

# Micrium

Empowering Embedded Systems

## μC/OS-II

for the  
ARM1176JZF-S CP RTSM  
(using ARM'S RealView Development Suite)

### Application Note

AN-1176

## About Micrium

Micrium provides high-quality embedded software components in the industry by way of engineer-friendly source code, unsurpassed documentation, and customer support. The company's world-renowned real-time operating system, the Micrium μC/OS-II, features the highest-quality source code available for today's embedded market. Micrium delivers to the embedded marketplace a full portfolio of embedded software components that complement μC/OS-II. A TCP/IP stack, USB stack, CAN stack, File System (FS), Graphical User Interface (GUI), as well as many other high quality embedded components. Micrium's products consistently shorten time-to-market throughout all product development cycles. For additional information on Micrium, please visit [www.micrium.com](http://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 ARM's Real Time System Model (RTSM) for the ARM1176JZF-S on the Integrator/CP. This model integrates with ARM's RealView Debugger (RVD), allowing the ARM1176JZF-S processor core (and many of its peripherals) to be simulated in a Windows environment.

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

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

## Licensing

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

## Manual Version

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

Version	Date	By	Description
V.1.00	2007/01/18	BAN	Initial version.

## Software Versions

This document may or may not have been downloaded as part of an executable file, *Micrium-ARM-uCOS-II-ARM1176-CP.exe*, 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	V2.83	

## Document Conventions

### Numbers and Number Bases

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

### Typographical Conventions

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

## Table of Contents

<b>1.</b>	<b>Introduction</b>	<b>6</b>
<b>2.</b>	<b>Getting Started</b>	<b>7</b>
2.01	Software Requirements and Description	7
2.02	Opening and Viewing the Project	7
2.03	Using the Project and Loading the Target	8
2.04	Example Application	13
<b>3.</b>	<b>Directories and Files</b>	<b>14</b>
<b>4.</b>	<b>Application Code</b>	<b>16</b>
<b>5.</b>	<b>Board Support Package (BSP)</b>	<b>20</b>
5.01	RVDS-Specific BSP Files	20
5.02	Startup Code	20
5.03	BSP, <i>bsp.c</i> and <i>bsp.h</i>	21
5.03	Processor Initialization Functions	22
5.04	Exception Vector Handling and Servicing	24
	<b>Licensing</b>	<b>25</b>
	<b>References</b>	<b>25</b>
	<b>Contacts</b>	<b>25</b>

# 1. Introduction

This document, *AN-1176*, explains how to use **μC/OS-II** with ARM's RealView Development Suite (RVDS) toolchain and, specifically, with ARM's Real Time System Model (RTSM) for the ARM1176JZF-S processor on the Integrator/CP. The Integrator RTSM implementation includes models for on-chip peripherals such as timers, Ethernet, color LCD (CLCD), and interrupt controllers, each matched with the appropriate form of emulation for a Windows platform. The basic ARM1176 RTSM Window, as shown in Figure 1-1, allocates the appropriate amount of window space for the CLCD once the controller has been configured, as shown in Figure 1-2.

Only two model peripherals are used: the timer (to generate **μC/OS-II**'s tick interrupt) and the CLCD (for display). Consequently, the features used by the demonstration application described herein make use of only a fraction of the ARM1176 RTSM's capabilities. Both the simulator and the board have a Keyboard/Mouse Interface (KMI), allowing, for the simulator, relative mouse motion data to be streamed to the peripheral model's FIFOs. Transmissions from the UARTs are translated into network socket packets, which may be connected to by another application. Also, the board's LAN91C111 network interface can be used to communicate with a network.

If this appnote was downloaded in a packaged executable zip file, then it should have been found in the directory */Micrium/Appnotes/AN1xxx-RTOS/AN1176-uCOS-II-ARM-ARM1176-CP* and the code files referred to herein are located in the directory structure displayed in Section 2.02; these files are described in Section 3.

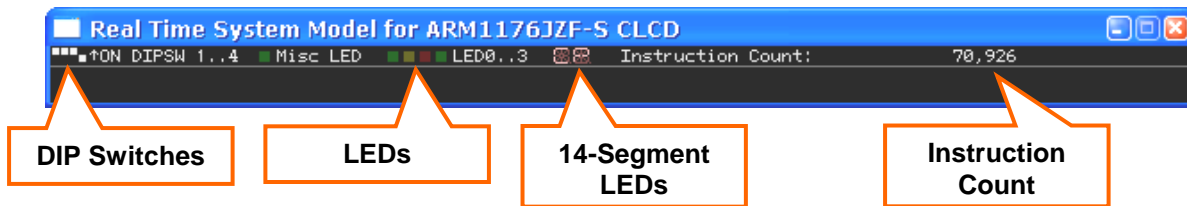


Figure 1-1. ARM1176JZF-S RTSM CLCD Window.



Figure 1-2. ARM1176JZF-S RTSM Interface with CLCD Allocation.

## 2. Getting Started

The following sections step through the prerequisites for using the demonstration application described in this document, *AN-1176*. The software requirements for using this project will be outlined including basic descriptions of the components used to build and test this project. The procedure for building and testing the project will then be delineated with a brief overview of the functionality presented in the demonstration application.

### 2.01 Software Requirements and Description

The RVDS is a multi-component system, covering the spectrum between an IDE for project development and compilation tools and a debugger. The port described in this document was programmed, compiled, and debugged using RVDS v3.0; the process of using these components (to the limited extent they were employed) is described. By so doing, this application presupposes that RVDS v3.0 or an equivalent utility is installed to compile and debug the application. In addition, the ARM1176JZF-S RTSM need be downloaded or obtained from ARM and installed as directed by ARM's documentation.

An IDE is provided with RVDS v3.0, being a version of CodeWarrior ("CodeWarrior for RVDS"); this possesses the management capabilities typical of a ARM development environment and is configured to invoke the ARM compilation tools, the RealView Compilation Tools (RVCT). Though RVCT can interface with utilities other than CodeWarrior for RVDS (the command line, for example), no other route was used. In development of this project (as evidenced by the descriptions herein), the compilation, assembly, and linking was all shepherded by CodeWarrior.

The "Debug" and "Run" commands in the CodeWarrior for RVDS automatically invoke the RealView Debugger (RVD). Code can be loaded onto the simulated target; the program can then be run and, upon stopping execution, updates of register values, static variables, and a memory dump, can be viewed as would be expected in a debugger. Beyond loading the code onto the target model, no description is made of the RVD's debugging capabilities or features, which are best described in the RVD documentation that accompanies RVDS v3.0.

### 2.02 Opening and Viewing the Project

If this file were downloaded as part of an executable zip file (which should have been named *Micrium-ARM-uCOS-II-ARM1176-CP.exe*), then the code files referred to herein are located in the directory structure shown in Figure 2-2.

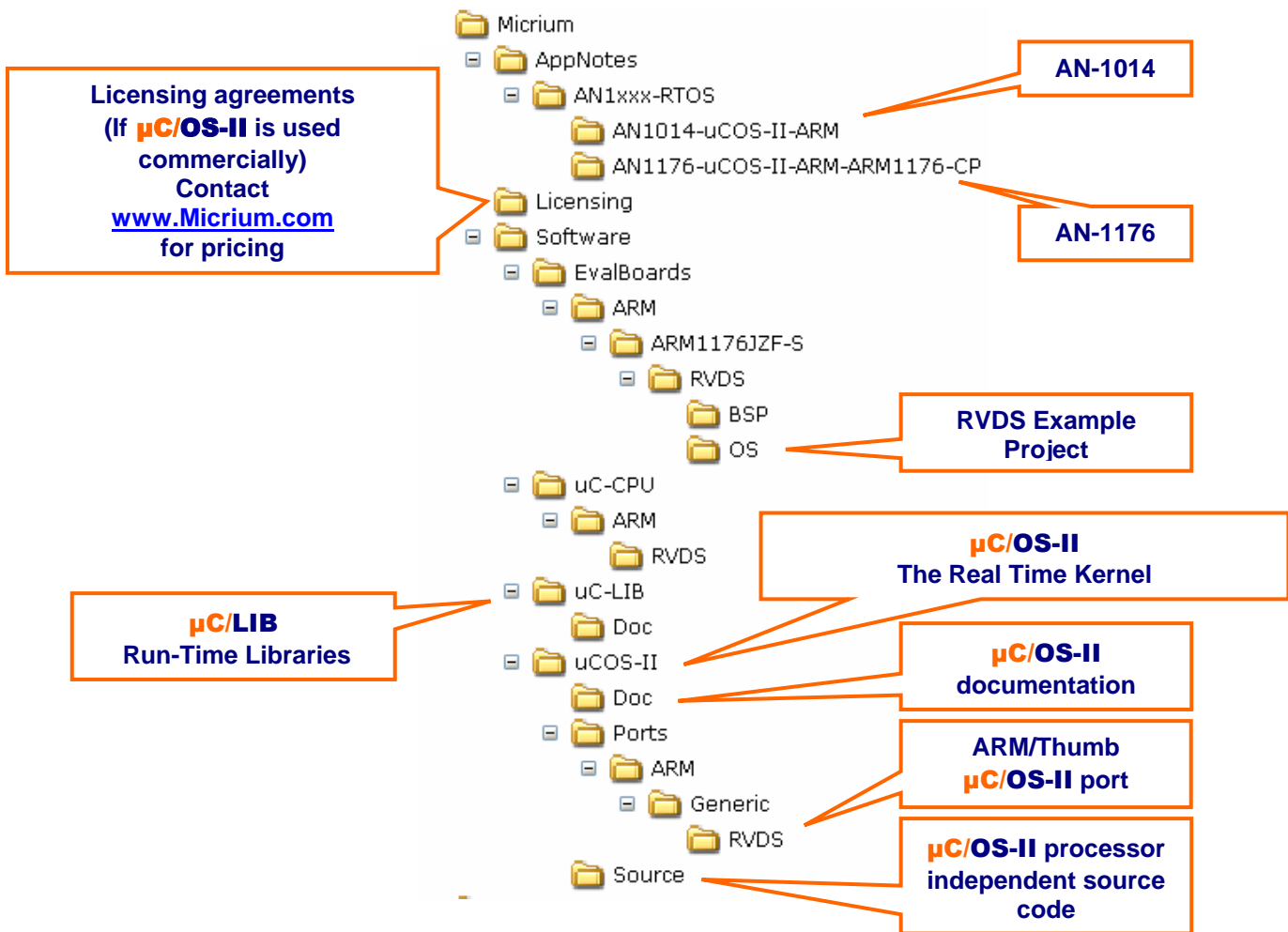


Figure 2-2, Directory Structure

An IAR project file named *ARM1176JZFS-OS.mcp* is located in the directory (marked “RVDS Example Project” in Figure 2-2)

*/Micrium/Software/EvalBoards/ARM/ARM1176JZF-S/RVDS/OS*

To view this example project, begin an instance of CodeWarrior for RVDS, select “Open...” from the “File” menu, navigate to the project directory, and select the project file.

## 2.03 Using the Project and Loading the Target

The full project file listing is revealed in Figure 2-3. Two configurations are included: one in which the code and data will be loaded into RAM; another, in which code will be loaded into Flash. The active configuration can be selected as shown in Figure 2-4, using the drop-down box above the project file listing.

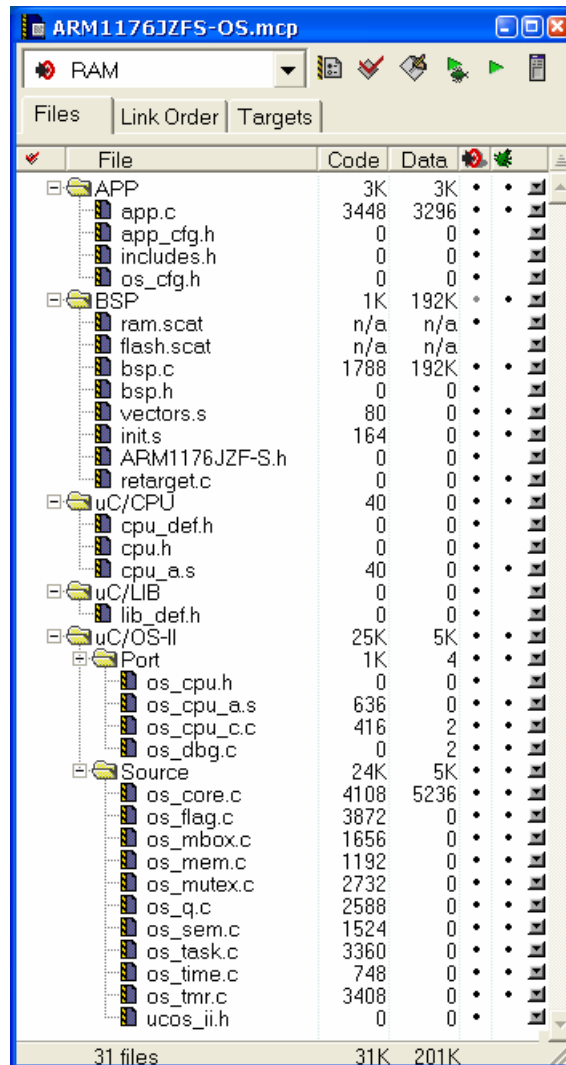


Figure 2-3, CodeWarrior Project Tree for ARM1176JZFS-OS.mcp

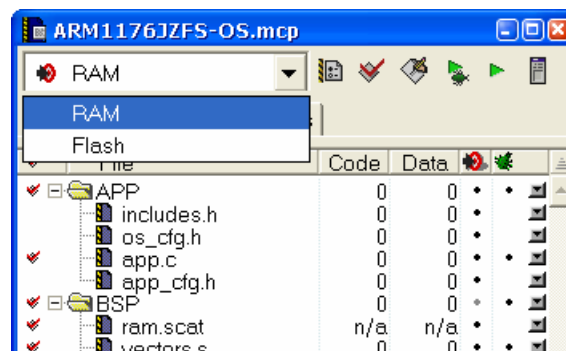
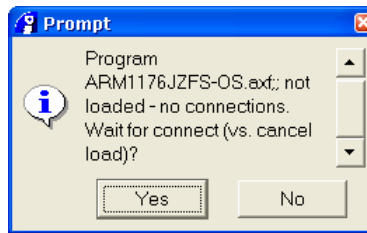


Figure 2-4, Selecting between Configurations in ARM1176JZFS-OS.mcp

Once RVDS V3.0 has been installed and the project has been opened, the image of one of the configurations may be loaded onto the model target.

The procedure for loading the RAM configuration is most straightforward:

1. **Run the project.** Select the RAM configuration (using the drop-down menu above the project tree, as shown in Figure 2-4) and select “Run” from the “Project” menu. Doing so will invoke RVD. The message box shown in Figure 2-4 will almost immediately appear; select “Yes” to proceed toward actual initiating the RTSM.



**Figure 2-4, Selecting between Configurations in *ARM1176JZFS-OS.mcp***

2. **Connect to target.** The primary program window is shown in Figure 2-5. As indicated in the upper-left pane (where disassembly or source code will later be displayed), RVD is currently unconnected to a target. To connect, either click the underlined text or select “Connect to Target...” from the “Target” menu. In the “Connection Control” dialog (Figure 2-6), double-click on the “localhost” entry (or click the “Open Target Access” button which the “localhost” entry is selected) and then double-click the “new\_ARM1176-CP-RT” option that appears. Several events will occur simultaneously: the ARM1176JZF-S RTSM window will open (Figure 2-7); a DOS window will appear; and the disassembly and code will load into RVD’s code window.
3. **Run the code.** The code is now loaded onto the model; to run the code, select “Run” from the “Debug” menu.

The flashloader image, *afu.axf*, included with the ARM1176JZF-S RTSM download (in the *apps* directory) will be used to program the model target with the image for the Flash configuration. (If you did not receive the *afu.axf* file or have not downloaded the RTSM, please contact ARM.) Assuming *afu.axf* is present on your system, load the Flash image as follows:

1. **Make the project.** Open the example Codewarrior for RVDS project and select the Flash configuration (as shown in Figure 2-4). Choose “Make” from the “Project” menu to produce the image, *ARM1176-OS.axf*, which should appear in the directory

`\\Micrium\Software\EvalBoards\ARM\ARM1176JZF-S\RVDS\OS\Build`

For the sake of convenience, move this file into the uppermost directory of your C:\ drive; this eliminates the hassle of typing an extended path into a command line.

2. **Connect to target.** Open RVD and connect to the ARM1176JZF-S target (as delineated above).

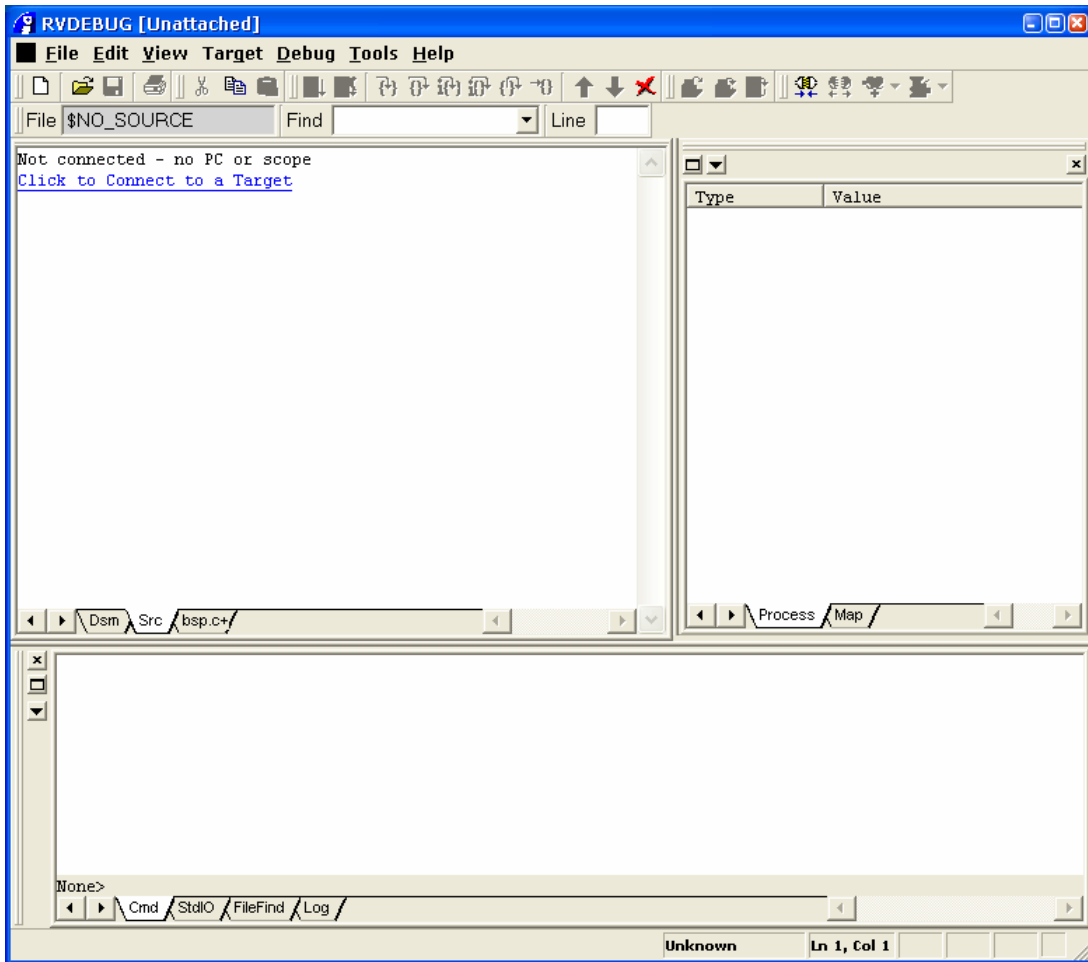


Figure 2-5, Primary RVD Window (Code Window)

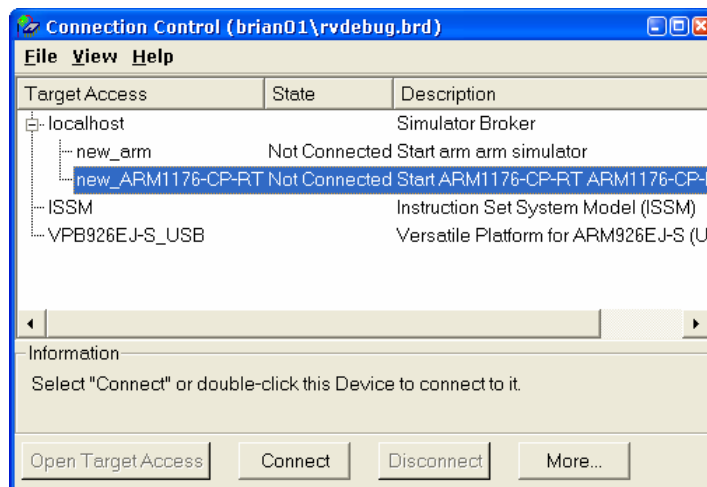
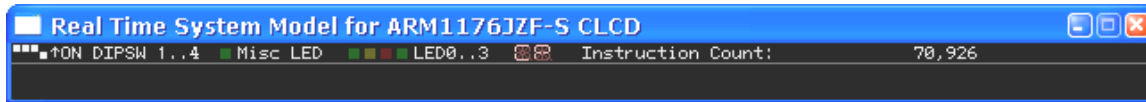
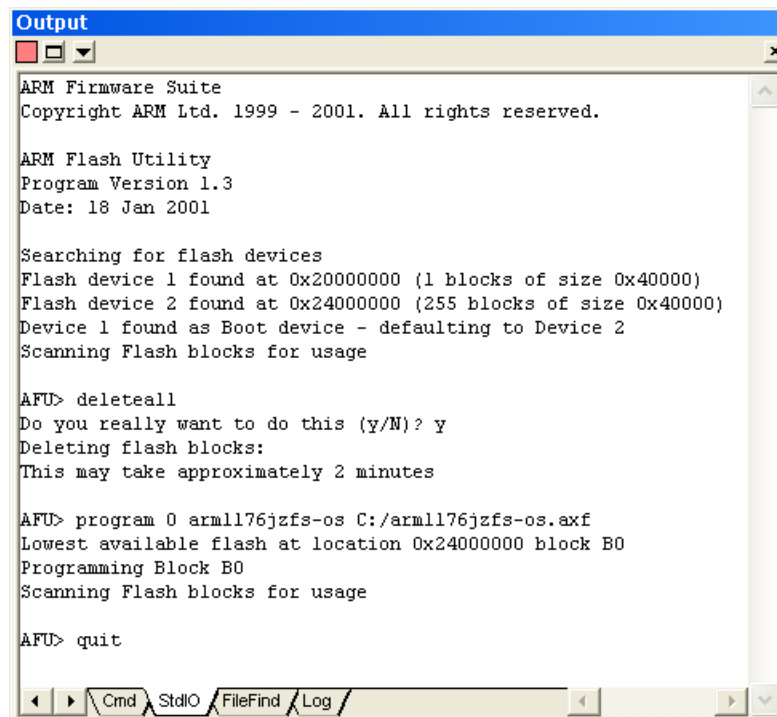


Figure 2-6, RVD Connection Control Window



**Figure 2-7, ARM1176JZF-S RTSM Window**

3. **Load *afu.axf* onto the target.** The text “No source for context: `_ENTRY_<entry point>`” should appear in the upper-left of the code window because no code has been loaded onto the target. Select “Load Image” from the “Target” menu and select the file *afu.axf* in the *apps* directory in the ARM1176JZF-S RTSM download from ARM. Select “Run” from the “Debug” menu.
4. **Load *ARM1176JZFS-OS.axf* into Flash.** Type in the StdIO pane (select “Output” from the “View” menu; choose “StdIO” tab) the following commands, producing the output shown in Figure 2-8:
  - a. `deleteall`
  - b. `program 0 arm1176jzfs-os c:/arm1176jzfs-os.axf`
  - c. `quit`
5. **Set the PC and run the code.** Open the registers window (choose “Registers” from the “View” menu and set the PC to `0x24000000` (the beginning of Flash). Choose “Run” from the “Debug” menu to begin running the example.



**Figure 2-8, StdIO Output from Flashloader (*afu.axf*)**

## 2.04 Example Application

When the example application is started, a summary of the current μC/OS-II state is displayed on the CLCD (as shown in Figure 2-9). This summary includes the tick rate (i.e., the number of ticks per second), the CPU usage, and two cumulative variables, one indicating the total number of ticks and the other the number of context switches that have occurred since the application was started. Below this are bar-graphs generated by two demonstration tasks. Eight μC/OS-II timers are initialized in AppTaskStart() with periods forming an arithmetic progression from 20 to 160 ticks (of OSTmrTime). The upper bar graph contains number of ticks left until each of these fires.

The lower bar graph contains the stack size, stack use, and maximum stack use for each task managed by μC/OS-II. The bar for each task is preceded by its name. The last task, “Factorial”, achieves variable stack usage by calculating a factorial using a recursive function (which delays before returning).

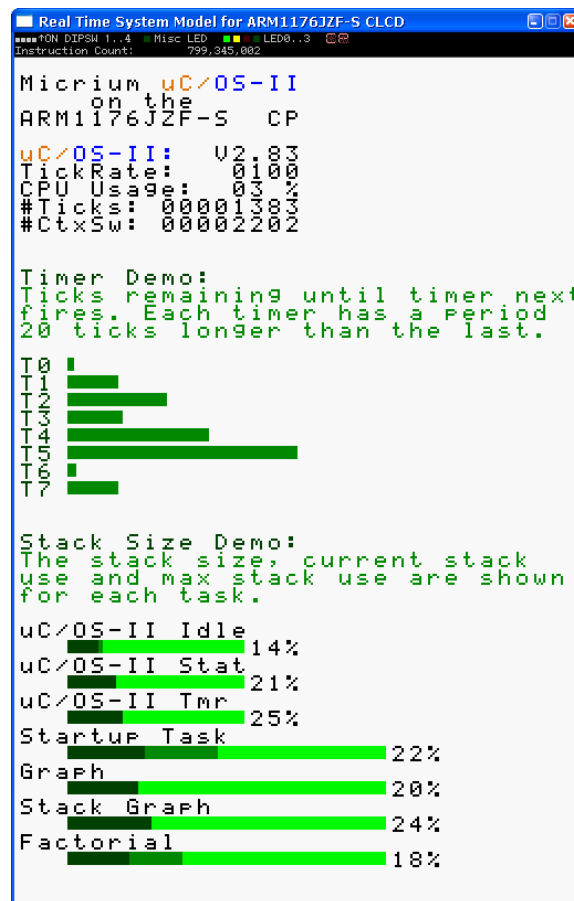


Figure 2-9, CLCD Window for Example Application

## 3. Directories and Files

### Application Notes

`\Micrium\AppNotes\ANxxx-RTOS\AN1014-uCOS-II-ARM`

This directory contains *AN-1014.pdf*, the application note describing the ARM port for μC/OS-II, and *AN-1014-PPT.pdf*, a supplement to *AN-1014.pdf*.

`\Micrium\AppNotes\AN9xxx-MULT\AN1176-ARM-ARM1176-CP`

This directory contains this application note, *AN-1176.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 in a commercial product, you must contact Micrium 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\ARM\Generic\RVDS`

This directory contains the standard processor-specific files for the generic μC/OS-II ARM port assuming the RVDS 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. The following files are in this directory:

- *os\_cpu.h*
- *os\_cpu\_a.asm*
- *os\_cpu\_c.c*
- *os\_dbg.c*

With this port, μC/OS-II can be used in either ARM or Thumb mode. The RVDS version of the ARM port was adapted from V1.70 of the port intended for the IAR toolchain; the required changes were minimal and largely relate to minor differences in assembler directives. The ARM/Thumb port is described in application note *AN-1014* which is available from the Micrium web site.

`\Micrium\Software\uCOS-II\Source`

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

### μC/CPU Files

`\Micrium\Software\uC-CPU`

This directory contains *cpu\_def.h*, which declares #define constants for CPU alignment, endianness, and other generic CPU properties.

## `\Micrium\Software\uC-CPU\ARM\RVDS`

This directory contains *cpu.h* and *cpu\_a.s*. *cpu.h* defines the Micrium portable data types for 8-, 16-, and 32-bit signed and unsigned integers (such as `CPU_INT16U`, a 16-bit unsigned integer). These allow code to be independent of processor and compiler word size definitions. *cpu\_a.s* contains generic assembly code for ARM7 and ARM9 processors which is used to enable and disable interrupts within the operating system. This code is called from C with `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`.

## µC/LIB Files

### `\Micrium\Software\uC-LIB`

This directory contains *lib\_def.h*, which provides `#defines` for useful constants (like `DEF_TRUE` and `DEF_DISABLED`) and macros.

### `\Micrium\Software\uC-LIB\Doc`

This directory contains the documentation for µC/LIB.

## Application Code

### `\Micrium\Software\EvalBoards\ARM\ARM1176JZF-S\RVDS\OS`

This directory contains the source code the example application:

- *app.c* contains the test code for the example application including calls to the functions that start multitasking within µC/OS-II, register tasks with the kernel, and update the user interface (the LEDs and LCD). *app\_cfg.h* is a configuration file specifying stack sizes and priorities for all user tasks and `#defines` for important global application constants.
- *includes.h* is the master include file used by the application.
- *os\_cfg.h* is the µC/OS-II configuration file.
- *ARM1176JZF-S-OS.mcp* is the CodeWarrior for RVDS project file.

### `\Micrium\Software\EvalBoards\ARM\ARM1176JZF-S\RVDS\OS`

This directory contains the Board Support Package for the ARM Integrator/CP board and ARM1176JZF-S RTSM:

- *bsp.c* contains the board support package functions which initialize critical functions (e.g., the tick interrupt for µC/OS-II) and provide support for peripherals such as the CLCD display and the LEDs on the board. *bsp.h* contains prototypes for functions that may be called by the user.
- *ARM1176JZF-S* contains `#defines` for processor and peripheral registers and fields of registers..
- *ram.scat* and *flash.scat* are scatter-loading files describing the placement of data in memory. The former instructs the linker that all data and code are to be placed in RAM; the latter instructs the linker to place data in RAM and code in Flash.
- *retarget.c* could define functions which allow, for example, `printf()` statements to be directed to a UART rather than the RVD console. Because such functionality is not needed, this file is empty.
- *init.s* and *vector.s* are assembly-language files providing low-level configuration. *init.s* performs a remap if the code is loaded in Flash and, no matter whether code is loaded in RAM or Flash, assigns the processor stack pointers for each processor mode and enables interrupts. *vector.s* defines the exception vectors which will either be loaded at `0x00000000` or remapped there; these will direct any IRQs or FIQs to µC/OS-II's interrupt-handling logic.

## 4. Application Code

The example application described in this appnote, *AN-1176*, is a simple demonstration of μC/OS-II for the ARM 1176JZF-S RTSM. The basic procedure for setting up and using μC/OS-II, “The Real-Time Kernel”, can be gleaned from an inspection of *app.c*, which could serve as a beginning template for further use of the software on the platform.

Four tasks are created in *app.c*: `AppTaskStart()`, `AppTaskGraph()`, `AppTaskStacks()`, and `AppTaskFactorial()`. Each of these, as with all tasks managed by μC/OS-II, will enter an infinite loop, performing some duty and intermittently yielding to other tasks (by delaying, pending on a semaphore, waiting in a queue, etc.). These tasks perform the following duties:

1. `AppTaskStart()`, after calling `BSP_Init()`, initializing the system timers used in the demonstration, and creating the other tasks, periodically updates the system information displayed on the CLCD. See Listing 4-2.
2. `AppTaskGraph()` updates the timer bargraph on the CLCD.
3. `AppTaskStacks()` updates the μC/OS-II task stack usage bargraph on the CLCD.
4. `AppTaskFactorial()` repeatedly calculates the factorial of a small integer (between 0 and 60) using a recursive factorial routine. This task, as can be seen in the task stack usage bargraph on the CLCD, has variable stack usage due to the recursive routine it calls.

The function `main()` serves as the entry point for the application, as it does in most C programs. This function initializes the operating system, creates the initial application task, `AppTaskStart()`, begins multitasking, and exits.

### Listing 4-1, `main()`

```

void main (void)                                     /* (1) */
{
    CPU_INT08U  err;

    BSP_IntDisAll();                                 /* (2) */

    OSInit();                                        /* (3) */

    OSTaskCreateExt(AppTaskStart,                    /* (4) */
                    (void *)0,
                    (OS_STK *)&AppTaskStartStk[APP_TASK_START_STK_SIZE - 1],
                    APP_TASK_START_PRIO,
                    APP_TASK_START_PRIO,
                    (OS_STK *)&AppTask_StartStk[0],
                    APP_TASK_START_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

    #if OS_TASK_NAME_SIZE > 13                       /* (5) */
        OSTaskNameSet(APP_TASK_START_PRIO, "Start Task", &err);
    #endif

    OSStart();                                       /* (6) */
}

```

- L4-1(1)** As with most C applications, the code starts in `main()`.
- L4-1(2)** All interrupts are disabled to make sure the application does not get interrupted until it is fully initialized.
- L4-1(3)** `OSInit()` must be called before creating a task or any other kernel object, as must be done with all **μC/OS-II** applications.
- L4-1(4)** At least one task must be created (in this case, using `OSTaskCreateExt()` to obtain additional information about the task). In addition, **μC/OS-II** creates either one or two internal tasks in `OSInit()`. **μC/OS-II** always creates an idle task, `OS_TaskIdle()`, and will create a statistic task, `OS_TaskStat()` if you set `OS_TASK_STAT_EN` to 1 in `os_cfg.h`.
- L4-1(5)** As of V2.6x, you can now name **μC/OS-II** tasks (and other kernel objects) and display task names at run-time or with a debugger. In this case, the `AppTaskStart()` is given the name "Start Task".
- L4-1(6)** Finally multitasking under **μC/OS-II** is started by calling `OSStart()`. **μC/OS-II** will then begin executing `AppTaskStart()` since that is the highest-priority task created (both `OS_TaskStat()` and `OS_TaskIdle()` having lower priorities).

## Listing 4-2, AppTaskStart ( )

```
static void AppTaskStart (void *p_arg)
{
    CPU_INT08U    i;
    CPU_INT08U    j;
    CPU_INT08U    s[20];
    CPU_INT08U    err;
    CPU_INT32U    period;

    (void)p_arg;

    BSP_Init();                               /* (1) */

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

    #if OS_TMR_EN > 0                          /* (3) */
        for (i = 0; i < OS_TMR_CFG_MAX; i++) {
            OS_StrCopy(s, "Timer xx");
            s[6] = i / 10 + '0';
            s[7] = i % 10 + '0';
            period = (i + 1) * 20;
            AppTmrTbl[i].AppTmr = OSTmrCreate(0, period, OS_TMR_OPT_PERIODIC,
                (OS_TMR_CALLBACK)AppTmrCallback,
                (void *)i, s, &err);

            AppTmrTbl[i].AppTmrCtr = 0;
            OSTmrStart(AppTmrTbl[i].AppTmr, &err);
        }
    #endif

    AppTaskCreate();                           /* (4) */

    AppPrintPage();

    while (DEF_TRUE) {
        OSTimeDlyHMSM(0, 0, 0, 100);
        j = (j + 1) % 2;
        if (j == 0) {
            AppUpdatePage();                   /* (5) */
        }

        for (i = 0; i < 4; i++) {
            if (PB_GetStatus(i) == DEF_TRUE) {
                LED_Toggle(i);
            }
        }
    }
}
```

**L4-2(1)** BSP\_Init() initializes the Board Support Package—the CLCD and tick interrupt, for this application. See Section 5 for details.

**L4-2(2)** OSStatInit() initializes μC/OS-II's statistic task. This only occurs if you enable the statistic task by setting OS\_TASK\_STAT\_EN to 1 in *os\_cfg.h*. The statistic task measures overall CPU usage (expressed as a percentage) and performs stack checking for all the tasks that have been created with OSTaskCreateExt() with the stack checking option set.

**L4-2(3)** Eight timers are created with a linear progression of periods, extending from 20 to 160 ticks. When the timer fires, the callback function AppTmrCallback() will be called.

**L4-2(4)** The other example tasks, AppTaskGraph(), AppTaskStacks(), and AppTaskFactorial(), are created.

- L4-2(5)** Any task managed by **μC/OS-II** must either enter an infinite loop 'waiting' for some event to occur or terminate itself. In an infinite loop, the task updates the **μC/OS-II** information on the CLCD display.

## 5. Board Support Package (BSP)

The Board Support Package (BSP) provides functions to encapsulate common I/O access functions and make porting your application code easier. Essentially, these files are the interface between the application and the ARM1176JZF-S RTSM model board. Though one file, *bsp.c*, contains some functions which are intended to be called directly by the user (all of which are prototyped in *bsp.h*), the other files serve the compiler (as with *flash.scat*).

### 5.01 RVDS-Specific BSP Files

The BSP includes three files intended specifically for use with RVDS tools: *retarget.c*, *ram.scat*, and *flash.scat*. The first (which is currently empty) could be used to re-define (or “retarget”) functions such as `printf()` or `exit()` to the platform. `printf()` could be directed to write a character to a UART, for example, rather than to the debugger’s StdIO window. These serve to define the memory map and initialize the processor prior to loading or executing code.

Before the processor memories can be programmed, the compiler must know where code and data should be placed. With RVDS, complex memory maps can be defined using a scatter-loading file. *flash.scat*, which directs the compiler to place read-only data into Flash and read-write data into RAM, is shown in Listing 5-1. *ram.scat* is similar (but simpler): all data and code are placed into RAM.

**Listing 5-1, *flash.scat***

```
FLASH 0x24000000 0x40000000      /* The 64MB Flash begins at 0x24000000      */
{
  FLASH 0x24000000 0x40000000
  {
    init.o (INIT, +First)      /* Initialization code is placed in Flash  */
    * (+RO)                    /* Read-only data and code is placed in Flash*/
  }
  32bitRAM 0x0000 0x0003BFFF    /* The 64kB RAM begins at 0x00000000      */
  {
    vectors.o (VECT, +First)   /* The ARM vectors are placed into RAM    */
    * (+RW,+ZI)               /* All other data is placed into RAM     */
  }

                                /* The “two-region model” is selected by the */
                                /* following definitions. A default       */
                                /* __user_initial_stackheap() will be used. */
    ARM_LIB_HEAP 0x0003C000 EMPTY 0x00004000-0x00002000 {}
    ARM_LIB_STACK 0x00040000 EMPTY -0x00002000 {}
  }
}
```

### 5.02 Startup Code

Two files provide assembly-language start-up: *vectors.s* and *init.s*. In the former, the μC/OS-II IRQ and FIQ handlers are placed into the ARM exception vector table, which will either be written to or remapped to 0x00000000. The latter performs a remap (if code is to be loaded into Flash), initializes the stack pointers for each processor mode, and moves execution to further setup in `__main()`.

## 5.03 BSP, *bsp.c* and *bsp.h*

The file *bsp.c* implements several global functions, each providing some important service, be that the initialization of processor functions for μC/OS-II to operate or the toggling of an LED. Several local functions are defined as well to perform some atomic duty, such as initializing the CLCD. With a few exceptions (notably `Tmr_TickInit()`), the discussion of the BSP will be limited to the coverage of the global functions that might be called from user code (and may be called from the example application)

The global functions defined in *bsp.c* (and prototyped in *bsp.h*) may be roughly divided into two categories: critical processor initialization and user interface services. Four functions constitute the former:

- **BSP\_Init()** is called by the application code to initialize critical processor features (particularly the μC/OS-II tick interrupt) after multitasking has started (i.e., `OS_Start()` has been called). This function should be called before any other BSP functions are used. See Listing 5-1 for more details.
- **BSP\_IntDisAll()** is called to disable all interrupts, thereby preventing any interrupts until the processor is ready to handle them.
- **BSP\_VectSet()** initializes the IRQ vector for the specified IRQ number
- **BSP\_CPU\_ClkFreq()** returns the clock frequency in kHz. For the ARM1176 RTSM, where no verifiable, hardware basis exists for the clock, this value merely approximates the apparent clock frequency of the model processor.

Nine functions provide access to user interface components:

- **LED\_Toggle()**, **LED\_On()** and **LED\_Off()** will toggle, turn on, and turn off (respectively) the LED corresponding to the ID passed as the argument (on the ARM1176 RTSM, between 1 and 4, inclusive). If an argument of 0 is provided, the appropriate action will be performed on all LEDs.
- **PB\_GetStatus()** takes as its argument the ID (on the ARM1176 RTSM, between 1 and 4, inclusive) of a push button and returns `DEF_TRUE` if the switch is on (in the “up” position) and `DEF_FALSE` if the switch is off (in the “down” position).
- The remaining functions are the user interface for the LCD.
  - **LCD\_DispStr()** displays a string (the third argument) at a specified column and line (the first and second arguments) in a specified color (the fourth argument).
  - **LCD\_DispChar()** displays a character (the third argument) at a specified column and line (the first and second arguments) in a specified color (the fourth argument).
  - **LCD\_ClrScr()** sets the entire display to the value `LCD_BGColor`, a variable defined in *bsp.c*.
  - **LCD\_DispClrLine()** sets the specified line of display to the color `LCD_BGColor`.
  - **LCD\_DispHorBar()** can be used to construct bargraphs (as it is employed in this demonstration application). Besides the location of the bar (the starting line

and column are the first and second arguments of the function), two parameters are specified about the bar's size: the total length (in pixels) allowed for the bar (the function's fifth argument, `total_len`) and the length of the bar (in pixels) that should be displayed (the function's fourth argument, `bar_len`). The pixels in the bar with column number less than or equal to `bar_len` are displayed in the specified color (the function's sixth argument); the pixels in the bar with column number between `bar_len` and `total_len` are displayed in the background color, effectively overwriting the previous contents of the bar.

For some of the CLCD interface functions, the final argument is a 16-bit integer color, in which the red, green, and blue components are each allotted 5-bit fields. The final bit may be used by some displays, in which it specifies intensity or serves as an additional bit for one (or all) of the RGB components; however, this bit is not used on the CLCD display for the ARM1176JZF-S.

The line and column passed as arguments to the CLCD interface functions are the character line and character column of the display (as opposed to the pixel line/row and pixel column). An array of bit-mapped representations of the characters with ASCII codes between 0x20 and 0xB5 (' ' and '~') are provided in the array `LCD_Charset[][]` defined in *bsp.c*. The representation for a character passed to `LCD_Dispatch()` (or a character in the string passed to `LCD_DispatchStr()`) is retrieved from the table and displayed at the specified character location of the display. Consequently, the LCD interface functions effectively provide a character-driven interface for the CLCD.

## 5.03 Processor Initialization Functions

### Listing 5-1, `BSP_Init()`

```
void BSP_Init (void)
{
    LCD_Init();                /* (1) */
    Tmr_TickInit();           /* (2) */
}
```

**L5-1(1)** The LCD is initialized.

**L5-1(2)** The μC/OS-II tick interrupt source is initialized. See Listing 5-2 for details.

Listings 5-2 and 5-3 give the μC/OS-II timer tick initialization function, `Tmr_TickInit()`, the tick ISR handler, `Tmr_TickISR_Handler()`. These may serve as examples for initializing an interrupt and servicing that interrupt.

### Listing 5-2, `Tmr_TickInit()`

```
static void Tmr_TickInit (void)
{
    CPU_INT32U  reload;

    reload      = (CPU_CLK_FREQ >> 8) / OS_TICKS_PER_SEC;  /* (1) */

    BSP_VectSet(PIC_TIMERINT1, Tmr_TickISR_Handler);      /* (2) */

    TIMER1_CONTROL    = 0;
    TIMER1_INTCLR     = 0;

    TIMER1_LOAD       = reload;

    TIMER1_CONTROL    = TIMER_CONTROL_ENABLE              /* (3) */
                       | TIMER_CONTROL_PERIODIC
                       | TIMER_CONTROL_PRESCALE_256;

    PIC_IRQ_ENABLESET = PIC_IRQ_TIMERINT1;                /* (4) */
}
```

**L5-2(1)** The timer reload value is calculated. The right-shift by eight bits is a division by 256, accounting for the prescaler used.

**L5-2(2)** The timer tick ISR handler, `Tmr_TickISR_Handler()`, is assigned to the appropriate IRQ. See Listing 5-4 for more information.

**L5-2(3)** The timer is configured to be periodic with a prescaler of 256. In this configuration, the timer will count-down from its initial value; when it reach 0, a timer interrupt will be generated and the timer will be loaded with the value in the `TIMER1_LOAD` register.

**L5-2(4)** The interrupt is enabled in the PIC (Primary Interrupt Controller).

### Listing 5-3, `Tmr_TickISR_Handler()`

```
void Tmr_TickISR_Handler (void)
{
    OSTimeTick();                                         /* (1) */
    TIMER1_INTCLR    = 0;                                 /* (2) */
}
```

**L5-3(1)** `OSTimeTick()` is called to inform μC/OS-II of the tick interrupt.

**L5-3(2)** The interrupt is cleared within the `TIMER1` block by clearing the `TIMER1_INTCLR` register.

## 5.04 Exception Vector Handling and Servicing

### Listing 5-4, BSP\_VectSet ( )

BSP\_VectSet ( ) is used to associate a certain ISR with a particular IRQ vector within the PIC. The first argument is the IRQ number; the second is the pointer to the ISR. A table is maintained in *bsp.c* containing these ISR function pointers and when an interrupt occurs, the appropriate handler is retrieved and invoked.

```
void BSP_VectSet (CPU_INT08U irq, BSP_PFNCT isr)
{
    if (irq < 32) {
        BSP_PICVectors[irq] = isr;
    }
}
```

### Listing 5-5, OS\_CPU\_IRQ\_ISR\_Handler ( )

OS\_CPU\_IRQ\_ISR\_Handler ( ) is called by OS\_CPU\_IRQ\_ISR ( ) (which is placed on the ARM IRQ exception vector) to invoke the appropriate ISR.

```
void OS_CPU_IRQ_ISR_Handler (void)
{
    CPU_INT32U status;
    CPU_INT32U i;
    BSP_PFNCT pfncnt;

    status = PIC_IRQ_STATUS; /* (1) */

    for (i = 0; i < 32; i++) { /* (2) */
        if ((status & (1 << i)) != 0) {
            pfncnt = BSP_PICVectors[i];
            if (pfncnt != (BSP_PFNCT)0) {
                pfncnt();
            }
        }
    }
}
```

**L5-5(1)** The IRQ status is read from the PIC\_IRQ\_STATUS register. Each of the 32 vectors is represented by a bit in this register.

**L5-5(2)** If the *i*th bit is set (indicating that the associated interrupt has been triggered) and a ISR has been placed into the BSP\_PICVectors [ ] table for this interrupt, then the ISR is called.

## Licensing

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

## References

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

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

### *Embedded Systems Building Blocks*

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

## Contacts

### **ARM**

141 Caspian Court  
Sunnyvale, CA 94089-1013  
+1 408 734 5600  
+1 408 734 5050  
WEB : [www.arm.com](http://www.arm.com)

### **Micrium**

949 Crestview Circle  
Weston, FL 33327  
USA  
+1 954 217 2036  
+1 954 217 2037 (FAX)  
e-mail: [Jean.Labrosse@Micrium.com](mailto:Jean.Labrosse@Micrium.com)  
WEB : [www.Micrium.com](http://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](mailto:rushorders@cmpbooks.com)  
WEB : <http://www.cmpbooks.com>