**---------C语言内存与架构-------------**

**C语言内存管理？MCU内存管理？**

分为内存四区

**代码区**（Text Segment）

就是所有的可执行代码，编译为汇编指令后的结果存在这里，设置为只读属性

**常量区** .rodata段

字符串和const修饰的变量都被存在这里，只读属性防止被修改

**静态数据区**（Static Data Segment）

存放所有的全局变量和静态变量，初始化的放在**.data**段，未初始化的放在**.bss**段

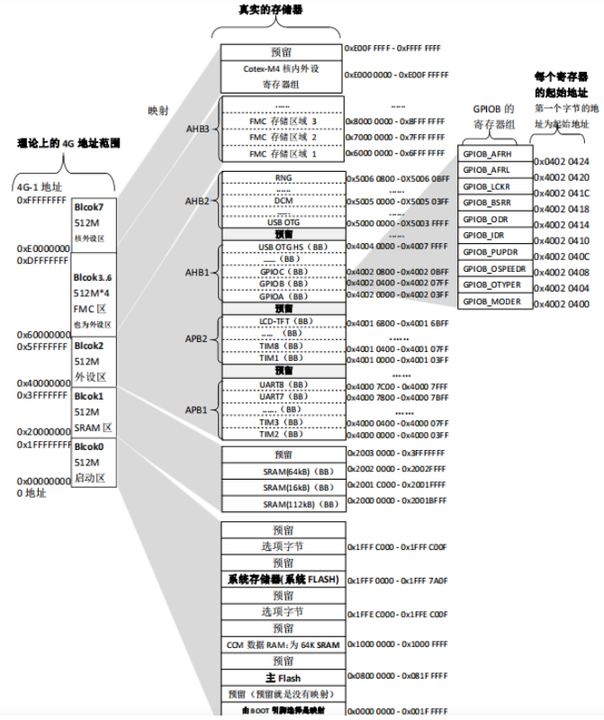
**堆区**（从低到高生长）

用户开辟，随用随释放，不然会内存泄漏

**栈区**（从高到低生长）

局部变量和形参以及返回地址存储的地方，随用随销毁，先进后出

**单片机常用的ARM M3核** 单片机被分为了8个段，每个段512M刚好4GB



**CODE(flash) SRAM 片上外设 FSMC12 34 FSMC寄存器 保留 内部外设**

代码段包括

**.text段** 包含程序中所有的代码函数以及中断向量表

**.rodata段** 只读的常量数据（const修饰的变量）

**.bss段** 未初始化数据段 存储未初始化的全局变量和静态变量

**.data段** 已初始化数据段 存放已初始化的全局变量和静态变量

**.Stack段** 栈段

**.heap** 堆段

**.reg** 寄存器段

**外设地址**是寄存器的**基地址**，寄存器的地址等于外设基地址+偏移量，每个外设的寄存器都有相应不同的控制位

单片机烧录时分为CODE RO RW RI ZI 段，

**Code** (inc. data)：包含代码以及与代码相关数据的大小总和

这里的代码指编译后的机器指令，存放在Flash中

**RO Data**：只读数据的大小，如程序中用const修饰的常量等 。

存放在 Flash里。

**RW Data**：可读写数据的大小，一般是已初始化的全局变量和静态变量 。

烧录时存在flash中（**掉电保留初始值**），运行时复制进入在SRAM里。

**ZI Data**：未初始化的全局变量和静态变量，程序启动时会被初始化为 0 。

存放在SRAM里。

**在程序中的栈空间和堆空间都是属于ZI-data区域的**

**内存泄漏？**

栈区：局部变量过大或者多层函数嵌套，让栈溢出

堆区：malloc内存后未释放

**什么是栈溢出？**

内存中栈和堆的地址是连续的，栈从高地址向低地址生长，堆从低到高生长，如果入栈元素过多就会影响到堆的内存，同样如果申请的动态内存(堆)太大，栈的内存也会被修改

**函数的参数压栈？**

对于ARM32架构，函数调用时前4个参数用通用寄存器（R0~3），剩余的用栈。对于ARM64架构，函数调用时前8个参数用通用寄存器(R0~7)，剩余的通过栈。

X86是涉及到 ebp、ebx、esi、edi等寄存器的保存

**局部变量未定义时为什么每次初始化的结果是不确定的？是个真随机数还是个伪随机数？**

**局部变量未初始化**

局部变量存储在栈中，是动态存储方式，如果声明了一个变量，系统给这个变量分配了一块内存，这块内存可能存有之前使用过的值，这个值可能是任意值，如果新声明的这个变量没有被初始化，就相当于系统把这块内存中之前存在的值赋给了新声明的变量，这个值可能是任意值，因此得到的就是一个随机值，这种结果无法预测，可能导致灾难性的后果。

**全局变量未初始化**

全部变量采取的是静态存储方式，在没有初始化之前，被放在BSS段中，操作系统在调用一个程序之前，会执行一段代码，这段代码会将BSS段中的空间全部清零，所以全局变量如果未初始化，其值就是0。

**嵌套调用时**：局部变量如未定义为静态变量的话会出现**悬空指针**

**auto static register volatile const关键字都有什么用？**

**Auto**修饰的变量被称作自动型变量，自动型变量是在动态存储去分配单元，函数返回的时候，编译系统会释放这些[存储单元](https://so.csdn.net/so/search?q=%E5%AD%98%E5%82%A8%E5%8D%95%E5%85%83&spm=1001.2101.3001.7020)，就是说函数结束后变量就会被释放

写局部变量时候实际上就是被修饰为auto型了，出函数就会被释放

**Static修饰的变量叫做静态类变量**，存储在内存中的静态区，直到程序结束才会被释放

**修饰局部变量时候**，作用域从函数变到了整个文件中

**修饰全局变量和函数时候**，作用域从整个工程变到了单个文件中，防止多个文件同名函数冲突

**Cons**t用来定义常量，此常量不可以**直接**被修改

**volatile**关键字叫编译器不要进行优化，也就是说让CPU每次都老老实实去内存中读取的寄存器中的值，慢点就慢点但是可以保证CPU读取的寄存器的值都是最新的！！！

STM32中外设的寄存器（如GPIO的各个控制位）都是用\_\_IO修饰，也就是volatile

**Static为什么只初始化一次？**

static的意思是函数第一次执行时对变量初始化，之后再调用就不再执行该变量初始化，而保留上一次执行的结果。

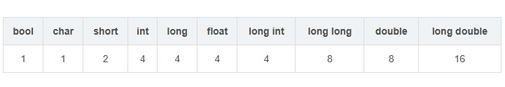
**extern的作用？**

Extern**函数**时，A文件想用B文件的某个函数时，没必要include，节省了编译时间

Extern**变量**时，同样是为了告诉编译器这个变量不要赋初值，他在别的文件中定义了

Extern和**不Extern**的汇编源代码的区别就是**有没有赋初值**

**Struct字节对齐？**



**条件①**：结构体当前大小%当前变量自身有效对齐字节数=0，如果无法对齐，则持续填充字节直至对齐。

**条件②**：结构体的总大小%结构体最大对齐字节数=0，如果无法对齐，则持续填充字节直至对齐。

**Int + char + short** = **4+1** +1 **+2 =** **|int|int|int|int|char|XXX|short|short|**

**九、结构体字节对齐的意义**

字节对齐很简单，但我们了解了规则后，更应该了解为什么要这样设计。

我们假设计算机总是从内存中取8个字节，如果一个double数据的地址对齐成8的倍数，那么一次内存操作就可以读或者写。

**但是如果这个double数据的地址没有对齐**，数据就可能被放在两个8字节块中，那么我们可能需要执行两次内存访问，才能读写完成。

显然在这样的情况下，是低效的,所以需要字节对齐来提高内存读写性能。

**十、关于字节对齐的一些思考**

**问：结构体到底该怎么写？如何保证字节对齐？**

答：根据前面的一些例子，相信大家也已经发现，其实只要按照字节大小的顺序排列就可以，无论**从大到小，或从小到大。**

之后，考虑到目前我们所使用的结构体成员变量大多是int与char[]类型。且int类型更多，更通用，所以为了易于扩展，也为了适配大多数人的习惯在结构体末尾扩展，将int类型放在末尾会更好一些。

如果无法在多人协同开发的过程中保证结构体变量大小的有序性，那么一种更加显式的方法是将相同类型的变量放在一起。这样也可以更很好的规避空间浪费的情况。

**问：既然为了节省空间，为什么不直接使用1字节对齐的方式呢？这样就不会存在空间浪费的情况。**

答：设置为1字节对齐，确实会节省空间，但这样对**存取效率**会有影响。

正如在最后一个例子中所抽象出的“变量”概念，1字节对齐造成的后果就是“变量”变多，那么在内存寻址方面的次数就会大大增加，耗时也会增加，这显然这不是我们想要的。

而且我们对速度的要求也往往大过内存，所以牺牲部分内存换取速度也是很值得的。

**C语言位段的作用？**

充分利用空间，弥补了内存对齐造成的内存浪费这个缺点。例如：我传递的多个信息都很小，只有几个比特位那么大，如果为每个信息都开辟一个字节的空间，这样会浪费很多超过信息本身的空间，位段就是让这些长度很短的信息尽可能地放在一起，节省空间。

**Define和Const区别？**

**Define强调字符的替换**，而且在预编译阶段就已经全部替换完成

有很多可以用的功能，比如替换函数中的内容，修饰CASE-break

而**Const强调对常量或原变量的！修饰！以及不可修改性**，在编译阶段完成，

**Const的特殊点在于可以修饰各类指针**

**指向常量的指针（指针可变，指向的值不可变）**

int value = 10;

const int\* ptr = &value; // ptr指向的值不可变

// \*ptr = 20; // 错误！**不能通过ptr修改**value

value = 20; // 正确！**直接修改value**是允许的

printf("新值: %d\n", \*ptr); // 输出20

int another = 30;

ptr = &another; // 正确！**ptr本身可以指向新地址**

**常量指针（指针不可变，指向的值可变）**

int num = 5;

int\* const ptr = &num; // ptr本身不可变

\*ptr = 10; // 正确！可以修改指向的值

printf("num = %d\n", num); // 输出10

int other = 8;

// ptr = &other; // 错误！ptr不能指向新地址

**指向常量的常量指针（指针和值都不可变）**

int id = 100;

**const int\* const ptr** = &id;

// \*ptr = 101; // 错误！不能修改值

// ptr = NULL; // 错误！不能修改指针

**共同点是都可以完成对某一常量的声明，比如常用的PI=3.1415926**

**TypeDef和#Define的区别？**

**（1）#define可以使用其他类型说明符对宏类型名进行扩展，但对 typedef 所定义的类型名却不能这样做。例如：**

#define INTERGE int;

unsigned INTERGE n; //没问题

typedef int INTERGE;

unsigned INTERGE n; //错误，不能在 INTERGE 前面添加 unsigned

**（2） 在连续定义几个变量的时候，typedef 能够保证定义的所有变量均为同一类型，而 #define 则无法保证。例如：**

#define PTR\_INT int \*

PTR\_INT p1, p2; //p1、p2 类型不相同，宏展开后变为int \*p1, p2;

typedef int \* PTR\_INT

PTR\_INT p1, p2; //p1、p2 类型相同，它们都是指向 int 类型的指针。

**程序编译过程？**

**编译器：**.c 预编译 .i 编译 .s 链接 .o

**链接器：**.exe 可调式

**预编译处理：**

所有引入的头文件内容拷贝到.c中

删除注释内容 用空格替换

将define宏定义进行替换

条件编译（#ifdefine）

**编译：**

将C语言转化为汇编代码，语法词法语义分析与符号汇总

**汇编：**

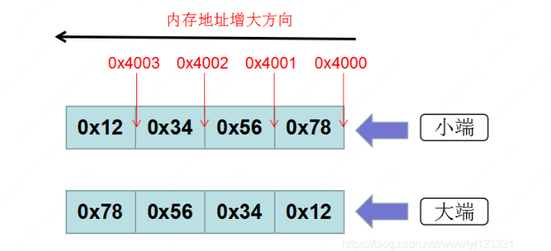
将汇编代码转化为二进制代码（指令）

**形成符号表**

**链接：**

合并段表 符号表的合并和重定位

**计算机大小端？**



**3级流水线是什么？**

pc代表程序计数器，CortexM的流水线使用三个阶段，因此指令分为三个阶段执行：

1.**取指**（从存储器装载一条指令）；

2.**译码**（识别将要被执行的指令）；

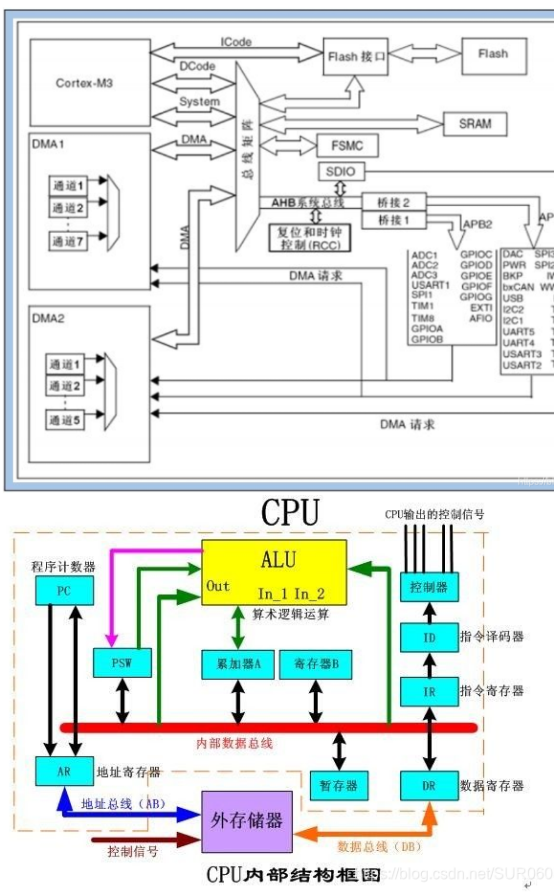
而**r15(pc)总是指向“正在取指”的指令**，而不是指向“正在执行”的指令或正在“译码”的指令。一般来说，人们习惯性约定将“正在执

行的指令作为参考点”，称之为当前第一条指令，因此pc总是指向第三条指令。当ARM状态时，每条指令为4字节长，所以pc始终指向该指

令地址加8字节的地址，**即：pc值=当前程序执行位置+8；**

3.**执行**（处理指令并将结果写回寄存器）。

**51和32的单片机架构？**



**Keil的代码优化？**

O0<O1<O2<O3

优化等级最低的时候，内存占用和执行效率是最差的，但是最贴合自己写的源代码，更好debug，打断点时候可以发现有些语句没法设置断点，实际上就是汇编语句被优化了

优化等级高的时候，内存占用和执行效率是最优的，同时也降低了功耗，但是不方便调试

**----------单片机与中断--------------**

**NVIC(嵌套向量中断控制器)是什么？**

用于为中断分组，从而分配抢占优先级和响应优先级；

而且包括了一些API函数，用于中断的优先级设置等操作

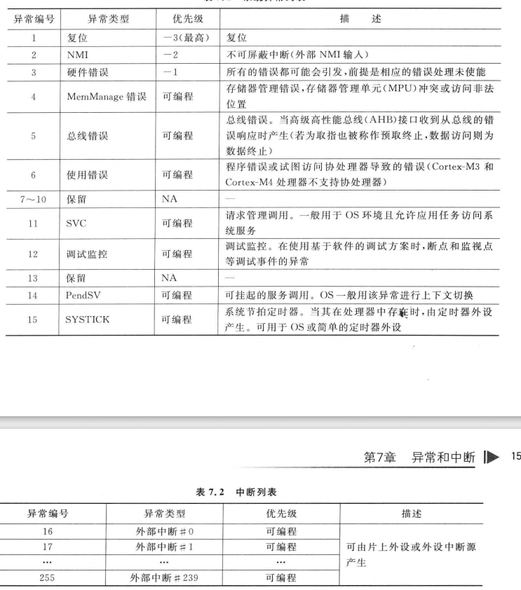


**中断向量表是什么？**

**指针就是地址，地址就是指针**，跳转到相应的函数需要指针，也就是地址

而CortexM的向量表把所有的中断和对应的地址都罗列了出来，每个中断号都用汇编存放着相应的中断服务函数地址的跳转指令，CortexA则是把中断分为了7大类（Reset、不识别、SWI软中断进入内核空间、PrefetchAbout预取指令出错时产生的中断、DataAbort访问数据出错中断、IRQ外设中断、FIQ快速中断），

在STM32单片机中，对M支持的中断进行了裁剪，F1有16个内核中断和68个外部中断，例如有一个DebugMonitor内核中断就是用来调试的中断，但现在GDB调试方便得多，而像SVCall和PendSVC中断则是在RTOS的任务切换中起作用

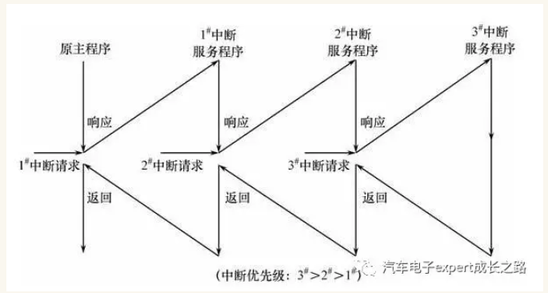


**中断响应流程？**

多级中断嵌套情况下先判断中断抢占与响应的优先级，之后保存现场，设置PC跳转中断服务函数，恢复现场回到主程序中

**中断嵌套是什么？**

就是在中断中被另外一个中断打断，进入了保存现场等流程



不同的外设中断源必须有不同的优先级

包括响应优先级和抢占优先级

在中断ISR中必须打开CPU全局中断

而且DMA也可以响应中断，协助CPU处理数据，避免中断嵌套次数过多

**中断内为什么不能延时？**

**表层原因：**

**中断的优先级高**：中断服务程序（ISR, Interrupt Service Routine）通常在中断发生时立即执行，打断当前正在执行的主程序或其他低优先级的中断服务程序。

**执行时间应尽可能短**：为了确保系统能够快速响应其他中断，ISR的执行时间应该尽可能短。长时间的中断处理可能会导致其他中断被延迟处理，甚至丢失。

**深层原因：**

**延时函数通常依赖于硬件定时器**，比如STM32的Delay用的就是systick，在中断中不断增加uw计数，计满了退出中断（delay结束），属于最低级的15级定时器，但在中断中写延时函数的话，相当于是高优先级的任务（中断服务函数）在等待低优先级任务（延时函数）完成，这相当于一个优先级反转的问题，此时在高优先级中断里面，低优先级的systick无法进入，也就无法增加uwtick的值，导致程序卡死

**解决：**

1不要在中断里面写延时

2使用硬件计时器，把Delay的Timebase改为其他的高优先级计时器

3提升systick的优先级

**保护现场具体保护的是什么？**

**保护所有寄存器的值**与**栈，也称作当前进程的信息**

**FreeRTOS**通常都是将所有寄存器的值保存到RAM内存中的栈空间中，例如保存CPU里的16个当前值寄存器，当中断任务结束重新从RAM内存中取值重新执行。

**如果是任务1里面调用任务2**，此时需要选择性保护寄存器的值，如在任务1里调用任务2的时候需要使用R0 R1进行传递参数，此时R0 R1不需要保护(因为本来就是要拿到任务2里面用的)，但其他的寄存器需要进程保护

**Systick是什么？优先级高还是低？**

SysTick：系统定时器是一个 **24bit 的向下递减的计数器**，计数器每计数一次的时间为 1/SYSCLK，一般我们设置

系统时钟 SYSCLK 等于 72M .它只能递减，存在于内核，嵌套在NVIC中，所有的Cortex-M内核的单片机都具有这个定时器。

“SysTick 的中断优先级在 STM32 上是完全可配置的，默认通过 SysTick\_Config() 设置为最低的 15。这个数值 15 会根据当前系统的中断优先级分组（Group）被解析成具体的抢占优先级和子优先级，但结果都是该分组下的最低或接近最低级别。例如2号分组 11 11 3优先级仍然比别的低

因此，默认情况下，**SysTick 的优先级低于大多数配置合理的外设中断**。但这并非固定不变，我们可以根据需求（比如需要极高精度的定时）将其设置为更高的优先级。关键点在于理解中断优先级分组规则以及数值 15 在不同分组下的含义，它决定了 SysTick 相对于外设中断的实际优先级位置。

**在 RTOS 场景下，默认低优先级确保了外设能及时响应，而将任务调度（由 SysTick 触发）放在后面执行。”**



**单片机启动流程？SOC启动流程？**

所谓的启动流程就是单片机断电再启动时到main()之前发生的事，

在startup.s中能看到相关汇编代码

先根据boot选择，确定起始地址

常用的就是**从flash启动**，既boot设置PC指针指向0X08000000

之后取出MSP栈顶指针，然后从向后4偏移地址把**reset\_handler()函数**的地址赋值给PC

开始执行**初始化程序**，包括时钟和看门狗初始化，内存和堆栈初始化，中断向量表的偏移地址设置初始化，

最后**跳转到main执行代码**

对于其他ARM系列的只是地址不同，从0X0000 0000启动，包括MSP初始栈顶地址和利用PC跳转指向Reset\_handler

**单片机都有哪些寄存器？有什么用？**

**R0‐R12** 都是 32 位通用寄存器，用于数据操作。但是注意：绝大多数 16 位 Thumb 指令只能访问 R0‐R7，而 32 位 Thumb‐2 指令可以访问所有寄存器。首先在程序的调用过程中，R0-R3寄存器都存储着调用函数的传参值，下一个函数用到了几个参数，就提前将参数存储在r0-r3里面，如果超出4个参数，就将多余的参数Push进堆栈中，在进入到调用的函数里面后，在Pop出栈给其他r4-r12寄存器。

**R12也叫IP寄存器，scratch寄存器**

**在ARMv6及以前的版本中用于存储临时数据，而在ARMv7及以后的版本中已经不再使用。R13**是堆栈指针MSP和PSP，永远指向栈顶，进入子程序和退出子程序时值必须相等，同时使用RTOS时候会用到PSP来对不同任务的栈区进行指向

**R14**是连接寄存器LR，一些1级嵌套的程序只需要把返回地址放到LR寄存器就可以了，如果还有一级子程序就需要先把LR压入栈中，再把1级子程序的返回地址存入LR，这样优化了性能，在 RISC 处理器中，为了强调访内操作越过了处理器的界线，并且带来了对性能的不利影响，给它取了一个专业的术语：**溅出**。

**R15**程序计数寄存器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。(三级流水线架构)

**特殊功能寄存器**：包括程序状态字寄存器组（PSRs）中断屏蔽寄存器组（PRIMASK, FAULTMASK, BASEPRI）控制寄存器（CONTROL）

xPSR：记录ALU标志(0、进位、负数、溢出标志)，执行状态与正在服务的

中断号

PRIMASK：除能所有的中断（除了不可屏蔽中断）

FAULTMASK：除能所有的fault，NMI不受影响

BASEPRI：除能所有优先级不高于某个具体数值的中断

CONTROL：定义特权状态

ARM属于RISC架构，使用三级流水线，一个时钟频率同时执行三个部分，包括取指(从存储器装载一条命令)译码（识别将要被执行的指令）和执行（处理指令并将结果写回寄存器）而PC总是指向”正在取指”的指令，

**单片机Flash起始地址在0x0800 0000?**

主flash启动时，是从0x0800 0000启动，但是M3，M4内核芯片上电复位后，要固定从0x0000 0000地址读取中断向量表，获取复位中断服务程序的入口地址后，进入复位中断服务程序，**其中0x0000 0000是栈顶地址，0x0000 0004存的是复位中断服务程序地址**。

这是因为STM32**不仅可以从内部Flash启动，还可以从系统存储器**（可以实现串口ISP，USB DFU等程序下载方式，这个程序是ST固化好的程序代码）**和从内部SRAM启动**，

我们将内部Flash安排到0x0000 0000显然是不行的。这样会导致系统存储器或者内部SRAM无法重映射到0x0000 0000了。

M7内核芯片比较灵活了，改变了固定从0x0000 0000地址读取中断向量表的问题，以STM32H7为例，可以从 0x0000 0000 到 0x3FFF 0000 所有地址进行启动。

**单片机看门狗是什么？**

本质上是个定时器，如果长时间没有重置计数器，就可以视为程序卡死了或进入死循环，就会产生复位信号重启程序

主要分为独立看门狗和窗口看门狗

**独立看门狗**：独立低速时钟线，内部递减计数器减到0既复位，超时时间与低速时钟频率与预分频系数和重装值有关

**窗口看门狗**：更为严格，喂早了喂晚了都会复位，使用APB1总线时钟，通过多位寄存器比较器与门或门等实现

https://blog.csdn.net/m0\_60503814/article/details/136058963

**单片机断点调试是怎么实现的？调试器调试的原理是什么？**

**硬件断点**硬件断点由正在使用的芯片组实现。在芯片级，它们是比较器，将提取的指令与外设寄存器中配置的指令进行比较。当指令匹配时，硬件将触发调试事件，暂停内核或生成异常。对于任何给定的芯片，通常都有少量固定数量的硬件断点可用。如果你曾经使用过一个没有断点的调试器，这就是原因！

**软件断点**另一方面，软件断点是由调试器实现的。他们靠修补您试图通过以某种方式触发调试事件的指令来执行的代码。这是通过注入断点指令来实现的，或者在不支持断点指令时，通过插入导致中断内核的错误的指令来实现。软件断点设置不在ROM或者FLASH里面，因为他的实现是通过替换断点处的内容再通过比较器对比实现，ROM和FLASH无法进行写入操作

**单片机有大量数据需要传输该怎么办？**

Ymodem协议

包含数据包和包长度信息，以及相应的CRC校验等

**单片机使用浮点计算对中断影响?**

**资源占用**：浮点运算通常需要使用到浮点库，如CMSIS-DSP库或math库，这些库中的函数执行浮点运算时，由于是软件模拟，会占用较多的CPU资源。这意味着在进行浮点计算时，CPU可能无法及时响应其他任务或中断请求，从而影响到中断的响应速度。

**执行速度**：软件模拟的浮点运算速度较慢。当中断服务程序（ISR）中包含浮点运算时，执行这些运算所需的时间会增加，这可能会导致中断处理延迟，影响整个系统的实时性能。

**中断频率限制**：对于某些型号的STM32微控制器，如STM32 F103，其中断频率性能极限为500KHz。如果中断服务程序中的浮点运算导致中断处理时间过长，那么在高中断频率下，微控制器可能无法达到这个性能极限，从而影响到中断效率。

**存储容量限制**：对于需要运行复杂浮点DSP算法的情况，STM32的存储器容量可能有限，可能需要外部RAM或闪存存储器支持。这种外部存储的使用可能会进一步降低数据处理和中断响应的效率。

**单片机IAP固件升级功能是什么？**

广义来说就是让产品可以自烧录更新版本，实现上要依靠RESET\_handler等函数，在跑main之前设置一个是否更新的标志位，如果标志位挂1需要更新时，后面接上生成bin文件和写flash擦写代码的程序。

**单片机最困难的DEBUG?**

**单片机可以跑Linux吗？**

有一个uclinux，要裁剪内核，网络之类的不需要，裁剪下去

实际上RTOS也算是精简版的操作系统，所有系统调度的内容用汇编编写，实现了基础功能的同时没占用过多内存，即便是SOC也需要外挂存储器运行Linux

**-----------ARM外设与算法------------**

**DMA原理？**

**外设和存储器之间或者存储器之间的高速数据传输**

**1 数据的源地址**

**2 数据传输位置的目标地址**

**3 传递数据多少的数据传输量**

**4 进行多少次传输的传输模式**

**DMA所需要的核心参数，便是这四个**

该寄存器控制着 DMA 的很多相关 信息，包**括数据宽度、外设及存储器的宽度、通道优先级、增量模式、传输方向、中断允许、 使能**等都是通过该寄存器来设置的。所**以 DMA\_CCRx 是 DMA 传输的核心控制寄存器**

无DMA时候，CPU要通过总线获取外设存的数据，在将数据转发到内存中，占用时间片、总线，并且程序无法向下执行

有DMA的话，外设获取的数据直接通过地址总线存入内存，CPU不需要管，认真跑程序就可以了，需要数据的时候直接去内存读DMA已经搬运好的就可以

**影响DMA速率的因素？**

1、DMA挂载在**AHB总线上，主频越高总线频率越高**，速率越快

2、总线如果被其他外设(SPI/UART)或CPU占用，影响读取速率，**提升DMA中断优先级可以优化,但别的外设也会被影响**

**3、DMA位宽**也会影响

stm32是32位处理器，所以一个word就对应32位，一个half word就对应16位。根据需要传输数据的大小选择对应Data Width即可。

例如我使用一个ADC的IC进行数据采集，**ADC的精度是12位的**，所以我的Data Width选择half word就够用了。

**4、传输是否对齐**也会影响

假设一个系统的数据总线宽度为32位，即一次可以传输4个字节。如果一个数据的起始地址是0x1003，那么DMA控制器在访问这个数据时，就需要两次访问：第一次访问0x1000，第二次访问0x1004。这样一来，就浪费了一次内存访问，降低了传输效率。

**缓存DMA一致性？**

在SOC或者F7H7此类的MCU中通常带有CACHE缓存来加速数据读写，但是一段数据(如SRAM)如果同时被如DMA等内存总线和CPU读写的话，容易出现CPU读到的数据往往是旧数据，新数据已经被DMA修改为别的值了

解决方法：写数据后顺便把CACHE清除

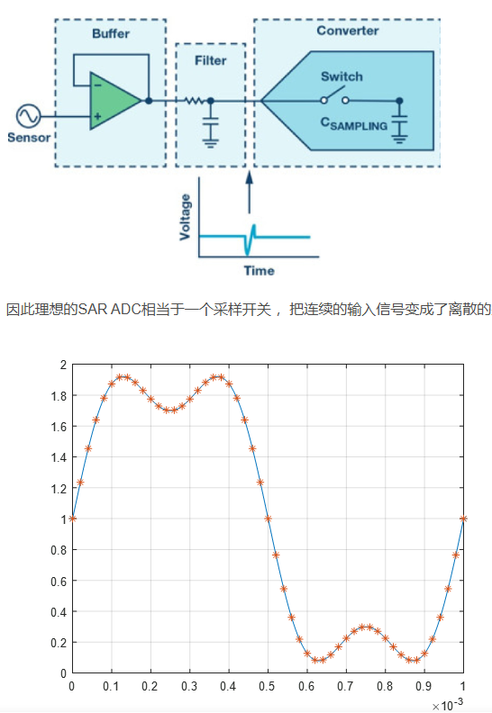
将CACHE改为透写模式

有些SOC厂商做好了硬件上的解决办法

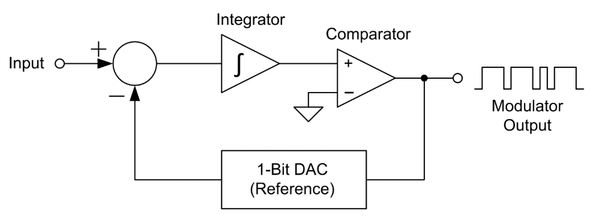
指令缓存Icache不用管，数据缓存D-Cache就是解决CPU加速访问SRAM。

**SAR-ADC和sigma delta ADC？**

**SAR**:逐次逼近，寄存器



**sigma**：Σ在数学中代表积分（求和），delta：Δ代表差分（求差），这两种过程是Σ-Δ ADC中的主要操作。



**ADC的位数/分辨率/精度是什么？**

**位数（分辨率）**： ADC的分辨率通常以输出二进制的位数表示，位数越多分辨率越高 F4的SAR ADC支持12位， 10 位， 8位和6位精度， F1只支持12位。

ADC的数字存储是12位的的话，“111111111111”=4095，转换的电压范围是0v-3.3v的话，转换器就会把0v-3.3v平均分成4096份。

精度：和位数（分辨率）不是一个概念，

**增量式PID和位置式PID区别？**

P：根据误差\*系数=调节量

I：消除稳态误差

D：**抑制超调 调节越大 –的越多 调节曲线越缓**

**位置式PID优缺点：**

**优点：**

①位置式PID是一种非递推式算法，可直接控制执行机构（如平衡小车），u(k)的值和执行机构的实际位置（如小车当前角度）是一一对应的，因此在执行机构不带积分部件的对象中可以很好应用

**缺点：**

①每次输出均与过去的状态有关，计算时要对e(k)进行累加，运算工作量大。

**增量式PID优缺点：**

**优点：**

①误动作时影响小，必要时可用逻辑判断的方法去掉出错数据。

②手动/自动切换时冲击小，便于实现无扰动切换。当计算机故障时，仍能保持原值。

③算式中不需要累加。控制增量Δu(k)的确定仅与最近3次的采样值有关。

**缺点：**

* 积分截断效应大，有稳态误差；

②溢出的影响大。有的被控对象用增量式则不太好；

**驱动开发？**

DAC3283芯片，要对芯片内一些寄存器进行配置，比如这颗IC就有32个寄存器一共32字节的位控制，使能，传输字节宽度和采样率等进行设置

**-------------通信协议----------------**

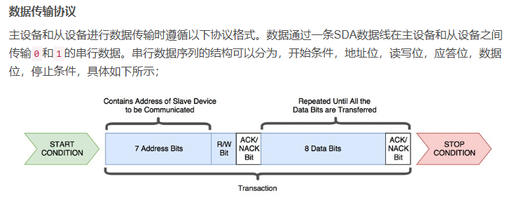
**I2C、SPI、UART 接收方式有哪三种？**

轮询，中断和DMA

**项目中的2.4g图传实现？**

**I2C的协议栈？**

起始位+地址位+读写位+ACK应答+数据位+停止位



**Modbus的协议栈？**

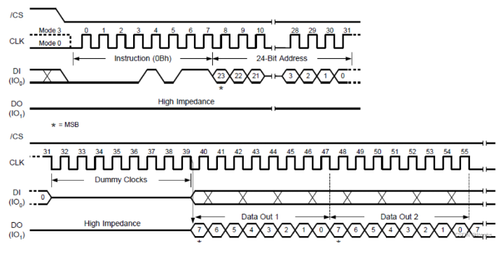
帧结构 = 地址 + 功能码+ 数据 + 校验，常用Modbus调试精灵直驱

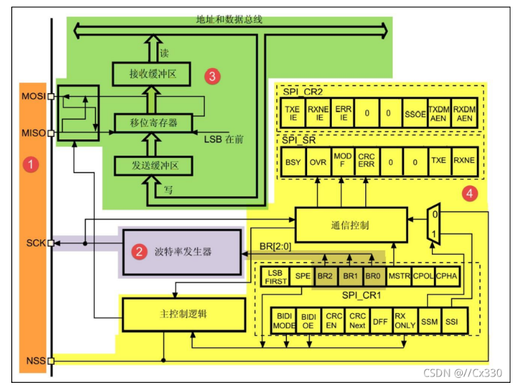
**串口的协议栈？**

起始位（1bit）+ 数据位（5~8bit）+ 奇偶校验位（1bit）+ 停止位（1~1.5bit）

**SPI的协议栈与内部结构？**

片选选择位+指令位（页写入等）+地址位+数据

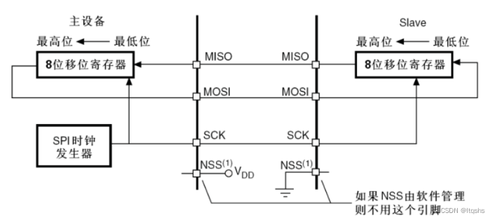




**SPI 的从是否可以主动读写主的数据？**

SPI---serial串行

SPI实际上不存在读写的概念，他是主从设备之间的数据交换，基本结构是主设备和从设备都有一个8位移位寄存器来实现的



**SPI四种模式？**

(1) CKPOL (Clock Polarity) = CPOL = POL = Polarity = （时钟）极性(2) CKPHA (Clock Phase)   = CPHA = PHA = Phase = （时钟）相位

模式0：CPOL= 0，CPHA=0。SCK串行时钟线空闲是为低电平，数据在SCK时钟的上升沿被采样，数据在SCK时钟的下降沿切换

模式1：CPOL= 0，CPHA=1。SCK串行时钟线空闲是为低电平，数据在SCK时钟的下降沿被采样，数据在SCK时钟的上升沿切换

模式2：CPOL= 1，CPHA=0。SCK串行时钟线空闲是为高电平，数据在SCK时钟的下降沿被采样，数据在SCK时钟的上升沿切换

模式3：CPOL= 1，CPHA=1。SCK串行时钟线空闲是为高电平，数据在SCK时钟的上升沿被采样，数据在SCK时钟的下降沿切换

**SBUS协议栈？**

**CAN总线仲裁？**

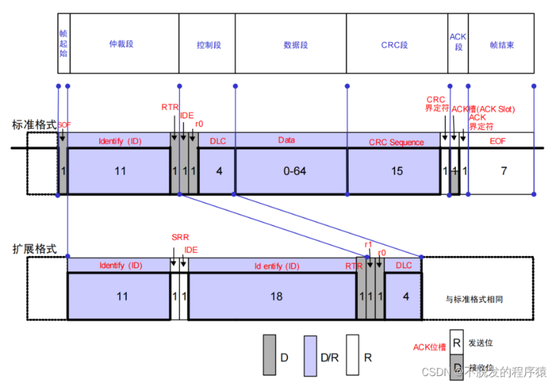
显性电平的逻辑为0，隐形电平的逻辑为1，这样才能和逻辑运算上的线与对应

关键在于显性电平覆盖隐性电平，线与机制，ID小优先级高获得总线控制权

如果ID前面都一样无法仲裁，则有对标准帧和拓展帧的不同位比较

**CAN总线报文？**

可以分为5种帧 数据帧 遥控帧 错误帧 过载帧 帧间隔



**TCP是什么，握手是什么？**

DNS 查询

Mac 地址查询

数据传输——套接字来帮忙

三次握手与四次挥手

**----------RTOS与Linux---------------**

**RTOS的任务创建原理？**

**任务的创建：**

1、**定义任务使用的栈，和主栈分开**

2、**定义任务函数**，实际跑的代码

(Vtaskdelay用到了挂起函数，实际上就是把任务从就绪队列挂载到阻塞队列中)

3、**定义任务控制块**，任务的优先级，链表结点，名称和栈顶指针等初始化

4、**任务创建函数**：包括指向任务函数的**函数指针**，人物名字任务栈深度和起始地址，任务函数的形参，任务控制块指针（地址）

**创建分为动态和静态**，动态创建时，任务控制块和栈的内存是创建任务时动态分配的，任务删除时内存可以释放。静态创建时，任务控制块和栈的内存需要事先定义好，任务删除内存不能释放

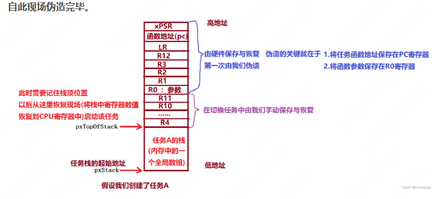
1、**创建任务内核**：定义栈顶指针，将任务栈起始地址+偏移量设置为栈顶地址

2、同时通过&0x07将栈顶地址**实现8字节对齐**（32位4字节系统适配）

3、**任务名字**存储在任务控制块数组中

4、**初始化结点**，将他插入到相应的任务结构体中

5、**初始化任务栈**，先伪造现场，其他的硬件帮我们保存，R4-R11需要自己保存



6、将**任务函数地址保存在PC寄存器**，将函数参数保存在R0寄存器，其他的存为0

最后返回栈顶指针存入任务结构体中的“栈顶指针”变量中，下一次可以直接恢复现场启动第一个任务

最后一步将任务句柄指向任务控制块，就可以开始控制这个创建好的任务了

**任务就绪链表：**

实际上就是一个List类型的数组，大小由最大任务优先级决定，同样优先级的任务挂载在同一个链表中

将任务控制块添加到就绪链表中，**其实就是调用一个链表插入结点函数将任务挂入链表之中。**

**RTOS的任务切换原理？**

**任务调度器启动（任务调度的核心是保护和恢复现场）：**

**调度器第一次启动**由**vTaskStartSchedule**r完成

1. 配置Pendsv和systick**中断优先级**
2. 设置主堆栈指针**MSP**的值
3. 使用**CPS指令**使能全局中断
4. 调用**SVC中断**去启动第一个任务

用到了我们之前说到的伪造现场中**已经伪造好的现场**

将**任务栈顶**指针赋给R0

从栈里**恢复R4~ R11**数值到CPU R4~R11寄存器中(**ldmia**)

将r0的值(任务的栈顶指针)更新到psp(任务的栈指针)，准备触发中断返回恢复硬件保存的那8个寄存器

1. 设置**basepri**为0，既开启所有中断
2. 最后将R14后四位或0x0D,让**硬件退出时使用PSP进程堆栈指针**完成出栈

当把这个值LDR到PC时候，触发中断返回，返回线程模式，不然此时在SVC终端里面用的是MSP主堆栈指针



**以后的任务切换：**

由Pendsv中断来完成，之前设置过Pendsv最低优先级，进入到任务切换函数 **taskYIELD()**任务切换函数后，调用System control block**寄存器悬起PENDsv**

之后**没有其他中断时候才会响应pendsv终端服务函数**

**假如A任务运行时，发生PendSV中断要切换为B任务**

1. 中断瞬间硬件自动保存现场，PSP将Xpsr、PC、R14、R12、R3、R2、R1、R0寄存器存入当前未切换任务(任务A)的栈中
2. 将PSP的值存入R0中，开始软件保存R4-R11

将任务控制块指针存入R2寄存器

1. 软件保存R4到R14到任务A的栈里(stmdb(PUSH))

**此时任务A的现场已经保护完毕了**

之后将此时的栈顶指针存入任务A的任务控制块的（str r0,[r2]）

第一个成员变量（\*pxTopOfStack）中，方便后续恢复现场

1. **入栈R3与R14到主栈中，**因为之前在SVC函数中已经在R3中保存了pxCurrentTCB的地址，而R14也已经被更新为了一个特殊值（中断返回寄存器），而后面的函数（vTaskSwitchContext）要调用函数的话，LR会被修改，所以要对他进行保护（R3同理）
2. **开启basepri中断保护，保护列表中的临界区资源，防止其他中断打断PENDsv**
3. **调用switchContext函数，切换为当前优先级最高的任务，更新pxCurrentTCB**

**此时已经让pxCurrentTCB指向了任务B函数**

1. **关闭basepri中断保护，退出临界区**
2. **从主栈中恢复**R3 、R14数值到CPU中
3. 取出下一个任务（任务B）的栈顶地址存入R0寄存器中
4. **软件恢**复任务栈的R4-R11到CPU中
5. **将PSP指向任务B的栈顶**

（再发生中断的话，以PSP为基地址将各寄存器加载到CPU中）

12、最后**退出PENDsv中断**，

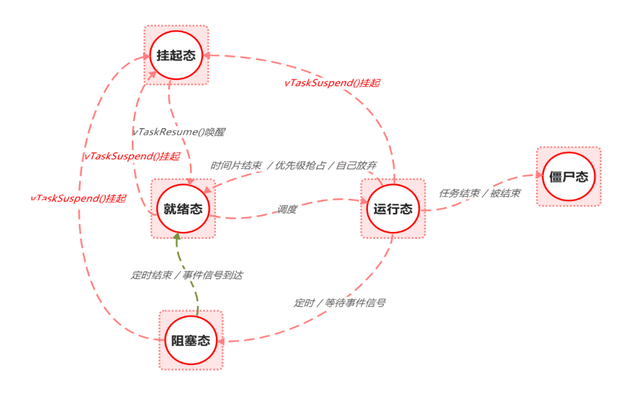
通过bx r14(之前被设置为了特殊的中断返回值)返回到线程模式、

**此时已经被切换到了B任务**，由PSP维护B的栈

**RTOS中的就绪态运行态阻塞态挂起态？**

实际上就是不同链表的切换，结点就是任务控制块中定义的结点，指向每一个任务函数

每个链表中有不同任务交给调度器根据优先级调度



**RTOS使用PENDSV的优势？**

PendSV是可悬起系统中断，具有【**缓期执行**】特征。

PendSV一般被嵌入式OS用于 **任务调度的上下文切换**。

使用PendSV时，一般会将PendSV的**优先级设置为最低**，让外部IRQ先执行，等到没有外部中断后，在执行上下文切换。

嵌入式实时操作系统的【实时】概念，并不仅仅指应用程序、任务的调度实时，而是指整个系统的实时性高，具备【可剥夺】特点，当有高优先级的中断、任务具备执行条件时，立刻中断当前正在运行的任务，去执行高优先级的IRQ和高优先级任务。

**RTOS中Systick的作用？**

**RTOS如何实现的高实时性？**

**任务调度**

RTOS通过优先级驱动的调度策略来实现任务的高效调度。它通常采用基于优先级的抢占式调度算法，允许高优先级任务抢占低优先级任务的CPU时间，从而确保关键任务能够及时执行。此外，RTOS还支持轮询调度等多种调度算法，以适应不同应用场景的需求。

**优先级管理**

任务的优先级是决定实时性能的重要因素。RTOS支持多级优先级，以便对任务的重要性进行细化控制。静态优先级在系统设计时确定，并在运行时保持不变，适用于实时性要求非常高的系统。动态优先级则根据系统运行情况动态调整任务优先级，以应对优先级反转等复杂情况

**中断处理**

中断是RTOS实现实时性的另一大关键。RTOS能够快速响应外部中断，并在中断服务程序中执行必要的处理。中断优先级机制使得高优先级的中断能够打断低优先级的中断处理，确保关键硬件事件得到最及时的响应。此外，RTOS还对中断的响应时间和任务切换时间进行了优化，以减少中断处理时间和任务切换的开销。

**定时器和时钟管理**

定时器和时钟管理是RTOS实现周期性任务和延时任务管理的基础。通过高精度的硬件时钟和定时器，RTOS能够精确地控制任务执行的时间。定时任务允许RTOS设置周期性定时器来定时触发任务执行，这对于控制应用（如发动机点火控制）非常关键。延时和超时处理则允许RTOS为任务设置超时时间，在超时发生时采取相应的措施（如重新尝试或触发备用方案），以避免因任务卡死而影响系统的实时性。

**同步与通信机制**

RTOS提供了多种同步与通信机制，如信号量、消息队列、邮箱、事件标志组等，以实现任务之间的同步和数据交换。这些机制对于实现复杂的实时应用至关重要，它们能够确保任务之间的协调一致运行，避免数据竞争和死锁等问题。

**资源管理**

常用的动态分配内存，5种动态内存管理方法：heap1-5

RTOS提供对硬件资源的有效管理，如内存、处理器时间等。通过合理的资源分配和调度，RTOS能够确保系统的高效运行。固定内存块分配和内存池等策略能够加快内存分配和释放的速度，从而满足实时性要求。

**RTOS消息队列是什么？有什么优点？**

**创建一个队列**：为队列分配内存，初始化队列结构体里面的成员变量，比如队列头，队列内存长度和消息数量等信息，创建过程如果涉及重置旧队列还包括对有写入队列需求的任务状态的判断

**向队列发送信息（入队）**：xQueueGenericSend()与xQueueGenericSendFromISR()函数为实际执行函数，一个是在任务中调用的，一个是在中断中调用的

写队列的时候先关中断（保护临界资源）

**从队列读取消息（出队）：**xQueueReceive()以及xQueueReceivefromISR(),以及读取后是否删除队列中读过的信息

**消息如何出队(依靠pcReadFrom指针）：**prvCopyDataFromQueue函数，通过移动pcread指针，来实现对环形队列头元素的读取

**RTOS的信号量？**

**信号量**是一种实现任务间通信的机制，可以实现任务之间同步或临界资源的互斥访问，实际上也是一种特殊的队列，依靠**uxMessagesWaiting（队列消息数）**

**二值信号量**：使用全局变量作flag时很浪费CPU资源，而信号量的引入保证了任务不会在无法获取响应信号资源时候进入挂起态，节省CPU资源

**获取信号量源码**：1.获取信号量资源数uxMessagesWaiting(在二值信号量中只能是0/1，代表信号量为空或满）

2.判断信号量是否有资源uxSemaphoreCount >0 (有资源才能被获取)

3.如果信号量有资源，uxSemaphoreCount 计数值减一

4.如果信号量无资源

(1).如果设置的不等待，等待时间为0，则即可返回获取信号量失败的错误

(2).如果设置的等待时间为0-portMAX\_DELAY，则判断任务是否超时，若未超时，即刻阻塞。

(3).如果设置的等待时间portMAX\_DELAY，则任务一直阻塞，直到信号量有资源才会被唤醒。

**计数信号量：**与二值相同，只不过是长度大于0的队列，当长度为1时与二值队列一样

例如停车场的车位，有车位才能让车进来（获取信号量），没车位就得等待（阻塞等待）

**互斥信号量**：重点在于优先级继承，能降低优先级反转带来的影响

比如两个任务都需要使用串口发送，就可以用互斥量来让低优先级的任务抓紧使用串口，用完之后能释放资源给高优先级任务

**优先级继承函数**：xTaskPriorityInherit()

**递归互斥信号量**：实际上就多了一个谁上锁由谁解锁的功能，防止不同的任务对同一个资源多次上锁或者多次解锁

**RTOS中的锁？**

**中断锁**：例如basepri寄存器，屏蔽优先级不高于x 的中断

**调度锁**：任务挂起Suspend和resume函数

**互斥锁**：DMA和cache的先后存入，对临界区资源的争夺

RTOS任务中的同步事件，A任务等B任务的资源时候就是进入了阻塞状态

实际上写FLAG也算一种互斥锁

**RTOS如何避免死锁？**

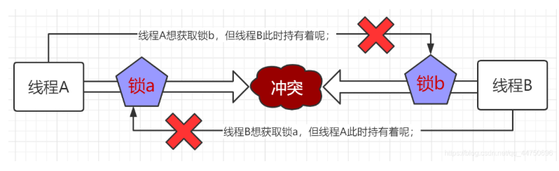
**死锁只有同时满足以下四个条件才会发生：**

互斥条件：两个形成死锁的进程拥有的资源不是共享的，CPU资源是共享的

持有并等待条件：A拥有资源的同时又想申请资源，他拥有的资源被别人申请，他申请的资源被别人占有

不可剥夺条件：自己使用完成之前不会释放资源

环路等待条件；



解决：资源有序分配法

两个进程申请资源的顺序要一致，这样就把环路打破了，不可能出现二者互相被锁的情况

**RTOS的互斥和同步怎么理解？**

**1.同步**

比如说，买包子

我要去买包子，如果包子店没有包子了，则需要等待卖包子的把包子做出来我才能买到包子，这个等待的过程就叫做同步。(在实际应用中：一个采集数据的传感器任务，一个处理数据的任务，则处理数据的任务需要等待传感器去采用数据，则在FreeRTOS系统中等待不能干等着，在该任务等待的过程中，CPU转而可以去执行其他任务，则就可以提高效率，则就是队列的阻塞机制)

**2.互斥**

比如说，抢厕所

厕所只有一个，一个人进去上了，另一个人也要上，则必须等待前人上完厕所才能上，等待的过程就是同步，而保护厕所的过程叫做互斥，则厕所就是所谓临界资源，同一时间只能一个人使用厕所，当然前人上完厕所应该提醒等待的人，厕所用完了可以上了，其中本质也是阻塞机制。

**RTOS优先级反转问题？**

**优先级反转（Priority inversion）**

优先级反转是指一个低优先级的任务持有一个被高优先级任务所需要的共享资源。高优先任务由于因资源缺乏而处于受阻状态，一直等到低优先级任务释放资源为止。而低优先级获得的CPU时间少，如果此时有优先级处于两者之间的任务，并且不需要那个共享资源，则该中优先级的任务反而超过这两个任务而获得CPU时间。如果高优先级等待资源时不是阻塞等待，而是忙循环，则可能永远无法获得资源，因为此时低优先级进程无法与高优先级进程争夺CPU时间，从而无法执行，进而无法释放资源，造成的后果就是高优先级任务无法获得资源而继续推进。

解决方案：

**（1）设置优先级上限**，给临界区一个高优先级，进入临界区的进程都将获得这个高优先级，如果其他试图进入临界区的进程的优先级都低于这个高优先级，那么优先级反转就不会发生。

**（2）优先级继承**，当一个高优先级进程等待一个低优先级进程持有的资源时，低优先级进程将暂时获得高优先级进程的优先级别，在释放共享资源后，低优先级进程回到原来的优先级别。嵌入式系统VxWorks就是采用这种策略。

**（3）第三种方法就是使用中断禁止**，通过禁止中断来保护临界区，采用此种策略的系统只有两种优先级：可抢占优先级和中断禁止优先级。前者为一般进程运行时的优先级，后者为运行于临界区的优先级。火星探路者正是由于在临界区中运行的气象任务被中断发生的通信任务所抢占才导致故障，如果有临界区的禁止中断保护，此一问题也不会发生。

**Linux内存泄漏诊断**

Valgrind工具，调试ARM核内存管理

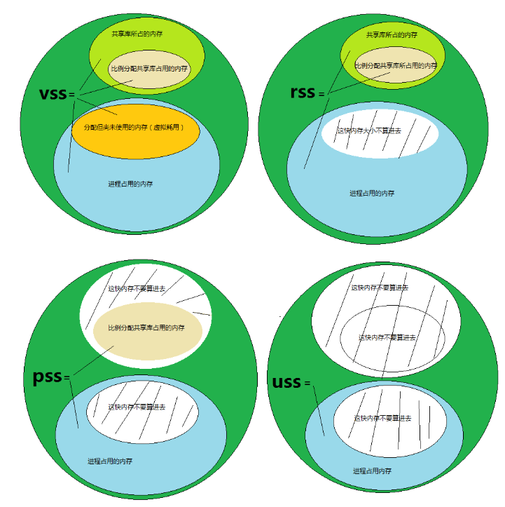
**Linux的VSS,RSS,PSS,USS内存字段？**

**VSS**  （从ps报告为VSZ）是  进程的总可访问地址空间

**RSS**  是 进程RAM中实际保存的总内存。

**PSS** 与RSS的不同之处在于它报告其共享库的比例大小

**USS**  是  进程的总私有内存，即该进程完全独特的内存



**Linux的内存管理（伙伴系统）？**

buddy system

按照2^x次方来分配内存段，减少内存碎片的产生

**Linux的进程调度？**

schedule()切换进程

**Linux在用户态还是内核态开发？**

正常的应用程序在用户态

用户态的应用程序可以通过三种方式来访问内核态的资源：

1）系统调用

2）库函数

3）Shell脚本

**Linux用户态和内核态的通信方式？**

**系统调用（System Call）**：这是最常见的方式，用户态程序通过系统调用接口（如open、read、write、fork等）请求内核执行特定的动作。系统调用是用户态和内核态之间进行交互的桥梁，它允许用户态程序请求内核提供服务。

**中断（Interrupts）：**中断包括软中断和硬中断。当中断到来时，CPU会暂停当前执行的用户态代码，切换到内核态来处理中断。中断机制允许内核在适当的时候介入用户态程序的执行，处理一些紧急或特殊的情况。

**信号（Signal）：**内核通过信号通知用户态进程发生了某些事件，用户态程序可以注册信号处理函数来响应特定的信号事件。例如，SIGTERM和SIGINT等信号就是用于通知进程终止或中断的信号。

**共享内存（Share Memory）：**允许多个进程在它们的地址空间中共享一块内存区域，从而实现用户态和内核态之间的高效通信。这种方式避免了频繁的用户态和内核态切换，但也需要处理数据同步和一致性的问题。

**IOCTL：**这是内核较早的一种用户态和内核态的交互方式。用户态程序通过命令的方式调用ioctl函数，然后内核态分发到对应驱动处理，最后将处理结果返回到用户态。

**Netlink：**本质上是一种特殊的socket，用于内核与多种用户进程之间的消息传递系统。netlink支持内核与用户态之间的双向通信，是一种全双工通信方式。

**procfs/sysfs：**在Linux中，procfs和sysfs是特殊的文件系统，用于内核与用户空间之间的信息交互。procfs提供了内核和进程的各种信息，而sysfs则提供了设备和驱动的信息。用户态程序可以通过读取这些文件系统中的文件来获取内核信息，也可以通过写入特定的文件来配置内核或驱动。

**Linux中断的上下部分？**

Linux和M核不一样，不能进行中断嵌套，要求快进快出快速执行，所以把一个中断服务函数中不耗时的部分放在上半部分执行，有些比较耗时的部分通过一些方式移到服务函数外执行，这样的话中断服务函数就可以做到快进快出

方法：softirqs软中断，tasklet，work queue

**Linux中断为什么不能休眠？**

1、 **中断处理的时候,不应该发生进程切换**，因为在中断context中，唯一能打断当前中断handler的只有更高优先级的中断，它不会被进程打断，如果在 中断context中休眠，则没有办法唤醒它，因为所有的wake\_up\_xxx都是针对某个进程而言的，而在中断context中，没有进程的概念，没 有一个task\_struct（这点对于softirq和tasklet一样），因此真的休眠了，比如调用了会导致block的例程，内核几乎肯定会死。

2、**schedule()在切换进程时**，保存当前的进程上下文（CPU寄存器的值、进程的状态以及堆栈中的内容），以便以后恢复此进程运行。中断发生后，内核会先保存当前被中断的进程上下文（在调用中断处理程序后恢复）；

但在中断处理程序里，CPU寄存器的值肯定已经变化了吧（最重要的程序计数器PC、堆栈SP等），如果此时因为睡眠或阻塞操作调用了schedule()，则保存的进程上下文就不是当前的进程context了.所以不可以在中断处理程序中调用schedule()。

3、2.4内核中schedule()函数本身在进来的时候判断是否处于中断上下文:

if(unlikely(in\_interrupt()))

BUG();

因此，强行调用schedule()的结果就是内核BUG，但我看2.6.18的内核schedule()的实现却没有这句，改掉了。

4、中断handler会使用被中断的进程内核堆栈，但不会对它有任何影响，因为handler使用完后会完全清除它使用的那部分堆栈，恢复被中断前的原貌。

5、处于中断context时候，内核是不可抢占的。因此，如果休眠，则内核一定挂起。

**Linux上下文切换？**

我们知道 Linux 是一个多任务操作系统，它能支持远大于 CPU 数量的任务同时运行。但实际上同一时刻只会有 CPU 数量的进程在运行，等 CPU 时间片到了之后，进程调度器就会把 CPU 资源分配给其他进程。在这个过程中就会涉及到进程之间的切换，这时候就需要将当前进程的上下文信息保存下来，随后加载被调度进程的上下文信息，这就是上下文切换。

这里所说的上下文信息，既包括虚拟内存、栈、全局变量等用户态的资源，也包括内核堆栈、寄存器等内核态的资源。不同类型的上下文切换，会涉及到不同类型资源的切换，例如：同一进程不同线程的切换，只需要切换内核态的资源，而不需要切换用户态的资源。

**Uboot是什么？**

**Uboot的启动流程？**

硬件设备的检测与初始化（串口、内存、b main\_loop进入C语言环境）

环境变量的加载和设置

**Uboot的引导阶段**

自检和预设选择逻辑：进行硬件自检，根据预设的逻辑确定下一步加载哪个引导加载器

加载辅助引导加载器，包括从网络和葛总设备中加载

**操作系统的加载和启动**

从某些存储介质（NAND闪存SD卡网络等）读取操作系统内核，存入内核中然后执行