Micriµm

© Copyright 2006, Micriµm
All Rights reserved

µC/OS-II
and the
Microchip PIC24

Application Note
AN-1024

www.Micrium.com

Table of Contents

1.00 Introduction

This application note describes how **µC/OS-II** has been ported to the Microchip PIC24 family of processors. This application note does not assume any specific PIC24 derivative.

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.

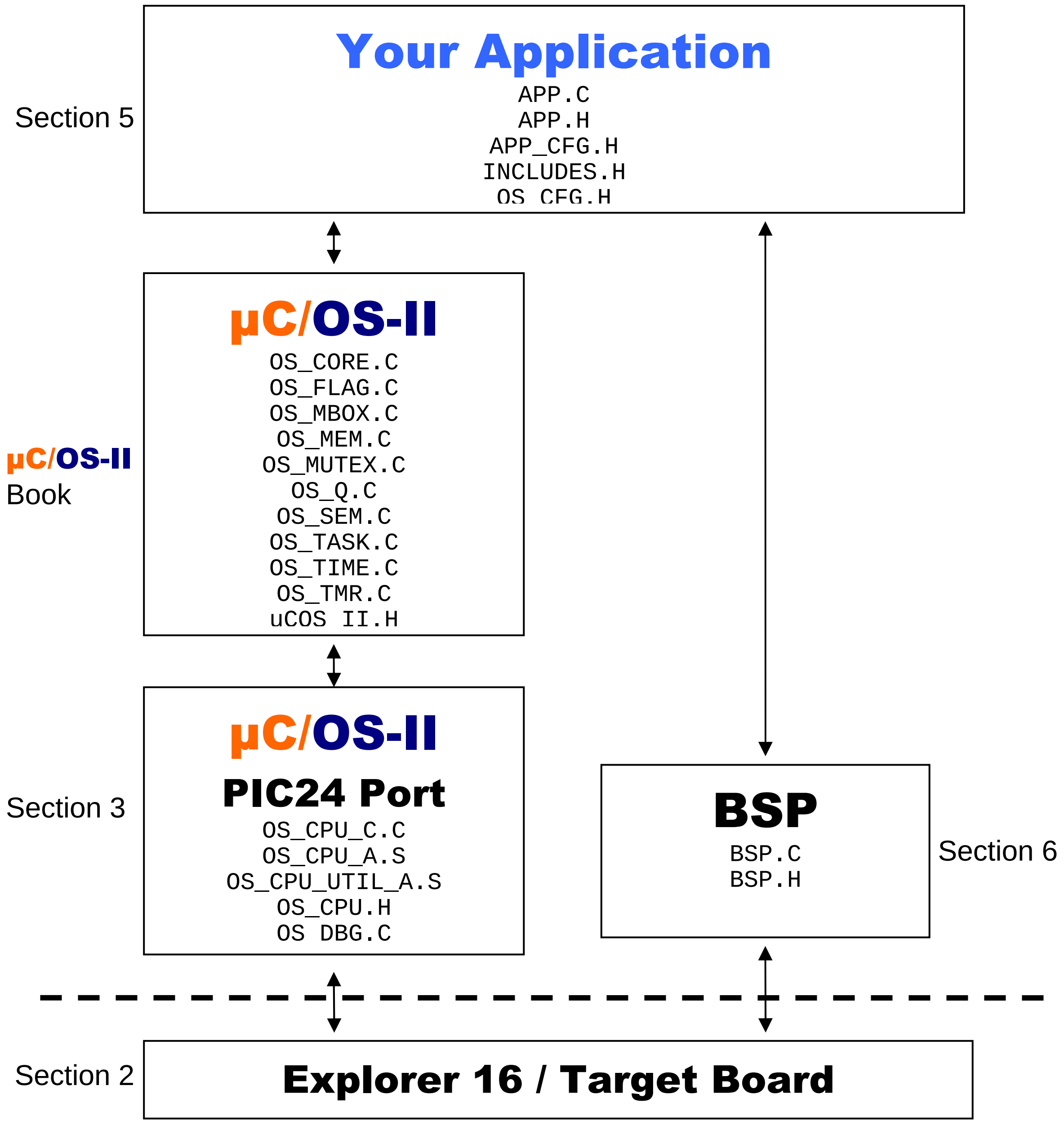


Figure 1-1, Relationship between modules.

2.00 The Microchip PIC24 programmer's model

This section provides a brief description of the PIC24 programmer's model. We present enough information in this section to provide a brief introduction. A complete description can be found in the Microchip documentation.

The PIC24 family of CPU cores features proprietary Microchip architecture and is designed for controller applications using a 16-bit (data) architecture. Below are some of the features of the PIC24 family of CPU cores:

- 16 General-purpose 'Working' registers
- 24 bit variable length instruction word, 23 bit program counter.
- 17 bit x 17 bit hardware multiplier
- Enhanced iterative integer divide support
- High speed vectored deterministic interrupt processing

The PIC24 is a ***Little Endian*** architecture and thus the least significant byte of a value is placed at a lower memory location than its most significant byte.

Figure 2-1 shows the register model of the PIC24 and consist of 16 Working registers, a Stack Pointer Limit register, the Program Counter, the Data Table Page register, the Program Space Visibility Page Address register, a Repeat Loop Counter, the Core Configuration register, and a CPU Status register.

W0-W15 (Working Registers)

- Registers W0-W1 are Divider Working registers
- Registers W2-W3 are Multiplier registers
- Registers W4-W13 are general purpose scratch pad registers
- Register W14 doubles as the Frame Pointer register
- Register W15 doubles as the CPU Stack Pointer register

The stack pointer always points to the last ***pushed*** element. In other words, when an element is pushed onto the stack, the stack pointer is first decremented and the value placed at the current location where the stack is pointing to. The stack is ***popped*** by first reading the contents of the stack and then incrementing the stack pointer.

SPLIM (Stack Pointer Limit Register)

- Causes a TRAP error when W15 is greater than SPLIM

PC (Program Counter)

- Non Writable, 23 bit register that points to the next instruction to be executed.

TBLPAGE (Table Page Register)

- The 8-bit Table Page register (TBLPAG) is used to define a 32K word region within the program space. This is concatenated with a 16-bit effective address in order to arrive at a full 24-bit program space address.

PSVPAG (Program Space Visibility Page Address Register)

- For remapping operations, the 8-bit Program Space Visibility register (PSVPAG) is used to define a 16K word page in the program space. When the Most Significant bit of the EA is ‘1’, PSVPAG is concatenated with the lower 15 bits of the EA to form a 23-bit program space address. Unlike table operations, this limits remapping operations strictly to the user memory area.

RCOUNT (Repeat Loop Counter)

- The RCOUNT register tracks the iteration count of an instruction that must be executed repetitively until its operation has been completed. Divide instructions are iterative instructions that make extensive use of the RCOUNT register.

CORECON (Core Configuration Register)

- Contains configuration bits pertaining to how the processor handles certain types of operations.

STATUS (CPU Status Register)

- Contains status CPU status flags indicating the state of the CPU after a given instruction.

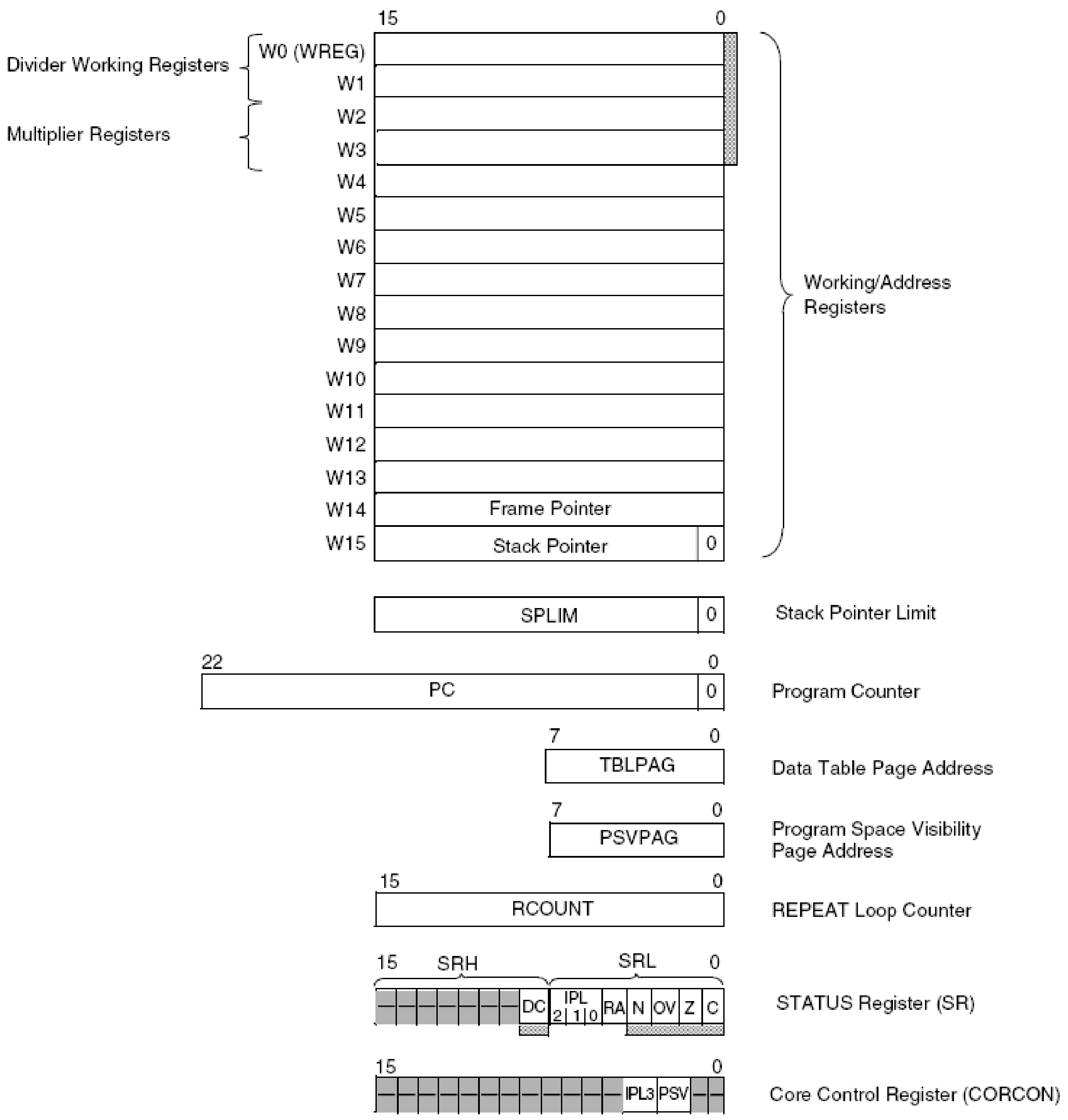


Figure 2-1, PIC24 Programmers Model.

3.00 µC/OS-II Port for PIC24

We used the Microchip MPLab tool chain to test the port. MPLab contains an editor, a C/C++ compiler, an assembler, a linker/locator and a debugger. The port was tested on an Explorer 16 EVB with a PIC24FJ128GA010 derivative. It is assumed that you have µC/OS-II V2.82 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\Ports\Microchip\ PIC24FJ128
```

The source code for the µC/OS-II PIC24 port is found in the following files:

`OS_CPU.H`	Section 3.02
`OS_CPU_C.C`	Section 3.03
`OS_CPU_A.S`	Section 3.04
`OS_CPU_UTIL_A.S`	Section 3.05
`OS_DBG.C`	Section 3.06

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. However, this port does not contain any global variables but the declarations have been included in case we need to add some in the future.

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 int     INT16U;          // (1)
typedef signed   int     INT16S;
typedef unsigned long    INT32U;
typedef signed   long    INT32S;
typedef float            FP32;            // (2)

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

L3-2(1) If you were to consult the MPLab compiler documentation, you would find that an `int` is 16 bits, an **`long`** is 32 bits.

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 PIC24 processor is always 16 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 (STATUS) on the PIC24 processor is 16 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 preferred 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. We recommend that you name the functions expected in `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()` as `OS_CPU_SR_Save()` and `OS_CPU_SR_Restore()`, respectively. These routines are generally stored in `OS_CPU_A.ASM`, however, in this case, since the registers are memory mapped, it makes sense to simply write and read to and from SR directly from C code.

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

```
#define  OS_CRITICAL_METHOD    3


#if      OS_CRITICAL_METHOD == 3
#define  OS_ENTER_CRITICAL()  {cpu_sr = SR; SRbits.IPL = 6;}
#define  OS_EXIT_CRITICAL()   {SR     = cpu_sr;}
#endif
```

3.02.04 OS_CPU.H, Stack growth

The stacks on the PIC24 grows from low to high memory and thus, `OS_STK_GROWTH` is set to `0` to indicate this to **µC/OS-II**.

Listing 3-4, OS_CPU.H, Stack Growth

```
#define  OS_STK_GROWTH        0
```

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. You should note that **µC/OS-II** disables all interrupts before calling `OSCtxSw()`.

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

```
#define  OS_TASK_SW()      {__asm__  volatile("call _OSCtxSw");}
```

3.02.06 OS_CPU.H, Function Prototypes

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 3-8, OS_CPU.H, Function Prototypes

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

3.03 OS_CPU_C.C

A **µC/OS-II** port requires that you write ten (10) 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.

IMPORTANT

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

```
void OSInitHookBegin (void)
{
#if OS_TMR_EN > 0
    OSTmrCtr = 0;
#endif
}
```

3.03.02 OS_CPU_C.C, OSInitHookEnd()

This function is called by **µC/OS-II**'s `OSInit()` at the very end of `OSInit()`. It gives the opportunity to add additional initialization code specific to the port. In this port, we don't perform any operations and thus the function is empty.

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

```
void OSInitHookEnd (void)
{
}
```

3.03.03 OS_CPU_C.C, OSTaskCreateHook()

This function is called by **µC/OS-II**'s OSTaskCreate() or OSTaskCreateExt() when a task is created. OSTaskCreateHook() gives the opportunity to add code specific to the port when a task is created. In 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 the #define OS_VIEW_MODULE 1 declaration 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 3-11, OS_CPU_C.C, OSInitHookEnd()

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

3.03.04 OS_CPU_C.C, OSTaskStkInit()

Recall (from the µC/OS-II documentation) that a task is declared as shown in listing 3-12.

Listing 3-12, µC/OS-II Task

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

The code in Listing 3-13 initializes the stack frame for the task being created. The task received an optional argument ‘`p_arg`’. The MPLab compiler passes a single argument in register `W0` and thus, ‘`p_arg`’ is passed in `W0` when the task is created.

The ‘Task Body’ MUST call either one of the `OS???Pend()` functions or `OSTimedly??()` functions. In other words, a task MUST always be waiting for an event to occur. An event can be the reception of a signal or a message from another task or ISR. An event can also be to wait for the passage of time.

Listing 3-13, OS_CPU_C.C, `OSTaskStkInit()`

```
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *p_arg, OS_STK *ptos, INT16U opt)
{
    INT16U x;
    INT8U  pc_high;


    pc_high =  (INT8U)(((INT32U)task >> 16) & 0x7F);

   *ptos++  =  (OS_STK)task;                 // Push the PC (Starting Address for the new task)
   *ptos++  =  (OS_STK)pc_high;              // Upper byte word of start address

   *ptos++  =  (OS_STK)task;                 // Simulate an Interrupt, Push the PC

    x       =  0;                            // Set SR to enable all interrupts
    if (CORCONbits.IPL3) {                   // Check the interrupt level mask bit IPL3, == 1?
        x   |= 0x0080;                       // Save the value of IPL3 in x, bit 7.
    }

   *ptos++  = (OS_STK)(x | (INT16U)pc_high); // Push the SR Low, CORCON IPL3 and PC (22..16)

   *ptos++  = x;                             // Push SR Low and CORCON IPL3 on to the stack

   *ptos++  = (OS_STK)p_arg;                 // W0, Push a dummy argument on to the stack
   *ptos++  = 0x1111;                        // W1
   *ptos++  = 0x2222;                        // W2
   *ptos++  = 0x3333;                        // W3
   *ptos++  = 0x4444;                        // W4
   *ptos++  = 0x5555;                        // W5
   *ptos++  = 0x6666;                        // W6
   *ptos++  = 0x7777;                        // W7
   *ptos++  = 0x8888;                        // W8
   *ptos++  = 0x9999;                        // W9
   *ptos++  = 0xAAAA;                        // W10
   *ptos++  = 0xBBBB;                        // W11
   *ptos++  = 0xCCCC;                        // W12
   *ptos++  = 0xDDDD;                        // W13
   *ptos++  = 0xEEEE;                        // W14

   *ptos++  = TBLPAG;                        // Table Page
   *ptos++  = PSVPAG;                        // Program Space Visibility Page Address
   *ptos++  = RCOUNT;                        // Repeat Loop Counter

   *ptos++  = 0;                             // Status Register (contents irrelevant)
   *ptos++  = CORCON;                        // Core Configuration Register

    return (ptos);                           // Return the address to the top of the stack
}
```

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

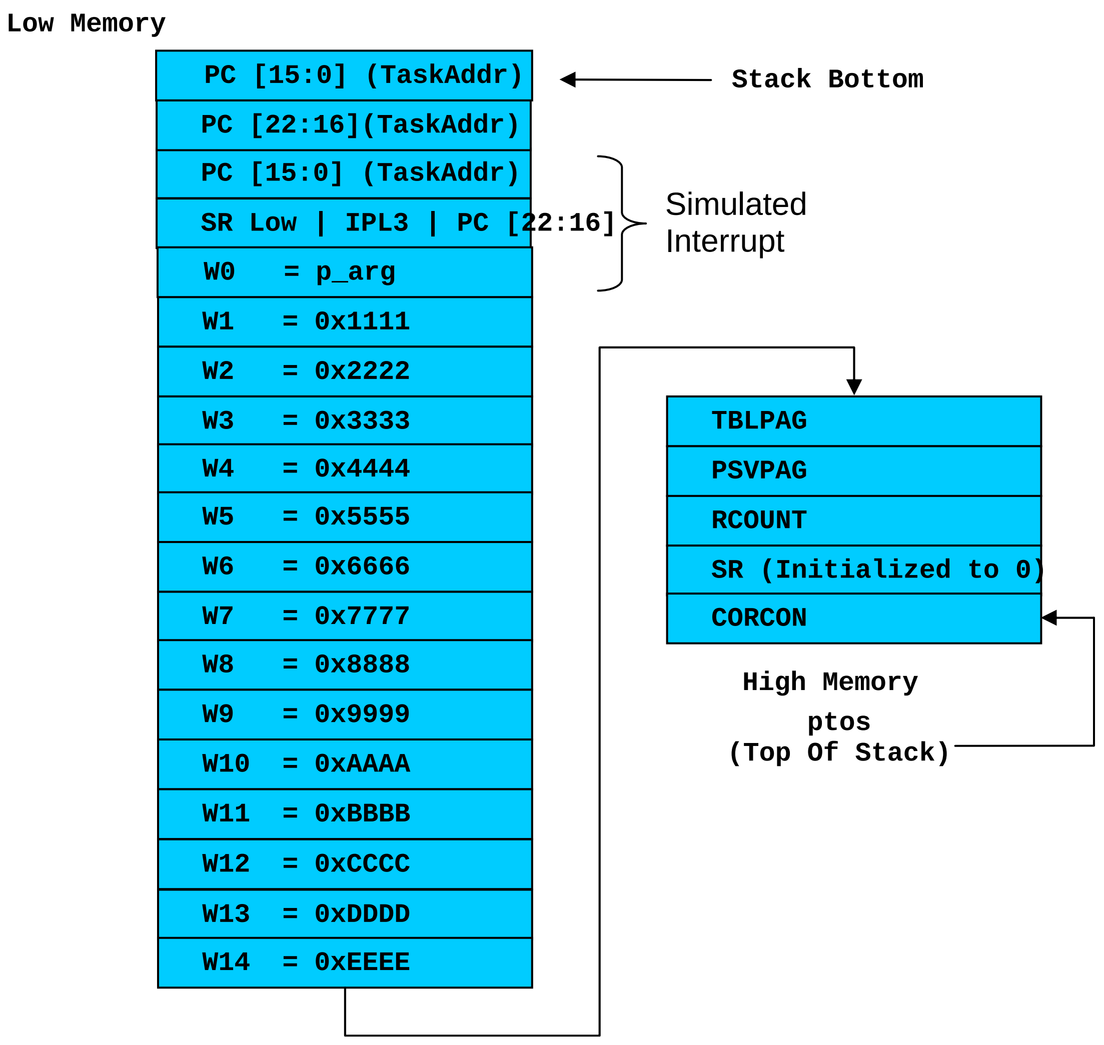


Figure 3-1, The Initial Stack Frame for each Task.

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

3.03.05 OS_CPU_C.C, OSTaskSwHook()

OSTaskSwHook() is called when a context switch occurs. This function allows the port code to be extended and do things such as measuring the execution time of a task, output a pulse on a port pin when a context switch occurs, etc. 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. If you did not purchase **µC/OS-View** from Micrium then you can simply set OS_VIEW_MODULE to 0 in OS_CFG.H and remove references to the **µC/OS-View** source files in MPLab.

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

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

3.03.06 OS_CPU_C.C, OSTimeTickHook()

`OSTimeTickHook()` is called at the very beginning of `OSTimeTick()`. This function allows the port code to be extended 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`. If you did not purchase **µC/OS-View** from Micrium then you can simply set `OS_VIEW_MODULE` to `0` in `OS_CFG.H` and remove references to the **µC/OS-View** source files in MPLab.

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

Listing 3-15, OS_CPU_C.C, OSTimeTickHook()

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

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

3.04 OS_CPU_A.S

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()
```

It should be noted that since the PIC24 has memory mapped registers, the first two functions, namely `OS_CPU_SR_Save()` and `OS_CPU_SR_Restore()` have been implemented as C macros. Since the other, bold font, functions involve modifying the program counter, they have been implemented in assembly language in order to leverage off of the `RETFIE` instruction. The `RETFIE` instruction will restore the PC from data located on the current tasks stack. This is why an interrupt is simulated during task stack initialization.

3.04.01 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 3-18, `OSStartHighRdy()`

```
_OSStartHighRdy:
    call   _OSTaskSwHook                    ; (1) Call user defined task switch hook

    mov    #0x0001, w0                      ; (2) Set OSRunning to TRUE
    mov.b  wreg, _OSRunning

    mov    _OSTCBHighRdy, w0                ; (3) Resume Stack Pointer
    mov    [w0], w15

    OS_REGS_RESTORE                         ; (4) Restore Context

    retfie                                  ; (5) Run Task
```

L3-18(1) Before starting the highest priority task, we call `OSTaskSwHook()` in case a hook function has been declared (see `OS_CPU_C.C`).

L3-18(2) The **µC/OS-II** flag `OSRunning` is set to `TRUE` indicating that **µC/OS-II** will be running once the first task is started.

L3-18(3) We then get the pointer to the task's top-of-stack (was stored by `OSTaskCreate()` or `OSTaskCreateExt()`). See figure 3-1 (`ptos` is stored in the `OS_TCB` of the created task).

L3-18(4) We then pop all the registers from the task's stack. This is done by calling an assembly language macro which is declared in `OS_CPU_UTIL_A.S` and will be described later.

L3-18(5) By executing a return from interrupt instruction, the PIC24 pops the PC and the STATUS register from the stack and thus, the PIC24 will start executing the task's code.

3.04.02 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.).

A task level context switch occurs when **µC/OS-II** invokes the macro `OS_TASK_SW()` which, in the case of the PIC24 port, corresponds to calling `OSCtxSw()`. Normally, it is ideal to use a software interrupt or TRAP in order to enter `OSCtxSw()`, however, since a software interrupt is not available, code must adjust the stack frame in order to simulate an interrupt such that the `RETFIE` instruction may be used to restore the PC, SR Low, and CORCON IPL3.

```
OS_TASK_SW()

OSCtxSw:
   Add SRL, IPL3 and the upper 7 bits of the return PC to simulate an ISR   /* (1) */

   Save the CPU registers onto the old task's stack;                        /* (2) */

   OSTCBCur->OSTCBStkPtr = SP;                                              /* (3) */
   OSTaskSwHook();                                                          /* (4) */
   OSPrioCur              = OSPrioHighRdy;                                  /* (5) */
   OSTCBCur               = OSTCBHighRdy;                                   /* (6) */
   SP                     = OSTCBHighRdy->OSTCBStkPtr;                      /* (7) */

   Restore the CPU registers from the new task's stack;                     /* (8) */

   Return from Interrupt;                                                   /* (9) */
```

The actual code for the task level context switch is shown in Listing 3-19.

Listing 3-19, `OSCtxSw()`

```
_OSCtxSw:
    mov.b   SRL, wreg             ; (1) Simulate an interrupt, save SRL and CORCON IPL3
    sl      w0, #8, w0
    btsc    CORCON, #IPL3
    bset    w0, #7;
    ior     w0, [--w15], w0
    mov     w0, [w15++]

    OS_REGS_SAVE                  ; (2) Save context of interrupted task

    mov     _OSTCBCur, w0         ; (3) OSTCBCur->OSTCBStkPtr = SP
    mov     w15, [w0]

    call    _OSTaskSwHook         ; (4) Call user defined task switch hook

    mov     _OSTCBHighRdy, w1     ; (5) OSPrioCur = OSPrioHighRdy
    mov     w1, _OSTCBCur

    mov.b   _OSPrioHighRdy, wreg  ; (6) OSTCBCur = OSTCBHighRdy
    mov.b   wreg, _OSPrioCur

    mov     [w1], w15             ; (7) SP = OSTCBHighRdy->OSTCBStkPtr

    OS_REGS_RESTORE               ; (8) Restore context of interrupted task

    retfie                        ; (9) Return from interrupt
```

3.04.03 OS_CPU_A.S, OSIntCtxSw()

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

Listing 3-20, `OSIntCtxSw()`

```
_OSIntCtxSw:
    call   _OSTaskSwHook              ; Call user defined task switch hook

    mov    _OSTCBHighRdy, w1          ; OSPrioCur = OSPrioHighRdy
    mov    w1, _OSTCBCur
    mov.b  _OSPrioHighRdy, wreg       ; OSTCBCur  = OSTCBHighRdy
    mov.b  wreg, _OSPrioCur

    mov    [w1], w15                  ; W15 (SP)  = OSTCBHighRdy->OSTCBStkPtr

    OS_REGS_RESTORE                   ; Restore context of interrupted task

    retfie                            ; Switch to the new context
```

3.04.04 OS_CPU_A.S, T2Interrupt()

Since the PIC24 uses a vectored interrupt controller, **ALL ISR's must follow the following format**. The example provided below is the actual OS Time Tick ISR for µC/OS-II. Generally, it is common practice to initialize the OS Tick timer from C code (`BSP.C`) and service the interrupt in assembly (`BSP_A.S`). From assembly, a C ISR Handler function may be called in order to handle more complex operations.

Note: `T2Interrupt()` is one of two possible OS Time Tick Interrupt vectors. The option to choose 1 of 2 possible timers to generate the OS Tick exists from within (`BSP.H`). If an alternate timer is selected the ISR code below will not run. See T4Interrupt below.

```
__T2Interrupt:
    OS_REGS_SAVE                        ; 1) Save processor registers

    inc.b   _OSIntNesting               ; 2) Call OSIntEnter() or increment OSIntNesting

    dec.b   _OSIntNesting, wreg         ; 3) Check OSIntNesting. if OSIntNesting == 1,
    bra     nz, T2_Cont                 ;    then save the stack pointer, otherwise jump
    mov     _OSTCBCur, w0               ;    to T2_Cont.
    mov     w15, [w0]

T2_Cont:
    call    _OS_Tick_ISR_Handler        ; 4) Call YOUR ISR Handler (May be a C function).
                                        ;    In this case, the OS Tick ISR Handler

    call    _OSIntExit                  ; 5) Call OSIntExit()

    OS_REGS_RESTORE                     ; 6) Restore registers

    retfie
```

3.04.05 OS_CPU_A.S, T4Interrupt()

Since the PIC24 uses a vectored interrupt controller, **ALL ISR's must follow the following format**. The example provided below is the actual OS Time Tick ISR for **µC/OS-II**. Generally, it is common practice to initialize the OS Tick timer from C code (`BSP.C`) and service the interrupt in assembly (`BSP_A.S`). From assembly, a C ISR Handler function may be called in order to handle more complex operations.

Note: `T4Interrupt()` is one of two possible OS Time Tick Interrupt vectors. The option to choose 1 of 2 possible timers to generate the OS Tick exists from within (`BSP.H`). If an alternate timer is selected the ISR code below will not run. See T2Interrupt above.

```
__T4Interrupt:
    OS_REGS_SAVE                        ; 1) Save processor registers

    inc.b   _OSIntNesting               ; 2) Call OSIntEnter() or increment OSIntNesting

    dec.b   _OSIntNesting, wreg         ; 3) Check OSIntNesting. if OSIntNesting == 1,
    bra     nz, T4_Cont                 ;    then save the stack pointer, otherwise jump
    mov     _OSTCBCur, w0               ;    to T2_Cont.
    mov     w15, [w0]

T4_Cont:
    call    _OS_Tick_ISR_Handler        ; 4) Call YOUR ISR Handler (May be a C function).
                                        ;    In this case, the OS Tick ISR Handler

    call    _OSIntExit                  ; 5) Call OSIntExit()

    OS_REGS_RESTORE                     ; 6) Restore registers

    retfie
```

3.05 OS_CPU_UTIL_A.S

OS_CPU_UTIL_A.S is an assembly language file that declares two macros used to save the context of the CPU. These macros are called OS_REGS_SAVE and OS_REGS_RESTORE and are declared as shown below.

```
.macro OS_REGS_SAVE
    push.d  w0
    push.d  w2
    push.d  w4
    push.d  w6
    push.d  w8
    push.d  w10
    push.d  w12
    push    w14

    push    TBLPAG
    push    PSVPAG
    push    RCOUNT

    push    SR
    push    CORCON
.endm


.macro OS_REGS_RESTORE
    pop     CORCON
    pop     SR

    pop     RCOUNT
    pop     PSVPAG
    pop     TBLPAG

    pop     w14
    pop.d   w12
    pop.d   w10
    pop.d   w8
    pop.d   w6
    pop.d   w4
    pop.d   w2
    pop.d   w0
.endm
```

IMPORTANT

You **MUST** include a reference to this file when you write your ISRs (see section 4.00, Interrupt Handling). This is done by using the assembler directive #include as follows:

```
.include    "os_cpu_util_a.s"
```

3.06 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. Unfortunately, the MPLab debugger is not **µC/OS-II** aware and thus this file is not needed but should be included in all builds for future reference.

4.00 Interrupt Handling

The PIC24 contains an interrupt and exception vector table which contains up to 118 entries. Each of these entries point to an interrupt or exception handler. For **µC/OS-II**, each of those interrupt handlers **MUST** be written in assembly language. In fact, only a portion must be in assembly language as shown in listing 4-1. Note that you **ONLY** need to change the portion in **RED** for your own ISR. The rest of the code is **IDENTICAL** from one ISR to the next. Of course, you will need to give a unique name to your ISRs.

Listing 4-1, Assembly Language ISR

```
_My_ISR:
    OS_REGS_SAVE                        ; 1) Save processor registers

    inc.b    _OSIntNesting              ; 2) Call OSIntEnter() or increment OSIntNesting

    dec.b    _OSIntNesting, wreg        ; 3) Check OSIntNesting. if OSIntNesting == 1,
    bra      nz, _My_ISR_1              ;    then save the stack pointer, otherwise jump
    mov      _OSTCBCur, w0              ;    to _My_ISR_1
    mov      w15, [w0]

_My_ISR_1:
     call    _My_ISR_Handler            ; 4) Call YOUR ISR Handler (May be a C function).
                                        ;    You MUST clear the interrupt source either
                                        ;    from within, or just after your ISR Handler.

    call     _OSIntExit                 ; 5) Call OSIntExit()

    OS_REGS_RESTORE                     ; 6) Restore registers

    retfie                              ; 7) Return to the interrupted task
```

L4-1(1) You then **MUST** save all of the registers using the `OS_REGS_SAVE` macro.

L4-1(2) You **MUST** increment **µC/OS-II**'s interrupt nesting counter (`OSIntNesting`)

L4-1(3) You **MUST** check to see whether this is the first nested ISR by checking if `OSIntNesting` got incremented to 1. If this is the first nested ISR level then you **MUST** save the stack pointer into the current task's `OS_TCB`.

L4-1(4) You can now call your actual ISR handler which could be written in C. You don't have to write the handler in C but it's generally more readable and portable. You may optionally choose to clear the interrupt source from within your ISR handler, or you may do it before calling `OSIntExit()` from assembly.

L4-1(5) When you are done handling the ISR (i.e. the code returns from `My_ISR_Handler()`), you **MUST** call `OSIntExit()`. `OSIntExit()` checks to see if this is the last nested ISR. If it is then `OSIntExit()` checks to see if a more important task has been made ready-to-run by the ISR (or any other nested ISRs). If a more important task is ready-to-run, `OSIntExit()` doesn't return but instead context switches to the more important task.

L4-1(6) If the interrupted task is still the most important task to run then `OSIntExit()` returns and we simply need to restore the saved CPU registers in order to return to the interrupted task. To restore the registers, you **MUST** invoke the `OS_REGS_RESTORE` macro.

L4-1(7) The `RETFIE` instruction **MUST** be executed to return program execution back to the interrupted task.

The pseudo-code for the C ISR handler is shown in Listing 4-2.

Listing 4-2, C ISR Handler

```
void  My_ISR_Handler (void)
{
    /* (1) Don’t forget to clear the interrupt source               */
    /* (2) Enable interrupts if you want to allow nested interrupts */
    /* (3) Handle the interrupt using C                             */
    /* (4) Disable interrupts (if you enabled them)                 */
}
```

L4-2(1)	Don’t forget to clear the interrupting device (i.e. acknowledge that you serviced the interrupt). Failure to do this will cause the ISR to be re-entered which may not be what you want.
L4-2(2)	As indicated, you may enable interrupts (by lowering the IPL level in the SR) if you want to allow nested interrupts.
L4-2(3)	You can now service the interrupting device using the C programming language (instead of doing that in assembly language).
L4-2(4)	If you disabled interrupts (see step #1) then you should disable them before returning to the caller of this function.

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 dsPIC CPU and the target hardware.

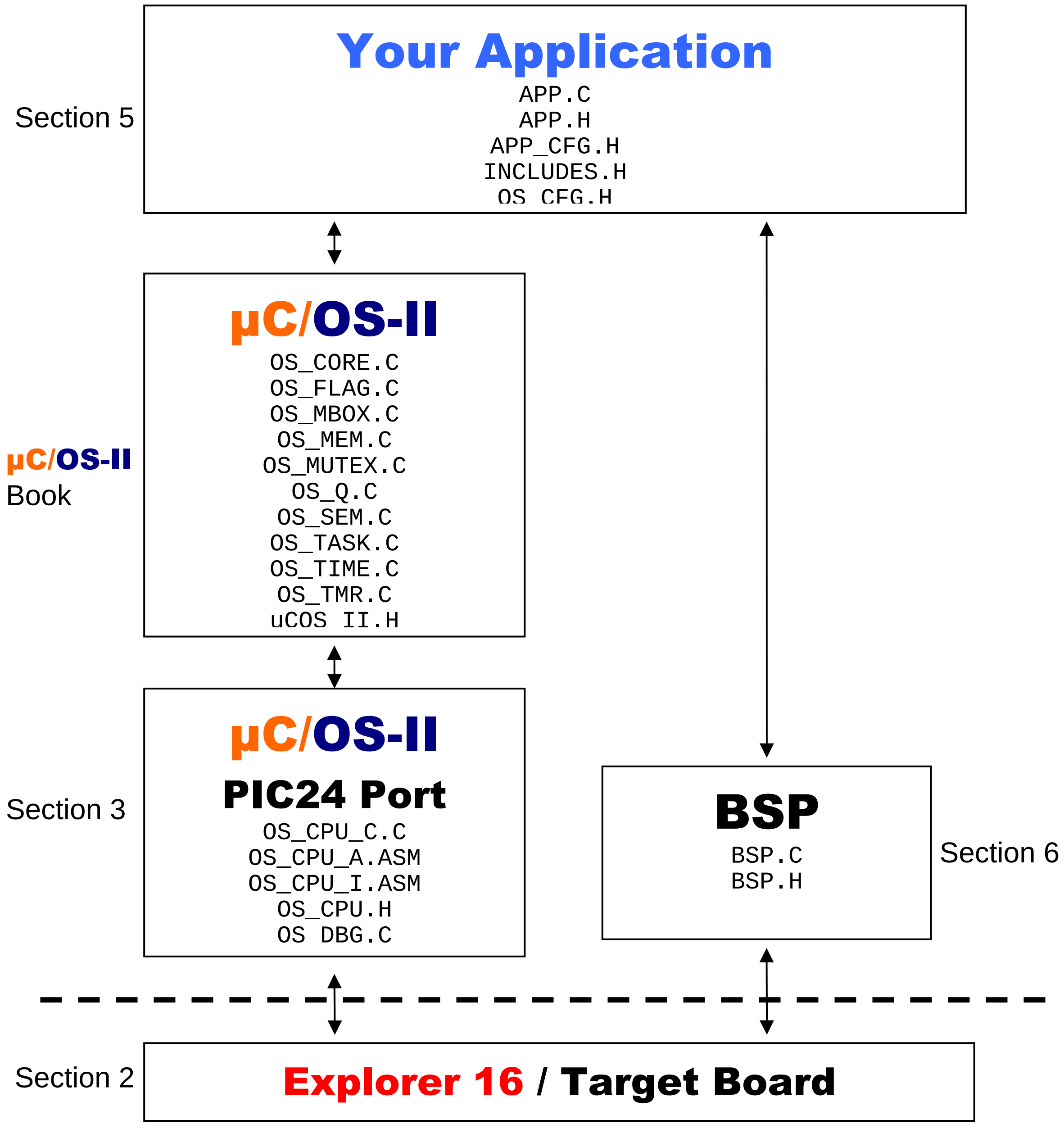


Figure 5-1, Relationship between modules.

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.H` contains `#define` constants, macros, prototypes, etc. that are specific to your application. This file may NOT exist in YOUR application and it is not included in examples provided for the PIC24.

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

`APP.C` is a standard test file for **µC/OS-II** examples. The two important functions are `main()` (listing 5-1) and `AppStartTask()` (listing 5-2).

Listing 5-1, `main()`

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


    BSP_IntDisAll();                                                     (1)

    OSInit();                                                            (2)

    OSTaskCreateExt(AppStartTask,                                        (3)
                    (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);                  (4)
#endif

    OSStart();                                                           (5)
}
```

L5-1(1) A BSP function called `BSP_IntDisAll()` is called to disable ALL interrupts. You would typically prevent the interrupt controller from issuing interrupts until your application is ready to service them.

L5-1(2) As with all **µC/OS-II** based applications, you need to initialize **µC/OS-II** by calling `OSInit()`.

L5-1(3) 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.

L5-1(4) We can now give a name for our task.

L5-1(5) 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)
{
    (void)p_arg;

    BSP_Init();                                     (1)

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

#if OS_VIEW_MODULE > 0
    OSView_Init(38400);                 /* Initialize uC/OS-View if module is present */
    OSView_TerminalRxSetCallback(AppTerminalRx);
    OSView_RxIntEn();
#endif

    AppTaskCreate();                                (3)

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

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(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.

L5-2(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.

L5-2(4) You can now perform whatever additional function you want for this task. We decided to toggle an LED by calling a BSP function called `LED_Toggle()`.

L5-2(5) Each of your tasks **MUST** invoke one of the **µC/OS-II** functions that will wait for an event to occur. We decided to use `OSTimeDly()` which suspends the task for a specified amount of time.

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. Listing 5-3 shows the typical contents of `INCLUDES.H`. Of course, you can add your own header files as needed.

Listing 5-3, INCLUDES.H

```
#ifndef     INCLUDES_H
#define     INCLUDES_H

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

#include    < p24FJ128GA010.h>

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

#include    <lib_def.h>
#include    <lib_str.h>
#include    <lib_mem.h>

#endif
```

5.03 OS_CFG.H

Every **µC/OS-II** requires that you configure the RTOS for your own application. The configuration of **µC/OS-II** allows to specify how many tasks your application will have, how many semaphores (if any), how many message queues (if any), etc. Configuring **µC/OS-II** allows **µC/OS-II**'s footprint to be only as big as it needs to be.

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 (OS Ticker and View)
PLL Initialization
ISR Handlers
LED control functions
Reading switches
Setting up communication channels
Etc.

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

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

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

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)`. Where `LED_PM` 'could' be defined as LED #2.

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 ISR handler and its initialization function in the BSP because they ideally belong in your application code and not as part of **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 that your application requires. The clock tick ISR is in `BSP_A.S` which in turn calls the `OS_Tick_ISR_Handler()` that is found in `BSP.C`.

7.00 Conclusion

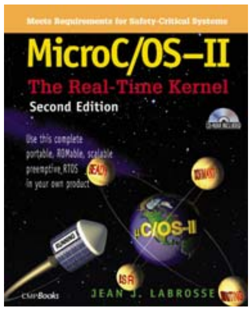
This application note presented a 'generic' port for the PIC24 processor. Of course, if you use **µC/OS-II** and use the port on actual hardware, you will need to initialize and properly handle hardware interrupts.

Licensing

If you intend to use **µC/OS-II** in a commercial product, remember that you need to contact **Micrium** 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 Technical Books, 2002
ISBN 1-5782-0103-9



Contacts

CMP Books, Inc.
6600 Silacci Way
Gilroy, CA 95020 USA
Phone Orders: 1-800-500-6875
or 1-408-848-3854
Fax Orders: 1-408-848-5784
e-mail: rushorders@cmpbooks.com
WEB: http://www.cmpbooks.com

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

Microchip Technology Inc.
2355 West Chandler Blvd.
Chandler, Arizona 85224-6199
USA
480-792-7200
WEB: www.MicroChip.com