk.c |
| 4 | DMA | am_hwconf_zlg116_dma.c |
| 5 | GPIO | am_hwconf_zlg116_gpio.c |
| 6 | I2 C 控制器 | am_hwconf_zlg116_i2c.c |
| 7 | I2 C 从机控制器 | am_hwconf_zlg116_i2c_slv.c |
| 8 | 电源管理 | am_hwconf_zlg116_pwr.c |
| 9 | SPI(DMA 传输方式) | am_hwconf_zlg116_spi_dma.c |
| 10 | SPI(中断方式) | am_hwconf_zlg116_spi_int.c |
| 11 | 标准定时器的捕获功能 | am_hwconf_zlg116_tim_cap.c |
| 12 | 标准定时器的 PWM 功能 | am_hwconf_zlg116_tim_pwm.c |
| 13 | 标准定时器的定时功能 | am_hwconf_zlg116_tim_timing.c |
| 14 | UART | am_hwconf_zlg116_uart.c |
| 15 | 窗口看门狗 | am_hwconf_zlg116_wwdg.c |
| 16 | 独立看门狗 | am_hwconf_zlg116_iwdg.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_zlg116_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_zlg116_gpio_dev_t     g_gpio_dev;
```

这里使用 am_zlg116_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}\ametal\soc\zlg\zlg116\drivers\include\am_zlg116_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_zlg116_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

```
const am_zlg116_gpio_devinfo_t ZLG116_GPIO_BASE,       g_gpio_devinfo = {
        ZLG116_GPIO_BASE,       /**< \brief GPIO 控制器寄存器块基址 */
        ZLG116_SYSCFG_BASE,  /**< \brief SYSCFG 配置寄存器块基址 */
        ZLG116_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 */
        zlg116_plfm_gpio_init,
        zlg116_plfm_gpio_deinit
};
```

这里使用 am_zlg116_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}\ametal\soc\zlg\zlg116\drivers\include\am_zlg116_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_zlg116_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_zlg116_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_zlg116_gpio_devinfo_t;
```

可见，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 SYSCFG、GPIO、IEXTI 共计 3 个部分的外设，因此，在 GPIO 的设备信息中，需要 3 个基地址，对应 3 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}\ametal\soc\zlg\zlg116\zlg116_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 ZLG116_GPIO_BASE     (0x48000000UL)

/** \brief SYSCFG 基地址 */
#define ZLG116_SYSCFG_BASE(0x40010000UL)

/** \brief 外部中断 (事件) 控制器 EXTI 基地址 */
#define ZLG116_EXTI_BASE     (0x40010400UL)
```

可见，列表 4.2 中，设备信息前 3 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。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}\ametal\soc\zlg\zlg116\zlg116_inum.h 文件中定义好了，与 GPIO 相关的中断号定义详见程序清单 4.6。

程序清单 4.6 PINT 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. 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。

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，详见程序清单 4.7。

程序清单 4.7 GPIO 设备信息结构体类型——内存分配相关成员定义

```
/** \brief GPIO 支持的引脚中断号数量 */
size_tpint_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_zlg116_gpio_trigger_info *p_triginfo;
```

- 成员 pint_count 确定用户实际使用到的引脚中断数目。
- 成员 p_infomap 用于保存触发信息的映射关系，对应内存的大小应该与 pint_count 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 pint_count 一致。类型 am_zlg116_gpio_trigger_info 同样在 GPIO 驱动头文件{SDK}\ametal\soc\zlg\zlg116\drivers\include\am_zlg116_gpio.h 中定义，详见程序清单 4.8。

程序清单 4.8 GPIO 触发信息结构体类型定义

```
/**
* \brief 引脚的触发信息
*/
struct am_zlg116_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 pint_count ，另外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，提供了默认的这其中两个成员的值定义，详见程序清单 4.9。

程序清单 4.9 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_zlg116_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];
```

其中 PIN_INT_MAX 在 {SDK}\ametal\soc\zlg\116\zlg116_pin.h 中定义

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。

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 4.10。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 4.10 GPIO 设备信息结构体类型——平台初始化函数指针定义

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

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_zlg116_gpio.c 文件中定义，详见程序清单 4.11。

程序清单 4.11 GPIO 平台初始化函数

```
/** \brief GPIO 平台初始化 */
void    zlg116_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_zlg116_clk_reset(CLK_GPIOA);
    am_zlg116_clk_reset(CLK_GPIOB);
    am_zlg116_clk_reset(CLK_GPIOC);
    am_zlg116_clk_reset(CLK_GPIOD);
    am_zlg116_clk_reset(CLK_SYSCFG);
}
```

平台初始化函数中，使使能了 GPIO 相关外设的时钟以及将这些外设复位。

5. 平台解初始化函数

GPIO 的平台去初始化函数 zlg116_plfm_gpio_deinit，详见程序清单 4.12。

程序清单 4.12 GPIO 平台初始化函数

```
/** \brief GPIO 平台去初始化 */
void    zlg116_plfm_gpio_deinit (void)
{
```

```
    /* 复位 GPIO 相关外设 */
    am_zlg116_clk_reset(CLK_GPIOA);
    am_zlg116_clk_reset(CLK_GPIOB);
    am_zlg116_clk_reset(CLK_GPIOC);
    am_zlg116_clk_reset(CLK_GPIOD);
    am_zlg116_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);
}
```

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。

### 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}\ametal\soc\zlg\zlg116\drivers\include\am_zlg116_gpio.h 中声明。详见程序清单 4.13。

程序清单 4.13 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_zlg116_gpio_init (am_zlg116_gpio_dev_t           *p_dev,
                         const am_zlg116_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 4.14。

程序清单 4.14 完成 GPIO 初始化

```
am_zlg116_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_zlg116_gpio.c 文件中，详见程序清单 4.15。

程序清单 4.15 GPIO 实例初始化函数

```
/** \brief GPIO 实例初始化 */
int am_zlg116_gpio_inst_init (void)
{
    return am_zlg116_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}\ametal\soc\zlg\zlg116\drivers\include\am_zlg116_gpio.h 中声明。详见程序清单 4.16。

程序清单 4.16 GPIO 解初始化函数

```
/**
*     \brief GPIO 去初始化
*
*     \param[in] 无
*
*     \return 无
*/
void am_zlg116_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 4.17

程序清单 4.17 完成 GPIO 解初始化

```
am_zlg116_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，完全不用关心驱动初始化函数和解初始化函数，只需要调用设备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_zlg116_gpio.c 文件中，详见程序清单 4.18。

程序清单 4.18 GPIO 实例解初始化函数

```
/** \brief GPIO 实例解初始化 */
void am_zlg116_gpio_inst_deinit (void)
{
    am_zlg116_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

ZLG116 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_zlg_adc_v0_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）。详见程序清单 4.19。

程序清单 4.19 ADC 参考电压默认配置

```
/** \brief ADC 设备信息 */
static const am_zlg_adc_v0_devinfo_t     g_adc_devinfo = {
    ZLG116_ADC_BASE,                                /**< \brief ADC */
    INUM_ADC_COMP, CLK_ADC1,                        /**< \brief ADC 的中断编号 */
    CLK_ADC1,                                       /**< \brief ADC 时钟号 */
    3300,                                           /**< \brief 参考电压 */
    AMHW_ZLG_ADC_V0_DATA_VALID_12BIT,   /**< \brief 转换精度 */
    __zlg_plfm_adc_init,                           /**< \brief ADC 的平台初始化 */
    __zlg_plfm_adc_deinit,                         /**< \brief ADC 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。

### 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 4.20。

程序清单 4.20 时钟默认设备信息

```
/**
 * \brief CLK 设备信息
 */
 static const am_zlg116_clk_devinfo_t __g_clk_devinfo =
 {
     /**
      * \brief HSEOSC 外部晶振频率
      *
      * 如果 pllin_src 选择 AMHW_ZLG116_PLLCLK_HSE 则 PLLIN = hse_osc）
      * PLLIN = 12000000
      */
      12000000,
```

```
/** \brief
* PLL 时钟源选择
* -# AMHW_ZLG116_PLLCLK_HSI_DIV4 : HSI 振荡器 4 分频作为 PLL 输入时钟
* -# AMHW_ZLG116_PLLCLK_HSE : HSE 作为 PLL 输入时钟
*/
AMHW_ZLG116_PLLCLK_HSE,

/**
* \brief PLL 倍频系数，可选 1-64
* PLLOUT = PLLIN * pll_mul / pll_div
* PLLOUT = 12000000 * 4 / 1 = 48Mhz
*/
4,

/**
* \brief PLL 分频系数，可选 1-8
* PLLOUT = PLLIN * pll_mul / pll_div
* PLLOUT = 12000000 * 4 / 1 = 48Mhz
*/
1,


/** \brief USB 分频系数， USBCLK = PLLOUT / (usb_div + 1)，建议配置成 48Mhz */
0,

/**
* \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 */
    __zlg116_clk_plfm_init,

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

可见，由于默认配置中，主时钟源头选择 PLL。时钟倍频系数为 4，分频系数为 1，因此系统时钟频率为 48MHz。用户可根据具体的需求配置相应的时钟频率。

### 4.2.3 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。

### 4.2.4 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。

### 4.2.5 I²C

平台只有一个 I²C 接口，定义为 I²C1。分别实现了主机控制器和从机控制器。讲述其配置内容。一般地，对于主机控制器只需要配置 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_zlg116_i2c.c 中的 I2C1 设备信息中第 4 个参数配置，详见程序清单 4.21。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 4.21 I²C1 速率配置

```
/**
* \brief I2C1 设备信息
*/
static const am_zlg_i2c_v0_devinfo_t      g_i2c1_devinfo = {
    ZLG116_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 */
    0,                          /**< \brief 超时值 0 */

    __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{HWCONFIG}\am_hwconf_zlg116_i2c.c 设备信息中第 5 个参数设置，详见程序清单 4.21 默认值为 0，即不使用超时机制。若需要使用，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，详见程序清单 4.22。

程序清单 4.22 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)
{

    /**
    * PIOB_6 ~ I2C1_SCL, PIOB_7 ~ I2C1_SDA
    */
    am_gpio_pin_cfg(PIOB_6, PIOB_6_I2C_SCL | PIOB_6_AF_OD | PIOB_6_SPEED_2MHz);
    am_gpio_pin_cfg(PIOB_7, PIOB_7_I2C_SDA | PIOB_7_AF_OD | PIOB_7_SPEED_2MHz);

    am_clk_enable(CLK_I2C1); am_zlg116_clk_reset(CLK_I2C1);
}
```

可见，程序中，将 PIOB_6 配置为 I²C1 的时钟线，PIOB_7 配置为 I²C1 的数据线。可以直接使用 am_gpio_pin_cfg() 函数将一个具有 I²C 功能的引脚配置为相应的 I²C 功能。用户可参考 {SDK}\ametal\soc\zlg\zlg116 目录下的 zlg116_pin.h 中相关的引脚功能。如果想使用其他引脚配置为 I²C 功能，则需要在这里修改。

对于从机控制器则只需要配置引脚信息配置信息保存在{HWCONFIG}\am_hwconf_zlg116_i2c_slv.c 中，以 I²C1 为例，详见程序清单 4.23。

程序清单 4.23 从机控制器配置

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

```
 {

     /**
      * PIOB_6 ~ I2C1_SCL, PIOB_7 ~ I2C1_SDA
      */
      am_gpio_pin_cfg(PIOB_6, PIOB_6_I2C_SCL | PIOB_6_AF_OD | PIOB_6_SPEED_2MHz);
      am_gpio_pin_cfg(PIOB_7, PIOB_7_I2C_SDA | PIOB_7_AF_OD | PIOB_7_SPEED_2MHz);

      am_clk_enable(CLK_I2C1);
      am_zlg116_clk_reset(CLK_I2C1);
      }

      ......

/**
* \brief I2C1 从 设备信息
*/
static const am_zlg_i2c_v0_slv_devinfo_t __g_i2c1_devinfo = {

      ZLG116_I2C1_BASE,                  /**< \brief I2C1 寄存器块基址 */
      INUM_I2C1,                         /**< \brief I2C1 中断编号 */

      __zlg_i2c1_plfm_init,              /**< \brief 平台初始化 */
      __zlg_i2c1_plfm_deinit             /**< \brief 平台去初始化 */
 };
```

### 4.2.6 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 4.24。

程序清单 4.24 引脚触发信息内存

```
/** \brief 引脚触发信息内存 */
static struct am_zlg116_pwr_mode_init     g_pwr_mode_init[3] = {
    {AM_ZLG116_PWR_MODE_SLEEP,   PIOA_8},
    {AM_ZLG116_PWR_MODE_STOP,    PIOA_8},
    {AM_ZLG116_PWR_MODE_STANBY,  PIOA_0},
};
```

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

注意：待机模式只能使用 WKUP 引脚的上升沿、NRST 引脚上外部复位、IWDG 复位这三个条件唤醒，所以触发引脚只能配置为 PIOA_0，不能修改。

### 4.2.7 SPI

平台有 2 个 SPI 总线接口，定义为 SPI1、SPI2。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

如果使用 SPI 的中断方式，需要在 {HWCONFIG}\am_hwconf_zlg116_spi_int.c 文件中进行 SPI 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}\ametal\common\include\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 4.25。

程序清单 4.25 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(PIOA_5, PIOA_5_SPI1_SCK | PIOA_5_AF_PP);
    am_gpio_pin_cfg(PIOA_6, PIOA_6_SPI1_MISO | PIOA_6_INPUT_FLOAT);
    am_gpio_pin_cfg(PIOA_7, PIOA_7_SPI1_MOSI | PIOA_7_AF_PP);

    am_clk_enable(CLK_SPI1);
}
```

可见这里 PIOA_5 设置为 SCK，PIOA_6 设置为 MISO，PIOA_7 设置为 MOSI。如果你需要设置别的引脚的话，需要在这里更改。用户可参考 {SDK}\ametal\soc\zlg\zlg116 目录下的 zlg116_pin.h 中相关的引脚功能。

2. DMA 传输方式

如果使用 SPI 的 DMA 方式，需要在 {HWCONFIG}\am_hwconf_zlg116_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。

### 4.2.8 Timer

zlg116 有 4 个 16 位通用定时器、1 个 32 位通用定时器、1 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 Timer1 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_zlg116_tim_cap.c，Timer1，Timer2，Timer3 支持 4 路捕获通道，Timer14，Timer16，Timer17 支持 1 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 4.26。

程序清单 4.26 引脚触发信息内存

```
/** \brief TIM1 用于捕获功能的设备信息 */
```

```
const am_zlg_tim_v0_cap_devinfo_t    g_tim1_cap_devinfo = {
    ZLG116_TIM1_BASE,                /**< \brief TIM1 寄存器块的基地址 */
    INUM_TIM1_CC,                    /**< \brief TIM1 中断编号 */
    CLK_TIM1,                        /**< \brief TIM1 时钟 ID */
    4,                               /**< \brief 4 个捕获通道 */
    & g_tim1_cap_ioinfo_list[0],     /**< \brief 引脚配置信息列表 */
    AMHW_ZLG_TIM_V0_TYPE0,           /**< \brief 定时器类型 */
    __zlg_plfm_tim1_cap_init,        /**< \brief 平台初始化函数 */
    __zlg_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，详见程序清单 4.27。

程序清单 4.27 Timer1_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_zlg_tim_v0_cap_ioinfo_t    g_tim1_cap_ioinfo_list[] = {

    /** \brief 通道 1 */
    {PIOA_8,  PIOA_8_TIM1_CH1  | PIOA_8_INPUT_FLOAT,  PIOA_8_GPIO  |
PIOA_8_INPUT_FLOAT},

    /** \brief 通道 2 */
    {PIOA_9,  PIOA_9_TIM1_CH2  | PIOA_9_INPUT_FLOAT,  PIOA_9_GPIO  |
PIOA_9_INPUT_FLOAT},

    /** \brief 通道 3 */
    {PIOA_10, PIOA_10_TIM1_CH3 | PIOA_10_INPUT_FLOAT, PIOA_10_GPIO |
PIOA_10_INPUT_FLOAT},

    /** \brief 通道 4 */
    {PIOA_11, PIOA_11_TIM1_CH4 | PIOA_11_INPUT_FLOAT, PIOA_11_GPIO |
PIOA_11_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_v0_cap_ioinfo_t，该类型在对应驱动头文件{SDK}\ametal\soc\zlg\common\drivers\include\am_zlg_timer_cap.h 中定义详见程序清单 4.28。

程序清单 4.28 Timer_CAP 通道引脚信息结构体类型

```
/**
* \brief TIM（版本 0）捕获功能相关的 GPIO 信息
*/
typedef struct am_zlg_tim_v0_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_v0_cap_ioinfo_t;
```

用户需要配置别的引脚作为捕获功能引脚的时候，用户可参考{SDK}\ametal\soc\zlg\zlg116 目录下的 zlg116_pin.h 中相关的引脚功能。

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_zlg116_tim_pwm.c，TIM1 用于 PWM 的设备信息详见程序清单 4.29。

程序清单 4.29 Timer_PWM 通道引脚信息结构体类型

```
/** \brief TIM1 用于 PWM 设备信息 */
const am_zlg_tim_v0_pwm_devinfo_t g_tim1_pwm_devinfo = {
    ZLG116_TIM1_BASE   ,                    /**< \brief TIM1 寄存器块的基地址 */
    CLK_TIM1,                               /**< \brief TIM1 时钟 ID */
    4,                                      /**< \brief 4 个 PWM 输出通道 */
    AMHW_ZLG_TIM_V0_PWM_MODE2,              /**< \brief PWM 输出模式 2 */
    0,                                      /**< \brief PWM 输出高电平有效 */
    & g_tim1_pwm_ioinfo_list[0],            /**< \brief 引脚配置信息列表 */
    AMHW_ZLG_TIM_V0_TYPE0,                  /**< \brief 定时器类型 */
    __zlg_plfm_tim1_pwm_init,               /**< \brief 平台初始化函数 */
    __zlg_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_zlg116_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。

### 4.2.9 UART

平台台有 2 个 UART 接口，定义为 UART1、UART2。在 SDK 提供的驱动中，只实现了 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}\ametal\common\interface\am_uart.h

1. 引脚配置

UART1 相关的引脚在配置文件{HWCONFIG}\am_hwconf_zlg116_uart.c 文件中配置，详见程序清单 4.30。

程序清单 4.30 UATRT 平台初始化函数

```
/** \brief 串口 1 平台初始化 */
static void     zlg_plfm_uart1_init (void)
```

```
{

    /* 引脚初始化 PIOA_9_UART1_TX   PIOA_10_UART1_RX */
    am_gpio_pin_cfg(PIOA_9, PIOA_9_UART1_TX | PIOA_9_AF_PP);
    am_gpio_pin_cfg(PIOA_10, PIOA_10_UART1_RX | PIOA_10_INPUT_FLOAT);

    /* 开启 UART1 时钟 */
    am_clk_enable(CLK_UART1);
}
```

程序中将 PIOA_9 作为 uart1 的发送引脚，PIOA_10 作为 uart1 的接收引脚。用户也可以选择其他具有 uart 功能的引脚作为串口的发送、接收引脚，只是要注意引脚复用的问题。用户可参考 {SDK}\ametal\soc\zlg\zlg116 目录下的 zlg116_pin.h 中相关的引脚功能。

2. 配置波特率

在串口 1 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 4.31。

程序清单 4.31 UATRT 平台初始化函数

```
/** \brief 串口 1 设备信息 */
static const am_zlg_uart_v0_devinfo_t __g_uart1_devinfo = {
    ZLG116_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_V0_DATA_8BIT |                /**< \brief 8 位数据 */
    AMHW_ZLG_UART_V0_PARITY_NO |                /**< \brief 无极性 */
    AMHW_ZLG_UART_V0_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 */
};
```

### 4.2.10 WWDT

看门狗 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。

## 4.3 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。

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_zlg116_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_zlg116_inst_init.h 头文件。各个外设对应的设备实例初始化函数的原型详见表 4-2。

表 4-2 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 | ADC | am_zlg116_adc_inst_init (void) |
| 2 | CLK | int am_zlg116_clk_inst_init(void) |
| 3 | DMA | int am_zlg116_dma_inst_init (void) |
| 4 | GPIO | int am_zlg116_gpio_inst_init (void) |
| 5 | I2C1 | am_zlg116_i2c1_inst_init (void) |
| 6 | SPI1_INT | am_zlg116_spi1_int_inst_init (void) |
| 7 | SPI1_DMA | am_zlg116_spi1_dma_inst_init (void) |
| 8 | Timer1_CAP | am_zlg116_tim1_cap_inst_init (void) |
| 9 | Timer2_CAP | am_zlg116_tim2_cap_inst_init (void) |
| 10 | Timer3_CAP | am_zlg116_tim3_cap_inst_init (void) |
| 11 | Timer14_CAP | am_zlg116_tim14_cap_inst_init (void) |
| 12 | Timer16_CAP | am_zlg116_tim16_cap_inst_init (void) |
| 13 | Timer17_CAP | am_zlg116_tim17_cap_inst_init (void) |
| 14 | Timer1_PWM | am_zlg116_tim1_pwm_inst_init (void) |
| 15 | Timer2_PWM | am_zlg116_tim2_pwm_inst_init (void) |
| 16 | Timer3_PWM | am_zlg116_tim3_pwm_inst_init (void) |
| 17 | Timer14_PWM | am_zlg116_tim14_pwm_inst_init (void) |
| 18 | Timer16_PWM | am_zlg116_tim16_pwm_inst_init (void) |
| 19 | Timer17_PWM | am_zlg116_tim17_pwm_inst_init (void) |
| 20 | Timer1_timing | am_zlg116_tim1_timing_inst_init (void) |
| 21 | Timer2_timing | am_zlg116_tim2_timing_inst_init(void) |
| 22 | Timer3_timing | am_zlg116_tim3_timing_inst_init(void) |

续上表

| 序号 | 外设 | 实例初始化函数原型 |
|---|---|---|
| 23 | Timer14_timing | am_zlg116_tim14_timing_inst_init(void) |
| 24 | Timer16_timing | am_zlg116_tim16_timing_inst_init(void) |
| 25 | Timer17_timing | am_zlg116_tim17_timing_inst_init(void) |
| 26 | INT | am_zlg116_nvic_inst_init (void) |
| 27 | PWR | am_zlg116_pwr_inst_init (void) |
| 28 | UART1 | am_zlg116_uart1_inst_init (void) |
| 29 | UART2 | am_zlg116_uart2_inst_init (void) |
| 30 | WWDT | am_zlg116_wwdg_inst_init (void) |
| 31 | IWDT | am_zlg116_iwdg_inst_init (void) |

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两类：

- 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；

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 4-3。

表 4-3 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_zlg116_clk_inst_init(void) |
| 2 | DMA | int am_zlg116_dma_inst_init(void) |
| 3 | GPIO | int am_zlg116_gpio_inst_init(void) |
| 4 | INT | int am_zlg116_nvic_inst_init() |

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}\ametal\common\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 4.32。

程序清单 4.32 GPIO 标准接口使用范例

```
am_gpio_pin_cfg(PIOA_10, AM_GPIO_OUTPUT); /* 将 GPIO 配置为输出模式 */
am_gpio_set(PIOA_10, 1);  /* 设置 PIOA_10 输出高电平 */
```

参见:

接口原型及详细的使用方法请参考 {SDK}\documents\《ametal_am116_core API 参考手

册.chm》或者 {SDK}\ametal\common\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}\ametal\soc\zlg\zlg116\drivers\include\am_zlg116_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。

(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 4-4。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 4-4 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_zlg116_adc_inst_init (void) |
| 2 | I2C1 | am_zlg116_i2c1_inst_init (void) |
| 3 | SPI1_INT | am_zlg116_spi1_int_inst_init (void) |
| 4 | SPI1_DMA | am_zlg116_spi1_dma_inst_init (void) |
| 5 | Timer1_CAP | am_zlg116_tim1_cap_inst_init (void) |
| 6 | Timer2_CAP | am_zlg116_tim2_cap_inst_init (void) |
| 7 | Timer3_CAP | am_zlg116_tim3_cap_inst_init (void) |
| 8 | Timer14_CAP | m_zlg116_tim14_cap_inst_init (void) |
| 9 | Timer16_CAP | am_zlg116_tim16_cap_inst_init (void) |
| 10 | Timer17_CAP | am_zlg116_tim17_cap_inst_init (void) |
| 11 | Timer1_PWM | am_zlg116_tim1_pwm_inst_init (void) |
| 12 | Timer2_PWM | am_zlg116_tim2_pwm_inst_init (void) |
| 13 | Timer3_PWM | am_zlg116_tim3_pwm_inst_init (void) |
| 14 | Timer14_PWM | am_zlg116_tim14_pwm_inst_init |
| 15 | Timer16_PWM | am_zlg116_tim16_pwm_inst_init |
| 16 | Timer17_PWM | am_zlg116_tim17_pwm_inst_init |
| 17 | Timer1_timing | am_zlg116_tim1_timing_inst_init |
| 18 | Timer2_timing | am_zlg116_tim2_timing_inst_init(void) |
| 19 | Timer3_timing | am_zlg116_tim3_timing_inst_init(void) |
| 20 | Timer14_timing | am_zlg116_tim14_timing_inst_init(voi |
| 21 | Timer16_timing | am_zlg116_tim16_timing_inst_init(voi |
| 22 | Timer17_timing | am_zlg116_tim17_timing_inst_init(voi |
| 23 | PWR | am_zlg116_pwr_inst_init (void) |
| 24 | UART1 | am_zlg116_uart1_inst_init (void) |
| 25 | UART2 | am_zlg116_uart2_inst_init (void) |
| 26 | WWDT | am_zlg116_wwdg_inst_init (void) |
| 27 | IWDT | am_zlg116_iwdg_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 4.33。

程序清单 4.33 ADC 简单操作示例

```
#include “ametal.h”
#include “am_adc.h”
#include "am_zlg116_inst_init.h"

uint32_t g_adc_val_buf[200]; am_adc_handle_t g_adc_handle;
int am_main (void) {

    g_adc_handle = am_zlg116_adc_inst_init();

    /* 读取 ADC 转换的电压值 */
    am_adc_read_mv(g_adc_handle, 0, g_adc_val_buf, 200);

    /* …… */
    while(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_zlg116_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_zlg116_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 4-5。

表 4-5 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 | ADC | void am_zlg116_adc_inst_deinit (void) |
| 2 | DMA | void am_zlg116_dma_inst_deinit (void) |
| 3 | GPIO | void am_zlg116_gpio_inst_deinit (void) |
| 4 | I2C1 | void am_zlg116_i2c1_inst_deinit (void) |
| 5 | SPI1_INT | void am_zlg116_spi1_int_inst_deinit (void) |
| 6 | SPI1_DMA | void am_zlg116_spi1_dma_inst_deinit (void) |
| 7 | Timer1_CAP | void am_zlg116_tim1_cap_inst_deinit (void) |
| 8 | Timer2_CAP | void am_zlg116_tim2_cap_inst_deinit (void) |
| 9 | Timer3_CAP | void am_zlg116_tim3_cap_inst_deinit (void) |
| 10 | Timer14_CAP | void m_zlg116_tim14_cap_inst_deinit (void) |
| 11 | Timer16_CAP | void am_zlg116_tim16_cap_inst_deinit (void) |

续上表

| 序号 | 外设 | 实例初始化函数原型 |
|---|---|---|
| 12 | Timer17_CAP | void am_zlg116_tim17_cap_inst_deinit (void) |
| 13 | Timer1_PWM | void am_zlg116_tim1_pwm_inst_deinit (void) |
| 14 | Timer2_PWM | void am_zlg116_tim2_pwm_inst_deinit (void) |
| 15 | Timer3_PWM | void am_zlg116_tim3_pwm_inst_deinit (void) |
| 16 | Timer14_PWM | void am_zlg116_tim14_pwm_inst_deinit (void) |
| 17 | Timer16_PWM | void am_zlg116_tim16_pwm_inst_deinit (void) |
| 18 | Timer17_PWM | void am_zlg116_tim17_pwm_inst_deinit (void) |
| 19 | Timer1_timing | void am_zlg116_tim1_timing_inst_deinit (void) |
| 20 | Timer2_timing | void am_zlg116_tim2_timing_inst_deinit(void) |
| 21 | Timer3_timing | void am_zlg116_tim3_timing_inst_deinit(void) |
| 22 | Timer14_timing | void am_zlg116_tim14_timing_inst_deinit(void) |
| 23 | Timer16_timing | void am_zlg116_tim16_timing_inst_deinit(void) |
| 24 | Timer17_timing | void am_zlg116_tim17_timing_inst_deinit(void) |
| 25 | INT | void am_zlg116_nvic_inst_deinit (void) |
| 26 | PWR | void am_zlg116_pwr_inst_deinit (void) |
| 27 | UART1 | void am_zlg116_uart1_inst_deinit (void) |
| 28 | UART2 | void am_zlg116_uart2_inst_deinit (void) |
| 29 | WWDT | void am_zlg116_wwdg_inst_deinit (void) |
| 30 | IWDT | void am_zlg116_iwdg_inst_init (void) |

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（特殊地，系统控制部分功能混杂，默认所有函数直接操作 SYSCON 各个功能，无需再传入相应的外设寄存器结构体指针）。

以 ADC 为例，所有硬件层函数均在 {SDK}\ametal\soc\zlg\common\hw\include\amhw_zlg_adc_v0.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 4.34。

程序清单 4.34 ADC 硬件层操作函数

```
/**
*      \brief ADC(Version 0) function disable set
```

```
*
*       \param[in] p_hw_adc   : The pointer to the structure ADC(Version 0) register
*       \param[in] flag    : see the filed of function enable mask
*
*       \return none
*/
am_static_inline
void amhw_zlg_adc_v0_ctrl_reg_clr (amhw_zlg_adc_v0_t *p_hw_adc,
                                   uint32_t       flag)
{
    p_hw_adc->adcr &= ~flag;
}

/**
*       \brief ADC(Version 0) input channel enable
*
*       \param[in] p_hw_adc   : The pointer to the structure ADC(Version 0) register
*       \param[in] chan   : see the filed of ADC(Version 0) channel
*
*       \return none
*/
am_static_inline
void amhw_zlg_adc_v0_input_chan_enable (amhw_zlg_adc_v0_t *p_hw_adc,
                                        uint32_t    chan)
{
    if (chan == 10 || chan == 11) { chan += 4;
    }
    p_hw_adc->adchs |= (1 << chan);
}

/**
*       \brief ADC(Version 0) input channel disable
*
*       \param[in] p_hw_adc   : The pointer to the structure ADC(Version 0) register
*       \param[in] flag    : see the filed of ADC(Version 0) input channel
*
*       \return none
*/
am_static_inline
void amhw_zlg_adc_v0_input_chan_disable (amhw_zlg_adc_v0_t *p_hw_adc,
                                         uint32_t    flag)
{
    p_hw_adc->adchs &= ~flag;
```

```
}
```

其它函数读者可自行打开 {SDK}\ametal\soc\zlg\common\hw\include\amhw_zlg_adc_v0.h 文件查看。这些函数均是以 amhw_zlg_adc_v0_t * 类型作为第一个参数。amhw_zlg_adc_v0_t 类型在 {SDK}\ametal\soc\zlg\common\hw\include\amhw_zlg_adc_v0.h 文件中定义，用于定义出 ADC 外设的各个寄存器程序清单 4.35。

程序清单 4.35 ADC 寄存器结构体定义

```
/**
* \brief ADC(Version 0) - Register Layout Typedef
*/
typedef struct amhw_zlg_adc_v0 {
    IO uint32_t addata;     /**< \brief data register */
    IO uint32_t adcfg;      /**< \brief configure register */
    IO uint32_t adcr;  /**< \brief control register */
    IO uint32_t adchs;      /**< \brief channel select register */
    IO uint32_t adcmpr;     /**< \brief window comparison register */
    IO uint32_t adsta;/**< \brief status register */
    IO uint32_t addr[9];    /**< \brief data address register */
} amhw_zlg_adc_v0_t;
```

该类型的指针已经{SDK}\ametal\soc\zlg\zlg116\hw\include\amhw_zlg116_periph_map.h 文件中定义，应用程序可以直接使用。详见程序清单 4.36。

程序清单 4.36 SPI 寄存器结构体指针定义

```
/** \brief 模数转换 (ADC) 寄存器块指针 */
#define AMHW_ZLG116_ADC   ((amhw_zlg_adc_v0_t *)ZLG116_ADC_BASE )
```

注意:其中的 ZLG116_ADC_BASE 是 ADC 外设寄存器的基地址，在 {SDK}\ametal\soc\zlg\zlg116\zlg116_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 ADC 寄存器结构体指针宏后，就可以直接使用 ADC 硬件层的相关函数了。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 ADC 中断，可以直接直接设置寄存器的值，详见程序清单 4.37。

程序清单 4.37 直接设置寄存器的值使能 ADC 中断

```
/*
* 设置 ADCR 寄存器的 bit0 为 1，使能 ADC 中断
*/
AMHW_ZLG116_ADC->adcr |= 0x01;
```

注意: 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}\ametal\soc\zlg\zlg116\hw\include\amhw_zlg116_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 4-6。

表 4-6 指向各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 | ADC | AMHW_ZLG116_ADC |
| 2 | BKP | AMHW_ZLG116_BKP |
| 3 | CMP | AMHW_ZLG116_CMP |
| 4 | DMA | AMHW_ZLG116_DMA |
| 5 | EXTI | AMHW_ZLG116_EXTI |
| 6 | FLASH | AMHW_ZLG116_FLASH |
| 7 | GPIO | AMHW_ZLG116_GPIO |
| 8 | GPIOA | AMHW_ZLG116_GPIOA |
| 9 | GPIOB | AMHW_ZLG116_GPIOB |
| 10 | GPIOC | AMHW_ZLG116_GPIOC |
| 11 | GPIOD | AMHW_ZLG116_GPIOD |
| 12 | I2C | AMHW_ZLG116_I2C |
| 13 | INT | AMHW_ZLG116_INT |
| 14 | PWR | AMHW_ZLG116_PWR |
| 15 | RAM | AMHW_ZLG116_RAM |
| 16 | RCC | AMHW_ZLG116_RCC |
| 17 | SPI1 | AMHW_ZLG116_SPI1 |
| 18 | SPI2 | AMHW_ZLG116_SPI2 |
| 19 | SYSCFG | AMHW_ZLG116_SYSCFG |
| 20 | SYSTICK | AMHW_ZLG116_SYSTICK |
| 21 | TIM1 | AMHW_ZLG116_TIM1 |
| 22 | TIM14 | AMHW_ZLG116_TIM14 |
| 23 | TIM16 | AMHW_ZLG116_TIM16 |
| 24 | TIM17 | AMHW_ZLG116_TIM17 |
| 25 | TIM2 | AMHW_ZLG116_TIM2 |
| 26 | TIM3 | AMHW_ZLG116_TIM3 |
| 27 | UART1 | AMHW_ZLG116_UART1 |
| 28 | UART2 | AMHW_ZLG116_UART2 |
| 29 | WWDT | AMHW_ZLG116_WWDT |
| 30 | IWDT | AMHW_ZLG116_IWDT |

# 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 LED 实例初始化函数调用

| 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 {SDK}\ametal\board\am116_core\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

AM116CORE 板上有两个 LED 灯， 默认引脚分别为 PIOB_1 和 PIOB_2， 使用时，

需要使用跳线帽短接 AM116CORE 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_1, PIOB_2};
/** \brief 定义 GPIO LED 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       /* 低电平点亮 */
};
```

其中，am_led_gpio_info_t 类型在 {SDK}\ametal\common\components\service\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 zlg116_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_1 和 PIOB_2 ，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}\ametal\common\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_1 和 PIOB_2，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 TIM16 的输出通道 1（TIM16_OUT1）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 TIM16 的 PWM 功能，若应用程序需要使用 TIM16，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

AM116CORE 有两个板载按键 KEY/RES 和 RST，KEY/RES 的默认引脚为 PIOA_8，使用时，需要使用跳线帽短接 AM116CORE 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。

程序清单 5.4 KEY 相关配置信息

```
static const int     g_key_pins[] = {PIOA_8};
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}\ametal\common\components\service\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 zlg116_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_8 ，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_8，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

AM116CORE 具有 2 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-UART1，2-UART2 |
| ____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 TIM14 定时器为其提供一个周期性的定时中断，默认使用 TIM14 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

AM116CORE 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 系统 TICK 频率配置

```
/* 定义 LM75 实例信息 */
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 5-4，使用方法可以参考 4.3.1。

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。

AM116-Core 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 ZLG116 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。

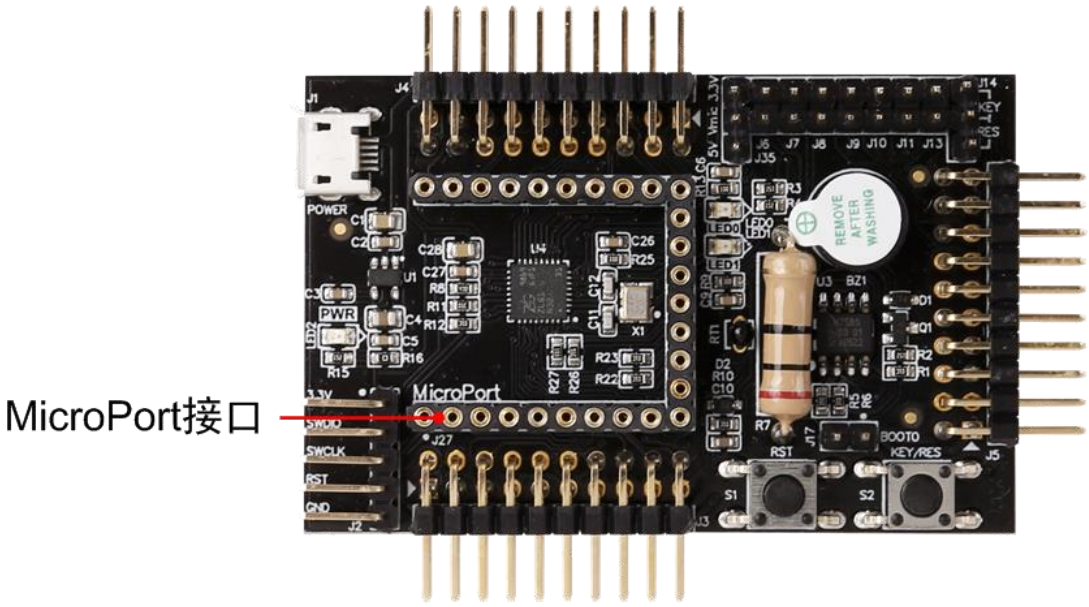


图 6.1 AM116-Core MicroPort

## 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-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-RTC | am_hwconf_microport_rtc.c |
| 5 | 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 4.3.1，也可以参考{SDK}\ametal\examples\am116_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 | 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 | 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-RTC(使用芯片特殊功 | am_pcf85063_handle_t am_microport_rtc_inst_init (void); |
| 10 | MicroPort-RTC(使用通用的 RTC 功能) | am_rtc_handle_t am_microport_rtc_rtc_inst_init (void); |
| 11 | MicroPort-RTC(用作系统时间) | int am_microport_rtc_time_inst_init (void); |
| 12 | MicroPort-RX8025T(使用芯片特殊功能) | am_rx8025t_handle_t am_microport_rx8025t_inst_init (void); |
| 13 | MicroPort-RX8025T(使用通用的 RTC 功能) | am_rtc_handle_t am_microport_rx8025t_rtc_inst_init (void); |
| 14 | MicroPort-RX8025T(用作系统时 | 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 $I^2C$ 接口；
- 支持 1 路 UART 接口；
- 支持 1 路 3.3V 和 1 路 5V 电源接口。

AM116-Core 开发板搭载了 1 路 MiniPort，接口标号 J4，如图 7.1 所示。

注意：由于 ZLG116 芯片 I/O 口有限，AM116-Core 平台中 J3 和 J4 接口定义完全相同，但是 J3 和 J5 一样都是扩展接口。在 I/O 口资源比较多的平台中，J3 扩展接口和 J4 MiniPort 接口的定义是不相同的。

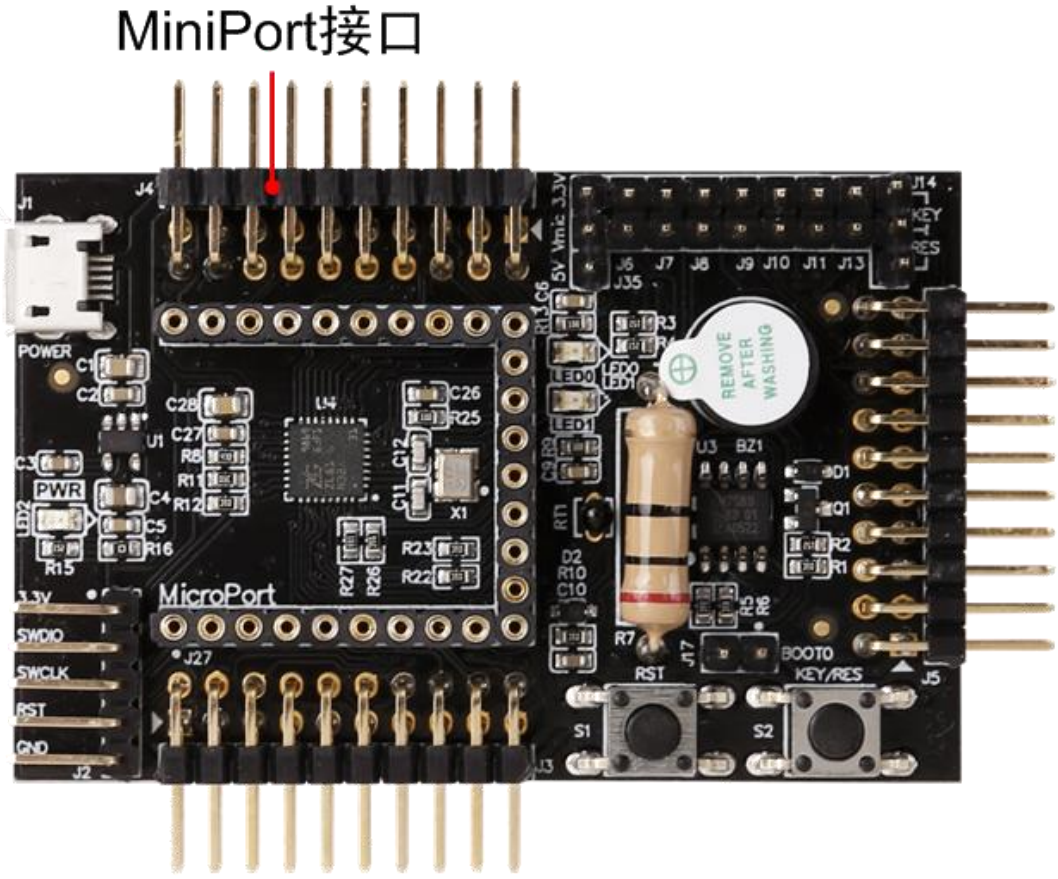


图 7.1 AM116-Core MiniPort

## 7.1 配置文件结构

当前可用的 MiniPort 扩展板有：MiniPort-595、MiniPort-LED 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-LED | am_hwconf_miniport_led.c |
| 3 | 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 4.3.1，也可以参考 {SDK}\ametal\examples\am116_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-LED | int am_miniport_led_inst_init (void); |
| 3 | MiniPort-LED(LED 595 联合初始 | int am_miniport_led_595_inst_init (void); |
| 4 | MiniPort-ZLG72128 | int am_miniport_zlg72128_inst_init (void); |

MiniPort 扩展板通过排母与 AM116-Core 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！