

## 第八章 移植μC/OS-II

这一章介绍如何将μC/OS-II移植到不同的处理器上。所谓移植，就是使一个实时内核能在某个微处理器或微控制器上运行。为了方便移植，大部分的μC/OS-II代码是用C语言写的；但仍需要用C和汇编语言写一些与处理器相关的代码，这是因为μC/OS-II在读写处理器寄存器时只能通过汇编语言来实现。由于μC/OS-II在设计时就已经充分考虑了可移植性，所以μC/OS-II的移植相对来说是比较容易的。如果已经有人在您使用的处理器上成功地移植了μC/OS-II，您也得到了相关代码，就不必看本章了。当然，本章介绍的内容将有助于用户了解μC/OS-II中与处理器相关的代码。

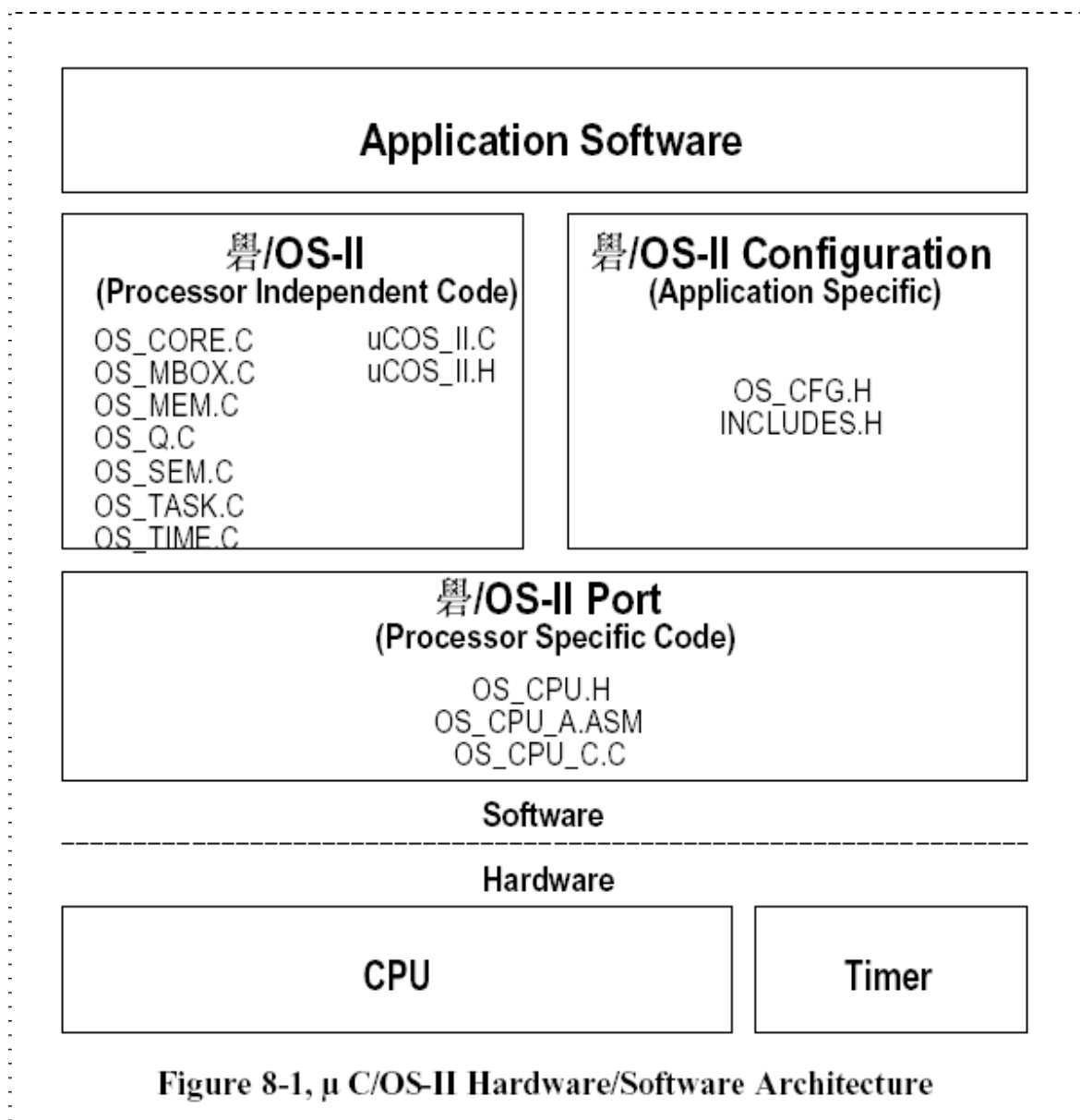
要使μC/OS-II正常运行，处理器必须满足以下要求：

1. 处理器的C编译器能产生可重入代码。
2. 用C语言就可以打开和关闭中断。
3. 处理器支持中断，并且能产生定时中断(通常在10至100Hz之间)。
4. 处理器支持能够容纳一定量数据(可能是几千字节)的硬件堆栈。
5. 处理器有将堆栈指针和其它CPU寄存器读出和存储到堆栈或内存中的指令。

像Motorola 6805系列的处理器不能满足上面的第4条和第5条要求，所以μC/OS-II不能在这类处理器上运行。

图8.1说明了μC/OS-II的结构以及它与硬件的关系。由于μC/OS-II为自由软件，当用户用到μC/OS-II时，有责任公开应用软件和μC/OS-II的配置代码。这本书和磁盘包含了所有与处理器无关的代码和Intel 80x86实模式下的与处理器相关的代码(C编译器大模式下编译)。如果用户打算在其它处理器上使用μC/OS-II，最好能找到一个现成的移植实例，如果没有只好自己编写了。用户可以在正式的μC/OS-II网站 [www. μCOS-II.com](http://www.μCOS-II.com) 中查找一些移植实例。

图 8.1  $\mu$ C/OS-II 硬件和软件体系结构



如果用户理解了处理器和 C 编译器的技术细节，移植 $\mu$ C/OS-II 的工作实际上是非常简单的。前提是您的处理器和编译器满足了 $\mu$ C/OS-II 的要求，并且已经有了必要工具。移植工作包括以下几个内容：

- 用#define 设置一个常量的值 (OS\_CPU.H)
- 声明 10 个数据类型 (OS\_CPU.H)
- 用#define 声明三个宏 (OS\_CPU.H)
- 用 C 语言编写六个简单的函数 (OS\_CPU\_C.C)
- 编写四个汇编语言函数 (OS\_CPU\_A.ASM)

根据处理器的不同，一个移植实例可能需要编写或改写 50 至 300 行的代码，需要的时间从几个小时到一星期不等。

一旦代码移植结束，下一步工作就是测试。测试一个象 $\mu$ C/OS-II 一样的多任务实时内核并不复杂。甚至可以在没有应用程序的情况下测试。换句话说，就是让内核自己测试自己。

这样做有两个好处：第一，避免使本来就复杂的事情更加复杂；第二，如果出现问题，可以知道问题出在内核代码上而不是应用程序。刚开始的时候可以运行一些简单的任务和时钟节拍中断服务例程。一旦多任务调度成功地运行了，再添加应用程序的任务就是非常简单的工作了。

## 8.00 开发工具

如前所述，移植 $\mu$ C/OS-II 需要一个 C 编译器，并且是针对用户用的 CPU 的。因为 $\mu$ C/OS-II 是一个可剥夺型内核，用户只有通过 C 编译器来产生可重入代码；C 编译器还要支持汇编语言程序。绝大部分的 C 编译器都是为嵌入式系统设计的，它包括汇编器、连接器和定位器。连接器用来将不同的模块(编译过和汇编过的文件)连接成目标文件。定位器则允许用户将代码和数据放置在目标处理器的指定内存映射空间中。所用的 C 编译器还必须提供一个机制来从 C 中打开和关闭中断。一些编译器允许用户在 C 源代码中插入汇编语言。这就使得插入合适的处理器指令来允许和禁止中断变得非常容易了。还有一些编译器实际上包括了语言扩展功能，可以直接从 C 中允许和禁止中断。

## 8.01 目录和文件

本书所付的磁盘中提供了 $\mu$ C/OS-II 的安装程序，可在硬盘上安装 $\mu$ C/OS-II 和移植实例代码（Intel 80x86 实模式，大模式编译）。我设计了一个连续的目录结构，使得用户更容易找到目标处理器的文件。如果想增加一个其它处理器的移植实例，您可以考虑采取同样的方法（包括目录的建立和文件的命名等等）。

所有的移植实例都应放在用户硬盘的\SOFTWARE\ $\mu$ COS-II 目录下。各个微处理器或微控制器的移植源代码必须在以下两个或三个文件中找到：OS\_CPU.H，OS\_CPU.C.C，OS\_CPU\_A.ASM。汇编语言文件 OS\_CPU\_A.ASM 是可选择的，因为某些 C 编译器允许用户在 C 语言中插入汇编语言，所以用户可以将所需的汇编语言代码直接放到 OS\_CPU.C.C 中。放置移植实例的目录决定于用户所用的处理器，例如在下面的表中所示的放置不同移植实例的目录结构。注意，各个目录虽然针对完全不同的目标处理器，但都包括了相同的文件名。

Intel/AMD 80186	\SOFTWARE\uCOS-II\Ix86S
	\OS_CPU.H
	\OS_CPU_A.ASM
	\OS_CPU_C.C
	\SOFTWARE\uCOS-II\Ix86L
	\OS_CPU.H
	\OS_CPU_A.ASM
	\OS_CPU_C.C
Motorola 68HC11	\SOFTWARE\uCOS-II\68HC11
	\OS_CPU.H
	\OS_CPU_A.ASM
	\OS_CPU_C.C

## 8.02 INCLUDES.H

在第一章中曾提到过, INCLUDES.H 是一个头文件, 它在所有.C 文件的第一行被包含。

```
#include "includes.h"
```

INCLUDES.H 使得用户项目中的每个.C 文件不用分别去考虑它实际上需要哪些头文件。使用 INCLUDES.H 的唯一缺点是它可能会包含一些实际不相关的头文件。这意味着每个文件的编译时间可能会增加。但由于它增强了代码的可移植性, 所以我们还是决定使用这一方法。用户可以通过编辑 INCLUDES.H 来增加自己的头文件, 但是用户的头文件必须添加在头文件列表的最后。

## 8.03 OS\_CPU.H

OS\_CPU.H 包括了用#define 定义的与处理器相关的常量, 宏和类型定义。OS\_CPU.H 的大体结构如程序清单 L8.1 所示。

### 程序清单 L8.1 OS\_CPU.H

```
#ifndef OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT extern
#endif

/*
*****
*
*          数据类型
*          (与编译器相关)
*****
*/

typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;      /* 无符号8位整数 */ (1)
typedef signed  char  INT8S;      /* 有符号8位整数 */
typedef unsigned int   INT16U;    /* 无符号16位整数 */
typedef signed  int   INT16S;    /* 有符号16位整数 */
typedef unsigned long  INT32U;    /* 无符号32位整数 */
typedef signed  long  INT32S;    /* 有符号32位整数 */
typedef float          FP32;      /* 单精度浮点数 */ (2)
typedef double         FP64;      /* 双精度浮点数 */

typedef unsigned int   OS_STK;    /* 堆栈入口宽度为16位 */

/*
*****
*
*          与处理器相关的代码
*****
*/
```

```

#define OS_ENTER_CRITICAL() ??? /* 禁止中断 */ (3)
#define OS_EXIT_CRITICAL() ??? /* 允许中断 */

#define OS_STK_GROWTH 1 /* 定义堆栈的增长方向： 1=向下， 0=向上 */ (4)

#define OS_TASK_SW() ??? (5)

```

### 8.03.01 与编译器相关的数据类型

因为不同的微处理器有不同的字长，所以 $\mu$ C/OS-II的移植包括了一系列的类型定义以确保其可移植性。尤其是， $\mu$ C/OS-II代码从不使用C的short, int 和 long 等数据类型，因为它们是与编译器相关的，不可移植。相反的，我定义的整型数据结构既是可移植的又是直观的[L8.1(2)]。为了方便，虽然 $\mu$ C/OS-II不使用浮点数据，但我还是定义了浮点数据类型[L8.1(2)]。

例如，INT16U 数据类型总是代表 16 位的无符号整数。现在， $\mu$ C/OS-II 和用户的应用程序就可以估计出声明为该数据类型的变量的数值范围是 0—65535。将 $\mu$ C/OS-II 移植到 32 位的处理器上也就意味着 INT16U 实际被声明为无符号短整型数据结构而不是无符号整型数据结构。但是， $\mu$ C/OS-II 所处理的仍然是 INT16U。

用户必须将任务堆栈的数据类型告诉给 $\mu$ C/OS-II。这个过程是通过为 OS\_STK 声明正确的 C 数据类型来完成的。如果用户的处理器上的堆栈成员是 32 位的，并且用户的编译文件指定整型为 32 位数，那么就应该将 OS\_STK 声明为无符号整型数据类型。所有的任务堆栈都必须用 OS\_STK 来声明数据类型。

用户所必须要做的就是查看编译器手册，并找到对应于 $\mu$ C/OS-II 的标准 C 数据类型。

### 8.03.02 OS\_ENTER\_CRITICAL() 和 OS\_EXIT\_CRITICAL()

与所有的实时内核一样， $\mu$ C/OS-II 需要先禁止中断再访问代码的临界段，并且在访问完毕后重新允许中断。这就使得 $\mu$ C/OS-II 能够保护临界段代码免受多任务或中断服务例程 (ISRs) 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 $\mu$ C/OS-II 尽量使中断禁止时间达到最短，但是 $\mu$ C/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 编译器必须要有一定的机制来直接从 C 中执行这些操作。有些编译器能够允许用户在 C 源代码中插入汇编语言声明。这样就使得插入处理器指令来允许和禁止中断变得很容易了。其它一些编译器实际上包括了语言扩展功能，可以直接从 C 中允许和禁止中断。为了隐藏编译器厂商提供的具体实现方法， $\mu$ C/OS-II 定义了两个宏来禁止和允许中断：OS\_ENTER\_CRITICAL() 和 OS\_EXIT\_CRITICAL() [L8.1(3)]。

```

{
    OS_ENTER_CRITICAL();
    /*  $\mu$ C/OS-II 临界代码段 */
    OS_EXIT_CRITICAL();
}

```

## 方法 1

执行这两个宏的第一个也是最简单的方法是在 `OS_ENTER_CRITICAL()` 中调用处理器指令来禁止中断, 以及在 `OS_EXIT_CRITICAL()` 中调用允许中断指令。但是, 在这个过程中还存在着小小的问题。如果用户在禁止中断的情况下调用  $\mu\text{C}/\text{OS-II}$  函数, 在从  $\mu\text{C}/\text{OS-II}$  返回的时候, 中断可能会变成是允许的了! 如果用户禁止中断就表明用户想在从  $\mu\text{C}/\text{OS-II}$  函数返回的时候中断还是禁止的。在这种情况下, 光靠这种执行方法可能是不够的。

## 方法 2

执行 `OS_ENTER_CRITICAL()` 的第二个方法是先将中断禁止状态保存到堆栈中, 然后禁止中断。而执行 `OS_EXIT_CRITICAL()` 的时候只是从堆栈中恢复中断状态。如果用这个方法的话, 不管用户是在中断禁止还是允许的情况下调用  $\mu\text{C}/\text{OS-II}$  服务, 在整个调用过程中都不会改变中断状态。如果用户在中断禁止的时候调用  $\mu\text{C}/\text{OS-II}$  服务, 其实用户是在延长应用程序的中断响应时间。用户的应用程序还可以用 `OS_ENTER_CRITICAL()` 和 `OS_EXIT_CRITICAL()` 来保护代码的临界段。但是, 用户在使用这种方法的时候还得十分小心, 因为如果用户在调用象 `OSTimeDly()` 之类的服务之前就禁止中断, 很有可能用户的应用程序会崩溃。发生这种情况的原因是任务被挂起直到时间期满, 而中断是禁止的, 因而用户不可能获得节拍中断! 很明显, 所有的 `PEND` 调用都会涉及到这个问题, 用户得十分小心。一个通用的办法是用户应该在中断允许的情况下调用  $\mu\text{C}/\text{OS-II}$  的系统服务!

问题是: 哪种方法更好一点? 这就得看用户想牺牲些什么。如果用户并不关心在调用  $\mu\text{C}/\text{OS-II}$  服务后用户的应用程序中中断是否是允许的, 那么用户应该选择第一种方法执行。如果用户想在调用  $\mu\text{C}/\text{OS-II}$  服务过程中保持中断禁止状态, 那么很明显用户应该选择第二种方法。

给用户举个例子吧, 通过执行 `STI` 命令在 Intel 80186 上禁止中断, 并用 `CLI` 命令来允许中断。用户可以用下面的方法来执行这两个宏:

```
#define OS_ENTER_CRITICAL()  asm CLI
#define OS_EXIT_CRITICAL()   asm STI
```

`CLI` 和 `STI` 指令都会在两个时钟周期内被马上执行(总共为四个周期)。为了保持中断状态, 用户需要用下面的方法来执行宏:

```
#define OS_ENTER_CRITICAL()  asm PUSHF; CLI
#define OS_EXIT_CRITICAL()   asm POPF
```

在这种情况下, `OS_ENTER_CRITICAL()` 需要 12 个时钟周期, 而 `OS_EXIT_CRITICAL()` 需要另外的 8 个时钟周期(总共有 20 个周期)。这样, 保持中断禁止状态要比简单的禁止/允许中断多花 16 个时钟周期的时间(至少在 80186 上是这样的)。当然, 如果用户有一个速度比较快的处理器(如 Intel Pentium II), 那么这两种方法的时间差别会很小。

### 8.03.03 OS\_STK\_GROWTH

绝大多数的微处理器和微控制器的堆栈是从上往下长的。但是某些处理器是用另外一种方式工作的。 $\mu\text{C}/\text{OS-II}$  被设计成两种情况都可以处理, 只要在结构常量 `OS_STK_GROWTH` [L8.1(4)]中指定堆栈的生长方式(如下所示)就可以了。

置 `OS_STK_GROWTH` 为 0 表示堆栈从下往上长。

置 `OS_STK_GROWTH` 为 1 表示堆栈从上往下长。

### 8.03.04 OS\_TASK\_SW()

`OS_TASK_SW()` [L8.1(5)]是一个宏, 它是在  $\mu\text{C}/\text{OS-II}$  从低优先级任务切换到最高优先级任务时被调用的。`OS_TASK_SW()` 总是在任务级代码中被调用的。另一个函数 `OSIntExit()` 被

用来在 ISR 使得更高优先级任务处于就绪状态时，执行任务切换功能。任务切换只是简单的将处理器寄存器保存到将被挂起的任务的堆栈中，并且将更高优先级的任务从堆栈中恢复出来。

在  $\mu\text{C}/\text{OS-II}$  中，处于就绪状态的任务的堆栈结构看起来就像刚发生过中断并将所有的寄存器保存到堆栈中的情形一样。换句话说， $\mu\text{C}/\text{OS-II}$  要运行处于就绪状态的任务必须要做的事就是将所有处理器寄存器从任务堆栈中恢复出来，并且执行中断的返回。为了切换任务可以通过执行 `OS_TASK_SW()` 来产生中断。大部分的处理器会提供软中断或是陷阱 (TRAP) 指令来完成这个功能。ISR 或是陷阱处理函数 (也叫做异常处理函数) 的向量地址必须指向汇编语言函数 `OSCtxSw()` (参看 8.04.02)。

例如，在 Intel 或者 AMD 80x86 处理器上可以使用 `INT` 指令。但是中断处理向量需要指向 `OSCtxSw()`。Motorola 68HC11 处理器使用的是 `SWI` 指令，同样，`SWI` 的向量地址仍是 `OSCtxSw()`。还有，Motorola 680x0/CPU32 可能会使用 16 个陷阱指令中的一个。当然，选中的陷阱向量地址还是 `OSCtxSw()`。

一些处理器如 Zilog Z80 并不提供软中断机制。在这种情况下，用户需要尽自己的所能将堆栈结构设置成与中断堆栈结构一样。`OS_TASK_SW()` 只会简单的调用 `OSCtxSw()` 而不是将某个向量指向 `OSCtxSw()`。 $\mu\text{C}/\text{OS}$  已经被移植到了 Z80 处理器上， $\mu\text{C}/\text{OS-II}$  也同样可以。

## 8.04 OS\_CPU\_A.ASM

$\mu\text{C}/\text{OS-II}$  的移植实例要求用户编写四个简单的汇编语言函数：

```
OSStartHighRdy()
OSCtxSw()
OSIntCtxSw()
OSTickISR()
```

如果用户的编译器支持插入汇编语言代码的话，用户就可以将所有与处理器相关的代码放到 `OS_CPU_C.C` 文件中，而不必再拥有一些分散的汇编语言文件。

### 8.04.01 OSStartHighRdy()

使就绪状态的任务开始运行的函数叫做 `OSStart()`，如下所示。在用户调用 `OSStart()` 之前，用户必须至少已经建立了自己的一个任务 (参看 `OSTaskCreate()` 和 `OSTaskCteateExt()`)。 `OSStartHighRdy()` 假设 `OSTCBHighRdy` 指向的是优先级最高的任务的任务控制块。前面曾提到过，在  $\mu\text{C}/\text{OS-II}$  中处于就绪状态的任务的堆栈结构看起来就像刚发生过中断并将所有的寄存器保存到堆栈中的情形一样。要想运行最高优先级任务，用户所要做的是将所有处理器寄存器按顺序从任务堆栈中恢复出来，并且执行中断的返回。为了简单一点，堆栈指针总是储存在任务控制块 (即它的 `OS_TCB`) 的开头。换句话说，也就是要想恢复的任务堆栈指针总是储存在 `OS_TCB` 的 0 偏址内存单元中。

```
void OSStartHighRdy (void)
{
    Call user definable OSTaskSwHook();
    Get the stack pointer of the task to resume:
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    OSRunning = TRUE;
    Restore all processor registers from the new task's stack;
    Execute a return from interrupt instruction;
}
```

注意，OSStartHighRdy() 必须调用 OSTaskSwHook()，因为用户正在进行任务切换的部分工作——用户在恢复最高优先级任务的寄存器。而 OSTaskSwHook() 可以通过检查 OSRunning 来知道是 OSStartHighRdy() 在调用它(OSRunning 为 FALSE) 还是正常的任务切换在调用它(OSRunning 为 TRUE)。

OSStartHighRdy() 还必须在最高优先级任务恢复之前和调用 OSTaskSwHook() 之后设置 OSRunning 为 TRUE。

#### 8.04.02 OSCtxSw()

如前面所述，任务级的切换问题是通过发软中断命令或依靠处理器执行陷阱指令来完成的。中断服务例程，陷阱或异常处理例程的向量地址必须指向 OSCtxSw()。

如果当前任务调用  $\mu$ C/OS-II 提供的系统服务，并使得更高优先级任务处于就绪状态， $\mu$ C/OS-II 就会借助上面提到的向量地址找到 OSCtxSw()。在系统服务调用的最后， $\mu$ C/OS-II 会调用 OSSched()，并由此来推断当前任务不再是要运行的最重要的任务了。OSSched() 先将最高优先级任务的地址装载到 OSTCBHighRdy 中，再通过调用 OS\_TASK\_SW() 来执行软中断或陷阱指令。注意，变量 OSTCBCur 早就包含了指向当前任务的任务控制块(OS\_TCB)的指针。软中断（或陷阱）指令会强制一些处理器寄存器(比如返回地址和处理器状态字)到当前任务的堆栈中，并使处理器执行 OSCtxSw()。OSCtxSw() 的原型如程序清单 L8.2 所示。这些代码必须写在汇编语言中，因为用户不能直接从 C 中访问 CPU 寄存器。注意在 OSCtxSw() 和用户定义的函数 OSTaskSwHook() 的执行过程中，中断是禁止的。

#### 程序清单 L8.2 OSCtxSw() 的原型

```
void OSCtxSw(void)
{
    保存处理器寄存器；
    将当前任务的堆栈指针保存到当前任务的OS_TCB中：
        OSTCBCur->OSTCBStkPtr = Stack pointer；
    调用用户定义的OSTaskSwHook()；
    OSTCBCur = OSTCBHighRdy；
    OSPrioCur = OSPrioHighRdy；
    得到需要恢复的任务的堆栈指针：
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr；
    将所有处理器寄存器从新任务的堆栈中恢复出来；
    执行中断返回指令；
}
```

#### 8.04.03 OSIntCtxSw()

OSIntExit() 通过调用 OSIntCtxSw() 来从 ISR 中执行切换功能。因为 OSIntCtxSw() 是在 ISR 中被调用的，所以可以断定所有的处理器寄存器都被正确地保存到了被中断的任务的堆栈之中。实际上除了我们需要的东西外，堆栈结构中还有其它的一些东西。OSIntCtxSw() 必须要清理堆栈，这样被中断的任务的堆栈结构内容才能满足我们的需要。

要了解 OSIntCtxSw()，用户可以看看  $\mu$ C/OS-II 调用该函数的过程。用户可以参看图 8.2 来帮助理解下面的描述。假定中断不能嵌套(即 ISR 不会被中断)，中断是允许的，并且处理器正在执行任务级的代码。当中断来临的时候，处理器会结束当前的指令，识别中断并且初始化中断处理过程，包括将处理器的状态寄存器和返回被中断的任务的地址保存到堆栈中[F8.2(1)]。至于究竟哪些寄存器保存到了堆栈上，以及保存的顺序是怎样的，并不重要。



图 8.2 在ISR执行过程中的堆栈内容

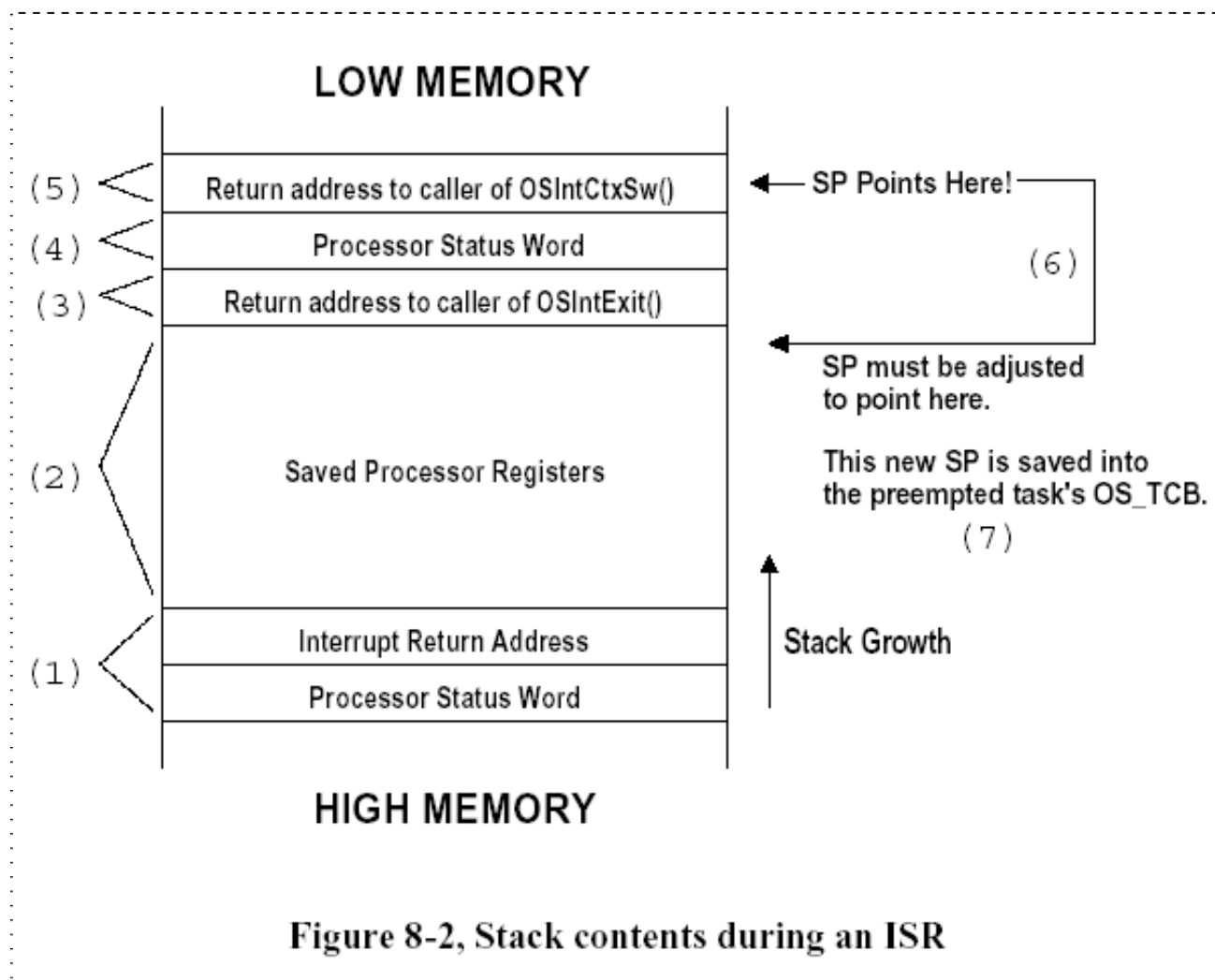


Figure 8-2, Stack contents during an ISR

接着，CPU 会调用正确的 ISR。 $\mu\text{C}/\text{OS-II}$  要求用户的 ISR 在开始时要保存剩下的处理器寄存器[F8.2(2)]。一旦寄存器保存好了， $\mu\text{C}/\text{OS-II}$  就要求用户或者调用 OSIntEnter()，或者将变量 OSIntNesting 加 1。在这个时候，被中断任务的堆栈中只包含了被中断任务的寄存器内容。现在，ISR 可以执行中断服务了。并且如果 ISR 发消息给任务(通过调用 OSMboxPost() 或 OSQPost())，恢复任务(通过调用 OSTaskResume())，或者调用 OSTimeTick() 或 OSTimeDlyResume() 的话，有可能使更高优先级的任务处于就绪状态。

假设有一个更高优先级的任务处于就绪状态。 $\mu\text{C}/\text{OS-II}$  要求用户的 ISR 在完成中断服务的时候调用 OSIntExit()。OSIntExit() 会告诉 $\mu\text{C}/\text{OS-II}$  到了返回任务级代码的时间了。调用 OSIntExit() 会导致调用者的返回地址被保存到被中断的任务的堆栈中[F8.2(3)]。

OSIntExit() 刚开始时会禁止中断，因为它需要执行临界段的代码。根据 OS\_ENTER\_CRITICAL() 的不同执行过程(参看 8.03.02)，处理器的状态寄存器会被保存到被中断的任务的堆栈中[F8.2(4)]。OSIntExit() 注意到由于有更高优先级的任务处于就绪状态，被中断的任务已经不再是要继续执行的任务了。在这种情况下，指针 OSTCBHighRdy 会被指向新任务的 OS\_TCB，并且 OSIntExit() 会调用 OSIntCtxSw() 来执行任务切换。调用 OSIntCtxSw() 也同样使返回地址被保存到被中断的任务的堆栈中[F8.2(5)]。

在用户切换任务的时候，用户只想将某些项([F8.2(1)]和[F8.2(2)])保留在堆栈中，

并忽略其它项 (F8.2(3), (4) 和 (5))。这是通过调整堆栈指针 (加一个数在堆栈指针上) 来完成的 [F8.2(6)]。加在堆栈指针上的数必须是明确的, 而这个数主要依赖于移植的目标处理器 (地址空间可能是 16, 32 或 64 位), 所用的编译器, 编译器选项, 内存模式等等。另外, 处理器状态字可能是 8, 16, 32 甚至 64 位宽, 并且 OSIntExit() 可能会分配局部变量。有些处理器允许用户直接增加常量到堆栈指针中, 而有些则不允许。在后一种情况下, 可以通过简单的执行一定数量的 pop (出栈) 指令来实现相同的功能。一旦堆栈指针完成调整, 新的堆栈指针会被保存到被切换出去的任务的 OS\_TCB 中 [F8.2(7)]。

---

OSIntCtxSw() 是  $\mu$ C/OS-II (和  $\mu$ C/OS) 中唯一的与编译器相关的函数; 在我收到的 e-mail 中, 关于该函数的 e-mail 明显多于关于  $\mu$ C/OS 其它方面的。如果在多次任务切换后用户的系统崩溃了, 用户应该怀疑堆栈指针在 OSIntCtxSw() 中是否被正确地调整了。

---

OSIntCtxSw() 的原型如程序清单 L8.3 所示。这些代码必须写在汇编语言中, 因为用户不能直接从 C 语言中访问 CPU 寄存器。如果用户的编译器支持插入汇编语言代码的话, 用户就可以将 OSIntCtxSw() 代码放到 OS\_CPU\_C.C 文件中, 而不放到 OS\_CPU\_A.ASM 文件中。正如用户所看到的那样, 除了第一行以外, OSIntCtxSw() 的代码与 OSCtxSw() 是一样的。这样在移植实例中, 用户可以通过“跳转”到 OSCtxSw() 中来减少 OSIntCtxSw() 代码量。

### 程序清单 L8.3 OSIntCtxSw() 的原型

```
void OSIntCtxSw(void)
{
    调整堆栈指针来去掉在调用:
        OSIntExit(),
        OSIntCtxSw() 过程中压入堆栈的多余内容;
    将当前任务堆栈指针保存到当前任务的 OS_TCB 中:
        OSTCBCur->OSTCBStkPtr = 堆栈指针;
    调用用户定义的 OSTaskSwHook();
    OSTCBCur = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    得到需要恢复的任务的堆栈指针:
        堆栈指针 = OSTCBHighRdy->OSTCBStkPtr;
    将所有处理器寄存器从新任务的堆栈中恢复出来;
    执行中断返回指令;
}
```

#### 8.04.04 OSTickISR()

$\mu$ C/OS-II 要求用户提供一个时钟资源来实现时间的延时和期满功能。时钟节拍应该每秒钟发生 10—100 次。为了完成该任务, 可以使用硬件时钟, 也可以从交流电中获得 50/60Hz 的时钟频率。

用户必须在开始多任务调度后 (即调用 OSStart() 后) 允许时钟节拍中断。换句话说, 就是用户应该在 OSStart() 运行后,  $\mu$ C/OS-II 启动运行的第一个任务中初始化节拍中断。通常所犯的错误是在调用 OSInit() 和 OSStart() 之间允许时钟节拍中断 (如程序清单 L8.4 所示)。

### 程序清单 L 8.4 在不正确的位置启动时钟节拍中断

```
void main(void)
{
    .
    .
    OSInit();          /* 初始化  $\mu$ C/OS-II          */
    .
    .
    /* 应用程序初始化代码 ...          */
    /* ... 调用OSTaskCreate()建立至少一个任务      */
    .
    .
    允许时钟节拍中断; /* 千万不要在这里允许!!!      */
    .
    .
    OSStart();         /* 开始多任务调度          */
}
```

有可能在 $\mu$ C/OS-II开始执行第一个任务前时钟节拍中断就发生了。在这种情况下， $\mu$ C/OS-II的运行状态不确定，用户的应用程序也可能会崩溃。

时钟节拍 ISR 的原型如程序清单 L8.5 所示。这些代码必须写在汇编语言中，因为用户不能直接从 C 语言中访问 CPU 寄存器。如果用户的处理器可以通过单条指令来增加 OSIntNesting，那么用户就没必要调用 OSIntEnter() 了。增加 OSIntNesting 要比通过函数调用和返回快得多。OSIntEnter() 只增加 OSIntNesting，并且作为临界段代码中受到保护。

### 程序清单 L 8.5 时钟节拍ISR的原型

```
void OSTickISR(void)
{
    保存处理器寄存器;
    调用OSIntEnter()或者直接将 OSIntNesting加1;

    调用OSTimeTick();

    调用OSIntExit();
    恢复处理器寄存器;
    执行中断返回指令;
}
```

## 8.05 OS\_CPU\_C.C

$\mu$ C/OS-II 的移植实例要求用户编写六个简单的 C 函数：

```
OSTaskStkInit()
OSTaskCreateHook()
OSTaskDelHook()
```

```

OSTaskSwHook()
OSTaskStatHook()
OSTimeTickHook()

```

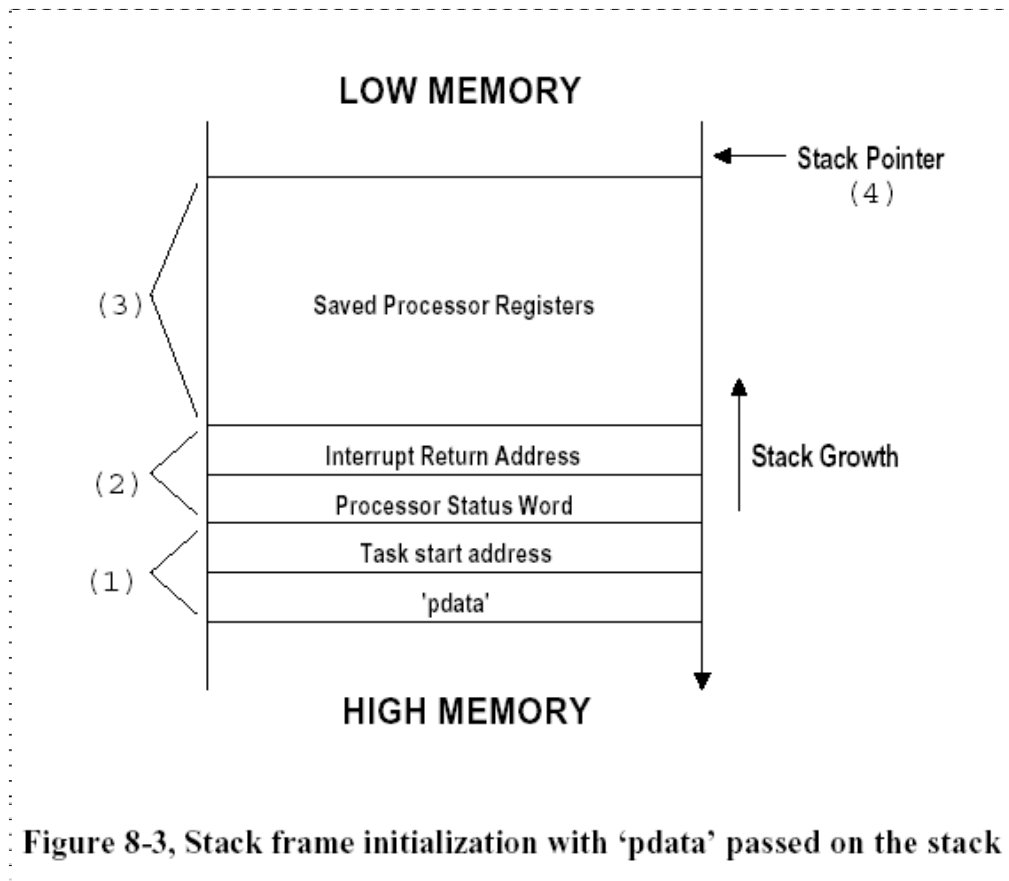
唯一必要的函数是 `OSTaskStkInit()`，其它五个函数必须得声明但没必要包含代码。

### 8.05.01 OSTaskStkInit()

`OSTaskCreate()` 和 `OSTaskCreateExt()` 通过调用 `OSTaskStkInit()` 来初始化任务的堆栈结构，因此，堆栈看起来就像刚发生过中断并将所有的寄存器保存到堆栈中的情形一样。图 8.3 显示了 `OSTaskStkInit()` 放到正被建立的任务堆栈中的东西。注意，在这里我假定了堆栈是从上往下长的。下面的讨论同样适用于从下往上长的堆栈。

在用户建立任务的时候，用户会传递任务的地址，`pdata` 指针，任务的堆栈栈顶和任务的优先级给 `OSTaskCreate()` 和 `OSTaskCreateExt()`。虽然 `OSTaskCreateExt()` 还要求有其它的参数，但这些参数在讨论 `OSTaskStkInit()` 的时候是无关紧要的。为了正确初始化堆栈结构，`OSTaskStkInit()` 只要求刚才提到的前三个参数和一个附加的选项，这个选项只能在 `OSTaskCreateExt()` 中得到。

**图 8.3 堆栈初始化 (*pdata* 通过堆栈传递)**



回顾一下，在  $\mu\text{C}/\text{OS-II}$  中，无限循环的任务看起来就像其它的 C 函数一样。当任务开始被  $\mu\text{C}/\text{OS-II}$  执行时，任务就会收到一个参数，好像它被其它的任务调用一样。

```

void MyTask (void *pdata)
{
    /* 对 'pdata' 做某些操作 */
}

```

```
for (;;) {  
    /* 任务代码 */  
}  
}
```

如果我想从其它的函数中调用 `MyTask()`，C 编译器就会先将调用 `MyTask()` 的函数的返回地址保存到堆栈中，再将参数保存到堆栈中。实际上有些编译器会将 `pdata` 参数传至一个或多个寄存器中。在后面我会讨论这类情况。假定 `pdata` 会被编译器保存到堆栈中，`OSTaskStkInit()` 就会简单的模仿编译器的这种动作，将 `pdata` 保存到堆栈中[F8.3(1)]。但是结果表明，与 C 函数调用不一样，调用者的返回地址是未知的。用户所拥有的是任务的开始地址，而不是调用该函数(任务)的函数的返回地址！事实上用户不必太在意这点，因为任务并不希望返回到其它函数中。

这时，用户需要将寄存器保存到堆栈中，当处理器发现并开始执行中断的时候，它会自动地完成该过程的。一些处理器会将所有的寄存器存入堆栈，而其它一些处理器只将部分寄存器存入堆栈。一般而言，处理器至少得将程序计数器的值（中断返回地址）和处理器的状态字存入堆栈[F8.3(2)]。很明显，处理器是按一定的顺序将寄存器存入堆栈的，而用户在进行寄存器存入堆栈的时候也就必须依照这一顺序。

接着，用户需要将剩下的处理器寄存器保存到堆栈中[F8.3(3)]。保存的命令依赖于用户的处理器是否允许用户保存它们。有些处理器用一个或多个指令就可以马上将许多寄存器都保存起来。用户必须用特定的指令来完成这一过程。例如，Intel 80x86 使用 `PUSHA` 指令将 8 个寄存器保存到堆栈中。对 Motorola 68HC11 处理器而言，在中断响应期间，所有的寄存器都会按一定顺序自动的保存到堆栈中，所以在用户将寄存器存入堆栈的时候，也必须依照这一顺序。

现在是时候讨论这个问题了：如果用户的 C 编译器将 `pdata` 参数传递到寄存器中而不是堆栈中该作些什么？用户需要从编译器的文档中找到 `pdata` 储存在哪个寄存器中。`pdata` 的内容就会随着这个寄存器的储存被放置在堆栈中。

图 8.4 堆栈初始化 (*pdata* 通过寄存器传递)

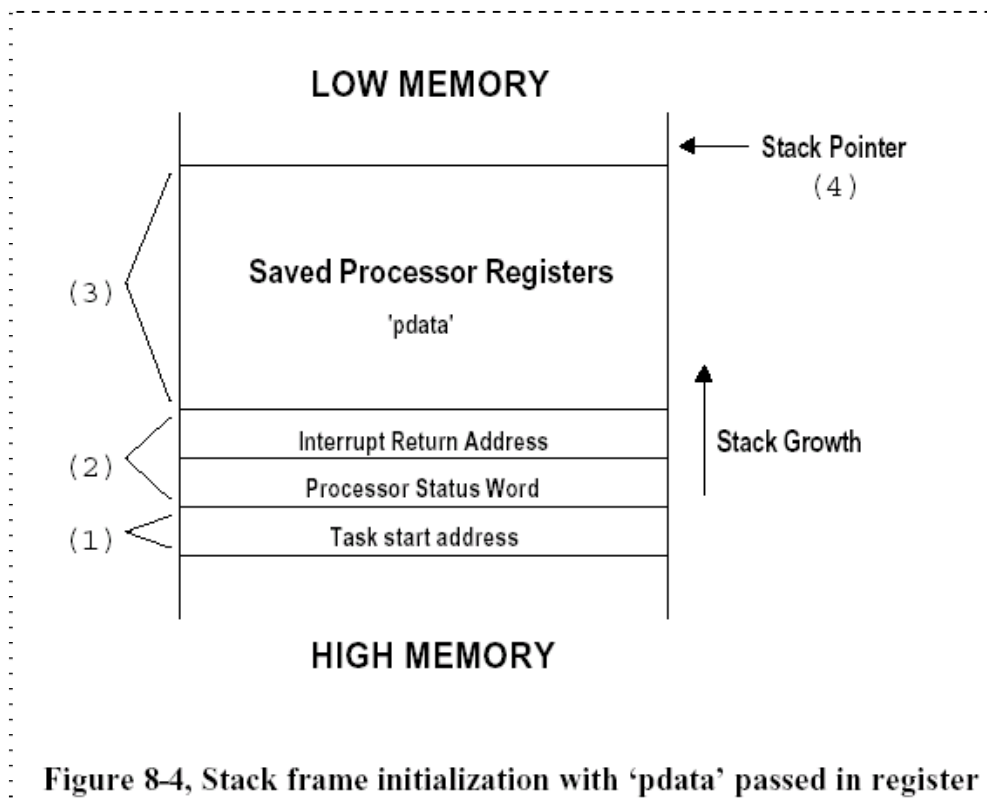


Figure 8-4, Stack frame initialization with 'pdata' passed in register

一旦用户初始化了堆栈，OSTaskStkInit() 就需要返回堆栈指针所指的地址[F8.3(4)]。OSTaskCreate() 和 OSTaskCreateExt() 会获得该地址并将它保存到任务控制块(OS\_TCB)中。处理器文档会告诉用户堆栈指针会指向下一个堆栈空闲位置，还是会指向最后存入数据的堆栈单元位置。例如，对 Intel 80x86 处理器而言，堆栈指针会指向最后存入数据的堆栈单元位置，而对 Motorola 68HC11 处理器而言，堆栈指针会指向下一个空闲的位置。

#### 8.05.02 OSTaskCreateHook()

当用 OSTaskCreate() 或 OSTaskCreateExt() 建立任务的时候就会调用 OSTaskCreateHook()。该函数允许用户或使用用户的移植实例的用户扩展  $\mu$ C/OS-II 的功能。当  $\mu$ C/OS-II 设置完了自己的内部结构后，会在调用任务调度程序之前调用 OSTaskCreateHook()。该函数被调用的时候中断是禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。

当 OSTaskCreateHook() 被调用的时候，它会收到指向已建立任务的 OS\_TCB 的指针，这样它就可以访问所有的结构成员了。当使用 OSTaskCreate() 建立任务时，OSTaskCreateHook() 的功能是有限的。但当用户使用 OSTaskCreateExt() 建立任务时，用户会得到 OS\_TCB 中的扩展指针(OSTCBExtPtr)，该指针可用来访问任务的附加数据，如浮点寄存器，MMU 寄存器，任务计数器的内容，以及调试信息。

只用当 OS\_CFG.H 中的 OS\_CPU\_HOOKS\_EN 被置为 1 时才会产生 OSTaskCreateHook() 的代码。这样，使用用户的移植实例的用户可以在其它的文件中重新定义 hook 函数。

#### 8.05.03 OSTaskDelHook()

当任务被删除的时候就会调用 OSTaskDelHook()。该函数在把任务从  $\mu$ C/OS-II 的内部任务链表中解开之前被调用。当 OSTaskDelHook() 被调用的时候，它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() 可以用来检验

TCB 扩展是否被建立了(一个非空指针)并进行一些清除操作。OSTaskDelHook() 不返回任何值。

只用当 OS\_CFG.H 中的 OS\_CPU\_HOOKS\_EN 被置为 1 时才会产生 OSTaskDelHook() 的代码。

#### **8.05.04 OSTaskSwHook()**

当发生任务切换的时候调用 OSTaskSwHook()。不管任务切换是通过 OSCtxSw() 还是 OSIntCtxSw() 来执行的都会调用该函数。OSTaskSwHook() 可以直接访问 OSTCBCur 和 OSTCBHighRdy, 因为它们是全局变量。OSTCBCur 指向被切换出去的任务的 OS\_TCB, 而 OSTCBHighRdy 指向新任务的 OS\_TCB。注意在调用 OSTaskSwHook() 期间中断一直是被禁止的。因为代码的多少会影响到中断的响应时间, 所以用户应尽量使代码简化。OSTaskSwHook() 没有任何参数, 也不返回任何值。

只用当 OS\_CFG.H 中的 OS\_CPU\_HOOKS\_EN 被置为 1 时才会产生 OSTaskSwHook() 的代码。

#### **8.05.05 OSTaskStatHook()**

OSTaskStatHook() 每秒钟都会被 OSTaskStat() 调用一次。用户可以用 OSTaskStatHook() 来扩展统计功能。例如, 用户可以保持并显示每个任务的执行时间, 每个任务所用的 CPU 份额, 以及每个任务执行的频率等等。OSTaskStatHook() 没有任何参数, 也不返回任何值。

只用当 OS\_CFG.H 中的 OS\_CPU\_HOOKS\_EN 被置为 1 时才会产生 OSTaskStatHook() 的代码。

#### **8.05.06 OSTimeTickHook()**

OSTimeTickHook() 在每个时钟节拍都会被 OSTaskTick() 调用。实际上, OSTimeTickHook() 是在节拍被  $\mu$ C/OS-II 真正处理, 并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。

只用当 OS\_CFG.H 中的 OS\_CPU\_HOOKS\_EN 被置为 1 时才会产生 OSTimeTickHook() 的代码。

## OSTaskCreateHook()

**void OSTaskCreateHook(OS\_TCB \*ptcb)**

<i>File</i>	<i>Called from</i>	<i>Code enabled by</i>
OS_CPU_C.C	OSTaskCreate() and OSTaskCreateExt()	OS_CPU_HOOKS_EN

无论何时建立任务，在分配好和初始化 TCB 后就会调用该函数，当然任务的堆栈结构也已经初始化好了。OSTaskCreateHook() 允许用户用自己的方式来扩展任务建立函数的功能。例如用户可以初始化和存储与任务相关的浮点寄存器，MMU 寄存器以及其它寄存器的内容。通常，用户可以存储用户的应用程序所分配的附加的内存信息。用户还可以通过使用 OSTaskCreateHook() 来触发示波器或逻辑分析仪，以及设置断点。

### 参数

ptcb 是指向所创建任务的任务控制块的指针。

### 返回值

无

### 注意事项

该函数在被调用的时候中断是禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。

### 范例

该例子假定了用户是用 OSTaskCreateExt() 建立任务的，因为它希望在任务 OS\_TCB 中有 OSTCBExtPtr 域，该域包含了指向浮点寄存器的指针。

```
Void OSTaskCreateHook (OS_TCB *ptcb)
{
    if (ptcb->OSTCBExtPtr != (void *)0) {
        /* 储存浮点寄存器的内容到.. */
        /* ..TCB扩展域中                      */
    }
}
```



# OSTaskDelHook()

void OSTaskDelHook(OS\_TCB \*ptcb)

<i>File</i>	<i>Called from</i>	<i>Code enabled by</i>
OS_CPU_C.C	OSTaskDel()	OS_CPU_HOOKS_EN

当用户通过调用 OSTaskDel() 来删除任务时都会调用该函数。这样用户就可以处理 OSTaskCreateHook() 所分配的内存。OSTaskDelHook() 就在 TCB 从 TCB 链中被移除前被调用。用户还可以通过使用 OSTaskDelHook() 来触发示波器或逻辑分析仪，以及设置断点。

## 参数

ptcb 是指向所创建任务的任务控制块的指针。

## 返回值

无

## 注意事项

该函数在被调用的时候中断是禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。

## 范例

```
void OSTaskDelHook (OS_TCB *ptcb)
{
    /* 输出信号触发示波器          */
}
```

# OSTaskSwHook()

void OSTaskSwHook(void)

<i>File</i>	<i>Called from</i>	<i>Code enabled by</i>
OS_CPU_C.C	OSCtxSw() and OSIntCtxSw()	OS_CPU_HOOKS_EN

当执行任务切换时都会调用该函数。全局变量 OSTCBHighRdy 指向得到 CPU 的任务的 TCB，而 OSTCBCur 指向被切换出去的任务的 TCB。OSTaskSwHook() 在保存好了任务的寄存器和保存好了指向当前任务 TCB 的堆栈指针后马上被调用。用户可以用该函数来保存或恢复浮点寄存器或 MMU 寄存器的内容，来得到任务执行时间的轨迹以及任务被切换进来的次数等等。

## 参数

无

## 返回值

无

## 注意事项

该函数在被调用的时候中断是禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。

## 范例

```
void OSTaskSwHook (void)
{
    /* 将浮点寄存器的内容储存在当前任务的TCB扩展域中。 */
    /* 用新任务的TCB扩展域中的值更新浮点寄存器的内容。 */
}
```

# OSTaskStatHook()

void OSTaskStatHook(void)

<i>File</i>	<i>Called from</i>	<i>Code enabled by</i>
OS_CPU_C.C	OSTaskStat()	OS_CPU_HOOKS_EN

该函数每秒钟都会被 $\mu$ C/OS-II 的统计任务调用。OSTaskStatHook() 允许用户加入自己的统计功能。

## 参数

无

## 返回值

无

## 注意事项

统计任务大概在调用 OSStart() 后再过 5 秒开始执行。注意，当 OS\_TASK\_STAT\_EN 或者 OS\_TASK\_CREATE\_EXT\_EN 被置为 0 时，该函数不会被调用。

## 范例

```
void OSTaskStatHook (void)
{
    /* 计算所有任务执行的总时间      */
    /* 计算每个任务的执行时间在总时间内所占的百分比      */
}
```

# OSTimeTickHook()

void OSTimeTickHook(void)

<i>File</i>	<i>Called from</i>	<i>Code enabled by</i>
OS_CPU_C.C	OSTimeTick()	OS_CPU_HOOKS_EN

只要发生时钟节拍，该函数就会被 OSTimeTick() 调用。一旦进入 OSTimeTick() 就会马上调用 OSTimeTickHook() 以允许执行用户的应用程序中的与时间密切相关的代码。用户还可以通过使用该函数触发示波器或逻辑分析仪来调试，或者为仿真器设置断点。

## 参数

无

## 返回值

无

## 注意事项

OSTimeTick() 通常是被 ISR 调用的，所以时钟节拍 ISR 的执行时间会因为用户在该函数中提供的代码而增加。当 OSTimeTick() 被调用的时候，中断可以是禁止的也可以是允许的，这主要取决于该处理器上的移植是怎样进行的。如果中断是禁止的，该函数将会影响到中断响应时间。

## 范例

```
void OSTimeTickHook (void)
{
    /* 触发示波器          */
}
```