Micriμm

Empowering Embedded Systems

μC/OS-II

and the

Blackfin Processor

(ADSP-BF533)

Application Note

AN-1533

www.Micrium.com

Table of Contents

1.00 Introduction 4
1.01 µC/OS-II 4

2.00 The Analog Devices Blackfin 7
2.01 The Blackfin architecture 8
2.02 Interrupt Processing 9
2.02.01 Global Enabling/Disabling of Interrupts 9
2.02.02 Servicing an Interrupt 9
2.02.03 Software Interrupts 9
2.02.04 Nesting of Interrupts 10
2.02.05 Exceptions 11
2.03 C Run-Time model and environment 12
2.03.01 Registers 12
2.03.02 Managing the Stack 13
2.03.03 Function Arguments and Return Values 15

3.00 µC/OS-II Blackfin Port 16
3.01 Reserved ressources 16
3.02 Task Stack Frame 17

4.00 Directories and Files 18
4.01 os_cpu.h 23
4.02 os_cpu_c.c 26
4.03 os_cpu_a.asm 32
4.03.01 os_cpu_a.asm, OSStartHighRdy() 33
4.03.02 os_cpu_a.asm, OSCtxSw() 34
4.03.03 os_cpu_a.asm, OSIntCtxSw() 37
4.03.04 os_cpu_a.asm, OS_CPU_SR_Save() 38
4.03.05 os_cpu_a.asm, OS_CPU_SR_Restore() 38
4.03.06 os_cpu_a.asm, OS_CPU_EnableIntEntry() 39
4.03.07 os_cpu_a.asm, OS_CPU_DisableIntEntry() 39
4.03.08 os_cpu_a.asm, Interrupts Handling 40
4.03.09 os_cpu_a.asm, OS_CPU_NESTING_ISR() 41
4.03.10 os_cpu_a.asm, OS_CPU_NON_NESTING_ISR() 42
4.03.11 os_cpu_a.asm, OS_CPU_ISR_Entry() 42
4.03.12 os_cpu_a.asm, OS_CPU_ISR_Exit() 45

5.00 An Example Application 46
5.01 bsp.h 48
5.02 bsp.c 49
5.03 os_cfg.h 51
5.04 app_cfg.h 51
5.05 app_c.c 53

6.00 Running the Example Application 60

Licensing 61

Acknowledgements 61

References 62

Contacts 63

1.00 Introduction

This application note describes the **µC/OS-II** port for the Analog Devices Blackfin processor ADSP-BF533. A 'port' is the part of the software that adapts **µC/OS-II** to different processor architectures. The files that comprise the Blackfin port are included with this document, as is an example **µC/OS-II**-based application. This application includes also a board support package for the Blackfin ADSP-BF533. In order to run the example application, however, you will need **µC/OS-II**'s processor-independent source code, which is included in the zip file that contains this document.

This document assumes that you have **µC/OS-II** V2.86 or higher.

1.01 µC/OS-II

µC/OS-II is a completely portable, ROMable, scalable, preemptive, real-time, multitasking kernel. **µC/OS-II** is written in ANSI C and contains a small portion of assembly language code to adapt it to different processor architectures. To date, **µC/OS-II** has been ported to over 45 different processor architectures.

µC/OS-II Book

The inner workings of **µC/OS-II** are described in the **µC/OS-II** book written by author Jean J. Labrosse (see References at the end of this application note).

Source Code

µC/OS-II consists of about 5500 lines of what is probably the cleanest source code in the industry.

Portable

Most of **µC/OS-II** is written in highly portable ANSI C, with target microprocessor-specific code written in assembly language. Assembly language is kept to a minimum to make **µC/OS-II** easy to port to other processors. **µC/OS-II** can be ported to nearly any microprocessor as long as the microprocessor provides a stack pointer, and the CPU registers can be pushed onto and popped from the stack. Also, the C compiler should provide either in-line assembly or language extensions that allow you to enable and disable interrupts from C. **µC/OS-II** can run on most 8-, 16-, 32-, or even 64-bit microprocessors or microcontrollers, and DSPs.

ROMable

µC/OS-II was designed for embedded applications and thus **µC/OS-II** can be embedded as part of a product.

Scalable

µC/OS-II was designed so that you can use only the services that you need in your application. This means that a product with minimal needs can use just a few **µC/OS-II** services, while another product can benefit from the full set of features. This allows you to reduce the amount of memory (both RAM and ROM) needed by **µC/OS-II** on a per-product basis. Scalability is accomplished with the use of conditional compilation. Simply specify (through #define constants) which features you need for your application or product. Everything has been done to reduce both the code and data space required by **µC/OS-II**.

Preemptive

µC/OS-II is a fully preemptive real-time kernel. This means that **µC/OS-II** always runs the highest priority task that is ready to run. Most commercial kernels are preemptive, and **µC/OS-II** can offer comparable or, in many cases, improved performance.

Multitasking

µC/OS-II can manage up to 256 tasks. Each task has a unique priority assigned to it, which means that **µC/OS-II** does not currently facilitate round-robin scheduling. Thus, there are 256 priority levels.

Deterministic

Execution time of all **µC/OS-II** functions and services is deterministic. This means that you can always know how much time **µC/OS-II** will take to execute a function or a service. Except for `OSTimeTick()` and some of the event flag services, execution time of **µC/OS-II** services does not depend on the number of tasks running in your application.

Task Stacks

Each task requires its own stack; however, **µC/OS-II** allows each task to have a different stack size. This allows you to reduce the amount of RAM needed in your application. With **µC/OS-II**'s stack-checking feature, you can determine exactly how much stack space each task actually requires.

Services

µC/OS-II provides a number of system services, such as semaphores, mutual exclusion semaphores, event flags, message mailboxes, message queues, fixed-sized memory partitions, task management, time management functions, and more.

Interrupt Management

Interrupts can suspend the execution of a task. If a high priority task is awakened as a result of the interrupt, then that task will run as soon as the ISR completes.

Robust and Reliable

In July of 2000, **µC/OS-II** was certified in an avionics product by the Federal Aviation Administration (FAA) for use in commercial aircraft by meeting the demanding requirements of the RTCA DO-178B standard for software used in avionics equipment. In order to meet the requirements of this standard it must be possible to demonstrate through documentation and testing that the software is both robust and safe. This is particularly important for an operating system as it demonstrates that it has the proven quality to be usable in any application. Every feature, function and line of code of **µC/OS-II** has been examined and tested to demonstrate that it is safe and robust enough to be used in Safety Critical Systems where human life is on the line.

MISRA C

µC/OS-II is 99% compliant with the Motor Industry Software Reliability Association (MISRA) C Coding Standards. These standards were created by MISRA to improve the reliability and predictability of C programs in critical automotive systems. Members of the MISRA consortium include Delco Electronics, Ford Motor Company, Jaguar Cars Ltd., Lotus Engineering, Lucas Electronics, Rolls-Royce, Rover Group Ltd., and other firms and universities dedicated to improving safety and reliability in automotive electronics. Full details of this standard can be obtained directly from the MISRA web site, http://www.misra.org.uk. A detailed **µC/OS-II** compliance matrix describing all of MISRA's 127 C Coding Rules is available from the Micriµm web site. This continues the quality focus for **µC/OS-II**. Customers demand that their RTOS perform reliably in safety-critical environments.

Used in Hundreds of Products

µC/OS-II has been used in hundreds of products from companies all around the world.

Colleges and Universities

Many colleges and universities around the world are using **µC/OS-II** in courses focused on real-time systems. This ensures that new engineers are trained and ready to use **µC/OS-II** in products.

2.00 The Analog Devices Blackfin

Blackfin processors provide both microcontroller (MCU) and DSP functionality in a unified architecture, allowing flexible partitioning between the needs of control and signal processing. If the application demands, the Blackfin processor can act as 100% MCU (with code density on par with industry standards), 100% DSP (with clock rates at the leading edge of DSP technology), or a combination of the two.

The Blackfin family of processors from Analog Devices, Inc. integrates a 32-bit RISC instruction set, an 8-bit video instruction set with dual 16-bit MAC units. The processor's variable length instruction set extends up to 64-bit opcodes used in DSP inner loops (one SIMD and two load/store/cycle), but is optimized so that 16-bit opcodes represent the most frequently used instructions. As a result, compiled code density figures are competitive with industry-leading MCUs, yet its interlocked pipeline and algebraic instruction syntax facilitate development in both C/C++ and assembly.

Blackfin processors support both protected and unprotected operating modes that prevent users from accessing or affecting shared parts of the system. In addition, the processors provide memory management capabilities that enable users to define separate application development spaces. This design feature prevents distinct code sections from being overwritten. At the same time, the Blackfin architecture allows asynchronous interrupts and synchronous exceptions, as well as programmable interrupt priorities. Thus, Blackfin processors are well suited as targets for embedded operating systems.

The attached Blackfin **μC/OS-II** port was developed and tested with Analog Devices' VisualDSP++ integrated software development and debugging environment (IDDE). Refer to VisualDSP++ 5.0 User's Guide, VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors, VisualDSP++ Assembler and Preprocessor Manual, VisualDSP++ 5.0 Linker and Utilities Manual and VisualDSP++ 5.0 Device Drivers and System Services Manual for Blackfin Processors.

2.01 The Blackfin architecture

The Blackfin ADSP-BF533 contains two 16-bit multipliers, two 40-bit accumulators, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs, and a 40-bit shifter, shown in Figure 2-1. For more information on the Blackfin Processor architecture, refer to the ADSP-BF533 Blackfin Processor Hardware Reference and ADSP-BF53x/BF56x Blackfin Processor Programming Reference.

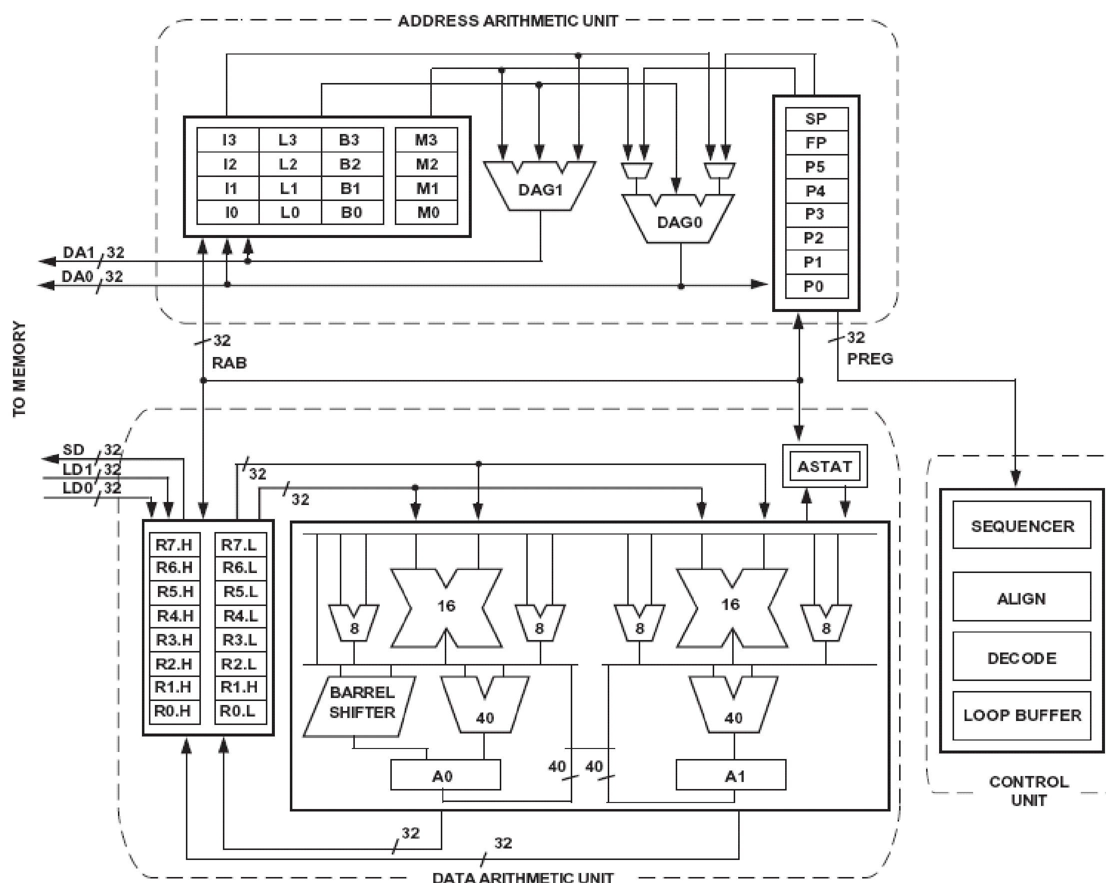


Figure 2-1, Blackfin Processor Core Architecture

The processor has a various non-memory-mapped Return Address registers that are related to the sequencer. Of these, the following two registers are of relevance to the discussion of the **μC/OS-II** port to Blackfin.

- **RETS (Subroutine return):** The `RETS` register is automatically loaded with return address when the CALL instruction is executed. The return address is the next sequential address after the CALL instruction.

- **RETI (Interrupt return):** The `RETI` register is automatically loaded with return address from an interrupt prior to jumping to the event vector. A typical interrupt service routine terminates with an RTI instruction that instructs the sequencer to reload the Program Counter, PC, from the `RETI` register.

2.02 Interrupt Processing

The following sections describe general interrupt processing concepts of the Blackfin processor.

2.02.01 Global Enabling/Disabling of Interrupts

General-purpose interrupts can be globally disabled with the `CLI` Dreg instruction and re-enabled with the `STI` Dreg instruction.

```
CLI R5; /* save IMASK to R5 and mask all */
/* place critical instructions here */
STI R5; /* restore IMASK from R5 again */
```

2.02.02 Servicing an Interrupt

If an interrupt is serviced, the program sequencer saves the return address into the `RETI` register prior to jumping to the event vector. A typical interrupt service routine terminates with an `RTI` instruction that instructs the sequencer to reload the Program Counter, `PC`, from the `RETI` register.
Servicing the highest priority interrupt involves these actions (this is a partial list. For a complete list, refer to References section for further documentation on the Blackfin Processor):

- The interrupt vector in the Event Vector Table (EVT) becomes the next fetch address. On an interrupt, most instructions currently in the pipeline are aborted.

- The return address is saved in the appropriate return register. The return register is `RETI` for interrupts. The return address is the address of the instruction after the last instruction executed from normal program flow.

- Processor mode is set to the level of the event taken.

2.02.03 Software Interrupts

The Blackfin Processor allows for software to set individual interrupt bits. The `RAISE` instruction safely sets the selected interrupt level without affecting other interrupt levels.

2.02.04 Nesting of Interrupts

- **Non-nested Interrupts:** If interrupts do not require nesting, all interrupts are disabled during the interrupt service routine. Note, however, that emulation, NMI, and exceptions are still serviced by the system. When the system does not need to support nested interrupts, there is no need to store the return address held in `RETI`. Only the portion of the machine state used in the interrupt service routine must be saved in the Supervisor stack. To return from a non-nested interrupt service routine, only the `RTI` instruction must be executed, because the return address is already held in the `RETI` register.

- **Nested Interrupts:** If interrupts require nesting, the return address to the interrupted point in the original interrupt service routine must be explicitly saved and subsequently restored when execution of the nested interrupt service routine has completed. The first instruction in an interrupt service routine that supports nesting must save the return address currently held in `RETI` by pushing it onto the Supervisor stack (`[--SP] = RETI`). This clears the global interrupt disable bit `IPEND[4]`, enabling interrupts. Next, all registers that are modified by the interrupt service routine are saved onto the Supervisor stack.
  The `RTI` instruction causes the return from an interrupt. The return address is popped into the `RETI` register from the stack, an action that suspends interrupts from the time that `RETI` is restored until `RTI` instruction finishes executing. The suspension of interrupts prevents a subsequent interrupt from corrupting the `RETI` register. Next, the `RTI` instruction clears the highest priority bit that is currently set in `IPEND`. The processor then jumps to the address pointed to by the value in the `RETI` register and re-enables interrupts by clearing global interrupt disable bit `IPEND[4]`.

- **Self-Nested Interrupts:** Interrupts that are "self-nested" can be interrupted by events at the same priority level. When the `SNEN` bit of the `SYSCFG` register is set, self-nesting of core interrupts is supported. Self-nesting is supported for any interrupt level generated with the `RAISE` instruction, as well as for core level interrupts.
  As an example, assume that the `SNEN` bit is set and the processor is servicing an interrupt generated by the **`RAISE 14;`** instruction. Once the `RETI` register has been saved to the stack within the service routine, a second **`RAISE 14;`** instruction would allow the processor to service the second interrupt. Self-nesting is not supported for system level peripheral interrupts such as the SPORT or SPI.

2.02.05 Exceptions

Exceptions are synchronous to the instruction stream. In other words, a particular instruction causes an exception when it attempts to finish execution. No instructions after the offending instruction are executed before the exception handler takes effect.

The `EXCAUSE[5:0]` field in the Sequencer Status register (`SEQSTAT`) is written whenever an exception is taken, and indicates to the exception handler which type of exception occurred. For a complete list of events that can generate exceptions, refer to *The ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

The Blackfin Instruction Set provides a "Force Exception" instruction, `EXCPT` that allows software to force exceptions with an unsigned 4-bit code. When the `EXCPT` instruction is issued, the sequencer vectors to the exception handler that the user provides.

Application-level code uses the Force Exception instruction for operating system calls. The instruction does not set the `EVSW` bit (bit 3) of the `ILAT` register.

For example,

```
EXCPT 4;
```

Would force an exception with a code of 0x4. The exception handler can handle exception appropriately based on the code passed in.

2.03 C Run-Time model and environment

The attached **μC/OS-II** Blackfin port was developed with Analog Devices' VisualDSP++ integrated software development and debugging environment (IDDE). Refer to VisualDSP++ 5.0 User's Guide, VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors, VisualDSP++ Assembler and Preprocessor Manual, VisualDSP++ 5.0 Linker.

This section provides a full description of the Blackfin processor run-time model and run-time environment. The run-time model, which applies to compiler-generated code, includes descriptions of layout of the stack, data access, and call/entry sequence. The C/C++ run-time environment includes the conventions that C/C++ routines must follow to run on Blackfin processors. Assembly routines linked to C/C++ routines must follow these conventions.

2.03.01 Registers

The C/C++ run-time environment specifies various set of registers, some of which need to be preserved across function calls, others that must be valid before calling any compiler generated code and even others that are explicitly for managing stack operations. This section provides an overview of these registers – refer to The VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors for complete reference on these topics.

- **Stack Registers:** The C/C++ run-time environment reserves a set of registers for controlling the run-time stack. These registers may be modified for stack management, but must be saved and restored. These registers are:
  `SP` – Stack pointer
  `FP` – Frame pointer

- **Dedicated registers:** The contents of this set of registers should not be changed except in specified circumstances. The contents must be valid for every function call and for any possible interrupt. Dedicated registers are:
  `SP – FP`
  `L0 – L3`

- **Scratch Registers:** The C/C++ run-time environment specifies a set of registers whose contents do not need to be saved and restored across function calls. These registers are:

Scratch Registers	Notes
P0	Used as the Aggregate Return Pointer
P1-P2	
R0-R3	The first three words of the argument list are always passed in R0, R1 and R2 if present (R3 is not used for parameters).
LB0-LB1	
LC0-LC1	
LT0-LT1	
ASTAT	Status registers.
A0-A1	
I0-I3	
B0-B3	
M0-M3	

2.03.02 Managing the Stack

The C/C++ run-time environment uses the run-time stack to store automatic variables and return addresses. The stack is managed by a Frame Pointer (`FP`) and a Stack Pointer (`SP`) and grows downward in memory, moving from higher to lower addresses.
A stack frame is a section of the stack used to hold information about the current context of the C/C++ program. Information in the frame includes local variables, compiler temporaries, and parameters for the next function.
The Frame Pointer serves as a base for accessing memory in the stack frame. Routines refer to locals, temporaries, and parameters by their offset from the Frame Pointer.

Figure 2-2 shows an example Run-Time stack where the currently executing routine, `Current()`, was called by `Previous()`, and `Current()` in turn calls `Next()`. The state of the stack is as if `Current()` has pushed all the arguments for `Next()` onto the stack and is just about to call `Next()`.

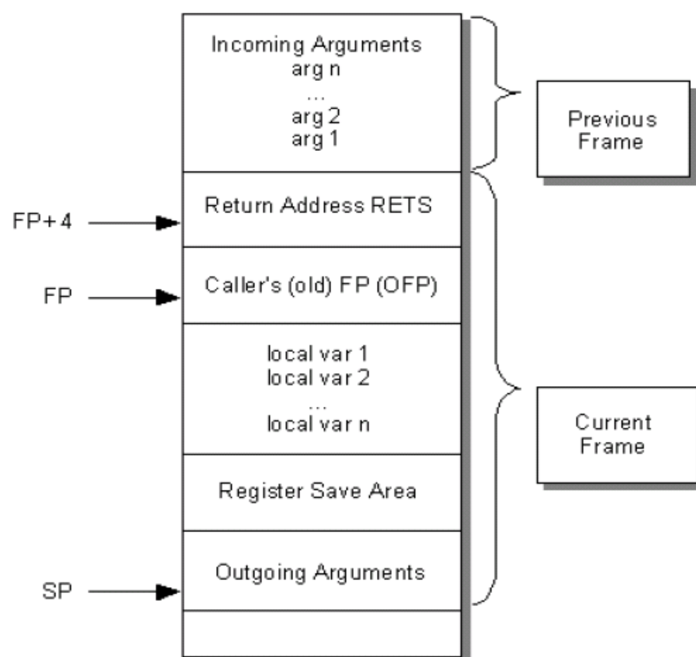


Figure 2-2: Example Run-Time stack

The compiler follows this sequence of steps when entering a function:

- **Linking Stack Frames** – The return address and the caller's `FP` are saved on the stack, and `FP` set pointing to the beginning of the new (callee) stack frame. `SP` is decremented to allocate space for local variables and compiler temporaries.

- **Register Saving** – Any registers that the function needs to preserve are saved on the stack frame, and `SP` is set pointing to the top of the stack frame.

At the end of the function, the compiler follows these steps:

- **Restore Registers** – Any registers that had been preserved are restored from the stack frame, and `SP` is set pointing to the top of the stack frame.

- **Unlinking Stack Frame** – The frame pointer is restored from the stack frame to the caller's `FP`, `RETS` is restored from the stack frame to the return address, and `SP` is set pointing to the top of the caller's stack frame.

To enable this, the Blackfin Instruction Set Architecture (ISA) features a pair of instructions that control the stack frame space on the stack and the Frame Pointer (`FP`) for that space, `LINK` and `UNLINK`.

- **`LINK`**: saves the current return address from function call (`RETS`) and `FP` registers to the stack, loads the `FP` register with the new frame address, then decrements the stack pointer (`SP`) by the user-supplied frame size value. The user-supplied argument for `LINK` determines the size of the allocated stack frame. `LINK` always saves `RETS` and `FP` on the stack, so the minimum frame size is 2 words when the argument is zero.

- **`UNLINK`**: performs the reciprocal of `LINK`, de-allocating the frame space by moving the current value of `FP` into `SP` and restoring previous values into `FP` and `RETS` from the stack.

2.03.03 Function Arguments and Return Values

The C/C++ run-time environment uses a set of registers and the run-time stack to transfer function parameters to assembly routines. The assembly language functions must follow these conventions when they call (or when called by) C/C++ functions.

Passing Arguments

This section only provides a quick overview of the conventions of passing arguments when calling compiler generated code. The complete details of passing arguments can be found in *The VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors*.
The details of argument passing are most easily understood in terms of a conceptual argument list. This is a list of words on the stack. Double arguments are placed starting on the next available word in the list, as are structures. Each argument appears in the argument list exactly as it would in storage, and each separate argument begins on a word boundary.
The actual argument list is like the conceptual argument list except that the contents of the first three words are placed in registers `R0, R1` and `R2`. Normally this means that the first three arguments (if they are integers or pointers) are passed in registers `R0` to `R2` with any additional arguments being passed on the stack.
If any argument is greater than one word, it occupies multiple registers. The caller is responsible for extending any char or short arguments to 32-bit values.

Note: When calling a C function, at least twelve bytes of stack space must be allocated for the function's arguments, corresponding to `R0-R2`. This applies even for functions that have less that 12 bytes of argument data or that have fewer than three arguments.

Return values

For functions returning aggregate values occupying less than or equal to 32 bits, the result is returned in `R0`.

For aggregate values occupying greater than 32 bits, and less than or equal to 64 bits, the result is returned in register pair `R0, R1`.

For functions returning aggregate values occupying more than 64 bits, the caller allocates the return value object on the stack and the address of this object is passed to the caller as a hidden argument in register `P0`.

3.00 µC/OS-II Blackfin Port

The following are some general notes on the attached Blackfin port of **µC/OS-II**

- The Blackfin Core Event Controller (CEC) supports nine general-purpose interrupts (IVG7 – IVG15) in addition to the dedicated interrupt and exception events that are described in the Balckfin Processor Programming manual. It is common for applications to reserve the lowest or the two lowest priority interrupts (IVG14 and IVG15) for software interrupts, leaving eight or seven prioritized interrupt inputs (IVG7 – IVG13) for peripheral purposes.

- The port supports nested interrupts for all interrupt levels, from Interrupt Level 6 (`IVG6`) through Interrupt Level 13 (`IVG13`). Interrupt Level 14 (`IVG14`) is reserved for use by **µC/OS-II** as a software trap for task-level context switches. Since **µC/OS-II** assumes that task-level context switching is uninterruptible and thus, `IVG14` is always non-nested. For interrupt levels from `IVG6` through `IVG13`, nesting can be enabled on any, some, or all interrupt levels, independent of the interrupt behavior on other interrupt levels

3.01 Reserved ressources

The attached Blackfin port uses the following resources. All other resources (including interrupt levels, peripherals, memory) are available for application use.

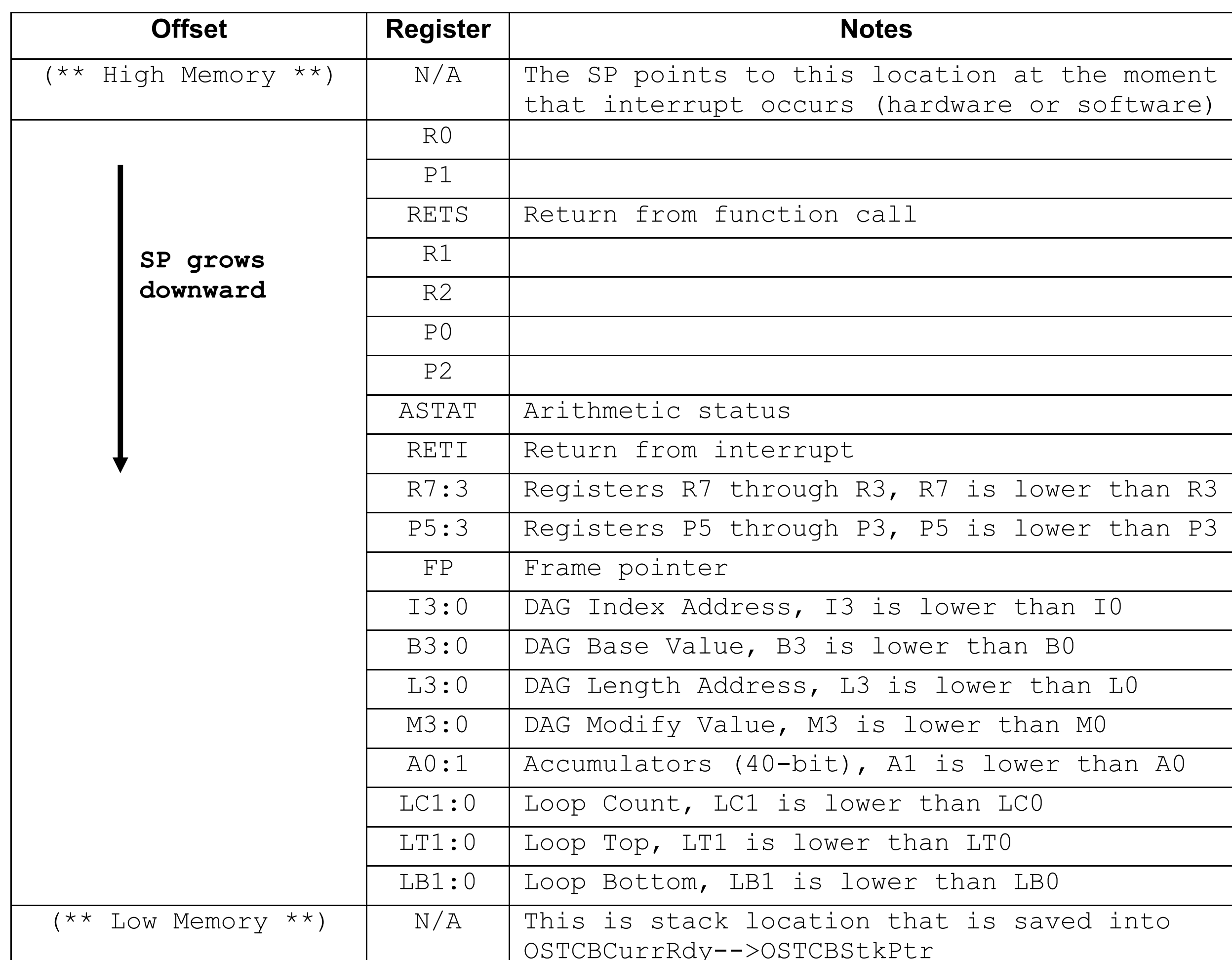
- **Core Timer:** By default, the port uses the ADSP-BF533 Core Timer (Interrupt Level 6) to provide a periodic time source to **µC/OS-II**. Thus, the Blackfin Core Timer and Interrupt Level 6 are reserved for internal `RTOS` services and not available for applications running with **µC/OS-II**. User application must initialize the core timer, and maps `IVG6` to **µC/OS-II** `OSTimeTick()` function by calling `OS_CPU_RegisterHandler()` function detailed later (See Listing 4-6).

- **Software Trap:** By default, the port uses Interrupt Level 14 (`IVG14`) as a software trap for **µC/OS-II** context switches. Thus, the `IVG14` is reserved for internal services and is not available for applications running with **µC/OS-II**. For more information, refer to Section 4.03.02, `OSCtxSw()`.

3.02 Task Stack Frame

As part of a context switch routine, a multitasking kernel such as **μC/OS-II** must save the interrupted (current) task's context to its stack and then restore the context of the new task from the new task's stack. To allow this, the kernel must follow a convention for the layout of a given task's context on its stack. Table 3.1 shows the convention followed by the attached Blackfin port of **μC/OS-II** for an interrupted task's stack frame.

Table 3.1 Interrupted task's stack after the context save is complete.



Offset	Register	Notes
(** High Memory **)	N/A	The SP points to this location at the moment that interrupt occurs (hardware or software)
SP grows downward	R0	
	P1	
	RETS	Return from function call
	R1	
	R2	
	P0	
	P2	
	ASTAT	Arithmetic status
	RETI	Return from interrupt
	R7:3	Registers R7 through R3, R7 is lower than R3
	P5:3	Registers P5 through P3, P5 is lower than P3
	FP	Frame pointer
	I3:0	DAG Index Address, I3 is lower than I0
	B3:0	DAG Base Value, B3 is lower than B0
	L3:0	DAG Length Address, L3 is lower than L0
	M3:0	DAG Modify Value, M3 is lower than M0
	A0:1	Accumulators (40-bit), A1 is lower than A0
	LC1:0	Loop Count, LC1 is lower than LC0
	LT1:0	Loop Top, LT1 is lower than LT0
	LB1:0	Loop Bottom, LB1 is lower than LB0
(** Low Memory **)	N/A	This is stack location that is saved into OSTCBCurrRdy-->OSTCBStkPtr

4.00 Directories and Files

The software included in `AN-1533-uCOS-II-v286-BF533.zip`, the zip file in which this document is often distributed, is described below.

When you unzip `AN-1533-uCOS-II-v286-BF533.zip`, you will find the following directories:

```
\Micrium\AppNotes\AN1xxx-RTOS\AN1533-uCOS-II-BF533
\Micrium\Software\EvalBoards\ADI\ADSP-BF533_EZKIT_Lite\VDSP50\OS\EX1
\Micrium\Software\EvalBoards\ADI\ADSP-BF533_EZKIT_Lite\VDSP50\BSP
\Micrium\Software\EvalBoards\ADI\ADSP-BF533_EZKIT_Lite\VDSP50\ucProbe_workspace

\Micrium\Software\uC-Probe\Target\Plugin\uCOS-II
\Micrium\Software\uC-Probe\Target\Communication\Generic\Cfg
\Micrium\Software\uC-Probe\Target\Communication\Generic\OS\uCOS-II
\Micrium\Software\uC-Probe\Target\Communication\Generic\RS-232\OS\uCOS-II
\Micrium\Software\uC-Probe\Target\Communication\Generic\RS-232\Ports\ADI\BF533\VDPS50
\Micrium\Software\uC-Probe\Target\Communication\Generic\RS-232\Source
\Micrium\Software\uC-Probe\Target\Communication\Generic\Source

\Micrium\Software\uCOS-II\Source
\Micrium\Software\uCOS-II\Ports\ADSP-BF533\VDSP50
\Micrium\Software\uCOS-II\Doc

\Micrium\Software\uC-LIB
\Micrium\Software\uC-CPU
```

The recommended locations for these folders are provided below, along with description of the contents of each folder.

`C:\Micrium\AppNotes\AN1xxx-RTOS\AN1533-uCOS-II-BF533`

This directory contains the documentation for the **μC/OS-II** ADSP-BF533 port (i.e. this document).

```
AN-1533.pdf
```

Application files

C:\Micrium\Software\EvalBoards\ADI\ADSP-BF533_EZKIT_Lite\VDSP50\OS\EX1
The files residing in this folder comprise an example **µC/OS-II**-based application. The example application was tested with ADSP-BF533 EZ-KIT Lite evaluation platform.

Application Files

```
app_c.c
app_cfg.h
os_cfg.h
```

Project Files (generated by VDSP 5.0)

```
Ex1.djp
Ex1.ldf
Ex1.mak
Ex1.pcf
Ex1_basiccrt.s
Ex1_heaptab
```

C:\Micrium\Software\EvalBoards\ADI\ADSP-BF533_EZKIT_Lite\VDSP50\BSP
The files in this folder comprise a generic board support package (BSP) for the example application.

```
bsp.c
bsp.h
```

C:\Micrium\Software\EvalBoards\ADI\ADSP-BF533_EZKIT_Lite\VDSP50\ucProbe_workspace
The file in this folder is provided as an example to demonstrate the use of **µC/Probe** within the application. This **µC/Probe** workspace is based on the one that you can find in the following directory:

```
<PATH>\Micrium\uC-Probe\Target\Plugins\uCOS-II\Workspaces
```

`<PATH>` is the path to **µC/Probe** installation directory on your system.

```
BF533-OS-Probe-Workspace.wsp
```

µC/Probe Plug-in files

C:\Micrium\Software\uC-Probe\Target\Plugin\uCOS-II
Two files are included in this directory. The functionality provided in these files is described in the **µC/Probe** user manual.

```
os_probe.c
os_probe.h
```

C:\Micrium\Software\uC-Probe\Target\Communication\Generic\Cfg
This folder contains a configuration file for the RS-232 communication module file. Refer to **μC/Probe** user manual for more information about the functionality of this file.

```
probe_com_cfg.h
```

C:\Micrium\Software\uC-Probe\Target\Communication\Generic\OS\uCOS-II
The file included in this directory is a part of the RS-232 communication module port. This module uses **μC/OS-II** services. Refer to **μC/Probe** user manual for more information.

```
probe_com_os.c
```

C:\Micrium\Software\uC-Probe\Target\Communication\Generic\RS-232\OS\uCOS-II
The file included in this directory is required to provide a task and a semaphore to allow the RS-232 module to parse packets at task level rather than in the ISR. Refer to **μC/Probe** user manual for more information.

```
probe_rs232_os.c
```

C:\Micrium\Software\uC-Probe\Target\Communication\Generic\RS-232 \Ports\ADI\BF533\VDSP50
Two files are included in this directory. These files represent the target specific port for the RS-232 communication module.

```
probe_rs232c.c
probe_rs232c.h
```

C:\Micrium\Software\uC-Probe\Target\Communication\Generic\RS-232\Source
Two files are included in this directory. These two files represent the processor-independent code for the RS-232 communication module.

```
probe_rs232.c
probe_rs232.h
```

C:\Micrium\Software\uC-Probe\Target\Communication\Generic\Source
This directory includes two files which are part of the processor-independent code for the generic communication module.

```
probe_com.c
probe_com.h
```

μC/OS-II Source and ADSP-BF533 port files

C:\Micrium\Software\uCOS-II\Source
This is the directory for the processor-independent **μC/OS-II** files. You should have the following files:

```
os_core.c
os_dbg_r.c
os_flag.c
os_mbox.c
os_mem.c
os_mutex.c
os_q.c
os_sem.c
os_task.c
os_time.c
os_tmr.c
os_cfg_r.h
ucos_ii.h
```

C:\Micrium\Software\uCOS-II\Ports\ADSP-BF533\VDSP50
The port files listed below adapt **μC/OS-II** to the Blackfin BF533 processor. These files are described in subsequent sections.

```
os_cpu.h
os_cpu_c.c
os_cpu_a.asm
```

C:\Micrium\Software\uCOS-II\Doc
This directory contains the documentation files.

```
uCOS-II-CfgMan.pdf
uCOS-II-RefMan.pdf
WhatsNewSince-v200.pdf
ReleaseNotes.pdf
Task-State-Diagram.pdf
```

C:\Micrium\Software\uC-LIB
The purpose of **μC/LIB** is to provide a clean, organized ANSI C implementation of the most common standard library functions, macros, and constants.

C:\Micrium\Software\uC-CPU
This directory contains `cpu_def.h`, which declares `#define` constants for CPU alignment, endianness, and other generic CPU properties.

C:\Micrium\Software\uC-CPU\ADSP-BF533\VDSP50
This directory contains ***cpu.h*** and ***cpu_a.asm***. *cpu.h* defines the Micriμm portable data types for 8-, 16-, and 32-bit signed and unsigned integers (such as `CPU_INT16U`, a 16-bit unsigned integer) for the Blackfin BF533 processors with the VDSP 5.0 toolchain. These allow code to be independent of processor and compiler word size definitions.

4.01 os_cpu.h

os_cpu.h contains processor and implementation-specific #define constants, macros, and typedefs. os_cpu.h for the Blackfin BF533 is shown in Listing 4-1.

Listing 4-1, 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 allow us to declare global variables that are specific to this port.

Listing 4-1, os_cpu.h, Data Types

```
typedef unsigned  int          BOOLEAN;
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;                                  /* (1) */
typedef double                 FP64;

typedef unsigned  int          OS_STK;                                /* (2) */
typedef unsigned  int          OS_CPU_SR;                             /* (3) */
typedef void (*FNCT_PTR) (void);                                      /* (4) */
```

L4-1(1) Floating-point data types are included even though **μC/OS-II** doesn't make use of floating-point numbers.

L4-1(2) A stack entry for the Blackfin BF533 processor is always 32 bits wide, and OS_STK is declared accordingly. All task stacks must be declared using OS_STK as their data type.

L4-1(3) The Blackfin Interrupt Mask register (IMASK) is 32-bits wide. The OS_CPU_SR data type is used when OS_CRITICAL_METHOD #3 is used (described below). In fact, this port only supports OS_CRITICAL_METHOD #3 because it's the preferred method for **μC/OS-II** ports.

L4-1(4) Pointer to a function which returns void and has no argument.

Listing 4-1, `os_cpu.h`, Nesting support

```
#define   NESTED               1
#define   NOT_NESTED           0
```

These defines are respectively used to support reentrant and no reentrant ISR.

Listing 4-1, `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, as with all real-time kernels, needs to disable interrupts in order to access critical sections of code and re-enable interrupts when done. **µC/OS-II** defines two macros to disable and enable interrupts: `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`, respectively. **µC/OS-II** defines three ways to implement these macros, but a **µC/OS-II** port only needs to use one of the three methods. *MicroC/OS-II, The Real-Time Kernel, 2nd Edition* (see References, at the end of this document) describes the three different methods. The appropriate method for a given project depends on the processor and compiler. The preferred method is `OS_CRITICAL_METHOD` #3, which is used for the Blackfin port

`OS_CRITICAL_METHOD` #3 implements `OS_ENTER_CRITICAL()` by calling `OS_CPU_SR_Save()` function that will retrieves the old value of IMASK register, save this value into `cpu_sr`, and disables interrupts by setting IMASK to all zeros. `OS_EXIT_CRITICAL()` invokes `OS_CPU_SR_Restore()` function to restore the status register IMASK from the variable `cpu_sr`, and enables the interrupt system according to the new mask stored

The code for functions `OS_CPU_SR_Save()` and `OS_CPU_SR_Restore()` is declared in `os_cpu_a.asm`, which is described in subsequent sections of this document.

Listing 4-1, `os_cpu.h`, Stack Growth

```
#define   OS_STK_GROWTH          1
```

The stacks on the Blackfin BF533 grow from high memory to low memory. This convention is indicated to **µC/OS-II** by setting `OS_STK_GROWTH` to `1`.

Listing 4-1, os_cpu.h, Task-Level Context Switch

```
#define  OS_TASK_SW()          asm("raise 14;");
```

OS_TASK_SW() macro is defined to 'raise' Interrupt Level 14 (IVG14).

Task-level context switches are performed when µC/OS-II invokes the macro OS_TASK_SW(). Because context switching is processor-specific, OS_TASK_SW() needs to execute an assembly language function called OSCtxSw(), which is declared in os_cpu_a.asm. Thus, OSCtxSw() is registered as an ISR for interrupt level 14 (IVG14).

The code for OSCtxSw() function is declared in os_cpu_a.asm, which is described in subsequent sections of this document.

Listing 4-1, os_cpu.h, Function Prototypes

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

void       OS_CPU_RegisterHandler(INT8U ivg, FNCT_PTR fn, BOOLEAN nesting);
```

The first two functions are prototyped in os_cpu.h but are declared in os_cpu_a.asm.

OS_CPU_SR_Save() saves the state of the IMASK into a local variable and then disables interrupts. The saved IMASK is returned to the caller. OS_CPU_SR_Restore() simply restores the IMASK register with the value of the argument passed. This restores the interrupt status to the value saved prior to the call to OS_CPU_SR_Save().

OS_CPU_SR_Save() and OS_CPU_SR_Restore() are 'mapped' to the macro functions OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL(), respectively. OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL() are always used in pairs, as shown below:

```
OS_CPU_SR  cpu_sr;


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

When expanded by the compiler, the code from the macros appears as follows:

```
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()  */
```

The third function, is prototyped in os_cpu.h but declared in os_cpu_c.c, and is used to register an interrupt handler (the second argument) for a given interrupt vector group (the first argument). The third argument indicates if nesting is supported or not for this interrupt handler.

Listing 4-1, os_cpu.h, Interrupt vector group number

```
#define  IVG0                 0
#define  IVG1                 1
#define  IVG2                 2
#define  IVG3                 3
#define  IVG4                 4
#define  IVG5                 5
#define  IVG6                 6
#define  IVG7                 7
#define  IVG8                 8
#define  IVG9                 9
#define  IVG10               10
#define  IVG11               11
#define  IVG12               12
#define  IVG13               13
#define  IVG14               14
#define  IVG15               15
```

The defines above are used to identify to which interrupt level group will be mapped a given ISR. More information about the core event table is available in the Blackfin Programming Manual.

4.02 os_cpu_c.c

Listing 4-2, os_cpu_c.c, Local defines

```
#define  EVENT_VECTOR_TABLE_ADDR  0xFFE02000                               /* (1) */
#define  IPEND                    0xFFE02108                               /* (2) */
#define  IPEND_BIT_4_MASK         0xFFFFFEFF                               /* (3) */
#define  IVG_NUM                          16                               /* (4) */
#define  pIPEND   ((volatile unsigned long *)IPEND)
```

L4-2(1) The core event table start address.

L4-2(2) The IPEND register address.

L4-2(3) Global interrupt disable bit IPEND[4]. When cleared, interrupts are enabled depending on the value of IMASK register, and when set interrupts are disabled.
This define is used by OS_CPU_IntHandler() to avoid the test of the IPEND[4].

L4-2(4) The core event table is 16 entries.

Listing 4-3, os_cpu_c.c, Interrupt Handler table

```
static  FNCT_PTR  OS_CPU_IntHanlderTab[IVG_NUM] = {(void *)0};
```

When an interrupt handler is registered by calling OS_CPU_RegisterHandler() function, it will be memorized into OS_CPU_IntHandlerTab table at the index represented by the first argument of OS_CPU_RegisterHandler() function.

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

```
OSInitHookBegin()
OSInitHookEnd()          Described in this section
OSTaskCreateHook()
OSTaskDelHook()
OSTaskIdleHook()
OSTaskStatHook()
OSTaskStkInit()          Described in this section
OSTaskSwHook()
OSTCBInitHook()
OSTimeTickHook()
```

μC/OS-II really only requires `OSTaskStkInit()`. The other functions MUST be declared but can be empty.
To use the **μC/OS-II** required functions declared in this file, the **MISCELLANEOUS** parameter `OS_CPU_HOOKS_EN` should be set to 1. In general this parameter resides in `os_cfg.h` file that allow you to configure others parameters. If `OS_CPU_HOOKS_EN` is set to `0`, then you must declare the functions listed above in another file. You should thus note that these functions MUST always be declared even though they might not contain any code.

`OSTaskIdleHook()` is called when **μC/OS-II** has no other task to run. As described in *MicroC/OS-II, The Real-Time Kernel, 2nd Edition* (see References, at the end of this document), `OSTaskIdleHook()` can be used to place the CPU in a low power mode.

Listing 4-4, `os_cpu_c.c`, OSInitHookEnd ()

```
void OSInitHookEnd (void)
{
    INT32U * pEventVectorTable;

    pEventVectorTable = ((INT32U*)EVENT_VECTOR_TABLE_ADDR);  /* Event Vector Table Pointer    */
    pEventVectorTable[IVG14] = (INT32U)&OSCtxSw;             /* Register the context switch  */
                                                             /* handler for IVG14            */
    OS_CPU_EnableIntEntry(IVG14);                            /* Enable Interrupt for IVG14    */
}
```

In the above function, `OSCtxSw()` is installed as the ISR for Interrupt Level 14 (IVG14, context switch software trap).
Also, the corresponding IMASK register bit is set, by calling `OS_CPU_EnableIntEntry()` function, to allow IVG14 to interrupt the core.

Listing 4-5, `os_cpu_c.c`, OSTaskStkInit ()

`OSTaskStkInit()` essentially simulates a function call to the task with an argument and simulates the context save when vectoring to an ISR

```
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT16U opt)
{
    OS_STK  *stk;
    INT8U    i;

    opt    = opt;                                   /* 'opt' is not used, prevent warning        */
    stk    = ptos;                                  /* Load stack pointer                        */
                                                    /* Simulate a function call to the task      */
                                                       with an argument                          */
    stk   -= 3;                                     /* 3 words assigned for incoming args        */
                                                    /* (R0, R1, R2)                              */

                                                    /* Now simulating vectoring to an ISR        */
    *--stk = (OS_STK) pdata;                        /* R0 value - caller's incoming argument #1  */
    *--stk = (OS_STK) 0;                            /* P1 value - value irrelevant               */

    *--stk = (OS_STK)OS_CPU_Invalid_Task_Return;    /* RETS value, NO task should return with RTS */
                                                    /* however OS_CPU_Invalid_Task_Return is a   */
                                                    /* safety catch-allfor tasks that return     */
                                                    /* with an  RTS                              */

    *--stk = (OS_STK) pdata;                        /* R1 value - caller's incoming argument #2  */
                                                    /* (not relevant in current test example)    */
    *--stk = (OS_STK) pdata;                        /* R2 value - caller's incoming argument #3  */
                                                    /* (not relevant in current test example)    */
    *--stk = (OS_STK) 0;                            /* P0 value - value irrelevant               */
    *--stk = (OS_STK) 0;                            /* P2 value - value irrelevant               */
    *--stk = (OS_STK) 0;                            /* ASTAT value - value irrelevant            */
    *--stk = (OS_STK) task;                         /* RETI value- pushing the start address     */
                                                    /* of the task                               */

    for (i = 35; i > 0; i--) {                      /* remaining reg values - R7:3, P5:3,        */
                                                    /* 4 words of A1:0(.W,.X), LT0, LT1,         */
        *--stk = (OS_STK)0;                         /* LC0, LC1, LB0, LB1,I3:0, M3:0, L3:0, B3:0 */
    }                                               /* All values irrelevant                     */

    return ((OS_STK *)stk);                         /* Return top-of-stack                       */
}
```

When the task is created, the final value of `stk` is placed in the `OS_TCB` of that task. Note that the `LB0` is the ‘last’ entry placed on the stack and it’s actually located at the top of the stack. `stk` thus points to the LAST element placed onto the stack.

Figure 4-1 shows how the stack frame is initialized for each task when it’s created.

Low Memory

stk → LB0 ← OSTCBStkPtr

LB1
LT0
LT1
LC0
LC1
A1.w
A1.x
A0.w
A0.x
M3:0
L3:0
B3:0
I3:0
FP
P5:3
R7:3
RETI
ASTAT
P2
P0
R2
R1
RETS
P1
R0

ptos →

High Memory

OS_TCB

M3:0 **M3 is lower than M0**
L3:0 **L3 is lower than L0**
B3:0 **B3 is lower than B0**
I3:0 **I3 is lower than I0**
P5:3 **P5 is lower than P3**
R7:3 **R7 is lower than R3**

Rx:y notation is used to represent a set of registers Rx to Ry.
One stack entry is reserved for each register,

Figure 4-1, The Stack Frame for Each Task.

In addition to the functions discussed above, two others functions are declared in `os_cpu_c.c`:

```
void  OS_CPU_RegisterHandler (INT8U ivg, FNCT_PTR fn, BOOLEAN nesting);
void  OS_CPU_IntHandler      (void);
```

Listing 4-6, os_cpu_c.c, OS_CPU_RegisterHandler ( )

Depending on the value of the last argument, this function installs `OS_CPU_NESTING_ISR()` or `OS_CPU_NON_NESTING_ISR()` as a default ISR entry for the given interrupt vector (represented by the first argument ivg). Then, it registers the Interrupt handler routine (represented by the second argument) within the table `OS_CPU_IntHanlderTab`. At the end it enables the interrupt for this IVG by setting the corresponding `IMASK` bit, this is achieved by `OS_CPU_EnableIntEntry()` function declared in `os_cpu_a.asm`

Thus, when an interrupt occurred, `OS_CPU_NESTING_ISR()` or `OS_CPU_NON_NESTING_ISR()` will execute. These ISRs save the current context, call `OS_CPU_IntHandler()` function (described below). Then, the corresponding interrupt handler routine (registered within the table `OS_CPU_IntHanlderTab`) will be executed to service the interrupt request. Then, the saved context at the start of `OS_CPU_NESTING_ISR()` or `OS_CPU_NON_NESTING_ISR()` will be restored.

```
void  OS_CPU_RegisterHandler (INT8U ivg, FNCT_PTR fn, BOOLEAN nesting)
{
    INT32U   *pEventVectorTable;


    if (ivg > IVG15) {                                           /* (1) */
        return;
    }

    if (ivg == IVG14)  {                                         /* (2) */
        return;
    }

    if (ivg == IVG4)  {                                          /* (3) */
        return;
    }

    pEventVectorTable = (INT32U*)EVENT_VECTOR_TABLE_ADDR;        /* (4) */

    if (nesting == NESTED)  {

       pEventVectorTable[ivg]  =  (INT32U)&OS_CPU_NESTING_ISR;        /* (5) */

    } else {

       pEventVectorTable[ivg]  =  (INT32U)&OS_CPU_NON_NESTING_ISR;    /* (6) */
    }

    OS_CPU_IntHanlderTab[ivg]  = fn;                             /* (7) */
    OS_CPU_EnableIntEntry(ivg);                                  /* (8) */
}
```

L4-6(1) The core event table of ADSP-BF533 is 16 entries.

L4-6(2) IVG14 is reserved for task-level context switching.

L4-6(3) IVG4 is a reserved vector.

L4-6(4) pEventVectorTable points to the start address of Event vector table.

L4-6(5) Select OS_CPU_NESTING_ISR() as the ISR for the given interrupt level (represented by the first argument ivg) if nesting is required.

L4-6(6) Select OS_CPU_NON_NESTING_ISR() as the ISR for the given interrupt level (represented by the first argument ivg) if nesting is not required.

L4-6(7) Register the Handler routine for the given interrupt level represented by ivg.

L4-6(8) Set the corresponding IMASK bit, this enables interrupt for the given interrupt level represented by ivg.

Listing 4-7, os_cpu_c.c, OS_CPU_IntHandler ( )

```
void  OS_CPU_IntHandler (void)
{
    INT32U  status;
    INT32U  mask;
    INT8U   i;


    mask   = 1;
    status = *pIPEND & IPEND_BIT_4_MASK;                              /* (1) */

    for (i =0; i < IVG_NUM; i++) {

        if ((1 << i) == (status & mask)) {

            if (OS_CPU_IntHanlderTab[i] != (void *)0) {              /* (2) */

                OS_CPU_IntHanlderTab[i]();                           /* (3) */
            }
            break;
        }
        mask <<=1;
    }
}
```

This function is called by OS_CPU_NESTING_ISR () or OS_CPU_NON_NESTING_ISR (). It tests the current IPEND register value and determines the highest priority interrupt request to serve, and then it calls the appropriate interrupt handler routine already registered within OS_CPU_IntHandlerTab.

L4-7(1) `IPEND[4]` bit is reserved for global interrupt disable.

L4-7(2) Be sure that a valid interrupt handler routine was registered.

L4-7(3) Call the appropriate interrupt handler routine.

4.03 os_cpu_a.asm

A **µC/OS-II** port requires a few fairly simple assembly language functions. These functions are needed because you normally cannot save/restore registers from C functions. These functions are:

```
OSStartHighRdy()
OSCtxSw()
OSIntCtxSw()
OS_CPU_NESTING_ISR()
OS_CPU_NON_NESTING_ISR()
```

A few other macros have been defined to allow for readable assembly code. Refer to ADSP-BFxx Blackfin Processor Programming Reference for complete details on D-register and P-register Load instructions.

Listing 4-8, os_cpu_a.asm, Defining Load macros

```
#define UPPER_(x)      (((x) >> 16) & 0x0000FFFF)
#define LOWER_(x)      ((x) & 0x0000FFFF)
#define LOAD(x, y)     x##.h = UPPER_(y); x##.l = LOWER_(y)         /* (1) */
#define LOADA(x, y)    x##.h = y; x##.l = y                         /* (2) */
```

L4-8(1) Load Immediate into register.

L4-8(2) Load address into register.

Listing 4-9, os_cpu_a.asm, Build / Destroy C-Run Time Stack

```
#define INIT_C_RUNTIME_STACK(frame_size)\                         /* (1) */
    LINK frame_size;                    \
    SP += -12;                                                    /* (2) */

#define DEL_C_RUNTIME_STACK()           \                         /* (3) */
    UNLINK;
```

L4-9(1) `INIT_C_RUNTIME_STACK` creates a frame of size = `frame_size` parameter. The application passes this parameter when it calls the ISR Entry routine (see Section 4.03.08).

L4-9(2) Create space on the stack for 3 outgoing arguments (each is 32-bits wide, for a total of 12 bytes) when calling C functions. For more information on managing C Run-Time stack, refer to Section 2.03, C Run-Time Model and Environment.

L4-9(3) `UNLINK` destroys the frame created in L4-9(1)

4.03.01 os_cpu_a.asm, OSStartHighRdy()

OSStartHighRdy() is shown in Listing 4-10. This function is called by OSStart(), which is declared in os_core.c, to start running the highest priority task that was created before calling OSStart(). OSStart() sets OSTCBHighRdy to point to the OS_TCB of the highest priority task.

Listing 4-10, os_cpu_a.asm, OSStartHighRdy ()

```
_OSStartHighRdy:

#if (OS_TASK_SW_HOOK_EN == 1)      /* Do the task switch hook function  - don't need to save    */
                                   /* RETS since _OSStartHighRdy never returns                  */
    INIT_C_RUNTIME_STACK(0x0)      /* Setup C-runtime stack - call OSTaskSwHook()               */
    CALL.X _OSTaskSwHook;          /* A call is made to OSTaskSwHook(), a user-defined function */
    DEL_C_RUNTIME_STACK()          /* Tear down C run-time stack - restores SP                  */

#endif

    LOADA(P1, _OSTCBHighRdy);      /* Get the SP for the highest ready task                     */
    P2      = [ P1 ];
    SP      = [ P2 ];
    LOADA(P1, _OSRunning);         /* Set OSRunning to true                                     */
    R1      = 1;
    [ P1 ] = R1;
                                   /* Restore CPU context without  popping off RETI             */
    P1      = 140;                 /* Skipping over LB1:0, LT1:0, LC1:0, A1:0, M3:0,            */
    SP      = SP + P1;             /* L3:0, B3:0, I3:0, FP, P5:3, R7:3                          */
    RETS    = [ SP ++ ];           /* Pop off RETI value into RETS                              */
    SP     += 12;                  /* Skipping over ASAT, P2, P0                                */

    R1      = 0;                   /* Zap loop counters to zero, to make sure                   */
    LC0     = R1;                  /* that hw loops are disabled                                */
    LC1     = R1;
    L0      = R1;                  /* Clear the DAG Length regs too, so that it's safe          */
    L1      = R1;
    L2      = R1;                  /* to use I-regs without them wrapping around.               */
    L3      = R1;
    R2      = [ SP ++ ];           /* Loading the 3rd argument of the C function - R2           */
    R1      = [ SP ++ ];           /* Loading the 2nd argument of the C function - R1           */
    SP     += 8;                   /* Skipping over RETS, P1                                    */
    R0      = [ SP ++ ];           /* Loading the 1st argument of the C function - R0           */

                                   /* Return to high ready task                                 */
_OSStartHighRdy.end:
    RTS;
```

4.03.02 os_cpu_a.asm, OSCtxSw()

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

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();
      OS_TASK_SW() performs the context switch;
      Execution resumes in the new task;

The attached ADSP-BF533 port of **μC/OS-II** maps OS_TASK_SW() to a RAISE (Force Interrupt) instruction. Specifically, it is mapped to raise Interrupt Level 14 (IVG14) and OSCtxSw() is defined as the ISR for IVG14.

OSCtxSw() simply consists of saving the CPU registers on the stack of the task to suspend and restoring the CPU registers of the new task from its stack. The pseudocode for this is shown below.

```
Save the CPU registers onto the old task's stack;
OSTCBCur->OSTCBStkPtr = SP;
OSTaskSwHook();
OSPrioCur             = OSPrioHighRdy;
OSTCBCur              = OSTCBHighRdy;
SP                    = OSTCBHighRdy->OSTCBStkPtr;
Restore the CPU registers from the new task's stack;
```

The code to perform a task-level context switch is shown below. OSCtxSw() is called when a higher priority task is made ready to run by another task, or when the current task can no longer execute (it calls OSTimeDly(), OSSemPend() and the semaphore is not available, etc.).

It's very important to note that OSCtxSw() executes while interrupts are disabled.

Listing 4-11, `os_cpu_a.asm`, OSCtxSw()

```
_OSCtxSw:
                                        /* Save context, interrupts disabled by IPEND[4] bit        */
    [ -- SP ]    = R0;
    [ -- SP ]    = P1;
    [ -- SP ]    = RETS;
    [ -- SP ]    = R1;
    [ -- SP ]    = R2;
    [ -- SP ]    = P0;
    [ -- SP ]    = P2;
    [ -- SP ]    = ASTAT;
    R1           = RETI;                /* IPEND[4] is currently set, globally disabling interrupts  */
                                        /* IPEND[4] will stay set when RETI is saved through R1      */

    [ -- SP ]    = R1;
    [ -- SP ]    = (R7:3, P5:3);
    [ -- SP ]    = FP;
    [ -- SP ]    = I0;
    [ -- SP ]    = I1;
    [ -- SP ]    = I2;
    [ -- SP ]    = I3;
    [ -- SP ]    = B0;
    [ -- SP ]    = B1;
    [ -- SP ]    = B2;
    [ -- SP ]    = B3;
    [ -- SP ]    = L0;
    [ -- SP ]    = L1;
    [ -- SP ]    = L2;
    [ -- SP ]    = L3;
    [ -- SP ]    = M0;
    [ -- SP ]    = M1;
    [ -- SP ]    = M2;
    [ -- SP ]    = M3;
    R1.L         = A0.x;
    [ -- SP ]    = R1;
    R1           = A0.w;
    [ -- SP ]    = R1;
    R1.L         = A1.x;
    [ -- SP ]    = R1;
    R1           = A1.w;
    [ -- SP ]    = R1;
    [ -- SP ]    = LC0;
    R3           = 0;
    LC0          = R3;
    [ -- SP ]    = LC1;
    R3           = 0;
    LC1          = R3;
    [ -- SP ]    = LT0;
    [ -- SP ]    = LT1;
    [ -- SP ]    = LB0;
    [ -- SP ]    = LB1;
    L0           = 0 ( X );
    L1           = 0 ( X );
    L2           = 0 ( X );
    L3           = 0 ( X );
```

```
    LOADA(P3, _OSPrioHighRdy);    /* Get a high ready task priority                      */
    R4          = B[ P3 ](Z);
    LOADA(P3, _OSPrioCur);        /* Get a current task priority                         */
    R5          = B[ P3 ](Z);
    CC          = R4 == R5;       /* If new priority == current priority, don't do anything. */
    IF CC JUMP _AbortOSCtxSw;
    LOADA(P4, _OSTCBCur);         /* Get a pointer to the current task's TCB             */
    P5          = [ P4 ];
    [ P5 ]      = SP;             /* Context save done so save SP in the TCB             */

#if (OS_TASK_SW_HOOK_EN == 1)
    INIT_C_RUNTIME_STACK(0x0)     /* Setup C-runtime stack - call OSTaskSwHook()         */
    CALL.X _OSTaskSwHook;
    DEL_C_RUNTIME_STACK()         /* Tear down C run-time stack - restores SP            */
#endif

    B[ P3 ]     = R4;             /* OSPrioCur = OSPrioHighRdy                           */
    LOADA(P3, _OSTCBHighRdy);     /* Get a pointer to the high ready task's TCB          */
    P5          = [ P3 ];
    [ P4 ]      = P5;             /* OSTCBCur = OSTCBHighRdy                             */

_OSCtxSw_modify_SP:
    SP          = [ P5 ];         /* Make it the current task by switching               */
                                  /* the stack pointer                                   */
_AbortOSCtxSw:
_CtxSwRestoreCtx:
_OSCtxSw.end:
    LB1         = [ SP ++ ];      /* Restoring CPU context and return to task            */
    LB0         = [ SP ++ ];
    LT1         = [ SP ++ ];
    LT0         = [ SP ++ ];
    LC1         = [ SP ++ ];
    LC0         = [ SP ++ ];
    R0          = [ SP ++ ];
    A1          = R0;
    R0          = [ SP ++ ];
    A1.x        = R0.L;
    R0          = [ SP ++ ];
    A0          = R0;
    R0          = [ SP ++ ];
    A0.x        = R0.L;
    M3          = [ SP ++ ];
    M2          = [ SP ++ ];
    M1          = [ SP ++ ];
    M0          = [ SP ++ ];
    L3          = [ SP ++ ];
    L2          = [ SP ++ ];
    L1          = [ SP ++ ];
    L0          = [ SP ++ ];
    B3          = [ SP ++ ];
    B2          = [ SP ++ ];
    B1          = [ SP ++ ];
    B0          = [ SP ++ ];
    I3          = [ SP ++ ];
    I2          = [ SP ++ ];
    I1          = [ SP ++ ];
    I0          = [ SP ++ ];
    FP          = [ SP ++ ];
    (R7:3, P5:3) = [ SP ++ ];
    RETI        = [ SP ++ ];      /* IPEND[4] will stay set when RETI popped from stack  */
    ASTAT       = [ SP ++ ];
    P2          = [ SP ++ ];
    P0          = [ SP ++ ];
    R2          = [ SP ++ ];
    R1          = [ SP ++ ];
    RETS        = [ SP ++ ];
    P1          = [ SP ++ ];
    R0          = [ SP ++ ];
    RTI;                          /* Reenable interrupts via IPEND[4] bit and Return to task */
```

4.03.03 os_cpu_a.asm, OSIntCtxSw()

OSIntCtxSw() is called by OSIntExit() if μC/OS-II determines that there is a more important task to run than the interrupted task. The code that implements OSIntCtxSw() is shown in Listing 4-12.

Listing 4-12, os_cpu_a.asm, OSIntCtxSw ()

```
_OSIntCtxSw:
#if (OS_TASK_SW_HOOK_EN == 1)

    INIT_C_RUNTIME_STACK (0x0)
    CALL _OSTaskSwHook;                             /* (1)   */
    DEL_C_RUNTIME_STACK ()

#endif

    LOADA (P0, _OSPrioCur);                         /* (2)   */
    LOADA (P1, _OSPrioHighRdy);
    R0       = B[ P1 ](Z);                          /* (3)   */
    B[ P0 ] = R0;                                   /* (4)   */
    LOADA(P0, _OSTCBCur);
    LOADA(P1, _OSTCBHighRdy);
    P2       = [ P1 ];                              /* (5)   */
    [ P0 ]   = P2;                                  /* (6)   */

_OSIntCtxSw_modify_SP:

    R0       = [ P2 ];                              /* (7)   */
    R0      += - 8;                                 /* (8)   */
    [ FP ]   = R0;                                  /* (9)   */
    SP      += -4;                                  /* (10) */
    RETI     = [ SP++ ];                            /* (11) */
_OSIntCtxSw.end:
   RTS;                                             /* (12) */
```

L4-12(1) Initialize C Run-Time Stack and call OSTaskSwHook().

L4-12(2) Load OSPrioCur into P0.

L4-12(3) R0 now has OSPrioHighRdy.

L4-12(4) OSPrioCur = OSPrioHighRdy.

L4-12(5) Get stack pointer from OSTCBHighRdy.

L4-12(6) OSTCBCur = OSTCBHighRdy

L4-12(7) Get the new task's SP from OSTCBHighRdy

L4-12(8) The stack pointer must be padded with two stack entries (8 bytes) to simulate the stored values of RETS and FP. However, the actual values of RETS and FP are irrelevant – they are not used.

L4-12(9) Overwrite the stored Frame Pointer of the ISR frame with stack pointer of the new task, essentially 'moving' the ISR C-Runtime stack frame underneath the new task's stack frame. Thus, when the ISR frame is UNLINK'ed, the context restore operations will restore from the new task's stack frame.

L4-12(10) For nested ISRs, interrupts are re-enabled when OSIntCtxSw () returns to OSIntExit () (where OS_EXIT_CRITICAL() restores original IMASK value). However, the context restore process must be uninterruptible – we accomplish this by an artificial stack pop into the RETI register, thus setting the global interrupts disable bit (IPEND[4]). Before the 'fake' stack pop, adjust the stack pointer.

L4-12(11) Popping into the RETI register globally disables interrupts until the executing the next RTI instruction

L4-12(12) Return to OSIntExit() with an RTS.

4.03.04 os_cpu_a.asm, OS_CPU_SR_Save()

OS_CPU_SR_Save() is responsible for saving the IMASK and disabling interrupts, as per the requirements of OS_CRITICAL_METHOD #3. When this function returns, R0 contains the state of the IMASK prior to disabling interrupts.

Listing 4-13, os_cpu_a.asm, OS_CPU_SR_Save()

```
_OS_CPU_SR_Save:

    CLI  R0;

_OS_CPU_SR_Save.end:
    RTS;
```

4.03.05 os_cpu_a.asm, OS_CPU_SR_Restore()

The code in the listing below implements the function that restores the IMASK register and re-enables interrupts for OS_CRITICAL_METHOD #3. When the function is called, it's assumed that R0 contains the desired state of the IMASK register.

Listing 4-14, os_cpu_a.asm, OS_CPU_SR_Restore()

```
_OS_CPU_SR_Restore:

    STI  R0;

_OS_CPU_SR_Restore.end:
    RTS;
```

4.03.06 os_cpu_a.asm, OS_CPU_EnableIntEntry()

This function sets a specific bit of IMASK register to enable interrupt for an interrupt vector corresponding to this bit. When the function is called, it's assumed that R0 contains the desired value of the interrupt vector to enable.

Listing 4-15, os_cpu_a.asm, OS_CPU_EnableIntEntry()

```
_OS_CPU_EnableIntEntry:

    R1   = 1;
    R1 <<= R0;
    CLI    R0;
    R1   = R1 | R0;
    STI    R1;

_OS_CPU_EnableIntEntry.end:

    RTS;
```

4.03.07 os_cpu_a.asm, OS_CPU_DisableIntEntry()

This function clears a specific bit of IMASK register to disable interrupt for an interrupt vector corresponding to this bit. When the function is called, it's assumed that R0 contains the desired value of the interrupt vector to disable.

Listing 4-16, os_cpu_a.asm, OS_CPU_DisableIntEntry()

```
_OS_CPU_DisableIntEntry:

    R1   =  1;
    R1 <<=  R0;
    R1   = ~R1;
    CLI     R0;
    R1   =  R1 & R0;
    STI     R1;

_OS_CPU_DisableIntEntry.end:

    RTS;
```

4.03.08 os_cpu_a.asm, Interrupts Handling

The core event table of ADSP-BF533 processor is 16 entries. It supports nine general-purpose interrupts (IVG7 – IVG15) in addition to the dedicated interrupt and exception events that are described in the Balckfin Processor Programming manual. It is common for applications to reserve the lowest or the two lowest priority interrupts (IVG14 and IVG15) for software interrupts, leaving eight or seven prioritized interrupt inputs (IVG7 – IVG13) for peripheral purposes.

This architecture allows us to assign one ISR to one interrupt vector. Thus, when an interrupt occurs, the processor continues the program execution at the start address of the corresponding ISR. When nesting is required, the ISR must re-enable interrupts after critical section code was executed.

A typical ISR within this port must do the following actions:

```
Save processor registers
Check if OSRunning is set
Increment OSIntNesting (if OSIntNesting < 255)
If OSIntNesting == 1, save SP to TCB
Re-enable Interrupts (push RETI onto stack) if nesting is required /* A */
Save remaining processor context
Servicing the highest priority interrupt.                          /* B */
Call OSIntExit() function.
Restore processor registers.
Return from Interrupt.
```

In the steps above, only actions A and B will differ from an ISR to another one. Thus, we decide to use only two types of ISR, and every interrupt vector will be mapped to one of them when calling OS_CPU_RegisterIntHandler() function, described in the Listing 4-6, depending if nesting is whether or not required. These two ISRs are:

void OS_CPU_NESTING_ISR (void) and **void OS_CPU_NON_NESTING_ISR (void).**

4.03.09 os_cpu_a.asm, OS_CPU_NESTING_ISR()

Listing 4-17, os_cpu_a.asm, OS_CPU_NESTING_ISR()

```
_OS_CPU_NESTING_ISR:

    [ -- SP ] = R0;
    [ -- SP ] = P1;
    [ -- SP ] = RETS;                          /* (1)  */
    R0        = NESTED;                        /* (2)  */
    CALL.X _OS_CPU_ISR_Entry;                  /* (3)  */

    WORKAROUND_05000283()                      /* (4)  */
    INIT_C_RUNTIME_STACK(0x0)                  /* (5)  */

    CALL.X _OS_CPU_IntHandler;                 /* (6)  */
    SP       += -4;                            /* (7)  */
    RETI      = [ SP++ ];                      /* (8)  */
    CALL.X _OSIntExit;                         /* (9)  */
    DEL_C_RUNTIME_STACK()                      /* (10) */
    JUMP.X _OS_CPU_ISR_Exit;                   /* (11) */

_OS_CPU_NESTING_ISR.end:
    NOP;
```

L4-17(1) The RETS register (return address from function call) is saved before calling save context function OS_CPU_ISR_Entry() which is described below. (See Listing 4-19).

L4-17(2) Set R0 value to NESTED (defined in os_cpu.h file) before calling OS_CPU_ISR_Entry().This indicate that the current ISR supports nesting and will re-enable interrupts once critical code is executed.

L4-17(3) Call OS_CPU_ISR_Entry() which performs μC/OS-II specific operations (incrementing OSIntNesting, saving SP to TCB) and saves the processor context. Refer to Listing 4-19 for more details on this routine.

L4-17(4) Refer to Analog Devices Web site for more information about anomaly 05-00-0283.

L4-17(5) Initialize the C-Run Time Stack before calling C function OS_CPU_IntHandler().

L4-17(6) Call OS_CPU_IntHandler() which services the highest priority pending interrupt. Refer to Listing 4-7 for more information.

L4-17(7, 8) Artificial pop of RETI register. It's used to set the global disable interrupt bit. Thus, interrupts are disabled prior to call OSIntExit().

L4-17(9) Call μC/OS-II function OSIntExit().

L4-17(10) Delete C-Run Time Stack created in L4-17(5).

L4-17(11) Jump to OS_CPU_ISR_Exit() which restores the processor context and executes RTI (return from interrupt) instruction. Listing 4-20 shows more details about this routine.

4.03.10 os_cpu_a.asm, OS_CPU_NON_NESTING_ISR()

Listing 4-18, os_cpu_a.asm, OS_CPU_NON_NESTING_ISR()

```
_OS_CPU_NON_NESTING_ISR:

    [ -- SP ] = R0;
    [ -- SP ] = P1;
    [ -- SP ] = RETS;
    R0        = NOT_NESTED;
    CALL.X _OS_CPU_ISR_Entry;

    WORKAROUND_05000283()
    INIT_C_RUNTIME_STACK(0x0)

    CALL.X _OS_CPU_IntHandler;
    CALL.X _OSIntExit;
    DEL_C_RUNTIME_STACK()
    JUMP.X _OS_CPU_ISR_Exit;


_OS_CPU_NON_NESTING_ISR.end:

    NOP;
```

OS_CPU_NON_NESTING_ISR() is not reentrant, thus register RO is set to NOT_NESTED constant before calling OS_CPU_ISR_Entry(). This ISR performs the same operations as OS_CPU_NESTING_ISR() does, but interrupts still disabled until OS_CPU_ISR_Exit() completes and executes RTI instruction which re-enables interrupts.

4.03.11 os_cpu_a.asm, OS_CPU_ISR_Entry()

Listing 4-19, os_cpu_a.asm, OS_CPU_ISR_Entry()

```
_OS_CPU_ISR_Entry:
ucos_ii_interrupt_entry_mgmt:

    [ -- SP ] = R1;                                          /* (1) */
    [ -- SP ] = R2;
    [ -- SP ] = P0;
    [ -- SP ] = P2;
    [ -- SP ] = ASTAT;

    LOADA(P0 , _OSRunning);                                  /* (2) */
    R2        = B [ P0 ] ( Z );
    CC        = R2 == 1;
    IF ! CC JUMP ucos_ii_interrupt_entry_mgmt.end;

    LOADA(P0, _OSIntNesting);                                /* (3) */
    R2        = B [ P0 ] ( Z );
    R1        = 255 ( X );
    CC        = R2 < R1;
    IF ! CC JUMP ucos_ii_interrupt_entry_mgmt.end;
```

```
    R2          = R2.B (X);                                   /* (4) */
    R2         += 1;
    B [ P0 ]    = R2;

    CC          = R2 == 1;                                    /* (5) */
    IF ! CC JUMP ucos_ii_interrupt_entry_mgmt.end;


    LOADA(P2, _OSTCBCur);                                     /* (6) */
    P0          = [ P2 ];
    R2          = SP;
    R1          = 144;                                        /* (7) */
    R2          = R2 - R1;                                    /* (8) */
    [ P0 ]      = R2;

ucos_ii_interrupt_entry_mgmt.end:
    NOP;

interrupt_cpu_save_context:

    CC          = R0 == NESTED;                               /* (9) */
    IF !CC JUMP non_reetrant_isr;

reetrant_isr:
    [ -- SP ] = RETI;                                         /* (10) */
    JUMP save_remaining_context;

non_reetrant_isr:
    R1          = RETI;                                       /* (11) */
    [ -- SP ] = R1;

save_remaining_context:
    [ -- SP ] = (R7:3, P5:3);                                 /* (12) */
    [ -- SP ] = FP;
    [ -- SP ] = I0;
    [ -- SP ] = I1;
    [ -- SP ] = I2;
    [ -- SP ] = I3;
    [ -- SP ] = B0;
    [ -- SP ] = B1;
    [ -- SP ] = B2;
    [ -- SP ] = B3;
    [ -- SP ] = L0;
    [ -- SP ] = L1;
    [ -- SP ] = L2;
    [ -- SP ] = L3;
    [ -- SP ] = M0;
    [ -- SP ] = M1;
    [ -- SP ] = M2;
    [ -- SP ] = M3;
    R1.L        = A0.x;
    [ -- SP ] = R1;
    R1          = A0.w;
    [ -- SP ] = R1;
    R1.L        = A1.x;
    [ -- SP ] = R1;
    R1          = A1.w;
    [ -- SP ] = R1;
    [ -- SP ] = LC0;
    R3          = 0;
    LC0         = R3;
    [ -- SP ] = LC1;
    R3          = 0;
    LC1         = R3;
    [ -- SP ] = LT0;
    [ -- SP ] = LT1;
    [ -- SP ] = LB0;
    [ -- SP ] = LB1;
```

```
    L0        = 0 ( X );
    L1        = 0 ( X );
    L2        = 0 ( X );
    L3        = 0 ( X );

interrupt_cpu_save_context.end:
_OS_CPU_ISR_Entry.end:

    RTS;                                                                 /* (13) */
```

L4-19(1) Save registers used in L4-19(2) through L4-19(8).

L4-19(2) Is `OSRunning` == 1? If not true, don't save SP to OS_TCB.

L4-19(3) Is `OSIntNesting < 255`? If not true, don't save `SP` to `OS_TCB`. **µC/OS-II** supports up to 255 deep nesting.

L4-19(4) Increment `OSIntNesting`.

L4-19(5) Check if `OSIntNesting == 1`. If not true, don't save `SP` to `OS_TCB`.

L4-19(6) Load the current task's `OS_TCB`.

L4-19(7) Cannot save the current `SP` value to `OS_TCB` because save context is not yet completed at this step. Calculate the stack pointer when rest of the registers will be saved (after a complete context save)

L4-19(8) Save the calculated stack pointer (note that the actual `SP` register is not modified during this calculation).

L4-19(9) Check if nesting is required for the ISR which is executing this code.

L4-19(10) Nesting is required for the ISR, thus RETI register is saved onto the stack. The push instruction clears IPEND[4] and re-enables interrupts.

L4-19(11) Nesting is not required for the ISR, thus RETI register is saved onto the stack through R1. IPEND[4] will stay set when saved through R1, and interrupts will stay disabled.

L4-19(12) Save the rest of the processor context.

L4-19(13) Execute a return from function call (`RETS`) to return back to caller.

4.03.12 os_cpu_a.asm,OS_CPU_ISR_Exit()

Listing 4-20, os_cpu_a.asm, OS_CPU_ISR_Exit()

```
_OS_CPU_ISR_Exit:
interrupt_cpu_restore_context:
    LB1          = [ SP ++ ];                         /* (1) */
    LB0          = [ SP ++ ];
    LT1          = [ SP ++ ];
    LT0          = [ SP ++ ];
    LC1          = [ SP ++ ];
    LC0          = [ SP ++ ];
    R0           = [ SP ++ ];
    A1           = R0;
    R0           = [ SP ++ ];
    A1.x         = R0.L;
    R0           = [ SP ++ ];
    A0           = R0;
    R0           = [ SP ++ ];
    A0.x         = R0.L;
    M3           = [ SP ++ ];
    M2           = [ SP ++ ];
    M1           = [ SP ++ ];
    M0           = [ SP ++ ];
    L3           = [ SP ++ ];
    L2           = [ SP ++ ];
    L1           = [ SP ++ ];
    L0           = [ SP ++ ];
    B3           = [ SP ++ ];
    B2           = [ SP ++ ];
    B1           = [ SP ++ ];
    B0           = [ SP ++ ];
    I3           = [ SP ++ ];
    I2           = [ SP ++ ];
    I1           = [ SP ++ ];
    I0           = [ SP ++ ];
    FP           = [ SP ++ ];
    (R7:3, P5:3) = [ SP ++ ];
    RETI         = [ SP ++ ];                        /* (2) */
    ASTAT        = [ SP ++ ];
    P2           = [ SP ++ ];
    P0           = [ SP ++ ];
    R2           = [ SP ++ ];
    R1           = [ SP ++ ];
    RETS         = [ SP ++ ];
    P1           = [ SP ++ ];

interrupt_cpu_restore_context.end:
    R0           = [ SP ++ ];
    RTI;                                             /* (3) */
    NOP;
    NOP;

_OS_CPU_ISR_Exit.end:
    NOP;
```

L4-20(1) Restore context of the new task from the task's stack.

L4-20(2) `RETI` is populated by a stack pop for both nested as well non-nested interrupts.

L4-20(3) Return from interrupt with an `RTI`.

5.00 An Example Application

An example **μC/OS-II**-based application named Ex1 is included with this document. `Ex1` is a multi-threaded application that utilizes the LEDs on the ADSP-BF533-EZKIT LITE evaluation platforms for visual validation of the attached Blackfin port of **μC/OS-II**. Some of the key points about the reference application:

- **Task-level context switch**: Ex1 launches 2 tasks that blink `LED5` and `LED6` on the evaluation platform with varying duty cycles by sleeping for varying ticks through `OSTimeDly()`. These tasks are intended to highlight the task-level context switching (`OSCtxSw()`) of the attached **μC/OS-II** port for the ADSP-BF533.

- **Interrupt-level context switch:** In addition, `Ex1` also launches another task that toggles `LED4` on the EZKIT Lite evaluation platform. However, this task 'pend' on `LED4_semaphore` that is posted in the ISR (`BSP_IVG12_ISR()`) triggered by the push-button switch `PF8` (See Figure 5.1). This ISR introduces asynchronous events into the system. This task is intended to highlight the interrupt-level context switching (`OSIntCtxSw()`) of the attached Blackfin port of **μC/OS-II**.

- **Interrupts and ISRs:** As mentioned above, the push buttons on the evaluation platform are serviced by `BSP_IVG12_ISR()`. This ISR is meant to serve as a reference for applications with ISR's under **μC/OS-II**. Refer to Section 5.02, `BSP_IVG12_ISR()` for more information.

- **Parameter passing:** The three tasks mentioned above are launched from the same task definition (see Section 5.05, `App_LEDTask()` for more information). The task definition expects a parameter that corresponds to an argument list, which itself contains a LED number and a pointer to **μC/OS-II** event, in this case a semaphore. The tasks pend on the semaphore, if it is a non-NULL pointer. This intended to highlight parameter passing to a **μC/OS-II** task.

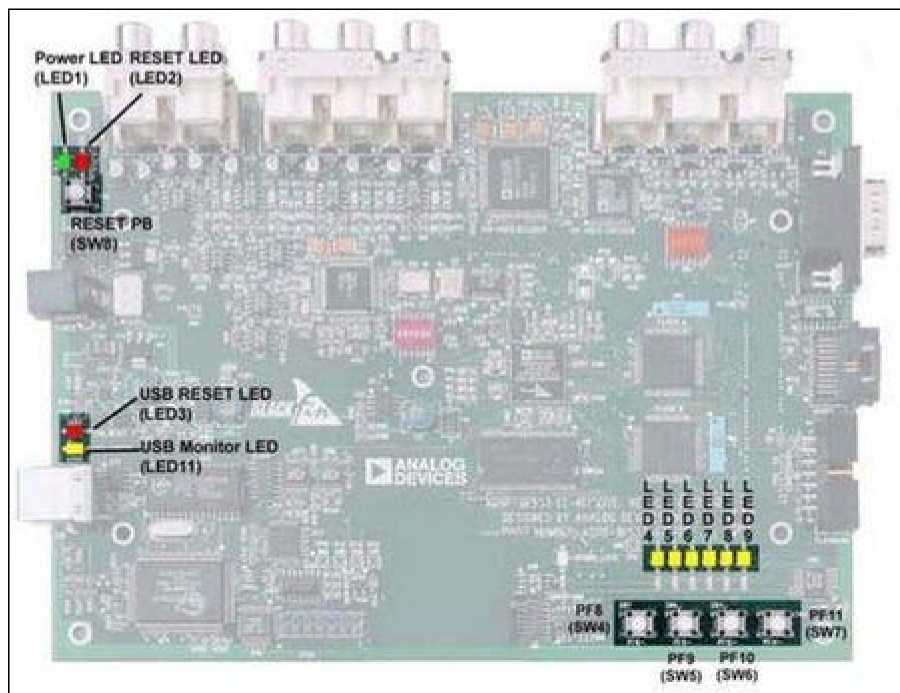
- **Boot Thread:** The application creates a 'boot task' – the first thread created in the application. It enables the ticker interrupt and launches all the other Tasks of the system, thus the name 'boot task'. The boot task is defined in `App_BootTask()` (refer to Section 5.05). After launching other tasks, the boot task suspends itself.

- **μC/Probe:** The application creates another task which is used to output a different information string once push-button switch `PF9` (See Figure 5.1) is pressed. This task 'pend' on `INFO_semaphore` that is posted in the ISR (`BSP_IVG12_ISR()`).

- **Interrupt priority**: In addition to these tasks, the application initializes and enables interrupt for timer0 and timer1. These two timers are configured with the same parameters (See `BSP_InitTimers(), section 5.02`) in order to test the application behavior when two interrupts occur simultaneously.

Listing 5-1 summarizes the various tasks in Ex1 for ADSP-BF533 reference example.

Figure 5.1 User LEDs and push button switches on ADSP-BF533 EZKIT Lite evaluation platform.



Listing 5-1 Various Tasks in Ex1 for ADSP-BF533 EZKIT Lite Evaluation Platform

Task	Priority	Task Definition	Notes
Boot task	5	App_BootTask()	- Installs the Core Timer for µC/OS-II. - Launches all other tasks of application. - Initialize µC/Probe global variables. - Suspends itself after creating other tasks in the system.
LED4 task	32	App_LEDTask()	Toggles LED4 and pends on LED4_semaphore. This is posted by ISR triggered by push-button PF8.
LED5 task	33	App_LEDTask()	Blinks LED5 after sleeping a number of cycles through `OSTimeDly()`
LED6 task	34	App_LEDTask()	Blinks LED6 after sleeping a number of cycles through `OSTimeDly()`
µC/Probe information task	12	App_TaskSer()	Prepares an information string to display on µC/Probe window and pends on INFO_semaphore. This is posted by ISR triggered by push-button PF9.

5.01 bsp.h

bsp.h declares the include files and all functions prototype used in the bsp.c file. It may also contain other defines or constants.

Listing 5-2, bsp.h, Include files

```
#include <ucos_ii.h>
#include <sysreg.h>                                    /* (1) */
#include <sys/platform.h>                              /* (2) */
#include <cpu.h>                                       /* (3) */

#if (uC_PROBE_OS_PLUGIN > 0)                           /* (4) */
#include    <os_probe.h>
#endif

#if (uC_PROBE_COM_MODULE > 0)
#include    <probe_com.h>
#endif

#if (PROBE_COM_METHOD_RS232 > 0)
#include    <probe_rs232.h>
#endif
```

L5-2(1) The processor specific include needed to read cycles register(reg_CYCLES).

L5-2(2) The processor specific include needed to have access to the platform defines.

L5-2(3) This file comes within uC-CPU package. It defines several types and other configurations.

L5-2(4) The include files needed to use **μC/Probe** within the application example.

The rest of the file bsp.h contains the functions prototype. These functions will be detailed in the next section.

5.02 bsp.c

bsp.c provides several functions for initializing peripherals and servicing interrupts. We explain below the functionality for each function whose source code can be consulted in bsp.c file.

```
section( "L1_code" ) void  BSP_Init (void) {}
```

This function re-programs the PLL to run at the Core CLK and System CLK as specified in the application configuration header file app_cfg.h. It also re-programs the SDRAM refresh rate settings for the new clock settings.

The instruction section("L1_code") places the code of BSP_Init() in L1 code memory.

```
void  BSP_CoreTmrInit (void)
```

This function is called to initialize **μC/OS-II**'s tick source (typically a timer generating interrupts every 1 to 100 ms). It also maps the interrupt level 6 (IVG6) to **μC/OS-II** OSTimeTick() function which will be called by the ISR OS_CPU_NESTING_ISR() or OS_CPU_NON_NESTING_ISR() when the Core Timer generates an interrupt.

```
void  BSP_InitTimers (void)
```

This function configures the Timer0 and the Timer1. It also maps Timers interrupt to interrupt vector group 11 (IVG11).

```
void  BSP_IVG11_ISR (void)
```

This function services Timer0 and Timer1 interrupts. It toggles LED7 and LED8 by calling BSP_ToggleLED() function.

```
void  BSP_InitLEDs   (void)
void  BSP_SetLED     (CPU_INT08U lite)
void  BSP_ClrLED     (CPU_INT08U lite)
void  BSP_ToggleLED  (CPU_INT08U lite)
```

These functions are provided to initialize and to control the LEDs on the evaluation platform.

```
void        BSP_InitPushButtons       (void)
void        BSP_PushButton_ISRHandler (CPU_INT08U button_num)
CPU_INT16U  BSP_PushButton_Status     (void)
```

These functions are provided to initialize and to service push-button switches interrupts.

Listing 5-3, bsp.h, BSP_InstallISRs()

This function installs all the default interrupt handlers for the application. Refer to section 4.02 for more information about OS_CPU_RegisterHandler().

```
void BSP_InstallISRs (void)
{
    OS_CPU_RegisterHandler (IVG7,  BSP_IVG7_ISR,  NOT_NESTED);
    OS_CPU_RegisterHandler (IVG10, BSP_IVG10_ISR, NOT_NESTED);
    OS_CPU_RegisterHandler (IVG12, BSP_IVG12_ISR, NESTED);
    OS_CPU_RegisterHandler (IVG11, BSP_IVG11_ISR, NESTED);

}
```

Interrupt level	Nesting	Peripheral	ISR	Handler routine
IVG7	NO	UART Error	OS_CPU_NON_NESTING_ISR()	BSP_IVG7_ISR()
IVG10	NO	UART0/1 RX/TX	OS_CPU_NON_NESTING_ISR()	BSP_IVG10_ISR()
IVG11	YES	Timer0 & Timer1	OS_CPU_NESTING_ISR()	BSP_IVG11_ISR()
IVG12	YES	Push-buttons	OS_CPU_NESTING_ISR()	BSP_IVG12_ISR()

```
void  OSProbe_TmrInit (void)
```

This function selects and initializes a timer for **μC/Probe** plug-in.

```
CPU_INT32U  OSProbe_TmrRd (void)
```

This function is needed by the **μC/Probe** plug -in. It reads the current counts of a 32-bit running timer.

```
CPU_INT32U  BSP_CPU_ClkFreq (void)
```

This function returns the current Core Clock (CCLK) frequency.

```
CPU_INT32U  BSP_SYS_ClkFreq (void)
```

This function returns the current System Clock (SCLK) frequency. The return value is needed by ProbeRS232_InitTarget() function declared in probe_rs232c.c file.

```
void BSP_InitEBIU (void)
```

This function is used to initialize the External Bus interface unit.

5.03 os_cfg.h

This file allows the application user to enable / disable some features of **µC/OS-II**. It is described in detail in *MicroC/OS-II, The Real-Time Kernel, 2nd Edition* (see References, at the end of this document).

5.04 app_cfg.h

µC/OS-II assumes the presence of a file called `app_cfg.h` which is declared in the user application. The purpose of this file is to assign task priorities, stack sizes and other configuration information for the application.

Listing 5-4, app_cfg.h, Application Defines

```
#define  APP_CPU_CFG_BLKFIN_CLKIN_FREQ_MHZ   (27u)                    /* (1) */
#define  APP_CPU_CFG_BLKFN_CORE_FREQ_MHZ    (526u)                    /* (2) */
#define  APP_CFG_OS_ucPROBE_UART_BAUD     (115200u)
#define  APP_CPU_CFG_BLKFN_SYS_FREQ_MHZ     (132u)                    /* (3) */

/*
*********************************************************************************************************
*                                          TASKS PRIORITIES
*********************************************************************************************************
*/

#define  APP_OS_CFG_BOOT_TASK_PRIO          5
#define  OS_PROBE_TASK_PRIO                 6
#define  OS_PROBE_TASK_ID                  OS_PROBE_TASK_PRIO
#define  APP_TASK_SER_PRIO                 12
#define  APP_OS_CFG_LED_TASK_BASE_PRIO     32

/*
*********************************************************************************************************
*                                             STACK SIZES
*                              Size of the task stacks (# of OS_STK entries)
*********************************************************************************************************
*/

#define  APP_OS_CFG_BOOT_TASK_STK_SIZE     256
#define  APP_OS_CFG_LED_TASK_STK_SIZE      256
#define  OS_PROBE_TASK_STK_SIZE            256
#define  APP_TASK_SER_STK_SIZE             256

/*
*********************************************************************************************************
*                                       uC/Probe CONFIGURATION
*********************************************************************************************************
*/

#define  OS_PROBE_TASK                     DEF_TRUE                   /* (4) */
#define  OS_PROBE_HOOKS_EN                 DEF_ENABLED
#define  OS_PROBE_TMR_32_BITS              DEF_ENABLED
#define  OS_PROBE_USE_FP                   DEF_TRUE
```

```
/*
*********************************************************************************************************
*                                           uC/Probe DEFINES
*********************************************************************************************************
*/

#define  uC_PROBE_OS_PLUGIN                 DEF_ENABLED
#define  uC_PROBE_COM_MODULE                DEF_ENABLED
#define  PROBE_RS232_COMM_SEL               PROBE_RS232_UART_0           /* (5) */
```

L5-4(1) The default ADSP-BF533 `CLKIN` frequency is `27 MHz`.

L5-4(2) The new Core clock (`CCLK`) frequency whose value will be set to `526MHz`.

L5-4(3) The new System clock (`SCLK`) frequency whose value will be set to `132MHz`.

L5-4(4) A task will be created for **μC/Probe** plug-in to calculate the task CPU usage and stack usage statistics.

L5-4(5) UART0 is selected for the RS-232 communication between **μC/Probe** and the application running on The Blackfin ADSP-BF533 evaluation platform.

5.05 app_c.c

app_c.c is an application that you can use to verify your **μC/OS-II** and BSP setup. You may wish to use this application as a starting point for your own **μC/OS-II**-based applications on your ADSP-BF533 system. The code in app_c.c is described in the following Listings

Listing 5-5, app_c.c, Typedefs and global variables

```
typedef struct  {                                           /* (1) */
    CPU_INT08U   Led;
    OS_EVENT    *pEvent;
} TASK_PARAMS_LIST;


#if (uC_PROBE_OS_PLUGIN > 0)                                /* (2) */

static  CPU_INT32U  Probe_B1Counts;
static  CPU_INT32U  Probe_B2Counts;
static  CPU_INT08U  Probe_SerOutput[80];
static  CPU_INT32U  Probe_Counts;
        OS_EVENT   *INFO_semaphore;

#endif

static TASK_PARAMS_LIST App_LEDTaskParams[] = {             /* (3) */
        { 4,   (void *)0},
        { 5,   (void *)0},
        { 6,   (void *)0}
};
```

L5-5(1) Declare a structure that will be used to pass parameters to the tasks.

L5-5(2) Declare several global variables which will be monitored by **μC/Probe**.

L5-5(3) Initialize the tasks parameters.

Listing 5-6, `app_c.c`, main()

```
int  main (void)
{
    CPU_INT08U  os_err;


    BSP_Init();                                                          /* (1) */
    printf(("\n\n\n"));
    printf(("Initialize OS...\n"));
    OSInit();                                                            /* (2) */

                                                                         /* (3) */
    OSTaskCreateExt( App_BootTask,
                    (void *)0,
                    (OS_STK *)&App_BootTaskStk[APP_OS_CFG_BOOT_TASK_STK_SIZE - 1],
                     APP_OS_CFG_BOOT_TASK_PRIO,
                     APP_OS_CFG_BOOT_TASK_PRIO,
                    (OS_STK *)&App_BootTaskStk[0],
                     APP_OS_CFG_BOOT_TASK_STK_SIZE,
                    (void *)0,
                     OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);


#if (OS_TASK_NAME_SIZE >= 16)                                            /* (4) */
    OSTaskNameSet(OS_TASK_IDLE_PRIO,          "Idle",       &os_err);
    OSTaskNameSet(APP_OS_CFG_BOOT_TASK_PRIO, "Boot Task", &os_err);
#endif

    printf("Start OS...\n");
    OSStart();                                                           /* (5)  */
}
```

L5-6(1) As described above `BSP_Init()`, which is declared in `bsp.c`, is used re-program the PLL, and setup the Core clock and System clock frequencies..

L5-6(2) `OSInit()` must be called to initialize **μC/OS-II** before you actually use any of the other services (i.e. functions) provided by the operating system.

L5-6(3) `OSTaskCreateExt()` creates a single application task called `App_BootTask()`. This task will be described later.

L5-6(4) `OSTaskNameSet()` provides a means of assigning a name to each task in the application. Task names can be used during debugging. Names can only be assigned to tasks that have been created using `OSTaskCreateExt()`.

L5-6(5) `OSStart()` is called to give control to **μC/OS-II** and start multitasking. `OSStart()` finds the highest priority task of all the tasks created (in this case, it's `App_BootTask()`) and starts executing that task. `OSStart()` never returns.

Listing 5-7, `app_c.c`, `App_BootTask()`

```
static  void  App_BootTask (void  *p_arg)
{
    CPU_INT08U  OS_result;


    (void)p_arg;
    printf("Initialize OS timer...\n");
    BSP_CoreTmrInit();                                              /* (1) */

#if (OS_TASK_STAT_EN > 0)
    printf("Initialize OS statistic task...\n");
    OSStatInit();                                                   /* (2) */
#endif


#if (uC_PROBE_OS_PLUGIN > 0)                                        /* (3) */
    printf("uC/Probe initialization...\n");
#if (uC_PROBE_COM_MODULE > 0)

    ProbeCom_Init();                                                /* (4) */
    ProbeRS232_Init(APP_CFG_OS_ucPROBE_UART_BAUD);                  /* (5) */
    ProbeRS232_RxIntEn();                                           /* (6) */

#endif

    Probe_Counts   = 0;
    Probe_B1Counts = 0;
    Probe_B2Counts = 0;
    Probe_Counts   = 0;
    OSProbe_Init();                                                 /* (7) */
    OSProbe_SetCallback(App_ProbeCallback);                         /* (8) */
    OSProbe_SetDelay(100);                                          /* (9) */

#endif

    printf("Create application tasks...\n");
    printf("\n*************************************************************************");
    printf("\n*                                                                       *");
    printf("\n*   Micrium uC/OS-II Reference Example# 1 for __ADSPBF533__ processor   *");

    printf("\n*                  __ADSPBF533__ on EZKIT Lite Platform                 *");
    printf("\n*                                                                       *");
    printf("\n*************************************************************************");
    printf("\n");

    App_TaskCreate();                                              /* (10) */

    printf("Application ressources initialization...\n");
    App_Init();                                                    /* (11) */

    OS_result = OSTaskSuspend(OS_PRIO_SELF);                       /* (12) */
    if (OS_result) {
        while (1) {
          ;
        }
    }
}
```

L5-7(1) `App_BootTask()` calls `BSP_CoreTmrInit()`(declared in `bsp.c` file) to initialize the core timer which will provide the ticks source for **μC/OS-II**.

L5-7(2) `OSStatInit()` is called to initialize a statistics task, which will determine the percentage of CPU time that is being used by your application. `OSStatInit()` should only be called if `OS_TASK_STAT_EN`, which can be set in `os_cfg.h` file, has a value of `1`. Additionally, `OSStatInit()` must be called after `BSP_CoreTmrInit()`, because it depends on the tick interrupt enabled by that function. Further information about `OSStatInit()` and the methods that it uses to compute CPU usage can be found in the **μC/OS-II** book (see References, at the end of this application note).

L5-7(3) Initialize **μC/Probe** global variables, and call the necessary function to initialize **μC/Probe** environment. Refer to **μC/Probe** user manual for more information.

L5-7(4) `ProbeCom_Init()` is called to initialize the generic communication module for **μC/Probe**

L5-7(5) `ProbeRS232_Init()` is called to initialize the RS-232 communication module. The argument of this function, a 32-bit integer, should be the desired baud rate of the UART.

L5-7(6) `ProbeRS232_RxIntEn()`is called to enable RS-232 receive interrupts. Thus, the target will be able to receive packets from a connected PC which is running **μC/Probe**.

L5-7(7) `OSProbe_Init()`is the **μC/Probe** plug-in initialization function.

L5-7(8) `OSProbe_SetCallback()` sets the user-defined function which will be called by the plug-in task.

L5-7(9) `OSProbe_SetDelay()` sets the delay between successive task CPU usage and stack usage calculations.

L5-7(10) `App_TaskCreate()` creates the remaining tasks in this application example. These tasks are described in the Listing 5-1.

L5-7(11) `App_Init()` completes the remaining initialization (semaphore creation, tasks arguments initialization). It installs the appropriate ISR for each interrupt level group, and completes peripheral settings.

L5-7(12) When the other application tasks are created and application initialization is done, the Boot task will suspend by its self.

Listing 5-8, `app_c.c`, `App_TaskCreate()`

```
static  void  App_TaskCreate (void)
{
    CPU_INT08U  i;


#if (OS_TASK_NAME_SIZE >= 16)
    CPU_INT08U  err;
#endif

    OS_STK* App_LEDTaskStkArr[] = {App_LED_4_TaskStk, App_LED_5_TaskStk, App_LED_7_TaskStk };

    for ( i=0; i<3; i++ ) {                                                 /* (1) */

           err = OSTaskCreateExt(App_LEDTask,
                                (void *)&App_LEDTaskParams[i],
                                (OS_STK *)&(App_LEDTaskStkArr[i][APP_OS_CFG_LED_TASK_STK_SIZE-1]),
                                i+APP_OS_CFG_LED_TASK_BASE_PRIO,
                                i+APP_OS_CFG_LED_TASK_BASE_PRIO,
                                (OS_STK *)&(App_LEDTaskStkArr[i][0]),
                                APP_OS_CFG_LED_TASK_STK_SIZE,
                                (void *)0,
                                (OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR) );
            if (err) {
                while (1) {
                    ;
                }
            }

    }

#if (uC_PROBE_OS_PLUGIN > 0)                                                /* (2) */
         err = OSTaskCreateExt(App_TaskSer,
                                (void *)0,
                                (OS_STK *)&AppTaskSerStk[APP_TASK_SER_STK_SIZE - 1],
                                APP_TASK_SER_PRIO,
                                APP_TASK_SER_PRIO,
                                (OS_STK *)&AppTaskSerStk[0],
                                APP_TASK_SER_STK_SIZE,
                                (void *)0,
                                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
         if (err) {
             while (1) {
                 ;
             }
         }

#if (OS_TASK_NAME_SIZE >= 16)
    OSTaskNameSet(APP_TASK_SER_PRIO, "Probe Str", &err);
#endif

#endif

#if (OS_TASK_NAME_SIZE >= 16)
    OSTaskNameSet(APP_OS_CFG_LED_TASK_BASE_PRIO+0, "LED4 Task", &err);
    OSTaskNameSet(APP_OS_CFG_LED_TASK_BASE_PRIO+1, "LED5 Task", &err);
    OSTaskNameSet(APP_OS_CFG_LED_TASK_BASE_PRIO+2, "LED6 Task", &err);
#endif

}
```

L5-8(1) Create the application tasks which will toggle `LEDs` as described in the Listing 5-1.

L5-8(2) Create the application task which will display a different information string on the **μC/Probe** window. This task is triggered by the push-button switch `PF9`.

Listing 5-9, app_c.c, App_LEDTask ()

```
static  void  App_LEDTask (void  *p_arg)
{
    TASK_PARAMS_LIST  *pTaskParams;
    CPU_INT08U         lite;
    OS_EVENT          *pSem;
    CPU_INT08U         err;


    pTaskParams = (TASK_PARAMS_LIST *)p_arg;
    lite        = pTaskParams->Led;
    pSem        = pTaskParams->pEvent;

    if (pSem != (OS_EVENT*) 0) {                         /* is the semaphore pointer valid?   */

        while(1) {

            OSSemPend(pSem, 0, &err);                    /* pSem is non-NULL, pend on it      */
            BSP_ToggleLED(lite);                         /* toggle LED passed as arg.         */

#if (uC_PROBE_OS_PLUGIN > 0)
            Probe_B1Counts++;                            /* increments Button 1 presses count */
#endif

        }

    } else {

        while(1) {

            OSTimeDly(50+(30*lite));                     /* pSem is NULL, yield the processor */
            BSP_ToggleLED(lite);                         /* toggle LED passed as arg.         */
        }
    }
}
```

The remaining functions, declared in the app_c.c file, are listed below, and are shortly described:

```
static  void  App_Init()
```

This function initializes semaphores and tasks arguments. It also installs the appropriate ISR for each interrupt level group, and completes peripheral settings.

```
static  void  App_TaskSer()
```

This task routine displays a different information string on the **μC/Probe** window. It essentially shows the capability to transmit a string from the application to **μC/Probe** running on your PC. In this application example, the communication between the target and **μC/Probe** uses the RS-232 communication module.

This routine calls the following functions:

- `static void App_DispScr_SignOn():` Copy a simple information string to the string which will be displayed on the **μC/Probe** window.

- `static void App_DispScr_VersionTickRate():` Copy the current **μC/OS-II** version and tick rate value to the string which will be displayed on the **μC/Probe** window.

- `static void App_DispScr_CPU ():` Copy the current CPU usage and speed to the string which will be displayed on the **μC/Probe** window.

- `static void App_DispScr_CtxSw():` Copy the tasks context switch counter to the string which will be displayed on the **μC/Probe** window.

- `static void App_FormatDec():` Convert a decimal value to string.

The following functions, declared in `app_c.c` file, are needed to get **μC/Probe** working correctly. They are called by the corresponding Hook functions declared in `oc_cpu_c.c` file which is a part of the **μC/OS-II** port. These functions are prototyped in `ucos_ii.h` as follows:

- `void App_TaskCreateHook(OS_TCB *ptcb):` This function is called when a task is created.

- `void App_TaskDelHook(OS_TCB *ptcb):` This function is called when a task is deleted.

- `void App_TaskIdleHook(void):` This function is called by `OSTaskIdleHook()`, which is called by the idle task. This hook has been added to allow you to do such things as `STOP` the `CPU` to conserve power.

- `void App_TaskStatHook(void):` This function is called by `OSTaskStatHook()`, which is called every second by **μC/OS-II**'s statistics task. This allows your application to add functionality to the statistics task.

- `void App_TaskSwHook(void):` This function is called when a task switch is performed. This allows you to perform other operations during a context switch.

- `void App_TCBInitHook(OS_TCB *ptcb):` This function is called by `OSTCBInitHook()`, which is called by `OS_TCBInit()` after setting up most of the TCB.

- `void App_TimeTickHook(void):` This function is called every tick.

6.00 Running the Example Application

Once the zip file AN-1533-uCOS-II-v286-BF533.zip is unzipped, you can load the application Ex1.dpj with Visual DSP++ 5.0. This project file is located in the following directory:

\Micrium\Software\EvalBoards\ADI\ADSP-BF533_EZKIT_Lite\VDSP50\OS\EX1.

By default, the needed settings to run **μC/Probe** for this application example are done. The constant uC_PROBE_OS_PLUGIN (declared in app_cfg.h file) allows you to disable the use of **μC/Probe.**

After you have compiled and loaded the application to the target, you can also start **μC/Probe** and load the workspace which comes with the application. This allows you to monitor your **μC/OS-II** based application and to verify the **μC/Probe** capabilities.

Licensing

µC/OS-II source and object code can be used by accredited Colleges and Universities without requiring a license, as long as there is no commercial application involved. In other words, no licensing is required if **µC/OS-II** is used for educational purposes.

You need to obtain an 'Object Code Distribution License' to embed **µC/OS-II** in a product that is sold with the intent to make a profit or if the product is not used for education or 'peaceful' research. For additional details, contact us at Licensing@Micrium.com or by calling us (see Contacts).

Acknowledgements

We would like to thank the following people for their support in making the **µC/OS-II** ADSP-BF533 port possible:

Analog Devices Inc:

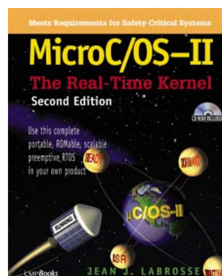
Mr. Bodapati, Deepankar.

Mr. Wu, Jiang.

Mr. Lukasiak, Tomasz.

References

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



2- ADSP-BF533 Hardware Reference, Revision 3.2, July 2006.

3- ADSP-BF53x/BF56x Blackfin Processor Programming Reference, Revision 1.2, February 2007.

4- uC-Probe user Manual.

Contacts

Micriμm
949 Crestview Circle
Weston, FL 33327
USA
+1 954 217 2036
+1 954 217 2037 (FAX)
e-mail: Jean.Labrosse@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

Analog Devices Inc.
One Technology Way, P.O. Box 9106
Norwood, MA 02062-9106 U.S.A.
+1 781.329.4700
+1 781.461.3113 (FAX).
WEB: http://www.analog.com