

嵌入式软件工程方法与实践丛书

AMetal适配指南

周立功 主编
AMetal团队 编著

AMetal

立功科技

目录

第 1 章 AMetal 简介	1
1.1.1 AMetal 的特点	1
1.1.2 AMetal 源码获取	1
1.1.3 AMetal 源码目录结构简介	3
1.1.4 AMetal 的快速使用	5
第 2 章 MCU 新型号适配概述	6
2.1 理论学习：AMetal 驱动框架解析	6
2.2 小试牛刀：让 MCU 跑起来	7
2.3 筑基打梁：通用外设驱动开发	7
2.4 查缺补漏：特殊驱动开发	8
2.5 共享发布：GIT 开源发布	8
第 3 章 AMetal 驱动框架解析	9
3.1 标准接口层	9
3.1.1 标准服务句柄 Handle	10
3.1.2 对功能进行抽象	10
3.1.3 基于抽象服务实现接口	11
3.2 驱动层	13
3.2.1 抽象服务的实现	13
3.2.2 实例初始化	15
3.2.3 实例解初始化	21
3.2.4 其它功能性函数	22
3.3 硬件层	22
3.3.1 使用硬件层接口	23
3.4 通用接口定义	25
3.4.1 接口定义	25
3.4.2 接口实现	26
3.5 设备的典型配置及使用	29
3.5.1 配置文件	30
3.5.2 典型配置	31
3.5.3 外设资源使用方法	35
3.6 中间件	39
3.7 基础组件	40
3.8 通用例程文件	40
3.8.1 板级例程	40
3.8.2 组件例程	41
3.8.3 硬件层例程	43
3.8.4 驱动层例程	44
第 4 章 让 MCU 跑起来	46
4.1 基础工程搭建	46
4.1.1 搭建基础工程	46
4.1.2 添加芯片基础文件	46
4.1.3 修改工程相关文件	48

4.2	让 MCU 跑起来	49
4.2.1	KEIL 工程配置修改	49
4.2.2	创建启动文件	52
4.2.3	Debug 测试验证	58
第 5 章	通用外设驱动开发——WDT	59
5.1	通用外设标准层了解	59
5.1.1	了解标准接口相关类型定义	59
5.1.2	了解标准接口函数	61
5.2	WDT 外设 HW 层开发	62
5.2.1	新建 HW 层文件	62
5.2.2	添加寄存器列表	65
5.2.3	编写寄存器相关操作函数	66
5.2.4	检查补充所需功能函数	70
5.2.5	添加指向寄存器块的指针	70
5.3	外设驱动层开发	71
5.3.1	新建驱动文件	72
5.3.2	编写结构性程序	73
5.3.3	驱动函数实现	77
5.4	外设用户配置文件开发	81
5.4.1	新建配置文件	81
5.4.2	定义设备常量于设备信息常量	82
5.4.3	添加平台初始化与平台解初始化函数	83
5.4.4	在头文件添加变量和函数声明	83
5.5	外设例程文件开发	84
5.5.1	新建例程文件	84
5.5.2	例程内容开发	85
第 6 章	通用外设驱动开发——CLK	88
6.1	标准层了解	88
6.1.1	了解相关类型定义	88
6.1.2	了解标准接口函数	88
6.2	HW 层开发	89
6.2.1	添加寄存器列表	89
6.2.2	编写寄存器的相关函数	90
6.2.3	添加指向寄存器块的指针	114
6.3	驱动层开发	114
6.3.1	编写结构性程序	115
6.3.2	驱动函数实现	119
6.4	用户配置文件编写	124
6.4.1	定义设备常量于设备信息常量	124
6.4.2	添加平台初始化与平台解初始化函数	126
6.4.3	在头文件添加变量和函数声明	126
6.5	外设例程文件开发	126
第 7 章	通用外设驱动开发——GPIO	127
7.1	标准层了解	127

7.1.1	了解相关类型定义	127
7.1.2	了解标准接口函数	128
7.2	HW 层开发	131
7.2.1	添加寄存器列表	131
7.2.2	编写寄存器的相关函数	132
7.2.3	添加其他功能性函数	135
7.2.4	添加指向寄存器块的指针	135
7.3	驱动层开发	135
7.3.1	编写结构性程序	135
7.3.2	驱动函数实现	138
7.4	用户配置文件编写	154
7.4.1	定义设备常量于设备信息常量	154
7.4.2	添加平台初始化与平台解初始化函数	154
7.4.3	在头文件添加变量和函数声明	156
7.5	例程文件开发	156
7.5.1	例程内容开发	156
第 8 章	通用外设驱动开发——CRC	160
8.1	标准层了解	160
8.1.1	了解相关类型定义	160
8.1.2	了解标准接口函数	162
8.2	HW 层开发	163
8.2.1	添加寄存器列表	163
8.2.2	编写寄存器的相关函数	164
8.2.3	添加其他功能性函数	166
8.2.4	添加指向寄存器块的指针	166
8.3	驱动层开发	167
8.3.1	编写结构性程序	167
8.3.2	驱动函数实现	170
8.4	用户配置文件编写	175
8.4.1	定义设备常量于设备信息常量	175
8.4.2	添加平台初始化与平台解初始化函数	175
8.4.3	在头文件添加变量和函数声明	176
8.5	例程文件开发	176
8.5.1	例程内容开发	176
第 9 章	通用外设驱动开发——ADC	178
9.1	标准层了解	178
9.1.1	了解相关类型定义	178
9.1.2	了解标准接口函数	182
9.2	HW 层开发	186
9.2.1	添加寄存器列表	186
9.2.2	编写寄存器的相关函数	187
9.2.3	添加其他功能性函数	211
9.2.4	添加指向寄存器块的指针	211
9.3	驱动层开发	212

9.3.1	编写结构性程序	212
9.3.2	驱动函数实现	218
9.4	用户配置文件编写	229
9.4.1	定义设备常量于设备信息常量	229
9.4.2	添加平台初始化与平台解初始化函数	230
9.4.3	在头文件添加变量和函数声明	230
9.5	例程文件开发	231
9.5.1	例程内容开发	231
第 10 章	通用外设驱动开发——UART	235
10.1	标准层了解	235
10.1.1	了解相关类型定义	235
10.1.2	了解标准接口函数	240
10.2	HW 层开发	243
10.2.1	添加寄存器列表	243
10.2.2	编写寄存器的相关函数	244
10.2.3	添加其他功能性函数	255
10.2.4	添加指向寄存器块的指针	257
10.3	驱动层开发	258
10.3.1	编写结构性程序	258
10.3.2	驱动函数实现	261
10.4	用户配置文件编写	267
10.4.1	定义设备常量于设备信息常量	267
10.4.2	添加平台初始化与平台解初始化函数	268
10.4.3	在头文件添加变量和函数声明	269
10.5	例程文件开发	270
10.5.1	例程内容开发	270
第 11 章	通用外设驱动开发——I2C	272
11.1	标准层了解	272
11.1.1	了解相关类型定义	272
11.1.2	了解标准接口函数	275
11.2	HW 层开发	279
11.2.1	添加寄存器列表	279
11.2.2	编写寄存器的相关函数	282
11.2.3	添加其他功能性函数	285
11.2.4	添加指向寄存器块的指针	285
11.3	驱动层开发	285
11.3.1	编写结构性程序	286
11.3.2	驱动函数实现	289
11.4	用户配置文件编写	293
11.4.1	定义设备常量于设备信息常量	293
11.4.2	添加平台初始化与平台解初始化函数	293
11.4.3	在头文件添加变量和函数声明	294
11.5	例程文件开发	294
11.5.1	例程内容开发	295

第 12 章 通用外设驱动开发——SPI	297
12.1 标准层了解	297
12.1.1 了解相关类型定义	297
12.1.2 了解标准接口函数	302
12.2 HW 层开发	311
12.2.1 添加寄存器列表	311
12.2.2 编写寄存器的相关函数	313
12.2.3 添加其他功能性函数	323
12.2.4 添加指向寄存器块的指针	323
12.3 驱动层开发	324
12.3.1 编写结构性程序	324
12.3.2 驱动函数实现	328
12.4 用户配置文件编写	345
12.4.1 定义设备常量于设备信息常量	345
12.4.2 添加平台初始化与平台解初始化函数	346
12.4.3 在头文件添加变量和函数声明	347
12.5 例程文件开发	348
12.5.1 例程内容开发	348
第 13 章 通用外设驱动开发——TIMER	354
13.1 标准层了解	354
13.1.1 了解相关类型定义	354
13.1.2 了解标准接口函数	357
13.2 HW 层开发	363
13.2.1 添加寄存器列表	363
13.2.2 编写寄存器的相关函数	365
13.2.3 添加其他功能性函数	373
13.2.4 添加指向寄存器块的指针	373
13.3 驱动层开发	374
13.3.1 编写结构性程序	374
13.3.2 驱动函数实现	376
13.4 用户配置文件编写	381
13.4.1 定义设备常量于设备信息常量	381
13.4.2 添加平台初始化与平台解初始化函数	382
13.4.3 在头文件添加变量和函数声明	382
13.5 例程文件开发	382
13.5.1 例程内容开发	383
第 14 章 通用外设驱动开发——CAP	388
14.1 标准层了解	388
14.1.1 了解相关类型定义	388
14.1.2 了解标准接口函数	390
14.2 HW 层开发	393
14.2.1 添加寄存器列表	393
14.2.2 编写寄存器的相关函数	393
14.2.3 添加其他功能性函数	396

14.2.4	添加指向寄存器块的指针	397
14.3	驱动层开发	397
14.3.1	编写结构性程序	397
14.3.2	驱动函数实现	400
14.4	用户配置文件编写	406
14.4.1	定义设备常量于设备信息常量	406
14.4.2	添加平台初始化与平台解初始化函数	407
14.4.3	在头文件添加变量和函数声明	408
14.5	例程文件开发	408
14.5.1	例程内容开发	408
第 15 章	通用外设驱动开发——PWM	417
15.1	标准层了解	417
15.1.1	了解相关类型定义	417
15.1.2	了解标准接口函数	418
15.2	HW 层开发	420
15.2.1	添加寄存器列表	420
15.2.2	编写寄存器的相关函数	420
15.2.3	添加其他功能性函数	426
15.2.4	添加指向寄存器块的指针	426
15.3	驱动层开发	426
15.3.1	编写结构性程序	426
15.3.2	驱动函数实现	429
15.4	用户配置文件编写	435
15.4.1	定义设备常量于设备信息常量	435
15.4.2	添加平台初始化与平台解初始化函数	436
15.4.3	在头文件添加变量和函数声明	436
15.5	例程文件开发	437
15.5.1	例程内容开发	437
第 16 章	通用外设驱动开发——RTC	444
16.1	标准层了解	444
16.1.1	了解相关类型定义	444
16.1.2	了解标准接口函数	445
16.2	HW 层开发	446
16.2.1	添加寄存器列表	446
16.2.2	编写寄存器的相关函数	447
16.2.3	添加指向寄存器块的指针	454
16.3	驱动层开发	455
16.3.1	编写结构性程序	455
16.3.2	驱动函数实现	460
16.4	用户配置文件编写	467
16.4.1	定义设备常量于设备信息常量	467
16.4.2	添加平台初始化与平台解初始化函数	468
16.4.3	在头文件添加变量和函数声明	469
16.5	例程文件开发	469

16.5.1	例程内容开发	470
第 17 章	通用外设驱动开发——CAN	471
17.1	标准层了解	471
17.1.1	了解相关类型定义	471
17.1.2	了解标准接口函数	479
17.2	HW 层开发	489
17.2.1	添加寄存器列表	489
17.2.2	编写寄存器的相关函数	491
17.2.3	添加其他功能性函数	497
17.2.4	添加指向寄存器块的指针	497
17.3	驱动层开发	498
17.3.1	编写结构性程序	498
17.3.2	驱动函数实现	507
17.4	用户配置文件编写	537
17.4.1	定义设备常量于设备信息常量	537
17.4.2	添加平台初始化与平台解初始化函数	539
17.4.3	在头文件添加变量和函数声明	539
17.5	例程文件开发	540
17.5.1	例程内容开发	540
第 18 章	特殊外设驱动开发——DMA	552
18.1	特殊外设驱动开发流程	552
18.2	HW 层开发	552
18.2.1	添加寄存器列表	552
18.2.2	编写寄存器的相关函数	553
18.2.3	添加其他功能性函数	558
18.3	驱动层开发	558
18.3.1	编写结构性程序	558
18.3.2	驱动函数实现	562
18.4	用户配置文件编写	570
18.4.1	定义设备常量于设备信息常量	570
18.4.2	添加平台初始化与平台解初始化函数	571
18.4.3	在头文件添加变量和函数声明	571
18.5	例程文件开发	572
18.5.1	例程内容开发	572
第 19 章	GIT 开源发布	578
19.1	GitHub 准备工作	578
19.1.1	安装 git for Windows	578
19.1.2	创建 GitHub 账号	579
19.2	协同开发	581
19.2.1	获取源码	581
19.2.2	源码同步	585
19.2.3	向上同步代码	586
19.2.4	请求合并到源码库	590

第1章 AMetal 简介

在 MCU 产业快速发展的今天，芯片厂商推出了越来越多的 MCU。不同厂商、型号之间，MCU 外设的使用方法千差万别。比如，不同芯片厂商的 I²C 外设寄存器定义和配置流程可能差异很大，使用方法大不相同。这给广大嵌入式开发人员带来了诸多烦恼，特别是在出于种种原因要更换 MCU 时（比如：更换资源更大的 MCU，性价比更高的 MCU 等等）。硬件外设的作用是为系统提供某种功能，AMetal 基于外设的共性（例如，无论何种硬件 I²C，它们都是为系统提供以 I²C 形式的发送或接收数据），对同一类外设功能进行了高度的抽象，由此形成了“服务”的抽象概念，即各种硬件外设可以为系统提供某种服务（例如，I²C 提供数据收发的服务）。服务是抽象的，不与具体硬件绑定。同时，为了使应用程序使用这些服务，AMetal 还定义了一系列标准化的软件接口。由于这些标准化的软件接口用于应用程序使用硬件外设提供的各种服务，因此，可以将其视为标准化的服务接口。

由于服务是对某一类功能高度抽象的结果，与具体芯片、外设、器件及实现方式均无关。使得不同厂商、型号的 MCU 外设都能以相同的一套服务接口进行操作。即使底层硬件千变万化，都可以通过使用一套简洁的接口控制相应的外设。如此一来，只要应用程序基于标准化的服务接口进行开发，那么，应用程序就可以跨平台使用，在任何满足资源需求的硬件平台上运行，即使更换了底层硬件，也不会影响到应用程序。换句话说，无论 MCU 如何改变，则基于 AMetal 平台的应用软件均可复用。

可以标准化的服务接口具有跨平台（可以在不同的 MCU 中使用）特性，为便于描述，通常也将其称之为“标准接口”或“通用接口”。

1.1.1 AMetal 的特点

AMetal 具有以下特点：

- 开源，开源地址：<https://github.com/zlgopen/ametal>;
- 外设功能标准化，提供了一系列跨平台 API，使应用程序可以跨平台复用；
- 不依赖于操作系统服务；
- 开放外设所有功能；
- 独立的命名空间 am_，可以避免与其它软件包冲突；
- 能独立运行，提供工程模板与 demo，用户在此基础上快速开发应用程序；
- 封装时将效率和变化部分放在第一位，用户不看手册也能使用；
- 上层系统基于 AMetal 开发外设驱动，无需针对各种繁杂外设分别开发驱动；

1.1.2 AMetal 源码获取

1. 获取源码（github）

AMetal 开源于 Github，其开源网址为“<https://github.com/zlgopen/ametal>”，用户可直接通过此页面下载 AMetal 开源代码包，如图 1.1 所示，点击“Clone or download”，再点击“Download ZIP”即可下载开源代码包。

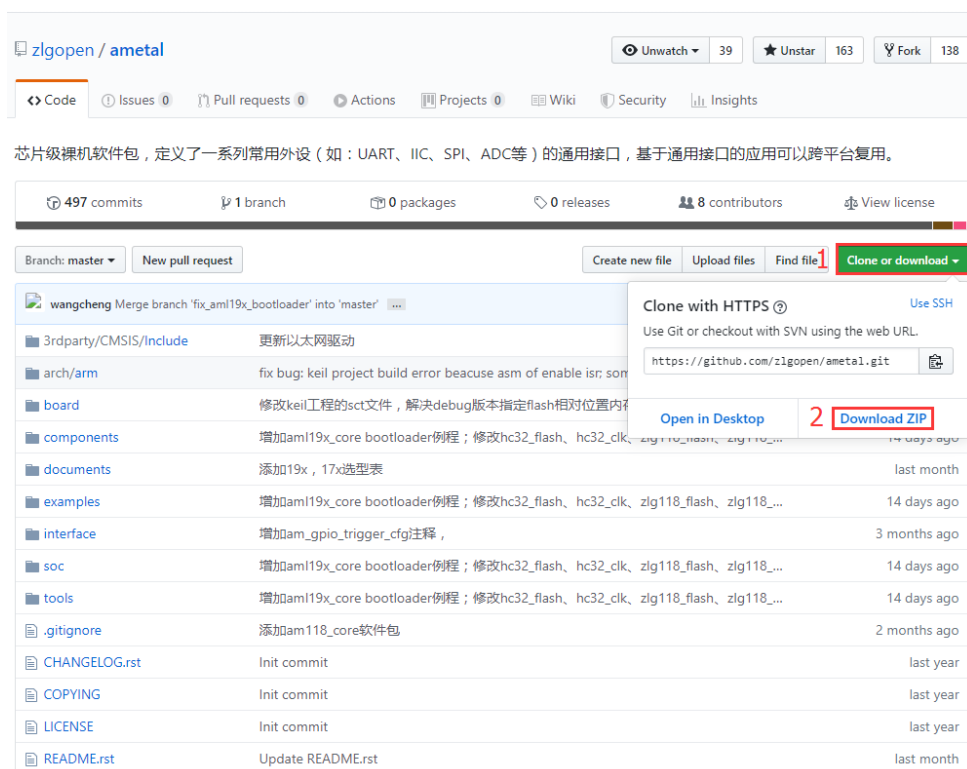


图 1.1 在 Github 上下载 AMetal 代码包

下载解压后,在“ametel/documents”目录下有一系列文档,可帮助用户快速了解 AMetal,并使用 AMetal 进行项目开发。

2. 获取源码(码云)

AMetal 同步开源于码云,其开源网址为“https://gitee.com/zlgopen/ametel”,用户可直接通过此页面下载 AMetal 开源代码包。如图 1.2 所示,点击“克隆/下载”,再点击“下载 ZIP”即可下载开源代码包。



图 1.2 在码云上下载 AMetal 代码包

下载解压后,在“ametal/documents”目录下有一系列文档,可帮助用户快速了解 AMetal,并使用 AMetal 进行项目开发。

1.1.3 AMetal 源码目录结构简介

本小节将详细介绍 AMetal 软件包的目录架构,用户可以通过查阅本节,了解应该将自己编写的文件放在软件包的哪个位置。

AMetal 根目录详见图 1.3,表 1.1 简单描述了这些文件夹及文件的功能。

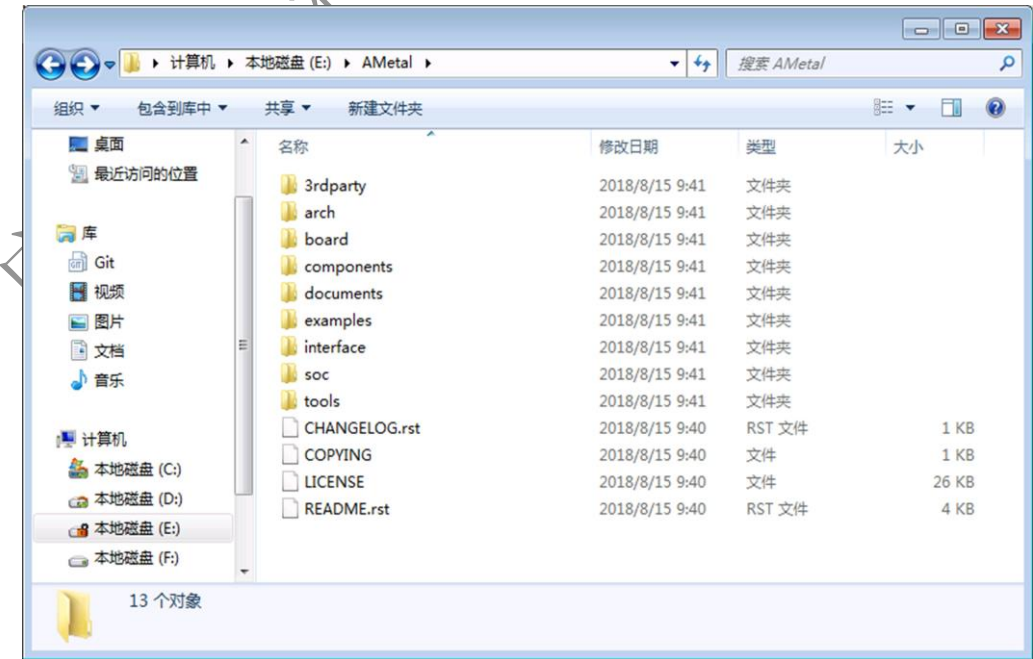


图 1.3 AMetal 根目录架构

表 1.1 AMetal 根目录架构

文件/文件名	描述
3rdparty	存放第三方软件包；
arch	存放内核相关文件；
board	存放板级相关文件；
components	存放组件文件；
documents	存放各类文档；
examples	存放各类例程；
interface	存放 AMetal 标准接口文件；
soc	存放片上系统（MCU，如 ZLG116）相关文件；
tools	存放一些工具包，如 Keil 的 PACK 包；
CHANGELOG	版本修改记录文件；
COPYING	版本声明文件；
LICENSE	版权许可证文件（AMetal 采用的是 LGPL 许可证）；
README	README 文件；

夹，如 arm 文件夹。

3. board

board 文件夹包含了与开发板相关的文件，如启动文件及与开发板相关的设置和初始化函数等。board 文件夹内分为板级通用文件夹“bsp_common”和若干某一型号开发板的专用文件夹，如“am116_core”。分别用于存放板级通用文件和对应型号开发板的专用板级文件。

4. components

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

5. documents

documents 用于存放 SDK 相关文档，各文档内容如下：

- 《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。
- 《用户手册.pdf》：用户手册详细介绍了 AMetal 架构、目录架构、平台资源以及通用外设常见的配置方法。
- 《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。
- 《引脚配置及查询.xlsm》：引脚配置及查询表可以用于查询引脚的上下拉模式和引脚速率，里面还详细介绍了各个引脚可以用于哪些外设，并提供了可以快速生成对应于外设的引脚配置代码。

documents 内部按不同型号的开发板分成不同的文件夹（如“am116_core 文件夹”），存放与其开发板硬件直接相关的文档（如《引脚配置及查询.xlsm》）。其他通用文件直接存放在 documents 第一级目录下。

6. examples

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. arch

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

examples 文件夹主要包含各级示例程序, 包括硬件层 demo、驱动层 demo、板级 demo 及组件 demo 等。

7. interface

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

以 “am_gpio.h” 文件为例, 它包含了用于控制 GPIO 的各个函数的函数原型及一些参数宏定义, 如 GPIO 配置函数 “int am_gpio_pin_cfg(int pin, uint32_t flags) ”, 用户可以直接调用此函数对 MCU 的 GPIO 引脚进行配置, 而不用考虑不同芯片之间的差异。

8. soc

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

9. tools

tools 目录下存放 SDK 相关工具, 如 Keil 的 PACK 包。

1.1.4 AMetal 的快速使用

在正式开发前, 先阅读 AMetal 仓库中几个入门级文档 (位于仓库中的 documents 目录下):

- 《AMetal 目录结构》

讲解 AMetal 的目录结构, 使用户对 AMetal 仓库的目录结构有着更加深入的了解, 知道各个文件夹下存放的何种文件。

- 《快速入门手册(keil)》

快速搭建 keil 集成开发环境, 讲解如何安装相关软件、新建工程、调试应用程序、下载应用程序等基本操作。

- 《快速入门手册(eclipse)》

快速搭建 eclipse 集成开发环境, 讲解如何安装相关软件、新建工程、调试应用程序、下载应用程序等基本操作。

第2章 MCU 新型号适配概述

约定如下, 本文选用广州致远微电子有限公司最新发布的 ZMF236 系列芯片做为开发适配范例, 该系列芯片采用 M0 内核, 主频可达 48M, 软件可兼容 ZMF216、ZMF226 系列芯片, 命名空间采用 amf2x6 作为关键字。接下来让我们从零开始, 逐步在 AMetal 上完成 ZMF2x6 的 SDK 开发。

驱动的开发需要严格遵守我司代码规范要求 and doxygen 文档的注释要求, 代码规范请参阅广州致远电子软件规范《代码规范》一文, 本文不做展开叙述。

2.1 理论学习：AMetal 驱动框架解析

在开始开发之前, 我们先要对 AMetal 平台架构有一定了解。AMetal 中, 所有外设都分为 3 层, 标准接口层、驱动层和 HW 层, 三层都有对应的头文件, 以引出相应的接口, 用户均可使用。同时为了方便用户使用, 还需要提供相应的用户配置文件, 所以开发 AMetal 的驱动软件, 就从这几部分开始入手。

这里以 ZMF2x6 通用外设为例, 给出相应的的文件结构关系图, 由图 2.1 可见, 一个外设相关的文件有: HW 层文件、驱动层文件和用户配置文件。通常情况下, HW 层提供了直接操作硬件寄存器的接口, 接口实现简洁, 往往以内联函数的形式放在 .h 文件中, 因此, HW 层通常只包含 .h 文件, 但当某些硬件功能设置较为复杂时, 也会提供对应的非内联函数, 存放在 .c 文件中。驱动层作为中间层, 其使用 HW 层接口, 实现了标准接口层中定义的接口, 以便用户使用标准 API 访问该外设。标准层已有抽象好的函数接口, 不需要开发者再重新开发, 只需了解即可。用户配置文件完成了相应驱动的配置, 如引脚数目、中断等。

用户在开发自己的应用程序时, 可以直接调用这 3 层的代码来实现自己的应用, 调用 HW 层代码(直接操作寄存器)执行效率最高不过可移植性最差, 开发难度也最高; 调用标准层实现应用, 软件最容易编写也最方便移植, 一般建议用户优先使用标准层实现软件应用, 可大大提高开发效率。

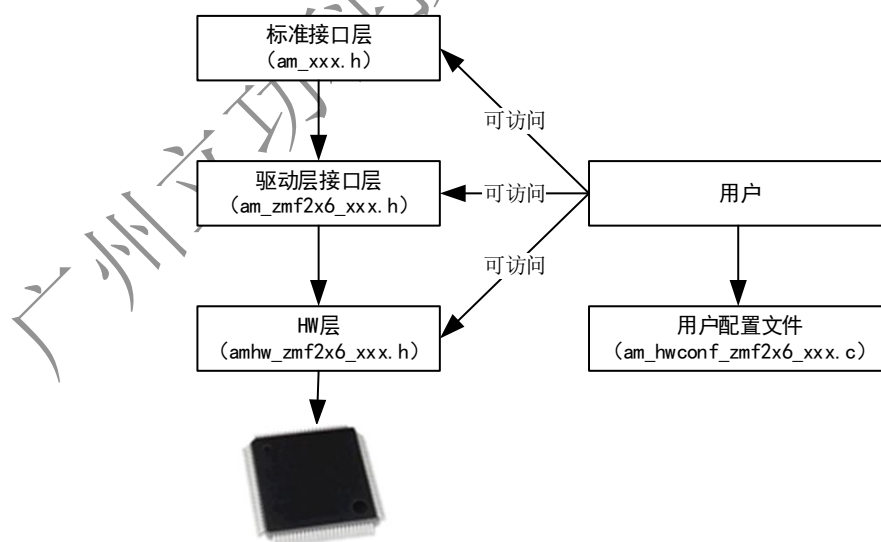


图 2.1 通用外设文件结构

本小节仅简单概述了 AMetal 的驱动的结构和各层之间的关系, 想要更清楚的了解各层实现及应用, 请参考 AMetal 驱动框架解析一章。

2.2 小试牛刀：让 MCU 跑起来

通过对 AMetal 架构的理论学习，我们基本了解了 AMetal 的驱动框架。本节开始动手逐步实现驱动开发。在开发外设驱动之前，我们需要在 AMetal 上添加一个 MCU 的基础工程，先让 MCU 跑起来，后续添加外设驱动等均是基于此工程添加和测试验证。

由于各外设驱动还没有编写，一般 MCU 启动后默认会使用内部 RCH 时钟运行，因此我们只需要编写好相应的启动文件就可以让 MCU 运行起来。搭建完成可以通过直接 Debug 程序来观察它是否能够进入主函数，如果可以就代表 MCU 运行成功，就可以进入下一步，通用驱动外设开发工作。

关于如何搭建基础工程，如何编写启动文件，如何让 MCU 跑起来进入调试模式，可直接参考让 MCU 跑起来一章。

2.3 筑基打梁：通用外设驱动开发

搭建好基础工程让 MCU 跑起来之后，我们就可以开始通用外设驱动开发了。外设驱动开发主要包含硬件层驱动函数编写、驱动层函数实现、通用示例程序及其相关外设的配置文件开发。主要是下面几个流程：

1. 了解标准函数接口

对于通用 MCU 而言，AMetal 基本提供了各个外设的标准层接口，不需要额外开发，作为开发者只需要去了解即可。虽然标准函数接口不用用户在编写，但驱动函数该如何实现，或者说该实现怎样的函数接口，还是需要先了解标准外设的接口函数。

2. 通用外设硬件层开发

通过对标准层的了解，我们知道标准层实现的函数的功能，以及需要驱动层提供的函数，但驱动层并不是凌空搭建的，驱动层介乎与 HW 层和标准接口层之间，简单来说就是调用 HW 层的函数来提供给标准层，所以在写驱动层前，要先来完成 HW 层。

HW 层对应到具体的外设，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，内联函数直接定义在.h 文件中。硬件层所有的符号命名以 amhw_/AMHW_ 开头。通常情况下，HW 层只有.h 文件，只有当某些硬件功能设置较为复杂时，才提供对应的非内联函数，存放在.c 文件中。

开发 HW 层前，先打开芯片用户手册，先了解一下该外设的寄存器，对该外设功能有一定的了解之后再开始开发，流程如下：

- 1) 添加寄存器列表
- 2) 编写寄存器相关操作函数
- 3) 检查补充所需功能函数
- 4) 添加指向寄存器块的指针

3. 外设驱动层开发

通过对标准接口层文件的分析理解，清楚了驱动层需要为标准接口层提供哪些内容，加上驱动本身应该提供给用户的初始化函数和解初始化函数。外设驱动层需要编写的总体内容再归纳总结如下：

- 1) 编写好标准接口层需要使用的基本函数；
- 2) 驱动初始化函数；
- 3) 驱动解初始化函数。
- 4) 外设用户配置文件开发

由于用户调用驱动初始化函数时需要传入设备地址和设备信息地址作为参数。为了方便用户使用,我们需要将设备变量和设备信息常量定义好,用户调用驱动初始化函数时,只需要使用定义好的设备变量和设备信息常量即可,不需要再额外定义,这个用于存放设备变量和设备常量信息和驱动初始化函数的文件,就是外设用户配置文件,我们需要为每一个外设提供一个独立的配置文件。

4. 外设例程文件开发

最后是编写例程文件,例程文件开发主要分为两个部分,主要有基于硬件层实现的 HW 例程、基于标准层实现的 STD 例程以及在板测试板级例程,到了这一步,我们需要依托硬件测试验证我们的代码功能。

注:调试例程时,应该在 template 工程下进行,不需要新建其它工程,只需要将新建的文件加入到工程结构的 user_code 下。例程文件会与原先的 main.c 文件冲突(因为都存在 main 函数),这时,只需要调试时将原先的 main.c 不要包含在工程编译中,例程文件开发完毕后,再恢复将 main.c 包含在工程中,同时将例程文件移除工程即可。

2.4 查缺补漏:特殊驱动开发

完成了上述的通用标准驱动,还需要检查是否还有其他特殊的驱动未添加。所谓的特殊驱动主要是指 AMetal 中目前还没有提供标准接口文件的外设,即并非多数芯片都有的外设,如少部分芯片带 LCD 外设,而大部分芯片不带,这种接口抽象出来也只有某芯片可用,无法共用到其他芯片的外设接口,也就无法随便移植到其他芯片,这类驱动就需要单独开发,又例如 DMA 外设,各家芯片 DMA 外设级联工作方式不完全相同,这类也很难抽象出统一的接口供用户使用,也需要独立开发驱动。

特殊驱动开发由于没有标准层,这就需要开发者对该外设的功能等有较清晰的理解,了解该外设可以实现哪些功能以及需要实现什么功能。这类驱动的开发流程与通用外设开发并无多大差别,同样从开发硬件层开始,然后实现驱动层函数、配置文件,最后是提供示例程序并测试其功能。

这类驱动开发可参考特殊驱动开发一章。

2.5 共享发布:GIT 开源发布

目前 AMetal 的源码基于 gitee 和 github 开源发布,地址为 <https://gitee.com/zlgopen/ametal> 和 <https://github.com/zlgopen/ametal>,非内部开发者获取软件途径只有上述两个地址,一般建议用户优选 gitee,服务器在国内,下载上传更新速度较 github 会快很多,同时建议用户或开发者均采用 git 管理代码,方便追溯和管理代码,对软件开发会有较大帮助。

个人开发者或第三方开发者添加自家芯片时,可基于发布的 master 分支上创建自己的分支去开发,开发完成之后,可以推送到 git 端发起合并申请,负责审核的审核人员验证代码之后,可以同意合并,这样,其他用户只要在本通过 git 下拉更新代码,就可以获取到你开发的代码,从而基于你的代码开发应用。同样,你也可以通过这种方式,去获取别人提交的代码,从而丰富自己的软件平台。

基于 git 开发到开源发布的流程参考 GIT 开源发布一章。

第3章 AMetal 驱动框架解析

如下为 AMetal 平台的基本架构，AMetal 共分为 3 层，硬件层、驱动层和标准接口层，详见图 3.1，基于 AMetal 开发添加新的芯片，也需要按照这个架构编写驱动代码，向用户提供各层的函数接口。

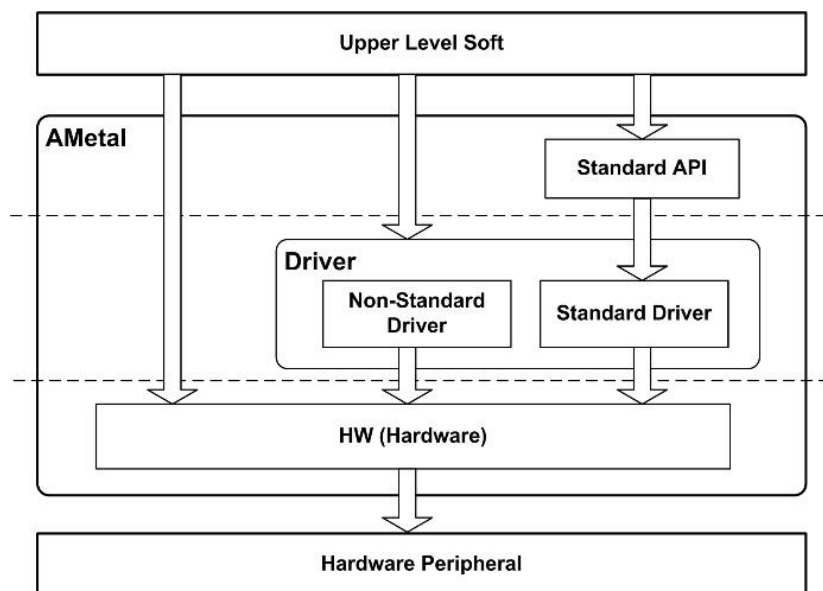


图 3.1 AMetal 架构

3.1 标准接口层

标准接口层对常见外设的操作进行了抽象，提出了一套标准 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。常用的设备均已定义了相应的接口，例如：LED、KEY、Buzzer、Digitron、GPIO、USB、CAN、Serial、ADC、DAC、I²C、SPI、PWM、CAP 等等。用户使用一个 GPIO 的过程：先调用驱动初始化函数，后续在编写应用程序时仅需直接调用标准接口函数即可。传统实现是直接实现接口。

可见，应用程序基于标准 API 实现的，标准 API 与硬件平台无关，使得应用程序可以轻松在不同的硬件平台上运行。用户应尽可能基于标准接口编程，以便应用程序跨平台复用，接口命名空间 am_/AM_。如下是 PWM 的三个标准接口函数，详见表 3.1。

表 3.1 PWM 标准接口函数

函数接口	功能
int am_pwm_config (am_pwm_handle_t handle, int chan, unsigned long duty_ns, unsigned long period_ns);	配置 PWM 通道的周期和脉宽
int am_pwm_enable (am_pwm_handle_t handle, int chan);	使能 PWM 输出
int am_pwm_disable (am_pwm_handle_t handle, int chan);	禁能 PWM 输出

a
m_p
wm_
confi
g 接
口 中
有 四
个 参
数，
hand

le、chan、duty_ns、period_ns，其中 chan、duty_ns、period_ns，分别是 PWM 的通道号，脉宽时间（高电平时间，单位：ns），PWM 周期时间（单位：ns），handle 参数可能对于初学者比较陌生，下面就用对 handle 做一个详细的介绍。

3.1.1 标准服务句柄 Handle

handle 是一个标准服务句柄，由实例初始化函数返回，提供标准服务。返回值为 NULL 表明初始化失败；其它值表明初始化成功。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，操作对应的设备。所以 handle 是一个可操作的设备对象，比如这里的 PWM 的 handle，就是一个可操作的设备对象，在使用前必须初始化 PWM，来获取一个可操作的设备对象，例如 ZLG116 的 PWM 获取 handle 的代码如下：

```
am_pwm_handle_t am_zlg116_sct0_pwm_inst_init(void);
```

函数的返回值为 am_pwm_handle_t 类型的标准服务句柄，该句柄将作为 PWM 通用接口中 handle 参数的实参。类型 am_pwm_handle_t (am_pwm.h) 定义如下：

```
typedef struct am_pwm_serv *am_pwm_handle_t;
```

因为初始化函数返回的 PWM 标准服务句柄仅作为参数传递给 PWM 通用接口，使用时不需要对该句柄作其它任何操作，所以完全不需要对该类型作任何了解。仅仅需要特别注意的是，若函数返回的实例句柄的值为 NULL，则表明初始化失败，该实例句柄不能被使用。

3.1.2 对功能进行抽象

为了使标准接口可以跨平台复用，则需要对功能进行抽象（即面向对象的思想）。在实现功能函数时，可以通过不同的控制方式实现，比如使用 LED 时，可以通过 GPIO 控制，也可以通过 HC595 控制，虽然它们对应 am_led_set() 和 am_led_toggle() 的实现方法不同，但它们要实现的功能却是一样的，这是它们的共性：均是实现设置 LED 状态和翻转 LED 状态的功能。再比如 PWM，也有不同的实现方式，甚至通过不同的评估板上实现，但实现的功能却是一样的，均是实现 PWM 的配置和使禁能。由于一个接口的实现代码只能有一份，因此它们的实现不能直接作为通用接口的实现代码。为此，可以对它们的共性进行抽象，抽象为对应的方法，下面对 PWM 抽象出来三个方法：

```
int (*pfn_pwm_config)( void *p_drv,
                       int chan,
                       unsigned long duty_ns,
                       unsigned long period_ns);

int (*pfn_pwm_enable)(void *p_drv, int chan);

int (*pfn_pwm_disable)(void *p_drv,int chan);
```

因此对功能进行抽象先要定义抽象设备和抽象设备的方法，然后根据抽象方法的定义和设备具体的功能，将功能一一抽象为方法，并将这些方法存放在一个虚函数表（虚函数表详见 funcs）中。这些抽象方法的第一个参数均为 p_drv。最后，将虚函数表和 p_drv 整合在一个新的结构体中，该结构体类型即为抽象设备类型。其抽象设备类型的结果如下：

```
typedef struct am_{dev_name}_drv_funcs {           // 虚函数表
    int (*pfn_function1)(void *p_drv, ...);        // 抽象方法 1
    int (*pfn_function2)(void *p_drv, ...);        // 抽象方法 2
    // .....
    // 抽象方法 n
}am_{dev_name}_drv_funcs_t;
```

```
typedef struct am_{dev_name}_dev {
    const am_{dev_name}_drv_funcs_t *p_funcs;    // 设备的驱动函数
    void *p_drv;                                // 驱动函数参数
    // 其它可能的成员，如抽象 LED 设备中的 p_next
    ...
}am_{dev_name}_dev_t;
```

伪代码中，{dev_name}(为设备名，例如:spi、GPIO、LED，PWM)是为了下文叙述的方便，使用的一个代号。其中，p_funcs 为指向驱动虚函数表的指针，p_drv 为指向设备的指针，即传递给驱动函数的第一个参数。相对通用接口来说，抽象方法多了一个 p_drv 参数。在面向对象的编程中，对象中的方法都能通过隐形指针 p_this 访问对象自身，引用自身的一些私有数据。而在 C 语言中则需要显式的声明，这里的 p_drv 就有相同的作用。下面是 PWM 设备类的定义详见程序清单 3.1。

程序清单 3.1 PWM 抽象设备

```
1  typedef struct am_pwm_serv {
2      struct am_pwm_drv_funcs *p_funcs;        /*PWM 驱动函数结构体指针 */
3      void *p_drv;                            /* 用于驱动函数的第一个参数 */
4  }am_pwm_serv_t;
```

结构体中包含了驱动虚函数表的指针，里面包含的是驱动设备底层的功能服务函数，p_drv 用于驱动函数的第一个参数。

p_funcs 为指向驱动虚函数表的指针，顾名思义虚函数表中存放的是驱动虚函数。

定义虚函数表时，需要包含设备的各个方法，例如 LED 分别用于设置 LED 的状态和翻转 LED 的方法。针对不同的硬件设备，可以根据自身特性实现这两个方法。而对于 PWM 则有配置、使能、禁能三个方法，因此需要将所有抽象方法放在一个结构体中，形成一个虚函数表，比如 PWM 的虚函数表的定义详见程序清单 3.2。

程序清单 3.2 PWM 虚函数表定义

```
1  struct am_pwm_drv_funcs {
2      int (*pfn_pwm_config)( void *p_drv,
3                              int chan,
4                              unsigned long duty_ns,
5                              unsigned long period_ns);
6      int (*pfn_pwm_enable)(void *p_drv, int chan);
7      int (*pfn_pwm_disable)(void *p_drv,int chan);
8  };
```

如此一来，标准接口的实现则可以直接调用抽象方法实现，而抽象方法的具体实现是由具体的驱动完成。

3.1.3 基于抽象服务实现接口

由于对功能进行了抽象，标准接口的实现和具体芯片没有直接关系，在标准接口函数中，标准接口函数通过调用设备实例初始化 handle 直接找到相应的方法，实现的标准接口的功

能。在标准接口实现中，没有与具体硬件相关的实现代码，仅仅是简单的调用对应的抽象方法，因此也就可以在不同的评估板上使用。抽象方法需要由具体的设备来完成。由于各个标准接口的实现非常简单，往往将其实现直接以内联函数的形式存放在.h 文件中。下面是 PWM 的配置标准接口的实现，如下：

```
int am_pwm_config( am_pwm_handle_t  handle,
                  int                chan,
                  unsigned long      duty_ns,
                  unsigned long      period_ns)
{
    return handle->p_funcs->pfn_pwm_config( handle->p_drv,
                                           chan,
                                           duty_ns,
                                           period_ns);
}
```

由此可见，由于 `handle` 是直接指向设备的指针，可以通过 `handle` 直接找到相应的方法，因此，在整个接口的实现过程中，没有任何查询搜索的过程，效率较高。除此之外，一些设备在一个系统中可能有多个，可以集中对各个设备进行管理，如 LED 设备可能不止一个，不得不使用单向链表将系统中的各个 LED 设备链起来，便于查找。下面是 LED 的设置标准接口，如下：

```
int am_led_set (int led_id, am_bool_t state)
{
    am_led_dev_t *p_dev = __gp_led_dev;
    if (p_dev == NULL) {
        return -AM_ENODEV;
    }
    if (p_dev->p_funcs->pfn_led_set) {
        return p_dev->p_funcs->pfn_led_set(p_dev->p_drv, led_id, state);
    }
    return -AM_ENOTSUP;
}
```

LED 设备为全局变量 `__gp_led_dev` 指向的设备，展示了 `pfn_led_set` 方法的调用形式。而实际上 LED 设备往往不止一个，比如，用 GPIO 控制的 LED 设备和使用 HC595 控制的 LED 设备，就需要在系统中管理多个 LED 设备。由于它们的具体数目无法确定，因此需要使用单向链表进行动态管理。在 `am_led_dev_t` 中增加一个 `p_next` 成员，用于指向下一个设备。具体的定义如下：

```
typedef struct am_led_dev {
    const am_led_drv_funcs_t *p_funcs;           // 本设备的驱动
    void *p_drv; // 驱动函数参数
    struct am_led_dev *p_next;
}
```



```

struct am_led_dev *p_next;           // 指向下一个 LED 设备
}am_led_dev_t;

```

此时，系统中的多个 LED 设备使用链表的形式管理。那么在标准接口的实现中，如何确定该使用哪个 LED 设备呢？在定义标准接口时，使用了 `led_id` 区分不同 LED，若将一个 LED 设备和该设备对应的 `led_id` 绑定在一起，则在标准接口的实现中，就可以根据 `led_id` 找到对应的 LED 设备，然后使用驱动中提供的相应方法完成 LED 的操作。可见如果需要，可以在抽象设备中添加其它需要的成员。

3.2 驱动层

驱动层重点是实现标准接口层定义的抽象服务，实现各个功能函数，接口命名空间：`am_{dev_name}/AM_{dev_name}`。主要包含三大类接口：设备初始化函数：`am_{dev_name}_init()`、设备解初始化函数：`am_{dev_name}_deinit()`、其它功能性函数：`am_{dev_name}_*()`。实际上用户在使用驱动时不需要手动调用设备初始化函数，只需要调用封装好的实例初始化函数，即可使用标准接口。设备初始化会在实例初始化中被调用。同样设备解初始化函数也不需要手动调用，只需要调用封装好的实例解初始化函数，设备解初始化会在实例解初始化中被调用。

3.2.1 抽象服务的实现

到此为止，已经完成了抽象设备框架的搭建，但抽象设备框架中并没有实质性的驱动，标准接口最终需要找到底层驱动。因此需要把底层的驱动方法实现，同时给虚函数表赋值。具体 PWM 抽象方法的定义，详见程序清单 3.2 中的三个抽象方法。

对于所有的 PWM 底层而言，都需要实现这三个抽象方法，这三个方法的实现可以通过多种控制方式完成，无论是哪一种控制方式，都只需要按抽象方法的定义形式，完成该功能，把对应的抽象方法填到虚函数表中，供标准接口调用。比如下面两个不同的板载实现的抽象方法，分别详见程序清单 3.3 和程序清单 3.4。

程序清单 3.3 ZLG116 的 PWM 驱动

```

1  static int __zlg116_tim_pwm_config ( void          *p_drv,
2
3                                     int             chan,
4
5                                     unsigned long    duty_ns,
6
7                                     unsigned long    period_ns)
8  {
9
10     // 根据具体硬件实现，如设置 ZLG116 的 TIM 定时器相关的定时器
11 }
12 static int __zlg116_tim_pwm_enable (void *p_drv, int chan)
13 {
14     //.....

```

```

15 }
16 static const struct am_pwm_drv_funcs __g_zlg116_tim_pwm_drv_funcs = {
17     __zlg116_tim_pwm_config,
18     __zlg116_tim_pwm_enable,
19     __zlg116_tim_pwm_disable,
20 };

```

程序清单 3.4 LPC82x 的 PWM 驱动

```

1  static int __lpc82x_tim_pwm_config ( void          *p_drv,
2                                     int              chan,
3                                     unsigned long    duty_ns,
4                                     unsigned long    period_ns)
5  {
6      // 根据具体硬件实现，如设置 LPC82x 的 TIM 定时器相关的定时器
7  }
8  static int __lpc82x_tim_pwm_enable (void *p_drv, int chan)
9  {
10     //.....
11 }
12 static int __lpc82x_tim_pwm_disable (void *p_drv, int chan)
13 {
14     //.....
15 }
16 static const struct am_pwm_drv_funcs __g_lpc82x_tim_pwm_drv_funcs = {
17     __lpc82x_tim_pwm_config,
18     __lpc82x_tim_pwm_enable,
19     __lpc82x_tim_pwm_disable,
20 };

```

显然，`__g_zlg116_tim_pwm_drv_funcs` 是通过 ZLG116 的定时器实现 PWM 的三个功能，`__g_lpc82x_tim_pwm_drv_funcs` 是通过 LPC824 的定时器实现 PWM 的三个功能，它们在形式上是两个不同的结构体常量，但其意义是相同的。

在这里分别实现了底层 LPC82x 和 ZLG116 的 PWM 驱动，其中都包含了三个函数，分别是 PWM 的配置函数，PWM 的使能函数，PWM 的禁能函数。要使标准层接口可以调用到这些底层函数，需要把这些抽象方法添加到抽象设备中。下面分别是两个评估板对抽象设备成员的赋值，详见程序清单 3.5 和程序清单 3.6。

程序清单 3.5 ZLG116 的 PWM 驱动成员赋值

```

1  am_pwm_serv_tzlg116_pwm_serv;
2

```

```
3   zlg116_pwm_serv.p_funcs= &__g_zlg116_tim_pwm_drv_funcs;
```

程序清单 3.6 LPC82x 的 PWM 驱动成员赋值

```
1   am_pwm_serv_t lpc82x_pwm_serv;
2
3   lpc82x_pwm_serv.p_funcs= &__g_lpc82x_tim_pwm_drv_funcs;
```

从以上代码可以看出，实现抽象服务，即是先把各设备的驱动函数实现，然后完成抽象服务中各成员的赋值，最后把抽象服务添加到服务列表中，提供给上层标准接口。这样解决了驱动共存的问题。

3.2.2 实例初始化

使用标准接口需要用到参数 `handle`，实例初始化的核心则是获取标准服务句柄 `handle`。为了减少用户的工作，避免用户自定义设备实例和实例信息。针对每个设备，都定义了默认的设备实例和实例信息。

任何设备使用前，都需要初始化。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，并且不需任何传入参数，该函数即为实例初始化函数，用于初始化一个外设。实例初始化函数原型详见程序清单 3.7。

程序清单 3.7 实例初始化函数原型

```
1   am_{dev_name}_handle_t am_{dev_name}_inst_init(void);
2   {
3       return am_{dev_name}_init(&__g_{dev_name}_dev,
4       &__g_{dev_name}_devinfo);
5   }
```

从上面代码可见，实例初始化函数动作行为很单一，仅仅是调用一下 `am_{dev_name}_init` 函数。其中需要传入的两个参数，`__g_{dev_name}_dev` 和 `__g_{dev_name}_devinfo` 分别是设备实例和设备信息，在设备配置文件中已经定义了默认的设备实例和实例信息。通常用户在使用时，仅仅需要修改 IO 口，若有特殊的需要，比如需要更改通道号等等，只需要在配置文件中修改对应部分的设备信息。

1. 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，如该外设对应的寄存器基地址、使用的中断号，引脚号等等。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_{dev_name}.c`，可以看到定义的设备信息，例如打开 `\am_hwconf_zlg116_gpio.c` 为 GPIO 的设备信息，详见程序清单 3.8 所示。

程序清单 3.8 设备信息定义

```
1   const am_{dev_name}_devinfo_t __g_{dev_name}_devinfo = {
2
3       /* 外设对应的寄存器基地址、使用的中断号等等 */
4   }
```

```
5  };
```

使用 `am_{dev_name}_devinfo_t` 类型定义了一个设备信息结构体。设备信息结构体类型在相应的驱动头文件中定义。该类型的具体路径为 `ametal\soc{core_type_name}\drivers\include\{dev_name}\am_{dev_name}.h`。例如 `ametal\soc{core_type_name}\drivers\include\gpio\am_zlg116_gpio.h` 文件中的 `am_zlg116_gpio_devinfo_t` 详见程序清单 3.9。

程序清单 3.9 设备信息结构体类型定义

```
1  /**
2   * 设备信息
3   */
4  typedef struct am_{dev_name}_devinfo {
5
6      /* 外设对应的寄存器基地址、使用的中断号、使用的引脚号等等 */
7
8      void (*pfn_plfm_init)(void);          /* 平台初始化函数 */
9      void (*pfn_plfm_deinit)(void);        /* 平台去初始化函数 */
10 }am_{dev_name}_devinfo_t;
```

作为一个设备用到的信息都可以添加到这个结构体中，设备信息中通常有外设对应的寄存器基地址、使用的中断号、有时需要用户根据实际情况分配的内存、还有平台初始化函数和平台解初始化函数等等，当然一些比较特殊的设备没有设置平台初始化函数和平台解初始化函数等等这些成员，这个根据设备的需要进行添加。

下面对几个经常使用到的设备信息成员，如寄存器基地址、平台初始化和平台解初始化以及有时需要用户根据实际情况分配的内存操作，做一个简单的介绍。

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，也有部分较为特殊的，如 GPIO 则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。寄存器基地址已经在 `ametal\soc{core_type_name}\{core_type_name}_regbase.h` 文件中使用宏定义好了，用户直接使用即可。

2) 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。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。

3) 平台初始化

平台初始化函数主要用于初始化与该外设相关的平台资源,如使能该外设的时钟,初始化与该外设相关的引脚等。一些通信接口,都需要配置引脚,如 UART、SPI、I²C 等,这些引脚的初始化都需要在平台初始化函数中完成。

在设备信息结构体类型中,会有一个用于存放平台初始化函数的指针,以指向平台初始化函数,详见程序清单 3.10。当驱动程序初始化相应外设前,将首先调用设备信息中提供的平台初始化函数。

程序清单 3.10 设备信息结构体类型-平台初始化函数指针定义

```
1/** 平台初始化函数 */
2 void (*pfn_plfm_init)(void);
```

平台初始化函数均在设备配置文件中定义,平台初始化函数在 {HWCONFIG}\am_hwconf_{dev_name}.c 这个初始化函数中主要是在外设使用前,先提前配置好所需的 IO 口、系统配置时钟使能禁能等等。

4) 平台解初始化

平台解初始化函数就是和平台初始化函数相对应。通常情况下,平台初始化函数时开启和板子相关的硬件资源例如:时钟,GPIO 口配置等。解初始化就是对应关闭平台初始化配置的资源。

2. 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_*.c, 可以看到设备实例已经定义好,详见程序清单 3.11。

程序清单 3.11 定义设备实例

```
1 /** 设备实例 */
2 am_{dev_name}_dev_t __g_{dev_name}_dev;
```

这里使用 am_{dev_name}_dev_t 类型定义了一个设备实例。设备结构体类型在相对应的驱动头文件中定义。该类型即在 ametal\soc\{core_type_name}\drivers\include\{dev_name}\am_{dev_name}.h 文件中定义。比如 PWM 的设备实例详见程序清单 3.12。

程序清单 3.12 PWM 输出功能设备结构体

```
1 /**
2  * PWM 输出功能设备结构体
3  */
4 typedef struct am_zlg_tim_pwm_dev {
5     /**< 标准 PWM 服务 */
```

```

6      am_pwm_serv_t pwm_serv;
7      /** 指向 TIM(PWM 输出功能) 设备信息常量的指针 */
8      const am_zlg_tim_pwm_devinfo_t *p_devinfo;
9  }am_zlg_tim_pwm_dev_t;

```

从上结构体可以看到设备实例中包含标准 PWM 服务结构体和设备信息常量的指针，但是未被初始化的设备实例为空。所以，需要对设备实例进行初始化，即为设备初始化。

3. 设备初始化

设备初始化函数主要的目的是初始化设备实例，对设备实例结构体中的成员进行一一赋值，此外还有根据设备具体的需要进行驱动设备的初始化，例如需要使用 io 口，则对 io 口进行配置，需要用到定时器，对定时器初始化。例如 PWM 的设备初始化函数详见程序清单 3.13。

程序清单 3.13 设备初始化函数

```

1  am_pwm_handle_t am_zlg_tim_pwm_init (
2  am_zlg_tim_pwm_dev_t *p_dev,
3  const am_zlg_tim_pwm_devinfo_t *p_devinfo)
4  {
5      amhw_zlg_tim_t *p_hw_tim =NULL;
6      if (p_dev ==NULL || p_devinfo ==NULL) {
7          returnNULL;
8      }
9
10     if (p_devinfo->pfn_plfm_init) {
11         p_devinfo->pfn_plfm_init();
12     }
13
14     p_dev->p_devinfo = p_devinfo;
15     p_hw_tim =
16         (amhw_zlg_tim_t *)p_dev->p_devinfo->tim_regbase;
17     p_dev->pwm_serv.p_funcs=
18         (struct am_pwm_drv_funcs *)&__g_tim_pwm_drv_funcs;
19     p_dev->pwm_serv.p_drv= p_dev;
20
21     __tim_pwm_init(p_hw_tim, p_devinfo->tim_type);
22
23     return &(p_dev->pwm_serv);
24 }

```

可见，设备初始化函数，最后返回的标准服务句柄，即是标准接口层中讲到的抽象设备

类的地址。从实例初始化函数原型可见，实例初始化函数的返回值和设备初始化函数返回值一致，实际用户在使用时只需要关心实例初始化的返回值。但实际上实例初始化的返回值不一定是标准服务句柄。具体的返回值还有两种，一种是全局资源外设对应的实例初始化函数的返回值为 `int` 类型的值，另外一种则是一些较为特殊的设备，功能还没有被标准接口层标准化，其返回值为驱动自定义服务句柄，本节会对这三种返回值类型做详细的说明。

4. 实例初始化返回值类型

从实例初始化的原型可见，关于实例初始化函数的返回值，和设备初始化函数返回值一致。因此，根据驱动设备不同，也可能有三种不同的返回值类型。

- `int` 类型；
- 标准服务句柄 `handle` 类型（`am_{dev_name}_handle_t`，类型由标准接口层定义），如 `am_adc_handle_t`；
- 返回值为驱动自定义服务句柄。

1) 返回值为 `int` 类型

常见的全局资源外设对应的实例初始化函数的返回值类型均为 `int` 类型，例如：CLK，GPIO，DMA 等等，返回值就是一个 `int` 值。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。以中断的设备初始化函数为例，源程序详见程序清单 3.14。

程序清单 3.14 中断的初始化函数

```

1  int am_arm_nvic_init (am_arm_nvic_dev_t      *p_dev,
2                      const am_arm_nvic_devinfo_t *p_devinfo)
3  {
4      int i;
5      if (NULL == p_dev || NULL == p_devinfo) {
6          return -AM_EINVAL;
7      }
8      p_dev->p_devinfo = p_devinfo;
9      __gp_nvic_dev = p_dev;
10     p_dev->valid_flg = AM_TRUE;
11     if ((NULL == p_devinfo->p_isrmap) || (NULL == p_devinfo->p_isrinfo)) {
12         p_dev->valid_flg = AM_FALSE;
13     }
14     if (p_dev->valid_flg) {
15         for (i = 0; i < p_devinfo->input_cnt; i++) {
16             p_devinfo->p_isrmap[i] = __INT_NOT_CONNECTED;
17         }
18         for (i = 0; i < p_devinfo->isrinfo_cnt; i++) {
19             p_devinfo->p_isrinfo[i].pfn_isr = NULL;
20         }

```

```

21     }
22     amhw_arm_nvic_priority_group_set (p_devinfo->group);
23     return AM_OK;
24 }

```

可见在实例初始化中，初始化全局资源外设时，返回值为 `int` 类型，仅仅是一个初始化结果标志(`int` 类型)。

通过实例初始化，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，标准化接口和未标准化接口。

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。以 `PWM` 设备初始化函数为例，其设备初始化函数原型详见程序清单 3.13。

其返回值为设备信息中的抽象设备类的地址，做为标准接口的参数，使得标准接口层函数的相关代码是可跨平台复用的。比如我们使用 `PWM` 标准接口，通过定时器通道对应的引脚输出频率为 `2KHz`，占空比为 `50%` 的 `PWM` 波，使用详情可见程序清单 3.15。

程序清单 3.15 标准 PWM 的使用例程

```

1 void demo_std_timer_pwm_entry (int pwm_chan)
2 {
3     am_pwm_handle_t pwm_handle =NULL;
4     /* 实例初始化 PWM 设备 */
5     pwm_handle = am_zlg116_timer0_pwm_inst_init ();
6     /* 配置 PWM 频率为 1000000000 / 500000 =2KHz */
7     am_pwm_config(pwm_handle, pwm_chan, 500000 /2, 500000);
8     am_pwm_enable(pwm_handle, pwm_chan);
9     AM_FOREVER {
10         ;    /* VOID */
11     }
12 }

```

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。下面我们看一个 `LPC82x` 特有的外设 `SCT`，功能还没有被标准接口层标准化，实例初始化函数如下：

```
am_lpc824_sct_handle_t am_lpc824_sct0_inst_init (void);
```


从函数的声明可见，返回驱动自定义服务句柄的实例初始化函数和返回标准服务句柄的操作类似，不同之处在于，返回自定义服务句柄，直接返回设备实例地址，设备 SCT 的设备初始化部分代码详见程序清单 3.16。

程序清单 3.16 SCT 初始化函数

```

1  am_lpc_sct_handle_t am_lpc_sct_init ( am_lpc_sct_dev_t      *p_dev,
2                                     const am_lpc_sct_devinfo_t *p_devinfo)
3  {
4      uint8_t i;
5      amhw_lpc_sct_t *p_hw_sct = NULL;
6      if ((NULL == p_devinfo) || (NULL == p_devinfo) ) {
7          return NULL;
8      }
9      if (p_devinfo->pfn_plfm_init) {
10         p_devinfo->pfn_plfm_init();
11     }
12     p_hw_sct = (amhw_lpc_sct_t *) (p_devinfo->sct_regbase);
13     p_dev->p_devinfo = p_devinfo;
14     p_dev->evt_stat = 0; /* 所有事件未被使用 */
15     p_dev->valid_flg = AM_TRUE;
16     if ((p_devinfo->p_isrmap != NULL) && (p_devinfo->p_isrinfo != NULL)) {
17         for (i = 0; i < p_devinfo->evt_isr_cnt; i++) {
18             p_devinfo->p_isrmap[i] = __INT_NOT_CONNECTED;
19         }
20         for (i = 0; i < p_devinfo->isrinfo_cnt; i++) {
21             p_devinfo->p_isrinfo[i].pfn_isr = NULL;
22         }
23     }
24     ... /* 配置 SCT 寄存器 */
25     return p_dev;
26 }

```

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。

3.2.3 实例解初始化

每个外设驱动都提供了对应的驱动解初始化函数，以便应用不再使用某个外设时，释放掉相关资源。外设的驱动解初始化函数在驱动头文件 `am_{dev_name}.h` 或者文件 `am_{dev_name}.h` 中声明。详见程序清单 3.17。

程序清单 3.17 设备解初始化函数声明

```

1  /**
2   * 设备去初始化
3   *
4   * \param[in] 无
5   *
6   * \return 无
7   */
8  void am_{dev_name}_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，如下所示：

```
am_{dev_name}_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。用于应用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，完全不用关心驱动设备的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。实例解初始化函数定义在{HWCONFIG}\am_hwconf_{dev_name}.c 或\am_hwconf_{dev_name}.c 文件中，详见程序清单 3.18。

程序清单 3.18 实例解初始化函数

```

1  /** 设备实例解初始化 */
2  void am_{dev_name}_inst_deinit(void)
3  {
4      am_{dev_name}_deinit();
5  }

```

所有实例解初始化函数均无返回值。解初始化后，该外设将不再可用。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。

3.2.4 其它功能性函数

初始化完成后，对于有标准接口的功能，可以直接使用标准接口访问，但部分外设或功能还未标准化，没有标准接口，则应按照常规驱动，再提供一系列功能性接口。例如，DMA 目前还没有标准化，则 DMA 驱动应直接提供 DMA 的功能接口，如下：

```

1  int am_zlg116_dma_chan_start()
2  {
3      // 直接操作寄存器实现。
4  }

```

3.3 硬件层

硬件层用于完成外设最底层的操作，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，或者对效率等有需求时，可以调用硬件层 API。

硬件层等价于传统 SOC 原厂的裸机包。硬件层接口使用 `amhw_/AMHW_ + 芯片名` 作为命名空间，如 `amhw_zlg116`、`AMHW_ZLG116`，具体位置在对应设备的 HW 文件中。下面我们看一个 adc 的硬件功能函数，详见程序清单 3.19。

程序清单 3.19 adc 检查数据标志值

```

1
2  am_bool_t amhw_zlg_adc_data_valid_check (amhw_zlg_adc_t *p_hw_adc)
3  {
4      return (((p_hw_adc->addata >>21) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
5  }

```

内联函数的修饰符，即 `static inline`。具体作用可以查看 C 语言语法中 `static` 和 `inline` 的作用。

`amhw_zlg_adc_t` 为系统控制器寄存器列表，硬件层接口是直接操作寄存器，所以使用硬件层接口的效率相对于标准层接口而言会高很多。

3.3.1 使用硬件层接口

HW 层的接口函数都是以外设寄存器结构体指针为参数（特殊地，系统控制部分功能混杂，默认所有函数直接操作 SYSCON 各个功能，无需再传入相应的外设寄存器结构体指针）。

以 ADC 为例，所有硬件层函数均在 `ametal\soc\{core_type_name}\drivers\include\adc\hw\amhw_{core_type_name}_adc.h` 文件中声明（一些简单的内联函数直接在该文件中定义）。简单列举一个函数，详见程序清单 3.20。

程序清单 3.20 ADC 硬件层操作函数

```

1  /**
2   * ADCfunction disable set
3   *
4   * \param[in] p_hw_adc: The pointer to the structureADCregister
5   * \param[in] flag : see the fiLEDof function enable mask
6   *
7   * \return none
8   */
9
10 void amhw_zlg_adc_ctrl_reg_clr (amhw_zlg_adc_t *p_hw_adc,
11                                uint32_t      flag)
12 {
13     p_hw_adc->adcr &= ~flag;
14 }

```

当然还有很多这一类的函数，读者可自行打开 `ametal\soc\{core_type_name}\drivers\include\adc\hw\amhw_{core_type_name}_adc.h` 文件查看。函数基本都是以 `amhw_{core_type_name}_adc_t *` 类型作为第一个参数

amhw_{core_type_name}_adc_t 类型在 ametal\soc\{core_type_name}\drivers\include\adc\hw\amhw_{core_type_name}_adc.h 文件中定义，用于定义出 ADC 外设的各个寄存器。如程序清单 3.21 所示给出了 ADC 寄存器的定义。

程序清单 3.21 ADC 寄存器结构体定义

```

1  /**
2   * ADC- Register Layout Typedef
3   */
4   typedef struct amhw_zlg_adc{
5       __IO uint32_t addata;           /* 数据寄存器 */
6       __IO uint32_t adcfg;           /* 配置寄存器 */
7       __IO uint32_t adcr;            /* 控制寄存器 */
8       __IO uint32_t adchs;           /* 通道选择寄存器 */
9       __IO uint32_t adcmp;           /* 窗口比较寄存器 */
10      __IO uint32_t adsta;            /* 状态寄存器 */
11      __IO uint32_t addr[9];          /* 数据地址寄存器 */
12  } amhw_zlg_adc_t;

```

该类型的指针已经在 ametal\soc\zlg\{soc_name}\{soc_name}_periph_map.h 文件中定义，应用程序可以直接使用。详见程序清单 3.22。

程序清单 3.22 GPIOA 寄存器结构体定义

```

1  /** GPIO 端口 A 寄存器块指针 */
2  #define ZLG116_ADC ((amhw_zlg_adc_t *)ZLG116_ADC_BASE)

```

其他系列芯片的设备也是如此，对应的寄存器结构体指针会在 ametal\soc\{core_type_name}\{board_name}\{board_name}_periph_map.h 文件中定义，应用程序可以直接使用。

注:其中的 {board_name}_ADC_BASE 是 GPIO 外设寄存器的基地址，在 ametal\soc\{core_type_name}\{board_name}\{board_name}_regbase.h 文件中定义，其他所有外设的基地址均在该文件中定义。

有了 ADC 寄存器结构体指针宏后，就可以直接使用 ADC 硬件层的相关函数了。对于一些特殊的功能，硬件层可能没有提供出相关接口，此时，可以基于各个外设指向寄存器结构体的指针，直接操作寄存器实现，例如，要使能 ADC 中断，可以直接直接设置寄存器的值，详见程序清单 3.23。

程序清单 3.23 直接设置寄存器使能 ADC 中断

```

1  /**
2   * 设置 ADCR 寄存器的 bit0 为 1，使能 ADC 中断
3   */
4   AMHW_{board_name}_ADC->adcr |= 0x01;

```

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

所有外设均在 `ametal\soc\{core_type_name}\{board_name}\{board_name}_periph_map.h` 文件中定义了指向外设寄存器的结构体指针。

3.4 通用接口定义

面向通用接口的编程，使得程序与具体的硬件无关，可以很容易的实现跨平台复用，但其具体如何实现，本节将详细描述。

3.4.1 接口定义

合理的接口应该是易阅读的、职责明确的，下面将以 LED 通用接口为例，从接口的命名、参数和返回值三个方面阐述在 AMetal 中定义接口的一般方法。

1. 接口命名

在 AMetal 中，所有通用接口均以“am_”开头，紧接着是操作对象的名字，对于 LED 控制接口来说，所有接口应该以“am_led_”为前缀。当接口的前缀定义好之后，需要考虑定义哪些功能性接口，然后根据功能完善接口名。

对于 LED 来说，核心的操作是控制 LED 的状态，点亮或熄灭 LED，因此需要提供一个设置（set）LED 状态的函数，比如：

- `am_led_set`

显然，通过该接口可以设置 LED 的状态，为了区分是点亮还是熄灭 LED，需要通过一个参数指定具体的操作。

在大多数应用场合中，可能需要频繁地开灯和关灯操作，每次开关灯都需要通过传递参数给 `am_led_set()` 接口实现开灯和关灯，这样做会非常繁琐。因此可以为常用的开灯和关灯操作定义专用的接口，也就不再需要额外参数区分具体的操作。比如，使用 `on` 和 `off` 分别表示开灯和关灯，则定义开灯和关灯的接口名为：

- `am_led_on`

- `am_led_off`

在一些特殊的应用场合中，比如，LED 闪烁，其可能并不关心具体的操作是开灯还是关灯，它仅仅需要 LED 的状态发生翻转。此时，可以定义一个用于翻转（toggle）LED 状态的接口，其接口名为：

- `am_led_toggle`

2. 接口参数

在 AMetal 中，通用接口的第一个参数表示要操作的具体对象。显然，一个系统可能有多个 LED，为了确定操作的 LED，最简单的方法是为每个 LED 分配一个唯一编号，即 ID 号，然后通过 ID 号确定需要操作的 LED。ID 号是一个从 0 开始的整数，其类型为 `int`，基于此，所有接口的第一个参数定义为 `int` 类型的 `led_id`。

对于 `am_led_set` 接口来说，除使用 `led_id` 确定需要控制的 LED 外，还需要使用一个参数区分是点亮 LED 还是熄灭 LED。由于是二选一的操作，因此该参数的类型使用布尔类型：`am_bool_t`。当值为真（`AM_TRUE`）时，则点亮 LED；当值为假（`AM_FALSE`）时，则熄灭 LED。基于此，包含参数的 `am_led_set` 接口函数原型为（还未定义返回值）：

```
am_led_set(int led_id, am_bool_t state);
```

对于 `am_led_on`、`am_led_off` 和 `am_led_toggle` 接口来说，它们的职责单一，仅仅需要指定控制的 LED，即可完成点亮、熄灭或翻转操作，无需额外的其它参数。因此对于这类接口，参数仅仅只需要 `led_id`。其函数原型如下：

```
am_led_on(int led_id);
am_led_off(int led_id);
am_led_toggle(int led_id);
```

实际上,在 AMetal 通用接口的第一个参数中,除使用 ID 号表示操作的具体对象外,还可能直接使用指向具体对象的指针,或者表示具体对象的一个句柄来表示,它们的作用在本质上是完全一样的。

3. 返回值

对于用户来说,调用通用接口后,应该可以获取本次执行的结果,成功还是失败,或一些其它的有用信息。比如,当调用接口时,如果指定的 led_id 超过有效范围时,由于没有与 led_id 对应的 LED 设备,操作必定会失败,此时必须返回错误,告知用户操作失败,且失败的原因是 led_id 不在有效范围内,无与之对应的 LED 设备。

在 AMetal 中,通过返回值返回接口执行的结果,其类型为 int,返回值的含义为:若返回值为 AM_OK,则表示操作成功;若返回值为负数,则表示操作失败,失败原因可根据返回值,查找 am_errno.h 文件中定义的宏,根据宏的含义确定失败的原因;若返回值为正数,其含义与具体接口相关,由具体接口定义,无特殊说明时,表明不会返回正数。

AM_OK 在 am_common.h 文件中定义,其定义如下:

```
#define AM_OK 0
```

3.4.2 接口实现

1. 通用的 LED 设备类

通用 LED 设备初始化函数 am_led_gpio_inst_init 位于 user_config/am_hwconf_led_gpio.c 中:

```
int am_led_gpio_inst_init(void)
{
    return am_led_gpio_init(&__g_led_gpio, &__g_led_gpio_info);
}
```

其中 __g_led_gpio 为 LED 实例, __g_led_gpio_info 为 LED 实例信息。当我们需要更改 LED 的引脚信息时,只需更改实例信息便可。实例信息结构体定义如下:

```
typedef struct am_led_gpio_info {
    am_led_servinfo_t    serv_info;
    const int             *p_pins;
    am_bool_t             active_low;
} am_led_gpio_info_t;
```

- 结构体 am_led_servinfo_t 仅包含起始 ID 和结束 ID,一般起始 ID 为 0,结束 ID 为总 LED 数减 1;
- 指针 p_pins 指向的是 LED 使用引脚的数组,数组中 LED 引脚的位置对应其 ID 号,例如某 LED 的引脚为 PIOB_3,它在 p_pins 指向的数组中排列第三,由于起始 ID 为 0,所以该 LED 对应的 ID 为 2;
- active_low 为 bool 类型,含义为 LED 是否是低电平点亮。也就是说 LED 低有效,

active_low 为真；LED 高有效，active_low 为假；

由于 AMetal 在初始化的时候会调用 LED 初始化函数，所以我们要更改 LED 引脚或数量时，可直接修改 am_hwconf_led_gpio.c 中的内容。例如添加一个引脚为 PIOB_3 的 LED，详见程序清单 3.24，在第 7 行定义的数组 __g_led_pins 后添加 PIOB_3，并把第 12 行的结束编号改为 2 即可。

程序清单 3.24 通用 LED 设备初始化

```

1  #include "ametal.h"
2  #include "am_led_gpio.h"
3  #include "zlg116_pin.h"
4  #include "am_input.h"
5  /* 定义 GPIO LED 实例 */
6  static am_led_gpio_dev_t __g_led_gpio;
7  static const int __g_led_pins[] = {PIOB_1, PIOB_2};
8  /* 定义 GPIO LED 实例信息 */
9  static const am_led_gpio_info_t __g_led_gpio_info = {
10     {
11         0,          /* 起始编号 0 */
12         1          /* 结束编号 1，共计 2 个 LED */
13     },
14     __g_led_pins,
15     AM_TRUE
16 };
17 int am_led_gpio_inst_init(void)
18 {
19     return am_led_gpio_init(&__g_led_gpio, &__g_led_gpio_info);
20 }

```

2. 抽象的 LED 设备类

还有一些 LED 不是由 GPIO 直接控制，而是通过 I2C 或 SPI 等通讯协议或某种方式，由另一个控制芯片来控制 LED。这个时候，可以通过将其抽象化，继续使用通用接口函数控制它们。添加抽象 LED 设备的函数 am_led_dev_add 位于 am_led_dev.h 中，具体内容为：

```

int am_led_dev_add(am_led_dev_t      *p_dev,
                  const am_led_servinfo_t *p_info,
                  const am_led_drv_funcs_t *p_funcs,
                  void                *p_cookie);

```

可以看到，相比于通用接口，抽象的 LED 设备新增了一个指向 LED 设备的驱动函数的 p_funcs 指针以及一个用于指向该设备的驱动函数的参数 p_cookie 指针。也就是说，我们要增加一个要通过某种方式间接控制的 LED 设备时，只要增加这两个参数，并调用 am_led_dev_add 函数，就能向控制通用 LED 设备那样控制 LED。

例如，将 MiniPort-LED 和 MiniPort-595 联合使用，增加了 8 个 LED；调用的 LED 初始化函数为：

```
static const am_led_drv_funcs_t __g_led_hc595_drv_funcs = {
    __led_hc595_set,
    __led_hc595_toggle
};

int am_led_hc595_init (am_led_hc595_dev_t      *p_dev,
                      const am_led_hc595_info_t *p_info,
                      am_hc595_handle_t        handle)
{
    if ((p_dev == NULL) || (p_info == NULL)) {
        return -AM_EINVAL;
    }
    p_dev->p_info = p_info;
    if (p_info->active_low) {
        memset(p_info->p_buf, 0xFF, p_info->hc595_num);
    } else {
        memset(p_info->p_buf, 0x00, p_info->hc595_num);
    }
    am_hc595_send(handle, p_info->p_buf, p_info->hc595_num);
    return am_led_dev_add(&p_dev->isa,
                          &p_info->serv_info,
                          &__g_led_hc595_drv_funcs,
                          p_dev);
}
```

__g_led_hc595_drv_funcs 中包含了新增 LED 的操作函数 __led_hc595_set 和 __led_hc595_toggle，然后 MiniPort-595 实例 am_hc595_handle_t 中又包含了这些函数的指针：

```
typedef struct am_hc595_dev {
    const struct am_hc595_drv_funcs *p_funcs;    /**<\brief 设备驱动函数 */
    void *p_cookie;    /**<\brief 设备驱动函数参数 */
} am_hc595_dev_t;
```

只需调用 MiniPort-595LED 初始化函数 am_led_hc595_init，我们就能像操作通用的 LED 设备那样操作抽象的 LED 设备，详见程序清单 3.25。

程序清单 3.25 抽象 LED 设备初始化

```
1  #include "ametal.h"
2  #include "am_led_hc595.h"
3  #include "am_led_gpio.h"
```



```

4  #include "am_hc595_gpio.h"
5  #include "am_hc595_spi.h"
6  #include "am_zlg116_inst_init.h"
7  #include "zlg116_pin.h"
8
9  static am_led_hc595_dev_t __g_miniport_led_595;
10 static uint8_t __g_miniport_led_595_buf[1];
11
12 static const am_led_hc595_info_t __g_miniport_led_595_info = {
13     {
14         2,                /* 起始编号 2 */
15         9                /* 结束编号 9, 共计 8 个 LED */
16     },
17     1,
18     __g_miniport_led_595_buf,
19     AM_TRUE
20 };
21 int am_miniport_led_595_inst_init(void)
22 {
23     return am_led_hc595_init( &__g_miniport_led_595,
24                               &__g_miniport_led_595_info,
25                               am_miniport_595_inst_init());
26 }

```

3.5 设备的典型配置及使用

每一个 MCU 上都包含了众多硬件外设资源，只要 AMetal 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_* 开头的.c 文件完成的，下文主要介绍 AMetal 片外设资源的结构。

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

不同评估板的硬件外设资源所对应的配置文件可查看 ametal\documents\{board_name}\{board_name} 用户手册.pdf 中的硬件外设资源对应的配置文件表，硬件外设资源的配置文件表详见表 3.2。

表 3.2 硬件外设及对应的配置文件

序号	外设	配置文件
1	INT(中断)	am_hwconf_arm_nvic.c

2	ADC	am_hwconf_{soc_name}_adc.c
3	时钟部分 (CLK)	am_hwconf_{soc_name}_clk.c

续上表

序号	外设	配置文件
4	DMA	am_hwconf_{soc_name}_dma.c
5	GPIO	am_hwconf_{soc_name}_gpio.c
6	I2C 控制器	am_hwconf_{soc_name}_i2c.c
7	I2C 从机控制器	am_hwconf_{soc_name}_i2c_slv.c
8	电源管理	am_hwconf_{soc_name}_pwr.c
9	SPI(DMA 传输方式)	am_hwconf_{soc_name}_spi_dma.c
10	SPI(中断方式)	am_hwconf_{soc_name}_spi_int.c
11	标准定时器的捕获功能	am_hwconf_{soc_name}_tim_cap.c
12	标准定时器的 PWM 功能	am_hwconf_{soc_name}_tim_pwm.c
13	标准定时器的定时功能	am_hwconf_{soc_name}_tim_timing.c
14	UART	am_hwconf_{soc_name}_uart.c
15	窗口看门狗	am_hwconf_{soc_name}_wwdg.c
16	独立看门狗	am_hwconf_{soc_name}_iwdg.c
17	按键	am_hwconf_key_gpio.c
18	LED	am_hwconf_led_gpio.c
19	蜂鸣器	am_hwconf_buzzer.c
20	温度传感器 (LM75)	am_hwconf_lm75.c
21	调试串口	am_hwconf_debug_uart.c
22	系统滴答和软件定时器	am_hwconf_system_tick_softimer.c

这里只是一部分的硬件外设，每个硬件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，即使用户有一些特殊的需求，也只需要修改当中极少部分对应的参数即可。

注：从配置文件我们可以看到，文件名分为 am_hwconf* 开头的.c 文件，和 am_hwconf_{soc_name} 开头的.c 文件，一个带有 mcu 名，如 am_hwconf_ZLG116_iwdg.c 这个文件，这一类属于 soc 片上外设资源，而不带 mcu 名为板级外设资源，用户如果感兴趣可以通过查看 mcu 芯片手册，不了解也没关系都把这一类资源统一为硬件外设资源处理即可。

3.5.1 配置文件

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。上文有详述整个配置的流程和细节。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 ametalboard{board_name}\project_template\user_config\amboard.c 中自动完成初始化。以 LED 为例，初始化代码详见程序清单 3.26。

程序清单 3.26 LED 实例初始化函数调用

```

1  /**
2   * 板级初始化
3   */
4   void am_board_init(void)
5   {
6       // .....
7       #if (AM_CFG_LED_ENABLE==1)
8           am_led_gpio_inst_init();
9       #endif /* (AM_CFG_LED_ENABLE==1) */
10      // .....
11  }
```

程序清单 3.27 是否使能 GPIO 的相关宏

```

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

所以一些设备我们只需要找到他对应的宏，进行修改，使能或禁能它，如果没有在 user_configam_board.c 文件中初始化，当我们使用到此外设资源时，手动调用即可，如程序清单 3.27。

3.5.2 典型配置

1. LED 配置

板上通常有两个 LED 灯，对应接两个 I/O 口，可以通过查看电路原理图，或者查看 LED 相关信息定义在 ametal\board\{board_name}\project_template\user_config\am_hwconf_usrcfg\am_hwconf_led_gpio.c 文件，详见程序清单 3.28。

程序清单 3.28 LED 相关配置信息

```

1  /** 定义 LED 相关的 GPIO 管脚信息 */
2   static const int __g_led_pins[] = {PIO{1}, PIO{2}};
3   /** 定义 GPIOLED 实例信息 */
4   static const am_led_gpio_info_t __g_led_gpio_info = {
5   {
6       0, /* 起始编号 0 */
7       AM_NELEMENTS(__g_led_pins)-1/* 结束编号 1, 共计 2 个 LED*/
8   },
9   __g_led_pins,
10  AM_TRUE /* 低电平点亮 */
11  };
```

注:PIO{1}, PIO{2}，分别为 LED 对应使用的 IO 口。

其中，am_led_gpio_info_t 类型在 ametal\components\drivers\include\am_led_gpio.h 文件中定义，详见程序清单 3.29。

程序清单 3.29 LED 引脚配置信息类型定义

```

1  typedef struct am_led_gpio_info {
2      /** LED 基础服务信息，包含起始编号和结束编号 */
3      am_led_servinfo_t serv_info;
4      /** 使用的 GPIO 引脚，引脚数目应该为（结束编号 - 起始编号 + 1） */
5      const int *p_pins;
6      /** LED 是否是低电平点亮 */
7      am_bool_t active_low;
8
9  }am_led_gpio_info_t;

```

其中，serv_info 为 LED 的基础服务信息，包含 LED 的起始编号和结束编号，p_pins 指向存放 LED 引脚的数组首地址，选择的管脚在 {soc_name}_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 分别对应的两个 I/O 口，均为低电平点亮。如需添加更多的 LED，只需在该配置信息数组中继续添加即可。

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

注：由于 LED 使用了两个 io 口，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。在使用时请查看电路，或跳线帽功能选择表，是否需要短接对应的跳线帽。

2. 蜂鸣器配置

板载蜂鸣器我们以无源蜂鸣器为例，因为使用多为无源蜂鸣器，需要使用 PWM 驱动才能实现发声。可以通过 {PROJECT}\user_config\am_hwconf_usrcfg\am_hwconf_buzzer_PWM.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 3.3。

表 3.3 蜂鸣器相关宏

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

了一个通道 PWM 功能，若应用程序需要使用此 PWM，建议通过使能/禁能宏来禁止蜂鸣器的使用，以免冲突。

3. 按键

一般有两个板载按键 KEY/RES 和 RST，使用时，需要其中 RST 为复位按键，可供使用的按键只有 KEY/RES。KEY 相关信息定义在 ametal\board\{board_name}\project_template\user_config\am_hwconf_usrcfg\am_hwconf_key_gpio.c 文件中，详见程序清单 3.30。

程序清单 3.30 KEY 相关配置信息

```

1  static const int __g_key_pins[] = {PIO{1}};
2  static const int __g_key_codes[] = {KEY_KP0};
3  /* 定义 GPIO 按键实例信息 */
4  static const am_key_gpio_info_t __g_key_gpio_info = {
5      __g_key_pins,
6      __g_key_codes,
7      AM_NELEMENTS(__g_key_pins),
8      AM_TRUE,
9      10
10 };

```

其中，KEY_KP0 为默认按键编号；AM_NELEMENTS 是计算按键个数的宏函数；am_key_gpio_info_t 类型在 ametal/components/drivers/include/am_key_gpio.h 文件中定义，详见程序清单 3.31。

程序清单 3.31 KEY 配置信息类型定义

```

1  /**
2   * 按键信息
3   */
4  typedef struct am_key_gpio_info {
5      const int *p_pins;          /* 使用的引脚号 */
6      const int *p_codes;        /* 各个按键对应的编码（上报） */
7      int pin_num;               /* 按键数目 */
8      am_bool_t active_low;      /* 是否低电平激活（按下为低电平） */
9      int scan_interval_ms;      /* 按键扫描时间间隔，一般 10ms */
10 } 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 {board_name}_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 对应的 I/O 口，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：由于 KEY/RES 使用了 io 口，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。在使用时请查看电路，可用跳线帽功能选择，是否需要短接对应的跳线帽。

4. 调试串口配置

AMetal 一般具有多个串口，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 3.4。

表 3.4 调试串口相关宏

宏名	含义
__DEBUG_UART	串口号, 1-UART1, 2-UART2
__DEBUG_BAUDRATE	使用的波特率, 默认 115200

注：
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, 详见外设部分的相关内容。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. 系统滴答和软件定时器配置

系统滴答需要定时器为其提供一个周期性的定时中断, 不同的核心板默认使用的定时器可能不同, 下面为不同核心板使用的默认定时器和通道详见表 3.5。

表 3.5 系统滴答定时器和对应定时器

核心板	默认定时器和通道
zlg116	TIM14 通道 0
lpc824	MRT 通道 0

其配置还需要使用 {PROJECT}\user_config\am_hwconf_usrcfg\am_hwconf_system_tick_softimer.c 文件中的 SYSTEM_TICK_RATE 宏来设置系统滴

答的频率, 默认为 1KHz。详细定义见程序清单 3.32。

程序清单 3.32 系统 TICK 频率配置

```

1  /**
2   * 设置系统滴答的频率, 默认 1KHz
3   *
4   * 系统滴答的使用详见 am_system.h
5   */
6  #define __SYSTEM_TICK_RATE 1000

```

软件定时器基于系统滴答实现。它的配置也需要使用 {PROJECT}\user_config\am_hwconf_usrcfg\am_hwconf_system_tick_softimer.c 文件中的 __SYSTEM_TICK_RATE 宏来设置运行频率, 默认 1KHz。详细定义见程序清单 3.32。

注解: 使用软件定时器时必须开启系统滴答。

6. 温度传感器 LM75

标准评估板都自带一个 LM75B 测温芯片, 具体需查看片上外设资源, 或电路原理图, 如 AM116, 和 AM824 等都有自带, 使用 LM75 传感器需要配置 {PROJECT}\user_config\am_hwconf_usrcfg\am_hwconf_lm75.c 文件中 LM75 的实例信息 __g_temp_lm75_info, __g_temp_lm75_info 存放的是 I2C 从机地址, 详细定义见程序清单 3.33。

程序清单 3.33 LM75 从机地址配置

```

1  /* 定义 LM75 实例信息 */

```

```

2 static const am__temp_lm75_info_t __g_temp_lm75_info = {
3     0x48
4 };

```

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。

7. 典型外设初始化函数表

常见的板级资源对应的设备实例初始化函数的原型详见表 3.6。

表 3.6 硬件外设及对应的配置文件

序号	板级资源	实例初始化函数原型
1	按键	<code>int am_key_gpio_inst_init (void);</code>
2	LED	<code>int am_led_gpio_inst_init (void);</code>
3	蜂鸣器	<code>am_pwm_handle_t am_buzzer_inst_init (void);</code>
4	温度传感器 (LM75)	<code>am_temp_handle_t am_temp_lm75_inst_init (void);</code>

续上表

序号	板级资源	实例初始化函数原型
5	调试串口	<code>am_uart_handle_t am_debug_uart_inst_init (void);</code>
6	系统滴答	<code>am_timer_handle_t am_system_tick_inst_init (void);</code>
7	系统滴答和软件定时器	<code>am_timer_handle_t am_system_tick_softimer_inst_init (void);</code>

注：不同的板级可能有一些细微不同。

3.5.3 外设资源使用方法

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。

1. 使用 AMetal 提供的驱动

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

1) 初始化

无论何种外设，在使用前均需初始化。所有外设的初始化操作均只需调用用户配置文件中提供的设备实例初始化函数即可。所有外设的实例初始化函数均在 `{PROJECT}\user_config\am_{board_name}_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_{board_name}_inst_init.h` 头文件。MCU 对应的各个外设的设备实例初始化函数可查看 `ametal\documents\《{board_name}用户手册.pdf》`。

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，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。

● 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 3.7。

表 3.7 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_{board_name}_clk_inst_init(void)
2	DMA	int am_{board_name}_dma_inst_init(void)
3	GPIO	int am_{board_name}_gpio_inst_init(void)
4	INT	int am_{board_name}_nvic_inst_init()

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

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

接口已标准化，如 GPIO 提供了标准接口，在 ametal\interface\am_gpio.h 文件中声明。则可以查看相关接口说明和示例，以使用 GPIO，简单示例详见程序清单 3.34。

程序清单 3.34 GPIO 标准接口使用范例

```

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

```

这两句则是把引脚号为 PIOA_10 的 I/O 口配置为输出模式，后面接着输出高电平。

接口原型及详细的使用方法请参考 ametal\documents\《ametal API 参考手册.chm》或者 ametal\interface\am_gpio.h 文件。接口未标准化，则相关接口由驱动头文件自行提供，例如 DMA，相关接口在 ametal\soc{core_type_name}\drivers\include\dma\am_dma.h 文件中声明。

注：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。

● 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，详细介绍可以参见 ametal\documents\《AMetal 中各 MCU 片上外设配置文件表.pdf》，其相关常用外设实例初始化函数详见表 3.8。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 3.8 返回值为标准服务句柄的实例初始化函数

序号	外设	实例初始化函数原型
1	ADC	am_{board_name}_adc_inst_init (void)
2	I2C1	am_{board_name}_i2c1_inst_init (void)
3	SPI1_INT	am_{board_name}_spi1_int_inst_init (void)
4	SPI1_DMA	am_{board_name}_spi1_dma_inst_init (void)
5	Timer1_CAP	am_{board_name}_tim1_cap_inst_init (void)
6	Timer2_CAP	am_{board_name}_tim2_cap_inst_init (void)
7	Timer1_PWM	am_{board_name}_tim1_pwm_inst_init (void)
8	Timer2_PWM	am_{board_name}_tim2_pwm_inst_init (void)

9	Timer1_timing	am_{board_name}_tim1_timing_inst_init(void)
10	Timer2_timing	am_{board_name}_tim2_timing_inst_init(void)
11	PWR	am_{board_name}_pwr_inst_init(void)
12	UART1	am_{board_name}_uart1_inst_init(void)
13	UART2	am_{board_name}_uart2_inst_init(void)
14	WWDT	am_{board_name}_wwdg_inst_init(void)
15	IWDT	am_{board_name}_iwdg_inst_init(void)
16	蜂鸣器	am_pwm_handle_t am_buzzer_inst_init(void);
17	温度传感器 (LM75)	am_temp_handle_t am_temp_lm75_inst_init(void);
18	调试串口	am_uart_handle_t am_debug_uart_inst_init(void);
19	系统滴答	am_timer_handle_t am_system_tick_inst_init(void);
20	系统滴答和软件定时器	am_timer_handle_t am_system_tick_softtimer_inst_init(void);

可见返回值为标准服务句柄的设备相对比较多，这里就没有列出所有的实例初始化函数，用户需要时可以查对应评估板的用户手册。

对于这些外设，后续可以利用返回的 handle 来使用相应的标准接口层函数。使用标准接口层函数的相关代码是可跨平台复用的。例如，ADC 设备的实例初始化函数的返回值类型为 am_adc_handle_t，为了方便后续使用，可以定义一个变量保存下该返回值，后续就可以使用该 handle 完成电压的采集了，详见程序清单 3.35。

程序清单 3.35 ADC 简单操作示例

```

1  #include "ametal.h"
2  #include "am_adc.h"
3  #include "am_{board_name}_inst_init.h"
4  uint32_t g_adc_val_buf[200];
5  am_adc_handle_t g_adc_handle;
6  int am_main(void) {
7      g_adc_handle = am_{board_name}_adc_inst_init();
8      /* 读取 ADC 转换的电压值 */
9      am_adc_read_mv(g_adc_handle, 0, g_adc_val_buf, 200);
10     /* ..... */
11     while(1);
12 }
```

● 返回值为驱动自定义服务句柄

有些外设实例初始化函数后返回的是驱动自定义服务句柄，不过只有少数特殊的外设实例初始化函数，返回的是自定义服务句柄。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。

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。例如 ZLG116 的 TIM0 相关驱动函数在 \ametal\soc\common\drivers\include\am_zlg_timer.h 文件中声明。相关接口及使用方法可以参考该文件或 \ametal\documents\《ametal API 参考手册.chm》。handle 的使用示例详见程序清单 3.36。

程序清单 3.36 timer handle 简单操作示例

```

1  #include "ametal.h
2  #include "am_zlg116_inst_init.h"
3  am_zlg116_timer_handle_t g_zlg116_timer_handle;
4  int am_main (void)
5  {
6      g_zlg116_timer_handle = am_zlg116_timer0_inst_init();
7
8      /* 设置 TIM 状态为 0 */
9      am_zlg_timer_state_set(g_zlg116_timer0_handle, 0);
10
11     // .....
12     while(1);
13 }

```

3) 解初始化的使用

外设使用完毕后，应该调用相应设备配置文件提供的实例解初始化函数，以释放相关资源。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。所有外设的实例解初始化函数均在 {PROJECT}\user_config\am_{board_name}_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_{board_name}_inst_init.h 头文件。对应板子各个外设对应的设备实例解初始化函数的原型可以参见 ametal\documents\《AMetal 中各 MCU 片上外设资源表.pdf》进行查询，这里列出部分外设对应的设备实例解初始化函数的原型详见表 3.9。

表 3.9 返回值为标准服务句柄的实例解初始化函数

序号	外设	实例解初始化函数原型
1	ADC	void am_{board_name}_adc_inst_deinit (void)
2	DMA	void am_{board_name}_dma_inst_deinit (void)
3	GPIO	void am_{board_name}_gpio_inst_deinit (void)
4	I2C1	void am_{board_name}_i2c1_inst_deinit (void)
5	SPI1_INT	void am_{board_name}_spi1_int_inst_deinit (void)
6	SPI1_DMA	void am_{board_name}_spi1_dma_inst_deinit (void)
7	Timer1_CAP	void am_{board_name}_tim1_cap_inst_deinit (void)
8	Timer2_CAP	void am_{board_name}_tim2_cap_inst_deinit (void)
9	Timer1_PWM	void am_{board_name}_tim1_pwm_inst_deinit (void)
10	Timer2_PWM	void am_{board_name}_tim2_pwm_inst_deinit (void)
11	INT	void am_{board_name}_nvic_inst_deinit (void)
12	PWR	void am_{board_name}_pwr_inst_deinit (void)
13	UART1	void am_{board_name}_uart1_inst_deinit (void)
14	UART2	void am_{board_name}_uart2_inst_deinit (void)
15	WWDT	void am_{board_name}_wwdg_inst_deinit (void)

16	IWDWT	void am_{board_name}_iwdg_inst_init (void)

此处未列出所有的实例解初始化函数，用户如有需要可以查看对应评估板的用户手册。

注：时钟部分不能被解初始化。

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 **GPIO** 为例，所有硬件层函数均在 `ametal\soc\{core_type_name}\drivers\include\gpio\hw\amhw_{core_type_name}_gpio.h` 文件中声明（一些简单的内联函数直接在该文件中定义）。简单列举一个函数，设置 **GPIO** 输出高电平，详见程序清单 3.37。

程序清单 3.37 设置 GPIO 引脚输出高电平硬件接口

```

1  /**
2   * 设置 GPIO 引脚输出高电平
3   *
4   * \param[in] p_hw_gpio: 指向 GPIO 寄存器块的指针
5   * \param[in] pin: 引脚编号，值为 PIO*
6   *
7   * \return 无.
8   *
9   * \note 该功能配置输入引脚的时候为使能上拉电阻并失能下拉电阻
10  */
11
12 void amhw_zlg_gpio_pin_out_high (amhw_zlg_gpio_t *p_hw_gpio, int pin)
13 {
14     amhw_zlg_gpio_t *p_gpio_addr = (amhw_zlg_gpio_t *)((int)p_hw_gpio \
15     + (pin >> 4) * 0x400);
16
17     /* 相应引脚输出高电平 */
18     p_gpio_addr->bsrr = (1UL << (pin & 0x0f));
19 }
```

3.6 中间件

中间件是一些通用的组件，与具体芯片无关，独立开发，例如：**Modbus**、**LoRa WAN**、**LoRa NET** 协议栈。用户使用 **AMetal** 时，可以直接使用这些成熟的协议栈，快速实现应用，

本文主要针对 MCU 的外设适配，对中间件的开发不做过多描述。

3.7 基础组件

基础组件是一些基础的软件工具，便于在应用程序中使用，例如：任务队列、链表、环形缓冲区、软件定时器等。这些实用的基础工具可以为应用程序提供很大的便利，其应用可参考《AMetal 从入门到实践》一书。

3.8 通用例程文件

AMatel 中 examples 文件下的 demo 分为四类，分别放到 board、components、soc、std 这四个文件中的 demo 源程序是给用户的一个使用范例，同时头文件包含了详尽的注释说明：操作步骤，demo 现象等，用户在使用相关的内容时可以先查看对应的 demo 文件，这样用户可以很快的上手使用相应的外设资源。

- board 可以在具体硬件板上运行的 demo，即板级例程；
- components 组件下相关驱动的 demo 源程序，组件例程；
- soc 芯片驱动的 demo 源程序，硬件层例程；
- std 驱动层例程，调用驱动层标准接口实现，标准例程。

除 board、soc 外，其它文件夹中（components、std）的 demo 是通用的，不与具体硬件绑定，可以应用在多个硬件板中。其正常运行可能需要一定的条件，这些条件通过入口函数的参数传入。

3.8.1 板级例程

板级例程主要存放在 board 文件夹中，它们调用驱动层和硬件层的例程，控制评估板的各个硬件。这些例程与板级硬件密切相关，不同的评估板，其硬件有所不同，例程文件也不同，因此，按照不同的评估板分成了不同的文件夹，如 am116_core 文件夹内存放的是 am116_core 这个评估板对应的例程文件。根据其实现过程是通过调用驱动层例程还是调用硬件层例程，这些例程又被分为了两类，通过其文件名中的“hw”或“std”来区分，文件名中含有“hw”的，表明其通过调用硬件层例程实现；文件名中含义“std”的，表明其通过调用驱动层例程来实现。

其中，“demo_am116_core_hw_adc_int.c”文件中存放的是 ADC 中断例程，实现电压采样并用串口输出，通过调用 HW 层接口实现。“demo_am116_core_std_adc_ntc.c”利用板子的热敏电阻（NTC），推算出温度，通过标准接口实现。ADC 中断例程的入口程序详见程序清单 3.38。

程序清单 3.38 ADC 中断例程

```

1  #include "ametal.h"
2  #include "am_clk.h"
3  #include "am_gpio.h"
4  #include "am_vdebug.h"
5  #include "am_zlg116.h"
6  #include "demo_zlg_entries.h"
7  /**
8   * \brief 例程入口
9   */
10 void demo_am116_core_hw_adc_int_entry (void)

```

```

11 {
12  AM_DBG_INFO("demo am116_core hw adc int!\r\n");
13  /* 运行硬件层例程前的环境设置，配置引脚，使能时钟等 */
14  am_gpio_pin_cfg(PIOA_1, PIOA_1_AIN);
15  am_clk_enable(CLK_ADC1);
16  demo_zlg_hw_adc_int_entry(    ZLG116_ADC,
17                               INUM_ADC_COMP,
18                               AMHW_ZLG_ADC_CHANNEL1,
19                               3300);
20 }

```

除了这些外设例程外，am116_core 文件夹中还有“demo_am116_core_entries.h”，包含了 am116_core 文件夹中的所有例程入口函数的声明，在使用到这些例程时，只需包含该头文件既可以。例如上边所使用的 adc 中断例程，其定义详见程序清单 3.39。

程序清单 3.39 ADC 中断声明

```

1 /**
2  * \brief ADC INT 例程，通过 HW 层接口实现
3  */
1  void demo_am116_core_hw_adc_int_entry (void);

```

3.8.2 组件例程

组件例程主要存放在 components 文件夹中，组件例程包含了一些常用芯片的例程，如常见的 flash 芯片“EP24Cxx”和“MX25xx”等。还有一些服务组件例程，如闪存转换层例程“demo_ftl.c”等。下面为 EEPROM 的例程入口，详见程序清单 3.40。

程序清单 3.40 EEPROM 例程入口

```

1  #include "ametal.h"
2  #include "am_delay.h"
3  #include "am_vdebug.h"
4  #include "am_ep24cxx.h"
5  #define __BUF_SIZE 16          /**< \brief 缓冲区大小 */
6  /**
7  * \brief 例程入口
8  */
9  void demo_ep24cxx_entry (am_ep24cxx_handle_t ep24cxx_handle, int32_t test_lenth)
10 {
11     uint8_t i;
12     uint8_t wr_buf[__BUF_SIZE] = {0}; /* 写数据缓存定义 */
13     uint8_t rd_buf[__BUF_SIZE] = {0}; /* 读数据缓存定义 */
14     int ret;
15     if (__BUF_SIZE < test_lenth) {
16         test_lenth = __BUF_SIZE;
17     }
18     /* 填充发送缓冲区 */

```

```

19     for (i = 0; i < test_lenth; i++) {
20         wr_buf[i] = (i + 6);
21     }
22     /* 写数据 */
23     ret = am_ep24cxx_write(ep24cxx_handle,
24                           0x00,
25                           &wr_buf[0],
26                           test_lenth);
27     if (ret != AM_OK) {
28         AM_DBG_INFO("am_ep24cxx_write error(id: %d).\r\n", ret);
29         return;
30     }
31     am_mdelay(5);
32     /* 读数据 */
33     ret = am_ep24cxx_read(ep24cxx_handle,
34                          0x00,
35                          &rd_buf[0],
36                          test_lenth);
37     if (ret != AM_OK) {
38         AM_DBG_INFO("am_ep24cxx_read error(id: %d).\r\n", ret);
39         return;
40     }
41     /* 校验写入和读取的数据是否一致 */
42     for (i = 0; i < test_lenth; i++) {
43         AM_DBG_INFO("Read EEPROM the %2dth data is 0x%02x\r\n", i, rd_buf[i]);
44         /* 校验失败 */
45         if (wr_buf[i] != rd_buf[i]) {
46             AM_DBG_INFO("verify failed at index %d.\r\n", i);
47             break;
48         }
49     }
50     if (test_lenth == i) {
51         AM_DBG_INFO("verify success!\r\n");
52     }
53     AM_FOREVER {
54         ; /* VOID */
55     }
56 }

```

从上面的例程中我们可以看到，主机写数据到 EEPROM，然后主机从 EEPROM 读取数据，并通过串口打印处理；最后串口打印出测试结果。在组件例程中有“demo_components_entries.h”文件用于存放这些例程入口函数的声明。在使用到这些例程时，只需包含该头文件既可以。如 EEPROM 的例程入口声明，详见程序清单 3.41。

程序清单 3.41 EEPROM 例程入口函数声明

```

1  /**
2   * \brief EP24CXX 器件例程
3   *
4   * \param[in] ep24cxx_handle EP24CXX 标准服务句柄
5   * \param[in] test_lenth 测试字节数
6   *
7   * \return 无
8   */
9   void demo_ep24cxx_entry (am_ep24cxx_handle_t ep24cxx_handle,
10    uint32_t test_lenth);

```

3.8.3 硬件层例程

硬件层例程放在 soc 文件夹中，这些例程通过调用硬件层函数实现。soc 有不同系列芯片，如 zlg、nxp 等，不同系列的 soc 对应不同的相关硬件层例程（以 ZLG 系列为例）。如“adc”、“clk”等例程，这些例程因缺少板级环境支持，一般无法直接运行，主要是通过被板级例程文件夹中的例程调用实现。下面是一个硬件层例程中串口打印指定的总线频率的例子，详见程序清单 3.42。

程序清单 3.42 硬件层 CLK 例程

```

1  #include "ametal.h"
2  #include "am_clk.h"
3  #include "am_vdebug.h"
4  /**
5   * \brief 例程入口
6   */
7   void demo_zlg_hw_clk_entry (am_clk_id_t *p_clk_id_buf, uint8_t buf_lenth)
8   {
9       int32_t i;
10      uint32_t clk = 0;
11      for (i = 0; i < buf_lenth; i++) {
12          clk = am_clk_rate_get(p_clk_id_buf[i]);
13          AM_DBG_INFO("CLK ID %d = %d\r\n", p_clk_id_buf[i], clk);
14      }
15      AM_FOREVER {
16          ; /* VOID */
17      }
18  }

```

同样的，每一个系列的 soc 文件夹目录下也有一个头文件“demo_zlg_entries.h”，用于存放其例程入口函数的声明，在使用到这些例程时，只需包含该头文件既可以。例如其 clk 例程声明详见程序清单 3.43。

程序清单 3.43 硬件层 CLK 例程声明

```

1  /**
2   * \brief CLK 例程，通过 HW 层接口实现

```

```

3  *
4  * \param[in] p_clk_id_buf 保存时钟号的缓冲区
5  * \param[in] buf_lenth 时钟号缓冲区大小
6  *
7  * \return 无
8  */
9  void demo_zlg_hw_clk_entry (am_clk_id_t *p_clk_id_buf, uint8_t buf_lenth);

```

3.8.4 驱动层例程

驱动层例程主要存放于 std 文件夹中，这些例程实现的功能与硬件层例程类似，不同的是这里的例程主要通过调用驱动层的函数来实现。如 adc 下，存放有两个例程，“demo_std_adc_ntc”和“demo_std_adc.c”，前者利用板子的热敏电阻（NTC）实现温度测量，后者利用查询方式获取 ADC 转换结果，两个例程都是通过标准接口实现。这些例程由于缺少相应板级环境支持，同样不能直接运行，通常在板级例程中被调用。例如查询方式获取 ADC 的例程，详见程序清单 3.44。

程序清单 3.44 驱动层查询方式获取 ADC 例程入口

```

1  /**
2  * \brief 例程入口
3  */
4  void demo_std_adc_entry (am_adc_handle_t handle, int chan)
5  {
6      int adc_bits = am_adc_bits_get(handle, chan); /* 获取 ADC 转换精度 */
7      int adc_vref = am_adc_vref_get(handle, chan);
8      uint32_t adc_code; /* 采样 Code 值 */
9      uint32_t adc_mv; /* 采样电压 */
10     am_kprintf("The ADC value channel is %d: \r\n", chan);
11     if (adc_bits < 0 || adc_bits >= 32) {
12         am_kprintf("The ADC channel is error, Please check! \r\n");
13         return;
14     }
15     while (1) {
16         adc_code = __adc_code_get(handle, chan);
17         adc_mv = adc_code * adc_vref / ((1UL << adc_bits) - 1);
18         /* 串口输出采样电压值 */
19         am_kprintf("Sample : %d, Vol: %d mv\r\n", adc_code, adc_mv);
20         am_mdelay(500);
21     }
22 }

```

同样的，“demo_std_entries.h”用于存放这些例程的入口函数声明。在使用到这些例程时，只需包含该头文件既可以。显然其 buzzer 例程声明在其中，详见程序清单 3.45。

程序清单 3.45 驱动层查询方式获取 ADC 函数声明

```

1  /**

```



```
2  * \brief ADC 例程，利用查询方式获取 ADC 转换结果，通过标准接口实现
3  *
4  * \param[in] handle ADC 标准服务句柄
5  * \param[in] chan ADC 通道号
6  *
7  * \return 无
8  */
9  void demo_std_adc_entry (am_adc_handle_t handle, int chan);
```

广州立功科技-工业技术研发中心

第4章 让 MCU 跑起来

4.1 基础工程搭建

要让新的芯片在 AMetal 平台上运行起来必然需要为它建立一个新的工程，AMetal 提供了搭建基础工程的模板工程（具体路径位于{SDK}\board\xxx_core），能够帮助开发者快速完成工程搭建。

4.1.1 搭建基础工程

将模板工程文件夹拷贝一份在当前目录（{SDK}\board），并更改名字为：{芯片简称}_core，此处为 zmf2x6_core，如图 4.1 所示。

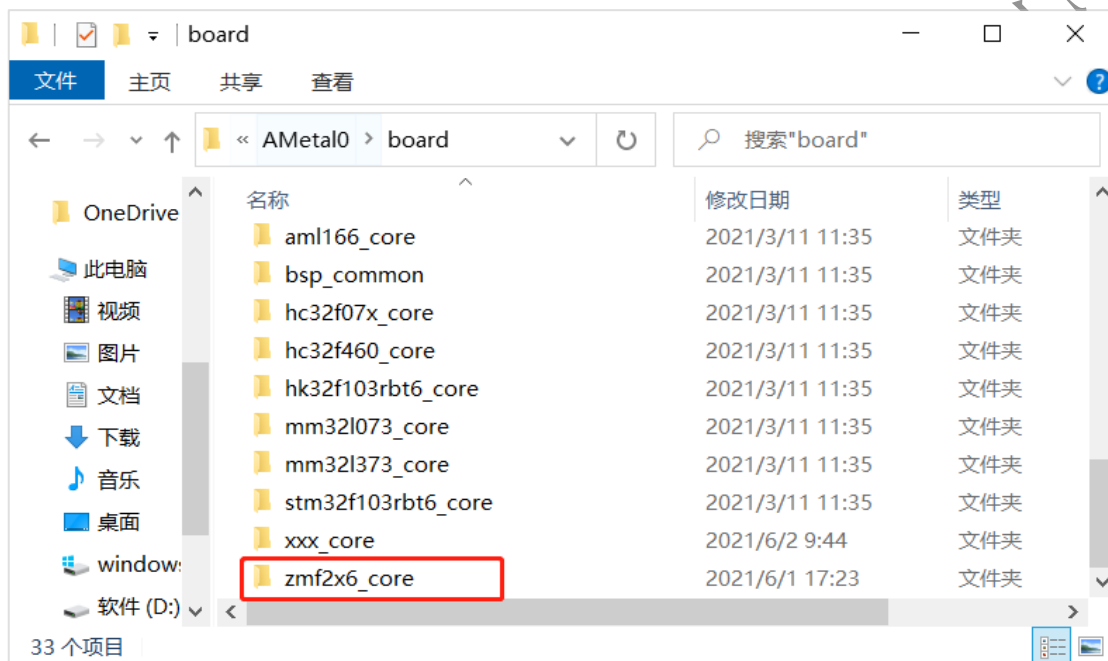


图 4.1 创建芯片基础工程文件夹

打开创建的芯片文件夹的 project_template 文件夹，将该文件夹（包含子目录）内所有 xxx 的字样改为所要适配芯片的简称，如图 4.2 所示，工程基本样式搭建完成。

example_zmf2x6_core.sct	2021/3/10 15:19	Windows Script ...	1 KB
example_zmf2x6_core.uvprojx	2021/3/23 15:12	Microvision5 Project	167 KB

图 4.2 工程文件名修改

4.1.2 添加芯片基础文件

由上文我们发现基础工程文件夹内仅包含该 MCU 的工程文件、相关配置文件和启动文件，关于该 MCU 的一些描述性文件几乎没有。在 AMetal 平台关于芯片相关文件被单独定义在了 soc 文件夹内，因此我们还需要在该文件夹下为芯片创建描述性文件，该操作主要分为 4 步。

1. 基于模板创建芯片文件夹

打开{SDK}\soc 文件夹。可以看到多个由芯片厂商命名的文件夹以及一个名为 xxx_soc 的文件夹，如图 4.3 所示。通过对所要适配芯片厂商进行区分，将 xxx_soc 文件夹

拷贝到对应的文件夹内。将该文件夹名更改为所要适配芯片的简称。

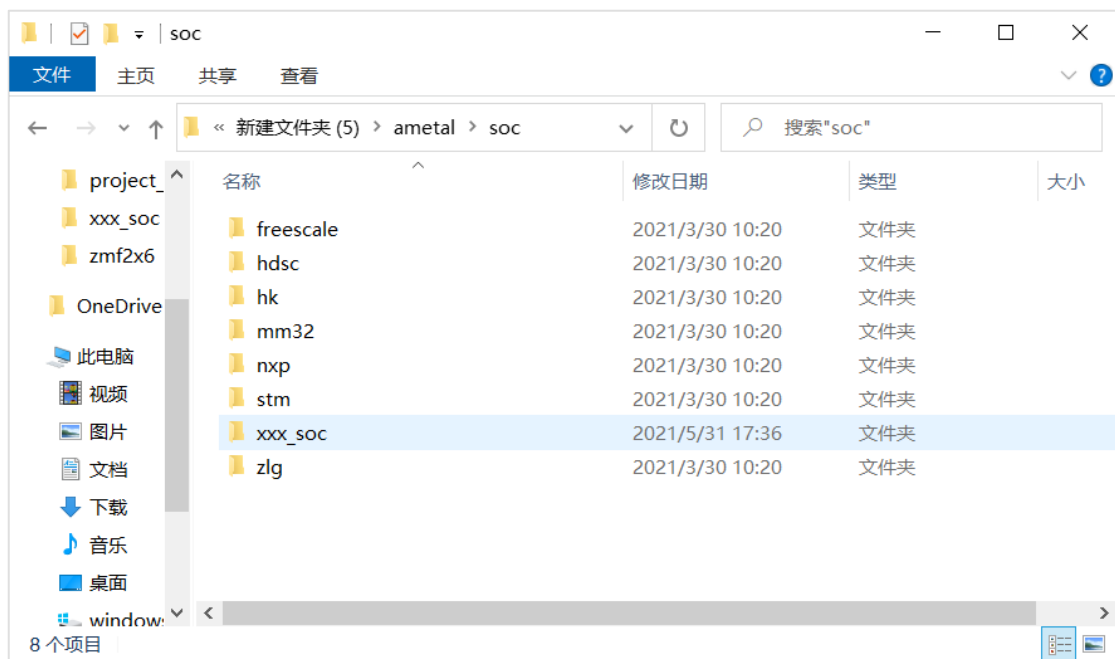


图 4.3 soc 文件夹目录

2. 修改模板文件名

打开该文件夹，把其中所有 xxx 字样改为所要适配芯片的简称。以 zmf2x6 为例，修改后如图 4.4 所示。

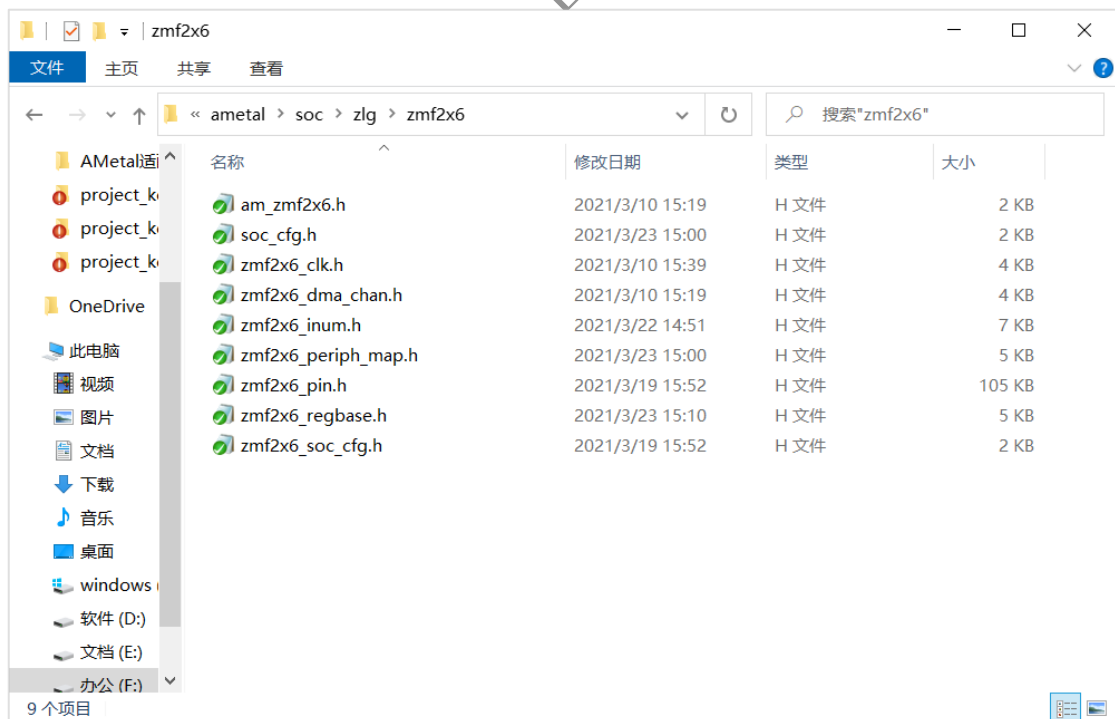


图 4.4 芯片基础文件定义

3. 修改文件芯片名称

打开文件夹下的所有文件（包括子目录文件），直接对全文件进行替换操作，将 xxx

全部替换为{芯片简称小写}，将 XXX 全部替换为{芯片简称大写}。

如果只是想先测试是否能够让 MCU 运行起来，到这里就已经可以直接进入下一小节了。

4. 修改文件内容

表 4.1 简单描述了该文件夹内包含文件的相关信息。

表 4.1 芯片基础文件简介

文件/文件名	描述
am_XXX.h	主要用于对以下头文件的包含，以及内核处理器的设置；
soc_cfg.h	芯片片上系统硬件配置；
XXX_clk.h	系统或外设时钟 ID 定义；
XXX_dma_chan.h	DMA 通道编号定义；
XXX_inum.h	中断号定义；
XXX_periph_map.h	外设映射定义；
XXX_pin.h	GPIO 引脚配置参数定义；
XXX_regbase.h	芯片外设寄存器基址定义；

在编写驱动程序前我们应完成 am_XXX.h、soc_cfg.h、XXX_inum.h、XXX_periph_map.h 文件内容的编写。

下面对各文件的编写做一个简单介绍。

- **am_XXX.h**: 对当前文件夹内的.h 文件进行包含。模板例程默认使用的是 Cortex-M0 的内核，若所适配芯片内核不同则需要对其进行修改（可参考其它同内核芯片该文件相关配置）；
- **soc_cfg.h**: 主要完成的是对通道数目、flash 大小、扇区大小的宏定义，需查阅芯片手册填写。
- **XXX_clk.h**: 该文件包含各总线和外设的时钟宏定义，它们的值直接决定 clk 外设驱动程序的编写。
- **XXX_dma_chan.h**: 该文件包含各外设可用 DMA 通道的情况，它的定义直接决定 DMA 外设驱动的编写。
- **XXX_inum.h**: 文件涉及到 CPU 中断号的宏定义。并定义枚举类型 IRQn_Type 提供给内核文件使用，模板例程为 Cortex-M0 内核，若所适配芯片内核不同则需要对它们进行修改。主要操作为中断向量的修改（可参考芯片启动文件）。
- **XXX_periph_map.h**: 外设映射定义，其主要目的是对各外设的基地址按照定义的对应外设结构体解析从而获取各外设寄存器块指针。该文件应包含所有外设硬件层头文件，因此此处全被注释掉了。
- **XXX_pin.h**: 包含该芯片各引脚的复用功能配置宏、模式配置宏、引脚速度配置宏，其应该在编写 GPIO 驱动时一同编写。
- **XXX_regbase.h**: 涉及到该芯片各外设的内存基地址定义。

4.1.3 修改工程相关文件

完成上述操作后就可以开始对工程配置文件进行修改了，该过程仅需两部。

1. 修改文件芯片名称

回到刚刚为芯片创建的工程文件夹内，进入 user_config 文件夹如图 4.5 所示。

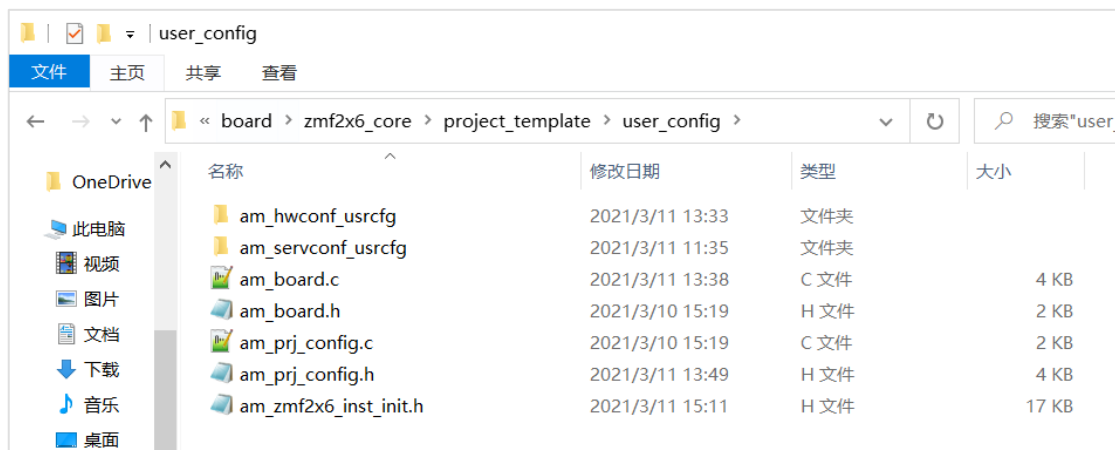


图 4.5 user_config 文件夹

将 user_config 文件夹下所有的文件打开（包括子目录文件），对全文件进行替换操作，将 xxx 全部替换为{芯片简称小写}，将 XXX 全部替换为{芯片简称大写}。

2. 修改 am_bord.c 文件

打开其中的 am_bord.c 文件，找到如程序清单 4.1 所示。该段代码主要用于描述 SRAM 大小，需根据所需适配芯片情况进行修改。

程序清单 4.1 芯片 SRAM 描述宏

```
/** \brief SRAM 信息,使用 ARMCC 时需要提供 SRAM 结束地址 */
#ifdef __CC_ARM
#define SRAM_SIZE      8
#define SRAM_START     0x20000000
#define SRAM_END       (SRAM_START + SRAM_SIZE * 1024)
```

至此，工程文件的修改工作完成，同时整个工程也初见雏形。现在我们可以使用 KEIL 打开该工程，准备下一小节的操作。

4.2 让 MCU 跑起来

由于各外设驱动还没有编写，MCU 启动后默认会使用内部中速时钟运行，因此我们可以通过 debug 程序来观察它是否能够进入 while（1），如果可以代表 MCU 运行成功。这里主要涉及到的相关操作为工程配置修改以及芯片启动文件的创建。

4.2.1 KEIL 工程配置修改

使用 KEIL 打开工程文件，在如图 4.6 所示界面下选择为所需适配的芯片型号。

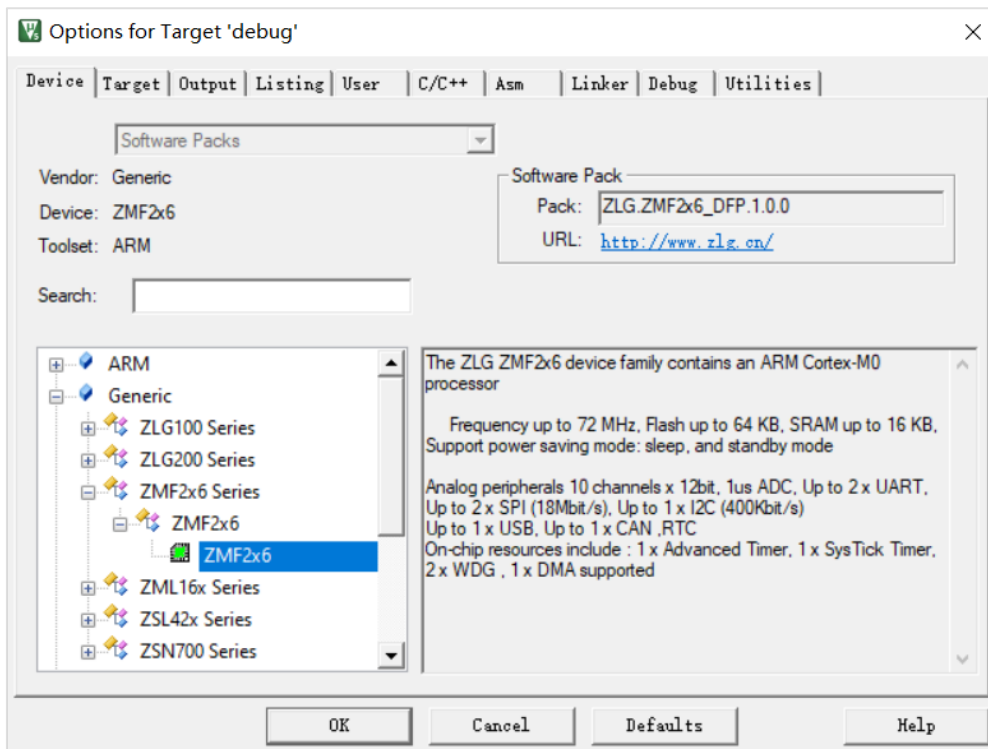


图 4.6 KEIL 芯片型号选择

在图 4.7 界面下修改内核的选择。

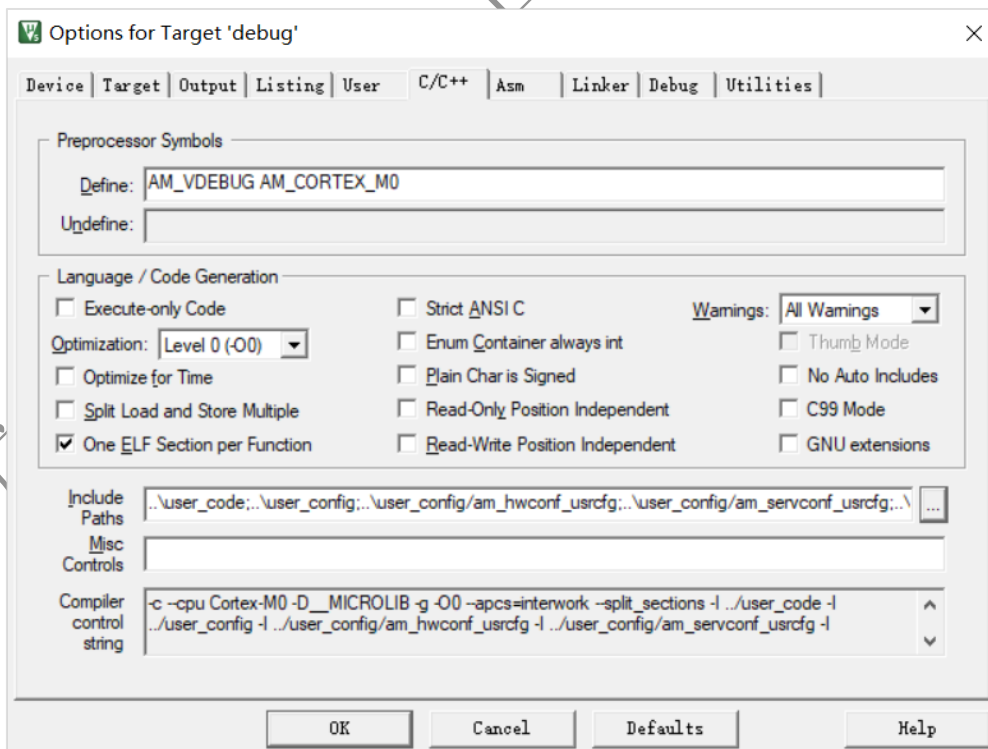


图 4.7 修改内核宏

给工程包含 4.1.2 创建的芯片基础文件夹的路径。此处为 zmf2x6，如图 4.8 所示。

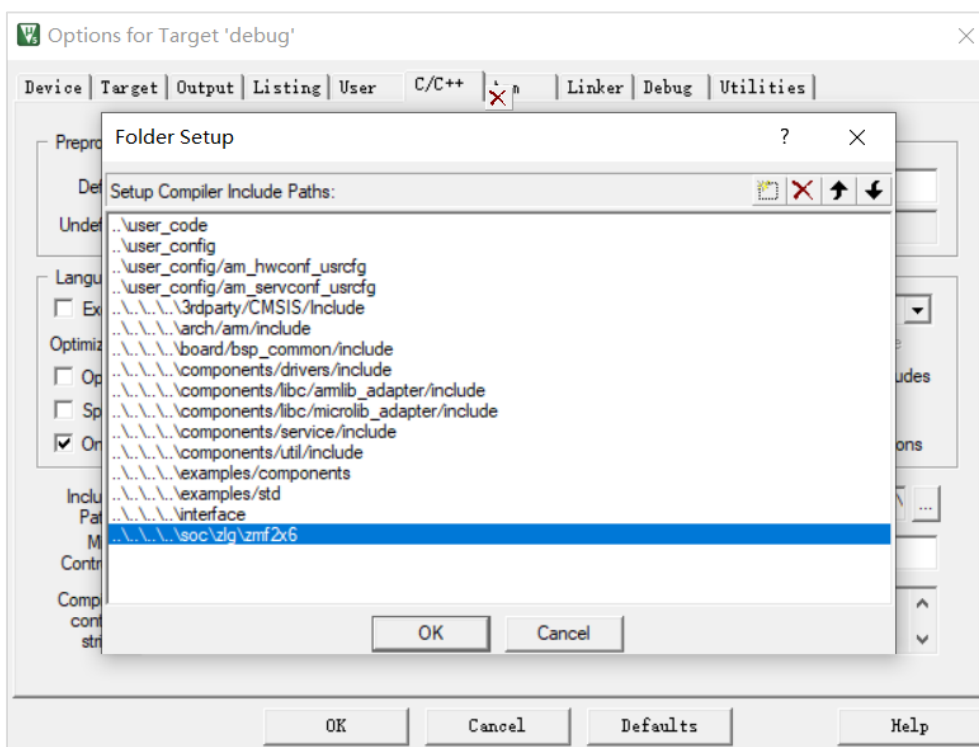


图 4.8 zmf2x6 基础文件路径包含

找到图 4.9 界面，修改其输出名为芯片的简称。

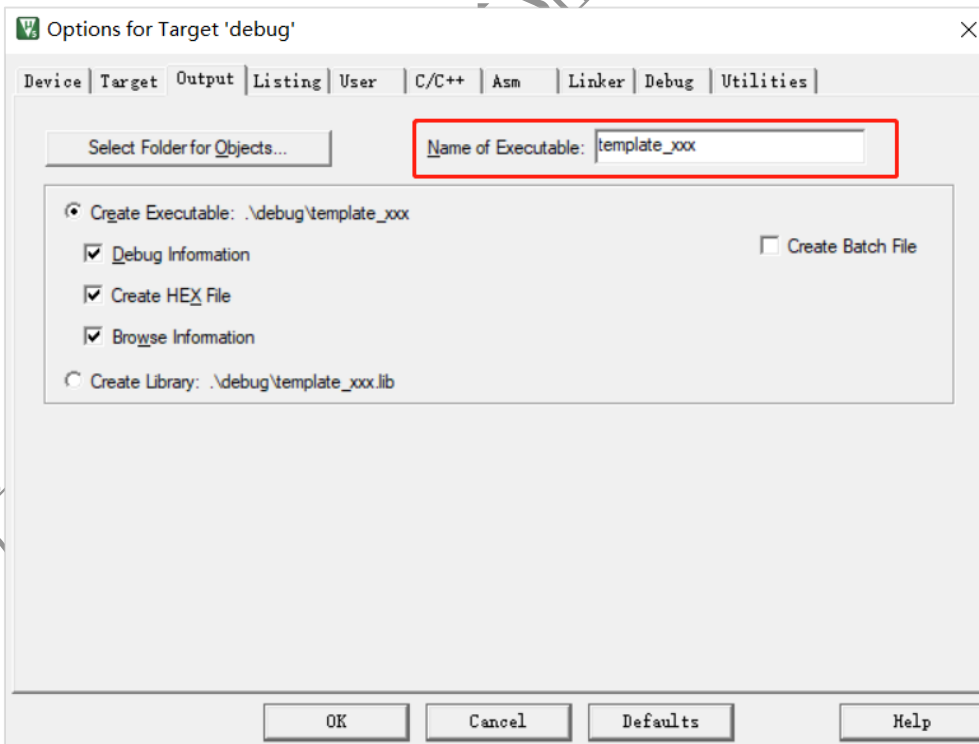


图 4.9 修改工程其输出文件名

在图 4.10 所示界面选择需要使用的烧录工具，并对烧录工具进行一些必要的配置。

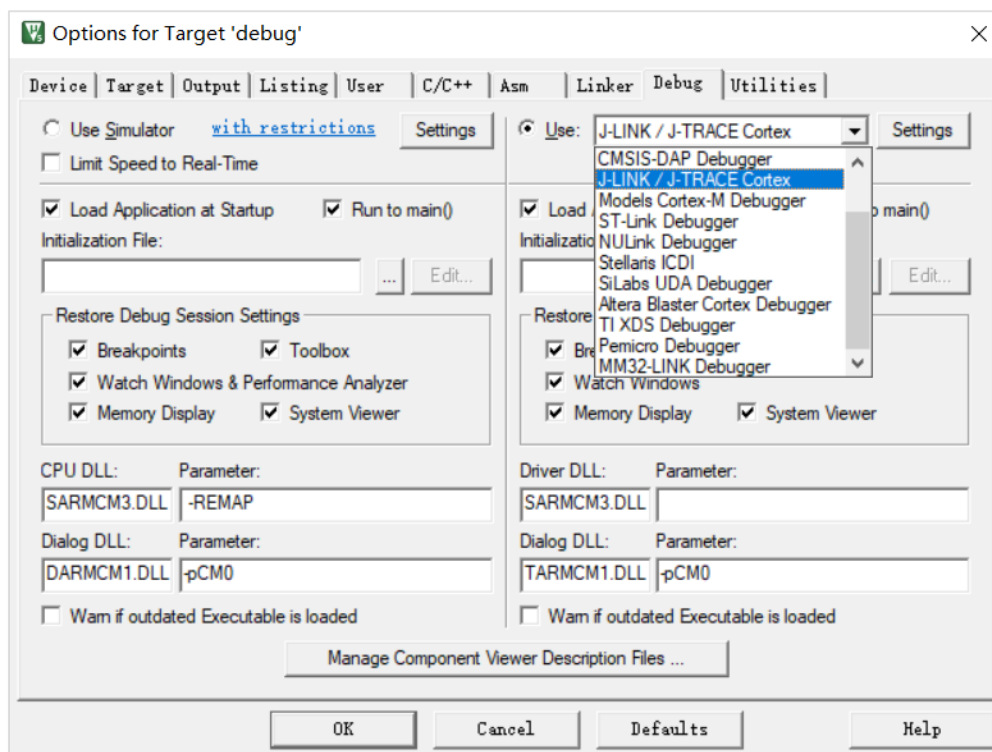


图 4.10 选择烧录工具

至此 KEIL 工程修改完成。编译链接工程，无错误。

4.2.2 创建启动文件

工程的启动文件应该存储于路径 {SDK}\board\{芯片简称}\{工程名}\startup 目录下，以 zmf2x6 为例如图 4.11 所示。启动文件可以通过将原厂启动文件进行修改获得，因此可以先将原厂的启动文件存入其中。

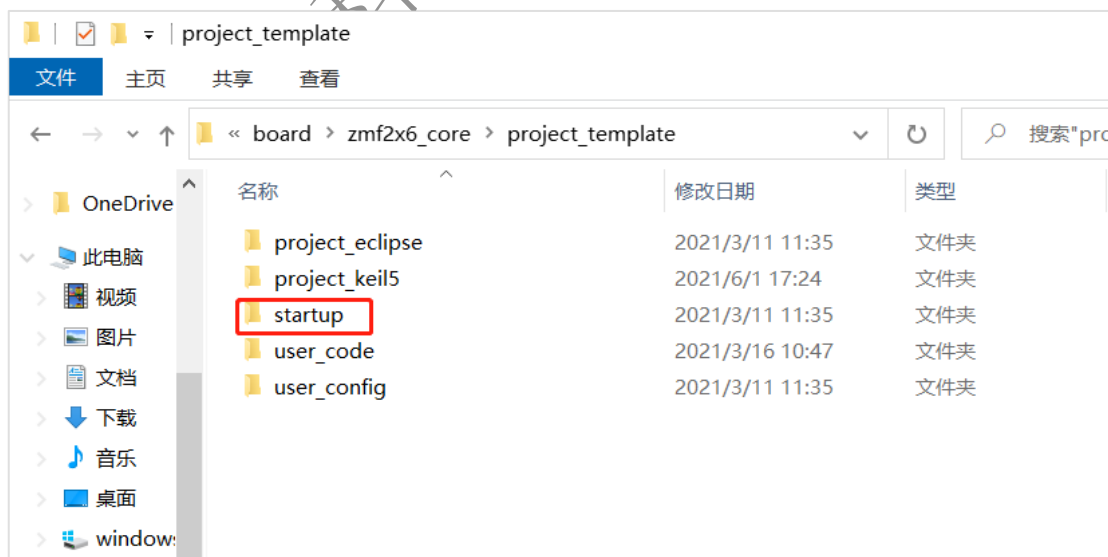


图 4.11 zmf2x6 启动文件存放位置

以 zmf2x6 为例，相较于原厂启动文件主要修改三个部分：

1. 外部中断函数连接

AMetal 平台需要将所有外部中断都连接到同一个中断函数，代码如程序清单 4.2 所示。

程序清单 4.2 zmf2x6 外部中断映射

```
; External Interrupts
DCD    am_exc_eint_handler    ; Window Watchdog
DCD    am_exc_eint_handler    ; PVD through EXTI Line detect
DCD    am_exc_eint_handler    ; RTC through EXTI Line & Tamper
DCD    am_exc_eint_handler    ; FLASH
DCD    am_exc_eint_handler    ; RCC & CRS
DCD    am_exc_eint_handler    ; EXTI Line 0 and 1
DCD    am_exc_eint_handler    ; EXTI Line 2 and 3
DCD    am_exc_eint_handler    ; EXTI Line 4 to 15
DCD    0                      ; Reserved
DCD    am_exc_eint_handler    ; DMA1 Channel 1
DCD    am_exc_eint_handler    ; DMA1 Channel 2 and Channel 3
DCD    am_exc_eint_handler    ; DMA1 Channel 4 and Channel 5
DCD    am_exc_eint_handler    ; ADC1 & COMP
DCD    am_exc_eint_handler    ; TIM1 Break, Update, Trigger and Commutation
DCD    am_exc_eint_handler    ; TIM1 Capture Compare
DCD    am_exc_eint_handler    ; TIM2
DCD    am_exc_eint_handler    ; TIM3
DCD    0                      ; Reserved
DCD    0                      ; Reserved
DCD    am_exc_eint_handler    ; TIM14
DCD    0                      ; Reserved
DCD    am_exc_eint_handler    ; TIM16
DCD    am_exc_eint_handler    ; TIM17
DCD    am_exc_eint_handler    ; I2C1
DCD    0                      ; Reserved
DCD    am_exc_eint_handler    ; SPI1
DCD    am_exc_eint_handler    ; SPI2
DCD    am_exc_eint_handler    ; UART1
DCD    am_exc_eint_handler    ; UART2
DCD    am_exc_eint_handler    ; AES
DCD    am_exc_eint_handler    ; CAN
DCD    am_exc_eint_handler    ; USB
```

2. Reset_Handler

只需将原厂的 Reset_Handler 修改成如程序清单 4.3 所示即可，AMetal 平台复位后直接执行 main 函数。

程序清单 4.3 zmf2x6 的 Reset_Handler

```
Reset_Handler    PROC
EXPORT  Reset_Handler    [WEAK]
IMPORT  __main
```

```

        LDR    R0, =__main
        BX     R0
    ENDP

```

3. Default_Handler

将原厂的 Reset_Handler 修改成如程序清单 4.4 所示即可。

程序清单 4.4 zmf2x6 的 Default_Handler

```

Default_Handler PROC

                EXPORT  am_exc_eint_handler      [WEAK]

am_exc_eint_handler
    B.

    ENDP

```

至此启动文件修改完成，如程序清单 4.5 为修改后完整的 zmf2x6 启动代码。

程序清单 4.5 zmf2x6 完整启动代码

```

Stack_Size      EQU      0x00000400

                AREA      STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem        SPACE    Stack_Size
__initial_sp


; <h> Heap Configuration
;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

Heap_Size        EQU      0x00000200

                AREA      HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem          SPACE    Heap_Size
__heap_limit


                PRESERVE8
                THUMB


; Vector Table Mapped to Address 0 at Reset
                AREA      RESET, DATA, READONLY
                EXPORT    __Vectors

```

__Vectors	DCD	__initial_sp	; Top of Stack
	DCD	Reset_Handler	; Reset Handler
	DCD	NMI_Handler	; NMI Handler
	DCD	HardFault_Handler	; Hard Fault Handler
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	SVC_Handler	; SVC Call Handler
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	PendSV_Handler	; PendSV Handler
	DCD	SysTick_Handler	; SysTick Handler
; External Interrupts			
	DCD	am_exc_eint_handler	; Window Watchdog
	DCD	am_exc_eint_handler	; PVD through EXTI Line detect
	DCD	am_exc_eint_handler	; RTC through EXTI Line & Tamper
	DCD	am_exc_eint_handler	; FLASH
	DCD	am_exc_eint_handler	; RCC & CRS
	DCD	am_exc_eint_handler	; EXTI Line 0 and 1
	DCD	am_exc_eint_handler	; EXTI Line 2 and 3
	DCD	am_exc_eint_handler	; EXTI Line 4 to 15
	DCD	0	; Reserved
	DCD	am_exc_eint_handler	; DMA1 Channel 1
	DCD	am_exc_eint_handler	; DMA1 Channel 2 and Channel 3
	DCD	am_exc_eint_handler	; DMA1 Channel 4 and Channel 5
	DCD	am_exc_eint_handler	; ADC1 & COMP
	DCD	am_exc_eint_handler	; TIM1 Break, Update, Trigger and Commutation
	DCD	am_exc_eint_handler	; TIM1 Capture Compare
	DCD	am_exc_eint_handler	; TIM2
	DCD	am_exc_eint_handler	; TIM3
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	am_exc_eint_handler	; TIM14
	DCD	0	; Reserved
	DCD	am_exc_eint_handler	; TIM16
	DCD	am_exc_eint_handler	; TIM17
	DCD	am_exc_eint_handler	; I2C1
	DCD	0	; Reserved
	DCD	am_exc_eint_handler	; SPI1

```

        DCD      am_exc_eint_handler      ; SPI2
        DCD      am_exc_eint_handler      ; UART1
        DCD      am_exc_eint_handler      ; UART2
        DCD      am_exc_eint_handler      ; AES
        DCD      am_exc_eint_handler      ; CAN
        DCD      am_exc_eint_handler      ; USB

        AREA     |.text|, CODE, READONLY

Reset_Handler PROC
    EXPORT Reset_Handler [WEAK]
    IMPORT __main
    LDR R0,=__main
    BX R0
    ENDP

NMI_Handler PROC
    EXPORT NMI_Handler [WEAK]
    B.
    ENDP

HardFault_Handler\
    PROC
    EXPORT HardFault_Handler [WEAK]
    B.
    ENDP

SVC_Handler PROC
    EXPORT SVC_Handler [WEAK]
    B.
    ENDP

PendSV_Handler PROC
    EXPORT PendSV_Handler [WEAK]
    B.
    ENDP

SysTick_HandlerPROC
    EXPORT SysTick_Handler [WEAK]
    B.
    ENDP

Default_Handler PROC

```

```

EXPORT am_exc_eint_handler [WEAK]

am_exc_eint_handler
    B.

    ENDP

    ALIGN

; User Initial Stack & Heap

    IF      :DEF:__MICROLIB

EXPORT __initial_sp
EXPORT __heap_base
EXPORT __heap_limit

    ELSE

        IMPORT __use_two_region_memory
        EXPORT __user_initial_stackheap
__user_initial_stackheap

        LDR    R0, = Heap_Mem
        LDR    R1, =(Stack_Mem + Stack_Size)
        LDR    R2, =(Heap_Mem  + Heap_Size )
        LDR    R3, = Stack_Mem
        BX     LR

        ALIGN

    ENDIF

END

```

启动文件创建完成后只需将其包含在芯片的工程文件内即可，如图 4.12 所示

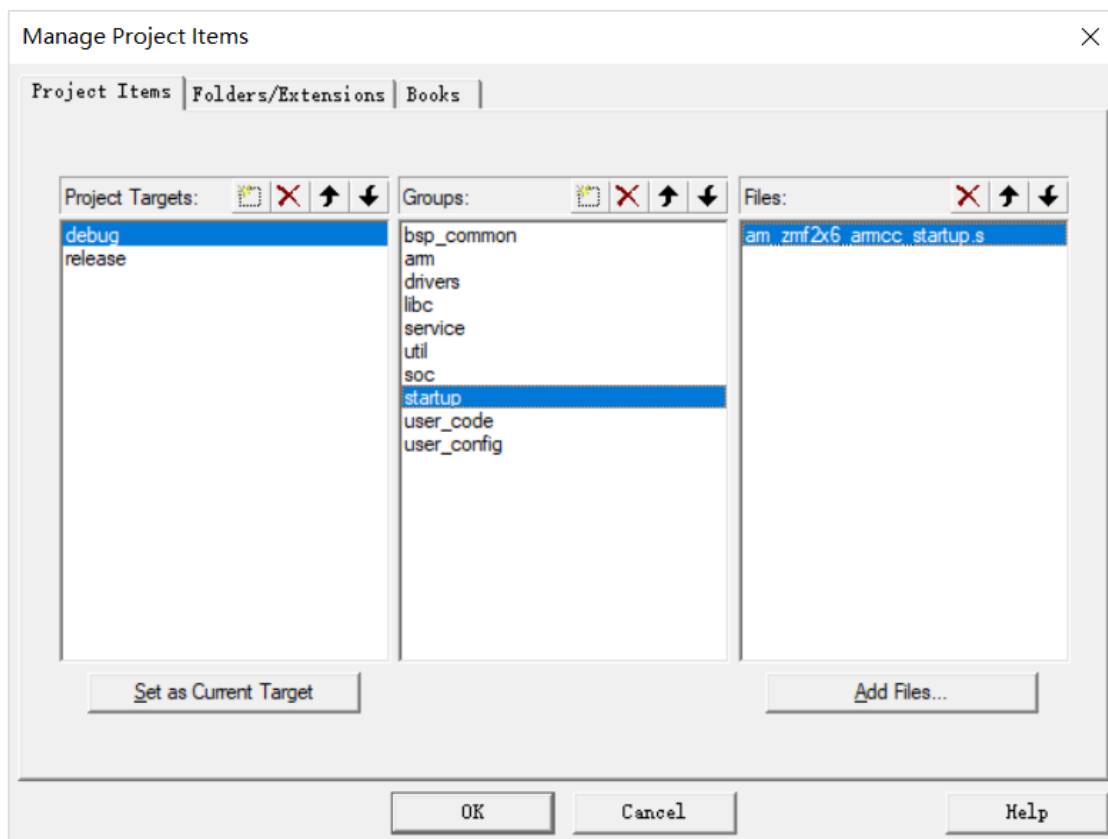


图 4.12 工程启动文件添加

4.2.3 Debug 测试验证

完成上述配置后，编译链接工程并调试程序，全速运行后停止调试，现象如图 4.13 所示，表示 MCU 运行成功。



图 4.13 MCU 运行成功现象

到此，第一步最小系统搭建完成，其他的外设开发工作就可以基于此工程开始。

第5章 通用外设驱动开发——WDT

5.1 通用外设标准层了解

关于标准接口层的说明，在 3.1 小节有较为详细的说明，可以去了解标准接口层函数是如何实现及如何应用，本节不再介绍其实现于抽象过程。对于通用 MCU 而言，AMetal 基本提供了各个外设的标准层接口，不需要额外开发，作为开发者只需要去了解即可。通用外设一般均是多实例设备，标准接口层的函数的第一个参数均为 `handle`，只有充分了解标准接口层，才能清楚驱动层需要开发的内容，AMetal 提供的所有的标准层驱动文件均在 `ametal\interface` 路径下。

本章以 WDT 外设为例，展示如何开发通用驱动外设代码。WDT 标准层仅包含一个文件，`am_wdt.h` 文件，在 `ametal\interface` 找到该文件打开，我们先对该文件提供的接口做简要的分析，明确驱动层该做什么。

5.1.1 了解标准接口相关类型定义

打开 `am_wdt.h` 文件，可以看到该文件主要定义了 3 个结构体和 1 个 `handle` 类型，如程序清单 5.1 所示，下面对每个结构体做详细介绍。

程序清单 5.1 WDT 标准接口文件结构体定义

```
/**
 * \brief WDT 超时信息
 */
typedef struct am_wdt_info {
    uint32_t min_timeout_ms;    /**< \brief 支持的最大超时时间，单位：ms */
    uint32_t max_timeout_ms;    /**< \brief 支持的最小超时时间，单位：ms */
} am_wdt_info_t;

/**
 * \brief WDT 驱动函数
 */
struct am_wdt_drv_funcs {

    /** \brief 获取 WDT 信息 */
    int (*pfn_wdt_info_get)(void *p_drv, am_wdt_info_t *p_info);

    /** \brief 设置超时时间并使能 WDT */
    int (*pfn_wdt_enable)(void *p_drv, uint32_t timeout_ms);

    /** \brief 看门狗喂狗 */
    int (*pfn_wdt_feed)(void *p_drv);
};

/**
 * \brief WDT 标准服务
 */
```

```
typedef struct am_wdt_serv {

    /** \brief WDT 驱动函数 */
    struct am_wdt_drv_funcs    *p_funcs;

    /** \brief 驱动函数第一个入口参数 */
    Void                      *p_drv;
} am_wdt_serv_t;

/** \brief WDT 标准服务操作句柄类型 */
typedef am_wdt_serv_t *am_wdt_handle_t;
```

1. struct am_wdt_info 结构体

该结构体包含了看门狗相关的信息，主要是最小超时时间和最大超时时间，后文提到的信息获取函数，就需要传入该结构体，以便获取到 WDT 相关的超时时间信息。

2. struct am_wdt_drv_funcs 结构体

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层要实现 3 个函数：

- 信息获取函数，获取 WDT 支持的最小超时时间和最大超时时间；
- 使能看门狗，并指定超时时间；
- 喂狗函数。

因此，在驱动层编写函数时，应该向标准接口层提供这三个函数。

这些函数的有一个共同点，就是第一个参数都是 void 型的指针，也就是说，驱动层提供的函数，标准接口层使用时传递的第一个参数并无确定类型，标准接口层不关心第一个参数的具体类型，使用时只负责传递就可以了。

那第一个参数从哪里来呢？驱动层怎么把这三个函数提供给标准接口层呢？看下一个结构体的定义。

3. typedef struct am_wdt_serv 结构体

该结构体定义了标准的 WDT 服务，包含了上文提到的驱动函数结构体的指针和一个 void 型的指针。

这个标准的 WDT 服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

那么，如何赋值呢？

● p_funcs 成员赋值

这是一个指向 struct am_wdt_drv_funcs 结构体的指针，我们只需要在驱动层定义一个 struct am_wdt_drv_funcs 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那三个函数地址。然后，将 struct am_wdt_drv_funcs 结构体变量的地址赋值给 p_funcs 成员即可。

● p_drv 成员赋值

这个变量是 void 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动自行决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

如果这个结构定义好了，标准接口层怎么知道有这个结构体呢？标准接口层怎么访问这个结构体呢？换句话说，驱动层怎么将该结构体提供给标准接口层呢？继续看下面 handle 的定义。

4. am_wdt_handle_t 的定义

这个是一个标准的操作句柄，本质上就是 struct am_wdt_serv 指针类型，驱动层的初始化函数需要返回该类型的一个 handle 给用户。这样，用户获取到 handle，就可以在需要使用标准接口层函数时，将该 handle 传递给标准接口层了。标准接口层就可以通过该 handle(struct am_wdt_serv 结构体指针) 访问到驱动层提供的函数了。标准接口层怎么使用 handle 指针？看一下标准接口层的函数实现就明白了。

5.1.2 了解标准接口函数

在 am_wdt.h 文件相关的结构体、类型定义后，就是具体函数的实现，如程序清单 5.2 所示。

程序清单 5.2 WDT 标准接口函数

```
/**
 * \brief 获取 WDT 信息
 *
 * \param[in] handle : WDT 标准服务操作句柄
 * \param[out] p_info : 指向 WDT 信息的指针
 *
 * \retval AM_OK : 操作成功
 * \retval -AM_EINVAL : 无效参数
 */
am_static_inline
int am_wdt_info_get(am_wdt_handle_t handle, am_wdt_info_t *p_info)
{
    return handle->p_funcs->pfn_wdt_info_get(handle->p_drv, p_info);
}

/**
 * \brief 设置超时时间并使能 WDT
 *
 * \param[in] handle : WDT 标准服务操作句柄
 * \param[in] timeout_ms : 超时时间值，单位：ms
 *
 * \retval AM_OK : 设置超时时间成功
 * \retval -AM_EINVAL : 参数无效，检查设定超时时间是否在允许范围内
 *
 * \note 如果设定的超时时间值不为 0，WDT 将使能。若超时不喂狗，会使芯片复位
 */
am_static_inline
int am_wdt_enable(am_wdt_handle_t handle, uint32_t timeout_ms)
{
```

```

        return handle->p_funcs->pfn_wdt_enable(handle->p_drv, timeout_ms);
    }

/**
 * \brief 看门狗喂狗
 *
 * \param[in] handle : WDT 标准服务操作句柄
 *
 * \retval  AM_OK    : 操作成功
 *
 * \note  如果不在设定的超时时间内喂狗，看门狗将复位芯片
 */
am_static_inline
int am_wdt_feed(am_wdt_handle_t handle)
{
    return handle->p_funcs->pfn_wdt_feed(handle->p_drv);
}

```

乍一看，很简单的三个函数，全都是用内联函数就搞定了。因为它本身就没做什么事情，只是简单的调用了一下驱动层提供的函数而已。可以看到，在调用驱动层提供的函数时，就是将 struct am_wdt_serv 结构体中的 p_drv 成员作为第一个参数传递。

可能疑惑的是，为什么需要定义一个 am_wdt_handle_t 这样的类型，直接定义成 struct am_wdt_serv 的指针多清晰易懂啊？在这里定义这样一个 handle 的概念，主要是为了减少用户的理解，用户只需要从驱动层获取一个 handle，就可以使用标准接口层函数，不需要关心这个 handle 具体是什么。只要驱动和标准接口知道这个 handle 是什么就可以了。

到此我们就分析完了 WDT 外设标准层文件，清楚了驱动层需要为标准接口层提供哪些内容。

5.2 WDT 外设 HW 层开发

通过上一节的内容，我们已经知道了标准层实现的函数的功能，以及需要驱动层提供的函数，但驱动层并不是凌空搭建的，驱动层介乎与 HW 层和标准接口层之间，简单来说就是调用 HW 层的函数来提供给标准层，所以在写驱动层前，我们先来完成 HW 层。

HW 层对应到具体的外设，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，内联函数直接定义在.h 文件中。硬件层所有的符号命名以 amhw_/AMHW_ 开头。通常情况下，HW 层只有.h 文件，只有当某些硬件功能设置较为复杂时，才提供对应的非内联函数，存放在.c 文件中。

现在我们打开 ZMF2x6 芯片用户手册，先了解一下该芯片 WDT 外设，可以从手册了解到，该芯片 WDT 外设包含独立看门狗和窗口看门狗，这里不解释两种看门狗的差别，我们只需知道，这两个看门狗均可以独立工作，独立向标准层提供所需的接口函数。IWDG（独立看门狗）和 WWDG（窗口看门狗）都需要独立实现。下面以 IWDG 为例，对照手册完成 HW 层的开发。

5.2.1 新建 HW 层文件

打开 ametal\soc\zlg\drivers\include\wdt\hw 路径，在该路径下创建名为 amhw_zmf2x6_iwdg.h 的文件，根据代码规范和生成 doxygen 文档的注释要求，在

amhw_zmf2x6_iwdg.h 文件中添加相关的注释以及防重复定义等必要内容，添加后整个文件的内容如程序清单 5.3 所示。

程序清单 5.3 amhw_zmf2x6_iwdg.h 文件初始内容添加

```
/**
 * \brief 获取 WDT 信息
 */
/*****

*
*                               AMetal
*
*                               -----
*
*                               innovating embedded platform
*
*
* Copyright (c) 2001-2018 Guangzhou ZHIYUAN Electronics Co., Ltd.
* All rights reserved.
*
* Contact information:
* web site:    http://www.zmf2x6.cn/
*****/

/**
 * \file
 * \brief IWDG 操作接口
 *
 *
 * \internal
 * \par Modification history
 * - 1.00 21-03-12 tee, first implementation
 * \endinternal
 */

#ifndef __AMHW_ZMF2X6_IWDG_H
#define __AMHW_ZMF2X6_IWDG_H

#ifdef __cplusplus
extern "C" {
#endif

#include "am_types.h"

/*
 * \brief 匿名结构体段的开始
 */

#ifdef __CC_ARM
    #pragma push
    #pragma anon_unions
#elif defined(__ICCARM__)
```

```

#pragma language=extended
#elif defined(__GNUC__)
#elif defined(__TMS470__)
#elif defined(__TASKING__)
    #pragma warning 586
#else
    #warning Not supported compiler t
#endif

/**
 * \addtogroup amhw_zmf2x6_if_iwdg
 * \copydoc amhw_zmf2x6_iwdg.h
 * @{
 */
/**
 * @}
 */
/*
 * \brief 匿名结构体段的结束
 */

#if defined(__CC_ARM)
    #pragma pop
#elif defined(__ICCARM__)
#elif defined(__GNUC__)
#elif defined(__TMS470__)
#elif defined(__TASKING__)
    #pragma warning restore
#else
    #warning Not supported compiler t
#endif

#ifdef __cplusplus
}
#endif /* __AMHW_ZMF2X6_IWDG_H */

#endif

/* end of file */

```

注：通常情况下，外设定义时均会使用到 AMetal 平台定义的类型，因此，默认情况下，均需要包含 am_types.h 文件。红色标识部分需要根据实际的日期和开发人员填写，日期应修改为正确的文件创建日期，tee 应修改为实际的文件创建者姓名，下同。

由于还不清楚是否需要创建.c 文件，可先不用创建，如果后续发现需要创建，可以使

用相同的方法创建。

文件创建完成之后，需要在 `zmf2x6_periph_map.h` 文件中包含新添加的 `amhw_zmf2x6_iwdg.h` 文件，打开 `ametal/soc/vlg/zmf2x6` 路径找到该文件，添加 `#include "hw/amhw_zmf2x6_iwdg.h"`。

5.2.2 添加寄存器列表

HW 层就是与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读该外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMF2x6 用户手册的 IWDG 章节，可以看到，IWDG 章节部分，共有 7 个寄存器，简单描述如下：

- IWDG_KR（键寄存器）：键值，以一定间隔写入 0xAAAA 喂狗，否则看门狗产生喂狗信号，写入 0x555 表示允许访问 IWDG-PR 和 IWDG_RLR 寄存器，写入 0xCCCC，启动看门狗。
- IWDG_PR（预分频寄存器）：通过该寄存器来选择计数器的预分频因子。要改变预分频因子，IWDG_SR 寄存器 PVU 必须为 0；
- IWDG_RLR（重载寄存器）：用于看门狗计数器的重载值，喂狗后，将该值传入计数器中，计数器从该值开始递减，看门狗超时时间可以用此计算；
- IWDG_SR（状态寄存器）：看门狗的状态控制寄存器，功能包含重载值更新标记、看门狗中断生成更新标志、重载值更新标志和看门狗预分频更新标志；
- IWDG_CR（控制寄存器）：独立看门狗中断标志清除和溢出操作方式选择（中断或复位）；
- IWDG_IGEN（中断生成寄存器）：中断生成阈值，当计数值递减等于该阈值，产生中断，只有 IWDG_SR 中的 IVU 位为 0，才能对此寄存器进行修改；
- IWDG_CNT（计数器寄存器）：IWDG 当前的计数器计数值；

到此 IWDG 的各个寄存器及其功能基本阅读完毕，我们也大致了解了各个寄存器功能以及要配置各个功能，其他寄存器位该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，程序清单如程序清单 5.4 所示。

程序清单 5.4 寄存器结构体定义

```
/**
 * \brief IWDG 寄存器块结构体
 */
typedef struct amhw_zmf2x6_iwdg {
    __I uint32_t kr;    /**< \brief 键寄存器 */
    __IO uint32_t pr;   /**< \brief 预分频寄存器 */
    __IO uint32_t rlr;  /**< \brief 重载寄存器 */
    __I uint32_t sr;    /**< \brief 状态寄存器 */
    __IO uint32_t cr;   /**< \brief 控制寄存器 */
    __IO uint32_t igen; /**< \brief 中断生成寄存器 */
    __I uint32_t cnt;   /**< \brief 计数寄存器 */
} amhw_zmf2x6_iwdg_t;
```

寄存器定义时，修饰宏 `__IO` 表示该寄存器是可读可写的，用户手册寄存器描述为 `rw`；`__O` 表示只写寄存器，用户手册描述为 `w`，`__I` 表示只读寄存器，用户手册描述为 `r`。不同原厂手册对此可能不完全一致，但该方法通用，即可读可写、只读、只写三种类型。

注：定义寄存器时，应该特别注意寄存器的地址，作为 32 位单片机，寄存器的地址应该是 4 字节 4 字节的增加。如果增加不是 4 字节，则寄存器与寄存器之间可能存在一个或多个保留寄存器，这些保留寄存器在用户手册中可能没有描述。保留的寄存器在定义结构体时也必须定义出来，这样才能确保寄存器地址是正确的。保留寄存器可以这样定义： `__I uint32_t reserved0; /**< \brief 保留 */`

这个小节我们定义了 IWDG 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

所有的内联函数都应该在定义时标识为内联函数，在函数的上方加入 `am_static_inline` 的关键字，具体见下文的示例。

5.2.3 编写寄存器相关操作函数

1. IWDG_KR（键寄存器）

通过用户手册，我们了解该寄存器有含义的寄存器就 1 个，由 16 位组成。因此它的操作函数也较为简单，填入的值主要有三种，操作函数如程序清单 5.5 所示。

程序清单 5.5 IWDG_KR 操作函数

```
/**
 * \brief 写键值
 *
 * \param[in] p_hw_iwdg：指向 IWDG 寄存器块的指针
 * \param[in] key      ：键值
 *
 * - 0x5555：允许访问 IWDG_PR 和 IWDG_RLR 寄存器
 * - 0xAAAA：喂狗
 * - 0xCCCC：启动看门狗工作
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_iwdg_keyvalue_set (amhw_zmf2x6_iwdg_t *p_hw_iwdg,
                                     uint16_t key)
{
    p_hw_iwdg->kr = key;
}
```

2. IWDG_PR（预分频寄存器）

该寄存器来选择计数器的预分频因子，同样只有 1 个有含义的位，由 3 个位组成，共 8 种分频值可用，其内联函数如程序清单 5.6 所示。

程序清单 5.6 IWDG_PR 操作函数

```

/**
 * \brief 设置预分频值
 *
 * \param[in] p_hw_iwdg: 指向 IWDG 寄存器块的指针
 * \param[in] div      : 分频值
 *
 * - 0 : 预分频因子 = 4
 * - 1 : 预分频因子 = 8
 * - 2 : 预分频因子 = 16
 * - 3 : 预分频因子 = 32
 * - 4 : 预分频因子 = 64
 * - 5 : 预分频因子 = 128
 * - 6 : 预分频因子 = 256
 * - 7 : 预分频因子 = 256
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_iwdg_div_set (amhw_zmf2x6_iwdg_t *p_hw_iwdg,
                               uint8_t div)
{
    p_hw_iwdg->pr = (div & 0x7ul);
}

```

3. IWDG_RLR（重载寄存器）

该寄存器用于看门狗计数器的重载值，喂狗后，将该值传入计数器中，计数器从该值开始递减，看门狗超时时间可以用此计算，通过了解，该寄存器有含义的只有这 1 个，其操作函数如程序清单 5.7 所示。

程序清单 5.7 IDWG_RLR 操作函数

```

/**
 * \brief 设置重载装值
 *
 * \param[in] p_hw_iwdg: 指向 IWDG 寄存器块的指针
 * \param[in] val      : 重载值
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_iwdg_reload_set (amhw_zmf2x6_iwdg_t *p_hw_iwdg,
                                   uint16_t val)
{
    p_hw_iwdg->rlr = (val & 0xfffful);
}

```

4. IWDG_SR（状态寄存器）

通过手册，我们了解到该寄存器有含义的寄存器共有 4 位，如表 5.1 所示，该寄存器。

表 5.1 IWDG_SR 寄存器分析

位	标识符	含义
0	PVU	看门狗预分频更新，由硬件置位来表示预分频值正在更新中
1	RVU	看门狗计数值重载值更新，由硬件置位来表示重载值的更新正在进行
2	IVU	看门狗中断生成值更新，此为用来指示中断更新正在生成
3	UPDATE	看门狗重载值更新标记，更新计数器

如表所示，所有寄存器均为只读寄存器，所以直接读即可，操作函数如程序清单 5.8 所示。

程序清单 5.8 IDWG_RLR 操作函数

```
/**
 * \brief 看门狗状态更新
 *
 * \param[in] p_hw_iwdg: 指向 IWDG 寄存器块的指针
 *
 * \return 状态值（各个状态是或的关系）
 *
 * - 0 : 无状态
 * - 1 : 看门狗预分频更新
 * - 2 : 看门狗计数器重装值更新
 * - 4 : 看门狗中断生成值更新
 * - 8 : 看门狗重载值更新标记
 */
am_static_inline
uint8_t amhw_zmf2x6_iwdg_status_get (amhw_zmf2x6_iwdg_t *p_hw_iwdg)
{
    return (uint8_t)(p_hw_iwdg->sr & 0xF);
}
```

5. IWDG_CR（控制寄存器）

独立看门狗中断标志清除和溢出操作方式选择（中断或复位），寄存器分析如表 5.2 所示。

表 5.2 IWDG_CR 寄存器分析

位	标识符	含义
0	IRQ_CLR	IWDG 中断清除，写 1 清除中断
1	IRQ_SEL	IWDG 溢出操作选择，写 1 溢出后中断，写 0 溢出后产生复位

● IRQ_CLR 操作函数

该位写 1 清除中断，函数实现如程序清单 5.9 所示。

程序清单 5.9 IRQ_CLR 操作函数


```

/**
 * \brief 看门狗清除中断
 *
 * \param[in] p_hw_iwdg: 指向 IWDG 寄存器块的指针
 */
am_static_inline
void amhw_zmf2x6_iwdg_irq_clr (amhw_zmf2x6_iwdg_t *p_hw_iwdg)
{
    p_hw_iwdg->cr &= 0x1;
}

```

● IRQ_SEL 操作函数

该位用于选择看门狗溢出后是中断还是复位，函数实现如程序清单 5.10 所示。

程序清单 5.10 IRQ_SEL 操作函数

```

/**
 * \brief 看门狗计数溢出操作
 *
 * \param[in] p_hw_iwdg: 指向 IWDG 寄存器块的指针
 * \param[in] irq_sel :
 *     - 0 : 溢出后产生复位
 *     - 1 : 溢出后产生中断
 */
am_static_inline
void amhw_zmf2x6_iwdg_irq_sel_set (amhw_zmf2x6_iwdg_t *p_hw_iwdg, uint8_t irq_sel)
{
    p_hw_iwdg->cr = (p_hw_iwdg->cr & 0x2) | irq_sel;
}

```

6. IWDG_IGEN (中断生成寄存器)

中断生成阈值，当计数值递减等于该阈值，产生中断，只有 IWDG_SR 中的 IVU 位为 0，才能对此寄存器进行修改，IGEN 由 12 位组成，其操作函数如程序清单 5.11 所示。

程序清单 5.11 IWDG_IGEN 操作函数

```

/**
 * \brief 设置看门狗中断生成值
 *
 * \param[in] p_hw_iwdg: 指向 IWDG 寄存器块的指针
 * \param[in] igen_val : 中断生成值 (12-bit)
 *
 * 只有当 IWDG_SR 寄存器中的 IVU 位为 0 时，才能对此寄存器进行修改。
 */
am_static_inline
void amhw_zmf2x6_iwdg_igen_set (amhw_zmf2x6_iwdg_t *p_hw_iwdg, uint16_t igen_val)
{

```

```
p_hw_iwdg->igen = igen_val;
}
```

7. IWDG_CNT（计数器寄存器）

IWDG 当前的计数器计数值，该寄存器值只读，操作函数如程序清单 5.12 所示。

程序清单 5.12 IWDG_CNT 操作函数

```
/**
 * \brief 获取看门狗计数器当前值
 *
 * \param[in] p_hw_iwdg: 指向 IWDG 寄存器块的指针
 *
 */
am_static_inline
uint16_t amhw_zmf2x6_iwdg_cnt_get (amhw_zmf2x6_iwdg_t *p_hw_iwdg)
{
    return p_hw_iwdg->cnt | 0xFFF00;
}
```

5.2.4 检查补充所需功能函数

在每个寄存器的操作函数编写完毕后，可能 HW 层的接口还有所欠缺，需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到，可以暂时不用考虑，直接进入驱动层的开发。在后续如果发现 HW 层欠缺哪些接口，可以反过来再增加。当然，也有可能 HW 层接口没有什么欠缺了，那样就无需再开发其它函数了。

我们回顾一下所有的函数，虽然对所有寄存器都提供了操作函数，但还欠缺返回时钟预分频的函数，这时候我们可以在继续补上，如程序清单 5.13 所示。

程序清单 5.13 时钟预分频获取操作函数

```
/**
 * \brief 获取看门狗预分频计数器当前值
 *
 * \param[in] p_hw_iwdg: 指向 IWDG 寄存器块的指针
 *
 */
am_static_inline
uint8_t amhw_zmf2x6_iwdg_ps_get (amhw_zmf2x6_iwdg_t *p_hw_iwdg)
{
    return p_hw_iwdg->cnt | 0xFF;
}
```

到此，IWDG 的 HW 层基本结束，如果有函数功能较为复杂的操作，我们可以在 ametal\soc\zlg\drivers\source\wdt\hw 下补充一个对应的.c 文件，编写好函数内容之后，还是需要在这个.h 文件中添加函数声明。

5.2.5 添加指向寄存器块的指针

可以看到，在 HW 层编写的函数中，第一个参数均为指向 IWDG 寄存器块的指针，这

个指针从哪里来呢？只需要把 IWDG 的基地址使用(amhw_iwdg_t *)强制转换即可作为这些函数的第一个参数。这样转换后，操作结构体的成员，实际上就是操作具体地址上的寄存器了。

但如果每次使用都要这样转换的话就太麻烦了，为了方便用户或驱动层使用，可以将该转换使用一个宏定义好，用户直接使用宏即可。在 AMetal 中，由于这些内容与具体型号的芯片相关，这样的宏统一定义到 zmf2x6_periph_map.h 文件中，如程序清单 5.14 所示。

程序清单 5.14 外设中加入 IWDG 的定义

```
/**
 * \name 外设定义
 * @{
 */
/** \brief NVIC 寄存器块指针 */
#define ZMF2X6_INT      ((amhw_arm_nvic_t      *)ZMF2X6_NVIC_BASE)

/** \brief SYSTICK 寄存器块指针 */
#define ZMF2X6_SYSTICK ((amhw_arm_systick_t    *)ZMF2X6_SYSTICK_BASE)

/** \brief 16 位通用定时器 17 寄存器指针 */
#define ZMF2X6_TIM17    ((amhw_zmf2x6_tim_t    *)ZMF2X6_TIM17_BASE)

/** \brief 独立看门狗(IWDG)寄存器块指针 */
#define ZMF2X6_IWDG     ((amhw_zmf2x6_iwdg_t    *)ZMF2X6_IWDG_BASE)

/** \brief 窗口看门狗(WWDG)寄存器块指针 */
#define ZMF2X6_WWDG     ((amhw_zmf2x6_wwdg_t    *)ZMF2X6_WWDG_BASE)
/**
 * @} zmf2x6_if_periph_map
 */
```

后续用户或驱动层使用 HW 层接口时，第一个参数直接使用 ZMF2X6_IWDG 即可。至此，整个 HW 层的开发工作就结束了，如果后续开发中发现缺少某些函数，还可以再添加。

5.3 外设驱动层开发

通过对标准接口层文件的分析理解，清楚了驱动层需要为标准接口层提供哪些内容，加上驱动本身应该提供给用户的初始化函数和解初始化函数。总体内容再归纳总结如下：

1. 编写好标准接口层需要使用的 3 个基本函数；
2. 定义 struct am_wdt_drv_funcs 结构体，将其 3 个成员赋值为编写好的 3 个函数；
3. 定义 struct am_wdt_serv 结构体，赋值好相关值，在驱动初始化函数中将其地址返回给用户；
4. 驱动初始化函数；
5. 驱动解初始化函数。

基本分析得差不多了，着手驱动层的开发吧！

驱动层开发主要分为建立驱动层文件和编写函数两个部分进行介绍。

5.3.1 新建驱动文件

驱动层文件有两个，am_zml2x6_iwdg.c 和 am_zml2x6_iwdg.h。存放路径分别为 ametal\soc\zlg\drivers\source\wdt 和 \metal\soc\zlg\drivers\include\wdt。

1. am_zml2x6_iwdg.h

根据代码规范和生成 doxygen 文档的注释要求，在 am_zml2x6_iwdg.h 文件中添加相关的注释以及防重复定义等必要内容，添加后整个文件的内容如程序清单 5.15 所示。

程序清单 5.15 am_zml2x6_iwdg.h 文件初始内容添加

```

/*****
*
*                                     AMetal
*
*      -----
*
*      innovating embedded platform
*
*
* Copyright (c) 2001-2018 Guangzhou ZHIYUAN Electronics Co., Ltd.
* All rights reserved.
*
*
* Contact information:
* web site:    http://www.zlg.cn/
*****/

/**
* \file
* \brief IWDG 驱动层接口
*
*
* \internal
* \par Modification history
* - 1.00 21-03-12  zjr, first implementation
* \endinternal
*/

#ifndef __AM_ZMF2X6_IWDG_H
#define __AM_ZMF2X6_IWDG_H

#ifdef __cplusplus
extern "C" {
#endif

#include "ametal.h"
#include "am_wdt.h"

/**
* \addtogroup am_zmf2x6_if_iwdg
* \copydoc am_zmf2x6_iwdg.h
* @{

```

```

*/

/**
 * @}
 */

#ifdef __cplusplus
}
#endif /* __AMHW_IWDG_H */

#endif

/* end of file */

```

2. am_zml2x6_iwdg.c

同样的创建 am_zml2x6_iwdg.c 文件，初始内容基本如程序清单 5.15 所示，添加相关的头文件声明，如程序清单 5.16 所示。

程序清单 5.16 am_zml2x6_iwdg.c 文件初始内容添加

```

/**
 * \file
 * \brief IWDG 驱动层接口
 * \internal
 * \par Modification history
 * - 1.00 15-07-10 sss, first implementation
 * \endinternal
 */

#include "ametal.h"
#include "am_zmf2x6_iwdg.h"
#include "hw/amhw_zmf2x6_iwdg.h"
#include "am_int.h"

/*****

* Functions Declaration

*****/

```

5.3.2 编写结构性程序

驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备（IWDG 就是一个设备）都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

可以暂时定义设备信息结构体如程序清单 5.17 所示。后续可以根据实际情况修改。

程序清单 5.17 IWDG 设备信息结构体定义

```
/**
 * \brief IWDG 设备信息结构体
 */
typedef struct am_zmf2x6_iwdg_devinfo {

    /** \brief IWDG 寄存器块基地址 */
    uint32_t iwdg_regbase;

    /** \brief IWDG 中断号 */
    uint8_t inum;

    /** \brief IWDG 中断函数 */
    void    (__iwdg_irq_handler)(void *);

    /** \brief 中断生成值 */
    uint16_t int_value;

    /** \brief 中断使能 */
    uint8_t int_en;

    /** \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_zmf2x6_iwdg_devinfo_t;
```

● 设备结构体定义

由于 WDT 设备需要提供标准的 WDT 服务，因此，该设备至少包含一个 WDT 标准服务结构体成员，也就是 `am_wdt_serv_t` 类型成员，其次，为了在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 5.18 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 5.18 IWDG 设备结构体定义

```
/**
 * \brief IWDG 设备结构体
 */
```

```
typedef struct am_zmf2x6_iwdg_dev {

    /** \brief IWDG 提供的标准服务 */
    am_wdt_serv_t          wdt_serv;

    /** \brief 指向 IWDG 设备的信息指针 */
    const am_zmf2x6_iwdg_devinfo_t *p_devinfo;
} am_zmf2x6_iwdg_dev_t;
```

2. 搭建函数框架

函数主要分为两部分一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于看门狗这种已经有标准接口层的函数，通常驱动层只提供两个函数给用户，初始化函数和解初始化函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为用于标准接口的 handle。

解初始化函数主要是当不需要再使用某个外设时，释放掉其占用的相关资源，参数为设备指针，无返回值。示例代码如程序清单 5.19 所示。

程序清单 5.19 驱动层函数定义示例

```
/** \brief 初始化 IWDG，获取标准服务句柄 */
am_wdt_handle_t am_zmf2x6_iwdg_init (am_zmf2x6_iwdg_dev_t          *p_dev,
                                     const am_zmf2x6_iwdg_devinfo_t *p_devinfo)
{
    return &(p_dev->wdt_serv);
}

/** \brief IWDG 解初始化 */
void am_zmf2x6_iwdg_deinit (am_wdt_handle_t handle)
{
}
```

同时为了用户能够使用，还需要将这两个函数的声明放入到相应的头文件中。示例代码如程序清单 5.20 所示。

程序清单 5.20 驱动函数声明

```
/**
 * \brief IWDG 初始化
 *
 * \param[in] p_dev      : 指向 IWDG 设备
 * \param[in] p_devinfo : 指向 IWDG 设备信息
 *
 * \return IWDG 标准服务句柄，失败则返回 NULL
 */
am_wdt_handle_t am_zmf2x6_iwdg_init (am_zmf2x6_iwdg_dev_t          *p_dev,
```

```

const am_zmf2x6_iwdg_devinfo_t *p_devinfo);

/**
 * \brief 解除 IWDG 设备初始化
 *
 * \param[in] handle：使用 IWDG 初始化函数 am_zmf2x6_iwdg_init()获取到的 handle
 *
 * \return 无
 */
void am_zmf2x6_iwdg_deinit (am_wdt_handle_t handle);

```

- 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 3 个函数，根据标准接口层中的 `struct am_wdt_drv_funcs` 结构体，我们需要在.c 文件中定义 3 个函数。

由于这些函数全部是放在结构体中，通过函数指针的形式提供给上层，并不需要将这些函数声明到.h 文件中供用户直接使用，因此全部以 `static` 的形式定义成内部函数。定义好函数后，再定义一个 `struct am_wdt_drv_funcs` 结构体，将函数放在其中。示例代码如程序清单 5.21 所示。

程序清单 5.21 提供给标准层函数定义

```

static int __iwdg_feed (void *p_drv);
static int __iwdg_enable (void *p_drv, uint32_t timeout);
static int __iwdg_info_get (void *p_drv, am_wdt_info_t *p_info);

/** \brief 看门狗标准服务函数结构体 */
static const struct am_wdt_drv_funcs __g_iwdg_drv_funcs = {
    __iwdg_info_get,
    __iwdg_enable,
    __iwdg_feed
};

/** \brief 获取看门狗信息 */
static int __iwdg_info_get (void *p_drv, am_wdt_info_t *p_info)
{
    return AM_OK;
}

/** \brief 喂狗操作 */
static int __iwdg_feed (void *p_drv)
{
    return AM_OK;
}

```



```

/** \brief 配置超时时间并启动看门狗 */
static int __iwdg_enable (void *p_drv, uint32_t timeout_ms)
{

    return AM_OK;

}

```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。可以编译一下，应该还是没有错误的。如果有错误，表明结构没有搭建好。可能存在警告，毕竟有些变量没有使用，可以暂时不用理会。做到这里，其实驱动层已经开发一大半了。

5.3.3 驱动函数实现

通过上面的函数框架搭建，共计 5 个函数，3 个功能性函数面向标准层，2 个初始解初始化函数面向用户，下面逐步实现。

1. am_zmf2x6_iwdg_init 函数

该函数主要完成两个功能：

- a) 初始化设备中的标准服务，将成员赋值，确定 `p_drv` 的值，`p_drv` 的值确定了提供给标准接口层的那 3 个函数的第一个参数是什么，那 3 个函数才能根据这儿的设置去使用。（`void *` 类型无法使用，必须转换为具体的一种类型才能使用），通常情况下，`p_drv` 都是设置为设备的地址，这样方便访问整个设备的相关内容。
- b) 初始化硬件相关设置，该函数的示例代码如程序清单 5.22 所示。

程序清单 5.22 am_zmf2x6_iwdg_init 函数示例代码

```

/** \brief 初始化 IWDG，获取标准服务句柄 */
am_wdt_handle_t am_zmf2x6_iwdg_init (am_zmf2x6_iwdg_dev_t *p_dev,
                                     const am_zmf2x6_iwdg_devinfo_t *p_devinfo)
{
    if (p_devinfo == NULL) {
        return NULL;
    }

    p_dev->p_devinfo = p_devinfo;
    p_dev->wdt_serv.p_funcs = (struct am_wdt_drv_funcs *)&__g_iwdg_drv_funcs;
    p_dev->wdt_serv.p_drv = p_dev;
    p_dev->div = 4;

    amhw_zmf2x6_iwdg_t *p_hw_iwdg =
        (amhw_zmf2x6_iwdg_t *)p_dev->p_devinfo->iwdg_regbase;

    /* 看门狗平台初始化 */
    if (p_devinfo->pfn_plfm_init) {
        p_devinfo->pfn_plfm_init();
    }
}

```

```

if(p_dev->p_devinfo->int_en == 1) {
    amhw_zmf2x6_iwdg_igen_set(p_hw_iwdg, p_dev->p_devinfo->int_value);
    am_int_connect(p_dev->p_devinfo->inum,
                  p_dev->p_devinfo->__iwdg_irq_handler,
                  (void *)p_dev);

    am_int_enable(p_dev->p_devinfo->inum);
}

return &(p_dev->wdt_serv);
}

```

2. am_zmf2x6_iwdg_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放，同时，由于 IWDG 一旦使能将无法关闭，所以也无法释放硬件资源。

因此，该函数仅简单的将几个不再使用的变量置为 NULL 即可。示例代码如程序清单 5.23 所示。

程序清单 5.23 am_zmf2x6_iwdg_deinit 函数示例代码

```

/** \brief IWDG 解初始化 */
void am_zmf2x6_iwdg_deinit(am_wdt_handle_t handle)
{
    am_zmf2x6_iwdg_dev_t *p_dev = (am_zmf2x6_iwdg_dev_t *)handle;

    if (NULL == p_dev) {
        return ;
    }

    p_dev->p_devinfo = NULL;

    p_dev->wdt_serv.p_funcs = NULL;
    p_dev->wdt_serv.p_drv = NULL;

    if (p_dev->p_devinfo->pfn_plfm_deinit) {
        p_dev->p_devinfo->pfn_plfm_deinit();
    }
}

```

3. __iwdg_enable 函数

该函数用于获取看门狗支持的最大超时时间(单位：ms)和最小超时时间(单位：ms)。因此需要知道看门狗的计数值范围和计数时钟。

计数值由 WDTC 寄存器确定，用户手册已经明确其范围为 0x00 ~ 0xFFFF。

计数时钟由时钟源频率分频得到。由于 IWDG 的时钟可以设置多种分频，因此，WDT

支持的超时时间范围与时钟分频值的设置有关。标准接口并不需要时钟分频设置接口，那么时钟源的选择就应该由设备信息提供，也就是说，用户需要不同的时钟分频，只需要设备信息修改为需要的分频值即可。

因此，我们需要在设备信息中加入时钟分频的选择信息。加入后设备信息的定义如程序清单 5.24 所示

程序清单 5.24 设备信息添加预分频因子

```
/**
 * \brief IWDG 设备结构体
 */
typedef struct am_zmf2x6_iwdg_dev {

    /** \brief IWDG 提供的标准服务 */
    am_wdt_serv_t          wdt_serv;

    /** \brief 预分频因子 */
    uint16_t               div;

    /** \brief 指向 IWDG 设备的信息指针 */
    const am_zmf2x6_iwdg_devinfo_t *p_devinfo;
} am_zmf2x6_iwdg_dev_t;
```

有了该信息之后，就可以在初始化函数中加入时钟分频的设置程序。不过考虑该芯片的计数器只有 12 位，为最大可能满足应用，可以把时钟分频放到看门狗使能函数，通过判断需要的超时时间来合理设置预分频值，如程序清单 5.25 所示。

程序清单 5.25 按超时时间配置预分频，使能看门狗

```
/** \brief 配置超时时间并启动看门狗 */
static int __iwdg_enable (void *p_drv, uint32_t timeout_ms)
{
    am_zmf2x6_iwdg_dev_t *p_dev      = (am_zmf2x6_iwdg_dev_t *)p_drv;
    amhw_zmf2x6_iwdg_t      *p_hw_iwdg =
        (amhw_zmf2x6_iwdg_t *)p_dev->p_devinfo->iwdg_regbase;

    uint32_t wdt_freq = 40000;
    uint32_t ticks;
    uint32_t div = 1;

    ticks = (uint64_t) (timeout_ms) * wdt_freq / 1000;

    div = ticks / 0xFFFF + 1;

    amhw_zmf2x6_iwdg_keyvalue_set(p_hw_iwdg, 0x5555);
    while(amhw_zmf2x6_iwdg_status_get(p_hw_iwdg) & 0x1ul);
```

```

if (div <= 4) {
    amhw_zmf2x6_iwdg_div_set (p_hw_iwdg, 0);
    p_dev->div = 4;
} else if (div <= 8) {
    amhw_zmf2x6_iwdg_div_set (p_hw_iwdg, 1);
    p_dev->div = 8;
} else if (div <= 16) {
    amhw_zmf2x6_iwdg_div_set (p_hw_iwdg, 2);
    p_dev->div = 16;
} else if (div <= 32) {
    amhw_zmf2x6_iwdg_div_set (p_hw_iwdg, 3);
    p_dev->div = 32;
} else if (div <= 64) {
    amhw_zmf2x6_iwdg_div_set (p_hw_iwdg, 4);
    p_dev->div = 64;
} else if (div <= 128) {
    amhw_zmf2x6_iwdg_div_set (p_hw_iwdg, 5);
    p_dev->div = 128;
} else {
    amhw_zmf2x6_iwdg_div_set (p_hw_iwdg, 6);
    p_dev->div = 256;
}

wdt_freq /= p_dev->div;
ticks = (uint64_t) (timeout_ms) * wdt_freq / 1000;

amhw_zmf2x6_iwdg_keyvalue_set(p_hw_iwdg, 0x5555);
while(amhw_zmf2x6_iwdg_status_get(p_hw_iwdg) & 0x2ul);
amhw_zmf2x6_iwdg_reload_set (p_hw_iwdg, ticks);

/* 启动看门狗 */
amhw_zmf2x6_iwdg_keyvalue_set(p_hw_iwdg, 0xAAAA);
amhw_zmf2x6_iwdg_keyvalue_set(p_hw_iwdg, 0xCCCC);

return AM_OK;
}

```

4. __iwdg_info_get 函数

独立看门狗的时钟为 40K，最大最小的分频值是 4 分频和 256 分频，通过这些信息，就可以计算出 IWDG 支持的超时时间范围。程序如程序清单 5.26 所示。

程序清单 5.26 获取看门狗超时时间信息

```

/** \brief 获取看门狗信息 */
static int __iwdg_info_get (void *p_drv, am_wdt_info_t *p_info)
{

```

```

uint32_t wdt_freq      = 40000;
uint32_t wdt_freq_max  = wdt_freq / 4;
uint32_t wdt_freq_min  = wdt_freq / 256;

p_info->min_timeout_ms = 1 * 1000 / wdt_freq_max + 1;
p_info->max_timeout_ms = (uint64_t)(0xFFFF + 1) * 1000 /
                        (uint64_t)wdt_freq_min ;

return AM_OK;
}

```

5. __iwdg_feed 函数

这个函数就是实现喂狗操作，由于 HW 层已经有喂狗函数了，所以该函数实现简单，直接调用 HW 层喂狗函数即可，示例代码如程序清单 5.27 所示。

程序清单 5.27 看门狗喂狗函数

```

/** \brief 喂狗操作 */
static int __iwdg_feed (void *p_drv)
{
    am_zmf2x6_iwdg_dev_t *p_dev      = (am_zmf2x6_iwdg_dev_t *)p_drv;
    amhw_zmf2x6_iwdg_t  *p_hw_iwdg =
        (amhw_zmf2x6_iwdg_t *)p_dev->p_devinfo->iwdg_regbase;

    amhw_zmf2x6_iwdg_keyvalue_set(p_hw_iwdg, 0xAAAA);

    return AM_OK;
}

```

到现在 IWDG 支持的所有函数开发完毕，整个 IWDG 的驱动层开发也就结束了。

5.4 外设用户配置文件开发

开发完驱动层我们知道，用户使用一个外设的过程是，先调用驱动初始化函数，该函数返回一个 handle，后续的所有操作，用户直接使用该 handle 作为参数调用标准接口层函数即可。

由于用户调用驱动初始化函数时需要传入设备地址和设备信息地址作为参数。为了方便用户使用，我们需要将设备变量和设备信息常量定义好，用户调用驱动初始化函数时，只需要使用定义好的设备变量和设备信息常量即可，不需要再额外定义。

5.4.1 新建配置文件

设备变量和设备信息常量定义在 am_hwconf_zmf2x6_iwdg.c 文件中，相应的声明放在 am_hwconf_zmf2x6_iwdg.h 文件中或在 am_zmf2x6_inst_init.h 中。所以我们需要创建的配置文件至少包含 1 个 am_hwconf_zmf2x6_iwdg.c 文件。

根据代码规范和生成 doxygen 文档的注释要求，在 am_hwconf_zmf2x6_iwdg.c 文件中添加相关注释等必要内容，添加后整个文件的内容如程序清单 5.28 所示。

程序清单 5.28 创建配置文件

```

/*****

*
*                               AMetal
*
*   -----
*
*   innovating embedded platform
*
*
* Copyright (c) 2001-2018 Guangzhou ZHIYUAN Electronics Co., Ltd.
* All rights reserved.
*
* Contact information:
* web site:    http://www.zmf2x6.cn/
*****/

/**
 * \file
 * \brief ZMF2X6 WWDT 用户配置文件
 * \sa am_hwconf_zmf2x6_iwdg.c
 *
 * \internal
 * \par Modification history
 * - 1.00 21-03-23   zjr, first implementation
 * \endinternal
 */

#include "am_zmf2x6.h"
#include "hw/amhw_zmf2x6_rcc.h"
#include "am_zmf2x6_iwdg.h"

/**
 * \addtogroup am_if_src_hwconf_zmf2x6_iwdg
 * \copydoc am_hwconf_zmf2x6_iwdg.c
 * @{
 */

/**
 * @}
 */

/* end of file */

```

5.4.2 定义设备常量于设备信息常量

创建配置文件后，我们需要在里面加入设备变量，设备变量只需要定义，不用初始化。设备信息常量定义时初始化，默认填充的信息是一种常用的配置方式，IWDG 设备信息配置如程序清单 5.29 所示。

程序清单 5.29 设备信息

```

/** \brief WWDT 设备信息 */
static const am_zmf2x6_iwdg_devinfo_t __g_iwdg_devinfo = {
    ZMF2X6_IWDG_BASE,          /**< \brief WWDT 寄存器基址 */
    INUM_WWDT,
    NULL,
    0,
    0,
    __zmf2x6_iwdg_plfm_init,    /**< \brief WWDT 平台初始化 */
    __zmf2x6_iwdg_plfm_deinit, /**< \brief 解除 WWDT 平台初始化 */
};

/** \brief WWDT 设备实例 */
static am_zmf2x6_iwdg_dev_t __g_iwdg_dev;

```

5.4.3 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

由于 IWDG 无相关引脚需要配置，时钟的话仅当时钟源选择 LSI 时需要配置一下并且使能 LSI。因此，平台初始化函数和解初始化函数定义如程序清单 5.30 所示。

程序清单 5.30 平台初始化解初始化

```

/** \brief WWDT 平台初始化 */
static void __zmf2x6_iwdg_plfm_init(void)
{
    /* 检查复位是否由看门狗造成的 */
    if (amhw_zmf2x6_rcc_reset_flag() & AMHW_ZMF2X6_RCC_AHB_IWDGRSTF) {
        amhw_zmf2x6_rcc_reset_flag_clear();
    }

    amhw_zmf2x6_rcc_lsi_enable();

    while (amhw_zmf2x6_rcc_lsirdy_read() == AM_FALSE);
}

/** \brief 解除 WWDT 平台初始化 */
static void __zmf2x6_iwdg_plfm_deinit(void)
{
}

```

5.4.4 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在头文件中加入它们的声明，如所示程序清单 5.31。

程序清单 5.31 平台初始化解初始化声明

```

/**
 * \brief IWDT 实例初始化，获得 WDT 标准服务句柄
 *
 * \param 无
 *
 * \return WDT 标准服务句柄，若为 NULL，表明初始化失败
 */
am_wdt_handle_t am_zmf2x6_iwdg_inst_init(void);

/**
 * \brief IWDT 实例解初始化
 *
 * \param[in] handle 通过 am_zmf2x6_iwdg_inst_init() 函数获得的 WDT 标准服务句柄
 *
 * \return 无
 */
void am_zmf2x6_iwdg_inst_deinit(am_wdt_handle_t handle);

```

用户使用时，应该先调用平台初始化函数 `am_zmf2x6_iwdg_inst_init` 函数，然后直接使用 `am_zmf2x6_iwdg_inst_init(&__g_iwdg_dev, &__g_iwdg_devinfo)` 初始化 IWDT，并保存其返回的 `handle`，后续直接使用保存的 `handle` 作为标准接口的第一个参数使用标准接口函数即可。

至此，用户配置文件开发完毕。

5.5 外设例程文件开发

例程文件开发主要分为两个部分，HW 例程和 STD 例程。

调试例程时，应该在 `template` 工程下进行，不需要新建其它工程，只需要将新建的文件加入到工程结构的 `user_code` 下。例程文件会与原先的 `main.c` 文件冲突（因为都存在 `main` 函数），这时，只需要调试时将原先的 `main.c` 不要包含在工程编译中，例程文件开发完毕后，再恢复将 `main.c` 包含在工程中，同时将例程文件移除工程即可。

5.5.1 新建例程文件

为了将外设所有的例程（包括 HW 例程和 STD 例程）统一管理，例程统一存放 `ametal/examples/board/zmf2x6_core/wdg` 路径。HW 层实现与标准层的实现，在文件上的命名，分别采用 `hw` 与 `std` 作为关键字，HW 的例程，命名可以采用 `demo_zmf2x6_core_hw_iwdg.c`，标准层的示例，可以采用 `demo_zmf2x6_core_std_iwdg.c`。

文件创建后，根据代码规范和生成 `doxygen` 文档的注释要求，在新创建的文件中添加相关注释等必要内容，添加后整个文件的内容如程序清单 5.32 所示。

程序清单 5.32 新建例程文件初始内容

```

/*****
 *
 *                               AMetal
 *
 * -----
 *
 *                               innovating embedded platform
 *
 *
 * Copyright (c) 2001-2018 Guangzhou ZHIYUAN Electronics Co., Ltd.

```



```

* All rights reserved.
*
* Contact information:
* web site:    http://www.zlg.cn/
*****/

/**
 * \file
 * \brief IWDG 例程，通过标准接口实现
 *
 * - 实验现象：
 *
 * \par 源代码
 * \snippet demo_zmf2x6_core_std_iwdg.c src_zmf2x6_core_std_iwdg
 *
 * \internal
 * \par Modification history
 * - 1.00 17-04-26   sdy, first implementation
 * \endinternal
 */

/**
 * \addtogroup demo_if_zmf2x6_core_std_iwdg
 * \copydoc demo_zmf2x6_core_std_iwdg.c
 */

/** [src_zmf2x6_core_std_iwdg] */
#include "ametal.h"
#include "am_vdebug.h"
#include "am_zmf2x6_inst_init.h"
#include "demo_std_entries.h"
#include "demo_zmf2x6_core_entries.h"
/** [src_zmf2x6_core_std_iwdg] */

/* end of file */

```

到此例程的框架搭建基本完成，下面我们来继续开发例程的实现内容。

5.5.2 例程内容开发

在前面所搭建的框架下，只需编写实际例程内容就可以。看门狗的例程，主要在于超时时间内喂狗或不喂狗的，是否可以正常的工作，即从现象而言，是否在及时喂狗之后，芯片稳定正常工作，在没有及时喂狗时，芯片是否能够正常的中断或复位。

了解了需要实现的功能，接下来我们来实现看门狗的功能，如程序清单 5.33 与程序清单 5.34 所示，直接调用驱动层实现的函数即可。

1. HW 层实现

程序清单 5.33 HW 层实现例程

```

/** [src_zmf2x6_core_hw_iwdg] */
#include "ametal.h"
#include "am_vdebug.h"
#include "am_zmf2x6.h"
#include "hw/amhw_zmf2x6_rcc.h"
#include "demo_zmf2x6_entries.h"
#include "demo_zmf2x6_core_entries.h"

/**
 * \brief 看门狗超时时间
 *
 * \note 因为看门狗内部时钟 WDTOSC 存在误差，所以__IWDG_TIMEOUT_MS 对应的实际时间
 *       存在误差。
 */
#define __IWDG_TIMEOUT_MS      1500

/**
 * \brief 看门狗喂狗时间，若喂狗时间超过__IWDG_TIMEOUT_MS 的值（大于 5ms 以上），
 *       会产生看门狗事件。
 */
#define __IWDG_FEED_TIME_MS    1000

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_hw_iwdg_entry(void)
{
    AM_DBG_INFO("demo zmf2x6_core hw iwdg!\r\n");

    amhw_zmf2x6_rcc_lsi_enable();

    demo_zmf2x6_hw_iwdg_entry(ZMF2X6_IWDG,
                              __IWDG_TIMEOUT_MS,
                              __IWDG_FEED_TIME_MS);
}
/** [src_zmf2x6_core_hw_iwdg] */

/* end of file */

```

2. 标准层实现

程序清单 5.34 标准层实现内容

```

/** [src_zmf2x6_core_std_iwdg] */

```

```

#include "ametal.h"
#include "am_vdebug.h"
#include "am_zmf2x6_inst_init.h"
#include "demo_std_entries.h"
#include "demo_zmf2x6_core_entries.h"

/**
 * \brief 看门狗超时时间
 *
 * \note 因为看门狗内部时钟 WDTOSC 存在误差，所以__IWDG_TIMEOUT_MS 对应的实际时间
 *       存在误差。
 */
#define __IWDG_TIMEOUT_MS      1500

/**
 * \brief 看门狗喂狗时间，若喂狗时间超过 IWDG_TIMEOUT_MS 的值（大于 5ms 以上），
 *       会产生看门狗事件。
 */
#define __IWDG_FEED_TIME_MS   2000

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_std_iwdg_entry (void)
{
    AM_DBG_INFO("demo zmf2x6_core std iwdg!\r\n");

    demo_std_wdt_entry(am_zmf2x6_iwdg_inst_init(),
                      __IWDG_TIMEOUT_MS,
                      __IWDG_FEED_TIME_MS);
}

/** [src_zmf2x6_core_std_iwdg] */

/* end of file */

```

所有例程开发完毕后，整个外设的开发工作也就圆满结束了。

可以提交相关文件，以完成整个开发任务。提交文件时，不用提交工程文件，只需要提交开发的源文件和头文件即可。

第6章 通用外设驱动开发——CLK

6.1 标准层了解

本章以适配 CLK 外设为例，展示如何开发驱动代码。CLK 标准层仅包含一个文件 am_clk.h，在 ametal\interface 找到该文件打开。

6.1.1 了解相关类型定义

打开该文件，可以看到该文件只定义了 1 个数据类型，如程序清单 6.1 所示。

程序清单 6.1 CLK 标准接口文件函数定义

```
/** \brief 时钟 ID 数据类型*/
typedef int am_clk_id_t;
```

am_clk_id_t 时钟 ID 数据类型本质上是 int 类型，这里取别名是为了后续定义函数入口参数更加直观。

6.1.2 了解标准接口函数

在 am_clk.h 文件相关的类型定义后，就是具体了解标准接口函数，如程序清单 6.2 所示。

程序清单 6.2 CLK 标准接口函数

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

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

/**
 * \brief 获取时钟频率
 */
```

```

* \param[in]   clk_id   时钟 ID (由平台定义), 参见 \ref grp_clk_id
*
* \retval      0         时钟频率为 0, 或获取时钟频率失败
* \retval      正值      时钟频率
*/
int am_clk_rate_get (am_clk_id_t clk_id);

```

1. am_clk_enable()时钟使能函数

该函数用于使能外设时钟，传入的参数为外设时钟 ID 号，并返回函数执行的状态。

2. am_clk_disable()时钟禁能函数

该函数用于禁能外设时钟，传入的参数为外设时钟 ID 号，并返回函数执行的状态。

3. am_clk_rate_get()时钟频率获取函数

该函数用于获取外设时钟的频率，传入的参数为外设时钟 ID 号，返回外设时钟频率。

到此我们就分析完了 CLK 外设标准层数据类型和接口函数，清楚了驱动层需要为标准接口层提供哪些内容。

6.2 HW 层开发

参考 5.2.1 章节，创建对应目录和文件，添加硬件层基本的内容。

6.2.1 添加寄存器列表

HW 层直接与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读时钟外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMF2x6 用户手册的 RCC 章节，可以看到 RCC 章节部分，共有 14 个寄存器，简单描述如下：

- RCC_CR（时钟控制寄存器）：通过该寄存器配置时钟使能和 PLL 倍频等参数；
- RCC_CFGR（时钟配置寄存器）：通过该寄存器来选择时钟源以及设置时钟分频系数；
- RCC_CIR（时钟中断寄存器）：用于使能时钟中断，获取时钟稳定状态；
- RCC_APB2RSTR（APB2 外设复位寄存器）：用于 APB2 外设复位；
- RCC_APB1RSTR（APB1 外设复位寄存器）：用于 APB1 外设复位；
- RCC_AHBENR（AHB 外设时钟使能寄存器）：用于 AHB 外设使能；
- RCC_APB2ENR（APB2 外设时钟使能寄存器）：用于 APB2 外设使能；
- RCC_APB1ENR（APB1 外设时钟使能寄存器）：用于 APB1 外设使能；
- RCC_BDCR（备份域控制寄存器）：用于备份域复位控制、RTC 时钟使能、RTC 时钟源选择；
- RCC_CSR（控制状态寄存器）：复位状态标志，内部低速使能以及状态；
- RCC_AHBRSTR（AHB 外设复位寄存器）：硬件除法、GPIO 复位状态；
- RCC_CFGR2（时钟配置寄存器）：TIM 定时器时钟源选择及分频系数配置；
- RCC_RNG（RNG 寄存器）：RNG 模块控制配置；
- RCC_SYSCFG（系统配置寄存器）：Flash 页擦除大小的设置。

到此 RCC 的各个寄存器及其功能基本阅读完毕，我们也大致了解了各个寄存器功能以及要配置各个功能，其他寄存器该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，程序清单如程序清单 6.3 所示。

程序清单 6.3 寄存器结构体定义

```
/**
 * \brief RCC 寄存器块结构体
 */
typedef struct amhw_zmf2x6_rcc {
    __IO uint32_t cr;           /**< \brief 时钟控制寄存器 */
    __IO uint32_t cfgr;        /**< \brief 时钟配置寄存器 */
    __IO uint32_t cir;         /**< \brief 时钟中断寄存器 */
    __IO uint32_t apb2rstr;     /**< \brief APB2 外设复位寄存器 */
    __IO uint32_t apb1rstr;     /**< \brief APB1 外设复位寄存器 */
    __IO uint32_t ahbenr;       /**< \brief AHB 外设时钟使能寄存器 */
    __IO uint32_t apb2enr;      /**< \brief APB2 外设时钟使能寄存器 */
    __IO uint32_t apb1enr;      /**< \brief APB1 外设时钟使能寄存器 */
    __IO uint32_t bdcrr;        /**< \brief 备份寄存器 */
    __IO uint32_t csr;          /**< \brief 控制状态寄存器 */
    __IO uint32_t ahbrstr;      /**< \brief AHB 外设复位寄存器 */
    __IO uint32_t cfgr2;        /**< \brief 时钟配置寄存器 */
    __IO uint32_t rng;          /**< \brief rng 寄存器 */
    uint32_t reserve[3];        /**< \brief 保留 */
    __IO uint32_t syscfg;       /**< \brief 系统配置寄存器 */
} amhw_zmf2x6_rcc_t;
```

这个小节我们定义了 RCC 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

6.2.2 编写寄存器相关函数

1. RCC_CR（时钟控制寄存器）

通过用户手册，我们可以看到该寄存器有 11 个位域，根据位域的功能编写相应的操作函数，因此它的操作函数比较多。

程序清单 6.4 内部高速时钟使能

```
/**
 * \brief 内部高速时钟使能
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_hsion_enable(void)
```

```
{
    ZMF2X6_RCC->cr |= 1ul;
}
```

程序清单 6.5 内部高速时钟禁能

```
/**
 * \brief 内部高速时钟禁能
 *
 * \return 无
 *
 */
am_static_inline
void amhw_zmf2x6_rcc_hsion_disable(void)
{
    ZMF2X6_RCC->cr &= ~1ul;
}
```

程序清单 6.6 内部高速时钟就绪标志读取

```
/**
 * \brief 内部高速时钟就绪标志读取
 *
 * \retval TRUE : 内部 8MHz 时钟没有就绪
 * \retval FALSE : 内部 8MHz 时钟就绪
 *
 */
am_static_inline
am_bool_t amhw_zmf2x6_rcc_hsiridy_read(void)
{
    return (am_bool_t)((ZMF2X6_RCC->cr & 0x2ul) ? AM_TRUE : AM_FALSE);
}
```

程序清单 6.7 内部高速时钟温度校准使能

```
/**
 * \brief 内部高速时钟(48MHz)温度校准使能
 *
 * \return 无
 *
 */
am_static_inline
void amhw_zmf2x6_rcc_hsitene_enable(void)
{
    ZMF2X6_RCC->cr |= (1ul << 2);
}
```

程序清单 6.8 外部高速时钟使能

```
/**
```

```

* \brief 外部高速时钟(HSE 振荡器)使能
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_hseon_enable (void)
{
    ZMF2X6_RCC->cr |= (1ul << 16);
}

```

程序清单 6.9 外部高速时钟就绪标志读取

```

/**
* \brief 外部高速时钟就绪标志读取
*
* \retval TRUE : 外部时钟没有就绪
* \retval FALSE : 外部时钟就绪
*/
am_static_inline
am_bool_t amhw_zmf2x6_rcc_hserdy_read (void)
{
    return (am_bool_t)((ZMF2X6_RCC->cr & (1ul << 17)) ? AM_TRUE : AM_FALSE);
}

```

程序清单 6.10 外部高速时钟旁路使能

```

/**
* \brief 外部高速时钟旁路使能
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_hsebyp_enable (void)
{
    ZMF2X6_RCC->cr |= (1ul << 18);
}

```

程序清单 6.11 时钟安全系统使能

```

/**
* \brief 时钟安全系统使能
*
* \details 如果外部 8-24MHz 振荡器就绪，时钟监测器开启。
*
* \return 无
*/
am_static_inline

```



```
void amhw_zmf2x6_rcc_ccson_enable (void)
{
    ZMF2X6_RCC->cr |= (1ul << 19);
}
```

程序清单 6.12 PLL 分频系数设置

```
/**
 * \brief PLL 分频系数设置
 *
 * \param[in] div : 分频值
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_plldm_set (uint8_t div)
{
    ZMF2X6_RCC->cr = ((div & 0x7ul) << 20) |
        (ZMF2X6_RCC->cr & ~(0x7ul << 20));
}
```

程序清单 6.13 PLL 倍频系数设置

```
/**
 * \brief PLL 倍频系数设置
 *
 * \param[in] mul : 倍频值
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_plldn_set (uint8_t mul)
{
    ZMF2X6_RCC->cr = ((mul & 0x3ful) << 26) |
        (ZMF2X6_RCC->cr & ~(0x3ful << 26));
}
```

程序清单 6.14 PLL 使能

```
/**
 * \brief PLL 使能
 *
 * \details 当进入待机和停止模式时，该位由硬件清零。
 *          当 PLL 时钟被用作或被选择将要作为系统时钟时，
 *          该位不能被清零。
```

```

*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_pll_enable (void)
{
    ZMF2X6_RCC->cr |= (1ul << 24);
}

```

程序清单 6.15 PLL 禁能

```

/**
* \brief PLL 禁能
*
* \details 当进入待机和停止模式时，该位由硬件清零。
*          当 PLL 时钟被用作或被选择将要作为系统时钟时，
*          该位不能被清零。
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_pll_disable (void)
{
    ZMF2X6_RCC->cr &= ~(1ul << 24);
}

```

程序清单 6.16 PLL 时钟就绪标志读取

```

/**
* \brief PLL 时钟就绪标志读取
*
* \retval TRUE : PLL 未锁定
* \retval FALSE : PLL 锁定
*/
am_static_inline
am_bool_t amhw_zmf2x6_rcc_pllrdy_read (void)
{
    return (am_bool_t)((ZMF2X6_RCC->cr & (1ul << 25)) ? AM_TRUE : AM_FALSE);
}

```

2. RCC_CFGR（时钟配置寄存器）

时钟配置寄存器也有 11 个位域，根据位域的功能编写相应的操作函数。

程序清单 6.17 系统时钟切换

```

/**
* \brief 系统时钟切换
*

```

```

* \param[in] src : 系统时钟源
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_sys_clk_set (amhw_zmf2x6_sys_clk_src src)
{
    ZMF2X6_RCC->cfgr = ((src & 0x3ul) << 0) |
        (ZMF2X6_RCC->cfgr & ~(0x3ul << 0));
}

```

程序清单 6.18 当前系统时钟获取

```

/**
* \brief 当前系统时钟获取
*
* \return 当前系统时钟源
*/
am_static_inline
amhw_zmf2x6_sys_clk_src amhw_zmf2x6_rcc_sys_clk_get (void)
{
    return (amhw_zmf2x6_sys_clk_src)((ZMF2X6_RCC->cfgr >> 2ul) & 0x3ul);
}

```

程序清单 6.19 设置 AHB 预分频

```

/**
* \brief 设置 AHB 预分频
*
* \param[in] div : 预分频值
*
* - 0-7 : 不分频
* - 8 : 2 分频
* - 9 : 4 分频
* - 10 : 8 分频
* - 11 : 16 分频
* - 12 : 64 分频
* - 13 : 128 分频
* - 14 : 256 分频
* - 15 : 512 分频
*
* \return 无
*/
am_static_inline

```

```

void amhw_zmf2x6_rcc_ahb_div_set (uint8_t div)
{

    ZMF2X6_RCC->cfgr = ((div & 0xful) << 4) |
                        (ZMF2X6_RCC->cfgr & ~(0xful << 4));

}

```

程序清单 6.20 设置低速 APB 预分频(APB1)

```

/**
 * \brief 设置 低速 APB 预分频 (APB1)
 *
 * \param[in] div : 预分频值
 *
 * - 0-3 : 不分频
 * - 4 : 2 分频
 * - 5 : 4 分频
 * - 6 : 8 分频
 * - 7 : 16 分频
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_apb1_div_set (uint8_t div)
{

    ZMF2X6_RCC->cfgr = ((div & 0x7ul) << 8) |
                        (ZMF2X6_RCC->cfgr & ~(0x7ul << 8));

}

```

程序清单 6.21 获取低速 APB 预分频(APB1)

```

/**
 * \brief 获取 低速 APB 预分频 (APB1)
 *
 * \return 预分频值
 *
 * - 0-3 : 不分频
 * - 4 : 2 分频
 * - 5 : 4 分频
 * - 6 : 8 分频
 * - 7 : 16 分频
 */
am_static_inline
uint8_t amhw_zmf2x6_rcc_apb1_div_get (void)
{
    return (uint8_t)((ZMF2X6_RCC->cfgr >> 8) & 0x7ul);
}

```

}

程序清单 6.22 设置高速 APB 预分频(APB2)

```

/**
 * \brief 设置 高速 APB 预分频 (APB2)
 *
 * \param[in] div : 预分频值
 *
 * - 0-3 : 不分频
 * - 4 : 2 分频
 * - 5 : 4 分频
 * - 6 : 8 分频
 * - 7 : 16 分频
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_apb2_div_set (uint8_t div)
{
    ZMF2X6_RCC->cfgr = ((div & 0x7ul) << 11) |
        (ZMF2X6_RCC->cfgr & ~(0x7ul << 11));
}

```

程序清单 6.23 获取高速 APB 预分频(APB2)

```

/**
 * \brief 获取 高速 APB 预分频 (APB2)
 *
 * \return 预分频值
 *
 * - 0-3 : 不分频
 * - 4 : 2 分频
 * - 5 : 4 分频
 * - 6 : 8 分频
 * - 7 : 16 分频
 *
 */
am_static_inline
uint8_t amhw_zmf2x6_rcc_apb2_div_get (void)
{
    return (uint8_t)((ZMF2X6_RCC->cfgr >> 11) & 0x7ul);
}

```

程序清单 6.24 USB 的 48M 时钟源选择

```

/**
 * \brief USB 的 48M 时钟源选择
 * \param[in] src : usb 输入时钟源

```

```

* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_usb_clk_set (amhw_zmf2x6_usb_clk_src src)
{
    ZMF2X6_RCC->cfgr = (ZMF2X6_RCC->cfgr & ~(1 << 15)) | (src << 15);
}

```

程序清单 6.25 PLL 输入时钟选择

```

/**
* \brief PLL 输入时钟选择
*
* \param[in] src : PLL 输入时钟源
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_pll_clk_set (amhw_zmf2x6_pll_clk_src src)
{
    ZMF2X6_RCC->cfgr = ((src & 0x1ul) << 16) |
        (ZMF2X6_RCC->cfgr & ~(0x1ul << 16));
}

```

程序清单 6.26 当前 PLL 输入时钟获取

```

/**
* \brief 当前 PLL 输入时钟获取
*
* \return 当前 PLL 输入时钟源
*/
am_static_inline
amhw_zmf2x6_pll_clk_src amhw_zmf2x6_rcc_pll_clk_get (void)
{
    return (amhw_zmf2x6_pll_clk_src)((ZMF2X6_RCC->cfgr >> 16ul) & 0x1ul);
}

```

程序清单 6.27 选择 HSE 分频器作为 PLL 输入

```

/**
* \brief 选择 HSE 分频器作为 PLL 输入
*
* \param[in] div : 分频值
*
* - 0 : 不分频
* - 1 : 2 分频
*

```

```

* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_hseforpll_set (uint8_t div)
{
    ZMF2X6_RCC->cfgr = ((div & 0x1ul) << 17) |
        (ZMF2X6_RCC->cfgr & ~(0x1ul << 17));
}

```

程序清单 6.28 选择 USB 预分频

```

/**
* \brief 选择 USB 预分频
*
* \param[in] div : 分频值
*
* - 0 : PLL 时钟直接作为 USB 时钟
* - 1 : PLL 时钟 2 分频作为 USB 时钟
* - 2 : PLL 时钟 3 分频作为 USB 时钟
* - 3 : PLL 时钟 4 分频作为 USB 时钟
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_pllforusb_set (uint8_t div)
{
    ZMF2X6_RCC->cfgr = ((div & 0x3ul) << 22) |
        (ZMF2X6_RCC->cfgr & ~(0x3ul << 22));
}

```

程序清单 6.29 USB 预分频获取

```

/**
* \brief USB 预分频获取
*
* \retval 0 : PLL 时钟直接作为 USB 时钟
* \retval 1 : PLL 时钟 2 分频作为 USB 时钟
* \retval 2 : PLL 时钟 3 分频作为 USB 时钟
* \retval 3 : PLL 时钟 4 分频作为 USB 时钟
*/
am_static_inline
uint8_t amhw_zmf2x6_rcc_pllforusb_get (void)
{
    return (uint8_t)((ZMF2X6_RCC->cfgr >> 22ul) & 0x3ul);
}

```

程序清单 6.30 时钟输出选择

```

/**
 * \brief 微控制器时钟输出选择
 *
 * \details 1.该时钟输出在启动和切换 MCO 时钟源时可能会被截断,
 *          2.在系统时钟作为输出至 MCO 管脚时, 请保证输出时钟频
 *          率不超过 50MHz (IO 口最高频率)
 *
 * \param[in] src : 时钟源
 *
 * - 0 : 无输出
 * - 1 : HSI/DIV4
 * - 2 : LSI 时钟输出
 * - 3 : LSE
 * - 4 : 系统时钟 (SYSCLK) 输出
 * - 5 : HSI 时钟输出
 * - 6 : HSE 时钟输出
 * - 7 : PLL 时钟 2 分频后输出
 * - 8 : PLL 时钟 12 分频后输出
 * - 9 : PLL 时钟 24 分频后输出
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_mco_src_set (uint8_t src)
{
    ZMF2X6_RCC->cfgr = ((src & 0xFul) << 24ul) |
        (ZMF2X6_RCC->cfgr & ~(0xFul << 24ul));
}

```

3. RCC_CIR (时钟中断寄存器)

时钟中断寄存器包括时钟中断使能、禁能, 中断状态标志, 以及清除标志。根据需要的功能编写相应的操作函数。

为了函数参数更加直观和明确, 采用枚举方式将参数列出, 如程序清单 6.31 所示。

程序清单 6.31 时钟中断标志

```

/**
 * \brief 时钟中断标志
 */
typedef enum {
    AMHW_ZMF2X6_CLKINT_INDEX_LSI = 0, /**< \brief LSI 标志 */
    AMHW_ZMF2X6_CLKINT_INDEX_LSE = 1, /**< \brief LSI 标志 */
    AMHW_ZMF2X6_CLKINT_INDEX_HSI = 2, /**< \brief HSI 标志 */
    AMHW_ZMF2X6_CLKINT_INDEX_HSE,    /**< \brief HSE 标志 */
    AMHW_ZMF2X6_CLKINT_INDEX_PLL,    /**< \brief PLL 标志 */
    AMHW_ZMF2X6_CLKINT_INDEX_CSS = 7, /**< \brief CSS(时钟安全系统中断)标志 */
}

```



```
}amhw_zmf2x6_clk_int_index;
```

程序清单 6.32 就绪中断使能

```
/**
 * \brief 就绪中断使能
 *
 * \param[in] index : 就绪标志
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_clkint_enable (amhw_zmf2x6_clk_int_index index)
{
    ZMF2X6_RCC->cir |= (((1<<index) & 0x1Ful) << 8);
}
```

程序清单 6.33 就绪中断禁能

```
/**
 * \brief 就绪中断禁能
 *
 * \param[in] index : 就绪标志
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_clkint_disable (amhw_zmf2x6_clk_int_index index)
{
    ZMF2X6_RCC->cir &= ~(((1<<index) & 0x1Ful) << 8);
}
```

程序清单 6.34 中断标志获取

```
/**
 * \brief 中断标志获取
 *
 * \return 中断标志(通过类型 amhw_zmf2x6_clk_int_index 进行解析)
 */
am_static_inline
uint8_t amhw_zmf2x6_rcc_clkint_flag_read (void)
{
    return (uint8_t)(ZMF2X6_RCC->cir & 0x9Ful);
}
```

程序清单 6.35 清除中断标志

```
/**
```

```

* \brief 清除中断标志
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_clkint_flag_clear (amhw_zmf2x6_clk_int_index index)
{
    ZMF2X6_RCC->cir |= (((1<<index) & 0x9Ful) << 16);
}

```

4. RCC_APB2RSTR（APB2 外设复位寄存器）

APB2 外设复位寄存器控制 APB2 总线上的外设复位和停止复位。由于控制多个外设，为了明确函数参数，采用枚举方式列出可用参数，接着编写 APB2 外设复位和停止复位的函数。

程序清单 6.36 APB2 外设枚举

```

/**
* \brief APB2 外设枚举
*/
typedef enum {
    AMHW_ZMF2X6_RCC_APB2_SYSCFG = 0, /**< \brief 系统配置寄存器 */
    AMHW_ZMF2X6_RCC_APB2_ADC1    = 9, /**< \brief ADC1 接口 */
    AMHW_ZMF2X6_RCC_APB2_TIM1    = 11, /**< \brief TIM1 定时器 */
    AMHW_ZMF2X6_RCC_APB2_SPI1    = 12, /**< \brief SPI1 */
    AMHW_ZMF2X6_RCC_APB2_UART1   = 14, /**< \brief UART1 */
    AMHW_ZMF2X6_RCC_APB2_CPT     = 15, /**< \brief 比较器 */
    AMHW_ZMF2X6_RCC_APB2_TIM14   = 16, /**< \brief TIM14 */
    AMHW_ZMF2X6_RCC_APB2_TIM16   = 17, /**< \brief TIM16 */
    AMHW_ZMF2X6_RCC_APB2_TIM17   = 18, /**< \brief TIM17 */
    AMHW_ZMF2X6_RCC_APB2_DBGMCU  = 22, /**< \brief DBGMCU */
}amhw_zmf2x6_apb2_peripheral;

```

程序清单 6.37 复位 APB2 外设

```

/**
* \brief 复位 APB2 外设
*
* \param[in] reset : 需要复位的外设
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_apb2_reset (amhw_zmf2x6_apb2_peripheral reset)
{
    ZMF2X6_RCC->apb2rstr |= (1ul << reset);
}

```

程序清单 6.38 停止复位 APB2 外设

```

/**
 * \brief 停止复位 APB2 外设
 *
 * \param[in] reset : 需要复位的外设
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_apb2_reset_stop (amhw_zmf2x6_apb2_peripheral reset)
{
    ZMF2X6_RCC->apb2rstr &= ~(1ul << reset);
}

```

5. RCC_APB1RSTR (APB1 外设复位寄存器)

APB1 外设复位寄存器控制 APB1 总线上的外设复位和停止复位。由于控制多个外设，为了明确函数参数，采用枚举方式列出可用参数，接着编写 APB1 外设复位和停止复位的函数。

程序清单 6.39 APB1 外设枚举

```

/**
 * \brief APB1 外设枚举
 */
typedef enum {
    AMHW_ZMF2X6_RCC_APB1_TIM2    = 0,    /**< \brief TIM2 定时器 */
    AMHW_ZMF2X6_RCC_APB1_TIM3    = 1,    /**< \brief TIM3 定时器 */
    AMHW_ZMF2X6_RCC_APB1_WWDG    = 11,   /**< \brief WWDG 窗口看门狗 */
    AMHW_ZMF2X6_RCC_APB1_SPI2    = 14,   /**< \brief SPI2 */
    AMHW_ZMF2X6_RCC_APB1_UART2   = 17,   /**< \brief UART2 */
    AMHW_ZMF2X6_RCC_APB1_I2C1    = 21,   /**< \brief I2C1 */
    AMHW_ZMF2X6_RCC_APB1_USB     = 23,   /**< \brief USB */
    AMHW_ZMF2X6_RCC_APB1_CAN     = 25,   /**< \brief CAN */
    AMHW_ZMF2X6_RCC_APB1_CSM     = 26,   /**< \brief CSM */
    AMHW_ZMF2X6_RCC_APB1_CRs     = 27,   /**< \brief CRs */
    AMHW_ZMF2X6_RCC_APB1_PWR     = 28,   /**< \brief 电源接口 */
}amhw_zmf2x6_apb1_peripheral;

```

程序清单 6.40 复位 APB1 外设

```

/**
 * \brief 复位 APB1 外设
 *
 * \param[in] reset : 需要复位的外设
 *
 * \return 无

```

```

*/
am_static_inline
void amhw_zmf2x6_rcc_apb1_reset (amhw_zmf2x6_apb1_peripheral reset)
{
    ZMF2X6_RCC->apb1rstr |= (1ul << reset);
}

```

程序清单 6.41 停止复位 APB1 外设

```

/**
 * \brief 停止复位 APB1 外设
 *
 * \param[in] reset : 需要复位的外设
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_apb1_reset_stop (amhw_zmf2x6_apb1_peripheral reset)
{
    ZMF2X6_RCC->apb1rstr &= ~(1ul << reset);
}

```

6. RCC_AHBRSTR (AHB 外设复位寄存器)

AHB 外设复位寄存器控制 AHB 总线上的外设复位和停止复位。由于控制多个外设，为了明确函数参数，采用枚举方式列出可用参数，接着编写 AHB 外设复位和停止复位的函数。

程序清单 6.42 AHB 外设枚举

```

/**
 * \brief AHB 外设枚举
 */
typedef enum {
    AMHW_ZMF2X6_RCC_AHB_DMA = 0,    /**< \brief DMA */
    AMHW_ZMF2X6_RCC_AHB_SRAM = 2,    /**< \brief SRAM */
    AMHW_ZMF2X6_RCC_AHB_FLITF = 4,    /**< \brief FLITF */
    AMHW_ZMF2X6_RCC_AHB_CRC = 7,     /**< \brief CRC */
    AMHW_ZMF2X6_RCC_AHB_GPIOA = 17,   /**< \brief GPIOA */
    AMHW_ZMF2X6_RCC_AHB_GPIOB = 18,   /**< \brief GPIOB */
    AMHW_ZMF2X6_RCC_AHB_GPIOC = 19,   /**< \brief GPIOC */
    AMHW_ZMF2X6_RCC_AHB_GPIOD = 20,   /**< \brief GPIOD */
    AMHW_ZMF2X6_RCC_AHB_HWDIV = 26,   /**< \brief HWDIV */
}amhw_zmf2x6_ahb_peripheral;

```

程序清单 6.43 复位 AHB 外设

```

/**
 * \brief 复位 AHB 外设
 *
 * GPIOA 后数据才可复位

```

```

*
* \param[in] reset : 需要复位的外设
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_ahb_reset (amhw_zmf2x6_ahb_peripheral reset)
{
    ZMF2X6_RCC->ahbrstr |= (1ul << reset);
}

```

程序清单 6.44 停止复位 AHB 外设

```

/**
* \brief 停止复位 AHB 外设
*
*          GPIOA 后数据才可复位
*
* \param[in] reset : 需要复位的外设
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_ahb_reset_stop (amhw_zmf2x6_ahb_peripheral reset)
{
    ZMF2X6_RCC->ahbrstr &= ~(1ul << reset);
}

```

7. RCC_APB1ENR (APB1 外设时钟使能寄存器)

APB1 外设时钟使能寄存器控制 APB1 总线上的外设的使能和禁能。因此编写两个函数，一个用于外设使能，一个用于外设禁能。

程序清单 6.45 使能 APB1 外设

```

/**
* \brief 使能 APB1 外设
*
* \param[in] peri : 需要使能的外设
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_apb1_enable (amhw_zmf2x6_apb1_peripheral peri)
{
    ZMF2X6_RCC->apb1enr |= (1ul << peri);
}

```

程序清单 6.46 禁能 APB1 外设

```

/**
 * \brief 禁能 APB1 外设
 *
 * \param[in] peri : 需要禁能的外设
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_apb1_disable (amhw_zmf2x6_apb1_peripheral peri)
{
    ZMF2X6_RCC->apb1enr &= ~(1ul << peri);
}

```

8. RCC_APB2ENR (APB2 外设时钟使能寄存器)

APB2 外设时钟使能寄存器控制 APB2 总线上的外设的使能和禁能。因此编写两个函数，一个用于外设使能，一个用于外设禁能。

程序清单 6.47 使能 APB2 外设

```

/**
 * \brief 使能 APB2 外设
 *
 * \param[in] peri : 需要使能的外设
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_apb2_enable (amhw_zmf2x6_apb2_peripheral peri)
{
    ZMF2X6_RCC->apb2enr |= (1ul << peri);
}

```

程序清单 6.48 禁能 APB2 外设

```

/**
 * \brief 禁能 APB2 外设
 *
 * \param[in] peri : 需要禁能的外设
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_apb2_disable (amhw_zmf2x6_apb2_peripheral peri)
{
    ZMF2X6_RCC->apb2enr &= ~(1ul << peri);
}

```

9. RCC_AHBENR (AHB 外设时钟使能寄存器)

AHB 外设时钟使能寄存器控制 AHB 总线上的外设的使能和禁能。因此编写两个函数，一个用于外设使能，一个用于外设禁能。

程序清单 6.49 使能 AHB 外设

```
/**
 * \brief 使能 AHB 外设
 *
 * \param[in] peri : 需要使能的外设
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_ahb_enable (amhw_zmf2x6_ahb_peripheral peri)
{
    ZMF2X6_RCC->ahbenr |= (1ul << peri);
}
```

程序清单 6.50 禁能 AHB 外设

```
/**
 * \brief 禁能 AHB 外设
 *
 * \param[in] peri : 需要禁能的外设
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_ahb_disable (amhw_zmf2x6_ahb_peripheral peri)
{
    ZMF2X6_RCC->ahbenr &= ~(1ul << peri);
}
```

10. RCC_CSR（控制状态寄存器）

控制状态寄存器包括复位标志状态、复位标志清除、内部低速时钟的配置。根据位域功能，编写对应的操作函数。

程序清单 6.51 使能 LSI

```
/**
 * \brief 使能 LSI
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_lsi_enable (void)
{
```

```
ZMF2X6_RCC->csr |= (1ul << 0);
}
```

程序清单 6.52 禁能 LSI

```
/**
 * \brief 禁能 LSI
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_lsi_disable(void)
{
    ZMF2X6_RCC->csr &= ~(1ul << 0);
}
```

程序清单 6.53 内部低速时钟就绪标志读取

```
/**
 * \brief 内部低速时钟就绪标志读取
 *
 * \retval TRUE : 内部 40kHz 振荡器时钟就绪
 * \retval FALSE : 内部 40kHz 振荡器时钟未就绪
 */
am_static_inline
am_bool_t amhw_zmf2x6_rcc_lsirdy_read(void)
{
    return (am_bool_t)((ZMF2X6_RCC->csr & (1ul << 1)) ? AM_TRUE : AM_FALSE);
}
```

程序清单 6.54 获取复位标志

```
/**
 * \brief 获取复位标志
 *
 * \return 复位标志(amhw_zmf2x6_reset_flag_t)
 */
am_static_inline
uint32_t amhw_zmf2x6_rcc_reset_flag(void)
{
    return (ZMF2X6_RCC->csr & 0x7dc00000);
}
```

程序清单 6.55 清除复位标志

```
/**
 * \brief 清除复位标志
 *
```



```

* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_reset_flag_clear (void)
{
    ZMF2X6_RCC->csr |= (1ul << 24);
}

```

程序清单 6.56 使能 pvd_reset

```

/**
* \brief 使能 pvd_reset
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_pvd_reset_enable (void)
{
    ZMF2X6_RCC->csr |= (1ul << 6);
}

```

程序清单 6.57 禁能 pvd_reset

```

/**
* \brief 禁能 pvd_reset
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_pvd_reset_disable (void)
{
    ZMF2X6_RCC->csr &= ~(1ul << 6);
}

```

程序清单 6.58 使能 lockup_reset

```

/**
* \brief 使能 lockup_reset
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_lockup_reset_enable (void)
{
    ZMF2X6_RCC->csr |= (1ul << 7);
}

```

程序清单 6.59 禁能 lockup_reset

```

/**
 * \brief 禁能 lockup_reset
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_lockup_reset_disable(void)
{
    ZMF2X6_RCC->csr &= ~(1ul << 7);
}

```

11. RCC_CFGR2（时钟配置寄存器）

时钟配置寄存器包括 `timadv_clk` 预分频和复位标志清除、内部低速时钟的配置。根据位域功能，编写对应的操作函数。

程序清单 6.60 设置 `timadv_clk` 预分频

```

/**
 * \brief 设置 timadv_clk 预分频
 *
 * \param[in] div : 预分频值
 *
 * - 0-3 : 不分频
 * - 4 : 2 分频
 * - 5 : 4 分频
 * - 6 : 8 分频
 * - 7 : 16 分频
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_timadv_div_set(uint8_t div)
{
    ZMF2X6_RCC->cfgr2 = ((div & 0x7ul) << 11) |
        (ZMF2X6_RCC->cfgr & ~(0x7ul << 11));
}

```

程序清单 6.61 TIMADV 时钟选择

```

/**
 * \brief TIMADV 时钟选择
 *
 * \param[in] sel : 0: PCLK2 2 分频 1: timadv_clk
 *
 * \return 无
 */
am_static_inline

```

```
void amhw_zmf2x6_rcc_timadv_clk_sel (uint8_t sel)
{
    ZMF2X6_RCC->cfgr2 = (ZMF2X6_RCC->cfgr2 & ~(0x1)) | sel;
}
```

12. RCC_SYSCFG（系统配置寄存器）

系统配置寄存器主要配置 Flash 擦除大小、写 Flash 时是否检查已擦除。根据位域功能，编写对应的操作函数。

程序清单 6.62 页擦除时擦除的大小

```
/**
 * \brief Flash 页擦除时擦除的大小
 *
 * \param[in] sel : 0: 512 字节    1: 1k 字节
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_flash_page_clean_num (uint8_t sel)
{
    ZMF2X6_RCC->syscfg = (ZMF2X6_RCC->syscfg & ~(0x2)) | sel;
}
```

程序清单 6.63 写 Flash 时是否检查 flash 内数据为 0xff

```
/**
 * \brief 写 Flash 时是否检查 flash 内数据为 0xff
 *
 * \param[in] sel : 0: 不检查    1: 检查
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_write_flash_check (uint8_t sel)
{
    ZMF2X6_RCC->syscfg = (ZMF2X6_RCC->syscfg & ~(0x1)) | sel;
}
```

13. RCC_BDCR（备份域控制寄存器）

备份域控制寄存器主要配置 RTC 模块，包括 RTC 使能、RTC 时钟源选择。根据位域功能，编写对应的操作函数。

程序清单 6.64 备份域软件复位

```
/**
 * \brief 备份域软件复位
 *
 * \param[in] flag :    AM_TRUE: 复位    AM_FALSE: 停止复位

```

```

*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_bdrst_set (am_bool_t flag)
{
    ZMF2X6_RCC->bdcrr = (ZMF2X6_RCC->bdcrr & ~(0x1 << 16)) | (flag << 16);
}

```

程序清单 6.65 RTC 时钟使能

```

/**
* \brief RTC 时钟使能
*
* \param[in] flag :    AM_TRUE: 时钟开启      AM_FALSE: 时钟关闭
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_rtcclock_enable (am_bool_t flag)
{
    ZMF2X6_RCC->bdcrr = (ZMF2X6_RCC->bdcrr & ~(0x1 << 15)) | (flag << 15);
}

```

程序清单 6.66 RTC 时钟源选择

```

/**
* \brief RTC 时钟源
*/
typedef enum {
    AMHW_ZMF2X6_RTCCLK_SRC_LSE = 1,  /**< \brief LSE 作为 RTC 时钟 */
    AMHW_ZMF2X6_RTCCLK_SRC_LSI = 2,  /**< \brief LSI 作为 RTC 时钟 */
    AMHW_ZMF2X6_RTCCLK_SRC_HSE = 3    /**< \brief HSE DIV128 作为 RTC 时钟 */
}amhw_zmf2x6_rtc_clk_src;

/**
* \brief RTC 时钟源选择
*
* \param[in]
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rcc_rtc_clk_src_sel (am_bool_t src)
{
    ZMF2X6_RCC->bdcrr = (ZMF2X6_RCC->bdcrr & ~(0x3 << 8)) | (src << 8);
}

```

}

14. RCC_RNG（RNG 寄存器）

RNG 寄存器用于配置 RNG 模块，包括 RNG 使能、随机数读取、随机数生成的状态。根据位域功能，编写对应的操作函数。

程序清单 6.67 RNG 模块使能

```
/**
 * \brief RNG 模块使能
 *
 * \param[in] flag:    AM_TRUE: 使能    AM_FALSE: 禁能
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_rng_enable(am_bool_t flag)
{
    ZMF2X6_RCC->rng = (ZMF2X6_RCC->rng & ~(0x1 << 0)) | (flag << 0);
}
```

程序清单 6.68 RNG 模块加载种子

```
/**
 * \brief RNG 模块加载种子
 *
 * \param[in] flag:    AM_TRUE: 加载    AM_FALSE: 不加载
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rcc_rng_load_sd_cfg(am_bool_t flag)
{
    ZMF2X6_RCC->rng = (ZMF2X6_RCC->rng & ~(0x1 << 1)) | (flag << 1);
}
```

程序清单 6.69 RNG 随机数生成完成标志

```
/**
 * \brief RNG 随机数生成完成标志
 *
 * \return AM_TRUE: 随机数生成完成 AM_FALSE: 随机数生成未完成
 */
am_static_inline
am_bool_t amhw_zmf2x6_rcc_rng_done_flag_get(void)
{
    return ((ZMF2X6_RCC->rng >> 2) & 0x1);
}
```

程序清单 6.70 RNG 随机数

```

/**
 * \brief RNG 随机数
 *
 * \return 随机数
 */
am_static_inline
uint16_t amhw_zmf2x6_rcc_rng_data_get(void)
{
    return ((ZMF2X6_RCC->rng >> 16) & 0xffff);
}

```

至此，已完成 RCC 相关寄存器所有操作函数的编写。

6.2.3 添加指向寄存器块的指针

为了方便用户或驱动层使用，将 RCC 外设基地址定义为 RCC 外设结构体指针，后续使用硬件接口层时直接使用宏即可。在 AMetal 中，由于这些内容与具体型号的芯片相关，这样的宏统一定义到 zmf2x6_periph_map.h 文件中，如程序清单 6.71 所示。

程序清单 6.71 外设中加入 RCC 结构体指针定义

```

/** \brief 系统配置(SYSCFG)寄存器块指针 */
#define ZMF2X6_SYSCFG ((amhw_zmf2x6_syscfg_t *)ZMF2X6_SYSCFG_BASE)

/** \brief EXTI 寄存器块指针 */
#define ZMF2X6_EXTI ((amhw_zmf2x6_exti_t *)ZMF2X6_EXTI_BASE)

/** \brief RCC 寄存器块指针 */
#define ZMF2X6_RCC ((amhw_zmf2x6_rcc_t *)ZMF2X6_RCC_BASE)

/** \brief FLASH 寄存器块指针 */
#define ZMF2X6_FLASH ((amhw_zmf2x6_flash_t *)ZMF2X6_FLASHCTR_BASE)

/** \brief 串口(UART1)寄存器块指针 */
#define ZMF2X6_UART1 ((amhw_zmf2x6_uart_t *)ZMF2X6_UART1_BASE)

/** \brief 串口(UART2)寄存器块指针 */
#define ZMF2X6_UART2 ((amhw_zmf2x6_uart_t *)ZMF2X6_UART2_BASE)

```

后续用户或驱动层使用 HW 层接口时，可直接使用 ZMF2X6_RCC 作为参数。至此，整个 HW 层的开发工作就结束了，如果后续开发中发现缺少某些函数，还可以再添加。

6.3 驱动层开发

参考 5.3.1 章节，新建 am_zmf2x6_clk.h 和 am_zmf2x6_clk.c 驱动文件，搭建相应的文件框架，并按规则放到对应的路径下。

6.3.1 编写结构性程序

驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

定义 CLK 外设设备信息结构体如程序清单 6.71 所示。后续可以根据实际情况修改。

程序清单 6.72 CLK 设备信息结构体定义

```
/**
 * \brief CLK 设备信息参数结构体
 */
typedef struct am_zmf2x6_clk_devinfo {

    /**
     * \brief HSEOSC 外部晶振频率
     *
     * 如果 pllin_src 选择 AMHW_ZMF2X6_PLLCLK_HSE 则 PLLIN = hse_osc
     */
    uint32_t hse_osc;

    /** \brief
     * PLL 时钟源选择
     * -# AMHW_ZMF2X6_PLLCLK_HSI_DIV4 : HSI 振荡器 4 分频作为 PLL 输入时钟
     * -# AMHW_ZMF2X6_PLLCLK_HSE : HSE 作为 PLL 输入时钟
     */
    amhw_zmf2x6_pll_clk_src pllin_src;

    /**
     * \brief PLL 倍频系数，可选 1-64
     *
     *  $PLLOUT = PLLIN * pll\_mul / pll\_div$ 
     */
    uint8_t pll_mul;

    /**
     * \brief PLL 分频系数，可选 1-8
     *
     *  $PLLOUT = PLLIN * pll\_mul / pll\_div$ 
     */
    uint8_t pll_div;
```

```

/** \brief USB 分频系数，USBCLK = PLLOUT / (usb_div + 1),建议配置成 48Mhz。usb_div 范围(0~3)*/
uint8_t usb_div;

/**
 * \brief TIM_ADV 分频系数，USBCLK = PLLOUT / (tim_div + 1),tim_div(0~7)
 *      0-3 :   不分频   TIM_ADV = PLLOUT / 2^(tim_div-3)
 */
uint8_t tim_div;

/**
 * \brief AHB 分频系数，AHBCLK = PLLOUT / DIV,AHB 最大频率为 48Mhz
 *
 *      ahb_div |  DIV
 *      -----
 *      0-7 |    1
 *      8   |    2
 *      9   |    4
 *      10  |    8
 *      11  |   16
 *      12  |   64
 *      13  |  128
 *      14  |  256
 *      15  |  512
 */
uint8_t ahb_div;

/**
 * \brief APB1 分频系数，APB1CLK = AHBCLK / (2 ^ apb1_div)
 *      APB1 最大频率为 48Mhz
 */
uint8_t apb1_div;

/**
 * \brief APB2 分频系数，APB2CLK = AHBCLK / (2 ^ apb2_div)
 *      APB2 最大频率为 48Mhz
 */
uint8_t apb2_div;

/** \brief 平台初始化函数，配置引脚等工作 */
void (*pfn_plfm_init)(void);

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

```



```
} am_zmf2x6_clk_devinfo_t;
```

● 设备结构体定义

由于 CLK 设备需要提供标准的 CLK 服务，包括时钟使能、时钟频率获取等。其次，为了在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 6.73 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 6.73 CLK 设备结构体定义

```
/**
 * \brief CLK 设备结构体
 */
typedef struct am_zmf2x6_clk_dev {

    /** \brief 指向 CLK 设备信息的指针 */
    const am_zmf2x6_clk_devinfo_t *p_devinfo;

    /** \brief PLL 输入频率 */
    uint32_t pllin_clk;

    /** \brief PLL 输出频率 */
    uint32_t pllout_clk;

    /** \brief AHB 时钟频率 */
    uint32_t ahb_clk;

    /** \brief APB1 时钟频率 */
    uint32_t apb1_clk;

    /** \brief APB2 时钟频率 */
    uint32_t apb2_clk;

    /** \brief USB 时钟频率 */
    uint32_t usb_clk;

} am_zmf2x6_clk_dev_t;
```

2. 搭建函数框架

函数主要分为两部分一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于 CLK 驱动层需要提供初始化函数之外，还需要实现标准接口层函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为时钟初始化的状态。示例代码如程序清单 6.74 所示。

程序清单 6.74 驱动层函数定义示例

```
/**
 * \brief CLK 初始化
 */
int am_zmf2x6_clk_init (am_zmf2x6_clk_dev_t      *p_dev,
                       const am_zmf2x6_clk_devinfo_t *p_devinfo)
{
    return AM_OK;
}
```

同时为了用户能够使用，还需要将这两个函数的声明放入到相应的头文件中。示例代码如程序清单 6.75 所示。

程序清单 6.75 驱动函数声明

```
/**
 * \brief CLK 初始化
 *
 * \param[in] p_dev      : 指向 CLK 设备的指针
 * \param[in] p_devinfo : 指向 CLK 设备信息的指针
 *
 * \retval AM_OK : 操作成功
 */
int am_zmf2x6_clk_init (am_zmf2x6_clk_dev_t      *p_dev,
                       const am_zmf2x6_clk_devinfo_t *p_devinfo);
```

- 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 3 个函数，根据标准接口层函数声明，我们需要在.c 文件中定义 3 个函数。示例代码如程序清单 6.76 所示。

程序清单 6.76 提供给标准层函数定义

```
/**
 * \brief 获取时钟频率
 */
int am_clk_rate_get (am_clk_id_t clk_id)
{
    int clk = 0;
    int peri, num;

    num = (clk_id & 0xff00) >> 8;
    peri = clk_id & 0xff;

    return clk;
}

/**
```

```

    * \brief 使能指定的系统或外设时钟
    */
int am_clk_enable (am_clk_id_t clk_id)
{
    int peri, num;

    num = (clk_id & 0xff00) >> 8;
    peri = clk_id & 0xff;

    return AM_OK;
}

/**
    * \brief 禁能指定的系统或外设时钟
    */
int am_clk_disable (am_clk_id_t clk_id)
{
    int peri, num;

    num = (clk_id & 0xff00) >> 8;
    peri = clk_id & 0xff;

    return AM_OK;
}

```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。

6.3.2 驱动函数实现

通过上面的函数框架搭建，共计 4 个函数，3 个功能性函数面向标准层，1 个初始化函数面向用户，下面逐步实现。

1. am_zmf2x6_clk_init 函数

该函数主要完成两个功能：

- a) 初始化设备中的标准服务，将成员赋值。
- b) 初始化硬件相关设置，根据硬件配置初始化时钟，如程序清单 6.77 所示。

程序清单 6.77 am_zmf2x6_clk_init 函数示例代码

```

/**
    * \brief CLK 初始化
    */
int am_zmf2x6_clk_init (am_zmf2x6_clk_dev_t *p_dev,
                        const am_zmf2x6_clk_devinfo_t *p_devinfo)
{
    uint8_t i = 0, div = 1;

    if (p_dev == NULL || p_devinfo == NULL) {

```

```

        return -AM_EINVAL;
    }

    p_dev->p_devinfo = p_devinfo;
    __gp_dev         = p_dev;

    /* 使用内部 hsi 时钟 */
    amhw_zmf2x6_rcc_hSION_enable();

    /* 等待其稳定 */
    while (amhw_zmf2x6_rcc_hSIRDY_read() == AM_FALSE);

    /* 复位 SW, HPRE, PPRE1, PPRE2, and MCO bits */
    ZMF2X6_RCC->cfgr &= (uint32_t)0xF8FFC00C;

    /* 复位禁能 HSEON, CSSON and PLLON bits */
    ZMF2X6_RCC->cr &= (uint32_t)0xFE6FFFFF;

    if (p_devinfo->pllsrc == AMHW_ZMF2X6_PLLCLK_HSE) {

        /* CLK 平台初始化, 配置时钟引脚 */
        if (p_devinfo->pfn_plfm_init) {
            p_devinfo->pfn_plfm_init();
        }
        __pll_hse_init();
    } else {
        __pll_hsi_init();
    }

    p_dev->pllout_clk = p_dev->pll_clk *
                        p_devinfo->pll_mul /
                        p_devinfo->pll_div;

    /* 计算 AHB 时钟频率 */
    if (p_devinfo->ahb_div < 12) {
        for (i = 0; i < p_devinfo->ahb_div - 7; i++) {
            div = div << 1;
        }
    } else {
        div = 64;
        for (i = 0; i < (p_devinfo->ahb_div & 0xf) - 12; i++) {
            div = div << 1;
        }
    }
}

```

```

p_dev->ahb_clk = p_dev->pllout_clk / div;

amhw_zmf2x6_rcc_usb_clk_set(AMHW_ZMF2X6_USBCLK_PLL_48M);
p_dev->usb_clk = p_dev->pllout_clk / (p_devinfo->usb_div + 1);

/* 计算 APB1 时钟频率 */
div = 1;
for (i = 0; i < (p_devinfo->apb1_div & 0x7); i++) {
    div = div << 1;
}
p_dev->apb1_clk = p_dev->ahb_clk / div;

/* 计算 APB2 时钟频率 */
div = 1;
for (i = 0; i < (p_devinfo->apb2_div & 0x7); i++) {
    div = div << 1;
}
p_dev->apb2_clk = p_dev->ahb_clk / div;

ZMF2X6_FLASH->acr |= 0x30;

amhw_zmf2x6_flash_latency_set(ZMF2X6_FLASH,
                               ((p_dev->pllout_clk - 1) / 24000000));

amhw_zmf2x6_rcc_ahb_div_set(p_devinfo->ahb_div);

amhw_zmf2x6_rcc_apb2_div_set(p_devinfo->apb2_div + 3);

amhw_zmf2x6_rcc_apb1_div_set(p_devinfo->apb1_div + 3);

amhw_zmf2x6_rcc_plldm_set(p_devinfo->pll_div - 1);
amhw_zmf2x6_rcc_plldn_set(p_devinfo->pll_mul - 1);

amhw_zmf2x6_rcc_pll_clk_set(p_devinfo->pll_src);

amhw_zmf2x6_rcc_pll_enable();

while (amhw_zmf2x6_rcc_pllrdy_read() == AM_FALSE); /* 等待 pll 就绪 */

amhw_zmf2x6_rcc_sys_clk_set(AMHW_ZMF2X6_SYSCCLK_PLL);

amhw_zmf2x6_rcc_pllforusb_set(p_devinfo->usb_div);
amhw_zmf2x6_rcc_timadv_div_set(p_devinfo->tim_div);

```

```

ZMF2X6_RCC->apb2enr &= ~0xc;

return AM_OK;
}

```

2. am_clk_rate_get 函数

该函数用于获取时钟频率，包括外设时钟频率和总线时钟频率，函数参数为时钟 ID。程序实现如程序清单 6.78 所示。

程序清单 6.78 获取时钟频率

```

/**
 * \brief 获取时钟频率
 */
int am_clk_rate_get (am_clk_id_t clk_id)
{
    int clk = 0;
    int peri, num;

    num = (clk_id & 0xff00) >> 8;
    peri = clk_id & 0xff;

    switch (num) {

    case 1:
        clk = __apb1_peri_clk(peri);
        break;

    case 2:
        clk = __apb2_peri_clk(peri);
        break;

    case 3:
        clk = __gp_dev->ahb_clk;
        break;

    case 4:
        clk = __other_clk(clk_id);
        break;

    default :
        break;
    }

    return clk;
}

```

3. am_clk_enable 函数

该函数用于使能外设时钟，函数参数为时钟 ID。程序实现如程序清单 6.79 所示。

程序清单 6.79 使能指定的系统或外设时钟

```
/**
 * \brief 使能指定的系统或外设时钟
 */
int am_clk_enable (am_clk_id_t clk_id)
{
    int peri, num;
    num = (clk_id & 0xff00) >> 8;
    peri = clk_id & 0xff;
    switch (num) {
    case 1:
        amhw_zmf2x6_rcc_apb1_enable ((amhw_zmf2x6_apb1_peripheral)peri);
        break;
    case 2:
        amhw_zmf2x6_rcc_apb2_enable ((amhw_zmf2x6_apb2_peripheral)peri);
        break;
    case 3:
        amhw_zmf2x6_rcc_ahb_enable ((amhw_zmf2x6_ahb_peripheral)peri);
        break;
    default :
        break;
    }
    return AM_OK;
}
```

4. am_clk_disable 函数

该函数用于禁能外设时钟，函数参数为时钟 ID。程序实现如程序清单 6.80 所示。

程序清单 6.80 禁能指定的系统或外设时钟

```
/**
 * \brief 禁能指定的系统或外设时钟
 */
int am_clk_disable (am_clk_id_t clk_id)
{
    int peri, num;
    num = (clk_id & 0xff00) >> 8;
    peri = clk_id & 0xff;

    switch (num) {
    case 1:
        amhw_zmf2x6_rcc_apb1_disable ((amhw_zmf2x6_apb1_peripheral)peri);
        break;
    case 2:
```

```

    amhw_zmf2x6_rcc_apb2_disable ((amhw_zmf2x6_apb2_peripheral)peri);
    break;
case 3:
    amhw_zmf2x6_rcc_ahb_disable ((amhw_zmf2x6_ahb_peripheral)peri);
    break;
default :
    break;
}
return AM_OK;
}

```

仔细观察会发现标准接口层调用了一些本地函数，比如 `am_clk_rate_get` 函数中调用了 `__apb1_peri_clk`、`__apb2_peri_clk` 等函数，这些函数功能更为详细具体，也让标准接口的程序更为明确和直观。关于本地函数的实现，请直接查看源文件。

到现在 CLK 支持的所有函数开发完毕，整个 CLK 的驱动层开发也就结束了。

6.4 用户配置文件编写

参考 5.4.1 章节新建用户配置文件，搭建文件的基本框架，准备配置文件内容的编写。

6.4.1 定义设备常量于设备信息常量

设备信息常量初始化，是对时钟参数的配置，详细的配置如程序清单 6.81 所示。

程序清单 6.81 时钟设备信息初始化配置

```

/**
 * \brief CLK 设备信息
 */
static const am_zmf2x6_clk_devinfo_t __g_clk_devinfo =
{
    /**
     * \brief HSEOSC 外部晶振频率
     *
     * 如果 pll_src 选择 AMHW_ZMF2X6_PLLCLK_HSE 则 PLLIN = hse_osc
     * PLLIN = 12000000
     */
    12000000,
    /** \brief
     * PLL 时钟源选择
     * -# AMHW_ZMF2X6_PLLCLK_HSI_DIV4 : HSI 振荡器 4 分频作为 PLL 输入时钟
     * -# AMHW_ZMF2X6_PLLCLK_HSE : HSE 作为 PLL 输入时钟
     */
    AMHW_ZMF2X6_PLLCLK_HSI_DIV4,
    /**
     * \brief PLL 倍频系数，可选 1-64
     *
     * PLLOUT = PLLIN * pll_mul / pll_div
     * PLLOUT = 12000000 * 4 / 1 = 48Mhz
     */

```



```

*/
4,
/**
 * \brief PLL 分频系数，可选 1-8
 *
 *      PLLOUT = PLLIN * pll_mul / pll_div
 *      PLLOUT  = 12000000 * 4 / 1 = 48Mhz
 */
1,
/** \brief USB 分频系数，USBCLK = PLLOUT / (usb_div + 1),建议配置成 48Mhz */
0,
/**
 * \brief TIM_ADV 分频系数，USBCLK = PLLOUT / (tim_div + 1),tim_div(0~7)
 *      0-3 :   不分频   TIM_ADV = PLLOUT / 2^(tim_div-3)
 */
0,
/**
 * \brief AHB 分频系数，AHBCLK = PLLOUT / DIV,AHB 最大频率为 48Mhz
 *
 *      ahb_div |  DIV
 *      -----
 *      0-7 |   1
 *      8   |   2
 *      9   |   4
 *      10  |   8
 *      11  |  16
 *      12  |  64
 *      13  | 128
 *      14  | 256
 *      15  | 512
 */
0,
/**
 * \brief APB1 分频系数，APB1CLK = AHBCLK / (2 ^ apb1_div)
 *      APB1 最大频率为 24Mhz
 */
1,
/**
 * \brief APB2 分频系数，APB2CLK = AHBCLK / (2 ^ apb2_div)
 *      APB2 最大频率为 48Mhz
 */
0,
/* 平台初始化函数，配置时钟引脚等操作 */
__zmf2x6_clk_plfm_init,
/* CLK 无平台去初始化函数 */

```

```

    NULL
};
/** \brief 时钟设备实例 */
static am_zmf2x6_clk_dev_t __g_clk_dev;

```

6.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

由于使用内部时钟，与外部引脚无关。因此，平台初始化函数如程序清单 6.82 所示。

程序清单 6.82 时钟平台初始化

```

/** \brief CLK 平台初始化 */
static void __zmf2x6_clk_plfm_init(void)
{
}

```

6.4.3 在头文件添加变量和函数声明

编写好平台初始化函数后，还需要在头文件中加入函数的声明，如程序清单 6.83 所示。

程序清单 6.83 平台初始化函数声明

```

/**
 * \brief zmf2x6 CLK 实例初始化，初始化系统时钟
 *
 * \param 无
 *
 * \return 返回 AW_OK 为初始化成功，其它为初始化失败
 */
int am_zmf2x6_clk_inst_init(void);

```

时钟初始化时，直接调用平台初始化函数 `am_zmf2x6_clk_inst_init` 即可完成时钟初始化。

至此，用户配置文件开发完毕。

6.5 外设例程文件开发

CLK 外设比较特殊，不需要开发额外的例程，在时钟初始化时直接调用平台初始化函数便可以完成时钟的配置。

第7章 通用外设驱动开发——GPIO

7.1 标准层了解

GPIO 标准层包含 2 个文件，am_gpio.h 和 am_gpio_util.h 文件，可在 ametal\interface 找到文件打开。

7.1.1 了解相关类型定义

打开 am_gpio.h 文件，我们可以看到，宏定义了 GPIO 引脚功能、模式、输出电平以及中断触发方式等，详见程序清单 7.1。

程序清单 7.1 GPIO 标准接口文件结构体定义

```

/** \brief GPIO 引脚输入 */
#define AM_GPIO_INPUT                AM_GPIO_COM_FUNC_CODE(AM_GPIO_INPUT_VAL)

/** \brief GPIO 引脚输出高电平 */
#define AM_GPIO_OUTPUT_INIT_HIGH    AM_GPIO_COM_FUNC_CODE(AM_GPIO_OUTPUT_INIT_HIGH_VAL)

/** \brief GPIO 引脚输出低电平 */
#define AM_GPIO_OUTPUT_INIT_LOW     AM_GPIO_COM_FUNC_CODE(AM_GPIO_OUTPUT_INIT_LOW_VAL)

/** \brief 上拉模式 */
#define AM_GPIO_PULLUP              AM_GPIO_COM_MODE_CODE(AM_GPIO_PULL_UP_VAL)

/** \brief 下拉模式 */
#define AM_GPIO_PULLDOWN            AM_GPIO_COM_MODE_CODE(AM_GPIO_PULL_DOWN_VAL)

/** \brief 浮动模式 */
#define AM_GPIO_FLOAT               AM_GPIO_COM_MODE_CODE(AM_GPIO_FLOAT_VAL)

/** \brief 开漏模式 */
#define AM_GPIO_OPEN_DRAIN          AM_GPIO_COM_MODE_CODE(AM_GPIO_OPEN_DRAIN_VAL)

/** \brief 推挽模式 */
#define AM_GPIO_PUSH_PULL           AM_GPIO_COM_MODE_CODE(AM_GPIO_PUSH_PULL_VAL)

/** \brief 关闭触发 */
#define AM_GPIO_TRIGGER_OFF         0

```

```

/** \brief 高电平触发 */
#define AM_GPIO_TRIGGER_HIGH      1

/** \brief 低电平触发 */
#define AM_GPIO_TRIGGER_LOW      2

/** \brief 上升沿触发 */
#define AM_GPIO_TRIGGER_RISE     3

/** \brief 下降沿触发 */
#define AM_GPIO_TRIGGER_FALL     4

/** \brief 双边沿触发 */
#define AM_GPIO_TRIGGER_BOTH_EDGES 5

/** \brief 低电平 */
#define AM_GPIO_LEVEL_LOW        0

/** \brief 高电平 */
#define AM_GPIO_LEVEL_HIGH       1

t;

```

7.1.2 了解标准接口函数

在 `am_gpio.h` 文件相关的类型定义后，就是具体了解标准接口函数，如程序清单 7.2 所示。

程序清单 7.2 GPIO 标准接口函数

```

/**
 * \brief 配置 GPIO 引脚功能
 *
 * 通过参数 \a flags 配置引脚功能 \a flags 的格式为:
 * (功能 | 模式 | 平台),
 * - 功能:
 *   - \ref grp_am_gpio_pin_function
 * - 模式:
 *   - \ref grp_am_gpio_pin_mode
 * - 平台:
 *   - xxx_pin.h
 *
 * 如果配置引脚的功能和模式与平台的配置冲突，功能域和模式域的优先级最高，而
 * 忽略平台的相关配置。
 */

```

```

* \param[in] pin      : 引脚编号, 值为 PIO* (#PIO0_0)
* \param[in] flags    : 配置参数, 参见
*
*                      \ref grp_am_gpio_pin_function 和
*                      \ref grp_am_gpio_pin_mode 和 xxx_pin.h
*
* \retval AM_OK       : 配置成功
* \retval -AM_ENOTSUP : 不支持该功能
*/
int am_gpio_pin_cfg(int pin, uint32_t flags);

/**
* \brief 获取 GPIO 引脚状态
* \param[in] pin : 引脚编号, 值为 PIO* (#PIO0_0)
* \return        引脚状态
*/
int am_gpio_get(int pin);

/**
* \brief 设置引脚输出状态
*
* \param[in] pin : 引脚编号, 值为 PIO* (#PIO0_0)
* \param[in] value : 引脚状态, 参见
*
*                      \ref grp_am_gpio_pin_level
*
* \retval AM_OK : 操作成功
*/
int am_gpio_set(int pin, int value);

/**
* \brief 翻转 GPIO 引脚输出状态
* \param[in] pin : 引脚编号, 值为 PIO* (#PIO0_0)
* \retval AM_OK : 操作成功
*/
int am_gpio_toggle(int pin);

/**
* \brief 配置 GPIO 引脚触发功能
*
* \param[in] pin : 引脚编号, 值为 PIO* (#PIO0_0)
* \param[in] flag : 配置参数, 参见
*
*                      \ref grp_am_gpio_pin_trigger_function
* \note: 1. 使用该接口, 应先调用 am_gpio_pin_cfg 设置 gpio 的引脚为输入并且选择合适的引脚模式
*        2. 使用该接口, 应首先调用 am_gpio_trigger_connect 连接引脚的中断回调函数
*
*/

```

```

* \retval AM_OK    : 配置成功
*/
int am_gpio_trigger_cfg(int pin, uint32_t flag);

/**
* \brief 连接回调函数到引脚
*
* \param[in] pin      : 引脚编号, 值为 PIO* (#PIO0_0)
* \param[in] pfn_callback : 回调函数指针
* \param[in] p_arg     : 回调函数的入口参数
*
* \retval AM_OK      : 操作成功
*/
int am_gpio_trigger_connect(int      pin,
                           am_pfnvoid_t pfn_callback,
                           void      *p_arg);

/**
* \brief 删除连接到引脚的回调函数
*
* \param[in] pin      : 引脚编号, 值为 PIO* (#PIO0_0)
* \param[in] pfn_callback : 回调函数指针
* \param[in] p_arg     : 回调函数的入口参数
*
* \retval AM_OK      : 操作成功
*/
int am_gpio_trigger_disconnect(int      pin,
                              am_pfnvoid_t pfn_callback,
                              void      *p_arg);

/**
* \brief 使能引脚触发中断
* \param[in] pin : 引脚编号, 值为 PIO* (#PIO0_0)
* \retval AM_OK : 操作成功
*/
int am_gpio_trigger_on(int pin);

/**
* \brief 禁能引脚触发中断
* \param[in] pin : 引脚编号, 值为 PIO* (#PIO0_0)
* \retval AM_OK : 操作成功
*/
int am_gpio_trigger_off(int pin);

```

1. am_gpio_pin_cfg()配置 GPIO 引脚功能函数

该函数用于配置 GPIO 引脚功能，传入的参数为引脚编号以及功能模式相关宏标志，并返回函数执行的状态。

2. am_gpio_get()获取 GPIO 引脚状态函数

该函数用于获取 GPIO 引脚状态，传入的参数为引脚编号，并返回函数执行的状态。

3. am_gpio_set()设置引脚输出状态函数

该函数用于设置引脚输出状态，传入的参数为引脚编号以及引脚状态，并返回函数执行的状态。

4. am_gpio_toggle()翻转 GPIO 引脚输出状态函数

该函数用于翻转 GPIO 引脚输出状态，传入的参数为引脚编号，并返回函数执行的状态。

5. am_gpio_trigger_cfg()配置 GPIO 引脚触发功能函数

该函数用于配置 GPIO 引脚触发功能，传入的参数为引脚编号以及触发配置标志，并返回函数执行的状态。

6. am_gpio_trigger_connect()连接到引脚的回调函数

该函数用于设置连接到引脚的回调函数，传入的参数为引脚编号、回调函数地址和回调函数入口参数，并返回函数执行的状态。

7. am_gpio_trigger_disconnect()删除连接到引脚的回调函数

该函数用于删除连接到引脚的回调函数，传入的参数为引脚编号、回调函数地址和回调函数入口参数，并返回函数执行的状态。

8. am_gpio_trigger_on()使能引脚触发中断函数

该函数用于使能引脚触发中断，传入的参数为引脚编号，并返回函数执行的状态。

9. am_gpio_trigger_off()禁能引脚触发中断函数

该函数用于禁能引脚触发中断，传入的参数为引脚编号，并返回函数执行的状态。

到此我们就分析完了 GPIO 外设标准层数据类型和接口函数，清楚了驱动层需要为标准接口层提供哪些内容。

7.2 HW 层开发

参考 5.2.1 章节，创建对应目录和文件，添加硬件层基本的内容。

7.2.1 添加寄存器列表

HW 层就是与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读该外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMF2x6 用户手册的 GPIO 章节，简单描述如下：

表 7.1 GPIO 寄存器概览

位	标识符	含义
1	GPIOx_CRL	端口配置低寄存器
2	GPIOx_CRH	端口配置高寄存器
3	GPIOx_IDR	端口输入数据寄存器
4	GPIOx_ODR	端口输出数据寄存器

5	GPIOx_BSRR	端口设置/清除寄存器
6	GPIOx_BRR	端口位清除寄存器
7	GPIOx_LCKR	端口配置锁定寄存器
8	GPIOx_DCR	端口输出开漏控制寄存器
9	GPIOx_AFR_L	端口复用功能低位寄存器
10	GPIOx_AFR_H	端口复用功能高位寄存器

到此 GPIO 的各个寄存器及其功能基本阅读完毕，我们也大致了解了各个寄存器功能以及要配置各个功能，其他寄存器位该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，如程序清单 7.3 所示。

程序清单 7.3 寄存器结构体定义

```
/**
 * \brief GPIO 寄存器块结构体
 */
typedef struct amhw_zmf2x6_gpio {
    __IO uint32_t cr[2];      /**< \brief 端口配置(高低)寄存器 */
    __IO uint32_t idr;        /**< \brief 端口输入数据寄存器 */
    __IO uint32_t odr;        /**< \brief 端口输出寄存器 */
    __IO uint32_t bsrr;       /**< \brief 端口设置/清除寄存器 */
    __IO uint32_t brr;        /**< \brief 端口位清除寄存器 */
    __IO uint32_t lckr;       /**< \brief 端口配置锁定寄存器 */
    __IO uint32_t dcr;        /**< \brief 端口输出开漏寄存器 */
    __IO uint32_t afr[2];     /**< \brief 端口复用功能(高低)寄存器 */
} amhw_zmf2x6_gpio_t;
```

这个小节我们定义了 GPIO 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

所有的内联函数都应该在定义时标识为内联函数，在函数的上方加入 `am_static_inline` 的关键字，具体见下文的示例。

7.2.2 编写寄存器相关函数

1. GPIOx_BSRR（端口设置/清除寄存器）

端口设置/清除寄存器包括设置对应的 ODRy 位为 1 的函数，如程序清单 7.4 所示。

程序清单 7.4 GPIOx_BSRR 操作函数及参数宏

```
/**
 * \brief 设置 GPIO 引脚输出高电平
 *
 * \param[in] p_hw_gpio : 指向 GPIO 寄存器块的指针
 * \param[in] pin       : 引脚编号，值为 PIO* (#PIOA_0)
 *
 * \return 无.
 */
```



```

* \note 该功能配置输入引脚的时候为使能上拉电阻并失能下拉电阻
*/
am_static_inline
void amhw_zmf2x6_gpio_pin_out_high (amhw_zmf2x6_gpio_t *p_hw_gpio, int pin)
{
    amhw_zmf2x6_gpio_t *p_gpio_addr = (amhw_zmf2x6_gpio_t *)((int)p_hw_gpio \
        + (pin >> 4) * 0x400);

    /* 相应引脚输出高电平 */
    p_gpio_addr->bsrr|= (1UL << (pin & 0x0f));
}

```

2. GPIOx_BRR（端口位清除寄存器）

端口位清除寄存器包括清除对应的 ODRy 位为 0 的函数，操作函数如程序清单 7.5 所示。

程序清单 7.5 GPIOx_BRR 操作函数

```

/**
* \brief 设置 GPIO 引脚输出低电平
*
* \param[in] p_hw_gpio : 指向 GPIO 寄存器块的指针
* \param[in] pin       : 引脚编号，值为 PIO* (#PIOA_0)
*
* \return 无
*
* \note 该功能配置输入引脚的时候为失能上拉电阻并使能下拉电阻
*/
am_static_inline
void amhw_zmf2x6_gpio_pin_out_low (amhw_zmf2x6_gpio_t *p_hw_gpio, int pin)
{
    amhw_zmf2x6_gpio_t *p_gpio_addr = (amhw_zmf2x6_gpio_t *)((int)p_hw_gpio \
        + (pin >> 4) * 0x400);

    p_gpio_addr->brr|= (1UL << (pin & 0x0f));
}

```

3. GPIOx_LCKR（端口配置锁定寄存器）

端口配置锁定寄存器用于设置 GPIO 引脚锁定以及端口锁键设置，操作函数如程序清单 7.6 所示。

程序清单 7.6 GPIOx_LCKR 操作函数

```

/**
* \brief 设置 GPIO 引脚锁定
*
* \param[in] p_hw_gpio : 指向 GPIO 寄存器块的指针
* \param[in] pin       : 引脚编号，值为 PIO* (#PIOA_0)
*

```

```

* \note 该函数调用多次锁定同一 GPIO 端口不同引脚的配置
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_gpio_pinlock_set (amhw_zmf2x6_gpio_t *p_hw_gpio, int pin)
{
    amhw_zmf2x6_gpio_t *p_gpio_addr = (amhw_zmf2x6_gpio_t *)((int)p_hw_gpio \
        + (pin >> 4) * 0x400);
    p_gpio_addr->lckr |= (1UL << (pin & 0x0f));
}

/**
* \brief GPIO 端口锁键设置
*
* \param[in] p_hw_gpio : 指向 GPIO 寄存器块的指针
* \param[in] port_num : 端口编号, 值为 PIO* (# AMHW_GPIO_PORT_A)
*
* \note 该函数一般 在 amhw_zmf2x6_gpio_pinlock_set 调用后紧接着调用
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_gpio_lckk_set (amhw_zmf2x6_gpio_t *p_hw_gpio, int port_num)
{
    uint32_t temp = (1UL << 16);

    amhw_zmf2x6_gpio_t *p_gpio_addr = (amhw_zmf2x6_gpio_t *)((int)p_hw_gpio \
        + (port_num >> 4) * 0x400);

    /* Set LCKK bit */
    p_gpio_addr->lckr |= temp;
    /* Reset LCKK bit */
    p_gpio_addr->lckr &= ~temp;
    /* Set LCKK bit */
    p_gpio_addr->lckr |= temp;
    /* Read LCKK bit */
    temp = p_gpio_addr->lckr;
    /* Read LCKK bit */
    temp = p_gpio_addr->lckr;
}

```

还有诸多寄存器, 可按上述样例逐一添加, GPIO 外设寄存器众多, 本节不再一一罗列。

7.2.3 添加其他功能性函数

在每个寄存器的操作函数编写完毕后，可能 HW 层的接口还有所欠缺，需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到，可以暂时不用考虑，直接进入驱动层的开发。在后续如果发现 HW 层欠缺哪些接口，可以反过来再增加。当然，也有可能 HW 层接口没有什么欠缺了，那样就无需再开发其它函数了。

7.2.4 添加指向寄存器块的指针

为了方便用户或驱动层使用，将 GPIO 外设基地址定义为 GPIO 外设结构体指针，后续使用硬件接口层时直接使用宏即可。在 AMetal 中，由于这些内容与具体型号的芯片相关，这样的宏统一定义到 zmf2x6_periph_map.h 文件中，如程序清单 7.7 所示。

程序清单 7.7 外设中加入 GPIO 结构体指针定义

```
/** \brief GPIO 寄存器块指针 */
#define ZMF2X6_GPIO ((amhw_zmf2x6_gpio_t *)ZMF2X6_GPIO_BASE)
```

后续用户或驱动层使用 HW 层接口时，可直接使用 ZMF2X6_GPIO 作为参数。至此，整个 HW 层的开发工作就结束了，如果后续开发中发现缺少某些函数，还可以再添加。

7.3 驱动层开发

参考 5.3.1 章节，新建 am_zmf2x6_gpio.h、am_zmf2x6_gpio_util.h 和 am_zmf2x6_gpio.c 驱动文件，搭建相应的文件框架，并按规则放到对应的路径下。

7.3.1 编写结构性程序

驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

定义 GPIO 外设设备信息结构体如程序清单 7.8 所示。后续可以根据实际情况修改。

程序清单 7.8 GPIO 设备信息结构体定义

```
/**
 * \brief 引脚的触发信息
 */
struct am_zmf2x6_gpio_trigger_info {

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

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

```

};

/**
 * \brief GPIO 设备信息
 */
typedef struct am_zmf2x6_gpio_devinfo {

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

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

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

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

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

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

    /** \brief 指向引脚触发信息的指针 */
    struct am_zmf2x6_gpio_trigger_info *p_triginfo;

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

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

} am_zmf2x6_gpio_devinfo_t;

```

● 设备结构体定义

由于 GPIO 设备需要提供标准的 GPIO 服务，包括引脚配置、引脚中断触发配置等。其次，为了在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 7.9 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 7.9 GPIO 设备结构体定义

```

/**
 * \brief GPIO 设备实例
 */

```

```
typedef struct am_zmf2x6_gpio_dev {

    /** \brief 指向 GPIO 设备信息的指针 */
    const am_zmf2x6_gpio_devinfo_t *p_devinfo;

    /** \brief 参数有效标志 */
    am_bool_t                valid_flg;

} am_zmf2x6_gpio_dev_t;
```

2. 搭建函数框架

函数主要分为两部分一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于 GPIO 驱动层需要提供初始化函数之外，还需要实现标准接口层函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为 GPIO 初始化的状态。示例代码如下程序清单 7.10 所示。

程序清单 7.10 驱动层函数定义示例

```
/**
 * \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_zmf2x6_gpio_init (am_zmf2x6_gpio_dev_t      *p_dev,
                        const am_zmf2x6_gpio_devinfo_t *p_devinfo);

/**
 * \brief GPIO 去初始化
 *
 * \param[in] 无
 *
 * \return 无
 */
void am_zmf2x6_gpio_deinit (void);
```

● 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 9 个函数，根据标准接口层函数声明，我们需要在.c 文件中定义 9 个函数。具体实现将在下一小节中描述。

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。

7.3.2 驱动函数实现

通过上面的函数框架搭建，共计 11 个函数，9 个功能性函数面向标准层，2 个初始解初始化函数面向用户，下面逐步实现。

1. am_zmf2x6_gpio_init 函数

该函数主要完成两个功能：

- a) 初始化设备中的标准服务，将成员赋值。
- b) 初始化硬件相关设置，该函数的示例代码如程序清单 7.11 所示。

程序清单 7.11 am_zmf2x6_gpio_init 函数示例代码

```
/**
 * \brief GPIO 初始化
 */
int am_zmf2x6_gpio_init (am_zmf2x6_gpio_dev_t *p_dev,
                        const am_zmf2x6_gpio_devinfo_t *p_devinfo)
{
    int i;

    if (p_dev == NULL || p_devinfo == NULL) {
        return -AM_EINVAL;
    }

    /* GPIO 平台初始化 */
    if (p_devinfo->pfn_plfm_init) {
        p_devinfo->pfn_plfm_init();
    }

    /* GPIO 设备结构体初始化 */
    p_dev->p_devinfo = p_devinfo;
    __gp_gpio_dev = p_dev;
    p_dev->valid_flg = AM_TRUE;

    if ((p_devinfo->p_trinfo == NULL) ||
        (p_devinfo->p_infomap == NULL)) {
        p_dev->valid_flg = AM_FALSE;
    }

    /* 初始化每个引脚的回调处理函数及触发方式内存映射 */
    if (p_dev->valid_flg) {
        for (i = 0; i < p_devinfo->pint_count; i++) {
            (p_devinfo->p_infomap)[i] = AM_ZMF2X6_GPIO_INVALID_PIN_MAP;
            (p_devinfo->p_trinfo)[i].pfn_callback = NULL;
        }
    }
}
```

```

/* 连接 GPIO 引脚的中断处理函数 */
am_int_connect((p_devinfo->inum_pin)[0], /* 写入中断号 */
              (am_pfnvoid_t)__port_exit0_1_int_isr,
              (void *)p_devinfo);
am_int_connect((p_devinfo->inum_pin)[1], /* 写入中断号 */
              (am_pfnvoid_t)__port_exit2_3_int_isr,
              (void *)p_devinfo);
am_int_connect((p_devinfo->inum_pin)[2], /* 写入中断号 */
              (am_pfnvoid_t)__port_exit4_15_int_isr,
              (void *)p_devinfo);

/* 使能 GPIO 的 NVIC 中断 */
am_int_enable(p_devinfo->inum_pin[0]);
am_int_enable(p_devinfo->inum_pin[1]);
am_int_enable(p_devinfo->inum_pin[2]);

return AM_OK;
}

```

2. am_zmf2x6_gpio_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放。

因此，该函数仅简单的将几个不再使用的变量置为 NULL 即可。示例代码如程序清单 7.12 所示。

程序清单 7.12 am_zmf2x6_gpio_deinit 函数示例代码

```

/**
 * \brief GPIO 去初始化
 */
void am_zmf2x6_gpio_deinit (void)
{
    __GPIO_DEVINFO_DECL(p_gpio_devinfo, __gp_gpio_dev);

    int i = 0;

    if (__gp_gpio_dev->valid_flg) {
        for (i = 0; i < p_gpio_devinfo->pint_count; i++) {
            (p_gpio_devinfo->p_infomap)[i] = AM_ZMF2X6_GPIO_INVALID_PIN_MAP;
            (p_gpio_devinfo->p_triginfo)[i].pfn_callback = NULL;
        }
    }

    am_int_disable((p_gpio_devinfo->inum_pin)[0]);
    am_int_disable((p_gpio_devinfo->inum_pin)[1]);
}

```

```

am_int_disable((p_gpio_devinfo->inum_pin)[2]);

am_int_disconnect((p_gpio_devinfo->inum_pin)[0], /* 写入中断号 */
                  (am_pfnvoid_t)__port_exit0_1_int_isr,
                  (void *)p_gpio_devinfo);
am_int_disconnect((p_gpio_devinfo->inum_pin)[1], /* 写入中断号 */
                  (am_pfnvoid_t)__port_exit2_3_int_isr,
                  (void *)p_gpio_devinfo);
am_int_disconnect((p_gpio_devinfo->inum_pin)[2], /* 写入中断号 */
                  (am_pfnvoid_t)__port_exit4_15_int_isr,
                  (void *)p_gpio_devinfo);

if (__gp_gpio_dev->p_devinfo->pfn_plfm_deinit) {
    __gp_gpio_dev->p_devinfo->pfn_plfm_deinit();
}
}

```

3. am_gpio_pin_cfg 函数

该函数主要用于引脚功能配置，功能实现代码详见程序清单 7.13。

程序清单 7.13 am_gpio_pin_cfg 函数示例代码

```

/**
 * \brief 引脚功能配置
 *
 * \param[in] pin      : 引脚编号，值为 PIO* (#PIOA_0)
 * \param[in] flags    : 引脚功能
 *
 * \return AM_OK       : 配置成功
 */
int am_gpio_pin_cfg (int pin, uint32_t flags)
{
    __GPIO_DEVINFO_DECL(p_gpio_devinfo, __gp_gpio_dev);

    /* IO 口方向 */
    uint8_t dir = 0;

    uint32_t func = 0, mode = 0;

    uint32_t iocon_cfg = 0;

    amhw_zmf2x6_gpio_t *p_hw_gpio = NULL;

    if (__gp_gpio_dev == NULL) {
        return -AM_ENXIO;
    }
}

```



```

p_hw_gpio = (amhw_zmf2x6_gpio_t*)p_gpio_devinfo->gpio_regbase;

func      = AM_GPIO_COM_FUNC_GET(flags);
mode      = AM_GPIO_COM_MODE_GET(flags) << 3;

/*
 * 共有功能配置
 */
if (func != 0x00) {

    /* 表明不需要再设置方向 */
    flags &= ~AM_ZMF2X6_GPIO_DIR;

    /* 表明不需要再设置引脚复用功能，仅使用 GPIO 功能 */
    flags &= ~AM_ZMF2X6_GPIO_FUNEN;

    if (amhw_zmf2x6_gpio_pin_afr_get(p_hw_gpio, pin) != AMHW_ZMF2X6_GPIO_AF_DEFAULT) {

        /* 复用功能回到默认状态，表明使用 GPIO 功能 */
        amhw_zmf2x6_gpio_pin_afr_set(p_hw_gpio, AMHW_ZMF2X6_GPIO_AF_DEFAULT, pin);

    }

    switch (func) {

    case AM_GPIO_INPUT:
        AM_BITS_SET(iocon_cfg, 0, 2, 0X00);
        break;

    case AM_GPIO_OUTPUT_INIT_HIGH:

        amhw_zmf2x6_gpio_pin_out_high(p_hw_gpio, pin);

        /* 输出时，若未配置管脚速率则默认为 10MHz */
        if ((flags & AM_ZMF2X6_GPIO_OUTRES_RATE) == 0) {
            AM_BITS_SET(iocon_cfg, 0, 2, AMHW_ZMF2X6_GPIO_SPEED_10MHz);
        }

        dir = 1;

        break;

    case AM_GPIO_OUTPUT_INIT_LOW:

```

```

    amhw_zmf2x6_gpio_pin_out_low(p_hw_gpio, pin);

    /* 输出时，若未配置管脚速率则默认为 10MHz */
    if ((flags & AM_ZMF2X6_GPIO_OUTRES_RATE) == 0) {
        AM_BITS_SET(iocon_cfg, 0, 2, AMHW_ZMF2X6_GPIO_SPEED_10MHz);
    }

    dir = 1;

    break;

default:
    break;
}
}

if (mode != 0x00) {

    switch (mode) {

case AM_GPIO_PULLUP:
    if (dir != 1) {

        /* 使 OD 位置为 1, 表明是上拉 */
        amhw_zmf2x6_gpio_pin_out_high(p_hw_gpio, pin);

        AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_IPU);
    } else {
        AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_OUT_PP);
    }
    break;

case AM_GPIO_PULLDOWN:
    if (dir != 1) {

        /* 使 OD 位置为 0, 表明是下拉 */
        amhw_zmf2x6_gpio_pin_out_low(p_hw_gpio, pin);
        AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_IPD);
    } else {
        AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_OUT_PP);
    }
    break;

case AM_GPIO_FLOAT:

```

```

        if (dir != 1) {
            AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_IN_FLOATING);
        } else {
            AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_OUT_OD);
        }
        break;

case AM_GPIO_OPEN_DRAIN:
    if (dir == 0) {
        AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_IN_FLOATING);
    } else {
        AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_OUT_OD);
    }
    break;

case AM_GPIO_PUSH_PULL:
    if (dir == 0) {
        AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_IPU);
    } else {
        AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_OUT_PP);
    }
    break;

default:
    return -AM_ENOTSUP;
}
}

/*
 * 平台相关配置
 */

/* 管脚的模式 */
if ((flags & AM_ZMF2X6_GPIO_MODE) != 0) {

    switch (AM_ZMF2X6_GPIO_MODE_GET(flags)) {

case 0:
    AM_BITS_SET(iocon_cfg, 0, 2, 0x00);
    AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_AIN);

    break;

case 1:

```

```

    AM_BITS_SET(iocon_cfg, 0, 2, 0x00);
    AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_IN_FLOATING);

    break;

case 2:

    /* 使 OD 位置为 0, 表明是下拉 */
    amhw_zmf2x6_gpio_pin_out_low(p_hw_gpio, pin);
    AM_BITS_SET(iocon_cfg, 0, 2, 0x00);
    AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_IPD);

    break;

case 3:

    /* 使 OD 位置为 1, 表明是上拉 */
    amhw_zmf2x6_gpio_pin_out_high(p_hw_gpio, pin);
    AM_BITS_SET(iocon_cfg, 0, 2, 0x00);
    AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_IPU);

    break;

case 4:

    AM_BITS_SET(iocon_cfg, 0, 2, AMHW_ZMF2X6_GPIO_SPEED_10MHz);
    AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_OUT_PP);

    break;

case 5:

    AM_BITS_SET(iocon_cfg, 0, 2, AMHW_ZMF2X6_GPIO_SPEED_10MHz);
    AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_OUT_OD);

    break;

case 6:

    AM_BITS_SET(iocon_cfg, 0, 2, AMHW_ZMF2X6_GPIO_SPEED_10MHz);
    AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_AF_PP);

    break;

case 7:

    AM_BITS_SET(iocon_cfg, 0, 2, AMHW_ZMF2X6_GPIO_SPEED_10MHz);
    AM_BITS_SET(iocon_cfg, 2, 2, AMHW_ZMF2X6_GPIO_MODE_AF_OD);

```

```

        break;

    default:
        return -AM_ENOTSUP;
    }
}

/* 设置管脚的输出速率 */
if ((flags & AM_ZMF2X6_GPIO_OUTRES_RATE) != 0) {

    AM_BITS_SET(iocon_cfg, 0, 2, AM_ZMF2X6_GPIO_OUTRES_RATE_GET(flags));

}

/* 设置复用功能时管脚模式 */
amhw_zmf2x6_gpio_pin_set(p_hw_gpio, iocon_cfg, pin);

/* 设置管脚的复用功能 */
if ((flags & AM_ZMF2X6_GPIO_FUNEN) != 0) {
    amhw_zmf2x6_gpio_pin_afr_set(p_hw_gpio, AM_ZMF2X6_GPIO_FUNC_GET(flags), pin);
}

return AM_OK;
}

```

4. am_gpio_get 函数

该函数主要用于获取引脚状态，功能实现代码如程序清单 7.14 所示。

程序清单 7.14 am_gpio_get 函数示例代码

```

/**
 * \brief 获取引脚状态
 *
 * \param[in] pin      : 引脚编号，值为 PIO* (#PIOA_0)
 *
 * \return 无
 */
int am_gpio_get (int pin)
{
    __GPIO_DEVINFO_DECL(p_gpio_devinfo, __gp_gpio_dev);

    amhw_zmf2x6_gpio_t *p_hw_gpio = (amhw_zmf2x6_gpio_t *)p_gpio_devinfo->gpio_regbase;

    return amhw_zmf2x6_gpio_pin_get(p_hw_gpio, pin);
}

```

5. am_gpio_set 函数

该函数主要用于设置输出引脚状态，功能实现代码如程序清单 7.15 所示。

程序清单 7.15 am_gpio_set 函数示例代码

```
/**
 * \brief 设置输出引脚状态
 *
 * \param[in] pin      : 引脚编号, 值为 PIO* (#PIOA_0)
 * \param[in] value    : 引脚值
 *
 * \return AM_OK      : 设置成功
 */
int am_gpio_set (int pin, int value)
{
    __GPIO_DEVINFO_DECL(p_gpio_devinfo, __gp_gpio_dev);

    amhw_zmf2x6_gpio_t *p_hw_gpio = (amhw_zmf2x6_gpio_t *)p_gpio_devinfo->gpio_regbase;

    if (value == 0) {
        amhw_zmf2x6_gpio_pin_out_low(p_hw_gpio, pin);
    } else {
        amhw_zmf2x6_gpio_pin_out_high(p_hw_gpio, pin);
    }

    return AM_OK;
}
```

6. am_gpio_toggle 函数

该函数主要用于引脚输出状态反转，功能实现代码如程序清单 7.16 所示。

程序清单 7.16 am_gpio_toggle 函数示例代码

```
/**
 * \brief 引脚输出状态反转
 */
int am_gpio_toggle (int pin)
{
    __GPIO_DEVINFO_DECL(p_gpio_devinfo, __gp_gpio_dev);

    amhw_zmf2x6_gpio_t *p_hw_gpio = (amhw_zmf2x6_gpio_t *)p_gpio_devinfo->gpio_regbase;

    amhw_zmf2x6_gpio_pin_toggle(p_hw_gpio, pin);

    return AM_OK;
}
```

7. am_gpio_trigger_cfg 函数

该函数主要用于引脚触发形式设置，功能实现代码如程序清单 7.17 所示。

程序清单 7.17 am_gpio_trigger_cfg 函数示例代码

```

/**
 * \brief 引脚触发形式设置
 */
int am_gpio_trigger_cfg (int pin, uint32_t flag)
{
    __GPIO_DEVINFO_DECL(p_gpio_devinfo, __gp_gpio_dev);

    int port = 0;
    int slot = -1;

    amhw_zmf2x6_gpio_t *p_hw_gpio = NULL;
    amhw_zmf2x6_syscfg_t *p_hw_syscfg = NULL;
    amhw_zmf2x6_exti_t *p_hw_exti = NULL;

    if (__gp_gpio_dev == NULL) {
        return -AM_ENXIO;
    }

    port = pin >> 4;      /* 获得引脚对应的端口号 */

    if (port > 3) {        /* 端口不存在 */
        return -AM_EINVAL;
    }

    p_hw_gpio = (amhw_zmf2x6_gpio_t *)p_gpio_devinfo->gpio_regbase;
    p_hw_syscfg = (amhw_zmf2x6_syscfg_t *)p_gpio_devinfo->syscfg_regbase;
    p_hw_exti = (amhw_zmf2x6_exti_t *)p_gpio_devinfo->exti_regbase;

    /* 还没有开始配置管脚 */
    if ((p_gpio_devinfo->p_infomap)[pin & 0x0f] == pin) {
        slot = pin & 0x0f;
    } else {
        return -AM_ENXIO;
    }

    /* 复用功能回到默认状态，表明使用 GPIO 功能 */
    amhw_zmf2x6_gpio_pin_afr_set(p_hw_gpio, AMHW_ZMF2X6_GPIO_AF_DEFAULT, pin);

    /* 可以添加该中断引脚配置为 GPIO 功能的函数代码 */

    /* 设置管脚为输入方向 */
    amhw_zmf2x6_gpio_pin_dir_input(p_hw_gpio, pin);

```

```

/* 管脚为上拉/下拉模式 */
amhw_zmf2x6_gpio_pin_mode_set(p_hw_gpio, AMHW_ZMF2X6_GPIO_MODE_IPD, pin);

/*
 * 中断线 0 只能连一个端口编号为 0 的 GPIO 引脚，比如说，PIOA_0 连接到了中断线 0，
 * PIOB_0、PIOC_0、PIOD_0 就不能连到中断线 0，但此时 PIOA_1 可以连接到中断线 1。
 */
amhw_zmf2x6_syscfg_exti_line_config(p_hw_syscfg, port, slot);

/* 清除中断线配置 */
amhw_zmf2x6_exti_imr_clear(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));
amhw_zmf2x6_exti_emr_clear(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));

/* 清除中断线触发方式配置 */
amhw_zmf2x6_exti_rtsr_clear(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));
amhw_zmf2x6_exti_ftsr_clear(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));

switch (flag) {

case AM_GPIO_TRIGGER_OFF:          /* 关闭触发 */
    amhw_zmf2x6_exti_imr_clear(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));
    amhw_zmf2x6_exti_emr_clear(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));
    break;

case AM_GPIO_TRIGGER_HIGH:         /* 高电平触发 */
    return -AM_ENOTSUP;

case AM_GPIO_TRIGGER_LOW:          /* 低电平触发 */
    return -AM_ENOTSUP;

case AM_GPIO_TRIGGER_RISE:         /* 上升沿触发 */

    /* 使 OD 位置为 0，表明是下拉 */
    amhw_zmf2x6_gpio_pin_out_low(p_hw_gpio, pin);

    /* rising edge */
    amhw_zmf2x6_exti_rtsr_set(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));

    break;

case AM_GPIO_TRIGGER_FALL:         /* 下降沿触发 */

    /* 使 OD 位置为 1，表明是上拉 */
    amhw_zmf2x6_gpio_pin_out_high(p_hw_gpio, pin);

```



```

    /* falling edge */
    amhw_zmf2x6_exti_ftsr_set(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));

    break;

case AM_GPIO_TRIGGER_BOTH_EDGES:    /* 双边沿触发 */

    /* 使 OD 位置为 1, 表明是上拉 */
    amhw_zmf2x6_gpio_pin_out_high(p_hw_gpio, pin);

    /* rising falling edge */
    amhw_zmf2x6_exti_rtsr_set(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));
    amhw_zmf2x6_exti_ftsr_set(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));

    break;

default:
    break;
}

return AM_OK;
}

```

8. am_gpio_trigger_on 函数

该函数主要用于使能引脚中断，功能实现代码如程序清单 7.18 所示。

程序清单 7.18 am_gpio_trigger_on 函数示例代码

```

/**
 * \brief 使能引脚中断。
 */
int am_gpio_trigger_on (int pin)
{
    __GPIO_DEVINFO_DECL(p_gpio_devinfo, __gp_gpio_dev);

    int slot = -1;
    int port = 0;

    int err = 0;

    uint32_t extipinsource = 0;

    amhw_zmf2x6_exti_t    *p_hw_exti    = NULL;

    if (__gp_gpio_dev == NULL) {

```

```

        return -AM_ENXIO;
    }

    port = pin >> 4;          /* 获得引脚对应的端口号 */

    if (port > 3) {           /* 端口不存在 */
        return -AM_EINVAL;
    }

    extipinsource = pin & 0x0f;

    /* 还没有开始配置管脚 */
    if ((p_gpio_devinfo->p_infomap)[extipinsource] == pin) {
        slot = extipinsource;
    } else {
        return -AM_ENXIO;
    }

    p_hw_exti = (amhw_zmf2x6_exti_t *)p_gpio_devinfo->exti_regbase;

    /* 清引脚中断线标志位 */
    amhw_zmf2x6_exti_pending_clear(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));

    /* 设置开放中断线请求 */
    amhw_zmf2x6_exti_imr_set(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));

    return err;
}

```

9. am_gpio_trigger_off 函数

该函数主要用于禁用引脚中断，功能实现代码如程序清单 7.19 所示。

程序清单 7.19 am_gpio_trigger_off 函数示例代码

```

/**
 * \brief 禁用引脚中断。
 */
int am_gpio_trigger_off (int pin)
{
    __GPIO_DEVINFO_DECL(p_gpio_devinfo, __gp_gpio_dev);

    int slot = -1;
    int port = 0;

    uint32_t extipinsource = 0;

```

```

amhw_zmf2x6_exti_t  *p_hw_exti  = NULL;

if (__gp_gpio_dev == NULL) {
    return -AM_ENXIO;
}

port = pin >> 4;      /* 获得引脚对应的端口号 */

if (port > 3) {        /* 端口不存在 */
    return -AM_EINVAL;
}

extipinsource = pin & 0x0f;

/* 还没有开始配置管脚 */
if ((p_gpio_devinfo->p_infomap)[extipinsource] == pin) {
    slot = extipinsource;
} else {
    return -AM_ENXIO;
}

p_hw_exti = (amhw_zmf2x6_exti_t *)p_gpio_devinfo->exti_regbase;

/* 屏蔽中断线请求 */
amhw_zmf2x6_exti_imr_clear(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));
amhw_zmf2x6_exti_emr_clear(p_hw_exti, (amhw_zmf2x6_exti_line_num_t)(1UL << slot));

return AM_OK;
}

```

10. am_gpio_trigger_connect 函数

该函数主要用于设置连接引脚中断回调函数，功能实现代码如程序清单 7.20 所示。

程序清单 7.20 am_gpio_trigger_connect 函数示例代码

```

/**
 * \brief 连接引脚中断回调函数
 */
int am_gpio_trigger_connect (int          pin,
                             am_pfnvoid_t pfn_callback,
                             void         *p_arg)
{
    __GPIO_DEVINFO_DECL(p_gpio_devinfo, __gp_gpio_dev);

    int    key;

```

```

int      slot      = -1;
int      port      = 0;

uint32_t extipinsource = 0;

am_bool_t already = AM_FALSE;

extipinsource = pin & 0x0f;

port = pin >> 4;

if (port > 3) {
    return -AM_EINVAL;
}

/* 关闭 CPU 中断 */
key = am_int_cpu_lock();

if ((p_gpio_devinfo->p_infomap)[extipinsource] == AM_ZMF2X6_GPIO_INVALID_PIN_MAP) {
    slot = extipinsource;
} else {
    already = AM_TRUE;
}

if (already) {

    /* 打开 CPU 中断 */
    am_int_cpu_unlock(key);
    return -AM_EINVAL;
}

if (-1 != slot) {

    /* 保存触发引脚及回调函数 */
    (p_gpio_devinfo->p_infomap)[slot]          = pin;
    (p_gpio_devinfo->p_triginfo)[slot].pfn_callback = pfn_callback;
    (p_gpio_devinfo->p_triginfo)[slot].p_arg      = p_arg;

    /* 打开中断 */
    am_int_cpu_unlock(key);

}

return AM_OK;

```

}

11. am_gpio_trigger_disconnect 函数

该函数主要用于删除连接引脚中断回调函数，功能实现代码如程序清单 7.21 所示。

程序清单 7.21 am_gpio_trigger_disconnect 函数示例代码

```

/**
 * \brief 删除引脚中断回调函数连接
 */
int am_gpio_trigger_disconnect (int          pin,
                                am_pfnvoid_t pfn_callback,
                                void          *p_arg)
{
    __GPIO_DEVINFO_DECL(p_gpio_devinfo, __gp_gpio_dev);

    int key;

    int port = 0;

    int slot = -1;

    uint32_t extipinsource = 0;

    port = pin >> 4;

    if (port > 3) {
        return -AM_EINVAL;
    }

    extipinsource = pin & 0x0f;

    /* 关闭 CPU 中断 */
    key = am_int_cpu_lock();

    if ((p_gpio_devinfo->p_infomap)[extipinsource] == pin) {
        slot = extipinsource;
    }

    if (slot == -1) {

        /* 打开 CPU 中断 */
        am_int_cpu_unlock(key);

        return -AM_EINVAL;
    }

```

```

(p_gpio_devinfo->p_infomap)[slot] = AM_ZMF2X6_GPIO_INVALID_PIN_MAP;
(p_gpio_devinfo->p_trinfo)[slot].pfn_callback = NULL;

/* 打开 CPU 中断 */
am_int_cpu_unlock(key);

return AM_OK;
}

```

7.4 用户配置文件编写

参考 5.4.1 章节新建用户配置文件，搭建文件的基本框架，准备配置文件内容的编写。

7.4.1 定义设备常量于设备信息常量

设备信息常量初始化，默认填充的信息是一种常用的配置方式，详细的配置如程序清单 7.22 所示。

程序清单 7.22 GPIO 设备信息初始化配置

```

/** \brief GPIO 设备信息 */
const am_zmf2x6_gpio_devinfo_t __g_gpio_devinfo = {
    ZMF2X6_GPIO_BASE,          /**< \brief GPIO 控制器寄存器块基址 */
    ZMF2X6_SYSCFG_BASE,        /**< \brief SYSCFG 配置寄存器块基址 */
    ZMF2X6_EXTI_BASE,          /**< \brief 外部事件控制器寄存器块基址 */

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

    PIN_INT_MAX,               /**< \brief GPIO 支持的引脚中断号数量 */
    __g_gpio_infomap,          /**< \brief 引脚触发信息映射 */
    __g_gpio_trinfos,          /**< \brief 引脚触发信息内存 */
    __zmf2x6_plfm_gpio_init,
    __zmf2x6_plfm_gpio_deinit
};

```

7.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

平台初始化函数和解初始化函数定义如程序清单 7.23 所示。

程序清单 7.23 平台初始化函数和解初始化函数定义

```

/** \brief GPIO 平台初始化 */

```

```

void __zmf2x6_plfm_gpio_init (void)
{

    /* 使能 GPIO 相关外设时钟 */

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

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

    /* 复位 GPIO 相关外设 */
    am_zmf2x6_clk_reset(CLK_GPIOA);
    am_zmf2x6_clk_reset(CLK_GPIOB);
    am_zmf2x6_clk_reset(CLK_GPIOC);
    am_zmf2x6_clk_reset(CLK_GPIOD);
    am_zmf2x6_clk_reset(CLK_SYSCFG);
}

/** \brief GPIO 平台去初始化 */
void __zmf2x6_plfm_gpio_deinit (void)
{

    /* 复位 GPIO 相关外设 */
    am_zmf2x6_clk_reset(CLK_GPIOA);
    am_zmf2x6_clk_reset(CLK_GPIOB);
    am_zmf2x6_clk_reset(CLK_GPIOC);
    am_zmf2x6_clk_reset(CLK_GPIOD);
    am_zmf2x6_clk_reset(CLK_SYSCFG);

    /* 禁能 GPIO 相关外设时钟 */

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

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

```

7.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在标准层实例公共头文件 `am_zmf2x6_inst_init.h` 中加入它们的声明，如所示程序清单 7.24。

程序清单 7.24 初始化解初始化声明

```
/**
 * \brief GPIO 实例初始化
 *
 * \param 无
 *
 * \return 返回 AW_OK 为初始化成功，其它为初始化失败
 */
int am_zmf2x6_gpio_inst_init (void);

/**
 * \brief GPIO 实例解初始化
 *
 * \param 无
 *
 * \return 无
 */
void am_zmf2x6_gpio_inst_deinit (void);
```

用户使用时，应该先调用实例初始化函数 `am_zmf2x6_gpio_inst_init` 函数，初始化 GPIO，并返回执行函数的状态。

至此，用户配置文件开发完毕。

7.5 例程文件开发

例程文件开发主要分为两个部分，HW 例程和 STD 例程。

调试例程时，应该在 `template` 工程下进行，不需要新建其它工程，只需要将新建的文件加入到工程结构的 `user_code` 下。例程文件会与原先的 `main.c` 文件冲突（因为都存在 `main` 函数），这时，只需要调试时将原先的 `main.c` 不要包含在工程编译中，例程文件开发完毕后，再恢复将 `main.c` 包含在工程中，同时将例程文件移除工程即可。

7.5.1 例程内容开发

1. HW 层实现示例

这里使用 HW 层来实现按键切换 LED 状态，相关代码如程序清单 7.25 所示。

程序清单 7.25 按键切换 LED 状态

```
/** [src_zmf2x6_core_hw_gpio] */
#include "ametal.h"
#include "am_gpio.h"
#include "am_delay.h"
#include "am_clk.h"
#include "am_vdebug.h"
```



```

#include "am_zmf2x6.h"
#include "hw/amhw_zmf2x6_gpio.h"
#include "demo_zmf2x6_entries.h"
#include "demo_zmf2x6_core_entries.h"

#define INPUT_PIN  PIOB_2 /**< \brief 输入引脚 */
#define OUTPUT_PIN PIOB_3 /**< \brief 输出引脚 */

void demo_zmf2x6_hw_gpio_entry (amhw_zmf2x6_gpio_t *p_hw_gpio,
                                int32_t          input_pin,
                                int32_t          output_pin)
{
    uint32_t dir_input = 0;

    /* 取消引脚的复用功能 */
    amhw_zmf2x6_gpio_pin_afr_set(p_hw_gpio, AMHW_ZMF2X6_GPIO_AF_DEFAULT, input_pin);

    /* 设置引脚的方向为输入及设置引脚的模式 */
    amhw_zmf2x6_gpio_pin_set(p_hw_gpio, AMHW_ZMF2X6_GPIO_MODE_IPU << 2 | 0x00, input_pin);

    /* 使能引脚的上拉电阻 */
    amhw_zmf2x6_gpio_pin_out_high(p_hw_gpio, input_pin);

    /* 取消引脚的复用功能 */
    amhw_zmf2x6_gpio_pin_afr_set(p_hw_gpio, AMHW_ZMF2X6_GPIO_AF_DEFAULT, output_pin);

    /* 设置引脚的方向为输出及设置引脚的模式 */
    amhw_zmf2x6_gpio_pin_set(p_hw_gpio, AMHW_ZMF2X6_GPIO_MODE_OUT_PP << 2 | 0x00,
output_pin);

    /* 设置引脚的输出速度 */
    amhw_zmf2x6_gpio_pin_dir_output(p_hw_gpio, AMHW_ZMF2X6_GPIO_SPEED_10MHz, output_pin);

    /* 使能引脚的上拉电阻 */
    amhw_zmf2x6_gpio_pin_out_high(p_hw_gpio, output_pin);

    while (1) {

        dir_input = amhw_zmf2x6_gpio_pin_get(p_hw_gpio, input_pin);

        if (dir_input == 0) {
            am_gpio_toggle(output_pin);
            am_mdelay(500);
        }
    }
}

```

```

    }
}

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_hw_gpio_entry (void)
{
    //    AM_DBG_INFO("demo zmf2x6_core hw gpio!\r\n");

    /* 使能时钟 */
    am_clk_enable(CLK_GPIOA);
    am_clk_enable(CLK_SYSCFG);

    demo_zmf2x6_hw_gpio_entry(ZMF2X6_GPIO, INPUT_PIN, OUTPUT_PIN);
}

```

2. 标准层实现示例

这里使用标准层来实现按键切换 LED 状态，通过标准接口实现，相关代码如程序清单 7.26 所示。

程序清单 7.26 按键切换 LED 状态

```

#include "ametal.h"
#include "am_vdebug.h"
#include "am_gpio.h"
#include "am_delay.h"
#include "am_zmf2x6.h"
#include "demo_std_entries.h"
#include "demo_zmf2x6_core_entries.h"

#define INPUT_PIN  PIOB_2 /**< \brief 输入引脚 */
#define OUTPUT_PIN PIOB_3 /**< \brief 输出引脚 */

void demo_std_gpio_entry (int input_pin, int output_pin)
{
    uint8_t dir_input = 0;

    /* 一般默认按键对应的 GPIO 引脚作为输入 */
    am_gpio_pin_cfg(input_pin, AM_GPIO_INPUT | AM_GPIO_PULLUP);

    /* 一般默认 LED 对应的 GPIO 引脚作为输出 */
    am_gpio_pin_cfg(output_pin, AM_GPIO_OUTPUT_INIT_LOW | AM_GPIO_PULLUP);

    while (1) {

```

```
dir_input = am_gpio_get(input_pin);
if (dir_input == 0) {

    /* 消除抖动 */
    while (am_gpio_get(input_pin) == 0);

    /* 翻转 output_pin */
    am_gpio_toggle(output_pin);
}
}

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_std_gpio_entry (void)
{
    AM_DBG_INFO("demo zmf2x6_core std gpio!\r\n");

    demo_std_gpio_entry(INPUT_PIN, OUTPUT_PIN);
}
```

所有例程开发完毕后，整个外设的开发工作也就圆满结束了。

可以提交相关文件，以完成整个开发任务。提交文件时，不用提交工程文件，只需要提交开发的源文件和头文件即可。

第8章 通用外设驱动开发——CRC

8.1 标准层了解

CRC 标准层仅包含一个文件，am_crc.h 文件，在 ametal\interface 找到该文件打开。

8.1.1 了解相关类型定义

打开 am_crc.h 文件，可以看到该文件主要定义了 3 个结构体和 1 个 handle 类型，如程序清单 8.1 所示，下面对每个结构体做详细介绍。

程序清单 8.1 CRC 标准接口文件结构体定义

```
/**
 * \brief CRC 模型
 */
typedef struct am_crc_pattern {
    uint8_t      width;      /**< \brief CRC 宽度(如：CRC5 的宽度为 5) */
    uint32_t      poly;      /**< \brief CRC 生成多项式 */
    uint32_t      initvalue; /**< \brief CRC 初始值 */
    am_bool_t     refin;     /**< \brief 如果输入数据需要位反转，则该值为 TRUE */
    am_bool_t     refout;    /**< \brief 如果输出结果值需要位反转，则该值为 TRUE */
    uint32_t      xorout;    /**< \brief 输出异或值 */
} am_crc_pattern_t;

/**
 * \brief CRC 驱动函数结构体
 */
struct am_crc_drv_funcs {

    /** \brief 根据 CRC 模型初始化 CRC，为 CRC 计算做好准备 */
    int (*pfn_crc_init)(void *p_cookie, am_crc_pattern_t *p_pattern);

    /** \brief CRC 计算 */
    int (*pfn_crc_cal)(void *p_cookie, const uint8_t *p_data, uint32_t nbytes);

    /** \brief 获取 CRC 计算结果 */
    int (*pfn_crc_final)(void *p_cookie, uint32_t *p_value);
};

/**
 * \brief CRC 服务
 */
typedef struct am_crc_serv {

    /** \brief CRC 驱动函数结构体指针 */
    struct am_crc_drv_funcs *p_funcs;
```

```

/** \brief 用于驱动函数的第一个参数 */
void
    *p_drv;
} am_crc_serv_t;

/** \brief CRC 标准服务操作句柄类型定义*/
typedef am_crc_serv_t *am_crc_handle_t;

```

1. struct am_crc_pattern 结构体

该结构体包含了 CRC 模型相关信息，主要是 CRC 宽度、生成多项式、初始值、输入输出位反转设置以及输出异或值，后文提到的 CRC 模型初始化函数，就需要设置并传入该结构体，根据 CRC 模型相关信息进行 CRC 计算并获取 CRC 计算结果。

2. struct am_crc_drv_funcs 结构体

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层需要实现 3 个函数：

- CRC 模型初始化函数，为 CRC 计算做好准备；
- CRC 计算；
- 获取 CRC 计算结果。

因此，在驱动层编写函数时，应该向标准接口层提供这三个函数。

这些函数的有一个共同点，就是第一个参数都是 void 型的指针，也就是说，驱动层提供的函数，标准接口层使用时传递的第一个参数并无确定类型，标准接口层不关心第一个参数的具体类型，使用时只负责传递就可以了。

3. typedef struct am_crc_serv 结构体

该结构体定义了标准的 CRC 服务，包含了上文提到的驱动函数结构体的指针和一个 void 型的指针。

这个标准的 CRC 服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

● p_funcs 成员赋值

这是一个指向 struct am_crc_drv_funcs 结构体的指针，我们只需要在驱动层定义一个 struct am_crc_drv_funcs 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那三个函数地址。然后，将 struct am_crc_drv_funcs 结构体变量的地址赋值给 p_funcs 成员即可。

● p_drv 成员赋值

这个变量是 void 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动自行决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

4. am_crc_handle_t 的定义

这个是一个标准的操作句柄，本质上就是 struct am_crc_serv 指针类型，驱动层的初始化函数需要返回该类型的一个 handle 给用户。这样，用户获取到 handle，就可以在需要使用标准接口层函数时，将该 handle 传递给标准接口层了。标准接口层就可以通过该 handle (struct am_crc_serv 结构体指针) 访问到驱动层提供的函数了。

8.1.2 了解标准接口函数

在 am_crc.h 文件相关的结构体、类型定义后，就是具体函数的实现，如程序清单 8.2 所示。

程序清单 8.2 CRC 标准接口函数

```
/**
 * \brief 根据 CRC 模型初始化 CRC，为 CRC 计算做好准备
 *
 * \param[in] handle : CRC 标准服务操作句柄
 * \param[in] p_pattern : 指向 CRC 模型信息结构体指针
 *
 * \retval AM_OK : 初始化成功
 * \retval -AM_EINVAL : 初始化失败，参数错误
 * \retval -AM_ENOTSUP : 初始化失败，CRC 模型不支持
 */
am_static_inline
int am_crc_init (am_crc_handle_t handle, am_crc_pattern_t *p_pattern)
{
    return handle->p_funcs->pfn_crc_init(handle->p_drv, p_pattern);
}

/**
 * \brief CRC 计算(数据处理)
 *
 * \param[in] handle : CRC 标准服务操作句柄
 * \param[in] p_data : 指向待计算数据缓冲区
 * \param[in] nbytes : 待计算数据的个数
 *
 * \retval AM_OK : CRC 计算成功
 * \retval -AM_EINVAL : CRC 计算失败，参数错误
 */
am_static_inline
int am_crc_cal (am_crc_handle_t handle, const uint8_t *p_data, uint32_t nbytes)
{
    return handle->p_funcs->pfn_crc_cal(handle->p_drv, p_data, nbytes);
}

/**
 * \brief 获取 CRC 计算结果值
 *
 * \param[in] handle : CRC 标准服务操作句柄
 * \param[out] p_value : 用于获取 CRC 计算结果值
 *
 * \retval AM_OK : 获取 CRC 计算结果值成功
 */
```

```

* \retval -AM_EINVAL : 获取 CRC 计算结果值失败, 参数错误
*/
am_static_inline
int am_crc_final (am_crc_handle_t handle, uint32_t *p_value)
{
    return handle->p_funcs->pfn_crc_final(handle->p_drv, p_value);
}

```

很简单的三个函数全都是用内联函数就搞定了。因为它本身就没做什么事情, 只是简单的调用了一下驱动层提供的函数而已。可以看到, 在调用驱动层提供的函数时, 就是将 struct am_crc_serv 结构体中的 p_drv 成员作为第一个参数传递。

到此我们就分析完了 CRC 外设标准层文件, 清楚了驱动层需要为标准接口层提供哪些内容。

8.2 HW 层开发

现在我们打开 ZMF2x6 芯片用户手册, 对照手册完成 CRC 外设 HW 层的开发。打开 ametal\soc\zlg\drivers\include\crc\hw 路径, 在该路径下创建名为 amhw_zmf2x6_crc.h 的文件。文件创建完成之后, 需要在 zmf2x6_periph_map.h 文件中包含新添加的 amhw_zmf2x6_crc.h 文件, 打开 ametal\soc\zlg\zmf2x6 路径找到该文件, 添加 #include "hw/amhw_zmf2x6_crc.h"。

8.2.1 添加寄存器列表

对照 ZMF2x6 用户手册的 CRC 章节, 可以看到, CRC 章节部分, 共有 4 个寄存器, 简单描述如下:

- CRC_DR (数据寄存器): 写入 CRC 计算器的新数据时, 作为输入寄存器; 读取时返回 CRC 计算结果;
- CRC_IDR (独立数据寄存器): 可用于临时存放 1 字节的数据。寄存器 CRC_CTRL 的 RESET 位产生的 CRC 复位对本寄存器没有影响;
- CRC_CTRL (控制寄存器): 复位 CRC 计算单元, 设置数据寄存器为 0xFFFFFFFF;
- CRC_REVERSE (数据翻转寄存器): 寄存器写入数据 HWDData[31:0], 寄存器读出数据 HWDData[0:31];

到此 CRC 的各个寄存器及其功能基本阅读完毕, 我们也大致了解了各个寄存器功能以及要配置各个功能, 其他寄存器该如何配合等。下面我们先定义一个寄存器的结构体, 结构体成员全部小写, 程序清单如程序清单 8.3 所示。

程序清单 8.3 寄存器结构体定义

```

/**
 * \brief CRC 寄存器块结构体
 */
typedef struct amhw_zmf2x6_crc {
    __IO uint32_t crcdr;    /**< \brief CRC Data register */
    __IO uint32_t crcidr;   /**< \brief CRC independent register */
    __IO uint32_t crcctl;   /**< \brief CRC control register */
    uint32_t reserved[5];
    __IO uint32_t crcrvs;   /**< \brief CRC REVERSE register */
}

```

```
} amhw_zmf2x6_crc_t;
```

这个小节我们定义了 CRC 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

8.2.2 编写寄存器相关函数

1. CRC_DR（数据寄存器）

通过用户手册，我们了解该寄存器有含义的寄存器就 1 个，由 32 位组成。因此它的操作函数也较为简单，操作函数如程序清单 10.5 所示。

程序清单 8.4 CRC_DR 操作函数

```
/**
 * \brief CRC 32bit write data
 *
 * \param[in] p_hw_crc : the pointer to the CRC register base address
 * \param[in] data      : the data to be caculated
 *
 * retval : none
 */
am_static_inline
void amhw_zmf2x6_crc_32bit_write_data (amhw_zmf2x6_crc_t *p_hw_crc,
                                       uint32_t data)
{
    p_hw_crc->crcdr = data;
}

/**
 * \brief CRC 32 bit read data
 *
 * \param[in] p_hw_crc : the pointer to the CRC register base address
 *
 * \retval : the result of caculate value
 */
am_static_inline
uint32_t amhw_zmf2x6_crc_32bit_read_data (amhw_zmf2x6_crc_t *p_hw_crc)
{
    return p_hw_crc->crcdr;
}
```

2. CRC_IDR（独立数据寄存器）

该寄存器有含义的寄存器就 1 个，由 8 位组成，其内联函数如程序清单 8.5 所示。

程序清单 8.5 CRC_IDR 操作函数

```
/**
```



```

* \brief CRC 8bit write data
*
* \param[in] p_hw_crc : the pointer to the CRC register base address
* \param[in] data      : the data to be stored
*
* retval : none
*
* \note this register can not caculate data
*/
am_static_inline
void amhw_zmf2x6_crc_8bit_write_data (amhw_zmf2x6_crc_t *p_hw_crc, uint8_t data)
{
    p_hw_crc->crcidr = data;
}

/**
* \brief CRC 8 bit read data
*
* \param[in] p_hw_crc : the pointer to the CRC register base address
*
* \retval : the stored data
*
* \note this register can not caculate data
*/
am_static_inline
uint8_t amhw_zmf2x6_crc_8bit_read_data (amhw_zmf2x6_crc_t *p_hw_crc)
{
    return p_hw_crc->crcidr;
}

```

3. CRC_CTRL (控制寄存器)

该寄存器的 RESET 位用于复位 CRC 计算单元，设置数据寄存器为 0xFFFFFFFF，通过了解，该寄存器有含义的位只有这 1 个，其操作函数如程序清单 8.6 所示。

程序清单 8.6 CRC_CTRL 操作函数

```

/**
* \brief CRC reset
*
* \param[in] p_hw_crc : the pointer to the CRC register base address
*
* \retval : none
*/
am_static_inline
void amhw_zmf2x6_crc_reset (amhw_zmf2x6_crc_t *p_hw_crc)
{

```

```
p_hw_crc->crctl = 1;
}
```

4. CRC_REVERSE（数据翻转寄存器）

该寄存器的 REVERSE 位用于设置数据翻转，该寄存器有含义的位只有这 1 个，操作函数如程序清单 8.7 所示。

程序清单 8.7 CRC_REVERSE 操作函数

```
/**
 * \brief CRC data reverse
 *
 * \param[in] p_hw_crc : the pointer to the CRC register base address
 * \param[in] flag : 1:数据翻转 0: 数据不翻转
 *
 * \retval : none
 */
am_static_inline
void amhw_zmf2x6_crc_reverse_set (amhw_zmf2x6_crc_t *p_hw_crc, uint8_t flag)
{
    p_hw_crc->crctx = (p_hw_crc->crctx & ~(0x1)) | flag;
}
```

8.2.3 添加其他功能性函数

在每个寄存器的操作函数编写完毕后，可能 HW 层的接口还有所欠缺，需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到，可以暂时不用考虑，直接进入驱动层的开发。在后续如果发现 HW 层欠缺哪些接口，可以反过来再增加。当然，也有可能 HW 层接口没有什么欠缺了，那样就无需再开发其它函数了。

8.2.4 添加指向寄存器块的指针

可以看到，在 HW 层编写的函数中，第一个参数均为指向 CRC 寄存器块的指针，只需要把 CRC 的基地址使用(amhw_crc_t *)强制转换即可作为这些函数的第一个参数。这样转换后，操作结构体的成员，实际上就是操作具体地址上的寄存器了。

为了方便用户或驱动层使用，可以将该转换使用一个宏定义好，用户直接使用宏即可。在 AMetal 中，由于这些内容与具体型号的芯片相关，这样的宏统一定义到 zmf2x6_periph_map.h 文件中，如程序清单 8.8 所示。

程序清单 8.8 外设中加入 CRC 的定义

```
/**
 * \name 外设定义
 * @{
 */
/** \brief NVIC 寄存器块指针 */
#define ZMF2X6_INT ((amhw_arm_nvic_t *)ZMF2X6_NVIC_BASE)

/** \brief 窗口看门狗(WWDG)寄存器块指针 */
```

```
#define ZMF2X6_WWDG      ((amhw_zmf2x6_wwdg_t      *)ZMF2X6_WWDG_BASE)

/** \brief 循环冗余校验计算单元(CRC)寄存器块指针 */
#define ZMF2X6_CRC      ((amhw_zmf2x6_crc_t      *)ZMF2X6_CRC_BASE)

/**
 * @} zmf2x6_if_periph_map
 */
```

后续用户或驱动层使用 HW 层接口时，第一个参数直接使用 ZMF2X6_CRC 即可。至此，整个 HW 层的开发工作就结束了，如果后续开发中发现缺少某些函数，还可以再添加。

8.3 驱动层开发

参考 5.3.1 章节，新建 am_zmf2x6_crc.c 和 am_zmf2x6_crc.h 驱动文件，搭建相应的文件框架，并按规则放到对应的路径下。

8.3.1 编写结构性程序

驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备（CRC 就是一个设备）都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

可以暂时定义设备信息结构体如程序清单 8.9 所示。后续可以根据实际情况修改。

程序清单 8.9 CRC 设备信息结构体定义

```
/**
 * \brief CRC 设备信息
 */
typedef struct am_zmf2x6_crc_devinfo {

    /** \brief CRC 寄存器块的基地址 */
    uint32_t    crc_reg_base;

    /** \brief 平台初始化函数，如打开时钟，配置引脚等工作 */
    void        (*pfn_plfm_init)(void);

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

} am_zmf2x6_crc_devinfo_t;
```

● 设备结构体定义

由于 CRC 设备需要提供标准的 CRC 服务，因此，该设备至少包含一个 CRC 标准服务结构体成员，也就是 `am_crc_serv_t` 类型成员，其次，为了在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 8.10 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 8.10 CRC 设备结构体定义

```
/**
 * \brief CRC 设备实例
 */
typedef struct am_zmf2x6_crc_dev {

    /** \brief CRC 标准服务 */
    am_crc_serv_t          crc_serve;

    /** \brief 指向 CRC 设备信息的指针 */
    const am_zmf2x6_crc_devinfo_t *p_devinfo;

} am_zmf2x6_crc_dev_t;
```

2. 搭建函数框架

函数主要分为两部分，一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于 CRC 这种已经有标准接口层的函数，通常驱动层只提供两个函数给用户，初始化函数和解初始化函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为用于标准接口的 `handle`。

解初始化函数主要是当不需要再使用某个外设时，释放掉其占用的相关资源，参数为设备指针，无返回值。示例代码如程序清单 8.11 所示。

程序清单 8.11 驱动层函数定义示例

```
/**
 * \brief CRC 初始化
 */
am_crc_handle_t am_zmf2x6_crc_init (am_zmf2x6_crc_dev_t      *p_dev,
                                   const am_zmf2x6_crc_devinfo_t *p_devinfo)
{
    return &(p_dev->crc_serve);
}

/**
 * \brief CRC 解初始化
 */
```

```
void am_zmf2x6_crc_deinit (am_crc_handle_t handle)
{
}
```

同时为了用户能够使用，还需要将这两个函数的声明放入到相应的头文件中。示例代码如程序清单 8.12 所示。

程序清单 8.12 驱动函数声明

```
/**
 * \brief CRC 初始化
 *
 * \param[in] p_dev      : 指向 CRC 设备的指针
 * \param[in] p_devinfo : 指向 CRC 设备信息的指针
 *
 * \return CRC 标准服务操作句柄，如果为 NULL，表明初始化失败
 */
am_crc_handle_t am_zmf2x6_crc_init (am_zmf2x6_crc_dev_t      *p_dev,
                                   const am_zmf2x6_crc_devinfo_t *p_devinfo);

/**
 * \brief CRC 去初始化
 *
 * \param[in] handle : am_zmf2x6_crc_init() 初始化函数获得的 CRC 服务句柄
 *
 * \return 无
 */
void am_zmf2x6_crc_deinit (am_crc_handle_t handle);
```

- 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 3 个函数，根据标准接口层中的 struct am_crc_drv_funcs 结构体，我们需要在.c 文件中定义 3 个函数。

由于这些函数全部是放在结构体中，通过函数指针的形式提供给上层，并不需要将这些函数声明到.h 文件中供用户直接使用，因此全部以 static 的形式定义成内部函数。定义好函数后，再定义一个 struct am_crc_drv_funcs 结构体，将函数放在其中。示例代码如程序清单 8.13 所示。

程序清单 8.13 提供给标准层函数定义

```
/** \brief 根据 CRC 模型初始化 CRC，为 CRC 计算做好准备 */
static int __crc_init (void *p_cookie, am_crc_pattern_t *p_pattern);

/** \brief CRC 计算 */
static int __crc_cal (void *p_cookie, const uint8_t *p_data, uint32_t nbytes);

/** \brief 获取 CRC 计算结果 */
static int __crc_final (void *p_cookie, uint32_t *p_value);
```

```

/**
 * \brief CRC 服务函数
 */
static const struct am_crc_drv_funcs __g_crc_drvfuncs = {
    __crc_init,
    __crc_cal,
    __crc_final
};

/**
 * \brief 初始化 CRC
 */
static int __crc_init (void *p_cookie, am_crc_pattern_t *p_pattern)
{

    return AM_OK;
}

/**
 * \brief CRC 计算
 */
static int __crc_cal (void *p_cookie, const uint8_t *p_data, uint32_t nbytes)
{

    return AM_OK;
}

/**
 * \brief 获取 CRC 计算结果
 */
static int __crc_final (void *p_cookie, uint32_t *p_value)
{

    return AM_OK;
}

```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。

8.3.2 驱动函数实现

通过上面的函数框架搭建，共计 5 个函数，3 个功能性函数面向标准层，2 个初始解初始化函数面向用户，下面逐步实现。

1. am_zmf2x6_crc_init 函数

该函数主要完成两个功能：

- a) 初始化设备中的标准服务，将成员赋值，确定 p_drv 的值，p_drv 的值确定了提供给标准接口层的那 3 个函数的第一个参数是什么。那 3 个函数才能根据这儿的设置

去使用。（void *类型无法使用，必须转换为具体的一种类型才能使用），通常情况下，p_drv 都是设置为设备的地址，这样方便访问整个设备的相关内容。

- b) 初始化硬件相关设置，该函数的示例代码如程序清单 8.14 所示。

程序清单 8.14 am_zmf2x6_crc_init 函数示例代码

```
/**
 * \brief CRC 初始化
 */
am_crc_handle_t am_zmf2x6_crc_init (am_zmf2x6_crc_dev_t      *p_dev,
                                   const am_zmf2x6_crc_devinfo_t *p_devinfo)
{
    if (p_dev == NULL || p_devinfo == NULL) {
        return NULL;
    }

    p_dev->p_devinfo      = p_devinfo;
    p_dev->p_pattern      = NULL;

    p_dev->crc_serve.p_funcs = (struct am_crc_drv_funcs *)&__g_crc_drvfuncs;
    p_dev->crc_serve.p_drv   = p_dev;

    if (p_dev->p_devinfo->pfn_plfm_init) {
        p_dev->p_devinfo->pfn_plfm_init();
    }

    return &(p_dev->crc_serve);
}
```

2. am_zmf2x6_crc_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放。因此，该函数仅简单的将几个不再使用的变量置为 NULL 即可。示例代码如程序清单 8.15 所示。

程序清单 8.15 am_zmf2x6_crc_deinit 函数示例代码

```
/**
 * \brief CRC 解初始化
 */
void am_zmf2x6_crc_deinit (am_crc_handle_t handle)
{
    am_zmf2x6_crc_dev_t *p_dev   = (am_zmf2x6_crc_dev_t *)handle;

    if (NULL == handle) {
        return ;
    }
}
```

```

p_dev->p_pattern = NULL;

p_dev->crc_serve.p_funcs = NULL;
p_dev->crc_serve.p_drv = NULL;

if (p_dev->p_devinfo->pfn_plfm_deinit) {
    p_dev->p_devinfo->pfn_plfm_deinit();
}
}

```

3. __crc_init 函数

该函数用于初始化 CRC 模型，包括 CRC 宽度、生成多项式、初始值、输入输出位反转设置以及输出异或值。

用户只需要定义并设置好 CRC 模型相关参数，将其指针传入该函数即可。

因此，我们需要在设备信息中加入指向 CRC 模型的指针。加入后设备信息的定义如程序清单 8.16 所示

程序清单 8.16 设备信息添加 CRC 模型指针

```

/**
 * \brief CRC 设备实例
 */
typedef struct am_zmf2x6_crc_dev {

    /** \brief CRC 标准服务 */
    am_crc_serv_t      crc_serve;

    /** \brief 指向 CRC 设备信息的指针 */
    const am_zmf2x6_crc_devinfo_t *p_devinfo;

    /** \brief 指向 CRC 模型的指针 */
    am_crc_pattern_t    *p_pattern;

} am_zmf2x6_crc_dev_t;

```

有了该信息之后，就可以根据 CRC 模型的设置进行 CRC 计算，获取 CRC 计算结果。CRC 模型初始化函数如程序清单 8.17 所示。

程序清单 8.17 CRC 模型初始化

```

/**
 * \brief 初始化 CRC
 */
static int __crc_init (void *p_cookie, am_crc_pattern_t *p_pattern)
{
    am_zmf2x6_crc_dev_t *p_dev = (am_zmf2x6_crc_dev_t *)p_cookie;

```



```

amhw_zmf2x6_crc_t  *p_hw_crc    = NULL;

if (p_dev == NULL || p_pattern == NULL) {
    return -AM_EINVAL;
}

if (p_pattern->poly    != 0x04C11DB7 || p_pattern->width != 32 ||
    p_pattern->initvalue != 0xFFFFFFFF ) {

    return -AM_ENOTSUP;
}

p_dev->p_pattern = p_pattern;

p_hw_crc = (amhw_zmf2x6_crc_t*)(p_dev->p_devinfo->crc_reg_base);

/* 重置寄存器 CRC_DATA 为 0xFFFF FFFF */
amhw_zmf2x6_crc_reset(p_hw_crc);

return AM_OK;
}

```

4. __crc_cal 函数

CRC 根据模型中位反转设置进行计算，如程序清单 8.18 所示。

程序清单 8.18 CRC 计算

```

/**
 * \brief CRC 计算
 */
static int __crc_cal(void *p_cookie, const uint8_t *p_data, uint32_t nbytes)
{
    uint32_t i;
    uint32_t tdata;
    am_zmf2x6_crc_dev_t *p_dev    = (am_zmf2x6_crc_dev_t *)p_cookie;
    amhw_zmf2x6_crc_t  *p_hw_crc = (amhw_zmf2x6_crc_t*)(p_dev->p_devinfo->crc_reg_base);

    if (p_dev->p_pattern == NULL || p_data == NULL) {
        return -AM_EINVAL;
    }

    if (nbytes % 4 != 0) {
        return -AM_ENOTSUP;
    }

    for (i = 0; i < nbytes; i+=4) {

```

```

if (p_dev->p_pattern->refin == AM_TRUE) {

    /* 输入的每一个字节都进行位倒置 */
    tdata = (__rev8bit(p_data[i] << 24) | (__rev8bit(p_data[i+1] << 16) |
        (__rev8bit(p_data[i+2] << 8) | __rev8bit(p_data[i+3]));
    } else {

        tdata = (p_data[i] << 24) | (p_data[i+1] << 16) |
            (p_data[i+2] << 8) | p_data[i+3];
    }

    amhw_zmf2x6_crc_32bit_write_data(p_hw_crc, tdata);
}

return AM_OK;
}

```

5. __crc_final 函数

这个函数就是获取 CRC 计算结果，示例代码如程序清单 8.19 所示。

程序清单 8.19 获取 CRC 计算结果

```

/**
 * \brief 获取 CRC 计算结果
 */
static int __crc_final(void *p_cookie, uint32_t *p_value)
{
    am_zmf2x6_crc_dev_t *p_dev = (am_zmf2x6_crc_dev_t *)p_cookie;
    amhw_zmf2x6_crc_t *p_hw_crc = (amhw_zmf2x6_crc_t *) (p_dev->p_devinfo->crc_reg_base);

    if (p_dev->p_pattern == NULL) {
        return -AM_EINVAL;
    }

    *p_value = amhw_zmf2x6_crc_32bit_read_data(p_hw_crc);

    if (p_dev->p_pattern->refout == AM_TRUE){

        /* 计算结果进行倒置 */
        *p_value = (__rev8bit((uint8_t)(*p_value & 0x000000ff) << 24) |
            (__rev8bit((uint8_t)((*p_value & 0x0000ff00) >> 8)) << 16) |
            (__rev8bit((uint8_t)((*p_value & 0x00ff0000) >> 16)) << 8) |
            __rev8bit((uint8_t)((*p_value & 0xff000000) >> 24)));
    }
}

```

```

/* 计算结果与 xorout 异或 */
*p_value ^= p_dev->p_pattern->xorout;

p_dev->p_pattern = NULL;

return AM_OK;
}

```

到现在 CRC 支持的所有函数开发完毕，整个 CRC 的驱动层开发也就结束了。

8.4 用户配置文件编写

开发完驱动层我们知道，用户使用一个外设的过程是，先调用驱动初始化函数，该函数返回一个 handle，后续的所有操作，用户直接使用该 handle 作为参数调用标准接口层函数即可。

由于用户调用驱动初始化函数时需要传入设备地址和设备信息地址作为参数。为了方便用户使用，我们需要将设备变量和设备信息常量定义好，用户调用驱动初始化函数时，只需要使用定义好的设备变量和设备信息常量即可，不需要再额外定义。

设备变量和设备信息常量定义在 am_hwconf_zmf2x6_crc.c 文件中，相应的声明放在 am_hwconf_zmf2x6_crc.h 文件中或在 am_zmf2x6_inst_init.h 中。所以我们需要创建的配置文件至少包含 1 个 am_hwconf_zmf2x6_crc.c 文件。

8.4.1 定义设备常量于设备信息常量

创建配置文件后，我们需要在里面加入设备变量，设备变量只需要定义，不用初始化。设备信息常量定义时初始化，默认填充的信息是一种常用的配置方式，CRC 设备信息配置如程序清单 8.20 所示。

程序清单 8.20 设备信息

```

/** \brief CRC 设备信息 */
static const am_zmf2x6_crc_devinfo_t __g_crc_devinfo = {
    ZMF2X6_CRC_BASE,                /**< \brief CRC 寄存器块基址 */
    __zmf2x6_crc_plfm_init,          /**< \brief 平台初始化 */
    __zmf2x6_crc_plfm_deinit        /**< \brief 平台去初始化 */
};

/** \brief CRC 设备定义 */
static am_zmf2x6_crc_dev_t __g_crc_dev;

```

8.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

由于 CRC 无相关引脚需要配置，时钟的话仅需使能 CRC 外设时钟。因此，平台初始化函数和解初始化函数定义如程序清单 8.21 所示。

程序清单 8.21 平台初始化解初始化

```

/** \brief CRC 平台初始化 */

```

```
static void __zmf2x6_crc_plfm_init (void)
{
    am_clk_enable(CLK_CRC);
}

/** \brief 解除 CRC 平台初始化 */
static void __zmf2x6_crc_plfm_deinit (void)
{
    am_clk_disable(CLK_CRC);
}
```

8.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在头文件中加入它们的声明，如程序清单 8.22 所示。

程序清单 8.22 初始化解初始化声明

```
/**
 * \brief CRC 初始化
 *
 * \param[in] p_dev      : 指向 CRC 设备的指针
 * \param[in] p_devinfo : 指向 CRC 设备信息的指针
 *
 * \return CRC 标准服务操作句柄，如果为 NULL，表明初始化失败
 */
am_crc_handle_t am_zmf2x6_crc_init (am_zmf2x6_crc_dev_t      *p_dev,
                                   const am_zmf2x6_crc_devinfo_t *p_devinfo);

/**
 * \brief CRC 去初始化
 *
 * \param[in] handle : am_zmf2x6_crc_init() 初始化函数获得的 CRC 服务句柄
 *
 * \return 无
 */
void am_zmf2x6_crc_deinit (am_crc_handle_t handle);
```

用户使用时，应该先调用平台初始化函数 `am_zmf2x6_crc_inst_init` 函数，然后直接使用 `am_zmf2x6_crc_init(&__g_crc_dev, &__g_crc_devinfo)` 初始化 CRC，并保存其返回的 `handle`，后续直接使用保存的 `handle` 作为标准接口的第一个参数使用标准接口函数即可。

至此，用户配置文件开发完毕。

8.5 例程文件开发

参考 5.5.1 章节新建用户例程文件，搭建文件的基本框架，准备 CRC 例程内容的编写。

8.5.1 例程内容开发

CRC 例程主要是根据固定的生成多项式得到任意 32 位全字的 CRC 计算结果，主要应用于核实数据传输或者数据存储的正确性和完整性。

了解了需要实现的功能，接下来我们来实现 CRC 的功能，如程序清单 8.23 所示，直接调用驱动层实现的函数即可。

程序清单 8.23 标准层实现内容

```
#include "ametal.h"
#include "am_vdebug.h"
#include "am_zmf2x6_crc.h"
#include "am_zmf2x6_inst_init.h"
#include "demo_std_entries.h"
#include "demo_zmf2x6_core_entries.h"

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_std_crc_entry (void)
{
    AM_DBG_INFO("demo zmf2x6_core std crc!\r\n");

    am_crc_handle_t crc_handle;

    crc_handle = am_zmf2x6_crc_inst_init();

    demo_std_crc_entry(crc_handle);
}
/** [src_zmf2x6_std_crc] */

/* end of file */
```

所有例程开发完毕后，整个 CRC 外设的开发工作也就圆满结束了。

可以提交相关文件，以完成整个开发任务。提交文件时，不用提交工程文件，只需要提交开发的源文件和头文件即可。

第9章 通用外设驱动开发——ADC

9.1 标准层了解

ADC 标准层包含两个文件 am_adc.h 和 am_adc.c 文件，首先分析一下 am_adc.h 文件，在 ametal\interface 找到该文件打开。

9.1.1 了解相关类型定义

打开 am_adc.h 文件，可以看到该文件主要定义了 2 个函数类型、3 个结构体和 1 个 handle 类型，如程序清单 9.1 所示，下面对每个结构体做详细介绍。

程序清单 9.1 标准接口文件类型定义

```
/**
 * \brief 缓冲区完成回调函数类型定义
 *
 * 启动 ADC 转换时，每个缓冲区都有一个对应的缓冲区完成回调函数，该缓冲区数据
 * 填充满后调用该回调函数。状态参数值(state)表明了本缓冲区数据的有效性：
 * - AM_OK : 数据正确，可以正常使用
 * - AM_ERROR : 数据有误，该缓冲区的数据应该丢弃。
 *
 * 处理转换数据前应该确保转换结束的状态为 AM_OK。如果不为 AM_OK，则序列的后续
 * 转换将自动取消，不会再转换。通常情况下，转换出错可能是由于缓冲区的 buf 过小，
 * 使得 buf 切换需要很频繁，数据处理时间稍长，数据就溢出了。可以增大缓冲区的 buf 大
 * 小以增加 buf 切换的时间间隔。
 */
typedef void (*am_adc_buf_cb_t)(void *p_arg, int state);

/**
 * \brief 转换完成回调函数类型定义
 *
 * 序列完成回调函数在一个序列中各缓冲区填充满后调用,以下情况可视为序列转换完成：
 * - 单个缓冲区，单个缓冲区填充完成即序列转换完成
 * - 多个缓冲区，从第一个缓冲区到最后一个缓冲区依次填充满才为序列转换完成
 *
 * 序列正常完成时，状态参数值为 AM_OK。如果填充某个缓冲区时数据出错，也会调
 * 用该函数，且传递参数为 AM_ERROR,表明该序列错误结束。此时，转换自动停止，如果
 * 该序列中还有未转换的序列，则会自动取消，不再继续转换。
 *
 * 该函数仅作为序列完成的一种通知，数据处理应该在每个缓冲区的回调函数中完成。
 */
typedef void (*am_adc_seq_cb_t)(void *p_arg, int state);

/**
 * \brief ADC 缓冲区描述
 */
```

```

typedef struct am_adc_buf_desc {

    /**
     * \brief ADC 数据缓冲区，其具体类型与 ADC 位数有关，
     *      ADC 位数可以由函数 am_adc_bits_get()获取。
     *
     *      ADC bits      |      buffer type
     *      ----- | -----
     *      1 ~ 8      |      uint8_t/int8_t
     *      9 ~ 16     |      uint16_t/int16_t
     *      17 ~ 32    |      uint32_t/int32_t
     */
    void *p_buf;

    /** \brief 数据缓冲区的长度 */
    uint32_t length;

    /** \brief 该缓冲区数据填满后的回调函数 */
    am_adc_buf_cb_t pfn_complete;

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

/**
 * \brief ADC 驱动函数
 */
struct am_adc_drv_funcs {

    /** \brief 启动 ADC 转换 */
    int (*pfn_adc_start)(void *p_drv,
                        int chan,
                        am_adc_buf_desc_t *p_desc,
                        uint32_t desc_num,
                        uint32_t count,
                        uint32_t flags,
                        am_adc_seq_cb_t pfn_callback,
                        void *p_arg);

    /** \brief 停止转换 */
    int (*pfn_adc_stop)(void *p_drv, int chan);

    /** \brief 获取 ADC 的采样率 */
    int (*pfn_rate_get)(void *p_drv,

```

```

        int        chan,
        uint32_t    *p_rate);

/** \brief 设置 ADC 的采样率，实际采样率可能存在差异 */
int (*pfn_rate_set)(void        *p_drv,
                    int        chan,
                    uint32_t    rate);

/** \brief 获取 ADC 转换精度 */
uint32_t (*pfn_bits_get)(void *p_drv, int chan);

/** \brief 获取 ADC 参考电压 */
uint32_t (*pfn_vref_get)(void *p_drv, int chan);
};

/**
 * \brief ADC 标准服务
 */
typedef struct am_adc_serv {

    /** \brief 指向 ADC 驱动函数的指针 */
    const struct am_adc_drv_funcs *p_funcs;

    /** \brief 驱动函数第一个入口参数 */
    void                        *p_drv;
} am_adc_serv_t;

/** \brief ADC 标准服务操作句柄定义 */
typedef am_adc_serv_t *am_adc_handle_t;

```

1. am_adc_buf_cb_t 类型

该类型为缓冲区完成回调函数类型，被用户定义并做为参数传递给标准接口，最终将被驱动层的__zmf2x6_adc_irq_handle 中断服务程序所调用。

2. am_adc_seq_cb_t 类型

该类型为转换完成回调函数类型，被用户定义并做为参数传递给标准接口，最终将被驱动层的__zmf2x6_adc_irq_handle 中断服务程序所调用。

3. am_adc_buf_desc 结构体

该结构体包含了 adc 缓冲区的信息描述，主要涉及到数据缓冲区的地址，数据缓冲区长度，数据填满后的回调函数（am_adc_buf_cb_t）及对应参数。用户通过实例化该结构体，并作为参数传给启动 ADC 转换标准驱动函数（am_adc_start）。

4. am_adc_drv_funcs 结构体

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层要实现 6 个函数：

- 启动 ADC 转换函数；

- 停止 ADC 转换函数；
- 获取 ADC 的采样率函数；
- 设置 ADC 的采样率函数；
- 获取 ADC 转换精度函数；
- 获取 ADC 参考电压函数。

因此，在驱动层编写函数时，应该向标准接口层提供这六个函数。

这些函数的有一个共同点，就是第一个参数都是 `void` 型的指针，也就是说，驱动层提供的函数，标准接口层使用时传递的第一个参数并无确定类型，标准接口层不关心第一个参数的具体类型，使用时只负责传递就可以了。

那第一个参数从哪里来呢？驱动层怎么把这六个函数提供给标准接口层呢？看下一个结构体的定义。

5. `typedef struct am_adc_serv` 结构体

该结构体定义了标准的 ADC 服务，包含了上文提到的驱动函数结构体的指针和一个 `void` 型的指针。

这个标准的 ADC 服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

那么，如何赋值呢？

- `p_funcs` 成员赋值

这是一个指向 `struct am_adc_drv_funcs` 结构体的指针，我们只需要在驱动层定义一个 `struct am_adc_drv_funcs` 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那六个函数地址。如程序清单 9.2 所示。然后，将 `__g_adc_drvfuncs` 的地址赋值给 `p_funcs` 成员即可。

程序清单 9.2 `struct am_adc_drv_funcs` 结构体变量

```
/**
 * \brief ADC 服务函数
 */
static const struct am_adc_drv_funcs __g_adc_drvfuncs = {
    __fn_adc_start,
    __fn_adc_stop,
    __fn_adc_rate_get,
    __fn_adc_rate_set,
    __fn_bits_get,
    __fn_vref_get
};
```

- `p_drv` 成员赋值

这个变量是 `void` 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动层决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

6. `am_adc_handle_t` 的定义

这个是一个标准的操作句柄，本质上就是 `struct am_adc_serv` 指针类型，驱动层的初始

化函数(am_zmf2x6_adc_init)需要返回该类型的一个 handle 给用户。这样,用户获取到 handle,就可以在需要使用标准接口层函数时,将该 handle 传递给标准接口层了。标准接口就可以通过该 handle (struct am_adc_serv 结构体指针)访问到驱动层提供的函数了。

9.1.2 了解标准接口函数

在 am_adc.h 文件相关的结构体、类型定义后,就是具体函数的实现,如程序清单 9.3 所示。

程序清单 9.3 标准函数实现

```
/**
 * \brief 启动 ADC 转换
 *
 * \param[in] handle      : ADC 标准服务操作句柄
 * \param[in] chan        : 待转换的通道
 * \param[in] p_desc      : 指向缓冲区描述的指针
 * \param[in] desc_num    : 缓冲区描述的个数
 * \param[in] count       : 序列的转换次数 (一个序列表示 desc_num 指定个数的缓
 *                          冲区依次填满), 为 0 表示需要持续不断的转换。
 * \param[in] flags       : 可用标志见"ADC 启动转换的标志"
 * \param[in] pfn_callback : 序列完成回调函数
 * \param[in] p_arg       : 序列完成回调函数用户自定义参数
 *
 * \retval  AM_OK        : 启动转换成功
 * \retval -AM_ENXIO    : 指定的通道不存在
 * \retval -AM_EINVAL   : 无效参数
 *
 */
am_static_inline
int am_adc_start (am_adc_handle_t      handle,
                  int                    chan,
                  am_adc_buf_desc_t     *p_desc,
                  uint32_t               desc_num,
                  uint32_t               count,
                  uint32_t               flags,
                  am_adc_seq_cb_t        pfn_callback,
                  void                   *p_arg)
{
    return handle->p_funcs->pfn_adc_start(handle->p_drv,
                                          chan,
                                          p_desc,
                                          desc_num,
                                          count,
                                          flags,
                                          pfn_callback,
                                          p_arg);
}
```

```

}

/**
 * \brief 停止 ADC 转换
 *
 * \param[in] handle : ADC 标准服务操作句柄
 * \param[in] chan   : 停止转换的通道
 *
 * \retval  AM_OK      : 停止转换成功
 * \retval -AM_ENXIO   : 指定的通道不存在
 * \retval -AM_EINVAL  : 无效参数
 */
am_static_inline
int am_adc_stop (am_adc_handle_t handle, int chan)
{
    return handle->p_funcs->pfn_adc_stop(handle->p_drv, chan);
}

/**
 * \brief 获取 ADC 转换精度
 *
 * \param[in] handle      : ADC 标准服务操作句柄
 * \param[in] chan        : 停止转换的通道
 *
 * \retval    >0          : ADC 转换位数
 * \retval -AM_EINVAL    : 无效参数
 */
am_static_inline
uint32_t am_adc_bits_get (am_adc_handle_t handle, int chan)
{
    return handle->p_funcs->pfn_bits_get(handle->p_drv, chan);
}

/**
 * \brief 获取 ADC 参考电压，单位：mV
 *
 * \param[in] handle : ADC 标准服务操作句柄
 * \param[in] chan   : 停止转换的通道
 *
 * \retval    >0      : ADC 参考电压
 * \retval -AM_EINVAL : 无效参数
 */
am_static_inline
int am_adc_vref_get (am_adc_handle_t handle, int chan)

```

```

{
    return handle->p_funcs->pfn_vref_get(handle->p_drv, chan);
}

/**
 * \brief 获取 ADC 的采样率，单位 Hz
 *
 * \param[in] handle : ADC 标准服务操作句柄
 * \param[in] chan    : 停止转换的通道
 * \param[out] p_rate : 获取采样率
 *
 * \retval AM_OK      : 停止转换成功
 * \retval -AM_EINVAL : 无效参数
 */
am_static_inline
int am_adc_rate_get (am_adc_handle_t handle, int chan, uint32_t *p_rate)
{
    return handle->p_funcs->pfn_rate_get(handle->p_drv, chan, p_rate);
}

/**
 * \brief 设置 ADC 的采样率，单位 Hz
 *
 * \param[in] handle : ADC 标准服务操作句柄
 * \param[in] chan    : 停止转换的通道
 * \param[out] rate    : 设置采样率
 *
 * \retval AM_OK      : 停止转换成功
 * \retval -AM_EINVAL : 无效参数
 */
am_static_inline
int am_adc_rate_set (am_adc_handle_t handle, int chan, uint32_t rate)
{
    return handle->p_funcs->pfn_rate_set(handle->p_drv, chan, rate);
}

```

六个函数全都是用内联函数就搞定了，因为它本身就没做什么事情，只是简单的调用了一下驱动层提供的函数而已。可以看到，在调用驱动层提供的函数时，就是将 struct am_adc_serv 结构体中的 p_drv 成员作为第一个参数传递。

值得一提的是在 am_adc.h 文件最下面还有一个函数 am_adc_read 如程序清单 9.4 所示。

程序清单 9.4 am_adc_read 函数声明

```

/**
 * \brief 读取指定通道的 ADC 转换值（读取结束后才返回）
 *

```

```

* \param[in] handle    : ADC 标准服务操作句柄
* \param[in] chan      : ADC 通道号
* \param[in] p_val     : 转换结果存放的缓冲区，数据右对齐
* \param[in] length    : 缓冲区的长度
*
* \retval  AM_OK       : 操作成功
* \retval  -AM_ENXIO   : ADC 通道号不存在
* \retval  -AM_EINVAL  : 无效参数
*/
int am_adc_read(am_adc_handle_t handle,
                int chan,
                void *p_val,
                uint32_t length);

```

其实体实际是在 am_adc.c 文件内，相关路径 ametal\components\service\source\am_adc.c。该文件内只包含两个函数，如程序清单 9.5 所示。

程序清单 9.5 am_adc.c 文件内容

```

volatile static int g_adc_satus = 0;

/* adc 转换回调函数 */
static void adc_convert_cb (void *p_arg, int state)
{
    am_wait_t *p_wait = (am_wait_t *)p_arg;
    g_adc_satus = state;
    am_wait_done(p_wait);
}

/*****

int am_adc_read (am_adc_handle_t handle,
                 int chan,
                 void *p_val,
                 uint32_t length)
{
    am_adc_buf_desc_t desc;
    am_wait_t wait;

    am_wait_init(&wait);
    am_adc_mkbufdesc(&desc, p_val, length, adc_convert_cb, (void *) &wait);
    am_adc_start(handle, chan, &desc, 1, 1, AM_ADC_DATA_ALIGN_RIGHT, NULL, NULL);
    am_wait_on(&wait);
    return g_adc_satus;
}

```

可以发现它其实是标准函数的一个应用，简化了获取 ADC 值的步骤。由于它的普适性

我称它为标准服务。通过该函数 (`am_adc_read`) 用户能够方便快捷的完成 A/D 值获取任务。

到此我们就分析完了 ADC 外设标准层文件，清楚了驱动层需要为标准接口层提供哪些内容。

9.2 HW 层开发

通过上一节的内容，我们已经知道了标准层实现的函数的功能，以及需要驱动层提供的函数，但驱动层并不是凌空搭建的，驱动层介乎与 HW 层和标准接口层之间，简单来说就是调用 HW 层的函数来提供给驱动层，所以在写驱动层前，我们先来完成 HW 层。

现在我们打开 ZMF2x6 芯片用户手册，先了解一下该芯片 ADC 外设，下面对照手册完成 ADC 外设的 HW 层的开发。

9.2.1 添加寄存器列表

HW 层就是与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读该外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMF2x6 用户手册的 ADC 章节，可以看到，ADC 章节部分，共有 12 个寄存器，简单描述如下：

- `ADC_ADDDATA`（数据寄存器）：涉及到当前通道转换数据的相关信息，所有位都为只读。包括数据有效位、数据覆盖标志位（读 `ADC_ADDDATA` 后硬件清零），以及此时数据结果的通道对应位，12 位 A/D 转换结果位。
- `ADC_ADCFG`（配置寄存器）：包含 ADC 预分频、通道采样时间、转换数据分辨率的设置，内部参考电压、温度传感器控制，窗口比较器及 ADC 转换的使能控制；
- `ADC_ADCCR`（控制寄存器）：包含边沿触发选择，外部触发源选择、外部触发延时采样配置，扫描顺序、数据对齐方式、转换模式、转换开始控制，DMA 使能、窗口比较器中断、A/D 转换结束中断控制；
- `ADC_ADCHS`（通道选择寄存器）：涉及到 A/D 通道的使能和禁能，包括通道 0~9 以及内部参考电压和温度传感器。
- `ADC_ADCMPR`（窗口比较寄存器）：涉及到窗口比较数值上下限设置。
- `ADC_ADSTA`（状态寄存器）：涉及到通道 0~9 数据覆盖标志、通道有效标志，当前转换通道标志、A/D 转换忙标志、比较标志、转换结束标志；
- `ADC_ADDR`（数据寄存器）：涉及到各通道转换数据的相关信息，所有位都为只读。包括数据有效位、数据覆盖标志位（读 `ADC_ADDDATA` 后硬件清零），12 位 A/D 转换结果位；
- `ADC_ADSTA_EXT`（扩展状态寄存器）：通道 14~15 数据覆盖标志位、通道有效标志位。
- `ADC_CHANY0`（任意通道选择寄存器 0）：涉及到通道 0~7 的通道选择。
- `ADC_CHANY1`（任意通道选择寄存器 1）：涉及到通道 8~15 的通道选择。
- `ADC_ANY_CFG`（任意通道配置寄存器 1）：任意通道配置数配置。
- `ADC_ANY_CR`（任意通道控制寄存器）：任意通道配置模式使能位。

到此 ADC 的各个寄存器及其功能基本阅读完毕，我们也大致了解了各个寄存器功能以及要配置的各个功能，其他寄存器位该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，如程序清单 9.6 所示。

程序清单 9.6 ADC 寄存器结构体定义

```

/**
 * \brief ADC - Register Layout Typedef
 */
typedef struct amhw_zmf2x6_adc {
    __IO uint32_t addata;      /**< \brief A/D 数据寄存器 */
    __IO uint32_t adcfg;      /**< \brief A/D 配置寄存器 */
    __IO uint32_t adcr;       /**< \brief A/D 控制寄存器 */
    __IO uint32_t adchs;      /**< \brief A/D 通道选择寄存器 */
    __IO uint32_t adcmp;      /**< \brief A/D 窗口比较寄存器 */
    __IO uint32_t adsta;      /**< \brief A/D 状态寄存器 */
    __IO uint32_t addr[16];   /**< \brief A/D 数据寄存器 */
    __IO uint32_t adsta_ext;   /**< \brief A/D 扩展状态寄存器 */
    __IO uint32_t chany[2];   /**< \brief A/D 任意通道选择寄存器 */
    __IO uint32_t any_cfg;    /**< \brief A/D 任意通道配置寄存器 */
    __IO uint32_t any_cr;     /**< \brief A/D 任意通道控制寄存器 */
} amhw_zmf2x6_adc_t;

```

这个小节我们定义了 ADC 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

9.2.2 编写寄存器的相关函数

1. ADC_ADDDATA（数据寄存器）

通过手册，我们了解到该寄存器有含义的寄存器共有 4 个，如表 9.1 所示。

表 9.1 ADC_ADDDATA 寄存器分析

位	标识符	含义
21	VALID	相应模拟通道转换完成后，该位置位，读 ADDATA 寄存器后，该位由硬件清除。
20	OVERRUN	新的转换结果装载至寄存器之前，若 DATA[11: 0]的数据没有被读取，OVERRUN 将置 1。读 ADDATA 寄存器硬件清除。
16:19	CHANNELSEL	该 4 位显示当前数据（DATA）所对应的通道
0:15	DATA	12 位 A/D 转换结果

由此我们确定编写三个函数：获取有效标志位状态函数、获取数据覆盖标志位函数、获取 ADC 通道转换数据函数。

程序清单 9.7 为获取数据有效标志位函数。读 ADDATA 寄存器后，该位由硬件清除。

程序清单 9.7 获取有效标志位状态函数

```

/**
 * \brief 获取有效标志位

```

```

*
* \param[in] p_hw_adc : 指向 ADC 结构体的指针
*
* \retval AM_TRUE : 数据有效
* \retval AM_FALSE: 数据无效
*/
am_static_inline
am_bool_t amhw_zmf2x6_adc_data_valid_check (amhw_zmf2x6_adc_t *p_hw_adc)
{
    return (((p_hw_adc->addata >> 21) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
}

```

程序清单 9.8 为获取数据覆盖标志位函数，新的转换结果装载至寄存器之前，若 ADDATA 的数据没有被读取，将置返回 1。读 ADDATA 寄存器后，该位由硬件清除。

程序清单 9.8 获取数据覆盖标志位函数

```

/**
* \brief 获取数据覆盖标志位
*
* \param[in] p_hw_adc : 指向 ADC 结构体的指针
*
* \retval AM_TRUE : 数据被覆盖
* \retval AM_FALSE: 数据最近一次转换结果
*/
am_static_inline
am_bool_t amhw_zmf2x6_adc_data_overrun_check (amhw_zmf2x6_adc_t *p_hw_adc)
{
    return (((p_hw_adc->addata >> 20) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
}

```

程序清单 9.9 为获取 A/D 相应通道转换结果函数，结果数据共 12bit，存储于输出参数 p_data 中。返回值为对应的通道值，相关通道定义如程序清单 9.10 所示

程序清单 9.9 获取 ADC 转换数据函数

```

/**
* \brief 获取 ADC 转换数据
*
* \param[in] p_hw_adc : 指向 ADC 结构体的指针
* \param[out] data : 存储转换数据的指针
*
* \return ADC 通道
*/
am_static_inline
uint8_t amhw_zmf2x6_adc_data_get (amhw_zmf2x6_adc_t *p_hw_adc,
                                   uint16_t *p_data)
{

```



```

*p_data = (uint16_t)(p_hw_adc->addata & 0xffffu);

return (uint8_t)((p_hw_adc->addata >> 16) & 0xfu);
}

```

程序清单 9.10 ADC 通道枚举定义

```

/**
 * \brief ADC 通道
 */
typedef enum amhw_zmf2x6_adc_channel {
    AMHW_ZMF2X6_ADC_CHAN_0 = 0,          /**< \brief 通道 0 */
    AMHW_ZMF2X6_ADC_CHAN_1,             /**< \brief 通道 1 */
    AMHW_ZMF2X6_ADC_CHAN_2,             /**< \brief 通道 2 */
    AMHW_ZMF2X6_ADC_CHAN_3,             /**< \brief 通道 3 */
    AMHW_ZMF2X6_ADC_CHAN_4,             /**< \brief 通道 4 */
    AMHW_ZMF2X6_ADC_CHAN_5,             /**< \brief 通道 5 */
    AMHW_ZMF2X6_ADC_CHAN_6,             /**< \brief 通道 6 */
    AMHW_ZMF2X6_ADC_CHAN_7,             /**< \brief 通道 7 */
    AMHW_ZMF2X6_ADC_CHAN_8,             /**< \brief 通道 8 */
    AMHW_ZMF2X6_ADC_CHAN_9,             /**< \brief 通道 9 */
    AMHW_ZMF2X6_ADC_TEMP_SENSOR = 14,   /**< \brief 内部温度传感器 */
    AMHW_ZMF2X6_ADC_VOL_SENSOR = 15,     /**< \brief 内部参考电压 */
} amhw_zmf2x6_adc_channel_t;

```

2. ADC_ADCCFG（配置寄存器）

通过手册，我们了解到该寄存器有含义的寄存器共有 8 个，如表 5.1 所示。

表 9.2 ADC_ADCCFG 寄存器分析

位	标识符	含义
14	ADCPRE	作为 ADCPRE[3:0] 的最低位
10:13	SAMCTL	这些位用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。
7:9	RSLTCTL	选择 ADCx 转换数据分辨率
4:6	ADCPRE	ADC 预分频设置
3	VSEN	内部参考电压使能控制位
2	TSEN	温度传感器使能控制位
1	ADWEN	A/D 窗口比较器使能控制位
0	ADEN	A/D 转换使能控制位

由此我们确定编写六个函数：ADC 通道采样时间配置及获取函数、ADC 分辨率设置及获取函数、ADC 预分频设置及获取函数以及 ADC_CFG 功能使能禁能函数。

程序清单 9.11 为 ADC 通道采样时间选择函数，采样时间通过枚举类型 amhw_zmf2x6_adc_sample_time_t 进行配置，如程序清单 9.12 所示。

程序清单 9.11 ADC 通道采样时间选择函数

```

/**
 * \brief ADC 通道采样时间选择
 *
 * \param[in] p_hw_adc      : 指向 ADC 结构体的指针
 * \param[in] sample_time  : 采样时间
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_chan_sample_time (amhw_zmf2x6_adc_t      *p_hw_adc,
                                       amhw_zmf2x6_adc_sample_time_t  sample_time)
{
    p_hw_adc->adcfg = (p_hw_adc->adcfg & ~(0xfu << 10))) | ((sample_time & 0xfu) << 10);
}

```

程序清单 9.12 ADC 采样时间设置值枚举

```

/**
 * \brief ADC 采样时间
 */
typedef enum amhw_zmf2x6_adc_sample_time {
    AMHW_ZMF2X6_ADC_CHAN_ST1_5 = 0,
    AMHW_ZMF2X6_ADC_CHAN_ST7_5,
    AMHW_ZMF2X6_ADC_CHAN_ST13_5,
    AMHW_ZMF2X6_ADC_CHAN_ST28_5,
    AMHW_ZMF2X6_ADC_CHAN_ST41_5,
    AMHW_ZMF2X6_ADC_CHAN_ST55_5,
    AMHW_ZMF2X6_ADC_CHAN_ST71_5,
    AMHW_ZMF2X6_ADC_CHAN_ST239_5,
    AMHW_ZMF2X6_ADC_CHAN_ST2_5,
    AMHW_ZMF2X6_ADC_CHAN_ST3_5,
    AMHW_ZMF2X6_ADC_CHAN_ST4_5,
    AMHW_ZMF2X6_ADC_CHAN_ST5_5,
    AMHW_ZMF2X6_ADC_CHAN_ST6_5,
} amhw_zmf2x6_adc_sample_time_t;

```

程序清单 9.13 为获取采样通道时间，其返回值为 `amhw_zmf2x6_adc_sample_time_t` 枚举类型。

程序清单 9.13 获取 ADC 通道采样时间函数

```

/**
 * \brief 获取 ADC 通道采样时间
 *
 * \param[in] p_hw_adc      : 指向 ADC 结构体的指针
 *

```

```

* \return none
*/
am_static_inline
amhw_zmf2x6_adc_sample_time_t
amhw_zmf2x6_adc_chan_sample_time_get (amhw_zmf2x6_adc_t *p_hw_adc)
{
    return (amhw_zmf2x6_adc_sample_time_t)((p_hw_adc->adcfg >> 10) & 0xfu);
}

```

程序清单 9.14 为 ADC 分辨率设置及获取函数，用于控制数据转换精度。其支持的精度通过宏的方式定义出来，之后可直接作为相关参数。

程序清单 9.14 ADC 分辨率函数

```

/**
 * \brief ADC 分辨率
 * \@{
 */

#define AMHW_ZMF2X6_ADC_DATA_VALID_12BIT 12u /**< \brief 12 bit */
#define AMHW_ZMF2X6_ADC_DATA_VALID_11BIT 11u /**< \brief 11 bit */
#define AMHW_ZMF2X6_ADC_DATA_VALID_10BIT 10u /**< \brief 10 bit */
#define AMHW_ZMF2X6_ADC_DATA_VALID_9BIT 9u /**< \brief 9 bit */
#define AMHW_ZMF2X6_ADC_DATA_VALID_8BIT 8u /**< \brief 8 bit */

/** @} */

/**
 * \brief ADC 分辨率设置
 *
 * \param[in] p_hw_adc : 指向 ADC 结构体的指针
 * \param[in] resolution : 分辨率
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_data_resolution (amhw_zmf2x6_adc_t *p_hw_adc,
                                     uint8_t resolution)
{
    p_hw_adc->adcfg = (p_hw_adc->adcfg & ~(0x7u << 7)) | (((12u - resolution) & 0x7u) << 7);
}

/**
 * \brief 获取 ADC 分辨率
 *
 * \param[in] p_hw_adc : 指向 ADC 结构体的指针
 */

```

```

    * \return bits
    */
am_static_inline
uint32_t amhw_zmf2x6_adc_data_resolution_get (amhw_zmf2x6_adc_t *p_hw_adc)
{
    return (uint32_t)((p_hw_adc->adcfg >> 7) & 0x7u);
}

```

程序清单 9.15 为 ADC 的预分频设置与获取函数。

程序清单 9.15 ADC 预分频函数

```

/**
 * \brief ADC 预分频设置
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] pre_val     : ADC 分频值
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_prescaler_val_set (amhw_zmf2x6_adc_t *p_hw_adc,
                                         uint32_t          pre_val)
{
    if ((pre_val % 2) == 0) {

        /* 分频值为偶数 */
        p_hw_adc->adcfg = (p_hw_adc->adcfg & ~(0x7u << 4)) | (((pre_val >> 1) - 1) & 0x7u) << 4;
        p_hw_adc->adcfg &= ~(0x1u << 14);
    } else {

        /* 分频值为奇数 */
        p_hw_adc->adcfg = (p_hw_adc->adcfg & ~(0x7u << 4)) | (((pre_val - 3) >> 1) & 0x7u) << 4;
        p_hw_adc->adcfg |= (0x1u << 14);
    }
}

/**
 * \brief 获取 ADC 预分频
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 *
 * \return ADC 分频值
 */
am_static_inline
uint8_t amhw_zmf2x6_adc_prescaler_val_get (amhw_zmf2x6_adc_t *p_hw_adc)

```

```

{
    if ((p_hw_adc->adcfg & (0x1u << 14)) == 0) {

        /* 分频值为偶数 */
        return (((p_hw_adc->adcfg >> 4) & 0x7u) + 1) << 1;
    } else {

        /* 分频值为奇数 */
        return (((p_hw_adc->adcfg >> 4) & 0x7u) << 1) + 3;
    }
}

```

程序清单 9.16 为 ADC_CFG 功能配置函数, 包含四个宏作为其的参数以控制相应功能。实际使用时也可以用于直接操作寄存器。

程序清单 9.16 ADC_CFG 功能配置函数

```

/**
 * \brief ADC_CFG 功能使能掩码
 * \@{
 */
#define AMHW_ZMF2X6_ADC_VOL_SENSOR_EN_MASK (1U << 3) /**< \brief 内部参考电压使能 */
#define AMHW_ZMF2X6_ADC_TEMP_SENSOR_EN_MASK (1U << 2) /**< \brief 温度传感器使能 */
#define AMHW_ZMF2X6_ADC_WINDOWS_EN_MASK (1U << 1) /**< \brief A/D 窗口比较器使能 */
#define AMHW_ZMF2X6_ADC_MODULE_EN_MASK (1U << 0) /**< \brief A/D 转换使能 */
/** @} */

/**
 * \brief ADC_CFG 功能使能
 *
 * \param[in] p_hw_adc : 指向 ADC 结构体的指针
 * \param[in] flag : ADC_CFG 功能使能掩码
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_cfg_reg_set (amhw_zmf2x6_adc_t *p_hw_adc,
                                   uint32_t flag)
{
    p_hw_adc->adcfg |= flag;
}

/**
 * \brief ADC_CFG 功能禁能
 */

```

```

* \param[in] p_hw_adc    : 指向 ADC 结构体的指针
* \param[in] flag        : ADC_CFG 功能使能掩码
*
* \return none
*/
am_static_inline
void amhw_zmf2x6_adc_cfg_reg_clr(amhw_zmf2x6_adc_t *p_hw_adc,
                                uint32_t          flag)
{
    p_hw_adc->adcfg &= ~flag;
}

```

3. ADC_ADCCR（控制寄存器）

通过手册，我们了解到该寄存器有含义的寄存器共有 8 个，如表 5.1 所示。

表 9.3 ADC_ADCCR 寄存器分析

位	标识符	含义
24:25	TRG_EDGE	触发边沿选择
19:21	TRGSHIFT	在触发信号产生后，延时 N 个 PCLK2 的时钟周期再开始采样。
17:18	TRGSEL	外部触发源选择与 Bit[6:4] 合用
16	SCANDIR	在单周期扫描或者连续扫描方式时，设置扫描通道的顺序
12:15	CMPCH	窗口比较通道选择
11	ALIGN	数据对齐方式选择
9:10	ADMD	A/D 转换模式
8	ADST	A/D 转换开始，在单次模式或者单周期模式下，转换完成后，ADST 将被硬件自动清除。
4:6	TRGSEL	选择外部触发源
3	DMAEN	DMA 使能
2	TRGEN	外部硬件触发源使能
1	ADWIE	A/D 窗口比较器中断使能
0	ADIE	A/D 中断使能

共 11 个函数：ADC 触发边沿选择函数、ADC 外部触发延时采样设置函数、ADC 扫描通道顺序设置函数、ADC 窗口比较通道选择函数、ADC 数据对齐设置函数、ADC 转换模式设置函数、ADC 转换状态设置及获取函数、ADC 外部触发源选择函数、ADC_CR 寄存器功能使能及禁止函数。

程序清单 9.17 为 ADC 触发边沿选择函数，即控制 ADC 触发方式。

程序清单 9.17 触发边沿选择函数

```

/**
* \brief ADC 触发边沿
*/
typedef enum amhw_zmf2x6_adc_trg_edge {
    AMHW_ZMF2X6_ADC_TRG_EDGE_BOTH = 0, /* 双沿触发 */
}

```

```

    AMHW_ZMF2X6_ADC_TRG_EDGE_FALLING, /* 下沿触发 */
    AMHW_ZMF2X6_ADC_TRG_EDGE_RISING, /* 上沿触发 */
    AMHW_ZMF2X6_ADC_TRG_EDGE_SHIELD, /* 屏蔽触发 */
} amhw_zmf2x6_adc_trg_edge_t;

/**
 * \brief ADC 触发边沿选择
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] trg_edge    : 触发边沿
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_trg_edge_set (amhw_zmf2x6_adc_t *p_hw_adc,
                                   amhw_zmf2x6_adc_trg_edge_t trg_edge)
{
    p_hw_adc->adcr = (p_hw_adc->adcr & ~(0x3u << 24)) | ((trg_edge & 0x3u) << 24);
}

```

程序清单 9.18 为外部触发延时采样周期设置函数,在触发信号产生后,延时 N 个 PCLK2 的时钟周期再开始采样。

程序清单 9.18 外部触发延时采样设置

```

/**
 * \brief ADC 外部触发延时采样
 */
typedef enum amhw_zmf2x6_adc_trgshift {
    AMHW_ZMF2X6_ADC_TRGSHIFT_NO_DELAY = 0, /* 不延时 */
    AMHW_ZMF2X6_ADC_TRGSHIFT_4_PERIOD, /* 4 个周期 */
    AMHW_ZMF2X6_ADC_TRGSHIFT_16_PERIOD, /* 16 个周期 */
    AMHW_ZMF2X6_ADC_TRGSHIFT_32_PERIOD, /* 32 个周期 */
    AMHW_ZMF2X6_ADC_TRGSHIFT_64_PERIOD, /* 64 个周期 */
    AMHW_ZMF2X6_ADC_TRGSHIFT_128_PERIOD, /* 128 个周期 */
    AMHW_ZMF2X6_ADC_TRGSHIFT_256_PERIOD, /* 256 个周期 */
    AMHW_ZMF2X6_ADC_TRGSHIFT_512_PERIOD, /* 512 个周期 */
} amhw_zmf2x6_adc_trgshift_t;

/**
 * \brief ADC 外部触发延时采样设置
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] trgshift    : 外部触发延时采样
 *
 * \return none
 */

```

```

*/
am_static_inline
void amhw_zmf2x6_adc_trgshift_set (amhw_zmf2x6_adc_t      *p_hw_adc,
                                   amhw_zmf2x6_adc_trgshift_t  trgshift)
{
    p_hw_adc->adcr = (p_hw_adc->adcr & ~(0x7u << 19)) | ((trgshift & 0x7u) << 19);
}

```

程序清单 9.19 为扫描通道顺序设置函数，控制通道扫描顺序，其参数为枚举类型 `amhw_zmf2x6_adc_scandir_t`。

程序清单 9.19 扫描通道顺序设置函数

```

/**
 * \brief ADC 扫描通道顺序
 */
typedef enum amhw_zmf2x6_adc_scandir {
    AMHW_ZMF2X6_ADC_SCANDIR_FROM_LOW_TO_HIGH = 0, /* ADC 通道选择寄存器按从低到
高的顺序扫描 */
    AMHW_ZMF2X6_ADC_SCANDIR_FROM_HIGH_TO_LOW, /* ADC 通道选择寄存器按从高
到低的顺序扫描 */
} amhw_zmf2x6_adc_scandir_t;

/**
 * \brief ADC 扫描通道顺序设置
 *
 * \param[in] p_hw_adc      : 指向 ADC 结构体的指针
 * \param[in] scandir       : 扫描通道顺序
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_scandir_set (amhw_zmf2x6_adc_t      *p_hw_adc,
                                   amhw_zmf2x6_adc_scandir_t  scandir)
{
    p_hw_adc->adcr = (p_hw_adc->adcr & ~(0x1u << 16)) | ((scandir & 0x1u) << 16);
}

```

程序清单 9.20 为窗口比较通道选择函数，其通道选择见程序清单 9.10 的 `amhw_zmf2x6_adc_channel_t` 枚举类型。

程序清单 9.20 窗口比较通道选择函数

```

/**
 * \brief ADC 窗口比较通道选择
 *
 * \param[in] p_hw_adc      : 指向 ADC 结构体的指针
 * \param[in] chan          : ADC 通道

```



```

*
* \return none
*/
am_static_inline
void amhw_zmf2x6_adc_win_chan_sel (amhw_zmf2x6_adc_t *p_hw_adc,
                                   uint32_t          chan)
{
    p_hw_adc->adcr = (p_hw_adc->adcr & ~(0xfu << 12)) | ((chan & 0xfu) << 12);
}

```

程序清单 9.21 为数据对齐设置函数，用于确定数据的左对齐还是右对齐。

程序清单 9.21 数据对齐设置函数

```

/**
 * \brief ADC 数据对齐
 * \@{
 */
#define AMHW_ZMF2X6_ADC_DATA_ALIGN_RIGHT (0U) /**< \brief 右对齐 */
#define AMHW_ZMF2X6_ADC_DATA_ALIGN_LEFT  (1U) /**< \brief 左对齐 */

/** @} */
/**
 * \brief ADC 数据对齐设置
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] flag        : 数据对齐
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_align_way_sel (amhw_zmf2x6_adc_t *p_hw_adc,
                                    uint32_t          flag)
{
    p_hw_adc->adcr = (p_hw_adc->adcr & ~(0x1u << 11)) | ((flag & 0x1u) << 11);
}

```

程序清单 9.22 为 ADC 转换模式设置函数，该 ADC 有三种转换模式：单次转换模式、单周期扫描模式、连续扫描模式，对种模式定义相应的宏，作为函数的参数。

程序清单 9.22 ADC 转换模式设置函数

```

/**
 * \brief ADC 转换模式
 * \@{
 */
#define AMHW_ZMF2X6_ADC_ONE_TIME_SCAN      (0U)    /**< \brief 单次转换 */
#define AMHW_ZMF2X6_ADC_SINGLE_PERIOD_SCAN (1U)    /**< \brief 单周期扫描 */

```

```

#define AMHW_ZMF2X6_ADC_CONTINUE_SCAN      (2U)    /**< \brief 连续扫描 */

/** @*/
/**
 * \brief ADC 转换模式设置
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] flag        : 转换模式
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_data_scan_set (amhw_zmf2x6_adc_t *p_hw_adc,
                                     uint32_t          flag)
{
    p_hw_adc->adcr = (p_hw_adc->adcr & ~(0x3u << 9)) | ((flag & 0x3u) << 9);
}

```

程序清单 9.23 为 ADC 转换状态设置及获取函数。在单次模式或者单周期模式下，转换完成后，ADST 将被硬件自动清除。在连续扫描模式下，A/D 转换将一直进行，直到软件写‘0’到该位或系统复位。

程序清单 9.23 ADC 转换状态相关函数

```

/**
 * \brief ADC 转换状态
 * \@{
 */
#define AMHW_ZMF2X6_ADC_DATA_TRANSFER_START (1U)    /**< \brief 转换开始 */
#define AMHW_ZMF2X6_ADC_DATA_TRANSFER_END   (0U)    /**< \brief 转换结束 */

/** @*/
/**
 * \brief ADC 转换状态设置
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] flag        : 转换状态设置
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_data_transfer_set (amhw_zmf2x6_adc_t *p_hw_adc,
                                         uint32_t          flag)
{
    p_hw_adc->adcr = (p_hw_adc->adcr & ~(0x1u << 8)) | ((flag & 0x1u) << 8);
}

```

```

}

/**
 * \brief 获取 ADC 转换状态
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 *
 * \retval    AM_TRUE : 转换开始
 *            AM_FALSE: 转换结束或进入空闲状态
 */
am_static_inline
am_bool_t amhw_zmf2x6_adc_data_transfer_start_check (amhw_zmf2x6_adc_t *p_hw_adc)
{
    return (((p_hw_adc->adcr >> 8) & 0x1u) ? AM_TRUE : AM_FALSE);
}

```

程序清单 9.24 为 ADC 外部触发源选择函数，用于选择 ADC 的触发源，对每个触发源都进行了相关的宏定义，使用时直接作为参数传入即可。

程序清单 9.24 ADC 外部触发源选择函数

```

/**
 * \brief ADC 外部触发源选择
 * \@{
 */

#define AMHW_ZMF2X6_ADC_EXTRG_TIM1_CC1      (0U)    /**< \brief TIM1_CC1 */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM1_CC2      (1U)    /**< \brief TIM1_CC2 */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM1_CC3      (2U)    /**< \brief TIM1_CC3 */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM2_CC2      (3U)    /**< \brief TIM2_CC2 */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM3_TRGO     (4U)    /**< \brief TIM3_TRGO */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM1_CC4_CC5  (5U)    /**< \brief TIM1_CC4 和 TIM1_CC5 */

#define AMHW_ZMF2X6_ADC_EXTRG_TIM3_CC1      (6U)    /**< \brief TIM3_CC1 */
#define AMHW_ZMF2X6_ADC_EXTRG_LINE_11       (7U)    /**< \brief LINE_11 */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM1_TRGO     (8U)    /**< \brief TIM1_TRGO */
#define AMHW_ZMF2X6_ADC_EXTRG_LINE_4        (9U)    /**< \brief LINE_4 */
#define AMHW_ZMF2X6_ADC_EXTRG_LINE_5        (10U)   /**< \brief LINE_5 */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM2_CC1      (11U)   /**< \brief TIM2_CC1 */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM3_CC4      (12U)   /**< \brief TIM3_CC4 */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM2_TRGO     (13U)   /**< \brief TIM2_TRGO */
#define AMHW_ZMF2X6_ADC_EXTRG_LINE_15       (15U)   /**< \brief LINE_15 */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM1_CC4      (16U)   /**< \brief TIM1_CC4 */
#define AMHW_ZMF2X6_ADC_EXTRG_TIM1_CC5      (17U)   /**< \brief TIM1_CC5 */

/** @} */

```

```

/**
 * \brief ADC 外部触发源选择
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] flag        : 外部触发源选择
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_extrg_select (amhw_zmf2x6_adc_t *p_hw_adc,
                                   uint32_t          flag)
{
    p_hw_adc->adcr = (p_hw_adc->adcr & ~(0x7u << 4)) | ((flag & 0x7u) << 4);
    p_hw_adc->adcr = (p_hw_adc->adcr & ~(0x3u << 17)) | (((flag >> 3) & 0x3u) << 17);
}

```

程序清单 9.25 为 ADC_CR 寄存器使能禁能函数，通过该寄存器可以完成最后四个寄存器位的相关操作，如下定义了四个位的宏。同时该函数也可以直接用做寄存器操作，传入特定数据，不过一般不建议这样。

程序清单 9.25 ADC_CR 寄存器控制函数

```

/**
 * \brief ADC_CR 功能使能掩码
 * \@{
 */
#define AMHW_ZMF2X6_ADC_DMA_EN_MASK      (1U << 3) /**< \brief DMA 使能 */
#define AMHW_ZMF2X6_ADC_EXTRG_EN_MASK    (1U << 2) /**< \brief 外部硬件出发源使能 */
#define AMHW_ZMF2X6_ADC_INT_WIN_EN_MASK (1U << 1) /**< \brief 窗口比较器中断使能 */
#define AMHW_ZMF2X6_ADC_INT_EN_MASK      (1U << 0) /**< \brief ADC 中断使能 */
/** @} */

/**
 * \brief ADC_CR 功能使能
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] flag        : ADC_CR 功能使能掩码
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_ctrl_reg_set (amhw_zmf2x6_adc_t *p_hw_adc,
                                   uint32_t          flag)
{
    p_hw_adc->adcr |= flag;
}

```

```

/**
 * \brief ADC_CR 功能禁能
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] flag        : ADC_CR 功能使能掩码
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_ctrl_reg_clr (amhw_zmf2x6_adc_t *p_hw_adc,
                                   uint32_t          flag)
{
    p_hw_adc->adcr &= ~flag;
}

```

4. ADC_ADCHS（通道选择寄存器）

该寄存器主要用于选择通道的使能和禁能，相关函数如程序清单 9.26 所示，相关通道信息见程序清单 9.10 的 amhw_zmf2x6_adc_channel_t 枚举类型。

程序清单 9.26 通道控制函数

```

/**
 * \brief ADC 通道使能
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] chan        : ADC 通道
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_input_chan_enable (amhw_zmf2x6_adc_t *p_hw_adc,
                                         amhw_zmf2x6_adc_channel_t chan)
{
    if ((chan == 10) || (chan == 11)) {
        chan += 4;
    }
    p_hw_adc->adchs |= (1 << chan);
}

/**
 * \brief ADC 通道禁能
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] chan        : ADC 通道
 *

```

```

* \return none
*/
am_static_inline
void amhw_zmf2x6_adc_input_chan_disable (amhw_zmf2x6_adc_t      *p_hw_adc,
                                         amhw_zmf2x6_adc_channel_t  chan)
{
    p_hw_adc->adchs &= ~(1 << chan);
}

```

5. ADC_ADCMPR（窗口比较寄存器）

该寄存器主要完成两个功能，即比较数值上下限的设置，表描述了该寄存器信息。

表 9.4 ADC_ADCMPR 寄存器分析

位	标识符	含义
16:27	CMPHDATA	比较数值上限，该 12 位数值将和指定通道的转换结果相比较，当 AD 采集值超过上限后比较标志位置位。
0:11	CMPLDATA	比较数值下限，该 12 位数值将和指定通道的转换结果相比较，当 AD 采集值超过上限后比较标志位置位。

因此我们可以列出两个函数：窗口比较数值上限设置函数，窗口比较数值下限设置函数。

程序清单 9.27 为 ADC 比较上限值设置函数，用于设置 ADC 比较的上限值，当 AD 采集值超过上限后比较标志位置位，若使能了窗口比较中断会产生相应的中断服务函数。

程序清单 9.27 ADC 比较上限值设置函数

```

/**
 * \brief ADC 比较上限值
 *
 * \param[in] p_hw_adc    : 指向 ADC 结构体的指针
 * \param[in] value      : 0 ~ 4095
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_cmp_high_limit (amhw_zmf2x6_adc_t *p_hw_adc,
                                     uint16_t          value)
{
    p_hw_adc->adcmp = (p_hw_adc->adcmp & ~(0xffff << 16)) |
                      ((value & 0xffff) << 16);
}

```

程序清单 9.28 为 ADC 比较下限值设置函数，用于设置 ADC 比较的下限值，当 AD 采集值低于下限值后比较标志位置位，若使能了窗口比较中断会产生相应的中断服务函数。

程序清单 9.28 ADC 比较下限值设置函数

```

/**

```

```

* \brief ADC 比较下限值
*
* \param[in] p_hw_adc    : 指向 ADC 结构体的指针
* \param[in] value      : 0 ~ 4095
*
* \return none
*/
am_static_inline
void amhw_zmf2x6_adc_cmp_low_limit (amhw_zmf2x6_adc_t *p_hw_adc,
                                     uint16_t          value)
{
    p_hw_adc->adcmp = (p_hw_adc->adcmp & ~(0xffff << 0)) |
                      ((value & 0xffff) << 0);
}

```

6. ADC_ADSTA（状态寄存器）

该寄存器拥有多个有效位，均为状态标注，详见表 9.5。

表 9.5 ADC_ADSTA 寄存器描述

位	标识符	含义
20:29	OVERRUN	通道 0~9 的数据覆盖标志位
8:17	VALID	通道 0~9 的有效标志位。
4:7	CHANNEL	当前转换通道该 4 位在 BUSY = 1 时表示进行转换中的通道。BUSY = 0 时表示可进行下次转换的通道。
2	BUSY	A/D 转换器忙/空闲标志
1	ADWIF	比较标志位
0	ADIF	A/D 转换结束标志位

因此我们可以列出六个函数：通道 0~9 数据覆盖标志检测函数、获取通道有效标志函数，获取当前转换通道标志函数、获取 A/D 转换忙标志函数、获取比较标志函数、获取转换结束标志函数。除了上述函数编写方法还可以使用单个函数配多个宏的方式对状态位进行控制。

程序清单 9.29 为获取通道数据覆盖标志位函数，其参数 amhw_zmf2x6_adc_channel_t 见程序清单 9.10。

程序清单 9.29 获取通道数据覆盖标志位函数

```

/**
* \brief 获取通道数据覆盖标志位
*
* \param[in] p_hw_adc : 指向 ADC 结构体的指针
* \param[in] chan     : amhw_zmf2x6_adc_channel_t 枚举
*
* \retval AM_TRUE : 数据被覆盖
* \retval AM_FALSE: 通道数据最近一次转换结果
*/
am_static_inline

```

```

am_bool_t amhw_zmf2x6_adc_stat_chan_data_overrun_check (amhw_zmf2x6_adc_t      *p_hw_adc,
                                                         amhw_zmf2x6_adc_channel_t  chan)
{
    if (chan <= AMHW_ZMF2X6_ADC_CHAN_9) {

        /* 通道 0~9 */
        return (((p_hw_adc->adsta >> (20 + chan)) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
    } else {

        /* 通道 14~15 */
        return (((p_hw_adc->adsta_ext >> (chan - 8)) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
    }
}

```

程序清单 9.30 为获取通道数据有效标志位函数，若返回为 1 则代表该通道数据有效。

程序清单 9.30 获取通道数据有效标志位函数

```

/**
 * \brief 获取通道数据有效标志位
 *
 * \param[in] p_hw_adc : 指向 ADC 结构体的指针
 * \param[in] chan      : amhw_zmf2x6_adc_channel_t 枚举
 *
 * \retval AM_TRUE : 数据有效
 * \retval AM_FALSE: 数据无效
 */
am_static_inline
am_bool_t amhw_zmf2x6_adc_stat_chan_data_valid_check (amhw_zmf2x6_adc_t      *p_hw_adc,
                                                         amhw_zmf2x6_adc_channel_t  chan)
{
    if (chan <= AMHW_ZMF2X6_ADC_CHAN_9) {

        /* 通道 0~9 */
        return (((p_hw_adc->adsta >> (8 + chan)) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
    } else {

        /* 通道 14~15 */
        return (((p_hw_adc->adsta_ext >> (chan - 12)) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
    }
}

```

程序清单 9.31 为获取 ADC 当前转换通道函数，该 4 位在 BUSY=1 时表示进行转换中的通道。BUSY = 0 时表示可进行下次转换的通道。

程序清单 9.31 获取 ADC 当前转换通道函数

```

/**

```



```

* \brief 获取 ADC 当前转换通道
*
* \param[in] p_hw_adc : 指向 ADC 结构体的指针
*
* \return ADC 通道
*/
am_static_inline
uint8_t amhw_zmf2x6_adc_currnt_tran_chan_get (amhw_zmf2x6_adc_t *p_hw_adc)
{
    return (uint8_t)((p_hw_adc->adsta >> 4) & 0xfu);
}

```

程序清单 9.32 为获取忙/空闲状态函数。

程序清单 9.32 获取忙/空闲状态函数

```

/**
* \brief 获取忙/空闲状态
*
* \param[in] p_hw_adc : 指向 ADC 结构体的指针
*
* \retval AM_TRUE : 忙状态
* \retval AM_FALSE: 空闲状态
*/
am_static_inline
am_bool_t amhw_zmf2x6_adc_stat_busy_check (amhw_zmf2x6_adc_t *p_hw_adc)
{
    return (((p_hw_adc->adsta >> 2) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
}

```

程序清单 9.33 为窗口比较中断标志位获取和清除函数。

程序清单 9.33 窗口比较中断标志位相关函数

```

/**
* \brief 获取窗口比较中断标志位
*
* \param[in] p_hw_adc : 指向 ADC 结构体的指针
*
* \retval AM_TRUE : 窗口比较器发生中断
* \retval AM_FALSE: 窗口比较器未发生中断
*/
am_static_inline
am_bool_t amhw_zmf2x6_adc_int_win_cmp_check (amhw_zmf2x6_adc_t *p_hw_adc)
{
    return (((p_hw_adc->adsta >> 1) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
}

```

```

/**
 * \brief 清除窗口比较中断标志位
 *
 * \param[in] p_hw_adc : 指向 ADC 结构体的指针
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_int_win_cmp_clr (amhw_zmf2x6_adc_t *p_hw_adc)
{
    p_hw_adc->adsta |= (1ul << 1);
}

```

程序清单 9.34 为通道组转换结束标志位获取和清除函数。

程序清单 9.34 通道组转换结束标志位相关函数

```

/**
 * \brief 获取通道组转换结束标志位
 *
 * \param[in] p_hw_adc : 指向 ADC 结构体的指针
 *
 * \retval AM_TRUE : 通道组转换完成
 * \retval AM_FALSE: 通道组转换未完成
 */
am_static_inline
am_bool_t amhw_zmf2x6_adc_int_tran_complete_check (amhw_zmf2x6_adc_t *p_hw_adc)
{
    return (((p_hw_adc->adsta >> 0) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
}

/**
 * \brief 清除通道组转换结束标志位
 *
 * \param[in] p_hw_adc : 指向 ADC 结构体的指针
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_int_tran_complete_clr (amhw_zmf2x6_adc_t *p_hw_adc)
{
    p_hw_adc->adsta |= (1ul << 0);
}

```

7. ADC_ADDR（数据寄存器）

存在 12 个该寄存器，分别对应一个 ADC 通道，寄存器结构详见表 9.6。

表 9.6 ADC_ADDR 寄存器分析

位	标识符	含义
21	VALID	相应模拟通道转换完成后，该位置位，读 ADDATA 寄存器后，该位由硬件清除。
20	OVERRUN	新的转换结果装载至寄存器之前，若 DATA[11: 0]的数据没有被读取，OVERRUN 将置 1。读 ADDATA 寄存器硬件清除。
0:15	DATA	用于获取 12 位 A/D 转换结果

通过分析可以定义 3 个函数：数据有效位检测函数、数据覆盖标志位函数，获取 12 位 A/D 转换结果函数。

程序清单 9.35 为通道 0~9、14~15 数据有效标志位检测函数。

程序清单 9.35 获取通道数据有效标志位函数

```
/**
 * \brief 获取通道 0~9、14~15 数据有效标志位
 *
 * \param[in] p_hw_adc: 指向 ADC 结构体的指针
 * \param[in] chan :ADC 通道
 *
 * \note channel 0~9、14~15
 *
 * \retval AM_TRUE: 数据有效
 * \retval AM_FALSE: 数据无效
 */
am_static_inline
am_bool_t amhw_zmf2x6_adc_chan0_15_valid_check(amhw_zmf2x6_adc_t *p_hw_adc,
                                                amhw_zmf2x6_adc_channel_t chan)
{
    return (((p_hw_adc->addr[chan] >> 21) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
}
```

程序清单 9.36 为通道 0~9、14~15 数据覆盖标志位检测函数。新的转换结果装载至寄存器之前，若 DATA[11: 0]的数据没有被读取，OVERRUN 将置‘1’，读 ADDATA 后，该位由硬件清除。

程序清单 9.36 获取通道数据覆盖标志位函数

```
/**
 * \brief 获取通道 0~9、14~15 数据覆盖标志位
 *
 * \param[in] p_hw_adc: 指向 ADC 结构体的指针
 * \param[in] chan :ADC 通道
 *
 * \note channel 0~9、14~15
 *
```

```

* \retval AM_TRUE : 数据被覆盖
* \retval AM_FALSE: 数据最近一次转换结果
*/
am_static_inline
am_bool_t amhw_zmf2x6_adc_chan0_15_overrun_check (amhw_zmf2x6_adc_t      *p_hw_adc,
                                                    amhw_zmf2x6_adc_channel_t chan)
{
    return (((p_hw_adc->addr[chan] >> 20) & 0x1u) > 0 ? AM_TRUE : AM_FALSE);
}

```

程序清单 9.37 为获取 ADC 通道数据函数，调用该函数后数据覆盖标志位、数据有效标志位清零。

程序清单 9.37 获取 ADC 通道数据函数

```

/**
* \brief 获取 ADC 通道数据
*
* \param[in] p_hw_adc : 指向 ADC 结构体的指针
* \param[in] chan      : ADC 通道
*
* \note channel 0~9 、 14~15
*
* \return ADC 转换数据
*/
am_static_inline
uint16_t amhw_zmf2x6_adc_chan0_15_data_get (amhw_zmf2x6_adc_t      *p_hw_adc,
                                              amhw_zmf2x6_adc_channel_t  chan)
{
    return (uint16_t)((p_hw_adc->addr[chan]) & 0xffffu);
}

```

8. ADC_ADSTA_EXT (扩展状态寄存器)

该寄存器为 ADC_ADSTA 的扩展，因此其功能是类似的，因此实现在了一起，详情见 ADC_ADSTA (状态寄存器)。

9. ADC_CHANY0 (任意通道选择寄存器 0)

包含通道 0~7 的通道选择，与 ADC_CHANY1 寄存器一起只使用一个函数，具体实现见。

10. ADC_CHANY1 (任意通道选择寄存器 1)

包含通道 8~9、14~15 的选择，只包含一个函数。程序清单 9.38 为通道序列配置选择函数。

程序清单 9.38 通道序列中通道设置函数

```

/**
* \brief ADC 次序

```

```

*/
typedef enum amhw_zmf2x6_adc_regular_channel_order {
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_1st = 0,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_2nd,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_3rd,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_4th,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_5th,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_6th,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_7th,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_8th,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_9th,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_10th,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_15th = 14,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_16th = 15,
} amhw_zmf2x6_adc_regular_channel_order_t;

/**
 * \brief 通道序列中通道设置
 * \param[in] p_hw_adc : 指向 ADC 结构体的指针
 * \param[in] order : 传入参数应为 amhw_zmf2x6_adc_regular_channel_order_t 中的枚举变量
 * \param[in] chan : 传入参数应为 amhw_zmf2x6_adc_channel_t 中的枚举变量
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_regular_channel_order_set (amhw_zmf2x6_adc_t
*p_hw_adc,
                                                amhw_zmf2x6_adc_regular_channel_order_t order,
                                                amhw_zmf2x6_adc_channel_t chan)
{
    if ((order / 8) == 0) {

        /* 第 1~8 个转换, 在 chany0 寄存器中设定 */
        p_hw_adc->chany[0] = (p_hw_adc->chany[0] & ~(0xf << (order * 4))) | (chan << (order * 4));
    } else {

        /* 第 9~16 个转换, 在 chany1 寄存器中设定 */
        p_hw_adc->chany[1] = (p_hw_adc->chany[1] & ~(0xf << ((order - 8) * 4))) | (chan << ((order - 8) *
4));
    }
}

```

11. ADC_ANY_CFG（任意通道配置寄存器 1）

只包含一个 4bit 的寄存器，即任意通道数配置，最多可配置 15 个通道，相关函数实现如程序清单 9.39 所示。

程序清单 9.39 通道序列长度设置

```

/**
 * \brief ADC 通道序列长度
 */
typedef enum amhw_zmf2x6_adc_regular_channel_length {
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_1 = 0,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_2,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_3,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_4,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_5,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_6,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_7,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_8,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_9,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_10,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_11,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_12,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_13,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_14,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_15,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_16,
} amhw_zmf2x6_adc_regular_channel_length_t;

/**
 * \brief 通道序列长度设置
 * \param[in] p_hw_adc : 指向 ADC 结构体的指针
 * \param[in] length : 传入参数应为 amhw_zmf2x6_adc_regular_channel_length_t 中的枚举变量
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_adc_regular_channel_length_set (amhw_zmf2x6_adc_t
*p_hw_adc,
amhw_zmf2x6_adc_regular_channel_length_t
length)
{
    p_hw_adc->any_cfg = (p_hw_adc->any_cfg & ~(0xful << 0)) | ((uint32_t)(length & 0xful) << 0);
}

```

12. ADC_ANY_CR（任意通道控制寄存器）

该寄存器只包含一个有效位，即任意通道配置模式使能位。在使能后，配置 ADC 通道功能发生变化。原先通道只由寄存器 ADC_ADCHS 的 CHENx 控制选择，现在由两部分共同控制，CHANY_NUM 是配置通道 0~15 中的通道数，然后通道 0~15 分别由 CHANY_SEL0~CHANY_SEL15 配置为任意 ADC 通道。

因此定义了两个函数,分别使任意通道配置模式禁能函数、任意通道配置模式使能函数。其函数代码如程序清单 9.40 所示。

程序清单 9.40 任意通道配置模式使能禁能函数

```
/**
 * \brief ADC 任意通道配置模式禁能
 */
am_static_inline
void amhw_zmf2x6_adc_chany_mode_disable (amhw_zmf2x6_adc_t *p_hw_adc)
{
    p_hw_adc->any_cr &= ~(1ul << 0);
}

/**
 * \brief ADC 任意通道配置模式使能
 */
am_static_inline
void amhw_zmf2x6_adc_chany_mode_enable (amhw_zmf2x6_adc_t *p_hw_adc)
{
    p_hw_adc->any_cr |= (1ul << 0);
}
```

9.2.3 添加其他功能性函数

在每个寄存器的操作函数编写完毕后,可能 HW 层的接口还有所欠缺,需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到,可以暂时不用考虑,直接进入驱动层的开发。在后续如果发现 HW 层欠缺哪些接口,可以反过来再增加。当然,也有可能 HW 层接口没有什么欠缺了,那样就无需再开发其它函数了。

9.2.4 添加指向寄存器块的指针

可以看到,在 HW 层编写的函数中,第一个参数均为指向 ADC 寄存器块的指针,这个指针从哪里来呢?只需要把 ADC 的基地址使用(amhw_adc_t *)强制转换即可作为这些函数的第一个参数。

为了方便用户或驱动层使用,可以将该转换使用一个宏定义好,用户直接使用宏即可。在 AMetal 中,由于这些内容与具体型号的芯片相关,这样的宏统一定义到 zmf2x6_periph_map.h 文件中,如程序清单 9.41 所示。

程序清单 9.41 外设中加入 ADC 的定义

```
/**
 * \name 外设定义
 * @{
 */
/** \brief 串口(UART2)寄存器块指针 */
#define ZMF2X6_UART2 ((amhw_zmf2x6_uart_t *)ZMF2X6_UART2_BASE)

/** \brief 电源管理单元(PWR)寄存器块指针 */
```

```

#define ZMF2X6_PWR      ((amhw_zmf2x6_pwr_t      *)ZMF2X6_PWR_BASE)

/** \brief 备份区域(BKP)寄存器块指针 */
#define ZMF2X6_BKP      ((amhw_zmf2x6_bkp_t      *)ZMF2X6_BKP_BASE)

/** \brief 模数转换(ADC)寄存器块指针 */
#define ZMF2X6_ADC      ((amhw_zmf2x6_adc_t      *)ZMF2X6_ADC_BASE)

/** \brief I2C 总线控制器(I2C0)寄存器块指针 */
#define ZMF2X6_I2C      ((amhw_zmf2x6_i2c_t      *)ZMF2X6_I2C1_BASE )
/**
 * @} zmf2x6_if_periph_map
 */

```

后续用户或驱动层使用 HW 层接口时,第一个参数直接使用 ZMF2X6_ADC 即可。至此,整个 HW 层的开发工作就结束了,如果后续开发中发现缺少某些函数,还可以再添加。

9.3 驱动层开发

参考 5.3.1 章节,新建 am_zmf2x6_adc.h 和 am_zmf2x6_adc.c 驱动文件,搭建相应的文件框架,并按规则放到对应的路径下。

9.3.1 编写结构性程序

驱动层首先应该搭建好相关结构,再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中,每个设备(ADC 就是一个设备)都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配,与设备相关的变量都应该定义在设备结构体中,而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下,设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员,用于驱动层使用 HW 层的接口。如果设备需要使用中断,还需要填入中断号信息。除此之外,一些设备的初始化配置信息,也可以由设备信息提供。

可以暂时定义设备信息结构体如程序清单 9.42 所示。后续可以根据实际情况修改。

程序清单 9.42 ADC 设备信息结构体定义

```

/**
 * \brief ADC 对应通道的 GPIO 信息
 */
typedef struct {
    amhw_zmf2x6_adc_channel_t chan_sel;    /**< \brief 对应的 ADC 通道类别 */
    uint32_t gpio;                        /**< \brief 对应的 GPIO 管脚号 */
    uint32_t func;                        /**< \brief 对应的 GPIO 功能设置 */
    uint32_t dfunc;                       /**< \brief 对应的 GPIO 功能默认设置 */
} am_zmf2x6_adc_ioinfo_t;

/**

```



```

* \brief ADC 设备信息
*/
typedef struct am_zmf2x6_adc_devinfo {

    /** \brief ADC 寄存器块的基地址 */
    uint32_t    adc_reg_base;

    /** \brief ADC 中断号 */
    int16_t     inum;

    /** \brief 时钟 ID */
    int         clk_num;

    /**
     * \brief ADC 参考电压，单位：mV
     *
     * 参考电压  $V_{ref} = (V_{refp} - V_{refn})$  为了在最大采样速率时
     * 稳定运行 ADC，确保： $V_{REFP} = V_{DDA}$   $V_{refn}$  典型值为：  $V_{ss}$  或  $V_{ssa}$  。
     *
     * \note 该参考电压由具体的电路决定
     *
     */
    uint32_t    vref;

    /** \brief 转换精度 */
    uint8_t     bits_mode;

    /** \brief ADC 对应通道的引脚信息 */
    am_zmf2x6_adc_ioinfo_t *p_ioinfo;

    /** \brief 平台初始化函数，如打开时钟，配置引脚等工作 */
    void        (*pfn_plfm_init)(void);

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

} am_zmf2x6_adc_devinfo_t;

```

● 设备结构体定义

由于 ADC 设备需要提供标准的 ADC 服务，因此，该设备至少包含一个 ADC 标准服务结构体成员，也就是 `am_adc_serv_t` 类型成员，其次，为了在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 9.43 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 9.43 ADC 设备结构体定义

```

/**
 * \brief ADC 设备实例
 */
typedef struct am_zmf2x6_adc_dev {

    /** \brief ADC 标准服务 */
    am_adc_serv_t          adc_serve;

    /** \brief 指向 ADC 设备信息的指针 */
    const am_zmf2x6_adc_devinfo_t    *p_devinfo;

    /** \brief 存放用户启动转换后的回调函数 */
    am_adc_seq_cb_t        pfn_callback;

    /** \brief 用户启动回调函数的回调函数参数 */
    void                    *p_arg;

    /** \brief 当前转换的通道 */
    uint32_t                chan;

    /** \brief 当前转换的序列描述符数组首地址 */
    am_adc_buf_desc_t       *p_desc;

    /** \brief 当前转换的序列描述符数量 */
    uint32_t                desc_num;

    /** \brief 转换的次数 */
    uint32_t                count;

    /** \brief 转换标志 */
    uint32_t                flags;

    /** \brief 对当前序列描述符已经采样的次数 */
    uint32_t                conv_cnt;

    /** \brief 正在执行当前序列描述符的索引 */
    uint32_t                desc_index;

    /** \brief 对整个序列转换完成的次数 */
    uint32_t                seq_cnt;

    /** \brief 数据右移位数 */
    uint16_t                right_bit;

```

```
} am_zmf2x6_adc_dev_t;
```

2. 搭建函数框架

函数主要分为两部分一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于 ADC 这种已经有标准接口层的函数，通常驱动层只提供两个函数给用户，初始化函数和解初始化函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为用于标准接口的 handle。

解初始化函数主要是当不需要再使用某个外设时，释放掉其占用的相关资源，参数为设备指针，无返回值。示例代码如程序清单 9.44 所示。

程序清单 9.44 驱动层函数定义示例

```
/** \brief 初始化 ADC，获取标准服务句柄 */
am_adc_handle_t am_zmf2x6_adc_init (am_zmf2x6_adc_dev_t      *p_dev,
                                   const am_zmf2x6_adc_devinfo_t *p_devinfo)
{
    return &(p_dev->adc_serv);
}

/** \brief ADC 解初始化 */
void am_zmf2x6_adc_deinit (am_adc_handle_t handle)
{
}
```

同时为了用户能够使用，还需要将这两个函数的声明放入到相应的头文件中。示例代码如程序清单 9.45 所示。

程序清单 9.45 驱动函数声明

```
/**
 * \brief ADC 初始化
 *
 * \param[in] p_dev      : 指向 ADC 设备的指针
 * \param[in] p_devinfo : 指向 ADC 设备信息的指针
 *
 * \return ADC 标准服务操作句柄 如果为 NULL，表明初始化失败
 */
am_adc_handle_t am_zmf2x6_adc_init (am_zmf2x6_adc_dev_t      *p_dev,
                                   const am_zmf2x6_adc_devinfo_t *p_devinfo);

/**
 * \brief ADC 去初始化
 *
 * \param[in] handle : am_zmf2x6_adc_init() 初始化函数获得的 ADC 服务句柄
 */
```

```

*
* \return 无
*/
void am_zmf2x6_adc_deinit (am_adc_handle_t handle);

```

- 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 6 个函数，根据标准接口层中的 struct am_adc_drv_funcs 结构体，我们需要在.c 文件中定义 6 个函数。

由于这些函数全部是放在结构体中，通过函数指针的形式提供给上层，并不需要将这些函数声明到.h 文件中供用户直接使用，因此全部以 static 的形式定义成内部函数。定义好函数后，再定义一个 struct am_adc_drv_funcs 结构体，将函数放在其中。示例代码如程序清单 9.46 所示。

程序清单 9.46 提供给标准层函数定义

```

static int __fn_adc_start (void *p_drv,
                           int chan,
                           am_adc_buf_desc_t *p_desc,
                           uint32_t desc_num,
                           uint32_t count,
                           uint32_t flags,
                           am_adc_seq_cb_t pfn_callback,
                           void *p_arg);

static int __fn_adc_stop (void *p_drv, int chan);
static int __fn_adc_rate_get (void *p_drv,
                              int chan,
                              uint32_t *p_rate);
static int __fn_adc_rate_set (void *p_drv,
                              int chan,
                              uint32_t rate);
static uint32_t __fn_bits_get (void *p_drv, int chan);
static uint32_t __fn_vref_get (void *p_drv, int chan);

/**
 * \brief ADC 服务函数
 */
static const struct am_adc_drv_funcs __g_adc_drvfuncs = {
    __fn_adc_start,
    __fn_adc_stop,
    __fn_adc_rate_get,
    __fn_adc_rate_set,
    __fn_bits_get,
    __fn_vref_get
};

/** \brief 启动 ADC 转换 */
static int __fn_adc_start (void *p_drv,

```

```

        int                chan,
        am_adc_buf_desc_t  *p_desc,
        uint32_t           desc_num,
        uint32_t           count,
        uint32_t           flags,
        am_adc_seq_cb_t     pfn_callback,
        void                *p_arg)
{

    return AM_OK;
}

/** \brief 停止转换 */
static int __fn_adc_stop (void *p_drv, int chan)
{

    return AM_OK;
}

/** \brief 获取 ADC 的采样率 */
static int __fn_adc_rate_get (void      *p_drv,
                              int        chan,
                              uint32_t   *p_rate)
{

    return AM_OK;
}

/** \brief 设置 ADC 的采样率，实际采样率可能存在差异 */
static int __fn_adc_rate_set (void      *p_drv,
                              int        chan,
                              uint32_t   rate)
{

    return AM_OK;
}

/** \brief 获取 ADC 转换精度 */
static uint32_t __fn_bits_get (void *p_drv, int chan)
{

    return AM_OK;
}

```

```

/** \brief 获取 ADC 参考电压 */
static uint32_t __fn_vref_get(void *p_drv, int chan)
{

    return AM_OK;
}

```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。

9.3.2 驱动函数实现

通过上面的函数框架搭建，共计 8 个函数，6 个功能性函数面向标准层，2 个初始解初始化函数面向用户。除了这上述 8 个函数外，我们还可能会将要调用的一些功能代码封装为相应的函数以保证程序的易维护性，在这里我们还添加了 __adc_int_work_startup 函数，其只被 __fn_adc_start 调用。由于 ADC 能够产生中断因此我们定义函数 __fn_adc_connect 用于连接并使能相关中断，并编写相应的中断服务函数（__zmf2x6_adc_irq_handle）。下面逐步实现。

1. am_zmf2x6_adc_init 函数

该函数主要完成两个功能：

- 初始化设备中的标准服务，将成员赋值，确定 p_drv 的值，p_drv 的值确定了提供给标准接口层的那 6 个函数的第一个参数是什么，那 6 个函数才能根据这儿的设置去使用。（void * 类型无法使用，必须转换为具体的一种类型才能使用），通常情况下，p_drv 都是设置为设备结构体的地址，这样方便访问整个设备的相关内容。
- 根据 p_devinfo 内配置的信息进行一些硬件的基础配置，并进行一些必要的初始化配置。

初始化硬件相关设置，该函数的示例代码如程序清单 9.47 所示。

程序清单 9.47 ADC 初始化函数

```

/**
 * \brief ADC 初始化
 */
am_adc_handle_t am_zmf2x6_adc_init(am_zmf2x6_adc_dev_t *p_dev,
                                   const am_zmf2x6_adc_devinfo_t *p_devinfo)
{
    uint16_t bit = 12;
    uint32_t sample_rate = 0;

    amhw_zmf2x6_adc_t *p_hw_adc = NULL;

    if (NULL == p_devinfo || NULL == p_dev) {
        return NULL;
    }

    p_dev->p_devinfo = p_devinfo;
    p_dev->adc_serve.p_funcs = &__g_adc_drvfuncs;
}

```

```

p_dev->adc_serve.p_drv    = p_dev;

p_dev->pfn_callback        = NULL;
p_dev->p_desc              = NULL;
p_dev->p_arg               = NULL;
p_dev->desc_num            = 0;
p_dev->flags               = 0;
p_dev->count              = 0;
p_dev->chan                = 0;
p_dev->seq_cnt            = 0;
p_dev->desc_index         = 0;
p_dev->conv_cnt           = 0;

/* 平台初始化 */
if (p_devinfo->pfn_plfm_init) {
    p_devinfo->pfn_plfm_init();
}

p_hw_adc = (amhw_zmf2x6_adc_t*)(p_dev->p_devinfo->adc_reg_base);

/* ADC 禁能*/
amhw_zmf2x6_adc_cgf_reg_clr(p_hw_adc, AMHW_ZMF2X6_ADC_MODULE_EN_MASK);

/* 单次转换 */
amhw_zmf2x6_adc_data_scan_set(p_hw_adc, AMHW_ZMF2X6_ADC_ONE_TIME_SCAN);

/* 设置分辨率 */
amhw_zmf2x6_adc_data_resolution(p_hw_adc, p_dev->p_devinfo->bits_mode);

/*
 * 经证实芯片对于左右对齐功能无效，
 * 采样率精度为 8,9,10,11,12 位，芯片采集的数字值都是从 11bit 位开始对齐
 * 如设置为 9 位精度数据如下：
 *  |_15_~_12_|__9bit_data_|_2_~_0|
 */
amhw_zmf2x6_adc_align_way_sel(p_hw_adc, AMHW_ZMF2X6_ADC_DATA_ALIGN_RIGHT);

bit = __fn_bits_get(p_dev, 0);
p_dev->right_bit = 12 - bit;

/* 设置采样率 */
__fn_adc_rate_set(p_dev, 0, 1000000);

/* 注册 ADC 相关中断 */

```

```

__fn_adc_connect(p_dev);

/* ADC 使能 */
amhw_zmf2x6_adc_cfg_reg_set(p_hw_adc, AMHW_ZMF2X6_ADC_MODULE_EN_MASK);

return (am_adc_handle_t)(&(p_dev->adc_serve));
}

```

2. am_zmf2x6_adc_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放，仅将部分结构体指向 NULL 即可。另外是对硬件资源的释放，包括禁能 ADC 及相关中断，执行 p_devinfo 内的平台接初始化函数详见程序清单 9.48。

程序清单 9.48 ADC 去初始化函数

```

/**
 * \brief ADC 去初始化
 */
void am_zmf2x6_adc_deinit (am_adc_handle_t handle)
{
    am_zmf2x6_adc_dev_t *p_dev    = (am_zmf2x6_adc_dev_t *)handle;
    amhw_zmf2x6_adc_t    *p_hw_adc = NULL;

    if (NULL == p_dev) {
        return ;
    }

    p_hw_adc = (amhw_zmf2x6_adc_t *) (p_dev->p_devinfo->adc_reg_base);
    am_int_disable(p_dev->p_devinfo->inum);

    /* 中断禁能 */
    amhw_zmf2x6_adc_ctrl_reg_clr(p_hw_adc, AMHW_ZMF2X6_ADC_INT_EN_MASK);

    /* ADC 禁能 */
    amhw_zmf2x6_adc_cfg_reg_clr(p_hw_adc, AMHW_ZMF2X6_ADC_MODULE_EN_MASK);

    p_dev->adc_serve.p_drv = NULL;

    if (p_dev->p_devinfo->pfn_plfm_deinit) {
        p_dev->p_devinfo->pfn_plfm_deinit();
    }
}

```

3. __fn_adc_start

程序清单 9.49 为 adc 启动函数，其内部主要涉及到 p_dev 结构体内容的填充，并使能中断。同时其还调用了函数 __adc_int_work_startup 用于启动 ADC 工作，详见程序清单 9.50，其主要涉及到引脚的初始化、ADC 采样通道相关操作。中断使能、ADC 转换使能并

启动转换。

程序清单 9.49 启动 ADC 转换函数

```
/**
 * \brief 启动 ADC 转换
 */
static int __fn_adc_start (void                *p_drv,
                          int                  chan,
                          am_adc_buf_desc_t   *p_desc,
                          uint32_t            desc_num,
                          uint32_t            count,
                          uint32_t            flags,
                          am_adc_seq_cb_t     pfn_callback,
                          void                 *p_arg)
{
    am_zmf2x6_adc_dev_t *p_dev = NULL;
    if (NULL == p_drv) {
        return -AM_EINVAL;
    }
    p_dev = (am_zmf2x6_adc_dev_t *)p_drv;
    p_dev->p_desc      = p_desc;
    p_dev->chan        = chan;
    p_dev->desc_num     = desc_num;
    p_dev->count        = count;
    p_dev->flags        = flags;
    p_dev->pfn_callback = pfn_callback;
    p_dev->p_arg        = p_arg;
    p_dev->seq_cnt      = 0;
    p_dev->desc_index   = 0;
    p_dev->conv_cnt     = 0;
    am_int_enable(p_dev->p_devinfo->inum); /* 开启中断 */
    __adc_int_work_startup(p_dev,chan); /* 中断工作模式启动配置 */
    return AM_OK;
}
```

程序清单 9.50 ADC 使用中断模式时的启动配置函数

```
/**
 * \brief ADC 使用中断模式时启动配置
 */
static void __adc_int_work_startup (am_zmf2x6_adc_dev_t *p_dev, uint32_t chan)
{
    amhw_zmf2x6_adc_t *p_hw_adc = (amhw_zmf2x6_adc_t *) (p_dev->p_devinfo->adc_reg_base);
    if (p_dev->p_devinfo->p_ioinfo != NULL) {
        /* ADC 禁能*/
    }
}
```

```

amhw_zmf2x6_adc_cfg_reg_clr(p_hw_adc, AMHW_ZMF2X6_ADC_MODULE_EN_MASK);

/* 初始化引脚 */
if (p_dev->p_devinfo->p_ioinfo[chan].chan_sel < AMHW_ZMF2X6_ADC_TEMP_SENSOR) {
    /* GPIO 引脚模式设置 */
    am_gpio_pin_cfg(p_dev->p_devinfo->p_ioinfo[chan].gpio,
                    p_dev->p_devinfo->p_ioinfo[chan].func);
}
/* 通道使能 */
amhw_zmf2x6_adc_input_chan_enable(p_hw_adc, (amhw_zmf2x6_adc_channel_t)chan);
/* 设置规则通道个数 1 个 */
amhw_zmf2x6_adc_regular_channel_length_set(
    p_hw_adc, AMHW_ZMF2X6_ADC_REGULAR_CHAN_LENGTH_1);
/* 连接规则序列通道和 ADC 采样通道 */
amhw_zmf2x6_adc_regular_channel_order_set(
    p_hw_adc,
    AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_1st,
    (amhw_zmf2x6_adc_channel_t)chan);
/* 任意通道配置模式使能 */
amhw_zmf2x6_adc_chany_mode_enable(p_hw_adc);
}
/* 中断使能 */
amhw_zmf2x6_adc_ctrl_reg_set(p_hw_adc, AMHW_ZMF2X6_ADC_INT_EN_MASK);
/* 清除标志位 */
amhw_zmf2x6_adc_int_tran_complete_clr(p_hw_adc);
amhw_zmf2x6_adc_int_win_cmp_clr(p_hw_adc);
/* ADC 使能 */
amhw_zmf2x6_adc_cfg_reg_set(p_hw_adc, AMHW_ZMF2X6_ADC_MODULE_EN_MASK);
/* 启动转换 */
amhw_zmf2x6_adc_data_transfer_set(p_hw_adc, AMHW_ZMF2X6_ADC_DATA_TRANSFER_START);
}

```

4. `__fn_adc_stop`

ADC 停止函数涉及到中断的禁止、ADC 转换停止、禁能 ADC 转换及 ADC 通道的禁能操作，并恢复引脚为默认配置，示例代码详见程序清单 9.51。

程序清单 9.51 ADC 停止转换函数

```

/**
 * \brief 停止转换
 */
static int __fn_adc_stop (void *p_drv, int chan)
{
    am_zmf2x6_adc_dev_t *p_dev    = (am_zmf2x6_adc_dev_t *)p_drv;
    amhw_zmf2x6_adc_t    *p_hw_adc = (amhw_zmf2x6_adc_t *) (p_dev->p_devinfo->adc_reg_base);
    if (NULL == p_drv) {

```

```

        return -AM_EINVAL;
    }
    /* 中断禁能 */
    amhw_zmf2x6_adc_ctrl_reg_clr(p_hw_adc, AMHW_ZMF2X6_ADC_INT_EN_MASK);
    /* 单次转换停止 */
    amhw_zmf2x6_adc_data_transfer_set(p_hw_adc, AMHW_ZMF2X6_ADC_DATA_TRANSFER_END);
    /* 通道禁能 */
    amhw_zmf2x6_adc_input_chan_disable(p_hw_adc, (amhw_zmf2x6_adc_channel_t)chan);
    /* ADC 禁能 */
    amhw_zmf2x6_adc_cfg_reg_clr(p_hw_adc, AMHW_ZMF2X6_ADC_MODULE_EN_MASK);
    if (p_dev->p_devinfo->p_ioinfo != NULL) {
        /* 引脚恢复默认设置 */
        if (p_dev->p_devinfo->p_ioinfo[chan].chan_sel < AMHW_ZMF2X6_ADC_TEMP_SENSOR) {
            /* GPIO 引脚模式设置 */
            am_gpio_pin_cfg(p_dev->p_devinfo->p_ioinfo[chan].gpio,
                           p_dev->p_devinfo->p_ioinfo[chan].dfunc);
        }
    }
    am_int_disable(p_dev->p_devinfo->inum);
    return AM_OK;
}

```

5. __fn_adc_rate_get

该函数为 ADC 采样率获取函数，通过相关硬件层函数获取到 ADC 的预分频值、分辨率和通道采样时间值，在调用 am_clk_rate_get 函数获取 ADC 的时钟源 APB2 的时钟频率，计算出此时的 ADC 采样率。具体计算公式如下：

采样时间 = (ADC 输入时钟 / 时钟分频) / (转换时间 + 采样时间)

示例代码详见程序清单 9.52。

程序清单 9.52 获取 ADC 的采样率

```

/**
 * \brief 获取 ADC 的采样率
 */
static int __fn_adc_rate_get(void *p_drv,
                             int chan,
                             uint32_t *p_rate)
{
    am_zmf2x6_adc_dev_t *p_dev;
    amhw_zmf2x6_adc_t *p_hw_adc;
    uint32_t apb2_clk, adc_clk;
    uint32_t i;
    uint16_t conv_time;
    uint16_t sample;
    uint16_t total_conv_time; /* 总转换时间 */
    uint8_t data_res; /* ADC 分辨率 */

```

```

uint32_t    adc_clkdiv = 1;
/* 转换时间 */
uint16_t    bit_conv_time[5] = {
    13, 12, 11, 10, 9
};
uint16_t    samp_time[13] = {
    2, 8, 14, 29, 42, 56, 72, 240,
    3, 4, 5, 6, 7
};
if (NULL == p_drv) {
    return -AM_EINVAL;
}
p_dev      = (am_zmf2x6_adc_dev_t *)p_drv;
p_hw_adc   = (amhw_zmf2x6_adc_t *) (p_dev->p_devinfo->adc_reg_base);

apb2_clk = am_clk_rate_get(p_dev->p_devinfo->clk_num);
/* 获取 ADC 分辨率 */
data_res = amhw_zmf2x6_adc_data_resolution_get(p_hw_adc);
/* 转换时间 */
conv_time = bit_conv_time[data_res];
i = amhw_zmf2x6_adc_chan_sample_time_get(p_hw_adc);
sample = samp_time[i];
total_conv_time = (uint16_t)(sample + conv_time);
/* 获取 ADC 分频值 */
adc_clkdiv = amhw_zmf2x6_adc_prescaler_val_get(p_hw_adc);
/* 获取 ADC 时钟 */
adc_clk = apb2_clk / adc_clkdiv;
*p_rate = adc_clk / total_conv_time;
return AM_OK;
}

```

6. fn_adc_rate_set

该函数用于设置 ADC 的采样率。关于 ADC 的采样率主要与输入时钟、分辨率、ADC 预分频值和通道采样时间相关。在实现时使用相关算法，通过对 ADC 输入时钟及所需要的采样率进行相关计数，获取得到最靠近的 ADC 预分频值和通道采样时间的值。具体计算公式如下：

采样时间 = (ADC 输入时钟 / 时钟分频) / (转换时间+采样时间)

示例代码详见程序清单 9.53。

程序清单 9.53 设置 ADC 的采样率函数

```

/**
 * \brief 设置 ADC 的采样率，实际采样率可能存在差异
 */
static int __fn_adc_rate_set(void    *p_drv,
                                int    chan,

```

```

        uint32_t rate)
{
    /* 转换时间 */
    uint16_t bit_conv_time[5] = {
        13, 12, 11, 10, 9
    };
    uint16_t samp_time[13] = {
        2, 8, 14, 29, 42, 56, 72, 240,
        3, 4, 5, 6, 7
    };
    uint32_t apb2_clk;
    uint32_t temp_rate = 0;
    uint32_t rate_err = 0;
    uint16_t conv_time;
    uint16_t total_conv_time; /* 总转换时间 */
    uint8_t data_res; /* ADC 分辨率 */
    uint8_t adc_div_min;
    uint8_t i = 0, j = 0;
    struct adc_err {
        uint32_t err;
        uint8_t sample_time;
        uint8_t adc_div;
    };
    struct adc_err adc_rate_err;
    am_zmf2x6_adc_dev_t *p_dev;
    amhw_zmf2x6_adc_t *p_hw_adc;
    if (NULL == p_drv) {
        return -AM_EINVAL;
    }
    p_dev = (am_zmf2x6_adc_dev_t *)p_drv;
    p_hw_adc = (amhw_zmf2x6_adc_t *) (p_dev->p_devinfo->adc_reg_base);

    /* 最大支持 1Msps 转换速率 */
    if (rate > 1000000) {
        rate = 1000000;
    }
    adc_rate_err.err = rate;
    adc_rate_err.sample_time = 0;
    adc_rate_err.adc_div = 0;
    apb2_clk = am_clk_rate_get(p_dev->p_devinfo->clk_num);
    /* ADC 的输入时钟不得超过 15 MHz */
    if ((apb2_clk % 15000000) != 0) {
        adc_div_min = apb2_clk / 15000000 + 1;
    } else {

```

```

        adc_div_min = apb2_clk / 15000000;
    }
    /* 获取 ADC 分辨率 */
    data_res = amhw_zmf2x6_adc_data_resolution_get(p_hw_adc);
    /* 转换时间 */
    conv_time = bit_conv_time[data_res];
    /* ADC 分频值在 2~17 */
    for(i = 2; i < 18; i++) {
        if (i < adc_div_min) { /* 小于 ADC 最小分频值 */
            continue;
        }
        for(j = 0; j < 13; j++) {
            total_conv_time = samp_time[j] + conv_time;
            temp_rate = (apb2_clk / (i * total_conv_time));
            rate_err = temp_rate >= rate ? temp_rate - rate : rate - temp_rate;
            if (rate_err < adc_rate_err.err) {
                adc_rate_err.err = rate_err;
                adc_rate_err.adc_div = i;
                adc_rate_err.sample_time = j;
            }
            if (adc_rate_err.err == 0) {
                break;
            }
        }
    }
    amhw_zmf2x6_adc_prescaler_val_set(p_hw_adc, adc_rate_err.adc_div);

    amhw_zmf2x6_adc_chan_sample_time(p_hw_adc,
                                     (amhw_zmf2x6_adc_sample_time_t)adc_rate_err.sample_time);

    return AM_OK;
}

```

7. `__fn_bits_get`

获取 ADC 转换精度，即通过 `amhw_zmf2x6_adc_data_resolution_get` 函数获取硬件的精度相关配置，再与 `p_devinfo` 中成员变量 `bits_mode` 进行比较，若一样则返回 ADC 的转换精度，返回 0 代表配置有问题。

示例代码详见程序清单 9.54。

程序清单 9.54 获取 ADC 转换精度函数

```

/**
 * \brief 获取 ADC 转换精度
 */
static uint32_t __fn_bits_get (void *p_drv, int chan)
{
    am_zmf2x6_adc_dev_t *p_dev = (am_zmf2x6_adc_dev_t *)p_drv;

```

```

amhw_zmf2x6_adc_t  *p_hw_adc = (amhw_zmf2x6_adc_t *)p_dev->p_devinfo->adc_reg_base;
uint8_t adc_bit = 12 - amhw_zmf2x6_adc_data_resolution_get(p_hw_adc);
if (adc_bit == p_dev->p_devinfo->bits_mode) {
    return p_dev->p_devinfo->bits_mode;
} else {
    return 0;
}
}

```

8. __fn_vref_get

获取 ADC 参考电压函数，由于 zmf2x6 无法进行参考电压选择，因此直接通过标准层传来的设置值作为返回。

示例代码详见程序清单 9.55。

程序清单 9.55 获取 ADC 参考电压函数

```

/**
 * \brief 获取 ADC 参考电压
 */
static uint32_t __fn_vref_get (void *p_drv, int chan)
{
    if (NULL == p_drv) {
        return 0; /* 资源已经释放，参考电压未知 */
    }
    return (uint32_t) (((am_zmf2x6_adc_dev_t *)p_drv)->p_devinfo->vref);
}

```

9. __zmf2x6_adc_irq_handle

程序清单 9.56 为 ADC 中断服务函数，在该函数内主要完成标志位的检测，并根据标志位确定是哪个动作引起的中断，从而调用相应的中断回调函数。

程序清单 9.56 ADC 数据转换完成中断

```

/**
 * \brief ADC 数据转换完成中断
 */
void __zmf2x6_adc_irq_handle (void *p_arg)
{
    am_zmf2x6_adc_dev_t  *p_dev      = (am_zmf2x6_adc_dev_t *)p_arg;
    amhw_zmf2x6_adc_t     *p_hw_adc  = NULL;
    /* 当前转换的序列描述符 */
    am_adc_buf_desc_t *p_desc      = &(p_dev->p_desc[p_dev->desc_index]);
    uint16_t             *p_buf16   = (uint16_t *)p_desc->p_buf;
    uint8_t              *p_buf8    = (uint8_t *)p_desc->p_buf;
    uint16_t             adc_dat = 0;
    if (p_buf16 == NULL) {
        return ;
    }
}

```

```

}
p_hw_adc = (amhw_zmf2x6_adc_t*)(p_dev->p_devinfo->adc_reg_base);
/* 判断是否当前数据是有效的 */
if (p_dev->conv_cnt < p_desc->length) {
    /* 从全局数据寄存器中读取数据，以清除中断标志 */
    amhw_zmf2x6_adc_data_get(p_hw_adc, &adc_dat);
    if (p_dev->p_devinfo->bits_mode ==
        AMHW_ZMF2X6_ADC_DATA_VALID_8BIT) {
        p_buf8[p_dev->conv_cnt] = adc_dat >> p_dev->right_bit;
    } else {
        /* 保存数据 */
        if (AM_ADC_DATA_ALIGN_LEFT & p_dev->flags) {
            p_buf16[p_dev->conv_cnt] = adc_dat << 4;
        } else {
            p_buf16[p_dev->conv_cnt] = adc_dat >> p_dev->right_bit;
        }
    }
}
p_dev->conv_cnt++;
/* 判断当前序列描述符是否已经完成转换 */
if (p_dev->conv_cnt >= p_desc->length) {
    p_dev->conv_cnt = 0;
    if (NULL != p_desc->pfn_complete) {
        p_desc->pfn_complete(p_desc->p_arg, AM_OK);
    }
    p_dev->desc_index++;
    /* 判断整个序列描述符是否完成一轮转换 */
    if (p_dev->desc_index == p_dev->desc_num) {
        p_dev->desc_index = 0;
        if (NULL != p_dev->pfn_callback) {
            p_dev->pfn_callback(p_dev->p_arg, AM_OK);
        }
        p_dev->seq_cnt++; /* 整个序列转换完成 */
        if (p_dev->count != 0 && p_dev->seq_cnt == p_dev->count) {
            p_dev->seq_cnt = 0;
            if (amhw_zmf2x6_adc_int_tran_complete_check(p_hw_adc)){
                amhw_zmf2x6_adc_int_tran_complete_clr(p_hw_adc);
            }
            __fn_adc_stop(p_dev, p_dev->chan); /* 关闭模块 */
            return ; /* 返回 */
        }
    }
}
} else {
    if (NULL != p_dev->pfn_callback) {

```



```

        p_dev->pfn_callback(p_dev->p_arg, AM_ERROR);
    }
    if (amhw_zmf2x6_adc_int_tran_complete_check(p_hw_adc)){
        amhw_zmf2x6_adc_int_tran_complete_clr(p_hw_adc);
    }
    __fn_adc_stop (p_dev, p_dev->chan);
}
}

```

9.4 用户配置文件编写

9.4.1 定义设备常量于设备信息常量

创建配置文件后，我们需要在里面加入设备变量，设备变量只需要定义，不用初始化。设备信息常量定义时初始化，默认填充的信息是一种常用的配置方式，ADC 设备信息配置如程序清单 10.37 所示。

程序清单 9.57 ADC 设备信息配置文件

```

/**
 * \brief ADC 的引脚配置信息列表
 *
 * \note 每个成员代表每个 ADC 通道的引脚配置,例如: AMHW_ZMF2X6_ADC_CHAN_0 ,代表 PA0
第一个有效元素引脚
 *
 * ADC 通道 0 的引脚配置
 *
 */
am_zmf2x6_adc_ioinfo_t __g_adc_ioinfo_list[] = {
    {AMHW_ZMF2X6_ADC_CHAN_0,  PIOA_0, PIOA_0_GPIO | PIOA_0_AIN, PIOA_0_GPIO |
PIOA_0_INPUT_PU },
    {AMHW_ZMF2X6_ADC_CHAN_1,  PIOA_1, PIOA_1_GPIO | PIOA_1_AIN, PIOA_1_GPIO |
PIOA_1_INPUT_PU },
    {AMHW_ZMF2X6_ADC_CHAN_2,  PIOA_2, PIOA_2_GPIO | PIOA_2_AIN, PIOA_2_GPIO |
PIOA_2_INPUT_PU },
    {AMHW_ZMF2X6_ADC_CHAN_3,  PIOA_3, PIOA_3_GPIO | PIOA_3_AIN, PIOA_3_GPIO |
PIOA_3_INPUT_PU },
    {AMHW_ZMF2X6_ADC_CHAN_4,  PIOA_4, PIOA_4_GPIO | PIOA_4_AIN, PIOA_4_GPIO |
PIOA_4_INPUT_PU },
    {AMHW_ZMF2X6_ADC_CHAN_5,  PIOA_5, PIOA_5_GPIO | PIOA_5_AIN, PIOA_5_GPIO |
PIOA_5_INPUT_PU },
    {AMHW_ZMF2X6_ADC_CHAN_6,  PIOA_6, PIOA_6_GPIO | PIOA_6_AIN, PIOA_6_GPIO |
PIOA_6_INPUT_PU },
    {AMHW_ZMF2X6_ADC_CHAN_7,  PIOA_7, PIOA_7_GPIO | PIOA_7_AIN, PIOA_7_GPIO |
PIOA_7_INPUT_PU },
    {AMHW_ZMF2X6_ADC_CHAN_8,  PIOB_0, PIOB_0_GPIO | PIOB_0_AIN, PIOB_0_GPIO |
PIOB_0_INPUT_PU },

```

```

    {AMHW_ZMF2X6_ADC_CHAN_9,  PIOB_1, PIOB_1_GPIO | PIOB_1_AIN, PIOB_1_GPIO |
PIOB_1_INPUT_PU },
};
/** \brief ADC 平台初始化 */
static void __zmf2x6_plfm_adc_init (void)
{
    am_clk_enable(CLK_ADC1);
}
/** \brief 解除 ADC 平台初始化 */
static void __zmf2x6_plfm_adc_deinit (void)
{
    am_clk_disable (CLK_ADC1);
}
/** \brief ADC 设备信息 */
static const am_zmf2x6_adc_devinfo_t __g_adc_devinfo = {
    ZMF2X6_ADC_BASE,                /**< \brief ADC */
    INUM_ADC_COMP,                  /**< \brief ADC 的中断编号 */
    CLK_ADC1,                        /**< \brief ADC 时钟号 */
    3300,                            /**< \brief 参考电压 */
    AMHW_ZMF2X6_ADC_DATA_VALID_12BIT, /**< \brief 转换精度 */
    &__g_adc_ioinfo_list[0],         /**< \brief 引脚信息列表 */
    __zmf2x6_plfm_adc_init,          /**< \brief ADC 的平台初始化 */
    __zmf2x6_plfm_adc_deinit,        /**< \brief ADC 的平台去初始化 */
};
static am_zmf2x6_adc_dev_t  __g_adc_dev; /**< \brief 定义 ADC 设备 */

```

9.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。平台初始化函数和解初始化函数定义详见程序清单 10.38。

程序清单 9.58 平台初始化函数和解初始化函数定义

```

/** \brief ADC 实例初始化，获得 ADC 标准服务句柄 */
am_adc_handle_t am_zmf2x6_adc_inst_init (void)
{
    return am_zmf2x6_adc_init(&__g_adc_dev, &__g_adc_devinfo);
}
/** \brief ADC 实例解初始化 */
void am_zmf2x6_adc_inst_deinit (am_adc_handle_t handle)
{
    am_zmf2x6_adc_deinit(handle);
}

```

9.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在标准层实例公共头文件 am_zmf2x6_inst_init.h 中加入

它们的声明，如所示程序清单 10.39。

程序清单 9.59 平台初始化解初始化声明

```
/**
 * \brief ADC 实例初始化，获得 ADC 标准服务句柄
 *
 * \param 无
 *
 * \return ADC 标准服务句柄，若为 NULL，表明初始化失败
 */
am_adc_handle_t am_zmf2x6_adc_inst_init(void);
/**
 * \brief ADC 实例解初始化
 *
 * \param[in] handle 通过 am_zmf2x6_adc_inst_init() 函数获得的 ADC 标准服务句柄
 *
 * \return 无
 */
void am_zmf2x6_adc_inst_deinit(am_adc_handle_t handle);
```

用户使用时，应该先调用平台初始化函数 `am_zmf2x6_adc_inst_init` 函数，初始化 ADC，并保存其返回的 `handle`，后续直接使用保存的 `handle` 作为标准接口的第一个参数使用标准接口函数即可。

至此，用户配置文件开发完毕。

9.5 例程文件开发

参考 5.5.1 章节新建用户例程文件，搭建文件的基本框架，准备 ADC 例程内容的编写。例程文件开发主要分为两个部分，HW 例程和 STD 例程。

9.5.1 例程内容开发

1. HW 层实现

这里使用 HW 层来实现 ADC 中断采集电压值。主要涉及到中断的使能，中断服务函数的编写，ADC 采集值的获取。为提高案例的普适性该案例分两个文件编写，相关代码如程序清单 9.60 程序清单 9.61 所示。

程序清单 9.60 ADC 中断采集电压值 HW 层 DEMO

```
/** [src_zmf2x6_hw_adc_int] */
#include "ametal.h"
#include "am_vdebug.h"
#include "am_delay.h"
#include "am_int.h"
#include "hw/amhw_zmf2x6_adc.h"

static volatile uint16_t __g_adc_dat[10];          /**< \brief 采样值缓存 */
static volatile am_bool_t __g_adc_complete = AM_FALSE;
```

```

/**
 * \brief ADC 中断服务函数
 */
static void __adc_isr (void *p_arg)
{
    static uint8_t i = 0;

    amhw_zmf2x6_adc_t *p_hw_adc = (amhw_zmf2x6_adc_t *)p_arg;

    amhw_zmf2x6_adc_data_get(p_hw_adc, (uint16_t *)&__g_adc_dat[i]);
    __g_adc_dat[i] &= 0x0fff;

    i++;
    if (i == 10) {
        i = 0;
        __g_adc_complete = AM_TRUE;

        amhw_zmf2x6_adc_ctrl_reg_clr(p_hw_adc, AMHW_ZMF2X6_ADC_INT_EN_MASK);

        amhw_zmf2x6_adc_data_transfer_set(p_hw_adc,
                                           AMHW_ZMF2X6_ADC_DATA_TRANSFER_END);
    }
}

/** \brief 例程入口 */
void demo_zmf2x6_hw_adc_int_entry (amhw_zmf2x6_adc_t *p_hw_adc,
                                   int int_num,
                                   int chan,
                                   uint32_t vref_mv)
{
    uint8_t i = 0;
    uint32_t adc_mv = 0; /* 采样电压 */
    uint32_t sum = 0;

    am_kprintf("The ADC HW Int Demo\r\n");

    amhw_zmf2x6_adc_data_resolution(p_hw_adc, AMHW_ZMF2X6_ADC_DATA_VALID_12BIT);
    amhw_zmf2x6_adc_prescaler_val_set(p_hw_adc, 4);

    /* 连接转换完成中断 */
    am_int_connect(int_num, __adc_isr, p_hw_adc);
    am_int_enable(int_num);
}

```

```

amhw_zmf2x6_adc_input_chan_enable(p_hw_adc, (amhw_zmf2x6_adc_channel_t)chan);
amhw_zmf2x6_adc_cfg_reg_set(p_hw_adc, AMHW_ZMF2X6_ADC_MODULE_EN_MASK);

while(1) {
    amhw_zmf2x6_adc_ctrl_reg_set(p_hw_adc, AMHW_ZMF2X6_ADC_INT_EN_MASK);

    amhw_zmf2x6_adc_data_transfer_set(p_hw_adc,
AMHW_ZMF2X6_ADC_DATA_TRANSFER_START);

    /* 等待转换完成 */
    while (__g_adc_complete == AM_FALSE);

    for (sum = 0, i = 0; i < 10; i++) {
        sum += __g_adc_dat[i];
    }

    sum = sum / 10;

    /* 转换为电压值对应的整数值 */
    adc_mv = sum * vref_mv / ((1UL << 12) - 1);

    am_kprintf("Sample : %d, Vol: %d mv\r\n", sum, adc_mv);

    am_mdelay(500);
}
}
/** [src_zmf2x6_hw_adc_int] */

```

程序清单 9.61 ADC 中断采集电压值案例入口

```

/** [src_zmf2x6_hw_adc_int] */
#include "ametal.h"
#include "am_vdebug.h"
#include "am_zmf2x6.h"
#include "am_gpio.h"
#include "am_clk.h"
#include "demo_zmf2x6_entries.h"
#include "demo_zmf2x6_core_entries.h"

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_hw_adc_int_entry (void)
{
    AM_DBG_INFO("demo zmf2x6_core hw adc int!\r\n");
}

```

```

/* 配置引脚 */
am_gpio_pin_cfg(PIOA_0, PIOA_0_GPIO | PIOA_0_AIN);

/* 使能时钟 */
am_clk_enable(CLK_ADC1);

demo_zmf2x6_hw_adc_int_entry(ZMF2X6_ADC,
                              INUM_ADC_COMP,
                              AMHW_ZMF2X6_ADC_CHAN_0,
                              3300);
}
/** [src_zmf2x6_hw_adc_int] */

```

2. 标准层实现

使用标准层实现 ADC 的多通道采集，由于使用的是标准层例程，因此直接调用标准的例程函数 `demo_std_adc_entry` 即可。具体操作如程序清单 9.62 所示。

程序清单 9.62 ADC 的多通道采集标准层 DEMO

```

/** [src_std_adc] */
#include "ametal.h"
#include "am_board.h"
#include "am_vdebug.h"
#include "am_delay.h"
#include "am_zmf2x6_inst_init.h"
#include "demo_std_entries.h"
#include "demo_zmf2x6_core_entries.h"

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_std_multichan_adc_entry (void)
{
    /* ADC 通道 0,1,2,3 */
    int adc_chan[] = {0, 1, 2, 3};

    AM_DBG_INFO("demo zmf2x6_core std multichan adc int!\r\n");

    demo_std_multichan_adc_entry(am_zmf2x6_adc_inst_init(),
                                adc_chan,
                                sizeof(adc_chan) / sizeof(adc_chan[0]));
}
/** [src_std_adc] */

```

第10章 通用外设驱动开发——UART

10.1 标准层了解

UART 标准层包含 2 个文件，am_uart.h 和 am_uart_rngbuf.h 文件，可在 ametal\interface 找到该文件打开。这里，我们先说一下两个文件的差别，am_uart.h 是通用 UART 的标准层接口，am_uart_rngbuf.h 是带缓冲区的 UART 的标准层接口。本章以 am_uart.h 标准层接口为例展开。

10.1.1 了解相关类型定义

打开 am_uart.h 文件，我们可以看到，前面定义了 UART 的模式、通用指令、硬件参数、流控制、编码方式错误码、回调函数类型以及驱动函数结构体等，详见程序清单 10.1。

程序清单 10.1 UART 标准接口文件结构体定义

```
/**
 * \name UART 模式定义
 * @{
 */
#define AM_UART_MODE_POLL      1    /**< \brief UART 查询模式 */
#define AM_UART_MODE_INT      2    /**< \brief UART 中断模式 */
/** @} */

/**
 * \name UART 通用控制指令
 * @{
 */
#define AM_UART_BAUD_SET      1    /**< \brief 设置波特率 */
#define AM_UART_BAUD_GET      2    /**< \brief 获取波特率 */

#define AM_UART_OPTS_SET      3    /**< \brief 设置硬件参数设置 */
#define AM_UART_OPTS_GET      4    /**< \brief 获取硬件参数设置 */

#define AM_UART_MODE_SET      5    /**< \brief 设置模式 */
#define AM_UART_MODE_GET      6    /**< \brief 获取当前模式 */
#define AM_UART_AVAIL_MODES_GET 7    /**< \brief 获取支持的有效模式 */

#define AM_UART_FLOWMODE_SET  8    /**< \brief 设置流控模式 */
#define AM_UART_FLOWSTAT_RX_SET 9    /**< \brief 设置接收器流控状态 */
#define AM_UART_FLOWSTAT_TX_GET 10   /**< \brief 获取发送器流控状态 */

#define AM_UART_RS485_SET      11   /**< \brief 设置 RS485 模式(使能 或 禁能) */
#define AM_UART_RS485_GET      12   /**< \brief 获取当前的 RS485 模式状态 */

/** @} */

/**
```

```

* \name 硬件参数设置选项，设置硬件参数时，可以是多个选项的或值
* @{
*/

#define AM_UART_CSIZE      0xc    /**< \brief 位 3 ~ 位 4 为数据宽度 */
#define AM_UART_CS5       0x0    /**< \brief 数据宽度为 5 位 */
#define AM_UART_CS6       0x4    /**< \brief 数据宽度为 6 位 */
#define AM_UART_CS7       0x8    /**< \brief 数据宽度为 7 位 */
#define AM_UART_CS8       0xc    /**< \brief 数据宽度为 8 位 */

#define AM_UART_STOPB      0x20   /**< \brief 设置停止位为 2 位，默认为 1 位 */
#define AM_UART_PARENB     0x40   /**< \brief 使能奇偶校验，默认奇偶校验是关闭的 */
#define AM_UART_PARODD     0x80   /**< \brief 设置校验为奇校验，默认是偶校验 */

/** @} */

/**
* \name 流控制设置选项
* @{
*/

#define AM_UART_FLOWCTL_NO  0xa0   /**< \brief 无流控 */
#define AM_UART_FLOWCTL_HW  0xa1   /**< \brief 硬件流控 */
#define AM_UART_FLOWCTL_SW  0xa2   /**< \brief 软件流控 */

#define AM_UART_FLOWSTAT_ON 0xb0   /**< \brief 开流，继续数据传输 */
#define AM_UART_FLOWSTAT_OFF 0xb1  /**< \brief 关流，停止数据传输 */

/** @} */

/**
* \name 串行设备回调函数类型编码，用于指定注册何种回调函数
* @{
*/

#define AM_UART_CALLBACK_TXCHAR_GET  0  /**< \brief 获取一个发送字符 */
#define AM_UART_CALLBACK_RXCHAR_PUT  1  /**< \brief 提交一个接收到的字符 */
#define AM_UART_CALLBACK_ERROR        2  /**< \brief 错误回调函数 */

/** @} */

/**
* \name 错误回调函数的错误码（code）
* @{

```



```

*/
#define AW_UART_ERR_CODE_NONE          (-1)      /**< \brief 无错误          */
#define AW_UART_ERR_CODE_FRAMING       0         /**< \brief 帧错误          */
#define AW_UART_ERR_CODE_PARITY        1         /**< \brief 校验错误        */
#define AW_UART_ERR_CODE_OFLOW         2         /**< \brief over flow 错误 */
#define AW_UART_ERR_CODE_UFLOW         3         /**< \brief under flow 错误 */
#define AW_UART_ERR_CODE_CONNECT       4         /**< \brief 连接错误        */
#define AW_UART_ERR_CODE_DISCONNECT    5         /**< \brief 断开连接错误    */
#define AW_UART_ERR_CODE_NO_CLK        6         /**< \brief 无可有时钟      */
#define AW_UART_ERR_CODE_UNKNWN        7         /**< \brief 未知错误        */
/** @} */

/**
 * \name 回调函数类型定义
 * @{
 */

/**
 * \brief 获取一个待发送字符
 *
 * \param[in] p_arg : 设置回调函数时指定的自定义参数
 * \param[out] p_char : 获取待发送数据的指针
 *
 * \retval -AM_EEMPTY : 获取待发送数据失败, 无更多数据需要发送
 * \retval AM_OK : 获取待发送数据成功, p_char 正确装载了待发送的数据
 */
typedef int (*am_uart_txchar_get_t)(void *p_arg, char *p_char);

/**
 * \brief 提交一个接收到的字符
 *
 * \param[in] p_arg : 设置回调函数时指定的自定义参数
 * \param[in] ch : 接收到的数据
 *
 * \retval -AM_EFULL : 用户处理接收数据失败, 没有足够的内存空间
 * \retval AM_OK : 用户处理接收数据成功
 */
typedef int (*am_uart_rxchar_put_t)(void *p_arg, char ch);

/**
 * \brief 错误回调函数
 *
 * \param[in] p_arg : 设置回调函数时指定的自定义参数
 * \param[in] code : 错误代码

```

```

* \param[in] p_data : 附件的错误诊断信息（由平台自定义，可能为 NULL）
* \param[in] size : 错误诊断信息的长度（由平台自定义，可能为 0）
*
* \return AM_OK
*/
typedef int (*am_uart_err_t)(void *p_arg, int code, void *p_data, int size);

/** @} */

/**
* \brief UART 驱动函数结构体
*/
struct am_uart_drv_funcs {

    /**\brief UART 控制函数 */
    int (*pfn_uart_ioctl)(void *p_drv, int request, void *p_arg);

    /**\brief 启动 UART 发送函数 */
    int (*pfn_uart_tx_startup)(void *p_drv);

    /**\brief 设置串口回调函数 */
    int (*pfn_uart_callback_set)(void *p_drv,
                                int callback_type,
                                void *pfn_callback,
                                void *p_arg);

    /**\brief 从串口获取一个字符（查询模式） */
    int (*pfn_uart_poll_getchar)(void *p_drv, char *p_inchar);

    /**\brief 输出一个字符（查询模式） */
    int (*pfn_uart_poll_putchar)(void *p_drv, char outchar);

};

/**
* \brief UART 服务
*/
typedef struct am_uart_serv {

    /** \brief UART 驱动函数结构体指针 */
    struct am_uart_drv_funcs *p_funcs;

    /** \brief 用于驱动函数的第一个参数 */
    void *p_drv;

```

```

} am_uart_serv_t;

/** \brief UART 标准服务操作句柄类型定义 */
typedef am_uart_serv_t *am_uart_handle_t;

```

1. UART 宏定义说明

标准文件中定义了诸多宏，包含 UART 的模式选择、通用指令设置、硬件参数设置、流控制选项、回调函数、错误类型码，程序注释中每个宏都有对应的解释说明，这里不一一赘述；

2. 回调函数类型说明

标准文件中定义了 3 个 UART 的回调函数，分别是接受、发送、错误回调函数，均用于中断模式下；

3. struct am_uart_drv_funcs 结构体

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层需要实现 5 个函数：

- UART 控制函数 pfn_uart_ioctl，用于控制 UART，修改 UART 的参数，包括波特率、硬件参数、设置模式、流控方式等等，可调参数可看 UART 的通用控制指令的宏及说明；
- UART 启动发送函数 pfn_uart_tx_startup，用于 UART 发送；
- UART 的串口回调函数 pfn_uart_callback_set；
- 查询模式读取字符函数 pfn_uart_poll_getchar；
- 查询模式输出字符 pfn_uart_poll_putchar；

因此，在驱动层编写函数时，应该向标准接口层提供这 5 个函数。

这些函数的有一个共同点，就是第一个参数都是 void 型的指针，也就是说，驱动层提供的函数，标准接口层使用时传递的第一个参数并无确定类型，标准接口层不关心第一个参数的具体类型，使用时只负责传递就可以了。

4. typedef struct am_uart_serv 结构体

该结构体定义了标准的 UART 服务，包含了上文提到的驱动函数结构体的指针和一个 void 型的指针。这个标准的 UART 服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

● p_funcs 成员赋值

这是一个指向 typedef struct am_uart_serv 结构体的指针，我们只需要在驱动层定义一个 struct am_uart_drv_funcs 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那 5 个函数地址。然后，将 struct am_uart_drv_funcs 结构体变量的地址赋值给 p_funcs 成员即可。

● p_drv 成员赋值

这个变量是 void 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动自行决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

5. am_uart_handle_t 的定义

这个是一个标准的操作句柄，本质上就是 `typedef am_uart_serv_t` 指针类型，驱动层的初始化函数需要返回该类型的一个 `handle` 给用户。这样，用户获取到 `handle`，就可以在需要使用标准接口层函数时，将该 `handle` 传递给标准接口层了。标准接口就可以通过该 `handle`（`typedef am_uart_serv_t` 结构体指针）访问到驱动层提供的函数了。标准接口层怎么使用 `handle` 指针？看一下标准接口层的函数实现就明白了。

10.1.2 了解标准接口函数

在 `am_uart.h` 文件相关的结构体、类型定义后，就是具体函数的实现，如程序清单 10.2 所示。

程序清单 10.2 UART 标准接口函数

```
/**
 * \brief 串口控制函数
 *
 * \param[in] handle : UART 标准服务操作句柄
 * \param[in] request : 控制指令
 *   - AM_UART_BAUD_SET : 设置波特率, p_arg 为 uint32_t 类型, 值为波特率
 *   - AM_UART_BAUD_GET : 获取波特率, p_arg 为 uint32_t 指针类型
 *   - AM_UART_OPTS_SET : 设置硬件参数, p_arg 为 uint32_t 类型 (#AM_UART_CS8)
 *   - AM_UART_OPTS_GET : 获取当前的硬件参数设置, p_arg 为 uint32_t 指针类型
 *   - AM_UART_MODE_SET : 设置模式, p_arg 值为 AM_UART_MODE_POLL 或
AM_UART_MODE_INT
 *   - AM_UART_MODE_GET : 获取当前模式, p_arg 为 uint32_t 指针类型
 *   - AM_UART_AVAIL_MODES_GET : 获取当前可用的模式, p_arg 为 uint32_t 指针类型
 *   - AM_UART_FLOWMODE_SET : 设置流控模式, p_arg 为 AM_UART_FLOWCTL_NO
 *                               或 AM_UART_FLOWCTL_OFF
 *   - AM_UART_FLOWSTAT_RX_SET : 设置接收器流控状态, p_arg 为 AM_UART_FLOWSTAT_ON
 *                               或 AM_UART_FLOWSTAT_OFF
 *   - AM_UART_FLOWSTAT_TX_GET : 获取发送器流控状态, p_arg 为 AM_UART_FLOWSTAT_ON
 *                               或 AM_UART_FLOWSTAT_OFF
 *
 *   - AM_UART_RS485_SET : 设置 RS485 模式, p_arg 为 bool_t 类型, TURE (使能), FALSE (禁能)
 *   - AM_UART_RS485_GET : 获取当前的 RS485 模式状态, 参数为 bool_t 指针类型
 *
 * \param[in,out] p_arg : 该指令对应的参数
 *
 * \retval AM_OK : 控制指令执行成功
 * \retval -AM_EIO : 执行错误
 * \retval -AM_ENOTSUP : 指令不支持
 */
am_static_inline
int am_uart_ioctl(am_uart_handle_t handle, int request, void *p_arg)
{
    return handle->p_funcs->pfn_uart_ioctl(handle->p_drv, request, p_arg);
}
```

```

/**
 * \brief 启动 UART 中断模式数据传输
 * \param[in] handle : UART 标准服务操作句柄
 * \retval AM_OK : 启动成功
 */
am_static_inline
int am_uart_tx_startup (am_uart_handle_t handle)
{
    return handle->p_funcs->pfn_uart_tx_startup(handle->p_drv);
}

/**
 * \brief 设置 UART 回调函数
 *
 * \param[in] handle          : UART 标准服务操作句柄
 * \param[in] callback_type   : 指明设置的何种回调函数
 *
 *      - AM_UART_CALLBACK_GET_TX_CHAR : 获取一个发送字符函数
 *      - AM_UART_CALLBACK_PUT_RCV_CHAR : 提交一个接收到的字符给应用程序
 *      - AM_UART_CALLBACK_ERROR       : 错误回调函数
 * \param[in] pfn_callback    : 指向回调函数的指针
 * \param[in] p_arg           : 回调函数的用户参数
 *
 * \retval  AM_OK           : 回调函数设置成功
 * \retval -AM_EINVAL      : 设置失败, 参数错误
 */
am_static_inline
int am_uart_callback_set (am_uart_handle_t  handle,
                          int                callback_type,
                          void               *pfn_callback,
                          void               *p_arg)
{
    return handle->p_funcs->pfn_uart_callback_set(handle->p_drv,
                                                  callback_type,
                                                  pfn_callback,
                                                  p_arg);
}

/**
 * \brief UART 接收一个数据(查询模式)
 *
 * \param[in]  handle      : UART 标准服务操作句柄
 * \param[out] p_inchar    : 用于获取数据的指针
 *
 * \retval  AM_OK          : 接收数据成功

```

```

* \retval -AM_EAGAIN : 接收数据未准备就绪，需重试
*/
am_static_inline
int am_uart_poll_getchar (am_uart_handle_t handle, char *p_inchar)
{
    return handle->p_funcs->pfn_uart_poll_getchar(handle->p_drv, p_inchar);
}
/**
* \brief UART 发送一个数据(查询模式)
*
* \param[in] handle : UART 标准服务操作句柄
* \param[in] outchar : 待发送的数据
*
* \retval AM_OK : 发送数据成功
* \retval -AM_EAGAIN : 发送未就绪，需重试
*/
am_static_inline
int am_uart_poll_putchar (am_uart_handle_t handle, char outchar)
{
    return handle->p_funcs->pfn_uart_poll_putchar(handle->p_drv, outchar);
}

```

很简单的 5 个函数全都是用内联函数就搞定了，只是简单的调用了一下驱动层提供的函数而已。可以看到，在调用驱动层提供的函数时，就是将 struct am_uart_serv 结构体中的 p_drv 成员作为第一个参数传递。

除去这几个内联函数，我们可以看到下面还有两个函数，但只是做了声明，详见程序清单 10.3。

程序清单 10.3 UART 标准接口函数

```

/**
* \brief UART 数据发送（查询模式）
*
* \param[in] handle : UART 标准服务操作句柄
* \param[in] p_txbuf : 发送数据缓冲区
* \param[in] nbytes : 待发送数据的个数
*
* \return 成功发送数据的个数
*/
int am_uart_poll_send(am_uart_handle_t handle,
                     const uint8_t *p_txbuf,
                     uint32_t nbytes);
/**
* \brief UART 数据接收（查询模式）

```

```

*
* \param[in] handle : UART 标准服务操作句柄
* \param[in] p_rxbuf : 接收数据缓冲区
* \param[in] nbytes : 待接收数据的个数
*
* \return 成功接收数据的个数
*/
int am_uart_poll_receive(am_uart_handle_t handle,
                        uint8_t *p_rxbuf,
                        uint32_t nbytes);

```

可以看到这里是声明了几个函数，并非简单内联函数，这部分需要由驱动层来实现，这里提供函数声明供调用。

到此我们就分析完了 UART 外设标准层文件，清楚了驱动层需要为标准接口层提供哪些内容。

10.2 HW 层开发

现在我们打开 ZMF2x6 芯片用户手册，对照手册完成 UART 外设 HW 层的开发。打开 ametal\soc\zlg\drivers\include\uart\hw 路径，在该路径下创建名为 amhw_zmf2x6_uart.h 的文件，初始内容参考程序清单 5.3。

10.2.1 添加寄存器列表

HW 层就是与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读该外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMF2x6 用户手册的 UART 章节，可以看到，UART 章节部分，共有 15 个寄存器，简单描述详见表 10.1。

表 10.1 UART 寄存器概览

位	标识符	含义
0	UART_TDR	发送数据寄存器
1	UART_RDR	接收数据寄存器
2	UART_CSR	当前状态寄存器
3	UART_ISR	中断状态寄存器
4	UART_IER	中断使能寄存器
5	UART_ICR	中断清除寄存器
6	UART_GCR	全局控制寄存器
7	UART_CCR	通用控制寄存器
8	UART_BRR	波特率设置寄存器
9	UART_FRA	波特率小数寄存器
10	UART_RXADDR	接收地址寄存器，
11	UART_RXMASK	接收同步中断请求寄存器
12	UART_SCR	ISO7816 相关配置寄存器
13	UART_IDLR	数据长度寄存器

14	UART_ABRCR	自动波特率寄存器

到此 UART 的各个寄存器及其功能基本阅读完毕，我们也大致了解了各个寄存器功能以及要配置各个功能，其他寄存器该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，程序清单如程序清单 10.4 所示。

程序清单 10.4 寄存器结构体定义

```
/**
 * \brief uart 寄存器块结构体
 */
typedef struct amhw_zmf2x6_uart {
    __IO uint32_t tdr;      /**< \brief 发送数据寄存器 */
    __IO uint32_t rdr;      /**< \brief 接收数据寄存器 */
    __IO uint32_t csr;      /**< \brief 当前状态寄存器 */
    __IO uint32_t isr;      /**< \brief 中断状态寄存器 */
    __IO uint32_t ier;      /**< \brief 中断使能寄存器 */
    __IO uint32_t icer;     /**< \brief 中断清除寄存器 */
    __IO uint32_t gcr;      /**< \brief 全局控制寄存器 */
    __IO uint32_t ccr;      /**< \brief 通用控制寄存器 */
    __IO uint32_t brr;      /**< \brief 波特率寄存器 */
    __IO uint32_t fra;      /**< \brief 分数波特率寄存器 */
    __IO uint32_t rxaddr;   /**< \brief RXADDR 寄存器 */
    __IO uint32_t rxmask;   /**< \brief RXMASK 寄存器 */
    __IO uint32_t scr;      /**< \brief SCR 寄存器 */
    __IO uint32_t idlr;     /**< \brief IDLE 数据寄存器 */
    __IO uint32_t abrcr;    /**< \brief 自动波特率控制寄存器 */
    __IO uint32_t irda;     /**< \brief IRDA 红外功能控制寄存器 */
} amhw_zmf2x6_uart_t;
```

这个小节我们定义了 UART 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

10.2.2 编写寄存器的相关函数

1. UART_TDR（发送数据寄存器）

通过用户手册，我们了解该寄存器的发送位由 8 位组成。因此它的操作函数也较为简单，操作函数如程序清单 10.5 所示。

程序清单 10.5 UART_TDR 操作函数

```
/**
 * \brief Write data to Transmit data register
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] data_w     : data to write
 */
```



```

* \return none
*/
am_static_inline
void amhw_zmf2x6_uart_data_write (amhw_zmf2x6_uart_t *p_hw_uart,
                                  uint8_t data_w)
{
    p_hw_uart->tdr = (uint32_t)(0xff & data_w);
}

```

2. UART_RDR（接收数据寄存器）

通过用户手册，我们了解该寄存器的接收数据位由 8 位组成。因此它的操作函数也较为简单，操作函数如程序清单 10.6 所示。

程序清单 10.6 UART_RDR 操作函数

```

/**
 * \brief read data form receive data register
 *
 * \param[in] p_hw_uart : pointer to UART register block
 *
 * \return the data of RDR register
 */
am_static_inline
uint8_t amhw_zmf2x6_uart_data_read (amhw_zmf2x6_uart_t *p_hw_uart)
{
    return (uint8_t)(0xffu & p_hw_uart->rdr);
}

```

3. UART_CSR（控制寄存器）

通过用户手册，我们了解该寄存器有 4 种状态可查询。我们先定义一下几种状态，再实现函数，操作函数如程序清单 10.7 所示。

程序清单 10.7 UART_CSR 控制函数

```

/**
 * \name Status flag
 * \anchor amhw_zmf2x6_uart_status_flag_t
 * @{
 */
typedef enum amhw_zmf2x6_uart_status_flag {
    AMHW_ZMF2X6_UART_TX_EMPTY_FLAG = AM_BIT(3), /* Transmit buffer empty flag */
    AMHW_ZMF2X6_UART_TX_FULL_FLAG = AM_BIT(2), /* Transmit buffer full flag */
    AMHW_ZMF2X6_UART_RX_VAL_FLAG = AM_BIT(1), /* Receive valid data flag */
    AMHW_ZMF2X6_UART_TX_COMPLETE_FALG = AM_BIT(0) /* Transmit complete flag */
} amhw_zmf2x6_uart_status_flag_t;

/** @} */

```

```

/**
 * \brief get the value of status register
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] flag      : status flag  see the filed of  Status flag
 *
 * \retval TRUE or FALSE
 */
am_static_inline
am_bool_t amhw_zmf2x6_uart_status_flag_check(amhw_zmf2x6_uart_t *p_hw_uart,
                                              amhw_zmf2x6_uart_status_flag_t flag)
{
    return ((0xfu & p_hw_uart->csr) & flag) ? AM_TRUE : AM_FALSE;
}

```

4. UART_ISR（中断状态寄存器）

通过用户手册，我们了解该寄存器有 12 种状态可查询。我们先定义一下几种状态，再实现函数，操作函数如程序清单 10.8 所示。

程序清单 10.8 UART_CSR 控制函数

```

/**
 * \name  Interrupt flag
 * \anchor amhw_zmf2x6_uart_int_flag_t
 * @{
 */
typedef enum amhw_zmf2x6_uart_int_flag {
    AMHW_ZMF2X6_UART_INT_RXBRK_FLAG    = AM_BIT(6),/* Receive  break frame interrupt flag
*/
    AMHW_ZMF2X6_UART_INT_RXFERR_FLAG   = AM_BIT(5),/* Frame error interrupt flag */
    AMHW_ZMF2X6_UART_INT_RXPERR_FLAG   = AM_BIT(4),/* Parity error interrupt flag */
    AMHW_ZMF2X6_UART_INT_RXOERR_FLAG   = AM_BIT(3),/* Receive overflow error interrupt flag
*/
    AMHW_ZMF2X6_UART_INT_RX_VAL_FLAG   = AM_BIT(1),/* Receive valid interrupt flag */
    AMHW_ZMF2X6_UART_INT_TX_EMPTY_FLAG = AM_BIT(0) /* Transmit buffer empty interrupt flag
*/
} amhw_zmf2x6_uart_int_flag_t;
/** @} */

/**
 * \brief get the value of interrupt status register
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] flag      : interrupt flag  see the filed of interrupt flag
 * \return TRUE or FALSE
 */

```

```

am_static_inline
am_bool_t amhw_zmf2x6_uart_int_flag_check (amhw_zmf2x6_uart_t      *p_hw_uart,
                                             amhw_zmf2x6_uart_int_flag_t flag)
{
    return ((0x7bu & p_hw_uart->isr) & flag) ? AM_TRUE : AM_FALSE;
}

```

5. UART_IER（中断使能寄存器）

通过用户手册，我们了解该寄存器有 12 种状态可配置。我们先定义一下几种状态，再实现函数，操作函数如程序清单 10.9 所示。

程序清单 10.9 UART_IER 控制函数

```

/**
 * \name Interrupt enable
 * \anchor amhw_zmf2x6_uart_int_enable_t
 * @{
 */

typedef enum amhw_zmf2x6_uart_int_enable {
    AMHW_ZMF2X6_UART_INT_RXBRK_ENABLE = AM_BIT(6),/* Receive break frame interrupt
enable */
    AMHW_ZMF2X6_UART_INT_RXFERR_ENABLE = AM_BIT(5),/* Frame error interrupt enable */
    AMHW_ZMF2X6_UART_INT_RXPERR_ENABLE = AM_BIT(4),/* Parity error interrupt enable */
    AMHW_ZMF2X6_UART_INT_RXOERR_ENABLE = AM_BIT(3),/* Receive overflow error interrupt
enable */
    AMHW_ZMF2X6_UART_INT_RX_VAL_ENABLE = AM_BIT(1),/* Receive valid interrupt enable */
    AMHW_ZMF2X6_UART_INT_TX_EMPTY_ENABLE = AM_BIT(0),/* Transmit buffer empty interrupt
enable */
    AMHW_ZMF2X6_UART_INT_ALL_ENABLE_MASK = 0x7F /* All interrupt mask */
} amhw_zmf2x6_uart_int_enable_t;

/** @} */

/**
 * \brief Enable interrupt
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] flag : interrupt enable see the filed of interrupt enable
 * \return none
 */
am_static_inline
void amhw_zmf2x6_uart_int_enable (amhw_zmf2x6_uart_t      *p_hw_uart,
                                   amhw_zmf2x6_uart_int_enable_t flag)
{
    p_hw_uart->ier = (p_hw_uart->ier & ~(flag)) | flag;
}

```

6. UART_ICR（中断清除寄存器）

通过用户手册，我们了解该寄存器有 12 种状态可配置。我们先定义一下几种状态，再实现函数，操作函数如程序清单 10.10 所示。

程序清单 10.10 UART_ICR 控制函数

```
/**
 * \name Interrupt flag clear
 * \anchor amhw_zmf2x6_uart_int_flag_clr_t
 * @{
 */

typedef enum amhw_zmf2x6_uart_int_flag_clr {
    AMHW_ZMF2X6_UART_INT_RXBRK_FLAG_CLR = AM_BIT(6),/* Receive break frame interrupt
flag */
    AMHW_ZMF2X6_UART_INT_RXFERR_FLAG_CLR = AM_BIT(5),/* Frame error interrupt flag */
    AMHW_ZMF2X6_UART_INT_RXPERR_FLAG_CLR = AM_BIT(4),/* Parity error interrupt flag */
    AMHW_ZMF2X6_UART_INT_RXOERR_FLAG_CLR = AM_BIT(3),/* Receive overflow error
interrupt flag */
    AMHW_ZMF2X6_UART_INT_RX_VAL_FLAG_CLR = AM_BIT(1),/* Receive valid interrupt flag */
    AMHW_ZMF2X6_UART_INT_TX_EMPTY_FLAG_CLR = AM_BIT(0) /* Transmit buffer empty interrupt
flag */
} amhw_zmf2x6_uart_int_flag_clr_t;
/** @} */

/**
 * \brief Clear interrupt flag
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] flag : interrupt flag see the filed of interrupt flag clear
 * \return none
 */
am_static_inline
void amhw_zmf2x6_uart_int_flag_clr (amhw_zmf2x6_uart_t *p_hw_uart,
amhw_zmf2x6_uart_int_flag_clr_t flag)
{
    p_hw_uart->icr = 0x7fu & flag;
}
```

7. UART_GCR（全局控制寄存器）

通过用户手册，我们了解该寄存器有 11 种状态可配置，但主要实现功能如表 10.2，该部分功能需要独立函数实现。

表 10.2 UART_GCR 寄存器分析

位	标识符	含义

0	GCR_UARTEN	UART 模块选择位
1	GCR_DMAMODE	DMA 模式选择位
2	GCR_FLOWEN	自动流控使能位
3	GCR_RXEN	接收使能位
4	GCR_TXEN	发送使能位

● GCR_TXEN 操作函数

该位写 1 使能发送功能，写 0 禁止该功能，函数实现如程序清单 10.11 所示。

程序清单 10.11 GCR_TXEN 操作函数

```
/**
 * \brief Transmit enable
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] flag      : TURE or FALSE
 * \return none
 */
am_static_inline
void amhw_zmf2x6_uart_tx_enable (amhw_zmf2x6_uart_t *p_hw_uart, am_bool_t flag)
{
    p_hw_uart->gcr = (p_hw_uart->gcr & ~(1u << 4)) | (flag << 4);
}
```

● GCR_RXEN 操作函数

该位写 1 使能接收功能，写 0 禁能，函数实现如程序清单 10.12 所示。

程序清单 10.12 GCR_RXEN 操作函数

```
/**
 * \brief Receive enable
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] flag      : TURE or FALSE
 * \return none
 */
am_static_inline
void amhw_zmf2x6_uart_rx_enable (amhw_zmf2x6_uart_t *p_hw_uart, am_bool_t flag)
{
    p_hw_uart->gcr = (p_hw_uart->gcr & ~(1u << 3)) | (flag << 3);
}
```

● GCR_FLOWEN 操作函数

该位写 1 使能流控功能，写 0 禁能，函数实现如程序清单 10.13 所示。

程序清单 10.13 GCR_FLOWEN 操作函数

```
/**
 * \brief Automatic flow control enable
```

```

*
* \param[in] p_hw_uart : pointer to UART register block
* \param[in] flag      : TURE or FALSE
* \return none
*/
am_static_inline
void amhw_zmf2x6_uart_auto_flow_enable (amhw_zmf2x6_uart_t *p_hw_uart,
                                         am_bool_t          flag)
{
    p_hw_uart->gcr = (p_hw_uart->gcr & ~(1u << 2)) | (flag << 2);
}

```

- GCR_MDAMODE 操作函数

该位写 1 使能 DMA 功能，写 0 禁能，函数实现如程序清单 10.14 所示。

程序清单 10.14 GCR_DMAMODE 操作函数

```

/**
* \brief DMA mode enable
*
* \param[in] p_hw_uart : pointer to UART register block
* \param[in] flag      : TURE or FALSE
* \return none
*/
am_static_inline
void amhw_zmf2x6_uart_dma_mode_enable (amhw_zmf2x6_uart_t *p_hw_uart,
                                         am_bool_t          flag)
{
    p_hw_uart->gcr = (p_hw_uart->gcr & ~(1u << 1)) | (flag << 1);
}

```

- GCR_UARTEN 操作函数

该位写 1 使能 UART 模块功能，写 0 禁能，函数实现如程序清单 10.15 所示。

程序清单 10.15 GCR_UARTEN 操作函数

```

/**
* \brief UART modules enable
*
* \param[in] p_hw_uart : pointer to UART register block
* \return none
*/
am_static_inline
void amhw_zmf2x6_uart_enable (amhw_zmf2x6_uart_t *p_hw_uart)
{
    p_hw_uart->gcr |= (1u << 0);
}

```

```

/**
 * \brief UART modules disable
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \return none
 */
am_static_inline
void amhw_zmf2x6_uart_disable (amhw_zmf2x6_uart_t *p_hw_uart)
{
    p_hw_uart->gcr &= ~(1u << 0);
}
}

```

8. UART_CCR（通用控制寄存器）

通过用户手册，我们了解该寄存器有 16 种状态可配置，但主要实现功能如表 10.2，该部分功能需要独立函数实现。

表 10.3 UART_CCR 寄存器分析

位	标识符	含义
5: 4	CCR_CHAR	数据位宽
3	CCR_BRK	发送断开帧
2	CCR_SPB	停止位选择
1	CCR_PSEL	校验位选择
0	CCR_PEN	校验使能位

● CCR_CHAR 操作函数

由两个位组成，用于选择数据宽度，4 种数据位选择，先定义数据宽度类型，函数实现如程序清单 10.16 所示。

程序清单 10.16 CCR_CHAR 操作函数

```

/**
 * \brief UART data length
 * @{
 */
typedef enum amhw_zmf2x6_uart_data_length {
    AMHW_ZMF2X6_UART_DATA_5BIT = (0u << 4),    /** <brie 5-bit data */
    AMHW_ZMF2X6_UART_DATA_6BIT = (1u << 4),    /** <brie 6-bit data */
    AMHW_ZMF2X6_UART_DATA_7BIT = (2u << 4),    /** <brie 7-bit data */
    AMHW_ZMF2X6_UART_DATA_8BIT = (3u << 4)     /** <brie 8-bit data */
} amhw_zmf2x6_uart_data_length_t;
/** @} */

/**
 * \brief UART data data length select

```

```

*
* \param[in] p_hw_uart : pointer to UART register block
* \param[in] data_len  : see the filed of  data length
*
*                          such as AMHW_ZMF2X6_UART_DATA_
* \return none
*/
am_static_inline
void amhw_zmf2x6_uart_data_length (amhw_zmf2x6_uart_t *p_hw_uart,
                                   amhw_zmf2x6_uart_data_length_t data_len)
{
    p_hw_uart->ccr = (p_hw_uart->ccr & ~(3u << 4)) | data_len;
}

```

● CCR_BRK 操作函数

设置 UART 的发送断开帧设置寄存器，函数实现如程序清单 10.17 所示。

程序清单 10.17 CCR_BRK 操作函数

```

/**
* \brief Transmit break frame
*
* \param[in] p_hw_uart : pointer to UART register block
* \param[in] flag      : TURE or FALSE
* \return none
*/
am_static_inline
void amhw_zmf2x6_uart_send_break (amhw_zmf2x6_uart_t *p_hw_uart, am_bool_t flag)
{
    p_hw_uart->ccr = (p_hw_uart->gcr & ~(1u << 3)) | (flag << 3);
}

```

● CCR_PSEL 操作函数

设置 UART 的停止位选择寄存器，函数实现如程序清单 10.18 所示。

程序清单 10.18 CCR_PSEL 操作函数

```

/**
* \brief UART stop bit select
* @{}
*/
typedef enum amhw_zmf2x6_uart_stop_bit {
    AMHW_ZMF2X6_UART_STOP_1BIT = (0u << 2), /** <brief One stop bit */
    AMHW_ZMF2X6_UART_STOP_2BIT = (1U << 2)  /** <brief two stop bit */
} amhw_zmf2x6_uart_stop_bit_t;
/** @} */

/**
* \brief UART stop bit select

```



```

*
* \param[in] p_hw_uart : pointer to UART register block
* \param[in] stop_bit : see the filed of stop bit select
*
* such as AMHW_ZMF2X6_UART_SIOP_*
* \return none
*/
am_static_inline
void amhw_zmf2x6_uart_stop_bit_sel (amhw_zmf2x6_uart_t *p_hw_uart,
                                     amhw_zmf2x6_uart_stop_bit_t stop_bit)
{
    p_hw_uart->ccr = (p_hw_uart->ccr & ~(1u << 2)) | stop_bit;
}

```

● CCR_SPB 操作函数

设置 UART 的校验方式选择寄存器，函数实现如程序清单 10.19 所示。

程序清单 10.19 CCR_SPB 操作函数

```

/**
* \brief UART parity bit select
* @{
*/
typedef enum amhw_zmf2x6_uart_parity_bit {
    AMHW_ZMF2X6_UART_PARITY_NO    = (0U),  /** <brief No parity */
    AMHW_ZMF2X6_UART_PARITY_ODD   = (1U),  /** <brief Odd parity */
    AMHW_ZMF2X6_UART_PARITY_EVEN  = (3U)   /** <brief Even parity */
} amhw_zmf2x6_uart_parity_bit_t;
/** @} */

/**
* \brief UART parity bit select
*
* \param[in] p_hw_uart : pointer to UART register block
* \param[in] parity : see the filed of parity bit select
*
* such as AMHW_ZMF2X6_UART_PARITY_*
* \return none
*/
am_static_inline
void amhw_zmf2x6_uart_parity_bit_sel (amhw_zmf2x6_uart_t *p_hw_uart,
                                       amhw_zmf2x6_uart_parity_bit_t parity)
{
    p_hw_uart->ccr = (p_hw_uart->ccr & ~(3u << 0)) | parity;
}

```

9. UART_BRR（波特率寄存器）

通过用户手册，我们了解该寄存器用于设置波特率的整数部分，函数实现如程序清单 10.20。

程序清单 10.20 UART_BRR 操作函数

```

/**
 * \brief baud rate divide factor mantissa set
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] value      : divide factor mantissa
 * \return none
 */
am_static_inline
void amhw_zmf2x6_uart_br_div_mantissa_set (amhw_zmf2x6_uart_t *p_hw_uart,
                                             uint16_t          value)
{
    p_hw_uart->brr = (0xffff & value);
}

```

10. UART_FRA（波特率分数寄存器）

通过用户手册，我们了解该寄存器用于设置波特率的分数部分，函数实现如程序清单 10.21。

程序清单 10.21 UART_FRA 操作函数

```

/**
 * \brief baud rate divide factor fraction set
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] value      : divide factor fraction
 * \return none
 */
am_static_inline
void amhw_zmf2x6_uart_br_div_fraction_set (amhw_zmf2x6_uart_t *p_hw_uart,
                                             uint8_t             value)
{
    p_hw_uart->fra = (0xf & value);
}

```

11. UART_RXADDR（接收地址寄存器）

通过用户手册，我们了解该寄存器用于设置同步帧数据本机匹配地址，函数实现如程序清单 10.22。

程序清单 10.22 UART_RXADDR 操作函数

```

/**
 * \brief UART 同步帧数据本机匹配地址获取
 *
 * \param[in] p_hw_uart : 寄存器基地址
 *

```

```

* \return 匹配地址
*/
am_static_inline
uint8_t amhw_zmf2x6_uart_rxaddr_get (amhw_zmf2x6_uart_t *p_hw_uart)
{
    return (p_hw_uart->rxaddr & 0xff);
}

```

10.2.3 添加其他功能性函数

我们回顾一下所有的函数,虽然对所有寄存器都提供了操作函数,但还欠缺波特率设置、串口收发的函数,这些函数在标准层也有提到,但依靠单一的内联函数,显然不能直接完成该函数功能,我们可以考虑在 c 文件中实现这些函数,并在该文件添加函数声明。

参考建立 h 文件的要求,我们在 ametal\soc\zlg\drivers\source\uart\hw 路径下创建 amhw_zmf2x6_uart.c 文件。

接下来,让我们来实现所需的 3 个函数,串口的收发和波特率设置函数。

- 1) 串口发送函数,需要传入的参数主要有要发送的数据和数据长度,实现如下程序清单 10.23:

程序清单 10.23 串口发送操作函数

```

/**
 * \brief UART 数据发送(查询模式)
 */
uint32_t amhw_zmf2x6_uart_poll_send (amhw_zmf2x6_uart_t *p_hw_uart,
                                     const uint8_t *p_txbuf,
                                     uint32_t nbytes)
{
    uint32_t len = nbytes;

    while (len--) {

        while(amhw_zmf2x6_uart_status_flag_check(
                                                    p_hw_uart,
                                                    AMHW_ZMF2X6_UART_TX_EMPTY_FLAG)
            == AM_FALSE);

        amhw_zmf2x6_uart_data_write(p_hw_uart, (uint8_t)(*p_txbuf++));
    }

    return nbytes;
}

```

- 2) 串口接收函数,需要传入的参数主要有要读取的数据和数据长度,实现如下程序清单 10.24:

程序清单 10.24 串口读取数据操作函数

```

/**

```

```

* \brief UART 数据接收(查询模式)
*/
uint32_t amhw_zmf2x6_uart_poll_receive (amhw_zmf2x6_uart_t *p_hw_uart,
                                         uint8_t          *p_rxbuf,
                                         uint32_t          nbytes)
{
    uint32_t len = nbytes;

    while (len--) {
        while(amhw_zmf2x6_uart_status_flag_check(p_hw_uart,
                                                  AMHW_ZMF2X6_UART_RX_VAL_FLAG)
              == AM_FALSE);

        *p_rxbuf++ = (uint8_t)amhw_zmf2x6_uart_data_read(p_hw_uart);
    }

    return nbytes;
}

```

- 3) 波特率设置函数，需要传入的参数主要有串口时钟和波特率参数，实现如程序清单 10.25:

程序清单 10.25 串口波特率设置函数

```

/**
* \brief UART baud ratio set
*/
int amhw_zmf2x6_uart_baudrate_set (amhw_zmf2x6_uart_t *p_hw_uart,
                                   uint32_t          uart_clk,
                                   uint32_t          baud)
{
    uint32_t int_val = 0;
    uint32_t deci_val = 0;
    /* 确保有效的 clock value */
    if ((uart_clk > 72000000)) {
        uart_clk = 0;
        /* 初始化异常，由于期望的系统时钟值无效 */
        return AM_ERROR;
    }
    int_val = (uart_clk / baud) / 16;
    deci_val = (uart_clk / baud) % 16;
    amhw_zmf2x6_uart_br_div_mantissa_set(p_hw_uart, int_val);
    amhw_zmf2x6_uart_br_div_fraction_set(p_hw_uart, deci_val);

    /* FRA 寄存器需要重新使能 UART 才能生效 */
    if (amhw_zmf2x6_uart_uarten_get(p_hw_uart)) {
        amhw_zmf2x6_uart_disable(p_hw_uart);
    }
}

```

```

        amhw_zmf2x6_uart_enable(p_hw_uart);
    }
    return baud;
}

```

到此，我们已添加完所需的 3 个函数，我们需要在对应的.h 文件添加函数声明，方便驱动层调用，如程序清单 10.26 所示。

程序清单 10.26 串口相关函数声明

```

/**
 * \brief baud rate set
 *
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] uart_clk  : the clock of UART module
 * \param[in] baud      : baud rate
 * \return baud rate
 */
int amhw_zmf2x6_uart_baudrate_set (amhw_zmf2x6_uart_t *p_hw_uart,
                                   uint32_t          uart_clk,
                                   uint32_t          baud);

/**
 * \brief UART receive data(polling mode)
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] p_rxbuf   : the pointer buffer
 * \param[in] nbytes    : bytes
 *
 * \return bytes
 */
uint32_t amhw_zmf2x6_uart_poll_receive (amhw_zmf2x6_uart_t *p_hw_uart,
                                         uint8_t            *p_rxbuf,
                                         uint32_t            nbytes);

/**
 * \brief UART transfer data (polling mode)
 * \param[in] p_hw_uart : pointer to UART register block
 * \param[in] p_txbuf   : the pointer to buffer
 * \param[in] nbytes    : bytes
 *
 * \return bytes
 */
uint32_t amhw_zmf2x6_uart_poll_send (amhw_zmf2x6_uart_t *p_hw_uart,
                                     const uint8_t        *p_txbuf,
                                     uint32_t              nbytes);

```

10.2.4 添加指向寄存器块的指针

可以看到，在 HW 层编写的函数中，第一个参数均为指向 UART 寄存器块的指针，只

需要把 UART 的基地址使用(amhw_zmf2x6_uart_t*)强制转换即可作为这些函数的第一个参数。这样转换后，操作结构体的成员，实际上就是操作具体地址上的寄存器了。

在 AMetal 中，由于这些内容与具体型号的芯片相关，这样的宏统一定义到 zmf2x6_periph_map.h 文件中，如程序清单 5.14 所示。

程序清单 10.27 外设中加入 UART 的定义

```
/** \brief 串口(UART1)寄存器块指针 */
#define ZMF2X6_UART1 ((amhw_zmf2x6_uart_t *)ZMF2X6_UART1_BASE)

/** \brief 串口(UART2)寄存器块指针 */
#define ZMF2X6_UART2 ((amhw_zmf2x6_uart_t *)ZMF2X6_UART2_BASE)
```

后续用户或驱动层使用 HW 层接口时，第一个参数直接使用 ZMF2X6_UARTx 即可。至此，整个 HW 层的开发工作就结束了，如果后续开发中发现缺少某些函数，还可以再添加。

10.3 驱动层开发

参考 5.3.1 章节，新建 am_zmf2x6_uart.h 和 am_zmf2x6_uart.c 驱动文件，搭建相应的文件框架，并按规则放到对应的路径下。

10.3.1 编写结构性程序

驱动层文件有两个，am_zmf2x6_uart.c 和 am_zmf2x6_uart.h。存放路径分别为 ametal\soc\zlg\drivers\source\uart 和 ametal\soc\zlg\drivers\include\uart。驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备（UART 就是一个设备）都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

可以暂时定义设备信息结构体如程序清单 10.28 所示。后续可以根据实际情况修改。

程序清单 10.28 UART 设备信息结构体定义

```
/**
 * \brief 串口设备信息结构体，该设备信息用于串口初始化
 */
typedef struct am_zmf2x6_uart_devinfo {

    uint32_t uart_reg_base;    /**< \brief 指向 UART 寄存器块的指针 */

    uint8_t inum;              /**< \brief 串口中断号 */
```

```

int      clk_num;          /**< \brief 时钟 ID */

/**
 * \brief 串口配置标志, AMHW_ZMF2X6_UART_DATA_*宏或 AMHW_ZMF2X6_UART_PARITY_*
宏
 *      或 AMHW_ZMF2X6_UART_STOP_*的或值
 *      (如: AMHW_ZMF2X6_UART_DATA_8BIT|AMHW_ZMF2X6_UART_PARITY_NO
 *              |AMHW_ZMF2X6_UART_STOP_1BIT)
 */
uint8_t  cfg_flags;

uint32_t baud_rate;        /**< \brief 初始化波特率 */

/**
 * \brief 指定使能的其它中断,AMHW_ZMF2X6_UART_INT_*宏值或多个
AMHW_ZMF2X6_UART_INT_*宏的
 *      或值, 除了(# AMHW_ZMF2X6_UART_INT_TX_EMPTY_ENABLE),
 *      (# AMHW_ZMF2X6_UART_INT_RX_VAL_ENABLE)这两个中断。
 */
uint32_t other_int_enable;

/** \brief RS485 方向控制函数, AM_TRUE: 发送模式,  AM_FALSE: 接收模式 */
void (*pfn_rs485_dir)(am_bool_t is_txmode);

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

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

} am_zmf2x6_uart_devinfo_t;

```

● 设备结构体定义

由于 UART 设备需要提供标准的 UART 服务, 因此, 该设备至少包含一个 UART 标准服务结构体成员, 也就是 `am_uart_serv_t` 类型成员, 其次, 为了在后续使用中, 可以通过设备访问到设备信息, 也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 5.18 所示, 后续如果需要使用到什么设备相关的全局变量, 就应该定义在这里。

程序清单 10.29 UART 设备结构体定义

```

/**
 * \brief 串口设备结构体定义
 */
typedef struct am_zmf2x6_uart_dev {

    am_uart_serv_t  uart_serv; /**< \brief 标准 UART 服务 */

```

```

/** \brief 指向用户注册的 txchar_get 函数 */
int (*pfn_txchar_get)(void *, char *);

/** \brief 指向用户注册的 rxchar_put 函数 */
int (*pfn_rxchar_put)(void *, char);

/** \brief 指向用户注册的错误回调函数 */
int (*pfn_err)(void *, int, void *, int);

void      *txget_arg;           /**< \brief txchar_get 函数参数 */
void      *rxput_arg;          /**< \brief rxchar_put 函数参数 */
void      *err_arg;            /**< \brief 错误回调函数用户参数 */

uint8_t   channel_mode;        /**< \brief 串口模式 中断/查询 */
uint32_t  baud_rate;           /**< \brief 串口波特率 */
uint16_t  options;             /**< \brief 硬件设置选项 */

uint32_t  other_int_enable;     /**< \brief 指定使能的其它中断 */

am_bool_t rs485_en;            /**< \brief 是否使能了 485 模式 */

const am_zmf2x6_uart_devinfo_t *p_devinfo; /**< \brief 指向设备信息常量的指针 */
} am_zmf2x6_uart_dev_t;

```

2. 搭建函数框架

函数主要分为两部分一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于 UART 这种已经有标准接口层的函数，通常驱动层只提供两个函数给用户，初始化函数和解初始化函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为用于标准接口的 handle。

解初始化函数主要是当不需要再使用某个外设时，释放掉其占用的相关资源，参数为设备指针，无返回值。示例代码如程序清单 10.30 所示。

程序清单 10.30 驱动层函数定义示例

```

/**
 * \brief 初始化 UART，返回 UART 标准服务操作句柄
 *
 * \param[in] p_dev      : 指向串口设备的指针
 * \param[in] p_devinfo : 指向串口设备信息常量的指针
 *
 * \return UART 标准服务操作句柄，值为 NULL 时表明初始化失败
 */

```



```

am_uart_handle_t am_zmf2x6_uart_init(am_zmf2x6_uart_dev_t          *p_dev,
                                     const am_zmf2x6_uart_devinfo_t *p_devinfo);

/**
 * \brief 不使用 UART 时，解初始化 UART，释放相关资源
 *
 * \param[in] p_dev: 指向串口设备的指针
 */
void am_zmf2x6_uart_deinit (am_zmf2x6_uart_dev_t *p_dev);

```

● 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 3 个函数，根据标准接口层中的 struct am_uart_drv_funcs 结构体，我们需要在.c 文件中定义 3 个函数。

由于这些函数全部是放在结构体中，通过函数指针的形式提供给上层，并不需要将这些函数声明到.h 文件中供用户直接使用，因此全部以 static 的形式定义成内部函数。定义好函数后，再定义一个 struct am_uart_drv_funcs 结构体，将函数放在其中。示例代码如程序清单 10.31 所示。

程序清单 10.31 提供给标准层函数定义

```

/* 串口驱动函数声明 */
static int __uart_ioctl (void *p_drv, int, void *);

static int __uart_tx_startup (void *p_drv);

static int __uart_callback_set (void *p_drv,
                               int    callback_type,
                               void *pfn_callback,
                               void *p_arg);

static int __uart_poll_getchar (void *p_drv, char *p_char);
static int __uart_poll_putchar (void *p_drv, char outchar);
static void __uart_irq_handler (void *p_arg);
/** \brief 标准层接口函数实现 */
static const struct am_uart_drv_funcs __g_uart_drv_funcs = {
    __uart_ioctl,
    __uart_tx_startup,
    __uart_callback_set,
    __uart_poll_getchar,
    __uart_poll_putchar,
};

```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。可以编译一下，应该还是没有错误的。如果有错误，表明结构没有搭建好。可能存在警告，毕竟有些变量没有使用，可以暂时不用理会。做到这里，其实驱动层已经开发一大半了。

10.3.2 驱动函数实现

通过上面的函数框架搭建，共计 7 个函数，5 个功能性函数面向标准层，2 个初始解初

始化函数面向用户，下面逐步实现。

1. am_zmf2x6_uart_init 函数

该函数主要完成两个功能：

- a) 初始化设备中的标准服务，将成员赋值，确定 p_drv 的值，p_drv 的值确定了提供给标准接口层的那 3 个函数的第一个参数是什么。那 3 个函数才能根据这儿的设置去使用。（void * 类型无法使用，必须转换为具体的一种类型才能使用），通常情况下，p_drv 都是设置为设备的地址，这样方便访问整个设备的相关内容。
- b) 初始化硬件相关设置，该函数的示例代码如程序清单 10.32 所示。

程序清单 10.32 am_zmf2x6_uart_init 函数示例代码

```
/**
 * \brief 串口初始化函数
 */
am_uart_handle_t am_zmf2x6_uart_init (am_zmf2x6_uart_dev_t *p_dev,
                                       const am_zmf2x6_uart_devinfo_t *p_devinfo)
{
    amhw_zmf2x6_uart_t *p_hw_uart;
    uint32_t tmp;

    if (p_devinfo == NULL) {
        return NULL;
    }
    /* 获取配置参数 */
    p_hw_uart = (amhw_zmf2x6_uart_t *)p_devinfo->uart_reg_base;
    p_dev->p_devinfo = p_devinfo;
    p_dev->uart_serv.p_funcs = (struct am_uart_drv_funcs *)&__g_uart_drv_funcs;
    p_dev->uart_serv.p_drv = p_dev;
    p_dev->baud_rate = p_devinfo->baud_rate;
    p_dev->options = 0;
    /* 初始化默认回调函数 */
    p_dev->pfn_txchar_get = (int (*)(void *, char*))__uart_dummy_callback;
    p_dev->txget_arg = NULL;
    p_dev->pfn_rxchar_put = (int (*)(void *, char))__uart_dummy_callback;
    p_dev->rxput_arg = NULL;
    p_dev->pfn_err =
        (int (*)(void *, int, void*, int))__uart_dummy_callback;
    p_dev->err_arg = NULL;
    p_dev->other_int_enable = p_devinfo->other_int_enable &
        ~(AMHW_ZMF2X6_UART_INT_TX_EMPTY_ENABLE |
         AMHW_ZMF2X6_UART_INT_RX_VAL_ENABLE);
    p_dev->rs485_en = AM_FALSE;
    /* 获取串口数据长度配置选项 */
    tmp = p_devinfo->cfg_flags;
    tmp = (tmp >> 4) & 0x03;
```

```

switch (tmp) {
case 0:
    p_dev->options |= AM_UART_CS5;
    break;
case 1:
    p_dev->options |= AM_UART_CS6;
    break;
case 2:
    p_dev->options |= AM_UART_CS7;
    break;
case 3:
    p_dev->options |= AM_UART_CS8;
    break;
default:
    p_dev->options |= AM_UART_CS8;
    break;
}
/* 开启 UART 时钟 */
am_clk_enable(p_devinfo->clk_num);
/* 获取串口检验方式配置选项 */
tmp = p_devinfo->cfg_flags;
tmp = (tmp >> 0) & 0x03;
if (tmp == 1) {
    p_dev->options |= AM_UART_PARENB | AM_UART_PARODD;
} else {
}
/* 获取串口停止位配置选项 */
if (p_devinfo->cfg_flags & (AMHW_ZMF2X6_UART_STOP_2BIT)) {
    p_dev->options |= AM_UART_STOPB;
} else {
}

/* 等待上一次传输完成 */
while (amhw_zmf2x6_uart_status_flag_check(p_hw_uart,
AMHW_ZMF2X6_UART_TX_COMPLETE_FALG) == AM_FALSE);
__uart_opt_set(p_dev, p_dev->options);
/* 设置波特率 */
p_dev->baud_rate = amhw_zmf2x6_uart_baudrate_set(
                                p_hw_uart,
                                am_clk_rate_get(p_dev->p_devinfo->clk_num),
                                p_devinfo->baud_rate);

/* 默认轮询模式 */
__uart_mode_set(p_dev, AM_UART_MODE_POLL);
/* uart 使能 */

```

```

amhw_zmf2x6_uart_rx_enable(p_hw_uart,AM_TRUE);
amhw_zmf2x6_uart_tx_enable(p_hw_uart,AM_TRUE);
amhw_zmf2x6_uart_enable(p_hw_uart);
if (p_dev->p_devinfo->pfn_plfm_init) {
    p_dev->p_devinfo->pfn_plfm_init();
}
if (p_dev->p_devinfo->pfn_rs485_dir) {
    /* 初始化 485 为接收模式 */
    p_dev->p_devinfo->pfn_rs485_dir(AM_FALSE);
}
return &(amp_dev->uart_serv);
}

```

2. am_zmf2x6_uart_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放，同时，由于 UART 一旦使能将无法关闭，所以也无法释放硬件资源。

因此，该函数仅简单的将几个不再使用的变量置为 NULL 即可。示例代码如程序清单 10.33 所示。

程序清单 10.33 am_zmf2x6_uart_deinit 函数示例代码

```

/**
 * \brief 串口去初始化
 */
void am_zmf2x6_uart_deinit(am_zmf2x6_uart_dev_t *p_dev)
{
    amhw_zmf2x6_uart_t *p_hw_uart = (amhw_zmf2x6_uart_t *)p_dev->p_devinfo->uart_reg_base;
    p_dev->uart_serv.p_funcs = NULL;
    p_dev->uart_serv.p_drv = NULL;

    if (p_dev->channel_mode == AM_UART_MODE_INT) {

        /* 默认为轮询模式 */
        __uart_mode_set(p_dev, AM_UART_MODE_POLL);
    }

    /* 关闭串口 */
    amhw_zmf2x6_uart_disable(p_hw_uart);

    am_int_disable(p_dev->p_devinfo->inum);

    if (p_dev->p_devinfo->pfn_plfm_deinit) {
        p_dev->p_devinfo->pfn_plfm_deinit();
    }
}

```

3. __uart_ioctl 函数

该函数主要用于波特率获取，模式设置，硬件功能配置等功能，功能实现代码详见程序清单 10.34。

程序清单 10.34 __uart_ioctl 函数示例代码

```
/**
 * \brief 设备控制函数
 *
 * 其中包含设置获取波特率，模式设置（中断/查询），获取支持的模式，硬件选项设置等功能。
 */
static int __uart_ioctl (void *p_drv, int request, void *p_arg)
{
    am_zmf2x6_uart_dev_t *p_dev      = (am_zmf2x6_uart_dev_t *)p_drv;
    amhw_zmf2x6_uart_t    *p_hw_uart = (amhw_zmf2x6_uart_t *)p_dev->p_devinfo->uart_reg_base;
    int status = AM_OK;
    switch (request) {
        /* 波特率设置 */
        case AM_UART_BAUD_SET:
            /* 只有在当前传输完成的基础上才允许修改波特率 */
            while (amhw_zmf2x6_uart_status_flag_check(p_hw_uart,
                AMHW_ZMF2X6_UART_TX_COMPLETE_FALG) == AM_FALSE);
            status = amhw_zmf2x6_uart_baudrate_set(p_hw_uart,
                am_clk_rate_get(p_dev->p_devinfo->clk_num),
                (uint32_t)p_arg);

            if (status > 0) {
                p_dev->baud_rate = status;
                status = AM_OK;
            } else {
                status = -AM_EIO;
            }
            break;
        /* 波特率获取 */
        case AM_UART_BAUD_GET:
            *(int *)p_arg = p_dev->baud_rate;
            break;
        /* 模式设置 */
        case AM_UART_MODE_SET:
            status = (__uart_mode_set(p_dev, (int)p_arg) == AM_OK)
                ? AM_OK : -AM_EIO;
            break;
        /* 模式获取 */
        case AM_UART_MODE_GET:
            *(int *)p_arg = p_dev->channel_mode;
            break;
    }
}
```

```

/* 获取串口可设置的模式 */
case AM_UART_AVAIL_MODES_GET:
    *(int *)p_arg = AM_UART_MODE_INT | AM_UART_MODE_POLL;
    break;
/* 串口选项设置 */
case AM_UART_OPTS_SET:
    status = (__uart_opt_set(p_dev, (int)p_arg) == AM_OK) ? AM_OK : -AM_EIO;
    break;
/* 串口选项获取 */
case AM_UART_OPTS_GET:
    *(int *)p_arg = p_dev->options;
    break;
case AM_UART_RS485_SET:
    if (p_dev->rs485_en != (am_bool_t)(int)p_arg) {
        p_dev->rs485_en = (am_bool_t)(int)p_arg;
    }
    break;
case AM_UART_RS485_GET:
    *(int *)p_arg = p_dev->rs485_en;
    break;
default:
    status = -AM_EIO;
    break;
}
return (status);
}

```

4. __uart_tx_startup 函数

该函数主要用于设置 UART 发送功能，功能实现代码详见程序清单 10.35。

程序清单 10.35 __uart_tx_startup 函数示例代码

```

/**
 * \brief 启动串口发送(仅限中断模式)
 */
int __uart_tx_startup (void *p_drv)
{
    char data = 0;
    am_zmf2x6_uart_dev_t *p_dev = (am_zmf2x6_uart_dev_t *)p_drv;
    amhw_zmf2x6_uart_t *p_hw_uart = (amhw_zmf2x6_uart_t *)p_dev->p_devinfo->uart_reg_base;
    /* 使能 485 发送控制引脚 */
    if (p_dev->rs485_en && p_dev->p_devinfo->pfn_rs485_dir) {
        p_dev->p_devinfo->pfn_rs485_dir(AM_TRUE);
    }
    /* 等待上一次传输完成 */

```

```

while (amhw_zmf2x6_uart_status_flag_check(p_hw_uart,
AMHW_ZMF2X6_UART_TX_COMPLETE_FALG) == AM_FALSE);
/* 获取发送数据并发送 */
if ((p_dev->pfn_txchar_get(p_dev->txget_arg, &data)) == AM_OK) {
    amhw_zmf2x6_uart_data_write(p_hw_uart, data);
}
/* 使能发送中断 */
amhw_zmf2x6_uart_int_enable(p_hw_uart, AMHW_ZMF2X6_UART_INT_TX_EMPTY_ENABLE);
return AM_OK;
}

```

5. __uart_poll_putchar 函数

该函数主要用于字符获取功能，功能实现代码详见程序清单 10.36。

程序清单 10.36 __uart_poll_putchar 函数示例代码

```

/**
 * \brief 轮询模式下发送一个字符
 */
static int __uart_poll_putchar (void *p_drv, char outchar)
{
    am_zmf2x6_uart_dev_t *p_dev      = (am_zmf2x6_uart_dev_t *)p_drv;
    amhw_zmf2x6_uart_t *p_hw_uart = (amhw_zmf2x6_uart_t *)p_dev->p_devinfo->uart_reg_base;
    /* 发送模块是否空闲, AM_FALSE:忙; TURE: 空闲 */
    if(amhw_zmf2x6_uart_status_flag_check(p_hw_uart, AMHW_ZMF2X6_UART_TX_EMPTY_FLAG) ==
AM_FALSE) {
        return (-AM_EAGAIN);
    } else {
        if ((p_dev->rs485_en) && (p_dev->p_devinfo->pfn_rs485_dir != NULL)) {
            /* 设置 485 为发送模式 */
            p_dev->p_devinfo->pfn_rs485_dir(AM_TRUE);
        }
        /* 发送一个字符 */
        amhw_zmf2x6_uart_data_write(p_hw_uart, outchar);
        if (p_dev->rs485_en && p_dev->p_devinfo->pfn_rs485_dir) {
            /* 等待发送完成 */
            while (!(p_hw_uart->csr & 0x01));
            p_dev->p_devinfo->pfn_rs485_dir(AM_FALSE);
        }
    }
    return (AM_OK);
}

```

10.4 用户配置文件编写

10.4.1 定义设备常量于设备信息常量

创建配置文件后，我们需要在里面加入设备变量，设备变量只需要定义，不用初始化。

设备信息常量定义时初始化，默认填充的信息是一种常用的配置方式，UART 设备信息配置如程序清单 10.37 所示。

程序清单 10.37 UART 设备信息配置文件

```
/** \brief 串口 1 设备信息 */
static const am_zmf2x6_uart_devinfo_t __g_uart1_devinfo = {
    ZMF2X6_UART1_BASE,          /**< \brief 串口 1 */
    INUM_UART1,                 /**< \brief 串口 1 的中断编号 */
    CLK_UART1,                  /**< \brief 串口 1 的时钟 */
    AMHW_ZMF2X6_UART_DATA_8BIT | /**< \brief 8 位数据 */
    AMHW_ZMF2X6_UART_PARITY_NO | /**< \brief 无极性 */
    AMHW_ZMF2X6_UART_STOP_1BIT, /**< \brief 1 个停止位 */
    115200,                     /**< \brief 设置的波特率 */
    0,                          /**< \brief 无其他中断 */
    NULL,                       /**< \brief USART1 使用 RS485 */
    __zmf2x6_plfm_uart1_init,    /**< \brief USART1 的平台初始化 */
    __zmf2x6_plfm_uart1_deinit, /**< \brief USART1 的平台去初始化 */
};

static am_zmf2x6_uart_dev_t __g_uart1_dev; /**< \brief 定义串口 1 设备 */
```

10.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

平台初始化函数和解初始化函数定义如程序清单 10.38 所示。

程序清单 10.38 平台初始化函数和解初始化函数定义

```
/** \brief 串口 1 平台初始化 */
static void __zmf2x6_plfm_uart1_init(void)
{
    /* 引脚初始化 PIOA_9_UART1_TX PIOA_10_UART1_RX */
    am_gpio_pin_cfg(PIOA_9, PIOA_9_UART1_TX | PIOA_9_AF_PP);
    am_gpio_pin_cfg(PIOA_10, PIOA_10_UART1_RX | PIOA_10_INPUT_FLOAT);
}

/** \brief 解除串口 1 平台初始化 */
static void __zmf2x6_plfm_uart1_deinit(void)
{
    /* 关闭 UART1 时钟 */
    am_clk_disable(CLK_UART1);

    am_gpio_pin_cfg(PIOA_9, AM_GPIO_INPUT);
    am_gpio_pin_cfg(PIOA_10, AM_GPIO_INPUT);
}
```



```

/** \brief 串口 1 设备信息 */
static const am_zmf2x6_uart_devinfo_t __g_uart1_devinfo = {
    ZMF2X6_UART1_BASE,           /**< \brief 串口 1 */
    INUM_UART1,                  /**< \brief 串口 1 的中断编号 */
    CLK_UART1,                   /**< \brief 串口 1 的时钟 */

    AMHW_ZMF2X6_UART_DATA_8BIT | /**< \brief 8 位数据 */
    AMHW_ZMF2X6_UART_PARITY_NO | /**< \brief 无极性 */
    AMHW_ZMF2X6_UART_STOP_1BIT,  /**< \brief 1 个停止位 */

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

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

    NULL,                        /**< \brief USART1 使用 RS485 */
    __zmf2x6_plfm_uart1_init,     /**< \brief USART1 的平台初始化 */
    __zmf2x6_plfm_uart1_deinit,  /**< \brief USART1 的平台去初始化 */
};

```

10.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在标准层实例公共头文件 `am_zmf2x6_inst_init.h` 中加入它们的声明，如所示程序清单 10.39。

程序清单 10.39 平台初始化解初始化声明

```

/**
 * \brief UART 实例初始化，获得 ADC 标准服务句柄
 *
 * \param 无
 *
 * \return UART 标准服务句柄，若为 NULL，表明初始化失败
 */
am_uart_handle_t am_zmf2x6_uart1_inst_init(void);

/**
 * \brief ADC 实例解初始化
 *
 * \param[in] handle 通过 am_zmf2x6_uart1_inst_init() 函数获得的 ADC 标准服务句柄
 *
 * \return 无
 */
void am_zmf2x6_uart1_inst_deinit(am_uart_handle_t handle);

```

用户使用时，应该先调用平台初始化函数 `am_zmf2x6_uart1_inst_init` 函数，初始化 UART，并保存其返回的 `handle`，后续直接使用保存的 `handle` 作为标准接口的第一个参数使用标准

接口函数即可。

至此，用户配置文件开发完毕。

10.5 例程文件开发

参考 5.5.1 章节新建用户例程文件，搭建文件的基本框架，准备 UART 例程内容的编写。例程文件开发主要分为两个部分，HW 例程和 STD 例程。

10.5.1 例程内容开发

1. HW 层实现示例

这里使用 HW 层来实现 UART 中断方式来发送数据，相关代码如程序清单 10.40 所示。

程序清单 10.40 UART 串口发送中断模式

```
/**
 * \brief 串口发送数据（中断模式下，该函数不会阻塞）
 *
 * \param[in] p_hw_uart 指向需控制的串口设备结构体
 * \param[in] p_buf 指向发送数据的缓冲区
 * \param[in] len 发送的字节数
 *
 * \retval 1 发送就绪
 * \retval 0 发送模块忙碌，无法发送
 *
 * \note 发送缓冲区最好是全局的，保证在所有数据发送完成前，缓冲区没有被释放掉
 */
static uint8_t uart_int_send(amhw_zmf2x6_uart_t *p_hw_uart,
                             const uint8_t *p_buf,
                             uint32_t len)
{
    if (g_tx_index >= g_tx_nbytes) { /* 上次发送已经发送完成 */
        gp_tx_buf = p_buf; /* 重新定义发送缓冲区 */
        g_tx_index = 0; /* 从缓存区起始字节开始发送 */
        g_tx_nbytes = len; /* 发送 g_tx_nbytes 个字节 */

        /* 必须先写一个字节的数据才以产生发送中断 */
        while(amhw_zmf2x6_uart_status_flag_check(p_hw_uart,
                                                  AMHW_ZMF2X6_UART_TX_COMPLETE_FALG) == AM_FALSE);

        amhw_zmf2x6_uart_data_write(p_hw_uart, p_buf[g_tx_index++]);

        amhw_zmf2x6_uart_int_enable(p_hw_uart,
                                     AMHW_ZMF2X6_UART_INT_TX_EMPTY_ENABLE);

        return 1;
    }
}
```

```

return 0;
}

```

2. 标准层实现示例

这里使用标准层来实现 UART 轮询方式来收发数据，相关代码如程序清单 10.41 所示。

程序清单 10.41 UART 串口收发轮训模式

```

/**
 * \brief 例程入口
 */
void demo_std_uart_polling_entry (am_uart_handle_t handle)
{
    uint8_t uart1_buf[5];    /* 数据缓冲区 */

    am_uart_poll_send(handle, __ch, sizeof(__ch));

    while (1) {
        /* 接收字符 */
        am_uart_poll_receive(handle, uart1_buf, 1);

        /* 发送刚刚接收的字符 */
        am_uart_poll_send(handle, uart1_buf, 1);
    }
}

```

所有例程开发完毕后，整个外设的开发工作也就圆满结束了。

可以提交相关文件，以完成整个开发任务。提交文件时，不用提交工程文件，只需要提交开发的源文件和头文件即可。

第11章 通用外设驱动开发——I2C

11.1 标准层了解

I2C 标准层仅包含一个文件，am_i2c.h 文件，在 ametal\interface 找到该文件打开。

11.1.1 了解相关类型定义

打开 am_i2c.h 文件，我们可以看到，前面定义了 I2C 的从设备地址位数、操作指令、字节传输模式、子地址宽度、从设备描述结构体、传输结构体、消息结构体以及驱动函数结构体等，详见程序清单 11.1。

程序清单 11.1 I2C 标准接口文件结构体定义

```
#define AM_I2C_M_7BIT          0x0000u    /**< \brief 7-bits 设备地址 */
#define AM_I2C_M_10BIT         0x0001u    /**< \brief 10-bits 设备地址 */
#define AM_I2C_M_WR            0x0000u    /**< \brief 写操作 */
#define AM_I2C_M_RD            0x0002u    /**< \brief 读操作 */
#define AM_I2C_M_NOSTART       0x0004u    /**< \brief 无需重新启动 */
#define AM_I2C_M_REV_DIR_ADDR  0x0008u    /**< \brief 读写标志位反转 */
#define AM_I2C_M_RECV_LEN      0x0010u    /**< \brief 暂不支持 */

/** \brief 忽略所有无应答错误 (包括从机地址无应答和数据无应答) */
#define AM_I2C_M_IGNORE_NAK    0x0020u

/** \brief 在读操作中接收数据时不发送应答 */
#define AM_I2C_M_NO_RD_ACK     0x0040u

/** \brief I2C 传输控制标志掩码 */
#define AM_I2C_M_MASK          0x00FFu

/** \brief 7 位地址模式(默认模式) */
#define AM_I2C_ADDR_7BIT       AM_I2C_M_7BIT

/** \brief 10 位地址模式 */
#define AM_I2C_ADDR_10BIT      AM_I2C_M_10BIT

/** \brief 忽略设备的无应答 */
#define AM_I2C_IGNORE_NAK      AM_I2C_M_IGNORE_NAK

/** \brief 器件内子地址高位字节先传输(默认) */
#define AM_I2C_SUBADDR_MSB_FIRST 0x0000u

/** \brief 器件内子地址低位字节先传输 */
#define AM_I2C_SUBADDR_LSB_FIRST 0x0100u

#define AM_I2C_SUBADDR_NONE     0x0000u    /**< \brief 无子地址(默认) */
```

```

#define AM_I2C_SUBADDR_1BYTE      0x1000u  /**< \brief 子地址宽度 1 字节 */
#define AM_I2C_SUBADDR_2BYTE      0x2000u  /**< \brief 子地址宽度 2 字节 */

/** \brief 从标志控制字中取得子地址宽度 */
#define AM_I2C_SUBADDR_LEN_GET(flags) (((flags) & 0xF000) >> 12)

struct am_i2c_message;

/**
 * \brief I2C 驱动函数结构体
 */
struct am_i2c_drv_funcs {

    /** \brief 开始消息处理 */
    int (*pfn_i2c_msg_start)(void *p_drv, struct am_i2c_message *p_trans);

};

/**
 * \brief I2C 标准服务结构体
 */
typedef struct am_i2c_serv {
    struct am_i2c_drv_funcs  *p_funcs;      /**< \brief I2C 驱动函数          */
    void                      *p_drv;       /**< \brief I2C 驱动函数入口参数 */
} am_i2c_serv_t;

/** \brief I2C 标准服务操作句柄定义 */
typedef am_i2c_serv_t *am_i2c_handle_t;

/**
 * \brief I2C 从设备描述结构体 (推荐使用 am_i2c_mkdev() 设置本数据结构)
 */
typedef struct am_i2c_device {
    am_i2c_handle_t handle;      /**< \brief 与从设备关联的 I2C 服务标准 handle */
    uint16_t        dev_addr;    /**< \brief 从机设备地址, 7 位或 10 位          */
    uint16_t        dev_flags;   /**< \brief 从机设备特性, 见 “I2C 从设备属性标志” */
} am_i2c_device_t;

/**
 * \brief I2C 传输结构体 (推荐使用 am_i2c_mktrans() 设置本数据结构)
 */
typedef struct am_i2c_transfer {
    uint16_t        addr;        /**< \brief I2C 设备地址          */
    uint16_t        flags;       /**< \brief I2C 传输标志 见 “I2C 传输控制标志” */

```

```

uint8_t    *p_buf;          /**< \brief 数据缓冲区          */
uint32_t    nbytes;         /**< \brief 数据个数          */
} am_i2c_transfer_t;

/**
 * \brief I2C 消息 (推荐使用 am_i2c_mkmsg() 设置本数据结构)
 */
typedef struct am_i2c_message {
    am_i2c_transfer_t *p_transfers; /**< \brief 组成消息的传输          */
    uint16_t          trans_num;     /**< \brief 请求处理的传输个数        */
    uint16_t          done_num;      /**< \brief 成功处理的传输个数        */
    am_pfnvoid_t       pfn_complete; /**< \brief 传输完成回调函数          */
    void              *p_arg;        /**< \brief 传递给 pfn_complete 的参数 */
    int                status;       /**< \brief 消息的状态              */
    void              *ctrlrdata[2]; /**< \brief 控制器使用              */
} am_i2c_message_t;

```

1. I2C 宏定义说明

标准文件中定义了诸多宏，包含 I2C 的从设备地址位数、操作指令、字节传输模式、子地址宽度，每个宏都有对应的解释说明，这里不一一赘述。

2. I2C 从设备描述结构体

这个结构体包含从机设备地址和特性，推荐使用 `am_i2c_mkdev` 函数设置本数据结构；

3. I2C 传输结构体

这个结构体包含 I2C 设备地址、传输标志、数据缓存区指针和数据个数，推荐使用 `am_i2c_mktrans` 函数设置本数据结构；

4. I2C 消息结构体

这个结构体包含传输结构体指针、请求处理的传输个数、成功处理的传输个数、传输完成回调函数和参数、消息状态和控制器指针，推荐使用 `am_i2c_mkmsg` 函数设置本数据结构；

5. struct am_i2c_drv_funcs 结构体

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层要实现 1 个函数：

- I2C 开始消息处理函数 `pfn_i2c_msg_start`；

在驱动层编写函数时，应该向标准接口层提供这 1 个函数。

这些函数的有一个共同点，就是第一个参数都是 `void` 型的指针，也就是说，驱动层提供的函数，标准接口层使用时传递的第一个参数并无确定类型，标准接口层不关心第一个参数的具体类型，使用时只负责传递就可以了。

6. typedef struct am_i2c_serv 结构体

该结构体定义了标准的 I2C 服务，包含了上文提到的驱动函数结构体的指针和一个 `void` 型的指针。

这个标准的 I2C 服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

- `p_funcs` 成员赋值

这是一个指向 `typedef struct am_i2c_serv` 结构体的指针，我们只需要在驱动层定义一个 `struct am_i2c_drv_funcs` 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那 1 个函数地址。然后，将 `struct am_i2c_drv_funcs` 结构体变量的地址赋值给 `p_funcs` 成员即可。

- `p_drv` 成员赋值

这个变量是 `void` 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动自行决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

如果这个结构定义好了，标准接口层怎么知道有这个结构体呢？标准接口层怎么访问这个结构体呢？换句话说，驱动层怎么将该结构体提供给标准接口层呢？继续看下面 `handle` 的定义。

7. `am_i2c_handle_t` 的定义

这个是一个标准的操作句柄，本质上就是 `typedef am_i2c_serv_t` 指针类型，驱动层的初始化函数需要返回该类型的一个 `handle` 给用户。这样，用户获取到 `handle`，就可以在需要使用标准接口层函数时，将该 `handle` 传递给标准接口层了。标准接口层就可以通过该 `handle`（`typedef am_i2c_serv_t` 结构体指针）访问到驱动层提供的函数了。标准接口层怎么使用 `handle` 指针？看一下标准接口层的函数实现就明白了。

11.1.2 了解标准接口函数

在 `am_i2c.h` 文件相关的结构体、类型定义后，就是具体函数的实现，如程序清单 11.2 所示。

程序清单 11.2 I2C 标准接口函数

```
/**
 * \brief I2C 从机设备描述结构体参数设置
 *
 * \param[in] p_dev      : 指向从机设备描述结构体的指针
 * \param[in] handle     : 与从设备关联的 I2C 标准服务操作句柄
 * \param[in] dev_addr   : 从机设备地址
 * \param[in] dev_flags  : 从机设备特性，见“I2C 从设备属性标志”
 *
 * \return 无
 */
am_static_inline
void am_i2c_mkdev (am_i2c_device_t *p_dev,
                  am_i2c_handle_t handle,
                  uint16_t dev_addr,
                  uint16_t dev_flags)
{
    p_dev->handle      = handle;
    p_dev->dev_addr     = dev_addr;
    p_dev->dev_flags    = dev_flags;
}
```

```

/**
 * \brief I2C 传输结构体信息参数设置
 *
 * \param[in] p_trans : 指向传输结构体的指针
 * \param[in] addr    : 操作的从机设备地址
 * \param[in] flags    : 传输过程中的控制标识位, AM_I2C_M_*的宏值或者多个
 *                      AM_I2C_M_*宏的或 (OR) 值 (#AM_I2C_M_RD)
 * \param[in] p_buf    : 发送或者接收数据缓冲区
 * \param[in] nbytes   : 传输的数据个数
 *
 * \retval  AM_OK      : 传输结构体参数设置完成
 * \retval -AM_EINVAL : 参数错误
 */
am_static_inline
void am_i2c_mktrans (am_i2c_transfer_t *p_trans,
                    uint16_t          addr,
                    uint16_t          flags,
                    uint8_t            *p_buf,
                    uint32_t           nbytes)
{
    p_trans->addr    = addr;
    p_trans->flags    = flags;
    p_trans->p_buf    = p_buf;
    p_trans->nbytes   = nbytes;
}

/**
 * \brief 设置 I2C 消息结构体参数
 *
 * \param[in] p_msg      : I2C 消息描述符指针
 * \param[in] p_transfers : 组成消息的 “I2C 传输”
 * \param[in] trans_num   : “I2C 传输” 的个数
 * \param[in] pfn_complete : 消息处理完成回调函数
 * \param[in] p_arg       : 传递给回调函数的参数
 *
 */
am_static_inline
void am_i2c_mkmsg (am_i2c_message_t *p_msg,
                  am_i2c_transfer_t *p_transfers,
                  uint16_t           trans_num,
                  am_pfnvoid_t       pfn_complete,
                  void                *p_arg)
{
    p_msg->p_transfers = p_transfers;

```



```

    p_msg->trans_num    = trans_num;
    p_msg->done_num      = 0;
    p_msg->pfn_complete = pfn_complete;
    p_msg->p_arg         = p_arg;
    p_msg->status        = -AM_ENOTCONN;
}

/**
 * \brief 开始处理一个消息
 *
 * 以异步的方式处理消息，若当前 I2C 控制器空闲，则该消息会得到立即执行，
 * 若处理器不空闲，则会将该消息加入一个队列中排队等待，I2C 控制器将顺序处理
 * 队列中的消息。消息的处理状态和结果反映在\p p_msg->status。
 * 消息处理结束(成功、超时或出错)时，将会调用\p p_msg->pfn_complete 并传递
 * 参数\p p_msg->p_arg。
 *
 * \param[in]    handle : I2C 标准服务操作句柄
 * \param[in,out] p_msg  : 要处理的消息
 *
 * 函数的返回值如下：
 * \retval  AM_OK      : 消息排队成功，等待处理
 * \retval -AM_EINVAL : 参数错误
 *
 * 成功处理的传输个数反映在\p p_msg->done_num
 *
 * 消息的处理状态和结果反映在\p p_msg->status :
 *
 * \li -AM_ENOTCONN    消息尚未排队
 * \li -AM_EISCONN     消息正在排队处理
 * \li -AM_EINPROGRESS 消息正在被处理
 * 下面为消息处理完毕的结果 (可通过\p p_msg->done_num 定位出错的传输)
 * \li AM_OK           所有传输成功处理
 * \li -AM_ENOTSUP     某次传输的控制标志不支持
 * \li -AM_ENODEV      某次传输的目标地址无应答
 * \li -AM_ENOENT      某次传输发送的数据无应答
 * \li -AM_EIO         传输消息时发生总线错误、总线仲裁失败过多等错误
 * \li -AM_ECANCELED   因控制器出错或处理队列满，消息被取消，可稍后再试
 *
 */
am_static_inline
int am_i2c_msg_start (am_i2c_handle_t handle, am_i2c_message_t *p_msg)
{
    return handle->p_funcs->pfn_i2c_msg_start(handle->p_drv, p_msg);
}

```

乍一看，很简单的 4 个函数，全都是用内联函数就搞定了。以上函数功能在类型定义小节已说明。

可能疑惑的是，为什么需要定义一个 `am_i2c_handle_t` 这样的类型，直接定义成 `struct am_i2c_serv` 的指针多清晰易懂啊？在这里定义这样一个 `handle` 的概念，主要是为了减少用户的理解，用户只需要从驱动层获取一个 `handle`，就可以使用标准接口层函数，不需要关心这个 `handle` 具体是什么。只要驱动和标准接口知道这个 `handle` 是什么就可以了。

除去这几个内联函数，我们可以看到下面还有两个函数，但只是做了声明，详见程序清单 11.3。

程序清单 11.3 I2C 标准接口函数

```
/**
 * \brief I2C 写数据
 *
 * \param[in] p_dev      : 指向从机设备信息的结构体的指针
 * \param[in] sub_addr : 从机设备子地址
 * \param[in] p_buf      : 指向发送/接收数据缓存
 * \param[in] nbytes     : 数据缓存长度
 *
 * \return  AM_OK : 写数据完成
 */
int am_i2c_write(am_i2c_device_t *p_dev,
                 uint32_t          sub_addr,
                 const uint8_t     *p_buf,
                 uint32_t          nbytes);

/**
 * \brief I2C 读数据
 *
 * \param[in] p_dev      : 指向从机设备信息的结构体的指针
 * \param[in] sub_addr : 从机设备子地址
 * \param[in] p_buf      : 指向发送/接收数据缓存
 * \param[in] nbytes     : 数据缓存长度
 *
 * \return  AM_OK : 读数据完成
 */
int am_i2c_read(am_i2c_device_t *p_dev,
                uint32_t          sub_addr,
                uint8_t           *p_buf,
                uint32_t          nbytes);
```

可以看到这里是声明了几个函数，并非简单内联函数，这部分需要由驱动层来实现，这里提供函数声明供调用。

到此我们就分析完了 I2C 外设标准层文件，清楚了驱动层需要为标准接口层提供哪些内容。

11.2 HW 层开发

现在我们打开 ZMF2x6 芯片用户手册，对照手册完成 I2C 外设 HW 层的开发。

打开 ametal\soc\zlg\drivers\include\i2c\hw 路径，在该路径下创建名为 amhw_zmf2x6_i2c.h 的文件，根据代码规范和生成 doxygen 文档的注释要求，在 amhw_zmf2x6_i2c.h 文件中添加相关的注释以及防重复定义等必要内容，参考程序清单 5.3。

11.2.1 添加寄存器列表

HW 层就是与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读该外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMF2x6 用户手册的 I2C 章节，可以看到，I2C 章节部分，共有 34 个寄存器，简单描述如下：

表 11.1 I2C 寄存器概览

位	标识符	含义
1	I2C_CR	I2C 控制寄存器
2	I2C_TAR	I2C 目标地址寄存器
3	I2C_SAR	I2C 从机地址寄存器
4	I2C_DR	I2C 数据命令寄存器
5	I2C_SSHR	标准模式 I2C 时钟高电平计数寄存器
6	I2C_SSLR	标准模式 I2C 时钟低电平计数寄存器
7	I2C_FSHR	快速模式 I2C 时钟高电平计数寄存器
8	I2C_FSLR	快速模式 I2C 时钟低电平计数寄存器
9	I2C_ISR	I2C 中断状态寄存器
10	I2C_IMR	I2C 中断屏蔽寄存器
11	I2C_RAWISR	I2C RAW 中断寄存器
12	I2C_RXTLR	I2C 接收阈值
13	I2C_TXTLR	I2C 发送阈值
14	I2C_ICR	I2C 组合和独立中断清除寄存器
15	I2C_RX_UNDER	I2C 清除 RX_UNDER 中断寄存器
16	I2C_RX_OVER	I2C 清除 RX_OVER 中断寄存器
17	I2C_TX_OVER	I2C 清除 TX_OVER 中断寄存器
18	I2C_RD_REQ	I2C 清除 RD_REQ 中断寄存器
19	I2C_TX_ABRT	I2C 清除 TX_ABRT 中断寄存器
20	I2C_RX_DONE	I2C 清除 RX_DONE 中断寄存器
21	I2C_ACTIV	I2C 清除 ACTIVITY 中断寄存器
22	I2C_STOP	I2C 清除 STOP_DET 中断寄存器
23	I2C_START	I2C 清除 START_DET 中断寄存器
24	I2C_GC	I2C 清除 GEN_CALL 中断寄存器
25	I2C_ENR	I2C 使能寄存器
26	I2C_SR	I2C 状态寄存器
27	I2C_TXFLR	I2C 发送缓冲水平寄存器

28	I2C_RXFLR	I2C 接收缓冲水平寄存器
29	I2C_HOLD	I2C SDA 保持时间寄存器
30	I2C_DMA	I2C DMA 控制寄存器
31	I2C_SETUP	I2C SDA 建立时间寄存器
32	I2C_GCR	I2C 广播呼叫 ACK 寄存器
33	I2C_SLVMASK	I2C 从机地址掩码寄存器
34	I2C_SLVRCVADDR	I2C 广播呼叫 ACK 寄存器

到此 I2C 的各个寄存器及其功能基本阅读完毕，我们也大致了解了各个寄存器功能以及要配置各个功能，其他寄存器位该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，如程序清单 11.4 所示。

程序清单 11.4 寄存器结构体定义

```
/**
 * \brief I2C 寄存器块结构体
 */
typedef struct amhw_zmf2x6_i2c {
    __IO uint16_t  cr;                /**< \brief I2C 控制寄存器 */
    uint16_t      reserved0;         /**< \brief 保留 0 */
    __IO uint16_t  tar;              /**< \brief I2C 目标地址寄存器 */
    uint16_t      reserved1;         /**< \brief 保留 1 */
    __IO uint16_t  sar;              /**< \brief I2C 从机地址寄存器 */
    uint16_t      reserved2;         /**< \brief 保留 2 */
    __IO uint16_t  hs_maddr;         /**< \brief 暂且保留 */
    uint16_t      reserved3;         /**< \brief 保留 3 */
    __IO uint16_t  dr;               /**< \brief I2C 数据命令寄存器 */
    uint16_t      reserved4;         /**< \brief 保留 4 */
    __IO uint16_t  ss_scl_hcnt;      /**< \brief 标准模式 I2C 时钟高电平计数寄存器 */
    uint16_t      reserved5;         /**< \brief 保留 5 */
    __IO uint16_t  ss_scl_lcnt;      /**< \brief 标准模式 I2C 时钟低电平计数寄存器 */
    uint16_t      reserved6;         /**< \brief 保留 6 */
    __IO uint16_t  fs_scl_hcnt;      /**< \brief 快速模式 I2C 时钟高电平计数寄存器 */
    uint16_t      reserved7;         /**< \brief 保留 7 */
    __IO uint16_t  fs_scl_lcnt;      /**< \brief 快速模式 I2C 时钟低电平计数寄存器 */
    uint16_t      reserved8;         /**< \brief 保留 8 */
    __IO uint16_t  hs_scl_hcnt;      /**< \brief 暂且保留 */
    uint16_t      reserved9;         /**< \brief 保留 9 */
    __IO uint16_t  hs_scl_lcnt;      /**< \brief 暂且保留 */
    uint16_t      reserved10;        /**< \brief 保留 10 */
    __IO uint16_t  isr;              /**< \brief I2C 中断状态寄存器 */
    uint16_t      reserved11;        /**< \brief 保留 11 */
    __IO uint16_t  imr;              /**< \brief I2C 中断屏蔽寄存器 */
    uint16_t      reserved12;        /**< \brief 保留 12 */
    __IO uint16_t  raw_isr;          /**< \brief I2C RAW 中断寄存器 */
    uint16_t      reserved13;        /**< \brief 保留 13 */
}
```

```

__IO uint16_t ic_rx_tlr;          /**< \brief I2C 接收阈值 */
    uint16_t reserved14;          /**< \brief 保留 14 */
__IO uint16_t ic_tx_tlr;          /**< \brief I2C 发送阈值 */
    uint16_t reserved15;          /**< \brief 保留 15 */
__IO uint16_t icr;                 /**< \brief I2C 组合和独立中断清除寄存器 */
    uint16_t reserved16;          /**< \brief 保留 16 */
__IO uint16_t clr_rx_under;        /**< \brief I2C 清除 RX_UNDER 中断寄存器 */
    uint16_t reserved17;          /**< \brief 保留 17 */
__IO uint16_t clr_rx_over;         /**< \brief I2C 清除 RX_OVER 中断寄存器 */
    uint16_t reserved18;          /**< \brief 保留 18 */
__IO uint16_t clr_tx_over;         /**< \brief I2C 清除 TX_OVER 中断寄存器 */
    uint16_t reserved19;          /**< \brief 保留 19 */
__IO uint16_t clr_rd_req;          /**< \brief I2C 清除 RD_REQ 中断寄存器 */
    uint16_t reserved20;          /**< \brief 保留 20 */
__IO uint16_t clr_tx_abrt;         /**< \brief I2C 清除 TX_ABRT 中断寄存器 */
    uint16_t reserved21;          /**< \brief 保留 21 */
__IO uint16_t clr_rx_done;         /**< \brief I2C 清除 RX_DONE 中断寄存器 */
    uint16_t reserved22;          /**< \brief 保留 22 */
__IO uint16_t clr_activity;        /**< \brief I2C 清除 ACTIVITY 中断寄存器 */
    uint16_t reserved23;          /**< \brief 保留 23 */
__IO uint16_t clr_stop_det;        /**< \brief I2C 清除 STOP_DET 中断寄存器 */
    uint16_t reserved24;          /**< \brief 保留 24 */
__IO uint16_t clr_start_det;       /**< \brief I2C 清除 START_DET 中断寄存器 */
    uint16_t reserved25;          /**< \brief 保留 25 */
__IO uint16_t clr_gen_call;        /**< \brief I2C 清除 GEN_CALL 中断寄存器 */
    uint16_t reserved26;          /**< \brief 保留 26 */
__IO uint16_t enable;              /**< \brief I2C 使能寄存器 */
    uint16_t reserved27;          /**< \brief 保留 27 */

__IO uint32_t status;              /**< \brief I2C 状态寄存器 */
__IO uint32_t txflr;               /**< \brief I2C 发送缓冲水平寄存器 */
__IO uint32_t rxflr;               /**< \brief I2C 接收缓冲水平寄存器 */
__IO uint32_t sda_hold;            /**< \brief I2C SDA 保持时间寄存器 */
__IO uint32_t tx_abrt_source;      /**< \brief 暂且保留 */
__IO uint32_t slv_data_nack_only;  /**< \brief 暂且保留 */
__IO uint32_t dma_cr;              /**< \brief I2C DMA 控制寄存器 */
__IO uint32_t dma_tdlr;            /**< \brief 暂且保留 */
__IO uint32_t dma_rdlr;            /**< \brief 暂且保留 */
__IO uint32_t sda_setup;           /**< \brief I2C SDA 建立时间寄存器 */
__IO uint32_t ack_general_call;    /**< \brief I2C 广播呼叫 ACK 寄存器 */

__IO uint32_t enable_status;       /**< \brief 暂且保留 */
__IO uint32_t fs_spklen;           /**< \brief 暂且保留 */
__IO uint32_t hs_spklen;           /**< \brief 暂且保留 */

```

```

__IO uint16_t ic_clr_restart_det;    /**< \brief 暂且保留 */
uint16_t reserved28;                /**< \brief 保留 28 */

uint32_t reserved29;                /**< \brief 保留 29 */
__IO uint32_t slvmask;               /**< \brief 从机地址掩码寄存器 */
__IO uint32_t slvrcvaddr;           /**< \brief I2C 广播呼叫 ACK 寄存器 */

} amhw_zmf2x6_i2c_t; /

```

这个小节我们定义了 I2C 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

所有的内联函数都应该在定义时标识为内联函数，在函数的上方加入 `am_static_inline` 的关键字，具体见下文的示例。

11.2.2 编写寄存器的相关函数

1. I2C_CR（控制寄存器）

控制寄存器包括 I2C 主/从机模式、寻址地址位数、工作速率模式、信号产生以及部分中断设置，根据需要的功能编写相应的操作函数。

为了函数参数更加直观和明确，采用宏定义将参数列出，如程序清单 11.5 所示。

程序清单 11.5 I2C_CR 操作函数及参数宏

```

/** \brief 主机模式 */
#define AMHW_ZMF2X6_I2C_MASTER          (1ul | (1ul << 6))

/** \brief 从机模式 */
#define AMHW_ZMF2X6_I2C_SLAVE           (0ul | (0ul << 6))

/** \brief 标准模式(0~100Kbps),主机模式下有效 */
#define AMHW_ZMF2X6_I2C_NORMAL_SPEED    (1ul << 1)

/** \brief 快速模式(<=400Kbps),主机模式下有效 */
#define AMHW_ZMF2X6_I2C_HIGH_SPEED      (2ul << 1)

/** \brief 7 位的寻址地址,从机模式下有效 */
#define AMHW_ZMF2X6_I2C_7BITADDR_SLAVE  (0ul << 3)

/** \brief 10 位的寻址地址,从机模式下有效 */
#define AMHW_ZMF2X6_I2C_10BITADDR_SLAVE (1ul << 3)

/** \brief 7 位的寻址地址,主机模式下有效 */
#define AMHW_ZMF2X6_I2C_7BITADDR_MASTER (0ul << 4)

```

```

/** \brief 10 位的寻址地址,主机模式下有效 */
#define AMHW_ZMF2X6_I2C_10BITADDR_MASTER      (1ul << 4)

/** \brief 主机模式下使能发送 RESTART 条件 */
#define AMHW_ZMF2X6_I2C_RESTART_ENABLE        (1ul << 5)

/** \brief 从机模式下使能产生 STOP_DET 中断 */
#define AMHW_ZMF2X6_I2C_STOP_DET_IFADDRESSED  (1ul << 7)

/**
 * \brief 设置控制寄存器值
 *
 * \param[in] p_hw_i2c : 指向 I2C 寄存器结构体的指针
 * \param[in] flag      : 标志
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_i2c_con_set (amhw_zmf2x6_i2c_t *p_hw_i2c,
                             uint16_t          flag)
{
    p_hw_i2c->cr |= flag;
}

/**
 * \brief 清除控制寄存器值
 *
 * \param[in] p_hw_i2c : 指向 I2C 寄存器结构体的指针
 * \param[in] flag      : 标志
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_i2c_con_clear (amhw_zmf2x6_i2c_t *p_hw_i2c,
                                uint16_t          flag)
{
    p_hw_i2c->cr &= ~((uint32_t)flag);
}

```

2. I2C_TAR（目标地址寄存器）

目标地址寄存器主要用于设置主操作的目标地址，操作函数如程序清单 11.6 所示。

程序清单 11.6 I2C_TAR 操作函数

```

/**
 * \brief 设置主操作的目标地址.

```

```

*
* \param[in] p_hw_i2c : 指向 I2C 寄存器结构体的指针
* \param[in] addr      : 目标地址
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_i2c_tar_set (amhw_zmf2x6_i2c_t *p_hw_i2c,
                             uint16_t          addr)
{
    p_hw_i2c->tar = (p_hw_i2c->tar & ~0x3ff) | (addr & 0x3ff);
}

```

3. I2C_SAR（从机地址寄存器）

从机地址寄存器适用于当 I2C 接口工作在从机模式下，这些存储从机地址对于 7 位的地址格式，只有 SAR[6:0]有效，操作函数如程序清单 11.7 所示。

程序清单 11.7 I2C_SAR 操作函数

```

/**
* \brief 设置从机地址.
*
* \param[in] p_hw_i2c : 指向 I2C 寄存器结构体的指针
* \param[in] addr      : 从机地址
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_i2c_slave_addr_set (amhw_zmf2x6_i2c_t *p_hw_i2c,
                                      uint16_t          addr)
{
    p_hw_i2c->sar = (p_hw_i2c->sar & ~0x3ff) | (addr & 0x3ff);
}

```

4. I2C_DR（数据命令寄存器）

数据命令寄存器主要用于发送或者接收数据，操作函数如程序清单 11.8 所示。

程序清单 11.8 I2C_DR 操作函数

```

/**
* \brief 写数据.
*
* \param[in] p_hw_i2c : 指向 I2C 寄存器结构体的指针
* \param[in] dat      : 数据
*
* \return 无
*/
am_static_inline

```



```

void amhw_zmf2x6_i2c_dat_write (amhw_zmf2x6_i2c_t *p_hw_i2c,
                               uint16_t          dat)
{
    p_hw_i2c->dr = dat;
}

/**
 * \brief 读数据.
 *
 * \param[in] p_hw_i2c : 指向 I2C 寄存器结构体的指针
 *
 * \return 数据
 */
am_static_inline
uint8_t amhw_zmf2x6_i2c_dat_read (amhw_zmf2x6_i2c_t *p_hw_i2c)
{
    return (uint8_t)(p_hw_i2c->dr);
}

```

还有诸多寄存器，可按上述样例逐一添加，I2C 外设寄存器众多，本节不再一一罗列。

11.2.3 添加其他功能性函数

在每个寄存器的操作函数编写完毕后，可能 HW 层的接口还有所欠缺，需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到，可以暂时不用考虑，直接进入驱动层的开发。在后续如果发现 HW 层欠缺哪些接口，可以反过来再增加。当然，也有可能 HW 层接口没有什么欠缺了，那样就无需再开发其它函数了。

11.2.4 添加指向寄存器块的指针

可以看到，在 HW 层编写的函数中，第一个参数均为指向 I2C 寄存器块的指针，这个指针从哪里来呢？只需要把 I2C 的基地址使用(amhw_zmf2x6_i2c_t *)强制转换即可作为这些函数的第一个参数。这样转换后，操作结构体的成员，实际上就是操作具体地址上的寄存器了。

为了方便用户或驱动层使用，可以将该转换使用一个宏定义好，用户直接使用宏即可。在 AMetal 中，由于这些内容与具体型号的芯片相关，这样的宏统一定义到 zmf2x6_periph_map.h 文件中，如程序清单 11.9 所示。

程序清单 11.9 外设中加入 I2C 的定义

```

/** \brief I2C 总线控制器(I2C)寄存器块指针 */
#define ZMF2X6_I2C      ((amhw_zmf2x6_i2c_t      *)ZMF2X6_I2C1_BASE )

```

后续用户或驱动层使用 HW 层接口时，第一个参数直接使用 ZMF2X6_I2C 即可。至此，整个 HW 层的开发工作就结束了，如果后续开发中发现缺少某些函数，还可以再添加。

11.3 驱动层开发

通过对标准接口层文件的分析理解，清楚了驱动层需要为标准接口层提供哪些内容，加上驱动本身应该提供给用户的初始化函数和解初始化函数。总体内容再归纳总结如下：

1. 编写好标准接口层需要使用的 1 个基本函数；
2. 定义 struct am_i2c_drv_funcs 结构体，将其 1 个成员赋值为编写好的 1 个函数；
3. 定义 struct am_i2c_serv 结构体，赋值好相关值，在驱动初始化函数中将其地址返回给用户；

基本分析得差不多了，着手驱动层的开发吧！

驱动层开发主要分为建立驱动层文件和编写函数两个部分进行介绍。

11.3.1 编写结构性程序

驱动层文件有两个，am_zmf2x6_i2c.c 和 am_zmf2x6_i2c.h。存放路径分别为 ametal\soc\zlg\drivers\source\i2c 和 ametal\soc\zlg\drivers\include\i2c。

驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备（I2C 就是一个设备）都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

可以暂时定义设备信息结构体如程序清单 11.10 所示。后续可以根据实际情况修改。

程序清单 11.10 I2C 设备信息结构体定义

```
/**
 * \brief I2C 设备信息参数结构体
 */
typedef struct am_zmf2x6_i2c_devinfo {

    /** \brief I2C 寄存器块基址 */
    uint32_t i2c_regbase;

    /** \brief 时钟 ID */
    int clk_num;

    /** \brief I2C 控制器的中断号 */
    uint16_t inum;

    /** \brief I2C 速率 */
    uint32_t speed;

    /**
     * \brief 超时时间值
     *
     * I2C 总线在数据传输期间，若此段时间内时钟线无任何响应，则视为超时。
     */
}
```

```

    * 结束当前传输并复位内部逻辑
    */
    uint32_t timeout_ms;

    /** \brief I2C 总线恢复函数 */
    void (*pfn_bus_clean)(void);

    /** \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_zmf2x6_i2c_devinfo_t;

```

● 设备结构体定义

由于 I2C 设备需要提供标准的 I2C 服务，因此，该设备至少包含一个 I2C 标准服务结构体成员，也就是 `am_i2c_serv_t` 类型成员，其次，为了在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 11.11 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 11.11 I2C 设备结构体定义

```

/**
 * \brief I2C 设备结构体
 */
typedef struct am_zmf2x6_i2c_dev {

    /** \brief 标准 I2C 服务 */
    am_i2c_serv_t i2c_serv;

    /** \brief I2C 控制器消息队列 */
    struct am_list_head msg_list;

    /** \brief 指向 I2C 传输结构体的指针,同一时间只能处理一个传输 */
    am_i2c_transfer_t *p_cur_trans;

    /** \brief 当前正在处理的消息 */
    am_i2c_message_t *p_cur_msg;

    /** \brief 是否完成本次消息 */
    volatile am_bool_t is_complete;

    /** \brief 用于数据接收/发送计数 */
    volatile uint32_t data_ptr;
}

```

```

/** \brief 当前传输状态（0：此次传输了从机地址，1：此次没有传输从机地址） */
volatile uint8_t                                trans_state;

/** \brief 忙标识 */
volatile am_bool_t                                busy;

/** \brief 状态 */
volatile uint8_t                                state;

/** \brief 从机地址 */
volatile uint16_t                                slave_addr;

/** \brief 超时软件定时器 */
am_softimer_t                                    softimer;

/** \brief 超时软件定时器状态 */
volatile am_bool_t                                softimer_state;

/** \brief 是否中断状态机 */
volatile am_bool_t                                is_abort;

/** \brief 指向 I2C 设备信息的指针 */
const am_zmf2x6_i2c_devinfo_t                    *p_devinfo;
} am_zmf2x6_i2c_dev_t;

```

2. 搭建函数框架

函数主要分为两部分一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于 I2C 这种已经有标准接口层的函数，通常驱动层只提供两个函数给用户，初始化函数和解初始化函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为用于标准接口的 handle。

解初始化函数主要是当不需要再使用某个外设时，释放掉其占用的相关资源，参数为设备指针，无返回值。示例代码如程序清单 11.12 所示。

程序清单 11.12 驱动层函数定义示例

```

/**
 * \brief I2C 初始化
 *
 * \param[in] p_dev      : 指向 I2C 设备结构体的指针
 * \param[in] p_devinfo : 指向 I2C 设备信息结构体的指针
 */

```

```

* \return I2C 标准服务操作句柄
*/
am_i2c_handle_t am_zmf2x6_i2c_init (am_zmf2x6_i2c_dev_t      *p_dev,
                                   const am_zmf2x6_i2c_devinfo_t *p_devinfo);

/**
* \brief 解除 I2C 初始化
*
* \param[in] handle : 与从设备关联的 I2C 标准服务操作句柄
*
* \return 无
*/
void am_zmf2x6_i2c_deinit (am_i2c_handle_t handle);

```

● 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 1 个函数，根据标准接口层中的 struct am_i2c_drv_funcs 结构体，我们需要在.c 文件中定义 1 个函数。

由于这些函数全部是放在结构体中，通过函数指针的形式提供给上层，并不需要将这些函数声明到.h 文件中供用户直接使用，因此全部以 static 的形式定义成内部函数。定义好函数后，再定义一个 struct am_i2c_drv_funcs 结构体，将函数放在其中。示例代码如程序清单 11.13 所示。

程序清单 11.13 提供给标准层函数定义

```

/** \brief I2C 消息处理函数 */
static int __i2c_msg_start (void *p_drv, am_i2c_message_t *p_trans);

/**
* \brief I2C 驱动函数定义
*/
static am_const struct am_i2c_drv_funcs __g_i2c_drv_funcs = {
    __i2c_msg_start
};

```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。

11.3.2 驱动函数实现

通过上面的函数框架搭建，共计 3 个函数，1 个功能性函数面向标准层，2 个初始解初始化函数面向用户，下面逐步实现。

1. am_zmf2x6_i2c_init 函数

该函数主要完成两个功能：

- 初始化设备中的标准服务，将成员赋值，确定 p_drv 的值，p_drv 的值确定了提供给标准接口层的那 1 个函数的第一个参数是什么。那 1 个函数才能根据这儿的设置去使用。（void *类型无法使用，必须转换为具体的一种类型才能使用），通常情况下，p_drv 都是设置为设备的地址，这样方便访问整个设备的相关内容。
- 初始化硬件相关设置，该函数的示例代码如程序清单 11.14 所示。

程序清单 11.14 am_zmf2x6_i2c_init 函数示例代码

```

/**
 * \brief I2C 初始化
 */
am_i2c_handle_t am_zmf2x6_i2c_init (am_zmf2x6_i2c_dev_t      *p_dev,
                                   const am_zmf2x6_i2c_devinfo_t *p_devinfo)
{
    if (p_dev == NULL || p_devinfo == NULL) {
        return NULL;
    }

    p_dev->i2c_serv.p_funcs = (struct am_i2c_drv_funcs *)&__g_i2c_drv_funcs;
    p_dev->i2c_serv.p_drv    = p_dev;

    p_dev->p_devinfo = p_devinfo;

    p_dev->p_cur_trans = NULL;
    p_dev->p_cur_msg    = NULL;
    p_dev->is_complete = AM_FALSE;
    p_dev->data_ptr     = 0;
    p_dev->trans_state = 0;
    p_dev->busy         = AM_FALSE;
    p_dev->state        = __I2C_ST_IDLE; /* 初始化为空闲状态 */
    p_dev->slave_addr   = 0;

    if (p_devinfo->timeout_ms != 0) {
        p_dev->softimer_state = AM_FALSE;
        if (am_softimer_init(&p_dev->softimer,
                             __softimer_callback,
                             p_dev) != AM_OK) {
            return NULL;
        }
    }

    p_dev->is_abort = AM_FALSE;

    if (p_devinfo->pfn_plfm_init) {
        p_devinfo->pfn_plfm_init();
    }

    am_list_head_init(&(p_dev->msg_list));

    __i2c_hard_init(p_dev);
}

```

```

if (p_dev->p_devinfo->pfn_bus_clean) {
    p_dev->p_devinfo->pfn_bus_clean();
}

/* 连接中断 */
am_int_connect(p_dev->p_devinfo->inum, __i2c_irq_handler, (void *)p_dev);
am_int_enable(p_dev->p_devinfo->inum);

return &(p_dev->i2c_serv);
}

```

2. am_zmf2x6_i2c_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放，同时，由于 I2C 一旦使能将无法关闭，所以也无法释放硬件资源。

因此，该函数仅简单的将几个不再使用的变量置为 NULL 即可。示例代码如程序清单 11.15 所示。

程序清单 11.15 am_zmf2x6_i2c_deinit 函数示例代码

```

/**
 * \brief I2C 解初始化
 */
void am_zmf2x6_i2c_deinit (am_i2c_handle_t handle)
{
    amhw_zmf2x6_i2c_t *p_hw_i2c = NULL;
    am_zmf2x6_i2c_dev_t *p_dev = NULL;

    if (NULL == handle) {
        return ;
    }

    p_dev = (am_zmf2x6_i2c_dev_t *)handle;
    p_hw_i2c = (amhw_zmf2x6_i2c_t *)p_dev->p_devinfo->i2c_regbase;

    p_dev->i2c_serv.p_funcs = NULL;
    p_dev->i2c_serv.p_drv = NULL;

    amhw_zmf2x6_i2c_disable (p_hw_i2c);

    /* 去除中断连接 */
    am_int_disable(p_dev->p_devinfo->inum);
    am_int_disconnect(p_dev->p_devinfo->inum,
                     __i2c_irq_handler,
                     (void *)p_dev);
}

```

```

__softimer_stop(p_dev);

if (p_dev->p_devinfo->pfn_plfm_deinit) {
    p_dev->p_devinfo->pfn_plfm_deinit();
}
}

```

3. __i2c_msg_start 函数

该函数主要用于开始消息处理功能，功能实现代码详见程序清单 11.16。

程序清单 11.16 __i2c_msg_start 函数示例代码

```

static int __i2c_msg_start (void *p_drv, am_i2c_message_t *p_msg)
{
    am_zmf2x6_i2c_dev_t *p_dev    = (am_zmf2x6_i2c_dev_t *)p_drv;
    amhw_zmf2x6_i2c_t    *p_hw_i2c = (amhw_zmf2x6_i2c_t *)p_dev->p_devinfo->i2c_regbase;
    int key;

    if ( (p_dev == NULL) ||
        (p_msg == NULL) ||
        (p_msg->p_transfers == NULL) ||
        (p_msg->trans_num == 0) ) {

        return -AM_EINVAL;
    }

    key = am_int_cpu_lock();

    /* 当前正在处理消息，只需要将新的消息加入链表即可 */
    if (p_dev->busy == AM_TRUE) {

        __i2c_msg_in(p_dev, p_msg);

        am_int_cpu_unlock(key);

        return AM_OK;
    } else {
        p_dev->busy = AM_TRUE;
        __i2c_msg_in(p_dev, p_msg);
        p_msg->status = -AM_EISCONN; /* 正在排队中 */
        am_int_cpu_unlock(key);

        /* 清除所有中断 */
        amhw_zmf2x6_i2c_intr_mask_clear(p_hw_i2c, 0xffff);
    }
}

```



```

        return __i2c_mst_sm_event(p_dev, __I2C_EVT_MSG_LAUNCH);
    }
}

```

11.4 用户配置文件编写

11.4.1 定义设备常量于设备信息常量

创建配置文件后，我们需要在里面加入设备变量，设备变量只需要定义，不用初始化。设备信息常量定义时初始化，默认填充的信息是一种常用的配置方式，I2C 设备信息配置如程序清单 11.17 所示。

程序清单 11.17 I2C 设备信息配置文件

```

/**
 * \brief I2C1 设备信息
 */
static const am_zmf2x6_i2c_devinfo_t __g_i2c1_devinfo = {

    ZMF2X6_I2C1_BASE,           /**< \brief I2C1 寄存器块基址 */
    CLK_I2C1,                   /**< \brief 时钟 ID 值 */
    INUM_I2C1,                  /**< \brief I2C1 中断编号 */

    100000,                     /**< \brief I2C 速率 */
    10,                         /**< \brief 超时值 10 */
    __zmf2x6_i2c1_bus_clean,    /**< \brief 总线恢复函数 */
    __zmf2x6_i2c1_plfm_init,    /**< \brief 平台初始化 */
    __zmf2x6_i2c1_plfm_deinit  /**< \brief 平台去初始化 */
};

static am_zmf2x6_i2c_dev_t __g_i2c1_dev;           /**< \brief I2C1 设备实例 */

```

11.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

平台初始化函数和解初始化函数定义如程序清单 11.18 所示。

程序清单 11.18 平台初始化函数和解初始化函数定义

```

/** \brief I2C1 平台初始化函数 */
static void __zmf2x6_i2c1_plfm_init(void)
{

    /**
     * PIOB_6 ~ I2C1_SCL, PIOB_7 ~ I2C1_SDA
     */
    am_gpio_pin_cfg(PIOB_6, PIOB_6_I2C_SCL | PIOB_6_AF_OD | PIOB_6_SPEED_20MHz);
    am_gpio_pin_cfg(PIOB_7, PIOB_7_I2C_SDA | PIOB_7_AF_OD | PIOB_7_SPEED_20MHz);
}

```

```

    am_clk_enable(CLK_I2C1);
    am_zmf2x6_clk_reset(CLK_I2C1);
}

/** \brief 解除 I2C1 平台初始化函数 */
static void __zmf2x6_i2c1_plfm_deinit(void)
{
    am_clk_disable(CLK_I2C1);
}

```

11.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在标准层实例公共头文件 `am_zmf2x6_inst_init.h` 中加入它们的声明，如所示程序清单 11.19。

程序清单 11.19 初始化解初始化声明

```

/**
 * \brief I2C1 实例初始化，获得 I2C 标准服务句柄
 *
 * \param 无
 *
 * \return I2C 标准服务句柄，若为 NULL，表明初始化失败
 */
am_i2c_handle_t am_zmf2x6_i2c1_inst_init(void);

/**
 * \brief I2C1 实例解初始化
 *
 * \param[in] handle 通过 am_zmf2x6_i2c1_inst_init() 函数获得的 I2C 标准服务句柄
 *
 * \return 无
 */
void am_zmf2x6_i2c1_inst_deinit(am_i2c_handle_t handle);

```

用户使用，应该先调用实例初始化函数 `am_zmf2x6_i2c1_inst_init` 函数，初始化 I2C，并保存其返回的 `handle`，后续直接使用保存的 `handle` 作为标准接口的第一个参数使用标准接口函数即可。

至此，用户配置文件开发完毕。

11.5 例程文件开发

例程文件开发主要分为两个部分，HW 例程和 STD 例程。

调试例程时，应该在 `template` 工程下进行，不需要新建其它工程，只需要将新建的文件加入到工程结构的 `user_code` 下。例程文件会与原先的 `main.c` 文件冲突（因为都存在 `main` 函数），这时，只需要调试时将原先的 `main.c` 不要包含在工程编译中，例程文件开发完毕后，再恢复将 `main.c` 包含在工程中，同时将例程文件移除工程即可。

11.5.1 例程内容开发

1. HW 层实现示例

这里使用 HW 层来实现 I2C 轮询模式下操作 EEPROM，相关代码如程序清单 11.20 所示。

程序清单 11.20 I2C 轮询模式下操作 EEPROM

```
/** [src_zmf2x6_core_hw_i2c_master_poll] */
#include "ametal.h"
#include "am_clk.h"
#include "am_gpio.h"
#include "am_vdebug.h"
#include "am_zmf2x6.h"
#include "am_zmf2x6_clk.h"
#include "hw/amhw_zmf2x6_i2c.h"
#include "demo_zmf2x6_entries.h"
#include "demo_zmf2x6_core_entries.h"

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_hw_i2c_master_poll_entry(void)
{
    am_kprintf("demo zmf2x6_core hw i2c master poll!\r\n");

    am_gpio_pin_cfg(PIOB_6, PIOB_6_I2C_SCL | PIOB_6_AF_OD | PIOB_6_SPEED_20MHz);
    am_gpio_pin_cfg(PIOB_7, PIOB_7_I2C_SDA | PIOB_7_AF_OD | PIOB_7_SPEED_20MHz);

    am_clk_enable(CLK_I2C1);
    am_zmf2x6_clk_reset(CLK_I2C1);

    demo_zmf2x6_hw_i2c_master_poll_entry(ZMF2X6_I2C, am_clk_rate_get(CLK_I2C1));
}
/** [src_zmf2x6_core_hw_i2c_master_poll] */

/* end of file */
```

2. 标准层实现示例

这里使用标准层来实现 I2C 主机访问 EEPROM，通过异步标准接口实现，相关代码如程序清单 11.21 所示。

程序清单 11.21 I2C 主机访问 EEPROM

```
#include "ametal.h"
#include "am_vdebug.h"
#include "am_zmf2x6_inst_init.h"
```

```

#include "demo_std_entries.h"
#include "demo_zmf2x6_core_entries.h"

#define EEPROM_ADDR 0x50 /**< \brief EEPROM 设备的地址 */
#define TEST_LEN    8    /**< \brief 操作 EEPROM 的页大小 */

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_std_i2c_master_async_entry (void)
{
    AM_DBG_INFO("demo zmf2x6_core std i2c master async!\r\n");

    demo_std_i2c_master_async_entry(am_zmf2x6_i2c1_inst_init(),
                                    EEPROM_ADDR,
                                    TEST_LEN);
}
/** [src_zmf2x6_core_std_i2c_master_async] */

/* end of file */

```

所有例程开发完毕后，整个外设的开发工作也就圆满结束了。

可以提交相关文件，以完成整个开发任务。提交文件时，不用提交工程文件，只需要提交开发的源文件和头文件即可。

第12章 通用外设驱动开发——SPI

12.1 标准层了解

SPI 标准层包含两个文件 am_spi.h 和 am_spi.c 文件，首先分析一下 am_spi.h 文件，在 ametal\interface 找到该文件打开。

12.1.1 了解相关类型定义

打开该文件，可以看到该文件主要定义了多个标准配置宏，6 个结构体和 1 个 handle 类型，如程序清单 12.1 所示，下面对每个结构体做详细介绍。

程序清单 12.1 标准接口文件类型定义

```
/**
 * \name SPI 模式标志，用于 am_spi_mkdev()函数的 mode 参数
 * @{
 */
#define AM_SPI_CPHA      0x01  /**< \brief 时钟相位可设置 */
#define AM_SPI_CPOL      0x02  /**< \brief 时钟极性可设置 */
#define AM_SPI_CS_HIGH   0x04  /**< \brief 片选高有效 */
#define AM_SPI_LSB_FIRST 0x08  /**< \brief 按低位先出的方式发送数据帧 */
#define AM_SPI_3WIRE      0x10  /**< \brief 3 线模式，SI/SO 信号线共享 */
#define AM_SPI_LOOP      0x20  /**< \brief 回环模式 */
#define AM_SPI_NO_CS     0x40  /**< \brief 单设备总线，无片选 */
#define AM_SPI_READY      0x80  /**< \brief READY 信号,从机拉低此信号暂停传输 */

#define AM_SPI_MODE_0     (0 | 0)          /**< \brief SPI 模式 0 */
#define AM_SPI_MODE_1     (0 | AM_SPI_CPHA) /**< \brief SPI 模式 1 */
#define AM_SPI_MODE_2     (AM_SPI_CPOL | 0) /**< \brief SPI 模式 2 */
#define AM_SPI_MODE_3     (AM_SPI_CPOL | AM_SPI_CPHA) /**< \brief SPI 模式 3 */
/** @} */

/**
 * \name SPI 传输特殊特殊设置标志，用于 am_spi_mktrans()函数的 flags 参数
 * @{
 */
/** \brief SPI 在读取过程中 MOSI 引脚输出高电平（默认为低电平） */
#define AM_SPI_READ_MOSI_HIGH 0x01
/** @} */

/**
 * \brief SPI 控制器信息结构体
 */
typedef struct am_spi_info {
    uint32_t min_speed;  /**< \brief 最小工作频率 */
    uint32_t max_speed;  /**< \brief 最大工作频率 */
}
```

```

    uint32_t features;          /**< \brief SPI 控制器特性,支持的见 “SPI 配置标志” */
} am_spi_info_t;

struct am_spi_device;          /**< \brief 声明 SPI 从机设备的结构体类型 */
struct am_spi_transfer;        /**< \brief 声明 SPI 传输的结构体类型      */
struct am_spi_message;         /**< \brief 声明 SPI 消息的结构体类型      */

/**
 * \brief SPI 驱动函数结构体
 */
struct am_spi_drv_funcs {

    /** \brief 获取 SPI 控制器信息 */
    int (*pfn_spi_info_get)(void *p_drv, am_spi_info_t *p_info);

    /** \brief 设置 SPI 从机设备 */
    int (*pfn_spi_setup)(void *p_drv, struct am_spi_device *p_dev);

    /** \brief 启动 SPI 消息传输, 完成后调用回调函数 */
    int (*pfn_spi_msg_start)(void *p_drv,
                               struct am_spi_device *p_dev,
                               struct am_spi_message *p_msg);
};

/**
 * \brief SPI 标准服务结构体
 */
typedef struct am_spi_serv {
    struct am_spi_drv_funcs *p_funcs;    /**< \brief SPI SPI 驱动函数 */
    void *p_drv;                        /**< \brief SPI SPI 驱动函数参数 */
} am_spi_serv_t;

/** \brief SPI 标准服务操作句柄定义 */
typedef am_spi_serv_t *am_spi_handle_t;

/**
 * \brief SPI 传输结构体
 *
 * - SPI 传输写的个数总是等于读的个数。协议驱动应该总是提供 \a rx_buf 和/或
 * \a tx_buf。在某些情况下, 它们同样可能想提供数据传输的 DMA 地址, 这样的话, 底
 * 层驱动可以使用 DMA, 以降低数据传输的代价。
 */
typedef struct am_spi_transfer {
    const void *p_txbuf;                /**< \brief 指向发送数据缓存的指针 */

```

```

void      *p_rxbuf;          /**< \brief 指向接收数据缓存的指针 */
size_t    nbytes;           /**< \brief 数据长度(字节数) */

/** \brief 字大小，若为 0，则使用 am_spi_device_t 中设置的值 */
uint8_t    bits_per_word;

/** \brief 本传输完成后，是否影响片选: 1-影响 0-不影响 */
uint8_t    cs_change;

/** \brief 在本传输之后，在改变片选状态之前延时的微秒数 */
uint16_t    delay_usecs;

/**
 * \brief 为本次传输选择一个与 am_spi_device_t 中的默认设置不同的速度，
 * 0 表示使用默认值
 */
uint32_t    speed_hz;

/**
 * \brief 本次传输过程的一些特殊设置
 *
 * 比如在读取过程 MOSI 引脚的状态
 */
uint32_t    flags;

/** \brief 链表结点，便于将消息中所有 transfer 组成一条链 */
struct am_list_head trans_node;
} am_spi_transfer_t;

/**
 * \brief SPI 消息结构体
 *
 * 1 个 SPI 消息被用来执行一个原子的数据传输序列，每个传输用 am_spi_transfer 表示。
 * 该序列是原子的，意味着任何其它 SPI 消息不可以使用 SPI 总线，直到该序列执行完成。
 * 在所有系统中，消息是被排队的，发送给某个 am_spi_device_t 的消息总是以 FIFO 的
 * 顺序被处理。
 */
typedef struct am_spi_message {
    struct am_list_head transfers;          /**< \brief 组成消息的传输段 */

    struct am_spi_device *p_spi_dev;       /**< \brief 发送消息的 SPI 从设备 */

    am_pfnvoid_t    pfn_complete;         /**< \brief 传输完成回调函数 */

```

```

void          *p_arg;          /**< \brief 传递给 pfn_complete 的参数 */
size_t        actual_length; /**< \brief 所有成功发送的数据字节总数 */
int           status;          /**< \brief 消息的状态 */

void          *ctrlrdata[2];   /**< \brief 控制器使用，应用不应操作 */
} am_spi_message_t;

/** \brief SPI 从设备描述结构 */
typedef struct am_spi_device {

    am_spi_handle_t  handle;     /**< \brief 该设备关联的 SPI 标准服务 handle */

    /**
     * \brief 数据传输包含 1 个或多个字，比较常见的字大小为 8-bit 或 12-bit，在内存
     * 中，字大小应该是 2 的幂次方，例如，20-bit 自大小应使用 32-bit 内存。本设置可以
     * 被设备驱动改变，或者设置为 0 表示使用默认的“字大小为 8-bit”。
     * “传输”的 am_spi_transfer_t.bits_per_word 可以重载此配置。
     */
    uint8_t        bits_per_word;

    /** \brief 从设备的 SPI 模式标志，请参考“SPI 模式标志” */
    uint16_t        mode;

    /**
     * \brief 设备支持的最高速度
     *
     * “传输”的 am_spi_transfer_t.speed_hz 可重载此设置。
     */
    uint32_t        max_speed_hz;

    /**
     * \brief 片选引脚编号
     *
     * 当 am_spi_device.pfunc_cs 为 NULL 时，才使用此设置。默认低电平有效；若
     * am_spi_device.mode 中 AM_SPI_CS_HIGH 标志被设置，则高电平有效。
     */
    int             cs_pin;

    /**
     * \brief 自定义片选控制函数
     *
     * 使用此函数控制设备的片选：参数 \a state 为 1 时片选有效，为 0 时片选无效。
     * 若此配置为 NULL 时，将使用 am_spi_device.cs_pin 作为片选引脚，且驱动可能
     * 重新设置一个合适的片选控制函数。

```



```

*
* \param p_dev
*/
void      (*pfunc_cs)(struct am_spi_device *p_dev, int state);

} am_spi_device_t;

```

1. am_spi_info 结构体

该结构体包含了 SPI 的信息描述，主要涉及 SPI 最大最小工作频率的信息，其中结构体变量 `features` 为 SPI 控制器的相关特性，其的值为上面宏定义的组合。仅通过标准驱动函数结构体 `pfn_spi_info_get` 来完成信息赋值工作。

2. am_spi_device 结构体

为 SPI 设备结构体，用于描述 SPI 从设备信息，该结构体通过标准函数 `am_spi_mkdev` 构建。包含 SPI 的标准服务句柄 `handle`，以及一些 SPI 的基本信息设置如字大小、SPI 模式、支持的最高速度、片选引脚编号。

该结构体内还定义了一个函数类型 `pfunc_cs`，通过实例化该函数，它的参数 `state` 用于显示需要的状态，1 即为需要片选有效反之则无效。

驱动应支持一个 SPI 模块能够与多个从设备通信。

3. am_spi_transfer 结构体

为 SPI 传输结构体，用于描述 SPI 本次传输的内容信息，该结构体通过标准函数 `am_spi_mktrans` 构建。（在使用中其会通过标准接口 `am_spi_trans_add_tail` 以链表的形式连接到 `am_spi_message` 结构体的 `transfers` 纯链表中。）

4. am_spi_message 结构体

为 SPI 消息结构体，用于描述 SPI 本次传输的，该结构体通过标准函数 `am_spi_msg_init` 构建。

5. am_spi_drv_funcs 结构体

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层需要实现 6 个函数：

- 获取 SPI 控制器信息函数；
- 设置 SPI 从机设备函数；
- 启动 SPI 消息传输，完成后调用回调函数。

因此，在驱动层编写函数时，应该向标准接口层提供这三个函数。

6. am_spi_serv 结构体

该结构体定义了标准的 SPI 服务，包含了上文提到的驱动函数结构体的指针和一个 `void` 型的指针。

这个标准的 SPI 服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

● p_funcs 成员赋值

这是一个指向 `struct am_spi_drv_funcs` 结构体的指针，我们只需要在驱动层定义一个 `struct am_spi_drv_funcs` 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那三个函数地址。如程序清单 12.2 所示。然后，将 `__g_spi_drvfuncs` 的地址赋值给 `p_funcs` 成

员即可。

程序清单 12.2 struct am_spi_drv_funcs 结构体变量

```
/**
 * \brief SPI 驱动函数
 */
am_local am_const struct am_spi_drv_funcs __g_spi_drv_funcs = {
    __spi_info_get,
    __spi_setup,
    __spi_msg_start,
};
```

● p_drv 成员赋值

这个变量是 void 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动层决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

7. am_spi_handle_t 的定义

这个是一个标准的操作句柄，本质上就是 struct am_spi_serv 指针类型，驱动层的初始化函数(am_zmf2x6_spi_init)需要返回该类型的一个 handle 给用户。这样，用户获取到 handle，就可以在需要使用标准接口层函数时，将该 handle 传递给标准接口层了。标准接口就可以通过该 handle（struct am_spi_serv 结构体指针）访问到驱动层提供的函数了。

12.1.2 了解标准接口函数

在 am_spi.h 文件相关的结构体、类型定义后，就是具体函数的实现，如程序清单 12.3 所示。

程序清单 12.3 标准函数实现

```
/**
 * \brief 设置 SPI 设备结构体参数
 *
 * \param[in] p_dev : SPI 从机设备描述符指针
 * \param[in] handle : SPI 标准服务操作句柄
 * \param[in] bits_per_word : 字大小，为 0 时使用默认的“8-bit 字大小”
 * \param[in] mode : 设备模式标志，见“SPI 模式标志”
 * \param[in] max_speed_hz : 设备支持的最大速度
 * \param[in] cs_pin : 片选引脚编号(\a pfunc_cs 为 NULL 时，本参数有效)
 * \param[in] pfunc_cs : 片选控制函数(本参数不为 NULL 时，cs_pin 参数无效)
 *
 * \return 无
 *
 * \par 范例
 * 见 am_spi_setup()
 */
am_static_inline
void am_spi_mkdev(am_spi_device_t *p_dev,
                  am_spi_handle_t handle,
```

```

        uint8_t      bits_per_word,
        uint16_t     mode,
        uint32_t     max_speed_hz,
        int          cs_pin,
        void          (*pfunc_cs)(am_spi_device_t *, int ))
{
    p_dev->handle      = handle;
    p_dev->bits_per_word = bits_per_word;
    p_dev->mode         = mode;
    p_dev->max_speed_hz = max_speed_hz;
    p_dev->cs_pin       = cs_pin;
    p_dev->pfunc_cs     = pfunc_cs;
}

/**
 * \brief SPI 传输结构体参数设置
 *
 * \param[in] p_trans      : 指向 SPI 传输结构体的指针
 * \param[in] p_txbuf      : 指向发送数据缓存的指针,仅接收时应该置为 NULL
 * \param[in] p_rxbuf      : 指向接收数据缓存的指针,仅发送时应该置为 NULL
 * \param[in] nbytes       : 当前传输数据长度, 字节数 (16 位数据时, 字节数应该
 *                          是 2 * N (数据个数))
 * \param[in] cs_change    : 传输结束是否影响片选, 0-不影响, 1-影响
 * \param[in] bits_per_word : 位宽, 为 0 时使用设备中默认的位宽
 * \param[in] delay_usecs   : 传输结束后的延时, 单位: us
 * \param[in] speed_hz      : 传输速率, 为 0 时使用设备中默认的最大传输速率
 * \param[in] flags        : SPI 传输控制标志位, AM_SPI_TRANS_* 的宏值或者
 *                          多个 AM_SPI_TRANS_* 宏的 或 (OR) 值
 *
 * \return 无
 *
 * \note 如果发送和接收数据缓冲区都有效, 务必确保它们的数据长度是一致的。
 *       关于缓冲区的类型和数据传输字节数:
 *       - 1 ~ 8 位宽时, uint8_t 类型, nbytes = 数据传输的个数
 *       - 9 ~ 16 位宽时, uint16_t 类型, nbytes = 2 * 数据传输的个数
 *       - 17 ~ 31 位宽时, uint32_t 类型, nbytes = 4 * 数据传输的个数
 */
am_static_inline
void am_spi_mktrans (am_spi_transfer_t *p_trans,
                    const void      *p_txbuf,
                    void            *p_rxbuf,
                    uint32_t        nbytes,
                    uint8_t         cs_change,
                    uint8_t         bits_per_word,

```

```

        uint16_t      delay_usecs,
        uint32_t      speed_hz,
        uint32_t      flags)
{
    p_trans->p_txbuf      = p_txbuf;
    p_trans->p_rxbuf      = p_rxbuf;
    p_trans->nbytes       = nbytes;
    p_trans->cs_change    = cs_change;
    p_trans->bits_per_word = bits_per_word;
    p_trans->delay_usecs  = delay_usecs;
    p_trans->speed_hz     = speed_hz;
    p_trans->flags        = flags;
}

/**
 * \brief 初始化消息
 *
 * \param[in] p_msg      : SPI 消息描述符指针
 * \param[in] pfn_complete : 传输完成回调函数
 * \param[in] p_arg      : 传递给回调函数的参数
 *
 * \return 无
 */
am_static_inline
void am_spi_msg_init (am_spi_message_t *p_msg,
                     am_pfnvoid_t      pfn_complete,
                     void               *p_arg)
{
    AM_INIT_LIST_HEAD(&p_msg->transfers);

    p_msg->p_spi_dev      = NULL;
    p_msg->actual_length = 0;
    p_msg->pfn_complete   = pfn_complete;
    p_msg->p_arg          = p_arg;
    p_msg->status         = -AM_ENOTCONN;
    p_msg->ctrlrdata[0]   = NULL;
    p_msg->ctrlrdata[1]   = NULL;
}

/**
 * \brief 将传输添加到消息末尾
 *
 * \param[in] p_msg      : 指向消息的指针
 * \param[in] p_trans    : 待加入消息的传输

```

```

*
* \return 无
*/
am_static_inline
void am_spi_trans_add_tail (am_spi_message_t *p_msg,
                           am_spi_transfer_t *p_trans)
{
    am_list_add_tail(&p_trans->trans_node, &p_msg->transfers);
}

/**
* \brief 将传输从消息中删除
* \param[in] p_trans : 待加入删除的传输
* \return 无
* \note 删除前，务必确保该传输确实已经加入了消息中
*/
am_static_inline
void am_spi_trans_del (am_spi_transfer_t *p_trans)
{
    am_list_del(&p_trans->trans_node);
}

/**
* \brief 获取 SPI 控制器的信息
*
* \param[in] handle : SPI 标准服务操作句柄
* \param[out] p_info : 用于获取 SPI 控制器信息的指针
*
* \retval AM_OK : 获取信息成功
* \retval -AM_EINVAL : 获取信息失败, 参数错误
*/
am_static_inline
int am_spi_info_get (am_spi_handle_t handle, am_spi_info_t *p_info)
{
    return handle->p_funcs->pfn_spi_info_get(handle->p_drv, p_info);
}

/**
* \brief 设置 SPI 从机设备
*
* \attention 在 SPI 从机设备被使用前，必须先调用本函数进行设置，且如本函数返回
*           错误，则一定不能再使用此 SPI 从机
*
* \param[in] p_dev : SPI 从机设备

```

```

*
* \retval   AM_OK           : 设置成功
* \retval  -AM_ENOTSUP      : 配置失败，不支持的位宽、模式等
*
* \par 示例
* \code
* am_spi_device_t spi_dev;    // 设备描述结构
* \endcode
*/
am_static_inline
int am_spi_setup (am_spi_device_t *p_dev)
{
    return p_dev->handle->p_funcs->pfn_spi_setup(p_dev->handle->p_drv, p_dev);
}

/**
* \brief SPI 传输函数
*
* \param[in] p_dev    : SPI 从机设备
* \param[in] p_msg    : 传输的消息
*
* \retval   AM_OK      : 传输正常进行处理
* \retval -AM_EINVAL : 输入参数错误
* 传输过程以及结果状态，通过传输回调函数获取状态位
* \li   AM_OK : 传输完成
* \li  -AM_EIO : 传输错误
*/
am_static_inline
int am_spi_msg_start (am_spi_device_t *p_dev,
                     am_spi_message_t *p_msg)
{
    return p_dev->handle->p_funcs->pfn_spi_msg_start(p_dev->handle->p_drv,
                                                    p_dev,
                                                    p_msg);
}

```

根据函数特性我们可以发现前三个函数 `am_spi_mkdev`、`am_spi_mktrans`、`am_spi_msg_init` 仅作为上面所提到的结构体类型 `am_spi_device_t`、`am_spi_transfer_t`、`am_spi_message_t` 的填充功能。

其次是两个链表操作函数，即将传输添加到消息链表末尾和将传输从消息链表中删除，主要用于构建相关数据结构，以便进行后面的状态机操作。

剩下的三个函数，本身就没做什么事情，只是简单的调用了一下驱动层提供的函数而已。可以看到，在调用驱动层提供的函数时，就是将 `struct am_spi_serv` 结构体中的 `p_drv` 成员作

为第一个参数传递。

值得一提的是在 `am_spi.h` 文件最下面还声明了两个函数 `am_spi_write_then_read`、`am_spi_write_then_write` 如程序清单 12.4 所示。

程序清单 12.4 spi 标准服务函数声明

```
/**
 * \brief 先写后读
 *
 * 本函数先进行 SPI 写操作，再进行读操作。
 * 在实际应用中，写缓冲区可以为地址，读缓冲区为要从该地址读取的数据。
 *
 * \param[in] p_dev : SPI 从机设备
 * \param[in] p_txbuf: 数据发送缓冲区
 * \param[in] n_tx : 要发送的数据字节个数
 * \param[out] p_rxbuf: 数据接收缓冲区
 * \param[in] n_rx : 要接收的数据字节个数
 *
 * \retval AM_OK : 消息处理成功
 * \retval -AM_EINVAL: 参数错误
 * \retval -AM_EIO : 传输出错
 */
int am_spi_write_then_read (am_spi_device_t *p_dev,
                           const uint8_t *p_txbuf,
                           size_t n_tx,
                           uint8_t *p_rxbuf,
                           size_t n_rx);

/**
 * \brief 执行两次写操作
 *
 * 本函数连续执行两次写操作，依次发送数据缓冲区 0 和缓冲区 1 中的数据。
 * 在实际应用中，缓冲区 0 可以为地址，缓冲区 1 为要写入该地址的数据。
 *
 * \param[in] p_dev : SPI 从机设备
 * \param[in] p_txbuf0: 数据发送缓冲区 0
 * \param[in] n_tx0 : 缓冲区 0 数据个数
 * \param[in] p_txbuf1: 数据发送缓冲区 1
 * \param[in] n_tx1 : 缓冲区 1 数据个数
 *
 * \retval AM_OK : 消息处理成功
 * \retval -AM_EINVAL: 参数错误
 * \retval -AM_EIO : 传输出错
 */
int am_spi_write_then_write (am_spi_device_t *p_dev,
```

```

const uint8_t  *p_txbuf0,
size_t         n_tx0,
const uint8_t  *p_txbuf1,
size_t         n_tx1);

```

其的实体实际是在 am_spi.c 文件内，相关路径 ametal\components\service\source\am_spi.c。该文件内包含三个函数，如程序清单 12.5 所示。

程序清单 12.5 am_spi.c 文件内容

```

/**
 * \brief SPI 消息完成回调函数
 */
static void __message_complete (void *p_arg)
{
    am_wait_t *p_wait = (am_wait_t *)p_arg;

    /* 结束等待序列 */
    am_wait_done(p_wait);
}

/**
 * \brief 先写后读
 */
int am_spi_write_then_read (am_spi_device_t *p_dev,
                             const uint8_t  *p_txbuf,
                             size_t         n_tx,
                             uint8_t        *p_rxbuf,
                             size_t         n_rx)
{
    am_spi_transfer_t trans[2];
    am_wait_t         msg_wait;
    am_spi_message_t  spi_msg;

    if (p_dev == NULL) {
        return -AM_EINVAL;
    }

    if(p_txbuf == NULL && p_rxbuf == NULL) {
        return AM_OK;
    }

    am_wait_init(&msg_wait);

    am_spi_msg_init(&spi_msg,
                    __message_complete,

```



```

        (void *)&msg_wait);

if ((p_txbuf != NULL) && (n_tx != 0)) {

    am_spi_mktrans(&trans[0],
                   p_txbuf,
                   NULL,
                   n_tx,
                   0,
                   0,
                   0,
                   0,
                   0);

    am_spi_trans_add_tail(&spi_msg, &trans[0]);
}

if ((p_rxbuf != NULL) && (n_rx != 0)) {

    am_spi_mktrans(&trans[1],
                   NULL,
                   p_rxbuf,
                   n_rx,
                   0,
                   0,
                   0,
                   0,
                   0);

    am_spi_trans_add_tail(&spi_msg, &trans[1]);
}

am_spi_msg_start(p_dev, &spi_msg);

am_wait_on(&msg_wait);

return spi_msg.status;
}

/**
 * \brief 执行两次写操作
 */
int am_spi_write_then_write (am_spi_device_t *p_dev,
                             const uint8_t *p_txbuf0,
```

```

                                size_t      n_tx0,
                                const uint8_t *p_txbuf1,
                                size_t      n_tx1)
{
    am_spi_transfer_t trans[2];
    am_wait_t          msg_wait;
    am_spi_message_t   spi_msg;

    if (p_dev == NULL) {
        return -AM_EINVAL;
    }

    if(p_txbuf0 == NULL && p_txbuf1 == NULL) {
        return AM_OK;
    }

    am_wait_init(&msg_wait);

    am_spi_msg_init(&spi_msg,
                    __message_complete,
                    (void *)&msg_wait);

    if ((p_txbuf0 != NULL) && (n_tx0 != 0)) {

        am_spi_mktrans(&trans[0],
                       p_txbuf0,
                       NULL,
                       n_tx0,
                       0,
                       0,
                       0,
                       0,
                       0);

        am_spi_trans_add_tail(&spi_msg, &trans[0]);
    }

    if ((p_txbuf1 != NULL) && (n_tx1 != 0)) {

        am_spi_mktrans(&trans[1],
                       p_txbuf1,
                       NULL,
                       n_tx1,
                       0,

```

```

        0,
        0,
        0,
        0);

    am_spi_trans_add_tail(&spi_msg, &trans[1]);
}

am_spi_msg_start(p_dev, &spi_msg);

am_wait_on(&msg_wait);

return spi_msg.status;
}

```

可以发现它其实是标准函数的一个应用，简化了使用 SPI 标准接口的步骤。由于它的普适性我们称它为标准服务。通过这些函数（`am_spi_write_then_write`、`am_spi_write_then_read`）用户能够方便快捷的实现 SPI 通信相关需求。

到此我们就分析完了 SPI 外设标准层文件，清楚了驱动层需要为标准接口层提供哪些内容。

12.2 HW 层开发

通过上一节的内容，我们已经知道了标准层实现的函数的功能，以及需要驱动层提供的函数，但驱动层并不是凌空搭建的，驱动层介乎与 HW 层和标准接口层之间，简单来说就是调用 HW 层的函数来提供给驱动层，所以在写驱动层前，我们先来完成 HW 层。

HW 层对应到具体的外设，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，内联函数直接定义在.h 文件中。硬件层所有的符号命名以 `amhw_`/`AMHW_` 开头。通常情况下，HW 层只有.h 文件，只有当某些硬件功能设置较为复杂时，才提供对应的非内联函数，存放在.c 文件中。

现在我们打开 ZMF2x6 芯片用户手册，先了解一下该芯片 SPI 外设，下面对照手册完成 SPI 外设的 HW 层的开发。其对应文件路径为 `{sdk}\soc\zlg\drivers\include\spi\hw`。

12.2.1 添加寄存器列表

HW 层就是与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读该外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMF2x6 用户手册的 SPI 章节，可以看到，SPI 章节部分，共有 12 个寄存器，简单描述如下：

- SPI_TXREG（发送数据寄存器）：用于存储 SPI 要发送的数据，其有效数据位由 SPI_GCTL 寄存器的 DW8_32 位控制；
- SPI_RXREG（接收数据寄存器）：用于存储 SPI 接收到的数据，其有效数据位由 SPI_GCTL 寄存器的 DW8_32 位控制；
- SPI_CSTAT（当前状态寄存器）：用于描述当前 SPI 模块的整体状态。涉及接收缓冲器中有效数据个数位、发送缓冲器中有效数据个数位、接收缓冲器中有效数据达

到 4 个字节标志位、发送缓冲器满标志位、接收有效字节数据信息位、发送端空位；

- **SPI_INTSTAT**（中断状态寄存器）：用于描述当前 SPI 模块的整体状态。涉及发送端空中断、接收端缓冲器满中断、接收指定字节数中断、接收端溢出错误中断、SPI 从机模式下溢中断、接收端数据有效中断、发送缓冲器有效中断的标志位信息。
- **SPI_INTEN**（中断使能寄存器）：用于控制 SPI 模块中断的使能和禁能。涉及到发送端空中断、接收端缓冲器满中断、接收指定字节数中断、接收端溢出错误中断、SPI 从机模式下溢中断、接收端数据有效中断、发送缓冲器有效中断的控制。
- **SPI_INTCLR**（中断清除寄存器）：用于清除 SPI 模块中断标志位。涉及到发送端空中断、接收端缓冲器满中断、接收指定字节数中断、接收端溢出错误中断、SPI 从机模式下溢中断、接收端数据有效中断、发送缓冲器有效中断清除操作；
- **SPI_GCTL**（全局控制寄存器）：涉及到从设备选择信号自动翻转、发送和接收数据寄存器有效数据选择、硬件或软件控制主模式下的 NSS 输出、接收和发送的 DMA 模式使能、发送缓冲器触发 DMA 请求的边沿选择、接收缓冲器触发 DMA 请求的边沿选择、接收使能位、发送使能位、主机模式位、SPI 中断使能位、SPI 选择位。
- **SPI_CCTL**（通用控制寄存器）：涉及到 CPHA 极性取反选择、发送数据相位调整位（从模式）、接收数据采样时钟沿选择位（主模式）、SPI 数据宽度位、LSB 在前使能位、时钟极性标志位、时钟相位选择位。
- **SPI_SPBRG**（波特率发生器）：SPI 波特率控制寄存器用于产生波特率。波特率 = $fpclk/SPBRG(fpclk/是 APB 时钟频率)$ 。
- **SPI_RXDNR**（接收数据个数寄存器）：该寄存器用于存储下次接收过程需要接收字节的个数（仅在 SPI 为主机接收模式下有效）。
- **SPI_NSSR**（从机片选寄存器）：主模式下片选输出信号。低有效，从模式下该位无效。
- **SPI_EXTCTL**（数据控制寄存器）：用于控制 SPI 数据长度（1~32bit），仅当 SPI_GCTL 寄存器 DW8_32 位为 ‘0’ 时有效，且 SPI_CCTL 寄存器的 LSBFE 位必须配置为 ‘1’，SPILEN 位也必须配置为 ‘1’。

到此 SPI 的各个寄存器及其功能基本阅读完毕，我们也大致了解了各个寄存器功能以及要配置各个功能，其他寄存器该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，如程序清单 12.6 所示。

程序清单 12.6 SPI 寄存器结构体定义

```
/**
 * \brief SPI structure of register
 */
typedef struct amhw_zmf2x6_spi {
    __IO uint32_t txreg;      /**< \brief SPI TX data register */
    __I uint32_t rxreg;      /**< \brief SPI RX data register */
    __I uint32_t cstat;      /**< \brief SPI current status register */
    __I uint32_t intstat;    /**< \brief SPI interrupt status register */
    __IO uint32_t inten;     /**< \brief SPI interrupt enable register */
}
```

```

__IO uint32_t  intclr;      /**< \brief SPI interrupt clear register */
__IO uint32_t  gctl;       /**< \brief SPI global control */
__IO uint32_t  cctl;       /**< \brief SPI common control */
__IO uint32_t  spbrg;      /**< \brief SPI baud rate register */
__IO uint32_t  rxdnr;      /**< \brief SPI RX data number */
__IO uint32_t  nssr;       /**< \brief SPI slave selection register */
__IO uint32_t  extctl;     /**< \brief SPI data control */
} amhw_zmf2x6_spi_t;

```

寄存器定义时，修饰宏__IO 表示该寄存器是可读可写的，用户手册寄存器描述为 rw；__O 表示只写寄存器，用户手册描述为 w，__I 表示只读寄存器，用户手册描述为 r。不同原厂手册对此可能不完全一致，但该方法通用，即可读可写、只读、只写三种类型。

所有的函数都应该在定义时标识为内联函数，在函数的上方加入 am_static_inline 的关键字，具体见下文的示例。

12.2.2 编写寄存器相关函数

1. SPI_TXREG（发送数据寄存器）

该寄存器仅一个功能，即用于存储 SPI 要发送的数据。由于其能够通过 SPI_GCTL 寄存器的 DW8_32 位设置其的有效位数，因此我们可以列出两个函数，即 8 位数据写、32 位数据写。如程序清单 12.7 所示

程序清单 12.7 发送数据函数

```

/**
 * \brief 8 bit data write
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] data      : data to write
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_tx_data8_put (amhw_zmf2x6_spi_t *p_hw_spi, uint8_t data)
{
    p_hw_spi->txreg = data;
}

/**
 * \brief 32 bit data write
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] data      : data to write
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_tx_data32_put (amhw_zmf2x6_spi_t *p_hw_spi, uint32_t data)
{

```

```
p_hw_spi->txreg = data;
}
```

2. SPI_RXREG（接收数据寄存器）

该寄存器仅一个功能，即用于存储 SPI 接收到的数据。由于其能够通过 SPI_GCTL 寄存器的 DW8_32 位设置其的有效位数，因此我们可以列出两个函数，即 8 位数据写、32 位数据写。如程序清单 12.8 所示

程序清单 12.8 接收数据函数

```
/**
 * \brief 8 bit data read
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 *
 * \return data
 */
am_static_inline
uint8_t amhw_zmf2x6_spi_rx_data8_get (amhw_zmf2x6_spi_t *p_hw_spi)
{
    return (p_hw_spi->rxreg & 0xff);
}

/**
 * \brief 32 bit data read
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \return none
 */
am_static_inline
uint32_t amhw_zmf2x6_spi_rx_data32_get (amhw_zmf2x6_spi_t *p_hw_spi)
{
    return p_hw_spi->rxreg;
}
```

3. SPI_CSTAT（当前状态寄存器）

通过手册，我们了解到该寄存器有含义的寄存器共有 6 个，如错误!未找到引用源。所示。

表 12.1 SPI_CSTAT 寄存器分析

位	标识符	含义
8:11	RXFADDR	涉及接收缓冲器中有效数据个数位
4:7	TXFADDR	发送缓冲器中有效数据个数位
3	RXAVL_4BYTE	接收缓冲器中有效数据达到 4 个字节标志位
2	TXFULL	发送缓冲器满标志位
1	RXAVL	接收有效字节数据信息位，当接收端缓冲器接收了一个完整字节的数据

		时置位该位
0	TXEPT	发送端空标志位

经过查看该状态寄存器主要使用到的状态位为 0~3 位, 将该四位状态位定义为宏的形式, 通过一个函数进行判断, 如程序清单 12.9 所示。

程序清单 12.9 SPI 状态获取函数

```
/**
 * \brief current status
 * @{
 */
#define AMHW_ZMF2X6_SPI_CSTAT_RXVAL_4BYTE (1U << 3) /**< \brief Receive available 4 byte data */
#define AMHW_ZMF2X6_SPI_CSTAT_TX_FULL (1U << 2) /**< \brief Transmitter FIFO full status bit */
#define AMHW_ZMF2X6_SPI_CSTAT_RXVAL (1U << 1) /**< \brief Receive available byte data message */
#define AMHW_ZMF2X6_SPI_CSTAT_TX_EMPTY (1U << 0) /**< \brief Transmitter empty bit */
/** @} */

/**
 * \brief current status get
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 *
 * \return the value of CSTAT
 */
am_static_inline
uint32_t amhw_zmf2x6_spi_reg_cstat_get(amhw_zmf2x6_spi_t *p_hw_spi)
{
    return p_hw_spi->cstat;
}
```

4. SPI_INTSTAT、SPI_INTEN、SPI_INTCLR（中断相关寄存器）

对于中断相关内容我们会把他们定义成相关的宏变量, 在后期不管是中断的使能, 状态的获取还是中断位的清除都通过这个宏来作为参数判断是哪个中断。相关函数如程序清单 12.10 所示。

程序清单 12.10 SPI 中断控制相关函数

```
/**
 * \brief interrupt flag
 * @{
 */
#define AMHW_ZMF2X6_SPI_INTSTAT_TXEPT_FALG (1U << 6) /**< \brief Transmitter empty interrupt flag */
#define AMHW_ZMF2X6_SPI_INTSTAT_RX_FULL_FALG (1U << 5) /**< \brief RX FIFO full interrupt flag */
```

```

#define AMHW_ZMF2X6_SPI_INTSTAT_RX_MATCH_FALG (1U << 4) /**< \brief Receive data match */
#define AMHW_ZMF2X6_SPI_INTSTAT_RX_OERR_FLAG   (1U << 3) /**< \brief Receive overrun error
interrupt flag */
#define AMHW_ZMF2X6_SPI_INTSTAT_UNDERR_FALG    (1U << 2) /**< \brief underrun interrupt flag */
#define AMHW_ZMF2X6_SPI_INTSTAT_RX_FALG        (1U << 1) /**< \brief Receive data available
interrupt flag */
#define AMHW_ZMF2X6_SPI_INTSTAT_TX_FALG        (1U << 0) /**< \brief Transfer FIFO available
interrupt flag */
#define AMHW_ZMF2X6_SPI_INTSTAT_INT_ALL        (0X7F)    /**< \brief All */

/** @ */

/**
 * \brief interrupt  status get
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 *
 * \return the value of INTSTAT
 */
am_static_inline
uint32_t amhw_zmf2x6_spi_reg_intstat_get (amhw_zmf2x6_spi_t *p_hw_spi)
{
    return p_hw_spi->intstat;
}

/**
 * \brief interrupt  enable
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : see the field of interrupt flag
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_int_enable (amhw_zmf2x6_spi_t *p_hw_spi, uint32_t flag)
{
    p_hw_spi->inten |= flag;
}

/**
 * \brief interrupt  disable
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : see the field of interrupt flag
 *
 * \return none
 */

```



```

am_static_inline
void amhw_zmf2x6_spi_int_disable (amhw_zmf2x6_spi_t *p_hw_spi, uint32_t flag)
{
    p_hw_spi->inten &= ~flag;
}

/**
 * \brief interrupt flag clear
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : see the field of interrupt flag
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_int_flag_clr (amhw_zmf2x6_spi_t *p_hw_spi, uint32_t flag)
{
    p_hw_spi->intclr |= flag;
}

```

5. SPI_GCTL（全局控制寄存器）

通过手册，我们了解到该寄存器有含义的寄存器共有 10 个，如表 12.2 所示。

表 12.2 SPI_SPIFG 寄存器分析

位	标识符	含义
12	NSSTOG	从设备选择信号自动翻转，该位只在主模式下有效
11	DW8_32	发送和接收数据寄存器有效数据选择
10	NSS	硬件或软件控制主模式下的 NSS 输出
9	DMAEN	接收和发送的 DMA 模式使能
7~8	TXTLF	发送缓冲器触发 DMA 请求的边沿选择
5~6	RXTLF	接收缓冲器触发 DMA 请求的边沿选择
4	RXEN	接收使能位
3	TXEN	发送使能位
2	MODE	主机模式位
1	INTEN	SPI 中断使能位
0	SPIEN	SPI 选择位

由此我们确定编写 10 个函数：发送和接收数据寄存器有效数据选择、硬件或软件控制主模式下的 NSS 输出、接收和发送的 DMA 模式使能、发送缓冲器触发 DMA 请求的边沿选择、接收缓冲器触发 DMA 请求的边沿选择、接收使能位、发送使能位、主机模式位、SPI 中断使能位、SPI 选择位。

程序清单 12.11 为 SPI 发送和接收数据寄存器有效数据选择函数，定义了两个宏分别表示有效数据为 8bit 和 32bit。

程序清单 12.11 SPI 有效数据位选择函数

```

/**
 * \brief valid data selection
 * @{
 */

#define AMHW_ZMF2X6_SPI_VALID_DATA_8BIT    (0U << 11) /**< \brief valid data is low 8 bit */
#define AMHW_ZMF2X6_SPI_VALID_DATA_32BIT   (1U << 11) /**< \brief valid data is 32 bit */

/** @} */

/**
 * \brief valid data selection
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : see the field of valid data selection
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_valid_data_sel(amhw_zmf2x6_spi_t *p_hw_spi, uint32_t flag)
{
    p_hw_spi->gctl = (p_hw_spi->gctl & ~(1u << 11)) | flag;
}

```

程序清单 12.12 为 SPI 硬件或软件控制主模式下的 NSS 输出模式函数，定义两个宏分别表示硬件自动控制 NSS 和软件寄存器控制。

程序清单 12.12 NSS 输出模式选择函数

```

/**
 * \brief NSS control selection
 * @{
 */

#define AMHW_ZMF2X6_SPI_NSS_CTRL_BY_REG    (0U << 10) /**< \brief NSS control by register */
#define AMHW_ZMF2X6_SPI_NSS_CTRL_BT_HD    (1U << 10) /**< \brief NSS control by hardware */

/** @} */

/**
 * \brief NSS control selection
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : see the field of NSS control selection
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_nss_ctrl_sel(amhw_zmf2x6_spi_t *p_hw_spi, uint32_t flag)
{

```

```

p_hw_spi->gctl = (p_hw_spi->gctl & ~(1u << 10)) | flag;
}

```

程序清单 12.13 为 SPI 接收和发送的 DMA 模式使能函数。

程序清单 12.13 SPI 接收和发送的 DMA 模式使能函数

```

/**
 * \brief DMA enable
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : TRUE or FALSE
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_dma_enable (amhw_zmf2x6_spi_t *p_hw_spi, am_bool_t flag)
{
    p_hw_spi->gctl = (p_hw_spi->gctl & ~(1u << 9)) | (flag << 9);
}

```

程序清单 12.14 为 SPI 发送缓冲器触发 DMA 请求的边沿选择函数。

程序清单 12.14 SPI 发送缓冲器 DMA 边沿触发选择函数

```

/**
 * \brief DMA TX trigger level
 * @{}
 */
#define AMHW_ZMF2X6_SPI_DMA_TRIG_ONE_DATA    (0U << 7) /**< \brief DMA trigger when FIFO has
one data idle space */
#define AMHW_ZMF2X6_SPI_DMA_TRIG_HALF_DATA   (1U << 7) /**< \brief DMA trigger when FIFO has
half data idle space */

/** @} */

/**
 * \brief DMA TX trigger level selection
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : see the field of DMA trigger level
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_dma_txtri_level_sel (amhw_zmf2x6_spi_t *p_hw_spi,
                                           uint32_t          flag)
{

```

```

    p_hw_spi->gctl = (p_hw_spi->gctl & ~(3u << 7)) | flag;
}

/** @*/

```

程序清单 12.15 为 SPI 接收缓冲器触发 DMA 请求的边沿选择函数。

程序清单 12.15 SPI 接收缓冲器触发 DMA 请求的边沿选择函数

```

/**
 * \brief DMA RX trigger level selection
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : see the field of DMA trigger level
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_dma_rxtri_level_sel (amhw_zmf2x6_spi_t *p_hw_spi,
                                           uint32_t          flag)
{
    p_hw_spi->gctl = (p_hw_spi->gctl & ~(3u << 5)) | flag;
}

```

程序清单 12.16 为 SPI 接收使能函数。

程序清单 12.16 SPI 接收使能函数

```

/**
 * \brief RX enable
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : TRUE or FALSE
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_rx_enable (amhw_zmf2x6_spi_t *p_hw_spi, am_bool_t flag)
{
    p_hw_spi->gctl = (p_hw_spi->gctl & ~(1u << 4)) | (flag << 4);
}

```

程序清单 12.17 为 SPI 发送使能函数。

程序清单 12.17 SPI 发送使能函数

```

/**
 * \brief TX enable
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : TRUE or FALSE

```

```

*
* \return none
*/
am_static_inline
void amhw_zmf2x6_spi_tx_enable (amhw_zmf2x6_spi_t *p_hw_spi, am_bool_t flag)
{
    p_hw_spi->gctl = (p_hw_spi->gctl & ~(1u << 3)) | (flag << 3);
}

```

程序清单 12.18 为 SPI 模式选择函数函数，模式涉及到主模式以及从模式，此处使用两个宏来代表它们。

程序清单 12.18 SPI 模式选择函数函数

```

/**
 * \brief SPI mode selection
 * \ @{
 */
#define AMHW_ZMF2X6_SPI_MODE_MASTER    (1U << 2) /**< \brief Master mode */
#define AMHW_ZMF2X6_SPI_MODE_SLAVE    (0U << 2) /**< \brief Slave mode */

/** @} */

/**
 * \brief mode selection
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : see the field of SPI mode selection
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_mode_sel (amhw_zmf2x6_spi_t *p_hw_spi, uint32_t flag)
{
    p_hw_spi->gctl = (p_hw_spi->gctl & ~(1u << 2)) | flag;
}

```

程序清单 12.19 为 SPI 全局中断使能函数。

程序清单 12.19 SPI 全局中断使能函数

```

/**
 * \brief Interrupt enable
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : TRUE or FALSE
 *
 * \return none

```

```

*/
am_static_inline
void amhw_zmf2x6_spi_int_open (amhw_zmf2x6_spi_t *p_hw_spi, am_bool_t flag)
{
    p_hw_spi->gctl = (p_hw_spi->gctl & ~(1u << 1)) | (flag << 1);
}

```

程序清单 12.20 为 SPI 模块使能禁能函数。

程序清单 12.20 SPI 模块使能或禁能函数

```

/**
 * \brief SPI module enable
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] flag      : TRUE or FALSE
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_spi_module_enable (amhw_zmf2x6_spi_t *p_hw_spi, am_bool_t flag)
{
    p_hw_spi->gctl = (p_hw_spi->gctl & ~(1u << 0)) | flag;
}

```

6. SPI_SPBRG（波特率发生器）

SPI 波特率控制寄存器用于产生波特率，波特率 = $fpclk/SPBRG$ ($fpclk$ 是 APB 时钟频率)。如程序清单 12.21 所示为 SPI 波特率寄存器设置及获取函数。

程序清单 12.21 SPI 波特率寄存器设置及获取函数

```

/**
 * \brief SPI baud rate get
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 *
 * \return baud rate
 */
am_static_inline
uint16_t amhw_zmf2x6_spi_baud_rate_get (amhw_zmf2x6_spi_t *p_hw_spi)
{
    return (p_hw_spi->spbrg & 0xffffu);
}

/**
 * \brief SPI baud rate set
 * \param[in] p_hw_spi : The pointer to the block of SPI register
 * \param[in] baud_rate : 0~65535
 *

```

```

* \return none
*/
am_static_inline
void amhw_zmf2x6_spi_baud_rate_set (amhw_zmf2x6_spi_t *p_hw_spi,
                                   uint32_t          baud_rate)
{
    p_hw_spi->spbrg = baud_rate & 0xffffu;
}

```

12.2.3 添加其他功能性函数

在每个寄存器的操作函数编写完毕后，可能 HW 层的接口还有所欠缺，需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到，可以暂时不用考虑，直接进入驱动层的开发。在后续如果发现 HW 层欠缺哪些接口，可以反过来再增加。当然，也有可能 HW 层接口没有什么欠缺了，那样就无需再开发其它函数了。

12.2.4 添加指向寄存器块的指针

可以看到，在 HW 层编写的函数中，第一个参数均为指向 SPI 寄存器块的指针，这个指针从哪里来呢？只需要把 SPI 的基地址使用(amhw_spi_t *)强制转换即可作为这些函数的第一个参数。这样转换后，操作结构体的成员，实际上就是操作具体地址上的寄存器了。

但如果每次使用都要这样转换的话就太麻烦了，为了方便用户或驱动层使用，可以将该转换使用一个宏定义好，用户直接使用宏即可。在 AMetal 中，由于这些内容与具体型号的芯片相关，这样的宏统一定义到 zmf2x6_periph_map.h 文件中，如程序清单 12.22 所示。

程序清单 12.22 外设中加入 SPI 的定义

```

/**
 * \name 外设定义
 * @{
 */
/** \brief 串口(UART2)寄存器块指针 */
#define ZMF2X6_UART2    ((amhw_zmf2x6_uart_t *)ZMF2X6_UART2_BASE)

/** \brief 电源管理单元(PWR)寄存器块指针 */
#define ZMF2X6_PWR      ((amhw_zmf2x6_pwr_t *)ZMF2X6_PWR_BASE)

/** \brief 备份区域(BKP)寄存器块指针 */
#define ZMF2X6_BKP      ((amhw_zmf2x6_bkp_t *)ZMF2X6_BKP_BASE)

/** \brief 模数转换(SPI1)寄存器块指针 */
#define ZMF2X6_SPI1     ((amhw_zmf2x6_spi_t *)ZMF2X6_SPI1_BASE)

/** \brief 模数转换(SPI2)寄存器块指针 */
#define ZMF2X6_SPI2     ((amhw_zmf2x6_spi_t *)ZMF2X6_SPI2_BASE)

/** \brief I2C 总线控制器(I2C0)寄存器块指针 */

```

```
#define ZMF2X6_I2C      ((amhw_zmf2x6_i2c_t      *)ZMF2X6_I2C1_BASE )
/**
 * @} zmf2x6_if_periph_map
 */
```

后续用户或驱动层使用 HW 层接口时，第一个参数直接使用 ZMF2X6_SPI 即可。至此，整个 HW 层的开发工作就结束了，如果后续开发中发现缺少某些函数，还可以再添加。

12.3 驱动层开发

对于 SPI 驱动层的开发一般情况下会创建多个文件，即 am_zmf2x6_spi_poll.h、am_zmf2x6_spi_poll.c、am_zmf2x6_spi_int.h、am_zmf2x6_spi_int.c、am_zmf2x6_spi_dma.c、am_zmf2x6_spi_dma.h，此处以 SPI 的中断为例，介绍其驱动的开发过程。路径分别位于 {sdk}\ametal\soc\zlg\drivers\source\spi\am_zmf2x6_spi_int.c; {sdk}\ametal\soc\zlg\drivers\include\spi\am_zmf2x6_spi_int.h。

12.3.1 编写结构性程序

驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备（SPI 就是一个设备）都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

可以暂时定义设备信息结构体如程序清单 12.23 所示。后续可以根据实际情况修改。

程序清单 12.23 SPI 设备信息结构体定义

```
/**
 * \brief SPI 设备信息结构体
 */
typedef struct am_zmf2x6_spi_int_devinfo {
    uint32_t spi_reg_base;    /**< \brief SPI 寄存器块的基地址 */
    int      clk_id;          /**< \brief 时钟 ID */
    uint16_t inum;            /**< \brief SPI 中断编号 */

    uint32_t flags;           /**< \brief SPI MOSI 引脚配置标识 */
    uint16_t mosi_pin;        /**< \brief mosi 引脚号 */

    /** \brief SPI 平台初始化函数，如打开时钟，配置引脚等工作 */
    void      (*pfn_plfm_init)(void);

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



```
} am_zmf2x6_spi_int_devinfo_t;
```

● 设备结构体定义

由于 SPI 设备需要提供标准的 SPI 服务，因此，该设备至少包含一个 SPI 标准服务结构体成员，也就是 `am_spi_serv_t` 类型成员，其次，为了在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

另外 SPI 驱动会使用到消息队列，因此我们需要定义相关成员变量来处理相关操作。

暂时定义设备信息如程序清单 12.24 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 12.24 SPI 设备结构体定义

```
/**
 * \brief SPI 设备
 */
typedef struct am_zmf2x6_spi_int_dev {

    am_spi_serv_t          spi_serv;    /**< \brief SPI 标准服务句柄 */
    const am_zmf2x6_spi_int_devinfo_t *p_devinfo; /**< \brief SPI 设备信息的指针 */

    struct am_list_head    msg_list;    /**< \brief SPI 控制器消息队列 */

    /** \brief 指向 SPI 消息结构体的指针,同一时间只能处理一个消息 */
    am_spi_message_t       *p_cur_msg;

    /** \brief 指向 SPI 传输结构体的指针,同一时间只能处理一个传输 */
    am_spi_transfer_t      *p_cur_trans;

    am_spi_device_t        *p_cur_spi_dev;    /**< \brief 当前传输的 SPI 设备 */
    am_spi_device_t        *p_tgl_dev;        /**< \brief 当前自锁的 SPI 设备 */

    uint32_t                nbytes_to_recv;    /**< \brief 待接收的字节数 */
    uint32_t                data_ptr;          /**< \brief 数据传输计数 */

    am_bool_t              busy;               /**< \brief SPI 忙标识 */
    uint32_t               state;              /**< \brief SPI 控制器状态机状态 */

} am_zmf2x6_spi_int_dev_t;
```

2. 搭建函数框架

函数主要分为两部分一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于 SPI 这种已经有标准接口层的函数，通常驱动层只提供两个函数给用户，初始化函数和解初始化函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为用于标准接口的 `handle`。

解初始化函数主要是当不需要再使用某个外设时，释放掉其占用的相关资源，参数为设备指针，无返回值。示例代码如程序清单 12.25 所示。

程序清单 12.25 驱动层函数定义示例

```
/** \brief 初始化 SPI，获取标准服务句柄 */
am_spi_handle_t am_zmf2x6_spi_int_init (am_zmf2x6_spi_int_dev_t      *p_dev,
                                       const am_zmf2x6_spi_int_devinfo_t *p_devinfo)
{
    return &(p_dev->spi_serv);
}

/** \brief SPI 解初始化 */
void am_zmf2x6_spi_int_deinit (am_spi_handle_t handle)
{
}
```

同时为了用户能够使用，还需要将这两个函数的声明放入到相应的头文件中。示例代码如程序清单 12.26 所示。

程序清单 12.26 驱动函数声明

```
/**
 * \brief SPI 初始化
 *
 * \param[in] p_dev      : 指向 SPI 设备的指针
 * \param[in] p_devinfo : 指向 SPI 设备信息的指针
 *
 * \return SPI 标准服务操作句柄 如果为 NULL，表明初始化失败
 */
am_spi_handle_t am_zmf2x6_spi_int_init (am_zmf2x6_spi_int_dev_t      *p_dev,
                                       const am_zmf2x6_spi_int_devinfo_t *p_devinfo);

/**
 * \brief SPI 去初始化
 *
 * \param[in] handle : am_zmf2x6_spi_int_init() 初始化函数获得的 SPI 服务句柄
 *
 * \return 无
 */
void am_zmf2x6_spi_int_deinit (am_spi_handle_t handle);
```

- 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供三个函数，根据标准接口层中的 struct am_spi_drv_funcs 结构体，我们需要在.c 文件中定义 3 个函数。

由于这些函数全部是放在结构体中，通过函数指针的形式提供给上层，并不需要将这些函数声明到.h 文件中供用户直接使用，因此全部以 static 的形式定义成内部函数。定义好函数后，再定义一个 struct am_spi_drv_funcs 结构体，将函数放在其中。示例代码如程序清单

12.27 所示。

程序清单 12.27 提供给标准层函数定义

```

/*****
SPI 驱动函数声明
*****/

am_local int __spi_info_get (void *p_arg, am_spi_info_t  *p_info);
am_local int __spi_setup    (void *p_arg, am_spi_device_t *p_dev );
am_local int __spi_msg_start (void                *p_drv,
                                am_spi_device_t    *p_dev,
                                am_spi_message_t    *p_msg);

/**
 * \brief SPI 驱动函数
 */
am_local am_const struct am_spi_drv_funcs __g_spi_drv_funcs = {
    __spi_info_get,
    __spi_setup,
    __spi_msg_start,
};

/** \brief SPI 信息获取 */
am_local int __spi_info_get (void *p_arg, am_spi_info_t  *p_info)
{

    return AM_OK;
}

/** \brief 设置 SPI 从机设备 */
am_local int __spi_setup    (void *p_arg, am_spi_device_t *p_dev )
{

    return AM_OK;
}

/** \brief 启动 SPI 消息传输，完成后调用回调函数 */
am_local int __spi_msg_start (void                *p_drv,
                                am_spi_device_t    *p_dev,
                                am_spi_message_t    *p_msg)
{

    return AM_OK;
}

```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。可以编译一下，应该还是没有错误的。如果有错误，表明结构没有搭建好。可能存在警告，毕竟有些变量没有使用，

可以暂时不用理会。

12.3.2 驱动函数实现

通过上面的函数框架搭建，共计 5 个函数，3 个功能性函数面向标准层，2 个初始解初始化函数面向用户。另外我们还可能会将要调用的一些功能代码封装为相应的函数以保证程序的易维护性。

除上述函数以外，SPI 需要产生中断，因此我们需要定义中断服务函数 `__spi_irq_handler`，同时本次 SPI 需要使用状态机的方法来进行数据传输因此我们需要定义一个状态机函数 `__spi_mst_sm_event`。另外 SPI 的一些基础函数 `__spi_write_data`、`__spi_read_data`、`__spi_config` 也都应该被定义好，它们都应该被状态机去调用。我们还需要使用到一些链表操作，因此需要定义函数 `__spi_msg_in`、`__spi_msg_out`、`__spi_trans_out` 用于处理相关链表。下面逐步实现。

1. `am_zmf2x6_spi_int_init` 函数

该函数主要完成三个功能：

- 1) 初始化设备中的标准服务，将成员赋值，确定 `p_drv` 的值，`p_drv` 的值确定了提供给标准接口层的那 3 个函数的第一个参数是什么。那 3 个函数才能根据这儿的设置去使用。（`void *` 类型无法使用，必须转换为具体的一种类型才能使用），通常情况下，`p_drv` 都是设置为设备结构体的地址，这样方便访问整个设备的相关内容。
- 2) 根据 `p_devinfo` 内配置的信息进行一些硬件的基础配置，并进行一些必要的初始化配置，此处封装为一个函数 `__spi_hard_init`。
- 3) 连接中断服务函数并使能全局中断。

初始化硬件相关设置，该函数的示例代码如程序清单 12.28 所示。

程序清单 12.28 SPI 初始化函数

```
/**
 * \brief SPI 硬件初始化
 */
am_local
int __spi_hard_init (am_zmf2x6_spi_poll_dev_t *p_this)
{
    amhw_zmf2x6_spi_t *p_hw_spi = (amhw_zmf2x6_spi_t *) (p_this->p_devinfo->spi_reg_base);

    if (p_this == NULL) {
        return -AM_EINVAL;
    }

    /* 配置为主机模式 */
    amhw_zmf2x6_spi_mode_sel(p_hw_spi, AMHW_ZMF2X6_SPI_MODE_MASTER);
    amhw_zmf2x6_spi_tx_enable(p_hw_spi, AM_TRUE);
    amhw_zmf2x6_spi_rx_enable(p_hw_spi, AM_TRUE);
    amhw_zmf2x6_spi_module_enable(p_hw_spi, AM_TRUE);

    /* 初始化配置 SPI */
```

```

    return AM_OK;
}

/**
 * \brief SPI 初始化
 */
am_spi_handle_t am_zmf2x6_spi_int_init (am_zmf2x6_spi_int_dev_t      *p_dev,
                                       const am_zmf2x6_spi_int_devinfo_t *p_devinfo)
{
    if (NULL == p_devinfo || NULL == p_dev ) {
        return NULL;
    }

    if (p_devinfo->pfn_plfm_init) {
        p_devinfo->pfn_plfm_init();
    }

    p_dev->spi_serve.p_funcs = (struct am_spi_drv_funcs *)&__g_spi_drv_funcs;
    p_dev->spi_serve.p_drv   = p_dev;

    p_dev->p_devinfo = p_devinfo;

    p_dev->p_cur_spi_dev   = NULL;
    p_dev->p_tgl_dev       = NULL;
    p_dev->busy            = AM_FALSE;
    p_dev->p_cur_msg       = NULL;
    p_dev->p_cur_trans     = NULL;
    p_dev->data_ptr        = 0;
    p_dev->nbytes_to_recv  = 0;
    p_dev->state           = __SPI_ST_IDLE;    /* 初始化为空闲状态 */

    am_list_head_init(&(p_dev->msg_list));

    if (__spi_hard_init(p_dev) != AM_OK) {
        return NULL;
    }

    am_int_connect(p_dev->p_devinfo->inum,
                  __spi_irq_handler,
                  (void *)p_dev);

    am_int_enable(p_dev->p_devinfo->inum);

    return &(p_dev->spi_serve);
}

```

}

2. am_zmf2x6_spi_int_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放，仅将部分结构体指向 NULL 即可。另外是对硬件资源的释放，包括禁能 SPI 及相关中断，执行 p_devinfo 内的平台解初始化函数。

示例代码如程序清单 12.29 所示。

程序清单 12.29 SPI 去初始化函数

```
/**
 * \brief SPI 去除初始化
 */
void am_zmf2x6_spi_int_deinit (am_spi_handle_t handle)
{
    am_zmf2x6_spi_int_dev_t *p_dev      = (am_zmf2x6_spi_int_dev_t *)handle;
    amhw_zmf2x6_spi_t          *p_hw_spi = (amhw_zmf2x6_spi_t *) (p_dev->p_devinfo->spi_reg_base);

    if (NULL == p_dev) {
        return ;
    }

    p_dev->spi_serve.p_funcs = NULL;
    p_dev->spi_serve.p_drv   = NULL;

    /* 禁能 SPI */
    amhw_zmf2x6_spi_module_enable(p_hw_spi, AM_FALSE);

    /* 关闭 SPI 中断号并断开连接 */
    am_int_disable(p_dev->p_devinfo->inum);
    am_int_disconnect(p_dev->p_devinfo->inum,
                     __spi_irq_handler,
                     (void *)p_dev);

    if (p_dev->p_devinfo->pfn_plfm_deinit) {
        p_dev->p_devinfo->pfn_plfm_deinit();
    }
}
```

3. __spi_info_get 函数

该函数为标准接口 am_spi_info_get 的驱动原型，主要用于获取 SPI 的相关信息，信息以参数类型 am_spi_info_t（关于其介绍见 am_spi_info 结构体）进行传递。因此写驱动程序时我们应根据 SPI 的最小工作频率、最大工作频率、SPI 控制器特性填充该结构体变量，如程序清单 12.30 所示。

程序清单 12.30 SPI 获取信息函数

```

am_local
int __spi_info_get (void *p_arg, am_spi_info_t *p_info)
{
    am_zmf2x6_spi_int_dev_t *p_this = (am_zmf2x6_spi_int_dev_t *)p_arg;

    if (p_info == NULL) {
        return -AM_EINVAL;
    }

    /* 最大速率等于 PCLK */
    p_info->max_speed = __SPI_MAXSPEED_GET(p_hw_spi);
    p_info->min_speed = __SPI_MINSPEED_GET(p_hw_spi);
    p_info->features = AM_SPI_CS_HIGH |
                     AM_SPI_LSB_FIRST |
                     AM_SPI_MODE_0 |
                     AM_SPI_MODE_1 |
                     AM_SPI_MODE_2 |
                     AM_SPI_MODE_3; /* features */

    return AM_OK;
}

```

4. __spi_setup 函数

该函数为标准接口 `am_spi_setup` 的驱动原型，用于设置连接 SPI 从设备的片选引脚相关控制信息，并对从设备信息进行相关判断，确定该 SPI 是否能够与 SPI 从设备通信。相关实现如程序清单 12.31 所示。

程序清单 12.31 SPI 从设备转载函数

```

am_local
int __spi_setup (void *p_arg, am_spi_device_t *p_dev)
{
    am_zmf2x6_spi_int_dev_t *p_this = (am_zmf2x6_spi_int_dev_t *)p_arg;

    uint32_t max_speed, min_speed;

    if (p_dev == NULL) {
        return -AM_EINVAL;
    }

    /* 默认数据为 8 位，最大不超过 16 位 */
    if (p_dev->bits_per_word == 0) {
        p_dev->bits_per_word = 8;
    } else if (p_dev->bits_per_word > 32) {
        return -AM_ENOTSUP;
    }
}

```

```

/* 最大 SPI 速率不能超过主时钟，最小不能小于主时钟 65536 分频 */
max_speed = __SPI_MAXSPEED_GET(p_hw_spi);
min_speed = __SPI_MINSPEED_GET(p_hw_spi);

if (p_dev->max_speed_hz > max_speed) {
    p_dev->max_speed_hz = max_speed;
} else if (p_dev->max_speed_hz < min_speed) {
    return -AM_ENOTSUP;
}

/* 无片选函数 */
if (p_dev->mode & AM_SPI_NO_CS) {
    p_dev->pfunc_cs = __spi_default_cs_dummy;

/* 有片选函数 */
} else {

    /* 不提供则默认片选函数 */
    if (p_dev->pfunc_cs == NULL) {

        /* 片选高电平有效 */
        if (p_dev->mode & AM_SPI_CS_HIGH) {
            am_gpio_pin_cfg(p_dev->cs_pin, AM_GPIO_OUTPUT_INIT_LOW);
            p_dev->pfunc_cs = __spi_default_cs_ha;

            /* 片选低电平有效 */
        } else {
            am_gpio_pin_cfg(p_dev->cs_pin, AM_GPIO_OUTPUT_INIT_HIGH);
            p_dev->pfunc_cs = __spi_default_cs_la;
        }
    }
}

/* 解除片选信号 */
__spi_cs_off(p_this, p_dev);

return AM_OK;
}

```

5. __spi_msg_start 函数

该函数为标准接口 `am_spi_msg_start` 的驱动原型，用于启动一次 SPI 消息传输。由于使用到消息队列以及状态机处理，因此程序内部仅仅是将需要传递的消息加入到链表中（使用函数 `__spi_msg_in`）。通过对消息处理状态的区分，若正在处理消息则只需要把消息连接上链表即可，若此时无消息再处理，则还会掉用状态机函数处理消息。

相关实现如程序清单 12.32 所示。

程序清单 12.32 SPI 消息启动函数

```

/**
 * \brief SPI 传输数据函数
 */
am_local
int __spi_msg_start(void          *p_drv,
                    am_spi_device_t *p_dev,
                    am_spi_message_t *p_msg)
{
    am_zmf2x6_spi_int_dev_t *p_this = (am_zmf2x6_spi_int_dev_t *)p_drv;

    int key;

    /* 设备有效性检查 */
    if ((p_drv == NULL) ||
        (p_dev == NULL) ||
        (p_msg == NULL) ){
        return -AM_EINVAL;
    }

    p_msg->p_spi_dev = p_dev; /* 设备参数信息存入到消息中 */
    p_this->p_cur_msg = p_msg; /* 将当前设备传输消息存入 */
    p_this->nbytes_to_recv = 0; /* 待接收字符数清 0 */
    p_this->data_ptr = 0; /* 已接收字符数清 0 */

    key = am_int_cpu_lock();

    /* 当前正在处理消息，只需要将新的消息加入链表即可 */
    if (p_this->busy == AM_TRUE) {
        __spi_msg_in(p_this, p_msg);
        am_int_cpu_unlock(key);
        return AM_OK;
    } else {
        p_this->busy = AM_TRUE;
        __spi_msg_in(p_this, p_msg);
        p_msg->status = -AM_EISCONN; /* 正在排队中 */
        am_int_cpu_unlock(key);

        /* 启动状态机 */
        return __spi_mst_sm_event(p_this, __SPI_EVT_TRANS_LAUNCH);
    }
}

```

6. __spi_write_data

该函数用于发送一个数据。对于 SPI 来说，发送一个数据也对应着接收一个数据，因此在该函数末尾使用到 `p_dev->nbytes_to_recv` 用于通知接收函数需要接收一个位宽的数据。由于该 SPI 模块支持 1~32 位宽的传输，因此我们需要给他分类，小于等于 8 则使用函数 `amhw_zmf2x6_spi_tx_data8_put`，否则调用 `amhw_zmf2x6_spi_tx_data32_put`。

值得注意的是函数内使用到了 `p_dev->data_ptr` 变量作为缓冲区数据的偏移，查看数据接收函数后我们可以发现它的值在接收完之后才被增加，同时该变量在状态机函数内也对其进行了判断，当该值大于等于此次要传输的数据长度（`p_cur_trans->nbytes`）时完成此次传输。

相关代码实现如程序清单 12.33 所示。

程序清单 12.33 SPI 写数据函数

```
am_local
void __spi_write_data (am_zmf2x6_spi_int_dev_t *p_dev)
{
    amhw_zmf2x6_spi_t *p_hw_spi = (amhw_zmf2x6_spi_t *) (p_dev->p_devinfo->spi_reg_base);
    am_spi_transfer_t *p_trans = p_dev->p_cur_trans;

    if (p_dev == NULL) {
        return ;
    }

    while(!(amhw_zmf2x6_spi_reg_cstat_get (p_hw_spi) & AMHW_ZMF2X6_SPI_CSTAT_TX_EMPTY));

    /* tx_buf 有效 */
    if (p_trans->p_txbuf != NULL) {
        if ((p_dev->p_cur_spi_dev->bits_per_word) <= 8) {

            /** \brief 待发送数据的基址+偏移 */
            uint8_t *ptr = (uint8_t *) (p_trans->p_txbuf) + p_dev->data_ptr;
            amhw_zmf2x6_spi_tx_data8_put(p_hw_spi, *ptr);

        } else {
            uint32_t *ptr = (uint32_t *) (p_trans->p_txbuf) + p_dev->data_ptr;
            amhw_zmf2x6_spi_tx_data32_put(p_hw_spi, *ptr);
        }
    }

    /* tx_buf 无效 */
    } else {

        /* 待发送数据无效 直接发 0 */
        if ((p_dev->p_cur_spi_dev->bits_per_word) <= 8) {
            amhw_zmf2x6_spi_tx_data8_put(p_hw_spi, 0x00);
        } else {
            amhw_zmf2x6_spi_tx_data32_put(p_hw_spi, 0x0000);
        }
    }
}
```

```

    }
}

/** \brief 需接收数据的 byte 数 */
p_dev->nbytes_to_rcv = __bits_to_bytes(p_dev->p_cur_spi_dev->bits_per_word); //todo
p_dev->p_cur_msg->actual_length += p_dev->nbytes_to_rcv;
}

```

7. __spi_read_data

该函数用于获取一个数据，每次调用它之前应当先执行__spi_write_data 函数。因此该函数在最底部将待接收变量（p_dev->nbytes_to_rcv）清除，等待下一次数据发送后被调用。

定义相关指针以指向接收缓冲区现在要接收数据的位置，即偏移量 p_dev->data_ptr，获取到对应地址之后方可调用 HW 层函数获取数据。在最后它还将 p_dev->data_ptr 变量加一，具体原因在__spi_write_data 已经介绍。

相关代码实现如程序清单 12.34 所示。

程序清单 12.34 SPI 读数据函数

```

am_local
void __spi_read_data (am_zmf2x6_spi_int_dev_t *p_dev)
{
    amhw_zmf2x6_spi_t *p_hw_spi = (amhw_zmf2x6_spi_t *) (p_dev->p_devinfo->spi_reg_base);

    am_spi_transfer_t *p_trans = p_dev->p_cur_trans;
    uint32_t *p_buf32;
    uint8_t *p_buf8;

    if ((p_dev->p_cur_spi_dev->bits_per_word) <= 8) {
        p_buf8 = (uint8_t *) p_trans->p_rxbuf + p_dev->data_ptr;
    } else {
        p_buf32 = (uint32_t *) p_trans->p_rxbuf + p_dev->data_ptr;
    }

    if (p_dev == NULL) {
        return ;
    }

    if (amhw_zmf2x6_spi_reg_cstat_get (p_hw_spi) & AMHW_ZMF2X6_SPI_CSTAT_RXVAL) {

        /* rx_buf 有效 */
        if (p_trans->p_rxbuf != NULL && p_dev->nbytes_to_rcv) {
            if ((p_dev->p_cur_spi_dev->bits_per_word) <= 8) {
                *p_buf8 = amhw_zmf2x6_spi_rx_data8_get(p_hw_spi);
            } else {
                *p_buf32 = amhw_zmf2x6_spi_rx_data32_get(p_hw_spi);
            }
        }
    }
}

```

```

/* rx_buf 无效或者不需要接收数据 */
} else {
    if ((p_dev->p_cur_spi_dev->bits_per_word) <= 8) {
        (void)amhw_zmf2x6_spi_rx_data8_get(p_hw_spi);

    } else {
        (void)amhw_zmf2x6_spi_rx_data32_get(p_hw_spi);
    }
}

/* 已经发送或接收的数据 byte 数 */
p_dev->data_ptr += p_dev->nbytes_to_rcv;
p_dev->nbytes_to_rcv = 0;

}
}

```

8. __spi_config

该函数涉及到 SPI 模块开始一个消息传输前的一些基础参数的配置，除通用配置外，另外主要需要区分设置的是设备模式、位方向、字大小、通信速率。该函数仅通过状态机调用。

相关代码实现如程序清单 12.35 所示。

程序清单 12.35 SPI 配置函数

```

am_local
int __spi_config(am_zmf2x6_spi_int_dev_t *p_this)
{
    const am_zmf2x6_spi_int_devinfo_t *p_devinfo = p_this->p_devinfo;
    amhw_zmf2x6_spi_t *p_hw_spi = (amhw_zmf2x6_spi_t *) (p_devinfo->spi_reg_base);
    am_spi_transfer_t *p_trans = p_this->p_cur_trans;

    uint32_t mode_flag = 0;

    /* 如果为 0，使用默认参数值 */
    if (p_trans->bits_per_word == 0) {
        p_trans->bits_per_word = p_this->p_cur_spi_dev->bits_per_word;
    }

    if (p_trans->speed_hz == 0) {
        p_trans->speed_hz = p_this->p_cur_spi_dev->max_speed_hz;
    }

    /* 设置字节数有效性检查 */
    if (p_trans->bits_per_word > 32 || p_trans->bits_per_word < 1) {
        return -AM_EINVAL;
    }
}

```

```

/* 设置分频值有效性检查 */
if (p_trans->speed_hz > __SPI_MAXSPEED_GET(p_hw_spi) ||
    p_trans->speed_hz < __SPI_MINSPEED_GET(p_hw_spi)) {
    return -AM_EINVAL;
}

/* 发送和接收缓冲区有效性检查 */
if ((p_trans->p_txbuf == NULL) && (p_trans->p_rxbuf == NULL)) {
    return -AM_EINVAL;
}

/* 发送字节数检查 */
if (p_trans->nbytes == 0) {
    return -AM_ELOW;
}

/**
 * 配置当前设备模式
 */
mode_flag = 0;

if (p_this->p_cur_spi_dev->mode & AM_SPI_LSB_FIRST) {
    mode_flag |= AMHW_ZMF2X6_SPI_DATA_LSB;
}

/* 配置 SPI 模式（时钟相位和极性） */
amhw_zmf2x6_spi_clk_mode_set(p_hw_spi, (0x3 & p_this->p_cur_spi_dev->mode) ^ 0x01);

/* 配置 SPI 工作特性 */
amhw_zmf2x6_spi_first_bit_sel(p_hw_spi, (1u << 2) & mode_flag);

amhw_zmf2x6_spi_data_len_sel(p_hw_spi, AMHW_ZMF2X6_SPI_DATA_LEN_8BIT);

/* 配置为主机模式 */
amhw_zmf2x6_spi_mode_sel(p_hw_spi, AMHW_ZMF2X6_SPI_MODE_MASTER);

if (p_this->p_cur_spi_dev->bits_per_word > 8) {
    amhw_zmf2x6_spi_valid_data_sel(p_hw_spi, AMHW_ZMF2X6_SPI_VALID_DATA_32BIT);
    amhw_zmf2x6_spi_first_bit_sel(p_hw_spi, AMHW_ZMF2X6_SPI_DATA_LSB);
    if (p_this->p_cur_spi_dev->bits_per_word == 32) {
        amhw_zmf2x6_spi_extlen_set(p_hw_spi, 0);
    } else {
        amhw_zmf2x6_spi_extlen_set(p_hw_spi, p_this->p_cur_spi_dev->bits_per_word);
    }
}

```

```

    }
}

if (p_trans->speed_hz > (72000000 / 5)) {

    /* 高速 */
    amhw_zmf2x6_spi_tx_data_edge_sel(p_hw_spi, AMHW_ZMF2X6_SPI_TX_DATA_IMMEDIATELY);
    amhw_zmf2x6_spi_rx_data_edge_sel(p_hw_spi, AMHW_ZMF2X6_SPI_RX_DATA_LAST_CLOCK);

} else {

    /* 低速 */
    amhw_zmf2x6_spi_tx_data_edge_sel(p_hw_spi, AMHW_ZMF2X6_SPI_TX_DATA_ONE_CLOCK);
    amhw_zmf2x6_spi_rx_data_edge_sel(p_hw_spi, AMHW_ZMF2X6_SPI_RX_DATA_MID_CLOCK);

}

/* 设置 SPI 速率 */
__spi_speed_cfg(p_this, p_trans->speed_hz);

return AM_OK;
}

```

9. __spi_mst_sm_event

为 SPI 的主要工作逻辑函数，使用状态机的方法使操作变得更加简单。其中关于状态相关描述如表 12.3 所示，状态的执行需要保证产生了相关的事件，关于事件的定义如表 12.4 所示。

表 12.3 状态机状态详情

状态	状态对应宏	含义
空闲状态	__SPI_ST_IDLE	初始化后默认为空闲状态，仅当检测到传输就绪事件后进入消息开始状态。
消息开始	__SPI_ST_MSG_START	仅处于传输就绪事件下可进入执行。检测到若无消息处理则进入空闲态，并事件置为无事件。有消息处理进入传输开始状态，保持参数就绪事件。
传输开始	__SPI_ST_TRANS_START	仅处于传输就绪事件下可进入执行。分为传输完成和正在传输中。如果为传输完成则进入消息开始状态，保持参数就绪事件。若正在传输则会进入主机发送状态并置发送数据事件。
主机发送	__SPI_ST_M_SEND_DATA	仅处于发送数据事件下可进入执行（发送中断能够产生发送事件）。分为无数据要传输和还有数据。无数据传输时进入传输开始状态，并置参数就绪事件。还有数据需要发送则进入接收数据状态，置无事件。
主机接收	__SPI_ST_M_RECV_DATA	仅处于接收数据事件下可进入执行（接收中断能够产生接收事件）。数据接收完成后会进入主机发送状态并置发送数据事件。

表 12.4 状态机事件详情

事件	状态对应宏	含义
无事件	__SPI_EVT_NONE	空闲状态
传输就绪	__SPI_EVT_TRANS_LAUNCH	用于表示传输数据已经就绪
发送数据	__SPI_EVT_M_SEND_DATA	用于表示要发送数据
接收数据	__SPI_EVT_M_RECV_DATA	用于表示要接收数据

通过对上述的了解后剩下的就是各状态需要完成的事情了。

- 空闲状态

无实际执行代码，作为默认状态。仅当产生传输就绪事件进入消息开始状态。

- 消息开始状态

该状态会从消息链表内获取消息，判断消息（对应结构体 `am_spi_message_t`）是否为空，若为空禁能所有中断，回到空闲状态。若不为空从消息中提取 SPI 从设备信息，进入传输开始状态。

- 传输开始状态

检测信息中传输链表是否为空：

若不为空提取出传输链表中的一个节点，执行 `__spi_config` 函数配置 SPI 传输参数，选通片选，执行相关信息传输（即数据的发送和接收），进入主机发送状态。

若为空（消息传输完成），执行消息完成中断函数，关闭片选，进入消息开始状态。

- 主机发送状态

该状态主要分为两部分，即正在传输中以及传输完成。

在传输过程中，我们会判断发送缓冲器和发送移位寄存器是否为空，若为空则直接发送数据，使能接收中断并设置主机接收状态置无事件（等待接收中断调用状态机函数）状态机结束。若不为空则使能发送中断，并结束状态机，等待发送中断到来（通过发送中断调用状态机函数）再次进入将执行为空时的操作。

在传输完成时，需要禁能所有中断，并回到传输开始状态，等待其进行下一个传输。

- 主机接收状态

该状态仅能够在接收中断调用状态机后执行，默认直接调用函数 `__spi_read_data` 读取相关数据，并禁能发送中断和接收中断，进入主机发送状态传输剩余的数据。

就此状态机函数编写完成，代码详情见程序清单 12.36。

程序清单 12.36 SPI 状态机传输函数

```
/**
 * \brief SPI 使用状态机传输
 */
am_local
int __spi_mst_sm_event(am_zmf2x6_spi_int_dev_t *p_dev, uint32_t event)
{
    amhw_zmf2x6_spi_t *p_hw_spi = (amhw_zmf2x6_spi_t *) (p_dev->p_devinfo->spi_reg_base);

    volatile uint32_t new_event = __SPI_EVT_NONE;
```

```

if (p_dev == NULL) {
    return -AM_EINVAL;
}

while (1) {

    if (new_event != __SPI_EVT_NONE) {        /* 检查新事件是否来自内部 */
        event      = new_event;
        new_event  = __SPI_EVT_NONE;
    }

    switch (p_dev->state) {

    case __SPI_ST_IDLE:        /* 控制器处于空闲状态 */
    {
        if (event != __SPI_EVT_TRANS_LAUNCH) {
            return -AM_EINVAL;    /* 空闲状态等待的消息必须是启动传输 */
        }

        /* 切换到消息开始状态，不用 break */
    }
    /* no break */

    case __SPI_ST_MSG_START:    /* 消息开始 */
    {
        am_spi_message_t *p_cur_msg  = NULL;

        int key;

        if (event != __SPI_EVT_TRANS_LAUNCH) {
            return -AM_EINVAL;    /* 消息开始状态等待的消息必须是启动传输 */
        }

        key = am_int_cpu_lock();
        p_cur_msg      = __spi_msg_out(p_dev);
        p_dev->p_cur_msg = p_cur_msg;

        if (p_cur_msg) {
            p_cur_msg->status = -AM_EINPROGRESS;
        } else {

            /* 禁能所有中断 */
            amhw_zmf2x6_spi_int_disable(p_hw_spi, AMHW_ZMF2X6_SPI_INTSTAT_INT_ALL);

```



```

        p_dev->busy = AM_FALSE;
    }
    am_int_cpu_unlock(key);

    /* 无需要处理的消息 */
    if (p_cur_msg == NULL) {
        __SPI_NEXT_STATE(__SPI_ST_IDLE, __SPI_EVT_NONE);
        break;
    } else {
        p_dev->p_cur_spi_dev = p_cur_msg->p_spi_dev;

        /* 直接进入下一个状态，开始一个传输，此处无需 break */
        __SPI_NEXT_STATE(__SPI_ST_TRANS_START, __SPI_EVT_TRANS_LAUNCH);

        event      = new_event;
        new_event = __SPI_EVT_NONE;
    }
}

/* no break */

case __SPI_ST_TRANS_START: /* 传输开始 */
{
    am_spi_message_t *p_cur_msg = p_dev->p_cur_msg;

    if (event != __SPI_EVT_TRANS_LAUNCH) {
        return -AM_EINVAL; /* 传输开始状态等待的消息必须是启动传输 */
    }

    /* 当前消息传输完成 */
    if (am_list_empty(&(p_cur_msg->transfers))) {

        p_cur_msg->actual_length = 0;

        /* 消息正在处理中 */
        if (p_cur_msg->status == -AM_EINPROGRESS) {
            p_cur_msg->status = AM_OK;
        }

        /* notify the caller */
        if (p_cur_msg->pfn_complete != NULL) {
            p_cur_msg->pfn_complete(p_cur_msg->p_arg);
        }

        /* 片选关闭 */

```

```

        __spi_cs_off(p_dev, p_dev->p_cur_spi_dev);

        __SPI_NEXT_STATE(__SPI_ST_MSG_START, __SPI_EVT_TRANS_LAUNCH);

    } else {

        /* 获取到一个传输，正确处理该传输即可 */
        am_spi_transfer_t *p_cur_trans = __spi_trans_out(p_cur_msg);
        p_dev->p_cur_trans = p_cur_trans;

        /* reset current data pointer */
        p_dev->data_ptr = 0;
        p_dev->nbytes_to_rcv = 0;

        /* 配置 SPI 传输参数 */
        __spi_config(p_dev);

        /* 若为三线模式且接收缓存 rx_buf 不为空的时候 */
        if ((p_dev->p_cur_spi_dev->mode & AM_SPI_3WIRE) &&
            (p_dev->p_cur_trans->rxbuf)) {

            /* 把 MOSI 置为输入浮空 */
            am_gpio_pin_cfg(p_dev->p_devinfo->mosi_pin, AM_GPIO_INPUT |
AM_GPIO_FLOAT);
        }

        /* CS 选通 */
        __spi_cs_on(p_dev, p_dev->p_cur_spi_dev);

        /* 下一状态是使用中断方式发送数据 */
        __SPI_NEXT_STATE(__SPI_ST_M_SEND_DATA, __SPI_EVT_M_SEND_DATA);
    }
    break;
}

case __SPI_ST_M_SEND_DATA: /* 发送数据 */
{
    am_spi_transfer_t *p_cur_trans = p_dev->p_cur_trans;

    if (event != __SPI_EVT_M_SEND_DATA) {
        return -AM_EINVAL; /* 主机发送状态等待的消息必须是发送数据 */
    }

    /* 没有更多需要传送的数据了 */

```

```

if (p_dev->data_ptr >= p_cur_trans->nbytes) {

    /* 关闭中断等待发送就绪 */
    amhw_zmf2x6_spi_int_disable(p_hw_spi, AMHW_ZMF2X6_SPI_INTSTAT_INT_ALL);

    /* 回到传输开始状态 */
    __SPI_NEXT_STATE(__SPI_ST_TRANS_START, __SPI_EVT_TRANS_LAUNCH);

    break;
} else {
    if ((amhw_zmf2x6_spi_reg_cstat_get(p_hw_spi) &
AMHW_ZMF2X6_SPI_CSTAT_TX_EMPTY) != 0) {

        /* 向从机写数据 */
        __spi_write_data(p_dev);

        __SPI_NEXT_STATE(__SPI_ST_M_RECV_DATA, __SPI_EVT_NONE);

        /* Enable RX interrupt */
        amhw_zmf2x6_spi_int_enable(p_hw_spi,
AMHW_ZMF2X6_SPI_INTSTAT_RX_FALG);
    } else {

        /* 打开中断等待发送就绪 */
        amhw_zmf2x6_spi_int_enable(p_hw_spi,
AMHW_ZMF2X6_SPI_INTSTAT_TX_FALG);
    }
}
break;
}

case __SPI_ST_M_RECV_DATA: /* 接收数据 */
{
    if (event != __SPI_EVT_M_RECV_DATA) {
        return -AM_EINVAL; /* 主机接收状态等待的消息必须是接收数据 */
    }

    /* 读取数据 */
    __spi_read_data(p_dev);

    /* 没有更多的数据需要接收了 */
    if (p_dev->nbytes_to_rcv == 0) {

        /* 若为三线模式且接收缓存 rx_buf 不为空的时候 */

```

```

        if ((p_dev->p_cur_spi_dev->mode & AM_SPI_3WIRE) &&
            (p_dev->p_cur_trans->p_rxbuf)) {

            /* 重新置回 MOSI 模式 */
            am_gpio_pin_cfg(p_dev->p_devinfo->mosi_pin, p_dev->p_devinfo->flags);
        }

        /* 传输剩余数据 */
        __SPI_NEXT_STATE(__SPI_ST_M_SEND_DATA, __SPI_EVT_M_SEND_DATA);

        /* 禁能发送及接收中断 */
        amhw_zmf2x6_spi_int_disable(p_hw_spi, AMHW_ZMF2X6_SPI_INTSTAT_RX_FALG |
AMHW_ZMF2X6_SPI_INTSTAT_TX_FALG);

        /* 需要接收更多数据 */
        } else {

            /* 这儿什么也不要做 */
        }
        break;
    }

    /*
     * 永远也不该运行到这儿
     */
    default:
        break;
    }

    /* 没有来自内部的信息, 跳出 */
    if (new_event == __SPI_EVT_NONE) {
        break;
    }
}
return AM_OK;
}

```

10. __spi_irq_handler

该函数为 SPI 的中断处理函数，只需要在发送中断中调用传递发送事件参数的状态机函数，在接收中断中调用传递接收事件参数的状态机函数。

相关代码实现如程序清单 12.37 所示。

程序清单 12.37 SPI 中断处理函数

```
/**
```

```

* \brief SPI 中断句柄定义
*/
am_local
void __spi_irq_handler (void *p_arg)
{
    am_zmf2x6_spi_int_dev_t *p_dev      = (am_zmf2x6_spi_int_dev_t *)p_arg;
    amhw_zmf2x6_spi_t      *p_hw_spi = (amhw_zmf2x6_spi_t *) (p_dev->p_devinfo->spi_reg_base);

    uint32_t spi_status = (uint32_t)amhw_zmf2x6_spi_reg_intstat_get(p_hw_spi);

    /* 发送数据 */
    if((spi_status & AMHW_ZMF2X6_SPI_INTSTAT_TX_FALG)) {
        amhw_zmf2x6_spi_int_flag_clr(p_hw_spi, AMHW_ZMF2X6_SPI_INTSTAT_TX_FALG);
        __spi_mst_sm_event(p_dev, __SPI_EVT_M_SEND_DATA);
    }

    /* 接收数据 */
    if((spi_status & AMHW_ZMF2X6_SPI_INTSTAT_RX_FALG) != 0) {
        amhw_zmf2x6_spi_int_flag_clr(p_hw_spi, AMHW_ZMF2X6_SPI_INTSTAT_RX_FALG);
        __spi_mst_sm_event(p_dev, __SPI_EVT_M_RECV_DATA);
    }
}

```

12.4 用户配置文件编写

12.4.1 定义设备常量于设备信息常量

创建配置文件后，我们需要在里面加入设备变量，设备变量只需要定义，不用初始化。设备信息常量定义时初始化，默认填充的信息是一种常用的配置方式，SPI 设备信息配置如程序清单 12.38 所示。

程序清单 12.38 SPI 设备信息配置文件

```

/** \brief SPI1 平台初始化 */
static void __zmf2x6_plfm_spi1_int_init (void)
{
    am_gpio_pin_cfg(PIOA_5, PIOA_5_SPI1_SCK | PIOA_5_AF_PP);
    am_gpio_pin_cfg(PIOA_6, PIOA_6_SPI1_MISO | PIOA_6_INPUT_FLOAT);
    am_gpio_pin_cfg(PIOA_7, PIOA_7_SPI1_MOSI | PIOA_7_AF_PP);

    am_clk_enable(CLK_SPI1);
}

/** \brief 解除 SPI1 平台初始化 */
static void __zmf2x6_plfm_spi1_int_deinit (void)
{

```

```

/* 释放引脚为输入模式 */
am_gpio_pin_cfg(PIOA_5, AM_GPIO_INPUT);
am_gpio_pin_cfg(PIOA_6, AM_GPIO_INPUT);
am_gpio_pin_cfg(PIOA_7, AM_GPIO_INPUT);

am_clk_disable(CLK_SPI1);
}

/**
 * \brief SPI1 设备信息
 */
const struct am_zmf2x6_spi_int_devinfo __g_spi1_int_devinfo = {
    ZMF2X6_SPI1_BASE,          /**< \brief SPI1 寄存器指针 */
    CLK_SPI1,                  /**< \brief 时钟 ID 号 */
    INUM_SPI1,                 /**< \brief SPI1 中断号 */
    PIOA_7_SPI1_MOSI | PIOA_7_AF_PP, /**< \brief SPI1 配置标识 */
    PIOA_7,                    /**< \brief MOSI 引脚号 */
    __zmf2x6_plfm_spi1_int_init, /**< \brief SPI1 平台初始化函数 */
    __zmf2x6_plfm_spi1_int_deinit /**< \brief SPI1 平台解初始化函数 */
};

/** \brief SPI1 设备实例 */
static am_zmf2x6_spi_int_dev_t __g_spi1_int_dev;

/** \brief SPI1 实例初始化，获得 SPI 标准服务句柄 */
am_spi_handle_t am_zmf2x6_spi1_int_inst_init(void)
{
    return am_zmf2x6_spi_int_init(&__g_spi1_int_dev, &__g_spi1_int_devinfo);
}

/** \brief SPI1 实例解初始化 */
void am_zmf2x6_spi1_int_inst_deinit(am_spi_handle_t handle)
{
    am_zmf2x6_spi_int_deinit(handle);
}

```

12.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

平台初始化函数和解初始化函数定义如程序清单 12.39 所示。

程序清单 12.39 平台初始化函数和解初始化函数定义

```

/** \brief SPI1 平台初始化 */
static void __zmf2x6_plfm_spi1_int_init(void)

```

```

{
    am_gpio_pin_cfg(PIOA_5, PIOA_5_SPI1_SCK | PIOA_5_AF_PP);
    am_gpio_pin_cfg(PIOA_6, PIOA_6_SPI1_MISO | PIOA_6_INPUT_FLOAT);
    am_gpio_pin_cfg(PIOA_7, PIOA_7_SPI1_MOSI | PIOA_7_AF_PP);

    am_clk_enable(CLK_SPI1);
}

/** \brief 解除 SPI1 平台初始化 */
static void __zmf2x6_plfm_spi1_int_deinit(void)
{
    /* 释放引脚为输入模式 */
    am_gpio_pin_cfg(PIOA_5, AM_GPIO_INPUT);
    am_gpio_pin_cfg(PIOA_6, AM_GPIO_INPUT);
    am_gpio_pin_cfg(PIOA_7, AM_GPIO_INPUT);

    am_clk_disable(CLK_SPI1);
}

```

12.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在标准层实例公共头文件 `am_zmf2x6_inst_init.h` 中加入它们的声明，如所示程序清单 12.40。

程序清单 12.40 平台初始化解初始化声明

```

/**
 * \brief SPI1 INT 实例初始化，获得 SPI 标准服务句柄
 *
 * \param 无
 *
 * \return SPI 标准服务句柄，若为 NULL，表明初始化失败
 */
am_spi_handle_t am_zmf2x6_spi1_int_inst_init(void);

/**
 * \brief SPI1 INT 实例解初始化
 *
 * \param[in] handle 通过 am_zmf2x6_spi1_int_inst_init() 函数
 *                  获得的 SPI 标准服务句柄
 *
 * \return 无
 */
void am_zmf2x6_spi1_int_inst_deinit(am_spi_handle_t handle);

```

用户使用时，应该先调用平台初始化函数 `am_zmf2x6_spi_inst_init` 函数，初始化 SPI，并保存其返回的 `handle`，后续直接使用保存的 `handle` 作为标准接口的第一个参数使用标准接口函数即可。

至此，用户配置文件开发完毕。

12.5 例程文件开发

例程文件开发主要分为两个部分，HW 例程和 STD 例程。

12.5.1 例程内容开发

1. HW 层实现

这里使用 HW 层来实现 SPI 的轮询传输。主要涉及到 SPI 的相关基础配置及写入和读取操作。为提高案例的普适性该案例分两个文件编写，相关代码如程序清单 12.41 程序清单 12.42 所示。

程序清单 12.41 SPI 中断采集电压值 HW 层 DEMO

```
/** [src_zmf2x6_hw_spi_master] */
#include "ametal.h"
#include "am_gpio.h"
#include "am_vdebug.h"
#include "hw/amhw_zmf2x6_spi.h"

/**
 * \name SPI 传输结构体配置参数
 */
#define SPI_CFG_LSB          AM_BIT(0)    /**< \brief 低位优先发送 */
#define SPI_CFG_16BIT       AM_BIT(1)    /**< \brief 传输数据宽度为 16 位 */

#define SPI_CFG_MODE_0      0             /**< \brief 模式 0 */
#define SPI_CFG_MODE_1      1             /**< \brief 模式 1 */
#define SPI_CFG_MODE_2      2             /**< \brief 模式 2 */
#define SPI_CFG_MODE_3      3             /**< \brief 模式 3 */

/** @} */

/**
 * \brief SPI 传输结构体
 */
typedef struct spi_transfer {
    const void    *p_txbuf;                /**< \brief 指向发送数据缓存的指针 */
    void          *p_rxbuf;                /**< \brief 指向接收数据缓存的指针 */
    uint32_t      nbytes;                  /**< \brief 数据长度 */
    uint32_t      flags;                   /**< \brief SPI 传输控制标志位 */
} spi_transfer_t;
```



```

spi_transfer_t g_spi_transfer;      /* 定义一个 SPI 传输结构体实例 */

static void spi_speed_cfg (amhw_zmf2x6_spi_t *p_hw_spi,
                           uint32_t          target_speed,
                           uint32_t          clk_rate)
{
    uint32_t best_pdiv;              /* 计算出的速度 */

    best_pdiv = clk_rate / target_speed;

    amhw_zmf2x6_spi_baud_rate_set(p_hw_spi, best_pdiv);
}

static void spi_master_cfg (amhw_zmf2x6_spi_t *p_hw_spi)
{
    amhw_zmf2x6_spi_first_bit_sel(p_hw_spi, AMHW_ZMF2X6_SPI_DATA_MSB);
    amhw_zmf2x6_spi_data_len_sel(p_hw_spi, AMHW_ZMF2X6_SPI_DATA_LEN_8BIT);
    amhw_zmf2x6_spi_valid_data_sel(p_hw_spi, AMHW_ZMF2X6_SPI_VALID_DATA_8BIT);

    /* 配置时钟相位和极性 */
    amhw_zmf2x6_spi_clk_mode_set(p_hw_spi, SPI_CFG_MODE_1 ^ 0x01);
}

static void spi_master_cs_ctr (amhw_zmf2x6_spi_t *p_hw_spi,
                               int              pin,
                               uint8_t          active_level,
                               am_bool_t        state)
{
    /* 片选有效 */
    if (state) {
        amhw_zmf2x6_spi_cs_sel(p_hw_spi, AM_TRUE);

        /* 片选无效 */
    } else {
        amhw_zmf2x6_spi_cs_sel(p_hw_spi, AM_FALSE);
    }
}

/**
 * \brief SPI 回环传输测试
 */
void spi_loop_trans (amhw_zmf2x6_spi_t *p_hw_spi,
                     spi_transfer_t      *p_trans,

```

```

                                int32_t          cs_pin)
{
    uint32_t pos = 0;

    spi_master_cfg(p_hw_spi);

    spi_master_cs_ctr(p_hw_spi, cs_pin, AM_GPIO_LEVEL_LOW, AM_TRUE);

    while(pos < p_trans->nbytes) {

        /* 等待可以发送 */
        while ((amhw_zmf2x6_spi_reg_cstat_get(p_hw_spi) &
                AMHW_ZMF2X6_SPI_CSTAT_TX_EMPTY) == AM_FALSE);

        amhw_zmf2x6_spi_tx_data8_put(p_hw_spi,
                                     *((uint8_t*)(p_trans->p_txbuf) + pos));

        /* 等待可以接收 */
        while ((amhw_zmf2x6_spi_reg_cstat_get(p_hw_spi) &
                AMHW_ZMF2X6_SPI_CSTAT_RXVAL) == AM_FALSE);

        *(uint8_t*)((uint8_t*)p_trans->p_rxbuf + pos) =
            amhw_zmf2x6_spi_rx_data8_get(p_hw_spi);

        pos += 1;
    }

    /* 处理片选 */
    spi_master_cs_ctr(p_hw_spi, cs_pin, AM_GPIO_LEVEL_LOW, AM_FALSE);
}

/**
 * \brief 例程入口
 */
void demo_zmf2x6_hw_spi_master_entry (amhw_zmf2x6_spi_t *p_hw_spi,
                                int32_t          cs_pin,
                                uint32_t          clk_rate)
{
    uint8_t          i;
    uint8_t          spi_send_buf[16];
    uint8_t          spi_rcv_buf[16];
    const uint32_t    length      = 16;
    am_bool_t         error_ocur = AM_FALSE;
    spi_transfer_t *p_trans      = &g_spi_transfer;

```

```

p_trans->p_txbuf = spi_send_buf;
p_trans->p_rxbuf = spi_recv_buf;
p_trans->nbytes = length;

amhw_zmf2x6_spi_mode_sel(p_hw_spi, AMHW_ZMF2X6_SPI_MODE_MASTER);
spi_speed_cfg(p_hw_spi, 100000, clk_rate);
amhw_zmf2x6_spi_tx_enable(p_hw_spi, AM_TRUE);
amhw_zmf2x6_spi_rx_enable(p_hw_spi, AM_TRUE);
amhw_zmf2x6_spi_module_enable(p_hw_spi, AM_TRUE);

/* 构造数据 */
for (i = 0; i < length; i++) {
    spi_send_buf[i] = i;
}

/* 数据传输 */
spi_loop_trans(p_hw_spi, p_trans, cs_pin);

/* 数据检验 */
for (i = 0; i < length; i++) {
    if (spi_recv_buf[i] != spi_send_buf[i]) {
        error_ocur = AM_TRUE;
        break;
    }
}

if (!error_ocur) {
    am_kprintf("trans done!\n");
} else {
    am_kprintf("trans error!\n");
}

AM_FOREVER {
    ; /* VOID */
}
}

/** [src_zmf2x6_hw_spi_master] */

```

程序清单 12.42 SPI 轮询通信案例入口

```

/** [src_zmf2x6_core_hw_spi_master] */
#include "ametal.h"
#include "am_clk.h"
#include "am_vdebug.h"
#include "am_zmf2x6.h"

```

```

#include "am_zmf2x6_inst_init.h"
#include "demo_zmf2x6_entries.h"
#include "demo_zmf2x6_core_entries.h"

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_hw_spi_master_entry (void)
{
    am_kprintf("demo zmf2x6_core hw spi master!\r\n");

    /* 开启时钟 */
    am_clk_enable(CLK_SPI1);

    /* 配置引脚 */
    am_gpio_pin_cfg(PIOA_4, PIOA_4_SPI1_NSS | PIOA_4_AF_PP);
    am_gpio_pin_cfg(PIOA_5, PIOA_5_SPI1_SCK | PIOA_5_AF_PP);
    am_gpio_pin_cfg(PIOA_6, PIOA_6_SPI1_MISO | PIOA_6_INPUT_FLOAT);
    am_gpio_pin_cfg(PIOA_7, PIOA_7_SPI1_MOSI | PIOA_7_AF_PP);

    demo_zmf2x6_hw_spi_master_entry(ZMF2X6_SPI1,
                                     PIOA_4,
                                     am_clk_rate_get(CLK_SPI1));
}
/** [src_zmf2x6_core_hw_spi_master] */

```

2. 标准层实现

使用标准层实现 SPI 的中断方式通信，由于使用的是标准层例程，因此直接调用标准的例程函数 `demo_std_spi_master_entry` 即可。具体操作如程序清单 12.43 所示。

程序清单 12.43 SPI 的中断传输标准层 DEMO

```

/** [src_zmf2x6_core_std_spi_master_int] */
#include "ametal.h"
#include "am_vdebug.h"
#include "am_zmf2x6.h"
#include "am_zmf2x6_inst_init.h"
#include "demo_std_entries.h"
#include "demo_zmf2x6_core_entries.h"

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_std_spi_master_int_entry (void)
{
    AM_DBG_INFO("demo zmf2x6_core std spi master int!\r\n");
}

```

```
demo_std_spi_master_entry(am_zmf2x6_spi1_int_inst_init(), PIOA_4);  
}  
/** [src_zmf2x6_core_std_spi_master_int] */
```

广州立功科技-工业技术研发中心

第13章 通用外设驱动开发——TIMER

13.1 标准层了解

定时器标准层包含两个文件 `am_timer.h` 和 `am_timer.c` 文件，首先分析一下 `am_timer.h` 文件，在 `ametal\interface` 找到该文件打开。

13.1.1 了解相关类型定义

打开该文件，可以看到该文件主要定义了 2 个结构体和 1 个 `handle` 类型以及，如程序清单 13.1 所示，下面对每个结构体做详细介绍。

程序清单 13.1 标准接口文件类型定义

```
/**
 * \brief 定时器信息结构体
 */
typedef struct am_timer_info {
    uint8_t    counter_width;          /**< \brief 定时器位数          */
    uint8_t    chan_num;               /**< \brief 定时器通道数          */
    uint16_t   features;               /**< \brief 定时器特性          */
    /**
     * \brief 预分频器信息
     *
     * 该值表示预分频器支持的预分频值，该值的含义与定时器的预分频器特性相关，
     * 详见该特性的解释 #AM_TIMER_SUPPORT_ALL_PRESCALE_1_TO_MAX
     */
    uint32_t   prescaler;

} am_timer_info_t;

/**
 * \brief 定时器驱动函数结构体
 */
struct am_timer_drv_funcs {
    /**
     * \brief 获取定时器基础信息
     *
     * 该信息是定时器的固有属性，在驱动内部完成定义，该函数用以返回指向该常量信
     * 息的指针，以便用户使用。
     */
    const am_timer_info_t * (*pfn_info_get) (void *p_drv);

    /**
     * \brief 获取定时器的输入时钟频率（所有定时器通道共用一个输入频率）
     *
     * 输入时钟频率与时钟设置、具体硬件等相关，该频率并非定时器的计时频率，定时
     * 器的实际计时频率可能是通过一个预分频器分频后的频率。
     */
}
```

```

*/
int (*pfn_clkin_freq_get) (void *p_drv, uint32_t *p_freq);
/**
 * \brief 设置定时器的预分频值
 *
 * 若定时器不支持预分频，则 该函数可以为 NULL。
 */
int (*pfn_prescale_set) (void *p_drv, uint8_t chan, uint32_t prescale);
/**
 * \brief 获取定时器当前使用的预分频值
 *
 * 使用输入频率除以该预分频值，即可得到定时器的计数频率
 */
int (*pfn_prescale_get) (void *p_drv, uint8_t chan, uint32_t *p_prescale);
/**
 * \brief 获取当前定时器计数值
 *
 * 若定时器的位数不超过 32 位，则 p_count 为 uint32_t* 类型
 * 若定时器的位数超过 32 位，则 p_count 为 uint64_t* 类型
 */
int (*pfn_count_get) (void *p_drv, uint8_t chan, void *p_count);
/**
 * \brief 获取定时器翻转值
 *
 * 若定时器的位数不超过 32 位，则 p_count 为 uint32_t* 类型
 * 若定时器的位数超过 32 位，则 p_count 为 uint64_t* 类型
 */
int (*pfn_rollover_get) (void *p_drv, uint8_t chan, void *p_rollover);
/**
 * \brief 使能定时器，同时设定定时 count 值
 *
 * 若定时器的位数不超过 32 位，则 p_count 为 uint32_t* 类型
 * 若定时器的位数超过 32 位，则 p_count 为 uint64_t* 类型
 *
 * 计数器从 0 开始计数，达到该值时将调用用户设定的回调函数（若回调函数有效）
 * 同时，计数器的值回到 0 重新开始计数。
 */
int (*pfn_enable) (void *p_drv, uint8_t chan, void *p_count);
/** \brief 禁能定时器 */
int (*pfn_disable) (void *p_drv, uint8_t chan);
/** \brief 设置回调函数，每当一个定时周期完成时调用 */
int (*pfn_callback_set)(void *p_drv,
                        uint8_t chan,
                        void (*pfn_callback)(void *),

```

```

        void      *p_arg);
};

/**
 * \brief 定时器服务
 */
typedef struct am_timer_serv {
    /** \brief 定时器驱动函数结构体指针 */
    struct am_timer_drv_funcs *p_funcs;

    /** \brief 用于驱动函数的第一个参数 */
    void      *p_drv;
} am_timer_serv_t;
/** \brief 定时器标准服务操作句柄类型定义 */
typedef am_timer_serv_t *am_timer_handle_t;

```

1. struct am_timer_info 结构体

该结构体包含了定时器相关的信息，主要是定时器位数、定时器通道数量和定时器特性，后文提到的诸多函数所需信息和返回值信息，就与此对应。

2. struct am_timer_drv_funcs 结构体

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层需要实现 8 个函数：

- 时钟频率获取函数，获取输入时钟频率；
- 设置定时器的预分频值，无预分频函数 NULL；
- 获取预分频值；
- 获取当前计数值；
- 获取定时器翻转值；
- 使能定时器，同时设置计数值；
- 禁能定时器；
- 设置回调函数。

因此，在驱动层编写函数时，应该向标准接口层提供这 8 个函数。

3. typedef struct am_timer_serv 结构体

该结构体定义了标准的定时器服务，包含了上文提到的驱动函数结构体的指针和一个 void 型的指针。这个标准的定时器服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

● p_funcs 成员赋值

这是一个指向 struct am_timer_drv_funcs 结构体的指针，我们只需要在驱动层定义一个 struct am_timer_drv_funcs 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那 8 个函数地址。然后，将 struct am_timer_drv_funcs 结构体变量的地址赋值给 p_funcs 成员即可。

● p_drv 成员赋值

这个变量是 `void` 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动自行决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

4. `am_timer_handle_t` 的定义

这个是一个标准的操作句柄，本质上就是 `struct am_timer_serv` 指针类型，驱动层的初始化函数需要返回该类型的一个 `handle` 给用户。这样，用户获取到 `handle`，就可以在需要使用标准接口层函数时，将该 `handle` 传递给标准接口层了。标准接口就可以通过该 `handle`（`struct am_timer_serv` 结构体指针）访问到驱动层提供的函数了。

13.1.2 了解标准接口函数

在 `am_timer.h` 文件相关的结构体、类型定义后，就是具体函数的实现，详见程序清单 13.2。

程序清单 13.2 timer 标准接口函数

```
/**
 * \brief 获取定时器信息
 * \param[in] handle : 定时器标准服务操作句柄
 * \return 指向定时器信息常量的指针，若为 NULL，则表示信息获取失败
 */
am_static_inline
const am_timer_info_t * am_timer_info_get (am_timer_handle_t handle)
{
    return handle->p_funcs->pfn_info_get(handle->p_drv);
}
/**
 * \brief 获取当前定时器的输入时钟频率（所有定时器通道共用一个输入频率）
 *
 * 定时器的输入频率并非定时器的计数频率，实际定时器计数频率是由该频率分频的结果，
 * 支持的分频值详见特性：#AM_TIMER_SUPPORT_ALL_PRESCALE_1_TO_MAX 的描述。具体分频
 * 值可以通过 am_timer_prescale_set() 接口设置。
 *
 * \param[in] handle : 定时器标准服务操作句柄
 * \param[out] p_freq : 获取定时器输入频率的指针
 *
 * \return 无
 */
am_static_inline
int am_timer_clkin_freq_get (am_timer_handle_t handle, uint32_t *p_freq)
{
    return handle->p_funcs->pfn_clkin_freq_get(handle->p_drv, p_freq);
}
/**
 * \brief 设置定时器的预分频值
 */
```

```

* \param[in] handle    : 定时器标准服务操作句柄
* \param[in] chan      : 定时器通道
* \param[in] prescale : 预分频值
*
* \retval  AM_OK        : 设置成功
* \retval -AM_EINVAL    : 设置失败, 参数错误
* \retval -AM_ENOTSUP   : 设置失败, 不支持的预分频值, 支持的预分频值详见定时器信息
*/
am_static_inline
int am_timer_prescale_set (am_timer_handle_t handle,
                           uint8_t          chan,
                           uint32_t         prescale)
{
    return handle->p_funcs->pfn_prescale_set(handle->p_drv, chan, prescale);
}
/**
* \brief 设置定时器的预分频值
*
* \param[in] handle    : 定时器标准服务操作句柄
* \param[in] chan      : 定时器通道
* \param[in] prescale : 预分频值
*
* \retval  AM_OK        : 设置成功
* \retval -AM_EINVAL    : 设置失败, 参数错误
* \retval -AM_ENOTSUP   : 设置失败, 不支持的预分频值, 支持的预分频值详见定时器信息
*/
am_static_inline
int am_timer_prescale_get (am_timer_handle_t handle,
                           uint8_t          chan,
                           uint32_t         *p_prescale)
{
    return handle->p_funcs->pfn_prescale_get(handle->p_drv, chan, p_prescale);
}
/**
* \brief 获取定时器当前的计数频率
*
* \param[in] handle    : 定时器标准服务操作句柄
* \param[in] chan      : 定时器通道
* \param[out] p_freq   : 用于获取定时器当前计数频率的指针
*
* \retval  AM_OK        : 获取定时器当前计数值成功
* \retval -AM_EINVAL    : 获取失败, 参数错误
*/

```

```

int am_timer_count_freq_get (am_timer_handle_t handle,
                             uint8_t          chan,
                             uint32_t          *p_freq);

/**
 * \brief 获取定时器当前的计数值
 *
 * 返回的计数值应该是假设定时器工作在向上计数模式下的当前计数值
 *
 * \param[in]  handle   : 定时器标准服务操作句柄
 * \param[in]  chan     : 定时器通道
 * \param[out] p_count  : 用于获取定时器当前计数值的指针
 *
 * \retval  AM_OK       : 获取定时器当前计数值成功
 * \retval -AM_EINVAL   : 获取失败, 参数错误
 */
am_static_inline
int am_timer_count_get (am_timer_handle_t handle,
                        uint8_t          chan,
                        uint32_t          *p_count)
{
    return handle->p_funcs->pfn_count_get(handle->p_drv, chan, (void*)p_count);
}

/**
 * \brief 获取定时器翻转值
 *
 * \param[in]  handle   : 定时器标准服务操作句柄
 * \param[in]  chan     : 定时器通道
 * \param[out] p_rollover : 用于获取定时器翻转值的指针
 *
 * \retval  AM_OK       : 获取定时器翻转值成功
 * \retval -AM_EINVAL   : 获取失败, 参数错误
 */
am_static_inline
int am_timer_rollover_get (am_timer_handle_t handle,
                           uint8_t          chan,
                           uint32_t          *p_rollover)
{
    return handle->p_funcs->pfn_rollover_get(handle->p_drv,
                                           chan,
                                           (void *)p_rollover);
}

```

```

/**
 * \brief 禁能定时器（停止计时）
 *
 * \param[in] handle : 定时器标准服务操作句柄
 * \param[in] chan   : 定时器通道
 *
 * \retval AM_OK      : 禁能定时器成功
 * \retval -AM_EINVAL : 禁能失败, 参数错误
 */
am_static_inline
int am_timer_disable (am_timer_handle_t handle, uint8_t chan)
{
    return handle->p_funcs->pfn_disable(handle->p_drv, chan);
}

/**
 * \brief 使能定时器，同时设定定时 count 值
 *
 * \param[in] handle : 定时器标准服务操作句柄
 * \param[in] chan   : 定时器通道
 * \param[in] count  : 定时 count 值
 *
 * \retval AM_OK      : 使能成功
 * \retval -AM_EINVAL : 失败, 参数错误
 *
 * \note 定时器计数值达到设定的定时 count 值时，会产生中断。同时，复位计数值
 *       为 0，紧接着继续计时。
 */
am_static_inline
int am_timer_enable (am_timer_handle_t handle, uint8_t chan, uint32_t count)
{
    return handle->p_funcs->pfn_enable(handle->p_drv, chan, (void *)&count);
}

/**
 * \brief 设置回调函数，每当一个定时周期完成时调用
 *
 * \param[in] handle      : 定时器标准服务操作句柄
 * \param[in] chan        : 定时器通道
 * \param[in] pfn_callback : 回调函数
 * \param[in] p_arg       : 回调函数的用户参数
 *
 * \retval AM_OK      : 设置成功
 * \retval -AM_EINVAL : 设置失败, 参数错误
 */

```

```

*
*/
am_static_inline
int am_timer_callback_set (am_timer_handle_t  handle,
                           uint8_t           chan,
                           void               (*pfn_callback)(void *),
                           void               *p_arg)
{
    return handle->p_funcs->pfn_callback_set(handle->p_drv,
                                             chan,
                                             pfn_callback,
                                             p_arg);
}

/**
 * \brief 获取 33~64 位定时器的当前计数值
 *
 * 返回的计数值应该是假设定时器工作在向上计数模式下的当前计数值
 *
 * \param[in]  handle   : 定时器标准服务操作句柄
 * \param[in]  chan     : 定时器通道
 * \param[out] p_count  : 用于获取定时器当前计数值的指针
 *
 * \retval  AM_OK       : 获取到的当前定时器计数值
 * \retval -AM_EINVAL   : 获取失败, 参数错误
 *
 * \note 如果定时器位数在 33 到 64 之间, 必须调用该函数获取定时器的当前计数值
 */
am_static_inline
int am_timer_count_get64 (am_timer_handle_t  handle,
                           uint8_t           chan,
                           uint64_t          *p_count)
{
    return handle->p_funcs->pfn_count_get(handle->p_drv, chan, (void *)p_count);
}

/**
 * \brief 获取 33~64 位定时器的翻转值
 *
 * \param[in]  handle   : 定时器标准服务操作句柄
 * \param[in]  chan     : 定时器通道
 * \param[out] p_rollover : 用于获取定时器翻转值的指针
 *
 * \retval  AM_OK       : 获取翻转值成功

```

```

* \retval -AM_EINVAL      : 获取翻转值失败, 参数错误
*
* \note 如果定时器位数在 33 到 64 之间, 必须调用该函数获取翻转值
*/
am_static_inline
int am_timer_rollover_get64 (am_timer_handle_t  handle,
                             uint8_t           chan,
                             uint64_t           *p_rollover)
{
    return handle->p_funcs->pfn_rollover_get(handle->p_drv,
                                           chan,
                                           (void *)p_rollover);
}

/**
* \brief 使能 33 到 64 位定时器, 并设置定时 count 值
*
* \param[in] handle : 定时器标准服务操作句柄
* \param[in] chan   : 定时器通道
* \param[in] count  : 定时器 count 值
*
* \retval  AM_OK      : 使能成功
* \retval -AM_EINVAL : 使能失败, 参数错误
*
* \note 如果定时器位数在 33 到 64 之间, 必须调用该函数使能定时器
*/
am_static_inline
int am_timer_enable64 (am_timer_handle_t handle, uint8_t chan, uint64_t count)
{
    return handle->p_funcs->pfn_enable(handle->p_drv, chan, (void *)&count);
}

/**
* \brief 使能定时器, 直接设定定时时间
*
* 一般情况下, 用户需要使用一个定时时间, 需要根据输入频率和定时器位数, 计算好预
* 分频值和定时 count 值, 然后使用 am_timer_prescale_set() 设置预分频值, 最后使
* 用 am_timer_enable() 或 am_timer_enable64() 设置定时 count 值并使能定时器。
*
* 手动计算需要考虑定时器输入频率、定时器位数、定时器特性 (部分定时器可能不支持
* 或只支持几种预分频值), 较为繁琐。基于此, 用户可以直接使用该函数按照指定的时
* 间值 (单位: us) 使能定时器。系统会自动根据定时器位数、定时器特性等设置好定时
* 器。
*
*/

```

```

* 计时时间可能存在偏差，实际定时时间可以由 计数频率 和 翻转值 得到：翻转值乘以
* 计数频率的倒数即可。
*
* 由于定时时间到会产生中断，因此，不建议将该时间设置得过短，至少应在几百 us，建议
* 设置到 ms 级别。
*
* \param[in] handle：定时器标准服务操作句柄
* \param[in] chan   ：定时器通道
* \param[in] nus    ：设置的定时时间（单位：us）
*
* \retval  AM_OK      ：使能成功
* \retval -AM_EINVAL：失败，参数错误
* \retval -AM_NOTSUP ：不支持该定时时间，时间太长（定时器位数不够）
*/
int am_timer_enable_us (am_timer_handle_t handle, uint8_t chan, uint32_t nus);

```

很简单的 8 个函数，全都是用内联函数就搞定了。因为它本身就没做什么事情，只是简单的调用了一下驱动层提供的函数而已，每个函数都有详细注释，传入参数和返回值。可以看到，在调用驱动层提供的函数时，就是将 struct am_timer_serv 结构体中的 p_drv 成员作为第一个参数传递。

到此我们就分析完了 timer 外设标准层文件，清楚了驱动层需要为标准接口层提供哪些内容。

13.2 HW 层开发

现在我们打开 ZMF2x6 芯片用户手册，先了解一下该芯片 timer 外设，下面对照手册完成 timer 外设的 HW 层的开发。

13.2.1 添加寄存器列表

HW 层就是与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读该外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMF2x6 用户手册的定时器章节，可以看到，定时器章节部分，共有 20 个寄存器，如图 13.1 所示。

16.4 寄存器描述

表 48. TIM1 寄存器概览

Offset	Acronym	Register Name	Reset	Section
0x00	TIMx_CR1	控制寄存器 1	0x00000000	小节 16.4.1
0x04	TIMx_CR2	控制寄存器 2	0x00000000	小节 16.4.2
0x08	TIMx_SMCR	从模式控制寄存器	0x00000000	小节 16.4.3
0x0C	TIMx_DIER	DMA/中断使能寄存器	0x00000000	小节 16.4.4
0x10	TIMx_SR	状态寄存器	0x00000000	小节 16.4.5
0x14	TIMx_EGR	事件产生寄存器	0x00000000	小节 16.4.6
0x18	TIMx_CCMR1	捕捉/比较模式寄存器 1	0x00000000	小节 16.4.7
0x1C	TIMx_CCMR2	捕捉/比较模式寄存器 2	0x00000000	小节 16.4.8
0x20	TIMx_CCER	捕捉/比较使能寄存器	0x00000000	小节 16.4.9
0x24	TIMx_CNT	计数器	0x00000000	小节 16.4.10
0x28	TIMx_PSC	预分频率器	0x00000000	小节 16.4.11
0x2C	TIMx_ARR	自动装载寄存器	0x00000000	小节 16.4.12
0x30	TIMx_RCR	重复计数寄存器	0x00000000	小节 16.4.13
0x34	TIMx_CCR1	捕获/比较寄存器 1	0x00000000	小节 16.4.14
0x38	TIMx_CCR2	捕获/比较寄存器 2	0x00000000	小节 16.4.15
0x3C	TIMx_CCR3	捕获/比较寄存器 3	0x00000000	小节 16.4.16

ZLG

©2020 Guangzhou ZHIYUAN Micro Electronics Co., Ltd.

207/624

ZMF2x6

基于 ARM Cortex M0 核的 32 位微控制器

User Manual

Offset	Acronym	Register Name	Reset	Section
0x40	TIMx_CCR4	捕获/比较寄存器 4	0x00000000	小节 16.4.17
0x44	TIMx_BDTR	刹车和死区寄存器	0x00000000	小节 16.4.18
0x48	TIMx_DCR	DMA 控制寄存器	0x00000000	小节 16.4.19
0x4C	TIMx_DMAR	连续模式的 DMA 地址	0x00000000	小节 16.4.20

图 13.1 Timer 相关寄存器

我们也大致了解了各个寄存器功能以及要配置的各个功能，其他寄存器该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，如程序清单 13.3 所示。

程序清单 13.3 Timer 寄存器结构体定义

```
/**
 * \brief 定时器寄存器块结构体
 *
 * \note 只有 TIM1 存在 dmar 以下的寄存器
 *       TIM3、2 没有 rcr 及 bdtr 寄存器
 *       TIM14 只有一个 cr、ccmr 及 ccr 寄存器，没有 smcr、rcr 及 bdtr 以下的寄存器
 *       TIM16、17 只有一个 ccmr 及 ccr 寄存器，没有 smcr
 */
typedef struct amhw_zmf2x6_tim {
```



```

__IO uint32_t cr[2];          /**< \brief 控制寄存器 */
__IO uint32_t smcr;           /**< \brief 从模式控制寄存器 */
__IO uint32_t dier;           /**< \brief DMA/中断使能寄存器 */
__IO uint32_t sr;             /**< \brief 状态寄存器 */
__IO uint32_t egr;            /**< \brief 事件产生寄存器 */
__IO uint32_t ccmr[2];        /**< \brief 捕捉/比较模式寄存器 */
__IO uint32_t ccer;           /**< \brief 捕捉/比较使能寄存器 */
__IO uint32_t cnt;            /**< \brief 计数器 */
__IO uint32_t psc;            /**< \brief 预分频器 */
__IO uint32_t arr;            /**< \brief 自动装载寄存器 */
__IO uint32_t rcr;            /**< \brief 重复计数寄存器 */
__IO uint32_t ccr[4];         /**< \brief 捕获/比较寄存器 */
__IO uint32_t bdtr;           /**< \brief 刹车和死区寄存器 */
__IO uint32_t dcr;            /**< \brief 定时器 DMA 控制寄存器 */
__IO uint32_t dmar;           /**< \brief 连续模式的 DMA 地址 */
__IO uint32_t reserve0;       /**< \brief 保留 */
__IO uint32_t ccmr3;          /**< \brief 捕捉/比较模式寄存器 3 */
__IO uint32_t ccr5;           /**< \brief 捕获/比较寄存器 5 */
__IO uint32_t pder;           /**< \brief PWM 移相/DMA 更新请求使能寄存器 */
__IO uint32_t ccrfall[5];     /**< \brief PWM 移相向下计数捕获/比较寄存器 */
} amhw_zmf2x6_tim_t;

```

这个小节我们定义了 Timer 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

13.2.2 编写寄存器的相关函数

1. TIMx_CR1（控制寄存器）

通过手册，我们了解到该寄存器有含义的寄存器共有 8 个，如表 13.1 所示。

表 13.1 TIMx_CR1 寄存器分析

位	标识符	含义
9:8	CDK【1:0】	时钟分频因子。
7	ARPE	自动重载预装载允许位。
6:5	CMS【1:0】	对齐方式选择。
4	DIR	计数方向
3	OPM	单脉冲模式
2	URS	更新事件请求源
1	UDIS	禁止更新
0	CEN	允许计数器

由此我们确定编写 8 个函数，下面我们分别实现。

● CR1_CDK 函数

该位为时钟分频因子，由两个位组成，写 0~3 分别对应不同的分频参数，函数实现详见程序清单 13.4。

程序清单 13.4 CR1_CDK 操作函数

```
/**
 * \brief 时钟分频因子设置
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ckd_opt : 时钟分频因子选择
 *
 *          0: tDTS = tCK_INT(定时器计数时钟)
 *          1: tDTS = 2 * tCK_INT(定时器计数时钟)
 *          2: tDTS = 4 * tCK_INT(定时器计数时钟)
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ckd_set (amhw_zmf2x6_tim_t *p_hw_tim, uint8_t ckd_opt)
{
    AM_BITS_SET(p_hw_tim->cr[0], 8, 2, ckd_opt);
}
```

● CR1_ARPE 函数

该位为自动重载允许使能位，提供两个函数，使能和禁能，函数实现详见程序清单 13.5。

程序清单 13.5 CR1_ARPE 操作函数

```
/**
 * \brief 自动重载预装载允许使能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_arpe_enable (amhw_zmf2x6_tim_t *p_hw_tim)
{
    AM_BIT_MODIFY(p_hw_tim->cr[0], 7, 1);
}

/**
 * \brief 自动重载预装载允许禁能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 *
 * \return 无
 */
```

```
am_static_inline
void amhw_zmf2x6_tim_arpe_disable (amhw_zmf2x6_tim_t *p_hw_tim)
{
    AM_BIT_MODIFY(p_hw_tim->cr[0], 7, 0);
}
```

● CR1_CMS 函数

该位为对齐方式选择，函数实现详见程序清单 13.6。

程序清单 13.6 CR1_CMS 操作函数

```
/**
 * \brief 对齐模式设置
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] cms_opt : 对齐模式选择
 *
 *          0: 边沿对齐模式
 *          1: 中央对齐模式 1
 *          2: 中央对齐模式 2
 *          3: 中央对齐模式 3
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_cms_set (amhw_zmf2x6_tim_t *p_hw_tim, uint8_t cms_opt)
{
    AM_BITS_SET(p_hw_tim->cr[0], 5, 2, cms_opt);
}
```

● CR1_DIR 函数

该位为计数器计数方向选择位，函数实现详见程序清单 13.7。

程序清单 13.7 CR1_DIR 操作函数

```
/**
 * \brief 设置计数的方向
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] dir_opt : 计数方向选择
 *
 *          0: 计数器向上计数
 *          1: 计数器向下计数
 *
 * \note :当计数器配置为中央对齐模式或编码器模式时，该位为只读
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_dir_set (amhw_zmf2x6_tim_t *p_hw_tim, uint8_t dir_opt)
```

```
{
    AM_BIT_MODIFY(p_hw_tim->cr[0], 4, dir_opt);
}
```

- CR1_OPM 函数

该位为单脉冲模式选择位，函数实现详见程序清单 13.8。

程序清单 13.8 CR1_OPM 操作函数

```
am_static_inline
void amhw_zmf2x6_tim_opm_set (amhw_zmf2x6_tim_t *p_hw_tim, uint8_t opm_opt)
{
    AM_BIT_MODIFY(p_hw_tim->cr[0], 3, opm_opt);
}
```

- CR1_URS 函数

该位为单脉冲模式选择位，函数实现详见程序清单 13.9。

程序清单 13.9 CR1_URS 操作函数

```
am_static_inline
void amhw_zmf2x6_tim_urs_set (amhw_zmf2x6_tim_t *p_hw_tim, uint8_t urs_opt)
{
    AM_BIT_MODIFY(p_hw_tim->cr[0], 2, urs_opt);
}
```

- CR1_UDIS 函数

该位为更新事件选择位，函数实现详见程序清单 13.10。

程序清单 13.10 CR1_UDIS 操作函数

```
/**
 * \brief 清除 UDIS 位，允许更新事件 UEV 发生
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_udis_enable (amhw_zmf2x6_tim_t *p_hw_tim)
{
    AM_BIT_MODIFY(p_hw_tim->cr[0], 1, 0);
}
/**
 * \brief 置 UDIS 位，禁止更新事件 UEV 发生
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 *
 * \return 无
 */
```

```
*/
am_static_inline
void amhw_zmf2x6_tim_u-disable (amhw_zmf2x6_tim_t *p_hw_tim)
{
    AM_BIT_MODIFY(p_hw_tim->cr[0], 1, 1);
}
```

- CR1_CEN 函数
该位为计数器计数使能和禁能位，函数实现详见程序清单 13.11。

程序清单 13.11 CR1_CEN 操作函数

```
/**
 * \brief 使能定时器(开始计数)
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_enable (amhw_zmf2x6_tim_t *p_hw_tim)
{
    AM_BIT_MODIFY(p_hw_tim->cr[0], 0, 1);
}

/**
 * \brief 禁止定时器(停止计数)
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_disable (amhw_zmf2x6_tim_t *p_hw_tim)
{
    AM_BIT_MODIFY(p_hw_tim->cr[0], 0, 0);
}
```

2. TIMx_CR2（控制寄存器）
通过手册，我们了解到该寄存器有含义的寄存器共有 13 个，如表 13.2 所示。

表 13.2 TIMx_CR2 寄存器分析

位	标识符	含义
16	OSI5	输出空闲状态 5。
14	OSI4	输出空闲状态 4。
13	OSI3N	输出空闲状态 3N。

12	OSI3	输出空闲状态 3。
11	OSI2N	输出空闲状态 2N。
10	OSI2	输出空闲状态 2。
9	OSI1N	输出空闲状态 1N。
8	OSI1	输出空闲状态 1。
7	TI1S	TI1 选择
6:4	MMS	主模式选择
3	CCDS	捕获比较的 DMA 选择
2	CCUS	捕获比较更新控制选择
0	CCPC	捕获比较预装载控制位

由此我们确定编写 8 个函数，下面我们分别实现。

● CR1_OSI 控制函数

OSI 控制位，8 种模式可以配置，先定义这 8 种结构的状态枚举，函数实现详见程序清单 13.12。

程序清单 13.12 CR1_OSI 操作函数

```
/**
 * \brief 输出空闲状态设置枚举类型
 */
typedef enum amhw_zmf2x6_tim_ois {
    AMHW_ZMF2X6_TIM_OIS5 = (1UL << 16), /**< \brief 输出空闲状态 5(OC5 输出) */
    AMHW_ZMF2X6_TIM_OIS4 = (1UL << 14), /**< \brief 输出空闲状态 4(OC4 输出) */
    AMHW_ZMF2X6_TIM_OIS3N = (1UL << 13), /**< \brief 输出空闲状态 3(OC3N 输出) */
    AMHW_ZMF2X6_TIM_OIS3 = (1UL << 12), /**< \brief 输出空闲状态 3(OC3 输出) */
    AMHW_ZMF2X6_TIM_OIS2N = (1UL << 11), /**< \brief 输出空闲状态 2(OC2N 输出) */
    AMHW_ZMF2X6_TIM_OIS2 = (1UL << 10), /**< \brief 输出空闲状态 2(OC2 输出) */
    AMHW_ZMF2X6_TIM_OIS1N = (1UL << 9), /**< \brief 输出空闲状态 1(OC1N 输出) */
    AMHW_ZMF2X6_TIM_OIS1 = (1UL << 8), /**< \brief 输出空闲状态 1(OC1 输出) */
} amhw_zmf2x6_tim_ois_t;

/**
 * \brief 输出空闲状态置 1
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ois_opt : 输出空闲状态选择，值为 amhw_zmf2x6_tim_ois_t 这一类型枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ois_set(amhw_zmf2x6_tim_t *p_hw_tim,
                             amhw_zmf2x6_tim_ois_t ois_opt)
{
    p_hw_tim->cr[1] |= ois_opt;
}
```

```

}
/**
 * \brief 输出空闲状态清 0
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ois_opt : 出空闲状态选择, 值为 amhw_zmf2x6_tim_ois_t 这一类型枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ois_clr (amhw_zmf2x6_tim_t *p_hw_tim,
                             amhw_zmf2x6_tim_ois_t ois_opt)
{
    p_hw_tim->cr[1] &= ~ois_opt;
}

```

● CR1_TIS1 控制函数

CR1_TIS1 用于选择管脚输入，函数实现详见程序清单 13.13。

程序清单 13.13 CR1_TIS1 操作函数

```

/**
 * \brief TI1 设置
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ti1s_opt : TI1 选择
 *
 *          0: TIMx_CH1 管脚连到 TI1 输入
 *          1: TIMx_CH1、TIMx_CH2 和 TIMx_CH3 管脚经异或后连到 TI1 输入
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_tils_set (amhw_zmf2x6_tim_t *p_hw_tim, uint8_t ti1s_opt)
{
    AM_BIT_MODIFY(p_hw_tim->cr[1], 7, ti1s_opt);
}

```

● CR1_MMS 模式选择函数

CR1_MMS 模式选择，函数实现详见程序清单 13.14。

程序清单 13.14 CR1_MMS1 操作函数

```

/**
 * \brief 主模式设置
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] mms_opt : 主模式选择
 *
 *          0: 复位; 1: 使能; 2: 更新; 3: 比较脉冲
 *          4: 比较 - OC1REF 信号被用于作为触发输出 (TRGO)

```

```

*          5: 比较 - OC2REF 信号被用于作为触发输出 (TRGO)
*          6: 比较 - OC3REF 信号被用于作为触发输出 (TRGO)
*          7: 比较 - OC4REF 信号被用于作为触发输出 (TRGO)
* \return 无
*/
am_static_inline
void amhw_zmf2x6_tim_mms_set (amhw_zmf2x6_tim_t *p_hw_tim, uint8_t mms_opt)
{
    AM_BITS_SET(p_hw_tim->cr[1], 4, 3, mms_opt);
}

```

● CR1_CCDS 模式选择函数

CR1_CCDS 模式选择 DMA 用于捕获或比较，函数实现详见程序清单 13.15。

程序清单 13.15 CR1_CCDS 操作函数

```

/**
* \brief 捕获/比较的 DMA 设置
*
* \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
* \param[in] ccds_opt : 捕获/比较的 DMA 选择选择
*
*          0: 当发生 CCx 事件时，送出 CCx 的 DMA 请求
*          1: 当发生更新事件时，送出 CCx 的 DMA 请求
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_tim_ccds_set (amhw_zmf2x6_tim_t *p_hw_tim, uint8_t ccds_opt)
{
    AM_BIT_MODIFY(p_hw_tim->cr[1], 3, ccds_opt);
}

```

● CR1_CCUS 模式选择函数

CR1_CCUS 模式选择更新事件用于捕获或比较，函数实现详见程序清单 13.16。

程序清单 13.16 CR1_CCUS 操作函数

```

/**
* \brief 捕获/比较控制更新设置
*
* \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
* \param[in] ccus_opt : 捕获/比较控制更新选择
*
*          0: 捕获/比较控制位是预装载的 (CCPC = 1)，只能通过设置 COM 位更新
*          1: 捕获/比较控制位是预装载的 (CCPC = 1)，可以通过设置 COM 位或 TRGI
    上的一个上升沿更新
*
* \note : 该位只对具有互补输出的通道起作用。
*

```



```

* \return 无
*/
am_static_inline
void amhw_zmf2x6_tim_ccus_set (amhw_zmf2x6_tim_t *p_hw_tim, uint8_t ccus_opt)
{
    AM_BIT_MODIFY(p_hw_tim->cr[1], 2, ccus_opt);
}

```

- CR1_CCPC 捕获比较预装载控制函数

CR1_CCPC 捕获比较预装载控制，函数实现详见程序清单 13.17。

程序清单 13.17 CR1_CCPC 操作函数

```

/**
 * \brief 捕获/比较预装载控制使能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 *
 * \note : 该位只对具有互补输出的通道起作用。
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ccpc_enable (amhw_zmf2x6_tim_t *p_hw_tim)
{
    AM_BIT_MODIFY(p_hw_tim->cr[1], 0, 1);
}

```

还有诸多寄存器，可按上述样例逐一添加，定时器外设寄存器众多，本节不再一一罗列。

13.2.3 添加其他功能性函数

我们回顾一下所有的函数，对所有寄存器都提供了操作函数，在每个寄存器的操作函数编写完毕后，可能 HW 层的接口还有所欠缺，需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到，可以暂时不用考虑，直接进入驱动层的开发。

13.2.4 添加指向寄存器块的指针

可以看到，在 HW 层编写的函数中，第一个参数均为指向 Timer 寄存器块的指针，只需要把 Timer 的基地址使用(amhw_zmf2x6_tim_t*)强制转换即可作为这些函数的第一个参数。这样转换后，操作结构体的成员，实际上就是操作具体地址上的寄存器了。

在 AMetal 中，由于这些内容与具体型号的芯片相关，这样的宏统一定义到 zmf2x6_periph_map.h 文件中，如程序清单 13.18 所示。

程序清单 13.18 外设中加入 UART 的定义

```

/** \brief 定时器(TIMER)寄存器块指针 */
#define ZMF2X6_TIMER ((amhw_zmf2x6_uart_t *)ZMF2X6_TIMER_BASE)

```

后续用户或驱动层使用 HW 层接口时，第一个参数直接使用 ZMF2X6_TIMx 即可。至

此，整个 HW 层的开发工作就结束了，如果后续开发中发现缺少某些函数，还可以再添加。

13.3 驱动层开发

参考 5.3.1 章节，新建 am_zmf2x6_tim.h 和 am_zmf2x6_tim.c 驱动文件，搭建相应的文件框架，并按规则放到对应的路径下。

13.3.1 编写结构性程序

驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备（TIMER 就是一个设备）都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

可以暂时定义设备信息结构体如程序清单 13.19 所示。后续可以根据实际情况修改。

程序清单 13.19 TIMER 设备信息结构体定义

```
/**
 * \brief TIM 定时功能相关的设备信息
 */
typedef struct am_zmf2x6_tim_timing_devinfo {

    uint32_t          tim_regbase; /**< \brief TIM 寄存器块基址 */

    uint8_t           inum;         /**< \brief TIM 中断号 */

    int               clk_num;      /**< \brief 时钟 ID */

    amhw_zmf2x6_tim_type_t tim_type; /**< \brief 定时器类型 */

    /** \brief 平台初始化函数，如打开时钟，配置引脚等工作 */
    void              (*pfn_plfm_init)(void);

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

} am_zmf2x6_tim_timing_devinfo_t;
```

● 设备结构体定义

由于 TIMER 设备需要提供标准的 TIMER 服务，因此，该设备至少包含一个 TIMER 标准服务结构体成员，也就是 am_tim_serv_t 类型成员，其次，为了在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 13.20 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 13.20 TIMER 设备结构体定义

```
/**
 * \brief TIM 定时功能设备
 */
typedef struct am_zmf2x6_tim_timing_dev {

    am_timer_serv_t timer_serv;          /**< \brief 标准定时(Timer)服务 */

    void (*pfn_callback)(void *);        /**< \brief 回调函数 */
    void *p_arg;                          /**< \brief 回调函数的用户参数 */

    /** \brief 指向 TIM(定时功能)设备信息常量的指针 */
    const am_zmf2x6_tim_timing_devinfo_t *p_devinfo;

} am_zmf2x6_tim_timing_dev_t;
```

2. 搭建函数框架

函数主要分为两部分一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于定时器这种已经有标准接口层的函数，通常驱动层只提供两个函数给用户，初始化函数和解初始化函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为用于标准接口的 handle。

解初始化函数主要是当不需要再使用某个外设时，释放掉其占用的相关资源，参数为设备指针，无返回值。示例代码如程序清单 13.21 所示。

程序清单 13.21 驱动层函数定义示例

```
/**
 * \brief 初始化 TIM 为定时功能
 *
 * \param[in] p_dev      : 指向 TIM(定时功能)设备的指针
 * \param[in] p_devinfo : 指向 TIM(定时功能)设备信息常量的指针
 *
 * \return Timer 标准服务操作句柄，值为 NULL 时表明初始化失败
 */
am_timer_handle_t
am_zmf2x6_tim_timing_init (am_zmf2x6_tim_timing_dev_t *p_dev,
                           const am_zmf2x6_tim_timing_devinfo_t *p_devinfo);

/**
 * \brief 不使用 TIM 定时功能时，解初始化 TIM 定时功能，释放相关资源
```

```

*
* \param[in] handle : am_zmf2x6_tim_timing_init() 初始化函数获得的 Timer 服务句柄
*
* \return 无
*/
void am_zmf2x6_tim_timing_deinit (am_timer_handle_t handle);

```

● 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 3 个函数，根据标准接口层中的 struct am_tim_drv_funcs 结构体，我们需要在.c 文件中定义 3 个函数。

由于这些函数全部是放在结构体中，通过函数指针的形式提供给上层，并不需要将这些函数声明到.h 文件中供用户直接使用，因此全部以 static 的形式定义成内部函数。定义好函数后，再定义一个 struct am_tim_drv_funcs 结构体，将函数放在其中。示例代码如程序清单 13.22 所示。

程序清单 13.22 提供给标准层函数定义

```

/* TIM 定时器驱动函数 */
static const struct am_timer_drv_funcs __g_tim_timing_drv_funcs = {
    __tim_timing_info_get,
    __tim_timing_clkin_freq_get,
    __tim_timing_prescale_set,
    __tim_timing_prescale_get,
    __tim_timing_count_get,
    __tim_timing_rollover_get,
    __tim_timing_enable,
    __tim_timing_disable,
    __tim_timing_callback_set,
};

```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。可以编译一下，应该没有错误的。如果有错误，表明结构没有搭建好。可能存在警告，毕竟有些变量没有使用，可以暂时不用理会。做到这里，其实驱动层已经开发一大半了。

13.3.2 驱动函数实现

通过上面的函数框架搭建，共计 11 个函数，9 个功能性函数面向标准层，2 个初始解初始化函数面向用户，下面逐步实现。

1. am_zmf2x6_tim_timing_init 函数

该函数主要完成两个功能：

- 初始化设备中的标准服务，将成员赋值，确定 p_drv 的值，p_drv 的值确定了提供给标准接口层的那 3 个函数的第一个参数是什么。那 3 个函数才能根据这儿的设置去使用。（void *类型无法使用，必须转换为具体的一种类型才能使用），通常情况下，p_drv 都是设置为设备的地址，这样方便访问整个设备的相关内容。
- 初始化硬件相关设置，该函数的示例代码如程序清单 13.23 所示。

程序清单 13.23 am_zmf2x6_tim_timing_init 函数示例代码

```

am_timer_handle_t am_zmf2x6_tim_timing_init (
    am_zmf2x6_tim_timing_dev_t      *p_dev,
    const am_zmf2x6_tim_timing_devinfo_t  *p_devinfo)
{
    amhw_zmf2x6_tim_t *p_hw_tim = NULL;
    if (NULL == p_dev || NULL == p_devinfo ) {
        return NULL;
    }
    if (p_devinfo->pfn_plfm_init) {
        p_devinfo->pfn_plfm_init();
    }
    p_dev->p_devinfo      = p_devinfo;
    p_hw_tim              = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    p_dev->timer_serv.p_funcs = (struct am_timer_drv_funcs *)&__g_tim_timing_drv_funcs;
    p_dev->timer_serv.p_drv   = p_dev;
    p_dev->pfn_callback = NULL;
    p_dev->p_arg          = NULL;
    if ((p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE0) ||
        (p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE1)) {
        /* 边沿对齐模式 */
        amhw_zmf2x6_tim_cms_set(p_hw_tim, 0);
        /* 向上计数 */
        amhw_zmf2x6_tim_dir_set(p_hw_tim, 0);
    }
    /* 设置时钟分割:TCTS = Tck_tin */
    amhw_zmf2x6_tim_ckd_set(p_hw_tim, 0);
    /* 允许更新事件 */
    amhw_zmf2x6_tim_udis_enable(p_hw_tim);
    am_int_connect(p_dev->p_devinfo->inum, __tim_irq_handler, (void *)p_dev);
    am_int_enable(p_dev->p_devinfo->inum);
    return &(p_dev->timer_serv);
}

```

2. am_zmf2x6_tim_timing_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放，同时，由于 TIMER 一旦使能将无法关闭，所以也无法释放硬件资源。

程序清单 13.24 am_zmf2x6_tim_timing_deinit 函数示例代码

```

void am_zmf2x6_tim_timing_deinit (am_timer_handle_t handle)
{
    am_zmf2x6_tim_timing_dev_t *p_dev      = (am_zmf2x6_tim_timing_dev_t *)handle;
    amhw_zmf2x6_tim_t          *p_hw_tim = NULL;

    if (p_dev == NULL || p_dev->p_devinfo == NULL ) {

```

```

        return ;
    }

    p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    /* 关闭更新中断 */
    amhw_zmf2x6_tim_int_disable(p_hw_tim, AMHW_ZMF2X6_TIM_UIE);
    /* 关闭时钟，停止计数 */
    amhw_zmf2x6_tim_disable(p_hw_tim);
    am_int_disconnect(p_dev->p_devinfo->inum, __tim_irq_handler, (void *)p_dev);
    am_int_disable(p_dev->p_devinfo->inum);
    p_dev->timer_serv.p_drv = NULL;
    if (p_dev->p_devinfo->pfn_plfm_deinit) {
        p_dev->p_devinfo->pfn_plfm_deinit();
    }
}

```

3. __tim_timing_clkin_freq_get 函数

该函数用于获取当前系统时钟频率。

程序清单 13.25 __tim_timing_clkin_freq_get 函数示例代码

```

static int __tim_timing_clkin_freq_get (void *p_drv, uint32_t *p_freq)
{
    am_zmf2x6_tim_timing_dev_t *p_dev = (am_zmf2x6_tim_timing_dev_t *)p_drv;

    if ((p_dev == NULL) || (p_freq == NULL)) {
        return -AM_EINVAL;
    }

    /* 获取当前系统时钟频率，systick 使用的是系统时钟 */
    *p_freq = am_clk_rate_get(p_dev->p_devinfo->clk_num);

    return AM_OK;
}

```

4. __tim_timing_prescale_set 函数

该函数用于设置预分频值。

程序清单 13.26 __tim_timing_prescale_set 函数示例代码

```

uint32_t prescale)
{
    am_zmf2x6_tim_timing_dev_t *p_dev = (am_zmf2x6_tim_timing_dev_t *)p_drv;
    amhw_zmf2x6_tim_t *p_hw_tim;

    if ((p_drv == NULL) || (chan != 0)) {
        return -AM_EINVAL;
    }

    p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    if ((prescale != 0) && (prescale <= 65536)) {

```

```

    amhw_zmf2x6_tim_prescale_set(p_hw_tim, prescale - 1); /* 设置分频值 */
    return AM_OK;
}
return -AM_ENOTSUP;
}

```

5. __tim_timing_prescale_get 函数

该函数用于获取分频值。

程序清单 13.27 __tim_timing_prescale_get 函数示例代码

```

static int __tim_timing_prescale_get(void *p_drv,
                                     uint8_t chan,
                                     uint32_t *p_prescale)
{
    am_zmf2x6_tim_timing_dev_t *p_dev = (am_zmf2x6_tim_timing_dev_t *)p_drv;
    amhw_zmf2x6_tim_t *p_hw_tim;
    if ((p_dev == NULL) || (p_prescale == NULL) || (chan != 0)) {
        return -AM_EINVAL;
    }
    p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    /* 获取分频值 */
    *p_prescale = amhw_zmf2x6_tim_prescale_get(p_hw_tim);
    return AM_OK;
}

```

6. __tim_timing_count_get 函数

该函数用于获取计数值。

程序清单 13.28 __tim_timing_count_get 函数示例代码

```

static int __tim_timing_count_get(void *p_drv, uint8_t chan, void *p_count)
{
    am_zmf2x6_tim_timing_dev_t *p_dev = (am_zmf2x6_tim_timing_dev_t *)p_drv;
    amhw_zmf2x6_tim_t *p_hw_tim;

    if ((p_dev == NULL) || (p_count == NULL) || (chan != 0)) {
        return -AM_EINVAL;
    }

    p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;

    *(uint32_t *)p_count = amhw_zmf2x6_tim_count_get(p_hw_tim);

    return AM_OK;
}

```

7. __tim_timing_rollover_get 函数

该函数用于获取溢出值。

程序清单 13.29 __tim_timing_rollover_get 函数示例代码

```
static int __tim_timing_rollover_get (void *p_drv,uint8_t chan,void *p_rollover)
{
    am_zmf2x6_tim_timing_dev_t *p_dev = (am_zmf2x6_tim_timing_dev_t *)p_drv;
    amhw_zmf2x6_tim_t *p_hw_tim;
    if ((p_dev == NULL) || (p_rollover == NULL) || (chan != 0)) {
        return -AM_EINVAL;
    }
    p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    *(uint32_t *)p_rollover = amhw_zmf2x6_tim_arr_get(p_hw_tim);
    return AM_OK;
}
```

8. __tim_timing_disable 函数

该函数用于定时器禁能。

程序清单 13.30 __tim_timing_disable 函数示例代码

```
static int __tim_timing_disable (void *p_drv, uint8_t chan)
{
    am_zmf2x6_tim_timing_dev_t *p_dev = (am_zmf2x6_tim_timing_dev_t *)p_drv;
    amhw_zmf2x6_tim_t *p_hw_tim;

    if ((p_dev == NULL) || (chan != 0)) {
        return -AM_EINVAL;
    }
    p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    /* 禁能更新中断 */
    amhw_zmf2x6_tim_int_disable(p_hw_tim, AMHW_ZMF2X6_TIM_UIE);
    amhw_zmf2x6_tim_disable(p_hw_tim);
    return AM_OK;
}
```

9. __tim_timing_enable 函数

该函数用于定时器使能。

程序清单 13.31 __tim_timing_enable 函数示例代码

```
static int __tim_timing_enable (void *p_drv, uint8_t chan, void *p_count)
{
    am_zmf2x6_tim_timing_dev_t *p_dev = (am_zmf2x6_tim_timing_dev_t *)p_drv;
    amhw_zmf2x6_tim_t *p_hw_tim;

    if ((p_dev == NULL) || (chan != 0) || (p_count == NULL)) {
        return -AM_EINVAL;
    }
}
```



```

p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;

/* 禁止定时器产生中断 */
amhw_zmf2x6_tim_int_disable(p_hw_tim, AMHW_ZMF2X6_TIM_UIE);

/* 设置自动重装寄存器的值 */
amhw_zmf2x6_tim_arr_set(p_hw_tim, *((uint32_t *)p_count));

/* 计数器清 0 */
amhw_zmf2x6_tim_count_set(p_hw_tim, 0);

/* 产生更新事件, 重新初始化 Prescaler 计数器 及 Repetition 计数器 */
amhw_zmf2x6_tim_egr_set(p_hw_tim, AMHW_ZMF2X6_TIM_UG);

/* 更新定时器时会产生更新事件,清除标志位 */
amhw_zmf2x6_tim_status_flg_clr(p_hw_tim, AMHW_ZMF2X6_TIM_UG);

if (p_dev->pfn_callback != NULL) {

    /* 允许更新中断 */
    amhw_zmf2x6_tim_int_enable(p_hw_tim, AMHW_ZMF2X6_TIM_UIE);
}

/* 使能定时器 TIM 允许计数 */
amhw_zmf2x6_tim_enable(p_hw_tim);

return AM_OK;
}

```

到现在定时器支持的所有函数开发完毕，整个定时器的驱动层开发也就结束了。

13.4 用户配置文件编写

13.4.1 定义设备常量于设备信息常量

创建配置文件后，我们需要在里面加入设备变量，设备变量只需要定义，不用初始化。设备信息常量定义时初始化，默认填充的信息是一种常用的配置方式，定时器设备信息配置如程序清单 13.32 所示。

程序清单 13.32 TIM 设备信息

```

/** \brief TIM1 设备信息 */
const am_zmf2x6_tim_timing_devinfo_t __g_tim1_timing_devinfo = {
    ZMF2X6_TIM1_BASE,                /**< \brief TIM1 寄存器块的基地址 */
    INUM_TIM1_BRK_UP_TRG_COM,        /**< \brief TIM1 中断编号 */
    CLK_TIM1,                        /**< \brief TIM1 时钟 ID */
    AMHW_ZMF2X6_TIM_TYPE0,           /**< \brief 定时器类型 */

```

```

__zmf2x6_plfm_tim1_timing_init,    /**< \brief 平台初始化函数 */
__zmf2x6_plfm_tim1_timing_deinit  /**< \brief 平台解析初始化函数 */
};

```

13.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

程序清单 13.33 平台初始化解初始化

```

/** \brief 定时器 TIM1 Timing 实例初始化，获得 Timer 标准服务句柄 */
am_timer_handle_t am_zmf2x6_tim1_timing_inst_init(void)
{
    return am_zmf2x6_tim_timing_init(&__g_tim1_timing_dev,
                                     &__g_tim1_timing_devinfo);
}

/** \brief TIM1 Timing 实例解初始化 */
void am_zmf2x6_tim1_timing_inst_deinit(am_timer_handle_t handle)
{
    am_zmf2x6_tim_timing_deinit(handle);
}

```

13.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在头文件中加入它们的声明，如所示程序清单 13.34。

程序清单 13.34 平台初始化解初始化声明

```

am_timer_handle_t am_zmf2x6_tim1_timing_inst_init(void);

/**
 * \brief TIM1 TIMING 实例解初始化
 *
 * \param[in] handle 通过 am_zmf2x6_tim1_timing_inst_init() 函数
 *                  获得的 TIMER 标准服务句柄
 *
 * \return 无
 */
void am_zmf2x6_tim1_timing_inst_deinit(am_timer_handle_t handle);

```

用户使用时，应该先调用平台初始化函数 `am_zmf2x6_tim1_timing_inst_init` 函数，然后直接使用 `am_zmf2x6_tim_timing_init(&__g_tim1_timing_dev, &__g_tim1_timing_devinfo)` 初始化 TIM1，并保存其返回的 `handle`，后续直接使用保存的 `handle` 作为标准接口的第一个参数使用标准接口函数即可。

至此，用户配置文件开发完毕。

13.5 例程文件开发

参考 5.5.1 章节新建用户例程文件，搭建文件的基本框架，准备定时器例程内容的编写。

例程文件开发主要分为两个部分，HW 例程和 STD 例程。

13.5.1 例程内容开发

1. HW 层实现示例

这里使用 HW 层来实现定时器定时功能，定时时间到，在定时中断中打印信息提示，并通过 LED 提示代码详见程序清单 13.35。

程序清单 13.35 定时器定时中断示例

```
static void __zmf2x6_tim_hw_timing_irq_handler (void *p_arg)
{
    amhw_zmf2x6_tim_t *p_hw_tim = (amhw_zmf2x6_tim_t *)p_arg;

    AM_DBG_INFO("timing irq!\r\n");

    am_led_toggle(LED0);

    /* 清除更新标志 */
    amhw_zmf2x6_tim_status_flg_clr(p_hw_tim, AMHW_ZMF2X6_TIM_UIF);
}

/**
 * \brief 配置定时器 TIM 为定时功能 timing
 *
 * \param[in] p_hw_tim : 指向定时器寄存器块的指针
 * \param[in] cnt      : 计数值，相对于定时器输入时钟频率来计算
 *
 * \return 无
 */
void tim_timing_chan_config (amhw_zmf2x6_tim_t *p_hw_tim, uint32_t cnt)
{
    uint32_t match, temp;
    uint16_t pre_real = 1, pre_reg = 0;

    {
        temp = cnt;

        /* 16 位计数器最大值 65536, 分频器最大值 65536 */
        if (temp > 4294967295UL) {
            return ;
        }

        /* 当计数小于 65536 时，不分频(值为 1,1 代表为 1 分频) */
        temp = cnt / 65536 + 1;
    }
}
```

```

    /* 16 位定时器需要运算取得合适的分频值 */
    for (pre_real = 1; pre_real < temp; ) {
        pre_reg++;          /* 计算写入寄存器的分频值 0,1,2,... */
        pre_real++;         /* 分频数 */
    }
}

/* 计算自动重载值 */
match = cnt / pre_real;

/* 设置分频值 */
amhw_zmf2x6_tim_prescale_set(p_hw_tim, pre_reg);

/* 设置自动重装寄存器的值 */
amhw_zmf2x6_tim_arr_set(p_hw_tim, match - 1);

/* 计数器清 0 */
amhw_zmf2x6_tim_count_set(p_hw_tim, 0);
}

/**
 * \brief 使能定时器为定时功能
 */
void tim_timing_enable (amhw_zmf2x6_tim_t *p_hw_tim, uint8_t int_num)
{
    /* 产生更新事件，重新初始化 Prescaler 计数器 及 Repetition 计数器 */
    amhw_zmf2x6_tim_egr_set(p_hw_tim, AMHW_ZMF2X6_TIM_UG);

    if (amhw_zmf2x6_tim_status_flg_get(p_hw_tim, AMHW_ZMF2X6_TIM_UG) != 0) {

        /* 更新定时器时会产生更新事件,清除标志位 */
        amhw_zmf2x6_tim_status_flg_clr(p_hw_tim, AMHW_ZMF2X6_TIM_UG);
    }

    /* 连接中断回调函数 */
    am_int_connect(int_num, __zmf2x6_tim_hw_timing_irq_handler, (void *)p_hw_tim);

    /* 使能更新中断 */
    amhw_zmf2x6_tim_int_enable(p_hw_tim, AMHW_ZMF2X6_TIM_UIE);

    am_int_enable(int_num);

    /* 使能定时器 TIM 允许计数 */

```

```

    amhw_zmf2x6_tim_enable(p_hw_tim);
}

/**
 * \brief 定时器 TIM timing 初始化函数
 */
void tim_timing_init (amhw_zmf2x6_tim_t *p_hw_tim, amhw_zmf2x6_tim_type_t type)
{
    if ((AMHW_ZMF2X6_TIM_TYPE0 == type) || (AMHW_ZMF2X6_TIM_TYPE1 == type)) {

        /* 边沿对齐模式 */
        amhw_zmf2x6_tim_cms_set(p_hw_tim, 0);

        /* 向上计数 */
        amhw_zmf2x6_tim_dir_set(p_hw_tim, 0);
    }

    /* 设置时钟分割:TCTS = Tck_tin */
    amhw_zmf2x6_tim_ckd_set(p_hw_tim, 0);

    /* 允许更新事件 */
    amhw_zmf2x6_tim_udis_enable(p_hw_tim);
}

/**
 * \brief 例程入口
 */
void demo_zmf2x6_hw_tim_timing_entry (amhw_zmf2x6_tim_t      *p_hw_tim,
                                       amhw_zmf2x6_tim_type_t type,
                                       uint32_t                 clk_rate,
                                       int32_t                   int_num)
{
    /* 初始化定时器为定时功能 */
    tim_timing_init(p_hw_tim, type);

    tim_timing_chan_config(p_hw_tim, clk_rate / 2);

    tim_timing_enable(p_hw_tim, int_num);

    AM_FOREVER {
        ; /* VOID */
    }
}

```

```
}
```

2. 标准层实现示例

这里使用标准层来实现定时器定时功能，定时时间到，在定时中断中打印信息提示，并通过 LED 提示代码详见程序清单 13.36。

程序清单 13.36 定时器定时中断示例

```
/**
 * \brief 定时器回调函数
 */
static void __tim_timing_callback (void *p_arg)
{
    AM_DBG_INFO("The timing frq is 10Hz\r\n");
    am_led_toggle(LED0);
}

/**
 * \brief 例程入口
 */
void demo_std_timer_timing_entry (am_timer_handle_t handle,
                                   int timing_chan)
{
    const am_timer_info_t *p_info = am_timer_info_get(handle);

    uint32_t clkkin;

    am_timer_clkin_freq_get(handle, &clkkin);

    AM_DBG_INFO("The timer size is : %d-bit \r\n", p_info->counter_width);
    AM_DBG_INFO("The timer clk_frequency: %d Hz\r\n", clkkin);

    /* 设置回调函数 */
    am_timer_callback_set(handle, timing_chan, __tim_timing_callback, NULL);

    /* 设置定时时间为 0.1s （即 1000000 / 10 us） */
    am_timer_enable_us(handle, timing_chan, 1000000 / 10);

    AM_FOREVER {
        ; /* VOID */
    }
}
```

所有例程开发完毕后，整个外设的开发工作也就圆满结束了。

可以提交相关文件，以完成整个开发任务。提交文件时，不用提交工程文件，只需要提交开发的源文件和头文件即可。

广州立功科技-工业技术研发中心

第14章 通用外设驱动开发——CAP

14.1 标准层了解

定时器捕获标准层仅包含一个文件，am_cap.h 文件，在 ametal\interface 找到该文件打开。

14.1.1 了解相关类型定义

打开 am_cap.h 文件，可以看到该文件主要定义了 2 个结构体和 1 个 handle 类型以及捕获方式，如程序清单 14.1 所示，下面对每个结构体做详细介绍。

程序清单 14.1 CAP 标准接口文件结构体定义

```

/** \brief 上升沿触发捕获 */
#define AM_CAP_TRIGGER_RISE          0x01

/** \brief 下降沿触发捕获 */
#define AM_CAP_TRIGGER_FALL          0x02

/** \brief 双边沿触发捕获 */
#define AM_CAP_TRIGGER_BOTH_EDGES    0x04

/** \brief 捕获事件的回调函数类型 */
typedef void (*am_cap_callback_t) (void *p_arg, unsigned int count);

/**
 * \brief 输入捕获驱动函数结构体
 */
struct am_cap_drv_funcs {

    /** \brief 配置一个捕获通道 */
    int (*pfn_cap_config) (void *p_cookie,
                           int chan,
                           unsigned int options,
                           am_cap_callback_t pfn_callback,
                           void *p_arg);

    /** \brief 使能捕获通道 */
    int (*pfn_cap_enable) (void *p_cookie, int chan);

    /** \brief 禁能捕获通道 */
    int (*pfn_cap_disable) (void *p_cookie, int chan);

    /** \brief 复位一个捕获通道，使计数值归零 */
    int (*pfn_cap_reset) (void *p_cookie, int chan);

    /** \brief 转换两次捕获值为时间值（时间单位：ns） */

```



```

int (*pfn_cap_count_to_time)(void          *p_cookie,
                                int          chan,
                                unsigned int  count1,
                                unsigned int  count2,
                                unsigned int *p_time_ns);
};

/**
 * \brief 输入捕获服务
 */
typedef struct am_cap_serv {

    /** \brief 输入捕获驱动函数结构体指针 */
    struct am_cap_drv_funcs *p_funcs;

    /** \brief 用于驱动函数的第一个参数 */
    void *p_drv;
} am_cap_serv_t;

/** \brief 输入捕获标准服务操作句柄类型定义 */
typedef am_cap_serv_t *am_cap_handle_t;

```

1. struct am_cap_drv_funcs 结构体

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层需要实现 5 个函数：

- pfn_cap_config, 用于配置一个捕获通道的函数；
- pfn_cap_enable, 用于使能捕获通道的函数。
- pfn_cap_disable, 用于禁能捕获通道的函数。
- pfn_cap_reset, 用于复位一个捕获通道，使计数值归零的函数。
- pfn_cap_count_to_time, 用于转换两次捕获值为时间值的函数。

因此，在驱动层编写函数时，应该向标准接口层提供这 5 个函数。

这些函数的有一个共同点，就是第一个参数都是 void 型的指针，也就是说，驱动层提供的函数，标准接口层使用时传递的第一个参数并无确定类型，标准接口层不关心第一个参数的具体类型，使用时只负责传递就可以了。

2. typedef struct am_cap_serv 结构体

该结构体定义了标准的 CAP 服务，包含了上文提到的驱动函数结构体的指针和一个 void 型的指针。

这个标准的 CAP 服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

● p_funcs 成员赋值

这是一个指向 struct am_cap_drv_funcs 结构体的指针，我们只需要在驱动层定义一个 struct am_cap_drv_funcs 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那 5

个函数地址。然后，将 `struct am_cap_drv_funcs` 结构体变量的地址赋值给 `p_funcs` 成员即可。

- `p_drv` 成员赋值

这个变量是 `void` 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动自行决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

3. `am_cap_handle_t` 的定义

这个是一个标准的操作句柄，本质上就是 `struct am_cap_serv` 指针类型，驱动层的初始化函数需要返回该类型的一个 `handle` 给用户。这样，用户获取到 `handle`，就可以在需要使用标准接口层函数时，将该 `handle` 传递给标准接口层了。标准接口就可以通过该 `handle` (`struct am_cap_serv` 结构体指针) 访问到驱动层提供的函数了。

14.1.2 了解标准接口函数

在了解 `am_cap.h` 文件相关的结构体、数据类型定义后，接下来就是具体函数的实现，如程序清单 14.2 所示。

程序清单 14.2 CAP 标准接口函数

```
/**
 * \brief 配置一个输入捕获通道
 *
 * \param[in] handle      : 输入捕获标准服务操作句柄
 * \param[in] chan        : 输入捕获通道
 * \param[in] options      : AM_CAP_TRIGGER_*宏值，如：  #AM_CAP_TRIGGER_RISE
 * \param[in] pfn_callback : 回调函数
 * \param[in] p_arg        : 回调函数的用户参数
 *
 * \retval  AM_OK          : 配置 CAP 通道成功
 * \retval -AM_EINVAL      : 配置失败，参数错误
 */
am_static_inline
int am_cap_config (am_cap_handle_t  handle,
                  int                chan,
                  unsigned int       options,
                  am_cap_callback_t pfn_callback,
                  void                *p_arg)
{
    return handle->p_funcs->pfn_cap_config(handle->p_drv,
                                           chan,
                                           options,
                                           pfn_callback,
                                           p_arg);
}

/**
```

```

* \brief 使能捕获通道
*
* \param[in] handle : 输入捕获标准服务操作句柄
* \param[in] chan   : 输入捕获通道
*
* \retval AM_OK      : 使能捕获通道成功
* \retval -AM_EINVAL : 使能捕获通道失败，参数错误
*/
am_static_inline
int am_cap_enable(am_cap_handle_t handle, int chan)
{
    return handle->p_funcs->pfn_cap_enable(handle->p_drv, chan);
}

/**
* \brief 禁能捕获通道
*
* \param[in] handle : 输入捕获标准服务操作句柄
* \param[in] chan   : 输入捕获通道
*
* \retval AM_OK      : 禁能捕获通道成功
* \retval -AM_EINVAL : 禁能捕获通道失败，参数错误
*/
am_static_inline
int am_cap_disable(am_cap_handle_t handle, int chan)
{
    return handle->p_funcs->pfn_cap_disable(handle->p_drv, chan);
}

/**
* \brief 复位捕获通道计数值为 0
*
* \param[in] handle : 输入捕获标准服务操作句柄
* \param[in] chan   : 输入捕获通道
*
* \retval AM_OK      : 复位成功
* \retval -AM_EINVAL : 复位失败，参数错误
*/
am_static_inline
int am_cap_reset(am_cap_handle_t handle, int chan)
{
    return handle->p_funcs->pfn_cap_reset(handle->p_drv, chan);
}

```

```

/**
 * \brief 转换两次捕获值为时间值（时间单位：ns）
 *
 * \param[in] handle    : 输入捕获标准服务操作句柄
 * \param[in] chan      : 输入捕获通道
 * \param[in] count1    : 第一次捕获值
 * \param[in] count2    : 第二次捕获值
 * \param[out] p_time_ns : 用于获取转换结果时间值
 *
 * \retval AM_OK        : 转换成功
 * \retval -AM_EINVAL   : 转换失败，参数错误
 */
am_static_inline
int am_cap_count_to_time(am_cap_handle_t handle,
                        int chan,
                        unsigned int count1,
                        unsigned int count2,
                        unsigned int *p_time_ns)
{
    return handle->p_funcs->pfncap_count_to_time(handle->p_drv,
                                                chan,
                                                count1,
                                                count2,
                                                p_time_ns);
}

```

1. am_cap_config()配置一个输入捕获通道函数

该函数用于配置一个输入捕获通道，传入的第一个参数为实例初始化获取的 **handle**，第二个参数为待设置的通道号，第三个参数为捕获方式选择，第四个参数为回调函数地址，第五个参数为回调函数的用户参数。

2. am_cap_enable()使能捕获通道函数

该函数用于使能捕获通道，传入的第一个参数为实例初始化获取的 **handle**，第二个参数为待设置的通道号。

3. am_cap_disable()禁能捕获通道函数

该函数用于禁能捕获通道，传入的第一个参数为实例初始化获取的 **handle**，第二个参数为待设置的通道号。

4. am_cap_reset()复位捕获通道计数值为 0 函数

该函数用于复位捕获通道计数值为 0，传入的第一个参数为实例初始化获取的 **handle**，第二个参数为待设置的通道号。

5. am_cap_count_to_time()转换两次捕获值为时间值函数

该函数用于转换两次捕获值为时间值，传入的第一个参数为实例初始化获取的 **handle**，第二个参数为待设置的通道号，第三个参数为第一次捕获值，第四个参数为第二次捕获值，

第五个参数为获取转换结果时间值的指针。

到此我们就分析完了 CAP 外设标准层数据类型和接口函数，清楚了驱动层需要为标准接口层提供哪些内容。

14.2 HW 层开发

参考 5.2.1 章节，创建对应目录和文件，添加硬件层基本的内容。

14.2.1 添加寄存器列表

CAP 是定时器中的一项功能，此节内容与 13.2.1 章节一致。

14.2.2 编写寄存器相关函数

1. TIMx_CCRy（捕获/比较寄存器）

用作 CAP 功能时，该寄存器包含了由上一次输入捕获事件 y 传输的计数器值的函数，如程序清单 14.3 所示。

程序清单 14.3 TIMx_CCRy 寄存器函数

```
/**
 * \brief 从一个捕获通道获取捕获值
 *
 * \param[in] p_hw_tim：指向 TIM 定时器寄存器块的指针
 * \param[in] ch      ：捕获通道，有效值 0-3
 *
 * \return 返回指定捕获通道的捕获值
 */
am_static_inline
uint32_t amhw_zmf2x6_tim_ccr_cap_val_get (amhw_zmf2x6_tim_t *p_hw_tim,
                                           uint8_t          ch)
{
    return p_hw_tim->ccr[ch];
}
```

2. TIMx_CCMR（捕获/比较模式寄存器）

该寄存器包括选择输入捕获通道映射、预分频，采样频率及数字滤波器长度的操作函数，如程序清单 14.4 所示。

程序清单 14.4 TIMx_CCMR 寄存器操作函数

```
am_static_inline
void amhw_zmf2x6_tim_ccs_set (amhw_zmf2x6_tim_t *p_hw_tim,
                              uint8_t          ccs_opt,
                              uint8_t          ccs_chan)
{
    if (ccs_chan != AMHW_ZMF2X6_TIM_CC5S) {
        AM_BITS_SET(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01)), 2, ccs_opt);
    }
}
```

```

am_static_inline
void amhw_zmf2x6_tim_icf_set (amhw_zmf2x6_tim_t *p_hw_tim,
                             uint8_t          icf_opt,
                             uint8_t          ccs_chan)
{
    AM_BITS_SET(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01) + 4) , 4, icf_opt);
}

am_static_inline
void amhw_zmf2x6_tim_icpsc_set (amhw_zmf2x6_tim_t *p_hw_tim,
                                uint8_t          icpsc_opt,
                                uint8_t          ccs_chan)
{
    AM_BITS_SET(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01) + 2) , 2, icpsc_opt);
}

```

3. TIMx_CCER（捕获/比较使能寄存器）

该寄存器包括设置输入捕获通道极性和使能，输入捕获互补输出通道极性和使能的操作函数，如程序清单 14.5 所示。

程序清单 14.5 TIMx_CCER 寄存器操作函数

```

/**
 * \brief 输入捕获(外部触发)电平极性
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] type      : 定时器类型
 * \param[in] ccp_opt   : 输入捕获通道电平极性选择
 *
 *      1: 不反相：捕获发生在 ICx 的上升沿；当用作外部触发器时，ICx 不反相；
 *      2: 反相：捕获发生在 ICx 的下降沿；当用作外部触发器时，ICx 反相
 *
 *      其中 x 代表捕获通道，根据定时器的不同而通道数量也有所不同，最多为 4
 * 个通道
 *
 * \param[in] ccs_chan : ccs 通道选择，值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \note : 一旦 LOCK 级别（TIMx_BDTR 寄存器中的 LCKCK 位）设为 3 或 2，则该位不能被修改
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ccp_captrigger_set (amhw_zmf2x6_tim_t *p_hw_tim,
                                           amhw_zmf2x6_tim_type_t type,
                                           uint8_t          ccp_opt,
                                           uint8_t          ccs_chan)
{
    AM_BIT_MODIFY(p_hw_tim->ccer, (4 * ccs_chan + 1), ccp_opt >> 1);
}

```

```

/**
 * \brief 输入捕获通道使能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_cce_cap_enable (amhw_zmf2x6_tim_t *p_hw_tim,
                                     uint8_t          ccs_chan)
{
    AM_BIT_MODIFY(p_hw_tim->ccer, (4 * ccs_chan), 1);
}

/**
 * \brief 输入捕获/比较通道禁能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_cce_cap_disable (amhw_zmf2x6_tim_t *p_hw_tim,
                                       uint8_t          ccs_chan)
{
    AM_BIT_MODIFY(p_hw_tim->ccer, (4 * ccs_chan), 0);
}

/**
 * \brief 输入捕获互补输出极性
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ccnp_opt : 输入捕获互补输出极性选择
 *
 *                      0: OCN 高电平有效;
 *                      1: OCN 低电平有效;
 *
 * \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \note : 一旦 LOCK 级别 (TIMx_BDTR 寄存器中的 LCKCK 位) 设为 3 或 2, 则该位不能被修改
 *
 * \return 无

```

```

*/
am_static_inline
void amhw_zmf2x6_tim_ccne_set (amhw_zmf2x6_tim_t *p_hw_tim,
                               uint8_t          ccnp_opt,
                               uint8_t          ccs_chan)
{
    AM_BIT_MODIFY(p_hw_tim->ccer, (4 * ccs_chan + 3), ccnp_opt);
}

/**
 * \brief 输入捕获互补输出通道使能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ccne_enable (amhw_zmf2x6_tim_t *p_hw_tim,
                                  uint8_t          ccs_chan)
{
    AM_BIT_MODIFY(p_hw_tim->ccer, (4 * ccs_chan + 2), 1);
}

/**
 * \brief 输入捕获互补输出通道禁能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ccne_cap_disable (amhw_zmf2x6_tim_t *p_hw_tim,
                                        uint8_t          ccs_chan)
{
    AM_BIT_MODIFY(p_hw_tim->ccer, (4 * ccs_chan + 2), 0);
}

```

还有诸多寄存器, 可按上述样例逐一添加, CAP 外设寄存器众多, 本节不再一一罗列。

14.2.3 添加其他功能性函数

在每个寄存器的操作函数编写完毕后, 可能 HW 层的接口还有所欠缺, 需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到, 可以暂时不用考虑, 直接进入驱动层的开发。在后续如果发现 HW 层欠缺哪些接口, 可以反过来再增加。当然, 也

有可能 HW 层接口没有什么欠缺了，那样就无需再开发其它函数了。

14.2.4 添加指向寄存器块的指针

CAP 是定时器的其中一项功能，此节内容与 13.2.4 章节一致。

14.3 驱动层开发

参考 5.3.1 章节，新建 am_zmf2x6_tim_cap.h 和 am_zmf2x6_tim_cap.c 驱动文件，搭建相应的文件框架，并按规则放到对应的路径下。

14.3.1 编写结构性程序

驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

定义 CAP 外设设备信息结构体如程序清单 14.6 所示。后续可以根据实际情况修改。

程序清单 14.6 CAP 设备信息结构体定义

```
/**
 * \brief TIM 捕获功能相关的 GPIO 信息
 */
typedef struct am_zmf2x6_tim_cap_ioinfo {
    uint32_t gpio;          /**< \brief 对应的 GPIO 管脚 */
    uint32_t func;          /**< \brief 为捕获功能时的 GPIO 功能设置 */
    uint32_t dfunc;         /**< \brief 禁能管脚捕获功能时的默认 GPIO 功能设置 */
} am_zmf2x6_tim_cap_ioinfo_t;

/**
 * \brief TIM 捕获功能相关的设备信息
 */
typedef struct am_zmf2x6_tim_cap_devinfo {
    uint32_t          tim_regbase; /**< \brief TIM 寄存器块基址 */
    uint8_t           inum;         /**< \brief TIM 对应的中断号 */
    int               clk_num;      /**< \brief 时钟 ID */
    int               clk_div;      /**< \brief 时钟分频系数 */
    uint8_t           channels_num; /**< \brief 使用的通道数，最大为 6 */
    am_zmf2x6_tim_cap_ioinfo_t *p_ioinfo; /**< \brief 指向捕获管脚信息结构体 */
    amhw_zmf2x6_tim_type_t tim_type; /**< \brief 定时器类型 */

    /** \brief 平台初始化函数，如打开时钟，配置引脚等工作 */
}
```

```

void                                (*pfn_plfm_init)(void);

/** \brief 平台解初始化函数 */
void                                (*pfn_plfm_deinit)(void);
} am_zmf2x6_tim_cap_devinfo_t;

```

● 设备结构体定义

由于 CAP 设备需要提供标准的 CAP 服务, 包括 CAP 捕获触发配置、捕获通道使能等。其次, 为了在后续使用中, 可以通过设备访问到设备信息, 也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 14.7 所示, 后续如果需要使用到什么设备相关的全局变量, 就应该定义在这里。

程序清单 14.7 CAP 设备结构体定义

```

/**
 * \brief TIM 捕获功能设备
 */
typedef struct am_zmf2x6_tim_cap_dev {
    am_cap_serv_t      cap_serv;      /**< \brief 标准 CAP 服务 */

    struct {
        am_cap_callback_t callback_func; /**< \brief 回调函数 */
        void                *p_arg;      /**< \brief 回调函数的用户参数 */
    } callback_info[4];                /**< \brief 存储最大 6 通道的回调函数信息 */

    /** \brief 指向 TIM(捕获功能)设备信息常量的指针 */
    const am_zmf2x6_tim_cap_devinfo_t *p_devinfo;
} am_zmf2x6_tim_cap_dev_t;

```

2. 搭建函数框架

函数主要分为两部分, 一部分是提供给用户使用的驱动层函数, 一部分是提供给标准层的函数。

● 驱动函数

对于 CAP 驱动层需要提供初始化函数之外, 还需要实现标准接口层函数。

初始化函数的参数是设备指针和设备信息常量指针, 返回值为用于标准接口的 handle。示例代码如程序清单 14.8 所示。

程序清单 14.8 驱动层函数定义示例

```

/**
 * \brief 初始化 TIM 为捕获功能
 *
 * \param[in] p_dev      : 指向 TIM(捕获功能)设备的指针
 * \param[in] p_devinfo : 指向 TIM(捕获功能)设备信息常量的指针
 */

```

```

* \return CAP 标准服务操作句柄，值为 NULL 时表明初始化失败
*/
am_cap_handle_t am_zmf2x6_tim_cap_init (am_zmf2x6_tim_cap_dev_t      *p_dev,
                                         const am_zmf2x6_tim_cap_devinfo_t *p_devinfo);

/**
* \brief 不使用 TIM 捕获功能时，解初始化 TIM 捕获功能，释放相关资源
*
* \param[in] handle : am_zmf2x6_tim_cap_init() 初始化函数获得的 cap 服务句柄
*
* \return 无
*/
void am_zmf2x6_tim_cap_deinit (am_cap_handle_t handle);

```

● 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 5 个函数，根据标准接口层中的 struct am_cap_drv_funcs 结构体，我们需要在.c 文件中定义 5 个函数。

由于这些函数全部是放在结构体中，通过函数指针的形式提供给上层，并不需要将这些函数声明到.h 文件中供用户直接使用，因此全部以 static 的形式定义成内部函数。定义好函数后，再定义一个 struct am_cap_drv_funcs 结构体，将函数放在其中。示例代码如程序清单 14.9 所示。

程序清单 14.9 提供给标准层函数定义

```

/** \brief 捕获参数配置 */
static int __zmf2x6_tim_cap_config (void      *p_cookie,
                                     int        chan,
                                     unsigned int flags,
                                     am_cap_callback_t pfn_callback,
                                     void        *p_arg);

/** \brief 使能捕获通道 */
static int __zmf2x6_tim_cap_enable (void *p_drv, int chan);

/** \brief 禁能捕获通道 */
static int __zmf2x6_tim_cap_disable (void *p_drv, int chan);

/** \brief 复位捕获通道计数值 */
static int __zmf2x6_tim_cap_reset (void *p_drv, int chan);

static int __zmf2x6_tim_cap_count_to_time (void      *p_drv,
                                           int        chan,
                                           unsigned int count1,
                                           unsigned int count2,
                                           unsigned int *p_time_ns);

```

```
static void __zmf2x6_tim_cap_irq_handler (void *p_arg);

/** \brief 捕获服务驱动函数 */
static const struct am_cap_drv_funcs __g_tim_cap_drv_funcs = {
    __zmf2x6_tim_cap_config,
    __zmf2x6_tim_cap_enable,
    __zmf2x6_tim_cap_disable,
    __zmf2x6_tim_cap_reset,
    __zmf2x6_tim_cap_count_to_time,
};
```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。

14.3.2 驱动函数实现

通过上面的函数框架搭建，共计 7 个函数，5 个功能性函数面向标准层，2 个初始解初始化函数面向用户，下面逐步实现。

1. am_zmf2x6_tim_cap_init 函数

该函数主要完成两个功能：

- a) 初始化设备中的标准服务，将成员赋值。
- b) 初始化硬件相关设置，该函数的示例代码如程序清单 14.10 所示。

程序清单 14.10 am_zmf2x6_tim_cap_init 函数示例代码

```
am_cap_handle_t am_zmf2x6_tim_cap_init (am_zmf2x6_tim_cap_dev_t      *p_dev,
                                         const am_zmf2x6_tim_cap_devinfo_t *p_devinfo)
{
    amhw_zmf2x6_tim_t *p_hw_tim = NULL;
    uint8_t          i;

    if (p_dev == NULL || p_devinfo == NULL) {
        return NULL;
    }

    if (p_devinfo->pfn_plfm_init) {
        p_devinfo->pfn_plfm_init();
    }

    p_dev->p_devinfo      = p_devinfo;
    p_hw_tim              = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    p_dev->cap_serv.p_funcs = (struct am_cap_drv_funcs *)&__g_tim_cap_drv_funcs;
    p_dev->cap_serv.p_drv   = p_dev;

    for (i = 0; i < p_devinfo->channels_num; i++) {
        p_dev->callback_info[i].callback_func = NULL;
    }
}
```

```

/* 捕获初始化 */
__zmf2x6_tim_cap_init(p_hw_tim, p_devinfo, p_devinfo->tim_type);

return &(p_dev->cap_serv);
}

```

2. am_zmf2x6_tim_cap_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放。

因此，该函数仅简单的将几个不再使用的变量置为 NULL 即可。示例代码如程序清单 14.11 所示。

程序清单 14.11 am_zmf2x6_tim_cap_deinit 函数示例代码

```

void am_zmf2x6_tim_cap_deinit (am_cap_handle_t handle)
{
    uint8_t i = 0;

    am_zmf2x6_tim_cap_dev_t *p_dev      = (am_zmf2x6_tim_cap_dev_t *)handle;
    amhw_zmf2x6_tim_t          *p_hw_tim = NULL;

    am_zmf2x6_tim_cap_ioinfo_t *p_ioinfo = p_dev->p_devinfo->p_ioinfo;

    if (p_dev == NULL || p_dev->p_devinfo == NULL ) {
        return ;
    }

    p_hw_tim      = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;

    /* 清零计数器 */
    amhw_zmf2x6_tim_count_set(p_hw_tim, 0);

    /* 关闭 TIM 模块 */
    amhw_zmf2x6_tim_disable(p_hw_tim);

    am_int_disable(p_dev->p_devinfo->inum);

    p_dev->cap_serv.p_drv = NULL;

    /* 还原 GPIO 配置 */
    for (i = 0; i < p_dev->p_devinfo->channels_num; i++) {

        am_gpio_pin_cfg(p_ioinfo[i].gpio, p_ioinfo[i].dfunc);
    }

    if (p_dev->p_devinfo->pfn_plfm_deinit) {

```

```

        p_dev->p_devinfo->pfn_plfm_deinit();
    }
}

```

3. __zmf2x6_tim_cap_config

该函数主要用于配置 CAP 触发方式及回调函数,功能实现代码如程序清单 14.12 所示。

程序清单 14.12 __zmf2x6_tim_cap_config 函数示例代码

```

/** \brief 配置一个输入捕获通道 */
static int __zmf2x6_tim_cap_config (void          *p_drv,
                                     int            chan,
                                     unsigned int    options,
                                     am_cap_callback_t pfn_callback,
                                     void            *p_arg)
{
    am_zmf2x6_tim_cap_dev_t *p_dev    = (am_zmf2x6_tim_cap_dev_t *)p_drv;
    amhw_zmf2x6_tim_t       *p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;

    /* 有效通道号范围 0 ~ (channels_num - 1) */
    if (chan >= p_dev->p_devinfo->channels_num) {
        return -AM_EINVAL;
    }

    /* 不支持双边沿触发 */
    if ( (options & AM_CAP_TRIGGER_BOTH_EDGES) == AM_CAP_TRIGGER_BOTH_EDGES ) {
        return -AM_ENOTSUP;
    }

    /* 配置输入不分频, 1 次边沿就触发 1 次捕获 */
    amhw_zmf2x6_tim_icpsc_set(p_hw_tim, 0, chan);

    /* 选择输入端通道映射不反相 */
    amhw_zmf2x6_tim_ccs_set(p_hw_tim, 1, chan);

    /* 设置输入滤波器的分频值为 0(默认不使用滤波) */
    amhw_zmf2x6_tim_icf_set(p_hw_tim, AMHW_ZMF2X6_TIM_ICF_FSAMPLING0, chan);

    /* 选择上升沿触发 */
    if ((options & AM_CAP_TRIGGER_RISE) == AM_CAP_TRIGGER_RISE) {
        amhw_zmf2x6_tim_ccp_captrigger_set(p_hw_tim,
                                           p_dev->p_devinfo->tim_type,
                                           AM_CAP_TRIGGER_RISE,
                                           chan);
    }
}

```

```

/* 选择下降沿触发 */
if ((options & AM_CAP_TRIGGER_FALL) == AM_CAP_TRIGGER_FALL) {
    amhw_zmf2x6_tim_ccp_captrigger_set(p_hw_tim,
                                       p_dev->p_devinfo->tim_type,
                                       AM_CAP_TRIGGER_FALL,
                                       chan);
}

p_dev->callback_info[chan].callback_func = pfn_callback;
p_dev->callback_info[chan].p_arg      = p_arg;

return AM_OK;
}

```

4. __zmf2x6_tim_cap_enable

该函数主要用于使能捕获通道，功能实现代码如程序清单 14.13 所示。

程序清单 14.13 __zmf2x6_tim_cap_enable 函数示例代码

```

/** \brief 捕获使能 */
static int __zmf2x6_tim_cap_enable (void *p_drv, int chan)
{
    am_zmf2x6_tim_cap_dev_t    *p_dev    = (am_zmf2x6_tim_cap_dev_t *)p_drv;
    amhw_zmf2x6_tim_t          *p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    am_zmf2x6_tim_cap_ioinfo_t *p_ioinfo = p_dev->p_devinfo->p_ioinfo;

    /* 有效通道号范围 0 ~ (channels_num - 1) */
    if (chan >= p_dev->p_devinfo->channels_num) {
        return -AM_EINVAL;
    }

    /* 引脚配置 */
    am_gpio_pin_cfg(p_ioinfo[chan].gpio, p_ioinfo[chan].func);

    /* 中断连接并使能 */
    am_int_connect(p_dev->p_devinfo->inum, __zmf2x6_tim_cap_irq_handler, (void *)p_dev);
    am_int_enable(p_dev->p_devinfo->inum);

    /* 设置自动重装寄存器的值 */
    amhw_zmf2x6_tim_arr_set(p_hw_tim, 0xffffffff);

    /* 清零计数器 */
    amhw_zmf2x6_tim_count_set(p_hw_tim, 0);

    /* 产生更新事件，重新初始化 Prescaler 计数器 及 Repetition 计数器 */
    amhw_zmf2x6_tim_egr_set(p_hw_tim, AMHW_ZMF2X6_TIM_UG);
}

```

```

if (amhw_zmf2x6_tim_status_flg_get(p_hw_tim, AMHW_ZMF2X6_TIM_UG) != 0) {

    /* 更新定时器时会产生更新事件,清除标志位 */
    amhw_zmf2x6_tim_status_flg_clr(p_hw_tim, AMHW_ZMF2X6_TIM_UG);
}

/* 允许捕获计数器的值到捕获寄存器中 */
amhw_zmf2x6_tim_cce_cap_enable(p_hw_tim, chan);

/* 允许捕获中断 */
amhw_zmf2x6_tim_int_enable(p_hw_tim, (1UL << (chan + 1)));

/* 使能定时器 TIM 允许计数 */
amhw_zmf2x6_tim_enable(p_hw_tim);

return AM_OK;
}

```

5. __zmf2x6_tim_cap_disable

该函数主要用于禁用捕获通道，功能实现代码如程序清单 14.14 所示。

程序清单 14.14 __zmf2x6_tim_cap_disable 函数示例代码

```

/** \brief 捕获禁用 */
static int __zmf2x6_tim_cap_disable (void *p_drv, int chan)
{
    am_zmf2x6_tim_cap_dev_t    *p_dev    = (am_zmf2x6_tim_cap_dev_t *)p_drv;
    amhw_zmf2x6_tim_t          *p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    am_zmf2x6_tim_cap_ioinfo_t *p_ioinfo = p_dev->p_devinfo->p_ioinfo;

    /* 有效通道号范围 0 ~ (channels_num - 1) */
    if (chan >= p_dev->p_devinfo->channels_num) {
        return -AM_EINVAL;
    }

    /* 还原 GPIO 配置 */
    am_gpio_pin_cfg(p_ioinfo[chan].gpio, p_ioinfo[chan].dfunc);

    /* 断开 NVIC 中断回调函数 */
    am_int_disconnect(p_dev->p_devinfo->inum, __zmf2x6_tim_cap_irq_handler, (void *)p_dev);
    am_int_disable(p_dev->p_devinfo->inum);

    /* 禁用捕获中断 */
    amhw_zmf2x6_tim_int_disable(p_hw_tim, (1UL << (chan + 1)));
}

```



```

/* 禁能定时器 TIM 允许计数 */
amhw_zmf2x6_tim_disable(p_hw_tim);

return AM_OK;
}

```

6. __zmf2x6_tim_cap_reset

该函数主要用于复位捕获通道计数值，功能实现代码如程序清单 14.15 所示。

程序清单 14.15 __zmf2x6_tim_cap_reset 函数示例代码

```

/**
 * \brief 复位捕获通道计数值
 */
static int __zmf2x6_tim_cap_reset (void *p_drv, int chan)
{
    am_zmf2x6_tim_cap_dev_t *p_dev = (am_zmf2x6_tim_cap_dev_t *)p_drv;
    amhw_zmf2x6_tim_t *p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;

    /* 禁能定时器 TIM 允许计数 */
    amhw_zmf2x6_tim_disable(p_hw_tim);

    /* 清零计数器 */
    amhw_zmf2x6_tim_count_set(p_hw_tim, 0);

    /* 产生更新事件，重新初始化 Prescaler 计数器 及 Repetition 计数器 */
    amhw_zmf2x6_tim_egr_set(p_hw_tim, AMHW_ZMF2X6_TIM_UG);

    if (amhw_zmf2x6_tim_status_flg_get(p_hw_tim, AMHW_ZMF2X6_TIM_UG) != 0) {

        /* 更新定时器时会产生更新事件,清除标志位 */
        amhw_zmf2x6_tim_status_flg_clr(p_hw_tim, AMHW_ZMF2X6_TIM_UG);
    }

    return AM_OK;
}

```

7. __zmf2x6_tim_cap_count_to_time

该函数主要用于转换两次捕获值为时间值，功能实现代码如程序清单 14.16 所示。

程序清单 14.16 __zmf2x6_tim_cap_count_to_time 函数示例代码

```

/**
 * \brief 转换两次捕获值为时间值
 */
static int __zmf2x6_tim_cap_count_to_time (void *p_drv,
                                             int chan,
                                             unsigned int count1,

```

```

                                unsigned int  count2,
                                unsigned int *p_time_ns)
{
    am_zmf2x6_tim_cap_dev_t *p_dev    = (am_zmf2x6_tim_cap_dev_t *)p_drv;
    amhw_zmf2x6_tim_t        *p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;

    uint32_t count_err;
    uint64_t time_ns;
    uint16_t reg_pre, pre;

    uint32_t clkfreq  = am_clk_rate_get(p_dev->p_devinfo->clk_num);

    reg_pre = (uint16_t)amhw_zmf2x6_tim_prescale_get(p_hw_tim);

    pre = reg_pre + 1;

    count_err = count2 - count1;

    /* 将两次读取值的差转换成时间 */
    time_ns = (uint64_t)1000000000 * (uint64_t)count_err * pre / (uint64_t)clkfreq;

    *p_time_ns = time_ns;

    return AM_OK;
}

```

14.4 用户配置文件编写

参考 5.4.1 章节新建用户配置文件，搭建文件的基本框架，准备配置文件内容的编写。

14.4.1 定义设备常量于设备信息常量

设备信息常量初始化，默认填充的信息是一种常用的配置方式，使用 TIM1 的 CAP 捕获功能为例，详细的配置如程序清单 14.17 所示。

程序清单 14.17 CAP 设备信息初始化配置

```

/** \brief TIM1 用于捕获功能的引脚配置信息列表 */
am_zmf2x6_tim_cap_ioinfo_t __g_tim1_cap_ioinfo_list[] = {

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

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

```

```

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

/** \brief 通道 4 */
{PIOA_11, PIOA_11_TIM1_CH4 | PIOA_11_INPUT_FLOAT, PIOA_11_GPIO |
PIOA_11_INPUT_FLOAT}
};

/** \brief TIM1 用于捕获功能的设备信息 */
const am_zmf2x6_tim_cap_devinfo_t __g_tim1_cap_devinfo = {
    ZMF2X6_TIM1_BASE,          /**< \brief TIM1 寄存器块的基地址 */
    INUM_TIM1_CC,              /**< \brief TIM1 中断编号 */
    CLK_TIM1,                  /**< \brief TIM1 时钟 ID */
    0x00,                      /**< \brief 设置的预分频寄存器值（最终的分频为 value+1） */
    4,                         /**< \brief 4 个捕获通道 */
    &__g_tim1_cap_ioinfo_list[0], /**< \brief 引脚配置信息列表 */
    AMHW_ZMF2X6_TIM_TYPE0,     /**< \brief 定时器类型 */
    __zmf2x6_plfm_tim1_cap_init, /**< \brief 平台初始化函数 */
    __zmf2x6_plfm_tim1_cap_deinit /**< \brief 平台解初始化函数 */
};

```

14.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

平台初始化函数和解初始化函数定义如程序清单 14.18 所示。

程序清单 14.18 平台初始化函数和解初始化函数定义

```

/** \brief TIM1 用于捕获功能的平台初始化 */
void __zmf2x6_plfm_tim1_cap_init(void)
{
    /* 使能高级定时器 1 时钟 */
    am_clk_enable(CLK_TIM1);

    /* 复位高级定时器 1 */
    am_zmf2x6_clk_reset(CLK_TIM1);
}

/** \brief 解除 TIM1 平台初始化 */
void __zmf2x6_plfm_tim1_cap_deinit(void)
{
    /* 复位高级定时器 1 */
}

```

```

am_zmf2x6_clk_reset(CLK_TIM1);

/* 禁用高级定时器 1 时钟 */
am_clk_disable(CLK_TIM1);
}

```

14.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在标准层实例公共头文件 `am_zmf2x6_inst_init.h` 中加入它们的声明，如程序清单 14.19 所示。

程序清单 14.19 初始化解初始化声明

```

/**
 * \brief TIM1 CAP 实例初始化，获得 CAP 标准服务句柄
 *
 * \param 无
 *
 * \return CAP 标准服务句柄，若为 NULL，表明初始化失败
 */
am_cap_handle_t am_zmf2x6_tim1_cap_inst_init(void);

/**
 * \brief TIM1 CAP 实例解初始化
 *
 * \param[in] handle 通过 am_zmf2x6_tim1_cap_inst_init() 函数
 *                  获得的 CAP 标准服务句柄
 *
 * \return 无
 */
void am_zmf2x6_tim1_cap_inst_deinit(am_cap_handle_t handle);

```

用户使用时，应该先调用实例初始化函数 `am_zmf2x6_tim1_cap_inst_init` 函数，初始化 TIM1 的 CAP 功能，并返回 CAP 标准服务句柄。

至此，用户配置文件开发完毕。

14.5 例程文件开发

例程文件开发主要分为两个部分，HW 例程和 STD 例程。

调试例程时，应该在 `template` 工程下进行，不需要新建其它工程，只需要将新建的文件加入到工程结构的 `user_code` 下。例程文件会与原先的 `main.c` 文件冲突（因为都存在 `main` 函数），这时，只需要调试时将原先的 `main.c` 不要包含在工程编译中，例程文件开发完毕后，再恢复将 `main.c` 包含在工程中，同时将例程文件移除工程即可。

14.5.1 例程内容开发

1. HW 层实现示例

这里使用 HW 层来实现定时器捕获，相关代码如程序清单 14.20 所示。

程序清单 14.20 定时器捕获

```

#include "ametal.h"
#include "am_clk.h"
#include "am_pwm.h"
#include "am_gpio.h"
#include "am_cap.h"
#include "am_int.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_zmf2x6.h"
#include "am_zmf2x6_clk.h"
#include "hw/amhw_zmf2x6_tim.h"
#include "am_zmf2x6_inst_init.h"
#include "demo_zmf2x6_entries.h"
#include "demo_zmf2x6_core_entries.h"

static volatile am_bool_t g_flag      = AM_FALSE; /**< \brief 捕获标志 */
static volatile uint32_t  g_time_ns   = 0;        /**< \brief 捕获计数值 */
static uint32_t          __g_clk_rate = 0;        /**< \brief 时钟频率 */

static void __zmf2x6_tim_hw_cap_irq_handler(void *p_arg)
{
    amhw_zmf2x6_tim_t *p_hw_tim = (amhw_zmf2x6_tim_t *)p_arg;
    uint8_t           chans_num = 4;

    uint8_t           i          = 1;
    uint32_t           value      = 0;

    uint32_t           count_err = 0;
    uint64_t           time_ns    = 0;
    uint16_t           reg_pre    = 0, pre = 1;

    static uint32_t     count16    = 0;

    static am_bool_t    first      = AM_TRUE;

    for (i = 1; i <= chans_num; i++) {
        if ((amhw_zmf2x6_tim_status_flg_get(p_hw_tim, 1UL << i) != 0) {

            /* 得到对应通道的值 */
            value = amhw_zmf2x6_tim_ccr_cap_val_get(p_hw_tim, i - 1);
            if(0 == i - 1) {
                if (g_flag == AM_FALSE) {

```

```

        if (first == AM_TRUE) {

            count16 = value;

            first = AM_FALSE;

        } else {

            /* 定时器 TIM 不是 32 位计数器时, 避免溢出时数据错误 */
            if(count16 < value) {

                reg_pre = (uint16_t)amhw_zmf2x6_tim_prescale_get(p_hw_tim);

                pre = reg_pre + 1;

                count_err = value - count16;

                /* 将两次读取值的差转换成时间 */
                time_ns = (uint64_t)1000000000 *
                           (uint64_t)count_err *
                           pre /
                           (uint64_t)__g_clk_rate;

                g_time_ns = time_ns;
            }

            first = AM_TRUE;

            /* 置标志为真, 表明捕获完成 */
            g_flag = AM_TRUE;
        }
    }

    /* 清除通道 i 标志 */
    amhw_zmf2x6_tim_status_flg_clr(p_hw_tim, (1UL << i));
}

}

/**
 * \brief 配置定时器为输入捕获
 */
static void tim_cap_chan_config (amhw_zmf2x6_tim_t      *p_hw_tim,
                                amhw_zmf2x6_tim_type_t type,

```

```

                                uint32_t      chan,
                                uint32_t      cap_edge)
{

    /* 配置输入不分频, 1 次边沿就触发 1 次捕获 */
    amhw_zmf2x6_tim_icpsc_set(p_hw_tim, 0, chan);

    /* 选择输入端通道映射不反相 */
    amhw_zmf2x6_tim_ccs_set(p_hw_tim, 1, chan);

    /* 设置输入滤波器的分频值为 0(默认不使用滤波) */
    amhw_zmf2x6_tim_icf_set(p_hw_tim, AMHW_ZMF2X6_TIM_ICF_FSAMPLING0, chan);

    /* 选择上升沿触发 */
    if ((cap_edge & AM_CAP_TRIGGER_RISE) == AM_CAP_TRIGGER_RISE) {
        amhw_zmf2x6_tim_ccp_captrigger_set(p_hw_tim,
                                           type,
                                           AM_CAP_TRIGGER_RISE,
                                           chan);
    }

    /* 选择下降沿触发 */
    if ((cap_edge & AM_CAP_TRIGGER_FALL) == AM_CAP_TRIGGER_FALL) {
        amhw_zmf2x6_tim_ccp_captrigger_set(p_hw_tim,
                                           type,
                                           AM_CAP_TRIGGER_FALL,
                                           chan);
    }
}

static void tim_cap_enable (amhw_zmf2x6_tim_t *p_hw_tim,
                           uint32_t      chan,
                           uint8_t      int_num)
{

    /* 中断连接并使能 */
    am_int_connect(int_num, __zmf2x6_tim_hw_cap_irq_handler, (void *)p_hw_tim);
    am_int_enable(int_num);

    /* 设置自动重装寄存器的值 */
    amhw_zmf2x6_tim_arr_set(p_hw_tim, 0xffffffff);

    /* 清零计数器 */
    amhw_zmf2x6_tim_count_set(p_hw_tim, 0);

```

```

/* 产生更新事件，重新初始化 Prescaler 计数器 及 Repetition 计数器 */
amhw_zmf2x6_tim_egr_set(p_hw_tim, AMHW_ZMF2X6_TIM_UG);

if (amhw_zmf2x6_tim_status_flg_get(p_hw_tim, AMHW_ZMF2X6_TIM_UG) != 0) {

    /* 更新定时器时会产生更新事件,清除标志位 */
    amhw_zmf2x6_tim_status_flg_clr(p_hw_tim, AMHW_ZMF2X6_TIM_UG);
}

/* 允许捕获计数器的值到捕获寄存器中 */
amhw_zmf2x6_tim_cce_cap_enable(p_hw_tim, chan);

/* 允许捕获中断 */
amhw_zmf2x6_tim_int_enable(p_hw_tim, (1UL << (chan + 1)));

/* 使能定时器 TIM 允许计数 */
amhw_zmf2x6_tim_enable(p_hw_tim);
}

/**
 * \brief 定时器输入捕获初始化函数
 */
static void tim_cap_init (amhw_zmf2x6_tim_t *p_hw_tim, amhw_zmf2x6_tim_type_t type)
{
    if ((AMHW_ZMF2X6_TIM_TYPE0 == type) || (AMHW_ZMF2X6_TIM_TYPE1 == type)) {

        /* 边沿对齐模式 */
        amhw_zmf2x6_tim_cms_set(p_hw_tim, 0);

        /* 向上计数 */
        amhw_zmf2x6_tim_dir_set(p_hw_tim, 0);
    }

    /* 设置时钟分割(用于确定输入滤波器的输入时钟频率,默认 Fdts = Fck_in */
    amhw_zmf2x6_tim_ckd_set(p_hw_tim, 0);

    /* 清零计数器 */
    amhw_zmf2x6_tim_count_set(p_hw_tim, 0);

    /* 设置分频器 */
    amhw_zmf2x6_tim_prescale_set(p_hw_tim, 0x00);

    /* 允许更新事件 */

```



```

    amhw_zmf2x6_tim_udis_enable(p_hw_tim);
}

void demo_zmf2x6_hw_tim_cap_entry (amhw_zmf2x6_tim_t      *p_hw_tim,
                                   amhw_zmf2x6_tim_type_t   type,
                                   uint32_t                  chan,
                                   uint32_t                  clk_rate,
                                   int32_t                    inum)
{
    uint32_t freq = 0; /* 捕获到的频率 */

    __g_clk_rate = clk_rate;

    /* 初始化定时器为捕获功能 */
    tim_cap_init(p_hw_tim, type);

    /* 配置定时器捕获通道 (不支持双边沿触发捕获) */
    tim_cap_chan_config(p_hw_tim,
                        type,
                        chan,
                        AM_CAP_TRIGGER_RISE);

    tim_cap_enable(p_hw_tim, chan, inum);

    while (1) {

        if (g_flag == AM_TRUE) {

            freq = 1000000000 / g_time_ns;
            AM_DBG_INFO("The period is %d ns, The freq is %d Hz \r\n",
                        g_time_ns,
                        freq);
            g_flag = AM_FALSE;
        }
    }
}

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_hw_tim_cap_entry (void)
{
    am_pwm_handle_t pwm_handle = am_zmf2x6_tim2_pwm_inst_init();

```

```

AM_DBG_INFO("demo zmf2x6_core hw tim cap!\r\n");

/* TIM2 输出频率为 2KHz 的 PWM */
am_pwm_config(pwm_handle, 0, 500000 / 2, 500000);
am_pwm_enable(pwm_handle, 0);

/* 初始化引脚 */
am_gpio_pin_cfg(PIOA_6, PIOA_6_TIM3_CH1 | PIOA_6_INPUT_FLOAT);

/* 使能定时器时钟 */
am_clk_enable(CLK_TIM3);

/* 复位定时器 */
am_zmf2x6_clk_reset(CLK_TIM3);

demo_zmf2x6_hw_tim_cap_entry(ZMF2X6_TIM3,
                             AMHW_ZMF2X6_TIM_TYPE1,
                             0,
                             am_clk_rate_get(CLK_TIM3),
                             INUM_TIM3);
}

```

2. 标准层实现示例

这里使用标准层来实现定时器捕获，通过标准接口实现，相关代码如程序清单 14.21 所示。

程序清单 14.21 定时器捕获

```

#include "ametal.h"
#include "am_pwm.h"
#include "am_cap.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_zmf2x6_inst_init.h"
#include "demo_std_entries.h"
#include "demo_zmf2x6_core_entries.h"

static volatile am_bool_t __g_flag = AM_FALSE; /**< \brief 捕获标志 */
static volatile uint32_t __g_time_ns = 0; /**< \brief 捕获计数值 */

/**
 * \brief 捕获回调函数
 */
static void __cap_callback(void *p_arg, unsigned int cap_val)
{
    static uint32_t cap_val_prev = 0xffffffff;

```

```

    if (cap_val > cap_val_prev) {
        am_cap_count_to_time((am_cap_handle_t)p_arg, 0, cap_val_prev, cap_val, (unsigned int
*)&__g_time_ns);
        __g_flag = AM_TRUE;
    }
    cap_val_prev = cap_val;
}

/**
 * \brief 单次读取捕获频率
 */
static int read_freq(void)
{
    int32_t freq = -1; // init as invalid value
    uint32_t time_ns = __g_time_ns;

    if (time_ns > 0) {
        freq = 1000000000 / time_ns;
    }

    return freq;
}

void demo_std_timer_cap_entry (am_cap_handle_t cap_handle, int cap_chan)
{
    int32_t freq = 0;

    /* 捕获输入配置 */
    am_cap_config(cap_handle,
                  cap_chan,
                  AM_CAP_TRIGGER_RISE,
                  __cap_callback,
                  (void *)cap_handle);
    am_cap_enable(cap_handle, cap_chan);

    while (1) {
        if (__g_flag == AM_TRUE) {
            freq = read_freq();
            AM_DBG_INFO("The period is %d ns, The freq is %d Hz \r\n",
                        __g_time_ns,
                        freq);
            __g_flag = AM_FALSE;
        }
    }
}

```

```

    }
}

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_std_tim_cap_entry (void)
{
    am_pwm_handle_t pwm_handle = am_zmf2x6_tim3_pwm_inst_init();

    AM_DBG_INFO("demo zmf2x6_core std tim cap!\r\n");

    /* TIM2 输出频率为 2KHz 的 PWM */
    am_pwm_config(pwm_handle, 1, 500000 / 2, 500000);
    am_pwm_enable(pwm_handle, 1);

    demo_std_timer_cap_entry(am_zmf2x6_tim1_cap_inst_init(), 0);
    // demo_std_timer_cap_entry(am_zmf2x6_tim2_cap_inst_init(), 0);
    // demo_std_timer_cap_entry(am_zmf2x6_tim14_cap_inst_init(), 0);
    // demo_std_timer_cap_entry(am_zmf2x6_tim16_cap_inst_init(), 0);
    // demo_std_timer_cap_entry(am_zmf2x6_tim17_cap_inst_init(), 0);
}

```

所有例程开发完毕后，整个外设的开发工作也就圆满结束了。

可以提交相关文件，以完成整个开发任务。提交文件时，不用提交工程文件，只需要提交开发的源文件和头文件即可。

第15章 通用外设驱动开发——PWM

15.1 标准层了解

PWM 标准层仅包含一个文件，am_pwm.h 文件，在 ametal\interface 找到该文件打开。

15.1.1 了解相关类型定义

打开 am_pwm.h 文件，可以看到该文件主要定义了 2 个结构体和 1 个 handle 类型，如程序清单 15.1 所示，下面对每个结构体做详细介绍。

程序清单 15.1 PWM 标准接口文件结构体定义

```
/**
 * \brief PWM 驱动函数结构体
 */
struct am_pwm_drv_funcs {

    /** \brief 配置一个 PWM 通道 */
    int (*pfn_pwm_config) (void          *p_drv,
                           int            chan,
                           unsigned long  duty_ns,
                           unsigned long  period_ns);

    /** \brief 使能 PWM 输出 */
    int (*pfn_pwm_enable) (void *p_drv, int chan);

    /** \brief 禁能 PWM 输出 */
    int (*pfn_pwm_disable) (void *p_drv,int chan);
};

/**
 * \brief PWM 服务
 */
typedef struct am_pwm_serv {

    /** \brief PWM 驱动函数结构体指针 */
    struct am_pwm_drv_funcs  *p_funcs;

    /** \brief 用于驱动函数的第一个参数 */
    void                      *p_drv;
} am_pwm_serv_t;

/** \brief PWM 标准服务操作句柄类型定义 */
typedef am_pwm_serv_t *am_pwm_handle_t;
```

1. struct am_pwm_drv_funcs 结构体

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层要实现 3 个函数：

- `pfn_pwm_config`，用于配置一个 PWM 通道的函数；
- `pfn_pwm_enable`，用于使能 PWM 输出的函数。
- `pfn_pwm_disable`，用于禁能 PWM 输出的函数。

因此，在驱动层编写函数时，应该向标准接口层提供这 3 个函数。

这些函数的有一个共同点，就是第一个参数都是 `void` 型的指针，也就是说，驱动层提供的函数，标准接口层使用时传递的第一个参数并无确定类型，标准接口层不关心第一个参数的具体类型，使用时只负责传递就可以了。

2. `typedef struct am_pwm_serv` 结构体

该结构体定义了标准的 PWM 服务，包含了上文提到的驱动函数结构体的指针和一个 `void` 型的指针。

这个标准的 PWM 服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

- `p_funcs` 成员赋值

这是一个指向 `struct am_pwm_drv_funcs` 结构体的指针，我们只需要在驱动层定义一个 `struct am_pwm_drv_funcs` 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那 3 个函数地址。然后，将 `struct am_pwm_drv_funcs` 结构体变量的地址赋值给 `p_funcs` 成员即可。

- `p_drv` 成员赋值

这个变量是 `void` 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动自行决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

3. `am_pwm_handle_t` 的定义

这个是一个标准的操作句柄，本质上就是 `struct am_pwm_serv` 指针类型，驱动层的初始化函数需要返回该类型的一个 `handle` 给用户。这样，用户获取到 `handle`，就可以在需要使用标准接口层函数时，将该 `handle` 传递给标准接口层了。标准接口就可以通过该 `handle` (`struct am_pwm_serv` 结构体指针) 访问到驱动层提供的函数了。

15.1.2 了解标准接口函数

在了解 `am_pwm.h` 文件相关的结构体、数据类型定义后，接下来就是具体函数的实现，如程序清单 15.2 所示。

程序清单 15.2 PWM 标准接口函数

```
/**
 * \brief 配置一个 PWM 通道
 *
 * \param[in] handle    : PWM 标准服务操作句柄
 * \param[in] chan      : PWM 通道
 * \param[in] duty_ns   : PWM 脉宽时间（单位：纳秒）
 * \param[in] period_ns : PWM 周期时间（单位：纳秒）
 */
```

```

* \retval  AM_OK      : 配置 PWM 通道成功
* \retval -AM_EINVAL  : 配置失败, 参数错误
*/
am_static_inline
int am_pwm_config (am_pwm_handle_t handle,
                  int chan,
                  unsigned long duty_ns,
                  unsigned long period_ns)
{
    return handle->p_funcs->pfn_pwm_config(handle->p_drv,
                                          chan,
                                          duty_ns,
                                          period_ns);
}

/**
 * \brief 使能 PWM 输出
 *
 * \param[in] handle : PWM 标准服务操作句柄
 * \param[in] chan   : PWM 通道
 *
 * \retval  AM_OK      : 使能 PWM 输出成功
 * \retval -AM_EINVAL : 失败, 参数错误
 */
am_static_inline
int am_pwm_enable (am_pwm_handle_t handle, int chan)
{
    return handle->p_funcs->pfn_pwm_enable(handle->p_drv, chan);
}

/**
 * \brief 禁能 PWM 输出
 *
 * \param[in] handle : PWM 标准服务操作句柄
 * \param[in] chan   : PWM 通道
 *
 * \retval  AM_OK      : 禁能 PWM 输出成功
 * \retval -AM_EINVAL : 失败, 参数错误
 */
am_static_inline
int am_pwm_disable (am_pwm_handle_t handle, int chan)
{
    return handle->p_funcs->pfn_pwm_disable(handle->p_drv, chan);
}

```

1. am_pwm_config()配置一个 PWM 通道函数

该函数用于配置一个 PWM 通道，传入的第一个参数为实例初始化获取的 handle，第二个参数为待设置的通道号，第三个参数为脉宽时间，第四个参数为周期时间。

2. am_pwm_enable()使能 PWM 输出函数

该函数用于使能 PWM 输出，传入的第一个参数为实例初始化获取的 handle，第二个参数为待设置的通道号。

3. am_pwm_disable()禁能 PWM 输出函数

该函数用于禁能 PWM 输出，传入的第一个参数为实例初始化获取的 handle，第二个参数为待设置的通道号。

到此我们就分析完了 PWM 外设标准层数据类型和接口函数，清楚了驱动层需要为标准接口层提供哪些内容。

15.2 HW 层开发

参考 5.2.1 章节，创建对应目录和文件，添加硬件层基本的内容。

15.2.1 添加寄存器列表

PWM 是定时器中的一项功能，此节内容与 13.2.1 章节一致。

15.2.2 编写寄存器相关函数

1. TIMx_CCRy（捕获/比较寄存器）

用作 PWM 输出功能时，该寄存器包含装入当前比较 y 寄存器的值的函数，如程序清单 15.3 所示。

程序清单 15.3 TIMx_CCRy 寄存器函数

```
/**
 * \brief 获取设置比较输出通道的预装载值
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ch       : 匹配通道编号 有效值 0-4 (4 仅在高级定时器 TIM1 有效)
 *
 * \note: 通用定时器 TIM2 该寄存器为 32 位，其它为 16 位
 *
 * \return 比较输出通道的预装载的值
 */
am_static_inline
uint32_t amhw_zmf2x6_tim_output_reload_val_get (amhw_zmf2x6_tim_t *p_hw_tim,
                                                  uint8_t ch)
{
    if (ch != AMHW_ZMF2X6_TIM_CC5S) {
        return p_hw_tim->ccr[ch];
    } else {
        return p_hw_tim->ccr5;
    }
}
```



```

/**
 * \brief 设置比较输出通道的预装载值
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] value    : 比较输出通道的预装载值
 * \param[in] ch       : 输出通道, 有效值 0-4 (4 仅在高级定时器 TIM1 有效)
 *
 * \note: 通用定时器 TIM2 该寄存器为 32 位, 其它为 16 位
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ccr_output_reload_val_set (amhw_zmf2x6_tim_t *p_hw_tim,
                                                  uint32_t      value,
                                                  uint8_t      ch)
{
    if (ch != AMHW_ZMF2X6_TIM_CC5S) {
        p_hw_tim->ccr[ch] = value;
    } else {
        p_hw_tim->ccr5 = value;
    }
}

```

2. TIMx_CCMR (捕获/比较模式寄存器)

该寄存器包括输出比较通道选择、通道模式, 快速使能、预装载使能以及清 0 使能的操作函数, 如程序清单 15.4 所示。

程序清单 15.4 TIMx_CCMR 寄存器操作函数

```

am_static_inline
void amhw_zmf2x6_tim_ccs_set (amhw_zmf2x6_tim_t *p_hw_tim,
                              uint8_t      ccs_opt,
                              uint8_t      ccs_chan)
{
    if (ccs_chan != AMHW_ZMF2X6_TIM_CC5S) {
        AM_BITS_SET(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01)), 2, ccs_opt);
    }
}

/**
 * \brief 通道输出比较清 0 使能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 *
 * \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 */

```

```

* \return 无
*/
am_static_inline
void amhw_zmf2x6_tim_ccs_occe_enable (amhw_zmf2x6_tim_t *p_hw_tim,
                                     uint8_t          ccs_chan)
{
    if (ccs_chan != AMHW_ZMF2X6_TIM_CC5S) {
        AM_BIT_MODIFY(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01) + 7), 1);
    } else {
        AM_BIT_MODIFY(p_hw_tim->ccmr3, 7, 1);
    }
}

/**
* \brief 通道输出比较清 0 禁能
*
* \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
*
* \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_tim_ccs_occe_disable (amhw_zmf2x6_tim_t *p_hw_tim,
                                       uint8_t          ccs_chan)
{
    if (ccs_chan != AMHW_ZMF2X6_TIM_CC5S) {
        AM_BIT_MODIFY(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01) + 7), 0);
    } else {
        AM_BIT_MODIFY(p_hw_tim->ccmr3, 7, 0);
    }
}

/**
* \brief 通道输出比较模式
*
* \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
* \param[in] ocm_opt : 输出比较模式选择, 值为 amhw_zmf2x6_tim_output_cmp_mode_t 这一枚举类型
*
* \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
*
* \note : 1. OC*M 一旦 LOCK 级别设为 3 (TIMx_BDTR 寄存器中的 LOCK 位) 并且 CC*S = 00 (该
*          通道配置成输出) 则该位不能被修改
*          2. 在 PWM 模式 1 或 PWM 模式 2 中, 只有当比较结果改变了或在输出比较模式中从

```

```

*          冻结模式切换到 PWM 模式时，OC*REF 电平才改变
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_tim_ocm_set (amhw_zmf2x6_tim_t *p_hw_tim,
                             uint8_t          ocm_opt,
                             uint8_t          ccs_chan)
{
    if (ccs_chan != AMHW_ZMF2X6_TIM_CC5S) {
        AM_BITS_SET(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01) + 4) , 3, ocm_opt);
    } else {
        AM_BITS_SET(p_hw_tim->ccmr3, 4, 3, ocm_opt);
    }
}

/**
* \brief 通道输出比较预装载使能
*
* \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
*
* \param[in] ccs_chan : ccs 通道选择，值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_tim_ccs_ocpe_enable (amhw_zmf2x6_tim_t *p_hw_tim,
                                       uint8_t          ccs_chan)
{
    if (ccs_chan != AMHW_ZMF2X6_TIM_CC5S) {
        AM_BIT_MODIFY(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01) + 3), 1);
    } else {
        AM_BIT_MODIFY(p_hw_tim->ccmr3, 3, 1);
    }
}

/**
* \brief 通道输出比较预装载禁能
*
* \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
*
* \param[in] ccs_chan : ccs 通道选择，值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
*
* \return 无

```

```

*/
am_static_inline
void amhw_zmf2x6_tim_ccs_ocpe_disable (amhw_zmf2x6_tim_t *p_hw_tim,
                                      uint8_t          ccs_chan)
{
    if (ccs_chan != AMHW_ZMF2X6_TIM_CC5S) {
        AM_BIT_MODIFY(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01) + 3), 0);
    } else {
        AM_BIT_MODIFY(p_hw_tim->ccmr3, 3, 0);
    }
}

/**
 * \brief 通道输出比较快速使能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 *
 * \param[in] ccs_chan : ccs 通道选择，值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ccs_ocfe_enable (amhw_zmf2x6_tim_t *p_hw_tim,
                                      uint8_t          ccs_chan)
{
    if (ccs_chan != AMHW_ZMF2X6_TIM_CC5S) {
        AM_BIT_MODIFY(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01) + 2), 1);
    } else {
        AM_BIT_MODIFY(p_hw_tim->ccmr3, 2, 1);
    }
}

/**
 * \brief 通道输出比较快速禁能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 *
 * \param[in] ccs_chan : ccs 通道选择，值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ccs_ocfe_disable (amhw_zmf2x6_tim_t *p_hw_tim,
                                       uint8_t          ccs_chan)

```

```

{
    if (ccs_chan != AMHW_ZMF2X6_TIM_CC5S) {
        AM_BIT_MODIFY(p_hw_tim->ccmr[ccs_chan >> 1], (8 * (ccs_chan & 0x01) + 2), 0);
    } else {
        AM_BIT_MODIFY(p_hw_tim->ccmr3, 2, 0);
    }
}
}

```

3. TIMx_CCER（捕获/比较使能寄存器）

该寄存器包括输出比较使能和极性、互补输出使能和极性的操作函数，如程序清单 15.5 所示。

程序清单 15.5 TIMx_CCER 寄存器操作函数

```

/**
 * \brief 通道比较输出电平极性
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ccp_opt : 通道比较输出电平极性选择, 0: OC1 高电平有效; 1: OC1 低电平有效
 *
 * \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_ccp_output_set (amhw_zmf2x6_tim_t *p_hw_tim,
                                     uint8_t          ccp_opt,
                                     uint8_t          ccs_chan)
{
    AM_BIT_MODIFY(p_hw_tim->ccer, (4 * ccs_chan + 1), ccp_opt);
}

/**
 * \brief 比较输出通道使能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_cce_output_enable (amhw_zmf2x6_tim_t *p_hw_tim,
                                         uint8_t          ccs_chan)
{
    AM_BIT_MODIFY(p_hw_tim->ccer, (4 * ccs_chan), 1);
}

```

```

/**
 * \brief 比较输出通道禁能
 *
 * \param[in] p_hw_tim : 指向 TIM 定时器寄存器块的指针
 * \param[in] ccs_chan : ccs 通道选择, 值为 amhw_zmf2x6_tim_ccs_chan_t 这一类型的枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_tim_cce_output_disable(amhw_zmf2x6_tim_t *p_hw_tim,
                                         uint8_t          ccs_chan)
{
    AM_BIT_MODIFY(p_hw_tim->ccer, (4 * ccs_chan), 0);
}

```

还有诸多寄存器, 可按上述样例逐一添加, PWM 外设寄存器众多, 本节不再一一罗列。

15.2.3 添加其他功能性函数

在每个寄存器的操作函数编写完毕后, 可能 HW 层的接口还有所欠缺, 需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到, 可以暂时不用考虑, 直接进入驱动层的开发。在后续如果发现 HW 层欠缺哪些接口, 可以反过来再增加。当然, 也有可能 HW 层接口没有什么欠缺了, 那样就无需再开发其它函数了。

15.2.4 添加指向寄存器块的指针

PWM 是定时器的其中一项功能, 此节内容与 13.2.4 章节一致。

15.3 驱动层开发

参考 5.3.1 章节, 新建 am_zmf2x6_tim_pwm.h 和 am_zmf2x6_tim_pwm.c 驱动文件, 搭建相应的文件框架, 并按规则放到对应的路径下。

15.3.1 编写结构性程序

驱动层首先应该搭建好相关结构, 再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中, 每个设备都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配, 与设备相关的变量都应该定义在设备结构体中, 而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下, 设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员, 用于驱动层使用 HW 层的接口。如果设备需要使用中断, 还需要填入中断号信息。除此之外, 一些设备的初始化配置信息, 也可以由设备信息提供。

定义 PWM 外设设备信息结构体如程序清单 15.6 所示。后续可以根据实际情况修改。

程序清单 15.6 PWM 设备信息结构体定义

```

/**
 * \brief TIM_PWM 输出功能相关的 GPIO 信息

```

```

*/
typedef struct am_zmf2x6_tim_pwm_chaninfo {
    int8_t    channel;          /**< \brief PWM 所使用的通道标识符 */
    int8_t    gpio;             /**< \brief PWM 输出所用的 GPIO 引脚 */
    uint32_t  func;             /**< \brief PWM 功能的 GPIO 功能设置值 */
    uint32_t  dfunc;            /**< \brief 禁能 PWM 模式后，默认 GPIO 功能设置值 */
} am_zmf2x6_tim_pwm_chaninfo_t;

/**
 * \brief TIMPWM 输出功能相关的设备信息
 */
typedef struct am_zmf2x6_tim_pwm_devinfo {
    uint32_t    tim_regbase;     /**< \brief TIM 寄存器块基址 */

    int          clk_num;         /**< \brief 时钟 ID */

    uint8_t      channels_num;    /**< \brief 使用的通道数，最大为 4 */

    /** \brief PWM 的模式， AMHW_ZMF2X6_TIM_PWM_MODE2 或
    AMHW_ZMF2X6_TIM_PWM_MODE1 */
    uint8_t      pwm_mode;

    /** \brief PWM 输出极性,0 为高电平有效, 1 为低电平有效 */
    uint8_t      ocpolarity;

    am_zmf2x6_tim_pwm_chaninfo_t *p_chaninfo;    /**< \brief 指向 PWM 输出通道信息结构体 */

    amhw_zmf2x6_tim_type_t    tim_type;          /**< \brief 定时器类型 */

    /** \brief 平台初始化函数，如打开时钟，配置引脚等工作 */
    void (*pfn_plfm_init)(void);

    /** \brief 平台解初始化函数 */
    void (*pfn_plfm_deinit)(void);
} am_zmf2x6_tim_pwm_devinfo_t;

```

● 设备结构体定义

由于 PWM 设备需要提供标准的 PWM 服务，包括 PWM 脉宽周期配置、输出使能等。其次，为了在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 15.7 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 15.7 PWM 设备结构体定义

```

/**
 * \brief TIMPWM 输出功能设备结构体
 */
typedef struct am_zmf2x6_tim_pwm_dev {

    am_pwm_serv_t                pwm_serv; /**< \brief 标准 PWM 服务 */

    /** \brief 指向 TIM(PWM 输出功能)设备信息常量的指针 */
    const am_zmf2x6_tim_pwm_devinfo_t  *p_devinfo;

} am_zmf2x6_tim_pwm_dev_t;

```

2. 搭建函数框架

函数主要分为两部分，一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于 PWM 驱动层需要提供初始化函数之外，还需要实现标准接口层函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为用于标准接口的 handle。示例代码如程序清单 15.8 所示。

程序清单 15.8 驱动层函数定义示例

```

/**
 * \brief 初始化 TIM 为 PWM 输出功能
 *
 * \param[in] p_dev      : 指向 TIM(PWM 输出功能)设备的指针
 * \param[in] p_devinfo : 指向 TIM(PWM 输出功能)设备信息常量的指针
 *
 * \return PWM 标准服务操作句柄，值为 NULL 时表明初始化失败
 */
am_pwm_handle_t am_zmf2x6_tim_pwm_init(am_zmf2x6_tim_pwm_dev_t      *p_dev,
                                       const am_zmf2x6_tim_pwm_devinfo_t  *p_devinfo);

/**
 * \brief 不使用 TIMPWM 输出功能时，解初始化 TIMPWM 输出功能，释放相关资源
 *
 * \param[in] handle : am_zmf2x6_tim_pwm_init() 初始化函数获得的 PWM 服务句柄
 *
 * \return 无
 */
void am_zmf2x6_tim_pwm_deinit (am_pwm_handle_t handle);

```

● 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 3 个函数，根据标准接口层中的 struct am_pwm_drv_funcs 结构体，我们需要在.c 文件中定义 3 个函数。

由于这些函数全部是放在结构体中，通过函数指针的形式提供给上层，并不需要将这些

函数声明到.h 文件中供用户直接使用，因此全部以 static 的形式定义成内部函数。定义好函数后，再定义一个 struct am_pwm_drv_funcs 结构体，将函数放在其中。示例代码如程序清单 15.9 所示。

程序清单 15.9 提供给标准层函数定义

```
/** \brief 配置 PWM */
static int __tim_pwm_config (void          *p_drv,
                               int          chan,
                               unsigned long duty_ns,
                               unsigned long period_ns);

/** \brief 使能 PWM 输出 */
static int __tim_pwm_enable (void *p_drv, int chan);

/** \brief 禁能 PWM 输出 */
static int __tim_pwm_disable (void *p_drv, int chan);

/** \brief PWM 驱动函数 */
static const struct am_pwm_drv_funcs __g_tim_pwm_drv_funcs = {
    __tim_pwm_config,
    __tim_pwm_enable,
    __tim_pwm_disable,
};
```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。

15.3.2 驱动函数实现

通过上面的函数框架搭建，共计 5 个函数，3 个功能性函数面向标准层，2 个初始解初始化函数面向用户，下面逐步实现。

1. am_zmf2x6_tim_pwm_init 函数

该函数主要完成两个功能：

- a) 初始化设备中的标准服务，将成员赋值。
- b) 初始化硬件相关设置，该函数的示例代码如程序清单 15.10 所示。

程序清单 15.10 am_zmf2x6_tim_pwm_init 函数示例代码

```
/**
 * \brief tim pwm 服务初始化
 */
am_pwm_handle_t am_zmf2x6_tim_pwm_init (am_zmf2x6_tim_pwm_dev_t *p_dev,
                                          const am_zmf2x6_tim_pwm_devinfo_t *p_devinfo)
{
    amhw_zmf2x6_tim_t *p_hw_tim = NULL;

    if (p_dev == NULL || p_devinfo == NULL) {
        return NULL;
    }
```

```

}

if (p_devinfo->pfn_plfm_init) {
    p_devinfo->pfn_plfm_init();
}

p_dev->p_devinfo      = p_devinfo;
p_hw_tim             = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
p_dev->pwm_serv.p_funcs = (struct am_pwm_drv_funcs *)&__g_tim_pwm_drv_funcs;
p_dev->pwm_serv.p_drv   = p_dev;

__tim_pwm_init(p_hw_tim, p_devinfo->tim_type);

return &(p_dev->pwm_serv);
}

```

2. am_zmf2x6_tim_pwm_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放。

因此，该函数仅简单的将几个不再使用的变量置为 NULL 即可。示例代码如程序清单 15.11 所示。

程序清单 15.11 am_zmf2x6_tim_pwm_deinit 函数示例代码

```

void am_zmf2x6_tim_pwm_deinit (am_pwm_handle_t handle)
{

    am_zmf2x6_tim_pwm_dev_t *p_dev      = (am_zmf2x6_tim_pwm_dev_t *)handle;
    amhw_zmf2x6_tim_t      *p_hw_tim = NULL;

    if (p_dev == NULL) {
        return;
    }

    p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;

    /* 清零计数器 */
    amhw_zmf2x6_tim_count_set(p_hw_tim, 0);

    /* 禁能定时器 TIM 允许计数 */
    amhw_zmf2x6_tim_disable(p_hw_tim);

    p_dev->pwm_serv.p_drv = NULL;

    if (p_dev->p_devinfo->pfn_plfm_deinit) {
        p_dev->p_devinfo->pfn_plfm_deinit();
    }
}

```

```

}
}

```

3. __tim_pwm_config

该函数主要用于配置 PWM 脉宽和周期时间，功能实现代码如程序清单 15.12 所示。

程序清单 15.12 __tim_pwm_config 函数示例代码

```

/** \brief 配置 PWM */
static int __tim_pwm_config (void          *p_drv,
                             int           chan,
                             unsigned long duty_ns,
                             unsigned long period_ns)
{
    am_zmf2x6_tim_pwm_dev_t *p_dev      = (am_zmf2x6_tim_pwm_dev_t *)p_drv;
    amhw_zmf2x6_tim_t       *p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;

    uint32_t clkfreq;
    uint32_t period_c, duty_c, temp;
    uint16_t pre_real = 1, pre_reg = 0;

    /* 参数不合法 */
    if ((period_ns == 0) || (duty_ns > 4294967295UL) ||
        (period_ns > 4294967295UL) || (duty_ns > period_ns)) {
        return -AM_EINVAL;
    }

    clkfreq = am_clk_rate_get(p_dev->p_devinfo->clk_num);

    /* 计算出来得到的是计数值 CNT, 公式 ns * 10e-9= cnt * (1/clkfreq) */
    period_c = (uint64_t)(period_ns) * (clkfreq) / (uint64_t)1000000000;
    duty_c   = (uint64_t)(duty_ns)   * (clkfreq) / (uint64_t)1000000000;

    {
        /* 当计数小于 65536 时，不分频(值为 1,1 代表为 1 分频) */
        temp = period_c / 65536 + 1;

        /* 16 位定时器需要运算取得合适的分频值 */
        for (pre_real = 1; pre_real < temp; ) {
            pre_reg++;          /* 计算写入寄存器的分频值 0,1,2,... */
            pre_real++;          /* 分频数 */
        }
    }

    /* 写入分频值 */

```

```

amhw_zmf2x6_tim_prescale_set(p_hw_tim, pre_reg);

/* 重新计算 PWM 的周期及脉冲频率 */
period_c = period_c / pre_real;
duty_c = duty_c / pre_real;

/* 共用一个重载寄存器，PWM 波所有的周期都相同 */
amhw_zmf2x6_tim_arr_set(p_hw_tim, period_c - 1);

/* 使能定 ARR 预装载 */
amhw_zmf2x6_tim_arpe_enable(p_hw_tim);

/* Low level time */
duty_c = period_c - duty_c;

/* 填充的值为 PWM 输出低电平的时间 */
amhw_zmf2x6_tim_ccr_ouput_reload_val_set(p_hw_tim, duty_c - 1, chan);

/* 选择该通道为输出 */
amhw_zmf2x6_tim_ccs_set(p_hw_tim, 0, chan);

/* 选择该通道的 PWM 模式 */
amhw_zmf2x6_tim_ocm_set(p_hw_tim, p_dev->p_devinfo->pwm_mode, chan);

/* PWM 输出通道预装载使能 */
amhw_zmf2x6_tim_ccs_ocpe_enable(p_hw_tim, chan);

if ((p_dev->p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE0) ||
    (p_dev->p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE2) ||
    (p_dev->p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE3)) {

    /* 设置 CR2 寄存器确定 N 极极性 */
}

/* 使能互补输出高电平有效 */
amhw_zmf2x6_tim_ccne_set(p_hw_tim, p_dev->p_devinfo->ocpolarity, chan);

/* 设置比较输出通道高电平极性有效 */
amhw_zmf2x6_tim_ccp_output_set(p_hw_tim, p_dev->p_devinfo->ocpolarity, chan);

return AM_OK;
}

```

4. __tim_pwm_enable

该函数主要用于使能 PWM 通道输出，功能实现代码如程序清单 15.13 所示。

程序清单 15.13 __tim_pwm_enable 函数示例代码

```

/**
 * \brief 使能 PWM 通道输出
 */
static int __tim_pwm_enable(void *p_drv, int chan)
{
    int i = 0, enable_flag = 0;
    am_zmf2x6_tim_pwm_dev_t *p_dev = (am_zmf2x6_tim_pwm_dev_t *)p_drv;
    amhw_zmf2x6_tim_t *p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    am_zmf2x6_tim_pwm_chaninfo_t *p_chaninfo = p_dev->p_devinfo->p_chaninfo;

    /* 判断引脚列表中是否有对应通道配置信息 */
    for(i = 0; i <= p_dev->p_devinfo->channels_num; i++){
        if((p_chaninfo[i].channel & 0x7f) == chan){
            am_gpio_pin_cfg(p_chaninfo[i].gpio, p_chaninfo[i].func);
            if((p_chaninfo[i].channel & 0x80) == 0x80){
                /* 使能互补输出 */
                amhw_zmf2x6_tim_ccne_enable(p_hw_tim, chan);
                enable_flag = 1;
            }else{
                /* 使能通道 PWM 输出 */
                amhw_zmf2x6_tim_cce_output_enable(p_hw_tim, chan);
                enable_flag = 1;
            }
        }
    }

    /* 输入通道号无效 */
    if(enable_flag == 0){
        return -AM_EINVAL;
    }

    /* 高级定时器使能主输出 MOE */
    if ((p_dev->p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE0) ||
        (p_dev->p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE2) ||
        (p_dev->p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE3)) {

        amhw_zmf2x6_tim_bdtr_enable(p_hw_tim, AMHW_ZMF2X6_TIM_MOE);
    }

    /* 产生更新事件，重新初始化 Prescaler 计数器及 Repetition 计数器 */
    amhw_zmf2x6_tim_egr_set(p_hw_tim, AMHW_ZMF2X6_TIM_UG);

    if (amhw_zmf2x6_tim_status_flg_get(p_hw_tim, AMHW_ZMF2X6_TIM_UG) != 0) {

```

```

    /* 更新定时器时会产生更新事件,清除标志位 */
    amhw_zmf2x6_tim_status_flg_clr(p_hw_tim, AMHW_ZMF2X6_TIM_UG);
}

/* 使能定时器 TIM 允许计数 */
amhw_zmf2x6_tim_enable(p_hw_tim);

return AM_OK;
}

```

5. __tim_pwm_disable

该函数主要用于禁能 PWM 通道输出，功能实现代码如程序清单 15.14 所示。

程序清单 15.14 __tim_pwm_disable 函数示例代码

```

/**
 * \brief 禁能 PWM 通道输出
 */
static int __tim_pwm_disable (void *p_drv, int chan)
{
    int i = 0, disable_flag = 0;
    am_zmf2x6_tim_pwm_dev_t *p_dev = (am_zmf2x6_tim_pwm_dev_t *)p_drv;
    amhw_zmf2x6_tim_t *p_hw_tim = (amhw_zmf2x6_tim_t *)p_dev->p_devinfo->tim_regbase;
    am_zmf2x6_tim_pwm_chaninfo_t *p_chaninfo = p_dev->p_devinfo->p_chaninfo;

    /* 判断引脚列表中是否有对应通道配置信息 */
    for(i = 0; i <= p_dev->p_devinfo->channels_num; i++){
        if((p_chaninfo[i].channel & 0x7f) == chan){
            am_gpio_pin_cfg(p_chaninfo[i].gpio, p_chaninfo[i].dfunc);
            disable_flag = 1;
        }
    }

    /* 输入通道号无效 */
    if(disable_flag == 0){
        return -AM_EINVAL;
    }

    /* 禁能定时器 TIM 允许计数 */
    amhw_zmf2x6_tim_disable(p_hw_tim);

    /* 禁能通道 PWM 输出 */
    amhw_zmf2x6_tim_cce_output_disable(p_hw_tim, chan);

    /* 高级定时器禁能主输出 MOE */

```

```

if ((p_dev->p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE0) ||
    (p_dev->p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE2) ||
    (p_dev->p_devinfo->tim_type == AMHW_ZMF2X6_TIM_TYPE3)) {

    amhw_zmf2x6_tim_bdtr_disable(p_hw_tim, AMHW_ZMF2X6_TIM_MOE);

}

return AM_OK;
}

```

15.4 用户配置文件编写

参考 5.4.1 章节新建用户配置文件，搭建文件的基本框架，准备配置文件内容的编写。

15.4.1 定义设备常量于设备信息常量

设备信息常量初始化，默认填充的信息是一种常用的配置方式，使用 TIM1 的 PWM 输出功能为例，详细的配置如程序清单 15.15 所示。

程序清单 15.15 PWM 设备信息初始化配置

```

/** \brief TIM1 用于 PWM 功能的通道配置信息列表 */
static am_zmf2x6_tim_pwm_chaninfo_t __g_tim1_pwm_chaninfo_list[] = {
    /** \brief 通道 1 引脚配置 */
    {AM_ZMF2X6_TIM_PWM_CH1, PIOA_8, PIOA_8_TIM1_CH1 | PIOA_8_AF_PP, PIOA_8_GPIO |
    PIOA_8_INPUT_FLOAT},
    /** \brief 通道 1 互补引脚配置 */
    {AM_ZMF2X6_TIM_PWM_CH1N, PIOB_13, PIOB_13_TIM1_CH1N | PIOB_13_AF_PP, PIOB_13_GPIO
    | PIOB_13_INPUT_FLOAT},
    /** \brief 通道 2 引脚配置 */
    {AM_ZMF2X6_TIM_PWM_CH2, PIOA_9, PIOA_9_TIM1_CH2 | PIOA_9_AF_PP, PIOA_9_GPIO |
    PIOA_9_INPUT_FLOAT},
    /** \brief 通道 2 互补引脚配置 */
    {AM_ZMF2X6_TIM_PWM_CH2N, PIOB_14, PIOB_14_TIM1_CH2N | PIOB_14_AF_PP, PIOB_14_GPIO
    | PIOB_14_INPUT_FLOAT},
    /** \brief 通道 3 引脚配置 */
    {AM_ZMF2X6_TIM_PWM_CH3, PIOA_10, PIOA_10_TIM1_CH3 | PIOA_10_AF_PP, PIOA_10_GPIO |
    PIOA_10_INPUT_FLOAT},
    /** \brief 通道 3 互补引脚配置 */
    {AM_ZMF2X6_TIM_PWM_CH3N, PIOB_1, PIOB_1_TIM1_CH3N | PIOB_1_AF_PP, PIOB_1_GPIO |
    PIOB_1_INPUT_FLOAT},
    /** \brief 通道 4 引脚配置 */
    {AM_ZMF2X6_TIM_PWM_CH4, PIOA_11, PIOA_11_TIM1_CH4 | PIOA_11_AF_PP, PIOA_11_GPIO |
    PIOA_11_INPUT_FLOAT},
};

/** \brief TIM1 用于 PWM 设备信息 */
static const am_zmf2x6_tim_pwm_devinfo_t __g_tim1_pwm_devinfo = {

```

```

ZMF2X6_TIM1_BASE,                /**< \brief TIM1 寄存器块的基地址 */
CLK_TIM1,                        /**< \brief TIM1 时钟 ID */
AM_NELEMENTS(__g_tim1_pwm_chaninfo_list), /**< \brief 配置输出通道个数 */
AMHW_ZMF2X6_TIM_PWM_MODE2,       /**< \brief PWM 输出模式 2
*/
0,                                /**< \brief PWM 输出高电平有效 */
&__g_tim1_pwm_chaninfo_list[0],  /**< \brief 通道配置信息列表 */
AMHW_ZMF2X6_TIM_TYPE0,           /**< \brief 定时器类型 */
__zmf2x6_plfm_tim1_pwm_init,      /**< \brief 平台初始化函数 */
__zmf2x6_plfm_tim1_pwm_deinit    /**< \brief 平台解初始化函数 */
};

```

15.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

平台初始化函数和解初始化函数定义如程序清单 15.16 所示。

程序清单 15.16 平台初始化函数和解初始化函数定义

```

/** \brief TIM1 用于 PWM 输出 平台初始化 */
static void __zmf2x6_plfm_tim1_pwm_init(void)
{
    /* 使能高级定时器 1 时钟 */
    am_clk_enable(CLK_TIM1);

    /* 复位高级定时器 1 */
    am_zmf2x6_clk_reset(CLK_TIM1);
}

/** \brief 解除 TIM1 平台初始化 */
static void __zmf2x6_plfm_tim1_pwm_deinit(void)
{
    /* 复位高级定时器 1 */
    am_zmf2x6_clk_reset(CLK_TIM1);

    /* 禁能高级定时器 1 时钟 */
    am_clk_disable(CLK_TIM1);
}

```

15.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在标准层实例公共头文件 am_zmf2x6_inst_init.h 中加入它们的声明，如程序清单 15.17 所示。

程序清单 15.17 初始化解初始化声明

```

/**
 * \brief TIM1 PWM 实例初始化，获得 PWM 标准服务句柄

```



```

*
* \param 无
*
* \return PWM 标准服务句柄，若为 NULL，表明初始化失败
*/
am_pwm_handle_t am_zmf2x6_tim1_pwm_inst_init(void);

/**
* \brief TIM1 PWM 实例解初始化
*
* \param[in] handle 通过 am_zmf2x6_tim1_pwm_inst_init() 函数
*                  获得的 PWM 标准服务句柄
*
* \return 无
*/
void am_zmf2x6_tim1_pwm_inst_deinit(am_pwm_handle_t handle);

```

用户使用时，应该先调用实例初始化函数 `am_zmf2x6_tim1_pwm_inst_init` 函数，初始化 TIM1 的 PWM 功能，并返回 PWM 标准服务句柄。

至此，用户配置文件开发完毕。

15.5 例程文件开发

例程文件开发主要分为两个部分，HW 例程和 STD 例程。

调试例程时，应该在 `template` 工程下进行，不需要新建其它工程，只需要将新建的文件加入到工程结构的 `user_code` 下。例程文件会与原先的 `main.c` 文件冲突（因为都存在 `main` 函数），这时，只需要调试时将原先的 `main.c` 不要包含在工程编译中，例程文件开发完毕后，再恢复将 `main.c` 包含在工程中，同时将例程文件移除工程即可。

15.5.1 例程内容开发

1. HW 层实现示例

这里使用 HW 层来实现定时器输出 PWM，相关代码如程序清单 15.18 所示。

程序清单 15.18 定时器输出 PWM

```

/** [src_zmf2x6_core_hw_tim_pwm] */
#include "ametal.h"
#include "am_clk.h"
#include "am_gpio.h"
#include "am_led.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_zmf2x6.h"
#include "am_zmf2x6_clk.h"
#include "am_zmf2x6_tim_pwm.h"
#include "demo_zmf2x6_entries.h"
#include "demo_zmf2x6_core_entries.h"

```

```

#define LED0          0

/**
 * \brief PWM 中断服务函数
 */
static void __zmf2x6_tim_hw_pwm_irq_handler(void *p_arg)
{
    amhw_zmf2x6_tim_t *p_hw_tim = (amhw_zmf2x6_tim_t *)p_arg;
    uint8_t i = 1;

    for (i = 1; i <= 4; i++) {
        if ((amhw_zmf2x6_tim_status_flg_get(p_hw_tim, 1UL << i) != 0) {

            am_led_toggle(LED0);

            /* 清除通道 i 标志 */
            amhw_zmf2x6_tim_status_flg_clr(p_hw_tim, (1UL << i));
        }
    }
}

/**
 * \brief 配置定时器为 PWM 输出(PWM 模式 2)
 */
void tim_pwm_chan_config(amhw_zmf2x6_tim_t *p_hw_tim,
                        uint32_t chan,
                        uint32_t duty_ns,
                        uint32_t period_ns,
                        uint32_t clk_rate)
{
    uint32_t period_c, duty_c, temp;
    uint16_t pre_real = 1, pre_reg = 0;

    /* 参数不合法 */
    if ((period_ns == 0) || (duty_ns > 4294967295UL) ||
        (period_ns > 4294967295UL) || (duty_ns > period_ns)) {
        return ;
    }

    /* 计算出来得到的是计数值 CNT, 公式 ns * 10e-9= cnt * (1/clkfrq) */
    period_c = (uint64_t)(period_ns) * (clk_rate) / (uint64_t)1000000000;
    duty_c = (uint64_t)(duty_ns) * (clk_rate) / (uint64_t)1000000000;

```

```

{
    /* 当计数小于 65536 时，不分频(值为 1,1 代表为 1 分频) */
    temp = period_c / 65536 + 1;

    /* 16 位定时器需要运算取得合适的分频值 */
    for (pre_real = 1; pre_real < temp; ) {
        pre_reg++;          /* 计算写入寄存器的分频值 0,1,2,... */
        pre_real++;         /* 分频数 */
    }
}

/* 设置分频值 */
amhw_zmf2x6_tim_prescale_set(p_hw_tim, pre_reg);

/* 重新计算 PWM 的周期及脉冲频率 */
period_c = period_c / pre_real;
duty_c = duty_c / pre_real;

/* 设置自动重装寄存器的值 */
amhw_zmf2x6_tim_arr_set(p_hw_tim, period_c - 1);

/* 设置比较输出通道的匹配值 */
amhw_zmf2x6_tim_ccr_output_reload_val_set(p_hw_tim, duty_c - 1, chan);

/* 计数器清 0 */
amhw_zmf2x6_tim_count_set(p_hw_tim, 0);

/* 使能定 ARR 预装载 */
amhw_zmf2x6_tim_arpe_enable(p_hw_tim);

/* 选择该通道为输出 */
amhw_zmf2x6_tim_ccs_set(p_hw_tim, 0, chan);

/* 选择该通道的模式为 PWM 模式 2 */
amhw_zmf2x6_tim_ocm_set(p_hw_tim, AMHW_ZMF2X6_TIM_PWM_MODE2, chan);

/* PWM 输出通道预装载使能 */
amhw_zmf2x6_tim_ccs_ocpe_enable(p_hw_tim, chan);

/* 设置比较输出通道 ccp 高电平极性有效 */
amhw_zmf2x6_tim_ccp_output_set(p_hw_tim, 0, chan);
}

void tim_pwm_enable (amhw_zmf2x6_tim_t      *p_hw_tim,

```

```

        amhw_zmf2x6_tim_type_t  type,
        uint32_t                chan,
        int32_t                 int_num)
{

    /* 使能通道 PWM 输出 */
    amhw_zmf2x6_tim_cce_output_enable(p_hw_tim, chan);

    /* 高级定时器使能主输出 MOE */
    if ((AMHW_ZMF2X6_TIM_TYPE0 == type) ||
        (AMHW_ZMF2X6_TIM_TYPE2 == type) ||
        (AMHW_ZMF2X6_TIM_TYPE3 == type)) {

        amhw_zmf2x6_tim_bdtr_enable(p_hw_tim, AMHW_ZMF2X6_TIM_MOE);
    }

    /* 产生更新事件，重新初始化 Prescaler 计数器 及 Repetition 计数器 */
    amhw_zmf2x6_tim_egr_set(p_hw_tim, AMHW_ZMF2X6_TIM_UG);

    if (amhw_zmf2x6_tim_status_flg_get(p_hw_tim, AMHW_ZMF2X6_TIM_UG) != 0) {
        /* 更新定时器时会产生更新事件,清除标志位 */
        amhw_zmf2x6_tim_status_flg_clr(p_hw_tim, AMHW_ZMF2X6_TIM_UG);
    }

    /* 连接中断回调函数 */
    am_int_connect(int_num, __zmf2x6_tim_hw_pwm_irq_handler, (void *)p_hw_tim);

    /* 使能中断 */
    amhw_zmf2x6_tim_int_enable(p_hw_tim, (1ul << (chan + 1)));

    am_int_enable(int_num);

    /* 使能定时器 TIM 允许计数 */
    amhw_zmf2x6_tim_enable(p_hw_tim);
}

/**
 * \brief 定时器 TIM PWM 输出初始化函数
 *
 * \param[in] p_hw_tim : 指向定时器寄存器块的指针
 * \param[in] clk_id   : 时钟 ID (由平台定义), 参见 \ref grp_clk_id
 *
 * \return 无
 */

```

```

void tim_pwm_init (amhw_zmf2x6_tim_t      *p_hw_tim, amhw_zmf2x6_tim_type_t type)
{
    if ((AMHW_ZMF2X6_TIM_TYPE0 == type) || (AMHW_ZMF2X6_TIM_TYPE1 == type)) {

        /* 边沿对齐模式 */
        amhw_zmf2x6_tim_cms_set(p_hw_tim, 0);

        /* 向上计数 */
        amhw_zmf2x6_tim_dir_set(p_hw_tim, 0);
    }

    /* 设置时钟分割:TDTs = Tck_tin */
    amhw_zmf2x6_tim_ckd_set(p_hw_tim, 0);

    /* 允许更新事件 */
    amhw_zmf2x6_tim_udis_enable(p_hw_tim);
}

void demo_zmf2x6_hw_tim_pwm_entry (amhw_zmf2x6_tim_t      *p_hw_tim,
                                   amhw_zmf2x6_tim_type_t type,
                                   uint32_t                  chan,
                                   uint32_t                  clk_rate,
                                   int32_t                   inum)
{
    /* 初始化定时器 TIM 为 PWM 功能 */
    tim_pwm_init(p_hw_tim, type);

    /* 配置定时器 TIM PWM 输出通道 */
    tim_pwm_chan_config(p_hw_tim,
                        chan,
                        250000 / 2,
                        250000,
                        clk_rate);

    tim_pwm_enable(p_hw_tim, type, chan, inum);

    AM_FOREVER {
        ; /* VOID */
    }
}

/**
 * \brief 例程入口

```

```

*/
void demo_zmf2x6_core_hw_tim_pwm_entry (void)
{
    AM_DBG_INFO("demo zmf2x6_core hw tim pwm!\r\n");

    /* 初始化引脚 */
    am_gpio_pin_cfg(PIOA_0, PIOA_0_TIM2_CH1_ETR | PIOA_0_AF_PP);

    /* 使能定时器时钟 */
    am_clk_enable(CLK_TIM2);

    /* 复位定时器 */
    am_zmf2x6_clk_reset(CLK_TIM2);

    demo_zmf2x6_hw_tim_pwm_entry(ZMF2X6_TIM2,
                                   AMHW_ZMF2X6_TIM_TYPE1,
                                   AM_ZMF2X6_TIM_PWM_CH1,
                                   am_clk_rate_get(CLK_TIM2),
                                   INUM_TIM2);
}

```

2. 标准层实现示例

这里使用标准层来实现定时器输出 PWM，通过标准接口实现，相关代码如程序清单 15.19 所示。

程序清单 15.19 定时器输出 PWM

```

#include "ametal.h"
#include "am_vdebug.h"
#include "am_pwm.h"
#include "am_zmf2x6_tim_pwm.h"
#include "am_zmf2x6_inst_init.h"
#include "demo_std_entries.h"
#include "demo_zmf2x6_core_entries.h"

void demo_std_timer_pwm_entry (am_pwm_handle_t pwm_handle, int pwm_chan)
{
    am_pwm_config(pwm_handle, pwm_chan, 10000000 / 2, 10000000); /* 频率为 1000000000 / 10000000 = 100Hz */
    am_pwm_enable(pwm_handle, pwm_chan);

    AM_FOREVER {
        ; /* VOID */
    }
}

```

```
/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_std_tim_pwm_entry (void)
{
    AM_DBG_INFO("demo zmf2x6_core std tim pwm!\r\n");

    demo_std_timer_pwm_entry(am_zmf2x6_tim1_pwm_inst_init(), AM_ZMF2X6_TIM_PWM_CH1);
    // demo_std_timer_pwm_entry(am_zmf2x6_tim2_pwm_inst_init(), AM_ZMF2X6_TIM_PWM_CH1);
    // demo_std_timer_pwm_entry(am_zmf2x6_tim3_pwm_inst_init(), AM_ZMF2X6_TIM_PWM_CH1);
    // demo_std_timer_pwm_entry(am_zmf2x6_tim14_pwm_inst_init(), AM_ZMF2X6_TIM_PWM_CH1);
    // demo_std_timer_pwm_entry(am_zmf2x6_tim16_pwm_inst_init(), AM_ZMF2X6_TIM_PWM_CH1);
    // demo_std_timer_pwm_entry(am_zmf2x6_tim17_pwm_inst_init(), AM_ZMF2X6_TIM_PWM_CH1);
}
```

所有例程开发完毕后，整个外设的开发工作也就圆满结束了。

可以提交相关文件，以完成整个开发任务。提交文件时，不用提交工程文件，只需要提交开发的源文件和头文件即可。

第16章 通用外设驱动开发——RTC

16.1 标准层了解

本章以适配 RTC 外设为例，展示如何开发驱动代码。RTC 标准层仅包含一个文件 am_rtc.h，在 ametal\interface 找到该文件打开。

16.1.1 了解相关类型定义

打开该文件，可以看到该文件主要定义了 2 个结构体和 1 个 handle 类型，如程序清单 16.1 所示，下面对每个结构体做详细介绍。

程序清单 16.1 RTC 标准接口数据结构定义

```
/**
 * \brief RTC 驱动函数
 */
struct am_rtc_drv_funcs {

    /** \brief 设置当前的时间 */
    int (*pfn_rtc_time_set)(void *p_drv, am_tm_t *p_tm);

    /** \brief 获取当前的时间 */
    int (*pfn_rtc_time_get)(void *p_drv, am_tm_t *p_tm);
};

/**
 * \brief RTC 标准服务
 */
typedef struct am_rtc_serv {

    /** \brief RTC 驱动函数 */
    struct am_rtc_drv_funcs *p_funcs;

    /** \brief 驱动函数第一个入口参数 */
    void *p_drv;
} am_rtc_serv_t;

/** \brief RTC 标准服务操作句柄类型 */
typedef am_rtc_serv_t *am_rtc_handle_t;
```

1. struct am_rtc_drv_funcs 结构体

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层需要实现 2 个函数：

- pfn_rtc_time_set，用于设置当前时间的函数；
- pfn_rtc_time_get，用于获取当前时间的函数。

因此，在驱动层编写函数时，应该向标准接口层提供这 2 个函数。

这些函数的有一个共同点，就是第一个参数都是 void 型的指针，也就是说，驱动层提供的函数，标准接口层使用时传递的第一个参数并无确定类型，标准接口层不关心第一个参数的具体类型，使用时只负责传递就可以了。

2. typedef struct am_rtc_serv 结构体

该结构体定义了标准的 RTC 服务，包含了上文提到的驱动函数结构体的指针和一个 void 型的指针。

这个标准的 RTC 服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

● p_funcs 成员赋值

这是一个指向 struct am_rtc_drv_funcs 结构体的指针，我们只需要在驱动层定义一个 struct am_rtc_drv_funcs 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那 2 个函数地址。然后，将 struct am_rtc_drv_funcs 结构体变量的地址赋值给 p_funcs 成员即可。

● p_drv 成员赋值

这个变量是 void 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动自行决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

3. am_rtc_handle_t 的定义

这个是一个标准的操作句柄，本质上就是 struct am_rtc_serv 指针类型，驱动层的初始化函数需要返回该类型的一个 handle 给用户。这样，用户获取到 handle，就可以在需要使用标准接口层函数时，将该 handle 传递给标准接口层了。标准接口就可以通过该 handle (struct am_rtc_serv 结构体指针) 访问到驱动层提供的函数了。

16.1.2 了解标准接口函数

在了解 am_rtc.h 文件相关的结构体、数据类型定义后，接下来就是具体函数的实现，如程序清单 16.2 所示。

程序清单 16.2 RTC 标准接口函数

```
/**
 * \brief 设置当前的时间
 *
 * \param[in] handle : RTC 标准服务操作句柄
 * \param[in] p_tm    : 要设置的时间
 *
 * \return    AM_OK    : 设置成功
 * \retval    -AM_EINVAL : 参数错误
 */
am_static_inline
int am_rtc_time_set (am_rtc_handle_t handle, am_tm_t *p_tm)
{
    return handle->p_funcs->pfn_rtc_time_set(handle->p_drv, p_tm);
}

/**
```

```

* \brief 获取当前的时间
*
* \param[in] handle : RTC 标准服务操作句柄
* \param[out] p_tm : 读取的时间
*
* \return AM_OK : 获取时间成功
* \retval -AM_EINVAL : 参数错误
*/
am_static_inline
int am_rtc_time_get(am_rtc_handle_t handle, am_tm_t *p_tm)
{
    return handle->p_funcs->pfn_rtc_time_get(handle->p_drv, p_tm);
}

```

4. am_rtc_time_set()设置当前时间函数

该函数用于设置时间，传入的第一个参数为实例初始化获取的 `handle`，第二个参数为待设置的时间。

5. am_rtc_time_get ()获取当前时间函数

该函数用于获取时间，传入的第一个参数为实例初始化获取的 `handle`，第二个参数为用于存放时间的指针。

到此我们就分析完了 RTC 外设标准层数据类型和接口函数，清楚了驱动层需要为标准接口层提供哪些内容。

16.2 HW 层开发

参考 5.2.1 章节，创建对应目录和文件，添加硬件层基本的内容。

16.2.1 添加寄存器列表

HW 层直接与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读时钟外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMP2x6 用户手册的 RTC 章节，可以看到共有 12 个寄存器，简单描述如下：

- RCC_CRH (RTC 控制寄存器高位)：用于中断控制使能、禁能；
- RCC_CRL (RTC 控制寄存器低位)：RTC 标志位，包括秒标志、闹钟标志、溢出标志等；
- RCC_PRLH (RTC 预分频装载寄存器高位)：预分频装载寄存器高位的值；
- RCC_PRLH (RTC 预分频装载寄存器低位)：预分频装载寄存器低位的值；
- RCC_DIVH (RTC 预分频器分频因子寄存器高位)：预分频器分频因子高位值；
- RCC_DIVL (RTC 预分频器分频因子寄存器低位)：预分频器分频因子低位值；
- RCC_CNTH (RTC 计数寄存器高位)：计数寄存器高位值；
- RCC_CNTL (RTC 计数寄存器低位)：计数寄存器低位值；
- RCC_ALRH (RTC 闹钟寄存器高位)：闹钟时间高位值；
- RCC_ALRL (RTC 闹钟寄存器低位)：闹钟时间低位值；

- RCC_MSRH（RTC 毫秒寄存器高位）：毫秒闹钟时间高位；
- RCC_MSRL（RTC 毫秒寄存器低位）：毫秒闹钟时间低位；

到此 RTC 的各个寄存器及其功能基本阅读完毕，我们也大致了解了各个寄存器功能以及要配置各个功能，其他寄存器位该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，程序清单如程序清单 16.3 所示。

程序清单 16.3 RTC 寄存器结构体定义

```
/**
 * \brief RTC 寄存器结构体定义
 */
typedef struct amhw_zmf2x6_rtc {
    __IO uint16_t crh;          /**< \brief RTC 控制寄存器高位 */
    __I  uint16_t reserve0;     /**< \brief 保留位 */
    __IO uint16_t crl;          /**< \brief RTC 控制寄存器低位 */
    __I  uint16_t reserve1;     /**< \brief 保留位 */
    __IO uint16_t prlh;         /**< \brief RTC 预分频装载寄存器高位 */
    __I  uint16_t reserve2;     /**< \brief 保留位 */
    __IO uint16_t prll;         /**< \brief RTC 预分频装载寄存器低位 */
    __I  uint16_t reserve3;     /**< \brief 保留位 */
    __IO uint16_t divh;         /**< \brief RTC 预分频器分频因子寄存器高位 */
    __I  uint16_t reserve4;     /**< \brief 保留位 */
    __IO uint16_t divl;         /**< \brief RTC 预分频器分频因子寄存器低位 */
    __I  uint16_t reserve5;     /**< \brief 保留位 */
    __IO uint16_t cnth;         /**< \brief RTC 计数器寄存器高位 */
    __I  uint16_t reserve6;     /**< \brief 保留位 */
    __IO uint16_t cntl;         /**< \brief RTC 计数器寄存器低位 */
    __I  uint16_t reserve7;     /**< \brief 保留位 */
    __IO uint16_t alrh;         /**< \brief RTC 闹钟寄存器高位 */
    __I  uint16_t reserve8;     /**< \brief 保留位 */
    __IO uint16_t alrl;         /**< \brief RTC 闹钟寄存器低位 */
    __I  uint16_t reserve9;     /**< \brief 保留位 */
    __IO uint16_t msrh;         /**< \brief RTC 毫秒寄存器高位 */
    __I  uint16_t reserve10;     /**< \brief 保留位 */
    __IO uint16_t msrl;         /**< \brief RTC 毫秒寄存器低位 */
    __I  uint16_t reserve11;     /**< \brief 保留位 */
} amhw_zmf2x6_rtc_t;
```

这个小节我们定义了 RTC 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

16.2.2 编写寄存器的相关函数

1. RTC_CRH（控制寄存器高位）

通过用户手册，我们可以看到该寄存器有 3 个位域，均为中断控制位，可用枚举方式列

出。根据位域的功能可编写两个操作函数，一个用于中断使能，一个用于中断禁能，如程序清单 16.4 所示。

程序清单 16.4 RTC 中断控制函数

```
/**
 * \brief RTC 控制寄存器高位中断允许枚举
 */
typedef enum {
    AMHW_ZMF2X6_RTC_SECIE = 0,
    AMHW_ZMF2X6_RTC_ALRIE = 1,
    AMHW_ZMF2X6_RTC_OWIE = 2,
} amhw_zmf2x6_rtc_crh_int_permit;

/**
 * \brief 中断允许函数
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 * \param[in] permit    RTC 控制寄存器高位中断允许枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rtc_crh_allow_int (amhw_zmf2x6_rtc_t      *p_hw_rtc,
                                   amhw_zmf2x6_rtc_crh_int_permit permit)
{
    p_hw_rtc->crh |= (1 << permit);
}

/**
 * \brief 中断禁止函数
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 * \param[in] permit    RTC 控制寄存器高位中断允许枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rtc_crh_forbid_int (amhw_zmf2x6_rtc_t      *p_hw_rtc,
                                     amhw_zmf2x6_rtc_crh_int_permit permit)
{
    p_hw_rtc->crh &= ~(1 << permit);
}
```

2. RCC_CRL（控制寄存器低位）

该寄存器位域较多，包括标志位、配置位，根据位域功能编写对应的操作函数，如程序

清单 16.5 所示。

程序清单 16.5 RTC_CRL 寄存器操作函数

```
/**
 * \brief RTC 控制寄存器低位状态标志枚举
 */
typedef enum {
    AMHW_ZMF2X6_RTC_SECF = 0,
    AMHW_ZMF2X6_RTC_ALRF = 1,
    AMHW_ZMF2X6_RTC_OWF = 2,
    AMHW_ZMF2X6_RTC_RSF = 3,
    AMHW_ZMF2X6_RTC_RTOFF = 5,
} amhw_zmf2x6_rtc_crl_status;

/**
 * \brief 读控制寄存器低位状态位
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 * \param[in] status    RTC 控制寄存器低位状态标志枚举
 *
 * \retval AM_TRUE 标志位置位
 * \retval AM_FALSE 标志位未置位
 */
am_static_inline
am_bool_t amhw_zmf2x6_rtc_crl_read_statu (amhw_zmf2x6_rtc_t      *p_hw_rtc,
                                           amhw_zmf2x6_rtc_crl_status status)
{
    return (am_bool_t)((p_hw_rtc->crl & (1ul << status)) ? AM_TRUE : AM_FALSE);
}

/**
 * \brief 控制寄存器低位状态位清除
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 * \param[in] status    RTC 控制寄存器低位状态标志枚举
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rtc_clr_status_clear (amhw_zmf2x6_rtc_t      *p_hw_rtc,
                                       amhw_zmf2x6_rtc_crl_status status)
{
    p_hw_rtc->crl &= ~(1 << status);
}
```

```

/**
 * \brief 进入 RTC 配置模式
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rtc_crl_cnf_enter (amhw_zmf2x6_rtc_t *p_hw_rtc)
{
    p_hw_rtc->crl |= (1 << 4);
}

/**
 * \brief 退出 RTC 配置模式
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rtc_crl_cnf_out (amhw_zmf2x6_rtc_t *p_hw_rtc)
{
    p_hw_rtc->crl &= ~(1 << 4);
}

```

3. RCC_PRLH/RCC_PRL (预分频装载寄存器)

预分频装载寄存器分为高位和低位两个寄存器，操作方式一样，如程序清单 16.6 所示。

程序清单 16.6 预分频装载寄存器操作函数

```

/**
 * \brief RTC 预分频装载寄存器高位写函数
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 * \param[in] divh 预分频高位
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rtc_prlh_div_write (amhw_zmf2x6_rtc_t *p_hw_rtc, uint16_t divh)
{
    p_hw_rtc->prlh = (divh & 0x000F);
}

/**
 * \brief RTC 预分频装载寄存器低位写函数

```

```

*
* \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
* \param[in] divl      预分频低位
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rtc_prll_div_write (amhw_zmf2x6_rtc_t *p_hw_rtc, uint16_t divl)
{
    p_hw_rtc->prll = divl;
}

```

4. RCC_DIVH/RCC_DIVL（预分频器分频因子寄存器）

预分频器分频因子分为高位和低位两个寄存器，操作方式一样，如程序清单 16.7 所示。

程序清单 16.7 预分频器分频因子读取函数

```

/**
 * \brief RTC 预分频装载寄存器高位读函数
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 *
 * \return RTC 预分频寄存器高位状态值
 */
am_static_inline
uint16_t amhw_zmf2x6_rtc_divh_div_read (amhw_zmf2x6_rtc_t *p_hw_rtc)
{
    return p_hw_rtc->divh;
}

/**
 * \brief RTC 预分频装载寄存器低位读函数
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 *
 * \return RTC 预分频寄存器低位状态值
 */
am_static_inline
uint16_t amhw_zmf2x6_rtc_divl_div_read (amhw_zmf2x6_rtc_t *p_hw_rtc)
{
    return p_hw_rtc->divl;
}

```

5. RCC_CNTH/RCC_CNTL（计数器寄存器）

计数器为高位和低位两个寄存器，操作方式一样，如程序清单 16.8 所示。

程序清单 16.8 计数器寄存器操作函数

```

/**

```

```

* \brief RTC 计数器寄存器高位设置函数
*
* \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
* \param[in] cnth      计数器高位
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rtc_cnth_set (amhw_zmf2x6_rtc_t *p_hw_rtc, uint16_t cnth)
{
    p_hw_rtc->cnth = cnth;
}

/**
* \brief RTC 计数器寄存器低位设置函数
*
* \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
* \param[in] cntl      计数器低位
*
* \return 无
*/
am_static_inline
void amhw_zmf2x6_rtc_cntl_set (amhw_zmf2x6_rtc_t *p_hw_rtc, uint16_t cntl)
{
    p_hw_rtc->cntl = cntl;
}

/**
* \brief 获取 RTC 计数器寄存器高位计数值
*
* \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
*
* \return 计数器高位数值
*/
am_static_inline
uint16_t amhw_zmf2x6_rtc_cnth_get (amhw_zmf2x6_rtc_t *p_hw_rtc)
{
    return p_hw_rtc->cnth;
}

```



```

/**
 * \brief 获取 RTC 计数器寄存器低位计数值
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 *
 * \return 寄存器低位数值
 */
am_static_inline
uint16_t amhw_zmf2x6_rtc_cntl_get (amhw_zmf2x6_rtc_t *p_hw_rtc)
{
    return p_hw_rtc->cntl;
}

```

6. RCC_ALRH/RCC_ALRL（闹钟寄存器）

闹钟为高位和低位两个寄存器，操作方式一样，如程序清单 16.9 所示。

程序清单 16.9 闹钟寄存器操作函数

```

/**
 * \brief RTC 闹钟寄存器高位写函数
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 * \param[in] alarmh 闹钟寄存器高位
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rtc_alrh_set (amhw_zmf2x6_rtc_t *p_hw_rtc, uint16_t alarmh)
{
    p_hw_rtc->alrh = alarmh;
}

/**
 * \brief RTC 闹钟寄存器低位写函数
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 * \param[in] alarml 闹钟寄存器低位
 *
 * \return 无
 */
am_static_inline

```

```
void amhw_zmf2x6_rtc_alrl_set (amhw_zmf2x6_rtc_t *p_hw_rtc, uint16_t alarml)
{
    p_hw_rtc->alrl = alarml;
}
```

7. RCC_MSRH/RCC_MSRL（毫秒寄存器）

毫秒为高位和低位两个寄存器，操作方式一样，如程序清单 16.10 所示。

程序清单 16.10 毫秒寄存器操作函数

```
/**
 * \brief RTC 毫秒寄存器高位写函数
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 * \param[in] msh      毫秒寄存器高位
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rtc_msrh_set (amhw_zmf2x6_rtc_t *p_hw_rtc, uint16_t msh)
{
    p_hw_rtc->msrh = msh;
}

/**
 * \brief RTC 毫秒寄存器低位写函数
 *
 * \param[in] p_hw_rtc 指向 amhw_zmf2x6_rtc_t 结构的指针
 * \param[in] msl      毫秒寄存器低位
 *
 * \return 无
 */
am_static_inline
void amhw_zmf2x6_rtc_msrl_set (amhw_zmf2x6_rtc_t *p_hw_rtc, uint16_t msl)
{
    p_hw_rtc->msrl = msl;
}
```

至此，已完成 RTC 相关寄存器所有操作函数的编写。

16.2.3 添加指向寄存器块的指针

为了方便用户或驱动层使用，将 RTC 外设基地址定义为 RTC 外设结构体指针，后续使用硬件接口层时直接使用宏即可。在 AMetal 中，由于这些内容与具体型号的芯片相关，这

样的宏统一定义到 zmf2x6_periph_map.h 文件中，如程序清单 16.11 所示。

程序清单 16.11 外设中加入 RTC 结构体指针定义

```
/** \brief 16 位通用定时器 3 寄存器指针 */
#define ZMF2X6_TIM3      ((amhw_zmf2x6_tim_t      *)ZMF2X6_TIM3_BASE)

/** \brief 16 位通用定时器 14 寄存器指针 */
#define ZMF2X6_TIM14     ((amhw_zmf2x6_tim_t      *)ZMF2X6_TIM14_BASE)

/** \brief 16 位通用定时器 16 寄存器指针 */
#define ZMF2X6_TIM16     ((amhw_zmf2x6_tim_t      *)ZMF2X6_TIM16_BASE)

/** \brief 16 位通用定时器 17 寄存器指针 */
#define ZMF2X6_TIM17     ((amhw_zmf2x6_tim_t      *)ZMF2X6_TIM17_BASE)

/** \brief RTC 寄存器块指针 */
#define ZMF2X6_RTC       ((amhw_zmf2x6_rtc_t       *)ZMF2X6_RTC_BASE)

/** \brief 独立看门狗(IWDG)寄存器块指针 */
#define ZMF2X6_IWDG      ((amhw_zmf2x6_iwdg_t      *)ZMF2X6_IWDG_BASE)

/** \brief 窗口看门狗(WWDG)寄存器块指针 */
#define ZMF2X6_WWDG      ((amhw_zmf2x6_wwdg_t      *)ZMF2X6_WWDG_BASE)
```

后续用户或驱动层使用 HW 层接口时，可直接使用 ZMF2X6_RTC 作为参数。至此，整个 HW 层的开发工作就结束了，如果后续开发中发现缺少某些函数，还可以再添加。

16.3 驱动层开发

参考 5.3.1 章节，新建 am_zmf2x6_rtc.h 和 am_zmf2x6_rtc.c 驱动文件，搭建相应的文件框架，并按规则放到对应的路径下。

16.3.1 编写结构性程序

驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

定义 RTC 外设设备信息结构体如程序清单 16.12 所示，后续可以根据实际情况修改。

程序清单 16.12 RTC 设备信息结构体定义

```
/**
```

```

* \brief ZMF2X6 RTC 设备信息
*/
typedef struct am_zmf2x6_rtc_devinfo {
    uint32_t rtc_regbase;           /**< \brief RTC 寄存器基地址 */
    uint32_t pwr_regbase;           /**< \brief PWR 寄存器基地址 */
    uint32_t bkp_regbase;           /**< \brief BKP 寄存器基地址 */
    uint32_t rtc_clk_sour;          /**< \brief RTC 时钟源 */
    int      rtc_inum;              /**< \brief RTC 中断号 */
    void      (*pfn_plfm_init)(void); /**< \brief 平台初始化函数 */
    void      (*pfn_plfm_deinit)(void); /**< \brief 平台解初始化函数 */
} am_zmf2x6_rtc_devinfo_t;

```

● 设备结构体定义

由于 RTC 设备需要提供标准的 RTC 服务,包括时间设置、时间获取、中断回调等功能。其次,为了在后续使用中,可以通过设备访问到设备信息,也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 16.13 所示,后续如果需要使用到什么设备相关的全局变量,就应该定义在这里。

程序清单 16.13 RTC 设备结构体定义

```

/**
* \brief ZMF2X6 RTC 设备
*/
typedef struct am_zmf2x6_rtc_dev {
    am_rtc_serv_t      rtc_serv;      /**< \brief RTC 标准服务 */
    uint8_t            int_state;      /**< \brief 中断使能标志 */
    am_bool_t          rtc_continue;   /**< \brief RTC 是否继续运行 */
    am_pfnvoid_t        pfn_callback[3]; /**< \brief 回调函数 */
    void                *p_arg[3];     /**< \brief 回调函数参数 */
    const am_zmf2x6_rtc_devinfo_t *p_devinfo; /**< \brief RTC 设备信息 */
} am_zmf2x6_rtc_dev_t;

```

2. 搭建函数框架

函数主要分为两部分一部分是提供给用户使用的驱动层函数,一部分是提供给标准层的函数。

● 驱动函数

对于 RTC 驱动层需要提供初始化、解初始化函数之外,还包括其他功能性函数。

初始化函数的参数是设备指针和设备信息常量指针,返回值为时钟初始化的状态。示例代码如程序清单 16.14 所示。

程序清单 16.14 初始化解初始化函数定义

```

/**
* \brief 初始化 RTC
*/
am_rtc_handle_t am_zmf2x6_rtc_init (am_zmf2x6_rtc_dev_t *p_dev,

```

```

                                const am_zmf2x6_rtc_devinfo_t *p_devinfo)
{
    uint32_t          prl_val  = 0;
    uint32_t          rtc_clk  = 0;
    amhw_zmf2x6_rtc_t *p_hw_rtc = NULL;
    amhw_zmf2x6_pwr_t *p_hw_pwr = NULL;
    amhw_zmf2x6_bkp_t *p_hw_bkp = NULL;

    if ((NULL == p_dev) || (NULL == p_devinfo)) {
        return NULL;
    }

    return &p_dev->rtc_serv;
}

/**
 * \brief 解初始化 RTC
 */
void am_zmf2x6_rtc_deinit (am_rtc_handle_t handle)
{
    am_zmf2x6_rtc_dev_t *p_dev      = (am_zmf2x6_rtc_dev_t *)handle;
    amhw_zmf2x6_pwr_t    *p_hw_pwr   = NULL;

    if ((NULL == p_dev) || (NULL == p_dev->p_devinfo)) {
        return;
    }
}

```

同时为了用户能够使用，还需要将这两个函数的声明放入到相应的头文件中。示例代码如程序清单 16.15 所示。

程序清单 16.15 初始化和解初始化函数声明

```

/**
 * \brief 初始化 RTC
 *
 * \param[in] p_dev      指向 RTC 设备的指针
 * \param[in] p_devinfo 指向 RTC 设备信息的指针
 *
 * \return RTC 标准服务操作句柄，如果为 NULL，表明初始化失败
 */
am_rtc_handle_t am_zmf2x6_rtc_init (am_zmf2x6_rtc_dev_t          *p_dev,
                                    const am_zmf2x6_rtc_devinfo_t *p_devinfo);

/**
 * \brief 解初始化 RTC

```

```

*
* \param[in] handle 使用 am_zmf2x6_rtc_init() 获取到的 RTC 标准服务操作句柄
*
* \return 无
*/
void am_zmf2x6_rtc_deinit (am_rtc_handle_t handle);

```

有一些功能与 MCU 硬件相关，例如闹钟设置、中断回调函数类型，这些功能无法通过标准层接口使用，所以需要额外提供功能性接口。

RTC 部分功能性接口示例代码如程序清单 16.16 所示。

程序清单 16.16 其他功能性函数定义

```

/**
 * \brief 设置 RTC 闹钟时间
 */
am_err_t am_zmf2x6_rtc_alarm_set (am_rtc_handle_t handle, am_tm_t *p_tm)
{
    uint32_t sec = 0;

    if ((NULL == handle) || (NULL == p_tm)) {
        return -AM_EINVAL;
    }

    return AM_OK;
}

/**
 * \brief 设置 RTC 中断回调
 */
am_err_t am_zmf2x6_rtc_callback_set (am_rtc_handle_t handle,
                                     uint8_t          type,
                                     am_pfnvoid_t       pfn_callback,
                                     void                *p_arg)
{
    am_zmf2x6_rtc_dev_t *p_dev = NULL;

    if (NULL == handle) {
        return -AM_EINVAL;
    }

    return AM_OK;
}

/**
 * \brief RTC 中断使能

```

```

*/
am_err_t am_zmf2x6_rtc_int_enable (am_rtc_handle_t handle,
                                   uint8_t      type)
{
    am_zmf2x6_rtc_dev_t *p_dev    = NULL;
    amhw_zmf2x6_rtc_t   *p_hw_rtc = NULL;

    if (NULL == handle) {
        return -AM_EINVAL;
    }
    return AM_OK;
}

/**
 * \brief RTC 中断失能
 */
am_err_t am_zmf2x6_rtc_int_disable (am_rtc_handle_t handle,
                                     uint8_t      type)
{
    am_zmf2x6_rtc_dev_t *p_dev    = NULL;
    amhw_zmf2x6_rtc_t   *p_hw_rtc = NULL;

    if (NULL == handle) {
        return -AM_EINVAL;
    }
    return AM_OK;
}

```

- 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 2 个函数，根据标准接口层函数声明，我们需要在.c 文件中定义 2 个函数。示例代码如程序清单 16.17 所示。

程序清单 16.17 提供给标准层函数定义

```

/**
 * \brief 设置当前的细分时间
 */
am_local int __rtc_time_set (void *p_drv, am_tm_t *p_tm);

/**
 * \brief 获取当前的细分时间
 */
am_local int __rtc_time_get (void *p_drv, am_tm_t *p_tm);

/** \brief 适配 RTC 标准接口需要的函数 */
am_local am_const struct am_rtc_drv_funcs __g_rtc_drv_funcs = {

```

```

__rtc_time_set, /* 设置当前的细分时间 */
__rtc_time_get, /* 获取当前的细分时间 */
};
/**
 * \brief 获取当前的细分时间
 */
am_local int __rtc_time_get (void *p_drv, am_tm_t *p_tm)
{
    am_zmf2x6_rtc_dev_t *p_dev    = (am_zmf2x6_rtc_dev_t *)p_drv;
    amhw_zmf2x6_rtc_t    *p_hw_rtc = (amhw_zmf2x6_rtc_t  *)p_dev->p_devinfo->rtc_regbase;
    uint32_t              sec;

    if ((NULL == p_tm) || (NULL == p_tm)) {
        return -AM_EINVAL;
    }

    return AM_OK;
}

/**
 * \brief 设置当前的细分时间
 */
am_local int __rtc_time_set (void *p_drv, am_tm_t *p_tm)
{
    am_zmf2x6_rtc_dev_t *p_dev = (am_zmf2x6_rtc_dev_t *)p_drv;
    uint32_t              sec;

    if ((NULL == p_tm) || (NULL == p_tm)) {
        return -AM_EINVAL;
    }

    return AM_OK;
}

```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。

16.3.2 驱动函数实现

通过上面的函数框架搭建，共计 8 个函数。2 个面向标准层的驱动函数，1 个初始化函数，1 个解初始化函数、4 个面向用户的功能性函数，下面逐步实现。

1. am_zmf2x6_rtc_init 函数

该函数主要完成两个功能：

- a) 初始化设备中的标准服务，将成员赋值。
- b) 初始化硬件相关设置，如程序清单 16.18 所示。

程序清单 16.18 RTC 初始化函数


```

/**
 * \brief 初始化 RTC
 */
am_rtc_handle_t am_zmf2x6_rtc_init (am_zmf2x6_rtc_dev_t      *p_dev,
                                   const am_zmf2x6_rtc_devinfo_t *p_devinfo)
{
    uint32_t      prl_val  = 0;
    uint32_t      rtc_clk  = 0;
    amhw_zmf2x6_rtc_t *p_hw_rtc = NULL;
    amhw_zmf2x6_pwr_t *p_hw_pwr = NULL;
    amhw_zmf2x6_bkp_t *p_hw_bkp = NULL;

    if ((NULL == p_dev) || (NULL == p_devinfo)) {
        return NULL;
    }

    p_hw_rtc      = (amhw_zmf2x6_rtc_t *)p_devinfo->rtc_regbase;
    p_hw_pwr      = (amhw_zmf2x6_pwr_t *)p_devinfo->pwr_regbase;
    p_hw_bkp      = (amhw_zmf2x6_bkp_t *)p_devinfo->bkp_regbase;
    p_dev->rtc_serv.p_funcs = (struct am_rtc_drv_funcs *)&__g_rtc_drv_funcs;
    p_dev->rtc_serv.p_drv   = p_dev;
    p_dev->int_state        = 0;
    p_dev->rtc_continue     = AM_FALSE;
    p_dev->pfn_callback[0]  = NULL;
    p_dev->pfn_callback[1]  = NULL;
    p_dev->pfn_callback[2]  = NULL;
    p_dev->p_devinfo        = p_devinfo;

    if (p_devinfo->pfn_plfm_init) {
        p_devinfo->pfn_plfm_init();
    }

    amhw_zmf2x6_rcc_apb1_enable(AMHW_ZMF2X6_RCC_APB1_PWR); /* 使能电源时钟 */
    amhw_zmf2x6_pwr_bkp_access_enable(p_hw_pwr, 1);         /* 取消备份域的写保护 */

    /* 如果从备份寄存器中读出的值不符合预期，需要重新初始化 */
    if (0x5A5A != amhw_zmf2x6_bkp_dr_read(p_hw_bkp, 0)) {

        amhw_zmf2x6_rcc_bdc_r_bdrst_reset(); /* 备份区域软件复位 */
        amhw_zmf2x6_rcc_bdc_r_bdrst_reset_end(); /* 备份域软件复位结束 */

        /* 获取 RTC 时钟源，使能时钟并计算预分频值 */
        switch (p_dev->p_devinfo->rtc_clk_sour){

```

```

case AMHW_ZMF2X6_RTCCLK_LSE:

    /* 使能 LSE */
    amhw_zmf2x6_rcc_bdcr_lseon_enable();
    while (!amhw_zmf2x6_rcc_bdcr_lserdy_read());

    prl_val = 32768 - 1;
    break;
case AMHW_ZMF2X6_RTCCLK_LSI:
    /* 使能 LSI */
    amhw_zmf2x6_rcc_lsi_enable();
    while (!amhw_zmf2x6_rcc_lsirdy_read());
    prl_val = 40000 - 1;
    break;
case AMHW_ZMF2X6_RTCCLK_HSE_DIV128:
    /* 使能 HSE */
    amhw_zmf2x6_rcc_hseon_enable();
    while (!amhw_zmf2x6_rcc_hserdy_read());

    rtc_clk = am_clk_rate_get(CLK_HSEOSC) / 128;
    prl_val = rtc_clk - 1;
    break;

default:
    break;
}

/* RTC 时钟源选择 */
amhw_zmf2x6_rcc_bdcr_rtc_clk_set((amhw_zmf2x6_rtc_clk_src)p_devinfo->rtc_clk_sour);
/* RTC 时钟使能 */
amhw_zmf2x6_rcc_bdcr_rtc_enable();
amhw_zmf2x6_rtc_clr_status_clear(p_hw_rtc, AMHW_ZMF2X6_RTC_RSf);
__sync_wait(p_hw_rtc); /* 等待 RTC 寄存器同步 */
__operation_wait(p_hw_rtc); /* 等待 RTC 操作完成 */
amhw_zmf2x6_rtc_clr_status_clear(p_hw_rtc, AMHW_ZMF2X6_RTC_RSf);
amhw_zmf2x6_rtc_crl_cnf_enter(p_hw_rtc); /* 允许配置 RTC */
amhw_zmf2x6_rtc_prlh_div_write(p_hw_rtc, prl_val >> 16);
amhw_zmf2x6_rtc_prll_div_write(p_hw_rtc, prl_val & 0xffff);
amhw_zmf2x6_rtc_crl_cnf_out(p_hw_rtc); /* 配置更新 */
__operation_wait(p_hw_rtc); /* 等待 RTC 操作完成 */

amhw_zmf2x6_bkp_dr_write(ZMF2X6_BKP, 0, 0x5A5A);
p_dev->rtc_continue = AM_FALSE;
} else {
    if (AMHW_ZMF2X6_RTCCLK_LSI == p_dev->p_devinfo->rtc_clk_sour) {

```

```

    /* 使能 LSI */
    amhw_zmf2x6_rcc_lsi_enable();
    while (!amhw_zmf2x6_rcc_lsirdy_read());
} else if (AMHW_ZMF2X6_RTCCLK_HSE_DIV128 == p_dev->p_devinfo->rtc_clk_sour) {
    /* 使能 HSE */
    amhw_zmf2x6_rcc_hseon_enable();
    while (!amhw_zmf2x6_rcc_hserdy_read());
}
amhw_zmf2x6_rtc_clr_status_clear(p_hw_rtc, AMHW_ZMF2X6_RTC_RSF);
__sync_wait(p_hw_rtc); /* 等待 RTC 寄存器同步 */
__operation_wait(p_hw_rtc); /* 等待 RTC 操作完成 */
p_dev->rtc_continue = AM_TRUE;
}
/* 连接中断 */
am_int_connect(p_devinfo->rtc_inum, __rtc_sec_isr, p_dev);
am_int_disable(p_devinfo->rtc_inum);
return &p_dev->rtc_serv;
}

```

2. am_zmf2x6_rtc_deinit 解初始化函数

该函数用于禁能 RTC 外设，以及释放占用的内存空间。程序实现如程序清单 16.19 所示。

程序清单 16.19 RTC 解初始化函数

```

/**
 * \brief 解初始化 RTC
 */
void am_zmf2x6_rtc_deinit (am_rtc_handle_t handle)
{
    am_zmf2x6_rtc_dev_t *p_dev    = (am_zmf2x6_rtc_dev_t *)handle;
    amhw_zmf2x6_pwr_t    *p_hw_pwr    = NULL;
    if ((NULL == p_dev) || (NULL == p_dev->p_devinfo)) {
        return;
    }
    p_hw_pwr = (amhw_zmf2x6_pwr_t *)p_dev->p_devinfo->pwr_regbase;
    p_dev->pfn_callback[0] = NULL;
    p_dev->pfn_callback[1] = NULL;
    p_dev->pfn_callback[2] = NULL;
    p_dev->p_devinfo        = NULL;
    p_dev->rtc_serv.p_funcs = NULL;
    p_dev->rtc_serv.p_drv   = NULL;
    p_dev                    = NULL;
    amhw_zmf2x6_bkp_dr_write(ZMF2X6_BKP, 0, 0); /* 清除标记 */
    amhw_zmf2x6_rcc_apb1_disable(AMHW_ZMF2X6_RCC_APB1_PWR); /* 禁能电源时钟 */
    amhw_zmf2x6_pwr_bkp_access_enable(p_hw_pwr, 0); /* 备份域的写保护 */
}

```

```

amhw_zmf2x6_rcc_bdc_rtc_disable();                /* RTC 时钟失能 */
if (p_dev->p_devinfo->pfn_plfm_deinit) {
    p_dev->p_devinfo->pfn_plfm_deinit();
}
}

```

3. 面向标准层的驱动函数

RTC 驱动面向标准层的函数有两个，一个获取时间函数，一个设置时间函数。程序实现如程序清单 16.20 所示。

程序清单 16.20 面向标准层的驱动函数

```

/**
 * \brief 获取当前的细分时间
 */
am_local int __rtc_time_get (void *p_drv, am_tm_t *p_tm)
{
    am_zmf2x6_rtc_dev_t *p_dev = (am_zmf2x6_rtc_dev_t *)p_drv;
    amhw_zmf2x6_rtc_t *p_hw_rtc = (amhw_zmf2x6_rtc_t *)p_dev->p_devinfo->rtc_regbase;
    uint32_t sec;
    if ((NULL == p_tm) || (NULL == p_tm)) {
        return -AM_EINVAL;
    }
    __sync_wait(p_hw_rtc); /* 等待 RTC 寄存器同步 */
    if (AM_OK != am_zmf2x6_rtc_cnt_get(&p_dev->rtc_serv, &sec)) {
        return -AM_EINVAL;
    }
    /* 将秒转换成时间信息 */
    __sec2tm(sec, p_tm);
    return AM_OK;
}

/**
 * \brief 设置当前的细分时间
 */
am_local int __rtc_time_set (void *p_drv, am_tm_t *p_tm)
{
    am_zmf2x6_rtc_dev_t *p_dev = (am_zmf2x6_rtc_dev_t *)p_drv;
    uint32_t sec;
    if ((NULL == p_tm) || (NULL == p_tm)) {
        return -AM_EINVAL;
    }
    sec = mktime((struct tm *)p_tm);
    am_zmf2x6_rtc_cnt_set(&p_dev->rtc_serv, sec);
    return AM_OK;
}

```

4. 功能性函数

有些功能通过标准层接口无法实现，需要额外提供驱动接口，例如闹钟设置、中断使能、中断回调。部分功能性接口函数实现如程序清单 16.21 所示。

程序清单 16.21 部分功能性接口函数

```
/**
 * \brief 设置 RTC 闹钟时间
 */
am_err_t am_zmf2x6_rtc_alarm_set (am_rtc_handle_t handle, am_tm_t *p_tm)
{
    uint32_t sec = 0;
    if ((NULL == handle) || (NULL == p_tm)) {
        return -AM_EINVAL;
    }
    sec = mktime((struct tm *)p_tm);
    am_zmf2x6_rtc_alr_set(handle, sec);
    return AM_OK;
}

/**
 * \brief 设置 RTC 中断回调
 */
am_err_t am_zmf2x6_rtc_callback_set (am_rtc_handle_t handle,
                                     uint8_t          type,
                                     am_pfnvoid_t      pfn_callback,
                                     void              *p_arg)
{
    am_zmf2x6_rtc_dev_t *p_dev = NULL;
    if (NULL == handle) {
        return -AM_EINVAL;
    }
    p_dev = (am_zmf2x6_rtc_dev_t *)AM_CONTAINER_OF(handle, am_zmf2x6_rtc_dev_t, rtc_serv);
    if (AM_ZMF2X6_RTC_CALLBACK_SECOND == type) {
        p_dev->pfn_callback[0] = pfn_callback;
        p_dev->p_arg[0] = p_arg;
    } else if (AM_ZMF2X6_RTC_CALLBACK_ALARM == type) {
        p_dev->pfn_callback[1] = pfn_callback;
        p_dev->p_arg[1] = p_arg;
    } else if (AM_ZMF2X6_RTC_CALLBACK_OVER == type) {
        p_dev->pfn_callback[2] = pfn_callback;
        p_dev->p_arg[2] = p_arg;
    } else {
        return -AM_EINVAL;
    }
    return AM_OK;
}
```

```

/**
 * \brief RTC 中断使能
 */
am_err_t am_zmf2x6_rtc_int_enable (am_rtc_handle_t handle,
                                   uint8_t      type)
{
    am_zmf2x6_rtc_dev_t *p_dev    = NULL;
    amhw_zmf2x6_rtc_t   *p_hw_rtc = NULL;
    if (NULL == handle) {
        return -AM_EINVAL;
    }
    p_dev = (am_zmf2x6_rtc_dev_t *)AM_CONTAINER_OF(handle, am_zmf2x6_rtc_dev_t, rtc_serv);
    p_hw_rtc = (amhw_zmf2x6_rtc_t *)p_dev->p_devinfo->rtc_regbase;
    __operation_wait(p_hw_rtc); /* 等待 RTC 操作完成 */
    if (AM_ZMF2X6_RTC_CALLBACK_SECOND == type) {
        amhw_zmf2x6_rtc_crh_allow_int(p_hw_rtc, AMHW_ZMF2X6_RTC_SECIE);
        p_dev->int_state |= AM_ZMF2X6_RTC_CALLBACK_SECOND;
    } else if (AM_ZMF2X6_RTC_CALLBACK_ALARM == type) {
        amhw_zmf2x6_rtc_crh_allow_int(p_hw_rtc, AMHW_ZMF2X6_RTC_ALRIE);
        p_dev->int_state |= AM_ZMF2X6_RTC_CALLBACK_ALARM;
    } else if (AM_ZMF2X6_RTC_CALLBACK_OVER == type) {
        amhw_zmf2x6_rtc_crh_allow_int(p_hw_rtc, AMHW_ZMF2X6_RTC_OWIE);
        p_dev->int_state |= AM_ZMF2X6_RTC_CALLBACK_OVER;
    } else {
        return -AM_EINVAL;
    }
    am_int_enable(p_dev->p_devinfo->rtc_inum);
    return AM_OK;
}

/**
 * \brief RTC 中断失能
 */
am_err_t am_zmf2x6_rtc_int_disable (am_rtc_handle_t handle,
                                     uint8_t      type)
{
    am_zmf2x6_rtc_dev_t *p_dev    = NULL;
    amhw_zmf2x6_rtc_t   *p_hw_rtc = NULL;
    if (NULL == handle) {
        return -AM_EINVAL;
    }

    p_dev = (am_zmf2x6_rtc_dev_t *)AM_CONTAINER_OF(handle, am_zmf2x6_rtc_dev_t, rtc_serv);
    p_hw_rtc = (amhw_zmf2x6_rtc_t *)p_dev->p_devinfo->rtc_regbase;
    __operation_wait(p_hw_rtc); /* 等待 RTC 操作完成 */

```

```

if (AM_ZMF2X6_RTC_CALLBACK_SECOND == type) {
    amhw_zmf2x6_rtc_crh_forbid_int(p_hw_rtc, AMHW_ZMF2X6_RTC_SECIE);
    p_dev->int_state &= ~AM_ZMF2X6_RTC_CALLBACK_SECOND;
} else if (AM_ZMF2X6_RTC_CALLBACK_ALARM == type) {
    amhw_zmf2x6_rtc_crh_forbid_int(p_hw_rtc, AMHW_ZMF2X6_RTC_ALRIE);
    p_dev->int_state &= ~AM_ZMF2X6_RTC_CALLBACK_ALARM;
} else if (AM_ZMF2X6_RTC_CALLBACK_OVER == type) {
    amhw_zmf2x6_rtc_crh_forbid_int(p_hw_rtc, AMHW_ZMF2X6_RTC_OWIE);
    p_dev->int_state &= ~AM_ZMF2X6_RTC_CALLBACK_OVER;
} else {
    return -AM_EINVAL;
}
if (0 == p_dev->int_state) {
    am_int_disable(p_dev->p_devinfo->rtc_inum);
}
return AM_OK;
}

```

仔细观察会发现驱动函数内部调用了一些本地函数，比如 `_rtc_time_get` 函数中调用了 `__sync_wait`、`__sec2tm` 等函数，这些函数功能更为详细具体，也让标准接口的程序更为直观和易读。关于本地函数的实现，请直接查看源文件。

到现在 RTC 驱动的所有函数开发完毕，整个 RTC 的驱动层开发也就结束了。

16.4 用户配置文件编写

参考 5.4.1 章节新建用户配置文件，搭建文件的基本框架，准备配置文件内容的编写。

16.4.1 定义设备常量于设备信息常量

创建配置文件后，我们需要在里面加入设备变量，设备变量只需要定义，不用初始化。设备信息常量定义时初始化，默认填充的信息是一种常用的配置方式，RTC 设备信息配置如程序清单 16.22 所示。

程序清单 16.22 RTC 设备信息

```

#define RTC_CLK_SOUR  AMHW_ZMF2X6_RTCCLK_HSE_DIV128
/** \brief RTC 设备信息 */
const struct am_zmf2x6_rtc_devinfo __g_rtc_devinfo = {

    /** \brief RTC 设备基地址 */
    ZMF2X6_RTC_BASE,

    /** \brief 电源控制 PWR 基地址 */
    ZMF2X6_PWR_BASE,

    /** \brief 备份控制 BKP 基地址 */
    ZMF2X6_BKP_BASE,

```

```

/**<\brief RTC 中断号 */
INUM_RTC_BKP,

/** \brief RTC 设备时钟源 */
RTC_CLK_SOUR,

/** \brief 平台初始化函数 */
__zmf2x6_plfm_rtc_init,

/** \brief 平台去初始化函数 */
__zmf2x6_plfm_rtc_deinit
};

/** \brief RTC 设备 */
am_zmf2x6_rtc_dev_t __g_rtc_dev;

```

16.4.2 添加平台初始化与平台解初始化函数

RTC 外设正常运行，还需要取消备份域写保护、选用外部低速时钟需要提前使能等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数，如程序清单 16.23 所示。

程序清单 16.23 RTC 平台初始化和解初始化函数

```

/** \brief RTC 平台初始化 */
void __zmf2x6_plfm_rtc_init()
{
    amhw_zmf2x6_rcc_apb1_enable(AMHW_ZMF2X6_RCC_APB1_PWR); /* 使能电源时钟 */
    amhw_zmf2x6_pwr_bkp_access_enable(ZMF2X6_PWR,1);        /* 取消备份域的写保护 */
    amhw_zmf2x6_rcc_bdcr_bdrst_reset();                       /* 备份区域软件复位 */
    am_udelay(5);
    amhw_zmf2x6_rcc_bdcr_bdrst_reset_end();                   /* 备份域软件复位结束 */
    amhw_zmf2x6_rcc_bdcr_rtc_clk_set((amhw_zmf2x6_rtc_clk_src)RTC_CLK_SOUR);/* RTC 时钟源选择
为外部 RTC 时钟源 */
    am_mdelay(1);
    amhw_zmf2x6_rcc_bdcr_rtc_enable();                        /* RTC 时钟使能 */
}

/** 解除 RTC 平台初始化 */
void __zmf2x6_plfm_rtc_deinit(void)
{
    amhw_zmf2x6_rcc_apb1_disable(AMHW_ZMF2X6_RCC_APB1_PWR);/* 禁能电源时钟 */
    amhw_zmf2x6_pwr_bkp_access_enable(ZMF2X6_PWR,0);        /* 备份域的写保护 */
    amhw_zmf2x6_rcc_bdcr_rtc_disable();                      /* RTC 时钟禁能 */
}

```

接下来调用 RTC 初始化函数对实例进行初始化或解初始化，如程序清单 16.24 所示。

程序清单 16.24 实例初始化解初始化

```

/** \brief rtc 实例初始化，获得 rtc 标准服务句柄 */
am_rtc_handle_t am_zmf2x6_rtc_inst_init(void)
{
    return am_zmf2x6_rtc_init(&__g_rtc_dev, &__g_rtc_devinfo);
}

/**
 * \brief rtc 实例解初始化
 * \param[in] handle : rtc 句柄值
 */
void am_zmf2x6_rtc_inst_deinit(am_rtc_handle_t handle)
{
    am_zmf2x6_rtc_deinit(handle);
}

```

16.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在头文件中加入它们的声明，如程序清单 16.25 所示。

程序清单 16.25 实例初始化解初始化声明

```

/**
 * \brief RTC 实例初始化，获得 RTC 标准服务句柄
 *
 * \param 无
 *
 * \return RTC 标准服务句柄，若为 NULL，表明初始化失败
 */
am_rtc_handle_t am_zmf2x6_rtc_inst_init(void);

/**
 * \brief RTC 实例解初始化
 *
 * \param[in] handle 通过 am_zmf2x6_rtc_inst_init() 函数获得的 RTC 标准服务句柄
 *
 * \return 无
 */
void am_zmf2x6_rtc_inst_deinit(am_rtc_handle_t handle);

```

用户使用时，应该先调用实例初始化函数 `am_zmf2x6_rtc_inst_init` 函数初始化 RTC，并保存其返回的 `handle`，后续直接使用保存的 `handle` 作为标准接口的第一个参数使用标准接口函数即可。

至此，用户配置文件开发完毕。

16.5 例程文件开发

参考 5.5.1 章节新建用户例程文件，搭建文件的基本框架，准备 RTC 例程内容的编写。

16.5.1 例程内容开发

在前面所搭建的框架下，只需编写实际例程内容就可以。RTC 的例程，主要是初始化时设置一次时间，后续不断读取 RTC 的实时时间。

RTC 已有标准例程，只需调用实例初始化函数返回的 handle 作为参数传入即可，如程序清单 16.26 所示。

程序清单 16.26 RTC 标准例程

```
/** [src_zmf2x6_std_rtc] */

#include "ametal.h"
#include "am_vdebug.h"
#include "demo_std_entries.h"
#include "am_zmf2x6_inst_init.h"
#include "demo_zmf2x6_core_entries.h"

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_std_rtc_entry (void)
{
    AM_DBG_INFO("demo zmf2x6_core rtc entry!\r\n");

    demo_std_rtc_entry(am_zmf2x6_rtc_inst_init());
}
/** [src_zmf2x6_std_rtc] */

/* end of file */
```

例程开发完毕后，整个 RTC 外设的开发工作也就圆满结束了。

可以提交相关文件，以完成整个开发任务。提交文件时，不用提交工程文件，只需要提交开发的源文件和头文件即可。

第17章 通用外设驱动开发——CAN

17.1 标准层了解

CAN 标准层包含 1 个文件 am_can.h 文件，我们来分析一下 am_can.h 文件，在 ametal\interface 找到该文件并打开。

17.1.1 了解相关类型定义

打开 am_can.h 文件，可以看到该文件有多个结构体类型分别为：can 发送接收数据结构体、错误计数结构体、波特率配置结构体、滤波表配置结构体，CAN 标准驱动函数结构体，CAN 设备结构体，另外还定义了与 CAN 相关的大量宏定义。如程序清单 17.1 所示，下面对每个结构体做详细介绍。

程序清单 17.1 标准接口文件类型定义

```

/** \brief flags 报文信息的标志
 *
 * | 15:10 | 9:8 controller type |
 *
 * | 7:reseverd | 6:reseverd | 5:SEF | 4:SDF | 3:0 sendtype |
 *
 * bit7:reseverd 保留
 * bit6:reseverd 保留
 * bit5:SEF 帧格式 (1:extended 扩展帧 0:std 标准帧)
 * bit4:SDF 帧类型 (1:remote 远程帧 0:data 数据帧)
 * bit0~3:发送类型 (sendtype:见宏定义)
 */
typedef uint16_t am_can_flag_t;          /**< \brief CAN 标志 */
#define AM_CAN_NORMAL_SEND_FLAG      0X00  /**< \brief 正常发送 */
#define AM_CAN_ONCE_SEND_FLAG        0X01  /**< \brief 单次发送 */
#define AM_CAN_SELF_SEND_FLAG        0X02  /**< \brief 自发自收 */
#define AM_CAN_SELFONCE_SEND_FLAG    0X03  /**< \brief 单次自发自收 */
#define AM_CAN_REMOTE_FLAG           0X10  /**< \brief 远程帧标志位 */
#define AM_CAN_EXTERN_FLAG           0X20  /**< \brief 扩展帧标志位 */
#define AM_CAN_FD_BRS_FLAG            (1<<6) /**< \brief 使用 FD 部分数据波特率位 */
#define AM_CAN_FD_ESI_FLAG            (1<<7) /**< \brief 错误状态指示器（需使能 ESI 模式） */
#define AM_CAN_FD_CTRL_FLAG           0x0100 /**< \brief CANFD 标志位 */

/** @} */

/** \brief CAN 接收发送 信息 */
typedef struct am_can_message {
    uint32_t      id;          /**< \brief id 帧 ID */
    am_can_flag_t flags;       /**< \brief flags 标志位 */
    uint16_t      msglen;      /**< \brief msglen 报文长度 */
    uint8_t       msgdata[8];  /**< \brief msgdata 报文缓冲区 */
}

```

```

} am_can_message_t;

/** \brief 接收信息 */
typedef struct am_canfd_message {
    am_can_message_t can_msg;          /**< \brief CAN 信息
*/
    uint8_t msgdata[64-8];             /**< \brief CANFD 数据长度为 64B */
} am_canfd_message_t;

/**
 * \name CAN 错误类型
 * \anchor grp_am_can_err_t
 * @{
 */

typedef uint32_t am_can_err_t;          /**< \brief 错误类型 */
#define AM_CAN_NOERROR                  0x00    /**< \brief 无错误 */
#define AM_CAN_NOTINITIALIZED           0x01    /**< \brief 未初始化 */
#define AM_CAN_ILLEGAL_CHANNEL_NO      0x02    /**< \brief 非法通道号 */
#define AM_CAN_ILLEGAL_CONFIG          0x03    /**< \brief 非法配置 */
#define AM_CAN_ILLEGAL_DATA_LENGTH     0x04    /**< \brief 非法数据长度 */
#define AM_CAN_ILLEGAL_MASK_VALUE      0x05    /**< \brief 非法屏蔽码 */
#define AM_CAN_NO_AVAIL_CHANNELS        0x06    /**< \brief 通道不可用 */
#define AM_CAN_INVALID_PARAMETER        0x07    /**< \brief 无效参数 */
#define AM_CAN_ILLEGAL_OFFSET          0x08    /**< \brief 无效的偏置 */
#define AM_CAN_CANNOT_SET_ERRINT        0x09    /**< \brief 未使能错误中断 */
#define AM_CAN_CANNOT_SET_BOFFINT       0x10    /**< \brief 未使能总线关闭中断 */
#define AM_CAN_CANNOT_SET_WAKEUPINT     0x11    /**< \brief 未使能唤醒中断 */
#define AM_CAN_INCOMPATIBLE_TYPE        0x12    /**< \brief 类型不兼容 */
#define AM_CAN_BAUDRATE_ERROR           0x13    /**< \brief 波特率错误 */

/** @} */

/**
 * \name CAN 总线错误类型
 * \anchor grp_am_can_bus_err
 * @{
 */

typedef uint32_t am_can_bus_err_t;      /**< \brief CAN 总线错误类型 */
#define AM_CAN_BUS_ERR_NONE              0x00    /**< \brief 无错误 */
#define AM_CAN_BUS_ERR_BIT               0x01    /**< \brief 位错误 */

```

```

#define AM_CAN_BUS_ERR_ACK          0x02    /**< \brief 应答错误 */
#define AM_CAN_BUS_ERR_CRC          0x04    /**< \brief CRC 错误 */
#define AM_CAN_BUS_ERR_FORM         0x08    /**< \brief 格式错误 */
#define AM_CAN_BUS_ERR_STUFF         0x10    /**< \brief 填充错误 */
#define AM_CAN_BUS_ERR_UNKNOWN      0x20    /**< \brief 未知错误 */

/** @} */

/**
 * \name CAN 中断类型
 * \anchor grp_am_can_int_type
 * @{
 */

typedef uint32_t am_can_int_type_t;          /**< \brief 中断类型 */
#define AM_CAN_INT_NONE              0x00    /**< \brief 无类型 */
#define AM_CAN_INT_ERROR              0x01    /**< \brief 错误中断 */
#define AM_CAN_INT_BUS_OFF            0x02    /**< \brief 总线关闭中断 */
#define AM_CAN_INT_WAKE_UP            0x04    /**< \brief 唤醒中断 */
#define AM_CAN_INT_TX                 0x08    /**< \brief 发送中断 */
#define AM_CAN_INT_RX                 0x10    /**< \brief 接收中断 */
#define AM_CAN_INT_DATAOVER           0x20    /**< \brief 总线超载中断 */
#define AM_CAN_INT_WARN               0x40    /**< \brief 警告中断 */
#define AM_CAN_INT_ERROR_PASSIVE      0x80    /**< \brief 错误被动中断 */
#define AM_CAN_INT_ALL                0xffffffff /**< \brief 所有中断 */

/** @} */

/**
 * \name CAN 模式类型
 * \anchor grp_am_can_mode_type
 * @{
 */

typedef uint8_t am_can_mode_type_t;          /**< \brief CAN 模式类型 */
#define AM_CAN_MODE_NROMAL            0x00    /**< \brief 正常工作模式 */
#define AM_CAN_MODE_LISTEN_ONLY      0x01    /**< \brief 只听工作模式 */

/**
 * \name CAN 帧类型
 * \anchor grp_am_can_frame_type
 * @{
 */

```

```

typedef uint8_t am_can_frame_type_t;          /**< \brief CAN 帧类型 */
#define AM_CAN_FRAME_TYPE_STD                 0x01    /**< \brief 标准帧 */
#define AM_CAN_FRAME_TYPE_EXT                 0x02    /**< \brief 扩展帧 */

/**
 * \name CAN 帧格式
 * \anchor grp_am_can_frame_format_t
 * @{
 */

typedef uint8_t am_can_frame_format_t;        /**< \brief CAN 帧类型 */
#define AM_CAN_FRAME_FORMAT_NOCARE            0x00    /**< \brief 数据帧和远程帧都可接收 */
#define AM_CAN_FRAME_FORMAT_DATA              0x01    /**< \brief 数据帧 */
#define AM_CAN_FRAME_FORMAT_REMOTE            0x02    /**< \brief 远程帧 */

/** @} */

/** \brief 错误计数 */
typedef struct am_can_err_cnt {
    uint8_t rx_error_cnt;                      /**< \brief 接收错误计数器 */
    uint8_t tx_error_cnt;                      /**< \brief 发送错误计数器 */
} am_can_err_cnt_t;

/** \brief 波特率参数 */
typedef struct am_can_bps_param {
    uint8_t tesg1;                             /**< \brief tseg1 相位段 1 */
    uint8_t tesg2;                             /**< \brief tseg2 相位段 2 */
    uint8_t sjw;                               /**< \brief sjw 同步跳转宽度 */
    uint8_t smp;                               /**< \brief smp 采样模式 */
    uint16_t brp;                              /**< \brief brp 分频值 */
    uint8_t canfd_flag;                        /**< \brief CAN FD 波特率配置标记
                                                1: CAN 波特率 1: CAN FD 波特率 */
} am_can_bps_param_t;

/** \brief 滤波表配置 */
typedef struct am_can_filter {
    am_can_frame_type_t ide;                   /**< \brief 滤波器帧类型 */
    am_can_frame_format_t rtr;                 /**< \brief 滤波器帧格式 */

    uint32_t id[4];                           /**< \brief ID 验收码 */
    uint32_t mask[4];                         /**< \brief ID 掩码 */

```

```

} am_can_filter_t;

/**
 * \brief CAN reconfigure information .
 */
typedef struct am_can_cfg_info {
    am_can_mode_type_t    can_mode;          /**< \brief CAN 模式    */
} am_can_cfg_info_t;

/**
 * \brief CAN drive functions.
 */
struct am_can_drv_funcs {

    /** \brief CAN 启动 */
    am_can_err_t (*pfn_can_start)(void *p_drv);

    /** \brief CAN 复位（停止） */
    am_can_err_t (*pfn_can_reset)(void *p_drv);

    /** \brief CAN 休眠 */
    am_can_err_t (*pfn_can_sleep)(void *p_drv);

    /** \brief CAN 唤醒 */
    am_can_err_t (*pfn_can_wakeup)(void *p_drv);

    /** \brief 中断使能 */
    am_can_err_t (*pfn_can_int_enable)(void *p_drv, am_can_int_type_t int_mask);

    /** \brief 中断禁能*/
    am_can_err_t (*pfn_can_int_disable)(void *p_drv, am_can_int_type_t int_mask);

    /** \brief 模式设置 */
    am_can_err_t (*pfn_can_mode_set)(void *p_drv, am_can_mode_type_t mode);

    /** \brief 波特率设置 */
    am_can_err_t (*pfn_can_baudrate_set)(void *p_drv,
                                          am_can_bps_param_t *p_can_baudrate);

    /** \brief 波特率获取 */
    am_can_err_t (*pfn_can_baudrate_get)(void *p_drv,
                                          am_can_bps_param_t *p_can_baudrate);

```

```

/** \brief 获取错误计数 */
am_can_err_t (*pfn_can_err_cnt_get)(void          *p_drv,
                                         am_can_err_cnt_t *p_can_err_reg);

/** \brief 清除错误计数 */
am_can_err_t (*pfn_can_err_cnt_clr)(void *p_drv);

/** \brief 发送信息 */
am_can_err_t (*pfn_can_msg_send)(void *p_drv, am_can_message_t *p_txmsg);

/** \brief 接收信息 */
am_can_err_t (*pfn_can_msg_rcv)(void *p_drv, am_can_message_t *p_rxmsg);

/** \brief 停止发送信息 */
am_can_err_t (*pfn_can_stop_msg_snd)(void *p_drv);

/** \brief 设置滤波函数 */
am_can_err_t (*pfn_can_filter_tab_set)(void      *p_drv,
                                         uint8_t *p_filterbuff,
                                         size_t   lenth);

/** \brief 获取滤波函数*/
am_can_err_t (*pfn_can_filter_tab_get)(void      *p_drv,
                                         uint8_t *p_filterbuff,
                                         size_t   *p_lenth);

/** \brief CAN 状态 */
am_can_err_t (*pfn_can_status_get)(void          *p_drv,
                                     am_can_int_type_t *p_int_type,
                                     am_can_bus_err_t  *p_bus_err);

/** \brief 中断连接 */
am_can_err_t (*pfn_can_connect)(void          *p_drv,
                                 am_pfnvoid_t   pfn_isr,
                                 void            *p_arg);

/** \brief 删除中断连接 */
am_can_err_t (*pfn_can_disconnect)(void          *p_drv,
                                    am_pfnvoid_t   pfn_isr,
                                    void            *p_arg);

/** \brief 中断回调函数的连接 */
am_can_err_t (*pfn_can_intcb_connect)(void          *p_drv,
                                       am_can_int_type_t inttype,
                                       am_pfnvoid_t     pfn_callback,

```



```

                                void                *p_arg);

/** \brief 删除中断回调函数的连接 */
am_can_err_t (*pfn_can_intcb_disconnect)(void                *p_drv,
                                           am_can_int_type_t  inttype);

/** \brief 设置滤波函数(扩展) */
am_can_err_t (*pfn_can_filter_tab_ext_set)(void                *p_drv,
                                           am_can_filter_t *p_filterbuff,
                                           size_t            lenth);

/** \brief 获取滤波函数(扩展)*/
am_can_err_t (*pfn_can_filter_tab_ext_get)(void                *p_drv,
                                           am_can_filter_t *p_filterbuff,
                                           size_t            *p_lenth);
};

/**
 * \brief CAN 设备.
 */
typedef struct am_can_serv {

    struct am_can_drv_funcs  *p_drv_funcs;  /**< \brief 驱动函数 */
    void                    *p_drv;        /**< \brief 驱动设备 */

} am_can_serv_t;

/** \brief handle 标准服务 */
typedef am_can_serv_t *am_can_handle_t;

```

1. am_can_message_t 类型

用于描述 CAN 进行一次传输的相关信息，包含帧 ID 号、CAN 传输标志位、报文长度、报文缓冲区。

2. am_canfd_message_t 类型

用于描述 CANFD 进行一次传输的相关信息，它是 am_can_message_t 的扩展，即拥有更多的标志位和数据量的选择。

3. am_can_err_cnt_t 结构体

用于描述 CAN 发送接收错误计数的计数值。

4. am_can_bps_param_t 结构体

该结构体用于描述 CAN 总线波特率的配置。包含 tseg1、tesg2、sjw、smp、时钟分频值、以及 CANFD 波特率配置标记（波特率大于 1M），通过对该结构体进行配置，调用标准波特率配置函数，可实现不同波特率和采样点的配置。

5. am_can_filter_t 结构体

该结构体用于描述 CAN 模块的滤波表，用于控制 CAN 模块能够接收到的信息，其元

素包含滤波器帧类型、滤波器帧格式、ID 验收码、ID 掩码

6. am_can_drv_funcs

这个结构体表明了驱动层应该提供的函数。从结构体定义可以看出，驱动层需要实现 21 个函数：

- CAN 启动；
- CAN 复位（停止）；
- CAN 休眠；
- CAN 唤醒；
- 中断使能；
- 中断禁能；
- 模式设置；
- 波特率设置；
- 波特率获取；
- 获取错误计数；
- 清除错误计数；
- 发送信息；
- 接收信息；
- 停止发送信息；
- CAN 状态获取；
- 中断连接；
- 删除中断连接；
- 中断回调函数的连接；
- 删除中断回调函数的连接；
- 设置滤波函数(扩展)；
- 获取滤波函数(扩展)。

因此，在驱动层编写函数时，应该向标准接口层提供这 21 个函数。

这些函数的有一个共同点，就是第一个参数都是 void 型的指针，也就是说，驱动层提供的函数，标准接口层使用时传递的第一个参数并无确定类型，标准接口层不关心第一个参数的具体类型，使用时只负责传递就可以了。

这个标准的 CAN 服务就包含了需要驱动层为标准接口层提供的全部内容。驱动层只需要定义这样一个结构体变量，然后赋好相应的值，再以某种方式提供给标准接口层即可。

● p_funcs 成员赋值

这是一个指向 struct am_can_drv_funcs 结构体的指针，我们只需要在驱动层定义一个 struct am_can_drv_funcs 结构体变量，并将该结构体成员初始化赋值为需要驱动层提供的那几个函数地址。如程序清单 17.2 所示。然后，将 __g_can_drvfuncs 的地址赋值给 p_funcs 成员即可。值得一提的是我们并不一定要把所有函数都实现，实现其中主要部分也是能够使程序运行正常。

程序清单 17.2 struct am_can_drv_funcs 结构体变量

```

struct am_can_drv_funcs __zmf2x6_can_dev_func = {
    __can_zmf2x6_start,
    __can_zmf2x6_reset,
    __can_zmf2x6_sleep,
    __can_zmf2x6_wakeup,
    __can_zmf2x6_int_enable,
    __can_zmf2x6_int_disable,
    __can_zmf2x6_mode_set,
    __can_zmf2x6_baudrate_set,
    __can_zmf2x6_baudrate_get,
    __can_zmf2x6_err_cnt_get,
    NULL,
    __can_zmf2x6_msg_send,
    __can_zmf2x6_msg_recv,
    __can_zmf2x6_stop_msg_snd,
    NULL,
    NULL,
    __can_zmf2x6_status_get,
    __can_zmf2x6_connect,
    __can_zmf2x6_disconnect,
    __can_zmf2x6_intcb_connect,
    __can_zmf2x6_intcb_disconnect,
    __can_zmf2x6_filter_tab_ext_set,
    __can_zmf2x6_filter_tab_ext_get,
};

```

● p_drv 成员赋值

这个变量是 void 型的指针，标准接口层不需要知道其具体类型。这个完全由驱动层决定赋值内容，标准接口层在调用驱动层提供的函数时，会将该变量作为函数的第一个参数传递。

7. am_can_serv_t

该结构体用于描述一个 CAN 设备，其主要成员是 am_can_drv_funcs 类型以及一个 void *类型，其中 void *类型一般会用于存放对应芯片 CAN 外设的设备结构体

8. am_can_handle_t 的定义

这个是一个标准的操作句柄，本质上就是 struct am_can_serv 指针类型，驱动层的初始化函数(am_zmf2x6_can_init)需要返回该类型的一个 handle 给用户。这样，用户获取到 handle，就可以在需要使用标准接口层函数时，将该 handle 传递给标准接口层了。标准接口就可以通过该 handle（struct am_can_serv 结构体指针）访问到驱动层提供的函数了。

17.1.2 了解标准接口函数

在 am_can.h 文件相关的结构体、类型定义后，就是具体函数的实现，如程序清单 17.3 所示。

程序清单 17.3 标准函数实现

```

/**
 * \brief CAN 启动.
 *
 * \param[in] handle : CAN 标准服务 handle.
 *
 * \retval  AM_CAN_NOERROR          : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_start (am_can_handle_t handle)
{
    return handle->p_drv_funcs->pfn_can_start(handle->p_drv);
}

/**
 * \brief CAN 复位.
 *
 * \param[in] handle : CAN 标准服务 handle.
 *
 * \retval  AM_CAN_NOERROR          : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_reset (am_can_handle_t handle)
{
    return handle->p_drv_funcs->pfn_can_reset(handle->p_drv);
}

/**
 * \brief CAN 睡眠.
 *
 * \param[in] handle : CAN 标准服务 handle.
 *
 * \retval  AM_CAN_NOERROR          : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_sleep (am_can_handle_t handle)
{
    return handle->p_drv_funcs->pfn_can_sleep(handle->p_drv);
}

/**
 * \brief CAN 唤醒.

```

```

*
* \param[in] handle : CAN 标准服务 handle.
*
* \retval AM_CAN_NOERROR          : 成功.
* \retval -AM_CAN_INVALID_PARAMETER : 参数错误
*/
am_static_inline
am_can_err_t am_can_wakeup (am_can_handle_t handle)
{
    return handle->p_drv_funcs->pfn_can_wakeup(handle->p_drv);
}

/**
* \brief CAN 中断使能.
*
* \param[in] handle : CAN 标准服务 handle.
* \param[in] mask    : 中断类型
*
* \retval AM_CAN_NOERROR          : 成功.
* \retval -AM_CAN_INVALID_PARAMETER : 参数错误
*/
am_static_inline
am_can_err_t am_can_int_enable (am_can_handle_t handle, am_can_int_type_t mask)
{
    return handle->p_drv_funcs->pfn_can_int_enable(handle->p_drv, mask);
}

/**
* \brief CAN 中断失能.
*
* \param[in] handle : CAN 标准服务 handle.
* \param[in] mask    : 中断类型
*
* \retval AM_CAN_NOERROR          : 成功.
* \retval -AM_CAN_INVALID_PARAMETER : 参数错误
*/
am_static_inline
am_can_err_t am_can_int_disable (am_can_handle_t handle, am_can_int_type_t mask)
{
    return handle->p_drv_funcs->pfn_can_int_disable(handle->p_drv, mask);
}

/**
* \brief CAN 设置模式.

```

```

*
* \param[in] handle : CAN 标准服务 handle.
* \param[in] mode   : 工作模式
*
* \retval  AM_CAN_NOERROR           : 成功.
* \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
*/
am_static_inline
am_can_err_t am_can_mode_set (am_can_handle_t   handle,
                              am_can_mode_type_t mode)
{
    return handle->p_drv_funcs->pfn_can_mode_set(handle->p_drv, mode);
}

/**
* \brief CAN 设置波特率.
*
* \param[in] handle           : CAN 标准服务 handle.
* \param[in] p_can_baudrate   : 波特率
*
* \retval  AM_CAN_NOERROR           : 成功.
* \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
*/
am_static_inline
am_can_err_t am_can_baudrate_set (am_can_handle_t   handle,
                                  struct am_can_bps_param *p_can_baudrate)
{
    return handle->p_drv_funcs->pfn_can_baudrate_set(handle->p_drv,
                                                    p_can_baudrate);
}

/**
* \brief CAN 获取波特率.
*
* \param[in] handle           : CAN 标准服务 handle.
* \param[out] p_can_baudrate  : 波特率
*
* \retval  AM_CAN_NOERROR           : 成功.
* \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
*/
am_static_inline
am_can_err_t am_can_baudrate_get (am_can_handle_t   handle,
                                  struct am_can_bps_param *p_can_baudrate)
{

```

```

        return handle->p_drv_funcs->pfn_can_baudrate_get(handle->p_drv,
                                                         p_can_baudrate);
    }

/**
 * \brief CAN 获取错误计数
 *
 * \param[in] handle      : CAN 标准服务 handle.
 * \param[out] p_can_err_cnt : 错误计数
 *
 * \retval  AM_CAN_NOERROR      : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_err_cnt_get (am_can_handle_t handle,
                                 struct am_can_err_cnt *p_can_err_cnt)
{
    return handle->p_drv_funcs->pfn_can_err_cnt_get(handle->p_drv,
                                                    p_can_err_cnt);
}

/**
 * \brief CAN 清除错误计数
 *
 * \param[in] handle : CAN 标准服务 handle.
 *
 * \retval  AM_CAN_NOERROR      : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_err_cnt_clr (am_can_handle_t handle)
{
    return handle->p_drv_funcs->pfn_can_err_cnt_clr(handle->p_drv);
}

/**
 * \brief CAN 发送信息.
 *
 * \param[in] handle : CAN 标准服务 handle.
 * \param[in] p_txmsg : 发送的消息
 *
 * \retval  AM_CAN_NOERROR      : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
 */

```

```

am_static_inline
am_can_err_t am_can_msg_send(am_can_handle_t handle, am_can_message_t *p_txmsg)
{
    return handle->p_drv_funcs->pfn_can_msg_send(handle->p_drv, p_txmsg);
}

/**
 * \brief CAN 接收消息
 *
 * \param[in] handle    : CAN 标准服务 handle.
 * \param[out] p_rxmsg : 接收的消息
 *
 * \retval  AM_CAN_NOERROR          : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_msg_rcv(am_can_handle_t handle, am_can_message_t *p_rxmsg)
{
    return handle->p_drv_funcs->pfn_can_msg_rcv(handle->p_drv, p_rxmsg);
}

/**
 * \brief CAN 停止发送
 *
 * \param[in] handle    : CAN 标准服务 handle.
 *
 * \retval  AM_CAN_NOERROR          : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_msg_stop_send(am_can_handle_t handle)
{
    return handle->p_drv_funcs->pfn_can_stop_msg_snd(handle->p_drv);
}

/**
 * \brief CAN 设置滤波
 *
 * \param[in] handle    : CAN 标准服务 handle.
 * \param[in] p_filterbuff : 滤波表首地址
 * \param[in] lenth      : 滤波表长度
 *
 * \retval  AM_CAN_NOERROR          : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误

```



```

*/
am_static_inline
am_can_err_t am_can_filter_tab_set (am_can_handle_t handle,
                                   uint8_t      *p_filterbuff,
                                   size_t      lenth)
{
    return handle->p_drv_funcs->pfn_can_filter_tab_set(handle->p_drv,
                                                    p_filterbuff,
                                                    lenth);
}

/**
 * \brief CAN 设置滤波(扩展)
 *
 * \param[in] handle : CAN 标准服务 handle.
 * \param[in] p_filterbuff : 滤波表 详见 am_can_filter_t
 * \param[in] lenth : 传输的 am_can_filter_t 的个数
 *
 * \retval AM_CAN_NOERROR : 成功.
 * \retval -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_filter_tab_ext_set (am_can_handle_t handle,
                                       am_can_filter_t *p_filterbuff,
                                       size_t lenth)
{
    return handle->p_drv_funcs->pfn_can_filter_tab_ext_set(handle->p_drv,
                                                         p_filterbuff,
                                                         lenth);
}

/**
 * \brief CAN 获取滤波表设置
 *
 * \param[in] handle : CAN 标准服务 handle.
 * \param[out] p_filterbuff : 滤波表首地址
 * \param[out] p_lenth : 滤波表长度
 *
 * \retval AM_CAN_NOERROR : 成功.
 * \retval -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_filter_tab_get (am_can_handle_t handle,
                                   uint8_t      *p_filterbuff,

```

```

                                size_t      *p_lenth)
{
    return handle->p_drv_funcs->pfn_can_filter_tab_get(handle->p_drv,
                                                    p_filterbuff,
                                                    p_lenth);
}

/**
 * \brief CAN 获取滤波表设置(扩展)
 *
 * \param[in]  handle      : CAN 标准服务 handle.
 * \param[out] p_filterbuff : 滤波表 详见 am_can_filter_t
 * \param[out] p_lenth     : 设置的 am_can_filter_t 个数
 *
 * \retval  AM_CAN_NOERROR      : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_filter_tab_ext_get (am_can_handle_t  handle,
                                       am_can_filter_t *p_filterbuff,
                                       size_t      *p_lenth)
{
    return handle->p_drv_funcs->pfn_can_filter_tab_ext_get(handle->p_drv,
                                                         p_filterbuff,
                                                         p_lenth);
}

/**
 * \brief CAN 获取状态
 *
 * \param[in] handle      : CAN 标准服务 handle.
 * \param[out] p_int_type : 中断类型.
 * \param[out] p_bus_err  : 总线错误类型.
 *
 * \retval  AM_CAN_NOERROR      : 成功.
 * \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_status_get (am_can_handle_t  handle,
                               am_can_int_type_t *p_int_type,
                               am_can_bus_err_t  *p_bus_err)
{
    return handle->p_drv_funcs->pfn_can_status_get(handle->p_drv,
                                                  p_int_type,

```

```

                                p_bus_err);
}

/**
 * \brief CAN 中断连接
 *
 * \param[in] handle : CAN 标准服务 handle.
 * \param[in] pfn_isr : CAN 中断服务函数（NULL 时则使用内部中断服务）。
 * \param[in] p_arg : CAN 中断服务函数参数.
 *
 * \retval AM_CAN_NOERROR : 成功.
 * \retval -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_connect (am_can_handle_t handle,
                             am_pfnvoid_t pfn_isr,
                             void *p_arg)
{
    return handle->p_drv_funcs->pfn_can_connect(handle->p_drv, pfn_isr, p_arg);
}

/**
 * \brief CAN 删除中断连接
 *
 * \param[in] handle : CAN 标准服务 handle.
 * \param[in] pfn_isr : CAN 中断服务函数.
 * \param[in] p_arg : CAN 中断服务函数参数.
 *
 * \retval AM_CAN_NOERROR : 成功.
 * \retval -AM_CAN_INVALID_PARAMETER : 参数错误
 */
am_static_inline
am_can_err_t am_can_disconnect (am_can_handle_t handle,
                                 am_pfnvoid_t pfn_isr,
                                 void *p_arg)
{
    return handle->p_drv_funcs->pfn_can_disconnect(handle->p_drv,
                                                    pfn_isr,
                                                    p_arg);
}

/**
 * \brief CAN 注册中断回调函数
 *

```

```

* \param[in] handle          : CAN 标准服务 handle.
* \param[in] inttype         : 中断类型
* \param[in] pfn_callback    : 中断回调函数
* \param[in] p_arg           : 回调函数参数
*
* \retval  AM_CAN_NOERROR      : 成功.
* \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
* \retval  -AM_CAN_ILLEGAL_MASK_VALUE : 中断类型不合法
*/
am_static_inline
am_can_err_t am_can_intcb_connect (am_can_handle_t    handle,
                                   am_can_int_type_t   inttype,
                                   am_pfnvoid_t        pfn_callback,
                                   void                *p_arg)
{
    return handle->p_drv_funcs->pfn_can_intcb_connect(handle->p_drv,
                                                    inttype,
                                                    pfn_callback, p_arg);
}

/**
* \brief CAN 解除中断回调函数的注册
*
* \param[in] handle          : CAN 标准服务 handle.
* \param[in] inttype         : 中断类型
*
* \retval  AM_CAN_NOERROR      : 成功.
* \retval  -AM_CAN_INVALID_PARAMETER : 参数错误
* \retval  -AM_CAN_ILLEGAL_MASK_VALUE : 中断类型不合法
*/
am_static_inline
am_can_err_t am_can_intcb_disconnect (am_can_handle_t    handle,
                                       am_can_int_type_t   inttype)
{
    return handle->p_drv_funcs->pfn_can_intcb_disconnect(handle->p_drv,
                                                         inttype);
}

```

相关函数全都是用内联函数就搞定了，因为它本身就没做什么事情，只是简单的调用了一下驱动层提供的函数而已。可以看到，在调用驱动层提供的函数时，就是将 `struct am_can_serv` 结构体中的 `p_drv` 成员作为第一个参数传递。

到此我们就分析完了 CAN 外设标准层文件，清楚了驱动层需要为标准接口层提供哪些内容。

17.2 HW 层开发

通过上一节的内容，我们已经知道了标准层实现的函数的功能，以及需要驱动层提供的函数，但驱动层并不是凌空搭建的，驱动层介乎与 HW 层和标准接口层之间，简单来说就是调用 HW 层的函数来提供给驱动层，所以在写驱动层前，我们先来完成 HW 层。

现在我们打开 ZMF2x6 芯片用户手册，先了解一下该芯片 CAN 外设，下面对照手册完成 CAN 外设的 HW 层的开发。

17.2.1 添加寄存器列表

HW 层就是与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读该外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMF2x6 用户手册的 CAN 章节，可以看到，CAN 章节部分，共有 31 个寄存器，由于其中涉及到两种状态，即 basicCAN 以及 peliCAN，它们共用了一些地址。我们在这里仅实现 peliCAN 功能，因此与 basicCAN 相关功能寄存器可以忽略。根据 peliCAN 寄存器功能进行分类简单描述如下：

1. 主要配置相关

- CAN_MODE（模式寄存器）：涉及到 CAN 的模式配置，即只听模式、正常模式、自检测模式、验收滤波器模式；
- CAN_CMRR（命令寄存器）：涉及到 CAN 的相关命令，包含清空接收 fifo 数据、自接收请求、数据溢出清除、接收缓冲区释放；
- CAN_SR（状态寄存器）：包含 CAN 的运行状态，总线状态、错误状态、发送状态、接收状态、发送完成状态、发送缓冲区状态、数据溢出状态、接收缓冲区状态；
- CAN_IR（中断寄存器）：涉及到 CAN 模块的中断状态，其中中断种类有总线错误中断、仲裁丢失中断、错误消极中断、数据溢出中断、错误中断、发送中断、接收中断；
- CAN_IER（中断使能寄存器）：由于使能上述中断种类。

2. 错误检测相关

- CAN_ALC（仲裁丢失捕捉寄存器）：用于捕捉仲裁丢失的位置；
- CAN_ECC（错误代码捕捉寄存器）：用于判断发生错误时错误的位置；
- CAN_EWLR（错误报警限制寄存器）：用于配置错误计数器的报警值；
- CAN_RXERR（接收错误寄存器 1）：记录接收错误次数；
- CAN_TXERR（发送错误寄存器 1）：记录发送错误次数。

3. 波特率采样点相关：

- CAN_BTR（总线定时寄存器）：涉及到波特率、采样点的设置，即 tseg1、tesg2、sjw、smp 的配置；
- CAN_CDR（时钟分频寄存器）涉及到时钟分频值的配置：。

4. 传输相关

- CAN_SFF（发送帧信息寄存器）：涉及到 CAN 发送帧结构的相关信息配置；
- CAN_TXID（发送 ID 寄存器）：用于配置发送的 ID；
- CAN_TXDATA（发送数据寄存器）：用于配置发送的数据。

5. 滤波器相关

- CAN_AFM（滤波模式寄存器）：用于配置各滤波器的模式即单验收滤波或双验收滤波；
- CAN_FGA（滤波组使能寄存器）：用于使能滤波器；
- GROUPx_ACR（验收代码寄存器组）：配置可通过的 ID 号；
- GROUPx_AMR（验收屏蔽寄存器组）：配置可通过的 ID 掩码号。

到此 CAN 的各个寄存器及其功能基本阅读完毕，我们也大致了解了各个寄存器功能以及要配置的各个功能，其他寄存器该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，如程序清单 17.4 所示。

程序清单 17.4 CAN 寄存器结构体定义

```

/* \brief 发送接收邮箱寄存器 */
typedef struct amhw_zmf2x6_can_trx {
    __IO uint32_t sff;          /**< \brief 0x40 发送帧信息寄存器 */
    __IO uint32_t txid[2];      /**< \brief 0x44 发送识别码寄存器 */
    __IO uint32_t txdata[10];   /**< \brief 0x4c~0x70 扩展帧 ID 需要占用前两个 txdata */
}amhw_zmf2x6_can_trx_t;

/**
 * \brief ZMF2X6 CAN 寄存器块结构体
 */
typedef struct amhw_zmf2x6_can {

    __IO uint32_t mcr;          /**< \brief 0x00 模式寄存器 */
    __IO uint32_t cmd;          /**< \brief 0x04 命令寄存器 */
    __IO uint32_t msr;          /**< \brief 0x08 主状态寄存器 */
    __IO uint32_t ir;           /**< \brief 0x0c 中断寄存器 */
    __IO uint32_t ier;          /**< \brief 0x10 中断使能寄存器 */
    uint32_t reserve0;

    __IO uint32_t btr0;         /**< \brief 0x18 位时序使能寄存器 */
    __IO uint32_t btr1;         /**< \brief 0x1c 位时序使能寄存器 */
    uint32_t reserve1[3];

    __IO uint32_t alc;          /**< \brief 0x2c 位时序使能寄存器 */
    __IO uint32_t ecc;          /**< \brief 0x30 位时序使能寄存器 */
    __IO uint32_t ewlr;         /**< \brief 0x34 位时序使能寄存器 */
    __IO uint32_t rxerr;        /**< \brief 0x38 位时序使能寄存器 */
    __IO uint32_t txerr;        /**< \brief 0x3c 位时序使能寄存器 */

    /**< \brief 0x40...0x70 发送接收邮箱寄存器 */
    __IO amhw_zmf2x6_can_trx_t tx_rx_mail;

    uint32_t reserve2[2];       /**< \brief RMC RBSA */

```

```

__IO uint32_t cdr;          /**< \brief 0x7c 时钟分频寄存器 */

}amhw_zmf2x6_can_t;

/* \brief 过滤器组寄存器 */
typedef struct amhw_zmf2x6_can_firx {
    __IO uint32_t acr[4];    /**< \brief 过滤器组寄存器 1 */
    __IO uint32_t amr[4];    /**< \brief 过滤器组寄存器 2 */
}amhw_zmf2x6_canflt_group_t;

/**
 * \brief ZMF2X6 CAN 寄存器块结构体
 */
typedef struct amhw_zmf2x6_canflt {
    __IO amhw_zmf2x6_canflt_group_t firx0;
    uint32_t reserve1[8];
    __IO uint32_t afm[3];    /**< \brief 0x20c 过滤器位宽寄存器 fs_1r */
    __IO uint32_t fga[3];    /**< \brief 0x21c 过滤器激活寄存器 fa_1r */
    /**< \brief 0x98...0x2d8 过滤器组 1~19 寄存器 */
    __IO amhw_zmf2x6_canflt_group_t firx[19];
}amhw_zmf2x6_canflt_t;

```

这个小节我们定义了 CAN 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

17.2.2 编写寄存器的相关函数

1. CAN_MODE（模式寄存器）

通过手册，我们了解到该寄存器有含义的寄存器共有 4 个，如表 17.1 所示。

表 17.1 CAN_MODE 寄存器分析

位	标识符	含义
4	SLP	睡眠模式
3	AFM	设置验收滤波器模式（16 位长度或 32）
2	STM	置位为自检测模式，其它为正常模式
1	LOM	置位为只听模式，其它为正常模式
0	RM	置位为复位模式，其它为正常模式

前三位寄存器均与 CAN 的运行模式有关，通过它们我们可以编写出四个函数，即 CAN 运行模式设置函数，CAN 工作模式选择函数，CAN 进入睡眠模式，唤醒 CAN。

程序清单 17.5 为 CAN 运行模式设置函数。用于控制 CAN 运行模式为初始化模式、正常模式以及睡眠模式。

程序清单 17.5 CAN 运行模式设置函数

```

/**
 * \brief CAN 运行模式设置
 *
 * \param[in] p_hw_can      : 指向 CAN 寄存器块的指针
 * \param[in] operating_mode : 指向 amhw_zmf2x6_can_operating_type_t 类型
 *
 * \return  CAN 的模式设置状态是否成功  0  1
 */
uint8_t amhw_zmf2x6_can_operating_mode_request (amhw_zmf2x6_can_t      *p_hw_can,
                                                amhw_zmf2x6_can_mode_t mode)
{
    uint8_t status = CAN_DISABLE;

    if (mode == AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT) {

        /* 请求初始化 */
        p_hw_can->mcr = (uint32_t)((p_hw_can->mcr & (uint32_t)
                                   (~((uint32_t)AMHW_ZMF2X6_CAN_MCR_SLEEP))) |
                                AMHW_ZMF2X6_CAN_MCR_INRQ);

        /* 超时或 ACK 后确认状态 */
        if ((p_hw_can->mcr & (uint32_t)AMHW_ZMF2X6_CAN_MCR_INRQ) != 0) {
            status = CAN_ENABLE;
        } else {
            status = CAN_DISABLE;
        }
    } else if (mode == AMHW_ZMF2X6_CAN_OPERATING_NORMAL) {

        /* 请求离开初始化模式或睡眠模式 进入正常模式 */
        p_hw_can->mcr &= ~AMHW_ZMF2X6_CAN_MCR_INRQ;

        if ((p_hw_can->mcr & (uint32_t)AMHW_ZMF2X6_CAN_MCR_INRQ) == 0) {
            status = CAN_ENABLE;
        } else {
            status = CAN_DISABLE;
        }
    } else if (mode == AMHW_ZMF2X6_CAN_OPERATING_SLEEP) {

        /* 请求睡眠模式 */
        p_hw_can->mcr = (uint32_t)(p_hw_can->mcr | AMHW_ZMF2X6_CAN_MCR_SLEEP);

        /* 超时或 ACK 后确认状态 */
        if ((p_hw_can->mcr & (uint32_t)AMHW_ZMF2X6_CAN_MCR_SLEEP) != 0) {

```



```

        status = CAN_ENABLE;
    } else {
        status = CAN_DISABLE;
    }
    /* 测试模式 */
} else {
    status = CAN_DISABLE;
}

return (uint8_t) status;
}

```

程序清单 17.6 为 CAN 工作模式选择函数，用于配置 CAN 的当前工作模式。

程序清单 17.6 CAN 工作模式选择函数

```

am_static_inline
void amhw_zmf2x6_can_test_mode_request(amhw_zmf2x6_can_t *p_hw_can, amhw_zmf2x6_can_test_mode_t
mode)
{
    switch(mode)
    {
        case AMHW_ZMF2X6_CAN_MODE_NORMAL: p_hw_can->mcr &= ~(0x3 << 1); break;
        case AMHW_ZMF2X6_CAN_MODE_LISTEN_ONLY: p_hw_can->mcr |= (1 << 1); break;
        case AMHW_ZMF2X6_CAN_MODE_SELF_TEST: p_hw_can->mcr |= (1 << 2); break;
        case AMHW_ZMF2X6_CAN_MODE_ACCEP_FILTER: p_hw_can->mcr |= (1 << 3); break;
    }
}

```

程序清单 17.7 为 CAN 进入低功耗模式函数。程序清单 17.8 为 CAN 唤醒函数。

程序清单 17.7 CAN 进入低功耗模式函数

```

/**
 * \brief 进入低功耗模式
 *
 * \param[in] p_hw_can      : 指向 CAN 寄存器块的指针
 *
 * \return 返回睡眠模式状态 0 1
 */
am_static_inline
uint8_t amhw_zmf2x6_can_sleep(amhw_zmf2x6_can_t *p_hw_can)
{
    uint8_t sleepstatus = CAN_DISABLE;

    /* 请求睡眠模式 */
    p_hw_can->mcr = (((p_hw_can->mcr) & (uint32_t)
                    (~((uint32_t)AMHW_ZMF2X6_CAN_MCR_INRQ))) |

```

```
        AMHW_ZMF2X6_CAN_MCR_SLEEP);
/* 睡眠模式状态 */
if ( (p_hw_can->mcr & AMHW_ZMF2X6_CAN_MCR_SLEEP) ==
        AMHW_ZMF2X6_CAN_MCR_SLEEP) {
    sleepstatus = CAN_ENABLE;
}

return (uint8_t)sleepstatus;
}
```

程序清单 17.8 CAN 唤醒函数

```
/**
 * \brief 唤醒 CAN
 *
 * \param[in] p_hw_can      : 指向 CAN 寄存器块的指针
 *
 * \return  是否成功唤醒  0  1
 */
am_static_inline
uint8_t amhw_zmf2x6_can_wakeup (amhw_zmf2x6_can_t *p_hw_can)
{
    uint8_t  wakeupstatus = CAN_DISABLE;

    /* 唤醒请求 */
    p_hw_can->mcr &= ~(uint32_t)AMHW_ZMF2X6_CAN_MCR_SLEEP;

    /* 睡眠模式状态 */
    if ( (p_hw_can->mcr & AMHW_ZMF2X6_CAN_MCR_SLEEP) == 0) {
        wakeupstatus = CAN_ENABLE;
    }

    return (uint8_t)wakeupstatus;
}
```

2. CAN_CMRR(CAN 命令寄存器)

该寄存器涉及到 CAN 使用时会使用到的一些命令如表 17.2 列出了这些命令及其对应的含义。

表 17.2 CAN 命令寄存器分析

位	标识符	含义
5	ERB	清空所有接收 FIFO 数据
4	SRR	自接收请求
3	CDO	清除数据溢出
2	RRB	释放接收缓冲区

1	AT	终止传输
0	TR	发送请求

观察这些寄存器位可以明显看出终止传输以及发送请求两个位更具有单独设立函数的必要，其它命令仅需通过定义多个宏配合一个函数的方式完成定义。因此此处我们需要定义 3 个函数即 can 命令发布函数、can 传输启动函数、can 传输停止函数，分别如程序清单 17.9 程序清单 17.10 程序清单 17.11 所示

程序清单 17.9 can 命令发布函数

```
#define AMHW_ZMF2X6_CAN_CMD_RXFIFO_CLEAR      (1<<5)    /**< \brief 清空接收 fifo 数据 */
#define AMHW_ZMF2X6_CAN_CMD_SELF_RES_REQUEST (1<<4)    /**< \brief 自接收请求 */
#define AMHW_ZMF2X6_CAN_CMD_DATA_OVER_CLEAR  (1<<3)    /**< \brief 数据溢出清除 */
#define AMHW_ZMF2X6_CAN_CMD_REC_BUF_CLEAR     (1<<2)    /**< \brief 接收缓冲区释放 */

am_static_inline
void amhw_zmf2x6_can_cmd_set(amhw_zmf2x6_can_t *p_hw_can, uint8_t mode)
{
    switch(mode)
    {
        case AMHW_ZMF2X6_CAN_CMD_RXFIFO_CLEAR: p_hw_can->cmd |=
AMHW_ZMF2X6_CAN_CMD_RXFIFO_CLEAR; break;
        case AMHW_ZMF2X6_CAN_CMD_SELF_RES_REQUEST: p_hw_can->cmd |=
AMHW_ZMF2X6_CAN_CMD_SELF_RES_REQUEST; break;
        case AMHW_ZMF2X6_CAN_CMD_DATA_OVER_CLEAR: p_hw_can->cmd |=
AMHW_ZMF2X6_CAN_CMD_DATA_OVER_CLEAR; break;
        case AMHW_ZMF2X6_CAN_CMD_REC_BUF_CLEAR: p_hw_can->cmd |=
AMHW_ZMF2X6_CAN_CMD_REC_BUF_CLEAR; break;
    }
}
```

程序清单 17.10 can 传输启动函数

```
am_static_inline
void amhw_zmf2x6_can_trans_start(amhw_zmf2x6_can_t *p_hw_can)
{
    p_hw_can->cmd = (3 << 0);
}
```

程序清单 17.11 can 传输停止函数

```
am_static_inline
void amhw_zmf2x6_can_trans_stop(amhw_zmf2x6_can_t *p_hw_can)
{
    p_hw_can->cmd |= (1 << 1);
}
```

3. CAN_SR (CAN 状态寄存器)

该寄存器涉及到 CAN 的一些主要状态标记位，如表 17.3 为这些标记位的含义。

表 17.3 CAN_CANFG 寄存器分析

位	标识符	含义
7	BS	总线状态
6	ES	出错状态
5	TS	发送状态
4	RS	接收状态
3	TCS	发送完毕状态
2	TBS	发送缓冲区状态
1	DOS	数据溢出状态
0	RBS	接收缓冲区状态

我们仅需编写一系列的状态宏以及状态清除函数和状态获取函数即可。如程序清单 17.12 所示为状态清除函数以及状态获取函数。

程序清单 17.12 CAN 通道采样时间选择函数

```

/**< \brief CAN_SR */

#define AMHW_ZMF2X6_CAN_BUS_STA      (1<<7)  /**< \brief 1:CLOSE 0:OPEN */
#define AMHW_ZMF2X6_CAN_ERR_STA      (1<<6)  /**< \brief 1:ERR   0:NO ERR */
#define AMHW_ZMF2X6_CAN_TX_STA      (1<<5)  /**< \brief 1:TRANS 0:IDLE */
#define AMHW_ZMF2X6_CAN_RX_STA      (1<<4)  /**< \brief 1:Receive 0:IDLE */
#define AMHW_ZMF2X6_CAN_TX_COMP_STA  (1<<3)  /**< \brief Transmission complete status */
#define AMHW_ZMF2X6_CAN_TX_BUFF_STA  (1<<2)  /**< \brief 1:can TX 2:can not TX */
#define AMHW_ZMF2X6_CAN_DATA_OVER_STA (1<<1)  /**< \brief Data overrun status */
#define AMHW_ZMF2X6_CAN_RXFIFO_STA   (1<<0)  /**< \brief Receive buffer status */
#define AMHW_ZMF2X6_CAN_ALL_STA      (0xff)

/**
 * \brief CAN 检测主要标志位是否被设置
 *
 * \param[in] p_hw_can      : 指向 CAN 寄存器块的指针
 * \param[in] can_flag      : CAN_SR (AMHW_ZMF2X6_CAN_BUS_STA)
 * \return 无
 */
am_static_inline
uint8_t amhw_zmf2x6_can_get_flag_status(amhw_zmf2x6_can_t *p_hw_can, uint32_t can_flag)
{
    return (p_hw_can->msr & can_flag) == can_flag;
}

/**

```

```

* \brief CAN 清除主要标志位挂号状态
*
* \param[in] p_hw_can      : 指向 CAN 寄存器块的指针
* \param[in] can_it        : CAN_SR (AMHW_ZMF2X6_CAN_BUS_STA)
* \return 无
*/
am_static_inline
void amhw_zmf2x6_can_clear_flag(amhw_zmf2x6_can_t      *p_hw_can, uint32_t      can_flag)
{
    p_hw_can->msr &= ~can_flag;
}

```

以此类推，完成主要需要使用到的寄存器硬件层函数配置。

17.2.3 添加其他功能性函数

在每个寄存器的操作函数编写完毕后，可能 HW 层的接口还有所欠缺，需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到，可以暂时不用考虑，直接进入驱动层的开发。在后续如果发现 HW 层欠缺哪些接口，可以反过来再增加。当然，也有可能 HW 层接口没有什么欠缺了，那样就无需再开发其它函数了。

17.2.4 添加指向寄存器块的指针

可以看到，在 HW 层编写的函数中，第一个参数均为指向 CAN 寄存器块的指针，这个指针从哪里来呢？只需要把 CAN 的基地址使用(amhw_zmf2x6_can_t *)强制转换即可作为这些函数的第一个参数。这样转换后，操作结构体的成员，实际上就是操作具体地址上的寄存器了。

但如果每次使用都要这样转换的话就太麻烦了，为了方便用户或驱动层使用，可以将该转换使用一个宏定义好，用户直接使用宏即可。在 AMetal 中，由于这些内容与具体型号的芯片相关，这样的宏统一定义到 zmf2x6_periph_map.h 文件中，如程序清单 17.13 所示。

程序清单 17.13 外设中加入 CAN 的定义

```

/**
* \name 外设定义
* @{
*/
/** \brief 串口(UART2)寄存器块指针 */
#define ZMF2X6_UART2      ((amhw_zmf2x6_uart_t      *)ZMF2X6_UART2_BASE)

/** \brief 电源管理单元(PWR)寄存器块指针 */
#define ZMF2X6_PWR        ((amhw_zmf2x6_pwr_t      *)ZMF2X6_PWR_BASE)

/** \brief 备份区域(BKP)寄存器块指针 */
#define ZMF2X6_BKP        ((amhw_zmf2x6_bkp_t      *)ZMF2X6_BKP_BASE)

/** \brief 模数转换(CAN)寄存器块指针 */

```

```
#define ZMF2X6_CAN      ((amhw_zmf2x6_can_t      *)ZMF2X6_CAN_BASE)

/** \brief I2C 总线控制器(I2C0)寄存器块指针 */
#define ZMF2X6_I2C      ((amhw_zmf2x6_i2c_t      *)ZMF2X6_I2C1_BASE )
/**
 * @} zmf2x6_if_periph_map
 */
```

后续用户或驱动层使用 HW 层接口时,第一个参数直接使用 ZMF2X6_CAN 即可。至此,整个 HW 层的开发工作就结束了,如果后续开发中发现缺少某些函数,还可以再添加。

17.3 驱动层开发

参考 5.3.1 章节,新建 am_zmf2x6_can.h 和 am_zmf2x6_can.c 驱动文件,搭建相应的文件框架,并按规则放到对应的路径下。

17.3.1 编写结构性程序

驱动层首先应该搭建好相关结构,再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中,每个设备(CAN 就是一个设备)都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配,与设备相关的变量都应该定义在设备结构体中,而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下,设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员,用于驱动层使用 HW 层的接口。如果设备需要使用中断,还需要填入中断号信息。除此之外,一些设备的初始化配置信息,也可以由设备信息提供。

可以暂时定义设备信息结构体如程序清单 17.14 所示。后续可以根据实际情况修改。

程序清单 17.14 CAN 设备信息结构体定义

```
/**
 * \brief can 初始化配置信息
 */
typedef struct am_zmf2x6_can_init_info {

    /** \brief 0 正常模式 1 只听模式 2 自检测模式 */
    amhw_zmf2x6_can_test_mode_t    can_mode;
    /** \brief 1: 3 倍 0: 1 倍 */
    uint8_t                        sampling;
    /** \brief 错误报警限制 */
    uint8_t                        err_ewl;
}am_zmf2x6_can_init_info_t;

/**<brief 滤波器参数配置信息 */
typedef struct {
    uint8_t    can_filter_scale;    /** <brief 滤波器位宽 0 2 个 16 位 1 单个 32 位*/
```

```

}amhw_zmf2x6_can_filter_t;

/**
 * \brief 中断回调信息
 */
typedef struct am_zmf2x6_can_intcb_info {

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

    /** \brief 回调函数的参数 */
    void          *p_arg;
} am_zmf2x6_can_intcb_info_t;

/**
 * \brief CAN 设备信息
 */
typedef struct am_zmf2x6_can_devinfo {

    /** \brief 寄存器基地址 */
    uint32_t      regbase;

    /** \brief 滤波寄存器基地址 */
    uint32_t      fil_regbase;

    /** \brief 中断号 */
    uint32_t      int_num;

    /** \brief 回调信息 */
    am_zmf2x6_can_intcb_info_t *p_intcb_info;

    /** \brief 回调信息内存大小 */
    uint8_t       p_intcb_num;

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

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

    /** \brief 滤波参数配置 */
    amhw_zmf2x6_can_filter_t *filter;

    /** \brief 信息初始化 */
    am_zmf2x6_can_init_info_t *init_info;

```

```
} am_zmf2x6_can_devinfo_t;
```

● 设备结构体定义

由于 CAN 设备需要提供标准的 CAN 服务，因此，该设备至少包含一个 CAN 标准服务结构体成员，也就是 `am_can_serv_t` 类型成员，其次，为了在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 17.15 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 17.15 CAN 设备结构体定义

```
/**
 * \brief 设备
 */
typedef struct am_zmf2x6_can_dev {
    am_can_serv_t          handle;

    /** \brief 工作模式 */
    amhw_zmf2x6_can_mode_t    mode;

    const am_zmf2x6_can_devinfo_t *p_devinfo;
} am_zmf2x6_can_dev_t;
```

2. 搭建函数框架

函数主要分为两部分一部分是提供给用户使用的驱动层函数，一部分是提供给标准层的函数。

● 驱动函数

对于 CAN 这种已经有标准接口层的函数，通常驱动层只提供两个函数给用户，初始化函数和解初始化函数。

初始化函数的参数是设备指针和设备信息常量指针，返回值为用于标准接口的 `handle`。

解初始化函数主要是当不需要再使用某个外设时，释放掉其占用的相关资源，参数为设备指针，无返回值。示例代码如程序清单 17.16 所示。

程序清单 17.16 驱动层函数定义示例

```
/** \brief 初始化 CAN，获取标准服务句柄 */
am_can_handle_t am_zmf2x6_can_init (am_zmf2x6_can_dev_t          *p_dev,
                                   const am_zmf2x6_can_devinfo_t *p_devinfo)
{
    return (am_can_handle_t)&(p_dev->handle);
}

/** \brief CAN 解初始化 */
void am_zmf2x6_can_deinit (am_can_handle_t handle)
{
}
```



```
}

```

同时为了用户能够使用，还需要将这两个函数的声明放入到相应的头文件中。示例代码如程序清单 17.17 所示。

程序清单 17.17 驱动函数声明

```
/**
 * \brief CAN 初始化
 *
 * \param[in] p_dev      : 指向 CAN 设备的指针
 * \param[in] p_devinfo : 指向 CAN 设备信息的指针
 *
 * \return CAN 标准服务操作句柄 如果为 NULL，表明初始化失败
 */
am_can_handle_t am_zmf2x6_can_init (am_zmf2x6_can_dev_t      *p_dev,
                                   const am_zmf2x6_can_devinfo_t *p_devinfo);

/**
 * \brief CAN 去初始化
 *
 * \param[in] handle : am_zmf2x6_can_init() 初始化函数获得的 CAN 服务句柄
 *
 * \return 无
 */
void am_zmf2x6_can_deinit (am_can_handle_t handle);
```

- 提供给标准层的函数

如前面分析的标准层函数实现，驱动层应该给标准接口层提供 21 个函数，根据标准接口层中的 struct am_can_drv_funcs 结构体，我们需要在.c 文件中定义这些函数，当然并非一定要全部定义，只需要保证一些必须功能被定义即可。

由于这些函数全部是放在结构体中，通过函数指针的形式提供给上层，并不需要将这些函数声明到.h 文件中供用户直接使用，因此全部以 static 的形式定义成内部函数。定义好函数后，再定义一个 struct am_can_drv_funcs 结构体，将函数放在其中。示例代码如程序清单 17.18 所示。

程序清单 17.18 提供给标准层函数定义

```
/** \brief CAN 启动 */
am_can_err_t __can_zmf2x6_start (void *p_drv);
/** \brief CAN 复位（停止） */
am_can_err_t __can_zmf2x6_reset (void *p_drv);
/** \brief CAN 休眠 */
am_can_err_t __can_zmf2x6_sleep (void *p_drv);
/** \brief CAN 唤醒 */
am_can_err_t __can_zmf2x6_wakeup (void *p_drv);
/** \brief 中断使能 */
am_can_err_t __can_zmf2x6_int_enable (void *p_drv, am_can_int_type_t int_mask);
```

```

/** \brief 中断禁能*/
am_can_err_t __can_zmf2x6_int_disable (void *p_drv, am_can_int_type_t int_mask);

/** \brief 模式设置 */
am_can_err_t __can_zmf2x6_mode_set (void *p_drv, am_can_mode_type_t mode);

/** \brief 波特率设置 */
am_can_err_t __can_zmf2x6_baudrate_set (void *p_drv,
                                         am_can_bps_param_t *p_can_baudrate);

/** \brief 波特率获取 */
am_can_err_t __can_zmf2x6_baudrate_get (void *p_drv,
                                         am_can_bps_param_t *p_can_baudrate);

/** \brief 获取错误计数 */
am_can_err_t __can_zmf2x6_err_cnt_get (void *p_drv,
                                       am_can_err_cnt_t *p_can_err_reg);

/** \brief 发送信息 */
am_can_err_t __can_zmf2x6_msg_send (void *p_drv, am_can_message_t *p_txmsg);

/** \brief 接收信息 */
am_can_err_t __can_zmf2x6_msg_rcv (void *p_drv, am_can_message_t *p_rxmsg);

/** \brief 停止发送信息 */
am_can_err_t __can_zmf2x6_stop_msg_snd (void *p_drv);

/** \brief CAN 状态 */
am_can_err_t __can_zmf2x6_status_get (void *p_drv,
                                       am_can_int_type_t *p_int_type,
                                       am_can_bus_err_t *p_bus_err);

/** \brief 中断连接 */
am_can_err_t __can_zmf2x6_connect (void *p_drv,
                                   am_pfnvoid_t pfn_isr,
                                   void *p_arg);

/** \brief 删除中断连接 */
am_can_err_t __can_zmf2x6_disconnect (void *p_drv,
                                       am_pfnvoid_t pfn_isr,
                                       void *p_arg);

/** \brief 中断回调函数的连接 */
am_can_err_t __can_zmf2x6_intcb_connect (void *p_drv,
                                         am_can_int_type_t inttype,
                                         am_pfnvoid_t pfn_callback,
                                         void *p_arg);

/** \brief 删除中断回调函数的连接 */
am_can_err_t __can_zmf2x6_intcb_disconnect (void *p_drv,
                                             am_can_int_type_t inttype);

/** \brief 设置滤波函数 (扩展)*/

```

```

am_can_err_t __can_zmf2x6_filter_tab_ext_set(void          *p_drv,
                                              am_can_filter_t *p_filterbuff,
                                              size_t          lenth);

/** \brief 获取滤波函数 (扩展)*/
am_can_err_t __can_zmf2x6_filter_tab_ext_get(void          *p_drv,
                                              am_can_filter_t *p_filterbuff,
                                              size_t          *p_lenth);

struct am_can_drv_funcs __zmf2x6_can_dev_func = {
    __can_zmf2x6_start,
    __can_zmf2x6_reset,
    __can_zmf2x6_sleep,
    __can_zmf2x6_wakeup,
    __can_zmf2x6_int_enable,
    __can_zmf2x6_int_disable,
    __can_zmf2x6_mode_set,
    __can_zmf2x6_baudrate_set,
    __can_zmf2x6_baudrate_get,
    __can_zmf2x6_err_cnt_get,
    NULL,
    __can_zmf2x6_msg_send,
    __can_zmf2x6_msg_rcv,
    __can_zmf2x6_stop_msg_snd,
    NULL,
    NULL,
    __can_zmf2x6_status_get,
    __can_zmf2x6_connect,
    __can_zmf2x6_disconnect,
    __can_zmf2x6_intcb_connect,
    __can_zmf2x6_intcb_disconnect,
    __can_zmf2x6_filter_tab_ext_set,
    __can_zmf2x6_filter_tab_ext_get,
};

/** \brief 启动 CAN 转换 */
am_can_err_t __can_zmf2x6_start(void *p_drv)
{
    return AM_OK;
}

/** \brief CAN 复位（停止） */
am_can_err_t __can_zmf2x6_reset(void *p_drv)
{

```

```

        return AM_OK;
    }

    /** \brief CAN 休眠 */
    am_can_err_t __can_zmf2x6_sleep (void *p_drv)
    {

        return AM_OK;
    }

    /** \brief CAN 唤醒 */
    am_can_err_t __can_zmf2x6_wakeup (void *p_drv)
    {

        return AM_OK;
    }

    /** \brief 中断使能 */
    am_can_err_t __can_zmf2x6_int_enable (void *p_drv, am_can_int_type_t int_mask)
    {

        return AM_OK;
    }

    /** \brief 中断禁能*/
    am_can_err_t __can_zmf2x6_int_disable (void *p_drv, am_can_int_type_t int_mask)
    {

        return AM_OK;
    }

    /** \brief 模式设置 */
    am_can_err_t __can_zmf2x6_mode_set (void *p_drv, am_can_mode_type_t mode)
    {

        return AM_OK;
    }

    /** \brief 波特率设置 */
    am_can_err_t __can_zmf2x6_baudrate_set (void *p_drv,
                                              am_can_bps_param_t *p_can_baudrate)
    {

        return AM_OK;
    }

```

```

}

/** \brief 波特率获取 */
am_can_err_t __can_zmf2x6_baudrate_get (void          *p_drv,
                                         am_can_bps_param_t *p_can_baudrate)
{

    return AM_OK;
}

/** \brief 获取错误计数 */
am_can_err_t __can_zmf2x6_err_cnt_get (void          *p_drv,
                                         am_can_err_cnt_t *p_can_err_reg)
{

    return AM_OK;
}

/** \brief 发送信息 */
am_can_err_t __can_zmf2x6_msg_send (void *p_drv, am_can_message_t *p_txmsg)
{

    return AM_OK;
}

/** \brief 接收信息 */
am_can_err_t __can_zmf2x6_msg_rcv (void *p_drv, am_can_message_t *p_rxmsg)
{

    return AM_OK;
}

/** \brief 停止发送信息 */
am_can_err_t __can_zmf2x6_stop_msg_snd (void *p_drv)
{

    return AM_OK;
}

/** \brief CAN 状态 */
am_can_err_t __can_zmf2x6_status_get (void          *p_drv,
                                         am_can_int_type_t *p_int_type,
                                         am_can_bus_err_t *p_bus_err)
{

```

```

        return AM_OK;
    }

/** \brief 中断连接 */
am_can_err_t __can_zmf2x6_connect (void          *p_drv,
                                   am_pfnvoid_t   pfn_isr,
                                   void           *p_arg)
{

    return AM_OK;
}

/** \brief 删除中断连接 */
am_can_err_t __can_zmf2x6_disconnect (void          *p_drv,
                                      am_pfnvoid_t   pfn_isr,
                                      void           *p_arg)
{

    return AM_OK;
}

/** \brief 中断回调函数的连接 */
am_can_err_t __can_zmf2x6_intcb_connect (void          *p_drv,
                                          am_can_int_type_t inttype,
                                          am_pfnvoid_t   pfn_callback,
                                          void           *p_arg)
{

    return AM_OK;
}

/** \brief 删除中断回调函数的连接 */
am_can_err_t __can_zmf2x6_intcb_disconnect (void          *p_drv,
                                             am_can_int_type_t inttype)
{

    return AM_OK;
}

/** \brief 设置滤波函数 (扩展)*/
am_can_err_t __can_zmf2x6_filter_tab_ext_set (void          *p_drv,
                                                am_can_filter_t *p_filterbuff,
                                                size_t         lenth)

```

```

{

    return AM_OK;
}

/** \brief 获取滤波函数 (扩展)*/
am_can_err_t __can_zmf2x6_filter_tab_ext_get (void          *p_drv,
                                              am_can_filter_t *p_filterbuff,
                                              size_t          *p_lenth)
{

    return AM_OK;
}

```

至此，整个函数结构就搭建好了，接下来就是简单的填充工作了。可以编译一下，应该是没有错误的。如果有错误，表明结构没有搭建好。可能存在警告，毕竟有些变量没有使用，可以暂时不用理会。

17.3.2 驱动函数实现

通过上面的函数框架搭建，共计 22 个函数，20 个功能性函数面向标准层，2 个初始解初始化函数面向用户。除了这上述 20 个函数外，我们还可能会将要调用的一些功能代码封装为相应的函数以保证程序的易维护性，在这里我们还添加了 __can_zmf2x6_init 函数，其只被 __can_zmf2x6_start 调用，__can_zmf2x6_filter_32bit_idmask、__can_zmf2x6_filter_16bit_idmask 仅被 __can_zmf2x6_filter_tab_ext_set 调用。由于 CAN 能够产生中断因此我们需要编写相应的中断服务函数 (__can_zmf2x6_int_handle)。除这些函数以外，根据 zmf2x6 CAN 模块的特性，我们还定义了两个函数 am_zmf2x6_can_alc_get、am_zmf2x6_can_err_code_get 用于扩展该模块功能，下面逐步实现。

1. am_zmf2x6_can_init 函数

该函数主要完成两个功能：

- a) 初始化设备中的标准服务，将成员赋值，确定 p_drv 的值，p_drv 的值确定了提供给标准接口层的那 20 个函数的第一个参数是什么。那 20 个函数才能根据这儿的设置去使用。（void *类型无法使用，必须转换为具体的一种类型才能使用），通常情况下，p_drv 都是设置为设备结构体的地址，这样方便访问整个设备的相关内容。
- b) 根据 p_devinfo 内配置的信息进行一些硬件的基础配置，并进行一些必要的初始化配置。

该函数的示例代码如程序清单 17.19 所示。

程序清单 17.19 CAN 初始化函数

```

/**
 * \brief CAN 初始化
 */
am_can_handle_t am_zmf2x6_can_init (am_zmf2x6_can_dev_t *p_dev,
                                   const am_zmf2x6_can_devinfo_t *p_devinfo)
{

```

```

amhw_zmf2x6_can_t      *p_hw_can = NULL;
amhw_zmf2x6_canflt_t   *p_hw_filcan;
uint8_t                i = 0;

if (NULL == p_dev || NULL == p_devinfo) {
    return NULL;
}

if (p_devinfo->pfn_plfm_init) {
    p_devinfo->pfn_plfm_init();
}

p_dev->p_devinfo = p_devinfo;

p_dev->handle.p_drv_funcs = &__zmf2x6_can_dev_func;
p_dev->handle.p_drv       = (void *)p_dev;

p_hw_can   = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;
p_hw_filcan = (amhw_zmf2x6_canflt_t *)p_dev->p_devinfo->fil_regbase;

/* 退出睡眠  请求进入初始化模式 */
amhw_zmf2x6_can_operating_mode_request(
    p_hw_can, AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT);
p_dev->mode = AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT;
amhw_zmf2x6_can_peli_mode_set(p_hw_can);

/* 禁能滤波表 */
p_hw_filcan->fga[0] = 0x0;
p_hw_filcan->fga[1] = 0x0;
p_hw_filcan->fga[2] = 0x0;

for (i = 0 ; i < p_devinfo->p_intcb_num ; i++) {
    p_devinfo->p_intcb_info[i].pfn_callback = NULL;
    p_devinfo->p_intcb_info[i].p_arg       = NULL;
}

return (am_can_handle_t)&(p_dev->handle);
}

```

2. am_zmf2x6_can_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放，仅将部分结构体指向 NULL 即可。另外是对硬件资源的释放，包括禁能 CAN 及相关中断，执行平台去初始化函数详见程序清单 17.20。

程序清单 17.20 CAN 去初始化函数


```

/**
 * \brief 解初始化
 */
void am_zmf2x6_can_deinit (am_can_handle_t handle)
{
    amhw_zmf2x6_can_t *p_hw_can = NULL;
    am_zmf2x6_can_dev_t *p_dev = (am_zmf2x6_can_dev_t *)handle->p_drv;

    if (NULL == p_dev) {
        return;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    amhw_zmf2x6_can_operating_mode_request(p_hw_can,
        AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT);

    if (p_dev->p_devinfo->pfn_plfm_deinit) {
        p_dev->p_devinfo->pfn_plfm_deinit();
    }

    /* 中断禁能 */
    am_int_disable(INUM_CAN);
}

```

3. __can_zmf2x6_start

程序清单 17.21 为 can 启动函数，其主要进行的是对 CAN 模块硬件的初始化，并使能 CAN 全局中断。其中对硬件的初始化部分我们封装在了 __can_zmf2x6_init 函数内。

程序清单 17.21 启动 CAN 转换函数

```

/**
 * \brief CAN 启动
 */
am_can_err_t __can_zmf2x6_start (void *p_drv)
{
    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    /* 初始化 CAN 控制信息 */
    __can_zmf2x6_init(p_drv);

    /* 使能中断 */
    am_int_enable(INUM_CAN);
}

```

```

return AM_CAN_NOERROR;
}

```

如程序清单 17.22 所示为 CAN 的硬件初始化函数，其主要完成的操作是对前面配置的信息结构体 `am_zmf2x6_can_init_info_t` 的一个实现。同时进行一些必要的硬件配置，此处没有。由于这个 CAN 模块仅在初始化模式下才可进行部分寄存器的配置，因此因在函数的头尾分别添加一个进入初始化模式操作以及一个进入正常模式操作。

程序清单 17.22 硬件初始化函数

```

/**
 * \brief CAN 初始化
 */
uint8_t __can_zmf2x6_init(void *p_drv)
{
    am_zmf2x6_can_dev_t      *p_dev      = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t        *p_hw_can   = NULL;
    am_zmf2x6_can_init_info_t *p_can_info = NULL;
    uint8_t                  init_staus  = 0;

    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_can_info = (am_zmf2x6_can_init_info_t *)p_dev->p_devinfo->init_info;
    p_hw_can   = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    /* 退出睡眠 请求进入初始化模式 */
    if (amhw_zmf2x6_can_operating_mode_request(
        p_hw_can, AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT)) {

        amhw_zmf2x6_can_test_mode_request(p_hw_can, p_can_info->can_mode);

        amhw_zmf2x6_can_sam_set(p_hw_can, p_can_info->sampling);
        /* 错误报警限制 */
        p_hw_can->ewlr = p_can_info->err_ewl;

        /* 进入正常模式 */
        if (amhw_zmf2x6_can_operating_mode_request(
            p_hw_can, AMHW_ZMF2X6_CAN_OPERATING_NORMAL)) {
            init_staus = 1;
            p_dev->mode = AMHW_ZMF2X6_CAN_OPERATING_NORMAL;
        } else {
            init_staus = 0;
        }
    }
}

```

```

    } else {
        init_staus = 0;
    }

    return init_staus;
}

```

4. __can_zmf2x6_reset

程序清单 17.23 为 CAN 停止函数，在此处主要完成的是让 CAN 模块进入睡眠，并关闭中断操作。

程序清单 17.23 CAN 停止函数

```

/**
 * \brief CAN 复位（停止）
 */
am_can_err_t __can_zmf2x6_reset (void *p_drv)
{
    am_zmf2x6_can_dev_t *p_dev = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t *p_hw_can = NULL;

    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    amhw_zmf2x6_can_operating_mode_request(
        p_hw_can, AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT);
    amhw_zmf2x6_can_operating_mode_request(
        p_hw_can, AMHW_ZMF2X6_CAN_OPERATING_SLEEP);

    /* 禁能中断 */
    am_int_disable(INUM_CAN);

    return AM_CAN_NOERROR;
}

```

5. __can_zmf2x6_sleep

程序清单 17.24 为 CAN 的睡眠函数，在 hw 层已实现其的配置，直接调用即可。

程序清单 17.24 CAN 睡眠函数

```

/**
 * \brief CAN 休眠
 */
am_can_err_t __can_zmf2x6_sleep (void *p_drv)
{

```

```

am_zmf2x6_can_dev_t    *p_dev    = (am_zmf2x6_can_dev_t *)p_drv;
amhw_zmf2x6_can_t      *p_hw_can = NULL;

if (NULL == p_drv) {
    return AM_CAN_INVALID_PARAMETER;
}

p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

/* 切换到睡眠模式*/
amhw_zmf2x6_can_operating_mode_request(
    p_hw_can, AMHW_ZMF2X6_CAN_OPERATING_SLEEP);

return AM_CAN_NOERROR;
}

```

6. __can_zmf2x6_wakeup

程序清单 17.25 为 CAN 的唤醒函数，在 hw 层已实现其的配置，直接调用即可。

程序清单 17.25 CAN 唤醒函数

```

/**
 * \brief CAN 唤醒
 */
am_can_err_t __can_zmf2x6_wakeup (void *p_drv)
{
    am_zmf2x6_can_dev_t    *p_dev    = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t      *p_hw_can = NULL;

    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    /* 切换到正常模式*/
    amhw_zmf2x6_can_operating_mode_request(
        p_hw_can, AMHW_ZMF2X6_CAN_OPERATING_NORMAL);

    return AM_CAN_NOERROR;
}

```

7. __can_zmf2x6_int_enable

CAN 中断使能函数，其主要完成对 CAN 可能发生的各种中断的一个使能操作。相关中断描述宏如表 17.4 所示。

表 17.4 CAN 中断类型

中断描述宏	含义
AM_CAN_INT_ERROR	错误中断
AM_CAN_INT_BUS_OFF	总线关闭中断
AM_CAN_INT_WAKE_UP	唤醒中断
AM_CAN_INT_TX	发送中断
AM_CAN_INT_RX	接收中断
AM_CAN_INT_DATAOVER	总线超载中断
AM_CAN_INT_WARN	警告中断
AM_CAN_INT_ERROR_PASSIVE	错误被动中断

直接调用
hw 层
编写的
函数
amhw_
zmf2x
6_can_
int_ena
ble 配
合中断
对应的
宏完成

该函数的编写，不同的 CAN 模块可能相关中断的类型会有些许区别，需要对他们进行区分后再填充到对应的标准宏下面。相关代码如程序清单 17.26 所示。

程序清单 17.26 CAN 中断使能函数

```
/**
 * \brief 中断使能
 */
am_can_err_t __can_zmf2x6_int_enable(void *p_drv, am_can_int_type_t int_mask)
{
    am_zmf2x6_can_dev_t      *p_dev      = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t        *p_hw_can = NULL;
    amhw_zmf2x6_can_mode_t    mode      = p_dev->mode;

    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;
    amhw_zmf2x6_can_operating_mode_request(p_hw_can,
        AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT);

    /* 接收中断 */
    if (int_mask & AM_CAN_INT_RX) {

        /* 接收中断使能 */
        amhw_zmf2x6_can_int_enable(p_hw_can, AMHW_ZMF2X6_CAN_RI_FLAG);
        int_mask &= ~AM_CAN_INT_RX;
    }

    /* 发送中断 */
    if (int_mask & AM_CAN_INT_TX) {
```

```

/* 发送中断使能 */
amhw_zmf2x6_can_int_enable(p_hw_can, AMHW_ZMF2X6_CAN_TI_FLAG);
int_mask &= ~AM_CAN_INT_TX;
}

/* 错误\警告中断 */
if ((int_mask & AM_CAN_INT_ERROR)) {

    /* 错误警告 和 错误中断使能*/
    amhw_zmf2x6_can_int_enable(p_hw_can, AMHW_ZMF2X6_CAN_BEI_FLAG);
    amhw_zmf2x6_can_int_enable(p_hw_can, AMHW_ZMF2X6_CAN_EI_FLAG);
    int_mask &= ~AM_CAN_INT_ERROR;
}

/* 总线超载中断 */
if (int_mask & AM_CAN_INT_DATAOVER) {

    /* 使能数据溢出中断 FIFO 0 1 */
    amhw_zmf2x6_can_int_enable(p_hw_can, AMHW_ZMF2X6_CAN_DOI_FLAG);
    int_mask &= ~AM_CAN_INT_DATAOVER;
}

/* 错误被动中断 */
if ((int_mask & AM_CAN_INT_ERROR_PASSIVE)) {

    /* 使能错误被动中断 */
    amhw_zmf2x6_can_int_enable(p_hw_can, AMHW_ZMF2X6_CAN_EPI_FLAG);
    int_mask &= ~AM_CAN_INT_ERROR_PASSIVE;
}

/* 警告中断 */
if ((int_mask & AM_CAN_INT_WARN)) {

    /* 仲裁丢失中断 */
    amhw_zmf2x6_can_int_enable(p_hw_can, AMHW_ZMF2X6_CAN_ALI_FLAG);
    int_mask &= ~AM_CAN_INT_WARN;
}

/* 恢复 CAN 状态 */
if (mode != AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT) {

    amhw_zmf2x6_can_operating_mode_request(p_hw_can, mode);
}

```

```

    }

    return AM_CAN_NOERROR;
}

```

8. __can_zmf2x6_int_disable

与 CAN 中断使能代码类似，直接调用 hw 层编写的函数 `amhw_zmf2x6_can_int_disable` 配合中断对应的宏完成该函数的编写，不同的 CAN 模块可能相关中断的类型会有些许区别，需要对它们进行区分后再填充到对应的标准宏下面。相关代码如程序清单 17.27 所示。

程序清单 17.27 获取 CAN 参考电压函数

```

/**
 * \brief 中断禁能
 */
am_can_err_t __can_zmf2x6_int_disable(void *p_drv, am_can_int_type_t int_mask)
{
    am_zmf2x6_can_dev_t      *p_dev      = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t        *p_hw_can   = NULL;
    amhw_zmf2x6_can_mode_t    mode       = p_dev->mode;

    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }
    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    amhw_zmf2x6_can_operating_mode_request(p_hw_can,
        AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT);

    /* 接收中断 */
    if (int_mask & AM_CAN_INT_RX) {

        /* 接收中断使能 */
        amhw_zmf2x6_can_int_disable(p_hw_can, AMHW_ZMF2X6_CAN_RI_FLAG);
        int_mask &= ~AM_CAN_INT_RX;
    }

    /* 发送中断 */
    if (int_mask & AM_CAN_INT_TX) {

        /* 发送中断使能 */
        amhw_zmf2x6_can_int_disable(p_hw_can, AMHW_ZMF2X6_CAN_TI_FLAG);
        int_mask &= ~AM_CAN_INT_TX;
    }
}

```

```

/* 错误\警告中断 */
if ((int_mask & AM_CAN_INT_ERROR)) {

    /* 错误警告 和 错误中断使能*/
    amhw_zmf2x6_can_int_disable(p_hw_can, AMHW_ZMF2X6_CAN_BEI_FLAG);
    amhw_zmf2x6_can_int_disable(p_hw_can, AMHW_ZMF2X6_CAN_EI_FLAG);
    int_mask &= ~AM_CAN_INT_ERROR;
}

/* 总线超载中断 */
if (int_mask & AM_CAN_INT_DATAOVER) {

    /* 使能数据溢出中断 */
    amhw_zmf2x6_can_int_disable(p_hw_can, AMHW_ZMF2X6_CAN_DOI_FLAG);
    int_mask &= ~AM_CAN_INT_DATAOVER;
}

/* 错误被动中断 */
if ((int_mask & AM_CAN_INT_ERROR_PASSIVE)) {

    /* 使能数据溢出中断 */
    amhw_zmf2x6_can_int_disable(p_hw_can, AMHW_ZMF2X6_CAN_EPI_FLAG);
    int_mask &= ~AM_CAN_INT_ERROR_PASSIVE;
}

/* 警告中断 */
if ((int_mask & AM_CAN_INT_WARN)) {

    /* 仲裁丢失中断 */
    amhw_zmf2x6_can_int_disable(p_hw_can, AMHW_ZMF2X6_CAN_ALI_FLAG);
    int_mask &= ~AM_CAN_INT_WARN;
}

/* 恢复 CAN 状态 */
if (mode != AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT) {

    amhw_zmf2x6_can_operating_mode_request(p_hw_can, mode);
}

return AM_CAN_NOERROR;
}

```

9. __can_zmf2x6_mode_set

CAN 模式设置函数，它的实现仅需根据参数 mode 的值配置 CAN 模块的模式，此处

mode 可能的值在 am_can.h 中已经定义如下所示。

```
typedef uint8_t am_can_mode_type_t;          /**< \brief CAN 模式类型 */
#define AM_CAN_MODE_NROMAL                   0x00 /**< \brief 正常工作模式 */
#define AM_CAN_MODE_LISTEN_ONLY              0x01 /**< \brief 只听工作模式 */
```

直接调用 hw 层接口函数 amhw_zmf2x6_can_test_mode_request 完成该函数的编写。详细代码见程序清单 17.28

程序清单 17.28 CAN 数据转换完成中断

```
/**
 * \brief 模式设置
 */
am_can_err_t __can_zmf2x6_mode_set(void *p_drv, am_can_mode_type_t mode)
{
    am_zmf2x6_can_dev_t *p_dev    = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t    *p_hw_can = NULL;

    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    if (AM_CAN_MODE_NROMAL == mode) {

        /* 正常模式 */
        amhw_zmf2x6_can_test_mode_request(p_hw_can, AMHW_ZMF2X6_CAN_MODE_NORMAL);
    } else if (AM_CAN_MODE_LISTEN_ONLY == mode) {

        /* 静默模式*/
        amhw_zmf2x6_can_test_mode_request(p_hw_can, AMHW_ZMF2X6_CAN_MODE_LISTEN_ONLY);
    } else {

        /* 回环 和 回环静默未设置 */

        return AM_CAN_INVALID_PARAMETER;
    }

    return AM_CAN_NOERROR;
}
```

10. __can_zmf2x6_baudrate_set

波特率配置函数，该函数除了能够配置波特率外也能够同时配置采样点的值，它的一个参数为波特率配置结构体，如程序清单 17.29 所示。

程序清单 17.29 波特率配置结构体

```
/** \brief 波特率参数 */
```

```
typedef struct am_can_bps_param {
    uint8_t    tseg1;                /**< \brief tseg1 相位段 1    */
    uint8_t    tseg2;                /**< \brief tseg2 相位段 2    */
    uint8_t    sjw;                  /**< \brief sjw 同步跳转宽度 */
    uint8_t    smp;                  /**< \brief smp 采样模式      */
    uint16_t    brp;                 /**< \brief brp 分频值        */
} am_can_bps_param_t;
```

该函数主要涉及到对波特率分频值、同步跳转宽度、相位段 1、相位段 2 的对应寄存器配置。值得注意的是波特率寄存器仅能在初始化模式下进行配置。详细代码见程序清单 17.30。

程序清单 17.30 波特率的配置函数

```
/**
 * \brief 波特率设置
 */
am_can_err_t __can_zmf2x6_baudrate_set(void *p_drv,
                                         am_can_bps_param_t *p_can_baudrate)
{
    am_zmf2x6_can_dev_t *p_dev = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t *p_hw_can = NULL;
    amhw_zmf2x6_can_mode_t mode = p_dev->mode;

    if (NULL == p_drv || NULL == p_can_baudrate) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    if (AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT != mode) {

        /* 波特率寄存器只在初始化模式下可进行访问，设置前进入初始化模式 */
        amhw_zmf2x6_can_operating_mode_request(p_hw_can,
                                                AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT);
    }

    /* 设置 波特率分频器 同步跳转宽度 和相位段 */
    amhw_zmf2x6_can_brp_set(p_hw_can, p_can_baudrate->brp);
    amhw_zmf2x6_can_sjw_set(p_hw_can, p_can_baudrate->sjw);

    amhw_zmf2x6_can_tseg1_set(p_hw_can, p_can_baudrate->tseg1);
    amhw_zmf2x6_can_tseg2_set(p_hw_can, p_can_baudrate->tseg2);

    if (mode != AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT) {
```

```

    /* 切换到正常模式 */
    amhw_zmf2x6_can_operating_mode_request(p_hw_can,
                                           mode);
}

return AM_CAN_NOERROR;
}

```

11. __can_zmf2x6_baudrate_get

波特率获取函数与波特率设置函数是一个相反的过程，仅仅是将设置相关寄存器改为获取而已。代码如程序清单 17.31 所示。

程序清单 17.31 波特率获取函数

```

/**
 * \brief 波特率获取
 */
am_can_err_t __can_zmf2x6_baudrate_get(void *p_drv,
                                         am_can_bps_param_t *p_can_baudrate)
{
    am_zmf2x6_can_dev_t *p_dev = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t *p_hw_can = NULL;
    amhw_zmf2x6_can_mode_t mode = p_dev->mode;

    if (NULL == p_drv || NULL == p_can_baudrate) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    if (AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT != p_dev->mode) {
        /* 波特率寄存器只在初始化模式下可进行访问，设置前进入初始化模式 */
        amhw_zmf2x6_can_operating_mode_request(p_hw_can,
                                                AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT);
    }

    p_can_baudrate->brp = amhw_zmf2x6_can_brp_get(p_hw_can);
    p_can_baudrate->sjw = amhw_zmf2x6_can_sjw_get(p_hw_can);

    p_can_baudrate->tesg1 = amhw_zmf2x6_can_tseg1_get(p_hw_can);
    p_can_baudrate->tesg2 = amhw_zmf2x6_can_tseg2_get(p_hw_can);

    if (mode != AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT) {

```

```

    /* 切换到正常模式 */
    amhw_zmf2x6_can_operating_mode_request(p_hw_can,
        mode);
}

return AM_CAN_NOERROR;
}

```

12. __can_zmf2x6_err_cnt_get

获取错误计数器的值，传入参数为错误计数器结构体，存在于 am_can.h 内，如程序清单 17.32 所示。

程序清单 17.32 错误计数器结构体

```

/** \brief 错误计数 */
typedef struct am_can_err_cnt {
    uint8_t rx_error_cnt;           /**< \brief 接收错误计数器 */
    uint8_t tx_error_cnt;          /**< \brief 发送错误计数器 */
} am_can_err_cnt_t;

```

该函数主要完成的工作即从寄存器中读出错误计数器的值，然后传递给该结构体变量。详细代码如程序清单 17.33 所示。

程序清单 17.33 获取错误计数器值函数

```

/**
 * \brief 获取错误计数
 */
am_can_err_t __can_zmf2x6_err_cnt_get(void *p_drv,
                                       am_can_err_cnt_t *p_can_err_reg)
{
    am_zmf2x6_can_dev_t *p_dev = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t *p_hw_can = NULL;

    if (NULL == p_drv || NULL == p_can_err_reg) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    /* 读 TX/RX 错误计数器 */
    p_can_err_reg->rx_error_cnt = amhw_zmf2x6_can_get_recverrorcount(p_hw_can);
    p_can_err_reg->tx_error_cnt = amhw_zmf2x6_can_get_transerrorcount(p_hw_can);

    return AM_CAN_NOERROR;
}

```

13. __can_zmf2x6_msg_send

该函数实现 CAN 消息的发送，主要涉及到根据传入消息类型 `am_can_message_t`，去构建发送帧结构，一般来说 CAN 模块对帧结构的构建都简化为了对寄存器的编写，其主要涉及到发送帧信息寄存器、发送识别码寄存器、发送数据寄存器的配置。详细内容见程序清单 17.34。

程序清单 17.34 CAN 消息发送

```
/**
 * \brief 发送信息
 */
am_can_err_t __can_zmf2x6_msg_send (void *p_drv, am_can_message_t *p_txmsg)
{
    am_zmf2x6_can_dev_t    *p_dev      = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t      *p_hw_can = NULL;
    am_bool_t              is_remote = AM_FALSE;
    am_bool_t              is_extern = AM_FALSE;
    uint32_t               reg_sff = 0;
    uint32_t               msg_id = 0;

    uint8_t i = 0;

    if (NULL == p_drv || NULL == p_txmsg) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    /* 检查发送缓冲区是否空闲 */
    if(amhw_zmf2x6_can_get_flag_status(p_hw_can, AMHW_ZMF2X6_CAN_TX_BUFF_STA) == 0) {
        return AM_CAN_NO_AVAIL_CHANNELS;
    };

    is_remote = ((p_txmsg->flags & AM_CAN_REMOTE_FLAG) ? AM_TRUE : AM_FALSE);
    is_extern = ((p_txmsg->flags & AM_CAN_EXTERN_FLAG) ? AM_TRUE : AM_FALSE);

    /* 8 字节数据 */
    if (p_txmsg->msglen > 8) {
        p_txmsg->msglen = 8;
    }

    /* 设置 数据长度*/
    reg_sff &= (uint32_t)0xffffffff;
    reg_sff |= p_txmsg->msglen ;
}
```

```

/* 远程标志 */
if (is_remote) {
    reg_sff |= (1 << 6);
} else {
    reg_sff &= ~(1 << 6);
}

/* 扩展标识符 */
if (is_extern) {
    /* 设置扩展标识符 ID 和 IDE 位 */
    reg_sff |= (1 << 7);
    msg_id = ((p_txmsg->id & 0xFFFFFFFF) << 3);
    p_hw_can->tx_rx_mail.txid[0] = (msg_id >> 24) & 0xff;
    p_hw_can->tx_rx_mail.txid[1] = (msg_id >> 16) & 0xff;
    p_hw_can->tx_rx_mail.txdata[0] = (msg_id >> 8) & 0xff;
    p_hw_can->tx_rx_mail.txdata[1] = (msg_id >> 0) & 0xff;

} else {
    /* 设置标准标识符 ID 和 IDE 位 */
    reg_sff &= ~(1 << 7);
    msg_id = ((p_txmsg->id & 0x7ff) << 21);
    p_hw_can->tx_rx_mail.txid[0] = ((msg_id >> 24) & 0xFF);
    p_hw_can->tx_rx_mail.txid[1] = ((msg_id >> 16) & 0xE0);
}

/* 必须一次性写入，中间不能读 */
p_hw_can->tx_rx_mail.sff = reg_sff;

/* 数据帧写如数据 */
if (!is_remote) {
    if (is_extern) {
        for(i = 0; i < p_txmsg->msglen; i++) {
            p_hw_can->tx_rx_mail.txdata[i+2] = (uint32_t)p_txmsg->msgdata[i];
        }
    } else {
        for(i = 0; i < p_txmsg->msglen; i++) {
            p_hw_can->tx_rx_mail.txdata[i] = (uint32_t)p_txmsg->msgdata[i];
        }
    }
}

/* 请求传输 */
amhw_zmf2x6_can_trans_start(p_hw_can);

```

```

return AM_CAN_NOERROR;
}

```

14. __can_zmf2x6_msg_rcv

该函数几乎是 CAN 消息发送的一个逆操作，zmf2x6 的发送接收相关寄存器是共用的，因此只需对特定寄存器进行读操作，并填充参数 p_rxmsg 即可。详细代码见程序清单 17.35。

程序清单 17.35 CAN 消息接收

```

/**
 * \brief 接收信息
 */
am_can_err_t __can_zmf2x6_msg_rcv (void *p_drv, am_can_message_t *p_rxmsg)
{
    am_zmf2x6_can_dev_t      *p_dev      = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t        *p_hw_can = NULL;
    am_bool_t                is_extern = AM_FALSE;

    if (NULL == p_drv || NULL == p_rxmsg) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    /* 判断接收缓冲区是否为空 */
    if (amhw_zmf2x6_can_get_flag_status(p_hw_can, AMHW_ZMF2X6_CAN_RXFIFO_STA) == 0) {
        return AM_CAN_NO_AVAIL_CHANNELS;
    }

    /* 初始化 */
    p_rxmsg->id      = 0;
    p_rxmsg->flags    = 0;
    p_rxmsg->msglen = 0;

    if (p_hw_can->tx_rx_mail.sff & (1 << 7)) {

        /* 使用扩展标识符 */
        p_rxmsg->flags |= AM_CAN_EXTERN_FLAG;
        is_extern = AM_TRUE;
        /* 获取 ID */
        p_rxmsg->id = ((p_hw_can->tx_rx_mail.txdata[1] & 0xff) >> 3)|
                    ((p_hw_can->tx_rx_mail.txdata[0] & 0xff) << 5)|
                    ((p_hw_can->tx_rx_mail.txid[1]   & 0xff) << 13)|
                    ((p_hw_can->tx_rx_mail.txid[0]   & 0xff) << 21);
    }
}

```

```

    } else {
        /* 使用标准标识符 */

        /* 获取 ID */
        p_rxmsg->id = ((p_hw_can->tx_rx_mail.txid[1] & 0xff) >> 5)|
            ((p_hw_can->tx_rx_mail.txid[0] & 0xff) << 3);
    }

    /* 远程帧处理 */
    if (p_hw_can->tx_rx_mail.sff & (1 << 6)) {

        /* 远程帧 */
        p_rxmsg->flags |= AM_CAN_REMOTE_FLAG;
        p_rxmsg->msglen = 0;
    }

    /* 获取发送数据长度 */
    p_rxmsg->msglen = (uint8_t)0x0f & p_hw_can->tx_rx_mail.sff;

    /* 获取数据 */
    if (is_extern) {
        for(int i = 0; i < p_rxmsg->msglen; i++) {
            p_rxmsg->msgdata[i] = (uint8_t)p_hw_can->tx_rx_mail.txdata[i+2];
        }
    } else {
        for(int i = 0; i < p_rxmsg->msglen; i++) {
            p_rxmsg->msgdata[i] = (uint8_t)p_hw_can->tx_rx_mail.txdata[i];
        }
    }

    /* 释放 FIFO */
    amhw_zmf2x6_can_cmd_set(p_hw_can, AMHW_ZMF2X6_CAN_CMD_REC_BUF_CLEAR);

    return AM_CAN_NOERROR;
}

```

15. __can_zmf2x6_stop_msg_snd

程序清单 17.36 为发送消息停止函数，此处只需调用 hw 层编写的 amhw_zmf2x6_can_trans_stop 函数即可，实际是操作了 CMD 寄存器的传输停止位。

程序清单 17.36 发送消息停止函数

```
/**
```



```

* \brief 停止发送信息
*/
am_can_err_t __can_zmf2x6_stop_msg_snd (void *p_drv)
{
    am_zmf2x6_can_dev_t    *p_dev          = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t      *p_hw_can       = NULL;
    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    /* 检查发送缓冲区是否空闲 */
    if(amhw_zmf2x6_can_get_flag_status(p_hw_can, AMHW_ZMF2X6_CAN_TX_BUFF_STA) == 0) {
        return AM_CAN_NO_AVAIL_CHANNELS;
    };

    amhw_zmf2x6_can_trans_stop(p_hw_can);

    return AM_CAN_NOERROR;
}

```

16. __can_zmf2x6_filter_tab_ext_set

滤波表配置函数, 根据 zmf2x6 CAN 模块的特性, 其滤波表能够配置为 16 位或者 32 位, 因此我们需分开分析并分别建立函数 __can_zmf2x6_filter_32bit_idmask、__can_zmf2x6_filter_16bit_idmask 来进行处理。同时它具有 19 个滤波器能够被控制, 因此我们需要使用 for 循环依次遍历需要配置的滤波器。为简单起见所有的滤波器都被配置为一种模式, 即要么都是 16 位要么都是 32 位。详细内容见程序清单 17.37 所示。

程序清单 17.37 设置滤波函数

```

/**
* \brief 设置滤波函数(扩展)
*/
am_can_err_t __can_zmf2x6_filter_tab_ext_set (void          *p_drv,
                                              am_can_filter_t *p_filterbuff,
                                              size_t          lenth)
{
    am_zmf2x6_can_dev_t    *p_dev    = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_filter_t *p_filter = NULL;
    amhw_zmf2x6_can_t      *p_hw_can = NULL;
    amhw_zmf2x6_canflt_t    *p_hw_filcan = NULL;
    uint8_t                 filt_num = 0;
}

```

```

am_can_err_t          state    = AM_CAN_NOERROR;
amhw_zmf2x6_canflt_group_t *pflt_group;

if (NULL == p_drv || NULL == p_filterbuff || 0 == lenth || (lenth > 20)) {
    return AM_CAN_INVALID_PARAMETER;
}

p_hw_filcan = (amhw_zmf2x6_canflt_t *)p_dev->p_devinfo->fil_regbase;
p_filter = (amhw_zmf2x6_canfilter_t *)p_dev->p_devinfo->filter;
p_hw_can = (amhw_zmf2x6_cant_t *)p_dev->p_devinfo->regbase;

/* 选择滤波器位宽模式 */
if(p_dev->p_devinfo->filter->can_filter_scale == 1){
    /* 32 */
    p_hw_can->mcr |= (1 << 3); /* 滤波器 0 的模式控制位 */
    p_hw_filcan->afm[0] = 0xf;
    p_hw_filcan->afm[1] = 0xf;
    p_hw_filcan->afm[2] = 0xf;
} else {
    /* 16 */
    p_hw_can->mcr &= ~(1 << 3); /* 滤波器 0 的模式控制位 */
    p_hw_filcan->afm[0] = 0x0;
    p_hw_filcan->afm[1] = 0x0;
    p_hw_filcan->afm[2] = 0x0;
}

if (AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT != p_dev->mode) {
    /* 滤波器寄存器只在初始化模式下可进行访问，设置前进入初始化模式 */
    amhw_zmf2x6_can_operating_mode_request(p_hw_can,
        AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT);
}

for (int i = 0; i < lenth; i++) {

    /* 滤波器组号 */
    filt_num = i;

    if (filt_num > 19) {
        /* 跳出当前 */
        state = AM_CAN_ILLEGAL_CHANNEL_NO;
        break;
    }

    /* 禁用滤波器 */

```

```

    amhw_zmf2x6_can_fil_group_disable(p_hw_filcan, filt_num);

    /* 获取当前配置滤波组地址 */
    if (filt_num == 0) {
        pflt_group = ((amhw_zmf2x6_canflt_group_t *)&p_hw_filcan->fi_rx0);
    } else {
        pflt_group = ((amhw_zmf2x6_canflt_group_t *)&p_hw_filcan->fi_rx[filt_num-1]);
    }

    if (p_filter->can_filter_scale) {
        /* 32 位，配置 ID 及 ID 掩码 */
        __can_zmf2x6_filter_32bit_idmask(pflt_group, p_filterbuff);
    } else {
        /* 16 位，配置 ID 及 ID 掩码 */
        __can_zmf2x6_filter_16bit_idmask(pflt_group, p_filterbuff);
    }

    /* 使能滤波器 */
    amhw_zmf2x6_can_fil_group_enable(p_hw_filcan, filt_num);

    /* 滤波器组后移 */
    p_filterbuff++;
}

if (p_dev->mode != AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT) {
    /* 切换到正常模式 */
    amhw_zmf2x6_can_operating_mode_request(p_hw_can,
        p_dev->mode);
}

return state;
}

static void __can_zmf2x6_filter_32bit_idmask(amhw_zmf2x6_canflt_group_t *pflt_group,
        am_can_filter_t *p_filterbuff)
{
    uint32_t idCodeTemp1, idMaskTemp1;

    if (p_filterbuff->ide == AM_CAN_FRAME_TYPE_STD) {
        idCodeTemp1 = (p_filterbuff->id[0]) << (32 - 11);
        idMaskTemp1 = (p_filterbuff->mask[0]) << (32 - 11);
        idMaskTemp1 |= 0x1FFFFFFF; // Other shielding positions 1
    } else {
        idCodeTemp1 = (p_filterbuff->id[0]) << (32 - 29);
        idMaskTemp1 = (p_filterbuff->mask[0]) << (3);
    }
}

```

```

        idMaskTemp1 |= 0x7; //Other shielding positions 1
    }

    pflt_group->acr[0] = ( idCodeTemp1 >> 24 ) & 0xff;
    pflt_group->acr[1] = ( idCodeTemp1 >> 16 ) & 0xff;
    pflt_group->acr[2] = ( idCodeTemp1 >> 8 ) & 0xff;
    pflt_group->acr[3] = ( idCodeTemp1 >> 0 ) & 0xff;

    pflt_group->amr[0] = (idMaskTemp1 >> 24 ) & 0xff;
    pflt_group->amr[1] = (idMaskTemp1 >> 16 ) & 0xff;
    pflt_group->amr[2] = (idMaskTemp1 >> 8 ) & 0xff;
    pflt_group->amr[3] = (idMaskTemp1 >> 0 ) & 0xff;
}

static void __can_zmf2x6_filter_16bit_idmask(amhw_zmf2x6_canflt_group_t *pflt_group,
                                             amcan_filter_t *pfilterbuff)
{
    uint32_t idCodeTemp1, idMaskTemp1, idCodeTemp2, idMaskTemp2;

    idCodeTemp1 = (pfilterbuff->id[0]);
    idMaskTemp1 = (pfilterbuff->mask[0]);
    idCodeTemp2 = (pfilterbuff->id[1]);
    idMaskTemp2 = (pfilterbuff->mask[1]);
    if (pfilterbuff->ide == AM_CAN_FRAME_TYPE_STD) {
        idCodeTemp1 = (pfilterbuff->id[0]) << (32 - 11);
        idMaskTemp1 = (pfilterbuff->mask[0]) << (32 - 11);
        idCodeTemp2 = (pfilterbuff->id[1]) << (32 - 11);
        idMaskTemp2 = (pfilterbuff->mask[1]) << (32 - 11);
        idMaskTemp1 |= 0x1FFFFFFF;
        idMaskTemp2 |= 0x1FFFFFFF;
    } else {
        idCodeTemp1 = (pfilterbuff->id[0]) << (32 - 29);
        idMaskTemp1 = (pfilterbuff->mask[0]) << (32 - 29);
        idCodeTemp2 = (pfilterbuff->id[0]) << (32 - 29);
        idMaskTemp2 = (pfilterbuff->mask[0]) << (32 - 29);
        idMaskTemp1 |= 0x1FFFFFFF;
        idMaskTemp2 |= 0x1FFFFFFF;
    }

    pflt_group->acr[0] = ( idCodeTemp1 >> 24 ) & 0xff;
    pflt_group->acr[1] = ( idCodeTemp1 >> 16 ) & 0xff;
    pflt_group->amr[0] = (idMaskTemp1 >> 24 ) & 0xff;
    pflt_group->amr[1] = (idMaskTemp1 >> 16 ) & 0xff;
}

```

```

pflt_group->acr[2] = ( idCodeTemp2 >> 24 ) & 0xff;
pflt_group->acr[3] = ( idCodeTemp2 >> 16 ) & 0xff;
pflt_group->amr[2] = (idMaskTemp2 >> 24) & 0xff;
pflt_group->amr[3] = (idMaskTemp2 >> 16) & 0xff;
}

```

17. __can_zmf2x6_filter_tab_ext_get

程序清单 17.38 为获取滤波表配置，实际使用频率较低，因此并未实现。

程序清单 17.38 获取滤波函数

```

/**
 * \brief 获取滤波函数
 */
am_can_err_t __can_zmf2x6_filter_tab_ext_get (void *p_drv,
                                              am_can_filter_t *p_filterbuff,
                                              size_t *p_lenth)
{
    return AM_CAN_NOERROR;
}

```

18. 中断服务函数

该函数为 CAN 的全局中断服务函数，单独定义了一个函数 __can_zmf2x6_intcb_func_run 用于调用对应中断所定义的回调函数。

中断服务函数中主要涉及到的是对中断标志位的检测，判断是何种中断产生，并做出相应的响应，相关中断类型见。如发送、接收、总线错误、错误被动、仲裁丢失在此处均只需要调用其相应注册的回调函数，而对于数据溢出中断还进行了一个 FIFO 释放的操作。代码详情见程序清单 17.39。

程序清单 17.39 CAN 全局中断服务函数

```

static void __can_zmf2x6_intcb_func_run (am_zmf2x6_can_dev_t *p_dev,
                                         am_can_int_type_t can_int)
{
    uint32_t temp = (uint32_t)can_int;
    am_pfnvoid_t intcb_fun = NULL;
    void *intcb_arg = NULL;
    uint8_t i = 0;

    if ((NULL == p_dev) || (AM_CAN_INT_ALL == can_int)) {
        return ;
    }
}

```

```

while (temp >> 1) {
    i++;
    temp = temp >> 1;
}

intcb_fun = p_dev->p_devinfo->p_intcb_info[i].pfn_callback;
intcb_arg = p_dev->p_devinfo->p_intcb_info[i].p_arg;

if (NULL != intcb_fun) {
    intcb_fun(intcb_arg);
}
}

/**
 * \brief can 中断服务函数
 */
static void __can_zmf2x6_int_handle (void *p_drv)
{
    am_zmf2x6_can_dev_t    *p_dev      = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t      *p_hw_can   = NULL;

    if (NULL == p_drv) {
        return;
    }
    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    ir_sta_reg = p_hw_can->ir;

    /* 接收中断产生，并且 rxfifo 不为空 */
    if ((ir_sta_reg & AMHW_ZMF2X6_CAN_RI_FLAG) &&
        amhw_zmf2x6_can_get_flag_status(p_hw_can, AMHW_ZMF2X6_CAN_RXFIFO_STA)) {

        /* 执行接收回调函数 */
        __can_zmf2x6_intcb_func_run(p_dev, AM_CAN_INT_RX);

        /* */
        amhw_zmf2x6_can_clear_int_flag(p_hw_can, AMHW_ZMF2X6_CAN_RI_FLAG);
    }

    /* 发送中断产生，并且发送完成 */
    if ((ir_sta_reg & AMHW_ZMF2X6_CAN_TI_FLAG) &&
        amhw_zmf2x6_can_get_flag_status(p_hw_can, AMHW_ZMF2X6_CAN_TX_BUFF_STA)) {

        /* 执行发送回调函数 */
        __can_zmf2x6_intcb_func_run(p_dev, AM_CAN_INT_TX);
    }
}

```

```

    amhw_zmf2x6_can_clear_int_flag (p_hw_can, AMHW_ZMF2X6_CAN_TI_FLAG);
}
/* 数据溢出中断产生 */
if ((ir_sta_reg & AMHW_ZMF2X6_CAN_DOI_FLAG)) {

    /* 执行数据溢出中断回调函数 */
    __can_zmf2x6_intcb_func_run(p_dev, AM_CAN_INT_DATAOVER);

    /* 释放 FIFO 在中断函数中处理或者 在此处理 FIFO 清空 FIFO 邮箱 */
    amhw_zmf2x6_can_clear_int_flag(p_hw_can, AMHW_ZMF2X6_CAN_DOI_FLAG);
    amhw_zmf2x6_can_cmd_set(p_hw_can, AMHW_ZMF2X6_CAN_CMD_RXFIFO_CLEAR);
    amhw_zmf2x6_can_cmd_set(p_hw_can, AMHW_ZMF2X6_CAN_CMD_DATA_OVER_CLEAR);
}
/* 总线错误中断 */
if ((ir_sta_reg & AMHW_ZMF2X6_CAN_BEI_FLAG)) {

    /* 总线错误中断回调函数 */
    __can_zmf2x6_intcb_func_run(p_dev, AM_CAN_INT_BUS_OFF);

    amhw_zmf2x6_can_clear_int_flag (p_hw_can,
                                    AMHW_ZMF2X6_CAN_BEI_FLAG);
}
/* 错误被动中断 */
if ((ir_sta_reg & AMHW_ZMF2X6_CAN_EPI_FLAG)) {

    /* 错误被动中断回调函数 */
    __can_zmf2x6_intcb_func_run(p_dev, AM_CAN_INT_WARN);

    amhw_zmf2x6_can_clear_int_flag (p_hw_can,
                                    AMHW_ZMF2X6_CAN_EPI_FLAG);
}
/* 仲裁丢失中断 */
if ((ir_sta_reg & AMHW_ZMF2X6_CAN_ALI_FLAG)) {

    /* 仲裁丢失中断回调函数 */
    __can_zmf2x6_intcb_func_run(p_dev, AM_CAN_INT_ERROR_PASSIVE);

    amhw_zmf2x6_can_clear_int_flag (p_hw_can,
                                    AMHW_ZMF2X6_CAN_ALI_FLAG);
}

#endif
}

```

19. __can_zmf2x6_status_get

程序清单 17.40 为 CAN 状态获取函数，该函数主要包含 CAN 所有可能产生的状态，包括中断状态、错误状态等。

程序清单 17.40 CAN 状态获取函数

```
/**
 * \brief CAN 状态
 */
am_can_err_t __can_zmf2x6_status_get (void          *p_drv,
                                       am_can_int_type_t *p_int_type,
                                       am_can_bus_err_t *p_bus_err)
{
    am_zmf2x6_can_dev_t    *p_dev    = (am_zmf2x6_can_dev_t *)p_drv;
    amhw_zmf2x6_can_t      *p_hw_can = NULL;
    uint8_t                err_reg;

    if (NULL == p_drv || NULL == p_int_type || NULL == p_bus_err) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    *p_int_type = 0;

    /* 接收中断 */
    if (ir_sta_reg & AMHW_ZMF2X6_CAN_RI_FLAG) {

        *p_int_type |= AM_CAN_INT_RX;
    }

    /* 发送中断 */
    if (ir_sta_reg & AMHW_ZMF2X6_CAN_TI_FLAG){
        *p_int_type |= AM_CAN_INT_TX;
    }

    /* 数据溢出中断 */
    if ((ir_sta_reg & AMHW_ZMF2X6_CAN_DOI_FLAG)) {

        *p_int_type |= AM_CAN_INT_DATAOVER;
    }

    /* 错误中断、总线错误中断、总线仲裁丢失中断 */
    if ((ir_sta_reg & AMHW_ZMF2X6_CAN_BEI_FLAG) ||
        (ir_sta_reg & AMHW_ZMF2X6_CAN_EPI_FLAG) ||
```



```

(ir_sta_reg & AMHW_ZMF2X6_CAN_ALI_FLAG)) {

    *p_int_type |= AM_CAN_INT_ERROR;

    /* 判断是否总线错误 */
    if ((ir_sta_reg & AMHW_ZMF2X6_CAN_BEI_FLAG)) {
        *p_int_type |= AM_CAN_INT_BUS_OFF;
    }

    /* 判断是否是错误被动中断 */
    if ((ir_sta_reg & AMHW_ZMF2X6_CAN_EPI_FLAG)) {
        *p_int_type |= AM_CAN_INT_WARN;
    }

    /* 判断是否是因为仲裁丢失中断 */
    if ((ir_sta_reg & AMHW_ZMF2X6_CAN_ALI_FLAG)) {
        *p_int_type |= AM_CAN_INT_ERROR_PASSIVE;
    }
}

err_reg = amhw_zmf2x6_can_err_code_get(p_hw_can);
/* 从寄存器获取错误代码 */
*p_bus_err = 0;

/* 格式错误 */
if (err_reg & AMHW_ZMF2X6_CAN_GEHA_ERR) {
    *p_bus_err |= AM_CAN_BUS_ERR_FORM;
}

/* 填充错误 */
if (err_reg & AMHW_ZMF2X6_CAN_FILL_ERR) {
    *p_bus_err |= AM_CAN_BUS_ERR_STUFF;
}

/* ACK 错误 */
if ((err_reg & 0x1f) == 0x19 || (err_reg & 0x1f) == 0x1b) {
    *p_bus_err |= AM_CAN_BUS_ERR_ACK;
}

/* CRC 错误 */
if ((err_reg & 0x1f) == (1 << 3) || (err_reg & 0x1f) == (3 << 3)){
    *p_bus_err |= AM_CAN_BUS_ERR_CRC;
}

```

```

return AM_CAN_NOERROR;
}

```

20. __can_zmf2x6_connect

CAN 全局中断连接函数，此处实现的是对全局中断的连接，其参数 pfn_isr 能够传递一个回调函数用于控制全局中断，不过我们在前面已经编写了全局中断的回调函数。因此该参数直接传递 NULL 即可，默认连接到程序清单 17.41 所定义的回调函数。

程序清单 17.41 CAN 全局中断连接函数

```

/**
 * \brief 中断连接
 */
am_can_err_t __can_zmf2x6_connect (void      *p_drv,
                                   am_pfnvoid_t pfn_isr,
                                   void      *p_arg)
{
    am_zmf2x6_can_dev_t *p_dev = (am_zmf2x6_can_dev_t *)p_drv;

    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    if (NULL != pfn_isr) {
        /* 连接传入函数 */
        am_int_connect(INUM_CAN, pfn_isr, p_arg);
    } else {
        /* 连接默认函数 */
        am_int_connect(INUM_CAN, __can_zmf2x6_int_handle, p_dev);
    }

    return AM_CAN_NOERROR;
}

```

21. __can_zmf2x6_disconnect

程序清单 17.42 为 CAN 全局中断断开连接函数，在这里除了断开连接外还同时禁能了全局中断。

程序清单 17.42 CAN 全局中断断开连接函数

```

/**
 * \brief 删除中断连接
 */
am_can_err_t __can_zmf2x6_disconnect (void      *p_drv,
                                       am_pfnvoid_t pfn_isr,

```

```

                                void        *p_arg)
{
    am_zmf2x6_can_dev_t    *p_dev    = (am_zmf2x6_can_dev_t *)p_drv;

    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    /* 中断禁能 */
    am_int_disable(INUM_CAN);
    if (NULL != pfn_isr) {
        /* 删除连接函数 */
        am_int_disconnect(INUM_CAN, pfn_isr, p_arg);
    } else {
        /* 删除连接函数 */
        am_int_disconnect(INUM_CAN, __can_zmf2x6_int_handle, p_dev);
    }

    return AM_CAN_NOERROR;
}

```

22. __can_zmf2x6_intcb_connect

程序清单 17.43 为中断回调函数连接函数，该函数用于对 CAN 模块的特定中断进行连接，实现方法很简单，就是将传递过来的各类型回调函数规则的存入到中断回调函数结构体内。

程序清单 17.43 中断回调函数连接函数

```

/**
 * \brief 中断回调函数的连接
 */
am_can_err_t __can_zmf2x6_intcb_connect (void        *p_drv,
                                           am_can_int_type_t  inttype,
                                           am_pfnvoid_t      pfn_callback,
                                           void        *p_arg)
{
    am_zmf2x6_can_dev_t    *p_dev    = (am_zmf2x6_can_dev_t *)p_drv;
    uint8_t    i = 0;

    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    while (inttype >> 1) {
        i++;
    }
}

```

```

        inttype = inttype >> 1;
    }

    if (i >= p_dev->p_devinfo->p_intcb_num) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_dev->p_devinfo->p_intcb_info[i].pfn_callback = pfn_callback;
    p_dev->p_devinfo->p_intcb_info[i].p_arg      = p_arg;

    return AM_CAN_NOERROR;
}

```

23. __can_zmf2x6_intcb_disconnect

程序清单 17.44 为中断回调函数断开连接函数，与上面的连接中断回调函数是一个相反的过程。

程序清单 17.44 中断回调函数断开连接函数

```

/**
 * \brief 删除中断回调函数的连接
 */
am_can_err_t __can_zmf2x6_intcb_disconnect (void          *p_drv,
                                             am_can_int_type_t inttype)
{
    am_zmf2x6_can_dev_t *p_dev = (am_zmf2x6_can_dev_t *)p_drv;
    uint8_t i = 0;

    if (NULL == p_drv) {
        return AM_CAN_INVALID_PARAMETER;
    }

    while (inttype >> 1) {
        i++;
        inttype = inttype >> 1;
    }

    if (i >= p_dev->p_devinfo->p_intcb_num) {
        return AM_CAN_INVALID_PARAMETER;
    }

    p_dev->p_devinfo->p_intcb_info[i].pfn_callback = NULL;
    p_dev->p_devinfo->p_intcb_info[i].p_arg      = NULL;

    return AM_CAN_NOERROR;
}

```

}

24. am_zmf2x6_can_alc_get

该函数不属于标准接口范畴，为 zmf2x6 特有，由于其功能较为重要，因此单独列出函数。该函数用于对仲裁丢失捕捉，可有效的判断出仲裁失败的具体位置，帮助程序进行相关判断，具体代码如程序清单 17.45 所示。

程序清单 17.45 仲裁丢失捕捉函数

```
/**
 * \brief CAN 仲裁丢失捕捉 数据对应含义见用户手册
 *
 * counter :| 0~10 | 11 | 12 | 13~30 | 31 |
 *
 * 仲裁丢失在: | bit 1~11 | SRTR | IDE | bit 12~29 | RTR |
 */
uint8_t am_zmf2x6_can_alc_get(am_can_handle_t handle)
{
    amhw_zmf2x6_can_t *p_hw_can = NULL;
    am_zmf2x6_can_dev_t *p_dev = (am_zmf2x6_can_dev_t *)handle->p_drv;
    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    return amhw_zmf2x6_can_alc_get(p_hw_can);
}
```

25. am_zmf2x6_can_err_code_get

该函数不属于标准接口范畴，为 zmf2x6 特有，由于其功能较为重要，因此单独列出函数。该函数可用于对 CAN 模块具体错误的分析，帮助应用程序更好的定位相关错误，具体代码如程序清单 17.46 所示。

程序清单 17.46 错误码获取函数

```
/**
 * \brief 包含错误代码、方向错误、段错误，详见用户手册
 */
uint8_t am_zmf2x6_can_err_code_get(am_can_handle_t handle)
{
    amhw_zmf2x6_can_t *p_hw_can = NULL;
    am_zmf2x6_can_dev_t *p_dev = (am_zmf2x6_can_dev_t *)handle->p_drv;
    p_hw_can = (amhw_zmf2x6_can_t *)p_dev->p_devinfo->regbase;

    return amhw_zmf2x6_can_err_code_get(p_hw_can);
}
```

17.4 用户配置文件编写

17.4.1 定义设备常量于设备信息常量

创建配置文件后，我们需要在里面加入设备变量，设备变量只需要定义，不用初始化。设备信息常量定义时初始化，默认填充的信息是一种常用的配置方式，CAN 设备信息配置

如程序清单 17.47 所示。

程序清单 17.47 CAN 设备信息配置文件

```
#define MAX      10

static am_zmf2x6_can_intcb_info_t    __g_can_intcb_info[MAX];

amhw_zmf2x6_can_filter_t __g_can_filter = {

    /* 滤波器位宽 */
    1,

};

am_zmf2x6_can_init_info_t __g_can_initfo = {

    /** \brief 0 正常模式 1 只听模式 2 自检测模式 */
    AMHW_ZMF2X6_CAN_MODE_NORMAL,

    /** \brief 总线采样 1: 3 倍 0: 1 倍 */
    0,

    /** \brief 错误报警限制[0~7] */
    96,

};

/** \brief 设备信息 */
static const am_zmf2x6_can_devinfo_t __g_can_devinfo = {

    ZMF2X6_CAN_BASE,                /**< \brief CAN */
    (ZMF2X6_CAN_BASE + 0x40),
    INUM_CAN,                        /**< \brief CAN 的中断编号 此处不可修改*/
    __g_can_intcb_info,             /**< \brief 回调信息 */
    MAX,                            /**< \brief 回调信息内存大小*/
    __zmf2x6_plfm_can_init,         /**< \brief CAN1 的平台初始化 */
    __zmf2x6_plfm_can_deinit,      /**< \brief CAN1 的平台去初始化 */
    &__g_can_filter,
    &__g_can_initfo

};

/**< \brief 设备实例 */
static am_zmf2x6_can_dev_t    __g_can_dev;
```

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需

要为设备提供平台初始化函数和解初始化函数。平台初始化函数和解初始化函数定义详见程序清单 17.49。

程序清单 17.48 平台初始化、解初始化函数

```
/** \brief CAN 平台初始化 */
static void __zmf2x6_plfm_can_init (void)
{
    am_zmf2x6_clk_reset(CLK_CAN);
    am_clk_enable(CLK_CAN);

    am_gpio_pin_cfg(PIOB_8, PIOB_8_CAN1_RX | PIOB_8_INPUT_FLOAT);
    am_gpio_pin_cfg(PIOB_9, PIOB_9_CAN1_TX | PIOB_9_AF_PP | PIOB_9_SPEED_50MHz );
}

/** \brief 解除 CAN 平台初始化 */
static void __zmf2x6_plfm_can_deinit (void)
{
    am_clk_disable (CLK_CAN);
}
```

17.4.2 添加平台初始化与平台解初始化函数

添加平台初始化与平台解初始化函数，如程序清单 17.49 所示。

程序清单 17.49 平台初始化函数和解初始化函数定义

```
/** \brief CAN 实例初始化，获得 CAN 标准服务句柄 */
am_can_handle_t am_zmf2x6_can_inst_init (void)
{
    return am_zmf2x6_can_init(&__g_can_dev, &__g_can_devinfo);
}

/** \brief CAN 实例解初始化 */
void am_zmf2x6_can_inst_deinit (am_can_handle_t handle)
{
    am_zmf2x6_can_deinit(handle);
}
```

17.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在标准层实例公共头文件 am_zmf2x6_inst_init.h 中加入它们的声明，如程序清单 17.50 所示。

程序清单 17.50 平台初始化解初始化声明

```
/**
 * \brief CAN 实例初始化，获得 CAN 标准服务句柄
 *
 * \param 无
```

```

*
* \return CAN 标准服务句柄，若为 NULL，表明初始化失败
*/
am_can_handle_t am_zmf2x6_can_inst_init(void);
/**
* \brief CAN 实例解初始化
*
* \param[in] handle 通过 am_zmf2x6_can_inst_init() 函数获得的 CAN 标准服务句柄
*
* \return 无
*/
void am_zmf2x6_can_inst_deinit(am_can_handle_t handle);

```

用户使用时，应该先调用平台初始化函数 `am_zmf2x6_can_inst_init` 函数，初始化 CAN，并保存其返回的 `handle`，后续直接使用保存的 `handle` 作为标准接口的第一个参数使用标准接口函数即可。

至此，用户配置文件开发完毕。

17.5 例程文件开发

参考 5.5.1 章节新建用户例程文件，搭建文件的基本框架，准备 CAN 例程内容的编写。例程文件开发主要分为两个部分，HW 例程和 STD 例程。

17.5.1 例程内容开发

1. HW 层实现

这里使用 HW 层来实现 CAN 的轮询传输。主要涉及到 CAN 硬件初始化函数定义、滤波表初始化函数定义、`can` 启动函数定义、发送接收函数定义。为提高案例的普适性该案例分两个文件编写，相关代码如程序清单 9.60 程序清单 17.52 所示。

程序清单 17.51 CAN 轮询发送接收 HW 层 DEMO

```

#include "ametal.h"
#include "am_can.h"
#include "am_zmf2x6.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_zmf2x6_can.h"
#include "hw/amhw_zmf2x6_can.h"
#include "demo_zmf2x6_entries.h"

#define __ZMF2X6_CAN_STD      0
#define __ZMF2X6_CAN_EXT     1
#define __ZMF2X6_CAN_BIT_16  0
#define __ZMF2X6_CAN_BIT_32  1

/** \brief 滤波表参数结构体 */
typedef struct __can_filter {

```



```

uint8_t   ide;      /**< \brief 滤波器帧类型  1: 扩展帧  0: 标准帧  */
uint8_t   bw;      /**< \brief 滤波器位数   1: 32     0: 16     */

uint32_t id[2];    /**< \brief ID 验收码   */
uint32_t mask[2];  /**< \brief ID 掩码     */

} __can_filter_t;

static void __can_zmf2x6_filter_32bit(amhw_zmf2x6_canflt_group_t  *pflt_group,
                                     __can_filter_t              can_filter)
{
    uint32_t idCodeTemp1, idMaskTemp1;

    if (can_filter.ide == __ZMF2X6_CAN_STD) {
        /* 标准帧 */
        idCodeTemp1 = (can_filter.id[0]) << (32 - 11);
        idMaskTemp1 = (can_filter.mask[0]) << (32 - 11);
        idMaskTemp1 |= 0x1FFFFFFF; // Other shielding positions 1
    } else {
        /* 扩展帧 */
        idCodeTemp1 = (can_filter.id[0]) << (32 - 29);
        idMaskTemp1 = (can_filter.mask[0]) << (3);
        idMaskTemp1 |= 0x7; // Other shielding positions 1
    }

    pflt_group->acr[0] = ( idCodeTemp1 >> 24 ) & 0xff;
    pflt_group->acr[1] = ( idCodeTemp1 >> 16 ) & 0xff;
    pflt_group->acr[2] = ( idCodeTemp1 >> 8 ) & 0xff;
    pflt_group->acr[3] = ( idCodeTemp1 >> 0 ) & 0xff;

    pflt_group->amr[0] = (idMaskTemp1 >> 24 ) & 0xff;
    pflt_group->amr[1] = (idMaskTemp1 >> 16 ) & 0xff;
    pflt_group->amr[2] = (idMaskTemp1 >> 8 ) & 0xff;
    pflt_group->amr[3] = (idMaskTemp1 >> 0 ) & 0xff;
}

static void __can_zmf2x6_filter_16bit(amhw_zmf2x6_canflt_group_t  *pflt_group,
                                     __can_filter_t              can_filter)
{
    uint32_t idCodeTemp1, idMaskTemp1, idCodeTemp2, idMaskTemp2;

    idCodeTemp1 = (can_filter.id[0]);
    idMaskTemp1 = (can_filter.mask[0]);

```

```

idCodeTemp2 = (can_filter.id[1]);
idMaskTemp2 = (can_filter.mask[1]);
if (can_filter.ide == __ZMF2X6_CAN_STD) {
    /* 标准帧 */
    idCodeTemp1 = (can_filter.id[0]) << (32 - 11);
    idMaskTemp1 = (can_filter.mask[0]) << (32 - 11);
    idCodeTemp2 = (can_filter.id[1]) << (32 - 11);
    idMaskTemp2 = (can_filter.mask[1]) << (32 - 11);
    idMaskTemp1 |= 0x1FFFFFFF;
    idMaskTemp2 |= 0x1FFFFFFF;
} else {
    idCodeTemp1 = (can_filter.id[0]) << (32 - 29);
    idMaskTemp1 = (can_filter.mask[0]) << (32 - 29);
    idCodeTemp2 = (can_filter.id[0]) << (32 - 29);
    idMaskTemp2 = (can_filter.mask[0]) << (32 - 29);
    idMaskTemp1 |= 0x1FFFFFFF;
    idMaskTemp2 |= 0x1FFFFFFF;
}
pflt_group->acr[0] = (idCodeTemp1 >> 24) & 0xff;
pflt_group->acr[1] = (idCodeTemp1 >> 16) & 0xff;
pflt_group->amr[0] = (idMaskTemp1 >> 24) & 0xff;
pflt_group->amr[1] = (idMaskTemp1 >> 16) & 0xff;

pflt_group->acr[2] = (idCodeTemp2 >> 24) & 0xff;
pflt_group->acr[3] = (idCodeTemp2 >> 16) & 0xff;
pflt_group->amr[2] = (idMaskTemp2 >> 24) & 0xff;
pflt_group->amr[3] = (idMaskTemp2 >> 16) & 0xff;
}

/**
 * \brief 设置滤波函数
 * \param[in] p_hw_can      : 指向 CAN 寄存器块的指针
 * \param[in] filt_num      : 滤波组数
 * \param[in] can_filter    : 滤波器相关参数
 * \return 无
 */
am_can_err_t __can_zmf2x6_filter_set(amhw_zmf2x6_can_t *p_hw_can,
                                     uint8_t      filt_num,
                                     __can_filter_t can_filter)
{
    amhw_zmf2x6_canflt_t *p_hw_filcan;
    am_can_err_t state = AM_CAN_NOERROR;
    amhw_zmf2x6_canflt_group_t *pflt_group;
    if (filt_num > 19 || p_hw_can == NULL) {

```

```

        return AM_CAN_INVALID_PARAMETER;
    }

    /* 获取滤波表配置的起始地址，即 CAN 地址偏移 0x40 */
    p_hw_filcan = (amhw_zmf2x6_canflt_t*)((int)p_hw_can+0x40);

    /* 滤波器寄存器只在初始化模式下可进行访问，设置前进入初始化模式 */
    amhw_zmf2x6_can_operating_mode_request(p_hw_can,
        AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT);

    /* 选择滤波器位宽模式 */
    amhw_zmf2x6_can_fil_group_mode_set(p_hw_can, p_hw_filcan, filt_num, can_filter.bw);

    /* 禁用滤波器 */
    amhw_zmf2x6_can_fil_group_disable(p_hw_filcan, filt_num);

    /* 获取当前配置滤波组地址 */
    if (filt_num == 0) {
        pflt_group = ((amhw_zmf2x6_canflt_group_t*)&p_hw_filcan->fi_rx0);
    } else {
        pflt_group = ((amhw_zmf2x6_canflt_group_t*)&p_hw_filcan->fi_rx[filt_num-1]);
    }

    if (can_filter.bw == __ZMF2X6_CAN_BIT_32) {
        /* 32 位，配置 ID 及 ID 掩码 */
        __can_zmf2x6_filter_32bit(pflt_group, can_filter);
    } else {
        /* 16 位，配置 ID 及 ID 掩码 */
        __can_zmf2x6_filter_16bit(pflt_group, can_filter);
    }

    /* 使能滤波器 */
    amhw_zmf2x6_can_fil_group_enable(p_hw_filcan, filt_num);

    /* 切换到正常模式 */
    amhw_zmf2x6_can_operating_mode_request(p_hw_can,
        AMHW_ZMF2X6_CAN_OPERATING_NORMAL);

    return state;
}

/**
 * \brief 接收信息
 */
static am_can_err_t __can_zmf2x6_msg_rx (amhw_zmf2x6_can_t *p_hw_can, am_can_message_t *p_rxmsg)

```

```

{
    am_bool_t      is_extern = AM_FALSE;

    if (NULL == p_rxmsg) {
        return AM_CAN_INVALID_PARAMETER;
    }

    /* 判断接收缓冲区是否为空 */
    if (amhw_zmf2x6_can_get_flag_status(p_hw_can, AMHW_ZMF2X6_CAN_RXFIFO_STA) == 0) {
        return AM_CAN_NO_AVAIL_CHANNELS;
    }

    /* 初始化 */
    p_rxmsg->id      = 0;
    p_rxmsg->flags    = 0;
    p_rxmsg->msglen = 0;

    if (p_hw_can->tx_rx_mail.sff & (1 << 7)) {

        /* 使用扩展标识符 */
        p_rxmsg->flags |= AM_CAN_EXTERN_FLAG;
        is_extern = AM_TRUE;
        /* 获取 ID */
        p_rxmsg->id = ((p_hw_can->tx_rx_mail.txdata[1] & 0xff) >> 3)|
                    ((p_hw_can->tx_rx_mail.txdata[0] & 0xff) << 5)|
                    ((p_hw_can->tx_rx_mail.txid[1]   & 0xff) << 13)|
                    ((p_hw_can->tx_rx_mail.txid[0]   & 0xff) << 21);

    } else {

        /* 使用标准标识符 */

        /* 获取 ID */
        p_rxmsg->id = ((p_hw_can->tx_rx_mail.txid[1] & 0xff) >> 5)|
                    ((p_hw_can->tx_rx_mail.txid[0] & 0xff) << 3);
    }

    /* 远程帧处理 */
    if (p_hw_can->tx_rx_mail.sff & (1 << 6)) {

        /* 远程帧 */
        p_rxmsg->flags |= AM_CAN_REMOTE_FLAG;
        p_rxmsg->msglen = 0;
    }
}

```

```

    /* 获取发送数据长度*/
    p_rxmsg->msglen = (uint8_t)0x0f & p_hw_can->tx_rx_mail.sff;

    /* 获取数据 */
    if (is_extern) {
        for(int i = 0; i < p_rxmsg->msglen; i++) {
            p_rxmsg->msgdata[i] = (uint8_t)p_hw_can->tx_rx_mail.txdata[i+2];
        }
    } else {
        for(int i = 0; i < p_rxmsg->msglen; i++) {
            p_rxmsg->msgdata[i] = (uint8_t)p_hw_can->tx_rx_mail.txdata[i];
        }
    }

    /* 释放 FIFO */
    amhw_zmf2x6_can_cmd_set(p_hw_can, AMHW_ZMF2X6_CAN_CMD_REC_BUF_CLEAR);

    return AM_CAN_NOERROR;
}

/**
 * \brief 发送信息
 */
static am_can_err_t __can_zmf2x6_msg_tx (amhw_zmf2x6_can_t *p_hw_can, am_can_message_t *p_txmsg)
{
    am_bool_t          is_remote = AM_FALSE;
    am_bool_t          is_extern = AM_FALSE;
    uint32_t           reg_sff = 0;
    uint32_t           msg_id = 0;

    uint8_t i = 0;

    if (NULL == p_txmsg) {
        return AM_CAN_INVALID_PARAMETER;
    }

    /* 检查发送缓冲区是否空闲 */
    if(amhw_zmf2x6_can_get_flag_status(p_hw_can, AMHW_ZMF2X6_CAN_TX_BUFF_STA) == 0) {
        return AM_CAN_NO_AVAIL_CHANNELS;
    };

    is_remote = ((p_txmsg->flags & AM_CAN_REMOTE_FLAG) ? AM_TRUE : AM_FALSE);

```

```

is_extern = ((p_txmsg->flags & AM_CAN_EXTERN_FLAG) ? AM_TRUE : AM_FALSE);

/* 8 字节数据 */
if (p_txmsg->msglen > 8) {
    p_txmsg->msglen = 8;
}
/* 设置 数据长度*/
reg_sff &= (uint32_t)0xffffffff;
reg_sff |= p_txmsg->msglen;

/* 远程标志 */
if (is_remote) {
    reg_sff |= (1 << 6);
} else {
    reg_sff &= ~(1 << 6);
}

/* 扩展标识符 */
if (is_extern) {
    /* 设置扩展标识符 ID 和 IDE 位 */
    reg_sff |= (1 << 7);
    msg_id = ((p_txmsg->id & 0x1FFFFFFF) << 3);
    p_hw_can->tx_rx_mail.txid[0] = (msg_id >> 24) & 0xff;
    p_hw_can->tx_rx_mail.txid[1] = (msg_id >> 16) & 0xff;
    p_hw_can->tx_rx_mail.txdata[0] = (msg_id >> 8) & 0xff;
    p_hw_can->tx_rx_mail.txdata[1] = (msg_id >> 0) & 0xff;

} else {
    /* 设置标准标识符 ID 和 IDE 位 */
    reg_sff &= ~(1 << 7);
    msg_id = ((p_txmsg->id & 0x7ff) << 21);
    p_hw_can->tx_rx_mail.txid[0] = ((msg_id >> 24) & 0xFF);
    p_hw_can->tx_rx_mail.txid[1] = ((msg_id >> 16) & 0xE0);
}

/* 必须一次性写入，中间不能读 */
p_hw_can->tx_rx_mail.sff = reg_sff;

/* 数据帧写如数据 */
if (!is_remote) {
    if (is_extern) {
        for(i = 0; i < p_txmsg->msglen; i++) {
            p_hw_can->tx_rx_mail.txdata[i+2] = (uint32_t)p_txmsg->msgdata[i];
        }
    }
}

```

```

        }
    } else {
        for(i = 0; i < p_txmsg->msglen; i++) {
            p_hw_can->tx_rx_mail.txdata[i] = (uint32_t)p_txmsg->msgdata[i];
        }
    }
}

/* 请求传输 */
amhw_zmf2x6_can_trans_start(p_hw_can);

return AM_CAN_NOERROR;
}

static void __can_init (amhw_zmf2x6_can_t *p_hw_can, am_can_bps_param_t *p_can_baudrate)
{
    /* 退出睡眠 请求进入初始化模式 */
    amhw_zmf2x6_can_operating_mode_request(
        p_hw_can, AMHW_ZMF2X6_CAN_OPERATING_MODE_INIT);
    amhw_zmf2x6_can_peli_mode_set(p_hw_can);
    /* 设置 波特率分频器 同步跳转宽度和相位段 */
    amhw_zmf2x6_can_brp_set(p_hw_can, p_can_baudrate->brp);
    amhw_zmf2x6_can_sjw_set(p_hw_can, p_can_baudrate->sjw);
    amhw_zmf2x6_can_tseg1_set(p_hw_can, p_can_baudrate->tesg1);
    amhw_zmf2x6_can_tseg2_set(p_hw_can, p_can_baudrate->tesg2);
    amhw_zmf2x6_can_sam_set(p_hw_can, 0);

    amhw_zmf2x6_can_test_mode_request(p_hw_can, AMHW_ZMF2X6_CAN_MODE_NORMAL);
    p_hw_can->ewlr = 0x96;
}

static void __can_start(amhw_zmf2x6_can_t *p_hw_can)
{
    p_hw_can->mcr &= ~AMHW_ZMF2X6_CAN_MCR_INRQ;
}

/**
 * \brief CAN 例程入口
 *
 * \return 无
 */
void demo_zmf2x6_hw_can_entry (amhw_zmf2x6_can_t *p_hw_can, am_can_bps_param_t *can_btr_baud)
{
    uint8_t num = 0;

```

```

am_can_message_t can_message;
__can_init(p_hw_can, can_btr_baud);
__can_zmf2x6_filter_set(p_hw_can, 0, (__can_filter_t){
    __ZMF2X6_CAN_EXT, __ZMF2X6_CAN_BIT_32, {0x1}, {0x0}} );
__can_zmf2x6_filter_set(p_hw_can, 1, (__can_filter_t){
    __ZMF2X6_CAN_STD, __ZMF2X6_CAN_BIT_32, {0x2}, {0x0}} );
__can_start(p_hw_can);

can_message.id = 0x321;
can_message.flags = AM_CAN_EXTERN_FLAG ;
can_message.msglen = 8;
for(int i = 0; i < can_message.msglen; i++) {
    can_message.msgdata[i] = i;
}
while(1){
    num++;
    __can_zmf2x6_msg_rx(p_hw_can, &can_message);
    if(num > 100) {
        am_kprintf("can recv id: 0x%x\r\n", can_message.id);
        /* 不打印远程帧时的数据缓存 */
        if ((can_message.flags & AM_CAN_REMOTE_FLAG) != AM_CAN_REMOTE_FLAG) {
            for (int i = 0; i < can_message.msglen; i++) {
                am_kprintf("data: 0x%x \r\n", can_message.msgdata[i]);
            }
        }
        __can_zmf2x6_msg_tx(p_hw_can, &can_message);
        num = 0;
    }
    am_mdelay(10);
}
}

/** [src_test_flexcan] */

```

程序清单 17.52 CAN 轮询发送接收硬件层案例入口

```

#include "ametal.h"
#include "am_can.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_zmf2x6.h"
#include "am_zmf2x6_clk.h"
#include "am_zmf2x6_can.h"
#include "am_zmf2x6_inst_init.h"

```



```

#include "demo_zmf2x6_entries.h"

void demo_zmf2x6_hw_can_entry (amhw_zmf2x6_can_t *p_hw_can, am_can_bps_param_t *can_btr_baud);

#define APB1_CLK          am_clk_rate_get(CLK_APB1) /*CAN 的输入时钟 */
#define CAN_BAUD          250000 /*CAN 的波特率 250k*/

/**
 * \brief 获取对应波特率配置值 CAN_BAUD = APB1/((BRP + 1) * (TEG1 + TEG2 + 3))
 */
static void __can_auto_cfg_baud(am_can_bps_param_t *can_btr_baud,
                                uint32_t          src_clk,
                                uint32_t          baud )
{
    uint32_t i,value = baud,record = 1;
    uint32_t remain = 0,sum_prescaler = 0;
    while(( baud == 0 )||( src_clk == 0 ));
    sum_prescaler = src_clk / baud;
    sum_prescaler = sum_prescaler / 2;
    for ( i = 25; i > 3; i-- ) {
        remain = sum_prescaler - ((sum_prescaler / i)*i);
        if( remain == 0 ) {
            record = i;
            break;
        } else {
            if (remain < value) {
                value = remain;
                record = i;
            }
        }
    }
}

/* 设置重新同步跳跃宽度为 2 个时间单位 */
can_btr_baud->sjw = 0;
can_btr_baud->brp = (sum_prescaler/record) - 1;
can_btr_baud->tesg2 = (record - 3) / 3;
can_btr_baud->tesg1 = (record - 3) - can_btr_baud->tesg2;

}

/**
 * \brief CAN 历程入口
 *
 * \return 无
 */
void demo_zmf2x6_core_hw_can_entry (void)

```

```

{
    am_can_bps_param_t  can_btr_baud = {0};

    AM_DBG_INFO("demo zmf2x6 core hw can!\r\n");
    am_zmf2x6_clk_reset(CLK_CAN);
    am_clk_enable(CLK_CAN);

    am_gpio_pin_cfg(PIOB_8, PIOB_8_CAN1_RX | PIOB_8_INPUT_FLOAT);
    am_gpio_pin_cfg(PIOB_9, PIOB_9_CAN1_TX | PIOB_9_AF_PP);

    __can_auto_cfg_baud(&can_btr_baud, APB1_CLK, CAN_BAUD);

    demo_zmf2x6_hw_can_entry ((amhw_zmf2x6_can_t *)ZMF2X6_CAN_BASE, &can_btr_baud);
}

/** [src_test_flexcan] */

```

2. 标准层实现

使用标准层实现 CAN 的轮询发送接收案例，由于使用的是标准层例程，因此直接调用标准的例程函数 `demo_std_can_entry` 即可。具体操作如程序清单 17.53 所示。

程序清单 17.53 CAN 的多通道采集标准层 DEMO

```

#include "ametal.h"
#include "am_can.h"
#include "am_clk.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_zmf2x6.h"
#include "am_zmf2x6_inst_init.h"
#include "demo_std_entries.h"

#define APB1_CLK      am_clk_rate_get(CLK_APB1) /*CAN 的输入时钟 */
#define CAN_BAUD      250000 /*CAN 的波特率 250k*/

/**
 * \brief 获取对应波特率配置值  CAN_BAUD = APB1/((BRP + 1) * (TEG1 + TEG2 + 3))
 */
static void __can_auto_cfg_baud(am_can_bps_param_t *can_btr_baud,
                                uint32_t          src_clk,
                                uint32_t          baud )
{
    uint32_t i,value = baud,record = 1;
    uint32_t remain = 0,sum_prescaler = 0;
    while(( baud == 0 )||( src_clk == 0 ));

```

```

sum_prescaler = src_clk / baud;
sum_prescaler = sum_prescaler / 2;
for ( i = 25; i > 3; i-- ) {
    remain = sum_prescaler - ((sum_prescaler / i)*i);
    if( remain == 0 ) {
        record = i;
        break;
    } else {
        if (remain < value) {
            value = remain;
            record = i;
        }
    }
}

/* 设置重新同步跳跃宽度为 2 个时间单位 */
can_btr_baud->sjw = 0;
can_btr_baud->brp = (sum_prescaler/record) - 1;
can_btr_baud->tesg2 = (record - 3) / 3;
can_btr_baud->tesg1 = (record - 3) - can_btr_baud->tesg2;

}

/**
 * \brief CAN 历程入口
 *
 * \return 无
 */
void demo_zmf2x6_core_std_can_entry (void)
{
    am_can_bps_param_t  can_btr_baud = {0};

    AM_DBG_INFO("demo zmf2x6_core std can!\r\n");

    __can_auto_cfg_baud(&can_btr_baud, APB1_CLK, CAN_BAUD);

    demo_std_can_entry (am_zmf2x6_can_inst_init (), &can_btr_baud);
}

/** [src_test_flexcan] */

```

第18章 特殊外设驱动开发——DMA

18.1 特殊外设驱动开发流程

与通用外设开发比较，特殊外设驱动开发没有标准层。这类驱动的开发流程同样需从开发硬件层开始，然后实现驱动层函数、配置文件，最后提供示例程序并测试其功能。本章以 DMA 外设为例，展示如何开发特殊外设驱动代码。

18.2 HW 层开发

参考 5.2.1 章节，创建对应目录和文件，添加硬件层基本的内容。

18.2.1 添加寄存器列表

HW 层就是与寄存器打交道的，因此我们先列出寄存器，方便后续操作。作为开发人员，我们都应该在这之前先整体阅读该外设的描述，以确保 HW 层接口的完整和准确性，对我们开发驱动了解该外设的工作原理都是一种帮助。

对照 ZMF2x6 用户手册的 DMA 章节，可以看到，DMA 外设寄存器，简单描述如下：

表 18.1 DMA 寄存器概览

位	标识符	含义
1	DMA_ISR	DMA 中断状态寄存器
2	DMA_IFCR	DMA 中断标志清除寄存器
3	DMA_CCRx(x = 1...5)	DMA 通道 x 配置寄存器
4	DMA_CNDTRx(x = 1...5)	DMA 通道 x 传输数量寄存器
5	DMA_CPARx(x = 1...5)	DMA 通道 x 外设地址寄存器
6	DMA_CMARx(x = 1...5)	DMA 通道 x 存储器地址寄存器

到此 DMA 的各个寄存器及其功能基本阅读完毕，我们也大致了解了各个寄存器功能以及要配置各个功能，其他寄存器位该如何配合等。下面我们先定义一个寄存器的结构体，结构体成员全部小写，如程序清单 18.1 所示。

程序清单 18.1 寄存器结构体定义

```
/**
 * \brief structure of DMA controller register
 */
typedef struct amhw_zmf2x6_dma_chan {
    __IO uint32_t dma_isr;          /**< \brief DMA interrupt status register */
    __IO uint32_t dma_ifcr;        /**< \brief DMA interrupt flag clear register */
    struct {
        __IO uint32_t dma_ccr;      /**< \brief DMA channel configuration register */
        __IO uint32_t dma_cndtr;    /**< \brief DMA channel Number of data to transfer */
        __IO uint32_t dma_cpar;     /**< \brief DMA channel peripheral address */
        __IO uint32_t dma_cmar;     /**< \brief DMA channel memory address */
        uint32_t reserved[1];       /**< \brief Reserved */
    }
} chcfg[AMHW_ZMF2X6_DMA_CHAN_CNT]; /**< \brief DMA Channel Struct */
```

```
} amhw_zmf2x6_dma_t;
```

这个小节我们定义了 DMA 的寄存器列表，分析了各个寄存器的功能。编写寄存器相关的函数，我们分为两个阶段，第一阶段，先写出每个寄存器的功能操作函数即可，第二阶段站在用户的角度，看看是否需要提供同时操作多个寄存器的函数，我们先进行第一阶段函数编写，第二阶段留在下一个小节。

所有的内联函数都应该在定义时标识为内联函数，在函数的上方加入 `am_static_inline` 的关键字，具体见下文的示例。

18.2.2 编写寄存器的相关函数

1. DMA_ISR（中断状态寄存器）

中断状态寄存器包括通道传输错误标志、通道半传输标志、通道传输完成标志和通道全局中断标志，根据需要的功能编写相应的操作函数。

为了函数参数更加直观和明确，采用宏定义将参数列出，如程序清单 18.2 所示。

程序清单 18.2 DMA_ISR 操作函数及参数宏

```
/**
 * \brief DMA interrupt status
 * @{
 */
/** \brief Channel x transfer error flag */
#define AMHW_ZMF2X6_DMA_CHAN_TX_ERR_FLAG(n)      (1u << ((n) * 4 + 3))

/** \brief Channel x half transfer flag */
#define AMHW_ZMF2X6_DMA_CHAN_TX_HALF_FLAG(n)      (1u << ((n) * 4 + 2))

/** \brief Channel x transfer complete flag */
#define AMHW_ZMF2X6_DMA_CHAN_TX_COMP_FLAG(n)      (1u << ((n) * 4 + 1))

/** \brief Channel x global interrupt flag */
#define AMHW_ZMF2X6_DMA_CHAN_GLOBAL_INT_FLAG(n)   (1u << ((n) * 4 + 0))
/** @} */

/**
 * \brief Interrupt flag check
 *
 * \param[in] p_hw_dma      : The pointer to the structure of DMA register
 * \param[in] flag          : see the filed of DMA interrupt status
 *
 * \return AM_TRUE or AM_FALSE
 */
am_static_inline
am_bool_t amhw_zmf2x6_dma_chan_stat_check(amhw_zmf2x6_dma_t *p_hw_dma,
                                           uint32_t flag)
```

```
{
    return ((p_hw_dma->dma_isr & flag) ? AM_TRUE : AM_FALSE);
}
```

2. DMA_IFCR（中断标志清除寄存器）

中断标志清除寄存器用于清除中断状态寄存器对应位，操作函数如程序清单 18.3 所示。

程序清单 18.3 DMA_IFCR 操作函数

```
/**
 * \brief Interrupt flag clear
 *
 * \param[in] p_hw_dma    : The pointer to the structure of DMA register
 * \param[in] flag        : see the filed of DMA interrupt status
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_dma_chan_flag_clear(amhw_zmf2x6_dma_t *p_hw_dma,
                                     uint32_t          flag)
{
    p_hw_dma->dma_ifcr |= flag;
}
```

3. DMA_CCRx（通道 x 配置寄存器）

通道 x 配置寄存器包括各种模式、中断、通道优先级、外设和存储器数据宽度及地址增量、通道使能等设置，操作函数如程序清单 18.4 所示。

程序清单 18.4 DMA_CCRx 操作函数

```
/**
 * \brief DMA channel configuration
 *
 * \param[in] p_hw_dma    : The pointer to the structure of DMA register
 * \param[in] flag        : see the filed of DMA configuration
 * \param[in] channel     : DMA channel
 *
 * \return none
 */
am_static_inline
void amhw_zmf2x6_dma_chan_config_set(amhw_zmf2x6_dma_t *p_hw_dma,
                                     uint32_t          flag,
                                     uint8_t           channel)
{
    p_hw_dma->chcfg[channel].dma_ccr = (p_hw_dma->chcfg[channel].dma_ccr & (~0x7ffe)) | flag;
}

/**
```

```

* \brief DMA channel Interrupt enable
*
* \param[in] p_hw_dma    : The pointer to the structure of DMA register
* \param[in] flag        : see the filed of DMA channel Interrupt flag
* \param[in] channel     : DMA channel
*
* \return none
*/
am_static_inline
void amhw_zmf2x6_dma_chan_int_enable(amhw_zmf2x6_dma_t *p_hw_dma,
                                     uint32_t          flag,
                                     uint8_t            channel)
{
    p_hw_dma->chcfg[channel].dma_ccr |= flag;
}

/**
* \brief DMA channel Interrupt disable
*
* \param[in] p_hw_dma    : The pointer to the structure of DMA register
* \param[in] flag        : see the filed of DMA channel Interrupt flag
* \param[in] channel     : DMA channel
*
* \return none
*/
am_static_inline
void amhw_zmf2x6_dma_chan_int_disable(amhw_zmf2x6_dma_t *p_hw_dma,
                                       uint32_t          flag,
                                       uint8_t            channel)
{
    p_hw_dma->chcfg[channel].dma_ccr &= ~flag;
}

/**
* \brief DMA channel  enable
*
* \param[in] p_hw_dma    : The pointer to the structure of DMA register
* \param[in] channel     : DMA channel
* \param[in] flag        : AM_FALSE or AM_TRUE
*
* \return none
*/
am_static_inline
void amhw_zmf2x6_dma_chan_enable(amhw_zmf2x6_dma_t *p_hw_dma,

```

```

                                uint8_t      channel,
                                am_bool_t     flag)
{
    p_hw_dma->chcfg[channel].dma_ccr = (p_hw_dma->chcfg[channel].dma_ccr & ~(1u << 0)) |
                                flag << 0;
}

```

4. DMA_CNDTRx（通道 x 传输数量寄存器）

通道 x 传输数量寄存器用于设置数据传输数量，操作函数如程序清单 18.5 所示。

程序清单 18.5 DMA_CNDTRx 操作函数

```

/**
 * \brief The number of data to transfer
 *
 * \param[in] p_hw_dma    : The pointer to the structure of DMA register
 * \param[in] size        : 0~65535
 * \param[in] channel     : DMA channel
 *
 * \note This register can be modified when DMA channel is disable(DMA_CCRx[EN] = 0)
 * \return none
 */
am_static_inline
void amhw_zmf2x6_dma_chan_tran_data_size(amhw_zmf2x6_dma_t *p_hw_dma,
                                uint32_t      size,
                                uint8_t      channel)
{
    p_hw_dma->chcfg[channel].dma_cndtr = size;
}

/**
 * \brief The number of data to transfer
 *
 * \param[in] p_hw_dma The pointer to the structure of DMA register
 * \param[in] channel  DMA channel
 *
 * \return size
 *
 * \note This register can be modified when DMA channel is disable(DMA_CCRx[EN] = 0)
 */
am_static_inline
uint16_t amhw_zmf2x6_dma_chan_tran_data_get(amhw_zmf2x6_dma_t *p_hw_dma,
                                uint8_t      channel)
{
    return p_hw_dma->chcfg[channel].dma_cndtr;
}

```


5. DMA_CPARx（通道 x 外设地址寄存器）

通道 x 外设地址寄存器用于设置外设数据寄存器的基地址，操作函数如程序清单 18.6 所示。

程序清单 18.6 DMA_CPARx 操作函数

```
/**
 * \brief Peripheral address set
 *
 * \param[in] p_hw_dma    : The pointer to the structure of DMA register
 * \param[in] address     : peripheral address
 * \param[in] channel     : DMA channel
 *
 * \note This register can be modified when DMA channel is disable(DMA_CCRx[EN] = 0)
 * \return none
 */
am_static_inline
void amhw_zmf2x6_dma_chan_per_address_set(amhw_zmf2x6_dma_t *p_hw_dma,
                                          uint32_t          address,
                                          uint8_t           channel)
{
    p_hw_dma->chcfg[channel].dma_cpar = address;
}
```

6. DMA_CMARx（通道 x 存储器地址寄存器）

通道 x 存储器地址寄存器用于设置作为数据传输的源或目标的存储器地址，操作函数如程序清单 18.7 所示。

程序清单 18.7 DMA_CMARx 操作函数

```
/**
 * \brief Memory address set
 *
 * \param[in] p_hw_dma    : The pointer to the structure of DMA register
 * \param[in] address     : Memory address
 * \param[in] channel     : DMA channel
 *
 * \note This register can be modified when DMA channel is disable(DMA_CCRx[EN] = 0)
 * \return none
 */
am_static_inline
void amhw_zmf2x6_dma_chan_mem_address_set(amhw_zmf2x6_dma_t *p_hw_dma,
                                          uint32_t          address,
                                          uint8_t           channel)
{
    p_hw_dma->chcfg[channel].dma_cmar = address;
}
```

18.2.3 添加其他功能性函数

在每个寄存器的操作函数编写完毕后，可能 HW 层的接口还有所欠缺，需要提供一些方便用户使用的简单的功能性函数。这些函数可能一时间想不到，可以暂时不用考虑，直接进入驱动层的开发。在后续如果发现 HW 层欠缺哪些接口，可以反过来再增加。当然，也有可能 HW 层接口没有什么欠缺了，那样就无需再开发其它函数了。

至此，整个 HW 层的开发工作就结束了，如果后续开发中发现缺少某些函数，还可以再添加。

18.3 驱动层开发

参考 5.3.1 章节，新建 am_zmf2x6_dma.h 和 am_zmf2x6_dma.c 驱动文件，搭建相应的文件框架，并按规则放到对应的路径下。

18.3.1 编写结构性程序

驱动层首先应该搭建好相关结构，再去实现某些功能函数。

1. 定义相关结构体

在 AMetal 中，每个设备（DMA 就是一个设备）都应该有与之对应的设备结构体和设备信息结构体。设备结构体主要用于内存分配，与设备相关的变量都应该定义在设备结构体中，而不是使用全局变量。设备信息结构体主要包含该设备的一些相关信息。

● 设备信息结构体定义

通常情况下，设备信息至少包含一个 HW 层定义的指向寄存器基址的指针类型成员，用于驱动层使用 HW 层的接口。如果设备需要使用中断，还需要填入中断号信息。除此之外，一些设备的初始化配置信息，也可以由设备信息提供。

可以暂时定义设备信息结构体如程序清单 18.8 所示。后续可以根据实际情况修改。

程序清单 18.8 DMA 设备信息结构体定义

```
/** \brief DMA 设备信息 */
typedef struct am_zmf2x6_dma_devinfo {

    /** \brief 指向 DMA 寄存器块的指针 */
    uint32_t    dma_reg_base;

    /** \brief DMA 中断号开始 */
    uint8_t     inum_start;

    /** \brief DMA 中断号结束 */
    uint8_t     inum_end;

    /** \brief 平台初始化函数，如打开时钟，配置引脚等工作 */
    void        (*pfn_plfm_init)(void);

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

```
} am_zmf2x6_dma_devinfo_t;
```

● 设备结构体定义

由于 DMA 设备无需提供标准服务，为了该设备在后续使用中，可以通过设备访问到设备信息，也需要包含一个指向设备信息的指针。

暂时定义设备信息如程序清单 18.9 所示，后续如果需要使用到什么设备相关的全局变量，就应该定义在这里。

程序清单 18.9 DMA 设备结构体定义

```
/**
 * \brief DMA 设备实例
 */
typedef struct am_zmf2x6_dma_dev {

    /** \brief 指向 DMA 设备信息的指针 */
    const am_zmf2x6_dma_devinfo_t *p_devinfo;

}am_zmf2x6_dma_dev_t;
```

2. 搭建函数框架

函数提供给用户使用的驱动层函数。

● 驱动函数

开发者需要对特殊外设的功能有较清晰的理解，了解该外设可以实现哪些功能以及需要实现什么功能。DMA 驱动层需提供建立传输描述符、配置传输通道、开启和停止通道传输、连接和删除回调函数，还有初始化函数和解初始化函数，以使用户使用。

初始化函数的参数是设备指针和设备信息常量指针，返回值为初始化状态。

解初始化函数主要是当不需要再使用某个外设时，释放掉其占用的相关资源，无参数，无返回值。示例代码如程序清单 18.10 所示。

程序清单 18.10 驱动层函数定义示例

```
/**
 * \brief 开始通道传输
 *
 * \param[in] chan :DMA 通道号，值为：DMA_CHAN_*(#DMA_CHAN_1) 或
(#DMA_CHAN_UART1_TX)
 *
 * \retval AM_OK : 配置成功
 * \retval AM_ERROR : 配置错误，该通道之前配置过
 */
int am_zmf2x6_dma_chan_start (int chan);

/**
 * \brief 停止通道传输
 * \param[in] chan :DMA 通道号，值为：DMA_CHAN_*(#DMA_CHAN_1) 或
(#DMA_CHAN_UART1_TX)
```

```

*
* \retval AM_OK    : 操作成功
*/
int am_zmf2x6_dma_chan_stop (int chan);

/**
* \brief 建立传输描述符
*
* \param[in] p_desc    : 指向 DMA 传输描述符的指针
* \param[in] src_addr : 源端首地址
* \param[in] dst_addr : 目标端首地址
* \param[in] nbytes   : 传输字节数
* \param[in] flags    : 配置参数
*
* \retval  AM_OK      : 建立成功
* \retval -AM_EINVAL  : 未对齐, 无效参数
*/
int am_zmf2x6_dma_xfer_desc_build (amhw_zmf2x6_dma_xfer_desc_t *p_desc,
                                   uint32_t src_addr,
                                   uint32_t dst_addr,
                                   uint32_t nbytes,
                                   uint32_t flags);

/**
* \brief 配置 DMA 传输通道
*
* \param[in] p_desc : 指向 DMA 传输描述符的指针
* \param[in] type   : DMA 传输模式设置 : 有对应的枚举体 dma_transfer_type_t
* \param[in] chan   : DMA 通道号, 值为: DMA_CHAN_* (#DMA_CHAN_1) 或
(#DMA_CHAN_UART1_TX)
*
* \retval  AM_OK      : 操作成功
* \retval -AM_EINVAL : 参数无效
*/
int am_zmf2x6_dma_xfer_desc_chan_cfg (amhw_zmf2x6_dma_xfer_desc_t *p_desc,
                                       amhw_zmf2x6_dma_transfer_type_t type,
                                       uint8_t chan);

/**
* \brief 连接 DMA 回调函数
*
* \attention 该回调函数的第二个参数可从驱动获得, 该参数的取值范围是 AM_ZMF2X6_DMA_INT*
*            (#AM_ZMF2X6_DMA_INT_ERROR)或(#AM_ZMF2X6_DMA_INT_NORMAL)
*

```

```

* \param[in] chan      : DMA 通道号，值为：DMA_CHAN_* (#DMA_CHAN_1) 或
(DMA_CHAN_UART1_TX)
* \param[in] pfn_isr : 回调函数指针
* \param[in] p_arg    : 回调函数的第一个入口参数，在连接过程中，对应通道写上对应数字，
*                      例如 DMA0 通道，则该参数为(void *)0
*
* \retval AM_OK       : 连接成功
* \retval -AM_EPERM   : 参数错误
*/
int am_zmf2x6_dma_isr_connect (int chan,
                               am_zmf2x6_pfn_dma_isr_t pfn_isr,
                               void *p_arg);

/**
* \brief 删除 DMA 回调函数的连接
*
* \attention 该回调函数的第二个参数可从驱动获得，该参数的取值范围是 AM_ZMF2X6_DMA_INT*
*            (#AM_ZMF2X6_DMA_INT_ERROR)或(#AM_ZMF2X6_DMA_INT_NORMAL)
*            该回调函数的第三个参数是产生中断的通道，其取值范围是 DMA_CHAN_*
*            (#DMA_CHAN_1)
*
* \param[in] chan      : DMA 通道号，值为：DMA_CHAN_* (#DMA_CHAN_1) 或
(DMA_CHAN_UART1_TX)
* \param[in] pfn_isr : 回调函数指针
* \param[in] p_arg    : 回调函数的第一个入口参数，在连接过程中，对应通道写上对应数字，
*                      例如 DMA0 通道，则该参数为(void *)0
*
* \retval AM_OK       : 删除成功
* \retval -AM_EPERM   : 参数错误
*/
int am_zmf2x6_dma_isr_disconnect (int chan,
                                   am_zmf2x6_pfn_dma_isr_t pfn_isr,
                                   void *p_arg);

/**
* \brief 获取 某通道当前传输剩余的字节数
*
* \param[in] chan DMA 通道号，值为 DMA_CHAN_* (#DMA_CHAN_1) 或 (#DMA_CHAN_UART1_TX)
*
* \retval 字节数
*/
uint16_t am_zmf2x6_dma_tran_data_get (int chan);

/**

```

```

* \brief 设置 某通道当前传输剩余的字节数
*
* \param[in] chan          DMA 通道号, 值为 DMA_CHAN_* (#DMA_CHAN_1) 或
(#DMA_CHAN_UART1_TX)
* \param[in] trans_data_byte 要传输的字节数
*
* \retval 无
*
* \note 在设置传输的字节数之前必须停止通道传输
*/
void am_zmf2x6_dma_tran_data_size (int chan, uint32_t trans_data_byte);

/**
* \brief DMA 初始化
*
* \param[in] p_dev          : 指向 DMA 设备的指针
* \param[in] p_dma_devinfo : 指向 DMA 设备信息的指针
*
* \retval AM_OK            : 初始化成功
*/
int am_zmf2x6_dma_init (am_zmf2x6_dma_dev_t      *p_dev,
                        const am_zmf2x6_dma_devinfo_t *p_dma_devinfo);

/**
* \brief DMA 去初始化
*
* \return 无
*/
void am_zmf2x6_dma_deinit (void);

```

至此, 整个函数结构就搭建好了, 接下来就是简单的填充工作了。可以编译一下, 应该没有错误的。如果有错误, 表明结构没有搭建好。可能存在警告, 毕竟有些变量没有使用, 可以暂时不用理会。做到这里, 其实驱动层已经开发一大半了。

18.3.2 驱动函数实现

通过上面的函数框架搭建, 共计 10 个函数, 均面向用户, 下面逐步实现。

1. am_zmf2x6_dma_init 函数

该函数主要完成两个功能:

- a) 初始化设备, 将成员赋值。
- b) 初始化硬件相关设置, 该函数的示例代码如程序清单 18.11 所示。

程序清单 18.11 am_zmf2x6_dma_init 函数示例代码

```

/* DMA 初始化 */
int am_zmf2x6_dma_init (am_zmf2x6_dma_dev_t      *p_dev,
                        const am_zmf2x6_dma_devinfo_t *p_devinfo)

```

```

{
    uint32_t i = 0;

    if (p_dev == NULL || p_devinfo == NULL) {
        return AM_ERROR;
    }

    p_dev->p_devinfo = p_devinfo;
    __gp_dma_dev     = p_dev;

    if (p_devinfo->pfn_plfm_init) {
        p_devinfo->pfn_plfm_init();
    }

    for (i = 0; i < AMHW_ZMF2X6_DMA_CHAN_CNT; i++) {
        __dma_int_info[i].pfn_isr = NULL;
        __dma_int_map[i]         = __INT_NOT_CONNECTED;
    }

    for (i = p_devinfo->inum_start; i <= p_devinfo->inum_end; i++) {
        am_int_connect(i, __dma_int_handler, (void *)0);
        am_int_enable(i);
    }

    return AM_OK;
}

```

2. am_zmf2x6_dma_deinit 函数

该函数主要用于资源释放，包括内存资源和硬件资源。由于驱动中全部采用静态内存分配，因此无内存资源可以释放。

因此，该函数将几个不再使用的变量置为 NULL，并且断开中断连接。示例代码如程序清单 18.12 所示。

程序清单 18.12 am_zmf2x6_dma_deinit 函数示例代码

```

/* DMA 去初始化 */
void am_zmf2x6_dma_deinit(void)
{
    am_zmf2x6_dma_devinfo_t *p_dma_devinfo = NULL;
    amhw_zmf2x6_dma_t       *p_hw_dma;

    int i = 0;

    if (NULL == __gp_dma_dev) {
        return;
    }
}

```

```

p_dma_devinfo = (am_zmf2x6_dma_devinfo_t *)__gp_dma_dev->p_devinfo;
p_hw_dma      = (amhw_zmf2x6_dma_t *)p_dma_devinfo->dma_reg_base;

for (i = 0; i < AMHW_ZMF2X6_DMA_CHAN_CNT; i++) {
    __dma_int_info[i].pfn_isr = NULL;
    __dma_int_map[i]         = 0;
    amhw_zmf2x6_dma_chan_enable(p_hw_dma, i, AM_FALSE);
}

for (i = p_dma_devinfo->inum_start; i <= p_dma_devinfo->inum_end; i++) {
    am_int_disconnect(i,
                      __dma_int_handler,
                      (void *)0);

    am_int_disable(i);
}

if (p_dma_devinfo->pfn_plfm_deinit) {
    p_dma_devinfo->pfn_plfm_deinit();
}
}

```

3. __dma_int_handler 函数

该函数为 DMA 中断处理函数，功能实现代码详见程序清单 18.13。

程序清单 18.13 __dma_int_handler 函数示例代码

```

/* DMA 中断处理函数 */
static void __dma_int_handler (void *p_arg)
{
    __DMA_DEVINFO_DECL(p_dma_devinfo, __gp_dma_dev);
    amhw_zmf2x6_dma_t *p_hw_dma = (amhw_zmf2x6_dma_t *)p_dma_devinfo->dma_reg_base;

    int      i;
    int      chan = 0xFF;
    uint8_t  flag = 0xFF;

    am_zmf2x6_pfn_dma_isr_t pfn_isr;
    void      *p_isr_arg;

    for (i = 0; i < AMHW_ZMF2X6_DMA_CHAN_CNT; i++) {
        if (amhw_zmf2x6_dma_chan_stat_check(p_hw_dma,
        AMHW_ZMF2X6_DMA_CHAN_TX_COMP_FLAG(i))) {
            if (amhw_zmf2x6_dma_chan_stat_check(p_hw_dma,
        AMHW_ZMF2X6_DMA_CHAN_TX_HALF_FLAG(i))) {

```



```

        amhw_zmf2x6_dma_chan_flag_clear(p_hw_dma,
AMHW_ZMF2X6_DMA_CHAN_TX_HALF_FLAG(i));
    }
    chan = i;
    if(amhw_zmf2x6_dma_chan_stat_check(p_hw_dma ,
AMHW_ZMF2X6_DMA_CHAN_TX_ERR_FLAG(i)) {
        flag = AM_ZMF2X6_DMA_INT_ERROR ;
        amhw_zmf2x6_dma_chan_flag_clear(p_hw_dma,
AMHW_ZMF2X6_DMA_CHAN_TX_ERR_FLAG(i));
    } else {
        flag = AM_ZMF2X6_DMA_INT_NORMAL ;
    }
    amhw_zmf2x6_dma_chan_flag_clear(p_hw_dma,
AMHW_ZMF2X6_DMA_CHAN_TX_COMP_FLAG(i));
    break;
}
}

if (0xFF != chan) {

    pfn_isr    = __dma_int_info[chan].pfn_isr;
    p_isr_arg = __dma_int_info[chan].p_arg;

    if (pfn_isr) {
        pfn_isr(p_isr_arg , flag);
    }
}
}
}

```

4. am_zmf2x6_dma_chan_start 函数

该函数主要用于开始通道传输功能，功能实现代码详见程序清单 18.14。

程序清单 18.14 am_zmf2x6_dma_chan_start 函数示例代码

```

/* 开始通道传输 */
int am_zmf2x6_dma_chan_start (int chan)
{
    __DMA_DEVINFO_DECL(p_dma_devinfo, __gp_dma_dev);

    amhw_zmf2x6_dma_t *p_hw_dma = (amhw_zmf2x6_dma_t *)p_dma_devinfo->dma_reg_base;

    amhw_zmf2x6_dma_chan_int_enable(p_hw_dma,
                                    AMHW_ZMF2X6_DMA_CHAN_INT_TX_CMP_MASK,
                                    chan);

    amhw_zmf2x6_dma_chan_enable(p_hw_dma, chan, AM_TRUE);
}

```

```

return AM_OK;
}

```

5. am_zmf2x6_dma_chan_stop 函数

该函数主要用于停止通道传输功能，功能实现代码详见程序清单 18.15。

程序清单 18.15 am_zmf2x6_dma_chan_stop 函数示例代码

```

/* 停止通道传输 */
int am_zmf2x6_dma_chan_stop (int chan)
{
    __DMA_DEVINFO_DECL(p_dma_devinfo, __gp_dma_dev);

    amhw_zmf2x6_dma_t *p_hw_dma = (amhw_zmf2x6_dma_t *)p_dma_devinfo->dma_reg_base;

    amhw_zmf2x6_dma_chan_int_disable(p_hw_dma,
                                     AMHW_ZMF2X6_DMA_CHAN_INT_TX_CMP_MASK,
                                     chan);

    amhw_zmf2x6_dma_chan_enable(p_hw_dma, chan, AM_FALSE);

    return AM_OK;
}

```

6. am_zmf2x6_dma_xfer_desc_build 函数

该函数主要用于建立传输描述符功能，功能实现代码详见程序清单 18.16。

程序清单 18.16 am_zmf2x6_dma_xfer_desc_build 函数示例代码

```

/* 建立传输描述符 */
int am_zmf2x6_dma_xfer_desc_build (amhw_zmf2x6_dma_xfer_desc_t *p_desc,
                                   uint32_t src_addr,
                                   uint32_t dst_addr,
                                   uint32_t nbytes,
                                   uint32_t flags)
{
    if (p_desc == NULL) {
        return AM_ERROR;
    }

    if (nbytes > 0xFFFF) {
        return AM_ERROR;
    }

    p_desc->xfercfg = flags;
    p_desc->src_addr = src_addr;
    p_desc->dst_addr = dst_addr;
    p_desc->nbytes = nbytes;
}

```

```

return AM_OK;
}

```

7. am_zmf2x6_dma_xfer_desc_chan_cfg 函数

该函数主要用于配置传输通道功能，功能实现代码详见程序清单 18.17。

程序清单 18.17 am_zmf2x6_dma_xfer_desc_chan_cfg 函数示例代码

```

/* 开始 DMA 传输 */
int am_zmf2x6_dma_xfer_desc_chan_cfg (amhw_zmf2x6_dma_xfer_desc_t *p_desc,
                                     amhw_zmf2x6_dma_transfer_type_t type,
                                     uint8_t chan)
{
    uint32_t per_data_size = 8;
    uint32_t mem_data_size = 8;
    uint32_t trans_data_byte = 0;

    __DMA_DEVINFO_DECL(p_dma_devinfo, __gp_dma_dev);

    amhw_zmf2x6_dma_t *p_hw_dma = (amhw_zmf2x6_dma_t *)p_dma_devinfo->dma_reg_base;

    if (p_desc == NULL) {
        return AM_ERROR;
    }

    if (chan > AMHW_ZMF2X6_DMA_CHAN_CNT) {
        return AM_ERROR;
    }

    mem_data_size = 1 << (((p_desc->xfercfg >> 8) & 0x3u) + 3);

    switch(type) {

    case AMHW_ZMF2X6_DMA_PER_TO_MER:
        p_desc->xfercfg |= (AMHW_ZMF2X6_DMA_CHAN_DIR_FROM_PER |
                           AMHW_ZMF2X6_DMA_CHAN_MEM_TO_MEM_DISABLE);

        per_data_size = 1 << (((p_desc->xfercfg >> 10) & 0x3u) + 3);
        trans_data_byte = p_desc->nbytes / (per_data_size / 8);

        amhw_zmf2x6_dma_chan_per_address_set(p_hw_dma,
                                             p_desc->src_addr,
                                             chan);

        amhw_zmf2x6_dma_chan_mem_address_set(p_hw_dma,
                                             p_desc->dst_addr,
                                             chan);
    }
}

```

```

        break;

case AMHW_ZMF2X6_DMA_MER_TO_PER:
    p_desc->xfercfg |= (AMHW_ZMF2X6_DMA_CHAN_DIR_FROM_MEM |
                       AMHW_ZMF2X6_DMA_CHAN_MEM_TO_MEM_DISABLE);

    mem_data_size    = 1 << (((p_desc->xfercfg >> 10) & 0x3u) + 3);
    trans_data_byte = p_desc->nbytes / (mem_data_size / 8);

    amhw_zmf2x6_dma_chan_per_address_set(p_hw_dma,
                                          p_desc->dst_addr,
                                          chan);

    amhw_zmf2x6_dma_chan_mem_address_set(p_hw_dma,
                                          p_desc->src_addr,
                                          chan);

    break;

case AMHW_ZMF2X6_DMA_MER_TO_MER:
    p_desc->xfercfg |= (AMHW_ZMF2X6_DMA_CHAN_DIR_FROM_MEM |
                       AMHW_ZMF2X6_DMA_CHAN_MEM_TO_MEM_ENABLE);

    mem_data_size    = 1 << (((p_desc->xfercfg >> 10) & 0x3u) + 3);
    trans_data_byte = p_desc->nbytes / (mem_data_size / 8);

    amhw_zmf2x6_dma_chan_per_address_set(p_hw_dma,
                                          p_desc->dst_addr,
                                          chan);

    amhw_zmf2x6_dma_chan_mem_address_set(p_hw_dma,
                                          p_desc->src_addr,
                                          chan);

    break;

default:
    break;
}

amhw_zmf2x6_dma_chan_tran_data_size(p_hw_dma,
                                     trans_data_byte,
                                     chan);

amhw_zmf2x6_dma_chan_config_set(p_hw_dma,
                                p_desc->xfercfg,
                                chan);

return AM_OK;

```

```
}
```

8. am_zmf2x6_dma_isr_connect 函数

该函数主要用于连接回调函数功能，功能实现代码详见程序清单 18.18。

程序清单 18.18 am_zmf2x6_dma_isr_connect 函数示例代码

```
/* 连接 DMA 中断服务函数 */
int am_zmf2x6_dma_isr_connect (int chan,
                               am_zmf2x6_pfn_dma_isr_t pfn_isr,
                               void *p_arg)
{
    if (__dma_int_map[chan] == __INT_NOT_CONNECTED) {
        __dma_int_info[chan].pfn_isr = pfn_isr;
        __dma_int_info[chan].p_arg = p_arg;
        __dma_int_map[chan] = chan;
    } else {
        return -AM_EPERM;
    }

    return AM_OK;
}
```

9. am_zmf2x6_dma_isr_disconnect 函数

该函数主要用于删除连接回调函数功能，功能实现代码详见程序清单 18.19。

程序清单 18.19 am_zmf2x6_dma_isr_disconnect 函数示例代码

```
/* 删除 DMA 中断服务函数连接 */
int am_zmf2x6_dma_isr_disconnect (int chan,
                                   am_zmf2x6_pfn_dma_isr_t pfn_isr,
                                   void *p_arg)
{
    if (__dma_int_map[chan] == chan) {
        __dma_int_info[chan].pfn_isr = NULL;
        __dma_int_map[chan] = __INT_NOT_CONNECTED;
    } else {
        return -AM_EPERM;
    }

    return AM_OK;
}
```

10. am_zmf2x6_dma_tran_data_get 函数

该函数主要用于获取某通道当前传输剩余的字节数功能，功能实现代码详见程序清单 18.20。

程序清单 18.20 am_zmf2x6_dma_tran_data_get 函数示例代码

```

/**
 * \brief 获取某通道当前传输剩余的字节数
 */
uint16_t am_zmf2x6_dma_tran_data_get (int chan)
{
    __DMA_DEVINFO_DECL(p_dma_devinfo, __gp_dma_dev);
    amhw_zmf2x6_dma_t *p_hw_dma = (amhw_zmf2x6_dma_t *)p_dma_devinfo->dma_reg_base;

    return amhw_zmf2x6_dma_chan_tran_data_get(p_hw_dma, chan);
}

```

11. am_zmf2x6_dma_tran_data_size 函数

该函数主要用于设置某通道当前传输剩余的字节数功能，功能实现代码详见程序清单 18.21。

程序清单 18.21 am_zmf2x6_dma_tran_data_size 函数示例代码

```

/**
 * \brief 设置传输的字节数
 */
void am_zmf2x6_dma_tran_data_size (int chan, uint32_t trans_data_byte)
{
    __DMA_DEVINFO_DECL(p_dma_devinfo, __gp_dma_dev);
    amhw_zmf2x6_dma_t *p_hw_dma = (amhw_zmf2x6_dma_t *)p_dma_devinfo->dma_reg_base;

    amhw_zmf2x6_dma_chan_tran_data_size(p_hw_dma,
                                         trans_data_byte,
                                         chan);
}

```

18.4 用户配置文件编写

18.4.1 定义设备常量于设备信息常量

创建配置文件后，我们需要在里面加入设备变量，设备变量只需要定义，不用初始化。设备信息常量定义时初始化，默认填充的信息是一种常用的配置方式，DMA 设备信息配置如程序清单 18.22 所示。

程序清单 18.22 DMA 设备信息配置文件

```

/** \brief DMA 设备信息 */
static const am_zmf2x6_dma_devinfo_t __g_dma_devinfo = {
    ZMF2X6_DMA_BASE,           /**< \brief 指向 DMA 寄存器块的指针 */
    INUM_DMA1_1,               /**< \brief DMA 中断向量号开始 */
    INUM_DMA1_4_5,            /**< \brief DMA 中断向量号结束 */
    __zmf2x6_plfm_dma_init,    /**< \brief DMA 平台初始化 */
    __zmf2x6_plfm_dma_deinit /**< \brief DMA 平台解初始化 */
};

```

```
/** \brief DMA 设备实例 */
static am_zmf2x6_dma_dev_t __g_dma_dev;
```

18.4.2 添加平台初始化与平台解初始化函数

通常某个外设正常运行，还需要 GPIO 管脚、时钟等平台相关的初始化支持，因此还需要为设备提供平台初始化函数和解初始化函数。

由于 DMA 无相关引脚需要配置，只需使能外设时钟，因此，平台初始化函数和解初始化函数定义如程序清单 18.23 所示。

程序清单 18.23 平台初始化函数和解初始化函数定义

```
/**
 * \brief DMA 平台初始化。
 */
void __zmf2x6_plfm_dma_init (void)
{
    /* 时能 DMA 相关模块时钟或电源 */
    am_clk_enable(CLK_DMA);
}

/**
 * \brief EDMA 平台去初始化。
 */
void __zmf2x6_plfm_dma_deinit (void)
{
    /* 时能 DMA 相关模块时钟或电源 */
    am_clk_disable(CLK_DMA);
}
```

18.4.3 在头文件添加变量和函数声明

定义好了变量和函数后，还需要在标准层实例公共头文件 am_zmf2x6_inst_init.h 中加入它们的声明，如所示程序清单 18.24。

程序清单 18.24 初始化解初始化声明

```
/**
 * \brief DMA 实例初始化
 *
 * \param 无
 *
 * \return 返回 AW_OK 为初始化成功，其它为初始化失败
 */
int am_zmf2x6_dma_inst_init (void);

/**
```

```

* \brief DMA 实例解初始化
*
* \param 无
*
* \return 无
*/
void am_zmf2x6_dma_inst_deinit(void);

```

用户使用时，直接调用实例初始化函数 `am_zmf2x6_dma_inst_init` 即可完成 DMA 初始化。

至此，用户配置文件开发完毕。

18.5 例程文件开发

例程文件开发为 HW 例程。

调试例程时，应该在 `template` 工程下进行，不需要新建其它工程，只需要将新建的文件加入到工程结构的 `user_code` 下。例程文件会与原先的 `main.c` 文件冲突（因为都存在 `main` 函数），这时，只需要调试时将原先的 `main.c` 不要包含在工程编译中，例程文件开发完毕后，再恢复将 `main.c` 包含在工程中，同时将例程文件移除工程即可。

18.5.1 例程内容开发

1. HW 层实现示例

这里使用 HW 层来实现 ADC 采集 DMA 传输方式，相关代码如程序清单 18.25 所示。

程序清单 18.25 ADC 采集 DMA 传输方式

```

/** [src_zmf2x6_hw_adc_dma] */
#include "ametal.h"
#include "am_vdebug.h"
#include "am_zmf2x6.h"
#include "am_gpio.h"
#include "am_clk.h"
#include "demo_zmf2x6_entries.h"
#include "demo_zmf2x6_core_entries.h"
#include "am_zmf2x6_dma.h"
#include "hw/amhw_zmf2x6_dma.h"
#include "hw/amhw_zmf2x6_adc.h"

#define BUF_SIZE 10

static uint32_t __g_buf_dst[BUF_SIZE];    /**< \brief 目标端数据缓冲区 */

static uint16_t __g_adc_dat[BUF_SIZE];    /**< \brief ADC 数据缓冲区 */

static am_bool_t g_trans_done;            /**< \brief 传输完成标志 */

static amhw_zmf2x6_dma_xfer_desc_t g_desc; /**< \brief DMA 描述符 */

```



```

/**
 * \brief DMA 中断服务程序
 */
static void dma_isr (void *p_arg , uint32_t flag)
{
    int i;
    int cur_chan;

    if (flag == AM_ZMF2X6_DMA_INT_NORMAL) {

        for (i = 0; i < BUF_SIZE; i++) {

            /* 获取当前请求 DMA 的 ADC 通道 */
            cur_chan = (__g_buf_dst[i] >> 16) & 0x0f;

            /* 低 12 位为 ADC 采集数据 */
            __g_adc_dat[cur_chan] = __g_buf_dst[i] & 0xfff;
        }

        g_trans_done = AM_TRUE;
    }
}

static void __zmf2x6_adc_init (amhw_zmf2x6_adc_t *p_hw_adc,
                               int                *p_adc_chan,
                               int                adc_chan_num)
{
    int i;

    /* ADC 禁能*/
    amhw_zmf2x6_adc_cgf_reg_clr(p_hw_adc, AMHW_ZMF2X6_ADC_MODULE_EN_MASK);

    /* 设置 ADC 工作频率，72 MHz 的 8 分频 (ADC 输入时钟不超过 15 MHz) */
    amhw_zmf2x6_adc_prescaler_val_set(p_hw_adc, 8);

    /* 单周期扫描模式 */
    amhw_zmf2x6_adc_data_scan_set(p_hw_adc, AMHW_ZMF2X6_ADC_SINGLE_PERIOD_SCAN);

    /* 对齐方式 -- 右对齐 */
    amhw_zmf2x6_adc_align_way_sel(p_hw_adc, AMHW_ZMF2X6_ADC_DATA_ALIGN_RIGHT);

    /* 分辨率设置 -- 12 位 */
    amhw_zmf2x6_adc_data_resolution(p_hw_adc, AMHW_ZMF2X6_ADC_DATA_VALID_12BIT);

```

```

/* 设置 ADC 规则通道长度 */
amhw_zmf2x6_adc_regular_channel_length_set(
    p_hw_adc,
    (amhw_zmf2x6_adc_regular_channel_length_t)(adc_chan_num - 1));

for(i = 0; i < adc_chan_num; i++) {

    /* 通道使能 */
    amhw_zmf2x6_adc_input_chan_enable(p_hw_adc,
        (amhw_zmf2x6_adc_channel_t)p_adc_chan[i]);

    /* 链接规则通道序列和 ADC 采样通道 */
    amhw_zmf2x6_adc_regular_channel_order_set(
        p_hw_adc,
        (amhw_zmf2x6_adc_regular_channel_order_t)(AMHW_ZMF2X6_ADC_REGULAR_CHAN_ORDER_1st + i),
        (amhw_zmf2x6_adc_channel_t)p_adc_chan[i];
    }

    /* 设置采样周期 */
    amhw_zmf2x6_adc_chan_sample_time(
        p_hw_adc,
        AMHW_ZMF2X6_ADC_CHAN_ST55_5);

    /* 任意通道配置模式使能 */
    amhw_zmf2x6_adc_chany_mode_enable(p_hw_adc);

    /* 使能 ADC DMA 请求 */
    amhw_zmf2x6_adc_ctrl_reg_set(p_hw_adc,
        AMHW_ZMF2X6_ADC_DMA_EN_MASK);

    /* ADC 使能 */
    amhw_zmf2x6_adc_cgf_reg_set(p_hw_adc, AMHW_ZMF2X6_ADC_MODULE_EN_MASK);
}

/**
 * \brief 例程入口
 */
void demo_zmf2x6_hw_adc_dma_entry (amhw_zmf2x6_adc_t    *p_hw_adc,
                                   int                    *p_adc_chan,
                                   int                    adc_chan_num,
                                   uint32_t               vref_mv,
                                   int                    dma_chan)

```

```

{
    uint32_t adc_mv = 0;    /* 采样电压 */
    uint32_t flags;
    int      i;

    am_kprintf("The ADC HW dma Demo\r\n");

    __zmf2x6_adc_init(p_hw_adc, p_adc_chan, adc_chan_num);

    /* 连接 DMA 中断服务函数 */
    am_zmf2x6_dma_isr_connect(dma_chan, dma_isr, (void *)p_hw_adc);

    /* DMA 传输配置 */
    flags = AMHW_ZMF2X6_DMA_CHAN_PRIORITY_HIGH          | /* 中断优先级 高 */
            AMHW_ZMF2X6_DMA_CHAN_MEM_SIZE_32BIT         | /* 内存数据宽度 4 字节 */
            AMHW_ZMF2X6_DMA_CHAN_PER_SIZE_32BIT        | /* 外设数据宽度 4 字节 */
            AMHW_ZMF2X6_DMA_CHAN_MEM_ADD_INC_ENABLE    | /* 内存地址自增 */
            AMHW_ZMF2X6_DMA_CHAN_PER_ADD_INC_DISABLE   | /* 外设地址不自增 */
            AMHW_ZMF2X6_DMA_CHAN_CIRCULAR_MODE_ENABLE  | /* 打开循环模式 */
            AMHW_ZMF2X6_DMA_CHAN_INT_TX_CMP_ENABLE;     /* 使能 DMA 传输完成中
断 */

    /* 建立通道描述符 */
    am_zmf2x6_dma_xfer_desc_build(&g_desc,                /* 通道描述符 */
                                   (uint32_t)(&p_hw_adc->addata), /* 源端数据缓冲区 */
                                   (uint32_t)(__g_buf_dst),        /* 目标端数据缓冲区 */
                                   sizeof(__g_buf_dst),            /* 传输字节数 */
                                   flags);                        /* 传输配置 */

    am_zmf2x6_dma_xfer_desc_chan_cfg(&g_desc,
                                      AMHW_ZMF2X6_DMA_PER_TO_MER, /* 外设到内存 */
                                      dma_chan);

    /* 启动 DMA 传输，马上开始传输 */
    am_zmf2x6_dma_chan_start(dma_chan);

    while(1) {

        /* 转换开始 */
        amhw_zmf2x6_adc_data_transfer_set(p_hw_adc,
                                            AMHW_ZMF2X6_ADC_DATA_TRANSFER_START);

        /* 等待传输完成 */
        while(g_trans_done == AM_FALSE);
    }
}

```

```

    for (i = 0; i < adc_chan_num; i++) {

        /* 转换为电压值对应的整数值 */
        adc_mv = __g_adc_dat[p_adc_chan[i]] * vref_mv / ((1UL << 12) - 1);

        am_kprintf("chan: %d, Sample : %d, Vol: %d mv\r\n", p_adc_chan[i],
                                                            __g_adc_dat[p_adc_chan[i]],
                                                            adc_mv);

    }

    am_kprintf("\r\n");
    am_mdelay(500);

    g_trans_done = AM_FALSE;
}

/**
 * \brief 例程入口
 */
void demo_zmf2x6_core_hw_adc_dma_entry (void)
{
    /* 10 个顺序通道 ADC 采集 */
    int adc_chan[] = {AMHW_ZMF2X6_ADC_CHAN_0,
                      AMHW_ZMF2X6_ADC_CHAN_1,
                      AMHW_ZMF2X6_ADC_CHAN_2,
                      AMHW_ZMF2X6_ADC_CHAN_3,
                      AMHW_ZMF2X6_ADC_CHAN_4,
                      AMHW_ZMF2X6_ADC_CHAN_5,
                      AMHW_ZMF2X6_ADC_CHAN_6,
                      AMHW_ZMF2X6_ADC_CHAN_7,
                      AMHW_ZMF2X6_ADC_CHAN_8,
                      AMHW_ZMF2X6_ADC_CHAN_9};

    AM_DBG_INFO("demo zmf2x6_core hw adc dma!\r\n");

    /* 配置引脚 */
    am_gpio_pin_cfg(PIOA_0, PIOA_0_GPIO | PIOA_0_AIN);
    am_gpio_pin_cfg(PIOA_1, PIOA_1_GPIO | PIOA_1_AIN);
    am_gpio_pin_cfg(PIOA_2, PIOA_2_GPIO | PIOA_2_AIN);
    am_gpio_pin_cfg(PIOA_3, PIOA_3_GPIO | PIOA_3_AIN);
    am_gpio_pin_cfg(PIOA_4, PIOA_4_GPIO | PIOA_4_AIN);
    am_gpio_pin_cfg(PIOA_5, PIOA_5_GPIO | PIOA_5_AIN);

```

```
am_gpio_pin_cfg(PIOA_6, PIOA_6_GPIO | PIOA_6_AIN);
am_gpio_pin_cfg(PIOA_7, PIOA_7_GPIO | PIOA_7_AIN);
am_gpio_pin_cfg(PIOB_0, PIOB_0_GPIO | PIOB_0_AIN);
am_gpio_pin_cfg(PIOB_1, PIOB_1_GPIO | PIOB_1_AIN);

am_clk_enable(CLK_ADC1);

demo_zmf2x6_hw_adc_dma_entry(ZMF2X6_ADC,
                               adc_chan,
                               sizeof(adc_chan)/sizeof(adc_chan[0]),
                               3300,
                               DMA_CHAN_ADC);
}
```

所有例程开发完毕后，整个外设的开发工作也就圆满结束了。

可以提交相关文件，以完成整个开发任务。提交文件时，不用提交工程文件，只需要提交开发的源文件和头文件即可。

第19章 GIT 开源发布

本章展示如何安装 GIT 插件，如何获取源码，如何创建账号，如何将源码导入自己的仓库，如何克隆到本地进行使用，如何进行代码同步，如何将修改的代码上传到仓库，以及如何贡献自己的代码到原仓库中等。

19.1 GitHub 准备工作

在进行源码获取之前，我们需要安装 git for Windows 软件和创建一个自己的 GitHub 账号。

19.1.1 安装 git for Windows

1. 下载 git for Windows

访问 <https://gitforwindows.org> 该网址，下载 git for Windows 工具，如图 2 所示。

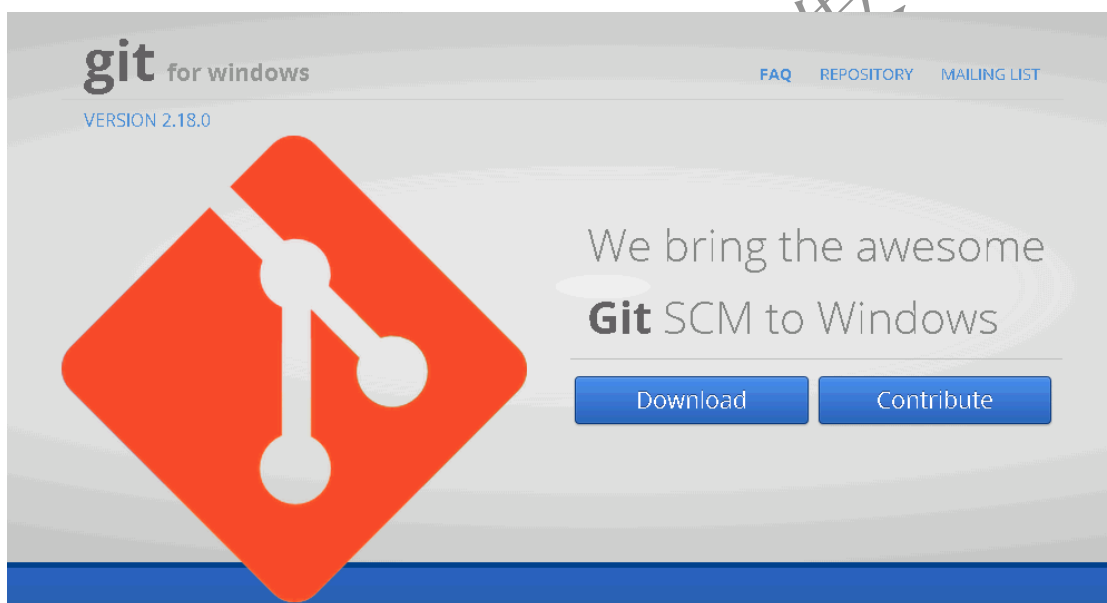


图 2 下载 git for Windows

2. 安装 git for Windows

下载得到的是 exe 文件，如图 3 所示，双击启动安装（全部默认设置即可）。

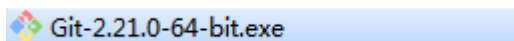


图 3 安装 git for Windows

3. 安装完成

安装完成后，点击鼠标右键可以看到 Git bash here 和 Git GUI here 选项，点击 Git bash here，可进入 Git bash 命令窗口，如图 4 所示。

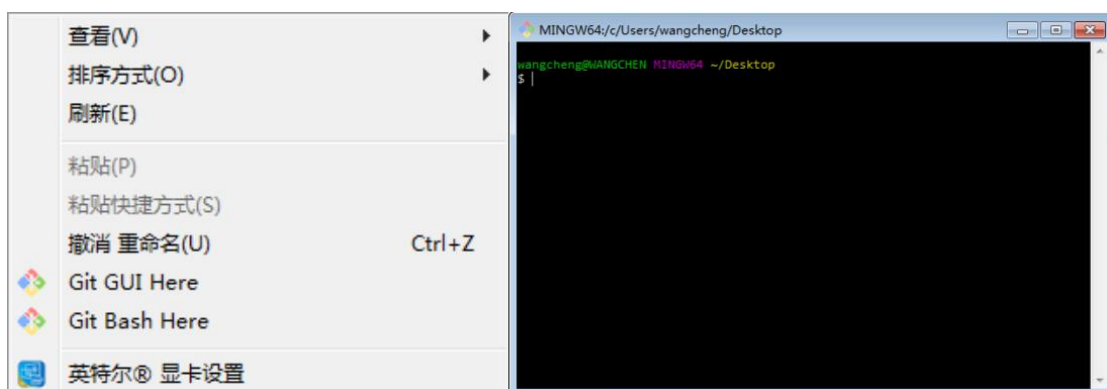


图 4 安装完成

19.1.2 创建 GitHub 账号

1. 访问 GitHub 官网，登录 <https://github.com>，注册一个自己的账号。填入自己的信息后，点击注册。如图 5 所示。

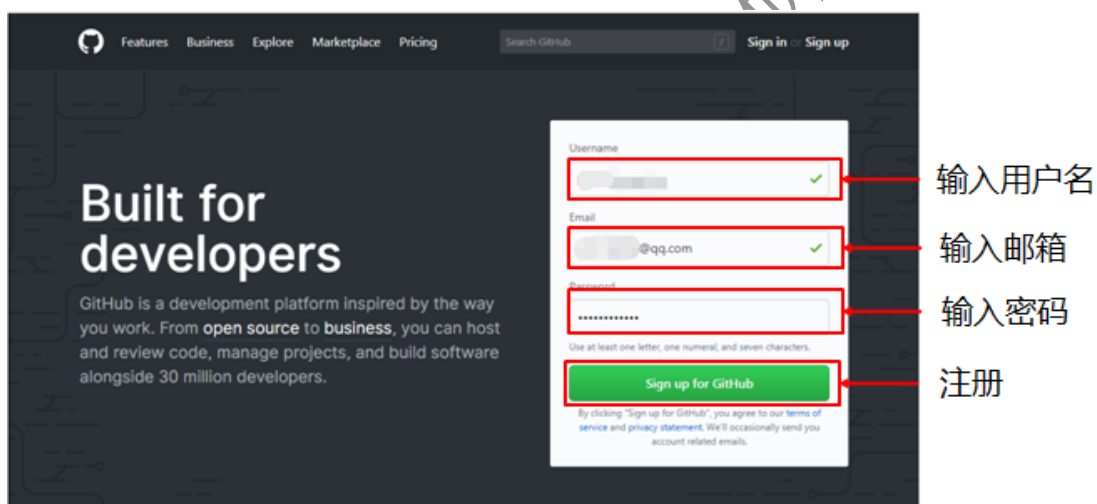


图 5 创建 GitHub 账号

2. 如图 6 所示，点击 continue（免费使用，所建仓库为公共仓库，所有人均可见）

Welcome to GitHub

You've taken your first step into a larger world, @wangcheng-zlg.

✓ Completed
Set up a personal account

🔧 Step 2:
Choose your plan

⚙️ Step 3:
Tailor your experience

Choose your personal plan

☒ Unlimited public repositories for free.

☐ Unlimited private repositories for \$7/month. [\(view in CNY\)](#)

Don't worry, you can cancel or upgrade at any time.

☐ Help me set up an organization next
Organizations are separate from personal accounts and are best suited for businesses who need to manage permissions for many employees. [Learn more about organizations](#)

☐ Be the first to discover new resources from GitHub
Unsubscribe anytime in your email preferences. [Learn more](#)

Continue

Both plans include:

✓ Collaborative code review

✓ Issue tracking

✓ Open source community

✓ Unlimited public repositories

✓ Join any organization

图 6 点击 continue

3. 选择计划用 GitHub 做什么，信息收集表（直接跳过即可），如图 7 所示。

Welcome to GitHub

You'll find endless opportunities to learn, code, and create, @wangcheng-zlg.

✓ Completed
Set up a personal account

🔧 Step 2:
Choose your plan

⚙️ Step 3:
Tailor your experience

How would you describe your level of programming experience?

☐ Totally new to programming

☐ Somewhat experienced

☐ Very experienced

What do you plan to use GitHub for? (check all that apply)

☐ School projects

☐ Research

☐ Project Management

☐ Design

☐ Development

☐ Other (please specify)

Which is closest to how you would describe yourself?

☐ I'm a hobbyist

☐ I'm a student

☐ I'm a professional

☐ Other (please specify)

What are you interested in?

e.g. tutorials, android, ruby, web-development, machine-learning, open-source

Submit

[skip this step](#)

图 7 点击跳过

580

4. 激活邮箱，注册邮箱会收到一封邮件，点击其中的激活链接即可，如图 8 所示。到此准备工作完成。

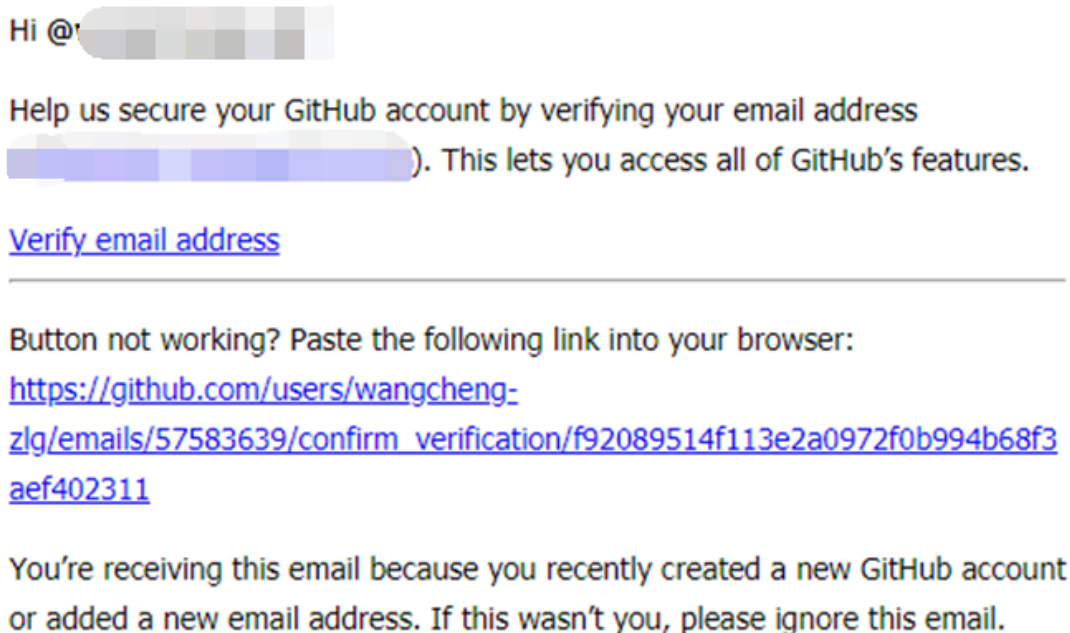


图 8 激活邮箱

19.2 协同开发

19.2.1 获取源码

1. 账号注册完成之后，使用自己的账号登录到 GitHub 官网，搜索 AMetal 仓库，如图 9 所示。

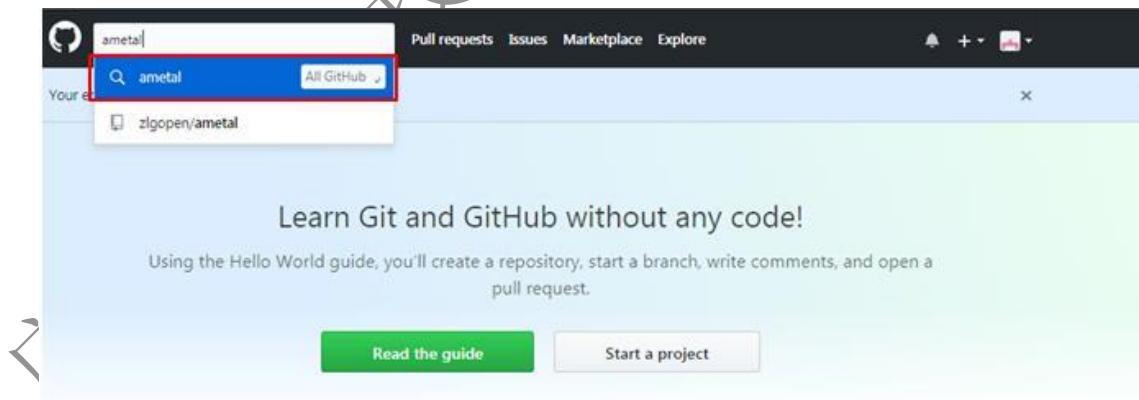


图 9 搜索 AMetal 仓库

2. 在搜索结果中选择 zlgopen/ametal，一般为搜索结果第一个既是，如图 10，也可在浏览器地址栏中直接输入 AMetal 开源地址：<https://github.com/zlgopen/ametal>。

4 repository results



图 10 搜索 AMetal

3. 将 AMetal 仓库 fork 到自己的仓库，如图 11 所示，fork 前，可以点击 watch 和 star。watch 后，如果仓库有更新会邮件通知。

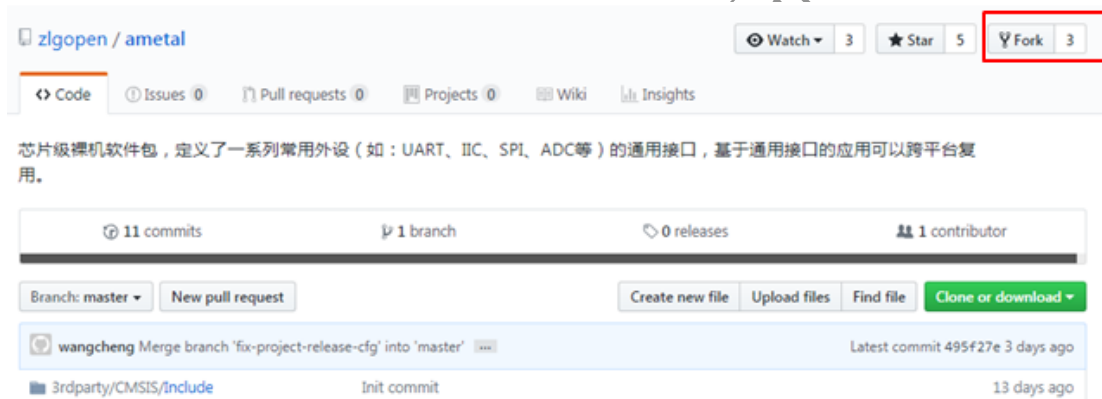


图 11 fork 操作

4. fork 完成后，自己账号下也有一个 ametal 仓库了，如图 12 所示，自己账号下的仓库即有权限修改，这个仓库的地址为：<https://github.com/yanjianxiong/ametal>。

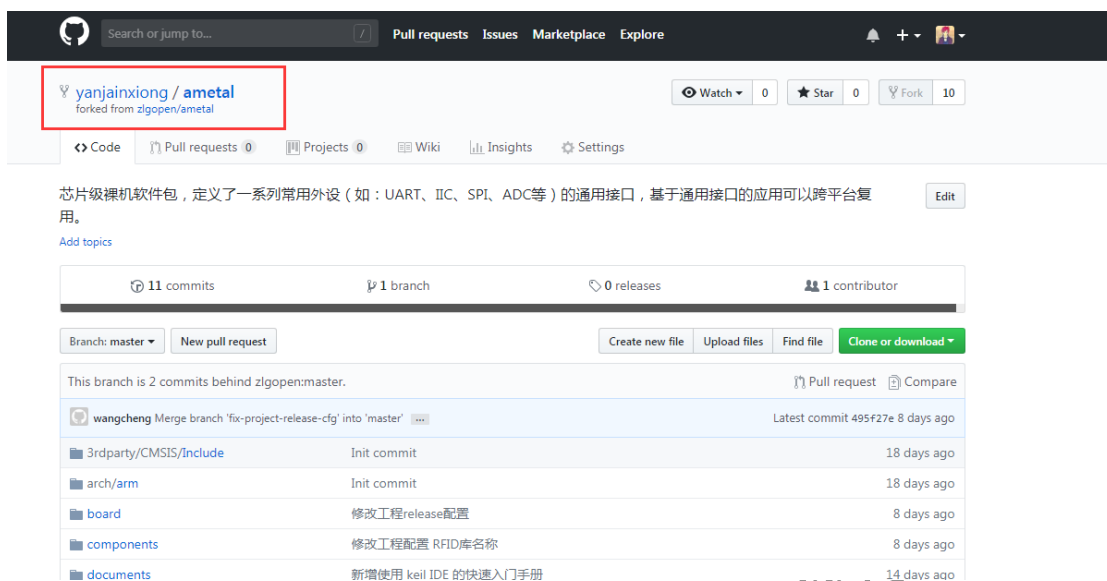


图 12 fork 完成

5. 将仓库克隆到本地以便使用，查看 clone 地址，如图13所示。

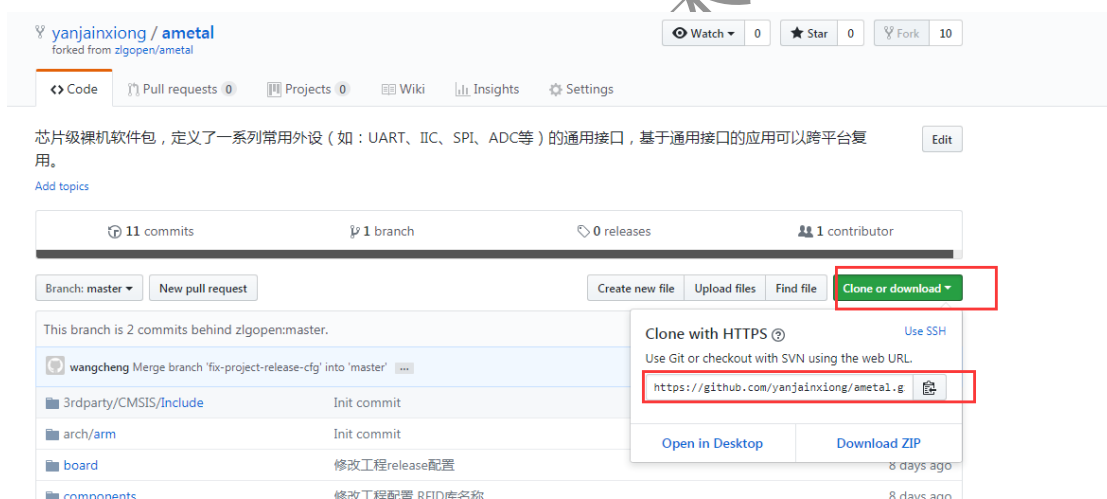


图 13 查看 clone 地址

6. 将仓库克隆到本地以便使用，在本地建立一个目录，将仓库克隆到该位置，如图14所示。



图 14 建立克隆目录

7. 在该目录下，右键，然后点击 Git bash here，运行 Git clone 命令，如图 15 所示。

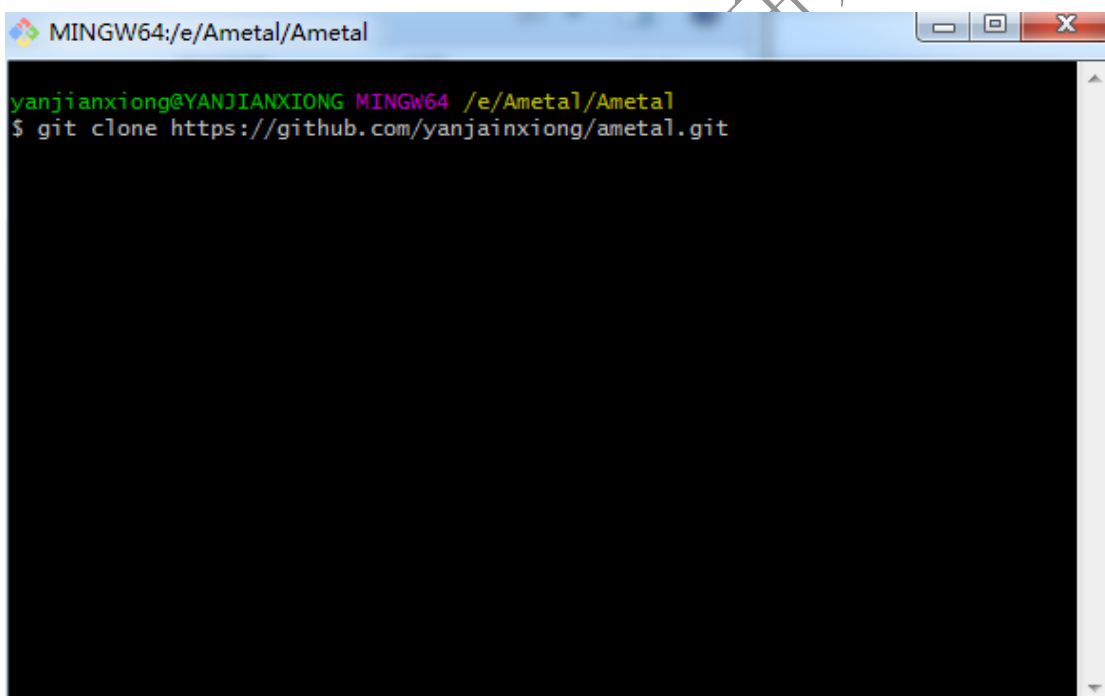


图 15 克隆源码到本地

8. clone 完成，在本地目录下生成了一个 ametal 的目录，即克隆的源码，如图 16 所示。

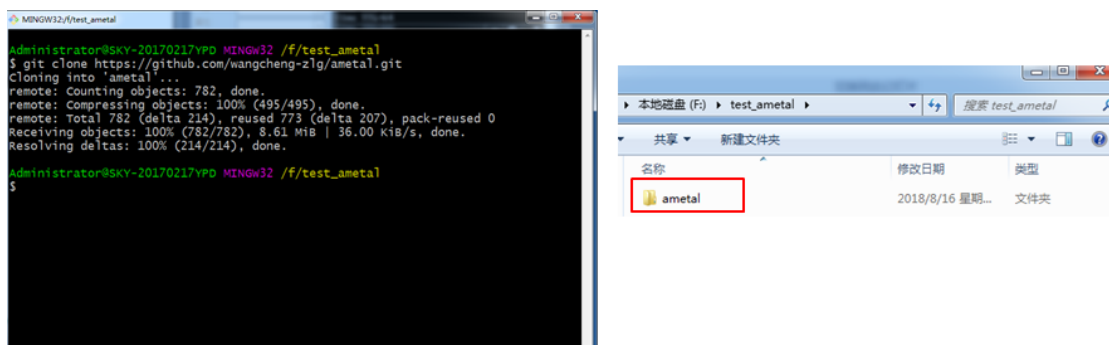


图 16 克隆完成

19.2.2 源码同步

zlgopen/ametal 项目在不断维护,可能会上传新的代码,或者修复已有代码的一些问题。本地使用的代码应该与 zlgopen/ametal 上的代码经常同步,保持一致。

1. 添加 zlgopen/ametal 的克隆地址为本地仓库的远端仓库,如图 17 所示,右键点击进入 Git bash here,输入 `git remote add zlgopen`,然后拷贝自己的仓库地址,回车,如图所示。(这里 zlgopen 仅仅是后面链接的代号,也可以是其他,如 zlgopencode 等,方便记忆即可)

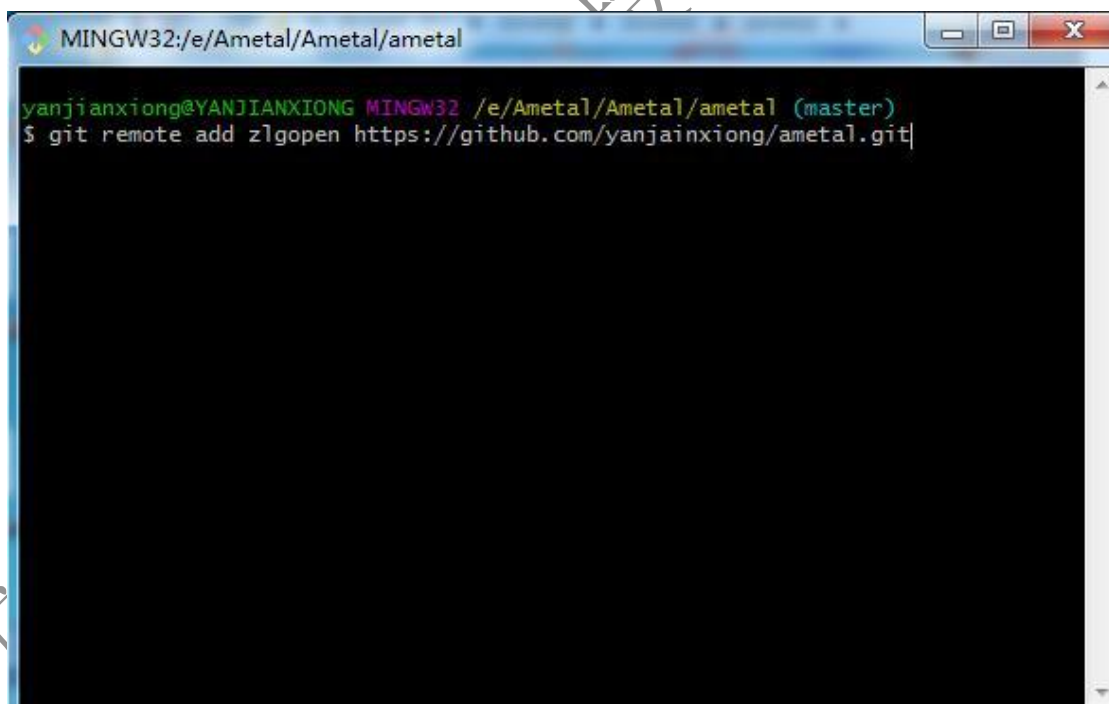
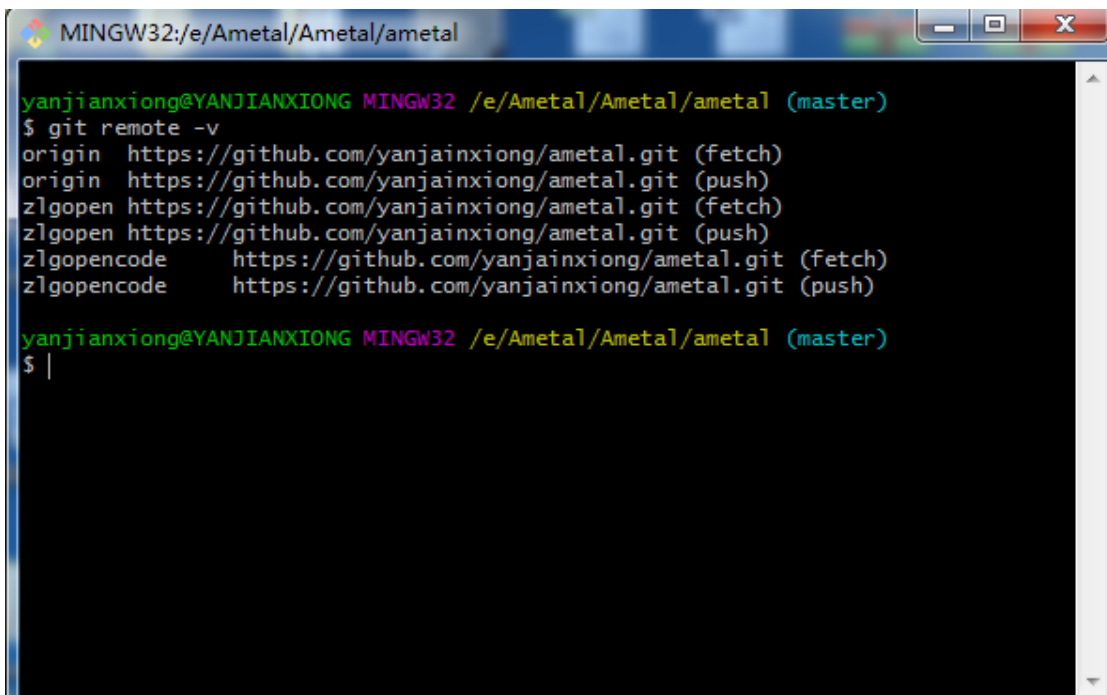


图 17 添加克隆地址为本地仓库的远程仓库

2. 查看当前的远程仓库,前面已添加克隆地址作为本地仓库的远端仓库,这里我们需要查看一下是否设置成功,接着输入命令行 `git remote -v`,回车即可查看,origin 为 clone 时的链接,后面的 zlgopen 和 zlgopencode 为前面添加的远端仓库地址,如图 18 所示。



```

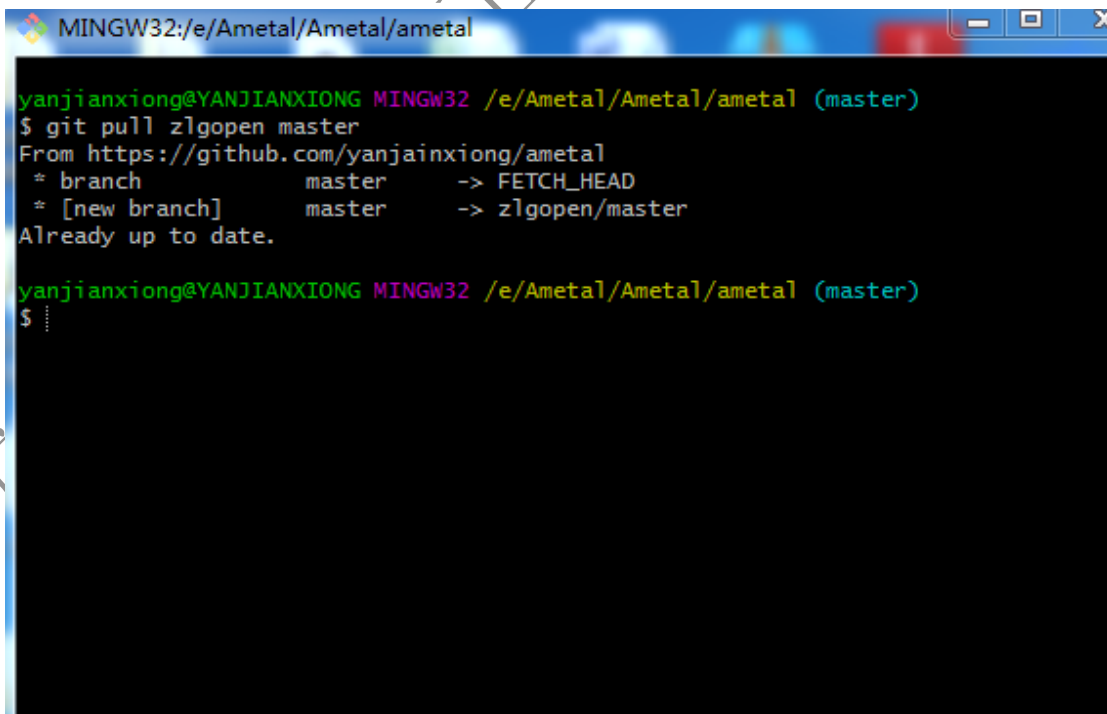
MINGW32:/e/Ametal/Ametal/ametal
yanjianxiong@YANJIANXIONG MINGW32 /e/Ametal/Ametal/ametal (master)
$ git remote -v
origin https://github.com/yanjainxiong/ametal.git (fetch)
origin https://github.com/yanjainxiong/ametal.git (push)
zlgopen https://github.com/yanjainxiong/ametal.git (fetch)
zlgopen https://github.com/yanjainxiong/ametal.git (push)
zlgopencode https://github.com/yanjainxiong/ametal.git (fetch)
zlgopencode https://github.com/yanjainxiong/ametal.git (push)

yanjianxiong@YANJIANXIONG MINGW32 /e/Ametal/Ametal/ametal (master)
$ |

```

图 18 查看添加的地址

3. 将 zlgopen/ametal 上的更新拉取到本地，输入命令 `git pull zlgopen master`，回车，之后会更新 zlgopen/ametal 到本地，建议在每次开发之前，可以执行一次该操作，确保得到最新的代码，如图 19 所示。



```

MINGW32:/e/Ametal/Ametal/ametal
yanjianxiong@YANJIANXIONG MINGW32 /e/Ametal/Ametal/ametal (master)
$ git pull zlgopen master
From https://github.com/yanjainxiong/ametal
 * branch          master      -> FETCH_HEAD
 * [new branch]     master      -> zlgopen/master
Already up to date.

yanjianxiong@YANJIANXIONG MINGW32 /e/Ametal/Ametal/ametal (master)
$ .....

```

图 19 更新代码到本地

19.2.3 向上同步代码

1. 在本地修改自己的代码后，自己的远程仓库（yanjianxiong/ametal）还未更新，需

要上传更新自己的代码到自己的远程仓库，操作如图 20 所示。输入指令 `git push origin master`，回车，会弹出 GitHub 的登录窗口。

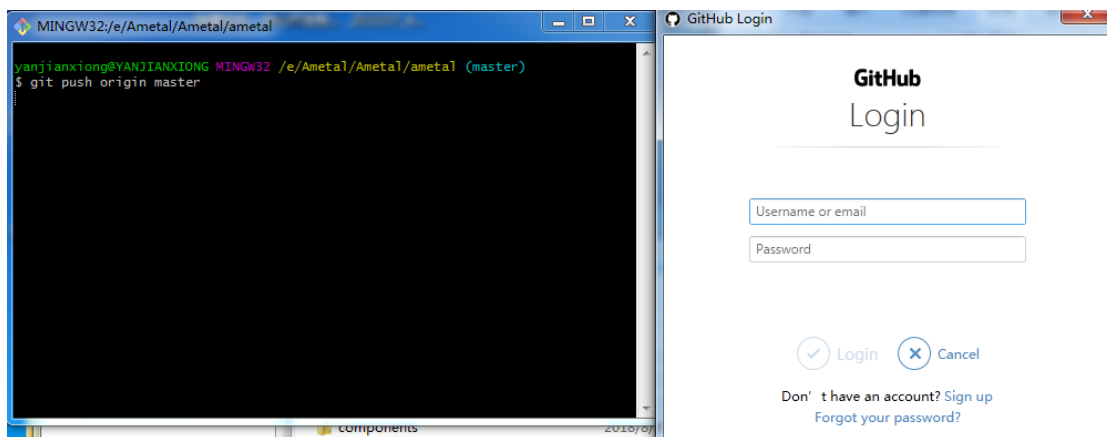


图 20 输入上传指令

2. 首次上传，需配置邮箱和用户名，操作方法如图 21 所示。

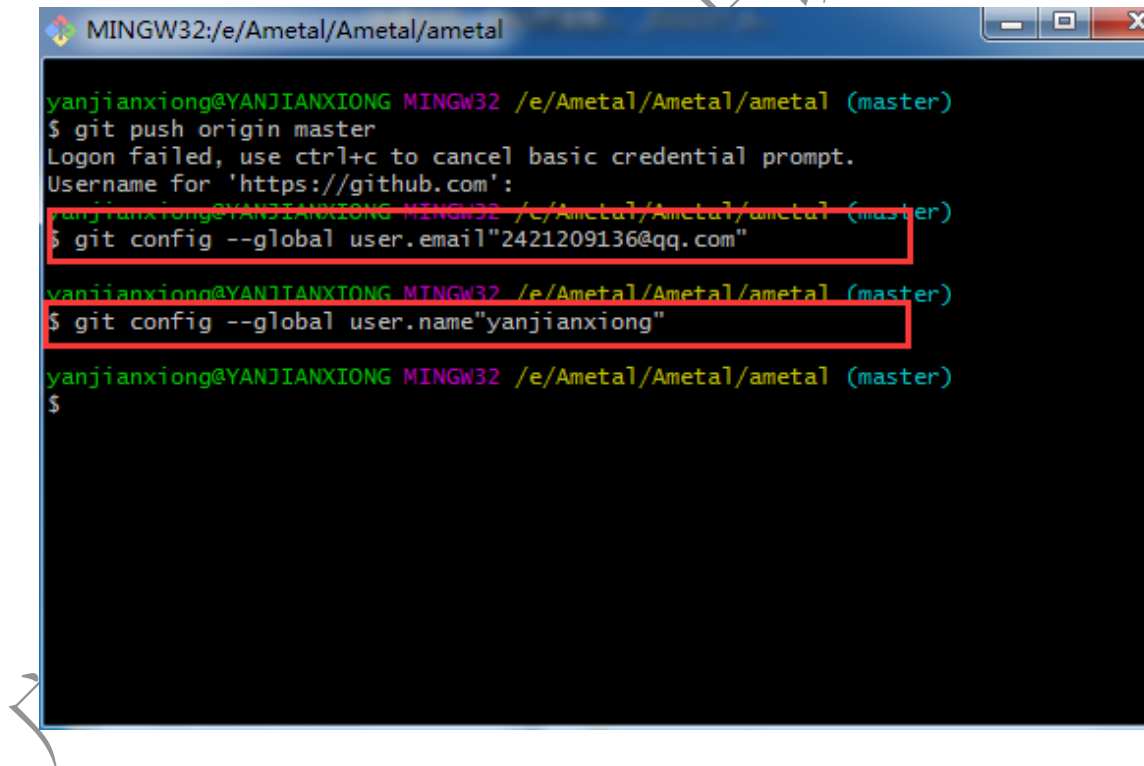


图 21 配置邮箱和用户名

3. 例如添加一个 `readme` 文本文件，`readme` 文本的内容不得为空，可向里面加一行代码，如加一行 `hello` 到文本中，如图 22 所示。

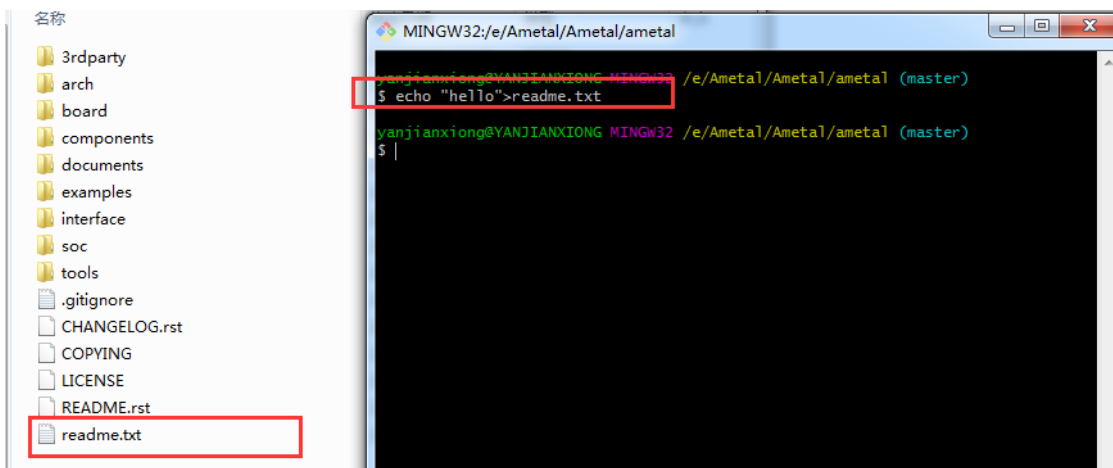


图 22 添加文本测试

4. 使用 `git add` 命令将其添加到缓存中，准备提交，如图 23 所示。

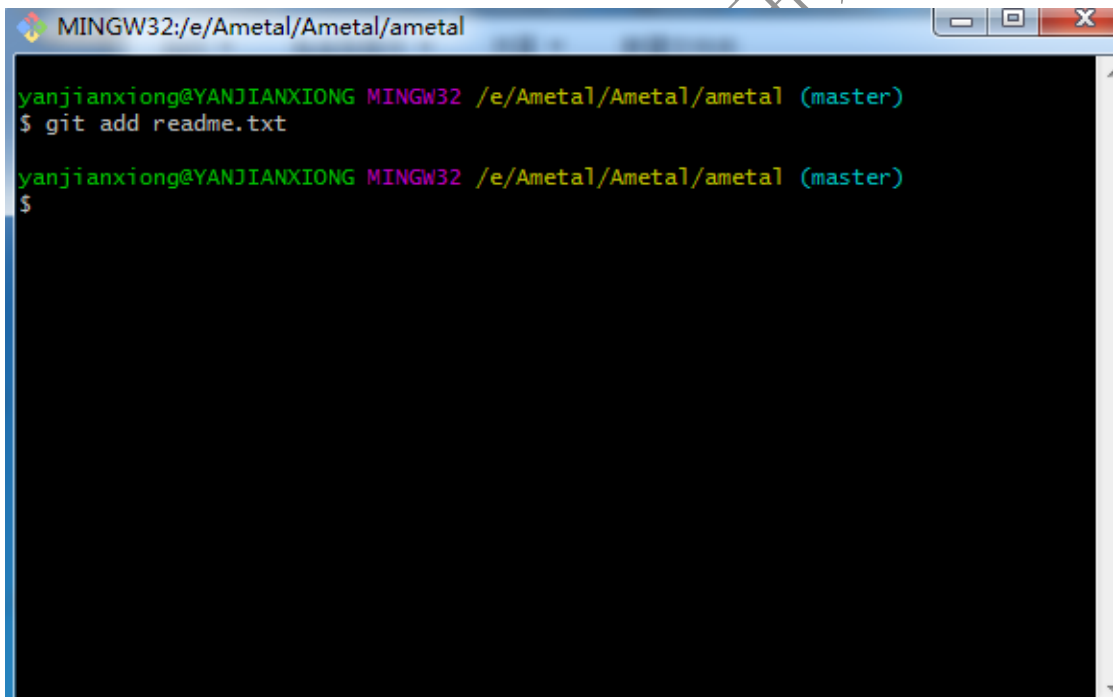
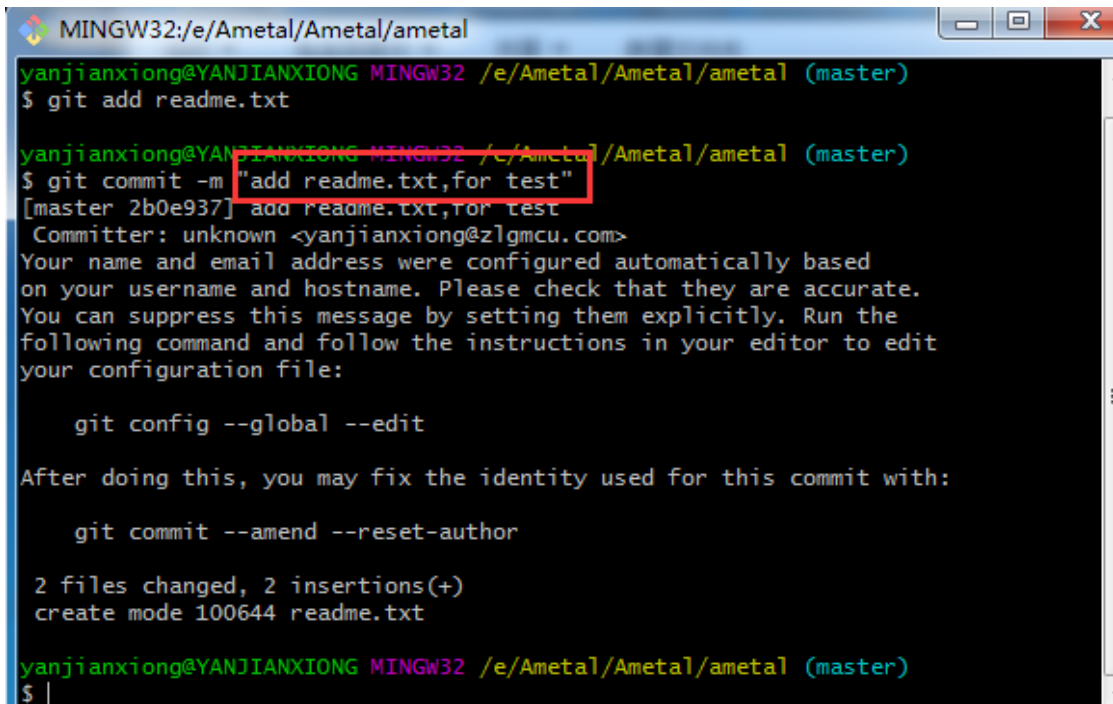


图 23 添加内容到缓存

5. 使用 `git commit` 命令提交到本地仓库，如图 24 所示，建议每次提交时都添加日志记录，明确修改了什么内容，成为一种习惯。建议设置登录的邮箱为自己电脑的本机邮箱，因公司电脑无法登录外网，绑定的邮箱也非公司邮箱，所以出现下面的验证提示，可忽略该提示；如果使用电脑上的验证邮箱与 GitHub 邮箱一致，则不会出现后面的提示。



```

MINGW32:/e/Ametal/Ametal/ametal
yanjianxiong@YANJIANXIONG MINGW32 /e/Ametal/Ametal/ametal (master)
$ git add readme.txt

yanjianxiong@YANJIANXIONG MINGW32 /e/Ametal/Ametal/ametal (master)
$ git commit -m "add readme.txt, for test"
[master 2b0e937] add readme.txt, for test
Committer: unknown <yanjianxiong@zlgmcu.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

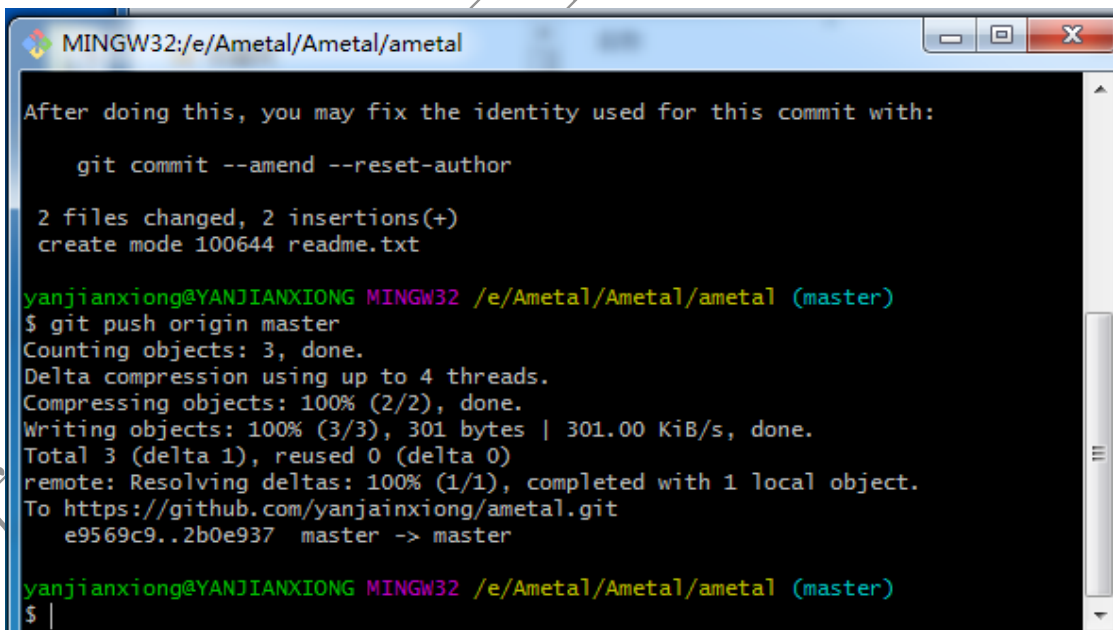
2 files changed, 2 insertions(+)
create mode 100644 readme.txt

yanjianxiong@YANJIANXIONG MINGW32 /e/Ametal/Ametal/ametal (master)
$

```

图 24 添加日志

6. 将修改上传到自己的 github 上的仓库中, 操作如图 25 所示。输入命令 `git push origin master`, 回车即可。



```

MINGW32:/e/Ametal/Ametal/ametal

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

2 files changed, 2 insertions(+)
create mode 100644 readme.txt

yanjianxiong@YANJIANXIONG MINGW32 /e/Ametal/Ametal/ametal (master)
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 301 bytes | 301.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/yanjainxiong/ametal.git
   e9569c9..2b0e937  master -> master

yanjianxiong@YANJIANXIONG MINGW32 /e/Ametal/Ametal/ametal (master)
$

```

图 25 上传到自己的 GitHub 库

7. 登录到仓库查看, 可看到已上传成功, 如图 26 所示。

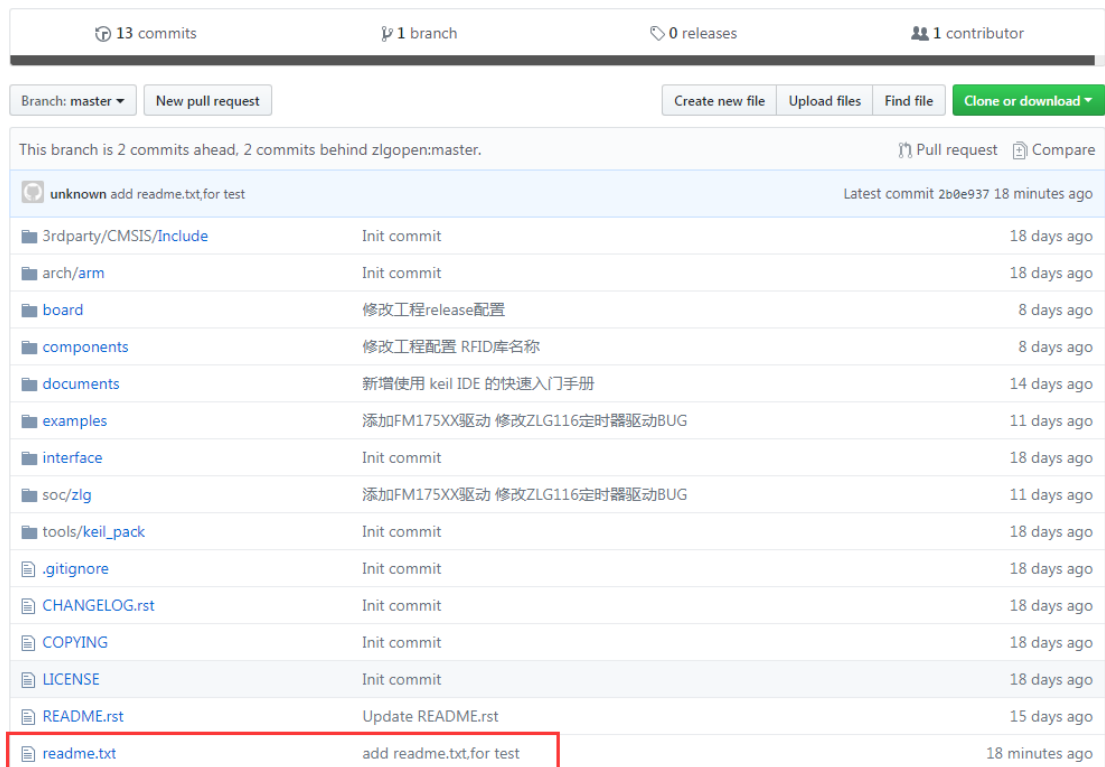


图 26 登录查看上传情况

19.2.4 请求合并到源码库

开源项目永远期待更多的人加入，为开源贡献自己的力量。若修改没有问题，可以请求将修改合并到 zlgopen/ameal 开源项目中。

1. 登录到 github，查看自己的修改内容，如图 27 所示。（这里由于修改时公司电脑绑定邮箱无法收到外部邮件，故这里提示修改人为 unknown，如用自己电脑，可以看到修改人为自己的用户名）

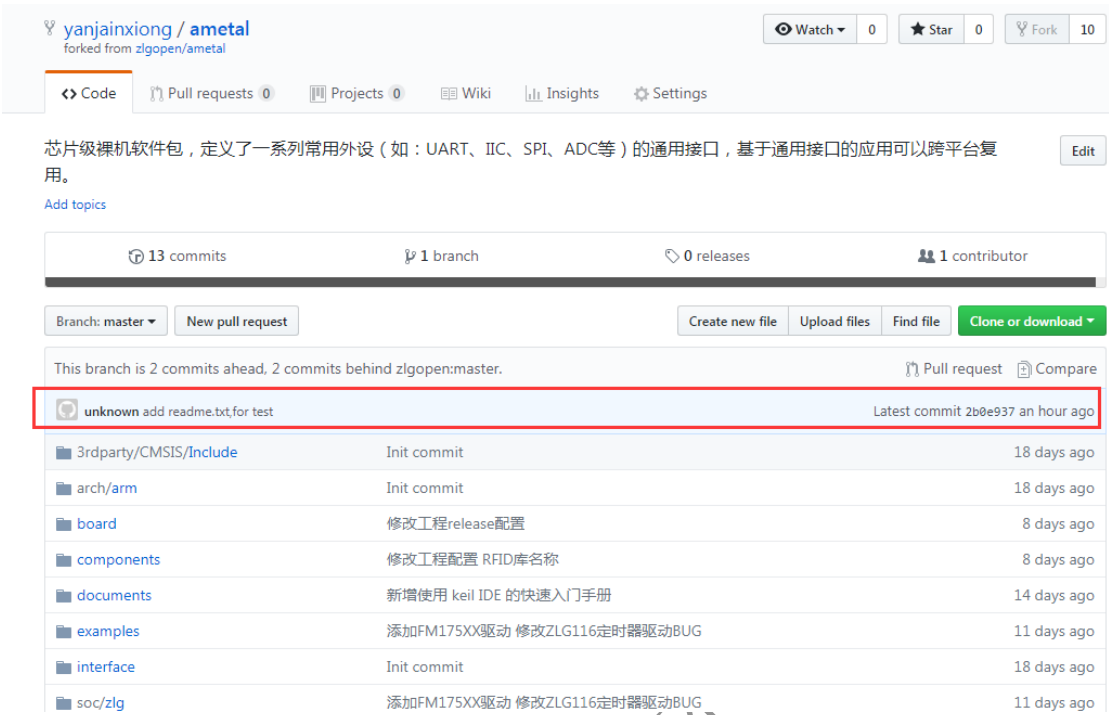


图 27 查看自己的修改内容

2. 检查修改，确认无误，如图 28 所示。

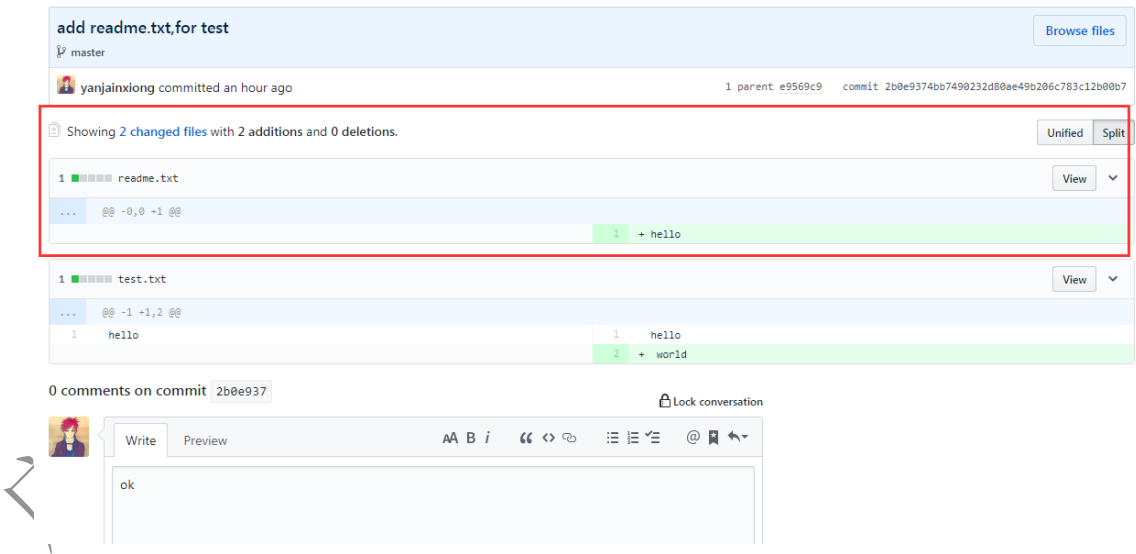


图 28 检查修改

3. 创建新的 pull 请求，如图 29 所示。

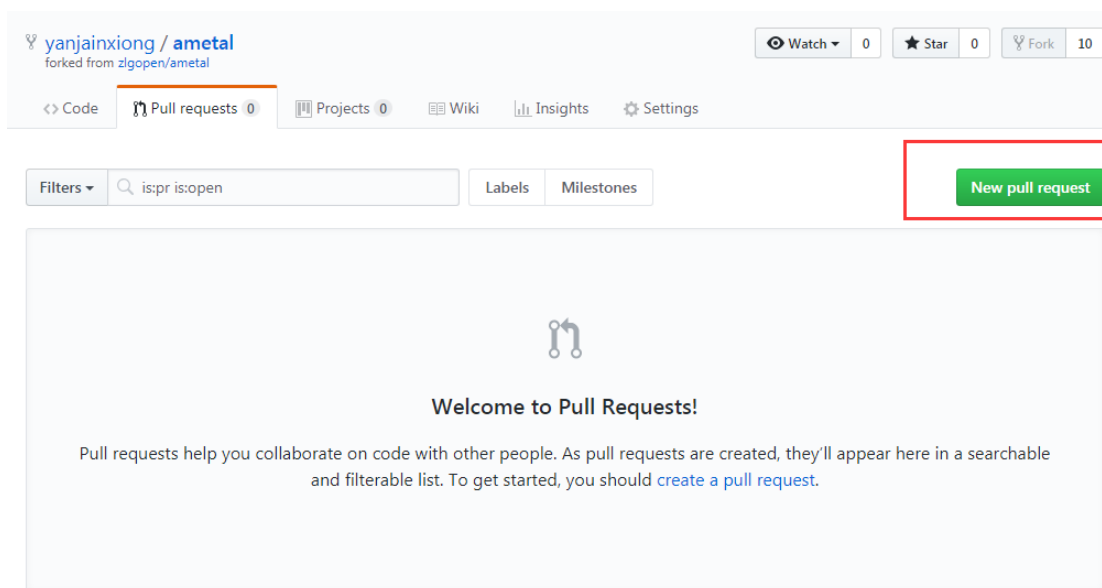


图 29 点击红框所示添加请求

4. 发起 pull 请求，如图 30 所示。

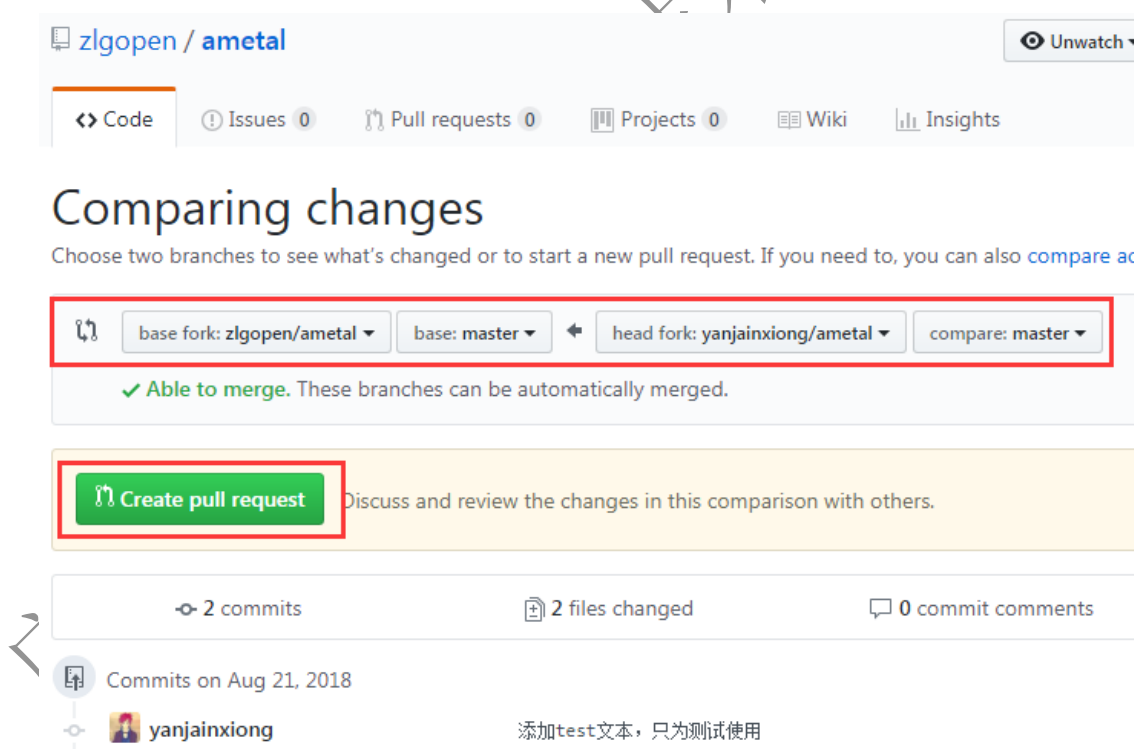


图 30 发起 pull 请求

5. 添加修改内容的描述主题等，方便原库管理人员检查，如图 31 所示。

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base fork: zlgopen/amental

base: master

head fork: yanjainxiong/amental

compare: master

✓ **Able to merge.** These branches can be automatically merged.



add readme.txt,just for test

Write

Preview

AA B i “ < > 🔗 ☰ ☷ ☰ @ 📌 ↶

Leave a comment

Attach files by dragging & dropping, [selecting them](#), or pasting from the clipboard.

☒ Allow edits from maintainers. [Learn more](#)

Create pull request

图 31 添加申请原因

- 在合并之前，可以关闭请求，如图 32 所示。

593

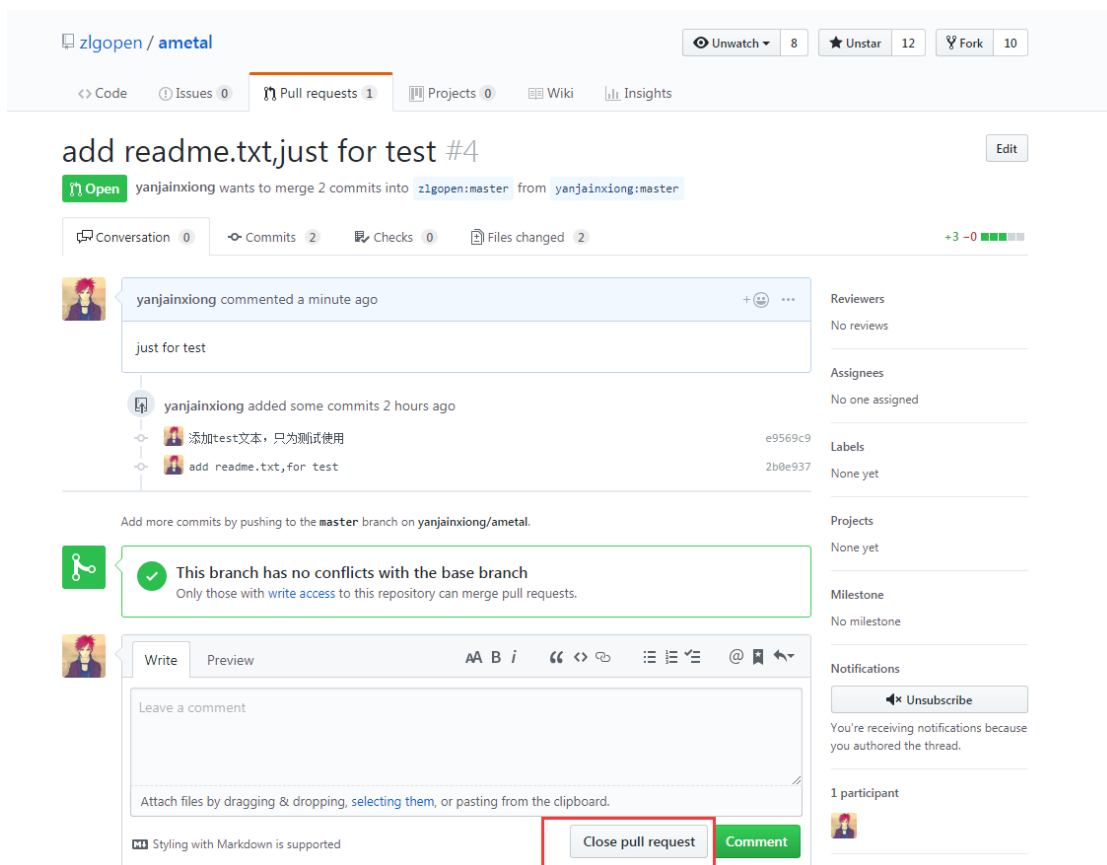


图 32 关闭合并请求

更多关于 git 的使用,可到 <https://www.zlg.cn/ametal/down/index.html> 下载相关参考资料,这里不在细述。

