ARM释放你的创造力

**概述部分**

2019年11月13日星期三正式开始学习嵌入式，希望自己不要中途放弃。

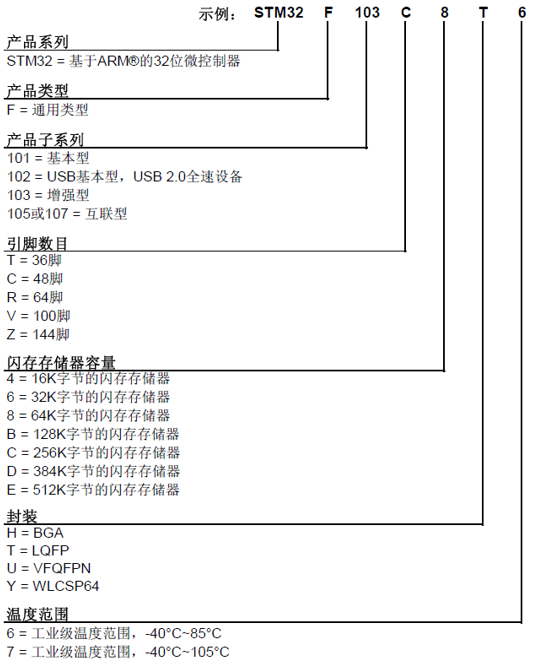
1. 关于MCU
2. stm32命名规则

图 二

1. CM3内核优点
2. 高性能的Cortex-M内核
3. 最佳的代码密度
4. 可预见的运行时间

中断控制器嵌在内核中，中断之间的间隔最少可达6个CPU周期

从低功耗模式唤醒只需要6个CPU周期

1. 改进的调试功能

串行单步调试和JTAG调试

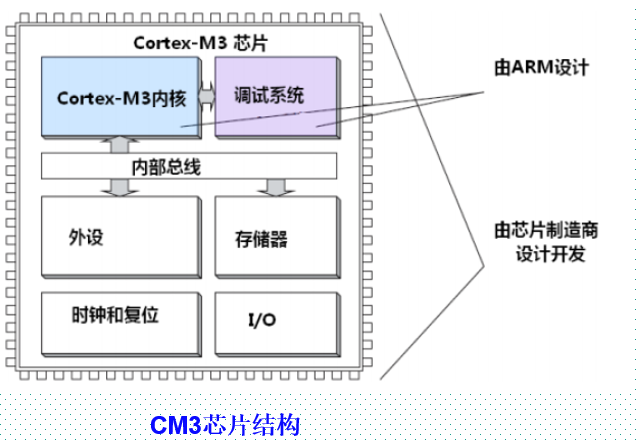
1. CM3芯片结构
2. STM32F103系列

图 二

1. 2V-3.6V供电电压
2. 5V的I/O电压容限
3. 出色的时钟安全模式
4. 带有唤醒功能的低功耗模式
5. 嵌入的RESET
6. -40/+85°C（工业级）
7. 芯片内部资源描述（以stm32f103zet6为例子）
8. 内核
9. IO口

大部分可以容忍5v（模拟通道除外），比如有器件直接5v接到IO口上，是不会烧掉IO口的。

1. 存储容量

主要关注flash和SRAM的容量大小

1. 芯片支持调试

SWD和JTAG，其中SWD只要两根数据线

1. PIN to PIN兼容吗？

zet6和zdt6是PIN to PIN兼容的，只有容量的大小不一样

1. 时钟、复位和电源管理
2. 2-3.6的电源和IO电压
3. 上电复位、掉电复位和可编程的电压监控
4. 强大的时钟系统
5. 4-16M外部告诉晶振
6. 8MHz的RC振荡器（这个不经常用到，如果外部晶振挂掉了，就会自动切换到该晶振）
7. 内部40KHz低速RC振荡器，看门狗电路
8. 内部锁相环（PLL，倍频），一般系统时钟都是外部或者内部高速时钟经过PLL倍频后得到的
9. 外部低速32.768K的晶振，主要做RTC的时钟源
10. 低功耗
11. 睡眠、停止和待机三种低功耗模式
12. 可用电池RTC和备份寄存器供电
13. 外设资源
14. AD
15. 三个12位的AD（多达21个外部测量通道）
16. 转换范围
17. 0-3.6（电源电压）
18. 内部通道可以用于内部温度测量

内部有温度传感器，直接可以通过内部通道测量温度

1. 内置参考电压
2. DA
3. 2个12位的DA
4. 12个DMA通道

直接寄存器访问，处理很多高速的数据的时候，可以用DMA，支持外设：定时器、ADC、DAC、SDIO、I2S、SPI、I2C、USART

1. 定时器（非常重要的东西）
2. 通用定时器
3. 基本定时器
4. 高级定时器
5. 系统定时器
6. 看门狗定时器
7. 通信接口
8. I2C
9. 串口
10. SPI
11. CAN2.0
12. USB FS
13. SDIO
14. stm32最小系统包括的内容
15. 供电
16. 复位
17. 时钟：外部晶振（2个）

有时候晶振内部自带旁路电容，因此就不需要再焊接旁路电容了

1. Boot启动模式选择

Boot0和Boot1接不同的电平，选择是下载模（下载代码）式还是执行模式（执行代码），如果有自动下载电路的话，就不需要手动的去设置Boot0和Boot1的高低电平了

1. 下载电路（串口/JTAG/SWD）

如果选择串口下载的话，所有的103芯片都是一样的，通过PA9和PA10下载。不过需要一个USB转TTL的电路。

1. 后备电池
2. stm32的电源引脚说明



图 stm32电源引脚说明图

1. 各种手册的使用方法
2. 芯片设计手册

编程的时候会用到，比如要把串口对应到哪个IO口等。

1. 芯片选型手册（芯片选型的方法）

通过该手册可以了解到芯片的内部资源，比如该芯片有多少个定时器、多少个ADC和DAC等等

在选用芯片的时候，通常不以Flash和RAM为标准，通常是考虑需要多少外设来确定芯片，比如需要多少定时器和ADC/DAC转换器等等。

如果在开发的过程中，发现所选取的芯片的容量过大或者过小且需要改变芯片的容量，那么需要选取引脚数兼容的芯片（pin-to-pin）。或者只有容量不一样，而其他的部分都一样的芯片。这是芯片选型需要注意的东西。

1. 关于开发环境keil
2. stm32f103在keil5中的开发环境配置方式
3. 新建工程模板的方法

略……

1. 调试工具
2. 下载程序的方式和对应原理
3. 寄存器配置
4. 基本内部外设
5. GPIO的输入和输出
6. 外部中断
7. 定时器
8. 串口

*注：理解了这四个外设，基本就入门的一款MCU*

1. 基本外设接口（一些通信协议）
2. SPI
3. IIC
4. WDG
5. FSMC
6. ADC/DAC
7. SDIO

*注：这些外设接口的功能原理对每个芯片几乎是一样的，对芯片而言就是加减法而已*

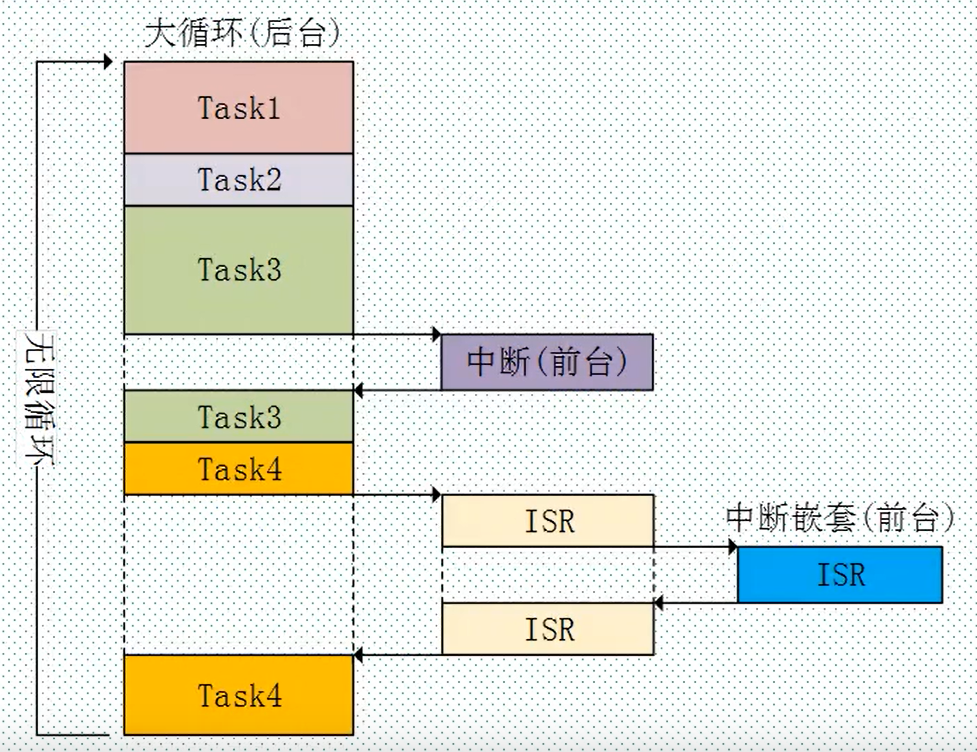
1. 高级功能（操作系统部分）
2. 前后台系统

图 前后台系统的特点

由上图可以看到，前后台系统的特点是没有实时性，只有执行完Task1、task2、task3才能执行task4。倘若在执行task1的过程中，突然要求立刻执行task4，那么前后台系统是不能满足这样的要求的。

在RTOS中，有任务优先级的概念，可以满足这样的要求。

1. RTOS

实时操作系统，给人的感觉是好像芯片有多个CPU，多任务管理实现了CPU资源的最大化利用。多任务管理有助于实现程序的模块化开发，能够实现负复杂的实时应用。

经典代表是ucos

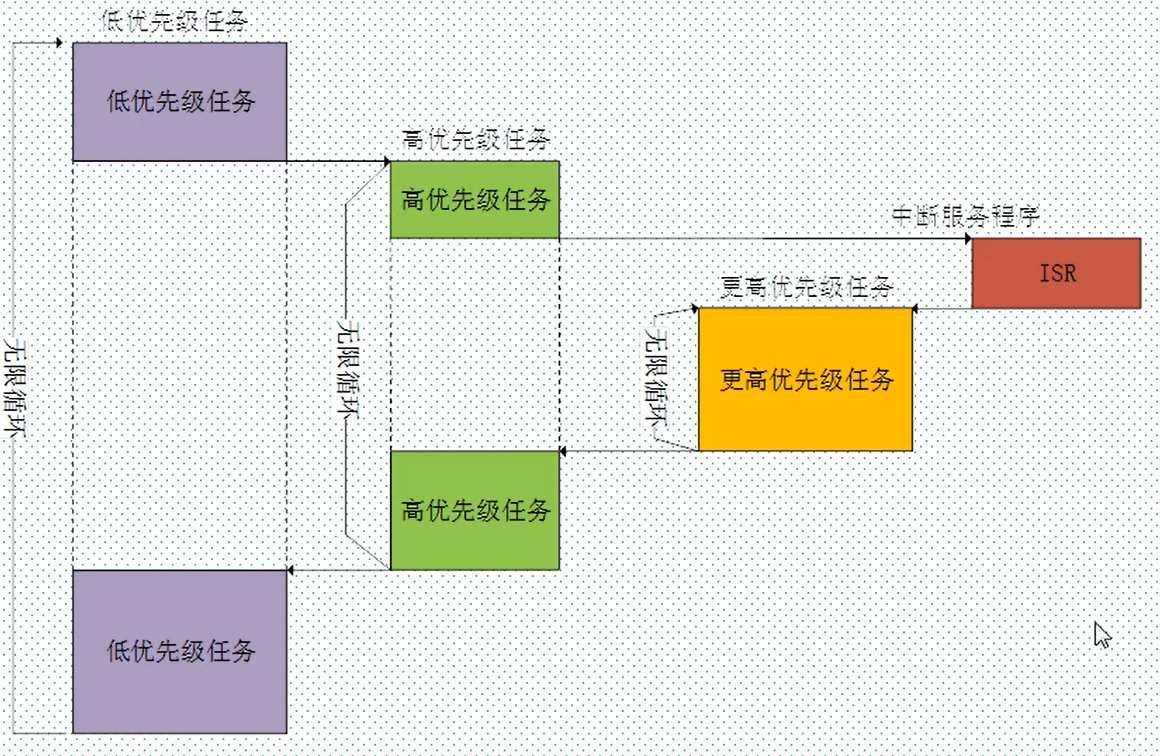


图 可剥夺的使用权

**实践部分**

1. GPIO
2. GPIO的工作方式

输入上拉、输入下拉、输入浮空、模拟输入、开漏输出、推挽输出、推挽式复用功能、开漏复用功能

另外，有FT标识的IO口可以容忍5V。大部分的IO口可以容忍5V

可以在程序中设置最大的翻转速度

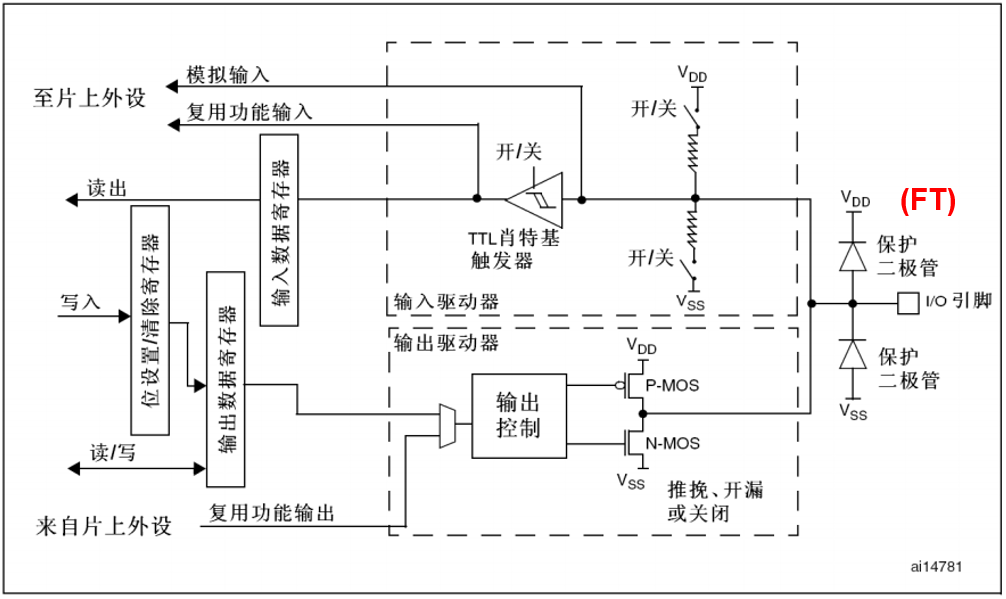
大致结构如图三

图 三

1. 输入上拉

利用上拉电阻，在没有输入信号的时候，把IO口的电位嵌位在高电平。如果输入上拉的话，那么输入信号为低电平才能有效。

输入上拉和浮空输入的方式差不多（参考输入浮空方式），它们从GPIO采集到的电平信号都要通过TTL肖基特触发器才能到输入数据寄存器。

1. 输入下拉

道理和输入上拉是一样的。

1. 输入浮空

输入浮空模式下，CPU可以通过输入数据寄存器读取到外部引脚的电平情况。其中，上下拉电阻都是断开的。

1. 模拟输入
2. 开漏输出
3. 推挽输出

可以输出强的高低电平。

所谓的电流从外部流入到单片机，实际是电流直接通过推挽电路中的mos管回流到地线（系统的最低电位）。

所谓的电流从单片机流出到外部，实际是电流从电源（系统的最高电位）通过推挽电路的mos管流到单片机的引脚。由此可见，单片机在通过引脚和外部交流信号的时候，是通过控制引线与高低电位之间的通断来实现的。

1. 推挽式复用功能（输出模式）
2. 开漏复用功能（输出模式）
3. GPIO的配置

每组GPIO都有7个寄存器（每个寄存器都是32位），由于CET6有7组IO口，故共有49组关于GPIO配置的寄存器。

这7个寄存器分别是：

2个32位的端口配置寄存器CRL和CRH

2个32位的IDR和ODR

1个32位的置位/复位寄存器BSRR

1个16位的复位寄存器BRR

1个32位的锁存寄存器CLOCK

最常用的是：

CRL、CRH、IDR、ODR

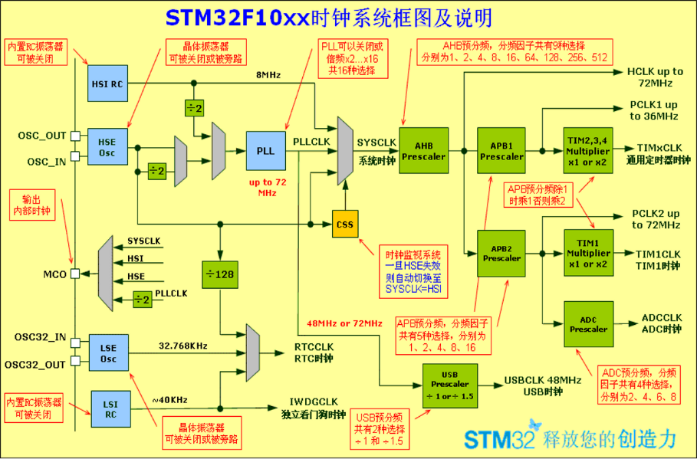
IDR寄存器是端口输入数据寄存器，只用的低16位。为只读寄存器，并且只能以16位的形式读出

ODR寄存器，端口输出数据寄存器，可读写，从该端口读出的数据可以用来判断端口的状态。而向该寄存器写数据，可以用来控制某个IO的输出高低电平。

CRH和CRL分别控制GPIO的高八位和低八位。CRH和CRL中的每四个位控制一个IO口。在这四个位中有分为MODE（两个位）和CNF0（两个位）其中MODE控制着输入模式下的翻转速率。

外设时钟使能寄存器。在使用一个外设之前，必须要先使能该外设的时钟。使能外设时钟是通过外设使能寄存器来完成的。

其实这些东西到底是怎样配置的，并没有确定的规律，需要查询相关的寄存器手册才行。

1. GPIO的翻转速率
2. 中断
3. 时钟
4. 时钟系统的框图如图四所示

图四

1. 时钟源，图中蓝色正方形
2. HSI RC

高速内部时钟，由RC振荡器产生，由于RC振荡器产生的时钟不够精确稳定，所以，该时钟频率是一个大约的数值。

可以作为系统时钟的一个选择。（灰色的五边形的意思是选择）

1. HSE OSC

高速外部时钟，通常是接外部晶振产生的。

1. PLL（2-16倍，通常作为系统的时钟来源）

锁相环，用来倍频（模电内容）

1. LSE OSC

外部低速时钟，也是由晶振产生震荡。

为stm32的实时时钟，为RTCCLK提供时钟源

1. LSI RC

内部RC振荡器产生的低速时钟。为独立看门狗电路提供时钟源。

1. MCO

对应单片机的PA8引脚，可以用来输出单片机的内部时钟，如下

SYSCLK

HIS

HSE

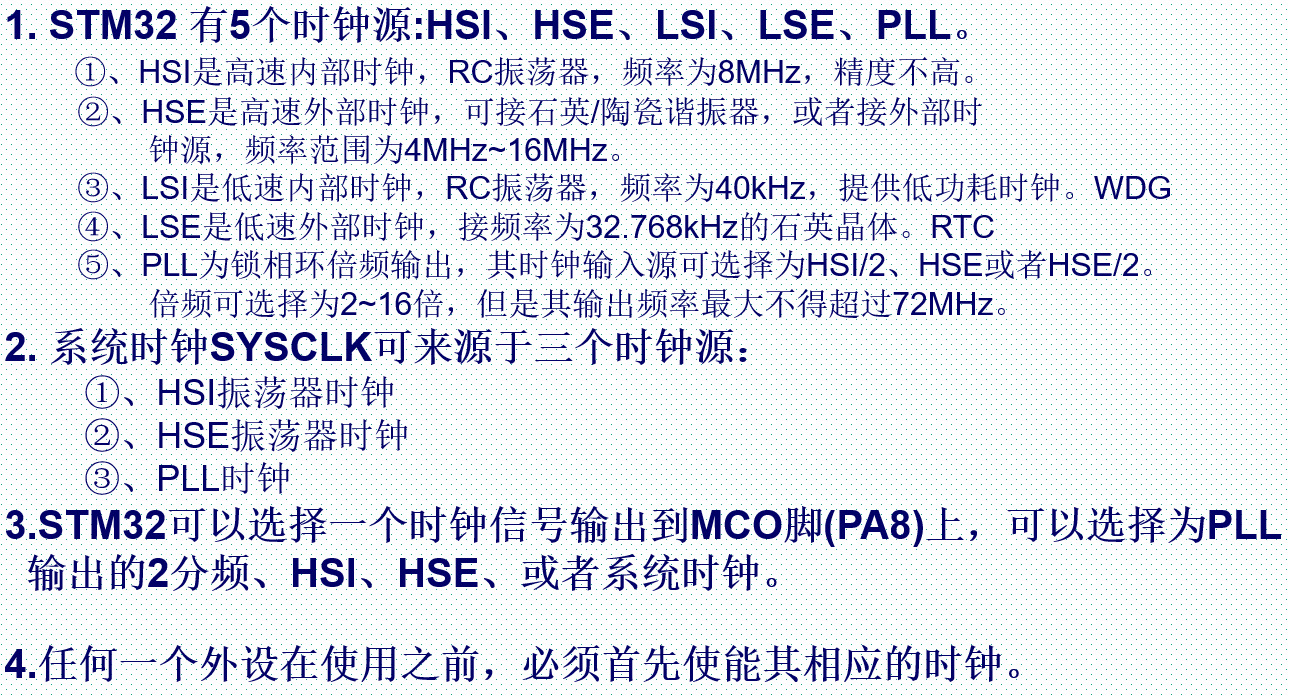
PLLCLK/2

1. CSS

时钟监控系统。假如系统时钟使用的外部时钟源，当外部时钟源突然失效的时候，那么时钟监控系统就可以自动切换时钟源。

1. USB分频器时钟（prescaler）
2. AHB预分频器
3. APB1和APB2

外设的时钟，可以挂载许多的外设

1. 时钟系统总结（如图五）

图五

1. 时钟系统的寄存器
2. RCC寄存器描述

代码如下：

typedef struct

{

\_\_IO uint32\_t CR; //HSI,HSE,CSS,PLL等的使能和就绪标志位

\_\_IO uint32\_t CFGR; //PLL等的时钟源选择，分频系数设定

\_\_IO uint32\_t CIR; //清除/使能 时钟就绪中断

\_\_IO uint32\_t APB2RSTR; //APB2线上外设复位寄存器

\_\_IO uint32\_t APB1RSTR; //APB1线上外设复位寄存器

\_\_IO uint32\_t AHBENR; //DMA，SDIO等时钟使能

\_\_IO uint32\_t APB2ENR; //APB2线上外设时钟使能

\_\_IO uint32\_t APB1ENR; //APB1线上外设时钟使能

\_\_IO uint32\_t BDCR; //备份域控制寄存器

\_\_IO uint32\_t CSR; //控制状态寄存器

} RCC\_TypeDef;

1. 定时器
2. C语言知识的补充
3. static
4. 修饰函数

如果某个文件中的函数被该关键字修饰了，证明该函数只能在该文件中被调用，其它的文件中的函数都不能调用该函数。

1. 修饰变量
2. 修饰函数内的变量（即局部变量）

在函数内被static修饰的变量不随函数的消亡而清零。被static修饰的变量具有记忆功能。

1. 修饰函数外面的变量（即全局变量）

起限定作用，限定该变量只能在其所在的文件调用，其它的文件不能调用该变量

1. struct指针

把函数的参数设置为struct类型的指针会有很多的好处。原因是：

1. 在传递参数的时候传递的指针而非传递整个结构体

这在结构体足够大的时候会体现出来传递指针的好处

1. 便于修改和维护

倘若我们要修改已经写好的结构体，但我们的程序有几百个地方用该结构体声明了对象。首先要明白的一点是，我们不能删除该结构体中已经存在的 内容，但是可以给该结构体添加新的内容。

这算是一种对程序的扩展了。

1. 用结构体类型的指针去做强制类型转换会发生什么？

参见下文！！

1. 寄存器配置

库函数其实就是对寄存器的配置给封装了一遍。在stm32中，寄存器的映射在顶层头文件stm32f10x.h中。

1. 基地址

？？？？？？？？？待定

1. 关于stm32寄存器的地址映射方法

#include <iostream>

struct MyStruct

{

int a;

int b;

int c;

};

#define BASE1 0x4000000

#define BASE2 ((MyStruct \*) BASE1)

using namespace std;

int main()

{

cout << BASE2 << endl;

BASE2->a;

}

代码解释：首先BASE1是一个常量，在“#define BASE2 ((MyStruct \*) BASE1)”中，利用强制类型转换，把BASE1转换为了结构体类型。因此BASE2就是一个结构体常量，它指向一个结构体的地址。

在stm32中，该结构体的地址代表了寄存器中地址。

这或许有一个疑问就是：

如果我在写代码的时候，定义了一个指针P，并给该指针指赋随便的一个数值，在这里强调的是随便。如果，我们不小心修改了P所指向的内容（我们并不知道P指向的内容是什么），后果也是不可预知的。P可能指向的是某个控制寄存器，也或许是某个数据寄存器。总而言之，该指针的不确定性很可能会导致程序的崩溃，因为不小心修改了寄存器。

有如下代码就会报错，错误如图六所示：

#include<stdio.h>

int main()

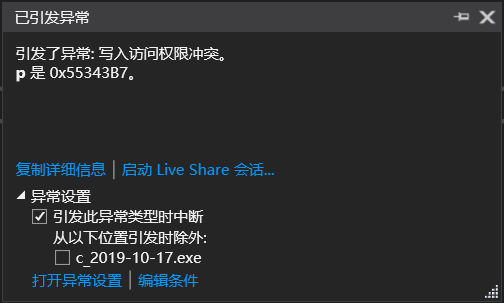
{

int\* p;

p = 89342903;

\*p = 9;

return 0;

}

图六

这虽然是在Windows操作系统下的运行结果，但是本质都是一样的，如果不给一个指针赋值，那么，该指针的值我们就无法预料（即无法预料该指针到底指向了哪里），那么修改该指针指向的内容就会带来很大的风险。

在stm32中，在用C语言编程的时候，没有用到拓展的关键字（在C51中，就有C语言关键字，如“sfr”、“sbit”等等），stm32完全使用C的语法规范。这就产生了一个问题：

如下，在stm32中，在顶层头文件stm32f10x.h中存放都是关于寄存器映射的代码，而且都是一些指针。若用户自己定义了一个指针类型的变量，而且该变量被不小心指向了某个寄存器（编程者并不希望这样），这会导致程序的bug。这就是所谓的野指针。

所以，在写代码的时候，我们通常要给一个指针类型的变量赋一个初始值，并且保证该初始值不是stm中任何一个寄存器的地址。比如通常会有如下的写法：

int main()

{

int a = 0;

int\* p = &a;

return 0;

}

其实，通常不会让一个指针乱指！

如下代码，是stm32中寄存器地址映射的方式

typedef unsigned int uint32\_t;

#define PERIPH\_BASE ((uint32\_t)0x40000000)

#define APB2PERIPH\_BASE (PERIPH\_BASE + 0x10000)

#define GPIOB\_BASE (APB2PERIPH\_BASE + 0x0C00)

#define GPIOB\_ODR\_Addr (GPIOB\_BASE+12)

#define BITBAND(addr, bitnum) ((addr & 0xF0000000)+0x2000000+((addr &0xFFFFF)<<5)+(bitnum<<2))

#define MEM\_ADDR(addr) \*((volatile unsigned long \*)(addr))

#define BIT\_ADDR(addr, bitnum) MEM\_ADDR(BITBAND(addr, bitnum))

#define PBout(n) BIT\_ADDR(GPIOB\_ODR\_Addr,n)

#define LED0 PBout(5)

**ucos操作系统**

1. ucos操作系统概述

二、关于任务

1. 任务

在ucos中，任务就是程序实体，ucos能够管理和调度这些小任务。ucos中的任务由三部分组成，任务堆栈、任务控制块、任务函数

1. 任务堆栈

上下文切换（任务切换）的时候用来保存任务工作环境，就是stm32中内部寄存器的值。

堆栈的创建方法

定义一个数组来作为堆栈

1. 任务控制块

用来记录任务的各个属性

1. 任务函数

又用户编写的任务处理代码，就是实实在在干活的东西，一般的写法如下：

1. ucosIII的系统任务
2. 空闲任务

必须创建的任务，由ucosIII自动创建

1. 时钟节拍任务

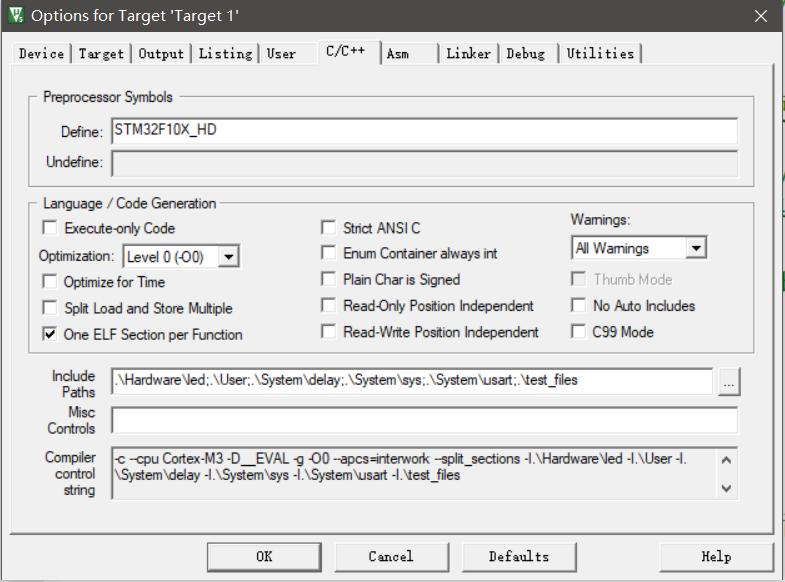
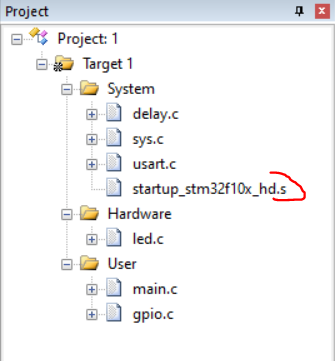
必须创建的任务

1. 统计任务
2. 定时任务
3. 中断服务管理任务
4. 从用户的角度看，ucos任务的状态有5种
5. 休眠态

在flash中，但是不受ucos的管理

1. 就绪态
2. 运行态
3. 等待态

正在运行的需要等待一段时间，或者等待某个事件，这个任务就进入了等待态，此时CPU会把使用权交给别的任务

1. 中断服务态
2. 常见问题
3. 在keil环境下配置工程的方法
4. 一定要注意在路径包含（include path）的时候，不能有中文符号，也不能有空格，不然编译不能通过。
5. 要注意添加启动文件
6. 要注意添加main函数

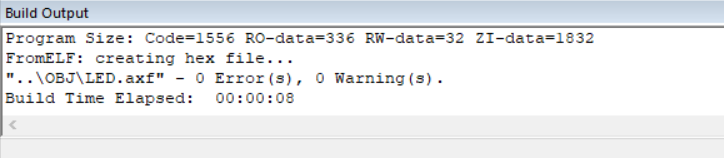
有了启动文件和main函数后，程序就可以编译通过了。

1. 编译后完成后程序的大小

Code 代表执行的代码，程序中所有的函数都位于此处。

RO-data 代表只读数据，程序中所定义的全局常量数据和字符串都位于此处。

RW-data 代表已初始化的读写数据，程序中定义并且初始化的全局变量和静态变量位于此处。

ZI-data 代表未初始化的读写数据，程序中定义了但没有初始化的全局变量和静态变量位于此处。

注：程序的大小是Code和RO-data的大小之和。其中，RO-data包括RW-data和ZI-data。不要五认为是hex文件的大小。

1. 单片机作为驱动的时候，使用灌电流好还是拉电流好？

拉电流：外部器件从单片拉走电流。

灌电流：外部器件向单片机输入电流。

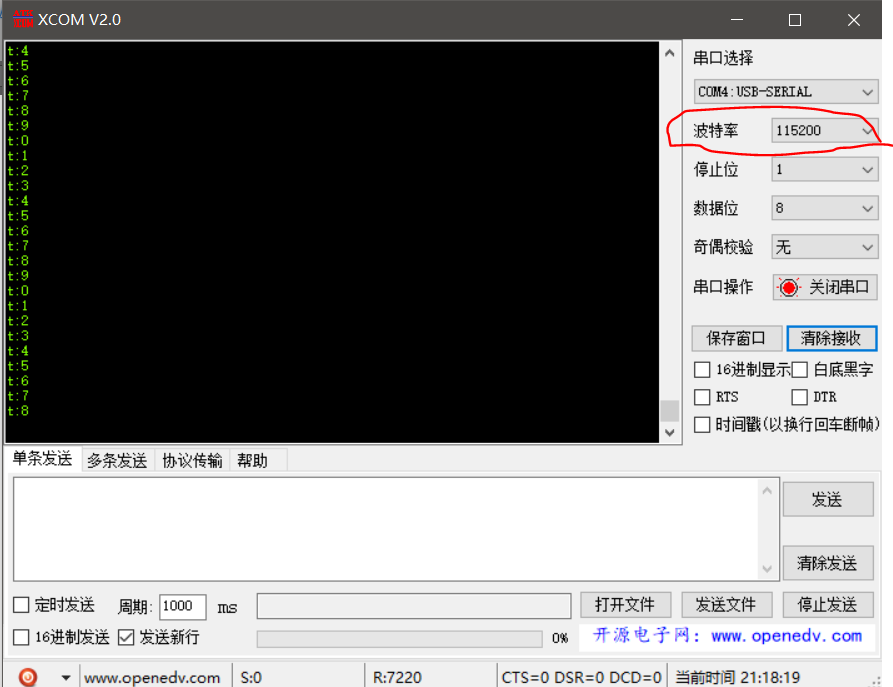
单片机输入低电平的时候驱动能力尚可，而输出高电平的时候就几乎没有驱动能力了。

1. 串口调试时出现乱码

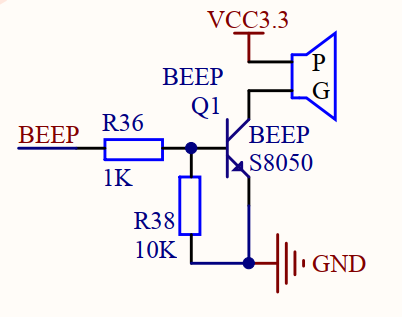
出现这种情况通常是波特率不匹配造成的。

所谓的不匹配就是：

在Windows端的波特率和在stm32的波特率不一致，修改Windows端的波特率比较的方便，但是修改stm32的波特率还需要修改代码（通常需要计算）。所以选择修改Windows端的波特率。

修改方法如下图：

波特率的值要和keil中的一样。

1. 常见电路设计方法总结
2. 蜂鸣器（来自stm32f103ZET6开发板）
3. 常见代码语句解释

RCC->APB2ENR|=1<<3;

将该寄存器的第三位置为1，使能某个外设的时钟，3位的寄存器

GPIOB->CRH&=0XFFFFFFF0;

GPIOB->CRH|=0X00000003;

由于寄存器里面本身会存在不确定的电平，在给寄存器赋值之前，最好要先清空一下比较好。