AN-IAR-Cortex-M3-OS2

Developing µC/OS-II-Based Projects for the Cortex-M3 Using Embedded Workbench

Micrium

INTRODUCTION

AUDIENCE

This application note explains how to use the Embedded Workbench IDE from IAR to quickly and easily put together μC/OS-II-based projects targeting microcontrollers that incorporate an ARM Cortex-M3 CPU. The document's intended audience consists of developers who are using a hardware platform for which there is not an example Embedded Workbench project available from Micriμm. The Download Center on Micriμm's Web site offers a number of projects for a wide range of Cortex-M3-based boards, and developers can use any of these projects as a foundation for application development.

It is recommended that developers who do not find an example project for their platform on the Micriμm Web site call or e-mail for additional information on hardware support, since the actual collection of available projects may be larger than the site indicates. (Contact information is provided at the close of this document.) In cases where no projects exist, developers have the option of implementing a custom project. This document provides the basic information needed to put together such a project for a Cortex-M3-based platform using Embedded Workbench.

REQUIRED SOFTWARE

The source code and IAR project template described in this document are provided in a zip file named *AN-IAR-Cortex-M3-OS2.zip*. This zip is available from Micriμm's Web site, while the Embedded Workbench IDE can be downloaded from IAR's site. The contents of *AN-IAR-Cortex-M3-OS2.zip* were tested using Embedded Workbench v6.60 but may be compatible with other versions of the IDE.

REQUIRED HARDWARE

Since Micriμm offers generic Cortex-M3 kernel ports, the information provided herein applies to all hardware platforms with Cortex-M3-based CPUs.

INSTALLATION

TOOL INSTALLATION

After you've downloaded Embedded Workbench and run its installer, you'll be presented with dialogs guiding you through the installation process. You can simply follow the instructions provided on these dialogs.

SOURCE CODE AND TEMPLATE INSTALLATION

1. As indicated in the previous section of this document, the zip file *AN-IAR-Cortex-M3-OS2.zip* contains the source code and project template that you will need in order to put together your custom project. You should now unzip this file. The contents of the file are organized into a directory structure with a *Micrium* folder at its top level. Micriµm's engineers typically place this folder at the root (creating *C:\Micrium*), but you are free to choose your own path, as long as the underlying directory structure is maintained.

2. To make use of the Embedded Workbench project template provided via *AN-IAR-Cortex-M3-OS2.zip*, you'll need to copy this template into the IDE's own folders. Exactly which files you should copy depends on whether or not you have previously installed project templates from Micriµm. Source and destination folders for both cases are listed below. (The source path in each case is based on the *Micrium* folder mentioned in the previous step.)

   Previous Installation
   Source: *Micrium\Software\uCOS-II\Templates\IAR\Micrium*
   (Copy everything in this folder, including subfolders)

   Destination: *<Embedded Workbench Install Path>\arm\config\template\project\Micrium*

   No Previous Installation
   Source: *Micrium\Software\uCOS-II\Templates\IAR*
   (Copy everything in this folder, including subfolders)

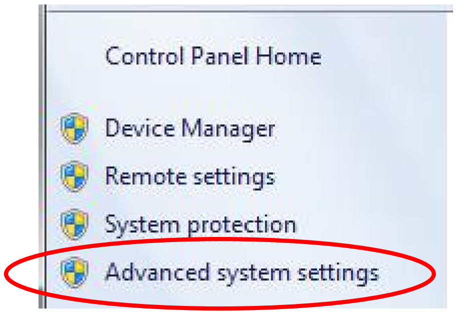
   Destination: *<Embedded Workbench Install Path>\arm\config\template\project*

3. The Embedded Workbench project template uses an environment variable to locate the Micriµm source code on your PC. You should create this variable now. You should name the variable `MICRIUM` and you should give it a value based on the location of your top-level *Micrium* folder. If, for example, you copied *Micrium* into your root folder, then you should give your variable the value *C:\*. The procedure for creating an environment variable in Windows 7 is listed on the next page of this document. If you're using another version of Windows, then you'll need to consult the operating system's documentation to determine the appropriate procedure for creating the variable.

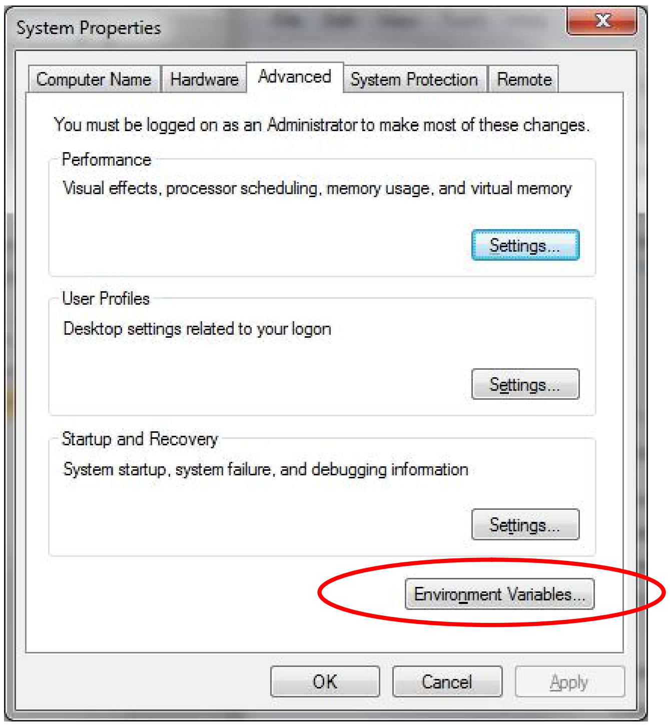
CREATING THE ENVIRONMENT VARIABLE (WINDOWS 7)

1. You can create environment variables through Windows' **Advanced system settings**. To access these settings, you should first open the **Control Panel** and click **System**.

2. The menu shown below should appear on the left-hand side of the **System** window. You should click **Advanced system settings** within this menu.



3. You should now click the **Environment Variables** button shown below.



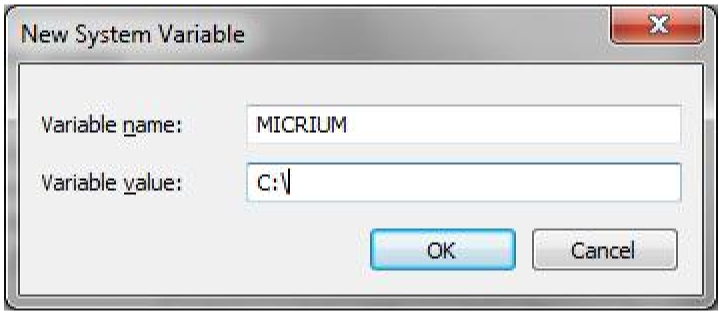
You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your logon

Startup and Recovery
System startup, system failure, and debugging information

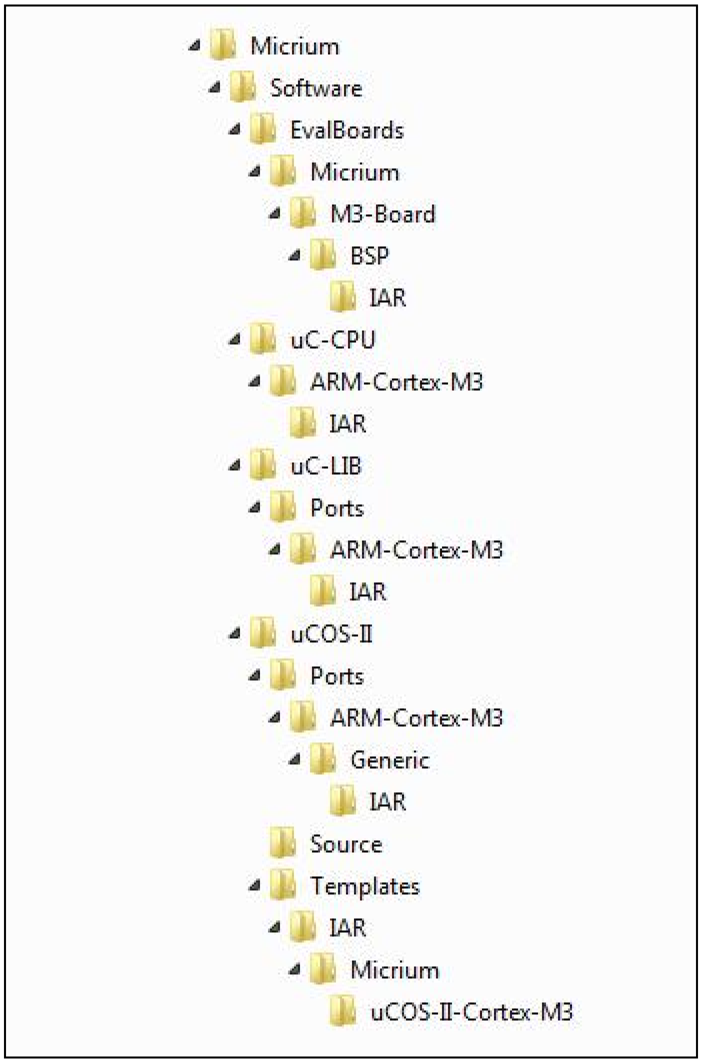
4. In the **Environment Variables** dialog, you should click the **New…** button for **User variables**. You should then enter the name and value of your variable in the **New User Variable** dialog, as shown below. (The value, of course, might be different than what is shown, depending on the location of your *Micrium* folder.) Once you've entered the information, you can click **OK** to close the dialog. You can also close any other open dialogs.


New System Variable

Variable name: MICRIUM

Variable value: C:\

Directory Structure and Source Code Overview



```
Micrium
  Software
    EvalBoards
      Micrium
        M3-Board
          BSP
            IAR
    uC-CPU
      ARM-Cortex-M3
        IAR
    uC-LIB
      Ports
        ARM-Cortex-M3
          IAR
    uCOS-II
      Ports
        ARM-Cortex-M3
          Generic
            IAR
      Source
      Templates
        IAR
          Micrium
            uCOS-II-Cortex-M3
```

The extracted contents of *AN-IAR-Cortex-M3-OS2.zip* comprise the folders shown above. These folders are organized according to Micriμm's standard directory structure. Additional information on the directory structure is available from *AN-2002*, an application note that can be downloaded from the Micriμm Web site.

In putting together your μC/OS-II-based project, you'll manipulate only a few of the many folders yielded by the extraction of *AN-IAR-Cortex-M3-OS2.zip*. Nonetheless, descriptions pertaining to all of the folders that contain source code are provided in this section of the document. The descriptions may help you to better understand the code that your project will ultimately contain.

Micrium\Software\uCOS-II\Source
All of the code making up Micriμm's µC/OS-II kernel can be placed into one of two categories: hardware-independent code and port code. The hardware-independent portion, which is the larger of the two, is where the kernel's API functions are implemented. It is contained in this folder.

Micrium\Software\uCOS-II\Ports\ARM-Cortex-M3\Generic\IAR
A µC/OS-II port is specific to both a CPU architecture and a tool-chain. This folder contains the kernel's port for ARM Cortex-M3 CPUs on the IAR tools.

Micrium\Software\uCOS-II\Templates\IAR\Micrium\uCOS-II-Cortex-M3
This folder contains example application code that the Embedded Workbench template will automatically add to your µC/OS-II project. The code will be copied into your project folder.

Micrium\Software\uC-CPU
µC/CPU is a module that is present in all of Micriμm's example projects. It contains type definitions and utility functions that are needed by Micriμm's other modules, and that are also available for application code's use. Like µC/OS-II, µC/CPU consists of both hardware-independent code and a port. The hardware-independent code is contained in this folder.

Micrium\Software\uC-CPU\ARM-Cortex-M3\IAR
The µC/CPU port for the Cortex-M3 CPU and IAR tool-chain is contained in this folder.

Micrium\Software\uC-LIB
Micriμm's modules do not invoke standard library routines like `memset()` and `strcpy()`. Instead, as part of a strategy to limit the software's tool-chain dependence, they use Micriμm equivalents of the library routines. The Micriμm functions are amassed in a module named µC/LIB, the source code of which is mostly contained in this folder.

Micrium\Software\uC-LIB\Ports\ARM-Cortex-M3\IAR
An optimized, Cortex-M3-specific memory-copying routine is implemented in the single assembly language file contained in this folder. µC/LIB can be configured to use either this optimized routine, or a C version of the function.

Micrium\Software\EvalBoards\Micrium\M3-Board\BSP
A typical µC/OS-II-based project, in addition to requiring port files for µC/CPU and the kernel itself, needs a small board support package (BSP). Example BSP code is contained in this folder. The example code is described in the "Completing the Project" section of this document.

Micrium\Software\EvalBoards\Micrium\M3-Board\BSP\IAR
Most of the example BSP code could be used with any tool-chain. The small portion contained in this subfolder, however, is IAR-specific. This code implements an example interrupt vector table, and in doing so it uses directives that are unique to Embedded Workbench.

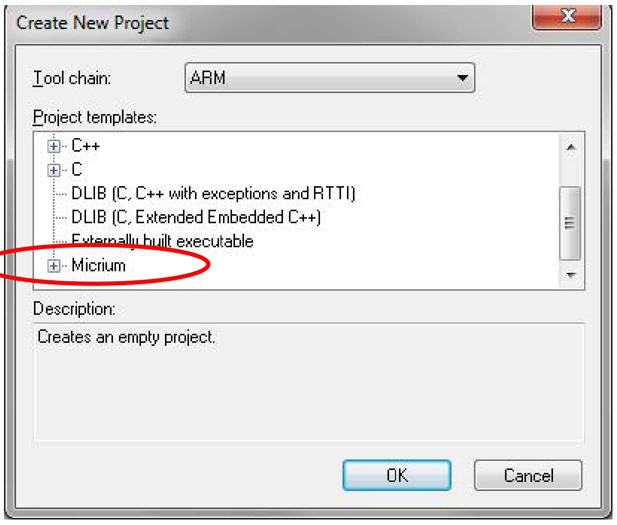
USING PROJECT TEMPLATES

The project template contained in *AN-IAR-Cortex-M3-OS2.zip* helps to accelerate the process of putting together a µC/OS-II-based project in Embedded Workbench. All of Micriµm's example µC/OS-II-based projects for this IDE have a similar structure; they are composed of a number of source files (from the kernel and other modules) that are organized into groups. When you use the project template according to the below instructions, Embedded Workbench will automatically create the groups for your project, and will populate most of these groups with the appropriate files.

1. Every project in Embedded Workbench is part of a workspace. When you start the IDE, an empty workspace is opened. If you would prefer for your project to be part of an existing workspace, then you should open that workspace now. Otherwise, you can simply use the default, empty workspace.

2. You should now initiate the process of adding a new project to your workspace by selecting **Create New Project…** from Embedded Workbench's **Project** menu.

3. Amongst the **Project templates** listed in the **Create New Project** dialog, there should be a **Micrium** entry, as shown below. You should expand this entry, select **uC/OS-II Cortex-M3**, and then click the **OK** button.

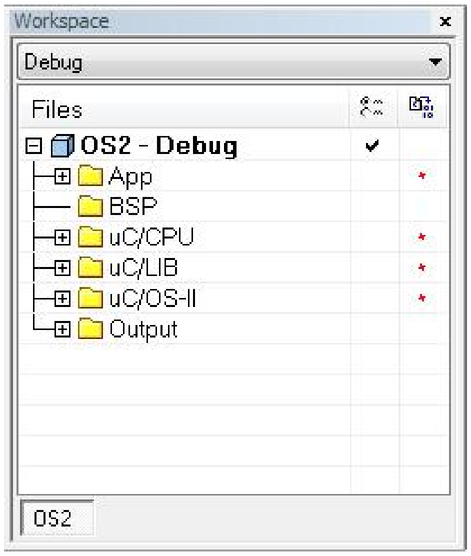

Create New Project
Tool chain:
ARM
Project templates:
C++
C
DLIB (C, C++ with exceptions and RTTI)
DLIB (C, Extended Embedded C++)
Externally built executable
Micrium
Description:
Creates an empty project.
OK
Cancel

4. Before attempting to create your new project, Embedded Workbench should present you with a dialog for selecting the name and location of the corresponding project file. Micriµm typically locates these files within *Micrium\Software\EvalBoards*, since each project targets a particular evaluation board. The standard that Micriµm follows calls for a number of folders in between the project and *EvalBoards*, with each folder named according to the rules shown below.

   EvalBoards\<Name of Board Manufacturer>\<Board Name>\<Example Name>\<IDE Name>

According to this standard, an Embedded Workbench project named *Example-1* and targeting company ABC's XYZ board would be located in *Micrium\Software\EvalBoards\ABC\XYZ\Example-1\IAR*. It's recommended that you rely on a similar structure for your own projects. The project template, however, should function correctly regardless of where you choose to locate your files.

5. Once you've chosen a name and location for your project file, the project should appear in Embedded Workbench's **Workspace** window, as shown below. Although the project should already contain all of the source code for µC/OS-II, µC/CPU, and µC/LIB, there are a few additions that you will need to make before attempting to build the project or download its code to an actual board. These additions are discussed in the next section.



COMPLETING THE PROJECT

If you followed the steps listed in the previous section, then you should now have a project that contains all of the hardware-independent code of µC/OS-II, µC/CPU, and µC/LIB, as well as the architecture-specific code required by these modules. Your project, though, still must be adapted to your particular Cortex-M3-based device. To meet this objective, you'll just need to write a simple BSP and make a few adjustments to the project's settings.

BSP

The code contained in a µC/OS-II port enables the kernel to manipulate the CPU registers of the Cortex-M3 core, but this code does not provide access to peripheral devices. Fortunately, µC/OS-II itself requires very little from the devices surrounding the CPU core. In general, the kernel needs startup code (a requirement that can often be satisfied with generic startup code from a tool or chip vendor), and a means of processing periodic interrupts, typically produced by a timer.

On Cortex-M3-based platforms, the kernel actually has fewer device-specific requirements than on other architectures, because it can use the CPU's core timer as its periodic interrupt source. This timer can be assumed to exist on every implementation of the Cortex-M3 core, so the majority of the code needed to utilize it is provided in the kernel port. A BSP for a Cortex-M3 platform, then, may consist of little more than startup code and a few interrupt-related routines.

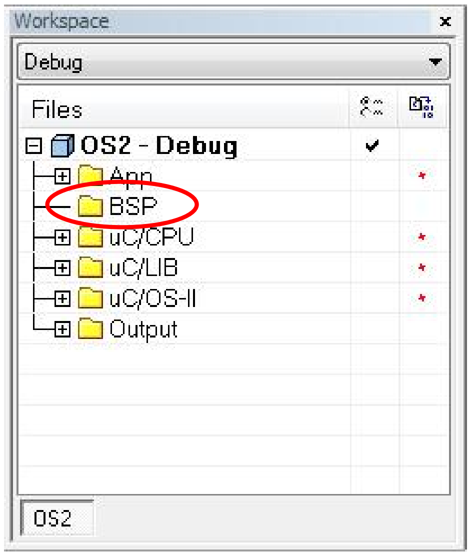
The source files delivered in the example *BSP* folder that was described in the "Directory Structure and Source Code Overview" section of this document provide a starting point that you can use to quickly implement a simple BSP for your hardware. The contents of the example BSP include empty functions for which you'll need to provide device-specific declarations, as well as fully implemented routines suitable for use with most or all Cortex-M3-based MCUs. The information that follows is intended to be a brief guide through the example BSP, explaining which functions to modify and how to add your finished BSP to your project.

DIRECTORY STRUCTURE

Rather than overwrite the example BSP, you should copy these files into your own folder where you can then begin modifying and updating the code. It is recommended that you organize the copied BSP files according to the directory structure briefly described in step 4 of this document's previous section. For the example "XYZ" board given in that section, your BSP would be located in *Micrium\Software\EvalBoards\ABC\XYZ\BSP*, with your tool-chain-specific file(s) in the subfolder *Micrium\Software\EvalBoards\ABC\XYZ\BSP\IAR*. The IAR project template, though, is not hardcoded with the location of any BSP files, so you're ultimately free to place your files in any location.

ADDING FILES TO A PROJECT

Regardless of where you choose to locate the copied BSP files, you'll need to manually add these files to your project. The procedure for doing so is simple: To add each BSP file, you should simply right-click on the **BSP** group shown in the screenshot on the next page of this document and select **Add>Add Files…** from the ensuing menu. You can then navigate to your files and add them. For any new header files, you'll likely need to update your project's options with include paths, as described in the "Project Options" section of this document.



FILE DESCRIPTIONS

Micrium\Software\EvalBoards\Micrium\M3-Board\BSP\bsp.c

`BSP_Init()` – This routine is used in µC/OS-II-based projects to initialize the peripherals required by the kernel and application code. As indicated elsewhere in this document, the peripheral needs of the kernel are minimal. What is accomplished in `BSP_Init()` will mostly be decided by your application's needs, then. Oftentimes, this routine is used to initialize, amongst other peripherals, clock-related hardware, since the default settings for PLLs and similar devices typically do not facilitate the best possible performance.

Regardless of the hardware that you choose to initialize in `BSP_Init()`, you should keep in mind that this function does not technically constitute startup code. It is not called, in other words, prior to `main()`; it's actually invoked by the first task, `AppTaskStart()`, that the example application code provided in *AN-IAR-Cortex-M3-OS2.zip* creates. Since µC/OS-II is typically able to rely on the standard startup files provided with Embedded Workbench, there is not any example startup code in *AN-IAR-Cortex-M3-OS2.zip*. If your application requires custom startup routines, you can consult the Embedded Workbench documentation for an overview of how such code must be structured in the IDE.

`BSP_CPU_ClkFreq()` – The purpose of this function is to report the clock speed (in Hz) of the CPU. The kernel needs the clock speed in order to properly establish periodic interrupts from the core timer at a rate specified by application code. In the example application, these interrupts are set up by `AppTaskStart()`, via the code shown below.

```
hclk_freq = BSP_CPU_ClkFreq();
cnts  = hclk_freq / (CPU_INT32U)OS_TICKS_PER_SEC;
OS_CPU_SysTickInit(cnts);
```

Application-specified tick frequency

BSP_LED_Init() – µC/OS-II does not require any LED-related routines. However, blinking an LED is a common means of demonstrating that the kernel is running on a particular board. Thus, most Micriµm BSPs include a few functions for manipulating LEDs. BSP_LED_Init(), which is expected to be invoked by BSP_Init(), is typically used to lay the ground for a BSP's other LED routines by initializing any I/O ports that those routines access.

BSP_LED_On() – This routine provides application code with a means of turning on LEDs. The 8-bit value passed through the function's single argument specifies which particular LED should be asserted. A value of zero is normally taken to mean that all available LEDs should be turned on, while each non-zero value is associated with a single LED. (A value of 1 might be used to turn on an LED labeled **LED1**, for example.)

BSP_LED_Off() – Application code can use this routine to turn off LEDs. The convention for the function's argument is usually the same as that followed by BSP_LED_On(): A zero signifies all LEDs, while a non-zero value identifies a single LED.

BSP_LED_Toggle() – This function is provided as a means of toggling one or all of a board's LEDs. It's argument, like those for BSP_LED_On() and BSP_LED_Off(), identifies the LED(s).

CPU_TS_TmrInit() – Although this function is contained in the BSP, it is often implemented in a somewhat board-independent way in Cortex-M3 projects. The objective of the function is to initialize a free-running timer used by the µC/CPU module for making performance measurements. In the provided implementation of the function, the Cycle Count Register of the Cortex-M3's Data Watchpoint and Trace Unit serves as the timer. Since this facility exists on many Cortex-M3-based MCUs, there's a good chance that you won't need to modify the example code. Changes will only be necessary if your MCU's documentation indicates that your device lacks the Data Watchpoint and Trace Unit, or that the unit is present but that it does not contain the Cycle Count Register.

CPU_TS_TmrRd() – This function enables µC/CPU to read the timer initialized by CPU_TS_TmrInit(). Like the initialization function, the read function has already been implemented for you, and the provided code is suitable for many Cortex-M3-based devices.

Micrium\Software\EvalBoards\Micrium\M3-Board\BSP\bsp.h

The provided implementation of this header file contains several #include statements, along with prototypes of the functions declared in *bsp.c*. You can augment the file according to the needs of both your application and the other parts of your BSP. If, for example, the functions that you implement in *bsp.c* make use of register definitions provided by your chip vendor in a header file, you can add a #include statement for that header file to *bsp.h*.

Micrium\Software\EvalBoards\Micrium\M3-Board\BSP\bsp_int.c

This file contains functions that provide application code with a variety of interrupt-related capabilities. Since all Cortex-M3 CPUs feature similar interrupt controllers, you'll likely be able to use the provided implementations of many of these functions. The following summaries of the functions include explanations of when and why modifications might be necessary.

BSP_IntClr() – This routine is typically provided in Micriμm BSPs as a mechanism by which application code can clear interrupt flags. On the Cortex-M3's interrupt controller, though, interrupts don't need to be cleared, and the function can be left empty.

BSP_IntDis() – This routine manipulates the Cortex-M3's interrupt controller to disable interrupts from a particular peripheral. The routine's single argument designates that peripheral. An implementation suitable for all Cortex-M3-based MCUs has been provided, so you won't need to rewrite the function. It is suggested, however, that you redefine the ID constants (BSP_INT_ID_XXX) provided in *bsp_int.h* to match the interrupts that actually exist in your system. Most of these constants will be used by your application code to set up interrupts, so they can be named according to your own preferences. The only exception is BSP_INT_ID_MAX. This constant is expected to indicate the total number of interrupt sources in your system, and it determines the size of BSP_IntVectTbl[], an array used by two of the functions in *bsp_int.c*.

BSP_IntDisAll() – The objective of this function is to disable all interrupts. You won't need to make any changes to the function's code; the provided implementation is suitable for any Cortex-M3.

BSP_IntEn() – Using this function, application code can enable interrupts from a designated peripheral. In the provided implementation of the function, the enable operation involves the interrupt controller, much like the disable operation performed by BSP_IntDis(). These two routines do not actually access the peripheral devices from which the interrupts originate, and, accordingly, neither of them will need to be modified for your hardware.

BSP_IntVectSet() – This function allows application code to associate a handler, or interrupt service routine (ISR), with a particular interrupt source. The first argument accepted by the function is an ID, like that used in BSP_IntDis() and BSP_IntEn(), while the second argument is an ISR function pointer. It is important to note that the provided implementation of BSP_IntVectSet() does not actually manipulate any interrupt controller registers. This implementation, which should be suitable for any Cortex-M3-based MCU, simply places the function pointer provided through its second argument into the array BSP_IntVectTbl[].

BSP_IntPrioSet() – A priority can be associated with an interrupt source using this function. As in most of the other routines declared in *bsp_int.c*, the interrupt source is specified via an ID. The ID is the function's first argument, and the priority for the corresponding interrupt source is the second. You won't need to make any changes to the provided implementation in order for the function to run properly on your hardware.

BSP_IntInit() – The objective of this initialization routine is to fill BSP_IntVectTbl[] with references to a dummy handler, BSP_IntHandlerDummy(). To accomplish this objective, the routine simply makes repeated calls to BSP_IntVectSet(). The provided implementation of the function is suitable for any Cortex-M3-based MCU.

BSP_IntHandlerXXX() – Each of these functions is intended to be an ISR for one of the interrupt sources in your system. Since the set of interrupting devices is not the same on every Cortex-M3-based MCU, you'll likely need to modify the provided implementations of the handlers and, possibly, write additional functions. In any new code that you write, however, you should follow the simple format of the existing handlers, each of which consists of a single call to `BSP_IntHandler()`. If your system incorporates, for example, a USB interrupt that is not represented by the handlers provided in *bsp_int.c*, then you should write a function that calls `BSP_IntHandler()` with the ID of the USB interrupt. (The preceding description does not apply to `BSP_IntHandlerDummy()`, which is covered below.)

`BSP_IntHandler()` – This function, since it is supposed to be invoked by each of the above-mentioned interrupt handlers, should ultimately be executed in response to any interrupt, regardless of source. The job of the function is to use the interrupt ID provided as its single argument to retrieve an application-level handler for the corresponding interrupt from `BSP_IntVectTbl[]`. The function includes calls to the `OSIntEnter()` and `OSIntExit()` routines that are necessary in µC/OS-II-based interrupt handlers, and it can be used without modification on any Cortex-M3-based MCU.

`BSP_IntHandlerDummy()` – A reference to this function is placed in every entry of `BSP_IntVectTbl[]` by `BSP_IntInit()`. Each reference is overwritten only when application code calls `BSP_IntVectSet()` to specify a handler for the corresponding interrupt. `BSP_IntHandlerDummy()`, then, is what runs when an unexpected interrupt occurs. The provided implementation of this function consists only of an empty loop, but you are free to provide a more-complex implementation to meet the error-handling needs of your project.

Micrium\Software\EvalBoards\Micrium\M3-Board\BSP\bsp_int.h

Constants and function prototypes used in *bsp_int.c* are contained in the provided implementation of this header file. You should make sure that the interrupt ID constants (`BSP_INT_ID_XXX`) contained in the file match the actual interrupt sources in your system, and that function prototypes are provided at the bottom of the file for all of the interrupt handlers in *bsp_int.c*.

Micrium\Software\EvalBoards\Micrium\M3-Board\BSP\IAR\bsp_vect.c

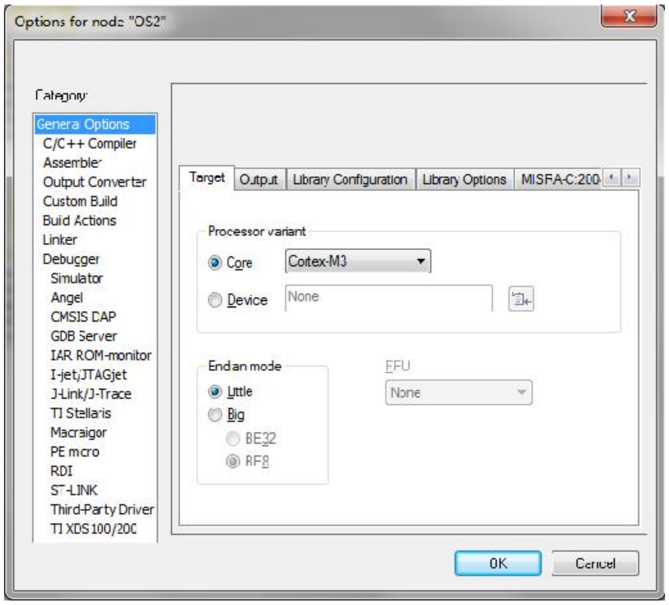
The centerpiece of this file is the declaration of `__vector_table[]`, the array that serves as the interrupt vector table for your project. Also contained in the file are declarations of handlers for a number of different types of events and exceptions. You'll likely need to adjust the vector table to reflect the interrupt sources existing in your system. You should, on the other hand, be able to use the handler functions without modification on any Cortex-M3-based MCU.

`__vector_table[]` – When you load your project's code onto your board, Embedded Workbench will ensure that this array resides at the interrupt vector location. Thus, each time an interrupt occurs, one of the functions referenced in the array will be executed. The first fifteen functions referenced by the array, beginning with `__iar_program_start()` and ending with `OS_CPU_SysTickHandler()`, should be the same for all Cortex-M3-based MCU's. The remaining functions correspond to the `BSP_IntHandlerXXX()` ISRs defined in *bsp_int.c*, and, accordingly, they will vary across devices. You should make sure that all of your `BSP_IntHandlerXXX()` functions are represented in the vector table, and that each one of these functions resides at the proper location. If, for example, vector 36 is set aside for A-D interrupts on your MCU, then you should place a reference to your A-D handler at entry 36 of `__vector_table[]`.

BSP_XXX_ISR() – These routines are a means of handling different events and exceptions. Each of the provided implementations consists of just an empty loop, with the exception of the reset handler, BSP_Reset_ISR(), which encompasses code for enabling floating-point hardware. You're free to modify any of the functions to suit the requirements of your application. However, you should be able to build and run your project's code using the provided versions.

Project Options

In Embedded Workbench, there are numerous options that can be set for every project. You can access options for your µC/OS-II-based by project by right-clicking the project's name in the IDE's Workspace window and selecting **Options…** from the ensuing menu. You should then be presented with the below dialog, the left-hand side of which lists several categories of options.

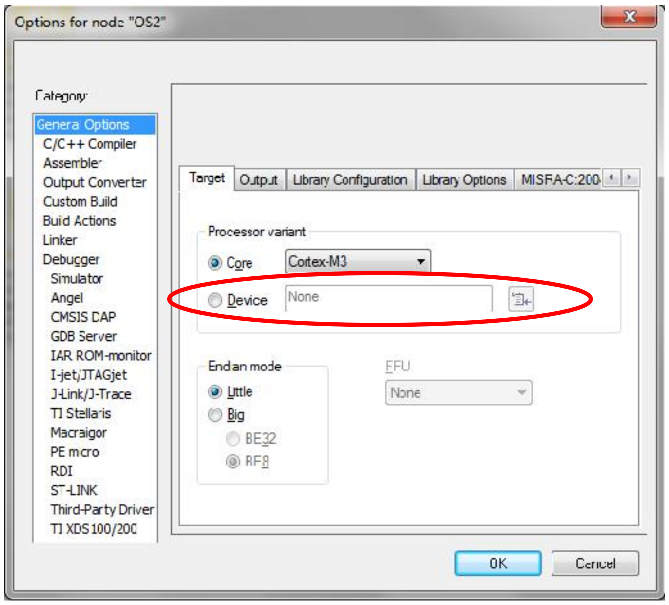


Processor variant: Core: Cortex-M3
Device: None
Endian mode: Little
FPU: None

You'll likely need to make only a few changes within the **Options** dialog in order to prepare your µC/OS-II project for building and downloading to your board. Below are descriptions of these changes, organized according to category.

General Options

Each version of Embedded Workbench supports a multitude of different devices. You should select your device on the Target page within General Options, as shown in the screenshot on the next page.

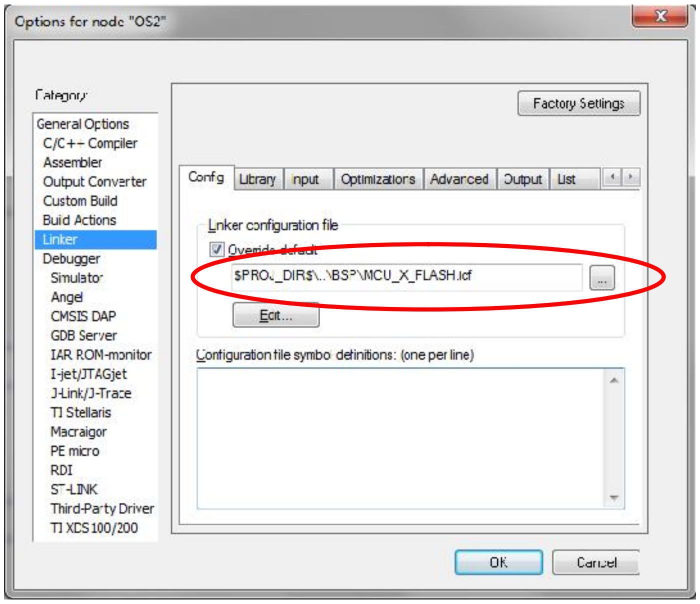


C/C++ Compiler

How you should set the compiler options grouped into this category mostly depends on the needs of your application. There is, though, one page within **C/C++ Compiler** that you'll likely need to update regardless of your application's requirements: the **Preprocessor** page. The directories where the compiler will search for your project's include paths are specified here. Directories should already be listed for all of the files added to the project automatically by the template, but you'll need to provide directories for the BSP files and any other files that you manually added to the project.

Linker

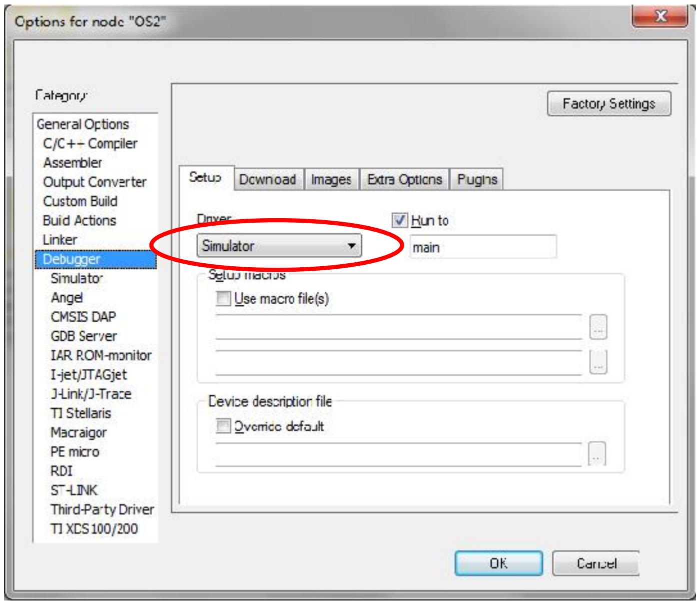
Within Embedded Workbench, there is a default linker command file for Cortex-M3-based projects. However, the locations of RAM and ROM differ widely across MCUs, so most of Micriμm's projects use a device-specific file in place of the default. Device-specific files for a number of MCUs can actually be found with the example projects that accompany Embedded Workbench, and such files are also typically available from hardware vendors. If you have a device-specific linker command file for your project, you can specify its location to Embedded Workbench via the **Config** page within the **Linker** category, as shown in the screenshot on the next page.



Linker configuration file
☑ Override default
$PROJ_DIR$\..\BSP\MCU_X_FLASH.icf
Configuration file symbol definitions: (one per line)

DEBUGGER

This category allows you to set up a project for use with your debugger. At a minimum, you'll need to specify your debugger on the category's **Setup** page, as indicated in the below screenshot. You should consult the documentation for both Embedded Workbench and your debugger to determine what other options you might need to set in this category.

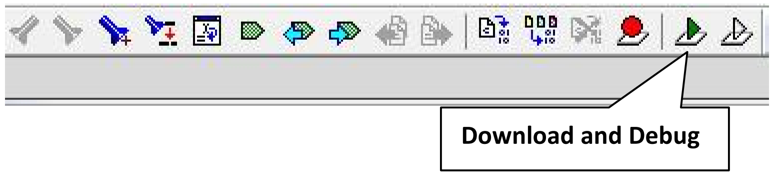


Driver: Simulator
☑ Run to: main
Setup macros
☐ Use macro file(s)
Device description file
☐ Override default

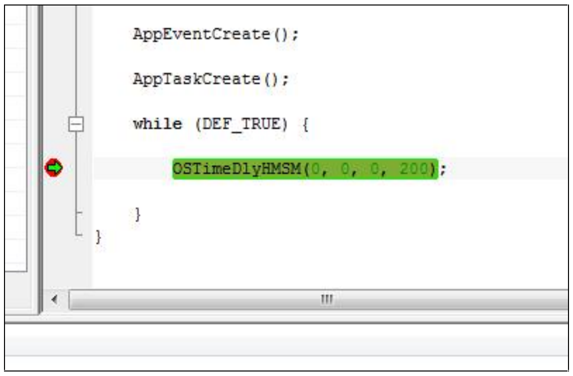
BUILDING AND RUNNING THE PROJECT

Once you've completed your BSP, you can attempt to build your project by right-clicking the project name in Embedded Workbench's **Workspace** window and selecting **Rebuild All** from the ensuing menu. If you've not yet saved the workspace containing the project, then Embedded Workbench will prompt you to do so. Although you can choose any location for the workspace, Micriμm recommends placing this file in the same folder as the project file.

You can use the **Download and Debug** button shown below to load your project to your board after a successful build. When you subsequently run the code (via the **Go** button on Embedded Workbench's **Debug** toolbar), you will not actually see much evidence that μC/OS-II is multitasking unless you've modified the example application. The lack of activity is due to the simplicity of the example code; this code performs few I/O-related operations, since it is intended to support a wide variety of platforms. If you'd like for your board to provide you with a visual indication that the kernel is running, then you'll need to update the example with appropriate BSP calls (to LED routines, for example).



When using the example without modifications, you'll need to rely on breakpoints to determine that the code is running correctly. In Embedded Workbench, you can set a breakpoint on a particular line of code by right-clicking that line in the text editor and selecting **Toggle Breakpoint (Code)** from the menu that then appears. After you've loaded the example, you should set a breakpoint on line 139 of the file *app.c*. This line of code corresponds to a delay function call made by the single μC/OS-II task that the example creates, `AppTaskStart()`. When you run your code, the debugger should almost immediately stop at the breakpoint, as shown below. Since the function call corresponding to the breakpoint is made within a loop (meaning that the task delays itself repeatedly), you should continue to stop at the breakpoint each time you click the **Go** button.



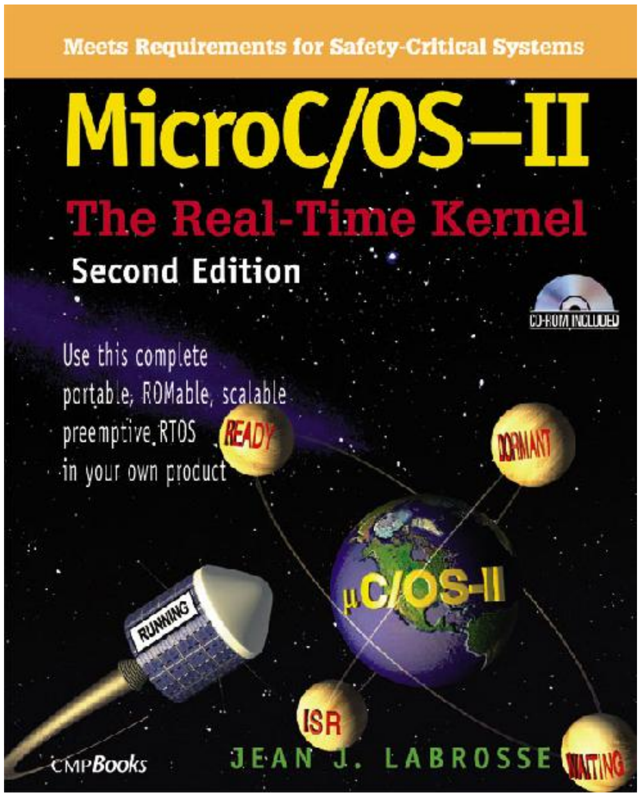
```
    AppEventCreate();

    AppTaskCreate();

    while (DEF_TRUE) {

        OSTimeDlyHMSM(0, 0, 0, 200);

    }
}
```

If the breakpoint does not perform as expected in your code, then you should begin looking over your BSP for potential errors. The BSP is the most likely source of problems because most of the other software in the project has been thoroughly tested on the Cortex-M3. You can consult Micriµm's Web site for BSP example code beyond that provided with the project template in *AN-IAR-Cortex-M3-OS2.zip*. The site's Download Center offers access to numerous example projects, each of which incorporates at least the minimal BSP needed to run the kernel.

ADDITIONAL INFORMATION

THE μC/OS-II BOOK

The preceding pages of this document focus primarily on project creation and BSP development; they are not intended to serve as a reference source for μC/OS-II. If you'd like detailed information on the kernel and the services it provides, then you should consult the highly popular μC/OS-II book. Written by Micriμm Founder and CEO Jean Labrosse, who also wrote the kernel itself, this book explains practically every line of code contained in μC/OS-II. It is available as a hardcover from Amazon and other retailers, and can also be downloaded as a free PDF from the Micriμm Web site.


Meets Requirements for Safety-Critical Systems
MicroC/OS–II
The Real-Time Kernel
Second Edition
CD-ROM INCLUDED
Use this complete portable, ROMable, scalable preemptive RTOS in your own product
READY
DORMANT
μC/OS-II
RUNNING
ISR
WAITING
CMPBooks
JEAN J. LABROSSE

LICENSING

µC/OS-II is a source-available real-time kernel; it is not open source. Under the source-available model, which Micriµm pioneered, the kernel's full source code can be evaluated at no cost. This code can also be used free of charge in academic projects. Developers planning to use the code to develop a product, however, must purchase a license. Additional licensing information can be obtained from Micriµm; contact information is provided below.

MICRIUM CONTACT INFORMATION

Phone: +1 954 217 2036
Fax: +1 954 217 2037
E-Mail: sales@micrium.com
URL: http://www.micrium.com