目录

第1章 自动点胶机应用方案

1.1 产品概述

自动点胶机/焊接/螺丝控制器，主要是给外部的电机机构或者电机平台提供控制信号，有序的控制多个电机，从而实现自动点胶、自动焊接或者自动拧螺丝的功能。

产品实物如图 1.1 所示，分为主控板卡和手持示教器两部分。驱动控制算法由主控板卡完成，主控板卡通过 UART 接口与手持示教器进行通信，实时获取手持示教器的运行状态，从而做出相应的控制功能。手持示教器中的主控采用 ZLG116N32A 设计。

图 1.1 产品展示

1.2 方案介绍

1.2.1 功能框图

自动点胶机手持示教器功能框图如图 1.2 所示，自动点胶机手持示教器通过 5V 输入电压供电，经过 LDO 转化为 3.3V 给 MCU 供电，外置 1 个 LED 状态指示灯和 1 个蜂鸣器，支持 48 个按键（14 路 I/O 提供 6×8 矩阵式按键检测），外扩 4MByte SPI NOR Flash 用于存储控制相关数据。手持示教器通过 UART 转化为 RS232 电平串口与主控板卡通讯，以保证更远的通信距离。

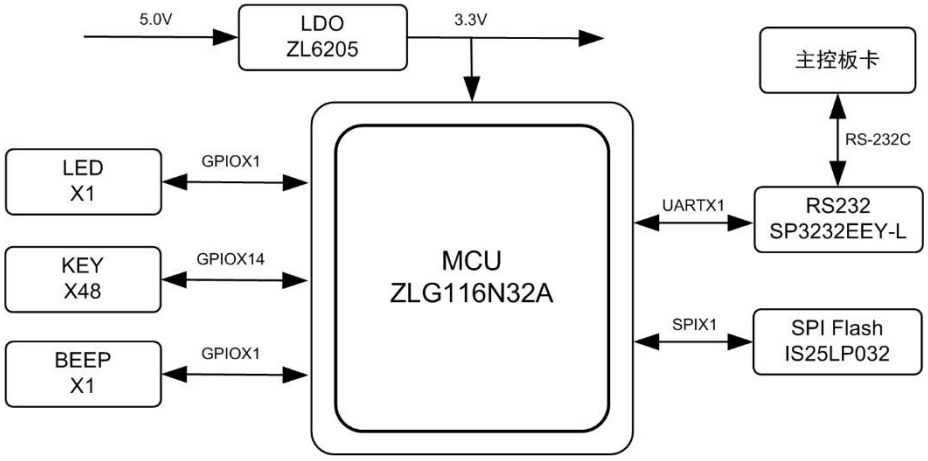

图 1.2 功能框图

1.2.2 资源需求

- 1 路 I/O 实现 LED 状态显示；
- 1 路 I/O 实现 BEEP 发音提示；
- 14 路 I/O 提供 6×8 矩阵式按键检测实现 48 个按键输入；
- 1 路 SPI 接口用于外扩 SPI NOR Flash；
- 1 路 UART 接口转化为 RS232 电平串口用于与主控板卡通信。

1.2.3 优势特点

对比自动点胶机手持示教器同类产品现用芯片方案，有如下优势：

- MCU 资源丰富（64kB Flash / 8K SRAM）、开发更为灵活，超高性价比；
- 完善齐全的 DEMO 软件和详尽的技术文档，帮助客户快速完成产品开发；
- 提供矩阵键盘、SPI NOR Flash 存储器驱动、带缓冲区的 UART 接口通讯等丰富的组件软件，极大简化客户的开发，更专注于核心应用软件的设计；
- 采用 AMetal 软件架构，真正实现跨平台移植，帮助客户快速完成产品升级换代。

1.2.4 推荐器件

ZLG 提供点胶机手持示教器全套 BOM 解决方案，全套 BOM 打包，一站式采购，降低整体成本。主控芯片采用 ZLG IoT MCU ZLG116N32A 设计，详细器件推荐如表 1.1 所示。

表 1.1 推荐器件

产品名称	型号	厂家	器件特点
MCU	ZLG116N32A	ZLG	Cortex-M0 内核，64kB Flash / 8K SRAM； 运行频率高达 48MHz； 支持宽电压输入 2.0~5.5V； 多路 UART、SPI、I2C 等外设接口。
Flash	IS25LP032	ISSI	数据存储、低功耗、低电压。
串口通信	SP3232EEY-L	EXAR	支持波特率高达 200K； 支持 3.3~5.0V 电压范围； 两线制异步串行通信。
LDO	ZL6205	ZLG	成本低、噪音小、静态电流小。

1.3 参考设计

1.3.1 矩阵键盘

本方案采用 6×8 矩阵按键设计，仅用 14 个 I/O 就能实现 48 个按键检测，如图 1.3 所示，行线为 6，列线为 8，行线为输出，列线为输入。

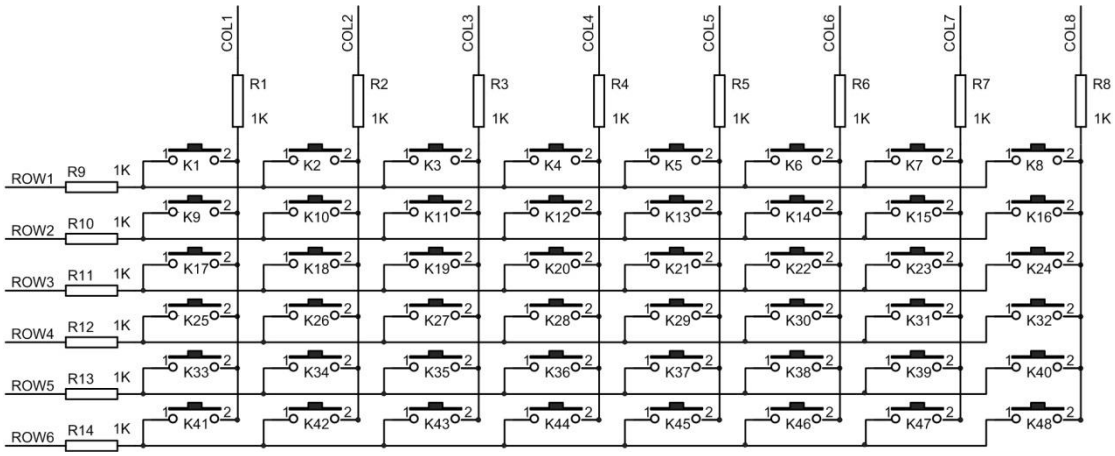

图 1.3 矩阵键盘电路

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。

AMetal 已经集成了矩阵按键组件软件，客户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 1.1。

程序清单 1.1 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 1.2 所示；

程序清单 1.2　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  }
```

更多矩阵按键组件的使用请参考《面向 AMetal 框架与接口的编程》第 4.3 章节。

1.3.2 SPI NOR Flash 存储器

本方案需要外扩一个 4Mbyte SPI NOR Flash 用于存储控制相关数据，采用 ISSI 的 IS25LP032 设计。

IS25LP032 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 1.4。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

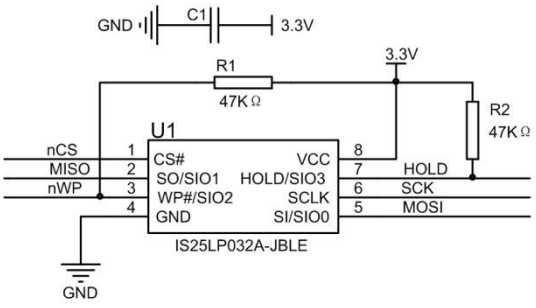

图 1.4　SPI NOR Flash 电路原理图

不会使用，可通过上拉电阻上拉至高电平。

AMetal 提供了支持常见的 IS25LP064、 MX25L8006、MX25L1606……等系列 SPI NOR Flash 器件的驱动函数，SPI NOR 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，因此除初始化、读写函数外，还有一个擦除函数，其接口函数详见表 1.2。

表 1.2 IS25xx 接口函数

函数原型	功能简介
am_is25xx_handle_t is25xx_handle = **am_is25xx_inst_init();**	实例初始化
int **am_is25xx_erase**(am_is25xx_handle_t handle, uint32_t addr, uint32_t len);	擦除
int **am_is25xx_write**(am_is25xx_handle_t handle, uint32_t addr, uint8_t *p_buf, uint32_t len);	写入数据
int **am_is25xx_read**(am_is25xx_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如程序清单 1.3 所示。

程序清单 1.3 擦除函数

```
int am_is25xx_erase(am_is25xx_handle_thandle,uint32_taddr, uint32_t  len);
```

其中，handle 为 IS25LP032 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 1.4。

程序清单 1.4 擦除范例程序

```
1    am_is25xx_erase(is25xx_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。

2. 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如程序清单 1.5 所示。

程序清单 1.5 写入函数原型

```
int am_is25xx_write(
    am_is25xx_handle_t      handle,                 // is25xx 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 1.6。

程序清单 1.6 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_is25xx_erase(is25xx_handle, 0x001000, 4096);              // 擦除一个扇区
5    am_is25xx_write(is25xx_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 1.7。

程序清单 1.7 写入数据范例程序

```
1    am_is25xx_write(is25xx_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。

3. 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如程序清单 1.8 所示。

程序清单 1.8 读取函数

```
int am_is25xx_read(
    am_is25xx_handle_t  handle,                         // is25xx 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 1.9。

程序清单 1.9 读取数据范例程序

```
1   uint8_tdata[128];
2   am_is25xx_read(is25xx_handle, 0x001000, buf, 128);          // 读取 128 字节数据
```

由于函数中的参数为 IS25LP032 的实例 handle，与 IS25LP032 器件具有依赖关系，因此无法实现跨平台调用。对此，AMetal 将其进行抽象为一个读写 IS25LP032 的 MTD（Memory Technology Device），使之与器件无关，实现跨平台调用。

另外，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 IS25LP032 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，IS25LP032 的使用寿命大打折扣。为了延长 flash 的使用寿命，AMetal 提供了 FTL（Flash Translation Layer）通用接口供用户使用，在实际写入时，将数据写入到擦除次数最少的区域。

关于 MTD 和 FTL 详细说明和使用请参考《面向 AMetal 框架与接口的编程》第 5.2 章节。

1.3.3 带缓冲区的 UART 接口

手持示教器通过 UART 转化为 RS232 电平串口与主控板通信。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 1.3，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 1.3 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,	发送数据

const uint8_t *p_txbuf, uint32_t nbytes);	
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

1. 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 1.10 所示。

程序清单 1.10 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 1.11。

程序清单 1.11 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。

2. 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。其函数原型如程序清单 1.12 所示。

程序清单 1.12 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 1.13。

程序清单 1.13 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。

3. 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，其函数原型如程序清单 1.14 所示。

程序清单 1.14 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 1.15。

程序清单 1.15 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
```

4. 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。其函数原型如程序清单 1.16 所示。

程序清单 1.16 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。

更多串口带缓冲组件的使用请参考《面向 AMetal 框架与接口的编程》第 4.8 章节。

第2章 出租车打印机应用方案

2.1 产品概述

出租车打印机如图 2.1 所示，出租车打印机属于梭式点阵打印机，是利用机械和电路驱动打印针撞击色带和打印介质，进而打印出点阵，通过打印的字符或组成的图形来完成打印，出租车打印机具有结构简单、技术成熟、性价比高、消耗费用低等特点。

主机通过 UART 和 MCU 通信，MCU 控制打印机完成信息打印。本出租车打印机方案的主控采用 ZLG217P64A 设计。

图 2.1 产品展示

2.2 方案介绍

2.2.1 功能框图

出租车打印机功能框图如图 2.2 所示，打印机机头使用 5V 供电，通过 LDO 将 5V 电压转化为 3.3V 给系统供电，出租车打印机外置 2 个 LED 状态指示灯，1 路 I/O 驱动蜂鸣器作为打印提示。UART 转化为 RS232 电平串口与主机通讯，获取打印数据。5 路 I/O 通过逻辑器控制打印机机头工作，其中 1 路用于控制电机走纸，4 路用于控制 4 个打印针工作。2 路 I/O 连接打印机机头，其中 1 路检测打印复位信号，1 路检测打印机脉冲信号。

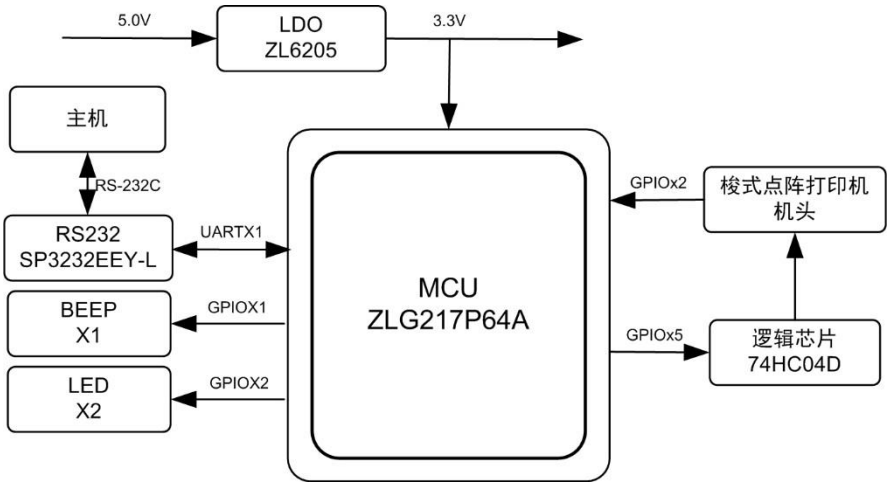

图 2.2 功能框图

2.2.2 资源需求

- 2 路 I/O 实现 LED 状态显示；
- 1 路 I/O 用于驱动蜂鸣器；
- 5 路 I/O 通过逻辑器控制打印机机头，其中 4 路控制打印针，1 路控制电机；
- 2 路 I/O 连接打印机机头，其中 1 路连接用于检测复位信号，1 路检测打印机脉冲信号；
- 1 路 UART 接口转化为 RS232 电平串口用于与主机通信。

2.2.3 优势特点

对比出租车打印机同类产品现用芯片方案，有如下优势：

- MCU 资源丰富（128kB Flash / 20K SRAM）、开发更为灵活，超高性价比；
- 完善齐全的 DEMO 软件和详尽的技术文档，帮助客户快速完成产品开发；
- 提供梭式点阵打印机驱动组件、带缓冲区的 UART 接口通讯等丰富的组件软件，极大简化客户的开发，更专注于核心应用软件的设计；
- 采用 AMetal 软件架构，真正实现跨平台移植，帮助客户快速完成产品升级换代。

2.2.4 推荐器件

ZLG 提供出租车打印机全套 BOM 解决方案，全套 BOM 打包，一站式采购，降低整体成本。主控芯片采用 ZLG IoT MCU ZLG217P64A 设计，详细器件推荐如表 2.1 示。

表 2.1 推荐器件

产品名称	型号	厂家	器件特点
MCU	ZLG217P64A	ZLG	Cortex-M3 内核，20K SRAM/128kB Flash； 单指令周期 32 位硬件乘法器； 运行频率高达 96MHz； 支持宽电压输入 2.0-5.5V； 多路 SPI I2C 等外设接口。
逻辑器	74HC04D	NXP	成本低、支持多达 6 路独立通道。
串口通信	SP3232EEY-L	EXAR	支持波特率高达 200K； 支持 3.3~5.0V 电压范围； 两线制异步串行通信。
LDO	ZL6205	ZLG	成本低、噪音小、静态电流小。

2.3 参考设计

2.3.1 梭式点阵打印机介绍

1. M-150II 打印机

本方案采用爱普生微型打印机中的 M-150II 梭式点阵打印机，打印机体积小巧且高度可靠，其重量约为 60g，但性能依然很强。因为体积小巧，所以 M-150II 满足各种小型设备的打印需求，包括从手持终端到笔记本电脑以及小型测量仪器等。由于运行所需电量较小，这款打印机可以选择采用电池供电，产品如图 2.3 所示。

图 2.3 M-150II 梭式点阵打印机

2. 工作原理

M-150II 梭式点阵打印机，由打印头、电机、定时和复位检测器等组成。其中打印头由 4 个水平放置的打印电磁阀（A.B.C.D）组成。打印头在打印状态下从左侧向右移动，移动

量为每个打印电磁阀 24 个点。当打印头移动时，通过逐个驱动打印电磁阀打点形成一条点线，每个点线的总点数为 96 个点（24 点×4 个打印电磁阀），当打印头从右侧返回左侧时，纸张自动送入 0.35mm（一个间距）重复此点线的打印，如图 2.4 所示。

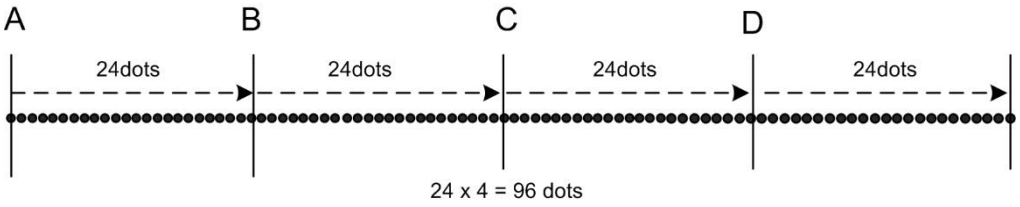

图 2.4　一条点线

电机采用直流有刷电机，当打印机处于待机状态（即非打印状态）时，电机处于暂停状态。定时检测器(Timing Detector)是与电机直接连接的转速计发生器。检测器每个点线产生 168 个输出信号，其中 96 个输出信号对应打印头的点位置，72 个输出信号对应打印头返回。这些输出信号按脉冲排列的波形，用作定时脉冲，如图 2.5 所示。

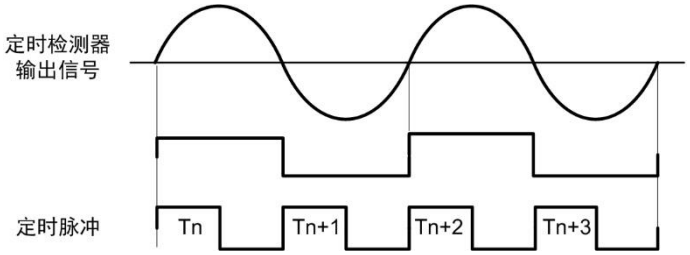

图 2.5　定时脉冲

复位检测器(Reset Detector)具有簧片开关，每条点线都会产生。在每次打印周期中点位置的标准位置，复位检测器输出的信号用作复位，时序图如图 2.6 所示。

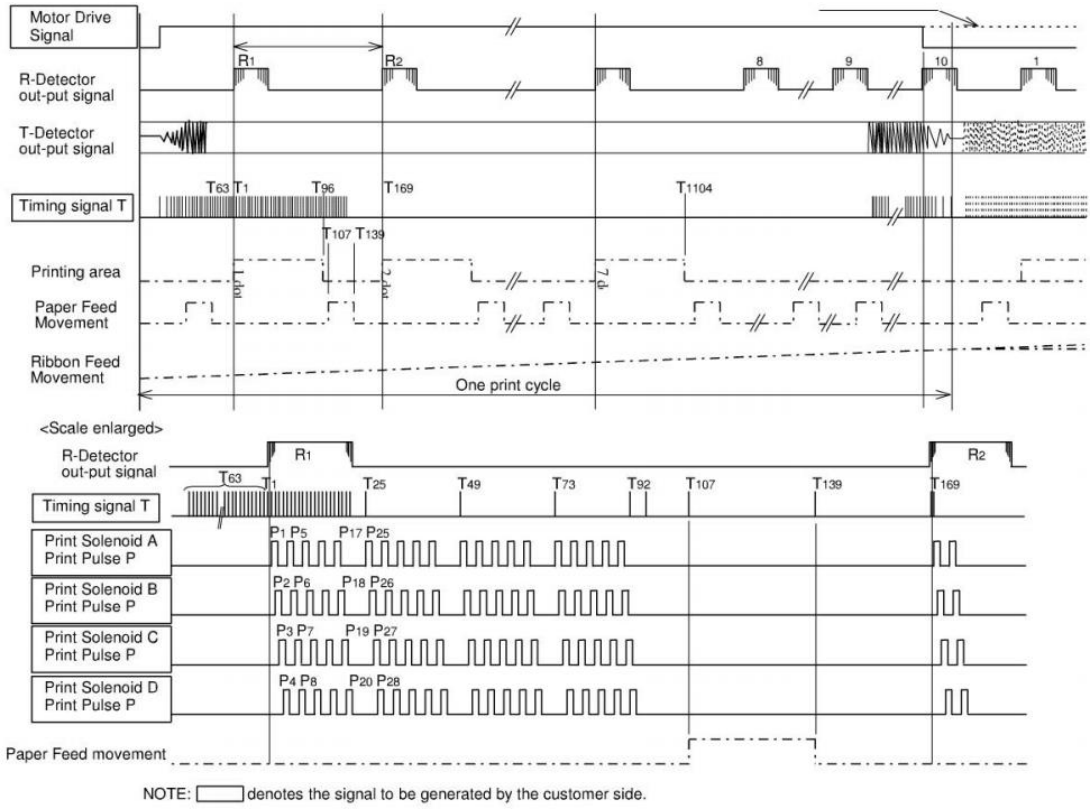

图 2.6 时序图

2.3.2 梭式点阵打印机组件

AMetal 已经集成了 M-150II 打印机组件软件，客户只需配置与打印机相关的信息即可，操作非常简单。在 am_printer_m150.h 文件中，定义了与 M-150II 打印机设备相关信息结构体，详见程序清单 2.1。

程序清单 2.1 打印机配置信息定义

```
1   typedef struct am_printer_m150_info {
2       int     mot_pin;                    // 电机控制引脚
3       int     tir_pin;                    // 时钟脉冲输入引脚
4       int     rst_pin;                    // 复位信号检测引脚
5       int     print_a_pin;                // 电磁阀 A 控制引脚
6       int     print_b_pin;                // 电磁阀 B 控制引脚
7       int     print_c_pin;                // 电磁阀 C 控制引脚
8       int     print_d_pin;                // 电磁阀 D 控制引脚
9       uint8_t *p_addr_buf;                // 指向字库数组，存储字符相对地址
10  } am_printer_m150_info_t;
```

在 am_printer_m150_info 结构体成员中包含了驱动 M-150II 打印机的所有信息，包括电机控制引脚、时钟脉冲输入引脚、复位信号检测引脚、ABCD 电磁阀控制引脚等，在 am_hwconf_printer_m150.c 中进行赋值，对应信息配置示例如程序清单 2.2 所示。

程序清单 2.2 打印机配置信息示例

```
1   static const am_printer_m150_info_t    __g_printer_m150_info = {
```

```
2        PIOB_15,                    // 电机控制引脚
3        PIOB_14,                    // 时钟脉冲输入引脚
4        PIOB_13,                    // 复位信号检测引脚
5        PIOC_6,                     // 电磁阀 A 控制引脚
6        PIOC_7,                     // 电磁阀 B 控制引脚
7        PIOC_8,                     // 电磁阀 C 控制引脚
8        PIOC_9,                     // 电磁阀 D 控制引脚
9        __g_add_buf,                // 指向字符地址缓存
10   };
```

配置 ZLG217P64A 的 PIOB_15 引脚控制电机，PIOB_14 引脚输入时钟脉冲，PIOB_13 引脚检测复位信号，PIOC_6.7.8.9 引脚分别控制电磁阀 A.B.C.D。

AMetal 提供了 M-150II 打印机的驱动函数，其接口函数详见表 2.2。

表 2.2 M-150II 接口函数

函数原型	功能简介
am_printer _m150_handle_t handle = am_printer_m150_inst_init();	实例初始化
void am_printer_m150_print_line_char(am_m150_handle_t handle, unsigned char *data_buf);	打印一行字符
void am_printer_m150_print_line_ chinese (am_m150_handle_t handle, unsigned char *data_buf);	打印一行汉字

其中 handle 为服务句柄，即为初始化 M-150II 打印机获取的句柄，databuf 为指向字符或汉字数据相对地址的指针。

1. 打印一行字符

字符是由 5x7 的点阵组成，将一个打印电磁阀可打印的 24 个点分成四个相等的部分，并且一个部分中的 6 个点用作一列，即用于打印的 5 个点和用于列空间的一个点。因此一个点线由 96 个点形成，其被分成 16 个部分并且可以通过在送纸方向上重复 7 次获得 5×7 点阵的字符，如图 2.7 所示。

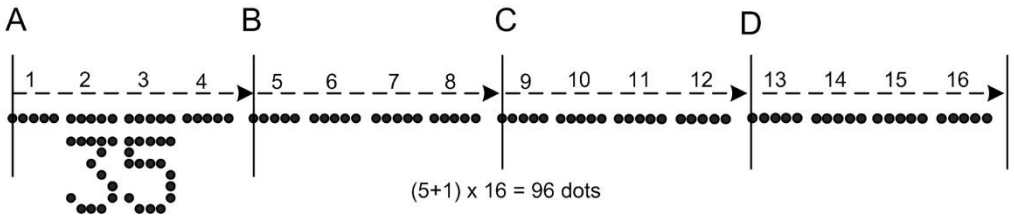

图 2.7 5x7 点阵字符-35

以打印一行 16 个字符中的第 2、3 个字符‘35’为例：

- 字符‘3’在字库中的编码为：0x07, 0xef, 0xdf, 0xcf, 0xf7, 0x77, 0x8f,
- 字符‘5’在字库中的编码为：0x07, 0x7f, 0x0f, 0xf7, 0xf7, 0x77, 0x8f,

字符转换过程详见表 2.3。

表 2.3 字库字符转换示例

字符‘3’	二进制	0 的位	字符‘5’	二进制	0 的位
0x07	00000111	00000___	0x07	00000111	00000___
0xef	11101111	___0____	0x7f	01111111	0_____
0xdf	11011111	__0_____	0x0f	00001111	0000____
0xcf	11101111	___0____	0xf7	11110111	____0___
0xf7	11110111	____0___	0xf7	11110111	____0___
0x77	01110111	0___0___	0x77	01110111	0___0___
0x8f	10001111	_000____	0x8f	10001111	_000____

打印一行字符的函数定义如程序清单 2.3 所示。

程序清单 2.3 打印字符函数定义

```
1  void am_printer_m150_print_line_char(am_m150_handle_t  handle,
2                                       unsigned char*     data_buf)
3  {
4      unsigned char i;
5      am_m150_dev_t *p_m150_dev = handle;
6      // 连接时钟脉冲引脚中断服务函数
7      am_gpio_trigger_connect(p_m150_dev->p_info->tir_pin,
8                              __gpio_isr_handler,
9                              p_m150_dev);
10     // 配置时钟脉冲引脚中断触发方式
11     am_gpio_trigger_cfg(p_m150_dev->p_info->tir_pin,
12                         AM_GPIO_TRIGGER_BOTH_EDGES);
13     // 寻找第一行地址
14     __addr_search(0, p_m150_dev->p_info->p_addr_buf, data_buf);
15     am_gpio_set(p_m150_dev->p_info->mot_pin, 0);    // 打开电机
16     am_udelay (10);
17     for (i = 1; i <= 7; i++) {                      // 循环打印 7 行
18         __print_point_line(p_m150_dev);             // 打印一行墨点
19         if(i < 7){
20             // 寻找下一行地址
21             __addr_search(i, p_m150_dev->p_info->p_addr_buf, data_buf);
22         }
23     }
24     __print_end(p_m150_dev);                        //打印结束
25 }
```

2. 打印一行汉字

汉字是由 11x12 的点阵组成，将一个打印电磁阀可打印的 24 个点分成 2 个相等的部分，并且一个部分中的 12 个点用作一列，即用于打印的 11 个点和用于列空间的一个点。因此一个点线由 96 个点形成，其被分成 8 个部分并且可以通过在送纸方向上重复 12 次获得 11×12 点阵的汉字，如图 2.8 所示。

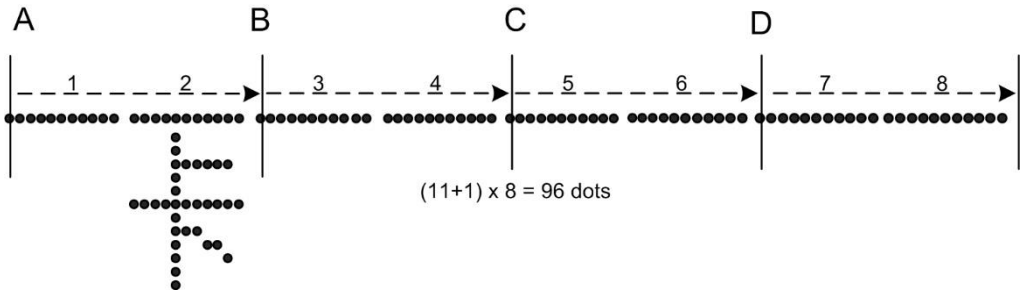

图 2.8　11x12 点阵汉字-卡

以打印一行 8 个汉字中的第 2 个汉字‘卡’为例，汉字‘卡’在字库中的编码为：

```
0x08,0x08,0x0f,0x08,0x08,0xff,0x08,0x0e,0x09,0x08,0x08,0x08,
0x00,0x00,0xc0,0x00,0x00,0xe0,0x00,0x00,0x80,0x40,0x00,0x00,
```

汉字转换过程详见表 2.4。

表 2.4　字库汉字转换示例

汉字‘卡’		二进制		1 的位	
0x08,	0x00,	00001000	00000000	____1___	_____
0x08,	0x00,	00001000	00000000	____1___	_____
0x0f,	0xc0,	00001111	11000000	____1111	11_____
0x08,	0x00,	00001000	00000000	____1___	_____
0x08,	0x00,	00001000	00000000	____1___	_____
0xff,	0xe0,	11111111	11100000	11111111	111_____
0x08,	0x00,	00001000	00000000	____1___	_____
0x0e,	0x00,	00001110	00000000	____111_	_____
0x09,	0x80,	00001001	10000000	____1__1	1_____
0x08,	0x40,	00001000	01000000	____1___	_1_____
0x08,	0x00,	00001000	00000000	____1___	_____
0x08,	0x00,	00001000	00000000	____1___	_____

打印一行汉字的函数定义如程序清单 2.4 所示。

程序清单 2.4　打印汉字函数定义

```
1   void am_printer_m150_print_line_chinese(am_m150_handle_t    handle,
2                                           unsigned char*          data_buf)
3   {
4       unsigned char i;
5       am_m150_dev_t *p_m150_dev = handle;
6       // 连接时钟脉冲引脚中断服务函数
7       am_gpio_trigger_connect(p_m150_dev->p_info->tir_pin,
8                               __gpio_isr_handler,
9                               p_m150_dev);
10      // 配置时钟脉冲引脚中断触发方式
11      am_gpio_trigger_cfg(p_m150_dev->p_info->tir_pin,
```

```
12                                  AM_GPIO_TRIGGER_BOTH_EDGES);
13          // 寻找第一行的地址
14          __addr_search_chinese(0, p_m150_dev->p_info->p_addr_buf, data_buf);
15          am_gpio_set(p_m150_dev->p_info->mot_pin, 0);    // 打开电机
16          am_udelay (10);
17          for (i = 1; i <= 12; i++) {                     // 循环打印 12 行
18              __print_point_line(p_m150_dev);             // 打印一行墨点
19              if(i < 12){
20                  // 寻找下一行汉字的地址
21                  __addr_search_chinese(i, p_m150_dev->p_info->p_addr_buf, data_buf);
22              }
23          }
24          __print_end(p_m150_dev);                        // 打印结束
25      }
```

2.3.3 带缓冲区的 UART 接口

出租车打印机通过 UART 转化为 RS232 电平串口与主机通信。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。带缓冲的 UART 接口组件使用请参考第 1.3.3 章节。

第3章 读卡应用方案

3.1 产品概述

读卡方案，产品实物如图 3.1 所示。 采用 FM17520 作为读卡芯片，通过 SPI 与主控芯片通信，读卡距离达 5cm，支持多种卡片。

采用非接触式读卡安全性较高，使用方便。非接触式读卡是目前主流的读卡方案，应用广泛，支持多种应用场合，可以存储用户信息，离线管理数据和上传等。读卡方案中的主控采用 ZLG116N32A 设计。

图 3.1 产品展示

3.2 方案介绍

3.2.1 功能框图

读卡功能框图如图 3.2 所示，读卡系统采用 DC12V 供电，经过 LDO 转化为 3.3V 和 5.0V 给 MCU 和外设器件供电，外置两个 LED 状态指示灯。通过 SPI 和读卡芯片通信，从而对卡片进行读写。

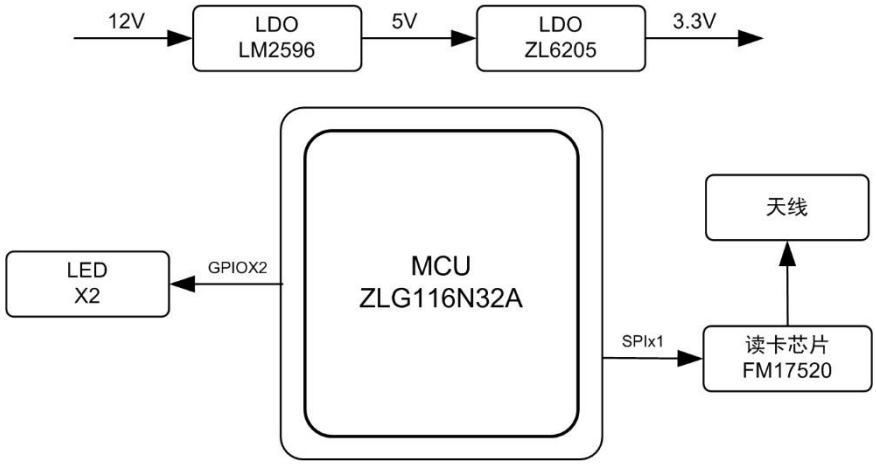

图 3.2 功能框图

3.2.2 资源需求

- 2 路 I/O 实现 LED 状态显示；
- 1 路 SPI 接口用于和 FM17520 通信。

3.2.3 优势特点

对比读卡系统同类产品现用芯片方案，有如下优势：

- MCU 资源丰富（64kB Flash / 8K SRAM）、开发更为灵活，超高性价比；
- 完善齐全的 DEMO 软件和详尽的技术文档，帮助客户快速完成产品开发；
- 提供 SPI 驱动通讯组件等丰富的组件软件，极大简化客户的开发，更专注于核心应用软件的设计；
- 采用 AMetal 软件架构，真正实现跨平台移植，帮助客户快速完成产品升级换代。

3.2.4 推荐器件

ZLG 提供读卡系统全套 BOM 解决方案，全套 BOM 打包，一站式采购，降低整体成本。主控芯片采用 ZLG IoT MCU ZLG116N32A 设计，详细器件推荐如表 3.1 所示。

表 3.1 推荐器件

产品名称	型号	厂家	器件特点
MCU	ZLG116N32A	ZLG	Cortex-M0 内核，64kB Flash / 8K SRAM； 运行频率高达 48MHz； 支持宽电压输入 2.0~5.5V； 多路 UART、SPI、I2C 等外设接口。
读卡芯片	FM17520	复旦微	非接触式读卡、低电压、长距离。
LDO	ZL6205	ZLG	成本低、噪音小、静态电流小。
LDO	LM2596	TI	成本低、噪音小、静态电流小。

3.3 参考设计

3.3.1 读卡电路设计

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 3.3 所示。

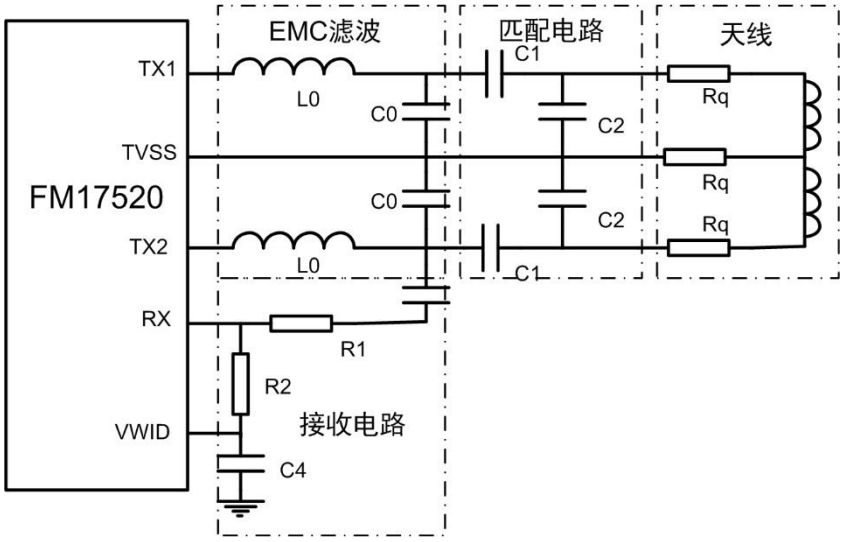

图 3.3 读卡电路

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。

3.3.2 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 3.2。

表 3.2 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。

2. 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;
```

3.3.3 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 3.3 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    */
```

3.3.4 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 3.4。

表 3.4 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
```

3.3.5 密钥和权限控制

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。

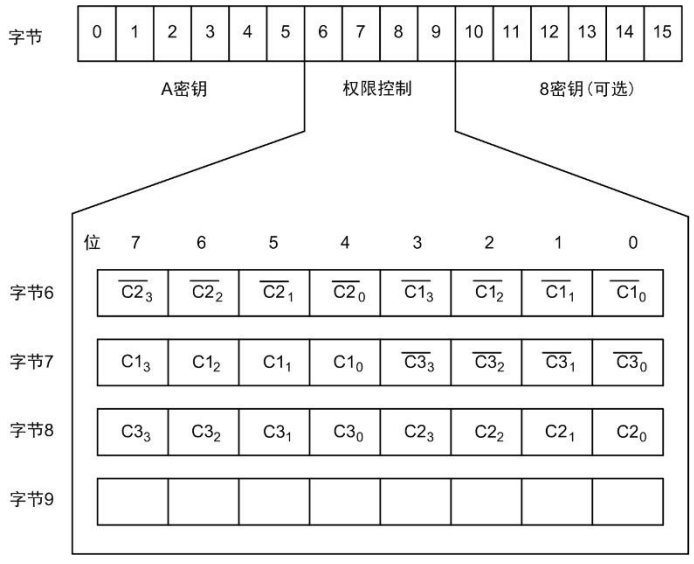

图 3.4 尾块格式

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 3.4。

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、$\overline{C1}$、$\overline{C2}$、$\overline{C3}$）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$\overline{C10}$ 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 3.5。

表 3.5 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 3.6。

表 3.6 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 3.5。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。

更多读卡功能组件的使用请参考《面向 AMetal 框架与接口的编程》第 5.4 章节。

第4章 智能门锁应用方案

4.1 产品介绍

智能门锁产品实物如图 4.1 所示，广泛应用于银行、政府部门，以及酒店、学校宿舍等。智能门锁区别于传统机械锁，在用户安全性、识别、管理性方面更具优势。

智能门锁内部集成非接触式读卡模块、蓝牙模块、NB 模块，支持刷卡、手机 APP 蓝牙控制、远程控制多种开门方式。本智能门锁方案主控采用 KL16Z128，通过 SPI 接口与读卡芯片 FM17550 通信，实现非接触式读卡，通过 UART 接口与 BLE ZLG52810、NB ZM7100M 通信，实现 APP 开锁、远程控制及管理。

图 4.1 产品展示

4.2 方案介绍

4.2.1 功能框图

智能门锁功能框图如图 4.2 所示，智能门锁通过 7.8V 锂电池供电，经过 LDO 转化为 3.3V 给 MCU 供电，外置 2 个 LED 状态指示灯和 1 个蜂鸣器，支持触摸按键，通过 4 个 I/O 控制门锁电机，外扩 EEPROM 用于存储用户数据，RTC 用于时间管理。主控通过 UART 接口与 BLE 模块、NB 无线模块通信，用于实现手机 APP 开锁、远程控制，方便用户管理和使用。

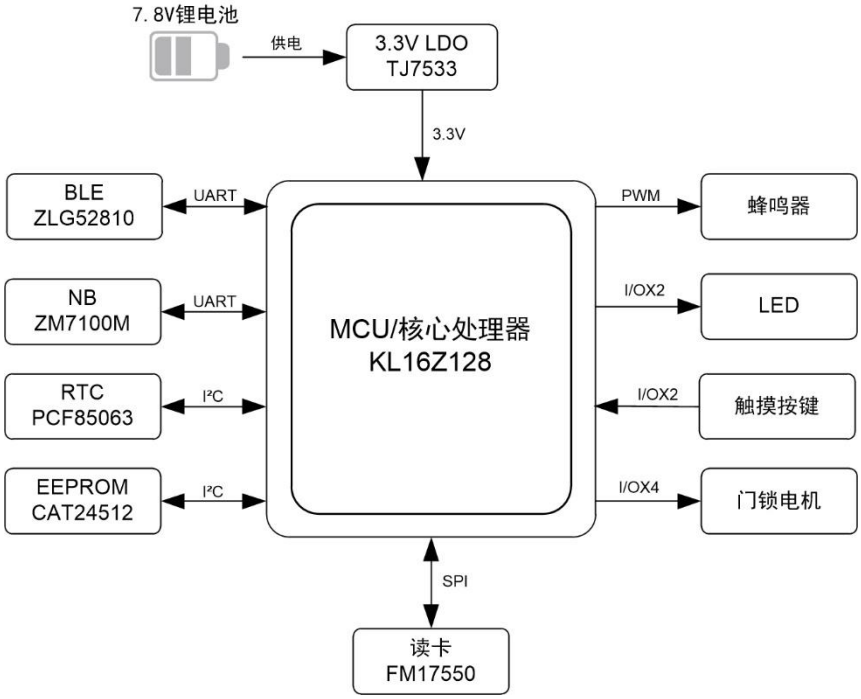

图 4.2 功能框图

4.2.2 资源需求

- 2 路 I/O 实现 LED 状态显示；
- 1 路 I/O 实现蜂鸣器发音提示；
- 2 路 I/O 实现触摸按键检测输入；

- 4 路 I/O 用于控制门锁电机；
- 1 路 SPI 接口用于与读卡芯片 FM17550 通信；
- 1 路 I2C 接口用于外扩 EEPPROM；
- 1 路 I2C 接口用于读取 RTC 时钟；
- 1 路 UART 接口用于与 NB ZM7100M 模块通信；
- 1 路 UART 接口用于与 BLE ZLG52810 模块通信。

4.2.3 优势特点

对比智能门锁同类产品现用芯片方案，有如下优势：

- MCU 资源丰富（128KB Flash / 16K SRAM）、开发更为灵活，超高性价比；
- 完善齐全的 DEMO 软件和详尽的技术文档，帮助客户快速完成产品开发；
- 提供 SPI 读卡驱动、EEPPROM 驱动、RTC 驱动、带缓冲区的 UART 接口通讯等丰富的组件软件，极大简化客户的开发，更专注于核心应用软件的设计；
- 采用 AMetal 软件架构，真正实现跨平台移植，帮助客户快速完成产品升级换代。

4.2.4 推荐器件

ZLG 提供智能门锁全套 BOM 解决方案，全套 BOM 打包，一站式采购，降低整体成本。主控芯片采用 NXP 低功耗芯片 KL16Z128 设计，详细器件推荐如表 1.1 所示。

表 4.1 推荐器件

产品名称	型号	厂家	器件特点
MCU	KL16Z128	NXP	Cortex-M0+内核，128 KB Flash / 16K SRAM； 运行频率高达 48MHz； 支持宽电压输入 1.71–3.6 V； 低功耗，低功耗停止模式典型电流 2.71μA； 多路 UART、SPI、I2C 等外设接口。
读卡芯片	FM17550	复旦微	非接触式读卡、低电压、长距离。
LDO	TJ7333	MPS	成本低、噪音小、静态电流小。
EEPROM	CAT24512	安森美	宽电压输入 1.7~5.5V、低功耗、静态电流小。
RTC	PCF85063	NXP	成本低、低功耗、静态电流小。
BLE	ZLG52810	ZLG	成本低、低功耗、静态电流小。
NB	ZM7100M	ZLG	成本低、低功耗、静态电流小。

4.3 参考设计

4.3.1 电路设计

4.3.2 RTC 实时时钟

本智能门锁方案采用 NXP 半导体公司的 PCF85063，是一款低功耗实时时钟芯片，它提供了实时时间的设置与获取、闹钟、可编程时钟输出、中断输出等功能。

PCF85063 电路如图 4.1,其中 SCL 和 SDA 为 I2C 接口引脚，VDD 和 VSS 分别为电源和地；OSCI 和 OSCO 为 32.768KHz 的晶振连接引脚，作为 PCF85063 的时钟源；CLKOUT 为时钟信号输出引脚，供外部电路使用；INT 为中断引脚，主要用于定时、闹钟等功能。PCF85063 的 7 位 I2C 从机地址为 0x51。

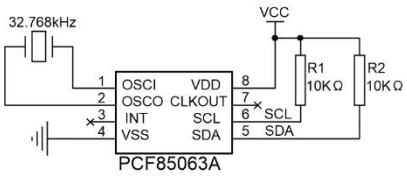

图 4.3 PCF85063 电路原理图

AMetal 提供了 PCF85063 组件及 RTC 通用接口，PCF85063 初始化完成后，即可调用通用接口设置时间和获取时间，其接口函数详见表 4.2。

表 4.2 RTC 接口函数

函数原型	功能简介
am_rtc_handle_t am_microport_rtc_std_inst_init (void);	初始化
am_static_inline int am_rtc_time_set (am_rtc_handle_t handle, am_tm_t *p_tm);	设置时间
am_static_inline int am_rtc_time_get(am_rtc_handle_t handle, am_tm_t *p_tm);	获取时间

1. 初始化

在使用 PCF85063 前，必须调用初始化函数完成 PCF85063 的初始化操作，以获取对应的操作句柄，进而才能使用 PCF85063 的各种功能。PCF85063 的实例初始化函数原型（am_hwconf_microport_rtc.h）为：

```
am_rtc_handle_t am_microport_rtc_std_inst_init (void);
```

使用无参数的 PCF85063 实例初始化函数，即可获取 RTC 实例句柄，进而通过 RTC 通用接口使用 PCF85063 的各种功能。

2. 设置时间

该函数意在设置 RTC 的当前时间，其函数原型为：

```
am_static_inline int am_rtc_time_set (am_rtc_handle_t handle, am_tm_t *p_tm);
```

其中，handle 为 PCF85063 的实例句柄，p_tm 为待设置的时间值的指针。返回 AM_OK，表示设置成功，反之失败。其类型 am_tm_t 是在 am_time.h 中定义的细分时间的结构体类型，用于表示年/月/日/时/分/秒等信息，结构体原型为：

```
typedef struct am_tm {
    int tm_sec;                 // 秒，0 ~ 59
    int tm_min;                 // 分，0 ~ 59
    int tm_hour;                // 小时， 0 ~ 23
    int tm_mday;                // 日期， 1 ~ 31
    int tm_mon;                 // 月份， 1 ~ 12
    int tm_year;                // 年
```

```
    int tm_wday;                    // 星期
    int tm_yday;                    // 天数
    int tm_isdst;                   // 夏令时
} am_tm_t;
```

其中，tm_year 表示年，1900 年至今的年数，其实际年为该值加上 1900。tm_wday 表示星期，0~6 分别对应星期日~星期六。tm_yday 表示 1 月 1 日以来的天数（0~365），0 对应 1 月 1 日。tm_isdst 表示夏令时，夏季将调快 1 个小时。如果不使用，可设置为-1。设置当前时间的程序详见程序清单 4.1，星期等附加的一些信息无需用户设置，主要便于在获取时间是获得更多的信息。

程序清单 4.1　设置时间范例程序

```
1   am_local am_tm_t __g_current_time = {
2       55,                         // 秒
3       59,                         // 分
4       11,                         // 小时
5       9,                          // 日期
6       1-1,                        // 月份
7       2019-1900,                  // 年
8       0,                          // 星期
9       0,                          // 天数
10      -1                          // 夏令时
11  };
12  am_rtc_time_set(rtc_handle, &__g_current_time);
```

3. 获取时间

该函数意在设置 RTC 的当前时间，其函数原型为：

```
am_static_inline int am_rtc_time_get (am_rtc_handle_t handle, am_tm_t *p_tm);
```

其中，handle 为 PCF85063 的实例句柄，p_tm 为指向时间值的指针。返回 AM_OK，表示获取时间成功，反之失败。获取时间的程序详见程序清单 4.2。

程序清单 4.2　获取时间范例程序

```
1   am_tm_t time;
2   am_rtc_time_get(rtc_handle, &time);
```

关于 AMetal RTC 实时时钟组件、通用接口的实现和使用请参考《面向 AMetal 框架与接口的编程》第 6.3 章节。

4.3.3 EEPROM 储存器

本智能门锁方案需要外扩 EEPROM 用于存储用户数据，采用安森美的 CAT24C512。CAT24C512 总容量为 512K（512*1024）bits，即 65536（512*1024/8）字节。每个字节对应一个储存地址，因此其储存数据地址范围为 0x0000~0xFFFF。CAT24C512 页（page）的大小为 128 字节，分 512 页。支持按字节读写和按页读写，按页读写一次可高达 128 字节。

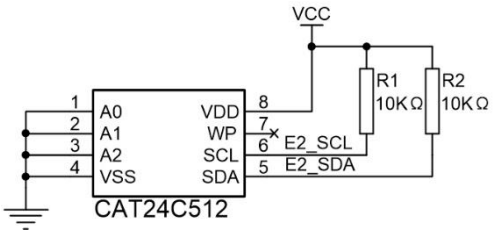

图 4.4 CAT24C512 电路原理图

CAT24C512 的通信接口为标准的 I²C 接口，仅需 SDA 和 SCL 两根信号线，其电路原理图如图 4.4。

4.3.4 52810 BLE 组件

第5章 待续