《CORTEX-M3+UCOS-II 嵌入式系统开发入门与应用》

作者：华清远见

第 6 章 μC/OS-Ⅱ操作系统基础及其移植开发初步

本章目标

μC/OS-II内核作为一种代码公开的嵌入式实时操作系统的内核非常有特色，在规模不大的代码内实现了抢占式任务调度和多任务间通信等功能，任务调度算法也很有特点。该内核裁剪到最小状态后编译出来只有 8KB 左右，全部内核功能（添加 LWIP 网络协议栈等）也就 100KB 左右，资源消耗非常小。市面上一些 ARM 微处理器片上所带的内存就足够一个裁剪合适的内核的简单应用，非常方便产品的开发设计。

当前，μC/OS-Ⅱ是一个基本完整的嵌入式操作系统解决方案套件，包括μC/TCP-IP（IP 网络协议栈）、μC/FS（文件系统）、μC/GUI（图形界面）、μC/USB（USB 驱动）和μC/FL（Flash 加载器）等部件，但是这些部件是不公开代码的。

还有一些在嵌入式环境中发挥重要作用的部件包括嵌入式数据库、POSIX 兼容性接口、常用设备的驱动模块等，将来还会产生更多的重要部件需求。在互联网上的开源社区通常能够找到相应的开源代码包，并且可以进行移植。

6.1 实时操作系统基本原理与技术

本节将主要讲述实时操作系统的基本原理和技术。通过对本章的学习，读者可以了解 RTOS（Real Time operation System，实时操作系统）的基本特征、结构体系、重要指标、性能参数等重要理论，为全面掌握 RTOS 打下基础。

6.1.1 实时操作系统基本特征

实时操作系统与其相对的是分时操作系统，UNIX 就是典型的分时操作系统。当分时操作系统允许对中断处理的优先级做调整，使系统对外部事件的响应速度保证不大于某一特定时间间隔时，就构成了实时操作系统。根据 IEEE 对实时操作系统的定义，实时操作系统的基本特征应表现为以下几个方面。

- 实时性：响应外部事件的时间必须在限定的时间范围内，在某些情况下还需要是确定的、可重复实现的，不管当时系统内部状态如何，都必须是可预测的。
- 抢占式调度：为确保响应时间，实时操作系统必须允许高优先级的任务一旦进入就绪状态，就可以马上抢占正在运行的低优先级任务的执行权。
- 具有异步响应能力：异步事件是指无一定时序关系、随机发生的事件。如实时控制设备出现异常等突发事件，都属于随机事件。实际环境中，嵌入式实时系统需要处理多个外部事件，这些事件往往同时出现，而且发生的时刻也是随机的。实时操作系统应有能力对这类同时发生的外部事件进行有效的处理。
- 内存锁定：必须具有将程序部分代码锁定在内存的能力。将频繁访问的数据锁定在内存，减少了为获得该数据而访问磁盘的时间，从而保证了快速的响应时间。
- 具有优先级调度机制：实时操作系统必须允许用户定义中断和任务的优先级，并具有相应的优先级调度机制。
- 同步/互斥机制：提供对共享数据的同步和互斥手段。

实时操作系统能对外部事件和信号在限定的时间范围内作出响应，它所强调的是实时性、可靠性和灵活性。实时操作系统一般与实时应用软件相结合成为有机整体：用实时操作系统来管理和调度实时应用软件的各项任务，为应用软件提供良好的运行和开发环境。一般来说，实时操作系统以库的形式提供系统调用来实现对上层实时应用程序的支持；而应用程序通过链接实时操作系统的库来实现实时任务调度。

6.1.2 实时操作系统的关键技术指标

一般来说，评价一个实时操作系统的优劣可以用以下几个技术指标来衡量。

- 任务调度算法：一个实时操作系统的任务调度算法，在很大程度上决定了其系统实时性和其多任务调度能力。常用的任务调度算法有优先级调度策略和时间片轮转调度策略；调度的方式可分为可抢占式、不可抢占式和选择可抢占式等；常用的调度算法有 Rate Monotonic（发生率单调）、优先级与发生率成正比（LiuLay 1973）、Lottery Scheduler（彩票调度，Wald&Weih194）等。
- 上下文切换时间（Context Switching Time）：指当处理器的控制权由运行任务转移到另一个就绪任务时所耗费的系统时间。RTOS 的上下文切换时间可以由以下公式算出：上下文切换时间 = 系统保持当前任务

的状态所需时间 + 从就绪任务表中查找最高优先级任务的时间+将优先级最高的就绪任务转到运行态所需要的时间。保护和恢复上下文的方法很大程度上依赖于处理器的架构，所以衡量一个实时操作系统是否适合在某种体系结构的处理器上运行，上下文切换时间是一个重要的衡量指标。

- 系统确定性（Determinism）：在实时操作系统中，在一定的条件下，系统调用运行的时间是可以预测的，但这并不意味着无论系统的负载如何，所有的系统调用都总是执行一个固定长度的时间，而是指系统调用的最大执行时间可以确定。
- 最小内存开销：在某些领域，如工业控制领域，基于降低成本的考虑，其内存的配置一般都不大。因此在实时操作系统的设计中，其占用内存大小是一个很重要的衡量指标。同时，这也是 RTOS 设计与其他操作系统设计的明显区别之一。
- 最大中断禁止时间：当操作系统在执行某些系统调用时，是需要关闭中断响应的，即中断被屏蔽，只有当操作系统重新回到用户态时才重新响应外部中断请求，这一过程所需的最大时间就是最大中断禁止时间。由此可以看出操作系统的最大中断禁止时间越长，系统丢中断的可能性越大，所以最大中断禁止时间成为衡量一个操作系统实时性的重要指标。

上述几项中，上下文切换时间和最大中断禁止时间是评价一个实时操作系统实时性最重要的两个技术指标。

6.1.3 实时操作系统基本术语

本节将主要介绍在实时操作系统领域中常用的专业术语，以便读者更好地掌握本书后面的内容。

- 硬实时（Hard Real-Time）：通常将具有优先级驱动的、时间确定性的、可抢占调度的系统称为硬实时系统。所谓“硬实时”，主要强调对实时性和确定性的要求较高。
- 优先级驱动（Priority-Driven）：在一个多任务系统中，正在运行的任务总是优先级最高的任务。在任何给定的时间内，总是把处理器分配给优先级最高的任务。
- 优先级反转（Priority Inversion）：当一个任务等待比它优先级低的任务释放资源而被阻塞时，这种现象被称为优先级反转。优先级继承技术可以解决优先级反转问题。目前市场上大多数商用操作系统都使用了优先级继承技术。
- 优先级继承（Priority-Inheritance）：优先级继承是用来解决优先级反转问题的技术。当优先级反转发生时，较低优先级任务的优先级暂时提高，以匹配较高优先级任务的优先级。这样，就可以使较低优先级任务尽快地执行并且释放较高优先级任务所需要的资源。
- 实时执行体（Realtime Executive）：实时执行程序包括一套支持实时系统所必需的机制，如多任务、CPU 调度、通信和存储分配等。在嵌入式应用中，这一套机制被称为实时操作系统或实时执行体或实时内核。
- 重调度过程（Rescheduling Procedure）：重调度过程是判断任务优先级和执行状态的过程。
- 任务（Task）：实时操作系统中的任务相当于一般操作系统的进程（Process），一个任务就是操作系统的一个可以运行的历程。
- 任务上下文（Task context）：任务上下文指一个未运行的任务状态，如堆栈指针、计数器、内存字段和通用寄存器等。
- 调度延时（Scheduling Latency）：调度延时是指当一个事件从引起更高优先级的任务就绪到这个任务开始运行之间的时间。简而言之，就是一个任务被触发后，由就绪到开始运行的时间。
- 可伸缩的体系结构（Scalable Architecture）：可伸缩的体系结构指一个软系统能够支持多种应用而无需在接口上做很大的变动。这种结构往往提供可选用的系统组件，供开发者“量体裁衣”。
- 中断延时（Interrupt Latency）：中断延时指从中断发生到开始执行中断处理程序的这一段时间。
- 互斥（Mutual Exclusion）：互斥是用于控制多任务对共享数据进行顺序访问的同步机制。在多任务应用中，当两个或更多的任务同时访问同一数据区时，就会造成访问冲突。互斥能使它们依次访问共享数据而不引起冲突。
- 抢占（Preemptive）：抢占是指当系统处于核心态的内核运行时，允许任务重新调度。也就是说，一个正在执行的任务可以被打断而让另一个任务运行，这提高了应用对外部中断的响应性。许多实时操作系统都是以抢占方式运行。但这并不是说调度在任何时候都是可以发生的。例如，当实时操作系统的一个任务

正在通过系统调用访问共享数据时，重新调度和中断都是不允许的。

6.2 μC/OS-Ⅱ应用程序开发

μC/OS-Ⅱ是一个专门为嵌入式设备设计的硬实时操作系统内核，它自 1992 年发布以来，在世界各地获得了广泛的应用。鉴于μC/OS-Ⅱ可免费获得代码，对于嵌入式开发而言，μC/OS-Ⅱ无疑是最经济的选择。应用μC/OS-Ⅱ目的是要在其之上开发应用程序。下面简单介绍基于μC/OS-Ⅱ应用程序的基本结构以及与应用程序开发相关的知识，本书后续章节会对应用程序开发做更详细的讲解。

6.2.1 μC/OS-Ⅱ的变量类型

由于 C 语言变量类型的长度与编译器类型相关，为了便于在各个平台间移植，在μC/OS-Ⅱ中没有使用标准 C 语言的数据类型，而是定义了自己的数据类型。具体的变量类型如表 6.1 所示。这些变量的定义，可参见μC/OS-Ⅱ源码的 OS_CPU.H 文件。这种方式的类型定义，很大程度上方便了系统在不同编译器间的移植。

表 6.1　　μC/OS-Ⅱ使用的变量类型

类 型 符 合	类　　型	宽　　度
BOOLEAN	布尔型	8
INT8U	8 位无符号整形	8

续表

类 型 符 合	类　　型	宽　　度
INT8S	8 位有符号整形	8
INT16U	16 位无符号整形	16
INT16S	16 位有符号整形	16
INT32U	32 位无符号整形	32
INT32S	32 位有符号整形	32
FP32	单精度浮点数	32
FP64	双精度浮点数	64

6.2.2 应用程序的基本结构

每个μC/OS-Ⅱ应用至少要求有一个任务。每个任务必须被写成无限循环的形式。下面代码是μC/OS-Ⅱ推荐的结构：

```
Void task(void * pdata)
{
  INT8U err;
  InitTimer();
  While(1)
   {
      …                          //应用程序代码
      OSTimeDly(1) ;             //可选

  }
}
```

系统运行时，μC/OS-Ⅱ会为每一个任务保留一个堆栈空间。系统在任务切换时要恢复上下文并执行一条返回指令，如果允许任务执行完并返回，那么很可能会破坏系统的堆栈空间，从而给应用程序的执行带来不确定性。换句话说，程序“跑飞”了。所以，每一个任务必须被写成无限循环的形式。但任务的无限循环，并不意味着任务永远占有 CPU 的使用权，任务可以通过 ISR 或调用操作系统 API（如任务挂起 API），使任务放弃对 CPU 的使用权。

在上面的任务结构示例代码中，值得一提的是 InitTimer()函数。这个函数应由系统提供，开发者需要在优先级最高的任务内调用它，且不能在 for 循环内调用，而且该函数和所使用的 CPU 相关，每种系统都有自己的 Timer 初始化程序。

在μC/OS-Ⅱ的帮助手册中，作者强调绝不能在 OSInit()或 OSStart()内调用 Timer 初始化函数 InitTimer()，那样会破坏系统的可移植性，同时也会带来性能上的损失。所以，一个折中的办法就是如上所述，在优先级最高的任务内调用，这样可保证当 OSStart()调用系统内部函数 OSStartHighRdy()开始多任务后，首先执行的是 Timer 初始化程序；或专门执行一个优先级最高的任务，只做一件事情，那就是执行 Timer 初始化，之后通过调用 OSTaskSuppend()将自己挂起，永远不再执行，不过这样会浪费一个 TCB 空间。对于那些 RAM 内存空间有限的系统来说，应该尽量不用。

μC/OS-Ⅱ是多任务内核，函数可能会被多个任务调用，因此还需考虑函数的可重入性。由于每个任务有各自的堆栈，而任务的局部变量是放在当前的任务堆栈中的，所以要保证函数代码的可重入性，只要不使用全局变量即可。

利用μC/OS-Ⅱ的消息队列可实现消息驱动程序。在编写任务代码时，先完成任务初始化，然后在消息循环过程中在某个消息上等待，当其他任务或者中断服务程序返回消息后，根据消息的内容调用相应的函数模块，函数调用后，重新回到消息循环，继续等待消息。

6.2.3 μC/OS-Ⅱ API 介绍

任何一个操作系统都会提供大量的 API 供开发者使用，μC/OS-Ⅱ亦如此。由于μC/OS-Ⅱ面向的是实时嵌入式系统开发，并不要求大而全，所以内核提供的 API 也就大多与多任务相关。

下面介绍几个比较重要的 API 函数。

1．OSTaskCreate()函数

该函数应至少在 main()函数内调用一次，在 OSInit()函数调用之后调用，它的作用就是创建一个任务。该函数有 4 个参数，分别是任务的入口函数、任务的参数、任务堆栈的首地址和任务的优先级。调用该函数，系统会首先从 TCB 空闲队列内申请一个空的 TCB 指针；然后根据用户给出的参数初始化任务堆栈，并在内部的任务就绪表内标记该任务为就绪状态；最后返回。这样一个任务就创建成功了。

2．OSTaskSuspend()函数

该函数可将指定的任务挂起。如果挂起的是当前任务，那么还会引发系统执行任务切换先导函数 OSShed()来进行一次任务切换。这个函数只是一个指定任务优先级的参数。事实上在系统内部，优先级除了表示一个任务执行的先后次序外，还起着区分每一个任务的作用。换句话说，优先级也就是任务的 ID，所以μC/OS-Ⅱ不允许出现相同优先级的任务。

3．OSTaskResume()函数

该函数和 OSTaskSuspend()函数正好相反，它用于将指定的已经挂起的函数恢复为就绪状态。如果恢复任务的优先级高于当前任务，那么还将引发一次任务切换。其参数类似于 OSTaskSuspend()函数，用来指定任务的优先级。需要特别说明的是，该函数并不要求和 OSTaskSuspend()函数成对出现。

4．OS_ENTER_CRITICAL()宏

由 OS_CPU.H 文件可知，OS_ENTER_CRITICAL()和下面要谈到的 OS_EXIT_CRITICAL()都是宏，它们都与特定的 CPU 相关，一般都被替换为一条或者几条嵌入式汇编代码。由于系统希望向上层开发者隐藏内部实现，故一般都宣称执行此条指令后系统进入临界区。其实，该指令只是进行了关中断操作而已。这样，只要任务不主动放弃 CPU 使用权，别的任务就没有占用 CPU 的机会了，相对这个任务而言，它就是独占了，所以说进入临界区了。这个宏应尽量少用，因为它会破坏系统的一些服务，尤其是时间服务，并使系统对外界响应的性能降低。

5．OS_EXIT_CRITICAL()宏

该宏与上面 OS_ENTER_CRITICAL()宏配套使用，在退出临界区时使用。其实它就是重新开中断。需要注意的是，它必须和 OS_ENTER_CRITICAL()宏成对出现，否则会带来意想不到的后果，最坏情况下，系统会崩溃。

6．OSTimeDly()函数

该函数实现的功能是先挂起当前任务，然后进行任务切换，在指定的时间到了之后，将当前任务恢复为就绪状态，但并不一定运行；如果恢复后是优先级最高的就绪任务，那么运行之。简而言之，就是可使任务延时一定时间后再次执行它；或者说，暂时放弃 CPU 的使用权。一个任务可以不显示地调用这些可导致放弃 CPU 使用权的 API，但那样多任务性能会大大降低，因为此时仅仅依靠时钟机制在进行任务切换。一个好的任务应在完成一些操作后主动放弃 CPU 的使用权。

6.2.4 μC/OS-Ⅱ多任务实现机制

μC/OS-Ⅱ是一种基于优先级的可剥夺型多任务内核，了解它的多任务机制原理，有助于写出更加强壮的代码。其实在单 CPU 情况下，是不存在真正多任务机制的，存在的只是不同的任务轮流使用 CPU，所以本质上还是单任务。但由于 CPU 执行速度非常快，加上任务切换十分频繁，所以感觉好像有很多任务同时在运行，这就是所谓的多任务机制。

由上述内容不难发现，要实现多任务机制，目标 CPU 必须具有一种在运行期间更改 PC 的途径，否则无法做到切换。遗憾的是，目前还没有哪个 CPU 支持直接设置 PC 指针的汇编指令。但一般 CPU 都允许通过类似 JMP 和 CALL 这样的指令来间接修改 PC，主要是软中断。但在一些 CPU 上，并不存在软中断这个概念，所以在那些 CPU 上，需要使用 PUSH 指令加上一条 CALL 指令来模拟一次软中断发生。

μC/OS-Ⅱ中，每个任务都有一个任务控制块，这是一个复杂的数据结构。在任务控制块偏移为 0 的地方，存储着一个指针，记录了所属任务的专用堆栈地址。事实上，在μC/OS-Ⅱ中，每个任务都有自己的专用堆栈，彼此之间不能侵犯，这点要求开发者在他们的程序中保证。一般的做法是，把它们声明成静态数组而且声明成 OS_STK 类型。当任务有了自己的堆栈时，就可将每一个任务堆栈记录到前面提到的任务控制块偏移为 0 的地方，以后每当发生任务切换时，系统必然会先进入一个中断，这一般是通过软中断或者时钟中断实现的。然后会把当前任务的堆栈地址保存起来，接着恢复要切换的任务的堆栈地址。由于那个任务

的堆栈里也一定存的是地址（每当发生任务切换时，系统必然会进入一个中断，而一旦中断，CPU 就会把地址压入栈中），这样就达到了修改 PC 为下一个任务的地址的目的。开发者可利用μC/OS-Ⅱ的多任务实现机制，写出更健壮、更有效率的代码来。

6.3 μC/OS-Ⅱ在 STM32F103 处理器上的移植

6.3.1 移植条件

移植μC/OS-Ⅱ到处理器上必须满足以下条件。

（1）处理器的 C 编译器能产生可重入代码。

μC/OS-Ⅱ是多任务内核，函数可能会被多个任务调用，代码的重入性是保证完成多任务的基础。可重入代码指的是可被多个任务同时调用，而不会破坏数据的一段代码，或者说代码具有在执行过程中打断后再次被调用的能力。

下面列举了两个函数例子，它们的区别在于变量 temp 保存的位置不同。swap1 函数中 temp 作为全局变量存在，swap2 函数中 temp 作为函数的局部变量存在，因此 swap1 函数是不可重入的，而 swap2 函数是可重入的。

swap1 函数代码如下：

```
int temp;
void swap1(int * x, int * y)
{
   temp = *x;
   *x= *y;
   *y= temp;
}
```

swap2 函数代码如下：

```
void swap(int * x, int * y)
{
   int temp;
   temp= *x;
   *x=*y;
   *y= *temp;
}
```

此外，除了在 C 程序中使用局部变量外，还需要 C 编译器的支持。使用 MDK RealView 开发集成环境，可生成可重入的代码。

（2）用 C 语言可打开和关闭中断。

ARM 处理器核包含一个 CPSR 寄存器，该寄存器包括一个全局的中断禁止位，控制它便可打开和关闭中断。

（3）处理器支持中断并且能产生定时中断。

μC/OS-Ⅱ通过处理器产生的定时器中断来实现多任务之间的调度。ARM7TDMI 的处理器都支持中断并能产生定时器中断。

（4）处理器支持能够容纳一定量数据的硬件堆栈。

对于一些只有 10 根地址线的 8 位控制器，芯片最多可访问 1KB 存储单元，在这样的条件下，移植是比较困难的。

（5）处理器有将堆栈指针和其他 CPU 寄存器读出和存储到堆栈（或内存）的指令。

μC/OS-Ⅱ进行任务调度时，会把当前任务的 CPU 寄存器存放到此任务的堆栈中，然后，再从另一个任务的堆栈中恢复原来的工作寄存器，继续运行另一个任务。所以，寄存器的入栈和出栈是μC/OS-Ⅱ多任务调

度的基础。

6.3.2 移植步骤

所谓移植，就是使一个实时操作系统能够在某个微处理器平台上或微控制器上运行。由μC/OS-Ⅱ的文件系统可知，在移植过程中，用户所需要关注的就是与处理器相关的代码。这部分包括一个头文件 OS_CPU.H、一个汇编文件 OS_CPU_A.ASM 和一个 C 代码文件 OS_CPU_C.C。

以下介绍当使用 MDK RealView 编译器时，移植μC/OS-Ⅱ的主要内容。

（1）基本的配置和定义。

需要完成的基本配置和定义全部集中在 OS_CPU.H 头文件中。

① 定义与编译器相关的数据类型。

为了保证可移植性，程序中没有直接使用 C 语言中的 short、int 和 long 等数据类型的定义，因为它们与处理器类型有关，隐含着不可移植性。程序中自己定义了一套数据类型，如 INT16U 表示 16 位无符号整型。对于 ARM 这样的 32 位内核，INT16U 是 unsigned short 型；如果是 16 位的处理器，则是 unsighed int 型。在 STM32F103 处理器上实现的数据类型定义代码如下：

```
typedef unsigned char BOOLEAN;
typedef unsigned char INT8U;
typedef signed char INT8S;
typedef unsigned short INT16U;
typedef signed short INT16S;
typedef unsigned int INT32U;
typedef signed int INT32S;
typedef float FP32;
typedef double FP64;
typedef unsigned int OS_STK;
typedef unsigned int OS_CPU_SR;
```

② 定义允许和禁止中断宏。

与所有实时内核一样，μC/OS-Ⅱ需要先禁止中断，再访问代码的临界区，并且在访问完毕后，重新允许中断。这就使得μC/OS-Ⅱ能够保护临界段代码免受多任务或中断服务历程 ISR 的破坏。中断禁止时间是商业实时内核公司提供的重要指标之一，因为它将影响到用户的系统对实时事件的响应能力。虽然μC/OS-Ⅱ尽量使中断禁止时间达到最短，但是μC/OS-Ⅱ的中断禁止时间还主要依赖于处理器结构和编译器产生的代码的质量。通常每个处理器都会提供一定的指令来禁止/允许中断，因此用户的 C 编译器必须由一定的机制来直接从 C 中执行这些操作。

μC/OS-Ⅱ定义了两个宏来禁止和允许中断：OS_ENTER_CRITICAL()和 OS_EXIT_CRITIVAL()。

在 STM32F103 处理器上实现的代码如下：

```
#define  OS_CRITICAL_METHOD   3

#define  OS_ENTER_CRITICAL()  {cpu_sr = OS_CPU_SR_Save();}
#define  OS_EXIT_CRITICAL()   {OS_CPU_SR_Restore(cpu_sr);}
```

其中，OS_CPU_SR_Save()和 OS_CPU_SR_Restore 用汇编语言定义，代码如下：

```
OS_CPU_SR_Save
       MRS     R0, PRIMASK      ; set prio int mask to mask all (except faults)
       CPSID   I
       BX      LR

OS_CPU_SR_Restore
       MSR     PRIMASK, R0
       BX      LR
```

③ 定义栈的增长方向。

μC/OS-Ⅱ使用结构常量 OS_STK_GROWTH 来指定堆栈的增长方式：

- 置 OS_STK_GROWTH 为 0，表示堆栈从下往上增长；
- 置 OS_STK_GROWTH 为 1，表示堆栈从上往下增长。

虽然 ARM 处理器核对两种方式均支持，但 GCC 的 C 语言编译器仅支持一种方式，即从上往下增长，并且是满递减堆栈。所以 OS_STK_GROWTH 的值为 1，它在 OS_CPU.H 中定义。用户规划好栈的增长方向后，便定义了符合 OS_STK_GROWTH 的值。

STM32F103 处理器上实现定义堆栈增长方向的代码如下：

```
#define OS_STK_GROWTH 1
```

④ 定义 OS_TASK_SW()宏。

OS_TASK_SW()宏是μC/OS-Ⅱ从低优先级任务切换到高优先级任务时被调用的。可采用下面两种方式定义：如果处理器支持软中断，则可使用软中断将中断向量指向 OSCtxSW()函数；或者直接调用 OSCtxSw()函数。

μC/OS-Ⅱ在 STM32F103 处理器上实现 OSCtxSw 的代码如下，该段代码由汇编语言实现。

```
OSCtxSw
    LDR     R4, =NVIC_INT_CTRL      ; trigger the PendSV exception (causes context switch)
    LDR     R5, =NVIC_PENDSVSET
    STR     R5, [R4]
    BX      LR
```

（2）移植汇编语言编写的 4 个与处理器相关的函数 OS_CPU_A.ASM。

① OSStartHighRdy()：运行优先级最高的就绪任务。

OSStartHighRdy()函数是在 OSStart()多任务启动之后，负责从最高优先级任务的 TCB 控制块中获得该任务的堆栈指针 SP，并通过 SP 依次将 CPU 现场恢复。这时系统就将控制权交给用户创建的任务进程，直到该任务被阻塞或者被其他更高优先级的任务抢占 CPU。该函数仅仅在多任务启动时被执行一次，用来启动最高优先级的任务执行。移植该函数的原因是，它涉及将处理器寄存器保存到堆栈的操作。

μC/OS-Ⅱ在 STM32F103 处理器上实现 OSStartHighRdy 的代码如下：

```
OSStartHighRdy

    LDR     R4, =NVIC_SYSPRI2       ; set the PendSV exception priority
    LDR     R5, =NVIC_PENDSV_PRI
    STR     R5, [R4]

    MOV     R4, #0        ; set the PSP to 0 for initial context switch call
    MSR     PSP, R4

    LDR     R4, =OSRunning          ;//设置 OSRunning = TRUE
    MOV     R5, #1
    STRB    R5, [R4]

                                    ;//切换到最高优先级的任务
    LDR     R4, =NVIC_INT_CTRL      ; trigger the PendSV exception (causes context switch)
    LDR     R5, =NVIC_PENDSVSET
    STR     R5, [R4]

    CPSIE   I           ; enable interrupts at processor levelOSStartHang
    B       OSStartHang             ; should never get here
```

② OSCtxSw()：任务优先级切换函数。

该函数由 OS_TASK_SW()宏调用，OS_TASK_SW()由 OSSched()函数调用，OSSched()函数负责任务之间的调度。OSCtxSw()函数的工作是，先将当前任务的 CPU 现场保存到该任务的堆栈中，然后获得最高优先级任务的堆栈指针，并从该堆栈中恢复此任务的 CPU 现场，使之继续执行，该函数就完成了一次任务切换。

③ OSInitCtxSw()：中断级的任务切换函数。

该函数由 OSIntExit()调用。由于中断可能会使更高优先级的任务进入就绪态，因此，为了让更高优先级的任务能立即运行，在中断服务子程序的最后，OSInitExit()函数会调用 OSIntCtxSw()做任务切换。这样做的目的主要是能够尽快地让高优先级的任务得到响应，保证系统的实时性能。OSIntCtxSw()与 OSCtxSw()都是用于任务切换的函数，其区别在于，在 OSIntCtxSw()中无需再保存 CPU 寄存器，因为在调用 OSIntCtxSw()之前已发生了中断，OSIntCtxSw()已将默认的 CPU 寄存器保存到了被中断的任务堆栈中。

④ OSTickISR()：时钟节拍中断服务函数。

时钟节拍是特定的周期性中断，是由硬件定时器产生的。时钟的节拍式中断使得内核可将任务延时若干个整数时钟节拍，以及当任务等待事件发生时，提供等待超时的依据。时钟节拍频率越高，系统的额外开销越大。中断间的时间间隔取决于不同的应用。

OSTickISR()首先将 CPU 寄存器的值保存在被中断任务的堆栈中，之后调用 OSIntEnter()。随后，OSTickISR()调用 OSTimeTick，检查所有处于延时等待状态的任务，判断是否有延时结束就绪的任务。OSTickISR()最后调用 OSIntExit()。如果在中断中（或其他嵌套的中断）有更高优先级的任务就绪，并且当前中断为中断嵌套的最后一层，那么 OSIntExit()将进行任务调度。

（3）移植 C 语言编写的 6 个与操作系统相关的函数 OS_CPU_C.C。

OS_CPU_C.C 文件中包含 6 个和 CPU 相关的函数，这 6 个函数为 OSTaskStkInit()、OSTask DelHook()、OSTaskSwHook()、OSTaskStartHook()及 OSTimeTickHook()。

这些函数中，唯一必须移植的是任务堆栈初始化函数 OSTaskStkInit()。这个函数在任务创建时被调用，负责初始化任务的堆栈结构并返回新堆栈的指针 stk。堆栈初始化工作结束后，返回新的堆栈栈顶指针。

μC/OS-Ⅱ在 STM32F103 处理器上实现 OSTaskStkInit 的代码如下：

```
    OS_STK *OSTaskStkInit (void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT16U opt)
    {
        OS_STK *stk;

        (void)opt;           /* 'opt' is not used, prevent warning         */
        stk      = ptos;     /* Load stack pointer                         */

                             /* Registers stacked as if auto-saved on exception*/
        *(stk)    = (INT32U)0x01000000L;  /* xPSR                              */
        *(--stk)  = (INT32U)task;     /* Entry Point                               */
        *(--stk)  = (INT32U)0xFFFFFFFEL;      /* R14 (LR) (init value will cause fault if ever
used)*/
        *(--stk)  = (INT32U)0x12121212L;  /* R12                           */
        *(--stk)  = (INT32U)0x03030303L;      /* R3                                */
        *(--stk)  = (INT32U)0x02020202L;      /* R2                            */
        *(--stk)  = (INT32U)0x01010101L;  /* R1                            */
        *(--stk)  = (INT32U)p_arg;        /* R0 : argument                 */

                                 /* Remaining registers saved on process stack   */
        *(--stk)  = (INT32U)0x11111111L;      /* R11                           */
        *(--stk)  = (INT32U)0x10101010L;      /* R10                               */
        *(--stk)  = (INT32U)0x09090909L;      /* R9                                */
        *(--stk)  = (INT32U)0x08080808L;  /* R8                            */
        *(--stk)  = (INT32U)0x07070707L;      /* R7                            */
        *(--stk)  = (INT32U)0x06060606L;  /* R6                            */
        *(--stk)  = (INT32U)0x05050505L;      /* R5                            */
        *(--stk)  = (INT32U)0x04040404L;      /* R4                            */

        return (stk);
```

```
    }
```

其他 5 个均为 Hook 函数，又被称为钩子函数，主要用来控制μC/OS-Ⅱ功能，必须被声明，但并不一定要包含任何代码。

- OSTaskCreateHook()：当用 OSTaskCreate()或 OSTaskCreateExt()建立任务时，就会调用 OSTaskCreateHook()。μC/OS-Ⅱ设置完自己的内部结构后，会在调用任务调度程序之前调用 OSTaskCreateHook()。该函数被调用时中断是禁止的，因此应尽量减少该函数中的代码，以缩短中断的响应时间。
- OSTaskDelHook()：当任务被删除时，就会调用 OSTaskDelHook()。函数在把任务从μC/OS-Ⅱ的内部任务链表中解开之前被调用。当 OSTaskDelHook()被调用时，会收到指向正被删除任务的 OS_TCB 的指针，这样它就可以访问所有的结构成员了。OSTaskDelHook()可用来检验 TCB 扩展是否被建立了（一个非空指针），并进行一些清除操作。此函数不返回任何值。
- OSTaskSwHook()：当发生任务切换时，调用 OSTaskSwHook()。不管任务切换是通过 OSCtxSw()还是通过 OSIntCtxSw()来执行的，都会调用该函数。OSTaskSwHook()可直接访问 OSTCBCur 和 OSTCBHighRdy，因为它们都是全局变量。OSTCBur 指向被切换出去的任务的 OS_TCB，而 OSTCBHighRdy 指向新任务的 OS_TCB。

因为代码的多少会影响到中断的响应时间，所以应尽量使代码简化。此函数没有任何参数，也不返回任何值。

- OSTaskStatHook()：OSTaskStatHook()每秒会被 OSTaskStart()调用一次。可用 OSTask StatHook()来扩展统计功能。例如，可保持并显示每个任务的执行时间、每个任务所占用的 CPU 份额以及每个任务执行的频率等。该函数没有任何参数，也不返回任何值。
- OSTimeTickHook()：该函数在每个时钟节拍都会被 OSTimeTick()调用。实际上，OSTimeTickHook()是在节拍被μC/OS-Ⅱ处理，并在通知用户的移植实例或应用程序之前被调用的。OSTimeTickHook()没有任何参数，也不返回任何值。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218