**编号**

JNDX_M

**本科生毕业设计（论文）**

**题目：** 移动便携式测量系统开发

机械工程 学院 机械电子工程 专业

学 号 1041180920

学生姓名 肖竟成

指导教师 郑坤明 讲师

卢东升 工程经理

二〇二二 年 四月

摘 要

针对野外作业以及车上作业等非实验室环境中的作业环境, 从当前操作复杂、设备冗余、启停延迟大、无法脱离电源等十分不利于实验人员在户外做测试的问题出发。结合效率与生成，使用STM32F407系列芯片作为核心，详细分析定制化的硬件设备；深度剖析软件的设计；MCU上采用FreeRTOS作为操作系统，兼顾实时性与标准化；HMI方面使用一体化串口屏制作GUI，便捷操作与及时反馈。技术上实现了采集系统的一体化、便捷化，定制化硬件设计结构精简，集成度高，十分具有灵活性，可以适应不同的使用场景，软件设计标准化，便于后续的迭代与兼容，综上进行嵌入式开发。

**关键词：**STM32F407；FreeRTOS；数据传输；GUI；嵌入式

ABSTRACT

For non-laboratory environments such as field work and on-board work, the current problems such as complex operation, redundant equipment, large start-stop delay, and inability to disconnect from the power supply are very unfavorable for experimenters to test outdoors. Combining efficiency and generation, using STM32F407 series chips as the core, detailed analysis of customized hardware devices; in-depth analysis of software design; MCU uses FreeRTOS as the operating system, taking into account real-time and standardization; HMI uses an integrated serial screen to make GUI, Convenient operation and timely feedback. Technically, the integration and convenience of the acquisition system are realized. The customized hardware design structure is simplified, the integration is high, and it is very flexible. It can adapt to different usage scenarios. The software design is standardized, which is convenient for subsequent iteration and compatibility. Based on the above, the embedded device is developed.

**Keywords**: STM32F407； FreeRTOS； Data transmission; GUI； Embedded development

目 录

[摘 要 I](#_Toc104051578)

[ABSTRACT II](#_Toc104051579)

[目 录 i](#_Toc104051580)

[第1章 绪论 1](#_Toc104051581)

[1.1 国外研究状况 1](#_Toc104051582)

[1.2 国内研究状况 1](#_Toc104051583)

[1.3 项目需求分析 2](#_Toc104051584)

[1.4 本文结构介绍 2](#_Toc104051585)

[第2章 系统架构 4](#_Toc104051586)

[2.1 总体架构 4](#_Toc104051587)

[2.2 硬件架构 4](#_Toc104051588)

[2.3 软件架构 5](#_Toc104051589)

[2.4 本章小结 6](#_Toc104051590)

[第3章 硬件设计 7](#_Toc104051591)

[3.1 主控芯片STM32F407 7](#_Toc104051592)

[3.1.1 STM32F407的SD卡接口描述 7](#_Toc104051593)

[3.1.2 STM32F407的串口描述 8](#_Toc104051594)

[3.2 HMI屏幕 8](#_Toc104051595)

[3.3 采集模块 9](#_Toc104051596)

[3.3.1 初版设计 9](#_Toc104051597)

[3.3.2 初版测试 11](#_Toc104051598)

[3.3.3 第二版设计与测试 13](#_Toc104051599)

[3.4 PCB设计 15](#_Toc104051600)

[3.4.1 硬件架构与原理图设计 15](#_Toc104051601)

[3.4.2 PCB layout 23](#_Toc104051602)

[第4章 软件设计 28](#_Toc104051603)

[4.1 FreeRTOS 28](#_Toc104051604)

[4.1.1 FreeRTOS介绍 28](#_Toc104051605)

[4.1.2 FreeRTOS porting 29](#_Toc104051606)

[4.1.3 FreeRTOS 启动流程 29](#_Toc104051607)

[4.1.4 FreeRTOS 内存管理 30](#_Toc104051608)

[4.2 FAT文件系统 31](#_Toc104051609)

[4.2.1 FAT介绍 31](#_Toc104051610)

[4.2.2 FAT移植 31](#_Toc104051611)

[4.2.3 SDIO简介 32](#_Toc104051612)

[4.3 用户程序设计 35](#_Toc104051613)

[4.4 GUI设计 36](#_Toc104051614)

[4.5 本章小结 38](#_Toc104051615)

[第5章 结论与展望 39](#_Toc104051616)

[5.1结论 39](#_Toc104051617)

[5.2不足之处及未来展望 39](#_Toc104051618)

[参考文献 41](#_Toc104051619)

[致 谢 42](#_Toc104051620)

[附录A： 作者在校期间发表的论文 43](#_Toc104051621)

[附录B： 主程序 44](#_Toc104051622)

第1章 绪论

1.1 国外研究状况

采集测量系统在1956年前后就已出现，最早起源于美国下军事领域对其测试系统的研究，为了在测试中不依赖操作人员的熟练度，并且整个流程是由相应的测试系统自动化完成。因为这种数据采集系统可以被灵活的适用于各类的使用场景，它可以满足许多传统测试方法不方便甚至无法完成的数据采集任务，因此得到了初步的肯定。国外的成套数据采集系统的产品于60年代后期就已进入市场，但本阶段的采集系统多为专用系统，其使用的通用性和便捷性并不佳。

70年代末，伴随着微型机的诞生,采集器、仪器等采集设备也慢慢地可以和微型计算机一并集成为一个泛用性强的大数据信息收集体系。由于有其自身较好的功能特性设计和方便简单易用的界面操作,数据采集系统也得到了一个巨大飞跃的技术发展,迅速地超过了传统的自动检测系统和专用的数据自动采集系统。此后，数据采集系统现在逐渐被分成了二个大类别,一个是实验室系统中的数据自动采集系统,另一个是工业中使用的现场数据采集系统。两者的主要区别体现在数据传输所使用的总线的不同上。实验室的数据采集系统大多采用并行总线，而工业领域的数据主要采用串行总线采集。

约10年后，由于微型计算机应用的日益普及,相应地的,数据自动采集系统技术也获得到了更为飞速的技术进展,产生出了数量众多的并具有高度泛用性特点的自动数据自动采集处理系统,其组成系统一般大致上存在着二种结构形式,一种结构是由仪器仪表、采集器、通用的接口总线系统以及微型计算机等所构成。如ICE625(GPIB)总线系列便是其中一种典范,该类产品大多可应用于实验室场景系统中。第二种总线是指由数据采集卡、总线控制器和总线计算机等构成,如STD总线。该接口系统将对应的接口卡加装在专门的机箱中，由一台中央计算机控制，为积木式结构。在这两种应用中，倘若采集的目标发生改变。仅需将新的传感器介入系统即可完成硬件重构，显而易见的，这种设计会使得整个系统相较专用系统更为灵活。80年代后期，PLC、大规模集成电路以及单片机的出现让数据采集系统有了较大的优化，使得整体体积变小，生产、操作成本降低、功能更加强大。

上世纪末，数据采集系统在军事、宇宙航行、工业等领域被部分国家广泛采用。随着集成电路的飞速发展，更多的高可靠性、高性能的产品逐渐出现，如DAS。数据采集已经成为了一个独立的技术在工业上被普遍使用。

串行总线数据采集系统向分布式系统结构和智能化方向发展，可靠性不断提高。[1]在嵌入式领域发展出了如RS232、RS485和USART等串行通信方式。

1.2 国内研究状况

20世纪70年代后，我国也逐渐开发出了一些用于数据采集的设备，有单通道的也有双通道的，其配套的开发环境基本包含设备维修管理和基本频谱分析这两块，日常的采集监控需求都可以满足，并且在出现故障时会有基本的故障诊断，和国外数据采集的初期水平相当。

直到目前，与国外的数据采集技术相比，我国仍有着一定的差距，主要在如下几点：

* 受国内的传感器自身性能限制，可以分析的范围较小，部分机械或轴承的分析有着一定的困难。
* 受采集器的内存所限，其本身信号处理能力不强，一般在现场环境中只能完成一些简单的诊断与数据记录，更详细、具体的分许需要在更强的机器上实现。
* 受软件水平所限，其诊断系统与人机交互性较弱，还有待加强。

数据采集技术是整个工业实现自动化的核心部分，也是其最前沿的部分，系统可以测试的精度、测试的速度和实现的成本是其的核心因素。高速、实时的完成数据的采集在很多方面都有其相应的使用场景，如：运动控制、医疗设备和军工等诸多领域均有很大的应用占比。这些行业对高速数据采集系统的要求也加快了技术的更进一步发展与新产品的出现。

在工业领域，各种控制设备、仪器仪表的软件设计中要求严格的实时性，实时性是嵌入式系统的主要特征。[2]国内近些年也逐步发展了一些嵌入式实时操作系统，以RT-Thread为例，其自2011年发布了0.4版开始，在近些年有了较快的发展，越来越多的国产项目选择使用其作为项目的操作系统。

1.3 项目需求分析

近年来，越来越多的户外作业，部门同事对于现有的采集方式即实验室模式倍感繁琐，故多次会议讨论解决方案。

实验人员在野外主要需要测量温度、压力、缸压和增压器转速以及发动机转速这4种模拟量。测量系统中的温度信号主要采样热电阻和热电偶，其精度为：热电阻±0.15℃，热电偶正负0.4%t；压力主要采样4~20mA或者0~10V的电信号，其精度为正负-0.3%F.S；缸压和增压器转速同样为0~10V信号；发动机转速为正弦波或矩形波，由磁电式或者霍尔传感器输出。、

主要工作是将每个模块定制化与集成化，减少总体体积，在不丢失整体性能的前提下尽可能的使整个系统更加的人性化，强化整个设备的人机交互性能用以提高用户体验，增加设备的通用性，用以适应不同的使用场景，以及增加安全性校验，保障设备的信息安全。

1.4 本文结构介绍

本文绪论对国内外数据采集系统进行了研究状况介绍，对本项目的需求进行了分析。

第二章的系统架构部分首先系统性的将系统拆分为各个功能模块，接着对硬件部分的架构进行阐述，介绍了每个模块之间如何通讯，模块间相互的关系，然后分析了本项目的软件架构，梳理了软件的层次结构。

第三章是本项目的硬件设计，介绍了其使用的主控STM32F407系列MCU，并对MCU的SD卡接口、串口部分做了详细的介绍，还提及了项目中使用的HMI屏幕，对其进行了一定的说明。紧接着是系统采集模块的设计，第一代使用AD7190尝试完成，但是测试分析时发现其并不能很好的完成项目需要，因此后续更换使用了SP-4017+采集模块。后续对项目的PCB设计进行了详细介绍，着重分析了其的原理图，对于其layout也简单的进行了说明。

第四章中对系统软件的设计进行了剖析，先是介绍了系统使用的嵌入式实时操作系统FreeRTOS，从移植的角度对其进行了深入的分析，接着是对系统使用的文件系统进行介绍，从移植、硬件接口、SD卡使用的时序逻辑等对FAT文件系统详细的解读。然后分析用户程序，即主函数的工作逻辑，并在最后介绍了本系统的GUI设计。

第五章回顾了整个项目的开发，总结了在其实现过程中的不足与体会。

第2章 系统架构

2.1 总体架构

为充分满足公司项目迭代须要，本系统采选模块化、任务导向式开发，本人负责系统软硬件主体架构开发。多次研讨后初步决定系统架构如图2-1所示。

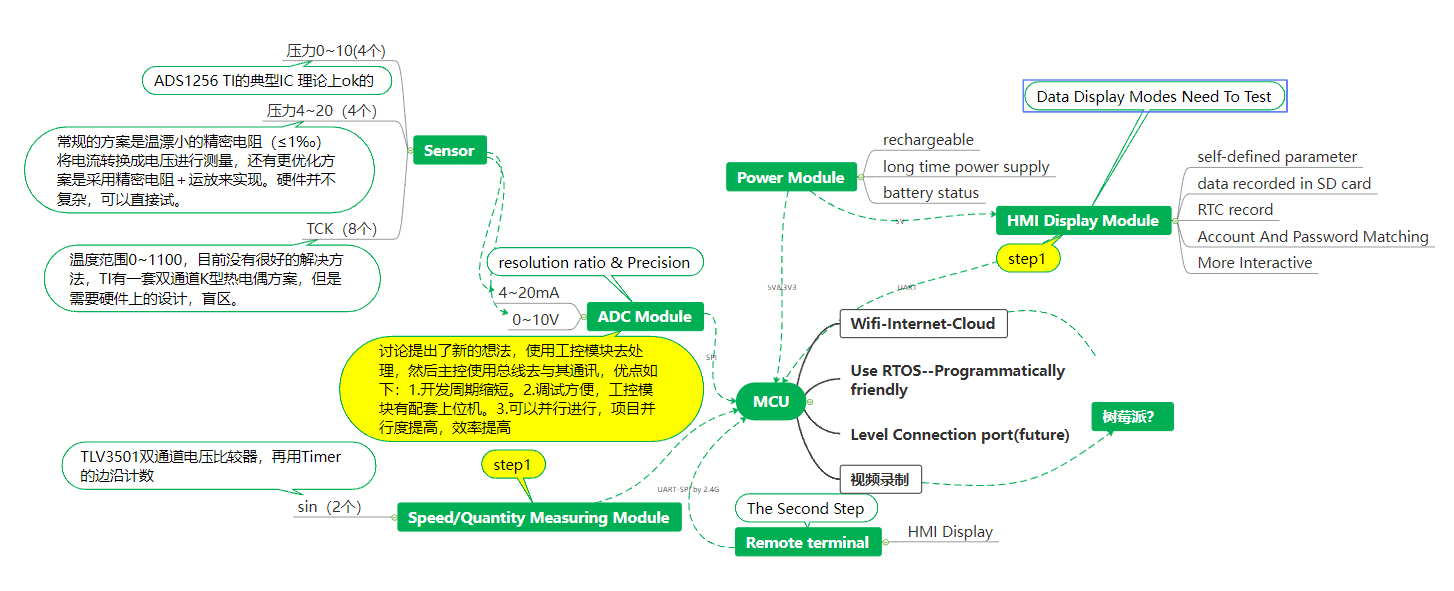


图2-1.系统整体架构导图

大致分为如下几个模块：Sensor模块涵盖所有可能会使用的传感器，主要包括采集缸压需要的压力传感器以及采集温度需要的热电偶传感器。Power模块负责整个系统的供电，整个系统将可以持续工作12小时以上，也留有220V和380V的电源接口，可以在有电的环境中持续使用。HMI模块负责系统的人机交互，包括部分参数的配置、传感器配置、实时数据显示等。Remote模块为远程监视器模块，其与HMI模块较为类似，但十分小巧便携且为无线模块，可以在监控的同时观察到发动机具体的情况。[1 2]树莓派模块主要负责图像采集、数据远程传输等。MCU模块为系统核心模块，其将诸多功能模块相互连接，协调模块工作。

2.2 硬件架构

本次采用的MCU为ST公司的STM32F4系列的芯片，该芯片采用ARM（Advanced RISC Machines）v7架构，使用哈佛总线与三级流水线，处理能力大大提升，并且在数字信号处理扩展方面可以使用单周期16位或32位的MAC、单周期双16位MAC、8位或16位的SIMD运算、硬件除法等，存在可选FPU（float point unit）。该MCU同时包含2种指令集，分别为16位的TURMB以及32位的ARM指令集并且他们之间可以在运行过程中切换。同时MCU还集成了诸多外设，例如硬件I2C、SPI等总线，多个定时器，16位精度的ADC、DAC，IO口复用器、DMA等强大而精巧。

以其为核心，配合sd卡、CAN收发模块等外设，定制化PCB（Printed Circuit Board）作为MCU模块。sensor采集模块初期采用ADC7606模块采集电压信息，测试后续测试发现其稳定性不佳，若使用软件滤波会较为影响开发进度与MCU模块算力，后期替换使用工控模块。HMI模块使用串口屏，其优势在于通讯标准化，编程较为方便，功耗较低。

其硬件架构如图2-2所示。

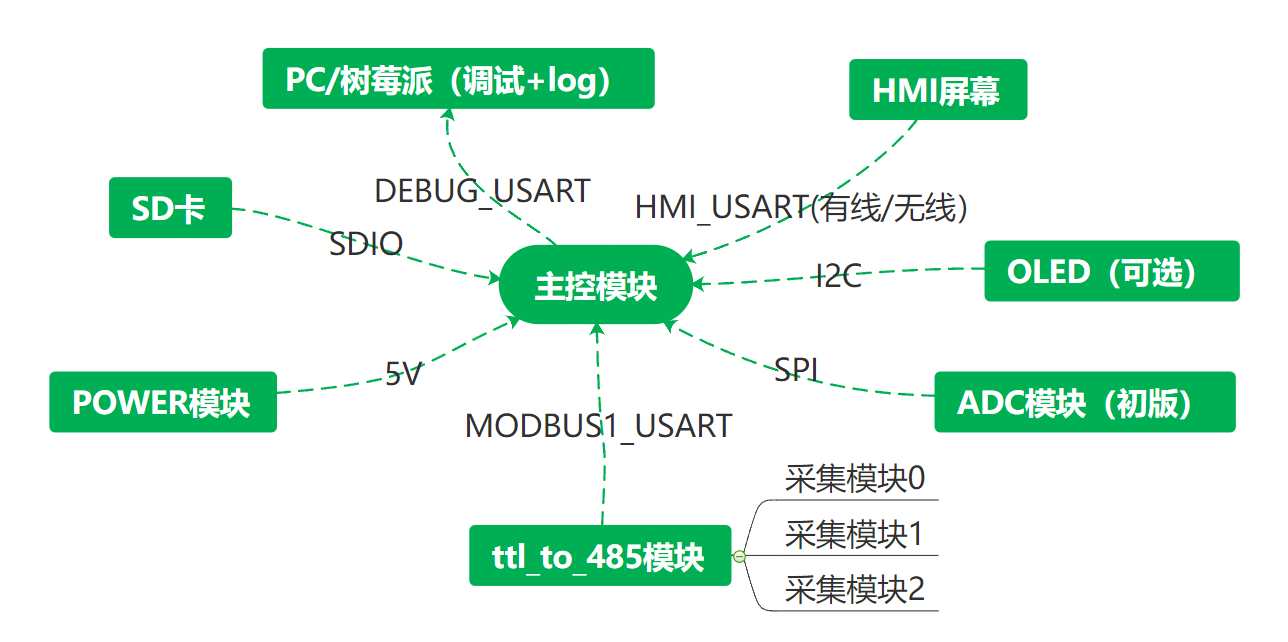


图2-2. 硬件架构简图

主控模块为整个系统的桥梁，将所有其他模块连接。HMI屏幕通过串口与主控连接并通讯，在开发计划中该模块可以有线连接也可以无线连接达到远程监控的效果。OLED是一个可选的模块，其使用I2C总线与主控进行通讯，该模块是为了在箱体上显示部分测试箱信息的，如电量提醒、运行时间等信息。ADC模块为初版设计中的部分，其使用SPI与主控通讯。ttl\_to\_485是将ttl通讯电平转化为485通讯电平的模块，相较于ttl，485使用的差分信号逻辑更适合长距离、高速通讯，其也是通过串口与主控相连，采集模块串接至485，采用串行式总线。POWER模块是整个系统的电源管理部分，其输出5V稳压电源给主控供其稳定工作。SD卡部分板载于主控模块上，通过SDIO与MCU通讯，负责记录系统采集到的数据。PC/树莓派是系统log信息的输出对象，可以通过系统的信息输出监控系统状态与采取相应措施，UART是双攻总线，其也可发送紧急控制至主控。

2.3 软件架构

因为实时操作系统即RTOS的运行会使用一定的系统资源(主要是内存资源)，因此只有部分实时操作系统能够在拥有相对较小的RAM的MCU上运行。较μC/OS-II、embOS等商业化RTOS，FreeRTOS是一个开源的操作系统，他的灵活性很强，尤其是可以根据用户的需求对内核做相应的裁剪或者修改，可以以较低成本地移植到不同的单片机上使用。因此本次设计中软件将FreeRTOS作为其系统，但即使是一个轻量级的RTOS，FreeRTOS也包含操作系统最典型的功能，如：任务管理、消息队列、内存管理等，完全可以满足系统使用需要。

此外，课题中的数据存储在本地SD卡，SD卡采用FAT（file allocation table）文件系统，其具有标准化、易于存取、管理便捷、可读性强等优势。虽然其具有诸多的缺陷，例如不支持大于4G的文件、空间利用效率较NTFS较低等，但其作为该场景使用应为最适合的。

其软件架构如图2-3所示。

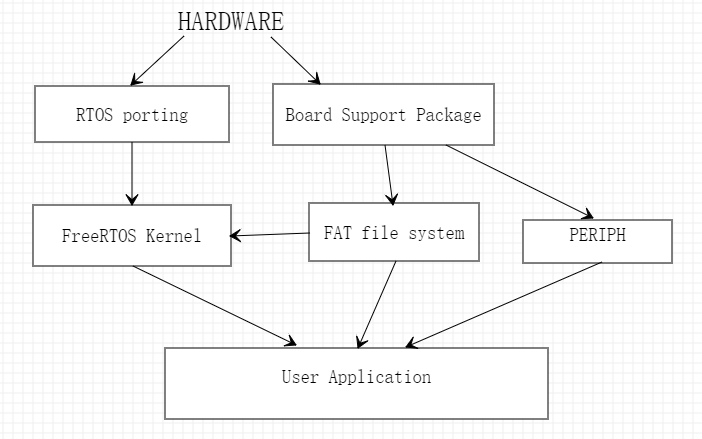


图2-3. 软件架构简图

其中FreeRTOS的porting部分是和硬件强相关的，需要根据实际的硬件平台修改。BSP主要是部分系统资源的初始化和部分API，供外设使用。FreeRTOS的Kernel是一个平台无关的文件集，其和平台相关的部分全部位于porting处。Fat文件系统是一个中间文件，其调用BSP的SDIO部分的API并且使用FreeRTOS的部分系统资源。PERIPH部分是一些其他的外设部分，与系统无关。User Application即为用户进程，通过使用系统资源和外设、fs的API实现项目逻辑。

2.4 本章小结

本章主要介绍了整个项目中的系统架构，以及系统部署的硬件、软件环境。 整个系统的硬件几乎都是模块化总线接口，十分便于后续的迭代优化。在开发的过程中模块化的硬件也可以使得硬件测试更加的标准化、泛用化。软件结构层次化，模块相互之间的引用关系十分明确，便于调试、定位，并且实现了软硬件的分离，后续如果运行逻辑有需要修改的部分直接修改User Application即可，若硬件有修改则直接对硬件代码部分修改即可，只需确保API不变，底层实现逻辑是可以肆意更改的。

第3章 硬件设计

3.1 主控芯片STM32F407

STM32F407适用于在较小面积的封装内实现高较高的集成度，并且拥有不俗的性能的嵌入式情形中。并且STM32F407的内核是一个168 MHz的Cortex™-M4强大内核。

在主频为168 MHz的情形下，当程序于Flash开始执行时，STM32F407系列MCU能够提供210 DMIPS与566 CoreMark，并且利用ST的ART加速器也实现了FLASH零等待状态。此外，STM32F407还具有方便的DSP指令和FPU。

该系列MCU使用了ST的90 nm工艺和ART加速器，可以动态的切换MCU状态以实现功耗的调整，最低能够实现以238 µA/MHz的低电流工作（在 168 MHz时）。

与STM32F4x5系列MCU对比，STM32F407系列MCU还具有符合IEEE 1588 v2标准要求的以太网MAC10/100和能够连接CMOS照相机传感器的8~14位并行照相机接口。

其包含：

* 2个USB OTG（其中一个支持HS）
* 音频：特用音频锁相环和2个全双工的I²S
* 包括6个速度高达11.25 Mb/s的USART、3个45 Mb/s的高速SPI、3个I²C、2个CAN和1个SDIO，共15个通用的总线接口
* 模拟部分：包含2个12位DAC、3个速度为2.4 MSPS或7.2 MSPS（交错模式）的12位ADC
* 17个定时器：频率最高为168 MHz的16和32位定时器
* 可以使用支持SRAM、PSRAM、Compact Flash、NOR和NAND存储器的灵活静态存储器控制器简单的扩展存储容量
* 基于模拟电子技术的真随机数发生器
* STM32F417还整合了加密处理器，为AES 128、192、256、3DES 、MD5和SHA-1等算法实现了硬件加速。

此外，其还额外集成了512KB~1MB的Flash与192KB的SRAM。

3.1.1 STM32F407的SD卡接口描述

其提供了SDIO主机接口，支持 Multi Media Card 系统。有着三种不同的数据总线模式，分别为：1 位（默认）、4 位和 8 位，该接口允许数据流以高达 48 MHz 的频率流式传输，并且与 SD 存储器兼容。SDIO 卡的2.0规范版本允许使用1 位（默认）或 4 位的数据总线模式。当前版本在任何时候只支持一张SD卡和一个堆栈。

其高速模式下的时序图如图3-1所示。

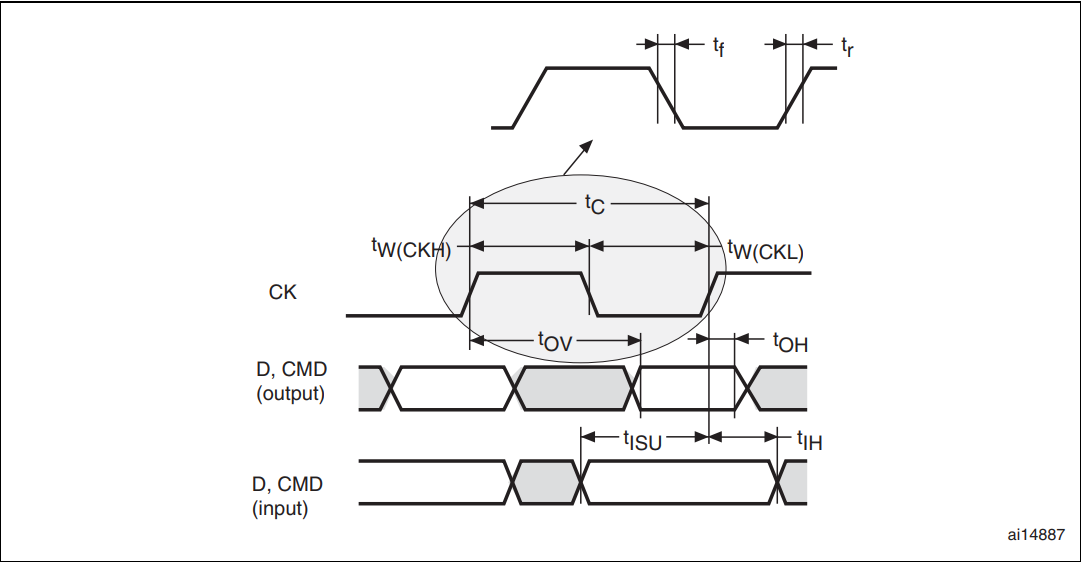


图3-1. SDIO 高速模式

3.1.2 STM32F407的串口描述

STM32具有十分丰富的串口资源，其功能也十分强劲，本次使用的IC共有6个串口，但由于IO pin mux，只可使用其中的5个。其中USART1、USART2、USART3 和 USART6为同步/异步收发器，UART4 和 UART5为同步收发器。均具有单线半双工通讯模式以及LIN主/从功能，通常的UART可以以5.25Mbit/s的速度通讯，其中USART1和USART6可以达到2倍于其他接口的通讯速度。除此之外USART1、USART2、USART3与USART6还提供了可以使用得CTS和RTS硬件管理信号、符合ISO7816标准的智能卡模式与类似SPI的通讯。并且所有的串口总线均可提供DMA服务。

3.2 HMI屏幕

HMI（Human Machine Interface）屏幕具有很好的复用性，也有着较为完备的一套开发环境。本次采用的为淘晶驰的TJC1060X5A1\_011CS\_I屏幕，其分辨率为1024\*600，有效显示尺寸为222.72\*125.28（mm），具有较小的工作电流与休眠电流、较宽的电压输入，flash容量、RAM、RTC功能均十分完备，还具有SD卡拓展。其官方的开发环境如图3所示。

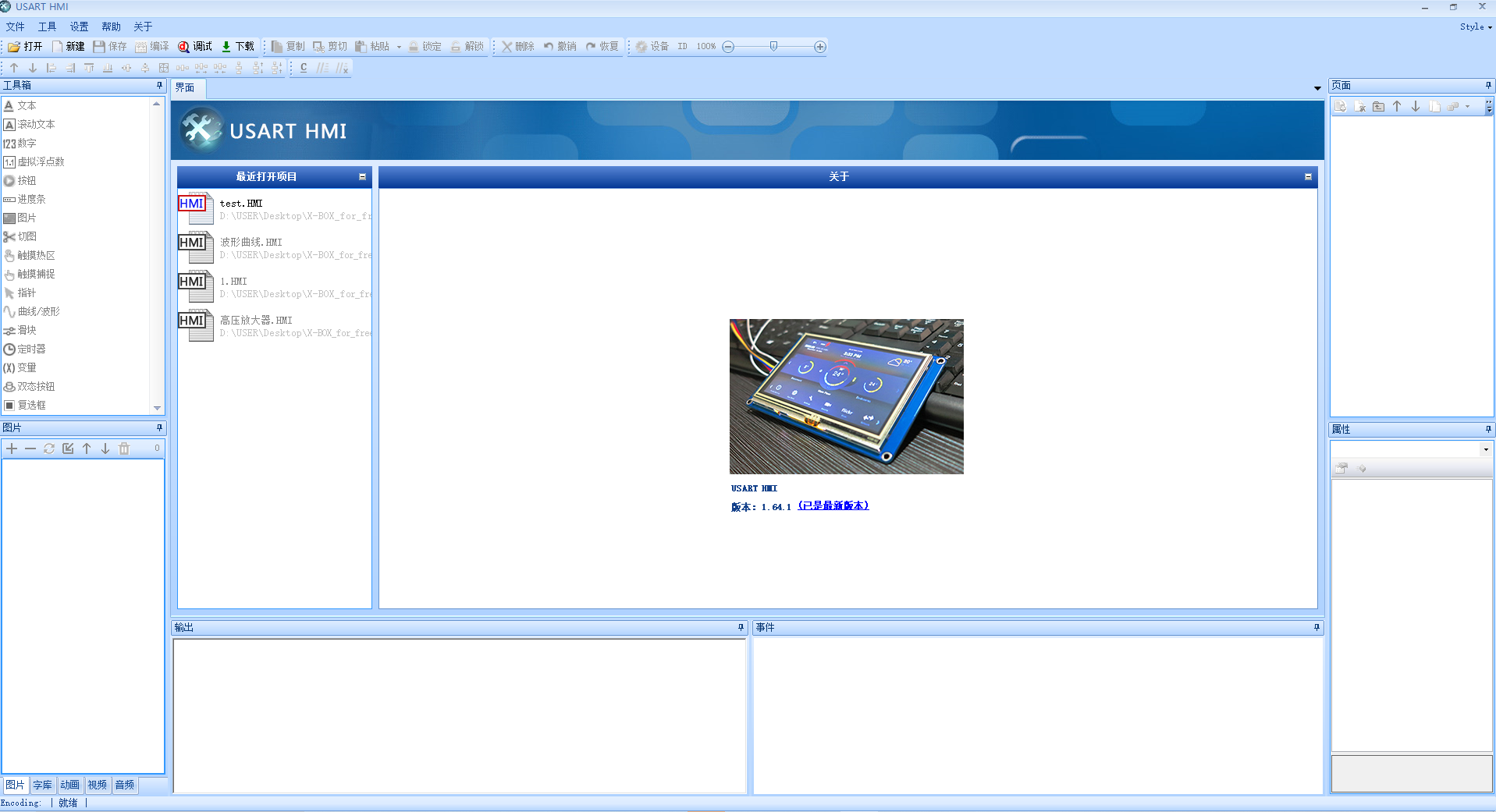


图3.HMI屏幕开发环境

3.3 采集模块

3.3.1 初版设计

初版设计中，采集模块由AD7190与外围电路完成。AD7190为一款ADI的适合高精度测量应用的低噪声模拟前端，其内部集成了低噪声、24位的Σ-Δ型ADC。该设备可配置为具有两个差分输入或四个伪差分输入。片上通道定序器允许启用多个通道，并且AD7190在每个启用的通道上按顺序转换。这简化了与其他器件的沟通。其存在一个4.92MHz的内置时钟信号源，也可使用外部晶振。其输出速率可达4.8kHz。该IC还提供2种数字滤波器，具有零延迟特性。其采用5V模拟电源与2.7V~5.25V的数字电源供电，功耗可达6mA。其功能框图如图3-2所示。

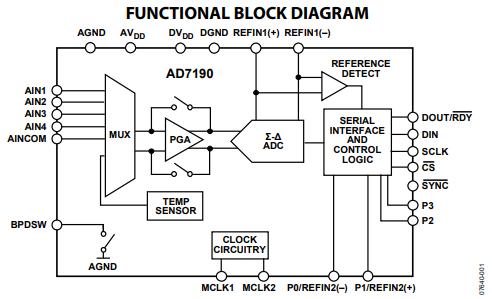


图3-2.AD7190功能框图

AD7190模块与MCU之间采用SPI作为通讯方式，SPI是一种全双工的同步串行总线（Serial Peripheral Interface）多用于Flash存储器（如NOR Flash&Nand Flash），ADC、LCD控制器等外围器件的通讯接口。大大增强了处理器的外设扩展能力。除开电源外，一般使用4根信号线，分别为：SSEL：slave select，常常也被写作CS（chip select）或SS（slave select），负责SPI设备的片选信号；SCK：serial clock，常常也写作SCLK或SCL，提供设备的时钟信号；MISO：master input slave output，常常被简写为SO（slave output），从机输出接口；MOSI：master output slave input，常常被简写为SI（slave input）从机输入接口。图3-3为一个典型的SPI设备接法。

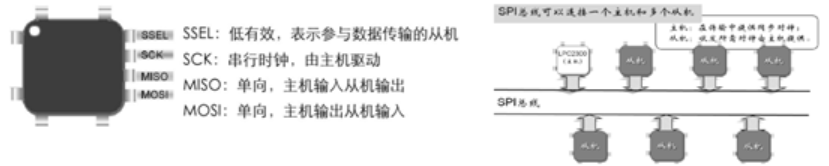


图3-3.典型SPI设备接法

在SPI总线上，产生时钟信号的器件称为主机。主机和从机之间传输的数据与主机产生的时钟同步。与I2C接口相比，SPI器件支持更高的时钟频率。SPI接口只能有一个主机，但可以有一个或多个从机。来自主机的片选信号用于选择从机。这通常是一个低电平有效信号，拉高时从机与SPI总线断开连接。当使用多个从机时，主机需要为每个从机提供单独的片选信号。MOSI和MISO这两根为SPI的数据通路，数据从主机输出至从机，通过MOSI，从从机返回至主机，通过MISO。当一个主机和多个从机进行通讯时，通过CS来选择和那个设备进行通讯，可以将CS理解为enable信号，低电平有效。SPI总线上存在多个从设备时，SPI的从机设备的IO口需要有三态特性，当该从设备没有被片选使能时其IO口表现为高阻抗。当前多数SPI设备在不做通讯时，默认的状态通常就是高阻抗状态。

在SPI中，主机可以选择时钟极性和时钟相位。时钟极性CPOL（Clock Polarity）：是用来配置SCLK的电平处于哪种状态时有效。CPOL=0：表示高电平有效，低电平处于空闲态。CPOL=1：表示低电平有效，高电平处于空闲态。时钟相位CPHA（Clock Phase）：是用来配置数据采样是在第几个边沿，0表示第一个边沿（前沿Leading edge），1表示第二个边沿（后沿Trailing edge）。CPHA=0：表示数据采样是在第1个边沿，数据发送在第2个边沿。CPHA=1：表示数据采样是在第2个边沿，数据发送在第1个边沿。主机必须根据从机的要求选择时钟极性和时钟相位，即从机的传输方式决定了主机的传输方式。因此需要先得到从机的工作模式，并将主机的模式设置至与从机一样。根据CPOL和CPHA位的选择，有四种SPI模式可用。

当需要指定从机进行SPI通讯时，主机需先提供时钟信号，再片选从机。片选的使能通常是低电平有效，因此主机需要将片选脚的电平设置为逻辑0。

SPI是全双工接口，主机和从机可以分别通过MOSI和MISO线路同时发送数据。在SPI通信期间，数据的发送（串行移出到MOSI/SDO总线上）和接收（采样或读取MISO/SDI总线上的数据）同时进行。串行时钟沿同步数据的移位和采样。

SPI协议规定一个SPI设备不能在数据通信过程中仅仅只充当一个"发送者(Transmitter)" 或者"接收者(Receiver)"。在每个时钟周期内，SPI的环形总线上会有一个bit被发送与接受，即主从设备会有一个比特的数据被交换。SPI功能框图如图3-4所示。

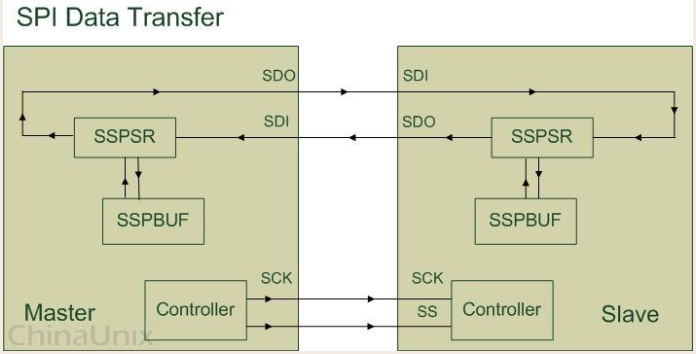


图3-4. SPI功能框图

在每个时钟周期内，Master与Slave之间交换的数据其实都是SPI移位寄存器从SSPBUF里面拷贝的。我们可以通过往SSPBUF对应的寄存器 (Tx-Data/Rx-Data register) 里读写数据，间接地操控SPI设备内部的SSPBUF。

例如,在发送数据之前,我们应该先往Master的Tx-Data寄存器写入将要发送出去的数据，这些数据会被Master-SSPSR移位寄存器根据Bus-Width自动移入Master-SSPBUF里, 然后这些数据又会被Master-SSPSR根据Channel-Width从Master-SSPBUF中移出,通过Master-SDO管脚传给Slave-SDI管脚,Slave-SSPSR则把从Slave-SDI接收到的数据移入Slave-SSPBUF里。与此同时,Slave-SSPBUF里面的数据根据每次接收数据的大小(Channel-Width), 通过Slave-SDO发往Master-SDI,Master-SSPSR再把从Master-SDI接收的数据移入 Master-SSPBUF。在单次数据传输完成之后,用户程序可以通过从Master设备的Rx-Data寄存器读取Master设备数据交换得到的数据。

3.3.2 初版测试

测试平台为正点原子探索者开发板，核心文件为HARDWARE/drv\_spi.c、HARDWARE/drv\_spi.h、PERIPH/bsp\_ad7190.c、PERIPH/bsp\_ad7190.h。

其中drv\_spi系列文件为MCU硬件SPI初始化相关文件。在drv\_spi.h头文件中定义了相关宏、宏函数与函数声明。因使用硬件SPI，因此相关宏\_\_USE\_SOFT\_SPI\_INTERFACE\_\_不需配置。在drv\_spi.c文件中实现了如下函数：

* void drv\_spi\_init( void )

该函数初始化了STM32F407的硬件SPI，调用drv\_gpio.c中的GPIO\_Format\_Init API初始化由宏定义SPI\_CLK\_GPIO\_PORT、SPI\_CLK\_GPIO\_CLK等定义好的IO口为AF模式，然后调用CMSIS的GPIO\_PinAFConfig API配置IO PIN MUX为SPI1。接着通过调用标准库的GPIO\_SetBits与SPI\_I2S\_DeInit API复位片选信号与SPI，并且根据SPI号使能总线时钟，SPI2与3为APB1总线，其余为APB2总线。最后使用SPI\_Cmd API失能SPI，将其配置为双线全双工、主机模式、空闲状态为低电平、第一个边沿采集数据、8位数据、从机软件管理、64分频、最高位先发送与CRC多项式后配置并使能SPI。

* uint8\_t drv\_spi\_read\_write\_byte( uint8\_t TxByte )

先等待发送缓冲区清空，然后重新设置接收等待时间。因为SPI的速度很快，正常情况下在发送完成之后会立即收到数据，等待时间不需要过长。将发送数据写入其DR寄存器，等待接收缓冲区非空，接着将位移结束的DR寄存器的数据读出并返回。

* void drv\_spi\_read\_write\_string( uint8\_t\* ReadBuffer, uint8\_t\* WriteBuffer, uint16\_t Length )

多次调用drv\_spi\_read\_write\_byte实现收发字符串的目的，更加适应一般的使用场景。

bsp\_ad7190系列文件为AD7190相关文件，头文件中定义了相关宏和函数声明。其c文件中实现了如下函数：

* void AD7190\_Reset(void)

发送7字节指令用于复位AD7190。

* uint32\_t AD7190\_GetRegisterValue(uint8\_t registerAddress,uint8\_t bytesNumber)

读取指定的AD7190的寄存器值。

* uint8\_t AD7190\_Init(void)

初始化SPI、复位AD7190并且测试链接情况打印log。

* void AD7190\_SetRegisterValue(unsigned char registerAddress,unsigned int registerValue,unsigned char bytesNumber)

写入值至指定的AD7190寄存器。

* void AD7190\_ChopSetting(uint8\_t chop)

设置AD7190的chop使能与否。

* AD7190\_ChannelSelect(uint16\_t channel)与AD7190\_MultiChannelSelect(uint16\_t channel)

选择AD7190的通道。

* void AD7190\_WaitRdyGoLow(void)

等待就绪信号，阻塞函数。

* void AD7190\_Calibrate(uint8\_t mode, uint8\_t channel)

执行指定通道的模式校准。

* void AD7190\_Continuous\_ReadData(unsigned char cread)

设置连续读取AD7190的数据使能与否。

* void AD7190\_RangeSetup(unsigned char polarity, unsigned char range)

选择转换极性和ADC输入范围。

* void AD7190\_Unipolar\_Multichannel\_Conf(void)

多路单端输入AD采集配置。

* Get\_ADC\_flag/ Get\_ad7190\_data/ Get\_bias\_data/ Get\_voltage\_data/ Set\_ADC\_flag/ Set\_number/ Set\_ad7190\_data/ Set\_bias\_data/ Set\_voltage\_data

存取私有数据的接口函数。

测试数据部分如图3-5所示，可以看出其线性度不佳，需要较为复杂的拟合。

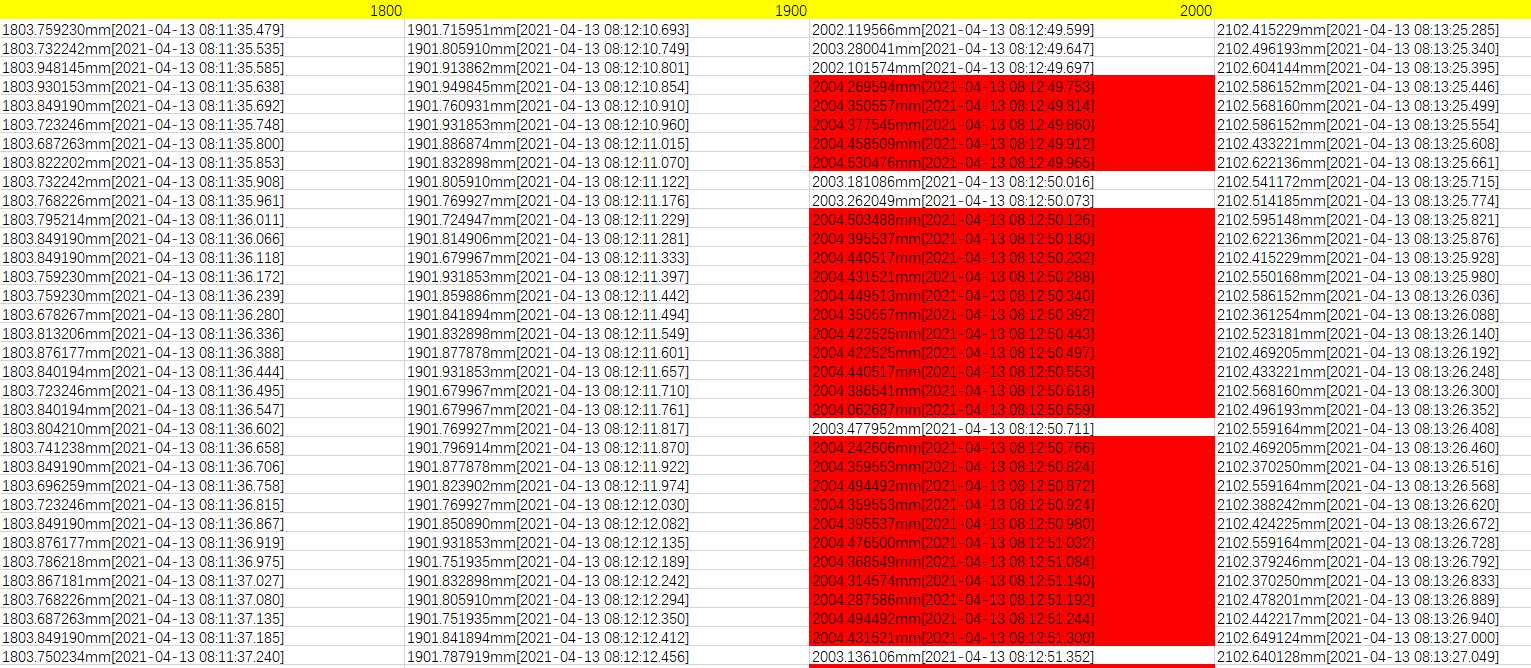


图3-5. 部分测试数据

由于MCU资源有限且拟合所需开发时间较长，并且实际测试分辨率与采集速度与AD7190匹配度也不是十分高，经项目组会议讨论，决定更换采集模块。

3.3.3 第二版设计与测试

第二版选择工业控制上常用的PLC采集模块SP-4017+，它是一款 16bit A/D 8 通道模拟量输入+2 通道数字量（晶体管）输出模块，可以采集电压、电流模拟量输入信号，同时提供 2 路数字量输出。其所有通道可单独配置输入范围，广泛用于工业测量、智能仪器仪表及过程控制等应用，具备极高性价比；RS-485的通道与模块间存在1500V的电气隔离，可以避免输入端口浪涌干扰等其它高压冲击而引起的损坏。SP-4017+支持 8 路真差分信号，使用标准 Modbus RTU 协议，各通道可独立配置其输入范围。

MODBUS 是MODICON公司（现为施耐德电气公司的一个品牌）提出的一种通讯协议，在业内普遍使用后逐渐成为一种标准的通讯协议，只要遵循其通讯的协议，哪怕双方的设备是不同的硬件、不同的系统，两者之间也可通讯。在RS495通讯中，MODBUS协议占比较大。

SP-4017+使用的是MODBUS RTU协议。RTU协议规定了数据帧的格式，这是通讯的前提。该协议为半双工协议，采用的是主从应答式通讯，由主机发出请求后，从机在接收到对应请求后回传应答，完成一次数据传输。

其协议采用查询-回应模式。查询消息中的功能代码告之被选中的从设备要执行何种功能。数据段包含了从设备要执行功能的任何附加信息。错误检测域为从设备提供了一种验证消息内容是否正确的方法。如果本次通讯无异常则从设备返回的应答中包含请求的响应信息。数据段包括了从设备收集的数据：如寄存器值或状态。倘若通讯过程发声了错误，则功能码会支出错误的信息，用于主设备检测通讯链路存在的问题。错误检测域允许主设备确认消息内容是否可用。

其寄存器定义如图3-6所示。





图3-6. SP-4017+寄存器定义

其双极性信号的计算公式为：

单极性信号 4-20mA 计算公式为：

经其官方软件调试测试，如图3-7所示，均满足使用需求。

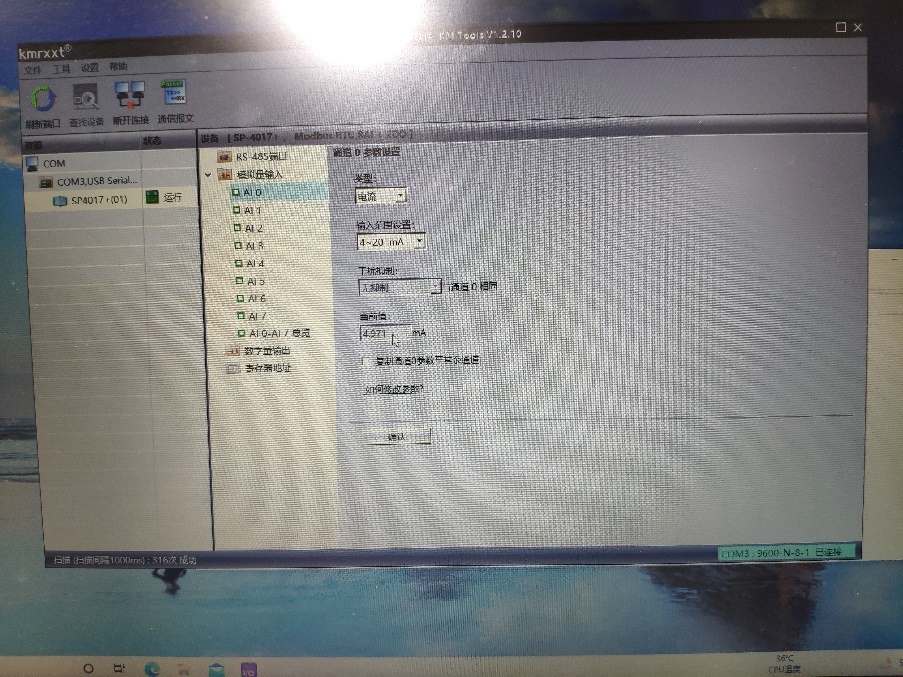


图3-7. 调试过程

3.4 PCB设计

3.4.1 硬件架构与原理图设计

MCU为STM32F407VET6（100 pin），电源管理部分为5V输入，LDO降压稳压。外设部分包含LED、KEY、BEEP、SD卡、CAN和UART等。整个测试系统的电源为标称12V，因此还有一块基于LM2596的开关电源降压板。硬件部分采用Altium Designer17进行绘制。

* POWER

主控的电源输入为一个XH2.54-2p母座，这种接口可以在确保电源防反的同时有较好的锁紧作用。MCU所需标称输入为3.3V，该主控降压使用LDO。

LDO即low drop out regulator，是一种低压差线性稳压器。这是相对于传统的线性稳压器来说的。较为传统的线性稳压要求输入和输出之间存在一个较大的压差，至少需要3V左右，否则无法正常工作。但对于弱电嵌入式使用场景，这个条件有些苛刻，经常在使用中，压差仅为1V左右甚至零点几伏。针对这种情况，芯片制造商们才研发出了LDO类的电压转换芯片。

低压降（LDO）线性稳压器的优点有很多，包括但不限于成本、噪声以及电路简单等。其正常仅需部分旁路电容就可满足使用，最多为防止输入波动增加一个稳流电感。

具AMS1117的datasheet的推荐容感值，L1为10uH，C2为22uF。除此之外，还设计了2个LED用于指示电源输入状况，其仅当红绿2个LED均正常点亮时表示电源输入正常。具体电路设计如图3-8所示。

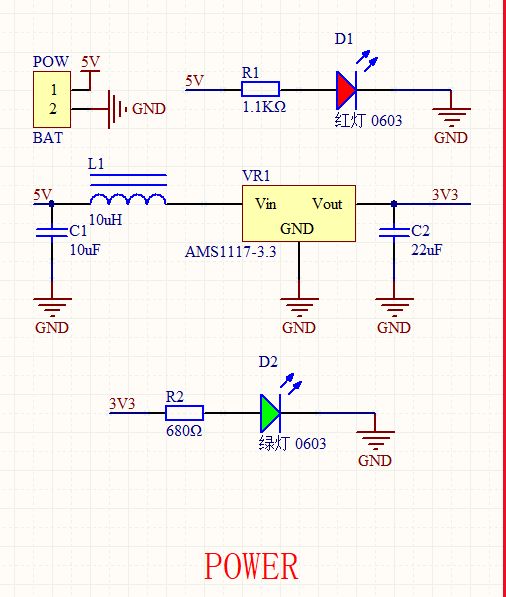


图3-8. POWER模块

* KEY

板载了2个mini按键，按键采用上拉设计，因为板载按键基本只是供测试使用，因此并不需要诸如电容按键、压力按键等复杂设计，具体电路如图3-9所示。

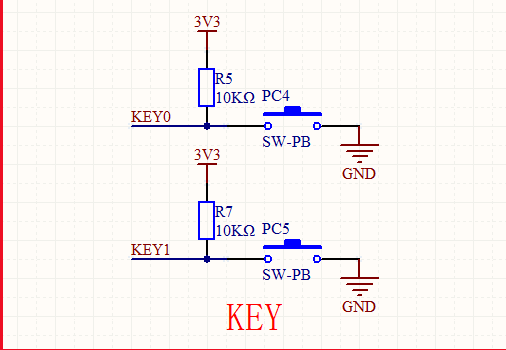


图3-9. KEY模块

* BEEP

板载一个有源蜂鸣器。与无源蜂鸣器相比，电磁感应现象是通过吸引或排斥音圈与交流电连接后形成的电磁铁和永磁体来促进振膜发出声音的。倘若一直是直流电，那么膜片只会一直被推动而不产生声音，只有在连接或断开电流时才会产生声音。有源蜂鸣器是一种一体化结构的电子讯响器，仅需提供直流电就可以发声，使用较为简单，因此本次使用一个三极管去控制其通断，具体电路如图3-10所示。

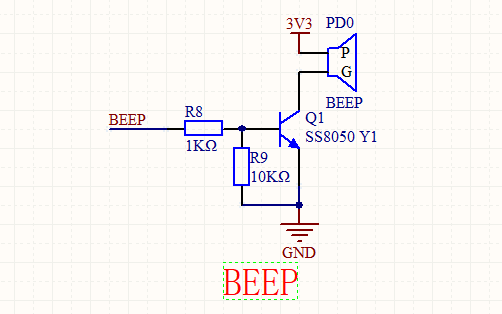


图3-10. BEEP模块

* LED

LED设计较为简单，板载了4个颜色不同的上拉式LED，因为LED和按键相比所需电流较大，所以上拉是较为理想的实现方式，LED所需电流仅为几毫安，因此使用1个4位1kΩ的排阻用作限流。具体电路设计如图3-11所示。

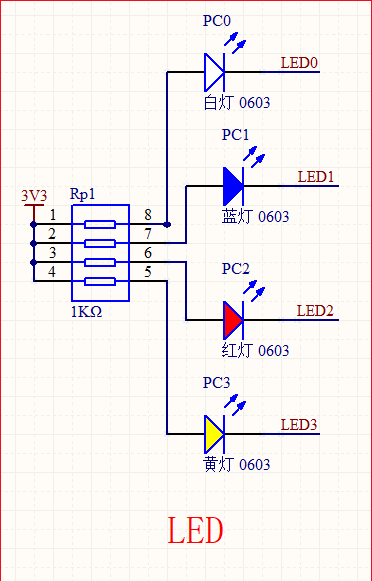


图3-11. LED模块

* CAN

CAN 是Controller Area Network 的缩写（以下称为CAN），是ISO国际标准化的串行通信协议，主要用于汽车行业。汽车行业一般会使用很多不同的系统，每个系统间的通讯方式均不同，因此需要很多的通讯线束。为了减少线束，统一通讯，德国博世公司于1986 年提出了面向汽车的CAN 通信协议。此后，CAN 通过ISO11898 及ISO11519 进行了标准化，现在已是欧洲汽车网络的标准协议。

CAN 控制器根据两根线上的电位差来判断总线电平。总线电平分为显性电平和隐性电平，二者必居其一。

现在的CAN控制器一般都与MCU集成在一起，其发送和接收TTL信号就是MCU引脚(高或低)信号。以前还有独立的CAN控制器，一个CAN网络节点会包含三块芯片：MCU芯片、CAN控制器、CAN收发器。现在是将前两者集成在一起了，STM32F4系列MCU的CAN通讯功能并不包含收发器部分，CAN收发器就是一块类似232或485的转换芯片，它的主要功能是将CAN控制器的TTL信号转换成CAN总线的差分信号。因此在电路设计上需要额外加上一块CAN收发器芯片TJA1050。

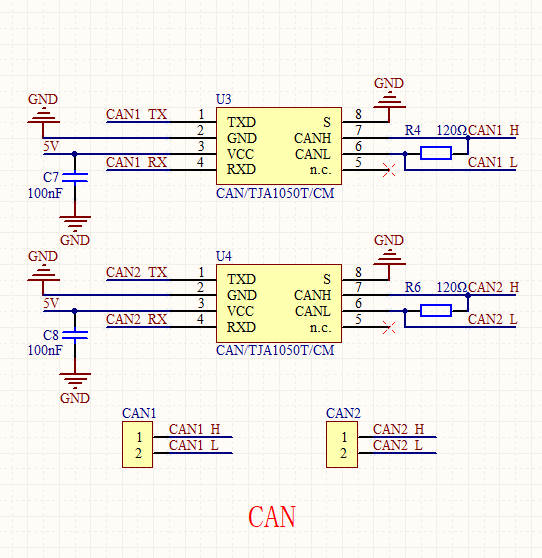


图3-12. CAN模块

* SD\_CARD

因为STM32F4系列MCU自带硬件SDIO，因此电路设计较为简单，仅需加上上拉电阻后连接至对应IO port即可，电源部分使用100nF的电容滤波。具体设计如图3-13所示。

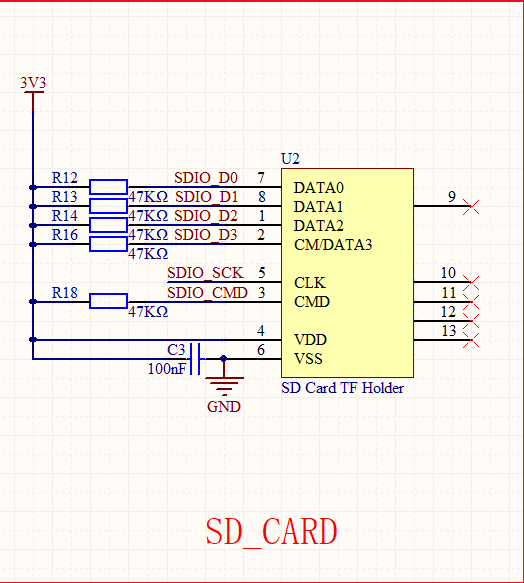


图3-13. SD\_CARD模块

* UART与SWD

UART的硬件接口电路比较简单，其大致原理如图3-14所示。

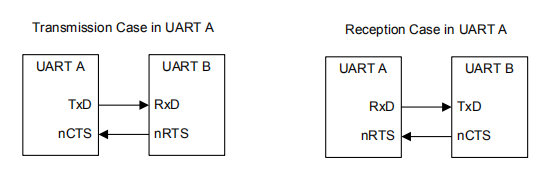


图3-14. 串口工作原理

SWD部分为MCU的调试、下载接口，ARM常用的调试器有JTAG、JLINK和ST-LINK这几种。ARM体系原先是支持JTAG硬件调试接口的，随着cortex系列增加SWD调试规范，现在越来越多的调试器选择SWD接口。其接线少，速率高，也比较稳定。JTAG 20个口太占用pin资源了。因此本次设计也使用了SWD模式。

SWD仅需4根信号线就可以工作了，分别如下：

Ref：目标板参考电压信号。用于检查目标板是否供电，直接与目标板VDD联，并不向外输出电压；

GND：公共地信号；

SWDIO：串行数据输入输出，作为仿真信号的双向数据信号线，建议上拉；

SWCLK：串行时钟输入，作为仿真信号的时钟信号线，建议下拉；

SWO：串行数据输出引脚，CPU调试接口可通过SWO引脚输出一些调试信息。该引脚是可选的；

RESET：仿真器输出至目标CPU的系统复位信号。该引脚也可是可选的。

具体设计如图3-15所示。

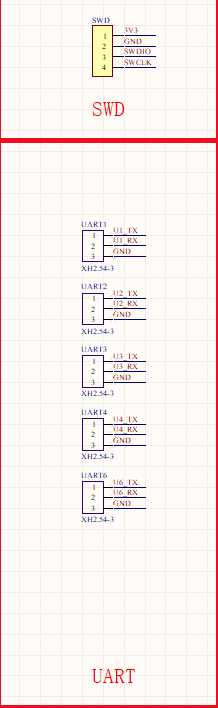


图3-15. UART与SWD模块

* MCU相关

STM32在上电是会根据BOOT引脚的电平去选择启动模式，共有三种。第一种模式boot0= 0，boot1=0或1（默认0），这种模式下，STM 32处于正常的读取芯片内部存储空间的程序即运行正常程序。第二种模式，boot0=1，boot1=0时，进入调试模式，可以通过烧录工具把修改好的代码所生成的HEX文件烧录到芯片里面，然后把boot0跳线跳回0，执行代码。第三种模式，boot0=1，boot1=1，多数情况下，因为SRAM掉电丢失数据的特性，这种模式一般只是在调试时使用，写一段小的，可能会有问题的测试代码运行，这样即便代码存在问题，也不会影响下一次使用，同时这种方法也可以消除内部flash的保护，但在接触保护的同时，flash的数据也会被刷写。

STM32F4系列MCU有一个RESET引脚用于复位系统进入reset中断，该引脚低电平有效，因此在电路实现上需要使用上拉，并且由于该引脚较为重要，需要额外引入一个100nF的电容进行滤波、消抖，防止因电源波动导致系统重启。

STM32F4系列MCU存在内部的RC振荡器。LSI为低速内部时钟，为32kHz，但只供独立看门狗和自动唤醒单元使用。HIS为内部高速时钟，为16MHz，可以作为系统时钟或者用作PLL的输入，但内部RC振荡器由于电阻电容的精度问题所以RC振荡器的震荡频率会有误差，同时受到温度、湿度的影响，这个跟元器件的工艺有关。因此外接一个晶振，其优点是相对来说振荡频率一般都比较稳定。缺点的话就是价格要稍微高点了，还有用晶体振荡器一般还需要接两个 15-33pF起振电容。

一般MCU中很少用RC振荡器，可能在实验室环境会用，而在实际的工程、工业上很少用到，常用的也就是晶体振荡器。因为很多时候单片机需要一个精度的机器周期作定时、通讯等用途，如果震荡频率不准的话对产品的功能是有很大影响的。

本次电路设计中采用的为8MHz的晶振，起振电路中的电容为20pF。STM32F4的时钟树如图3-16所示。

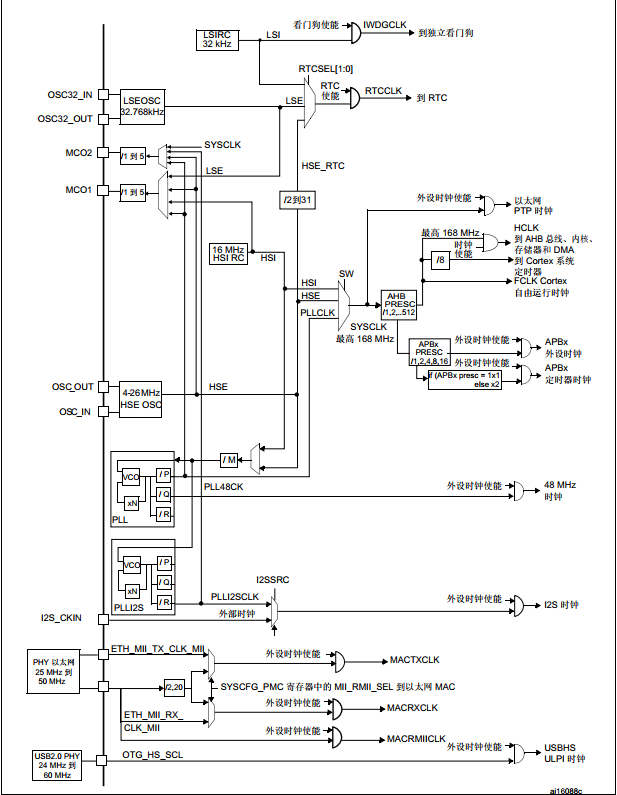


图3-16. STM32F4的时钟树

根据时钟树，输入晶振经过PLL后会输出48MHz的外设时钟以及最高168Mhz的系统时钟。在CMSIS的system\_stm32f4xx.c文件中，规定了其PLL中的各种分频系数，因为选用外部晶振为8MHz，因此PLL\_M为8，PLL\_N为336，PLL\_P为2，PLL\_Q为7，其外设时钟为输入晶振经PLL\_M分频后通过与PLL\_N倍频再经PLL\_Q分频得到48NHz，其系统时钟为输入晶振经PLL\_M分频后通过与PLL\_N倍频再经PLL\_P分频得到168NHz。

该部分具体电路设计如图3-17所示。

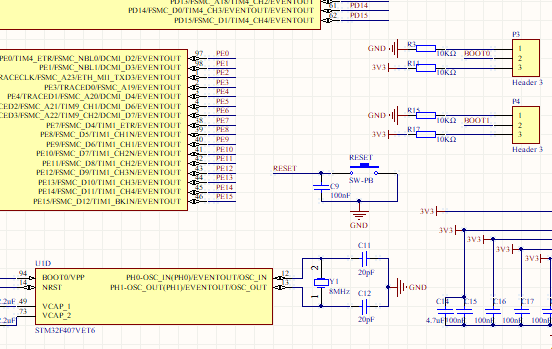


图3-17. MCU相关部分电路

* 电源管理

电源管理部分核心使用LM2596作为其降压芯片，LM2596是降压型电源管理单片集成电路的开关电压调节器，能够输出3A的驱动电流，同时具有很好的线性和负载调节特性。固定输出版本有3.3V、5V、12V，可调版本可以输出小于37V的各种电压。本次使用的是固定输出版，经理论计算与实际测试，最终敲定输出端接68uH的电感与220uF的铝电解电容的降压效果较好，输出波形毛刺较少，振铃不明显，具体的电路设计如图3-18所示。

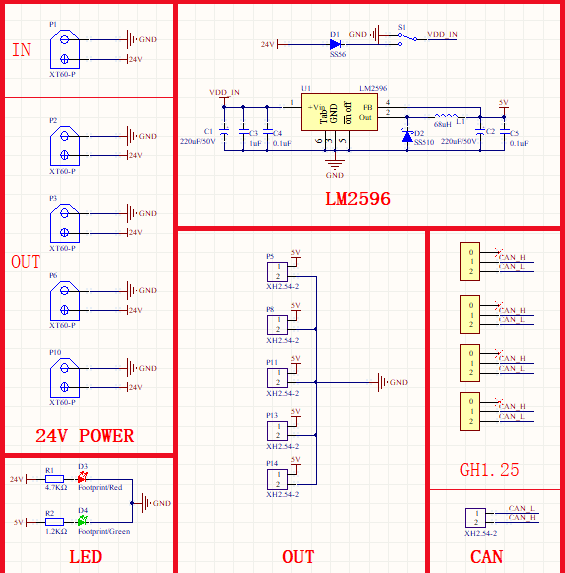


图3-18. 电源管理原理图设计

3.4.2 PCB layout

本次的PCB layout综合考虑使用场景、开发周期和量产成本等因素，使用双层板layout。在PCBlayout中，电源网络的走线是比较重要的，在layout过程中PCB走线和电流的关系一般是如表3-19所示。

表3-19. PCB设计铜箔厚度、线宽和电流关系表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **铜厚/35um** | | **铜厚/50um** | | **铜厚/70um** | |
| 电流(A) | 线宽(mm) | 电流(A) | 线宽(mm) | 电流(A) | 线宽(mm) |
| 4.5 | 2.5 | 5.1 | 2.5 | 6 | 2.5 |
| 4 | 2 | 4.3 | 2.5 | 5.1 | 2 |
| 3.2 | 1.5 | 3.5 | 1.5 | 4.2 | 1.5 |
| 2.7 | 1.2 | 3 | 1.2 | 3.6 | 1.2 |
| 3.2 | 1 | 2.6 | 1 | 2.3 | 1 |
| 2 | 0.8 | 2.4 | 0.8 | 2.8 | 0.8 |
| 1.6 | 0.6 | 1.9 | 0.6 | 2.3 | 0.6 |
| 1.35 | 0.5 | 1.7 | 0.5 | 2 | 0.5 |
| 1.1 | 0.4 | 1.35 | 0.4 | 1.7 | 0.4 |
| 0.8 | 0.3 | 1.1 | 0.3 | 1.3 | 0.3 |
| 0.55 | 0.2 | 0.7 | 0.2 | 0.9 | 0.2 |
| 0.2 | 0.15 | 0.5 | 0.15 | 0.7 | 0.15 |

据经验估算，整体主控板的功率不超过9W，因此在电源输入端的网络走线宽度选择40mil，并且在LDO附近有做过孔散热处理。UART与CAN等接口使用XH接线端子，该端子防反接且带有自锁的功能，在PCB设计中较为常用，SWD端子选择PH接线端子，防止其与通讯接口等端子弄混。为防止控制芯片因附近电流回流干扰，在芯片附近均采用了Polygon Pour Cutout。

该PCB设计力求最大化单面空间利用率，板框为70mm\*100mm，因此大部分阻容件均采用0603的封装格式。电源管理的PCB考虑其可能会有大电流经过，因此出了基础的线宽加大外，还在solder层做了开窗处理，可以容纳更大的电流。

因为晶振由石英晶体构成，容易受外力撞击或跌落的影响，所以在布局时，最好不要放在PCB边缘，尽量靠近芯片摆放，并且晶振的摆放需要远离热源，因为高温也会影响晶振频偏。OSC\_IN和OSC\_OUT的走线应为差分走线，避免其精准度出现问题影响系统。

考虑电源可能会存在波动，因此，有必要在印刷板上增加必要的去耦电容器，以滤除电源上的干扰信号，从而使电源信号稳定。

除此之外在设计过程中还有较多的小细节，比如说走线时应避免直角和锐角走线，防止其出现容性负载、器件和装配孔之间间距需留有装配空间的间距、地平面铺铜时注意死铜的移除与确保地平面回流等。主控板具体的top、bottom layout如图3-20与3-21所示，仿真如图3-22所示。电源管理板的layout如图3-23所示。

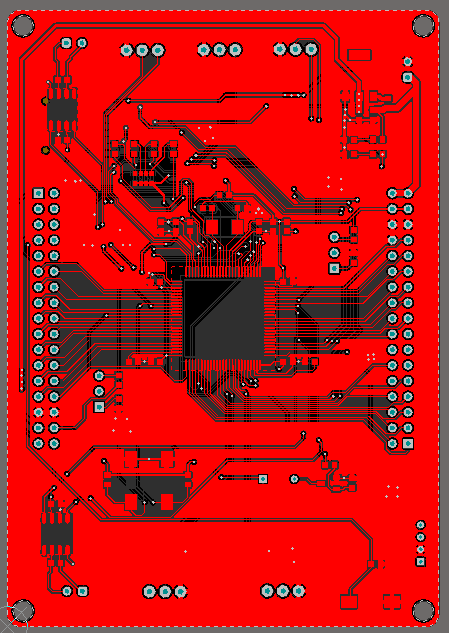


图3-20. 主控顶层layout

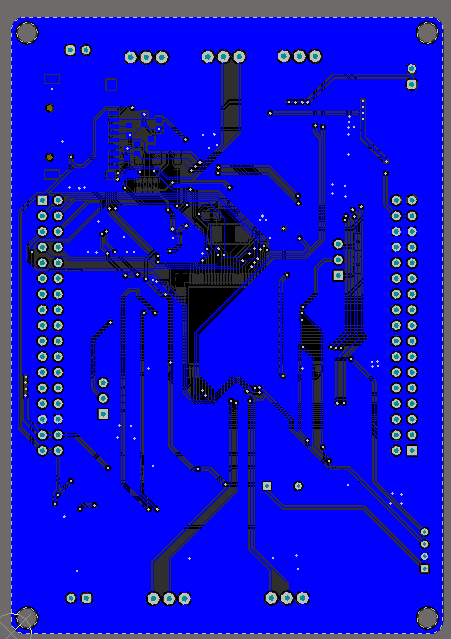


图3-21. 主控底层 layout

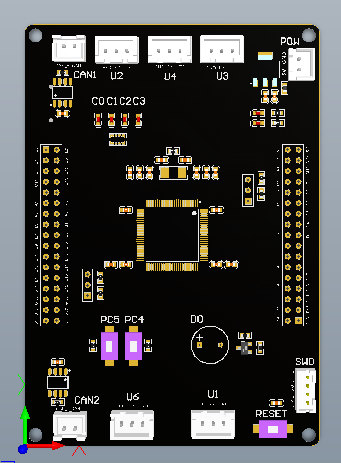


图3-22. 主控PCB仿真图

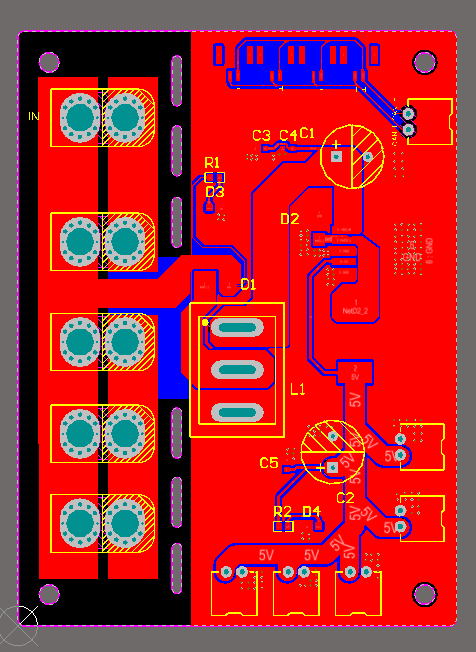


图3-23. 电源管理layout

第4章 软件设计

4.1 FreeRTOS

4.1.1 FreeRTOS介绍

通常一个计算机系统可以大致分为四个部分：硬件（Hardware），操作系统（operating system），应用程序（application programs），使用者（user）。硬件包含了CPU，存储空间，输入输出设备等为整个系统提供了基础的计算资源。操作系统是一个控制程序，作为硬件和应用程序之间的桥梁，主要是和硬件打交道，负责协调分配计算资源和内存资源给不同的应用程序使用，并防止系统出现故障。面对来自不同应用程序的大量且互相竞争的资源请求，操作系统通过一个调度算法和内存管理算法尽可能把资源公平，有效率地分配给不同的程序。应用程序则通过调用操作系统提供的API接口获得相应资源完成指定的任务。

一个通用型的操作系统一般具有调度管理、堆栈和内存管理、文件管理、队列管理、中断和定时器管理、资源管理和输入输出管理等功能。并且操作系统自身需要占用一定的计算资源和内存资源，对于片上资源有限的单片机，可能需要根据需求剪裁操作系统的大小和功能。

而实时操作系统（RTOS-Real Time Operating System）中实时（Real Time）指的是任务（Task）或者说实现一个功能的线程（Thread）必须在给定的时间(Deadline)内完成。业界总有种误解认为如果能堆砌更多的处理器核心数目，更高的处理器频率，更大的内存，更快的总线速度系统就能一定达到实时性的表现。但是强大的计算能力并不能保证系统的实时性。举一个简单例子比如汽车中的安全气囊，在传感器检测到汽车发生碰撞后，安全气囊需要在30ms内完全打开，不然司机和乘客的人身安全将受到极大的威胁。即使车载ECU有很强大的计算能力，但是如果因为要执行其他复杂计算任务或者任务调度的问题导致对汽车状态的监测和安全气囊的响应时间超过规定的时间，实时性无法得到保障导致系统失效和人员伤亡，这将会是非常严重的系统问题。

FreeRTOS允许将软件应用程序编写为一组独立的任务。每个任务都有一个优先级，操作系统有责任确保能够运行的优先级最高的任务就是正在运行的任务。其具有如下的基本概念：

* 多任务

每个正在执行的程序都是操作系统控制下的一个任务（或线程）。如果一个操作系统可以以这种方式执行多个任务，则称之为多任务系统。

* 调度机制

调度器是内核的一部分，负责决定在任何特定时间执行哪个任务。内核可以在任务生命周期内多次挂起和恢复任务。

* 上下文切换

当任务执行时，它使用处理器/微控制器寄存器，并像其他任何程序一样访问RAM和ROM。这些资源（处理器寄存器、堆栈等）一起构成了任务执行上下文。保存被暂停任务的上下文并恢复被恢复任务的上下文的过程称为上下文切换。

4.1.2 FreeRTOS porting

FreeRTOS的源码是包含2部分的，一部分是硬件平台无关的头文件与c文件，适用于各种的编译器与处理器，是通用的，无需更改。另一部分存放于portable文件夹中，是编译器相关的文件夹，不同的编译器使用不同的支持文件，里面包含了各种处理器相关的文件夹，porting过程主要就是处理其中的c文件与头文件。主要为port.c、portmarco.h和内存管理相关的heapx.c文件。

4.1.3 FreeRTOS 启动流程

前半部分为ARM标准启动流程，在s文件初始化MCU堆栈、设置初始的栈指针SP、并将PC设置为Reset\_Handler、设置中断向量表入口地址并初始化向量表。Cortex内核具有强大的异常响应系统，它把能够打断当前代码执行流程的事件分为异常(exception)和中断(interrupt)，并把它们用一个表管理起来，编号为0~15的称为内核异常，而16以上的则称为外部中断，这个表就称为中断向量表。在Reset\_Handler中调用SystemInit，SystemInit在库文件system\_stm32f4xx.c定义，将系统时钟配置为168MHz，最终跳转至C文件的main函数。

main函数中先做板级硬件初始化，第一步要配置STM32的NVIC， NVIC即嵌套向量中断控制器(Nested Vectored Interrupt Controller)。STM32的中有一个强大而方便的NVIC，它是属于Cortex内核的器件，不可屏蔽中断 (NMI)和外部中断都由它来处理，而SYSTICK不是由 NVIC来控制的。STM32(Cortex-M4)中有两个优先级的概念–抢占式优先级和响应优先级，每个中断源都需要被指定这两种优先级。

具有高抢占式优先级的中断可以在具有低抢占式优先级的中断处理过程中被响应，即中断嵌套，或者说高抢占式优先级的中断可以嵌套在低抢占式优先级的中断中。

当两个中断源的抢占式优先级相同时，这两个中断将没有嵌套关系，当一个中断到来后，如果正在处理另一个中断，这个后到来的中断就要等到前一个中断处理完之后才能被处理。如果两个中断同时到达，则中断控制器根据它们的响应优先级来决定要首先处理哪些中断。如果它们的抢占和响应优先级都相等，则它们在中断表中排名的顺序决定了首先处理哪一个。

在BSP\_Init函数中，先将中断优先级分组配置为4，即所有4位用于指定抢占式优先级。接着依次调用各个硬件的初始化函数，在结束时打印log。

紧接着打印log表示RTOS初始化开始，先创建任务入口函数，FreeRTOS中所有的初始任务均是由这一个task创建，并且该task在创建完成后会自己删除。

创建完task后起动调度器，通过调用task.c文件中的vTaskStartScheduler函数，该函数内部会创建idle task，FreeRTOS程序在任意时刻，必须至少有一个任务处于运行状态，为了达到这个要求，FreeRTOS使用了Idle任务，这个任务的任务函数就是一个连续性工作的任务，所以他总是可以处于就绪态（在运行态和就绪态之间转换，没有其他状态）。由于Idle任务的优先级是最低的（优先级为0），所以Idle任务不会抢占用户任务的运行。当其他高优先级的任务需要运行时，他们会抢占Idle任务。接着调用port.c文件中的xPortStartScheduler函数，该函数为平台适用型，不同的编译器和硬件平台的实现均有区别。在本次平台中，该函数先设置中断相关参数并启动SYSTICK定时器，SysTick系统定时器是属于内核中的一个外设，内嵌在NVIC中。该定时器是一个24位的向下递减的计数器。在裸机编程中常用做延时函数，在FreeRTOS中SysTick定时器尤为重要，因为它是给FreeRTOS系统提供时钟的。在FreeRTOS中任务的切换即每个任务运行的时间是由SysTick定时器提供的。最后跳转至prvStartFirstTask函数，该函数由汇编实现，主要功能通过“svc 0”是触发SVC中断，进入其中断服务函数。SVC中断服务函数实现于vPortSVCHandler中，SVC属于系统调用，是为了防止用户直接操作硬件而设计的中间层，用户要想操作硬件底层，需要先发起一个SVC，由SVC来操作硬件。中断服务函数中依次声明外部变量 pxCurrentTCB，pxCurrentTCB 是一个在 task.c 中定义的全局指针，用于指向当前正在运行或者即将要运行的任务的任务控制块。加载 pxCurrentTCB 的地址到 r3。加载 pxCurrentTCB 到 r3。加载 pxCurrentTCB 指向的任务控制块到 r0，任务控制块的第一个成员就是栈顶指针，所以此时 r0 等于栈顶指针。以 r0 为基地址，将栈中向上增长的 8 个字的内容加载到 CPU 寄存器 r4~r11，同时 r0 也会跟着自增。将新的栈顶指针 r0 更新到 psp，任务执行的时候使用的堆栈指针是psp。将寄存器 r0清 0。设置 basepri 寄存器的值为 0，即打开所有中断。basepri 是一个中断屏蔽寄存器，大于等于此寄存器值的中断都将被屏蔽。当从 SVC 中断服务退出前，通过向 r14 寄存器最后 4 位按位或上0x0D，使得硬件在退出时使用进程堆栈指针 PSP 完成出栈操作并返回后进入任务模式、返回 Thumb 状态。在 SVC 中断服务里面，使用的是 MSP 堆栈指针，是处在 ARM 状态。当 r14 为 0xFFFFFFFX，执行是中断返回指令，cortext-m3 的做法，X 的 bit0 为 1 表示返回 thumb 状态，bit1 和 bit2 分别表示返回后 sp 用 msp 还是 psp、以及返回到特权模式还是用户模式。异常返回，这个时候出栈使用的是 PSP 指针，自动将栈中的剩下内容加载到 CPU 寄存器： xPSR，PC（任务入口地址），R14，R12，R3，R2，R1，R0（任务的形参）同时 PSP 的值也将更新，即指向任务栈的栈顶。至此，FreeRTOS已经启动。

4.1.4 FreeRTOS 内存管理

对于任务创建、信号量、消息队列等都需要FreeRTOS操作系统中的内存管理。FreeRTOS支持5中动态内存管理方式，分别在文件heap\_1、heap\_2、heap\_3、heap\_4和heap\_5。对于FreeRTOS的内存管理方式都是在一个全局的数组中进行的，因此是动态内存管理实际上是在静态存储区。

本次使用的是heap4管理方式，其采用一种最佳匹配算法，并且可以回收已分配的内存空间。回收到的空间会尝试合并，将相邻的空闲内存块合并成一个更大的空闲块。其实操作也是在申请的数组ucHeap中进行。

heap4的pvPortMalloc函数中对BlockLink\_t结构体成员xBlockSize的最高位赋予了新的意义，即用来表示本结构体所表示的内存块是空闲还是已经被分配给了应用程序。当=1时，表示空闲内存块；当=1时，表示已被分配给应用程序。还有，该内存分配方式中，多加了一个变量xMinimumEverFreeBytesRemaining用来记录经过多次内存释放和分配之后，空闲内存历史最小值。其对相邻空闲块合并的算法的实现是在空闲链表插入中static void prvInsertBlockIntoFreeList( BlockLink\_t \*pxBlockToInsert )，实现其的回收合并，该分配方式可以一定的缓解频繁分配回收所带来的内存碎片的问题。

4.2 FAT文件系统

4.2.1 FAT介绍

FAT(short for File Allocation Table)文件系统是个通用文件系统，兼容主流OS，如Linux、Windows等，基础于简单的技术，是Wimdows的默认文件系统。由于过于简单的结构，FAT存在过度碎片化、文件损坏与文件名、大小限制等问题。

FAT具有以下属性：

* 分区不能超过2TB(WIndows无法将大于32G的光盘格式化为FAT32，但是Mac可以。)，FAT32允许磁盘容量8TB。
* 存储到FAT分区的文件不能超过4GB。
* FAT分区需要经常进行碎片整理以保持性能。(碎片化原因：当文件删除后再写入新文件时，FAT不会将文件整理成完整片段再去写入，因此长期使用会导致文件资料逐渐分散，因此减缓读写速度。)
* 通常不建议大于32G的FAT分区存在，因为这样的空间两不适配FAT过于简单的组织结构。

因此FAT多用于小容量设备，其中操作系统之间的可移植性很重要。文件系统主要可以提供统一的存取接口，层次化管理，提高存取效率并且减轻管理压力。

FAT文件系统在历史上一共有3中不同的格式，分别为FAT12，FAT16和FAT32。其中FAT12是第一台个人计算机——IBM的MS-DOS1.0中所采用的文件系统，主要用于软盘，其最大的限制为容量，FAT12的最大容量为16MB，但因为当时所采用的基本均为软盘，所以并无大碍。FAT16被许多操作系统所支持，其有着很好的兼容性，但最大的限制还是容量，其最大存储分区只能达到2G，并且还有这较为严重的空间浪费，以及因为其为单文件任务系统，所以并不支持长文件名与安全性相关操作。FAT32也有着很好的兼容性，并且其与FAT16相比显著的提高了其空间利用率，并且其分区上限也达到了32G，一般的使用需求已经完全可以满足。

4.2.2 FAT移植

FAT FS其层次结构大致分为3层，底层接口主要由disk I/O以及RTC所组成，这部分为平台相关文件，需要根据硬件平台以及存储介质的不同编写代码。中间层主要实现FAT的读写协议，主要使用其ff.c和ff.h文件提供的API，一般不需修改。最上层的是应用层，用户使用FATFS提供的API，无需关注其内部的结构以及复杂的协议。FATFS的结构如表4-1所示。

4-1. FATfs的系统文件结构

|  |  |  |
| --- | --- | --- |
| **文件名** | **功能** | **说明** |
| ffcont.h | FATFS配置文件 | 根据具体使用需求配置参数 |
| ff.h | FATFS和应用模块共用引入文件 | 无需修改 |
| ff.c | FATFS源码 | 无需修改 |
| diskio.h | FATFS和disk I/O共用引入文件 | 无需修改 |
| diskio.c | FATFS的diskI/O接口文件 | 平台相关代码，需用户自行实现 |
| interger.h | 数据类型定义 | 编译器相关 |
| option文件夹 | 可选其他功能，如中文支持 |  |

在FAT的移植中，最核心的部分是diskio文件，主要通过需要使用SDIO的API实现5个函数，分别为：disk\_status、disk\_initialize、disk\_read、disk\_write和disk\_ioctl。

disk\_status函数会返回当前存储设备的状态，以返回的status的值的不同区别。FatFS支持同时花在多个设备，SD卡一般将其定义为编号0，串行flash一般定义为1。SD卡的块大小通过宏定义设置，一般情况下，不论是哪种SD卡都是512字节。

disk\_initialize函数负责初始化存储设备，其调用SDIO的初始化API SD\_Init实现对sd卡的初始化，并根据返回值区分其初始化结果。

disk\_read函数的作用是在存储设备的指定地址处开始读取一定量的数据至指定区域里。其使用SDIO的SD\_ReadMultiBlocks函数去将多块数据读取至存储区内。因为SDIO的函数是使用数据大小为32bit的DMA实现的，因此存储区必须是32bit对齐的。若buff地址不是以32bit对齐的话，DMA传输会失败，因此需要检测地址是否为32bit对其，若是，则可以直接使用，若不是，则需要使用一个32bit的对其缓冲区，借助于DWORD数据类型可以使其自动的完成32bit对齐。这样便可以将一个完整的块数据放置于缓冲区内，在将缓冲区内的数据完整的复制与buff上即可。

disk\_write函数的作用是向存储设备写入指定量的数据。其使用SDIO的SD\_WriteMultiBlocks函数完成。其与disk\_read实现逻辑类似，也需检测32bit对齐，若是则正常执行，若不是则申请一个32bit对齐的缓冲区，先将代写数据复制于缓冲区后再写入至SD卡。最后会将执行完的状态以status的形式返回。

disk\_ioctl函数的三个形参分别是pdrv：设备的物理编号，cmd：控制指令，主要是同步信号、获取的扇区数目和大小等，buff：对应的数据指针。

4.2.3 SDIO简介

SD卡一般由存储单元、电源检测、存储单元接口、卡与接口控制器和接口驱动器组成。其中数据存储于存储单元，数据传输则是通过存储单元接口与卡控制单元完成，电源检测会确保其工作与稳定电压下，若出现掉电等现象时复位接口，控制单元主要通过8个寄存器来控制其运行状态，接口控制器则负责管控其输入输出流。其结构图如图4-2所示。

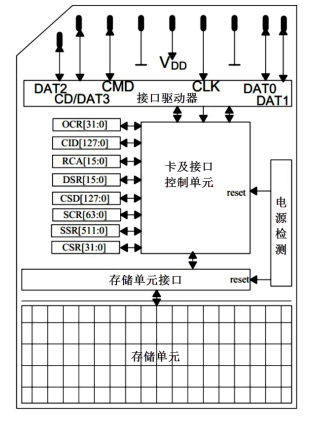


图4-2. SD卡物理结构

SDIO所发出的指令多为2种，广播与寻址，其中广播是针对总线上的所有设备的，而寻址则是针对特定地址的设备。其命令的格式为48bit固定，具体的指令构成如图4-3所示。

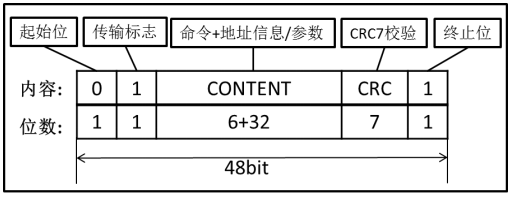


图4-3. SDIO指令构成

SD卡存在多个版本，对于STM32F4系列MCU来说其最高支持2.0版本的SD卡，SDIO在数据传输前需要先识别出SD卡的种类，STM32F4系列MCU会将其识别为：V1.0标准卡、V2.0标准卡、V2.0大容量卡与无法识别卡。

SD卡使用2种操作模式，分别为卡识别模式与数据传输模式。卡识别模式是在系统复位后寻找挂载于总线上的设备的。当卡被识别到并且接收到CMD3即SEND\_RCA命令后，SD卡将进入数据传输模式。卡识别模式的状态切换流程如图4-4所示。

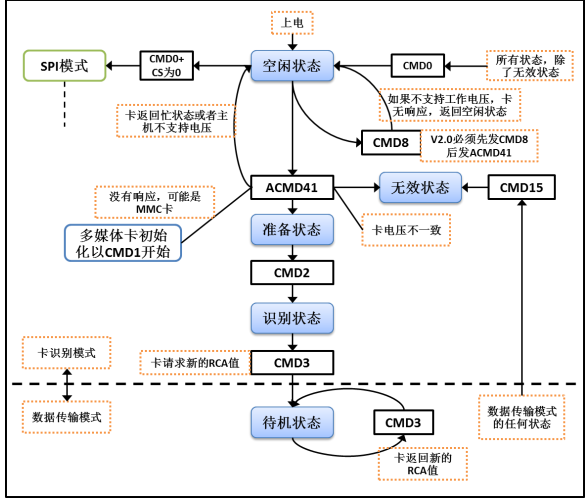


图4-4. 卡识别模式的状态切换流程图

当SD卡位于数据传输模式下时，其可以和主机进行数据的读写交换，时钟频率FPP最高可达25MHz，这也是其默认的数值。当其位于数据传输模式下时主机与SD卡的通讯均为点对点执行。当然，位于数据传输模式下的SD卡也可以通过使用CMD0和CMD15指令来返回到卡识别模式，但这可能会使SD卡中的数据受到损坏。其数据传输模式的状态切换流程如图4-5所示。

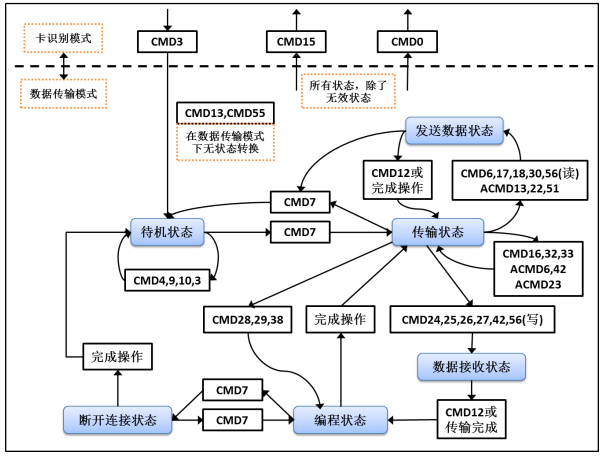


图4-5. 数据传输模式的状态切换流程图

SD卡的init函数除了初始化硬件设备外，主要是识别SD卡种类以及读取SD卡的相关状态，其整个识别的流程如图4-6所示。

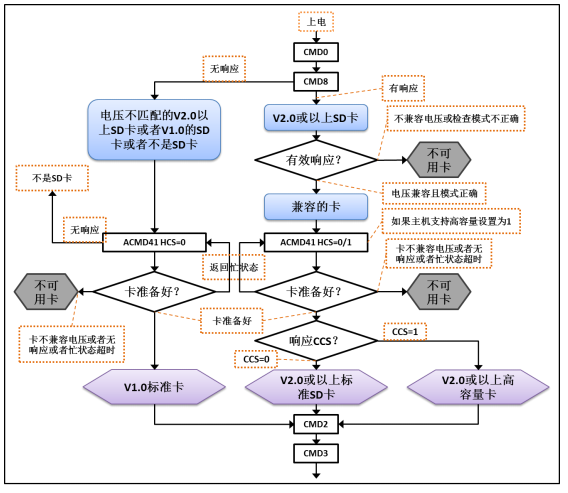


图4-6. SD卡识别流程图

其先通过NVIC将需要使用的SDIO中断和DMA相关中断配置，接着对底层硬件进行初始化操作，然后复位SDIO接口，恢复默认的设置值，最后调用SD\_PowerON，识别SD卡并返回。

4.3 用户程序设计

main.c文件中较为主要的是4个Task，分别为SD\_Task、LOG\_Task、HMI\_Task和MODBUS\_Task。线程之间通过二值信号量进行同步，其具体的相关关系如图4-7所示。

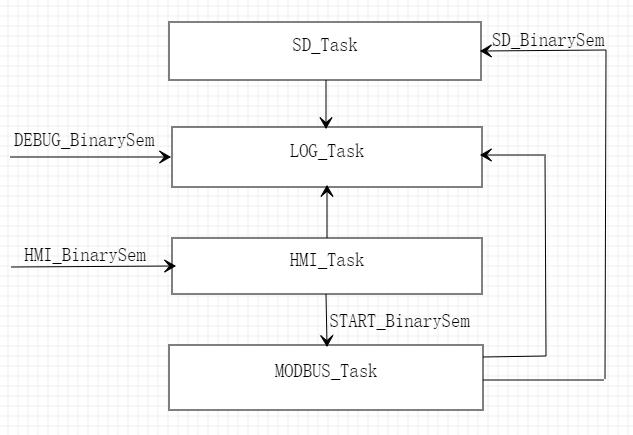


图4-7. 线程关系图

系统在初始化后所有线程均处于阻塞状态，系统运行于IDLE空闲程序中，可以将其置于低功耗模式以达到节约功耗的目的。待HMI屏幕输入启动讯号后中断服务函数会检测并给出二值信号量从而唤醒HMI\_Task做进一步的检测，通过HMI\_Task逐渐释放其他二值信号量逐步唤醒其他线程，最终4个线程均被唤醒，整个系统完全处于运行中。

HMI\_Task收到二值信号量后首先翻转LED，通过使用LED2\_TOGGLE这个定义于bsp\_led的宏函数提示接收数据。数据是通过DMA从外设搬运至内存指定位置的位置，HMI的缓冲区设置的大小是1k，完全可以满足目前的使用情况以及未来的拓展。检测完数据帧的头部后，根据不同的功能码设置不同的标志位，以决定后续的任务如何执行。

MODBUS\_Task为采集模块的通讯线程，其在收到HMI\_Task的二值信号量后，会循环发送查询指令，查询指令的完整数据帧均为固定帧，因此全部使用静态const数组定义。再每次发送查询指令后，理论上应答会在较短时间内回复，因此这部分的阻塞定义了最大阻塞时间为300ms，若应答超出了等待时限则忽略此次应答。在接收到应答的同时翻转另一个LED用以提示数据收到，对应答数据首先要进行CRC的校验，以确保收到的数据的完整性，若CRC校验失败则忽略本次应答，进行下一次的请求。若CRC检验通过则进行功能码的区分，因目前只作为数据接收，因此其他功能码暂时没有实现。整个流程完整的运行一次需要经过24次请求——应答，当24次全部完成时会翻转一个LED提示整个通讯完成，并且释放SD\_Task的二值信号量。

SD\_Task线程负责数据的保存与发送，在阻塞前其会先完成SD卡的挂载，然后阻塞等待二值信号量的释放，得到信号量后根据标志位和文件统计变量进行数据的保存，其在追加写入时需先偏移文件指针，再调用文件系统的API将信息写入。根据显示标志位决定是否给HMI屏幕发送显示指令。

4.4 GUI设计

GUI采用陶晶驰的界面开发软件完成，其具有可视化编辑、自带模拟调试窗口、C语言兼容以及多达数十种控件的优点。并且其可以使用户按自己需要的字体和字号选择参数，自动生成字库。

GUI目前由3个页面组成，其相互关系如图4-8所示。

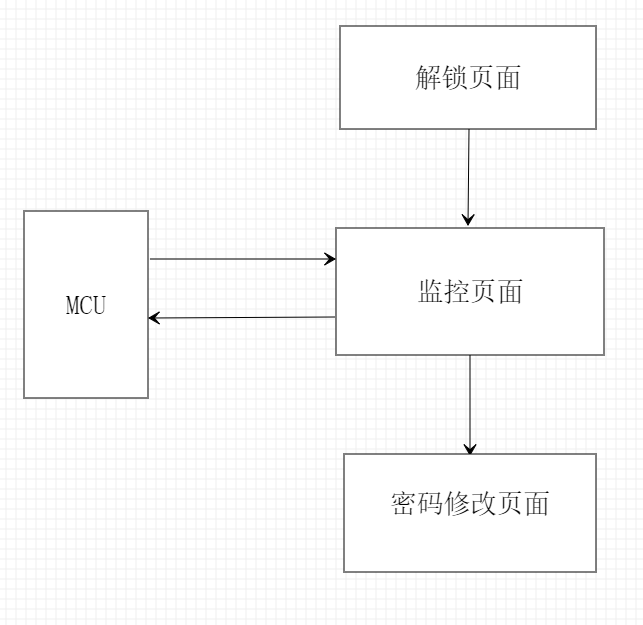


图4-8. GUI页面构成

解锁页面需要用户输入系统密码，整个系统在进入监控页面前都为停滞状态，确保信息安全。密码默认不可见，避免密码泄露，但设置了一个可见按钮，当其按下时密码暂时可见，防止误输入。密码输入错误会有提示，并且在多次错误后系统会进入死锁，仅管理员密码可以使用，密码正确则进入监控页面。解锁页面如图4-9所示。

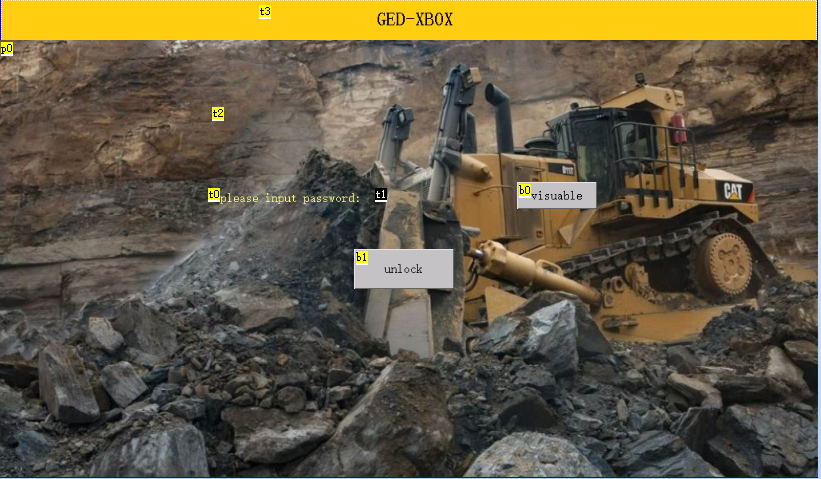


图4-9. 解锁页面

监控页面显示采集到的数据，通过右上角的下拉菜单选择当前监控的传感器组，因传感器过多，因此将其以4个为一组分为6组。start按钮按下后系统开始采集并将数据显示在图表中，其量程会根据不同的传感器组自动的调整。save按钮按下后开始将采集到的数据保存至SD卡，因为刚开始采集到的数据可能存在比较大的波动，属于无效数据，因此待操作者确认数据稳定且无误后开始保存。stop按钮按下后系统会停止显示与记录。监控画面如图4-10所示。

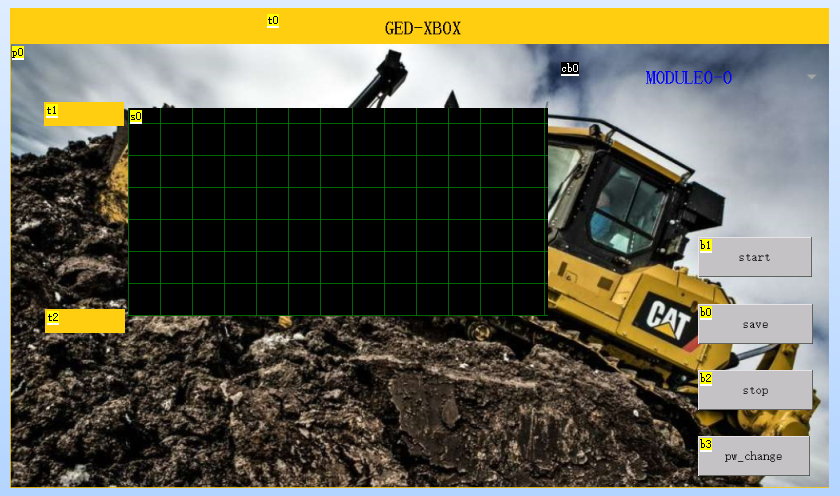


图4-10. 监控页面

密码修改页面与解锁页面类似，但是没有错误锁死功能，新密码需要输入2遍且一致，防止误输入。具体设计图4-11所示。

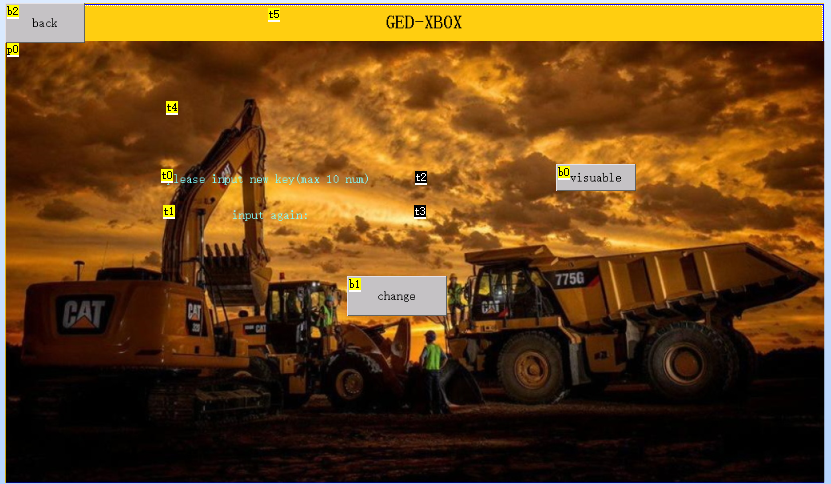


图4-11. 密码修改页面

4.5 本章小结

本章系统性的介绍了系统的软件设计，首先宏观的分析了软件使用的嵌入式实时操作系统——FreeRTOS，侧重于其在本系统硬件平台上的实现以及其对内存的管理策略。接着说明了本系统使用的文件系统，在数据采集中，数据的保存是和采集一样重要的，若使用了不恰当的保存方式，可能使得采集到的数据杂乱无章，甚至可能会丢失、错误记录等，严重影响系统鲁棒性。本系统使用的FAT文件系统实现逻辑精简，十分适合于系统的使用场景，并较为详细的剖析了其与硬件的耦合。然后是软件的程序逻辑解析，多线程同步的运行逻辑更加的鲁棒，各个任务之间较为独立，并有着系统的调度，倘若某个线程出现问题对整个系统与其他的线程的影响也会降到最低。最后展示了系统GUI的设计，页面间的逻辑与MCU与屏幕的联系。

第5章 结论与展望

5.1结论

在系统全部搭建完成后进行了联调测试，并与项目组进行了讨论分析，逐步迭代，保持系统的目标不变的前提下尽可能的完善功能，并且结合实际情况与开发周期对计划进行修正。受疫情影响，后期的开发被迫远程进行，在硬件环境无法满足的情况下，使用软件去模拟、仿真。实际验证时发现仿真、模拟与现实的环境还是有着一定的差距，很多无关变量的影响逐步累计最后影响到功能的实现。搭建实际硬件环境时电路的连接、测试等也有着很多的细节。

总的来说，在本项目的进行中我学习到了很多，也发现了很多自己的不足，因为自己的粗心给项目带来了实际的金钱、时间损失，理论与实践还是存在一定的差别，理论多是理想的模型，现实中存在着各种各样的无关变量，可能单论某一个都是无关紧要的，但是多个无关变量相互作用、耦合，最终影响到了整个系统。

5.2不足之处及未来展望

整个项目很早就已经开始实施，从最初的项目组讨论到方案制定、硬件选型、采购、测试、系统搭建、软件开发、联调结束。在初期讨论时，并未考虑到很多后期的实现，某些理论上可行的方案就默认可以实现去确认了，在后续开发中发现后对开发周期有着较大的影响。幸运的是项目组和我的主管都是经验丰富的开发人员，给出了很多我以前不了解的解决方案。在与负责采购的同事沟通时我才发现企业的采购流程十分繁杂与严谨，我计划中给出的时间完全不满足，这也大大延缓了开发进度。除此之外还有很多细节上的东西，我最初所想的都是比较理想的，也并未全面考虑各个部分，在实际流程中遇到了诸多的麻烦。最终在项目组的帮助下均解决。

未来在项目制定时会更多的考虑到实际落地的难度，而非仅仅做理论上的分析；遇到困难也会主动和大家一起集思广益，而非自己闷声尝试解决；在测试与迭代阶段会留出更多的时间与空间，而非简单的仅将其当做一个验证的过程。

参考文献

1. 王琳, 商周, 王学伟. 数据采集系统的发展与应用[J]. 电测与仪表, 2004, 41(8):5.
2. 王建萍, 费跃农. 嵌入式高速数据采集系统的实时性研究[J]. 电子测量与仪器学报, 2006, 20(6):4.
3. Rashmi Sharan Sinha, Yiqiao Wei, Seung-Hoon Hwang. A survey on LPWA technology: LoRa and NB-IoT. [J] ICT Express 3 (2017) 14–21
4. sai K L , Leu F Y , You I , et al. Low-Power AES Data Encryption Architecture for a LoRaWAN[J]. IEEE Access, 2019, PP(99):1-1.
5. 李宁，《模拟电路》[M]. 清华大学出版社，2011.
6. 杨立，《微机原理》[M]. 人民邮电出版社，2007.
7. 景德胜.基于多核处理器的弹载计算机低功耗设计研究[J].微型机与应用2016, 35(10):3.
8. 陈超群, 魏宏磊. 嵌入式操作系统的低功耗组件设计[J]. 单片机与嵌入式系统应用, 2021, 21(11):4.
9. 李庆义, 武彩宏, LiQing-yi,等. 基于μC/OS-Ⅱ实时嵌入式操作系统平台上的应用接口编程[J]. 科技创新与生产力, 2010(2):101-103.
10. 杨静远, 项涛, 陈程. 多核下嵌入式操作系统原子操作考虑要点[J]. 电脑编程技巧与维护, 2021(2):3.
11. 唐亮. 一种嵌入式操作系统下串口设备高速率数据传输方法:, CN112231265A[P]. 2021.
12. 陈丹桂,唐佳.嵌入式系统动态低功耗研究[J].现代计算机（专业版）, 2015(12):4.
13. Voltage tunable low damping YIG/PMN-PT multiferroic heterostructure for low-power RF/microwave devices[J]. Journal of Physics D: Applied Physics, 2021, 54(24):245002 (7pp).
14. Spaur C W , Braitberg M F , Kennedy P J , et al. Mobile portable wireless communication system: US, US5732074 A[P]. 1997.
15. 王东庆, 周建华. 一种基于脑电的智能家居控制系统[J]. 电子技术应用.
16. 张旭, 亓学广, 李世光,等. 基于STM32电力数据采集系统的设计[J]. 电子测量技术, 2010(11):4.
17. 武利珍, 张文超, 程春荣. 基于STM32的便携式心电图仪设计[J]. 电子器件, 2009, 32(5):4.
18. 丁力, 宋志平, 徐萌萌,等. 基于STM32的嵌入式测控系统设计[J]. 中南大学学报：自然科学版, 2013(S1):6.
19. 陈进京, 姜志轩, 王泓平. 温度采集系统设计[J]. 电子测试, 2022(3):3.

致 谢

附录A： 作者在校期间发表的论文

[1] [1]王纲, 平雪良, 肖竟成,等. 具有爬梯功能的新型电动轮椅[J]. 机械制造, 2021, 59(10):3.

附录B： 主程序

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file main.c

\* @author jOKERII

\* @version V1.0

\* @date 2022-03-01

\* @brief DEMO

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @attention

\*

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 包含的头文件

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

/\* FreeRTOS头文件 \*/

#include "FreeRTOS.h"

#include "task.h"

#include "queue.h"

#include "semphr.h"

/\* 开发板硬件bsp头文件 \*/

#include "bsp\_led.h"

#include "bsp\_debug\_usart.h"

#include "bsp\_modbus\_usart.h"

#include "bsp\_hmi\_usart.h"

#include "bsp\_key.h"

#include "ff.h"

#include "./FatFs\_Test/FatFs\_test.h"

/\* 标准库头文件 \*/

#include <string.h>

#include <stdio.h>

/\* CRC-16 MODBUS \*/

#include "CRC16.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 任务句柄 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* 任务句柄是一个指针，用于指向一个任务，当任务创建好之后，它就具有了一个任务句柄

\* 以后我们要想操作这个任务都需要通过这个任务句柄，如果是自身的任务操作自己，那么

\* 这个句柄可以为NULL。

\*/

static TaskHandle\_t AppTaskCreate\_Handle = NULL;/\* 创建任务句柄 \*/

static TaskHandle\_t Test\_Task\_Handle = NULL;/\* LED任务句柄 \*/

static TaskHandle\_t MODBUS\_Task\_Handle = NULL;/\* KEY任务句柄 \*/

static TaskHandle\_t Uart\_Task\_Handle = NULL;/\* KEY任务句柄 \*/

static TaskHandle\_t HMI\_Task\_Handle = NULL;/\* KEY任务句柄 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 内核对象句柄 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* 信号量，消息队列，事件标志组，软件定时器这些都属于内核的对象，要想使用这些内核

\* 对象，必须先创建，创建成功之后会返回一个相应的句柄。实际上就是一个指针，后续我

\* 们就可以通过这个句柄操作这些内核对象。

\*

\* 内核对象说白了就是一种全局的数据结构，通过这些数据结构我们可以实现任务间的通信，

\* 任务间的事件同步等各种功能。至于这些功能的实现我们是通过调用这些内核对象的函数

\* 来完成的

\*

\*/

SemaphoreHandle\_t DEBUG\_BinarySem\_Handle = NULL;

SemaphoreHandle\_t MODBUS\_BinarySem\_Handle = NULL;

SemaphoreHandle\_t SD\_BinarySem\_Handle = NULL;

SemaphoreHandle\_t HMI\_BinarySem\_Handle = NULL;

SemaphoreHandle\_t START\_BinarySem\_Handle = NULL;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 全局变量声明 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* 当我们在写应用程序的时候，可能需要用到一些全局变量。

\*/

extern uint8\_t Debug\_Usart\_Rx\_Buf[DEBUG\_USART\_RBUFF\_SIZE];

extern uint8\_t MODBUS1\_Usart\_Rx\_Buf[MODBUS1\_USART\_RBUFF\_SIZE];

extern uint8\_t HMI\_Usart\_Rx\_Buf[HMI\_USART\_RBUFF\_SIZE];

uint8\_t MODBUS1[8][8] = {0x01, 0x03, 0x00, 0x00, 0x00, 0x01, 0x84, 0x0A,

0x01, 0x03, 0x00, 0x00, 0x01, 0x01, 0x85, 0x9A,

0x01, 0x03, 0x00, 0x00, 0x02, 0x01, 0x85, 0x6A,

0x01, 0x03, 0x00, 0x00, 0x03, 0x01, 0x84, 0xFA,

0x01, 0x03, 0x00, 0x00, 0x04, 0x01, 0x86, 0xCA,

0x01, 0x03, 0x00, 0x00, 0x05, 0x01, 0x87, 0x5A,

0x01, 0x03, 0x00, 0x00, 0x06, 0x01, 0x87, 0xAA,

0x01, 0x03, 0x00, 0x00, 0x07, 0x01, 0x86, 0x3A}; //站号 01 功能码 03 起始地址 0000 寄存器数量 0001 CRC 840A

/\*---------------- ------------------\*/

uint8\_t MODBUS2[8][8] = {0x02, 0x03, 0x00, 0x00, 0x00, 0x01, 0x84, 0x39,

0x02, 0x03, 0x00, 0x00, 0x01, 0x01, 0x85, 0xA9,

0x02, 0x03, 0x00, 0x00, 0x02, 0x01, 0x85, 0x59,

0x02, 0x03, 0x00, 0x00, 0x03, 0x01, 0x84, 0xC9,

0x02, 0x03, 0x00, 0x00, 0x04, 0x01, 0x86, 0xF9,

0x02, 0x03, 0x00, 0x00, 0x05, 0x01, 0x87, 0xC8,

0x02, 0x03, 0x00, 0x00, 0x06, 0x01, 0x87, 0x99,

0x02, 0x03, 0x00, 0x00, 0x07, 0x01, 0x86, 0x09}; //站号 02 功能码 03 起始地址 0000 寄存器数量 0001 CRC 840A

uint8\_t MODBUS3[8][8] = {0x03, 0x03, 0x00, 0x00, 0x00, 0x01, 0x85, 0xE8,

0x03, 0x03, 0x00, 0x00, 0x01, 0x01, 0x84, 0x78,

0x03, 0x03, 0x00, 0x00, 0x02, 0x01, 0x84, 0x88,

0x03, 0x03, 0x00, 0x00, 0x03, 0x01, 0x85, 0x18,

0x03, 0x03, 0x00, 0x00, 0x04, 0x01, 0x87, 0x28,

0x03, 0x03, 0x00, 0x00, 0x05, 0x01, 0x86, 0xB8,

0x03, 0x03, 0x00, 0x00, 0x06, 0x01, 0x86, 0x48,

0x03, 0x03, 0x00, 0x00, 0x07, 0x01, 0x87, 0xD8}; //站号 03 功能码 03 起始地址 0000 寄存器数量 0001 CRC 840A

/\*-------------------------------\*/

uint16\_t channel\_data\_buf[3][50] = {0};

char ascii\_buf[100] = {0};

char file\_name[100] = {0};

char text1[] = "\r\n第一组：\r\n";

char text2[] = "\r\n第二组：\r\n";

char text3[] = "\r\n第三组：\r\n";

int8\_t channel\_num = 0;

uint8\_t display = 0;

uint8\_t save\_flag = 0;

uint8\_t file\_num = 0;

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 函数声明

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

static void AppTaskCreate(void);/\* 用于创建任务 \*/

static void Test\_Task(void\* pvParameters);/\* Test\_Task任务实现 \*/

static void MODBUS\_Task(void\* pvParameters);/\* MODBUS\_Task任务实现 \*/

static void Uart\_Task(void\* pvParameters);/\* KEY\_Task任务实现 \*/

static void HMI\_Task(void\* pvParameters);/\* KEY\_Task任务实现 \*/

static void BSP\_Init(void);/\* 用于初始化板载相关资源 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @brief 主函数

\* @param 无

\* @retval 无

\* @note 第一步：开发板硬件初始化

第二步：创建APP应用任务

第三步：启动FreeRTOS，开始多任务调度

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

BaseType\_t xReturn = pdPASS;/\* 定义一个创建信息返回值，默认为pdPASS \*/

/\* 开发板硬件初始化 \*/

BSP\_Init();

printf("RTOS\_Init\n\n");

/\* 创建AppTaskCreate任务 \*/

xReturn = xTaskCreate((TaskFunction\_t )AppTaskCreate, /\* 任务入口函数 \*/

(const char\* )"AppTaskCreate",/\* 任务名字 \*/

(uint16\_t )512, /\* 任务栈大小 \*/

(void\* )NULL,/\* 任务入口函数参数 \*/

(UBaseType\_t )1, /\* 任务的优先级 \*/

(TaskHandle\_t\* )&AppTaskCreate\_Handle);/\* 任务控制块指针 \*/

/\* 启动任务调度 \*/

if(pdPASS == xReturn)

vTaskStartScheduler(); /\* 启动任务，开启调度 \*/

else

return -1;

while(1); /\* 正常不会执行到这里 \*/

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @ 函数名 ： AppTaskCreate

\* @ 功能说明： 为了方便管理，所有的任务创建函数都放在这个函数里面

\* @ 参数 ： 无

\* @ 返回值 ： 无

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void AppTaskCreate(void)

{

BaseType\_t xReturn = pdPASS;/\* 定义一个创建信息返回值，默认为pdPASS \*/

taskENTER\_CRITICAL(); //进入临界区

/\* 创建 DEBUG\_BinarySem\_Handle \*/

DEBUG\_BinarySem\_Handle = xSemaphoreCreateBinary();

if(NULL != DEBUG\_BinarySem\_Handle)

printf("DEBUG\_BinarySem\_Handle二值信号量创建成功!\n");

/\* 创建 BinarySem \*/

MODBUS\_BinarySem\_Handle = xSemaphoreCreateBinary();

if(NULL != MODBUS\_BinarySem\_Handle)

printf("MODBUS\_BinarySem\_Handle二值信号量创建成功!\n");

/\* 创建 BinarySem \*/

SD\_BinarySem\_Handle = xSemaphoreCreateBinary();

if(NULL != SD\_BinarySem\_Handle)

printf("SD\_BinarySem\_Handle二值信号量创建成功!\n");

/\* 创建 BinarySem \*/

HMI\_BinarySem\_Handle = xSemaphoreCreateBinary();

if(NULL != HMI\_BinarySem\_Handle)

printf("HMI\_BinarySem\_Handle二值信号量创建成功!\n");

/\* 创建 BinarySem \*/

START\_BinarySem\_Handle = xSemaphoreCreateBinary();

if(NULL != START\_BinarySem\_Handle)

printf("START\_BinarySem\_Handle二值信号量创建成功!\n");

/\* 创建Test\_Task任务 \*/

xReturn = xTaskCreate((TaskFunction\_t )Test\_Task, /\* 任务入口函数 \*/

(const char\* )"Test\_Task",/\* 任务名字 \*/

(uint16\_t )512, /\* 任务栈大小 \*/

(void\* )NULL, /\* 任务入口函数参数 \*/

(UBaseType\_t )5, /\* 任务的优先级 \*/

(TaskHandle\_t\* )&Test\_Task\_Handle);/\* 任务控制块指针 \*/

if(pdPASS == xReturn)

printf("创建Test\_Task任务成功!\r\n");

/\* 创建Uart\_Task任务 \*/

xReturn = xTaskCreate((TaskFunction\_t )Uart\_Task, /\* 任务入口函数 \*/

(const char\* )"Uart\_Task",/\* 任务名字 \*/

(uint16\_t )512, /\* 任务栈大小 \*/

(void\* )NULL,/\* 任务入口函数参数 \*/

(UBaseType\_t )3, /\* 任务的优先级 \*/

(TaskHandle\_t\* )&Uart\_Task\_Handle);/\* 任务控制块指针 \*/

if(pdPASS == xReturn)

printf("创建Uart\_Task任务成功!\r\n");

/\* 创建MODBUS\_Task任务 \*/

xReturn = xTaskCreate((TaskFunction\_t )MODBUS\_Task, /\* 任务入口函数 \*/

(const char\* )"MODBUS\_Task",/\* 任务名字 \*/

(uint16\_t )512, /\* 任务栈大小 \*/

(void\* )NULL,/\* 任务入口函数参数 \*/

(UBaseType\_t )4, /\* 任务的优先级 \*/

(TaskHandle\_t\* )&MODBUS\_Task\_Handle);/\* 任务控制块指针 \*/

if(pdPASS == xReturn)

printf("创建MODBUS\_Task任务成功!\r\n");

/\* 创建MODBUS\_Task任务 \*/

xReturn = xTaskCreate((TaskFunction\_t )HMI\_Task, /\* 任务入口函数 \*/

(const char\* )"HMI\_Task",/\* 任务名字 \*/

(uint16\_t )512, /\* 任务栈大小 \*/

(void\* )NULL,/\* 任务入口函数参数 \*/

(UBaseType\_t )6, /\* 任务的优先级 \*/

(TaskHandle\_t\* )&HMI\_Task\_Handle);/\* 任务控制块指针 \*/

if(pdPASS == xReturn)

printf("创建HMI\_Task任务成功!\r\n");

vTaskDelete(AppTaskCreate\_Handle); //删除AppTaskCreate任务

taskEXIT\_CRITICAL(); //退出临界区

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @ 函数名 ： Test\_Task

\* @ 功能说明： Test\_Task任务主体

\* @ 参数 ：

\* @ 返回值 ： 无

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void Test\_Task(void\* parameter)

{

BaseType\_t xReturn = pdPASS;/\* 定义一个创建信息返回值，默认为pdPASS \*/

FRESULT res\_sd; /\* 文件操作结果 \*/

FIL fnew; /\* 文件对象 \*/

UINT fnum;

uint8\_t flag = 0;

/\* 文件成功读写数量 \*/

char cmd[80] = {0};

/\* 文件系统初始化 \*/

FileSystem\_Init();

//sprintf(file\_name, "0:data%d.txt", file\_num);

//printf("\r\n\*\*\*\*\*\* 新建txt... \*\*\*\*\*\*\r\n");

//res\_sd = f\_open(&fnew, file\_name,FA\_CREATE\_ALWAYS | FA\_WRITE );

//f\_close(&fnew);

///\* 文件系统读写测试 \*/

//FileSystem\_Test();

while (1)

{

xReturn = xSemaphoreTake(SD\_BinarySem\_Handle,/\* 二值信号量句柄 \*/

portMAX\_DELAY); /\* 等待时间 \*/

if(pdPASS == xReturn)

{

if(save\_flag)

{

LED1\_TOGGLE;

if(flag != file\_num)

{

printf("\r\n\*\*\*\*\*\* 新建txt... \*\*\*\*\*\*\r\n");

res\_sd = f\_open(&fnew, file\_name,FA\_CREATE\_ALWAYS | FA\_WRITE );

f\_close(&fnew);

flag = file\_num;

}

res\_sd = f\_open(&fnew, file\_name,FA\_OPEN\_EXISTING | FA\_WRITE );

for(int temp = 0; temp < 3; temp++) {

res\_sd=f\_lseek(&fnew,f\_size(&fnew));

if(res\_sd==FR\_OK)

{

printf("偏移完毕\r\n");

}

switch(temp) {

case(0): {

res\_sd=f\_write(&fnew,text1,sizeof(text1),&fnum);

break;

}

case(1): {

res\_sd=f\_write(&fnew,text2,sizeof(text2),&fnum);

break;

}

case(2): {

res\_sd=f\_write(&fnew,text3,sizeof(text3),&fnum);

break;

}

}

res\_sd=f\_lseek(&fnew,f\_size(&fnew));

if(res\_sd==FR\_OK)

{

printf("偏移完毕\r\n");

}

sprintf(ascii\_buf, "%d %d %d %d %d %d %d %d\r\n", (int)channel\_data\_buf[temp][0], (int)channel\_data\_buf[temp][1], (int)channel\_data\_buf[temp][2], (int)channel\_data\_buf[temp][3], (int)channel\_data\_buf[temp][4], (int)channel\_data\_buf[temp][5], (int)channel\_data\_buf[temp][6], (int)channel\_data\_buf[temp][7]);

res\_sd=f\_write(&fnew,ascii\_buf,sizeof(ascii\_buf),&fnum);

memset(ascii\_buf,0,sizeof(ascii\_buf));/\* 清零 \*/

if(res\_sd==FR\_OK)

{

printf("》文件写入成功，写入字节数据：%d\r\n",fnum);

//printf("》向文件写入的数据为：\r\n%s\r\n",(char\*)channel\_data\_buf[temp]);

}

}

f\_close(&fnew);

}

if(display)

{

sprintf(cmd, "add 2,0,%d",(uint16\_t)channel\_data\_buf[channel\_num/2][0+(channel\_num%2)\*4]/2621);

Usart\_SendString(HMI\_USART, cmd);

Usart\_SendByte(HMI\_USART, 0xFF);

Usart\_SendByte(HMI\_USART, 0xFF);

Usart\_SendByte(HMI\_USART, 0xFF);

sprintf(cmd, "add 2,1,%d",(uint16\_t)channel\_data\_buf[channel\_num/2][1+(channel\_num%2)\*4]/2621);

Usart\_SendString(HMI\_USART, cmd);

Usart\_SendByte(HMI\_USART, 0xFF);

Usart\_SendByte(HMI\_USART, 0xFF);

Usart\_SendByte(HMI\_USART, 0xFF);

sprintf(cmd, "add 2,2,%d",(uint16\_t)channel\_data\_buf[channel\_num/2][2+(channel\_num%2)\*4]/2621);

Usart\_SendString(HMI\_USART, cmd);

Usart\_SendByte(HMI\_USART, 0xFF);

Usart\_SendByte(HMI\_USART, 0xFF);

Usart\_SendByte(HMI\_USART, 0xFF);

sprintf(cmd, "add 2,3,%d",(uint16\_t)channel\_data\_buf[channel\_num/2][3+(channel\_num%2)\*4]/2621);

Usart\_SendString(HMI\_USART, cmd);

Usart\_SendByte(HMI\_USART, 0xFF);

Usart\_SendByte(HMI\_USART, 0xFF);

Usart\_SendByte(HMI\_USART, 0xFF);

}

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @ 函数名 ： LED\_Task

\* @ 功能说明： LED\_Task任务主体

\* @ 参数 ：

\* @ 返回值 ： 无

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void Uart\_Task(void\* parameter)

{

BaseType\_t xReturn = pdPASS;/\* 定义一个创建信息返回值，默认为pdPASS \*/

while (1)

{

//获取二值信号量 xSemaphore,没获取到则一直等待

xReturn = xSemaphoreTake(DEBUG\_BinarySem\_Handle,/\* 二值信号量句柄 \*/

portMAX\_DELAY); /\* 等待时间 \*/

if(pdPASS == xReturn)

{

LED2\_TOGGLE;

printf("收到数据:%s\n",Debug\_Usart\_Rx\_Buf);

memset(Debug\_Usart\_Rx\_Buf,0,DEBUG\_USART\_RBUFF\_SIZE);/\* 清零 \*/

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @ 函数名 ： LED\_Task

\* @ 功能说明： LED\_Task任务主体

\* @ 参数 ：

\* @ 返回值 ： 无

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void HMI\_Task(void\* parameter)

{

BaseType\_t xReturn = pdPASS;/\* 定义一个创建信息返回值，默认为pdPASS \*/

FRESULT res\_sd; /\* 文件操作结果 \*/

FIL fnew; /\* 文件对象 \*/

while (1)

{

//获取二值信号量 xSemaphore,没获取到则一直等待

xReturn = xSemaphoreTake(HMI\_BinarySem\_Handle,/\* 二值信号量句柄 \*/

portMAX\_DELAY); /\* 等待时间 \*/

if(pdPASS == xReturn)

{

LED2\_TOGGLE;

//printf("收到数据:%s\n",HMI\_Usart\_Rx\_Buf);

if(HMI\_Usart\_Rx\_Buf[0] == 0x55 && HMI\_Usart\_Rx\_Buf[1] == 0xAA)

{

channel\_num = HMI\_Usart\_Rx\_Buf[2]-48;

if(channel\_num > -1 && HMI\_Usart\_Rx\_Buf[3] == 0x00 && HMI\_Usart\_Rx\_Buf[4] == 0xAA && HMI\_Usart\_Rx\_Buf[5] == 0x55)

{

//给出二值信号量 ，发送接收到新数据标志，供前台程序查询

xSemaphoreGive(START\_BinarySem\_Handle); //释放二值信号量

display = 1;

}

else

{

if(channel\_num == -48 && HMI\_Usart\_Rx\_Buf[3] == 0x00 && HMI\_Usart\_Rx\_Buf[4] == 0xAA && HMI\_Usart\_Rx\_Buf[5] == 0x55)

{

display = 0;

save\_flag = 0;

}

if(HMI\_Usart\_Rx\_Buf[2] == 0xFF && HMI\_Usart\_Rx\_Buf[3] == 0x00 && HMI\_Usart\_Rx\_Buf[4] == 0xAA && HMI\_Usart\_Rx\_Buf[5] == 0x55)

{

save\_flag = 1;

file\_num++;

sprintf(file\_name, "0:data%d.txt", file\_num);

}

}

}

else

{

}

printf("channel num = %d\r\n", channel\_num);

//printf("HMI\_Usart\_Rx\_Buf[2] = %u\r\n", HMI\_Usart\_Rx\_Buf[2]);

memset(HMI\_Usart\_Rx\_Buf,0,HMI\_USART\_RBUFF\_SIZE);/\* 清零 \*/

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @ 函数名 ： Test\_Task

\* @ 功能说明： Test\_Task任务主体

\* @ 参数 ：

\* @ 返回值 ： 无

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void MODBUS\_Task(void\* parameter)

{

uint8\_t channel = 0, num = 0;

unsigned int crc;

unsigned char crch, crcl;

uint8\_t len = 7, data\_len = 0;

BaseType\_t xReturn = pdPASS;/\* 定义一个创建信息返回值，默认为pdPASS \*/

xReturn = xSemaphoreTake(START\_BinarySem\_Handle,/\* 二值信号量句柄 \*/

portMAX\_DELAY); /\* 等待时间 \*/

if(pdPASS == xReturn)

{

while (1)

{

for(num = 0; num < 3; num++) {

for(channel = 0; channel < 8; channel ++) {

switch(num)

{

case(0):

{

for(int temp = 0; temp < 8; temp++) {

Usart\_SendByte(MODBUS1\_USART, MODBUS1[channel][temp]);

}

break;

}

case(1):

{

for(int temp = 0; temp < 8; temp++) {

Usart\_SendByte(MODBUS1\_USART, MODBUS2[channel][temp]);

}

break;

}

case(2):

{

for(int temp = 0; temp < 8; temp++) {

Usart\_SendByte(MODBUS1\_USART, MODBUS3[channel][temp]);

}

break;

}

}

//获取二值信号量 xSemaphore,没获取到则一直等待

xReturn = xSemaphoreTake(MODBUS\_BinarySem\_Handle,/\* 二值信号量句柄 \*/

portMAX\_DELAY); /\* 等待时间 \*/

if(pdPASS == xReturn)

{

LED3\_TOGGLE;

//printf("收到数据:%s \r\n",MODBUS1\_Usart\_Rx\_Buf);

if(MODBUS1\_Usart\_Rx\_Buf[0] == num+1) //判断地址是不是01

{

crc = GetCRC16((uint8\_t \*)MODBUS1\_Usart\_Rx\_Buf, len-2); //计算CRC校验值

crch = crc >> 8;

crcl = crc & 0xFF;

if ((MODBUS1\_Usart\_Rx\_Buf[len-2] == crch) && (MODBUS1\_Usart\_Rx\_Buf[len-1] == crcl)) //判断CRC校验是否正确

{

switch (MODBUS1\_Usart\_Rx\_Buf[1]) //按功能码执行操作

{

case 0x03: //读取一个或连续的寄存器

{

data\_len = MODBUS1\_Usart\_Rx\_Buf[2];

for(int temp = 0; temp < data\_len; temp++)

{

channel\_data\_buf[num][channel] = channel\_data\_buf[num][channel] << 8;

channel\_data\_buf[num][channel] |= MODBUS1\_Usart\_Rx\_Buf[3+temp];

}

break;

}

default: //其它不支持的功能码

printf("err，功能码\r\n");

channel\_data\_buf[num][channel] = 0;

break;

}

printf("收到数据:%d \r\n",channel\_data\_buf[num][channel]);

}

else

{

printf("err, CRC\r\n");

channel\_data\_buf[num][channel] = 0;

}

}

else

{

printf("err, 设备号\r\n");

channel\_data\_buf[num][channel] = 0;

}

memset(MODBUS1\_Usart\_Rx\_Buf,0,DEBUG\_USART\_RBUFF\_SIZE);/\* 清零 \*/

}

}

}

/\*-----完整的一组出来了-----\*/

LED4\_TOGGLE;

//给出二值信号量 ，发送接收到新数据标志，供前台程序查询

xSemaphoreGive(SD\_BinarySem\_Handle); //释放二值信号量

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @ 函数名 ： BSP\_Init

\* @ 功能说明： 板级外设初始化，所有板子上的初始化均可放在这个函数里面

\* @ 参数 ：

\* @ 返回值 ： 无

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void BSP\_Init(void)

{

/\*

\* STM32中断优先级分组为4，即4bit都用来表示抢占优先级，范围为：0~15

\* 优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，

\* 都统一用这个优先级分组，千万不要再分组，切忌。

\*/

NVIC\_PriorityGroupConfig( NVIC\_PriorityGroup\_4 );

/\* LED 初始化 \*/

LED\_GPIO\_Config();

/\* DMA初始化 \*/

Debug\_DMA\_Config();

/\* 串口初始化 \*/

Debug\_USART\_Config();

/\* DMA初始化 \*/

MODBUS1\_DMA\_Config();

/\* 串口初始化 \*/

MODBUS1\_USART\_Config();

/\* DMA初始化 \*/

HMI\_DMA\_Config();

/\* 串口初始化 \*/

HMI\_USART\_Config();

/\* 按键初始化 \*/

Key\_GPIO\_Config();

printf("Init finish\r\n");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END OF FILE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/