User Manual

AMetal-AM824-Core 用户手册

AMetal

V2.0.2 Date:2019/03/21

产品用户手册

类别	内容
关键词	AM824-Core、功能介绍
摘　要	本文档简述了 AM824-Core 硬件资源，详细介绍了 ametal_am824_core 软件包的结构、配置方法等。

广州周立功单片机科技有限公司

修订历史

版本	日期	原因
发布 2.0.2	2019/3/21	创建文档

目 录

1. 开发平台简介

AM824-Core 开发平台主要用于 LPC824 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 **AM824-Core Rev.A 150917 P/N: 1.14.07.1242**，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。

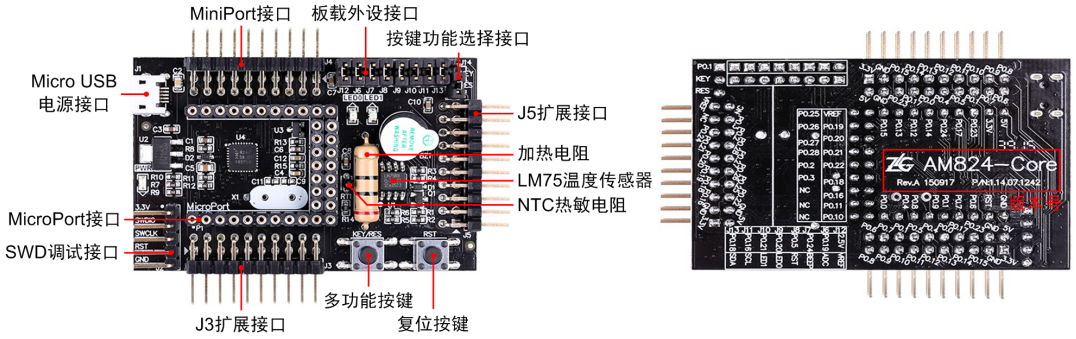

图 1.1: AM824-Core 开发平台

AM824-Core 开发板基于 NXP 半导体的 LPC824M201 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所有 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 NXP LPC824

- Cortex-M0+ 内核，主频可达 30MHz，超低功耗；
- 32KB 的 FLASH，8KB 的 SRAM；
- 29×GPIO，支持单周期 I/O 接口访问，可以实现更快的 IO 操作速度；
- 丰富的串行接口：2×SPI、4×I^2C、3×USART；
- 12 位 12 通道 ADC，高达 1.2Ms/sec 采样率；
- 支持状态可配置定时器 SCT，能实现复杂时序；
- 支持开关矩阵 SWM 功能，数字外设引脚可自由支配；
- WWDT、比较器、硬件 CRC 等。

1.2 AM824-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 REF3330 基准源；
- 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_am824_core 软件包

软件包名为 ametal_am824_core_2.0.0（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 2.2 所示，AMetal 共分为 3 层，硬件层、驱动层和标准接口层。

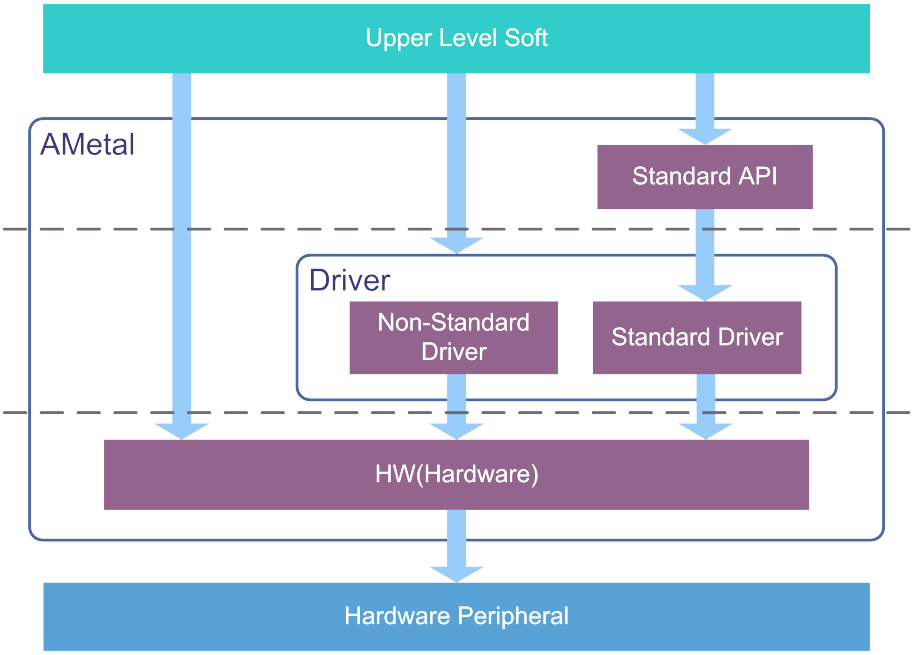

图 2.2: AMetal 框架

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_lpc82x** 、**AMHW_LPC82X** 。

参见:

更多的硬件层接口定义及示例请参考 {SDK}**documents**\《**AMetal AM824-Core API 参考手册.chm**》 或者 {SDK}**ametal\soc\nxp\lpc\lpc82x\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、TSI 等。驱动层接口使用 **am_/AM_ + 芯片名**作为命名空间，如 **am_lpc82x** 、**AM_LPC82X** 。

参见:

更多的驱动层接口定义及示例请参考 {SDK}**documents**\《**AMetal AM824-Core API 参考手册.chm**》 或者 {SDK}**ametal\soc\nxp\lpc\lpc82x\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 AM824-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 4 个文件夹，如 图 2.3 所示。

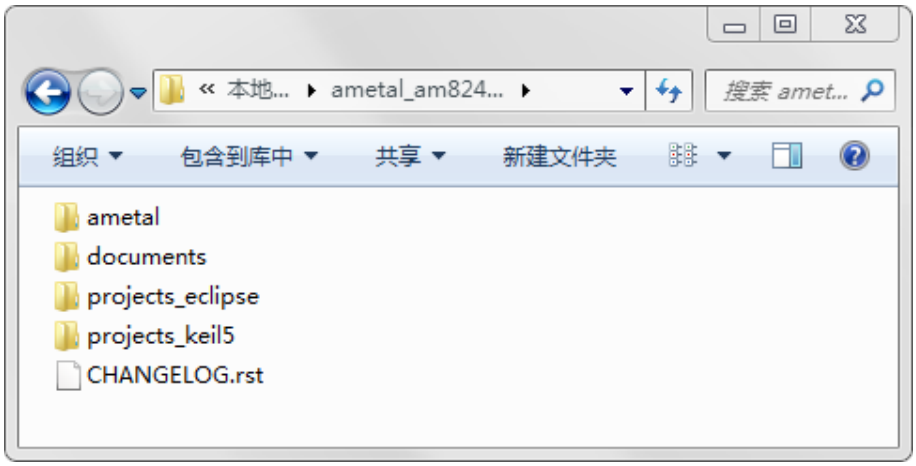

图 2.3: 软件包目录结构

- 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。

注意： projects_eclipse 与 projects_keil5 可能并不是所有的 {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 2.4 所示。

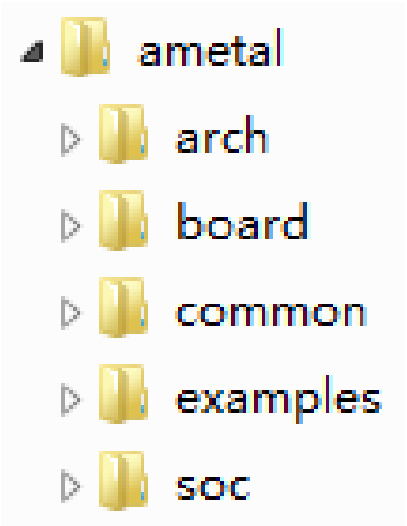

图 2.4: ametal 目录结构

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 2.5 所示：

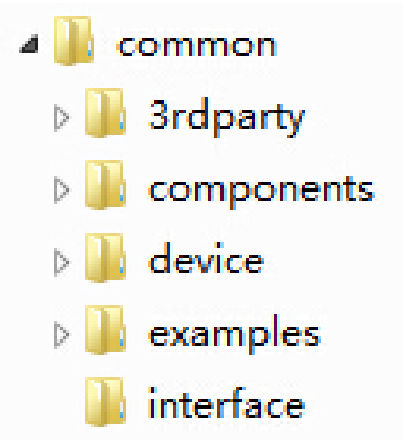

图 2.5: common 目录结构

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

3rdparty 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 AM824-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 2.6 所示。

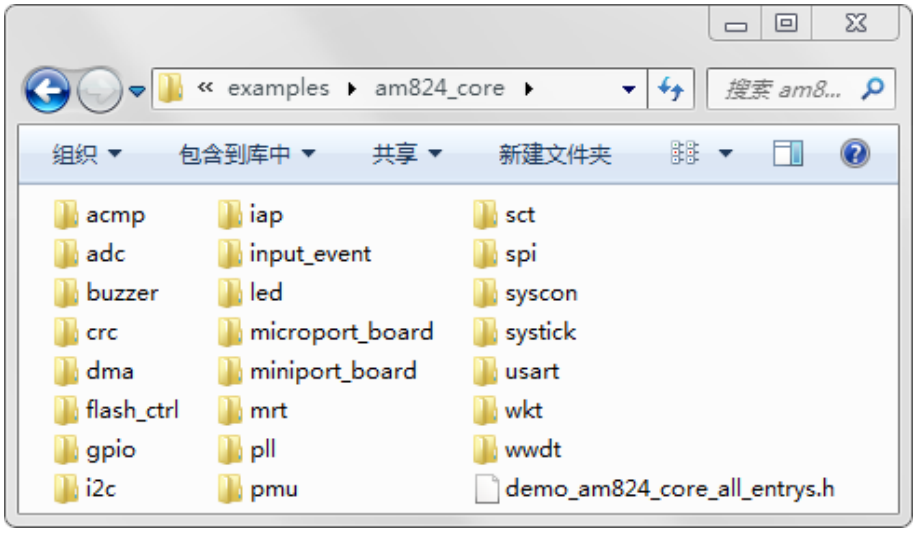

图 2.6: 例程源文件目录视图

在 {SDK}**ametal\examples\am824_core** 文件夹目录包含了各个外设的例程源文件，am824_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_am824_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 SCT 相关外设为例，打开 SCT 目录，如 图 2.7 所示，可以看到该目录下提供了 10 个 demo 源程序。

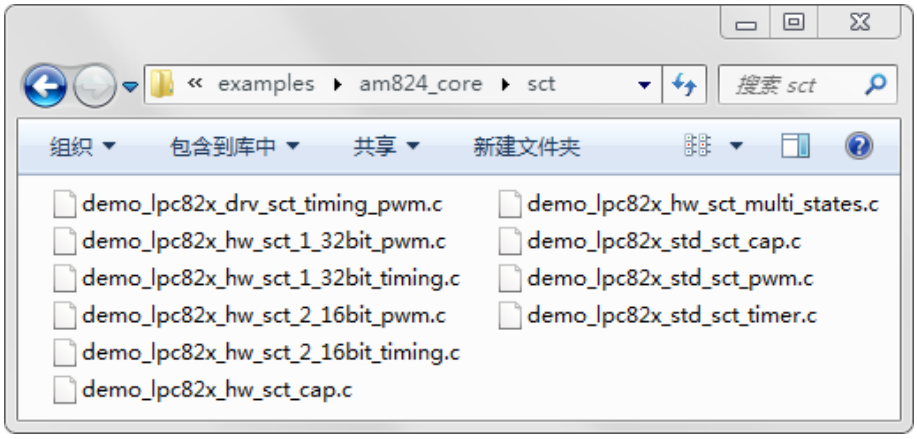

图 2.7: SCT 外设所有 demo 源程序

所有芯片外设的 demo 源程序命名为：demo_lpc82x_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_lpc82x_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_lpc82x_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_lpc82x_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 2.8 所示。

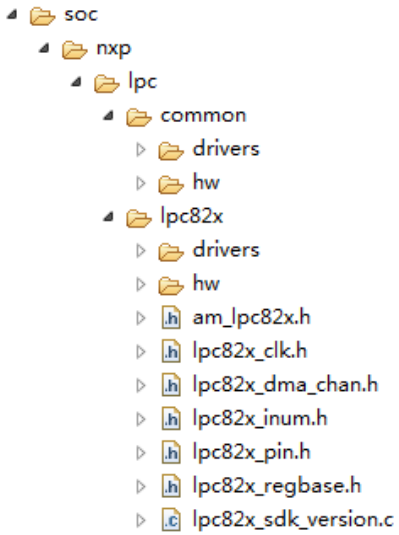

图 2.8: soc 目录结构

在 图 2.8 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\nxp\lpc\lpc82x** 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: LPC82X 芯片各公共文件内容简介

文件名	内容简介
am_lpc82x.h	包含了其它 LPC82X 公共部分头文件，只要使用 LPC82X 的代码，建议默认均包含此头文件
lpc82x_clk.h	时钟 ID 号，如 CLK_ADC0、CLK_SPI0 等
lpc82x_dma_chan.h	DMA 通道号相关定义，如 DMA_CHAN_0，DMA_CHAN_1 等
lpc82x_inum.h	中断号定义，如 INUM_SPI0，INUM_I2C0 等
lpc82x_pin.h	IO 引脚号定义，包含 PIO0 端口的引脚
lpc82x_regbase.h	LPC82X 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_lpc82x.h** 文件中，实际应用程序在使用时，只需要简单的包含 **am_lpc82x.h** 即可。

2.2.2 documents 目录

documents 目录中存放了 5 个文件，分别为《AMetal-AM824-Core 快速入门手册(Keil)》、《AMetal AM824-Core API 参考手册.chm》、《AMetal-AM824-Core 用户手册.pdf》、《AMetal LPC824 引脚配置及查询.xlsm》、《AMetal-AM824-Core 快速入门手册(Eclipse)》。

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，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。

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 2.9 所示。

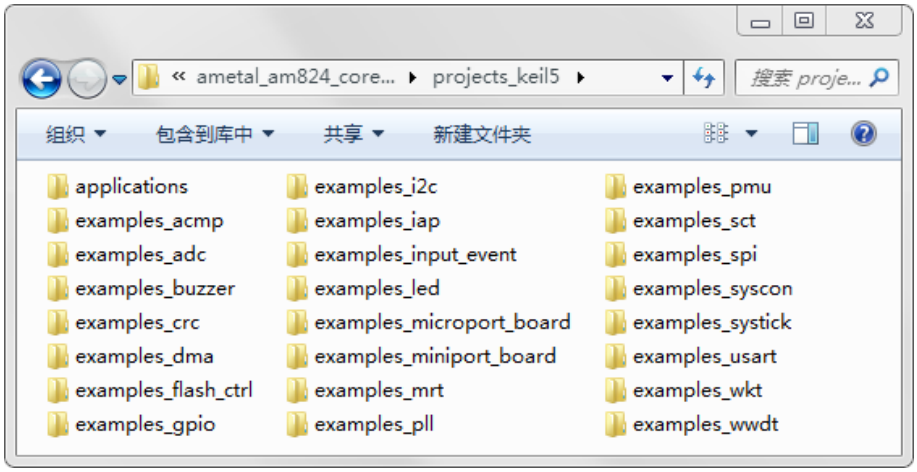

图 2.9: projects_keil5 目录视图

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 2.10 所示。

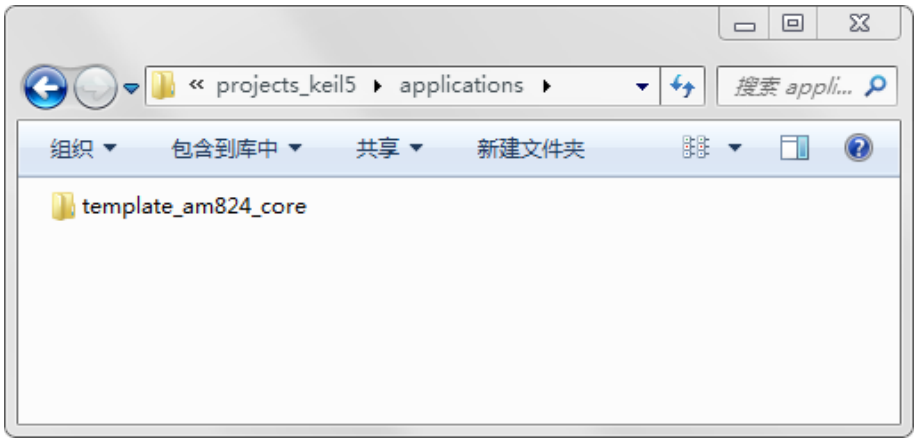

图 2.10: applications 目录视图

如需新建工程，只需要拷贝一份 template_am824_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 2.11 所示，通过拷贝的方式新建了一个命名为 led 的工程。

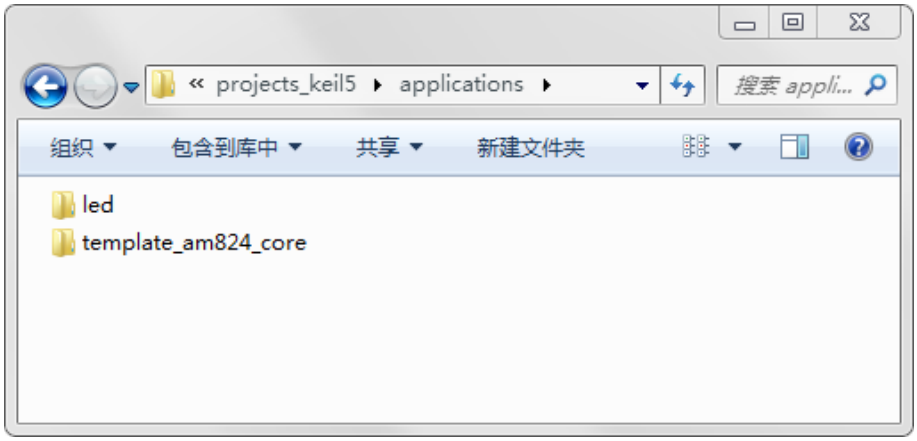

图 2.11: 通过拷贝新建一个 LED 工程目录

注解： 在拷贝工程时，不可任意拷贝至其它目录，只能与 template_am824_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 2.12 所示。template_am824_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_am824_core.uvprojx 重命名为 led.uvprojx。重命名后如 图 2.13 所示。

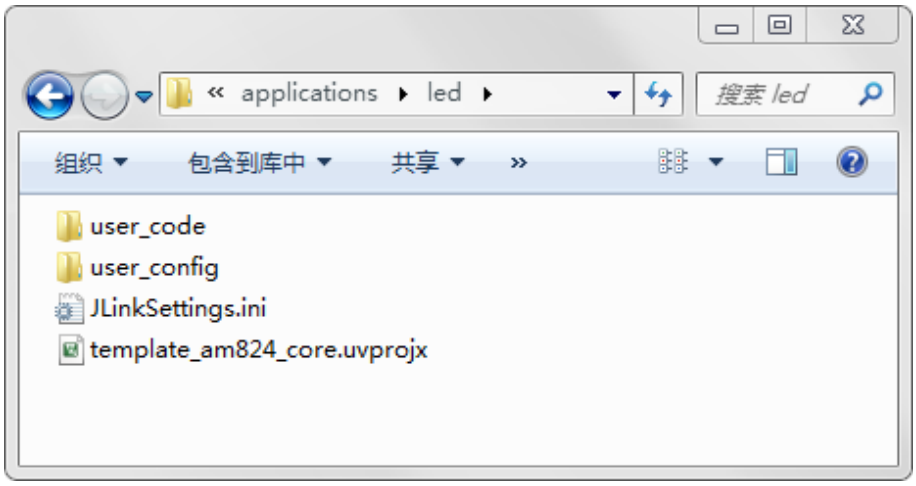

图 2.12: 工程初始视图

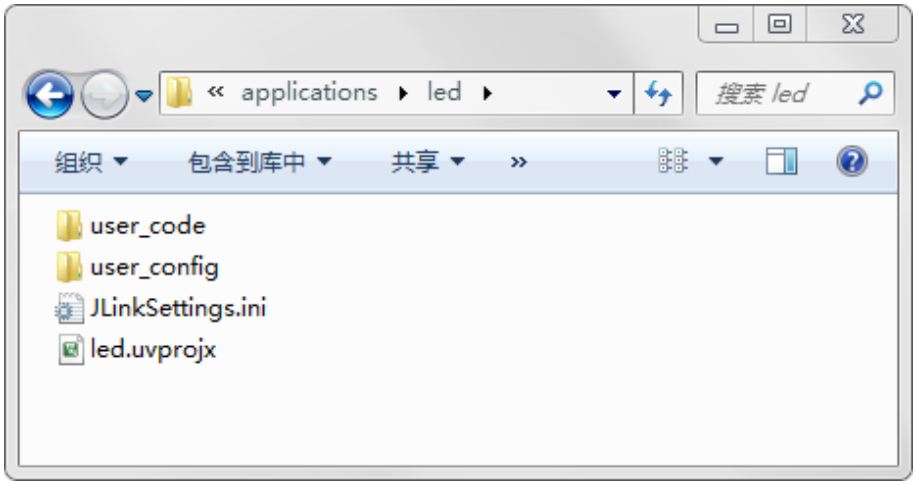

图 2.13: 重命名 Keil 工程文件

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 SCT 外设，打开 examples_sct，如 图 2.14 所示，可以看到有 10 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。

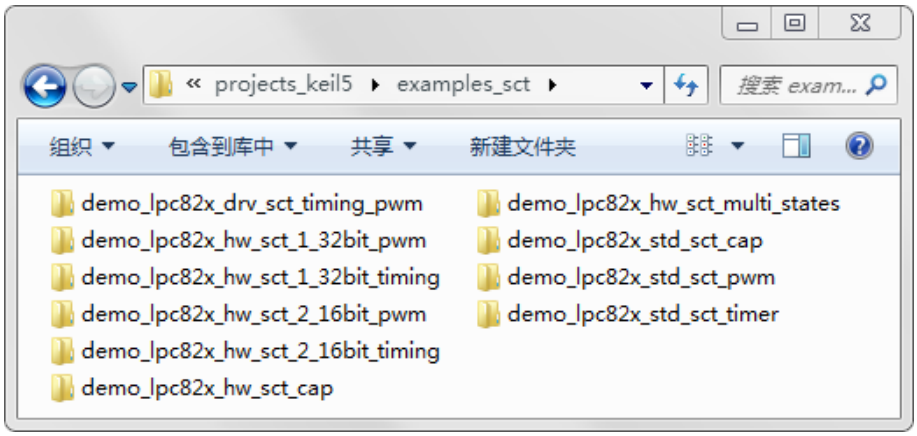

图 2.14: SCT 外设所有例程的工程

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_lpc82x_std_sct_timer 为例，打开该文件夹，如 图 2.15 所示，双击 Keil5 工程文件 demo_lpc82x_std_sct_timer.uvprojx 文件即可打开工程。

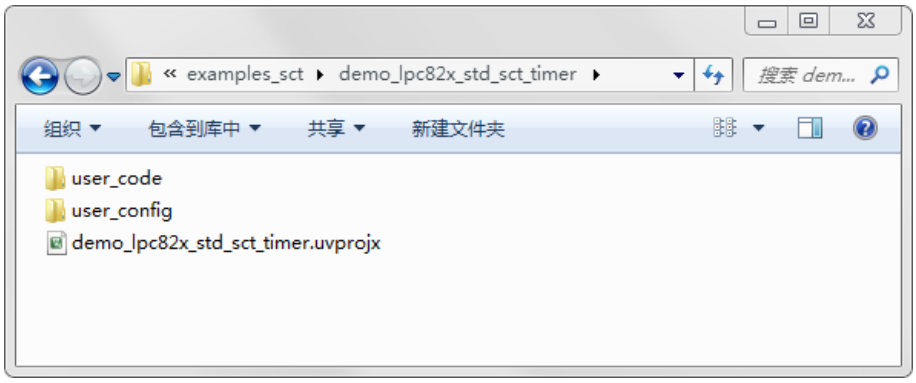

图 2.15: demo_lpc82x_std_sct_timer 工程文件夹

注解： 具体如何编译工程以及下载程序，请参考《AMetal-AM824-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 图 2.16 所示。

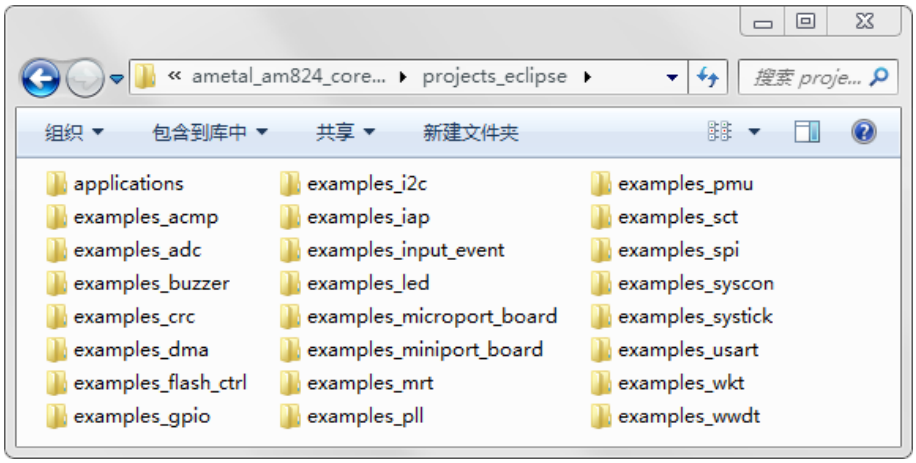

图 2.16: projects_eclipse 目录视图

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 2.17 所示。

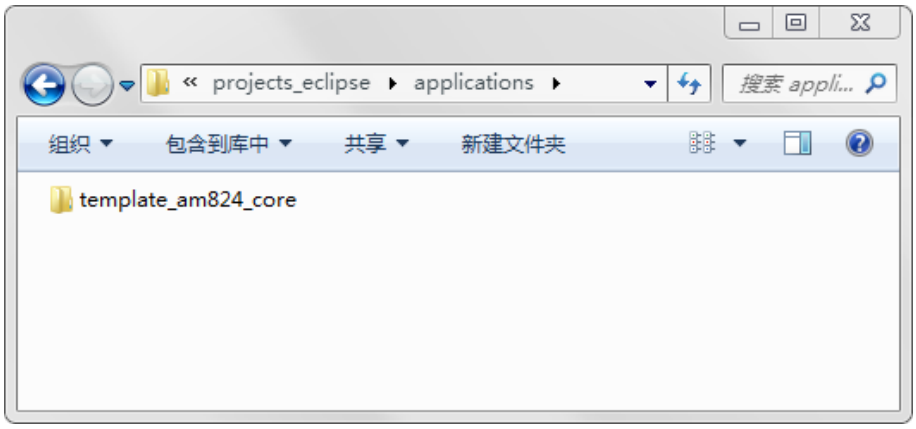

图 2.17: applications 目录视图

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 lpc82x_flash.ld、template_am824_core Debug.launch 和 template_am824_core Release.launch 这三个文件，如 图 2.18 所示。

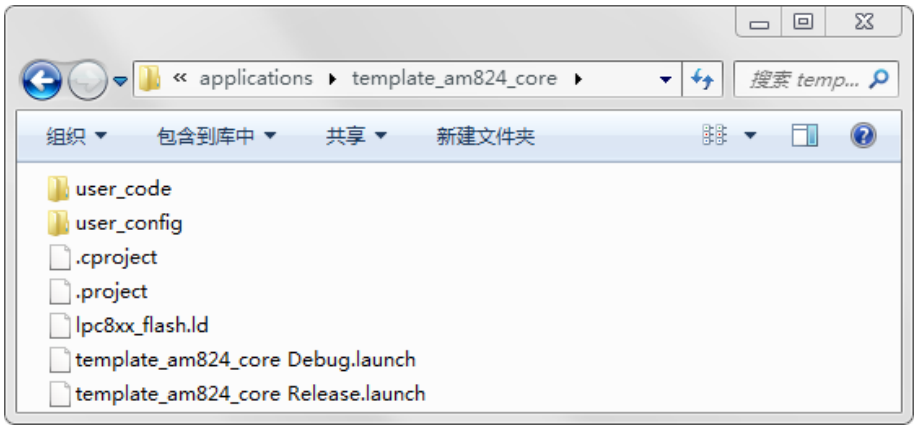

图 2.18: Eclipse 模板工程目录结构

• 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。
- lpc82x_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_am824_core Debug.launch 和 template_am824_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 SCT 外设，打开 examples_sct，如 图 2.19 所示，可以看到有 10 个工程文件夹，文件名与对应源文件的文件名保持一致。

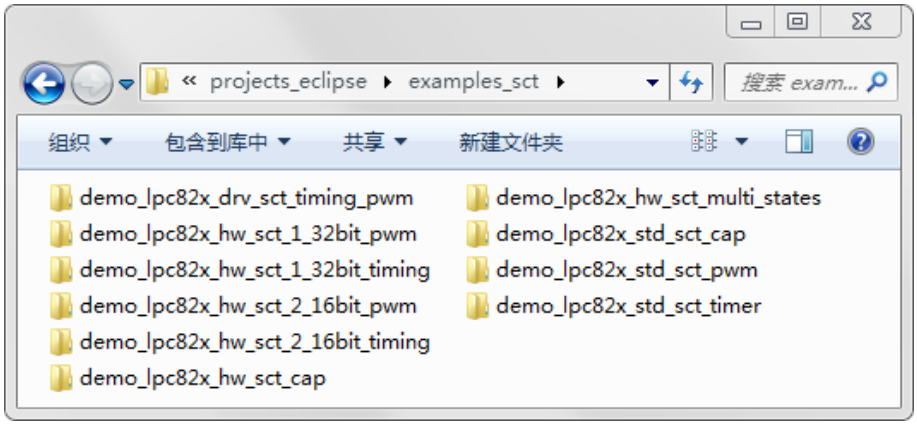

图 2.19: SCT 外设所有例程的工程

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-AM824-Core 快速入门手册 (Eclipse).pdf》。

2.3 工程结构

2.3.1 Keil 工程结构

打开 Keil 版本的 template_am824_core 模板工程，其工程结构如 图 2.20 所示。

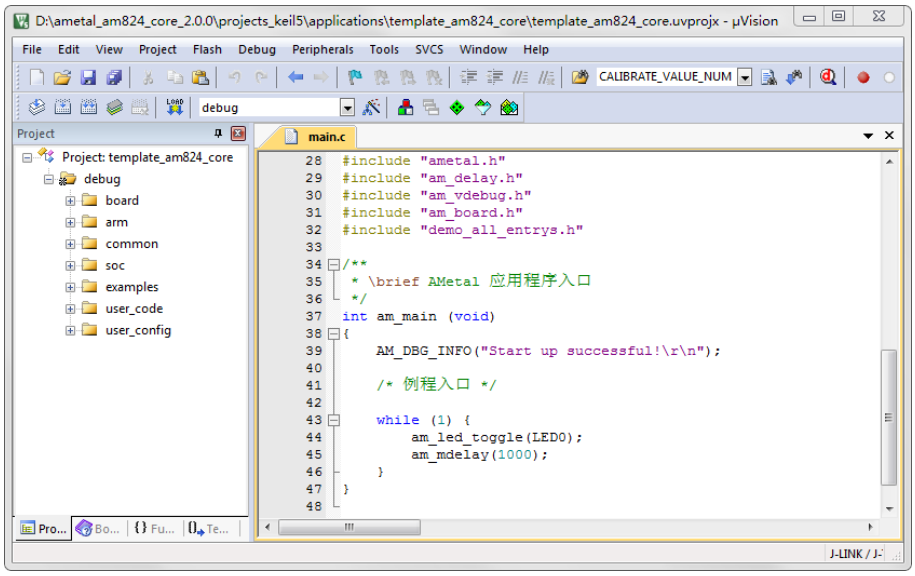

图 2.20: Keil 版本工程模板结构

在工程结构中，包含了 board、arm、common、soc、examples、user_code、user_config 共 7 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。
- **examples** 文件夹主要包含 AM824-Core 开发板各个外设及板载器件的例程源文件，examples 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_am824_core 工程，{PROJECT} 等价于 {SDK}**projects_keil5\applications\template_am824_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_am824_core 模板工程，其工程结构如 图 2.21 所示。

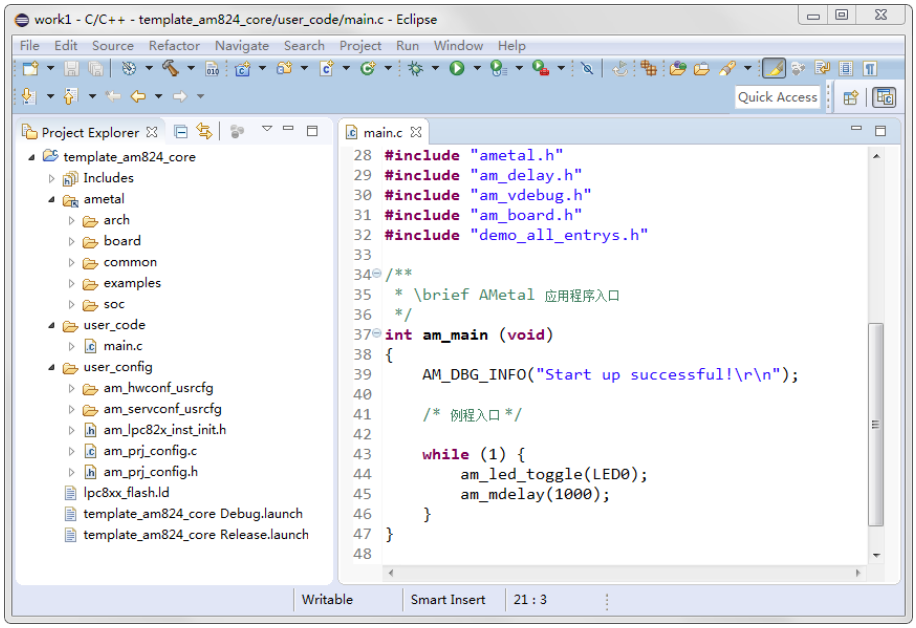

图 2.21: Eclipse 版本工程模板结构

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 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_am824_core 模板工程，{PROJECT} 等价于 {SDK}**projects_eclipses\applications\template_am824_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 列表 3.1 。宏值默认为 1，即 GPIO 外设在系统启动时自动初始化，如果确定系统不使用 GPIO 资源或希望由应用程序自行完成初始化操作，则可以将该宏的宏值修改为 0。

列表 3.1: GPIO 自动初始化使能/禁能配置

```
1 /** \brief 为 1，启动时，完成 GPIO 初始化 */
2 #define AM_CFG_GPIO_ENABLE              1
```

其它一些外设初始化使能/禁能宏定义详见 表 3.2 。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 3.2: 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 列表 3.2 。宏值默认为 1，即 LED 在系统启动时自动完成初始化，如果确定系统不使用 LED 资源或希望由应用程序自行完成初始化操作，则可以将该宏的宏值修改为 0。

列表 3.2: LED 自动初始化使能/禁能配置

```
1 /**
2  * \brief 如果为 1，则初始化 LED 的相关功能，板上默认有两个 LED
3  *
4  * ID: 0 --- PIO0_20 （需要短接跳线帽 J9）
5  * ID: 1 --- PIO0_21 （需要短接跳线帽 J10）
6  */
7 #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 3.3 。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 3.3: 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 MRT 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 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 {PROJECT}**board\am_delay.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. 片上外设资源

LPC82x 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_lpc82x_* 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，它们的配置文件位于 {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 4.4 所示。

表 4.4: 片上外设及对应的配置文件

序号	外设	配置文件
1	ADC(中断方式)	am_hwconf_lpc82x_adc0_int.c
2	ADC(DMA 方式)	am_hwconf_lpc82x_adc0_dma.c
3	时钟部分 (CLK)	am_hwconf_lpc82x_clk.c
4	循环冗余检验 (CRC)	am_hwconf_lpc82x_crc.c
5	DMA	am_hwconf_lpc82x_dma.c
6	GPIO	am_hwconf_lpc82x_gpio.c
7	I^2C0	am_hwconf_lpc82x_i2c0.c
8	I^2C1	am_hwconf_lpc82x_i2c1.c
9	I^2C2	am_hwconf_lpc82x_i2c2.c
10	I^2C3	am_hwconf_lpc82x_i2c3.c
11	多速率定时器 (MRT)	am_hwconf_lpc82x_mrt.c
12	状态可配置定时器 (SCT0)	am_hwconf_lpc82x_sct0.c
13	SCT0 用作捕获功能 (SCT0_CAP)	am_hwconf_lpc82x_sct0_cap.c
14	SCT0 用作 PWM 功能 (SCT0_PWM)	am_hwconf_lpc82x_sct0_pwm.c
15	SCT0 用作定时功能 (SCT0_Timing)	am_hwconf_lpc82x_sct0_timing.c
16	SPI0(DMA 方式)	am_hwconf_lpc82x_spi0_dma.c
17	SPI0(中断方式)	am_hwconf_lpc82x_spi0_int.c
18	SPI1(DAM 方式)	am_hwconf_lpc82x_spi1_dma.c
19	SPI1(中断方式)	am_hwconf_lpc82x_spi1_int.c
20	USART0	am_hwconf_lpc82x_usart0.c
21	USART1	am_hwconf_lpc82x_usart1.c
22	USART2	am_hwconf_lpc82x_usart2.c
23	自动唤醒定时器 (WKT)	am_hwconf_lpc82x_wkt.c
24	窗口看门狗 (WWDT)	am_hwconf_lpc82x_wwdt.c
25	NVIC 中断	am_hwconf_arm_nvic.c
26	滴答定时器 (Systick)	am_hwconf_arm_systick.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_lpc82x_gpio.c**，可以看到设备实例已经定义好。详见 列表 4.1 。

列表 4.1: 定义设备实例

```
1  /** \brief GPIO 设备实例 */
2  am_local am_lpc82x_gpio_dev_t __g_lpc82x_gpio_dev;
```

这里使用 **am_lpc82x_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\nxp\lpc\lpc82x\drivers\include\am_lpc82x_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_lpc82x_gpio.c**，可以看到定义的设备信息如 列表 4.2 所示。

列表 4.2: GPIO 设备信息定义

```
1  /** \brief GPIO 设备信息 */
2  am_local am_const am_lpc82x_gpio_devinfo_t __g_lpc82x_gpio_devinfo = {
3
4      LPC82X_SWM_BASE,              /* SWM 寄存器块基址 */
5      LPC82X_GPIO_BASE,             /* GPIO 寄存器块基址 */
6      LPC82X_IOCON_BASE,            /* IOCON 寄存器块基址 */
7      LPC82X_PINT_BASE,             /* PINT 寄存器块基址 */
8      {
9          INUM_PIN_INT0,            /* PINT0 中断号 */
10         INUM_PIN_INT1,            /* PINT1 中断号 */
11         INUM_PIN_INT2,            /* PINT2 中断号 */
12         INUM_PIN_INT3,            /* PINT3 中断号 */
13         INUM_PIN_INT4,            /* PINT4 中断号 */
14         INUM_PIN_INT5,            /* PINT5 中断号 */
15         INUM_PIN_INT6,            /* PINT6 中断号 */
16         INUM_PIN_INT7,            /* PINT7 中断号 */
17     },
18     __GPIO_PINT_USE_COUNT,        /* GPIO 支持的引脚中断号数量 */
19     &__g_gpio_infomap[0],         /* 引脚触发信息映射 */
20     &__g_gpio_triginfos[0],       /* 引脚触发信息内存 */
21     __lpc82x_gpio_plfm_init,      /* 平台初始化函数 */
22     __lpc82x_gpio_plfm_deinit     /* 平台解初始化函数 */
23 };
```

这里使用 **am_lpc82x_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\nxp\lpc\lpc82x\drivers\include\am_lpc82x_gpio.h** 文件中定义。详见 列表 4.3 。

列表 4.3: GPIO 设备信息结构体类型定义

```
/**
 * \brief GPIO 设备信息
 */
typedef struct am_lpc82x_gpio_devinfo {

    /** \brief SWM 寄存器块基址 */
    uint32_t  swm_regbase;

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

    /** \brief IOCON 寄存器块基址 */
    uint32_t iocon_regbase;

    /** \brief 引脚中断寄存器块基址 */
    uint32_t  pint_regbase;

    /** \brief GPIO 引脚中断号列表 */
    const int8_t inum_pin[AMHW_LPC82X_PINT_CHAN_NUM];

    /** \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_lpc82x_gpio_trigger_info *p_triginfo;

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

    /** \brief 平台解初始化函数 */
    void     (*pfn_plfm_deinit)(void);

} am_lpc82x_gpio_devinfo_t;
```

可见，共计有 10 个成员，看似很多，这是由于 GPIO 关联的外设较多，SWM、GPIO、IOCON、PINT 等都统一归到 GPIO 下管理，仅寄存器基地址就有四个成员。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 SWM、GPIO、IOCON、PINT 共计 4 个部分的外设，因此，在 GPIO 的设备信息中，需要四个基地址，对应四个成员变量，分别为：swm_regbase、gpio_regbase、iocon_regbase 和 pint_regbase。

寄存器基地址已经在 {SDK}**ametal\soc\nxp\lpc\lpc82x\lpc82x_regbase.h** 文件中使用宏定义好了，用户直接使用即可。对于 GPIO 相关的寄存器基地址，详见 列表 4.4 。

列表 4.4: 外设寄存器基地址定义

```
/** \brief IOCON 基地址          */
#define LPC82X_IOCON_BASE           (0x40044000UL)

/** \brief GPIO 通道基地址      */
#define LPC82X_GPIO_BASE            (0xA0000000UL)

/** \brief PIN_INT 基地址       */
#define LPC82X_PINT_BASE            (0xA0004000UL)

/** \brief SWM 基地址          */
#define LPC82X_SWM_BASE             (0x4000C000UL)
```

可见，列表 4.2 中，设备信息前两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 PINT，PINT 最高可提供 8 路中断，为了快速响应，与之对应地，系统提供了 8 路中断服务入口给 PINT，因此，PINT 共计有 8 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 8 的数组。详见 列表 4.5 。

列表 4.5: GPIO 设备信息结构体类型——中断号成员定义

```
/** \brief GPIO 引脚中断号列表 */
const int8_t inum_pin[AMHW_LPC82X_PINT_CHAN_NUM];
```

其中。AMHW_LPC82X_PINT_CHAN_NUM 在 PINT 的 HW 层文件（路径：{SDK}**ametal\lpc82x\hwinclude\amhw_lpc82x_pint.h** ）定义，详见 列表 4.6 。

列表 4.6: PINT 中断通道数量定义

```
/** \brief 中断通道数量 */
#define AMHW_LPC82X_PINT_CHAN_NUM     8
```

所有中断号已经在 {SDK}**ametal\soc\nxp\lpc\lpc82x\lpc82x_inum.h** 文件中定义好了，与 PINT 相关的中断号定义详见 列表 4.7 。

列表 4.7: PINT 各个中断号定义

```
#define INUM_PIN_INT0            24   /**< \brief 引脚中断 0 */
#define INUM_PIN_INT1            25   /**< \brief 引脚中断 1 */
#define INUM_PIN_INT2            26   /**< \brief 引脚中断 2 */
#define INUM_PIN_INT3            27   /**< \brief 引脚中断 3 */
#define INUM_PIN_INT4            28   /**< \brief 引脚中断 4 */
#define INUM_PIN_INT5            29   /**< \brief 引脚中断 5 */
#define INUM_PIN_INT6            30   /**< \brief 引脚中断 6 */
#define INUM_PIN_INT7            31   /**< \brief 引脚中断 7 */
```

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

3. 时钟 ID 号

时钟 ID 号对应了外设的时钟来源，需要将该时钟 ID 号传递给驱动，以

便驱动中可以概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}**ametal\soc\nxp\lpc\lpc82x\lpc82x_clk.h** 文件中定义好了，详见 列表 4.8 。

列表 4.8: 时钟 ID 号

```
#define CLK_FLASH       (4 << 8)     /**< \brief FLASH 时钟          */
#define CLK_I2C0        (5 << 8)     /**< \brief I2C0 时钟           */
#define CLK_GPIO        (6 << 8)     /**< \brief GPIO 时钟           */
#define CLK_SWM         (7 << 8)     /**< \brief SWM 时钟            */
#define CLK_SCT         (8 << 8)     /**< \brief SCT 时钟            */
#define CLK_WKT         (9 << 8)     /**< \brief WKT 时钟            */
#define CLK_MRT         (10 << 8)    /**< \brief MRT 时钟            */
#define CLK_SPI0        (11 << 8)    /**< \brief SPI0 时钟           */
#define CLK_SPI1        (12 << 8)    /**< \brief SPI1 时钟           */
#define CLK_CRC         (13 << 8)    /**< \brief CRC 时钟            */
#define CLK_UART0       (14 << 8)    /**< \brief UART0 时钟          */
#define CLK_UART1       (15 << 8)    /**< \brief UART1 时钟          */
#define CLK_UART2       (16 << 8)    /**< \brief UART2 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 列表 4.9 。

列表 4.9: 串口外设时钟 ID 号

```
/** \brief USART0 设备信息 */
am_local am_const am_lpc_usart_devinfo_t __g_lpc82x_usart0_devinfo = {
    LPC82X_USART0_BASE,             /* USART0 寄存器块基地址 */
    INUM_USART0,                    /* USART0 中断号 */
    CLK_UART0,                      /* USART0 时钟号 */
    __lpc82x_plfm_usart0_init,      /* 平台初始化函数 */
    __lpc82x_plfm_usart0_deinit,    /* 平台解初始化函数 */
    NULL                            /* 无 RS485 方向控制函数 */
};
```

4. 需要用户根据实际情况分配的内存

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

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

以 PINT 为例，系统提供了 8 路中断，因此，最多可以将 8 个 GPIO 用作触发模式。每一路 GPIO 触发模式就需要内存来保存用户设定的触发回调函数。如果按照默认，假定用户可能会使用到所有的 8 路 GPIO 触发，则就需要 8 份用于保存相关信息的内存空间。而实际上，用户可能只使用 1 路，这就导致了不必要的空间浪费。

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

在设备信息结构体类型中，相关的成员有 3 个，详见 列表 4.10 。

列表 4.10: 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_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 */
```

```
8  struct am_lpc82x_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_lpc82x_gpio_trigger_info** 同样在 GPIO 驱动头文件 {SDK}**ametal\soc\nxp\lpc\lpc82x\drivers\include\am_lpc82x_gpio.h** 中定义，详见 列表 4.11 。

列表 4.11: GPIO 触发信息结构体类型定义

```
1  /**
2   * \brief 引脚的触发信息
3   */
4  struct am_lpc82x_gpio_trigger_info {
5
6      /** \brief 触发回调函数 */
7      am_pfnvoid_t  pfn_callback;
8
9      /** \brief 回调函数的参数 */
10     void *p_arg;
11 };
```

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，提供了默认的这三个成员的值定义，详见 列表 4.12 。

列表 4.12: 需要用户根据实际情况分配的内存定义

```
1  /**
2   * \brief 使用的中断通道数量
3   *
4   * 默认使用所有的中断通道，用户可以根据实际使用通道数，更改此值，减少内存的占用
5   *
6   * \note 如果此值为 0，将无法使用 GPIO 中断功能，但是可以使用其他 GPIO 功能
7   */
8  #define __GPIO_PINT_USE_COUNT    AMHW_LPC82X_PINT_CHAN_NUM
9
10 /** \brief 引脚触发信息内存 */
11 am_local
12 struct am_lpc82x_gpio_trigger_info __g_gpio_triginfos[__GPIO_PINT_USE_COUNT];
13
14 /** \brief 引脚触发信息映射 */
15 am_local uint8_t                    __g_gpio_infomap[__GPIO_PINT_USE_COUNT];
```

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

默认情况下，**__GPIO_PINT_USE_COUNT** 的值即为硬件支持的最大 PINT 通道数目，如果用户实际使用到的 GPIO 触发数目小于该值，可以修改为实际使用到的数量，如 5。

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

该部分内容。

5. 平台初始化函数

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

在设备信息结构体类型中，均有一个用于存放平台初始化函数的指针，以指向平台初始化函数，详见 列表 4.13 。当驱动程序初始化相应外设前，将首先调用设备信息中提供的平台初始化函数。

列表 4.13: 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 {HWCONFIG}**am_hwconf_lpc82x_gpio.c** 文件中定义，详见 列表 4.14 。

列表 4.14: GPIO 平台初始化函数

```
/**
 * \brief GPIO 平台初始化
 */
am_local void __lpc82x_gpio_plfm_init (void)
{
    amhw_lpc82x_syscon_periph_reset(AMHW_LPC82X_RESET_GPIO);

    amhw_lpc82x_clk_periph_enable(AMHW_LPC82X_CLK_GPIO);
    amhw_lpc82x_clk_periph_enable(AMHW_LPC82X_CLK_IOCON);
    amhw_lpc82x_clk_periph_enable(AMHW_LPC82X_CLK_SWM);
}
```

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。

amhw_lpc82x_syscon_periph_reset() 函数用于复位一个外设，在 {SDK}**ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_syscon.h** 文件中定义。函数原型详见 列表 4.15 。

列表 4.15: amhw_lpc82x_syscon_periph_reset() 函数原型

```
void amhw_lpc82x_syscon_periph_reset (amhw_lpc82x_syscon_periph_reset_t periph);
```

参数为 **amhw_lpc82x_syscon_periph_reset_t** 类型的枚举值，用于指定需要使能的外设时钟门控，同样在 {SDK}**ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_syscon.h** 文件中定义。详见 列表 4.16 。

列表 4.16: amhw_lpc82x_syscon_periph_reset_t 类型定义

```
typedef enum amhw_lpc82x_syscon_periph_reset {
    AMHW_LPC82X_RESET_SPI0 = 0,             /**< \brief SPI0 */
    AMHW_LPC82X_RESET_SPI1,                 /**< \brief SPI1 */
    AMHW_LPC82X_RESET_UARTFRG ,             /**< \brief UARTFRG */
    AMHW_LPC82X_RESET_UART0,                /**< \brief UART0 */
    AMHW_LPC82X_RESET_UART1,                /**< \brief UART1 */
    AMHW_LPC82X_RESET_UART2,                /**< \brief UART2 */
    AMHW_LPC82X_RESET_I2C0,                 /**< \brief I2C0 */
    AMHW_LPC82X_RESET_MRT,                  /**< \brief MRT */
```

```
    AMHW_LPC82X_RESET_SCT,                  /**< \brief SCT */
    AMHW_LPC82X_RESET_WKT,                  /**< \brief WKT */
    AMHW_LPC82X_RESET_GPIO,                 /**< \brief GPIO */
    AMHW_LPC82X_RESET_FLASH,                /**< \brief FLASH */
    AMHW_LPC82X_RESET_ACMP,                 /**< \brief ACMP */
    AMHW_LPC82X_RESET_I2C1 = 14,            /**< \brief I2C1 */
    AMHW_LPC82X_RESET_I2C2,                 /**< \brief I2C2 */
    AMHW_LPC82X_RESET_I2C3                  /**< \brief I2C3 */
} amhw_lpc82x_syscon_periph_reset_t;
```

在平台初始化函数中，参数 AMHW_LPC82X_RESET_GPIO 表示复位 GPIO 外设。

amhw_lpc82x_clk_periph_enable() 函数用于使能一个外设的时钟，在{SDK}**ametal\soc\nxp\lpc\lpc82xhwincludeamhw_lpc82x_clk.h** 文件中定义。函数原型详见 列表 4.17 。

列表 4.17: amhw_lpc82x_clk_periph_enable() 函数原型

```
void amhw_lpc82x_clk_periph_enable (amhw_lpc82x_clk_periph_t clk);
```

参数为 amhw_lpc82x_clk_periph_t 类型的枚举值，用于指定需要使能时钟的外设，同样在 {SDK}**ametal\soc\nxp\lpc\lpc82x\hw\includeamhw_lpc82x_clk.h** 文件中定义。详见 列表 4.18 。

列表 4.18: amhw_lpc82x_clk_periph_t 类型定义

```
typedef enum amhw_lpc82x_clk_periph {
    AMHW_LPC82X_CLK_SYS = 0,            /**< \brief System 时钟          */
    AMHW_LPC82X_CLK_ROM,                /**< \brief ROM 时钟             */
    AMHW_LPC82X_CLK_SRAM0_1,            /**< \brief SRAM0 SRAM1 时钟     */
    AMHW_LPC82X_CLK_FLASHREG,           /**< \brief FLASH 寄存器接口时钟 */
    AMHW_LPC82X_CLK_FLASH,              /**< \brief FLASH 时钟           */
    AMHW_LPC82X_CLK_I2C0,               /**< \brief I2C0 时钟             */
    AMHW_LPC82X_CLK_GPIO,               /**< \brief GPIO 时钟            */
    AMHW_LPC82X_CLK_SWM,                /**< \brief SWM 时钟             */
    AMHW_LPC82X_CLK_SCT,                /**< \brief SCT 时钟             */
    AMHW_LPC82X_CLK_WKT,                /**< \brief WKT 时钟             */
    AMHW_LPC82X_CLK_MRT,                /**< \brief MRT 时钟              */
    AMHW_LPC82X_CLK_SPI0,               /**< \brief SPI0 时钟            */
    AMHW_LPC82X_CLK_SPI1,               /**< \brief SPI1 时钟            */
    AMHW_LPC82X_CLK_CRC,                /**< \brief CRC 时钟             */
    AMHW_LPC82X_CLK_UART0,              /**< \brief UART0 时钟           */
    AMHW_LPC82X_CLK_UART1,              /**< \brief UART1 时钟           */
    AMHW_LPC82X_CLK_UART2,              /**< \brief UART2 时钟           */
    AMHW_LPC82X_CLK_WWDT,               /**< \brief WWDT 时钟            */
    AMHW_LPC82X_CLK_IOCON,              /**< \brief IOCON 时钟           */
    AMHW_LPC82X_CLK_ACMP,               /**< \brief ACMP 时钟            */
    AMHW_LPC82X_CLK_I2C1 = 21,          /**< \brief I2C1 时钟            */
    AMHW_LPC82X_CLK_I2C2,               /**< \brief I2C2 时钟            */
    AMHW_LPC82X_CLK_I2C3,               /**< \brief I2C3 时钟            */
    AMHW_LPC82X_CLK_ADC0,               /**< \brief ADC 时钟             */
    AMHW_LPC82X_CLK_MTB = 26,           /**< \brief MTB 时钟             */
    AMHW_LPC82X_CLK_DMA = 29            /**< \brief DMA 时钟             */
} amhw_lpc82x_clk_periph_t;
```

平台初始化函数中，参数 AMHW_LPC82X_CLK_GPIO、AMHW_LPC82X_CLK_IOCON 和 AMHW_LPC82X_CLK_SWM 分别使能了 GPIO、IOCON 和 SWM 的时钟。

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

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

6. 平台解初始化函数

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

在设备信息结构体类型中，均有一个用于存放平台解初始化函数的指针，以指向平台解初始化函数，详见 列表 4.19 。当不再需要使用某个外设时，驱动在解初始化相应外设后，将调用设备信息中提供的平台解初始化函数，以释放掉平台提供的相关资源。

列表 4.19: GPIO 设备信息结构体类型——平台解初始化函数指针定义

```
1  /** \brief 平台解初始化函数 */
2  void (*pfn_plfm_deinit)(void);
```

平台解初始化函数均在设备配置文件中定义，GPIO 的平台解初始化函数在 {HWCONFIG}**\am_hwconf_lpc82x_gpio.c** 文件中定义，详见 列表 4.20 。

列表 4.20: GPIO 平台解初始化函数

```
1  /**
2   * \brief GPIO 平台解初始化
3   */
4  am_local void __lpc82x_gpio_plfm_deinit (void)
5  {
6      amhw_lpc82x_syscon_periph_reset(AMHW_LPC82X_RESET_GPIO);
7
8      amhw_lpc82x_clk_periph_disable(AMHW_LPC82X_CLK_GPIO);
9      amhw_lpc82x_clk_periph_disable(AMHW_LPC82X_CLK_IOCON);
10     amhw_lpc82x_clk_periph_disable(AMHW_LPC82X_CLK_SWM);
11 }
```

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。

amhw_lpc82x_clk_periph_disable() 函数与 **amhw_lpc82x_clk_periph_enable()** 对应，用于关闭相关外设的时钟，该函数在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_clk.h** 文件中定义。函数原型详见 列表 4.21 。

列表 4.21: amhw_lpc82x_clk_periph_disable() 函数原型

```
1  void amhw_lpc82x_clk_periph_disable (amhw_lpc82x_clk_periph_t clk);
```

参数为 **amhw_lpc82x_clk_periph_t** 类型的枚举值，用于指定需要关闭时钟的外设，同样在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_clk.h** 文件中定义。详见 列表 4.16 。

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

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

综上，以 GPIO 为例，讲述了设备信息结构体中的 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 10 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 **__GPIO_PINT_USE_COUNT**（详见 列表 4.12 ）。

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}**\ametal\soc\nxp\lpc\lpc82x\drivers\include\am_lpc82x_gpio.h** 中声明。详见 列表 4.22 。

列表 4.22: 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_lpc82x_gpio_init (am_lpc82x_gpio_dev_t             *p_dev,
                         const am_lpc82x_gpio_devinfo_t   *p_devinfo);
```

因此，要完成 GPIO 的初始化，只需要调用一下该函数即可，详见 列表 4.23 。

列表 4.23: 完成 GPIO 初始化

```
am_lpc82x_gpio_init(&__g_lpc82x_gpio_dev, &__g_lpc82x_gpio_devinfo);
```

__g_lpc82x_gpio_dev 和 **__g_lpc82x_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_lpc82x_gpio.c** 文件中，详见 列表 4.24 。

列表 4.24: GPIO 实例初始化函数

```
/**
 * \brief GPIO 实例初始化
 */
int am_lpc82x_gpio_inst_init (void)
{
    return am_lpc82x_gpio_init(&__g_lpc82x_gpio_dev, &__g_lpc82x_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\nxp\lpc\lpc82x\drivers\include\am_lpc82x_gpio.h** 中声明。详见 列表 4.25 。

列表 4.25: GPIO 解初始化函数

```
1  /**
2   * \brief GPIO 解初始化
3   *
4   * \param[in] p_dev : 指向 GPIO 设备的指针
5   *
6   * \return 无
7   */
8  void am_lpc82x_gpio_deinit (void);
```

当应用不再使用该外设时，只需要调用一下该函数即可，详见 列表 4.26 。

列表 4.26: 完成 GPIO 解初始化

```
1  am_lpc82x_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_lpc82x_gpio.c** 文件中，详见 列表 4.27 。

列表 4.27: GPIO 实例解初始化函数

```
/**
 * \brief GPIO 实例解初始化
 */
void am_lpc82x_gpio_inst_deinit (void)
{
    am_lpc82x_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

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，对于 AM824-Core，VREFN 接地，VREFP 默认接 TL431 基准源输出（2.5V）。因此，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 列表 4.28 。

列表 4.28: ADC 参考电压默认配置

```
/** \brief ADC0 (中断方式) 设备信息 */
am_local am_const am_lpc82x_adc_int_devinfo_t __g_lpc82x_adc0_int_devinfo = {
    LPC82X_ADC0_BASE,                   /* ADC0 寄存器块基地址 */
    CLK_ADC0,                           /* ADC0 时钟号 */
    2500,                               /* ADC 参考电压，单位：mV */
    INUM_ADC0_SEQA,                     /* ADC0 序列 A 中断号 */
    INUM_ADC0_OVR,                      /* ADC0 overrun 中断号 */
    __lpc82x_adc0_int_plfm_init,        /* 平台初始化函数 */
    __lpc82x_adc0_int_plfm_deinit,      /* 平台解初始化函数 */
};
```

如需修改，只需要将该值修改为实际的值即可。

2. ADC 工作模式

由于 ADC 具有两种工作方式，一种是中断方式，代码体积小，一种为 DMA 方式，代码体积大。它们分别对应于两个不同的配置文件 {HWCONFIG}**am_hwconf_lpc82x_adc0_dma.c** 与 {HWCONFIG}**am_hwconf_lpc82x_adc0_int.c** 。用户如果为了提高 ADC 数据的传输速率，可以考虑使 ADC 工作在 DMA 模式，即 ADC 采样值通过 DMA 传输。此时，可以通过调用不同工作方式的实例初始化即可使用相应的模式。如 ADC 工作在 DMA 方式，则调用 {HWCONFIG}**am_hwconf_lpc82x_adc0_dma.c** 这个配置文件的 **am_lpc82x_adc0_dma_inst_init** 这个实例初始化函数即可。

注解： 一般情况下，建议使用 DMA 模式。若为了不占用 DMA 资源且在 ADC 采样率较低的情况下，才切换到中断模式，因为中断模式在每次数据采样完毕后，都需要在中断服务函数中读取数据，效率低，若中断模式下采样速率过高，则很有可能由于来不及读取数据导致 ADC 采样值溢出。

3. ADC 通道引脚配置

LPC824 中，ADC0 最高可支持 12 个通道，各通道对应的引脚详见 表 4.5 。

表 4.5: ADC 通道引脚及对应的配置宏

ADC 通道	引脚号	配置功能宏
0	PIO0_7	PIO0_7_ADC_0 | PIO0_7_INACTIVE
1	PIO0_6	PIO0_6_ADC_1 | PIO0_6_INACTIVE
2	PIO0_14	PIO0_14_ADC_2 | PIO0_14_INACTIVE
3	PIO0_23	PIO0_23_ADC_3 | PIO0_23_INACTIVE
4	PIO0_22	PIO0_22_ADC_4 | PIO0_22_INACTIVE
5	PIO0_21	PIO0_21_ADC_5 | PIO0_21_INACTIVE
6	PIO0_20	PIO0_20_ADC_6 | PIO0_20_INACTIVE
7	PIO0_19	PIO0_19_ADC_7 | PIO0_19_INACTIVE
8	PIO0_18	PIO0_18_ADC_8 | PIO0_18_INACTIVE
9	PIO0_17	PIO0_17_ADC_9 | PIO0_17_INACTIVE
10	PIO0_13	PIO0_13_ADC_10 | PIO0_13_INACTIVE
11	PIO0_4	PIO0_4_ADC_11 | PIO0_4_INACTIVE

其中，配置功能宏用于使用 am_gpio_pin_cfg() 函数配置一个引脚时对应的功能宏。

注 解： am_gpio_pin_cfg() 是 GPIO 标准接口函数，详见 {SDK}**ametal\common\interface\am_gpio.h** 文件。

如需使用一个通道，需要先配置一个与该通道对应的引脚。引脚的配置在平台初始化中完成。默认平台初始化函数详见 列表 4.29 。

列表 4.29: ADC0 平台初始化函数

```
1 /**
2  * \brief ADC0 平台初始化
3  */
```

```
am_local void __lpc82x_adc0_int_plfm_init (void)
{
    amhw_lpc82x_syscon_powerup(AMHW_LPC82X_SYSCON_PD_ADC0);
    amhw_lpc82x_clk_periph_enable(AMHW_LPC82X_CLK_ADC0);

    /* 配置通道，引脚配置为消极模式 INACTIVE */
    am_gpio_pin_cfg(PIO0_7,  PIO0_7_ADC_0   | PIO0_7_INACTIVE);
//    am_gpio_pin_cfg(PIO0_6,  PIO0_6_ADC_1   | PIO0_6_INACTIVE);
//    am_gpio_pin_cfg(PIO0_14, PIO0_14_ADC_2  | PIO0_14_INACTIVE);
//    am_gpio_pin_cfg(PIO0_23, PIO0_23_ADC_3  | PIO0_23_INACTIVE);
//    am_gpio_pin_cfg(PIO0_22, PIO0_22_ADC_4  | PIO0_22_INACTIVE);
//    am_gpio_pin_cfg(PIO0_21, PIO0_21_ADC_5  | PIO0_21_INACTIVE);
//    am_gpio_pin_cfg(PIO0_20, PIO0_20_ADC_6  | PIO0_20_INACTIVE);
    am_gpio_pin_cfg(PIO0_19, PIO0_19_ADC_7  | PIO0_19_INACTIVE);
//    am_gpio_pin_cfg(PIO0_18, PIO0_18_ADC_8  | PIO0_18_INACTIVE);
//    am_gpio_pin_cfg(PIO0_17, PIO0_17_ADC_9  | PIO0_17_INACTIVE);
//    am_gpio_pin_cfg(PIO0_13, PIO0_13_ADC_10 | PIO0_13_INACTIVE);
//    am_gpio_pin_cfg(PIO0_4,  PIO0_4_ADC_11  | PIO0_4_INACTIVE);

    /* ADC 自动矫正 */
    amhw_lpc82x_adc_calibrate(AMHW_LPC82X_ADC0,
                              amhw_lpc82x_clk_system_clkrate_get());
}
```

为了方便用户配置，默认情况下，列出了通道 0～11 的引脚配置，当用户需要使用一个通道时，只需要简单的将该通道引脚配置前的注释去掉即可。

注解： 通常仅将使用到的通道对应的引脚配置为模拟引脚功能，如果不使用，应注释掉对应引脚的配置语句，以便其它地方正常使用引脚的 IO 功能。将引脚配置为模拟通道引脚时，一定要加上 PIO*_*_INACTIVE 标志宏，以禁能该引脚的上拉电阻和下拉电阻，否则，采样值可能极不准确。

4.2.2 CLK

LPC82x 系统时钟最高可达 30MHz，由于 AM824-Core 没有焊接晶振，因此，默认情况下，使用内部 IRC 振荡器（12MHz），并将 PLL 时钟倍频至 60MHz，再 2 分频作为系统时钟。相关的时钟频率值均可通过设备信息配置。

时钟配置的核心是完成主时钟（main clock）的配置，系统时钟（system clock）仅为主时钟的一个分频，该分频值由设备信息中的系统时钟分频参数决定。详见 列表 4.30 。

列表 4.30: 时钟默认设备信息——系统时钟分频数

```
/** \brief CLK 设备信息 */
am_local am_const am_lpc82x_clk_devinfo_t __g_lpc82x_clk_devinfo = {

    ......

    /*
     * PLL 控制寄存器 MSEL，使 FCLKOUT 在范围 100MHz 内
     * FCLKOUT = FCLKIN * (MSEL + 1) = 12MHz * 5 = 60 MHz
     */
    4,

    /*
     * PLL 控制寄存器 PSEL，使 FCCO 在范围 156MHz - 320MHz
```

```
     * FCCO = FCLKOUT * 2 * 2^(PSEL) = 60MHz * 2 * 2 = 240MHz
     */
    1,

    ......

    /*
     * 系统时钟分频数，可填 1- 255 之间的数值
     * system_clk = main_clk / div = 60MHz / 2 = 30 MHz
     */
    2,

    ......
};
```

可见，由于默认配置中，主时钟为 60MHz，因此将系统时钟分频设置为 2，从而得到 30MHz 的系统时钟。

下面，我们重点介绍主时钟的配置，一般情况下，会选择 PLL 输出作为主时钟源，因此先讲述使用 PLL 的配置。由于用户可能有特殊需求，最后，也会介绍不使用 PLL 时的配置。

1. 使用 PLL

使用 PLL 时，首先应确定 PLL 的时钟源，由 {HWCONFIG}**\am_hwconf_lpc82x_clk.c** 文件的 **__LPC82X_CLK_PLL_SRC** 宏指定 PLL 的输入时钟源，详见 列表 4.31 。

列表 4.31: PLL 时钟源配置

```
/** \brief PLL 时钟源为 IRC */
#define __LPC82X_CLK_PLL_SRC  AMHW_LPC82X_CLK_PLLIN_SRC_IRC
```

可选时钟源见 表 4.6 ,相关内容在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_clk.h** 文件中定义。

表 4.6: PLL 可选时钟源

PLL 时钟源宏值	含义
AMHW_LPC82X_CLK_PLLIN_SRC_IRC	使用内部 IRC，12MHz
AMHW_LPC82X_CLK_PLLIN_SRC_SYSOSC	使用外部晶振，具体频率与晶振相关，常用 12MHz
AMHW_LPC82X_CLK_PLLIN_SRC_CLKIN	外部时钟输入（通过 PIO0_1 输入时钟）

默认情况下，PLL 时钟源使用内部 IRC，可以修改 **__LPC82X_CLK_PLL_SRC** 宏值将时钟源修改为其它可用时钟源，如外部焊接了晶振，为了提高时钟精度，可以使用外部晶振，此时，只需要将 **__LPC82X_CLK_PLL_SRC** 宏值修改为 **AMHW_LPC82X_CLK_PLLIN_SRC_SYSOSC** 即可，详见 列表 4.32 。

列表 4.32: PLL 时钟源修改

```
/** \brief 修改 PLL 时钟源为外部晶振 */
#define __LPC82X_CLK_PLL_SRC  AMHW_LPC82X_CLK_PLLIN_SRC_SYSOSC
```

注解： 由于外部晶振占用了 PIO0_8 和 PIO0_9 两个引脚，因此，使用了外部晶振后，这两个引脚不能再用作其它用途。同理，若使用外部时钟输入，因其占用了 PIO0_1，因此该引脚也不能再用作其它用途。

至此，即可完成时钟源的修改。若还需修改 PLL 的输出频率，则需要修改 PLL 的倍频系数。默认 PLL 的倍数是 5，即将 12MHz 频率倍频为 60MHz。

PLL 倍频数由设备信息的第二个参数 MSEL 确定：

$$\mathrm{PLL_{out}} = \mathrm{PLL_{in}} \times (MSEL + 1)$$

例如，在默认的配置信息中，MSEL 为 4，即 5 倍频，输出频率即为 12MHz 的 5 倍，即 PLL 输出频率为 60MHz。

设备信息的第三个参数为 PLL 控制器的 PSEL 值，该值主要是为了确保 PLL 的 FCCO 输出频率在有效范围内。PSEL 的值必须确保：

$$156 \le \mathrm{PLL_{out}} \times 2 \times 2^{PSEL} \le 320$$

例如，在默认的配置信息中，PSEL 为 1，则：

$$\mathrm{PLL_{out}} \times 2 \times 2^{PSEL} = 60MHz \times 2 \times 2 = 240MHz$$

该值处于 156～320MHz 之间，满足要求。

注解： 可能读者会有疑问，虽然配置 PLL 频率并不复杂，但是也还是有一定量的计算工作，为什么不直接在设备信息中填一个频率，由驱动自行计算确定各个 PLL 控制参数的值呢？这是由于，时钟部分一般修改极少，一个应用程序，一般也就首次使用时配置一下即可，若由驱动自动计算完成设置，则每次系统启动时都需要花费一定的时间去计算应该设置的控制值，效率低。

---

2. 不使用 PLL

由于默认是使用 PLL 输出作为主时钟源，若不使用 PLL 输出作为时钟源，则需要重新指定主时钟源。主时钟源的选择由设备信息的第四个参数确定。详见 列表 4.33 。

列表 4.33: 时钟默认设备信息——主时钟源配置

```
/** \brief CLK 设备信息 */
am_local am_const am_lpc82x_clk_devinfo_t __g_lpc82x_clk_devinfo = {

    ......

    /*
     * 主时钟源选择 main_clk = FCLKOUT = 60MHz
     * - 内部 IRC: AMHW_LPC82X_CLK_MAIN_SRC_IRC
     * - PLL 输入: AMHW_LPC82X_CLK_MAIN_SRC_PLLIN
     * - 看门狗振荡器: AMHW_LPC82X_CLK_MAIN_SRC_WDTOSC
     * - PLL 输出: AMHW_LPC82X_CLK_MAIN_SRC_PLLOUT
     */
    AMHW_LPC82X_CLK_MAIN_SRC_PLLOUT,

    ......
};
```

主时钟源可选的参数见 表 4.7 。需要设置为其它时钟源时，将 列表 4.33 中的

AMHW_LPC82X_CLK_MAIN_SRC_PLLOUT 修改为 表 4.7 中相应的宏值即可。

表 4.7: 可选主时钟源

主时钟源宏值	含义
AMHW_LPC82X_CLK_MAIN_SRC_IRC	直接使用内部 IRC 作为主时钟
AMHW_LPC82X_CLK_MAIN_SRC_PLLIN	使用 PLL 输入时钟作为主时钟
AMHW_LPC82X_CLK_MAIN_SRC_WDTOSC	使用看门狗振荡器作为时钟源
AMHW_LPC82X_CLK_MAIN_SRC_PLLOUT	使用 PLL 输出作为时钟源

注解： 表中的相关内容在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_clk.h** 文件中定义。

可见，除使用 PLL 输出作为时钟源外，还可以有三种设置。

使用内部 IRC 振荡器，该模式直接使用，无需再额外设置其它参数。

使用 PLL 输入，此时，应该正确配置 PLL 的输入时钟源，即正确设置宏 **__LPC82X_CLK_PLL_SRC** 的值，详见 列表 4.31 和 表 4.6 。

若使用看门狗振荡器作为主时钟源，则必须配置看看门狗振荡器的时钟频率和分频系数，以得到期望的时钟频率。详见 列表 4.34 。

列表 4.34: 时钟默认设备信息——看门狗振荡器配置

```
/** \brief CLK 设备信息 */
am_local am_const am_lpc82x_clk_devinfo_t __g_lpc82x_clk_devinfo = {

    ......

    /*
     * WDTOSC 频率分频系数，可填 2 - 64 之间的偶数
     * 若主时钟源选择看门狗振荡器时，需要配置 WDTOSC 频率分频系数和频率选择
     * 这里为默认值 0.6MHz，64 分频，时钟频率 9.375KHz
     */
    64,

    /*
     * WDTOSC 频率选择
     * wdtosc_freq = 600000UL Hz
     */
    AMHW_LPC82X_CLK_WDTOSC_RATE_0_6MHZ,

    ......
};
```

默认的设备信息中，分频系数为 64，WDTOSC 的频率为 0.6MHz，则最终主时钟的频率即为：

$$Freq_{Mainclk} = Freq_{WDTOSC} \div div = 0.6MHz \div 64 = 9.375KHz$$

注意，分频系数只能是 2～64 的偶数，WDTOSC 的输出频率并不能随意设置，所有可选的频率已在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_clk.h** 文件中使用宏定义出来了，详见 表 4.8 。

表 4.8: WDTOSC 可选输出频率

WDTOSC 可选输出频率设置宏	对应频率
AMHW_LPC82X_CLK_WDTOSC_RATE_0_6MHZ	0.6MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_1_05MHZ	1.05MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_1_4MHZ	1.4MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_1_75MHZ	1.75MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_2_1MHZ	2.1MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_2_4MHZ	2.4MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_2_7MHZ	2.7MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_3_0MHZ	3.0MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_3_25MHZ	3.25MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_3_5MHZ	3.5MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_3_75MHZ	3.75MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_4_0MHZ	4.0MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_4_2MHZ	4.2MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_4_4MHZ	4.4MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_4_6MHZ	4.6MHz

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 可支持 18 路通道，该值与硬件相关，定义在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_dma.h** 文件中。详见 列表 4.35 。

列表 4.35: DMA 支持的通道数

```
1 /** \brief DMA 通道数量 */
2 #define AMHW_LPC82X_DMA_CHAN_CNT     18
```

由于使用每个通道时，都需要耗费一定的内存空间，为此，实际使用到的通道数目可由用户根据实际情况配置。该值由 {HWCONFIG}**\am_hwconf_lpc82x_dma.c** 文件中的 **__DMA_CHAN_USE_COUNT** 宏确定。默认使用所有通道数。详见 列表 4.36 。

列表 4.36: DMA 实际使用通道数配置

```
1 /**
2  * \brief 使用的 DMA 通道数量
3  *
4  * 默认使用所有的 DMA 通道，用户可以根据实际使用通道数，更改此值，减少 DMA 内存的占用
5  *
6  * \note 如果需要使用 DMA，则此值至少应该为 1，否则 DMA 初始化不成功
7  */
8 #define __DMA_CHAN_USE_COUNT    AMHW_LPC82X_DMA_CHAN_CNT
```

如需修改，只需将该宏对应的值修改为实际使用的通道数目即可。如将该值修改为 5，

详见 列表 4.37 。

列表 4.37: DMA 配置示例

```
/** \brief 使用的 DMA 通道数 */
#define __DMA_CHAN_USE_COUNT    5
```

4.2.5 GPIO

LPC824M201JHI33 共计 29 个 IO 口，编号从 PIO0_0 ~ PIO0_28，所有编号已经使用宏在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\lpc82x_pin.h** 文件中定义。所有引脚均可用作触发模式（即 GPIO 中断），但最多只能将 8 个 IO 口用作触发模式。若用户实际使用的触发模式引脚数小于 8，可以修改 {HWCONFIG}**\am_hwconf_lpc82x_gpio.c** 配置文件中 **__GPIO_PINT_USE_COUNT** 的宏值，将其修改为实际使用到的引脚触发数目。例如：实际只将 5 个引脚用作了触发模式，则可以将该值修改为 5，详见 列表 4.38 。

列表 4.38: GPIO 配置

```
/** \brief 使用的中断通道数量 */
#define  __GPIO_PINT_USE_COUNT   5
```

4.2.6 I²C

平台共计有 4 个 I²C 总线接口，定义为 I²C0、I²C1、I²C2、I²C3。以 I²C0 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_lpc82x_i2c0.c** 中的宏 **BUS_SPEED_I2C0** 配置，详见 列表 4.39 。默认为标准 I²C 速率，即 400KHz，如需修改为其它频率，直接修改该宏对应的值即可。

列表 4.39: I²C0 速率配置

```
/** \brief I2C0 总线速率参数定义 */
#define __BUS_SPEED_I2C0     (400000)
```

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。

超时时间的设置在 {HWCONFIG}**\am_hwconf_lpc82x_i2c0.c** 设备信息中设置，详见 列表 4.40 。

列表 4.40: I²C0 设备信息——超时时间配置

```
/** \brief I2C0 设备信息 */
am_local am_const am_lpc_i2c_devinfo_t __g_lpc82x_i2c0_devinfo = {
    __BUS_SPEED_I2C0,           /* I2C0 总线速率 */
    LPC82X_I2C0_BASE,           /* I2C0 寄存器块基址 */
    INUM_I2C0,                  /* I2C0 中断号 */
    CLK_I2C0,                   /* I2C0 时钟号 */
    10,                         /* 超时时间 */
    __lpc82x_i2c0_plfm_init,    /* 平台初始化函数 */
    __lpc82x_i2c0_plfm_deinit   /* 平台解初始化函数 */
```

```
};
```

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 I²C0 为例，详见 列表 4.41 。

列表 4.41: I²C0 平台初始化函数——引脚配置

```
/**
 * \brief I2C0 平台初始化函数
 */
am_local void __lpc82x_i2c0_plfm_init (void)
{
    am_gpio_pin_cfg(PIO0_10, PIO0_10_I2C0_SCL);
    am_gpio_pin_cfg(PIO0_11, PIO0_11_I2C0_SDA);

    amhw_lpc82x_clk_periph_enable(AMHW_LPC82X_CLK_I2C0);
    amhw_lpc82x_syscon_periph_reset(AMHW_LPC82X_RESET_I2C0);
}
```

可见，程序中，将 PIO0_10 配置为 I²C0 的时钟线，PIO0_11 配置为 I²C0 的数据线。

特别注意，PIO0_10 和 PIO0_11 为特殊的 I²C 引脚，属于标准的开漏引脚，且内部无任何上拉电阻。因此，在使用时，必须外接上拉电阻。LPC82x 仅这两个标准的开漏引脚，固定提供给 I²C0 使用，因此，I²C0 的引脚必须为 PIO0_10 和 PIO0_11，不能配置为其它引脚。

对于其它 I²C（包括 I²C1、I²C2、I²C3），由于 LPC82x 内部具有 SWM 开关矩阵，因此，这些 I²C 相关的引脚可以任意配置。

各 I²C 可使用的引脚详见 表 4.9 。

表 4.9: I²C 引脚选择

I²C 总线接口	SCL 引脚	SDA 引脚
I²C0	PIO0_10	PIO0_11
I²C1 I²C2 I²C3	任意引脚	任意引脚

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。各个 I²C 引脚的功能宏详见 表 4.10 。

表 4.10: I²C 功能宏选择

I²C 总线接口	SCL 功能宏	SDA 功能宏
I²C0	PIO0_10_I2C0_SCL	PIO0_11_I2C0_SDA
I²C1	PIO_FUNC_I2C1_SCL	PIO_FUNC_I2C1_SDA
I²C2	PIO_FUNC_I2C2_SCL	PIO_FUNC_I2C2_SDA
I²C3	PIO_FUNC_I2C3_SCL	PIO_FUNC_I2C3_SDA

例如，使用 I²C1 时，假定使用 PIO0_16 为 I²C1 的 SCL，PIO0_18 为 I²C1 的 SDA。则可以如 列表 4.42 所示配置 I²C1 的相关引脚。

列表 4.42: I²C1 引脚配置范例

```
am_gpio_pin_cfg(PIO0_16, PIO_FUNC_I2C1_SCL | PIO0_16_OPEN_DRAIN);
am_gpio_pin_cfg(PIO0_18, PIO_FUNC_I2C1_SDA | PIO0_18_OPEN_DRAIN);
```

注解： 如果引脚使用了软件开漏模式，则 I²C 总线必须外接上拉电阻才可以正常通信。

4.2.7 MRT

MRT 为多速率定时器，最高可以同时支持 4 路定时。默认使用所有通道，用户可以在 {HWCONFIG}**am_hwconf_lpc82x_mrt.c** 文件中的设备信息中将该值配置为实际使用到的通道数目。详见 列表 4.43 。

列表 4.43: MRT 设备信息结构体定义

```
/** \brief MRT 设备信息 */
am_local am_const am_lpc_mrt_devinfo_t __g_lpc82x_mrt_devinfo = {
    LPC82X_MRT_BASE,              /* MRT 寄存器块基址 */
    INUM_MRT,                     /* MRT 中断号 */
    CLK_MRT,                      /* MRT 时钟号 */
    4,                            /* 使用的通道数 */
    __lpc82x_mrt_plfm_init,       /* 平台初始化函数 */
    __lpc82x_mrt_plfm_deinit,     /* 平台解初始化函数 */
};
```

如实际只使用两个通道，则可以将 列表 4.43 中的 4 修改为 2，修改后见 列表 4.44 。

列表 4.44: MRT 配置范例

```
/** \brief MRT 设备信息 */
am_local am_const am_lpc_mrt_devinfo_t __g_lpc82x_mrt_devinfo = {
    LPC82X_MRT_BASE,              /* MRT 寄存器块基址 */
    INUM_MRT,                     /* MRT 中断号 */
    CLK_MRT,                      /* MRT 时钟号 */
    2,                            /* 使用的通道数 */
    __lpc82x_mrt_plfm_init,       /* 平台初始化函数 */
    __lpc82x_mrt_plfm_deinit,     /* 平台解初始化函数 */
};
```

4.2.8 SCT

SCT 是状态可配置定时器，使用起来非常灵活，可以实现复杂的时序。SCT 可以提供一些 AMetal 抽象好的标准接口服务，如 PWM、CAP、Timing。这些抽象出来的功能都比较单一，无法体现 SCT 更多特殊的功能，因此，SCT 也可以不用于标准服务，直接使用驱动层提供的相关接口操作 SCT 的特殊功能，如状态、事件等。由于 SCT 的特殊功能和抽象出来的标准服务功能并不能同时使用，因此，在初始化时，就需要确定将 SCT 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 SCT 部分对应了 4 套设备配置文件，使用 SCT 的何种功能，就使用对应的配置文件。

1. 不使用标准服务，使用 SCT 驱动提供的特殊功能函数

该模式下，对应的配置文件为 {HWCONFIG}**am_hwconf_lpc82x_sct0.c** ，SCT0 最多可支持 8 个事件. 同理，由于使用每个事件时，都需要耗费一定的内存空间。因此，实际使用到的事件数目可由用户根据实际情况配置。该值由 {HWCONFIG}**am_hwconf_lpc82x_sct0.c**

文件中的 **__SCT_EVT_ISRINFO_COUNT** 宏确定。默认使用所有通道数。详见 列表 4.45 。

列表 4.45: SCT0 实际使用事件数配置

```
#define __SCT_EVT_ISRINFO_COUNT    8
```

如需修改，只需将该宏对应的值修改为实际使用的事件数目即可。如将该值修改为 5，详见 列表 4.46 。

列表 4.46: SCT0 实际使用事件数配置示例

```
#define __SCT_EVT_ISRINFO_COUNT    5
```

同理，SCT 当中最多可以支持 6 个输出通道，2 个 DMA 请求通道，用户可以根据需要自行配置这两个 **__SCT_OUTPUT_COUNT** 与 **__SCT_DMA_REQ_COUNT** 这两个宏值。

2. 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_lpc82x_sct0_cap.c** ，SCT0 最多可以支持 4 路捕获通道，实际使用到的通道数目可以在配置文件中的设备信息中修改。详见 列表 4.47 。

列表 4.47: SCT0 CAP 设备信息

```
/** \brief SCT0 CAP 设备信息 */
am_local am_const am_lpc_sct_cap_devinfo_t __g_lpc82x_cap_devinfo = {
    LPC82X_SCT0_BASE,                   /* SCT0 寄存器块基地址 */
    INUM_SCT0,                          /* SCT0 中断号 */
    CLK_SCT,                            /* SCT0 时钟号 */
    4,                                  /* 4 个捕获通道 */
    &__g_sct0_cap_ioinfo_list[0],       /* 所有 PWM 引脚配置信息，用首地址参数传递 */
    __lpc82x_sct0_cap_plfm_init,        /* 平台初始化函数 */
    __lpc82x_sct0_cap_plfm_deinit,      /* 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_sct0_cap_ioinfo_list** 数组定义，详见 列表 4.48 。

列表 4.48: SCT0 CAP 各捕获通道相关引脚设置

```
/** \brief SCT0 用于 CAP 的引脚配置信息列表，CAP 的输入个数为 4 个 */
am_local am_lpc_sct_cap_ioinfo_t __g_sct0_cap_ioinfo_list[] = {
    {PIO0_25, PIO_FUNC_SCT_PIN0, PIO0_25_GPIO | PIO0_25_GPIO_INPUT}, /* 通道 0 */
    {PIO0_26, PIO_FUNC_SCT_PIN1, PIO0_26_GPIO | PIO0_26_GPIO_INPUT}, /* 通道 1 */
    {PIO0_27, PIO_FUNC_SCT_PIN2, PIO0_27_GPIO | PIO0_27_GPIO_INPUT}, /* 通道 2 */
    {PIO0_28, PIO_FUNC_SCT_PIN3, PIO0_28_GPIO | PIO0_28_GPIO_INPUT}, /* 通道 3 */
};
```

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_lpc82x_sct_cap_ioinfo_t** ，该类型在对应驱动头文件 {SDK}**\ametal\soc\nxp\lpc\common\drivers\include\am_lpc_sct_cap.h** 中定义，详见 列表 4.49 。

列表 4.49: SCT0 CAP 通道引脚信息结构体类型

```
typedef struct am_lpc_sct_cap_ioinfo {
    uint32_t gpio;          /**< \brief GPIO 引脚 */
    uint32_t func;          /**< \brief GPIO 功能 */
    uint32_t dfunc;         /**< \brief 取消功能后默认功能 */
} am_lpc82x_sct_cap_ioinfo_t;
```

可见，每个捕获 IO 的信息包括了三点信息：

- 引脚号：LPC824 支持的引脚号为：PIO0_0 ~ PIO0_28。
- GPIO 功能：即当被用作捕获通道时，应该配置为的 GPIO 功能宏，对应的宏值应该是将引脚配置为 SCT 输入的功能宏，如 **PIO_FUNC_SCT_PIN0** 。
- 取消捕获功能后的默认功能：由于每个捕获通道可以被使能或禁能，当该捕获通道被禁能后，相对应的 GPIO 资源可以被释放（即暂时不用作捕获功能，可以被用作它用），驱动将自动设置 GPIO 的功能为 dfunc 指定的功能。

默认设置中，将通道 0 ~ 3 配置为了 PIO0_25 ~ PIO028，如有需要，用户可以修改为其它任意期望的引脚。

3. 使用标准 PWM 服务

在该模式下，对应的配置文件为 {HWCONFIG}**am_hwconf_lpc82x_sct0_pwm.c** ，SCT0 最多可以支持 6 路 PWM 输出，实际使用到的通道数目可以在配置文件中的设备信息中修改。详见 列表 4.50 。

列表 4.50: SCT0 PWM 设备信息

```
/** \brief SCT0 PWM 设备信息 */
am_local am_const am_lpc_sct_pwm_devinfo_t __g_lpc82x_pwm_devinfo = {
    LPC82X_SCT0_BASE,                   /* SCT0 寄存器块基地址 */
    CLK_SCT,                            /* SCT0 时钟号 */
    6,                                  /* 6 个 PWM 输出通道 */
    &__g_sct0_pwm_ioinfo_list[0],       /* 所有 PWM 引脚配置信息，用首地址参数传递 */
    __lpc82x_sct0_pwm_plfm_init,        /* 平台初始化函数 */
    __lpc82x_sct0_pwm_plfm_deinit,      /* 平台解初始化函数 */
};
```

使用 PWM 功能时，每个 PWM 输出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_sct0_pwm_ioinfo_list** 数组定义，详见 列表 4.51 。

列表 4.51: SCT0 PWM 各输出通道相关引脚设置

```
/** \brief SCT0 用于 PWM 的引脚配置信息列表，PWM 的输出个数为 6 个 */
am_local am_lpc_sct_pwm_ioinfo_t __g_sct0_pwm_ioinfo_list[] = {
    {PIO0_23, PIO_FUNC_SCT_OUT0, PIO0_23_GPIO | PIO0_23_GPIO_INPUT}, /* 通道 0␣
↪*/
    {PIO0_24, PIO_FUNC_SCT_OUT1, PIO0_24_GPIO | PIO0_24_GPIO_INPUT}, /* 通道 1␣
↪*/
    {PIO0_25, PIO_FUNC_SCT_OUT2, PIO0_25_GPIO | PIO0_25_GPIO_INPUT}, /* 通道 2␣
↪*/
    {PIO0_26, PIO_FUNC_SCT_OUT3, PIO0_26_GPIO | PIO0_26_GPIO_INPUT}, /* 通道 3␣
↪*/
    {PIO0_27, PIO_FUNC_SCT_OUT4, PIO0_27_GPIO | PIO0_27_GPIO_INPUT}, /* 通道 4␣
↪*/
    {PIO0_15, PIO_FUNC_SCT_OUT5, PIO0_15_GPIO | PIO0_15_GPIO_INPUT}, /* 通道 5␣
↪*/
};
```

每个数组元素对应了一个 PWM 输出通道，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_lpc82x_sct_pwm_ioinfo_t**，该类型在对应驱动头文件 {SDK}**\ametal\soc\nxp\lpc\common\drivers\include\am_lpc_sct_pwm.h** 文件中定义，详见 列表 4.52 。

列表 4.52: SCT0 PWM 通道引脚信息结构体类型

```
typedef struct am_lpc_sct_pwm_ioinfo {
    uint32_t gpio;              /**< \brief PWM GPIO */
    uint32_t func;              /**< \brief PWM 默认功能 */
    uint32_t dfunc;             /**< \brief PWM 禁止状态的功能 */
} am_lpc_sct_pwm_ioinfo_t;
```

可见，每个捕获 IO 的信息包括了三点信息：

- 引脚号：LPC824 支持的引脚号为：PIO0_0 ~ PIO0_28。
- GPIO 功能：即当被用作 PWM 通道时，应该配置为的 GPIO 功能宏，对应的宏值应该是将引脚配置为 SCT 输出的功能宏，如 PIO_FUNC_SCT_OUT0。
- 取消 PWM 功能后的默认功能：由于每个 PWM 输出通道可以被使能或禁能，当该 PWM 通道被禁能后，相对应的 GPIO 资源可以被释放（即暂时不用作 PWM 功能，可以被用作它用），驱动将自动设置 GPIO 的功能为 dfunc 指定的功能。

默认设置中，将通道 0 ~ 4 配置为了 PIO0_23 ~ PIO0 ~ 27，通道 5 配置为了 PIO0_15。如有需要，用户可以修改为其它任意期望的引脚。

4. 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_lpc82x_sct0_timing.c** ，SCT0 非常灵活，可以被用作两个 16 位定时器，也可以被用作 1 个 32 位定时器。可以根据实际需要进行配置。配置文件中，默认设置为将 SCT0 用做两个 16 位定时器，详见 列表 4.53 。

列表 4.53: SCT0 Timing 设备配置信息

```
/** \brief SCT TIMING 设备信息 */
am_local am_const am_lpc_sct_timing_devinfo_t __g_lpc82x_sct0_timing_devinfo = {
    LPC82X_SCT0_BASE,                   /* SCT0 寄存器块基地址 */
    INUM_SCT0,                          /* SCT0 中断号 */
    CLK_SCT,                            /* SCT0 时钟号 */
    AM_LPC_SCT_TIMING_2_16BIT,          /* SCT 用于 2 个 16 位定时器 */
    __lpc82x_sct0_timing_plfm_init,     /* 平台初始化函数 */
    __lpc82x_sct0_timing_plfm_deinit,   /* 平台解初始化函数 */
};
```

两种模式对应的宏在对应的驱动头文件 {SDK}**\ametal\soc\nxp\lpc\common\drivers\include\am_lpc_sct_timi** 文件中定义。详见 列表 4.54 。

列表 4.54: 定时器的工作模式

```
/** \brief SCT 运行在 1 个 32 位定时器模式，仅能提供 1 个定时器通道 */
#define AM_LPC82X_SCT_TIMING_1_32BIT      1

/** \brief SCT 运行在 2 个 16 位定时器模式，能提供 2 个定时器通道 */
#define AM_LPC82X_SCT_TIMING_2_16BIT      2
```

例如，需要将定时器的工作模式设置为将 SCT0 用做 1 个 32 位定时器，详见 列表 4.55 。

列表 4.55: SCT0 Timing 配置范例

```
/** \brief SCT TIMING 设备信息 */
am_local am_const am_lpc_sct_timing_devinfo_t __g_lpc82x_sct0_timing_devinfo = {
    LPC82X_SCT0_BASE,                   /* SCT0 寄存器块基地址 */
    INUM_SCT0,                          /* SCT0 中断号 */
    CLK_SCT,                            /* SCT0 时钟号 */
    AM_LPC_SCT_TIMING_1_32BIT,          /* SCT0 用于 1 个 32-bit 定时器 */
    __lpc82x_sct0_timing_plfm_init,     /* 平台初始化函数 */
    __lpc82x_sct0_timing_plfm_deinit,   /* 平台解初始化函数 */
};
```

4.2.9 SPI

平台有 2 个 SPI 总线接口，定义为 SPI0、SPI1。一般地，只需要配置 SPI 工作模式和相关引脚即可。下面以 SPI0 为例，讲述其配置内容。

1. SPI 工作模式

由于 SPI 具有两种工作方式，一种是中断方式，代码体积小，一种为 DMA 方式，代码体积大。它们分别对应于两个不同的配置文件 {HWCONFIG}**\am_hwconf_lpc82x_spi0_dma.c** 与 {HWCONFIG}**\am_hwconf_lpc82x_spi0_int.c** 。用户如果为了提高 SPI 读写数据的传输速率，可以考虑使 SPI 工作在 DMA 模式。此时，可以通过调用不同工作方式的实例初始化即可使用相应的模式。如 SPI0 工作在 DMA 方式，则调用 {HWCONFIG}**\am_hwconf_lpc82x_spi0_dma.c** 这个配置文件里面的 **am_lpc82x_spi0_dma_inst_init** 这个实例初始化函数即可。

注解: 一般情况下，建议使用 DMA 模式。若为了不占用 DMA 资源且对 SPI 读写速度没有要求的情况下，才切换到中断模式。

---

2. SPI 相关引脚设置

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\interface\am_spi.h** ）时，片选引脚可以作为参数任意设置。

相关引脚的设置在设备配置文件 {HWCONFIG}**\am_hwconf_lpc82x_spi0_xxx.c** 中的平台初始化函数中完成，详见 列表 4.56 。

列表 4.56: SPI0 平台初始化函数

```
/**
 * \brief SPI0 平台初始化
 */
am_local void __lpc82x_spi0_int_plfm_init (void)
{
    am_gpio_pin_cfg(PIO0_15, PIO_FUNC_SPI0_SCK);
    am_gpio_pin_cfg(PIO0_12, PIO_FUNC_SPI0_MOSI);
    am_gpio_pin_cfg(PIO0_13, PIO_FUNC_SPI0_MISO);

    /* CS_Pin 由用户调用 STD 函数时自行传入，此处不需配置 */

    amhw_lpc82x_clk_periph_enable(AMHW_LPC82X_CLK_SPI0);
    amhw_lpc82x_syscon_periph_reset(AMHW_LPC82X_RESET_SPI0);
}
```

由于 LPC82x 具有开关矩阵，SPI0 的相关引脚可以任意配置。各引脚对应 GPIO 功能宏详见 表 4.11 。

表 4.11: SPI 引脚功能宏定义

SPI 总线接口	SPI0	SPI1
SCK	PIO_FUNC_SPI0_SCK	PIO_FUNC_SPI1_SCK
MOSI	PIO_FUNC_SPI0_MOSI	PIO_FUNC_SPI1_MOSI
MISO	PIO_FUNC_SPI0_MISO	PIO_FUNC_SPI1_MISO

4.2.10 USART

平台有 3 个 USART 接口，定义为 USART0、USART1、USART2。在 SDK 提供的驱动中，仅只实现了 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 USART0 为例，讲解其配置内容。

注解： 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

USART0 相关的引脚在配置文件 {HWCONFIG}**am_hwconf_lpc82x_usart0.c** 文件中配置，详见 列表 4.57 。

列表 4.57: USATRT0 平台初始化函数

```
/**
 * \brief USART0 平台初始化
 */
am_local void __lpc82x_usart0_plfm_init (void)
{

    /* 设置串口基础时钟 */
    amhw_lpc82x_clk_usart_baseclkrate_set(__LPC82X_UASART_BASE_RATE);

    amhw_lpc82x_clk_periph_enable(AMHW_LPC82X_CLK_UART0);
    amhw_lpc82x_syscon_periph_reset(AMHW_LPC82X_RESET_UART0);

    am_gpio_pin_cfg(PIO0_4, PIO_FUNC_U0_TXD);
    am_gpio_pin_cfg(PIO0_0, PIO_FUNC_U0_RXD);
}
```

程序中，将 PIO0_4 配置为了串口 0 的发送引脚，PIO0_0 配置为了串口 0 的接收引脚。串口相关的引脚均可利用开关矩阵任意配置。各个串口的发送和接收引脚对应的 GPIO 功能宏详见 表 4.12 。

表 4.12: USART 引脚功能宏定义

USART 接口	RXD 功能宏	TXD 功能宏
USART0	PIO_FUNC_U0_RXD	PIO_FUNC_U0_TXD
USART1	PIO_FUNC_U1_RXD	PIO_FUNC_U1_TXD
USART2	PIO_FUNC_U2_RXD	PIO_FUNC_U2_TXD

2. 串口的输入频率配置

由于默认主时钟为 60MHz，串口无法从该值通过整数分频得到一些常见的波特率，如 115200。因此，LPC82x 在主时钟和串口时钟输入之间，增加了一个小数分频器（FRG），以便得到期望的波特率整数倍的时钟频率，输入至串口，在串口内部再使用整数分频器得到期望的波特率。

例如，常见地，我们将小数分频器的输出频率设置为 11059200（11.0592MHz），基于该频率，串口可以自由使用 2400、4800、9600、19200、115200、230400 等常见的波特率。但是，小数分频器只有一个，是所有串口共用的，即所有串口的输入频率是一样的。这样，对串口波特率的值就会有要求，所有串口的波特率均只能通过小数分频器的输出频率再整数分频得到。

如将小数分频器的输出频率设置为 11059200，就不能得到如 500Kbps 等不常见的波特率。若要使用一些不常见的波特率，就需要修改小数分频器的输出频率，特别注意，这样的修改对所有串口都会有影响。

默认情况下，该值设置为最常见的 11059200，由设备配置文件中的 **__LPC82X_UASART_BASE_RATE** 宏指定。详见 列表 4.58 。

列表 4.58: 设置串口的输入频率

```
1 #define  __LPC82X_UASART_BASE_RATE  11059200
```

如需使用一些特殊的波特率，应先将该值设置为期望波特率的整数倍，以便得到精确的波特率。例如，需要使用 500Kbps 的波特率，则可以将该值修改为 10000000（10MHz），详见 列表 4.59 。

列表 4.59: 修改串口的输入频率

```
1 #define  __LPC82X_UASART_BASE_RATE  10000000
```

4.2.11 WKT

WKT 作为一种唤醒定时器，可以提供一种标准定时器服务，需要配置的仅有时钟源配置。可选时钟源有 3 个，详见 表 4.13 。

表 4.13: WKT 可选时钟源

PLL 时钟源宏值	含义
AMHW_LPC_WKT_IRC_CLOCK	使用内部 IRC，12MHz
AMHW_LPC_WKT_LOW_POWER_CLOCK	低功耗时钟，标称 10kHz
AMHW_LPC_WKT_EXT_CLOCK	外部引脚（PIO0.28/WKTCLKIN）输入时钟

注解： 这些时钟源在 {SDK}**\ametal\soc\nxp\lpc\common\hw\include\amhw_lpc_wkt.h** 文件中定义。

具体配置在设备配置文件 {HWCONFIG}**\am_hwconf_lpc82x_wkt.c** 中的设备信息中配置。默认选择内部 IRC 作为时钟源，详见 列表 4.60 。

列表 4.60: WKT 设备信息结构体

```
/** \brief WKT 设备信息 */
am_local am_const am_lpc_wkt_devinfo_t __g_lpc82x_wkt_devinfo = {
    LPC82X_WKT_BASE,                /* WKT 寄存器块基地址 */
    INUM_WKT,                       /* WKT 中断号 */
    1,                              /* 支持 1 个通道 */
    AMHW_LPC_WKT_LOW_POWER_CLOCK,   /* 选择 IRC 时钟 */
    0,                              /* 使用外部时钟时的频率设置 */
    __lpc82x_wkt_plfm_clk_init,     /* 平台时钟初始化函数 */
    __lpc82x_wkt_plfm_init,         /* 平台初始化函数 */
    __lpc82x_wkt_plfm_deinit        /* 平台解初始化函数 */
};
```

若需要修改为其它时钟源，只需要将设备信息中 **AMHW_LPC_WKT_IRC_CLOCK** 修改为期望的时钟源即可。

值得注意的是，当选择 **AMHW_LPC_WKT_EXT_CLOCK** 作为时钟源时，必须由第 5 个参数指明输入时钟的频率（其它时钟源时，该参数无效）。例如，选择外部时钟引脚输入作为时钟源，其频率为 10KHz，则可以修改设备信息如 列表 4.61 所示。

列表 4.61: 修改 WKT 的时钟源为外部引脚输入

```
/** \brief WKT 设备信息 */
am_local am_const am_lpc_wkt_devinfo_t __g_lpc82x_wkt_devinfo = {
    LPC82X_WKT_BASE,                /* WKT 寄存器块基地址 */
    INUM_WKT,                       /* WKT 中断号 */
    1,                              /* 支持 1 个通道 */
    AMHW_LPC_WKT_EXT_CLOCK,         /* 选择外部时钟 */
    10000,                          /* 使用外部时钟时的频率设置 */
    __lpc82x_wkt_plfm_clk_init,     /* 平台时钟初始化函数 */
    __lpc82x_wkt_plfm_init,         /* 平台初始化函数 */
    __lpc82x_wkt_plfm_deinit        /* 平台解初始化函数 */
};
```

4.2.12 WWDT

WWDT 对应的配置文件为 {HWCONFIG}**am_hwconf_lpc82x_wwdt.c** ，需要配置的是 WDTOSC 的输出频率和分频，以便为 WWDT 提供时钟。相关时钟的配置在设备配置文件的平台初始化函数中完成，详见 列表 4.62 。

列表 4.62: WWDT 平台初始化函数

```
/**
 * \brief WWDT 平台初始化
 */
am_local void __lpc82x_wwdt_plfm_init (void)
{
    amhw_lpc82x_clk_periph_enable(AMHW_LPC82X_CLK_WWDT);

    /* 设置 WDT 时钟，0.6MHz，64 分频，时钟频率 9.375KHz */
    amhw_lpc82x_clk_wdtoscc_cfg(AMHW_LPC82X_CLK_WDTOSC_RATE_0_6MHZ, 64);

    /* 使能 WDTOSC */
    amhw_lpc82x_syscon_powerup(AMHW_LPC82X_SYSCON_PD_WDT_OSC);
}
```

注解： 在前面讲述 CLK 配置（详见 4.2.2 ）时知道，可以选择 WDTOSC 作为系统的主时钟，

当选择 WDTOSC 作为系统主时钟时，已经配置好了 WDTOSC，该处则无需重复配置。

WDTOSC 时钟的配置由 **amhw_lpc82x_clk_wdtoscc_cfg()** 函数完成，该函数在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_clk.h** 文件中定义，函数原型详见 列表 4.63 。

列表 4.63: amhw_lpc82x_clk_wdtoscc_cfg() 函数原型

```
1 void amhw_lpc82x_clk_wdtoscc_cfg (uint8_t  rate,  uint8_t  div);
```

rate 参数指明 WDTOSC 的输出频率，div 为输出频率选择一个分频系数。默认的设备信息中，分频系数为 64，WDTOSC 的频率为 0.6MHz，则最终 WWDT 的输入频率即为：

$$WWDT_{freq} = 0.6MHz \div 64 = 9.375KHz$$

注意，分频系数只能是 2 ~ 64 的偶数，WDTOSC 的输出频率并不能随意设置，所有可选的频率已在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_clk.h** 文件中定义，详见 表 4.14 。

表 4.14: WDTOSC 可选输出频率

主时钟源宏值	对应频率
AMHW_LPC82X_CLK_WDTOSC_RATE_0_6MHZ	0.6MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_1_05MHZ	1.05MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_1_4MHZ	1.4MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_1_75MHZ	1.75MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_2_1MHZ	2.1MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_2_4MHZ	2.4MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_2_7MHZ	2.7MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_3_0MHZ	3.0MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_3_25MHZ	3.25MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_3_5MHZ	3.5MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_3_75MHZ	3.75MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_4_0MHZ	4.0MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_4_2MHZ	4.2MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_4_4MHZ	4.4MHz
AMHW_LPC82X_CLK_WDTOSC_RATE_4_6MHZ	4.6MHz

4.2.13 NVIC

在 LPC82x 中，可以产生中断的外设都有对应的中断号，每个中断号都可以对应一个中断服务的入口，LPC82x 可支持 32 路中断，该值定义在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\lpc82x_inum.h** 文件中。详见 列表 4.64 。

列表 4.64: 中断号相关定义

```
1 #define INUM_SPI0          0   /**< \brief SPI0 中断 */
2 #define INUM_SPI1          1   /**< \brief SPI1 中断 */
```

```
#define INUM_USART0         3   /**< \brief USART0 中断 */
#define INUM_USART1         4   /**< \brief USART1 中断 */
#define INUM_USART2         5   /**< \brief USART2 中断 */
#define INUM_I2C1           7   /**< \brief I2C1 中断 */
#define INUM_I2C0           8   /**< \brief I2C0 中断 */
#define INUM_SCT0           9   /**< \brief SCT 中断 */
#define INUM_MRT            10  /**< \brief Multi-rate 定时器中断 */
#define INUM_ACMP           11  /**< \brief 模拟比较器中断 */
#define INUM_WDT            12  /**< \brief 看门狗中断 */
#define INUM_BOD            13  /**< \brief BOD 中断 */
#define INUM_FLASH          14  /**< \brief FLASH 中断 */
#define INUM_WKT            15  /**< \brief WKT 中断 */
#define INUM_ADC0_SEQA      16  /**< \brief ADC0 序列 A 中断 */
#define INUM_ADC0_SEQB      17  /**< \brief ADC0 序列 B 中断 */
#define INUM_ADC0_THCMP     18  /**< \brief ADC0 阈值比较和错误中断 */
#define INUM_ADC0_OVR       19  /**< \brief ADC0 overrun 中断 */
#define INUM_DMA            20  /**< \brief DMA 中断 */
#define INUM_I2C2           21  /**< \brief I2C2 中断 */
#define INUM_I2C3           22  /**< \brief I2C3 中断 */
#define INUM_PIN_INT0       24  /**< \brief 引脚中断 0 */
#define INUM_PIN_INT1       25  /**< \brief 引脚中断 1 */
#define INUM_PIN_INT2       26  /**< \brief 引脚中断 2 */
#define INUM_PIN_INT3       27  /**< \brief 引脚中断 3 */
#define INUM_PIN_INT4       28  /**< \brief 引脚中断 4 */
#define INUM_PIN_INT5       29  /**< \brief 引脚中断 5 */
#define INUM_PIN_INT6       30  /**< \brief 引脚中断 6 */
#define INUM_PIN_INT7       31  /**< \brief 引脚中断 7 */

/**
 * \brief 总中断数为: (INUM_PIN_INT7 - INUM_SPI0 + 1),
*/
#define INUM_INTERNAL_COUNT     (INUM_PIN_INT7 - INUM_SPI0 + 1)

/**
 * \brief 最大中断号为: INUM_PIN_INT7
 */
#define INUM_INTERNAL_MAX        INUM_PIN_INT7

/** \brief 最小中断号 */
#define INUM_INTERNAL_MIN        INUM_SPI0
```

可以看到，总中断数目的最终值为 32。同理，由于使用每个中断号对应的中断时，都需要耗费一定的内存空间，为此，实际使用到的中断数目可由用户根据实际情况配置。该值由 {HWCONFIG}**am_hwconf_arm_nvic.c** 文件中的 **__ISRINFO_COUNT** 宏确定。默认使用所有中断。详见 列表 4.65 。

列表 4.65: 实际使用中断数配置

```
/** \brief 中断信息数量 */
#define __ISRINFO_COUNT    INUM_INTERNAL_COUNT
```

如需修改，只需将该宏对应的值修改为实际使用的中断数目即可。如将该值修改为 20，详见 列表 4.66 。

列表 4.66: NVIC 中断数配置示例

```
/** \brief 实际使用的中断数目 */
#define __ISRINFO_COUNT    20
```

同时，用户还可以在设备信息中配置 LPC824 这款芯片的内核类型为

AM_ARM_NVIC_CORE_M0PLUS ，这款芯片使用的优先级位数有 2 位（一般来说，M3 与 M4 内核的芯片，NVIC 最多可有 8 位用于设置优先级，在这 8 位里面可以分成多少位设置组中断，多少位分成设置子中断。而 M0 与 M0+，NVIC 仅有 2 位用于设置优先级）。NVIC 的设备信息详见 列表 4.67 。

列表 4.67: NVIC 设备信息

```
/** \brief 中断设备信息 */
am_local am_const am_arm_nvic_devinfo_t __g_arm_nvic_devinfo =
{
    {
        INUM_INTERNAL_MIN,    /* 起始中断号 */
        INUM_INTERNAL_MAX     /* 末尾中断号 */
    },

    AM_ARM_NVIC_CORE_M0PLUS, /* Cortex-M0+ 内核 */

    2,                        /* 仅有子优先级，且子优先级有 2 位 */
    0,                        /* 组中断 */

    INUM_INTERNAL_COUNT,      /* 总中断数量 */
    __nvic_isr_map,           /* 中断信息映射 */
    __ISRINFO_COUNT,          /* 中断信息数量 */
    __nvic_isr_infor,         /* 中断信息映射内存 */

    NULL,                     /* 无需平台初始化函数 */
    NULL                      /* 无需平台解初始化函数 */
};
```

4.2.14 Systick

Systick 是系统滴答定时器，一般 Cortex-M 系列内核的 MCU 均有此定时器。滴答定时器可配置的仅有时钟源，可选时钟源有系统时钟或系统时钟的 2 分频，默认选择系统时钟。相关配置在配置文件 {HWCONFIG}**\am_hwconf_arm_systick.c** 中的设备信息中指定。详见 列表 4.68 。

列表 4.68: Systick 设备信息

```
/**
 * \brief Systick 设备信息
 *
 * \note 时钟源分为系统时钟 (#AMHW_ARM_SYSTICK_CONFIG_CLKSRC_SYSTEM) 和系统
 *       时钟的 1/2 (#AMHW_ARM_SYSTICK_CONFIG_CLKSRC_SYSTEM_HALF)
 */
am_local am_const am_arm_systick_devinfo_t __g_arm_systick_devinfo = {
    LPC82X_SYSTICK_BASE,                      /* 指向 Systick 寄存器块指针 */
    CLK_SYSTEM,                               /* Systick 时钟号，来源于主时钟 */
    AMHW_ARM_SYSTICK_CONFIG_CLKSRC_SYSTEM,    /* Systick 时钟选择系统时钟 */
    NULL,                                     /* 无需平台初始化函数 */
    NULL                                      /* 无需平台解初始化函数 */
};
```

注解： 由于 Systick 无需平台初始化函数和平台解初始化函数，因此在 Systick 的设备信息结构体中，直接将相关的函数指针赋值为 NULL。

Systick 的可选时钟源，已在 {SDK}**\ametal\arch\arm\common\hw\include\amhw_arm_systick.h**

文件中定义，详见 列表 4.69 。

列表 4.69: Systick 可选时钟源宏定义

```
1 /** \brief Systick 时钟源为系统时钟 */
2 #define AMHW_ARM_SYSTICK_CONFIG_CLKSRC_SYSTEM           (1 << 2)
3
4 /** \brief Systick 时钟源的为系统时钟的 1/2 */
5 #define AMHW_ARM_SYSTICK_CONFIG_CLKSRC_SYSTEM_HALF      (0 << 2)
```

若需使用系统时钟二分频作为 Systick 的时钟源，只需将设备信息中的时钟源选择由 **AMHW_ARM_SYSTICK_CONFIG_CLKSRC_SYSTEM** 修改为 **AMHW_ARM_SYSTICK_CONFIG_CLKSRC_SYSTEM_HALF** 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

4.3.1.1 初始化

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

所有外设的实例初始化函数均在 {PROJECT}**user_config\am_lpc82x_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_lpc82x_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 4.15 。

表 4.15: 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(中断方式)	am_adc_handle_t am_lpc82x_adc0_int_inst_init (void);
2	ADC(DMA 方式)	am_adc_handle_t am_lpc82x_adc0_dma_inmst_init (void);
3	CLK	int am _lpc82x_clk_inst_init (void);
4	CRC	am_crc_handle_t am_lpc82x_crc_inst_init (void);
5	DMA	int am_lpc82x_dma_inst_init (void);
6	GPIO	int am_lpc82x_gpio_inst_init (void);
7	I^2C0	am_i2c_handle_t am_lpc82x_i2c0_inst_init (void);
8	I^2C1	am_i2c_handle_t am_lpc82x_i2c1_inst_init (void);
9	I^2C2	am_i2c_handle_t am_lpc82x_i2c2_inst_init (void);
10	I^2C3	am_i2c_handle_t am_lpc82x_i2c3_inst_init (void);
11	MRT	am_timer_handle_t am_lpc82x_mrt_inst_init (void);
12	SCT0	am_lpc82x_sct_handle_t am_lpc82x_sct0_inst_init (void);
13	SCT0_CAP	am_cap_handle_t am_lpc82x_sct0_cap_inst_init (void);
14	SCT0_PWM	am_pwm_handle_t am_lpc82x_sct0_pwm_inst_init (void);
15	SCT0_Timing	am_timer_handle_t am_lpc82x_sct0_timing_inst_init (void);
16	SPI0(中断方式)	am_spi_handle_t am_lpc82x_spi0_int_inst_init (void);
17	SPI0(DMA 方式)	am_spi_handle_t am_lpc82x_spi0_dma_inst_init (void);
18	SPI1(中断方式)	am_spi_handle_t am_lpc82x_spi1_int_inst_init (void);
19	SPI1(DMA 方式)	am_spi_handle_t am_lpc82x_spi1_dma_inst_init (void);
20	USART0	am_uart_handle_t am_lpc82x_usart0_inst_init (void);
21	USART1	am_uart_handle_t am_lpc82x_usart1_inst_init (void);
22	USART2	am_uart_handle_t am_lpc82x_usart2_inst_init (void);
23	WKT	am_timer_handle_t am_lpc82x_wkt_inst_init (void);
24	WWDT	am_wdt_handle_t am_lpc82x_wwdt_inst_init (void);
25	NVIC 中断	int am_arm_nvic_inst_init (void);
26	Systick	am_timer_handle_t am_arm_systick_inst_init (void);

4.3.1.2 操作外设

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

- int 型；
- 标准服务 handle 类型（**am_*_handle_t**，类型由标准接口层定义），如 **am_adc_handle_t**；
- 驱动自定义 handle 类型（**am_lpc82x_*_handle_t**，类型由驱动头文件自定义），如 **am_lpc82x_sct_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 4.16 。

表 4.16: 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_lpc82x_clk_inst_init (void);
2	DMA	int am_lpc82x_dma_inst_init (void);
3	GPIO	int am_lpc82x_gpio_inst_init (void);
4	NVIC 中断	int am_arm_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}**\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 列表 4.70 。

列表 4.70: GPIO 标准接口使用范例

```
1 am_gpio_pin_cfg(PIO0_20, 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 AM824-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\nxp\lpc\lpc82x\drivers\include\am_lpc82x_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 4.17 。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 4.17: 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(中断方式)	am_adc_handle_t am_lpc82x_adc0_int_inst_init (void);
2	ADC(DMA 方式)	am_adc_handle_t am_lpc82x_adc0_dma_inmst_init (void);
3	CRC	am_crc_handle_t am_lpc82x_crc_inst_init (void);
4	I^2C0	am_i2c_handle_t am_lpc82x_i2c0_inst_init (void);
5	I^2C1	am_i2c_handle_t am_lpc82x_i2c1_inst_init (void);
6	I^2C2	am_i2c_handle_t am_lpc82x_i2c2_inst_init (void);
7	I^2C3	am_i2c_handle_t am_lpc82x_i2c3_inst_init (void);
8	MRT	am_timer_handle_t am_lpc82x_mrt_inst_init (void);
9	SCT0_CAP	am_cap_handle_t am_lpc82x_sct0_cap_inst_init (void);
10	SCT0_PWM	am_pwm_handle_t am_lpc82x_sct0_pwm_inst_init (void);
11	SCT0_Timing	am_timer_handle_t am_lpc82x_sct0_timing_inst_init (void);
12	SPI0(中断方式)	am_spi_handle_t am_lpc82x_spi0_int_inst_init (void);
13	SPI0(DMA 方式)	am_spi_handle_t am_lpc82x_spi0_dma_inst_init (void);
14	SPI1(中断方式)	am_spi_handle_t am_lpc82x_spi1_int_inst_init (void);
15	SPI1(DMA 方式)	am_spi_handle_t am_lpc82x_spi1_dma_inst_init (void);
16	USART0	am_uart_handle_t am_lpc82x_usart0_inst_init (void);
17	USART1	am_uart_handle_t am_lpc82x_usart1_inst_init (void);
18	USART2	am_uart_handle_t am_lpc82x_usart2_inst_init (void);
19	WKT	am_timer_handle_t am_lpc82x_wkt_inst_init (void);
20	WWDT	am_wdt_handle_t am_lpc82x_wwdt_inst_init (void);
21	Systick	am_timer_handle_t am_arm_systick_inst_init (void);

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

例如，ADC 设备的实例初始化函数的返回值类型为 **am_adc_handle_t** ，为了方便后续使用，可以定义一个变量保存下该返回值，后续就可以使用该 handle 完成电压的采集了。详见 列表 4.71 。

列表 4.71: ADC 简单操作示例

```
#include "ametal.h"
#include "am_vdebug.h"
#include "am_board.h"
#include "am_lpc82x.h"
#include "am_lpc82x_adc.h"
#include "am_lpc82x_inst_init.h"

/**
 * \brief ADC 硬件触发转换，DMA 传输转换结果，通过标准接口实现
 *
 * \return 无
 */
int am_main (void)
{
    int             i;
    int             chan = 0;     /* 通道 0 */
```

```
    uint32_t        adc_mv[5];    /* 采样电压 */
    am_adc_handle_t adc0_handle;  /* ADC 标准服务操作句柄 */

    am_lpc82x_dma_inst_init ();      /* DMA 实例初始化 */
    adc0_handle = am_lpc82x_adc0_hw_trg_inst_init (); /* ADC 实例初始化并获取句柄值 */

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

    while (1) {

        /* 获取 ADC 采集电压，采样完成才返回 */
        am_adc_read_mv(adc0_handle, chan, adc_mv, 5);
        for (i = 1; i < 5; i++) {
            adc_mv[0] += adc_mv[i];
        }
        adc_mv[0] /= 5;
        am_kprintf("Vol: %d mv\r\n", adc_mv[0]);
    }
}
```

3. 返回值为组件定义服务句柄

这类外设功能较为特殊，AMetal 未对其进行标准服务的抽象，这时，使用该外设对应的实例初始化函数返回的即是驱动自定义的服务句柄，相关外设详见 表 4.18 。可见，仅 SCT0 外设提供了一个该类型的实例初始化函数。

表 4.18: 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	SCT0	am_lpc82x_sct_handle_t am_lpc82x_sct0_inst_init (void);

注意： SCT0 可以用作标准的 PWM、CAP 和定时服务，也可以用作驱动自定义的特殊功能服务。但同一时间，只能使用一种服务。

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。

SCT0 相关驱动函数在 {SDK}**\ametal\soc\common\drivers\include\am_lpc_sct.h** 文件中声明。相关接口及使用方法可以参考该文件或 {SDK}**\documents\《AMetal AM824-Core API 参考手册.chm》** 。handle 的使用示例详见 列表 4.72 。

列表 4.72: SCT0 handle 简单操作示例

```
#include “ametal.h
#include "am_lpc82x_inst_init.h"

am_lpc82x_sct_handle_t g_lpc82x_sct0_handle;

int am_main (void)
{
    g_lpc82x_sct0_handle = am_lpc82x_sct0_inst_init ();

    /* 设置 SCT 状态为 0 */
     am_lpc_sct_state_set(g_lpc82x_sct0_handle, 0);
```

```
12
13      ......
14
15      while(1);
16      }
17  }
```

注解： 更加详细使用示例可参考 {SDK}**ametal\examples\sct\am824_core\demo_lpc82x_drv_sct_timing_pwm.c** 文件中的源码。

4.3.1.3 解初始化

外设使用完毕后，应该调用相应设备配置文件提供的设备实例解初始化函数，以释放相关资源。所有外设的实例解初始化函数均在 {PROJECT}**user_config\am_lpc82x_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_lpc82x_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 4.19 。

表 4.19: 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	ADC(中断方式)	void am_lpc82x_adc0_int_inst_deinit (am_adc_handle_t handle);
2	ADC(DMA 方式)	void am_lpc82x_adc0_dma_inst_deinit (am_adc_handle_t handle);
3	CRC	void am_lpc82x_crc_inst_deinit (am_crc_handle_t handle);
4	DMA	void am_lpc82x_dma_inst_deinit (void);
5	GPIO	void am_lpc82x_gpio_inst_deinit (void);
6	I^2C0	void am_lpc82x_i2c0_inst_deinit (am_i2c_handle_t handle);
7	I^2C1	void am_lpc82x_i2c1_inst_deinit (am_i2c_handle_t handle);
8	I^2C2	void am_lpc82x_i2c2_inst_deinit (am_i2c_handle_t handle);
9	I^2C3	void am_lpc82x_i2c3_inst_deinit (am_i2c_handle_t handle);
10	MRT	void am_lpc82x_mrt_inst_deinit (am_timer_handle_t handle);
11	SCT0	void am_lpc82x_sct0_inst_deinit (am_lpc82x_sct_handle_t handle);
12	SCT0_CAP	void am_lpc82x_sct0_cap_inst_deinit (am_cap_handle_t handle);
13	SCT0_PWM	void am_lpc82x_sct0_pwm_inst_deinit (am_pwm_handle_t handle);
14	SCT0_Timing	void am_lpc82x_sct0_timing_inst_deinit (am_timer_handle_t handle);
15	SPI0(中断方式)	void am_lpc82x_spi0_int_inst_deinit (am_spi_handle_t handle);
16	SPI0(DMA 方式)	void am_lpc82x_spi0_dma_inst_deinit (am_spi_handle_t handle);
17	SPI1(中断方式)	void am_lpc82x_spi1_int_inst_deinit (am_spi_handle_t handle);
18	SPI1(DMA 方式)	void am_lpc82x_spi1_dma_inst_deinit (am_spi_handle_t handle);
19	USART0	void am_lpc82x_usart0_inst_deinit (am_uart_handle_t handle);
20	USART1	void am_lpc82x_usart1_inst_deinit (am_uart_handle_t handle);
21	USART2	void am_lpc82x_usart2_inst_deinit (am_uart_handle_t handle);
22	WKT	void am_lpc82x_wkt_inst_deinit (am_timer_handle_t handle);
23	WWDT	void am_lpc82x_wwdt_inst_deinit (am_wdt_handle_t handle);
24	NVIC 中断	void am_arm_nvic_inst_deinit (void);
25	Systick	void am_arm_systick_inst_deinit (am_timer_handle_t handle);

注意：时钟部分不能被解初始化。

外设实例解初始化函数相对简单，所有实例解初始化函数均无返回值。

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

4.3.2 直接使用硬件层函数

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

通常，HW 层的接口函数都是以外设寄存器结构体指针为参数（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 SPI 为例,所有硬件层函数均在 {SDK}**\ametal\soc\nxp\common\hw\include\amhw_lpc_spi.h** 文件中声明（一些简单的内联函数直接在该文件中定义）。简单列举几个函数，详见 列表 4.73 。

列表 4.73: SPI 硬件层操作函数

```
/**
 * \brief 使能 SPI
 * \param[in] p_hw_spi : 指向 SPI 寄存器块的指针
 * \return 无
 */
am_static_inline
void amhw_lpc_spi_enable (amhw_lpc82x_spi_t *p_hw_spi)
{
    p_hw_spi->cfg |= AMHW_LPC_SPI_CFG_ENABLE;
}

/**
 * \brief 禁能 SPI
 * \param[in] p_hw_spi : 指向 SPI 寄存器块的指针
 * \return 无
 */
am_static_inline
void amhw_lpc_spi_disable (amhw_lpc82x_spi_t *p_hw_spi)
{
    p_hw_spi->cfg &= ~AMHW_LPC_SPI_CFG_ENABLE;
}
```

其它一些函数读者可自行打开 {SDK}** \ametal\soc\nxp\common\hw\include\amhw_lpc_spi.h** 文件查看。这些函数均是以 amhw_lpc_spi_t * 类型作为第一个参数。**amhw_lpc_spi_t** 类型在 {SDK}**\ametal\soc\nxp\common\hw\include\amhw_lpc_spi.h** 文件中定义，用于定义出 SPI 外设的各个寄存器。详见 列表 4.74 。

列表 4.74: SPI 寄存器结构体定义

```
/**
 * \brief SPI 寄存器块结构体
 */
typedef struct amhw_lpc82x_spi {
  __IO uint32_t     cfg;          /**< \brief SPI 配置寄存器 */
  __IO uint32_t     dly;          /**< \brief SPI 延迟寄存器 */
  __IO uint32_t     stat;         /**< \brief SPI 状态寄存器 */
  __IO uint32_t     intenset;     /**< \brief SPI 中断使能读取和设置 */
  __O  uint32_t     intenclr;     /**< \brief SPI 中断使能清零 */
  __I  uint32_t     rxdat;        /**< \brief SPI 接收数据 */
  __IO uint32_t     txdatctl;     /**< \brief SPI 控制数据传输 */
  __IO uint32_t     txdat;        /**< \brief SPI 传输数据 */
  __IO uint32_t     txctl;        /**< \brief SPI 传输控制 */
  __IO uint32_t     div;          /**< \brief SPI 时钟分频器 */
  __I  uint32_t     intstat;      /**< \brief SPI 中断状态 */
} amhw_lpc82x_spi_t;
```

该类型的指针已经在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\hw\includeamhw_lpc82x_periph_map.h** 文件中定义，应用程序可以直接使用。详见 列表 4.75 。

列表 4.75: SPI 寄存器结构体指针定义

```
/** \brief 串行外设接口 (SPI0) 寄存器块指针 */
#define AMHW_LPC82X_SPI0        ((amhw_lpc_spi_t  *)LPC82X_SPI0_BASE)

/** \brief 串行外设接口 (SPI1) 寄存器块指针 */
#define AMHW_LPC82X_SPI1        ((amhw_lpc_spi_t  *)LPC82X_SPI1_BASE)
```

注解： 其中的 **AMHW_LPC82X_SPI0** 和 **AMHW_LPC82X_SPI1** 是 SPI0 和 SPI1 外设寄存器的基地址，在 {SDK}**\ametal\soc\nxp\lpc\lpc82x\lpc82x_regbase.h** 文件中定义，其他所有外设的基地址均在该文件中定义。

有了这两个 SPI 寄存器结构体指针宏后，就可以直接使用 SPI 硬件层的相关函数了。使用硬件层函数时，若传入参数为 **AMHW_LPC82X_SPI0** ，则表示操作的是 SPI0；若传入参数为 **AMHW_LPC82X_SPI1** ，则表示操作的是 SPI1。如 列表 4.76 和 列表 4.77 所示，分别用于使能 SPI0 和 SPI1。

列表 4.76: 使能 SPI0

```
amhw_lpc_spi_enable(AMHW_LPC82X_SPI0);
```

列表 4.77: 使能 SPI1

```
amhw_lpc_spi_enable(AMHW_LPC82X_SPI1);
```

特别地，可能想要操作的功能，硬件层也没有提供出相关接口，此时，可以基于各个外设指向寄存器结构体的指针，直接操作寄存器实现，例如，要使能 SPI0 和 SPI1，也可以直接直接设置寄存器的值，详见 列表 4.78 和 列表 4.79 。

列表 4.78: 直接设置寄存器的值使能 SPI0

```
/*
 * 设置 CFG 寄存器的 bit0 为 1，使能 SPI0
 */
AMHW_LPC82X_SPI0->cfg |= 0x01;
```

列表 4.79: 直接设置寄存器的值使能 SPI1

```
/*
 * 设置 CFG 寄存器的 bit0 为 1，使能 SPI1
 */
AMHW_LPC82X_SPI1->cfg |= 0x01;
```

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}**\ametal\soc\nxp\lpc\lpc82x\hw\include\amhw_lpc82x_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 4.20 。

表 4.20: 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	Systick	AMHW_LPC82X_SYSTICK
2	IOCON	AMHW_LPC82X_IOCON
3	GPIO	AMHW_LPC82X_GPIO
4	PINT	AMHW_LPC82X_PINT
5	SWM	AMHW_LPC82X_SWM
6	INMUX	AMHW_LPC82X_INMUX
7	DMA	AMHW_LPC82X_DMA
8	MRT	AMHW_LPC82X_MRT
9	SCT0	AMHW_LPC82X_SCT0
10	SYSCON	AMHW_LPC82X_SYSCON
11	SPI0	AMHW_LPC82X_SPI0
12	SPI1	AMHW_LPC82X_SPI1
13	I^2C0	AMHW_LPC82X_I2C0
14	I^2C1	AMHW_LPC82X_I2C1
15	I^2C2	AMHW_LPC82X_I2C2
16	I^2C3	AMHW_LPC82X_I2C3
17	USART0	AMHW_LPC82X_USART0
18	USART1	AMHW_LPC82X_USART1
19	USART2	AMHW_LPC82X_USART2
20	CRC	AMHW_LPC82X_CRC
21	WKT	AMHW_LPC82X_WKT
22	PMU	AMHW_LPC82X_PMU
23	FMC	AMHW_LPC82X_FMC
24	ADC0	AMHW_LPC82X_ADC0
25	ACMP	AMHW_LPC82X_ACMP
26	WWDT	AMHW_LPC82X_WWDT

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 5.21 所示。

表 5.21: 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 {SDK}**ametal\board\am824_core\am_board.c** 中自动完成初始化。以 LED 为例，初始化代码详见 列表 5.1 。

列表 5.1: LED 实例初始化函数调用

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

5.2 典型配置

5.2.1 LED 配置

AM824-Core 板上有两个 LED 灯，默认引脚分别为 PIO0_20 和 PIO0_21，使用时，需要使用跳线帽短接 AM824-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 列表 5.2 。

列表 5.2: LED 相关配置信息

```
/** \brief LED 引脚号 */
am_local am_const int __g_led_pins[] = {
    PIO0_20,    /* LED0 引脚 */
    PIO0_21     /* LED1 引脚 */
};

/** \brief LED 设备信息 */
am_local am_const am_led_gpio_info_t __g_led_gpio_devinfo = {
    {
        0,                                  /* LED 起始编号 */
        AM_NELEMENTS(__g_led_pins) - 1      /* LED 结束编号 */
    },
```

```
    __g_led_pins,
    AM_TRUE
};
```

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 列表 5.3 。

列表 5.3: LED 引脚配置信息类型定义

```
typedef struct am_led_gpio_info {

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

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

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

} am_led_gpio_info_t;
```

其中，serv_info 为 LED 的基础服务信息，包含 LED 的起始编号和介绍编号，p_pins 指向存放 LED 引脚的数组首地址，在本平台可选择的管脚在 lpc82x_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 PIO0_20 和 PIO0_21 ，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 PIO0_20 和 PIO0_21，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 SCT 的输出通道 1（SCT_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 5.22 。

表 5.22: 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 SCT 的 PWM 功能，若应用程序需要使用 SCT，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

AM824-Core 有两个板载按键 KEY/ RES 和 RST，默认引脚分别为 PIO0_1 和 PIO0_5，使用时，需要使用跳线帽短接 AM824-Core 板上的 J14 和 J8。其中 RST 默认为复位按键，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 列表 5.4 。

列表 5.4: KEY 相关配置信息

```
/** \brief 按键引脚号 */
am_local am_const int __g_key_pins[] = {
    PIO0_1    /* KEY/RES 按键引脚 */
};

/** \brief 按键编码 */
am_local am_const int __g_key_codes[] = {
    KEY_KP0    /* KEY/RES 按键编码 */
};

/** \brief 按键设备信息 */
am_local am_const am_key_gpio_info_t __g_key_gpio_devinfo = {
    __g_key_pins,                /* 按键引脚号 */
    __g_key_codes,               /* 各个按键对应的编码（上报） */
    AM_NELEMENTS(__g_key_pins), /* 按键数目 */
    AM_TRUE,                     /* 是否低电平激活（按下为低电平） */
    10                           /* 按键扫描时间间隔，一般为 10ms */
};
```

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 列表 5.5 。

列表 5.5: KEY 引脚配置信息类型定义

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

p_pins 指向存放 KEY 引脚的数组首地址，在本平台可选择的管脚在 lpc82x_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 PIO0_1 ，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 PIO0_1，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。为保证芯片能正常复位，复位按键 RST 对应的 PIO0_5 管脚不建议配置为普通的 GPIO 口。

5.2.4 调试串口配置

AM824-Core 具有 3 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 5.23 。

表 5.23: 调试串口相关配置

宏名	含义
__DEBUG_UART	串口号，0-UART0，1-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 MRT 定时器为其提供一个周期性的定时中断，默认使用 MRT 定时器的通道 0。其配置还需要使用 {PROJECT}**user_config\am_hwconf_usrcfg\am_hwconf_system_tick_softimer.c** 文件中的 **__SYSTEM_TICK_RATE** 宏来设置系统滴答的频率，默认为 1KHz。详细定义见 列表 5.6 。

列表 5.6: 系统滴答频率配置

```
1 /** \brief 设置系统滴答的频率，默认 1KHz */
2 #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 列表 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

AM824-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 I^2C 从机地址，详细定义见 列表 5.7 。

列表 5.7: LM75 地址配置

```
/** \brief LM75 设备信息 */
am_local am_const am_temp_lm75_info_t __g_temp_lm75_devinfo = {
    0x48    /* LM75 I2C 7 位地址  */
};
```

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 5.24 ，使用方法可以参考 4.3.1.2 。

表 5.24: 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、I^2C、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。

AM824-Core 板载 1 路带扩展的 MicroPort 接口，如 图 6.22 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 LPC824 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。

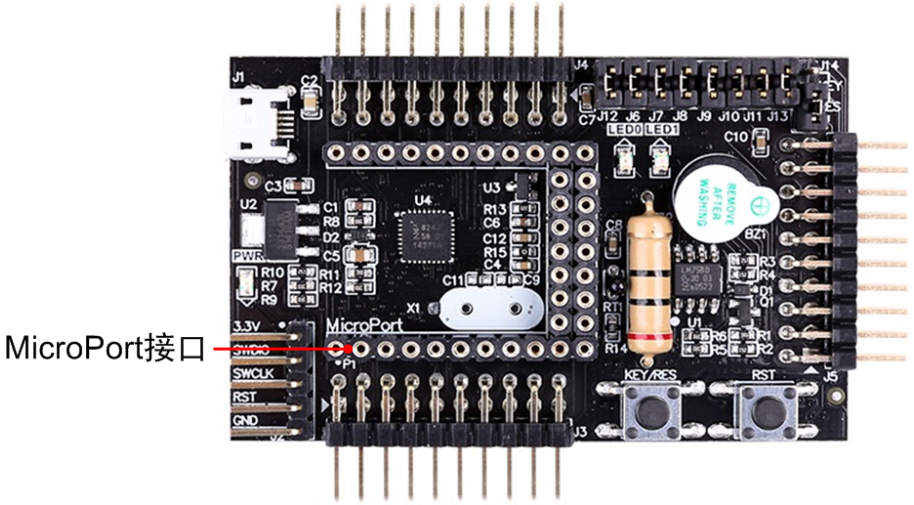

图 6.22: AM824-Core MicroPort

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-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 6.25 。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 6.25: 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 6.26 。使用方法可以参考 4.3.1.2 ，也可以参考 {SDK}**ametal\examples\am824_core\microport_board** 目录下的例程。

表 6.26: 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 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。

AM824-Core 开发板搭载了 1 路 MiniPort，接口标号 J4，如 图 7.23 所示。

注意：由于 LPC824 芯片 I/O 口有限，AM824-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。

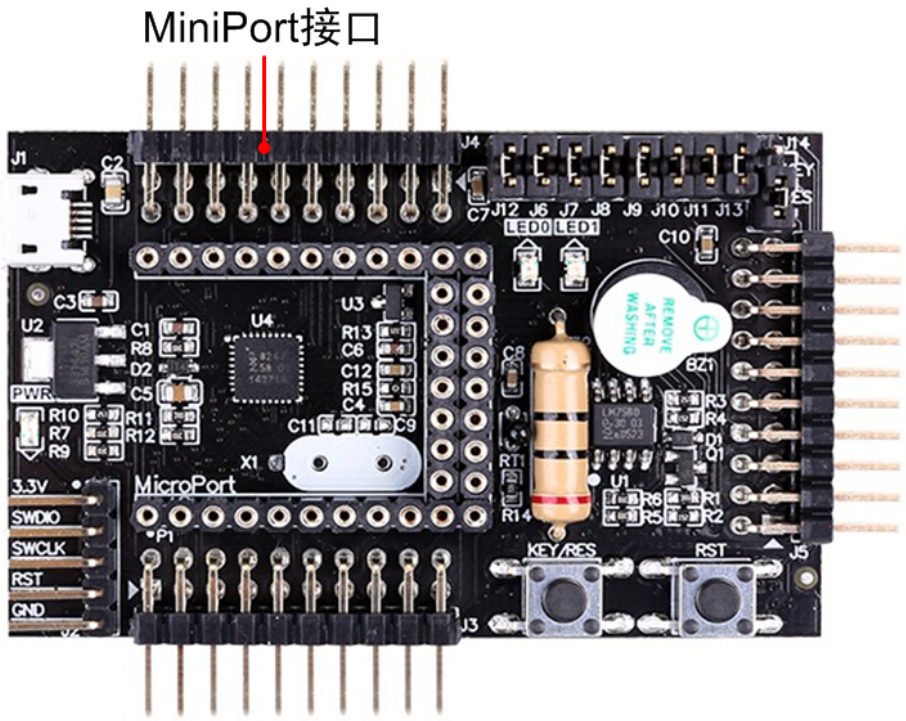

图 7.23: AM824-Core MiniPort

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 7.27 。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 7.27: 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 7.28 。使用方法可以参考 4.3.1.2 ，也可以参考 {SDK}**ametal\examples\am824_core\miniport_board** 目录下的例程。

表 7.28: 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 AM824-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

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

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

销售与服务网络

广州周立功单片机科技有限公司

地址：广州市天河区龙怡路 117 号银汇大厦 16 楼
邮编：510630
电话：020-38730916 38730917 38730976 38730977
网址：www.zlgmcu.com
传真：020-38730925

广州专卖店

地址：广州市天河区新赛格电子城 203-204 室
电话：020-87578634/87569917
传真：020-87578842

南京周立功

地址：南京市秦淮区汉中路 27 号友谊广场 17 层 F、G 区
电话：025-68123901/68123902/68123919
传真：025-68123900

北京周立功

地址：北京市海淀区紫金数码园 3 号楼（东华合创大厦）8 层 0802 室
电话：010-62635033/62635573/62635884
传真：010-82164433

重庆周立功

地址：重庆市渝北区龙溪街道新溉大道 18 号山顶国宾城 11 幢 4-14
电话：023-68796438/68796439/68797619
传真：023-68796439

杭州周立功

地址：杭州市西湖区紫荆花路 2 号杭州联合大厦 A 座 4 单元 508
电话：0571-89719484/89719499/89719498
传真：0571-89719494

成都周立功

地址：成都市一环路南二段 1 号数码科技大厦 403 室
电话：028-85439836/85432683/85437446
传真：028-68796439

深圳周立功（一部）

地址：深圳市福田区深南中路 2072 号电子大厦 1203 室
电话：0755-82941683/82907445
传真：0755-83793285

深圳周立功（二部）

地址：深圳市坪山区比亚迪路大万文化广场 A 座 1705
电话：0755-83781788/83782922
传真：0755-83793285

武汉周立功

地址：武汉市武昌区武珞路 282 号思特大厦 807 室
电话：027-87168497/87168297/87168397
传真：027-87163755

上海周立功

地址：上海市黄浦区北京东路 668 号科技京城东座 12E 室
电话：021-53083451/53083452/53083453
传真：021-53083491

周立功厦门办

地址：厦门市思明区厦禾路 855 号英才商厦 618 室
电话：18650195588

周立功苏州办

地址：江苏省苏州市广济南路 258 号（百脑汇科技中心 1301 室）
电话：0512-68266786 & 18616749830

周立功合肥办

地址：安徽省合肥市蜀山区黄山路 665 号汇峰大厦 1607
电话：13851513746

周立功宁波办

地址：浙江省宁波市高新区星海南路 16 号轿辰大厦 1003
电话：0574-87228513/87229313

周立功天津办

地址：天津市河东区十一经路与津塘公路交口鼎泰大厦 1004 室
电话：18622359231

周立功山东办

地址：山东省青岛市李沧区青山路 689 号宝龙公寓 3 号楼 311
电话：13810794370

周立功郑州办

地址：河南郑州市中原区百花路与建设路东南角锦绣华庭 A 座 1502 室
电话：17737307206

周立功沈阳办

地址：沈阳市浑南新区营盘西街 17 号万达广场 A4 座 2722 室
电话：18940293816

香港周立功

地址：香港新界沙田火炭禾香街 9-15 力坚工业大厦 13 层
电话：(852)26568073 26568077

周立功长沙办

地址：湖南省长沙市岳麓区奥克斯广场国际公寓 A 栋 2309 房
电话：0731-85161853