Micriμm

Empowering Embedded Systems

μC/OS-II

and

MIPS32 4K Processor Cores

Application Note

AN-Microchip-PIC32

www.Micrium.com

Document Conventions

Numbers and Number Bases

- Hexadecimal numbers are preceded by "0x" and displayed in a mono-spaced 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 mono-spaced 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 mono-spaced font.
- Code excerpts, variable names, and function names are displayed in a mono-spaced font. Functions names are always followed by empty parentheses (e.g., `OSStart()`). 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 emphasize key points.
- Any other text is displayed in a sans-serif font.

Table of Contents

1.00 Introduction

Micriμm's μC/OS-II is an efficient and reliable real-time kernel, offering embedded applications in need of an operating system's services a powerful solution with a small memory footprint. The services that μC/OS-II provides can be utilized on a large number of hardware platforms, because the majority of the operating system is, essentially, processor-independent, being written in highly-portable ANSI C. Only a small group of files, known collectively as a port, must be compiled alongside μC/OS-II's processor-independent code in order to render the operating system compatible with a given processor architecture. One such port, developed for MIPS32 4K processor cores, is detailed in this application note.

Much like other μC/OS-II ports developed by Micriμm, the port described in this document is intended to be applicable to numerous devices. The port's files follow the conventions of the MIPS32 architecture (as it is described in the specifications cited in the References section at the end of this application note) and are, as much as possible, free of code reliant on a particular piece of hardware. However, because of the latitude that the MIPS32 architecture offers its implementers, particularly with respect to privileged resources, the port is presented as a means of running μC/OS-II on 4K cores, not all MIPS32 processors. In fact, the many variations that are allowed by the 4K cores' specification (which is also cited in the References section) further limit the range of devices that are fully compatible with the port, introducing the possibility that the port's files would require modifications in order to be used with a particular 4K core.

Fortunately, any modifications that might be needed to allow the μC/OS-II MIPS32 4K port to be used with a given implementation of a 4K core, or any other MIPS32 processor for that matter, would likely be minor in scale, and would apply to sections of code thoroughly explained in this document. These explanations, in addition to serving a benefit when modifications of the port are required, support this application note's principle goal of conveying the information needed to begin developing μC/OS-II-based applications for 4K cores. In subsequent sections, this information is organized according to the components of such applications, as shown in Figure 1-1. Descriptions are provided for each of these components, with the exception of μC/OS-II's processor-independent files, which are thoroughly described in **MicroC/OS-II, The Real-Time Kernel**, one of the books cited with the aforementioned MIPS specifications in the References section.

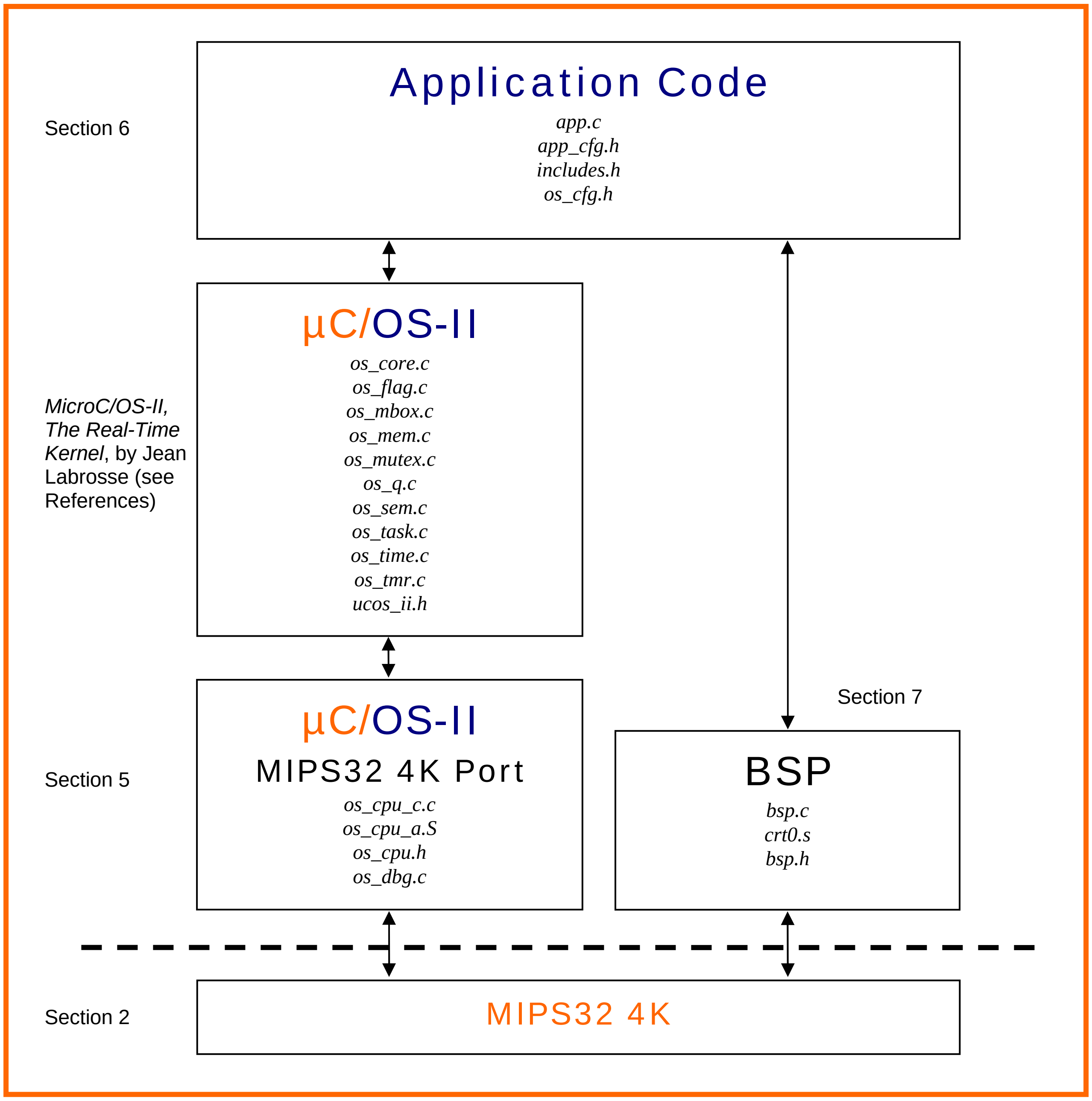


Figure 1-1, A Typical μC/OS-II-Based Application

Although μC/OS-II's processor-independent files are not described in this application note, these files are, of course, intended to be used with the port code that serves as the focus of the explanations contained herein. Therefore, *AN-Microchip-PIC32.zip*, the zip file in which this application note is normally distributed, furnishes all of the components found in a typical μC/OS-II-based application, including the operating system's processor-independent files. With the contents of this zip file, the development of realistic applications running on actual hardware is possible, making the evaluation of μC/OS-II a straightforward and worthwhile process. However, μC/OS-II is not free software, and continued use of the operating system, beyond the 45-day evaluation terms under which its source files are provided, requires appropriate licensing. Questions regarding μC/OS-II's licensing requirements should be directed to Micriμm, using the contact information provided at this document's conclusion.

2.00 The MIPS32 Architecture

The MIPS32 architecture, with its roots in previous versions of MIPS, facilitates the creation of robust hardware platforms that leverage an established instruction set. Lending potential economic benefits to such platforms, the architecture promotes a balance between cost and performance, two design characteristics that are often of equal importance in embedded applications. As might be imagined, such applications, which are served by an increasingly significant portion of MIPS implementations, heavily influenced the development of the MIPS32 architecture.

The MIPS32 4K cores provide an example of the MIPS32 architecture's suitability for embedded applications. Available in multiple configurations, all of which were designed for embedded use, these simple, yet capable, cores are devoid of superfluous features that would unnecessarily increase costs. Accordingly, memory management units, interrupt-related hardware, and other resources that each core makes available to system software are efficient and easy to use.

2.01 Key Aspects of MIPS32 4K Cores

As Figure 2-1 indicates, software running on a MIPS32 4K core has access to 32 general-purpose registers (GPRs), in addition to two special-purpose registers. The latter two registers, HI and LO, which together form a 64-bit location that is reserved for the results of multiply and divide instructions, are accompanied by a fixture of nearly all processor architectures: a program counter. As in other architectures, this resource indicates the address of the instruction being executed by the processor at any given time. However, unlike the program counters offered by many other devices, those in MIPS32 4K cores are not visible to software and, accordingly, must be accessed indirectly.

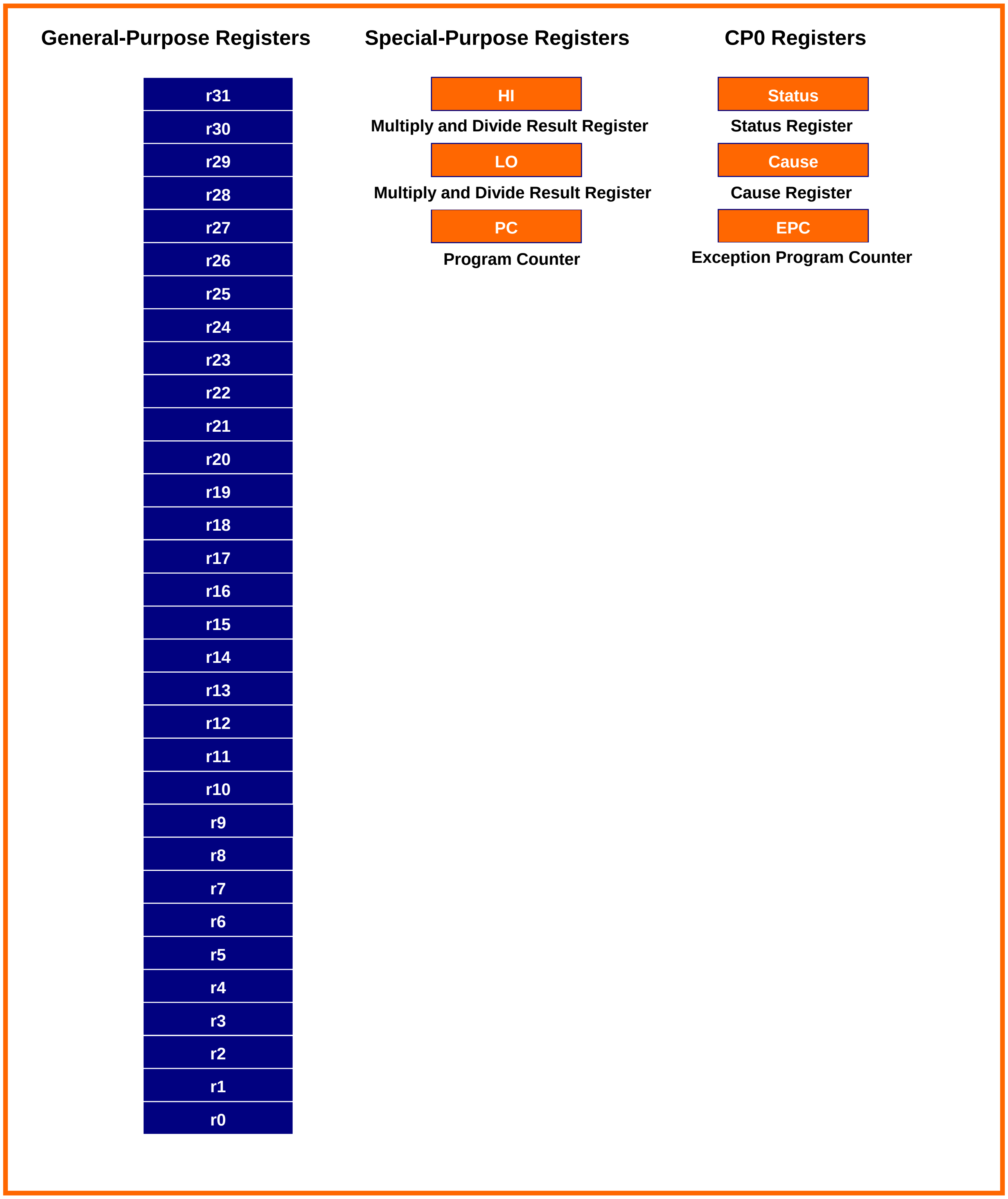


Figure 2-1, MIPS32 4K Registers

Just as the program counter is not a standard register and cannot be accessed through arithmetic or load/store instructions, the registers that are used to configure MIPS32 4K cores are not entirely conventional. These registers are part of the System Control Coprocessor (CP0), which governs many of the cores' resources, including those associated with memory management, debugging, and interrupt processing. Since resources of this sort are not implemented identically on each 4K core, the interface that CP0 presents is likewise not entirely consistent across cores. Thus, as much as possible, the µC/OS-II MIPS32 4K port's use of this coprocessor has been minimized, resulting in code that only accesses the registers shown in Figure 2-1.

In adherence to the policy of using CP0 registers sparingly, the µC/OS-II port for MIPS32 4K cores does not contain code that is specific to the Memory Management Units (MMUs) normally provided by the cores. These MMUs, which can be classified according to their use of either TLBs or a fixed mapping scheme, translate the regions of the address space that Figure 2-2 designates as "mapped", while ignoring the remaining, "unmapped" addresses. Thus, without the addition of translation tables and other provisions typically needed to support memory management, the µC/OS-II MIPS32 4K port should only be used with application code that resides in "unmapped" memory. As Figure 2-2 notes, these portions of the memory space are only accessible when the core is running in either kernel or debug mode, and, conveniently, the former is assumed to preside while the µC/OS-II MIPS32 4K port executes.

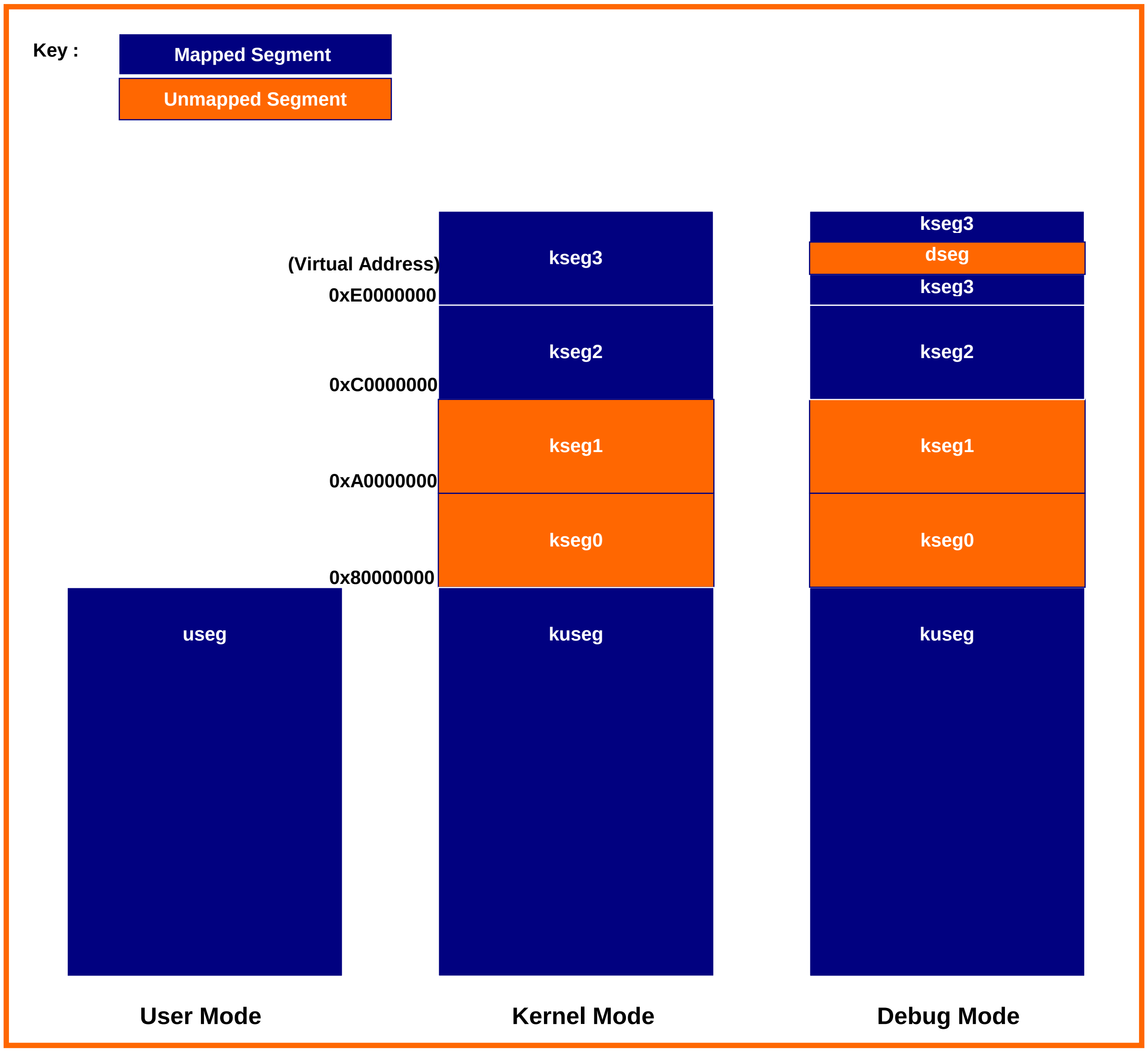


Figure 2-2, The MIPS32 4K Address Space

The MIPS32 4K cores' different operating modes, which, along with kernel mode and debug mode, include user mode, each afford software different capabilities. User mode is the most restrictive of the three modes and is associated with the reduced address space illustrated in Figure 2-2, while kernel mode and debug mode make the entire MIPS32 address space available to software and offer unfettered access to processor resources. Since debug mode, as its name implies, is provided mainly for the benefit of debugging software, and user mode imposes significant limitations, the μC/OS-II MIPS32 4K port utilizes kernel mode exclusively. This arrangement facilitates application code that, when disabling interrupts and performing other simple tasks, is not dependent on operating system routines that could unnecessarily complicate such operations.

With the μC/OS-II MIPS32 4K port affording unrestricted access to processor resources, code lying outside of the port is capable of taking on responsibilities normally associated with system software. Thus, even though the port does not contain any code that is expected to initialize interrupt-related hardware, applications that incorporate the operating system can easily accommodate any one of the three different interrupt modes allowed by the MIPS architecture. This flexibility is possible not only because interrupt-related initializations are delegated to application code, but also because of the simplicity of the sole interrupt handler provided with the µC/OS-II MIPS32 4K port. This handler, `CoreTimerIntHandler()`, which is described alongside the other components of the port in subsequent sections of this document, has only been tested under one of the interrupt modes, external interrupt controller mode, but the routine's code could easily be adapted to the demands of one of the other two modes, either vectored interrupt mode or compatibility mode.

3.00 Using the µC/OS-II MIPS32 4K Port

As the illustration provided in this document's introduction indicates, if you have chosen to develop an application around µC/OS-II, you will need three distinct groups of files, in addition to your application code. The first of these groups, starting with the upper layers of a typical application and moving down, towards hardware, consists of µC/OS-II's processor-independent code, in which the majority of the numerous functions needed to implement the operating system's API are declared. These files are written entirely in ANSI C, and, as a result of this language's inherent portability, they must be supplemented by a port, a group of files containing the type of code that is described in this application note. Although a port is essential for manipulating processor registers and other architecture-specific features, it cannot afford the operating system or your application access to peripheral devices defined outside of the appropriate architecture, so yet another group of files, which together are known as a board support package (BSP), is required.

Because of its relative proximity to hardware, the BSP is the most likely, of the groups of files common to µC/OS-II-based applications, to need significant modifications in order to facilitate a new hardware platform. For this reason, Micriµm's Web site offers BSP code for many of the evaluation boards commonly used to run µC/OS-II, and each of these BSPs is accompanied by an application note that imparts useful information about adding the code to your application. Running the operating system on a given board, then, normally involves combining this BSP code with appropriate port files, which are usually described in another application note, such as this one, that offers guidelines for using the operating system with a particular architecture, or a family of processors, rather than a single evaluation board.

Thus, if you have decided to use µC/OS-II with a MIPS32 4K core, you should first check Micriµm's Web site for an application note that summarizes a BSP suitable for your hardware. If an appropriate document can be located, you should be able to immediately begin developing applications for your board, using the provided BSP code in conjunction with the µC/OS-II MIPS32 4K port detailed in this document. Otherwise, if no BSP code is available, or if your project is dependent on custom hardware, you will need to develop your own BSP, and you can use the example files furnished with this document as a template.

Like the example BSP files, the port files described herein constitute an optimal starting point for writing your own files, and, indeed, changes to the original files may be necessary if your MIPS device is not based on a 4K core. Fortunately, since the port files were designed to be amenable to different MIPS implementations, few changes should be required, unless the relevant characteristics of your device differ drastically from those emphasized in this document's description of the µC/OS-II MIPS32 4K port files. In fact, using the information provided by this application note, along with the files to which this information applies, minimum effort should be needed to run a simple, µC/OS-II-based application on your MIPS processor.

4.00 Directories and Files

AN-Microchip-PIC32.zip, a zip file that is available from Micriμm's Web site, offers access to this application note and to all of the associated source files. The contents of this zip file are organized according to conventions devised by Micriμm, resulting in a directory structure in which source files occupy the folders described below. Although these files could conceivably be copied to any location on your PC, you will likely find that maintaining the directory structure encouraged by *AN-Microchip-PIC32.zip* ultimately simplifies the task of updating your software.

\Micrium\Software\uCOS-II\Source

μC/OS-II's processor-independent code, a portion of which can be found in each of the files listed below, resides in this folder. This code, which must be licensed if it is used commercially, is thoroughly described in **MicroC/OS-II, The Real-Time Kernel**, a book that, in addition to its descriptions of μC/OS-II, offers a general introduction to real-time operating systems. This book is listed in the References section at the end of this document.

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

\Micrium\Software\uCOS-II\Ports\MIPS32-4K\Vectored-Interrupt\MPLAB-PIC32-GCC

The μC/OS-II MIPS32 4K port, consisting of the files listed below, is contained in this folder. The code in these files is the focus of the next section of this document.

os_cpu.h
os_cpu_c.c
os_cpu_a.S
os_dbg.c

\Micrium\Software\uC-CPU

cpu_def.h, the file located in this folder, is part of μC/CPU, a module that contains processor-specific definitions and data types. μC/OS-II-based applications should use μC/CPU data types in place of both the standard C data types and the μC/OS-II data types described in Section 5.01.

\Micrium\Software\uC-CPU\MIPS32-4K\MPLAB-PIC32-GCC

The below μC/CPU files, which are specific to MIPS32 4K cores, can be found in this folder

cpu.h
cpu_a.s

\Micrium\Software\EvalBoards\Microchip\Explorer16\PIC32MX360\MPLAB-PIC32-GCC\BSP

An example BSP, encompassing the two files listed below, is located in this folder. As the folder's path indicates, the example BSP was developed for Microchip's Explorer16 board, a platform that can be used to evaluate that company's MIPS32-based microcontroller, the PIC32. Even if you are not using the Explorer16 board, though, the example files, which can serve as templates for a new BSP, should prove useful. For additional information pertaining to the example files, you should consult Section 7.00.

bsp.c
bsp.h

C:\Micrium\Software\CPU\Microchip\PIC32

This folder holds the below files, each of which contain library routines written for the above mentioned hardware platform. These files were furnished by Microchip.

CoreTimer.h
INT.h
p32xxxx.h
ports.h
PIC32INTlib.c

C:\Micrium\Software\EvalBoards\Microchip\Explorer16\PIC32MX360\MPLAB-PIC32-GCC\EX1_OS

This folder contains example application code that offers a means of verifying whether or not µC/OS-II is running properly on your hardware. This code, which makes up the files named below, also can be used as the basis for your own applications. Further information relating to this example application code is provided in Section 6.00.

app.c
app_cfg.h
os_cfg.h
includes.h

5.00 The μC/OS-II MIPS32 4K Port

This portion of the application note covers the port files that enable μC/OS-II to be used with MIPS32 4K processor cores. For most hardware platforms that implement these cores, you will only need a basic understanding of the port files' contents in order to begin using the operating system. However, if your hardware forces you to alter the port files, then this document's exhaustive analysis may prove beneficial. Additionally, regardless of your hardware's requirements, reviewing the descriptions that follow will increase your familiarity with the μC/OS-II MIPS32 4K port and might improve your capacity to write and debug applications that harness the associated files.

5.01 *os_cpu.h*

os_cpu.h, which is shown in Listing 5-1, defines constants and data types that are used by both the μC/OS-II MIPS32 4K port and the processor-independent portions of the operating system. As described below, this file's contents were shaped by the conventions of the MIPS32 architecture, as well as those of the MPLAB C32 C compiler that helped facilitate the development of the port. Thus, if you attempt to use the port with another compiler, you will likely need to modify *os_cpu.h*.

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

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

`OS_CPU_GLOBALS` and `OS_CPU_EXT` enable the use of global variables. This port, however, doesn't require any such variables.

Listing 5-1 (cont.), *os_cpu.h*, Data Types

```
typedef  unsigned  char            BOOLEAN;                        /* (1) */
typedef  unsigned  char            INT8U;
typedef  signed    char            INT8S;
typedef  unsigned  short           INT16U;
typedef  signed    short           INT16S;
typedef  unsigned  int             INT32U;
typedef  signed    int             INT32S;
typedef  float                     FP32;                           /* (2) */
typedef  double                    FP64;

typedef  unsigned  int             OS_STK;                         /* (3) */
typedef  unsigned  int  volatile   OS_CPU_SR;                      /* (4) */
```

L5-1(1) These data types are appropriate for the MPLAB C32 C compiler, the tool that was used to develop the μC/OS-II MIPS32 4K port. If you are not using this compiler, then you should verify that these definitions are suitable for your project.

L5-1(2) Floating-point data types, although not employed by μC/OS-II, are provided for your application's benefit. The above caveat also applies to these data types.

L5-1(3) In a μC/OS-II port, the size of the `OS_STK` data type is supposed to correspond to that of a single entry in one of the stacks manipulated by the port's code. For the MPLAB C32 compiler, this expectation was met by defining `OS_STK` as an `unsigned int`. If you are using another tool, though, you may need to redefine `OS_STK`.

L5-1(4) In order to account for the 32-bit status register defined by the MIPS32 architecture, `OS_CPU_SR`, which, as the next portion of this document explains, is actually associated with the variables that allow μC/OS-II to save processor status information, must be a 32-bit data type. Thus, if your compiler does not treat `unsigned int` variables as 32-bit values, you will need to redefine `OS_CPU_SR`.

Listing 5-1 (cont.), *os_cpu.h*, `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`

```
#define  OS_CRITICAL_METHOD     3
#define  OS_ENTER_CRITICAL()    cpu_sr = OS_CPU_SR_Save();
#define  OS_EXIT_CRITICAL()     OS_CPU_SR_Restore(cpu_sr);
```

μC/OS-II uses `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()` to denote critical sections of code in which interrupts should be disabled. The operating system supports three possible methods of disabling and then re-enabling interrupts, and the method chosen for a given port is specified via the `OS_CRITICAL_METHOD` constant. The μC/OS-II MIPS32 4K port defines `OS_CRITICAL_METHOD` as 3, meaning that a local variable, `cpu_sr`, is loaded with the processor's status upon entering a critical section and is used to restore that status when the section is exited. `cpu_sr`, then, which is expected to be of the previously discussed `OS_CPU_SR` type, must be declared in any function that defines a critical section pursuant to an `OS_CRITICAL_METHOD` value of 3. This declaration is present in each of the μC/OS-II internal functions that define critical sections, but you will need to add it to any functions in your application that call `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`.

In addition to declaring `cpu_sr`, any function that calls `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()` must execute in kernel mode. This requirement exists because `OS_CPU_SR_Save()` and `OS_CPU_SR_Restore()`, the two routines that actually save and restore the processor's status, must access CP0, and the instructions that provide this access cannot be executed in user mode. You can learn more about these two routines by referring to Section 5.03.05; however, when you define critical sections in your code, it is most important to remember that, as Section 2.01 of this document notes, the μC/OS-II MIPS32 4K port executes in kernel mode, so your application's tasks will as well.

Listing 5-1 (cont.), *os_cpu.h*, Stack Growth

```
#define  OS_STK_GROWTH          1
```

`OS_STK_GROWTH` is used to indicate to μC/OS-II the direction in which tasks' stacks should grow. As per the conventions of the MPLAB C32 compiler, this constant is defined as `1`, indicating that stacks will start at higher memory addresses and expand towards lower addresses.

Listing 5-1 (cont.), *os_cpu.h*, Task Level Context Switch

```
#define  OS_TASK_SW()         asm volatile(“syscall”)
```

µC/OS-II uses `OS_TASK_SW()` to initiate task-level context switches. Since a context switch entails saving registers and manipulating a task's stack, the nature of the code that `OS_TASK_SW()` invokes will vary from processor to processor. On MIPS32 4K cores, `OS_TASK_SW()` causes a system call exception, via the assembly instruction shown above.

The system call exceptions triggered by `OS_TASK_SW()` are handled by `OSCtxSw()`, which actually performs the processor-specific operations that make up a context switch. Since `OSCtxSw()` is always executed as the result of an exception and not a conventional function call, its context-saving code is analogous to that which partially comprises the sole interrupt handler included with the µC/OS-II MIPS32 4K port, `CoreTimerIntHandler()`. Thus, whenever a task's context must be restored, the code that is responsible for doing so can safely assume that an exception of some sort, whether an interrupt or a general exception, originally triggered the context switch. This assumption actually eliminates the need for multiple means of restoring tasks, resulting in a simpler port.

Listing 5-1 (cont.), *os_cpu.h*, Function Prototypes

```
OS_CPU_SR  OS_CPU_SR_Save(void);
void       OS_CPU_SR_Restore(OS_CPU_SR);
```

`OS_CPU_SR_Save()` and `OS_CPU_SR_Restore()` are prototyped in *os_cpu.h*, but these two routines are actually declared in *os_cpu_a.S*. `OS_CPU_SR_Save()` is used to disable interrupts and obtain the current state of the status register, while `OS_CPU_SR_Restore()` allows a specified value to be loaded into this register. When used together, the two functions are capable of defining critical sections, so they are mapped to the previously described macro functions `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`. Thus, critical sections of code should be defined as shown below:

```
      OS_CPU_SR  cpu_sr;


      OS_ENTER_CRITICAL();
      /* Code executing in a critical section */
      OS_EXIT_CRITICAL();
```

The above code results in the following two function calls:

```
      OS_CPU_SR  cpu_sr;


      os_cpu_sr = OS_CPU_SR_Save();    /* Expansion of OS_ENTER_CRITICAL() */

      /* Code executing in a critical section */

      OS_CPU_SR_Restore(cpu_sr);       /* Expansion of OS_EXIT_CRITICAL()  */
```

5.02 *os_cpu_c.c*

os_cpu_c.c contains the following C functions:

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

As the names in this list indicate, most of the routines that are declared in *os_cpu_c.c* are hook functions. Such functions, which are invoked by the processor-independent portions of μC/OS-II, allow you to safely insert application-specific code into the operating system. For example, if you would like the operating system's idle task to be appended with code that places your processor in a low-power mode, `OSTaskIdleHook()` can be used to accomplish this goal.

Since, ideally, you would not need to modify the μC/OS-II MIPS32 4K port in order to take advantage of the above hook functions, the operating system offers two configuration constants that dictate how much of *os_cpu_c.c* is sent to your compiler. If the first of these constants, `OS_APP_HOOK_EN`, is defined as `1`, a portion of the hook functions (`OSTaskCreateHook()`, `OSTaskDelHook()`, `OSTaskIdleHook()`, `OSTaskStatHook()`, `OSTaskSwHook()`, `OSTCBInitHook()`, and `OSTimeTickHook()`) will actually call a second set of hook functions, the declarations of which would need to be located in application code. Applications that incorporate μC/Probe, another of Micriμm's products, typically employ such a configuration, since `OS_APP_HOOK_EN` affects only the hook functions used by this product.

Unlike `OS_APP_HOOK_EN`, `OS_CPU_HOOKS_EN` affects all of the hook functions declared in *os_cpu_c.c*. In fact, your definition of `OS_CPU_HOOKS_EN` determines whether or not these functions' declarations will be passed to your compiler. Although this role might seem to differ substantially from that of `OS_APP_HOOK_EN`, both constants ultimately allow you to add application specific code to μC/OS-II without modifying *os_cpu_c.c*. If you use `OS_CPU_HOOKS_EN` to this end, though, please remember that you will need to provide new declarations for all of μC/OS-II's hook functions.

`OSTaskStkInit()`, which is not a hook function, is the only function in *os_cpu_c.c* with contents that are not application-specific, so it is the sole function described in this section. Listing 5-2 elaborates on key sections of the function's code, highlighting important details regarding the initialization of your tasks' stacks. Figure 5-1 provides an illustration of one of these stacks.

Listing 5-2, *os_cpu_c.c*, `OSTaskStkInit()`

```
OS_STK  *OSTaskStkInit (void (*task)(void *pd), void *p_arg, OS_STK *ptos,
                        INT16U opt)
{
    INT32U  *pstk;
    INT32U  sr_val;
    INT32U  gp_val;


    (void)opt;

    asm volatile(“mfc0  %0,$12”   : “=r”(sr_val));
    sr_val |= 0x00000001;

    asm volatile(“addi  %0,$28,0” : “=r”(gp_val));               /* (1) */

    pstk    = (INT32U *)ptos;                                     /* (2) */

    pstk--;
    *pstk-- = (INT32U)task;                                       /* (3) */
    *pstk-- = (INT32U)0x30303030;                                 /* (4) */
    *pstk-- = gp_val;
    *pstk-- = (INT32U)0x27272727;
    *pstk-- = (INT32U)0x26262626;
    *pstk-- = (INT32U)0x25252525;
    *pstk-- = (INT32U)0x24242424;
    *pstk-- = (INT32U)0x23232323;
    *pstk-- = (INT32U)0x22222222;
    *pstk-- = (INT32U)0x21212121;
    *pstk-- = (INT32U)0x20202020;
    *pstk-- = (INT32U)0x19191919;
    *pstk-- = (INT32U)0x18181818;
    *pstk-- = (INT32U)0x17171717;
    *pstk-- = (INT32U)0x16161616;
    *pstk-- = (INT32U)0x15151515;
    *pstk-- = (INT32U)0x14141414;
    *pstk-- = (INT32U)0x13131313;
    *pstk-- = (INT32U)0x12121212;
    *pstk-- = (INT32U)0x11111111;
    *pstk-- = (INT32U)0x10101010;
    *pstk-- = (INT32U)0x09090909;
    *pstk-- = (INT32U)0x08080808;
    *pstk-- = (INT32U)0x07070707;
    *pstk-- = (INT32U)0x06060606;
    *pstk-- = (INT32U)0x05050505;
    *pstk-- = (INT32U)p_arg;                                      /* (5) */
    *pstk-- = (INT32U)0x03030303;
    *pstk-- = (INT32U)0x02020202;
    *pstk-- = (INT32U)0x01010101;
    *pstk-- = (INT32U)0x00000000;                                 /* (6) */
    *pstk-- = (INT32U)0x00000000;
    *pstk-- = (INT32U)task;                                       /* (7) */
    *pstk-- = sr_val;

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

L5-2(1) An assembly language instruction is used to read the status register's contents. The value of r28, the GPR that is normally used as the global pointer, is similarly obtained. Both the value of the global pointer and that of the status register are ultimately placed on the stack. First, though, bit 0 of the latter value is set, ensuring that, when µC/OS-II first runs each of an application's tasks, interrupts will be enabled.

L5-2(2) A reference to the top of the stack is provided to `OSTaskStkInit()` via `ptos`. As a consequence of the stack usage conventions mentioned in Section 5.01, this pointer represents the highest address in the stack.

L5-2(3) After decrementing `pstk` and, thereby, leaving the stack's first entry empty, `OSTaskStkInit()` prepares an entry that references the task associated with the stack. This entry allows another of the µC/OS-II MIPS32 4K port's functions, `OSStartHighRdy()` (which is described in Section 5.03.01), to run the task.

L5-2(4) The stack entries designated to hold the contents of GPRs are initialized with values reflecting the registers' numbers. For example, the entry created for r30 is given a value of `0x30303030`. Although most of these stack entries could be initialized with arbitrary values, using register numbers allows the entries to be easily recognized during debugging.

L5-2(5) In µC/OS-II, all tasks accept a single argument. `OSTaskStkInit()`, then, must place each task's argument on the relevant stack. For any given task, `OSTaskStkInit()` stores the argument in the stack entry that represents r4, the register in which the MPLAB C32 compiler normally places a function's first argument.

L5-2(6) The two stack entries following those that represent GPRs are each assigned a value of `0x00000000`. Before the task that corresponds to the stack first runs, these entries will be used to initialize HI and LO, the special purpose registers associated with multiplication and division.

L5-2(7) The final stack entry written by `OSTaskStkInit()` contains the modified status register value mentioned above. Before writing this location, though, the stack initialization routine again prepares an entry that references the task associated with the stack. Whichever of the µC/OS-II MIPS32 4K port's routines first copies the stack's values into registers will use this entry to initialize the Exception Program Counter (EPC).

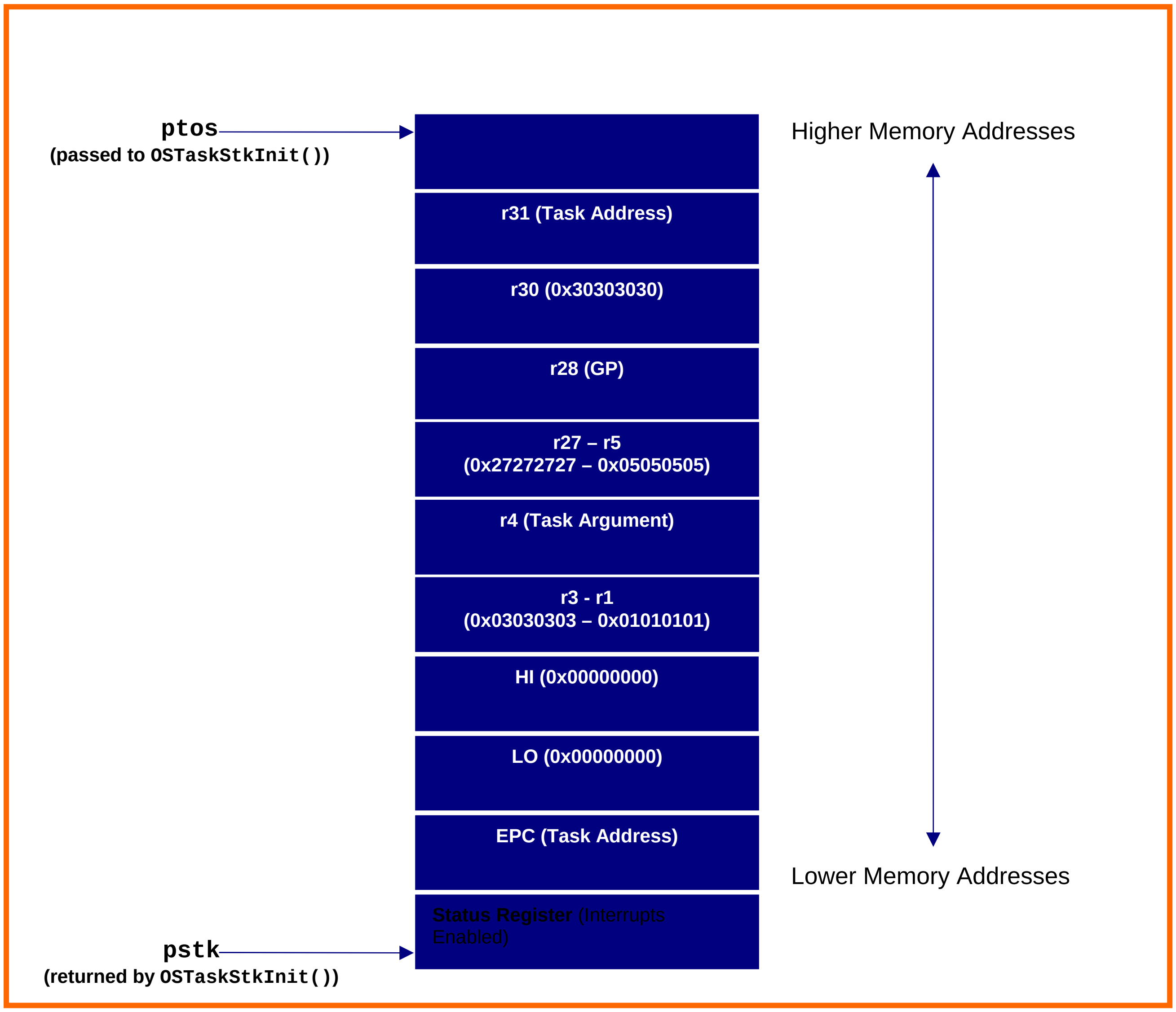


Figure 5-1, The Stack Frame for Each Task

5.03 *os_cpu_a.S*

While the processor-independent portions of µC/OS-II are written entirely in ANSI C, the operating system's ports, which are responsible for performing many operations that are not readily facilitated by C, generally incorporate at least one assembly language file. Interrupt and/or exception handlers are common constituents of these files, since routines of this sort are typically responsible for saving and restoring registers. The MIPS32 4K port's assembly language file, *os_cpu_a.S*, contains two such functions, `OSCtxSw()` and `CoreTimerIntHandler()`. The former handler, as section 5.01 of this document notes, deals with system call exceptions, while the latter function services µC/OS-II's tick interrupts.

In addition to such exception handlers, *os_cpu_a.S* defines `OS_CPU_SR_Save()` and `OS_CPU_SR_Restore()`, the interrupt disabling and re-enabling functions mentioned in the previous section. The file also defines `OSIntCtxSw()` and `OSStartHighRdy()`, two functions that, like most of the other routines defined in this file, play vital roles in multi-tasking. `OSIntCtxSw()` provides the ability to perform context switches after a task has been readied by an interrupt handler, and `OSStartHighRdy()` is used by the µC/OS-II processor-independent code to begin running your application's highest priority task, an action that signals the start of multi-tasking.

5.03.01 *os_cpu_a.s*, `OSStartHighRdy()`

`OSStartHighRdy()` is responsible for running your application's highest priority task. It is normally invoked via a call to `OSStart()`, the µC/OS-II API function that initiates multi-tasking. Before calling `OSStartHighRdy()`, `OSStart()` initializes `OSTCBHighRdy`, ensuring that this variable references the task control block (TCB) of the task that should be started. The code that is then executed to start the task is further explained in Listing 5-3.

Listing 5-3, `OSStartHighRdy()`

```
OSStartHighRdy:

    la    $8,    OSTaskSwHook                                    /* (1) */
    jalr  $8
    nop

    addi  $8,   $0, 1                                            /* (2) */
    la    $9,    OSRunning
    sb    $8,    0($9)

    la    $8,    OSTCBHighRdy                                    /* (3) */
    lw    $9,    0($8)
    lw   $29,    0($9)

    lw    $8,    STK_OFFSET_SR($29)                              /* (4) */
    mtc0  $8,   $12, 0

    lw    $8,    STK_OFFSET_EPC($29)
    mtc0  $8,   $14, 0

    lw    $8,    STK_OFFSET_LO($29)                              /* (5) */
    lw    $9,    STK_OFFSET_HI($29)
    mtlo  $8
    mthi  $9

    lw    $31,   STK_OFFSET_GPR31($29)                           /* (6) */
    lw    $30,   STK_OFFSET_GPR30($29)
    lw    $28,   STK_OFFSET_GPR28($29)
    lw    $27,   STK_OFFSET_GPR27($29)
    lw    $26,   STK_OFFSET_GPR26($29)
    lw    $25,   STK_OFFSET_GPR25($29)
    lw    $24,   STK_OFFSET_GPR24($29)
    lw    $23,   STK_OFFSET_GPR23($29)
    lw    $22,   STK_OFFSET_GPR22($29)
    lw    $21,   STK_OFFSET_GPR21($29)
    lw    $20,   STK_OFFSET_GPR20($29)
    lw    $19,   STK_OFFSET_GPR19($29)
    lw    $18,   STK_OFFSET_GPR18($29)
    lw    $17,   STK_OFFSET_GPR17($29)
    lw    $16,   STK_OFFSET_GPR16($29)
    lw    $15,   STK_OFFSET_GPR15($29)
    lw    $14,   STK_OFFSET_GPR14($29)
    lw    $13,   STK_OFFSET_GPR13($29)
    lw    $12,   STK_OFFSET_GPR12($29)
    lw    $11,   STK_OFFSET_GPR11($29)
    lw    $10,   STK_OFFSET_GPR10($29)
    lw      $9,    STK_OFFSET_GPR9($29)
    lw      $8,    STK_OFFSET_GPR8($29)
    lw      $7,    STK_OFFSET_GPR7($29)
    lw      $6,    STK_OFFSET_GPR6($29)
    lw      $5,    STK_OFFSET_GPR5($29)
    lw      $4,    STK_OFFSET_GPR4($29)
    lw      $3,    STK_OFFSET_GPR3($29)
    lw      $2,    STK_OFFSET_GPR2($29)
    lw      $1,    STK_OFFSET_GPR1($29)

    jr     $31                                                   /* (7) */
    addi   $29, $29, STK_CTX_SIZE                                /* (8) */
```

L5-3(1) `OSTaskSwHook()`, one of the hook functions described in Section 5.02, is called at the onset of `OSStartHighRdy()`. The application-specific contents of `OSTaskSwHook()` are executed whenever a context switch occurs.

L5-3(2) `OSRunning` is set to `1`, or `OS_TRUE`, to indicate that μC/OS-II is running.

L5-3(3) Since, in μC/OS-II, the address of a given task's stack can be found in the first entry of that task's TCB, `OSTCBHighRdy` actually references the stack pointer that corresponds to the task `OSStartHighRdy()` is expected to start. Thus, the value to which `OSTCBHighRdy` refers is loaded into r29, the register that is normally designated as the stack pointer.

L5-3(4) The first 32-bit word taken from the stack is placed into the status register. As Section 5.02 explains, the status register's new contents should cause interrupts to be enabled.

L5-3(5) The stack entries associated with HI and LO are loaded into these two registers. You may recall from this document's description of `OSTaskStkInit()` that these registers are simply initialized to an arbitrary value of `0x00000000`.

L5-3(6) Each of the GPRs, with the exception of r29 and r0 (the first being the stack pointer and the second containing a constant value of `0`), is loaded with a value from the stack. As the above mentioned description notes, most of these registers are originally set to a value that reflects their register number.

L5-3(7) The task that needs to be started is referenced by r31, so a `jr` instruction, with this register as its operand, is executed.

L5-3(8) The stack pointer is adjusted to account for the entries that were loaded into registers. This operation occurs in the previous `jr` instruction's branch delay slot.

5.03.02 *os_cpu_a.s*, `OSCtxSw()`

μC/OS-II performs context switching either as the result of an interrupt or in accordance with a task's use of operating system services provided for triggering such switches. When a task initiates a context switch, by calling `OSTimeDly()`, for example, `OSCtxSw()` is ultimately used to save the registers used by the task and restore those associated with the new task. However, since some processors require additional steps to be taken before saving and restoring registers, `OSCtxSw()` is not called directly by the μC/OS-II processor-independent code. Instead, this code calls the macro function `OS_TASK_SW()`, causing, in the case of the MIPS32 4K port, a system call exception. Assuming that `OSCtxSw()` resides at the general exception vector address, the system call exception ultimately results in the expected context switch. The events that compose a context switch, then, resemble those which are listed below.

```
Your task calls OSTimeDly(10);
        OSTimeDly() calls OS_Sched();
                OS_Sched() disables interrupts;
                OS_Sched() finds the highest priority task that's ready to run;
                OS_Sched() calls OS_TASK_SW();
                        A system call exception occurs;
                        OSCtxSw() handles the system call exception by performing the context switch;
                        Execution resumes in the new task;
```

As mentioned above, `OSCtxSw()` begins a context switch by saving pertinent processor registers on the stack belonging to the task that initiated the switch. The function then must load the stack pointer that will be used by the new task into r29, so that the processor registers can be set to the values that were saved on the new task's stack. The assembly language code that implements these operations is described in Listing 5-4.

Listing 5-4, OSCtxSw()

```
OSCtxSw:

    addi  $29, $29,  -STK_CTX_SIZE                                  /* (1) */

    sw     $1,   STK_OFFSET_GPR1($29)                               /* (2) */
    sw     $2,   STK_OFFSET_GPR2($29)
    sw     $3,   STK_OFFSET_GPR3($29)
    sw     $4,   STK_OFFSET_GPR4($29)
    sw     $5,   STK_OFFSET_GPR5($29)
    sw     $6,   STK_OFFSET_GPR6($29)
    sw     $7,   STK_OFFSET_GPR7($29)
    sw     $8,   STK_OFFSET_GPR8($29)
    sw     $9,   STK_OFFSET_GPR9($29)
    sw    $10,   STK_OFFSET_GPR10($29)
    sw    $11,   STK_OFFSET_GPR11($29)
    sw    $12,   STK_OFFSET_GPR12($29)
    sw    $13,   STK_OFFSET_GPR13($29)
    sw    $14,   STK_OFFSET_GPR14($29)
    sw    $15,   STK_OFFSET_GPR15($29)
    sw    $16,   STK_OFFSET_GPR16($29)
    sw    $17,   STK_OFFSET_GPR17($29)
    sw    $18,   STK_OFFSET_GPR18($29)
    sw    $19,   STK_OFFSET_GPR19($29)
    sw    $20,   STK_OFFSET_GPR20($29)
    sw    $21,   STK_OFFSET_GPR21($29)
    sw    $22,   STK_OFFSET_GPR22($29)
    sw    $23,   STK_OFFSET_GPR23($29)
    sw    $24,   STK_OFFSET_GPR24($29)
    sw    $25,   STK_OFFSET_GPR25($29)
    sw    $26,   STK_OFFSET_GPR26($29)
    sw    $27,   STK_OFFSET_GPR27($29)
    sw    $28,   STK_OFFSET_GPR28($29)
    sw    $30,   STK_OFFSET_GPR30($29)
    sw    $31,   STK_OFFSET_GPR31($29)

    mflo   $8                                                       /* (3) */
    mfhi   $9
    sw     $8,   STK_OFFSET_LO($29)
    sw     $9,   STK_OFFSET_HI($29)

    mfc0   $8, $14,  0
    addi   $8,  $8,  4
    sw     $8,   STK_OFFSET_EPC($29)

    mfc0   $8, $12,  0
    sw     $8,   STK_OFFSET_SR($29)

    ori    $8,  $0,  0x007C                                         /* (4) */
    mfc0   $9, $13,  0
    and    $9,  $9, $8
    ori   $10,  $0,  0x0020
    beq    $9, $10,  SAVE_SP
    nop

    la     $8,   BSP_Except_Handler                                 /* (5) */
    jalr   $8
    nop
    b     RESTORE_CTX
    nop

SAVE_SP:

    la     $8,    OSTCBCur                                          /* (6) */
    lw     $9,    0($8)
    sw    $29,    0($9)

    la     $8,    OSTaskSwHook                                      /* (7) */
    jalr   $8
    nop
    la     $8,    OSPrioHighRdy                                     /* (8) */
    lbu    $9,    0($8)
    la    $10,    OSPrioCur
    sb     $9,    0($10)

    la     $8,    OSTCBHighRdy                                      /* (9) */
    lw     $9,    0($8)
    la    $10,    OSTCBCur
    sw     $9,    0($10)

    lw    $29,    0($9)                                             /* (10) */

RESTORE_CTX:

    lw     $8,    STK_OFFSET_SR($29)                                /* (11) */
    mtc0   $8, $12,  0

    lw     $8,    STK_OFFSET_EPC($29)
    mtc0   $8, $14,  0

    lw     $8,    STK_OFFSET_LO($29)
    lw     $9,    STK_OFFSET_HI($29)
    mtlo   $8
    mthi   $9

    lw    $31,    STK_OFFSET_GPR31($29)
    lw    $30,    STK_OFFSET_GPR30($29)
    lw    $28,    STK_OFFSET_GPR28($29)
    lw    $27,    STK_OFFSET_GPR27($29)
    lw    $26,    STK_OFFSET_GPR26($29)
    lw    $25,    STK_OFFSET_GPR25($29)
    lw    $24,    STK_OFFSET_GPR24($29)
    lw    $23,    STK_OFFSET_GPR23($29)
    lw    $22,    STK_OFFSET_GPR22($29)
    lw    $21,    STK_OFFSET_GPR21($29)
    lw    $20,    STK_OFFSET_GPR20($29)
    lw    $19,    STK_OFFSET_GPR19($29)
    lw    $18,    STK_OFFSET_GPR18($29)
    lw    $17,    STK_OFFSET_GPR17($29)
    lw    $16,    STK_OFFSET_GPR16($29)
    lw    $15,    STK_OFFSET_GPR15($29)
    lw    $14,    STK_OFFSET_GPR14($29)
    lw    $13,    STK_OFFSET_GPR13($29)
    lw    $12,    STK_OFFSET_GPR12($29)
    lw    $11,    STK_OFFSET_GPR11($29)
    lw    $10,    STK_OFFSET_GPR10($29)
    lw     $9,    STK_OFFSET_GPR9($29)
    lw     $8,    STK_OFFSET_GPR8($29)
    lw     $7,    STK_OFFSET_GPR7($29)
    lw     $6,    STK_OFFSET_GPR6($29)
    lw     $5,    STK_OFFSET_GPR5($29)
    lw     $4,    STK_OFFSET_GPR4($29)
    lw     $3,    STK_OFFSET_GPR3($29)
    lw     $2,    STK_OFFSET_GPR2($29)
    lw     $1,    STK_OFFSET_GPR1($29)

    addi  $29, $29,   STK_CTX_SIZE

    eret                                                    /* (12) */
```

L5-4(1) The stack pointer is adjusted to accommodate the current task's registers. `STK_CTX_SIZE`, the constant value by which the pointer is decremented, is defined at the top of *os_cpu_a.S*.

L5-4(2) Nearly all of the GPRs are placed on the stack. Only r0, which always contains `0`, and r29, the register that is presumed to be the stack pointer, are not saved.

L5-4(3) The HI and LO registers, as well as the EPC and the status register, are stored at the appropriate stack locations.

L5-4(4) `OSCtxSw()` reads the cause register in order to determine which type of exception occurred. Context switches are only performed following system call exceptions.

L5-4(5) If a system call exception did not cause `OSCtxSw()` to be invoked, then a BSP routine, `BSP_Except_Handler()` is called. Via this BSP function, which is discussed further in Section 7.01 of this document, the μC/OS-II MIPS32 4K port affords applications a convenient means of handling any of the numerous exceptions associated with the general exception vector.

L5-4(6) The stack pointer's value is stored in the first field of the TCB associated with the task that requested the context switch. μC/OS-II reserves this field (which lies at the TCB's base address) for the stack pointer in order to simplify `OSCtxSw()` and other routines typically written in assembly language.

L5-4(7) `OSTaskSwHook()` is invoked to allow each context switch to be supplemented with application-specific processing.

L5-4(8) `OSPrioCur` is used by μC/OS-II's scheduler when the priority of the task executing at a given time must be determined. Thus, `OSCtxSw()` updates this variable with the priority of the task that will run when the context switch completes.

L5-4(9) `OSTCBCur`, like `OSPrioCur`, must be given a new value each time a new task begins to run. In the case of `OSTCBCur`, the new value is taken from `OSTCBHighRdy`, a variable that `OS_Sched()` initializes before `OSCtxSw()` is invoked.

L5-4(10) The address of the new task's stack is loaded into r29. This address is obtained from the first field of the TCB associated with that task.

L5-4(11) Using the stack referenced by r29, all of the registers that are normally saved during a context switch are restored.

L5-4(12) Once all of the registers have been restored, the EPC should contain an address indicating where, within the new task, the processor is to begin fetching new instructions. Accordingly, `OSCtxSw()` concludes with an `eret` instruction, the execution of which causes the contents of the EPC to be copied into the PC.

5.03.03 *os_cpu_a.S*, `CoreTimerIntHandler()`

As Section 2.01 of this document notes, only one interrupt handler is provided with the µC/OS-II MIPS32 4K port. In addition to being the sole interrupt handler, this function, `CoreTimerIntHandler()`, is also one of only two exception handlers encompassed by the port, the other such function being `OSCtxSw()`. Like its counterpart (which is described in Section 5.03.02), `CoreTimerIntHandler()` is a relatively straightforward function but is of vital importance to µC/OS-II. In order to run successfully, the operating system needs both the context switch support furnished by `OSCtxSw()` and the tick interrupts for which `CoreTimerIntHandler()` is responsible.

Although the tick interrupts upon which µC/OS-II is reliant are typically originated by a single device and warrant only one, simple interrupt handler, many applications that incorporate the operating system need to service multiple interrupt sources. Fortunately, with `CoreTimerIntHandler()` serving as a template, the µC/OS-II MIPS32 4K port's interrupt capabilities can easily be expanded. If you plan on making any such modifications to the port, however, you should first review both the description of `CoreTimerIntHandler()` that is contained in Listing 5-5 and the overview of interrupt processing offered by Figure 5-2. Additionally, you should peruse the reference materials that describe your processor's interrupt-related resources, since the nuances of this hardware will likely influence any changes that you make to the MIPS32 4K port.

Even if you are not interested in increasing the number of interrupt sources supported by the µC/OS-II MIPS32 4K port, you would likely benefit from learning how your processor services interrupts. The MIPS32 4K core documentation mentioned in Section 1.00 seems to indicate that, with respect to interrupt-related resources, there is little variation amongst these cores, but the MIPS32 architecture actually facilitates three different interrupt modes: interrupt compatibility mode, vectored interrupt mode, and external interrupt controller mode. The MIPS32 4K port's code, which is best-suited for external interrupt mode, would likely need to be modified in order to be used with a processor operating in compatibility or vectored interrupt mode.

Your processor's interrupt-related resources, in addition to dictating whether or not you will need to modify the µC/OS-II MIPS32 4K port, determine what sort of initializations your applications that incorporate the port will need to perform. Unless you have access to both the hardware platform and C compiler described in Section 4.00, then, in order to use the port, you will likely need to furnish C or assembly language routines that initialize your processor's interrupt-related registers. You will also need to ensure, perhaps by preparing a linker script, that the port's exception handlers are placed in the proper locations. For additional information concerning these requirements, you should consult Sections 7.00 – 7.03.

Listing 5-5, `CoreTimerIntHandler()`

```
CoreTimerIntHandler:

    addi  $29, $29,  -STK_CTX_SIZE                                   /* (1) */

    sw     $1,   STK_OFFSET_GPR1($29)                                /* (2) */
    sw     $2,   STK_OFFSET_GPR2($29)
    sw     $3,   STK_OFFSET_GPR3($29)
    sw     $4,   STK_OFFSET_GPR4($29)
    sw     $5,   STK_OFFSET_GPR5($29)
    sw     $6,   STK_OFFSET_GPR6($29)
    sw     $7,   STK_OFFSET_GPR7($29)
    sw     $8,   STK_OFFSET_GPR8($29)
    sw     $9,   STK_OFFSET_GPR9($29)
    sw    $10,   STK_OFFSET_GPR10($29)
    sw    $11,   STK_OFFSET_GPR11($29)
    sw    $12,   STK_OFFSET_GPR12($29)
    sw    $13,   STK_OFFSET_GPR13($29)
    sw    $14,   STK_OFFSET_GPR14($29)
    sw    $15,   STK_OFFSET_GPR15($29)
    sw    $16,   STK_OFFSET_GPR16($29)
    sw    $17,   STK_OFFSET_GPR17($29)
    sw    $18,   STK_OFFSET_GPR18($29)
    sw    $19,   STK_OFFSET_GPR19($29)
    sw    $20,   STK_OFFSET_GPR20($29)
    sw    $21,   STK_OFFSET_GPR21($29)
    sw    $22,   STK_OFFSET_GPR22($29)
    sw    $23,   STK_OFFSET_GPR23($29)
    sw    $24,   STK_OFFSET_GPR24($29)
    sw    $25,   STK_OFFSET_GPR25($29)
    sw    $26,   STK_OFFSET_GPR26($29)
    sw    $27,   STK_OFFSET_GPR27($29)
    sw    $28,   STK_OFFSET_GPR28($29)
    sw    $30,   STK_OFFSET_GPR30($30)
    sw    $31,   STK_OFFSET_GPR31($31)

    mflo   $8
    mfhi   $9
    sw     $8,   STK_OFFSET_LO($29)
    sw     $9,   STK_OFFSET_HI($29)

    mfc0   $8, $14,  0
    sw     $8,   STK_OFFSET_EPC($29)

    mfc0   $8, $12,  0
    sw     $8,   STK_OFFSET_SR($29)

    la     $8,   OSIntNesting                                        /* (3) */
    lbu    $9,   0($8)
    bne    $0,  $9,  TICK_INC_NESTING
    nop

    la    $10,   OSTCBCur
    lw    $11,   0($10)
    sw    $29,   0($11)

TICK_INC_NESTING
```

```
addi   $9,  $9,  1                                          /* (4) */
sb     $9,   0($8)

mfc0   $8, $12,  0                                          /* (5) */
mfc0   $9, $13,  0
andi   $9,  $9,  0xFC00
ins    $8,  $0, 10, 6
or     $8,  $8, $9
mtc0   $8, $12,  0

la     $8,   BSP_TickISR_Handler                            /* (6) */
jalr   $8
nop
la     $8,   OSTimeTick                                     /* (7) */
jalr   $8
nop

la     $8,   OSIntExit                                      /* (8) */
jalr   $8
nop

lw     $8,   STK_OFFSET_SR($29)                             /* (9) */
mtc0   $8, $12,  0

lw     $8,   STK_OFFSET_EPC($29)
mtc0   $8, $14,  0

lw     $8,   STK_OFFSET_LO($29)
lw     $9,   STK_OFFSET_HI($29)
mtlo   $8
mthi   $9

lw    $31,   STK_OFFSET_GPR31($29)
lw    $30,   STK_OFFSET_GPR30($29)
lw    $28,   STK_OFFSET_GPR28($29)
lw    $27,   STK_OFFSET_GPR27($29)
lw    $26,   STK_OFFSET_GPR26($29)
lw    $25,   STK_OFFSET_GPR25($29)
lw    $24,   STK_OFFSET_GPR24($29)
lw    $23,   STK_OFFSET_GPR23($29)
lw    $22,   STK_OFFSET_GPR22($29)
lw    $21,   STK_OFFSET_GPR21($29)
lw    $20,   STK_OFFSET_GPR20($29)
lw    $19,   STK_OFFSET_GPR19($29)
lw    $18,   STK_OFFSET_GPR18($29)
lw    $17,   STK_OFFSET_GPR17($29)
lw    $16,   STK_OFFSET_GPR16($29)
lw    $15,   STK_OFFSET_GPR15($29)
lw    $14,   STK_OFFSET_GPR14($29)
lw    $13,   STK_OFFSET_GPR13($29)
lw    $12,   STK_OFFSET_GPR12($29)
lw    $11,   STK_OFFSET_GPR11($29)
lw    $10,   STK_OFFSET_GPR10($29)
lw     $9,   STK_OFFSET_GPR9($29)
lw     $8,   STK_OFFSET_GPR8($29)
lw     $7,   STK_OFFSET_GPR7($29)
lw     $6,   STK_OFFSET_GPR6($29)
lw     $5,   STK_OFFSET_GPR5($29)
lw     $4,   STK_OFFSET_GPR4($29)
lw     $3,   STK_OFFSET_GPR3($29)
```

```
    lw      $2,    STK_OFFSET_GPR2($29)
    lw      $1,    STK_OFFSET_GPR1($29)

    addi  $29, $29,  STK_CTX_SIZE

    eret                                                               /* (10) */
```

L5-5(1) STK_CTX_SIZE, a constant that indicates the number of stack entries needed to accommodate a µC/OS-II task's context, is subtracted from the stack pointer.

L5-5(2) The values of most of the GPRs, and of the HI and LO registers, are recorded in the appropriate stack entries. Similarly, the status register and the EPC are saved on the stack. The resulting stack frame is identical to that created by the first half of OSCtxSw().

L5-5(3) Via OSIntNesting, CoreTimerIntHandler() can determine whether the address taken from the EPC corresponds to application code or an interrupt handler. If OSIntNesting indicates that application code was interrupted, CoreTimerIntHandler copies the stack pointer into the appropriate task's TCB. When an interrupt handler is interrupted, though, the contents of the stack pointer do not need to be recorded, since a nested interrupt cannot immediately trigger a context switch. Actually, because CoreTimerIntHandler() does not explicitly clear the status register's EXL bit, this function does not need to account for nested interrupts. As the descriptions below indicate, however, this routine could be modified by application developers wishing to use these interrupts, so the code that deals with OSIntNesting is not entirely unwarranted.

L5-5(4) OSIntNesting, the global variable that is described above, is incremented. Any interrupts occurring between the update of OSIntNesting and the ensuing call to OSIntExit() will be considered nested interrupts, and they will be capable of causing a context switch only after the completion of any handlers that were invoked before them. You should keep in mind that, unless you have modified CoreTimerIntHandler(), nested interrupts will not occur.

L5-5(5) The value of the cause register's RIPL field is copied into the status register's IPL field. The code that performs this action, like the portions of CoreTimerIntHandler() that deal with OSIntNesting, is somewhat frivolous in the absence of nested interrupts. If you plan on utilizing these interrupts, though, both such sections of code should prove useful. In fact, if you update CoreTimerIntHandler() with code that re-enables interrupts, your additions should follow the code that sets the IPL bits. When making such changes, or when attempting to use the µC/OS-II MIPS32 4K port with a device that is not operating in external interrupt controller mode, you should be aware that the portions of the cause and status registers that the port assumes to be the RIPL and IPL fields, respectively, are not defined as such by all MIPS devices.

L5-5(6) CoreTimerIntHandler() does not actually clear the timer interrupt and set up a subsequent interrupt. Instead, these action are left to BSP_TickISR_Handler(), the BSP function described in Section 7.01 of this document. With such responsibilities relegated to BSP code, CoreTimerIntHandler() can serve as a model for other interrupt handlers.

L5-5(7) By calling OSTimeTick(), CoreTimerIntHandler() notifies µC/OS-II of the tick interrupt. In any given application, only the handler that is associated with the operating system's tick interrupt needs to call OSTimeTick().

L5-5(8) `OSIntExit()` decrements `OSIntNesting` and then determines whether a context switch must be performed. If you plan on using nested interrupts and, accordingly, need to modify `CoreTimerIntHandler()`, your code that disables interrupts should precede the call to `OSIntExit()`.

L5-5(9) If `OSIntExit()` determines that there are no grounds for a context switch, `CoreTimerIntHandler()` is responsible for restoring context. Thus, the call to `OSIntExit()` is followed by instructions that load the pertinent registers with values from the stack.

L5-5(10) The closing `eret` instruction causes the processor to begin executing instructions at the address held in the EPC. Assuming that you have not implemented the modifications mentioned in the above descriptions, this address should correspond to a task that was interrupted.

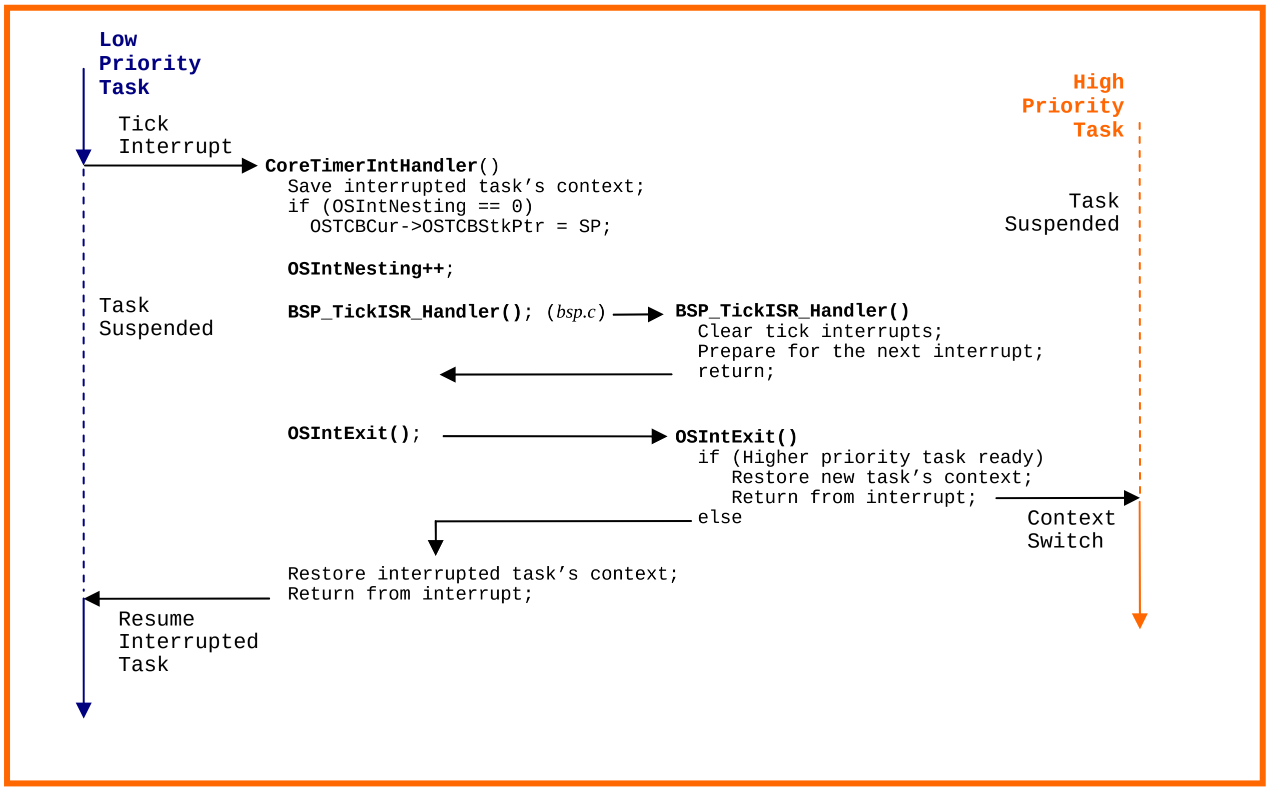


Figure 5-2, Responding to Tick Interrupts

5.03.04 *os_cpu_a.S*, `OSIntCtxSw()`

`OSIntCtxSw()` is used to perform context switches necessitated by the activity of ISRs. It is invoked by `OS_IntExit()`, the processor-independent μC/OS-II function that determines when such context switches should take place. `OS_IntExit()` initializes `OSTCBHighRdy` so that, when `OSIntCtxSw()` runs, a reference to the TCB of the task that needs to be restored is available. As Listing 5-6 indicates, `OSIntCtxSw()` loads r29 with the stack pointer value stored in this TCB and then begins a process common to many of the routines defined in *os_cpu_a.S*, involving restoring registers and running the new task.

Listing 5-6, `OSIntCtxSw()`

```
OSIntCtxSw:

    la    $8,    OSTaskSwHook                              /* (1) */
    jalr  $8
    nop

    la    $8,    OSPrioHighRdy                             /* (2) */
    lbu   $9,    0($8)
    la   $10,    OSPrioCur
    sb    $9,    0($10)

    la    $8,    OSTCBHighRdy                              /* (3) */
    lw    $9,    0($8)
    la   $10,    OSTCBCur
    sw    $9,    0($10)

    lw   $29,    0($9)                                     /* (4) */

    lw    $8,    STK_OFFSET_SR($29)                        /* (5) */
    mtc0  $8, $12,  0

    lw    $8,    STK_OFFSET_EPC($29)
    mtc0  $8, $14,  0

    lw    $8,    STK_OFFSET_LO($29)
    lw    $9,    STK_OFFSET_HI($29)
    mtlo  $8
    mthi  $9

    lw   $31,    STK_OFFSET_GPR31($29)
    lw   $30,    STK_OFFSET_GPR30($29)
    lw   $28,    STK_OFFSET_GPR28($29)
    lw   $27,    STK_OFFSET_GPR27($29)
    lw   $26,    STK_OFFSET_GPR26($29)
    lw   $25,    STK_OFFSET_GPR25($29)
    lw   $24,    STK_OFFSET_GPR24($29)
    lw   $23,    STK_OFFSET_GPR23($29)
    lw   $22,    STK_OFFSET_GPR22($29)
    lw   $21,    STK_OFFSET_GPR21($29)
    lw   $20,    STK_OFFSET_GPR20($29)
    lw   $19,    STK_OFFSET_GPR19($29)
    lw   $18,    STK_OFFSET_GPR18($29)
    lw   $17,    STK_OFFSET_GPR17($29)
    lw   $16,    STK_OFFSET_GPR16($29)
    lw   $15,   STK_OFFSET_GPR15($29)
    lw   $14,   STK_OFFSET_GPR14($29)
    lw   $13,   STK_OFFSET_GPR13($29)
    lw   $12,   STK_OFFSET_GPR12($29)
    lw   $11,   STK_OFFSET_GPR11($29)
    lw   $10,   STK_OFFSET_GPR10($29)
    lw    $9,   STK_OFFSET_GPR9($29)
    lw    $8,   STK_OFFSET_GPR8($29)
    lw    $7,   STK_OFFSET_GPR7($29)
    lw    $6,   STK_OFFSET_GPR6($29)
    lw    $5,   STK_OFFSET_GPR5($29)
    lw    $4,   STK_OFFSET_GPR4($29)
    lw    $3,   STK_OFFSET_GPR3($29)
    lw    $2,   STK_OFFSET_GPR2($29)
    lw    $1,   STK_OFFSET_GPR1($29)

    addi $29, $29, STK_CTX_SIZE

    eret                                                             /* (6) */
```

L5-6(1) `OSTaskSwHook()`, which allows application-specific processing to accompany each task switch, is invoked by both `OSCtxSw()` and `OSIntCtxSw()`, the two routines that perform context switches.

L5-6(2) `OSPrioCur` is used by μC/OS-II's scheduler when the priority of the task executing at a given time must be determined. Thus, `OSIntCtxSw()` updates this variable with the priority of the task that will run when the context switch completes.

L5-6(3) `OSTCBCur`, like `OSPrioCur`, must be given a new value each time a new task begins to run. In the case of `OSTCBCur`, the new value is taken from `OSTCBHighRdy`, a variable that μC/OS-II initializes before `OSIntCtxSw()` is invoked.

L5-6(4) The address of the new task's stack is loaded into r29. This address is obtained from the first field of the TCB associated with that task.

L5-6(5) Using the stack referenced by r29, all of the registers that are normally saved during a context switch are restored.

L5-6(6) Once all of the registers have been restored, the EPC should contain an address indicating where, within the new task, the processor is to begin fetching new instructions. Accordingly, `OSIntCtxSw()` concludes with an `eret` instruction, the execution of which causes the contents of the EPC to be copied into the PC.

5.03.05 *os_cpu_a.S*, `OS_CPU_SR_Save()`

`OS_CPU_SR_Save()` and `OS_CPU_SR_Restore()` are shown in Listing 5-7 and Listing 5-8, respectively. These two functions are used in tandem by μC/OS-II to define critical sections of code in which interrupts should be disabled. `OS_CPU_SR_Save()` and `OS_CPU_SR_Restore()` can be used by your application for similar purposes, but you should not call either of the functions directly. Instead, you should mimic μC/OS-II's use of the functions by calling `OS_ENTER_CRITICAL` and `OS_EXIT_CRITICAL`, the two macro functions described in Section 5.01.

Listing 5-7, `OS_CPU_SR_Save()`

```
OS_CPU_SR_Save:

    jr    $31
    di     $2
```

Listing 5-8, `OS_CPU_SR_Restore()`

```
OS_CPU_SR_Restore:

    jr    $31
    mtc0   $4, $12, 0
```

5.04 *os_dbg.c*

os_dbg.c is included with the μC/OS-II MIPS32 4K port, but this file does not actually declare any functions that are used by the other files comprising the port. Rather, *os_dbg.c* defines constants that are intended to be used by kernel-aware debuggers seeking information about the operating system. For the most part, even if you are using such a debugger, you need not be familiar with the contents of this file. Nonetheless, *os_dbg.c* should not be removed from the folder containing the other port files.

6.00 Application Code

Before you begin designing complex μC/OS-II-based applications on your MIPS32 processor, you should attempt to compile and run the simple application code provided with this document. This application will allow you to quickly verify that μC/OS-II is performing as expected on your hardware. Although the application is somewhat basic, it employs a structure typical of μC/OS-II-based applications, so, in addition to its potential value as a test application, it represents an ideal foundation for your future projects.

6.01 *app.c*

μC/OS-II allows you to divide your application into tasks, and *app.c*, which is explained in Listing 6-1 and Listing 6-2, features the sole task comprising the example application. This task utilizes the BSP functions described in Section 7.01 to blink an LED. If your hardware platform lacks LEDs, you will need to remove the calls to those BSP functions, and you may want to replace them with code that provides a similar visual indication that the operating system is running.

Listing 6-1, *app.c*, `main()`

```
void  main (void)
{
    CPU_INT08U  err;


    BSP_IntDisAll();                                            /* (1) */
    OSInit();                                                   /* (2) */

    OSTaskCreateExt(AppTaskStart,                               /* (3) */
                    (void *)0,
                    (OS_STK *)&AppTaskStartStk[APP_TASK_START_STK_SIZE – 1],
                    APP_TASK_START_PRIO,
                    APP_TASK_START_PRIO,
                    (OS_STK *)&AppTaskStartStk[0]
                    APP_TASK_START_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

#if OS_TASK_NAME_SIZE > 13
    OSTaskNameSet(APP_TASK_START_PRIO, “Startup”, &err);        /* (4) */
#endif

    OSStart();                                                  /* (5) */
}
```

L6-1(1) `BSP_IntDisAll()`, a routine declared in *bsp.c*, ensures that all interrupts are disabled. Interrupts are not re-enabled until the application is prepared to accept them.

L6-1(2) `OSInit()`, which, as its name implies, initializes μC/OS-II, should be called prior to any of the operating system’s other API functions.

L6-1(3) The example application's only task is created by `OSTaskCreateExt()`, one of two functions that μC/OS-II provides for the purpose of creating tasks. The other function, `OSTaskCreate()`, is analyzed alongside `OSTaskCreateExt()` in **MicroC/OS-II, The Real-Time Kernel**, the book that is described briefly in Section 4.00 and listed in the References section at the end of this document.

L6-1(4) `OSTaskNameSet()` can be used to assign names to tasks. μC/OS-II's functions do not actually use these names to identify tasks, but you may find the names to be helpful during debugging.

L6-1(5) `OSStart()` is the function through which control is transferred to μC/OS-II. Following the call to this function, the code in `AppTaskStart()`, which represents the example application's only task, will execute.

Listing 6-2, *app.c*, `AppTaskStart()`

```
static  void  AppTaskStart (void *p_arg)
{
    (void)p_arg;

    BSP_InitIO();                       /* (1) */

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

    while (1) {
        LED_Toggle(1);                  /* (3) */
        OSTimeDlyHMSM(0,0,0,100)
    }
}
```

L6-2(1) Most μC/OS-II-based applications use `BSP_InitIO()`, or a similarly-named function, to initialize all peripheral devices that will be employed by subsequent code. To ensure proper operation of the example application, this function, as Section 7.01 notes, need only initialize a couple of peripherals. Often, though, applications warrant additional initializations.

L6-2(2) `OSStatInit()` initializes μC/OS-II's statistics task, a helpful tool that determines your application's impact on your processor's resources. `OSStatInit()` should only be called if `OS_TASK_STAT_EN`, a configuration constant defined in *os_cfg.h*, is `1`. Additionally, `OSStatInit()` must be called after `BSP_InitIO()`, because it depends on the tick interrupt enabled by that function. Further information about `OSStatInit()` and the methods that it uses to gather statistics can be found in the μC/OS-II book, which is listed in the References section at the end of this application note.

L6-2(3) In μC/OS-II-based applications, the body of each task is an infinite loop. `AppTaskStart()` simply toggles an LED within this loop, indicating that everything is running properly.

6.02 *app_cfg.h*

A μC/OS-II-based application normally defines all of its tasks' priorities and stack sizes in a single file, *app_cfg.h*. μC/OS-II is not dependent on any of this file's definitions, however, unless the operating system's timer module is enabled, in which case *app_cfg.h* should define the timer task's priority, `OS_TASK_TMR_PRIO`. Despite the file's seeming unimportance, large applications normally use *app_cfg.h* to define numerous priorities and stack sizes, as well as configuration constants for other modules from Micriμm. Indeed, although *app_cfg.h* may appear to be unnecessary in relation to the example application, for larger applications, this file represents a logical way of organizing similar parameters.

6.03 *os_cfg.h*

os_cfg.h is the header file that normally defines μC/OS-II's configuration constants. These constants, which are detailed in the μC/OS-II book, *MicroC/OS-II, The Real-Time Kernel*, offer a means of scaling each of the services provided by the operating system. In order to run the example application, you will not need to make any changes to the accompanying version of *os_cfg.h*.

6.04 *includes.h*

app.c explicitly includes only a single header file, *includes.h*, in which `#include` statements for all of the header files needed by the example application can be found. The use of *includes.h* in this manner, as a master include file, is typical of μC/OS-II-based applications, even though the operating system itself doesn't rely on this file. Of course, including all of an application's header files in every C file can increase compile times, but this disadvantage must be weighed against the convenience that *includes.h* affords.

7.00 BSP (Board Support Package)

The example application, like most applications built around μC/OS-II, relegates any functions responsible for accessing peripheral devices to a single group of files, the BSP. Although BSPs often incorporate numerous functions, offering access to multiple peripheral devices, the files described in the next two sections are brief, since the aim of this application note is to provide a useful description of the μC/OS-II MIPS32 4K port, not an in-depth analysis of thousands of lines of code compatible with only one hardware platform and irrelevant to the majority of readers. An additional reason for a minimal BSP is the corresponding terseness of *app.c*, a file that is intended to serve as an example of a simple μC/OS-II-based application, and, accordingly, invokes only a modicum of BSP functions. However, even *app.c*, which features portable, easy-to-understand code, demonstrates the benefits of encapsulating hardware-specific code in a BSP.

7.01 *bsp.c*

Each of the handful of BSP functions employed by *app.c* is relatively simple and is implemented using only a few lines of code. One such function is `BSP_IntDisAll()`, a call to which normally allows μC/OS-II-based applications to prevent unexpected interrupts from occurring. Interestingly, the example BSP's implementation of this routine is empty, since *crt0.s*, the BSP file described in Section 7.03, contains startup code that explicitly disables interrupts.

As a result of the startup code's actions, the example application does not expect to receive any interrupts until `BSP_InitIO()` is called. Typically, this function initializes all of the peripheral devices used by your application, enabling these devices' interrupts when necessary. In the BSP included in *AN-Microchip-PIC32.zip*, `BSP_InitIO()` initializes three devices: a timer, an interrupt controller, and an I/O port through which LEDs can be controlled. Although failing to initialize a device capable of driving LEDs might hinder your ability to visually confirm that the application is running, the interrupt controller and the timer are the only devices that must be initialized in order for the example application to function properly. In fact, nearly all μC/OS-II-based applications must initialize a timer, since the operating system itself is dependent on tick interrupts.

Thus, even if you are not using the files described in this document, your BSP should include some form of `Tmr_Init()`, the function that the example BSP uses for timer initializations. The example implementation of this function was actually written for the core timer defined by the MIPS32 architecture. Thus, even though the library routines employed by this implementation are specific to the MPLAB C32 compiler that was used to develop the example code, the assembly instructions ultimately invoked by these routines should be suitable for initializing any MIPS32 processor's timer (although the appropriate method for setting up the corresponding interrupts depends on the interrupt mode being utilized by the processor). The assembly code corresponding to the portion of `Tmr_Init()` that is not associated with interrupt initialization is summarized in Listing 7-1.

Listing 7-1, Core Timer Initialization Instructions

```
mtc0    $0,  $9                                          /* (1) */

addi    $8,  $0, 20000
mtc0    $8, $11                                          /* (2) */
```

L7-1(1) The count register, which, along with the compare register, implements the core timer, is cleared.

L7-1(2) A value corresponding to the desired period of the tick interrupts is loaded into the compare register. Although μC/OS-II-based applications typically employ a tick frequency of 1 kHz, a much slower or faster tick rate would likely be feasible on most hardware platforms.

The example BSP provides `BSP_TickISR_Handler()` as a means of handling the tick interrupts that `Tmr_Init()` initiates. Since, as Section 5.03.03 of this document notes, `BSP_TickISR_Handler()`, is only responsible for the timer-specific aspects of handling these interrupts, the example version of this routine does not store context or invoke μC/OS-II API functions. This approach should be mimicked by any such interrupt handlers that you develop. For example, if your processor is running in external interrupt mode and you would like to create an I2C interrupt handler, you should first prepare an assembly language interrupt handler, analogous to `CoreTimerIntHandler()`, and place this handler at the appropriate vector. You should then write an I2C equivalent of `BSP_TickISR_Handler()`, meaning that you should develop a simple C function that clears the I2C interrupt and, if necessary, prepares for subsequent interrupts.

If you imagine that your application, in addition to utilizing interrupts from a timer and, perhaps, other sources, will need to be able to respond to many of the different types of exceptions supported by the MIPS architecture, then you should consider expanding `BSP_Except_Handler()`, another of the functions provided in the example BSP. This function, which is called by `OSCtxSw()` when any exception aside from a system call occurs, represents a simple way of handling any such exceptions. Although the example implementation of the `BSP_Except_Handler()` is empty, this routine can certainly be expanded according to an application's needs. This final observation, of course, applies to any portion of the example BSP; if you are not satisfied with the services that one of the BSP's routines provides, you are welcome to enhance the function to your liking.

One group of functions that you might determine to be in need of such changes, is that responsible for manipulating LEDs. The example BSP contains four such routines: `LED_On()`, `LED_Off()`, `LED_Toggle()`, and `LED_Init()`. Each of these functions, except `LED_Init()`, which is responsible for initializing the I/O port used to access the LEDs, accepts a single parameter that indicates, for evaluation boards that feature multiple LEDs, which specific device will be subject to the operation specified by the function's name, with a value of 0 used to select all available LEDs. Thus, invoking `LED_Toggle()` with 0 should cause all of your evaluation board's LEDs to change state, while passing 3 to this function will result in only the third LED changing state.

Because not all evaluation boards are equipped with LEDs, `LED_Toggle()` and the other three LED functions might not even be applicable to your hardware platform. In this case, you should remember that, if the declarations of the LED functions are removed altogether, the example application's call to `LED_Toggle()`, made by its only task, `AppTaskStart()`, must similarly be deleted. Initially, such changes should be kept to a minimum, since maintaining a relatively simple application will allow you to easily identify any problematic code.

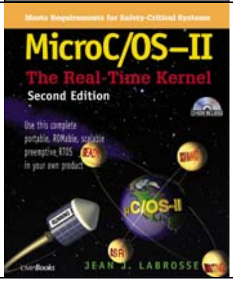
7.02 *bsp.h*

bsp.h normally features constants and function prototypes that support the routines declared in *bsp.c*. Accordingly, the example BSP, which, as the previous section notes, contains the declarations of only a few functions, provides a minimal implementation of *bsp.h*. Although this implementation can be modified if a larger BSP is needed, *bsp.h* should always maintain the basic form of the example file.

7.03 *crt0.s*

crt0.s contains startup code that is intended to be used alongside the remainder of the BSP. For the most part, this startup code is identical to that normally employed by the MPLAB C32 compiler for which the example BSP was prepared. There are two notable exceptions, though, both of which involve the exception handlers declared in *crt0.s*. In the example BSP's version of this file, the section of code placed at the general exception vector references the μC/OS-II MIPS32 4K port routine described in Section 5.03.02, `OSCtxSw()`. Similarly, rather than referencing a generic handler associated with the compiler, the code lying at the vector that corresponds to the core timer interrupt jumps to `CoreTimerIntHandler()`, the port function that is described in Section 5.03.03. Assuming that your processor operates in external interrupt controller mode, you will need to ensure that your vectors are setup in a similar manner.

References

MIPS32 Architecture For Programmers MIPS Technologies, Inc. Document Number: MD00082 Revision 2.50 July 1, 2005	 MIPS32® Architecture For Programmers Volume I: Introduction to the MIPS32® Architecture Document Number: MD00082 Revision 2.50 July 1, 2005 MIPS Technologies, Inc. 1225 Charleston Road Mountain View, CA 94043-1353 Copyright © 2001-2003,2005 MIPS Technologies Inc. All rights reserved.	
MIPS32 4K Processor Core Family **Software User's Manual** MIPS Technologies, Inc. Document Number: MD00016 Revision 1.17 September 25, 2002	 MIPS32 4K™ Processor Core Family Software User's Manual Document Number: MD00016 Revision 01.17 September 25, 2002 MIPS Technologies, Inc. 1225 Charleston Road Mountain View, CA 94043-1353 Copyright © 1998-2002 MIPS Technologies Inc. All rights reserved.	
µC/OS-II, The Real-Time Kernel, 2nd Edition Jean J. Labrosse R&D Technical Books, 2002 ISBN 1-5782-0103-9	 Meets Requirements for Safety-Critical Systems MicroC/OS-II The Real-Time Kernel Second Edition Use this complete portable, ROMable, scalable preemptive RTOS in your own product µC/OS-II JEAN J. LABROSSE CMP Books	

Contacts

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

CMP Books, Inc.
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-9950
USA
+1 785 841 1631
+1 785 841 2624 (FAX)
WEB: http://www.rdbooks.com
e-mail: rdorders@rdbooks.com