Micriμm

Empowering Embedded Systems

μC/OS-II

μC/OS-III

and the
Renesas RL78
Architecture

Application Note

AN-Renesas-RL78

About Micriμm

Micriμm 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 Micriμm μC/OS-II, features the highest-quality source code available for today's embedded market. Micriμm 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. Micriμm's products consistently shorten time-to-market throughout all product development cycles. For additional information on Micriμm, please visit www.micrium.com.

About μC/OS-II and μC/OS-III

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 RL78 MCU available from Renesas.

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.

μC/OS-III is the newest Kernel that Micrium provides, which is designed to save time on embedded systems projects. Besides the inherited features from μC/OS-II, μC/OS-III provides an unlimited number of tasks to be created, as well as it allows multiple tasks to run at the same priority level.

Licensing

μC/OS-II and μC/OS-III are provided in source form for **FREE** evaluation, for educational use or for peaceful research. If you plan on using μC/OS-II and/or μC/OS-III in a commercial product you need to contact Micriμm 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 and μC/OS-III. The fact that the source is provided **DOES NOT** means 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	2011/09/20	PC	Initial Version
V.1.01	2011/10/03	PC	Added **μ**C/CPU, **μ**C/OS-II, and **μ**C/OS-III Port Sections
V.2.00	2011/11/08	PC	Separate App Note for architecture and evaluation board

Software Versions

This document may or may not have been downloaded as part of an executable file containing the code and projects described here. If so, then the versions of the Micrium software modules in the table below would be 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	V2.92.01	
μC/OS-III	V3.02.00	
μC/CPU	V1.29.00	
μC/LIB	V1.36.00	

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/Sofware/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

This document, *AN-Renesas-RL78*, explains the μC/CPU, μC/OS-II and μC/OS-III ports for the Renesas RL78 architecture.

If this app note was downloaded in a packaged executable zip file, then it should have been found in the directory ***/Micrium/AppNotes/AN1xxx-RTOS/AN-Renesas-RL78*** and the code files referred to herein are described in Section 2.

2. Directories and Files

Application Notes

\Micrium\AppNotes\AN1xxx-RTOS\AN-Renesas-RL78

This directory contains this application note, AN-Renesas-RL78.pdf.

Licensing Information

\Micrium\Licensing

Licensing agreements are located in this directory. Any source code accompanying this appnote is provided for evaluation purposes only. If you choose to use μC/OS-II or μC/OS-III in a commercial product, you must contact Micriμm regarding the necessary licensing.

μC/OS-II Files

\Micrium\Software\uCOS-II\Doc

This directory contains documentation for μC/OS-II.

\Micrium\Software\uCOS-II\Ports\RL78\IAR

This directory contains the standard processor-specific files for the μC/OS-II RL78 port using the IAR toolchain. These files could easily be modified to work with other toolchains (i.e., compiler/assembler/linker/locator/debugger); however, the modified files should be placed into a different directory.

- *os_cpu.h*
- *os_cpu_a.asm*
- *os_cpu_c.c*
- *os_dbg.c*

\Micrium\Software\uCOS-II\Source

This directory contains the processor-independent source code for μC/OS-II.

- *os_core.c*
- *os_flag.c*
- *os_mbox.c*
- *os_mem.c*
- *os_mutex.c*
- *os_q.c*
- *os_sem.c*
- *os_task.c*
- *os_time.c*
- *os_tmr.c*
- *ucos_ii.h*

μC/OS-III Files

Micrium\Software\uCOS-III\Doc

This directory contains documentation for μC/OS-III.

Micrium\Software\uCOS-III\Ports\Renesas\RL78\IAR

This directory contains the standard processor-specific files for the μC/OS-III RL78 port using the IAR toolchain. These files could easily be modified to work with other toolchains (i.e., compiler/assembler/linker/locator/debugger); however, the modified files should be placed into a different directory.

- *os_cpu.h*
- *os_cpu_a.asm*
- *os_cpu_c.c*

Micrium\Software\uCOS-III\Source

This directory contains the processor-independent source code for μC/OS-III.

- *os h*
- *os_cfg_app.c*
- *os_core.c*
- *os_dbg.c*
- *os_flag.c*
- *os_int.c*
- *os_mem.c*
- *os_msg.c*
- *os_mutex.c*
- *os_pend_multi.c*
- *os_prio.c*
- *os_q.c*
- *os_sem.c*
- *os_stat.c*
- *os_task.c*
- *os_tick.c*
- *os_time.c*
- *os_tmr.c*
- *os_type.h*
- *os_var.c*

μC/CPU Files

\Micrium\Software\uC-CPU

This directory contains the CPU definition file which declares #define constants for CPU alignment, endianess, and other generic CPU properties.

- *cpu_core.c*
- *cpu_core.h*
- *cpu_def.h*

\Micrium\Software\uC-CPU\RL78\IAR

This directory defines the Micriμm portable data types for 8-, 16-, and 32-bit signed and unsigned integers. These allow the code to be independent of processor and compiler word size definitions.

- *cpu.h*

μC/LIB Files

\Micrium\Software\uC-LIB

This directory provides `#defines` for useful constants and macros.

- *lib_ascii.c*
- *lib_ascii.h*
- *lib_def.h*
- *lib_math.c*
- *lib_math.h*
- *lib_mem.c*
- *lib_mem.h*
- *lib_str.c*
- *lib_str.h*

\Micrium\Software\uC-LIB\Doc

This directory contains the documentation for **μ**C/LIB.

3. μC/CPU Port for Renesas RL78

This port holds CPU-specific definitions for the Renesas RL78 architecture, which is the same as the port for the NEC 78K0R. The μC/CPU port file is located in the following directory:

\Micrium\Software\uC-CPU\RL78\IAR

The following is the name of the μC/CPU port file.

- ***cpu.h***

3.01 *cpu.h*

In *cpu.h*, the Micriμm portable data types for 8-, 16-, and 32-bit signed and unsigned integers are defined. These allow the code to be independent of processor and compiler word size definitions. CPU addresses, standard data types, CPU stack, and the critical section method are configured in this file.

3.01.01 *cpu.h*, Data Types

All data types used on the μC/OS-II and μC/OS-III ports are referenced from the μC/CPU. This allows a coherent data type scheme between CPU- and OS-specific code sections. The complete data types list is shown in listing 3-1. Note that 64-bit data types are NOT supported by this CPU/compiler. However, in order to compile functionality/features requiring 64-bit support, 64-bit data types must be defined (even if defined with a precision lower than 64-bits (e.g. 32-bits)

Listing 3-1, *cpu.h*, Data Types

```
typedef            void        CPU_VOID;
typedef            char        CPU_CHAR;       /*  8-bit character                        */
typedef  unsigned  char        CPU_BOOLEAN;    /*  8-bit boolean or logical               */
typedef  unsigned  char        CPU_INT08U;     /*  8-bit unsigned integer                 */
typedef    signed  char        CPU_INT08S;     /*  8-bit   signed integer                 */
typedef  unsigned  short  int  CPU_INT16U;     /* 16-bit unsigned integer                 */
typedef    signed  short  int  CPU_INT16S;     /* 16-bit   signed integer                 */
typedef  unsigned  long        CPU_INT32U;     /* 32-bit unsigned integer                 */
typedef    signed  long        CPU_INT32S;     /* 32-bit   signed integer                 */
typedef  unsigned  long        CPU_INT64U;     /* 64-bit unsigned integer NOT supported   */
typedef    signed  long        CPU_INT64S;     /* 64-bit signed integer NOT supported     */

typedef            float       CPU_FP32;       /* 32-bit floating point                   */
typedef            float       CPU_FP64;       /* 64-bit floating point   NOT supported   */

typedef  volatile  CPU_INT08U  CPU_REG08;      /*  8-bit register                         */
typedef  volatile  CPU_INT16U  CPU_REG16;      /* 16-bit register                         */
typedef  volatile  CPU_INT32U  CPU_REG32;      /* 32-bit register                         */
typedef  volatile  CPU_INT64U  CPU_REG64;      /* 64-bit register NOT supported           */

typedef            void      (*CPU_FNCT_VOID)(void);                          /* 1 */
typedef            void      (*CPU_FNCT_PTR )(void *p_obj);                   /* 2 */
```

L3-1(1) The CPU_FNCT_VOID data type defined to replace the commonly-used function pointer data type of a pointer to a function which returns void and has not arguments.

L3-1(2) CPU_FNCT_PTR data type defined to replace the commonly-used function pointer data type of a pointer to a function which returns void and has a single void pointer argument.

3.01.02 *cpu.h*, Critical Sections

μC/CPU port defines two macros to enter and exit critical sections: CPU_CRITICAL_ENTER() and CPU_CRITICAL_EXIT(), respectively. There are three different methods for dealing with critical sections. For further reference, please refer to the book **MicroC/OS-II, The Real-Time Kernel**.

Listing 3-2, *cpu.h*, CPU_CRITICAL_ENTER() and CPU_CRITICAL_EXIT()

```
                                                                        /* 1 */
#define  CPU_CFG_CRITICAL_METHOD    CPU_CRITICAL_METHOD_STATUS_LOCAL

typedef  CPU_INT32U                 CPU_SR;                             /* 2 */

                                                                        /* 3 */
#if     (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_LOCAL)
#define  CPU_SR_ALLOC()             CPU_SR  cpu_sr = (CPU_SR)0
#else
#define  CPU_SR_ALLOC()
#endif

#if     (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_INT_DIS_EN)
#define  CPU_INT_DIS()          do { __disable_interrupt(); } while (0)     /* 4 */
#define  CPU_INT_EN()           do { __enable_interrupt();  } while (0)     /* 5 */
#endif

#if     (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_LOCAL)
                                                                        /* 6 */
#define  CPU_INT_DIS()          do { cpu_sr = __get_interrupt_state(); \
                                     __disable_interrupt();           } while (0)
                                                                        /* 7 */
#define  CPU_INT_EN()           do { __set_interrupt_state(cpu_sr);    } while (0)
#endif

#ifdef   CPU_CFG_INT_DIS_MEAS_EN
                                                                        /* 8 */
#define  CPU_CRITICAL_ENTER()  do { CPU_INT_DIS();         \
                                    CPU_IntDisMeasStart(); } while (0)
                                                                        /* 9 */
#define  CPU_CRITICAL_EXIT()   do { CPU_IntDisMeasStop();  \
                                    CPU_INT_EN();          } while (0)

#else

#define  CPU_CRITICAL_ENTER()  do { CPU_INT_DIS(); } while (0)                  /* 10 */
#define  CPU_CRITICAL_EXIT()   do { CPU_INT_EN();  } while (0)                  /* 11 */

#endif
```

L3-2(1) Configures CPU Critical Method.

L3-2(2) Defines CPU status register size.

L3-2(3) Allocates CPU status register word.

L3-2(4) Disable interrupts.

L3-2(5) Enable interrupts.

L3-2(6) Save CPU status word and disable interrupts.

L3-2(7) Restore CPU status word.

L3-2(8) Disable interrupts and start interrupts disabled time measurement.

L3-2(9) Stop and measure interrupts disabled time, and re-enable interrupts.

L3-2(10) Disable interrupts.

L3-2(11) Re-enable interrupts.

3.01.03 *cpu.h*, Function Prototypes

The prototypes in listing 3-3 are for CPU-specific assembly functions required from C-level.

Listing 3-3, *cpu.h* Function Prototypes

```
CPU_SR    CPU_SR_Save   (void);                                  /* 1 */
void      CPU_SR_Restore(CPU_SR  cpu_sr);                        /* 2 */
```

L3-3(1) Save CPU status word and disable interrupts.

L3-3(2) Restore CPU status word.

4. µC/OS-II Port for Renesas RL78

The µC/OS-II port for the Renesas RL78 architecture is the same as the one for the NEC 78K0R. This port assumes version 2.92.01 of µC/OS-II is being used. The µC/OS-II port files are located in the following directory:

\Micrium\Software\uCOS-II\Ports\RL78\IAR

The following are the names of the µC/OS-II port files.

- ***os_cpu.h***
- ***os_cpu_c.c***
- ***os_cpu_a.asm***
- ***os_dbg.c***

4.01 *os_cpu.h*

os_cpu.h contains processor- and implementation-specific `#defines` constants, macros, and typedefs.

4.01.01 *os_cpu.h*, macros for 'externals'

OS_CPU_GLOBALS and OS_CPU_EXT allow declaring global variables that are specific to this port. However, this port does not contain any global variables but the declarations have been included for future use.

Listing 4-1, *os_cpu.h*, Globals and Externs

```
#ifdef  OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT  extern
#endif
```

4.01.02 *os_cpu.h*, Data Types

All data types used on the µC/OS-II port are referenced from the µC/CPU port. This allows a coherent data type scheme between CPU- and OS-specific code sections.

Listing 4-2, *os_cpu.h*, Data Types

```
typedef unsigned  char        BOOLEAN;
typedef unsigned  char        INT8U;          /* Unsigned  8 bit quantity                */
typedef   signed  char        INT8S;          /* Signed    8 bit quantity                */
typedef unsigned  short  int  INT16U;         /* Unsigned 16 bit quantity                */
typedef   signed  short  int  INT16S;         /* Signed   16 bit quantity                */
typedef unsigned  long        INT32U;         /* Unsigned 32 bit quantity                */
typedef   signed  long        INT32S;         /* Signed   32 bit quantity                */
typedef            float      FP32;           /* Single precision floating point         */
typedef            double     FP64;           /* Double precision floating point         */

typedef unsigned short int  OS_STK;           /* Each stack entry is 16-bit wide         */
typedef unsigned char       OS_CPU_SR;        /* The status register (SR) is 8-bits wide */
```

4.01.03 *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. The book (**MicroC/OS-II, The Real-Time Kerne**l) describes the three different methods. The one to choose depends on the processor and compiler. μC/OS-II uses the critical section implementation from the μC/CPU port.

In most cases, the preferred method is `OS_CRITICAL_METHOD #3`, which implements `OS_ENTER_CRITICAL()` by writing a function that will save the status register of the CPU in a variable and forces a global interrupt disable. `OS_EXIT_CRITICAL()` invokes another function to restore the status register with its previous value before disabling interrupts.

Listing 4-3, *os_cpu.h*, Critical Sections

```
#define  OS_CRITICAL_METHOD    3

#if      OS_CRITICAL_METHOD == 1

                                          /* Disable interrupts             */
#define  OS_ENTER_CRITICAL()    __disable_interrupt()
                                          /* Enable interrupts              */
#define  OS_EXIT_CRITICAL()     __enable_interrupt()

#endif

#if      OS_CRITICAL_METHOD == 2
                                          /* Disable interrupts             */
#define  OS_ENTER_CRITICAL()    __asm("PUSH PSW");                  \
                                __asm("DI");

                                          /* Enable interrupts              */
#define  OS_EXIT_CRITICAL()     __asm("POP PSW");                   \
                                __asm("EI");

#endif

#if      OS_CRITICAL_METHOD == 3
                                          /* Disable interrupts             */
#define  OS_ENTER_CRITICAL()    cpu_sr = __get_interrupt_state();   \
                                __disable_interrupt()
                                          /* Enable interrupts              */
#define  OS_EXIT_CRITICAL()     __set_interrupt_state(cpu_sr)

#endif
```

4.01.04 *os_cpu.h*, Function Prototypes

The prototypes for `OSCtxSw()`, `OSIntCtxSw()`, and `OSStartHighRdy()` need to be placed in *os_cpu.h*, since these are all port specific files.

Listing 4-4, *os_cpu.h*, Function Prototypes

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

4.01.05 *os_cpu.h*, Miscellaneous

Listing 4-5 shows miscellaneous `#defines` for the RL78 architecture. This includes Task level context switch and stack growth. 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 halt the CPU execution flow; detect if any higher priority task is waiting to be executed; if so, switch to the higher priority task; and, reestablish CPU execution flow.

Listing 4-5, *os_cpu.h*, Miscellaneous

```
#define  OS_STK_GROWTH      1                                          /* 1 */
#define  OS_TASK_SW()       __break()                                  /* 2 */
```

L4-5(1) Stack grows from high to low memory on the NEC 78K0R.

L4-5(2) Issues a break instruction.

4.02 *os_cpu_c.c*

A µC/OS-II port requires eleven fairly simple C functions:

`OSInitHookBegin()`
`OSInitHookEnd()`
`OSTaskCreateHook()`
`OSTaskDelHook()`
`OSTaskIdleHook()`
`OSTaskReturnHook()`
`OSTaskStatHook()`
`OSTaskStkInit()`
`OSTaskSwHook()`
`OSTCBInitHook()`
`OSTimeTickHook()`

Typically, µC/OS-II only requires `OSTaskStkInit()`. The other functions allow the functionality of the OS to be extended with user-defined functions. The bolded functions are discussed in this section.

4.02.01 *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, the global variable (global to *os_cpu_c.c*) `OSTmrCtr()`is initialized (which is used by the *os_tmr.c* module, if OS_TMR_EN is set to 1).

Listing 4-6, *os_cpu_c.c*, `OSInitHookBegin()`

```
void  OSInitHookBegin (void)
{
#if (OS_VERSION >= 281) && (OS_TMR_EN > 0)
    OSTmrCtr = 0;
#endif
}
#endif
```

4.02.02 *os_cpu_c.c*, OSTaskCreateHook()

This function is called by μC/OS-II's `OSTaskCreate()` or `OSTaskCreateExt()` when a task is created. `OSTaskCreateHook()` gives the opportunity to add code specific to the port when a task is created. In this case, the application hook, `App_TaskCreateHook()` may be called if the `#define OS_APP_HOOKS_EN 1` is included in the *os_cfg.h* of the application. The application hooks are often used by optional modules, such as `μC/Probe` (a Real-Time Embedded Monitoring tool, see www.Micrium.com for details).

Note that if `OS_APP_HOOKS_EN` is `0`, the `ptcb` is not actually used and it is added in the body of the function (i.e. `(void)ptcb`) to avoid a compiler warning.

Listing 4-7, *os_cpu_c.c*, OSTaskCreateHook()

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

4.02.03 *os_cpu_c.c*, OSTaskStkInit()

A task is declared as shown in listing 4-8.

Listing 4-8, μC/OS-II Task

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

The code in listing 4-9 initializes the stack frame for the task being created. The task received an optional argument '`p_arg`'. The initial value of most of the CPU registers are not important, thus they are initialized to values corresponding to their register number. This makes it convenient when debugging and examining stacks in RAM. The initial values are useful when the task is first created but, the register values will change as the task code is executed. The code in listing 4-9 always returns the location of the new top-of-stack once the processor registers have been placed on the stack in the proper order.

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, or simply be a wait for the passage of time. If the task body does not perform any of these function calls, the OS will never have an opportunity to switch between different tasks.

Listing 4-9, *os_cpu_c.c*, OSTaskStkInit()

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


    opt    = opt;                           /* 'opt' is not used, prevent warning  */
    stk    = (INT16U *)ptos;                /* Load stack pointer                  */
    *stk-- = (INT16U)((INT32U)pdata >> 16);
    *stk-- = (INT16U)pdata;
    *stk-- = (INT16U)(((INT32U)task >> 16) & 0x000F) | 0x8600;                /* 1 */
    *stk-- = (INT16U)task;                  /* PC bits 0-15                        */
    *stk-- = 0x1100;                        /* RP0                                 */
    *stk-- = 0x3322;                        /* RP1                                 */
    *stk-- = 0x5544;                        /* RP2                                 */
    *stk   = 0x7766;                        /* RP3                                 */

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

L4-9(1) PC bits 16-19 in lower 8 bits, PSW in upper 16 bits.

4.02.04 *os_cpu_c.c*, OSTaskSwHook()

`OSTaskSwHook()` is called when a task switch is performed. This allows you to perform other operations during a context switch. This function allows the port code to be extended to perform measures of the execution time of a task, output a pulse on a port pin when a context switch occurs, etc. In this case, the application task switch hook, `App_TaskSwHook()`, is called. This assumes the `#define OS_APP_HOOKS_EN 1` is included in the *os_cfg.h* of the application.

Listing 4-10, *os_cpu_c.c*, OSTaskSwHook()

```
void  OSTaskSwHook (void)
{
#if OS_APP_HOOKS_EN > 0
    App_TaskSwHook();
#endif
}
```

4.02.05 *os_cpu_c.c*, `OSTimeTickHook()`

`OSTimeTickHook()` is called at the very beginning of every tick and it is assumed to be called from the tick ISR. This function allows the port code to be extended by allowing an application time tick hook, `App_TimeTickHook()`, to be called. This assumes the `#define OS_APP_HOOKS_EN 1` is included in the *os_cfg.h* of the application.

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

Listing 4-11, *os_cpu_c.c*, `OSTimeTickHook()`

```
void  OSTimeTickHook (void)
{
#if OS_APP_HOOKS_EN > 0
    App_TimeTickHook();
#endif

#if (OS_VERSION >= 281) && (OS_TMR_EN > 0)
    OSTmrCtr++;
    if (OSTmrCtr >= (OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC)) {
        OSTmrCtr = 0;
        OSTmrSignal();
    }
#endif

}
```

4.03 *os_cpu_a.asm*

A μC/OS-II port requires that you write three assembly language functions for context switch and at least one assembly language function for interrupt support (two were used in this case). These functions are needed because saving and restoring registers from C functions are not possible. These functions are:

```
OSCtxSw()
OSIntCtxSw()
OSStartHighRdy()
OSTickISR()and OSTickISR1()
```

4.03.01 *os_cpu_a.asm*, `OSCtxSw()`

A task level context switch occurs when a task is no longer able to run. For example, if the code calls `OSTimeDly(10)`, then the current task needs to be suspended until `10` clock ticks expire. Since the task can no longer run, μC/OS-II needs to find the next most important task ready to run and resume execution of that task.

`OSCtxSw()` simply consist of saving the CPU registers on stack of the task to suspend and restoring the CPU registers of the new task to resume from its stack. The pseudo-code for this is shown in listing 4-12.

Listing 4-12, `OSCtxSW()` pseudo-code

```
a) Save the current task's registers onto the current task stack
b) Save the SP into the current task's OS_TCB
c) Call OSTaskSwHook()
d) Copy OSPrioHighRdy to OSPrioCur
e) Copy OSTCBHighRdy to OSTCBCur
f) Load the SP with OSTCBHighRdy->OSTCBStkPtr
g) Restore all the registers from the high priority task stack
h) Perform a return from interrupt
```

The code to perform a 'task level' context switch is shown in listing 4-13, `OSCtxSw()`. `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 (when it calls `OSTimeDly()`, `OSSemPend()` and the semaphore is not available, etc).

Listing 4-13, *os_cpu_a.asm*, `OSCtxSw()`

```
OSCtxSw:

        PUSHALL                             ; save processor registers on the stack

                                            ; save current task's stack pointer into current
task's OS_TCB
        MOVW    RP2, OSTCBCur               ; OSTCBCur in RP2
        MOVW    RP0, SP
        MOVW    [RP2], RP0                  ; OSTCBCur->OSTCBStkPtr = SP

        CALL    OSTaskSwHook                ; call OSTaskSwHook

        MOVW    RP0, OSTCBHighRdy           ; get address of OSTCBHighRdy
        MOVW    OSTCBCur, RP0               ; OSTCBCur = OSTCBHighRdy

        MOV     R1, OSPrioHighRdy
        MOV     OSPrioCur, R1               ; OSPrioCur = OSPrioHighRdy

        MOVW    RP1, OSTCBHighRdy           ; get address of OSTCBHighRdy
        MOVW    RP0, 0x0000[RP1]            ; RP0 = OSTCBHighRdy->OSTCBStkPtr
        MOVW    SP, RP0                     ; stack pointer = RP0

        POPALL                              ; restore all processor registers from new
task's stack

        RETI                                ; return from interrupt
```

4.03.02 *os_cpu_a.asm*, `OSIntCtxSw()`

When an ISR completes, `OSIntExit()` is called to determine whether a more important task than the interrupted task needs to execute. If that is the case, `OSIntExit()` determines which task to run next and calls `OSIntCtxSw()`. At this point, the ISR would have saved the CPU registers of the interrupted task and thus, all that is left to do is restore the CPU registers of the new task.

Listing 4-14, *os_cpu_a.asm*, `OSIntCtxSw()`

```
OSIntCtxSw:
        CALL    OSTaskSwHook            ; call OSTaskSwHook

        MOVW    RP0, OSTCBHighRdy       ; get address of OSTCBHighRdy
        MOVW    OSTCBCur, RP0           ; OSTCBCur = OSTCBHighRdy

        MOV     R1, OSPrioHighRdy
        MOV     OSPrioCur, R1           ; OSPrioCur = OSPrioHighRdy

        MOVW    RP1, OSTCBHighRdy       ; get address of OSTCBHighRdy
        MOVW    RP0, 0x0000[RP1]        ; RP0 = OSTCBHighRdy->OSTCBStkPtr
        MOVW    SP, RP0                 ; stack pointer = RP0

        POPALL                          ; restore all processor registers from new task's stack

        RETI                            ; return from interrupt
```

4.03.03 *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 4-15, *os_cpu_a.asm*, OSStartHighRdy()

```
OSStartHighRdy:

        CALL    OSTaskSwHook          ; call OSTaskSwHook()
        MOVW    RP1, OSTCBHighRdy     ; address of OSTCBHighRdy in RP1
        MOVW    RP0, 0x0000[RP1]      ; RP0 = OSTCBHighRdy->OSTCBStkPtr
        MOVW    SP, RP0               ; stack pointer = RP0

        MOV     OSRunning, #0x01      ; OSRunning = True

        POPALL                        ; restore all processor registers from new task's stack

        RETI                          ; return from interrupt
```

4.03.04 *os_cpu_a.asm*, OSTickISR() and OSTickISR1()

OSTickISR()handles the tick interrupts. This ISR uses the Watchdog timer as the tick source.

Listing 4-16, *os_cpu_a.asm*, OSTickISR() and OSTickISR1()

```
OSTickISR:

        PUSHALL                             ; Save processor registers on the stack

        INC     OSIntNesting                ; increment OSIntNesting
        CMP     OSIntNesting, #1            ; if OSIntNesting != 1
        BNZ     OSTickISR1                  ; jump to OSTickISR1

                                            ; else

        MOVW    RP2, OSTCBCur               ; OSTCBCur in RP2
        MOVW    RP0, SP
        MOVW    [RP2], RP0                  ; OSTCBCur->OSTCBStkPtr = SP

OSTickISR1:

        CALL    Tmr_TickISR_Handler         ; clear timer interrupt source

        CALL    OSTimeTick                  ; call OSTimeTick()

        CALL    OSIntExit                   ; call OSIntExit()

        POPALL                              ; restore all processor registers from stack

        RETI                                ; return from interrupt
```

4.04 *os_dbg.c*

os_dbg.c is a file that has been added in μC/OS-II V2.62 to allow Kernel Aware debuggers 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. If a debugger is not used, this file can be omitted from the project build.

5. μC/OS-III Port for Renesas RL78

The μC/OS-III port for the Renesas RL78 architecture is the same as the one for the NEC 78K0R. This port assumes version 3.02.00 of μC/OS-III is being used. The μC/OS-III port files are located in the following directory:

\Micrium\Software\uCOS-III\Ports\Renesas\RL78\IAR

The following are the names of the μC/OS-III port files.

- ***os_cpu.h***
- ***os_cpu_c.c***
- ***os_cpu_a.asm***

5.01 *os_cpu.h*

os_cpu.h contains processor- and implementation-specific `#defines` constants, macros, and typedefs.

5.01.01 *os_cpu.h*, macros for 'externals'

`OS_CPU_GLOBALS` and `OS_CPU_EXT` allow declaring global variables that are specific to this port. However, this port does not contain any global variables but the declarations have been included for future use.

Listing 5-1, *os_cpu.h*, Globals and Externs

```
#ifdef  OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT  extern
#endif
```

5.01.02 *os_cpu.h*, Task Level Context Switch

Task level context switches are performed when μC/OS-III invokes the macro `OS_TASK_SW()`. Because context switching is processor specific, `OS_TASK_SW()` needs to halt the CPU execution flow; detect if any higher priority task is waiting to be executed; if so, switch to the higher priority task; and, reestablish CPU execution flow.

Listing 5-2, *os_cpu.h*, Task Level Context Switch (MWERKS)

```
#define  OS_TASK_SW()       __break()          /* Issues a break instruction           */
```

5.01.03 *os_cpu.h*, Timestamp Configuration

`OS_TS_GET()` is generally defined as `CPU_TS_Get32()` to allow CPU timestamp timer to be of any data type size. For architectures that provide 32-bit or higher precision free running counters (i.e. cycle count registers): `OS_TS_GET()` may be defined as `CPU_TS_TmrRd()` to improve performance when retrieving the timestamp, and `CPU_TS_TmrRd()`must be configured to be greater than or equal to 32-bits to avoid truncation of TS.

Listing 5-3, *os_cpu.h*, Timestamp Configuration

```
#if      OS_CFG_TS_EN == 1u
#define  OS_TS_GET()               (CPU_TS)CPU_TS_Get32()                        /* 1 */
#else
#define  OS_TS_GET()               (CPU_TS)0u
#endif

#if (CPU_CFG_TS_32_EN    == DEF_ENABLED) && \
    (CPU_CFG_TS_TMR_SIZE  < CPU_WORD_SIZE_32)
                                                                               /* 2 */
#error  "cpu_cfg.h, CPU_CFG_TS_TMR_SIZE MUST be >= CPU_WORD_SIZE_32"
#endif
```

L5-3(1) `OS_TS_GET()` may be defined as `CPU_TS_TmrRd()` to improve performance when retrieving the timestamp.

L5-3(2) `OS_TS_TmrRd()` must be configured to be greater or equal to 32-bits to avoid truncation of timestamp.

5.01.04 *os_cpu.h*, Function Prototypes

The prototypes for `OSCtxSw()`, `OSIntCtxSw()`, and `OSStartHighRdy()` need to be placed in *os_cpu.h*, since these are all port specific files.

Listing 5-4, *os_cpu.h*, Function Prototypes

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

5.02 *os_cpu_c.c*

A µC/OS-III port requires nine fairly simple C functions:

```
OSIdleTaskHook ()
OSInitHook()
OSStatTaskHook()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskReturnHook()
OSTaskStkInit()
OSTaskSwHook()
OSTimeTickHook()
```

Typically, µC/OS-III only requires `OSTaskStkInit()`. The other functions allow the functionality of the OS to be extended with user-defined functions. The bolded functions are discussed in this section.

5.02.01 *os_cpu_c.c*, `OSTaskCreateHook()`

This function is called by µC/OS-III'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 this case, the application hook, `App_TaskCreateHook()` may be called if the `#define OS_CFG_APP_HOOKS_EN 1` is included in the *os_cfg.h* of the application. The application hooks are often used by optional modules, such as µC/Probe (a Real-Time Embedded Monitoring tool, see www.Micrium.com for details).

Note that if `OS_CFG_APP_HOOKS_EN` is `0`, the `p_tcb` is not actually used and it is added in the body of the function (i.e. `(void)p_tcb`) to avoid a compiler warning.

Listing 5-5, *os_cpu_c.c*, `OSTaskCreateHook()`

```
void  OSTaskCreateHook (OS_TCB  *p_tcb)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppTaskCreateHookPtr != (OS_APP_HOOK_TCB)0) {
        (*OS_AppTaskCreateHookPtr)(p_tcb);
    }
#else
    (void)p_tcb;                                        /* Prevent compiler warning   */
#endif
}
```

5.02.02 *os_cpu_c.c*, `OSTaskStkInit()`

A task is declared as shown in listing 5-6.

Listing 5-6, μC/OS-III Task

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

The code in listing 5-7 initializes the stack frame for the task being created. The task received an optional argument '`p_arg`'. The initial value of most of the CPU registers are not important, thus they are initialized to values corresponding to their register number. This makes it convenient when debugging and examining stacks in RAM. The initial values are useful when the task is first created but, the register values will change as the task code is executed. The code in listing 5-7 always returns the location of the new top-of-stack once the processor registers have been placed on the stack in the proper order.

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, or simply be a wait for the passage of time. If the task body does not perform any of these function calls, the OS will never have an opportunity to switch between different tasks.

Listing 5-7, *os_cpu_c.c*, `OSTaskStkInit()`

```
CPU_STK  *OSTaskStkInit (OS_TASK_PTR    p_task,
                         void          *p_arg,
                         CPU_STK       *p_stk_base,
                         CPU_STK       *p_stk_limit,
                         CPU_STK_SIZE   stk_size,
                         OS_OPT         opt)
{
    CPU_STK  *stk;


    opt    = opt;                                   /* 'opt' is not used, prevent warning  */
    stk    = (CPU_STK *)(p_stk_base + stk_size);    /* Load stack pointer                  */
    *stk-- = (CPU_STK)((CPU_INT32U)p_arg >> 16);
    *stk-- = (CPU_STK)p_arg;
    *stk-- = (CPU_STK)(((CPU_INT32U)p_task >> 16) & 0x000F) | 0x8600;              /* 1 */
    *stk-- = (CPU_STK)p_task;                       /* PC bits 0-15                        */
    *stk-- = 0x1100;                                /* RP0                                 */
    *stk-- = 0x3322;                                /* RP1                                 */
    *stk-- = 0x5544;                                /* RP2                                 */
    *stk   = 0x7766;                                /* RP3                                 */

    return ((CPU_STK *)stk);
}
```

L5-7(1) PC bits 16-19 in lower 8 bits, PSW in upper 16 bits.

5.02.03 *os_cpu_c.c*, OSTaskSwHook()

OSTaskSwHook() is called when a task switch is performed. This allows you to perform other operations during a context switch. This function allows the port code to be extended to perform measures of the execution time of a task, output a pulse on a port pin when a context switch occurs, etc. In this case, the application task switch hook, App_TaskSwHook(), is called. This assumes the #define OS_CFG_APP_HOOKS_EN 1 is included in the *os_cfg.h* of the application.

Listing 5-8, *os_cpu_c.c*, OSTaskSwHook()

```
void  OSTaskSwHook (void)
{
#if OS_CFG_TASK_PROFILE_EN > 0u
    CPU_TS  ts;
#endif
#ifdef  CPU_CFG_INT_DIS_MEAS_EN
    CPU_TS  int_dis_time;
#endif



#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppTaskSwHookPtr != (OS_APP_HOOK_VOID)0) {
        (*OS_AppTaskSwHookPtr)();
    }
#endif

#if OS_CFG_TASK_PROFILE_EN > 0u
    ts = OS_TS_GET();
    if (OSTCBCurPtr != OSTCBHighRdyPtr) {
        OSTCBCurPtr->CyclesDelta  = ts - OSTCBCurPtr->CyclesStart;
        OSTCBCurPtr->CyclesTotal += (OS_CYCLES)OSTCBCurPtr->CyclesDelta;
    }

    OSTCBHighRdyPtr->CyclesStart = ts;
#endif

#ifdef  CPU_CFG_INT_DIS_MEAS_EN
    int_dis_time = CPU_IntDisMeasMaxCurReset();                             /* 1 */
    if (OSTCBCurPtr->IntDisTimeMax < int_dis_time) {
        OSTCBCurPtr->IntDisTimeMax = int_dis_time;
    }
#endif

#if OS_CFG_SCHED_LOCK_TIME_MEAS_EN > 0u
                                                                            /* 2 */
    if (OSTCBCurPtr->SchedLockTimeMax < OSSchedLockTimeMaxCur) {
        OSTCBCurPtr->SchedLockTimeMax = OSSchedLockTimeMaxCur;
    }
    OSSchedLockTimeMaxCur = (CPU_TS)0;                                      /* 3 */
#endif
}
```

L5-8(1) Keep track of per-task interrupt disable time.

L5-8(2) Keep track of per-task scheduler lock time.

L5-8(3) Reset the per-task value.

5.02.04 *os_cpu_c.c*, `OSTimeTickHook()`

`OSTimeTickHook()` is called at the very beginning of every tick and it is assumed to be called from the tick ISR. This function allows the port code to be extended by allowing an application time tick hook, `App_TimeTickHook()`, to be called. This assumes the `#define OS_CFG_APP_HOOKS_EN 1` is included in the *os_cfg.h* of the application.

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

Listing 5-9, *os_cpu_c.c*, `OSTimeTickHook()`

```
void  OSTimeTickHook (void)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppTimeTickHookPtr != (OS_APP_HOOK_VOID)0) {
        (*OS_AppTimeTickHookPtr)();
    }
#endif
}
```

5.03 *os_cpu_a.asm*

A μC/OS-III port requires that you write three assembly language functions for context switch and one assembly language function for interrupt support. These functions are needed because saving and restoring registers from C functions are not possible. These functions are:

```
OSCtxSw()
OSIntCtxSw()
OSStartHighRdy()
OS_CPU_TickHandler()
```

5.03.01 *os_cpu_a.asm*, `OSCtxSw()`

A task level context switch occurs when a task is no longer able to run. For example, if the code calls `OSTimeDly(10)`, then the current task needs to be suspended until `10` clock ticks expire. Since the task can no longer run, μC/OS-III needs to find the next most important task ready to run and resume execution of that task.

`OSCtxSw()` simply consist of saving the CPU registers on stack of the task to suspend and restoring the CPU registers of the new task to resume from its stack. The pseudo-code for this is shown in listing 5-10.

Listing 5-10, `OSCtxSW()` pseudo-code

```
Save the current task's context onto the current task's stack,
OSTCBCurPtr->StkPtr = SP;
OSTaskSwHook();
OSPrioCur           = OSPrioHighRdy;
OSTCBCurPtr         = OSTCBHighRdyPtr;
SP                  = OSTCBHighRdyPtr->StkPtr;
Restore the new task's context from the new task's stack,
Return to new task's code.
```

The code to perform a 'task level' context switch is shown in listing 5-11, `OSCtxSw()`. `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 (when it calls `OSTimeDly()`, `OSSemPend()` and the semaphore is not available, etc).

Listing 5-11, *os_cpu_a.asm*, `OSCtxSw()`

```
OSCtxSw:

        PUSHALL                       ; save processor registers on the stack

                                                                              /* 1 */

        MOVW    RP2, OSTCBCurPtr      ; OSTCBCurPtr in RP2
        MOVW    RP0, SP
        MOVW    [RP2], RP0            ; OSTCBCurPtr->OSTCBStkPtr = SP

        CALL    OSTaskSwHook          ; call OSTaskSwHook

        MOVW    RP0, OSTCBHighRdyPtr  ; get address of OSTCBHighRdyPtr
        MOVW    OSTCBCurPtr, RP0      ; OSTCBCurPtr = OSTCBHighRdyPtr

        MOV     R1, OSPrioHighRdy
        MOV     OSPrioCur, R1        ; OSPrioCur = OSPrioHighRdy

        MOVW    RP1, OSTCBHighRdyPtr ; get address of OSTCBHighRdyPtr
        MOVW    RP0, 0x0000[RP1]     ; RP0 = OSTCBHighRdyPtr->OSTCBStkPtr
        MOVW    SP, RP0              ; stack pointer = RP0

        POPALL                       ; restore all processor registers from new task's stack

        RETI                         ; return from interrupt
```

L5-11(1) Save current task's stack pointer into current task's OS_TCB.

5.03.02 *os_cpu_a.asm*, `OSIntCtxSw()`

When an ISR completes, `OSIntExit()` is called to determine whether a more important task than the interrupted task needs to execute. If that is the case, `OSIntExit()` determines which task to run next and calls `OSIntCtxSw()`. At this point, the ISR would have saved the CPU registers of the interrupted task and thus, all that is left to do is restore the CPU registers of the new task.

`OSIntCtxSw()` is called when an ISR makes a higher priority task ready-to-run.

Listing 5-12, *os_cpu_a.asm*, `OSIntCtxSw()`

```
OSIntCtxSw:
        CALL    OSTaskSwHook         ; call OSTaskSwHook

        MOVW    RP0, OSTCBHighRdyPtr ; get address of OSTCBHighRdyPtr
        MOVW    OSTCBCurPtr, RP0     ; OSTCBCurPtr = OSTCBHighRdyPtr

        MOV     R1, OSPrioHighRdy
        MOV     OSPrioCur, R1        ; OSPrioCur = OSPrioHighRdy

        MOVW    RP1, OSTCBHighRdyPtr ; get address of OSTCBHighRdyPtr
        MOVW    RP0, 0x0000[RP1]     ; RP0 = OSTCBHighRdyPtr->OSTCBStkPtr
        MOVW    SP, RP0              ; stack pointer = RP0

        POPALL                       ; restore all processor registers from new task's stack

        RETI                         ; return from interrupt
```

5.03.03 *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 5-13, *os_cpu_a.asm*, OSStartHighRdy()

```
OSStartHighRdy:

        CALL    OSTaskSwHook          ; call OSTaskSwHook()
        MOVW    RP1, OSTCBHighRdyPtr  ; address of OSTCBHighRdyPtr in RP1
        MOVW    RP0, 0x0000[RP1]      ; RP0 = OSTCBHighRdyPtr->OSTCBStkPtr
        MOVW    SP, RP0               ; stack pointer = RP0

        POPALL                        ; restore all processor registers from new task's stack

        RETI                          ; return from interrupt
```

5.03.04 *os_cpu_a.asm*, OSTickISR() and OSTickISR1()

OSTickISR()handles the tick interrupts. This ISR uses the Watchdog timer as the tick source.

Listing 5-14, *os_cpu_a.asm*, OSTickISR() and OSTickISR1()

```
OSTickISR:

        PUSHALL                             ; Save processor registers on the stack

        INC     OSIntNestingCtr             ; increment OSIntNestingCtr
        CMP     OSIntNestingCtr, #1         ; if OSIntNestingCtr != 1
        BNZ     OSTickISR1                  ; jump to OSTickISR1

                                            ; else

        MOVW    RP2, OSTCBCurPtr            ; OSTCBCurPtr in RP2
        MOVW    RP0, SP
        MOVW    [RP2], RP0                  ; OSTCBCurPtr->OSTCBStkPtr = SP

OSTickISR1:

        CALL    Tmr_TickISR_Handler         ; clear timer interrupt source

        CALL    OSTimeTick                  ; call OSTimeTick()

        CALL    OSIntExit                   ; call OSIntExit()

        POPALL                              ; restore all processor registers from stack

        RETI                                ; return from interrupt
```

Licensing

μC/OS-II and **μ**C/OS-III are provided in source form for **FREE** evaluation, for educational use or for peaceful research. If you plan on using **μ**C/OS-II and/or **μ**C/OS-III in a commercial product you need to contact Micriμm 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 **μ**C/OS-II and **μ**C/OS-III. 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

μC/OS-III, User's Manual
Micriμm
Micriμm Press, 2010
ISBN 978-0-9823375-9-2

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

Contacts

Micriμm
1290 Weston Road, Suite 306
Weston, FL 33326
USA

+1 954 217 2036
+1 954 217 2037 (FAX)

e-mail: sales@Micrium.com
WEB : www.Micrium.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