# Micriμm



# μC/OS-II

and the

## AVR32 UC3 Architecture

**Application Note**

AN-1030

www.Micrium.com

## 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

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 AVR32 UC3 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 an 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 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**. 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 any error is found in this document, please inform Micriμm in order for the appropriate corrections to be present in future releases.

| Version | Date | By | Description |
|---|---|---|---|
| V 1.00 | 2007/05/08 | FGK | Initial version. |
| V 1.01 | 2008/07/14 | FGK | Update uC/OS-II version<br>Update BSP |

## Software Versions

This document may or may not have been downloaded as part of an executable file containing the code described herein or any additional application or board support code. If so, then the versions of the Micriμm 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 | V2.86 | |

# Table of Contents

# 1.00 Introduction

This document describes the official Micriμm port for **μC/OS-II** to the AVR32 UC3 architecture. Figure 1-1 shows a block diagram with 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. The AVR32 UC3 has been ported on both the IAR and AVR32Studio tools.

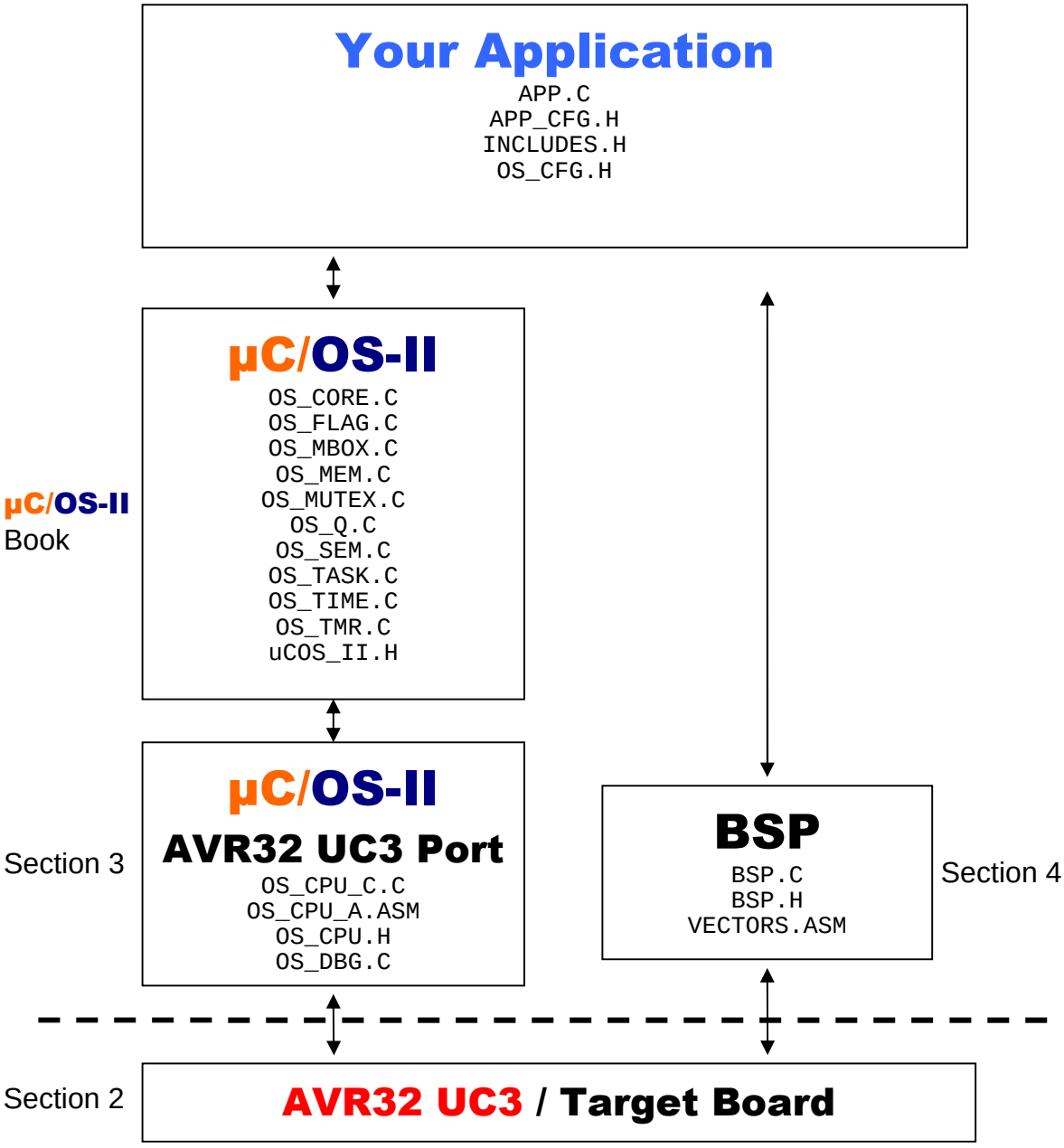



**Figure 1-1, Relationship between modules.**

# 1.01 The AVR32 architecture

The AVR family was launched by Atmel® in 1996 and has had remarkable success in the 8-and 16-bit flash microcontroller market. In 2007, the AVR®32 was introduced to complement the current AVR microcontrollers. The AVR®32 is a new high-performance 32-bit RISC microprocessor core, designed for cost-sensitive embedded applications, with particular emphasis on low power consumption and high code density. Through the AVR32 family, the AVR is extended into a new range of higher performance applications that is currently served by 32- and 64-bit processors.

The instruction set architecture in the AVR32 has been tuned to allow for a variety of microarchitectures, enabling the AVR32 to be implemented as low-, mid- or high-performance processors. The new AVR32 architecture is not binary compatible with earlier AVR architectures. The instruction format has both compact instructions with 16 bits length and extended 32-bit instructions in order to achieve high code density. While the instruction length is only 16 bits for most instructions, powerful 32-bit instructions are implemented to further increase performance. Compact and extended instructions can be freely mixed in the instruction stream.

Memory load and store operations are provided for byte, half-word, word and double word data with automatic sign- or zero extension of half-word and byte data. The C-compiler is closely linked to the architecture and is able to exploit code optimization features, both for size and speed. In addition, the processor supports byte and half-word data types without penalty in code size and performance

The register file in the AVR32 is organized as 16 32-bit registers and includes the Program Counter (PC), the Link Register (LR), and the Stack Pointer (SP). In addition, one register (R12) is designed to hold return values from function calls and is used implicitly by some instructions.

The AVR32 architecture defines different microarchitectures. This enables implementations that are tailored to specific needs and applications. The microarchitectures provide different performance levels at the expense of area and power consumption. The AVR32 UC3 implements the AVR32A microarchitecture.

The AVR32A microarchitecture does not provide dedicated hardware registers for shadowing of register file registers in interrupt contexts. Additionally, it does not provide hardware registers for the return address registers and return status registers. Instead, all this information is stored on the system stack.

Upon interrupt initiation, registers R8-R12, and LR are automatically pushed to the system stack. These registers are pushed regardless of the priority level of the pending interrupt. The return address and status register (SR) are also automatically pushed to stack. The interrupt handler can therefore use R8-R12, and LR freely. Upon interrupt completion, the old R8-R12 registers, LR and SR are restored, and execution continues at the return address stored popped from stack.

The stack is also used to store the status register and return address for exceptions and system calls. Executing the `rete` or `rets` instruction at the completion of an exception or system call will pop this status register and continue execution at the popped return address.

Documentation on the AVR32 UC3 architecture can be found on the ATMEL website: www.atmel.com.

Figure 1-2 shows the Unit Programmer's Model and basically, the registers that need to be saved and restored during a context switch. Note this **μC/OS-II** port assumes that tasks are executing in Supervisor Mode and thus, the stack pointer is the SP_SYS (System Stack Pointer). The AVR32A microarchitecture defines many more registers but these are the ones that are mostly concerned with for this **μC/OS-II** port.

**Register File in AVR32A**

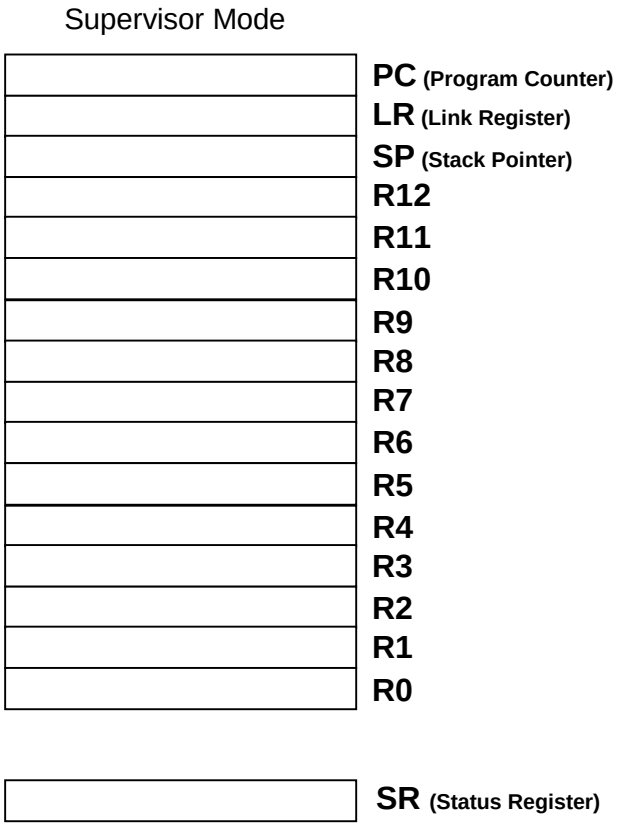



**Figure 1-2, Integer Unit Programmer's Model**

# 2.00 μC/OS-II Port for AVR32 UC3

The port for the AVR32 UC3 processors assumes that a μC/OS-II V2.86 or higher is being used.

## 2.01 Directories and Files

The software described in this section of this application note is assumed to be placed in the following directories:

```
\Micrium\Software\uCOS-II\Ports\AVR32\UC3\GNU

\Micrium\Software\uCOS-II\Ports\AVR32\UC3\IAR
```

Like all μC/OS-II ports, the source code for the port is found in the following files:

| | |
|---|---|
| `OS_CPU.H` | Section 2.02 |
| `OS_CPU_C.C` | Section 2.03 |
| `OS_CPU_A.ASM` | Section 2.04 |
| `OS_DBG.C` | Section 2.05 |

Test code and configuration files are found in their appropriate directories and are described later.

## 2.02 OS_CPU.H

OS_CPU.H contains processor- and implementation-specific #defines constants, macros, and typedefs.

## 2.02.01 OS_CPU.H, macros for 'externals'

OS_CPU_GLOBALS and OS_CPU_EXT allow 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 for future use.

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

**Listing 2-1, OS_CPU.H, Globals and Externs**

## 2.02.02 OS_CPU.H, Data Types

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

```
typedef  CPU_BOOLEAN    BOOLEAN;
typedef  CPU_INT08U     INT8U;
typedef  CPU_INT08S     INT8S;
typedef  CPU_INT16U     INT16U;
typedef  CPU_INT16S     INT16S;
typedef  CPU_INT32U     INT32U;
typedef  CPU_INT32S     INT32S;
typedef  CPU_FP32       FP32;
typedef  CPU_FP64       FP64;

typedef  CPU_STK        OS_STK;
typedef  CPU_SR         OS_CPU_SR;
```

**Listing 2-2, OS_CPU.H, Data Types**

## 2.02.03 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. 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 and forces a global interrupt disable. `OS_EXIT_CRITICAL()` invokes another function to restore the status register with its previous value before disabling interrupts. **μC/OS-II** uses the critical section implementation from the **μC/CPU** port, which are described later.

```
#define  OS_CRITICAL_METHOD     CPU_CRITICAL_METHOD

#define  OS_ENTER_CRITICAL()   (CPU_CRITICAL_ENTER())
#define  OS_EXIT_CRITICAL()    (CPU_CRITICAL_EXIT())
```

**Listing 2-3, OS_CPU.H, `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`**

Note that if the application code uses these macros, and `OS_CRITICAL_METHOD` is #3, a local variable called `cpu_sr` **MUST** be declared and initialized to `0`, as shown in listing 2-4:

```
#if OS_CRITICAL_METHOD == 3                 /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr = 0;
#endif
```

**Listing 2-4, Local storage for Critical Method #3**

## 2.02.04 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 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. In this case, a supervisor call (SCALL) instruction is used to create an exception. The benefit of using a SCALL is that it does not alter the CPU registers when OS_TASK_SW() is called. Current program counter (PC) and status register (SR) are pushed into the stack, the execution mode is then set to supervisor mode, and the execution flow is redirect to the exception vector table at the offset EVBA+0x100. For this vector address, declared in VECTORS.ASM (described later), a branch to _handle_Supervisor_Call redirects the execution flow to OSCtxSw(), which is declared in OS_CPU_A.ASM. More information on exception handling is available in the section Handling Exceptions and Interrupts.

```
#define  OS_TASK_SW()          __asm__ __volatile__ ("scall")
```

**Listing 2-5a, OS_CPU.H, Task Level Context Switch (GNU)**

```
#define  OS_TASK_SW()          (OSCtxSw())
```

**Listing 2-5b, OS_CPU.H, Task Level Context Switch (IAR)**

## 2.02.05 OS_CPU.H, Stack growth

The stack on the AVR32 grows from high memory to low memory and thus, OS_STK_GROWTH is set to 1 to indicate this to μC/OS-II.

```
#define  OS_STK_GROWTH        1
```

**Listing 2-6, OS_CPU.H, Stack Growth**

## 2.02.06 OS_CPU.H, Function Prototypes

The prototype in listing 2-7 is for the context switch function. For AVR32Studio, OS_TASK_SW() is defined directly by a SCALL instruction. In IAR, OS_TASK_SW() calls OSCtxSw(), a supervisor call routine. The function prototype on IAR has a preceding __*scall* extended keyword to inform the compiler that this function is a supervisor call. The compiler automatically introduces a SCALL instruction in place of the function call. The use of the extended keyword helps the compiler to be aware of the SCALL instruction during optimization. Otherwise, simply including the instruction through an inline assembly would make the compiler ignore the presence of this instruction and not optimize the code as expected.

```
#define  OS_TASK_SW()          __asm__ __volatile__ ("scall")
```

**Listing 2-7a, OS_CPU.H, Function Prototypes (GNU)**

```
#define  OS_TASK_SW()          (OSCtxSw())

__scall  extern  void  OSCtxSw(void);
```

**Listing 2-7b, OS_CPU.H, Function Prototypes (IAR)**

As of V2.77, the prototypes for OSCtxSw(), OSIntCtxSw(), and OSStartHighRdy() need to be placed in OS_CPU.H, since these are all port specific files. OSCtxRestore() is used restore the context stored on the stack pointed by sp. OSIntCtxRestore() is used restore the context stored on the stack pointed by sp, when executing from an interrupt context. OSCtxSw(), OSCtxRestore(), and OSIntCtxRestore() are declared in OS_CPU_A.ASM.

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

void    OSCtxRestore(INT32U *sp);
void    OSIntCtxRestore(INT32U *sp);
```

**Listing 2-8, OS_CPU.H, Function Prototypes**

## 2.03 OS_CPU_C.C

A **μC/OS-II** port requires ten 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 the functionality of the OS to be extended with user-defined functions. The highlighted functions are discussed in this section.

**IMPORTANT**

In order for the compiler to use the functions declared in this file, the #define constant `OS_CPU_HOOKS_EN` needs to be set to `1` in `OS_CFG.H`.

## 2.03.01 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, 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).

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

**Listing 2-9, OS_CPU_C.C, `OSInitHookEnd()`**

## 2.03.02 OS_CPU_C.C, OSTaskCreateHook()

This function is called by **μC/OS-II**'s `OSTaskCreate()` or `OSTaskCreateExt()` when a task is created. `OSTaskCreateHook()` gives the opportunity to add code specific to the port when a task is created. In 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) and **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 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.

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

**Listing 2-10, OS_CPU_C.C, `OSTaskCreateHook()`**

## 2.03.03 OS_CPU_C.C, OSTaskStkInit()

A task is declared as shown in listing 2-11.

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

**Listing 2-11, μC/OS-II Task**

The code in listing 2-12 initializes the stack frame for the task being created. The task received an optional argument '`p_arg`'. For the IAR AVR32 compiler, '`p_arg`' is passed in `R12` when the task is created. 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 '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.

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


    (void)opt;                                  /* 'opt' is not used, prevent warning   */

     stk   =          ptos;
    *stk-- = (INT32U) 0x08080808;               /* R8                                   */
    *stk-- = (INT32U) 0x09090909;               /* R9                                   */
    *stk-- = (INT32U) 0x10101010;               /* R10                                  */
    *stk-- = (INT32U) 0x11111111;               /* R11                                  */
    *stk-- = (INT32U) p_arg;                    /* R12                                  */
    *stk-- = (INT32U) 0xXXXXXXXX;               /* R14/LR (Note 1)                      */
    *stk-- = (INT32U) task;                     /* R15/PC                               */
    *stk-- = (INT32U) OS_CPU_SR_M0_MASK;        /* SR: Supervisor mode                  */
    *stk-- = (INT32U) 0xFF0000FF;               /* R0                                   */
    *stk-- = (INT32U) 0x01010101;               /* R1                                   */
    *stk-- = (INT32U) 0x02020202;               /* R2                                   */
    *stk-- = (INT32U) 0x03030303;               /* R3                                   */
    *stk-- = (INT32U) 0x04040404;               /* R4                                   */
    *stk-- = (INT32U) 0x05050505;               /* R5                                   */
    *stk-- = (INT32U) 0x06060606;               /* R6                                   */
    *stk   = (INT32U) 0xXXXXXXXX;               /* R7     (Note 2)                      */

    return (stk);
 }
```

**Listing 2-12, OS_CPU_C.C, `OSTaskStkInit()`**

L2-12(1) On GNU compiler, `R14/LR` is loaded with `0x80000000`. On IAR, it is loaded with `0x14141414`.

L2-12(2) On GNU compiler, `R7` is loaded with `0x00000000`. On IAR, it is loaded with `0x07070707`.

Figure 2-1 shows the stack frame initialization previously of each task's creation.

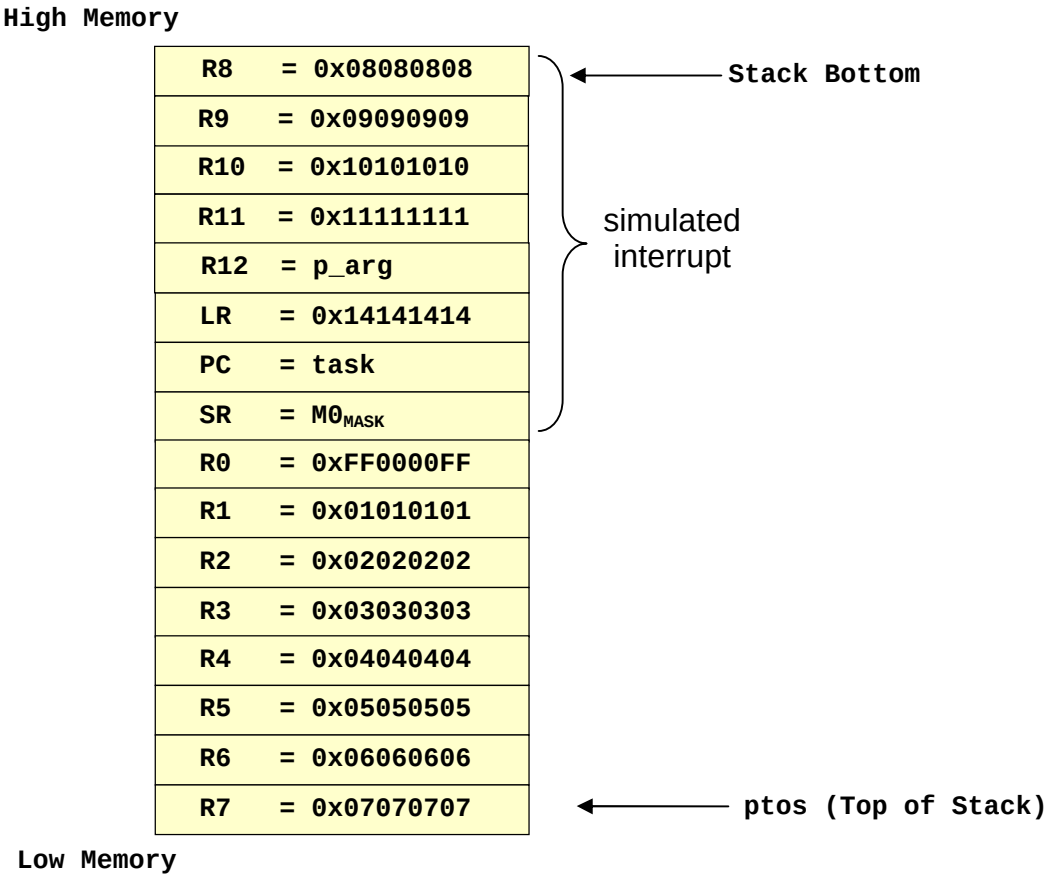



**Figure 2-1, The Initial Stack Frame for each Task.**

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

Note that the bottom of the stack frame has a simulated interrupt stack. This layout reduces latency when task switching under an interrupt call. In the context switch under an interrupt context, `R0-R7` can be restored in a single instruction call, and the return from event handler (`RETE`) instruction takes care of restoring the rest of the registers and performs the context switch.

## 2.03.04 OS_CPU_C.C, OSTaskSwHook()

`OSTaskSwHook()` is called when a context switch occurs. 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.

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

**Listing 2-13, OS_CPU_C.C, `OSTaskSwHook()`**

## 2.03.05 OS_CPU_C.C, OSTimeTickHook()

`OSTimeTickHook()` is called at the very beginning of `OSTimeTick()`. This function allows the port code to be extended by allowing a 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.

```
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
}
```

**Listing 2-14, OS_CPU_C.C, `OSTimeTickHook()`**

## 2.03.06 OS_CPU_C.C, 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.

```
void  OSIntCtxSw (void)
{
    OSTaskSwHook();                                 /* (1)                                  */

    OSTCBCur  =  OSTCBHighRdy;
    OSPrioCur =  OSPrioHighRdy;

    OSIntCtxRestore(OSTCBHighRdy->OSTCBStkPtr);     /* (2)                                  */
}
```

**Listing 2-15, OS_CPU_C.C, `OSIntCtxSw()`**

L2-15(1) Call user defined context switch hook, `OSTaskSwHook.`

L2-15(2) `OSIntCtxRestore()` is called to restore the stack pointer with the `OS_TCB` of the highest priority task (pointed to by `OSTCBHighRdy`).

## 2.03.07 OS_CPU_C.C, 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.

```
void  OSStartHighRdy (void)
{
    OSTaskSwHook();                             /* (1)                                  */
    OSRunning =  1;                             /* (2)                                  */
    OSCtxRestore(OSTCBHighRdy->OSTCBStkPtr);    /* (3)                                  */
}
```

**Listing 2-16, OS_CPU_C.C, `OSStartHighRdy()`**

L2-16(1) Call `OSTaskSwHook()`. The hook knows that a full context switch is NOT occurring by examining the value of `OSRunning` which is set to `OS_FALSE`.

L2-16(2) Once `OSTaskSwHook()` is returned, `OSRunning` is set to `OS_TRUE` which indicates that now the RTOS will be running.

L2-16(3) `OSCtxRestore()` is called to restore the stack pointer with the `OS_TCB` of the highest priority task (pointed to by `OSTCBHighRdy`).

## 2.04 OS_CPU_A.ASM

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

```
OSCtxSw()
OSCtxRestore()
OSIntCtxRestore()
OSIntX()
OSIntISRHandler()
```

## 2.04.01 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. The flow of code is as follows:

The task calls `OSTimeDly(10);`
- `OSTimeDly()` calls `OS_Sched();`
  - `OS_Sched()` disables interrupts;
  - `OS_Sched()` finds the highest priority task that is ready to run;
  - `OS_Sched()` calls invokes the macro `OS_TASK_SW()` which performs a `SCALL`;
    - `SCALL` vectors to `_handle_Supervisor_Call`;
    - `_handle_Supervisor_Call` branches to `OSCtxSw()`;
    - `OSCtxSw()` performs the context switch;
    - Execution resumes in the new 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 2-17.

```
void  OSCtxSw (void)
{
    Save processor registers;

    OSTCBCur->OSTCBStkPtr =  SP;

    OSTaskSwHook();

    OSTCBCur              =  OSTCBHighRdy;
    OSPrioCur             =  OSPrioHighRdy;

    SP                    =  OSTCBHighRdy->OSTCBStkPtr;

    Restore processor registers;
    Return from supervisor call;
}
```

**Listing 2-17, `OSCtxSw()` pseudo-code**

The actual code to perform a 'task level' context switch is shown in listing 2-18, `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.). Figure 2-2 also shows the process graphically.

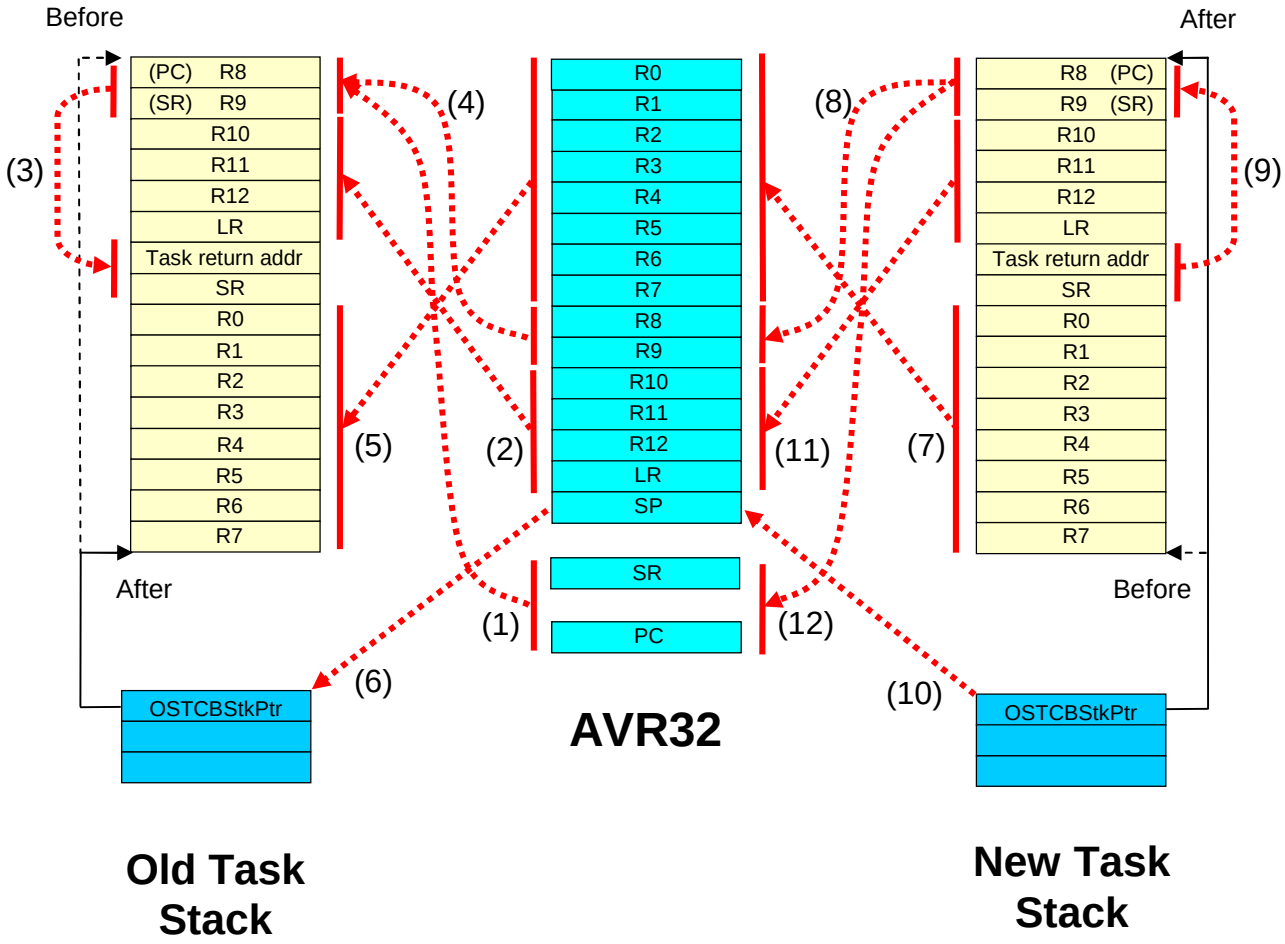



**Figure 2-2, `OSCtxSw()`, Task Level Context Switch.**

```
OSCtxSw:                                ; (1) SCALL
    PUSHM   R10-R12, LR                 ; (2) Save R10-R12, LR into stack
    LD.D    R10, SP[4*4]                ;     Copy SR and PC from bottom of stack
    PUSHM   R10-R11                     ; (3) Save SR and PC into stack
    ST.D    SP[6*4], R8                 ; (4) Save R8 and R9 into stack
    PUSHM   R0-R7                       ; (5) Save R0-R7 into stack

    MOV     R8, LWRD OSTCBCur
    ORH     R8, HWRD OSTCBCur           ;     OSTCBCur
    LD.W    R9, R8[0]                   ;    *OSTCBCur
    ST.W    R9[0], SP                   ; (6) OSTCBCur->OSTCBStkPtr = SP

    RCALL   OSTaskSwHook

    MOV     R12, LWRD OSTCBHighRdy
    ORH     R12, HWRD OSTCBHighRdy      ;     OSTCBHighRdy
    LD.W    R10, R12[0]                 ;    *OSTCBHighRdy
    MOV     R8, LWRD OSTCBCur
    ORH     R8, HWRD OSTCBCur           ;     OSTCBCur
    ST.W    R8[0], R10                  ;     OSTCBCur = OSTCBHighRdy

    MOV     R12, LWRD OSPrioHighRdy
    ORH     R12, HWRD OSPrioHighRdy
    LD.UB   R11, R12[0]                 ;     *OSPrioHighRdy
    MOV     R12, LWRD OSPrioCur
    ORH     R12, HWRD OSPrioCur
    ST.B    R12[0], R11                 ;      OSPrioCur = OSPrioHighRdy

    LD.W    R12, R10[0]                 ;      Retrieve OSTCBHighRdy->OSTCBStkPtr

    LDM     R12, R0-R7                  ; (7)  Restore R0-R7
    LD.D    R8,  R12[14*4]              ; (8)  Restore R8-R9
    LD.D    R10, R12[ 8*4]              ;      Retrieve PC and SR from stack frame
    ST.D    R12[14*4], R10              ; (9)  Store PC and SR to bottom of stack
    SUB     R12, -10*4                  ;      Move Stack Pointer Reference to LR
    SUB     SP, R12, -4*4               ; (10) Restore Stack Pointer
    LDM     R12, R10-R12, LR            ; (11) Restore R10-R12, LR

    RETS                                ; (12) Restore PC and SR {restore task}
```

**Listing 2-18, OS_CPU_A.ASM, `OSCtxSw()`**

## 2.04.02 OS_CPU_A.ASM, OSCtxRestore()

`OSCtxRestore()` restores the CPU registers from the stack pointer passed as its parameter. Since it also restores the program counter (PC), a task switch is performed, returning the execution to the interrupted task. This function is called by `OSStartHighRdy()` to start the highest priority task.

The prototype of this function is defined as follows:

```
        void  OSCtxRestore(INT32U *sp);
```

The code for `OSCtxRestore()` is shown in listing 2-19. Figure 2-3 also shows the process graphically.

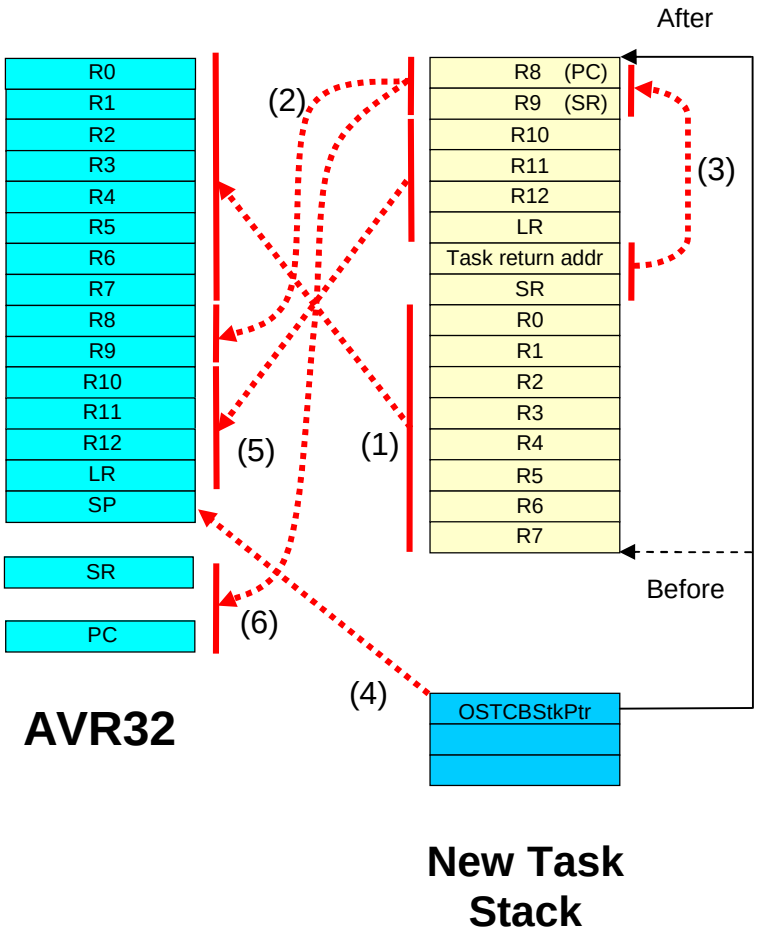



**Figure 2-3, `OSCtxRestore()`, Task Level Context Restore.**

```
OSCtxRestore:
    LDM     R12, R0-R7              ; (1) Restore R0-R7
    LD.D    R8,  R12[14*4]          ; (2) Restore R8-R9
    LD.D    R10, R12[ 8*4]          ;     Retrieve PC and SR from stack frame
    ST.D    R12[14*4], R10          ; (3) Store PC and SR at bottom of stack frame
    SUB     R12, -10*4              ;     Move Stack Pointer Reference to LR
    SUB     SP, R12, -4*4           ; (4) Restore Stack Pointer
    LDM     R12, R10-R12, LR        ; (5) Restore R10-R12, LR
    RETS                            ; (6) Restore PC and SR {restore task}
```

**Listing 2-19, OS_CPU_A.ASM, `OSCtxRestore()`**

**IMPORTANT**

The assembly code in this function assumes that the first parameter in a function call is passed on R12. If the calling convention from a different compiler does not follow this standard, this function will not work.

## 2.04.03 OS_CPU_A.ASM, OSIntCtxRestore()

`OSIntCtxRestore()` restores the CPU registers from the stack pointer passed as its parameter under an interrupt context. Since it also restores the program counter (PC), a task switch is performed, returning the execution to the interrupted task. This function is called by `OSIntCtxSw()`to switch to the highest priority task.

The prototype of this function is defined as follows:

```
void  OSIntCtxRestore(INT32U *sp);
```

The code for `OSIntCtxRestore()` is shown in listing 2-20. Figure 2-4 also shows the process graphically.

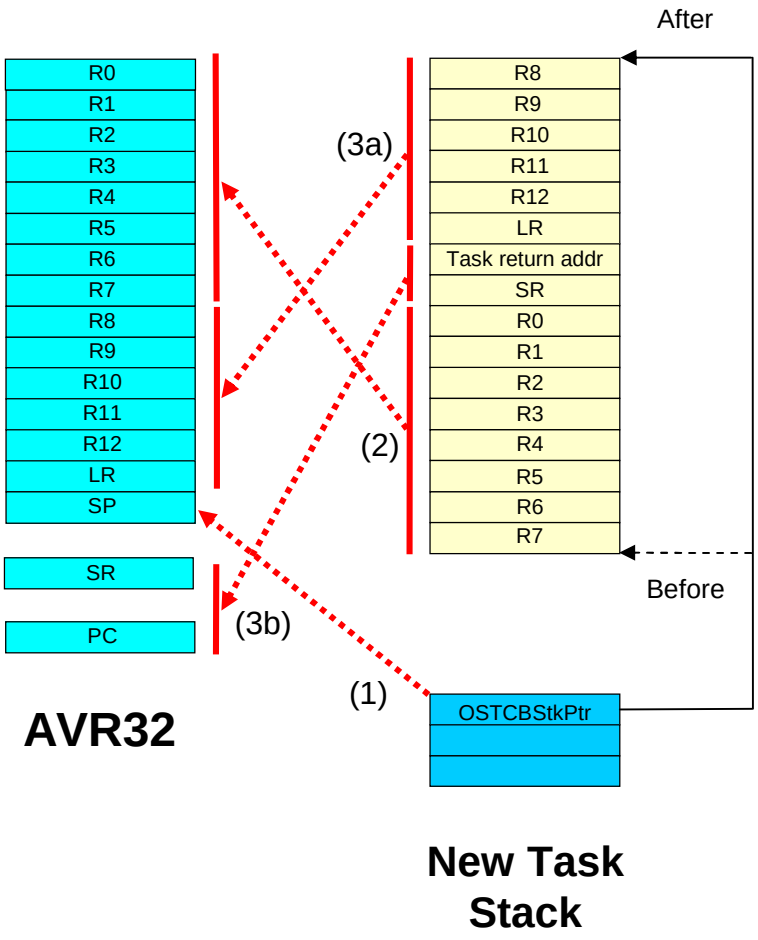



**Figure 2-4, `OSIntCtxRestore()`, Task Level Context Restore under Interrupt Context.**

```
OSIntCtxRestore:
    MOV     SP, R12             ; (1) Restore SP (Stack Pointer)
    POPM    R0-R7               ; (2) Restore R0-R7
    RETE                        ; (3) Restore R8-R12, LR, PC and SR {restore task}
```

**Listing 2-20, OS_CPU_A.ASM, `OSIntCtxRestore()`**

**IMPORTANT**

The assembly code in this function assumes that the first parameter in a function call is passed on R12. If the calling convention from a different compiler does not follow this standard, this function will not work.

## 2.05 Handling Exceptions and Interrupts

The AVR32 UC3 has a exception handling scheme that allows the exception vector table to be placed at any addressable memory section. This address which contains the base address of the exception vector table is stored in the EVBA (Exception Vector Base Address) system register. Each exception has a EVBA-relative offset address. The target address of the event handler is calculated as (EVBA | event handler offset). The same mechanism is used to service interrupt requests. The EVBA is initialized to `0x80004000` in the linker command file for IAR (`$PROJ_DIR$\..\..\BSP\lnkuc3xXXXX.xcl`), and on AVR32Studio, EVBA is initialized at the section `.gcc_except_table` aligned on a `0x1000` address boundary.

## 2.05.01 VECTORS.ASM, Exception Handlers

VECTORS.ASM initializes the exception handling mechanism and holds the exception handlers for the AVR32. A runtime library **MUST** be specified in order to have the EVBA (Exception Vector Base Address) initialized at startup. For this reason, the address where the exception vector table is defined needs to be available for the initialization function. Under IAR, the runtime library configuration is available in the library configuration section of the project general options. Under AVR32Studio, the runtime library is included in the project by default.

The DLIB runtime library on IAR has its own initialization function defined as `??init_EVBA`. It uses the variable `??EVBA` to initialize the EVBA in the system register. The GNU runtime library on AVR32Studio uses the variable `_evba` to initialize the EVBA in the system register. Listing 2-21 shows how to utilize the runtime library function to initialize the EVBA at program start.

```
    COMMON  EVTAB:CODE:NOROOT                          ; (1)

    EXTERN  ??init_EVBA                                ; (2)
    REQUIRE ??init_EVBA                                ; (3)

    PUBLIC  ??EVBA                                     ; (4)
    PUBLIC  _evba                                      ; (5)
```

**Listing 2-21, VECTORS.ASM, EVBA Initialization**

L2-21(1) Start the exception vector table in a common segment at EVTAB. EVTAB is initialized in the linker command file at `0x80004000`.

L2-21(2) Import `??init_EVBA` external reference into `VECTORS.ASM` current module.

L2-21(3) Forces `??init_EVBA` to be referenced by current module.

L2-21(4) Exports ??EVBA to be used by the runtime library.

L2-21(5) Alias to ??EVBA used by the `OSIntPrioReg` to define the address offsets of the interrupt handlers.

Each exception vector has a predefined address offset from the EVBA. The exception vectors are defined as branches to their respective exception handlers as shown in listing 2-22.

```
??EVBA:                                                 ; (1)
_evba:                                                  ; (2)
    ORG     0x000                                       ; Unrecoverable Exception
    BR      _handle_Unrecoverable_Exception

    ORG     0x004                                       ; TLB Multiple Hit: UNUSED IN AVR32A
    BR      _handle_TLB_Multiple_Hit

    ORG     0x008                                       ; Bus Error Data Fetch
    BR      _handle_Bus_Error_Data_Fetch

    ORG     0x00C                                       ; Bus Error Instruction Fetch
    BR      _handle_Bus_Error_Instruction_Fetch

    ORG     0x010                                       ; NMI
    BR      _handle_NMI

    ORG     0x014                                       ; Instruction Address
    BR      _handle_Instruction_Address

    ORG     0x018                                       ; ITLB Protection
    BR      _handle_ITLB_Protection

    ORG     0x01C                                       ; Breakpoint
    BR      _handle_Breakpoint

    ORG     0x020                                       ; Illegal Opcode
    BR      _handle_Illegal_Opcode

    ORG     0x024                                       ; Unimplemented Instruction
    BR      _handle_Unimplemented_Instruction

    ORG     0x028                                       ; Privilege Violation
    BR      _handle_Privilege_Violation

    ORG     0x02C                                       ; Floating-Point: UNUSED IN AVR32A
    BR      _handle_Floating_Point

    ORG     0x030                                       ; Coprocessor Absent: UNUSED IN AVR32A
    BR      _handle_Coprocessor_Absent

    ORG     0x034                                       ; Data Address (Read)
    BR      _handle_Data_Address_Read

    ORG     0x038                                       ; Data Address (Write)
    BR      _handle_Data_Address_Write

    ORG     0x03C                                       ; DTLB Protection (Read)
    BR      _handle_DTLB_Protection_Read

    ORG     0x040                                       ; DTLB Protection (Write)
    BR      _handle_DTLB_Protection_Write

    ORG     0x044                                       ; DTLB Modified: UNUSED IN AVR32A
    BR      _handle_DTLB_Modified

    ORG     0x050                                       ; ITLB Miss: UNUSED IN AVR32A
    BR      _handle_ITLB_Miss

    ORG     0x060                                       ; DTLB Miss (Read): UNUSED IN AVR32A
    BR      _handle_DTLB_Miss_Read

    ORG     0x070                                       ; DTLB Miss (Write): UNUSED IN AVR32A
    BR      _handle_DTLB_Miss_Write

    ORG     0x100                                       ; Supervisor Call
    BR      _handle_Supervisor_Call
```

**Listing 5-14, VECTORS.ASM, Exception Vector Table**

L2-22(1)-(2) ??EVBA and _evba are both defined as the beginning of the exception vector table.

The SR has a exception mask (EM) bit. This bit set masks the exceptions. If exceptions are masked, all exceptions besides supervisor call exception are vectorized to the unrecoverable exception handler.

Unhandled exceptions can be caught by the `__unhandled_exception()` debug function in IAR. The prototype of this function is shown in listing 2-23. This function displays a dialog box showing which exception has been caught. On AVR32Studio, this function traps the execution flow and never returns. The exception handlers are shown in listing 2-24.

```
    void __unhandled_exception(void* offending_PC, unsigned int vector_number);
```

**Listing 2-23, `__unhandled_exception()` prototype**

L2-23 `vector_number` is passed thru `R11`, and `offending_PC` is retrieved from the stack.

```
    RSEG    EVSEG:CODE                                  ; (1)

    ALIGN   4

    EXTERN  __unhandled_exception                       ; (2)
    EXTERN  OSCtxSw                                     ; (3)

_handle_Unrecoverable_Exception:
    PUSHM   R11
    MOV     R11, 0x000                                  ; (4)
    BRAL    __unhandled_exception

_handle_TLB_Multiple_Hit:
    PUSHM   R11
    MOV     R11, 0x004                                  ; (5)
    BRAL    __unhandled_exception

_handle_Bus_Error_Data_Fetch:
    PUSHM   R11
    MOV     R11, 0x008                                  ; (6)
    BRAL    __unhandled_exception

_handle_Bus_Error_Instruction_Fetch:
    PUSHM   R11
    MOV     R11, 0x00C                                  ; (7)
    BRAL    __unhandled_exception

_handle_NMI:
    PUSHM   R11
    MOV     R11, 0x010                                  ; (8)
    BRAL    __unhandled_exception

_handle_Instruction_Address:
    PUSHM   R11
    MOV     R11, 0x014                                  ; (9)
    BRAL    __unhandled_exception

_handle_ITLB_Protection:
    PUSHM   R11
    MOV     R11, 0x018                                  ; (10)
    BRAL    __unhandled_exception

_handle_Breakpoint:
    PUSHM   R11
    MOV     R11, 0x01C                                  ; (11)
    BRAL    __unhandled_exception

_handle_Illegal_Opcode:
    PUSHM   R11
    MOV     R11, 0x020                                  ; (12)
```

```
    BRAL    __unhandled_exception

_handle_Unimplemented_Instruction:
    PUSHM   R11
    MOV     R11, 0x024                              ; (13)
    BRAL    __unhandled_exception

_handle_Privilege_Violation:
    PUSHM   R11
    MOV     R11, 0x028                              ; (14)
    BRAL    __unhandled_exception

_handle_Floating_Point:
    PUSHM   R11
    MOV     R11, 0x02C                              ; (15)
    BRAL    __unhandled_exception

_handle_Coprocessor_Absent:
    PUSHM   R11
    MOV     R11, 0x030                              ; (16)
    BRAL    __unhandled_exception

_handle_Data_Address_Read:
    PUSHM   R11
    MOV     R11, 0x034                              ; (17)
    BRAL    __unhandled_exception

_handle_Data_Address_Write:
    PUSHM   R11
    MOV     R11, 0x038                              ; (18)
    BRAL    __unhandled_exception

_handle_DTLB_Protection_Read:
    PUSHM   R11
    MOV     R11, 0x03C                              ; (19)
    BRAL    __unhandled_exception

_handle_DTLB_Protection_Write:
    PUSHM   R11
    MOV     R11, 0x040                              ; (20)
    BRAL    __unhandled_exception

_handle_DTLB_Modified:
    PUSHM   R11
    MOV     R11, 0x044                              ; (21)
    BRAL    __unhandled_exception

_handle_ITLB_Miss:
    PUSHM   R11
    MOV     R11, 0x050                              ; (22)
    BRAL    __unhandled_exception

_handle_DTLB_Miss_Read:
    PUSHM   R11
    MOV     R11, 0x060                              ; (23)
    BRAL    __unhandled_exception

_handle_DTLB_Miss_Write:
    PUSHM   R11
    MOV     R11, 0x070                              ; (24)
    BRAL    __unhandled_exception                   ; (25)

_handle_Supervisor_Call:
    LDDPC   PC, __OSCtxSw                           ; (26)

__OSCtxSw:
    DC32    OSCtxSw
```

**Listing 2-24, VECTORS.ASM, Exception handlers**

L2-24(1) Start exception handlers at a new segment. Since the exception vector branch to the exception handlers, they can be located anywhere a branch can jump to.

L2-24(2) Import __unhandled_exception() symbol to be used by the module.

L2-24(3) Import OSCtxSw() symbol to be used by the module.

L2-24(4)-(24) Load vector number into R11 to be passed to __unhandled_exception().

L2-24(25) Jump to __unhandled_exception().

L2-24(26) Jump to OSCtxSw() to perform OS context switch.

Exceptions can also force a CPU reset by executing the function CPU_Reset(). In order to add this functionality, first, the CPU_Reset() symbol needs to be imported in the module (EXTERN CPU_Reset), then the handler code must be replaced by a branch to CPU_Reset.

## 2.05.02 VECTORS.ASM, Interrupt Handlers

The AVR32 UC3 has an interrupt controller which allows the CPU to vector directly to the appropriate interrupt service routine (ISR). The AVR32 maintains a vector table which can accommodate up to 64 vectors, one for each group of interrupt requests. All interrupt signals in the same group share the same autovector address and priority level. When an interrupt occurs, the CPU saves R8-R12, LR, SR and PC onto the stack and extracts the appropriate ISR handler pointer from the vector table and simply jumps to that location. The base address of the vector table is an offset to the EVBA (Exception Vector Base Address) register.

For **μC/OS-II**, **EVERY** ISR must be written as shown in the pseudo-code in listing 2-25.

```
My_ISR()                                          (1)
    Disable ALL interrupts;                       (2)
    Save ALL registers;                           (3)
    OSIntNesting++;                               (4)
    if (OSIntNesting == 1) {                      (5)
        OSTCBCur->OSTCBStkPtr = SP;               (6)
    }
    /* Optionally, enable interrupts */           (7)
    My_ISR_Handler();                             (8)
    OSIntExit();                                  (9)
    Restore ALL registers                        (10)
    Return from Exception;                       (11)
```

**Listing 2-25, Pseudo-code for all ISRs under μC/OS-II**

L2-25(1) The ISR is entered and R8-R12, LR, SR and PC of the interrupted task are saved onto the current task's stack.

L2-25(2) Interrupts must be disabled because the code that follows must not be interrupted. This is done because the AVR32 architecture allows higher priority interrupts to be nested. Thus, for this point on, the code is guaranteed to be in a critical section.

L2-25(3) All CPU registers are saved onto the interrupted task's stack.

L2-25(4) Then, the interrupt nesting counter is incremented. `OSIntNesting` needs to be incremented by the ISR because `OSIntExit()` checks the value of this counter to determine whether it will return to task level code or, the next nested ISR.

L2-25(5)-(6) If this is the first nested ISR, then the task's stack pointer is saved onto the `TCB` (Task Control Block) of the interrupted task.

L2-25(7) At this point, interrupts can be re-enabled to allow higher-priority interrupts to be nested.

L2-25(8) Then call a user-defined C-level interrupt handler for the specific ISR. In other words, it is **NOT** necessary to have ALL the ISR code in assembly language, only what is not possible to do in C. Note also that the interrupt device must be cleared in the user-defined interrupt handler to prevent 're-entering' the interrupt when done.

L2-25(9) The call to `OSIntExit()`allows **μC/OS-II** to determine whether the ISR (or any nested ISRs) made a more important task (than the interrupted task) ready-to-run. If the interrupted task is still the most important task then `OSIntExit()` will return to this ISR.

L2-25(10) If `OSIntExit()` returns here, it means that the interrupted task is still the most important task. Then, all the registers are restored from the interrupted task's stack.

L2-25(11) The last step is to execute a return from exception instruction which causes the interrupted task to be resumed.

The process described above is shown graphically in Figure 2-5.

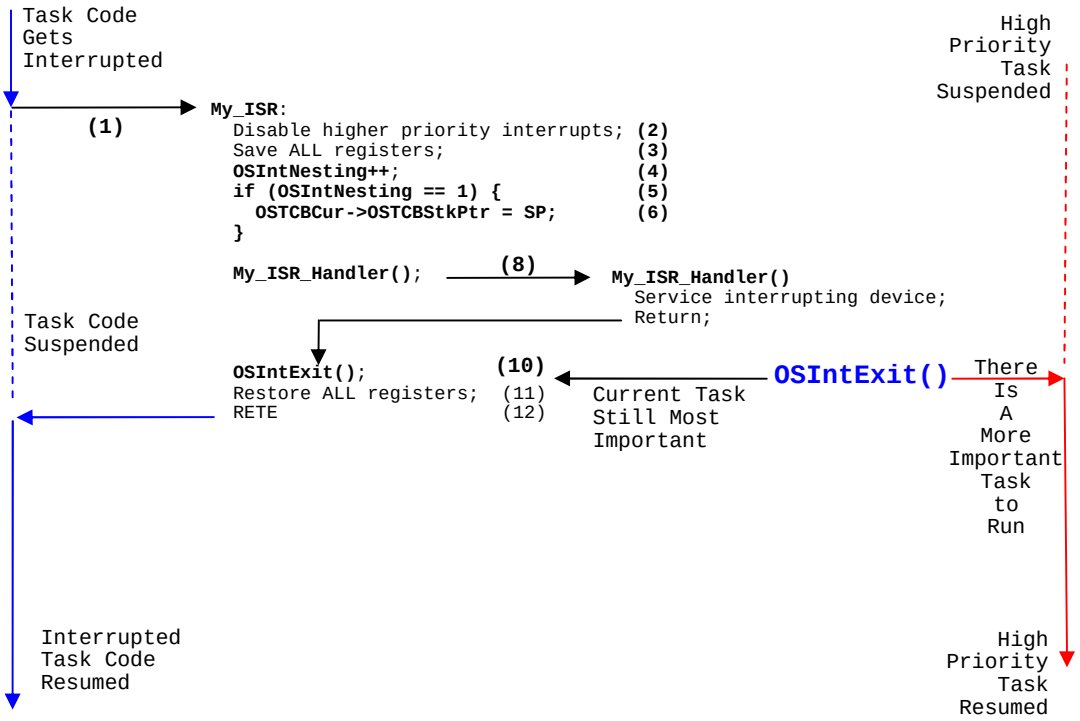



**Figure 2-5, Servicing Interrupts**

Since AVR32 UC3 only has 4 levels of interrupt priority, it was decided to implement an interrupt service routine for each interrupt priority levels, and inside the ISR determine the proper user-defined interrupt handler to call. This has the advantage that the user-defined interrupt handler only needs to worry about clearing the interrupt device, as opposed to handle all steps described in the pseudo-code for all ISR.

Prior to use any interrupt service, it is necessary to initialize the interrupt controller in the AVR32. This is accomplished by calling the function `BSP_INTC_Init()`. Then, every interrupt service request can be register to be handled by a user-defined handler with the function `BSP_INTC_IntReg()`. These functions are described in detail in the Board Support Package section.

## 2.05.03 VECTORS.ASM, IntX()

Each interrupt priority level is vectorized in a separate `IntX()` function: `Int0()`, `Int1()`, `Int2()`, and `Int3()`. This functions are responsible for retrieving the user-defined interrupt handler for the interrupt device requesting the interrupt service. In case a user-defined interrupt handler is registered, `IntX()` jumps to `OSIntISRHandler()` to execute it. The pseudo-code for the `IntX()` function is shown in listing 2-26.

```
IntX()                                                  (1)
    Disable ALL interrupts;                             (2)
    if (SR[M0:M2] >= 110b) {                            (3)
        Adjust stack to exception stack;                (4)
        Return from Exception;                          (5)
    }
    int_level = X;                                      (6)
    pHandler  = BSP_INTC_IntGetHandler(int_level);      (7)
    if (pHandler != NULL) {                             (8)
        OSIntISRHandler(pHandler);                      (9)
    }
    Return from Exception;                             (10)
```

**Listing 2-26, Pseudo-code for all `IntX()` functions**

L2-26(1) The `IntX()` function is replicated for each interrupt priority level, thus `X` take the values of: 0, 1, 2, and 3.

L2-26(2) Interrupts must be disabled because the code that follows must not be interrupted. This is done because the AVR32 architecture allows higher priority interrupts to be nested. Thus, for this point on, the code is guaranteed to be in a critical section.

L2-26(3)-(5) Check the current execution mode. If the execution mode is either Exception Mode or Non Maskable Interrupt Mode, then `SR` and `PC` are moved to the bottom of stack, `SP` is adjusted to match the exception stack frame, and the interrupted task is resumed.

L2-26(6) Store in `int_level` the respective interrupt priority level of the `IntX()`.

L2-26(7) Retrieve the user-defined interrupt handler by calling `BSP_INTC_IntGetHandler()`. This function is described in the Board Support Package section.

L2-26(8)-(9) Check if the user-defined interrupt handler points to a valid address, then calls `OSIntISRHandler()` to prepare the OS prior to call the user-defined interrupt handler.

L2-26(10) Return to the interrupted section of the code.

The actual `IntX()` code is shown in listing 2-27.

```
IntX()                                                  ; (1)
    SSRF    VECTORS_SR_GM_OFFSET                        ; (2)
    NOP
    NOP

    MFSR    R12, VECTORS_SR_OFFSET
    BFEXTU  R12, R12, VECTORS_SR_MX_OFFSET, 3
    CP.W    R12, VECTORS_SR_MX_EXCEPTION_MODE           ; (3)
    BRHS    __exception_stack                           ; (4)

    MOV     R12, VECTORS_INTX                           ; (6)
    RCALL   BSP_INTC_IntGetHandler                      ; (7)
    CP.W    R12, 0                                      ; (8)
    BRNE    OSIntISRHandler                             ; (9)
    RETE                                                ; (10)

__exception_stack:
    LDDSP   R12, SP[0 * 4]                              ; (4) Retrieve SR from stack
    STDSP   SP[6 * 4], R12                              ;     Store SR at bottom of stack
    LDDSP   R12, SP[1 * 4]                              ;     Retrieve PC from stack
    STDSP   SP[7 * 4], R12                              ;     Store PC at bottom of stack
    LDDSP   R12, SP[3 * 4]                              ;     Retrieve R12 back from stack
    SUB     SP, -6 * 4                                  ;     Adjust SP to match exception stack frame
    RETE                                                ; (5)
```

**Listing 2-27, Assembly code for all `OSIntX()` functions**

L2-27(2) Note that at least two `NOP`'s are required to guarantee interrupts are disabled when executing the next section of the function.

L2-27(7) Note that the user-defined interrupt handler is returned in `R12` by `BSP_INTC_IntGetHandler()`. This function is described in the Board Support Package section.

## 2.05.04 OS_CPU_A.ASM, OSIntISRHandler()

`OSIntISRHandler()`prepares the OS prior to call the user-defined interrupt handler. This function is called by `IntX()` when an user-defined interrupt handler is registered with `BSP_INTC_IntReg()` for a specific interrupt request.

`OSIntISRHandler()`is responsible for executing all steps required in a interrupt service routine under **μC/OS-II**. The pseudo-code for `OSIntISRHandler()`is shown in listing 2-28. Note that it follows all steps required for an ISR as discussed in the Handling Interrupts section.

```
OSIntISRHandler(CPU_FNCT_PTR ptrUserISR) {          (1)
    Save ALL registers;                             (2)
    OSIntNesting++;                                 (3)
    if (OSIntNesting == 1) {                        (4)
        OSTCBCur->OSTCBStkPtr = SP;                 (5)
    }
    /* Optionally, enable interrupts */             (6)
    ptrUserISR();                                   (7)
    OSIntExit();                                    (8)
    Restore ALL registers                           (9)
    Return from Exception;                         (10)
}
```

**Listing 2-28, Pseudo-code for `OSIntISRHandler()`**

L2-28(1) Once the ISR is entered, `R8-R12`, `LR`, `SR` and `PC` of the interrupted task are saved onto the current task's stack.

L2-28(2) All CPU registers need to be saved onto the interrupted task's stack.

L2-28(3) Then, the interrupt nesting counter is incremented. `OSIntNesting` needs to be incremented by the ISR because `OSIntExit()` checks the value of this counter to determine whether it will return to task level code or, the next nested ISR.

L2-28(4)-(5) If this is the first nested ISR, then the task's stack pointer is saved onto the `TCB` (Task Control Block) of the interrupted task.

L2-28(6) At this point, interrupts can be re-enabled to allow higher-priority interrupts to be nested.

L2-28(7) Then call a user-defined C-level interrupt handler for the specific ISR. Note that the interrupt device must be cleared in the user-defined interrupt handler to prevent 're-entering' the interrupt when done.

L2-28(8) The call to `OSIntExit()`allows **μC/OS-II** to determine whether the ISR (or any nested ISRs) made a more important task (than the interrupted task) ready-to-run. If the interrupted task is still the most important task then `OSIntExit()` will return to this ISR.

L2-28(9) If `OSIntExit()` returns here, it means that the interrupted task is still the most important task. Then, all the registers are restored from the interrupted task's stack.

L2-28(10) The last step is to execute a return from exception instruction which causes the interrupted task to be resumed.

The assembly code for `OSIntISRHandler()` is shown in listing 2-29.

```
OSIntISRHandler:                                ; (1)
    PUSHM   R0-R7                               ; (2)

    MOV     R11, WORD_LO(OSIntNesting)          ; (3)
    ORH     R11, WORD_HI(OSIntNesting)
    LD.UB   R10, R11[0]
    SUB     R10, -1
    ST.B    R11[0], R10

    CP.W    R10, 1                              ; (4)
    BRNE    OSIntISRHandler_1

    MOV     R11, WORD_LO(OSTCBCur)              ; (5)
    ORH     R11, WORD_HI(OSTCBCur)
    LD.W    R10, R11[0]
    ST.W    R10[0], SP

OSIntISRHandler_1:
    CSRF    OS_CPU_SR_GM_OFFSET                 ; (6)

    ICALL   R12                                 ; (7)
    RCALL   OSIntExit                           ; (8)

    POPM    R0-R7                               ; (9)
    RETE                                        ; (10)
```

**Listing 2-29, OS_CPU_A.ASM, `OSIntISRHandler()`**

L2-29(2) Since `R8-R12`, `LR`, `SR` and `PC` are pushed into the stack when the ISR is entered, only the remaining registers (`R0-R7`) need to be saved into the stack.

L2-29(6) If nested higher-priority interrupts need to be prevented from interrupting lower-priority ones, this line should be removed or commented out with a semi-colon.

L2-29(7) The user function is treated as a subroutine by the ISR handler. Note that all interrupt services with same or lower priority will not be served until the interrupt device is cleared in the user-defined interrupt handler.

## 2.05.05 VECTORS.ASM, IntFX()

In case an application requires low latency interrupts, a fast interrupt handler routine is available. This fast interrupt routine skips the OS context switch procedure, therefore it MUST not be used on the OS tick interrupt handler. Similarly to the standard interrupt handler, each interrupt priority level is vectorized in a separate `IntFX()` function: `IntF0()`, `IntF1()`, `IntF2()`, and `IntF3()`. This functions are responsible for retrieving the user-defined interrupt handler for the interrupt device requesting the interrupt service. In case a user-defined interrupt handler is registered, `IntFX()` jumps to `OSFastIntISRHandler()` to execute it. The pseudo-code for the `IntFX()` function is shown in listing 2-30.

```
IntFX()                                                  (1)
    Disable ALL interrupts;                              (2)
    int_level = X;                                       (3)
    pHandler  = BSP_INTC_IntGetHandler(int_level);       (4)
    if (pHandler != NULL) {                              (5)
        OSFastIntISRHandler(pHandler);                   (6)
    }
    Return from Exception;                               (7)
```

**Listing 2-30, Pseudo-code for all `IntFX()` functions**

L2-30(1) The `IntFX()` function is replicated for each interrupt priority level, thus `X` take the values of: 0, 1, 2, and 3.

L2-30(2) Interrupts must be disabled because the code that follows must not be interrupted. This is done because the AVR32 architecture allows higher priority interrupts to be nested. Thus, for this point on, the code is guaranteed to be in a critical section.

L2-30(3) Store in `int_level` the respective interrupt priority level of the `IntFX()`.

L2-30(4) Retrieve the user-defined interrupt handler by calling `BSP_INTC_IntGetHandler()`. This function is described in the Board Support Package section.

L2-30(5)-(6) Check if the user-defined interrupt handler points to a valid address, then calls `OSFastIntISRHandler()` to prepare the OS prior to call the user-defined interrupt handler.

L2-30(7) Return to the interrupted section of the code.

The actual IntFX() code is shown in listing 2-31.

```
IntX()                                                  ; (1)
    SSRF    VECTORS_SR_GM_OFFSET                        ; (2)
    NOP
    NOP

    MOV     R12, VECTORS_INTX                           ; (3)
    RCALL   BSP_INTC_IntGetHandler                      ; (4)
    CP.W    R12, 0                                      ; (5)
    BRNE    OSFastIntISRHandler                         ; (6)
    RETE                                                ; (7)
```

**Listing 2-31, Assembly code for all OSIntFX() functions**

L2-31(2) Note that at least two NOP's are required to guarantee interrupts are disabled when executing the next section of the function.

L2-31(4) Note that the user-defined interrupt handler is returned in R12 by BSP_INTC_IntGetHandler(). This function is described in the Board Support Package section.

## 2.05.06 OS_CPU_A.ASM, OSFastIntISRHandler()

OSFastIntISRHandler()prepares the OS prior to call the user-defined interrupt handler. This function is called by IntFX() when an user-defined interrupt handler is registered with BSP_INTC_FastIntReg() for a specific interrupt request.

OSFastIntISRHandler()is responsible for executing all steps required in a interrupt service routine under μC/OS-II. The pseudo-code for OSFastIntISRHandler()is shown in listing 2-32. Note that it does not follow all steps required for an ISR as discussed in the Handling Interrupts section. This is because the fast interrupt handler only prevents the OS from executing the context switch procedure.

```
OSFastIntISRHandler(CPU_FNCT_PTR ptrUserISR) {      (1)
    OSIntNesting++;                                 (2)
    /* Optionally, enable interrupts */             (3)
    ptrUserISR();                                   (4)
    Disable ALL interrupts;                         (5)
    OSIntNesting--;                                 (6)
    Return from Exception;                          (7)
}
```

**Listing 2-32, Pseudo-code for OSFastIntISRHandler()**

L2-32(1) Context is not required to be saved since there is no context switch performed.

L2-32(2) Interrupt nesting counter is incremented. OSIntNesting needs to be incremented by the ISR because it indicates to the OS that an interrupt service routine is being executed. Therefore, no context switch is performed calling any OS function.

L2-32(3) At this point, interrupts can be re-enabled to allow higher-priority interrupts to be nested.

L2-32(4) Then call a user-defined C-level interrupt handler for the specific ISR. Note that the interrupt device must be cleared in the user-defined interrupt handler to prevent 're-entering' the interrupt when done.

L2-32(5) Interrupts must be disabled because the code that follows must not be interrupted. This is done because the AVR32 architecture allows higher priority interrupts to be nested. Thus, for this point on, the code is guaranteed to be in a critical section.

L2-32(6) Interrupt nesting counter is decremented. `OSIntNesting` needs to be decremented by the ISR to indicate to the OS that the interrupt service routine has finished execution.

L2-32(7) The last step is to execute a return from exception instruction which causes the interrupted task to be resumed.

The assembly code for `OSFastIntISRHandler()` is shown in listing 2-33.

```
OSFastIntISRHandler:                          ; (1)
    MOV     R11, WORD_LO(OSIntNesting)        ; (2)
    ORH     R11, WORD_HI(OSIntNesting)
    LD.UB   R10, R11[0]
    SUB     R10, -1
    ST.B    R11[0], R10

    CSRF    OS_CPU_SR_GM_OFFSET               ; (3)

    ICALL   R12                               ; (4)

    SSRF    OS_CPU_SR_GM_OFFSET               ; (5)
    NOP
    NOP

    MOV     R11, WORD_LO(OSIntNesting)        ; (6)
    ORH     R11, WORD_HI(OSIntNesting)
    LD.UB   R10, R11[0]
    SUB     R10, 1
    ST.B    R11[0], R10

    RETE                                      ; (7)
```

**Listing 2-33, OS_CPU_A.ASM, `OSIntISRHandler()`**

L2-33(3) If nested higher-priority interrupts need to be prevented from interrupting lower-priority ones, this line should be removed or commented out with a semi-colon.

L2-33(4) The user function is treated as a subroutine by the ISR handler. Note that all interrupt services with same or lower priority will not be served until the interrupt device is cleared in the user-defined interrupt handler.

L2-33(5) Note that at least two `NOP`'s are required to guarantee interrupts are disabled when executing the next section of the function.

## 2.06 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.

For the IAR compiler, Micriμm has introduced a Windows-based 'Plug-In' module that makes use of this file. If IAR's C-Spy is used this files is required in the project build.

# 3.00 μC/CPU Port for AVR32 UC3

This port holds CPU-specific definitions for the AVR32 UC3 architecture.

## 3.01 Directories and Files

The software described in this section of this application note is assumed to be placed in the following directory:

```
\Micrium\Software\uC-CPU\AVR32\UC3\GNU

\Micrium\Software\uC-CPU\AVR32\UC3\IAR
```

The section for each source code file is found as follows:

| | |
|---|---|
| `CPU.H` | Section 3.02 |
| `CPU_A.ASM` | Section 3.03 |

## 3.02 CPU.H

CPU.H contains processor-specific #defines constants, macros, and typedefs.

### 3.02.01 CPU.H, Data Types

All data types used on the μC/OS-II port 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.

```
typedef            void      CPU_VOID;
typedef  unsigned  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     CPU_INT16U;            /* 16-bit unsigned integer      */
typedef    signed  short     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            float     CPU_FP32;              /* 32-bit floating point        */
typedef            double    CPU_FP64;              /* 64-bit floating point        */


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

**Listing 3-1, CPU.H, Data Types**

L3-1(1) Data type defined to replace commonly-used cast(s) for function pointers. Pointer to a function which returns void and has no arguments.

L3-1(2) Data type defined to replace commonly-used cast(s) for function pointers. Pointer to a function which returns void and has a void pointer argument.

## 3.02.02 CPU.H, Critical Sections

**μC/CPU** port defines two macros to enter and exit critical sections: `CPU_CRITICAL_ENTER()` and `CPU_CRITICAL_EXIT()`, respectively. The book (MicroC/OS-II, The Real-Time Kernel) describes the three different methods for dealing with critical sections. The one to choose depends on the processor and compiler. The **μC/CPU** port implements only the critical method #3.

The other critical method are not implemented and if selected as a critical method type, an error will abort the compilation of the project.

```
typedef  CPU_INT32U  CPU_SR;                                        /* Note 1           */

#define  CPU_CFG_CRITICAL_METHOD        CPU_CRITICAL_METHOD_STATUS_LOCAL  /* Note 2           */

#define  CPU_CRITICAL_ENTER()           { cpu_sr = CPU_SR_Save(); }       /* Note 3           */
#define  CPU_CRITICAL_EXIT()            { CPU_SR_Restore(cpu_sr); }       /* Note 4           */
```

**Listing 3-2, CPU.H, `CPU_CRITICAL_ENTER()` and `CPU_CRITICAL_EXIT()`**

L3-2(1) Defines CPU status register size.

L3-2(2) Configure CPU critical method.

L3-2(3) Call `CPU_SR_Save()` to enter in a critical section.

L3-2(4) Call `CPU_SR_Restore()` to exit a critical section.

## 3.02.03 CPU.H, Function Prototypes

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

```
CPU_SR      CPU_SR_Save(void);
void        CPU_SR_Restore(CPU_SR      cpu_sr);
void        CPU_IntDis(void);
void        CPU_IntEn(void);

void        CPU_ExceptDis(void);
void        CPU_ExceptEn(void);

void        CPU_Reset(void);

CPU_INT32U  CPU_SysReg_Get_Count(void);
CPU_INT32U  CPU_SysReg_Get_Config0(void);
CPU_INT32U  CPU_SysReg_Get_Compare(void);
void        CPU_SysReg_Set_Compare(CPU_INT32U value);

CPU_INT32U  CPU_CntLeadZeros(CPU_INT32U value);
```

**Listing 3-3, CPU.H, Function Prototypes**

## 3.03 CPU_A.ASM

CPU_A.ASM contains twelve assembly language functions for extending access to CPU registers to C-level. These functions are:

```
CPU_SR_Save()
CPU_SR_Restore()

CPU_IntDis()
CPU_IntEn()

CPU_ExceptDis()
CPU_ExceptEn()

CPU_Reset()

CPU_SysReg_Get_Count()
CPU_SysReg_Get_Config0()
CPU_SysReg_Get_Compare()
CPU_SysReg_Set_Compare()

CPU_CntLeadZeros()
```

## 3.03.01 CPU_A.ASM, Critical Sections

CPU_A.ASM contains the assembly language functions for entering and exiting critical sections. The **μC/CPU** port implements only the critical method #3. In Critical method #3, the status register of the CPU is saved in a local variable and all interrupts are disabled when a critical section is entered. To leave a critical section, the status register previously saved in the local variable is restored. By restoring the previous value of the CPU's status register, interrupts are only re-enabled if they were enabled just before entering the critical section. This guarantees nested critical sections to work properly.

```
CPU_SR_Save:
    CSRFCZ  CPU_SR_GM_OFFSET                        ; (1)
    SRCS    R12                                     ; (2)
    SSRF    CPU_SR_GM_OFFSET                        ; (3)
    NOP
    NOP
    MOV     PC, LR                                  ; (4)
```

**Listing 3-4, CPU_A.ASM, CPU_SR_Save()**

L3-4(1) Retrieve global mask (GM) bit from SR.

L3-4(2) Set R12 if GM is set.

L3-4(3) Disables interrupt by setting the GM bit in SR. Note that at least two NOP's are required to guarantee interrupts are disabled when returning from CPU_SR_Save().

L3-4(4) Restore PC with LR, which returns from subroutine. Note that this function returns R12 with the previous value of the global mask bit.

```
CPU_SR_Restore:
    PUSHM   R11                                     ; (1)

    MFSR    R11, CPU_SR_OFFSET                      ; (2)
    LSR     R12, 1                                  ; (3)
    BST     R11, CPU_SR_GM_OFFSET                   ; (4)

    MTSR    CPU_SR_OFFSET, R11                      ; (5)

    POPM    R11                                     ; (6)
    MOV     PC, LR                                  ; (7)
```

**Listing 3-5, CPU_A.ASM, CPU_SR_Restore()**

L3-5(1) Push R11 into stack, so it can be used internally by the function.

L3-5(2) Retrieve SR.

L3-5(3) Copy R12 status to Carry bit. Note that R12 is the function *sr* parameter.

L3-5(4) Overwrite GM bit with Carry bit.

L3-5(5) Restore SR GM bit with previous value.

L3-5(6) Restore R11 value from stack.

L3-5(7) Restore PC with LR, which returns from subroutine.

## 3.03.02 CPU_A.ASM, Disabling and Enabling Interrupts

Interrupts can be disabled and enabled using the `CPU_IntDis()` and `CPU_IntEn()` functions. The code for these functions are shown in listing 3-6 and listing 3-7, respectively.

```
CPU_IntDis:
    SSRF    CPU_SR_GM_OFFSET                        ; (1)
    NOP
    NOP
    MOV     PC, LR                                  ; (2)
```

**Listing 3-6, CPU_A.ASM, `CPU_IntDis()`**

L3-6(1) Disables interrupt by setting the GM bit in SR. Note that at least two `NOP`'s are required to guarantee interrupts are disabled when returning from `CPU_IntDis()`.

L3-6(2) Restore PC with LR, which returns from subroutine.

```
CPU_IntEn:
    CSRF    CPU_SR_GM_OFFSET                        ; (1)
    MOV     PC, LR                                  ; (2)
```

**Listing 3-7, CPU_A.ASM, `CPU_IntEn()`**

L3-7(1) Enables interrupt by clearing the GM bit in SR.

L3-7(2) Restore PC with LR, which returns from subroutine.

## 3.03.03 CPU_A.ASM, Disabling and Enabling Exceptions

Exceptions can be disabled and enabled using the `CPU_ExceptDis()` and `CPU_ExceptEn()` functions. If exceptions are disabled, all exceptions but supervisor calls will be handled by the `_handle_Unrecoverable_Exception` handler. The code for these functions are shown in listing 3-8 and listing 3-9, respectively.

```
CPU_ExceptDis:
    SSRF    CPU_SR_EM_OFFSET                        ; (1)
    MOV     PC, LR                                  ; (2)
```

**Listing 3-8, CPU_A.ASM, `CPU_ExceptDis()`**

L3-8(1) Disables exceptions by setting the EM bit in SR.

L3-8(2) Restore PC with LR, which returns from subroutine.

```
CPU_ExceptEn:
    CSRF    CPU_SR_EM_OFFSET                        ; (1)
    MOV     PC, LR                                  ; (2)
```

**Listing 3-9, CPU_A.ASM, `CPU_ExceptEn()`**

L3-9(1) Enables exceptions by clearing the EM bit in SR.

L3-9(2) Restore PC with LR, which returns from subroutine.

## 3.03.04 CPU_A.ASM, CPU_Reset()

This function is used to perform a software reset on the CPU by returning to the start-up function. The start-up function is always declared at the reset vector of the AVR32. On IAR, the start-up function is declared as `__program_start()`. On AVR32Studio, it is declared as `_start()`.

```
CPU_Reset:
    MFSR    R8, CPU_SR_OFFSET                                       ; (1)
    BFEXTU  R8, R8, CPU_SR_MX_OFFSET, 3                             ; (2)
    CP.W    R8, CPU_SR_MX_SUPERVISOR_MODE                           ; (3)
    BRNE    CPU_Reset_RETE                                          ; (4)

    MOV     R8, WORD_LO(CPU_START)                                  ; (5)
    ORH     R8, WORD_HI(CPU_START)                                  ; (6)
    MOV     R9, WORD_LO(CPU_SR_GM_MASK | CPU_SR_M0_MASK)            ; (7)
    ORH     R9, WORD_HI(CPU_SR_GM_MASK | CPU_SR_M0_MASK)            ; (8)
    STM     --SP, R8-R9                                             ; (9)
    RETS                                                            ; (10)

CPU_Reset_RETE:
    MOV     R8,  0x0808                                             ; (11)
    ORH     R8,  0x0808                                             ; (12)
    MOV     R9,  0x0909                                             ; (13)
    ORH     R9,  0x0909                                             ; (14)
    MOV     R10, 0x1010                                             ; (15)
    ORH     R10, 0x1010                                             ; (16)
    MOV     R11, 0x1111                                             ; (17)
    ORH     R11, 0x1111                                             ; (18)
    MOV     R12, 0x0000                                             ; (19)
    ORH     R12, 0x0000                                             ; (20)
    MOV     LR,  0x1414                                             ; (21)
    ORH     LR,  0x1414                                             ; (22)
    STM     --SP, R8-R12, LR                                        ; (23)
    MOV     R8, WORD_LO(CPU_START)                                  ; (24)
    ORH     R8, WORD_HI(CPU_START)                                  ; (25)
    MOV     R9, WORD_LO(CPU_SR_GM_MASK | CPU_SR_M0_MASK)            ; (26)
    ORH     R9, WORD_HI(CPU_SR_GM_MASK | CPU_SR_M0_MASK)            ; (27)
    STM     --SP, R8-R9                                             ; (28)
    RETE                                                            ; (29)
```

**Listing 3-10, CPU_A.ASM, `CPU_Reset()`**

L3-10(1)-(2) Retrieve SR and extract execution mode (M0:M3).

L3-10(3)-(4) If execution mode is not Supervisor Mode, then branch to `CPU_Reset_RETE`.

L3-10(5)-(8) Prepare the start-up function address, and a new SR with global interrupts disabled and supervisor mode.

L3-10(9) Push PC and SR into stack.

L3-10(10) Execute a return from supervisor call to load PC and SR into CPU registers. This jumps to the start-up function address.

L3-10(11)-(23) Load `R8-R12`, and `LR` with clean values and push them into the stack.

L3-10(24)-(27) Prepare the start-up function address, and a new SR with global interrupts disabled and supervisor mode.

L3-10(28) Push PC and SR into stack.

L3-10(29) Execute a return from event handler to load PC, SR, `R8-R12`, and `LR` into CPU registers. This jumps to the start-up function address.

## 3.03.05 CPU_A.ASM, System Registers

Some system registers need to be accessed from C-level in order to setup some functionality of the **μC/OS-II**, **μC/OS-View** and **μC/Probe**. These registers are: Count, Compare, and Config0.

The Count system register access the internal free-running counter. This free-running counter are always incrementing regardless of pipeline stalls or flushes. The Compare system register is used in conjunction with the Count system register to create a timer with interrupt functionality. The Config0 system register is used to retrieve information about the processor on which the operating system is running. Listing 3-11 shows the collection of assembly functions to retrieve the system registers information. Listing 3-12 shows the assembly function to alter the Compare system register.

```
CPU_SysReg_Get_Count:
    MFSR    R12, CPU_COUNT_OFFSET                   ; Retrieve COUNT system register
    MOV     PC, LR                                  ; Restore Program Counter (return)

CPU_SysReg_Get_Config0:
    MFSR    R12, CPU_CONFIG0_OFFSET                 ; Retrieve CONFIG0 system register
    MOV     PC, LR                                  ; Restore Program Counter (return)

CPU_SysReg_Get_Compare:
    MFSR    R12, CPU_COMPARE_OFFSET                 ; Retrieve COMPARE system register
    MOV     PC, LR                                  ; Restore Program Counter (return)
```

**Listing 3-11, CPU_A.ASM, Retrieve system registers**

```
CPU_SysReg_Set_Compare:
    MTSR    CPU_COMPARE_OFFSET, R12                 ; Save COMPARE system register
    MOV     PC, LR                                  ; Restore Program Counter (return)
```

**Listing 3-12, CPU_A.ASM, `CPU_SysReg_Set_Compare()`**

## 3.03.06 CPU_A.ASM, CPU_CntLeadZeros()

This function wraps the count leading zeros (CLZ) instruction into an assembly function to be accessed from C-level. This function counts the number of binary zero bits before the first binary one bit in the provided argument. If the argument is zero, the value 32 is returned. Listing 3-13 shows the assembly code for this function.

```
CPU_CntLeadZeros:
    CLZ     R12, R12                                    ; Count leading zeros
    MOV     PC, LR                                      ; Restore Program Counter (return)
```

**Listing 3-13, CPU_A.ASM, CPU_CntLeadZeros()**

This function, in particular, have a great use in determine the interrupt source respecting the prioritization used by the interrupt controller. The count leading zeros returns the highest interrupt line among multiple active interrupt sources.

# 4.00 BSP (Board Support Package) for AVR32 UC3

It is often convenient to create a Board Support Package (BSP) for the target hardware. A BSP allows the encapsulation of commonly used functionality which eases the application port, since the functions provided can be freely used by the application. A BSP can usually supports the following functionality:

Configuration of CPU and peripheral clock frequencies;
Determine CPU and peripheral clock frequencies;
Configure the LED I/Os;
Configuration and handling of the **μC/OS-II** tick timer;
Interrupt handling for the **μC/OS-II** tick timer;
Initialization of the **μC/OS-View** and **μC/Probe** communication channel;
Configuration and handling of the **μC/OS-View** and **μC/Probe** measurement timer;
Interrupt handling for the **μC/OS-View** and **μC/Probe** data transfers;
Configuration of the interrupt services;
General I/O access functions.

The BSP consist of 2 files: `BSP.C` and `BSP.H`. The `VECTORS.ASM` contains the exception vectors and it is also included under the BSP section. Since in general only the supervisor call exception is required in the application, `VECTORS.ASM` would remain unaltered.

The complete description of the BSP is available in the application notes for each evaluation board based on the AVR32 UC3 architecture.

# 5.00 μC/OS-View Port for AVR32 UC3

**μC/OS-View** is a module that allows to display useful statistics gathered from **μC/OS-II**. After licensing **μC/OS-View**'s source files from Micriμm and obtaining the module's Windows application, **μC/OS-View** can be used after completing a few simple operations.

First, a RS-232C serial connection is required between the target board the PC. Depending on the model of the evaluation board, the serial port is marked either "UART_1" or "USART1". Second, include **μC/OS-View** source and port files into the project. Third, in the **μC/OS-II** configuration file (`OS_CFG.H`), the `OS_VIEW_MODULE` has to be set to 1 to indicate the use of the USART 1.

After these preparations are completed, the application has to be rebuilt. The application can then be started and the **μC/OS-View**'s Windows application executed. The computer's communication settings must be configured to match the settings on the **μC/OS-View** initialization code. The baud rate and COM port used on the PC can be configure thru the 'setup' menu on the **μC/OS-View**'s Windows application. Once the configuration between the target board and the PC are matching, the Windows application will begin receiving packets from the board. These packets contain OS information that are displayed in textual and graphical format on the **μC/OS-View**'s Windows application.

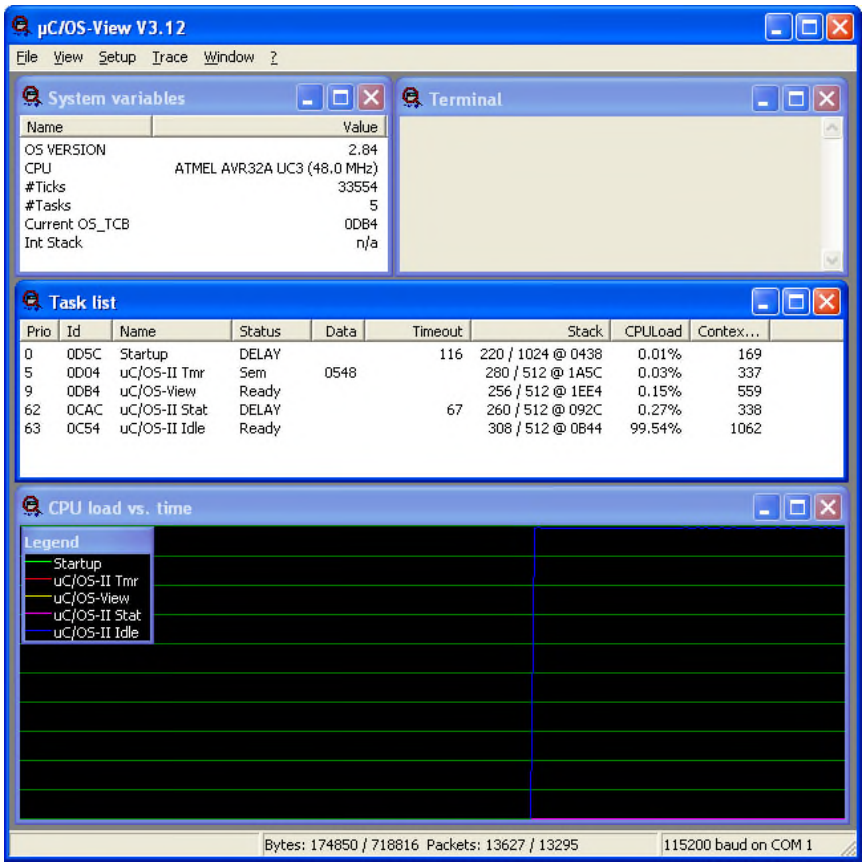



**Figure 5-1, μC/OS-View, Windows Application running on the EVK1100.**

**μC/OS-View** is a combination of a Microsoft Windows application program and code that resides in the target system (AVR32 UC3 evaluation board). The Windows application connects to the system via a RS-232C serial port. The status of the tasks which are managed by **μC/OS-II** are displayed with the Windows application.

**μC/OS-View** allows the following information from a **μC/OS-II** based product to be displayed:

Address of the TCB of each task (up to 63 tasks);
Name of each task (up to 63 tasks);
Status (e.g., ready, delayed, waiting on event) of each task;
Number of ticks remaining for a timeout or if task is delayed;
Amount of stack space used and left for each task;
Percentage of CPU usage time for each task;
Number of times each task has been 'switched-in';
Execution profile of each task.

**μC/OS-View** also allows commands to be sent to the target and replies to be received. These information are displayed in a 'terminal window'.

**μC/OS-View** is currently configured to use `USART1` as the default serial port. This may be changed by adjusting the macro `OS_VIEW_COMM_SEL` in `APP_CFG.H` to either `OS_VIEW_UART_0`, or `OS_VIEW_UART_1` based on the requirements of your application and evaluation board.

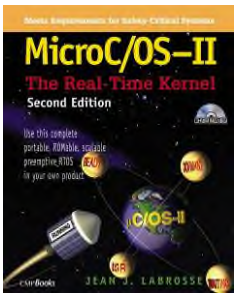
**μC/OS-View** is licensed on a per-developer basis. In other words, **μC/OS-View** is allowed to be installed on multiple PCs as long as the PC is used by the same developer. If multiple developers are using **μC/OS-View** then each one needs to obtain their respective copy. Contact Micriμm for pricing information.

## 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 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**. 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

***MicroC/OS-II, The Real-Time Kernel, 2nd Edition***
Jean J. Labrosse
CMP Books, 2002
ISBN 1-5782-0103-9




## Contacts

**Atmel Corporation**
2325 Orchard Parkway
San Jose, CA 95131
USA
+1 408 441 0311
WEB: www.Atmel.com

**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

**IAR Systems, Inc.**
Century Plaza
1065 E. Hillsdale Blvd
Foster City, CA 94404
USA
+1 650 287 4250
+1 650 287 4253 (FAX)
WEB: http://www.IAR.com
e-mail: info@IAR.com

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

## Notes