

Micrium

© Copyright 2006, Micrium
All Rights reserved

μ C/OS-II

and

Fujitsu RISC (FR)

Application Note

AN-1015

www.Micrium.com

Table of Contents

1.00	Introduction	4
2.00	The Fujitsu RISC (FR) programmer's model.....	5
3.00	µC/OS-II Port for FR	10
3.01	Directories and Files	10
3.02	OS_CPU.H	11
3.02.01	OS_CPU.H, macros for 'externals'	11
3.02.02	OS_CPU.H, Data Types	11
3.02.03	OS_CPU.H, Critical Sections	13
3.02.04	OS_CPU.H, Stack growth.....	13
3.02.05	OS_CPU.H, Task Level Context Switch	13
3.02.06	OS_CPU.H, Initial ILM	14
3.02.06	OS_CPU.H, Function Prototypes.....	14
3.03	OS_CPU_C.C.....	15
3.03.01	OS_CPU_C.C, OSInitHookBegin()	15
3.03.02	OS_CPU_C.C, OSInitHookEnd()	16
3.03.03	OS_CPU_C.C, OSTaskCreateHook().....	16
3.03.04	OS_CPU_C.C, OSTaskStkInit().....	17
3.03.05	OS_CPU_C.C, OSTaskSwHook()	19
3.03.06	OS_CPU_C.C, OSTimeTickHook().....	19
3.04	OS_CPU_A.ASM	20
3.04.01	OS_CPU_A.ASM, OS_CPU_SR_Save().....	20
3.04.02	OS_CPU_A.ASM, OS_CPU_SR_Restore()	20
3.04.03	OS_CPU_A.ASM, OSStartHighRdy()	21
3.04.04	OS_CPU_A.ASM, OSCtxSw()	22
3.04.05	OS_CPU_A.ASM, OSIntCtxSw()	24
3.05	OS_CPU_I.ASM	25
3.06	OS_DBG.C	26
4.00	Interrupt Handling	27
5.00	Application Code.....	29
5.01	APP.C, APP.H and APP_CFG.H	30
5.02	INCLUDES.H	32
5.03	OS_CFG.H	32
6.00	BSP (Board Support Package)	33
7.00	Conclusion	34

Licensing.....	34
References	34
Contacts.....	34
Notes	35

1.00 Introduction

This application note describes how μC/OS-II has been ported to the Fujitsu RISC (FR) family of processors. This application note does not assume any specific implementation (i.e. a specific FR part).

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.

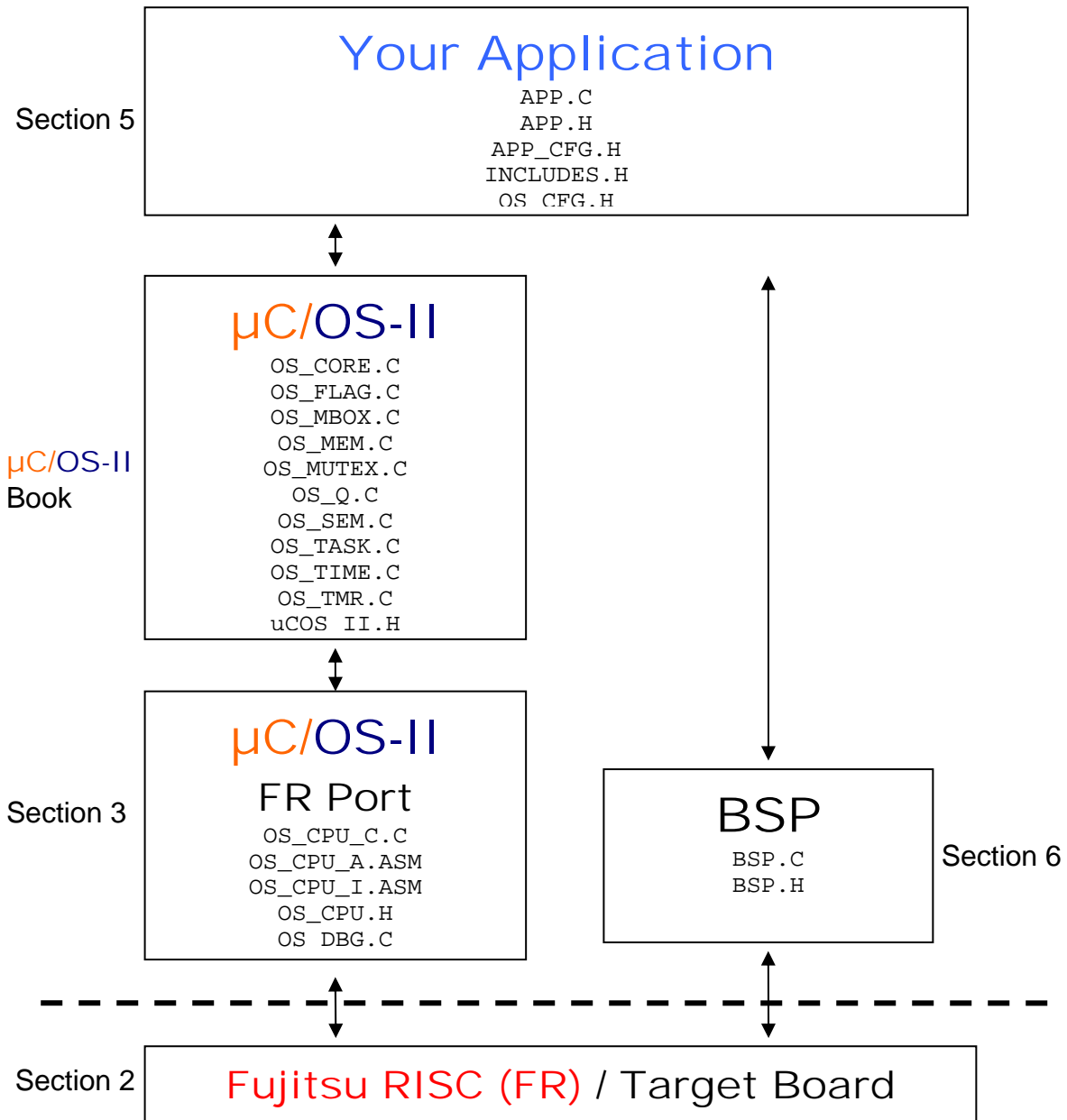


Figure 1-1, Relationship between modules.

2.00 The Fujitsu RISC (FR) programmer's model

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

The FR family of CPU cores (from now on FR) features proprietary Fujitsu architecture and is designed for controller applications using 32-bit "RISC" based computing. The architecture is optimized for use in microcontroller CPU cores for built-in control applications where high-speed control is required. Below are some of the features of the FR family of CPU cores:

- General-purpose register architecture
- Linear space for 32-bit (4 Gbyte) addressing
- 16-bit fixed instruction length (excluding immediate data, coprocessor instructions)
- 5-stage pipeline configuration for basic instructions, one-instruction one-cycle execution
- 32-bit by 32-bit computation enables completion of multiplication instructions within five cycles
- Stepwise division instructions enable 32-bit/ 32-bit division
- Direct addressing instructions for peripheral circuit access
- Coprocessor instructions for direct designation of peripheral accelerator
- High speed interrupt processing complete within 6 cycles

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

Figure 2-1 shows the register model of the FR and consist of 16 32-bit general purpose registers as well as 6 dedicated registers. Out of the general purpose registers, R13, R14 and R15 have special use as described below:

R13 (Accumulator: AC)

- Base address register for load/store to memory instructions
- Accumulator for direct address designation
- Memory pointer for direct address designation

R14 (Frame Pointer: FP)

- Index register for load/store to memory instructions
- Frame pointer for reserve/release of dynamic memory area (i.e. function local variables)

R15 (Stack Pointer: SP)

- Index register for load/store to memory instructions
- Stack pointer
- Stack pointer for reserve/release of dynamic memory area

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 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.

R15 functions physically as either the system stack pointer (SSP) or user stack pointer (USP) for the general-purpose registers. When the notation R15 is used in an instruction, this register will function as the "USP" if the "S" flag in the condition code register (CCR) section of the program

status register (PS) is set to '1'. The R15 register will function as the "SSP" if the "S" flag is set to '0'.

Ensure that the S flag value is set to 0 when R15 is recovered from the EIT handler with the RETI instruction.

NOTE

The μC/OS-II FR port never makes use of the USP (User Stack Pointer) and thus, application tasks always execute in **system mode**.

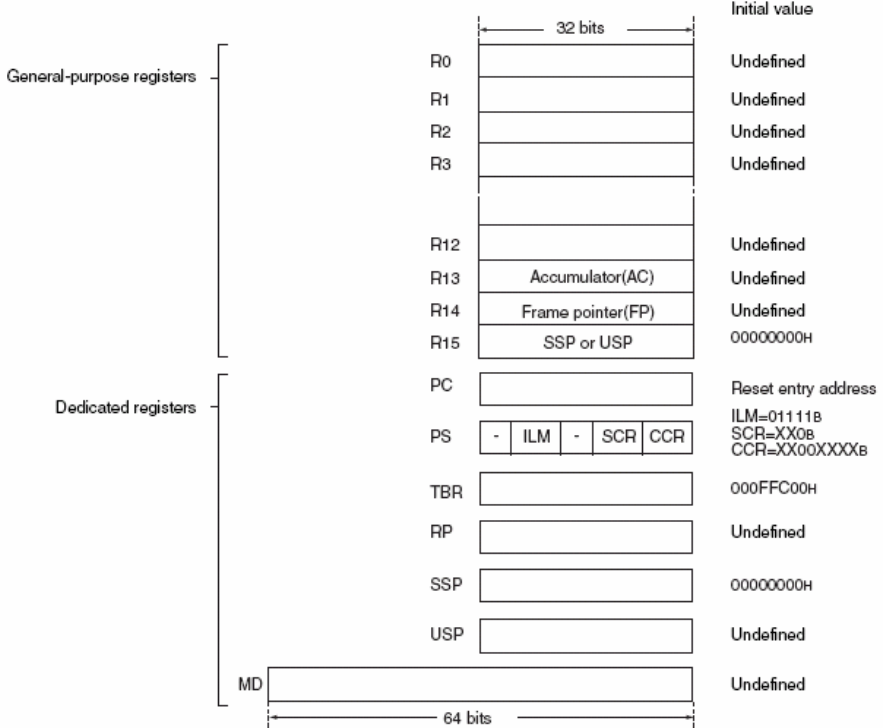


Figure 2-1, FR Programmers Model.

PC (Program Counter)

This register indicates the address containing the instruction that is currently executing. The value of the lowest bit is always read as '0', and therefore all instructions must be written to addresses that are multiples of 2. Following a reset, the contents of the PC are set to the reset entry address contained in the vector table. Because initialization is applied first to the table base register (TBR), the value of the reset vector address will be 0x000FFFC.

PS (Program Status)

The program status register consists of sections that set the interrupt enable level, control the program trace break function in the CPU, and indicate the status of instruction execution.

As shown in Figure 2-2, the PS register is a 32-bit register and is split into five sections, two of which are reserved for future use.

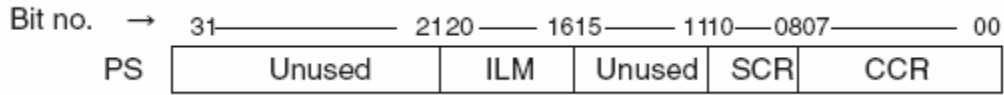


Figure 2-2, Program Status Register.

ILM

The ILM (Interrupt Level Mask) contains 5 bits and determines the level of interrupt that will be accepted. Whenever the “I” flag in the “CCR” register is ‘1’ (i.e. interrupts are enabled), the contents of this register are compared to the level of the current interrupt request. If the value of this register is greater than the level of the request, interrupt processing is activated (i.e. an interrupt will be recognized). Interrupt levels are higher in priority at value approaching ‘0’, and lower in priority at increasing values up to ‘31’. In other words, ALL interrupt levels are enabled when the ILM is set to 31 and ALL interrupts are disabled when ILM is set to 0.

Note that bit “ILM4” differs from the other bits in the register, in that setting values for this bit are restricted. If the original value of the register is in the range 16 to 31, the new value may be set in the range 16 to 31. If an instruction attempts to set a value between 0 and 15, that value will be converted to ‘setting value + 16’ and then transferred. If the original value is in the range 0 to 15, any new value from 0 to 31 may be set. At RESET, the ILM is set to 0x0F (15).

SCR

The SCR (System Condition Code Register) is used by the processor to hold intermediate values during a division instruction. The SCR also contains a bit (the T-bit) which indicates when the CPU is in step trace mode. This is used by a debugger.

CCR

The CCR (Condition Code Register), see figure 2-3, contains six (6) single bit flags that are used to keep track of conditions which occur during instruction execution.

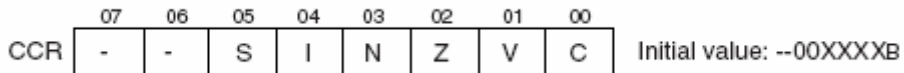


Figure 2-3, Condition Code Register.

“S” Flag

This flag selects the stack pointer to be used. The value ‘0’ selects the system stack pointer (SSP), and ‘1’ selects the user stack pointer (USP). RETI instruction is executable only when the S flag is 0. The μC/OS-II port always assumes that the “S” Flag is cleared (i.e. 0)

“I” Flag

This flag is used to enable/disable system interrupts. The value '0' disables, and '1' enables interrupts.

“N” Flag

This flag is used to indicate positive or negative values when the results of a calculation are expressed in two's complement form. The value '0' indicates positive, and '1' indicates negative.

“Z” Flag

This flag indicates whether the results of a calculations are zero. The value '0' indicates a non-zero value, and '1' indicates a zero value.

“V” Flag

This flag indicates that an overflow occurred when the results of a calculation are expressed in two's complement form. The value '0' indicates no overflow, and '1' indicates an overflow.

“C” Flag

This flag indicates whether a carry or borrow condition has occurred in the highest bit of the results of a calculation. The value '0' indicates no carry or borrow, and '1' indicates a carry or borrow condition. This bit is also used with shift instructions, and contains the value of the last bit that is 'shifted out'.

TBR (Table Base Register)

The TBR designates the table containing the entry address for “EIT” operations (i.e. the TBR points to the Interrupt and Exception Vector Table). When an “EIT” condition occurs, the address of the vector reference is determined by the sum of the contents of this register and the vector offset corresponding to the “EIT” operation. In other words, each interrupt or exception handler contains an address which is stored in this table. When an interrupt or exception occurs, the address of the handler is read from the table based on the offset corresponding to the interrupt or exception:

```
Interrupt or Exception Handler Address = TBR[vector #];
```

or

```
Interrupt or Exception Handler Address = TBR[offset];
```

“**vector #**” is a number between 0 and 256. The RESET vector is 0.

“**offset**” is given by:

$$0x03FC - (\mathbf{vector \ #}) * 4$$

RP (Return Pointer)

The contents of the return pointer (RP) depend on the type of instruction. For a call instruction with a delay slot, the value is the address stored +4, and for a call instruction with no delay slot, the value is the address stored +2. The save data is returned from the “RP” pointer to the “PC” counter by execution of a “RET” instruction.

MD (Multiplication/Division Register)

The multiplication/division register (MD) is a register used to contain the result of multiplication operations, as well as the dividend and result of division operations. The products of multiplication are stored in the "MD" in 64-bit format. In division operations, the dividend must first be placed in the lower 32 bits of the "MD" beforehand. Then as the division process is executed, the remainder is placed in the higher 32 bits of the "MD", and the quotient in the lower 32 bits.

3.00 μC/OS-II Port for FR

We used the Fujitsu Softune toolchain to test the port. Softune contains an editor, a C/C++ compiler, an assembler, a linker/locator and a debugger. The debugger actually contains an FR simulator which allows you to test code prior to running it on actual hardware. We tested the FR port on actual hardware (an MB91403).

Below are a few assumptions about the port:

- You have μC/OS-II V2.81 or higher
- μC/OS-II and application tasks run in **system 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\FR\Softune
```

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

OS_CPU.H	Section 3.02
OS_CPU_C.C	Section 3.03
OS_CPU_A.ASM	Section 3.04
OS_CPU_I.ASM	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 short INT16U;           // (1)
typedef signed   short INT16S;
typedef unsigned long  INT32U;
typedef signed   long  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 Softune compiler documentation, you would find that an **short** is 16 bits, an **int** is 32 bits and a **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 FR 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 PS) on the FR 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 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 the functions expected in OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL() are OS_CPU_SR_Save() and OS_CPU_SR_Restore(), respectively. The code for these two functions is declared in OS_CPU_A.ASM (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
#define OS_ENTER_CRITICAL()  {cpu_sr = OS_CPU_SR_Save();}
#define OS_EXIT_CRITICAL()   {OS_CPU_SR_Restore(cpu_sr);}
#endif
```

3.02.04 OS_CPU.H, Stack growth

The stacks on the FR 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, we force a **software interrupt** and we decided to use vector 64 (0x40). The interrupt handler for this software interrupt is described later (see OS_CPU_A.ASM).

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

```
#define OS_TASK_SW()        __asm(" INT #0x40");
```

3.02.06 OS_CPU.H, Initial ILM

When a task is created, we assume that all interrupts are enabled and that the ILM (Interrupt Level Mask) is set to 31 (0x1F).

Listing 3-6, OS_CPU.H, Task Level Context Switch

```
#define CPU_ILM()          0x1F
```

3.02.06 OS_CPU.H, Function Prototypes

The prototypes in Listing 3-7 are for the functions used to disable and re-enable interrupts using OS_CRITICAL_METHOD #3 and are described later.

Listing 3-7, 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 3-8, 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. The functions that are highlighted will be discussed in this section.

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()

µC/OS-II assumes that tasks run in system mode (the S-flag of the PS register is set to 0).

The Softune compiler passes the a single argument in register R4. Recall 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'. That's why 'p_arg' is passed in R4 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 3-13, OS_CPU_C.C, OSTaskStkInit()

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

    (void)opt; // 'opt' is not used, prevent warning
    pstk = (INT32U *)ptos; // Load stack pointer

    *pstk-- = (INT32U)(CPU_ILM << 16) + 0x00000010; // PS = Enable interrupts
    *pstk-- = (INT32U)task; // PC
    *pstk-- = (INT32U)task; // RP
    *pstk-- = (INT32U)0x14141414; // R14
    *pstk-- = (INT32U)0x13131313; // R13
    *pstk-- = (INT32U)0x12121212; // R12
    *pstk-- = (INT32U)0x11111111; // R11
    *pstk-- = (INT32U)0x10101010; // R10
    *pstk-- = (INT32U)0x09090909; // R9
    *pstk-- = (INT32U)0x08080808; // R8
    *pstk-- = (INT32U)0x07070707; // R7
    *pstk-- = (INT32U)0x06060606; // R6
    *pstk-- = (INT32U)0x05050505; // R5
    *pstk-- = (INT32U)p_arg; // Call to function with argument
    *pstk-- = (INT32U)0x03030303; // R3
    *pstk-- = (INT32U)0x02020202; // R2
    *pstk-- = (INT32U)0x01010101; // R1
    *pstk-- = (INT32U)0x00000000; // R0
    *pstk-- = (INT32U)0xD1D1D1D1; // MDH
    *pstk = (INT32U)0xD0D0D0D0; // MDL

    return ((OS_STK *)pstk);
}
```

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

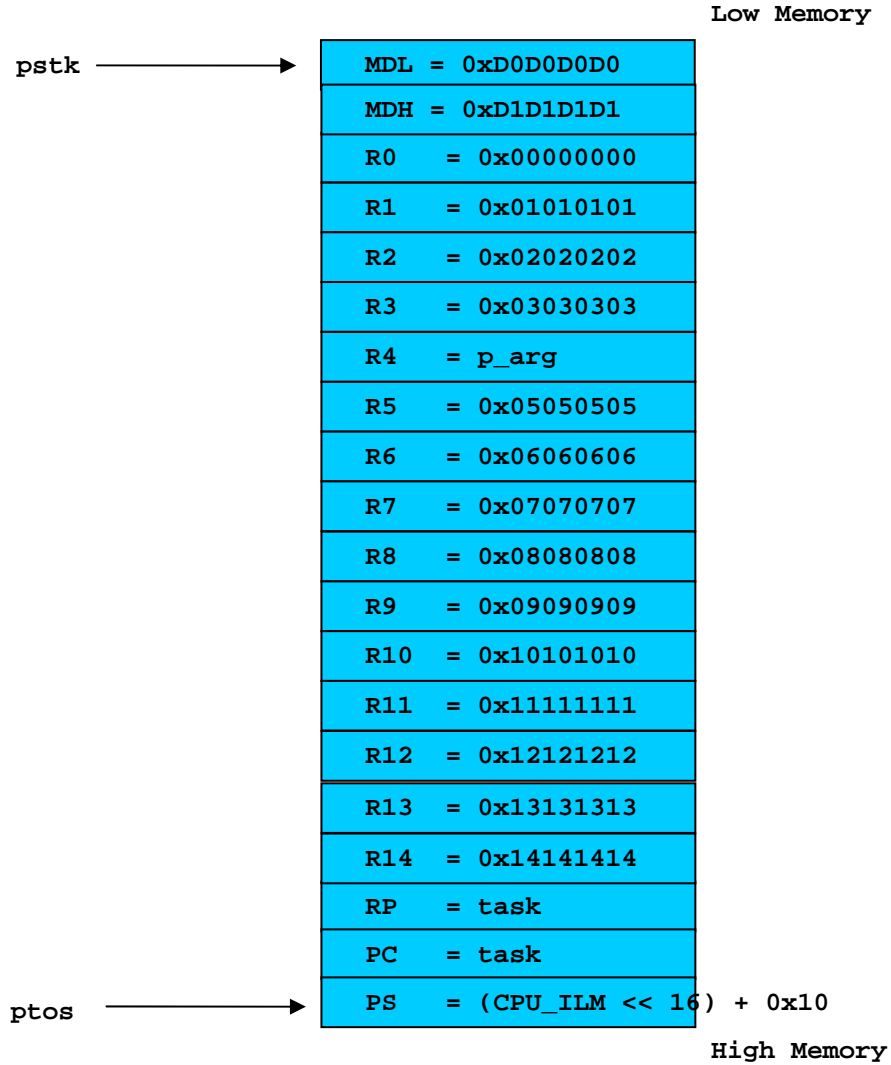


Figure 3-1, The Initial Stack Frame for each Task.

When the task is created, the final value of pstk 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 Softune.

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 Softune.

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.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()
```

3.04.01 OS_CPU_A.ASM, OS_CPU_SR_Save()

The code in listing 3-16 implements the saving of the PS register and then disabling interrupts for OS_CRITICAL_METHOD #3.

When this function returns, R4 contains the state of the PS register prior to disabling interrupts.

Listing 3-16, OS_CPU_SR_Save()

```
_OS_CPU_SR_Save:
    MOV     PS, R4           ; Save state of PS
    ANDCCR #0xEF           ; Disable interrupts
    RET
```

3.04.02 OS_CPU_A.ASM, OS_CPU_SR_Restore()

The code in the listing below implements the function to restore the PS register for OS_CRITICAL_METHOD #3. When called, it's assumed that R4 contains the desired state of the PS register.

Listing 3-17, OS_CPU_SR_Restore()

```
_OS_CPU_SR_Restore:
    MOV     R4, PS         ; Restore state of PS
    RET
```

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 3-18, OSStartHighRdy()

`_OSStartHighRdy:`

```

CALL      _OSTaskSwHook          ; (1) Call user defined task switch hook

LDI       #1, R1                 ; (2) OSRunning = TRUE
LDI       #_OSRunning, R0
STB      R1, @R0

LDI       #_OSTCBHighRdy, R12    ; (3) Resume stack pointer
LD        @R12, R13
LD        @R13, R15

POP_ALL                                ; (4) Restore context of registers

RETI                                           ; (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 (pstk is stored in the OS_TCB of the created task).
- L3-18(4) We then pop ALL the registers (except the PC and PS registers) from the task's stack. This is done by calling an assembly language macro which is declared in OS_CPU_I.ASM and will be described later.
- L3-18(5) By executing a return from interrupt instruction, the FR pops the PC and the PS register from the stack and thus, the FR will start executing the task's code.

3.04.04 OS_CPU_A.ASM, OSCtxSw()

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

A task level context switch occurs when µC/OS-II invokes the macro OS_TASK_SW() which, in the case of the FR port, corresponds to executing a software interrupt instruction (INT 0x40) which causes the CPU to push the PC and PS registers onto the current task’s stack. The pseudo code for this is shown below:

```

OS_TASK_SW()    (i.e. INT 0x40)
    PUSH PS          /* (1) */
    PUSH PC

OSCtxSw:
    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:
    PUSH_ALL                ; (2) Save context of interrupted task

    LDI    #_OSTCBCur, R0   ; (3) OSTCBCur->OSTCBStkPtr = SSP
    LD     @R0, R1
    ST     R15, @R1

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

    LDI    #_OSPrioHighRdy, R0 ; (5) OSPrioCur = OSPrioHighRdy
    LDUB   @R0, R3
    LDI    #_OSPrioCur, R1
    STB    R3, @R1

    LDI    #_OSTCBHighRdy, R0 ; (6) OSTCBCur = OSTCBHighRdy
    LD     @R0, R2
    LDI    #_OSTCBCur, R1
    ST     R2, @R1
    LD     @R2, R15         ; (7) SSP = OSTCBHighRdy->OSTCBStkPtr

    POP_ALL                ; (8) Restore context of interrupted task

    RETI                    ; (9) Return from interrupt

```

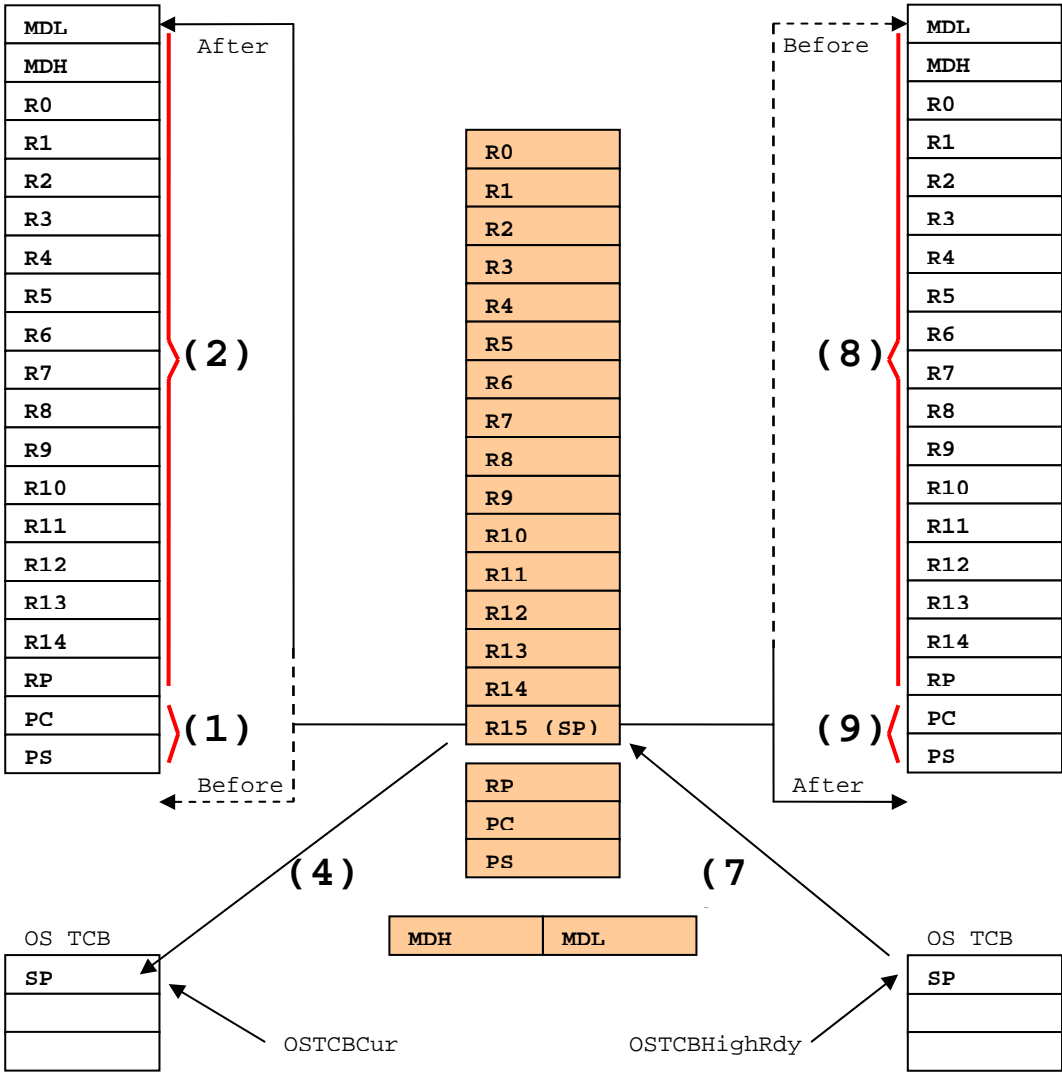


Figure 3-2, Task Level Context Switch.

3.04.05 OS_CPU_A.ASM, 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

    LDI        #_OSPrioHighRdy, R0     ; OSPrioCur = OSPrioHighRdy
    LDUB       @R0, R3
    LDI        #_OSPrioCur, R1
    STB        R3, @R1

    LDI        #_OSTCBHighRdy, R0      ; OSTCBCur = OSTCBHighRdy
    LD         @R0, R2
    LDI        #_OSTCBCur, R1
    ST         R2, @R1
    LD         @R2, R15                ; SSP = OSTCBHighRdy->OSTCBStkPtr

    POP_ALL                                       ; Restore context of interrupted task

    RETI

```

3.05 OS_CPU_I.ASM

OS_CPU_I.ASM is an assembly language file that declares two macros used to save the context of the CPU. These macros are called PUSH_ALL and POP_ALL and are declared as shown below.

```
#macro PUSH_ALL
    ST      RP, @-R15
    STM1    (R8,R9,R10,R11,R12,R13,R14)
    STM0    (R0,R1,R2,R3,R4,R5,R6,R7)
    ST      MDH, @-R15
    ST      MDL, @-R15
#endm

#macro POP_ALL
    LD      @R15+, MDL
    LD      @R15+, MDH
    LDM0    (R7,R6,R5,R4,R3,R2,R1,R0)
    LDM1    (R14,R13,R12,R11,R10,R9,R8)
    LD      @R15+, RP
#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_i.asm"
```

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 Softune debugger is not μC/OS-II aware and thus this file is not needed but should be included in all build for future reference.

4.00 Interrupt Handling

The FR contains an interrupt and exception vector table which contains up to 256 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:
    ANDCCR    #0xEF                                ; (1) Disable interrupts
    PUSH_ALL                                     ; (2) Save context of interrupted task

    LDI      #_OSIntNesting, R0                  ; (3) OSIntNesting++;
    LDUB    @R0, R1
    ADD     #1, R1
    STB     R1, @R0

    CMP     #1, R1                                ; (4) if (OSIntNesting == 1) {
    BNE     _My_ISR_1                          ;
    LDI     #_OSTCBCur, R0                        ; (5)   OSTCBCur->OSTCBStkPtr = SSP;
    LD      @R0, R1
    ST      R15, @R1                             ;   }

_My_ISR_1:
    LDI     #_My_ISR_Handler, R0                ; (6) Call C handler: My_ISR_Handler()
    CALL    @R0

    LDI     #_OSIntExit, R0                       ; (7) OSIntExit();
    CALL    @R0

    POP_ALL                                     ; (8) Restore context of task
    RETI                                         ; (9) Return to task

```

- L4-1(1) When you enter an ISR, you **MUST** disable all interrupts by clearing the I-bit in the CCR.
- L4-1(2) You then **MUST** save all of the registers using the PUSH_ALL macro.
- L4-1(3) You **MUST** to increment µC/OS-II's interrupt nesting counter (OSIntNesting)
- L4-1(4) You **MUST** check to see whether this is the first nested ISR by checking if OSIntNesting got incremented to 1.
- L4-1(5) If this is the first nested ISR level then you **MUST** save the stack pointer into the current task's OS_TCB.
- L4-1(6) 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.
- L4-1(7) 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(8) 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 `POP_ALL` macro.
- L4-1(9) The `RETI` instruction **MUST** be executed to return program execution back to the interrupted task. Note that the PS register is restored which has the I-bit set (i.e. interrupts will be enabled upon returning to the task, even if interrupts were disabled during the execution of the ISR code).

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) Enable interrupts if you want to allow nested interrupts */
    /* (2) Handler the interrupt using C */
    /* (3) Don't forget to clear the interrupt source */
    /* (4) Disable interrupts (if you enabled them) */
}
```

- L4-2(1) As indicated, you may enable interrupts (by setting the I-bit in the CCR) if you want to allow nested interrupts.
- L4-2(2) You can now service the interrupting device using the C programming language (instead of doing that in assembly language).
- L4-2(3) 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(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 FR CPU and the target hardware.

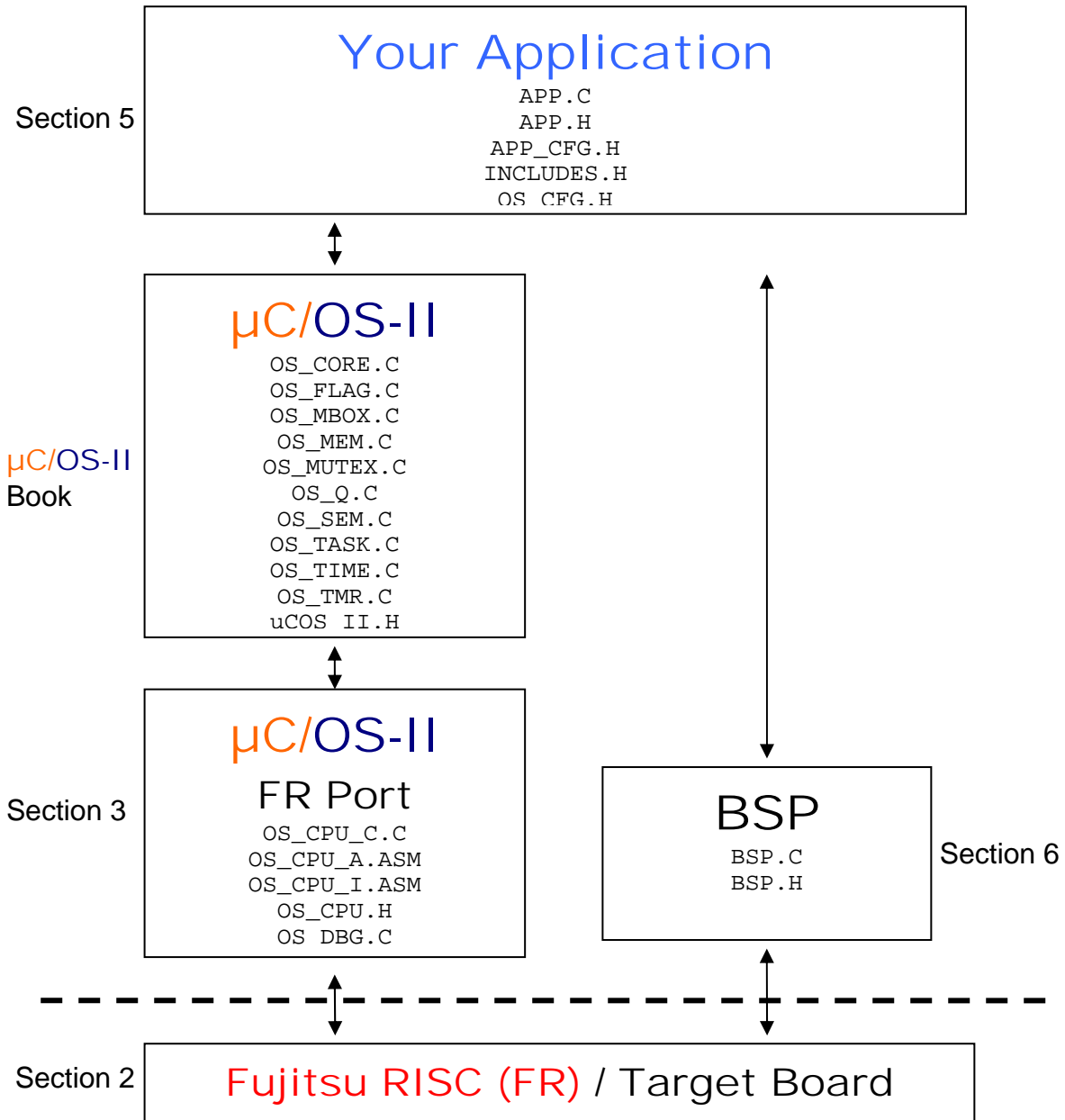


Figure 5-1, Relationship between modules.

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.

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)

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

#ifdef 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 */ (4)
        LED_Toggle(1);                         (5)
        OSTimeDly(OS_TICKS_PER_SEC / 10);
    }
}

```

- 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 BPS 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

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

#include <app_cfg.h>
#include <ucos_ii.h>

#include <_fr.h>

#if OS_VIEW_MODULE > 0
#include <lib_def.h>
#include <lib_mem.h>
#include <lib_str.h>
#endif

#if OS_VIEW_MODULE > 0
#include <os_viewc.h>
#include <os_view.h>
#endif

#include <bsp.h>
```

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
- ISR Handlers
- LED control functions
- Reading switches
- Setting up the interrupt controller
- Setting up communication channels
- Etc.

A BSP consist of 2 files: `BSP.C` , `BSP_A.ASM` 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 ISR 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 is in `BSP_A.ASM` and is called `BSP_TickISR()`. The C ISR handler is found in `BSP.C` and is called `BSP_TickISR_Handler()`.

7.00 Conclusion

This application note presented a 'generic' port for the FR processors. 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.

References

MicroC/OS-II, The Real-Time Kernel, 2nd Edition

Jean J. Labrosse
R&D Technical Books, 2002
ISBN 1-5782-0103-9

Contacts

CMP Books, Inc.

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

Micrium

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

Notes