

Micrium

Empowering Embedded Systems

μC/OS-II

and the
Renesas M32C Processors

Application Note

AN-1020

About Micrium

Micrium provides high-quality embedded software components in the industry by way of engineer-friendly source code, unsurpassed documentation, and customer support. The company's world-renowned real-time operating system, the Micrium **μC/OS-II**, features the highest-quality source code available for today's embedded market. Micrium delivers to the embedded marketplace a full portfolio of embedded software components that complement **μC/OS-II**. A TCP/IP stack, USB stack, CAN stack, File System (FS), Graphical User Interface (GUI), as well as many other high quality embedded components. Micrium's products consistently shorten time-to-market throughout all product development cycles. For additional information on Micrium, please visit www.micrium.com.

About μC/OS-II

Thank you for your interest in **μC/OS-II**. **μC/OS-II** is a preemptive, real-time, multitasking kernel. **μC/OS-II** has been ported to over 45 different CPU architectures and now, has been ported to the Renesas M32C CPU.

μC/OS-II is small yet provides all the services you would expect from an RTOS: task management, time and timer management, semaphore and mutex, message mailboxes and queues, event flags and much more.

You will find that **μC/OS-II** delivers on all your expectations and you will be pleased by its ease of use.

Licensing

μC/OS-II is provided in source form for **FREE** evaluation, for educational use or for peaceful research. If you plan on using **μC/OS-II** in a commercial product you need to contact Micrium to properly license its use in your product. We provide ALL the source code with this application note for your convenience and to help you experience **μC/OS-II**. The fact that the source is provided **DOES NOT** mean that you can use it without paying a licensing fee. Please help us continue to provide the Embedded community with the finest software available. Your honesty is greatly appreciated.

Manual Version

If you find any errors in this document, please inform us and we will make the appropriate corrections for future releases.

Version	Date	By	Description
V.1.00	2007/02/26	BAN	Initial version.

Software Versions

This document may or may not have been downloaded as part of an executable file containing the code described herein plus (possibly) additional application or board support code. If so, then the versions of the Micrium software modules in the table below would be probably included. In either case, the software port described in this document uses the module versions in the table below

Module	Version	Comment
μC/OS-II	V2.83	

Document Conventions

Numbers and Number Bases

- Hexadecimal numbers are preceded by the “0x” prefix and displayed in a monospaced font. Example: `0xFF886633`.
- Binary numbers are followed by the suffix “b”; for longer numbers, groups of four digits are separated with a space. These are also displayed in a monospaced font. Example: `0101 1010 0011 1100b`.
- Other numbers in the document are decimal. These are displayed in the proportional font prevailing where the number is used.

Typographical Conventions

- Hexadecimal and binary numbers are displayed in a monospaced font.
- Code excerpts, variable names, and function names are displayed in a monospaced font. Functions names are always followed by empty parentheses (e.g., `OS_Start()`). Array names are always followed by empty square brackets (e.g., `BSP_Vector_Array[]`).
- File and directory names are always displayed in an italicized serif font. Example: */Micrium/Software/uCOS-II/Source/*.
- A bold style may be layered on any of the preceding conventions—or in ordinary text—to more strongly emphasize a particular detail.
- Any other text is displayed in a sans-serif font.

Table of Contents

1.	Introduction	6
2.	The M32C Programmer's Model	7
3.	μC/OS-II Port for the M32C	9
3.01	Directories and Files	10
3.02	μC/OS-II Port: <i>os_cpu.h</i>	12
3.02.01	Data Types	12
3.02.02	Critical Sections	12
3.02.03	Stack Growth	13
3.02.04	Task-Level Context Switch	13
3.02.05	Function Prototypes	13
3.03	μC/OS-II Port: <i>os_cpu.c.c</i>	14
3.03.01	Task Stack Initialization: <code>OSTaskStkInit()</code>	14
3.03.02	M32C Stack Frame	16
3.03	μC/OS-II Port: <i>os_cpu_a.asm</i>	17
3.03.01	Beginning Multitasking: <code>OSStartHighRdy()</code>	17
3.03.02	Performing a Task-Level Context Switch: <code>OSCtxSw()</code>	18
3.03.03	Performing an Interrupt-Level Context Switch: <code>OSIntCtxSw()</code>	19
3.03.04	Tick ISR: <code>OSTickISR()</code>	20
3.04	μC/OS-II Port: <i>os_dbg.c</i>	21
4.	Interrupt Handling for the M32C	22
4.01	Defining Vector Tables	23
4.01.01	IAR Vector Table	23
4.01.02	HEW Vector Table	25
4.02	Example Interrupt Service Routine	27
	Licensing	28
	References	28
	Contacts	28

1. Introduction

This document describes the official Micrium port for μC/OS-II to the Renesas M32C family of processors. Figure 1-1 diagrams the relationship between your application, μC/OS-II, the port code and the BSP (Board Support Package). Relevant sections of this application note are labeled on the figure

If this appnote was downloaded in a packaged executable zip file, then it should have been found in the directory `/Micrium/Appnotes/AN1xxx-RTOS/AN1020-uCOS-II-Renesas-M32C` and the code files referred to herein are located in the directory structure displayed in Section 3.01; these files are also described in Section 3.01. The M32C has been ported on both the IAR and HEW tools.

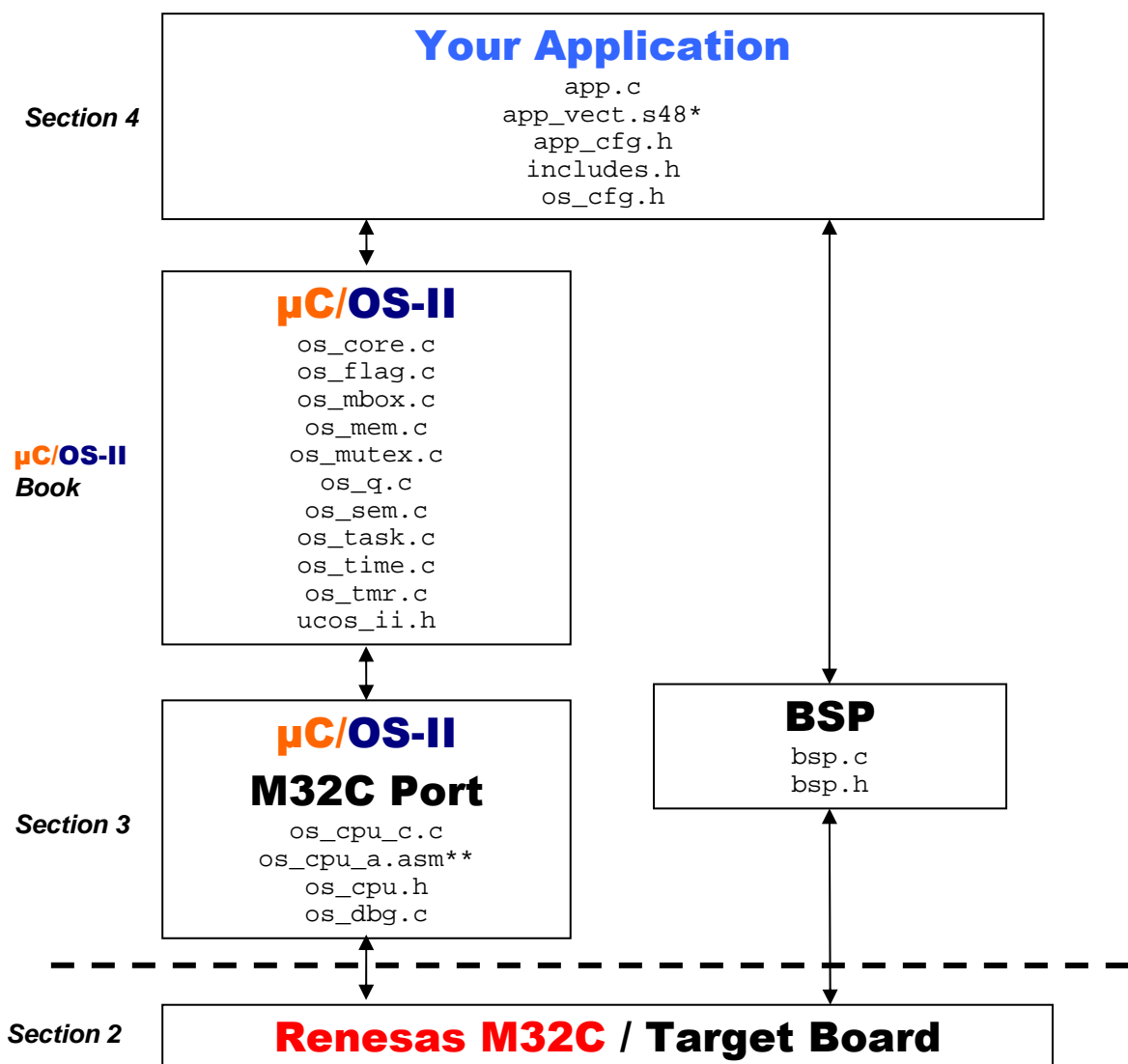


Figure 1-1. Code Block Diagram.

*For the HEW port, this file is named `linker.inc` and is part of the BSP.

**For the HEW port, this file is named `os_cpu_a.a30`.

2. The M32C Programmer's Model

The visible CPU registers in a M32C processor are shown in Figure 2-2. Besides the data registers $R0-R3$ and the FLG (Flag) registers, which are each 16 bits wide, these registers are all 24-bit registers. The registers are as follows:

- **Data Registers: $R0, R1, R2, R3$.** These are mainly used for transfers and arithmetic/logic operations. The $R0$ and $R1$ registers can each be separated into two 8-bit registers. 32-bit registers can be formed by combining $R2$ and $R0$ or by combining $R3$ and $R1$; these registers are referred to as $R2R0$ and $R3R1$.
- **Address Registers: $A0, A1$.** These are used primarily for address register indirect addressing and address register relative addressing, in addition to being used for transfers and arithmetic/logic operations.
- **Static Base Register, SB .**
- **Frame Base Register, FB .**
- **Interrupt Table Register, $INTB$.** This register contains the start address of the relocatable interrupt vector table.
- **Program Counter, PC .** This register indicates the address of the next instruction to be executed.
- **Flag Register, FLG .** This register indicates the CPU states. The bits of this register are described in Figure 2-3.
- **Stack Pointers, USP and ISP .** These are the stack pointers for the two processor modes, user and interrupt. The μC/OS-II M32C port requires the processor to be initialized in interrupt mode, and only the ISP will be used.

The data registers, address registers, and frame base register comprise a register bank, of which two are present on the processor. This second bank of register could be used to provide fast context switch (when an interrupt occurs, for instance); however the μC/OS-II port does not use the second bank of registers.

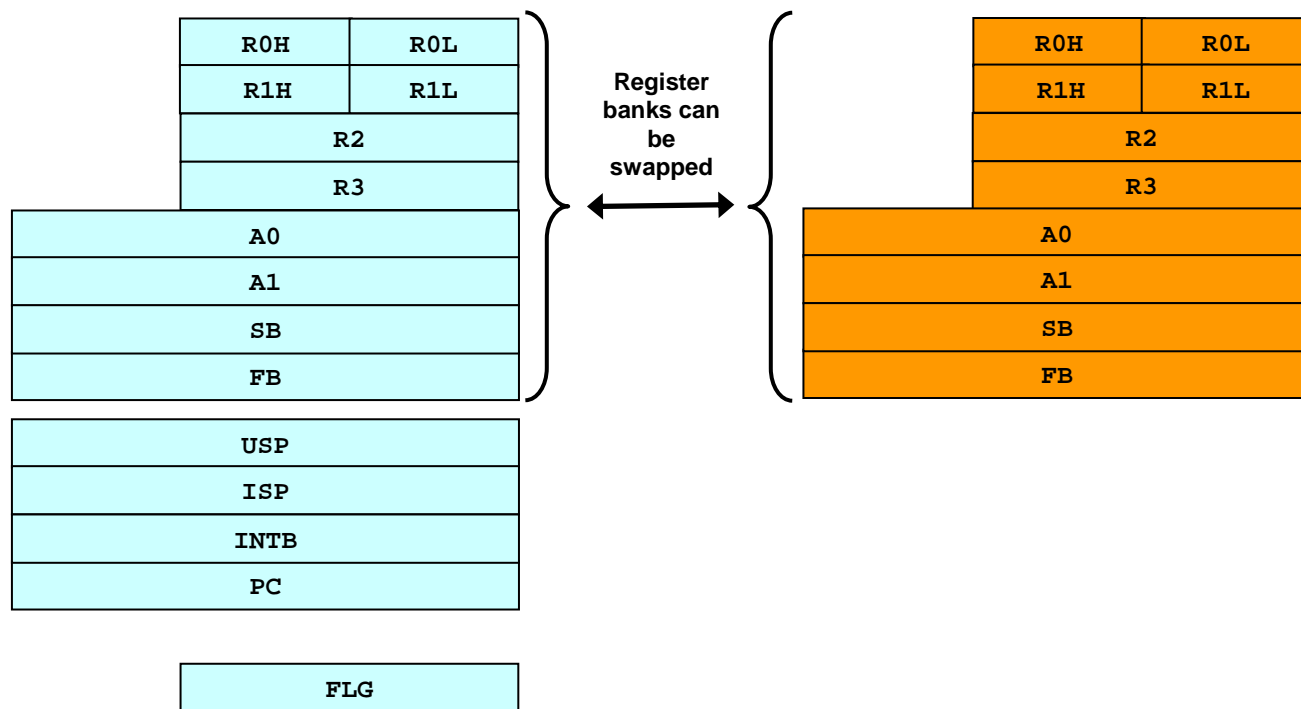


Figure 2-2. M32C Registers

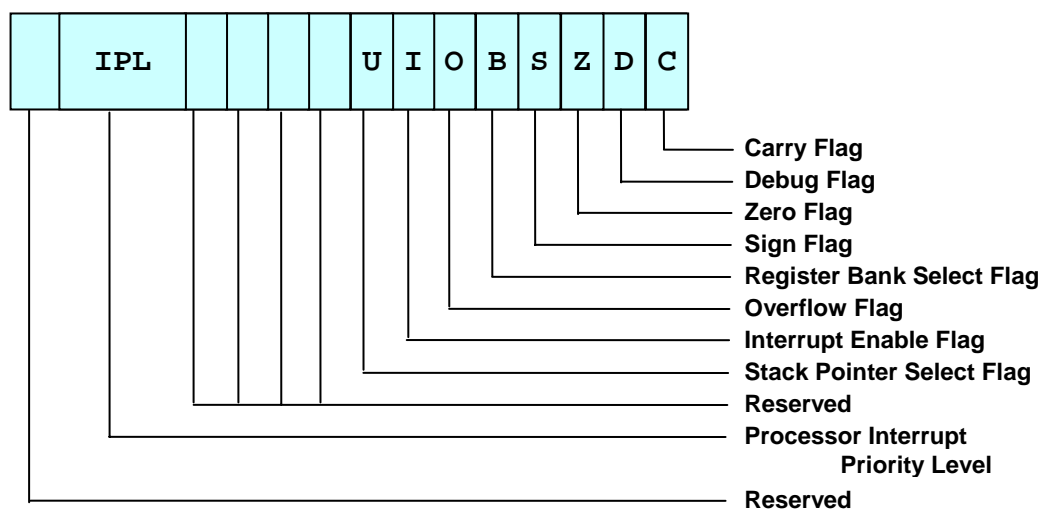


Figure 2-3. M32C FLAG Register

3. μC/OS-II Port for the M32C

Two toolchains were used to compile the M32C port. The first is IAR EW (Embedded Workbench) for M32C processors, V3.21A. The second is Renesas's HEW (High-Performance Embedded Workshop) V4.00 using the NC308WA compiler. Though code for these tools are compatible, meaning that a project compilable under one can be adapted for the other, the expectations of the tools are not identical. Table 3-1 summarizes the differences between the assembly code for the projects created for these toolchains. Differences between the tools will also be discussed at appropriate points in the appnote, where some specific difference is encountered.

The project was tested on a Renesas SKP32C84 evaluation board, as shown in Figure 3-1. The port could also be used on other M32C processors with no change to the μC/OS-II port; however, a suitable BSP would need to be provided. This would initialize the timer used for the μC/OS-II tick interrupt in addition to initializing the hardware peripherals used by the application (such as GPIO pins or a UART).



Figure 3-1. Renesas SKP32C84 Evaluation Board

Item	IAR	HEW
Extern directive	EXTERN	.glb
Import directive	PUBLIC	.glb
Declare 32-bit constant	DC32	.lword
End of file	END	.END
Section directive (for function)	Functions are preceded by: .EVEN FncName: <function> (The name of the function is externed at the beginning of the assembly file.)	Functions are be preceded by: .SECTION program .GLB _FncName _FncName: <function>
Variables	As used in C	C name preceded by underscore

Table 3-1. Differences between IAR and HEW in Port's Assembly Code

3.01 Directories and Files

If this file, *AN-1020*, were downloaded as part of an executable zip file, then the code files referred to herein are located in the directory structure shown in Figure 3-2. Additional application or BSP code may have been included in that executable zip file that is not listed in Figure 3-2.

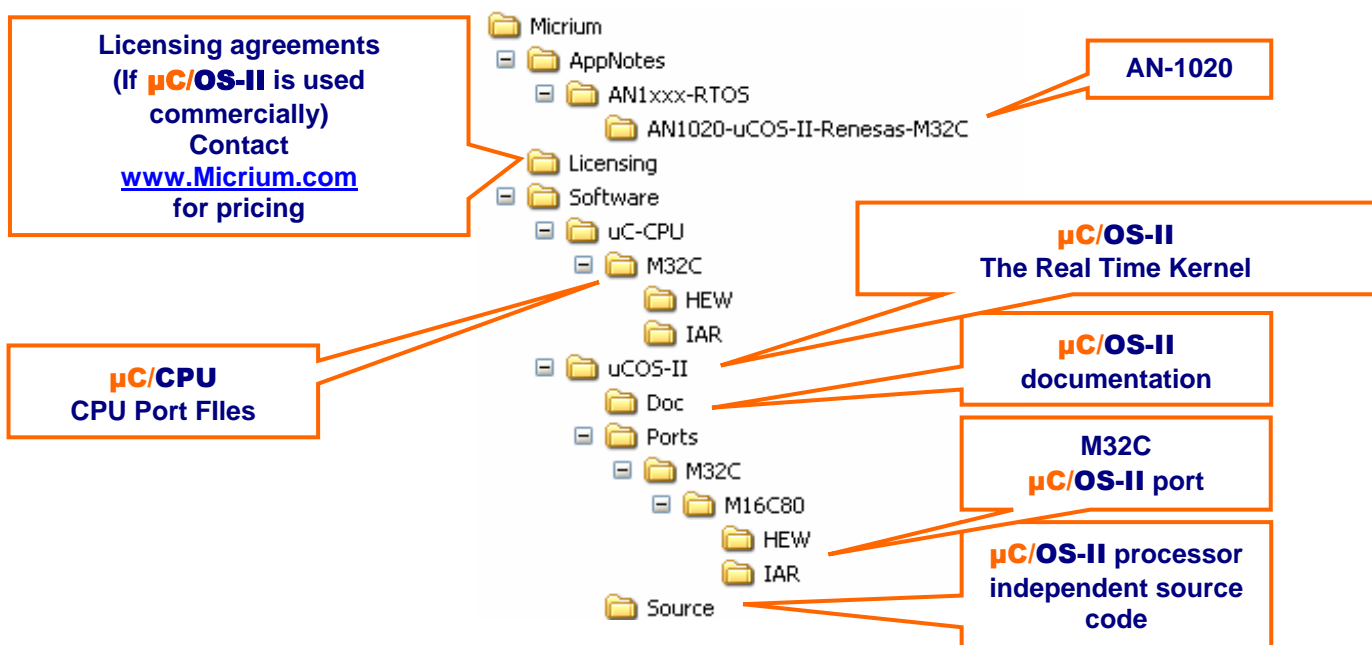


Figure 3-2, Directory Structure

The source code for the **μC/OS-II** port with the IAR toolchain is found in the following directory:

`\Micrium\Software\uCOS-II\M32C\M16C80\IAR`

Similarly, for the HEW toolchain using the NC308WA compiler, the **μC/OS-II** port is in

`\Micrium\Software\uCOS-II\M32C\M16C80\HEW`

Each port consists of four files:

- *os_cpu.h* contains processor- and implementation-specific `#define` constants, macros, and typedefs.
- *os_cpu_a.asm* (named *os_cpu_a.a30* for the HEW port) contains five simple assembly language functions.
- *os_cpu_c.c* contains ten simple C-language functions, including the function that initializes the task stacks.
- *os_dbg.c* was added in V2.62 of **μC/OS-II** to allow a kernel-aware debugger to extract information about **μC/OS-II** and its configuration.

Also included is the **μC/CPU** port; for the IAR toolchain, this is found in the directory

`\Micrium\Software\uC-CPU\M32C\IAR`

Similarly, for the HEW toolchain using the NC308WA compiler, the **μC/CPU** port is in

`\Micrium\Software\uC-CPU\M32C\HEW`

In both cases, the port consists of two files:

- *cpu.h* contains processor- and implementation-specific `#define` constants, macros, and typedefs. The application code should use these defines to create platform-independent source code.
- *cpu_a.s48* (named *cpu_a.a30* for the HEW port) includes functions, written in assembly language, for enabling and disabling interrupts and saving and restoring the CPU flags.

3.02 μC/OS-II Port: *os_cpu.h*

3.02.01 Data Types

μC/OS-II is made compiler- and CPU-independent by defining in the port the data types used in the source code. These, shown in Listing 3-1, are the same for both the HEW and IAR toolchains.

```
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;
typedef signed   char  INT8S;
typedef unsigned int   INT16U;
typedef signed   int   INT16S;
typedef unsigned long  INT32U;
typedef signed   long  INT32S;
typedef float         FP32;
typedef double        FP64;

typedef unsigned int   OS_STK;
typedef INT16U        OS_CPU_SR;
```

Listing 3-1. *os_cpu.h*: Data Type Definitions

3.02.02 Critical Sections

μC/OS-II, as with all real-time kernels, needs to disable interrupts in order to ensure critical sections are evaluated atomically. Macros are provided to disable and enable interrupts: `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`, respectively. μC/OS-II defines three ways to disable interrupts, but only one of these methods needs to be used. The preferred method is typically method number 3. Note that if critical method 3 is used, a local variable `cpu_sr` must be allocated and initialized to zero in the user code.

```
#if OS_CRITICAL_METHOD == 1
#define OS_ENTER_CRITICAL() asm("FCLR I")
#define OS_EXIT_CRITICAL()  asm("FSET I")
#endif

#if OS_CRITICAL_METHOD == 2
#define OS_ENTER_CRITICAL() asm("PUSHC FLG"); asm("FCLR I") /* (1) */
#define OS_EXIT_CRITICAL()  asm("POPC FLG")
#endif

#if OS_CRITICAL_METHOD == 3
#define OS_ENTER_CRITICAL() asm("STC FLG, $@", cpu_sr);asm("FCLR I")
#define OS_EXIT_CRITICAL()  asm("LDC $@, FLG", cpu_sr)
#endif
```

Listing 3-2. *os_cpu.h*: Critical Sections

Listing 3-2, Note 1: Code is provided for critical method 2 for completeness only. This code will not work if data have been stored on the stack after entering the critical section, but have not been all removed when exiting the critical section.

3.02.03 Stack Growth

The stacks on the Renesas M32C grow from high memory to low memory; consequently, `OS_STK_GROWTH` is set to 1.

```
#define OS_STK_GROWTH 1
```

Listing 3-3. *os_cpu.h*: Stack Growth

3.02.04 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 assembly-language code. For the M32C port, a task switch is triggered using software interrupt 0. The task switch handler, `OSCtxSw()`, will need to be placed on vector 0.

```
#define OS_TASK_SW() asm("INT #0")
```

Listing 3-4. *os_cpu.h*: Context Switch

3.02.05 Function Prototypes

The prototypes in Listing 3-5 are for function defined in *os_cpu_a.asm* (which is named *os_cpu_a.a30* for the HEW port).

```
void OScTxSw      (void);  
void OSIntCtxSw   (void);  
void OSStartHighRdy (void);  
void OSTickISR    (void);
```

Listing 3-5. *os_cpu.h*: Function Prototypes

3.03 μC/OS-II Port: *os_cpu_c.c*

A **μC/OS-II** port requires that you write ten fairly simple C functions, as listed below. The functions whose names are written in bold are non-empty. However, any of the hooks may be configured to provide some functionality important for your application.

- **OSTaskStkInit()** is typically the only function that need be defined for a port. This function is discussed in the subsection 3.03.01.
- **OSInitHookBegin()** is called by **μC/OS-II's** `OSInit()` at the very beginning of `OSInit()`, giving the opportunity to add additional port-specific initialization. In this case, the global variable `OSTmrCtr` (used by the `os_tmr.c` module) is initialized to 0.
- `OSInitHookEnd()`
- **OSTaskCreateHook()** is called by **μC/OS-II's** `OSTaskCreate()` or `OSTaskCreateExt()` when a task is created. If **μC/OS-View** (which performs task profiling as run-time) is included as part of the build, the function `OSView_TaskCreateHook()` is called, initializing **μC/OS-View's** data for that task.
- `OSTaskDelHook()`
- `OSTaskIdleHook()`
- `OSTaskStatHook()`
- **OSTaskSwHook()** is called when a context switch occurs. This function allows the code to measure the execution time of a task for instance. In this case, the **μC/OS-View** task switch hook, `OSView_TaskSwHook()`, is called, allowing **μC/OS-View** to do exactly that.
- `OSTCBInitHook()`
- **OSTimeTickHook()** is called at the beginning of `OSTimeTick()`. This function calls `OSView_TickHook()`. It also determines whether it is time to update the **μC/OS-II** timers; if it is time to update the timers, the timer task is signalled.

3.03.01 Task Stack Initialization: **OSTaskStkInit()**

The code in Listing 3-6 initializes the stack frame for the task being created. The task receives an additional argument, `p_arg`, which is assigned into a register when the task stack is created. Because the initial value of the CPU registers is largely unimportant, the initial values include some indication of the register name, which might be helpful for debugging and examining the stacks in RAM.

The task stack structure is shown in Figure 3-3. When a context switch occurs, the bottom eight stack entries will be returned to the CPU registers using a stack pop (**POP.M**) command. Because the upper two entries are returned to the CPU registers with a **REIT** command, and these are formatted in the manner expected by the processor architecture.

```
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT16U opt)
```

```

{
  INT16U  *pstk16;

  pstk16  = (INT16U *)ptos;

  *pstk16-- = (INT32U)pdata >> 16L;
  *pstk16-- = (INT32U)pdata & 0x0000FFFFL;
  *pstk16-- = (INT32U)task >> 16L;
  *pstk16-- = (INT32U)task & 0x0000FFFFL;

  *pstk16-- = (INT16U)0x0040;                /* (1) */
  *pstk16-- = (INT32U)task >> 16L;
  *pstk16-- = (INT32U)task & 0x0000FFFFL;

  *pstk16-- = (INT16U)0xFBFB;
  *pstk16-- = (INT16U)0xFBFB;
  *pstk16-- = (INT16U)0x3B3B;
  *pstk16-- = (INT16U)0x3B3B;
  *pstk16-- = (INT16U)0xA1A1;
  *pstk16-- = (INT16U)0xA1A1;
  *pstk16-- = (INT16U)0xA0A0;
  *pstk16-- = (INT16U)0xA0A0;
  *pstk16-- = (INT16U)0x3333;
  *pstk16-- = (INT16U)0x2222;
  *pstk16-- = (INT16U)0x1111;
  *pstk16  = (INT16U)0x0000;

  return ((OS_STK *)pstk16);                /* (2) */
}

```

Listing 3-6. *os_cpu.c*: OSTaskStkInit()

Listing 3-6, Note 1: The task stack is initialized so that interrupts are enabled the first time the task executes. If interrupts were not enabled, then no context switch could ever occur.

Listing 3-6, Note 2: The pointer to the top of the stack is returned.

3.03.02 M32C Stack Frame

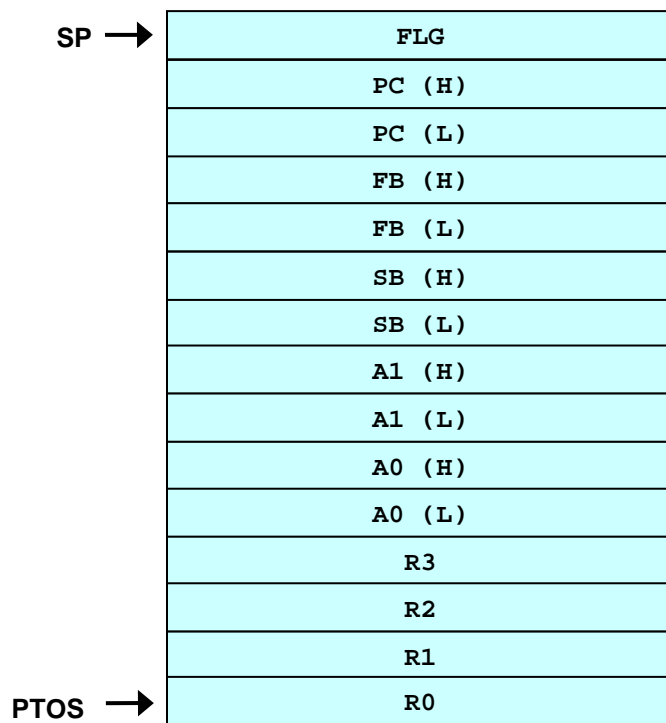


Figure 3-3. Task Stack Frame

3.03 μC/OS-II Port: *os_cpu_a.asm*

A μC/OS-II port requires three fairly simple assembly-language function:

- `OSStartHighRdy()` is called by `OSStart()` to start running the highest-priority task.
- `OSTxSw()` performs a task-level context switch.
- `OSIntCtxSw()` performs an interrupt-level context switch.

In addition, a fourth function, `OSTickISR()` is defined in *os_cpu_a.asm*. This function is the ISR for the μC/OS-II tick IRQ. The function `OSTickISR()` should be placed on the appropriate IRQ vector.

The assembly code shown in this chapter is for the IAR compiler. The HEW code differs only in the assembler directives used, such as those which specify the memory segment into which the code will be placed. Table 3-1 lists these difference; in addition, please consult *os_cpu_a.a30* in

`\Micrium\Software\uCOS-II\M32C\M16C80\HEW`

for a full listing of the HEW assembly source code.

3.03.01 Beginning Multitasking: `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.

```

OSStartHighRdy:
  JSR      OSTaskSwHook          ; Note 1

  MOV.L   OSTCBHighRdy, A0      ; ISP = OSTCBHighRdy->OSTCBStkPtr
  LDC     [A0], ISP

  MOV.B   #01H, OSRunning      ; Note 2: OSRunning = TRUE
  POPM    R0,R1,R2,R3,A0,A1,SB,FB ; Note 3

  REIT
  
```

Listing 3-7. *os_cpu_a.asm*: `OSStartHighRdy()`

Listing 3-7, Note 1: The task switch hook is called.

Listing 3-7, Note 2: Set `OSRunning` to `TRUE`.

Listing 3-7, Note 3: The data registers, address registers, stack base register, and frame base register are popped from the task stack into the CPU registers. The next two stack entries, which contain the `FLG` and `PC` registers will be popped with the `REIT` command.

3.03.02 Performing a Task-Level Context Switch: `OSCtxSw()`

When a task yields control of the CPU, the `OS_TASK_SW()` macro is invoked. For the M32C, this macro causes a interrupt, IRQ 0. The function `OSCtxSw()` should be placed on the IRQ 0 vector.

```

OSCtxSw:
  PUSHM      R0,R1,R2,R3,A0,A1,SB,FB      ; Note 1

  MOV.L     OSTCBCur, A0                  ; Note 2: OSTCBCur->OSTCBStkPtr = SP
  STC      ISP, [A0]

  JSR      OSTaskSwHook                  ; Note 3: OSTaskSwHook()

  MOV.L     OSTCBHighRdy, OSTCBCur       ; Note 4: OSTCBCur = OSTCBHighRdy

  MOV.B     OSPrioHighRdy, OSPrioCur    ;           OSPrioCur = OSPrioHighRdy

  MOV.L     OSTCBHighRdy, A0             ; Note 5: SP           = OSTCBHighRdy->OSTCBStkPtr
  LDC      [A0], ISP

  POPM      R0,R1,R2,R3,A0,A1,SB,FB     ; Note 6

  REIT
  
```

Listing 3-8. *os_cpu_a.asm*: `OSCtxSw()`

Listing 3-8, Note 1: The data registers, address registers, stack base register, and frame base register are saved on the previous task's stack (the stack of the task yielding the processor). The PC and FLG register were already placed on the stack when the interrupt occurred.

Listing 3-8, Note 2: The stack pointer is saved.

Listing 3-8, Note 3: The task switch hook function is invoked.

Listing 3-8, Note 4: `OSTCBCur` and `OSPrioCur` are assigned the proper values for the new task, which are currently `OSTCBHighRdy` and `OSPrioHighRdy`.

Listing 3-8, Note 5: The stack pointer is initialized.

Listing 3-8, Note 6: The processor registers are popped from the new task's stack. The PC and FLG registers will be popped from the stack with the `REIT` command.

3.03.03 Performing an Interrupt-Level Context Switch: OSIntCtxSw()

When an ISR completes, OSIntExit() is called to determine whether a more important task than the interrupt task needs to execute. If that is the case, OSIntExit() determines which task should run next and calls OSIntCtxSw(). The code for this function is identical to OSCtxSw(), except that neither the registers nor the stack pointer need to be save on entry to the function.

```

OSIntCtxSw:
  JSR      OSTaskSwHook          ; Note 1: OSTaskSwHook()

  MOV.L   OSTCBHighRdy, OSTCBCur ; Note 2: OSTCBCur = OSTCBHighRdy

  MOV.B   OSPrioHighRdy, OSPrioCur ;      OSPrioCur = OSPrioHighRdy

  MOV.L   OSTCBHighRdy, A0      ; Note 3: SP      = OSTCBHighRdy->OSTCBStkPtr
  LDC     [A0], ISP

  POPM    R0,R1,R2,R3,A0,A1,SB,FB ; Note 4
  REIT
  
```

Listing 3-9. *os_cpu_a.asm*: OSIntCtxSw()

Listing 3-9, Note 1: The task switch hook function is invoked.

Listing 3-9, Note 2: OSTCBCur and OSPrioCur are assigned the proper values for the new task, which are currently OSTCBHighRdy and OSPrioHighRdy.

Listing 3-9, Note 3: The stack pointer is initialized.

Listing 3-9, Note 4: The processor registers are popped from the new task's stack. The PC and FLG registers will be popped from the stack with the REIT command.

3.03.04 Tick ISR: OSTickISR()

The μC/OS-II tick ISR, OSTickISR() is also included in *os_cpu_a.asm*. This function can be used on any M32C platform, as long as no further acknowledgement is required for the timer hardware (like resetting the timer counts, for instance). In any case, this function should serve as a template for any ISR for the M32C.

```

OSTickISR:
    PUSHM      R0,R1,R2,R3,A0,A1,SB,FB      ; Note 1: Save current task's registers
    INC.B      OSIntNesting                 ; Note 2: OSIntNesting++
    CMP.B      #1,OSIntNesting              ; Note 3: if (OSIntNesting == 1) {
    JNE        OSTickISR1
    MOV.L      OSTCBCur, A0                  ;           OSTCBCur->OSTCBStkPtr = SP
    STC        ISP, [A0]                    ;           }
OSTickISR1:
    JSR        OSTimeTick                   ; Note 4: OSTimeTick()
    JSR        OSIntExit                    ; Note 5: OSIntExit()
    POPM      R0,R1,R2,R3,A0,A1,SB,FB      ; Note 6: Restore current task's registers
    REIT
  
```

Listing 3-10. *os_cpu_a.asm*: OSTickISR()

Listing 3-10, Note 1: The current task's registers are saved.

Listing 3-10, Note 2: The interrupt nesting level is incrementing.

Listing 3-10, Note 3: If a task was interrupted, then the stack pointer is saved..

Listing 3-10, Note 4: OSTimeTick() is called to inform μC/OS-II of the interrupt.

Listing 3-10, Note 5: OSIntExit() is called to determine if a task with a higher priority than the one interrupted exists. If one does, then OSIntExit() never returns.

Listing 3-10, Note 6: The current task's registers are restored.

3.04 μC/OS-II Port: *os_dbg.c*

This file was added in μC/OS-II V2.62 to allow a kernel-aware debugger to extract information about μC/OS-II and its configuration. If your debugger does not need this file, you may omit it in your build.

4. Interrupt Handling for the M32C

The M32C processors contain two vector tables. The **fixed vector table** is located at the highest addresses—from 0xFFFFDC to 0xFFFFF—contains software and hardware non-maskable interrupts. These are listed in Table 4-1. The **relocatable vector table** contains maskable peripheral-function interrupts and, as its name implies, does not need to be at a fixed location. The location of this table should be assigned to the INTB register in the processor startup code. The purpose of the vectors in the fixed vector table does not vary between different M32C processors; however, the purpose of the vectors in the relocatable vector table depends on the peripherals present on a particular chip. An example relocatable vector table is given in Table 4-2 for the M32C processor.

Interrupt source	Vector Table Address
Undefined instruction	0xFFFFDC
Overflow	0xFFFFE0
BRK instruction	0xFFFFE4
Address match	0xFFFFE8
Single step	0xFFFFEC
Watchdog timer	0xFFFFF0
DBC	0xFFFFF4
NMI	0xFFFFF8
Reset	0xFFFFFC

Table 4-1. R32C Fixed Vector Table

Interrupt source	IRQ Number
BRK instruction	0
DMA0	8
DMA1	9
DMA2	10
DMA3	11
Timer A0	12
Timer A1	13
Timer A2	14
Timer A3	15
Timer A4	16
UART0 Transmission, NACK	17
.	.
.	.
.	.
.	.

Table 4-2. Example Relocatable Vector Table (for the M32C/84)

4.01 Defining Vector Tables

The BSP or application code should define fixed and relocatable vector tables. The vectors in the fixed vector table need not be used; indeed, some should not be used. The reset vector, for instance, should only direct program execution to the startup code. The other vectors in the fixed vector table should execute a dummy handler containing the return from interrupt instruction (REIT) if unused for other purposes.

The relocatable vector table must contain two entries; the others may be set to zero or assigned a dummy handler. The first entry that must be present is the μC/OS-II context switch handler, OSctxSw(), which should be placed on vector zero. The second is the μC/OS-II tick ISR handler, OSTickISR(), which should be placed on the vector associated with the timer that generates the μC/OS-II time tick.

4.01.01 IAR Vector Table

For the IAR project ports, the vector tables are defined in the assembly file *app_vect.s48*, included as part of the application code. Listing 4-1 and 4-2 present the fixed and relocatable vector tables, respectively.

```

MODULE ?vectors1

EXTERN __program_start          ; Note 1
EXTERN UndefHandler            ; Note 2
EXTERN OverflowHandler
EXTERN BreakHandler
EXTERN AddressMatchHandler
EXTERN SingleStepHandler
EXTERN WatchdogHandler
EXTERN DBCHandler
EXTERN NMHandler

COMMON INTVEC1:NOROOT

DC32 UndefHandler
DC32 OverflowHandler
DC32 BreakHandler
DC32 AddressMatchHandler
DC32 SingleStepHandler
DC32 WatchdogHandler
DC32 DBCHandler
DC32 NMHandler
DC32 __program_start           ; Reset vector

ENDMOD
```

Listing 4-1. *app_vect.s48*: Fixed Vector Table, IAR Port

Listing 4-1, Note 1: This is externed from the BSP's *cstartup.s48* code.

Listing 4-1, Note 2: The dummy handlers are defined from a different module in the same file. These dummy handlers just execute the REIT instruction.

```

MODULE ?vectors2

EXTERN OSCtxSw
EXTERN OSTickISR

PUBLIC RelocatableVectTbl          ; Note 1

RSEG INTVEC:NOROOT

RelocatableVectTbl:
  ORG 0
  DC32 OSCtxSw                      ; Note 2: Vector 0: BRK
  DC32 OSCtxSw                      ; Vector 0: BRK
  DC32 0                            ; Vector 1: Reserved
  DC32 0                            ; Vector 2: Reserved
  DC32 0                            ; Vector 3: Reserved
  DC32 0                            ; Vector 4: Reserved
  DC32 0                            ; Vector 5: Reserved
  DC32 0                            ; Vector 6: Reserved
  DC32 0                            ; Vector 7: Reserved
  DC32 0                            ; Vector 8: DMA0
  DC32 0                            ; Vector 9: DMA1
  DC32 0                            ; Vector 10: DMA2
  DC32 0                            ; Vector 11: DMA3
  DC32 0                            ; Vector 12: Timer A0
  DC32 0                            ; Vector 13: Timer A1
  DC32 0                            ; Vector 14: Timer A2
  DC32 0                            ; Vector 15: Timer A3
  DC32 0                            ; Vector 16: Timer A4
  DC32 0                            ; Vector 17: UART0 Transmission, NACK
  DC32 0                            ; Vector 18: UART0 Reception, ACK
  DC32 0                            ; Vector 19: UART1 Transmission, NACK
  DC32 0                            ; Vector 20: UART1 Reception, ACK
  DC32 OSTickISR                    ; Note 3: Vector 21: Timer B0
  DC32 0                            ; Vector 22: Timer B1
  DC32 0                            ; Vector 23: Timer B2
  DC32 0                            ; Vector 24: Timer B3
  DC32 0                            ; Vector 25: Timer B4
  DC32 0                            ; Vector 26: INT5
  DC32 0                            ; Vector 27: INT4
  DC32 0                            ; Vector 28: INT3
  DC32 0                            ; Vector 29: INT2
  DC32 0                            ; Vector 30: INT1
  DC32 0                            ; Vector 31: INT0
  DC32 0                            ; Vector 32: Timer B5
  .
  .
  .
  .
  .
  DC32 0                            ; Vector 58: ?
  DC32 0                            ; Vector 59: ?
  DC32 0                            ; Vector 60: ?
  DC32 0                            ; Vector 61: ?
  DC32 0                            ; Vector 62: ?
  DC32 0                            ; Vector 63: ?

ENDMOD

```

Listing 4-2. *app_vect.s48*: Relocatable Vector Table, IAR Port

Listing 4-2, Note 1: This will be used in *cstartup.s34* to initialize the INTB register.

Listing 4-2, Note 2: *OSCtxSw()* is placed on IRQ vector 0.

Listing 4-2, Note 3: The BSP (Board Support Package) uses timer B0 to generate the μC/OS-II tick interrupt. Consequently, *OSTickISR()* is placed on IRQ vector 26.

4.01.02 HEW Vector Table

For the HEW project ports, the vector tables are defined in the assembly include file *linker.inc*, included as part of the BSP code. Listing 4-3 and 4-4 present the fixed and relocatable vector tables, respectively.

```
UDI:          .lword dummy_int          ; Note 1
OVER_FLOW:   .lword dummy_int
BRKI:        .lword dummy_int
ADDRESS_MATCH: .lword dummy_int
SINGLE_STEP:  .lword dummy_int
WDT:         .lword dummy_int
DBC:         .lword dummy_int
NMI:         .lword dummy_int
RESET:       .lword start              ; Note 2
```

Listing 4-3. *_linker.inc*: Fixed Vector Table, HEW Port

Listing 4-3, Note 1: The dummy handler just executes the `REIT` instruction.

Listing 4-3, Note 2: This is defined in the BSP's `_cstartup.a30` code.

4.02 Example Interrupt Service Routine

The interrupt controller for the M32C is naturally a vectored controller; consequently, a small amount of assembly-language code must be written for each interrupt vector (in either the relocatable or fixed vector tables) that will be used. This will be the same code for each interrupt vector.

```

ExampleISR:
    PUSHM      R0,R1,R2,R3,A0,A1,SB,FB      ; Note 1: Save current task's registers
    INC.B      OSIntNesting                  ; Note 2: OSIntNesting++
    CMP.B      #1,OSIntNesting              ; Note 3: if (OSIntNesting == 1) {
    JNE        OSTickISR1
    MOV.L      OSTCBCur, A0                  ;           OSTCBCur->OSTCBStkPtr = SP
    STC        ISP, [A0]                    ;           }
ExampleISR1:
    JSR        ExampleISRHandler            ; Note 4: ExampleISRHandler()
    JSR        OSIntExit                    ; Note 5: OSIntExit()
    POPM      R0,R1,R2,R3,A0,A1,SB,FB      ; Note 6: Restore current task's registers
    REIT
  
```

Listing 4-5. Example ISR

Listing 4-5, Note 1: The current task's registers are saved.

Listing 4-5, Note 2: The interrupt nesting level is incrementing.

Listing 4-5, Note 3: If a task was interrupted, then the stack pointer is saved..

Listing 4-5, Note 4: ISR performs whatever function it should. This example ISR calls a function that could be written in C, `ExampleISRHandler()`.

Listing 4-5, Note 5: `OSIntExit()` is called to determine if a task with a higher priority than the one interrupted exists. If one does, then `OSIntExit()` never returns.

Listing 4-5, Note 6: The current task's registers are restored.

Licensing

μC/OS-II is provided in source form for **FREE** evaluation, for educational use or for peaceful research. If you plan on using μC/OS-II in a commercial product you need to contact Micrium to properly license its use in your product. We provide **ALL** the source code with this application note for your convenience and to help you experience μC/OS-II. The fact that the source is provided does **NOT** mean that you can use it without paying a licensing fee. Please help us continue to provide the Embedded community with the finest software available. Your honesty is greatly appreciated.

References

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

Jean J. Labrosse
R&D Technical Books, 2002
ISBN 1-57820-103-9

Embedded Systems Building Blocks

Jean J. Labrosse
R&D Technical Books, 2000
ISBN 0-87930-604-1

Contacts

IAR Systems

Century Plaza
1065 E. Hillside Blvd
Foster City, CA 94404
USA
+1 650 287 4250
+1 650 287 4253 (FAX)
e-mail: Info@IAR.com
WEB : www.IAR.com

CMP Books, Inc.

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

Micrium

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

Renesas

450 Holger Way
San Jose, CA 95134-1368
USA
+1 408 382 7500
+1 408 382 7501 (FAX)
WEB : www.renesas.com