嵌入式基础与实践手札

# 嵌入式手札一

## 1.1.1、嵌入式系统的定义

嵌入式系统是一种计算机硬件和软件的组合，也许还有机械装置，用千实现一个特定功能。在某些特定情况下，嵌入式系统是一个大系统或产品的一部分

## 1.1.2嵌入式系统的由来及发展简史

通俗地说，计算机是因科学家需要一个高速的计算工具而产生的。为了区分两种计算机类型，通常把满足海量高速数值计算的计算机称为通用计算机系统，而把嵌入到实际应用系统中， 实现嵌入式应用的计算机称为嵌入式计算机系统，简称嵌入式系统。可以说，是因为通信、测控与数据传输等领域对计算机技术的需求催生了嵌入式系统的产生。

## 1.1.3 嵌入式系统的分类

从嵌入式系统的学习角度来看，因应用于不同领域的嵌入式系统，其知识要素与学习方法有所不同，所以可以按应用范围简单地把嵌入式系统分为电子系统智能化（微控制器类）和计算机应用延伸（应用处理器）这两大类。一般来说，微控制器与应用处理器的主要区别在于可靠性、数据处理盘、工作频率等方面，相对应用处理器来说，微控制器的可靠性要求更高、数据处理批较小、工作频率较低。

1.电子系统智能化类

电子系统智能化类的嵌入式系统，主要用于工业控制、现代衣业、家用电器、汽车电子、测控系统、数据采集等，这类应用所使用的嵌入式处理器一般被称为微控制器

(Microcontroller Unit, MCU) 。这类嵌入式系统产品， 从形态上看，更类似千早期的电子系统，但内部计算程序起核心控制作用。

从学习与开发角度，电子系统智能化类的嵌入式应用，需

要终端产品开发者面向应用对象设计硬件、软件，注重软件、硬件协同开发。因此，开发者必须掌握底层硬件接口、底层驱动及软硬件密切结合的开发调试技能。电子系统智能化类的嵌入式系统，即微控制器，是嵌入式系统的软硬件基础， 是学习嵌入式系统的入门环节，且为重要的一环。

2.计算机应用延伸类

计算机应用延伸类的嵌入式系统，主要用千平板电脑、智能手机、电视机顶盒、企业网络设备等，这类应用所使用的嵌入式处理器一般被称为应用处理器(Application Processor) 。这类嵌入式系统产品，从形态上看，更接近通用计算机系统。开发方式上，也类似于通用计算机的软件开发方式。从学习与开发角度，计算机应用延伸类的嵌入式应用，终端产品开发者大多购买厂家制作好的硬件实体在嵌入式操作系统下进行软件开发，或者还需要掌握少量的对外接口方式。因此，从知识结构角度，学习这类嵌入式系统，对硬件的要求相对较少。计算机应用延伸类的嵌入式系统， 即应用处理器， 也是嵌入式系统学习中重要的一环。但是，从学习规律角度看，若是要全面学习掌握嵌入式系统，应该先学习掌握微控制器，然后在此基础上，进一步学习掌握应用处理器编程，而不要倒过来学习。

## 1.1.4 嵌入式系统的特点

1. 嵌入式系统属于计算机系统，但不单独以通用计算机的面目出现

嵌入式系统的本名叫嵌入式计算机系统(Embedded Computer System), 它不仅具有通

用计算机的主要特点，又具有自身特点。嵌入式系统也必须要有软件才能运行，但其隐含在种类众多的具体产品中。同时，通用计算机种类屈指可数，而嵌入式系统不仅芯片种类繁多，而且由于应用对象大小各异，嵌入式系统作为控制核心，已经融入到各个行业的产品之中。

2. 嵌入式系统开发需要专用工具和特殊方法嵌入式系统不像通用计算机那样有了计算机系统就可以进行应用软件的开发。一般情况下，微控制器或应用处理器芯片本身不具备开发功能，必须要有一套与相应芯片配套的开发工具和开发环境。这些工具和环境一般基于通用计算机上的软硬件设备以及逻辑分析仪、示波器等。开发过程中往往有工具机（ 一般为PC 或笔记本）和目标机（实际产品所使用的芯片）之分，工具机用于程序的开发，目标机作为程序的执行机，开发时需要交替结合进行。编辑、编译、链接生成机器码在工具机完成，通过写入调试器将机器码下载到目标机中，进行运行与调试。

3. 使用MCU 设计嵌入式系统，数据与程序空间采用不同存储介质在通用计算机系统中， 程序存储在硬盘上。实际运行时，通过操作系统将要运行的程序从硬盘调入内存(RAM), 运行中的程序、常数、变最均在RAM 中。而在以MCU 为核心的嵌入式系统中， 一般情况下，其程序被固化到非易失性存储器中气变量及堆栈使用RAM存储器。

4. 开发嵌入式系统涉及软件、硬件及应用领域的知识嵌入式系统与硬件紧密相关，嵌入式系统的开发需要硬件、软件协同设计、协同测试。同时，由于嵌入式系统专用性很强，通常是用在特定应用领域，如嵌入在手机、冰箱、空调、各种机械设备、智能仪器仪表中起核心控制作用，功能专用，因此，进行嵌入式系统的开发，还需要对领域知识有一定的理解。

## 1.2 嵌入式系统的学习困惑，知识体系和学习建议

## 1.2.1 嵌入式系统的学习困惑

## 1.2.2 嵌入式系统的知识体系

从由浅入深、由简到繁的学习规律来说，嵌入式学习的入门应该选择微控制器，而不是应用处理器，应通过对微控制器基本原理与应用的学习，逐步掌握嵌入式系统的软件与硬件基础，然后在此基础上进行嵌入式系统其他方面知识的学习。

概括地说，学习以MCU 为核心的嵌入式系统，需要以下软件硬件基础知识与实践训

练，即以MCU 为核心嵌入式系统的基本知识体系如下觅

(1) 掌握硬件最小系统与软件最小系统框架。硬件最小系统是包括电源、晶振、复位、写入调试器接口等可使内部程序得以运行的、规范的、可复用的核心构件系统气软件最小系统框架是一个能够点亮一个发光二极管的，甚至带有串口调试构件的，包含工程规范完整要素的可移植与可复用的工程模板

(2) 掌握常用基本输出的概念、知识要素、构件使用方法及构件设计方法。如通用I/0

CG PIO) 、模数转换AD 、数模转换DA 、定时器模块等。

(3) 掌握若干嵌入式通信的概念、知识要素、构件使用方法及构件设计方法。如串行通信接口UART 、串行外设接口SPI , 集成电路互连总线IZC 、CAN 、USB 、嵌入式以太网、无线射频通信等。

(4) 掌握常用应用模块的构件设计方法及使用方法及数据处理方法。如显示模块

(LED 、LCD 、触摸屏等）、控制模块（控制各种设备，包括PWM 等控制技术）等。数据处理如图形、图像、语音、视频等处理或识别等。

(5) 掌握一门实时操作系统RTOS 的基本用法与基本原理。

(6) 掌握嵌入式软硬件的基本调试方法。如断点调试、打桩调试、printf 调试方法等。

在嵌入式调试过程中，特别要注意确保在正确硬件环境下调试未知软件，在正确软件环境下调试未知硬件。

## 1.2.3 基础阶段的学习建议

(1) 遵循“先易后难，由浅入深＂的原则，打好软硬件基础。跟随本书，充分利用本书提

供的软硬件资源及辅助视频材料，逐步实验与实践矶充分理解硬件基本原理、掌握功能模

块的知识要素、掌握底层驱动构件的使用方法、掌握一两个底层驱动构件的设计过程与方

法；熟练掌握在底层驱动构件基础上，利用C 语言编程实践。理解学习嵌入式系统， 必须勤千实践。

( 2) 充分理解知识要素、掌握底层驱动构件的使用方法。本书对诸如GPIO 、UART 、定时器、PWM 、AD 、DA 、Flash 在线编程、USB 等模块，首先阐述其通用知识要素，随后给出其底层驱动构件的基本内容。期望读者在充分理解通用知识要素基础上，学会底层驱动构件使用方法。

(3) 基本掌握底层驱动构件的设计方法。

(4)掌握单步跟踪调试、打桩调试、printf 输出调试等调试手段。

(5) 日积月累，勤学好问，充分利用本书及相关资源。

## 1.3 微控制器和应用处理器简介

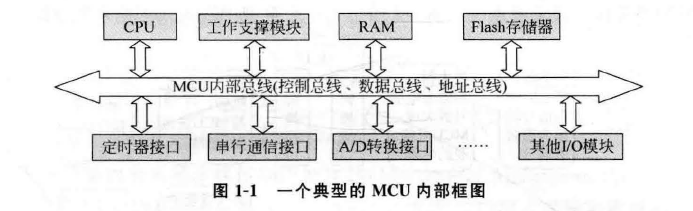
## 1.3.1 微控制器简介

1、微控制器的基本含义

MCU 是单片微观计算机（单片机）的简称，早期的英文名是Sing l e-chi p

Microcomputer, 后来大多数称之为微控制器(Mi crocon troll er) 或嵌入式计算机(EmbeddedComp uter) 。

MCU的基本含义是：在一块芯片内集成了中央处理单元( Central Processing Unit, CPU存储器(RAM/ROM 等）、定时器／计数器及多种输入输出(1/0) 接口的比较完整的数字处~系统。



MCU 是在计算机制造技术发展到一定阶段的背景下出现的，它使计算机技术从科学

计算领域进入到智能化控制领域。从此，计算机技术在两个重要领域一通用计算机领域

和嵌人式(Emb edded) 计算机领域都获得了极其重要的发展，为计算机的应用开辟了更广阔的空间。

2. 嵌入式系统与MCU 的关系

大部分嵌入式系统以MCU 为核心进行设计。MCU 从体系结构到指令系统都是按照嵌入式系统的应用特点专门设计的，它能很好地满足应用系统的嵌入、面向测控对象、现场可靠运行等方面的要求。因此以MCU 为核心的系统是应用最广的嵌入式系统

## 1.3.2 以MCU 为核心的嵌入式测控产品的基本组成

一个以M<::U 为核心，比较复杂的嵌入式产品或实际嵌入式应用系统，包含模拟量的输入、模拟量的输出、开关量的输入、开关量的输出及数据通信的部分

1. MCU 工作支撑电路

MCU 工作支撑电路也就是MCU 硬件最小系统，它保障MCU 能正常运行，如电源电

路、晶振电路及必要的滤波电路等， 甚至可包含程序写入器接口电路。

2. 模拟信号检入电路

实际模拟信号一般来自相应的传感器

3. 开关量信号输入电路

实际开关信号一般也来自相应的开关类传感器，如光电开关、电磁开关、干簧管（磁开

关）、声控开关、红外开关等， 一些儿童电子玩具中就有一些类似的开关。手动开关也可作为开关信号送到MCU 中。对MCU 来说，开关信号就是只有"O" 和"1" 两种可能值的数字

信号。

4. 其他输入信号或通信电路

其他输入信号通过某些通信方式与MCU 沟通。常用的通信方式有异步串行(UART)通信、串行外设接口(SPD 通信、并行通信、USB 通信、网络通信等。

5. 捈出执行机构电路

在执行机构中，有开关量执行机构，也有模拟量执行机构。开关量执行机构只有“开““关”两种状态。模拟量执行机构需要连续变化的模拟最控制。MCU 一般不能直接控制这些执行机构，需要通过相应的隔离和驱动电路实现。还有一些执行机构，既不是通常开关量控制，也不是通常DA 转换量控制，而是“脉冲“最控制，如控制调频电动机， MCU 则通过软件对其控制。

## 1.3.3 应用处理器简介

应用处理器的全名是多媒体应用处理器(Multimedia Application Processor, MAP) 。它是在低功耗CPU 的基础上扩展音视频功能和专用接口的超大规模集成电路。与MCU相比， MAP 的最主要特点是：工作频率高；硬件设计更为复杂；软件开发需要选用一个嵌入式操作系统；计算功能更强；抗干扰性能较弱；较少直接应用千控制目标对象；此外，一般情况下， MAP 芯片价格也高千MCU 。

## 1.4 嵌入式系统常用术语

1、封装

集成电路的封装( Package) 是指用塑料、金属或陶瓷材料等把集成电路封在其中。封装可以保护芯片， 并使芯片与外部世界连接。常用的封装形式可分为通孔封装和贴片封装两大类。

2. 印刷电路板

印刷电路板(Printed C汀cu it Board, PCB) 是组装电子元件用的基板，是在通用基材上按

预定设计形成点间连接及印制元件的印制板，是电路原理图的实物化。PCB 的主要功能是

提供集成电路等各种电子元器件固定、装配的机械支撑；实现集成电路等各种电子元器件

之间的布线和电气连接（信号传输）或电绝缘；为自动装配提供阻焊图形，为元器件插装、检查、维修提供识别字符和图形等。

3. 动态可读写随机存储器与静态可读写随机存储器

动态可读写随机存储器(Dynamic Random Access Memory, DRAM), 由一个MOS 管组成一个二进制存储位。MOS 管的放电导致表示"1" 的电压会慢慢降低。一般每隔一段时间就要控制刷新信息，给其充电。DRAM 价格低，但控制烦琐，接口复杂。

4.只读存储器

只读存储器(Read Only Memory,ROM), 数据可以读出，但不可以修改，所以称为只读存储器。通常存储一些固定不变的信息，如常数、数据、换码表、程序等。它具有断电后数据不丢失的特点。ROM 有固定ROM 、可编程ROM( 即PROM) 和可擦除ROM( 即EPROM)三种。

5.闪速存储器

闪速存储器(Fl as h M emory) 简称闪存，是一种新型快速的E2 PROM 。由于工艺和结构上的改进，闪存比普通的E2 PROM 的擦除速度更快，集成度更高。闪存相对千传统的E2 PROM 来说，其最大的优点是系统内编程，也就是说不需要另外的器件来修改内容

6.模拟量和开关量

模拟最是指时间连续、数值也连续的物理量，如温度、压力、流量、速度、声音等。在工程技术上，为了便千分析，常用传感器、变换器将模拟蜇转换为电流、电压或电阻等电学量。开关量是指一种二值信号，用两个电平（高电平和低电平）分别来表示两个逻辑值（逻辑1 和逻辑0) 。

## 1.4.2 与通信相关的术语

1. 并行通信

并行通信是指数据的各位同时在多根并行数据线上进行传输的通信方式，数据的各位同时由源到达目的地。适合近距离、高速通信。常用的有4 位、8 位、16 位、32 位等同时传输

2. 串行通信

串行通信是指数据在单线（电平高低表征信号）或双线（差分信号）上，按时间先后一位一位地传送，其优点是节省传输线，但相对于并行通信来说，速度较慢。

3. 串行外设接口

串行外设接口(Seri a l Peripheral Interface, SPI) 也是一种串行通信方式，主要用于MCU 扩展外围芯片使用。这些芯片可以是具有SPI 接口的AD 转换、时钟芯片等。

4. 集成电路互连总线

集成电路互连总线(Inter-Integrated C订cuit, I 2C) 是一种由PHILIPS 公司开发的两线式串行总线，有的书籍也记为IIC 或12 C, 主要用于用户电路板内MCU 与其外围电路的连接。

5. 通用串行总线

通用串行总线(Universal Serial Bus, USB) 是MCU 与外界进行数据通信的一种新的方式，其速度快，抗干扰能力强，在嵌入式系统中得到了广泛的应用。USB 不仅成为通用计算机上最重要的通信接口，也是手机、家电等嵌入式产品的重要通信接口。

6. 控制器局域网

控制器局域网(Controller Area Network ,CAN) 是一种全数字、全开放的现场总线控制网络，目前在汽车电子中应用最广。

7. 背景调试模式

背景调试模式(Background Debug Mode, BDM) 是Freescale 半导体公司提出的一种调试接口，主要用于嵌入式MCU 的程序下载与程序调试。

8. 边界扫描测试协议

边界扫描测试协议(Joint Test Action Group, JTAG) 是由国际联合测试行动组开发，对芯片进行测试的一种方式，可将其用于对MCU 的程序进行载入与调试。JTAG 能获取芯片寄存器等内容，或者测试遵守IEEE 规范的器件之间引脚连接情况。

9. 串行线调试技术

串行线调试(Seria l Wire Debug, SWD) 技术使用2 针调试端口， 是JTAG 的低针数和高性能替代产品，通常用于小封装微控制器的程序写入与调试。SWD 适用千所有ARM 处理器，兼容JTAG 。

## 1.4.3 与功能模块相关的术语

1. 通用捡入／ 给出

通用输入／输出(General Purpose I/ 0,GPIO), 即基本的输入／输出，有时也称并行I/0 。作为通用输入引脚时， MCU 内部程序可以读取该引脚，知道该引脚是"l" ( 高电平）或" O "（低电平），即开关橄输入。作为通用输出引脚时， MCU 内部程序向该引脚输出"1" (高电平）或"O"( 低电平），即开关量输出。

2. 模数转换与数模转换

模数转换(ADC) 的功能是将电压信号（模拟量） 转换为对应的数字量。实际应用中，这个电压信号可能由温度、湿度、压力等实际物理量经过传感器和相应的变换电路转化而来。经过AD 转换， MCU 就可以处理这些物理最。而与之相反，数模转换CDAC) 的功能则是将数字最转换为电压信号（模拟拯）。

3. 脉冲宽度调制器脉冲宽度调制器(Pulse Width Modulator,PWM), 是一个DA 转换器，可以产生一个高电平和低电平之间重复交替的输出信号，这个信号就是PWM 信号。

4. 看门狗

看门狗( Watch Dog), 是一种为了防止程序跑飞而设计的自动定时器。当程序跑飞时，由千无法正常执行清除看门狗定时器，看门狗定时器会自动溢出，使系统程序复位。

5. 液晶显示

液晶显示(Liquid Crystal Dispaly, LCD), 是电子信息产品的一种显示器件，可分为字段型、点阵字符型、点阵图形型三类。

6. 发光二极管

发光二极管(Light Emitting Diode,LED), 是一种将电流顺向通到半导体PN 结处而发光的器件。常用于家电指示灯、汽车灯和交通警示灯。

7. 键盘

键盘是嵌入式系统中最常见的输入设备。

1.5.1、C语言的运算符与数据类型

见P20

第二章 ARM Cortex-M0+处理器

## 2.1 ARM Cortex M0+处理器简介

Cortex-MO 十处理器不仅延续了易用性、C 语言编程模型等优势，而且能够兼容巳有的Cortex- MO 处理器的工具。作为Cortex-M 处理器系列中的一员， Cortex-MO 十处理器同样可获得ARM Cortex-M 整个系统的全面支持，其良好的软件兼容性，使其能够方便地被移植到更高性能的Cortex-M3 或Cortex-M4 等系列处理器。

## 2.1.1 ARM Cortex-MO 十处理器内部结构概要

1.M0+内核

32 位ARM Cortex-MO 十处理器是以一个“处理器子系统”的形式出现的，其CPU 内核本身与N VIC 和一系列调试块都亲密耦合。Cortex-MO 十处理器基于2 级流水线冯· 诺依曼架构CD , 内核版本为ARMv6-M, 支持16 位Thumb 指令集，同时采用Thumb2 技术。MO + 内核的性能是接近8 位或16 位竞争产品CoreMark/mA® 的二倍。

2.嵌套中断向量控制器

3.总线网络

4.调试组件

MO 十处理器实现了一个完全基于硬件的调试解决方案。该调试方案通过2 针脚串行线协议(SWD) 访问处理器、内存和外设

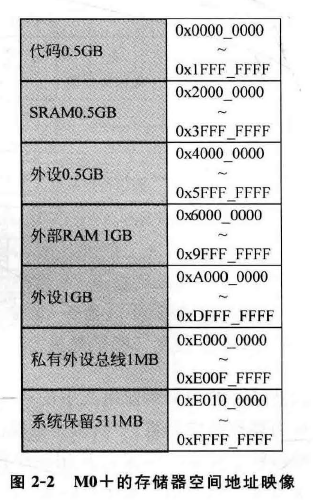
5.总线接口

CMO 十处理器提供一个基千被称之为高级微控制器总线体系结构的总线规范AMBA技术的单一32 位系统接口，可以高速整体访问所有系统外设和内存。

6.其他模块

## 2.1.2 ARM Cortex M0+处理器映像

CMO+ 处理器直接寻址空间为4GB, 地址范围是： OxOOOO\_OOOO ~ OxFFFF FFFF 。这里所说的存储器映像，其含义是指，把这4GB 空间当作存储器来看待，分成若干区间，都可安排一些什么实际的物理资源



## 2. 1. 3 ARM Cortex-MO 十处理器的寄存器

1. 通用寄存器RO~Rl2RO ~ Rl 2 是最具“通用目的＂的32 位通用寄存器，用千数据操作，复位后初始值为随机值。

2. 堆栈指针Rl3

Rl3 是堆栈指针。在CMO 十处理器内核中共有两个堆栈指针， 主堆栈指针MSP 和进程堆栈指针PSP, 若用户用到其中一个， 另一个必须用特殊指令来访问(MRS 、MSR 指令），因此任一时刻只能使用其中的一个

3. 连接寄存器R 14

当调用一个子程序时，由Rl4 存储返回地址

4. 程序计数寄存器R1 5

R15 是程序计数器(Program Counter, PC), 指向当前的程序地址。如果修改它的值，就能改变程序的执行流程（很多高级技巧隐藏其中） 。在汇编代码中也可以使用名字" PC"来访问它。

5. 特殊功能寄存器

MO + 内核包括一组特殊功能寄存器，包括程序状态字寄存器组(xPSR) 、中断屏蔽寄存器(PRIMASK) 和控制寄存器(CONTROL) 。

1) 程序状态字寄存器

程序状态字寄存器在内部分为以下几个子寄存器： APSR 、IPSR 、EPSR 。用户可以使用MRS 和MSR 指令访问寄存器。三个子寄存器既可以单独访问，也可以两个或三个组合到一起访问。使用三合一方式访问时， 把该寄存器称为xPSR

(1) 应用程序状态寄存器(Application Program Status Register, APSR) : 显示算术运算单元ALU 状态位的一些信息。

(2) 中断程序状态寄存器(Int errupt Program Status Register, IPSR) : 每次异常完成之后，处理器会实时更新IPSR 内的异常号。只

(3) 执行程序状态寄存器(Execution Program Status Register, EPSR): T 标志位指示当前运行的是否是Thumb 指令，该位是不能被软件读取的。运行复位向量对应的代码时置1 。如果该位为o, 会发生硬件异常，进入硬件中断服务例程。

2) 中断屏蔽寄存器

中断屏蔽寄存器PRIMASK 的D31~01 位保留，只有DO 位（记为PM) 有意义。当该位被置位时，除不可屏蔽中断和硬件错误之外的所有中断都被屏蔽

3) 控制寄存器

内核中的控制寄存器CONTROL 的D31 ~ 02 位保留， Dl 、DO 位含义如下。

Dl(SPSEL) -—堆栈指针选择位

DO(nPRIV) 一一－如果权限扩展，在线程模式下定义执行特权：

## 2.2 ARM Cortex-M0 +处理器指令系统

CPU 的功能是从外部设备获得数据，通过加工、处理，再把处理结果送到CP U 的外部世界。设计一个CPU, 首先需要设计一套可以执行特定功能的操作命令，这种操作命令称为指令。CPU 所能执行的各种指令的集合，称为该CPU 的指令系统

## 2.2.1 ARM Cortex-MO 十指令简表与寻址方式

1. ARM Cortex- MO 十指令简表



2. CMO 十的寻址方式

指令是对数据的操作， 通常把指令中所要操作的数据称为操作数， CMO 十处理器所需的操作数可能来自寄存器、指令代码、存储单元。而确定指令中所需操作数的各种方法称为寻址方式( Addressing Mode )

1）立即数寻址

在立即数寻址方式中，操作数直接通过指令给出，数据包含指令编码中，随着指令一起被编译成机器码存储于程序空间中。

2) 寄存器寻址

在寄存器寻址中，操作数来自千寄存器。

3) 偏移寻址及寄存器间接寻址

在偏移寻址中，操作数来自于存储单元，指令中通过寄存器及偏移量给出存储单元的地址。偏移址不超过4KB( 指令编码中偏移蜇为12 位）。偏移量为0 的偏移寻址也称为寄存器间接寻址。

4) 直接寻址

在直接寻址方式中，操作数来自于存储单元，指令中直接给出存储单元地址。

## 2.2.2 数据传送类指令

数据传送类指令的功能有两种情况， 一是取存储器地址空间中的数传送到寄存器中，二是将寄存器中的数传送到另一寄存器或存储器地址空间中。MO 十数据传送类基本指令有16 条。

1.取数指令

存储器中内容加载到寄存器中的指令见表2-4, 其中， LDR 、LDRH 、LDRB 指令分别表示加载来自存储器单元的一个字、半字和单字节（不足部分以0 填充）。LDRSH 和LDRSB指令指加载存储单元的半字、字节有符号数扩展成32 位到指定寄存器Rt 。

2. 存数指令

寄存器中内容存储至存储器中的指令见表2-5 。STR 、STRH 和STRB 指令存储Rt 寄存器中的字、低半字或低字节至存储器单元。存储器单元地址由Rn 与Rm 之和决定。Rt 、Rn 和Rm 必须为RO ~ R7 之一。

3. 寄存器间数据传送指令

4. 堆栈操作指令

堆栈操作指令见表2-7 。PUSH 指令将寄存器值存于堆栈中，最低编号寄存器使用最低存储地址空间， 最高编号寄存器使用最高存储地址空间； POP 指令将值从堆栈中弹回寄存器， 最低编号寄存器使用最低存储地址空间， 最高编号寄存器使用最高存储地址空间

## 2.2.3 数据操作类指令

1.算术运算类指令

2.逻辑运算类指令

3.移位类指令

4.位测试指令

5.数据序转之令

6.扩展类指令

2.2.4 跳转控制类指令

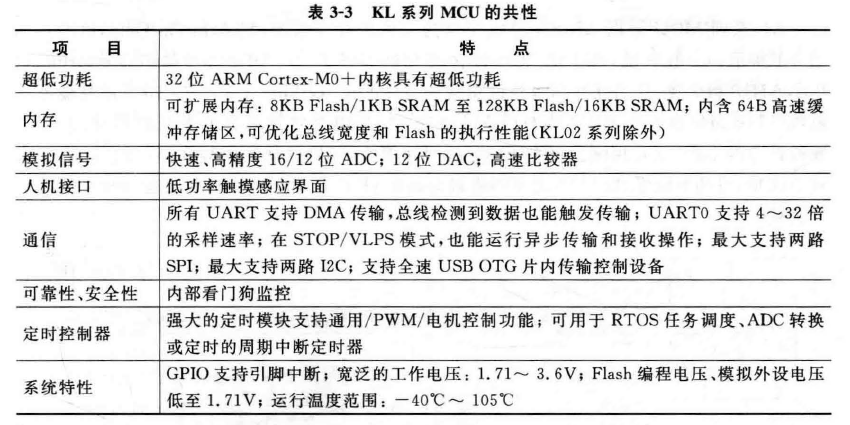
2.2.5 其他类指令

第三章 存储映像、中断源与硬件最小系统

## 3.2 KL系列MCU简介与体系结构概述

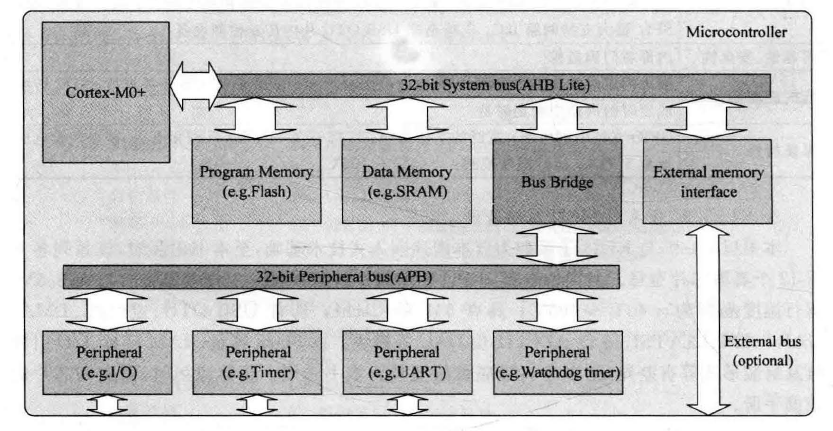
KL 系列MCU 千2012 年6 月提供样片， 2013 年正式上市。该系列MCU 是业内首款基于ARM Cortex-MO+ 内核的MCU, 具有超低功耗、应用设计方便、扩展性好、系列品种齐全等特点。目标市场是传统8 位MCU 应用领域的32 位升级换代

KL 系列MCU 在内核、低功耗、存储器、模拟信号、人机接口、安全性、定时器及系统特性等方面具有一些共同特点，简要总结



## 3. 2. 2 KL 系列MCU 体系结构概述

KL 系列MCU 是以AMBA 总线规范为架构的片上系统(System On Chip, SOC) , 如图3-1 所示。一般来说， AMBA 架构包含高性能系统总线( Advanced High PerformanceBus,AHB) 和低速、低功耗的高级外设总线( Adva nced PeriPheral Bus, APB) 。高性能系统总线AHB 是负责连接ARM 内核、DMA 控制器、片内存储器或其他需要高带宽的模块。而外设总线APB 则是用来连接系统的外围慢速模块，其协议规则相对系统总线AHB 来说较为简单，它与系统总线AHB 之间则通过总线桥( Bus Bridge ) 相连， 期望能减少系统总线的负载。



1.AMBA总线规范

ARM 公司定义了AMBA (Advanced Microcontroller Bus Arch i tecture) 总线规范，它

是一组针对基千ARM 内核、片内系统之间通信而设计的标准协议。在AMBA 总线规范

中，定义三种总线，分别是：1）高性能总线 2）高级系统总线 3）高级外设总线

2.总线桥

外设桥的作用是把交叉开关(CrossbarSwitch) 接口协议，转换成私有外设总线协议CIPS/ APB) 。本书中MCU 外设桥以外设槽(Slot) 形式，最多可连接128 个外设，给每个外设分配4KB 的寄存器映像空间。外设桥为每个外设槽提供了独立时钟，以便更好地支持低速外设。

3.交叉开关

交叉开关(Crossbar Switch) 将总线主机与总线从机相连，该结构允许多达4 路主机同时访问不同总线从机。所谓总线主机是指其可在总线上产生与控制所有时序

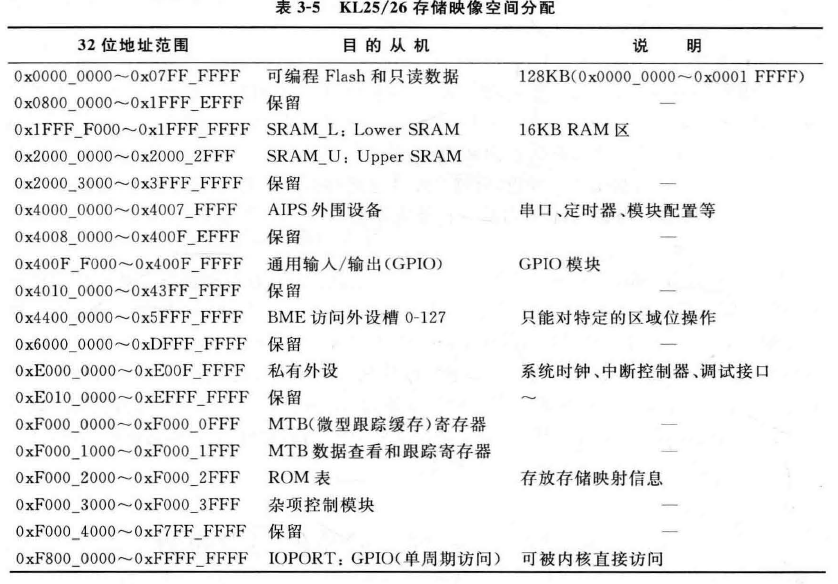
## 3.3 KL25/26 系列存储映像与中断源

## 3.3.1 KL25/26系列存储映像

所谓存储映像(Memory Mapping) 在这里可以直观地理解为， MO + 寻址的4GB 地址空

间(OxOOOO\_OOOO ~ OxFFFF\_FFFF严被如何使用，都对应了哪些实际的物理介质。为区别于CPU 内部寄存器， GPIO 寄存器也被称为＂映像寄存器"(MappingRegister), 相对应的地址被称为＂映像地址" (Mapping Address) 。整个可直接寻址的空间被称为＂映像地址空间"(Mapping Address Space) 。

KL25 /2 6 把MO + 内核之外的模块，用类似存储器编址的方式，统一分配地址。在4GB的映像地址空间内，分布着片内F lash 、SRAM 、系统配置寄存器以及其他外设等，以便CPU通过直接地址进行访问



1. 片内Fl as h 区存储映像

KL25/26 片内Flash 大小为128KB, 地址范围是： OxOOOO\_OOOO ~ OxOOOI\_ FFFF, 一般被用来存放中断向量、程序代码、常数等，其中前1928 为中断向量表。

2. 片内RAM 区存储映像

KL25/26 片内RAM 为静态随机存储器SRAM, 大小为16KB, 地址范围是： OxlFFF\_FOOO~Ox2000\_2FFF, 一般被用来存储全局变量、静态变量、临时变量（堆栈空间）等。

3. 其他存储映像

其他存储映像，如外设区存储映像（外设桥、GPIO 、位操作引擎等）、私有外设总线存储映像系统保留段存储映像等

## 3. 3. 2 KL25/26 中断源

中断是计算机发展中一个重要的技术，它的出现很大程度上解放了处理器，提高了处理器的执行效率。所谓中断，是指MCU 在正常运行程序时，由于MCU 内核异常或者MCU各模块发出请求事件，引起MCU 停止正在运行的程序，而转去处理异常或执行处理外部事件的程序（又称中断服务程序）。

KL25 / 26 的中断源分为两类，如表3-6 所示，

一类是内核中断，另一类是非内核中断。内核中断主要是异常中断， 也就是说，当出现错误的时候，这些中断会复位芯片或是做出其他处理。非内核中断是指MCU 各个模块被中断源引起的中断， MCU 执行完中断服务程序后，又回到刚才正在执行的程序，从停止的位置继续执行后续的指令。非内核中断又称可屏蔽中断，这类中断可以通过编程控制，开启或关闭该中断。



## 3. 4. 1 硬件最小系统引脚

KL2 5 / 26 硬件最小系统引脚是我们需要为芯片提供服务的引脚，包括电源类引脚、复位引脚、晶振引脚等

## 3.4 KL25/26的引脚功能

每个引脚都可能有多个复用功能，有的引脚有两个复用功能，有的有4 个复用功能，实际嵌入式产品的硬件系统设计时必须注意只能使用其中的一个功能。进行硬件最小系统设计时， 一般以引脚的第一功能作为引脚名进行原理图设计，若实际使用的是其另一功能，可以用括号加以标注，这样设计的硬件最小系统就比较通用。下面从需求与供给的角度把MCU 的引脚分为“硬件最小系统引脚”与"I/0 端口资源类引脚”两大类。

## 3.4.2 对外提供服务的引脚

除了需要我们为芯片服务的引脚（最小硬件系统引脚）之外，芯片的其他引脚为我们提供服务， 也可称之为I / 0 端口资源类引脚



## 3. 5 KL25/26 硬件最小系统原理图

MCU 的硬件最小系统是指包括电源、晶振复位、写入调试器接口等可使内部程序得以运行的、规范的、可复用的核心构件系统

## 3. 5. 1 电源及其滤波电路

电路中需要大量的电源类引脚用来提供足够的电流容量同时保持芯片电流平衡，所有的电源引脚必须外接适当的滤波电容抑制高频噪声。

## 3.5.2 复位电路及复位功能

复位，意味着MCU 一切重新开始。复位引脚为RESET 。若复位引脚有效（低电平） ，则会引起MCU 复位。复位电路原理如下： 正常工作时，复位引脚RESET 通过一个lOk.O.的电阻接到电源正极， 所以应为高电平。若按下复位按钮，则RESET 脚接地为低电平， 导致芯片复位。若是系统重新上电，芯片内部电路会使RESET 脚拉低，使芯片复位

从引起MCU 复位的内部与外部因素来区分，复位可分为外部复位和内部复位两种。

从复位时芯片是否处千上电状态来区分， 复位可分为冷复位和热复位。芯片从无电状态到上电状态的复位属于冷复位，芯片处于带电状态时的复位叫热复位。冷复位后， MCU内部 RAM 的内容是随机的。而热复位后， MCU 内部RAM 的内容会保持复位前的内容，即热复位并不会引起RAM 中内容的丢失。

从CPU 响应快慢来区分， 复位还可分为异步复位与同步复位。异步复位源的复位请求一般表示一种紧要的事件，因此复位控制逻辑不等到当前总线周期结束， 复位立即有效。异步复位源有上电、低电压复位等。同步复位的处理方法与异步复位不同： 当一个同步复位源给出复位请求时， 复位控制器并不使之立即起作用，而是等到当前总线周期结束之后，这是为了保护数据的完整性。

第四章 GPIO及程序框架

## 4.1 通用I/0 接口基本概念及连接方法

1. I / 0 接口的概念

I/0 接口，即输入／输出(Input/ Output) 接口， 是MCU 同外界进行交互的重要通道，MCU 与外部设备的数据交换通过1/ 0 接口来实现。I/ 0 接口是一个电子电路，其内由若干专用寄存器和相应的控制逻辑电路构成。

有时把1/ 0 引脚称为接口(Interface) , 而把用千对I / 0 引脚进行编程的寄存器称为端口(Port), 实际上它们是紧密相连的。因此，不必深究它们之间的区别。有些书中甚至直接称1/ 0 接口（端口）为I/ 0口。在嵌人式系统中，接口千变万化，种类繁多，有显而易见的人机交互接口，如操纵杆、键盘、显示器；也有无人介入的接口，如网络接口、机器设备接口等。

2. 通用I/ 0

所谓通用1/0, 也记为GPIO(General Purpose 1/0), 即基本的输入／输出，有时也称并行I/0, 或普通I/ 0, 它是1/ 0 的最基本形式。本书中使用正逻辑，电源(Vee) 代表高电平，对应数字信号"l" ; 地(GND) 代表低电平，对应数字信号"O" 。作为通用输入引脚， MCU 内部程序可以通过端口寄存器获取该引脚状态，以确定该引脚是"l" (高电平）或"O"( 低电平），即开关撮输入。作为通用输出引脚， MCU 内部程序通过端口寄存器控制该引脚状态，使得引脚输出"l"( 高电平）或"O"( 低电平），即开关筐输出。大多数通用I/ 0 引脚可以通过编程来设定其工作方式为输入或输出，称为双向通用I/ 0 。

3. 上拉下拉电阻与输入引脚的基本接法

芯片输入引脚的外部有三种不同的连接方式：带上拉电阻的连接、带下拉电阻的连接和“悬空“连接。通俗地说，若MCU 的某个引脚通过一个电阻接到电源(Vee ) 上，这个电阻被称为“上拉电阻＂；与之相对应，若MCU 的某个引脚通过一个电阻接到地(GND) 上，则相应的电阻被称为“下拉电阻＂ 。这种做法使得悬空的芯片引脚被上拉电阻或下拉电阻初始化为高电平或低电平。

4. 输出引脚的基本接法

作为通用输出引脚， MCU 内部程序向该引脚输出高电平或低电平来驱动器件工作，即开关量输出

## 4.2 端口控制模块与GPIO模块的编程结构

## 4.2.1 端口控制模块——决定引脚复用功能

KL2 5 的大部分引脚具有复用功能，可以通过端口控制模块(Port Control andInterrupts, PORT ) 提供的寄存器编程指定其为某一具体功能。

PORT 模块内含三类寄存器，分别是引脚控制寄存器(Pin Control Register入全局引脚控制寄存器(Global Pin Control Regi ster汃中断状态标志寄存器(Interrupt Status Flag Register) 。

1.寄存器映像地址分析

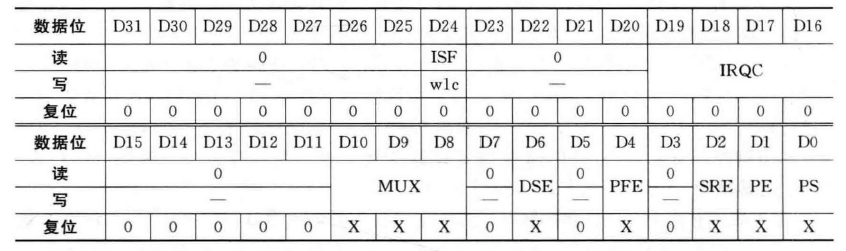
KL2 5 芯片有5 个端口A~ E 。每个端口有32 个引脚控制寄存器PORTx\_PCRn( 其中x=A~ E,n=0~31), 两个全局引脚控制寄存器(PORTx \_G PCLR 、PORTx\_GPCHR) 、一个中断状态标志寄存器(PORTx\_ ISFR) 。以下地址分析计算均为十六进制，为书写简化起见，在不致引起歧义的情况下，略去十六进制前缀"Ox" 不写。

每个端口有32 个引脚控制寄存器PORTx\_PCRn 。端口x 的基地址= 4004\_9000+ xXlOOO(x= A~ E, 对应0 ~ 4) 。端口x 的每个引脚控制寄存器PORTx\_PCRn 的地址为＝4004\_9000+xX lOOO+nX 4(x= A~E, 对应0~4, n =0~31) 。这样5 个端口，共5X32 =1 60 个引脚控制寄存器，每个引脚控制寄存器的地址很容易计算出来。例如， PORTA \_PCRl 的地址为： 4004\_9000 + 0 X 1000+ 1 X 4 = 4004\_ 9004 。

每个端口有两个全局引脚控制寄存器。全局引脚控制寄存器（低） PORTx\_GPCLR, 地址= 4004 \_ 9080 + x X 1000 (x= A ~ E, 对应0 ~ 4) ; 全局引脚控制寄存器（高） PORTx\_GPCHR, 地址=4004\_9084+ x X l OOO(x=A~E, 对应0~4) 。每个端口有一个中断状态标志寄存器。地址=4004\_90 AO+ xX 1000(x= A~ E, 对应0~4) 。

2.引脚控制寄存器

每个端口的每个引脚均有一个对应的引脚控制寄存器(PORTx\_ PCRn ), 可以配置引脚中断或DMA 传输请求，可以配置引脚为GPIO 功能或其他功能，可以配置是否启用上拉或下拉，可以配置选择输出引脚的驱动强度，可以配置选择输入引脚是否使用内部滤波等。



其中， "X"表示复位后状态不确定。下面给出有关功能说明，未说明的位或字段均为保留（只读，值为0) 。

D24CISF)—中断状态标志（只读） 。数字引脚模式下有效。ISF = O, 未检测到引脚中

断； ISF= 1 , 检测到引脚中断。向该位写1 , 可清除中断状态标志

D1 9 ~ D16CIRQC) 一— 中断配置情况（读／写） 。数字引脚模式下有效。IRQC = O O O O ,关闭引脚中断/DMA 请求； IRQC = OOOl ~ OOll ——分别对应上升沿、下降沿、沿跳变，触发DMA 请求； 0100一保留； 1000 ~ 1100-分别对应逻辑低电平（逻辑0 ) 、上升沿、下降沿、沿跳变、高电平（逻辑1)' 触发引脚中断。其他值一保留。特别注意：井不是所有KL25 的引脚均可配置为中断功能，只有A 口、D 口的引脚具有上述这种中断功能。

D6(DSE)一驱动能力使能位（读／写）。表明引脚被配置为数字输出时的驱动能力状况，数字引脚模式下有效。

D4(PFE)一无源滤波使能位（读／ 写） 。数字引脚模式下有效。PFE =O , 相应的引脚

禁止无源输入滤波； PFE= l, 相应的引脚启用无源输入滤波

DZ(SRE)一转换速率使能位（读／写）。数字引脚模式下有效。0一—引脚配置成快转换速率； 1一—引脚配置成慢转换速率。

DlCPE) -—上拉或下拉使能位（读／写）。数字引脚模式下有效。0——相应的引脚关闭内部上拉或下拉电阻； 1-一相应的引脚启用内部上拉或下拉电阻，引脚作为数字输入。

DO(PS) -—－上拉或下拉选择（ 读／ 写）。数字引脚模式下有效。PS =O , 如果PE = l, 引脚下拉电阻使能； PS= l, 如果PE = l, 引脚上拉电阻使能。KL2 5 内部上下拉电阻大小为20~50k0 。

3. 全局引脚控制寄存器

4. 中断状态标志寄存器

## 4. 2. 2 GPIO 模块一—对外引脚与内部寄存器

1. KL25 的GPIO 引脚

KL25 的大部分引脚具有多重复用功能，可以通过给出的寄存器编程来设定使用其中某一种功能。80 引脚封装的KL25 芯片的GPIO 引脚分为5 个端口，标记为A 、B 、C 、D 、E , 共含61 个引脚。端口作为GPIO 引脚时，逻辑1 对应高电平，逻辑0 对应低电平

2. GPIO寄存器

每个GPIO 口均有6 个寄存器， 5 个GPIO 口共有30 个寄存器。A 、B 、C 、D 、E 各口寄存器的基地址分别为400F\_FOOOh 、400F\_F040h 、400F\_F0080h 、400F\_FOCOh 、400F\_Fl OOh ,所以各口基地址相差40h 。各GPIO 口的6 个寄存器分别是数据输出寄存器、输出置1 寄存器、输出清0 寄存器、输出反转寄存器、数据输入寄存器、数据方向寄存器其中，输出寄存器的地址就是口的基地址，其他寄存器的地址依次加4 。



## 4.2.3 GPIO基本编程步骤和基本打通程序

1. GPIO 基本编程步骤

要使芯片某一引脚为GPIO 功能，并定义为输入／输出，随后进行应用，基本编程步骤如下。

(1) 通过端口控制模块(PORT) 的引脚控制寄存器PORTx\_ PCRn 的引脚复用控制字段(MUX) 设定其为GPI O 功能（即令MUX=ObOOl) 。

(2) 通过GPIO 模块相应口的“数据方向寄存器”来指定相应引脚为输入或输出功能。若指定位为o, 则为对应引脚输入；若指定位为I, 则为对应引脚输出。

(3) 若是输出引脚，则通过设置“数据输出寄存器”来指定相应引脚输出低电平或高电平，对应值为0 或1 。也可通过“输出置位寄存器”“输出清位寄存器”“输出取反寄存器”改变引脚状态，参见表4-1 中关于寄存器的说明。

(4) 若是输入引脚，则通过“数据输入寄存器”获得引脚的状态。若指定位为o, 表示当前该引脚上为低电平； 若为1, 则为高电平。

## 实例演示:

1、改绿灯亮，且用一个输入控制灯的亮，且进行单步调试

步骤：

1）通过端口控制模块将绿灯的引脚LIGHT\_GREEN (PTB\_NUM|18) 的全局引脚寄存器的MUX字段赋值001——设定为GPIO引脚，并指定为输出功能

(1)计算给出PORTB18的引脚控制寄存器地址

//PORTB端口的引脚控制寄存器基地址为0x4004A000u（后缀u表示无符号数）

volatile uint\_32 \*portB\_ptr = (uint\_32\*) 0x4004A000u;

//PORTB9的引脚控制寄存器地址=基地址+偏移量

volatile uint\_32 \*portB\_PCR\_18 = portB\_ptr + 18;

//PORTB端口（作为GPIO功能）的基地址为0x400FF040u

volatile uint\_32 \*gpioB\_ptr = (uint\_32\*) 0x400FF040u;

//PORTB的数据方向寄存器地址=基地址+偏移量

volatile uint\_32 \*portB\_PDDR = gpioB\_ptr + 5;

//PORTB的输出寄存器地址=基地址+偏移量

volatile uint\_32 \*portB\_PDO = gpioB\_ptr + 0;

//PORTB的输出反转寄存器地址=基地址+偏移量

volatile uint\_32 \*portB\_PTOR = gpioB\_ptr + 3;

//(4)通过PORTB的输出寄存器(给蓝色小灯的寄存器）赋初值1，保证定义为输出时为暗

\*portB\_PDO |= (1 << 18);

//(5)通过PORTB的方向寄存器，定义PORT18引脚输出

\*portB\_PDDR |= (1 << 18);

2）制定A端口的A1为GPIO引脚并指定为输入功能，并计算相关寄存器的地址

volatile uint\_32 \*portA\_ptr = (uint\_32\*) 0x40049000u;

volatile uint\_32 \*portA\_PCR\_1 =portA\_ptr+1;//端口A引脚1的寄存器

\*portA\_PCR\_1 &= 0b11111111111111111111100011111111; //清MUX位段

\*portA\_PCR\_1 |= 0b00000000000000000000000100000000;

//PORTA的GPIO寄存器的基地址和输入寄存器的和方向寄存器的基地址

volatile uint\_32 \*gpioA\_ptr = (uint\_32\*) 0x400FF000u;

volatile uint\_32 \*portA\_PDIR =gpioA\_ptr+4 ;

volatile uint\_32 \*portA\_PDDR =gpioA\_ptr+5 ;

\*portA\_PDDR &= ~(1 << 1);

3）设定主程序当A1引脚输入为0时，灯亮，输入为1时，灯灭。

for (;;) {

if((\*portA\_PDIR) & (1<<1)){

\*portB\_PDO &= ~(1<<18);//输出寄存器的第18位控制小灯的输出——亮灭

}else{

\*portB\_PDO |= (1<<18);

}

单步调试:进入调试模式并且按F5/F6分别向前向后进行调试；

# 嵌入式手札二

## 4.3 GPIO驱动构件封装方法与驱动构件封装规范

### 4.3.1 设计GPIO 驱动构件的必要性及GPIO 驱动构件封装要点分析

#### 1.设计GPIO驱动构件的必要性

软件构件(Software Component) 技术的出现，为实现软件构件的工业化生产提供了理论与技术基石。将软件构件技术应用到嵌入式软件开发中，可以大大提高嵌入式开发的开发效率与稳定性。软件构件的封装性、可移植性与可复用性是软件构件的基本特性，采用构件技术设计软件，可以使软件具有更好地开放性、通用性和适应性。

以KL25 的GPIO 为例， 它有61 个引脚可以作为GPIO, 分布在5 个端口，不可能使用直接地址去操作相关寄存器，那样无法实现软件移植与复用。应该把对GPIO 引脚的操作封装成构件，通过函数调用与传参的方式实现对引脚的干预与状态获取，这样的软件才便于维护与移植，因此设计GPIO 驱动构件十分必要。同时，底层驱动构件的封装，也为在操作系统下对底层硬件的操作提供了基础。

#### 2. GPIO 驱动构件封装要点分析

同样以GPIO 驱动构件为例，进行封装要点分析。即分析应该设计哪几个函数及入口参数。GPIO 引脚可以被定义成输入、输出两种情况：若是输入，程序需要获得引脚的状态（逻辑1 或O); 若是输出，程序可以设置引脚状态（逻辑1 或0) 。MCU 的PORT 模块分为许多端口，每个端口有若于引脚。GPIO 驱动构件可以实现对所有GPIO 引脚统一编程。GPIO 驱动构件由gpio. h 、gpio. C 两个文件组成，如要使用GPIO 驱动构件，只需要将两个文件加入到所建工程中，由此方便了对GPIO 的编程操作。

##### 1）模块初始化（gpio\_init）

由于芯片引脚具有复用特性，应把引脚设置成GPIO 功能；同时定义成输入或输出；若是输出，还要给出初始状态。所以GPIO 模块初始化函数gpio\_init 的参数为哪个引脚、是输人还是输出、若是输出其状态是什么，函数不必有返回值。其中，引脚可用一个16 位数据描述，高8 位表示端口号，低8 位表示端口内的引脚号

void gpio\_init(uint\_16 port\_pin,uint\_8 dir,uint\_8 state)

##### 2) 设置引脚状态(gpio\_set)

对于输出，希望通过函数设置引脚是高电平（逻辑1) 还是低电平（逻辑0) 。人口参数应该是哪个引脚，输出其状态是什么，函数不必有返回值。这样设置引脚状态的函数原型可以设计为：

void gpio\_set(uint\_16 port\_pin, uint\_8 state)

##### 3) 获得引脚状态(gpio\_get)

对于输入，希望通过函数获得引脚的状态是高电平（逻辑1) 还是低电平（逻辑0), 入口参数应该是哪个引脚，函数需要返回值引脚状态。这样设置引脚状态的函数原型可以设计为：

uint\_8 gpio\_get(uint\_16 port\_pin)

##### 4 ) 引脚状态反转(void gpio\_ reve rse)

类似的分析，可以设计引脚状态反转函数的原型为：void gpio\_reverse(uint\_16 port\_pin)

##### 5 ) 引脚上下拉使能函数(void gpio\_pull)

若引脚被设置成输入， 还可以设定内部上下拉， KL2 5 内部上下拉电阻大小为20 ~50 kΩ 。引脚上下拉使能函数的原型为：void g,pio\_pull(uint\_16 port\_pin, uint\_8 pullselect)

### 4.3.2 底层驱动构件封装规范概要与构件封装的前期准备

底层驱动构件封装规范见5. 3 节，本节给出概要与前期准备，以便读者在认识第一个构件前以及在开始设计构件时，少走弯路，做出来的构件符合基本规范，便千移植、复用、交流。

#### 1. 底层驱动构件封装规范概要

##### 1) 底层驱动构件的组成、存放位置与内容

每个构件由头文件(. h) 与源文件(. c) 两个独立文件组成，放在以构件名命名的文件夹中。驱动构件头文件(. h ) 中仅包含对外接口函数的声明，是构件的使用指南. 以构件名命名。

例如， GIPO 构件命名为gpio( 使用小写，目的是与内部函数名前缀统一）。设计好的GPIO 构件存放于".. KL25 共用驱动\KL25 底层驱动构件\gpio" 文件夹中，供复制使用。基本要求是调用者只看头文件即可使用构件。对外接口函数及内部函数的实现在构件源程序文件(. c) 中。同时应注意，头文件声明对外接口函数的顺序与源程序文件实现对外接口函数的顺序应保持一致。

##### 2) 设计构件的最基本要求

这里摘要给出设计构件的最基本要求。

(1) 考虑使用与移植方便。要对构件的共性与个性进行分析，抽取出构件的属性和对

外接口函数。

(2) 要有统一、规范的编码风格与注释。主要涉及文件、函数、变械、宏及结构体类型的命名规范；涉及空格与空行、缩进、断行等的排版规范；涉及文件头、函数头、行及边等的注释规范。

(3) 宏的使用限制。宏的使用具有两面性， 有提高可维护性一面，也有降低阅读性一面，因此不要随意使用宏。

(4) 不使用全局变量。构件封装时，禁止使用全局变量。

## 4.4 实验：利用构件方法控制小灯闪烁

实例解释：

定义文件 key.h key.c 分别代表 驱动构件的头文件和源文件，下面解释

#### 1)key.h

#ifndef \_LIGHT\_H //防止重复定义（\_LIGHT\_H 开头)

#define \_LIGHT\_H

//头文件包含

#include "common.h" //包含公共要素头文件

#include "gpio.h" //用到gpio构件

**//指示灯端口及引脚定义**

#define LIGHT\_RED (PTB\_NUM|19) //红色RUN灯使用的端口号/引脚

#define LIGHT\_BLUE (PTB\_NUM|9) //蓝色RUN灯使用的端口号/引脚

#define LIGHT\_GREEN (PTB\_NUM|18) //绿色RUN灯使用的端口号/引脚

**//定义接受高低电平的输入引脚**

#define key1 (PTB\_NUM|0)

#define key2 (PTB\_NUM|2)

#define key3 (PTA\_NUM|2)

#define key4 (PTA\_NUM|5)

//灯状态宏定义（灯亮、灯暗对应的物理电平由硬件接法决定）

#define LIGHT\_ON 0 //灯亮

#define LIGHT\_OFF 1 //灯暗

//=================接口函数声明=======================================

//==========================================================

//函数名称：light\_init

//函数参数：port\_pin：(端口号)|(引脚号)（如：(PTB\_NUM)|(9) 表示为B口9号脚）

// state：设定小灯状态。由宏定义。

//函数返回：无

//功能概要：指示灯驱动初始化。

//==========================================================

void light\_init(uint\_16 port\_pin, uint\_8 state);

//==========================================================

//函数名称：light\_control

//函数参数：port\_pin：(端口号)|(引脚号)（如：(PTB\_NUM)|(9) 表示为B口9号脚）

// state：设定小灯状态。由宏定义。

//函数返回：无

//功能概要：控制指示灯亮暗。

//======================================================================

void light\_control(uint\_16 port\_pin, uint\_8 state);

//==========================================================

//函数名称：light\_change

//函数参数：port\_pin：(端口号)|(引脚号)（如：(PTB\_NUM)|(9) 表示为B口9号脚）

//函数返回：无

//功能概要：切换指示灯亮暗。

//======================================================================

void light\_change(uint\_16 port\_pin);

**//定义输入引脚key初始化函数，需要有返回值,定义为输入引脚**

void key\_init(uint\_16 port\_pin);

uint\_8 key\_get(uint\_16 port\_pin);//获取输入引脚的输入信号

#endif //防止重复定义（\_LIGHT\_H 结尾)

#### 2)key.c

//=====================================================================

//文件名称：key.c

//功能概要：小灯构件源文件

//=====================================================================

#include "key.h"

//=====================================================================

//函数名称：light\_init

//函数参数：port\_pin：(端口号)|(引脚号)（如：(PTB\_NUM)|(9) 表示为B口9号脚）

// state：设定小灯状态。由light.h中宏定义。

//函数返回：无

//功能概要：指示灯驱动初始化。

//=====================================================================

void light\_init(uint\_16 port\_pin, uint\_8 state)

{

gpio\_init(port\_pin, GPIO\_OUTPUT, state);

}

//=====================================================================

//函数名称：light\_control

//函数参数：port\_pin：(端口号)|(引脚号)（如：(PTB\_NUM)|(9) 表示为B口9号脚）

// state：设定小灯状态。由light.h中宏定义。

//函数返回：无

//功能概要：控制指示灯亮暗。

//=====================================================================

void light\_control(uint\_16 port\_pin, uint\_8 state)

{

gpio\_set(port\_pin, state);

}

//==========================================================

//函数名称：light\_change

//函数参数：port\_pin：(端口号)|(引脚号)（如：(PTB\_NUM)|(9) 表示为B口9号脚）

//函数返回：无

//功能概要：切换指示灯亮暗。

void light\_change(uint\_16 port\_pin)

{

gpio\_reverse(port\_pin);

}

void key\_init(uint\_16 port\_pin)

{

gpio\_init(port\_pin,0,1);//定义为输入引脚

}

uint\_8 key\_get(uint\_16 port\_pin)

{

return gpio\_get(port\_pin);//获得输入引脚的输入

}

#### 3)源文件main.c

注：通过B0,B2两个输入引脚分别获取输入的状态，制作两个呼吸灯，分别对应两种不同的高低组合来实现

源文件：

//说明见工程文件夹下的Doc文件夹内Readme.txt文件

//======================================================================

#include "includes.h" //包含总头文件

int main(void) {

//（1） 声明主函数使用的变量

uint\_32 mRuncount; //主循环计数器

uint\_8 mflag;

uint\_8 k1;

uint\_8 k2;//B0,B2

uint\_8 k3;

uint\_8 k4;

//定义四个状态变量负责返回输入引脚接受的输入

//（2） 关总中断

DISABLE\_INTERRUPTS;

//（3） 给有关变量赋初值

mRuncount = 0; //主循环计数器

mflag = 0; //灯控制标志

//（4） 初始化外设模块

light\_init(LIGHT\_RED, LIGHT\_OFF); //红灯初始化

light\_init(LIGHT\_BLUE, LIGHT\_OFF); //蓝灯初始化

light\_init(LIGHT\_GREEN, LIGHT\_OFF); //绿灯初始化

//初始化开关为输入引脚

key\_init(key1);

key\_init(key2);

//（5） 使能模块中断

//（6） 开总中断

ENABLE\_INTERRUPTS;

//进入主循环

//主循环开始==================================================================

for (;;) {

k1 = key\_get(key1);

k2 = key\_get(key2);

if ((k1 == 1) && (k2 == 0)) {

for (;;) {

mRuncount++; //主循环次数计数器+1

if (mRuncount >= RUN\_COUNTER\_MAX) //主循环次数计数器大于设定的宏常数

{

mRuncount = 0;

k1 = key\_get(key1);

k2 = key\_get(key2);

if ((k1 == 0) && (k2 == 1))

break;

switch (mflag) {

case 0: {

light\_change(LIGHT\_RED);

light\_control(LIGHT\_BLUE, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

mflag = 1;

break;

}

case 1: {

light\_change(LIGHT\_BLUE);

light\_control(LIGHT\_RED, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

mflag = 0;

break;

}

default: {

}

}

}

}

} else if ((k1 == 0) && (k2 == 1)) {

for (;;) {

mRuncount++; //主循环次数计数器+1

if (mRuncount >= RUN\_COUNTER\_MAX) //主循环次数计数器大于设定的宏常数

{

mRuncount = 0;

k1 = key\_get(key1);

k2 = key\_get(key2);

if ((k1 == 1) && (k2 == 0))

break;

switch (mflag) {

case 0: {

light\_change(LIGHT\_GREEN);

light\_control(LIGHT\_BLUE, LIGHT\_OFF);

light\_control(LIGHT\_RED, LIGHT\_OFF);

mflag = 1;

break;

}

case 1: {

light\_change(LIGHT\_BLUE);

light\_control(LIGHT\_RED, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

mflag = 0;

break;

}

default: {

}

}

}

}

} else {

}

}

//以下加入用户程序--------------------------------------------------------

} //主循环end\_for

//主循环结束==================================================================

#### 4) 实验分析

这次实验通过利用GPIO构件写了light.h 组件 控制小灯的亮暗

在主函数main.c中通过初始化两个引脚作为开关输入，设定了两对不同的输入状态作为开关，通过选择语句控制不同的输入状态对应不同的流水灯。

## ch 6串口通信模块

## 6.1 异步串行通信的通用基础知识

### 6.1.1 串行通信的基本概念

在计算机中，通常一个信息单位用8 位二进制表示，称为一个” 字节"(Byte) 。串行通信的特点是：数据以字节为单位，按位的顺序（例如最高位优先）从一条传输线上发送出去。

1.异步串行通信的格式

格式的空闲状态为"l"' 发送器通过发送一个" O"表示一个字节传输的开始，随后是数据位（在MCU 中一般是8 位或9 位，可以包含校验位） 。最后，发送器发送1 位或2 位的停止位，表示一个字节传送结束。若继续发送下一字节，则重新发送开始位（这就是异步的含义了），开始一个新的字节传送。若不发送新的字节， 则维持"l" 的状态，使发送数据线处千空闲。从开始位到停止位结束的时间间隔称为一字节帧(Byte Frame) 。

2.串行通信的波特率

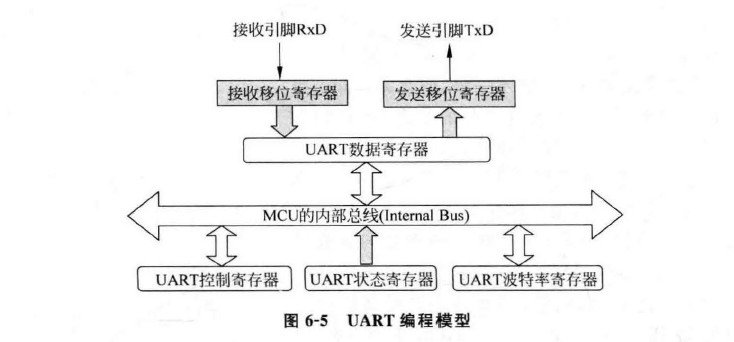
位长(Bit Length), 也称为位的持续时间(Bit Duration), 其倒数就是单位时间内传送的位数。人们把每秒内传送的位数叫作波特率(Baud Rate) 。波特率的单位是：位／ 秒，记为bps

3.奇偶校验

字符奇偶校验检查(Character~a rity Chec king) 称为垂直冗余检查( Vertical Redundancy Checking, VRC), 它是为每个字符增加一个额外位使字符中"1" 的个数为奇数或偶数

### 6.1.4 串行通信编程模型

从基本原理角度看，串行通信接口UART 的主要功能是： 接收时，把外部的单线输入的数据变成一个字节的并行数据送入MCU 内部；发送时，把需要发送的一个字节的并行数据转换为单线输出编程时，程序员并不直接与“发送移位寄存器”和“接收移位寄存器“打交道，只与数据寄存器打交道，所以MCU 中并没有设置”发送移位寄存器”和“接收移位寄存器”的映像地址。发送时，程序员通过判定状态寄存器的相应位，了解是否可以发送一个新的数据。若可以发送， 则将待发送的数据放入"UA RT 数据寄存器”中就可以了，剩下的工作由MCU 自动完成：将数据从"UART 数据寄存器”送到“发送移位寄存器＂，硬件驱动将“发送移位寄存器” 的数据一位一位地按照规定的波特率移到发送引脚TxD, 供对方接收。接收时，数据一位一位地从接收引脚RxD 进入“接收移位寄存器”，当收到一个完整字节时， MCU 会自动将数据送入"UART 数据寄存器”，并将状态寄存器的相应位改变，供程序员判定并取出数据。



## 6.2 KL25 /26 芯片UART 驱动构件及使用方法

### 6.2.2 UART 驱动构件基本要素分析与头文件

UART 驱动构件由头文件uart. h 及源代码文件uart. c 组成，放入uart 文件夹中，供应用程序开发调用。UART 具有初始化、发送和接收三种基本操作

UART 驱动构件的基本函数，与寄存器直接打交道的有：初始化、发送单个字节与接收单个字节的函数，以及使能及禁止接收中断、获取接收中断状态的函数

串口驱动构件可封装下列9 个基本功能函数

1）串口初始化

2）发送单个字节

3）发送字符串

4）接受单个字节

5）接受N个字节

6）使能串口接收中断

7）禁止串口接收中断

8）获取接收中断状态

9）发送N个字节

## 实验：通过初始化UART通信串口，完成通信

#### 1.实验准备：KL25实验板 KDS开发环境

#### 2.实验目的：

通过改写UART构件的源码来实现串口输入字符改变灯亮的颜色

#### 3.实验源码：

需要的构件一个是控制灯亮的LIGHT.H的GPIO构件，用来控制小灯的亮灭

另一个则是在isr.c中改写串口通信源码，完成串口通信的要求（其中控制小灯亮灭仍然采用上次实验的控件light）

下面给出isr.h中的源码

##### isr.h

**void** **UART2\_IRQHandler**(**void**) {

light\_init(LIGHT\_RED, LIGHT\_OFF); //红灯初始化

light\_init(LIGHT\_BLUE, LIGHT\_OFF); //蓝灯初始化

light\_init(LIGHT\_GREEN, LIGHT\_OFF);

uint\_8 ch;

uint\_8 flag;

DISABLE\_INTERRUPTS; //关总中断

**if** (uart\_get\_re\_int(UART\_2)) {

ch = uart\_re1(UART\_2, &flag); //调用接收一个字节的函数，清接收中断位

**if** (flag) {

uart\_send1(UART\_2, ch);

**switch** (ch) {

**case** 'b': {

light\_change(LIGHT\_BLUE);

light\_control(LIGHT\_RED, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

uart\_send\_string(UART\_2, "blue\_UART\_2!\r\n");

**break**; //向原串口发回一个字节

}

**case** 'r': {

light\_change(LIGHT\_RED);

light\_control(LIGHT\_BLUE, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

uart\_send\_string(UART\_2, "red\_UART\_2!\r\n");

**break**;

}

**case** 'g': {

light\_change(LIGHT\_GREEN);

light\_control(LIGHT\_BLUE, LIGHT\_OFF);

light\_control(LIGHT\_RED, LIGHT\_OFF);

uart\_send\_string(UART\_2, "green\_UART\_2!\r\n");

**break**;

}

**default**: {

}

}

}

}

ENABLE\_INTERRUPTS;

}

PS:这里是改写串口2的源码，完成输入字符 ‘b g r’ 控制灯的亮暗；

#### 4.实验分析:

这次试验只是简单的改写串口通信构件，完成输入字符控制灯的亮暗。通过这次实验，初步的认识了UART构件的作用以及怎么样使用该构件完成串口通信。但是对进一步的深层次使用还是存在很多的生疏，需要花很多时间来熟悉使用

# 嵌入式手札三

**手札说明：我听取老师上次的建议后，修改手札内容组成如下：主要内容时每次实验，将其用实验报告的形式来完成；第二部分就是书本每章提纲挈领的总结和重要概念的简单提要。**

**不再像以前那样书本内容所占篇幅过大。**

## 一：实验部分：

### 实验任务：

1）设计实验：（说明：本次实验我将老师给的实验要求改了一下，通过字符控制灯的颜色）通过串口发送程序发送字符：‘r’,‘g’,‘b’ 分别控制红绿蓝三种灯的亮）。

2）进阶实验：（说明：因为本次实验直接涉及到组帧功能，所以将设计实验中的open close一起放到进阶实验当中实现）设计createframe() 函数实现在接收中断服务例程中每次接受单字节字符输入并进行组帧，通过对每次组帧成功后保存的字符串进行判断来控制不同颜色的小灯的亮灭。

并且在C#的串口通信窗口程序中封装八个按钮控制灯的亮暗和颜色切换

### 实验环境：

MCU编程环境IDE: KDS

C#窗口程序IDE:VS2017

### 实验目的：

通过完成两个实验来学习并使用的串口通信构件的基本功能和自主添加额外的新函数实现新的功能进行组帧识别来控制不同颜色的灯的亮暗；

#### 实验任务一：

初步学习了串口通信构件中的接收中断服务例程的使用：实现MCU接受单字节字符内容的输入和发送。

#### 实验任务二：

进一步学习了如何通过使用组帧的方式来用MCU串口接受单字节字符输入并通过creatframe（）函数来组成字符串帧并进行识别判断来控制七种颜色灯切换和实现关灯控制操作；

#### 实验任务三：

将控制小灯颜色的功能封装在C#编写的串口通信窗口程序中

### 实验内容：

（实验中问题解决方法和思路的具体阐述）

#### 实验任务一：

思路：调用接收中断服务例程**void** **UART1\_IRQHandler**(**void**)；在接受每次单字节字符输入后用开关语句进行选择，控制不同颜色的灯的亮灭。

##### 源码：

//串口2接收中断服务例程

**void** **UART2\_IRQHandler**(**void**) {

uint\_8 ch;

uint\_8 flag;

DISABLE\_INTERRUPTS; //关总中断

**if** (uart\_get\_re\_int(UART\_2)) {

ch = uart\_re1(UART\_2, &flag); //调用接收一个字节的函数，清接收中断位

**if** (flag) {

uart\_send1(UART\_2, ch);

**switch** (ch) {//使用开关语句进行条件判断来控制不同的灯的亮暗；

**case** 'b': {

light\_change(LIGHT\_BLUE);

light\_control(LIGHT\_RED, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

uart\_send\_string(UART\_2, "blue\_UART\_2!\r\n");

**break**;

}

**case** 'r': {

light\_change(LIGHT\_RED);

light\_control(LIGHT\_BLUE, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

uart\_send\_string(UART\_2, "red\_UART\_2!\r\n");

**break**;

}

**case** 'g': {

light\_change(LIGHT\_GREEN);

light\_control(LIGHT\_BLUE, LIGHT\_OFF);

light\_control(LIGHT\_RED, LIGHT\_OFF);

uart\_send\_string(UART\_2, "green\_UART\_2!\r\n");

**break**;

}

**default**: {

}

}

}

}

ENABLE\_INTERRUPTS;

}

##### main.c：

//说明见工程文件夹下的Doc文件夹内Readme.txt文件

//======================================================================

#include "includes.h" //包含总头文件

int main(void) {

//1. 声明主函数使用的变量

uint\_32 mRuncount; //主循环计数器

uint\_32 mLight\_chang\_num; //灯状态变换次数

//2. 关总中断

DISABLE\_INTERRUPTS;

//3. 初始化外设模块

light\_init(LIGHT\_RED, LIGHT\_OFF); //红灯初始化

light\_init(LIGHT\_BLUE, LIGHT\_OFF); //蓝灯初始化

light\_init(LIGHT\_GREEN, LIGHT\_OFF);

uart\_init(UART\_0, 9600); //初始化串口0，波特率为9600

uart\_init(UART\_1, 9600); //初始化串口1，波特率为9600

uart\_init(UART\_2, 9600); //初始化串口2，波特率为9600

//uart\_init(UART\_Debug, 9600); //初始化UART\_Debug，波特率为9600

//串口发送初始化提示

uart\_send\_string(UART\_0, "使用uart\_send\_string发送：Hello Uart\_0!\r\n");

uart\_send\_string(UART\_1, "使用uart\_send\_string发送：Hello Uart\_1!\r\n");

uart\_send\_string(UART\_2, "使用uart\_send\_string发送：Hello Uart\_2!\r\n");

printf("使用printf函数发送：Hello Uart\_2!\r\n");

//4. 给有关变量赋初值

mRuncount = 0; //主循环计数器

mLight\_chang\_num = 0; //灯状态变换次数

//5. 使能模块中断

uart\_enable\_re\_int(UART\_0); //使能串口0接收中断

uart\_enable\_re\_int(UART\_1); //使能串口1接收中断

uart\_enable\_re\_int(UART\_2); //使能串口2接收中断

//6. 开总中断

ENABLE\_INTERRUPTS; //开总中断

//进入主循环

//主循环开始==================================================================

for (;;) {

//运行指示灯（RUN\_LIGHT）闪烁---------------------------------------------

/\*mRuncount++; //主循环次数计数器+1

if (mRuncount >= RUN\_COUNTER\_MAX) //主循环次数计数器大于设定的宏常数

{

mRuncount=0; //主循环次数计数器清零

//light\_change(LIGHT\_BLUE); //蓝色运行指示灯（RUN\_LIGHT\_BLUE）状态变化

mLight\_chang\_num++; //灯状态变换次数+1

printf("灯状态变换次数mLight\_chang\_num=%d\n",mLight\_chang\_num);

uart\_send\_string(UART\_0, "UART\_0!\r\n");

uart\_send\_string(UART\_1, "UART\_1!\r\n");

uart\_send\_string(UART\_2, "UART\_2!\r\n");

}\*/

//以下加入用户程序--------------------------------------------------------

} //主循环end\_for

//主循环结束==================================================================

}

##### 实验小结：

（关于这次实验所遇到的问题和解决的方法）

这次试验是初步学习使用UART构件中的串口的接收中断服务例程来实现字符的接受和转发；而且结合了前几次实验所学的GPIO控件的使用：控制小灯的亮暗；

发现的问题：一次只能接受一个字节的字符输入，及不论一次输入多少字符发送，只要含有能改变灯的颜色的字符在输入的内容中，则会改变灯的颜色，不能精准的判断命令的正确与否，所以需要设计一个能精准识别命令来控制小灯的亮暗的函数——组帧createframe() ；

#### 实验任务二：

思路：通过组帧函数来对每次输入的字符进行组帧——字符串buffer[64]，每次组帧成功后通过if-else if …else进行条件判断，依次进行判断，对帧内容符合编码要求的字符串 转入控制灯亮的程序；

##### 源码：

**#define** FrameHead (0x50) //帧头 ASCII码对应P

**#define** FrameTail (0x43) //帧尾 ASCII码对应C

uint8\_t **CreateFrame**(uint\_8 Data, uint\_8 \* buffer) {

**static** uint\_8 frameLen = 0; //帧的计数器

uint\_8 frameFlag; //组帧状态

frameFlag = 0; //组帧状态初始化

//根据静态变量frameCount组帧

**switch** (frameLen) {

**case** 0: //第一个数据

{

**if** (Data == FrameHead) //收到数据是帧头FrameHead

{

buffer[0] = Data;

frameLen++;

frameFlag = 0; //组帧开始

}

**break**;

}

**case** 1: //第二个数据，该数据是随后接收的数据个数

{

buffer[1] = Data - 0x30;

frameLen++;

**break**;

}

**default**: //其他情况

{

//第二位数据是有效数据长度,根据它接收余下的数据直到帧尾前一位

**if** (frameLen >= 2 && frameLen <= (buffer[1] + 1)) {

buffer[frameLen] = Data;

frameLen++;

**break**;

}

//若是末尾数据则执行

**if** (frameLen >= (buffer[1] + 2)) {

**if** (Data == FrameTail) //若是帧尾

{

buffer[frameLen] = Data; //将帧尾存入缓冲区

frameFlag = 1; //组帧成功

}

frameLen = 0; //计数清0，准备重新组帧

**break**;

}

}

} //switch\_END

**return** frameFlag; //返回组帧状态

}

##### 实验小结：

组帧函数是利用老师给的组帧函数，并没有做过多的修改。但是我仔细学习了老师源码，对源码的分析如下:老师首先限定了帧的格式：首尾P,C,第二位控制字段。对错误帧也运用设置了framelen这个变量来进行检验——每次若帧头帧尾不符合要求则自动指针指向buffer第一位即起到了检验输入的字符串是否满足正确的帧格式的作用；

##### 源码：

//串口一的终端服务例程

void UART1\_IRQHandler(void) {

uint\_8 ch;

uint\_8 flag;

uint\_8 buffer[64];//函数内部的局部变量，用来保存录入的正确格式的帧

DISABLE\_INTERRUPTS; //关总中断

if (uart\_get\_re\_int(UART\_1)) {

ch = uart\_re1(UART\_1, &flag); //调用接收一个字节的函数，清接收中断位

if(CreateFrame(ch,buffer)!=0){

if (buffer[2] == 'r' && buffer[3] == 'e' && buffer[4] == 'd') {

light\_control(LIGHT\_BLUE, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

light\_control(LIGHT\_RED, LIGHT\_ON);

}

else if (buffer[2] == 'b' && buffer[3] == 'l' && buffer[4] == 'u'

&& buffer[5] == 'e') {

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

light\_control(LIGHT\_RED, LIGHT\_OFF);

light\_control(LIGHT\_BLUE, LIGHT\_ON);

}

else if (buffer[2] == 'g' && buffer[3] == 'r' && buffer[4] == 'e'

&& buffer[5] == 'e' && buffer[6] == 'n') {

light\_control(LIGHT\_BLUE, LIGHT\_OFF);

light\_control(LIGHT\_RED, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_ON);

}

else if (buffer[2] == 'c' && buffer[3] == 'l' && buffer[4] == 'o'

&& buffer[5] == 's' && buffer[6] == 'e') {

light\_control(LIGHT\_BLUE, LIGHT\_OFF);

light\_control(LIGHT\_RED, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

}

else if (buffer[2] == 'v' && buffer[3] == 'i' && buffer[4] == 'o'

&& buffer[5] == 'l' && buffer[6] == 'e'

&& buffer[7] == 't') {

light\_control(LIGHT\_GREEN, LIGHT\_OFF);

light\_control(LIGHT\_BLUE, LIGHT\_ON);

light\_control(LIGHT\_RED, LIGHT\_ON);

}

else if (buffer[2] == 'w' && buffer[3] == 'h' && buffer[4] == 'i'

&& buffer[5] == 't' && buffer[6] == 'e') {

light\_control(LIGHT\_BLUE, LIGHT\_ON);

light\_control(LIGHT\_RED, LIGHT\_ON);

light\_control(LIGHT\_GREEN, LIGHT\_ON);

}

else if (buffer[2] == 'c' && buffer[3] == 'y' && buffer[4] == 'a'

&& buffer[5] == 'n') {

light\_control(LIGHT\_RED, LIGHT\_OFF);

light\_control(LIGHT\_BLUE, LIGHT\_ON);

light\_control(LIGHT\_GREEN, LIGHT\_ON);

}

else if(buffer[2] == 'y' && buffer[3] == 'e' && buffer[4] == 'l'

&& buffer[5] == 'l' && buffer[6] == 'o'

&& buffer[7] == 'w') {

light\_control(LIGHT\_BLUE, LIGHT\_OFF);

light\_control(LIGHT\_GREEN, LIGHT\_ON);

light\_control(LIGHT\_RED, LIGHT\_ON);

}

else {

}

}

if (flag) {

uart\_send1(UART\_1, ch); //向原串口发回一个字节

}

}

ENABLE\_INTERRUPTS;

}

##### 实验小结：

这个终端服务例程就是比较简单了，只是每次调用终端服务例程时调用组帧函数createframe() 通过返回值framflag 是否为1进行判断：是否组帧成功。一旦成功则进行帧判断，条件语句判断灯的情况。

#### 实验任务三：

在C#编写的串口通信窗口程序项目中添加八个按钮分别实现对应的功能，创建八个按钮，在每个按钮的方法里分别通过已经封装好的发送函数，向MCU串口里发送对应的帧来控制小灯。最后在打开/关闭串口按钮中添加this.button\*.Enable=false/true；来控制按钮的有效与否。

##### 源码：

private void button1\_Click(object sender, EventArgs e)

{

if (!sci.IsOpen)

{

//状态条进行提示

this.TSSLState.Text += "请先打开串口!";

return;

}

else

{

//创建容器存储发送数据

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes("P3redC");

sci.SCISendData(ref PublicVar.g\_SendByteArray);

this.TSSLState.Text = "红灯亮";

}

}

private void button2\_Click(object sender, EventArgs e)

{

if (!sci.IsOpen)

{

//状态条进行提示

this.TSSLState.Text += "请先打开串口!";

return;

}

else

{

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes("P5greenC");

sci.SCISendData(ref PublicVar.g\_SendByteArray);

this.TSSLState.Text = "绿灯亮";

}

}

private void button3\_Click(object sender, EventArgs e)

{

if (!sci.IsOpen)

{

//状态条进行提示

this.TSSLState.Text += "请先打开串口!";

return;

}

else

{

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes("P4blueC");

sci.SCISendData(ref PublicVar.g\_SendByteArray);

this.TSSLState.Text = "蓝灯亮";

}

}

private void button4\_Click(object sender, EventArgs e)

{

if (!sci.IsOpen)

{

//状态条进行提示

this.TSSLState.Text += "请先打开串口!";

return;

}

else

{

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes("P6violetC");

sci.SCISendData(ref PublicVar.g\_SendByteArray);

this.TSSLState.Text = "紫灯亮";

}

}

private void button5\_Click(object sender, EventArgs e)

{

if (!sci.IsOpen)

{

//状态条进行提示

this.TSSLState.Text += "请先打开串口!";

return;

}

else

{

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes("P6yellowC");

sci.SCISendData(ref PublicVar.g\_SendByteArray);

this.TSSLState.Text = "黄灯亮";

}

}

private void button6\_Click(object sender, EventArgs e)

{

if (!sci.IsOpen)

{

//状态条进行提示

this.TSSLState.Text += "请先打开串口!";

return;

}

else

{

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes("P5whiteC");

sci.SCISendData(ref PublicVar.g\_SendByteArray);

this.TSSLState.Text = "白灯亮";

}

}

private void button7\_Click(object sender, EventArgs e)

{

if (!sci.IsOpen)

{

//状态条进行提示

this.TSSLState.Text += "请先打开串口!";

return;

}

else

{

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes("P4cyanC");

sci.SCISendData(ref PublicVar.g\_SendByteArray);

this.TSSLState.Text = "青紫灯亮";

}

}

private void button8\_Click(object sender, EventArgs e)

{

if (!sci.IsOpen)

{

//状态条进行提示

this.TSSLState.Text += "请先打开串口!";

return;

}

else

{

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes("P5closeC");

sci.SCISendData(ref PublicVar.g\_SendByteArray);

this.TSSLState.Text = "关灯";

}

}



##### 实验小结：

学习了C#的GUI编程。通过这次实验让我初步了解了怎样通过串口程序实现窗口界面程序和MCU来进行通信。

### 实验总结：

这三个小实验花了不少时间，让我发现自身的编程能力还有待加强。对MCU的理解还是不深入，是要花大功夫才能有所进步的。实验过程中要有耐心，不能浮躁。对待实验态度要认真。

# 嵌入式手札四

## 实验一：利用SysTick模块进行不同闪烁模式小灯的设计

## 实验目的：

1、利用SysTick模块和UART模块 设计五种以上的不同闪烁模式流水灯用来识别不同情况；

2、输入数字（1-5）分别对应5种不同闪烁模式——代表连续闪烁对应的（1/2/3/4/5）次数后暂停1.5S后继续闪烁相应的次数，如此循环往复。

## 实验环境：

1）PC机一台，KL25开发板一片，导线若干

2) KDS开发环境，VS2017或更高平台

## 实验内容：

### 1）：设计串口接收模块UART

#### #1目的：

通过设计串口接收模块来接受输入的数字且进行判断：1-5分别对应不同的闪烁模式

#### #2问题：

##### 1）如何传递闪烁模式选择符：（tickCount）

@方法：

通过设置静态全局变量：uint\_8 tickCount = 1; 传递模式选择标识到SysTick模块中

@源码：

**void** **UART1\_IRQHandler**(**void**) {

uint\_8 ch;

uint\_8 flag;

uint\_8 buffer[64];

DISABLE\_INTERRUPTS;

**if** (uart\_get\_re\_int(UART\_1)) {

ch = uart\_re1(UART\_1, &flag);

**if** (ch == '1') {

tickCount = 1;

modeChange=1;

light\_control(LIGHT\_BLUE, LIGHT\_ON);

} **else** **if** (ch == '2') {

tickCount = 2;

modeChange=1;

light\_control(LIGHT\_BLUE, LIGHT\_ON);

} **else** **if** (ch == '3') {

tickCount = 3;

modeChange=1;

light\_control(LIGHT\_BLUE, LIGHT\_ON);

} **else** **if** (ch == '4') {

tickCount = 4;

modeChange=1;

light\_control(LIGHT\_BLUE, LIGHT\_ON);

} **else** **if** (ch == '5') {

tickCount = 5;

modeChange=1;

light\_control(LIGHT\_BLUE, LIGHT\_ON);

} **else** {

tickCount = 1;

modeChange=1;

light\_control(LIGHT\_BLUE, LIGHT\_ON);

}

}

**if** (flag) {

uart\_send1(UART\_1, ch);//若不是1-5，默认进入闪烁模式1

}

ENABLE\_INTERRUPTS;

}

### 2）设计SysTick模块

#### #目的：

控制每种闪烁模式下的闪烁情况：包括密集闪烁模式下的密集闪烁次数和闪烁间隔

#### #问题：

##### 1）如何控制不同模式对应不同的闪烁模式

@方法：通过if\_-esle if -else 选择判断语句来控制不同闪烁模式下的选择

##### 2）如何控制每种闪烁模式下的密集闪烁模式和停止闪烁间隔之间切换

@方法：通过在设置一个全局的静态变量控制

**static** uint\_8 modeChange = 0; //模式切换标志，用于切换不同闪烁子模式

modeChange=1时，密集闪烁中

modeChange=0时，闪烁间隔

##### 3）如何控制每次不同闪烁模式下的密集闪烁次数

@方法：通过设置一个控制闪烁次数的方法内的静态变量：

**static** uint\_8 twinkleCount = 0; //密集闪烁指示灯次数变量

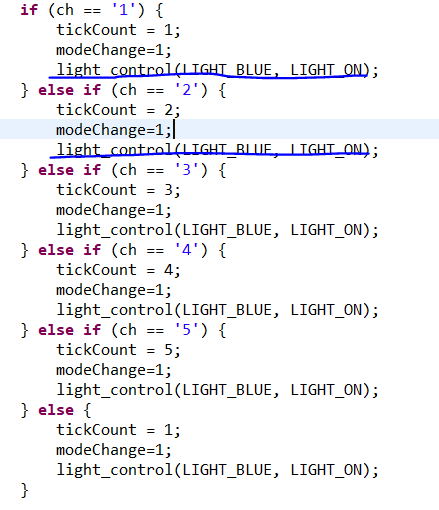
来控制闪烁次数；

##### 4）如何正确控制每种闪烁模式开启

@方法：

除了第一种只闪烁一次的情况（这种情况亮灭间隔相同不需要区分密集闪烁）；其他控制密集闪烁的闪烁次数都遵循一个公式：twinkleCount <= (tickCount \* 2 - 2)

1、每次在串口输入模式选择符后，都会设置小灯变亮



这为了符合密集闪烁的第一种状态是亮的状态

2、每次在串口输入模式选择符后，都会设置modeChange=1；

这是为了先进入每种闪烁模式下的密集闪烁模式，配合上面的控制灯亮就可以设置正确的闪烁模式了

##### 5）如何控制密集闪烁模式和熄灭间隔是1.5s：

@方法：

密集闪烁模式最后一次状态变换时亮变暗：SysTickcount=30 即0.3S改变状态一次；为了剩余的1.5s间隔；需要在modeChange=0的情况下设置 SysTickcount<=150 的间隔用来保持1.5s的灯暗状态的时间。1.5s后才会让灯重新切换为亮状态；**这样保证了每次密集闪烁的初始状态始终为亮；**

@源码：

**void** **SysTick\_Handler**(**void**) //深入了解中断的内涵

//默认值为100,则表示1S闪一次

{

**static** uint\_8 SysTickcount = 0; //静态变量,负责控制每次亮灭间隔的时间

**static** uint\_8 twinkleCount = 0; //密集闪烁指示灯次数变量

SysTickcount++;

**if** (tickCount == 1) {

**if** (SysTickcount >= 30)

{

SysTickcount = 0;

//秒计时程序

SecAdd1(g\_time); //g\_time是时分秒全局变量数组

}

} **else** **if** (tickCount == 2 || tickCount == 3 || tickCount == 4

|| tickCount == 5) {

**switch** (modeChange) {

**case** 1: {

**if** (twinkleCount <= (tickCount \* 2 - 2)) {

**if** (SysTickcount >= 30){

SysTickcount = 0;

//秒计时程序

SecAdd1(g\_time); //g\_time是时分秒全局变量数组

twinkleCount++;

}

} **else** {

twinkleCount = 0; //初始化密集闪烁次数

modeChange = 0; //更改闪烁指示标志

}

}

**case** 0: {

**if** (SysTickcount >= 150) //设置等待时间，足够区分闪烁模式

{

SysTickcount = 0; //1.5s以后，小灯延续了1.5s的暗状态才会切换为亮状态

//秒计时程序

SecAdd1(g\_time); //g\_time是时分秒全局变量数组

modeChange = 1; //更改闪烁指示标志

}

}

}

}

**else**{}

}

## 实验总结：

1、这次试验初步理解了systick模块的使用，定时器每次的系统中断产生的时间间隔是不变的，只是通过改变 SysTickcount 的次数来改变g\_time 数组的秒元素递增来间接控制小灯的亮灭切换的

2、本次实验有相当一部分需要比较扎实的编程功底，特别是对整个闪烁模式的选择——tickcount和闪烁密集模式和熄灭间隔之间的切换标志 modeChange ；需要设置合理的静态变量来进行控制！

3、对PWM还是很生疏，需要花许多时间学习！

# 嵌入式手札五

## 实验一：利用TMP模块控制PWM波占空比继而控制小灯的亮暗程度；

## 实验目的：

1、利用TPM模块用来控制PWM波的占空比从而控制小灯的亮度 ；

2、通过UART模块实现输入1-100之间的数字来控制小灯亮暗

3、通过C#编程，实现窗口内滑动滚动条控制小灯的亮度

## 实验环境：

1）PC机一台，KL25开发板一片，导线若干

2) KDS开发环境，VS2017或更高平台

## 实验内容：

### 1）设计串口接收模块UART

#### #1目的：

通过设计串口接收模块来接受输入的亮度值，根据对用的亮度值，改变duty——占空比

#### #2问题：

##### 1）如何正确识别每次输入的亮度值？

@方法：

1）考虑到接受中断一次只能接受一个字节，所以无法很好的区分输入的亮度；所以决定通过前面章节组帧的知识，来进行组帧传输；从符合条件的帧中提取亮度值；（只有满足亮度值范围的帧才会被提取亮度值）

2）通过设置一个静态全局变量conDuty来向TPM2模块中的函数pwm\_update(TPM2\_CH0, conDuty); 进行duty传值，每次TPM2模块中断都会传入更新后对应的值来更新占空比；

@源码：

**void** **UART1\_IRQHandler**(**void**) {

uint\_8 ch;

uint\_8 flag;

**static** uint\_8 buffer[64];

DISABLE\_INTERRUPTS;

//获取一个字节的输入

**if** (uart\_get\_re\_int(UART\_1)) {

ch = uart\_re1(UART\_1, &flag);

//调用组帧函数；

**if** (CreateFrame(ch, buffer) != 0) {

//进行条件选择，看看是几位数；

**if** (buffer[1] == 1) {

conDuty = buffer[2]; //进行值传递;

} **else** **if** (buffer[1] == 2) {

conDuty=0;//通过设置for循环来将conDuty递增到满足条件的值

uint\_8 count=buffer[2]\*10+buffer[3];

uint\_8 i=0;

**for**(i;i<count;i++){

conDuty++;

}

} **else** **if** (buffer[1] == 3) {

**if** (buffer[2] == '1' && buffer[3] == '0' && buffer[4] == '0') {

conDuty = 100; //进行值传递;

}

} **else** {

}

}

}

**if** (flag) {

uart\_send1(UART\_1, ch); //向原串口发回一个字节

}

ENABLE\_INTERRUPTS;

}

### 2）设计TPM2模块

#### #目的：

控制UART模块提取的亮度值来设置占空比duty进而控制小灯的亮暗程度；

#### #问题：

##### 1）如何将提取的亮度值传入更新占空比的函数中

@方法：

通过设置一个静态全局变量conDuty来连接TPM2模块和UART模块；在两个模块之间进行占空比的值传递

@源码：

**void TPM2\_IRQHandler(void) {**//深入了解中断的内涵

//默认值为100,则表示1S闪一次

**static** **float** duty = 90.0; //静态变量duty（占空比），数值越高，灯越暗

**static** uint\_8 Up\_Down = 1; //占空比增减标志

**if** (tpm\_get\_int(1) == 1) //若有TPM1的溢出中断

{

tpm\_clear\_int(1); //清TPM1的溢出中断标志

pwm\_update(TPM2\_CH0, conDuty); //PWN更新，注意参数！指明了哪个模块的那个通道！

}

}

### 3）进行C#窗口编程，

#### #目的：实现滑动滚动条控制小灯的亮暗

#### #问题：

##### 1）如何或取滚动条的位置的取值然后进行组帧发送给MCU板进行亮度控制

@方法：1、定义HScrollBar 控件，定义一个字符串进行组成组帧；最后通过发送函数，将字符串发送给MCU板子，来控制小灯；

@源码：//滑动条控制小灯亮度

private void hScrollBar1\_Scroll(object sender, ScrollEventArgs e)

{

if (!sci.IsOpen)

{

//状态条进行提示

this.TSSLState.Text += "请先打开串口!";

return;

}

else

{

string str1 = "P";

string str2 = "";

int ex = 0;//存放滑动条的value

ex=hScrollBar1.Value;

if (ex <= 9)

{

str1 += "1";

}

else if (ex < 100)

{

str1 += "2";

}

else

{

str += "3";

}

str2 = ex.ToString();//整型转换为字符串进行组帧

str1 += str2;

str1 += "C";

//组帧成功；

//发送给板子；

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes(str1);//发送消息；

sci.SCISendData(ref PublicVar.g\_SendByteArray);

}

}



## 实验总结：

1、这次试验初步理解了TPM模块的使用；学习了占空比和小灯亮暗成都之间的关系

2、进一步学习了C#窗口编程里的组件的使用；

3、硬件这方面真的需要花很多时间去学习，我现在学的都只是皮毛而已；

# 嵌入式手札六

## 实验二：发送系统时间至MCU实验板，并发送时间至C#窗口显示，统计时差；

## 实验目的：

1、利用MCU中的TPM模块用来计时；

2、通过UART模块实现输入发送系统时间至MCU中并且计时

3、通过C#编程，实现窗口内显示MCU返回的时间

## 实验环境：

1）PC机一台，KL25开发板一片，导线若干

2) KDS开发环境，VS2017或更高平台

## 实验内容：

### 1）设计串口接收模块UART

#### #1目的：通过设计串口接收模块来接受系统发送的时间，通过组帧函数进行判断从而将系统发送过来的时间存放到g\_time[ ]数组中；

#### #2问题：

##### 1）如何正确将发送过来的时间存放到g\_time[ ]数组里

@方法：

1）考虑到C#输入的数字都是字符型，每个数字字符都对应一个（码）值；buffer缓存中的数字部分每位字节运算都是用对应的值计算；但是发送给c#的时候，必须发送数字相应的字符码；通过减去字符码和实际数值的差值进行存放；

@源码：

**void** **UART1\_IRQHandler**(**void**) {

uint\_8 ch;

uint\_8 flag;

**static** uint\_8 buffer[64];

DISABLE\_INTERRUPTS;

**if** (uart\_get\_re\_int(UART\_1)) {

ch = uart\_re1(UART\_1, &flag);

**if** (CreateFrame(ch, buffer) != 0) {

**if** (buffer[1] == 8) { //C#输入的数字都是字符型，每个数字字符都对应一个（码）值

//buffer缓存中的数字部分每位字节运算都是用对应的ASCII码值计算

//但是发送给c#的时候，必须发送数字相应的字符ASCII码值；

//计算hh:mm:ss

uint\_8 g\_time0 = (buffer[2]);

g\_time0 -= 48;

g\_time0 \*= 10;

buffer[3] -= 48;

g\_time0 += (buffer[3]);

uint\_8 g\_time1 = (buffer[5]);

g\_time1 -= 48;

g\_time1 \*= 10;

buffer[6] -= 48;

g\_time1 += (buffer[6]);

uint\_8 g\_time2 = (buffer[8]);

g\_time2 -= 48;

g\_time2 \*= 10;

buffer[9] -= 48;

g\_time2 += (buffer[9]);

g\_time[0] = g\_time0;

g\_time[1] = g\_time1;

g\_time[2] = g\_time2;

}

}

}

**if** (flag) {

uart\_send1(UART\_1, ch);

}

ENABLE\_INTERRUPTS;

}

### 2）设计TPM0模块

#### #目的：控制TPM0模块进行计时，改变g\_time[ ]的数值

#### #问题：

##### 1）如何将MCU内部的g\_time[ ]数组内部的时间用字符串形式发送给C#界面；

@方法：

通过将g\_time[ ]内部的三位时分秒的数值进行位拆分，拆分出时分秒每个数值的十位，个位；加入字符‘ ：’，组合成时间格式发送给C#通信窗口

@源码：

**void** **TPM0\_IRQHandler**(**void**) {

**static** uint\_32 cnt; //中断次数

**if** (tpm\_get\_int(0) == 1) //若有TPM0的溢出中断

{

tpm\_clear\_int(0); //清TPM0的溢出中断标志

cnt++; //中断次数+1

**if** (cnt >= 100) //若达到100次（即1秒）

{

//实现数值向数字字符ASCII码值的转换,组成时间字符串

uart\_send1(UART\_1, (g\_time[0] / 10) + 48);

uart\_send1(UART\_1, (g\_time[0] % 10) + 48);

uart\_send1(UART\_1, ':');

uart\_send1(UART\_1, (g\_time[1] / 10) + 48);

uart\_send1(UART\_1, (g\_time[1] % 10) + 48);

uart\_send1(UART\_1, ':');

uart\_send1(UART\_1, (g\_time[2] / 10) + 48);

uart\_send1(UART\_1, (g\_time[2] % 10) + 48);

cnt = 0; //中断次数清0

//调用“秒+1计时子函数”，给全局变量数组g\_time赋值（时、分、秒）

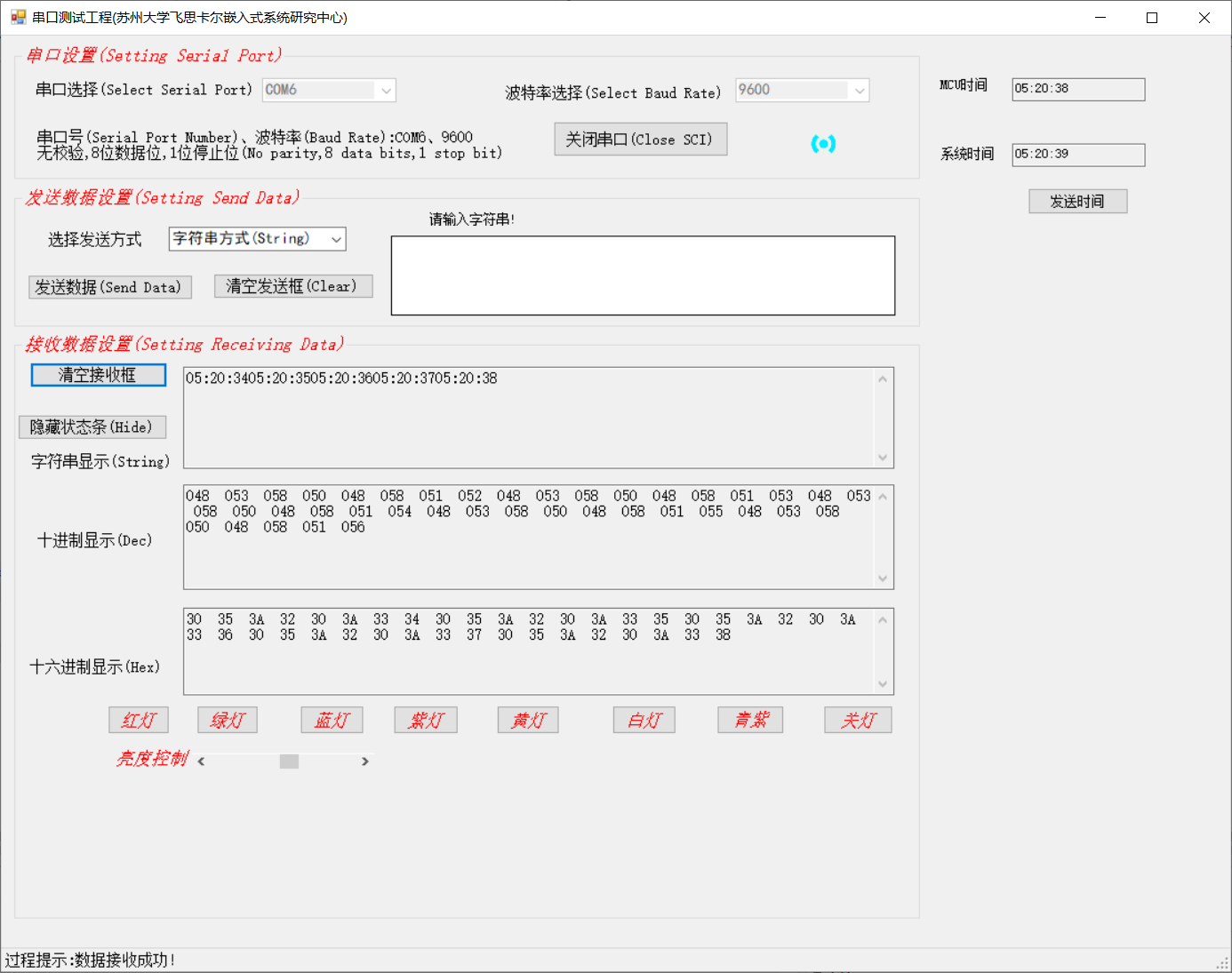
SecAdd1(g\_time);

}

}

}

### 3）进行C#窗口编程



#### #目的：实现动态显示系统时间

#### #问题：

##### 1）如何每秒动态更新时间

@方法：1、定义Timer控件，利用方法private void timer1\_Tick\_1(object sender, EventArgs e)

用于每秒更新时间；

@源码：private void timer1\_Tick\_1(object sender, EventArgs e)

{

String date = DateTime.Now.ToString("hh:mm:ss");

systemTime.Text = date;

}

##### 2）发送系统时间至MCU板子

@方法：定义一个按钮控件，进行点击控件事件编程，实现系统时间发送；

@源码：

private void sendButton\_Click(object sender, EventArgs e)

{

//更改全局变量值

change\_flag = 1;

String date = "P8" + DateTime.Now.ToString("hh:mm:ss") + "C";

//方法二

//date2用于分割时间字符串

String date2 = DateTime.Now.ToString("hh:mm:ss");

String[] date\_split = date2.Split(':');

for (int i = 0; i < 3; i++)

{

g\_time[i] = Int32.Parse(date\_split[i]);

}

//发送日期数据给板子；

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes(date);//发送消息；

sci.SCISendData(ref PublicVar.g\_SendByteArray);

}

##### 3）如何接收MCU发送的时间到Text控件里

@方法：定义private void SCIUpdateRevtxtbox\_2(Object textbox, string text) 方法，实现发送数据到TextBox mcu\_time 控件当中；在private void SCIPort\_DataReceived(object sender,

System.IO.Ports.SerialDataReceivedEventArgs e) 方法中调用SCIUpdateRevtxtbox\_2（）方法实现每次发送新时间时清空原来的旧时间；

@源码：

private void SCIPort\_DataReceived(object sender,

System.IO.Ports.SerialDataReceivedEventArgs e)

{

String str = String.Empty;

bool Flag;//标记串口接收数据是否成功

int len;//标记接收的数据的长度

byte[] ch2 = new byte[2];

//ComDevice.Encoding = System.Text.Encoding.GetEncoding("GB2312");

//调用串口接收函数,并返回结果

Flag = sci.SCIReceiveData(ref PublicVar.g\_ReceiveByteArray);

if (Flag == true)//串口接收数据成功

{

len = PublicVar.g\_ReceiveByteArray.Length;

//对于字符串形式,考虑到可能有汉字,

//直接调用系统定义的函数,处理整个字符串

str = Encoding.GetEncoding("GB2312").GetString(PublicVar.g\_ReceiveByteArray);

//

SCIUpdateRevtxtbox(TbShowString, str);

//发送MCU时间到C#窗口上

if (change\_flag == 1)

{

SecAdd1(g\_time);

String date = g\_time[0].ToString() +":"+ g\_time[1].ToString() + ":"

+ g\_time[2].ToString();

date = String.Format("{0:hh:mm:ss}", date);

//这是第一种最好的方法；通过MCU发送MCU里面的g\_time[]数组的时间显示到文本框里面

SCIUpdateRevtxtbox\_2(mcuTime, str);

//第二种办法是通过在C#内设置一个int g\_time[]数组；

//将时间存放到数组中，通过MCU发送秒加一信号，实现MCU同样的计时功能；

//缺点：时间显示格式不能统一

//SCIUpdateRevtxtbox\_2(mcuTime, date);

}

//十进制和十六进制形式按字节进行处理

for (int i = 0; i < len; i++)

{

//十进制都是按照三位来显示,字节之间有空格表示区分

SCIUpdateRevtxtbox(TbShowDec,

PublicVar.g\_ReceiveByteArray[i].ToString("D3") + " ");

//十六进制都是按照两位来显示,字节之间有空格表示区分

SCIUpdateRevtxtbox(TbShowHex,

PublicVar.g\_ReceiveByteArray[i].ToString("X2") + " ");

}

// sci.SCIReceInt(SCIPort, 1);//设置产生接收中断的字节数【2014-5-5 注释，否则会导致程序无响应】

this.TSSLState.Text = "过程提示:数据接收成功!";

}

//接收数据失败

else

{

//sci.SCIReceInt(SCIPort, 1);//设置产生接收中断的字节数【2014-5-5 注释，否则会导致程序无响应】

this.TSSLState.Text = "过程提示:数据接收失败!";

}

}

private void SCIUpdateRevtxtbox\_2(Object textbox, string text)

{

//textbox显示文本与串口执行不在同一个线程中

if (((TextBox)textbox).InvokeRequired)

{

handleinterfaceupdatedelegate InterFaceUpdate = new

handleinterfaceupdatedelegate(SCIUpdateRevtxtbox\_2);

this.Invoke(InterFaceUpdate, new object[] { textbox, text });

}

else

{

((TextBox)textbox).Text = text;

//把光标放在第一行

((TextBox)textbox).SelectionStart = 0;

//将文本框中的内容调整到当前插入符号位置

((TextBox)textbox).ScrollToCaret();

}

}

4）如何在点击发送按钮后才发送MCU时间到C#通信窗口；

@方法：设置一个显示MCU系统时间标志private int change\_flag = 0——传送时间开始标志

@源码：private void sendButton\_Click(object sender, EventArgs e)

{

//更改全局变量值

change\_flag = 1;

String date = "P8" + DateTime.Now.ToString("hh:mm:ss") + "C";

//date2用于分割时间字符串

String date2 = DateTime.Now.ToString("hh:mm:ss");

String[] date\_split = date2.Split(':');

for (int i = 0; i < 3; i++)

{

g\_time[i] = Int32.Parse(date\_split[i]);

}

//发送日期数据给板子；

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes(date);//发送消息；

sci.SCISendData(ref PublicVar.g\_SendByteArray);

}

## 实验总结：

1、这次试验初步理解了TPM模块的使用；学习了C#中的时间组件使用，能利用系统内部方法实现动态地更新时间了；

2、还有一点就是花了很长的时间弄明白了C#与MCU进行传输的字符具体使用方法，每一个字符字节的值是对应他的ASCII码的十进制数值；一开始在编写MCU中的UART\_1控件的内容的时候给我带来了很大的麻烦；花了很长时间才弄懂了其中地转换方法；

3、还有就是明白了静态全局变量的生存周期和时间数组的时间保存的关系，从而能够通过buffer[ ]传输时间到MCU中的g\_time[ ] 中

3、对C#的语法学习也在编程的过程中得到一定的锻炼，对程序结构的设计也得到了训练，学习嵌入式可以训练自己很多方面的能力！全面提高！

4、嵌入式还是需要花很大很大的气力去学习，要提高自己的学习效率，积极向同学老师请教才行，要培养自己的兴趣！才能有源源不断学习的乐趣！

# 嵌入式手札七 LCD，键盘的应用

## 实验目的：

1、利用MCU中的sysTick模块用来计时；

2、通过UART模块实现输入发送系统时间至MCU中并且计时发送到LCD上显示

3、实验一：通过串口发送系统时间到MCU中，MCU计时并发送LCD动态显示；

4、实验二：如何通过试验箱的键盘输入6位数的时间数值，发送给g\_time[]数组中，开始计时送LCD显示

5、实验三：通过在C#窗口程序上设置虚拟键盘实现MCU实体键盘和虚拟键盘的同步

## 实验环境：

1）PC机一台，KL25开发板一片，导线若干

2) KDS开发环境，VS2017或更高平台

## 实验内容：

## 实验一：

### 1）设计串口接收模块UART

#### #1目的：通过设计串口接收模块来接受系统发送的时间，通过组帧函数进行判断从而将系统发送过来的时间存放到g\_time[ ]数组中，从而让sysTick模块开始计时发送到LCD中；

#### #2问题：

##### 问题1）如何正确将发送过来的时间存放到g\_time[ ]数组里

@方法：（这里用到的其实和上次定时器实验很接近，与上次相比，多作的事情就是将LCD组件利用起来每秒显示时间）

#1考虑到C#输入的数字都是字符型，每个数字字符都对应一个（码）值；buffer缓存中的数字部分每位字节运算都是用对应的值计算；但是发送给c#的时候，必须发送数字相应的字符码；通过减去字符码和实际数值的差值进行存放；

问题2）如何在组帧成功时同步让sysTick模块开始同步计时

@方法：设置全局变量flagTime标志，同志sysTick模块开始计时

@源码：

**void** **UART1\_IRQHandler**(**void**) {

uint\_8 ch;

uint\_8 flag;

**static** uint\_8 buffer[64];

DISABLE\_INTERRUPTS;

**if** (uart\_get\_re\_int(UART\_1)) {

ch = uart\_re1(UART\_1, &flag);

//flagInput = 1;

//用于接收输入到MCU的字符

//g\_LCDBuffer[31] = ch;

**if** (CreateFrame(ch, buffer) != 0) {

//组帧成功，更改标志

flagTime = 1;

**if** (buffer[1] == 8) { //C#输入的数字都是字符型，每个数字字符都对应一个ASCII（码）值

//LCD\_buffer[]更新时间，更新显示内容

uint\_8 g\_temp[32] = " ";

uint\_8 i = 0;

**for** (i = 0; i < 32; i++) {

g\_LCDBuffer[i] = g\_temp[i];

} //LCD缓冲区赋值

//buffer缓存中的数字部分每位字节运算都是用对应的ASCII码值计算

//但是发送给c#的时候，必须发送数字相应的字符ASCII码值；

//计算hh:mm:ss

uint\_8 g\_time0 = (buffer[2]);

g\_time0 -= 48;

g\_time0 \*= 10;

buffer[3] -= 48;

g\_time0 += (buffer[3]);

uint\_8 g\_time1 = (buffer[5]);

g\_time1 -= 48;

g\_time1 \*= 10;

buffer[6] -= 48;

g\_time1 += (buffer[6]);

uint\_8 g\_time2 = (buffer[8]);

g\_time2 -= 48;

g\_time2 \*= 10;

buffer[9] -= 48;

g\_time2 += (buffer[9]);

g\_time[0] = g\_time0;

g\_time[1] = g\_time1;

g\_time[2] = g\_time2;

}

}

}

**if** (flag) {

uart\_send1(UART\_1, ch);

}

ENABLE\_INTERRUPTS;

}

问题三：

### 2）设计sysTick模块

#### #目的：控制sysTick模块进行计时，改变g\_time[ ]的数值,并且动态更新时间到LCD

#### #问题：

##### 1）如何将MCU内部的g\_time[ ]数组内部的时间分别赋给LCD发送缓存g\_LCDBuffer[]中；

@方法：

通过将g\_time[ ]内部的三位时分秒的数值进行位拆分，拆分出时分秒每个数值的十位，个位；加入字符‘ ：’，组合成时间格式发送LCD发送缓存g\_LCDBuffer[]中

2）

@源码：

**if** (flagTime == 1) {

//静态变量,负责控制每次亮灭间隔的时间

**static** uint\_8 SysTickcount = 0;

SysTickcount++;

**if** (SysTickcount >= 100) {

SysTickcount = 0;

SecAdd1(g\_time); //时间加一

g\_LCDBuffer[24] = ((g\_time[0] / 10) + 48);

g\_LCDBuffer[25] = ((g\_time[0] % 10) + 48);

g\_LCDBuffer[26] = ':';

g\_LCDBuffer[27] = ((g\_time[1] / 10) + 48);

g\_LCDBuffer[28] = ((g\_time[1] % 10) + 48);

g\_LCDBuffer[29] = ':';

g\_LCDBuffer[30] = ((g\_time[2] / 10) + 48);

g\_LCDBuffer[31] = ((g\_time[2] % 10) + 48);

LCDShow(g\_LCDBuffer); //LCD显示

}

}

## 实验二：

### 1）设计sysTick模块

#### #目的：通过键盘按下通过试验箱的键盘输入6位数的时间数值，发送给g\_time[]数组中，开始计时送LCD显示

#### #问题：

##### 问题1）如何准确识别按下的六个时间数值后，开始计时

@方法：设计一个全局变量计时器flagInput=0，初始按键时钟计时标志flagTime=0,停止计时，通过按六次，每次将时间数值送往g\_time[]更改计时标志flagTime开启计时

@源码：

**if** (kb != 0xff) //有实际按键

{

//设计实验实验

flagTime = 0; //输出时间，暂停标志

**if** (countIn == 0 || countIn == 1 || countIn == 2 || countIn == 3

|| countIn == 4 || countIn == 5) {

countIn++; //数值加一

**switch** (countIn) {

**case** 0: {

g\_time0 = (KBDef(kb) - 48);

}

**case** 1: {

g\_time0 \*= 10;

g\_time0 += (KBDef(kb) - 48);

}

**case** 2: {

g\_time1 = (KBDef(kb) - 48);

}

**case** 3: {

g\_time1 \*= 10;

g\_time1 += (KBDef(kb) - 48);

}

**case** 4: {

g\_time2 = (KBDef(kb) - 48);

}

**case** 5: {

g\_time2 \*= 10;

g\_time2 += (KBDef(kb) - 48);

//赋值给时间数组

g\_time[0] = g\_time0;

g\_time[1] = g\_time1;

g\_time[2] = g\_time2;

//复位计数值，表示输入结束;

countIn = 0;

flagTime = 1; //显示时间标志赋值

}

}

}

}

## 实验三：

### 1）设计设计串口接收模块UART

#### #目的：接收C#虚拟键盘发送的数据，同步显示在LCD上

#### #问题

##### 问题1）：如何识别虚拟键盘发送的按键

@方法：通过组帧的方式，C#按下一个键，发送一个帧；通过条件判断语句筛选出发送的组帧对应的按键，将按键值赋给LCD输出缓存g\_LCDBuffer[]，调用显示函数进行显示；

PS:帧尾用到了C,所以在发送C按键对应的值时，用小写的c进行组帧发送

@源码：

**if** (CreateFrame(ch, buffer) != 0) {

//组帧成功，更改标志

**if** (buffer[1] == 1) {

//更改buffer显示内容

uint\_8 g\_temp[32] = " ";

uint\_8 i = 0;

**for** (i = 0; i < 32; i++) {

g\_LCDBuffer[i] = g\_temp[i];

}

**switch** (buffer[2]) {

**case** '1': {

g\_LCDBuffer[31] = '1';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** '2': {

g\_LCDBuffer[31] = '2';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** '3': {

g\_LCDBuffer[31] = '3';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** '4': {

g\_LCDBuffer[31] = '4';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** '5': {

g\_LCDBuffer[31] = '5';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** '6': {

g\_LCDBuffer[31] = '6';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** '7': {

g\_LCDBuffer[31] = '7';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** '8': {

g\_LCDBuffer[31] = '8';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** '9': {

g\_LCDBuffer[31] = '9';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** 'A': {

g\_LCDBuffer[31] = 'A';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** 'B': {

g\_LCDBuffer[31] = 'B';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** 'c': {

//这里有个需要注意的地方，因为，组帧最后用的是C，所以按键发送的内容需要用c来判断

g\_LCDBuffer[31] = 'C';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** 'D': {

g\_LCDBuffer[31] = 'D';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** 'E': {

g\_LCDBuffer[31] = 'E';

LCDShow(g\_LCDBuffer);

**break**;

}

**case** 'F': {

g\_LCDBuffer[31] = 'F';

LCDShow(g\_LCDBuffer);

**break**;

}

**default**: {

}

}

}

}

}

### 2）设计sysTick模块

#### #目的：通过追踪试验箱键盘按下的键，发送到LCD上显示并发送给C#串口助手进行虚拟键盘同步显示

#### #问题：

##### 问题1）：如何通过sysTick模块发送给C#;且让C#识别出发送的按键值

@方法:利用组帧的思想，将每一个按键的值组成 【“\*”+”键值”】的形式，从而在c#端让程序自动识别

@源码：

**if** (kb != 0xff) //有实际按键

{

//进阶性实验

uint\_8 g\_temp[32] = " ";

uint\_8 i = 0;

**for** (i = 0; i < 32; i++) {

g\_LCDBuffer[i] = g\_temp[i];

} //LCD缓冲区赋值

uint\_8 pressKey = KBDef(kb);

**switch** (pressKey) {

**case** '1': {

g\_LCDBuffer[31] = '1';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, '1');

**break**;

}

**case** '2': {

g\_LCDBuffer[31] = '2';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, '2');

**break**;

}

**case** '3': {

g\_LCDBuffer[31] = '3';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, '3');

**break**;

}

**case** '4': {

g\_LCDBuffer[31] = '4';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, '4');

**break**;

}

**case** '5': {

g\_LCDBuffer[31] = '5';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, '5');

**break**;

}

**case** '6': {

g\_LCDBuffer[31] = '6';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, '6');

**break**;

}

**case** '7': {

g\_LCDBuffer[31] = '7';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, '7');

**break**;

}

**case** '8': {

g\_LCDBuffer[31] = '8';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, '8');

**break**;

}

**case** '9': {

g\_LCDBuffer[31] = '9';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, '9');

**break**;

}

**case** 'A': {

g\_LCDBuffer[31] = 'A';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, 'A');

**break**;

}

**case** 'B': {

g\_LCDBuffer[31] = 'B';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, 'B');

**break**;

}

**case** 'C': {

//这里有个需要注意的地方，因为，组帧最后用的是C，所以按键发送的内容需要用c来判断

g\_LCDBuffer[31] = 'C';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, 'C');

**break**;

}

**case** 'D': {

g\_LCDBuffer[31] = 'D';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, 'D');

**break**;

}

**case** 'E': {

g\_LCDBuffer[31] = 'E';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, 'E');

**break**;

}

**case** 'F': {

g\_LCDBuffer[31] = 'F';

LCDShow(g\_LCDBuffer);

uart\_send1(UART\_1, '\*');

uart\_send1(UART\_1, 'F');

**break**;

}

**default**: {

**break**;

}

}

}

### 3）进行C#窗口编程

#### #目的：实现虚拟键盘和实体键盘同步显示

#### #方法：

##### 1）通过在C#的串口接收程序中定义switch语句，于每个MCU发送的str 进行判断；根据需要完成相关操作；

@操作：

#1：在设计见面设计如下的用按钮组成的键盘（只有串口成功打开时，才能显示）



#2、对于实体按键发送的帧只要识别就动态控制对那个按键进行状态切换以达到按压的效果

#3：对于从虚拟键盘按键的每个按键编写对应的click（）方法，按压后发送帧，按钮使能关闭；对从MCU返回帧进行识别后，识别成功则按钮是能开启；达到按键按压的效果；

；

#### #源码：

##### 1）按键按压（篇幅所限，只列出样例）按键0的对应key0\_Click（）方法

private void key0\_Click(object sender, EventArgs e)

{

PublicVar.g\_SendByteArray = new byte[10];

PublicVar.g\_SendByteArray =

System.Text.Encoding.Default.GetBytes("P10C");//发送消息；

sci.SCISendData(ref PublicVar.g\_SendByteArray);

this.key0.Enabled = false;

}

##### 2）对按下虚拟键盘或实体键盘的返回帧的条件筛选

private void SCIPort\_DataReceived(object sender,

System.IO.Ports.SerialDataReceivedEventArgs e)

{

String str = String.Empty;

bool Flag;//标记串口接收数据是否成功

int len;//标记接收的数据的长度

byte[] ch2 = new byte[2];

//ComDevice.Encoding = System.Text.Encoding.GetEncoding("GB2312");

//调用串口接收函数,并返回结果

Flag = sci.SCIReceiveData(ref PublicVar.g\_ReceiveByteArray);

if (Flag == true)//串口接收数据成功

{

len = PublicVar.g\_ReceiveByteArray.Length;

//对于字符串形式,考虑到可能有汉字,

//直接调用系统定义的函数,处理整个字符串

str = Encoding.GetEncoding("GB2312").GetString(PublicVar.g\_ReceiveByteArray);

//

SCIUpdateRevtxtbox(TbShowString, str);

switch (str)

{

//虚拟键盘返回帧判断

case "P10C":

{

this.key1.Enabled = true;

break;

}

.

.

.

//实体键盘返回帧判断

case "\*1":

{

this.keyF.Enabled = false;

this.key1.Enabled = true;

break;

}

.

.

.

default: { break; }

}

.

.

.

## 实验总结：

1、这次试验真的花了很多的时间，我发现，越往后越是对编程能力的考验。对整个程序框架的大致原理理解不难，难在灵活地通过编程控制这些组件，来达到自己想要的目的，编程技术不到家是很致命的。

2、老师课上说过要能够扛得住挫折，让我体会很深，越深入的学习越有难度，不可能像起初入门那样一帆风顺，肯定会有很多很多的困难，不能灰心丧气，要怀着兴奋的态度去挑战问题，解决问题，自己才能有所成长。

3、学习的节奏一定要掌控好，加快学习效率。抓住重点学习；

# 嵌入式手扎八 ADC/DAC（传感器）

说明：因为这次ADC和DAC模块做的比较晚，所以这次的实验报告就结合了本次期末考试做的实验第41题来进行书写；（因为有一些普通的传感器主要用到了AD模块接收模拟量的通道输入进行处理，和温度传感器原理类似。所以这次手札主要围绕如何再程序里写时序来接收传感器发送过来的数据的这个问题）

## 实验目的：

1、通过ADC模块进行传感器温度读取显示

2、通过DAC模块进行三角波的C#端显示

3、通过编写温湿度传感器DHT11和MCU之间的数据传输时的时序程序来控制温度数据的输入到MCU

## 实验环境：

1）PC机一台，KL25开发板一片，导线若干

2) KDS开发环境，VS2017或更高平台

## 实验内容：

## 实验一： 1、通过ADC模块进行传感器温度读取显示

### 1）设计AD模块

#### #1目的：通过设计AD模块来将MCU的AD模块的接收温度模拟量的通道值发送给C#,让C#在接收到了模拟量后进行数字量转换

#2问题：

##### 问题1）如何正确将温度模拟量发送给C#

@方法：通过AD模块中的函数 adc\_read（通道号）来采集每个AD模块的输入通道的值，存放到数组ADCResult[]里面;读取温度模拟量对应的通道的值存放到数组的ADCResult[15]里面。

@源码：

//A组初始化（通道组、单端输入，采样精度，硬件均值）

adc\_init(MUXSEL\_A, 0, 16, SAMPLE32);

//加头标志

ADCResult[0] = 0x1122;

//采集数据

ADCResult[1] = adc\_read(0);

**for** (i = 2; i <= 8; i++)

ADCResult[i] = adc\_read(i + 1);

**for** (i = 9; i <= 13; i++)

ADCResult[i] = adc\_read(i + 2);

ADCResult[14] = adc\_read(23);

ADCResult[15] = adc\_read(26); //芯片温度采集通道

//B组初始化（通道组、单端输入，采样精度，硬件均值）

adc\_init(MUXSEL\_B, 0, 16, SAMPLE32);

//采样

**for** (i = 16; i <= 20; i++)

ADCResult[i] = adc\_read(i - 12);

//加末尾标志

ADCResult[20] = 0x8899;

//将采集的A/D值通过串口发送到PC

uart\_sendN(UART\_TEST, 42, (uint\_8\*) ADCResult);

Delay\_ms(50);

### 2）设计C#模块

#### #目的：串口接收MCU传递过来的模拟量进行处理转成温度的数字量，进行字符转换输出到对应的text控件里面

#### #问题：

##### 1）如何接收对应的温度通道的模拟量并处理成数字量；

@方法：

因为在MCU发送每个通道的模拟量组成的模拟量数组时，对应通道的模拟量存放在对应位置的数组元素里，位置已经一一确定，所以C#只要按照正确的数组元素的顺序来接受传递的模拟量，则可以按照温度模拟量存放的位置来正确的接收模拟量；在进行处理即可；

2）

@源码：

private void SCIPort\_DataReceived(object sender, System.IO.Ports.SerialDataReceivedEventArgs e)

{

byte len = 0; //标记接收的数据的长度

Int32 i;

Thread.Sleep(50);

sci.SCIReceiveData(SCIPort, ref PublicVar.g\_ReceiveByteArray, ref len);

if ((len == 42) && (PublicVar.g\_ReceiveByteArray[0] == 0x22) && (PublicVar.g\_ReceiveByteArray[41] == 0x88))

{

//将串口的数据显示到文本

//通道A

i = PublicVar.g\_ReceiveByteArray[3] \* 256 + PublicVar.g\_ReceiveByteArray[2];

SCIUpdateRevtxtbox(txtchannel0, Convert.ToString(i, 16));

。.

.

.

.

i = PublicVar.g\_ReceiveByteArray[17] \* 256 + PublicVar.g\_ReceiveByteArray[16];

SCIUpdateRevtxtbox(txtchannel9, Convert.ToString(i, 16));

//接受电压值

double temp = Convert.ToInt32(i \* 3300) / 65535.0;//按照比例转换成数字量

.

.

.

.

i = PublicVar.g\_ReceiveByteArray[39] \* 256 + PublicVar.g\_ReceiveByteArray[38];

SCIUpdateRevtxtbox(txtchannel7b, Convert.ToString(i, 16));

//温度（26通道）数据处理

i = PublicVar.g\_ReceiveByteArray[31] \* 256 + PublicVar.g\_ReceiveByteArray[30];

double VTemp = Convert.ToInt32(i \* 3300) / 65535.0;

//将电压值转换为温度值

//719为25度时的标准电压采样值，1.715为温度传感器的电压/温度曲线（近似线性关系）的斜率

double t = 25 - (VTemp - 719) / 1.715;

//绘制坐标

double x = (double)new XDate(DateTime.Now);

double y = t;

list.Add(x, t);

String strT = t.ToString().Substring(0, 5);

SCIUpdateRevtxtbox\_2(textBoxTemperature, strT);//显示温度

Refresh(zedGraphControl1);

this.TSSLState.Text = "过程提示:数据接收!";

}

//接收数据失败

else

{

//sci.SCIReceInt(SCIPort, 1);//设置产生接收中断的字节数

this.TSSLState.Text = "过程提示:数据接收!";

}

//timer1.Enabled = true;

}

## 实验二： 2、通过DAC模块进行三角波的C#端显示

### 1）设计DA/AD模块

#### #目的：通过编写三角波的数字量变化代码，每次for循环将变化后的的数字量通过DA模块转化为模拟量（电压值从PTE\_30端口输出），选取和时AD模块的通道进行模拟量接受，存放到数组ADCResult[]内部通过MCU传递给C#

#### #问题：

##### 问题1）如何正确的将每次变化的数字量转化为模拟量

@方法：使用DA模块的dac\_convert（）函数进行转换，从唯一的PTE\_30输出，选取AD模块的通道9，进行采集对应的模拟量输出（在C#端体现）

@源码：

for(; ;){

dac\_convert(VReference);

x++;

//根据标志位，设置小灯慢慢点亮或慢慢熄灭

//三角波

**if** (light\_flag == 1)

VReference += 10;

**else** **if** (light\_flag == 0)

VReference -= 10;

//VReference限幅，并反转灯点亮或熄灭的标志位

**if** (VReference >= 2600)

light\_flag = 0;

**if** (VReference <= 2200)

light\_flag = 1;

//A组初始化（通道组、单端输入，采样精度，硬件均值）

adc\_init(MUXSEL\_A, 0, 16, SAMPLE32);

//加头标志

ADCResult[0] = 0x1122;

//采集数据

ADCResult[1] = adc\_read(0);

**for** (i = 2; i <= 8; i++)

ADCResult[i] = adc\_read(i + 1);

**for** (i = 9; i <= 13; i++)

ADCResult[i] = adc\_read(i + 2);

ADCResult[14] = adc\_read(23);

ADCResult[15] = adc\_read(26); //芯片温度采集通道

//B组初始化（通道组、单端输入，采样精度，硬件均值）

adc\_init(MUXSEL\_B, 0, 16, SAMPLE32);

//采样

**for** (i = 16; i <= 20; i++)

ADCResult[i] = adc\_read(i - 12);

//加末尾标志

ADCResult[20] = 0x8899;

//将采集的A/D值通过串口发送到PC

uart\_sendN(UART\_TEST, 42, (uint\_8\*) ADCResult);

Delay\_ms(50);

}

### 2)C#模块的设计

@方法：和上面的步骤类似，只是使用的时我们和PTE\_30相连的通道引脚的模拟量，直接用作画图函数list(x,y)的纵坐标的值；

@源码：

i = PublicVar.g\_ReceiveByteArray[17] \* 256 + PublicVar.g\_ReceiveByteArray[16];

SCIUpdateRevtxtbox(txtchannel9, Convert.ToString(i, 16));

//接受电压值

double temp = Convert.ToInt32(i \* 3300) / 65535.0;//按照比例转换成数字量

..

.

.

.

list.Add(x, t);//画图

## 实验三：通过编写温湿度传感器DHT11和MCU之间的数据传输时的时序程序来控制温度数据的输入到MCU

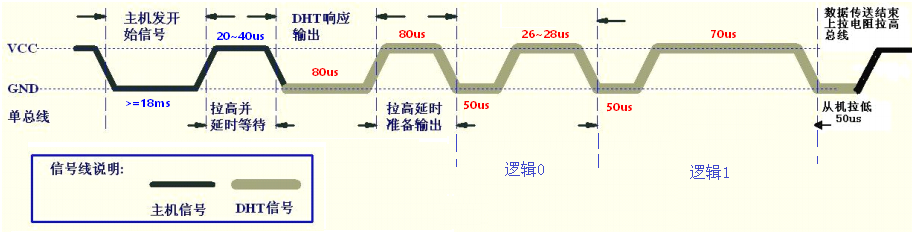
### 1）设计DHT11模块

#### #目的：初始化I/O引脚，并设计MCU和传感器传输数据的时序，已完成一个完整的数据传输周期

#### #问题

##### 问题1）：如何设计时序

@方法：通过上网查询DHT11的读取时序图，学习DHT和主机进行通信是的时序，从而在MCU端进行编程完成相关的引脚的输入输出功能转换，并控制相应引脚功能转换的时间；



@源码：

/\*

\* dht11.c

\*

\* Created on: 2019年12月25日

\* Author: 刘乾

\*/

//file:dht11.c

//author:lulipro

//date:2019-5-5

#include "dht11.h"

//包含引脚设置模块，延时模块

#include "gpio.h"

#include "delay.h"

#include "adc.h"

#include "common.h"

#include <string.h> //for memset()

static uint\_16 DHT11\_dataPin\_; //驱动DHT11的数据线引脚

static uint8\_t DHT11\_recvData\_[5]; //存放 从DHT11读取的数据的缓冲数组，40个bit

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

\*作用：初始化

\*参数：pin，Arduino驱动DHT11的总线使用的IO引脚

\*返回：无

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

void DHT11\_init(uint\_16 pin)

{

//初始IO引脚号赋给静态全局变量DHT11\_dataPin\_

DHT11\_dataPin\_ = pin;

}

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

\*作用：读取一次DHT11的传感器数据

\*参数：temperature，指向存放温度数据的指针；humidity，指向存放湿度数据的指针

\*返回：函数的执行状态：DHT11\_OK，DHT11\_TIMEOUT,DHT11\_CHECKEROR

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

int DHT11\_read(uint8\_t\* temperature,uint8\_t\* humidity )

{

uint8\_t time\_cnt;

uint8\_t i;

uint8\_t bit\_position;

memset(DHT11\_recvData\_,0,5); //缓冲数组内容清0

/\*--------------主机发送起始信号----------------\*/

gpio\_init(DHT11\_dataPin\_,1,0);

gpio\_set(DHT11\_dataPin\_, 0); // 主机将总线拉低（时间>=18ms），使得DHT11能够接收到起始信号。

delay\_us(18000); //至少 18 ms

gpio\_set(DHT11\_dataPin\_, 1); // 主机将总线拉高，代表起始信号结束。

delay\_us(35); //延时20~40us

/\*--------------引脚配置为输入模式，准备接收传感器回传的数据-------------------\*/

gpio\_init(DHT11\_dataPin\_,0,0); //配置为输入模式

//DHT11将总线拉低至少80us，作为DHT11的响应信号（ACK）。

time\_cnt=0;

while(0 == gpio\_get(DHT11\_dataPin\_))

{

delay\_us(5);

++time\_cnt;

if(time\_cnt > 16) return DHT11\_TIMEOUT;

}

//DHT11将总线拉高至少80us，为发送传感器数据做准备。

time\_cnt=0;

while(1 == gpio\_get(DHT11\_dataPin\_))

{

delay\_us(5);

++time\_cnt;

if(time\_cnt > 16) return DHT11\_TIMEOUT;

}

/\*-------------------DHT11数据帧的接收和解析------------------\*/

for( i=0 ; i < 40 ; ++i )

{

time\_cnt = 0;

while( 0 == gpio\_get(DHT11\_dataPin\_) ) //拉低50us作为bit信号的起始标志

{

time\_cnt++;

delay\_us(5);

if(time\_cnt>10) return DHT11\_TIMEOUT;

}

time\_cnt = 0;

while( 1 == gpio\_get(DHT11\_dataPin\_) ) //拉高。持续26~28us表示bit0，持续70us表示bit1

{

time\_cnt++;

delay\_us(5);

if(time\_cnt>14) return DHT11\_TIMEOUT;

}

if(time\_cnt>6){ //说明是bit1

bit\_position = 7 - i%8;

DHT11\_recvData\_[i/8] |= (uint8\_t)(1<<bit\_position);

}

}

//------------检验和的比对-------------

i = (uint8\_t)(DHT11\_recvData\_[0] + DHT11\_recvData\_[1] + DHT11\_recvData\_[2] + DHT11\_recvData\_[3]) ;

if(i != DHT11\_recvData\_[4] ) return DHT11\_CHECKERROR;

\*humidity = DHT11\_recvData\_[0]; //回传湿度数据

\*temperature = DHT11\_recvData\_[2]; //回传温度数据

return DHT11\_OK;

}

### 2）设计main函数

#### #目的：初始化相关功能模块，完成DHT11的循环调用

#### #问题：

##### 问题1）：如何通过保存每次读取的温度数据

@方法:设置全局变量 uint8\_t temperature; uint8\_t humidity; 当作参数，每次传入其指针给DHT11\_read(&temperature, &humidity);函数，每次读取后根据此函数的返回标志判断是都读取成功，若读取成功，则通过串口发送发送给C#端；

@源码：

//说明见工程文件夹下的Doc文件夹内Readme.txt文件

//=====================================================================

#include "includes.h" //包含总头文件

extern void Delay\_ms(uint16\_t u16ms);

//存放温湿度值的变量

uint8\_t temperature;

uint8\_t humidity;

uint\_8 Tem[5];

uint\_8 Hum[5];

int flag=0;

int main(void) {

//1. 声明主函数使用的变量

//2. 关总中断

DISABLE\_INTERRUPTS;

//3. 初始化外设模块

light\_init(LIGHT\_BLUE, LIGHT\_OFF); //蓝灯初始化

uart\_init(UART\_1, 115200); //使能串口1，波特率为9600

uart\_init(UART\_2, 9600); //使能串口2，波特率为9600

delay\_init(0);

DHT11\_init(DHT\_DATA); //初始化接口为PTB\_10

//4. 给有关变量赋初值

//5. 使能模块中断

uart\_enable\_re\_int(UART\_1); //使能串口1接收中断

uart\_enable\_re\_int(UART\_2); //使能串口2接收中断

//6. 开总中断

ENABLE\_INTERRUPTS;

//进入主循环

//主循环开始==================================================

while (1) {

flag=DHT11\_read(&temperature, &humidity);

switch (flag) {

case DHT11\_OK:

Tem[0] = temperature / 10 + '0';

Tem[1] = temperature % 10 + '0';

Hum[0] = humidity / 10 + '0';

Hum[1] = humidity % 10 + '0';

uart\_send\_string(UART\_1, "temperature ");

uart\_sendN(UART\_1, 2, Tem);

uart\_send\_string(UART\_1, "°C ");

uart\_send\_string(UART\_1, "humidity ");

uart\_sendN(UART\_1, 2, Hum);

uart\_send\_string(UART\_1, "%RF ");

break;

case DHT11\_TIMEOUT:

uart\_send\_string(UART\_1, "time out ");

break;

case DHT11\_CHECKERROR:

uart\_send\_string(UART\_1, "check error ");

break;

default:

uart\_send\_string(UART\_1, "unknown error ");

break;

}

Delay\_ms(1000); //每次读取后延时1s后下一次读取

} //主循环end\_for

//主循环结束==================================================

}

## 实验总结：

其实我认为这次抽到41题，和写40给我带来了极大的启示性意义！

假如我只抽到了前面几个简单的只用到ad通道的传感器，我估计我不是很有可能接触到自己写时序的这个小任务，也很难获得到宝贵的程序设计思想和MCU和传感器之间的简单的总线时序的编写经验。

一开始我认为这是一项很艰难的任务，但是通过这次我发现，问题不去尽心尽力解决之前，说难是没有太大意义的，把他学会了才是最重要的。

还有就是对待学习的心态，完成实验是一方面，更重要的是自己在完成实验当中获得了怎么样的成长。

# 嵌入式手扎九 USB+(SPI/UART)

说明：这次手札是和课程报告一起在写的，说来惭愧，直到课程设计的时候才接触到USB编程，双机通信的IIC也只单独使用了UART进行了测试；并没有和SPI一样用USB进行主机通信；但对USB编程也有了一定的学习认知；

## 实验目的：

1、通过USB+SPI组合进行板间通信和PC机之间信息传递

## 实验环境：

1）PC机一台，KL25开发板一片，导线若干

2) KDS开发环境，VS2017或更高平台

## 实验内容：

## 实验一： 1、通过USB+SPI组合进行板间通信和PC机之间信息传递

### 1）USB模块的通信原理

#### 一、USB通信 两个模块：USB主机/USB从机

USB主机：主要负责发送： 接收从机数据命令：*cmdINTESTDATA* 发送数据给从机命令：*cmdOUTTESTDATA* 给USB从机来控制主从机之间的数据传输；

USB从机：主要负责接收主机发送过来的命令，接收主机发送的数据或者向主机发送数据；

注：本实验统一让MCU当作USB从机，PC端当作USB主机，进行SPI+USB板间通信；

#### 二、USB通讯 发送数据/接收数据 的过程分析：

##### 1）USB发送数据



**@源码：**

1）USB发送数据到USB从机函数

///函数名:sendData

///功 能:通过USB发送一个字节数组

///参 数:(1)sendData:存放需要发送的字节数组

/// (2)count:需要发送的字节数量，从数组第0个元素算起

///返 回:布尔值,当发送成功后,返回True，否则返回False

/// ----------------------------------------------------------------

public bool sendData(byte[] sendData, int count)

{

ErrorCode ec = ErrorCode.None;

int bytesWritten;

try

{

ec = writer.Write(sendData, 0, count, 100, out bytesWritten);

if (ec != ErrorCode.None)

throw new Exception(ec.ToString());

return true;

}

catch (Exception ex)

{

Console.WriteLine("Error:" + ex.Message);

return false;

}

}

///函数名:sendData

///功 能:通过USB发送一个字节

///参 数:(1)sendData:存放需要发送的一个字节

///返 回:布尔值,当发送成功后,返回True，否则返回False

/// ----------------------------------------------------------------

public bool sendData(byte sendData)

{

ErrorCode ec = ErrorCode.None;

int bytesWritten;

try

{

ec = writer.Write(sendData, 100, out bytesWritten);

if (ec != ErrorCode.None)

throw new Exception(ec.ToString());

return true;

}

catch (Exception ex)

{

Console.WriteLine("Error:" + ex.Message);

return false;

}

}

##### 2）USB接收数据





**@源码**

1）USB主机接收USB从机的函数

/// ----------------------------------------------------------------

///函数名:recvString

///功 能:通过USB接收一个字符换

///参 数:无

///返 回:从USB接收到的字符串

/// ----------------------------------------------------------------

public string recvString()

{

byte[] readBuffer = new byte[64];

int bytesRead;

reader.Read(readBuffer, 100, out bytesRead);

return Encoding.Default.GetString(readBuffer, 0, bytesRead);

}

/// ----------------------------------------------------------------

///函数名:recvByte

///功 能:通过USB接收一个字节数组

///参 数:无

///返 回:从USB接收到的字节数组

/// ----------------------------------------------------------------

public byte[] recvByte()

{

byte[] readBuffer = new byte[64];

int bytesRead;

reader.Read(readBuffer, 100, out bytesRead);

if (bytesRead > 0)

{

byte[] recvDataBuffer = new byte[bytesRead];

Array.Copy(readBuffer, recvDataBuffer, bytesRead);

return recvDataBuffer;

}

else

{

return null;

}

}

### 2）SPI模块分析

#### 一、串行外设接口SPI的通用基础知识：

串行外设接口SPI是一种同步串行通信接口，用于微处理器和外围扩展芯片指尖的串行连接，发展成为一种工业标准。

（1）主机与从机概念：一个SPI系统，由一个主机和一个或多个从机构成，主机启动一个与从机的同步通信，从而完成数据交换。

（2）主出从入引脚MOSI与主入从出引脚MISP

主出从入引脚是主机输出，从机出入数据线。

主入从出引脚是主机输入，从机输出数据线。

（3）SPI串行时钟引脚：SPI串行时钟引脚用于控制主机与从机之间的数据传输。

（4）时钟极性与时钟相位：时钟极性表示时钟信号在空闲时是高电平还是低电平。

（5）从机选择引脚SS：一些芯片带有从机选择引脚SS，也称为片选引脚。

#### 二、传输原理：

从主机CPU发出启动传输信号开始，将要传送的数据装入8位移位寄存器，并同时产生8个时钟信号，以此从SCK引脚送出，在SCK信号的控制下，主机中8位移位寄存器中的数据依次从MOSI引脚送出，到从机的MOSI引脚后送入它的8位移位寄存器。

SPI的时序：

（1）空闲电平低电平，上升沿取数

（2）空闲电平低电平，下降沿取数

（3）空闲电平高电平，下降沿取数

（4）空闲电平高电平，上升沿取数

#### 三、双向通信的两种方式

##### 1）八线双向

###### @1：设计思路：

这种方式是最简单的，两个MCU同时充当主机和从机，发送数据只用到每个板子的主机模块；接收数据只用到每个板子的从机模块

###### @2：源码：

#1：main.c

**#include** "includes.h" //包含总头文件

//USB序列号

//注意：修改USB序列号时，要将设备名的第一个字节数据长度字段进行修改

uint\_8 Serial\_String[] = {

0x10, //设备名长度字段(包含该字段本身长度)

0x03, //设备名索引

//以下是用户自定义设备名

'H', 0x00, 'W', 0x00, 'W', 0x00, '\_', 0x00, 'U', 0x00, 'S', 0x00, 'B',

0x00, };

//==========================================================

//函数名称：Delay\_ms

//函数返回：无

//参数说明：无

//功能概要：延时 - 毫秒级

//==========================================================

**void** **Delay\_ms**(uint16\_t u16ms) {

uint32\_t u32ctr;

**for** (u32ctr = 0; u32ctr < ((48000 / 10) \* u16ms); u32ctr++) {

**asm** ("NOP");

}

}

**int** **main**(**void**) {

uint\_32 mRuncount; //主循环计数器

// uint\_8 i;

uint\_8 j;

//2. 关总中断

DISABLE\_INTERRUPTS;

//3. 初始化外设模块

light\_init(RUN\_LIGHT\_BLUE, LIGHT\_ON); //蓝灯初始化

uart\_init(UART\_1, 9600); //串口1初始化,波特率9600

//把SPI0初始化为主机,波特率6000，时钟极性0，时钟相位0

SPI\_init(SPI\_0, 1, 6000, 0, 0);

//把SPI1初始化为从机,波特率6000，时钟极性0，时钟相位0

SPI\_init(SPI\_1, 0, 6000, 0, 0);

//USB设备初始化

usb\_init(Serial\_String);

//4. 给有关变量赋初值

mRuncount = 0; //主循环计数器

g\_USBRecvLength = 0; //USB接收数据长度初始化

g\_USBSendLength = 0; //USB发送数据长度初始化

g\_USBSendFlag = 0; //USB执行发送标志初始化

//5. 使能模块中断

uart\_enable\_re\_int(UART\_1); //uart接收中断

SPI\_enable\_re\_int(SPI\_1); //从机SPI\_1的接收中断

//6. 开总中断

ENABLE\_INTERRUPTS;

//进入主循环

//主循环开始===========================================================

**for** (;;) {

//运行指示灯（RUN\_LIGHT）闪烁---------------------------------------------

mRuncount++; //主循环次数计数器+1

**if** (mRuncount >= RUN\_COUNTER\_MAX) //主循环次数计数器大于设定的宏常数

{

mRuncount = 0; //主循环次数计数器清零

}

//以下加入用户程序--------------------------------------------------------

//USB主机向USB设备发送要数据命令

**if** (g\_USBRecv[0] == *cmdINTESTDATA* && g\_USBSendLength != 0) {

g\_USBRecv[0] = *cmdNULL*;

g\_USBSend[0] = *cmdINTESTDATA*;

g\_USBSendFlag = 1;

}

//USB主机发送数据，写入到发送数组中，以便主机要数据时进行发送

**else** **if** (g\_USBRecv[0] == *cmdOUTTESTDATA* && g\_USBRecvLength != 0) {

//禁止USB中断

disable\_irq(USB\_INTERRUPT\_IRQ);

//将接收数据的内容传输到发送缓存数组中进行保存，待发送

**for** (j = 0; j < g\_USBRecvLength; j++) {

spi\_Send[j + 2] = g\_USBRecv[j]; //Recv数组存放到SPI缓存数组中

}

//组成帧的格式

spi\_Send[0] = 'P';

spi\_Send[1] = g\_USBRecvLength + '0';

spi\_Send[g\_USBRecvLength + 2] = 'C';

spi\_SendLength = g\_USBRecvLength + 3;

g\_USBRecv[0] = *cmdNULL*;

g\_USBRecvLength = 0;

//组帧成功，发送操作，再SPI从机的接收中断中执行

//注释掉SPI主机发送程序

**int** m = 0;

**for** (m = 0; m < spi\_SendLength; m++) {

SPI\_send1(SPI\_0, spi\_Send[m]);

uart\_send1(UART\_1, spi\_Send[m]);

}

//使能USB中断

enable\_irq(USB\_INTERRUPT\_IRQ);

}

//将spi从机接受的内容传递给usb\_send[]内，准备发送给USB主机

**if** (spi\_flag == 1) {

uint\_8 t = 0;

**for** (t = 0; t < spi\_RecvLength ; t++) {

g\_USBSend[t] = spi\_Recv[t + 2];

}

g\_USBSendLength = spi\_RecvLength;

spi\_flag = 0;

}

} //主循环end\_for

//主循环结束=====================================================

}

#2：isr.c

**void** **USB0\_IRQHandler**(**void**) {

uint\_8 isr\_type;

DISABLE\_INTERRUPTS; //关总中断

//1. 获取中断类型

isr\_type = usb\_get\_isr();

//2. 若不是令牌完成中断，调用相应处理程序

**if** (isr\_type != USB\_TOKDNE\_INT) {

usb\_isr\_handler(isr\_type); //调用非令牌完成中断的处理程序

**goto** USB0\_IRQ\_exit;

}

//3. 是令牌完成中断，执行块数据传输或设备枚举

//(3.1)清收发ODD区，并指定EVEN区

FLAG\_SET(USB\_CTL\_ODDRST\_SHIFT, USB0\_CTL);

//(3.2)若是设备枚举请求，进行设备枚举

**if** ((USB0\_STAT >> 4) == 0) {

usb\_enumerate();

**goto** USB0\_IRQ\_exit;

}

//(3.3)若不是设备枚举请求，进行数据收发操作

//从端点2发送数据至主机

**if** ((USB0\_STAT & 0xF8) == mEP2\_IN) {

//若执行发送标志被置1，则执行发送数据函数

**if** (g\_USBSendFlag == 1) {

//调用发送数据处理函数

usb\_send(g\_USBSend, &g\_USBSendLength);

**if** (g\_USBSendLength == 0) //剩余未发送的数据长度为0

{

g\_USBSendFlag = 0; //发送完成，执行发送标志清0

}

**goto** USB0\_IRQ\_exit;

}

//若无数据需要发送，则向USB主机发送一个确认包

BDTtable[*bEP2IN\_ODD*].Cnt = 0;

vEP2State ^= 0x40;

BDTtable[*bEP2IN\_ODD*].Stat.\_byte = vEP2State;

//从端点3接收主机发来的数据

} **else** **if** ((USB0\_STAT & 0xF8) == mEP3\_OUT) {

usb\_recv(g\_USBRecv, &g\_USBRecvLength);

send\_flag = 1;

}

USB0\_IRQ\_exit:

//4. 清除中断标志，结束中断处理

FLAG\_SET(USB\_ISTAT\_TOKDNE\_SHIFT, USB0\_ISTAT);

ENABLE\_INTERRUPTS; //开总中断

**return**;

}

//=====================================================================

//函数名: SPI1\_IRQHandler

//功 能: SPI1接收中断服务程序

//参 数: 无

//返 回: 无

//说 明: 需要启动中断并注册才可使用

//=====================================================================

**void** **SPI1\_IRQHandler**(**void**) {

uint\_8 redata;

DISABLE\_INTERRUPTS;

redata = SPI\_receive1(SPI\_1); //接收主机发送过来的一个字节数据。

**if** (CreateFrame(redata, spi\_Recv) != 0) {

//组帧成功

spi\_RecvLength = spi\_Recv[1];

spi\_flag = 1;

}

ENABLE\_INTERRUPTS;

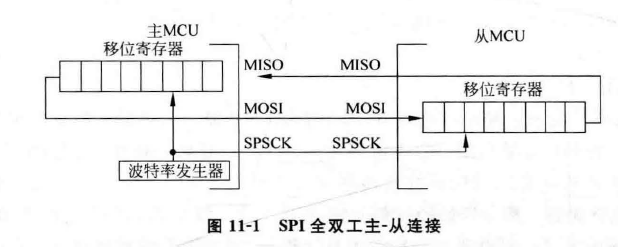
}

##### 2）四线双向

###### @1：设计思路：

这种思路是有一定难度的，但是最有价值的，两个MCU设定唯一的主机和从机；

通过利用SPI双机通信的特点来进行从机向主机的信息传递；



主机发送从机过程跟上面的八线双向相同，但是从机发往主机则利用主机发送的时序信号‘0’来将从机移位寄存器中的待发送数据按字节逐次发送给主机；

###### @2：问题解决

#1：（主机/从机--接收/发送） 数据的信号控制

通过定义全局变量uint\_8 rec\_flag(主机); uint\_8 spi\_flag(从机)来控制；每次接收到USB发送数据后通过USB中断设置为1；

1）主机rec\_flag信号设置为1代表：

1、不提供从机发主机的时序信号‘0’

2、主机接收中断开始判断从机移位寄存器发送过来的内容

2）从机spi\_flag信号设置为1代表：

1、从机开始根据主机发送的时序信号开始将数据内容逐字节发送到主机

3）源码

#1 USB主机

**void** **SPI0\_IRQHandler**(**void**) {

//仅在从机发送数据时 rec\_flag == 0 调用

uint\_8 redata;

DISABLE\_INTERRUPTS;

redata = SPI\_read1(SPI\_0); //接收主机发送过来的一个字节数据。

**if** (rec\_flag == 0) {//SPI主机不处于发送数据时间

**if** (redata == '0') { //若是时序信号则不进行操作

} **else** **if** (CreateFrame(redata, spi\_Recv) != 0) {

//组帧成功

spi\_RecvLength = spi\_Recv[1];

spi\_flag = 1;

}

}

ENABLE\_INTERRUPTS;

}

#2 USB从机

**void** **SPI1\_IRQHandler**(**void**) {

uint\_8 redata;

DISABLE\_INTERRUPTS;

redata = SPI\_receive1(SPI\_1); //接收主机发送过来的一个字节数据。

//SPI从机发送模块

**if** (send\_flag == 1) {

**if** (redata == '0') {

**int** m = 0;

**if** (m < spi\_SendLength) {

//收到一个时序信号，后将待发送内容转入移位寄存器

//待一个时序信号发送过来后，循环发送至SPI主机的移位寄存器中

SPI1\_D = spi\_Send[m];

m++;

}

**if** (m == spi\_SendLength) {

send\_flag = 0;//发送完毕后，SPI从机发送标志重新清零

}

}

}

//SPI从机接收模块

**if** (CreateFrame(redata, spi\_Recv) != 0) {

//组帧成功

spi\_RecvLength = spi\_Recv[1];

spi\_flag = 1;

}

ENABLE\_INTERRUPTS;

}

#2：主机发往从机的 时序信号控制

1）通过主机main.c中对的rec\_flag标志的判断：

为1则不发送时序标志；为0则发送时序标志

2）源码

**for** (;;) {

//运行指示灯（RUN\_LIGHT）闪烁---------------------------------------------

mRuncount++; //主循环次数计数器+1

**if** (mRuncount >= RUN\_COUNTER\_MAX) //主循环次数计数器大于设定的宏常数

{

mRuncount = 0; //主循环次数计数器清零

}

//以下加入用户程序--------------------------------------------------------

//SPI主机只要不发送数据，就一直为SPI从机提供时序信号‘0’，根据rec\_flag标志判断

**if** (rec\_flag == 0) {

SPI\_send1(SPI\_0, '0');

}

//USB主机向USB设备发送要数据命令

**if** (g\_USBRecv[0] == *cmdINTESTDATA* && g\_USBSendLength != 0) {

g\_USBRecv[0] = *cmdNULL*;

g\_USBSend[0] = *cmdINTESTDATA*;

g\_USBSendFlag = 1;

}

//USB主机发送数据，写入到发送数组中，以便主机要数据时进行发送

**else** **if** (g\_USBRecv[0] == *cmdOUTTESTDATA* && g\_USBRecvLength != 0) {

//禁止USB中断

disable\_irq(USB\_INTERRUPT\_IRQ);

//将接收数据的内容传输到发送缓存数组中进行保存，待发送

**for** (j = 0; j < g\_USBRecvLength; j++) {

spi\_Send[j + 2] = g\_USBRecv[j]; //Recv数组存放到SPI缓存数组中

}

//组成帧的格式

spi\_Send[0] = 'P';

spi\_Send[1] = g\_USBRecvLength + '0';

spi\_Send[g\_USBRecvLength + 2] = 'C';

spi\_SendLength = g\_USBRecvLength + 3;

g\_USBRecv[0] = *cmdNULL*;

g\_USBRecvLength = 0;

**int** m = 0;

**for** (m = 0; m < spi\_SendLength; m++) {

SPI\_send1(SPI\_0, spi\_Send[m]);

}

//发送一个时序信号，清空SPI从机移位寄存器的原有的内容

//保证下一次SPI主机中断接收的非‘0’ 内容一定是SPI从机发送的有效数据

SPI\_send1(SPI\_0, '0');

//SPI主机发送数据完毕，重新设置SPI主机接收数据信号标志为0；

rec\_flag = 0;

//使能USB中断

enable\_irq(USB\_INTERRUPT\_IRQ);

}

//将spi主机接受的内容传递给usb\_send[]内，准备发送给USB主机

**if** (spi\_flag == 1) {

uint\_8 t = 0;

**for** (t = 0; t < spi\_RecvLength; t++) {

g\_USBSend[t] = spi\_Recv[t + 2];

}

g\_USBSendLength = spi\_RecvLength;

spi\_flag = 0;

}

} //主循环end\_for

#3: 从机对发送数据的组帧处理

1）方法：

跟SPI主机发送到SPI从机对发送数据的main.c里的组帧方法相同，只不过将发送到主机的SPI的操作放在了SPI从机的接收中断函数里；

## 实验总结

通过这次试验深入理解了USB通信和SPI主从双向通信的原理；一开始学习的时候确实花了相当大的时间。

不同于以往，这次实验综合了两个模块，需要对两个模块的整体结构有一个很好的了解才能着手设计整体的通讯。

特别是在主从双向通讯的时候，一开始只完成了八线双向，往后深入做四线双向通信的时候遇到相当大的困难，我通过向身边优秀的同学请教，一点一点的尝试，才学会了。

这算是学期最后一次的实验了，这学期的嵌入式课程带给我的成长是最大的，给了我一个机会去了解硬件的相关知识，同时培养了自己编程能力和学习习惯。

通过嵌入式课程让我学会了：在遇到不懂的问题时懂得不放弃，肯花时间钻研。以往遇到的问题往往是种类比较单一的，但是练习嵌入式的时候，出的问题五花八门，解决它更需要相当大的精力。

甚至到现在，我还有很多很多的知识没学到。但是我相信：有了这种经验，以后尽管遇到困难，我也不会退缩了，问题没能解决之前都很困难，最好的办法就是解决它，自然问题就不困难了。