µC/OS-II

and

The Blackfin Processor

(ADSP-BF5xx)

Application Note

AN-1530

Blackfin Port Release 1.1

Table of Contents

1. Introduction

This application note describes the port of **µC/OS-II** v2.85 for the Blackfin® Processors and is applicable to all ADSP-BF5xx processor variants except the ADSP-BF535. This port was developed and tested with VisualDSP++ 5.0 tool chain (version 5.0, updated August 2007) on these EZ-KIT Lite evaluation platforms: BF533 EZ-KIT Lite, BF537 EZ-KIT Lite, BF561 EZ-KIT Lite, BF548 EZ-KIT Lite, and the B527 EZ-KIT Lite.

> **Note: µC/OS-II** is not a multi-core aware real time operating system (RTOS). On dual core Blackfin processor derivatives (like the ADSP-BF561), **µC/OS-II** can run as a single core RTOS on either or both cores. However, there is no infrastructure provided by the RTOS itself for inter-core synchronization – it is up to the application to implement this feature.

Throughout this document, the notation ADSP-B5xx is used to signify any of the available Blackfin processor variants.

1.1. *µC/OS-II Licensing Policy*

Even though **µC/OS-II** is provided in source form, **µC/OS-II** is not freeware nor is it Open Source software.

1.1.1. Colleges and Universities

µC/OS-II source and object code can be freely distributed (to students) by accredited colleges and universities without requiring a license, as long as no commercial application is involved. In other words, no licensing is required if **µC/OS-II** is used for educational use. Colleges and universities should register their courses by sending a class syllabus and providing a Web link so the class can be added to the Micriμm Web site.

Please send this information to:
Universities@Micrium.com

1.1.2. Commercial Use

You must obtain an Object Code Distribution License to embed **µC/OS-II** in a commercial product. This is a license to put **µC/OS-II** in a product that is sold with the intent to make a profit. A license fee is required for such situations, and you need to contact Micriμm, Inc., (see below) for pricing.

Licensing@Micrium.com

or

Micrium
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.
1-954-217-2036 (Phone)
1-954-217-2037 (Fax)
http://www.Micrium.com

2. Blackfin Background

The following section gives a brief background on the Blackfin processor.

> **Note:** These sections are only excerpts from the Blackfin Technical Documentation Library that are relevant to the Blackfin port of **µC/OS-II**. For complete documentation, please refer to the References Section.

2.1. *The Blackfin Processor*

2.1.1. Introduction

Blackfin processors embody a new type of 16/32-bit embedded processor designed specifically to meet the computational demands and power constraints of today's embedded audio, video, automotive, industrial/instrumentation, and communications applications.

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.

Analog Devices has assembled benchmarks used to test Blackfin processors. Comparative data can be accessed to see how Blackfin processors compare to other manufacturers' parts. This page also contains links to the BDTI Benchmark™ results and Embedded Microprocessor Benchmark Consortium (EEMBC) Web sites:

http://www.analog.com/processors/processors/blackfin/benchmarks/index.html

2.1.2. Core Architecture

The Blackfin processor core 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-BF5xx Blackfin Processor Hardware Reference*.

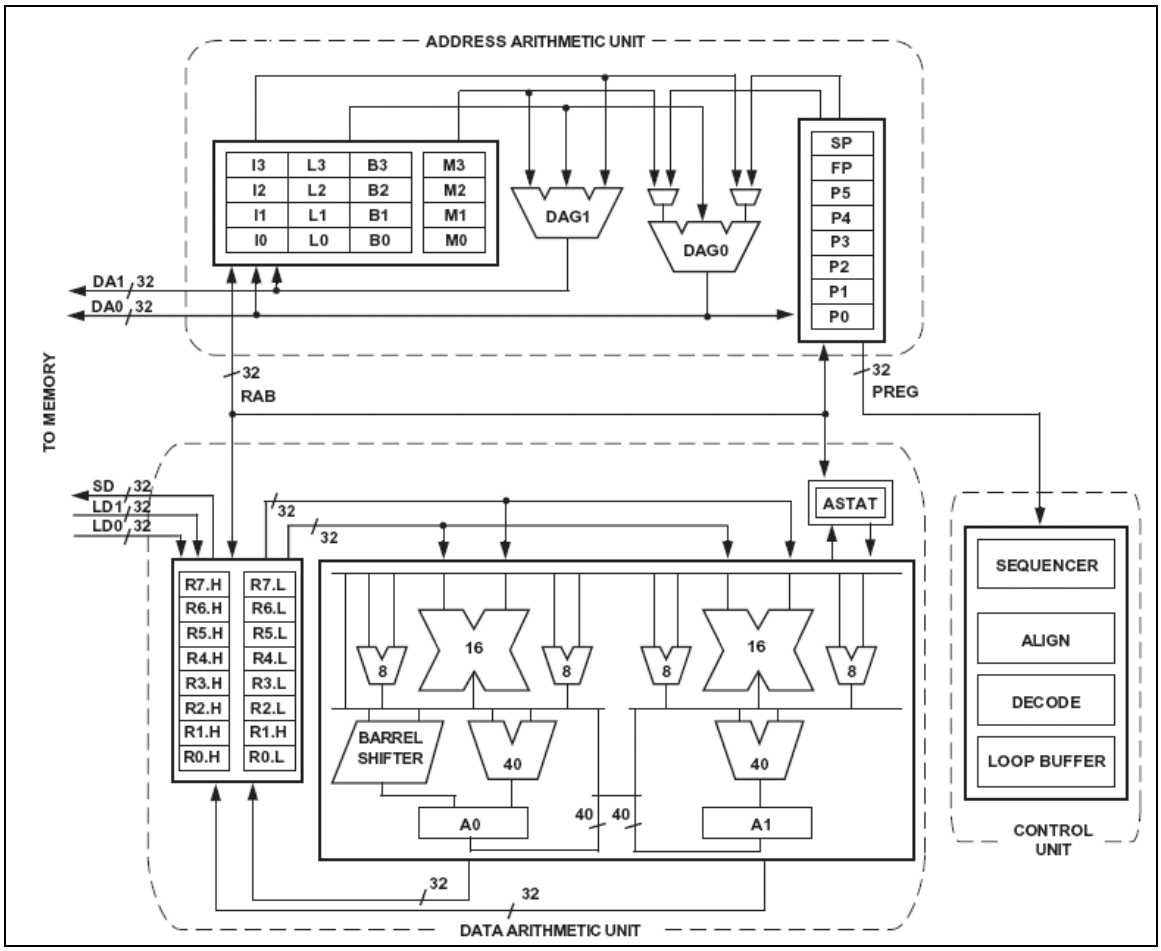


Figure 2-1: Processor Core Architecture

2.1.3. Program Sequencer

In the processor, the program sequencer controls program flow, constantly providing the address of the next instruction to be executed by other parts of the processor. Program flow in the chip is mostly linear, with the processor executing program instructions sequentially. The linear flow varies occasionally when the program uses non sequential program structures, such as those illustrated in Figure 2-2.

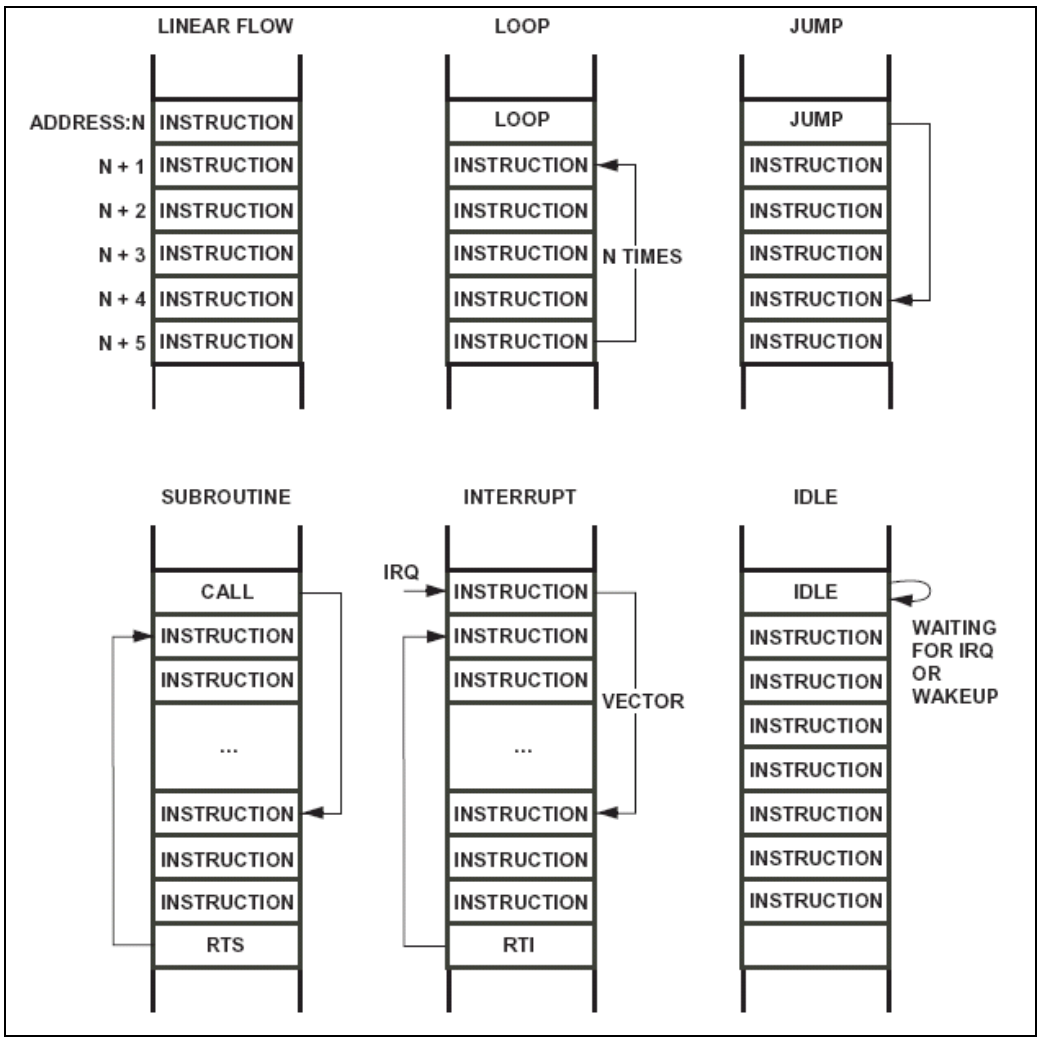


Figure 2-2: Program Flow Variations

Non sequential structures direct the processor to execute an instruction that is not at the next sequential address. These structures include:

- **Loops:** One sequence of instructions executes several times with zero overhead.
- **Subroutines:** The processor temporarily interrupts sequential flow to execute instructions from another part of memory.
- **Jumps:** Program flow transfers permanently to another part of memory.
- **Interrupts and Exceptions:** A runtime event or instruction triggers the execution of a subroutine.
- **Idle:** An instruction causes the processor to stop operating and hold its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

2.1.3.1. Return Address Registers

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.

2.1.4. Subroutines

One type of non-sequential program flow that the sequencer supports is branching. A branch occurs when a JUMP or CALL instruction begins execution at a new location other than the next sequential address.

Subroutines are code sequences that are invoked by a CALL instruction. Assuming the stack pointer SP has been initialized properly, a typical scenario could look like the following:

```
                                /* parent function               */
R0           = 0x1234 (Z);      /* pass a parameter              */
CALL myfunction;
                                /* continue here after the call  */
[P0]         = R0;              /* save return value             */
JUMP somewhereelse;
myfunction:                     /* subroutine label              */
[--SP]       = (R7:7, P5:5);    /* multiple push instruction     */
P5.H         = HI(myregister); /* P5 used locally               */
P5.L         = LO(myregister);
R7           = [P5];            /* R7 used locally               */
R0           = R0 + R7;         /* R0 user for parameter passing */
(R7:7, P5:5) = [SP++];          /* multiple pop instruction      */
RTS;                            /* return from subroutine        */
myfunction.end:                 /* closing subroutine label      */
```

The CALL instruction not only redirects the program flow to the myfunction routine, it also writes the return address into the RETS register. The RETS register holds the address where program execution resumes after the RTS instruction executes. In the example, this is the location that holds the [P0]=R0; instruction.

The return address is not passed to any stack in the background. Rather, the RETS register functions as single-entry hardware stack. This scheme enables “leaf functions” (subroutines that do not contain further CALL instructions) to execute with less possible overhead, as no bus transfers need to be performed.

If a subroutine calls other functions, it must temporarily save the content of the RETS register explicitly. Most likely this is performed by stack operations as shown below.

```
                                /* parent function                */
    CALL function_a;
                                /* continue here after the call   */
    JUMP somewhereelse;
    function_a:                 /* subroutine label               */
    [--SP]       = (R7:7, P5:5); /* opt. multiple push instruction */
    [--SP]       = RETS;        /* save RETS onto stack           */
    CALL function_b;            /* call further subroutines       */
    CALL function_c;
    RETS         = [SP++];      /* restore RETS                   */
    (R7:7, P5:5) = [SP++];      /* opt. multiple pop instruction  */
    RTS;                        /* return from subroutine         */
    function_a.end:             /* closing subroutine label       */
    function_b:
                                /* do something                   */
    RTS;
    function_b.end:
    function_c:
                                /* do something else              */
    RTS;
    function_c.end:
```

2.1.4.1. Stack Variable and Parameter Passing

Many subroutines require input arguments from the calling function and need to return their results. Often, this is accomplished by project-wide conventions, that certain core registers are used for passing arguments, where others return the result. The conventions used by the VisualDSP++ C/C++ compiler are detailed in the Tools section of this application note for details.

Since the stack pointer (SP) is modified inside the subroutine for local stack operations, the frame pointer (FP) is used to save the original state of SP. Because the 32-bit frame pointer itself must be pushed onto the stack first, the FP is four bytes off the original SP value.

The Blackfin instruction set features a pair of instructions that provides cleaner and more efficient functionality than the above example: the LINK and UNLINK instructions. These multi-cycle instructions perform multiple operations that can be best explained by the equivalent code sequences in Figure 2-3.

LINK n;	UNLINK;
[--SP] = RETS; [--SP] = FP; FP = SP; SP += -n;	SP = FP; FP = [SP++]; RETS = [SP++];

Figure 2-3: LINK and UNLINK code sequence

If subroutines require local, private, and temporary variables beyond the capabilities of core registers, it is a good idea to place these variables on the stack as well. The LINK instruction takes a parameter that specifies the size of the stack memory required for this local purpose.

2.1.5. Interrupt Processing

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

> **Note:** For interrupt processing that is specific to the Blackfin port of µC/OS-II (such as writing an interrupt service routine for an application running), refer to Section 4.2.2.1, `IVG12_ISR` in the Example Application section of this application note.

2.1.5.1. 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   */
```

2.1.5.2. Servicing Interrupts

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.

2.1.5.3. 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.

2.1.5.4. 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. O
  nce 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.

2.1.5.5. 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-BFxx 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.

2.2. *Evaluation Platform (EZ-KIT Lite)*

The example application that is included in the attached Blackfin **µC/OS-II** port was tested on these platforms: BF533 EZ-KIT Lite, BF537 EZ-KIT Lite, BF561 EZ-KIT Lite, BF548 EZ-KIT Lite, and the B527 EZ-KIT Lite.

All board specific peripheral setup is included in the following folder in the attached port:

```
\Software\Blocks\ADI\Blackfin\VDSP50\
```

Refer to *ADSP-BF5xx EZ-KIT Lite Evaluation System Manual*s for all the relevant documentation including board schematics. The References section has more information on these manuals.

2.3. *Tools (VisualDSP++)*

The attached Blackfin **µC/OS-II** 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 and Utilities Manual and VisualDSP++ 5.0 Device Drivers and System Services Manual for Blackfin Processors*.

2.3.1. C Run-Time Model and Environment

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.

> **Note:** The context switching routines (`OSCtxSw()` and `OSIntCtxSw()`, see Section 3.3.2.7 and Section 3.3.2.8 for more information) in the Blackfin **µC/OS-II** port are written in assembly and are linked to compiler-generated C routines. Thus, the context switching routines are compliant with the C Run-Time Model described herein.

2.3.1.1. 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	Including CC
A0-A1	
I0-I3	
B0-B3	
M0-M3	

- **Call Preserved Registers:** The call preserved registers must be saved and restored if they are modified within the assembly function; if a function does not change a particular register, it does not need to save and restore the register. These registers are:
  P3–P5
  R4–R7

2.3.1.2. 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 2-4 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().`

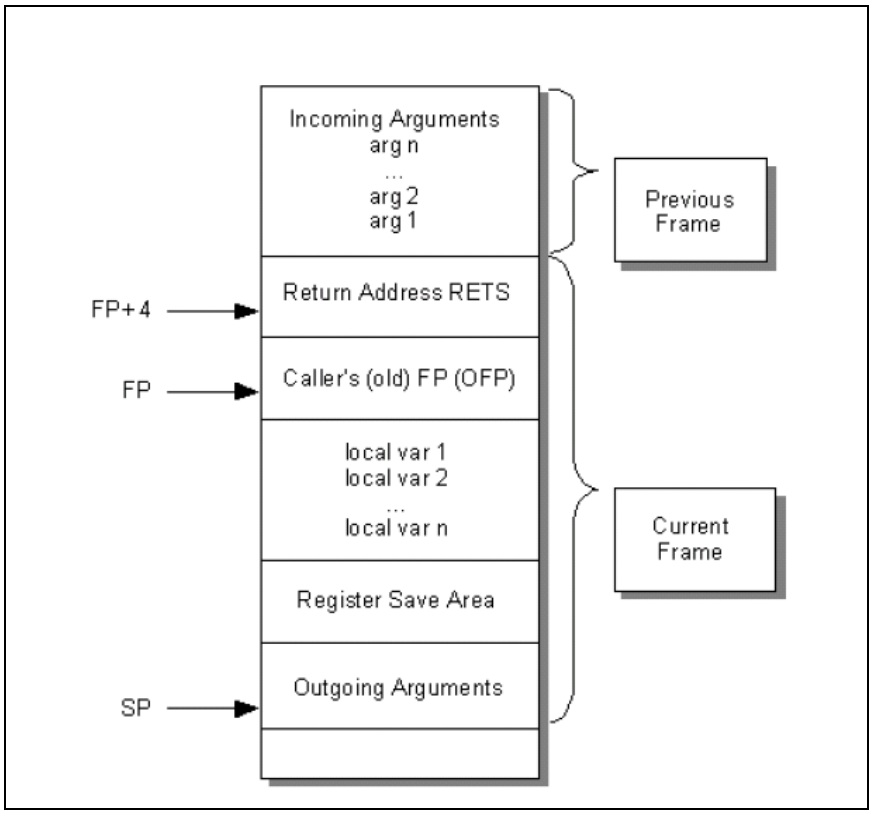


Figure 2-4: Example Run-Time stack

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, 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**: 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. The maximum stack frame size is $2^{18} = 262152$ bytes in 4-byte increments.
- **UNLINK**: 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.

Thus, a typical compiler-generated function prologue would be:

```
LINK 16;
[--SP]=(R7:4);
SP +  = –16;
[FP+8]=R0; [FP+12]=R1; [FP+16]=R2;
```

Where:

`LINK 16;`
is a special linkage instruction that saves the return address and the frame pointer, and updates the Stack Pointer to allocate 16 bytes for local variables.

`[--SP]=(R7:4);`
allocates space on the stack and saves the registers in the save area.

`SP + = –16;`
allocates space on the stack for outgoing arguments. Always allocate at least 12 bytes on the stack for outgoing arguments, even if the function being called requires less than this.

`[FP+8]=R0; [FP+12]=R1; [FP+16]=R2;`
saves the argument registers in the argument area.

A matching compiler-generated function epilogue would be:

```
SP +   = 16;
P0     = [FP+4];
(R7:4) = [SP++];
UNLINK;
JUMP (P0);
```

Where:

`SP += 16;`
reclaims the space on the stack that was used for outgoing arguments.

`P0 = [FP+4];`
loads the return address into register P0.

`(R7:4) = [SP++];`
restores the registers from the save area and reclaims the area.

`UNLINK;`
is a special instruction that restores the frame pointer and stack pointer.

`JUMP (P0);`
returns to the caller.

2.3.1.3. Transferring 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.

2.3.1.3.1. 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.

2.3.1.3.2. 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 callee as a hidden argument in register `P0`.

2.3.2. Device Drivers/System Services

The System Services form a collection of functions that are commonly found in embedded systems. Each system service focuses on a specific set of functionality such as Direct Memory Access (DMA), Power Management (PM) and Interrupt Control (IC). Collectively, the system services provide a wealth of pre-built, optimized code that simplifies software development for users, allowing them to get their Blackfin processor-based designs to market more quickly.

Similarly, a set of Device Drivers are available to provide a simple, clean and familiar interface into the Blackfin processor. The primary objective of the device driver model is to create a concise, effective and easy to use interface through which applications can communicate with device drivers. Secondarily, the model and device manager software, significantly simplifies the development of device drivers, making it very straightforward for the development of new device drivers.

> **Note:** The attached Blackfin port of **µC/OS-II** is identical for applications that conform to the DD/SSL model as well as those that do not (standalone applications). In fact, it is possible for the same application to service certain interrupt levels through DD/SSL model, while others interrupt levels are serviced through standalone ISRs written by the application.

Applications using the Device Drivers/System Services architecture require System Services libraries targeted for the **µC/OS-II** operating environment in installation directory for VisualDSP++:

`<VisualDSP++ install path>\Blackfin\lib\libsslxxx_uCOS.dlb`

where xxx represents the processor variant

and

uCOS suffix that these libraries are targeted for **µC/OS-II**

The default Linker Description File (`.ldf` file) of has to be **modified** to notify the VisualDSP++ Linker to link the application with System Services library files targeted for **µC/OS-II** (and **not the standalone version of these libraries**). All **µC/OS-II** applications wishing to use Device Drivers and System Services **must modify** the respective `.ldf` files as shown here. For more information on .ldf files, refer to *The VisualDSP++ 5.0 Linker and Utilities Manual*.

These modifications must be placed within the "*`$VDSG<insert-user-libraries-at-beginning>`*" comments to preserve them across auto-generations of the `.ldf` file (VisualDSP++ 5.0 based projects can choose for auto-generated `.ldf` files). Inserting the name of System Services Libraries targeted for **µC/OS-II** is the **ONLY** modification required, as shown in Listing 4-31.

> **Note:** The VisualDSP++ Linker links libraries in the order specified in the `.ldf` file. Thus, the user must place name of the **µC/OS-II** specific System Services Library within "*`$VDSG<insert-user-libraries-at-beginning>`*" comments and **not** the "*`$VDSG<insert-user-libraries-at-end>`*" comments.

Listing 2-1 details the `.ldf` file modifications required of all **µC/OS-II** based projects using Device Drivers/System Services Libraries.

Listing 2-1: Required modification of `.ldf` files for applications using DD/SSL under µC/OS-II

```
$LIBRARIES =

/*$VDSG<insert-user-libraries-at-beginning>                        */
/* Text inserted between these $VDSG comments will be preserved */
      RT_LIB_NAME(ssl532_uCOS),                                 // (1)
/*$VDSG<insert-user-libraries-at-beginning>                        */

   RT_LIB_NAME_MT(small532)
   ,RT_LIB_NAME_MT(io532)
   ,RT_LIB_NAME_MT(c532)
   ,RT_LIB_NAME_MT(event532)
   ,RT_LIB_NAME_MT(x532)
   ,RT_LIB_NAME_EH_MT(cpp532)
   ,RT_LIB_NAME_EH_MT(cpprt532)
   ,RT_LIB_NAME(f64ieee532)
   ,RT_LIB_NAME(dsp532)
   ,RT_LIB_NAME(sftflt532)
   ,RT_LIB_NAME(etsi532)
   ,RT_LIB_NAME(ssl532)
   ,RT_LIB_NAME(drv532)
   ,RT_OBJ_NAME_MT(idle532)
   ,RT_LIB_NAME_MT(rt_fileio532)

/*$VDSG<insert-user-libraries-at-end>                              */
/* Text inserted between these $VDSG comments will be preserved */
/*$VDSG<insert-user-libraries-at-end>                              */
   ;
```

L4-31 (1): Place "**RT_LIB_NAME**(ssl532_uCOS)," within the "*$VDSG<insert-user-libraries-**at-beginning**>*" comments. This will ensure that VisualDSP++ Linker links with System Services Libraries targeted for **µC/OS-II** operating environment.

2.3.3. VisualDSP++ Compiler Extensions

2.3.3.1. Built-in Functions

VisualDSP++ Compiler allows for built-in functions that allows access to system facilities on the Blackfin Processors such as the Core Interrupt Mask Register (`IMASK`). Thus, the attached Blackfin port of **µC/OS-II** uses the `OS_CRITICAL_METHOD#3` – obtain current value of processor status word (`IMASK`) and save it into a local variable declared within a C function.

> **Note:** The compiler built-in functions are defined in the `ccblkfn.h` header file in the system include directory:
>
> ```
> <VDSP++ Install path>\Blackfin\include
> ```
>
> Thus, it is necessary to include the `ccblkfn.h` file before using these functions to avoid unresolved symbols errors at link time.

2.3.4. VisualDSP++ Linker

2.3.4.1. Elimination

The VisualDSP++ Linker supports Code and Data Elimination to reduce the memory requirements of the executable. When enabled, the linker eliminates removes symbols from the executable file if they are not called and provides reports on each eliminated symbol. From the VisualDSP++ GUI, this feature can be enabled through the Elimination Tab in 'Link' category of 'Project Options' for each project. In addition, the -ev command line allows the same functionality. This feature is enabled by default on all example project that come with the attached Blackfin port of **µC/OS-II**.

3. µC/OS-II Blackfin Port

3.1. *Overview*

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

- **µC/OS-II port guidelines** – The attached port closely follows the port described in the book by Jean J. Labrosse (2002), *MicroC/OS-II – The Real Time Kernel*. New York: CMP Books. The functions `OSStartHighRdy()`, `OSTickISR()`, `OSTaskStkInit()` follow the guidelines in the book. However all context switch related routines – `OS_TASK_SW()`, `OSCtxSw()` and `OSIntCtxSw()` deviate slightly from the book's recommendations owing to architectural differences between Blackfin's architecture and the architectural assumptions of the target processor by the porting guidelines. See Section 3.3.2.7 `OSCtxSw()` and 3.3.2.8 `OSIntCtxSw()` for more information.
- **Nested Interrupts –** The port supports nested interrupts for all interrupt levels, from Interrupt Level 6 (`IVGTMR`) 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 `IVGTMR` through `IVG13`, nesting can be enabled on any, some, or all interrupt levels, independent of the interrupt behavior on other interrupt levels. For more information, refer to Section 3.3.1.6, **µC/OS-II** Nested Interrupts Support.
- **Device Drivers/System Services Libraries –** Refer to Section 2.3.2 for more information on release date of System Services Libraries targeted for **µC/OS-II**. Note that the attached Blackfin port is identical for applications that conform to the DD/SSL model as well as those that do not (standalone applications). In fact, it is possible for the same application to service certain interrupt levels through DD/SSL model, while others interrupt levels are serviced through standalone ISRs written by the application.

3.1.1. Reserved Resources

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 Blackfin Core Timer (Interrupt Level 6 `IVGTMR`) 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**. For more information, refer to Section 3.3.2.9, `OSTickISR()`.
- **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 3.3.2.7, `OSCtxSw()`.
- **Memory Footprint** – A few preliminary measurements were made to calculate the memory usage (code and data) of **µC/OS-II** based applications on Blackfin processors with the VisualDSP++ tool chain – refer to Listing 3-1 for more information. These metrics are listed in the embedded Excel spreadsheet below.

  > **Note:** The metrics listed here only provide an approximation of the memory requirements of **µC/OS-II** based applications. These results could vary significantly for other applications, especially when the Elimination feature of VisualDSP++ Linker is enabled – refer to Section 2.3.4.1, Elimination for more details on this feature.

Listing 3-1: µC/OS-II Blackfin footprint (embedded Excel object)

µC/OS-II Blackfin footprint

Notes		ROM (Bytes)	RAM (Bytes)		
			Global	Heap	**Total**
Blackfin specfic Port		**600**	4		**4**
Scheduler, Idle Task ONLY		**1602**	353	400	**753**
common event code	With Events	432	9	--	--
common flag code	With Flags	414	7	--	--
common Queues code	With Queues	72	24	--	--
common stat code	With Stat	306	22	400	--
string copy, mem copy	Internal	276	--	--	--
	OS Services				
advanced task	Adv. Tasks	1936	--	400	--
	Sem	900	--	--	--
	Mutex	1714	--	--	--
	Mbox	1016	--	--	--
	Queue	1488	--	--	--
	Flag	1480	--	--	--
Alternative to malloc	Mem	340	20	--	--
Total:					
Typical Application with complete µC/OS-II services enabled (20 tasks with 400 bytes of stack space each)		**12976**	561	8000	**8561**

*** All footprint numbers in bytes, OS_ARG_CHK_EN=0**
*** VisualDSP++ 4.5 Compiler optimization set to 100% Speed**

(Later versions of the toolchain should yield very similar results)

3.1.2. 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. This section details the convention followed by the attached Blackfin port of **µC/OS-II** for an interrupted task's stack frame.

Figure 3-1 shows a typical context save of the attached Blackfin port of **µC/OS-II**.

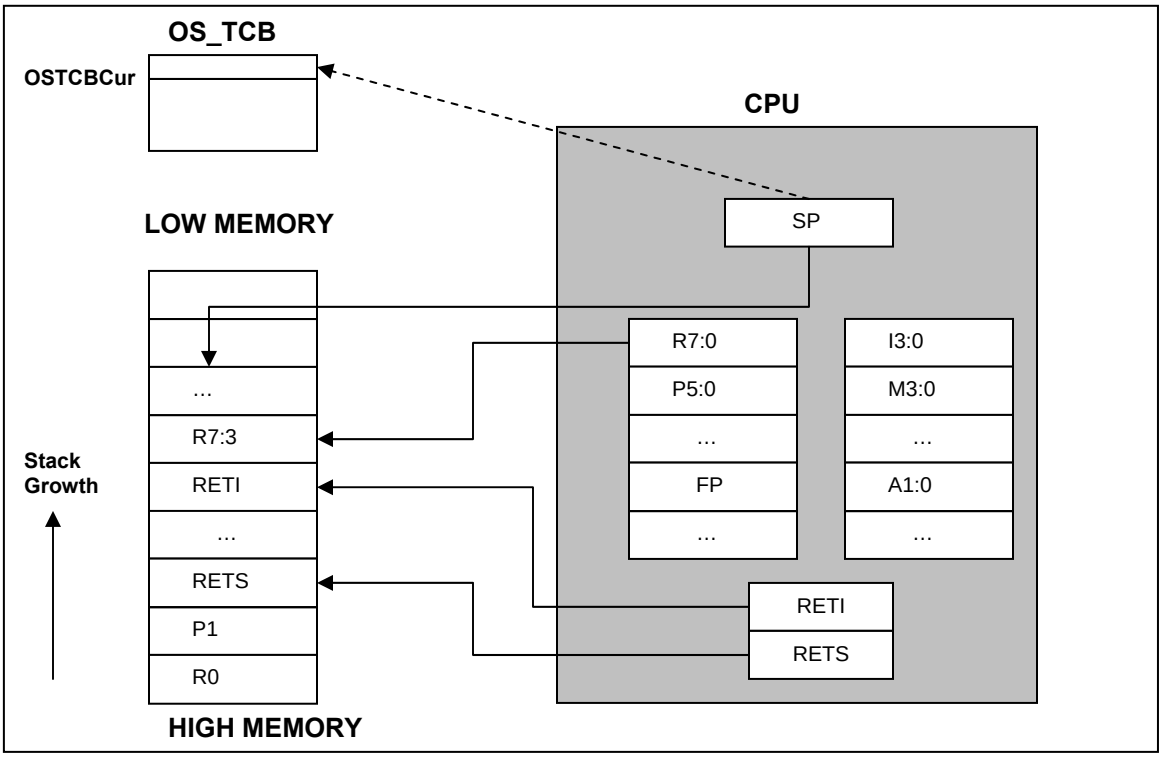


Figure 3-1: Blackfin µC/OS-II port context save – refer to Listing 3-2 for more details

Listing 3-2 gives complete details on the layout of an interrupted task after the context save is complete.

Listing 3-2: Blackfin µC/OS-II port Task Stack Frame – this represents the interrupted task's stack after the context save is complete

Offset	Register	Notes
(** High Memory **)	N/A	See L3-2 (1) below
0x0	R0	See L3-2 (3) below
0x4	P1	See L3-2 (3) below
0x8	RETS	Return from Function Call, See L3-2 (2) below
0xC	R1	See L3-2 (3) below
0x10	R2	See L3-2 (3) below
0x14	P0	See L3-2 (3) below
0x18	P2	See L3-2 (3) below
0x1C	ASTAT	Arithmetic Status – See L3-2 (3) below
0x20	RETI	Return from Interrupt – See L3-2 (3) below
0x34	R7:3	Registers R7 through R3, R7 is lower than R3
0x44	P5:3	Registers P5 through P3, P5 is lower than R3
0x48	FP	Frame Pointer
0x58	I3:0	DAG Index Address, I3 is lower than I0
0x68	B3:0	DAG Base Value, B3 is lower than B0
0x78	L3:0	DAG Length Address, L3 is lower than L0
0x88	M3:0	DAG Modify Value, M3 is lower than M0
0x98	A0:1	Accumulators (40-bit), A1 is lower than A0
0xA0	LC1:0	Loop Count, LC1 is lower than LC0
0xA8	LT1:0	Loop Top, LT1 is lower than LT0
0xB0	LB1:0	Loop Bottom, LB1 is lower than LB0
(** Low Memory **)	N/A	See L3-2 (4) below

L3-2 (1): The SP points to this location at the moment that interrupt occurs (hardware or software) i.e. before the context save has started. As the stack grows with the context saved, SP grows downward from this point.

L3-2 (2): The RETS register (return address from function call) is saved before calling the context save routines. Refer to Section 3.3.2.3 and 3.3.2.4 for more information on these routines.

L3-2 (3): For more information on the out-of-order saving of the registers R0:2, P0:2 and ASTAT, refer to Section 3.3.2.3.

L3-2 (4): This is stack location that is saved into OSTCBCurrRdy-->OSTCBStkPtr.

3.2. *Directories and Files*

The code and documentation of the port are placed in a directory structure according to Micrium Application Note, AN-2002 "**µC/OS-II** *Directory Structure"*.

Application notes are located in the following directory:

```
\Micrium\AppNotes
```

All processor independent files for **µC/OS-II** are located in the following directory. The port has been tested with **µC/OS-II** v2.85 and higher.

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

All processor dependent files (`os_cpu.h, os_cpu_a.asm, os_cpu_c.c`) for the VisualDSP++ tool chain (version 5.0, updated August 2007) are located in the following directory.

```
\Micrium\Software\uCOS-II\Ports\Blackfin\VDSP50\
```

All **µC/OS-II** specific documentation can be found in the following directory.

```
\Micrium\Software\uCOS-II\Doc
```

The attached Blackfin port of **µC/OS-II** comes with reference applications located in the following directories:

- `Micrium\Software\EvalBoards\ADI\ADSP-BF5xx_EZKIT_Lite\VDSP50\OS` for the various ADSP-BF5xx EZ-KIT Lite evaluation platforms

All platform dependent source code is located in:

- `\Micrium\Software\Blocks\ADI\Blackfin\VDSP50\ADSP-BF5xx_EZKIT_Lite\bsp.c(and .h)` for the various ADSP-BF5xx EZ-KIT Lite evaluation platforms

3.3. *Port Files*

3.3.1. OS_CPU.H

Note: `OS_CPU.H` is included in both assembly files as well as C files (such as `OS_CPU_A.ASM` and `OS_CPU_C.C`). Thus, it is necessary to differentiate portions of `OS_CPU.H` for assembly files and C files to avoid compiler/assembler syntactical errors. This is made possible through the use of pre-defined macros, `_LANGUAGE_ASM` and `_LANGUAGE_C`.

Listing 3-3: Using pre-defined assembler macros

```
#if defined(_LANGUAGE_ASM)        // (1)
```

L3-3 (1): The VSDP++ assembler always sets _LANGUAGE_ASM to 1 while the _LANGUAGE_C is set to 1 when the VisualDSP++ C compiler calls use it to specify headers.

3.3.1.1. Miscellaneous Macros

The definitions for TRUE and FALSE are defined for legacy reasons.

Listing 3-4: Defining TRUE and FALSE

```
#if (!defined (FALSE))
#define  FALSE                          false      // (1)
#endif
#if (!defined (TRUE))
#define  TRUE                           true       // (2)
#endif
```

L3-4 (1): Defining FALSE for legacy reasons

L3-4 (2): Defining TRUE for legacy reasons

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 3-5: Defining P-register 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)
```

L3-5 (1): Load Immediate into register

L3-5 (2): Load Address into register

3.3.1.2. External Global Variables

`OS_CPU_A.ASM` accesses various global variables defined in the processor independent source files of **µC/OS-II**. These are declared as externs here.

Listing 3-6: Global variables

```
.extern _OSTaskSwHook;
.extern _OSIntEnter;
.extern _OSIntExit;
.extern _OSTimeTick;
.extern _OSTimeTickHook;
.extern _OSIntNesting;
.extern _OSPrioCur;
.extern _OSPrioHighRdy;
.extern _OSRunning;
.extern _OSTCBCur;
.extern _OSTCBHighRdy;
.extern __uCOS_II_reentrant_interrupt_entry;
.extern __uCOS_II_reentrant_interrupt_exit;
.extern __uCOS_II_non_reentrant_interrupt_entry;
.extern __uCOS_II_non_reentrant_interrupt_exit;
.extern __uCOS_II_exception_entry;
.extern __uCOS_II_exception_exit;
```

3.3.1.3. Data Types

Listing 3-7: Data Types (Compiler Specific)

```
#ifndef _OS_TYPES
typedef unsigned int   BOOLEAN;
typedef unsigned char  INT8U;       // (1)
typedef signed   char  INT8S;
typedef unsigned short INT16U;      // (2)
typedef signed   short INT16S;
typedef unsigned int   INT32U;      // (3)
typedef signed   int   INT32S;
typedef float          FP32;        // (4)
typedef double         FP64;
#define _OS_TYPES
#endif

typedef INT32U         OS_STK;      // (5)
typedef INT32U         OS_CPU_SR;   // (6)
```

L3-7 (1): The VisualDSP++ Compiler supports an 8-bit unsigned integer for an `unsigned char`

L3-7 (2): The VisualDSP++ Compiler supports a 16-bit unsigned integer for an `unsigned short`

L3-7 (3): The VisualDSP++ Compiler supports a 32-bit unsigned integer for an `unsigned int`

L3-7 (4): The VisualDSP++ Compiler supports a 32-bit unsigned integer for a `float`

L3-7 (5): Blackfin hardware supports a 32-bit wide stack

L3-7 (6): 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 (see Section 3.3.1.10, **µC/OS-II** Critical Sections). In fact, this port only supports `OS_CRITICAL_METHOD#3` because it's the preferred method for **µC/OS-II** ports.

3.3.1.4. ADI DD/SSL Library build definitions

Note: The following section is only relevant for users who wish to re-build the DD/SSL Libraries. For information on building µC/OS-II applications with DD/SSL, refer to Section 4 Reference Examples.

The DD/SSL libraries are distributed as part of the VisualDSP++ tool chain. In addition, the VisualDSP++ too chain also comes with the complete source code and include files necessary to rebuild System Services Libraries to provide the user the ability to tailor the libraries to their particular needs.

Refer to Section 2.3.2 for more information on release date of System Services Libraries targeted for µC/OS-II.

To aid in re-building System Services targeted for µC/OS-II operating environment, the following definitions are defined in `OS_CPU.H`.

Listing 3-8: Definition to build DD/SSL libraries targeted for µC/OS-II.

```
#if (!defined(ADI_SSL_UCOS))
#define ADI_SSL_UCOS          // (1)
#endif
#if (!defined(ADI_DEV_UCOS))
#define ADI_DEV_UCOS          // (2)
#endif
```

L3-8 (1): With this definition, the System Services Library re-build is targeted for µC/OS-II operating environment.

L3-8 (2): With this definition, the Device Driver Library re-build is targeted for µC/OS-II operating environment.

3.3.1.5. Blackfin C Run-Time stack/frame macros

The attached Blackfin port of **µC/OS-II** supports calling compiler generated C functions from within the ISR, allowing the application to write most of the ISR code in C. In addition, any application under the DD/SSL model must support calling C functions from within an ISR, thus requiring the setup C Run-Time stack within an ISR. The following macros create and destroy C Run-Time stacks. For more information on `LINK` and `UNLINK`, refer to Section 2.3.1.2, Managing the Stack).

Listing 3-9: Macros to build/destroy C Run-Time stacks

```
#define INIT_C_RUNTIME_STACK(frame_size)  \  // (1)
      LINK frame_size;                    \
      SP += -12 ;                            // (2)
#define DEL_C_RUNTIME_STACK()             \  // (3)
      UNLINK ;
```

L3-9 (1): `INIT_C_RUNTIME_STACK` creates a frame of size = `frame_size` parameter. The application passes this parameter when it calls the ISR Prolog (see Section 3.3.1.7, **µC/OS-II** ISR Prolog and Epilog macros)

L3-9 (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 2.3.1, C Run-Time Model and Environment.

L3-9 (3): `UNLINK` destroys the frame created in L3-9(1)

3.3.1.6. µC/OS-II Nested Interrupts Support

For general information on the Blackfin architecture and Interrupt Processing, see Section 2.1.5, Interrupt Processing.

The port supports nested interrupts for all interrupt levels, from Interrupt Level 6 (`IVGTMR`) 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, `IVG14` is always non-nested. For interrupt levels from `IVGTMR` through `IVG13`, nesting can be enabled on any, some, or all interrupt levels, independent of the interrupt behavior on other interrupt levels.

Under the DD/SSL model, the Interrupt Control module generates ISR code and thus, the application chooses nested nature of each interrupt level through the DD/SSL API. For more information, refer to *The Device Drivers and System Services Manual for Blackfin Processors*.

For non-DD/SSL enabled applications, the application decides the nature of each interrupt level by calling the appropriate interrupt entry macro – `UCOS_II_REENTRANT_ISR_PROLOG()` for an reentrant (nested) ISR and `UCOS_II_NON_REENTRANT_ISR_PROLOG()` for a non-reentrant (non-nested) ISR.

The definitions in Listing 3-10 help identify the nature of the ISR. As an example, nature of the Core Timer ISR (`IVGTMR`) is defined herein.

Listing 3-10: IVGTMR Nested interrupts support

```
#define OS_ADI_Reentrant_ISR                0xad  // (1)
#define OS_ADI_NonReentrant_ISR             0xda  // (2)
#define OS_ADI_IVGTMR_ISR_Type              OS_ADI_Reentrant_ISR  // (3)
```

L3-10 (1): Definition to identify a reentrant ISR

L3-10 (2): Definition to identify a non-reentrant ISR

L3-10 (3): `OS_ADI_IVGTMR_ISR_Type` defines the ISR type of `IVGTMR` (Core Timer). Based on this definition, `OSTickISR()` in `OS_CPU_A.ASM` calls the appropriate macro. See Section 3.3.2.9, `OSTickISR` for more information. Change `OS_ADI_IVGTMR_ISR_Type` to `OS_ADI_NonReentrant_ISR` to disable nesting for `IVGTMR`.

3.3.1.7. µCOS-II ISR Prolog and Epilog macros

The **µC/OS-II** ISR Prolog and Epilog macros were defined to provide the convenience of a single call when writing ISRs for applications that do not use the DD/SSL model.

The application calls either –

- `UCOS_II_REENTRANT_ISR_PROLOG()/EPILOG` for reentrant ISRs (nested).

OR

- `UCOS_II_NON_REENTRANT_ISR_PROLOG()/EPILOG` for non-reentrant ISRs (non-nested).

Listing 3-11 on the next page details the prolog/epilogs for reentrant ISRs

Listing 3-11: UCOS_II_REENTRANT_ISR_PROLOG()/EPILOG

```
#define UCOS_II_REENTRANT_ISR_PROLOG(frame_size)\          // (1)
      [ -- SP ] = R0 ;                           \          // (2)
      [ -- SP ] = P1 ;                           \
      [ -- SP ] = RETS ;                         \          // (3)
      CALL __uCOS_II_reentrant_interrupt_entry; \          // (4)
      CC = R0 == R0 ;                            \          // (5)
      P0.L = 0x14 ;                              \
      P0.H = 0xffc0 ;                            \
      IF CC JUMP 4 ;                             \
      R0 =  [ P0 ] ;                             \
      INIT_C_RUNTIME_STACK(frame_size)                     // (6)

#define UCOS_II_REENTRANT_ISR_EPILOG()             \          // (7)
      CALL _OSIntExit;                           \          // (8)
      DEL_C_RUNTIME_STACK()                      \          // (9)
      JUMP __uCOS_II_reentrant_interrupt_exit;             // (10)
```

L3-11 (1): Applications installing reentrant interrupt handlers call this macro for an ISR prolog with the `frame_size` parameter (this parameter is the size of C Run-Time stack frame of the ISR to be created, any value between zero and 262152 bytes, including both values in increments of 4). This parameter represents the local stack storage requirements of the ISR i.e. local variables, register saves etc. The typical value of `frame_size` is 0.

L3-11 (2): This is the first instruction to execute when the processor vectors to the ISR. `R0` and `P1` are pushed onto the stack first to allow interoperability with DD/SSL libraries with a single **µC/OS-II** port for Blackfin (DD/SSL libraries always push `R0` and `P1` onto stack as part of their ISRs prolog).

L3-11 (3): The `RETS` register (return address from function call) is saved before calling the reentrant ISR context save routine, `__uCOS_II_reentrant_interrupt_entry`. Refer to Section 3.3.2.3 for more information on this routine.

L3-11 (4): Call `__uCOS_II_reentrant_interrupt_entry` routine which performs **µC/OS-II** specific ISR operations (incrementing `OSIntNesting`, saving `SP` to `TCB` etc.) and saves the processor context. Refer to Section 3.3.2.3 for more information on this routine.

L3-11 (5): Refer to the references sections for more information on Blackfin Anomaly 05-00-0283.

L3-11 (6): Initialize the C Run-Time Stack with the application supplied frame size.

L3-11 (7): This is the corresponding epilog for reentrant ISR prologs listed in L3-11(1) – applications installing reentrant interrupt handlers call this macro for an ISR epilog

L3-11 (8): As part of the ISR epilog, call `OSIntExit()` as per the **µC/OS-II** port guidelines [1].

L3-11 (9): Delete the C Run-Time stack created in L3-11(6).

L3-11 (10): Jump to `__uCOS_II_reentrant_interrupt_exit`, which is a straightforward implementation of processor context restore and executes an `RTI` instruction at the end of the restore process. Refer to Section 3.3.2.5 for more information.

Listing 3-12 lists the prolog/epilogs for non-reentrant ISRs

Listing 3-12: UCOS_II_NON_REENTRANT_ISR_PROLOG()/EPILOG

```
#define UCOS_II_NON_REENTRANT_ISR_PROLOG(frame_size)\     // (1)
      [ -- SP ] = R0 ;                              \     // (2)
      [ -- SP ] = P1 ;                              \
      [ -- SP ] = RETS ;                            \     // (3)
      CALL __uCOS_II_non_reentrant_interrupt_entry; \     // (4)
      CC = R0 == R0 ;                               \     // (5)
      P0.L = 0x14 ;                                 \
      P0.H = 0xffc0 ;                               \
      IF CC JUMP 4 ;                                \
      R0 =  [ P0 ] ;                                \
      INIT_C_RUNTIME_STACK(frame_size)                    // (6)

#define UCOS_II_NON_REENTRANT_ISR_EPILOG()          \     // (7)
      CALL _OSIntExit;                              \     // (8)
      DEL_C_RUNTIME_STACK()                         \     // (9)
      JUMP __uCOS_II_non_reentrant_interrupt_exit;        // (10)
```

L3-12 (1): Applications installing non-reentrant interrupt handlers call this macro for an ISR prolog with the `frame_size` parameter (this parameter is the size of C Run-Time stack frame of the ISR to be created, any value between zero and 262152 bytes, including both values in increments of 4). This parameter represents the local stack storage requirements of the ISR i.e. local variables, register saves etc. The typical value of `frame_size` is 0. For more information on managing C Run-Time stack, refer to Section 2.3.1, C Run-Time Model and Environment.

L3-12 (2): This is the first instruction to execute when the processor vectors to the ISR. `R0` and `P1` are pushed onto the stack first to allow interoperability with DD/SSL libraries with a single **µC/OS-II** port for Blackfin (DD/SSL libraries always push `R0` and `P1` onto stack as part of their ISRs prolog).

L3-12 (3): The `RETS` register (return address from function call) is saved before calling the non-reentrant ISR context save routine, `__uCOS_II_non_reentrant_interrupt_entry`. Refer to Section 3.3.2.4 for more information on this routine.

L3-12 (4): Call `__uCOS_II_non_reentrant_interrupt_entry` routine which performs **µC/OS-II** specific ISR operations (incrementing `OSIntNesting`, saving `SP` to `TCB` etc.) and saves the processor context. Refer to Section 3.3.2.4 for more information on this routine.

L3-12 (5): Refer to the references sections for more information on Blackfin Anomaly 05-00-0283.

L3-12 (6): Initialize the C Run-Time Stack with the application supplied frame size.

L3-12 (7): This is the corresponding epilog for non-reentrant ISR prologs listed in L3-12(1) – applications installing non-reentrant interrupt handlers call this macro for an ISR epilog

L3-12 (8): As part of the ISR epilog, call `OSIntExit()` as per the **µC/OS-II** port guidelines [1].

L3-12 (9): Delete the C Run-Time stack created in L3-12(6).

L3-12 (10): Jump to `__uCOS_II_non_reentrant_interrupt_exit`, which is a straightforward implementation of processor context restore and executes an `RTI` instruction at the end of the restore process. Refer to Section 3.3.2.5 for more information.

Note: The prolog and epilog macros must be called in matching pairs for a given ISR i.e. if the prolog for an ISR is chosen to be `UCOS_II_REENTRANT_ISR_PROLOG`, the corresponding epilog must be `UCOS_II_REENTRANT_ISR_EPILOG` (this is a reentrant ISR). Similarly, the corresponding epilog for `UCOS_II_NON_REENTRANT_ISR_PROLOG` is `UCOS_II_NON_REENTRANT_ISR_EPILOG` (non-reentrant ISR).

3.3.1.8. µCOS-II Stack Growth

Listing 3-13: Definition of OS_STK_GROWTH

```
#define  OS_STK_GROWTH        1    // (1)
```

L3-13 (1): 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. Thus, `OS_STK_GROWTH` is defined as 1 as per the **µC/OS-II** port guidelines [1].

3.3.1.9. µCOS-II Context Switch Trap

The attached port of **µC/OS-II** uses Interrupt Level 14 (`IVG14`) as a software trap for task-level context switches. This is made possible through software invoked hardware interrupt, the `RAISE` instruction. For more information refer to Section 2.1.5.3.

Note: Using `IVG14` to perform a context switch violates the porting guidelines of a "synchronous software event" – for a complete discussion, refer to Section 3.3.2.7.

Listing 3-14: Definition of OS_TASK_SW (software trap)

```
#define  OS_TASK_SW()         asm("raise 14;");    // (1)
```

L3-14 (1): `OS_TASK_SW()` macro is defined to 'raise' Interrupt Level 14 (`IVG14`).

3.3.1.10. µCOS-II Critical Sections

Listing 3-15: µCOS-II Critical Sections

```
#include <ccblkfn.h>                                    // (1)

#define  OS_CRITICAL_METHOD    3                        // (2)

#if      OS_CRITICAL_METHOD == 3
#define  OS_ENTER_CRITICAL()  (cpu_sr = cli())          // (3)
#define  OS_EXIT_CRITICAL()   (sti(cpu_sr))             // (4)
#endif
```

L3-15 (1): Include ccblkfn.h for compiler built-in functions – see Section 2.3.3.1, VisualDSP++ Compiler Built-in Functions.

L3-15 (2): The attached Blackfin port of **µC/OS-II** uses `OS_CRITICAL_METHOD#3` since this is preferred method for **µC/OS-II** ports.

L3-15 (3): The `cli()` function retrieves the old value of `IMASK` register, and disables interrupts by setting `IMASK` to all zeros.

L3-15 (4): The `sti()` function installs a new value into `IMASK` register, enabling the interrupt system according to the new mask stored..

3.3.2. OS_CPU_A.ASM

3.3.2.1. Include files

Listing 3-16: `OS_CPU_A.ASM` include files

```
#include <os_cfg.h>                     // (1)
#include <os_cpu.h>                     // (2)
```

L3-16 (1): Include `OS_CFG.H` for **µC/OS-II** kernel configuration [1].

L3-16 (2): Include `OS_CPU.H` for macros defined. See Section 3.3.1, `OS_CPU.H`

3.3.2.2. Global Variables

Listing 3-17: Global variables in `OS_CPU_A.ASM`

```
.extern _OSTaskSwHook;
.extern _OSIntEnter;
.extern _OSIntExit;
.extern _OSTimeTick;
.extern _OSTimeTickHook;
.extern _OSIntNesting;
.extern _OSTCBCur;
.extern _OSTCBHighRdy;
.extern _OSPrioCur;
.extern _OSPrioHighRdy;
.extern _OSRunning;
```

3.3.2.3. __uCOS_II_reentrant_interrupt_entry

While the functionality of ISRs for various interrupt levels is different based on the interrupting source, the prolog and epilog code for all interrupt levels is nearly identical – they must all perform the operations listed in Listing 3-18. This repetition wastes precious L1 code memory – to avoid this, the prolog code for all ISRs was made modular and callable. `__uCOS_II_reentrant_interrupt_entry` encapsulates this functionality for reentrant (nested) ISRs.

Listing 3-18: ISR entry operations under µC/OS-II

```
ISR_start:
Save processor registers                                (1)
Check if OSRunning is set                               (2)
Increment OSIntNesting (if OSIntNesting <255u)          (3)
If OSIntNesting == 1, save SP to TCB                    (4)
Re-enable Interrupts (push RETI onto stack)             (5)
Save remaining processor context                        (6)
```

Listing 3-18 is the pseudo-code the ISR entry operations as per the µC/OS-II porting guidelines for ISRs [1]. From this listing, it is apparent that for reentrant (nested) ISR, further interrupts in the system are held off until L3-18(5) completes execution. In other words, the interrupt latency of the port could be affected by the execution time of L3-18(1) through L3-18(4). Thus, to reduce interrupt latency, it was decided that:

- L3-18(2) through L3-18(4) would be coded in assembly – Listing 3-19 has details.
- The step of context saving (L3-18(1)) was reduced to saving only those registers that were used in steps L3-18(2) through L3-18(4) as shown Listing 3-19 below.

Listing 3-19 on the next page details the interrupt entry function for reentrant interrupts.

Listing 3-19: __uCOS_II_reentrant_interrupt_entry

```
.global __uCOS_II_reentrant_interrupt_entry
__uCOS_II_reentrant_interrupt_entry:

ucos_ii_reentrant_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_reentrant_interrupt_entry_mgmt.end;

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

      R2       = R2.B ( X ) ;                           // (4)
      R2      += 1 ;
      B [ P0 ] = R2 ;
      CC = R2 == 1 ;                                    // (5)
      IF ! CC JUMP ucos_ii_reentrant_interrupt_entry_mgmt.end;

      LOADA( P2 , _OSTCBCur ) ;                         // (6)
      P0     = [ P2 ] ;
      R2     = SP ;
      R1     = 144;                                     // (7)
      R2     = R2 - R1 ;                                // (8)
      [ P0 ] = R2 ;
ucos_ii_reentrant_interrupt_entry_mgmt.end:
      NOP;

reentrant_interrupt_cpu_save_context:
      [ -- SP ] = RETI ;                                // (9)
      [ -- SP ] = ( R7:3 , P5:3 ) ;                     // (10)
      [ -- 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 ) ;

reennтrant_ISR_cpu_save_context.end:
__uCOS_II_reentrant_interrupt_entry.end:
      RTS;                                          // (11)
```

L3-19 (1): Save only those registers used in L3-19(2) through L3-19(8) below.

L3-19 (2): Is `OSRunning` == 1? If not set, don't save `SP` to `OS_TCB`.

L3-19 (3): Is `OSIntNesting < 255`? If not true, don't save `SP` to `OS_TCB`. **µC/OS-II** supports up to 255 deep nesting.

L3-19 (4): Increment `OSIntNesting`

L3-19 (5): Check if `OSIntNesting` == 1. If not true, don't save `SP` to `OS_TCB`.

L3-19 (6): Load the current task's `OS_TCB`

L3-19 (7): Cannot save the current `SP` value to `OS_TCB` because in step L3-19(1) above, the ISR prolog did not save the entire processor context. Calculate the stack pointer when rest of the registers will be saved (after a complete context save) – refer to Section 3.1.2, Task Stack Frame for more information on a task's stack after a context save is complete.

L3-19 (8): Save the calculated stack pointer (note that the actual `SP` register is not modified during this calculation)

L3-19 (9): This is called from the prolog to a reentrant ISR (nested). So, push `RETI` onto stack – this instruction re-enables interrupts.

L3-19 (10): Save the rest of the processor context – this process can be interrupted because `OSIntNesting` has already been incremented and `SP` has been saved to `OS_TCB`.

L3-19 (11): Execute a return from function call (`RETS`) to return back to caller.

3.3.2.4. __uCOS_II_non_reentrant_interrupt_entry

The code for `__uCOS_II_non_reentrant_interrupt_entry` is almost identical to `__uCOS_II_reentrant_interrupt_entry` – see Section 3.3.2.3.

The only difference is that since this is a prolog to a non-reentrant (non-nested) ISR, the `RETI` register is indirectly pushed onto the stack (through `R1`) leaving interrupts disabled for the duration of the ISR (see L3-20(9)). As mentioned in Section 3.3.2.3, the prolog code for non-nested interrupts modular and callable to save L1 code memory.

Listing 3-20 on the next page details the interrupt entry function for non-reentrant interrupts.

Listing 3-20: __uCOS_II_non_reentrant_interrupt_entry

```
.global __uCOS_II_non_reentrant_interrupt_entry
__uCOS_II_non_reentrant_interrupt_entry:

ucos_ii_non_reentrant_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_non_reentrant_interrupt_entry_mgmt.end;

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

      R2       = R2.B ( X ) ;                             // (4)
      R2      += 1 ;
      B [ P0 ] = R2 ;
      CC = R2 == 1 ;                                      // (5)
      IF ! CC JUMP ucos_ii_non_reentrant_interrupt_entry_mgmt.end;

      LOADA( P2 , _OSTCBCur ) ;                           // (6)
      P0     = [ P2 ] ;
      R2     = SP ;
      R1     = 144;                                       // (7)
      R2     = R2 - R1 ;                                  // (8)
      [ P0 ] = R2 ;
ucos_ii_non_reentrant_interrupt_entry_mgmt.end:
      NOP;

non_reentrant_interrupt_cpu_save_context:
      R1        = RETI ;                                  // (9)
      [ -- SP ] = R1 ;
      [ -- SP ] = ( R7:3 , P5:3 ) ;                       // (10)
      [ -- 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 ) ;

non_reennntrant_ISR_cpu_save_context.end:
__uCOS_II_non_reentrant_interrupt_entry.end:
      RTS;                                          // (11)
```

L3-20 (1): Save only those registers used in L3-20(2) through L3-20(8) below.

L3-20 (2): Is `OSRunning` == 1? If not set, don't save `SP` to `OS_TCB`.

L3-20 (3): Is `OSIntNesting < 255`? If not true, don't save `SP` to `OS_TCB`. **µC/OS-II** supports up to 255 deep nesting.

L3-20 (4): Increment `OSIntNesting`

L3-20 (5): Check if `OSIntNesting` == 1. If not true, don't save `SP` to `OS_TCB`.

L3-20 (6): Load the current task's `OS_TCB`

L3-20 (7): Cannot save the current `SP` value to `OS_TCB` because in the step L3-20(1) above, the ISR prolog did not save the entire processor context. Calculate the stack pointer when rest of the registers will be saved (after a complete context save) – refer to Section 3.1.2, Task Stack Frame for more information on a task's stack after a context save is complete.

L3-20 (8): Save the calculated stack pointer (note that the actual `SP` register is not modified during this calculation)

L3-20 (9): This is called from the prolog to a non-reentrant ISR (non-nested). So, save `RETI` through `R1` – this instruction leaves interrupts disabled through `IPEND[4]`.

L3-20 (10): Save the rest of the processor context.

L3-20 (11): Execute a return from function call (`RETS`) to return back to caller.

3.3.2.5. __uCOS_II_reentrant_interrupt_exit/ __uCOS_II_non_reentrant_interrupt_exit

These routines are a straightforward implementation restoring the processor's context as per the task stack frame convention (see Section 3.1.2., Task Stack Frame). They exit by executing a return from the interrupt (`RTI`) instruction.

Again, as mentioned in Section 3.3.2.3, L1 code memory is saved by making the epilog code for non-nested interrupts modular and callable. Also, since the context save routine is independent of the nature of the interrupt (nested or non-nested) `__uCOS_II_reentrant_interrupt_exit` is mapped to `__uCOS_II_non_reentrant_interrupt_exit` – `RETI` is populated by a stack pop for both.

Listing 3-21 on the next page details the interrupt exit function for both reentrant and non-reentrant interrupts.

Listing 3-21: `__uCOS_II_reentrant_interrupt_exit` (also `__uCOS_II_non_reentrant_interrupt_exit`)

```
.global __uCOS_II_reentrant_interrupt_exit
.global __uCOS_II_non_reentrant_interrupt_exit

__uCOS_II_reentrant_interrupt_exit:                    // (1)
__uCOS_II_non_reentrant_interrupt_exit:
interrupt_cpu_save_context:
      LB1  = [ SP ++ ] ;                               // (2)
      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 ++ ] ;                               // (3)
      ASTAT = [ SP ++ ] ;
      P2    = [ SP ++ ] ;
      P0    = [ SP ++ ] ;
      R2    = [ SP ++ ] ;
      R1    = [ SP ++ ] ;
      RETS  = [ SP ++ ] ;
      P1    = [ SP ++ ] ;
interrupt_cpu_save_context.end:
      R0    = [ SP ++ ] ;
      RTI;                                             // (4)
      NOP;
      NOP;
__uCOS_II_reentrant_interrupt_exit.end:
__uCOS_II_non_reentrant_interrupt_exit.end:
      NOP;
```

L3-21 (1): The epilog for interrupts is identical for reentrant (nested) and non-reentrant (non-nested) ISRs. So, both

`__uCOS_II_reentrant_interrupt_exit` and `__uCOS_II_non_reentrant_interrupt_exit` are mapped to the same piece of code.

L3-21 (2): Restore context of the new task from the task's stack.

L3-21 (3): `RETI` is populated by a stack pop for both nested as well non-nested interrupts.

L3-21 (4): Return from interrupt with an `RTI`.

3.3.2.6. OSStartHighRdy

`OSStartHighRdy` is a straightforward implementation of processor context restoration as per the task stack frame convention described in Section 3.1.2, Task Stack Frame.

However, `__uCOS_II_reentrant_interrupt_exit` cannot be called because of the unwanted side-effects of `RETI` restoration through a stack pop (`IPEND[4]` is set disabling global interrupts, further explained below in L3-22 (13)). So, `OSStartHighRdy()` restores processor context manually. To save code space and save execution time, not all registers are restored – only those of consequence are restored to a known value.

Listing 3-22 on the next page details the context restoration performed by `OSStartHighRdy()`.

Listing 3-22: `OSStartHighRdy`

```
.global _OSStartHighRdy;
_OSStartHighRdy:
#if (OS_TASK_SW_HOOK_EN == 1)
      INIT_C_RUNTIME_STACK(0x0)
      CALL _OSTaskSwHook;                              // (1)
      DEL_C_RUNTIME_STACK()
#endif
      LOADA(P1, _OSTCBHighRdy);
      P2 = [ P1 ];
      SP = [ P2 + OFFSETOF(OS_TCB, OSTCBStkPtr)]; // (2)

      LOADA(P1, _OSRunning);
      R1     = 1;
      [ P1 ] = R1;                                     // (3)

      P1   = 140;                                      // (4)
      SP   = SP + P1;

      RETS = [ SP ++ ];                                // (5)
      SP  += 12;                                       // (6)

      R1   = 0;
      LC0  = R1; LC1 = R1;                             // (7)
      L0   = R1; L1 = R1;                              // (8)
      L2   = R1; L3 = R1;

      R2   = [ SP ++ ];                                // (9)
      R1   = [ SP ++ ];                                // (10)
      SP  += 8;                                        // (11)
      R0   = [ SP ++ ];                                // (12)

_OSStartHighRdy.end:
      RTS;                                             // (13)
```

L3-22 (1): Initialize C Run-Time Stack and call `OSTaskSwHook()`.

L3-22 (2): Get stack pointer from `OSTCBHighRdy`.

L3-22 (3): Set `OSRunning` to 1

L3-22 (4): Skip restoring the following registers – `LB1:0, LT1:0, LC1:0, A1:0, M3:0, L3:0, B3:0, I3:0, FP, P5:3, R7:3`. These values are irrelevant to the new task being restored.

L3-22 (5): Do not pop `RETI` off of stack because this sets `IPEND[4]`, disabling global interrupts. For more information, refer to Section 2.1.5, Interrupt Processing. Instead, retrieve `RETI` register value from the stack and load into `RETS` register. `OSStartHighRdy()`returns with a `RTS`, not `RTI` – see L3-22 (13) for more information.

L3-22 (6): Skip over `ASTAT`, `P2`, `P0`. These values are irrelevant to the new task being restored.

L3-22 (7): Set loop counters to zero, to make sure that hardware loops are disabled. The VisualDSP++ compiler does not ensure that the loop counter registers `LC0` and `LC1` are zero on entry or exit from a function.

L3-22 (8): Clear the DAG Length registers too, so that it's safe to use `I-Reg` without them wrapping around. The compiler assumes that the Length registers are zero, both on entry to functions and on return from functions, and ensures this is the case when it generates calls or returns.

L3-22 (9): Restoring stored value for `R2` – as mentioned Section 2.3.1.3.1, Passing Arguments, `R2` contains the third argument when calling C functions.

L3-22 (10): Restoring stored value for `R1` – as mentioned Section 2.3.1.3.1, Passing Arguments, `R1` contains the second argument when calling C functions.

L3-22 (11): Skipping `RETS` and `P1` registers. These values are irrelevant to the new task being restored.

L3-22 (12): Restoring stored value for `R0` – as mentioned Section 2.3.1.3.1, Passing Arguments, `R0` contains the first argument when calling C functions. .

L3-22 (13): The **µC/OS-II** porting guidelines recommend that `OSStartHighRdy()` perform a return from interrupt to start the new task – however, on the Blackfin architecture, executing an `RTI` instruction can have unwanted side-effects. For more information, refer to Section 2.1.5, Interrupt Processing. Thus, the `RETI` value is loaded into the `RETS` register and a return from function call (`RTS`) is performed to start the new task.

3.3.2.7. OSCtxSw

The `OSCtxSw()` implementation in the attached Blackfin port deviates from the **µC/OS-II** porting guidelines. As per guidelines, `OS_TASK_SW()` macro must be mapped to a synchronous software event. In case of the Blackfin architecture, this concept maps to Exceptions – refer to Section 2.1.5.5 for more information on Exceptions. However, this implementation suffers from the unwanted side-effects explained below in Listing 3-23.

Listing 3-23 is a copy of `OSSched()` from `OS_CORE.C`.

Listing 3-23 : `OSSched()` from `OS_CORE.C`

```
void  OS_Sched (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr = 0;
#endif

    OS_ENTER_CRITICAL();                                    // (1)
    if (OSIntNesting == 0) {
        if (OSLockNesting == 0) {
            OS_SchedNew();                                  // (2)
            if (OSPrioHighRdy != OSPrioCur) {
                OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
#if OS_TASK_PROFILE_EN > 0
                OSTCBHighRdy->OSTCBCtxSwCtr++;
#endif
                OSCtxSwCtr++;
                OS_TASK_SW();                               // (3)
            }
        }
    }
    OS_EXIT_CRITICAL();                                     // (4)
}
```

L3-23 (1): `OSSched()` enters critical section by saving of the `IMASK` register to the local variable `cpu_sr`. Depending on the compiler optimization level, the value of `IMASK` could be either be stored as a local variable declared on the stack or in a call-preserved data register (call preserved registers are `R4:7` according to VisualDSP++ compiler C Run-Time Model).

L3-23 (2): The call to `OSSchedNew()` will now search 'ready-task' list and modify `OSPrioHighRdy` and `OSTCBHighRdy` to point to ready task with the highest priority.

L3-23 (3): `OS_TASK_SW()` macro is now called. According to the **µC/OS-II** guidelines [1], this macro maps to a synchronous software event (exceptions, in the case of Blackfin architecture). This call thus swaps the stack pointer to the newly readied high priority task, restores the context of the new task and performs a return from interrupt.

L3-23 (4): L3-23(3) implies that means that `OS_EXIT_CRITICAL()` is not executed until the interrupted task comes back into context. In other words, the `IMASK` register is restored with a value from the new task's stack frame. In other words, each task retains a local 'stack' copy of `IMASK` and loads that local value into the `IMASK` register when it is brought into context.

The implementation described above suffers from the following two drawbacks –

1. The implication of each task maintaining a 'local' copy of `IMASK` (from L3-23 (4)) could present problems with certain applications on the Blackfin processor.

   For example, consider the following multi-threaded application. `Thread1` unmasks the `IMASK` register to allow Parallel Peripheral Interface (PPI) DMA interrupts and then starts the DMA engine to transfer data to the PPI. While the DMA controller is transferring data to the PPI, `Thread1` yields the processor to the next high priority thread, expecting to be brought into context when the PPI DMA is complete (the PPI DMA ISR can post a semaphore to accomplish this). However, the new thread, `Thread2` loads a new value of `IMASK`, which might not have PPI DMA unmasked. In other words, the PPI DMA completion interrupt will not be serviced by the processor, until `Thread1` is brought back into context.

   Maintaining a global `static` copy of the `IMASK` register could be a possible solution – this copy can be used to restore the `IMASK` register value when a new task is brought into context. However, this `static` copy must be coherent with actual value of the `IMASK` register at all times. In other words, the port would suffer from the expense of applications informing **µC/OS-II** every time the `IMASK` register is modified (this could pretty often, especially for libraries, TCP/IP stacks etc.). In addition, additional infrastructure (for example, an atomically incrementing/decrementing counter) is required to support nested calls to enter and exit critical sections for applications.

2. The second drawback is specific to using the exception handler to perform context switches. The Blackfin architecture (as with other architectures) maps multiple events in the system to the exception handler (for the complete list, refer to *The ADSP-BF5xx Blackfin Processor Programming Reference* in the References Section). This implies that exception handler is not dedicated to context switches and the **µC/OS-II** port will have to provide software hooks for applications that require access to the exception handler. For example, cache-enabled applications would 'register' their CPLB manager with **µC/OS-II**'s exception handler to manage CPLB misses. This leads a cumbersome **µC/OS-II** exception handler that might be invoked for every CPLB miss. Also, since the exception handler must check the exception cause before vectoring to the context switch routine (`OSCtxSW()`), the port will incur a performance hit because of cycle penalties associated with conditional branches.

Thus, the attached Blackfin port of **µC/OS-II** does **NOT** map `OS_TASK_SW()` to an `EXCPT` (Force Exception) instruction. Instead, it 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 `IVG14`.

Thus the typical flow for a task-level context switch is as follows:

A1). `OS_Sched()` is called at a task level (`IVG15`) through a **µC/OS-II** service call. `OS_Sched()` then enters a critical section by disabling interrupts as noted in L3-23(1) (by clearing `IMASK` register value and saving the old `IMASK` value through `cli` instruction).

A2). `OS_Sched()` decides to perform a context switch and calls `OS_SchedNew()`, which updates both `OSPrioHighRdy` and `OSTCBHighRdy`, and calls `OS_TASK_SW()`.

A3). `OS_TASK_SW()` raises Interrupt Level 14 (`IVG14`). This interrupt is also latched in the `ILAT` register because interrupts are still disabled.

A4). `OS_Sched()` exits critical section by re-enabling interrupts (restoring the old `IMASK` register value through `sti` instruction).

A5). The processor now vectors to the ISR for `IVG14` i.e. `OSCtxSw()`.

A6). `OSCtxSw()` saves the processor context of the old task and restores the context of the next ready task with the highest priority. `OSCtxSw()` also updates **µC/OS-II** specific globals such as `OSPrioCur`, `OSTCBHighRdy` etc. `OSCtxSw()` is non-interruptible so any further hardware interrupts are held off until the completion of the context switch.

However, note that the operations of `OS_TASK_SW()` (raising `IVG14`) is not a synchronous software event. This could present a problem if the following scenario arises within the system –

Consider the following scenario –

B1). As usual, `OS_Sched()` is called at a task level (`IVG15`) through a **µC/OS-II** service call. `OS_Sched()` then enters a critical section by disabling interrupts as noted in L3-23(1) (by clearing `IMASK` register value and saving the old `IMASK` value through `cli` instruction).

B2). At this point, assume that a hardware interrupt is triggered while the processor is executing within this critical section. This interrupt will be latched in the `ILAT` register (but not serviced yet) because interrupts are disabled.

B3). `OS_Sched()` decides to perform a context switch and calls `OS_SchedNew()`, which updates both `OSPrioHighRdy` and `OSTCBHighRdy`, and calls `OS_TASK_SW()`.

B4). `OS_TASK_SW()` raises Interrupt Level 14 (`IVG14`). This interrupt is also latched in the `ILAT` register because interrupts are still disabled.

B5). `OS_Sched()` exits critical section by re-enabling interrupts (restoring the old `IMASK` register value through `sti` instruction).

B6). At this point, note that the processor will vector to the ISR for hardware interrupt, not `IVG14` since all hardware interrupts are higher than `IVG14` on the Blackfin architecture. The higher priority HW interrupt results in an event that makes the original task (switched out by Step**#B3**) ready to run again. This results in `OSPrioHighRdy` to be set back to `OSPrioCur`.

B7). The scheduling logic in `OSIntExit()` decides to do nothing because `OSPrioHighRdy` equals `OSPrioCur`. This means `OSTCBHighRdy` is now invalid and cannot be used because it points to the context of the task scheduled in Step**#B3**. The processor returns from the hardware interrupt by executing `RTI` instruction.

B8). The processor now vectors to the ISR for `IVG14` i.e. `OSCtxSw()`. Note that `IVG14` has been latched since Step# **B4**.

B9). `OSCtxSw()` cannot perform a context switch operation because `OSTCBHighRdy` does not point to a valid `TCB` anymore (because of the intervening hardware interrupt from Steps#**B2** and #**B6**).

To accommodate for this scenario, `OSCtxSw()` must be modified slightly. `OSCtxSw()` must check if `OSPrioHighRdy == OSPrioCur`. If this is true, an intervening hardware interrupt has already performed a context switch, thus invalidating the `OSTCBHighRdy` pointer. In this event, `OSCtxSw()` simply ignores the invalid `OSTCBHighRdy` and continues running the task that was originally running upon entering Step#**B1**, as shown in L3-24(4).

> **Note:** The scenario described in Steps **#B1** through **#B9** is not a frequent occurrence in typical system. Thus, the `OSCtxSw()` assembly is written such that the conditional branch (for `OSPrioHighRdy == OSPrioCur`) is statically predicted as 'not taken'. This implies that in typical scenarios (described in Steps **#A1** through **#A6**), the conditional branch is not taken and the processor does not suffer a pipeline penalty.

Listing 3-24 details the operations of `OSCtxSw()` in the attached Blackfin port of **µC/OS-II**.

Listing 3-24: `OSCtxSw()`

```
.global _OSCtxSw;
_OSCtxSw:
      [ -- SP ] = R0 ;
      [ -- SP ] = P1 ;
      [ -- SP ] = RETS ;
      [ -- SP ] = R1 ;
      [ -- SP ] = R2 ;
      [ -- SP ] = P0 ;
      [ -- SP ] = P2 ;
      [ -- SP ] = ASTAT ;
      CALL.X non_reentrant_interrupt_cpu_save_context; // (1)

      LOADA(P3, _OSPrioHighRdy);
      R4 = B[ P3 ](Z);                                 // (2)

      LOADA(P3, _OSPrioCur);
      R5 = B[ P3 ](Z);                                 // (3)

      CC = R4 == R5;                                   // (4)
      IF CC JUMP _AbortOSCtxSw;

      LOADA(P4, _OSTCBCur);
      P5     = [ P4 ];                                 // (5)

      [ P5 ] = SP;                                     // (6)

#if (OS_TASK_SW_HOOK_EN == 1)
      INIT_C_RUNTIME_STACK(0x0)
      CALL _OSTaskSwHook;                              // (7)
      DEL_C_RUNTIME_STACK()
#endif

      B[ P3 ] = R4;                                    // (8)

      LOADA(P3, _OSTCBHighRdy);
      P5      = [ P3 ];                                // (9)

      [ P4 ]  = P5;                                    // (10)

_OSCtxSw_modify_SP:
      SP      = [ P5 ];                                // (11)

_AbortOSCtxSw:
_CtxSwRestoreCtx:
_OSCtxSw.end:
      JUMP __uCOS_II_non_reentrant_interrupt_exit;     // (12)
```

L3-24 (1): Save `R0:2, P0:2`, `RETS` and `ASTAT` and save the remaining context of the current task by calling `non_reentrant_interrupt_cpu_save_context`.

L3-24 (2): `R4` contains `OSPrioHighRdy`.

L3-24 (3): `R5` contains `OSPrioCur`.

L3-24(4): If (`OSPrioHighRdy == OSPrioCur`) is `TRUE`, abort the context switch and continue running the current task with priority of `OSPrioCur` (refer to the scenario described above in Steps #**B1** through #**B9**). Note that this conditional branch is statically predicated "not taken" and thus, the processor does not suffer pipeline penalties under typical context switch scenarios (i.e. the typical case is (`OSPrioHighRdy == OSPrioCur`) evaluating to `FALSE` and the branch is "not taken", resulting in a context switch).

L3-24 (5): Get the pointer to `OSTCBCur`.

L3-24 (6): Save the current task's stack pointer (`SP`) into the `OSTCBCur`

L3-24 (7): Initialize C Run-Time Stack and call `OSTaskSwHook()`

L3-24 (8): `OSPrioCur` = `OSPrioHighRdy`

L3-24 (9): Get a pointer to `OSTCBHighRdy`.

L3-24 (10): `OSTCBCur` = `OSTCBHighRdy`

L3-24 (11): Load the new `SP` from `OSTCBHighRdy`.

L3-24 (12): Restore all processor registers from the new task's stack frame by executing a jump to `__uCOS_II_non_reentrant_interrupt_exit`. The exit routine executes a return from interrupt (`RTI`) to resume with the new task.

3.3.2.8. OSIntCtxSw

The `OSCtxSw()` implementation in the attached Blackfin port deviates from the **µC/OS-II** porting guidelines. According to the guidelines, `OSIntCtxSw()`performs most of the operations of `OSCtxSw()` except the initial context saving and resumes the new task by executing a return from the interrupt instruction [1]. Once again, similar to the discussion in Section 3.3.2.7, `OSCtxSw()`, this implementation also suffers from similar drawbacks as explained below in Listing 3-25.

Listing 3-25 is a copy of `OSIntExit()` from `OS_CORE.C`.

Listing 3-25: `OSIntExit()` from OS_CORE.C

```
void  OSIntExit (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr = 0;
#endif

    if (OSRunning == OS_TRUE) {
        OS_ENTER_CRITICAL();                          // (1)
        if (OSIntNesting > 0) {
            OSIntNesting--;
        }
        if (OSIntNesting == 0) {
            if (OSLockNesting == 0) {
                OS_SchedNew();
                if (OSPrioHighRdy != OSPrioCur) {
                    OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
#if OS_TASK_PROFILE_EN > 0
                    OSTCBHighRdy->OSTCBCtxSwCtr++;
#endif
                    OSCtxSwCtr++;
                    OSIntCtxSw();                     // (2)
                }
            }
        }
        OS_EXIT_CRITICAL();                           // (3)
    }
}
```

L3-25 (1): `OSIntExit()` enters critical section by saving of the `IMASK` register to the local variable `cpu_sr`. Depending on the compiler optimization level, the value of `IMASK` could be either be stored as a local variable declared on the stack or in a call-preserved data register (call preserved registers are `R4:7` according to VisualDSP++ compiler C Run-Time Model).

L3-25 (2): Assuming that the interrupt in question readies a higher priority task, `OSIntCtxSw()` is called. According to the **µC/OS-II** guidelines [1], this call swaps the stack pointer to the newly readied high priority task, restores the context of the new task and performs a return from interrupt.

L3-25 (3): L3-25(2) implies that means that `OS_EXIT_CRITICAL()` is not executed until the interrupted task comes back into context. It also implies that the `IMASK` register is restored with a value from the new task's stack frame. In other words, each task retains a local 'stack' copy of `IMASK` and loads that local value into the `IMASK` register when it is brought into context.

Once again, as in the case of `OSCtxSw()`, the porting guidelines have unwanted side-effect of creating a 'local' copy of `IMASK` for each task. The disadvantages of this side-effect are discussed in Section 3.3.2.7, `OSCtxSw()`.

The attached Blackfin port of **µC/OS-II** gets around this side-effect by deviating slightly from the recommended implementation. The rest of this section discusses this approach.

> **Note:** It is highly recommended that the reader become familiar with the C Run-Time Environment described in Section 2.3.1 before reading this section further.

The following background information is worth noting:
Every ISR under **µC/OS-II** must call `OSIntExit()` before returning from the interrupt. Since `OSIntExit()` is compiler generated C function, it implies the following (as mentioned in Section 2.3.1, C Run-Time Model and Environment) –

- Every ISR must ensure that a C Run-Time Environment is set up before `OSIntExit()` can be called.
- The ISR must also destroy the C Run-Time Environment after `OSIntExit()` returns but before executing the return from interrupt instruction (`RTI`).

Also, as mentioned in Section 2.3.1, all compiler generated code creates a stack frame as part of a function's epilog through the `LINK` instruction. In addition, the compiler also generates a function epilog section that destroys the stack frame through the `UNLINK` instruction. The code sequences for `LINK` and `UNLINK` instructions are listed in Figure 3-2 for reference.

`LINK n;`	`UNLINK;`

`[--SP] = RETS;`
`[--SP] = FP;`
`FP = SP;`
`SP += -n;`
	`SP = FP;`
	`FP = [SP++];`
	`RETS = [SP++];`

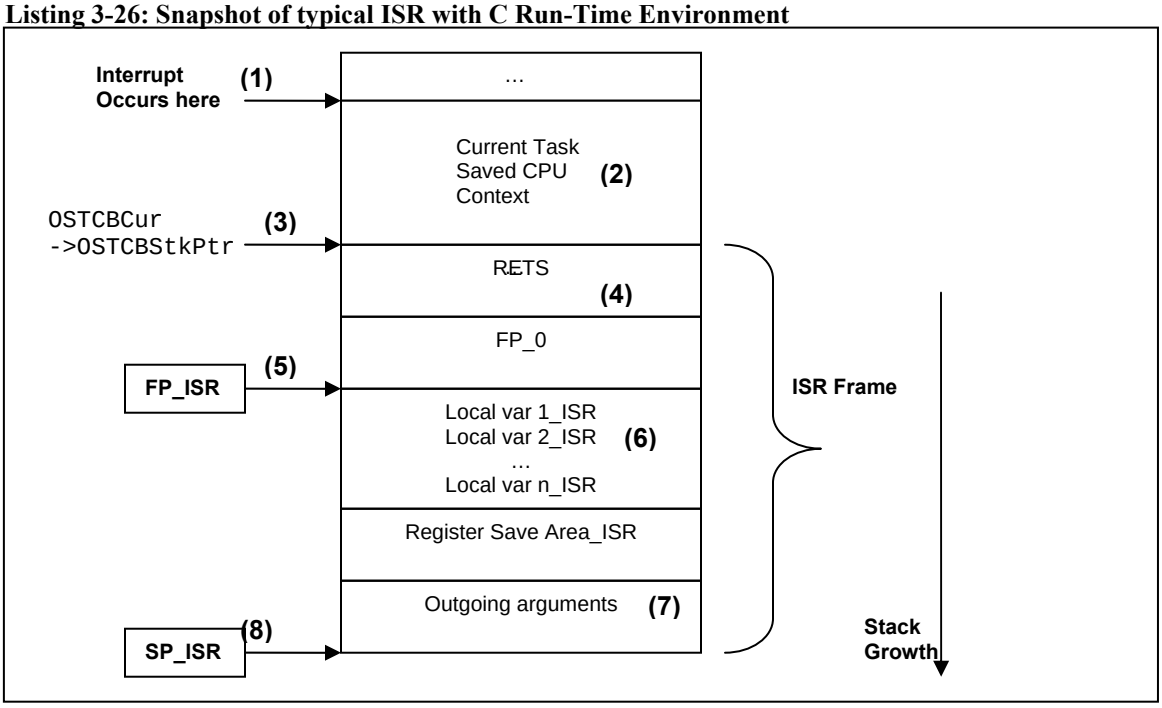
Figure 3-2: `LINK` and `UNLINK` code sequencer

With that background information in perspective, Listing 3-26 and Listing 3-27 illustrate the implementation details of `OSIntCtxSw()` as a 2-step process.

First, Listing 3-26 shows a typical ISR with C Run-Time Stack setup. This is a snapshot of the stack right before `OSIntExit()` is called but after all other calls within the ISR (presumably to service the interrupt, post semaphores etc.) have completed.

Next, Listing 3-27 is a snap-shot of the stack right after `OSIntCtxSw()` has performed a context-switch to the newly readied task.

Listing 3-26: Snapshot of typical ISR with C Run-Time Environment



L3-26 (1): Interrupt occurs at this point.

L3-26 (2): ISR saves the context as per the task stack frame convention detailed in Section 3.1.2.

L3-26 (3): The stack pointer has been saved to the `OSTCBCur` when the application ISR calls to either `UCOS_II_REENTRANT_ISR_PROLOG()/UCOS_II_NON_REENTRANT_ISR_PROLOG()`. Refer to Section 3.3.1.7 for more information on these macros.

L3-26 (4): The ISR sets up a C Run-Time Environment setup by calling `INIT_C_RUNTIME_STACK()` macro. The `LINK` instruction in this macro saves the `RETS` and `FP` to the stack. The location where `RETS` is stored marks the start of ISR's frame. Note that these values of `FP` (`FP_0`) and `RETS` values are irrelevant because there was no real 'caller' this ISR – however, this is alright since we don't use either of these values.

L3-26 (5): As part of the `LINK` instruction sequence, the hardware also re-assigns the Frame pointer (`FP`) register to point to `FP_ISR`, the location where `FP_0` is stored on the stack. `LINK` instruction sequence is shown in Figure 3-2.

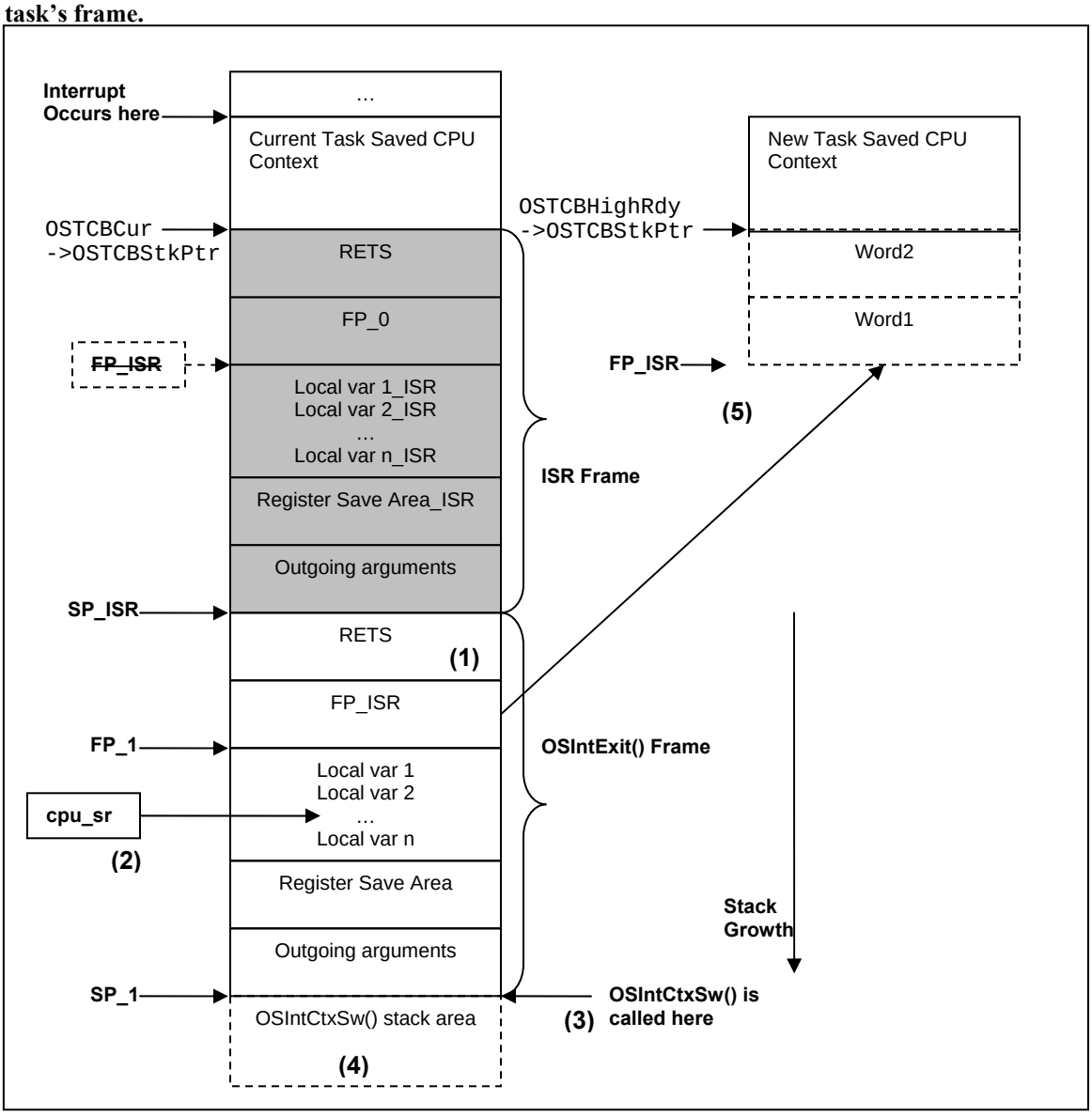
L3-26 (6): Also, as part of the `LINK` instruction sequence in Figure 3-2, the hardware also decrements `SP` by parameter `frame_size`. The application ISR provides this parameter when calling either `UCOS_II_REENTRANT_ISR_PROLOG()/UCOS_II_NON_REENTRANT_ISR_PROLOG()`. This parameter corresponds to (the number of local variables + register save locations) that the application ISR needs. Typically, this parameter is zero since most of the ISR handling will be done in calls made to C-functions, which create their own frames.

L3-26 (7): As mentioned in Section 2.3.1.3.1, Passing Arguments, every assembly routine must create space for 12 bytes of outgoing arguments, even if the function being called has no arguments. The application ISR does not need to account for Outgoing Arguments – the `INIT_C_RUNTIME_STACK()` macro accounts for the necessary 12-bytes for outgoing parameters before calling any C functions.

L3-26 (8): This location marks the new stack pointer (`SP_ISR`) of the ISR frame.

Listing 3-27 continues from Listing 3-26 left off – the operations were broken down into two Listings to reduce the clutter in a single Listing.

Listing 3-27: Snapshot of typical ISR stack right before `OSIntCtxSw()` is about to complete. Note that `FP_ISR` has been overwritten, essentially 'moving' the ISR stack frame underneath the new task's frame.



L3-27 (1): This point represents a time in execution when (a) the prolog to a typical ISR has already setup the C Run-Time Environment and (b) all function calls to clear the interrupt device, service the interrupt, post semaphores/flags etc. within the ISR are complete. Now, the ISR code calls `OSIntExit()` as part of the `UCOS_II_REENTRANT_ISR_EPILOG()` or `UCOS_II_NON_REENTRANT_ISR_EPILOG()` macro. Since `OSIntExit()` is compiler generated code, the function epilog will create a new frame on the stack through the `LINK` instruction, which saves `RETS` and `FP_ISR` (frame pointer of the ISR frame) to the stack and creates space for local variables.

L3-27 (2): As shown in L3-25 (1), `OSIntExit()` enters a critical section by saving of the `IMASK` register to the local variable `cpu_sr`. Depending on the compiler optimization level, the

value of IMASK could be either be stored as a local variable declared on the stack or in a call-preserved data register (R4:7 as per C Run-Time model).

L3-27 (3): Assuming that the interrupt in question readies a higher priority task, OSIntCtxSw() is called.

L3-27 (4): OSIntCtxSw() retrieves the stack pointer of the new task. However, the stack pointer register (SP) is not modified because it would be overwritten by the UNLINK instruction sequence as part of the function epilog of OSIntExit()with the value of FP_ISR anyway. Instead, OSIntCtxSw() overwrites the stored value of FP_ISR with the value of newly retrieved SP – note that it must pad it by 2 stack entries to mimic stored values of RETS and FP – these stored values are irrelevant and are not used (as explained in L3-26(4) above). This action essentially 'moves' the ISR stack frame underneath new task's stack frame. Furthermore, OSIntCtxSw() performs a 'fake pop' into the RETI register – this action globally disable interrupts through IPEND[4] until the next RTI (return from interrupt) instruction - for more information, see section 2.1.5.4, Nesting of Interrupts. This action ensures that the context switch process is non-interruptible. Without this 'fake pop' into RETI register, interrupts would be re-enabled when the macro OS_EXIT_CRITICAL() in OSIntExit() executes (refer to L3-25(3)) when OSIntCtxSw() returns. These operations of OSIntCtxSw() are further explained in Listing 3-28. . OSIntCtxSw() returns with a return from function call instruction (RTS).

L3-27 (5): When the UNLINK instruction in the function epilog of OSIntExit() completes execution, the frame pointer register of the ISR frame (FP_ISR) will point to the new task's stack frame (instead of the old task's frame). Thus, when the final stack frame of the ISR (the gray frame in Listing 3-27) is torn down through the UNLINK instruction in DEL_C_RUNTIME_STACK in ISR epilog macro, the SP register will point to the first word of the new task's saved frame. At this point, the ISR epilog macro jumps to the context restore routine __uCOS_II_non_reentrant_interrupt_exit, which restores the new task's context and resumes with new task by executing a return from interrupt (RTI) instruction. The RTI instruction re-enables interrupts through IPEND[4].

As explained in Listing 3-26 and Listing 3-27, OSIntCtxSw() returns with a return from function instruction (RTS) without changing the stack pointer. This means that OSIntExit() still has access to the local variable, cpu_sr and when the macro OS_EXIT_CRITICAL() in L3-25(3) is executed, the original IMASK is restored. Thus, this approach does not require global management of the IMASK value.

Listing 3-28 details the operations of OSIntCtxSw() in Blackfin assembly.

Listing 3-28: `OSIntCtxSw()` code

```
.global _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)
```

L3-28 (1): Initialize C Run-Time Stack and call `OSTaskSwHook()`

L3-28 (2): Load now has `OSPrioCur` into `P0`

L3-28 (3): `R0` now has `OSPrioHighRdy`

L3-28 (4): `OSPrioCur` = `OSPrioHighRdy`

L3-28 (5): Get stack pointer from `OSTCBHighRdy`.

L3-28 (6): `OSTCBCur` = `OSTCBHighRdy`

L3-28 (7): Get the new task's `SP` from `OSTCBHighRdy`

L3-28 (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. This corresponds to the description of L3-27(4).

L3-28 (9): Overwrite the stored Frame Pointer of the ISR (`FP_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. This corresponds to the description of L3-27(4).

L3-28 (10): As explained earlier in L3-27(4), 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.

L3-28(11): Popping into the `RETI` register globally disables interrupts until the executing the next `RTI` instruction

L3-28 (12): Return to `OSIntExit()` with an `RTS`.

3.3.2.9. OSTickISR

Listing 3-29: `OSTickISR()`

```
.global _OSTickISR
_OSTickISR:

#if (OS_ADI_IVGTMR_ISR_Type == OS_ADI_Reentrant_ISR)// (1)
      UCOS_II_REENTRANT_ISR_PROLOG(0x0)
#else
      UCOS_II_NON_REENTRANT_ISR_PROLOG(0x0)
#endif

      CALL _OSTimeTick;                              // (2)

#if (OS_ADI_IVGTMR_ISR_Type == OS_ADI_Reentrant_ISR)// (3)
      UCOS_II_REENTRANT_ISR_EPILOG ()
#else
      UCOS_II_NON_REENTRANT_ISR_EPILOG ()
#endif
_OSTickISR.end:
```

L3-29 (1): The nature of the Core Timer ISR (reentrant or non-reentrant) is defined by `OS_ADI_IVGTMR_ISR_Type` as described in Section 3.3.1.6. For more information on the ISR prolog macros, refer to Section 3.3.1.7.

L3-29 (2): Call `OSTimeTick()`

L3-29 (3): For more information on ISR epilog macros, refer to Section 3.3.1.7.

3.3.2.10. OS_ADI_Invalid_RTS_from_Task

Listing 3-30: OS_ADI_Invalid_RTS_from_Task

```
.global _OS_ADI_Invalid_RTS_from_Task
_OS_ADI_Invalid_RTS_from_Task:

      JUMP 0;                                          // (1)

_OS_ADI_Invalid_RTS_from_Task.end:
```

L3-30 (1): All tasks (threads) within µCOS-II are `while(1)` loops - once done with its operations, task can suspend, delete itself etc. to yield the processor. Under NO circumstances can a task return with a return from function (`RTS`). However, `OS_ADI_Invalid_RTS_from_Task` serves as a placeholder for debugging purposes for tasks that do return.

> **Note:** `OS_ADI_Invalid_RTS_from_Task` catches all tasks that return with an `RTS` except the first task, which is started by `OSStartHighRdy` – this is because `OSStartHighRdy` overwrites `RETS` value with the start address of the first task. Refer to Section 3.3.2.6, `OSStartHighRdy` for more information.

3.3.3. OS_CPU_C.C

3.3.3.1. Include Files

Listing 3-31: `OS_CPU_C.C` include files

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

L3-31 (1): `sys/platform.h` contains all the memory mapped register definitions for the variants of the Blackfin processors. These are used for installing the core timer.

L3-31 (2): `sys/exception.h` contains the prototype for `register_handler_ex()` – see Section 3.3.3.3., `OSInitHookEnd()` for more information.

L3-31 (3): `ucos_ii.h` provides the prototypes for all **µC/OS-II** -related APIs/structures.

3.3.3.2. Global Variables

There are no global variables for this file.

3.3.3.3. OSInitHookEnd

Listing 3-32: OSInitHookEnd

```
Void OSInitHookEnd(void)
{
    register_handler_ex (ik_ivg14, (ex_handler_fn)OSCtxSw,    // (1)
EX_INT_ENABLE);
}
```

L3-32 (1): Installs `OSCtxSw()` as the ISR for Interrupt Level 14 (`IVG14`, context switch software trap). Also unmasks `IMASK` register to allow `IVG14` to interrupt the core.

3.3.3.4. OSTaskDelHook

Listing 3-33: OSTaskDelHook

```
void OSTaskDelHook(OS_TCB *ptcb)
{
#if OS_TASK_CREATE_EXT_EN == 1
      free((void *)ptcb->OSTCBStkBottom);                    // (1)
#endif
}
```

L3-33 (1): Free the memory allocated to task's stack – `OSTCBStkBottom` always points to the bottom of the stack.

3.3.3.5. OSTaskStkInit

`OSTaskStkInit()` essentially simulates the context save when vectoring to an ISR – refer to Section 3.1.2, Task Stack Frame for more information on the convention followed for an interrupted task's stack frame.

Listing 3-34: `OSTaskStkInit()`

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

      OS_STK fill;

      opt    = opt;                          // (1)
      stk    = ptos;
      stk    -= 3;                           // (2)
      *--stk = (OS_STK) pdata;               // (3)
      *--stk = (OS_STK) 0;                   // (4)
      *--stk = (OS_STK) pdata;               // (5)
      *--stk = (OS_STK) pdata;               // (6)
      *--stk = (OS_STK) 0;                   // (7)
      *--stk = (OS_STK) 0;                   // (8)
      *--stk = (OS_STK) 0;                   // (9)

      *--stk = (OS_STK) task;                // (10)

      for (i=35; i>0; i--)
      {
            *--stk = (OS_STK) 0;              // (11)
      }

      *--stk = (OS_STK) OS_ADI_Invalid_RTS_from_Task;   // (12)

      return ((OS_STK *)stk);                 // (13)
}
```

L3-34 (1): The `opt` parameter is not used so to avoid compiler warnings, it's assigned to itself.

L3-34 (2): As mentioned in Section 2.3.1.3.1, Passing Arguments, the C Run-Time model requires that a the caller to a C-function allocate at least 3 stack entries (12 bytes corresponding to `R0`, `R1` and `R2` registers) on the stack.

L3-34 (3): Populating the stored value for `R0` – as mentioned Section 2.3.1.3.1, Passing Arguments, `R0` contains the first argument when calling C functions.

L3-34 (4): Populating the stored value for `P1` – irrelevant value for a new task.

L3-34 (5): Populating the stored value for `R1` – as mentioned Section 2.3.1.3.1, Passing Arguments, `R1` contains the second argument when calling C functions.

L3-34 (6): Populating the stored value for `R2` – as mentioned Section 2.3.1.3.1, Passing Arguments, `R2` contains the first argument when calling C functions.

L3-34 (7): Populating the stored value for `P0` – irrelevant value for a new task.

L3-34 (8): Populating the stored value for `P2` – irrelevant value for a new task.

L3-34 (9): Populating the stored value for `ASTAT` – irrelevant value for a new task.

L3-34 (10): Populating the stored value for `RETI` – return from interrupt register. This must contain the start address of the task.

L3-34 (11): Populating the rest of the registers (`R7:3, P5:3`, 4 words of `A1:0(.W,.X), LT0, LT1, LC0, LC1, LB0, LB1, I3:0, M3:0, L3:0, B3:0`) – all values irrelevant.

L3-34 (12): Populating the stored value for `RETS` – return from function register. For debugging purposes, this contains the placeholder for tasks that might return without a `while(1)` loop. Refer to Section 3.3.2.10, `OS_ADI_Invalid_RTS_from_Task`.

L3-34 (13): Return the top of the stack

3.3.3.6. CoreTimerInit

`CoreTimerInit` is called to initialize the core timer register file and provide **µC/OS-II** with a periodic heartbeat. The parameters passed in determine the periodicity.

Listing 3-35: `OSCoreTimerInit`

```
void CoreTimerInit( INT32U scale, INT32U counter)
{
      *pTCNTL   = 1;                                  // (1)
      *pTSCALE  = scale;                              // (2)
      *pTCOUNT  = counter;                            // (3)
      *pTPERIOD = counter;                            // (4)
      register_handler_ex (ik_timer,                  // (5)
                           OSTickISR, EX_INT_ENABLE);
      *pTCNTL   = 0x7;                                // (6)
}
```

L3-35 (1): Disable timer while configuring it: Write `0x1` to Core Timer Control Register

L3-35 (2): Program the scale of the core timer: Write the parameter scale to Core Timer Scale Register.

L3-35 (3): Program the count of the core timer before interrupting the core: Write the parameter counter to the Core Timer Count Register.

L3-35 (4): Program the period of the core timer before interrupting the core (for auto-reloading): Write the parameter counter to the Core Timer Period Register.

L3-35 (5): Installs `OSTickISR()` as the Interrupt Service Routine for Interrupt Level 6 (`IVGTMR`). Also unmasks `IMASK` register to allow `IVGTMR` to interrupt the core.

L3-35 (6): Enable Timer for auto reload mode.

4. Example Applications

4.1. *Overview*

The attached Blackfin port of **µC/OS-II** comes with example applications located in the following directories:

- `Micrium\Software\EvalBoards\ADI\ADSP-BF5xx_EZKIT_Lite\VDSP50\OS\`
  for the various ADSP-BF5xx EZ-KIT Lite evaluation platforms

5. References

- Jean J. Labrosse (2002), *MicroC/OS-II – The Real Time Kernel*. New York: CMP Books.

5.1. *Processor Manuals*

The following manuals were referenced in this application note. For these manuals, refer to Blackfin Processor Manuals section:
http://www.analog.com/processors/blackfin/technicalLibrary/manuals/index.html#Processor%20Manuals

- Getting Started with Blackfin Processors
- ADSP-BF53x/BF56x Blackfin Processor Programming Reference
- ADSP-BF533 Blackfin Processor Hardware Reference
- ADSP-BF537 Blackfin Processor Hardware Reference
- ADSP-BF561 Blackfin Processor Hardware Reference
- ADSP-BF54x Blackfin Processor Hardware Reference
- ADSP-BF54x Blackfin Processor Peripheral Reference
- Blackfin Processor Errata (Anomaly) Lists

5.2. *Tools Manuals*

The following manuals were referenced in this application note. For these manuals, refer to the Blackfin Tools Manuals section:

http://www.analog.com/processors/blackfin/technicalLibrary/manuals/index.html#Software%20Manuals

- 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
- VisualDSP++ 5.0 Device Drivers and System Services Manual for Blackfin Processors

5.3. *Evaluation Kit Manuals*

The following manuals were referenced in this application note. For these manuals, refer to the Blackfin Evaluation Kit Manuals section:

http://www.analog.com/processors/blackfin/technicalLibrary/manuals/index.html#Evaluation%20Kit%20Manuals

- ADSP-BF533 EZ-KIT Lite Evaluation System Manual
- ADSP-BF537 EZ-KIT Lite Evaluation System Manual
- ADSP-BF561 EZ-KIT Lite Evaluation System Manual
- ADSP-BF548 EZ-KIT Lite Evaluation System Manual

6. Contacts

For all µC/OS-II -specific questions, please contact:

Micriμm
949 Crestview Circle
Weston, FL 33327
USA
+1 954 217 2036
+1 954 217 2037 (FAX)
WEB: www.Micrium.com
Email: Support@Micrium.com

For Blackfin specific questions, please contact:

Analog Devices Inc.
Blackfin Application Team
3 Technology Way,
Norwood, MA 02062
Email: processor.support@analog.com
WEB: www.analog.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
Email: rdorders@rdbooks.com

7. Revision History

Revision	Author	Notes
1.0	Deep B. (ADI)	Port Submission (November 2006)
1.1	TL and JW (ADI)	Port Update (August 2007)