RTEMS Porting Guide

Edition 4.9.6, for RTEMS 4.9.6

24 July 2011

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 2009-08-14.15

The RTEMS Project is hosted at `http://www.rtems.com`. Any inquiries concerning RTEMS, its related support components, its documentation, or any custom services for RTEMS should be directed to the contacts listed on that site. A current list of RTEMS Support Providers is at `http://www.rtems.com/oarsupport`.

Table of Contents

Preface **1**

1 Development Tools **3**

2 Source Code Organization **5**

2.1 Introduction 5

3 CPU Model Variations **7**

3.1 Overview of RTEMS Portability 7
3.1.1 Processor Families 7
3.1.2 Boards 8
3.1.3 Applications 8
3.2 Coding Issues 9

4 CPU Initialization **11**

4.1 Introduction 11
4.2 Initializing the CPU 11

5 Interrupts **13**

5.1 Introduction 13
5.2 Interrupt Levels 13
5.2.1 Interrupt Level Mask 13
5.2.2 Obtaining the Current Interrupt Level 13
5.2.3 Set the Interrupt Level 13
5.2.4 Disable Interrupts 14
5.2.5 Enable Interrupts 14
5.2.6 Flash Interrupts 14
5.3 Interrupt Stack Management 15
5.3.1 Hardware or Software Managed Interrupt Stack 15
5.3.2 Allocation of Interrupt Stack Memory 15
5.3.3 Install the Interrupt Stack 16
5.4 ISR Installation 16
5.4.1 Install a Raw Interrupt Handler 16
5.4.2 Interrupt Context 16
5.4.3 Maximum Number of Vectors 16
5.4.4 Install RTEMS Interrupt Handler 16
5.5 Interrupt Processing 17
5.5.1 Interrupt Frame Data Structure 17
5.5.2 Interrupt Dispatching 17
5.5.3 ISR Invoked with Frame Pointer 19
5.5.4 Pointer to _Thread_Dispatch Routine 19

6 Task Context Management 21
6.1 Introduction 21
6.2 Task Stacks 21
6.2.1 Direction of Stack Growth 21
6.2.2 Minimum Task Stack Size 21
6.2.3 Stack Alignment Requirements 21
6.3 Task Context 22
6.3.1 Basic Context Data Structure 22
6.3.2 Initializing a Context 22
6.3.3 Performing a Context Switch 23
6.3.4 Restoring a Context 24
6.3.5 Restarting the Currently Executing Task 24
6.4 Floating Point Context 25
6.4.1 CPU_HAS_FPU Macro Definition 25
6.4.2 CPU_ALL_TASKS_ARE_FP Macro Setting 25
6.4.3 CPU_USE_DEFERRED_FP_SWITCH Macro Setting 26
6.4.4 Floating Point Context Data Structure 27
6.4.5 Size of Floating Point Context Macro 27
6.4.6 Start of Floating Point Context Area Macro 27
6.4.7 Initializing a Floating Point Context 28
6.4.8 Saving a Floating Point Context 28
6.4.9 Restoring a Floating Point Context 29

7 IDLE Thread 31
7.1 Does Idle Thread Have a Floating Point Context? 31
7.2 CPU Dependent Idle Thread Body 31
7.2.1 CPU_PROVIDES_IDLE_THREAD_BODY Macro Setting 31
7.2.2 Idle Thread Body 32

8 Priority Bitmap Manipulation 33
8.1 Introduction 33
8.2 _Priority_Bit_map_control Type 33
8.3 Find First Bit Routine 33
8.4 Build Bit Field Mask 35
8.5 Bit Scan Support 35

9 Code Tuning Parameters 37
9.1 Inline Thread_Enable_dispatch 37
9.2 Inline Thread_queue_Enqueue_priority 37
9.3 Structure Alignment Optimization 37
9.4 Data Alignment Requirements 38
9.4.1 Data Element Alignment 38
9.4.2 Heap Element Alignment 38
9.4.3 Partition Element Alignment 38

10 Miscellaneous . . . 41
10.1 Fatal Error Default Handler . . . 41
10.2 Processor Endianness . . . 41
10.2.1 Specifying Processor Endianness . . . 41
10.2.2 Endian Swap Unsigned Integers . . . 41

Command and Variable Index . . . 45

Concept Index . . . 47

Preface

The purpose of this manual is to provide a roadmap to those people porting RTEMS to a new CPU family. This process includes a variety of activities including the following:

- targeting the GNU development tools
- porting the RTEMS executive code
- developing a Board Support Package
- writing an RTEMS CPU Supplement manual for the completed port.

This document focuses on the process of actually porting the RTEMS executive code proper. Each of the data structures, routines, and macro definitions required of a port of RTEMS is described in this document.

Porting any operating system, including RTEMS, requires knowledge of the operating system, target CPU architecture, and debug environment. It is very desirable to have a CPU simulator or hardware emulator when debugging the port. This manual assumes that the user is familiar with building and using RTEMS, the C programming language, and the target CPU architecture. It is desirable to be familiar with the assembly language for the target CPU family but since only a limited amount of assembly is required to port RTEMS.

1 Development Tools

When porting RTEMS to a new CPU architecture, one will have to have a development environment including compiler, assembler, linker, and debugger. The GNU development tool suite used by RTEMS supports most modern CPU families. Often all that is required is to add RTEMS configurations for the target CPU family. RTEMS targets for the GNU tools usually start life as little more than aliases for existing embedded configurations. At this point in time, ELF is supported on most of the CPU families with a tool target of the form CPU-elf. If this target is not supported by all of the GNU tools, then it will be necessary to determine the configuration that makes the best starting point regardless of the target object format.

Porting and retargetting the GNU tools is beyond the scope of this manual. The best advice that can be offered is to look at the existing RTEMS targets in the tool source and use that as a guideline.

2 Source Code Organization

This section describes the organization of the source code within RTEMS that is CPU family and CPU model dependent.

2.1 Introduction

The CPU family dependent files associated with a port of the RTEMS executive code proper to a particular processor family are found in cpukit/score/cpu. Support code for this port as well as processor dependent code which may be reused across multiple Board Support Packages is found in c/src/lib/libcpu.

XXX list the files and directories here

3 CPU Model Variations

XXX enhance using portability presentation from CS595 class. See general/portability.ppt.

Since the text in the next section was written, RTEMS view of portability has grown to distinguish totally portable, CPU family dependent, CPU model dependent, peripheral chip dependent and board dependent. This text was part of a larger paper that did not even cover portability completely as it existed when this was written and certainly is out of date now. :)

3.1 Overview of RTEMS Portability

RTEMS was designed to be a highly portable, reusable software component. This reflects the fundamental nature of embedded systems in which hardware components, ranging from boards to peripherals to even the processor itself, are selected specifically to meet the requirements of a particular project.

3.1.1 Processor Families

Since there are a wide variety of embedded systems, there are a wide variety of processors targeting embedded systems. RTEMS alleviates some of the burden on the embedded systems programmer by providing a consistent, high-performance environment regardless of the target processor. RTEMS has been ported to a variety of microprocessor families including:

- Motorola ColdFire
- Motorola MC68xxx
- Motorola MC683xx
- Intel ix86 (i386, i486, Pentium and above)
- ARM
- MIPS
- PowerPC 4xx, 5xx, 6xx, 7xx, 8xx, and 84xx
- SPARC
- Hitachi H8/300
- Hitachi SH

In addition, there is a port of RTEMS to UNIX that uses standard UNIX services to simulate the embedded environment.

Each RTEMS port supplies a well-defined set of services that are the foundation for the highly portable RTEMS and POSIX API implementations. When porting to a new processor family, one must provide the processor dependent implementation of these services. This set of processor dependent core services includes software to perform interrupt dispatching, context switches, and manipulate task register sets.

The RTEMS approach to handling varying processor models reflects the approach taken by the designers of the processors themselves. In each processor family, there is a core architecture that must be implemented on all processor models within the family to provide any level of compatibility. Many of the modern RISC architectures refer to this as the Architectural Definition. The Architectural Definition is intended to be independent of any particular implementation. Additionally, there is a feature set which is allowed to vary in a defined way amongst the processor models. These feature sets may be defined as Optional in the Architectural Definition, be left as implementation defined characteristics, or be processor model specific extensions. Support for floating point, virtual memory, and low power mode are common Optional features included in an Architectural Definition.

The processor family dependent software in RTEMS includes a definition of which features are present in each supported processor model. This often makes adding support for a new processor model within a supported family as simple as determining which features are present in the new processor implementation. If the new processor model varies in a way previously unaccounted for, then this must be addressed. This could be the result of a new Optional feature set being added to the Architectural Definition. Alternatively, this particular processor model could have a new and different implementation of a feature left as undefined in the Architectural Definition. This would require software to be written to utilize that feature.

There is a relatively small set of features that may vary in a processor family. As the number of processor models in the family grow, the addition of each new model only requires adding an entry for the new model to the single feature table. It does not require searching for every conditional based on processor model and adding the new model in the appropriate place. This significantly eases the burden of adding a new processor model as it centralizes and logically simplifies the process.

3.1.2 Boards

Being portable both between models within a processor family and across processor families is not enough to address the needs of embedded systems developers. Custom board development is the norm for embedded systems. Each of these boards is optimized for a particular project. The processor and peripheral set have been chosen to meet a particular set of system requirements. The tools in the embedded systems developers toolbox must support their projects unique board. RTEMS addresses this issue via the Board Support Package.

RTEMS segregates board specific code to make it possible for the embedded systems developer to easily replace and customize this software. A minimal Board Support Package includes device drivers for a clock tick, console I/O, and a benchmark timer (optional) as well as startup and miscellaneous support code. The Board Support Package for a project may be extended to include the device drivers for any peripherals on the custom board.

3.1.3 Applications

One important design goal of RTEMS was to provide a bridge between the application software and the target hardware. Most hardware dependencies for real-time applications can be localized to the low level device drivers which provide an abstracted view of the hardware. The RTEMS I/O interface manager provides an efficient tool for incorporating these hardware dependencies into the system while simultaneously providing a general mechanism to the application code that accesses them. A well designed real-time system can benefit from this architecture by building a rich library of standard application components which can be used repeatedly in other real-time projects. The following figure illustrates how RTEMS serves as a buffer between the project dependent application code and the target hardware.

3.2 Coding Issues

XXX deal with this as it applies to score/cpu. Section name may be bad.

4 CPU Initialization

This section describes the general CPU and system initialization sequence as it pertains to the CPU dependent code.

4.1 Introduction

XXX general startup sequence description rewritten to make it more applicable to CPU depdent code in executive

4.2 Initializing the CPU

The _CPU_Initialize routine performs processor dependent initialization.

```
void _CPU_Initialize(
  void            (*thread_dispatch)  /* may be ignored */
)
```

The thread_dispatch argument is the address of the entry point for the routine called at the end of an ISR once it has been decided a context switch is necessary. On some compilation systems it is difficult to call a high-level language routine from assembly. Providing the address of the _Thread_ISR_Dispatch routine allows the porter an easy way to obtain this critical address and thus provides an easy way to work around this limitation on these systems.

If you encounter this problem save the entry point in a CPU dependent variable as shown below:

```
_CPU_Thread_dispatch_pointer = thread_dispatch;
```

During the initialization of the context for tasks with floating point, the CPU dependent code is responsible for initializing the floating point context. If there is not an easy way to initialize the FP context during Context_Initialize, then it is usually easier to save an "uninitialized" FP context here and copy it to the task's during Context_Initialize. If this technique is used to initialize the FP contexts, then it is important to ensure that the state of the floating point unit is in a coherent, initialized state.

5 Interrupts

5.1 Introduction

5.2 Interrupt Levels

RTEMS is designed assuming that a CPU family has a level associated with interrupts. Interrupts below the current interrupt level are masked and do not interrupt the CPU until the interrupt level is lowered. This design provides for 256 distinct interrupt levels even though most CPU implementations support far fewer levels. Interrupt level 0 is assumed to map to the hardware settings for all interrupts enabled.

Over the years that RTEMS has been available, there has been much discussion on how to handle CPU families which support very few interrupt levels such as the i386, PowerPC, and HP-PA RISC. XXX

5.2.1 Interrupt Level Mask

The CPU_MODES_INTERRUPT_MASK macro defines the number of bits actually used in the interrupt field of the task mode. How those bits map to the CPU interrupt levels is defined by the routine _CPU_ISR_Set_level().

The following illustrates how the CPU_MODES_INTERRUPT_MASK is set on a CPU family like the Intel i386 where the CPU itself only recognizes two interrupt levels - enabled and disabled.

```
#define CPU_MODES_INTERRUPT_MASK   0x00000001
```

5.2.2 Obtaining the Current Interrupt Level

The _CPU_ISR_Get_level function returns the current interrupt level.

```
uint32_t _CPU_ISR_Get_level( void )
```

5.2.3 Set the Interrupt Level

The _CPU_ISR_Set_level routine maps the interrupt level in the Classic API task mode onto the hardware that the CPU actually provides. Currently, interrupt levels that do not map onto the CPU in a generic fashion are undefined. Someday, it would be nice if these were "mapped" by the application via a callout. For example, the Motorola m68k has 8 levels 0 - 7, and levels 8 - 255 are currently undefined. Levels 8 - 255 would be available for bsp/application specific meaning. This could be used to manage a programmable interrupt controller via the rtems_task_mode directive.

The following is a dummy implementation of the _CPU_ISR_Set_level routine:

```
#define _CPU_ISR_Set_level( new_level ) \
  { \
  }
```

The following is the implementation from the Motorola M68K:

```
XXX insert m68k implementation here
```

5.2.4 Disable Interrupts

The _CPU_ISR_Disable routine disable all external interrupts. It returns the previous interrupt level in the single parameter _isr_cookie. This routine is used to disable interrupts during a critical section in the RTEMS executive. Great care is taken inside the executive to ensure that interrupts are disabled for a minimum length of time. It is important to note that the way the previous level is returned forces the implementation to be a macro that translates to either inline assembly language or a function call whose return value is placed into _isr_cookie.

It is important for the porter to realize that the value of _isr_cookie has no defined meaning except that it is the most convenient format for the _CPU_ISR_Disable, _CPU_ISR_Enable, and _CPU_ISR_Disable routines to manipulate. It is typically the contents of the processor status register. It is NOT the same format as manipulated by the _CPU_ISR_Get_level and _CPU_ISR_Set_level routines. The following is a dummy implementation that simply sets the previous level to 0.

```
#define _CPU_ISR_Disable( _isr_cookie ) \
  { \
    (_isr_cookie) = 0;   /* do something to prevent warnings */ \
  }
```

The following is the implementation from the Motorola M68K port:

```
XXX insert m68k port here
```

5.2.5 Enable Interrupts

The _CPU_ISR_Enable routines enables interrupts to the previous level (returned by _CPU_ISR_Disable). This routine is invoked at the end of an RTEMS critical section to reenable interrupts. The parameter _level is not modified but indicates that level that interrupts should be enabled to. The following illustrates a dummy implementation of the _CPU_ISR_Enable routine:

```
#define _CPU_ISR_Enable( _isr_cookie )  \
  { \
  }
```

The following is the implementation from the Motorola M68K port:

```
XXX insert m68k version here
```

5.2.6 Flash Interrupts

The _CPU_ISR_Flash routine temporarily restores the interrupt to _level before immediately disabling them again. This is used to divide long RTEMS critical sections into two or more parts. This routine is always preceded by a call to _CPU_ISR_Disable and followed by a call to _CPU_ISR_Enable. The parameter _level is not modified.

The following is a dummy implementation of the _CPU_ISR_Flash routine:

```
#define _CPU_ISR_Flash( _isr_cookie ) \
  { \
  }
```

The following is the implementation from the Motorola M68K port:

```
XXX insert m68k version here
```

5.3 Interrupt Stack Management

5.3.1 Hardware or Software Managed Interrupt Stack

The setting of the CPU_HAS_SOFTWARE_INTERRUPT_STACK indicates whether the interrupt stack is managed by RTEMS in software or the CPU has direct support for an interrupt stack. If RTEMS is to manage a dedicated interrupt stack in software, then this macro should be set to TRUE and the memory for the software managed interrupt stack is allocated in `_ISR_Handler_initialization`. If this macro is set to FALSE, then RTEMS assumes that the hardware managed interrupt stack is supported by this CPU. If the CPU has a hardware managed interrupt stack, then the porter has the option of letting the BSP allcoate and initialize the interrupt stack or letting RTEMS do this. If RTEMS is to allocate the memory for the interrupt stack, then the macro CPU_ALLOCATE_INTERRUPT_STACK should be set to TRUE. If this macro is set to FALSE, then it is the responsibility of the BSP to allocate the memory for this stack and initialize it.

If the CPU does not support a dedicated interrupt stack, then the porter has two options: (1) execute interrupts on the stack of the interrupted task, and (2) have RTEMS manage a dedicated interrupt stack.

NOTE: If CPU_HAS_SOFTWARE_INTERRUPT_STACK is TRUE, then the macro CPU_ALLOCATE_INTERRUPT_STACK should also be set to TRUE.

Only one of CPU_HAS_SOFTWARE_INTERRUPT_STACK and CPU_HAS_HARDWARE_INTERRUPT_STACK should be set to TRUE. It is possible that both are FALSE for a particular CPU. Although it is unclear what that would imply about the interrupt processing procedure on that CPU.

5.3.2 Allocation of Interrupt Stack Memory

Whether or not the interrupt stack is hardware or software managed, RTEMS may allocate memory for the interrupt stack from the Executive Workspace. If RTEMS is going to allocate the memory for a dedicated interrupt stack in the Interrupt Manager, then the macro CPU_ALLOCATE_INTERRUPT_STACK should be set to TRUE.

NOTE: This should be TRUE is CPU_HAS_SOFTWARE_INTERRUPT_STACK is TRUE.

```
#define CPU_ALLOCATE_INTERRUPT_STACK TRUE
```

If the CPU_HAS_SOFTWARE_INTERRUPT_STACK macro is set to TRUE, then RTEMS automatically allocates the stack memory in the initialization of the Interrupt Manager and the switch to that stack is performed in `_ISR_Handler` on the outermost interrupt. The _CPU_Interrupt_stack_low and _CPU_Interrupt_stack_high variables contain the addresses of the the lowest and highest addresses of the memory allocated for the interrupt stack. Although technically only one of these addresses is required to switch to the interrupt stack, by always providing both addresses, the port has more options avaialble to it without requiring modifications to the portable parts of the executive. Whether the stack grows up or down, this give the CPU dependent code the option of picking the version it wants to use.

```
SCORE_EXTERN void               *_CPU_Interrupt_stack_low;
SCORE_EXTERN void               *_CPU_Interrupt_stack_high;
```

NOTE: These two variables are required if the macro CPU_HAS_SOFTWARE_INTERRUPT_STACK is defined as TRUE.

5.3.3 Install the Interrupt Stack

The _CPU_Install_interrupt_stack routine XXX

This routine installs the hardware interrupt stack pointer.

NOTE: It need only be provided if CPU_HAS_HARDWARE_INTERRUPT_STAC is TRUE.

```
void _CPU_Install_interrupt_stack( void )
```

5.4 ISR Installation

5.4.1 Install a Raw Interrupt Handler

The _CPU_ISR_install_raw_handler XXX

```
void _CPU_ISR_install_raw_handler(
  unsigned32  vector,
  proc_ptr    new_handler,
  proc_ptr   *old_handler
)
```

This is where we install the interrupt handler into the "raw" interrupt table used by the CPU to dispatch interrupt handlers.

5.4.2 Interrupt Context

5.4.3 Maximum Number of Vectors

There are two related macros used to defines the number of entries in the _ISR_Vector_table managed by RTEMS. The macro CPU_INTERRUPT_NUMBER_OF_VECTORS is the actual number of vectors supported by this CPU model. The second macro is the CPU_INTERRUPT_MAXIMUM_VECTOR_NUMBER. Since the table is zero-based, this indicates the highest vector number which can be looked up in the table and mapped into a user provided handler.

```
#define CPU_INTERRUPT_NUMBER_OF_VECTORS      32
#define CPU_INTERRUPT_MAXIMUM_VECTOR_NUMBER \
        (CPU_INTERRUPT_NUMBER_OF_VECTORS - 1)
```

5.4.4 Install RTEMS Interrupt Handler

The _CPU_ISR_install_vector routine installs the RTEMS handler for the specified vector.

XXX Input parameters: vector - interrupt vector number old_handler - former ISR for this vector number new_handler - replacement ISR for this vector number

```
void _CPU_ISR_install_vector(
  unsigned32  vector,
  proc_ptr    new_handler,
  proc_ptr   *old_handler
)

*old_handler = _ISR_Vector_table[ vector ];
```

If the interrupt vector table is a table of pointer to isr entry points, then we need to install the appropriate RTEMS interrupt handler for this vector number.

```
_CPU_ISR_install_raw_handler( vector, new_handler, old_handler );
```

We put the actual user ISR address in _ISR_vector_table. This will be used by the `_ISR_Handler` so the user gets control.

```
_ISR_Vector_table[ vector ] = new_handler;
```

5.5 Interrupt Processing

5.5.1 Interrupt Frame Data Structure

When an interrupt occurs, it is the responsibility of the interrupt dispatching software to save the context of the processor such that an ISR written in a high-level language (typically C) can be invoked without damaging the state of the task that was interrupted. In general, this results in the saving of registers which are NOT preserved across subroutine calls as well as any special interrupt state. A port should define the `CPU_Interrupt_frame` structure so that application code can examine the saved state.

```
typedef struct {
    unsigned32 not_preserved_register_1;
    unsigned32 special_interrupt_register;
} CPU_Interrupt_frame;
```

5.5.2 Interrupt Dispatching

The `_ISR_Handler` routine provides the RTEMS interrupt management.

```
void _ISR_Handler()
```

This discussion ignores a lot of the ugly details in a real implementation such as saving enough registers/state to be able to do something real. Keep in mind that the goal is to invoke a user's ISR handler which is written in C. That ISR handler uses a known set of registers thus allowing the ISR to preserve only those that would normally be corrupted by a subroutine call.

Also note that the exact order is to a large extent flexible. Hardware will dictate a sequence for a certain subset of `_ISR_Handler` while requirements for setting the RTEMS state variables that indicate the interrupt nest level (`_ISR_Nest_level`) and dispatching disable level (`_Thread_Dispatch_disable_level`) will also restrict the allowable order.

Upon entry to `_ISR_Handler`, `_Thread_Dispatch_disable_level` is zero if the interrupt occurred while outside an RTEMS service call. Conversely, it will be non-zero if interrupting an RTEMS service call. Thus, `_Thread_Dispatch_disable_level` will always be greater than or equal to `_ISR_Nest_level` and not strictly equal.

Upon entry to the "common" `_ISR_Handler`, the vector number must be available. On some CPUs the hardware puts either the vector number or the offset into the vector table for this ISR in a known place. If the hardware does not provide this information, then the assembly portion of RTEMS for this port will contain a set of distinct interrupt entry points which somehow place the vector number in a known place (which is safe if another interrupt nests this one) and branches to `_ISR_Handler`.

```
save some or all context on stack
may need to save some special interrupt information for exit

#if ( CPU_HAS_SOFTWARE_INTERRUPT_STACK == TRUE )
    if ( _ISR_Nest_level == 0 )
        switch to software interrupt stack
#endif
_ISR_Nest_level++;
_Thread_Dispatch_disable_level++;
(*_ISR_Vector_table[ vector ])( vector );
--_ISR_Nest_level;
if ( _ISR_Nest_level )
    goto the label "exit interrupt (simple case)"
#if ( CPU_HAS_SOFTWARE_INTERRUPT_STACK == TRUE )
    restore stack
#endif

if ( _Thread_Dispatch_disable_level )
   _ISR_Signals_to_thread_executing = FALSE;
    goto the label "exit interrupt (simple case)"

if ( _Context_Switch_necessary || _ISR_Signals_to_thread_executing )
   _ISR_Signals_to_thread_executing = FALSE;
   call _Thread_Dispatch() or prepare to return to _ISR_Dispatch
   prepare to get out of interrupt
   return from interrupt  (maybe to _ISR_Dispatch)

LABEL "exit interrupt (simple case):
 prepare to get out of interrupt
 return from interrupt
```

Some ports have the special routine `_ISR_Dispatch` because the CPU has a special "interrupt mode" and RTEMS must switch back to the task stack and/or non-interrupt mode before invoking `_Thread_Dispatch`. For example, consider the MC68020 where upon return from the outermost interrupt, the CPU must switch from the interrupt stack to the master stack before invoking `_Thread_Dispatch`. `_ISR_Dispatch` is the special port specific wrapper for `_Thread_Dispatch` used in this case.

5.5.3 ISR Invoked with Frame Pointer

Does the RTEMS invoke the user's ISR with the vector number and a pointer to the saved interrupt frame (1) or just the vector number (0)?

```
#define CPU_ISR_PASSES_FRAME_POINTER 0
```

NOTE: It is desirable to include a pointer to the interrupt stack frame as an argument to the interrupt service routine. Eventually, it would be nice if all ports included this parameter.

5.5.4 Pointer to _Thread_Dispatch Routine

With some compilation systems, it is difficult if not impossible to call a high-level language routine from assembly language. This is especially true of commercial Ada compilers and name mangling C++ ones. This variable can be optionally defined by the CPU porter and contains the address of the routine _Thread_Dispatch. This can make it easier to invoke that routine at the end of the interrupt sequence (if a dispatch is necessary).

```
void (*_CPU_Thread_dispatch_pointer)();
```

6 Task Context Management

6.1 Introduction

XXX

6.2 Task Stacks

XXX

6.2.1 Direction of Stack Growth

The CPU_STACK_GROWS_UP macro is set based upon the answer to the following question: Does the stack grow up (toward higher addresses) or down (toward lower addresses)? If the stack grows upward in memory, then this macro should be set to TRUE. Otherwise, it should be set to FALSE to indicate that the stack grows downward toward smaller addresses.

The following illustrates how the CPU_STACK_GROWS_UP macro is set:

```
#define CPU_STACK_GROWS_UP               TRUE
```

6.2.2 Minimum Task Stack Size

The CPU_STACK_MINIMUM_SIZE macro should be set to the minimum size of each task stack. This size is specified as the number of bytes. This minimum stack size should be large enough to run all RTEMS tests. The minimum stack size is chosen such that a "reasonable" small application should not have any problems. Choosing a minimum stack size that is too small will result in the RTEMS tests "blowing" their stack and not executing properly.

There are many reasons a task could require a stack size larger than the minimum. For example, a task could have a very deep call path or declare large data structures on the stack. Tasks which utilize C++ exceptions tend to require larger stacks as do Ada tasks.

The following illustrates setting the minimum stack size to 4 kilobytes per task.

```
#define CPU_STACK_MINIMUM_SIZE          (1024*4)
```

6.2.3 Stack Alignment Requirements

The CPU_STACK_ALIGNMENT macro is set to indicate the byte alignment requirement for the stack. This alignment requirement may be stricter than that for the data types alignment specified by CPU_ALIGNMENT. If the CPU_ALIGNMENT is strict enough for the stack, then this should be set to 0.

The following illustrates how the CPU_STACK_ALIGNMENT macro should be set when there are no special requirements:

```
#define CPU_STACK_ALIGNMENT        0
```

NOTE: This must be a power of 2 either 0 or greater than CPU_ALIGNMENT. [XXX is this true?]

6.3 Task Context

Associated with each task is a context that distinguishes it from other tasks in the system and logically gives it its own scratch pad area for computations. In addition, when an interrupt occurs some processor context information must be saved and restored. This is managed in RTEMS as three items:

- Basic task level context (e.g. the Context_Control structure)
- Floating point task context (e.g. Context_Control_fp structure)
- Interrupt level context (e.g. the Context_Control_interrupt structure)

The integer and floating point context structures and the routines that manipulate them are discussed in detail in this section, while the interrupt level context structure is discussed in the XXX.

Additionally, if the GNU debugger gdb is to be made aware of RTEMS tasks for this CPU, then care should be used in designing the context area.

```
typedef struct {
    unsigned32 special_interrupt_register;
} CPU_Interrupt_frame;
```

6.3.1 Basic Context Data Structure

The Context_Control data structure contains the basic integer context of a task. In addition, this context area contains stack and frame pointers, processor status register(s), and any other registers that are normally altered by compiler generated code. In addition, this context must contain the processor interrupt level since the processor interrupt level is maintained on a per-task basis. This is necessary to support the interrupt level portion of the task mode as provided by the Classic RTEMS API.

On some processors, it is cost-effective to save only the callee preserved registers during a task context switch. This means that the ISR code needs to save those registers which do not persist across function calls. It is not mandatory to make this distinctions between the caller/callee saves registers for the purpose of minimizing context saved during task switch and on interrupts. If the cost of saving extra registers is minimal, simplicity is the choice. Save the same context on interrupt entry as for tasks in this case.

The Context_Control data structure should be defined such that the order of elements results in the simplest, most efficient implementation of XXX. A typical implementation starts with a definition such as the following:

```
typedef struct {
    unsigned32 some_integer_register;
    unsigned32 another_integer_register;
    unsigned32 some_system_register;
} Context_Control;
```

6.3.2 Initializing a Context

The _CPU_Context_Initialize routine initializes the context to a state suitable for starting a task after a context restore operation. Generally, this involves:

- setting a starting address,
- preparing the stack,
- preparing the stack and frame pointers,
- setting the proper interrupt level in the context, and
- initializing the floating point context

This routine generally does not set any unnecessary register in the context. The state of the "general data" registers is undefined at task start time. The _CPU_Context_initialize routine is prototyped as follows:

```
void _CPU_Context_Initialize(
    Context_Control *_the_context,
    void            *_stack_base,
    unsigned32       _size,
    unsigned32       _isr,
    void            *_entry_point,
    unsigned32       _is_fp
);
```

The `is_fp` parameter is TRUE if the thread is to be a floating point thread. This is typically only used on CPUs where the FPU may be easily disabled by software such as on the SPARC where the PSR contains an enable FPU bit. The use of an FPU enable bit allows RTEMS to ensure that a non-floating point task is unable to access the FPU. This guarantees that a deferred floating point context switch is safe.

The `_stack_base` parameter is the base address of the memory area allocated for use as the task stack. It is critical to understand that `_stack_base` may not be the starting stack pointer for this task. On CPU families where the stack grows from high addresses to lower ones, (i.e. `CPU_STACK_GROWS_UP` is FALSE) the starting stack point will be near the end of the stack memory area or close to `_stack_base + _size`. Even on CPU families where the stack grows from low to higher addresses, there may be some required outermost stack frame that must be put at the address `_stack_base`.

The `_size` parameter is the requested size in bytes of the stack for this task. It is assumed that the memory area `_stack_base` is of this size.

XXX explain other parameters and check prototype

6.3.3 Performing a Context Switch

The _CPU_Context_switch performs a normal non-FP context switch from the context of the current executing thread to the context of the heir thread.

```
void _CPU_Context_switch(
  Context_Control  *run,
  Context_Control  *heir
);
```

This routine begins by saving the current state of the CPU (i.e. the context) in the context area at `run`. Then the routine should load the CPU context pointed to by `heir`. Loading the new context will cause a branch to its task code, so the task that invoked `_CPU_Context_switch` will not run for a while. When, eventually, a context switch is made to load context from `*run` again, this task will resume and `_CPU_Context_switch` will return to its caller.

Care should be exercise when writing this routine. All registers assumed to be preserved across subroutine calls must be preserved. These registers may be saved in the task's context area or on its stack. However, the stack pointer and address to resume executing the task at must be included in the context (normally the subroutine return address to the caller of `_Thread_Dispatch`. The decision of where to store the task's context is based on numerous factors including the capabilities of the CPU architecture itself and simplicity as well as external considerations such as debuggers wishing to examine a task's context. In this case, it is often simpler to save all data in the context area.

Also there may be special considerations when loading the stack pointers or interrupt level of the incoming task. Independent of CPU specific considerations, if some context is saved on the task stack, then the porter must ensure that the stack pointer is adjusted **BEFORE** to make room for this context information before the information is written. Otherwise, an interrupt could occur writing over the context data. The following is an example of an **INCORRECT** sequence:

```
save part of context beyond current top of stack
interrupt pushes context -- overwriting written context
interrupt returns
adjust stack pointer
```

6.3.4 Restoring a Context

The _CPU_Context_restore routine is generally used only to restart the currently executing thread (i.e. self) in an efficient manner. In many ports, it can simply be a label in _CPU_Context_switch. It may be unnecessary to reload some registers.

```
void _CPU_Context_restore(
  Context_Control *new_context
);
```

6.3.5 Restarting the Currently Executing Task

The _CPU_Context_Restart_self is responsible for somehow restarting the currently executing task. If you are lucky when porting RTEMS, then all that is necessary is restoring the context. Otherwise, there will need to be a routine that does something special in this case. Performing a _CPU_Context_Restore on the currently executing task after reinitializing that context should work on most ports. It will not work if restarting self conflicts with the stack frame assumptions of restoring a context.

The following is an implementation of _CPU_Context_Restart_self that can be used when no special handling is required for this case.

```
#define _CPU_Context_Restart_self( _the_context ) \
   _CPU_Context_restore( (_the_context) )
```

XXX find a port which does not do it this way and include it here

6.4 Floating Point Context

6.4.1 CPU_HAS_FPU Macro Definition

The CPU_HAS_FPU macro is set based on the answer to the question: Does the CPU have hardware floating point? If the CPU has an FPU, then this should be set to TRUE. Otherwise, it should be set to FALSE. The primary implication of setting this macro to TRUE is that it indicates that tasks may have floating point contexts. In the Classic API, this means that the RTEMS_FLOATING_POINT task attribute specified as part of rtems_task_create is supported on this CPU. If CPU_HAS_FPU is set to FALSE, then no tasks or threads may be floating point and the RTEMS_FLOATING_POINT task attribute is ignored. On an API such as POSIX where all threads implicitly have a floating point context, then the setting of this macro determines whether every POSIX thread has a floating point context.

The following example illustrates how the CPU_HARDWARE_FP (XXX macro name is varying) macro is set based on the CPU family dependent macro.

```
#if ( THIS_CPU_FAMILY_HAS_FPU == 1 ) /* where THIS_CPU_FAMILY */
                                     /* might be M68K */
#define CPU_HARDWARE_FP     TRUE
#else
#define CPU_HARDWARE_FP     FALSE
#endif
```

The macro name THIS_CPU_FAMILY_HAS_FPU should be made CPU specific. It indicates whether or not this CPU model has FP support. For example, the definition of the i386ex and i386sx CPU models would set I386_HAS_FPU to FALSE to indicate that these CPU models are i386's without an i387 and wish to leave floating point support out of RTEMS when built for the i386_nofp processor model. On a CPU with a built-in FPU like the i486, this would be defined as TRUE.

On some processor families, the setting of the THIS_CPU_FAMILY_HAS_FPU macro may be derived from compiler predefinitions. This can be used when the compiler distinguishes the individual CPU models for this CPU family as distinctly as RTEMS requires. Often RTEMS needs to need more about the CPU model than the compiler because of differences at the system level such as caching, interrupt structure.

6.4.2 CPU_ALL_TASKS_ARE_FP Macro Setting

The CPU_ALL_TASKS_ARE_FP macro is set to TRUE or FALSE based upon the answer to the following question: Are all tasks RTEMS_FLOATING_POINT tasks implicitly? If this macro is set TRUE, then all tasks and threads are assumed to have a floating point context. In the Classic API, this is equivalent to setting the RTEMS_FLOATING_POINT task attribute on all rtems_task_create calls. If the CPU_ALL_TASKS_ARE_FP macro is set to FALSE, then the RTEMS_FLOATING_POINT task attribute in the Classic API is honored.

The rationale for this macro is that if a function that an application developer would not think utilize the FP unit DOES, then one can not easily predict which tasks will use the FP hardware. In this case, this option should be TRUE. So far, the only CPU families for which this macro has been to TRUE are the HP PA-RISC and PowerPC. For the HP PA-RISC, the HP C compiler and gcc both implicitly use the floating point registers to perform integer multiplies. For the PowerPC, this feature macro is set to TRUE because the printf routine saves a floating point register whether or not a floating point number is actually printed. If the newlib implementation of printf were restructured to avoid this, then the PowerPC port would not have to have this option set to TRUE.

The following example illustrates how the CPU_ALL_TASKS_ARE_FP is set on the PowerPC. On this CPU family, this macro is set to TRUE if the CPU model has hardware floating point.

```
#if (CPU_HARDWARE_FP == TRUE)
#define CPU_ALL_TASKS_ARE_FP     TRUE
#else
#define CPU_ALL_TASKS_ARE_FP     FALSE
#endif
```

NOTE: If CPU_HARDWARE_FP is FALSE, then this should be FALSE as well.

6.4.3 CPU_USE_DEFERRED_FP_SWITCH Macro Setting

The CPU_USE_DEFERRED_FP_SWITCH macro is set based upon the answer to the following question: Should the saving of the floating point registers be deferred until a context switch is made to another different floating point task? If the floating point context will not be stored until necessary, then this macro should be set to TRUE. When set to TRUE, the floating point context of a task will remain in the floating point registers and not disturbed until another floating point task is switched to.

If the CPU_USE_DEFERRED_FP_SWITCH is set to FALSE, then the floating point context is saved each time a floating point task is switched out and restored when the next floating point task is restored. The state of the floating point registers between those two operations is not specified.

There are a couple of known cases where the port should not defer saving the floating point context. The first case is when the compiler generates instructions that use the FPU when floating point is not actually used. This occurs on the HP PA-RISC for example when an integer multiply is performed. On the PowerPC, the printf routine includes a save of a floating point register to support printing floating point numbers even if the path that actually prints the floating point number is not invoked. In both of these cases, deferred floating point context switches can not be used. If the floating point context has to be saved as part of interrupt dispatching, then it may also be necessary to disable deferred context switches.

Setting this flag to TRUE results in using a different algorithm for deciding when to save and restore the floating point context. The deferred FP switch algorithm minimizes the number of times the FP context is saved and restored. The FP context is not saved until a context switch is made to another, different FP task. Thus in a system with only one FP task, the FP context will never be saved or restored.

The following illustrates setting the CPU_USE_DEFERRED_FP_SWITCH macro on a processor family such as the M68K or i386 which can use deferred floating point context switches.

```
#define CPU_USE_DEFERRED_FP_SWITCH       TRUE
```

6.4.4 Floating Point Context Data Structure

The Context_Control_fp contains the per task information for the floating point unit. The organization of this structure may be a list of floating point registers along with any floating point control and status registers or it simply consist of an array of a fixed number of bytes. Defining the floating point context area as an array of bytes is done when the floating point context is dumped by a "FP save context" type instruction and the format is either not completely defined by the CPU documentation or the format is not critical for the implementation of the floating point context switch routines. In this case, there is no need to figure out the exact format – only the size. Of course, although this is enough information for RTEMS, it is probably not enough for a debugger such as gdb. But that is another problem.

```
typedef struct {
    double      some_float_register;
} Context_Control_fp;
```

On some CPUs with hardware floating point support, the Context_Control_fp structure will not be used.

6.4.5 Size of Floating Point Context Macro

The CPU_CONTEXT_FP_SIZE macro is set to the size of the floating point context area. On some CPUs this will not be a "sizeof" because the format of the floating point area is not defined – only the size is. This is usually on CPUs with a "floating point save context" instruction. In general, though it is easier to define the structure as a "sizeof" operation and define the Context_Control_fp structure to be an area of bytes of the required size in this case.

```
#define CPU_CONTEXT_FP_SIZE sizeof( Context_Control_fp )
```

6.4.6 Start of Floating Point Context Area Macro

The _CPU_Context_Fp_start macro is used in the XXX routine and allows the initial pointer into a floating point context area (used to save the floating point context) to be at an arbitrary place in the floating point context area. This is necessary because some FP units are designed to have their context saved as a stack which grows into lower addresses. Other FP units can be saved by simply moving registers into offsets from the base of the context area. Finally some FP units provide a "dump context" instruction which could fill in from high to low or low to high based on the whim of the CPU designers. Regardless, the address at which that floating point context area pointer should start within the actual floating point context area varies between ports and this macro provides a clean way of addressing this.

This is a common implementation of the _CPU_Context_Fp_start routine which is suitable for many processors. In particular, RISC processors tend to use this implementation since the floating point context is saved as a sequence of store operations.

```
#define _CPU_Context_Fp_start( _base, _offset ) \
   ( (void *) _Addresses_Add_offset( (_base), (_offset) ) )
```

In contrast, the m68k treats the floating point context area as a stack which grows downward in memory. Thus the following implementation of _CPU_Context_Fp_start is used in that port:

```
XXX insert m68k version here
```

6.4.7 Initializing a Floating Point Context

The _CPU_Context_Initialize_fp routine initializes the floating point context area passed to it to. There are a few standard ways in which to initialize the floating point context. The simplest, and least deterministic behaviorally, is to do nothing. This leaves the FPU in a random state and is generally not a suitable way to implement this routine. The second common implementation is to place a "null FP status word" into some status/control register in the FPU. This mechanism is simple and works on many FPUs. Another common way is to initialize the FPU to a known state during _CPU_Initialize and save the context (using _CPU_Context_save_fp_context) into the special floating point context _CPU_Null_fp_context. Then all that is required to initialize a floating point context is to copy _CPU_Null_fp_context to the destination floating point context passed to it. The following example implementation shows how to accomplish this:

```
#define _CPU_Context_Initialize_fp( _destination ) \
  { \
   *((Context_Control_fp *) *((void **) _destination)) = \
       _CPU_Null_fp_context; \
  }
```

The _CPU_Null_fp_context is optional. A port need only include this variable when it uses the above mechanism to initialize a floating point context. This is typically done on CPUs where it is difficult to generate an "uninitialized" FP context. If the port requires this variable, then it is declared as follows:

```
Context_Control_fp  _CPU_Null_fp_context;
```

6.4.8 Saving a Floating Point Context

The _CPU_Context_save_fp_context routine is responsible for saving the FP context at *fp_context_ptr. If the point to load the FP context from is changed then the pointer is modified by this routine.

Sometimes a macro implementation of this is in cpu.h which dereferences the ** and a similarly named routine in this file is passed something like a (Context_Control_fp *). The general rule on making this decision is to avoid writing assembly language.

```
void _CPU_Context_save_fp(
  void **fp_context_ptr
)
```

6.4.9 Restoring a Floating Point Context

The _CPU_Context_restore_fp_context is responsible for restoring the FP context at *fp_context_ptr. If the point to load the FP context from is changed then the pointer is modified by this routine.

Sometimes a macro implementation of this is in cpu.h which dereferences the ** and a similarly named routine in this file is passed something like a (Context_Control_fp *). The general rule on making this decision is to avoid writing assembly language.

```
void _CPU_Context_restore_fp(
  void **fp_context_ptr
);
```

7 IDLE Thread

7.1 Does Idle Thread Have a Floating Point Context?

The setting of the macro CPU_IDLE_TASK_IS_FP is based on the answer to the question: Should the IDLE task have a floating point context? If the answer to this question is TRUE, then the IDLE task has a floating point context associated. This is equivalent to creating a task in the Classic API (using rtems_task_create) as a RTEMS_FLOATING_POINT task. If CPU_IDLE_TASK_IS_FP is set to TRUE, then a floating point context switch occurs when the IDLE task is switched in and out. This adds to the execution overhead of the system but is necessary on some ports.

If FALSE, then the IDLE task does not have a floating point context.

NOTE: Setting CPU_IDLE_TASK_IS_FP to TRUE negatively impacts the time required to preempt the IDLE task from an interrupt because the floating point context must be saved as part of the preemption.

The following illustrates how to set this macro:

```
#define CPU_IDLE_TASK_IS_FP      FALSE
```

7.2 CPU Dependent Idle Thread Body

7.2.1 CPU_PROVIDES_IDLE_THREAD_BODY Macro Setting

The CPU_PROVIDES_IDLE_THREAD_BODY macro setting is based upon the answer to the question: Does this port provide a CPU dependent IDLE task implementation? If the answer to this question is yes, then the CPU_PROVIDES_IDLE_THREAD_BODY macro should be set to TRUE, and the routine _CPU_Thread_Idle_body must be provided. This routine overrides the default IDLE thread body of _Thread_Idle_body. If the CPU_PROVIDES_IDLE_THREAD_BODY macro is set to FALSE, then the generic _Thread_Idle_body is the default IDLE thread body for this port. Regardless of whether or not a CPU dependent IDLE thread implementation is provided, the BSP can still override it.

This is intended to allow for supporting processors which have a low power or idle mode. When the IDLE thread is executed, then the CPU can be powered down when the processor is idle.

The order of precedence for selecting the IDLE thread body is:

1. BSP provided
2. CPU dependent (if provided)
3. generic (if no BSP and no CPU dependent)

The following illustrates setting the CPU_PROVIDES_IDLE_THREAD_BODY macro:

```
#define CPU_PROVIDES_IDLE_THREAD_BODY    TRUE
```

Implementation details of a CPU model specific IDLE thread body are in the next section.

7.2.2 Idle Thread Body

The _CPU_Thread_Idle_body routine only needs to be provided if the porter wishes to include a CPU dependent IDLE thread body. If the port includes a CPU dependent implementation of the IDLE thread body, then the CPU_PROVIDES_IDLE_THREAD_BODY macro should be defined to TRUE. This routine is prototyped as follows:

```
void *_CPU_Thread_Idle_body( uint32_t );
```

As mentioned above, RTEMS does not require that a CPU dependent IDLE thread body be provided as part of the port. If CPU_PROVIDES_IDLE_THREAD_BODY is defined to FALSE, then the CPU independent algorithm is used. This algorithm consists of a "branch to self" which is implemented in a routine as follows.

```
void *_Thread_Idle_body( uint32 ignored )
{
  while( 1 ) ;
}
```

If the CPU dependent IDLE thread body is implementation centers upon using a "halt", "idle", or "shutdown" instruction, then don't forget to put it in an infinite loop as the CPU will have to reexecute this instruction each time the IDLE thread is dispatched.

```
void *_CPU_Thread_Idle_body( uint32_t ignored )
{

  for( ; ; )
    /* insert your "halt" instruction here */ ;
}
```

Be warned. Some processors with onboard DMA have been known to stop the DMA if the CPU were put in IDLE mode. This might also be a problem with other on-chip peripherals. So use this hook with caution.

8 Priority Bitmap Manipulation

8.1 Introduction

The RTEMS chain of ready tasks is implemented as an array of FIFOs with each priority having its own FIFO. This makes it very efficient to determine the first and last ready task at each priority. In addition, blocking a task only requires appending the task to the end of the FIFO for its priority rather than a lengthy search down a single chain of all ready tasks. This works extremely well except for one problem. When the currently executing task blocks, there may be no easy way to determine what is the next most important ready task. If the blocking task was the only ready task at its priority, then RTEMS must search all of the FIFOs in the ready chain to determine the highest priority with a ready task.

RTEMS uses a bitmap array to efficiently solve this problem. The state of each bit in the priority map bit array indicates whether or not there is a ready task at that priority. The bit array can be efficiently searched to determine the highest priority ready task. This family of data type and routines is used to maintain and search the bit map array.

When manipulating the bitmap array, RTEMS internally divides the 8 bits of the task priority into "major" and "minor" components. The most significant 4 bits are the major component, while the least significant are the minor component. The major component of a priority value is used to determine which 16-bit wide entry in the `_Priority_Bit_map` array is associated with this priority. Each element in the `_Priority_Bit_map` array has a bit in the `_Priority_Major_bit_map` associated with it. That bit is cleared when all of the bits in a particular `_Priority_Bit_map` array entry are zero.

The minor component of a priority is used to determine specifically which bit in `_Priority_Bit_map[major]` indicates whether or not there is a ready to execute task at the priority.

8.2 _Priority_Bit_map_control Type

The `_Priority_Bit_map_Control` type is the fundamental data type of the priority bit map array used to determine which priorities have ready tasks. This type may be either 16 or 32 bits wide although only the 16 least significant bits will be used. The data type is based upon what is the most efficient type for this CPU to manipulate. For example, some CPUs have bit scan instructions that only operate on a particular size of data. In this case, this type will probably be defined to work with this instruction.

8.3 Find First Bit Routine

The _CPU_Bitfield_Find_first_bit routine sets _output to the bit number of the first bit set in `_value`. `_value` is of CPU dependent type `Priority_Bit_map_control`. A stub version of this routine is as follows:

```
#define _CPU_Bitfield_Find_first_bit( _value, _output ) \
  { \
    (_output) = 0;   /* do something to prevent warnings */ \
  }
```

There are a number of variables in using a "find first bit" type instruction.

1. What happens when run on a value of zero?
2. Bits may be numbered from MSB to LSB or vice-versa.
3. The numbering may be zero or one based.
4. The "find first bit" instruction may search from MSB or LSB.

RTEMS guarantees that (1) will never happen so it is not a concern. Cases (2),(3), (4) are handled by the macros _CPU_Priority_mask() and _CPU_Priority_bits_index(). These three form a set of routines which must logically operate together. Bits in the `_value` are set and cleared based on masks built by CPU_Priority_mask(). The basic major and minor values calculated by _Priority_Major() and _Priority_Minor() are "massaged" by _CPU_Priority_bits_index() to properly range between the values returned by the "find first bit" instruction. This makes it possible for _Priority_Get_highest() to calculate the major and directly index into the minor table. This mapping is necessary to ensure that 0 (a high priority major/minor) is the first bit found.

This entire "find first bit" and mapping process depends heavily on the manner in which a priority is broken into a major and minor components with the major being the 4 MSB of a priority and minor the 4 LSB. Thus (0 << 4) + 0 corresponds to priority 0 – the highest priority. And (15 << 4) + 14 corresponds to priority 254 – the next to the lowest priority.

If your CPU does not have a "find first bit" instruction, then there are ways to make do without it. Here are a handful of ways to implement this in software:

- a series of 16 bit test instructions
- a "binary search using if's"
- the following algorithm based upon a 16 entry lookup table. In this pseudo-code, bit_set_table[16] has values which indicate the first bit set:

```
_number = 0 if _value > 0x00ff
     _value >>=8
     _number = 8;
 if _value > 0x0000f
     _value >=8
     _number += 4

_number += bit_set_table[ _value ]
```

The following illustrates how the CPU_USE_GENERIC_BITFIELD_CODE macro may be so the port can use the generic implementation of this bitfield code. This can be used temporarily during the porting process to avoid writing these routines until the end. This results in a functional although lower performance port. This is perfectly acceptable during development and testing phases.

```
#define CPU_USE_GENERIC_BITFIELD_CODE TRUE
#define CPU_USE_GENERIC_BITFIELD_DATA TRUE
```

Eventually, CPU specific implementations of these routines are usually written since they dramatically impact the performance of blocking operations. However they may take advantage of instructions which are not available on all models in the CPU family. In this case, one might find something like this stub example did:

```
#if (CPU_USE_GENERIC_BITFIELD_CODE == FALSE)
#define _CPU_Bitfield_Find_first_bit( _value, _output ) \
  { \
    (_output) = 0;   /* do something to prevent warnings */ \
  }
#endif
```

8.4 Build Bit Field Mask

The _CPU_Priority_Mask routine builds the mask that corresponds to the bit fields searched by _CPU_Bitfield_Find_first_bit(). See the discussion of that routine for more details.

The following is a typical implementation when the _CPU_Bitfield_Find_first_bit searches for the most significant bit set:

```
#if (CPU_USE_GENERIC_BITFIELD_CODE == FALSE)
#define _CPU_Priority_Mask( _bit_number ) \
  ( 1 << (_bit_number) )
#endif
```

8.5 Bit Scan Support

The `_CPU_Priority_bits_index` routine translates the bit numbers returned by `_CPU_Bitfield_Find_first_bit()` into something suitable for use as a major or minor component of a priority. The find first bit routine may number the bits in a way that is difficult to map into the major and minor components of the priority. For example, when finding the first bit set in the value 0x8000, a CPU may indicate that bit 15 or 16 is set based on whether the least significant bit is "zero" or "one". Similarly, a CPU may only scan 32-bit values and consider the most significant bit to be bit zero or one. In this case, this would be bit 16 or 17.

This routine allows that unwieldy form to be converted into a normalized form that is easier to process and use as an index.

```
#if (CPU_USE_GENERIC_BITFIELD_CODE == FALSE)
#define _CPU_Priority_bits_index( _priority ) \
  (_priority)
#endif
```

9 Code Tuning Parameters

9.1 Inline Thread_Enable_dispatch

Should the calls to _Thread_Enable_dispatch be inlined?

If TRUE, then they are inlined.

If FALSE, then a subroutine call is made.

Basically this is an example of the classic trade-off of size versus speed. Inlining the call (TRUE) typically increases the size of RTEMS while speeding up the enabling of dispatching.

[NOTE: In general, the _Thread_Dispatch_disable_level will only be 0 or 1 unless you are in an interrupt handler and that interrupt handler invokes the executive.] When not inlined something calls _Thread_Enable_dispatch which in turns calls _Thread_Dispatch. If the enable dispatch is inlined, then one subroutine call is avoided entirely.]

```
#define CPU_INLINE_ENABLE_DISPATCH       FALSE
```

9.2 Inline Thread_queue_Enqueue_priority

Should the body of the search loops in _Thread_queue_Enqueue_priority be unrolled one time? In unrolled each iteration of the loop examines two "nodes" on the chain being searched. Otherwise, only one node is examined per iteration.

If TRUE, then the loops are unrolled.

If FALSE, then the loops are not unrolled.

The primary factor in making this decision is the cost of disabling and enabling interrupts (_ISR_Flash) versus the cost of rest of the body of the loop. On some CPUs, the flash is more expensive than one iteration of the loop body. In this case, it might be desirable to unroll the loop. It is important to note that on some CPUs, this code is the longest interrupt disable period in RTEMS. So it is necessary to strike a balance when setting this parameter.

```
#define CPU_UNROLL_ENQUEUE_PRIORITY      TRUE
```

9.3 Structure Alignment Optimization

The following macro may be defined to the attribute setting used to force alignment of critical RTEMS structures. On some processors it may make sense to have these aligned on tighter boundaries than the minimum requirements of the compiler in order to have as much of the critical data area as possible in a cache line. This ensures that the first access of an element in that structure fetches most, if not all, of the data structure and places it in the data cache. Modern CPUs often have cache lines of at least 16 bytes and thus a single access implicitly fetches some surrounding data and places that unreferenced data in the cache. Taking advantage of this allows RTEMS to essentially prefetch critical data elements.

The placement of this macro in the declaration of the variables is based on the syntactically requirements of the GNU C "__attribute__" extension. For another toolset, the placement of this macro could be incorrect. For example with GNU C, use the following definition of CPU_STRUCTURE_ALIGNMENT to force a structures to a 32 byte boundary.

#define CPU_STRUCTURE_ALIGNMENT __attribute__ ((aligned (32)))

To benefit from using this, the data must be heavily used so it will stay in the cache and used frequently enough in the executive to justify turning this on. NOTE: Because of this, only the Priority Bit Map table currently uses this feature.

The following illustrates how the CPU_STRUCTURE_ALIGNMENT is defined on ports which require no special alignment for optimized access to data structures:

```
#define CPU_STRUCTURE_ALIGNMENT
```

9.4 Data Alignment Requirements

9.4.1 Data Element Alignment

The CPU_ALIGNMENT macro should be set to the CPU's worst alignment requirement for data types on a byte boundary. This is typically the alignment requirement for a C double. This alignment does not take into account the requirements for the stack.

The following sets the CPU_ALIGNMENT macro to 8 which indicates that there is a basic C data type for this port which much be aligned to an 8 byte boundary.

```
#define CPU_ALIGNMENT              8
```

9.4.2 Heap Element Alignment

The CPU_HEAP_ALIGNMENT macro is set to indicate the byte alignment requirement for data allocated by the RTEMS Code Heap Handler. This alignment requirement may be stricter than that for the data types alignment specified by CPU_ALIGNMENT. It is common for the heap to follow the same alignment requirement as CPU_ALIGNMENT. If the CPU_ALIGNMENT is strict enough for the heap, then this should be set to CPU_ALIGNMENT. This macro is necessary to ensure that allocated memory is properly aligned for use by high level language routines.

The following example illustrates how the CPU_HEAP_ALIGNMENT macro is set when the required alignment for elements from the heap is the same as the basic CPU alignment requirements.

```
#define CPU_HEAP_ALIGNMENT         CPU_ALIGNMENT
```

NOTE: This does not have to be a power of 2. It does have to be greater or equal to than CPU_ALIGNMENT.

9.4.3 Partition Element Alignment

The CPU_PARTITION_ALIGNMENT macro is set to indicate the byte alignment requirement for memory buffers allocated by the RTEMS Partition Manager that is part of the Classic API. This alignment requirement may be stricter than that for the data types alignment specified by CPU_ALIGNMENT. It is common for the partition to follow the same alignment requirement as CPU_ALIGNMENT. If the CPU_ALIGNMENT is strict enough for the partition, then this should be set to CPU_ALIGNMENT. This macro is necessary to ensure that allocated memory is properly aligned for use by high level language routines.

The following example illustrates how the CPU_PARTITION_ALIGNMENT macro is set when the required alignment for elements from the RTEMS Partition Manager is the same as the basic CPU alignment requirements.

```
#define CPU_PARTITION_ALIGNMENT    CPU_ALIGNMENT
```

NOTE: This does not have to be a power of 2. It does have to be greater or equal to than CPU_ALIGNMENT.

10 Miscellaneous

10.1 Fatal Error Default Handler

The `_CPU_Fatal_halt` routine is the default fatal error handler. This routine copies _error into a known place – typically a stack location or a register, optionally disables interrupts, and halts/stops the CPU. It is prototyped as follows and is often implemented as a macro:

```
void _CPU_Fatal_halt(
    unsigned32 _error
);
```

10.2 Processor Endianness

Endianness refers to the order in which numeric values are stored in memory by the microprocessor. Big endian architectures store the most significant byte of a multi-byte numeric value in the byte with the lowest address. This results in the hexadecimal value 0x12345678 being stored as 0x12345678 with 0x12 in the byte at offset zero, 0x34 in the byte at offset one, etc.. The Motorola M68K and numerous RISC processor families is big endian. Conversely, little endian architectures store the least significant byte of a multi-byte numeric value in the byte with the lowest address. This results in the hexadecimal value 0x12345678 being stored as 0x78563412 with 0x78 in the byte at offset zero, 0x56 in the byte at offset one, etc.. The Intel ix86 family is little endian. Interestingly, some CPU models within the PowerPC and MIPS architectures can be switched between big and little endian modes. Most embedded systems use these families strictly in big endian mode.

RTEMS must be informed of the byte ordering for this microprocessor family and, optionally, endian conversion routines may be provided as part of the port. Conversion between endian formats is often necessary in multiprocessor environments and sometimes needed when interfacing with peripheral controllers.

10.2.1 Specifying Processor Endianness

The `CPU_BIG_ENDIAN` and `CPU_LITTLE_ENDIAN` are set to specify the endian format used by this microprocessor. These macros should not be set to the same value. The following example illustrates how these macros should be set on a processor family that is big endian.

```
#define CPU_BIG_ENDIAN                                    TRUE
#define CPU_LITTLE_ENDIAN                                 FALSE
```

The `CPU_MPCI_RECEIVE_SERVER_EXTRA_STACK` macro is set to the amount of stack space above the minimum thread stack space required by the MPCI Receive Server Thread. This macro is needed because in a multiprocessor system the MPCI Receive Server Thread must be able to process all directives.

```
#define CPU_MPCI_RECEIVE_SERVER_EXTRA_STACK 0
```

10.2.2 Endian Swap Unsigned Integers

The port should provide routines to swap sixteen (`CPU_swap_u16`) and thirty-bit (`CPU_swap_u32`) unsigned integers. These are primarily used in two areas of RTEMS - multiprocessing support and the network endian swap routines. The `CPU_swap_u32` routine must be implemented as a static routine rather than a macro because its address is taken and used indirectly. On the other hand, the `CPU_swap_u16` routine may be implemented as a macro.

Some CPUs have special instructions that swap a 32-bit quantity in a single instruction (e.g. i486). It is probably best to avoid an "endian swapping control bit" in the CPU. One good reason is that interrupts would probably have to be disabled to insure that an interrupt does not try to access the same "chunk" with the wrong endian. Another good reason is that on some CPUs, the endian bit endianness for ALL fetches – both code and data – so the code will be fetched incorrectly.

The following is an implementation of the `CPU_swap_u32` routine that will work on any CPU. It operates by breaking the unsigned thirty-two bit integer into four byte-wide quantities and reassemblying them.

```
static inline unsigned int CPU_swap_u32(
  unsigned int value
)
{
  unsigned32 byte1, byte2, byte3, byte4, swapped;

  byte4 = (value >> 24) & 0xff;
  byte3 = (value >> 16) & 0xff;
  byte2 = (value >> 8)  & 0xff;
  byte1 =  value        & 0xff;

  swapped = (byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4;
  return( swapped );
}
```

Although the above implementation is portable, it is not particularly efficient. So if there is a better way to implement this on a particular CPU family or model, please do so. The efficiency of this routine has significant impact on the efficiency of the multiprocessing support code in the shared memory driver and in network applications using the ntohl() family of routines.

Most microprocessor families have rotate instructions which can be used to greatly improve the `CPU_swap_u32` routine. The most common way to do this is to:

```
swap least significant two bytes with 16-bit rotate
swap upper and lower 16-bits
swap most significant two bytes with 16-bit rotate
```

Some CPUs have special instructions that swap a 32-bit quantity in a single instruction (e.g. i486). It is probably best to avoid an "endian swapping control bit" in the CPU. One good reason is that interrupts would probably have to be disabled to insure that an interrupt does not try to access the same "chunk" with the wrong endian. Another good reason is that on some CPUs, the endian bit endianness for ALL fetches – both code and data – so the code will be fetched incorrectly.

Similarly, here is a portable implementation of the `CPU_swap_u16` routine. Just as with the `CPU_swap_u32` routine, the porter should provide a better implementation if possible.

```
#define CPU_swap_u16( value ) \
  (((value&0xff) << 8) | ((value >> 8)&0xff))
```

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.