Micriμm

μC/OS-II

and

ARM Cortex-M4 Processor

with the

Kinetis K60

Application Note

Kinetis K60_OSII

www.Micrium.com

Table of Contents

1.00 Introduction

ARM has been working on a new architecture called the Cortex for a number of years. During development, **µC/OS-II** was used to validate some of the design aspects and was used as a source of ideas to create new capabilities to support RTOSs. In other words, **µC/OS-II** was the first RTOS ported to the Cortex.

This application note describes the 'official' Micrium port for **µC/OS-II** on the Cortex-M4 processor. 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.

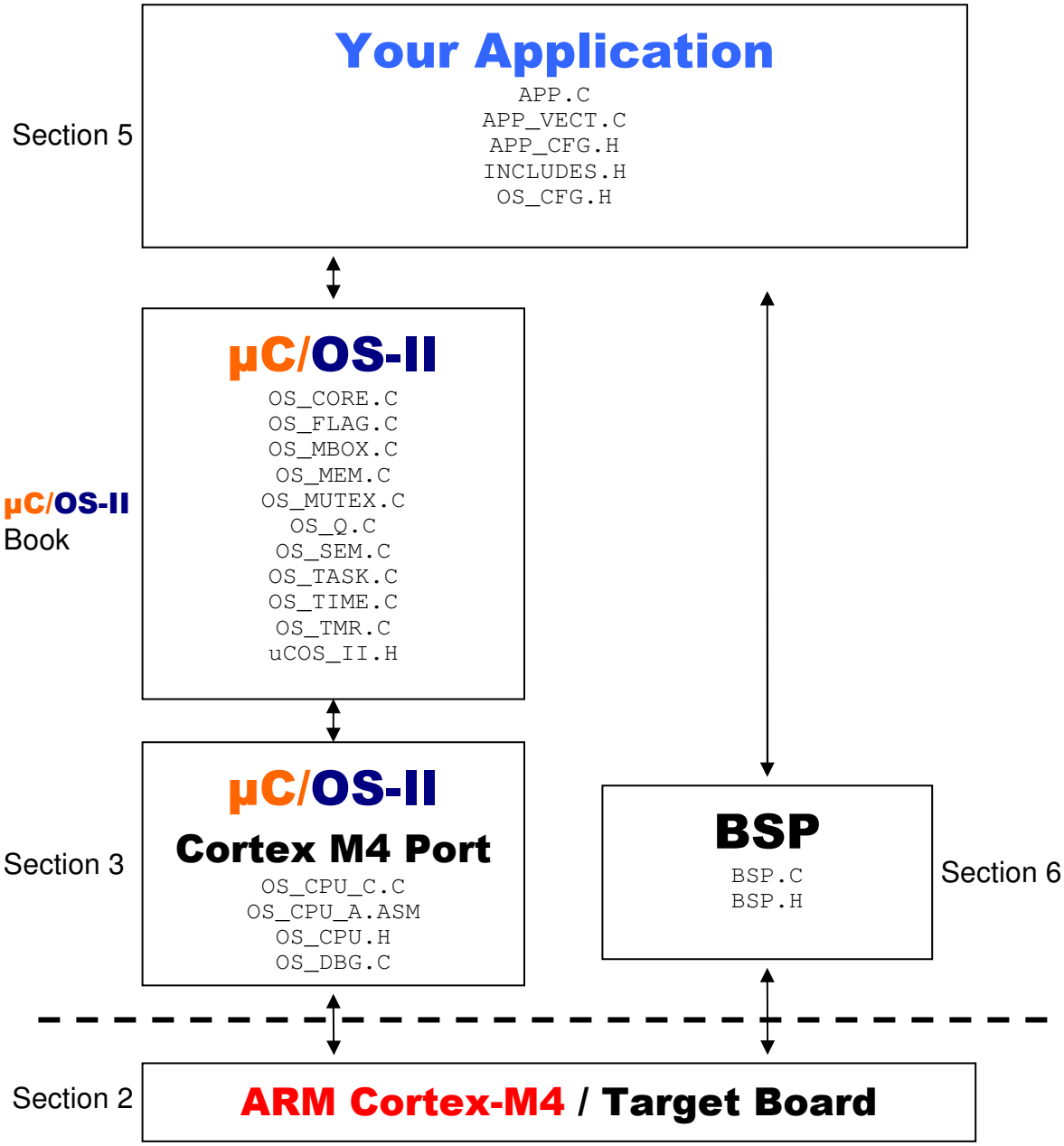


Figure 1-1, Relationship between modules.

2.00 The ARM Cortex-M4 programmer's model

The visible registers in an ARM Cortex-M4 processor are shown in Figure 2-1. The ARM Cortex-M4 has a total of 20 registers. Each register is 32 bits wide.

`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. There are actually two stack pointers (`SP_process` and `SP_main`) but only one is visible at any given time. `SP_process` is used for task level code and `SP_main` is used for exception processing.

`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.

`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 or 4 depending on the instruction.

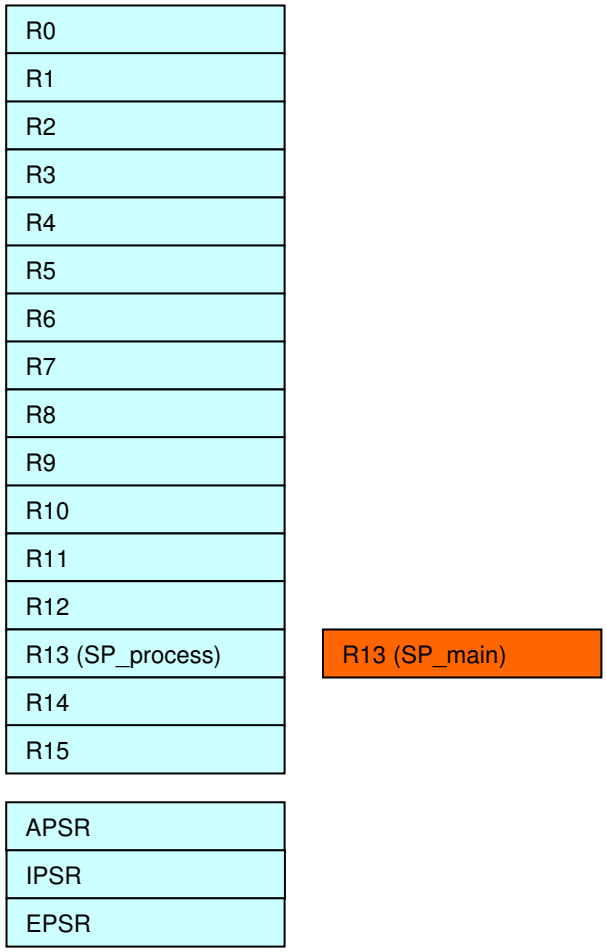


Figure 2-1, ARM Cortex-M4 Register Model.

xPSR There are three separate registers to hold the sate of the CPU: APSR, IPSR and EPSR. The **APSR** contains application status such as shown in Figure 2-2.

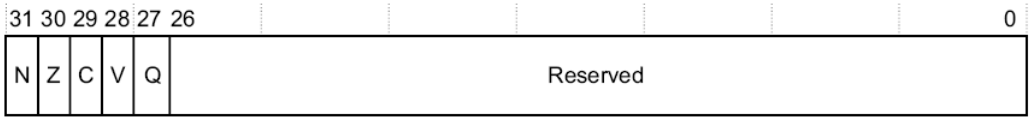


Figure 2-2, The APSR Register.

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.

Q
Bit 27 is the sticky saturation flag.

The Interrupt PSR (**IPSR**) contains the ISR number of the current exception activation and is shown in Figure 2-3.

Figure 2-3, The IPSR Register.

The Execution PSR (**EPSR**) contains two overlapping fields:

• the Interruptible-Continuable Instruction (ICI) field for interrupted load multiple and store multiple instructions

• the execution state field for the If-Then (IT) instruction, and the T-bit (Thumb state bit).

Figure 2-4, The EPSR Register.

On entering an exception, the processor saves the combined information from the three status registers (referred to as xPSR) onto the stack.

3.00 µC/OS-II Port for the ARM Cortex-M4 processors

We used the IAR EWARM V6.10 (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 Cortex-M4 simulator which allows you to test code prior to run it on actual hardware. We tested the ARM Cortex-M4 port on a Freescale TWR-K60N512 development board as shown in Figure 3-1.

Figure 3-1, Freescale TWR-K60N512 Development Kit (Kinetis K60 chip)

You can adapt the port provided in this application note to other ARM Cortex-M4 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.

The port assumes that you are using µC/OS-II V2.92 or higher.

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-Cortex-M4\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 Cortex-M4 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 Cortex-M4 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 xPSR) on the Cortex-M4 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


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

Note that if your application code uses these macros, you MUST allocate a local variable called '`cpu_sr`' and initialize it to `0`, as shown below:

```
    OS_CPU_SR  cpu_sr = 0;
```

3.02.04 OS_CPU.H, Stack growth

The stacks on the ARM Cortex-M4 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.ASM` (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.

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
```

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.

Listing 3-7, OS_CPU.H, Function Prototypes

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

void       OS_CPU_PendSVHandler(void);

void       OS_CPU_SysTickHandler(void);
void       OS_CPU_SysTickInit(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()
OSTaskReturnHook()
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.

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`).

Listing 3-8, OS_CPU_C.C, `OSInitHookEnd()`

```
void OSInitHookBegin (void)
{
#if OS_TMR_EN > 0
    OSTmrCtr = 0;
#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 application task create hook, `App_TaskCreateHook()`.

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

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

3.03.03 OS_CPU_C.C, OSTaskStkInit()

It is typical for ARM compilers (the Cortex-M4 also) to pass the first argument of a function into the `R0` register. Recall that a task is declared as shown in listing 3-10.

Listing 3-10, µC/OS-II Task

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

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

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


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

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

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

    return (stk);
}
```

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

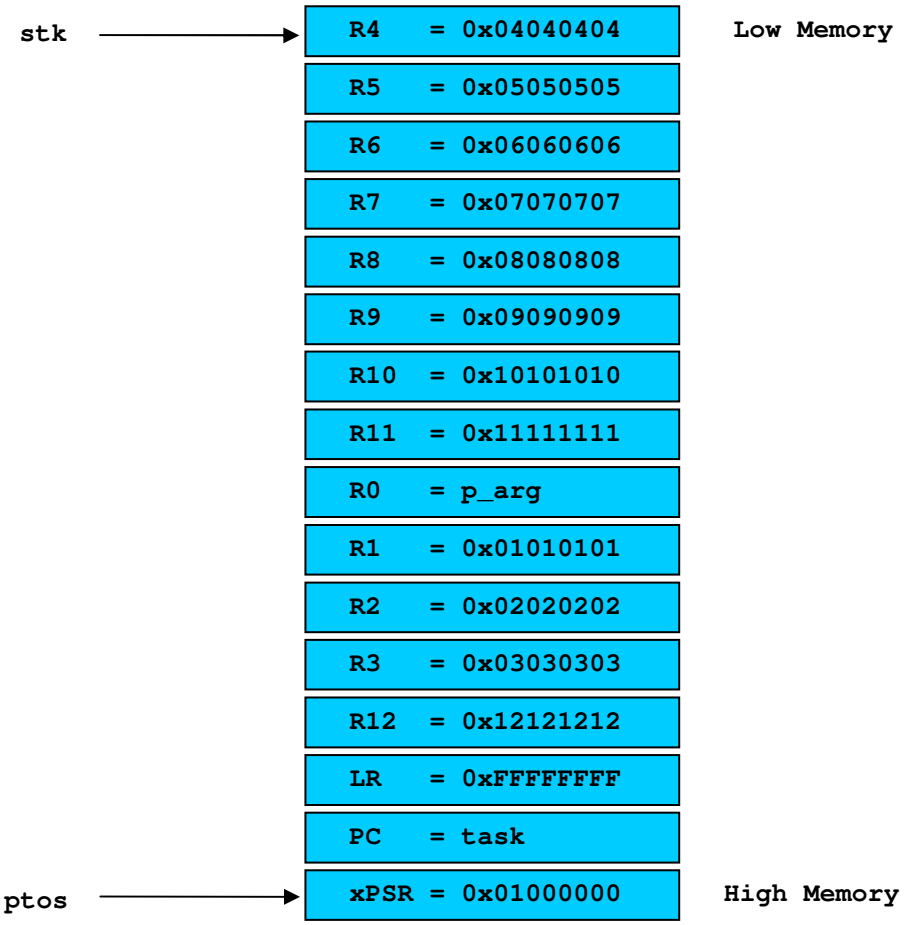


Figure 3-2, The Stack Frame for each Task for ARM Cortex-M4 port.

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()`). The ordering of the registers is important since it matches the way ARM Cortex-M4 stacks them on exception.

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, output a pulse on a port pin when a contact switch occurs, etc. In this case, we call the application task switch hook called `App_TaskSwHook()`.

Listing 3-12, OS_CPU_C.C, OSTaskSwHook()

```
void  OSTaskSwHook (void)
{
#if OS_APP_HOOKS_EN > 0
    App_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 application hook function `App_TimeTickHook()`.

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

Listing 3-13, 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
}
```

3.03.06 OS_CPU_C.C, OS_CPU_SysTickInit()

`OS_CPU_SysTickInit()` should be called by the first application task to intiailize the SysTick timer, which provides the **µC/OS-II** time tick. `OS_CPU_SysTickInit()` calls `OS_CPU_SysTickClkFreq()`, which the user must provide in the BSP, to get the processor clock frequency.

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

```
void  OS_CPU_SysTickInit (void)
{
    INT32U  cnts;


    cnts = OS_CPU_SysTickClkFreq() / OS_TICKS_PER_SEC;

    OS_CPU_CM4_NVIC_ST_RELOAD = (cnts - 1);
                                                            /* Enable timer.           */
    OS_CPU_CM4_NVIC_ST_CTRL  |= OS_CPU_CM4_NVIC_ST_CTRL_CLK_SRC
                             |  OS_CPU_CM4_NVIC_ST_CTRL_ENABLE;

                                                            /* Enable timer interrupt. */
    OS_CPU_CM4_NVIC_ST_CTRL  |= OS_CPU_CM4_NVIC_ST_CTRL_INTEN;
}
```

3.04 OS_CPU_A.ASM

A µC/OS-II port requires that you write five fairly simple assembly language functions. These functions are needed because you normally cannot save/restore registers from C functions. The five functions are:

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

The ARM Cortex-M4 uses a clever way to perform a context switch. This is done via a special exception handler which needs to be defined (it will be described later). The handler is:

```
OS_CPU_PendSVHandler()
```

3.04.01 OS_CPU_A.ASM, OS_CPU_SR_Save()

The code in listing 3-14 implements the saving of the interrupt mask register and then disabling interrupts to implement `OS_CRITICAL_METHOD` #3. This function is invoked by the `OS_ENTER_CRITICAL()` macro.

When this function returns, `R0` contains the state of the `PRIMASK` register which contains the global interrupt mask prior to disabling interrupts.

Listing 3-14, `OS_CPU_SR_Save()`

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

3.04.02 OS_CPU_A.ASM, OS_CPU_SR_Restore()

The code in the listing below implements the function to restore the interrupt disable mask to its original value prior to calling `OS_ENTER_CRITICAL()` (see previous section). In other words, if interrupts were disabled prior to calling `OS_ENTER_CRITICAL()`, they would be disabled after calling `OS_EXIT_CRITICAL()`.

Listing 3-15, `OS_CPU_SR_Restore()`

```
OS_CPU_SR_Restore
        MSR     PRIMASK,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 3-16, `OSStartHighRdy()`

```
OSStartHighRdy

        LDR     R0, =NVIC_SYSPRI14    ; (1) Set the PendSV exception priority
        LDR     R1, =NVIC_PENDSV_PRI
        STRB    R1, [R0]

        MOV     R0, #0                ; (2) Set PSP to 0 for initial context switch call
        MSR     PSP, R0

                                      ; (3)Initialize the MSP to the OS_CPU_ExceptStkBase
        LDR     R0, =OS_CPU_ExceptStkBase
        LDR     R1, [R0]
        MSR     MSP, R1

        LDR     R0, __OS_Running      ; (4) OSRunning = TRUE
        MOV     R1, #1
        STRB    R1, [R0]

        LDR     R0, =NVIC_INT_CTRL    ; (5) Trigger the PendSV exception
        LDR     R1, =NVIC_PENDSVSET
        STR     R1, [R0]

        CPSIE   I                     ; (6) Enable interrupts at processor level
```

L3-16(1) The ARM Cortex-M4 provides a special mechanism to perform a context switch. Specifically, the ARM Cortex-M4 provides a special exception handler called the PendSV (Pend Service call). The PendSV is basically an interrupt mechanism that is triggered called by software. It's like a software interrupt except that the interrupt is not taken until interrupts are enabled. This step sets the PendSV interrupt priority to the lowest priority.

L3-16(2) Here we setup the `PSP` stack pointer to run the very first task but by the PendSV handler. This is done by setting the `PSP` to 0 to inform the PendSV handler to not save the context of the task (because there is no task to save the context for since it will be the first task to run). It is assumed that `OSTCBHighRdy` contains the pointer to the `OS_TCB` of the task to start.

L3-16(3) Here we set the main stack to OS_CPU_ExceptStkBase

L3-16(4) Here we set `OSRunning` to `TRUE` to indicate that multitasking will start.

L3-16(5) We are ready to trigger the PendSV handler which will be starting the first task. The PendSV handler will run only when interrupts are enabled (see next step).

L3-16(6) Once interrupts are enabled the ARM Cortex-M4 processor will branch to the PendSV handler (described later).

3.04.04 OS_CPU_A.ASM, OSCtxSw()

When a task gives up control of the CPU, the `OS_TASK_SW()` macro is invoked (see `OS_CPU.H`) which is translated to a call to `OSCtxSw()`. Normally, `OSCtxSw()` performs a task level context switch but, on the ARM Cortex-M4, all context switching is deferred to the PendSV handler. `OSCtxSw()` thus simply triggers the PendSV handler and returns to the caller. The PendSV handler does not execute immediately because `OS_TASK_SW()` (and thus `OSCtxSw()`) is invoked with interrupts disabled. The PendSV handler will only execute when interrupts are re-enabled.

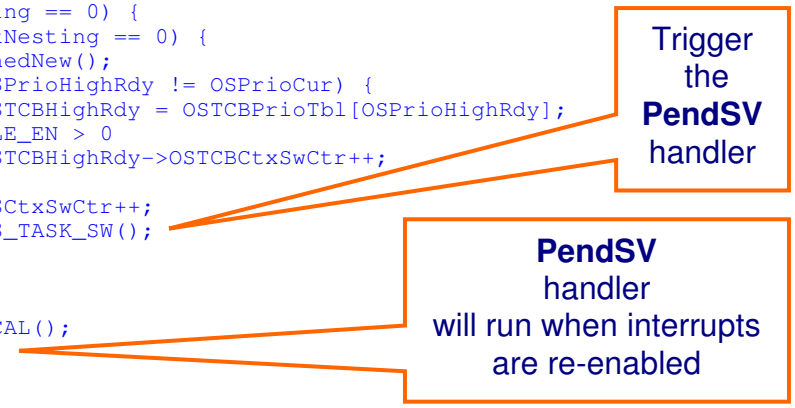
`OS_TASK_SW()` is always called from `OS_Sched()` (see `OS_CORE.C`). The current version of `OS_Sched()` is shown in Listing 3-17.

Listing 3-17, `OS_Sched()`

```
void  OS_Sched (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr = 0;
#endif



    OS_ENTER_CRITICAL();
    if (OSIntNesting == 0) {
        if (OSLockNesting == 0) {
            OS_SchedNew();
            if (OSPrioHighRdy != OSPrioCur) {
                OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
#if OS_TASK_PROFILE_EN > 0
                OSTCBHighRdy->OSTCBCtxSwCtr++;
#endif
                OSCtxSwCtr++;
                OS_TASK_SW();
            }
        }
    }
    OS_EXIT_CRITICAL();
}
```



The code for `OSCtxSw()` is shown in Listing 3-18. Again, all we do here is trigger the PendSV handler. Note that `OS_Sched()` sets `OSTCBHighRdy` to point to the `OS_TCB` of the task we wish to switch to.

Listing 3-18, `OSCtxSw()`

```
OSCtxSw
        LDR     R0, =NVIC_INT_CTRL      ; trigger the PendSV exception
        LDR     R1, =NVIC_PENDSVSET
        STR     R1, [R0]
        BX      LR
```

3.04.05 OS_CPU_A.ASM, 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()`. However, unlike other **µC/OS-II** ports where `OSIntCtxSw()` actually performs the context switch, `OSIntCtxSw()` for the ARM Cortex-M4 simply triggers the PendSV handler and returns as shown in Listing 3-19.

Listing 3-19, `OSIntCtxSw()`

```
OSCtxSw
        LDR     R0, =NVIC_INT_CTRL      ; trigger the PendSV exception
        LDR     R1, =NVIC_PENDSVSET
        STR     R1, [R0]
        BX      LR
```

3.04.06 OS_CPU_A.ASM, OS_CPU_PendSVHandler()

`OSPendSV()` is the PendSV handler which handles all context switching for **µC/OS-II**. This is a recommended method for performing context switches with the ARM Cortex-M4. This is because the ARM Cortex-M4 auto-saves half of the processor context on any exception, and restores those same registers upon return from exception. The PendSV handler thus only needs to save R4-R11 and adjust the stack pointers. Using the PendSV exception this way means that context saving and restoring uses an identical method whether it's initiated from a task or occurs due to an interrupt or exception.

Note that you must place a pointer to `OS_CPU_PendSVHandler()` in the exception vector table at vector location 14 (based of the vector table + 4 * 14 or, offset 56).

The pseudo-code for the PendSV handler is:

Listing 3-20, `OS_CPU_PendSVHandler()`

```
OS_CPU_PendSVHandler:
       if (PSP != NULL) {                                (1)
              Save R4-R11 onto task stack;                       (2)
              OSTCBCur->OSTCBStkPtr = SP;               (3)
       }
       OSTaskSwHook();                                   (4)
       OSPrioCur = OSPrioHighRdy;                        (5)
       OSTCBCur  = OSTCBHighRdy;                         (6)
       PSP       = OSTCBHighRdy->OSTCBStkPtr;            (7)
       Restore R4-R11 from new task stack;                       (8)
       Return from exception;                            (9)
```

L3-20(0) Note that when `OS_CPU_PendSVHandler()` is started by the CPU, the CPU automatically saves the `xPSR`, `PC`, `LR`, `R12` and `R0-R3` registers onto the task stack. After saving half the CPU registers onto the task's stack, the CPU switches stack pointer to use the `MSP`.

L3-20(1) Here we check to see if this is the `PSP` stack pointer is set to `NULL` or not. Recall that `OSStartHighRdy()` sets `PSP` to `NULL` to avoid saving the task's context when we start the first task.

L3-20(2) If `OS_CPU_PendSVHandler()` is actually triggered to perform a full task switch then we simply save the remaining registers (R4-R11) on the task's stack and not the ISR stack.

L3-20(3) Once the context of the task being switched out is saved, we simply save the task stack pointer (`PSP`) into that task's `OS_TCB`.

L3-20(4) We then call the µC/OS-II context switch hook (see `OS_CPU_C.C`).

L3-20(5) As with all µC/OS-II ports, we need to copy the new high priority into the current priority.

L3-20(6) Similarly, we need to copy `OSTCBHighRdy` into `OSTCBCur`.

L3-20(7) We then retrieve the current top-of-stack pointer of the task we now want to switch to. Recall that the top-of-stack pointer is saved in `OSTCBHighRdy->OSTCBStkPtr`. µC/OS-II always places `.OSTCBStkPtr` at the beginning of the `OS_TCB` , so there is no need to find the offset of the SP, since it is always at offset 0.

L3-20(8) We restore the context of the task to execute (i.e. its register values) from the task's stack frame.

L3-20(9) We then perform a return from exception which causes the ARM Cortex-M4 to restore `R3-R0`, `R12`, `LR`, `PC` and `xPSR` registers from the task's stack frame. At this point, we are running the new task.

The actual code for the `OS_CPU_PendSVHandler()` handler is shown in Listing 3-21. Note that the reference numbers in the comments correspond to the same elements in the pseudo-code of Listing 3-20.

Figure 3-3 shows the context switch graphically (again with the corresponding references).

You should note that interrupts are disabled at the beginning of the PendSV handler to ensure that the context switch is performed atomically. If an interrupt occurs while we perform the context switch, it will be serviced once the new task is restored.

Listing 3-21, `OSPendSV()`

```
OS_CPU_PendSVHandler                    ; (0) CPU saved xPSR, PC, LR, R12, R0-R3
        CPSID   I                       ;     Prevent interruption during context switch
        MRS     R0, PSP                 ; (1) PSP is process stack pointer
        CBZ     R0, OSPendSV_nosave     ;     Skip register save the first time

        SUB     R0, R0, #0x20           ; (2) Save remaining regs r4-11 on process stack
        STM     R0, {R4-R11}

        LDR     R1, __OS_TCBCur         ; (3) OSTCBCur->OSTCBStkPtr = SP;
        LDR     R1, [R1]
        STR     R0, [R1]                ;     R0 is SP of process being switched out

OSPendSV_nosave
        PUSH    {R14}                   ; (4) OSTaskSwHook();
        LDR     R0, __OS_TaskSwHook
        BLX     R0
        POP     {R14}

        LDR     R0, __OS_PrioCur        ; (5) OSPrioCur = OSPrioHighRdy
        LDR     R1, __OS_PrioHighRdy
        LDRB    R2, [R1]
        STRB    R2, [R0]

        LDR     R0, __OS_TCBCur         ; (6) OSTCBCur  = OSTCBHighRdy;
        LDR     R1, __OS_TCBHighRdy
        LDR     R2, [R1]
        STR     R2, [R0]

        LDR     R0, [R2]                ; (7) R0 is new task SP
                                        ;     SP = OSTCBHighRdy->OSTCBStkPtr;

        LDM     R0, {R4-R11}            ; (8) Restore R4-R11 from new task stack
        ADD     R0, R0, #0x20
        MSR     PSP, R0                 ;     Load PSP with new task SP
        ORR     LR, LR, #0x04           ;     Ensure exception return uses process stack
        CPSIE   I
        BX      LR                      ; (9) Exception return
```

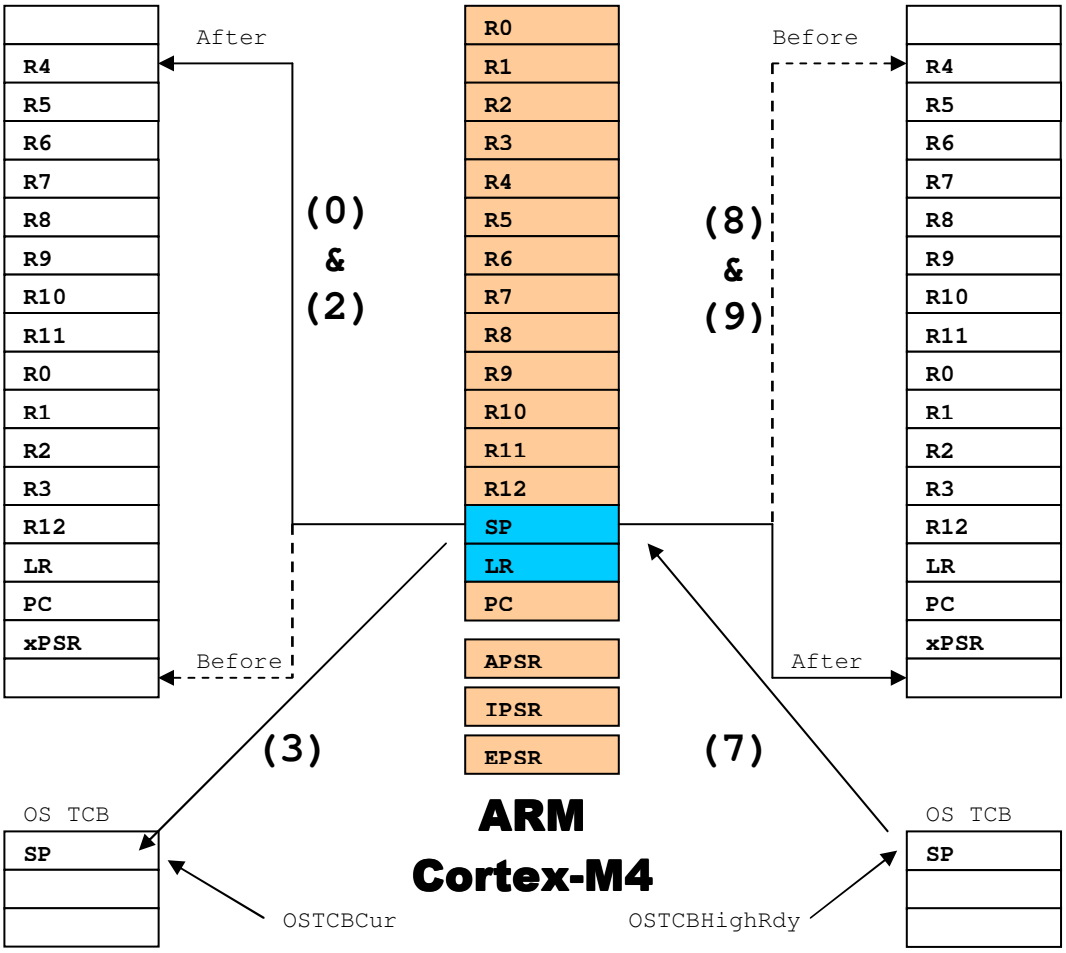


Figure 3-3, ARM Cortex-M4 Context Switch.

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 IAR's C-Spy debugger, Micriμm has introduced a Windows-based 'Plug-In' module that makes use of this file and thus needs to be included if you use C-Spy.

4.00 Exception Vector Table

The ARM Cortex-M4 contains an exception vector table (also called the interrupt vector table) starting at address `0x00000000`. The table can contain up to 256 entries (can be up to 1 Kbytes since each entry is a 32-bit pointer). Each entry in the table is a pointer to the corresponding exception or interrupt handler.

The exception vector table for the ARM Cortex-M4 is shown in table 4-1:

Position	Exception / Interrupt	Priority	Vector Address
0		-	0x00000000
1	Reset	-3 (highest)	0x00000004
2	Non-maskable Interrupt	-2	0x00000008
3	Hard Fault	-1	0x0000000C
4	Memory Management	settable	0x00000010
5	Bus Fault	Settable	0x00000014
6	Usage Fault	Settable	0x00000018
7	Reserved	-	0x0000001C
8	Reserved	-	0x00000020
9	Reserved	-	0x00000024
10	Reserved	-	0x00000028
11	SVCall	Settable	0x0000002C
12	Debug Monitor	Settable	0x00000030
13	Reserved	-	0x00000034
14	**PendSV**	Settable	0x00000038
15	**SysTick**	Settable	0x0000003C
16	INTSIR[239]	Settable	0x00000040
17	INTISR[238]	Settable	0x00000044
:	:	Settable	:
:	:	Settable	:
255	INTISR[0]	Settable	0x000003FC

Table 4-1, ARM Cortex-M4 Exception Vector Table

µC/OS-II uses the PendSV handler for context switching and the SysTick handler to process system ticks (i.e. clock ticks). The PendSV handler disables interrupts so that it can execute atomically.

The ARM Cortex-M4 has a built-in timer which was designed specifically for RTOS use. The timer can be configured to run at just about any tick rate. The application's BSP should set this timer to `OS_TICKS_PER_SEC`.

Note that it's up to the application code to setup the Exception Vector Table. To help you with this task, we created a file called `APP_VECT.C` that you can edit for each project.

4.01 Exception / Interrupt Handling Sequence

When the CPU invokes an exception or interrupt handler, the CPU automatically pushes the xPSR, PC, LR, R12 and R0-R3 registers onto the SP_process stack.

The CPU then reads the vector table to extract the address of the exception/interrupt handler and updates the PC with this address. The CPU builds the exception stack frame which includes the old PC. The LR actually gets a special value that looks something like `0xFFFFFFF9`. This means it is in handler mode, and when CM-3 sees this value attempt to load into the PC (as in `BX LR`), it recognizes that as an exception return and gets the PC from the registers saved when the exception was entered. The CPU then switches to use the `SP_main` stack pointer.

4.02 Interrupt Controllers

The ARM Cortex-M4 also comes with an integrated Nestable Vectored Interrupt Controller (NVIC).

4.03 Interrupt Service Routines

Interrupt Service Routines (ISRs) that need to use **µC/OS-II** services should be written as shown in Listing 4-1 for the ARM Cortex-M4.

Listing 4-1, Interrupt Service Routines using µC/OS-II services.

```
void OS_CPU_IRQ_ISR_Handler (void)
{
    OS_CPU_SR  cpu_sr = 0;


    OS_ENTER_CRITICAL();    /* Tell uC/OS-II that we are starting an ISR          */
    OSIntNesting++;
    OS_EXIT_CRITICAL();

    /* Handle the Interrupt … don't forget to clear the interrupt source          */

    OSIntExit();            /* Tell uC/OS-II that we are leaving the ISR          */
}
```

You should note that you MUST disable interrupts in order to increment `OSIntNesting` to ensure that the operation is performed atomically. We do this by calling the `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()` macros.

It's possible that some ISRs don't need to signal a task. In those cases, your ISRs would not need to increment `OSIntNesting` and call `OSIntExit()`.

5.00 Application Code

Your application code can make use of the port presented in this application note as described in this section. Figure 5-1 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 Cortex-M4 CPU and the target hardware.

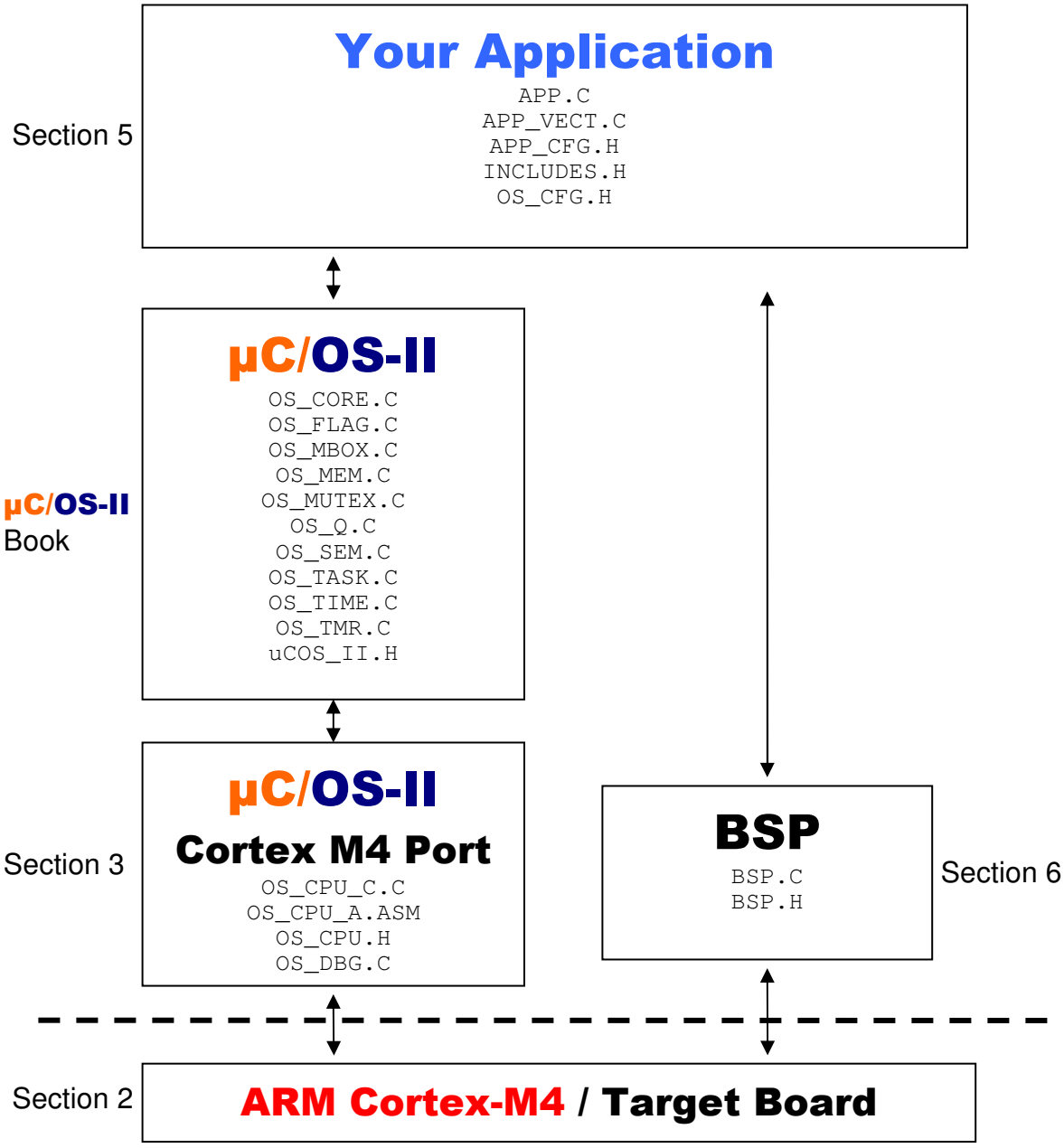


Figure 5-1, Relationship between modules.

5.01 APP.C, APP.H and APP_CFG.H

For sake of discussion, your application is placed in files called `APP.C`, `APP.H` 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_VECT.C` contains the exception / interrupt vector table for the application. You can edit this file to add your own interrupt handlers (or at least pointers to them). At vector 14, the vector table needs to point to `OS_CPU_PendSVHandler()` (see `OS_CPU_A.ASM`) and at vector 15, the vector table need to point to `OS_CPU_SysTickHandler()` (see `BSP.C`).

`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 find 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 5-1) and `AppStartTask()` (listing 5-2).

Listing 6-1, `main()`

```
void main (void)
{
#if (OS_TASK_NAME_EN > 0)
    CPU_INT08U  err;
#endif

#if (CPU_CFG_NAME_EN == DEF_ENABLED)
    CPU_ERR     cpu_err;
#endif

    CPU_Init();                                                         (1)

    Mem_Init();                                                         (2)
    Math_Init();                                                        (3)

#if (CPU_CFG_NAME_EN == DEF_ENABLED)
    CPU_NameSet((CPU_CHAR *)"TMPM364FD",
                (CPU_ERR  *)&cpu_err);
#endif

    BSP_IntDisAll();                                                    (4)

    OSInit();                                                           (5)

    OSTaskCreateExt(AppStartTask,                                       (6)
                    (void *)0,
                    (OS_STK *)&AppStartTaskStk[APP_TASK_START_STK_SIZE-1],
                    APP_TASK_START_PRIO,
                    APP_TASK_START_PRIO,
                    (OS_STK *)&AppStartTaskStk[0],
                    APP_TASK_START_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

#if (OS_TASK_NAME_EN > 0)
    OSTaskNameSet(APP_CFG_TASK_START_PRIO, "Start", &err);              (7)
#endif

    OSStart();                                                          (8)
```

```
    return (1);
}
```

L5-1(1) Initialize the CPU module which initializes CPU timestamps, CPU interrupt disable time measurements and the CPU host name.

L5-1(2) Initialize the Memory Management Module.

L5-1(3) Initialize the Mathematic Module.

L5-1(4) We need to disable interrupts to ensure we do not get interrupted until we complete the initialization sequence.

L5-1(5) As with all **µC/OS-II** based applications, you need to initialize **µC/OS-II** by calling `OSInit()`.

L5-1(6) 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.

L5-1(7) We can now give names to tasks and those can be displayed by Kernel Aware debuggers such as IAR's C-Spy.

L5-1(8) In order to start multitasking, you need to call `OSStart()`. Note that `OSStart()` will not return from this call.

Listing 5-2, `AppStartTask()`

```
static  void  AppStartTask (void *p_arg)
{
    CPU_INT32U  cpu_clk_freq;
    CPU_INT32U  cnts;
   (void)p_arg;

    BSP_Init();                                                 (1)

    cpu_clk_freq = BSP_CPU_ClkFreq();
    cnts         = cpu_clk_freq / (CPU_INT32U)OS_TICKS_PER_SEC;
    OS_CPU_SysTickInit(cnts);                                   (2)

#if OS_TASK_STAT_EN > 0
    OSStatInit();                                               (3)
#endif

    APP_TRACE_INFO(("Creating Application Events...\n\r"));     (4)
    App_EventCreate();

    APP_TRACE_INFO(("Creating Application Tasks...\n\r"));      (5)
    App_TaskCreate();

    while (TRUE) {
        APP_TRACE_INFO(("Hello World from Start Task\n\r"));    (6)
        OSTimeDlyHMSM(0, 0, 1, 0);                              (7)
    }
}
```

L5-2(1) If you decided to implement a BSP (see section 6, Board Support Package) for your target board, you would initialize it here.

L5-2(1) You should now initialize the SysTick, which will provided the **µC/OS-II** time tick.

L5-2(3) 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.

L5-2(4) At this point, you can create additional events. We decided to place all our task initialization in one function called `App_EventCreate()` but, you are certainly welcome to use a different technique.

L5-2(5) At this point, you can create additional tasks. We decided to place all our task initialization in one function called `App_TaskCreate()` but, you are certainly welcome to use a different technique.

L5-2(6) You can now perform whatever additional function you want for this task. The example outputs "Hello World from Start Task"

L5-2(7) We decided to toggle the message every second.

5.02 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.

6.00 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 Micriμm BSP consist of at least 2 files: `BSP.C` and `BSP.H`.

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

6.01 BSP (Board Support Package) – LED Management

A number of evaluation boards are equipped with LEDs, we decided to create LED control functions as follows:

```
void  BSP_LED_On(CPU_INT08U  led_id);
void  BSP_LED_Off(CPU_INT08U led_id);
void  BSP_LED_Toggle(CPU_INT08U 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 `BSP_LED_On(1)`. If you want to toggle LED #2, you simply call `BSP_LED_Toggle(2)`. In fact, you can (and should) associate names to your LEDs using `#define`s. You could thus specify `BSP_LED_Off(LED_PM)`.

7.00 Conclusion

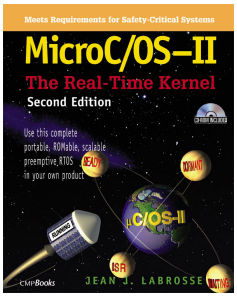
This application note presented a 'generic' port ARM Cortex-M4 processors. The port should be easily adapted to different compilers (the code itself should be identical). Of course, if you use µC/OS-II and use the port on actual hardware, you will need to initialize and properly handle hardware interrupts.

Licensing

If you intend to use **µC/OS-II** in a commercial product, remember that you need to contact **Micriμm** to properly license its use in your product. The use of **µC/OS-II** in commercial applications is **NOT-FREE**. Your honesty is greatly appreciated.

References

MicroC/OS-II, The Real-Time Kernel, 2nd Edition
Jean J. Labrosse
CMP 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

Freescale
6501 William Cannon Drive West
Austin, Texas 78735
USA
+1 800 521 6274
WEB: http://www.freescale.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

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

Notes