目录

[第1章 AMetal平台简介与获取 1](#_Toc6175764)

[1.1 AMetal简介 1](#_Toc6175765)

[1.1.1 AMetal起源 1](#_Toc6175766)

[1.1.2 AMetal发展历程 1](#_Toc6175767)

[1.1.3 AMetal特点 1](#_Toc6175768)

[1.2 AMetal架构 2](#_Toc6175769)

[1.2.1 AMetal架构 2](#_Toc6175770)

[1.2.2 AMetal目录架构 3](#_Toc6175771)

[1.2.3 AMetal工程架构 5](#_Toc6175772)

[1.3 AMetal入门 6](#_Toc6175773)

[1.3.1 源码获取与贡献 6](#_Toc6175774)

[1.3.2 编译环境介绍 6](#_Toc6175775)

[第2章 硬件平台AM116-Core 7](#_Toc6175776)

[2.1 ZLG116微控制器 7](#_Toc6175777)

[2.1.1 特性 7](#_Toc6175778)

[2.1.2 概述 8](#_Toc6175779)

[2.2 AM116-Core 11](#_Toc6175780)

[2.2.1 电源电路 11](#_Toc6175781)

[2.2.2 最小系统 12](#_Toc6175782)

[2.2.3 复位与调试电路 12](#_Toc6175783)

[2.2.4 板载外设电路 13](#_Toc6175784)

[2.2.5 跳帽使用 16](#_Toc6175785)

[2.2.6 拓展接口说明 16](#_Toc6175786)

[第3章 AMetal接口定义 20](#_Toc6175787)

[3.1 LED通用接口 20](#_Toc6175788)

[3.1.1 定义接口 20](#_Toc6175789)

[3.1.2 实现接口 20](#_Toc6175790)

[3.2 温度采集接口 23](#_Toc6175791)

[3.2.1 定义接口 23](#_Toc6175792)

[3.2.2 实现接口 23](#_Toc6175793)

[3.3 通用按键接口 25](#_Toc6175794)

[3.3.1 定义接口 25](#_Toc6175795)

[3.3.2 实现接口 26](#_Toc6175796)

[3.4 通用数码管接口 26](#_Toc6175797)

[3.4.1 定义接口 26](#_Toc6175798)

[3.4.2 实现接口 28](#_Toc6175799)

[3.5 待续 44](#_Toc6175800)

[第4章 AMetal外设应用详解 45](#_Toc6175801)

[4.1 GPIO外设应用详解 45](#_Toc6175802)

[4.1.1 新建GPIO工程 45](#_Toc6175803)

[4.1.2 编写GPIO LED程序 46](#_Toc6175804)

[4.1.3 编写GPIO按键程序 47](#_Toc6175805)

[4.2 UART外设应用详解 48](#_Toc6175806)

[4.2.1 新建UART工程 48](#_Toc6175807)

[4.2.2 UART介绍 48](#_Toc6175808)

[4.2.3 编写UART应用程序 49](#_Toc6175809)

[4.3 I2C外设应用详解 53](#_Toc6175810)

[4.3.1 I2C主机程序 53](#_Toc6175811)

[4.3.2 I2C从机程序 55](#_Toc6175812)

[4.4 SPI外设应用详解 58](#_Toc6175813)

[4.4.1 新建SPI工程 58](#_Toc6175814)

[4.4.2 SPI总线介绍 58](#_Toc6175815)

[4.4.3 SPI初始化 59](#_Toc6175816)

[4.4.4 SPI接口函数 59](#_Toc6175817)

[4.4.5 SPI从机初始化 60](#_Toc6175818)

[4.4.6 Flash擦读写 64](#_Toc6175819)

[4.4.7 SPI flash读写范例 66](#_Toc6175820)

[4.5 ADC外设应用详解 67](#_Toc6175821)

[4.6 定时器外设应用详解 69](#_Toc6175822)

[4.6.1 Timer外设应用详解 69](#_Toc6175823)

[4.6.2 PWM外设应用详解 72](#_Toc6175824)

[4.6.3 CAP外设详解 74](#_Toc6175825)

[第5章 AMetal组件详解 78](#_Toc6175826)

[5.1 矩阵键盘组件 78](#_Toc6175827)

[5.2 UART缓冲接口 80](#_Toc6175828)

[5.3 E2PROM组件 83](#_Toc6175829)

[5.3.1 器件介绍 83](#_Toc6175830)

[5.3.2 初始化 84](#_Toc6175831)

[5.3.3 读写函数 85](#_Toc6175832)

[5.3.4 NVRAM通用接口函数 87](#_Toc6175833)

[5.4 SPI Flash组件 93](#_Toc6175834)

[5.4.1 基本功能 93](#_Toc6175835)

[5.4.2 初始化 93](#_Toc6175836)

[5.4.3 接口函数 95](#_Toc6175837)

[5.4.4 MTD通用接口函数 98](#_Toc6175838)

[5.4.5 FTL通用接口函数 102](#_Toc6175839)

[5.4.6 微型数据库 107](#_Toc6175840)

[5.5 数码管组件 109](#_Toc6175841)

[5.6 72128组件 109](#_Toc6175842)

[5.7 液晶驱动组件 109](#_Toc6175843)

[5.8 RTC组件 109](#_Toc6175844)

[5.9 读卡组件 109](#_Toc6175845)

[5.9.1 读卡通信设计 110](#_Toc6175846)

[5.9.2 设备控制类接口函数 111](#_Toc6175847)

[5.9.3 操作接口函数 112](#_Toc6175848)

[5.9.4 密钥和权限控制 114](#_Toc6175849)

[5.10 LSM6DSL传感器组件 116](#_Toc6175850)

[5.10.1 器件简介 116](#_Toc6175851)

[5.10.2 LSM6DSL初始化 116](#_Toc6175852)

[5.10.3 LSM6DSL接口函数 118](#_Toc6175853)

[5.11 BMG160传感器组件 121](#_Toc6175854)

[5.11.1 器件介绍 121](#_Toc6175855)

[5.11.2 BMG160初始化 121](#_Toc6175856)

[5.11.3 BMG160接口函数 124](#_Toc6175857)

[5.12 BME280传感器组件 126](#_Toc6175858)

[5.12.1 器件介绍 126](#_Toc6175859)

[5.12.2 BME280初始化 126](#_Toc6175860)

[5.12.3 BME280获取数据 128](#_Toc6175861)

[5.13 BMP280传感器组件 131](#_Toc6175862)

[5.13.1 器件介绍 131](#_Toc6175863)

[5.13.2 BMP280初始化 131](#_Toc6175864)

[5.13.3 BMP280接口函数 133](#_Toc6175865)

[5.14 BH1730FVC传感器组件 136](#_Toc6175866)

[5.14.1 器件介绍 136](#_Toc6175867)

[5.14.2 BH1730FVC初始化 138](#_Toc6175868)

[5.14.3 BH1730FVC接口函数 140](#_Toc6175869)

[5.15 BMA253传感器组件 141](#_Toc6175870)

[5.15.1 器件介绍 141](#_Toc6175871)

[5.15.2 BMA253初始化 141](#_Toc6175872)

[5.15.3 BMA253接口函数 143](#_Toc6175873)

[5.16 LPS22HB传感器组件 146](#_Toc6175874)

[5.16.1 器件介绍 146](#_Toc6175875)

[5.16.2 LPS22HB初始化 146](#_Toc6175876)

[5.16.3 LPS22HB接口函数 148](#_Toc6175877)

[5.17 LTR-553ALS传感器组件 151](#_Toc6175878)

[5.17.1 器件介绍 151](#_Toc6175879)

[5.17.2 LTR-553ALS初始化 151](#_Toc6175880)

[5.17.3 LTR\_553ALS接口函数 154](#_Toc6175881)

[5.18 HTS221 传感器组件 157](#_Toc6175882)

[5.18.1 器件介绍 157](#_Toc6175883)

[5.18.2 HTS221初始化 157](#_Toc6175884)

[5.18.3 HTS221接口函数 159](#_Toc6175885)

[5.19 LIS3MDL传感器组件 161](#_Toc6175886)

[5.19.1 器件介绍 161](#_Toc6175887)

[5.19.2 LIS3MDL初始化 161](#_Toc6175888)

[5.19.3 LIS3MDL接口函数 164](#_Toc6175889)

[5.20 蜂鸣器组件 166](#_Toc6175890)

[5.20.1 器件介绍 166](#_Toc6175891)

[5.20.2 蜂鸣器接口函数 167](#_Toc6175892)

[5.20.3 无源蜂鸣器驱动 168](#_Toc6175893)

[5.21 M150-II打印机组件 169](#_Toc6175894)

[5.21.1 器件介绍 169](#_Toc6175895)

[5.21.2 M150-II打印机初始化 169](#_Toc6175896)

[5.21.3 M150-II打印机接口函数 169](#_Toc6175897)

[5.22 HC595组件 169](#_Toc6175898)

[5.22.1 器件介绍 169](#_Toc6175899)

[5.22.2 HC595初始化 169](#_Toc6175900)

[5.22.3 HC595接口函数 169](#_Toc6175901)

# AMetal平台简介与获取

## AMetal简介

AMetal是一个芯片级的裸机软件包，定义了跨平台的通用接口（使得基于AMetal的应用程序可以和具体硬件完全分离，实现跨平台复用），并提供了一系列驱动及常用的软件服务。

### AMetal起源

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外设可能千差万别，例如，对于UART，不同芯片所对应的寄存器含义、波特率计算方法，中断处理方法等可能全然不同，这给广大嵌入式开发人员带来了诸多烦恼（例如，出于芯片资源、芯片供货情况等种种原因，产品需要更换使用的芯片，此时，底层大部分操作硬件的程序以及与之关联的应用程序可能都需要全部推翻重写）。实际上，对于同类外设，虽然底层差异性很大，但用途却是基本相同的，无论何种芯片上的UART外设，其提供的核心功能无非是发送串口数据或接收串口数据。对于上层操作系统而言，例如，AWorks、Linux等，对各个外设都需要编写驱动。在编写特定操作系统下的驱动时，必须要熟悉特定的驱动框架及操作系统调用，这往往会花费开发人员相当大的精力。对于同一个外设而言，如果要支持多个操作系统，就需要编写多个驱动，其实，驱动底层对硬件的操作是有相通之处的。

AMetal平台就是为了解决传统嵌入式开发模式难道高、周期长的现状，实现软件的可复用、跨平台而发展起来的。

### AMetal发展历程

2014年，周立功教授提出了软件要可复用、跨平台、标准化的愿景，随后创建了全新的AMetal平台及口号：一次编程，终生使用。

* 2014年9月1日，写下了AMetal的第一行代码；
* 2015年12月15日，以board对应SDK包的形式发布全新的AMetal平台；
* 2016年11月24日，AMetal的第一个应用案例“广州慧城第二代读卡器”上线；
* 2017年8月18日，AMetal的第一本教程《面向AMetal框架与接口编程》出版；
* 2018年4月26日，以board对应SDK包的形式开源发布；
* 2018年8月3日，AMetal软件包上传github托管，正式开源。

### AMetal特点

AMetal 对同一类外设进行了抽象，为用户提供了与具体芯片无关、仅与外设功能相关的通用接口，屏蔽了不同芯片底层的差异性，基于外设功能设计了相应的通用接口（也称之为标准接口），使得不同厂商、型号的MCU外设都能以通用接口操作。外设驱动基于AMetal通用接口编写，驱动将与具体芯片无关，只要AMetal支持了该芯片，为上层操作系统编写的驱动也就支持了该芯片。同时，AMetal也提供了直接操作寄存器的接口，用户可直接使用这部分接口操作寄存器，实现特殊功能。

总的来说，AMetal有以下特点：

* 开源、高效、组件齐全，提供详细的demo程序，支持丰富的硬件平台；
* 独立的命名空间，将外设操作标准化，避免上层软件、驱动的再次开发；
* 开放外设所有功能特性，提供跨平台标准API接口，不依赖操作系统服务。

## AMetal架构

### AMetal架构

AMetal共分为3层，硬件层、驱动层和标准接口层，详见图1.1。

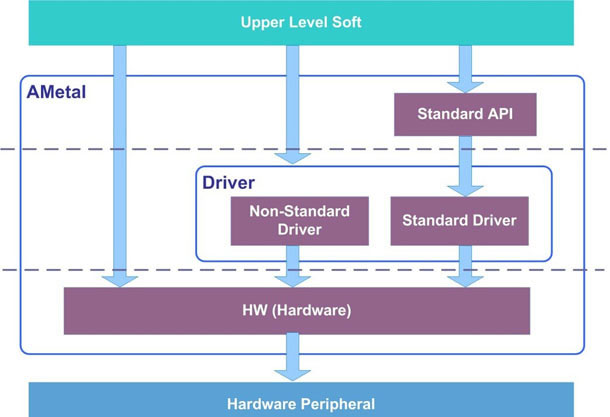


图1.1 AMetal架构

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

#### 硬件层

硬件层对SOC做最原始封装，其提供的API基本上是直接操作寄存器的内联函数，效率最高。当需要操作外设的特殊功能，或者对效率等有需求时，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。更多的硬件层接口定义及示例请参考ametal\ documents\《AMetal API参考手册.chm》或ametal\soc\zlg\drivers文件夹中的相关文件。

#### 驱动层

虽然硬件层对外设做了封装，但其通常与外设寄存器的联系比较紧密，用起来比较繁琐。为了方便使用，驱动层在硬件层的基础上做了进一步封装，进一步简化对外设的操作。根据是否实现了标准层接口可以划分为标准驱动和非标准驱动，前者实现了标准层的接口，例如 GPIO、UART、SPI 等常见的外设；后者因为某些外设的特殊性，并未实现标准层接口，需要自定义接口，例如DMA等。驱动层接口使用am\_/AM\_ + 芯片名作为命名空间，如 am\_zlg116、AM\_ZLG116。更多的驱动层接口定义及示例请参考ametal \documents\《AMetal API 参考手册.chm》或ametal\soc\zlg\drivers文件夹中的相关文件。

#### 标准接口层

标准接口层对常见外设的操作进行了抽象，提取出了一套标准 API 接口，可以保证在不同的硬件上，标准API的行为都是一样的。标准层接口使用am\_/AM\_ 作为命名空间。更多的标准接口定义及示例请参考ametal\documents\《AMetal API 参考手册.chm》或者 ametal \interface文件夹中的相关文件。

### AMetal目录架构

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1.2， 表1.1简单描述了这些文件夹及文件的功能。

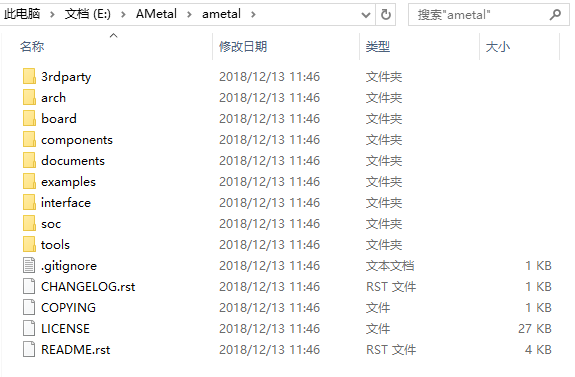


图1.2 AMetal根目录架构

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； |

#### 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。

#### arch

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

#### 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。

#### 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。

#### 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。

#### examples

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

#### 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。

#### 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。

#### tools

tools目录下存放SDK相关工具，如Keil的PACK包。

### AMetal工程架构

打开 Keil 版本的 template\_am116\_core模板工程，其工程架构详见图1.3。

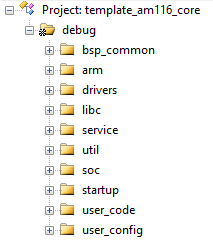


图1.3 AMetal Keil模板工程架构

在工程架构中，包含了bsp\_common、arm、drivers、libc、server、util、soc、startup、user\_code、user\_config 共10个节点。

* bsp\_common包含一些通用的板级文件，例如C库的适配文件“am\_bsp\_armlib.c”、系统堆的适配文件“arm\_bsp\_system\_heap.c”等；
* arm文件夹存放了与内核相关的通用文件，如NVIC、Systick等；
* drivers包含外围设备的驱动文件，如ZLG71128驱动；
* libc包含了C库适配器文件，如arm\_adpater，主要用于适配相应的C库；
* server包含通用服务组件的驱动文件，如ADC、蜂鸣器等；
* util包含AMetal通用辅助工具文件，如堆管理器、软件定时器以及打印输出函数等；
* soc文件夹主要包含了与芯片密切相关的文件，主要是硬件层和驱动层文件；
* startup文件夹包含了一些与芯片相关的启动文件；
* user\_code下为用户程序，每个工程对应的应用程序均存放在该目录下，该目录下默认有一个main.c文件，其中包含了AMetal 软件包的应用程序入口函数 am\_main()。用户开发的其它程序源文件均应存放在user\_code目录下；
* user\_config下为配置文件，不同工程可以有不同的配置。

## AMetal入门

### 源码获取与贡献

AMetal由广州致远电子有限公司开发，是完全开源的，相关资源共享在github平台上 (链接：<https://github.com/zlgopen/ametal>)。用户可直接通过此页面下载AMetal开源代码包，点击“Clone or download”，再点击“Download ZIP”即可下载开源代码包，详见图1.4。

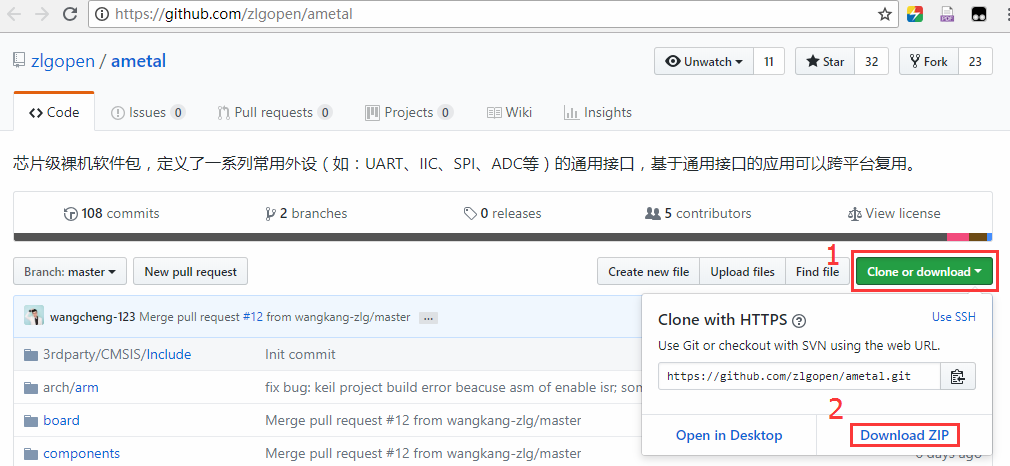


图1.4 在Github上下载AMetal代码包

下载解压后，在“ametal-master/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。由于AMetal 开源项目在持续的更新，用户在使用ZIP方式下载时，有如下几处缺陷：

* 不易对代码进行维护；
* 不易具体的知道代码更新情况；
* 不易定位更新代码位置。

用户可参考“ametal-master/documents”目录下的“AMeta代码仓库使用说明(Tortoise Git).pdf”，搭建好Git环境。使用Git可以直接在本地进行更新、获取更新信息并且能定位更新文件，使得维护代码变得非常简单。

在本地工作时，若开发者编写了比较优秀的代码或者在使用AMetal过程中发现了某些错误并对其进行了修改，开发者可以将添加或者修改的代码上传至版本库，向 AMetal贡献代码。贡献代码的相关操作流程参照上述文档的第四章。

### 编译环境介绍

AMetal支持的编译环境包括Keil和eclipse；同样，用户可参考“ametal-master/documents”目录下的“快速入门手册(eclipse) .pdf”以及“快速入门手册(keil).pdf”来进行IDE的下载和环境的搭建，包括示例程序的编译与下载。

# 硬件平台AM116-Core

## ZLG116微控制器

### 特性

* 内核与系统：

－32位ARM Cortex-M0处理器内核，最高工作频率可达48MHZ；

－单指令周期32位硬件乘法器。

* 存储器：

－高达128K字节的闪存程序存储器；

－高达8K字节的SRAM；

－Boot loader支持片内Flash、UART在线用户编程（IAP）/在线系统编程（ISP）。

* 时钟、复位和电源管理：

－2.0V~5.5V供电；

－上电/断电复位（POR/PDR）、可编程电压监测器（PVD）；

－外部8~24MHz高速晶体振荡器；

－内嵌经出厂调校的48MHz高速振荡器；

－内嵌40KHz低速振荡器；

－PLL支持CPU最高运行在48MHz。

* 低功耗：

－睡眠、停机和待机模式。

* 1个12位模数转换器，1μS转换时间（多达10个输入通道）：

－转换范围：0~VDDA。

* 2个比较器。
* 5通道DMA控制器：

－支持的外设：Timer、UART、I2C、SPI和ADC。

* 多达39个快速I/O端口：

－所有I/O口可以映像到16个外部中断；

－所有端口均可输入输出5V信号。

* 调试模式：

－串行单线调试（SWD）。

* 多达9个定时器：

－1个16位高级定时器4通道高级控制定时器，有4通道PWM输出，以及死区生成和紧急停止功能；

－1个32位定时器和1个16位定时器，有高达4个IC/OC，可用于IR控制解码；

－2个16位定时器，有1个IC/OC和1个OCN，死区生成，紧急停止，调制器门电路用于IR控制；

－1个16位定时器，有1个IC/OC；

－2个看门狗定时器（独立的和窗口型的）；

－系统时间定时器：24位自减型计数器。

* 多达 5 个通信接口：

－2个UART接口；

－1个I2C接口；

－2个SPI接口。

* 96位的芯片唯一ID（UID）。
* 采用QFN32封装。

### 概述

如图2.1所示的ZLG116系列微控制器（MCU）具有丰富的片上外设，除GPIO外，还包含1个12位的ADC、4个16位通用定时器、1个32位通用定时器、1个高级PWM定时器，还包含标准的通信接口：2个UART接口、1个I2C接口和2个SPI接口。

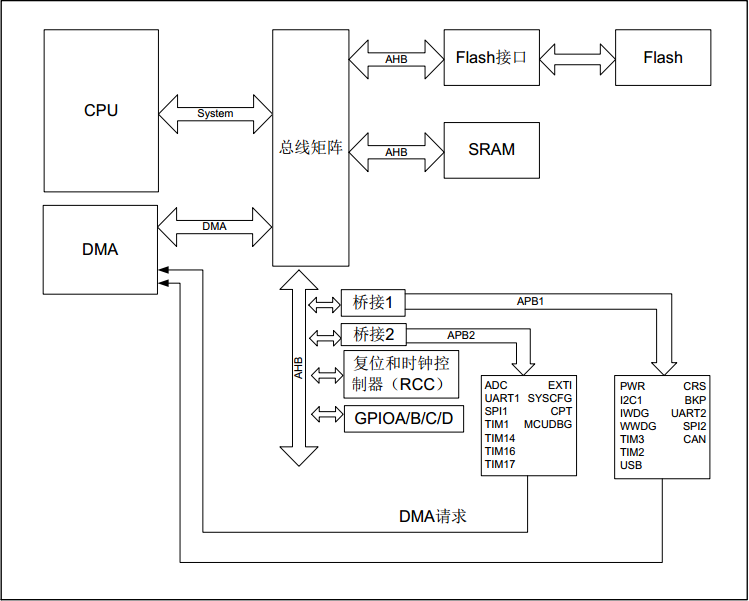


图2.1 ZLG116功能框图

ZLG116产品系列工作电压为2.0V~5.5V，工作温度范围包含-40℃~+85℃常规型和-40℃~+105℃扩展型。多种省电工作模式保证低功耗应用的要求。

虽然这些概念对于初学者来说可能会感到非常陌生，但也不要害怕，如同你使用的计算机一样，尽管也很复杂，但毫不影响你使用计算机编程和上网。

ZLG116N32A/ZLG217P64A同属于ZLG系列MCU，ZLG217P64A是ZLG116N32A的增强版本，不仅内核升级到Cortex—M3，而且外设资源更加丰富，可以更好地满足不同场合的应用需求。可以根据不同需求进行合理选择，详见表2.1。

表2.1 ZLG系列MCU选型表

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 器件型号 | 内核 | Flash （KB） | RAM （KB） | UART | I2C | SPI | 比较器 | I/O | ADC | 封装 |
| ZLG116N32A | CortexM0 | 64 | 8 | 2 | 1 | 1 | 2 | 27 | 10/12bit | QFN32 |
| ZLG217P64A | CortexM3 | 128 | 20 | 3 | 2 | 2 | 2 | 51 | 16/12bit | LQFP64 |

ZLG116N32A MCU只有一种封装，为QFN32，其引脚分布详见图2.2。



图2.2 封装示意图

ZLG116N32A系列MCU的引脚描述及主要功能详见表2.2。

表2.2 ZLG116 MCU的引脚描述

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 引脚编号 | 引脚名称 | 类型 | I/O电平 | 主功能 | 可选的复用功能 | 附加功能 |
| 0 | DVSS | S | - | DVSS | - | - |
| 1 | DVDD | S | - | DVDD | - | - |
| 2 | PD0-OSC\_IN | I | - | OSC\_IN | - | OSC\_IN |
| 3 | PD1-OSC\_OUT | O | - | OSC\_OUT | - | OSC\_OUT |
| 4 | NRST | I/O | - | NRST | - | - |
| 5 | VDDA | S | - | VDDA | - | - |
| 6 | PA0-WKUP | I/O | - | PA0 | TIM2\_CH1\_ETR/ UART2\_CTS/ADC\_IN0 | WKUP/  COMP1\_OUT |
| 7 | PA1 | I/O | - | PA1 | TIM2\_CH2/UART2\_RTS/ ADC\_IN1 | - |
| 8 | PA2 | I/O | - | PA2 | TIM2\_CH3/ UART2\_TX/  ADC\_IN2 | COMP2\_OUT |
| 9 | PA3 | I/O | - | PA3 | TIM2\_CH4/ UART2\_RX/  ADC\_IN3 | - |
| 10 | PA4 | I/O | - | PA4 | SPI1\_NSS/ TIM14\_CH1/  ADC\_IN4 | - |
| 11 | PA5 | I/O | - | PA5 | SPI1\_SCK/ TIM2\_CH1\_ETR/  ADC\_IN5 | - |
| 12 | PA6 | I/O | - | PA6 | SPI1\_MISO/TIM3\_CH1/ TIM16\_CH1/TIM1\_BKIN/  ADC\_IN6 | COMP1\_OUT |
| 13 | PA7 | I/O | - | PA7 | TIM1\_CH1N/SPI1\_MOSI/TIM3\_CH2/ TIM14\_CH1/TIM17\_CH1/  ADC\_IN7 | COMP2\_OUT |
| 14 | PB0 | I/O | - | PB0 | TIM3\_CH3/TIM1\_CH2N/ ADC\_IN8 | - |
| 15 | PB1 | I/O | - | PB1 | TIM3\_CH4/TIM14\_CH1/TIM1\_CH3N/ ADC\_IN9 | - |
| 16 | PB2 | I/O | FT | PB2 | - | - |
| 17 | DVDD | S | - | DVDD | - | - |
| 18 | PA8 | I/O | FT | PA8 | TIM1\_CH1/MCO |  |
| 19 | PA9 | I/O | FT | PA9 | UART1\_TX/TIM1\_CH2/ UART1\_RX/I2C\_SCL/ MCO | - |
| 20 | PA10 | I/O | FT | PA10 | UART1\_RX/TIM1\_CH3/UART1\_TX/TIM17\_BKIN/I2C\_SDA | - |
| 21 | PA11 | I/O | FT | PA11 | UART1\_CTS/TIM1\_CH4/I2C\_SCL | COMP1\_OUT |
| 22 | PA12 | I/O | FT | PA12 | UART1\_RTS/TIM1\_ETR/ I2C\_SDA | COMP2\_OUT |
| 23 | PA13 | I/O | FT | PA13 | SWDIO | - |
| 24 | PA14 | I/O | FT | PA14 | SWCLK/UART2\_TX | - |
| 25 | PA15 | I/O | FT | PA15 | TIM2\_CH1\_ETR/  SPI1\_NSS/UART2\_RX | - |
| 26 | PB3 | I/O | FT | PB3 | TIM2\_CH2/SPI1\_SCK | - |
| 27 | PB4 | I/O | FT | PB4 | TIM3\_CH1/SPI1\_MISO | - |
| 28 | PB5 | I/O | FT | PB5 | TIM3\_CH2/SPI1\_MOS/TIM16\_BKN | - |
| 29 | PB6 | I/O | FT | PB6 | UART1\_TX/I2C\_SCL/TIM16\_CH1N | - |
| 30 | PB7 | I/O | FT | PB7 | UART1\_RX/I2C\_SDA/TIM17\_CH1N | - |
| 31 | BOOT0 | I | - | BOOT0 | - | - |
| 32 | PB8 | I/O | FT | PB8 | I2C\_SCL/TIM16\_CH1 | - |

## AM116-Core

AM116开发套件包括AM116-Core和MiniCK100仿真器，如图2.3所示为AM116-Core的示意图，MCU为ZLG的ZLG116，包括2个MiniPort接口、1个MicroPort接口和1个2×10扩展接口。这些接口不仅将MCU的所有I/O资源引出，还可以借助MiniPort接口和MicroPort接口外扩多种模块。片上资源包括2个LED发光二极管、1个无源蜂鸣器、1个加热电阻、1个LM75B测温芯片、1个热敏电阻、1个多功能独立按键和1个复位按键，可以完成多种基础实验。

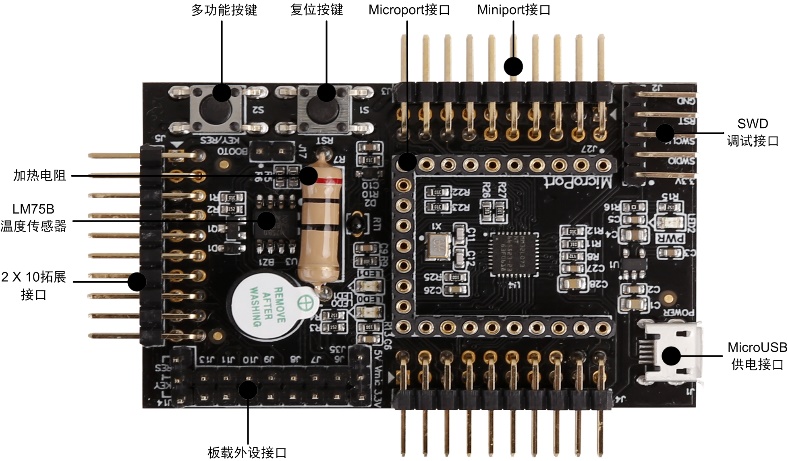


图2.3 AM116-Core开发板接口分布

AM116-Core的出现简化了用户的硬件设计，使得学习ZLG系列MCU的难度大大降低，可以帮助初学者快速掌握基于32位Cortex™-M0内核微控制器的应用开发。

### 电源电路

AM116-Core开发板套件采用USB供电，USB输入电压为5V，需要将电压转为3.3V给ZLG116N32A、LM75B和蜂鸣器使用。为了实现5V到3.3V的转换，AM116-Core采用安森美的一颗CMOS LDO芯片--CAT6219-330TD。

LDO（即Low Dropout Regular）是相对于传统的线性稳压器来说的，意为低压差线性稳压器。相比较于传统的线性稳压器，如78xx芯片。LDO对于输入与输出的压差要求更小，非常适合在5V转3.3V应用。

CAT6219-330TD的主要特性：

* 输入电压为3.3V~5.5V，最大输出电流可达500mA；
* 负载为500mA时典型压差为0.3V；
* SOT-23-5封装。

CAT6219-330TD的典型应用电路如图2.4所示。整个电路非常简单，芯片的输入端和输出端分别接有两个滤波电容。其中电容值大的电容的主要作用是滤除低频的交流信号，电容值小的电容主要作用是滤除高频的交流信号，通过滤波电容保障电压的稳定，减少毛刺干扰。

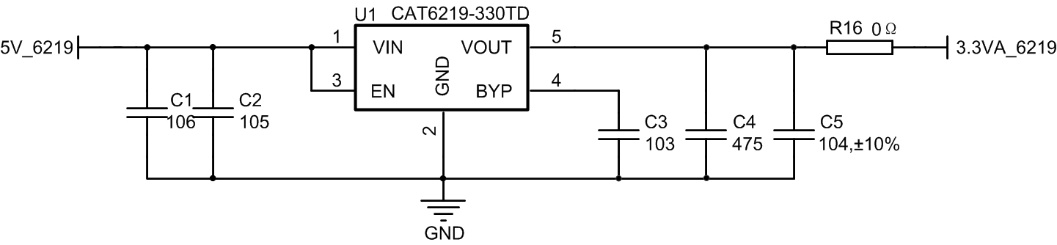


图2.4 CAT6219-330TD电源电路

### 最小系统

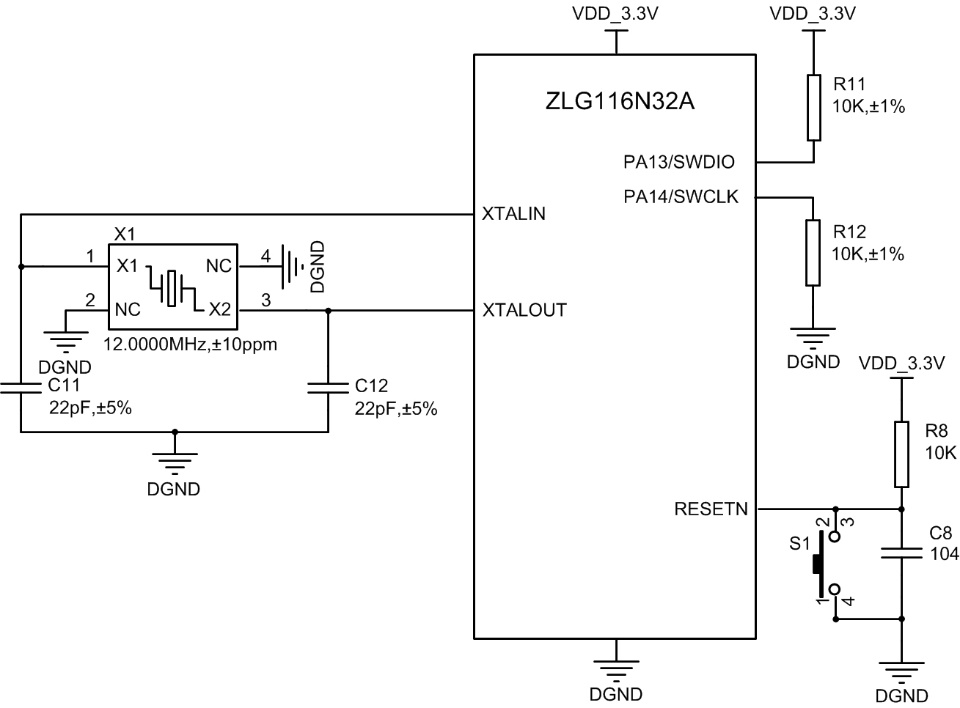


图2.5 最小系统电路

ZLG116N32A微控制器的最小系统电路主要包括复位电路、时钟电路和SWD调试接口电路等三部分，详见图2.5。

### 复位与调试电路

#### 复位电路

AM116-Core开发板的复位电路采用常见的RC复位电路，详见图2.6。按键S1按下或者网络标号为RST的位置给一个低电平脉冲，从而通过RESET引脚给MCU一个复位脉冲。

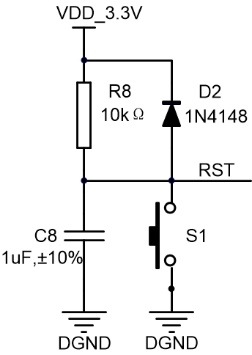


图2.6 按键复位电路

#### 调试电路

ZLG116N32A微控制器支持ARM串行调试模式（SWD）；相对于JTAG调试模式来说，SWD调试模式速度更快且使用的I/O口更少，因此AM116-Core开发板板载了SWD调试接口，其参考电路如图2.7所示。

表2.3 调试管脚说明

|  |  |  |
| --- | --- | --- |
| 管脚号 | 标号 | 芯片引脚 |
| 1 | 3.3V | 3.3V |
| 2 | SWDIO | PA13 |
| 3 | SWCLK | PA14 |
| 4 | RST | NRST |
| 5 | GND | 地 |



图2.7 SWD调试接口

### 板载外设电路

#### LED电路



图2.8 板载LED电路

AM116-Core开发板板载了两路LED发光二极管，可以完成简单的显示任务，电路如图2.8所示，LED为低电平有效（低电平有效）。LED电路的控制引脚与微控制器的I/O引脚通过J9和J10相连。电路中的R3和R4为LED的限流电阻，选择1.5kΩ这个值可以避免LED点亮时过亮。

AM116-Core开发板对应的引脚分配详见表2.4。

表2.4 LED电路微控制器对应引脚

|  |  |
| --- | --- |
| 引脚标号 | 微控制器引脚 |
| PIO\_LED0 | PB1 |
| PIO\_LED1 | PB2 |

#### 蜂鸣器电路

为了便于调试，AM116-Core开发套件设计了蜂鸣器驱动电路，如图2.9所示。AM116-Core开发板使用的是无源蜂鸣器。D1起保护三极管的作用，当突然截止时无源蜂鸣器两端产生瞬感应电动势可以通过D1迅速释放掉，避免叠加效应击穿三极管集电极。若使用有源蜂鸣器则D1不用焊接。当不使用蜂鸣器的时候也可以用J7断开蜂鸣器电路与I/O口的连接。

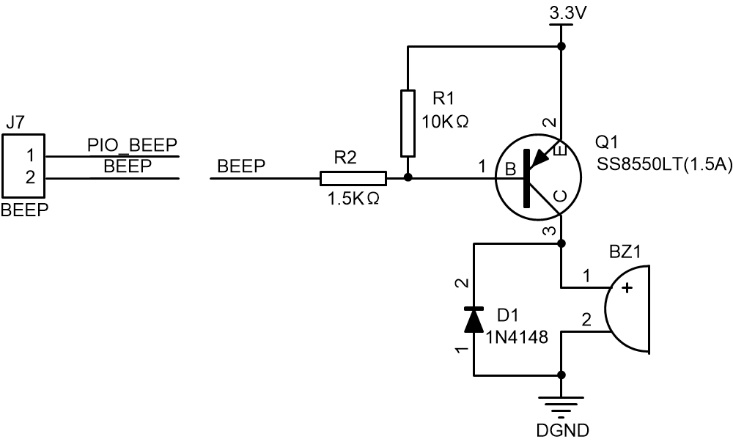


图2.9 板载蜂鸣器电路

AM824-Core开发板对应的引脚分配详见表2.5。

表2.5 蜂鸣器电路微控制器对应引脚

|  |  |
| --- | --- |
| 引脚标号 | 微控制器引脚 |
| PIO\_BEEP | PB8 |

#### 加热电阻与按键电路

图2.10 加热电路

AM116-Core开发板创新性的设计了一套测温实验电路。包含加热电路和数字/模拟测温电路。其中加热电路采用了一个阻值为20~50Ω的功率电阻（2W），通过按键来控制，如图2.10所示。GPIO口（PA8）输出需要上拉电阻R8。电阻越小通过其电流越大，产生的热量越大，因此R7若焊接小电阻时，不宜加热时间过长。按键的功能需要用J14上的跳线帽来选择为加热按键。当按键按下时电路导通，电阻上产生的热量会导致电阻周围的温度上升，这时可以通过测温电路观察温度上升情况。

AM116-Core开发板的多功能按键可以当做普通按键来使用，也可以当做加热按键来使用，可以通过J14选择对应的功能。AM116-Core开发板对应的引脚分配详见表2.6。

表2.6 多功能按键电路微控制器对应引脚

|  |  |
| --- | --- |
| 引脚标号 | 微控制器引脚 |
| GPIO\_KEY | PA8 |

#### 数字测温电路

AM116-Core开发板选择LM75B作为数字测温电路的主芯片，LM75B与LM75A完全兼容，只是静态功耗会稍低一些，电路如图2.11所示。LM75B是一款内置带隙温度传感器和∑－Δ模数转换功能的温度数字转换器，它也是温度检测器，并且可提供过热输出功能。LM75B的主要特性如下：

* 具有良好的温度精度，可达0.125℃的精度；
* 较宽电源电压范围：2.8V~5.5V；
* 环境温度范围：Tamb=-55℃~+125℃；
* 较低的功耗，关断模式下消耗的电流仅为1μA；
* I2C总线接口，同一总线上可连接多达8个器件。

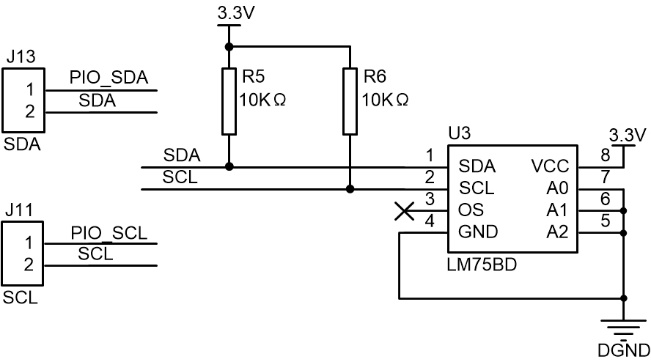


图2.11 LM75B电路

在电路设计上，R5和R6是I2C总线的上拉电阻。由于板载只有一片LM75B，不用考虑芯片的地址问题，因此芯片的A0~A2引脚可以直接接地。OS为芯片的过热输出，可以外接继电器等器件实现一个独立温控器的功能，这里由于温控是通过单片机控制的，因此这个引脚可以不使用。

AM116-Core开发板对应的引脚分配详见表2.7。

表2.7 LM75B电路微控制器对应引脚

|  |  |
| --- | --- |
| 引脚标号 | 微控制器引脚 |
| PIO\_SDA | PB7 |
| PIO\_SCL | PB6 |

#### 模拟测温电路

图2.12 热敏电阻电路

模拟测温电路一般利用热敏元件的电特定随温度变化的特点来进行测量。AM116-Core开发套件选择了热敏电阻作为测温元件，热敏电阻选用的是MF52E-103F3435FB-A，当测温范围在0~85℃，电阻变化范围为27.6~1.45KΩ。测温电路图2.12所示，采用的是简单的电阻分压电路，其中C9是为了电路输出更加稳定。单片机通过ADC采集分压电阻上的电压值，当温度变化时，热敏电阻的阻值发生变化，单片机采集到的ADC值也会发生变化。通过计算得到热敏电阻的阻值，再对比热敏电阻阻值与温度的对照表，就可以得到当前的温度值。



AM116-Core开发板对应的引脚分配详见表2.8。

表2.8 热敏电路微控制器对应引脚

|  |  |
| --- | --- |
| 引脚标号 | 微控制器引脚 |
| PIO\_ADC | PB0 |

### 跳帽使用

板载外设接口设计在微控制器引脚和板载外设电路之间，可以通过跳线帽进行短接，如图2.13所示。这样设计是为了外设电路在不使用的时候可以断开与微控制器引脚的连接，而不会影响到这些引脚的其它地方的使用。具体的引脚定义详见表2.9。



图2.13 板载外设接口引脚图示

表2.9 板载外设接口管脚说明

|  |  |  |  |
| --- | --- | --- | --- |
| 接口 | 标号 | 引脚序号 | 功能 |
| LED电路 | J9 | PB1 | 短接时，PB1与LED0控制引脚相连接，LED0低电平点亮； |
| J10 | PB2 | 短接时，PB2与LED1控制引脚相连接，LED1低电平点亮； |
| 蜂鸣器电路 | J7 | PB8 | 短接时，PB8与无源蜂鸣器控制引脚相连接； |
| 加热电阻与按键电路 | J14 | PA8 | 短接2、3时，按键用于加热；  短接1、2时，按键与PA8连接； |
| 数字测温电路 | J11 | PB6 | PB6，I2C总线SCL引脚，与LM75的SCL相连接； |
| J13 | PB7 | PB7，I2C总线SDA引脚，与LM75的SDA相连接； |
| 其他 | J8 | - | 短接时，外接看门狗电路与MCU复位连接； |
| J35 | 无 | 短接1、2时，MicroPort的“5V/3.3VA”引脚与3.3VA相连；  短接2、3时，MicroPort的“5V/3.3VA”引脚与5V相连； |

### 拓展接口说明

#### MiniPort接口

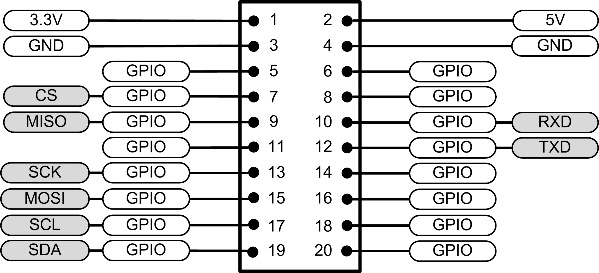


图2.14 MiniPort（2×10）接口功能说明

MiniPort（2×10）接口是一个通用板载标准硬件接口，通过该接口可以与配套的标准模块相连，便于进一步简化硬件设计和扩展。其特点如下：

* 采用标准的接口定义，采用2×10间距2.54mm的90°弯针；
* 可同时连接多个扩展接口模块；
* 具有16个通用I/O端口；
* 支持1路SPI接口；
* 支持1路I2C接口；

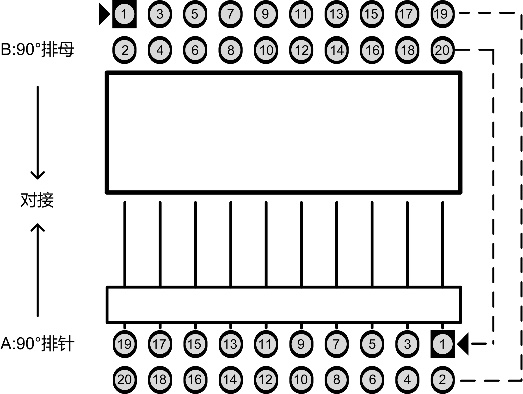


图2.15 MiniPort（2×10）接口连接器

* 支持1路UART接口；
* 支持一路3.3V和一路5V电源接口。

标准MiniPort（2×10）接口功能说明详见图2.14，MiniPort（2×10）接口使用的连接器为2.54mm间距的2×10排针/排母（90°），其封装样式详见图2.15。主控制器底板选用90°排针，功能模块选用90°排母与主机相连，同时采用90°排针将所有引脚引出，实现模块的横向堆叠。

MiniPort的90°排针与90°排母之间连接关系，A1 - B20、A2 - B19……A19 – B2、A20 - B1（A代表排针，B代表排母）。

MiniPort（2×10）目前支持的模块为MiniPort-Key按键模块、MiniPort-LED模块、MiniPort-View数码管模块和MiniPort-595模块，这些模块不仅可以直接插入MiniPort，而且还可以通过杜邦线与其它各种开发板相连。

AM116-Core开发板搭载了2路MiniPort，接口标号为J3和J4。J3与J4接口引脚完全相同，用户可根据习惯选择使用，其具体的引脚分配详见表2.10。

表2.10 MiniPort引脚分配

|  |  |  |  |
| --- | --- | --- | --- |
| 排针管脚号 | 功能 | 排针管脚号 | 功能 |
| 1 | 3.3V | 2 | 5V |
| 3 | GND | 4 | GND |
| 5 | PA0 | 6 | PA1 |
| 7 | PA4 | 8 | PA15 |
| 9 | PA6 | 10 | PA3 |
| 11 | PA11 | 12 | PA2 |
| 13 | PA5 | 14 | PA12 |
| 15 | PA7 | 16 | PB3 |
| 17 | PB6 | 18 | PB4 |
| 19 | PB7 | 20 | PB5 |

#### 2×10扩展接口说明

ZLG116N32A有32个引脚，I/O管脚有27个。由于MiniPort仅定义了16个I/O，因此还有部分I/O未引出。

为了便于扩充外围接口，因此需要将剩余的引脚全部引出，基于此，还为AM116-Core设计了一个2×10扩展接口，不仅将MiniPort未使用的I/O引出，还包含了一组电源接口和一个串口复用接口（PA10\_RXD、PA9\_TXD），详见图2.16。



图2.16 2×10扩展接口引脚图

#### MicroPort接口

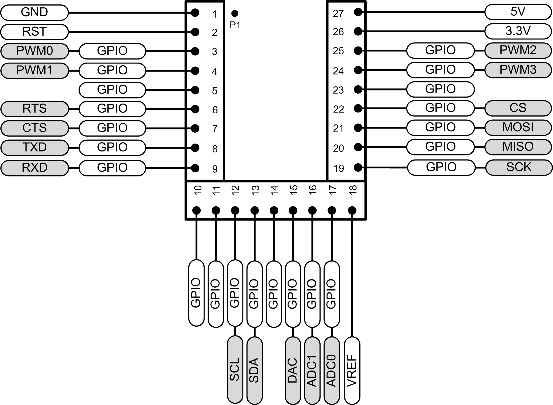


图2.17 MicroPort标准接口引脚定义

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

* 具有标准的接口定义；
* 接口包括丰富的外设资源，支持

UART、I2C、SPI、PWM、ADC等功能；

* 配套功能模块将会越来越丰富；
* 支持上下堆叠扩展。

MicroPort接口使用的连接器为2.54间距的1×9圆孔排针，高度为7.5mm，实现上下堆叠连接。MicroPort标准接口采用U型设计，三边各9个引脚，共27个引脚，其引脚功能定义详见图2.17。目前支持MicroPort接口的外设模块有：EEPROM模块（MicroPort- EEPROM）、NorFlash模块（MicroPort - NorFlash）、Zigbee模块（MicroPort - Zigbee）和PCF8563模块（MicroPort - RTC）等。

AM116-Core开发板板载1路MicroPort接口，可以支持上述几款MicroPort外设模块。用户可依据自己需求，配合各种MicroPort模块，快速灵活的实现功能扩展。AM116-Core的MicroPort接口引脚分配详见表2.11。

表2.11 AM824-Core MicroPort引脚分配表

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 接口管脚号 | 功能 | MCU管脚号 | 接口管脚号 | 功能 | MCU管脚号 | 接口管脚号 | 功能 | MCU管脚号 |
| 1 | GND | — | 10 | I/O3 | PA13 | 19 | SCK | PA5 |
| 2 | RST | NRST | 11 | I/O2 | PA14 | 20 | MISO | PA6 |
| 3 | PWM0 | PB3 | 12 | SCL | PB6 | 21 | MOSI | PA7 |
| 4 | PWM1 | PB8 | 13 | SDA | PB7 | 22 | CS | PA4 |
| 5 | I/O0 | PA15 | 14 | I/O4 | PB4 | 23 | I/O1 | PA2 |
| 6 | RTS | PA12 | 15 | DAC | PB5 | 24 | PWM3 | PA3 |
| 7 | CTS | PA11 | 16 | ADC1 | PA0 | 25 | PWM2 | NC |
| 8 | TXD | PA9 | 17 | ADC0 | PA1 | 26 | 3.3V | - |
| 9 | RXD | PA10 | 18 | VREF | - | 27 | 5V | - |

# AMetal接口定义

引言---说明接口定义规则和使用方法。

## LED通用接口

### 定义接口

#### 接口命名

由于操作的对象是LED，因此接口命名以“am\_led\_”作为前缀。显然，我们对LED的操作不外乎点亮、熄灭、翻转以及最重要的初始化。通常系统存在多个LED，为了区分不同的LED，需要为每个LED分配一个唯一的ID，基于此，将所有接口的参数设定为LED的ID，用于指定需要操作的LED，其对应的接口名为：

am\_led\_on(int led\_id)

am\_led\_off(int led\_id)

am\_led\_toggle(int led\_id)

am\_led\_gpio\_inst\_init(void)

#### 返回值

接口无特殊说明，直接将接口的返回值定义为int类型的标准错误号。LED接口的完整定义详见表3.1。

表3.1 LED通用接口

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_led\_on (int led\_id); | 打开LED |
| int am\_led\_off (int led\_id); | 关闭LED |
| int am\_led\_toggle (int led\_id); | 翻转LED |
| int am\_led\_gpio\_inst\_init (void); | 初始化LED |

### 实现接口

#### 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3.1，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3.1 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 }

#### 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3.2。

程序清单3.2 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 }

## 温度采集接口

### 定义接口

温度采集接口定义的接口就只有一个，那就是读取温度。其接口名为：

am\_temp\_read (am\_temp\_handle\_t handle, int32\_t \*p\_temp);

### 实现接口

由于温度采集设备的多样性，所以将读取温度函数的抽象化，将其定义为一个统一的接口函数，并放在一个结构体中：

struct am\_temp\_drv\_funcs {

int (\*pfn\_temp\_read) (void \*p\_drv, int32\_t \*p\_temp);

};

然后定义一个温度标准服务结构体将它包含进去：

typedef struct am\_temp\_dev {

const struct am\_temp\_drv\_funcs \*p\_funcs;

void \*p\_cookie;

} am\_temp\_dev\_t;

整个温度标准服务结构体中，仅有一个指向温度采集设备的读取温度函数的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。

以LM75为例，LM75的设备实例结构体为：

typedef struct am\_temp\_lm75 {

am\_temp\_dev\_t isa;

am\_i2c\_device\_t dev;

const am\_temp\_lm75\_info\_t \*p\_info;

} am\_temp\_lm75\_t;

其中isa为温度标准服务结构体，dev为I2C从设备描述结构体，am\_temp\_lm75\_info\_t为LM75的I2C从机地址。

LM75的初始化详见程序清单3.3。

程序清单3.3 LM75初始化

1 static am\_temp\_lm75\_t \_\_g\_temp\_lm75;

2 static const am\_temp\_lm75\_info\_t \_\_g\_temp\_lm75\_info = {

3 0x48

4 };

5 am\_temp\_handle\_t am\_temp\_lm75\_inst\_init (void)

6 {

7 return am\_temp\_lm75\_init( &\_\_g\_temp\_lm75,

8 &\_\_g\_temp\_lm75\_info,

9 am\_zlg116\_i2c1\_inst\_init());

10 }

11 am\_temp\_handle\_t am\_temp\_lm75\_init (am\_temp\_lm75\_t \*p\_lm75,

12 const am\_temp\_lm75\_info\_t \*p\_info,

13 am\_i2c\_handle\_t handle)

14 {

15 if ((p\_lm75 == NULL) || (p\_info == NULL)) {

16 return NULL;

17 }

18 am\_i2c\_mkdev(&p\_lm75->dev,

19 handle,

20 p\_info->i2c\_addr,

21 AM\_I2C\_ADDR\_7BIT | AM\_I2C\_SUBADDR\_1BYTE);

22 p\_lm75->p\_info = p\_info;

23 p\_lm75->isa.p\_funcs = &\_\_g\_temp\_lm75\_drv\_funcs;

24 p\_lm75->isa.p\_cookie = p\_lm75;

25 return &p\_lm75->isa;

26 }

初始化完成后，只需提供LM75的操作句柄，即可在任意位置调用温度读取函数，详见程序清单3.4。

程序清单3.4 LM75读取温度

1 void demo\_std\_i2c\_lm75\_entry (am\_temp\_handle\_t handle)

2 {

3 int32\_t temperature = 0;

4 AM\_FOREVER {

5 if (am\_temp\_read(handle, &temperature) != AM\_OK) {

6 AM\_DBG\_INFO("am\_temp\_read failed!\r\n");

7 } else {

8 AM\_DBG\_INFO("Current temperature is %d.%1d°C\r\n",

9 (int)temperature / 1000,

10 (int)(temperature - temperature / 1000 \* 1000) / 100);

11 }

12 am\_mdelay(1000);

13 }

14 }

## 通用按键接口

### 定义接口

由于操作的对象是按键（key），因此接口命名以“am\_key\_”作为前缀。按键一般分为普通IO按键和矩阵键盘；其中普通IO按键的初始化函数为：

int am\_key\_gpio\_init (am\_key\_gpio\_t \*p\_dev, const am\_key\_gpio\_info\_t \*p\_info);

p\_info指向按键信息结构体：

typedef struct am\_key\_gpio\_info {

const int \*p\_pins; /\*\*< \brief 使用的引脚号 \*/

const int \*p\_codes; /\*\*< \brief 各个按键对应的编码（上报） \*/

int pin\_num; /\*\*< \brief 按键数目 \*/

am\_bool\_t active\_low; /\*\*< \brief 是否低电平激活（按下为低电平） \*/

int scan\_interval\_ms; /\*\*< \brief 按键扫描时间间隔，一般10ms \*/

} am\_key\_gpio\_info\_t;

用户只需定义一个按键信息结构体并将信息填充，再定义一个am\_key\_gpio\_t的指针后调用am\_key\_gpio\_init初始化函数即可完成初始化；

矩阵键盘接口的命名稍有不同，增加了“matrix”变成了“am\_key\_matrix\_”；矩阵键盘的初始化函数为：

int am\_key\_matrix\_gpio\_softimer\_init (

am\_key\_matrix\_gpio\_softimer\_t \*p\_dev,

const am\_key\_matrix\_gpio\_softimer\_info\_t \*p\_info);

p\_info指向矩阵键盘信息结构体：

typedef struct am\_key\_matrix\_gpio\_softimer\_info {

am\_key\_matrix\_gpio\_info\_t key\_matrix\_gpio\_info;

int scan\_interval\_ms;

} am\_key\_matrix\_gpio\_softimer\_info\_t;

结构体key\_matrix\_gpio\_info包含了矩阵键盘的信息：

typedef struct am\_key\_matrix\_gpio\_info {

am\_key\_matrix\_base\_info\_t base\_info; /\*\*< \brief 矩阵键盘基础信息 \*/

const int \*p\_pins\_row; /\*\*< \brief 行线引脚 \*/

const int \*p\_pins\_col; /\*\*< \brief 列线引脚 \*/

} am\_key\_matrix\_gpio\_info\_t;

base\_info又指向矩阵键盘的基础信息：

typedef struct am\_key\_matrix\_base\_info {

int row; /\*\*< \brief 行数目 \*/

int col; /\*\*< \brief 列数目 \*/

const int \*p\_codes; /\*\*< \brief 各个按键对应的编码，按行的顺序依次对应 \*/

am\_bool\_t active\_low; /\*\*< \brief 按键按下后是否为低电平 \*/

uint8\_t scan\_mode; /\*\*< \brief 扫描方式 （按行扫描或按列扫描）\*/

} am\_key\_matrix\_base\_info\_t;

在使用的时候，需将这些信息填充完毕，再调用初始化函数。

### 实现接口

## 通用数码管接口

### 定义接口

#### 接口命名

由于操作的对象是数码管（digitron），因此接口命名以“am\_digitron\_”作为前缀。数码管最常见的操作是设置数码管的显示内容，提供一个显示字符和字符串的接口，其对应的接口名为：

am\_digitron\_disp\_char\_at

am\_digitron\_disp\_str

当显示字符或字符串时，需要将各个字符解码为对应的段码后，数码管才能正常显示。为此需要提供一个设置解码函数的接口，便于用户根据实际数码管自定义解码函数，然后通过该接口设置到系统中。当需要显示一个字符时，系统首先会使用该解码函数将字符解码为段码。其对应的接口名为：

am\_digitron\_disp\_decode\_set

在一些应用场合，可能需要显示特殊的图形，此时仅仅有显示字符或字符串的接口是不够的，还需要提供一个直接显示段码的接口，其对应的接口为：

am\_digitron\_disp\_at

此外，作为一个显示器，还需要清除当前数码管显示的所有内容，便于重新设置显示内容，其对应的接口名为：

am\_digitron\_disp\_clr

特别地，除设置显示内容相关的操作外，还需要数码管闪烁显示，其对应的接口名为：

am\_digitron\_disp\_blink\_set

1. 接口参数

通常系统存在多个数码管，比如，同时使用MiniPort-View和MiniPort-ZLG72128。在一个数码管设备中，又可能包含多个数码管，比如，MiniPort-View和MiniPort-ZLG72128均包含两个数码管。

为了区分不同的数码管设备，需要为每个数码管设备分配一个唯一的ID，基于此，将所有接口的第一个参数设定为数码管的ID，用于指定需要操作的数码管设备。

am\_digitron\_disp\_char\_at()接口用于显示一个字符，虽然有数码管设备ID用于确定显示该字符的数码管设备，但仅仅通过数码管设备ID还不能确定在数码管设备中显示的具体位置，为此需要新增一个索引参数，用于指定字符显示的位置，索引的有效范围为0~（数码管个数-1），如MiniPort-View有两个数码管，则索引的有效范围为0~1。此外，该接口还需要一个参数用以指定要显示的字符。定义该接口的函数原型为（暂未定义返回值类型）：

am\_digitron\_disp\_char\_at(int id, int index, const char ch);

对于am\_digitron\_disp\_str()接口，其用于显示一个字符串，除数码管设备ID外，同样需要一个索引参数以指定字符串显示的起始位置，此外，还需要使用参数指定要显示的字符串以及显示的字符串的长度。定义该接口的函数原型为（暂未定义返回值类型）：

int am\_digitron\_disp\_str(int id, int index, int len, const char \*p\_str);

其中，len指定显示的长度，p\_str指定要显示的字符串，实际显示的长度为字符串长度和len中的较小值。

对于am\_digitron\_disp\_decode\_set()接口，其用于设定字符的解码函数，显然，一个数码管设备中的多个数码管往往是相同的，可以使用同样的解码规则，共用一个解码函数。因而接口仅需要使用ID指定数码管设备，无需使用index指定具体的数码管索引。解码函数的作用是对字符进行解码，输入一个字符，输出该字符对应的编码。基于此，定义该接口的函数原型为（暂未定义返回值类型）：

am\_digitron\_disp\_decode\_set(int id, uint16\_t(\*pfn\_decode)(uint16\_t ch));

其中，pfn\_decode是指向解码函数的指针，表明了解码函数的类型：具有一个16位无符号类型的ch参数，返回值为16位无符号类型的编码。这里使用16位的数据表示字符和编码，是为了更好的扩展性。如除8段数码管外，还存在14段的米字符数码管、16段数码管等，这些情况下，8位数据就无法表示完整的段码了。

对于am\_digitron\_disp\_at()接口，其用于直接设置显示的段码，和显示一个字符类似，除数码管设备ID外，同样需要使用参数指定显示的位置以及要显示的内容（段码），可定义该接口的原型为（暂未定义返回值类型）：

am\_digitron\_disp\_at(int id, int index, uint16\_t seg);

对于am\_digitron\_disp\_clr()接口，其用于清除一个数码管设备显示的所有内容，仅需使用ID指定需要清除的数码管设备，无需其它额外参数。定义该接口的原型为（暂未定义返回值类型）：

am\_digitron\_disp\_clr(int id);

对于am\_digitron\_disp\_blink\_set()接口，其用于设置数码管的闪烁属性，除数码管ID外，还需要使用参数指定设置闪烁属性的数码管位置以及使用本次设置的闪烁属性（打开闪烁还是关闭闪烁），定义该接口的函数原型为（暂未定义返回值类型）：

am\_digitron\_disp\_blink\_set(int id, int index, am\_bool\_t blink);

其中，index指定本次设置闪烁属性的数码管位置，blink指定闪烁属性，当值为AM\_TRUE时，则打开闪烁；当值为AM\_FALSE时，则关闭闪烁。

#### 返回值

接口无特殊说明，直接将接口的返回值定义为int类型的标准错误号。数码管接口的完整定义详见表3.2。

表3.2 数码管通用接口（am\_digitron\_disp.h）

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_digitron\_disp\_decode\_set(  int id,  uint16\_t(\*pfn\_decode)(uint16\_t ch)); | 设置段码解码函数； |
| int am\_digitron\_disp\_blink\_set(int id, int index, am\_bool\_t blink); | 设置数码管闪烁； |
| int am\_digitron\_disp\_at(int id, int index, uint16\_t seg); | 显示指定的段码图形； |
| int am\_digitron\_disp\_char\_at(int id, int index, const char ch); | 显示字符； |
| int am\_digitron\_disp\_str(int id, int index, int len, const char \*p\_str); | 显示字符串； |
| int am\_digitron\_disp\_clr(int id); | 显示清屏； |

### 实现接口

#### 抽象的数码管设备类

类似地，应该根据通用数码管接口，定义相应的抽象方法，以显示字符函数为例，按照前面的通用做法，其抽象方法定义为：

int(\*pfn\_disp\_char\_at)(void \*p\_cookie, int id, int index, const char ch);

相比于通用接口，其新增了一个用于指向设备自身的p\_cookie参数。在定义数码管通用接口时，使用ID唯一的代表了一个数码管设备，可见，在这里，p\_cookie和id均代指了某一确定的数码管设备，由于抽象方法是由具体数码管设备实现的，p\_cookie也用于指向设备自身，通过p\_cookie已经能够唯一的确定某一具体设备，因此，ID参数在抽象方法中再无实际用处，将ID参数从抽象方法中移除，即：

int (\*pfn\_disp\_char\_at)(void \*p\_cookie, int index, const char ch);

实际上，可以看作是参数由抽象意义的ID（仅是一个数字）变为了具有实际意义的p\_cookie（指向设备自身的指针）。

基于此，根据其它通用数码管通用接口，为它们一一定义相应的抽象方法，并将其存放在一个虚函数表中，即：

typedef struct am\_digitron\_disp\_ops{

int(\*pfn\_decode\_set)(void \*p\_cookie, uint16\_t(\*pfn\_decode)(uint16\_t ch));

int(\*pfn\_blink\_set)(void \*p\_cookie, int index, am\_bool\_t blink);

int(\*pfn\_disp\_at)(void \*p\_cookie, int index, uint16\_t seg);

int(\*pfn\_disp\_char\_at)(void \*p\_cookie, int index, const char ch);

int(\*pfn\_disp\_str)(void \*p\_cookie, int index, int len, const char \*p\_str);

int(\*pfn\_clr)(void \*p\_cookie);

}am\_digitron\_disp\_ops\_t;

读者可能会发现，在实现LED接口时，定义的抽象方法同时包含了p\_cookie和led\_id参数。即：

typedef struct am\_led\_drv\_funcs{

int(\*pfn\_led\_set)(void \*p\_cookie, int led\_id, am\_bool\_t state);

int(\*pfn\_led\_toggle)(void \*p\_cookie, int led\_id);

}am\_led\_drv\_funcs\_t;

这是由于在通用LED接口的设计中，ID并非是对LED设备进行的编号，而是对于系统中所有LED进行的编号，如AM116-Core板载了2个LED，MiniPort-LED上有8个LED，如果它们同时使用，则系统中有两个LED设备，但总共有10个LED，LED编号为0~9。因此，虽然p\_cookie能够确定要操作的LED设备，但还是不能确定要操作的具体LED，因此，必须将LED的编号作为参数传递给具体方法，以便准确的操作到某一个具体的LED。

在通用数码管接口的设计中，ID是对数码管设备的编号，如同时使用MiniPort-View和MiniPort-ZLG72128时，系统中有两个数码管设备，虽总共有4个数码管，但数码管设备的编号只会是0~1。因为如此，数码管设备ID中并不包含具体数码管的位置信息，为了将显示内容显示到某一确定的数码管上，需要使用额外的index参数指定。

类似地，将抽象方法和p\_cookie定义在一起，即为抽象的数码管设备。比如：

typedef struct am\_digitron\_dev{

const am\_digitron\_disp\_ops\_t \*p\_ops; //驱动函数（抽象方法）

void \*p\_cookie; //驱动函数参数

}am\_digitrion\_dev\_t;

和LED抽象设备类似，实际上可能存在多个数码管设备，由于它们的具体数目是无法预先确定的，因此这里使用单向链表进行动态管理，在am\_digitron\_dev\_t中增加一个p\_next成员，用以指向下一个设备。即：

typedef struct am\_digitron\_dev{

const am\_digitron\_disp\_ops\_t \*p\_ops; //驱动函数（抽象方法）

void \*p\_cookie; //驱动函数参数

struct am\_digitron\_dev \*p\_next; //指向下一个数码管设备

}am\_digitron\_dev\_t;

此时，系统中的多个数码管设备使用链表的形式管理。由于在通用接口中，使用ID区分不同的数码管设备。因此，在通用接口的实现中，需要能够通过ID号找到对应的数码管设备，以便使用其中的抽象方法。和LED设备类似，可以将一个数码管设备和该设备对应的ID信息绑定在一起，就可以通过ID找到对应的数码管设备。

一个数码管设备对应一个唯一的ID，可以定义数码管设备ID信息的类型为：

typedef struct am\_digitron\_devinfo{

uint8\_t id; //数码管设备ID

}am\_digitron\_devinfo\_t;

在设备中新增指向ID信息的p\_info指针，便于在通用接口实现中根据ID查找到对应的数码管设备，即：

typedef struct am\_digitron\_dev{

const am\_digitron\_disp\_ops\_t \*p\_ops; //驱动函数（抽象方法）

void \*p\_cookie; //驱动函数参数

const am\_digitron\_devinfo\_t \*p\_info; //数码管设备ID信息

struct am\_digitron\_dev \*p\_next; //指向下一个数码管设备

}am\_digitron\_dev\_t;

基于此，am\_digitron\_disp\_char\_at()函数的实现详见程序清单3.3。

程序清单3.5 am\_digitron\_disp\_char\_at()接口实现范例

1 int am\_digitron\_disp\_char\_at(int id, int index, const char ch)

2 {

3 am\_digitron\_dev\_t \*p\_dd = \_digitron\_dev\_find\_with\_id(id);

4 if(p\_dd == NULL){

5 return –AM\_ENODEV;

6 }

7 if(p\_dd->p\_ops->pfn\_disp\_char\_at){

8 return p\_dd->p\_ops->pfn\_disp\_char\_at(p\_dd->p\_cookie, index, ch);

9 }

10 return –AM\_ENOTSUP;

11 }

其中，\_\_digitron\_dev\_find\_with\_id()的作用就是遍历设备链表，与各个设备中的ID信息一一比对，以找到数码管ID对应的数码管设备，其实现详见程序清单3.4。

程序清单3.6 查找指定ID的数码管设备

1 static am\_digitron\_dev\_t \*\_\_gp\_head;

2

3 static am\_digitron\_dev\_t \*\_\_digitron\_dev\_find\_with\_id(int id)

4 {

5 am\_digitron\_dev\_t \*p\_cur = \_\_gp\_head;

6 while(p\_cur != NULL){

7 if(p\_cur->p\_info->id == id){

8 break; //找到该ID对应的设备

9 }

10 p\_cur = p\_cur->p\_next;

11 }

12 return p\_cur; //返回找到的设备，若未找到，则为NULL

13 }

其中，\_\_gp\_head是一个全局变量，指向数码管设备的链表头，初始为NULL，表示初始时系统中无任何数码管设备。同理可得到其它接口的实现，详见程序清单3.5，它们的实现都非常类似，均为首先通过\_\_digitron\_dev\_find\_with\_id()函数找到ID对应的数码管设备，然后直接调用设备中的抽象方法。

程序清单3.7 其它数码管接口的实现范例

1 int am\_digitron\_disp\_decode\_set(int id, uint16\_t(\*pfn\_decode)(uint16\_t ch))

2 {

3 am\_digitron\_dev\_t \*p\_dd = \_\_digitron\_dev\_find\_with\_id(id);

4 if(p\_dd == NULL){

5 return –AM\_ENODEV;

6 }

7 if(p\_dd->p\_ops->pfn->decode\_set){

8 return p\_dd->p\_ops->pfn\_decode\_set(p\_dd->p\_cookie, pfn->decode);

9 }

10 return –AM\_ENOTSUP;

11 }

12

13 int am\_digitron\_disp\_blinkset(int id, int index, am\_bool\_t blink)

14 {

15 am\_digitron\_dev\_t \*p\_dd = \_\_digitron\_dev\_find\_with\_id(id);

16 if(p\_dd == NULL){

17 return –AM\_ENODEV;

18 }

19 if(p\_dd->p\_ops->pfn\_blink\_set){

20 return p\_dd->p\_ops->pfn\_blink\_set(p\_dd->p\_cookie, index, blink);

21 }

22 return –AM\_ENOTSUP;

23 }

24

25 int am\_digitron\_disp\_at(int id, int index, uint16\_t seg)

26 {

27 am\_digitron\_dev\_t \*p\_dd = \_\_digitron\_dev\_find\_with\_id(id);

28 if(p\_dd == NULL){

29 return –AM\_ENODEV;

30 }

31 if(p\_dd->p\_ops->pfn\_disp\_at){

32 return p\_dd->p\_ops->pfn\_disp\_at(p\_dd->p\_cookie, index, seg);

33 }

34 return –AM\_ENOTSUP;

35 }

36

37 int am\_digitron\_disp\_str(int id, int index, int len, const char \*p\_str)

38 {

39 am\_digitron\_dev\_t \*p\_dd = \_\_digitron\_dev\_find\_with\_id(id);

40 if(p\_dd == NULL){

41 return –AM\_ENODEV;

42 }

43 if(p\_dd->p\_ops->pfn\_disp\_str){

44 return p\_dd->p\_ops->pfn\_disp\_str(p\_dd->p\_cookie, index, len, p\_str);

45 }

46 return –AM\_ENOTSUP;

47 }

48

49 int am\_digitron\_disp\_clr(int id)

50 {

51 am\_digitron\_dev\_t \*p\_dd = \_\_digitron\_dev\_find\_with\_id(id);

52 if(p\_dd == NULL){

53 return –AM\_ENODEV;

54 }

55 if(p\_dd->p\_ops->pfn\_clr){

56 return p\_dd->p\_ops->pfn\_clr(p\_dd->p\_cookie);

57 }

58 return –AM\_ENOTSUP;

59 }

由于当前没有任何数码管设备，因此\_\_digitron\_dev\_find\_with\_id()的返回值始终为NULL，使得通用接口的返回值始终为-AM\_ENODEV（错误：无此设备）。

为了使通用接口能够操作到具体有效的数码管设备，就必须在使用通用数码管接口前，向系统中添加有效的数码管设备。根据数码管设备类型的定义，添加一个设备时，需要完成p\_ops、p\_cookie、和p\_info的正确赋值，这些成员的值是由具体数码管设备实现或定义的，为此，可以为具体数码管的设备驱动提供一个添加数码管设备的接口，定义其函数原型为：

int am\_digitron\_dev\_add(

am\_digitron\_dev\_t \*pdd,

const am\_digitron\_devinfo\_t \*p\_info,

const am\_digitron\_disp\_ops\_t \*p\_ops,

void \*p\_cookie) ;

其中，为了方便直接添加一个设备，避免直接操作数码管设备的各个成员，将需要赋值的成员通过参数传递给接口函数。其实现详见程序清单3.6。

程序清单3.8 向系统中添加数码管设备

1 int am\_digitron\_dev\_add(

2 am\_digitron\_dev\_t \*pdd,

3 const am\_digitron\_devinfo\_t \*p\_info,

4 const am\_digitron\_disp\_ops\_t \*p\_ops,

5 void \*p\_cookie)

6 {

7 if((p\_dd == NULL)) || (p\_ops == NULL)||(p\_info == NULL)){

8 return –AM\_EINVAL;

9 }

10 if(\_\_digitron\_dev\_find\_with\_id(p\_info->id) != NULL){

11 return –AM\_EPERM;

12 }

13 p\_dd->p\_info = p\_info;

14 p\_dd->p\_ops = p\_ops;

15 p\_dd->p\_next = NULL;

16 p\_dd->p\_cookie = p\_cookie;

17 p\_dd->p\_next = \_\_gp\_head;

18 \_\_gp\_head = pdd;

19 return AM\_OK;

20 }

首先检查了各个参数的有效性，然后使用\_\_digitron\_dev\_find\_with\_id()函数判断新设备的ID号是否已经在系统中，若系统中已经存在该ID，则添加失败，直接返回操作不允许错误（-AM\_EPERM）。若系统中不存在该ID，则继续执行，以确保添加的各个数码管设备的ID不冲突，保证了数码管设备编号的唯一性，接着将设备中的各个成员赋值，最后通过程序清单3.6的17~18这2行代码将新设备添加到链表首部。

显然，接下来需要的具体的数码管设备中，实现相应的抽象方法，然后使用am\_digitron\_dev\_add()接口将设备添加到系统中，使得用户可以使用数码管通用接口操作到具体有效的数码管。为了便于查阅，如程序清单3.7所示展示了数码管设备接口文件（am\_digitron\_dev.h）的内容。

程序清单3.9 am\_digitron\_dev.hw文件内容

1 #pragma once

2 #include "ametal.h"

3

4 typedef struct am\_digitron\_disp\_ops{

5 int(\*pfn\_decode\_set)(void \*p\_cookie, uint16\_t (\*pfn\_decode)(uint16\_t ch));

6 int(\*pfn\_blink\_set)(void \*p\_cookie, int index, am\_bool\_t blink);

7 int(\*pfn\_disp\_at)(void \*p\_cookie, int index, uint16\_t seg);

8 int(\*pfn\_disp\_char\_at)(void \*p\_cookie, int index, const char ch);

9 int(\*pfn\_disp\_str)(void \*p\_cookie, int index, int len, const char \*p\_str);

10 int(\*pfn\_clr)(void \*p\_cookie);

11 }am\_digitron\_disp\_ops\_t;

12

13 typedef struct am\_digitron\_devinfo{

14 uint8\_t id;

15 }am\_digitron\_devinfo\_t;

16

17 typedef struct am\_digitron\_dev{

18 const am\_digitron\_disp\_ops\_t \*p\_ops; //驱动函数（抽象方法）

19 void \*p\_cookie; //驱动函数参数

20 const am\_digitron\_devinfo\_t \*p\_info; //数码管设备ID信息

21 struct am\_digitron\_dev \*p\_next; //指向下一个数码管设备

22 }am\_digitron\_dev\_t;

23

24 int am\_digitron\_dev\_add(

25 am\_digitron\_dev\_t \*p\_dd,

26 const am\_digitron\_devinfo\_t \*p\_info,

27 const am\_digitron\_disp\_ops\_t \*p\_ops,

28 void \*p\_cookie);

#### 具体的数码管设备类

以使用GPIO的MiniPort-View为例，简述具体数码管设备的实现方法。首先应该基于抽象设备类派生一个具体的设备类，定义具体的数码管设备类如下：

typedef struct am\_digitron\_miniport\_view{

am\_digitron\_dev\_t isa; //派生至抽象的数码管设备

}am\_digitron\_miniport\_view\_t;

am\_digitron\_miniport\_view\_t即为具体的数码管设备类。具有该类型后，即可使用该类型定义一个具体的数码管设备实例，即：

am\_digitron\_scan\_gpio\_dev\_t g\_digitron\_miniport\_view;

对于动态扫描类的数码管，需要将数码管显示的段码缓存到一段内存中，然后定时扫描，依次扫描各个数码管，从缓存中取出当前扫描数码管的段码，然后将段码输送到相应的引脚上显示。

为了实现数码管定时自动扫描，需要使用到软件定时器，可以新增一个软件定时器timer成员；在扫描过程中，需要实时记录当前的扫描位置，以便从相应的数码管缓存中取出对应的段码，一个数码管扫描结束后，扫描位置要更新为下一个数码管的位置，可以增加scan\_idx成员来实时存储当前数码管的扫描位置。即设备类型可定义为：

typedef struct am\_digitron\_miniport\_view{

am\_digitron\_dev\_t isa; //派生至抽象的数码管设备

am\_softtimer\_t timer; //使用软件定时器以自动扫描

uint8\_t scan\_idx; //当前扫描索引

}am\_digitron\_miniport\_view\_t;

此外，为了保证闪烁属性，可以新增一个blink\_flags的成员表示各个需要数码管的闪烁属性，某一位的值为1时，表明对应的数码管需要闪烁。在一个闪烁周期中，一段时间需要点亮，一段时间需要熄灭，为了判定当前应该处于何种状态，可以新增一个闪烁计时器成员blink\_cnt，用于在一个闪烁周期内计时。特别地，在通用接口中，有一个设置数码管解码函数的接口，为了在用户显示字符时，能够使用其设置的解码函数对字符进行解码，则需要一个函数指针保存用户设置的解码函数，基于此，设备类型可定义为：

typedef struct am\_digitron\_miniport\_view{

am\_digitron\_dev\_t isa; //派生至抽象的数码管设备

am\_softtimer\_t timer; //使用软件定时器以自动扫描

uint8\_t scan\_idx; //当前扫描索引

uint32\_t blink\_flags; //闪烁属性标志

uint16\_t blink\_cnt; //闪烁计时器

uint16\_t (\*pfn\_decode)(uint16\_t code); //函数指针，指向用户设定的解码函数

}

此外，为了正常使用数码管，还需要知道一些硬件相关的基本信息，如：位选引脚信息、段码引脚信息、数码管个数，段码数目等，据此，可以定义数码管设备的信息类型为：

typedef struct am\_digitron\_miniport\_view\_info{

uint8\_t num\_digitron; //数码管个数

uint8\_t num\_segment; //数码管段码个数

am\_bool\_t seg\_active\_low; //数码管段端的极性

am\_bool\_t com\_active\_low; //数码管公共端的极性

const int \*p\_seg\_pins; //段码GPIO驱动引脚

const int \*p\_com\_pins; //位码GPIO驱动引脚

}am\_digitron\_miniport\_view\_info\_t;

同时，对于动态扫描类数码管，需要一个缓存用以存储显示的段码，缓存的大小应该与数码管个数相同，可以新增一个d\_disp\_buf指针以指向相应的缓存。此外，数码管动态扫描时，扫描的频率必须大于25Hz,才能使肉眼看不到动态扫描的过程，使整个数码管的显示完整、流畅。显然，频率越高，扫描越快，显示就越流畅，但扫描数码管时占用的CPU资源也就越多；频率越低，系统CPU资源占用也就越少，但也不能过低。为此，可以在设备信息中新增一个scan\_freq成员，用以指定扫描的频率，使得扫描频率可以由用户根据实际情况配置。扫描频率直接决定了定时器定时扫描的周期，若扫描频率为50Hz,则扫描一次数码管的时间为20ms，由于MiniPort-View存在两个数码管，因此定时器定时扫描的周期为10ms。可见，定时器定时扫描的时间间隔为：1000/scan\_freq/digitron\_num。

此外，当数码管需要闪烁时，为了更加个性化的定制闪烁的效果，可以使用blink\_on\_time和blink\_off\_time分别指定一个闪烁周期内点亮的时间和熄灭的时间，它们的时间之和即为闪烁周期，决定了闪烁的频率。

同时，在数码管通用接口中，各个数码管设备使用ID号进行区分，显然，这就要求为具体的数码管设备分配一个唯一ID，可以在设备信息中新增表示ID信息的成员dev\_info。完整的数码管设备信息的类型定义为：

typedef struct am\_digitron\_miniport\_view\_info{

am\_digitron\_devinfo\_t devinfo; //数码管设备的ID信息

uint8\_t scan\_freq; //整个数码管的扫描频率，一般50Hz

uint16\_t blink\_on\_time; //一个闪烁周期内，点亮的时间，比如500ms

uint16\_t blink\_off\_time; //一个闪烁周期内，熄灭的时间，比如500ms

uint8\_t \*p\_disp\_buf; //数码管显示缓存

uint8\_t num\_digitron; //数码管个数

uint8\_t num\_segment; //数码管段码个数

am\_bool\_t seg\_active\_low; //数码管段端的极性

am\_bool\_t com\_active\_low; //数码管公共端的极性

const int \*p\_seg\_pins; //段码GPIO驱动引脚

const int \*p\_com\_pins; //位码GPIO驱动引脚

}am\_digitron\_miniport\_view\_info\_t;

当将AM116-Core与数码管配板MiniPort-View联合使用时，若分配给数码管设备的ID为0，扫描频率为50Hz，在一个闪烁周期内，数码管点亮和熄灭的时间均为500ms，基于等效电路图以及数码管信息，定义与MiniPort-View对应的设备实例信息为：

uint8\_t g\_disp\_buf[2];

static const int g\_digitron\_seg\_pins[] = {

PIO0\_8,PIO0\_9, PIO0\_10,PIO0\_11, PIO0\_12,PIO0\_13, PIO0\_14,PIO0\_15

};

static const int g\_digitron\_com\_pins[] = {

PIO0\_17,PIO0\_23

}

am\_digitron\_miniport\_view\_t g\_miniport\_view;

const am\_digitron\_miniport\_view\_info\_t g\_miniport\_view\_info = {

{

0, //ID为0

}

50, //扫描频率50Hz

500, //一次闪烁中，点亮时间为500ms

500, //一次闪烁中，熄灭时间为500ms

g\_disp\_buf, //显示缓存

2, //数码管个数

8, //8段数码管

AM\_TURE, //数码管段码引脚低电平有效

AM\_TURE, //数码管位选引脚低电平有效

g\_digitron\_seg\_pins, //段码引脚

g\_digitron\_com\_pins //位选引脚

};

类似地，在数码管的设备类型中需要维持一个指向数码管设备信息的指针，以便在任何时候都可以从数码管设备中取出相关的信息使用，完整的数码管设备类型定义为：

typedef struct am\_digitron\_miniport\_view{

am\_digitron\_dev\_t isa; //派生至抽象的数码管设备

am\_softtimer\_t timer; //使用软件定时器以自动扫描

uint8\_t scan\_idx; //当前扫描索引

uint32\_t blink\_flags; //闪烁属性标志

uint16\_t blink\_cnt; //闪烁计时器

uint16\_t (\*pfn\_decode)(uint16\_t code); //函数指针，指向用户设定的解码函数

const am\_digitron\_miniport\_view\_info\_t \*p\_info; //指向数码管设备信息的指针

}am\_digitron\_miniport\_view\_t;

实际开发过程中，通常并不能一次性完整的定义出设备或设备信息的结构体类型，往往是在定义好基本结构后，在后续实现各个抽象方法的过程中，根据需要增加成员，不断完善结构体类型的定义。

为了正常扫描数码管，需要完成设备中各成员的赋值，在完成初始赋值后，则可以启动软件定时器，进而以设备信息中指定的扫描频率自动扫描数码管。这些工作通常在驱动的初始化函数中完成，定义初始化函数的原型为：

int am\_digitron\_miniport\_view\_init(

am\_digitron\_miniport\_view\_t \*p\_dev,

const am\_digitron\_miniport\_view\_info\_t \*p\_info);

其中，p\_dev为指向am\_digitron\_miniport\_view\_t类型实例的指针，p\_info为指向am\_digitron\_miniport\_view\_info\_t类型实例信息的指针，其调用形式如下：

am\_digitron\_miniport\_view\_init(&g\_digitron\_miniport\_view,&g\_digitron\_miniport\_view\_info);

初始化完成后，可使用通用数码管接口操作编号0的数码管设备，初始化函数的实现范例详见程序清单3.8。

程序清单3.10 初始化函数实现范例

1 int am\_digitron\_miniport\_view\_init(

2 am\_digitron\_miniport\_view\_t \*p\_dev,

3 const am\_digitron\_miniport\_view\_info\_t \*p\_info)

4 {

5 int i;

6 if((p\_dev == NULL)||(p\_info == NULL)){

7 retun –AM\_EINVAL

8 }

9 p\_dev->info = p\_info; //将设备信息保存到设备中

10 p\_dev->pfn\_decode = NULL; //解码函数，初始为NULL

11 p\_dev->blink\_flages = 0x00; //闪烁标识，初始为0，所有数码管不闪烁

12 p\_dev->blink\_cnt = 0; //闪烁计数值，初始为0

13 p\_dev->scan\_idx = 0; //扫描索引，初始为0

14 if(p\_info->com\_active\_low){

15 for(i=0;i<p\_info->num\_digitron;i++){

16 am\_gpio\_pin\_cfg(p\_info->p\_com\_pins[i],AM\_GPIO\_OUTPUT\_INIT\_HIGH);

17 }

18 }else{

19 for(i = 0; i<p\_info->num\_digitron;i++){

20 am\_gpio\_pin\_cfg(p\_info->p\_com\_pins[i],AM\_GPIO\_OUTPUT\_INIT\_LOW);

21 }

22 }

23 if(p\_info->seg\_active\_low){ //根据激活电平，初始段码引脚

24 for(i = 0; i<p\_info->num\_segment;i++){

25 am\_gpio\_pin\_cfg(p\_info->p\_seg\_pins[i],AM\_GPIO\_OUTPUT\_INIT\_HIGH);

26 }

27 }else{

28 for(i = 0; i<p\_info->num\_segment; i++){

29 am\_gpio\_pin\_cfg(p\_info->p\_seg\_pins[i],AM\_GPIO\_OUTPUT\_INIT\_LOW);

30 }

31 }

32 am\_softtimer\_init(&p\_dev->timer,\_\_digitron\_dynamic\_scan\_timer\_cb,p\_dev);

33 am\_softtimer\_init(&p\_dev->timer,1000/p\_info->scan\_freq/p\_info->num\_digitron);

34 //添加标准的数码管设备

35 return am\_digitron\_dev\_add(&p\_dev->isa,&p\_info->devinfo,&\_\_g\_digitron\_dev\_ops,p\_dev);

36 }

该程序首先检查了参数的有效性，然后完成了设备中各个成员的初始赋值，接着根据位选引脚和段码引脚的激活电平，将位选引脚和段码引脚配置成输出模式，并将初始电平设置为未激活电平，以使数码管初始处于完全熄灭状态。

紧接着初始化并启动了软件定时器，根据扫描频率设定了软件定时器的定时周期，必将软件定时器的周期性回调函数设置了\_\_digitron\_dynamic\_scan\_timer\_cb()，即为该函数中完成数码管的扫描。最后，使用am\_digitron\_dev\_add()接口将设备添加到了系统中，并将数码管ID号信息作为该接口p\_info的实参，&\_\_g\_digitron\_dev\_ops作为该接口p\_ops的实参，指向自身的指针p\_dev作为了接口p\_cookie的实参，\_\_g\_digitron\_dev\_ops中即包含了各个抽象方法的实现。

由此可见，实现整个具体数码管设备的关键，是在\_\_digitron\_dynamic\_scan\_timer\_cb()中完成数码管的扫描，以及实现各个抽象方法并存于\_\_g\_digitron\_dev\_ops中。

定时器回调函数\_\_digitron\_dynamic\_scan\_timer\_cb()的实现详见程序清单3.9。

程序清单3.11 定时器回调函数的实现（数码管扫描）

1 static void \_\_digitron\_dynamic\_scan\_timer\_cb(void \*p\_arg)

2 {

3 am\_digitron\_miniport\_view\_t \*p\_dev = (am\_digitron\_miniport\_view\_t \*)pag;

4

5 if(p\_dev->blink\_flages == 0){

6 p\_dev->blink\_cnt = 0;

7 }else{

8 p\_dev->blink\_cnt += 1000/p\_dev->p\_info->scan\_freq/p\_dev->p\_info->num\_digitron;

9 if(p\_dev->blink\_cnt>=p\_dev->p\_info->blink\_on\_time+p\_dev->p\_info->blink\_off\_time){

10 p\_dev->blink\_cnt = 0;

11 }

12 }

13 \_\_scan\_seg\_send(p\_dev,p\_dev->p\_info->seg\_active\_low?0xff:0x00);

14 \_\_scan\_com\_sel(p\_dev,p\_dev->scan\_idx);

15 if((!(p\_dev->blink\_flags&(1<<p\_dev->scan\_idx))) ||

16 (p\_dev->blink\_cnt <= p\_dev->p\_info->blink\_on\_time)){

17 \_\_scan\_seg\_send(p\_dev,p\_dev->p\_info->p\_disp\_buf[p\_dev->scan\_idx]);

18 }

19 p\_dev->scan\_idx = (p\_dev->scan\_idx + 1)%p\_dev->p\_info->num\_digitron;

20 }

该程序首先处理闪烁计时器，若存在闪烁的数码管，则增加闪烁计时器p\_dev->blink\_cnt,增加的值即为扫描时间间隔。特别地，若值增加后超过了闪烁周期，则重新回到0。然后使用\_\_scan\_seg\_send()发送消影段码，使用\_\_scan\_com\_sel()处理位选。若当前数码管需要正常显示，即当前数码管不需要闪烁，或者虽然需要闪烁，但根据闪烁计时器判定当前时间处在点亮数码管的时间周期，则从显示缓存中取出当前数码管的段码，并使用\_\_scan\_seg\_send()发送出去。最后更新了扫描位置索引scan\_idx的值，以便下一次扫描时继续扫描下一个数码管，段码发送函数和位选函数的实现详见程序清单3.10。

程序清单3.12 段码发送函数和位选函数的实现

1 static void \_\_scan\_seg\_send(am\_digitron\_miniport\_view\_t \*p\_dev, uint16\_t seg)

2 {

3 const int \*p\_pins = p\_dev->p\_info->p\_seg\_pins;

4 int bit;

5 for(bit = 0;bit < p\_dev->p\_info->num\_segment;bit++){

6 if(seg&(1<<bit)){

7 am\_gpio\_set(p\_pins[bit],1);

8 }else{

9 am\_gpio\_set(p\_pins[bit],0);

10 }

11 }

12 }

13 static void \_\_scan\_com\_sel(am\_digitron\_miniport\_view\_t \*p\_dev, int idx)

14 {

15 const int \*p\_pins = p\_dev->p\_info->p\_com\_pins;

16 int i;

17 for(i = 0; i < p\_dev->p\_info->num\_digitron;i++){

18 if(i == idx){

19 am\_gpio\_set(p\_pins[i],!p\_dev->p\_info->com\_active\_low);

20 }else{

21 am\_gpio\_set(p\_pins[i],p\_dev->p\_info->com\_active\_low);

22 }

23 }

24 }

接下来，需要一一实现抽象数码管设备中共计定义了6个抽象方法，以完成\_\_g\_digitron\_dev\_ops的定义。

* pfn\_decode\_set

该方法用于设定解码函数，便于显示时，对各个字符进行解码。显然，需要将其保存到设备中，以便后续使用。范例程序详见程序清单3.11。

程序清单3.13 设置解码函数的实现范例程序

1 static int \_\_digitron\_decode\_set(void \*p\_cookie, int16\_t (\*pfn\_decode)(uint16\_t ch))

2 {

3 am\_digitron\_miniport\_view\_t \*p\_dev = (am\_digitron\_miniport\_view\_t \*)p\_cookie;

4 if((p\_dev == NULL) || (pfn\_decode == NULL)){

5 return –AM\_EINVAL;

6 }

7 p\_dev->pfn\_decode = pfn\_decode; //将解码函数保存到设备中

8 return AM\_OK;

9 }

* pfn\_blink\_set

该方法用于设定某一个数码管的闪烁属性，设置闪烁属性时，只需要将设备中的闪烁标记blink\_flags相应位置1（闪烁）或清零（不闪烁）即可，范例程序详见程序清单3.12。

程序清单3.14 设置闪烁属性函数的实现范例程序

1 static int \_\_digitron\_blink\_set(void \*p\_cookie, int index, am\_bool\_t blink)

2 {

3 am\_digitron\_miniport\_view\_t \*p\_dev = (am\_digitron\_miniport\_view\_t \*)p\_cookie;

4 if((p\_dev == null) || (index >= p\_dev->p\_info->num\_digitron)){

5 return –am\_einval;

6 }

7 if(blink){

8 p\_dev->blink\_flags |= (1 << index);

9 }else{

10 p\_dev->blink\_flags &= ~(1 << index);

11 }

12 return am\_ok;

13 }

* pfn\_disp\_at

该方法用于指定数码管上显示指定的段码图形，只需要将段码存放在数码管缓存中即可，范例程序详见程序清单3.13。

程序清单3.15 显示段码函数的实现范例程序

1 static int \_\_digitron\_disp\_at(void \*p\_cookie, int index, uint16\_t seg)

2 {

3 am\_digitron\_miniport\_view\_t \*p\_dev = (am\_digitron\_miniport\_view\_t \*)p\_cookie;

4 if((p\_dev == null) || (index >= p\_dev->p\_info->num\_digitron)){

5 return –am\_einval;

6 }

7 \_\_digitron\_disp\_buf\_set(p\_dev, index, seg);

8 return am\_ok;

9 }

该程序调用了\_\_digitron\_disp\_buf\_set()函数将段码设置到缓存中，详见程序清单3.14。

程序清单3.16 \_\_digitron\_disp\_buf\_set()函数实现

1 static void \_\_digitron\_disp\_buf\_set(am\_digitron\_miniport\_view\_t \*p\_dev, int index, uint16\_t data)

2 {

3 if(p\_dev->p\_info->seg\_active\_low){

4 p\_dev->p\_info->p\_disp\_buf[index] = ~data;

5 }else{

6 p\_dev->p\_info->p\_disp\_buf[index] = data;

7 }

8 }

该程序根据数码管段的激活电平决定是否需要将用户设置的段码取反后存入缓冲区中。

* pfn\_disp\_char\_at

该方法用于在指定位置显示字符，这就需要先使用解码函数得到字符对应的段码，然后将段码设置到缓存区中，范例程序详见程序清单3.15。

程序清单3.17 显示字符函数的实现范例程序

1 static int \_\_digitron\_disp\_char\_at(void \*p\_cookie, int index, const char ch)

2 {

3 am\_digitron\_miniport\_view\_t \*p\_dev = (am\_digitron\_miniport\_view\_t\*)p\_cookie;

4 uint16\_t seg = 0x00;

5 if((p\_dev == NULL) || (index >= p\_dev->p\_info->num\_digitron)) {

6 return –AM\_EINVAL;

7 }

8 if(p\_dev->pfn\_decode){

9 seg = p\_dev->pfn\_decode(ch);

10 }

11 if( '.' == ch ){

12 \_\_digitron\_disp\_buf\_xor(p\_dev, index, seg);

13 }else{

14 \_\_digitron\_disp\_buf\_set(p\_dev, index, seg);

15 }

16 return AM\_OK;

17 }

该程序首先通过解码函数得到字符的段码存于seg中。然后将段码保存到相应的缓冲区中，且在保存段码时对小数点作了特殊的处理。

当字符为小数点时，则使用\_\_digitron\_disp\_buf\_xor()函数将段码设置到函数区；否则直接使用\_\_digitron\_disp\_buf\_set()函数将段码设置到缓存区中。

由于小数点比较特殊，因此显示小数点时，往往不希望影响该位数码管的正常显示内容，比如，如果当前数码管显示数字3，又需要在该数码管添加小数点，则期望的结果是显示“3.”，而不仅仅显示一个小数点，将之前的3覆盖掉。由此可见，在显示小数点时，可视为显示内容的一种叠加，而不是直接改变显示内容，\_\_digitron\_disp\_buf\_xor()函数的实现详见程序清单3.16。

程序清单3.18 \_\_digitron\_disp\_buf\_xor()函数实现

1 static void \_\_digitron\_disp\_buf\_xor(am\_digitron\_miniport\_view\_t \*p\_dev, int index, uint16\_t data)

2 {

3 if(p\_dev->p\_info->seg\_active\_low){

4 p\_dev->p\_info->p\_disp\_buf[index] &= ~data;

5 } else {

6 p\_dev->p\_info->p\_disp\_buf[index] |= data;

7 }

8 }

若数码管段为低电平激活，则需要在原缓冲区段码的基础上，将显示小数点需要点亮的段清零，以显示小数点；反之则将显示小数点需要点亮的段置1，以显示小数点。

* pfn\_disp\_str

该方法用于从指定位置开始显示一个字符串，范例程序详见程序清单3.17。

程序清单3.19 字符串显示函数实现范例程序

1 static int \_\_digitron\_disp\_str(void \*p\_cookie, int index, int len, const char \*p\_str)

2 {

3 am\_digitron\_miniport\_view\_t \*p\_dev = (am\_digitron\_miniport\_view\_t \*)p\_cookie;

4 if((p\_dev == NULL) || (index >= p\_dev->p\_info->num\_digitron) || (p\_str == NULL) ){

5 return –AM\_EINVAL;

6 }

7 uint16\_t last\_ch = 0,ch;

8 int idx = index – 1;

9 uint8\_t str\_len = strlen(p\_str);

10 //字符串长度取字符串实际长度和参数len中的较小值

11 if(len>str\_len){

12 len = str\_len;

13 }

14 while((len)&&((ch = \*p\_str++)!= '\0')){

15 if(('.'!=ch)||( '.' == last\_ch)||(idx<0)){

16 ++idx;

17 }

18 if(idx>=p\_dev->p\_info->num\_digitron){

19 return AM\_OK;

20 }

21 \_\_digitron\_disp\_char\_at(p\_dev, idx, ch);

22 len--;

23 last\_ch = ch ;

24 }

25 return AM\_OK;

26 }

该程序首先确定了字符串的长度，字符串长度取字符串实际长度和len中的较小值，然后再while循环中，调用了\_\_digitron\_disp\_char\_at()函数依次显示单个字符。

其中的idx用于指定显示位置，初始值为index-1，即字符串显示起始位置的上一个数码管位置，若起始位置为0，则idx的初始值表示一个无效的位置。每次显示新内容前，需要更新idx的值（将idx加1）。但在某种特殊情况下，小数点不需要更新显示位置，例如，显示字符串“3.5”，期望显示的效果是只占用2个数码管，分别显示“3.”和“5.”，而不是占用3个数码管，这种情况下，当显示小数点时，直接显示在“3”所在的数码管中即可，无需将其单独显示到一个数码管上。程序需要更新显示位置的条件为：

if(('.' != ch) || ('.' == last\_ch ) || (idx < 0)) {

++idx;

}

由此可见，不需要更新位置的条件即为上述条件的反面：

if(('.' != ch) && ('.'!= last\_ch) && (idx >= 0)) {

//不需要更新位置

}

不需要更新位置的条件为：当前显示的字符为小数点，且上一个字符不为小数点，同时idx指定的显示位置有效。

* pfn\_clr

该方法用于清空数码管显示，需要将缓冲区中的内容全部设置为熄灭段码，范例程序详见程序清单3.18。

程序清单3.20 清空显示内容函数的实现范例程序

1 static int \_\_digitron\_clr(void \*p\_cookie)

2 {

3 am\_digitron\_miniport\_view\_t \*p\_dev = (am\_digitron\_miniport\_view\_t \*)p\_cookie;

4 int i

5 for(i = 0; i < (p\_dev->p\_info->num\_digitron);i++){

6 \_\_digitron\_disp\_buf\_set(p\_dev,i,0x00);

7 }

8 return am\_ok;

9 }

至此，实现了各个抽象方法，基于各个抽象方法的实现函数，\_\_g\_digitron\_dev\_ops的定义详见程序清单3.19。

程序清单3.21 \_\_g\_digitron\_dev\_ops的定义

1 static const am\_digitron\_disp\_ops\_t \_\_g\_digitron\_dev\_ops = {

2 \_\_digitron\_decode\_set,

3 \_\_digitron\_blink\_set,

4 \_\_digitron\_disp\_at,

5 \_\_digitron\_disp\_char\_at,

6 \_\_digitron\_disp\_str,

7 \_\_digitron\_clr,

8 };

当用户使用初始化函数完成一个具体数码管设备的初始化后，即可使用通用数码管接口操作数码管，显示具体内容。但是，在显示数字或字符前，必须使用通用接口设置一个解码函数。对于8段数码管，可以将各个ASCII字符的段码定义在一个数组中，然后实现一个解码函数，详见程序清单3.20。

程序清单3.22 解码函数的实现

1 static const uint8\_t segcodeTab[] = {

2 0x40,0x00,0x00,0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D, //45~54

3 0x07,0x7F,0x6F,0x00,0x00,0x00,0x00,0x00,0x00,0x00, //55~64

4 0x77,0x7C,0x39,0x5E,0x79,0x71, 0x00,0x00,0x00,0x00, //65~74

5 0x00,0x00,0x00,0x37,0x3F,0x73,0x00,0x50,0x00,0x00, //75~84

6 0x00,0x00,0x00, 0x00,0x00,0x00, 0x00,0x00,0x00,0x00, //85~94

7 0x00,0x00,0x77,0x7C,0x39,0x5E,0x79, 0x71,0x00,0x00, //95~104

8 0x00,0x00,0x00,0x00,0x00,0x37,0x3F,0x73,0x00,0x50, //105~114

9 };

10 uint16\_t am\_digitron\_seg8\_ascii\_decode(uint16\_t ch)

11 {

12 if(ch >= '-' && (ch < '-'+sizeof(segcodetab))){ //编码表从‘-‘开始

13 return segcodetab[ch – '-'];

14 }

15 return 0x00;

16 }

基于此，在用户使用数码管接口显示字符或字符串前，可以使用设置解码函数的接口将该解码函数设置到系统中，以便正确解码。比如：

am\_digitron\_disp\_decode\_set(0,am\_digitron\_seg8\_ascii\_decode);

如果用户每次使用数码管前，都需要自定义一个解码函数，则显得非常麻烦。对于8段数码管，常见图形的显示方法是固定的，对应的段码是可以确定的，如数字0~9。用户如果没有特殊需求，使用程序清单3.20所示的解码函数是能满足绝大部分应用的。基于此，可以将程序清单3.20所示的解码函数定义在系统中，直接供用户使用。为方便用户使用，可以将该解码函数声明到数码管接口文件中。

为了便于查阅，如程序清单3.21所示展示了具体数码管设备（MiniPort\_View）接口文件（am\_digitron\_miniport\_view.h）的内容。

程序清单3.23 am\_digitron\_miniport\_view.h文件内容

1 #pragma once

2 #include "am\_common.h"

3 #include "am\_digitron\_disp.h"

4 #include "am\_digitron\_dev.h"

5

6 typedef struct am\_digitron\_miniport\_view\_info{

7 am\_digitron\_devinfo\_t devinfo; //数码管设备的ID信息

8 uint8\_t scan\_freq; //整个数码管的扫描频率，一般50Hz

9 uint16\_t blink\_on\_time; //一个闪烁周期内，点亮的时间

10 uint16\_t blink\_off\_time; //一个闪烁周期内，熄灭的时间

11 uint8\_t \*p\_disp\_buf; //数码管显示缓存

12 uint8\_t num\_digitron; //数码管个数

13 uint8\_t num\_segment; //数码管段码个数

14 am\_bool\_t seg\_active\_low; //数码管段端的极性

15 am\_bool\_t com\_active\_low; //数码管公共端的极性

16 const int \*p\_seg\_pins; //段码GPIO驱动引脚

17 const int \*p\_com\_pins; //位码GPIO驱动引脚

18 }am\_digitron\_miniport\_view\_info\_t;

19

20 typedef struct am\_digitron\_miniport\_view{

21 am\_digitron\_dev\_t isa; //派生至抽象的数码管设备

22 am\_softimer\_t timer; //使用软件定时器以自动扫描

23 uint8\_t scan\_idx; //当前扫描索引

24 uint32\_t blink\_flags; //闪烁属性标志

25 uint16\_t link\_cnt; //闪烁计时器

26 uint6\_t (\*pfn\_decode)(uint16\_t code); //函数指针，指向用户设定的解码函数

27 const am\_digitron\_miniport\_view\_info\_t \*p\_info; //指向数码管设备信息的指针

28 }am\_digitron\_miniport\_view\_t;

29

30 int am\_digitron\_miniport\_view\_init(

31 am\_digitron\_miniport\_view\_t \*p\_dev,

32 const am\_digitron\_miniport\_view\_info\_t \*p\_info);

## 待续

# AMetal外设应用详解

## GPIO外设应用详解

### 新建GPIO工程

* 首先在模板工程的基础上新建自己的应用工程，以GPIO应用为例。打开AMetal软件开发包，ametal\board\am116\_core路径下包含project\_example、project\_template两个文件夹，详见图4.1。其中project\_template文件夹包含模板工程。

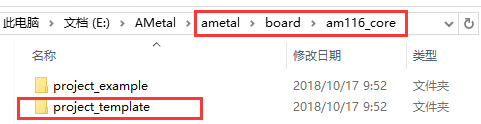


图4.1 模板工程

* 复制project\_template模板工程文件，粘贴并重命名为gpio，详见图4.2。

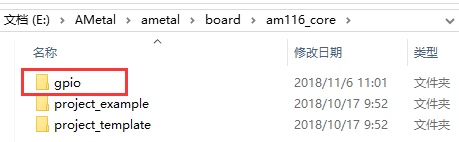


图4.2 gpio工程

* 打开gpio文件夹，找到MDK的模板工程文件，详见图4.3。将template\_am116 \_ core.uvprojx文件重命名为gpio.uvprojx，重命名文件详见图4.4。至此，新建gpio工程完成。

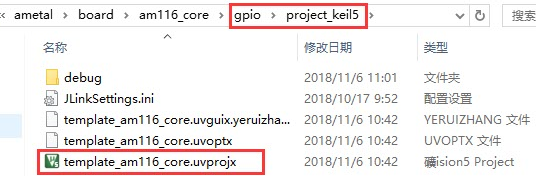


图4.3 模板工程文件

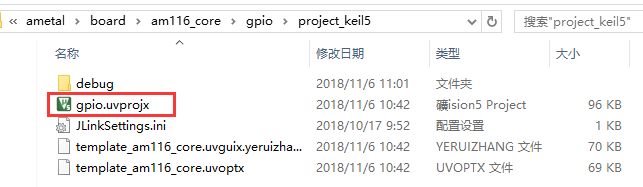


图4.4 gpio工程文件

* 打开新建的gpio工程，编译后零错误零警告，编译成功，准备编写gpio应用程序。

### 编写GPIO LED程序

以gpio控制LED为例，首先需要配置引脚功能、输出电平两个参数。AMetal平台提供了gpio初始化函数、输出配置函数等，函数在soc目录下am\_zlg116\_gpio.c中定义，在头文件am\_ gpio.h中声明。因此调用函数配置gpio时需要包含头文件am\_gpio.h。

#### 引脚初始化

使用gpio功能，必须先配置引脚功能，配置函数原型（am\_gpio.h）为：

int am\_gpio\_pin\_cfg (int pin, uint32\_t flags);

该函数意在配置引脚功能，包含两个参数，其中：

* pin为使用的I/O引脚，其可用的值已在zlg116\_pin.h中宏定义，zlg116\_pin.h可在user\_config目录下am\_hwconf\_zlg116\_gpio.c包含的头文件中找到。例如使用PB1，参数为PIOB\_1。
* flags为配置引脚功能、模式，其可用的值已在am\_gpio.h中宏定义，引脚功能及模式详见表4.1。

表4.1 引脚功能及模式

|  |  |
| --- | --- |
| 宏名 | 功能说明 |
| AM\_GPIO\_INPUT | GPIO引脚输入 |
| AM\_GPIO\_OUTPUT\_INIT\_HIGH | GPIO引脚输出高电平 |
| AM\_GPIO\_OUTPUT\_INIT\_LOW | GPIO引脚输出低电平 |
| AM\_GPIO\_PULLUP | 上拉模式 |
| AM\_GPIO\_PULLDOWN | 下拉模式 |
| AM\_GPIO\_FLOAT | 浮动模式 |
| AM\_GPIO\_OPEN\_DRAIN | 开漏模式 |
| AM\_GPIO\_PUSH\_PULL | 推挽模式 |

基于以上信息，配置引脚功能和模式，例如使用PB1控制LED，配置程序为：

am\_gpio\_pin\_cfg (PIOB\_1, AM\_GPIO\_OUTPUT\_INIT\_HIGH);

#### 输出控制

引脚初始化后，可调用设置引脚状态函数，设置引脚输出低电平或高电平，控制LED灯的亮、灭。函数原型（am\_gpio.h）为：

int am\_gpio\_set(int pin, int value);

该函数意在设置引脚输出电平，其包含两个参数：

* pin为使用的I/O引脚，其值和初始化时的参数一致，例如初始化时为PB1引脚，参数为PIOB\_1；
* value为引脚输出的电平状态，要使引脚输出高电平，value值为1；要使引脚输出低电平，value值为0。其可用的值已用宏定义，分别为AM\_GPIO\_LEVEL\_HIGH、AM\_GPIO\_LEVEL\_LOW；

基于以上信息，设置PB1引脚输出高电平的程序为：

am\_gpio\_set (PIOB\_1, AM\_GPIO\_LEVEL\_HIGH);

同理可设置PB1引脚输出低电平的程序为：

am\_gpio\_set (PIOB\_1, AM\_GPIO\_LEVEL\_LOW);

在输出高、低电平的程序后面运行延时函数，便可以看到LED闪烁。AMetal平台提供了延时函数，函数在am\_delay.h中声明，am\_delay.h包含在bsp\_common目录下am\_bsp\_delay \_timer.c的头文件中，函数原型为：

void am\_mdelay(uint32\_t nms);

* nms为延时时间，单位为ms。

#### gpio控制LED程序

基于上述配置，编写程序，详见程序清单4.1，编译下载到AM116-core板子，复位后可看到LED0闪烁。

程序清单4.1 GPIO输出范例

1 #include "ametal.h"

2 #include "am\_board.h"

3 #include "am\_vdebug.h"

4 #include "am\_delay.h"

5 #include "am\_led.h"

6 #include "zlg116\_pin.h"

7 #include "am\_gpio.h"

8 int am\_main (void)

9 {

10 uint8\_t key\_value = 0;

11 AM\_DBG\_INFO("Start up successful!\r\n");

12 am\_gpio\_pin\_cfg (PIOB\_1, AM\_GPIO\_OUTPUT\_INIT\_HIGH);

13 am\_gpio\_pin\_cfg (PIOA\_8, AM\_GPIO\_INPUT | AM\_GPIO\_PULLUP);

14 while (1) {

15 key\_value = am\_gpio\_get(PIOA\_8);

16 if (key\_value) {

17 am\_gpio\_set (PIOB\_1, AM\_GPIO\_LEVEL\_HIGH);

18 } else {

19 am\_gpio\_set (PIOB\_1, AM\_GPIO\_LEVEL\_LOW);

20 }

21 }

22 }

### 编写GPIO按键程序

在4.1.2节中，已经使用了GPIO控制LED，学会了GPIO初始化及输出控制。本节通过按键控制LED的亮灭，学习GPIO输入功能，先配置I/O引脚，然后调用获取引脚状态函数获取引脚电平状态，进而控制LED。

#### 引脚初始化

用于一个按键控制一个LED，需要两个GPIO，以PB1控制LED、PA8连接按键为例，需要初始化PB1引脚为输出、PA8为输入。配置程序如下：

am\_gpio\_pin\_cfg (PIOB\_1, AM\_GPIO\_OUTPUT\_INIT\_HIGH); // 配置为输出，并输出高电平

am\_gpio\_pin\_cfg (PIOA\_8, AM\_GPIO\_INPUT | AM\_GPIO\_PULLUP); // 配置为上拉输入

#### 获取引脚状态

GPIO初始化完成后，可调用获取引脚状态函数，获取按键引脚电平状态，程序如下：

uint8\_t key\_value = 0;

key\_value = am\_gpio\_get(PIOA\_8);

#### 按键控制LED程序

获取按键引脚状态后，可根据引脚电平控制LED的亮灭。程序详见程序清单4.2。

程序清单4.2 按键控制LED

1 #include "ametal.h"

2 #include "am\_board.h"

3 #include "am\_vdebug.h"

4 #include "am\_delay.h"

5 #include "am\_led.h"

6 #include "zlg116\_pin.h"

7 #include "am\_gpio.h"

8 int am\_main (void)

9 {

10 uint8\_t key\_value = 0;

11 AM\_DBG\_INFO("Start up successful!\r\n");

12 am\_gpio\_pin\_cfg (PIOB\_1, AM\_GPIO\_OUTPUT\_INIT\_HIGH);

13 am\_gpio\_pin\_cfg (PIOA\_8, AM\_GPIO\_INPUT | AM\_GPIO\_PULLUP);

14 while (1) {

15 key\_value = am\_gpio\_get(PIOA\_8);

16 if (key\_value) {

17 am\_gpio\_set (PIOB\_1, AM\_GPIO\_LEVEL\_HIGH);

18 } else {

19 am\_gpio\_set (PIOB\_1, AM\_GPIO\_LEVEL\_LOW);

20 }

21 }

22 }

## UART外设应用详解

### 新建UART工程

本节将详解UART通信应用，首先仿照4.1.1节内容，建立一个名为test\_uart的工程。

### UART介绍

UART是一种通用串行数据总线，用于异步通信。该总线双向通信，通过TXD和RXD实现全双工传输和接收。数据的传输过程是一条数据线上一位一位地进行，它将要传输的资料在串行通信与并行通信之间加以转换，UART通信接口详见图4.5。

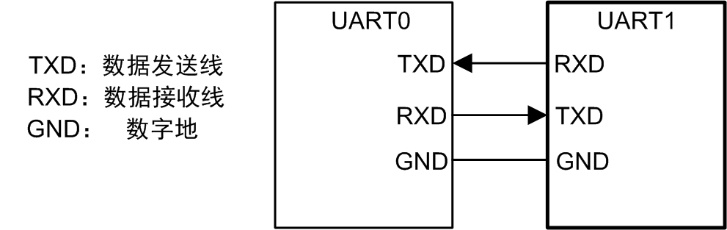


图4.5 UART通信接口

### 编写UART应用程序

#### UART初始化

AMetal平台提供了UART初始化函数，可以直接调用初始化函数。以UART1初始化函数为例，函数原型为：

am\_uart\_handle\_t am\_zlg116\_uart1\_inst\_init(void);

该函数在user\_config目录下的am\_hwconf\_zlg116\_uart.c文件中定义为：

/\*\* \brief UART1实例初始化，获得uart1标准服务句柄 \*/

am\_uart\_handle\_t am\_zlg116\_uart1\_inst\_init (void)

{

return am\_zlg\_uart\_init(&\_\_g\_uart1\_dev, &\_\_g\_uart1\_devinfo);

}

在am\_zlg 116\_inst\_init.h中声明为：

/\*\*

\* \brief UART1 实例初始化，获得 UART 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\_zlg116\_uart1\_inst\_init (void);

因此使用UART初始化函数时，需要包含头文件am\_zlg116\_inst\_init.h。

初始化uart1，调用该函数时需要定义一个am\_uart\_handle\_t类型的变量，用于保存获取的uart1服务句柄，uart1初始化程序为：

am\_uart\_handle\_t uart\_handle;

uart\_handle = am\_zlg116\_uart1\_inst\_init();

#### UART接口函数调用

AMetal平台已提供UART标准接口函数，在am\_uart.h中声明，am\_uart.h可在service目录下的am\_uart.c包含的头文件中找到。

其中包含的UART标准接口函数详见表4.2。

表4.2 UART标准接口函数

|  |  |
| --- | --- |
| UART标准接口函数 | 功能说明 |
| am\_uart\_ioctl( ); | 串口控制函数； |
| am\_uart\_tx\_startup( ); | 启动UART中断模式数据传输； |
| am\_uart\_callback\_set( ); | 设置UART回调函数； |
| am\_uart\_poll\_putchar( ); | UART发送一个数据（查询模式）； |
| am\_uart\_poll\_getchar( ); | UART接收一个数据（查询模式）； |
| am\_uart\_poll\_send( ); | UART数据发送（查询模式）； |
| am\_uart\_poll\_receive( ); | UART数据接收（查询模式）； |

1. UART控制函数详解

串口控制函数函数原型为：

am\_static\_inline int am\_uart\_ioctl (am\_uart\_handle\_t handle, int request, void \*p\_arg);

* handle为UART的服务句柄，即为初始化UART1获取的句柄；
* request为控制指令；
* p\_arg为该指令对应的参数。
* 其中request控制指令及p\_arg参数详见表4.3。

表4.3 控制指令及参数

|  |  |  |
| --- | --- | --- |
| request控制指令 | 指令说明 | p\_arg参数 |
| AM\_UART\_BAUD\_SET; | 设置波特率； | uint32\_t指针类型、值为波特率； |
| AM\_UART\_BAUD\_GET; | 获取波特率； | uint32\_t指针类型； |
| AM\_UART\_OPTS\_SET; | 设置硬件参数； |
| AM\_UART\_OPTS\_GET; | 获取硬件参数； |
| AM\_UART\_AVAIL\_MODES\_GET; | 获取当前可用的模式； |
| AM\_UART\_MODE\_GET; | 获取当前模式； |
| AM\_UART\_MODE\_SET; | 设置模式； | 值为AM\_UART\_MODE\_POLL  或AM\_UART\_MODE\_INT； |
| AM\_UART\_FLOWMODE\_SET; | 设置流控模式； | 值为AM\_UART\_FLOWCTL\_NO  或AM\_UART\_FLOWCTL\_OFF； |
| AM\_UART\_FLOWSTAT\_RX\_SET; | 设置接收器流控状态； | 值为AM\_UART\_FLOWSTAT\_ON 或AM\_UART\_FLOWSTAT\_OFF； |
| AM\_UART\_FLOWSTAT\_TX\_GET; | 获取发送器流控状态； |
| AM\_UART\_RS485\_SET; | 设置RS485模式； | 值为bool\_t类型，TURE（使能），FALSE（禁能）； |
| AM\_UART\_RS485\_GET; | 获取RS485模式状态； | 参数为 bool\_t 指针类型； |

基于以上信息，UART控制函数设置波特率程序为：

uint32\_t BAUD = 115200;

am\_uart\_ioctl (uart\_handle, AM\_UART\_BAUD\_SET, &BAUD);

1. 启动UART中断模式数据传输详解

启动UART中断模式数据传输函数原型为：

am\_static\_inline int am\_uart\_tx\_startup (am\_uart\_handle\_t handle);

* handle为UART的服务句柄，即为初始化UART1获取的句柄。

基于以上信息，启动UART中断模式数据传输程序为：

am\_uart\_tx\_startup(uart\_handle);

1. 设置UART回调函数详解

当发生错误时，调用回调函数，执行错误回调函数。回调函数与普通函数一样，程序详见程序清单4.3。

程序清单4.3 uart回调函数

1 void uart\_callback(void \*p\_arg)

2 {

3 AM\_DBG\_INFO(“UART ERROR!\r\n”);

4 }

设置UART回调函数函数原型为：

am\_static\_inline int am\_uart\_callback\_set ( am\_uart\_handle\_t handle,

int callback\_type,

void \*pfn\_callback,

void \*p\_arg);

* handle为UART的服务句柄，即为初始化UART1获取的句柄；
* callback\_type为指明设置的何种回调函数；
* pfn\_callback为指向回调函数的指针；
* p\_arg为回调函数的用户参数。

其中callback\_type回调函数类型详见表4.4。

表4.4 callback\_type回调函数类型

|  |  |
| --- | --- |
| 回调函数类型 | 类型说明 |
| 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； |

基于以上信息，设置错误回调函数程序为：

am\_uart\_callback\_set(uart\_handle, AM\_UART\_CALLBACK\_ERROR, uart\_callback, NULL);

1. UART 查询方式发送、接收一个数据详解

UART发送一个数据(查询模式)函数原型为：

am\_static\_inline int am\_uart\_poll\_putchar (am\_uart\_handle\_t handle, char outchar);

* handle为UART的服务句柄，即为初始化UART1获取的句柄；
* outchar为待发送的数据。

UART接收一个数据(查询模式)函数原型为：

am\_static\_inline int am\_uart\_poll\_getchar (am\_uart\_handle\_t handle, char \*p\_inchar);

* handle为UART的服务句柄，即为初始化UART1获取的句柄；
* p\_inchar用于获取数据的指针，即传入一个内存地址。

UART 查询方式发送、接收一个数据示例详见程序清单4.4。

程序清单4.4 uart查询方式收发一个数据

1 #include "ametal.h"

2 #include "am\_board.h"

3 #include "am\_vdebug.h"

4 #include "am\_delay.h"

5 #include "am\_zlg116\_inst\_init.h"

6 int am\_main (void)

7 {

8 char uart1\_buf[1] ={0}; // 数据缓冲区

9 am\_uart\_handle\_t uart\_handle; // 串口标准服务句柄

10 uart\_handle = am\_zlg116\_uart1\_inst\_init(); // UART初始化

11 while (1) {

12 am\_uart\_poll\_getchar(uart\_handle, uart1\_buf); // 接收字符

13 am\_uart\_poll\_putchar(uart\_handle, uart1\_buf[0]); // 发送刚刚接收的字符

14 am\_mdelay(1000);

15 }

16 }

1. UART 查询方式发送、接收数据详解

UART数据发送（查询模式）函数原型为：

int am\_uart\_poll\_send (am\_uart\_handle\_t handle, const uint8\_t \*p\_txbuf, uint32\_t nbytes);

* handle为UART的服务句柄，即为初始化UART1获取的句柄；
* p\_txbuf为指向发送缓冲区的指针，即传入一个内存地址；
* nbytes为发送的字节数。

UART数据接收（查询模式）函数原型为：

int am\_uart\_poll\_receive (am\_uart\_handle\_t handle, uint8\_t \*p\_rxbuf, uint32\_t nbytes);

* handle为UART的服务句柄，即为初始化UART1获取的句柄；
* p\_rxbuf为指向接收缓冲区的指针，即传入一个内存地址；
* nbytes为接收的字节数。

UART 查询方式发送、接收数据示例详见程序清单4.5。

程序清单4.5 uart查询方式收发数据

1 #include "ametal.h"

2 #include "am\_board.h"

3 #include "am\_vdebug.h"

4 #include "am\_zlg116\_inst\_init.h"

5 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

6 全局变量

7 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

8 static const uint8\_t \_\_ch[ ] = {"STD-UART test in polling mode:\r\n"};

9 int am\_main (void)

10 {

11 uint8\_t uart1\_buf[5]; // 数据缓冲区

12 am\_uart\_handle\_t uart\_handle; // 串口标准服务句柄

13 uart\_handle = am\_zlg116\_uart1\_inst\_init(); // UART初始化

14 am\_uart\_poll\_send(uart\_handle, \_\_ch, sizeof(\_\_ch));

15 while (1) {

16 am\_uart\_poll\_receive(uart\_handle, uart1\_buf, 1); // 接收字符

17 am\_uart\_poll\_send(uart\_handle, uart1\_buf, 1); // 发送刚刚接收的字符

18 }

19 }

## I2C外设应用详解

本节将讲解I2C通信应用，ZLG116仅有一个I2C，PIOB\_6为I2C1\_SCL，PIOB\_7为I2C1\_SDA，首先仿照4.1.1节内容，建立一个名为test\_i2c的工程。

### I2C主机程序

1. **I2C主机初始化**

AMetal平台提供了I2C初始化函数，可以直接调用初始化函数。函数原型为：

am\_i2c\_handle\_t am\_zlg116\_i2c1\_inst\_init (void);

该函数在user\_config目录下的am\_hwconf\_zlg116\_i2c.c文件中定义，在am\_zlg116\_ inst\_init.h中声明。因此使用I2C初始化函数时，需要包含头文件am\_zlg116\_inst\_init.h。

初始化I2C，调用该函数时需要定义一个am\_i2c\_handle\_t类型的变量，用于保存获取的I2C服务句柄，初始化程序为：

am\_i2c\_handle\_t i2c\_handle;

i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

1. **构造I2C设备**

获取了I2C服务句柄后，还应该有一个描述I2C从设备的结构体，构造I2C设备函数的原型为：

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为指向am\_i2c\_device\_t的结构体指针；
* handle为I2C服务句柄；dev\_addr为从机设备地址；
* dev\_flags为传输过程中的控制标识位，其可用的值已在am\_i2c.h中宏定义，详见表4.5。

表4.5 I2C控制标识位宏定义

|  |  |  |
| --- | --- | --- |
| 属性 | 宏名 | 功能说明 |
| 从机地址 | AM\_I2C\_M\_7BIT | 7-bit从机地址（默认）； |
| AM\_I2C\_M\_10BIT | 10-bit从机地址； |
| 器件内子地址 | AM\_I2C\_SUBADDR\_NONE | 无子地址（默认）； |
| AM\_I2C\_SUBADDR\_1BYTE | 1字节子地址； |
| AM\_I2C\_SUBADDR\_2BYTE | 2字节子地址； |
| 传输顺序 | AM\_I2C\_SUBADDR\_MSB\_FIRST | 高位字节先传输（默认）； |
| AM\_I2C\_SUBADDR\_LSB \_FIRST | 低位字节先传输； |

以通过同步标准接口实现的I2C主机访问EEPROM设备例程为例：

am\_i2c\_device\_t i2cdev\_eeprom; // 定义I2C设备描述结构体

am\_i2c\_mkdev(&i2cdev\_eeprom, // 构造I2C设备

i2c\_handle,

eeprom\_addr,

AM\_I2C\_ADDR\_7BIT | AM\_I2C\_SUBADDR\_1BYTE);

首先，定义一个I2C设备描述结构体，它的类型是am\_i2c\_device\_t，然后调用构造I2C设备函数来将其初始化，初始化完成后，即可调用I2C读写函数。

1. **读写数据**

获取了I2C设备描述结构体后，即可直接调用I2C读写函数来通过I2C传输数据；写入数据的函数原型为：

int am\_i2c\_write (am\_i2c\_device\_t \*p\_dev,

uint32\_t sub\_addr,

const uint8\_t \*p\_buf,

uint32\_t nbytes)

其中p\_dev指向获取到的I2C设备描述结构体；sub\_addr为I2C设备子地址；p\_buf指向需要写入数据；nbytes为指定写入多少字节。

读取数据的函数与写入数据的函数基本一致，函数原型为：

int am\_i2c\_read (am\_i2c\_device\_t \*p\_dev,

uint32\_t sub\_addr,

uint8\_t \*p\_buf,

uint32\_t nbytes)

与写入数据函数不同的是：p\_buf指向的是保存读取数据的数组；nbytes为指定读取多少字节。

1. **I2C主机范例**

以I**2**C主机为例，在从机地址为0x00的设备中读写数据的范例详见程序清单4.6。

程序清单4.6 I**2**C主机范例

1 #include "ametal.h"

2 #include "am\_board.h"

3 #include "am\_vdebug.h"

4 #include "am\_zlg116\_inst\_init.h"

5 int am\_main (void)

6 {

7 am\_err\_t ret;

8 uint8\_t wr\_buf[8] = {0,1,2,3,4,5,6,7}; // 写数据缓存定义

9 uint8\_t rd\_buf[8] = {0}; // 读数据缓存定义

10

11 am\_i2c\_handle\_t handle = am\_zlg116\_i2c1\_inst\_init();

12 am\_i2c\_device\_t i2cdev;

13 am\_i2c\_mkdev(&i2cdev,

14 handle,

15 0x00, // 从机地址

16 AM\_I2C\_ADDR\_7BIT | AM\_I2C\_SUBADDR\_1BYTE);

17

18 ret = am\_i2c\_write(&i2cdev, // 写入数据

19 0x00, // 从机子地址

20 &wr\_buf[0],

21 8);

22 if (ret != AM\_OK) {

23 AM\_DBG\_INFO("am\_i2c\_write error(id: %d).\r\n", ret);

24 }

25

26 ret = am\_i2c\_write(&i2cdev, // 读取数据

27 0x00, // 从机子地址

28 &rd\_buf[0],

29 8);

30 if (ret != AM\_OK) {

31 AM\_DBG\_INFO("am\_i2c\_write error(id: %d).\r\n", ret);

32 }

33

34 while (1) {

35 }

36 }

### I2C从机程序

1. **I2C从机初始化**

和ZLG116作为主机一样，AMetal平台也提供了I2C从机初始化函数，可以直接调用初始化函数。函数原型为：

am\_i2c\_slv\_handle\_t am\_zlg116\_i2c1\_slv\_inst\_init (void)

该函数在user\_config目录下的am\_hwconf\_zlg116\_i2c\_slv.c文件中定义，在am\_zlg116 \_inst\_init.h中声明。因此使用定时器初始化函数时，需要包含头文件am\_zlg116\_inst\_init.h。

初始化I2C，调用该函数时需要定义一个am\_i2c\_slv\_device\_t类型的变量，用于保存获取的I2C从机服务句柄，初始化程序为：

am\_i2c\_slv\_handle\_t slv\_handle = am\_zlg116\_i2c1\_slv\_inst\_init()

1. **构造I2C设备**

获取了I2C服务句柄后，还应该有一个描述I2C从设备的结构体，构造I2C设备函数的原型为：

void am\_i2c\_slv\_mkdev (am\_i2c\_slv\_device\_t \*p\_dev,

am\_i2c\_slv\_handle\_t handle,

am\_i2c\_slv\_cb\_funcs\_t \*p\_cb\_funs,

uint16\_t dev\_addr,

uint16\_t dev\_flags,

void \*p\_arg)

* p\_dev为指向从机设备描述结构体的指针；
* handle是与从设备关联的I2C标准服务操作句柄；
* p\_cb\_funs是回调函数的函数指针；
* dev\_addr为从机设备地址；
* dev\_flags为从机设备特性；
* p\_arg指向回调函数参数。

其中p\_cb\_funs指向的回调函数结构体包括：

* 从机地址匹配时回调函数指针；
* 获取一个发送字节回调函数指针；
* 提交一个接收到的字节回调函数指针；
* 停止传输回调函数指针；
* 广播回调函数指针。

回调函数得用户定义，如果不需要某个回调函数，可以不定义并将其函数指针指向NULL；

从机设备特性已在am\_i2c\_slv.h中宏定义，其宏定义及其功能详见表4.6。

表4.6 从机设备特性宏定义

|  |  |  |
| --- | --- | --- |
| 属性 | 宏名 | 功能说明 |
| 从机地址 | AM\_I2C\_SLV\_ADDR\_7BIT | 7-bit从机地址（默认）； |
| AM\_I2C\_SLV\_ADDR\_10BIT | 10-bit从机地址； |
| 是否响应广播地址 | AM\_I2C\_SLV\_GEN\_CALL\_NACK | 不响应广播地址（默认）； |
| AM\_I2C\_SLV\_GEN\_CALL\_ACK | 响应广播地址； |
| 地址宽度 | AM\_I2C\_SLV\_SUBADDR\_1BYTE | 地址宽度1字节（默认）； |
| AM\_I2C\_SLV\_SUBADDR\_2BYTE | 地址宽度2字节； |
| 子地址自增 | AM\_I2C\_SLV\_SUBADDR\_SELF\_INCREASING | 支持子地址自增 (默认)； |
| AM\_I2C\_SLV\_SUBADDR\_NSELF\_INCREASING | 不支持子地址自增； |

构造I2C设备范例详见程序清单4.7。

程序清单4.7 构造I2C设备范例

1 #include "am\_i2c\_slv.h"

2 int main()

3 {

4 am\_i2c\_slv\_device\_t i2c\_slv\_dev;

5 am\_i2c\_slv\_mkdev(&i2c\_slv\_dev,

6 i2c\_slv\_handle,

7 &\_\_g\_\_cb\_funs,

8 0x55,

9 AM\_I2C\_SLV\_ADDR\_7BIT,

10 NULL);

11 }

1. **开启I2C从机设备**

构建完成后，即可开启I2C从机设备；开启I2C从机设备，函数原型为：

int am\_i2c\_slv\_setup (am\_i2c\_slv\_device\_t \*p\_dev);

1. **I2C从机范例**

在使用过程中，应该首先定义需要的回调函数，然后定义一个am\_i2c\_slv\_cb\_funcs\_t类型的结构体保存定义的回调函数，然后在开始初始化I2C从机设备，详见程序清单4.8

程序清单4.8 I2C从机范例

1 #include "ametal.h"

2 #include "am\_i2c\_slv.h"

3 #include "am\_zlg116\_inst\_init.h"

4

5 #define \_\_REG\_SIZE 20 //寄存器数量

6 static uint8\_t \_\_g\_reg\_buf[\_\_REG\_SIZE];

7 static uint8\_t \_\_g\_subaddr;

8

9 static uint8\_t \_\_g\_count = 0;

10 static int \_\_addr\_matching (void \* p\_arg, am\_bool\_t is\_rx) // 地址匹配回调函数

11 {

12 /\* do some thing \*/

13 \_\_g\_count++; // 子地址自增

14 return AM\_OK;

15 }

16

17 static int \_\_txbyte\_get (void \*p\_arg, uint8\_t \*p\_byte) // 获取发送一个字节回调函数

18 {

19 \*p\_byte = \_\_g\_reg\_buf[\_\_g\_subaddr];

20 \_\_g\_subaddr++; // 子地址自增

21 return AM\_OK;

22 }

23

24 static int \_\_rxbyte\_put(void \*p\_arg, uint8\_t byte) // 接受一个字节回调

25 {

26 \_\_g\_reg\_buf[\_\_g\_subaddr] = byte;

27 \_\_g\_subaddr++; // 子地址自增

28 return AM\_OK;

29 }

30

31 static void \_\_tran\_stop(void \*p\_arg) // 停止传输回调函数

32 {

33 \_\_g\_subaddr = 0;

34 }

35

36 static am\_i2c\_slv\_cb\_funcs\_t \_\_g\_i2c\_slv\_cb\_funs = { // 回调函数指针结构体

37 \_\_addr\_matching,

38 \_\_txbyte\_get,

39 \_\_rxbyte\_put,

40 \_\_tran\_stop,

41 };

42

43 int main()

44 {

45 am\_i2c\_slv\_handle\_t handle = am\_zlg116\_i2c1\_slv\_inst\_init();

46 uint16\_t dev\_addr = 0x50;

47 int i;

48 am\_i2c\_slv\_device\_t i2c\_slv\_dev;

49 am\_i2c\_slv\_mkdev(&i2c\_slv\_dev,

50 handle,

51 &\_\_g\_i2c\_slv\_cb\_funs,

52 dev\_addr,

53 AM\_I2C\_SLV\_ADDR\_7BIT,

54 NULL);

55

56 am\_i2c\_slv\_setup (&i2c\_slv\_dev); // 开始运行从机设备

57

58 for ( i = 0; i < \_\_REG\_SIZE; i++) { // 初始化从机缓存区

59 \_\_g\_reg\_buf[i] = 5+i;

60 }

61 AM\_FOREVER {

62 ; /\* VOID \*/

63 }

64 }

## SPI外设应用详解

### 新建SPI工程

本节将详解SPI通信应用，首先仿照4.1.1节内容，新建一个test\_spi工程。

### SPI总线介绍

无论采用哪种总线，都使用时钟信号和数据/控制线，时钟信号由MCU主机进行控制。首先回顾一下SPI的通信机制：用于控制的信号线中的SPI为4根，除了具有传输的信号外，还具有片选信号，通过该信号的有效与否，主机指定哪个器件作为目标对象，SPI通信接口具体描述如下：

* SSEL：片选输入

当SPI 作为主机时，则在串行数据启动前驱动SSEL信号，使之变为有效状态，并在串行数据发送后释放该信号，使之变为无效状态。默认SSEL为低电平有效，也可将其选为高电平有效。当SPI作为从机时，处于有效状态的任意SSEL信号都表示该从机正在被寻址。

* MOSI：主机输出从机输入

MOSI 信号可将串行数据从主机传送到从机。当SPI 作为主机时，则串行数据从MOSI输出；当SPI作为从机时，则串行数据从MOSI输入。

* MISO：主机输入从机输出

MISO 信号可将串行数据由从机传送到主机。当SPI作为主机时，则串行数据从MISO输入；当SPI作为从机时，则串行数据输出至MISO。

* SCK：时钟信号

SCK同步数据传送时钟信号。它由主机驱动从机接收，使用SPI 接口时，则时钟可编程为高电平有效或低电平有效。

### SPI初始化

在使用SPI通用接口前，必须先完成SPI的初始化，以获取标准的SPI实例句柄。ZLG116支持SPI功能的外设有SPI1和SPI2，为方便用户使用，AMetal提供了与各外设对应的实例初始化函数，详见表4.7。

表4.7 SPI实例初始化函数（am\_zlg116\_inst\_init.h）

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| am\_spi\_handle\_t am\_zlg116\_spi1\_int\_inst\_init (void); | SPI1实例初始化； |
| am\_spi\_handle\_t am\_zlg116\_spi2\_int\_inst\_init (void); | SPI2实例初始化； |

这些函数的返回值均为am\_spi\_handle\_t类型的SPI实例句柄，该句柄将作为SPI通用接口中handle参数的实参。类型am\_spi\_handle\_t（am\_spi.h）定义如下：

typedef struct am\_spi\_serv \*am\_spi\_handle\_t;

因为函数返回的SPI实例句柄仅作为参数传递给SPI通用接口，不需要对该句柄做其它任何操作，因此完全不需要了解该类型。注意，若函数返回的实例句柄的值为NULL，则表明初始化失败，不能使用该实例句柄。

如需使用SPI1，则直接调用SPI1实例初始化函数，即可获取对应的实例句柄：

am\_spi\_handle\_t spi1\_handle = am\_zlg116\_spi1\_int\_inst\_init ();

打开新建工程的main.c文件，添加SPI头文件和ZLG116的外设实例初始化函数声明，在am\_main函数中添加SPI1实例初始化函数，并编译该工程，即可完成SPI初始化。

### SPI接口函数

MCU的SPI主要用于主从机的通信，AMetal提供了8个接口函数，详见表4.8。

表4.8 SPI标准接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| 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 \*p\_dev, int state)); | SPI从机实例初始化； |
| int am\_spi\_setup (am\_spi\_device\_t \*p\_dev); | 设置SPI从机实例； |
| 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); | SPI传输初始化； |
| void am\_spi\_msg\_init(  am\_spi\_message\_t \*p\_msg,  am\_pfnvoid\_t pfn\_complete,  void \*p\_arg); | SPI消息初始化； |
| void am\_spi\_trans\_add\_tail(  am\_spi\_message\_t \*p\_msg,  am\_spi\_transfer\_t \*p\_trans); | 添加传输至消息中； |
| int am\_spi\_msg\_start (  am\_spi\_device\_t \*p\_dev,  am\_spi\_message\_t \*p\_msg); | 启动SPI消息处理； |
| 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); | SPI先写后读； |
| 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); | 执行SPI两次写； |

### SPI从机初始化

本例中选择MX25L1606为从机，MCU通过SPI对它写入数据。MX25L1606总容量为16M（16×1024×1024）bits，即2M字节。每个字节对应一个存储地址，因此其存储数据的地址范围为0x000000 ~ 0x1FFFFF，电路详见图4.6。

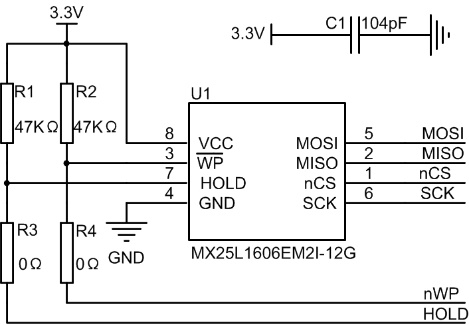


图4.6 SPI Flash电路原理图

#### 从机实例初始化

对于用户来说，使用SPI往往是直接操作一个从机器件，MCU作为SPI主机，为了与从机器件通信，需要知道从机器件的相关信息，比如，SPI模式、SPI速率、数据位宽等。

这就需要定义一个与从机器件对应的实例（从机实例），并使用相关信息完成对从机实例的初始化。其函数原型为：

void am\_spi\_mkdev (

am\_spi\_device\_t \*p\_dev, // 待初始化的从机实例

am\_spi\_handle\_t handle, // SPI句柄（通过SPI实例初始化函数获得）

uint8\_t bits\_per\_word, // 数据宽度，为0默认8bit

uint16\_t mode, // 模式选择，详见表4.9

uint32\_t max\_speed\_hz, // 从设备支持的最高时钟频率

int cs\_pin, // 片选引脚

void (\*pfunc\_cs)(am\_spi\_device\_t \*p\_dev, int state));

p\_dev是指向SPI从机实例描述符的指针，am\_spi\_device\_t在am\_spi.h文件中定义：

typedef struct am\_spi\_device am\_spi\_device\_t;

该类型用于定义从机实例，用户无需知道其定义的具体内容，只需要使用该类型定义一个从机实例。即：

am\_spi\_device\_t spi\_dev; // 定义一个SPI从机实例

表4.9 SPI常用模式标志

|  |  |  |
| --- | --- | --- |
| 模式标志 | 含义 | 解释 |
| AM\_SPI\_MODE\_0 | SPI模式0 | CPOL=0，CPHA=0 |
| AM\_SPI\_MODE\_1 | SPI模式1 | CPOL=0，CPHA=1 |
| AM\_SPI\_MODE\_2 | SPI模式2 | CPOL=1，CPHA=0 |
| AM\_SPI\_MODE\_3 | SPI模式3 | CPOL=1，CPHA=1 |

mode指定使用的模式，SPI协议定义了4种模式，详见表4.9。各种模式的主要区别在于空闲时钟极性（CPOL）和时钟相位选择（CPHA）的不同。CPOL和CPHA均有两种选择，因此两两组合可以构成4种不同的模式，即模式0~3。当CPOL为0时，表示时钟空闲时，时钟线为低电平，反之，空闲时为高电平；当CPHA为0时，表示数据在第1个时钟边沿采样，反之，则表示数据在第2个时钟边沿采样。

cs\_pin和pfunc\_cs均与片选引脚相关。pfunc\_cs是指向自定义片选控制函数的指针，若pfunc\_cs的值为NULL，驱动将自动控制由cs\_pin指定的引脚实现片选控制；若pfunc\_cs的值不为NULL，指向了有效的自定义片选控制函数，则cs\_pin不再被使用，片选控制将完全由应用实现。当需要片选引脚有效时，驱动将自动调用pfunc\_cs指向的函数，并传递state的值为1。当需要片选引脚无效时，也会调用pfunc\_cs指向的函数，并传递state的值为0。一般情况下，片选引脚自动控制即可，即设置pfunc\_cs的值为NULL，cs\_pin为片选引脚，如PIOA\_4。使用范例详见程序清单4.9。

程序清单4.9 am\_spi\_mkdev()范例程序

1 am\_spi\_handle\_t spi0\_hanlde = am\_ zlg116\_spi1\_inst\_init(); // 使用ZLG116的SPI1获取SPI句柄

2 am\_spi\_device\_tspi\_dev; // 定义从机设备

3 am\_spi\_mkdev(

4 &spi\_dev, // 传递从机设备

5 spi1\_handle, // SPI1操作句柄

6 8, // 数据宽度为8-bit

7 AM\_SPI\_MODE\_0, // 选择模式0

8 3000000, // 最大频率3000000Hz

9 PIOA\_4, // 片选引脚PIOA\_4

10 NULL); // 无自定义片选控制函数，设置为NULL

#### 设置从机实例

设置SPI从机实例时，会检查MCU的SPI主机是否支持从机实例的相关参数和模式。如果不能支持，则设置失败，说明该从机不能使用。其函数原型为：

int am\_spi\_setup (am\_spi\_device\_t \*p\_dev);

其中的p\_dev是指向SPI从机实例描述符的指针，如果返回AM\_OK，说明设置成功；

如果返回-AM\_ENOTSUP，说明设置失败，不支持的位宽、模式等，详见程序清单4.10。

程序清单4.10 am\_spi\_setup()范例程序

1 am\_spi\_handle\_t spi0\_handle = am\_ zlg116\_spi1\_inst\_init(); // 使用ZLG116的SPI1获取SPI句柄

2 am\_spi\_device\_t spi\_dev;

3 am\_spi\_mkdev(

4 &spi\_dev,

5 spi1\_handle,

6 8, // 数据宽度为8-bit

7 AM\_SPI\_MODE\_0, // 选择模式0

8 3000000, // 最大频率3000000Hz

9 PIOA\_4, // 片选引脚PIOA\_4

10 NULL); // 无自定义片选控制函数，设置为NULL

11 am\_spi\_setup(&spi\_dev); // 设置SPI从设备

#### 传输初始化

在AMetal中，将收发一次数据的过程抽象为一个“**传输”**的概念，要完成一次数据传输，首先就需要初始化一个传输结构体，指定该次数据传输的相关信息。其函数原型为：

void am\_spi\_mktrans(

am\_spi\_transfer\_t \*p\_trans, // 待初始化的SPI传输

const void \*p\_txbuf, // 发送数据缓冲区，NULL无数据

void \*p\_rxbuf, // 接收数据缓冲区，NULL无数据

uint32\_t nbytes, // 传输的字节数

uint8\_t cs\_change, // 传输是否影响片选，0-不影响，1-影响

uint8\_t bits\_per\_word, // 为0默认使用设备的字大小

uint16\_t delay\_usecs, // 传输结束后的延时（us）

uint32\_t speed\_hz, // 为0默认使用设备中的max\_speed\_hz

uint32\_t f lags); // 本次传输的特殊标志，详见表4.10

其中，p\_trans为指向SPI传输结构体的指针，am\_spi\_transfer\_t 类型是在am\_spi.h中定义的。即：

typedef struct am\_spi\_transfer am\_spi\_transfer\_t;

在实际使用时，只需要定义一个该类型的传输结构体即可。比如：

am\_spi\_transfer\_t spi\_trans; // 定义一个SPI传输结构体

表4.10传输特殊控制标志

|  |  |
| --- | --- |
| 标志宏 | 含义 |
| AM\_SPI\_READ\_MOSI\_HIGH | 读数据时，MOSI输出高电平，默认低电平 |

因为SPI是全双工通信协议，所以单次传输过程中同时包含了数据的发送和接收。函数的参数中，p\_txbuf指定了发送数据的缓冲区，p\_rxbuf指定了接收数据的缓冲区，nbytes指定了传输的字节数。特别地，有时候可能只希望单向传输数据，若只发送数据，则可以设置p\_rxbuf为NULL；若只接收数据，则可以设置p\_txbuf为NULL。

当传输正常进行时，片选会置为有效状态，cs\_change的值将影响片选何时被置为无效状态。若cs\_change的值为0，表明不影响片选，此时，仅当该次传输是消息（多次传输组成一个消息，消息的概念后文会介绍）的最后一次传输时，片选才会被置为无效状态。若cs\_change的值为1，表明影响片选，此时，若该次传输不是消息的最后一次传输，则在本次传输结束后会立即将片选设置为无效状态，若该次传输是消息的最后一次传输，则不会立即设置片选无效，而是保持有效直到下一个消息的第一次传输开始，详见程序清单4.11。

程序清单4.11 am\_spi\_mktrans()范例程序

1 uint8\_t tx\_buf[8];

2 uint8\_t rx\_buf[8];

3 am\_spi\_transfer\_t spi\_trans;

4

5 am\_spi\_mktrans(

6 &spi\_trans,

7 tx\_buf, // 发送数据缓冲区

8 rx\_buf, // 接收数据缓冲区

9 8, // 传输数据个数为8

10 0, // 本次传输不影响片选

11 0, // 位宽为0，使用默认位宽（设备中的位宽）

12 0, // 传输后无需延时

13 0, // 时钟频率，使用默认速率

14 0); // 无特殊标志

#### 消息初始化

一般来说，与实际的SPI器件通信时，往往采用的是“命令”+“数据”的格式，这就需要两次传输：一次传输命令，一次传输数据。为此，AMetal提出了“消息”的概念，一个消息的处理即为一次有实际意义的SPI通信，其间可能包含一次或多次传输。

一次消息处理中可能包含很多次的传输，耗时可能较长，为避免阻塞，消息的处理采用异步方式。这就要求指定一个完成回调函数，当消息处理完毕时，自动调用回调函数以通知用户消息处理完毕。回调函数的指定在初始化函数中完成，初始化函数的原型为：

void am\_spi\_msg\_init (

am\_spi\_message\_t \*p\_msg, // 待初始化的SPI传输

am\_pfnvoid\_t pfn\_complete, // 消息处理完成回调函数

void \*p\_arg); // 回调函数的参数

其中的p\_msg为指向SPI消息结构体的指针，am\_spi\_message\_t类型是在am\_spi.h中定义的。即：

typedef struct am\_spi\_message am\_spi\_message\_t;

实际使用时，仅需使用该类型定义一个消息结构体。即：

am\_spi\_message\_t spi\_msg; // 定义一个SPI消息结构体

pfn\_callback指向的是消息处理完成回调函数，当消息处理完毕时，将调用指针指向的函数。其类型am\_pfnvoid\_t在am\_types.h中定义的。即：

typedef void (\*am\_pfnvoid\_t) (void \*);

由此可见，函数指针指向的是参数为void \*类型的无返回值函数。驱动调用回调函数时，传递给该回调函数的void\*类型的参数即为p\_arg的设定值，详见程序清单4.12。

程序清单4.12 am\_spi\_msg\_init()范例程序

1 static void \_\_spi\_msg\_complete\_callback (void \*p\_arg)

2 {

3 // 消息处理完毕

4 }

5

6 int am\_main()

7 {

8 am\_spi\_message\_t spi\_msg; // 定义一个SPI消息结构体

9

10 am\_spi\_msg\_init (

11 &spi\_msg,

12 \_\_spi\_msg\_complete\_callback, // 消息处理完成回调函数

13 NULL); // 未使用回调函数的参数p\_arg，设置为NULL

14 }

### Flash擦读写

SPI传输和SPI消息实现数据的发送和接收使得SPI的使用非常灵活，可以支持丰富的SPI从机器件。但正因为其灵活性，使得接口较多，使用起来较为繁琐。对于绝大部分SPI从机器件，并不需要如此灵活，只需要实现简单的数据发送和接收就可以了，基于此，AMetal提供了两种十分常用的情形：写入一段数据后读取一段数据（先写后读）；写入一段数据后再写入一段数据（连续两次写）。

#### Flash擦除

AMetal提供了对SPI Flash的擦除函数，指定擦除数据的地址，其函数原型如下：

static void spi\_flash\_erase (am\_spi\_device\_t \*p\_dev,

uint32\_t addr )

#### Flash写数据

AMetal提供了对SPI Flash的写入函数，指定写入数据的地址及缓存数据长度，其函数原型如下：

static void spi\_flash\_write (am\_spi\_device\_t \*p\_dev,

uint32\_t addr,

uint32\_t length)

#### Flash读数据

AMetal提供了对SPI Flash的读取函数，指定读取数据的地址及缓存数据长度，其函数原型如下：

static void spi\_flash\_read (am\_spi\_device\_t \*p\_dev,

uint32\_t addr,

uint32\_t length)

#### Flash先读后写

先写后读即是主机先发送数据至从机（写），再自从机接收数据（读）。注意，该函数会等待数据传输完成后才会返回，因此该函数是阻塞式的，不应在中断环境中调用。其函数原型为：

int am\_spi\_write\_then\_read (

am\_spi\_device\_t \*p\_dev, // SPI从机设备描述符指针

const uint8\_t \*p\_txbuf, // 发送数据缓冲区

size\_t n\_tx, // 发送数据的字节个数

uint8\_t \*p\_rxbuf, // 接收数据缓冲区

size\_t n\_rx); // 接收数据的字节个数

如果返回AM\_OK，说明数据写和读成功完成；如果返回-AM\_EINVAL，说明由于参数错误导致数据写和读失败；如果返回-AM\_EIO，说明在数据写或读的过程中发生错误，详见程序清单4.13。

程序清单4.13 am\_spi\_write\_then\_read()范例程序

1 uint8\_t tx\_buf[2];

2 uint8\_t rx\_buf[8];

3

4 am\_spi\_write\_then\_read (&spi\_dev, tx\_buf, 2, rx\_buf, 8); // 先发送2个数据，再接收8个数据

#### 连续两次写

连续两次写即是主机先发送缓冲区0的数据至从机（写），再发送缓冲区1的数据至从机（写）。如果只需要发送一次数据，可以将第二次发送的数据缓冲区设置为NULL，并设置发送长度n\_tx1为0。值得注意的是，该函数同样是阻塞式的，会等待两次数据发送完成后才会返回，不应在中断环境中调用。其函数原型为：

int am\_spi\_write\_then\_write (

am\_spi\_device\_t \*p\_dev, // SPI从机设备描述符指针

const uint8\_t \*p\_txbuf0, // 发送数据缓冲区0

size\_t n\_tx0, // 缓冲区0的数据个数

const uint8\_t \*p\_txbuf1, // 发送数据缓冲区1

size\_t n\_tx1); // 缓冲区1的数据个数

如果返回AM\_OK，说明消息处理成功；如果返回-AM\_EINVAL，说明参数错误导致数据发送失败；如果返回-AM\_EIO，说明在发送数据的过程中发生错误，详见程序清单4.14。

程序清单4.14 am\_spi\_write\_then\_write()范例程序

1 // tx\_buf0[]和tx\_buf1中填入要发送的数据

2 uint8\_t tx\_buf0[2];

3 uint8\_t tx\_buf1[8];

4 am\_spi\_write\_then\_write (&spi\_dev, tx\_buf0, 2, tx\_buf1, 8); // 先发送2个数据，再发送8个数据

### SPI flash读写范例

对地址为0x0000进行擦写读操作，并将写入数据与读出数据做比较，详见程序清单4.15。

程序清单4.15 flash读写范例程序

1 #include "ametal.h"

2 #include "am\_board.h"

3 #include "am\_vdebug.h"

4 #include "am\_delay.h"

5 #include "am\_gpio.h"

6 #include "demo\_all\_entries.h"

7 #include "am\_spi.h"

8 #include "am\_zlg116\_inst\_init.h"

9 #include "zlg116\_pin.h"

10 #define FLASH\_PAGE\_SIZE 256 // SPI Flsah 页大小定义

11 #define TEST\_ADDR 0x0000 // 测试地址

12 #define TEST\_LEN FLASH\_PAGE\_SIZE // 测试字节长度

13 static uint8\_t g\_tx\_buf[TEST\_LEN]={0x9F}; // 写数据缓存

14 static uint8\_t g\_rx\_buf[TEST\_LEN]={0}; // 读数据缓存

15 int am\_main (void)

16 {

17 uint32\_t length , i;

18 AM\_DBG\_INFO("Start up successful!\r\n");

19 am\_spi\_handle\_t spi\_handle = am\_zlg116\_spi1\_int\_inst\_init();

20 am\_spi\_device\_t spi\_dev;

21 am\_spi\_mkdev(

22 &spi\_dev,

23 spi\_handle,

24 8, // 数据宽度8-bit

25 AM\_SPI\_MODE\_0, // 模式0

26 3000000, // 最大频率3000000Hz

27 PIOA\_4, // 片选PIOA\_4

28 NULL); // 无自定义函数，设置NULL

29 am\_spi\_setup(&spi\_dev); // 设置从机设备

30 spi\_flash\_erase(&spi\_dev, TEST\_ADDR); // 擦除当前地址中数据

31 AM\_DBG\_INFO("FLASH擦除完成\r\n");

32 for (i = 0; i < length; i++) { // 填充数据

33 g\_tx\_buf[i] = i + 1;

34 }

35 spi\_flash\_write(&spi\_dev, TEST\_ADDR, length); // 写入数据到设定SPI\_FLASH地址

36 am\_mdelay(10);

37 AM\_DBG\_INFO("FLASH数据写入完成\r\n");

38 for (i = 0; i < length; i++) {

39 g\_rx\_buf[i] = 0;

40 }

41 spi\_flash\_read(&spi\_dev, TEST\_ADDR, length); // 从设定的SPI\_FLASH地址中读取数据

42 am\_mdelay(10);

43 for (i = 0; i < length; i++) { // 数据校验

44 AM\_DBG\_INFO(" read %2dst data is : 0x%2x \r\n", i, g\_rx\_buf[i]);

45 if(g\_rx\_buf[i] != ((1 + i) & 0xFF)) {

46 AM\_DBG\_INFO("verify failed!\r\n");

47 while(1);

48 }

49 }

50 }

## ADC外设应用详解

#### ADC初始化

AMetal平台提供了ADC初始化函数，可以直接调用初始化函数。函数原型为：

am\_adc\_handle\_t am\_zlg116\_adc\_inst\_init (void);

该函数在user\_config目录下的am\_hwconf\_zlg116\_adc.c文件中定义，在am\_zlg116\_ inst\_init.h中声明。因此使用ADC初始化函数时，需要包含头文件am\_zlg116\_inst\_init.h。

调用该函数初始化ADC时，需要定义一个am\_adc\_handle\_t类型的变量，用于保存获取的ADC服务句柄，ADC初始化程序为：

am\_adc\_handle\_t adc\_handle;

adc\_handle = am\_zlg116\_adc\_inst\_init ();

使用前，一般需要修改引脚初始化函数；以ADC通道0为例，ADC\_IN0为PIOA\_0的复用功能，需把ADC平台初始化函数的更改为：

static void \_\_zlg\_plfm\_adc\_init (void)

{

am\_gpio\_pin\_cfg(PIOA\_0, PIOA\_0\_ADC\_IN0 | PIOA\_0\_AIN);

am\_clk\_enable(CLK\_ADC1);

}

把解除ADC平台初始化函数的更改为：

static void \_\_zlg\_plfm\_adc\_deinit (void)

{

am\_gpio\_pin\_cfg(PIOA\_0, PIOA\_0\_INPUT\_FLOAT);

am\_clk\_disable (CLK\_ADC1);

}

如需修改ADC的参考电压和转换精度等信息，可以直接调用宏。转换精度的宏定义详见表4.11。

表4.11 转换精度宏定义

|  |  |
| --- | --- |
| 宏名 | 转换精度 |
| AMHW\_ZLG\_ADC\_DATA\_VALID\_8BIT | 8位 |
| AMHW\_ZLG\_ADC\_DATA\_VALID\_9BIT | 9位 |
| AMHW\_ZLG\_ADC\_DATA\_VALID\_10BIT | 10位 |
| AMHW\_ZLG\_ADC\_DATA\_VALID\_11BIT | 11位 |
| AMHW\_ZLG\_ADC\_DATA\_VALID\_12BIT | 12位 |

#### ADC读取电压

在使用过程中，我们需要通过多次采样然后取平均值来减小误差，读取ADC采样值的函数原型为：

int am\_adc\_read (am\_adc\_handle\_t handle,

int chan,

void \*p\_val,

uint32\_t length);

* handle为ADC的服务句柄；
* chan为ADC的通道编号；
* p\_val指向保存采样值的数组；
* length代表着采样的次数。

采样完成后，我们需把p\_val指向的数组里的值加起来，除以length，来得到多次采样的平均值：

for (sum = 0, i = 0; i < length; i++) {

sum += p\_val [i];

}

sum /= length;

得到平均值后，可用AM\_ADC\_VAL\_TO\_MV()将它装换为电压值：

adc\_mv = AM\_ADC\_VAL\_TO\_MV(handle, chan, adc\_code);

其中adc\_mv为保存电压值的变量；handle为ADC的服务句柄；chan为ADC的通道编号；adc\_code 为多次采样额平均值。

#### ADC范例

每隔500ms读取ADC通道0的电压值并通过串口打印的范例详见程序清单4.16。

程序清单4.16 ADC范例

1 #include "ametal.h"

2 #include "am\_vdebug.h"

3 #include "am\_delay.h"

4 #include "am\_adc.h"

5 #include "am\_zlg116\_inst\_init.h"

6

7 static uint32\_t \_\_adc\_code\_get (am\_adc\_handle\_t handle, int chan) //获取 ADC 转换值

8 {

9 int i;

10 uint32\_t sum;

11 uint16\_t val\_buf[12];

12 am\_adc\_read(handle, chan, val\_buf, 12);

13 for (sum = 0, i = 0; i < 12; i++) { // 均值处理

14 sum += val\_buf[i];

15 }

16 return (sum / 12);

17 }

18

19 void am\_main ()

20 {

21 am\_adc\_handle\_t handle = am\_zlg116\_adc\_inst\_init();

22 uint32\_t adc\_code; // 采样 Code 值

23 uint32\_t adc\_mv; // 采样电压

24 while (1) {

25 adc\_code = \_\_adc\_code\_get(handle, 0);

26 adc\_mv = AM\_ADC\_VAL\_TO\_MV(handle, 0, adc\_code);

27 /\* 串口输出采样电压值 \*/

28 am\_kprintf("Sample : %d, Vol: %d mv\r\n", adc\_code, adc\_mv);

29 am\_mdelay(500);

30 }

31 }

## 定时器外设应用详解

定时器外设功能丰富，可用于定时、输出PWM、输入捕获多种应用场合。为了快速使用AMetal平台的定时器程序，从定时、输出PWM、输入捕获三个功能详细介绍定时器的使用。

### Timer外设应用详解

定时是定时器的基本功能，以定时器1定时500ms为例，每500ms产生一个中断，在中断服务函数中翻转连接LED的I/O引脚，使LED不停闪烁，并通过串口打印信息。

#### 定时器初始化

AMetal平台提供了定时器初始化函数，可以直接调用初始化函数。以定时器1初始化函数为例，函数原型为：

am\_timer\_handle\_t am\_zlg116\_tim1\_timing\_inst\_init (void);

该函数在user\_config目录下的am\_hwconf\_zlg116\_tim\_timing.c文件中定义，在am\_zlg 116\_inst\_init.h中声明。因此使用定时器初始化函数时，需要包含头文件am\_zlg116\_inst\_init.h。

初始化定时器1，调用该函数时需要定义一个am\_timer\_handle\_t类型的变量，用于保存获取的定时器1服务句柄，定时器初始化程序为：

am\_timer\_handle\_t tim1\_timing\_handle;

tim1\_timing\_handle = am\_zlg116\_tim1\_timing\_inst\_init();

#### 获取定时器信息

定时器的定时时间与时钟频率、分频系数、计数器位数有关。为了准确配置定时器的定时时间，需要知道定时器的时钟频率、可配置的分频系数、计数器位数。AMetal平台已提供获取定时器时钟频率、信息的函数，在am\_timer.h中以内敛函数的形式定义，am\_timer.h可在service目录下的am\_timer.c包含的头文件中找到。

获取定时器信息，主要获取时钟频率、可配置的分频系数值、定时器位数。以定时器1为例，获取定时器1的时钟频率，函数原型为：

am\_static\_inline int am\_timer\_clkin\_freq\_get (am\_timer\_handle\_t handle, uint32\_t \*p\_freq);

* handle为定时器的服务句柄，即为初始化定时器1获取的句柄；
* p\_freq为指向时钟频率的数据指针，即传入一个内存地址。

基于以上信息，获取时钟频率的程序为：

uint32\_t clkin\_freq = 0;

am\_timer\_clkin\_freq\_get(tim1\_timing\_handle, &clkin\_freq);

获取定时器信息的函数原型为：

am\_static\_inline const am\_timer\_info\_t \* am\_timer\_info\_get (am\_timer\_handle\_t handle);

* handle为定时器的服务句柄，即为初始化定时器1获取的句柄；
* 返回值为am\_timer\_info\_t类型的指针，因此需要定义一个am\_timer\_info\_t类型的指针，用于保存返回值。

基于以上信息，获取定时器1信息的程序为：

const am\_timer\_info\_t \*p\_tim1\_timing\_info;

p\_tim1\_timing\_info = am\_timer\_info\_get(tim1\_timing\_handle);

获取的信息，可以通过调试或用串口打印看到，进而根据定时器时钟频率和信息配置定时器。以串口打印获取的信息为例，程序详见程序清单4.17。通过串口信息，可以清楚的知道定时器的相关信息，为后面定时器的配置做了准备。

程序清单4.17 打印定时器信息

1 AM\_DBG\_INFO("clk\_freq = %d \r\n",clkin\_freq);

2 AM\_DBG\_INFO("width = %d \r\n",p\_tim1\_timing\_info->counter\_width);

3 AM\_DBG\_INFO("chan = %d \r\n",p\_tim1\_timing\_info->chan\_num);

4 AM\_DBG\_INFO("prescaler = %d \r\n",p\_tim1\_timing\_info->prescaler);

#### 编写回调函数

当定时时间到，调用回调函数，定时执行回调函数。回调函数与普通函数一样，程序详见程序清单4.18。

程序清单4.18 定时器1回调函数

1 void tim1\_timing\_callback(void \*p\_arg)

2 {

3 AM\_DBG\_INFO("Timing time!\r\n");

4 am\_gpio\_toggle(PIOB\_1);

5 }

编写的定时回调函数需要与定时器关联起来，需要设置回调函数。设置回调函数原型为：

am\_static\_inline int am\_timer\_callback\_set (

am\_timer\_handle\_t handle,

uint8\_t chan,

void (\*pfn\_callback)(void \*),

void \*p\_arg); http://www.keil.com/dd2/pack/

* handle为定时器的服务句柄，即为初始化定时器1获取的句柄；
* chan为使用的定时器通道，定时器1只有一个通道；
* (\*pfn\_callback)(void \*)为回调函数，即将定义的回调函数名传入；
* p\_arg为回调函数的参数。

基于以上信息，设置回调函数的程序为：

am\_timer\_callback\_set(tim1\_timing\_handle, 0, tim1\_timing\_callback, NULL);

#### 配置定时时间

通过获取定时器时钟频率、定时器信息，可知时钟频率为48MHz，计数器宽度为16位，最大计数值为65535，分频器最大值为65536。AMetal提供了定时器分频值设置函数、定时计数值设置函数，函数均定义在am\_timer.h中。

分频值设置函数在原型为：

am\_static\_inline int am\_timer\_prescale\_set (

am\_timer\_handle\_t handle,

uint8\_t chan,

uint32\_t prescale);

* handle为定时器的服务句柄，即为初始化定时器1获取的句柄；
* chan为使用的定时器通道；
* prescale为定时器时钟频率分频值。

定时计数值设置函数原型为：

am\_static\_inline int am\_timer\_enable (am\_timer\_handle\_t handle, uint8\_t chan, uint32\_t count);

* handle为定时器的服务句柄，即为初始化定时器1获取的句柄；
* chan为使用的定时器通道；
* count为设置的最大计数值，计数器计数到该数值，调用回调函数，计数值清零重新计数。

基于以上信息，设置分频系数为48000，计数值为500，即定时时间为500ms，具体程序为：

am\_timer\_prescale\_set (tim1\_timing\_handle, 0, 48000);

am\_timer\_enable(tim1\_timing\_handle, 0, 500);

#### 定时范例程序

定时500ms，在回调函数中翻转LED，串口打印信息的程序详见程序清单4.19。

程序清单4.19 定时器定时范例程序

1 #include "ametal.h"

2 #include "am\_board.h"

3 #include "am\_vdebug.h"

4 #include "am\_delay.h"

5 #include "am\_led.h"

6 #include "am\_gpio.h"

7 #include "zlg116\_pin.h"

8 #include "am\_zlg116\_inst\_init.h"

9 void tim1\_timing\_callback(void \*p\_arg)

10 {

11 AM\_DBG\_INFO("Timing time!\r\n");

12 am\_gpio\_toggle(PIOB\_1);

13 }

14 int am\_main (void)

15 {

16 am\_timer\_handle\_t tim1\_timing\_handle; // 定时器服务句柄

17 const am\_timer\_info\_t \*p\_tim1\_timing\_info; // 定时器信息

18 uint32\_t clkin\_freq = 0; // 定时器时钟频率

19 AM\_DBG\_INFO("Start up successful!\r\n");

20 // 初始化LED GPIO

21 am\_gpio\_pin\_cfg (PIOB\_1, AM\_GPIO\_OUTPUT\_INIT\_HIGH);

22 // 初始化定时器

23 tim1\_timing\_handle = am\_zlg116\_tim1\_timing\_inst\_init();

24 // 获取定时器时钟频率

25 am\_timer\_clkin\_freq\_get(tim1\_timing\_handle, &clkin\_freq);

26 AM\_DBG\_INFO("clk\_freq = %d \r\n",clkin\_freq);

27 // 获取定时器信息

28 p\_tim1\_timing\_info = am\_timer\_info\_get(tim1\_timing\_handle);

29 AM\_DBG\_INFO("width = %d \r\n",p\_tim1\_timing\_info->counter\_width);

30 AM\_DBG\_INFO("chan\_num = %d \r\n",p\_tim1\_timing\_info->chan\_num);

31 AM\_DBG\_INFO("prescaler = %d \r\n",p\_tim1\_timing\_info->prescaler);

32 // 设置回调函数

33 am\_timer\_callback\_set(tim1\_timing\_handle, 0, tim1\_timing\_callback, NULL);

34 // 设置时钟分频系数

35 am\_timer\_prescale\_set (tim1\_timing\_handle, 0, 48000);

36 // 设置定时中断频率为2Hz

37 am\_timer\_enable(tim1\_timing\_handle, 0, 500);

38 while (1) {

39 }

40 }

### PWM外设应用详解

PWM是常用的功能，常用于驱动无源蜂鸣器、亮度调节、电机调速、开关控制等。以配置周期为500ms、占空比为50%的PWM为例，控制LED闪烁。

#### 定时器PWM初始化

AMetal平台提供了定时器的PWM初始化函数，可以直接调用初始化函数。ZLG116核心板AM116-CORE上的LED可通过跳线帽连接到PB1引脚，PB1可复用为TIM3\_CH4。因此以定时器3 PWM初始化函数为例，函数原型为：

am\_pwm\_handle\_t am\_zlg116\_tim3\_pwm\_inst\_init (void);

该函数在user\_config目录下的am\_hwconf\_zlg116\_tim\_pwm.c文件中定义，在am\_zlg116 \_inst\_init.h中声明。因此使用定时器3的PWM初始化函数时，需要包含头文件am\_zlg116\_inst\_init.h。

初始化定时器3 PWM，调用该函数时需要定义一个am\_pwm\_handle\_t类型的变量，用于保存获取的定时器3 PWM服务句柄，定时器3 PWM初始化程序为：

am\_pwm\_handle\_t pwm\_handle;

pwm\_handle = am\_zlg116\_tim3\_pwm\_inst\_init();

#### PWM参数配置

PWM参数包括频率、占空比、通道。AMetal平台提供了PWM参数配置的函数，可以直接使用。该函数以内敛函数的形式定义在am\_pwm.h中，am\_pwm.h可以在soc目录下am \_zlg\_tim\_pwm.c包含的头文件中找到。其函数原型为：

am\_static\_inline int am\_pwm\_config (

am\_pwm\_handle\_t handle,

int chan,

unsigned long duty\_ns,

unsigned long period\_ns);

* handle为PWM的服务句柄，传入的实参为PWM初始化获取的服务句柄；
* chan为使用的PWM通道，通道号从0开始，因此TIM3\_CH4的通道号为3；
* duty\_ns为PWM脉宽时间，单位为ns，以PWM周期500ms为例，50%占空比时的脉宽时间为250ms，传入的是实参为250000000；
* period\_ns为PWM周期时间，单位ns，以PWM周期500ms为例，传入的实参为500000000。

根据以上信息，配置PWM周期为500ms，占空比为50%的程序为：

am\_pwm\_config(pwm\_handle, 3, 250000000, 500000000);

#### PWM使能

PWM初始化、配置参数、使能后即可从对应I/O引脚输出PWM波形。AMetal平台提高了PWM使能、禁能函数，函数定义am\_pwm.h中。PWM使能函数原型为：

am\_static\_inline int am\_pwm\_enable (am\_pwm\_handle\_t handle, int chan);

* handle为PWM的服务句柄，传入的实参为PWM初始化获取的服务句柄；
* chan为使用的PWM通道，即PWM配置参数时配置的通道；

基于以上信息，PWM使能程序为：

am\_pwm\_enable(pwm\_handle, 3);

#### PWM范例程序

以定时器3 的PWM通道4为例，从PB1引脚输出周期为500ms、50%占空比的PWM，控制LED闪烁，程序详见程序清单4.20。编译下载，可以看到LED0不断闪烁。

程序清单4.20 PWM范例程序

1 #include "ametal.h"

2 #include "am\_board.h"

3 #include "am\_vdebug.h"

4 #include "am\_delay.h"

5 #include "am\_led.h"

6 #include "am\_zlg116\_inst\_init.h"

7 int am\_main (void)

8 {

9 am\_pwm\_handle\_t pwm\_handle;

10 AM\_DBG\_INFO("Start up successful!\r\n");

11 pwm\_handle = am\_zlg116\_tim3\_pwm\_inst\_init();

12 am\_pwm\_config(pwm\_handle, 3, 250000000, 500000000);

13 am\_pwm\_enable(pwm\_handle, 3);

14 while (1) {

15 }

16 }

### CAP外设详解

定时器输入捕获可用于测量脉冲宽度、脉冲频率。以ZLG116定时器2的TIM2\_CH2通道为例，捕获输入的脉冲周期。

#### 输入捕获初始化

AMetal平台提供了定时器的输入捕获初始化函数，可以直接调用初始化函数。以定时器2的输入捕获初始化函数为例，函数原型为：

am\_cap\_handle\_t am\_zlg116\_tim2\_cap\_inst\_init (void);

该函数在user\_config目录下的am\_hwconf\_zlg116\_tim\_cap.c文件中定义，在am\_zlg116 \_inst\_init.h中声明。因此使用定时器2的输入捕获初始化函数时，需要包含头文件am\_zlg116\_inst\_init.h。

初始化定时器2输入捕获，调用该函数时需要定义一个am\_cap\_handle\_t类型的变量，用于保存获取的定时器2输入捕获的服务句柄，定时器2出入捕获初始化程序为：

am\_cap\_handle\_t cap\_handle;

cap\_handle = am\_zlg116\_tim2\_cap\_inst\_init();

#### 编写回调函数

当捕获事件发生时，调用回调函数，执行用户任务程序。以测量输入脉冲的周期为例，编写的回调函数详见程序清单4.21。其中形式参数count为计数器的值。

程序清单4.21 捕获事件回调函数

1 void cap\_callback (void \*p\_arg, unsigned int count)

2 {

3 static uint8\_t times = 0; // 标记事件发生次数

4 if (times == 0) { // 第一次上升沿事件

5 g\_time\_ns = count;

6 times = 1;

7 } else { // 第二次上升沿事件

8 if (count > g\_time\_ns) { // 计数器没有溢出

9 g\_time\_ns = count - g\_time\_ns;

10 } else { // 计数器溢出

11 g\_time\_ns = count + (0xffffffff - g\_time\_ns);

12 }

13 times = 0;

14 g\_cap\_flag = 1;

15 }

16 }

#### 输入捕获配置

输入捕获可选择捕获通道、捕获事件的类型，可设置捕获事件回调函数。AMetal平台提供了输入捕获配置函数，该函数定义在am\_cap.h中，am\_cap.h包含在soc目录下的am\_zlg\_tim\_cap.c的头文件。

输入捕获配置函数原型为：

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);

* handle为输入捕获的服务句柄，传入实参为初始化输入捕获时获取的句柄cap\_handle；
* chan为使用的输入捕获通道，通道号从0开始，使用TIM2\_CH2，通道为1；
* options为输入捕获事件类型，可配置为上升沿、下降沿、双边沿，可以值已在am\_cap.h中使用宏定义，详见表 4.12。以上升沿触发捕获为例，实参为AM\_CAP\_ TRIGGER\_RISE；
* pfn\_callback为输入捕获回调函数，即为回调函数名cap\_callback；
* p\_arg为回调函数参数，无参数即为NULL。

表 4.12 输入捕获事件类型

|  |  |
| --- | --- |
| 宏名 | 功能说明 |
| AM\_CAP\_TRIGGER\_RISE | 上升沿触发捕获 |
| AM\_CAP\_TRIGGER\_FALL | 下降沿触发捕获 |
| AM\_CAP\_TRIGGER\_BOTH\_EDGES | 双边沿触发捕获 |

基于以上信息，输入捕获配置程序为：

am\_cap\_config (cap\_handle, 1, AM\_CAP\_TRIGGER\_RISE, cap\_callback, NULL);

#### 使能输入捕获

配置完捕获通道、捕获事件类型、捕获回调函数，使能捕获通道后，当发生事件便会执行回调函数。使能捕获通道函数定义在am\_cap.h中，函数原型为：

am\_static\_inline int am\_cap\_enable (am\_cap\_handle\_t handle, int chan);

* handle为输入捕获的服务句柄，传入实参为初始化输入捕获时获取的句柄cap\_handle；
* chan为初始化的输入捕获通道，通道为1；

基于以上信息，使能输入捕获的程序为：

am\_cap\_enable (cap\_handle, 1);

#### 输入捕获范例

将PA1配置为定时器2的TIM2\_CH2输入捕获，测量从PB1引脚输出定时器3 的PWM的周期，范例程序详见程序清单4.22。短接PA1、PB2引脚，可以看到测量脉冲周期与配置的PWM周期一致。

程序清单4.22 输入捕获范例程序

1 #include "ametal.h"

2 #include "am\_board.h"

3 #include "am\_vdebug.h"

4 #include "am\_delay.h"

5 #include "am\_zlg116\_inst\_init.h"

6 uint32\_t g\_time\_ns = 0;

7 uint8\_t g\_cap\_flag = 0;

8 void cap\_callback (void \*p\_arg, unsigned int count)

9 {

10 static uint8\_t times = 0; // 标记事件发生次数

11 if (times == 0) { // 第一次上升沿事件

12 g\_time\_ns = count;

13 times = 1;

14 } else { // 第二次上升沿事件

15 if (count > g\_time\_ns) { // 计数器没有溢出

16 g\_time\_ns = count - g\_time\_ns;

17 } else { // 计数器溢出

18 g\_time\_ns = count + (0xffffffff - g\_time\_ns);

19 }

20 times = 0;

21 g\_cap\_flag = 1;

22 }

23 }

24 int am\_main (void)

25 {

26 am\_pwm\_handle\_t pwm\_handle;

27 am\_cap\_handle\_t cap\_handle;

28 AM\_DBG\_INFO("Start up successful!\r\n");

29 pwm\_handle = am\_zlg116\_tim3\_pwm\_inst\_init();

30 am\_pwm\_config(pwm\_handle, 3, 5000000, 10000000);

31 am\_pwm\_enable(pwm\_handle, 3);

32 cap\_handle = am\_zlg116\_tim2\_cap\_inst\_init();

33 am\_cap\_config (cap\_handle, 1, AM\_CAP\_TRIGGER\_RISE, cap\_callback, NULL);

34 am\_cap\_enable (cap\_handle, 1);

35 while (1) {

36 if (g\_cap\_flag) {

37 g\_cap\_flag = 0;

38 //将计数值换算成时间

39 am\_cap\_count\_to\_time(cap\_handle, 1, 0, g\_time\_ns, &g\_time\_ns);

40 AM\_DBG\_INFO("PWM\_period = %d ns\r\n",g\_time\_ns);

41 }

42 }

43 }

# AMetal组件详解

## 矩阵键盘组件

矩阵按键可以提高I/O的使用效率，但是要区分和判断按键动作的方法却比较复杂。每次扫描一行，扫描该行时，对应行线输出为低电平，其余行线输出为高电平，然后读取所有列线的电平，若有列线读到低电平，则表明该行与读到低电平的列对应的交叉点有按键按下。逐列扫描法恰好相反，其列线为输出，行线为输入，但基本原理还是一样的。

本节以8\*6矩阵键盘为例，详解该组件，矩阵键盘电路如图5.1所示。

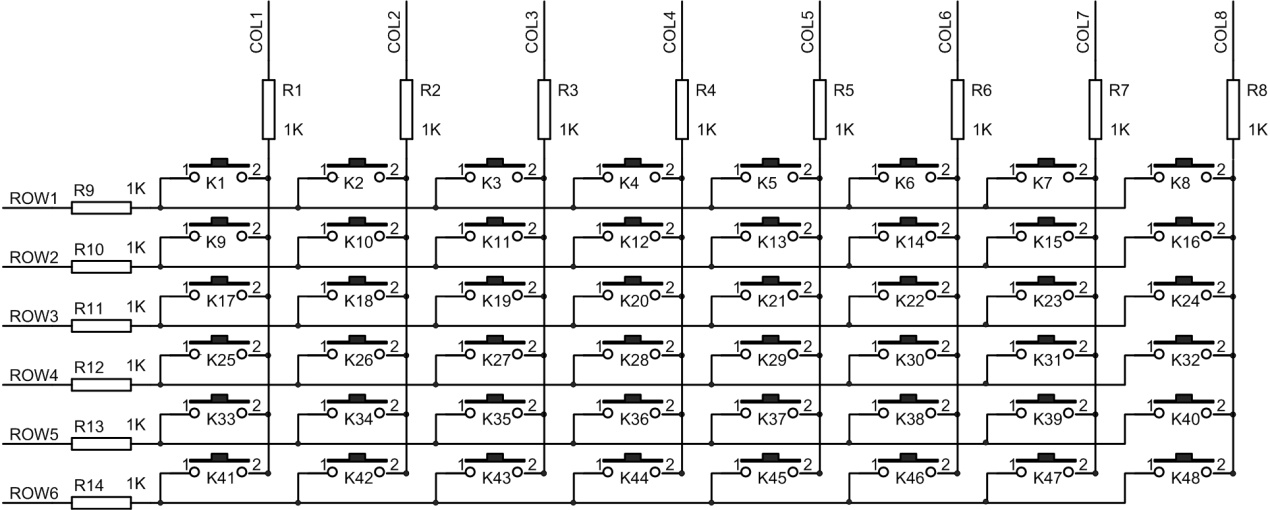


图5.1 矩阵键盘电路

AMetal已经集成了矩阵按键组件软件，只需配置与矩阵键盘相关的信息即可，操作非常简单。在am\_key\_matrix\_gpio.h文件中，定义了与矩阵键盘相关配置函数结构体，详见程序清单5.1。

程序清单5.1 矩阵键盘配置信息定义

typedef struct am\_key\_matrix\_gpio\_info {

am\_key\_matrix\_base\_info\_t base\_info; // 矩阵键盘基础信息

const int \*p\_pins\_row; // 行线引脚

const int \*p\_pins\_col; // 列线引脚

} am\_key\_matrix\_gpio\_info\_t;

/\*\*

\* \brief 矩阵键盘基础信息

\*/

typedef struct am\_key\_matrix\_base\_info {

int row; // 行数目

int col; // 列数目

const int \*p\_codes; // 各个按键对应的编码，按行的顺序依次对应

am\_bool\_t active\_low; // 按键按下后是否为低电平

uint8\_t scan\_mode; // 扫描方式 （按行扫描或按列扫描）

} am\_key\_matrix\_base\_info\_t;

在am\_key\_matrix\_gpio\_info成员中包含了GPIO驱动矩阵键盘的所有信息，包含了矩阵键盘的基础信息，如矩阵键盘的行数和列数、各按键对应的编码、按键扫描时间及扫描方式等，在am\_key\_matrix\_gpio.c文件中进行赋值，对应信息配置如下：

* \_\_g\_key\_pins\_row指向存放矩阵键盘行线对应引脚号的数组，在此填入行引脚；

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

按键GPIO行线引脚

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static const int \_\_g\_key\_pins\_row[] = {PIOB\_5, PIOB\_4, PIOB\_3,

PIOB\_2, PIOB\_1, PIOB\_0};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

* \_\_g\_key\_pins\_col指向存放矩阵键盘列线对应引脚号的数组，在此填入列引脚；

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

按键GPIO列线引脚

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static const int \_\_g\_key\_pins\_col[] = {PIOA\_15, PIOA\_14, PIOA\_13, PIOA\_12,

PIOA\_11, PIOA\_10, PIOA\_09, PIOA\_08};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

* \_\_g\_key\_codes指向按键编码数组，指定了各按键对应的编码，在此填入按键编码；

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

按编码信息

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static const int \_\_g\_key\_codes[] = {

KEY\_00, KEY\_01,KEY\_02, KEY\_03,KEY\_04, KEY\_05,KEY\_06, KEY\_07,

KEY\_10, KEY\_11,KEY\_12, KEY\_13,KEY\_14, KEY\_15,KEY\_16, KEY\_17,

KEY\_20, KEY\_21,KEY\_22, KEY\_23,KEY\_24, KEY\_25,KEY\_26, KEY\_27,

KEY\_30, KEY\_31,KEY\_32, KEY\_33,KEY\_34, KEY\_35,KEY\_36, KEY\_37,

KEY\_40, KEY\_41,KEY\_42, KEY\_43,KEY\_44, KEY\_45,KEY\_46, KEY\_47,

KEY\_50, KEY\_51,KEY\_52, KEY\_53,KEY\_54, KEY\_55,KEY\_56, KEY\_57,

};

* scan\_interval\_ms指定了按键扫描的时间间隔（单位：毫秒），即每隔该段时间执行一次按键检测，检测是否有按键事件发生（按键按下或按键释放），该值一般设置为5 ms，在结构体中直接赋值即可，如程序清单5.2所示；

程序清单5.2 按键实例化函数

1 int am\_miniport\_key\_inst\_init (void)

2 {

3 static am\_key\_matrix\_gpio\_softimer\_t miniport\_key;

4 static const am\_key\_matrix\_gpio\_softimer\_info\_t miniport\_key\_info = {

5 {

6 {

7 6, // 6行按键

8 8, // 8列按键

9 \_\_g\_key\_codes, // 各按键对应的编码

10 AM\_TRUE, // 按键低电平视为按下

11 AM\_KEY\_MATRIX\_SCAN\_MODE\_COL, // 扫描方式，按列扫描

12 },

13 \_\_g\_key\_pins\_row,

14 \_\_g\_key\_pins\_col,

15 },

16 5, // 扫描时间间隔，5ms

17 };

18 return am\_key\_matrix\_gpio\_softimer\_init(&miniport\_key, &miniport\_key\_info);

19 }

## UART缓冲接口

UART通信时，由于查询模式会阻塞整个应用，因此在实际应用中几乎都使用中断模式。但在中断模式下，UART每收到一个数据都会调用回调函数，如果将数据的处理放在回调函数中，很有可能因当前数据的处理还未结束而丢失下一个数据。

基于此，AMetal提供了一组带缓冲区的UART通用接口，详见表5.1，其实现是在UART中断接收与应用程序之间，增加一个接收缓冲区。当串口收到数据时，将数据存放在缓冲区中，应用程序直接访问缓冲区即可。

对于UART发送，虽然不存在丢失数据的问题，但为了便于开发应用程序，避免在UART中断模式下的回调函数接口中一次发送单个数据，同样提供了带缓冲区的UART发送函数。当应用程序发送数据时，将发送数据存放在发送缓冲区中，串口在发送空闲时提取发送缓冲区中的数据进行发送。

表5.1 带缓冲区的UART通用接口函数（am\_uart\_rngbuf.h）

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| am\_uart\_rngbuf\_handle\_t am\_uart\_rngbuf\_init(  am\_uart\_rngbuf\_dev\_t \*p\_dev,  am\_uart\_handle\_t handle,  uint8\_t \*p\_rxbuf,  uint32\_t rxbuf\_size,  uint8\_t \*p\_txbuf,  uint32\_t txbuf\_size); | 初始化 |
| int am\_uart\_rngbuf\_send(  am\_uart\_rngbuf\_handle\_t handle,  const uint8\_t \*p\_txbuf,  uint32\_t nbytes); | 发送数据 |
| int am\_uart\_rngbuf\_receive(  am\_uart\_rngbuf\_handle\_t handle,  uint8\_t \*p\_rxbuf,  uint32\_t nbytes); | 接收数据 |
| int am\_uart\_rngbuf\_ioctl(  am\_uart\_rngbuf\_handle\_t handle,  int request,  void \*p\_arg); | 控制函数 |

#### 初始化

指定关联的串口外设（相应串口的实例句柄handle），以及用于发送和接收的数据缓冲区，初始化一个带缓冲区的串口实例，其函数原型如程序清单5.3所示。

程序清单5.3 串口初始化函数原型

am\_uart\_rngbuf\_handle\_t am\_uart\_rngbuf\_init(

am\_uart\_rngbuf\_dev\_t \*p\_dev, // 带缓冲区的UART设备

am\_uart\_handle\_t handle, // UART实例句柄handle

char \*p\_rxbuf, // 接收数据缓冲区

uint32\_t rxbuf\_size, // 接收数据缓冲区的大小

char \*p\_txbuf, // 发送数据缓冲区

uint32\_t txbuf\_size); // 发送数据缓冲区的大小

其中， p\_dev为指向am\_uart\_rngbuf\_dev\_t类型的带缓冲区的串口实例指针，在使用时，只需要定义一个am\_uart\_rngbuf\_dev\_t类型（am\_uart\_rngbuf.h）的实例即可：

am\_uart\_rngbuf\_dev\_t g\_uart0\_rngbuf\_dev;

其中，g\_uart0\_rngbuf\_dev为用户自定义的实例，其地址作为p\_dev的实参传递。handle为 UART实例句柄，用于指定该带缓冲区的串口实际关联的串口。p\_rxbuf和rxbuf\_size用于指定接收缓冲区及其大小，p\_txbuf和txbuf\_size用于指定发送缓冲区及其大小。

函数的返回值为带缓冲区串口的实例句柄，可用作其它通用接口函数中handle参数的实参。其类型am\_uart\_rngbuf\_handle\_t（am\_uart\_rngbuf.h）定义如下：

typedef struct am\_uart\_rngbuf\_dev \* am\_uart\_rngbuf\_handle\_t;

如果返回值为NULL，表明初始化失败，初始化函数使用范例详见程序清单5.4。

程序清单5.4 am\_uart\_rngbuf\_init()范例程序

1 static uint8\_t uart\_rxbuf[128]; // 定义用于接收数据的缓冲区，大小为128

2 static uint8\_t uart\_txbuf[128]; // 定义用于发送数据的缓冲区，大小为128

3 am\_uart\_rngbuf\_dev\_t g\_uart\_rngbuf\_dev;

4 am\_uart\_rngbuf\_handle\_t g\_uart\_rngbuf\_handle;

5

6 g\_uart\_rngbuf\_handle = am\_uart\_rngbuf\_init(

7 &g\_uart\_rngbuf\_dev,

8 uart\_handle, // UART实例句柄handle

9 uart\_rxbuf, // 用于接收数据的缓冲区

10 128, // 接收缓冲区大小为128

11 uart\_txbuf, // 用于发送数据的缓冲区

12 128); // 发送缓冲区大小为128

虽然程序将缓冲区的大小设置为128，但实际上缓冲区的大小应根据实际情况确定。若接收数据的缓冲区过小，则可能在接收缓冲区满后又接收新的数据发生溢出而丢失数据。若发送缓冲区过大，则在发送数据时很可能因为发送缓冲区已满需要等待，直至发送缓冲区有空闲空间而造成等待过程。

#### 发送数据

发送数据就是将数据存放到am\_uart\_rngbuf\_init()指定的发送缓冲区中，串口可以进行数据发送时（发送空闲），从发送缓冲区中提取需要发送的数据进行发送。其函数原型详见程序清单5.5。

程序清单5.5 发送函数原型

int am\_uart\_rngbuf\_send(

am\_uart\_rngbuf\_handle\_t handle, // 带缓冲区的串口实例句柄

const uint8\_t \*p\_txbuf, // 应用程序发送数据缓冲区

uint32\_t nbytes); // 发送数据的个数

该函数将数据成功存放到发送缓冲区后返回，返回值为成功写入的数据个数。比如，发送一个字符串“Hello World!”，详见程序清单5.6。

程序清单5.6 am\_uart\_rngbuf\_send()范例程序

1 uint8\_t str[] = "Hello World!";

2 am\_uart\_rngbuf\_send(g\_uart0\_rngbuf\_handle, str, sizeof(str)); // 发送字符串"Hello World!"

注意，当该函数返回时，数据仅仅只是存放到了发送缓冲区中，并不代表已经成功地将数据发送出去了。

#### 接收数据

接收数据就是从am\_uart\_rngbuf\_init()指定的接收缓冲区中提取接收到的数据，其函数原型详见程序清单5.7。

程序清单5.7 接收函数

int am\_uart\_rngbuf\_receive(

am\_uart\_rngbuf\_handle\_t handle, // 带缓冲区的串口实例句柄

uint8\_t \*p\_rxbuf, // 应用程序接收数据缓冲区

uint32\_t nbytes); // 接收数据的个数

该函数返回值为成功读取数据的个数，使用范例详见程序清单5.8。

程序清单5.8 am\_uart\_rngbuf\_receive()范例程序

1 uint8\_t rxbuf[10];

2 am\_uart\_rngbuf\_receive(g\_uart0\_rngbuf\_handle, rxbuf,10);// 接收10个数据

#### 控制函数

与UART控制函数类似，用于完成一些基本的控制操作。其函数原型详见程序清单5.9。

程序清单5.9 控制函数

int am\_uart\_rngbuf\_ioctl(

am\_uart\_rngbuf\_handle\_t handle, // 带缓冲区的串口实例句柄

int request, // 控制命令

void \*p\_arg); // 对应命令的参数

“控制命令”和“对应命令的参数”，与UART控制函数am\_uart\_ioctl()的含义类似。带缓冲区的UART可以看作是在UART基础上的一个扩展，因此绝大部分UART控制函数的命令均可直接使用。

## E2PROM组件

E²PROM（Electrically Erasable Programable Read-Only Memory，电可擦除可编程只读存储器）是一种掉电后数据不丢失的存储芯片，本节以FM24C02为例详细介绍在AMetal中如何使用类似的非易失存储器。

### 器件介绍

FM24C02总容量为2K（2048）bits,即256（2048/8）字节。每个字节对应一个存储地址，因此其存储数据的地址范围为：0x00~0xFF。FM24C02页（page）的大小为8字节，每次写入数据不能超过页边界，即地址0x08、0x10、0x18……。如果写入数据越过页边界时，则必须分多次写入，其组织结构详见表5.2。

表5.2 FM24C02存储器组织结构

|  |  |
| --- | --- |
| 页号 | 地址范围 |
| 0 | 0x00~0x07 |
| 1 | 0x08~0x0F |
| … | … |
| 30 | 0xF0~0xF7 |
| 31 | 0xF8~0xFF |

FM24C02的通信接口为标准的I²C接口，仅需SDA和SCL两根信号线。这里以8PIN SOIC封装为例，详见图5.2。其中的WP为写保护，当该引脚接高电平时，将阻止一切写入操作。一般来说，该引脚直接接地，以便芯片正常读写。

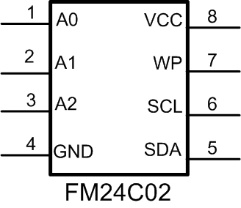


图5.2 FM24C02引脚定义

A2、A1、A0决定了FM24C02器件的I²C从机地址，其7-bit从机地址为0101 0A2A1A0。如果I²C总线上仅有一片FM24C02，则将A2、A1、A0直接接地，其地址为0x50。

在AMetal中，由于用户无需关心读/写方向位的控制，因此其地址使用7-bit地址表示。MicroPort-EEPROM模块通过MicroPort接口与AM116-Core相连，其中的E²PROM是复旦微半导体提供的256个字节FM24C02C。

### 初始化

AMetal提供了支持FM24C02、FM24C04、FM24C08……等系列I²C接口E²PROM的驱动函数，下面将以FM24C02为例予以说明，其函数原型（am\_ep24cxx.h）为：

am\_ep24cxx\_handle\_t am\_ep24cxx\_init(

am\_ep24cxx\_dev\_t \*p\_dev,

const am\_ep24cxx\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t i2c\_handle);

该函数意在获取器件实例句柄fm24c02\_handle，其中，p\_dev为指向am\_ep24cxx\_dev\_t类型实例的指针，p\_devinfo为指向am\_ep24cxx\_devinfo\_t类型实例信息的指针。

#### 实例

单个FM24C02可以看作EP24Cxx的一个实例，EP24Cxx只是抽象了代表一个系列或同种类型的E²PROM芯片，显然多个FM24C02是EP24Cxx的多个实例。如果I²C总线上只外接一个FM24C02，定义am\_ep24cxx\_dev\_t类型（am\_ep24cxx.h）实例如下：

am\_ep24cxx\_dev\_t g\_at24c02\_dev； //定义ep24cxx实例（FM24C02）

其中，g\_at24c02\_dev为用户自定义的实例，其地址作为p\_dev的实参传递。如果同一个I²C总线上外接了2个FM24C02，需要定义3个实例。即：

am\_ep24cxx\_dev\_t g\_24c02\_dev0; //FM24C02

am\_ep24cxx\_dev\_t g\_24c02\_dev1; //FM24C02

每个实例都要初始化，其每个实例的初始化均会返回一个该实例的handle。便于使用其它接口函数时，传递不同的handle操作不同的实例。

#### 实例信息

实例信息主要描述了具体器件固有的信息，即I²C器件的从机地址和具体型号，其类型am\_ep24cxx\_devinfo\_t的定义（am\_ep24cxx.h）如下：

typedef struct am\_ep24cxx\_devinfo{

uint8\_t slv\_addr; //器件7-bit从机地址

uint32\_t type; //器件型号

}am\_ep24cxx\_devinfo\_t;

当前已经支持的器件型号均在am\_ep24cxx.h中定义了对应的宏，比如，FM24C02对应的宏为AM\_EP24CXX\_FM24C2，实例信息定义如下：

const am\_ep24cxx\_devinfo\_t \_g\_24c02\_devinfo = {

0x50;

AM\_EP24CXX\_FM24C02

}

其中，g\_24c02\_devinfo为用户自定义的实例信息，其地址作为p\_devinfo的实参传递。

#### I²C句柄i²c\_handle

以I²C1为例，其实例初始化函数am\_lpc82x\_i2c1\_inst\_init()的返回值将作为实参传递给i2c\_handle。即：

i2c\_handle = am\_lpc82x\_i2c1\_inst\_init();

#### 实例句柄fm24c02\_handle

FM24C02初始化函数am\_ep24cxx\_init()的返回值fm24c02\_handle，作为实参传递给读写数据函数，其类型am\_ep24cxx\_handle\_t(am\_ep24cxx.h)定义如下：

typedef struct am\_ep24cxx\_dev \*am\_ep24cxx\_handle\_t;

若返回值为NULL，说明初始化失败；若返回值不为NULL，说明返回一个有效的handle。

基于模块化编程思想，将初始化相关的实例信息等的定义存放到对应的配置文件中，通过头文件引出实例初始化函数接口，源文件和头文件的程序范例分别详见程序清单5.10和程序清单5.11。

程序清单5.10 实例初始化函数范例程序（am\_hwconf\_ep24cxx.c）

1 #include "ametal.h"

2 #include "am\_ep24cxx.h"

3 #include "am\_lpc82x\_inst\_init.h"

4

5 static const am\_ep24cxx\_devinfo\_t \_\_g\_24c02\_devinfo = { //定义实例信息

6 0x50, //器件的I²C从机地址

7 AM\_EP24CXX\_FM24C02 //器件型号

8 }

9 static am\_ep24cxx\_dev\_t \_\_g\_24c02\_dev; //定义FM24C02器件实例

10

11 am\_ep24cxx\_handle\_t am\_fm24c02\_inst\_init(void)

12 {

13 am\_i2c\_handle\_t i2c\_handle = am\_lpc82x\_i2c1\_inst\_init();

14 return am\_ep24cxx\_init(&\_\_g\_24c02\_dev, &\_\_g\_24c02\_devinfo, i2c\_handle);

15 }

程序清单5.11 实例初始化函数接口（am\_hwconf\_ep24cxx.h）

1 #pragma once

2 #include "ametal.h"

3 #include "am\_ep24cxx.h"

4

5 am\_ep24cxx\_handle\_t am\_fm24c02\_inst\_init(void);

后续只需要使用无参数的实例初始化函数，即可获取到FM24C02的实例句柄。即：

am\_ep24cxx\_handle\_t fm24c02\_handle = am\_fm24c02\_inst\_init();

注意，i2c\_handle用于区分I²C0、I²C1、I²C2、I²C3，初始化函数返回值实例句柄用于区分同一系统中连接多个器件。

### 读写函数

读写EP24Cxx系列存储器的函数原型详见表5.3。

表5.3 ep24cxx读写函数（am\_ep24cxx.h）

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_ep24cxx\_write(  am\_ep24cxx\_handle\_t handle,  int start\_addr,  uint8\_t \*p\_buf,  int len); | 写入数据； |
| int am\_ep24cxx\_read(  am\_ep24cxx\_handle\_t handle,  int start\_addr,  uint8\_t \*p\_buf,  int len); | 读取数据； |

各API的返回值含义都是相同的：AM\_OK表示成功，负值表示失败，失败原因可根据具体的值查看am\_errno.h文件中相对应的宏定义。正值的含义由各API自行定义，无特殊说明时，表明不会返回正值。

#### 写入数据

从指定的地址开始写入一段数据的函数原型为：

int am\_ep24cxx\_write(

am\_ep24cxx\_handle\_t handle, //ep24cxx句柄

int start\_addr, //存储器起始数据地址

uint8\_t \*p\_buf, //待写入数据的缓冲区

int len); //写入数据的长度

如果返回值为AM\_OK，则说明写入成功，反之失败。假定从0x20地址开始，连续写入16字节，详见程序清单5.12。

程序清单5.12 写入数据范例程序

1 uint8\_t data[16] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

2 am\_ep24cxx\_write(fm24c02\_handle, 0x20, &data[0], 16);

#### 读取数据

从指定的起始地址开始读取一段数据的函数原型为：

int am\_ep24cxx\_read(

am\_ep24cxx\_handle\_t handle, //ep24cxx句柄

int start\_addr, //存储器起始数据地址

uint8\_t \*p\_buf, //存放读取数据的缓冲区

int len); //读取数据的长度

如果返回值为AM\_OK,则说明读取成功，反之失败。假定从0x20地址开始，连续读取16字节，详见程序清单5.13。

程序清单5.13 读取数据范例程序

1 uint8\_t data[16];

2 am\_ep24cxx\_read(fm24c02\_handle, 0x20, &data[0], 16);

如程序清单5.14所示为写入20个字节数据再读出来，然后比较是否相同的范例。

程序清单5.14 FM24C02读写范例程序

1 #include "ametal.h"

2 #include "am\_led.h"

3 #include "am\_delay.h"

4 #include "am\_lpc82x\_inst\_init.h"

5 #include "am\_ep24cxx.h"

6 #include "am\_hwconf\_ep24cxx.h"

7

8 int app\_test\_ep24cxx(am\_ep24cxx\_handle\_t handle)

9 {

10 int i;

11 uint8\_t data[20];

12

13 for(i=0;i<20;i++) //填充数据

14 data[i]=i;

15 am\_ep24cxx\_write(handle,0,&data[0],20); //从0地址开始，连续写入20字节数据

16 for(i=0;i<20;i++) //清零数据

17 data[i]=0;

18 am\_ep24cxx\_read(handle,0,&data[0],20); //从0地址开始，连续读出20字节数据

19 for(i=0;i<20;i++){ //比较数据

20 if(data[i]!=i)

21 return AM\_ERROR;

22 }

23 return AM\_OK;

24 }

25

26 int am\_main(void)

27 {

28 am\_ep24cxx\_handle\_t fm24c02\_handle = am\_fm24c02\_inst\_init();

29 if(app\_test\_ep24cxx(fm24c02\_handle)!=AM\_OK){

30 am\_led\_on(0);

31 }

32 while(1){

33 am\_led\_toggle(0);

34 am\_mdelay(100);

35 }

36 }

由于app\_test\_ep24cxx()的参数为实例handle，与EP24Cxx器件具有依赖关系，因此无法实现跨平台调用。

### NVRAM通用接口函数

由于FM24C02等E2PROM是典型的非易失存储器，因此使用NVRAM（非易失存储器）标准接口读写数据就无需关心具体的器件了。使用这些接口函数前，需将工程配置am\_prj\_config.h的AM\_CFG\_NVRAM\_ENABLE宏的值设置为1，相关函数原型详见表5.4。

表5.4 NVRAM通用接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_ep24cxx\_nvram\_init(  am\_ep24cxx\_handle\_t handle,  am\_nvram\_dev\_t \*p\_dev,  const char \*p\_dev\_name); | NVRAM初始化（am\_ep24cxx.h）； |
| int am\_nvram\_get(  char \*p\_name,  int unit,  uint8\_t \*p\_buf,  int offset,  int len); | 读取数据（am\_nvram.h）； |
| int am\_nvram\_set(  char \*p\_name,  int unit,  uint8\_t \*p\_buf,  int offset,  int len); | 写入数据（am\_nvram.h）； |

#### 初始化函数

NVRAM初始化函数意在初始化FM24C02的NVRAM功能，以便使用NVRAM标准接口读写函数。其函数原型为：

int am\_ep24cxx\_nvram\_init(

am\_ep24cxx\_handle\_t handle, //ep24cxx句柄

am\_nvram\_dev\_t \*p\_dev, //NVRAM设备实例

const char \*p\_dev\_name); //分配的NVRAM存储器设备的名字

其中，ep24cxx实例句柄fm24c02\_handle作为实参传递给handle，p\_dev为指向am\_nvram\_dev\_t类型实例的指针，p\_dev\_name为分配给FM24C02的一个NVRAM设备名，便于其它模块通过该名字定位到FM24C02存储器。

1. 实例（NVRAM存储器）

NVRAM抽象地代表了所有非易失存储器，FM24C02可以看作NVRAM存储器的一个具体实例。定义am\_nvram\_dev\_t类型（am\_nvram.h）实例如下：

am\_nvram\_dev\_t g\_24c02\_nvram\_dev; //定义一个NVRAM实例

其中，g\_24c02\_nvram\_dev为用户自定义的实例，其地址作为p\_dev的实参传递。

1. 实例信息

实例信息仅包含一个由p\_dev\_name指针指定的设备名。设备名为一个字符串，如“fm24c02”。初始化后，该名字就唯一的确定了一个FM24C02存储器设备，如果有多个FM24C02，则可以命名为“fm24c02\_0”、“fm24c02\_1”、“fm24c02\_2”……

基于模块化编程思想，将初始化FM24C02为标准的NVRAM设备的代码存放到对应的配置文件中，通过头文件引出相应的实例初始化函数接口，详见程序清单5.15和程序清单5.16。

程序清单5.15 新增NVRAM实例初始化函数（am\_hwconf\_ep24cxx.c）

1 #include "ametal.h"

2 #include "am\_ep24cxx.h"

3 #include "am\_hwconf\_ep24cxx.h"

4

5 static am\_nvram\_serv\_t \_\_g\_24c02\_nvram\_dev; //NVRAM设备实例定义

6 int am\_fm24c02\_nvram\_inst\_init(void) //NVRAM初始化

7 {

8 am\_ep24cxx\_handle\_t fm24c02\_handle = am\_fm24c02\_inst\_init();

9 am\_ep24cxx\_nvram\_init(fm24c02\_handle,&\_\_g\_24c02\_nvram\_dev,"fm24c02")

10 return AM\_OK;

11 }

程序清单5.16 am\_hwconf\_ep24cxx.h文件更新

1 #pragma once

2 #include "ametal.h"

3 #include "am\_ep24cxx.h"

4

5 am\_ep24cxx\_handle\_t am\_fm24c02\_inst\_init(void);

6 int am\_fm24c02\_nvram\_inst\_init(void);

后续只需要使用无参数的实例初始化函数，即可完成NVRAM设备初始化，将FM24C02初始化为名为“fm24c02”的NVRAM存储设备。即：

am\_fm24c02\_nvram\_inst\_init();

#### 存储段的定义

NVRAM定义了存储段的概念，读写函数均对特定的存储段操作。NVRAM存储器可以被划分为单个或多个存储段。存储段的类型am\_nvram\_segment\_t定义（am\_nvram.h）如下：

typedef struct am\_nvram\_segment{

char \*p\_name; //存储段名字，字符串

int unit; //存储段单元号，区分多个名字相同的存储段

uint32\_t seg\_size; //存储的的大小

const char \*p\_dev\_name; //存储段所处的存储器设备名

}am\_nvram\_segment\_t;

存储段的名字p\_name和单元号unit可以唯一确定一个存储段，当名字相同时，则使用单元号区分不同的存储段。存储段的名字使得每个存储段都被赋予了实际的意义，比如，名为“ip”的存储段表示保存IP地址的存储段，名为“temp\_limit”的存储段表示保存温度上限值的存储段。seg\_addr为该存储段在实际存储器中的起始地址，seg\_size为该存储段的容量大小。p\_dev\_name表示该存储段对应的实际存储设备的名字。

如需将存储段分配到FM24C02上，则需将存储段中的p\_dev\_name设定为“fm24c02”。后续针对该存储段的读写操作实际上就是对FM24C02进行读写操作。为了方便管理，所有存储段统一定义在am\_nvram\_cfg.c文件中，默认情况下存储段为空，其定义为：

static const am\_nvram\_segment\_t \_\_g\_nvram\_segs[]={

{NULL,0,0,0,NULL} //空存储段，必须保留

};

在具有FM24C02存储设备后，即可新增一些段的定义，如应用程序需要使用4个存储段分别存储2个IP地址（4字节x2）、温度上限值（4字节）、和系统系统（50字节），对应的存储段列表（存储段信息的数组）定义如下：

static const am\_nvram\_segment\_t \_\_g\_nvram\_segs[] = {

{"ip", 0, 0, 4, "fm24c02"}, //IP地址存储段0，起始地址0，长度4，存储地址：0~3

{"ip", 1, 4, 4, "fm24c02"}, //IP地址存储段1，起始地址4，长度4，存储地址：4~7

{"temp\_limit", 0, 8, 4, "fm24c02"}, //温度上限值存储段，起始地址8，长度4，存储地址：8~11

{"system", 0, 12, 50, "fm24c02"}, //系统参数存储段，起始地址12，长度50，存储地址：12~61

{"test", 0, 62, 194, "fm24c02"}, //用于测试。起始地址62，长度194，存储地址：62~255

{NULL, 0, 0, 0, NULL} //空存储段，必须保留

};

为了使存储段生效，必须在系统启动时调用am\_nvram\_inst\_init()函数(am\_nvram\_cfg.h),其函数原型为：

void am\_nvram\_inst\_init(void);

该函数往往在板级初始化函数中调用，可以通过工程配置文件（am\_prj\_config.h）中的AM\_CFG\_NVRAM\_ENABLE宏对其进行裁剪，详见程序清单5.17。

程序清单5.17 在板级初始化中初始化NVRAM

1 void am\_board\_init(void)

2 {

3 //......

4 #if(AM\_CFG\_NVRAM\_ENABLE == 1)

5 am\_nvram\_inst\_init();

6 #endif

7 //......

8 }

NVRAM初始化后，根据在am\_nvram\_cfg.c文件中定义的存储段可知，共计增加了5个存储段，它们的名字、单元号和大小分别详见表5.5，后续即可使用通用的NVRAM读写接口对这些存储段进行读写操作。

表5.5 定义的NVRAM存储段

|  |  |  |  |
| --- | --- | --- | --- |
| 新增存储段序号 | 名字 | 单元号 | 大小 |
| 1 | “ip” | 0 | 4 |
| 2 | “ip” | 1 | 4 |
| 3 | “temp\_limit” | 0 | 4 |
| 4 | “system” | 0 | 50 |
| 5 | “test” | 0 | 194 |

#### 写入数据

写入数据函数的原型为：

int am\_nvram\_set(char \*p\_name, int unit, uint8\_t \*p\_buf, int offset, int len);

其中，p\_name和unit分别表示存储段的名字和单元号，确定写入数据的存储段，p\_buf提供写入存储段的数据，offset表示从存储段指定的偏移开始写入数据，len为写入数据的长度。若返回值为AM\_OK，则说明写入成功，反之失败。比如，保存一个IP地址到IP存储段，详见程序清单5.18。

程序清单5.18 写入数据范例程序

1 uint8\_t ip[4] = {192, 168, 40, 12};

2 am\_nvram\_set("ip", 0, &ip[0], 0, 4); //写入非易失性数据"ip"

#### 读取数据

读取数据函数的原型为：

int am\_nvram\_get(char \*p\_name, int unit, uint8\_t \*p\_buf, int offset, int len);

其中，p\_name和unit分别为存储段的名字和单元号，确定读取数据的存储段。p\_buf保存从存储段读到的数据，offset表示从存储段指定的偏移开始读取数据，len为读取数据的长度。若返回值为AM\_OK，则说明读取成功，反之失败。比如，从IP存储段中读取出IP地址，详见程序清单5.19。

程序清单5.19 读取数据范例程序

1 uint8\_t ip[4];

2 am\_nvram\_get("ip", 0, &ip[0], 0, 4); //读取非易失性数据"ip"

现在编写NVRAM通用接口的简单测试程序，测试某个存储段的数据读写是否正常。虽然测试程序是一个简单的应用，但基于模块化编程思想，最好还是将测试相关程序分离出来，程序实现和对应接口的声明详见程序清单5.20和程序清单5.21。

程序清单5.20 测试程序实现（app\_test\_nvram.c）

1 #include "ametal.h"

2 #include "am\_nvram.h"

3 #include "app\_test\_nvram.h"

4

5 int app\_test\_nvram(char \*p\_name, uint8\_t unit)

6 {

7 int i;

8 uint8\_t data[20];

9

10 for(i=0;i<20;i++) //填充数据

11 data[i] = i;

12 am\_nvram\_set(p\_name,unit,&data[0],0,20); //向"test"存储段中写入20个字节数据

13 for(i=0;i<20;i++) //清零数据

14 data[i] = 0;

15 am\_nvram\_get(p\_name,unit,&data[0],0,20); //从"test"存储段中读取20个字节数据

16 for(i=0;i<20;i++){ //比较数据

17 if(data[i]!=i){

18 return AM\_ERROR;

19 }

20 }

21 return AM\_OK;

22 }

程序清单5.21 接口声明（app\_test\_nvram.h）

1 #pragma once

2 #include "ametal.h"

3

4 int app\_test\_nvram(char \*p\_name,uint8\_t unit);

将待测试的存储段（段名和单元号）通过参数传递给测试程序，NVRAM通用接口对测试段读写数据。若读写数据的结果完全相等，则返回AM\_OK，反之返回AM\_ERROR。

由此可见，应用程序的实现不包含任何器件相关的语句，仅仅调用NVRAM通用接口读写指定的存储段，因此该应用程序是跨平台的，在任何AMetal平台中均可使用，进一步整合NVRAM通用接口和测试程序的范例详见程序清单5.22。

程序清单5.22 NVRAM通用接口读写范例程序

1 #include "ametal.h"

2 #include "am\_led.h"

3 #include "am\_delay.h"

4 #include "am\_hwconf\_ep24cxx.h"

5 #include "app\_test\_nvram.h"

6

7 int am\_main(void)

8 {

9 am\_fm24c02\_nvram\_inst\_init();

10 if(app\_test\_nvram("test",0)!=AM\_OK){

11 am\_led\_on(1);

12 }

13 while(1){

14 am\_led\_toggle(0);

15 am\_mdelay(100);

16 }

17 }

显然，NVRAM通用接口赋予了名字的存储段，使得程序在可读性和可维护性方面都优于使用EP24Cxx读写接口。而调用NVRAM通用接口会耗费一定的内存和CPU资源，特别是在要求效率很高或内存紧缺的场合，建议使用EP24Cxx读写接口。

## SPI Flash组件

SPI NOR Flash是一种SPI接口的非易失闪存芯片，本节以台湾旺宏电子的MX25L1606为例详细介绍在AMetal中如何使用类似的Flash存储器。

表5.6 MX25L1606存储器组织结构

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 块号  （block） | 扇区号  （sector） | 页号  （page） | 地址范围 | |
| 起始地址 | 结束地址 |
| 0 | 0 | 0 | 0x000000 | 0x0000FF |
| … | … | … |
| 15 | 0x000F00 | 0x000FFF |
| … | … | … | … |
| 15 | 240 | 0x00F000 | 0x00F0FF |
| … | … | … |
| 255 | 0x00FF00 | 0x00FFFF |

### 基本功能

MX25L1606总容量为16M（16×1024×1024）bits，即2M字节。每个字节对应一个存储地址，因此其存储数据的地址范围为0x000000 ~ 0x1FFFFF。

在MX25L1606中，存储器有块（block）、扇区（sector）和页（page）的概念。页大小为256字节，每个扇区包含16页，扇区大小为4K（4096）字节，每个块包含16个扇区，块的大小为64K（65536）字节，其组织结构示意图详见表5.6。

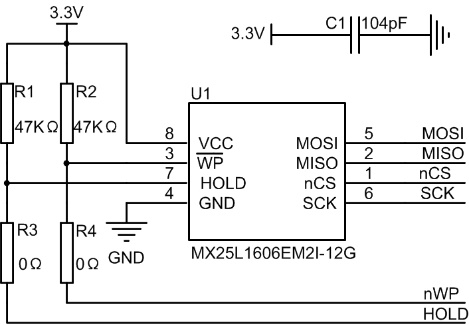


图5.3 SPI Flash电路原理图

MX25L1606的通信接口为标准4线SPI接口（支持模式0和模式3），即CS、MOSI、MISO、CLK，详见图4.6。其中，CS（#1）、SO（#2）、SI（#5）、SCLK（#6）分别为SPI的CS、MISO、MOSI和CLK信号引脚。特别地，WP（#3）用于写保护，HOLD（#7）用于暂停数据传输。一般来说，这两个引脚不会使用，可通过上拉电阻上拉至高电平。MicroPort-NorFlash模块通过MicroPort接口与AM824-Core相连。

### 初始化

AMetal提供了支持常见的MX25L8006、MX25L1606……等系列SPI Flash器件的驱动函数，使用其它各功能函数前必须先完成初始化，其函数原型（am\_mx25xx.h）为：

am\_mx25xx\_handle\_t am\_mx25xx\_init(

am\_mx25xx\_dev\_t \*p\_dev,

const am\_mx25xx\_devinfo\_t \*p\_devinfo,

am\_spi\_handle\_t spi\_handle);

该函数意在获取器件的实例句柄mx25xx\_handle。其中，p\_dev为指向am\_mx25xx\_dev\_t类型实例的指针，p\_devinfo为指向am\_mx25xx\_devinfo\_t类型实例信息的指针。

**（1）实例**

定义am\_mx25xx\_dev\_t类型（am\_mx25xx.h）实例如下：

am\_mx25xx\_dev\_t g\_mx25xx\_dev; // 定义一个mx25xx实例

其中，g\_mx25xx\_dev为用户自定义的实例，其地址作为p\_dev的实参传递。

**（2）实例信息**

实例信息主要描述了具体器件的固有信息，即使用的SPI片选引脚、SPI模式、SPI速率和器件具体型号等，其类型am\_mx25xx\_devinfo\_t的定义（am\_mx25xx.h）如下：

typedef struct am\_mx25xx\_devinfo {

uint16\_t spi\_mode; // SPI模式

int spi\_cs\_pin; // SPI片选引脚

uint32\_t spi\_speed; // SPI速率

am\_mx25xx\_type\_t type; // 器件型号

} am\_mx25xx\_devinfo\_t;

其中，spi\_mode为SPI模式，MX25L1606支持模式0（AM\_SPI\_MODE\_0）和模式3（AM\_SPI\_MODE\_3）。spi\_cs\_pin为与实际电路相关的片选引脚，MicroPort-NorFlash模块通过MicroPort接口与AM824-Core相连时，默认片选引脚为PIO0\_1。spi\_speed为时钟信号的频率，针对MX25L1606，其支持的最高频率为86MHz，因此可以将该值直接设置为86000000。但由于LPC824芯片的主频为30MHz，所以SPI最大速率仅30MHz。type为具体器件的型号，其包含了具体型号相关的信息，比如，页大小信息等，当前已经支持的器件型号详见am\_mx25xx.h中对应的宏，MX25L1606对应的宏为：AM\_MX25XX\_MX25L1606。

基于以上信息，实例信息定义如下：

const am\_mx25xx\_devinfo\_t g\_mx25xx\_devinfo = { // mx25xx实例信息

AM\_SPI\_MODE\_0, // 使用模式0

PIO0\_1, // 片选引脚

30000000, // 总线速率

AM\_MX25XX\_MX25L1606 // 器件型号

};

其中，g\_mx25xx\_devinfo为用户自定义的实例信息，其地址作为p\_devinfo的实参传递。

**（3）SPI句柄spi\_handle**

若使用LPC824的SPI0与MX25L1606通信，则通过LPC82x的SPI0实例初始化函数am\_lpc82x\_spi0\_inst\_init()获得SPI句柄。即：

am\_spi\_handle\_t spi\_handle = am\_lpc82x\_spi0\_inst\_init ();

SPI句柄即可直接作为spi\_handle的实参传递。

**（4）实例句柄**

MX25L1606初始化函数am\_mx25xx\_init ()的返回值MX25L1606实例的句柄，作为其它功能接口（擦除、读、写）的第一个参数（handle）的实参。

其类型am\_mx25xx\_handle\_t（am\_mx25xx.h）定义如下：

typedef struct am\_mx25xx\_dev \*am\_mx25xx\_handle\_t;

若返回值为NULL，说明初始化失败；若返回值不为NULL，说明返回了有效的handle。

基于模块化编程思想，将初始化相关的实例、实例信息等的定义存放到对应的配置文件中，通过头文件引出实例初始化函数接口，源文件和头文件的程序范例分别详见程序清单5.23和程序清单5.24。

程序清单5.23 实例初始化函数范例程序（am\_hwconf\_mx25xx.c）

1 #include "ametal.h"

2 #include "am\_lpc82x\_inst\_init.h"

3 #include "lpc82x\_pin.h"

4 #include "am\_mx25xx.h"

5 #include "am\_mtd.h"

6

7 static const am\_mx25xx\_devinfo\_t \_\_g\_mx25xx\_devinfo = { // mx25xx实例信息

8 AM\_SPI\_MODE\_0, // 使用模式0

9 PIO0\_1, // 片选引脚

10 30000000, // 总线速率

11 AM\_MX25XX\_MX25L1606 // 器件型号

12 };

13 static am\_mx25xx\_dev\_t \_\_g\_mx25xx\_dev; // 定义一个mx25xx实例

14

15 am\_mx25xx\_handle\_t am\_mx25xx\_inst\_init (void)

16 {

17 am\_spi\_handle\_t spi\_handle = am\_lpc82x\_spi0\_inst\_init();// 获取SPI0实例句柄

18 return am\_mx25xx\_init(&\_\_g\_mx25xx\_dev, &\_\_g\_mx25xx\_devinfo, spi\_handle);

19 }

程序清单5.24实例初始化函数接口（am\_hwconf\_mx25xx.h）

1 #pragma once

2 #include "ametal.h"

3 #include "am\_mx25xx.h"

4

5 am\_mx25xx\_handle\_t am\_mx25xx\_inst\_init (void);

后续只需要使用无参数的实例初始化函数，即可获取到MX25xx的实例句柄。即：

am\_mx25xx\_handle\_t mx25xx\_handle = am\_mx25xx\_inst\_init();

注意，spi\_handle用于区分SPI0、SPI1，mx25xx\_handle用于区分同一系统中的多个MX25xx器件。

### 接口函数

SPI Flash比较特殊，在写入数据前必须确保相应的地址单元已经被擦除，因此除读写函数外，还有一个擦除函数，其接口函数详见表5.7。

表5.7MX25xx接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int **am\_mx25xx\_erase**(  am\_mx25xx\_handle\_t handle,  uint32\_t addr,  uint32\_t len); | 擦除 |
| int **am\_mx25xx\_write**(  am\_mx25xx\_handle\_t handle,  uint32\_t addr,  uint8\_t \*p\_buf,  uint32\_t len); | 写入数据 |
| int **am\_mx25xx\_read**(  am\_mx25xx\_handle\_t handle,  uint32\_t addr,  uint8\_t \*p\_buf,  uint32\_t len); | 读取数据 |

各API的返回值含义都是相同的：AM\_OK表示成功，负值表示失败，失败原因可根据

具体的值查看am\_errno.h文件中相对应的宏定义。正值的含义由各API自行定义，无特殊说明时，表明不会返回正值。

1. 擦除

擦除就是将数据全部重置为0xFF，即所有存储单元的位设置为1。擦除操作并不能直接擦除某个单一地址单元，擦除的最小单元为扇区，即每次只能擦除单个或多个扇区。擦除一段地址空间的函数原型为：

int am\_mx25xx\_erase(am\_mx25xx\_handle\_thandle,uint32\_taddr, uint32\_t len);

其中，handle为MX25L1606的实例句柄，addr为待擦除区域的首地址，由于擦除的最小单元为扇区，因此该地址必须为某扇区的起始地址0x000000（0）、0x001000（4096）、0x002000（2×4096）……同时，擦除长度必须为扇区大小的整数倍。

如果返回AM\_OK，说明擦除成功，反之失败。假定需要从0x001000地址开始，连续擦除2个扇区，范例程序详见程序清单5.25。

程序清单5.25 擦除范例程序

1 am\_mx25xx\_erase(mx25xx\_handle, 0x001000, 2 \* 4096); // 擦除两个扇区

0x001000 ~ 0x3FFF空间被擦除了，即可向该段地址空间内写入数据。

1. 写入数据

在写入数据前，需确保写入地址已被擦除。即将需要变为0的位清0，但写入操作无法将0变为1。比如，写入数据0x55就是将bit1、bit3、bit5、bit7清0，其余位的值保持不变。若存储的数据已经是0x55，再写入0xAA（写入0xAA实际上就是将bit0、bit2、bit4、bit6清0，其余位不变），则最终存储的数据将变为0x00，而不是后面再写入的0xAA。因此为了保证正常写入数据，写入数据前必须确保相应的地址段已经被擦除了。

从指定的起始地址开始写入一段数据的函数原型为：

int am\_mx25xx\_write(

am\_mx25xx\_handle\_t handle, // mx25xx实例句柄

uint32\_t addr, // 写入数据的起始地址

uint8\_t \*p\_buf, // 写入数据缓冲区

uint32\_t len); // 写入数据的长度

如果返回AM\_OK，说明写入数据成功，反之失败。假定从0x001000地址开始，连续写入128字节数据，范例程序详见程序清单5.26。

程序清单5.26 写入数据范例程序

1 uint8\_t buf[128];

2 int i;

3 for (i = 0 ; i < 128; i++) buf[i] = i; // 装载数据

4 am\_mx25xx\_erase(mx25xx\_handle, 0x001000, 4096); // 擦除一个扇区

5 am\_mx25xx\_write(mx25xx\_handle, 0x001000, buf, 128); // 写入128字节数据

虽然只写入了128字节数据，但由于擦除的最小单元为扇区，因此擦除了4096字节（一个扇区）。已经擦除的区域后续可以直接写入数据，而不必再次擦除，比如，紧接着写入128字节数据后的地址，再写入128字节数据，详见程序清单5.27。

程序清单5.27 写入数据范例程序

1 am\_mx25xx\_write(mx25xx\_handle, 0x001000 + 128, buf, 128); // 再写入128字节数据

若需要再次从0x001000地址连续写入128字节数据，由于之前已经写入过数据，因此必须重新擦除后方可再次写入。

1. 读取数据

从指定的起始地址开始读取一段数据的函数原型为：

int am\_mx25xx\_read(

am\_mx25xx\_handle\_t handle, // mx25xx实例句柄

uint32\_t addr, // 读取数据的起始地址

uint8\_t \*p\_buf, // 读取数据的缓冲区

uint32\_t len); // 读取数据的长度

如果返回值为AM\_OK，则说明读取成功，反之失败。假定从0x001000地址开始，连续读取128字节数据，详见程序清单5.28。

程序清单5.28 读取数据范例程序

1 uint8\_tdata[128];

2 am\_mx25xx\_read(mx25xx\_handle, 0x001000, buf, 128); // 读取128字节数据

范例程序的实现和接口详见程序清单5.29和程序清单5.30。

程序清单5.29 MX25XX测试程序实现（app\_test\_mx25xx.c）

1 #include "ametal.h"

2 #include "am\_mx25xx.h"

3

4 int app\_test\_mx25xx (am\_mx25xx\_handle\_t mx25xx\_handle)

5 {

6 int i;

7 static uint8\_t buf[128];

8

9 am\_mx25xx\_erase(mx25xx\_handle, 0x001000, 4096); // 擦除扇区1

10 for (i = 0; i < 128; i++) buf[i] = i; // 装载数据

11 am\_mx25xx\_write(mx25xx\_handle, 0x001000, buf, 128);

12 for (i = 0; i < 128; i++) buf[i] = 0x0; // 将所有数据清0

13 am\_mx25xx\_read(mx25xx\_handle, 0x001000, buf, 128);

14 for(i = 0; i < 128; i++) {

15 if (buf[i] != i) {

16 return AM\_ERROR;

17 }

18 }

10 return AM\_OK;

20 }

由于读写数据需要的缓存空间较大（128字节），因此在缓存的定义前增加了static修饰符，使其内存空间从全局数据区域中分配。如果直接从函数的运行栈中分配128字节空间，则完全有可能导致栈溢出，进而系统崩溃。

程序清单5.30 MX25XX测试程序接口声明（app\_test\_mx25xx.h）

1 #pragma once

2 #include "ametal.h"

3 #include "am\_mx25xx.h"

4

5 int app\_test\_mx25xx (am\_mx25xx\_handle\_t mx25xx\_handle);

相应的范例程序详见程序清单5.31。

程序清单5.31 MX25L1602读写范例程序

1 #include "ametal.h"

2 #include "am\_delay.h"

3 #include "am\_led.h"

4 #include "am\_mx25xx.h"

5 #include "am\_hwconf\_mx25xx.h"

6 #include "app\_test\_mx25xx.h"

7

8 int am\_main (void)

9 {

10 am\_mx25xx\_handle\_t mx25xx\_handle = am\_mx25xx\_inst\_init();

11 if (app\_test\_mx25xx(mx25xx\_handle) != AM\_OK) {

12 am\_led\_on(0);

13 while(1);

14 }

15 while(1) {

16 am\_led\_toggle(0); am\_mdelay(200);

17 }

18 return 0;

19 }

由于app\_test\_mx25xx()的参数为MX25XX的实例handle，与MX25xx器件具有依赖关系，因此无法实现跨平台调用。

### MTD通用接口函数

由于MX25L1606是典型的FLASH存储器件，因此将其抽象为一个读写MX25L1606的MTD（Memory Technology Device），使之与具体器件无关，实现跨平台调用，其函数原型详见表5.8。

表5.8MTD通用接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| am\_mtd\_handle\_t **am\_mx25xx\_mtd\_init**(  am\_mx25xx\_handle\_t handle,  am\_mtd\_serv\_t \*p\_mtd,  uint32\_t reserved\_nblks); | MTD初始化  （am\_mx25xx.h） |
| int **am\_mtd\_erase** (  am\_mtd\_handle\_t handle,  uint32\_t addr,  uint32\_t len); | 擦除数据  （am\_mtd.h） |
| int **am\_mtd\_read** (  am\_mtd\_handle\_t handle,  uint32\_t addr,  void \*p\_buf,  uint32\_t len); | 写入数据  （am\_mtd.h） |
| int **am\_mtd\_read** (  am\_mtd\_handle\_t handle,  uint32\_t addr,  void \*p\_buf,  uint32\_t len); | 读取数据  （am\_mtd.h） |

#### MTD初始化函数

MTD初始化函数意在获取MTD实例句柄，其函数原型为：

am\_mtd\_handle\_t am\_mx25xx\_mtd\_init(

am\_mx25xx\_handle\_t handle,

am\_mtd\_serv\_t \*p\_mtd,

uint32\_t reserved\_nblks);

其中，MX25L1606实例句柄（mx25xx\_handle）作为实参传递给handle，p\_mtd为指向am\_mtd\_serv\_t类型实例的指针，reserved\_nblks作为实例信息，表明保留的块数。

* 实例（MTD存储设备）

定义am\_mtd\_serv\_t类型（am\_mtd.h）实例如下：

am\_mtd\_serv\_t g\_mx25xx\_mtd; // 定义一个 MTD 实例

其中，g\_mx25xx\_mtd为用户自定义的实例，其地址作为p\_mtd的实参传递。

* 实例信息

reserved\_nblks表示实例相关的信息，用于MX25L1606保留的块数，这些保留的块不会被MTD标准接口使用。保留的块从器件的起始块开始计算，若该值为5，则MX25XX器件的块0~块4将不会被MTD使用，MTD读写数据将从块5开始。如果没有特殊需求，则该值设置为0。

将MTD初始化函数的调用存放到配置文件中，引出对应的实例初始化接口，详见程序清单5.32和程序清单5.33。

程序清单5.32 新增MTD实例初始化函数（am\_hwconf\_mx25xx.c）

1 static am\_mtd\_serv\_t \_\_g\_mx25xx\_mtd; // 定义一个 MTD 实例

2

3 am\_mtd\_handle\_t am\_mx25xx\_mtd\_inst\_init(void)

4 {

5 am\_mx25xx\_handle\_t mx25xx\_handle = am\_mx25xx\_inst\_init();

6 return am\_mx25xx\_mtd\_init(mx25xx\_handle, &\_\_g\_mx25xx\_mtd, 0);

7 }

程序清单5.33 am\_hwconf\_mx25xx.h文件内容更新（1）

1 #pragma once

2 #include "ametal.h"

3 #include "am\_mx25xx.h"

4

5 am\_mx25xx\_handle\_t am\_mx25xx\_inst\_init (void);

6 am\_mtd\_handle\_t am\_mx25xx\_mtd\_inst\_init(void);

am\_mx25xx\_mtd\_inst\_init()函数无任何参数，与其相关实例和实例信息的定义均在文件内部完成，因此直接调用该函数即可获得MTD句柄。即：

am\_mtd\_handle\_t mtd\_handle = am\_mx25xx\_mtd\_inst\_init();

这样一来，在后续使用其它MTD通用接口函数时，均可使用该函数的返回值mtd\_handle作为第一个参数（handle）的实参传递。

显然，若使用MX25XX接口，则调用am\_mx25xx\_inst\_init()获取MX25XX实例句柄；若使用MTD通用接口，则调用am\_mx25xx\_mtd\_inst\_init()获取MTD实例句柄。

1. 擦除

写入数据前需要确保相应地址已经被擦除，其函数原型为：

int am\_mtd\_erase(

am\_mtd\_handle\_t handle, // MTD实例句柄

uint32\_t addr, // 待擦除区域的首地址

uint32\_t len); // 待擦除区域的长度

擦除单元的大小可以使用宏AM\_MTD\_ERASE\_UNIT\_SIZE\_GET()获得。比如：

uint32\_t erase\_size = AM\_MTD\_ERASE\_UNIT\_SIZE\_GET(mtd\_handle);

其中的addr表示擦除区域的首地址，必须为擦除单元大小的整数倍。同样地，len也必须为擦除单元大小的整数倍。由于MX25L1606擦除单元的大小与扇区大小（4096）一样，因此addr必须为某扇区的起始地址0x000000（0）、0x001000（4096）、0x002000（2×4096）……

如果返回AM\_OK，说明擦除成功，反之失败。假定从0x001000地址开始，连续擦除2个扇区，范例程序详见程序清单5.34。

程序清单5.34 擦除范例程序

am\_mtd\_erase(mx25xx\_handle, 0x001000, 2 \* 4096); // 擦除两个扇区

使用该段程序后，地址空间0x001000 ~ 0x3FFF即被擦除了，后续即可向该段地址空间内写入数据。

1. 写入数据

写入数据前需要确保写入地址已被擦除，其函数原型为：

int am\_mtd\_write(

am\_mtd\_handle\_t handle, // MTD实例句柄

uint32\_t addr, // 写入数据的起始地址

uint8\_t \*p\_buf, // 写入数据缓冲区

uint32\_t len); // 写入数据的长度

如果返回AM\_OK，说明写入数据成功，反之失败。假定从0x001000地址开始，连续写入128字节数据的范例程序详见程序清单5.35。

程序清单5.35 写入数据范例程序

1 uint8\_t buf[128];

2 int i;

3 for (i = 0 ; i < 128; i++)

4 buf[i] = i; // 装载数据

5 am\_mtd\_erase(mtd\_handle, 0x001000, 4096); // 擦除一个扇区

6 am\_mtd\_write(mtd\_handle, 0x001000, buf, 128); // 写入128字节数据

1. 读取数据

从指定的起始地址开始读取一段数据的函数原型为：

int am\_mx25xx\_read(

am\_mx25xx\_handle\_t handle, // MTD实例句柄

uint32\_t addr, // 读取数据的起始地址

uint8\_t \*p\_buf, // 读取数据的缓冲区

uint32\_t len); // 读取数据的长度

如果返回值为AM\_OK，则说明读取成功，反之失败。假定从0x001000地址开始，连续读取128字节数据的范例程序详见程序清单5.36。

程序清单5.36读取数据范例程序

1 uint8\_tdata[128];

2 am\_mtd\_read(mtd\_handle, 0x001000, buf, 128); // 读取128字节数据

MTD通用接口测试程序和接口分别详见程序清单5.37和程序清单5.38。

程序清单5.37 MTD测试程序实现（app\_test\_mtd.c）

1 #include "ametal.h"

2 #include "am\_mtd.h"

3

4 int app\_test\_mtd (am\_mtd\_handle\_t mtd\_handle)

5 {

6 int i;

7 static uint8\_t buf[128];

8

9 am\_mtd\_erase(mtd\_handle, 0x001000, 4096); // 擦除扇区1

10 for (i = 0; i < 128; i++)

11 buf[i] = i; // 装载数据

12 am\_mtd\_write(mtd\_handle, 0x001000 + 67, buf, 128);

13 for (i = 0; i < 128; i++)

14 buf[i] = 0x0; // 将所有数据清零

15 am\_mtd\_read(mtd\_handle, 0x001000 + 67, buf, 128);

16 for(i = 0; i < 128; i++) {

17 if (buf[i] != i) {

18 return AM\_ERROR;

19 }

20 }

21 return AM\_OK;

22 }

程序清单5.38 接口声明（app\_test\_mtd.h）

1 #pragma once

2 #include "ametal.h"

3 #include "am\_mtd.h"

4

5 int app\_test\_mtd (am\_mtd\_handle\_t mtd\_handle);

由于该程序只需要MTD句柄，因此与具体器件无关，可以实现跨平台复用。若读写数据的结果完全相等，则返回AM\_OK，反之返回AM\_ERROR，范例程序详见程序清单5.39。

程序清单5.39 MTD读写范例程序

1 #include "ametal.h"

2 #include "am\_delay.h"

3 #include "am\_led.h"

4 #include "am\_mx25xx.h"

5 #include "am\_hwconf\_mx25xx.h"

6 #include "app\_test\_mtd.h"

7

8 int am\_main (void)

9 {

10 am\_mtd\_handle\_t mtd\_handle = am\_mx25xx\_mtd\_inst\_init();

11 if (app\_test\_mtd(mtd\_handle) != AM\_OK) {

12 am\_led\_on(0);

13 while(1);

14 }

15 while(1) {

16 am\_led\_toggle(0);

17 am\_mdelay(200);

18 }

19 return 0;

20 }

### FTL通用接口函数

由于此前的接口需要在每次写入数据前，确保相应的存储空间已经被擦除，则势必会给编程带来很大的麻烦。与此同时，由于MX25L1606的某一地址段擦除次数超过10万次的上限，则在相应段地址空间存储数据将不再可靠。

假设将用户数据存放到0x001000~0x001FFF连续的4K地址中，则每次更新这些数据都要重新擦除该地址段。而其它存储空间完全没有使用过，MX25L1606的使用寿命大打折扣。AMetal提供了FTL（Flash Translation Layer）通用接口供用户使用，其函数原型详见表5.9。

表5.9 FTL通用接口函数（am\_ftl.h）

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| am\_ftl\_handle\_t**am\_ftl\_init**(  am\_ftl\_serv\_t \*p\_ftl,  uint8\_t \*p\_buf,  uint32\_t len,  am\_mtd\_handle\_t handle); | FTL初始化 |
| int **am\_ftl\_write** (  am\_mtd\_handle\_t handle,  uint32\_t addr,  void \*p\_buf,  uint32\_t len); | 写入数据 |
| int **am\_ftl\_read** (  am\_mtd\_handle\_t handle,  uint32\_t addr,  void \*p\_buf,  uint32\_t len); | 读取数据 |

#### FTL初始化函数

FTL初始化函数意在获取FTL实例句柄，其函数原型为：

am\_ftl\_handle\_t am\_ftl\_init(am\_ftl\_serv\_t \*p\_ftl, uint8\_t \*p\_buf, uint32\_t len, am\_mtd\_handle\_t mtd\_handle);

其中，p\_ftl为指向am\_ftl\_serv\_t类型实例的指针，p\_buf和len作为实例信息，为FTL驱动程序提供必要的RAM空间，MTD初始化函数获得mtd\_handle为MTD实例句柄。

**（1）实例**

定义am\_ftl\_serv\_t类型（am\_mtd.h）实例如下：

am\_ftl\_serv\_t g\_ftl; // 定义一个 FTL 实例

其中，g\_ftl为用户自定义的实例，其地址作为p\_ftl的实参传递。

**（2）实例信息**

FTL驱动程序需要使用一定的RAM空间，这也是使用FTL通用接口所要付出的代价。由于该空间的大小与具体器件的容量大小、擦除单元大小相关，因此该内存空间由用户根据实际情况提供。需要的内存大小（字节数）由下面的公式得到：



其中，sizeerase为擦除单元的大小，对于MX25L1606，其为扇区大小，即4096。sizemtd\_chip为MTD实例的总容量。MX25L1606对应的MTD实例，其大小为除去保留块的总容量，若保留块为0，就是MX25L1606的容量大小，即2M。需要的内存容量大小为：



对于MX25L1606，若使用FTL，则需要大约2.5KB的RAM。显然对于一些小型嵌入式系统来说，RAM的耗费实在“太大”了，所以要根据实际情况选择是否使用FTL。若RAM充足，而又比较在意Flash的使用寿命，可以选择使用FTL。容量大小使用am\_ftl.h中的宏：

AM\_FTL\_RAM\_SIZE\_GET(chip\_size, erase\_size)

该宏根据器件总容量和擦除单元大小，自动计算实际需要的RAM大小。

若使用FTL通用接口操作MX25L1606，则需要定义如下内存空间供FTL使用。即：

uint8\_t g\_ftl\_buf[AM\_FTL\_RAM\_SIZE\_GET(2 \* 1024 \* 1024, 4096)];

其中，g\_ftl\_buf为内存空间的首地址，其作为p\_buf的实参传递，内存空间的大小（即数组元素的个数）作为len的实参传递。

**（3）MTD句柄mtd\_handle**

该MTD句柄可以通过MTD实例初始化函数获得。即：

am\_mtd\_handle\_t mtd\_handle = am\_mx25xx\_mtd\_inst\_init();

获得的MTD句柄即可直接作为mtd\_handle的实参传递。

**（4）实例句柄**

FTL初始化函数am\_ftl\_init ()的返回值为FTL实例句柄，该句柄将作为读写接口第一个参数（handle）的实参。其类型am\_ftl\_handle\_t（am\_ftl.h）定义如下：

typedef am\_ftl\_serv\_t \*am\_ftl\_handle\_t;

若返回值为NULL，说明初始化失败；若返回值不为NULL，说明返回了有效的handle。

将FTL初始化函数的调用存放到配置文件中，引出对应的实例初始化接口，详见程序清单5.40和程序清单5.41。

程序清单5.40 新增FTL实例初始化函数（am\_hwconf\_mx25xx.c）

1 #include "am\_ftl.h"

2 static am\_ftl\_serv\_t \_\_g\_ftl; // 定义一个 FTL 实例

3 static uint8\_t \_\_g\_ftl\_buf[AM\_FTL\_RAM\_SIZE\_GET(2 \* 1024 \* 1024, 4096)];

4

5 am\_ftl\_handle\_t am\_mx25xx\_ftl\_inst\_init(void)

6 {

7 am\_mtd\_handle\_t mtd\_handle = am\_mx25xx\_mtd\_inst\_init();

8 return am\_ftl\_init(&\_\_g\_ftl, &\_\_g\_ftl\_buf[0], AM\_NELEMENTS(\_\_g\_ftl\_buf), mtd\_handle);

9 }

程序清单5.41 am\_hwconf\_mx25xx.h文件内容更新（2）

1 #pragma once

2 #include "ametal.h"

3 #include "am\_mx25xx.h"

4 #include "am\_ftl.h"

5

6 am\_mx25xx\_handle\_t am\_mx25xx\_inst\_init (void);

7 am\_mtd\_handle\_t am\_mx25xx\_mtd\_inst\_init(void);  
8 am\_ftl\_handle\_t am\_mx25xx\_ftl\_inst\_init(void);

am\_mx25xx\_ftl\_inst\_init()无任何参数，与其相关实例和实例信息的定义均在文件内部完成，因此直接调用该函数即可获得FTL句柄。即：

am\_ftl\_handle\_t ftl\_handle = am\_mx25xx\_ftl\_inst\_init();

这样一来，在后续使用其它FTL通用接口函数时，均可使用该函数的返回值ftl\_handle作为第一个参数（handle）的实参传递。

1. **写入数据**

当调用FTL通用接口时，读写数据都是以块为单位，每块数据的字节数固定为512字节。其函数原型为：

int am\_ftl\_write (

am\_ftl\_handle\_t handle, // FTL实例句柄

unsigned int lbn, // 逻辑块号

uint8\_t \*p\_buf); // 写入数据缓冲区（其大小必须为512）

为了延长Flash的使用寿命，在实际写入时，会数据写入到擦除次数最少的区域。因此lbn只是一个逻辑块序号，与实际的存储地址没有关系。逻辑块只是一个抽象的概念，每个逻辑块的大小固定为512字节，与MX25L1606的物理存储块没有任何关系。

由于MX25L1606每个逻辑块固定为512字节，因此理论上逻辑块的个数为4096（2×1024×1024÷512），lbn的有效值为0 ~ 4095。但实际上擦除每个单元都要耗费一个逻辑块，MX25L1606擦除单元的大小为4096，即512个擦除单元，因此FTL消耗了512个逻辑块，则可用的逻辑块为3584（4096~512）个，lbn的有效值为0~3583。

由此可见，FTL不仅要占用2.5K RAM，还要占用256K的MX25L1606存储空间（512个逻辑块，每个逻辑块大小为512字节），这也是使用FTL要付出的“代价”。如果返回AM\_OK，说明写入数据成功，反之失败。假定写入一块数据（512字节）至逻辑块2中，其范例程序详见程序清单5.42。

程序清单5.42 写入数据范例程序

1 uint8\_t buf[512];

2 int i;

3 for (i = 0 ; i < 512; i++)

4 buf[i] = i & 0xFF; // 装载数据

5 am\_ftl\_write(ftl\_handle, 2, buf, 512); // 向逻辑块2写入512字节数据

1. 读取数据

读取一块数据的函数原型为：

int am\_ftl\_read (

am\_ftl\_handle\_t handle, // FTL实例句柄

unsigned int lbn, // 逻辑块号

uint8\_t \*p\_buf); // 读取数据缓冲区（其大小必须为512）

如果返回值为AM\_OK，则说明读取成功，反之失败。假定从逻辑块2中读取一块（512字节）数据，其范例程序详见程序清单5.43。

程序清单5.43读取数据范例程序

1 uint8\_tbuf [512];

2 am\_ftl\_read(ftl\_handle, 2, buf, 512); // 从逻辑块2中读出512字节数据

FTL通用接口测试程序和接口分别详见程序清单5.44和程序清单5.45。

程序清单5.44 FTL测试程序实现（app\_test\_ftl.c）

1 #include "ametal.h"

2 #include "am\_ftl.h"

3

4 int app\_test\_ftl (am\_ftl\_handle\_t ftl\_handle)

5 {

6 int i;

7 static uint8\_t buf[512];

8

9 for (i = 0; i < 512; i++) buf[i] = i & 0xFF; // 装载数据

10 am\_ftl\_write(ftl\_handle, 2, buf);

11 for (i = 0; i < 512; i++) buf[i] = 0x0; // 将所有数据清零

12 am\_ftl\_read(ftl\_handle, 2, buf);

13 for(i = 0; i < 512; i++) {

14 if (buf[i] != (i & 0xFF)) {

15 return AM\_ERROR;

16 }

17 }

18 return AM\_OK;

19 }

程序清单5.45 FTL测试接口声明（app\_test\_ftl.h）

1 #pragma once

2 #include "ametal.h"

3 #include "am\_ftl.h"

4

5 int app\_test\_ftl (am\_ftl\_handle\_t ftl\_handle);

由于写入前无需再执行擦除操作，则编写应用程序更加便捷。同样，由于应用程序仅仅只需要FTL句柄，则所有接口也全部为FTL通用接口，因此应用程序是可以跨平台复用的，范例程序详见程序清单5.46。

程序清单5.46 FTL读写范例程序

1 #include "ametal.h"

2 #include "am\_delay.h"

3 #include "am\_led.h"

4 #include "am\_mx25xx.h"

5 #include "am\_hwconf\_mx25xx.h"

6 #include "app\_test\_mtd.h"

7

8 int am\_main (void)

9 {

10 am\_ftl\_handle\_t ftl\_handle = am\_mx25xx\_ftl\_inst\_init();

11 if (app\_test\_ftl(ftl\_handle) != AM\_OK) {

12 am\_led\_on(0); while(1);

13 }

14 while(1) {

15 am\_led\_toggle(0); am\_mdelay(200);

16 }

17 return 0;

18 }

### 微型数据库

由于哈希表所使用的链表头数组空间、关键字和记录值等都存储在malloc分配的动态空间中，这些信息在程序结束或系统掉电后都会丢失。在实际的应用中，往往希望将信息存储在非易失存储器中。典型的应用是将信息存储在文件中，从本质上来看，只要掌握了哈希表的原理，无论信息存储在什么地方，操作的方式都是一样的。

在AMetal中，基于非易失存储器实现了一套可以直接使用的哈希表接口，由于数据不会因为掉电或程序终止而丢失，因此可以将其视为一个微型数据库，相关接口详见表5.10。

表5.10 数据库接口（hash\_kv.h）

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int hash\_kv\_init (  hash\_kv\_t \*p\_hash,  uint16\_t size,  uint16\_t key\_size,  uint16\_t value\_size,  hash\_func\_t hash,  const char \*file\_name); | 哈希表初始化 |
| int hash\_kv\_add (hash\_kv\_t \*p\_hash, const void \*key, const void \*value); | 增加一条记录 |
| int hash\_kv\_search (hash\_kv\_t \*p\_hash, const void \*key, void \*value); | 根据关键字查找记录 |
| int hash\_kv\_del (hash\_kv\_t \*p\_hash, const void \*key); | 删除一条记录 |
| int hash\_kv\_deinit (hash\_kv\_t \*p\_hash); | 资源释放 |

显然，除命名空间由 hash\_db\_\*修改为了hash\_kv\_\*（为了与之前的程序进行区分）外，仅仅是初始化函数中，多了一个文件名参数，即内部不再使用malloc分配空间存储记录信息，而是使用该文件名指定的文件存储相关信息。如此一来记录存储在文件中，信息不会因掉电或程序终止而丢失。其中，hash\_kv\_t为数据库结构体类型，使用数据库前，应使用该类型定义一个数据库实例，比如：

hash\_kv\_t hash;

由于各个函数的功能与《程序设计与数据结构》一书中介绍的哈希表的各个函数的功能完全一致，因此可以使用如程序清单5.47所示的代码进行测试验证。

程序清单5.47 数据库综合范例程序

1 #include <stdio.h>

2 #include <stdlib.h>

3 #include "hash\_kv.h"

4

5 typedef struct \_student{

6 char name[10]; // 姓名

7 char sex; // 性别

8 float height, weight; // 身高、体重

9 } student\_t;

10

11 int db\_id\_to\_idx (unsigned char id[6]) // 通过ID得到数组索引

12 {

13 int i;

14 int sum = 0;

15 for (i = 0; i < 6; i++)

16 sum += id[0];

17 return sum % 250;

18 }

19

20 int student\_info\_generate (unsigned char \*p\_id, student\_t \*p\_student) // 随机产生一条学生记录

21 {

22 int i;

23 for (i = 0; i < 6; i++) // 随机产生一个学号

24 p\_id[i] = rand();

25 for (i = 0; i < 9; i++) // 随机名字，由 'a' ~ 'z' 组成

26 p\_student->name[i] = (rand() % ('z' - 'a')) + 'a';

27 p\_student->name[i]= '\0'; // 字符串结束符

28 p\_student->sex = (rand() & 0x01) ? 'F' : 'M'; // 随机性别

29 p\_student->height = (float)rand() / rand();

30 p\_student->weight = (float)rand() / rand();

31 return 0;

32 }

33

34 int main ()

35 {

36 student\_t stu;

37 unsigned char id[6];

38 int i;

39 hash\_kv\_t hash\_students;

40

41 hash\_kv\_init(&hash\_students, 250, 6, sizeof(student\_t), (hash\_func\_t)db\_id\_to\_idx, "hash\_students");

42 for (i = 0; i < 100; i++){ // 添加100个学生的信息

43 student\_info\_generate(id, &stu); // 设置学生信息，用随机数作为测试

44 if (hash\_kv\_search(&hash\_students, id, &stu) == 0){ // 查找到已经存在该ID的学生记录

45 printf("该ID的记录已经存在！\n");

46 continue;

47 }

48 printf("增加记录：ID : %02x%02x%02x%02x%02x%02x",id[0],id[1],id[2],id[3],id[4],id[5]);

49 printf("信息： %s %c %.2f %.2f\n", stu.name, stu.sex, stu.height, stu.weight);

50 if (hash\_kv\_add(&hash\_students, id, &stu) != 0){

51 printf("添加失败");

52 }

53 }

54 printf("查找ID为：%02x%02x%02x%02x%02x%02x的信息\n",id[0],id[1],id[2],id[3],id[4],id[5]);

55 if (hash\_kv\_search(&hash\_students, id, &stu) == 0)

56 printf("学生信息： %s %c %.2f %.2f\n", stu.name, stu.sex, stu.height, stu.weight);

57 else

58 printf("未找到该ID的记录！\r\n");

59 hash\_kv\_deinit(&hash\_students);

60 return 0;

61 }

## 数码管组件

## 72128组件

## 液晶驱动组件

## RTC组件

## 读卡组件

AMetal已经提供了读卡组件的驱动函数，本节以FM1750为例详解该组件。

读卡芯片FM17520内部集成了强大的内部电路，外部电路设计通常比较简单，主要组成为：供电电路、通信接口电路、天线电路和振荡电路。其中天线电路设计由为重要，读卡器天线电路主要分成四个部分：EMC滤波、匹配电路、天线和接收电路，如图5.1所示。

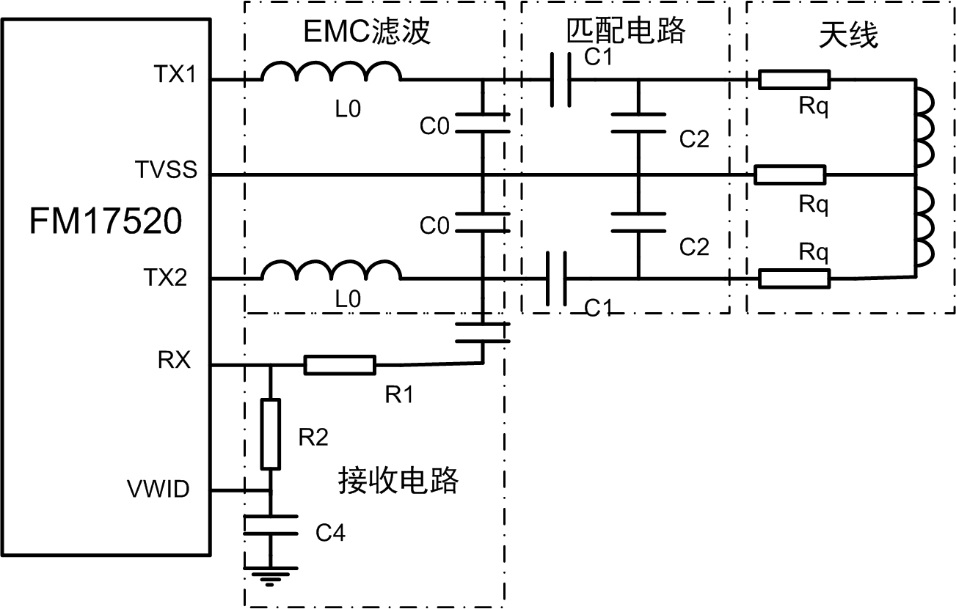


图5.4 读卡电路

滤波电路主要用于滤去高于13.56MHz的衍生谐波，由L0和C0组成的低通滤波器，滤波器截止频率设计在14MHz以上。这里推荐680nH电感和180pF电容或者1μH电感和120pF电容。这两组组合使得匹配网络既不偏感性也不偏容性，设计合理。

匹配电路用于调节发射负载和谐振频率。电路由电容C1和C2组成。射频电路发射功率一般受芯片内阻抗和外阻抗影响，当芯片内阻抗和外阻抗一致时，发射功率最大。C1是负载电容，天线电感量越大，C1取值越小。电容C2是谐振电容，通常设计由两个电容并联。C2取值和天线电感量直接相关，使得谐振频率在13.56MHz。

天线设计由Q值电阻Rq (通常1Ω或者0Ω)和印制PCB线路组成。 接收电路由R1、R2和C3、C4组成，其中C3=102pF、C4=104pF。R1和R2组成分压电路，使得RX端接收正弦波幅度电压在1.5V-3V之间。

接收电路由R1、R2和C3、C4组成接收电路。R1和R2组成分压电路，确保RX端接收到的正弦波峰峰值Vpp在1.5V-3V之间。

### 读卡通信设计

AMetal已经提供了FM175XX系列的驱动函数，在使用之前，必须先完成初始化，初始化函数详见表5.11。

表5.11 FM175xx初始化接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| uint8\_t am\_fm175xx\_init (am\_fm175xx\_dev\_t \*p\_dev,  am\_spi\_handle\_t spi\_handle,  const am\_fm175xx\_devinfo\_t \*p\_devinfo); | FM175xx初始化，得到  FM175xx设备信息。 |

1. 初始化

初始化意在获取FM175xx实例句柄（handle），该实例句柄将作为其他功能接口函数handle的实参。其函数初始化原型为：

uint8\_t am\_fm175xx\_init (am\_fm175xx\_dev\_t \*p\_dev,

am\_spi\_handle\_t spi\_handle,

const am\_fm175xx\_devinfo\_t \*p\_devinfo);

* p\_dev为指向am\_fm175xx \_dev\_t类型实例的指针；
* p\_devinfo为指向am\_ fm175xx \_devinfo\_t类型实例信息的指针；
* spi\_handle为获取SPI服务的实例化句柄。

1. 实例信息

实例信息主要描述了FM175xx的相关信息，包括SPI设备信息、软件定时器、超时计数器、保存读卡芯片协议、命令信息、天线状态、掉电标志等信息。其类型am\_fm175xx\_dev\_t定义（am\_fm175xx.h）如下：

/\*\*

\* \brief FM175XX 设备定义

\*/

typedef struct am\_fm175xx\_dev {

am\_spi\_device\_t spi\_dev; // SPI 设备

am\_softimer\_t timer; //软件定时器，用于超时

volatile uint32\_t tmo\_cnt; //超时计数器

am\_fm175xx\_prot\_type\_t iso\_type; //保存读卡芯片协议

am\_fm175xx\_cmd\_info\_t cmd\_info; //命令信息

volatile uint8\_t tx\_state; //天线状态

volatile am\_bool\_t power\_down; // 掉电标志

am\_fm175xx\_tpcl\_prot\_para\_t cur\_prot\_para; //T=CL通信协议参数

const am\_fm175xx\_devinfo\_t \*p\_devinfo; //设备信息

void (\*lpcd\_int\_cb)(void \*p\_arg);

void \*p\_lpcd\_cb\_arg;

} am\_fm175xx\_dev\_t;

### 设备控制类接口函数

FM17520支持多种IC卡，比如，Mifare S50/S70、ISO7816-3、ISO14443（PICC）、PLUS CPU卡等，每种卡都有对应的命令。命令与接口函数基本上是一一对应的关系，AMetal提供了标准接口函数，与具体卡片没有直接关系，直接作用于FM17520，获取相应的设备信息、通信加密、设置防碰撞及卡请求模式等。

表5.12 FM17520设备控制接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| uint8\_t am\_fm175xx\_crypto1 (am\_fm175xx\_dev\_t \*p\_dev,  uint8\_t mode,  const uint8\_t p\_key[6],  const uint8\_t p\_uid[4],  uint8\_t nblock); | 设置通信加密 |
| uint8\_t am\_fm175xx\_picca\_anticoll (am\_fm175xx\_dev\_t \*p\_dev,  uint8\_t anticoll\_level,  uint8\_t \*p\_uid,  uint8\_t \*p\_real\_uid\_len); | 设置防碰撞等级 |
| uint8\_t am\_fm175xx\_picca\_request (am\_fm175xx\_dev\_t \*p\_dev,  uint8\_t req\_mode,  uint8\_t p\_atq[2]); | 卡请求模式 |

* 设置通信加密

该函数意在设置通信加密类型，卡片内存储的数据均是加密的，必须验证成功后才能读写数据。验证就是将用户提供的密钥与卡片内部存储的密钥对比，只有相同才认为验证成功。设置密钥类型，主要有密钥A、密钥B，可设置为外部输入的密钥验证，或使用内部E2的密钥验证。使用内部密钥时，第一字节为密钥存放扇区。其函数定义如下：

uint8\_t am\_fm175xx\_crypto1 (am\_fm175xx\_dev\_t \*p\_dev,

uint8\_t mode,

const uint8\_t p\_key[6],

const uint8\_t p\_uid[4],

uint8\_t nblock);

IC卡密钥类型定义如下：

#define AM\_FM175XX\_IC\_KEY\_TYPE\_A 0x60 /\*\*< \brief 类型 A \*/

#define AM\_FM175XX\_IC\_KEY\_TYPE\_B 0x61 /\*\*< \brief 类型 B \*/

注：之所以存在两类密钥，是由于实际卡片中往往存在两类密钥，两类密钥可以更加方便地进行权限管理，比如，TypeA验证成功后只能读，而TypeB只有验证成功后才能写入，但权限可以自定义设置。

* 设置防碰撞等级

设置防碰撞等级，符合ISO14443A标准卡的序列号都是全球唯一的，正是这种唯一性，才能实现防碰撞的算法逻辑，当若干卡同时在天线感应区内，则这个函数能够找到一张序列号较大的卡来操作。该函数需要执行一次请求命令，并返回请求成功，才能执行防碰撞操作，否则返回错误。

uint8\_t am\_fm175xx\_picca\_anticoll (am\_fm175xx\_dev\_t \*p\_dev,

uint8\_t anticoll\_level,

uint8\_t \*p\_uid,

uint8\_t \*p\_real\_uid\_len);

防碰撞等级设置有如下三级设置：

#define AM\_FM175XX\_PICCA\_ANTICOLL\_1 0x93 /\*\*< \brief 第一级防碰撞 \*/

#define AM\_FM175XX\_PICCA\_ANTICOLL\_2 0x95 /\*\*< \brief 第二级防碰撞 \*/

#define AM\_FM175XX\_PICCA\_ANTICOLL\_3 0x97 /\*\*< \brief 第三级防碰撞 \*/

防碰撞等级设置参考卡的序列号长度，目前主流卡的序列号长度有三种，4字节、7字节和10字节，4字节选择第一级防碰撞即可得到完整的序列号，7字节使用第二等级防碰撞可得到完整序列号，前一级多得到的序列号的最低字节为级联标志0x88，在序列号内，只有3字节可用，后一级选择能得到4字节序列号，两者按顺序连接为7字节序列号，10字节以此类推。

* 卡请求模式设置

卡进入天线后，从射频场中获取能量，从而得电复位，复位后卡处于IDLE模式，用两种请求模式的任一种请求时，此时的卡均能响应，若对某一张卡成功挂起，则进入Halt模式，此时卡只响应ALL（0x52）模式的请求，除非将卡离开天线感应区后再进入。

uint8\_t am\_fm175xx\_picca\_request (am\_fm175xx\_dev\_t \*p\_dev,

uint8\_t req\_mode,

uint8\_t p\_atq[2]);

卡请求模式主要有IDLE和ALL两种，如下所示：

#define AM\_FM175XX\_PICCA\_REQ\_IDLE 0x26 /\*\*< \brief IDLE模式，请求空闲的卡 \*/

#define AM\_FM175XX\_PICCA\_REQ\_ALL 0x52 /\*\*< \brief ALL模式，请求所有的卡 \*/

### 操作接口函数

Mifare卡是一种符合ISO14443标准的A型卡，其接口函数详见表5.13。

表5.13 读卡操作接口函数

|  |  |  |
| --- | --- | --- |
| 接口类型 | 函数原型 | 功能简介 |
| 卡片自动  检测接口函数 | uint8\_t am\_fm175xx\_picca\_authent (am\_fm175xx\_dev\_t \*p\_dev,  uint8\_t key\_type,  const uint8\_t p\_uid[4],  const uint8\_t p\_key[6],  uint8\_t nblock); | 密钥验证 |
| 读卡操作接口函数 | uint8\_t am\_fm175xx\_picca\_read (am\_fm175xx\_dev\_t \*p\_dev,  uint8\_t nblock,  uint8\_t p\_buf[16]); | 卡读数据 |
| uint8\_t am\_fm175xx\_picca\_write (am\_fm175xx\_dev\_t \*p\_dev,  uint8\_t nblock,  const uint8\_t p\_buf[16]); | 卡写数据 |
| uint8\_t am\_fm175xx\_picca\_val\_set (am\_fm175xx\_dev\_t \*p\_dev,  uint8\_t nblock,  int32\_t value); | 卡值块写值 |
| uint8\_t am\_fm175xx\_picca\_val\_get (am\_fm175xx\_dev\_t \*p\_dev,  uint8\_t nblock,  int32\_t \*p\_value); | 卡值块读取 |

经常使用的公交卡、房卡、水卡和饭卡等均是Mifare卡。比如，S50和S70，它们的区别在于容量的不同。S50为1Kbyte，共16个扇区，每个扇区4块，每块16字节。S70为4K byte，共40个扇区，前32个扇区每个扇区4块，每块16字节，后8个扇区每个扇区16块，每块16字节。

* 密钥验证

由于绝大部分卡片在检测到时，都要先读取一块数据，因此可以将读取数据作为自动检测的一个附加功能。即在检测到卡片时，自动读取1块（16字节）数据。由于读取数据前均需要验证，这就需要在启动自动检测时，指定密钥验证相关的信息。将传入的密钥与卡的密钥进行验证，对应的卡的序列号有4字节和7字节之分，对于7字节的卡，只需将卡号的高4字节，即第二防碰撞等级得到的序列号作为验证的卡号即可。

每张卡片都具有一个唯一序列号，即UID。所有卡片的UID都是不相同的。卡的序列号长度有三种：4字节、7字节和10字节。uid\_len表明了读取到的UID的长度，uid[4]中存放了读取到的UID（字节数）。

uint8\_t am\_fm175xx\_picca\_authent (am\_fm175xx\_dev\_t \*p\_dev,

uint8\_t key\_type, //密钥类型

const uint8\_t p\_uid[4], //卡序列号，4字节

const uint8\_t p\_key[6], //密钥，6字节

uint8\_t nblock); //需验证卡块号，与卡有关

* 卡读数据

验证成功后，才能读相应的块数据，所验证的块号与读块号必须在同一个扇区内，Mifare1卡从块号0开始，按顺序每4个块1个扇区，若要对一张卡中的多个扇区进行操作，在对某一个扇区操作完成后，必须进行一条读命令才能对另一个扇区直接进行验证命令，否则必须从请求开始操作，对于PLUS CPU卡，若对下一个读扇区的密钥和当前扇区的密钥相同，则不需要再次验证密钥，直接读即可。

对应的密钥正确，验证成功，将读取启动自动检测时信息结构体的nblock成员指定的块（由信息结构体的nblock指定）的数据。读取的数据存放在p\_buf[16]数组中。

uint8\_t am\_fm175xx\_picca\_read (am\_fm175xx\_dev\_t \*p\_dev,

uint8\_t nblock, //读取数据的块号

uint8\_t p\_buf[16]); //存放读取数据，16bytes

* 卡写数据

对卡内某一块进行验证成功后，即可对同一个扇区的各个块进行写操作（只要访问条件允许），其中包括位于扇区尾的密码块，这是更改密码的唯一方法，对于PLUS CPU卡等级2、3的AES密钥则是其他位置修改密钥，写入数据缓冲区，大小必须为16。

uint8\_t am\_fm175xx\_picca\_write (am\_fm175xx\_dev\_t \*p\_dev,

uint8\_t nblock, //读取数据的块号

const uint8\_t p\_buf[16]); //写入缓冲区，大小必须为16

* 卡块值写操作

对Mifare卡块值的设置，其中，nblock指定写入的块号，value为指向写入数据的值，缓冲区大小为16字节。对卡内某一块进行验证成功后，并且访问条件允许，才能进行该写值操作。

uint8\_t am\_fm175xx\_picca\_val\_set (am\_fm175xx\_dev\_t \*p\_dev,

uint8\_t nblock, //块值地址

int32\_t value); //写入值

* 卡块值获取

对Mifare卡块值的读取，若验证成功，则开始读写已验证的块。读写数据都是以块为单位的，其大小为16字节，指定读取数据的值块地址，nblock指定本次验证的块号，可以使用该函数读取数值块的值。对卡内某一块进行验证成功后，并且访问条件允许，才能进行读值操作。

uint8\_t am\_fm175xx\_picca\_val\_get (am\_fm175xx\_dev\_t \*p\_dev,

uint8\_t nblock, //块值地址

int32\_t \*p\_value); //获取值指针

### 密钥和权限控制

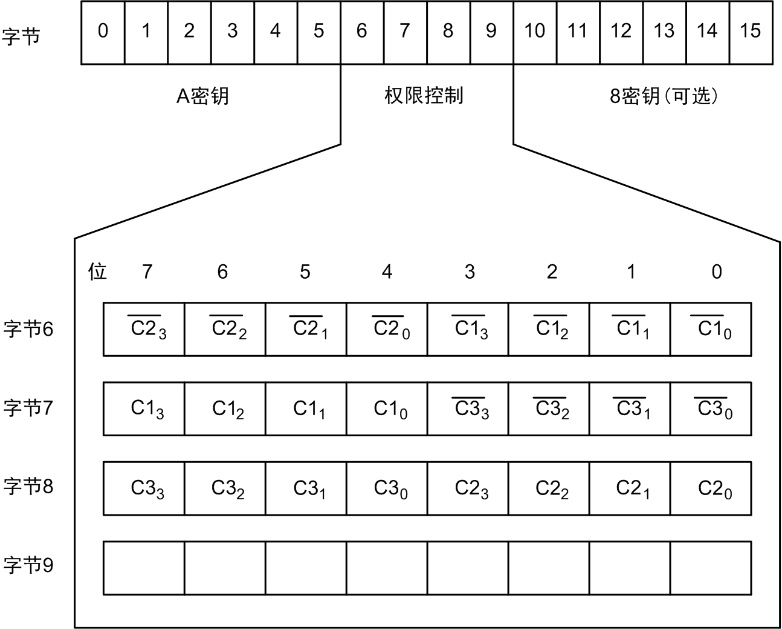


图5.5 尾块格式

Mifare S50/S70卡的初始密钥全为0xFF，显然，对于实际产品来讲，希望能够更改其密钥为其它值。由于存在密钥A和密钥B，可以对每个密钥设定不一样的权限，如验证密钥A后仅只读，验证密钥B后可写。下面以Mifare S50为例，介绍密钥和权限控制的修改方法。

密钥和权限控制是针对扇区而言的，即一个扇区的密钥是相同的，不同扇区的密钥可以不同。S50共计16个扇区，每个扇区4块，每块16字节，前3块为普通的数据块，最后一块（尾块）为密钥和权限控制块。对最后一块存储的数据进行修改，即可完成密钥和权限控制的修改。操作最后一块的存储数据时要格外小心，数据稍有错误，就可能导致扇区被锁死。尾块的前6字节为A密钥，后6字节为B密钥，中间4字节用于权限控制，详见图5.5。

如需修改密钥和控制权限，重点在理解字节6、7和8（字节9是一个普通的数据）的含义。3个字节共计24位，每6位（分别为C1、C2、C3、、、）控制扇区中的一个块，刚好可以控制4个块，图中的下标0、1、2、3对应块0、1、2、3。如 C10表示块0的C1控制位。同样的标志位，上方带横线的位必须与不带横线的位的值相反，即如果C10的值为1，则的值就必须为0。

由于存在此关系，因此实际控制位的含义仅通过C1、C2、C3三个位即可确定。控制尾块和数据块的控制位含义是不同的。对于尾块，其控制了密钥A、密钥B以及控制区域的访问权限。控制位的含义详见表5.14。

表5.14 尾块控制位含义

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 控制位 | | | 访问权限 | | | | | |
| 密钥A区域 | | 控制区域 | | 密钥B区域 | |
| C1 | C2 | C3 | 读 | 写 | 读 | 写 | 读 | 写 |
| 0 | 0 | 0 | × | KeyA | KeyA | × | KeyA | KeyA |
| 0 | 1 | 0 | × | × | KeyA | × | KeyA | × |
| 1 | 0 | 0 | × | KeyB | KeyA | B | × | × | KeyB |
| 1 | 1 | 0 | × | × | KeyA | B | × | × | × |
| 0 | 0 | 1 | × | KeyA | KeyA | KeyA | KeyA | KeyA |
| 0 | 1 | 1 | × | KeyB | KeyA | B | KeyB | × | KeyB |
| 1 | 0 | 1 | × | × | KeyA | B | KeyB | × | × |
| 1 | 1 | 1 | × | × | KeyA | B | × | × | × |

表中，“×”表示任何情况下都无权限，“KeyA”表示通过密钥A验证后可以取得权限，KeyB表示通过密钥B验证后可以取得权限，“KeyA | B”表示通过密钥A或者密钥B验证后均可取得权限。由此可见，密钥A的安全性很高，任何情况下都无法读出。特殊情况下，当C1C2C3的值为000、010或001时，验证密钥A后即可读取密钥B区域的数据。

无论什么情况，验证密钥A后，均可获得控制区域的读权限。通过读取控制区域，可以知道当前C1、C2、C3的值，以判断需要验证哪个密钥后可以获得密钥区域或控制区域的写权限，进而修改密钥和控制区域的值。比如，当前的C1、C2、C3的值为0、1、1，为了修改密钥A，则需要先验证密钥B，验证密钥B后，即可对尾块进行写入，写入时其它数据保持不变，仅修改前6字节（KeyA区域）的值即可完全对密钥A的修改。

对于数据块，C1、C2、C3控制了对块中存储数据的操作权限，详见表5.15。

表5.15 数据块控制位含义

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 控制位 | | | 数据访问权限 | | | |
| C1 | C2 | C3 | 读 | 写 | 加值操作 | 减值操作 |
| 0 | 0 |  | KeyA | B | KeyA | B | KeyA | B | KeyA | B |
| 0 | 1 | 0 | KeyA | B | × | × | × |
| 1 | 0 | 0 | KeyA | B | KeyB | × | × |
| 1 | 1 | 0 | KeyA | B | KeyB | KeyB | KeyA | B |
| 0 | 0 | 1 | KeyA | B | × | × | KeyA | B |
| 0 | 1 | 1 | KeyB | KeyB | × | × |
| 1 | 0 | 1 | KeyB | × | × | × |
| 1 | 1 | 1 | × | × | × | × |

同样，表中“×”表示任何情况下都无权限，“KeyA”表示通过密钥A验证后可以取得权限，KeyB表示通过密钥B验证后可以取得权限，“KeyA | B”表示通过密钥A或者密钥B验证后均可取得权限。

加值操作（相当于充值）和减值操作（相当于消费）是对块中存放的值进行增加和减少操作，加值和减值均有对应的命令可以直接使用。例如，当前块1的C1、C2、C3控制位的值为0、0、0（默认值），只要密钥A或密钥B验证通过后，均可取得数据块的读、写、加值、减值的权限。可以根据实际需要，修改尾块中相应控制位的值（修改时，需确保具有写入控制区域的权限），以对数据进行保护。

需要注意的是，凡是表中标识验证密钥B后可以取得权限的，在特殊情况下验证密钥B后可能并不能取得权限。在介绍尾块控制位含义时，当C1、C2、C3的值为000、010或001时，KeyB区域将可能被读取，详见表5.14。这些情况下，由于密钥B可能被读取，为了确保安全，此时密钥B验证将无效，即使密钥B验证通过，同样无法取得相应的权限。

## LSM6DSL传感器组件

LSM6DSL是ST半导体推出的具有I2C和SPI接口的数字加速度计和3D数字陀螺仪，可以检测方向、倾斜角度、温度等，高性能模式下功耗仅0.65mA，适用于计步等产品开发，其硬件使用较为灵活。

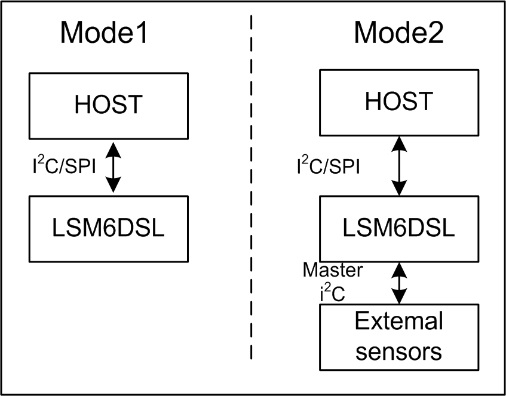


图5.6 LSM6DSL工作模式

### 器件简介

LSM6DSL是一种系统级封装，属高性能的3轴数字加速度计和3轴陀螺仪，集成的低功耗模式能够在高性能模式下将功耗降低至0.65mA，使高精度与低功耗相结合，可以检测方向和手势，提供设备定位到纵向和横向模式更复杂的特性和功能，事件检测中断可以实现高效的运动跟踪感知。

LSM6DSL使用灵活，详见图5.6。采用模式1，作为从机与处理器通信；采用模式2，可作为主机与外部传感器通信，同时可作为从机与处理器通信，以拓展功能。

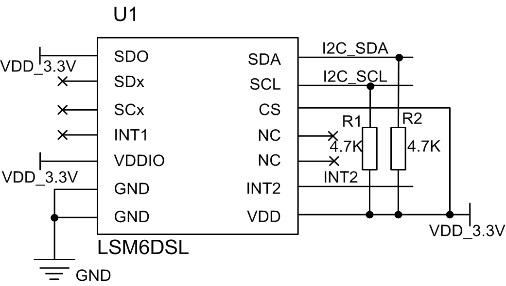


图5.7 LSM6DSL电路原理图

LSM6DSL传感器支持I2C和SPI通信（硬件I/O选择）。以使用I2C通信为例，只需使用SDA和SCL两根信号线，SDO引脚用于地址选择，从机地址为110101x。该引脚接高电平，则从机地址为0x6B，该引脚接低，则从机地址为0x6A。

以SDO引脚接高为例，该器件从机地址为0x6B，将传感器与ZLG116芯片连接通信，电路图详见图5.7。

### LSM6DSL初始化

AMetal提供了LSM6DSL的驱动，在使用LSM6DSL前，必须完成LSM6DSL初始化操作，以获取到对应的操作句柄，进而使用LSM6DSL的各种功能，初始化函数的原型（am\_lsm6dsl.h）为：

am\_lsm6dsl\_handle\_t am\_lsm6dsl\_init (

am\_lsm6dsl\_dev\_t \*p\_dev,

const am\_lsm6dsl\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t i2c\_handle);

该函数意在获取LSM6DSL传感器的初始化实例句柄，初始化完成后，即可使用通用接口读取应用数据，其中：

* p\_dev为指向am\_lsm6dsl\_dev\_t类型实例的指针；
* p\_devinfo为指向am\_lsm6dsl\_devinfo\_t类型实例信息的指针。

1. **实例**

定义am\_lsm6dsl\_dev\_t类型（am\_lsm6dsl.h）实例如下：

am\_local am\_lsm6dsl\_dev\_t \_\_g\_lsm6dsl\_dev;

其中，am\_lsm6dsl\_dev为用户自定义的实例，为p\_dev提供实参传递。

1. **实例信息**

实例信息主要描述了与LSM6DSL相关的信息，即LSM6DSL的配置信息等，其类型am\_lsm6dsl\_devinfo的定义（am\_lsm6dsl.h）如下：

typedef struct am\_lsm6dsl\_devinfo {

uint8\_t ctrl1\_xl; /\*\*< \brief 传感器状态配置 \*/

uint8\_t status\_reg; /\*\*< \brief 状态寄存器 \*/

uint8\_t fifo\_odr; /\*\*< \brief FIFO输出速率 \*/

} am\_lsm6dsl\_devinfo\_t;

其中，ctrl1\_xl为工作模式配置寄存器，配置数据输出速率、带宽及功率模式；status\_reg为数据读取状态寄存器，该位寄存器状态位表征温度传感器、陀螺仪和加速度计输出端是否有可用采集数据，fifo\_odr设置FIFO输出速率，最小输出速率为12.5HZ，最高输出速率可设置为6.66KHZ。

基于上述信息，实例信息如下：

/\*\* \brief 设备信息 \*/

const am\_local am\_const am\_lsm6dsl\_devinfo\_t \_\_g\_lsm6dsl\_devinfo = {

CTRL1\_XL, /\*\*< \brief CTRL1\_XL地址 \*/

STATUS\_REG, /\*\*< \brief STATUS\_REG地址 \*/

FIFO\_ODR\_12point5 /\*\*< \brief FIFO输出速率为12.5Hz \*/

};

其中，\_\_g\_lsm6dsl\_devinfo为用户自定义的实例信息，其地址作为am\_lsm6dsl\_dev\_t的实参传递。

1. **I2C句柄i2c\_handled**

使用ZLG116的I2C与LSM6DSL通信，则通过ZLG116的I2C实例初始化函数am\_zlg116\_i2c1\_inst\_init()获得句柄。即：

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

I2C句柄即可直接作为i2c\_handled的实参传递。

1. **实例句柄**

LSM6DSL初始化函数am\_lsm6dsl\_inst\_init ()的返回值LSM6DSL实例的句柄，作为其他功能接口（温度数据读取、加速度数据读取、陀螺仪数据读取）的实参。

其类型am\_lsm6dsl\_inst\_init （am\_lsm6dsl.h）定义如下；

typedef am\_lsm6dsl\_dev\_t \*am\_lsm6dsl\_handle\_t;

返回值为NULL，说明初始化失败；若返回值不为NULL，说明返回了有效的handle。

基于模块化编程思想，将初始化相关的实例、实例信息等的定义存放到对应的配置文件中，通过头文件引出实例初始化函数接口，源文件和头文件的程序范例分别详见程序清单5.48和程序清单 5.49。

程序清单5.48 实例初始化函数范例程序

#include "ametal.h"

#include "am\_lsm6dsl.h"

#include "am\_zlg116\_inst\_init.h"

#include "zlg116\_pin.h"

const am\_local am\_const am\_lsm6dsl\_devinfo\_t \_\_g\_lsm6dsl\_devinfo = { //lsm6dsl实例信息

CTRL1\_XL, //lsm6dsl状态配置

STATUS\_REG, //lsm6dsl数据状态

FIFO\_ODR\_12point5 //lsm6dsl数据速率

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LSM6DSL 实例初始化

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* \brief 设备实例定义 \*/

am\_local am\_lsm6dsl\_dev\_t \_\_g\_lsm6dsl\_dev;

/\*\* \brief 实例初始化 \*/

am\_lsm6dsl\_handle\_t am\_lsm6dsl\_inst\_init (void)

{

/\* 获取 I2C 实例句柄 \*/

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

return am\_lsm6dsl\_init (&\_\_g\_lsm6dsl\_dev,

&\_\_g\_lsm6dsl\_devinfo,

i2c\_handle);

}

程序清单 5.49 实例初始化函数接口（am\_hwconf\_lsm6dsl.h）

#include "ametal.h"

#include "am\_lsm6dsl.h"

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*\* \brief 实例初始化 \*/

am\_lsm6dsl\_handle\_t am\_lsm6dsl\_inst\_init (void);

后续只需使用无参数的实例初始化函数，即可获取到LSM6DSL的实例句柄。即：

am\_lsm6dsl\_handle\_t am\_lsm6dsl\_inst\_init (void);

### LSM6DSL接口函数

LSM6DSL功能较为丰富，使用接口函数的功能较为简单，接口函数详见表5.16。下面具体介绍各接口函数的使用方法。

表5.16 LSM6DSL接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_lsm6dsl\_read\_accel(  am\_lsm6dsl\_handle\_t handle,  int16\_t\* p\_accel) | 加速度读取 |
| int am\_lsm6dsl\_read\_temp(  am\_lsm6dsl\_handle\_t handle,  int16\_t\* p\_temp) | 温度读取 |

1. **三轴加速度读取**

读取三轴加速度参数，当实例初始化完成获取句柄后，获取加速度参数并做处理，实例初始化如下所示：

am\_lsm6dsl\_handle\_t handle;

handle = am\_lsm6dsl\_inst\_init();

当初始化完成，即可使用接口函数读取当前加速度信息，调用该函数，即可读取加速度信息，详见程序清单5.50。

程序清单5.50 使用LSM6DSL获取当前加速度

/\*\*

\* \brief 读取lsm6dsl三轴加速度

\* \return AM\_OK:成功 OTHER：失败

\*/

int am\_lsm6dsl\_read\_accel(am\_lsm6dsl\_handle\_t handle, int16\_t\* p\_accel)

{

int ret = AM\_OK;

/\* 读取X轴角速率 \*/

ret = \_\_lsm6dsl\_read\_reg(handle, LSM6DSL\_OUTX\_L\_XL, &p\_accel[0]);

/\* 检查读取是否失败 \*/

if (ret != AM\_OK) {

return ret;

}

/\* 读取Y轴加速度 \*/

ret = \_\_lsm6dsl\_read\_reg(handle, LSM6DSL\_OUTY\_L\_XL, &p\_accel[1]);

/\* 检查读取是否失败 \*/

if (ret != AM\_OK) {

return ret;

}

/\* 读取Z轴加速度 \*/

ret = \_\_lsm6dsl\_read\_reg(handle, LSM6DSL\_OUTZ\_L\_XL, &p\_accel[2]);

/\* 检查读取是否失败 \*/

if (ret != AM\_OK) {

return ret;

}

return ret;

}

1. **温度信息读取**

读取温度信息参数，当实例初始化完成获取句柄后，获取温度参数并做处理，实例初始化如下所示：

am\_lsm6dsl\_handle\_t handle;

handle = am\_lsm6dsl\_inst\_init();

当初始化完成，即可使用接口函数读取当前温度信息，调用该函数，即可读取当前温度信息，详见程序清单5.51。

程序清单5.51 使用LSM6DSL获取当前温度

int am\_lsm6dsl\_read\_temp(am\_lsm6dsl\_handle\_t handle, int16\_t\* p\_temp)

{

Int ret = AM\_OK;

//读取温度

ret = \_\_lsm6dsl\_read\_reg(handle, LSM6DSL\_OUT\_TEMP\_L, p\_temp);

//检查读取是否失败

if (ret != AM\_OK) {

return ret;

}

//计算温度

\*p\_temp = (int8\_t) (24 + \*p\_temp \* 1.0 / 2);

return ret;

}

1. **使用范例**

当完成初始化后，即可使用通用的加速度读取接口和温度读取接口获取当前加速度及温度值，读取并通过串口打印当前加速度及温度信息的范例程序详见程序清单5.52。

程序清单5.52 使用LSM6DSL范例程序

#include "ametal.h"

#include "am\_delay.h"

#include "am\_lsm6dsl.h"

#include "am\_hwconf\_lsm6dsl.h"

#include "am\_vdebug.h"

void demo\_lsm6dsl\_entry (void)

{

int16\_t temp = 0;

int16\_t accel[3] = {0};

am\_lsm6dsl\_handle\_t handle;

handle = am\_lsm6dsl\_inst\_init();

while (1) {

/\*\* \brief 读取X、Y、Z通道的加速度 \*/

am\_lsm6dsl\_read\_accel (handle, accel);

AM\_DBG\_INFO(" x\_accel = %d\r\n y\_accel = %d\r\n z\_accel = %d\r\n",

accel [0], accel [1], accel [2] );

/\*\* \brief 读取温度 \*/

am\_lsm6dsl\_read\_temp (handle, &temp);

AM\_DBG\_INFO(" temp = %d\r\n",temp);

am\_mdelay (1000);

}

}

## BMG160传感器组件

BMG160是BOSCH半导体推出的具有I2C和SPI接口的数字三轴陀螺仪。

### 器件介绍

BMG160是一款三轴陀螺仪传感器，旨在满足消费类应用的要求，例如图像稳定（DSC和拍照手机）、游戏和定点设备。它能够测量三轴（x轴，y轴和z轴）的角速率，并提供相应的输出信号。BMG160配有数字双向SPI和I2C接口，可实现最佳系统集成。BMG160提供1.2V至3.6V的可变VDDIO电压范围，可进行编程以优化客户特定应用中的功能、性能和功耗。此外，它还具有片上中断控制器，可在不使用微控制器的情况下实现基于运动的应用。BMG160原理图详见图5.8。

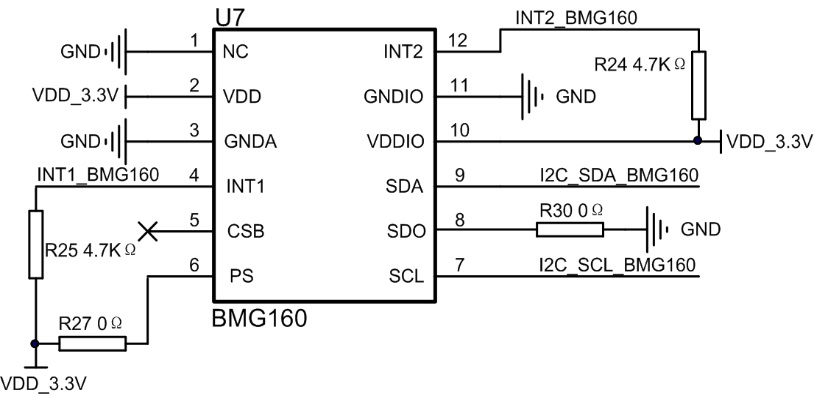


图5.8 BMG160原理图

### BMG160初始化

AMetal提供了BMG160的驱动，在使用BMG160前，必须完成BMG160初始化操作，以获取到对应的操作句柄，进而使用BMG160的各种功能，其函数原型（am\_bmg160.h）为：

am\_bmg160\_handle\_t am\_bmg160\_init (am\_bmg160\_dev\_t \*p\_dev,

const am\_bmg160\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t i2c\_handle);

该函数意在获取BMG160三轴陀螺仪传感器的实例句柄，进而使用通用接口读取角速率。其中：

* p\_dev为指向am\_bmg160\_dev\_t类型实例的指针；
* p\_devinfo为指向am\_bmg160\_devinfo\_t类型实例信息的指针。

1. **实例**

定义am\_bmg160\_dev\_t类型（am\_bmg160.h）实例如下：

/\*\*

\* \brief BMG160 设备结构体

\*/

typedef struct am\_bmg160\_dev {

am\_i2c\_device\_t i2c\_dev; /\*\*< \brief BMG160 I2C设备 \*/

} am\_bmg160\_dev\_t;

typedef am\_bmg160\_dev\_t \*am\_bmg160\_handle\_t; /\*\* \brief BMG160操作句柄定义 \*/

其中，am\_bmg160\_handle\_t为用户自定义的实例句柄，其地址作为p\_dev的实参传递。

1. **实例信息**

实例信息主要描述了与BMG160相关的信息，即BMG160的加速度范围、数据滤波带宽、电源模式和低功耗模式睡眠周期等，其类型am\_bmg160\_devinfo\_t的定义（am\_bmg160.h）如下：

typedef struct am\_bmg160\_devinfo {

uint8\_t range; /\*\*< \brief 角速率范围 \*/

uint8\_t bw; /\*\*< \brief 数据滤波带宽 \*/

uint8\_t power\_mode; /\*\*< \brief 电源模式 \*/

uint8\_t sleep\_period; /\*\*< \brief 低功耗模式睡眠周期 \*/

} am\_bmg160\_devinfo\_t;

其中，range配置角速率范围，可以在±125°/s至±2000°/s之间配置；bw配置数据滤波带宽,可以在32Hz至230Hz之间配置；power\_mode配置电源模式，可以配置为正常、挂起和深度挂起模式；sleep\_period配置低功耗睡眠周期，可以配置2-20ms的周期。

基于上述信息，实例信息如下：

/\*\* \brief 设备信息 \*/

am\_local am\_const am\_bmg160\_devinfo\_t \_\_g\_bmg160\_devinfo = {

RANGE\_125, /\*\*< \brief 角速率范围±125°/s \*/

BW\_230Hz, /\*\*< \brief 数据滤波带宽230Hz \*/

NORMAL\_MODE, /\*\*< \brief 正常模式 \*/

SLEEP\_DUR\_2MS /\*\*< \brief 配置的睡眠周期为2ms \*/

};

其中，\_\_g\_bmg160\_devinfo为用户自定义的实例信息，其地址作为am\_bmg160\_dev\_t的实参传递。

1. **I2C句柄i2c\_handle**

使用ZLG116的I2C与BMG160通信，则通过ZLG116的I2C实例初始化函数am\_zlg116\_i2c1\_inst\_init()获得句柄。即：

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

I2C句柄即可直接作为i2c\_handle的实参传递。

1. **实例句柄**

BMG160初始化函数am\_bmg160\_inst\_init()的返回BMG160实例的句柄，作为其他功能函数（温度数据读取、三轴角速率数据读取）的实参。

其类型am\_bmg160\_inst\_init(am\_bmg160.h)定义如下：

typedef am\_bmg160\_dev\_t \*am\_bmg160\_handle\_t;

返回值为NULL，说明初始化失败；若返回值不为NULL，说明返回了有效的handle。

基于模块化编程思想，将初始化相关的实例、实例信息等的定义存放到对应的配置文件中，通过头文件引出实例初始化函数接口，源文件和头文件的程序范例分别详见程序清单5.53和程序清单5.54。

程序清单5.53 实例初始化函数范例程序

#include "ametal.h"

#include "am\_bmg160.h"

#include "am\_zlg116\_inst\_init.h"

#include "zlg116\_pin.h"

am\_local am\_const am\_bmg160\_devinfo\_t \_\_g\_bmg160\_devinfo = {

RANGE\_125, /\*\*< \brief 角速率范围±125°/s \*/

BW\_230Hz, /\*\*< \brief 数据滤波带宽230Hz \*/

NORMAL\_MODE, /\*\*< \brief 正常模式 \*/

SLEEP\_DUR\_2MS /\*\*< \brief 配置的睡眠周期为2ms \*/

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BMG160 实例初始化

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* \brief 设备实例定义 \*/

am\_local am\_bmg160\_dev\_t \_\_g\_bmg160\_dev;

/\*\* \brief 实例初始化 \*/

am\_bmg160\_handle\_t am\_bmg160\_inst\_init (void)

{

/\* 获取 I2C 实例句柄 \*/

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

return am\_bmg160\_init (&\_\_g\_bmg160\_dev,

&\_\_g\_bmg160\_devinfo,

i2c\_handle);

}

程序清单5.54 实例初始化函数接口（am\_hwconf\_bmg160.h）

#include "ametal.h"

#include "am\_bmg160.h"

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*\* \brief 实例初始化 \*/

am\_bmg160\_handle\_t am\_bmg160\_inst\_init (void);

后续只需使用无参数的实例初始化函数，即可获取到BMG160的实例句柄。即：

am\_bmg160\_handle\_t am\_bmg160\_inst\_init(void);

### BMG160接口函数

BMG160功能较为丰富，使用接口函数的功能较为简单，详见表5.17，下面具体介绍各接口函数的使用方法。

表5.17 BMG160接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_bmg160\_read\_temp (  am\_bmg160\_handle\_t handle,  int8\_t \*p\_temp) | 温度读取 |
| int am\_bmg160\_read\_rate(  am\_bmg160\_handle\_t handle,  int16\_t \*p\_rate) | 角速率读取 |

1. **温度信息读取**

读取温度信息参数，当实例初始化完成获取句柄后，获取温度信息参数并做处理，实例初始化如下所示：

am\_bmg160\_handle\_t handle;

handle = am\_bmg160\_inst\_init();

当初始化完成，即可使用接口函数读取当前温度信息，调用该函数，即可读取温度信息，详见程序清单5.55。

程序清单5.55 使用BMG160获取当前温度

/\*\*

\* \brief 读取bmg160温度

\* \return AM\_OK:成功 OTHER：失败

\*/

int am\_bmg160\_read\_temp (am\_bmg160\_handle\_t handle, int8\_t\* p\_temp)

{

int ret = AM\_OK;

/\* I2C设备指针 \*/

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

ret = am\_i2c\_read(p\_i2c\_dev,BMG160\_TEMP,(uint8\_t\*)(p\_temp), 1);

/\* 检查发送是否失败 \*/

if (ret != AM\_OK) {

return ret;

}

/\* 计算温度 \*/

\*p\_temp = (int8\_t)(24 + \*p\_temp \* 1.0 / 2);

return ret;

}

1. **三轴角速率读取**

读取三轴角速率信息参数，当实例初始化完成获取句柄后，获取角速率参数并做处理，实例初始化如下所示：

am\_bmg160\_handle\_t handle;

handle = am\_bmg160\_inst\_init();

当初始化完成，即可使用接口函数读取当前角速率信息，调用该函数，即可读取当前三轴角速率信息，详见程序清单5.56。

程序清单5.56 使用BMG160获取当前三轴角速率

/\*\*

\* \brief 读取bmg160三轴陀螺仪

\* \return AM\_OK:成功 OTHER：失败

\*/

int am\_bmg160\_read\_rate(am\_bmg160\_handle\_t handle, int16\_t\* p\_rate)

{

int ret = AM\_OK;

/\* 读取X轴角速率 \*/

ret = \_\_bmg160\_read\_rate\_reg(handle, BMG160\_RATE\_X\_LSB, &p\_rate[0]);

/\* 检查读取是否失败 \*/

if (ret != AM\_OK) {

return ret;

}

/\* 读取Y轴加速度 \*/

ret = \_\_bmg160\_read\_rate\_reg(handle, BMG160\_RATE\_Y\_LSB, &p\_rate[1]);

/\* 检查读取是否失败 \*/

if (ret != AM\_OK) {

return ret;

}

/\* 读取Z轴加速度 \*/

ret = \_\_bmg160\_read\_rate\_reg(handle, BMG160\_RATE\_Z\_LSB, &p\_rate[2]);

/\* 检查读取是否失败 \*/

if (ret != AM\_OK) {

return ret;

}

return ret;

}

1. **使用范例**

当完成初始化后，即可使用通用的温度读取接口和三轴角速率读取接口获取当前温度及角速率值，读取并通过串口打印当前温度及角速率的范例程序详见程序清单5.57。

程序清单5.57 使用BMG160范例程序

#include "ametal.h"

#include "am\_delay.h"

#include "am\_bmg160.h"

#include "am\_hwconf\_bmg160.h"

#include "am\_vdebug.h"

void demo\_bmg160\_entry (void)

{

int8\_t temp = 0;

int16\_t rate[3] = {0};

am\_bmg160\_handle\_t handle = am\_bmg160\_inst\_init();

while (1) {

/\*\* \brief 读取温度 \*/

am\_bmg160\_read\_temp(handle, &temp);

AM\_DBG\_INFO("temperature = %d\r\n", temp);

/\*\* \brief 读取X、Y、Z通道的角速率 \*/

am\_bmg160\_read\_rate(handle, rate);

AM\_DBG\_INFO(" x\_rate = %d\r\n y\_rate = %d\r\n z\_rate = %d\r\n",

rate[0],rate[1],rate[2]);

am\_mdelay(1000);

}

}

## BME280传感器组件

### 器件介绍

BME280是BOSCH半导体推出的具有I2C和SPI接口的压力温湿度传感器。BME280是基于成熟传感原理的数字湿度、压力和温度传感器组合。传感器模块采用极其紧凑的金属盖LGA封装，体积小、功耗低、性能高，可用于手机、GPS模块或手表等电池驱动设备。BME280原理图详见图5.9。

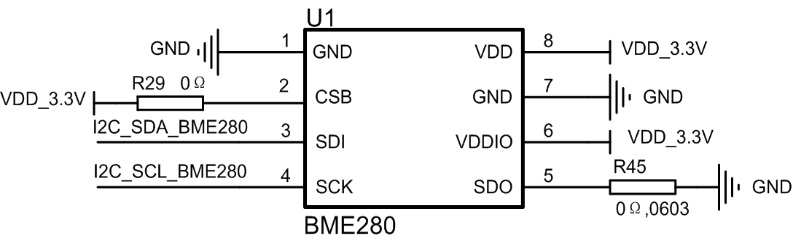


图5.9 BME280原理图

### BME280初始化

AMetal提供了BME280的驱动，在使用BME280前，必须完成BME280初始化操作，以获取到对应的操作句柄，进而使用BME280的各种功能，其函数原型（am\_sensor\_bme280.h）为：

am\_sensor\_handle\_t am\_sensor\_bme280\_init (

am\_sensor\_bme280\_dev\_t \*p\_dev,

const am\_sensor\_bme280\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t handle);

该函数意在获取BME280传感器的初始化实例句柄，初始化完成后，即可使用传感器标准接口读取应用数据，其中：

* p\_dev为指向am\_sensor\_bme280\_dev\_t类型实例的指针；
* p\_devinfo为指向am\_sensor\_bme280\_devinfo\_t类型实例信息的指针；
* handle为I2C初始化时获取的句柄。

1. **实例**

定义am\_sensor\_bme280\_dev\_t类型（am\_hwconf\_sensor\_bme280.c）实例如下：

am\_local struct am\_sensor\_bme280\_dev \_\_g\_bme280\_dev;

\_\_g\_bme280\_dev为用户自定义的示例，为p\_dev提供实参传递。

1. **实例信息**

实例信息主要描述了与BME280相关的信息，其类型am\_sensor\_bme280\_devinfo的定义（am\_sensor\_bme280.h）如下：

typedef struct am\_sensor\_bme280\_devinfo {

uint8\_t i2c\_addr; // I2C 7位设备地址

} am\_sensor\_bme280\_devinfo\_t;

i2c\_addr为BME280的7位从机地址。

基于上述信息，实例信息如下：

am\_const am\_local struct am\_sensor\_bme280\_devinfo \_\_g\_bme280\_info = {

0x76 // bme280 I2C地址

};

其中，\_\_g\_bme280\_devinfo为用户自定义的实例信息，其地址作为am\_sensor\_bme280 \_ dev\_t的实参传递。

1. **I2C句柄handle**

使用ZLG116的I2C与BME280通信，则通过ZLG116的I2C实例初始化函数am\_zlg116\_i2c1\_inst\_init()获取句柄。即：

am\_i2c\_handle\_t handle = am\_zlg116\_i2c1\_inst\_init();

I2C句柄即可直接作为handle的实参传递。

1. **实例句柄**

BME280初始化函数am\_sensor\_bme280\_inst\_init ()的返回值BME280实例的句柄，作为传感器标准接口的实参。

其类型am\_sensor\_bme280\_inst\_init(am\_hwconf\_sensor\_bme280.h)定义如下：

am\_sensor\_handle\_t am\_sensor\_bme280\_inst\_init (void);

返回值为NULL，说明初始化失败；若返回值不为NULL，说明返回了有效的handle。

基于模块化编程思想，将初始化相关的实例、实例信息等的定义存放到对应的配置文件中，通过头文件引出实例初始化函数接口，源文件和头文件的程序范例分别详见程序清单5.58和程序清单5.59。

程序清单5.58 实例初始化函数范例程序

#include "am\_sensor\_bme280.h"

#include "am\_common.h"

#include "zlg116\_pin.h"

#include "am\_zlg116\_inst\_init.h"

am\_const am\_local struct am\_sensor\_bme280\_devinfo \_\_g\_bme280\_info = {

0x76

};

am\_local struct am\_sensor\_bme280\_dev \_\_g\_bme280\_dev;

am\_sensor\_handle\_t am\_sensor\_bme280\_inst\_init (void)

{

return am\_sensor\_bme280\_init(&\_\_g\_bme280\_dev,

&\_\_g\_bme280\_info,

am\_zlg116\_i2c1\_inst\_init());

}

程序清单5.59 实例初始化函数接口（am\_hwconf\_sensor\_bme280.h）

#include "ametal.h"

#include "am\_sensor.h"

am\_sensor\_handle\_t am\_sensor\_bme280\_inst\_init (void);

后续只需使用无参数的实例初始化函数，即可获取到BME280的实例句柄。即：

am\_sensor\_handle\_t handle = am\_sensor\_bme280\_inst\_init (void);

### BME280获取数据

BME280初始化后，使用传感器标准接口配置参数、获取BME280的数据，传感器标准接口函数详见表5.18，下面具体介绍标准接口函数的使用方法。

表5.18 传感器标准接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| am\_static\_inline am\_err\_t am\_sensor\_enable (  am\_sensor\_handle\_t handle,  const int \*p\_ids,  int num,  am\_sensor\_val\_t \*p\_result); | 传感器通道使能 |
| am\_static\_inline am\_err\_t am\_sensor\_data\_get (  const int \*p\_ids,  int num,  am\_sensor\_val\_t \*p\_buf); | 传感器通道数据获取 |
| am\_err\_t am\_sensor\_val\_unit\_convert (  am\_sensor\_val\_t \*p\_buf,  int num,  int32\_t to\_unit); | 传感器通道数据单位转换 |

#### 使能通道

BME280传感器有多个通道，支持压力、温度、湿度测量，通过调用传感器通道使能接口函数可以使能指定的通道。使能传感器通道的接口函数原型为：

am\_static\_inline am\_err\_t am\_sensor\_enable (am\_sensor\_handle\_t handle,

const int \*p\_ids,

int num,

am\_sensor\_val\_t \*p\_result);

其中，handle为BME280的实例句柄，p\_ids为指向通道ID的指针，num为使能的通道数，p\_result为指向用于存放结果的地址。

如果返回AM\_OK，说明使能成功，反之失败。假设使能三个通道，范例程序详见程序清单5.60。

程序清单5.60 使能通道范例程序

const int id[3] = {AM\_BME280\_CHAN\_1, AM\_BME280\_CHAN\_2, AM\_BME280\_CHAN\_3};

am\_sensor\_val\_t data[3];

am\_sensor\_enable(handle, id, 3, data);

#### 获取通道数据

使能传感器通道后，可调用传感器标准接口获取传感器数据，获取数据的接口函数原型为：

am\_static\_inline am\_err\_t am\_sensor\_data\_get (am\_sensor\_handle\_t handle,

const int \*p\_ids,

int num,

am\_sensor\_val\_t \*p\_buf);

其中，handle为BME280的实例句柄，p\_ids为指向通道ID的指针，num为使能的通道数，p\_ buf为指向用于存放结果的地址。

如果返回AM\_OK，说明获取数据成功，反之失败。假设获取三个通道的数据，范例程序详见程序清单5.61。

程序清单5.61 获取通道数据范例程序

am\_sensor\_data\_get(handle, id, 3, data);

#### 数据单位转换

获取的通道数据是带有单位的，用户可以根据需要调用单位转换接口函数进行单位转换。数据单位转换接口函数原型为：

am\_err\_t am\_sensor\_val\_unit\_convert (am\_sensor\_val\_t \*p\_buf,

int num,

int32\_t to\_unit);

将数据单位转换为微为例，范例程序详见程序清单5.62。

程序清单5.62 数据单位转换范例程序

am\_sensor\_val\_unit\_convert(&data[1], 1, AM\_SENSOR\_UNIT\_MICRO);

#### 范例程序

范例程序的实现详见程序清单5.63。

程序清单5.63 范例程序实现

#include "ametal.h"

#include "am\_vdebug.h"

#include "am\_sensor.h"

#include "am\_sensor\_bme280.h"

#include "am\_delay.h"

void demo\_std\_bme280\_entry (am\_sensor\_handle\_t handle)

{

const int id[3] = {AM\_BME280\_CHAN\_1, AM\_BME280\_CHAN\_2, AM\_BME280\_CHAN\_3};

am\_sensor\_val\_t data[3];

const char \*data\_name\_string[] = {"pressure", "temperature", "humidity"};

const char \*data\_unit\_string[] = {"Pa", "℃", "%rH"};

am\_sensor\_enable(handle, id, 3, data);

while(1) {

am\_sensor\_data\_get(handle, id, 3, data);

if (AM\_SENSOR\_VAL\_IS\_VALID(data[0])) {

am\_kprintf("The %s is : %d %s.\r\n", data\_name\_string[0],

(data[0].val),

data\_unit\_string[0]);

} else {

am\_kprintf("The %s get failed!\r\n", data\_name\_string[0]);

}

if (AM\_SENSOR\_VAL\_IS\_VALID(data[1])) {

am\_sensor\_val\_unit\_convert(&data[1], 1, AM\_SENSOR\_UNIT\_MICRO);

am\_kprintf("The %s is : %d.%06d %s.\r\n", data\_name\_string[1],

(data[1].val)/1000000,

(data[1].val)%1000000,

data\_unit\_string[1]);

} else {

am\_kprintf("The %s get failed!\r\n", data\_name\_string[1]);

}

if (AM\_SENSOR\_VAL\_IS\_VALID(data[2])) {

am\_sensor\_val\_unit\_convert(&data[2], 1, AM\_SENSOR\_UNIT\_MICRO);

am\_kprintf("The %s is : %d.%06d %s.\r\n", data\_name\_string[2],

(data[2].val)/1000000,

(data[2].val)%1000000,

data\_unit\_string[2]);

} else {

am\_kprintf("The %s get failed!\r\n", data\_name\_string[2]);

}

am\_mdelay(1000);

}

}

范例程序详见程序清单5.64。

程序清单5.64 范例程序

#include "ametal.h"

#include "am\_led.h"

#include "am\_delay.h"

#include "am\_vdebug.h"

#include "am\_board.h"

#include "demo\_am116\_core\_entries.h"

#include "demo\_std\_entries.h"

#include "am\_hwconf\_sensor\_bme280.h"

int am\_main (void)

{

am\_sensor\_handle\_t handle = am\_sensor\_bme280\_inst\_init ();

demo\_std\_bme280\_entry (handle);

}

## BMP280传感器组件

BMP280是BOSCH半导体推出的具有I2C和SPI接口的数字气压温度传感器。

### 器件介绍

BMP280基于博世经过验证的压阻式压力传感器技术，具有高EMC稳健性，高精度和线性长期稳定性。BMP280是一款专为移动设备设计的绝对气压传感器应用。传感器模块采用极其紧凑的8针金属盖LGA封装。它很小的尺寸和2.7μA@1Hz的低功耗使其允许在电池中实现移动设备，如手机，GPS模块或手表。BMP280原理图详见图5.10。

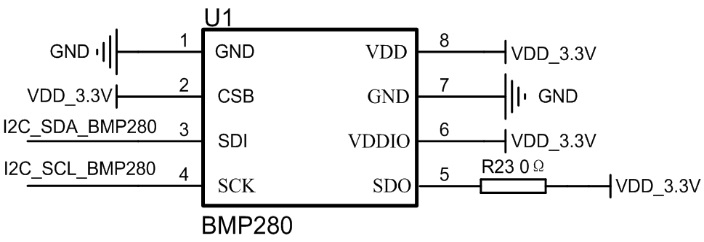


图5.10 BMP原理图

### BMP280初始化

AMetal提供了BMP280的驱动，在使用BMP280前，必须完成BMP280初始化操作，以获取到对应的操作句柄，进而使用BMP280的各种功能，其函数原型（am\_bmp280.h）为：

am\_bmp280\_handle\_t am\_bmp280\_init ( am\_bmp280\_dev\_t \*p\_dev,

const am\_bmp280\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t i2c\_handle);

该函数意在获取BMP280传感器的初始化实例句柄，初始化完成后，即可使用通用接口读取应用数据，其中：

* p\_dev为指向am\_bmp280\_dev\_t类型实例的指针；
* p\_devinfo为指向am\_bmp280\_devinfo\_t类型实例信息的指针。

1. **实例**

定义am\_bmp280\_dev\_t类型（am\_hwconf\_bmp280.c）实例如下：

am\_local am\_bmp280\_dev\_t \_\_g\_bmp280\_dev;

其中，\_\_g\_bmp280\_dev为用户自定义的示例，为p\_dev提供实参传递。

1. **实例信息**

实例信息主要描述了与BMP280相关的信息，即BMP280的配置信息等，其类型am\_bmp280\_devinfo的定义（am\_bmp280.h）如下：

typedef struct am\_bmp280\_devinfo {

uint8\_t press\_oversampe; /\*\*< \brief 压力过采样样本数 \*/

uint8\_t temp\_oversampe; /\*\*< \brief 温度过采样样本数 \*/

uint8\_t power\_mode; /\*\*< \brief 电源模式 \*/

uint8\_t standby\_time; /\*\*< \brief 配置待机时间 \*/

} am\_bmp280\_devinfo\_t;

其中，press\_oversampe为压力过采样样本数，temp\_oversampe为温度过采样样本数，它们最小过采样样本数为1，最大过采样样本数可配置为16；power\_mode为电源模式，可配置为睡眠、强制、正常模式；standby\_time为待机时间，最小待机时间为0.5ms,最大待机时间可配置为4000ms。

基于上述信息，实例信息如下：

/\*\* \brief 设备信息 \*/

am\_local am\_const am\_bmp280\_devinfo\_t \_\_g\_bmp280\_devinfo = {

PRESSURE\_OVERESAMPLE\_X16,

TEMP\_OVERESAMPLE\_X16,

NORMAL\_MODE,

STANDBY\_TIME\_HALF\_MS

};

其中，\_\_g\_bmp280\_devinfo为用户自定义的实例信息，其地址作为am\_bmp280\_dev\_t的实参传递。

1. **I2C句柄i2c\_handle**

使用ZLG116的I2C与BMP280通信，则通过ZLG116的I2C实例初始化函数am\_zlg116\_i2c1\_inst\_init()获取句柄。即：

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

I2C句柄即可直接作为i2c\_handled的实参传递。

1. **实例句柄**

BMP280初始化函数am\_bmp280\_inst\_init()的返回值BMP280实例的句柄，作为其他功能接口（温度数据读取、气压数据读取）的实参。

其类型am\_bmp280\_inst\_init(am\_bmp280.h)定义如下：

typedef am\_bmp280\_dev\_t \*am\_bmp280\_handle\_t;

返回值为NULL，说明初始化失败；若返回值不为NULL，说明返回了有效的handle。

基于模块化编程思想，将初始化相关的实例、实例信息等的定义存放到对应的配置文件中，通过头文件引出实例初始化函数接口，源文件和头文件的程序范例分别详见程序清单5.64和程序清单5.65。

程序清单5.65 实例初始化函数范例程序

#include "ametal.h"

#include "am\_bmp280.h"

#include "am\_zlg116\_inst\_init.h"

#include "zlg116\_pin.h"

am\_local am\_const am\_bmp280\_devinfo\_t \_\_g\_bmp280\_devinfo = {

PRESSURE\_OVERESAMPLE\_X16,

TEMP\_OVERESAMPLE\_X16,

NORMAL\_MODE,

STANDBY\_TIME\_HALF\_MS

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BMA253 实例初始化

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* \brief 设备定义 \*/

am\_local am\_bmp280\_dev\_t \_\_g\_bmp280\_dev;

/\*\* \brief 实例初始化 \*/

am\_bmp280\_handle\_t am\_bmp280\_inst\_init (void)

{

/\* 获取 I2C 实例句柄 \*/

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

return am\_bmp280\_init (&\_\_g\_bmp280\_dev,

&\_\_g\_bmp280\_devinfo,

i2c\_handle);

}

程序清单5.66 实例初始化函数接口(am\_hwconf\_bmp280.h)

#include "ametal.h"

#include "am\_bmp280.h"

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*\* \brief 实例初始化 \*/

am\_bmp280\_handle\_t am\_bmp280\_inst\_init (void);

后续只需要使用无参数的实例初始化函数，即可获取BMP280的实例句柄。即：

am\_bmp280\_handle\_t am\_bmp280\_inst\_init (void);

### BMP280接口函数

BMP280功能较为丰富，使用接口函数的功能较为简单，详见表5.19，下面具体介绍各接口函数的使用方法。

表5.19 BMP280接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_bmp280\_read\_temp(  am\_bmp280\_handle\_t handle,  int8\_t \*temp) | 温度读取 |
| int am\_bmp280\_read\_press (  am\_bmp280\_handle\_t handle,  uint32\_t \*press) | 气压读取 |

1. **温度信息读取**

读取温度信息参数，当实例初始化完成获取句柄后，获取温度信息参数并做处理，实例初始化如下所示：

am\_bmp280\_handle\_t handle;

handle = am\_bmp280\_inst\_init();

当初始化完成，即可使用接口函数读取当前温度信息，调用该函数，即可读取温度信息，详见程序清单5.66。

程序清单5.67 使用BMP280获取当前温度

/\*\*

\* \brief 读取BMP280温度

\*/

int am\_bmp280\_read\_temp (am\_bmp280\_handle\_t handle, int8\_t \*temp)

{

int32\_t adc\_temp = 0;

int32\_t var1 = 0;

int32\_t var2 = 0;

int ret = AM\_OK;

/\* 校准参数指针 \*/

bmp280\_calib\_param\_t \*calib\_param = &(handle->calib\_param);

/\* 读取温度ADC \*/

ret = \_\_bmp280\_read\_temp\_reg(handle, &adc\_temp);

/\* 检查读取是否失败 \*/

if (ret != AM\_OK) {

return ret;

}

/\* 转换成实际温度值 \*/

var1 = (((double)adc\_temp) / 16384.0

- ((double) calib\_param->dig\_t1) / 1024.0)

\* ((double) calib\_param->dig\_t2);

var2 = ((((double)adc\_temp) / 131072.0

- ((double) calib\_param->dig\_t1) / 8192.0)

\* (((double)adc\_temp)

/ 131072.0 - ((double) calib\_param->dig\_t1)

/ 8192.0)) \* ((double) calib\_param->dig\_t3);

/\* 保存用于气压校准 \*/

calib\_param->t\_fine = var1 + var2;

\*temp = (var1 + var2) / 5120.0;

return ret;

}

1. **气压信息读取**

读取气压信息参数，当实例初始化完成获取句柄后，获取气压参数并做处理，实例初始化如下所示：

am\_bmp280\_handle\_t handle;

handle = am\_bmp280\_inst\_init();

当初始化完成，即可使用接口函数读取当前气压信息，调用该函数，即可读取气压信息，详见程序清单5.67。

程序清单5.68 使用BMP280获取当前气压

/\*\*

\* \brief 读取BMP280气压

\*/

int am\_bmp280\_read\_press (am\_bmp280\_handle\_t handle, uint32\_t \*press)

{

int32\_t adc\_press = 0;

int32\_t var1 = 0;

int32\_t var2 = 0;

int32\_t t\_fine = 0;

int ret = AM\_OK;

/\* 校准参数指针 \*/

bmp280\_calib\_param\_t \*calib\_param = &(handle->calib\_param);

t\_fine = calib\_param->t\_fine;

/\* 读取气压值ADC \*/

ret = \_\_bmp280\_read\_press\_reg(handle,&adc\_press);

/\* 检查读取是否失败 \*/

if (ret != AM\_OK) {

return ret;

}

/\* 转换成实际气压值 \*/

var1 = ((double)t\_fine / 2.0)-64000.0;

var2 = var1 \* var1 \* ((double)calib\_param->dig\_p6) / 32768.0;

var2 = var2 + var1 \* ((double)calib\_param->dig\_p5) \* 2.0;

var2 = (var2 / 4.0) + (((double)calib\_param->dig\_p4) \* 65536.0);

var1 = (((double)calib\_param->dig\_p3) \* var1 \* var1 / 524288.0

+ ((double)calib\_param->dig\_p2) \* var1) / 524288.0;

var1 = (1.0 + var1 / 32768.0) \* ((double)calib\_param->dig\_p1);

\*press = 1048576.0 - (double)adc\_press;

\*press = (\*press - (var2 / 4096.0)) \* 6250.0 / var1;

var1 = ((double)calib\_param->dig\_p9) \* (\*press) \* (\*press) / 2147483648.0;

var2 = (\*press) \* ((double)calib\_param->dig\_p8) / 32768.0;

\*press = \*press + (var1 + var2 + ((double)calib\_param->dig\_p7)) / 16.0;

return ret;

}

1. **使用范例**

当完成初始化后，即可使用通用的温度读取接口和气压读取接口获取当前温度及气压值，读取并通过串口打印当前温度及气压信息的范例程序详见程序清单5.68。

程序清单5.69 使用BMP280范例程序

#include "ametal.h"

#include "am\_delay.h"

#include "am\_bmp280.h"

#include "am\_hwconf\_bmp280.h"

#include "am\_vdebug.h"

/\*\*

\* \brief 例程入口

\*/

void demo\_bmp280\_entry (void)

{

am\_bmp280\_handle\_t handle;

int8\_t temp = 0;

uint32\_t press = 0;

handle = am\_bmp280\_inst\_init();

while (1) {

am\_bmp280\_read\_temp (handle, &temp);

am\_bmp280\_read\_press (handle, &press);

AM\_DBG\_INFO("temp = %d ℃ \r\n",temp);

AM\_DBG\_INFO("press = %d Pa \r\n",press);

am\_mdelay(500);

}

}

## BH1730FVC传感器组件

### 器件介绍

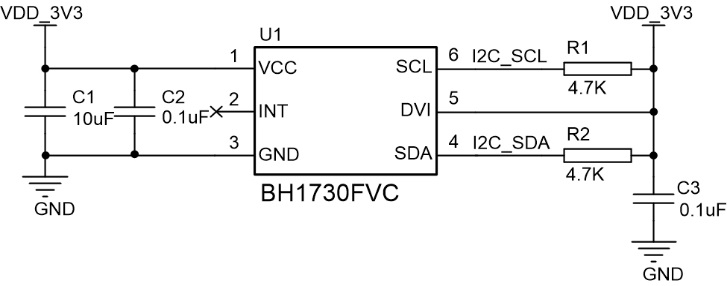


图5.11 BH1730FVC电路原理图

BH1730FVC是一款带I2C总线接口的数字环境光传感器IC，支持0.001 lux 到100k lux宽范围的亮度检测，数字直接输出亮度值，测量精度高，广泛用于手机、电脑、电视等需要自动调节屏幕亮度的设备上。BH1730FVC应用电路非常简单，使用I2C通信，具体连接方式详见图5.11。

### BH1730FVC初始化

AMetal提供了BH1730FVC的驱动，在使用BH1730FVC前，必须完成BH1730FVC初始化操作，以获取到对应的操作句柄，进而使用BH1730FVC的各种功能，其函数原型（am\_bh1730fvc.h）为：

am\_bh1730fvc\_handle\_t am\_bh1730fvc\_init (

am\_bh1730fvc\_dev\_t \*p\_dev,

const am\_bh1730fvc\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t i2c\_handle);

该函数意在获取BH1730FVC传感器的初始化实例句柄，初始化完成后，即可使用通用接口读取应用数据，其中：

* p\_dev为指向am\_bh1730fvc\_dev\_t类型实例的指针；
* p\_devinfo为指向am\_bh1730fvc\_devinfo\_t类型实例信息的指针；
* i2c\_handle为与BH1730FVC通信的I2C实例句柄。

1. **实例**

定义am\_bh1730fvc\_dev\_t类型（am\_hwconf\_bh1730fvc.c）实例如下：

am\_local am\_bh1730fvc\_dev\_t \_\_g\_bh1730fvc\_dev;

其中\_\_g\_bh1730fvc\_dev为用户自定义的实例，其地址作为p\_dev的实参传递。

1. **实例信息**

实例信息主要描述了与BH1730FVC相关的信息，即BH1730FVC的配置信息，如测量模式、数据类型、电源模式等，其类型am\_bh1730fvc\_devinfo的定义（am\_ bh1730fvc.h）如下：

typedef struct am\_bh1730fvc\_devinfo {

uint8\_t meas\_mode; /\*\*< \brief 配置测量模式 \*/

uint8\_t data\_sel; /\*\*< \brief 配置数据类型 \*/

uint8\_t adc\_en; /\*\*< \brief 配置测量状态 \*/

uint8\_t power\_mode; /\*\*< \brief 配置电源模式 \*/

uint8\_t integrate\_time; /\*\*< \brief 配置积分时间 \*/

uint8\_t adc\_res\_setting; /\*\*< \brief 配置ADC分辨率 \*/

} am\_bh1730fvc\_devinfo\_t;

其中meas\_mode用于配置测量模式，包括单次测量和连续测量；data\_sel用于选择数据类型；adc\_en配置测量状态；power\_mode配置电源模式；integrate\_time配置积分时间；adc\_res\_setting配置ADC转换精度，可调整测量范围。

基于上述信息，实例信息如下：

am\_local am\_const am\_bh1730fvc\_devinfo\_t \_\_g\_bh1730fvc\_devinfo = {

MEAS\_CONTINUE, /\*\*< \brief 连续测量 \*/

TYPE0\_ONLY, /\*\*< \brief 仅测量类型0 \*/

ADC\_EN, /\*\*< \brief 使能ADC转换 \*/

POWER\_ON, /\*\*< \brief 打开电源 \*/

0xDA, /\*\*< \ brief 配置积分时间 \*/

ADC\_MODE\_X64 /\*\*< \brief X64增益模式 \*/

};

其中，\_\_g\_bh1730fvc\_devinfo为用户自定义的实例信息，其地址作为p\_devinfo的实参传递。

1. **I2C句柄i2c\_handled**

使用ZLG116的I2C1与BH1730FVC通信，则I2C句柄可以通过ZLG116的I2C1实例初始化函数am\_zlg116\_i2c1\_inst\_init()获得句柄。即：

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

获得的I2C句柄即可直接作为i2c\_handled的实参传递。

基于实例、实例信息和I2C句柄，可以完成BH1730FVC的初始化。例如：

am\_bh1730fvc\_init (&\_\_g\_bh1730fvc\_dev, &\_\_g\_bh1730fvc\_devinfo, i2c\_handle);

为了便于配置实例信息，基于模块化编程思想，将初始化相关的实例、实例信息等的定义存放到相应的配置文件中，通过头文件引出实例初始化函数接口，源文件和头文件的程序范例分别详见程序清单5.69和程序清单5.70。

程序清单5.70 实例初始化范例程序（am\_hwconf\_bh1730fvc.c）

#include "ametal.h"

#include "am\_bh1730fvc.h"

#include "am\_zlg116\_inst\_init.h"

#include "zlg116\_pin.h"

// 设备信息

am\_local am\_const am\_bh1730fvc\_devinfo\_t \_\_g\_bh1730fvc\_devinfo = {

MEAS\_CONTINUE, /\*\*< \brief 连续测量 \*/

TYPE0\_ONLY, /\*\*< \brief 仅测量类型0 \*/

ADC\_EN, /\*\*< \brief 使能ADC转换 \*/

POWER\_ON, /\*\*< \brief 打开电源 \*/

0xDA, /\*\*< \brief 配置积分时间 \*/

ADC\_MODE\_X64 /\*\*< \brief X64增益模式 \*/

};

// 设备定义

am\_local am\_bh1730fvc\_dev\_t \_\_g\_bh1730fvc\_dev;

// 实例初始化

am\_bh1730fvc\_handle\_t am\_bh1730fvc\_inst\_init (void)

{

// 获取 I2C 实例句柄

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

return am\_bh1730fvc\_init (&\_\_g\_bh1730fvc\_dev,

&\_\_g\_bh1730fvc\_devinfo,

i2c\_handle);

}

程序清单5.71 实例初始化函数接口（am\_hwconf\_bh1730fvc.h）

#include "ametal.h"

#include "am\_bh1730fvc.h"

#ifdef \_\_cplusplus

extern "C" {

#endif

// 实例初始化

am\_bh1730fvc\_handle\_t am\_bh1730fvc\_inst\_init (void);

后续只需要使用无参数的实例初始化函数即可完成BH1730FVC实例的初始化，即执行如下语句：

am\_bh1730fvc\_inst\_init ();

### BH1730FVC接口函数

使用BH1730FVC接口函数即可获取当前光照强度信息，详见表5.20。下面具体介绍接口函数的使用方法。

表5.20 BH1730FVC接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_bh1730fvc\_read\_als\_adc (  am\_bh1730fvc\_handle\_t handle, uint16\_t \*p\_als\_adc) | 读取光照值 |

1. **读取光照值**

当实例初始化完成获取句柄后，通过调用光照度获取接口函数获取光照值并做处理，实例初始化如下所示：

am\_bh1730fvc\_handle\_t handle;

handle = am\_bh1730fvc\_inst\_init ();

当初始化完成，即可使用接口函数读取当前光照度信息，调用该函数，即可读取光照度，详见程序清单5.71。

程序清单5.72 使用BH1730FVC获取当前光照度

int am\_bh1730fvc\_read\_als\_adc (am\_bh1730fvc\_handle\_t handle,

uint16\_t \*p\_als\_adc)

{

// 用于存放读取的值

uint8\_t temp\_value[2] = {0};

int ret = AM\_OK;

// I2C设备指针

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

ret = am\_i2c\_read (p\_i2c\_dev, BH1730FVC\_DATA0LOW, temp\_value, 2);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

// 将读取出来的原始数据转换

\*p\_als\_adc = ((temp\_value[1] << 8) | temp\_value[0]) / 10;

return ret;

}

1. **使用范例**

当完成初始化后，即可使用通用的光照度读取接口获取当前光照度，读取并通过串口打印当前光照度信息的范例程序。详见程序清单5.72。

程序清单5.73 使用BH1730FVC范例程序

#include "ametal.h"

#include "am\_delay.h"

#include "am\_bh1730fvc.h"

#include "am\_hwconf\_bh1730fvc.h"

#include "am\_vdebug.h"

void demo\_bh1730fvc\_entry (void)

{

am\_bh1730fvc\_handle\_t handle;

uint16\_t als\_adc = 0;

handle = am\_bh1730fvc\_inst\_init();

while (1) {

am\_bh1730fvc\_read\_als\_adc (handle, &als\_adc);

AM\_DBG\_INFO("BH1730FVC\_ALS\_ADC = %d \r\n", als\_adc);

am\_mdelay(500);

}

}

## BMA253传感器组件

### 器件介绍

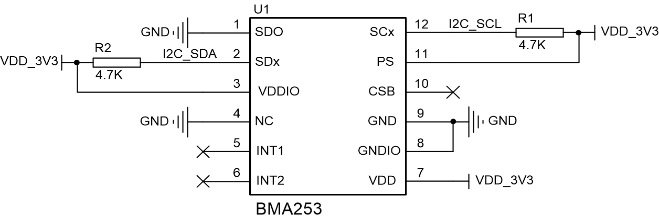


图5.12 BMA253电路原理图

BMA253是一款三轴加速度传感器，具有超小尺寸（2mm x 2mm）和扁平封装，可在1.2V至3.6V的宽电压范围工作，具有可编程测量范围、低通滤波带宽和电源模式，能满足特定应用中功能、性能和功耗的要求。广泛应用于手机、手持设备、计算机外围设备、人机界面、虚拟现实功能和游戏控制器等。

BMA253支持I2C和SPI数字接口，ZLG设计的电路采用I2C与ZLG116通信，具体电路如图5.12。

### BMA253初始化

AMetal提供了BMA253的驱动，在使用BMA253前，必须完成BMA253初始化操作，以获取到对应的操作句柄，进而使用BMA253的各种功能，其函数原型（am\_bma253.h）为：

am\_bma253\_handle\_t am\_bma253\_init (

am\_bma253\_dev\_t \*p\_dev,

const am\_bma253\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t i2c\_handle);

该函数意在获取BMA253传感器的初始化实例句柄，初始化完成后，即可使用通用接口读取应用数据，其中：

* p\_dev为指向am\_ bma253\_dev\_t类型实例的指针；
* p\_devinfo为指向am\_bma253\_devinfo\_t类型实例信息的指针；
* i2c\_handle为与BMA253通信的I2C实例句柄。

1. **实例**

定义am\_bma253\_dev\_t类型（am\_hwconf\_bma253.c）实例如下：

am\_local am\_bma253\_dev\_t \_\_g\_bma253\_dev;

其中\_\_g\_bma253\_dev为用户自定义的实例，其地址作为p\_dev的实参传递。

1. **实例信息**

实例信息主要描述了与BMA253相关的信息，即BMA253的配置信息，如加速度测量范围、数据滤波带宽、电源模式、低功耗睡眠周期，其类型am\_bma253\_devinfo的定义（am\_ bma253.h）如下：

typedef struct am\_bma253\_devinfo {

uint8\_t pmu\_range; /\*\*< \brief 加速度范围 \*/

uint8\_t pmu\_bw; /\*\*< \brief 数据滤波带宽 \*/

uint8\_t power\_mode; /\*\*< \brief 电源模式 \*/

uint8\_t sleep\_period; /\*\*< \brief 低功耗模式睡眠周期 \*/

} am\_bma253\_devinfo\_t;

其中pmu\_range为配置加速度测量范围，可配置为±2g、±4g、±8g、±16g四种测量范围；pmu\_bw为配置数据滤波带宽；power\_mode为配置电源模式，分别有工作模式、低功耗模式、挂起模式、深度挂起模式；sleep\_period配置睡眠周期。

基于上述信息，实例信息如下：

// 设备信息

am\_local am\_const am\_bma253\_devinfo\_t \_\_g\_bma253\_devinfo = {

PMU\_RANGE\_2G, /\*\*< \brief 加速度范围±2g \*/

PMU\_BW\_8Hz, /\*\*< \brief 数据滤波带宽7.81Hz \*/

NORMAL\_MODE, /\*\*< \brief 正常模式 \*/

SLEEP\_DUR\_HALF\_MS /\*\*< \brief 配置的睡眠周期为0.5ms \*/

};

其中，\_\_g\_bma253\_devinfo为用户自定义的实例信息，其地址作为p\_devinfo的实参传递。

1. **I2C句柄i2c\_handled**

使用ZLG116的I2C1与BMA253通信，则I2C句柄可以通过ZLG116的I2C1实例初始化函数am\_zlg116\_i2c1\_inst\_init()获得句柄。即：

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

获得的I2C句柄即可直接作为i2c\_handled的实参传递。

基于实例、实例信息和I2C句柄，可以完成BMA253的初始化。比如：

am\_bma253\_init (&\_\_g\_bma253\_dev, &\_\_g\_bma253\_devinfo, i2c\_handle);

为了便于配置实例信息，基于模块化编程思想，将初始化相关的实例、实例信息等的定义存放到相应的配置文件中，通过头文件引出实例初始化函数接口，源文件和头文件的程序范例分别详见程序清单5.73和程序清单5.74。

程序清单5.74 实例初始化范例程序（am\_hwconf\_bma253.c）

#include "ametal.h"

#include "am\_bma253.h"

#include "am\_zlg116\_inst\_init.h"

#include "zlg116\_pin.h"

// 设备信息

am\_local am\_const am\_bma253\_devinfo\_t \_\_g\_bma253\_devinfo = {

PMU\_RANGE\_2G, /\*\*< \brief 加速度范围±2g \*/

PMU\_BW\_8Hz, /\*\*< \brief 数据滤波带宽7.81Hz \*/

NORMAL\_MODE, /\*\*< \brief 正常模式 \*/

SLEEP\_DUR\_HALF\_MS /\*\*< \brief 配置的睡眠周期为0.5ms \*/

};

// 设备定义

am\_local am\_bma253\_dev\_t \_\_g\_bma253\_dev;

// 实例初始化

am\_bma253\_handle\_t am\_bma253\_inst\_init (void)

{

// 获取 I2C 实例句柄

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

return am\_bma253\_init (&\_\_g\_bma253\_dev,

&\_\_g\_bma253\_devinfo,

i2c\_handle);

}

程序清单5.75 实例初始化接口函数（am\_hwconf\_bma253.h）

#include "ametal.h"

#include "am\_bma253.h"

#ifdef \_\_cplusplus

extern "C" {

#endif

// 实例初始化

am\_bma253\_handle\_t am\_bma253\_inst\_init (void);

后续只需要使用无参数的实例初始化函数即可完成BMA253实例的初始化，即执行如下语句：

am\_bma253\_inst\_init ();

### BMA253接口函数

BMA253可以检测加速度和温度，使用接口函数即可获取加速度和温度数据信息，如表5.21所示。下面具体介绍各接口函数的使用方法。

表5.21 BMA253接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_bma253\_read\_accel(  am\_bma253\_handle\_t handle,  int16\_t \* p\_accel); | 读取三轴加速度 |
| int am\_bma253\_read\_temp (  am\_bma253\_handle\_t handle,  int8\_t \* p\_temp); | 读取温度值 |

1. **读取三轴加速度**

当实例初始化完成获取句柄后，通过调用加速度获取接口函数获取三轴加速度的值并做处理，实例初始化如下所示：

am\_bma253\_handle\_t handle;

handle = am\_bma253\_inst\_init();

当初始化完成，即可使用接口函数读取当前三轴加速度信息，调用该函数，即可读取三轴加速度信息，详见程序清单5.75。

程序清单5.76 使用BMA253获取当前三轴加速度

int am\_bma253\_read\_accel (am\_bma253\_handle\_t handle, int16\_t\* p\_accel)

{

int ret = AM\_OK;

// 读取X轴加速度

ret = \_\_bma253\_read\_accel\_reg(handle, BMA2x2\_X\_AXIS\_LSB\_ADDR, &p\_accel[0]);

// 检查读取是否失败

if (ret != AM\_OK) {

return ret;

}

// 读取Y轴加速度

ret = \_\_bma253\_read\_accel\_reg(handle, BMA2x2\_Y\_AXIS\_LSB\_ADDR, &p\_accel[1]);

// 检查读取是否失败

if (ret != AM\_OK) {

return ret;

}

// 读取Z轴加速度

ret = \_\_bma253\_read\_accel\_reg(handle, BMA2x2\_Z\_AXIS\_LSB\_ADDR, &p\_accel[2]);

// 检查读取是否失败

if (ret != AM\_OK) {

return ret;

}

return ret;

}

1. **读取温度值**

读取温度值，当实例初始化完成获取句柄后，获取温度值并做处理，实例初始化如下所示：

am\_bma253\_handle\_t handle;

handle = am\_bma253\_inst\_init();

当初始化完成，即可使用接口函数读取当前温度值信息，调用该函数，即可读取当前温度信息，详见程序清单5.76。

程序清单5.77 使用BMA253获取当前温度

int am\_bma253\_read\_temp (am\_bma253\_handle\_t handle, int8\_t\* p\_temp)

{

int ret = AM\_OK;

// I2C设备指针

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

ret = am\_i2c\_read(p\_i2c\_dev,BMA2x2\_TEMP\_ADDR, (uint8\_t\*)(p\_temp), 1);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

// 计算温度

\*p\_temp = (int8\_t) (24 + \*p\_temp \* 1.0 / 2);

return ret;

}

1. **使用范例**

当完成初始化后，即可使用通用的加速度读取接口和温度读取接口获取当前三轴加速度和温度值，读取并通过串口打印当前三轴加速度和温度信息的范例程序，详见程序清单5.77。

程序清单5.78 使用BMA253范例程序

#include "ametal.h"

#include "am\_delay.h"

#include "am\_bma253.h"

#include "am\_hwconf\_bma253.h"

#include "am\_vdebug.h"

void demo\_bma253\_entry (void)

{

int8\_t temp = 0;

int16\_t accel[3];

am\_bma253\_handle\_t handle;

handle = am\_bma253\_inst\_init();

while (1) {

// 读取温度

am\_bma253\_read\_temp(handle, &temp);

AM\_DBG\_INFO("temperature = %d ℃\r\n", temp);

// 读取三轴加速度

am\_bma253\_read\_accel (handle, accel);

AM\_DBG\_INFO("accel\_x = %d accel\_y = %d accel\_z = %d \r\n",

accel[0], accel[1], accel[2]);

am\_mdelay(500);

}

}

## LPS22HB传感器组件

LPS22HB是ST半导体推出的一款具有I2C和SPI接口的超紧凑压阻式绝对压力传感器，可输出24位分辨率压力数据、16位分辨率温度数据，功耗低至3uA，可用于便携设备的高度计、气压计和运动手表等。

### 器件介绍

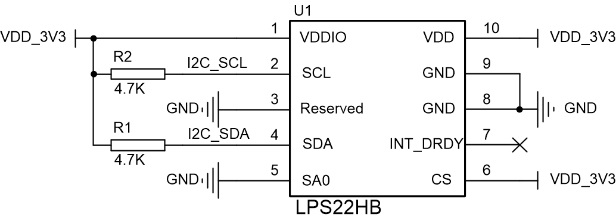


图5.13 LPS22HB电路原理图

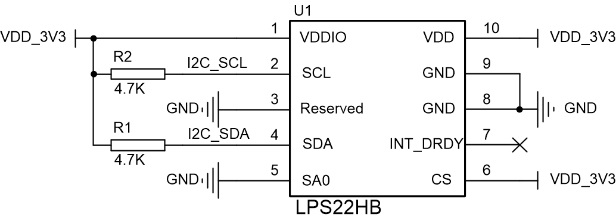
LPS22HB包括一个MEMS传感元件和一个IC接口。其传感元件包括用意法半导体开发的专有工艺制造的悬浮膜，用于检测260 ~ 1260hPa范围的绝对压力。IC接口通过I2C接口或SPI接口实现传感元件与应用间的通信。ZLG设计的LPS22HB电路如

图5.13所示，采用I2C通信，只需SDA和SCL两根信号线。SA0引脚用于地址选择，从机地址为101110x，当该引脚接高电平，则从机地址为0x5D，该引脚接地，则从机地址为0x5C。在此设计中，SA0引脚接地，该器件从机地址为0x5C，通过I2C与ZLG116核心板通信。

### LPS22HB初始化

AMetal提供了LPS22HB的驱动，在使用LPS22HB前，必须完成LPS22HB初始化操作，以获取到对应的操作句柄，进而使用LPS22HB的各种功能，其函数原型（am\_lps22hb.h）为：

am\_lps22hb\_handle\_t am\_lsp22hb\_init (

am\_lps22hb\_dev\_t \*p\_dev,

const am\_lps22hb\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t i2c\_handle);

该函数意在获取LPS22HB传感器的初始化实例句柄，初始化完成后，即可使用通用接口读取应用数据，其中：

* p\_dev为指向am\_lps22hb\_dev\_t类型实例的指针；
* p\_devinfo为指向am\_lps22hb\_devinfo\_t类型实例信息的指针；
* i2c\_handle为与LPS22HB通信的I2C实例句柄。

1. **实例**

定义am\_lps22hb\_dev\_t类型（am\_lps22hb.h）实例如下：

am\_local am\_lps22hb\_dev\_t \_\_g\_lps22hb\_dev;

其中\_\_g\_lps22hb\_dev为用户自定义的实例，其地址作为p\_dev的实参传递。

1. **实例信息**

实例信息主要描述了与LPS22HB相关的信息，即LPS22HB的配置信息，如数据更新速度、是否使能滤波等，其类型am\_lps22hb\_devinfo的定义（am\_lps22hb.h）如下：

typedef struct am\_lps22hb\_devinfo {

uint8\_t output\_data\_rate; /\*\*< \brief 配置数据更新速度 \*/

uint8\_t enable\_lpfp; /\*\*< \brief 是否使能低通滤波 \*/

uint8\_t lpfp\_bw; /\*\*< \brief 配置低通滤波带宽 \*/

} am\_lps22hb\_devinfo\_t;

其中output\_data\_rate为数据更新速度配置，可配置更新速度有1Hz、10 Hz、25 Hz、50 Hz、75 Hz；enable\_lpfp为低通滤波选项，配置是否使能低通滤波；lpfp\_bw为配置低通滤波带宽。

基于上述信息，实例信息如下：

// 设备信息

am\_local am\_const am\_lps22hb\_devinfo\_t \_\_g\_lps22hb\_devinfo = {

DATA\_UPDATE\_50Hz, /\*\*< \brief 数据更新速度为50Hz \*/

ENABLE\_LPFP, /\*\*< \brief 使能低通滤波 \*/

LPFP\_BW\_ODR20 /\*\*< \brief 滤波带宽为 ODR/20 \*/

};

其中，\_\_g\_lps22hb\_devinfo为用户自定义的实例信息，其地址作为am\_lps22hb\_devinfo\_t的实参传递。

1. **I2C句柄i2c\_handled**

使用ZLG116的I2C1与LPS22HB通信，则I2C句柄可以通过ZLG116的I2C1实例初始化函数am\_zlg116\_i2c1\_inst\_init()获得句柄。即：

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

获得的I2C句柄即可直接作为i2c\_handled的实参传递。

基于实例、实例信息和I2C句柄，可以完成LPS22HB的初始化。比如：

am\_lsp22hb\_init (&\_\_g\_lps22hb\_dev, &\_\_g\_lps22hb\_devinfo, i2c\_handle);

为了便于配置实例信息，基于模块化编程思想，将初始化相关的实例、实例信息等的定义存放到相应的配置文件中，通过头文件引出实例初始化函数接口，源文件和头文件的程序范例分别详见程序清单5.78和程序清单5.79。

程序清单5.79 实例初始化范例程序（am\_hwconf\_lps22hb.c）

#include "ametal.h"

#include "am\_lps22hb.h"

#include "am\_zlg116\_inst\_init.h"

#include "zlg116\_pin.h"

am\_local am\_const am\_lps22hb\_devinfo\_t \_\_g\_lps22hb\_devinfo = {

DATA\_UPDATE\_50Hz, /\*\*< \brief 数据更新速度为50Hz \*/

ENABLE\_LPFP, /\*\*< \brief 使能低通滤波 \*/

LPFP\_BW\_ODR20 /\*\*< \brief 滤波带宽为 ODR/20 \*/

};

// 设备定义

am\_local am\_lps22hb\_dev\_t \_\_g\_lps22hb\_dev;

// 实例初始化

am\_lps22hb\_handle\_t am\_lps22hb\_inst\_init (void)

{

// 获取 I2C 实例句柄

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

return am\_lsp22hb\_init (&\_\_g\_lps22hb\_dev,

&\_\_g\_lps22hb\_devinfo,

i2c\_handle);

}

程序清单5.80 实例初始化函数接口（am\_hwconf\_lps22hb.h）

#include "ametal.h"

#include "am\_lps22hb.h"

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*\* \brief 实例初始化 \*/

am\_lps22hb\_handle\_t am\_lps22hb\_inst\_init (void);

后续只需要使用无参数的实例初始化函数即可完成LPS22HB实例的初始化，即执行如下语句：

am\_lps22hb\_inst\_init ();

### LPS22HB接口函数

LPS22HB可以检测气压和温度，使用接口函数即可获取数据信息，如表5.22所示。下面具体介绍各接口函数的使用方法。

表5.22 LPS22HB接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_lsp22hb\_read\_temp (  am\_lps22hb\_handle\_t handle,  int16\_t \*p\_temp); | 读取温度值 |
| int am\_lsp22hb\_read\_press (  am\_lps22hb\_handle\_t handle,  uint32\_t \*p\_press); | 读取气压值 |

1. **读取温度值**

当实例初始化完成获取句柄后，通过调用温度获取接口函数获取温度值并做处理，实例初始化如下所示：

am\_lps22hb\_handle\_t handle;

handle = am\_lps22hb\_inst\_init();

当初始化完成，即可使用接口函数读取当前温度值信息，调用该函数，即可读取温度信息，详见程序清单5.80。

程序清单5.81 使用LPS22HB获取当前温度

int am\_lsp22hb\_read\_temp (am\_lps22hb\_handle\_t handle, int16\_t \*p\_temp)

{

uint8\_t status = 0;

uint8\_t temp\_low = 0;

uint8\_t temp\_high = 0;

int ret = AM\_OK;

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

// 读取状态寄存器

ret = am\_i2c\_read(p\_i2c\_dev, LPS22HB\_STATUS, &status, 1);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

if (status & STATUS\_T\_DA) {

ret = am\_i2c\_read(p\_i2c\_dev, LPS22HB\_TEMP\_OUT\_L, &temp\_low, 1);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

ret = am\_i2c\_read(p\_i2c\_dev, LPS22HB\_TEMP\_OUT\_H, &temp\_high, 1);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

}

\* p\_temp = (int16\_t)(((uint16\_t)(temp\_high) << 8)

| ((uint16\_t)(temp\_low)));

\* p\_temp = (double)(\*p\_temp) / 100.0;

return ret;

}

1. **读取气压值**

读取气压值，当实例初始化完成获取句柄后，获取气压值并做处理，实例初始化如下所示：

am\_lps22hb\_handle\_t handle;

handle = am\_lps22hb\_inst\_init();

当初始化完成，即可使用接口函数读取当前气压值信息，调用该函数，即可读取当前气压信息，详见程序清单5.81。

程序清单5.82 使用LPS22HB获取当前气压值

int am\_lsp22hb\_read\_press (am\_lps22hb\_handle\_t handle, uint32\_t \*p\_press)

{

uint8\_t status = 0;

uint8\_t press\_xl = 0;

uint8\_t press\_l = 0;

uint8\_t press\_h = 0;

int ret = AM\_OK;

// I2C设备指针

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

// 读取状态寄存器

ret = am\_i2c\_read(p\_i2c\_dev, LPS22HB\_STATUS, &status, 1);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

// 压力转换完成

if (status & STATUS\_P\_DA) {

am\_i2c\_read(p\_i2c\_dev, LPS22HB\_PRESS\_OUT\_XL, &press\_xl, 1);

am\_i2c\_read(p\_i2c\_dev, LPS22HB\_PRESS\_OUT\_L, &press\_l, 1);

am\_i2c\_read(p\_i2c\_dev, LPS22HB\_PRESS\_OUT\_H, &press\_h, 1);

}

// 转换成实际压力值

\*p\_press = (uint32\_t)(((uint32\_t)(press\_h) << 16)

| ((uint32\_t)(press\_l) << 8) | ((uint32\_t)(press\_xl)));

\*p\_press = (double)(\*p\_press) / 4096.0;

return ret;

}

1. **使用范例**

当完成初始化后，即可使用通用的温度读取接口和气压读取接口获取当前温度和气压值，读取并通过串口打印当前温度和气压信息的范例程序。详见程序清单5.82。

程序清单5.83 使用LPS22HB范例程序

#include "ametal.h"

#include "am\_delay.h"

#include "am\_lps22hb.h"

#include "am\_hwconf\_lps22hb.h"

#include "am\_vdebug.h"

void demo\_lps22hb\_entry (void)

{

am\_lps22hb\_handle\_t handle;

int16\_t temp = 0;

uint32\_t press = 0;

handle = am\_lps22hb\_inst\_init();

while (1) {

am\_lsp22hb\_read\_temp (handle, &temp);

am\_lsp22hb\_read\_press (handle, &press);

AM\_DBG\_INFO("temp = %d ℃ \r\n", temp);

AM\_DBG\_INFO("press = %d hPa \r\n", press);

am\_mdelay(500);

}

}

## LTR-553ALS传感器组件

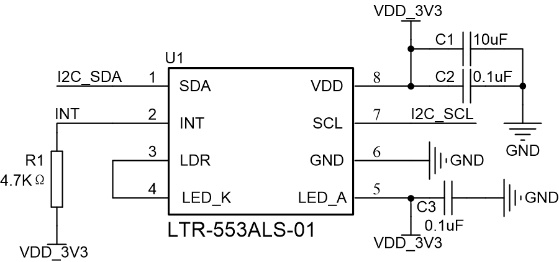


图5.14 LTR-553AL-01应用电路

### 器件介绍

LTR-553ALS-01内部集成低压I2C、光强传感器[ALS]和距离传感器[PS]；采用单个微型芯片无铅表面贴片封装。该传感器可将光强和距离转换为数值，并从I2C接口输出。它在0.01 lux至64k lux的宽动态范围内提供线性响应，非常适合高亮度环境下的应用；它内置的距离传感器（发射器和探测器），可在用户可配置的距离内检测物体。电路图详见图5.14。

### LTR-553ALS初始化

AMetal提供了LTR-553ALS的驱动，在使用LTR-553ALS前，必须完成LTR-553ALS初始化操作，以获取到对应的操作句柄，进而使用LTR-553ALS的各种功能，其函数（am\_ltr\_553als\_01.h）的原型为：

am\_ltr\_553als\_init (am\_ltr\_553als\_dev\_t \*p\_dev,

const am\_ltr\_553als\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t i2c\_handle);

该函数用于将LTR-553ALS-01按照用户配置的实例信息初始化，初始化完成后即可使用接口函数获取当前的光强和距离。p\_dev为指向am\_ltr\_553als\_dev\_t类型实例的指针，p\_devinfo为指向am\_ltr\_553als\_devinfo\_t类型实例信息的指针，i2c\_handle为与LTR-553ALS-01通信的I2C实例句柄。

1. **实例**

定义am\_ltr\_553als\_dev\_t类型（am\_ltr\_553als\_01.h）实例如下：

am\_local am\_ltr\_553als\_dev\_t \_\_g\_ltr\_553als\_dev; //定义LTR-553ALS实例

其中，\_\_g\_ltr\_553als\_dev为用户自定义的实例，其地址作为p\_dev的实参传递。

1. **实例信息**

实例信息主要描述了与LTR-553ALS传感器相关的信息，如光强测量范围，测量速度等信息。其类型am\_ltr\_553als\_devinfo\_t（am\_ltr\_553als\_01.h）的定义如下：

typedef struct am\_ltr\_553als\_devinfo {

uint8\_t als\_gain; /\*\*< \brief 光强测量范围 \*/

uint8\_t ps\_gain; /\*\*< \brief 距离测量范围 \*/

uint8\_t led\_pulse\_freq; /\*\*< \brief LED脉冲频率 \*/

uint8\_t led\_current\_duty; /\*\*< \brief LED电流占空比 \*/

uint8\_t led\_current; /\*\*< \brief LED电流 \*/

uint8\_t led\_pulse\_num; /\*\*< \brief LED脉冲数 \*/

uint8\_t ps\_meas\_rate; /\*\*< \brief 距离测量速度 \*/

uint8\_t als\_integre\_time; /\*\*< \brief 光照积分时间 \*/

uint8\_t als\_meas\_rate; /\*\*< \brief 光照测量速度 \*/

} am\_ltr\_553als\_devinfo\_t;

* als\_gain表示传感器光强测量的范围，如表5.23所示，有6种不同的量程可供选择；

表5.23 光强测量范围宏定义

|  |  |  |  |
| --- | --- | --- | --- |
| 宏名 | 代表测量范围 | 宏名 | 代表测量范围 |
| ALS\_GAIN\_1X | X1：1 lux ~ 64k lux | ALS\_GAIN\_8X | X8：0.125 lux ~ 8k lux |
| ALS\_GAIN\_2X | X2：0.5 lux ~ 32k lux | ALS\_GAIN\_48X | X48：0.02 lux ~ 1.3k lux |
| ALS\_GAIN\_4X | X4：0.25 lux ~ 16k lux | ALS\_GAIN\_96X | X96：0.01 lux ~ 600 lux |

表5.24 距离测量范围宏定义

|  |  |
| --- | --- |
| 宏名 | 代表测量范围 |
| PS\_GAIN\_16X | X16 |
| PS\_GAIN\_32X | X32 |
| PS\_GAIN\_64X | X64 |

* ps\_gain表示传感器距离测量的范围，如表5.24所示，总共有3种不同的增益倍数，每种增益倍数对应相应的测量范围；
* led\_pulse\_freq表示距离测量时LED的脉冲频率，如表5.25左侧所示，有30KHz到100KHz 8种频率可供选择；
* led\_current\_duty表示距离测量时LED的电流占空比，如表5.25右侧所示，可配置为25%到100%中的一种；

表5.25 LED脉冲频率和电流占空比宏定义

|  |  |  |  |
| --- | --- | --- | --- |
| 宏名 | 含义 | 宏名 | 含义 |
| LED\_PULSE\_FREQ\_30KHz | 30KHZ | LED\_CURRENT\_DUTY\_25 | 25% |
| LED\_PULSE\_FREQ\_40KHz | 40KHZ | LED\_CURRENT\_DUTY\_50 | 50% |
| …… | … | LED\_CURRENT\_DUTY\_75 | 75% |
| LED\_PULSE\_FREQ\_100KHz | 100KHz | LED\_CURRENT\_DUTY\_100 | 100% |

* led\_current表示距离测量时LED的电流，如表5.26左侧所示，有5种不同的电流可选；
* led\_pulse\_num表示距离测量时LED的脉冲数，如表5.26右侧所示，有15种脉冲数可选；

表5.26 LED电流及脉冲数宏定义

|  |  |  |  |
| --- | --- | --- | --- |
| 宏名 | 含义 | 宏名 | 含义 |
| LED\_CURRENT\_5MA | 5mA | LED\_PULSE\_NUM\_1 | 1个脉冲 |
| LED\_CURRENT\_10MA | 10mA | LED\_PULSE\_NUM\_2 | 2个脉冲 |
| LED\_CURRENT\_20MA | 20mA | LED\_PULSE\_NUM\_3 | 3个脉冲 |
| LED\_CURRENT\_50MA | 50mA | …… | … |
| LED\_CURRENT\_100MA | 100mA | LED\_PULSE\_NUM\_15 | 15个脉冲 |

* ps\_meas\_rate表示距离测量的速度，如表5.27左侧所示，从10ms到2000ms有8种速度可选；

表5.27 距离测量速度及光照积分时间宏定义

|  |  |  |  |
| --- | --- | --- | --- |
| 宏名 | 含义 | 宏名 | 含义 |
| PS\_MEAS\_RATE\_10MS | 10ms | ALS\_INTEGRATE\_50MS | 50ms |
| PS\_MEAS\_RATE\_50MS | 50ms | ALS\_INTEGRATE\_100MS | 100ms |
| PS\_MEAS\_RATE\_70MS | 70ms | ALS\_INTEGRATE\_150MS | 150ms |
| PS\_MEAS\_RATE\_100MS | 100ms | ALS\_INTEGRATE\_200MS | 200ms |
| PS\_MEAS\_RATE\_200MS | 200ms | ALS\_INTEGRATE\_250MS | 250ms |
| PS\_MEAS\_RATE\_500MS | 500ms | ALS\_INTEGRATE\_300MS | 300ms |
| PS\_MEAS\_RATE\_1000MS | 1000ms | ALS\_INTEGRATE\_350MS | 350ms |
| PS\_MEAS\_RATE\_2000MS | 2000ms | ALS\_INTEGRATE\_400MS | 400ms |

* als\_integre\_time表示光照积分时间，如表5.27右侧所示，从50ms到400ms一共有8种选择；
* als\_meas\_rate表示光照测量速度，如表5.28所示，从50ms到2000ms共有6种不同的测量速度可选；

表5.28 光强测量范围宏定义

|  |  |
| --- | --- |
| 宏名 | 代表测量范围 |
| ALS\_MEAS\_RATE\_50MS | 50ms |
| ALS\_MEAS\_RATE\_100MS | 100ms |
| ALS\_MEAS\_RATE\_200MS | 200ms |
| ALS\_MEAS\_RATE\_500MS | 500ms |
| ALS\_MEAS\_RATE\_1000MS | 1000ms |
| ALS\_MEAS\_RATE\_2000MS | 2000ms |

基于以上信息，实例信息可以定义如下：

am\_local am\_const am\_ltr\_553als\_devinfo\_t \_\_g\_ltr\_553als\_devinfo = {

ALS\_GAIN\_48X, /\*\*< \brief 0.02~1.3K lux \*/

PS\_GAIN\_32X, /\*\*< \brief X32 \*/

LED\_PULSE\_FREQ\_60KHz, /\*\*< \brief LED脉冲频率60KHz \*/

LED\_CURRENT\_DUTY\_100, /\*\*< \brief LED电流占空比为100% \*/

LED\_CURRENT\_100MA, /\*\*< \brief LED电流100mA \*/

LED\_PULSE\_NUM\_1, /\*\*< \brief LED发射1个脉冲 \*/

PS\_MEAS\_RATE\_100MS, /\*\*< \brief 距离测量速度为100ms \*/

ALS\_INTEGRATE\_100MS, /\*\*< \brief 光照积分时间为100ms \*/

ALS\_MEAS\_RATE\_500MS /\*\*< \brief 光强测量速度为500ms \*/

};

1. **I2C句柄i2c\_handle**

若使用ZLG116的I2C1与LTR-553ALS通信，则I2C句柄可以通过ZLG116的I2C1实例初始化函数am\_zlg116\_i2c1\_inst\_init()获得。即：

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

获得的I2C句柄可直接作为i2c\_handle的实参传递。

基于实例、实例信息和I2C句柄，可以完成LTR\_553ALS的初始化。如下：

am\_ltr\_553als\_init (&\_\_g\_ltr\_553als\_dev, &\_\_g\_ltr\_553als\_devinfo, i2c\_handle);

初始化函数的返回值即为LTR\_553ALS传感器的句柄，若返回值为NULL，说明初始化失败；若返回值不为NULL，说明返回了有效的handle，其可以作为LTR\_553ALS接口函数的参数。

### LTR\_553ALS接口函数

当初始化完成后，即可使用LTR\_553ALS的接口函数，详见表5.29。

表5.29 LTR\_553ALS的接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_ltr\_553als\_soft\_reset( am\_ltr\_553als\_handle\_t handle); | LTR\_553ALS软件复位 |
| int am\_ltr\_553als\_read\_als ( am\_ltr\_553als\_handle\_t handle,  uint16\_t \*p\_als); | 光照强度读取 |
| int am\_ltr\_553als\_read\_ps ( am\_ltr\_553als\_handle\_t handle,  uint16\_t \*p\_ps); | 距离读取 |

1. **LTR\_553ALS软件复位**

LTR\_553ALS软件复位，当实例初始化完成获取句柄后，即可使用此函数进行软件复位，实例初始化如下所示：

am\_ltr\_553als\_handle\_t handle;

handle = am\_ltr\_553als\_inst\_init();

初始化完成后，即可使用接口函数使传感器软件复位，详见程序清单5.83。

程序清单5.84 LTR\_553ALS软件复位

/\*\*

\* \brief LTR\_553ALS传感器软件复位

\* \return AM\_OK:成功 OTHER：失败

\*/

int am\_ltr\_553als\_soft\_reset (am\_ltr\_553als\_handle\_t handle)

{

int ret;

uint8\_t soft\_reset\_cmd = LTR\_553ALS\_SOFT\_RESET;

// I2C设备指针

am\_i2c\_device\_t \*p\_i2c\_dev = NULL;

// 验证参数的有效性

if (NULL == handle) {

return -AM\_EINVAL;

}

// 从handle中获取i2c设备指针

p\_i2c\_dev = &(handle->i2c\_dev);

// 写复位指令

ret = am\_i2c\_write(p\_i2c\_dev,

LTR\_553ALS\_CONTR,

&soft\_reset\_cmd,

1);

// 需要100ms初始化完成

am\_mdelay(100);

return ret;

}

1. **光照强度读取**

读取光照强度参数，当实例初始化完成获取句柄后，获取光照强度，实例初始化如下所示：

am\_ltr\_553als\_handle\_t handle;

handle = am\_ltr\_553als\_inst\_init();

当初始化完成，即可使用接口函数读取当前光照强度信息，调用该函数，即可读取当前光照强度信息，详见程序清单5.84。

程序清单5.85 使用LTR\_553ALS获取当前光照强度

/\*\*

\* \brief 读取LTR\_553ALS光照强度ADC

\* \return AM\_OK:成功 OTHER：失败

\*/

int am\_ltr\_553als\_read\_als (am\_ltr\_553als\_handle\_t handle, uint16\_t\* p\_als)

{

uint8\_t als\_temp[4];

int ret = AM\_OK;

// I2C设备指针

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

ret = am\_i2c\_read(p\_i2c\_dev, LTR\_553ALS\_DATA\_CH1\_0, als\_temp, 4);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

// 计算光强ADC

\*p\_als = (uint16\_t)((uint32\_t)(als\_temp[1] << 8 | als\_temp[0])

+ (uint32\_t)(als\_temp[3] << 8 | als\_temp[2])) / 2;

return ret;

}

1. **距离读取**

读取距离参数，当实例初始化完成获取句柄后，获取距离参数，实例初始化如下所示：

am\_ltr\_553als\_handle\_t handle;

handle = am\_ltr\_553als\_inst\_init();

当初始化完成，即可使用接口函数读取当前距离参数，调用该函数，即可读取当前距离参数，详见程序清单5.85。

程序清单5.86 使用LTR\_553ALS获取当前距离参数

/\*\*

\* \brief 读取LTR\_553ALS距离ADC

\* \return AM\_OK:成功 OTHER：失败

\*/

int am\_ltr\_553als\_read\_ps (am\_ltr\_553als\_handle\_t handle, uint16\_t\* p\_ps)

{

uint8\_t ps\_temp[2];

int ret = AM\_OK;

// I2C设备指针

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

ret = am\_i2c\_read(p\_i2c\_dev, LTR\_553ALS\_PS\_DATA\_0, ps\_temp, 2);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

// 数据合成

\*p\_ps = (uint16\_t)((ps\_temp[1] & 0x07) << 8 | ps\_temp[0]);

return ret;

}

1. **使用范例**

当完成初始化后，即可使用通用的光照轻度读取接口和距离读取接口获取当前光照强度及距离，读取并通过串口打印当前光照轻度及距离信息的范例程序详见程序清单5.86。

程序清单5.87 使用LTR-553ALS检测光强及距离范例程序

void demo\_ltr\_553als\_entry (void)

{

uint16\_t als = 0;

uint16\_t ps = 0;

am\_ltr\_553als\_handle\_t handle;

handle = am\_ltr\_553als\_inst\_init(); //LTR\_553ALS实例初始化

while (1) {

am\_ltr\_553als\_read\_als (handle, &als); //读取光照强度

am\_ltr\_553als\_read\_ps (handle, &ps); //读取距离值

AM\_DBG\_INFO("LTR\_553ALS = %d LTR\_553PS = %d\r\n", als, ps);

am\_mdelay(500);

}

}

## HTS221 传感器组件

### 器件介绍

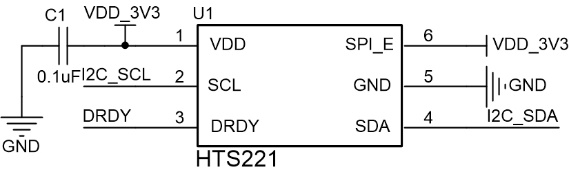


图5.15 HTS221应用电路

HTS221是一款测量相对湿度和温度的超紧凑型传感器。它包括一个传感元件和一个混合信号ASIC，通过数字串行接口传输测量信息。传感元件由能够检测相对湿度变化的电容器组成，并使用专用ST工艺制造。HTS221采用HLGA封装，可在-40℃至+ 120℃的温度范围内工作。连接方式详见图5.15。

### HTS221初始化

AMetal提供了HTS221的驱动，在使用HTS221前，必须完成HTS221初始化操作，以获取到对应的操作句柄，进而使用HTS221的各种功能，其函数（am\_hts221.h）的原型为：

am\_hts221\_handle\_t am\_hts221\_init (am\_hts221\_dev\_t \*p\_dev,

const am\_hts221\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t i2c\_handle)

该函数用于将HTS221按照用户配置的实例信息初始化，初始化完成后即可使用接口函数获取当前的光强和距离。p\_dev为指向am\_hts221\_dev\_t类型实例的指针，p\_devinfo为指向const am\_hts221\_devinfo\_t类型实例信息的指针，i2c\_handle为与HTS221通信的I2C实例句柄。

1. **实例**

定义am\_hts221\_dev\_t类型（am\_hts221.h）实例如下：

am\_local am\_hts221\_dev\_t \_\_g\_hts221\_dev; //定义HTS221实例

其中，\_\_g\_hts221\_dev为用户自定义的实例，其地址作为p\_dev的实参传递。

1. **实例信息**

实例信息主要描述了与HTS221传感器相关的信息，如温度样本数，湿度样本数等信息。其类型am\_hts221\_devinfo\_t（am\_hts221.h）的定义如下：

typedef struct am\_hts221\_devinfo {

uint8\_t temp\_sample; /\*\*< \brief 配置温度样本数 \*/

uint8\_t hum\_sample; /\*\*< \brief 配置湿度样本数 \*/

uint8\_t power\_mode; /\*\*< \brief 配置电源模式 \*/

uint8\_t data\_update\_config; /\*\*< \brief 配置数据更新方式 \*/

uint8\_t data\_update\_rate; /\*\*< \brief 配置数据更新速度 \*/

} am\_hts221\_devinfo\_t;

关于实例信息的配置选项都在枚举am\_hts221\_config\_t（am\_hts221.h）中，具体内容如程序清单5.87所示

程序清单5.88 hts221配置选项

typedef enum am\_hts221\_config {

/\*\*

\* \brief 可配置的温度采样样本数

\*/

TEMP\_SAMPLE\_4 = AM\_SBF(0,3), /\*\*< \brief 温度样本数为4 \*/

TEMP\_SAMPLE\_8 = AM\_SBF(1,3), /\*\*< \brief 温度样本数为8 \*/

TEMP\_SAMPLE\_16 = AM\_SBF(2,3), /\*\*< \brief 温度样本数为16 \*/

TEMP\_SAMPLE\_32 = AM\_SBF(3,3), /\*\*< \brief 温度样本数为32 \*/

TEMP\_SAMPLE\_64 = AM\_SBF(4,3), /\*\*< \brief 温度样本数为64 \*/

TEMP\_SAMPLE\_128 = AM\_SBF(5,3), /\*\*< \brief 温度样本数为128 \*/

TEMP\_SAMPLE\_256 = AM\_SBF(6,3), /\*\*< \brief 温度样本数为256 \*/

TEMP\_SAMPLE\_512 = AM\_SBF(7,3), /\*\*< \brief 温度样本数为512 \*/

/\*\*

\* \brief 可配置的湿度采样样本数

\*/

HUM\_SAMPLE\_4 = AM\_SBF(0,0), /\*\*< \brief 湿度样本数为4 \*/

HUM\_SAMPLE\_8 = AM\_SBF(1,0), /\*\*< \brief 湿度样本数为8 \*/

HUM\_SAMPLE\_16 = AM\_SBF(2,0), /\*\*< \brief 湿度样本数为16 \*/

HUM\_SAMPLE\_32 = AM\_SBF(3,0), /\*\*< \brief 湿度样本数为32 \*/

HUM\_SAMPLE\_64 = AM\_SBF(4,0), /\*\*< \brief 湿度样本数为64 \*/

HUM\_SAMPLE\_128 = AM\_SBF(5,0), /\*\*< \brief 湿度样本数为128 \*/

HUM\_SAMPLE\_256 = AM\_SBF(6,0), /\*\*< \brief 湿度样本数为256 \*/

HUM\_SAMPLE\_512 = AM\_SBF(7,0), /\*\*< \brief 湿度样本数为512 \*/

/\*\*

\* \brief 可配置的电源模式

\*/

POWER\_ON = AM\_SBF(1,7), /\*\*< \brief 打开电源 \*/

POWER\_DOWN = AM\_SBF(0,7), /\*\*< \brief 关闭电源 \*/

/\*\*

\* \brief 可配置的数据更新方式

\*/

UPDATE\_CONTINUE = AM\_SBF(0,2), /\*\*< \brief 数据更新一次 \*/

UPDATE\_ONE = AM\_SBF(1,2), /\*\*< \brief 数据连续更新 \*/

/\*\*

\* \brief 可配置的数据更新速度

\*/

DATA\_UPDATE\_1Hz = AM\_SBF(1,0), /\*\*< \brief 数据更新速度为1Hz \*/

DATA\_UPDATE\_7Hz = AM\_SBF(2,0), /\*\*< \brief 数据更新速度为7Hz \*/

DATA\_UPDATE\_12Hz = AM\_SBF(3,0), /\*\*< \brief 数据更新速度为12.5Hz \*/

} am\_hts221\_config\_t;

基于以上信息，实例信息可以定义如下：

am\_local am\_const am\_hts221\_devinfo\_t \_\_g\_hts221\_devinfo = {

TEMP\_SAMPLE\_128, /\*\*< \brief 温度样本数为128 \*/

HUM\_SAMPLE\_256, /\*\*< \brief 湿度样本数为256 \*/

POWER\_ON, /\*\*< \brief 打开电源 \*/

UPDATE\_CONTINUE, /\*\*< \brief 数据更新一次 \*/

DATA\_UPDATE\_12Hz /\*\*< \brief 数据更新速度为12.5Hz \*/

};

1. **I2C句柄i2c\_handle**

若使用ZLG116的I2C1与HTS221通信，则I2C句柄可以通过ZLG116的I2C1实例初始化函数am\_zlg116\_i2c1\_inst\_init()获得。即：

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

获得的I2C句柄可直接作为i2c\_handle的实参传递。

基于实例、实例信息和I2C句柄，可以完成HTS221的初始化。如下：

am\_hts221\_init (&\_\_g\_hts221\_dev, &\_\_g\_hts221\_devinfo, i2c\_handle);

初始化函数的返回值即为HTS221传感器的句柄，若返回值为NULL，说明初始化失败；若返回值不为NULL，说明返回了有效的handle，其可以作为HTS221接口函数的参数。

### HTS221接口函数

当初始化完成后，即可使用HTS221的接口函数，详见表5.30。

表5.30 HTS221的接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_hts221\_read\_temp (am\_hts221\_handle\_t handle,  int16\_t \*temp); | 温度读取 |
| int am\_hts221\_read\_hum (am\_hts221\_handle\_t handle,  uint8\_t \*hum); | 湿度读取 |

1. **温度读取**

读取温度信息参数，当实例初始化完成获取句柄后，获取温度参数并做处理，实例初始化如下所示：

am\_hts221\_handle\_t handle;

handle = am\_hts221\_inst\_init();

当初始化完成，即可使用接口函数读取当前温度信息，调用该函数，即可读取当前温度信息，详见程序清单5.88。

程序清单5.89 使用HTS221获取当前温度

/\*\*

\* \brief 读取温度

\*/

int am\_hts221\_read\_temp (am\_hts221\_handle\_t handle, int16\_t\* temp)

{

int16\_t temp\_adc = 0;

uint8\_t status = 0;

int ret = AM\_OK;

// I2C设备指针

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

// 读取数据转换状态

am\_i2c\_read(p\_i2c\_dev, HTS221\_STATUS\_REG, &status, 1);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

// 温度转换完成

if (status & HTS221\_T\_DA) {

\_\_hts221\_read\_adc(handle, HTS221\_TEMP\_OUT\_L, &temp\_adc);

\*temp = \_\_linear\_interpolation(&(handle->calib\_param\_temp), temp\_adc);

}

return ret;

}

1. **湿度读取**

读取湿度信息参数，当实例初始化完成获取句柄后，获取湿度参数并做处理，实例初始化如下所示：

am\_hts221\_handle\_t handle;

handle = am\_hts221\_inst\_init();

当初始化完成，即可使用接口函数读取当前湿度信息，调用该函数，即可读取当前湿度信息，详见程序清单5.89。

程序清单5.90 使用HTS221获取当前湿度

/\*\*

\* \brief 读取湿度

\*/

int am\_hts221\_read\_hum (am\_hts221\_handle\_t handle, uint8\_t \* hum)

{

int16\_t hum\_adc = 0;

uint8\_t status = 0;

int ret = AM\_OK;

// I2C设备指针

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

// 读取数据转换状态

am\_i2c\_read(p\_i2c\_dev, HTS221\_STATUS\_REG, &status, 1);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

// 湿度转换完成

if (status & HTS221\_H\_DA) {

\_\_hts221\_read\_adc(handle, HTS221\_HUMIDITY\_OUT\_L, &hum\_adc);

\*hum = \_\_linear\_interpolation(&(handle->calib\_param\_hum), hum\_adc);

// 转换湿度限幅

if (\*hum > 100) {

\*hum = 100;

}

}

return ret;

}

1. **使用范例**

当完成初始化后，即可使用通用的温度读取接口和湿度读取接口获取当前温度及湿度值，读取并通过串口打印当前温度及湿度信息的范例程序详见程序清单5.90。

程序清单5.91 使用HTS221检测温湿度范例程序

void demo\_hts221\_entry (void)

{

am\_hts221\_handle\_t handle;

int16\_t temp = 0;

uint8\_t hum = 0;

handle = am\_hts221\_inst\_init(); // HTS221实例初始化

while (1) {

am\_hts221\_read\_temp (handle, &temp); //读取当前相对湿度

am\_hts221\_read\_hum (handle, &hum); //读取当前温度

AM\_DBG\_INFO("temp = %d ℃ \r\n", temp);

AM\_DBG\_INFO("hum = %d rH \r\n", hum);

am\_mdelay(500);

}

}

## LIS3MDL传感器组件

### 器件介绍

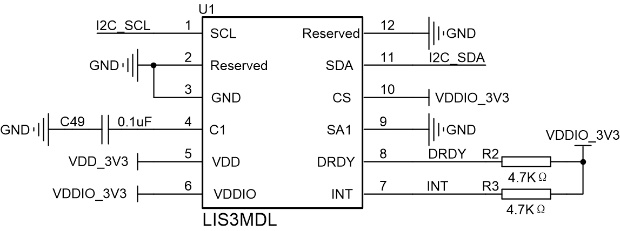


图5.16 LIS3MDL应用电路

LIS3MDL是一款超低功耗，高性能的三轴磁传感器。它量程为可选的±4 /±8/±12/±16高斯，有自检功能，并可以配置为响应磁场变化的中断。它包括一个I2C串行总线接口，支持标准和快速模式（100 kHz和400 kHz）和SPI串行标准接口。它采用LGA封装，能在-40℃至+ 85℃的温度范围内工作。应用电路如图5.16所示。

### LIS3MDL初始化

AMetal提供了LIS3MDL的驱动，在使用LIS3MDL前，必须完成LIS3MDL初始化操作，以获取到对应的操作句柄，进而使用LIS3MDL的各种功能，其函数（am\_lis3mdl.h）的原型为：

am\_lis3mdl\_handle\_t am\_lis3mdl\_init (am\_lis3mdl\_dev\_t \*p\_dev,

const am\_lis3mdl\_devinfo\_t \*p\_devinfo,

am\_i2c\_handle\_t i2c\_handle);

该函数用于将LIS3MDL按照用户配置的实例信息初始化，初始化完成后即可使用接口函数获取当前的光强和距离。p\_dev为指向am\_lis3mdl\_dev\_t类型实例的指针，p\_devinfo为指向const am\_lis3mdl\_devinfo\_t类型实例信息的指针，i2c\_handle为与LIS3MDL通信的I2C实例句柄。

1. **实例**

定义am\_lis3mdl\_dev\_t类型（am\_lis3mdl.h）实例如下：

am\_local am\_lis3mdl\_dev\_t \_\_g\_lis3mdl\_dev; //定义LIS3MDL实例

其中，\_\_g\_lis3mdl\_dev为用户自定义的实例，其地址作为p\_dev的实参传递。

1. **实例信息**

实例信息主要描述了与LIS3MDL传感器相关的信息，如光强测量范围，测量速度等信息。其类型am\_lis3mdl\_devinfo\_t（am\_lis3mdl.h）的定义如下：

typedef struct am\_lis3mdl\_devinfo {

uint8\_t temp\_en; /\*\*< \brief 温度测量使能位 \*/

uint8\_t operate\_mode\_xy; /\*\*< \brief X、Y轴操作模式选择 \*/

uint8\_t output\_data\_rate; /\*\*< \brief 数据输出速率 \*/

uint8\_t fast\_odr\_en; /\*\*< \brief 数据快速输出使能位 \*/

uint8\_t self\_test\_en; /\*\*< \brief 自检使能位 \*/

uint8\_t full\_scale; /\*\*< \brief 磁场测量范围 \*/

uint8\_t system\_mode; /\*\*< \brief 系统模式 \*/

uint8\_t operate\_mode\_z; /\*\*< \brief Z轴操作模式选择 \*/

} am\_lis3mdl\_devinfo\_t

关于实例信息的配置选项都在枚举am\_ lis3mdl\_config\_t（am\_lis3mdl.h）中，具体内容如程序清单5.91所示

程序清单5.92 lis3mdl配置选项

typedef enum am\_lis3mdl\_config {

/\*\*

\* \brief 温度传感器配置

\*/

TEMP\_ENABLE = AM\_SBF(1,7), /\*\*< \brief 使能温度测量 \*/

TEMP\_DISABLE = AM\_SBF(0,7), /\*\*< \brief 禁能温度测量 \*/

/\*\*

\* \brief X、Y轴操作模式选择

\*/

LOW\_POWER\_MODE\_XY = AM\_SBF(0,5), /\*\*< \brief 低功耗模式 \*/

MEDIUM\_MODE\_XY = AM\_SBF(1,5), /\*\*< \brief 中等性能模式 \*/

HIGH\_MODE\_XY = AM\_SBF(2,5), /\*\*< \brief 高性能模式 \*/

ULTRA\_HIGH\_MODE\_XY = AM\_SBF(3,5), /\*\*< \brief 超高性能模式 \*/

/\*\*

\* \brief 数据输出速率

\*/

ODR\_1Hz = AM\_SBF(0,2), /\*\*< \brief 数据输出频率0.625Hz \*/

ODR\_2Hz = AM\_SBF(1,2), /\*\*< \brief 数据输出频率1.25Hz \*/

ODR\_3Hz = AM\_SBF(2,2), /\*\*< \brief 数据输出频率2.5Hz \*/

ODR\_5Hz = AM\_SBF(3,2), /\*\*< \brief 数据输出频率5Hz \*/

ODR\_10Hz = AM\_SBF(4,2), /\*\*< \brief 数据输出频率10Hz \*/

ODR\_20Hz = AM\_SBF(5,2), /\*\*< \brief 数据输出频率20Hz \*/

ODR\_40Hz = AM\_SBF(6,2), /\*\*< \brief 数据输出频率40Hz \*/

ODR\_80Hz = AM\_SBF(7,2), /\*\*< \brief 数据输出频率80Hz \*/

/\*\*

\* \brief 数据快速输出使能控制

\*/

FAST\_ODR\_ENABLE = AM\_SBF(1,1), /\*\*< \brief 使能数据快速输出 \*/

FAST\_ODR\_DISABLE = AM\_SBF(0,1), /\*\*< \brief 不使能数据快速输出 \*/

/\*\*

\* \brief 自检使能控制

\*/

SELF\_TEST\_ENABLE = AM\_SBF(1,0), /\*\*< \brief 使能自检 \*/

SELF\_TEST\_DISABLE = AM\_SBF(0,0), /\*\*< \brief 不使能自检 \*/

/\*\*

\* \brief 可配置的磁场测量范围

\*/

FULL\_4GAUSS = AM\_SBF(0,5), /\*\*< \brief 磁场测量范围±4 gauss \*/

FULL\_8GAUSS = AM\_SBF(1,5), /\*\*< \brief 磁场测量范围±8 gauss \*/

FULL\_12GAUSS = AM\_SBF(2,5), /\*\*< \brief 磁场测量范围±12 gauss \*/

FULL\_16GAUSS = AM\_SBF(3,5), /\*\*< \brief 磁场测量范围±16 gauss \*/

/\*\*

\* \brief 可配置的系统模式

\*/

CONTINUE\_MODE = AM\_SBF(0,0), /\*\*< \brief 连续转换模式 \*/

SINGLE\_MODE = AM\_SBF(1,0), /\*\*< \brief 单次转换模式 \*/

POWER\_DOWN = AM\_SBF(2,0), /\*\*< \brief 掉电模式 \*/

/\*\*

\* \brief Z轴操作模式选择

\*/

LOW\_POWER\_MODE\_Z = AM\_SBF(0,2), /\*\*< \brief 低功耗模式 \*/

MEDIUM\_MODE\_Z = AM\_SBF(1,2), /\*\*< \brief 中等性能模式 \*/

HIGH\_MODE\_Z = AM\_SBF(2,2), /\*\*< \brief 高性能模式 \*/

ULTRA\_HIGH\_MODE\_Z = AM\_SBF(3,2), /\*\*< \brief 超高性能模式 \*/

} am\_lis3mdl\_config\_t;

基于以上信息，实例信息可以定义如下：

am\_local am\_const am\_lis3mdl\_devinfo\_t \_\_g\_lis3mdl\_devinfo = {

TEMP\_ENABLE, /\*\*< \brief 使能温度测量 \*/

HIGH\_MODE\_XY, /\*\*< \brief 高性能模式 \*/

ODR\_10Hz, /\*\*< \brief 数据输出频率10Hz \*/

FAST\_ODR\_DISABLE, /\*\*< \brief 不使能数据快速输出 \*/

SELF\_TEST\_ENABLE, /\*\*< \brief 使能自检 \*/

FULL\_4GAUSS, /\*\*< \brief 磁场测量范围±4 gauss \*/

CONTINUE\_MODE, /\*\*< \brief 连续转换模式 \*/

HIGH\_MODE\_Z /\*\*< \brief 高性能模式 \*/

};

1. **I2C句柄i2c\_handle**

若使用ZLG116的I2C1与LIS3MDL通信，则I2C句柄可以通过ZLG116的I2C1实例初始化函数am\_zlg116\_i2c1\_inst\_init()获得。即：

am\_i2c\_handle\_t i2c\_handle = am\_zlg116\_i2c1\_inst\_init();

获得的I2C句柄可直接作为i2c\_handle的实参传递。

基于实例、实例信息和I2C句柄，可以完成LIS3MDL的初始化。如下：

am\_lis3mdl\_init (&\_\_g\_lis3mdl\_dev, &\_\_g\_lis3mdl\_devinfo, i2c\_handle);

初始化函数的返回值即为LIS3MDL传感器的句柄，若返回值为NULL，说明初始化失败；若返回值不为NULL，说明返回了有效的handle，其可以作为LIS3MDL接口函数的参数。

### LIS3MDL接口函数

当初始化完成后，即可使用LIS3MDL的接口函数，详见表5.31。

表5.31 LIS3MSL的接口函数

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| int am\_lis3mdl\_read (am\_lis3mdl\_handle\_t handle,  uint16\_t \*p\_data); | 磁场强度读取 |
| int am\_lis3mdl\_read\_temp (am\_lis3mdl\_handle\_t handle,  int8\_t \*p\_temp); | 温度读取 |

1. **磁场强度读取**

读取磁场强度参数，当实例初始化完成获取句柄后，获取磁场强度并做处理，实例初始化如下所示：

am\_lis3mdl\_handle\_t handle;

handle = am\_lis3mdl\_inst\_init();

当初始化完成，即可使用接口函数读取当前磁场强度信息，调用该函数，即可读取磁场强度信息，详见程序清单5.92。

程序清单5.93 使用LIS3MDL获取当前磁场强度

/\*\*

\* \brief LIS3MDL传感器读取三轴磁数值

\*/

int am\_lis3mdl\_read (am\_lis3mdl\_handle\_t handle, uint16\_t \*p\_data)

{

// 用于存放读取的值

uint8\_t temp\_value[6] = {0};

int ret = AM\_OK;

// I2C设备指针

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

// 读取原始数据

ret = am\_i2c\_read (p\_i2c\_dev, LIS3MDL\_OUT\_X\_L, temp\_value, 6);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

// 将数据处理并输出

p\_data[0] = (temp\_value[1] << 8) | temp\_value[0];

p\_data[1] = (temp\_value[3] << 8) | temp\_value[2];

p\_data[2] = (temp\_value[5] << 8) | temp\_value[4];

return ret;

}

1. **温度信息读取**

读取温度信息参数，当实例初始化完成获取句柄后，获取温度参数并做处理，实例初始化如下所示：

am\_lis3mdl\_handle\_t handle;

handle = am\_lis3mdl\_inst\_init();

当初始化完成，即可使用接口函数读取当前温度信息，调用该函数，即可读取当前温度信息，详见程序清单5.93。

程序清单5.94 LIS3MDL获取当前温度

/\*\*

\* \brief LIS3MDL传感器读取温度

\*/

int am\_lis3mdl\_read\_temp (am\_lis3mdl\_handle\_t handle, int8\_t \*p\_temp)

{

// 用于存放读取的值

uint8\_t temp\_value[2] = {0};

int16\_t temp = 0;

int ret = AM\_OK;

// I2C设备指针

am\_i2c\_device\_t \*p\_i2c\_dev = &(handle->i2c\_dev);

// 检查是否使能温度测量

if (handle->p\_devinfo->temp\_en != TEMP\_ENABLE) {

return ret;

}

// 读取原始数据

ret = am\_i2c\_read (p\_i2c\_dev, LIS3MDL\_TEMP\_OUT\_L, temp\_value, 2);

// 检查发送是否失败

if (ret != AM\_OK) {

return ret;

}

// 将数据处理并输出

temp = (int16\_t)(temp\_value[1] << 8 | temp\_value[0]);

\*p\_temp = temp / 8 + 25;

return ret;

}

1. **使用范例**

当完成初始化后，即可使用通用的磁场强度读取接口和温度读取接口获取当前磁场强度及温度值，读取并通过串口打印当前磁场强度及温度信息的范例程序详见程序清单5.94。

程序清单5.95 使用LIS3MDL检测磁场与温度范例程序

void demo\_lis3mdl\_entry (void)

{

am\_lis3mdl\_handle\_t handle;

uint16\_t data[3] = {0};

int8\_t temp = 0;

handle = am\_lis3mdl\_inst\_init(); //LIS3MDL实例初始化

while (1) {

am\_lis3mdl\_read (handle, data); //读取传感器测得的磁场强度

am\_lis3mdl\_read\_temp (handle, &temp); //读取传感器测得的温度

AM\_DBG\_INFO("LIS3MDL = %d %d %d \r\n", data[0], data[1], data[2]);

AM\_DBG\_INFO("LIS3MDL\_temp = %d ℃ \r\n", temp);

am\_mdelay(500);

}

}

## 蜂鸣器组件

### 器件介绍

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于各种电子产品中作发声器件。蜂鸣器按内部结构主要分为压电式蜂鸣器和电磁式蜂鸣器两种类型，按驱动方式主要分为有源蜂鸣器和无源蜂鸣器两种类型。蜂鸣器驱动电路详见图5.17。

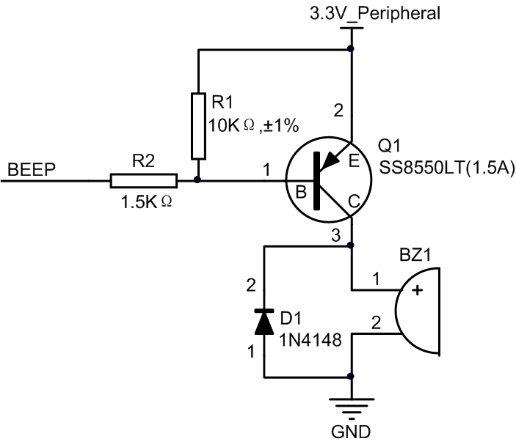


图5.17 蜂鸣器驱动电路

### 蜂鸣器接口函数

为了实现跨平台应用软件，AMetal提供了操作蜂鸣器的通用接口，详见表5.32。

表5.32 蜂鸣器通用接口（am\_buzzer.h）

|  |  |
| --- | --- |
| 函数原型 | 功能简介 |
| void am\_buzzer\_on(void); | 打开蜂鸣器 |
| void am\_buzzer\_off(void); | 关闭蜂鸣器 |
| void am\_buzzer\_beep(uint32\_t ms); | 蜂鸣器鸣叫指定时间（同步） |
| void am\_buzzer\_beep\_async(uint32\_t ms); | 蜂鸣器鸣叫指定时间（异步） |

1. **打开蜂鸣器**

打开蜂鸣器的函数原型为：

void am\_buzzer\_on(void);

打开蜂鸣器，使蜂鸣器开始鸣叫的范例程序详见程序清单5.95。

程序清单5.95 am\_buzzer\_on()范例程序

1 am\_buzzer\_on();

1. **关闭蜂鸣器**

关闭蜂鸣器的函数原型为：

void am\_buzzer\_off(void);

关闭蜂鸣器，使蜂鸣器停止鸣叫的范例程序详见程序清单5.96。

程序清单5.96 am\_buzzer\_off()范例程序

1 am\_buzzer\_off();

1. **蜂鸣器鸣叫指定时间（同步）**

该函数用于打开蜂鸣器，使蜂鸣器鸣叫指定时间后自动关闭，该函数会一直等到蜂鸣器鸣叫结束后返回。其函数原型为：

void am\_buzzer\_beep(uint32\_t ms);

使蜂鸣器鸣叫50毫秒（“嘀”一声）的范例程序详见程序清单5.97。

程序清单5.97 am\_buzzer\_beep()范例程序

1 am\_buzzer\_beep(50);

注意，由于该函数会一直等到蜂鸣器鸣叫结束后才返回，因此主程序调用该函数后会阻塞50ms。

1. **蜂鸣器鸣叫指定时间（异步）**

该函数用于打开蜂鸣器，使蜂鸣器鸣叫指定时间后自动关闭，与am\_buzzer\_beep()函数不同的是，该函数会立即返回，不会等待蜂鸣器鸣叫结束。其函数原型为：

void am\_buzzer\_beep\_async(uint32\_t ms);

使蜂鸣器鸣叫50毫秒（“嘀”一声）的范例程序详见程序清单5.98。

程序清单5.98 am\_buzzer\_beep\_async()范例程序

1 am\_buzzer\_beep\_async(50);

注意，由于该函数不会等待蜂鸣器鸣叫结束，因此，主程序调用该函数后，会立即返回，不会被阻塞。显然，要使应用程序可以使用通用接口操作蜂鸣器，就需要为具体的蜂鸣器设备提供相应的驱动。

### 无源蜂鸣器驱动

无源蜂鸣器内部没有振荡源，需要外部使用一定频率的方波信号驱动才能发声。AMetal已经提供了无源蜂鸣器的驱动，直接输出PWM驱动无源蜂鸣器发声。无源蜂鸣器驱动初始化函数原型详见程序清单5.99。

程序清单5.99 无源蜂鸣器驱动初始化函数原型

1 int am\_buzzer\_pwm\_init(

2 am\_pwm\_handle\_t pwm\_handle,

3 int chan,

4 unsigned int duty\_ns,

5 unsigned int period\_ns);

其中，pwm\_handle为标准的pwm服务句柄，chan为PWM通道号，duty\_ns和period\_ns分别指定了输出PWM波形的脉宽和周期，决定了蜂鸣器鸣叫的响度和频率。AM116-Core板载了一个无源蜂鸣器。只要短接J7，则蜂鸣器接入PIOB\_8。

以ZLG116N32A外设tim16提供PWM输出功能为例，AMetal提供了对应的实例初始化函数，其原型为：

am\_pwm\_handle\_t am\_zlg116\_tim16\_pwm\_inst\_init(void);

若需使用tim16输出PWM，只需调用其对应的实例初始化函数即可获取标准的PWM服务句柄：

am\_pwm\_handle\_t pwm\_handle = am\_zlg116\_tim16\_pwm\_inst\_init();

获取的PWM服务句柄即可作为am\_buzzer\_pwm\_init()函数pwm\_handle的实参传递。

由于无源蜂鸣器使用的引脚为PIOB\_8，对应的通道为1，因此，chan参数的值应设置为1。duty\_ns为和period\_ns分别指定了输出PWM波形的脉宽和周期，若频率设置为2500Hz，则对应的周期时间为：400000ns(1000000000/2500)，占空比通常为50%（脉宽时间为周期时间的一半），即脉宽时间为：200000ns。实际中，可以根据实际发声效果修改脉宽时间和周期时间。

基于对各个参数的分析，即可调用am\_buzzer\_pwm\_init()完成无源蜂鸣器的初始化：

am\_buzzer\_pwm\_init(am\_zlg116\_tim16\_pwm\_inst\_init(),1,200000,400000);

无源蜂鸣器作为一种板载资源，在系统启动时已经默认进行了初始化操作，因为应用程序无需再手动调用无源蜂鸣器初始化函数，就可以直接使用通用接口操作蜂鸣器。

若用户不需要使用蜂鸣器，为了节省内存空间，可以将工程配置文件（am\_prj\_config.h）中的AM\_CFG\_BUZZER\_ENABLE宏值修改为0，以裁剪掉蜂鸣器，该宏本质上控制了板级初始化函数中的一段程序，详见程序清单5.100。

程序清单5.100 在板级初始化中裁剪蜂鸣器的原理

1 void am\_board\_init(void)

2 {

3 //......

4 #if(AM\_CFG\_BUZZER\_ENABLE == 1)

5 am\_buzzer\_pwm\_inst\_init();

6 #endif

7 //......

8 }

注：板级初始化函数在系统启动时自动调用，初始化完毕后才会进入应用程序入口，即am\_main()。

## M150-II打印机组件

### 器件介绍

### M150-II打印机初始化

### M150-II打印机接口函数

## HC595组件

### 器件介绍

### HC595初始化

### HC595接口函数