uC/OS-II 实时操作系统在嵌入式平台进行移植的一般方法和技巧

作者：清华大学 曾鸣

引言

实时操作系统的使用，能够简化嵌入式系统的应用开发，有效地确保稳定性和可靠性，便于维护和二次开发。

μC/OS-II 是一个基于抢占式的实时多任务内核，可固化、可剪裁、具有高稳定性和可靠性，除此以外，μC/OS-II 的鲜明特点就是源码公开，便于移植和维护。

在 μC/OS-II 官方的主页上可以查找到一个比较全面的移植范例列表。但是，在实际的开发项目中，仍然没有针对项目所采用芯片或开发工具的合适版本。那么，不妨自己根据需要进行移植。

本文则以在 TMS320C6711 DSP 上的移植过程为例，分析了 μC/OS-II 在嵌入式开发平台上进行移植的一般方法和技巧。μC/OS-II 移植的基本步骤

在选定了系统平台和开发工具之后，进行 μC/OS-II 的移植工作，一般需要遵循以下的几个步骤：

- 深入了解所采用的系统核心
- 分析所采用的 C 语言开发工具的特点
- 编写移植代码
- 进行移植的测试
- 针对项目的开发平台，封装服务函数

（类似 80x86 版本的 PC.C 和 PC.H）

系统核心

无论项目所采用的系统核心是 MCU、DSP、MPU，进行 μC/OS-II 的移植时，所需要关注的细节都是相近的。

首先，是芯片的中断处理机制，如何开启、屏蔽中断，可否保存前一次中断状态等。还有，芯片是否有软中断或是陷阱指令，又是如何触发的。

此外，还需关注系统对于存储器的使用机制，诸如内存的地址空间，堆栈的增长方向，有无批量压栈的指令等。

在 本例中，使用的是 TMS320C6711 DSP。这是 TI 公司 6000 系列中的一款浮点型号，由于其时钟频率非常高，且采用了超常指令字（VLIW）结构、类 RISC 指令集、多级流水等技术，所 以运算性能相当强大，在通信设备、图像处理、医疗仪器等方面都有着广泛的应用。

在 C6711 中，中断有 3 种类型，即复位、不可屏蔽中 断（NMI）和可屏蔽中断（INT4-INT15）。可屏蔽中断由 CSR 寄存器控制全局使能，此外也可用 IER 寄存器分别置位使能。而在 C6711 中并没 有软中断机制，所以 μC/OS-II 的任务切换需要编写一个专门的函数实现。

此外，C6711 也没有专门的中断返回指令、批量压栈指令，所以相应的任务切换代码均需编程完成。由于采用了类 RISC 核心，C6711 的内核结构中，只有 A0-A15 和 B0-B15 这两组 32bit 的通用寄存器。

C 语言开发工具

无论所使用的系统核心是什么，C 语言开发工具对于 μC/OS-II 是必不可少的。

最简单的信息可以从开发工具的手册中查找，比如：C 语言各种数据类型分别编译为多少字节；是否支持嵌入式汇编，格式要求怎样；是否支持“interrupt”非标准关键字声明的中断函数；是否支持汇编代码列表(list)功能，等等。

上述的这样一些特性，会给嵌入式的开发带来很多便利。TI 的 C 语言开发工具 CCS for C6000 就包含上述的所有功能。

而在此基础上，可以进一步地弄清开发工具的一些技术细节，以便进行之后真正的移植工作。

首先，开启 C 编译器的“汇编代码列表(list)”功能，这样编译器就会为每个 C 语言源文件生成其对应的汇编代码文件。

在 CCS 开发环境中的方法是：在菜单“/Project/Build options”的“Feedback”栏中选择“Interlisting：Opt/C and ASM(-s)”；或者，也可以直接在 CCS 的 C 编译命令行中加上“-s”参数。

然后分别编写几个简单的函数进行编译，比较 C 源代码和编译生成的汇编代码。例如：

```
void FUNC_TEMP (void)
{
Func_tmp2(); //调用任一个函数
}
```

在 CCS 中编译后生成的 ASM 代码为：

```
.asg B15, SP //  宏定义
_FUNC_TEMP:
STW B3,*SP--(8) //  入栈
NOP 2
CALL _ Func_tmp2 //-----------
MVKL BACK, B3 //  函数调用
MVKH BACK, B3 //-----------
NOP 3
BACK: LDW *++SP(8),B3 //  出栈
NOP 4
RET B3 //  函数返回
NOP 5
```

由此可见，在 CCS 编译器的规则中，B15 寄存器被用作堆栈指针，使用通用存取指令进行栈操作，而且堆栈指针必须以 8 字节为单位改变。

此外，B3 寄存器被用来保存函数调用时的返回地址，在函数执行之前需要入栈保护，直到函数返回前再出栈。

当然，CCS 的 C 编译器对于每个通用寄存器都有约定的用途，但对于 μC/OS-II 的移植来说，了解以上信息就足够了。

最后，再编写一个用“interrupt”关键字声明的函数：

```
interrupt void ISR_TEMP (void)
{
int a;
a=0;
}
```

生成的 ASM 代码为：

```
_ISR_TEMP:
STW B4,*SP--(8) //  入栈
NOP 2
ZERO B4 //---------
STW B4,*+SP(4) // a=0
NOP 2 //----------
B IRP //  中断返回
LDW *++SP(8),B4 //  出栈
NOP 4
```

与前一段代码相比，对于中断函数的编译，有两点不同：

- 函数的返回地址不再使用 B3 寄存器，相应地也无需将 B3 入栈。（IRP 寄存器能自动保存中断发生时的程序地址）
- 编译器会自动统计中断函数所用到的寄存器，从而在中断一开始将他们全部入栈保护——例如上述程序段中，只用到了 B4 寄存器。

编写移植代码

在深入了解了系统核心与开发工具的基础上，真正编写移植代码的工作就相对比较简单了。

μC/OS-II 自身的代码绝大部分都是用 ANSI C 编写的，而且代码的层次结构十分干净，与平台相关的移植代码仅仅存在于 OS_CPU_A.ASM、OS_CPU_C.C 以及 OS_CPU.H 这三个文件当中。

在移植的时候，结合前面两个步骤中已经掌握的信息，基本上按照《嵌入式实时操作系统 μC/OS-II》一书的相关章节的指导来做就可以了。

但是，由于系统核心、开发工具的千差万别，在实际项目中，一般都会有一些处理方法上的不同，需要特别注意。以 C6711 的移植为例：

- 中断的开启和屏蔽的两个宏定义为：

```
#define OS_ENTER_CRITICAL() Disable_int()
#define OS_EXIT_CRITICAL() Enable_int()
```

Disable_int 和 Enable_int 是用汇编语言编写的两个函数。在这里使用了控制状态寄存器(CSR)的一个特性——CSR 中除了控制全局中断的 GIE 位之外，还有一个 PGIE 位，可用于保存之前的 GIE 状态。

因此在 Disable_int 中先将 GIE 的值写入 PGIE，然后再将 GIE 写 0，屏蔽中断。而在 Enable_int 中则从 PGIE 读出值，写入 GIE，从而回复到之前的中断设置。

这样，就可以避免使用这两个宏而意外改变了系统的中断状态——此外，也没有使用堆栈或局部变量，比原作者推荐的方法要好。

- 任务的切换：

前文说过，C6711 中没有软中断机制，所以任务的切换需要用汇编语言自行编写一个函数_OSCtxSw 来实现，并且

#define OS_TASK_SW() OSCtxSw()

在 C6711 中需要入栈保护的寄存器包括 A0-A15、B0-B15、CSR、IER、IRP 和 AMR，这些再加上当前的程序地址构成一个存储帧，需要入栈保存。

_OSCtxSw 函数中，需要像发生了一次中断那样，将上述存储帧入栈，然后获取被激活任务的 TCB 指针，将其存储帧的内容弹出，从而完成任务切换。

需要特别注意的是，在这里 OS_TASK_SW 是作为函数调用的，所以如前文所述，调用时的当前程序地址是保存在 B3 寄存器中的，这也就是任务重新激活时的返回地址。

- 中断的编写：

如前文所述，如果用“interrupt”关键字声明函数，CCS 在编译时，会自动将该函数中使用到的寄存器入栈、出栈保护。

但是，这会导致各种中断发生时，出入栈的内容各不相同。这对于 μC/OS-II 是会引起严重错误的。因为 μC/OS-II 要求中断发生时的入栈操作使用和发生任

务切换时完全一样的存储帧结构。

因此，在移植时、基于 μC/OS-II 进行开发时，都不应当使用“interrupt”关键字，而应用如下结构编写中断函数：

```
void OSTickISR (void)
{
DSP_C6x_Save(); // 服务函数，入栈
OSIntEnter();
if (OSIntNesting == 1) // v2.51 版本新增加
{
OSTCBCur->OSTCBStkPtr
=(OS_STK*) DSP_C6x_GetCurrentSP(); // 服务函数
} // 获取当前 SP 的值
// 允许中断嵌套 则在此处开中断
OSTimeTick();
OSIntExit();
DSP_C6x_Resume(); // 服务函数，出栈
}
```

DSP_C6x_Save 和 DSP_C6x_Resume 是两个服务函数，分别完成中断的出、入栈操作。它们与 OS_TASK_SW 函数的区别在于：中断发生时的当前程序地址是自 动保存在 IRP 寄存器的，应将其作为任务返回地址，而不再是 B3。此外，DSP_C6x_Resume 是一个永远不会返回的函数，在将所有内容出栈后，它 就直接跳转回到中断发生前的程序地址处，继续执行。

进行移植的测试

在编写完了所有的移植代码之后，就可以编写几个简单的任务程序进行测试了，大体上可以分三个步骤来进行，相关资料比较详尽，这里就不多作赘述了。

封装服务函数

最后这个步骤，往往是容易被忽视的，但对于保持项目代码的简洁、易维护有很重要的意义。

μC/OS-II 的原作者强烈建议将源代码分路径进行存储，例如本文例子中的所有源代码就应按如下路径结构存储：

```
uCOS-II
├─SOURCE // 平台无关代码
│  OS_CORE.C
│  ......
└─TI_C6711 // 系统核心
├─CCS // 开发工具
│  OS_CPU.H
│  OS_CPU_A.ASM
│  OS_CPU_C.C
│
├─ DSP_C6x_Service // 服务函数
│  DSP_C6x_ Service.H
│  DSP_C6x_ Service.ASM
│
└─ TEST // 具体的开发项目代码
```

OS_CFG.H
INCLUDES.H
TEST.C .....

如上，DSP_C6x_Service 中的服务函数，类似于原作者提供的 80x86 版本中的 PC.C 和 PC.H 文件。在本文的例子中，服务函数则包括了上文 提及的中断相关函数，以及系统初始化函数 DSP_C6x_SystemInit()和时钟初始化函数 DSP_C6x_TimerInit()等。

而具体的开发项目代码，则可以分别在“/TI_C6711”路径下新建自己的目录，就如同移植测试的“TEST”项目，而无需再关注 μC/OS-II 的源代码和服务函数。如此，就可以避免不必要的编译错误，也便于开发项目的维护。