选择 20'
简答 30'（5*6 / 6*5）
程序 20'（代码解释、填空）
论述 30'

第一章

嵌入式系统的概念、应用与特点：

概念：嵌入式系统是以应用为中心、以计算机技术为基础、软件硬件可裁剪、适应应用系统对功能、成本、可靠性、体积、功耗严格要求的专用计算机系统
应用：国防、工业控制、信息家电、各种商用设备、办公自动化以及近年来发展迅速的移动终端设备等
特点：专用于特定任务、多类型处理器和处理器系统支持、通常极其关注成本、一般是实时系统、可裁剪性好、可靠性高、大多有功耗约束

嵌入式系统的硬件（CPU、外设）：

嵌入式系统硬件主要指嵌入式微处理器和外围设备
其中嵌入式处理器是嵌入式系统的核心，一般只保留与用户需求紧密相关的功能部件，因此具有体积小、重量轻、成本低、可靠性高等特点
而外围设备根据功能一般可分为存储设备、通信设备和 I/O 设备
嵌入式处理器与通用处理器最大的不同点在于：
嵌入式CPU大多工作在为特定用户群所专门设计的系统中，它将通用CPU中许多由板卡完成的任务集成到芯片内部，从而有利于嵌入式系统在设计时趋于小型化，同时还具有很高的效率和可靠性
目前常用的嵌入式处理器可分为嵌入式微控制器（MCU）、嵌入式微处理器（MPU）、嵌入式DSP处理器（EDSP）、嵌入式片上系统（SOC）
在嵌入式硬件系统中，除了中心控制部件以外，用于完成存储、通信、调试、显示等辅助功能的其他部件，事实上都可以算作嵌入式外围设备，目前常用的嵌入式外围设备按功能可以分为存储设备、通信设备和显示设备等

主要嵌入式软件系统（应用及OS）：

嵌入式软件可分为嵌入式操作系统和应用软件两大类，其核心是嵌入式操作系统

嵌入式操作系统是嵌入式系统中最基本的软件，它负责分配、回收、控制和协调全部软硬件资源的并发活动，并且提供应用程序的运行环境和接口，是应用程序运行的基础

大部分嵌入式操作系统都是实时系统，而且是实时多任务系统

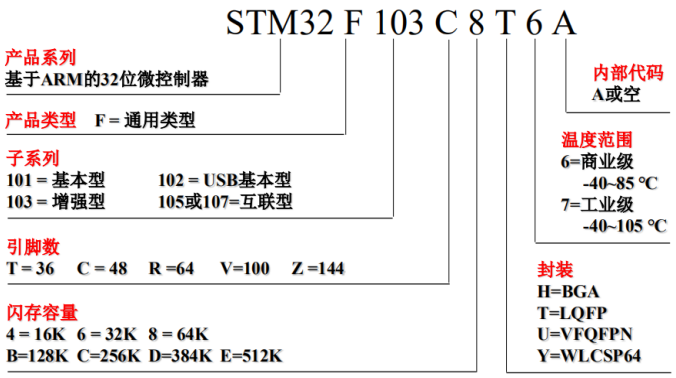
常见的5种嵌入式操作系统：μClinux、μC/OS-II、WinCE、FreeRTOS、RT-Thread

嵌入式应用软件则是服务于某种专用应用领域，基于某一特定的嵌入式硬件平台，用来达到用户预期任务的计算机软件，从功能性的角度分析，可以把嵌入式应用软件分为支撑软件和应用程序两大类

嵌入式系统的发展趋势：

① 嵌入式开发是一项系统工程；
② 网络化、信息化的要求；
③ 网络互联、移动互联成为必然趋势；
④ 精简系统内核、算法，降低功耗和软硬件成本；
⑤ 提供友好的多媒体人机界面；

第二章



ARM 处理器特点及应用：

ARM 微处理器采用 RISC 架构

特点：

① 体积小、低功耗、低成本、高性能；
② 支持 Thumb 或 Thumb-2 指令集，能很好地兼容 8 位/ 16 位器件；
③ 大量使用寄存器，指令执行速度更快；
④ 大多数数据操作都在寄存器中完成；
⑤ 寻址方式灵活简单，执行效率高；
⑥ 指令长度固定；

应用：工业控制领域、无线通讯领域、网络应用、消费类电子产品、成像和安全产品

作为 32 位的微处理器，ARM 体系结构支持的最大寻址空间为 2^32 个字节 = 4GB

CM3 微控制器简介：

ARM Cortex-M3 处理器是专门针对 存储器和处理器的尺寸对产品成本影响很大的各种应用而开发设计的，是一种嵌入式微控制器（MCU）

ARM Cortex-M3 处理器具有存储器保护单元（MPU）和嵌套向量中断控制器（NVIC）

ARM Cortex-M3 处理器采用 ARMv7-M 架构

ARM Cortex-M3 处理器不能执行 ARM 指令集中的指令，只能执行所有的 16 位 Thumb 指令集和基本的 32 位 Thumb-2 指令集中的指令

ARM Cortex-M3 处理器支持两种工作模式：线程模式、处理模式

ARM Cortex-M3 处理器有两种工作状态：Thumb 状态、调试状态

ARM Cortex-M3 内核采用三级流水线结构，每条指令的执行分取指、译码和执行三个阶段，可实现多条指令并行执行

ARM Cortex-M3 的主要特点：

① 采用 Thumb-2 指令集架构的子集；

② 采用哈佛处理器架构；

③ 采用三级流水线+分支预测；

④ 可实现 32 位单周期乘法；

⑤ 可实现 2~12 周期硬件除法；

CM3 存储格式类型：

ARM 体系结构将存储器看作是从零地址开始的字节的线性组合。不过，要注意 32 位处理器很多情况下是以字为单位进行处理的，一个字的长度为 4 个字节。因此在 ARM 的存储系统中，从第 0 字节到第 3 字节放置第 1 个字数据，以此类推

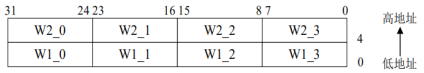
ARM 体系结构支持用两种方法存储字数据，称之为大端格式和小端格式。对于字数据，其具体存储规则为：

大端格式：字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中

小端格式：字数据的低字节存储在低地址中，而字数据的高字节则存放在高地址中

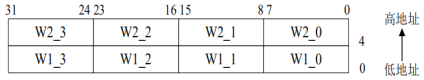
大端格式：字数据的高字节存储在低地址中(大位数在小地址)

- unsigned int value = 0x12345678　00000000：12 34 56 78



小端格式：字数据的低字节存放在低地址中(小位数在小地址)

- unsigned int value = 0x12345678　00000000：78 56 34 12



CM3 嵌套向量中断控制器 NVIC：

中断：是指正在执行的程序流程被某个事件打断、处理器转而去执行与事件有关的处理程序

ARM Cortex-M3 的所有中断机制都由 NVIC 实现，在 NVIC 的标准执行中，它提供了一个非屏蔽中断（NMI）和 32 个通用物理中断，这些中断带有 8 级的抢占优先权。NVIC 可以通过综合选择配置为 1 到 240 个物理中断中的任何一个，并带有多达 256 个优先级

ARM Cortex-M3 使用了末尾连锁技术，简化了激活的和未决的中断之间的切换，把需要用时 42 个时钟周期才能完成的连续的堆栈弹出和压入操作替换为 6 个周期就能完成的指令取指（中断向量），实现了延迟的降低

CM3 寄存器组织：

ARM Cortex-M3 拥有通用寄存器 R0~R15 以及一些特殊功能寄存器
其中 R0~R12 是 32 位的通用目的寄存器，它们都可以被 32 位指令访问，但是绝大多数的 16 位指令只能访问 R0~R7，而 32 位的 Thumb-2 指令则可以访问所有通用寄存器
通用目的寄存器结构：R0~R7 低寄存器、R8~R12 高寄存器、R13 堆栈指针、R14 连接寄存器、R15 程序计数器
特殊功能寄存器包括 1 个程序状态寄存器、3 个中断屏蔽寄存器以及 1 个控制寄存器

CM3 指令系统：

ARM Cortex-M3 处理器不支持 ARM 指令集，只单独支持 Thumb-2 指令集， Thumb-2 技术是 16 位和 32 位指令的集合，实现了 32 位 ARM 指令性能，匹配原始的 16 位 Thumb 指令集并与之向后兼容
ARM Cortex-M3 处理器的指令分为数据传送指令、数据处理指令、跳转指令等几类
数据传送指令：
在两个寄存器之间传递数据：
MOV 用于在寄存器间传递数据
MVN 把寄存器的内容取反后再传送
在寄存器和存储器之间传递数据：
LDR 把存储器中的内容加载到寄存器中，LDRB 读一个字节，LDRH 读一个半字，LDR 读一个字，LDRD 读一个双字
STR 把寄存器中的内容存储到存储器中
在寄存器和特殊功能寄存器之间传递数据：
MRS 读特殊功能寄存器的值到通用寄存器
MSR 写通用寄存器的值到特殊功能寄存器
把一个立即数加载到寄存器：
16 位指令 MOV 支持 8 位立即数加载，32 位指令 MOVW 和 MOVT 支持 16 位立即数加载，如果要加载 32 位立即数，可以通过组合使用 MOVW 和 MOVT，其中必须先使用 MOVW 然后再使用 MOVT，顺序不能颠倒，因为 MOVW 会清零高 16 位
数据处理指令包括：算术运算指令、逻辑运算指令、位运算指令
最基本的无条件跳转指令有两条： B Lable 跳转到 Lable 处对应的地址； BX reg 跳转到由寄存器 reg 给出的地址
调用子程序时，需要保存返回地址，使用的指令助记符为 BL， BL Lable 跳转到 Lable 处对应的地址，并将跳转前的下一条指令的地址保存到 LR； BLX reg 跳转到由寄存器 reg 给出的地址，并根据 reg 的 LSB 切换处理器状态，还要将跳转前的下一条指令的地址保存到 LR，执行这些指令后，就把返回地址存储到 LR 中了，从而才能使用 BX LR 等形式返回

CM3 存储映射：

ARM Cortex-M3 只有一个单一固定的存储器映射
1 MB 的位带区（8 M 个位），32 MB 的位带别名区（8 M 个字）
位带别名区中的每个字（32 位） 对应 位带区中的每个比特（1 位）
位带操作适用于数据访问，不适用于取指
借助位带的功能，可以把多个布尔型数据打包在单一的字中（位带区），却依然可以从位带别名区中像访问普通内存一样地使用它们，在使用时，真正起作用的是位带区中的位，但对于该位（位带区）的读写操作都可以变成对该位的对应的字（位带别名区）的操作
在许多情况下，某种工作模式或状态只需要通过 1 个二进制位来表示即可，把这种位操作所对应的存储单元集中起来，就构成了所谓的位带
另一方面，当需要访问位带区的位时，往往只是希望对所需要访问的某 1 位进行读或写操作，但程序的访存指令在执行时往往是按字（至少是字节）为单位进行操作，从而导致读或写了许多不需要的位，处理起来也比较麻烦，为了解决这个问题，进一步在存储器中设置了位带别名区
ARM Cortex-M3 处理器的存储器映射中包括两个位带区，它们分别是：① SRAM 区域中的最低的 1 MB；② 外设存储区域中的最低的 1 MB；这两个位带区的起始地址分别为 0x20000000 和 0x40000000，与这两个位带区相对应，有两个 32 MB 的位带别名区的起始地址分别为 0x22000000 和 0x42000000
对位带别名区的访问 映射为（等效于） 对位带区的访问
访问位带别名区：
向位带别名区写入一个字 与 对位带区相应的位所在的字执行 读—修改—写 操作具有相同的作用
写入位带别名区的字的第 0 位 决定了 写入位带区的目标位的值，位带别名区的字剩余的第 1 到 31 位的值不影响目标位
采用大端格式时，对位带别名区的访问必须以字节方式
访问位带区：
对于位带区，能够使用常规的读和写操作对该区域进行访问
位带别名区和位带区的映射公式：
对于位带别名区中的字，只需要知道其相对于位带别名区的首地址的偏移量即可
对于位带区中的位，要知道其在位带区中所在的那个字（或字节）及其在字（或字节）内部是第几位
一、根据位带中的位的位置计算其在位带别名区中的字的地址
[（位带中的位所在字节在位带区中的地址 - 位带区的首地址）× 8 + 位在该字节中的编号] × 4
即（在位带区中到该位为止总共有多少位）× 每 1 位对应 1 个字也就是 4 字节
二、根据位带别名区中的字的地址计算其在位带中的位的位置

1. （位带别名区中的字的地址 - 位带别名区的首地址）/ 4 即 第几个字 即 在位带区中到该位为止总共有多少位（位编号）
2. （位编号 / 8）向下取整 即 在位带中该位所在的字节相对于位带区首地址的偏移量
3. 位编号 - 字节偏移量 × 8 即 该位在所在字节内部的序号

CM3 异常与中断：

中断：由于 CPU 外部的原因而导致正常的程序执行流程发生改变，属于异步事件，又称为硬件中断
异常：是 CPU 自动产生的自陷，以处理异常事件，属于同步事件，例如处理一个外部的中断请求
自陷：通过处理器所拥有的软件指令、可预期地使处理器正在执行的程序执行流程发生改变，属于同步事件，也称为内部中断或是软件中断
广义的中断通常被分为：中断、自陷、异常等类别
在处理异常之前，当前处理器的状态必须保留，这样当异常处理完成之后，当前程序可以继续执行
处理器允许多个异常同时发生，它们将会按固定的优先级进行处理
优先级对于异常来说很关键，它会决定一个异常是否被屏蔽，以及在未屏蔽的情况下何时可以响应
异常的优先级的数值越小，则优先级越高
ARM Cortex-M3 支持嵌套中断，使得高优先级的异常会抢占低优先级的异常占用的 CPU 资源
一个外部中断的优先级值由主优先级（抢占优先级）和子优先级（响应优先级）构成
在不同的优先级构成模式下，一个优先级值对应的二进制代码中，有一部分表示优先级所在的分组（主优先级），剩下的部分表示优先级所在的分组中的编号（子优先级），表示主优先级的位段最多占 7 位，表示子优先级的位段至少占 1 位，所有可能的优先级可以不被分组，如果要分组则最多可以被分为 128 组
ARM Cortex-M3 在决定任意两个可配置异常哪一个优先处理时遵循：软件优先级中主优先级更高的先被处理；主优先级相同时，软件优先级中子优先级更高的先被处理；软件主优先级和子优先级都相同时，硬件优先级更高的先被处理

STM32 的时钟源和时钟树：

处理器需要时钟才能工作，不同片内外设也工作在不同的频率
每一个外设模块在 时钟电路相关的 时钟配置寄存器中 都有 相应的时钟使能位，当该位为 1 时，对应的时钟开启且相应的外设开始工作，当该位为 0 时，对应的时钟关闭且相应的外设停止工作
STM32 支持的时钟源包括：HSE 高速外部时钟、LSE 低速外部时钟、HSI 高速内部时钟、LSI 低速内部时钟、PLL 锁相环时钟，这些时钟源中的任何一个都可以独立的开启或关闭，其中 HSE 和 HSI 可以作为 PLL 的输入，HSE、HSI、PLL 用于驱动系统时钟

看门狗：

看门狗用于检测和解决由软件错误引起的故障。一般是倒计时器，作用是以一定的周期产生复位信号使系统复位。在设计嵌入式系统软件时，通过在看门狗产生复位信号前执行喂狗操作来避免看门狗倒计数到产生复位信号的值。如果在嵌入式系统中执行的软件正常运行，则该软件应该能够正常的执行喂狗操作，这样系统就不会被看门狗复位

STM32F103系列内置两个看门狗：独立看门狗和窗口看门狗，对于窗口看门狗，它不仅能够产生复位信号，还能在产生复位信号前触发一个中断，如果程序在相应的中断服务函数中重新装载倒计数值，则能够避免窗口看门狗产生复位信号

GPIO：

通用输入/输出端口，是一个灵活的由软件控制的数字信号，每个 GPIO 都代表一个连接到 CPU 特定引脚的一个位
STM32 的 GPIO 端口的每一位都可以由软件配置成多种模式：输入浮空、输入上拉、输入下拉、模拟输入、开漏输出、推挽式输出、推挽式复用功能、开漏复用功能
STM32 的每个 I/O 口都可以自由编程，然而必须按照 32 位字访问 I/O 端口寄存器，每个 I/O 端口有两个 32 位配置寄存器、两个 32 位数据寄存器、一个 32 位置位/复位寄存器、一个 16 位复位寄存器和一个 32 位锁定寄存器
所有 I/O 端口都有外部中断能力，可以作为外部中断的输入
端口复用功能：为了节约引出管脚，将 I/O 管脚作为外设模块的功能引脚
引脚重映射：一个外设的引脚除了具有默认的引脚位外，还可以通过配置重映射寄存器的方式，把这个外设的引脚映射到其它的引脚位

第三章

交叉编译：

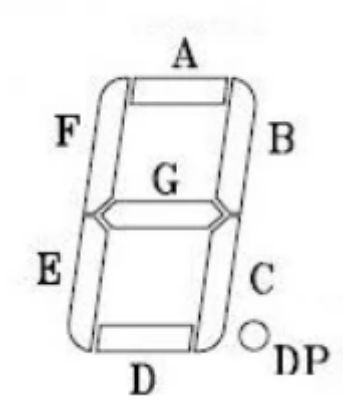
是指在某个主机平台上用交叉编译器编译出可在其它平台上运行的代码的过程

第四章

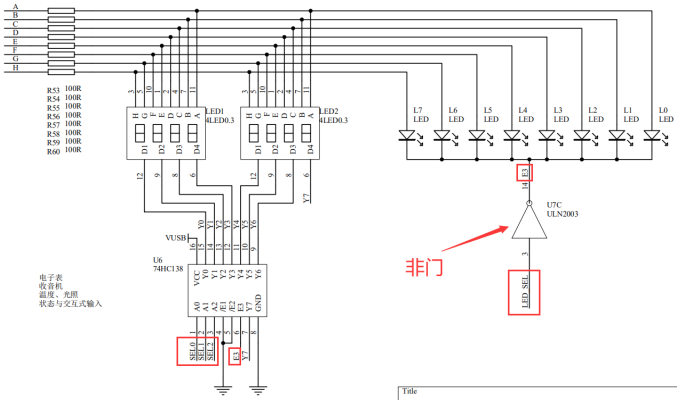
GPIO：

STM32F103VBT6 一共有 5 组输入/输出端口，分别是 GPIOA、GPIOB、GPIOC、GPIOD、GPIOE，每组 I/O 口有 16 个 IO，一共是 80 个 IO 端口
STM32 的每个 IO 端口都有 7 个相关寄存器来控制，其中常用的 IO 端口寄存器只有四个：用于配置模式的 2 个 32 位的端口配置寄存器 CRL 和 CRH，2 个 32 位的数据寄存器 IDR 和 ODR
CRL 和 CRH 控制着每个 IO 口的模式及输出速率，其中 CRL 控制低 8 位 IO 口，CRH 控制高 8 位 IO 口，每个 IO 口占用相应寄存器的 4 个位（4 × 8 = 32），这两个寄存器的复位值为 0x44444444 ，也就是配置端口为浮空输入模式
IDR 是一个端口输入数据寄存器，只用了低 16 位，该寄存器为只读，ODR 是一个端口输出数据寄存器，也只用了低 16 位，该寄存器为可读写，向该寄存器写数据，可以控制某个 IO 口的输出电平，从这两个寄存器中读出来的数据可以用于判断当前 IO 口的状态
七段数码管：

此处用到的为共阴极数码管，即将 8 个 LED 的阴极连在一起，让其接地，则给其中任何一个 LED 的另一端高电平即可将其点亮，如，若将每个二极管对应一个字节的 8 个位，A 为 Bit0，B 为 Bit1，...，则显示数字 0 的字符编码为 00111111，即 0x3f



引脚 LED_SEL 为 1 时，发光二极管工作，引脚 LED_SEL 为 0 时，七段数码管工作



七段数码管工作时，根据 SEL0~SEL2 的值确定哪个七段数码管被选中，即位选，再根据 A~H 引脚的高低电平，如 0 对应的 0x3f，点亮这一个被选中的七段数码管中相应的 LED，即段选

```
#define RCC_APB2ENR ((u32 *)0x40021018)   //定义APB2ENR寄存器        地址
#define AFIO_MAPR   ((u32 *)0x40010004)   //定义APIO的MAPR寄存器

#define GPIOB_CRL (*(u32 *)0x40010C00)    //定义GPIOB_CRL寄存器      值
#define GPIOB_ODR (*(u32 *)0x40010C0C)    //定义GPIOB_ODR寄存器
#define GPIOE_CRH (*(u32 *)0x40011804)    //定义GPIOE_CRH寄存器
#define GPIOE_ODR (*(u32 *)0x4001180C)    //定义GPIOE_ODR寄存器
u32 *gpio_odr=((u32 *)0x4001180c);        //定义成地址变量

u32 *PE08 = (u32 *)(0x42000000 +(0x4001180C-0x40000000)*32 + 8*4); //位带定义PE08
u32 *PE09 = (u32 *)(0x42000000 +(0x4001180C-0x40000000)*32 + 9*4); //位带定义PE09
```

使能端口 IO 时钟：

```
*RCC_APB2ENR|=1<<0;             // 使能AFIO
*RCC_APB2ENR|=1<<3;             // 使能PORTB时钟
*RCC_APB2ENR|=1<<6;             // 使能PORTE时钟

*AFIO_MAPR |= 0x02000000;       // 设置PB.3为I/O口可用，且可以SW仿真
```

初始化 IO 端口参数：

```
GPIOB_CRL &= 0xFFFF0FFF;              // 清除PB.3原先配置
GPIOB_CRL = GPIOB_CRL | 0x00003000;   // PB.3配置为推挽输出
GPIOB_ODR |= 0x00000008;              // PB.3输出高，选择控制LED灯

GPIOE_CRH &= 0X00000000;              // 清除PE.8-15原先配置
GPIOE_CRH |= 0X33333333;              // PE.8-15配置为推挽输出
GPIOE_ODR |= 0x0000FF00;              // PE.8-15输出高，八个LED灯全亮
```

通过位带别名区，进行位带操作，控制 IO 口：

```
*PEO8  = 1;   //LED1亮
*PEO9  = 1;   //LED2亮
```

通过端口输出数据寄存器 ODR，控制 IO 口：

```
GPIOE_ODR = 0x00000400; //仅LED3亮，其他灯灭   0000 0100 0000 0000 只有 PE.10 输出高
```

中断：

CM3 内核支持 256 个中断，其中包括了 16 个内核中断和 240 个外部中断，并且具有 256 级的可编程中断设置，STM32 只使用了其中的一部分中断资源，共 84 个，包括 16 个内核中断和 68 个可屏蔽中断，具有 16 级的可编程中断优先级，而 STM32F103 系列上面又只使用了其中的 60 个中断

ISER：中断使能寄存器，要使能某个中断，必须设置相应的 ISER 位为 1（但这仅仅是使能，还要配合中断分组、屏蔽、IO 口映射等设置才是一个完整的中断设置）

ICER：中断除能寄存器，要除能某个中断，必须设置相应的ICER 位为 1

ISPR：中断挂起控制寄存器，通过将某位置 1，可以将正在进行的相应中断挂起，转而执行同级或更高级别的中断

ICPR：中断解挂控制寄存器，通过将某位置 1，可以将被挂起的相应中断解挂

IABR：中断激活标志位寄存器，某位为 1 则表示该位所对应的中断正在被执行，中断完成后由硬件自动清零

IPR：中断优先级控制的寄存器，STM32 的中断分组与其密切相关，共 15 个 32 Bit 的寄存器，刚好可以表示 STM32 的 60 个外部中断（每个可屏蔽中断占用 8 Bit，15 × (32 / 8) = 60），每个可屏蔽中断占用的 8 Bit并没有完全被使用，而是只用了高四位，这 4 位分为抢占优先级（主优先级）和响应优先级（子优先级），抢占优先级在前，子优先级在后，这两个优先级各占几个位由 SCB->AIRCR 中的中断设置来决定

STM32 将中断分为 0 到 4 共 5 个组，组 0 的分配结果为 0 位抢占优先级，4 位响应优先级，组 4 的分配结果为 4 位抢占优先级，0 位响应优先级

如果两个中断的抢占优先级和响应优先级都是一样的话，则看哪个中断先发生就先执行

高优先级的抢占优先级中断可以打断正在执行的低优先级的抢占优先级中断

对于抢占优先级相同的中断，高优先级的响应优先级不可以打断低优先级的响应优先级中断

中断分组由 SCB->AIRCR 寄存器的 Bit 10~8 来设置

设置 NVIC 分组的函数：

```
//设置NVIC分组
//NVIC_Group:NVIC分组 0~4 总共5组
void MY_NVIC_PriorityGroupConfig(u8 NVIC_Group)
{
  u32 temp,temp1;
  temp1=(~NVIC_Group)&0x07;        // 取后三位
  temp1<<=8;

  temp=SCB->AIRCR;                 // 读取先前的设置
  temp&=0X0000F8FF;                // 清空先前分组
  temp|=0X05FA0000;                // 写入钥匙
  temp|=temp1;
  SCB->AIRCR=temp;                 // 设置分组
}
```

设置 NVIC：

```
// 设置NVIC
// NVIC_PreemptionPriority:抢占优先级
// NVIC_SubPriority       :响应优先级
// NVIC_Channel           :中断编号   STM32F103系列上面只有60个可屏蔽中断
// NVIC_Group             :中断分组 0~4
// 注意优先级不能超过设定的组的范围!否则会有意想不到的错误
// 组划分:
// 组0: 0位抢占优先级,4位响应优先级
// 组1: 1位抢占优先级,3位响应优先级
// 组2: 2位抢占优先级,2位响应优先级
// 组3: 3位抢占优先级,1位响应优先级
// 组4: 4位抢占优先级,0位响应优先级
// NVIC_SubPriority和NVIC_PreemptionPriority的原则是,数值越小,越优先
void MY_NVIC_Init(u8 NVIC_PreemptionPriority,u8 NVIC_SubPriority,u8 NVIC_Channel,u8 NVIC_Group)
{
  u32 temp;
  u8 IPRADDR = NVIC_Channel/4;      // 每个寄存器表示4个可屏蔽中断，IPR 寄存器组由 15 个 32 Bit的寄存器组成
  u8 IPROFFSET = NVIC_Channel%4;    // 在寄存器内是第几个可屏蔽中断
  IPROFFSET = IPROFFSET*8+4;        // 得到偏移的确切位置，每一个可屏蔽中断占用 8 Bit，这 8 Bit只使用了高 4 位

  MY_NVIC_PriorityGroupConfig(NVIC_Group);    // 设置分组

  temp = NVIC_PreemptionPriority<<(4-NVIC_Group);
  temp |= NVIC_SubPriority&(0x0f>>NVIC_Group);
  temp &= 0xf;                      // 取低四位

  if(NVIC_Channel<32)
    NVIC->ISER[0] |= 1<<NVIC_Channel;       // 使能中断位(要清除的话,相反操作就OK)
  else
    NVIC->ISER[1] |= 1<<(NVIC_Channel-32);

  NVIC->IPR[IPRADDR] |= temp<<IPROFFSET;    // 设置响应优先级和抢占优先级
}
```

对于外部中断的设置，还需要配置相关寄存器，STM32F103 的 EXTI 控制器支持 19 个外部中断/事件请求，每个中断设有状态位，每个中断/事件都有独立的触发和屏蔽设置

IMR：中断屏蔽寄存器，当某一位设置为 1 时，开启这条线上的中断

EMR：事件屏蔽寄存器，当某一位设置为 1 时，开启相应事件的中断

RTSR：上升沿触发选择寄存器，当某一位设置为 1 时，允许对应上升沿触发中断/事件

FTSR：下降沿触发选择寄存器

PR：挂起寄存器，某一位设置为 1 表示清除该中断线上的中断标志位

外部中断配置函数：

```
// 外部中断配置函数
// 只针对GPIOA~G;不包括PVD,RTC和USB唤醒这三个，即只对应外部 IO 口的输入中断
// 参数:GPIOx:0~6,代表GPIOA~G;  BITx:需要使能的位;  TRIM:触发模式,1下升沿;2上降沿;3任意电平触发
// 该函数一次只能配置1个IO口,多个IO口,需多次调用
// 该函数会自动开启对应中断,以及屏蔽线
void Ex_NVIC_Config(u8 GPIOx,u8 BITx,u8 TRIM)
{
  u8 EXTADDR;
  u8 EXTOFFSET;
  EXTADDR=BITx/4;          // 得到中断寄存器组的编号
  EXTOFFSET=(BITx%4)*4;

  RCC->APB2ENR|=0x01;      // 使能 IO 复用时钟

  AFIO->EXTICR[EXTADDR]&=~(0x000F<<EXTOFFSET);      // 清除原来设置
  AFIO->EXTICR[EXTADDR]|=GPIOx<<EXTOFFSET;          // EXTI.BITx映射到GPIOx.BITx

  //自动设置
  EXTI->IMR|=1<<BITx;          // 开启line BITx上的中断
  if(TRIM&0x01)
    EXTI->FTSR|=1<<BITx;       // line BITx上事件下降沿触发
  if(TRIM&0x02)
    EXTI->RTSR|=1<<BITx;       // line BITx上事件上升沿触发
}
```

该函数的流程：首先根据 GPIOx 的位得到中断寄存器组的编号，即 EXTICR 的编号（一共有 4 个，对应 16 根中断线），每个 EXTICR 只用了其低 16 位，来表示 4 根中断线，然后再设置该 EXTICR 来决定对应的中断线配置到哪一个 GPIO 上，最后使能该位的中断及事件，并配置触发中断的方式

键盘中断程序设计：

```c
//外部中断2服务程序
void EXTI2_IRQHandler(void)
{
   delay_ms(10);      // 消抖
   if(KEY1==0)        // 按键1
   {
     LED0=!LED0;
   }
   EXTI->PR=1<<2;     // 清除LINE2上的中断标志位
}

//外部中断初始化程序
//初始化PC0-2为中断输入.
void EXTIX_Init(void)
{
   RCC->APB2ENR|=1<<4;              // 使能PORTC时钟
   GPIOC->CRL&=0XFFFFF000;          // PC0-2设置成输入
   GPIOC->CRL|=0X00000888;

   Ex_NVIC_Config(GPIO_C,0,FTIR);  // 下降沿触发
   Ex_NVIC_Config(GPIO_C,1,FTIR);  // 下降沿触发
   Ex_NVIC_Config(GPIO_C,2,FTIR);  // 下降沿触发

   MY_NVIC_Init(2,2,EXTI0_IRQChannel,2);    // 抢占2，子优先级2，组2
   MY_NVIC_Init(2,1,EXTI1_IRQChannel,2);    // 抢占2，子优先级1，组2
   MY_NVIC_Init(2,0,EXTI2_IRQChannel,2);    // 抢占2，子优先级0，组2
}
```

串口：

UART，通用异步收发传输器，是一种通用串行数据通信接口，即作为发送器，又作为接收器，
处理器可以通过数据总线向 UART 的控制寄存器中写入控制字，对 UART 进行初始化
发送器从处理器接收并行数据，然后通过移位寄存器把数据以串行异步方式发出
接收器从串行通信链路上接收串行数据，然后通过移位寄存器把数据转换成 8 位并行数据，送往接收寄存器，等待处理器读取
处理器可以通过读取 UART 状态寄存器的信息获得当前 UART 的状态
比特率，表示单位时间内传输的 Bit 数，波特率，指串口通信时的速率，比特率 = 波特率 × 单个调制状态所需要的 Bit 数
串口最基本的设置就是对波特率进行设置
要使用一个串口，需要配置波特率、数据位长度、奇偶校验位等信息，并开启串口时钟，
STM32 的每个串口都有一个自己独立的波特率寄存器 USART_BRR，通过设置该寄存器就可以配置不同的波特率
USART_BRR 的最低 4 位用来存放波特率的小数部分，紧接着的 12 位用来存放波特率的整数部分
串口的时钟频率 /（波特率 × 16）= USARTDIV，USARTDIV 是一个无符号定点数，即 USART_BRR 中的低 16 位
USART_CR：控制寄存器
USART_DR：数据寄存器，当向该寄存器中写数据时，串口就会自动发送，当收到数据的时候，会存放在该寄存器中
USART_SR：状态寄存器
串口程序设计：
① 串口时钟使能，GPIO 时钟使能；② 串口复位；③ 初始化 IO 端口参数；④ 初始化串口参数；⑤ 开启中断并且初始化 NVIC；⑥ 使能串口；⑦ 编写中断处理函数；

```c
//重定义fputc函数
int fputc(int ch, FILE *f)
{
  while((USART1->SR&0X40)==0);     // 循环发送,直到发送完毕
  USART1->DR = (u8) ch;
  return ch;
}
```

```c
u8 USART_RX_BUF[64];              // 接收缓冲,最大64个字节

//接收状态
//bit7, 接收完成标志
//bit6, 接收到 0x0d , 即回车符的ASCII码
//bit5~0, 接收到的有效字节数目
u8 USART_RX_STA=0;                // 接收状态标记

void USART1_IRQHandler(void)      // 串口1中断服务程序
{
  u8 res;
  if(USART1->SR&(1<<5))           // 读数据寄存器非空，RXNE，接收到数据
  {
    res = USART1->DR;
    if((USART_RX_STA&0x80)==0)    // 接收未完成
    {
      if(USART_RX_STA&0x40)       // 接收到了 0x0d 回车符
      {
        if(res!=0x0a) USART_RX_STA=0;// 接收错误,重新开始，0x0a 换行符
        else USART_RX_STA|=0x80;  // 接收完成
      }
      else                        // 还没收到回车符
      {
        if(res==0x0d)             // 此次收到的是回车符
          USART_RX_STA|=0x40;
        else                      // 此次收到的不是回车符
        {
          USART_RX_BUF[USART_RX_STA&0X3F]=res;    // 将数据存到缓冲区中
          USART_RX_STA++;
          if(USART_RX_STA>63) USART_RX_STA=0;//接收数据错误,重新开始接收
        }
      }
    }
  }
}
```

```c
//初始化IO 串口1
//pclk2:PCLK2时钟频率(Mhz)
//bound:波特率
//CHECK OK
//091209
void uart_init(u32 pclk2,u32 bound)
{
  float temp;
  u16 mantissa;       // 小数部分，对应 USART_BRR 的最低 4 位
  u16 fraction;       // 整数部分，对应 USART_BRR 紧接着的 12 位
  temp=(float)(pclk2*1000000)/(bound*16);    // 得到USARTDIV
  mantissa=temp;                             // 得到整数部分
  fraction=(temp-mantissa)*16;               // 得到小数部分
  mantissa<<=4;
  mantissa+=fraction;                        // 将整数部分和小数部分拼在一起

  RCC->APB2ENR|=1<<2;                        // 使能 PORTA 口时钟
  RCC->APB2ENR|=1<<14;                       // 使能串口时钟
  GPIOA->CRH&=0XFFFFF00F;
  GPIOA->CRH|=0X000008B0;                    // 初始化 IO 端口参数

  RCC->APB2RSTR|=1<<14;                      // 复位串口1
  RCC->APB2RSTR&=~(1<<14);                   // 停止复位

  USART1->BRR=mantissa;                      // 在 USART1_BRR 中设置波特率
  USART1->CR1|=0X200C;                       // 1位停止,无校验位

  //使能接收中断
  USART1->CR1|=1<<8;                         // PE中断使能
  USART1->CR1|=1<<5;                         // 接收缓冲区非空中断使能
  MY_NVIC_Init(3,3,USART1_IRQChannel,2);     // 组2，最低优先级

}
```

模数转换：

ADC 是指模/数转换器，是一种将连续变化的模拟信号转换为离散的数字信号的器件

A/D 转换器具有四个功能：采样、保持、量化、编码

内部电容器组的变化会造成 ADC 精准度的误差，因此在上电时需要进行一次校准

STM32 ADC 是 12 位逐次逼近型的模拟数字转换器，有 18 个通道，可测量 16 个外部和 2 个内部信号源，各通道的 A/D 转换可以单次、连续、扫描或间断模式执行，最大转换速率为 1 MHz，STM32 将 ADC 的转换分为 2 个通道组，注入通道组和规则通道组，注入通道的转换可以打断规则通道的转换，在注入通道被转换完成后，规则通道才可以继续转换

对于每个要转换的通道，采样时间可以尽量长一些，以获得较高的准确度，但是这样会降低 ADC 的转换速率

使用 ADC1 通道 1 来进行 AD 转换的步骤：

① 开启 PORTA 口时钟，设置 PORTA1 为模拟输入；② 使能 ADC1 时钟，并设置分频因子；③ 设置 ADC1 的工作模式；④ 设置 ADC1 规则序列的相关信息；⑤ 开启 AD 转换器并校准；⑥ 读取 ADC 的值

```
void VoltageAdcInit(void)
{
  // 初始化IO口
  RCC->APB2ENR |= 1<<2;       // 使能 PORTA 口时钟
  GPIOA->CRL &= 0xfffffff00;  // 设置 PA.0 和 PA.1 为模拟输入

  RCC->CFGR &= ~(3<<14);      // 分频因子清零
  RCC->CFGR |= 2<<14;         // 6分频   SYSCLK/DIV2=12M ADC时钟设置为12M,ADC1最大时钟不能超过14M

  VoltageAdc1Init();          // ADC1 初始化
  VoltageAdc2Init();          // ADC2 初始化
}


void VoltageAdc1Init(void)    // ADC1 初始化
{
  RCC->APB2ENR |= 1<<9;       // ADC1 时钟使能
  RCC->APB2RSTR |= 1<<9;      // ADC1 复位
  RCC->APB2RSTR &= ~(1<<9);   // 复位结束

  ADC1->CR1 &= 0xf0ffff;      // 工作模式清零
  ADC1->CR1 |= 0<<16;         // 独立工作模式
  ADC1->CR1 &= ~(1<<8);       // 非扫描模式

  ADC1->CR2 &= ~(1<<1);       // 单次转换模式
  ADC1->CR2 &= ~(7<<17);
  ADC1->CR2 |= 7<<17;         // SWSTART：软件控制转换
  ADC1->CR2 |= 1<<20;         // 使用外部触发（SWSTART），必须使用一个事件来触发
  ADC1->CR2 &= ~(1<<11);      // 右对齐

  ADC1->SQR1 &= ~(0xf<<20);
  ADC1->SQR1 &= 0<<20;        // 1个转换在规则序列中，也就是只转换规则序列1

  ADC1->SMPR2 &= 0xfffffff0;  // 通道0采样时间清空
  ADC1->SMPR2 |= 7<<0;        // 通道0 239.5周期，提高采用时间可以提高精确度

  ADC1->CR2 |= 1<<0;          // 开启 AD 转换器

  ADC1->CR2 |= 1<<3;          // 使能复位校准
  while( ADC1->CR2 & 1<<3 );  // 等待校准结束

  ADC1->CR2 |= 1<<2;          // 开启 AD 校准
  while( ADC1->CR2 & 1<<2 );  // 等待校准结束
}
```

定时器：

```
Timer4Init( 9999,7199 );    // 10Khz的计数频率，计数到10000表示1s
```

```
/****************普通按键初始化函数*******************
* 通用定时器中断初始化
* 这里时钟选择为APB1的2倍，而APB1为36MHZ
* arr：自动重装值。
* psc：时钟预分频数
* 这里使用的是定时器4!
*******************************************************/
void Timer4Init(u16 arr, u16 psc)
{
  hour = 0, minute = 0, second = 0;

  RCC->APB1ENR|=1<<2;                         // TIM4时钟使能

  TIM4->ARR = arr;       //设定计数器自动重装值，10为1ms
  TIM4->PSC = psc;       //预分频器7200，得到10KHZ的计数时钟

  // 这两个要同时设置才可以使用中断
  TIM4->DIER|=1<<0;                           // 允许更新中断
  TIM4->DIER|=1<<6;                           // 允许触发中断

  TIM4->CR1 &=  ~(0x01);                      // 关闭定时器4
  TIM4->CR1 |= 0x01;                          // 使能定时器4

  MY_NVIC_Init(0, 2, TIM4_IRQChannel, 2); //抢占1，子优先级3，组2
}
```

（1）时钟使能（PCC->APB1ENR）：

STM32 中除非 APB1 的时钟分频数设置为 1，否则通用定时器的时钟是 APB1 时钟的 2 倍即 72 MHZ

RCC->APB1ENR|=1<<1;　TIM3时钟使能

RCC->APB1ENR|=1<<2;　TIM4时钟使能

（2）预分频寄存器（TIMx_PSC）：

对时钟进行分频，然后提供给计数器，作为计数器的时钟

72 MHZ / 7200 = 10 KHZ

（3）自动重装载寄存器（TIMx_ARR）：

设置计数器的自动重装载值

10 为 1 ms，10000 为 1 s

（4）中断使能寄存器（TIMx_DIER）：

TIM4->DIER|=1<<0;　允许更新中断

TIM4->DIER|=1<<6;　使能触发中断

（5）控制寄存器（TIMx_CR1）：

使能计数器

（6）中断分组设置：

设置嵌套向量中断控制器 NVIC，设置中断优先级、分组和中断服务函数

（7）编写中断服务函数：

中断服务函数用来处理定时器产生的相关中断，在中断产生后，通过状态寄存器（TIMx_SR）的值来判断此次产生的是什么类型的中断，然后执行相关的操作，这里是更新（溢出）中断

TIM4->SR &= ~(1<<0);　在处理完中断后向状态寄存器中相应位写 0，清除该中断标志

看门狗：

STM32 的独立看门狗 IWDG由内部专门的 40KHz 低速时钟驱动，即使主时钟发生故障，它也依然有效

启动独立看门狗的步骤：

(1) 向键值寄存器 IWDG_KR 中写入 0x5555，取消对预分频寄存器 IWDR_PR 和重装载寄存器 IWDG_RLR 的写保护，设置看门狗的分频系数和重装载的值，并由此计算看门狗的喂狗时间（4 × 2^分频数 × 重装载值 / 40）

(2) 向键值寄存器 IWDG_KR 中写入 0xAAAA，作用为使 STM32 重新加载重装载寄存器 IWDG_RLR 的值到看门狗计数器中，即实现独立看门狗的喂狗操作

(3) 向键值寄存器 IWDG_KR 中写入 0xCCCC，作用为启动 STM32 的看门狗，独立看门狗一旦启用，就不能再被关闭

使能看门狗后，在程序中必须间隔一定时间执行喂狗操作，否则系统将被看门狗复位

```
// 初始化独立看门狗
// prer：分频数:0~7(只有低3位有效!)
// 分频因子=4*2^prer，但最大值只能是256!
// rlr：重装载寄存器值，低11位有效
// 时间计算(大概)：Tout = ((4*2^prer) * rlr) / 40 (ms)
void IWDG_Init(u8 prer,u16 rlr)
{
  IWDG->KR=0X5555;    // 使能对IWDG->PR和IWDG->RLR的写
  IWDG->PR=prer;      // 设置分频系数
  IWDG->RLR=rlr;      // 从加载寄存器 IWDG->RLR

  IWDG->KR=0XAAAA;    // reload

  IWDG->KR=0XCCCC;    // 使能看门狗
}
```

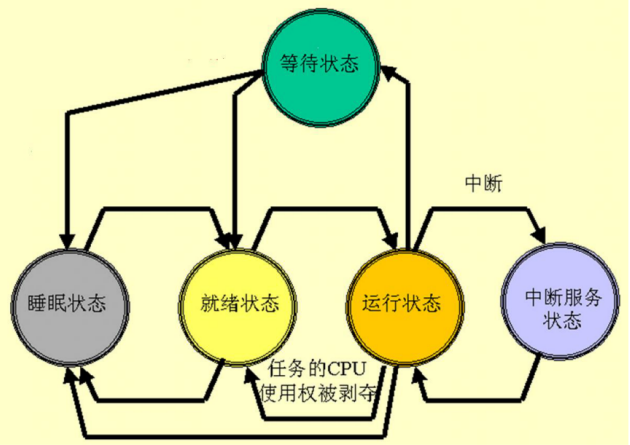
窗口看门狗 WWDG 通常用来检测由外部干扰或不可预见的逻辑条件造成的应用程序背离正常运行序列而产生的软件故障

启动窗口看门狗，并用中断的方式来喂狗的步骤：
(1) 使能 WWDG 时钟，WWDG 不同于 IWDG，IWDG 有自己独立的低速内部时钟，而 WWDG 使用的是 PCLK1 的时钟，使用前需要先使能时钟
(2) 设置 配置寄存器 WWDG_CFR 和 控制寄存器 WWDG_CR ，在时钟使能完以后，设置 WWDG 的配置寄存器和控制寄存器，对 WWDG 进行配置，包括使能窗口看门狗、开启中断、设置计数器的初始值、设置窗口值并设置分频数 WDGTB 等
(3) 开启 WWDG 中断并分组，设置完 WWDG 后，需要配置该中断的分组及使能
(3) 编写中断服务函数，通过该函数来喂狗，在中断服务函数里面也要将状态寄存器的 EWIF 位清空

第五章

μC/OS-II 是一个实时操作系统内核，它的任务调度是按抢占式多任务系统设计的，它只包含了任务调度、任务管理、时间管理、内存管理和任务间的通信和同步等基本功能，不包括输入输出管理、文件系统、网络等服务
μC/OS-II 规定所有任务的优先级必须不同，任务的优先级同时也唯一地标识了该任务
μC/OS-II 提供了四种同步对象，分别是信号量、邮箱、消息队列和事件，通过邮箱和消息队列还可以进行任务间的通信
为了消除多次动态分配与释放内存所引起的内存碎片，μC/OS-II 把连续的大块内存按分区来管理，每个分区中都包含整数个大小相同的内存块，不同分区之间内存块的大小可以不同
μC/OS-II 的特点：开源、可移植、可固化、可裁剪、占先式、多任务 ...
μC/OS-II 中的任务调度就是：查找准备就绪的优先级最高的任务并进行上下文切换

任务状态



睡眠态：任务驻留在程序空间(ROM或RAM)，还没有交给系统来管理的状态
就绪态：任务一旦创建就进入就绪态，准备运行
运行态：任何时刻只有一个任务处于运行态，就绪的任务只有当所有优先级高于它的任务都转为等待状态，或被删除后，才能进入运行态

等待状态：正在运行的任务可以通过 `OSTimeDly()` 或 `OSTimeDlyHMSM()` 进入等待状态。延迟时间到，立即强制执行任务切换，让下一个优先级最高并进入就绪态的任务执行

中断服务态：正在执行的任务是可以被中断的，被中断的任务便进入了中断服务态，响应中断后，正在运行的任务被挂起，中断服务子程序控制了CPU的使用权