# RTEMS CPU Architecture Supplement

Edition 4.7.1, for RTEMS 4.7.1

13 November 2006

**On-Line Applications Research Corporation**

On-Line Applications Research Corporation
TEXinfo 2006-10-04.17

COPYRIGHT © 1988 - 2006.
On-Line Applications Research Corporation (OAR).

The authors have used their best efforts in preparing this material. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. No warranty of any kind, expressed or implied, with regard to the software or the material contained in this document is provided. No liability arising out of the application or use of any product described in this document is assumed. The authors reserve the right to revise this material and to make changes from time to time in the content hereof without obligation to notify anyone of such revision or changes.

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/support.html`.

# Table of Contents

**Preface** ..... 1

**1 ARM Specific Information** ..... 3

1.1 CPU Model Dependent Features ..... 3
1.1.1 CPU Model Name ..... 4
1.1.2 Count Leading Zeroes Instruction ..... 4
1.1.3 Floating Point Unit ..... 4
1.2 Calling Conventions ..... 4
1.2.1 Processor Background ..... 4
1.2.2 Calling Mechanism ..... 5
1.2.3 Register Usage ..... 5
1.2.4 Parameter Passing ..... 5
1.2.5 User-Provided Routines ..... 5
1.3 Memory Model ..... 5
1.3.1 Flat Memory Model ..... 5
1.4 Interrupt Processing ..... 6
1.4.1 Vectoring of an Interrupt Handler ..... 6
1.4.2 Interrupt Levels ..... 6
1.4.3 Disabling of Interrupts by RTEMS ..... 7
1.4.4 Interrupt Stack ..... 7
1.5 Default Fatal Error Processing ..... 7
1.5.1 Default Fatal Error Handler Operations ..... 7
1.6 Board Support Packages ..... 8
1.6.1 System Reset ..... 8
1.6.2 Processor Initialization ..... 8
1.7 Processor Dependent Information Table ..... 9
1.7.1 CPU Dependent Information Table ..... 9

**2 Intel/AMD x86 Specific Information** ..... 11

2.1 CPU Model Dependent Features ..... 11
2.1.1 CPU Model Name ..... 12
2.1.2 bswap Instruction ..... 12
2.1.3 Floating Point Unit ..... 12
2.2 Calling Conventions ..... 12
2.2.1 Processor Background ..... 12
2.2.2 Calling Mechanism ..... 12
2.2.3 Register Usage ..... 12
2.2.4 Parameter Passing ..... 13
2.2.5 User-Provided Routines ..... 13
2.3 Memory Model ..... 13
2.3.1 Flat Memory Model ..... 13
2.4 Interrupt Processing ..... 14

2.4.1 Vectoring of Interrupt Handler ..... 14
2.4.2 Interrupt Stack Frame ..... 15
2.4.3 Interrupt Levels ..... 15
2.4.4 Disabling of Interrupts by RTEMS ..... 15
2.4.5 Interrupt Stack ..... 15
2.5 Default Fatal Error Processing ..... 16
2.5.1 Default Fatal Error Handler Operations ..... 16
2.6 Board Support Packages ..... 16
2.6.1 System Reset ..... 16
2.6.2 Processor Initialization ..... 17
2.7 Processor Dependent Information Table ..... 18
2.7.1 CPU Dependent Information Table ..... 18

**3 Motorola M68xxx and Coldfire Specific Information ..... 21**

3.1 CPU Model Dependent Features ..... 21
3.1.1 CPU Model Name ..... 22
3.1.2 Floating Point Unit ..... 22
3.1.3 BFFFO Instruction ..... 22
3.1.4 Vector Base Register ..... 22
3.1.5 Separate Stacks ..... 22
3.1.6 Pre-Indexing Address Mode ..... 22
3.1.7 Extend Byte to Long Instruction ..... 22
3.2 Calling Conventions ..... 23
3.2.1 Processor Background ..... 23
3.2.2 Calling Mechanism ..... 23
3.2.3 Register Usage ..... 23
3.2.4 Parameter Passing ..... 23
3.2.5 User-Provided Routines ..... 24
3.3 Memory Model ..... 24
3.3.1 Flat Memory Model ..... 24
3.4 Interrupt Processing ..... 24
3.4.1 Vectoring of an Interrupt Handler ..... 24
3.4.1.1 Models Without Separate Interrupt Stacks ..... 24
3.4.1.2 Models With Separate Interrupt Stacks ..... 25
3.4.2 CPU Models Without VBR and RAM at 0 ..... 25
3.4.3 Interrupt Levels ..... 27
3.4.4 Disabling of Interrupts by RTEMS ..... 27
3.4.5 Interrupt Stack ..... 27
3.5 Default Fatal Error Processing ..... 27
3.5.1 Default Fatal Error Handler Operations ..... 28
3.6 Board Support Packages ..... 28
3.6.1 System Reset ..... 28
3.6.2 Processor Initialization ..... 28
3.7 Processor Dependent Information Table ..... 29
3.7.1 CPU Dependent Information Table ..... 29

# 4 MIPS Specific Information ..... 31

4.1 CPU Model Dependent Features ..... 31
4.1.1 CPU Model Name ..... 32
4.1.2 Floating Point Unit ..... 32
4.1.3 Another Optional Feature ..... 32
4.2 Calling Conventions ..... 32
4.2.1 Processor Background ..... 32
4.2.2 Calling Mechanism ..... 33
4.2.3 Register Usage ..... 33
4.2.4 Parameter Passing ..... 33
4.2.5 User-Provided Routines ..... 33
4.3 Memory Model ..... 33
4.3.1 Flat Memory Model ..... 33
4.4 Interrupt Processing ..... 34
4.4.1 Vectoring of an Interrupt Handler ..... 34
4.4.1.1 Models Without Separate Interrupt Stacks ..... 34
4.4.1.2 Models With Separate Interrupt Stacks ..... 34
4.4.2 Interrupt Levels ..... 35
4.4.3 Disabling of Interrupts by RTEMS ..... 35
4.4.4 Interrupt Stack ..... 35
4.5 Default Fatal Error Processing ..... 36
4.5.1 Default Fatal Error Handler Operations ..... 36
4.6 Board Support Packages ..... 36
4.6.1 System Reset ..... 36
4.6.2 Processor Initialization ..... 37
4.7 Processor Dependent Information Table ..... 37
4.7.1 CPU Dependent Information Table ..... 37

# 5 PowerPC Specific Information ..... 41

5.1 CPU Model Dependent Features ..... 42
5.1.1 CPU Model Feature Flags ..... 42
5.1.1.1 CPU Model Name ..... 42
5.1.1.2 Floating Point Unit ..... 42
5.1.1.3 Alignment ..... 43
5.1.1.4 Cache Alignment ..... 43
5.1.1.5 Maximum Interrupts ..... 43
5.1.1.6 Has Double Precision Floating Point ..... 43
5.1.1.7 Critical Interrupts ..... 43
5.1.1.8 Use Multiword Load/Store Instructions ..... 43
5.1.1.9 Instruction Cache Size ..... 43
5.1.1.10 Data Cache Size ..... 43
5.1.1.11 Debug Model ..... 43
5.1.1.12 Low Power Model ..... 44
5.2 Calling Conventions ..... 44
5.2.1 Programming Model ..... 44
5.2.1.1 Non-Floating Point Registers ..... 44
5.2.1.2 Floating Point Registers ..... 45
5.2.1.3 Special Registers ..... 45

5.2.2 Call and Return Mechanism ... 45
5.2.3 Calling Mechanism ... 46
5.2.4 Register Usage ... 46
5.2.5 Parameter Passing ... 46
5.2.6 User-Provided Routines ... 46
5.3 Memory Model ... 46
5.3.1 Flat Memory Model ... 46
5.4 Interrupt Processing ... 47
5.4.1 Synchronous Versus Asynchronous Exceptions ... 47
5.4.2 Vectoring of Interrupt Handler ... 48
5.4.3 Interrupt Levels ... 48
5.4.4 Disabling of Interrupts by RTEMS ... 49
5.4.5 Interrupt Stack ... 49
5.5 Default Fatal Error Processing ... 50
5.5.1 Default Fatal Error Handler Operations ... 50
5.6 Board Support Packages ... 50
5.6.1 System Reset ... 50
5.6.2 Processor Initialization ... 50
5.7 Processor Dependent Information Table ... 51
5.7.1 CPU Dependent Information Table ... 51

**6 SuperH Specific Information ... 55**

6.1 CPU Model Dependent Features ... 55
6.1.1 CPU Model Name ... 56
6.1.2 Floating Point Unit ... 56
6.1.3 Another Optional Feature ... 56
6.2 Calling Conventions ... 56
6.2.1 Calling Mechanism ... 57
6.2.2 Register Usage ... 57
6.2.3 Parameter Passing ... 57
6.2.4 User-Provided Routines ... 57
6.3 Memory Model ... 57
6.3.1 Flat Memory Model ... 58
6.4 Interrupt Processing ... 58
6.4.1 Vectoring of an Interrupt Handler ... 58
6.4.1.1 Models Without Separate Interrupt Stacks ... 58
6.4.1.2 Models With Separate Interrupt Stacks ... 58
6.4.2 Interrupt Levels ... 59
6.4.3 Disabling of Interrupts by RTEMS ... 59
6.4.4 Interrupt Stack ... 60
6.5 Default Fatal Error Processing ... 60
6.5.1 Default Fatal Error Handler Operations ... 60
6.6 Board Support Packages ... 60
6.6.1 System Reset ... 60
6.6.2 Processor Initialization ... 61
6.7 Processor Dependent Information Table ... 61
6.7.1 CPU Dependent Information Table ... 61

**7 SPARC Specific Information ..... 65**

7.1 CPU Model Dependent Features ..... 66
7.1.1 CPU Model Feature Flags ..... 66
7.1.1.1 CPU Model Name ..... 66
7.1.1.2 Floating Point Unit ..... 66
7.1.1.3 Bitscan Instruction ..... 67
7.1.1.4 Number of Register Windows ..... 67
7.1.1.5 Low Power Mode ..... 67
7.1.2 CPU Model Implementation Notes ..... 67
7.2 Calling Conventions ..... 68
7.2.1 Programming Model ..... 68
7.2.1.1 Non-Floating Point Registers ..... 68
7.2.1.2 Floating Point Registers ..... 69
7.2.1.3 Special Registers ..... 69
7.2.2 Register Windows ..... 69
7.2.3 Call and Return Mechanism ..... 71
7.2.4 Calling Mechanism ..... 71
7.2.5 Register Usage ..... 71
7.2.6 Parameter Passing ..... 71
7.2.7 User-Provided Routines ..... 71
7.3 Memory Model ..... 71
7.3.1 Flat Memory Model ..... 72
7.4 Interrupt Processing ..... 72
7.4.1 Synchronous Versus Asynchronous Traps ..... 73
7.4.2 Vectoring of Interrupt Handler ..... 73
7.4.3 Traps and Register Windows ..... 74
7.4.4 Interrupt Levels ..... 74
7.4.5 Disabling of Interrupts by RTEMS ..... 74
7.4.6 Interrupt Stack ..... 75
7.5 Default Fatal Error Processing ..... 75
7.5.1 Default Fatal Error Handler Operations ..... 75
7.6 Board Support Packages ..... 75
7.6.1 System Reset ..... 75
7.6.2 Processor Initialization ..... 76
7.7 Processor Dependent Information Table ..... 76
7.7.1 CPU Dependent Information Table ..... 76

**Command and Variable Index ..... 79**

**Concept Index ..... 81**

# Preface

The Real Time Executive for Multiprocessor Systems (RTEMS) is designed to be portable across multiple processor architectures. However, the nature of real-time systems makes it essential that the application designer understand certain processor dependent implementation details. These processor dependencies include calling convention, board support package issues, interrupt processing, exact RTEMS memory requirements, performance data, header files, and the assembly language interface to the executive.

Each chapter in this document discusses the details of how RTEMS was ported.

# 1 ARM Specific Information

The Real Time Executive for Multiprocessor Systems (RTEMS) is designed to be portable across multiple processor architectures. However, the nature of real-time systems makes it essential that the application designer understand certain processor dependent implementation details. These processor dependencies include calling convention, board support package issues, interrupt processing, exact RTEMS memory requirements, performance data, header files, and the assembly language interface to the executive.

This document discusses the ARM architecture dependencies in this port of RTEMS. The ARM family has a wide variety of implementations by a wide range of vendors. Consequently, there are 100's of CPU models within it.

It is highly recommended that the ARM RTEMS application developer obtain and become familiar with the documentation for the processor being used as well as the documentation for the ARM architecture as a whole.

### Architecture Documents

For information on the ARM architecture, refer to the following documents available from Arm, Limited ('`http//www.arm.com/`'). There does not appear to be an electronic version of a manual on the architecture in general on that site. The following book is a good resource:

- *David Seal.* "*ARM Architecture Reference Manual.*" *Addison-Wesley.* **ISBN 0-201-73719-1**. *2001.*

## 1.1 CPU Model Dependent Features

Microprocessors are generally classified into families with a variety of CPU models or implementations within that family. Within a processor family, there is a high level of binary compatibility. This family may be based on either an architectural specification or on maintaining compatibility with a popular processor. Recent microprocessor families such as the ARM, SPARC, and PowerPC are based on an architectural specification which is independent or any particular CPU model or implementation. Older families such as the M68xxx and the iX86 evolved as the manufacturer strived to produce higher performance processor models which maintained binary compatibility with older models.

RTEMS takes advantage of the similarity of the various models within a CPU family. Although the models do vary in significant ways, the high level of compatibility makes it possible to share the bulk of the CPU dependent executive code across the entire family. Each processor family supported by RTEMS has a list of features which vary between CPU models within a family. For example, the most common model dependent feature regardless of CPU family is the presence or absence of a floating point unit or coprocessor. When defining the list of features present on a particular CPU model, one simply notes that floating point hardware is or is not present and defines a single constant appropriately. Conditional compilation is utilized to include the appropriate source code for this CPU model's feature set. It is important to note that this means that RTEMS is thus compiled using the appropriate feature set and compilation flags optimal for this CPU model used.

The alternative would be to generate a binary which would execute on all family members using only the features which were always present.

This chapter presents the set of features which vary across ARM implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file cpukit/score/cpu/arm/rtems/score/arm.h based upon the particular CPU model defined on the compilation command line.

### 1.1.1 CPU Model Name

The macro `CPU_MODEL_NAME` is a string which designates the architectural level of this CPU model. The following is a list of the settings for this string based upon `gcc` CPU model predefines:

```
__ARM_ARCH4__   "ARMv4"
__ARM_ARCH4T__  "ARMv4T"
__ARM_ARCH5__   "ARMv5"
__ARM_ARCH5T__  "ARMv5T"
__ARM_ARCH5E__  "ARMv5E"
__ARM_ARCH5TE__ "ARMv5TE"
```

### 1.1.2 Count Leading Zeroes Instruction

The ARMv5 and later has the count leading zeroes (`clz`) instruction which could be used to speed up the find first bit operation. The use of this instruction should significantly speed up the scheduling associated with a thread blocking.

### 1.1.3 Floating Point Unit

The macro ARM_HAS_FPU is set to 1 to indicate that this CPU model has a hardware floating point unit and 0 otherwise. It does not matter whether the hardware floating point support is incorporated on-chip or is an external coprocessor.

## 1.2 Calling Conventions

Each high-level language compiler generates subroutine entry and exit code based upon a set of rules known as the compiler's calling convention. These rules address the following issues:

- register preservation and usage
- parameter passing
- call and return mechanism

A compiler's calling convention is of importance when interfacing to subroutines written in another language either assembly or high-level. Even when the high-level language and target processor are the same, different compilers may use different calling conventions. As a result, calling conventions are both processor and compiler dependent.

### 1.2.1 Processor Background

The ARM architecture supports a simple yet effective call and return mechanism. A subroutine is invoked via the branch and link (`bl`) instruction. This instruction saves the

return address in the `lr` register. Returning from a subroutine only requires that the return address be moved into the program counter (`pc`), possibly with an offset. It is is important to note that the `bl` instruction does not automatically save or restore any registers. It is the responsibility of the high-level language compiler to define the register preservation and usage convention.

### 1.2.2 Calling Mechanism

All RTEMS directives are invoked using the `bl` instruction and return to the user application via the mechanism described above.

### 1.2.3 Register Usage

As discussed above, the ARM's call and return mechanism dos not automatically save any registers. RTEMS uses the registers `r0`, `r1`, `r2`, and `r3` as scratch registers and per ARM calling convention, the `lr` register is altered as well. These registers are not preserved by RTEMS directives therefore, the contents of these registers should not be assumed upon return from any RTEMS directive.

### 1.2.4 Parameter Passing

RTEMS assumes that ARM calling conventions are followed and that the first four arguments are placed in registers `r0` through `r3`. If there are more arguments, than that, then they are place on the stack.

### 1.2.5 User-Provided Routines

All user-provided routines invoked by RTEMS, such as user extensions, device drivers, and MPCI routines, must also adhere to these calling conventions.

## 1.3 Memory Model

A processor may support any combination of memory models ranging from pure physical addressing to complex demand paged virtual memory systems. RTEMS supports a flat memory model which ranges contiguously over the processor's allowable address space. RTEMS does not support segmentation or virtual memory of any kind. The appropriate memory model for RTEMS provided by the targeted processor and related characteristics of that model are described in this chapter.

### 1.3.1 Flat Memory Model

Members of the ARM family newer than Version 3 support a flat 32-bit address space with addresses ranging from 0x00000000 to 0xFFFFFFFF (4 gigabytes). Each address is represented by a 32-bit value and is byte addressable. The address may be used to reference a single byte, word (2-bytes), or long word (4 bytes). Memory accesses within this address space are performed in the endian mode that the processor is configured for. In general, ARM processors are used in little endian mode.

Some of the ARM family members such as the 920 and 720 include an MMU and thus support virtual memory and segmentation. RTEMS does not support virtual memory or segmentation on any of the ARM family members.

## 1.4 Interrupt Processing

Different types of processors respond to the occurrence of an interrupt in its own unique fashion. In addition, each processor type provides a control mechanism to allow for the proper handling of an interrupt. The processor dependent response to the interrupt modifies the current execution state and results in a change in the execution stream. Most processors require that an interrupt handler utilize some special control mechanisms to return to the normal processing stream. Although RTEMS hides many of the processor dependent details of interrupt processing, it is important to understand how the RTEMS interrupt manager is mapped onto the processor's unique architecture. Discussed in this chapter are the ARM's interrupt response and control mechanisms as they pertain to RTEMS.

The ARM has 7 exception types:

- Reset
- Undefined instruction
- Software interrupt (SWI)
- Prefetch Abort
- Data Abort
- Interrupt (IRQ)
- Fast Interrupt (FIQ)

Of these types, only IRQ and FIQ are handled through RTEMS's interrupt vectoring.

### 1.4.1 Vectoring of an Interrupt Handler

Unlike many other architectures, the ARM has seperate stacks for each interrupt. When the CPU receives an interrupt, it:

- switches to the exception mode corresponding to the interrupt,
- saves the Current Processor Status Register (CPSR) to the exception mode's Saved Processor Status Register (SPSR),
- masks off the IRQ and if the interrupt source was FIQ, the FIQ is masked off as well,
- saves the Program Counter (PC) to the exception mode's Link Register (LR - same as R14),
- and sets the PC to the exception's vector address.

The vectors for both IRQ and FIQ point to the _ISR_Handler function. _ISR_Handler() calls the BSP specific handler, ExecuteITHandler(). Before calling ExecuteITHandler(), registers R0-R3, R12, and R14(LR) are saved so that it is safe to call C functions. Even ExecuteITHandler() can be written in C.

### 1.4.2 Interrupt Levels

The ARM architecture supports two external interrupts - IRQ and FIQ. FIQ has a higher priority than IRQ, and has its own version of register R8 - R14, however RTEMS does not take advantage of them. Both interrupts are enabled through the CPSR.

The RTEMS interrupt level mapping scheme for the AEM is not a numeric level as on most RTEMS ports. It is a bit mapping that corresponds the enable bits's postions in the CPSR:

**FIQ** Setting bit 6 (0 is least significant bit) disables the FIQ.

**IRQ** Setting bit 7 (0 is least significant bit) disables the IRQ.

### 1.4.3 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables interrupts to level seven (7) before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than RTEMS_MAXIMUM_DISABLE_PERIOD microseconds on a RTEMS_MAXIMUM_DISABLE_PERIOD_MHZ Mhz processor with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled is hand calculated. This calculation was last performed for Release RTEMS_RELEASE_FOR_MAXIMUM_DISABLE_PERIOD.]

Non-maskable interrupts (NMI) cannot be disabled, and ISRs which execute at this level MUST NEVER issue RTEMS system calls. If a directive is invoked, unpredictable results may occur due to the inability of RTEMS to protect its critical sections. However, ISRs that make no system calls may safely execute as non-maskable interrupts.

### 1.4.4 Interrupt Stack

RTEMS expects the interrupt stacks to be set up in bsp_start(). The memory for the stacks is reserved in the linker script.

## 1.5 Default Fatal Error Processing

Upon detection of a fatal error by either the application or RTEMS the fatal error manager is invoked. The fatal error manager will invoke the user-supplied fatal error handlers. If no user-supplied handlers are configured, the RTEMS provided default fatal error handler is invoked. If the user-supplied fatal error handlers return to the executive the default fatal error handler is then invoked. This chapter describes the precise operations of the default fatal error handler.

### 1.5.1 Default Fatal Error Handler Operations

The default fatal error handler which is invoked by the `rtems_fatal_error_occurred` directive when there is no user handler configured or the user handler returns control to RTEMS. The default fatal error handler performs the following actions:

- disables processor interrupts,
- places the error code in **r0**, and
- executes an infinite loop (`while(0);` to simulate a halt processor instruction.

## 1.6 Board Support Packages

An RTEMS Board Support Package (BSP) must be designed to support a particular processor and target board combination. This chapter presents a discussion of XXX specific BSP issues. For more information on developing a BSP, refer to the chapter titled Board Support Packages in the RTEMS Applications User's Guide.

### 1.6.1 System Reset

An RTEMS based application is initiated or re-initiated when the XXX processor is reset. When the XXX is reset, the processor performs the following actions:

- The tracing bits of the status register are cleared to disable tracing.
- The supervisor interrupt state is entered by setting the supervisor (S) bit and clearing the master/interrupt (M) bit of the status register.
- The interrupt mask of the status register is set to level 7 to effectively disable all maskable interrupts.
- The vector base register (VBR) is set to zero.
- The cache control register (CACR) is set to zero to disable and freeze the processor cache.
- The interrupt stack pointer (ISP) is set to the value stored at vector 0 (bytes 0-3) of the exception vector table (EVT).
- The program counter (PC) is set to the value stored at vector 1 (bytes 4-7) of the EVT.
- The processor begins execution at the address stored in the PC.

### 1.6.2 Processor Initialization

The address of the application's initialization code should be stored in the first vector of the EVT which will allow the immediate vectoring to the application code. If the application requires that the VBR be some value besides zero, then it should be set to the required value at this point. All tasks share the same XXX's VBR value. Because interrupts are enabled automatically by RTEMS as part of the initialize executive directive, the VBR MUST be set before this directive is invoked to insure correct interrupt vectoring. If processor caching is to be utilized, then it should be enabled during the reset application initialization code.

In addition to the requirements described in the Board Support Packages chapter of the Applications User's Manual for the reset code which is executed before the call to initialize executive, the XXX version has the following specific requirements:

- Must leave the S bit of the status register set so that the XXX remains in the supervisor state.
- Must set the M bit of the status register to remove the XXX from the interrupt state.
- Must set the master stack pointer (MSP) such that a minimum stack size of MINIMUM_STACK_SIZE bytes is provided for the initialize executive directive.
- Must initialize the XXX's vector table.

Note that the BSP is not responsible for allocating or installing the interrupt stack. RTEMS does this automatically as part of initialization. If the BSP does not install an interrupt stack and – for whatever reason – an interrupt occurs before initialize_executive is invoked, then the results are unpredictable.

## 1.7 Processor Dependent Information Table

Any highly processor dependent information required to describe a processor to RTEMS is provided in the CPU Dependent Information Table. This table is not required for all processors supported by RTEMS. This chapter describes the contents, if any, for a particular processor type.

### 1.7.1 CPU Dependent Information Table

The XXX version of the RTEMS CPU Dependent Information Table contains the information required to interface a Board Support Package and RTEMS on the XXX. This information is provided to allow RTEMS to interoperate effectively with the BSP. The C structure definition is given here:

```
typedef struct {
  void       (*pretasking_hook)( void );
  void       (*predriver_hook)( void );
  void       (*postdriver_hook)( void );
  void       (*idle_task)( void );
  boolean      do_zero_of_workspace;
  unsigned32   idle_task_stack_size;
  unsigned32   interrupt_stack_size;
  unsigned32   extra_mpci_receive_server_stack;
  void *     (*stack_allocate_hook)( unsigned32 );
  void       (*stack_free_hook)( void* );
  /* end of fields required on all CPUs */

  /* XXX CPU family dependent stuff */
} rtems_cpu_table;
```

`pretasking_hook` is the address of the user provided routine which is invoked once RTEMS APIs are initialized. This routine will be invoked before any system tasks are created. Interrupts are disabled. This field may be NULL to indicate that the hook is not utilized.

`predriver_hook` is the address of the user provided routine that is invoked immediately before the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`postdriver_hook` is the address of the user provided routine that is invoked immediately after the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`idle_task` is the address of the optional user provided routine which is used as the system's IDLE task. If this field is not NULL, then the RTEMS default IDLE task is not used. This field may be NULL to indicate that the default IDLE is to be used.

`do_zero_of_workspace`
indicates whether RTEMS should zero the Workspace as part of its initialization. If set to TRUE, the Workspace is zeroed. Otherwise, it is not.

`idle_task_stack_size`
is the size of the RTEMS idle task stack in bytes. If this number is less than MINIMUM_STACK_SIZE, then the idle task's stack will be MINIMUM_STACK_SIZE in byte.

`interrupt_stack_size`
is the size of the RTEMS allocated interrupt stack in bytes. This value must be at least as large as MINIMUM_STACK_SIZE.

`extra_mpci_receive_server_stack`
is the extra stack space allocated for the RTEMS MPCI receive server task in bytes. The MPCI receive server may invoke nearly all directives and may require extra stack space on some targets.

`stack_allocate_hook`
is the address of the optional user provided routine which allocates memory for task stacks. If this hook is not NULL, then a stack_free_hook must be provided as well.

`stack_free_hook` is the address of the optional user provided routine which frees memory for task stacks. If this hook is not NULL, then a stack_allocate_hook must be provided as well.

`XXX` is where the CPU family dependent stuff goes.

# 2 Intel/AMD x86 Specific Information

The Real Time Executive for Multiprocessor Systems (RTEMS) is designed to be portable across multiple processor architectures. However, the nature of real-time systems makes it essential that the application designer understand certain processor dependent implementation details. These processor dependencies include calling convention, board support package issues, interrupt processing, exact RTEMS memory requirements, performance data, header files, and the assembly language interface to the executive.

For information on the i386 processor, refer to the following documents:

- *386 Programmer's Reference Manual, Intel, Order No. 230985-002.*
- *386 Microprocessor Hardware Reference Manual, Intel, Order No. 231732-003.*
- *80386 System Software Writer's Guide, Intel, Order No. 231499-001.*
- *80387 Programmer's Reference Manual, Intel, Order No. 231917-001.*

It is highly recommended that the i386 RTEMS application developer obtain and become familiar with Intel's 386 Programmer's Reference Manual.

## 2.1 CPU Model Dependent Features

Microprocessors are generally classified into families with a variety of CPU models or implementations within that family. Within a processor family, there is a high level of binary compatibility. This family may be based on either an architectural specification or on maintaining compatibility with a popular processor. Recent microprocessor families such as the SPARC or PowerPC are based on an architectural specification which is independent or any particular CPU model or implementation. Older families such as the M68xxx and the iX86 evolved as the manufacturer strived to produce higher performance processor models which maintained binary compatibility with older models.

RTEMS takes advantage of the similarity of the various models within a CPU family. Although the models do vary in significant ways, the high level of compatibility makes it possible to share the bulk of the CPU dependent executive code across the entire family. Each processor family supported by RTEMS has a list of features which vary between CPU models within a family. For example, the most common model dependent feature regardless of CPU family is the presence or absence of a floating point unit or coprocessor. When defining the list of features present on a particular CPU model, one simply notes that floating point hardware is or is not present and defines a single constant appropriately. Conditional compilation is utilized to include the appropriate source code for this CPU model's feature set. It is important to note that this means that RTEMS is thus compiled using the appropriate feature set and compilation flags optimal for this CPU model used. The alternative would be to generate a binary which would execute on all family members using only the features which were always present.

This chapter presents the set of features which vary across i386 implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file cpukit/score/cpu/i386/i386.h based upon the particular CPU model defined on the compilation command line.

### 2.1.1 CPU Model Name

The macro CPU_MODEL_NAME is a string which designates the name of this CPU model. For example, for the Intel i386 without an i387 coprocessor, this macro is set to the string "i386 with i387".

### 2.1.2 bswap Instruction

The macro I386_HAS_BSWAP is set to 1 to indicate that this CPU model has the `bswap` instruction which endian swaps a thirty-two bit quantity. This instruction appears to be present in all CPU models i486's and above.

### 2.1.3 Floating Point Unit

The macro I386_HAS_FPU is set to 1 to indicate that this CPU model has a hardware floating point unit and 0 otherwise. The hardware floating point may be on-chip (as in the case of an i486DX or Pentium) or as a coprocessor (as in the case of an i386/i387 combination).

## 2.2 Calling Conventions

Each high-level language compiler generates subroutine entry and exit code based upon a set of rules known as the compiler's calling convention. These rules address the following issues:

- register preservation and usage
- parameter passing
- call and return mechanism

A compiler's calling convention is of importance when interfacing to subroutines written in another language either assembly or high-level. Even when the high-level language and target processor are the same, different compilers may use different calling conventions. As a result, calling conventions are both processor and compiler dependent.

### 2.2.1 Processor Background

The i386 architecture supports a simple yet effective call and return mechanism. A subroutine is invoked via the call (call) instruction. This instruction pushes the return address on the stack. The return from subroutine (ret) instruction pops the return address off the current stack and transfers control to that instruction. It is is important to note that the i386 call and return mechanism does not automatically save or restore any registers. It is the responsibility of the high-level language compiler to define the register preservation and usage convention.

### 2.2.2 Calling Mechanism

All RTEMS directives are invoked using a call instruction and return to the user application via the ret instruction.

### 2.2.3 Register Usage

As discussed above, the call instruction does not automatically save any registers. RTEMS uses the registers EAX, ECX, and EDX as scratch registers. These registers are not pre-

served by RTEMS directives therefore, the contents of these registers should not be assumed upon return from any RTEMS directive.

### 2.2.4 Parameter Passing

RTEMS assumes that arguments are placed on the current stack before the directive is invoked via the call instruction. The first argument is assumed to be closest to the return address on the stack. This means that the first argument of the C calling sequence is pushed last. The following pseudo-code illustrates the typical sequence used to call a RTEMS directive with three (3) arguments:

```
push third argument
push second argument
push first argument
invoke directive
remove arguments from the stack
```

The arguments to RTEMS are typically pushed onto the stack using a push instruction. These arguments must be removed from the stack after control is returned to the caller. This removal is typically accomplished by adding the size of the argument list in bytes to the stack pointer.

### 2.2.5 User-Provided Routines

All user-provided routines invoked by RTEMS, such as user extensions, device drivers, and MPCI routines, must also adhere to these calling conventions.

## 2.3 Memory Model

A processor may support any combination of memory models ranging from pure physical addressing to complex demand paged virtual memory systems. RTEMS supports a flat memory model which ranges contiguously over the processor's allowable address space. RTEMS does not support segmentation or virtual memory of any kind. The appropriate memory model for RTEMS provided by the targeted processor and related characteristics of that model are described in this chapter.

### 2.3.1 Flat Memory Model

RTEMS supports the i386 protected mode, flat memory model with paging disabled. In this mode, the i386 automatically converts every address from a logical to a physical address each time it is used. The i386 uses information provided in the segment registers and the Global Descriptor Table to convert these addresses. RTEMS assumes the existence of the following segments:

- a single code segment at protection level (0) which contains all application and executive code.
- a single data segment at protection level zero (0) which contains all application and executive data.

The i386 segment registers and associated selectors must be initialized when the initialize_executive directive is invoked. RTEMS treats the segment registers as system registers and does not modify or context switch them.

This i386 memory model supports a flat 32-bit address space with addresses ranging from 0x00000000 to 0xFFFFFFFF (4 gigabytes). Each address is represented by a 32-bit value and is byte addressable. The address may be used to reference a single byte, half-word (2-bytes), or word (4 bytes).

RTEMS does not require that logical addresses map directly to physical addresses, although it is desirable in many applications to do so. If logical and physical addresses are not the same, then an additional selector will be required so RTEMS can access the Interrupt Descriptor Table to install interrupt service routines. The selector number of this segment is provided to RTEMS in the CPU Dependent Information Table.

By not requiring that logical addresses map directly to physical addresses, the memory space of an RTEMS application can be separated from that of a ROM monitor. For example, on the Force Computers CPU386, the ROM monitor loads application programs into a logical address space where logical address 0x00000000 corresponds to physical address 0x0002000. On this board, RTEMS and the application use virtual addresses which do not map to physical addresses.

RTEMS assumes that the DS and ES registers contain the selector for the single data segment when a directive is invoked. This assumption is especially important when developing interrupt service routines.

## 2.4 Interrupt Processing

Different types of processors respond to the occurrence of an interrupt in their own unique fashion. In addition, each processor type provides a control mechanism to allow the proper handling of an interrupt. The processor dependent response to the interrupt modifies the execution state and results in the modification of the execution stream. This modification usually requires that an interrupt handler utilize the provided control mechanisms to return to the normal processing stream. Although RTEMS hides many of the processor dependent details of interrupt processing, it is important to understand how the RTEMS interrupt manager is mapped onto the processor's unique architecture. Discussed in this chapter are the the processor's response and control mechanisms as they pertain to RTEMS.

### 2.4.1 Vectoring of Interrupt Handler

Although the i386 supports multiple privilege levels, RTEMS and all user software executes at privilege level 0. This decision was made by the RTEMS designers to enhance compatibility with processors which do not provide sophisticated protection facilities like those of the i386. This decision greatly simplifies the discussion of i386 processing, as one need only consider interrupts without privilege transitions.

Upon receipt of an interrupt the i386 automatically performs the following actions:

- pushes the EFLAGS register
- pushes the far address of the interrupted instruction
- vectors to the interrupt service routine (ISR).

A nested interrupt is processed similarly by the i386.

### 2.4.2 Interrupt Stack Frame

The structure of the Interrupt Stack Frame for the i386 which is placed on the interrupt stack by the processor in response to an interrupt is as follows:

| | | |
|---|---|---|
| Old EFLAGS | Register | ESP+8 |
| UNUSED | Old CS | ESP+4 |
| Old | EIP | ESP |

### 2.4.3 Interrupt Levels

Although RTEMS supports 256 interrupt levels, the i386 only supports two – enabled and disabled. Interrupts are enabled when the interrupt-enable flag (IF) in the extended flags (EFLAGS) is set. Conversely, interrupt processing is inhibited when the IF is cleared. During a non-maskable interrupt, all other interrupts, including other non-maskable ones, are inhibited.

RTEMS interrupt levels 0 and 1 such that level zero (0) indicates that interrupts are fully enabled and level one that interrupts are disabled. All other RTEMS interrupt levels are undefined and their behavior is unpredictable.

### 2.4.4 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables interrupts before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than RTEMS_MAXIMUM_DISABLE_PERIOD microseconds on a RTEMS_MAXIMUM_DISABLE_PERIOD_MHZ Mhz i386 with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled within RTEMS was last calculated for Release RTEMS_RELEASE_FOR_MAXIMUM_DISABLE_PERIOD.]

Non-maskable interrupts (NMI) cannot be disabled, and ISRs which execute at this level MUST NEVER issue RTEMS system calls. If a directive is invoked, unpredictable results may occur due to the inability of RTEMS to protect its critical sections. However, ISRs that make no system calls may safely execute as non-maskable interrupts.

### 2.4.5 Interrupt Stack

The i386 family does not support a dedicated hardware interrupt stack. On this processor, RTEMS allocates and manages a dedicated interrupt stack. As part of vectoring a non-nested interrupt service routine, RTEMS switches from the stack of the interrupted task to a dedicated interrupt stack. When a non-nested interrupt returns, RTEMS switches back to the stack of the interrupted stack. The current stack pointer is not altered by RTEMS on nested interrupt.

Without a dedicated interrupt stack, every task in the system MUST have enough stack space to accommodate the worst case stack usage of that particular task and the interrupt service routines COMBINED. By supporting a dedicated interrupt stack, RTEMS significantly lowers the stack requirements for each task.

RTEMS allocates the dedicated interrupt stack from the Workspace Area. The amount of memory allocated for the interrupt stack is determined by the interrupt_stack_size field in the CPU Configuration Table.

## 2.5 Default Fatal Error Processing

Upon detection of a fatal error by either the application or RTEMS the fatal error manager is invoked. The fatal error manager will invoke the user-supplied fatal error handlers. If no user-supplied handlers are configured, the RTEMS provided default fatal error handler is invoked. If the user-supplied fatal error handlers return to the executive the default fatal error handler is then invoked. This chapter describes the precise operations of the default fatal error handler.

### 2.5.1 Default Fatal Error Handler Operations

The default fatal error handler which is invoked by the fatal_error_occurred directive when there is no user handler configured or the user handler returns control to RTEMS. The default fatal error handler disables processor interrupts, places the error code in EAX, and executes a HLT instruction to halt the processor.

## 2.6 Board Support Packages

An RTEMS Board Support Package (BSP) must be designed to support a particular processor and target board combination. This chapter presents a discussion of i386 specific BSP issues. For more information on developing a BSP, refer to the chapter titled Board Support Packages in the RTEMS Applications User's Guide.

### 2.6.1 System Reset

An RTEMS based application is initiated when the i386 processor is reset. When the i386 is reset,

- The EAX register is set to indicate the results of the processor's power-up self test. If the self-test was not executed, the contents of this register are undefined. Otherwise, a non-zero value indicates the processor is faulty and a zero value indicates a successful self-test.
- The DX register holds a component identifier and revision level. DH contains 3 to indicate an i386 component and DL contains a unique revision level indicator.
- Control register zero (CR0) is set such that the processor is in real mode with paging disabled. Other portions of CR0 are used to indicate the presence of a numeric coprocessor.
- All bits in the extended flags register (EFLAG) which are not permanently set are cleared. This inhibits all maskable interrupts.
- The Interrupt Descriptor Register (IDTR) is set to point at address zero.
- All segment registers are set to zero.
- The instruction pointer is set to 0x0000FFF0. The first instruction executed after a reset is actually at 0xFFFFFFF0 because the i386 asserts the upper twelve address until the first intersegment (FAR) JMP or CALL instruction. When a JMP or

CALL is executed, the upper twelve address lines are lowered and the processor begins executing in the first megabyte of memory.

Typically, an intersegment JMP to the application's initialization code is placed at address 0xFFFFFFF0.

### 2.6.2 Processor Initialization

This initialization code is responsible for initializing all data structures required by the i386 in protected mode and for actually entering protected mode. The i386 must be placed in protected mode and the segment registers and associated selectors must be initialized before the initialize_executive directive is invoked.

The initialization code is responsible for initializing the Global Descriptor Table such that the i386 is in the thirty-two bit flat memory model with paging disabled. In this mode, the i386 automatically converts every address from a logical to a physical address each time it is used. For more information on the memory model used by RTEMS, please refer to the Memory Model chapter in this document.

Since the processor is in real mode upon reset, the processor must be switched to protected mode before RTEMS can execute. Before switching to protected mode, at least one descriptor table and two descriptors must be created. Descriptors are needed for a code segment and a data segment. ( This will give you the flat memory model.) The stack can be placed in a normal read/write data segment, so no descriptor for the stack is needed. Before the GDT can be used, the base address and limit must be loaded into the GDTR register using an LGDT instruction.

If the hardware allows an NMI to be generated, you need to create the IDT and a gate for the NMI interrupt handler. Before the IDT can be used, the base address and limit for the idt must be loaded into the IDTR register using an LIDT instruction.

Protected mode is entered by setting thye PE bit in the CR0 register. Either a LMSW or MOV CR0 instruction may be used to set this bit. Because the processor overlaps the interpretation of several instructions, it is necessary to discard the instructions from the read-ahead cache. A JMP instruction immediately after the LMSW changes the flow and empties the processor if intructions which have been pre-fetched and/or decoded. At this point, the processor is in protected mode and begins to perform protected mode application initialization.

If the application requires that the IDTR be some value besides zero, then it should set it to the required value at this point. All tasks share the same i386 IDTR value. Because interrupts are enabled automatically by RTEMS as part of the initialize_executive directive, the IDTR MUST be set properly before this directive is invoked to insure correct interrupt vectoring. If processor caching is to be utilized, then it should be enabled during the reset application initialization code. The reset code which is executed before the call to initialize_executive has the following requirements:

For more information regarding the i386s data structures and their contents, refer to Intel's 386 Programmer's Reference Manual.

## 2.7 Processor Dependent Information Table

Any highly processor dependent information required to describe a processor to RTEMS is provided in the CPU Dependent Information Table. This table is not required for all processors supported by RTEMS. This chapter describes the contents, if any, for a particular processor type.

### 2.7.1 CPU Dependent Information Table

The i386 version of the RTEMS CPU Dependent Information Table contains the information required to interface a Board Support Package and RTEMS on the i386. This information is provided to allow RTEMS to interoperate effectively with the BSP. The C structure definition is given here:

```
typedef struct {
  void       (*pretasking_hook)( void );
  void       (*predriver_hook)( void );
  void       (*idle_task)( void );
  boolean      do_zero_of_workspace;
  unsigned32   idle_task_stack_size;
  unsigned32   interrupt_stack_size;
  unsigned32   extra_mpci_receive_server_stack;
  void *     (*stack_allocate_hook)( unsigned32 );
  void       (*stack_free_hook)( void* );
  /* end of fields required on all CPUs */

  unsigned32   interrupt_segment;
  void        *interrupt_vector_table;
} rtems_cpu_table;
```

`pretasking_hook` is the address of the user provided routine which is invoked once RTEMS APIs are initialized. This routine will be invoked before any system tasks are created. Interrupts are disabled. This field may be NULL to indicate that the hook is not utilized.

`predriver_hook` is the address of the user provided routine that is invoked immediately before the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`postdriver_hook` is the address of the user provided routine that is invoked immediately after the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`idle_task` is the address of the optional user provided routine which is used as the system's IDLE task. If this field is not NULL, then the RTEMS default IDLE task is not used. This field may be NULL to indicate that the default IDLE is to be used.

`do_zero_of_workspace`
: indicates whether RTEMS should zero the Workspace as part of its initialization. If set to TRUE, the Workspace is zeroed. Otherwise, it is not.

`idle_task_stack_size`
: is the size of the RTEMS idle task stack in bytes. If this number is less than MINIMUM_STACK_SIZE, then the idle task's stack will be MINIMUM_STACK_SIZE in byte.

`interrupt_stack_size`
: is the size of the RTEMS allocated interrupt stack in bytes. This value must be at least as large as MINIMUM_STACK_SIZE.

`extra_mpci_receive_server_stack`
: is the extra stack space allocated for the RTEMS MPCI receive server task in bytes. The MPCI receive server may invoke nearly all directives and may require extra stack space on some targets.

`stack_allocate_hook`
: is the address of the optional user provided routine which allocates memory for task stacks. If this hook is not NULL, then a stack_free_hook must be provided as well.

`stack_free_hook` is the address of the optional user provided routine which frees memory for task stacks. If this hook is not NULL, then a stack_allocate_hook must be provided as well.

`interrupt_segment` is the value of the selector which should be placed in a segment register to access the Interrupt Descriptor Table.

`interrupt_vector_table`
: is the base address of the Interrupt Descriptor Table relative to the interrupt_segment.

The contents of the i386 Interrupt Descriptor Table are discussed in Intel's i386 User's Manual. Structure definitions for the i386 IDT is provided by including the file rtems.h.

# 3 Motorola M68xxx and Coldfire Specific Information

The Real Time Executive for Multiprocessor Systems (RTEMS) is designed to be portable across multiple processor architectures. However, the nature of real-time systems makes it essential that the application designer understand certain processor dependent implementation details. These processor dependencies include calling convention, board support package issues, interrupt processing, exact RTEMS memory requirements, performance data, header files, and the assembly language interface to the executive.

This document discusses the Motorola MC68xxx architecture dependencies in this port of RTEMS. The MC68xxx family has a wide variety of CPU models within it. The part numbers for these models are generally divided into MC680xx and MC683xx. The MC680xx models are more general purpose processors with no integrated peripherals. The MC683xx models, on the other hand, are more specialized and have a variety of peripherals on chip including sophisticated timers and serial communications controllers.

It is highly recommended that the Motorola MC68xxx RTEMS application developer obtain and become familiar with the documentation for the processor being used as well as the documentation for the family as a whole.

### Architecture Documents

For information on the Motorola MC68xxx architecture, refer to the following documents available from Motorola (‘`http//www.moto.com/`’):

- *M68000 Family Reference, Motorola, FR68K/D.*

### MODEL SPECIFIC DOCUMENTS

For information on specific processor models and their associated coprocessors, refer to the following documents:

- *MC68020 User’s Manual, Motorola, MC68020UM/AD.*
- *MC68881/MC68882 Floating-Point Coprocessor User’s Manual, Motorola, MC68881UM/AD.*

## 3.1 CPU Model Dependent Features

Microprocessors are generally classified into families with a variety of CPU models or implementations within that family. Within a processor family, there is a high level of binary compatibility. This family may be based on either an architectural specification or on maintaining compatibility with a popular processor. Recent microprocessor families such as the SPARC or PowerPC are based on an architectural specification which is independent or any particular CPU model or implementation. Older families such as the M68xxx and the iX86 evolved as the manufacturer strived to produce higher performance processor models which maintained binary compatibility with older models.

RTEMS takes advantage of the similarity of the various models within a CPU family. Although the models do vary in significant ways, the high level of compatibility makes it possible to share the bulk of the CPU dependent executive code across the entire family. Each processor family supported by RTEMS has a list of features which vary between

CPU models within a family. For example, the most common model dependent feature regardless of CPU family is the presence or absence of a floating point unit or coprocessor. When defining the list of features present on a particular CPU model, one simply notes that floating point hardware is or is not present and defines a single constant appropriately. Conditional compilation is utilized to include the appropriate source code for this CPU model's feature set. It is important to note that this means that RTEMS is thus compiled using the appropriate feature set and compilation flags optimal for this CPU model used. The alternative would be to generate a binary which would execute on all family members using only the features which were always present.

This chapter presents the set of features which vary across SPARC implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file cpukit/score/cpu/m68k/m68k.h based upon the particular CPU model defined on the compilation command line.

### 3.1.1 CPU Model Name

The macro CPU_MODEL_NAME is a string which designates the name of this CPU model. For example, for the MC68020 processor, this macro is set to the string "mc68020".

### 3.1.2 Floating Point Unit

The macro M68K_HAS_FPU is set to 1 to indicate that this CPU model has a hardware floating point unit and 0 otherwise. It does not matter whether the hardware floating point support is incorporated on-chip or is an external coprocessor.

### 3.1.3 BFFFO Instruction

The macro M68K_HAS_BFFFO is set to 1 to indicate that this CPU model has the bfffo instruction.

### 3.1.4 Vector Base Register

The macro M68K_HAS_VBR is set to 1 to indicate that this CPU model has a vector base register (vbr).

### 3.1.5 Separate Stacks

The macro M68K_HAS_SEPARATE_STACKS is set to 1 to indicate that this CPU model has separate interrupt, user, and supervisor mode stacks.

### 3.1.6 Pre-Indexing Address Mode

The macro M68K_HAS_PREINDEXING is set to 1 to indicate that this CPU model has the pre-indexing address mode.

### 3.1.7 Extend Byte to Long Instruction

The macro M68K_HAS_EXTB_L is set to 1 to indicate that this CPU model has the extb.l instruction. This instruction is supposed to be available in all models based on the cpu32 core as well as mc68020 and up models.

## 3.2 Calling Conventions

Each high-level language compiler generates subroutine entry and exit code based upon a set of rules known as the compiler's calling convention. These rules address the following issues:

- register preservation and usage
- parameter passing
- call and return mechanism

A compiler's calling convention is of importance when interfacing to subroutines written in another language either assembly or high-level. Even when the high-level language and target processor are the same, different compilers may use different calling conventions. As a result, calling conventions are both processor and compiler dependent.

### 3.2.1 Processor Background

The MC68xxx architecture supports a simple yet effective call and return mechanism. A subroutine is invoked via the branch to subroutine (bsr) or the jump to subroutine (jsr) instructions. These instructions push the return address on the current stack. The return from subroutine (rts) instruction pops the return address off the current stack and transfers control to that instruction. It is is important to note that the MC68xxx call and return mechanism does not automatically save or restore any registers. It is the responsibility of the high-level language compiler to define the register preservation and usage convention.

### 3.2.2 Calling Mechanism

All RTEMS directives are invoked using either a bsr or jsr instruction and return to the user application via the rts instruction.

### 3.2.3 Register Usage

As discussed above, the bsr and jsr instructions do not automatically save any registers. RTEMS uses the registers D0, D1, A0, and A1 as scratch registers. These registers are not preserved by RTEMS directives therefore, the contents of these registers should not be assumed upon return from any RTEMS directive.

### 3.2.4 Parameter Passing

RTEMS assumes that arguments are placed on the current stack before the directive is invoked via the bsr or jsr instruction. The first argument is assumed to be closest to the return address on the stack. This means that the first argument of the C calling sequence is pushed last. The following pseudo-code illustrates the typical sequence used to call a RTEMS directive with three (3) arguments:

```
push third argument
push second argument
push first argument
invoke directive
remove arguments from the stack
```

The arguments to RTEMS are typically pushed onto the stack using a move instruction with a pre-decremented stack pointer as the destination. These arguments must be removed from

the stack after control is returned to the caller. This removal is typically accomplished by adding the size of the argument list in bytes to the current stack pointer.

### 3.2.5 User-Provided Routines

All user-provided routines invoked by RTEMS, such as user extensions, device drivers, and MPCI routines, must also adhere to these calling conventions.

## 3.3 Memory Model

A processor may support any combination of memory models ranging from pure physical addressing to complex demand paged virtual memory systems. RTEMS supports a flat memory model which ranges contiguously over the processor's allowable address space. RTEMS does not support segmentation or virtual memory of any kind. The appropriate memory model for RTEMS provided by the targeted processor and related characteristics of that model are described in this chapter.

### 3.3.1 Flat Memory Model

The MC68xxx family supports a flat 32-bit address space with addresses ranging from 0x00000000 to 0xFFFFFFFF (4 gigabytes). Each address is represented by a 32-bit value and is byte addressable. The address may be used to reference a single byte, word (2-bytes), or long word (4 bytes). Memory accesses within this address space are performed in big endian fashion by the processors in this family.

Some of the MC68xxx family members such as the MC68020, MC68030, and MC68040 support virtual memory and segmentation. The MC68020 requires external hardware support such as the MC68851 Paged Memory Management Unit coprocessor which is typically used to perform address translations for these systems. RTEMS does not support virtual memory or segmentation on any of the MC68xxx family members.

## 3.4 Interrupt Processing

Different types of processors respond to the occurrence of an interrupt in its own unique fashion. In addition, each processor type provides a control mechanism to allow for the proper handling of an interrupt. The processor dependent response to the interrupt modifies the current execution state and results in a change in the execution stream. Most processors require that an interrupt handler utilize some special control mechanisms to return to the normal processing stream. Although RTEMS hides many of the processor dependent details of interrupt processing, it is important to understand how the RTEMS interrupt manager is mapped onto the processor's unique architecture. Discussed in this chapter are the MC68xxx's interrupt response and control mechanisms as they pertain to RTEMS.

### 3.4.1 Vectoring of an Interrupt Handler

Depending on whether or not the particular CPU supports a separate interrupt stack, the MC68xxx family has two different interrupt handling models.

#### 3.4.1.1 Models Without Separate Interrupt Stacks

Upon receipt of an interrupt the MC68xxx family members without separate interrupt stacks automatically perform the following actions:

- To Be Written

#### 3.4.1.2 Models With Separate Interrupt Stacks

Upon receipt of an interrupt the MC68xxx family members with separate interrupt stacks automatically perform the following actions:

- saves the current status register (SR),
- clears the master/interrupt (M) bit of the SR to indicate the switch from master state to interrupt state,
- sets the privilege mode to supervisor,
- suppresses tracing,
- sets the interrupt mask level equal to the level of the interrupt being serviced,
- pushes an interrupt stack frame (ISF), which includes the program counter (PC), the status register (SR), and the format/exception vector offset (FVO) word, onto the supervisor and interrupt stacks,
- switches the current stack to the interrupt stack and vectors to an interrupt service routine (ISR). If the ISR was installed with the interrupt_catch directive, then the RTEMS interrupt handler will begin execution. The RTEMS interrupt handler saves all registers which are not preserved according to the calling conventions and invokes the application's ISR.

A nested interrupt is processed similarly by these CPU models with the exception that only a single ISF is placed on the interrupt stack and the current stack need not be switched.

The FVO word in the Interrupt Stack Frame is examined by RTEMS to determine when an outer most interrupt is being exited. Since the FVO is used by RTEMS for this purpose, the user application code MUST NOT modify this field.

The following shows the Interrupt Stack Frame for MC68xxx CPU models with separate interrupt stacks:

| | |
|---|---|
| Status Register | 0x0 |
| Program Counter High | 0x2 |
| Program Counter Low | 0x4 |
| Format/Vector Offset | 0x6 |

### 3.4.2 CPU Models Without VBR and RAM at 0

This is from a post by Zoltan Kocsi <zoltan@bendor.com.au> and is a nice trick in certain situations. In his words:

I think somebody on this list asked about the interupt vector handling w/o VBR and RAM at 0. The usual trick is to initialise the vector table (except the first 2 two entries, of course) to point to the same location BUT you also add the vector number times 0x1000000 to them. That is, bits 31-24 contain the vector number and 23-0 the address of the common handler. Since the PC is 32 bit wide but the actual address bus is only 24, the top byte will be in the PC but will be ignored when jumping onto your routine.

Then your common interrupt routine gets this info by loading the PC into some register and based on that info, you can jump to a vector in a vector table pointed by a virtual VBR:

```
//
//  Real vector table at 0
//

    .long   initial_sp
    .long   initial_pc
    .long   myhandler+0x02000000
    .long   myhandler+0x03000000
    .long   myhandler+0x04000000
    ...
    .long   myhandler+0xff000000


//
// This handler will jump to the interrupt routine   of which
// the address is stored at VBR[ vector_no ]
// The registers and stackframe will be intact, the interrupt
// routine will see exactly what it would see if it was called
// directly from the HW vector table at 0.
//

    .comm    VBR,4,2        // This defines the 'virtual' VBR
                            // From C: extern void *VBR;

myhandler:                  // At entry, PC contains the full vector
    move.l  %d0,-(%sp)      // Save d0
    move.l  %a0,-(%sp)      // Save a0
    lea     0(%pc),%a0      // Get the value of the PC
    move.l  %a0,%d0         // Copy it to a data reg, d0 is VV??????
    swap    %d0             // Now d0 is ????VV??
    and.w   #0xff00,%d0     // Now d0 is ????VV00 (1)
    lsr.w   #6,%d0          // Now d0.w contains the VBR table offset
    move.l  VBR,%a0         // Get the address from VBR to a0
    move.l  (%a0,%d0.w),%a0 // Fetch the vector
    move.l  4(%sp),%d0      // Restore d0
    move.l  %a0,4(%sp)      // Place target address to the stack
    move.l  (%sp)+,%a0      // Restore a0, target address is on TOS
    ret                     // This will jump to the handler and
                            // restore the stack

(1) If 'myhandler' is guaranteed to be in the first 64K, e.g. just
    after the vector table then that insn is not needed.
```

There are probably shorter ways to do this, but it I believe is enough to illustrate the trick. Optimisation is left as an exercise to the reader :-)

### 3.4.3 Interrupt Levels

Eight levels (0-7) of interrupt priorities are supported by MC68xxx family members with level seven (7) being the highest priority. Level zero (0) indicates that interrupts are fully enabled. Interrupt requests for interrupts with priorities less than or equal to the current interrupt mask level are ignored.

Although RTEMS supports 256 interrupt levels, the MC68xxx family only supports eight. RTEMS interrupt levels 0 through 7 directly correspond to MC68xxx interrupt levels. All other RTEMS interrupt levels are undefined and their behavior is unpredictable.

### 3.4.4 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables interrupts to level seven (7) before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than RTEMS_MAXIMUM_DISABLE_PERIOD microseconds on a RTEMS_MAXIMUM_DISABLE_PERIOD_MHZ Mhz MC68020 with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled is hand calculated. This calculation was last performed for Release RTEMS_RELEASE_FOR_MAXIMUM_DISABLE_PERIOD.]

Non-maskable interrupts (NMI) cannot be disabled, and ISRs which execute at this level MUST NEVER issue RTEMS system calls. If a directive is invoked, unpredictable results may occur due to the inability of RTEMS to protect its critical sections. However, ISRs that make no system calls may safely execute as non-maskable interrupts.

### 3.4.5 Interrupt Stack

RTEMS allocates the interrupt stack from the Workspace Area. The amount of memory allocated for the interrupt stack is determined by the interrupt_stack_size field in the CPU Configuration Table. During the initialization process, RTEMS will install its interrupt stack.

The MC68xxx port of RTEMS supports a software managed dedicated interrupt stack on those CPU models which do not support a separate interrupt stack in hardware.

## 3.5 Default Fatal Error Processing

Upon detection of a fatal error by either the application or RTEMS the fatal error manager is invoked. The fatal error manager will invoke the user-supplied fatal error handlers. If no user-supplied handlers are configured, the RTEMS provided default fatal error handler is invoked. If the user-supplied fatal error handlers return to the executive the default fatal error handler is then invoked. This chapter describes the precise operations of the default fatal error handler.

### 3.5.1 Default Fatal Error Handler Operations

The default fatal error handler which is invoked by the fatal_error_occurred directive when there is no user handler configured or the user handler returns control to RTEMS. The default fatal error handler disables processor interrupts to level 7, places the error code in D0, and executes a stop instruction to simulate a halt processor instruction.

## 3.6 Board Support Packages

An RTEMS Board Support Package (BSP) must be designed to support a particular processor and target board combination. This chapter presents a discussion of MC68020 specific BSP issues. For more information on developing a BSP, refer to the chapter titled Board Support Packages in the RTEMS Applications User's Guide.

### 3.6.1 System Reset

An RTEMS based application is initiated or re-initiated when the MC68020 processor is reset. When the MC68020 is reset, the processor performs the following actions:

- The tracing bits of the status register are cleared to disable tracing.
- The supervisor interrupt state is entered by setting the supervisor (S) bit and clearing the master/interrupt (M) bit of the status register.
- The interrupt mask of the status register is set to level 7 to effectively disable all maskable interrupts.
- The vector base register (VBR) is set to zero.
- The cache control register (CACR) is set to zero to disable and freeze the processor cache.
- The interrupt stack pointer (ISP) is set to the value stored at vector 0 (bytes 0-3) of the exception vector table (EVT).
- The program counter (PC) is set to the value stored at vector 1 (bytes 4-7) of the EVT.
- The processor begins execution at the address stored in the PC.

### 3.6.2 Processor Initialization

The address of the application's initialization code should be stored in the first vector of the EVT which will allow the immediate vectoring to the application code. If the application requires that the VBR be some value besides zero, then it should be set to the required value at this point. All tasks share the same MC68020's VBR value. Because interrupts are enabled automatically by RTEMS as part of the initialize executive directive, the VBR MUST be set before this directive is invoked to insure correct interrupt vectoring. If processor caching is to be utilized, then it should be enabled during the reset application initialization code.

In addition to the requirements described in the Board Support Packages chapter of the Applications User's Manual for the reset code which is executed before the call to initialize executive, the MC68020 version has the following specific requirements:

- Must leave the S bit of the status register set so that the MC68020 remains in the supervisor state.

- Must set the M bit of the status register to remove the MC68020 from the interrupt state.
- Must set the master stack pointer (MSP) such that a minimum stack size of MINIMUM_STACK_SIZE bytes is provided for the initialize executive directive.
- Must initialize the MC68020's vector table.

Note that the BSP is not responsible for allocating or installing the interrupt stack. RTEMS does this automatically as part of initialization. If the BSP does not install an interrupt stack and – for whatever reason – an interrupt occurs before initialize_executive is invoked, then the results are unpredictable.

## 3.7 Processor Dependent Information Table

Any highly processor dependent information required to describe a processor to RTEMS is provided in the CPU Dependent Information Table. This table is not required for all processors supported by RTEMS. This chapter describes the contents, if any, for a particular processor type.

### 3.7.1 CPU Dependent Information Table

The MC68xxx version of the RTEMS CPU Dependent Information Table contains the information required to interface a Board Support Package and RTEMS on the MC68xxx. This information is provided to allow RTEMS to interoperate effectively with the BSP. The C structure definition is given here:

```
typedef struct {
  void       (*pretasking_hook)( void );
  void       (*predriver_hook)( void );
  void       (*postdriver_hook)( void );
  void       (*idle_task)( void );
  boolean      do_zero_of_workspace;
  unsigned32   idle_task_stack_size;
  unsigned32   interrupt_stack_size;
  unsigned32   extra_mpci_receive_server_stack;
  void *     (*stack_allocate_hook)( unsigned32 );
  void       (*stack_free_hook)( void* );
  /* end of fields required on all CPUs */

  m68k_isr    *interrupt_vector_table;
} rtems_cpu_table;
```

`pretasking_hook` is the address of the user provided routine which is invoked once RTEMS APIs are initialized. This routine will be invoked before any system tasks are created. Interrupts are disabled. This field may be NULL to indicate that the hook is not utilized.

`predriver_hook` is the address of the user provided routine that is invoked immediately before the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`postdriver_hook` is the address of the user provided routine that is invoked immediately after the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`idle_task` is the address of the optional user provided routine which is used as the system's IDLE task. If this field is not NULL, then the RTEMS default IDLE task is not used. This field may be NULL to indicate that the default IDLE is to be used.

`do_zero_of_workspace`
indicates whether RTEMS should zero the Workspace as part of its initialization. If set to TRUE, the Workspace is zeroed. Otherwise, it is not.

`idle_task_stack_size`
is the size of the RTEMS idle task stack in bytes. If this number is less than MINIMUM_STACK_SIZE, then the idle task's stack will be MINIMUM_STACK_SIZE in byte.

`interrupt_stack_size`
is the size of the RTEMS allocated interrupt stack in bytes. This value must be at least as large as MINIMUM_STACK_SIZE.

`extra_mpci_receive_server_stack`
is the extra stack space allocated for the RTEMS MPCI receive server task in bytes. The MPCI receive server may invoke nearly all directives and may require extra stack space on some targets.

`stack_allocate_hook`
is the address of the optional user provided routine which allocates memory for task stacks. If this hook is not NULL, then a stack_free_hook must be provided as well.

`stack_free_hook` is the address of the optional user provided routine which frees memory for task stacks. If this hook is not NULL, then a stack_allocate_hook must be provided as well.

`interrupt_vector_table`
is the base address of the CPU's Exception Vector Table.

# 4 MIPS Specific Information

The Real Time Executive for Multiprocessor Systems (RTEMS) is designed to be portable across multiple processor architectures. However, the nature of real-time systems makes it essential that the application designer understand certain processor dependent implementation details. These processor dependencies include calling convention, board support package issues, interrupt processing, exact RTEMS memory requirements, performance data, header files, and the assembly language interface to the executive.

This document discusses the VENDOR XXX architecture dependencies in this port of RTEMS. The XXX family has a wide variety of CPU models within it. The part numbers ...

XXX fill in some things here

It is highly recommended that the XXX RTEMS application developer obtain and become familiar with the documentation for the processor being used as well as the documentation for the family as a whole.

### Architecture Documents

IDT docs are online at http://www.idt.com/products/risc/Welcome.html

For information on the XXX architecture, refer to the following documents available from VENDOR (‘`http//www.XXX.com/`’):

- *XXX Family Reference, VENDOR, PART NUMBER.*

### MODEL SPECIFIC DOCUMENTS

For information on specific processor models and their associated coprocessors, refer to the following documents:

- *XXX MODEL Manual, VENDOR, PART NUMBER.*
- *XXX MODEL Manual, VENDOR, PART NUMBER.*

## 4.1 CPU Model Dependent Features

Microprocessors are generally classified into families with a variety of CPU models or implementations within that family. Within a processor family, there is a high level of binary compatibility. This family may be based on either an architectural specification or on maintaining compatibility with a popular processor. Recent microprocessor families such as the SPARC or PowerPC are based on an architectural specification which is independent or any particular CPU model or implementation. Older families such as the M68xxx and the iX86 evolved as the manufacturer strived to produce higher performance processor models which maintained binary compatibility with older models.

RTEMS takes advantage of the similarity of the various models within a CPU family. Although the models do vary in significant ways, the high level of compatibility makes it possible to share the bulk of the CPU dependent executive code across the entire family. Each processor family supported by RTEMS has a list of features which vary between CPU models within a family. For example, the most common model dependent feature

regardless of CPU family is the presence or absence of a floating point unit or coprocessor. When defining the list of features present on a particular CPU model, one simply notes that floating point hardware is or is not present and defines a single constant appropriately. Conditional compilation is utilized to include the appropriate source code for this CPU model's feature set. It is important to note that this means that RTEMS is thus compiled using the appropriate feature set and compilation flags optimal for this CPU model used. The alternative would be to generate a binary which would execute on all family members using only the features which were always present.

This chapter presents the set of features which vary across SPARC implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file cpukit/score/cpu/XXX/XXX.h based upon the particular CPU model defined on the compilation command line.

### 4.1.1 CPU Model Name

The macro CPU_MODEL_NAME is a string which designates the name of this CPU model. For example, for the MODEL processor, this macro is set to the string "XXX".

### 4.1.2 Floating Point Unit

The macro XXX_HAS_FPU is set to 1 to indicate that this CPU model has a hardware floating point unit and 0 otherwise. It does not matter whether the hardware floating point support is incorporated on-chip or is an external coprocessor.

### 4.1.3 Another Optional Feature

The macro XXX

## 4.2 Calling Conventions

Each high-level language compiler generates subroutine entry and exit code based upon a set of rules known as the compiler's calling convention. These rules address the following issues:

- register preservation and usage
- parameter passing
- call and return mechanism

A compiler's calling convention is of importance when interfacing to subroutines written in another language either assembly or high-level. Even when the high-level language and target processor are the same, different compilers may use different calling conventions. As a result, calling conventions are both processor and compiler dependent.

### 4.2.1 Processor Background

The MC68xxx architecture supports a simple yet effective call and return mechanism. A subroutine is invoked via the branch to subroutine (`XXX`) or the jump to subroutine (`XXX`) instructions. These instructions push the return address on the current stack. The return from subroutine (`XXX`) instruction pops the return address off the current stack and transfers control to that instruction. It is is important to note that the XXX call and return

mechanism does not automatically save or restore any registers. It is the responsibility of the high-level language compiler to define the register preservation and usage convention.

### 4.2.2 Calling Mechanism

All RTEMS directives are invoked using either a `XXX` or `XXX` instruction and return to the user application via the `XXX` instruction.

### 4.2.3 Register Usage

As discussed above, the `XXX` and `XXX` instructions do not automatically save any registers. RTEMS uses the registers **D0**, **D1**, **A0**, and **A1** as scratch registers. These registers are not preserved by RTEMS directives therefore, the contents of these registers should not be assumed upon return from any RTEMS directive.

### 4.2.4 Parameter Passing

RTEMS assumes that arguments are placed on the current stack before the directive is invoked via the `XXX` or `XXX` instruction. The first argument is assumed to be closest to the return address on the stack. This means that the first argument of the C calling sequence is pushed last. The following pseudo-code illustrates the typical sequence used to call a RTEMS directive with three (3) arguments:

```
push third argument
push second argument
push first argument
invoke directive
remove arguments from the stack
```

The arguments to RTEMS are typically pushed onto the stack using a move instruction with a pre-decremented stack pointer as the destination. These arguments must be removed from the stack after control is returned to the caller. This removal is typically accomplished by adding the size of the argument list in bytes to the current stack pointer.

### 4.2.5 User-Provided Routines

All user-provided routines invoked by RTEMS, such as user extensions, device drivers, and MPCI routines, must also adhere to these calling conventions.

## 4.3 Memory Model

A processor may support any combination of memory models ranging from pure physical addressing to complex demand paged virtual memory systems. RTEMS supports a flat memory model which ranges contiguously over the processor's allowable address space. RTEMS does not support segmentation or virtual memory of any kind. The appropriate memory model for RTEMS provided by the targeted processor and related characteristics of that model are described in this chapter.

### 4.3.1 Flat Memory Model

The XXX family supports a flat 32-bit address space with addresses ranging from 0x00000000 to 0xFFFFFFFF (4 gigabytes). Each address is represented by a 32-bit value and is byte addressable. The address may be used to reference a single byte, word (2-bytes),

or long word (4 bytes). Memory accesses within this address space are performed in big endian fashion by the processors in this family.

Some of the XXX family members such as the XXX, XXX, and XXX support virtual memory and segmentation. The XXX requires external hardware support such as the XXX Paged Memory Management Unit coprocessor which is typically used to perform address translations for these systems. RTEMS does not support virtual memory or segmentation on any of the XXX family members.

## 4.4 Interrupt Processing

Different types of processors respond to the occurrence of an interrupt in its own unique fashion. In addition, each processor type provides a control mechanism to allow for the proper handling of an interrupt. The processor dependent response to the interrupt modifies the current execution state and results in a change in the execution stream. Most processors require that an interrupt handler utilize some special control mechanisms to return to the normal processing stream. Although RTEMS hides many of the processor dependent details of interrupt processing, it is important to understand how the RTEMS interrupt manager is mapped onto the processor's unique architecture. Discussed in this chapter are the XXX's interrupt response and control mechanisms as they pertain to RTEMS.

### 4.4.1 Vectoring of an Interrupt Handler

Depending on whether or not the particular CPU supports a separate interrupt stack, the XXX family has two different interrupt handling models.

#### 4.4.1.1 Models Without Separate Interrupt Stacks

Upon receipt of an interrupt the XXX family members without separate interrupt stacks automatically perform the following actions:

- To Be Written

#### 4.4.1.2 Models With Separate Interrupt Stacks

Upon receipt of an interrupt the XXX family members with separate interrupt stacks automatically perform the following actions:

- saves the current status register (SR),
- clears the master/interrupt (M) bit of the SR to indicate the switch from master state to interrupt state,
- sets the privilege mode to supervisor,
- suppresses tracing,
- sets the interrupt mask level equal to the level of the interrupt being serviced,
- pushes an interrupt stack frame (ISF), which includes the program counter (PC), the status register (SR), and the format/exception vector offset (FVO) word, onto the supervisor and interrupt stacks,
- switches the current stack to the interrupt stack and vectors to an interrupt service routine (ISR). If the ISR was installed with the interrupt_catch directive, then the RTEMS interrupt handler will begin execution. The RTEMS interrupt handler saves

all registers which are not preserved according to the calling conventions and invokes the application's ISR.

A nested interrupt is processed similarly by these CPU models with the exception that only a single ISF is placed on the interrupt stack and the current stack need not be switched.

The FVO word in the Interrupt Stack Frame is examined by RTEMS to determine when an outer most interrupt is being exited. Since the FVO is used by RTEMS for this purpose, the user application code MUST NOT modify this field.

The following shows the Interrupt Stack Frame for XXX CPU models with separate interrupt stacks:

| | |
|---|---|
| Status Register | 0x0 |
| Program Counter High | 0x2 |
| Program Counter Low | 0x4 |
| Format/Vector Offset | 0x6 |

### 4.4.2 Interrupt Levels

Eight levels (0-7) of interrupt priorities are supported by XXX family members with level seven (7) being the highest priority. Level zero (0) indicates that interrupts are fully enabled. Interrupt requests for interrupts with priorities less than or equal to the current interrupt mask level are ignored.

Although RTEMS supports 256 interrupt levels, the XXX family only supports eight. RTEMS interrupt levels 0 through 7 directly correspond to XXX interrupt levels. All other RTEMS interrupt levels are undefined and their behavior is unpredictable.

### 4.4.3 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables interrupts to level seven (7) before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than RTEMS_MAXIMUM_DISABLE_PERIOD microseconds on a RTEMS_MAXIMUM_DISABLE_PERIOD_MHZ Mhz processor with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled is hand calculated. This calculation was last performed for Release RTEMS_RELEASE_FOR_MAXIMUM_DISABLE_PERIOD.]

Non-maskable interrupts (NMI) cannot be disabled, and ISRs which execute at this level MUST NEVER issue RTEMS system calls. If a directive is invoked, unpredictable results may occur due to the inability of RTEMS to protect its critical sections. However, ISRs that make no system calls may safely execute as non-maskable interrupts.

### 4.4.4 Interrupt Stack

RTEMS allocates the interrupt stack from the Workspace Area. The amount of memory allocated for the interrupt stack is determined by the interrupt_stack_size field in the CPU

Configuration Table. During the initialization process, RTEMS will install its interrupt stack.

The mips port of RTEMS supports a software managed dedicated interrupt stack on those CPU models which do not support a separate interrupt stack in hardware.

## 4.5 Default Fatal Error Processing

Upon detection of a fatal error by either the application or RTEMS the fatal error manager is invoked. The fatal error manager will invoke the user-supplied fatal error handlers. If no user-supplied handlers are configured, the RTEMS provided default fatal error handler is invoked. If the user-supplied fatal error handlers return to the executive the default fatal error handler is then invoked. This chapter describes the precise operations of the default fatal error handler.

### 4.5.1 Default Fatal Error Handler Operations

The default fatal error handler which is invoked by the `rtems_fatal_error_occurred` directive when there is no user handler configured or the user handler returns control to RTEMS. The default fatal error handler disables processor interrupts, places the error code in **XXX**, and executes a `XXX` instruction to simulate a halt processor instruction.

## 4.6 Board Support Packages

An RTEMS Board Support Package (BSP) must be designed to support a particular processor and target board combination. This chapter presents a discussion of XXX specific BSP issues. For more information on developing a BSP, refer to the chapter titled Board Support Packages in the RTEMS Applications User's Guide.

### 4.6.1 System Reset

An RTEMS based application is initiated or re-initiated when the XXX processor is reset. When the XXX is reset, the processor performs the following actions:

- The tracing bits of the status register are cleared to disable tracing.
- The supervisor interrupt state is entered by setting the supervisor (S) bit and clearing the master/interrupt (M) bit of the status register.
- The interrupt mask of the status register is set to level 7 to effectively disable all maskable interrupts.
- The vector base register (VBR) is set to zero.
- The cache control register (CACR) is set to zero to disable and freeze the processor cache.
- The interrupt stack pointer (ISP) is set to the value stored at vector 0 (bytes 0-3) of the exception vector table (EVT).
- The program counter (PC) is set to the value stored at vector 1 (bytes 4-7) of the EVT.
- The processor begins execution at the address stored in the PC.

### 4.6.2 Processor Initialization

The address of the application's initialization code should be stored in the first vector of the EVT which will allow the immediate vectoring to the application code. If the application requires that the VBR be some value besides zero, then it should be set to the required value at this point. All tasks share the same XXX's VBR value. Because interrupts are enabled automatically by RTEMS as part of the initialize executive directive, the VBR MUST be set before this directive is invoked to insure correct interrupt vectoring. If processor caching is to be utilized, then it should be enabled during the reset application initialization code.

In addition to the requirements described in the Board Support Packages chapter of the Applications User's Manual for the reset code which is executed before the call to initialize executive, the XXX version has the following specific requirements:

- Must leave the S bit of the status register set so that the XXX remains in the supervisor state.
- Must set the M bit of the status register to remove the XXX from the interrupt state.
- Must set the master stack pointer (MSP) such that a minimum stack size of MINIMUM_STACK_SIZE bytes is provided for the initialize executive directive.
- Must initialize the XXX's vector table.

Note that the BSP is not responsible for allocating or installing the interrupt stack. RTEMS does this automatically as part of initialization. If the BSP does not install an interrupt stack and – for whatever reason – an interrupt occurs before initialize_executive is invoked, then the results are unpredictable.

## 4.7 Processor Dependent Information Table

Any highly processor dependent information required to describe a processor to RTEMS is provided in the CPU Dependent Information Table. This table is not required for all processors supported by RTEMS. This chapter describes the contents, if any, for a particular processor type.

### 4.7.1 CPU Dependent Information Table

The XXX version of the RTEMS CPU Dependent Information Table contains the information required to interface a Board Support Package and RTEMS on the XXX. This information is provided to allow RTEMS to interoperate effectively with the BSP. The C structure definition is given here:

```
typedef struct {
  void       (*pretasking_hook)( void );
  void       (*predriver_hook)( void );
  void       (*postdriver_hook)( void );
  void       (*idle_task)( void );
  boolean      do_zero_of_workspace;
  unsigned32   idle_task_stack_size;
  unsigned32   interrupt_stack_size;
  unsigned32   extra_mpci_receive_server_stack;
  void *     (*stack_allocate_hook)( unsigned32 );
  void       (*stack_free_hook)( void* );
  /* end of fields required on all CPUs */

  /* XXX CPU family dependent stuff */
} rtems_cpu_table;
```

`pretasking_hook` is the address of the user provided routine which is invoked once RTEMS APIs are initialized. This routine will be invoked before any system tasks are created. Interrupts are disabled. This field may be NULL to indicate that the hook is not utilized.

`predriver_hook` is the address of the user provided routine that is invoked immediately before the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`postdriver_hook` is the address of the user provided routine that is invoked immediately after the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`idle_task` is the address of the optional user provided routine which is used as the system's IDLE task. If this field is not NULL, then the RTEMS default IDLE task is not used. This field may be NULL to indicate that the default IDLE is to be used.

`do_zero_of_workspace`
indicates whether RTEMS should zero the Workspace as part of its initialization. If set to TRUE, the Workspace is zeroed. Otherwise, it is not.

`idle_task_stack_size`
is the size of the RTEMS idle task stack in bytes. If this number is less than MINIMUM_STACK_SIZE, then the idle task's stack will be MINIMUM_STACK_SIZE in byte.

`interrupt_stack_size`
is the size of the RTEMS allocated interrupt stack in bytes. This value must be at least as large as MINIMUM_STACK_SIZE.

`extra_mpci_receive_server_stack`
: is the extra stack space allocated for the RTEMS MPCI receive server task in bytes. The MPCI receive server may invoke nearly all directives and may require extra stack space on some targets.

`stack_allocate_hook`
: is the address of the optional user provided routine which allocates memory for task stacks. If this hook is not NULL, then a stack_free_hook must be provided as well.

`stack_free_hook`
: is the address of the optional user provided routine which frees memory for task stacks. If this hook is not NULL, then a stack_allocate_hook must be provided as well.

`XXX`
: is where the CPU family dependent stuff goes.

# 5 PowerPC Specific Information

The Real Time Executive for Multiprocessor Systems (RTEMS) is designed to be portable across multiple processor architectures. However, the nature of real-time systems makes it essential that the application designer understand certain processor dependent implementation details. These processor dependencies include calling convention, board support package issues, interrupt processing, exact RTEMS memory requirements, performance data, header files, and the assembly language interface to the executive.

This document discusses the PowerPC architecture dependencies in this port of RTEMS.

It is highly recommended that the PowerPC RTEMS application developer obtain and become familiar with the documentation for the processor being used as well as the specification for the revision of the PowerPC architecture which corresponds to that processor.

## PowerPC Architecture Documents

For information on the PowerPC architecture, refer to the following documents available from Motorola and IBM:

- *PowerPC Microprocessor Family: The Programming Environment* (Motorola Document MPRPPCFPE-01).
- *IBM PPC403GB Embedded Controller User's Manual.*
- *PoweRisControl MPC500 Family RCPU RISC Central Processing Unit Reference Manual* (Motorola Document RCPUURM/AD).
- *PowerPC 601 RISC Microprocessor User's Manual* (Motorola Document MPR601UM/AD).
- *PowerPC 603 RISC Microprocessor User's Manual* (Motorola Document MPR603UM/AD).
- *PowerPC 603e RISC Microprocessor User's Manual* (Motorola Document MPR603EUM/AD).
- *PowerPC 604 RISC Microprocessor User's Manual* (Motorola Document MPR604UM/AD).
- *PowerPC MPC821 Portable Systems Microprocessor User's Manual* (Motorola Document MPC821UM/AD).
- *PowerQUICC MPC860 User's Manual* (Motorola Document MPC860UM/AD).

Motorola maintains an on-line electronic library for the PowerPC at the following URL:

*http://www.mot.com/powerpc/library/library.html*

This site has a a wealth of information and examples. Many of the manuals are available from that site in electronic format.

## PowerPC Processor Simulator Information

PSIM is a program which emulates the Instruction Set Architecture of the PowerPC microprocessor family. It is reely available in source code form under the terms of the GNU General Public License (version 2 or later). PSIM can be integrated with the GNU Debugger (gdb) to execute and debug PowerPC executables on non-PowerPC hosts. PSIM

supports the addition of user provided device models which can be used to allow one to develop and debug embedded applications using the simulator.

The latest version of PSIM is made available to the public via anonymous ftp at ftp://ftp.ci.com.au/pub/psim or ftp://cambridge.cygnus.com/pub/psim. There is also a mailing list at powerpc-psim@ci.com.au.

## 5.1 CPU Model Dependent Features

Microprocessors are generally classified into families with a variety of CPU models or implementations within that family. Within a processor family, there is a high level of binary compatibility. This family may be based on either an architectural specification or on maintaining compatibility with a popular processor. Recent microprocessor families such as the SPARC, and PowerPC are based on an architectural specification which is independent or any particular CPU model or implementation. Older families such as the M68xxx and the iX86 evolved as the manufacturer strived to produce higher performance processor models which maintained binary compatibility with older models.

RTEMS takes advantage of the similarity of the various models within a CPU family. Although the models do vary in significant ways, the high level of compatibility makes it possible to share the bulk of the CPU dependent executive code across the entire family.

### 5.1.1 CPU Model Feature Flags

Each processor family supported by RTEMS has a list of features which vary between CPU models within a family. For example, the most common model dependent feature regardless of CPU family is the presence or absence of a floating point unit or coprocessor. When defining the list of features present on a particular CPU model, one simply notes that floating point hardware is or is not present and defines a single constant appropriately. Conditional compilation is utilized to include the appropriate source code for this CPU model's feature set. It is important to note that this means that RTEMS is thus compiled using the appropriate feature set and compilation flags optimal for this CPU model used. The alternative would be to generate a binary which would execute on all family members using only the features which were always present.

This section presents the set of features which vary across PowerPC implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file cpukit/score/cpu/ppc/ppc.h based upon the particular CPU model defined on the compilation command line.

#### 5.1.1.1 CPU Model Name

The macro CPU_MODEL_NAME is a string which designates the name of this CPU model. For example, for the PowerPC 603e model, this macro is set to the string "PowerPC 603e".

#### 5.1.1.2 Floating Point Unit

The macro PPC_HAS_FPU is set to 1 to indicate that this CPU model has a hardware floating point unit and 0 otherwise.

#### 5.1.1.3 Alignment

The macro PPC_ALIGNMENT is set to the PowerPC model's worst case alignment requirement for data types on a byte boundary. This value is used to derive the alignment restrictions for memory allocated from regions and partitions.

#### 5.1.1.4 Cache Alignment

The macro PPC_CACHE_ALIGNMENT is set to the line size of the cache. It is used to align the entry point of critical routines so that as much code as possible can be retrieved with the initial read into cache. This is done for the interrupt handler as well as the context switch routines.

In addition, the "shortcut" data structure used by the PowerPC implementation to ease access to data elements frequently accessed by RTEMS routines implemented in assembly language is aligned using this value.

#### 5.1.1.5 Maximum Interrupts

The macro PPC_INTERRUPT_MAX is set to the number of exception sources supported by this PowerPC model.

#### 5.1.1.6 Has Double Precision Floating Point

The macro PPC_HAS_DOUBLE is set to 1 to indicate that the PowerPC model has support for double precision floating point numbers. This is important because the floating point registers need only be four bytes wide (not eight) if double precision is not supported.

#### 5.1.1.7 Critical Interrupts

The macro PPC_HAS_RFCI is set to 1 to indicate that the PowerPC model has the Critical Interrupt capability as defined by the IBM 403 models.

#### 5.1.1.8 Use Multiword Load/Store Instructions

The macro PPC_USE_MULTIPLE is set to 1 to indicate that multiword load and store instructions should be used to perform context switch operations. The relative efficiency of multiword load and store instructions versus an equivalent set of single word load and store instructions varies based upon the PowerPC model.

#### 5.1.1.9 Instruction Cache Size

The macro PPC_I_CACHE is set to the size in bytes of the instruction cache.

#### 5.1.1.10 Data Cache Size

The macro PPC_D_CACHE is set to the size in bytes of the data cache.

#### 5.1.1.11 Debug Model

The macro PPC_DEBUG_MODEL is set to indicate the debug support features present in this CPU model. The following debug support feature sets are currently supported:

`PPC_DEBUG_MODEL_STANDARD`
: indicates that the single-step trace enable (SE) and branch trace enable (BE) bits in the MSR are supported by this CPU model.

`PPC_DEBUG_MODEL_SINGLE_STEP_ONLY`
: indicates that only the single-step trace enable (SE) bit in the MSR is supported by this CPU model.

`PPC_DEBUG_MODEL_IBM4xx`
: indicates that the debug exception enable (DE) bit in the MSR is supported by this CPU model. At this time, this particular debug feature set has only been seen in the IBM 4xx series.

#### 5.1.1.12 Low Power Model

The macro PPC_LOW_POWER_MODE is set to indicate the low power model supported by this CPU model. The following low power modes are currently supported.

`PPC_LOW_POWER_MODE_NONE`
: indicates that this CPU model has no low power mode support.

`PPC_LOW_POWER_MODE_STANDARD`
: indicates that this CPU model follows the low power model defined for the PPC603e.

## 5.2 Calling Conventions

Each high-level language compiler generates subroutine entry and exit code based upon a set of rules known as the compiler's calling convention. These rules address the following issues:

- register preservation and usage
- parameter passing
- call and return mechanism

A compiler's calling convention is of importance when interfacing to subroutines written in another language either assembly or high-level. Even when the high-level language and target processor are the same, different compilers may use different calling conventions. As a result, calling conventions are both processor and compiler dependent.

RTEMS supports the Embedded Application Binary Interface (EABI) calling convention. Documentation for EABI is available by sending a message with a subject line of "EABI" to eabi@goth.sis.mot.com.

### 5.2.1 Programming Model

This section discusses the programming model for the PowerPC architecture.

#### 5.2.1.1 Non-Floating Point Registers

The PowerPC architecture defines thirty-two non-floating point registers directly visible to the programmer. In thirty-two bit implementations, each register is thirty-two bits wide. In sixty-four bit implementations, each register is sixty-four bits wide.

These registers are referred to as `gpr0` to `gpr31`.

Some of the registers serve defined roles in the EABI programming model. The following table describes the role of each of these registers:

| Register Name | Alternate Names | Description |
|---|---|---|
| r1 | sp | stack pointer |
| r2 | NA | global pointer to the Small Constant Area (SDA2) |
| r3 - r12 | NA | parameter and result passing |
| r13 | NA | global pointer to the Small Data Area (SDA2) |

#### 5.2.1.2 Floating Point Registers

The PowerPC architecture includes thirty-two, sixty-four bit floating point registers. All PowerPC floating point instructions interpret these registers as 32 double precision floating point registers, regardless of whether the processor has 64-bit or 32-bit implementation.

The floating point status and control register (fpscr) records exceptions and the type of result generated by floating-point operations. Additionally, it controls the rounding mode of operations and allows the reporting of floating exceptions to be enabled or disabled.

#### 5.2.1.3 Special Registers

The PowerPC architecture includes a number of special registers which are critical to the programming model:

**Machine State Register**

The MSR contains the processor mode, power management mode, endian mode, exception information, privilege level, floating point available and floating point excepiton mode, address translation information and the exception prefix.

**Link Register**

The LR contains the return address after a function call. This register must be saved before a subsequent subroutine call can be made. The use of this register is discussed further in the **Call and Return Mechanism** section below.

**Count Register**

The CTR contains the iteration variable for some loops. It may also be used for indirect function calls and jumps.

### 5.2.2 Call and Return Mechanism

The PowerPC architecture supports a simple yet effective call and return mechanism. A subroutine is invoked via the "branch and link" (`bl`) and "brank and link absolute" (`bla`) instructions. This instructions place the return address in the Link Register (LR). The callee returns to the caller by executing a "branch unconditional to the link register" (`blr`) instruction. Thus the callee returns to the caller via a jump to the return address which is stored in the LR.

The previous contents of the LR are not automatically saved by either the `bl` or `bla`. It is the responsibility of the callee to save the contents of the LR before invoking another subroutine. If the callee invokes another subroutine, it must restore the LR before executing the `blr` instruction to return to the caller.

It is important to note that the PowerPC subroutine call and return mechanism does not automatically save and restore any registers.

The LR may be accessed as special purpose register 8 (SPR8) using the "move from special register" (mfspr) and "move to special register" (mtspr) instructions.

### 5.2.3 Calling Mechanism

All RTEMS directives are invoked using the regular PowerPC EABI calling convention via the bl or bla instructions.

### 5.2.4 Register Usage

As discussed above, the call instruction does not automatically save any registers. It is the responsibility of the callee to save and restore any registers which must be preserved across subroutine calls. The callee is responsible for saving callee-preserved registers to the program stack and restoring them before returning to the caller.

### 5.2.5 Parameter Passing

RTEMS assumes that arguments are placed in the general purpose registers with the first argument in register 3 (r3), the second argument in general purpose register 4 (r4), and so forth until the seventh argument is in general purpose register 10 (r10). If there are more than seven arguments, then subsequent arguments are placed on the program stack. The following pseudo-code illustrates the typical sequence used to call a RTEMS directive with three (3) arguments:

```
load third argument into r5
load second argument into r4
load first argument into r3
invoke directive
```

### 5.2.6 User-Provided Routines

All user-provided routines invoked by RTEMS, such as user extensions, device drivers, and MPCI routines, must also adhere to these same calling conventions.

## 5.3 Memory Model

A processor may support any combination of memory models ranging from pure physical addressing to complex demand paged virtual memory systems. RTEMS supports a flat memory model which ranges contiguously over the processor's allowable address space. RTEMS does not support segmentation or virtual memory of any kind. The appropriate memory model for RTEMS provided by the targeted processor and related characteristics of that model are described in this chapter.

### 5.3.1 Flat Memory Model

The PowerPC architecture supports a variety of memory models. RTEMS supports the PowerPC using a flat memory model with paging disabled. In this mode, the PowerPC automatically converts every address from a logical to a physical address each time it is used. The PowerPC uses information provided in the Block Address Translation (BAT) to convert these addresses.

Implementations of the PowerPC architecture may be thirty-two or sixty-four bit. The PowerPC architecture supports a flat thirty-two or sixty-four bit address space with addresses ranging from 0x00000000 to 0xFFFFFFFF (4 gigabytes) in thirty-two bit implementations or to 0xFFFFFFFFFFFFFFFF in sixty-four bit implementations. Each address is represented by either a thirty-two bit or sixty-four bit value and is byte addressable. The address may be used to reference a single byte, half-word (2-bytes), word (4 bytes), or in sixty-four bit implementations a doubleword (8 bytes). Memory accesses within the address space are performed in big or little endian fashion by the PowerPC based upon the current setting of the Little-endian mode enable bit (LE) in the Machine State Register (MSR). While the processor is in big endian mode, memory accesses which are not properly aligned generate an "alignment exception" (vector offset 0x00600). In little endian mode, the PowerPC architecture does not require the processor to generate alignment exceptions.

The following table lists the alignment requirements for a variety of data accesses:

| Data Type | Alignment Requirement |
|---|---|
| byte | 1 |
| half-word | 2 |
| word | 4 |
| doubleword | 8 |

Doubleword load and store operations are only available in PowerPC CPU models which are sixty-four bit implementations.

RTEMS does not directly support any PowerPC Memory Management Units, therefore, virtual memory or segmentation systems involving the PowerPC are not supported.

## 5.4 Interrupt Processing

Different types of processors respond to the occurrence of an interrupt in its own unique fashion. In addition, each processor type provides a control mechanism to allow for the proper handling of an interrupt. The processor dependent response to the interrupt modifies the current execution state and results in a change in the execution stream. Most processors require that an interrupt handler utilize some special control mechanisms to return to the normal processing stream. Although RTEMS hides many of the processor dependent details of interrupt processing, it is important to understand how the RTEMS interrupt manager is mapped onto the processor's unique architecture. Discussed in this chapter are the PowerPC's interrupt response and control mechanisms as they pertain to RTEMS.

RTEMS and associated documentation uses the terms interrupt and vector. In the PowerPC architecture, these terms correspond to exception and exception handler, respectively. The terms will be used interchangeably in this manual.

### 5.4.1 Synchronous Versus Asynchronous Exceptions

In the PowerPC architecture exceptions can be either precise or imprecise and either synchronous or asynchronous. Asynchronous exceptions occur when an external event interrupts the processor. Synchronous exceptions are caused by the actions of an instruction. During an exception SRR0 is used to calculate where instruction processing should resume.

All instructions prior to the resume instruction will have completed execution. SRR1 is used to store the machine status.

There are two asynchronous nonmaskable, highest-priority exceptions system reset and machine check. There are two asynchrononous maskable low-priority exceptions external interrupt and decrementer. Nonmaskable execptions are never delayed, therefore if two nonmaskable, asynchronous exceptions occur in immediate succession, the state information saved by the first exception may be overwritten when the subsequent exception occurs.

The PowerPC arcitecure defines one imprecise exception, the imprecise floating point enabled exception. All other synchronous exceptions are precise. The synchronization occuring during asynchronous precise exceptions conforms to the requirements for context synchronization.

### 5.4.2 Vectoring of Interrupt Handler

Upon determining that an exception can be taken the PowerPC automatically performs the following actions:

- an instruction address is loaded into SRR0
- bits 33-36 and 42-47 of SRR1 are loaded with information specific to the exception.
- bits 0-32, 37-41, and 48-63 of SRR1 are loaded with corresponding bits from the MSR.
- the MSR is set based upon the exception type.
- instruction fetch and execution resumes, using the new MSR value, at a location specific to the execption type.

If the interrupt handler was installed as an RTEMS interrupt handler, then upon receipt of the interrupt, the processor passes control to the RTEMS interrupt handler which performs the following actions:

- saves the state of the interrupted task on it's stack,
- saves all registers which are not normally preserved by the calling sequence so the user's interrupt service routine can be written in a high-level language.
- if this is the outermost (i.e. non-nested) interrupt, then the RTEMS interrupt handler switches from the current stack to the interrupt stack,
- enables exceptions,
- invokes the vectors to a user interrupt service routine (ISR).

Asynchronous interrupts are ignored while exceptions are disabled. Synchronous interrupts which occur while are disabled result in the CPU being forced into an error mode.

A nested interrupt is processed similarly with the exception that the current stack need not be switched to the interrupt stack.

### 5.4.3 Interrupt Levels

The PowerPC architecture supports only a single external asynchronous interrupt source. This interrupt source may be enabled and disabled via the External Interrupt Enable (EE)

bit in the Machine State Register (MSR). Thus only two level (enabled and disabled) of external device interrupt priorities are directly supported by the PowerPC architecture.

Some PowerPC implementations include a Critical Interrupt capability which is often used to receive interrupts from high priority external devices.

The RTEMS interrupt level mapping scheme for the PowerPC is not a numeric level as on most RTEMS ports. It is a bit mapping in which the least three significiant bits of the interrupt level are mapped directly to the enabling of specific interrupt sources as follows:

**Critical Interrupt** Setting bit 0 (the least significant bit) of the interrupt level enables the Critical Interrupt source, if it is available on this CPU model.

**Machine Check** Setting bit 1 of the interrupt level enables Machine Check execptions.

**External Interrupt** Setting bit 2 of the interrupt level enables External Interrupt execptions.

All other bits in the RTEMS task interrupt level are ignored.

### 5.4.4 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables Critical Interrupts, External Interrupts and Machine Checks before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than RTEMS_MAXIMUM_DISABLE_PERIOD microseconds on a RTEMS_MAXIMUM_DISABLE_PERIOD_MHZ Mhz PowerPC 603e with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled is hand calculated. This calculation was last performed for Release RTEMS_RELEASE_FOR_MAXIMUM_DISABLE_PERIOD.]

If a PowerPC implementation provides non-maskable interrupts (NMI) which cannot be disabled, ISRs which process these interrupts MUST NEVER issue RTEMS system calls. If a directive is invoked, unpredictable results may occur due to the inability of RTEMS to protect its critical sections. However, ISRs that make no system calls may safely execute as non-maskable interrupts.

### 5.4.5 Interrupt Stack

The PowerPC architecture does not provide for a dedicated interrupt stack. Thus by default, exception handlers would execute on the stack of the RTEMS task which they interrupted. This artificially inflates the stack requirements for each task since EVERY task stack would have to include enough space to account for the worst case interrupt stack requirements in addition to it's own worst case usage. RTEMS addresses this problem on the PowerPC by providing a dedicated interrupt stack managed by software.

During system initialization, RTEMS allocates the interrupt stack from the Workspace Area. The amount of memory allocated for the interrupt stack is determined by the interrupt_stack_size field in the CPU Configuration Table. As part of processing a non-nested interrupt, RTEMS will switch to the interrupt stack before invoking the installed handler.

## 5.5 Default Fatal Error Processing

Upon detection of a fatal error by either the application or RTEMS the fatal error manager is invoked. The fatal error manager will invoke the user-supplied fatal error handlers. If no user-supplied handlers are configured, the RTEMS provided default fatal error handler is invoked. If the user-supplied fatal error handlers return to the executive the default fatal error handler is then invoked. This chapter describes the precise operations of the default fatal error handler.

### 5.5.1 Default Fatal Error Handler Operations

The default fatal error handler which is invoked by the `rtems_fatal_error_occurred` directive when there is no user handler configured or the user handler returns control to RTEMS. The default fatal error handler performs the following actions:

- places the error code in r3, and
- executes a trap instruction which results in a Program Exception.

If the Program Exception returns, then the following actions are performed:

- disables all processor exceptions by loading a 0 into the MSR, and
- goes into an infinite loop to simulate a halt processor instruction.

## 5.6 Board Support Packages

An RTEMS Board Support Package (BSP) must be designed to support a particular processor and target board combination. This chapter presents a discussion of PowerPC specific BSP issues. For more information on developing a BSP, refer to the chapter titled Board Support Packages in the RTEMS Applications User's Guide.

### 5.6.1 System Reset

An RTEMS based application is initiated or re-initiated when the PowerPC processor is reset. The PowerPC architecture defines a Reset Exception, but leaves the details of the CPU state as implementation specific. Please refer to the User's Manual for the CPU model in question.

In general, at power-up the PowerPC begin execution at address 0xFFF00100 in supervisor mode with all exceptions disabled. For soft resets, the CPU will vector to either 0xFFF00100 or 0x00000100 depending upon the setting of the Exception Prefix bit in the MSR. If during a soft reset, a Machine Check Exception occurs, then the CPU may execute a hard reset.

### 5.6.2 Processor Initialization

It is the responsibility of the application's initialization code to initialize the CPU and board to a quiescent state before invoking the `rtems_initialize_executive` directive. It is recommended that the BSP utilize the `predriver_hook` to install default handlers for all exceptions. These default handlers may be overwritten as various device drivers and subsystems install their own exception handlers. Upon completion of RTEMS executive initialization, all interrupts are enabled.

If this PowerPC implementation supports on-chip caching and this is to be utilized, then it should be enabled during the reset application initialization code. On-chip caching has

been observed to prevent some emulators from working properly, so it may be necessary to run with caching disabled to use these emulators.

In addition to the requirements described in the **Board Support Packages** chapter of the RTEMS C Applications User's Manual for the reset code which is executed before the call to `rtems_initialize_executive`, the PowrePC version has the following specific requirements:

- Must leave the PR bit of the Machine State Register (MSR) set to 0 so the PowerPC remains in the supervisor state.
- Must set stack pointer (sp or r1) such that a minimum stack size of MINIMUM_STACK_SIZE bytes is provided for the `rtems_initialize_executive` directive.
- Must disable all external interrupts (i.e. clear the EI (EE) bit of the machine state register).
- Must enable traps so window overflow and underflow conditions can be properly handled.
- Must initialize the PowerPC's initial Exception Table with default handlers.

## 5.7 Processor Dependent Information Table

Any highly processor dependent information required to describe a processor to RTEMS is provided in the CPU Dependent Information Table. This table is not required for all processors supported by RTEMS. This chapter describes the contents, if any, for a particular processor type.

### 5.7.1 CPU Dependent Information Table

The PowerPC version of the RTEMS CPU Dependent Information Table is given by the C structure definition is shown below:

```
typedef struct {
  void       (*pretasking_hook)( void );
  void       (*predriver_hook)( void );
  void       (*postdriver_hook)( void );
  void       (*idle_task)( void );
  boolean      do_zero_of_workspace;
  unsigned32   idle_task_stack_size;
  unsigned32   interrupt_stack_size;
  unsigned32   extra_mpci_receive_server_stack;
  void *     (*stack_allocate_hook)( unsigned32 );
  void       (*stack_free_hook)( void* );
  /* end of fields required on all CPUs */

  unsigned32   clicks_per_usec;       /* Timer clicks per microsecond */
  void       (*spurious_handler)(
                unsigned32 vector, CPU_Interrupt_frame *);
  boolean      exceptions_in_RAM;     /* TRUE if in RAM */
```

```
#if defined(ppc403)
  unsigned32   serial_per_sec;        /* Serial clocks per second */
  boolean      serial_external_clock;
  boolean      serial_xon_xoff;
  boolean      serial_cts_rts;
  unsigned32   serial_rate;
  unsigned32   timer_average_overhead; /* in ticks */
  unsigned32   timer_least_valid;   /* Least valid number from timer */
#endif
};
```

`pretasking_hook` is the address of the user provided routine which is invoked once RTEMS APIs are initialized. This routine will be invoked before any system tasks are created. Interrupts are disabled. This field may be NULL to indicate that the hook is not utilized.

`predriver_hook` is the address of the user provided routine that is invoked immediately before the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`postdriver_hook` is the address of the user provided routine that is invoked immediately after the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`idle_task` is the address of the optional user provided routine which is used as the system's IDLE task. If this field is not NULL, then the RTEMS default IDLE task is not used. This field may be NULL to indicate that the default IDLE is to be used.

`do_zero_of_workspace`
indicates whether RTEMS should zero the Workspace as part of its initialization. If set to TRUE, the Workspace is zeroed. Otherwise, it is not.

`idle_task_stack_size`
is the size of the RTEMS idle task stack in bytes. If this number is less than MINIMUM_STACK_SIZE, then the idle task's stack will be MINIMUM_STACK_SIZE in byte.

`interrupt_stack_size`
is the size of the RTEMS allocated interrupt stack in bytes. This value must be at least as large as MINIMUM_STACK_SIZE.

`extra_mpci_receive_server_stack`
is the extra stack space allocated for the RTEMS MPCI receive server task in bytes. The MPCI receive server may invoke nearly all directives and may require extra stack space on some targets.

`stack_allocate_hook`
: is the address of the optional user provided routine which allocates memory for task stacks. If this hook is not NULL, then a stack_free_hook must be provided as well.

`stack_free_hook`
: is the address of the optional user provided routine which frees memory for task stacks. If this hook is not NULL, then a stack_allocate_hook must be provided as well.

`clicks_per_usec`
: is the number of decrementer interupts that occur each microsecond.

`spurious_handler`
: is the address of the routine which is invoked when a spurious interrupt occurs.

`exceptions_in_RAM`
: indicates whether the exception vectors are located in RAM or ROM. If they are located in RAM dynamic vector installation occurs, otherwise it does not.

`serial_per_sec`
: is a PPC403 specific field which specifies the number of clock ticks per second for the PPC403 serial timer.

`serial_rate`
: is a PPC403 specific field which specifies the baud rate for the PPC403 serial port.

`serial_external_clock`
: is a PPC403 specific field which indicates whether or not to mask in a 0x2 into the Input/Output Configuration Register (IOCR) during initialization of the PPC403 console. (NOTE: This bit is defined as "reserved" 6-12?)

`serial_xon_xoff`
: is a PPC403 specific field which indicates whether or not XON/XOFF flow control is supported for the PPC403 serial port.

`serial_cts_rts`
: is a PPC403 specific field which indicates whether or not to set the least significant bit of the Input/Output Configuration Register (IOCR) during initialization of the PPC403 console. (NOTE: This bit is defined as "reserved" 6-12?)

`timer_average_overhead`
: is a PPC403 specific field which specifies the average number of overhead ticks that occur on the PPC403 timer.

`timer_least_valid`
: is a PPC403 specific field which specifies the maximum valid PPC403 timer value.

# 6 SuperH Specific Information

The Real Time Executive for Multiprocessor Systems (RTEMS) is designed to be portable across multiple processor architectures. However, the nature of real-time systems makes it essential that the application designer understand certain processor dependent implementation details. These processor dependencies include calling convention, board support package issues, interrupt processing, exact RTEMS memory requirements, performance data, header files, and the assembly language interface to the executive.

This document discusses the VENDOR XXX architecture dependencies in this port of RTEMS. The XXX family has a wide variety of CPU models within it. The part numbers ...

XXX fill in some things here

It is highly recommended that the XXX RTEMS application developer obtain and become familiar with the documentation for the processor being used as well as the documentation for the family as a whole.

### Architecture Documents

For information on the XXX architecture, refer to the following documents available from VENDOR (‘`http//www.XXX.com/`’):

- *XXX Family Reference, VENDOR, PART NUMBER.*

### MODEL SPECIFIC DOCUMENTS

For information on specific processor models and their associated coprocessors, refer to the following documents:

- *XXX MODEL Manual, VENDOR, PART NUMBER.*
- *XXX MODEL Manual, VENDOR, PART NUMBER.*

## 6.1 CPU Model Dependent Features

Microprocessors are generally classified into families with a variety of CPU models or implementations within that family. Within a processor family, there is a high level of binary compatibility. This family may be based on either an architectural specification or on maintaining compatibility with a popular processor. Recent microprocessor families such as the SPARC or PowerPC are based on an architectural specification which is independent or any particular CPU model or implementation. Older families such as the M68xxx and the iX86 evolved as the manufacturer strived to produce higher performance processor models which maintained binary compatibility with older models.

RTEMS takes advantage of the similarity of the various models within a CPU family. Although the models do vary in significant ways, the high level of compatibility makes it possible to share the bulk of the CPU dependent executive code across the entire family. Each processor family supported by RTEMS has a list of features which vary between CPU models within a family. For example, the most common model dependent feature regardless of CPU family is the presence or absence of a floating point unit or coprocessor. When defining the list of features present on a particular CPU model, one simply notes

that floating point hardware is or is not present and defines a single constant appropriately. Conditional compilation is utilized to include the appropriate source code for this CPU model's feature set. It is important to note that this means that RTEMS is thus compiled using the appropriate feature set and compilation flags optimal for this CPU model used. The alternative would be to generate a binary which would execute on all family members using only the features which were always present.

This chapter presents the set of features which vary across SPARC implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file cpukit/score/cpu/XXX/XXX.h based upon the particular CPU model defined on the compilation command line.

### 6.1.1 CPU Model Name

The macro CPU_MODEL_NAME is a string which designates the name of this CPU model. For example, for the MODEL processor, this macro is set to the string "XXX".

### 6.1.2 Floating Point Unit

The macro XXX_HAS_FPU is set to 1 to indicate that this CPU model has a hardware floating point unit and 0 otherwise. It does not matter whether the hardware floating point support is incorporated on-chip or is an external coprocessor.

### 6.1.3 Another Optional Feature

The macro XXX

## 6.2 Calling Conventions

Each high-level language compiler generates subroutine entry and exit code based upon a set of rules known as the compiler's calling convention. These rules address the following issues:

- register preservation and usage
- parameter passing
- call and return mechanism

A compiler's calling convention is of importance when interfacing to subroutines written in another language either assembly or high-level. Even when the high-level language and target processor are the same, different compilers may use different calling conventions. As a result, calling conventions are both processor and compiler dependent.

The Hitachi SH architecture supports a simple yet effective call and return mechanism. A subroutine is invoked via the branch to subroutine (XXX) or the jump to subroutine (XXX) instructions. These instructions push the return address on the current stack. The return from subroutine (rts) instruction pops the return address off the current stack and transfers control to that instruction. It is is important to note that the MC68xxx call and return mechanism does not automatically save or restore any registers. It is the responsibility of the high-level language compiler to define the register preservation and usage convention.

### 6.2.1 Calling Mechanism

All RTEMS directives are invoked using either a bsr or jsr instruction and return to the user application via the rts instruction.

### 6.2.2 Register Usage

As discussed above, the bsr and jsr instructions do not automatically save any registers. RTEMS uses the registers D0, D1, A0, and A1 as scratch registers. These registers are not preserved by RTEMS directives therefore, the contents of these registers should not be assumed upon return from any RTEMS directive.

> > The SH1 has 16 general registers (r0..r15) > > r0..r3 used as general volatile registers > > r4..r7 used to pass up to 4 arguments to functions, arguments above 4 are > > passed via the stack) > > r8..13 caller saved registers (i.e. push them to the stack if you need them > > inside of a function) > > r14 frame pointer > > r15 stack pointer >

### 6.2.3 Parameter Passing

RTEMS assumes that arguments are placed on the current stack before the directive is invoked via the bsr or jsr instruction. The first argument is assumed to be closest to the return address on the stack. This means that the first argument of the C calling sequence is pushed last. The following pseudo-code illustrates the typical sequence used to call a RTEMS directive with three (3) arguments:

```
push third argument
push second argument
push first argument
invoke directive
remove arguments from the stack
```

The arguments to RTEMS are typically pushed onto the stack using a move instruction with a pre-decremented stack pointer as the destination. These arguments must be removed from the stack after control is returned to the caller. This removal is typically accomplished by adding the size of the argument list in bytes to the current stack pointer.

### 6.2.4 User-Provided Routines

All user-provided routines invoked by RTEMS, such as user extensions, device drivers, and MPCI routines, must also adhere to these calling conventions.

## 6.3 Memory Model

A processor may support any combination of memory models ranging from pure physical addressing to complex demand paged virtual memory systems. RTEMS supports a flat memory model which ranges contiguously over the processor's allowable address space. RTEMS does not support segmentation or virtual memory of any kind. The appropriate memory model for RTEMS provided by the targeted processor and related characteristics of that model are described in this chapter.

### 6.3.1 Flat Memory Model

The XXX family supports a flat 32-bit address space with addresses ranging from 0x00000000 to 0xFFFFFFFF (4 gigabytes). Each address is represented by a 32-bit value and is byte addressable. The address may be used to reference a single byte, word (2-bytes), or long word (4 bytes). Memory accesses within this address space are performed in big endian fashion by the processors in this family.

Some of the XXX family members such as the XXX, XXX, and XXX support virtual memory and segmentation. The XXX requires external hardware support such as the XXX Paged Memory Management Unit coprocessor which is typically used to perform address translations for these systems. RTEMS does not support virtual memory or segmentation on any of the XXX family members.

## 6.4 Interrupt Processing

Different types of processors respond to the occurrence of an interrupt in its own unique fashion. In addition, each processor type provides a control mechanism to allow for the proper handling of an interrupt. The processor dependent response to the interrupt modifies the current execution state and results in a change in the execution stream. Most processors require that an interrupt handler utilize some special control mechanisms to return to the normal processing stream. Although RTEMS hides many of the processor dependent details of interrupt processing, it is important to understand how the RTEMS interrupt manager is mapped onto the processor's unique architecture. Discussed in this chapter are the SH's interrupt response and control mechanisms as they pertain to RTEMS.

### 6.4.1 Vectoring of an Interrupt Handler

Depending on whether or not the particular CPU supports a separate interrupt stack, the SH family has two different interrupt handling models.

#### 6.4.1.1 Models Without Separate Interrupt Stacks

Upon receipt of an interrupt the SH family members without separate interrupt stacks automatically perform the following actions:

- To Be Written

#### 6.4.1.2 Models With Separate Interrupt Stacks

Upon receipt of an interrupt the SH family members with separate interrupt stacks automatically perform the following actions:

- saves the current status register (SR),
- clears the master/interrupt (M) bit of the SR to indicate the switch from master state to interrupt state,
- sets the privilege mode to supervisor,
- suppresses tracing,
- sets the interrupt mask level equal to the level of the interrupt being serviced,
- pushes an interrupt stack frame (ISF), which includes the program counter (PC), the status register (SR), and the format/exception vector offset (FVO) word, onto the supervisor and interrupt stacks,

- switches the current stack to the interrupt stack and vectors to an interrupt service routine (ISR). If the ISR was installed with the interrupt_catch directive, then the RTEMS interrupt handler will begin execution. The RTEMS interrupt handler saves all registers which are not preserved according to the calling conventions and invokes the application's ISR.

A nested interrupt is processed similarly by these CPU models with the exception that only a single ISF is placed on the interrupt stack and the current stack need not be switched.

The FVO word in the Interrupt Stack Frame is examined by RTEMS to determine when an outer most interrupt is being exited. Since the FVO is used by RTEMS for this purpose, the user application code MUST NOT modify this field.

The following shows the Interrupt Stack Frame for XXX CPU models with separate interrupt stacks:

| | |
|---|---|
| Status Register | 0x0 |
| Program Counter High | 0x2 |
| Program Counter Low | 0x4 |
| Format/Vector Offset | 0x6 |

### 6.4.2 Interrupt Levels

Eight levels (0-7) of interrupt priorities are supported by XXX family members with level seven (7) being the highest priority. Level zero (0) indicates that interrupts are fully enabled. Interrupt requests for interrupts with priorities less than or equal to the current interrupt mask level are ignored.

Although RTEMS supports 256 interrupt levels, the XXX family only supports eight. RTEMS interrupt levels 0 through 7 directly correspond to XXX interrupt levels. All other RTEMS interrupt levels are undefined and their behavior is unpredictable.

### 6.4.3 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables interrupts to level seven (7) before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than RTEMS_MAXIMUM_DISABLE_PERIOD microseconds on a RTEMS_MAXIMUM_DISABLE_PERIOD_MHZ Mhz XXX with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled is hand calculated. This calculation was last performed for Release RTEMS_RELEASE_FOR_MAXIMUM_DISABLE_PERIOD.]

Non-maskable interrupts (NMI) cannot be disabled, and ISRs which execute at this level MUST NEVER issue RTEMS system calls. If a directive is invoked, unpredictable results may occur due to the inability of RTEMS to protect its critical sections. However, ISRs that make no system calls may safely execute as non-maskable interrupts.

### 6.4.4 Interrupt Stack

RTEMS allocates the interrupt stack from the Workspace Area. The amount of memory allocated for the interrupt stack is determined by the interrupt_stack_size field in the CPU Configuration Table. During the initialization process, RTEMS will install its interrupt stack.

The XXX port of RTEMS supports a software managed dedicated interrupt stack on those CPU models which do not support a separate interrupt stack in hardware.

## 6.5 Default Fatal Error Processing

Upon detection of a fatal error by either the application or RTEMS the fatal error manager is invoked. The fatal error manager will invoke the user-supplied fatal error handlers. If no user-supplied handlers are configured, the RTEMS provided default fatal error handler is invoked. If the user-supplied fatal error handlers return to the executive the default fatal error handler is then invoked. This chapter describes the precise operations of the default fatal error handler.

### 6.5.1 Default Fatal Error Handler Operations

The default fatal error handler which is invoked by the `rtems_fatal_error_occurred` directive when there is no user handler configured or the user handler returns control to RTEMS. The default fatal error handler disables processor interrupts, places the error code in **XXX**, and executes a `XXX` instruction to simulate a halt processor instruction.

## 6.6 Board Support Packages

An RTEMS Board Support Package (BSP) must be designed to support a particular processor and target board combination. This chapter presents a discussion of XXX specific BSP issues. For more information on developing a BSP, refer to the chapter titled Board Support Packages in the RTEMS Applications User's Guide.

### 6.6.1 System Reset

An RTEMS based application is initiated or re-initiated when the XXX processor is reset. When the XXX is reset, the processor performs the following actions:

- The tracing bits of the status register are cleared to disable tracing.
- The supervisor interrupt state is entered by setting the supervisor (S) bit and clearing the master/interrupt (M) bit of the status register.
- The interrupt mask of the status register is set to level 7 to effectively disable all maskable interrupts.
- The vector base register (VBR) is set to zero.
- The cache control register (CACR) is set to zero to disable and freeze the processor cache.
- The interrupt stack pointer (ISP) is set to the value stored at vector 0 (bytes 0-3) of the exception vector table (EVT).
- The program counter (PC) is set to the value stored at vector 1 (bytes 4-7) of the EVT.

- The processor begins execution at the address stored in the PC.

### 6.6.2 Processor Initialization

The address of the application's initialization code should be stored in the first vector of the EVT which will allow the immediate vectoring to the application code. If the application requires that the VBR be some value besides zero, then it should be set to the required value at this point. All tasks share the same XXX's VBR value. Because interrupts are enabled automatically by RTEMS as part of the initialize executive directive, the VBR MUST be set before this directive is invoked to insure correct interrupt vectoring. If processor caching is to be utilized, then it should be enabled during the reset application initialization code.

In addition to the requirements described in the Board Support Packages chapter of the Applications User's Manual for the reset code which is executed before the call to initialize executive, the XXX version has the following specific requirements:

- Must leave the S bit of the status register set so that the XXX remains in the supervisor state.
- Must set the M bit of the status register to remove the XXX from the interrupt state.
- Must set the master stack pointer (MSP) such that a minimum stack size of MINIMUM_STACK_SIZE bytes is provided for the initialize executive directive.
- Must initialize the XXX's vector table.

Note that the BSP is not responsible for allocating or installing the interrupt stack. RTEMS does this automatically as part of initialization. If the BSP does not install an interrupt stack and – for whatever reason – an interrupt occurs before initialize_executive is invoked, then the results are unpredictable.

## 6.7 Processor Dependent Information Table

Any highly processor dependent information required to describe a processor to RTEMS is provided in the CPU Dependent Information Table. This table is not required for all processors supported by RTEMS. This chapter describes the contents, if any, for a particular processor type.

### 6.7.1 CPU Dependent Information Table

The XXX version of the RTEMS CPU Dependent Information Table contains the information required to interface a Board Support Package and RTEMS on the XXX. This information is provided to allow RTEMS to interoperate effectively with the BSP. The C structure definition is given here:

```
typedef struct {
  void       (*pretasking_hook)( void );
  void       (*predriver_hook)( void );
  void       (*postdriver_hook)( void );
  void       (*idle_task)( void );
  boolean      do_zero_of_workspace;
  unsigned32   idle_task_stack_size;
  unsigned32   interrupt_stack_size;
  unsigned32   extra_mpci_receive_server_stack;
  void *     (*stack_allocate_hook)( unsigned32 );
  void       (*stack_free_hook)( void* );
  /* end of fields required on all CPUs */

  /* XXX CPU family dependent stuff */
} rtems_cpu_table;
```

`pretasking_hook` is the address of the user provided routine which is invoked once RTEMS APIs are initialized. This routine will be invoked before any system tasks are created. Interrupts are disabled. This field may be NULL to indicate that the hook is not utilized.

`predriver_hook` is the address of the user provided routine that is invoked immediately before the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`postdriver_hook` is the address of the user provided routine that is invoked immediately after the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`idle_task` is the address of the optional user provided routine which is used as the system's IDLE task. If this field is not NULL, then the RTEMS default IDLE task is not used. This field may be NULL to indicate that the default IDLE is to be used.

`do_zero_of_workspace`
indicates whether RTEMS should zero the Workspace as part of its initialization. If set to TRUE, the Workspace is zeroed. Otherwise, it is not.

`idle_task_stack_size`
is the size of the RTEMS idle task stack in bytes. If this number is less than MINIMUM_STACK_SIZE, then the idle task's stack will be MINIMUM_STACK_SIZE in byte.

`interrupt_stack_size`
is the size of the RTEMS allocated interrupt stack in bytes. This value must be at least as large as MINIMUM_STACK_SIZE.

`extra_mpci_receive_server_stack`
: is the extra stack space allocated for the RTEMS MPCI receive server task in bytes. The MPCI receive server may invoke nearly all directives and may require extra stack space on some targets.

`stack_allocate_hook`
: is the address of the optional user provided routine which allocates memory for task stacks. If this hook is not NULL, then a stack_free_hook must be provided as well.

`stack_free_hook`
: is the address of the optional user provided routine which frees memory for task stacks. If this hook is not NULL, then a stack_allocate_hook must be provided as well.

`XXX`
: is where the CPU family dependent stuff goes.

# 7 SPARC Specific Information

The Real Time Executive for Multiprocessor Systems (RTEMS) is designed to be portable across multiple processor architectures. However, the nature of real-time systems makes it essential that the application designer understand certain processor dependent implementation details. These processor dependencies include calling convention, board support package issues, interrupt processing, exact RTEMS memory requirements, performance data, header files, and the assembly language interface to the executive.

This document discusses the SPARC architecture dependencies in this port of RTEMS. Currently, only implementations of SPARC Version 7 are supported by RTEMS.

It is highly recommended that the SPARC RTEMS application developer obtain and become familiar with the documentation for the processor being used as well as the specification for the revision of the SPARC architecture which corresponds to that processor.

## SPARC Architecture Documents

For information on the SPARC architecture, refer to the following documents available from SPARC International, Inc. (http://www.sparc.com):

- SPARC Standard Version 7.
- SPARC Standard Version 8.
- SPARC Standard Version 9.

## ERC32 Specific Information

The European Space Agency's ERC32 is a three chip computing core implementing a SPARC V7 processor and associated support circuitry for embedded space applications. The integer and floating-point units (90C601E & 90C602E) are based on the Cypress 7C601 and 7C602, with additional error-detection and recovery functions. The memory controller (MEC) implements system support functions such as address decoding, memory interface, DMA interface, UARTs, timers, interrupt control, write-protection, memory reconfiguration and error-detection. The core is designed to work at 25MHz, but using space qualified memories limits the system frequency to around 15 MHz, resulting in a performance of 10 MIPS and 2 MFLOPS.

Information on the ERC32 and a number of development support tools, such as the SPARC Instruction Simulator (SIS), are freely available on the Internet. The following documents and SIS are available via anonymous ftp or pointing your web browser at ftp://ftp.estec.esa.nl/pub/ws/wsd/erc32.

- ERC32 System Design Document
- MEC Device Specification

Additionally, the SPARC RISC User's Guide from Matra MHS documents the functionality of the integer and floating point units including the instruction set information. To obtain this document as well as ERC32 components and VHDL models contact:

```
Matra MHS SA
3 Avenue du Centre, BP 309,
```

78054 St-Quentin-en-Yvelines,
Cedex, France
VOICE: +31-1-30607087
FAX: +31-1-30640693

Amar Guennon (amar.guennon@matramhs.fr) is familiar with the ERC32.

## 7.1 CPU Model Dependent Features

Microprocessors are generally classified into families with a variety of CPU models or implementations within that family. Within a processor family, there is a high level of binary compatibility. This family may be based on either an architectural specification or on maintaining compatibility with a popular processor. Recent microprocessor families such as the SPARC or PowerPC are based on an architectural specification which is independent or any particular CPU model or implementation. Older families such as the M68xxx and the iX86 evolved as the manufacturer strived to produce higher performance processor models which maintained binary compatibility with older models.

RTEMS takes advantage of the similarity of the various models within a CPU family. Although the models do vary in significant ways, the high level of compatibility makes it possible to share the bulk of the CPU dependent executive code across the entire family.

### 7.1.1 CPU Model Feature Flags

Each processor family supported by RTEMS has a list of features which vary between CPU models within a family. For example, the most common model dependent feature regardless of CPU family is the presence or absence of a floating point unit or coprocessor. When defining the list of features present on a particular CPU model, one simply notes that floating point hardware is or is not present and defines a single constant appropriately. Conditional compilation is utilized to include the appropriate source code for this CPU model's feature set. It is important to note that this means that RTEMS is thus compiled using the appropriate feature set and compilation flags optimal for this CPU model used. The alternative would be to generate a binary which would execute on all family members using only the features which were always present.

This section presents the set of features which vary across SPARC implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file cpukit/score/cpu/sparc/sparc.h based upon the particular CPU model defined on the compilation command line.

#### 7.1.1.1 CPU Model Name

The macro CPU_MODEL_NAME is a string which designates the name of this CPU model. For example, for the European Space Agency's ERC32 SPARC model, this macro is set to the string "erc32".

#### 7.1.1.2 Floating Point Unit

The macro SPARC_HAS_FPU is set to 1 to indicate that this CPU model has a hardware floating point unit and 0 otherwise.

#### 7.1.1.3 Bitscan Instruction

The macro SPARC_HAS_BITSCAN is set to 1 to indicate that this CPU model has the bitscan instruction. For example, this instruction is supported by the Fujitsu SPARClite family.

#### 7.1.1.4 Number of Register Windows

The macro SPARC_NUMBER_OF_REGISTER_WINDOWS is set to indicate the number of register window sets implemented by this CPU model. The SPARC architecture allows a for a maximum of thirty-two register window sets although most implementations only include eight.

#### 7.1.1.5 Low Power Mode

The macro SPARC_HAS_LOW_POWER_MODE is set to one to indicate that this CPU model has a low power mode. If low power is enabled, then there must be CPU model specific implementation of the IDLE task in cpukit/score/cpu/sparc/cpu.c. The low power mode IDLE task should be of the form:

```
while ( TRUE ) {
  enter low power mode
}
```

The code required to enter low power mode is CPU model specific.

### 7.1.2 CPU Model Implementation Notes

The ERC32 is a custom SPARC V7 implementation based on the Cypress 601/602 chipset. This CPU has a number of on-board peripherals and was developed by the European Space Agency to target space applications. RTEMS currently provides support for the following peripherals:

- UART Channels A and B
- General Purpose Timer
- Real Time Clock
- Watchdog Timer (so it can be disabled)
- Control Register (so powerdown mode can be enabled)
- Memory Control Register
- Interrupt Control

The General Purpose Timer and Real Time Clock Timer provided with the ERC32 share the Timer Control Register. Because the Timer Control Register is write only, we must mirror it in software and insure that writes to one timer do not alter the current settings and status of the other timer. Routines are provided in erc32.h which promote the view that the two timers are completely independent. By exclusively using these routines to access the Timer Control Register, the application can view the system as having a General Purpose Timer Control Register and a Real Time Clock Timer Control Register rather than the single shared value.

The RTEMS Idle thread take advantage of the low power mode provided by the ERC32. Low power mode is entered during idle loops and is enabled at initialization time.

## 7.2 Calling Conventions

Each high-level language compiler generates subroutine entry and exit code based upon a set of rules known as the compiler's calling convention. These rules address the following issues:

- register preservation and usage
- parameter passing
- call and return mechanism

A compiler's calling convention is of importance when interfacing to subroutines written in another language either assembly or high-level. Even when the high-level language and target processor are the same, different compilers may use different calling conventions. As a result, calling conventions are both processor and compiler dependent.

### 7.2.1 Programming Model

This section discusses the programming model for the SPARC architecture.

#### 7.2.1.1 Non-Floating Point Registers

The SPARC architecture defines thirty-two non-floating point registers directly visible to the programmer. These are divided into four sets:

- input registers
- local registers
- output registers
- global registers

Each register is referred to by either two or three names in the SPARC reference manuals. First, the registers are referred to as r0 through r31 or with the alternate notation r[0] through r[31]. Second, each register is a member of one of the four sets listed above. Finally, some registers have an architecturally defined role in the programming model which provides an alternate name. The following table describes the mapping between the 32 registers and the register sets:

| Register Number | Register Names | Description |
|---|---|---|
| 0 - 7 | g0 - g7 | Global Registers |
| 8 - 15 | o0 - o7 | Output Registers |
| 16 - 23 | l0 - l7 | Local Registers |
| 24 - 31 | i0 - i7 | Input Registers |

As mentioned above, some of the registers serve defined roles in the programming model. The following table describes the role of each of these registers:

| Register Name | Alternate Names | Description |
|---|---|---|
| g0 | NA | reads return 0; writes are ignored |
| o6 | sp | stack pointer |
| i6 | fp | frame pointer |
| i7 | NA | return address |

#### 7.2.1.2 Floating Point Registers

The SPARC V7 architecture includes thirty-two, thirty-two bit registers. These registers may be viewed as follows:

- 32 single precision floating point or integer registers (f0, f1, ... f31)
- 16 double precision floating point registers (f0, f2, f4, ... f30)
- 8 extended precision floating point registers (f0, f4, f8, ... f28)

The floating point status register (fpsr) specifies the behavior of the floating point unit for rounding, contains its condition codes, version specification, and trap information.

A queue of the floating point instructions which have started execution but not yet completed is maintained. This queue is needed to support the multiple cycle nature of floating point operations and to aid floating point exception trap handlers. Once a floating point exception has been encountered, the queue is frozen until it is emptied by the trap handler. The floating point queue is loaded by launching instructions. It is emptied normally when the floating point completes all outstanding instructions and by floating point exception handlers with the store double floating point queue (stdfq) instruction.

#### 7.2.1.3 Special Registers

The SPARC architecture includes two special registers which are critical to the programming model: the Processor State Register (psr) and the Window Invalid Mask (wim). The psr contains the condition codes, processor interrupt level, trap enable bit, supervisor mode and previous supervisor mode bits, version information, floating point unit and coprocessor enable bits, and the current window pointer (cwp). The cwp field of the psr and wim register are used to manage the register windows in the SPARC architecture. The register windows are discussed in more detail below.

### 7.2.2 Register Windows

The SPARC architecture includes the concept of register windows. An overly simplistic way to think of these windows is to imagine them as being an infinite supply of "fresh" register sets available for each subroutine to use. In reality, they are much more complicated.

The save instruction is used to obtain a new register window. This instruction decrements the current window pointer, thus providing a new set of registers for use. This register set includes eight fresh local registers for use exclusively by this subroutine. When done with a register set, the restore instruction increments the current window pointer and the previous register set is once again available.

The two primary issues complicating the use of register windows are that (1) the set of register windows is finite, and (2) some registers are shared between adjacent registers windows.

Because the set of register windows is finite, it is possible to execute enough save instructions without corresponding restore's to consume all of the register windows. This is easily accomplished in a high level language because each subroutine typically performs a save instruction upon entry. Thus having a subroutine call depth greater than the number of register windows will result in a window overflow condition. The window overflow condition generates a trap which must be handled in software. The window overflow trap handler is responsible for saving the contents of the oldest register window on the program stack.

Similarly, the subroutines will eventually complete and begin to perform restore's. If the restore results in the need for a register window which has previously been written to memory as part of an overflow, then a window underflow condition results. Just like the window overflow, the window underflow condition must be handled in software by a trap handler. The window underflow trap handler is responsible for reloading the contents of the register window requested by the restore instruction from the program stack.

The Window Invalid Mask (wim) and the Current Window Pointer (cwp) field in the psr are used in conjunction to manage the finite set of register windows and detect the window overflow and underflow conditions. The cwp contains the index of the register window currently in use. The save instruction decrements the cwp modulo the number of register windows. Similarly, the restore instruction increments the cwp modulo the number of register windows. Each bit in the wim represents represents whether a register window contains valid information. The value of 0 indicates the register window is valid and 1 indicates it is invalid. When a save instruction causes the cwp to point to a register window which is marked as invalid, a window overflow condition results. Conversely, the restore instruction may result in a window underflow condition.

Other than the assumption that a register window is always available for trap (i.e. interrupt) handlers, the SPARC architecture places no limits on the number of register windows simultaneously marked as invalid (i.e. number of bits set in the wim). However, RTEMS assumes that only one register window is marked invalid at a time (i.e. only one bit set in the wim). This makes the maximum possible number of register windows available to the user while still meeting the requirement that window overflow and underflow conditions can be detected.

The window overflow and window underflow trap handlers are a critical part of the run-time environment for a SPARC application. The SPARC architectural specification allows for the number of register windows to be any power of two less than or equal to 32. The most common choice for SPARC implementations appears to be 8 register windows. This results in the cwp ranging in value from 0 to 7 on most implementations.

The second complicating factor is the sharing of registers between adjacent register windows. While each register window has its own set of local registers, the input and output registers are shared between adjacent windows. The output registers for register window N are the same as the input registers for register window ((N - 1) modulo RW) where RW is the number of register windows. An alternative way to think of this is to remember how parameters are passed to a subroutine on the SPARC. The caller loads values into what are its output registers. Then after the callee executes a save instruction, those parameters are available in its input registers. This is a very efficient way to pass parameters as no data is actually moved by the save or restore instructions.

### 7.2.3 Call and Return Mechanism

The SPARC architecture supports a simple yet effective call and return mechanism. A subroutine is invoked via the call (call) instruction. This instruction places the return address in the caller's output register 7 (o7). After the callee executes a save instruction, this value is available in input register 7 (i7) until the corresponding restore instruction is executed.

The callee returns to the caller via a jmp to the return address. There is a delay slot following this instruction which is commonly used to execute a restore instruction – if a register window was allocated by this subroutine.

It is important to note that the SPARC subroutine call and return mechanism does not automatically save and restore any registers. This is accomplished via the save and restore instructions which manage the set of registers windows.

### 7.2.4 Calling Mechanism

All RTEMS directives are invoked using the regular SPARC calling convention via the call instruction.

### 7.2.5 Register Usage

As discussed above, the call instruction does not automatically save any registers. The save and restore instructions are used to allocate and deallocate register windows. When a register window is allocated, the new set of local registers are available for the exclusive use of the subroutine which allocated this register set.

### 7.2.6 Parameter Passing

RTEMS assumes that arguments are placed in the caller's output registers with the first argument in output register 0 (o0), the second argument in output register 1 (o1), and so forth. Until the callee executes a save instruction, the parameters are still visible in the output registers. After the callee executes a save instruction, the parameters are visible in the corresponding input registers. The following pseudo-code illustrates the typical sequence used to call a RTEMS directive with three (3) arguments:

```
load third argument into o2
load second argument into o1
load first argument into o0
invoke directive
```

### 7.2.7 User-Provided Routines

All user-provided routines invoked by RTEMS, such as user extensions, device drivers, and MPCI routines, must also adhere to these calling conventions.

## 7.3 Memory Model

A processor may support any combination of memory models ranging from pure physical addressing to complex demand paged virtual memory systems. RTEMS supports a flat memory model which ranges contiguously over the processor's allowable address space. RTEMS does not support segmentation or virtual memory of any kind. The appropriate

memory model for RTEMS provided by the targeted processor and related characteristics of that model are described in this chapter.

### 7.3.1 Flat Memory Model

The SPARC architecture supports a flat 32-bit address space with addresses ranging from 0x00000000 to 0xFFFFFFFF (4 gigabytes). Each address is represented by a 32-bit value and is byte addressable. The address may be used to reference a single byte, half-word (2-bytes), word (4 bytes), or doubleword (8 bytes). Memory accesses within this address space are performed in big endian fashion by the SPARC. Memory accesses which are not properly aligned generate a "memory address not aligned" trap (type number 7). The following table lists the alignment requirements for a variety of data accesses:

| Data Type | Alignment Requirement |
|---|---|
| byte | 1 |
| half-word | 2 |
| word | 4 |
| doubleword | 8 |

Doubleword load and store operations must use a pair of registers as their source or destination. This pair of registers must be an adjacent pair of registers with the first of the pair being even numbered. For example, a valid destination for a doubleword load might be input registers 0 and 1 (i0 and i1). The pair i1 and i2 would be invalid. [NOTE: Some assemblers for the SPARC do not generate an error if an odd numbered register is specified as the beginning register of the pair. In this case, the assembler assumes that what the programmer meant was to use the even-odd pair which ends at the specified register. This may or may not have been a correct assumption.]

RTEMS does not support any SPARC Memory Management Units, therefore, virtual memory or segmentation systems involving the SPARC are not supported.

## 7.4 Interrupt Processing

Different types of processors respond to the occurrence of an interrupt in its own unique fashion. In addition, each processor type provides a control mechanism to allow for the proper handling of an interrupt. The processor dependent response to the interrupt modifies the current execution state and results in a change in the execution stream. Most processors require that an interrupt handler utilize some special control mechanisms to return to the normal processing stream. Although RTEMS hides many of the processor dependent details of interrupt processing, it is important to understand how the RTEMS interrupt manager is mapped onto the processor's unique architecture. Discussed in this chapter are the SPARC's interrupt response and control mechanisms as they pertain to RTEMS.

RTEMS and associated documentation uses the terms interrupt and vector. In the SPARC architecture, these terms correspond to traps and trap type, respectively. The terms will be used interchangeably in this manual.

### 7.4.1 Synchronous Versus Asynchronous Traps

The SPARC architecture includes two classes of traps: synchronous and asynchronous. Asynchronous traps occur when an external event interrupts the processor. These traps are not associated with any instruction executed by the processor and logically occur between instructions. The instruction currently in the execute stage of the processor is allowed to complete although subsequent instructions are annulled. The return address reported by the processor for asynchronous traps is the pair of instructions following the current instruction.

Synchronous traps are caused by the actions of an instruction. The trap stimulus in this case either occurs internally to the processor or is from an external signal that was provoked by the instruction. These traps are taken immediately and the instruction that caused the trap is aborted before any state changes occur in the processor itself. The return address reported by the processor for synchronous traps is the instruction which caused the trap and the following instruction.

### 7.4.2 Vectoring of Interrupt Handler

Upon receipt of an interrupt the SPARC automatically performs the following actions:

- disables traps (sets the ET bit of the psr to 0),
- the S bit of the psr is copied into the Previous Supervisor Mode (PS) bit of the psr,
- the cwp is decremented by one (modulo the number of register windows) to activate a trap window,
- the PC and nPC are loaded into local register 1 and 2 (l0 and l1),
- the trap type (tt) field of the Trap Base Register (TBR) is set to the appropriate value, and
- if the trap is not a reset, then the PC is written with the contents of the TBR and the nPC is written with TBR + 4. If the trap is a reset, then the PC is set to zero and the nPC is set to 4.

Trap processing on the SPARC has two features which are noticeably different than interrupt processing on other architectures. First, the value of psr register in effect immediately before the trap occurred is not explicitly saved. Instead only reversible alterations are made to it. Second, the Processor Interrupt Level (pil) is not set to correspond to that of the interrupt being processed. When a trap occurs, ALL subsequent traps are disabled. In order to safely invoke a subroutine during trap handling, traps must be enabled to allow for the possibility of register window overflow and underflow traps.

If the interrupt handler was installed as an RTEMS interrupt handler, then upon receipt of the interrupt, the processor passes control to the RTEMS interrupt handler which performs the following actions:

- saves the state of the interrupted task on it's stack,
- insures that a register window is available for subsequent traps,
- if this is the outermost (i.e. non-nested) interrupt, then the RTEMS interrupt handler switches from the current stack to the interrupt stack,
- enables traps,

- invokes the vectors to a user interrupt service routine (ISR).

Asynchronous interrupts are ignored while traps are disabled. Synchronous traps which occur while traps are disabled result in the CPU being forced into an error mode.

A nested interrupt is processed similarly with the exception that the current stack need not be switched to the interrupt stack.

### 7.4.3 Traps and Register Windows

One of the register windows must be reserved at all times for trap processing. This is critical to the proper operation of the trap mechanism in the SPARC architecture. It is the responsibility of the trap handler to insure that there is a register window available for a subsequent trap before re-enabling traps. It is likely that any high level language routines invoked by the trap handler (such as a user-provided RTEMS interrupt handler) will allocate a new register window. The save operation could result in a window overflow trap. This trap cannot be correctly processed unless (1) traps are enabled and (2) a register window is reserved for traps. Thus, the RTEMS interrupt handler insures that a register window is available for subsequent traps before enabling traps and invoking the user's interrupt handler.

### 7.4.4 Interrupt Levels

Sixteen levels (0-15) of interrupt priorities are supported by the SPARC architecture with level fifteen (15) being the highest priority. Level zero (0) indicates that interrupts are fully enabled. Interrupt requests for interrupts with priorities less than or equal to the current interrupt mask level are ignored.

Although RTEMS supports 256 interrupt levels, the SPARC only supports sixteen. RTEMS interrupt levels 0 through 15 directly correspond to SPARC processor interrupt levels. All other RTEMS interrupt levels are undefined and their behavior is unpredictable.

### 7.4.5 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables interrupts to level seven (15) before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than RTEMS_MAXIMUM_DISABLE_PERIOD microseconds on a RTEMS_MAXIMUM_DISABLE_PERIOD_MHZ Mhz ERC32 with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled is hand calculated. This calculation was last performed for Release RTEMS_RELEASE_FOR_MAXIMUM_DISABLE_PERIOD.]

[NOTE: It is thought that the length of time at which the processor interrupt level is elevated to fifteen by RTEMS is not anywhere near as long as the length of time ALL traps are disabled as part of the "flush all register windows" operation.]

Non-maskable interrupts (NMI) cannot be disabled, and ISRs which execute at this level MUST NEVER issue RTEMS system calls. If a directive is invoked, unpredictable results

may occur due to the inability of RTEMS to protect its critical sections. However, ISRs that make no system calls may safely execute as non-maskable interrupts.

### 7.4.6 Interrupt Stack

The SPARC architecture does not provide for a dedicated interrupt stack. Thus by default, trap handlers would execute on the stack of the RTEMS task which they interrupted. This artificially inflates the stack requirements for each task since EVERY task stack would have to include enough space to account for the worst case interrupt stack requirements in addition to it's own worst case usage. RTEMS addresses this problem on the SPARC by providing a dedicated interrupt stack managed by software.

During system initialization, RTEMS allocates the interrupt stack from the Workspace Area. The amount of memory allocated for the interrupt stack is determined by the interrupt_stack_size field in the CPU Configuration Table. As part of processing a non-nested interrupt, RTEMS will switch to the interrupt stack before invoking the installed handler.

## 7.5 Default Fatal Error Processing

Upon detection of a fatal error by either the application or RTEMS the fatal error manager is invoked. The fatal error manager will invoke the user-supplied fatal error handlers. If no user-supplied handlers are configured, the RTEMS provided default fatal error handler is invoked. If the user-supplied fatal error handlers return to the executive the default fatal error handler is then invoked. This chapter describes the precise operations of the default fatal error handler.

### 7.5.1 Default Fatal Error Handler Operations

The default fatal error handler which is invoked by the fatal_error_occurred directive when there is no user handler configured or the user handler returns control to RTEMS. The default fatal error handler disables processor interrupts to level 15, places the error code in g1, and goes into an infinite loop to simulate a halt processor instruction.

## 7.6 Board Support Packages

An RTEMS Board Support Package (BSP) must be designed to support a particular processor and target board combination. This chapter presents a discussion of SPARC specific BSP issues. For more information on developing a BSP, refer to the chapter titled Board Support Packages in the RTEMS Applications User's Guide.

### 7.6.1 System Reset

An RTEMS based application is initiated or re-initiated when the SPARC processor is reset. When the SPARC is reset, the processor performs the following actions:

- the enable trap (ET) of the psr is set to 0 to disable traps,
- the supervisor bit (S) of the psr is set to 1 to enter supervisor mode, and
- the PC is set 0 and the nPC is set to 4.

The processor then begins to execute the code at location 0. It is important to note that all fields in the psr are not explicitly set by the above steps and all other registers retain

their value from the previous execution mode. This is true even of the Trap Base Register (TBR) whose contents reflect the last trap which occurred before the reset.

### 7.6.2 Processor Initialization

It is the responsibility of the application's initialization code to initialize the TBR and install trap handlers for at least the register window overflow and register window underflow conditions. Traps should be enabled before invoking any subroutines to allow for register window management. However, interrupts should be disabled by setting the Processor Interrupt Level (pil) field of the psr to 15. RTEMS installs it's own Trap Table as part of initialization which is initialized with the contents of the Trap Table in place when the `rtems_initialize_executive` directive was invoked. Upon completion of executive initialization, interrupts are enabled.

If this SPARC implementation supports on-chip caching and this is to be utilized, then it should be enabled during the reset application initialization code.

In addition to the requirements described in the Board Support Packages chapter of the [No value for "LANGUAGE"] Applications User's Manual for the reset code which is executed before the call to `rtems_initialize_executive`, the SPARC version has the following specific requirements:

- Must leave the S bit of the status register set so that the SPARC remains in the supervisor state.
- Must set stack pointer (sp) such that a minimum stack size of MINIMUM_STACK_SIZE bytes is provided for the `rtems_initialize_executive` directive.
- Must disable all external interrupts (i.e. set the pil to 15).
- Must enable traps so window overflow and underflow conditions can be properly handled.
- Must initialize the SPARC's initial trap table with at least trap handlers for register window overflow and register window underflow.

## 7.7 Processor Dependent Information Table

Any highly processor dependent information required to describe a processor to RTEMS is provided in the CPU Dependent Information Table. This table is not required for all processors supported by RTEMS. This chapter describes the contents, if any, for a particular processor type.

### 7.7.1 CPU Dependent Information Table

The SPARC version of the RTEMS CPU Dependent Information Table is given by the C structure definition is shown below:

```
typedef struct {
  void       (*pretasking_hook)( void );
  void       (*predriver_hook)( void );
  void       (*postdriver_hook)( void );
  void       (*idle_task)( void );
  boolean      do_zero_of_workspace;
  unsigned32   idle_task_stack_size;
  unsigned32   interrupt_stack_size;
  unsigned32   extra_mpci_receive_server_stack;
  void *     (*stack_allocate_hook)( unsigned32 );
  void       (*stack_free_hook)( void* );
  /* end of fields required on all CPUs */

} rtems_cpu_table;
```

`pretasking_hook` is the address of the user provided routine which is invoked once RTEMS APIs are initialized. This routine will be invoked before any system tasks are created. Interrupts are disabled. This field may be NULL to indicate that the hook is not utilized.

`predriver_hook` is the address of the user provided routine that is invoked immediately before the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`postdriver_hook` is the address of the user provided routine that is invoked immediately after the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`idle_task` is the address of the optional user provided routine which is used as the system's IDLE task. If this field is not NULL, then the RTEMS default IDLE task is not used. This field may be NULL to indicate that the default IDLE is to be used.

`do_zero_of_workspace`
indicates whether RTEMS should zero the Workspace as part of its initialization. If set to TRUE, the Workspace is zeroed. Otherwise, it is not.

`idle_task_stack_size`
is the size of the RTEMS idle task stack in bytes. If this number is less than MINIMUM_STACK_SIZE, then the idle task's stack will be MINIMUM_STACK_SIZE in byte.

`interrupt_stack_size`
is the size of the RTEMS allocated interrupt stack in bytes. This value must be at least as large as MINIMUM_STACK_SIZE.

`extra_mpci_receive_server_stack`
: is the extra stack space allocated for the RTEMS MPCI receive server task in bytes. The MPCI receive server may invoke nearly all directives and may require extra stack space on some targets.

`stack_allocate_hook`
: is the address of the optional user provided routine which allocates memory for task stacks. If this hook is not NULL, then a stack_free_hook must be provided as well.

`stack_free_hook`
: is the address of the optional user provided routine which frees memory for task stacks. If this hook is not NULL, then a stack_allocate_hook must be provided as well.

# Command and Variable Index

There are currently no Command and Variable Index entries.

# Concept Index

There are currently no Concept Index entries.