Micriμm

μC/OS-II

and

ARM Processors

(For ARM7 or ARM9)
(For ARM and Thumb Mode)

Application Note

AN-1014 Rev. F

www.Micrium.com

Table of Contents

1.00 Introduction

µC/OS-II has been running on ARM based processors since 1995 (in fact **µC/OS** V1.x has). There has been a number of ARM ports posted on the Micriμm web site. The differences have mostly to do with differences in compilers and what target board they run on.

This application note describes the 'official' Micrium port for **µC/OS-II**. Figure 1-1 shows a block diagram showing the relationship between your application, **µC/OS-II**, the port code and the BSP (Board Support Package). Relevant sections of this application note are referenced on the figure.

Note that the port described in this application note applies to both ARM7 and ARM9 processors and you can use this port for both ARM and Thumb-based applications. Previous ports either worked in ARM-mode or in Thumb-mode. This port handles both.

The ARM port now supports IRQ nesting. This is actually an optional feature; interrupt nesting must be explicitly enabled in your exception handler by calling `OS_CPU_SR_INT_En()` or `OS_CPU_SR_IRQ_En()` after clearing the interrupt source. If this is not done, IRQs will not nest. This new port (v1.83) is almost fully backward compatible with the older version (v1.82); however, it may require one slight change. For technical reasons that will be explained in this document, the exceptions must now be handled in SVC mode, rather than in the mode of the exception (for example, IRQs were previously handled in IRQ mode). A dedicated exception stack is used for handling exceptions. The size of this stack, `OS_CPU_EXCEPT_STK_SIZE`, defaults to 128 4-byte entries; depending on your application, you may need to increase or decrease this size. As a consequence of this change, the size of the IRQ, FIQ, data abort, undefined instruction and prefetch abort stacks (often defined in the linker file and assigned to the stack pointers in startup code) need only be 16 bytes each, because these will ONLY be used to store the exception context at the beginning of exception handling.

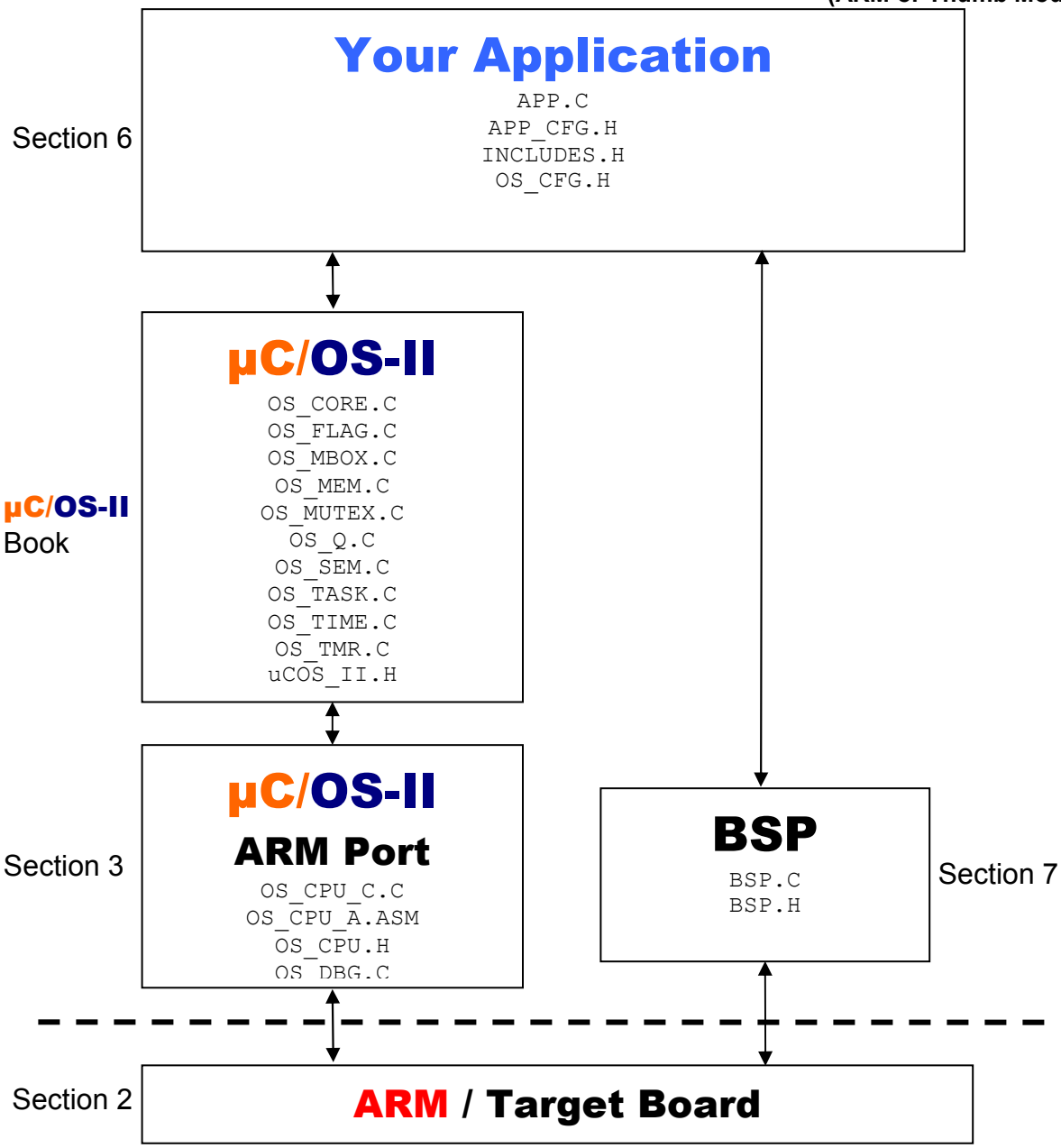

Figure 1-1, Relationship between modules.

2.00 The ARM programmer's model

Some of the most popular variant of the ARM processors are the ARM7TDMI and ARM92xT. The four letters stand for:

T (Thumb)
The T stands for *Thumb* instruction set which addresses the issue of code density. Specifically, Thumb mode allows instructions to be 16-bits instead of 32-bits thus reducing code density. A processor having the T suffix can thus run Thumb code.

D (Debug)
The D stands for debug support. This means that the specific ARM7 you are using offers on-chip debug support, generally through a J-Tag interface.

M (Multiply)
The M means that the CPU contains a hardware multiply instruction.

I (EmbeddedICE macrocell)
Is the debug hardware built into the processor that allows breakpoints and watchpoints to be set.

The visible registers in an ARM processor are shown in Figure 2-1. The ARM has a total of 37 registers. Each register is 32 bits wide. At any time, only 18 of those registers are directly 'visible' by the processor: `R0` through `R15`, `CPSR` and `SPSR` (`SPSR` is not visible in SYS mode).

`R0-R12` `R0` through `R12` are general purpose registers that can be used to hold data as well as pointers.

`R13` Is generally designated as the stack pointer (also called the `SP`) but could be the recipient of arithmetic operations.

`R14` Is called the Link Register (`LR`) and is used to store the contents of the `PC` when a Branch and Link (`BL`) instruction is executed. The `LR` allows you to return to the caller. The `LR` is also used during exception processing to store the contents of the `PC` prior to the exception.

`R15` Is dedicated to be used as the Program Counter (`PC`) and points to the current instruction being executed. As instructions are executed, the `PC` is incremented by either 2 (Thumb mode) or 4 (ARM mode).

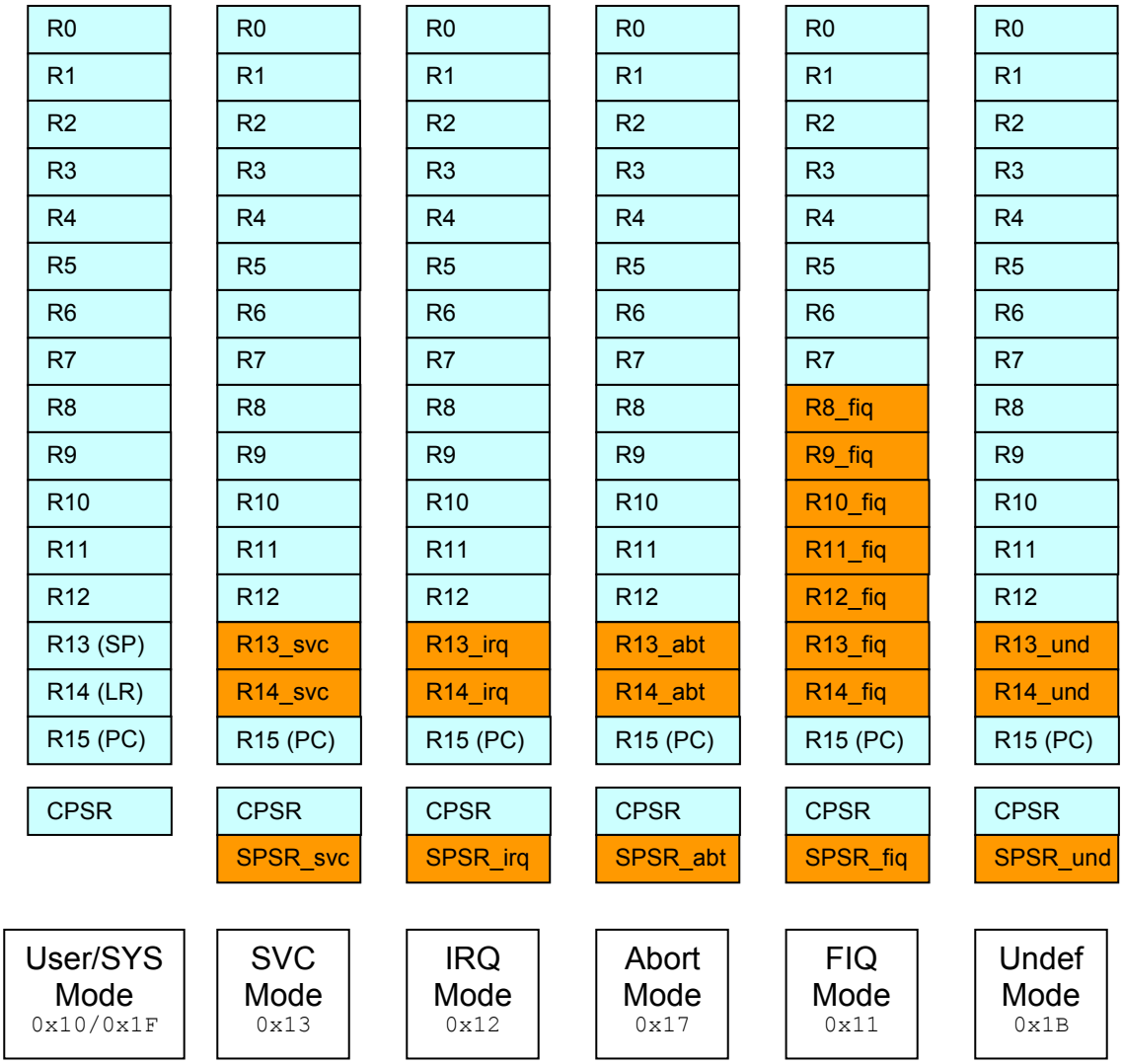

Figure 2-1, ARM Register Model.

CPSR

The CPSR (Current Processor Status Register) is used to store the condition code bits. These bits are used, for example, to record the result of a comparison operation and to control whether or not a conditional branch is taken. Figure 2-2 shows the contents of the CPSR.

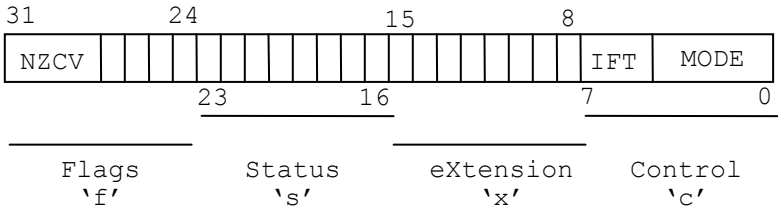

Figure 2-2, The CPSR Register.

MODE

The bottom 5 bits of the register control the processor mode (described later).

T
Bit 5 determines whether the processor is executing Thumb (T == 1) or ARM code (T == 0).

F
Bit 6 is the FIQ (Fast Interrupt Request) interrupt enable flag. Interrupts are recognized on the FIQ input of the processor when this bit is 0. Interrupts are disabled when it's a 1.

I
Bit 7 is the IRQ (Interrupt Request) interrupt enable flag. Interrupts are recognized when the bit is 0 and ignored when it's a 1.

N
Bit 31 is the 'negative' bit and is set when the last ALU operation produced a negative result (i.e. the top bit of a 32-bit result was a one).

Z
Bit 30 is the 'zero' bit and is set when the last ALU operation produced a zero result (every bit of the 32-bit result was zero).

C
Bit 29 is the 'carry' bit and is set when the last ALU operation generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifter.

V
Bit 28 is the 'overflow' bit and is set when the last arithmetic ALU operation generated an overflow into the sign bit.

The CPU can be in any of 7 modes: USER, SYS, SVC, IRQ, FIQ, ABORT and UNDEF (see Figure 2-1).

USER The USER mode is the least 'privileged' mode and in fact, certain instructions cannot be executed when in this mode. For this reason, **µC/OS-II** applications will never be in this mode. Only registers `R0-R15` and `CPSR` are 'visible' by the processor in this mode.

SYS The SYS mode uses the same registers as in USER mode except that code running in SYS mode has all the privileges of the other modes. Only registers `R0-R15` and `CPSR` are 'visible' by the processor in this mode.

SVC The SVC (Supervisor) mode is the default mode at power up. The processor can execute any instruction in this mode. In this mode, register `R13` and `R14` are not visible. Instead, alternate registers replace `R13` and `R14` and these are called `R13_svc` and `R14_svc`. In other words, only the registers in the SVC column of Figure 2-1 are visible. We decided to run the µC/OS-II port in SVC mode. The reason for choosing this will become apparent as we describe the port.

IRQ

When the I-bit of the CPSR is 0, the CPU will recognize interrupt requests from the IRQ input of the processor. When an interrupt occurs, the CPU does the following:

Switches mode to IRQ mode (MODE = 0x12)
Saves the CPSR into the SPSR_irq register
Saves the PC into R14_irq (i.e. the Link Register of the IRQ mode)
The I-bit of the CPSR is set to 1 disabling further IRQs
The PC is forced to address 0x00000018

Note that registers R0-R12 are the same as SYS mode except that the IRQ mode has its own set of R13_irq (the SP), R14_irq (the LR) and SPSR_irq registers. In fact, when an interrupts occurs, the CPSR of the SVC mode is saved in the SPSR_irq.

FIQ

When the F-bit of the CPSR is 0, the CPU will recognize interrupt requests from the FIQ input of the processor. When an interrupt occurs, the CPU does the following:

Switches mode to FIQ mode (MODE = 0x11)
Saves the CPSR into the SPSR_fiq register
Saves the PC into R14_fiq (i.e. the Link Register of the FIQ mode)
The F-bit and the I-bit of the CPSR are both set to 1 disabling further FIQs and IRQs
The PC is forced to address 0x0000001C

Note that registers R0-R7 are the same as SYS mode except that the FIQ mode has its own set of R8_fiq to R12_fiq and R13_fiq (the SP), R14_fiq (the LR) and SPSR_fiq registers. In fact, when an interrupts occurs, the CPSR of the current mode is saved in the SPSR_fiq.

ABORT

A memory abort is signaled by the memory system. Activating an abort in response to an instruction fetch marks the fetched instruction as invalid. An abort will take place if the processor attempts to execute the invalid instruction.

Switches mode to ABORT mode (MODE = 0x17)
Saves the CPSR into the SPSR_abt register
Saves the PC into R14_abt (i.e. the Link Register of the ABORT mode)
The I-bit of the CPSR is set to disable IRQs
The PC is forced to address 0x0000000C

Activating an abort in response to a data access (Load or Store) marks the data as invalid. A data abort will result in the following actions:

Switches mode to ABORT mode (MODE = 0x17)
Saves the CPSR into the SPSR_abt register
Saves the PC into R14_abt (i.e. the Link Register of the ABORT mode)
The I-bit of the CPSR is set to disable IRQs
The PC is forced to address 0x00000010

UNDEF If ARM executes a coprocessor instruction, it waits for any external coprocessor to acknowledge that it can execute the instruction. If no coprocessor responds, an undefined instruction exception occurs.

Switches mode to UNDEF mode (MODE = `0x1B`)
Saves the `CPSR` into the `SPSR_und` register
Saves the `PC` into `R14_und` (i.e. the Link Register of the UNDEF mode)
The I-bit of the `CPSR` is set to disable IRQs
The `PC` is forced to address `0x00000004`

3.00 µC/OS-II Port for ARM processors

We used the IAR EWARM V4.40A (Embedded Workbench for the ARM) to test the port. The EWARM contains an editor, a C/EC++ compiler, an assembler, a linker/locator and the C-Spy debugger. The C-Spy debugger actually contains an ARM simulator which allows you to test code prior to run it on actual hardware. We tested the ARM port on a number of different ARM7 and ARM9 target processors.

You can adapt the port provided in this application note to other ARM based compilers. The instructions (i.e. the code) should be identical and all you have to do is adapt the port to your compiler specifics. We will describe some of these when we cover the contents of the different files.

IMPORTANT

The IAR compiler version that we used assumed that application code was running in SYS mode. In fact, the compiler calls `main()` in SYS mode. However, when we start **µC/OS-II**, we switch the mode to SVC mode and run all tasks in SVC mode.

Below are a few assumptions about the port:

- You have **µC/OS-II** V2.77 or higher
- **µC/OS-II** runs in either ARM mode or Thumb mode
- Tasks are created in the same mode as the one selected for running **µC/OS-II**
  - Tasks can call either ARM or Thumb mode functions
- Tasks will run in SVC mode

You can build the example code using either ARM (see figure 3-1) or Thumb (see figure 3-2) mode. Note that you need to enable 'Generate interwork code'. The screen shots are for the IAR's EWARM toolchain.

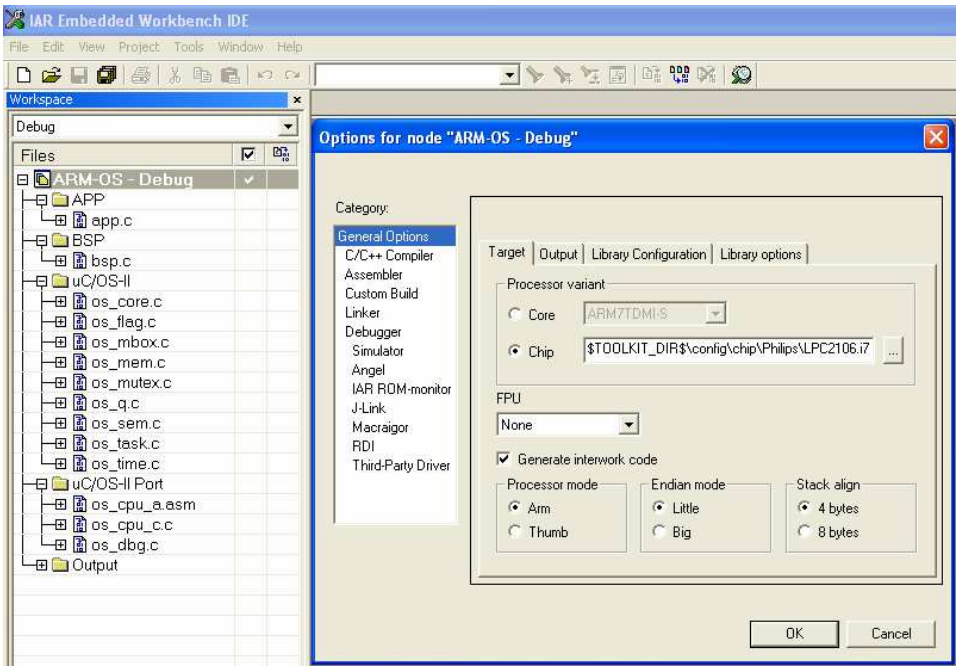

Figure 3-1, Building the example using ARM mode in IAR's EWARM.

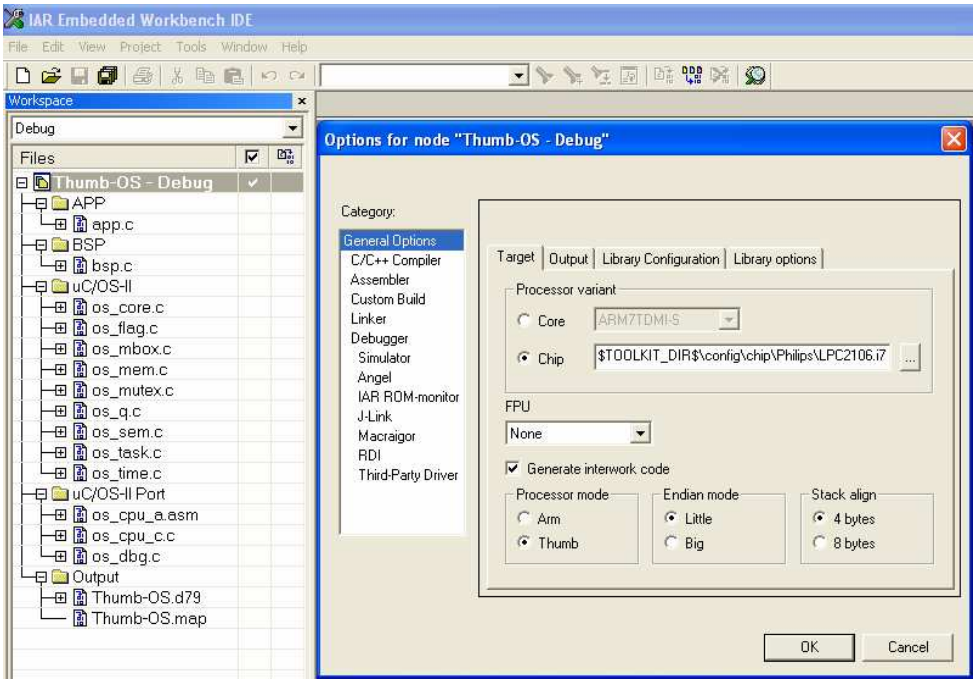

Figure 3-2, Building the example using Thumb mode in IAR's EWARM.

3.01 Directories and Files

The software that accompanies this application note is assumed to be placed in the following directory:

```
\Micrium\Software\uCOS-II\ARM\Generic\IAR
```

Like all **µC/OS-II** ports, the source code for the port is found in the following files:

```
OS_CPU.H
OS_CPU_C.C
OS_CPU_A.ASM
OS_DBG.C
```

Test code and configuration files are found in their appropriate directories and are described later.

3.02 OS_CPU.H

OS_CPU.H contains processor- and implementation-specific #defines constants, macros, and typedefs.

3.02.01 OS_CPU.H, macros for 'externals'

OS_CPU_GLOBALS and OS_CPU_EXT allows us to declare global variables that are specific to this port (described later).

Listing 3-1, OS_CPU.H, Globals and Externs

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

3.02.02 OS_CPU.H, Data Types

Listing 3-2, OS_CPU.H, Data Types

```
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;
typedef signed   char  INT8S;
typedef unsigned short INT16U;            // (1)
typedef signed   short INT16S;
typedef unsigned int   INT32U;
typedef signed   int   INT32S;
typedef float          FP32;              // (2)
typedef double         FP64;

typedef unsigned int   OS_STK;            // (3)
typedef unsigned int   OS_CPU_SR;         // (4)
```

L3-2(1) If you were to consult the IAR compiler documentation, you would find that an short is 16 bits and an int is 32 bits. Most ARM compilers should have the same definitions.

L3-2(2) Floating-point data types are included even though µC/OS-II doesn't make use of floating-point numbers.

L3-2(3) A stack entry for the ARM processor is always 32 bits wide; thus, OS_STK is declared accordingly. All task stacks must be declared using OS_STK as its data type.

L3-2(4) The status register (the CPSR and SPSR) on the ARM processor is 32 bits wide. The OS_CPU_SR data type is used when OS_CRITICAL_METHOD #3 is used (described below). In fact, this port only supports OS_CRITICAL_METHOD #3 because it's the preferred method for µC/OS-II ports.

3.02.03 OS_CPU.H, Critical Sections

µC/OS-II, as with all real-time kernels, needs to disable interrupts in order to access critical sections of code and re-enable interrupts when done. µC/OS-II defines two macros to disable and enable interrupts: `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`, respectively. µC/OS-II defines three ways to disable interrupts but, you only need to use one of the three methods for disabling and enabling interrupts. The book (MicroC/OS-II, The Real-Time Kernel) describes the three different methods. The one to choose depends on the processor and compiler. In most cases, the prefered method is `OS_CRITICAL_METHOD` #3.

`OS_CRITICAL_METHOD` #3 implements `OS_ENTER_CRITICAL()` by writing a function that will save the status register of the CPU in a variable. `OS_EXIT_CRITICAL()` invokes another function to restore the status register from the variable. In the book, Mr. Labrosse recommends that you call the functions expected in `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`: `OS_CPU_SR_Save()` and `OS_CPU_SR_Restore()`, respectively. The code for these two functions is declared in `OS_CPU_A.S` (described later).

Listing 3-3, OS_CPU.H, `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`

```
#define  OS_CRITICAL_METHOD    3


#if      OS_CRITICAL_METHOD == 3

#if      OS_CPU_INT_DIS_MEAS_EN > 0

#define  OS_ENTER_CRITICAL()  {cpu_sr = OS_CPU_SR_Save();   \
                               OS_CPU_IntDisMeasStart();}
#define  OS_EXIT_CRITICAL()   {OS_CPU_IntDisMeasStop();    \
                               OS_CPU_SR_Restore(cpu_sr);}

#else

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

#endif

#endif
```

3.02.04 OS_CPU.H, Stack growth

The stack on the ARM grows from high memory to low memory and thus, `OS_STK_GROWTH` is set to `1` to indicate this to µC/OS-II.

Listing 3-4, OS_CPU.H, Stack Growth

```
#define  OS_STK_GROWTH        1
```

3.02.05 OS_CPU.H, Exception Stack Size

The ARM port uses a special stack for all interrupts and exceptions. The size of this defaults to 128 4-byte entries; if you would like the stack to be a different size, just #define `OS_CPU_EXCEPT_STK_SIZE` to the desire size in **`app_cfg.h`**.

The function `OS_CPU_ExceptStkChk()` can be used to get the number of free `OS_STK`-wide entries in the exception stack. This just counts the number of zero entries at the bottom of the stack; since the stack entries are cleared upon initialization, this should indicate the maximum stack usage.

(The alternative to using this special stack is to use the stack of the interrupted task. Though this is a simpler solution, it would require that the stacks for all tasks account for stack usage of any interrupts that might occur. Since the ARM port now supports interrupt nesting, this could pose an arbitrary burden on task stacks.)

Listing 3-5, OS_CPU.H, Exception Stack Size

```
#ifndef  OS_CPU_EXCEPT_STK_SIZE
#define  OS_CPU_EXCEPT_STK_SIZE    128
#endif
```

Listing 3-6, OS_CPU.H, Exception Stack and Exception Stack Pointer

```
OS_CPU_EXT  OS_STK   OS_CPU_ExceptStk[OS_CPU_EXCEPT_STK_SIZE];
OS_CPU_EXT  OS_STK  *OS_CPU_ExceptStkBase;
OS_CPU_EXT  OS_STK  *OS_CPU_ExceptStkPtr;
```

Listing 3-7, OS_CPU.H, Exception Stack Size Checking

```
INT32U  OS_CPU_ExceptStkChk(void);
```

3.02.06 OS_CPU.H, Task Level Context Switch

Task level context switches are performed when **µC/OS-II** invokes the macro `OS_TASK_SW()`. Because context switching is processor specific, `OS_TASK_SW()` needs to execute an assembly language function. In this case, `OSCtxSw()` which is declared in `OS_CPU_A.ASM` (described later).

Listing 3-8, OS_CPU.H, Task Level Context Switch

```
#define  OS_TASK_SW()          OSCtxSw()
```

3.02.07 OS_CPU.H, Function Prototypes

The prototypes in Listing 3-9 are for the functions used to disable and re-enable interrupts using OS_CRITICAL_METHOD #3 and are described later. You should note that these prototypes are prefixed with the special keyword __arm. This is an IAR keyword that indicates that these functions will run in ARM mode and thus, when called, the compiler will generate the appropriate instructions.

Listing 3-9, OS_CPU.H, Function Prototypes

```
#if OS_CRITICAL_METHOD == 3
__arm  OS_CPU_SR  OS_CPU_SR_Save(void);
__arm  void       OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);
#endif
```

The prototypes in Listing 3-10 are the exception handling related functions. OS_CPU_InitExceptVect() must be called from the BSP to initialize the CPU exception vectors to the eight exception handlers. These eight exception handlers are the OS_CPU_ARM_XYZ assembly functions. These handlers save the CPU state and branch immediately to a common exception handler, OS_CPU_ARM_ExceptHndlr(). The common exception handler will do µC/OS-II internal task management (save state, etc) and will eventually call a board and application dependant exception handler, OS_CPU_ExceptHndlr(), located in BSP. Specifically, the __arm keyword indicates that these function will execute in ARM mode whether called from Thumb or ARM mode code.

Listing 3-10, OS_CPU.H, Function Prototypes

```
       void  OS_CPU_InitExceptVect(void);

__arm  void  OS_CPU_ARM_ExceptUndefInstrHndlr(void);
__arm  void  OS_CPU_ARM_ExceptSwiHndlr(void);
__arm  void  OS_CPU_ARM_ExceptPrefetchAbortHndlr(void);
__arm  void  OS_CPU_ARM_ExceptDataAbortHndlr(void);
__arm  void  OS_CPU_ARM_ExceptAddrAbortHndlr(void);
__arm  void  OS_CPU_ARM_ExceptIrqHndlr(void);
__arm  void  OS_CPU_ARM_ExceptFiqHndlr(void);

       void  OS_CPU_ExceptHndlr(INT32U  except_type);
```

As of V2.77, the prototypes for OSCtxSw(), OSIntCtxSw() and OSStartHighRdy() need to be placed in OS_CPU.H. In fact, it makes sense to do this since these are all port specific files. The reason we made the change is to allow for declarations as shown in Figure 3-11. Specifically, the __arm keyword indicates that these functions will execute in ARM mode whether called from Thumb or ARM mode code.

Listing 3-11, OS_CPU.H, Function Prototypes

```
__arm  void  OSCtxSw(void);
__arm  void  OSIntCtxSw(void);
__arm  void  OSStartHighRdy(void);
```

The prototypes in Listing 3-12 are for functions used to measure the interrupt disable time. Basically, we read the value of a timer just after disabling interrupts and read it again before enabling interrupts. The difference in timer counts indicates the amount of time interrupts were disabled. `OS_CPU_IntDisMeasStop()` actually keeps track of the highest value of this delta counts and thus, the maximum interrupt disable time. We'll describe this in greater details later.

Listing 3-12, OS_CPU.H, Function Prototypes

```
#if OS_CRITICAL_METHOD == 3
void    OS_CPU_IntDisMeasInit(void);
void    OS_CPU_IntDisMeasStart(void);
void    OS_CPU_IntDisMeasStop(void);
INT16U  OS_CPU_IntDisMeasTmrRd(void);
#endif
```

The prototypes in Listing 3-13 are for functions used to disable and enable interrupts so as to prevent or allow nesting. Once `OS_CPU_ExceptHndlr()` has cleared an IRQ source, it may call `OS_CPU_SR_IRQ_EN()`, `OS_CPU_SR_FIQ_EN()` or `OS_CPU_SR_INT_EN()` (which enables both IRQs and FIQs), to enable interrupts and allow interrupt nesting. This MUST be done after clearing the IRQ source; otherwise, the interrupt controller may re-generate the interrupt. **This should not be done for FIQs, which should generally not be nested.**

Listing 3-13, OS_CPU.H, Function Prototypes

```
__arm  void  OS_CPU_SR_INT_En(void);
__arm  void  OS_CPU_SR_INT_Dis(void);
__arm  void  OS_CPU_SR_IRQ_En(void);
__arm  void  OS_CPU_SR_IRQ_Dis(void);
__arm  void  OS_CPU_SR_FIQ_En(void);
__arm  void  OS_CPU_SR_FIQ_Dis(void);
```

3.03 OS_CPU_C.C

A µC/OS-II port requires that you write ten fairly simple C functions:

OSInitHookBegin()
OSInitHookEnd()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskIdleHook()
OSTaskStatHook()
OSTaskStkInit()
OSTaskSwHook()
OSTCBInitHook()
OSTimeTickHook()

Typically, µC/OS-II only requires `OSTaskStkInit()`. The other functions allow you to extend the functionality of the OS with your own functions. The functions that are highlighted will be discussed in this section. The following functions have been added in order to measure interrupt disable time and will be described later:

OS_CPU_IntDisMeasInit()
OS_CPU_IntDisMeasStart()
OS_CPU_IntDisMeasStop()

Note that you will also need to set the `#define` constant `OS_CPU_HOOKS_EN` to `1` in `OS_CFG.H` in order for the compiler to use the functions declared in this file.

3.03.01 OS_CPU_C.C, OSInitHookBegin()

This function is called by µC/OS-II's `OSInit()` at the very beginning of `OSInit()`. It gives the opportunity to add additional initialization code specific to the port. In this case, we initialize the global variable (global to `OS_CPU_C.C`) `OSTmrCtr` (which is used by the `OS_TMR.C` module (if `OS_TMR_EN` is set to `1`) and set the pointer to the top of the exception stack.

Listing 3-14, OS_CPU_C.C, `OSInitHookEnd()`

```
void OSInitHookBegin (void)
{
    INT32U   size;
    OS_STK  *pstk;


    pstk = &OS_CPU_ExceptStk[0];
    size = OS_CPU_EXCEPT_STK_SIZE;
    while (size > 0) {
        size--;
       *pstk = (OS_STK)0;
    }

#if OS_STK_GROWTH == 1
    OS_CPU_ExceptStkBase = &OS_CPU_ExceptStk[OS_CPU_ARM_EXCEPT_STK_SIZE - 1];
#else
    OS_CPU_ExceptStkBase = &OS_CPU_ExceptStk[0];
#endif

#if OS_TMR_EN > 0
    OSTmrCtr = 0;
#endif
}
```

3.03.02 OS_CPU_C.C, OSInitHookEnd()

This function is called by µC/OS-II's `OSInit()` at the very end of `OSInit()`. It gives the opportunity to add additional initialization code specific to the port. In this case, we initialize global variables which are used by the interrupt disable measurement code (if `OS_CPU_INT_DIS_MEAS_EN` is set to `1`).

Listing 3-15, OS_CPU_C.C, `OSInitHookEnd()`

```
void OSInitHookEnd (void)
{
#if OS_CPU_INT_DIS_MEAS_EN > 0
    OS_CPU_IntDisMeasInit();
#endif
}
```

3.03.03 OS_CPU_C.C, OSTaskCreateHook()

This function is called by **µC/OS-II**'s `OSTaskCreate()` or `OSTaskCreateExt()` when a task is created. `OSTaskCreateHook()` gives the opportunity to add code specific to the port when a task is created. In this case, we call the application task create hook, `App_TaskCreateHook()`. This may be used to call the task create hook for the **µC/OS-II** plug-in for **µC/Probe** (a Windows program which can monitor an embedded application at run-time; see www.micrium.com for details).

Note that for `App_TaskCreateHook()` to be called, you need to have defined this function and have configured `#define OS_APP_HOOKS_EN 1` in `OS_CFG.H` of your application.

Note that if `OS_APP_HOOKS_EN` is `0`, we simply tell the compiler that `ptcb` is not actually used (i.e. `(void)ptcb`) and thus avoid a compiler warning.

Listing 3-16, OS_CPU_C.C, `OSInitHookEnd()`

```
void OSTaskCreateHook (OS_TCB *ptcb)
{
#if OS_APP_HOOKS_EN > 0
    App_TaskCreateHook(ptcb);
#else
    (void)ptcb;
#endif }
```

3.03.04 OS_CPU_C.C, OSTaskStkInit()

µC/OS-II assumes that tasks run in SVC mode (the `CPSR` of the task is initialized to `ARM_SVC_MODE` (`0x13` if in ARM mode or `0x33` if in Thumb mode).

It is typical for ARM compilers to pass the first argument of a function into the `R0` register. Recall that a task is declared as shown in listing 3-17.

Listing 3-17, µC/OS-II Task

```
void  MyTask (void *p_arg)
{
    /* Do something with 'p_arg', optional */
    while (1) {
        /* Task body */
    }
}
```

The code in Listing 3-18 initializes the stack frame for the task being created. The task received an optional argument '`p_arg`'. That's why '`p_arg`' is passed in `R0` when the task is created. The initial value of most of the CPU registers is not important so, we decided to initialize them to values corresponding to their register number. This makes it convenient when debugging and examining stacks in RAM. The initial values are thus useful when the task is first created but, of course, the register values will most likely change as the task code is executed.

Listing 3-18, OS_CPU_C.C, `OSTaskStkInit()`

```
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *p_arg, OS_STK *ptos, INT16U opt)
{
    OS_STK *stk;
    INT32U  task_addr;

    opt       = opt;                        /* 'opt' is not used, prevent warning      */
    stk       = ptos;                       /* Load stack pointer                      */
    task_addr = (INT32U)task & ~1;
    *(stk)    = (INT32U)task_addr;          /* Entry Point                             */
    *(--stk)  = (INT32U)0x14141414L;        /* R14 (LR)                                */
    *(--stk)  = (INT32U)0x12121212L;        /* R12                                     */
    *(--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                                      */
    *(--stk)  = (INT32U)0x03030303L;        /* R3                                      */
    *(--stk)  = (INT32U)0x02020202L;        /* R2                                      */
    *(--stk)  = (INT32U)0x01010101L;        /* R1                                      */
    *(--stk)  = (INT32U)p_arg;              /* R0 : argument                           */
    if ((INT32U)task & 0x01) {              /* See if task runs in Thumb or ARM mode   */
        *(--stk) = (INT32U)ARM_SVC_MODE_THUMB;  /* CPSR, THUMB-mode                    */
    } else {
        *(--stk) = (INT32U)ARM_SVC_MODE_ARM;    /* CPSR, ARM-mode                      */
    }

    return (stk);
}
```

Figure 3-2 shows how the stack frame is initialized for each task when it's created.

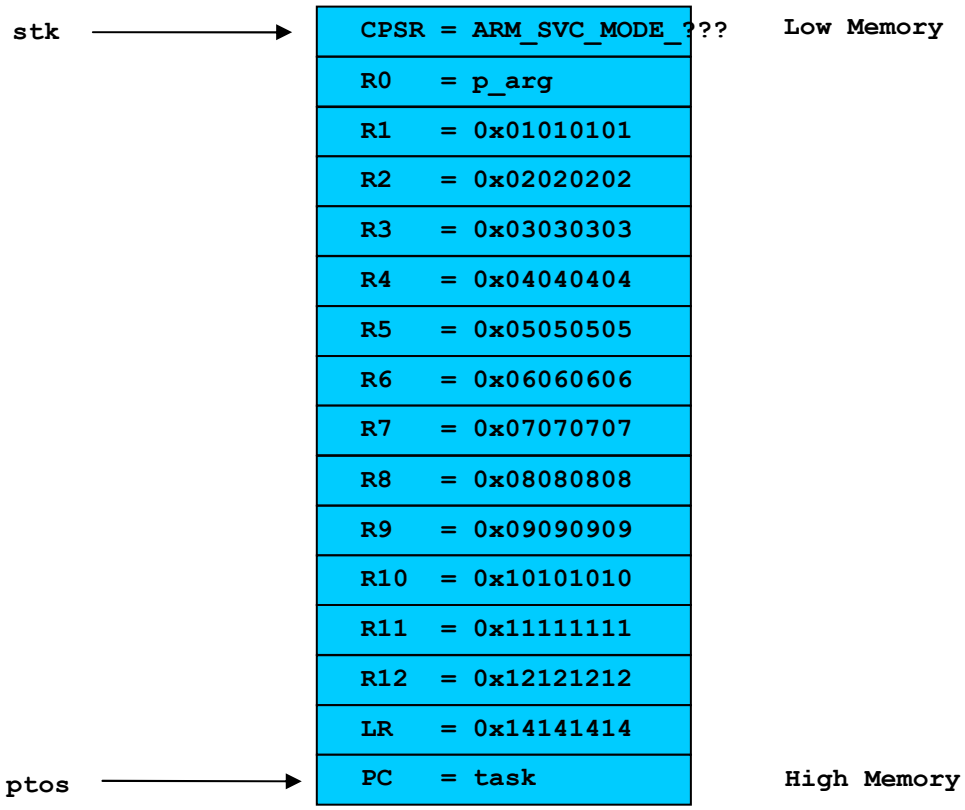

Figure 3-3, The Stack Frame for each Task for ARM port.

When the task is created, the final value of `stk` is placed in the `OS_TCB` of that task by the **µC/OS-II** function that calls `OSTaskStkInit()` (i.e. `OSTaskCreate()` or `OSTaskCreateExt()`).

3.03.05 OS_CPU_C.C, OSTaskSwHook()

`OSTaskSwHook()` is called when a context switch occurs. This function allows the port code to be extended and do things such as measuring the execution time of a task, output a pulse on a port pin when a contact switch occurs, etc. In this case, we call the application task switch hook, `App_TaskSwHook()`. This may be used to call the task switch hook for the **µC/OS-II** plug-in for **µC/Probe**.

Listing 3-19, OS_CPU_C.C, OSTaskSwHook()

```
void  OSTaskSwHook (void)
{
#if OS_APP_HOOKS_EN > 0
    App_TaskSwHook();
#endif
}
```

3.03.06 OS_CPU_C.C, OSTimeTickHook()

`OSTimeTickHook()` is called at the very beginning of `OSTimeTick()`. This function allows the port code to be extended. In this case, we call the application task switch hook, `App_TimeTickHook()`. This may be used to call the time hook for the **µC/OS-II** plug-in for **µC/Probe**.

`OSTimeTickHook()` also determines whether it's time to update the **µC/OS-II** timers. This is done by signaling the timer task.

Listing 3-20, OS_CPU_C.C, OSTimeTickHook()

```
void  OSTimeTickHook (void)
{
#if OS_APP_HOOKS_EN > 0
    App_TimeTickHook();
#endif

#if OS_TMR_EN > 0
    OSTmrCtr++;
    if (OSTmrCtr >= (OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC)) {
        OSTmrCtr = 0;
        OSTmrSignal();
    }
#endif

#if OS_CPU_ARM_DCC_EN > 0
    OSDCC_Handler();
#endif
}
```

3.03.07 OS_CPU_C.C, OS_CPU_IntDisMeasInit()

`OS_CPU_IntDisMeasInit()` is called by `OSInitHookEnd()` (see section 3.03.01) to initialize the interrupt disable time measurement variables as shown below.

Basically, we added functions to the port to allow us to measure the amount of time that interrupts are disabled. This is not something that is needed by the port but it can provide valuable information about the responsiveness of your system to interrupts.

The way interrupt disable time measurement works is simple. Just after disabling interrupts, we read the contents of a free running 16-bit (or 32-bit) timer. Just before re-enabling interrupts, we read the free running counter again and compute the difference between the two readings. Maximum interrupt disable time is obtained by tracking the highest value of the difference. The value of the difference represents timer counts and thus, to convert to actual time, you need to know how fast the counter is being incremented (or decremented).

The function in listing 3-21 initializes the measurement and can actually be called at any time to 'reset' the maximum count.

Listing 3-21, OS_CPU_C.C, OS_CPU_IntDisMeasInit()

```
#if OS_CPU_INT_DIS_MEAS_EN > 0
void  OS_CPU_IntDisMeasInit (void)
{
    OS_CPU_IntDisMeasNestingCtr = 0;      /* Clear variables used by these functions */
    OS_CPU_IntDisMeasCntsEnter  = 0;
    OS_CPU_IntDisMeasCntsExit   = 0;
    OS_CPU_IntDisMeasCntsMax    = 0;
    OS_CPU_IntDisMeasCntsDelta  = 0;
    OS_CPU_IntDisMeasCntsOvrhd  = 0;
    OS_CPU_IntDisMeasStart();             /* Measure the overhead of the functions   */
    OS_CPU_IntDisMeasStop();
    OS_CPU_IntDisMeasCntsOvrhd  = OS_CPU_IntDisMeasCntsDelta;
}
#endif
```

3.03.08 OS_CPU_C.C, OS_CPU_IntDisMeasStart()

`OS_CPU_IntDisMeasStart()` is called when interrupts are disabled by `OS_ENTER_CRITICAL()`.

Listing 3-22, OS_CPU_C.C, OS_CPU_IntDisMeasStart()

```
#if OS_CPU_INT_DIS_MEAS_EN > 0
void  OS_CPU_IntDisMeasStart (void)
{
    OS_CPU_IntDisMeasNestingCtr++;                              (1)
    if (OS_CPU_IntDisMeasNestingCtr == 1) {                     (2)
        OS_CPU_IntDisMeasCntsEnter = OS_CPU_IntDisMeasTmrRd();
    }
}
#endif
```

L3-17(1) A nesting counter is maintained in case you nest `OS_ENTER_CRITICAL()` calls.

L3-17(2) If this is the first level of nesting for `OS_ENTER_CRITICAL()` then, we call a function that you would define in your application called `OS_CPU_IntDisMeasTmrRd()` to read the value of a 16-bit free-running timer. Note that you could also use a 32-bit timer. In this case, you would simply redeclare the variables and prototypes accordingly. The value of the timer is saved in `OS_CPU_IntDisMeasCntsEnter`.

3.03.09 OS_CPU_C.C, OS_CPU_IntDisMeasStop()

`OS_CPU_IntDisMeasStop()` is called when interrupts are re-enabled by `OS_EXIT_CRITICAL()`.

Listing 3-23, OS_CPU_C.C, OS_CPU_IntDisMeasStop()

```
#if OS_CPU_INT_DIS_MEAS_EN > 0
void  OS_CPU_IntDisMeasStop (void)
{
    OS_CPU_IntDisMeasNestingCtr--;                                          (1)
    if (OS_CPU_IntDisMeasNestingCtr == 0) {
        OS_CPU_IntDisMeasCntsExit  = OS_CPU_IntDisMeasTmrRd();
        OS_CPU_IntDisMeasCntsDelta = OS_CPU_IntDisMeasCntsExit              (2)
                                   - OS_CPU_IntDisMeasCntsEnter;
        if (OS_CPU_IntDisMeasCntsDelta > OS_CPU_IntDisMeasCntsOvrhd) {      (3)
            OS_CPU_IntDisMeasCntsDelta -= OS_CPU_IntDisMeasCntsOvrhd;
        } else {
            OS_CPU_IntDisMeasCntsDelta  = OS_CPU_IntDisMeasCntsOvrhd;
        }
        if (OS_CPU_IntDisMeasCntsDelta > OS_CPU_IntDisMeasCntsMax) {        (4)
            OS_CPU_IntDisMeasCntsMax = OS_CPU_IntDisMeasCntsDelta;
        }
     }

}
#endif
```

L3-18(1) The nesting counter is decremented so that we only take a time measurement at the last nested `OS_EXIT_CRITICAL()` calls.

L3-18(2) We measure the difference in timer value since interrupts were disabled.

L3-18(3) We make sure that the counts are higher than the measured overhead so we don't subtract a number that is larger than the delta. This would cause a 'large' count for the measured interrupt disable time.

L3-18(4) We record the highest value in `OS_CPU_IntDisMeasCntsMax`.

3.04 OS_CPU_A.ASM

A µC/OS-II port requires that you write five fairly simple assembly language functions. The ARM port actually contains fourteen functions because portions of the exception handling code are written in assembly language as discussed in this section. These functions are needed because you normally cannot save/restore registers from C functions. The fourteen functions are:

```
OS_CPU_SR_Save()
OS_CPU_SR_Restore()
OSStartHighRdy()
OSCtxSw()
OSIntCtxSw()

OS_CPU_InitExceptVect()
OS_CPU_ARM_ExceptUndefInstrHndlr()
OS_CPU_ARM_ExceptSwiHndlr()
OS_CPU_ARM_ExceptPrefetchAbortHndlr()
OS_CPU_ARM_ExceptDataAbortHndlr()
OS_CPU_ARM_ExceptAddrAbortHndlr()
OS_CPU_ARM_ExceptIrqHndlr()
OS_CPU_ARM_ExceptFiqHndlr()
```

3.04.01 OS_CPU_A.ASM, OS_CPU_SR_Save()

The code in listing 3-24 implements the saving of the `CPSR` register and then disabling interrupts for `OS_CRITICAL_METHOD` #3. The code follows the application note published by Atmel ("Disabling Interrupts at Processor Level") for properly disabling interrupts on the ARM. In this implementation, both the FIQ and IRQ interrupts are disabled.

You should note that we use the `BX LR` instruction to return to the appropriate mode. Specifically, if `OS_CPU_SR_Save()` was called from ARM mode code, `CPSR` bit `5` would stay at `0`. If we return to Thumb mode code then `CPSR` bit `5` will be set to `1` by the `BX` instruction.

When this function returns, `R0` contains the state of the `CPSR` register prior to disabling interrupts.

Listing 3-24, `OS_CPU_SR_Save()`

```
OS_CPU_SR_Save
    MRS     R0, CPSR
                                ; Set IRQ and FIQ bits in CPSR to disable all interrupts.
    ORR     R1, R0, #OS_CPU_ARM_CONTROL_INT_DIS
    MSR     CPSR_c, R1
    BX      LR                  ; Disabled, return the original CPSR contents in R0.
```

3.04.02 OS_CPU_A.ASM, OS_CPU_SR_Restore()

The code in the listing below implements the function to restore the CPSR register for OS_CRITICAL_METHOD #3. When called, it's assumed that R0 contains the desired state of the CPSR register. You should note that we only update the 'control' field of the CPSR (i.e. lower 8 bits of the CPSR).

Again, the BX LR instruction returns to the appropriate mode (ARM or Thumb).

Listing 3-25, OS_CPU_SR_Restore()

```
OS_CPU_SR_Restore
    MSR     CPSR_c, R0
    BX      LR
```

3.04.03 OS_CPU_A.ASM, OSStartHighRdy()

OSStartHighRdy() is called by OSStart() to start running the highest priority task that was created before calling OSStart(). OSStart() sets OSTCBHighRdy to point to the OS_TCB of the highest priority task.

Listing 3-26, OSStartHighRdy()

```
OSStartHighRdy

                                ; (1) Change to SVC mode.
    MSR     CPSR_c, #(OS_CPU_ARM_CONTROL_INT_DIS | OS_CPU_ARM_MODE_SVC)

    LDR     R0, ?OS_TaskSwHook  ; (2) OSTaskSwHook();
    MOV     LR, PC
    BX      R0

    LDR     R0, ?OS_Running     ; (3) OSRunning = TRUE;
    MOV     R1, #1
    STRB    R1, [R0]

                                ; SWITCH TO HIGHEST PRIORITY TASK.
    LDR     R0, ?OS_TCBHighRdy  ;     (4) Get highest priority task TCB address.
    LDR     R0, [R0]            ;     get stack pointer.
    LDR     SP, [R0]            ;     switch to the new stack.

    LDR     R0, [SP], #4        ;     (5) Prepare to return to proper mode …
    MSR     SPSR_cxsf, R0               … (ARM or Thumb)

    LDMFD   SP!, {R0-R12, LR, PC}^;   (6) pop new task's context.
```

L3-21(1) The IAR compiler startup code sets the mode to SYS mode prior to calling `main()`. We decided to use SVC mode for the **µC/OS-II** because it allows us to use the SPSR register to return to the proper mode (ARM or Thumb) as described in L3-21(7). Interrupts should not be enabled at this point but, just to make sure, we disable them.

L3-21(2) Before starting the highest priority task, we call `OSTaskSwHook()` in case a hook call has been declared. Note that we use a `BX` instruction because `OSTaskSwHook()` could be compiled in either ARM or Thumb mode. All ARM instructions are all 32 bits and thus, the ARM is not able to specify a 32-bit address as part of the instruction. Because of that, the address of `OSTaskSwHook()` is actually declared at the end of the file and the ARM obtains this address via a PC-relative address. Specifically:

```
?OS_TaskSwHook:
   DC32    OSTaskSwHook
```

`DC32` is an assembler directive that declares storage for a 32 bit constant that resides in code. `?OS_Running` is thus just a local label.

L3-21(3) The **µC/OS-II** flag `OSRunning` is set to `TRUE` indicating that **µC/OS-II** will be running once the first task is started. All ARM instructions are all 32 bits and thus, the ARM is not able to specify a 32-bit address as part of the instruction. Because of that, the address of `OSRunning` is actually declared at the end of the file and the ARM obtains this address via a PC-relative address. Specifically:

```
?OS_Running:
   DC32    OSRunning
```

L3-21(4) We then get the pointer to the task's top-of-stack (was stored by `OSTaskCreate()` or `OSTaskCreateExt()`). See figure 3-1 (`stk` is stored in the `OS_TCB` of the created task).

L3-21(5) We then pop the `CPSR` from the task's stack but we place it in the SPSR register. Recall that when the task was created, the `CPSR` register on the stack frame was initialized with `ARM_SVC_MODE_???` (`0x00000013` for ARM mode or `0x00000033` for Thumb mode). The next instruction will restore the `CPSR` register from the `SPSR` register and place the task in the proper mode (ARM or Thumb) according to the value retrieved for the `SPSR`.

L3-21(6) We then pop the remaining registers of the task's context from the stack. Because the `PC` is the last element popped off the stack, the CPU immediately jumps to that address when it's loaded. In other words, we will run the beginning of the task code as soon as the `PC` is loaded. Note that the '^' indicates to also copy the `SPSR` to the `CPSR` register which places the task in the proper mode (ARM or Thumb).

3.04.04 OS_CPU_A.ASM, OSCtxSw()

The code to perform a 'task level' context switch is shown below in pseudo-code. `OSCtxSw()` is called when a higher priority task is made ready to run by another task or, when the current task can no longer execute (e.g. it calls `OSTimeDly()`, `OSSemPend()` and the semaphore is not available, etc.).

Recall that all tasks run in SVC mode. A task level context switch simply consists of saving the SVC registers on the task to suspend and restore the SVC registers of the new task (see also Figure 3-2). The pseudo code for this is shown below:

```
Save the CPU registers onto the old task's stack;       /* (1) */
OSPrioCur             = OSPrioHighRdy;                  /* (2) */
OSTCBCur->OSTCBStkPtr = SP;                             /* (3) */
OSTaskSwHook();                                         /* (4) */
SP                    = OSTCBHighRdy->OSTCBStkPtr;      /* (5) */
OSTCBCur              = OSTCBHighRdy;                   /* (6) */
Restore the CPU registers from the new task's stack;    /* (7) */
```

You will notice that we don't actually save and restore the `SPSR` register as part of a context switch. The reason is that the `SPSR` is only used to return to the appropriate task and is always used with interrupts disabled.

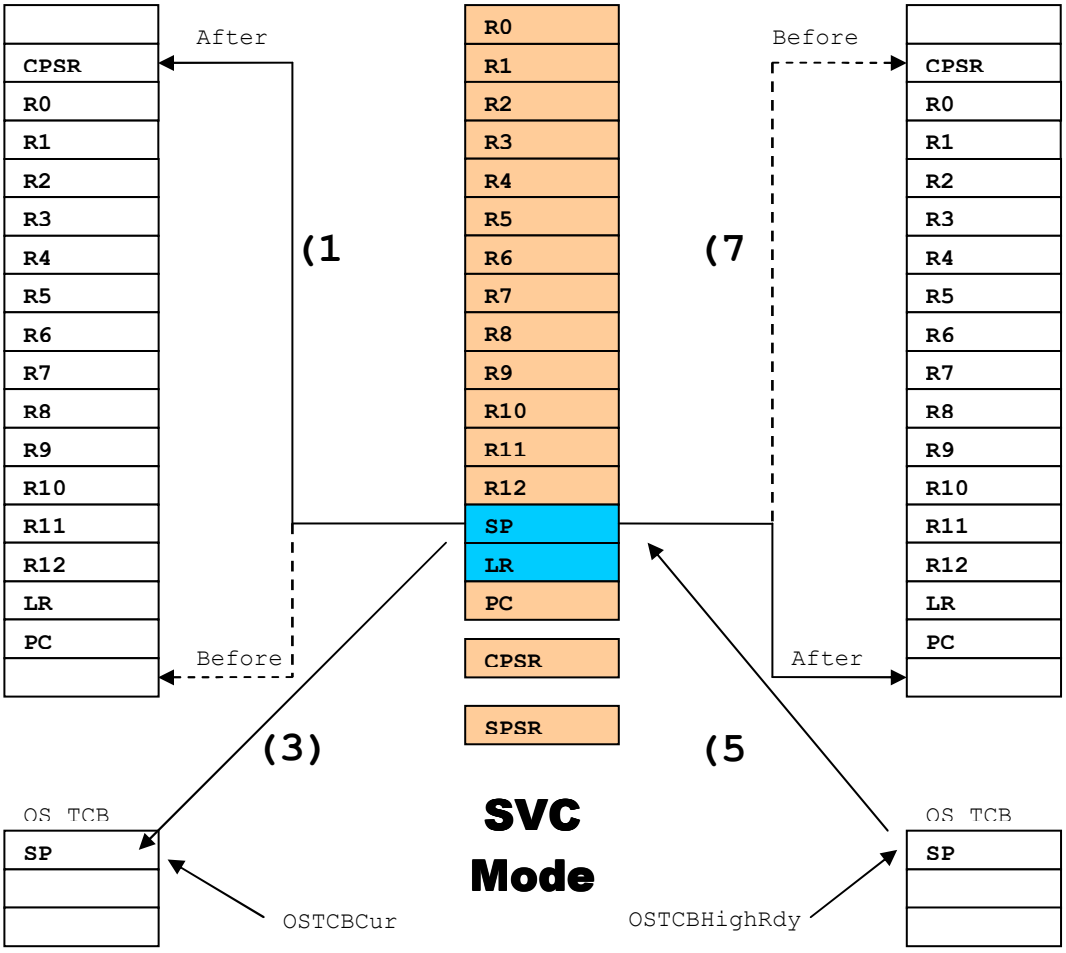

Figure 3-4, Task Level Context Switch.

The actual code for the task level context switch is shown in Listing 3-22.

Listing 3-27, `OSCtxSw()`

```
OSCtxSw
                                                ; SAVE CURRENT TASK'S CONTEXT
    STMFD   SP!, {LR}                           ;     Push return address
    STMFD   SP!, {LR}
    STMFD   SP!, {R0-R12}                       ;     Push registers
    MRS     R0, CPSR                            ;     Push current CPSR
    TST     LR, #1                              ;     See if called from Thumb mode
    ORRNE   R0, R0, #OS_CPU_ARM_CONTROL_THUMB   ;     If yes, Set the T-bit
    STMFD   SP!, {R0}

    LDR     R0, ?OS_TCBCur                      ; OSTCBCur->OSTCBStkPtr = SP;
    LDR     R1, [R0]
    STR     SP, [R1]

    LDR     R0, ?OS_TaskSwHook                  ; OSTaskSwHook();
    MOV     LR, PC
    BX      R0

    LDR     R0, ?OS_PrioCur                     ; OSPrioCur = OSPrioHighRdy;
    LDR     R1, ?OS_PrioHighRdy
    LDRB    R2, [R1]
    STRB    R2, [R0]

    LDR     R0, ?OS_TCBCur                      ; OSTCBCur  = OSTCBHighRdy;
    LDR     R1, ?OS_TCBHighRdy
    LDR     R2, [R1]
    STR     R2, [R0]

    LDR     SP, [R2]                            ; SP = OSTCBHighRdy->OSTCBStkPtr;

                                                ; RESTORE NEW TASK'S CONTEXT
    LDMFD   SP!, {R0}                           ;    Pop new task's CPSR
    MSR     SPSR_cxsf, R0

    LDMFD   SP!, {R0-R12, LR, PC}^              ;    Pop new task's context
```

3.04.05 OS_CPU_A.ASM, OSIntCtxSw()

When an exception handler completes, `OSIntExit()` is called to determine whether a more important task than the interrupted task needs to execute. If that's the case, `OSIntExit()` determines which task to run next and calls `OSIntCtxSw()` to perform the actual context switch to that task. You will notice that `OSIntCtxSw()` is identical to the second half of `OSCtxSw()`. The reason we have these as two separate functions is to simplify debugging. Specifically, if you wanted to set a breakpoint in `OSIntCtxSw()`, you would hit the breakpoint during a task level context switch (if `OSIntCtxSw()` was just a label in `OSCtxSw()`). Of course this would make debugging a bit difficult.

Listing 3-28, `OSIntCtxSw()`

```
OSIntCtxSw
    LDR     R0, ?OS_TaskSwHook              ; OSTaskSwHook();
    MOV     LR, PC
    BX      R0

    LDR     R0, ?OS_PrioCur                 ; OSPrioCur = OSPrioHighRdy;
    LDR     R1, ?OS_PrioHighRdy
    LDRB    R2, [R1]
    STRB    R2, [R0]

    LDR     R0, ?OS_TCBCur                  ; OSTCBCur  = OSTCBHighRdy;
    LDR     R1, ?OS_TCBHighRdy
    LDR     R2, [R1]
    STR     R2, [R0]

    LDR     SP, [R2]                        ; SP = OSTCBHighRdy->OSTCBStkPtr;

                                            ; RESTORE NEW TASK'S CONTEXT.
    LDMFD   SP!, {R0}                       ;    Pop new task's CPSR.
    MSR     SPSR_cxsf, R0

    LDMFD   SP!, {R0-R12, LR, PC}^          ;    Pop new task's context.
```

3.04.06 OS_CPU_A.ASM, Exception Handlers

The seven ARM exception handlers are part of the µC/OS-II port to reduce the amount of work needed by the programmer that is integrating µC/OS-II in his or her product.

In fact, the seven exception handlers are written in a generic way and can actually be used by ANY ARM processor whether it has a built-in interrupt controller or not.

The CPU exception vectors are initialized by the `OS_CPU_InitExceptVect()` function. This function maps the seven exception vectors (all vectors except the reset exception vectors) to seven handlers, `OS_CPU_ARM_Except_XYZ_Hndlr()`. Listing 3-29 presents one of those handlers, `OS_CPU_ARM_ExceptIrqHnldr()`.

The seven handlers all need to save registers R0 to R4 and branch to a global handler called `OS_CPU_ARM_ExceptHndlr()`, presented in listing 3-30. This handler determines if the exception broke a task or another exception and braches to `OS_CPU_ARM_ExceptHndlr_BreakTask()` (listing 3-31) or `OS_CPU_ARM_ExceptHndlr_BreakExcept()` (listing 3-32), respectively.

Both these branches eventually call a board- & CPU-dependent exception handler, `OS_CPU_ExceptHndlr()`, located in the BSP (Board Support Package).

All those handlers (except `OS_CPU_ExceptHndlr()`) are written in assembly language because we simply cannot manipulate CPU registers directly from C.

Note that `OS_CPU_ARM_ExceptHndlr()` moves the processor to SVC mode after saving the exception context and that the exception handler ends up being executed in SVC mode as well. Now that IRQ nesting is supported, IRQs can no longer be handled in the IRQ mode because of possible link register corruption. The problem is that if the exception handler enabled IRQs while in IRQ mode and then called a subroutine, the ARM hardware would overwrite the link register when another IRQ occurred (because the link register is used to store the PC at which the IRQ occurs). All exceptions, not just IRQs, are handled in SVC mode since this greatly simplifies the code.

Listing 3-29, `OS_CPU_ARM_ExceptIrqHndlr()`

```
;*********************************************************************************
;                    INTERRUPT REQUEST EXCEPTION HANDLER
;
; Register Usage:  R0     Exception Type
;                  R1
;                  R2
;                  R3     Return PC
;*********************************************************************************

OS_CPU_ARM_ExceptIrqHndlr
    SUB     LR, LR, #4
    STMFD   SP!, {R0-R3}
    MOV     R2, LR
    MOV     R0, #OS_CPU_ARM_EXCEPT_IRQ
    B            OS_CPU_ARM_ExceptHndlr
```

Listing 3-30, `OS_CPU_ARM_ExceptHndlr()`

```
;*********************************************************************************
;                          GLOBAL EXCEPTION HANDLER
;
; Register Usage:  R0     Exception Type
;                  R1     Exception's SPSR
;                  R2     Return PC
;                  R3     Exception's SP
;*********************************************************************************

OS_CPU_ARM_ExceptHndlr
    MRS     R1, SPSR                    ; Save CPSR (i.e. exception's SPSR).
    MOV     R3, SP                      ; Save exception's stack pointer.

                                        ; Adjust exception stack pointer.  This is needed because
                                        ; exception stack is not used when restoring task context.
    ADD     SP, SP, #(4 * 4)

                                        ; Change to SVC mode & disable interruptions.
    MSR     CPSR_c, #(OS_CPU_ARM_CONTROL_INT_DIS | OS_CPU_ARM_MODE_SVC)

                                        ; SAVE CONTEXT ONTO SVC STACK:
    STMFD   SP!, {R2}                   ;   Push task's PC,
    STMFD   SP!, {LR}                   ;   Push task's LR,
    STMFD   SP!, {R3-R12}               ;   Push task's R12-R3,
    LDMFD   R3!, {R5-R8}                ;   Move task's R4-R0 from exception stack to task's stack.
    STMFD   SP!, {R5-R8}
    STMFD   SP!, {R1}                   ;   Push task's CPSR (i.e. exception SPSR).

                                        ; if (OSRunning == 1)
    LDR     R3, ?OS_Running
    LDRB    R4, [R3]
    CMP     R4, #1
    BNE     OS_CPU_ARM_ExceptHndlr_BreakNothing

                                        ; HANDLE NESTING COUNTER:
    LDR     R3, ?OS_IntNesting          ;   OSIntNesting++;
    LDRB    R4, [R3]
    ADD     R4, R4, #1
    STRB    R4, [R3]

    CMP     R4, #1                      ; if (OSIntNesting == 1)
    BNE     OS_CPU_ARM_ExceptHndlr_BreakExcept
```

Listing 3-31, `OS_CPU_ARM_ExceptHndlr_BreakTask()`

```
;*********************************************************************
;                     EXCEPTION HANDLER: TASK INTERRUPTED
;
; Register Usage:  R0     Exception Type
;                  R1
;*********************************************************************

OS_CPU_ARM_ExceptHndlr_BreakTask
    LDR     R3, ?OS_CPU_ExceptStkPtr   ; OS_CPU_ExceptStkPtr = SP;
    LDR     SP, [R3]

    LDR     R3, ?OS_TCBCur             ; OSTCBCur->OSTCBStkPtr = SP;
    LDR     R4, [R3]
    STR     SP, [R4]

    LDR     R3, ?OS_CPU_ExceptStkBase  ; Switch to exception stack.
    LDR     SP, [R3]

                                       ; EXECUTE EXCEPTION HANDLER:
    LDR     R1, ?OS_CPU_ExceptHndlr    ; OS_CPU_ExceptHndlr(except_type = R0);
    MOV     LR, PC
    BX      R1

                                       ; Change to SVC mode & disable interruptions.
    MSR     CPSR_c, #(OS_CPU_ARM_CONTROL_INT_DIS | OS_CPU_ARM_MODE_SVC)

                                       ; Call OSIntExit().  This call MAY never return if a ready
                                       ; task with higher priority than the interrupted one is
                                       ; found.
    LDR     R0, ?OS_IntExit
    MOV     LR, PC
    BX      R0

    LDR     R3, ?OS_TCBCur             ; SP = OSTCBCur->OSTCBStkPtr;
    LDR     R4,  [R3]
    LDR     SP,  [R4]
                                       ; RESTORE NEW TASK'S CONTEXT:
    LDMFD   SP!, {R0}                  ;    Pop new task's CPSR,
    MSR     SPSR_cxsf, R0

    LDMFD   SP!, {R0-R12, LR, PC}^     ;    Pop new task's context.
```

Listing 3-32, `OS_CPU_ARM_ExceptHndlr_BreakExcept()`

```
;*********************************************************************
;                   EXCEPTION HANDLER: EXCEPTION INTERRUPTED
;
; Register Usage:  R0     Exception Type
;                  R1
;*********************************************************************

OS_CPU_ARM_ExceptHndlr_BreakExcept
                                        ; EXECUTE EXCEPTION HANDLER:
    LDR     R3, ?OS_CPU_ExceptHndlr     ; OS_CPU_ExceptHndlr(except_type = R0)

    MOV     LR, PC
    BX      R3

                                        ; Change to SVC mode & disable interruptions.
    MSR     CPSR_c, #(OS_CPU_ARM_CONTROL_INT_DIS | OS_CPU_ARM_MODE_SVC)

                                        ; HANDLE NESTING COUNTER:
    LDR     R3, ?OS_IntNesting          ;   OSIntNesting--;
    LDRB    R4, [R3]
    SUB     R4, R4, #1
    STRB    R4, [R3]

                                        ; RESTORE OLD CONTEXT:
    LDMFD   SP!, {R0}                   ;    Pop old CPSR,
    MSR     SPSR_cxsf, R0

    LDMFD   SP!, {R0-R12, LR, PC}^      ;   Pull working registers and return from exception.
```

Listing 3-33, `OS_CPU_ARM_ExceptHndlr_BreakNothing()`

```
;*********************************************************************
;                   EXCEPTION HANDLER: 'NOTHING' INTERRUPTED
;
; Register Usage:  R0     Exception Type
;                  R1
;*********************************************************************

OS_CPU_ARM_ExceptHndlr_BreakExcept
                                        ; EXECUTE EXCEPTION HANDLER:
    LDR     R3, ?OS_CPU_ExceptHndlr     ; OS_CPU_ExceptHndlr(except_type = R0)
    MOV     LR, PC
    BX      R3

                                        ; Change to SVC mode & disable interruptions.
    MSR     CPSR_c, #(OS_CPU_ARM_CONTROL_INT_DIS | OS_CPU_ARM_MODE_SVC)

                                        ; RESTORE OLD CONTEXT:
    LDMFD   SP!, {R0}                   ;    Pop old CPSR,
    MSR     SPSR_cxsf, R0

    LDMFD   SP!, {R0-R12, LR, PC}^      ;   Pull working registers and return from exception.
```

You should note that MOST of the work done by the exception handler is actually handled in `OS_CPU_ExceptHndlr()` (located in the BSP) which is written in C. The pseudo-code for `OS_CPU_ExceptHndlr()` is shown in listing 3-34. The handler is responsible for discriminate exceptions and interruptions, determining the source of the interruptions and for executing the appropriate code to handle the interrupting device.

Listing 3-34, OS_CPU_ExceptHndlr()

```
void  OS_CPU_ExceptHndlr (INT32U  except_type)
{
    /* Determine behavior according to exception type (except_type) */

    /* If an IRQ or FIQ */
    while (there are interrupting devices) {
        /* Clear interrupting device */
        OS_CPU_SR_INT_En();                     /* Enable nesting, if desired */
        /* Handle interrupt          */
    }
}
```

`OS_CPU_ExceptHndlr()` is actually part of **YOUR** application and not part of the µC/OS-II port. The reason is that the handler will most likely change depending on the presence of an interrupt controller or not and, if there is an interrupt controller, the actual type of controller.

It's important to note that the handler should 'look' to see whether there are more than one interrupting devices and process each one before returning to `OS_CPU_ARM_ExceptHndlr()`. This avoids going through the overhead of saving the CPU registers upon entry of the exception handlers and restoring them upon exit if multiple interruptions occur either at the same time or, during processing of an interruption.

Note that this port now supports nested interruptions **AND** nested IRQs.

Finally, as a general rule, you should always make your exception handlers as short as possible. Take care of the device, buffer data (if necessary) and signal a task to do most of the work of servicing the data. For example, if you have an Ethernet controller, simply notify a task that an Ethernet packet has arrived and let the task extract the packet from the Ethernet controller.

3.05 OS_DBG.C

`OS_DBG.C` is a file that has been added in V2.62 to provide Kernel Aware debugger to extract information about **µC/OS-II** and its configuration. Specifically, `OS_DBG.C` contains a number of constants that are placed in ROM (code space) which the debugger can read and display. Because you may not be using a debugger that needs that file, you may omit it in your build.

For the IAR compiler as well as Nohau's emulators, Micriµm has introduced a Windows-based 'Plug-In' module that makes use of this file and thus needs to be included if you use IAR's C-Spy or Nohau's Seehau.

4.00 Exception Vector Table

The ARM contains an exception vector table (also called the interrupt vector table) starting at address `0x00000000`. There are only eight (8) entries in the vector table. Each entry has enough room to hold a single 32-bit instruction. The instruction placed in this table is generally a branch instruction with a signed 26-bit destination address. In other words, the ARM can branch to an address that is roughly +/- `0x0200000` from the vector location. The code that you branch to has to determine the interrupt source because there is only one address for all devices that can interrupt the ARM.

The exception vector table for the ARM is shown in table 4-1:

Exception	Mode	Vector Address
Reset	SVC	`0x00000000`
Undefined Instruction	UND	`0x00000004`
Software Interrupt (SWI)	SVC	`0x00000008`
Prefetch abort	Abort	`0x0000000C`
Data abort	Abort	`0x00000010`
Address abort	Abort	`0x00000014`
IRQ (Normal Interrupt)	IRQ	`0x00000018`
FIQ (Fast Interrupt)	FIQ	`0x0000001C`

Table 4-1, ARM's Exception Vector Table

When the CPU recognizes an IRQ from an interrupting device (i.e. IRQ interrupts are enabled), the CPU vectors to address `0x00000018` where it expects to find an instruction that jumps to `OS_CPU_ARM_ExceptIrqHndlr()`. However, it's possible that the code for `OS_CPU_ARM_ExceptIrqHndlr()` is located outside the reach of a normal 'branch' instruction (i.e. beyond the reach of a 26-bit address) and thus we do not want to place a '`B OS_CPU_ARM_ExceptIrqHndlr`' at address `0x00000018`. Instead, we place the following instruction: '`LDR PC,[PC,#0x18]`'. This instruction simply specifies to load the PC with the contents of location `0x00000038`. At location `0x00000038`, we simply place the full 32-bit address of `OS_CPU_ARM_ExceptIrqHndlr()`. This allows the exception handler to be placed anywhere within the 32-bit addressing range of the ARM. The same reasoning applies to the FIQ. To summarize, we need to place the following values for the interrupt vectors:

Exception	Mode	Vector Address	Contents
IRQ (Normal Interrupt)	IRQ	**`0x00000018`**	**`LDR PC,[PC,#0x18]` or `0xE59FF018`**
FIQ (Fast Interrupt)	FIQ	**`0x0000001C`**	**`LDR PC,[PC,#0x18]` or `0xE59FF018`**
…	…	…	…
		`0x00000038`	**Address of `OS_CPU_ARM_ExceptIrqHndlr()`**
		`0x0000003C`	**Address of `OS_CPU_ARM_ExceptFiqHndlr()`**

Table 4-2, Interrupt Vectors

If you are debugging your code in RAM, ensure that the BSP calls the `OS_CPU_ARM_InitExceptVect()`. This will initialize exception vector table to exception handlers.

Listing 4-1, Installing the interrupt vectors in RAM

```
[…]

(*(INT32U *)OS_CPU_ARM_EXCEPT_IRQ_VECT_ADDR)              =
           OS_CPU_ARM_INSTR_JUMP_TO_HANDLER;
(*(INT32U *)OS_CPU_ARM_EXCEPT_IRQ_HANDLER_ADDR)           =
    (INT32U)OS_CPU_ARM_ExceptIrqHndlr;

[…]
```

This assumes that you have RAM at address `0x00000000`. Most ARM processors allow you to re-map RAM to location `0x00000000`. This is done in the example BSP before calling `OS_CPU_ARM_InitExceptVect()`.

If you have Flash (or ROM) at location `0x00000000`, ensure your startup file correctly initialize the exception vector table at compile time.

4.01 Exception Handling Sequence

Below is the sequence of events that take place when an IRQ occurs (assuming the I-bit in the `CPSR` is `0`):

The CPU switches mode to IRQ mode (MODE = `0x12`);
The `CPSR` is saved into the `SPSR_irq` register;
The return address `PC` is saved into `R14_irq` (i.e. the Link Register of the IRQ mode);
The I-bit of the `CPSR` is set to 1 disabling further IRQs;
The `PC` is forced to address `0x00000018;`
The `PC` is loaded with the address of `OS_CPU_ARM_ExceptIrqHndlr()` because of the `LDR PC,[PC,#0x18]` instruction that we placed at address `0x00000018`.
The CPU executes the code in `OS_CPU_ARM_ExceptIrqHndlr()`, then `OS_CPU_ARM_ExceptHndlr()` (found in `OS_CPU_A.S`).
`OS_CPU_ARM_ExceptHndlr()` calls `OS_CPU_ExceptHndlr()` (found in `BSP.C`) to determine the source of the interrupt and handle it accordingly.
When `OS_CPU_ARM_ExceptHndlr()` returns from `OS_CPU_ExceptHndlr()`, it calls `OSIntExit()` (in case of task interrupted) which determines whether there has been a more important task that has been made ready to run by the exception handler or, whether we simply need to return to the interrupted task.
If the interrupted task is still the highest priority task, `OSIntExit()` returns to `OS_CPU_ARM_ExceptHndlr()` which simply returns to this task.
If there is a more important task, `OSIntExit()` calls `OSIntCtxSw()` (see `OS_CPU_A.S`) which takes care of switching to the more important task.

A similar sequence occurs for FIQ interrupts.

4.02 Interrupt Controllers

Some ARM implementations contain a ‘smart’ interrupt controller that supplies a vector (i.e. an address) for each interrupt source. This allows the proper interrupt handler to be called quickly instead of having the interrupt handler ‘poll’ each possible interrupting device to determine if it needs servicing.

4.02.01 Interrupt Controllers, Atmel's AIC

The Atmel AT91 and SAM7 families of processors have an Advanced Interrupt Controller (AIC). Once initialized, the AIC provides the 32-bit address of the ISR for the highest priority interrupting device at location `0xFFFFF100`. In other words, the interrupting device's ISR address can be read from location `0xFFFFF100`. When there are no more interrupting devices, location `0xFFFFF100` contains `0x00000000`. Refer to the AIC documentation for additional details.

Similarly, the address of the ISR for the FIQ interrupting device is found at address `0xFFFFF104`. `OS_CPU_ExceptHndlr()` can thus be written as shown in listing 4-3.

Listing 4-3, OS_CPU_ExceptHndlr() for Atmel's AIC.

```
#define  AIC_IVR  (*(INT32U *)0xFFFFF100)
#define  AIC_FVR  (*(INT32U *)0xFFFFF104)

typedef  void  (*BSP_FNCT_PTR)(void);


void  OS_CPU_ExceptHndlr (CPU_DATA  except_type)
{
    BSP_FNCT_PTR   pfnct;
    CPU_INT32U    *sp;

    if (except_type ==  OS_CPU_ARM_EXCEPT_FIQ) {
        pfnct = (BSP_FNCT_PTR)*AT91C_AIC_FVR;       /* Read the FIQ handler from the AIC.     */
        while (pfnct != (BSP_FNCT_PTR)0) {          /* Make sure we don't have a NULL pointer.*/
            (*pfnct)();                             /* Execute the handler.                   */
            *AT91C_AIC_EOICR = ~0;                  /* End of handler.                        */
            pfnct = (BSP_FNCT_PTR)*AT91C_AIC_FVR;   /* Read the FIQ handler from the AIC.     */
        }
        *AT91C_AIC_EOICR = ~0;                      /* End of handler.                        */
    } else if (except_type == OS_CPU_ARM_EXCEPT_IRQ) {
        pfnct = (BSP_FNCT_PTR)*AT91C_AIC_IVR;       /* Read the IRQ handler from the AIC.     */
        while (pfnct != (BSP_FNCT_PTR)0) {          /* Make sure we don't have a NULL pointer.*/
            OS_CPU_SR_INT_En();                     /* OPTIONAL: Enable interrupt nesting.    */
            (*pfnct)();                             /* Execute the handler.                   */
            OS_CPU_SR_INT_Dis();                    /* Disable interrupt nesting.             */
            *AT91C_AIC_EOICR = ~0;                  /* End of handler.                        */
            pfnct = (BSP_FNCT_PTR)*AT91C_AIC_IVR;   /* Read the IRQ handler from the AIC.     */
        }
        *AT91C_AIC_EOICR = ~0;                      /* End of handler.                        */
    } else {
        /* Other exception handling */
        }
    }
}
```

It's **IMPORTANT** to note that you **MUST** place the address of the ISR ***handler*** in the proper AIC register in order for `OS_CPU_ExceptHndlr()` to work properly. You **DO NOT** want to place the address of `OS_CPU_ExceptHndlr()` as the ISR address for the AIC.

Your ISR handlers should be written as follows:

```
void  MyISR_Hndlr (void)
{
    /* Service the interrupting device                                  */
    /* Buffer the data (if any) and signal a task to process the data */
    /* Clear the interrupting device (i.e. acknowledge the device)    */
}
```

4.02.02 Interrupt Controllers, NXP and Sharp's VIC

The NXP LPC2000 series (ARM7), Sharp ARM7 and ARM9 families of processors have a Vectored Interrupt Controller (VIC). Once initialized, the VIC provides the 32-bit address of the ISR for the highest priority interrupting device at location `0xFFFFF030`. In other words, the interrupting device's ISR can be read from location `0xFFFFF030`. When there are no more interrupting devices, location `0xFFFFF030` contains `0x00000000`.

`OS_CPU_ExceptHndlr()` can thus be written as shown in listing 4-4.

Listing 4-4, OS_CPU_ExceptHndlr() for Philips and Sharp's VIC.

```
#define  VIC_VECTADDR  (*(INT32U *)0xFFFFF030)

typedef  void  (*BSP_FNCT_PTR)(void);

void  OS_CPU_ExceptHndlr (CPU_DATA  except_type)
{
    BSP_FNCT_PTR   pfnct;
    CPU_INT32U    *sp;

    if (except_type ==  OS_CPU_ARM_EXCEPT_FIQ) {
        pfnct = (BSP_FNCT_PTR)* VIC_VECTADDR;       /* Read the FIQ handler from the VIC.     */
        while (pfnct != (BSP_FNCT_PTR)0) {          /* Make sure we don't have a NULL pointer.*/
            (*pfnct)();                             /* Execute the handler.                   */
            *VIC_ADDR = ~0;                         /* End of handler.                        */
            pfnct = (BSP_FNCT_PTR)* VIC_VECTADDR;   /* Read the FIQ handler from the VIC.     */
        }
    } else if (except_type == OS_CPU_ARM_EXCEPT_IRQ) {
        pfnct = (BSP_FNCT_PTR)* VIC_VECTADDR;       /* Read the IRQ handler from the VIC.     */
        while (pfnct != (BSP_FNCT_PTR)0) {          /* Make sure we don't have a NULL pointer.*/
            OS_CPU_SR_INT_En();                     /* OPTIONAL: Enable interrupt nesting.    */
            (*pfnct)();                             /* Execute the handler.                   */
            OS_CPU_SR_INT_Dis();                    /* Disable interrupt nesting.             */
            *VIC_ADDR = ~0;                         /* End of handler.                        */
            pfnct = (BSP_FNCT_PTR)* VIC_VECTADDR;   /* Read the IRQ handler from the VIC.     */
        }
    } else {
        /* Other exception handling */
    }
}
```

It's **IMPORTANT** to note that you **MUST** place the address of the ISR ***handler*** in the proper VIC register in order for `OS_CPU_ExceptHndlr()` to work properly. You **DO NOT** want to place the address of `OS_CPU_ExceptHndlr()` as the ISR address for the VIC.

Your ISR handlers should be written as follows:

```
void  MyISR_Hndlr (void)
{
    /* Service the interrupting device                                  */
    /* Buffer the data (if any) and signal a task to process the data */
    /* Clear the interrupting device (i.e. acknowledge the device)    */
}
```

4.02.03 Interrupt Controllers, Freescale i.MX

The Freescale i.MX series have an Interrupt Controller called the AITC. Once initialized, the AITC provides the 'index' (a number between 0 and 63, incl.) of the highest priority interrupting device. The index can then be used as an index into a table of interrupt vectors. The index for the highest priority interrupting device is found at location `0x00223040` (for the i.MX1). This is called the Normal Interrupt Vector and Status Register (`NIVECSR`).

Similarly, the index of the interrupting device for the FIQ interrupting device is found at address `0x00223044`. The is called the Fast Interrupt Vector and Status Register (`FIVECSR`).

There are a number of things we need to setup to use the AITC as shown in the following listings. This code would normally be placed in the BSP of the target board.

Listing 4-5, #defines

```
#define  BSP_NIVECSR  (*(INT32U *)0x00223040L)
#define  BSP_FIVECSR  (*(INT32U *)0x00223044L)
```

These are the addresses of the `NIVECSR` and `FIVECSR` registers, respectively.

Listing 4-6, Data Types

```
typedef  void (*BSP_FNCT_PTR)(void);
```

This declares a new data type for a pointer to a function.

Listing 4-7, Exception handler address table

```
BSP_FNCT_PTR  BSP_ExceptHndlrVectTbl[64];
```

This declares an array of pointers to functions. Each interrupting device is identified by an index from `0` to `63` which is contained in the BSP_`NIVECSR` for an IRQ and the BSP_`FIVECSR` for an FIQ. We would use this index to extract the address of the exception handler from this table (see `OS_CPU_ExceptHndlr()` for details).

Listing 4-8, Unused exception handler

```
static  void  BSP_ExceptDummyHndlr(void)
{
}
```

Here we declare a 'dummy' function in order to populate the exception vector table (i.e. `BSP_ExceptHndlrVectTbl[]`) with a pointer to this function. This is used in case there is no handler associated with an interrupting device.

Listing 4-9, Initialization of the exception vector table

```
static  void  BSP_Init(void)
{
[…]
    INT16U  i;

[…]
    for (i = 0; i < 64; i++) {
        BSP_ExceptHndlrVectTbl[i] = BSP_ExceptDummyHndlr;
    }
}
```

We initialize the table containing the addresses of the exception handler for each interrupting device. When you want the CPU to service a specific device, you would simply 'install' the exception handler by calling `BSP_ExceptHndlrSet()` as described in Listing 4-10.

Listing 4-10, Specifying the address of an exception handler

```
void  BSP_ExceptHndlrSet (INT32U except_type, BSP_FNCT_PTR  pHndlr)        (1)
{
    if (except_type < 64) {                                               (2)
        BSP_ExceptHndlrVectTbl[except_type] = pHndlr;                     (3)
    }
}
```

L4-10(1) When you want the CPU to service a specific device, you would simply 'install' the exception handler by calling `BSP_ExceptHndlrSet()` and specify the 'except_type' as well as the address for the exception handler. You **MUST** declare your handlers as follows:

```
void MyExceptHndlr(void)
{
    Handle the device that generated the exception.
    Possibly buffer and signal a task to handle the data;
    Don't forget to 'CLEAR' the interrupting device.
}
```

L4-10(2) You **MUST** specify an exception id between 0 and 63, inclusively.

L4-10(3) The address of the exception handler is saved in the table.

Listing 4-11, OS_CPU_ExceptHndlr() for the Freescale's AITC

```
void  OS_CPU_ExceptHndlr (void)
{
    INT16U        except_type;
    BSP_FNCT_PTR  pfnct;

    except_type = (BSP_NIVECSR >> 16) & 0x00FF;          (1)
    while (except_type < 64) {                           (2)
        pfnct = BSP_ExceptHndlrVectTbl[except_type];     (3)
        if (pfnct != (BSP_FNCT_PTR)0) {                  (4)
            pfnct();                                     (5)
        }
        except_type = (BSP_NIVECSR >> 16) & 0x00FF;      (6)
    }
}
```

L4-11(1) We get the 'except_type' of the highest priority exception to service which is found in the upper 16 bits of the BSP_`NIVECSR` register.

L4-11(2) We want to service ALL interrupting devices. In other words, there is no point of returning from an exception if there are 'more' devices interrupting the CPU. This reduces the overhead associated with servicing multiple consecutive exceptions. Note the BSP_`NIVECSR` will contain an index higher than 63 when there are no more devices interrupting the CPU.

L4-11(3) If we have a valid index, we obtain the address of the exception handler associated with the interrupting device.

L4-11(4) Just in case, we make sure a 'distracted' programmer didn't decide to place a `NULL` pointer as an exception handler.

L4-11(5) We execute the exception handler for the interrupting device.

L4-11(6) Finally, we check to see whether there are other interrupts to service.

5.00 Debugging in RAM

A large number of ARM chips allow you to re-map RAM at location `0x00000000` which allows you to change exception and interrupt vectors at run-time (especially useful during debug).

The remapping of RAM at location `0x00000000` allows you to install the IRQ and FIQ interrupt vectors as discussed in the previous section.

Some ARM cores contain an MMU. In order to 'remap' RAM at address `0x00000000`, the MMU needs to be initialized and the remapping is actually done by the MMU. MMU initialization is assumed to be part of the application code. As far as **µC/OS-II** is concerned, you need to locate some RAM from address `0x00000000` to `0x0000003F` during debugging in order to setup the interrupt vectors.

6.00 Application Code

Your application code can make use of the port presented in this application note as described in this section. Figure 6-1 shows a block diagram of the relationship between your application, **µC/OS-II**, the **µC/OS-II** port, the BSP (Board Support Package), the ARM CPU and the target hardware.

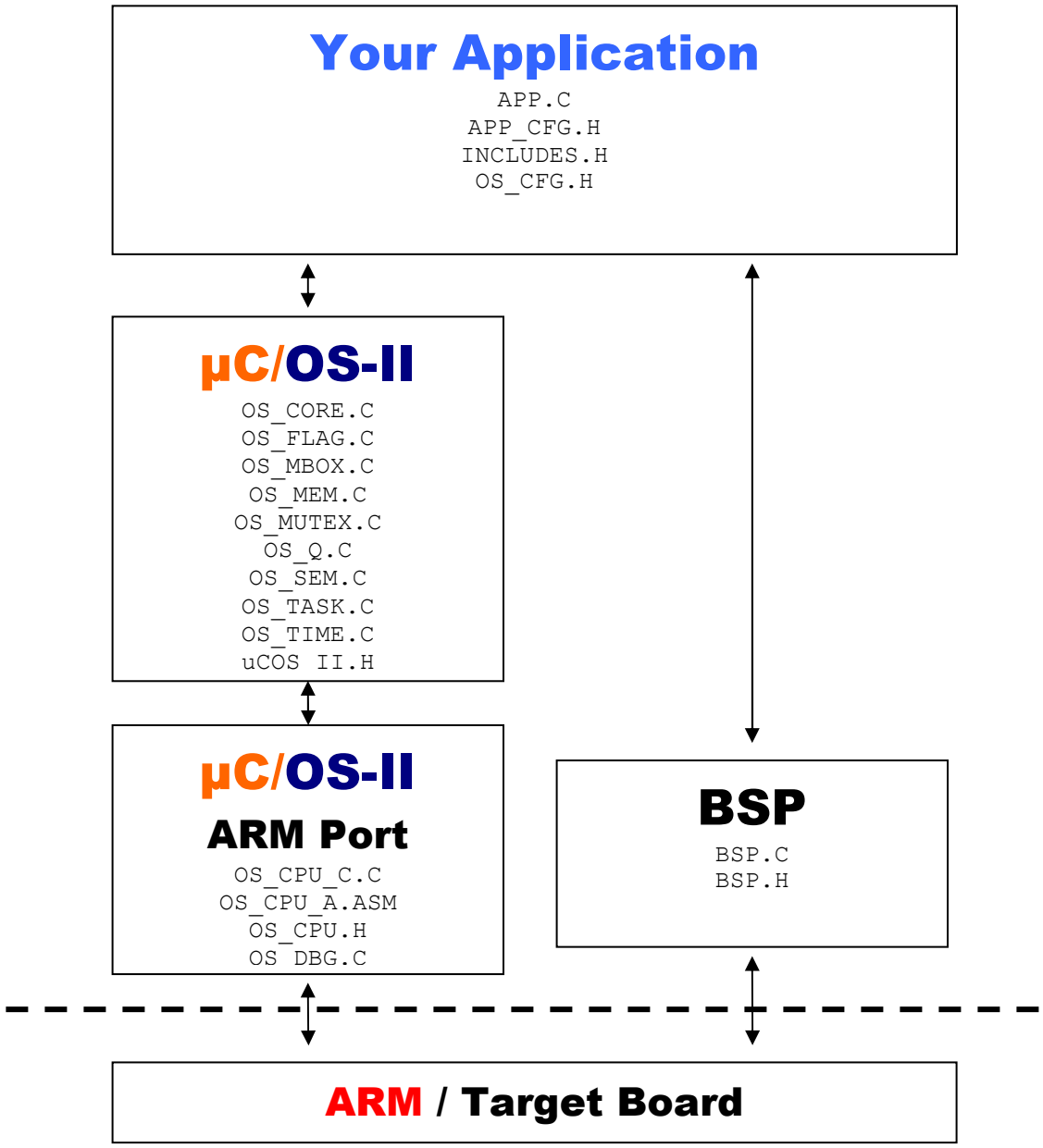

Figure 6-1, Relationship between modules.

6.01 APP.C, APP.H and APP_CFG.H

For sake of discussion, your application is placed in files called `APP.C` and `APP_CFG.H`. Of course, your application (i.e. product) can contain many more files.

`APP.C` would be where you would place `main()` but, of course, you can place `main()` anywhere you want.

`APP_CFG.H` contains `#define` constants to configure the application. We placed task stack sizes task priorities and other #defines in this file. This allows you to locate task priorities and sizes in one place.

`APP.C` is a standard test file for µC/OS-II examples. The two important functions are `main()` (listing 6-1) and `AppStartTask()` (listing 6-2).

Listing 6-1, `main()`

```
int  main (void)
{
#if (OS_TASK_NAME_SIZE >= 16)
    CPU_INT08U  os_err;
#endif


   (void)&App_Clk_UTC_Offset;

    os_err = 0;                  /* Warning: With some debuggers the first call is       */
                                 /* ignored.                                             */

    BSP_Init();                  /* Initialize BSP.                                      */
    CPU_Init();                  /* Initialize CPU.                                      */

    APP_TRACE_DEBUG(("\n\n\n"));
    APP_TRACE_DEBUG(("Initialize OS...\n"));
    OSInit();                    /* Initialize OS.                                  (1) */

                                 /* Create start task.                              (2) */
    OSTaskCreateExt( App_TaskStart,
                    (void *)0,
                    (OS_STK *)&App_StartTaskStk[APP_START_OS_CFG_TASK_STK_SIZE - 1],
                     APP_START_OS_CFG_TASK_PRIO,
                     APP_START_OS_CFG_TASK_PRIO,
                    (OS_STK *)&App_StartTaskStk[0],
                     APP_START_OS_CFG_TASK_STK_SIZE,
                    (void *)0,
                     OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

                                 /* Give a name to tasks.                                */
#if (OS_TASK_NAME_SIZE >= 16)
    OSTaskNameSet(OS_TASK_IDLE_PRIO,          "Idle",  &os_err);                  (3)
#if (OS_TASK_STAT_EN > 0)
    OSTaskNameSet(OS_TASK_STAT_PRIO,          "Stat",  &os_err);
#endif
    OSTaskNameSet(APP_START_OS_CFG_TASK_PRIO, "Start", &os_err);                  (4)
#endif

    APP_TRACE_DEBUG(("Start OS...\n"));
    OSStart();                   /* Start OS.                                       (5) */
}
```

L6-1(1) As with all **µC/OS-II** based applications, you need to initialize **µC/OS-II** by calling `OSInit()`.

L6-1(2) You need to create at least one task. In this case, we created the task using the extended task create call. This allows **µC/OS-II** to have more information about your task. Specifically, with the IAR toolchain, the extra information allows the C-Spy debugger to display stack usage information when you use the **µC/OS-II** Kernel Awareness Plug-In.

L6-1(3) **µC/OS-II** doesn't name the idle task nor the statistic task by default and thus, we can do this at this point. In fact, we could have name these task immediately after calling `OSInit()`.

L6-1(4) We can now give names to tasks and those can be displayed by Kernel Aware debuggers such as IAR's C-Spy.

L6-1(5) In order to start multitasking, you need to call `OSStart()`. Note that `OSStart()` will not return from this call.

Listing 6-2, `AppStartTask()`

```
static  void  App_TaskStart (void  *p_arg)
{
#if (CPU_CFG_NAME_EN == DEF_ENABLED)
    CPU_ERR  err;
#endif


    (void)&p_arg;                /* Prevent compiler warning.                          */

    APP_TRACE_DEBUG(("Initialize OS timer...\n"));
    Tmr_Init();                  /* Initialize OS timer.                               */

#if (OS_TASK_STAT_EN > 0)
    APP_TRACE_DEBUG(("Initialize OS statistic task...\n"));
    OSStatInit();                /* Initialize OS statistic task.                 (1) */
#endif

    APP_TRACE_DEBUG(("Create application task...\n"));
    App_TaskCreate();            /* Create application task.                       (2) */

[…]
                                                                                   (3)
    LED_Off(1);                                                                    (4)
    LED_Off(2);
    LED_Off(3);

    while (DEF_YES) {            /* Task body, always written as an infinite loop.     */
        OSTimeDlyHMSM(0, 0, 0, 500);

[…]
    }
}
```

L6-2(2) If you enabled the statistic task by setting `OS_TASK_STAT_EN` in `OS_CFG.H` to `1`) then, you need to call it here. Please note that you need to make sure that you initialized and enabled the **µC/OS-II** clock tick because `OSStatInit()` assumes the presence of clock ticks. In other words, if the tick interruption handler is not active when you call `OSStatInit()`, your application will end up in **µC/OS-II**'s idle task and not be able to run any other tasks.

L6-2(3) At this point, you can create additional tasks. We decided to place all our task initialization in one function called `AppTaskCreate()` but, you are certainly welcome to use a different technique.

L6-2(4) You can now perform whatever additional function you want for this task.

L6-2(5) We decided to toggle an LED at a rate of 10 Hz (LED will blink at 2 Hz) when this task is running (see section 7.00, Board Support Package).

6.02 INCLUDES.H

INCLUDES.H is a *master* include file and is found at the top of all .C files. INCLUDES.H allows every .C file in your project to be written without concern about which header file is actually needed. The only drawbacks to having a master include file are that INCLUDES.H may include header files that are not pertinent to the actual .C file being compiled and the compilation process may take longer. These inconveniences are offset by code portability. You can edit INCLUDES.H to add your own header files, but your header files should be added at the end of the list. Listing 6-3 shows the typical contents of INCLUDES.H. Of course, you can add your own header files as needed.

Listing 6-3, INCLUDES.H

```
#include  <ctype.h>
#include  <stdarg.h>
#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>

#include  <app_cfg.h>

#include  <ucos_ii.h>
#include  <bsp.h>
```

7.00 BSP (Board Support Package)

It is often convenient to create a Board Support Package (BSP) for your target hardware. A BSP could allow you to encapsulate the following functionality:

Timer initialization
Exception handlers
LED control functions
Reading switches
Setting up the interrupt controller
Setting up communication channels
Etc.

A BSP consist of 2 files: `BSP.C` and `BSP.H`.

For example, because a number of evaluation boards are equipped with LEDs, we decided to create LED control functions as follows:

```
void  LED_Init(void);
void  LED_On(INT8U  led_id);
void  LED_Off(INT8U led_id);
void  LED_Toggle(INT8U led_id);
```

In this case, LEDs are referenced 'logically' instead of physically. When you write the BSP, you determine which LED is LED #1, which is LED #2, etc. When you want to turn on LED #1, you simply call `LED_On(1)`. If you want to toggle LED #2, you simply call `LED_Toggle(2)`. In fact, you can (and should) associate names to your LEDs using `#define`s. You could thus specify `LED_Off(LED_PM)`.

Each BSP should contain a BSP initialization function. We called ours `BSP_Init()` and should be called by your application code.

We decided to encapsulate the **µC/OS-II** clock tick handler in the BSP because exception handlers really belong into your application code and not **µC/OS-II**. Doing this makes it easier to adapt the **µC/OS-II** port to different target hardware since you could simply change the BSP to select whichever timer or interrupt source for the clock tick. The clock tick interruption handler is found in `BSP.C` and is called `Tmr_TickHndlr()`.

It's assumed that the generic exception handler (`OS_CPU_ExceptHndlr()`) is declared in `BSP.C` (see section 4 for details).

8.00 Conclusion

This application note presented a 'generic' port for ARM processors (ARM7 or ARM9). The port should be easily adapted to different compilers (the code itself should be identical). Of course, if you use **µC/OS-II** and use the port on actual hardware, you will need to initialize and properly handle hardware interrupts.

Acknowledgements

I would like to thank Mr. Harry Barnett (R.I.P.) and Mr. Michael Anburaj for their contribution of the original ARM port.

Licensing

If you intend to use **µC/OS-II** in a commercial product, remember that you need to contact Micrium to properly license its use in your product.

References

MicroC/OS-II, The Real-Time Kernel, 2nd Edition
Jean J. Labrosse
R&D Technical Books, 2002
ISBN 1-5782-0103-9

Contacts

CMP Books, Inc.
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-9950
USA
+1 785 841 1631
+1 785 841 2624 (FAX)
WEB: http://www.rdbooks.com
e-mail: rdorders@rdbooks.com

IAR Systems, Inc.
Century Plaza
1065 E. Hillsdale Blvd
Foster City, CA 94404
USA
+1 650 287 4250
+1 650 287 4253 (FAX)
WEB: http://www.IAR.com
e-mail: info@IAR.com

Macraigor Systems LLC
PO Box 471008
Brookline Village, MA 02445
+1 206 855 9269
+1 206 855 9297 (FAX)
WEB: http://www.Macraigor.com

Micriμm
949 Crestview Circle
Weston, FL 33327
USA
+1 954 217 2036
+1 954 217 2037 (FAX)
e-mail: Licensing@Micrium.com
WEB: www.Micrium.com

Nohau Corporation
51 E. Campbell Ave
Campbell, CA 95008
USA
+1 408 866 1820
+1 408 378 7869 (FAX)
WEB: http://www.Nohau.com
e-mail: support@Nohau.com

Notes