Micriμm

μC/OS-II

and

The ARM Processor

Application Note
AN-1011C

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.

This application note is also accompanied by a Microsoft 'PowerPoint' presentation (AN-1011-PPT.PDF) that walks you through all the steps of a context switch. This will be referenced as needed in this application note.

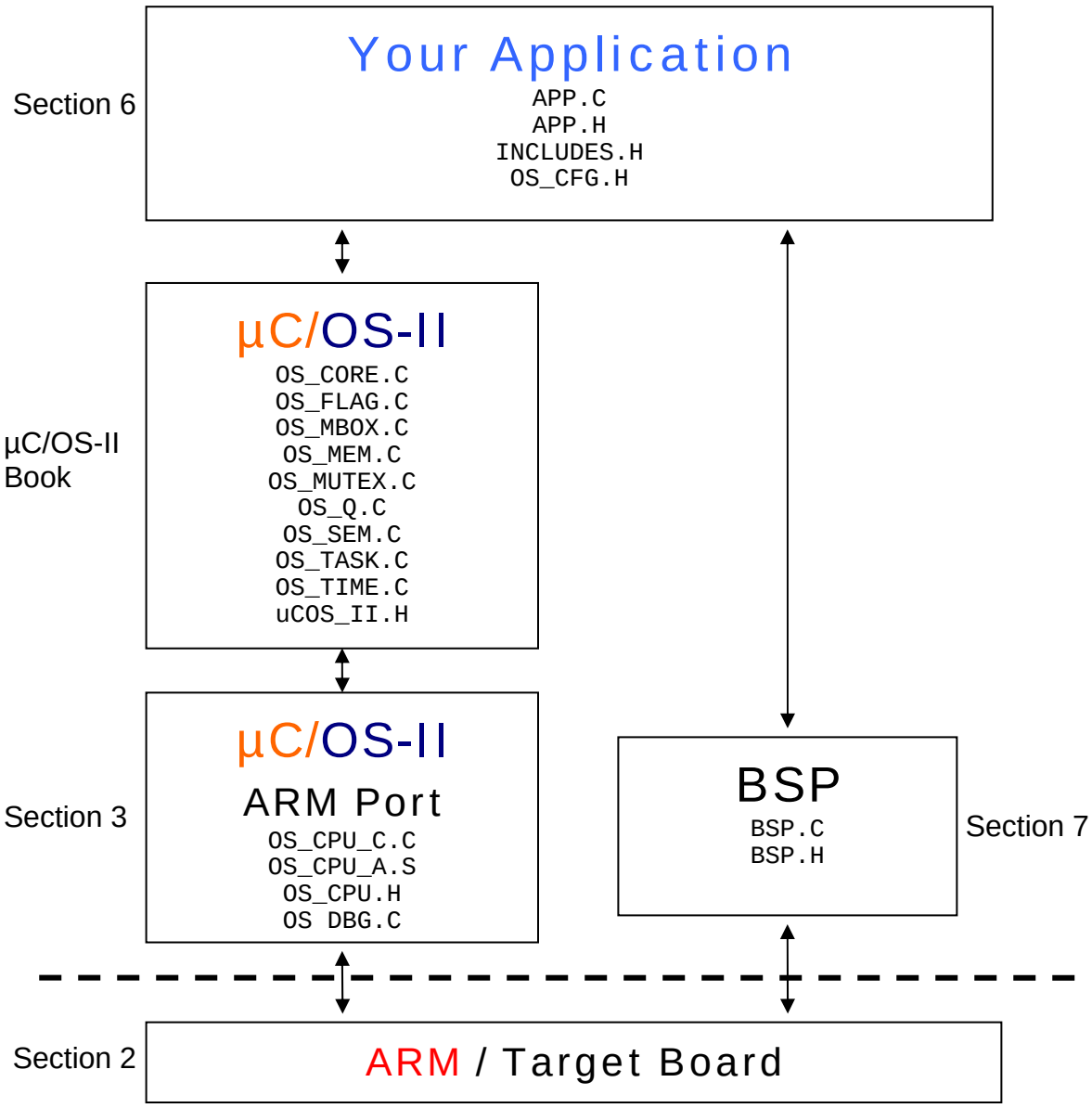

Figure 1-1, Relationship between modules.

2.00 The ARM programmer's model

One of the most popular variant of the ARM processor is called the ARM7TDMI and the four letters at the end stands 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 stand 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 tthrough 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).

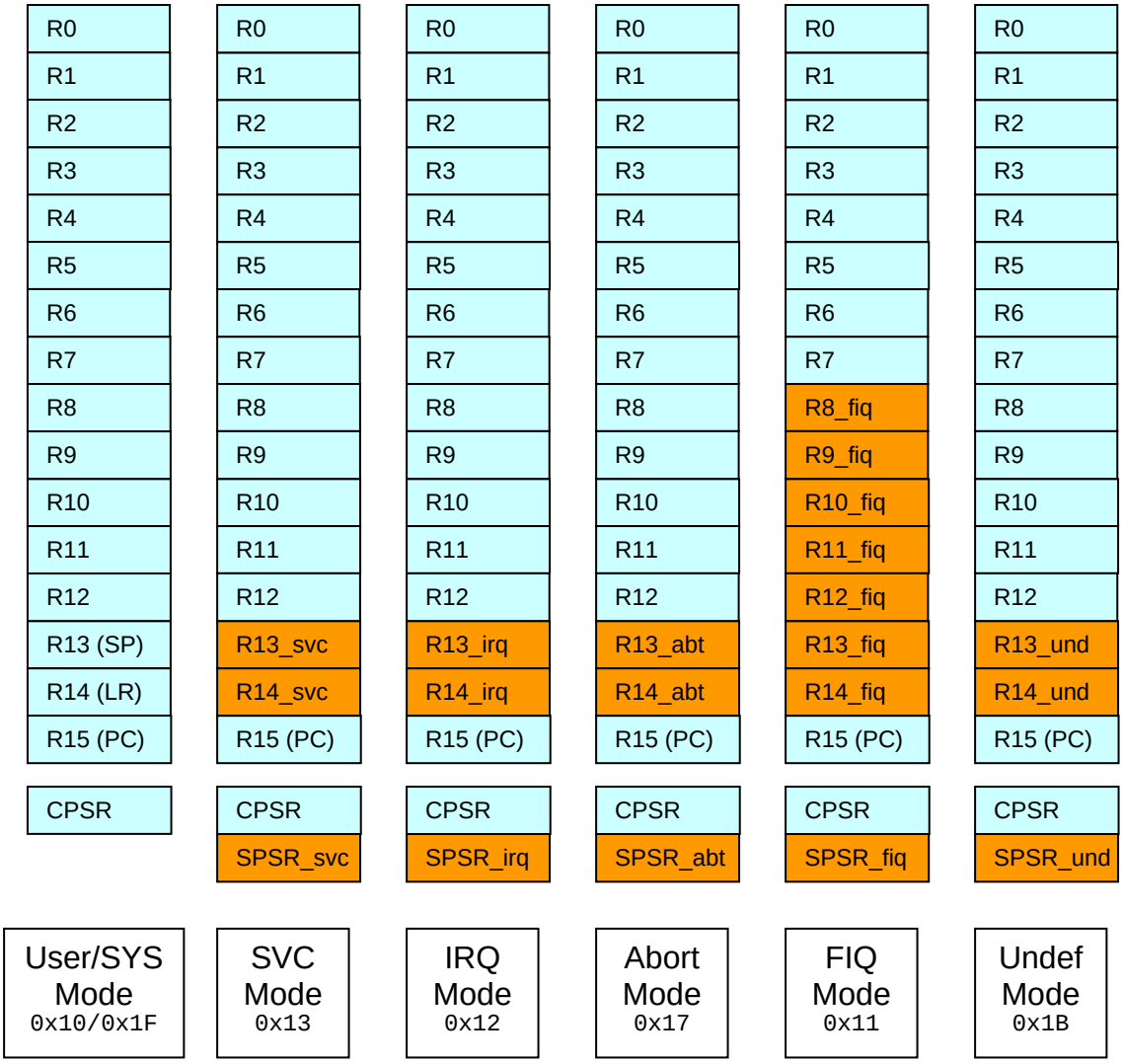

Figure 2-1, ARM Register Model.

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.

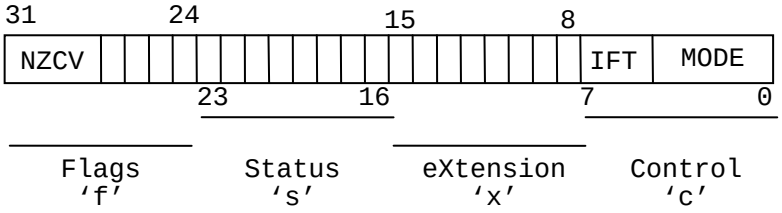

Figure 2-2, The CPSR Register.

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 'priviledged' 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. We decided to run user applications (i.e. tasks) in SYS 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.

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

This µC/OS-II port doesn't handle ABORT exceptions and thus, it's up to your application to deal with these types of exceptions.

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`

This µC/OS-II port doesn't handle UNDEF exceptions and thus, it's up to your application to deal with these types of exceptions.

3.00 µC/OS-II Port for the ARM

We used the IAR EWARM V4.11A (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 other words, the IAR compiler calls `main()` in SYS mode.

Below are a few assumptions about the port:

- You have µC/OS-II V2.7x and higher
- µC/OS-II runs in ARM mode
- Tasks are created in ARM mode
- Tasks will run in SYS mode

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\ARM\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.S
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 stacks 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, 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.S` (described later).

Listing 3-5, OS_CPU.H, Task Level Context Switch

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

3.02.06 OS_CPU.H, Function Prototypes

The prototypes in Listing 3-6 are for the functions used to disable and re-enable interrupts using `OS_CRITICAL_METHOD` #3 and are described later. Note that the macros are different depending on whether `OS_CPU_INT_DIS_MEAS_EN` is set to 0 or not. Basically, this allows us to measure the amount of time interrupts are disabled. This technique will be described later.

Listing 3-6, OS_CPU.H, Function Prototypes

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

#endif
```

The prototypes in Listing 3-7 are for the interrupt service routines (ISR) that handle both the IRQ and FIQ interrupts. `OS_CPU_IRQ_ISR()` is the ISR entry point for the IRQ interrupt and is written in assembly language. Most of the IRQ handling is actually done by `OS_CPU_IRQ_ISR_Handler()` which is written in C. Basically, we want to have as little assembly language code as possible. The same reasoning applies to the FIQ interrupt. The 'handlers' are assumed to reside in the application's BSP (Board Support Package) because the way we handle the interrupts depends on the actual ARM chip used. Some chips have on-chip interrupt controllers which greatly simplify the task of identifying the source of the interrupt and thus, allow us to quickly execute the appropriate interrupt service routine.

Listing 3-7, OS_CPU.H, Function Prototypes

```
void       OS_CPU_IRQ_ISR(void);
void       OS_CPU_IRQ_ISR_Handler(void);

void       OS_CPU_FIQ_ISR(void);
void       OS_CPU_FIQ_ISR_Handler(void);
```

The prototypes in Listing 3-8 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 3-8, 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
```

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:

```
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, 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 not set to 0).

Listing 3-9, OS_CPU_C.C, `OSInitHookEnd()`

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

3.03.02 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 our case, we call the initialization function of µC/OS-View (an optional module available for µC/OS-II which performs task profiling at run-time, See www.micrium.com for details).

Note that for `OSView_TaskCreateHook()` to be called, the target resident code for µC/OS-View must be included as part of your build. In this case, you need to add a `#define OS_VIEW_MODULE 1` in `OS_CFG.H` of your application.

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

Listing 3-10, OS_CPU_C.C, `OSInitHookEnd()`

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

3.03.03 OS_CPU_C.C, OSTaskStkInit()

µC/OS-II assumes that tasks run in SYS mode (the `CPSR` of the task is initialized to `ARM_SYS_MODE` (`0x1F`)).

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 3-12. The task received an optional argument '`p_arg`'. That's why '`p_arg`' is passed in `R0` when the task is created. You should note that we initialized the CPU registers (the SYS registers) to values corresponding to their register number. This makes it convenient when debugging and examining stacks. The initial values are very useful when the task is first created but, of course, the register values will most likely change as the task code is executed.

Listing 3-11, OS_CPU_C.C, `OSTaskStkInit()`

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


    opt      = opt;                         /* 'opt' is not used, prevent warning       */
    stk      = ptos;                        /* Load stack pointer                       */
    *(stk)   = (OS_STK)task;                /* 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                            */
    *(--stk) = (INT32U)ARM_SYS_MODE;        /* CPSR  (Enable both IRQ & FIQ interrupts) */

    return (stk);
}
```

Listing 3-12, µC/OS-II Task

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

Figure 3-1 shows how the stack frame is initialized for each task when it's created.

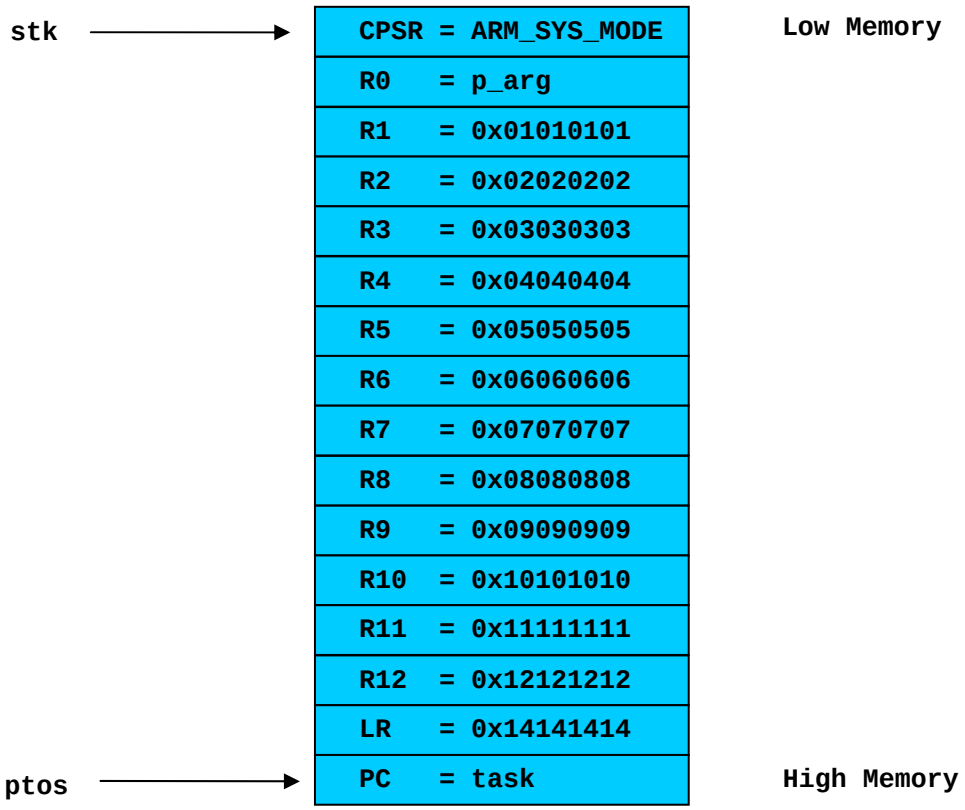

Figure 3-1, The Stack Frame for each Task for ARM port.

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()`.

3.03.04 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. In this case, we call the µC/OS-View task switch hook called `OSView_TaskSwHook()`. This assumes that you have µC/OS-View as part of your build and that you set `OS_VIEW_MODULE` to `1` in `OS_CFG.H`.

Listing 3-13, OS_CPU_C.C, OSIntCtxSw()

```
void  OSCtxSwHook (void)
{
#if OS_VIEW_MODULE > 0
    OSView_TaskSwHook();
#endif
}
```

3.03.05 OS_CPU_C.C, OSTimeTickHook()

`OSTimeTickHook()` is called at the very beginning of `OSTimeTick()`. This function allows the port code to be extended and, in our case, we call the µC/OS-View function `OSView_TickHook()`. Again, this assumes that you have µC/OS-View as part of your build and that you set `OS_VIEW_MODULE` to `1` in `OS_CFG.H`.

Listing 3-14, OS_CPU_C.C, OSTimeTickHook()

```
void  OSTimeTickHook (void)
{
#if OS_VIEW_MODULE > 0
    OSView_TickHook();
#endif
}
```

3.03.06 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 3-15 initializes the measurement and can actually be called at any time to ‘reset’ the maximum count.

Listing 3-15, 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
```

3.03.07 OS_CPU_C.C, OS_CPU_IntDisMeasStart()

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

Listing 3-16, 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
```

L3-16(1) A nesting counter is maintained in case you nest `OS_ENTER_CRITICAL()` calls.

L3-16(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`.

3.03.08 OS_CPU_C.C, OS_CPU_IntDisMeasStop()

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

Listing 3-17, 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
```

L3-17(1) The nesting counter is decremented so that we only take a time measurement at the last nested `OS_EXIT_CRITICAL()` calls.

L3-17(2) We measure the difference in timer value since interrupts were disabled.

L3-17(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.

L3-17(4) We record the highest value in `OS_CPU_IntDisMeasCntsMax`.

3.04 OS_CPU_A.S

A µC/OS-II port requires that you write five fairly simple assembly language functions. The ARM port actually contains 7 functions because portions of the ISR code is written in assembly language as discussed in this section. These functions are needed because you normally cannot save/restore registers from C functions. The four functions are:

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

`OSIntCtxSw()` was described in the previous section and was actually found in `OS_CPU_C.C` for this port.

3.04.01 OS_CPU_A.S, OS_CPU_SR_Save()

The code in listing 3-18 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. When this function returns, `R0` contains the state of the `CPSR` register prior to disabling interrupts.

Listing 3-18, OS_CPU_SR_Save()

```
        CODE32

OS_CPU_SR_Save
        MRS     R0,CPSR         ; Set IRQ and FIQ bits in CPSR to disable all interrupts
        ORR     R1,R0,#NO_INT
        MSR     CPSR_c,R1
        MRS     R1,CPSR         ; Confirm that CPSR contains the proper int. disable flags
        AND     R1,R1,#NO_INT
        CMP     R1,#NO_INT
        BNE     OS_CPU_SR_Save ; Not properly disabled (try again)
        MOV     PC,LR           ; Disabled, return the original CPSR contents in R0
```

3.04.02 OS_CPU_A.S, 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`).

Listing 3-19, `OS_CPU_SR_Restore()`

```
        CODE32

OS_CPU_SR_Restore
        MSR     CPSR_c,R0
        MOV     PC,LR
```

3.04.03 OS_CPU_A.S, 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 3-20, `OSStartHighRdy()`

```
        CODE32                          ; (1)

OSStartHighRdy

        MSR     CPSR_cxsf,#0xDF         ; (2) Switch to SYS mode with IRQ & FIQ disabled

        BL      OSTaskSwHook            ; (3) Call user defined task switch hook

        LDR     R4,??OS_Running         ; (4) OSRunning = TRUE
        MOV     R5,#1
        STRB    R5,[R4]

        LDR     R4,??OS_TCBHighRdy      ; (5) Get highest priority task TCB address
        LDR     R4,[R4]                 ;     get stack pointer
        LDR     SP,[R4]                 ;     switch to the new stack

        LDMFD   SP!,{R4}                ; (6) pop new task's CPSR
        MSR     CPSR_cxsf,R4
        LDMFD   SP!,{R0-R12,LR,PC}      ; (7) pop new task's r0-r12,lr & pc
```

L3-20(1) `CODE32` is an assembler directive that indicates that the code to follow is ARM code (`CODE16` would indicate Thumb code).

L3-20(2) The IAR compiler startup code sets the mode to SYS mode prior to calling `main()`. However, other compilers may not start in SYS mode so we decided to ensure that we are in SYS mode by changing the mode accordingly. Interrupts should not be enable at this point but, just to make sure, we disable them.

L3-20(3) Before starting the highest priority task, we call `OSTaskSwHook()` in case a hook call has been declared.

L3-20(4) 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
```

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.

L3-20(5) 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).

L3-20(6) We then pop the CPSR from the task's stack. Recall that when the task was created, the CPSR register on the stack frame was initialized with ARM_SYS_MODE or, 0x0000001F. Because OSStartHighRdy() already executes in SYS mode, loading the CPSR with the same value will not change the mode of the processor.

L3-20(7) The LDMFD instruction stands for "**L**oa**D** **M**ultiple **F**ull **D**ecendant" and means that multiple registers are popped off the stack with a single instruction. 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.

3.04.04 OS_CPU_A.S, 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.). The code below is described 'graphically' in an accompanying document (**AN-1011-PPT.PDF**) which was created with Microsoft's Power Point.

Recall that all tasks run in SYS mode. A task level context switch simply consist of saving the SYS registers on the task to suspend and restore the SYS 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) */
```

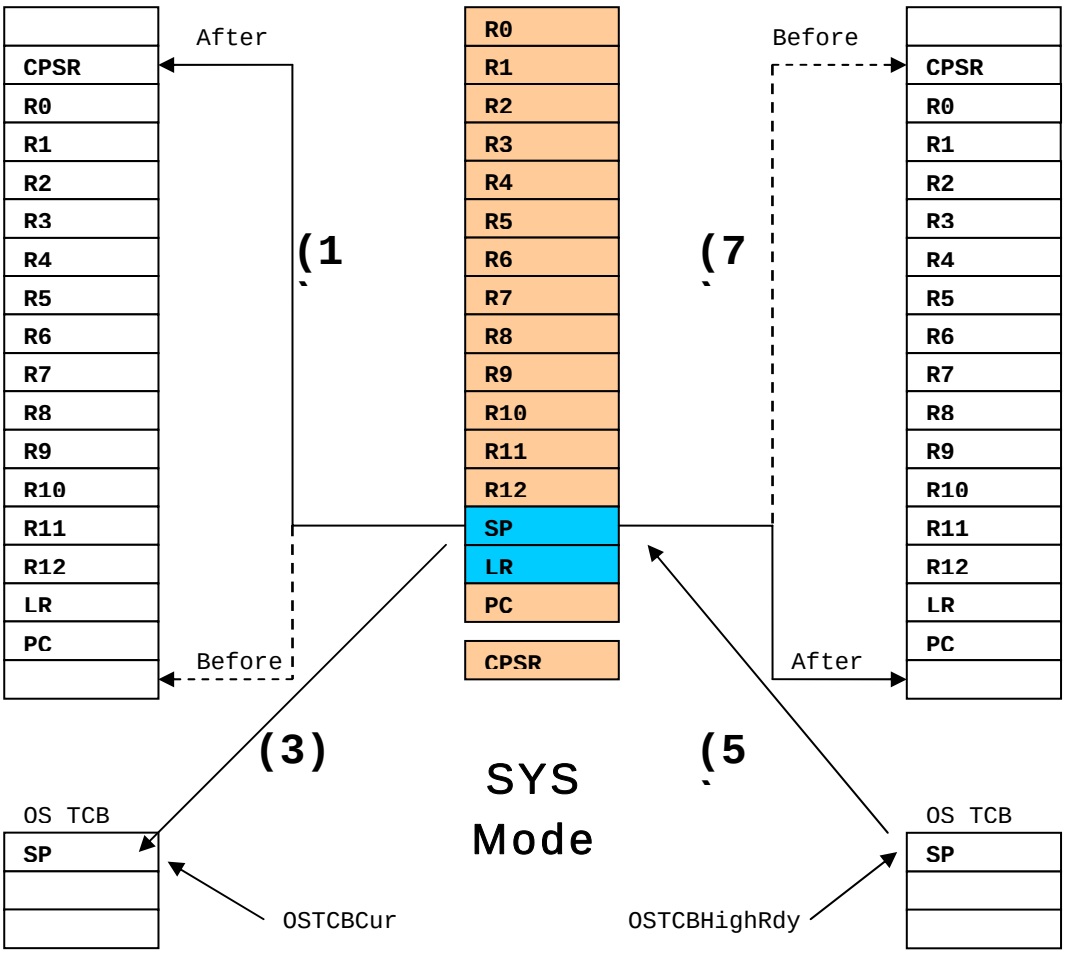

Figure 3-2, Task Level Context Switch.

The actual code for the task level context switch is shown in Listing 3-21. Refer to **AN-1011-PPT.PDF** for a description of the steps.

Listing 3-21, `OSCtxSw()`

```
        CODE32

OSCtxSw
        STMFD   SP!,{LR}                ; push PC (LR should be pushed in place of PC)
        STMFD   SP!,{R0-R12,LR}         ; push LR & register file
        MRS     R4,CPSR
        STMFD   SP!,{R4}                ; push current CPSR

        LDR     R4,??OS_TCBCur          ; Get current task's OS_TCB address
        LDR     R5,[R4]
        STR     SP,[R5]                 ; store SP in preempted tasks's TCB

        BL      OSTaskSwHook            ; call Task Switch Hook

        LDR     R4,??OS_PrioCur         ; OSPrioCur = OSPrioHighRdy
        LDR     R5,??OS_PrioHighRdy
        LDRB    R6,[R5]
        STRB    R6,[R4]

        LDR     R4,??OS_TCBCur          ; Get current task's OS_TCB address
        LDR     R6,??OS_TCBHighRdy      ; Get highest priority task's OS_TCB address
        LDR     R6,[R6]
        STR     R6,[R4]                 ; OSTCBCur = OSTCBHighRdy

        LDR     SP,[R6]                 ; get new task's stack pointer

        LDMFD   SP!,{R4}                ; pop new task's CPSR
        MSR     CPSR_cxsf,r4
        LDMFD   SP!,{R0-R12,LR,PC}      ; pop new task's R0-R12,LR & PC
```

3.04.05 OS_CPU_A.S, OSIntCtxSw()

When an ISR 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()`)..

Listing 3-22, `OSIntCtxSw()`

```
        CODE32

OSIntCtxSw
        BL      OSTaskSwHook            ; call Task Switch Hook

        LDR     R4,??OS_PrioCur         ; OSPrioCur = OSPrioHighRdy
        LDR     R5,??OS_PrioHighRdy
        LDRB    R6,[R5]
        STRB    R6,[R4]

        LDR     R4,??OS_TCBCur          ; Get current task's OS_TCB address
        LDR     R6,??OS_TCBHighRdy      ; Get highest priority task's OS_TCB address
        LDR     R6,[R6]
        STR     R6,[R4]                 ; OSTCBCur = OSTCBHighRdy

        LDR     SP,[R6]                 ; get new task's stack pointer

        LDMFD   SP!,{R4}                ; pop new task's CPSR
        MSR     CPSR_cxsf,r4
        LDMFD   SP!,{R0-R12,LR,PC}      ; pop new task's R0-R12,LR & PC
```

3.04.06 OS_CPU_A.S, OS_CPU_IRQ_ISR()

The ISR entry point for both IRQ and FIQ are part of the µC/OS-II port to reduce the amount of work needed by the programmer that's integrating µC/OS-II in his or her product.

In fact, the ISR code is 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 code for `OS_CPU_IRQ_ISR()` is shown in listing 3-23. The ISR is written in assembly language because we simply can't manipulate CPU registers directly from C. The code in Listing 3-23 is described graphically in a Power Point presentation, in the file **AN-1011-PPT.PDF**.

Listing 3-23, OS_CPU_IRQ_ISR()

```
OS_CPU_IRQ_ISR
        STMFD   SP!,{R1-R3}                     ; PUSH WORKING REGISTERS ONTO IRQ STACK

        MOV     R1,SP                           ; Save   IRQ stack pointer

        ADD     SP,SP,#12                       ; Adjust IRQ stack pointer

        SUB     R2,LR,#4                        ; Adjust PC for return address to task

        MRS     R3,SPSR                         ; Copy SPSR (i.e. interrupted task's CPSR)

        MSR     CPSR_c,#(NO_INT | SYS32_MODE) ; Change to SYS mode

                                                ; SAVE TASK'S CONTEXT ONTO TASK'S STACK
        STMFD   SP!,{R2}                        ;    Push task's Return PC
        STMFD   SP!,{R4-R12,LR}                 ;    Push task's LR,R12-R4

        LDMFD   R1!,{R4-R6}                     ;    Move R1-R3 from IRQ stack to SYS stack
        STMFD   SP!,{R4-R6}
        STMFD   SP!,{R0}                        ;    Push task's R0    onto task's stack
        STMFD   SP!,{R3}                        ;    Push task's CPSR (i.e. IRQ's SPSR)

                                                ; HANDLE NESTING COUNTER
        LDR     R0,??OS_IntNesting              ; OSIntNesting++;
        LDRB    R1,[R0]
        ADD     R1,R1,#1
        STRB    R1,[R0]

        CMP     R1,#1                           ; if (OSIntNesting == 1) {
        BNE     OS_CPU_IRQ_ISR_1

        LDR     R4,??OS_TCBCur                  ;     OSTCBCur->OSTCBStkPtr = SP
        LDR     R5,[R4]
        STR     SP,[R5]                         ; }

OS_CPU_IRQ_ISR_1
        MSR     CPSR_c,#(NO_INT | IRQ32_MODE) ; Change to IRQ mode

        BL      OS_CPU_IRQ_ISR_Handler          ; OS_CPU_IRQ_ISR_Handler();

        MSR     CPSR_c,#(NO_INT | SYS32_MODE) ; Change to SYS mode

        BL      OSIntExit                       ; OSIntExit();

                                                ; RESTORE TASK'S CONTEXT and RETURN TO TASK
        LDMFD   SP!,{R4}                        ; pop new task's CPSR
        MSR     CPSR_cxsf,r4
        LDMFD   SP!,{R0-R12,LR,PC}              ; pop new task's R0-R12,LR & PC
```

You should note that MOST of the work done by the ISR is actually handled in `OS_CPU_IRQ_ISR_Handler()` which is written in C. The pseudo-code for `OS_CPU_IRQ_ISR_Handler()` is shown in listing 3-24. The handler is responsible for determining the source of the interrupt and for executing the appropriate code to handle the interrupting device.

Listing 3-24, `Your_ISR_Handler()`

```
void  OS_CPU_IRQ_ISR_Handler (void)
{
    while (there are interrupting devices) {
        /* Clear interrupting device */
        /* Handle interrupt          */
    }
}
```

`OS_CPU_IRQ_ISR_Handler()` 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_IRQ_ISR()`. This avoids going through the overhead of saving the CPU registers upon entry of the ISR and restoring them upon exit if multiple interrupts occur either at the same time or, during processing of an interrupt.

Another important thing to do is to **NOT** enable CPU interrupts by clearing the I-bit in the `CPSR` register. In other words, **ALWAYS** execute the interrupt handler with interrupts **DISABLED**. The reason is that we **DON'T** want to nest interrupts because there is no need to - the handler will check all possible interrupting devices before leaving.

Finally, as a general rule, you should always make your interrupt handlers as shorts 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.

3.04.07 OS_CPU_A.S, OS_CPU_FIQ_ISR()

The code for `OS_CPU_FIQ_ISR()` and `OS_CPU_FIQ_ISR_Handler()` are almost identical to that of the IRQ. Of course, each has its own entry point and its own handler.

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 seven (7) 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
-	-	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_IRQ_ISR(). However, it's possible that the code for OS_CPU_IRQ_ISR() 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_IRQ_ISR' 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_IRQ_ISR(). This allows the interrupt service routine to be placed anywhere withing 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_IRQ_ISR()**
		0x0000003C	**Address of OS_CPU_FIQ_ISR()**

Table 4-2, Interrupt Vectors

If you are debugging your code in RAM, the easiest way to ensure that these 'opcodes' are placed at those locations is to write the following code in the initialization of your application:

Listing 4-1, Installing the interrupt vectors in RAM

```
*(INT32U *)0x00000018 = 0xE59FF018;           /* ldr pc,[pc,#+0x18] at 0x00000018  */
*(INT32U *)0x0000001C = 0xE59FF018;           /* ldr pc,[pc,#+0x18] at 0x0000001C  */

*(INT32U *)0x00000038 = (INT32U)OS_CPU_IRQ_ISR;    /* Address of OS_CPU_IRQ_ISR()  */
*(INT32U *)0x0000003C = (INT32U)OS_CPU_FIQ_ISR;    /* Address of OS_CPU_FIQ_ISR()  */
```

Of course, if you have Flash (or ROM) at location `0x00000000` then, you can simply include the following code:

Listing 4-2, Setting up the interrupt vectors in Flash (or ROM)

```
COMMON  INTVEC:CODE:ROOT(2)
CODE32
ORG     0x00000018
LDR     PC,[PC,#0x18]    ; Vector to the proper ISR for the AT91 using the AIC.

ORG     0x0000001C
LDR     PC,[PC,#0x18]

DATA
ORG     0x00000038

DC32    OS_CPU_IRQ_ISR
DC32    OS_CPU_FIQ_ISR
```

4.01 Interrupt 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_IRQ_ISR()` because of the `LDR PC,[PC,#0x18]` instruction that we placed at address `0x00000018`.
The CPU executes the code in `OS_CPU_IRQ_ISR()` (found in `OS_CPU_A.S`).
`OS_CPU_IRQ_ISR()` calls `OS_CPU_IRQ_ISR_Handler()` to determine the source of the interrupt and handle it accordingly.
When `OS_CPU_IRQ_ISR()` returns from `OS_CPU_IRQ_ISR_Handler()` (found in `BSP.C`), `OS_CPU_IRQ_ISR()` calls `OSIntExit()` which determines whether there has been a more important task that has been made ready to run by the ISR 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_IRQ_ISR()` 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 family 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 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_IRQ_ISR_Handler()` can thus be written as shown in listing 4-3.

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

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


typedef void (*PFNCT)(void);


void OS_CPU_IRQ_ISR_Handler (void)
{
    PFNCT  pfnct;


    pfnct = (PFNCT)AIC_IVR;         /* Read the interrupt vector from the AIC  */
    while (pfnct != (PFNCT)0) {     /* Handle ALL interrupting devices         */
      (*pfnct)();                   /* Call ISR for interrupting device        */
        pfnct = (PFNCT)AIC_IVR;     /* Read the interrupt vector from the AIC  */
    }
}
```

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_IRQ_ISR_Handler()` to work properly. You **DO NOT** want to place the address of `OS_CPU_IRQ_ISR()` as the ISR address for the AIC.

Your ISR handlers should be written as follows:

```
void  MyISR_Handler (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)    */
}
```

The code for `OS_CPU_FIQ_ISR_Handler()` is similar to the IRQ handler described above.

4.02.02 Interrupt Controllers, Philips and Sharp's VIC

The Philips LPC2000 series (ARM7) and Sharp ARM7 family 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`.

Similarly, the address of the ISR for the FIQ interrupting device is found at address `0xFFFFF034`.

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

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

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

typedef void (*PFNCT)(void);


void OS_CPU_IRQ_ISR_Handler() (void)
{
    PFNCT  pfnct;

    pfnct = (PFNCT)VIC_IRQ;         /* Read the interrupt vector from the VIC  */
    while (pfnct != (PFNCT)0) {     /* Handle ALL interrupting devices         */
      (*pfnct)();                   /* Call ISR for interrupting device        */
        pfnct = (PFNCT)VIC_IRQ;     /* Read the interrupt vector from the VIC  */
    }
}
```

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_IRQ_ISR_Handler()` to work properly. You **DO NOT** want to place the address of `OS_CPU_IRQ_ISR()` as the ISR address for the VIC.

Your ISR handlers should be written as follows:

```
void  MyISR_Handler (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)    */
}
```

Of course, the code is similar for the FIQ interrupt.

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_UNDEF_INSTRUCTION_VECTOR_ADDR   (*(INT32U *)0x00000004L)
#define  BSP_SWI_VECTOR_ADDR                 (*(INT32U *)0x00000008L)
#define  BSP_PREFETCH_ABORT_VECTOR_ADDR      (*(INT32U *)0x0000000CL)
#define  BSP_DATA_ABORT_VECTOR_ADDR          (*(INT32U *)0x00000010L)
#define  BSP_IRQ_VECTOR_ADDR                 (*(INT32U *)0x00000018L)          (1)
#define  BSP_FIQ_VECTOR_ADDR                 (*(INT32U *)0x0000001CL)

#define  BSP_IRQ_ISR_ADDR                    (*(INT32U *)0x00000038L)          (2)
#define  BSP_FIQ_ISR_ADDR                    (*(INT32U *)0x0000003CL)

#define  NIVECSR                             (*(INT32U *)0x00223040L)          (3)
#define  FIVECSR                             (*(INT32U *)0x00223044L)
```

L4-5(1) This specifies the location of the IRQ and FIQ interrupt vectors.

L4-5(2) These two memory locations are used to hold the address of `OS_CPU_IRQ_ISR()` and `OS_CPU_FIQ_ISR()`, respectively. This allows us to be able to locate these two functions anywhere in the 32-bit address space of the ARM9 processor.

L4-5(3) These are the addresses of the `NIVECSR` and `FIVECSR` registers, respectively.

Listing 4-6, Data Types

```
typedef  void (*BSP_FNCT_PTR)(void);                                          (1)
```

L4-6(1) This declares a new data type for a pointer to a function.

Listing 4-7, ISR Address Table

```
BSP_FNCT_PTR    BSP_IntVectTbl[64];                                           (1)
```

L4-7(1) 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 `NIVECSR` for an IRQ and the `FIVECSR` for an FIQ. We would use this index to extract the address of the ISR from this table (see `OS_CPU_IRQ_ISR_Handler()` for details).

Listing 4-8, Unused ISR Handler

```
static  void  BSP_ISR_Handler_Dummy (void)                                  (1)
{
}
```

L4-8(1) Here we declare a 'dummy' function in order to populate the interrupt vector table (i.e. `BSP_IntVectTbl[]`) with a pointer to this function. This is used in case there is no ISR associated with an interrupting device.

Listing 4-9, Initialization of the Interrupt Vector Table

```
static  void  BSP_IntCtrlInit (void)
{
    INT16U  i;


    BSP_UNDEF_INSTRUCTION_VECTOR_ADDR = 0xEAFFFFFE;               (1)
    BSP_SWI_VECTOR_ADDR               = 0xEAFFFFFE;
    BSP_PREFETCH_ABORT_VECTOR_ADDR    = 0xEAFFFFFE;
    BSP_DATA_ABORT_VECTOR_ADDR        = 0xEAFFFFFE;

    BSP_IRQ_VECTOR_ADDR               = 0xE59FF018;               (2)
    BSP_IRQ_ISR_ADDR                  = (INT32U)OS_CPU_IRQ_ISR;

    BSP_FIQ_VECTOR_ADDR               = 0xE59FF018;               (3)
    BSP_FIQ_ISR_ADDR                  = (INT32U)OS_CPU_FIQ_ISR;

    for (i = 0; i < 64; i++) {                                    (4)
        BSP_IntVectTbl[i] = BSP_ISR_Handler_Dummy;
    }
}
```

L4-9(1) Here we assume that locations `0x00000000` to `0x0000003F` contain RAM. We setup a 'jump to itself' instruction for the appropriate exception handlers because these handlers are not used. Of course, if you have code to handle these exceptions, you would replace these with the appropriate code.

L4-9(2) At location `0x00000018` we 'force' the opcode for the `LDR PC,[PC,#0x18]` such that when the CPU recognizes an IRQ interrupt, it will load the contents of location `0x00000038` into the PC (i.e. the address of `OS_CPU_IRQ_ISR()`).

L4-9(3) At location `0x0000001C` we 'force' the opcode for the `LDR PC,[PC,#0x18]` such that when the CPU recognizes an FIQ interrupt, it will load the contents of location `0x0000003C` into the PC (i.e. the address of `OS_CPU_FIQ_ISR()`).

L4-9(4) We initialize the table containing the addresses of the ISR for each interrupting device. When you want the CPU to service a specific device, you would simply 'install' the ISR handler by calling `BSP_IntVectSet()` as described in Listing 4-10.

Listing 4-10, Specifying the Address of an ISR

```
void  BSP_IntVectSet (INT32U int_nbr, BSP_FNCT_PTR  pisr)              (1)
{
    if (int_nbr < 64) {                                                (2)
        BSP_IntVectTbl[int_nbr] = pisr;                                (3)
    }
}
```

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

```
void MyISRHandler (void)
{
    Handle the device that generated the interrupt.
    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 index between 0 and 63, inclusively.

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

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

```
void  OS_CPU_IRQ_ISR_Handler (void)
{
    INT16U        int_vect;
    BSP_FNCT_PTR  pfnct;


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

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

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

L4-11(3) If we have a valid index, we obtain the address of the ISR 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 ISR handler.

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

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

Listing 4-12, OS_CPU_FIQ_ISR_Handler() for the Freescale's AITC

```
void  OS_CPU_FIQ_ISR_Handler (void)
{
    INT16U        int_vect;
    BSP_FNCT_PTR  pfnct;


    int_vect = (FIVECSR >> 16) & 0x00FF;     /* determine highest pending FIQ    */
    while (int_vect < 64) {
        pfnct = BSP_IntVectTbl[int_vect];    /* find the pointer to the ISR      */
        if (pfnct != (BSP_FNCT_PTR)0) {      /* Make sure it's been initialized  */
            pfnct();
        }
        int_vect = (FIVECSR >> 16) & 0x00FF; /* determine highest pending IRQ    */
    }
}
```

5.00 Debugging in RAM

A large number of ARM7 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.

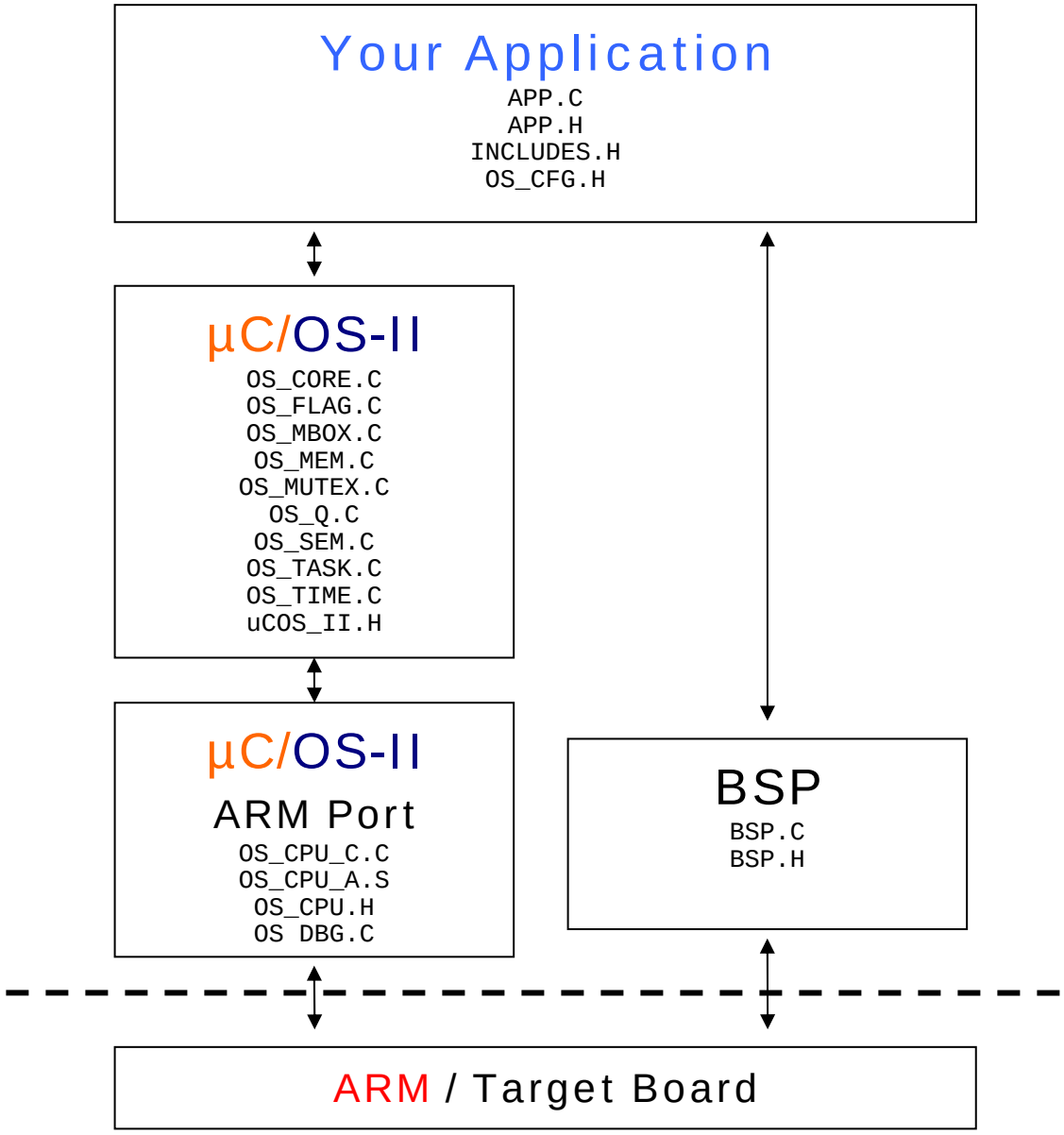

Figure 6-1, Relationship between modules.

6.01 APP.C and APP.H

For sake of discussion, your application is placed in files called `APP.C` and `APP.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.H` contains `#define` constants, macros, prototypes, etc. that are specific to your application.

`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()`

```
void main (void)
{
    INT8U  err;


    BSP_IntDisAll();

    OSInit();                                                          (1)

    OSTaskCreateExt(AppStartTask,                                      (2)
                    (void *)0,
                    (OS_STK *)&AppStartTaskStk[TASK_STK_SIZE-1],
                    TASK_START_PRIO,
                    TASK_START_PRIO,
                    (OS_STK *)&AppStartTaskStk[0],
                    TASK_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

#if OS_TASK_NAME_SIZE > 11
    OSTaskNameSet(TASK_START_PRIO, "Start Task", &err);                (3)
#endif

#if OS_TASK_NAME_SIZE > 14
    OSTaskNameSet(OS_IDLE_PRIO, "uC/OS-II Idle", &err);                (4)
#endif

#if (OS_TASK_NAME_SIZE > 14) && (OS_TASK_STAT_EN > 0)
    OSTaskNameSet(OS_STAT_PRIO, "uC/OS-II Stat", &err);
#endif

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

L6-1(4) 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(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  AppStartTask (void *p_arg)
{
    (void)p_arg;

    BSP_Init();                                          (1)

#if OS_TASK_STAT_EN > 0
    OSStatInit();                                        (2)
#endif

#if OS_VIEW_MODULE > 0
    OSView_Init(19200);                 /* Initialize uC/OS-View if module is present */
    OSView_TerminalRxSetCallback(AppTerminalRx);
#endif

    AppTaskCreate();                                     (3)

    while (TRUE) {
        /* Do something 'useful' in this task */         (4)
        LED_Toggle(1);                                   (5)
        OSTimeDly(OS_TICKS_PER_SEC / 10);
    }
}
```

L6-2(1) If you decided to implement a BSP (see section 6.03, Board Support Package) for your target board, you would initialize it here.

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 ISR 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 5 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    <stdio.h>
#include    <string.h>
#include    <ctype.h>
#include    <stdlib.h>

#include    <ucos_ii.h>
#include    <bsp.h>
#include    <app.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
ISR Handlers
LED control functions
Reading switches
Setting up the interrupt controller
Setting up communication channels
Etc.

A BSP consist of 3 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 `#defines`. 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 ISRs 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 ISR handler is found in `BSP.C` and is called. `Tmr_TickISR_Handler()`.

It's assumed that the ISR handlers (`OS_CPU_IRQ_ISR_Handler()` and `OS_CPU_FIQ_ISR_Handler()`) are declared in `BSP.C` (see section 4 for details).

8.00 Conclusion

This application note presented a 'generic' port 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 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

µC/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