**FreeRTOS**

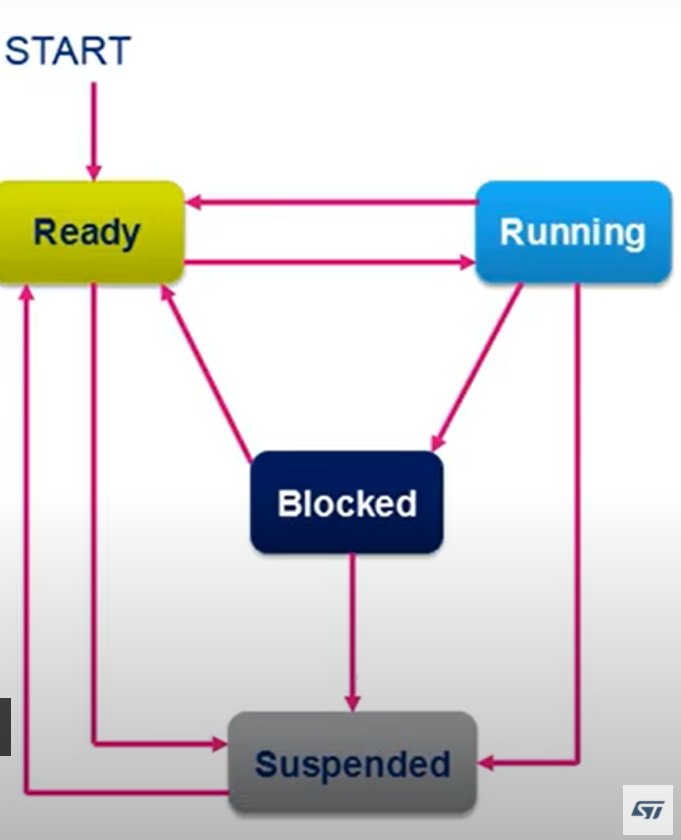
目前实时操作系统在市场上常用的主要是FreeRTOS和u-COS两种。前者免费开源但资料少，适合做产品；后者收费但资料全性能好，适合学习。（这里就直接学FreeRTOS了）

**CMSIS\_OS API**

目前Cortex-M系列芯片上会运行许多不同种类的实时操作系统，因此ARM公司为Cortex-M系列芯片制作了系统接口CMSIS\_OS API，用户通过使用此API接口实现具体所需功能，而无需考虑具体使用的是什么操作系统（FreeRTOS、u-COS、RTX等等）。也就是说CMSIS\_OS API封装好了各种实时操作系统，并向外提供了一套统一的API函数接口，提高了用户代码在不同实时操作系统上的可移植性。（比如ST公司的STM32CubeMX就提供了以FreeRTOS为系统实现了的CMSIS\_OS API，用户可以直接配置并在此基础上编写自己代码）

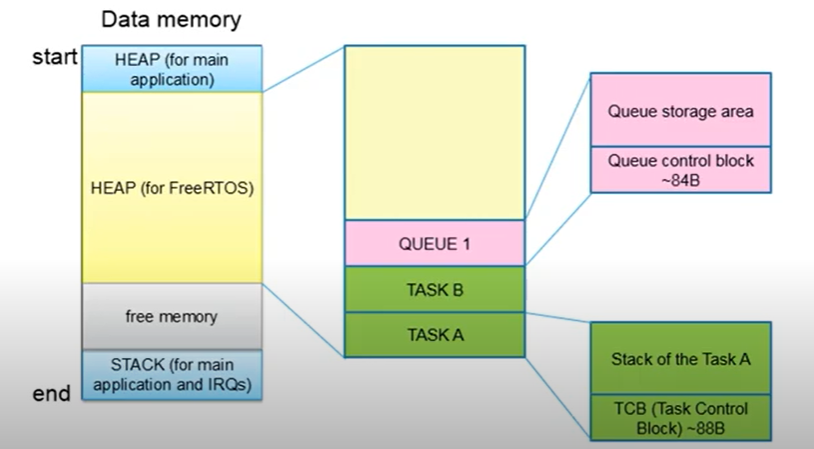
FreeRTOS核心是根据优先级动态调度执行Tasks。（实时性需求高的Task就要有更高的优先级，高优先级的Task就优先执行）

Task是个C语言程序，运行在一个无限循环的程序中，每个Task有自己的栈和（可调节）优先级，有着四种状态：Ready准备状态、Suspended挂起状态、Running运行状态、Blocked阻塞状态。Task可以通过API函数创建和删除。



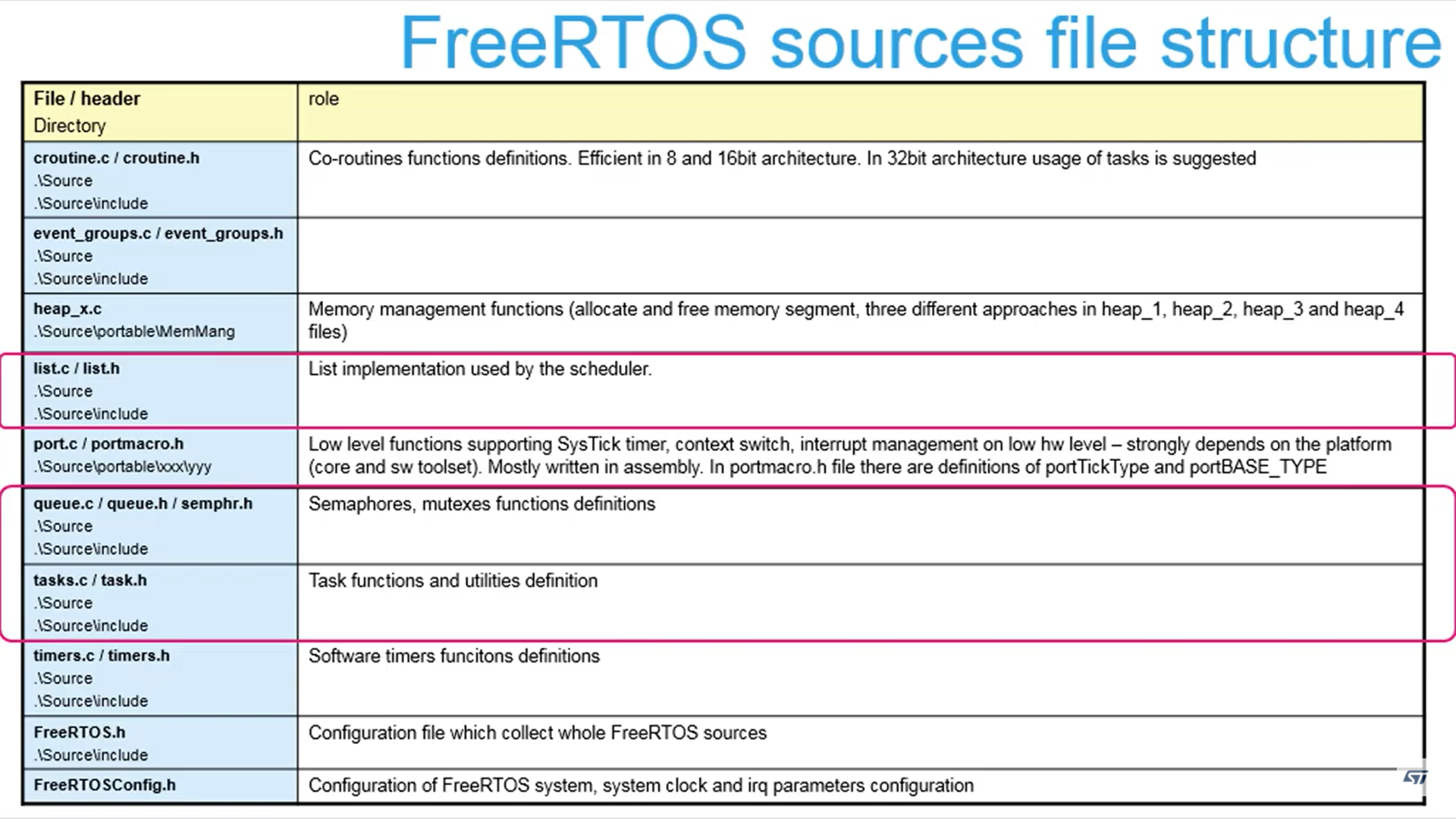
Scheduler是决定哪个Task执行的算法，它会选择一个处于Ready状态的Task执行，有些机制可以让Task得到CPU（时间片，抢占，空闲）。对于调度相同优先级的Tasks，FreeRTOS使用的是round-robin轮询调度算法，抢占式多任务或协作式多任务都可以应用此算法。当有两个优先级相同的Task，并且没在执行的Task的优先级突然变高时，抢占式多任务是优先级变高的Task会立刻抢占优先级低的Task（不建议，因为可能会打乱低优先级Task的时序造成难以解决的不稳定问题），而协作式多任务是优先级变高的Task依旧需要等待正在运行的优先级低的Task结束了才能执行。

OS heap为FreeRTOS分配的内存堆，是在主程序的堆空间分配出来的一段静态大小的内存空间，由FreeRTOS进行管理（OS heap和任务栈都是固定大小的）。和平常程序一样，靠近start地址的是程序的数据区、代码区、堆空间，靠近end地址的是系统中断和程序的栈空间，主要不一样的是FreeRTOS在程序堆空间里申请了一段自己管理的OS heap内存。在FreeRTOS中申请内存，也是在OS heap中提供一段内存（程序堆空间并没有额外分配）。由于OS heap中申请的堆和任务栈有着很强的动态性，为了防止溢出，会将OS heap和任务栈都设置的大一些，正常运行后通过API函数查看剩余空间，最终得到应该分配的空间。建议：当需要保存较大的数据量并且多个任务会使用到这些数据时，这会使得每个任务栈需要的空间都很大，芯片内存不够分配，这种情况可以选择申请一段内存作为公共栈，将数据保存在公共栈中供所有任务访问。

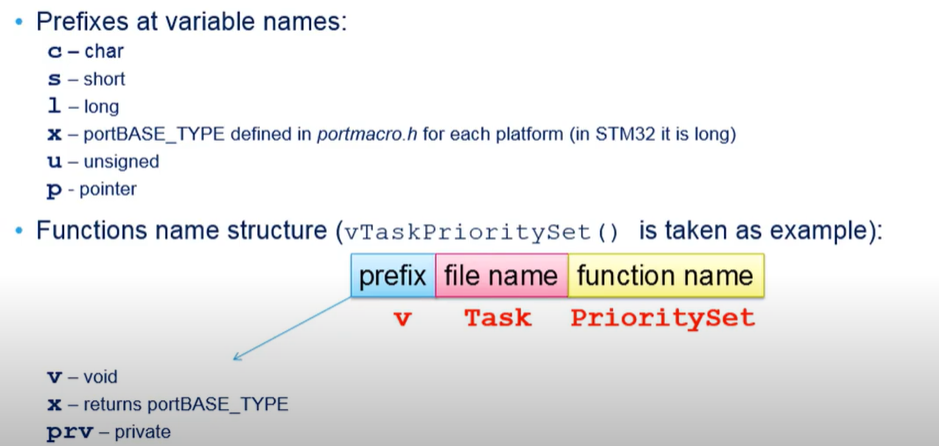


System timer是一个系统节拍定时器，会定时触发SVC中断对应的SWI处理函数，用于切换不同的Task上下文。这个定时器需要用户选择自己芯片合适的定时器并实现，SWI处理函数则由官方实现。

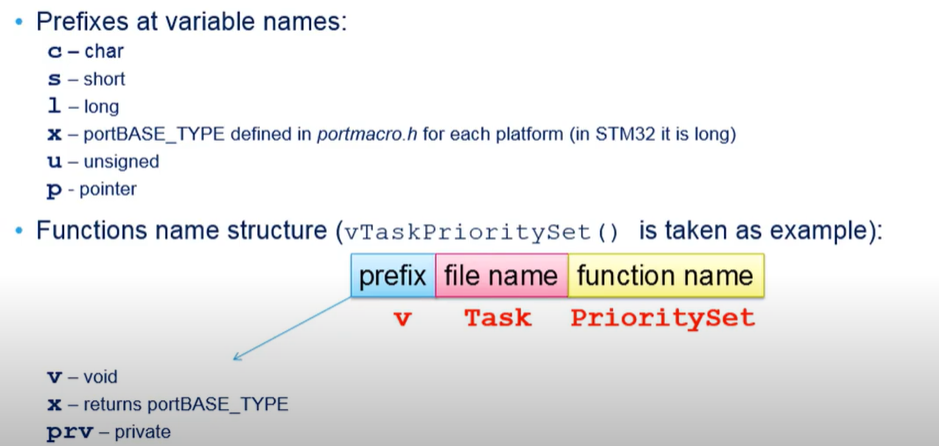
FreeRTOS里的源码目录及其功能如下：



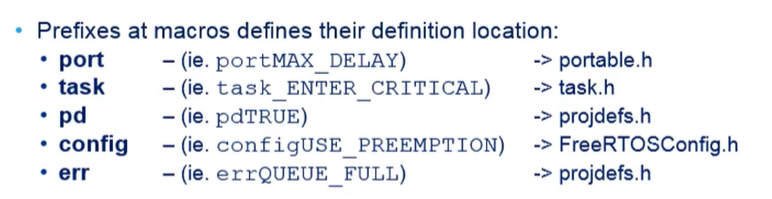
FreeRTOS的API操作函数名的定义由三个部分组成：前缀、文件名、函数功能名。前缀表示了API函数的返回值类型，文件名表示了定义API函数的文件、函数功能名表示了API函数的具体功能。



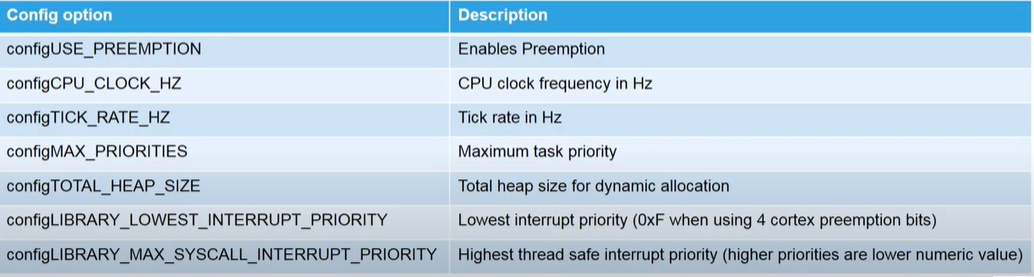
同时，FreeRTOS的变量定义也使用了前缀，用来表明此变量的类型：



而FreeRTOS的宏定义的前缀表示的是此宏定义所在的文件（默认pdTRUE和pdPASS为1；pdFALSE和pdFAIL为0）：



FreeRTOSConfig.h中有着大量可修改的配置信息，以下只是部分例子（注意：值越大，任务优先级就越高，而中断优先级就越低）：



**FreeRTOS移植I.MX6ULL案例**

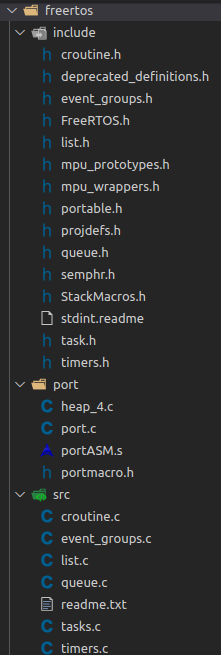
FreeRTOS提供了完整的源码和许多demo来方便用户在芯片上移植此系统，目前手头上有个正好有个NXP的I.MX6ULL芯片，因此尝试将FreeRTOS移植到芯片中。（此时，我只对FreeRTOS有个基本了解，而且源码没有针对Cortex-A7的demo，所以移植过程遇到了很多困难）

**1、创建工程**

首先创建一个I.MX6ULL的工程目录，工程目录下有6个文件夹：driver/（保存裸机驱动的头文件和源文件）、freertos/（保存FreeRTOS系统源码）、imx6u/（保存NXP提供的I.MX6ULL芯片信息的头文件）、obj/（保存编译输出的文件）、project/（保存main.c、main.h和start.s）、std/（保存标准库文件）。工程目录下有还有两个文件：Makefile和imx6u.lds链接脚本。

这里就讲一下freertos/目录和std/目录，其他文件和目录就不说了，因为普通的裸机驱动实验也是会用到那些的。

打开下载的FreeRTOS源码，这里用的是9.0.0版本的，将FreeRTOS\Source下所有文件拷贝到工程目录下freertos/src下，将FreeRTOS\Source\include下所有文件拷贝到工程目录下freertos/include下，在FreeRTOS\Source\portable下挑选适合自己的平台，这里是GCC/ARM\_CA9（因为没有Cortex-A7，所以选择接近的Cortex-A9），将里面的文件拷贝到工程目录下freertos/port下，同时FreeRTOS\Source\portable\MemMang下是系统内存管理代码，选一个合适的拷贝到工程目录下freertos/port下，这里选择heap\_4.c。这样，freertos/目录就搭建好了。（std/目录在第3步编译时才会用到，现在先空着）



**2、建立连接**

FreeRTOS源码到手了，需要和自己的代码连接在一起才会运行系统，这里参考的是demo里的CORTEX\_A9\_Zynq\_ZC702这个例程，原因也是Cortex-A9接近的Cortex-A7，将参考例程的配置头文件FreeRTOSConfig.h拷贝到工程目录下project/下。

**2.1、修改project/FreeRTOSConfig.h**

configCPU\_CLOCK\_HZ设置为自己芯片CPU主频（这里是528000000UL）。

configTICK\_RATE\_HZ设置需要的节拍频率（这里是1000）。

configPERIPHERAL\_CLOCK\_HZ设置芯片外部总线时钟频率（这里是396000000UL）。

configUSE\_TASK\_FPU\_SUPPORT设置FPU是否支持（这里设置为1不支持）。

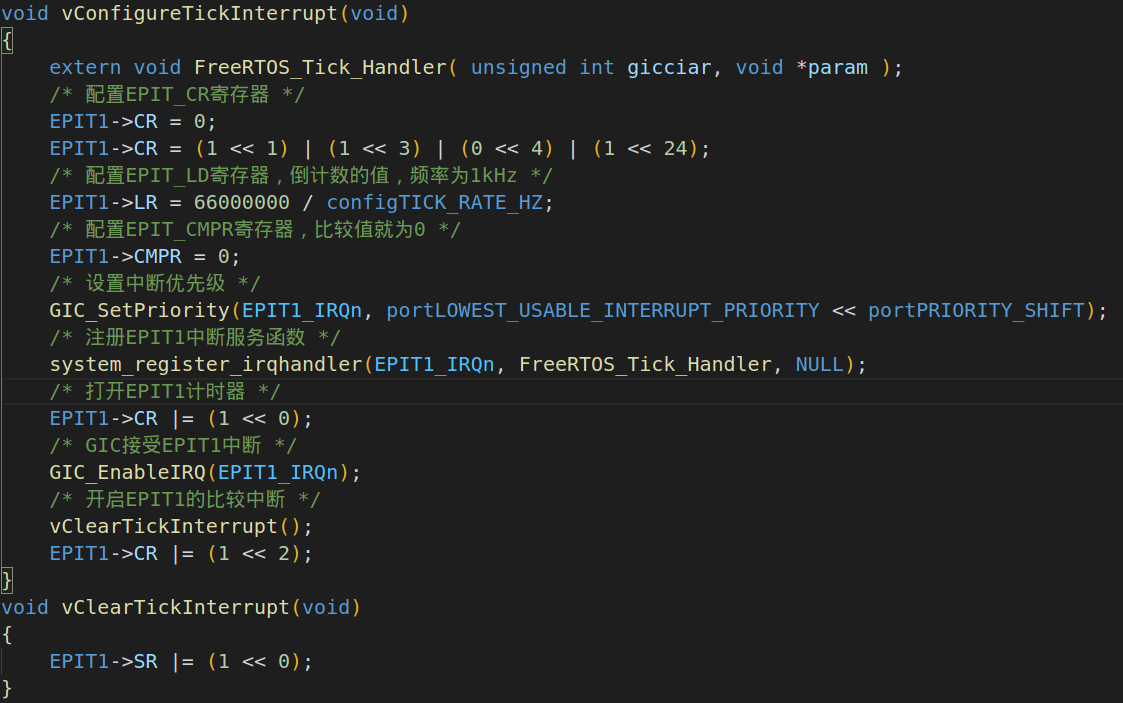
configINTERRUPT\_CONTROLLER\_BASE\_ADDRESS设置中断控制器基地址（这里I.MX6ULL的GIC基地址为0x00a00000）。

configINTERRUPT\_CONTROLLER\_CPU\_INTERFACE\_OFFSET设置中断CPU接口地址偏移（这里I.MX6ULL的GICC\_CTLR寄存器地址为0x00a02000，所以偏移值为0x2000）。

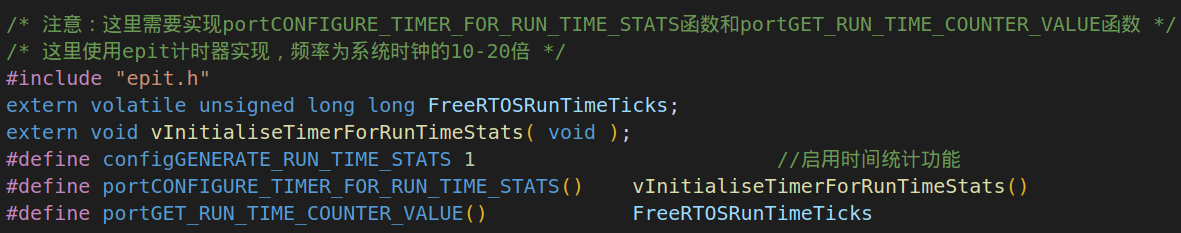
configUNIQUE\_INTERRUPT\_PRIORITIES设置中断优先级个数（这里使用的是32个不同的中断优先级）。



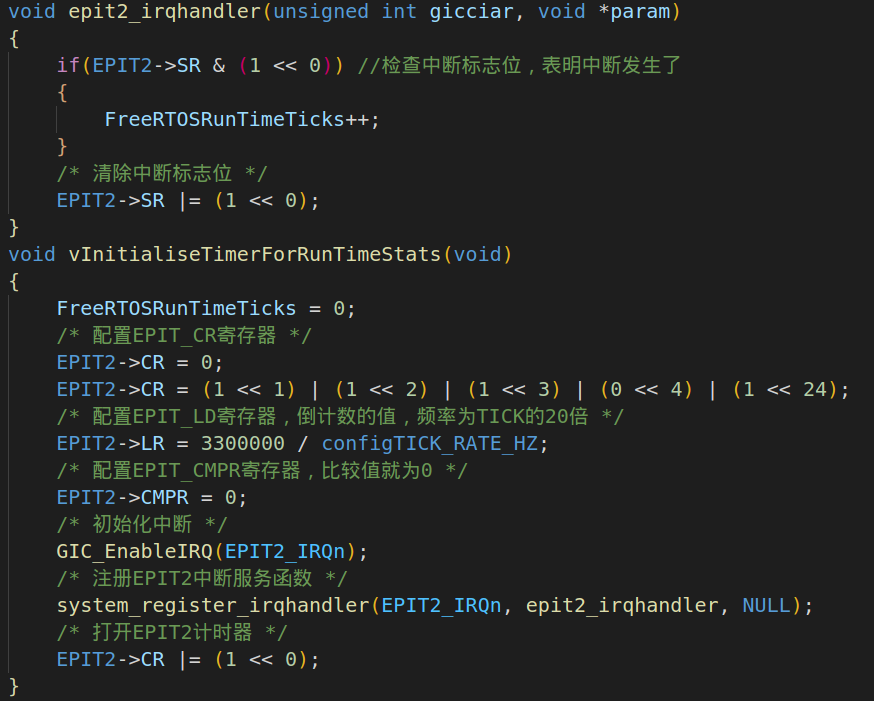
实现configSETUP\_TICK\_INTERRUPT()函数，这里是定义了一个vConfigureTickInterrupt()函数，配置EPIT1定时器并使能中断。注意，FreeRTOS源码中的FreeRTOS\_Tick\_Handler函数是没有参数的，但由于我这里要求注册的中断服务函数必须要有这两个参数，所以把port.c和portmacro.h中这个函数的定义和声明加了两个参数，但并不会使用到它们。EPIT1的输入时钟频率为66MHz，定时中断频率为系统节拍频率。



实现configCLEAR\_TICK\_INTERRUPT()函数，这里是定义一个vClearTickInterrupt()函数，使用EPIT1定时器中断实现，就是简单的清除EPIT1的定时中断。

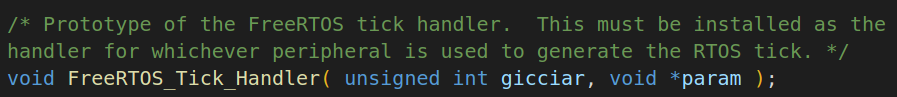


如果启用时间统计功能，要实现portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS()和portGET\_RUN\_TIME\_COUNTER\_VALUE()函数，这里是定义了一个vInitialiseTimerForRunTimeStats()函数，配置EPIT2定时器并使能中断。定义一个volatile unsigned long long FreeRTOSRunTimeTicks变量记录发生EPIT2定时中断的次数。时间统计的精度要比较高，所以设置的为系统节拍频率的20倍。

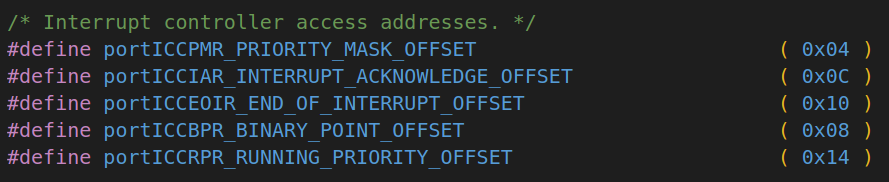


**2.2、修改freertos/port/portmacro.h**

给中断处理函数的声明加上两个参数，原因上面已经讲了：

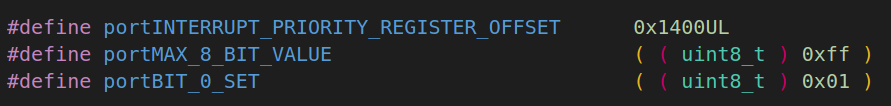


检查这里的寄存器基于中断寄存器GIC基地址的偏移设置是否正确，这里原版的配置是符合I.MX6ULL的：

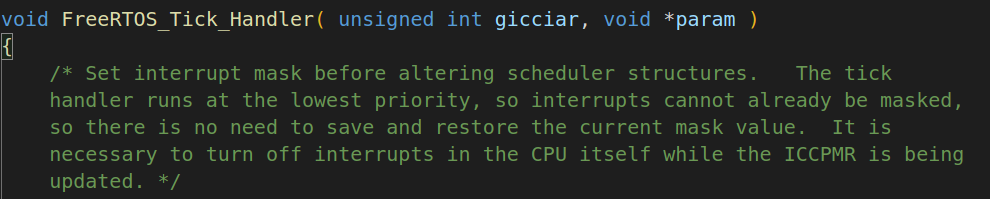


**2.3、修改freertos/port/port.c**

portINTERRUPT\_PRIORITY\_REGISTER\_OFFSET是中断优先级寄存器相对于中断控制器GIC基地址的偏移量，I.MX6ULL是0x1400。（这里找这个bug找了好久，一直卡在优先级个数的自检环节过不去）



给中断处理函数定义加上两个参数，原因上面已经讲了：

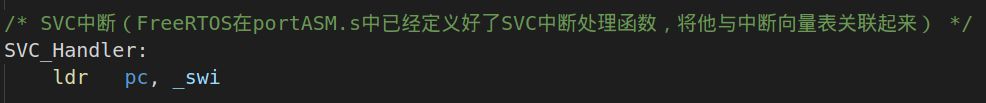


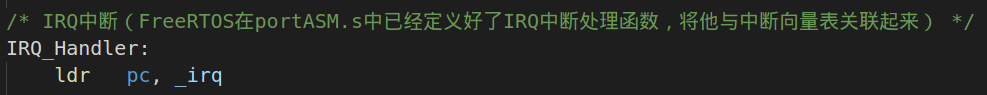
**2.4、修改freertos/port/portASM.s**

与其说是修改，不如说是将portASM.s和start.s关联，portASM.s中不需要任何修改，它已经写好了IRQ中断FreeRTOS\_IRQ\_Handler和SVC中断FreeRTOS\_SWI\_Handler，只需要在start.s的中断向量表中跳转到这两个对应中断处理即可。在start.s中声明这两个外来变量：

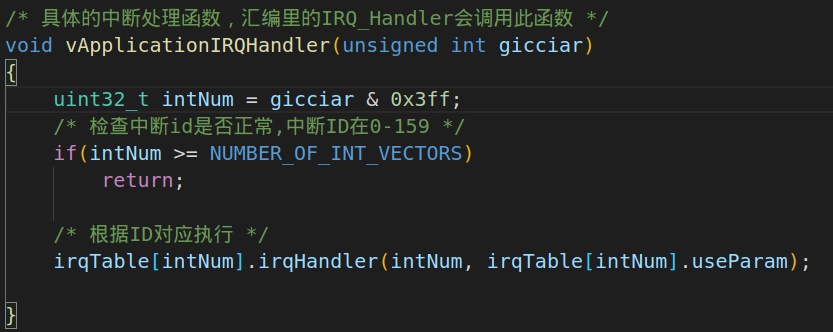


在对应的标签下，发生对应跳转：

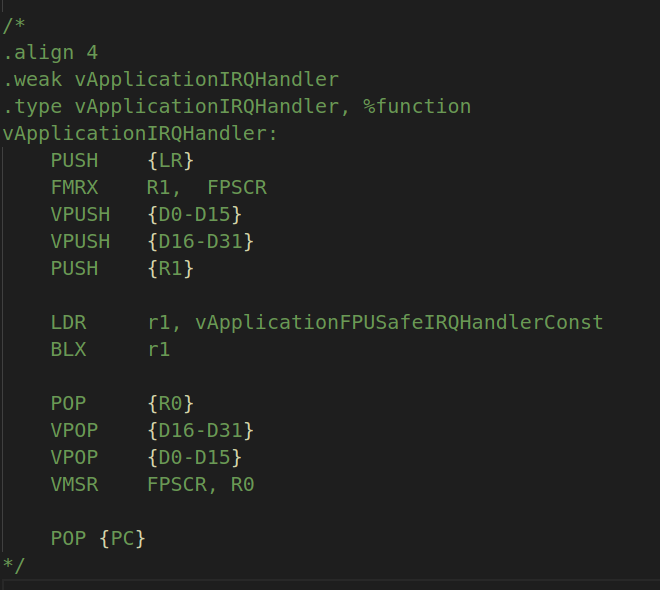




FreeRTOS\_SWI\_Handler完全由FreeRTOS系统实现了，而FreeRTOS\_IRQ\_Handler只实现了现场保存，中断嵌套管理和获取中断号，而需要用户实现vApplicationIRQHandler这个C语言中断处理函数，其主要功能是根据中断号执行不同的具体中断处理函数。



注意，如果是开启了FPU支持，FreeRTOS\_IRQ\_Handler还需要保存浮点寄存器的现场，用户需要实现的C语言中断处理函数变为vApplicationFPUSafeIRQHandler，此函数定义在port.c中供用户修改。但portASM.s里面的浮点寄存器指令，如VPUSH、VPOP，GCC编译表示不识别所以直接注释掉了。

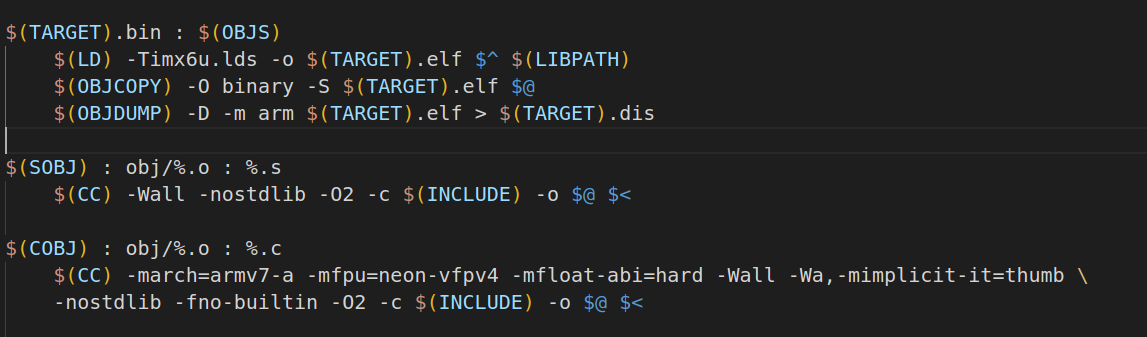


**3、工程编译测试**

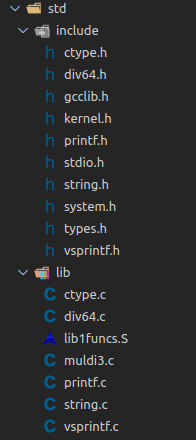
mian.c中配置时钟频率并开启硬件浮点，需要和FreeRTOSConfig.h设置的一样，初始化外设前需要调用portDISABLE\_INTERRUPTS()关闭中断。最后创建任务，开启调度器。调度器开启时，会自行开启中断。

编写完成后，进行编译：

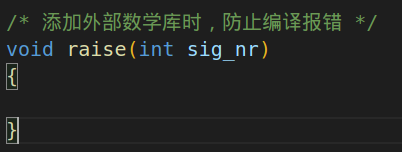




此时编译会有巨大问题，问题就是FreeRTOS中使用了标准C库函数，如strlen、strcpy、memcpy、sprintf等，编译时这些函数是没有定义的。尝试通过链接外部库来完成，但一直无法成功（bug调了很久，各种错）。最后在uboot源码中找到了标准C库的源码文件，直接拷贝到std\目录一起进行编译。需要的源码文件：



编译又提示putc、getc、puts、raise函数未定义，这里直接自己定义这四个函数，因为FreeRTOS并不会用到它们。将putc、getc、puts分别定义为串口写字节、串口读字节、串口写字符串函数，也就是把串口作为了标准输入输出流。raise根据网上建议，直接定义为空函数。



再次编译，编译通过。将得到的bin文件添加头部信息后写到SD卡中，开发板插入SD卡，上电，FreeRTOS系统终于在I.MX6ULL上正确运行了！！