RTEMS CPU Architecture Supplement

Release 6.150c2e8 (20th March 2024)

CONTENTS

The RTEMS Project is hosted at https://www.rtems.org. Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to the RTEMS Project community.

RTEMS Online Resources

Home	https://www.rtems.org
Documentation	https://docs.rtems.org
Mailing Lists	https://lists.rtems.org
Bug Reporting	https://devel.rtems.org/wiki/Developer/Bug_Reporting
Git Repositories	https://git.rtems.org
Developers	https://devel.rtems.org

CHAPTER

ONE

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 architecture represents a CPU family and usually there are a wide variety of CPU models within it. These models share a common Instruction Set Architecture (ISA) which often varies based upon some well-defined rules. There are often multiple implementations of the ISA and these may be from one or multiple vendors.

On top of variations in the ISA, there may also be variations which occur when a CPU core implementation is combined with a set of peripherals to form a system on chip. For example, there are many ARM CPU models from numerous semiconductor vendors and a wide variety of peripherals. But at the ISA level, they share a common compatibility.

RTEMS depends upon this core similarity across the CPU models and leverages that to minimize the source code that is specific to any particular CPU core implementation or CPU model.

This manual is separate and distinct from the RTEMS Porting Guide. That manual is a guide on porting RTEMS to a new architecture. This manual is focused on the more mundane CPU architecture specific issues that may impact application development. For example, if you need to write a subroutine in assembly language, it is critical to understand the calling conventions for the target architecture.

The first chapter in this manual describes these issues in general terms. In a sense, it is posing the questions one should be aware may need to be answered and understood when porting an RTEMS application to a new architecture. Each subsequent chapter gives the answers to those questions for a particular CPU architecture.

CHAPTER

TWO

PORT SPECIFIC INFORMATION

This chaper provides a general description of the type of architecture specific information which is in each of the architecture specific chapters that follow. The outline of this chapter is identical to that of the architecture specific chapters.

In each of the architecture specific chapters, this introductory section will provide an overview of the architecture:

Architecture Documents

In each of the architecture specific chapters, this section will provide pointers on where to obtain documentation.

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 Motorola 68000 and the Intel x86 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.

The set of CPU model feature macros are defined in the `cpukit/score/cpu/CPU/rtems/score/cpu.h` based upon the GNU tools multilib variant that is appropriate for the particular CPU model defined on the compilation command line.

In each of the architecture specific chapters, this section presents the set of features which vary across various implementations of the architecture that may be of importance to RTEMS application developers.

The subsections will vary amongst the target architecture chapters as the specific features may vary. However, each port will include a few common features such as the CPU Model Name and presence of a hardware Floating Point Unit. The common features are described here.

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 MC68020 processor model from the m68k architecture, this macro is set to the string "mc68020".

2.1.2 Floating Point Unit

In most architectures, the presence of a floating point unit is an option. It does not matter whether the hardware floating point support is incorporated on-chip or is an external coprocessor as long as it appears an FPU per the ISA. However, if a hardware FPU is not present, it is possible that the floating point emulation library for this CPU is not reentrant and thus context switched by RTEMS.

RTEMS provides two feature macros to indicate the FPU configuration:

- CPU_HARDWARE_FP is set to TRUE to indicate that a hardware FPU is present.

- CPU_SOFTWARE_FP is set to TRUE to indicate that a hardware FPU is not present and that the FP software emulation will be context switched.

2.2 Multilibs

Newlib and GCC provide several target libraries like the `libc.a`, `libm.a` and `libgcc.a`. These libraries are artifacts of the GCC build process. Newlib is built together with GCC. To provide optimal support for various chip derivatives and instruction set revisions multiple variants of these libraries are available for each architecture. For example one set may use software floating point support and another set may use hardware floating point instructions. These sets of libraries are called *multilibs*. Each library set corresponds to an application binary interface (ABI) and instruction set.

A multilib variant can be usually detected via built-in compiler defines at compile-time. This mechanism is used by RTEMS to select for example the context switch support for a particular BSP. The built-in compiler defines corresponding to multilibs are the only architecture specific defines allowed in the `cpukit` area of the RTEMS sources.

Invoking the GCC with the `-print-multi-lib` option lists the available multilibs. Each line of the output describes one multilib variant. The default variant is denoted by . which is selected when no or contradicting GCC machine options are selected. The multilib selection for a target is specified by target makefile fragments (see file `t-rtems` in the GCC sources and section *The Target Makefile Fragment* (https://gcc.gnu.org/onlinedocs/gccint/Target-Fragment.html#Target-Fragment) in the *GCC Internals Manual* (https://gcc.gnu.org/onlinedocs/gccint/).

2.3 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.

2.3.1 Calling Mechanism

In each of the architecture specific chapters, this subsection will describe the instruction(s) used to perform a *normal* subroutine invocation. All RTEMS directives are invoked as *normal* C language functions so it is important to the user application to understand the call and return mechanism.

2.3.2 Register Usage

In each of the architecture specific chapters, this subsection will detail the set of registers which are *NOT* preserved across subroutine invocations. The registers which are not preserved are assumed to be available for use as scratch registers. Therefore, the contents of these registers should not be assumed upon return from any RTEMS directive.

In some architectures, there may be a set of registers made available automatically as a side-effect of the subroutine invocation mechanism.

2.3.3 Parameter Passing

In each of the architecture specific chapters, this subsection will describe the mechanism by which the parameters or arguments are passed by the caller to a subroutine. In some architectures, all parameters are passed on the stack while in others some are passed in registers.

2.3.4 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.

2.4 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.

2.4.1 Flat Memory Model

Most RTEMS target processors can be initialized to support a flat address space. Although the size of addresses varies between architectures, on most RTEMS targets, an address is 32-bits wide which defines 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 may be performed in little or big endian fashion.

On smaller CPU architectures supported by RTEMS, the address space may only be 20 or 24 bits wide.

If the CPU model has support for virtual memory or segmentation, it is the responsibility of the Board Support Package (BSP) to initialize the MMU hardware to perform address translations which correspond to flat memory model.

In each of the architecture specific chapters, this subsection will describe any architecture characteristics that differ from this general description.

2.5 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.

RTEMS supports a dedicated interrupt stack for all architectures. On architectures with hardware support for a dedicated interrupt stack, it will be initialized such that when an interrupt occurs, the processor automatically switches to this dedicated stack. On architectures without hardware support for a dedicated interrupt stack which is separate from those of the tasks, RTEMS will support switching to a dedicated stack for interrupt processing.

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.

A nested interrupt is processed similarly with the exception that since the CPU is already executing on the interrupt stack, there is no need to switch to the interrupt stack.

The interrupt stacks (one for each configured processor) are statically allocated by the application configuration via `<rtems/confdefs.h>` in the special section `.rtemsstack`. This enables an optimal placement of the interrupt stacks by the Board Support Package (BSP), e.g. a fast on-chip memory. The amount of memory allocated for each interrupt stack is user configured and based upon the `<rtems/confdefs.h>` parameter `CONFIGURE_INTERRUPT_STACK_SIZE`. This parameter is described in detail in the Configuring a System chapter of the User's Guide. Since interrupts are disabled during the sequential system initialization and the `_Thread_Start_multitasking()` function does not return to the caller each interrupt stack may be used for the initialization stack on the corresponding processor.

In each of the architecture specific chapters, this section discusses the interrupt response and control mechanisms of the architecture as they pertain to RTEMS.

2.5.1 Vectoring of an Interrupt Handler

In each of the architecture specific chapters, this subsection will describe the architecture specific details of the interrupt vectoring process. In particular, it should include a description of the Interrupt Stack Frame (ISF).

2.5.2 Interrupt Levels

In each of the architecture specific chapters, this subsection will describe how the interrupt levels available on this particular architecture are mapped onto the 255 reserved in the task mode. The interrupt level value of zero (0) should always mean that interrupts are enabled.

Any use of an interrupt level that is is not undefined on a particular architecture may result in behavior that is unpredictable.

2.5.3 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 all external interrupts before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to ensure that interrupts are disabled for the shortest number of instructions possible. Since the precise number of instructions and their execution time varies based upon target CPU family, CPU model, board memory speed, compiler version, and optimization level, it is not practical to provide the precise number for all possible RTEMS configurations.

Historically, the measurements were made by hand analyzing and counting the execution time of instruction sequences during interrupt disable critical sections. For reference purposes, on a 16 Mhz Motorola MC68020, the maximum interrupt disable period was typically approximately ten (10) to thirteen (13) microseconds. This architecture was memory bound and had a slow bit scan instruction. In contrast, during the same period a 14 Mhz SPARC would have a worst case disable time of approximately two (2) to three (3) microseconds because it had a single cycle bit scan instruction and used fewer cycles for memory accesses.

If you are interested in knowing the worst case execution time for a particular version of RTEMS, please contact OAR Corporation and we will be happy to product the results as a consulting service.

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.

2.6 Default Fatal Error Processing

Upon detection of a fatal error by either the application or RTEMS during initialization the `rtems_fatal_error_occurred` directive supplied by 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 or all of them return without taking action to shutdown the processor or reset, a default fatal error handler is invoked.

Most of the action performed as part of processing the fatal error are described in detail in the Fatal Error Manager chapter in the User's Guide. However, the if no user provided extension or BSP specific fatal error handler takes action, the final default action is to invoke a CPU architecture specific function. Typically this function disables interrupts and halts the processor.

In each of the architecture specific chapters, this describes the precise operations of the default CPU specific fatal error handler.

2.7 Symmetric Multiprocessing

This section contains information about the Symmetric Multiprocessing (SMP) status of a particular architecture.

2.8 Thread-Local Storage

In order to support thread-local storage (TLS) the CPU port must implement the facilities mandated by the application binary interface (ABI) of the CPU architecture. The CPU port must initialize the TLS area in the `_CPU_Context_Initialize()` function. There are support functions available via `#include <rtems/score/tls.h>` which implement Variants I and II according to [Dre13].

`_TLS_TCB_at_area_begin_initialize()`
Uses Variant I, TLS offsets emitted by linker takes the TCB into account. For a reference implementation see `cpukit/score/cpu/arm/cpu.c`.

`_TLS_TCB_before_TLS_block_initialize()`
Uses Variant I, TLS offsets emitted by linker neglects the TCB. For a reference implementation see `c/src/lib/libcpu/powerpc/new-exceptions/cpu.c`.

`_TLS_TCB_after_TLS_block_initialize()`
Uses Variant II. For a reference implementation see `cpukit/score/cpu/sparc/cpu.c`.

The board support package (BSP) must provide the following sections and symbols in its linker command file:

```
.tdata : {
  _TLS_Data_begin = .;
  *(.tdata .tdata.* .gnu.linkonce.td.*)
  _TLS_Data_end = .;
}
.tbss : {
  _TLS_BSS_begin = .;
  *(.tbss .tbss.* .gnu.linkonce.tb.*) *(.tcommon)
  _TLS_BSS_end = .;
}
_TLS_Data_size = _TLS_Data_end - _TLS_Data_begin;
_TLS_Data_begin = _TLS_Data_size != 0 ? _TLS_Data_begin : _TLS_BSS_begin;
_TLS_Data_end = _TLS_Data_size != 0 ? _TLS_Data_end : _TLS_BSS_begin;
_TLS_BSS_size = _TLS_BSS_end - _TLS_BSS_begin;
_TLS_Size = _TLS_BSS_end - _TLS_Data_begin;
_TLS_Alignment = MAX (ALIGNOF (.tdata), ALIGNOF (.tbss));
```

2.9 CPU counter

The CPU support must implement the CPU counter interface. A CPU counter is some free-running counter. It ticks usually with a frequency close to the CPU or system bus clock. On some architectures the actual implementation is board support package dependent. The CPU counter is used for profiling of low-level functions. It is also used to implement two busy wait functions `rtems_counter_delay_ticks()` and `rtems_counter_delay_nanoseconds()` which may be used in device drivers. It may be also used as an entropy source for random number generators.

The CPU counter interface uses a CPU port specific unsigned integer type `CPU_Counter_ticks` to represent CPU counter values. The CPU port must provide the following two functions

- `_CPU_Counter_read()` to read the current CPU counter value, and
- `_CPU_Counter_difference()` to get the difference between two CPU counter values.

2.10 Interrupt Profiling

The RTEMS profiling needs support by the CPU port for the interrupt entry and exit times. In case profiling is enabled via the RTEMS build configuration option `--enable-profiling` (in this case the pre-processor symbol `RTEMS_PROFILING` is defined) the CPU port may provide data for the interrupt entry and exit times of the outer-most interrupt. The CPU port can feed interrupt entry and exit times with the `_Profiling_Outer_most_interrupt_entry_and_exit()` function (`#include <rtems/score/profiling.h>`). For an example please have a look at `cpukit/score/cpu/arm/arm_exc_interrupt.S`.

2.11 Board Support Packages

An RTEMS Board Support Package (BSP) must be designed to support a particular processor model and target board combination.

In each of the architecture specific chapters, this section will present a discussion of architecture specific BSP issues. For more information on developing a BSP, refer to *RTEMS BSP and Driver Guide* chapter titled `Board Support Packages` in the *RTEMS Classic API Guide*.

2.11.1 System Reset

An RTEMS based application is initiated or re-initiated when the processor is reset or transfer is passed to it from a boot monitor or ROM monitor.

In each of the architecture specific chapters, this subsection describes the actions that the BSP must take assuming the application gets control when the microprocessor is reset.

CHAPTER

THREE

AARCH64 SPECIFIC INFORMATION

This chapter discusses the dependencies of the *ARM AArch64 architecture* (https://en.wikipedia.org/wiki/ARM_architecture#AArch64_features) in this port of RTEMS. The ARMv8-A versions are supported by RTEMS. Processors with a MMU use a static configuration which is set up during system start. SMP is supported.

Architecture Documents

For information on the ARM AArch64 architecture refer to the *ARM Infocenter* (http://infocenter.arm.com/).

3.1 CPU Model Dependent Features

This section presents the set of features which vary across ARM AArch64 implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file `cpukit/score/cpu/aarch64/rtems/score/aarch64.h` based upon the particular CPU model flags specified on the compilation command line.

3.1.1 CPU Model Name

The macro `CPU_MODEL_NAME` is a string which designates the architectural level of this CPU model. See in `cpukit/score/cpu/aarch64/rtems/score/aarch64.h` for the values.

3.1.2 Floating Point Unit and SIMD

The Advanced SIMD (NEON) and Floating-point instruction set extension is supported and expected to be present since all ARMv8-A CPUs are expected to support it as per the *ARMv8-A Programmer's Guide Chapter 7 introduction* (https://developer.arm.com/docs/den0024/a/aarch64-floating-point-and-neon). As such, `CPU_HARDWARE_FP` will always be set to `TRUE`.

3.2 Multilibs

The following multilib variants are available:

1. `ILP32`: AArch64 instruction set and registers using 32bit long int and pointers
2. `LP64`: AArch64 instruction set and registers using 64bit long int and pointers

Use for example the following GCC options:

```
-mcpu=cortex-a53 -mabi=ilp32
```

to build an application or BSP for the ARMv8-A architecture and tune the code for a Cortex-A53 processor. It is important to select the correct ABI.

3.3 Calling Conventions

Please refer to the *Procedure Call Standard for the ARM 64-bit Architecture* (https://github.com/ARM-software/abi-aa/releases/download/2019Q4/aapcs64.pdf).

3.4 Memory Model

A flat 64-bit or 32-bit memory model is supported depending on the selected multilib variant. All AArch64 CPU variants support a built-in MMU for which basic initialization for a flat memory model is handled.

Note that memcpy() and memset() must not be used on device memory as those functions are hand-optimized and will take advantage of unaligned accesses. *As per ARM*(https://developer.arm.com/documentation/ka004708/latest), unaligned accesses are not permitted for device memory.

3.5 Interrupt Processing

The Reset Vector is determined using RVBAR and is Read-Only. RVBAR is set using configuration signals only sampled at reset. The ARMv8 architecture has four exception types:

- Synchronous Exception
- Interrupt (IRQ)
- Fast Interrupt (FIQ)
- System Error Exception

Of these types only the synchronous and IRQ exceptions have explicit operating system support. It is intentional that the FIQ is not supported by the operating system. Without operating system support for the FIQ it is not necessary to disable them during critical sections of the system.

3.5.1 Interrupt Levels

There are exactly two interrupt levels on ARMv8 with respect to RTEMS. Level zero corresponds to interrupts enabled. Level one corresponds to interrupts disabled.

3.5.2 Interrupt Stack

The board support package must initialize the interrupt stack. The memory for the stacks is usually reserved in the linker script. The interrupt stack pointer is stored in the EL0 stack pointer and is accessed by switching to SP0 mode at the beginning of interrupt calls and back to SPx mode after completion of interrupt calls using the *spsel* instruction.

3.6 Default Fatal Error Processing

The default fatal error handler for this architecture performs the following actions:

- disables operating system supported interrupts (IRQ),
- places the error code in `x0`, and
- executes an infinite loop to simulate a halt processor instruction.

3.7 Symmetric Multiprocessing

SMP is supported on ARMv8-A. Available platforms are:

- Xilinx ZynqMP (QEMU and hardware using PSCI via ARM Trusted Firmware)

3.8 Thread-Local Storage

Thread-local storage (TLS) is supported. AArch64 uses unmodified TLS variant I which is not explcitly stated, but can be inferred from the behavior of GCC and *Addenda to, and Errata in, the ABI for the Arm® Architecture* (https://developer.arm.com/documentation/ihi0045/g). This alters expectations for the size of the TLS Thread Control Block (TCB) such that, under the LP64 multilib variant, the TCB is 16 bytes in size instead of 8 bytes.

CHAPTER

FOUR

ARM SPECIFIC INFORMATION

This chapter discusses the *ARM architecture* (http://en.wikipedia.org/wiki/ARM_architecture) dependencies in this port of RTEMS. The ARMv4T (and compatible), ARMv7-A, ARMv7-R and ARMv7-M architecture versions are supported by RTEMS. Processors with a MMU use a static configuration which is set up during system start. SMP is supported.

Architecture Documents

For information on the ARM architecture refer to the *ARM Infocenter* (http://infocenter.arm.com/).

4.1 CPU Model Dependent Features

This section 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 flags specified on the compilation command line.

4.1.1 CPU Model Name

The macro `CPU_MODEL_NAME` is a string which designates the architectural level of this CPU model. See in `cpukit/score/cpu/arm/rtems/score/arm.h` for the values.

4.1.2 Count Leading Zeroes Instruction

The ARMv5 and later instruction sets have 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. This is currently not used.

4.1.3 Floating Point Unit

The following floating point units are supported:

- VFPv2 (for example available on ARM926EJ-S processors)
- VFPv3-D32/NEON (for example available on Cortex-A processors)
- VFPv3-D16 (for example available on Cortex-R processors)
- FPv4-SP-D16 (for example available on Cortex-M processors)
- FPv5-D16 (for example available on Cortex-M7 processors)

4.2 Multilibs

The following multilibs are available:

1. `.`: ARMv4T, ARM instruction set
2. `vfp/hard`: ARMv4T, ARM instruction set with hard-float ABI and VFPv2 support
3. `thumb`: ARMv4T, Thumb-1 instruction set
4. `thumb/armv6-m`: ARMv6M, subset of Thumb-2 instruction set
5. `thumb/armv7-a`: ARMv7-A, Thumb-2 instruction set
6. `thumb/armv7-a/neon/hard`: ARMv7-A, Thumb-2 instruction set with hard-float ABI Neon and VFP-D32 support
7. `thumb/armv7-r`: ARMv7-R, Thumb-2 instruction set
8. `thumb/armv7-r/vfpv3-d16/hard`: ARMv7-R, Thumb-2 instruction set with hard-float ABI VFP-D16 support
9. `thumb/cortex-m3`: Cortex-M3, Thumb-2 instruction set with hardware integer division (SDIV/UDIV) and a fix for Cortex-M3 Errata 602117.
10. `thumb/cortex-m4`: Cortex-M4, Thumb-2 instruction set with hardware integer division (SDIV/UDIV) and DSP instructions
11. `thumb/cortex-m4/fpv4-sp-d16`: Cortex-M4, Thumb-2 instruction set with hardware integer division (SDIV/UDIV), DSP instructions and hard-float ABI FPv4-SP support
12. `thumb/cortex-m7/fpv5-d16`: Cortex-M7, Thumb-2 instruction set with hard-float ABI and FPv5-D16 support
13. `eb/thumb/armv7-r`: ARMv7-R, Big-endian Thumb-2 instruction set
14. `eb/thumb/armv7-r/vfpv3-d16/hard`: ARMv7-R, Big-endian Thumb-2 instruction set with hard-float ABI VFP-D16 support

Multilib 1., 2. and 3. support the legacy ARM7TDMI and ARM926EJ-S processors.

Multilib 4. supports the Cortex-M0 and Cortex-M1 cores.

Multilib 5. and 6. support the Cortex-A processors.

Multilib 7., 8., 13. and 14. support the Cortex-R processors. Here also big-endian variants are available.

Use for example the following GCC options:

```
-mthumb -march=armv7-a -mfpu=neon -mfloat-abi=hard -mtune=cortex-a9
```

to build an application or BSP for the ARMv7-A architecture and tune the code for a Cortex-A9 processor. It is important to select the options used for the multilibs. For example:

```
-mthumb -mcpu=cortex-a9
```

alone will not select the ARMv7-A multilib.

4.3 Calling Conventions

Please refer to the *Procedure Call Standard for the ARM Architecture* (http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042c/IHI0042C_aapcs.pdf).

4.4 Memory Model

A flat 32-bit memory model is supported. The board support package must take care of initializing the MMU if necessary.

Note that architecture variants which support unaligned accesses must not use memcpy() or memset() on device memory as those functions are hand-optimized and will take advantage of unaligned accesses where available. *As per ARM* (https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Application-Level-Memory-Model/Alignment-support/Unaligned-data-access-restrictions-in-ARMv7-and-ARMv6), unaligned accesses are not permitted for device memory.

4.5 Interrupt Processing

The ARMv4T (and compatible) architecture has seven exception types:

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

Of these types only the IRQ has explicit operating system support. It is intentional that the FIQ is not supported by the operating system. Without operating system support for the FIQ it is not necessary to disable them during critical sections of the system.

The ARMv7-M architecture has a completely different exception model. Here interrupts are disabled with a write of 0x80 to the `basepri_max` register. This means that all exceptions and interrupts with a priority value of greater than or equal to 0x80 are disabled. Thus exceptions and interrupts with a priority value of less than 0x80 are non-maskable with respect to the operating system and therefore must not use operating system services. Several support libraries of chip vendors implicitly shift the priority value somehow before the value is written to the NVIC IPR register. This can easily lead to confusion.

4.5.1 Interrupt Levels

There are exactly two interrupt levels on ARM with respect to RTEMS. Level zero corresponds to interrupts enabled. Level one corresponds to interrupts disabled.

4.5.2 Interrupt Stack

The board support package must initialize the interrupt stack. The memory for the stacks is usually reserved in the linker script.

4.6 Default Fatal Error Processing

The default fatal error handler for this architecture performs the following actions:

- disables operating system supported interrupts (IRQ),
- places the error code in `r0`, and
- executes an infinite loop to simulate a halt processor instruction.

4.7 Symmetric Multiprocessing

SMP is supported on ARMv7-A. Available platforms are:

- Altera Cyclone V
- NXP i.MX 7
- Xilinx Zynq

4.8 Thread-Local Storage

Thread-local storage is supported.

CHAPTER

FIVE

ATMEL AVR SPECIFIC INFORMATION

This chapter discusses the AVR architecture dependencies in this port of RTEMS.

Architecture Documents

For information on the AVR architecture, refer to the following documents available from Atmel.

TBD

- See other CPUs for documentation reference formatting examples.

5.1 CPU Model Dependent Features

CPUs of the AVR 53X only differ in the peripherals and thus in the device drivers. This port does not yet support the 56X dual core variants.

5.1.1 Count Leading Zeroes Instruction

The AVR CPU has the XXX 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.

5.2 Calling Conventions

5.2.1 Processor Background

The AVR architecture supports a simple call and return mechanism. A subroutine is invoked via the call (`call`) instruction. This instruction saves the return address in the `RETS` register and transfers the execution to the given address.

It is the called funcions responsability to use the link instruction to reserve space on the stack for the local variables. Returning from a subroutine is done by using the RTS (`RTS`) instruction which loads the PC with the adress stored in RETS.

It is is important to note that the `call` 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.

5.2.2 Register Usage

A called function may clobber all registers, except RETS, R4-R7, P3-P5, FP and SP. It may also modify the first 12 bytes in the caller's stack frame which is used as an argument area for the first three arguments (which are passed in R0...R3 but may be placed on the stack by the called function).

5.2.3 Parameter Passing

RTEMS assumes that the AVR GCC calling convention is followed. The first three parameters are stored in registers R0, R1, and R2. All other parameters are put pushed on the stack. The result is returned through register R0.

5.3 Memory Model

The AVR family architecutre support a single unified 4 GB byte address space using 32-bit addresses. It maps all resources like internal and external memory and IO registers into separate sections of this common address space.

The AVR architcture supports some form of memory protection via its Memory Management Unit. Since the AVR port runs in supervisior mode this memory protection mechanisms are not used.

5.4 Interrupt Processing

Discussed in this chapter are the AVR's interrupt response and control mechanisms as they pertain to RTEMS.

5.4.1 Vectoring of an Interrupt Handler

TBD

5.4.2 Disabling of Interrupts by RTEMS

During interrupt disable critical sections, RTEMS disables interrupts to level N (N) before the execution of this section and restores them to the previous level upon completion of the section. RTEMS uses the instructions CLI and STI to enable and disable Interrupts. Emulation, Reset, NMI and Exception Interrupts are never disabled.

5.4.3 Interrupt Stack

The AVR Architecture works with two different kind of stacks, User and Supervisor Stack. Since RTEMS and its Application run in supervisor mode, all interrupts will use the interrupted tasks stack for execution.

5.5 Default Fatal Error Processing

The default fatal error handler for the AVR 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.

5.6 Symmetric Multiprocessing

SMP is not supported.

5.7 Thread-Local Storage

Thread-local storage is not supported due to a broken tool chain.

5.8 Board Support Packages

5.8.1 System Reset

TBD

CHAPTER

SIX

BLACKFIN SPECIFIC INFORMATION

This chapter discusses the Blackfin architecture dependencies in this port of RTEMS.

Architecture Documents

For information on the Blackfin architecture, refer to the following documents available from Analog Devices.

TBD

- *"ADSP-BF533 Blackfin Processor Hardware Reference."* http://www.analog.com/UploadedFiles/Associated_Docs/892485982bf533_hwr.pdf

6.1 CPU Model Dependent Features

CPUs of the Blackfin 53X only differ in the peripherals and thus in the device drivers. This port does not yet support the 56X dual core variants.

6.1.1 Count Leading Zeroes Instruction

The Blackfin CPU has the BITTST 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.

6.2 Calling Conventions

This section is heavily based on content taken from the Blackfin uCLinux documentation wiki which is edited by Analog Devices and Arcturus Networks. http://docs.blackfin.uclinux.org/

6.2.1 Processor Background

The Blackfin architecture supports a simple call and return mechanism. A subroutine is invoked via the call (`call`) instruction. This instruction saves the return address in the `RETS` register and transfers the execution to the given address.

It is the called funcions responsability to use the link instruction to reserve space on the stack for the local variables. Returning from a subroutine is done by using the RTS (`RTS`) instruction which loads the PC with the adress stored in RETS.

It is is important to note that the `call` 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.

6.2.2 Register Usage

A called function may clobber all registers, except RETS, R4-R7, P3-P5, FP and SP. It may also modify the first 12 bytes in the caller's stack frame which is used as an argument area for the first three arguments (which are passed in R0...R3 but may be placed on the stack by the called function).

6.2.3 Parameter Passing

RTEMS assumes that the Blackfin GCC calling convention is followed. The first three parameters are stored in registers R0, R1, and R2. All other parameters are put pushed on the stack. The result is returned through register R0.

6.3 Memory Model

The Blackfin family architecutre support a single unified 4 GB byte address space using 32-bit addresses. It maps all resources like internal and external memory and IO registers into separate sections of this common address space.

The Blackfin architcture supports some form of memory protection via its Memory Management Unit. Since the Blackfin port runs in supervisior mode this memory protection mechanisms are not used.

6.4 Interrupt Processing

Discussed in this chapter are the Blackfin's interrupt response and control mechanisms as they pertain to RTEMS. The Blackfin architecture support 16 kinds of interrupts broken down into Core and general-purpose interrupts.

6.4.1 Vectoring of an Interrupt Handler

RTEMS maps levels 0 -15 directly to Blackfins event vectors EVT0 - EVT15. Since EVT0 - EVT6 are core events and it is suggested to use EVT15 and EVT15 for Software interrupts, 7 Interrupts (EVT7-EVT13) are left for periferical use.

When installing an RTEMS interrupt handler RTEMS installs a generic Interrupt Handler which saves some context and enables nested interrupt servicing and then vectors to the users interrupt handler.

6.4.2 Disabling of Interrupts by RTEMS

During interrupt disable critical sections, RTEMS disables interrupts to level four (4) before the execution of this section and restores them to the previous level upon completion of the section. RTEMS uses the instructions CLI and STI to enable and disable Interrupts. Emulation, Reset, NMI and Exception Interrupts are never disabled.

6.4.3 Interrupt Stack

The Blackfin Architecture works with two different kind of stacks, User and Supervisor Stack. Since RTEMS and its Application run in supervisor mode, all interrupts will use the interrupted tasks stack for execution.

6.5 Default Fatal Error Processing

The default fatal error handler for the Blackfin 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.

6.6 Symmetric Multiprocessing

SMP is not supported.

6.7 Thread-Local Storage

Thread-local storage is not implemented.

6.8 Board Support Packages

6.8.1 System Reset

TBD

CHAPTER

SEVEN

EPIPHANY SPECIFIC INFORMATION

Due to an unmaintained toolchain (internal errors in GCC, no FSF GDB integration) the Epiphany architecture was obsoleted in RTEMS 5.1 and removed in RTEMS 6.1.

CHAPTER

EIGHT

INTEL/AMD X86 SPECIFIC INFORMATION

This chapter discusses the Intel x86 architecture dependencies in this port of RTEMS. This family has multiple implementations from multiple vendors and suffers more from having evolved rather than being designed for growth.

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*.

8.1 CPU Model Dependent Features

This section 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 `cpukit/score/cpu/i386/i386.h` based upon the particular CPU model specified on the compilation command line.

8.1.1 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.

8.2 Calling Conventions

8.2.1 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.

8.2.2 Calling Mechanism

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

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

8.2.4 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.

8.3 Memory Model

8.3.1 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).

8.4 Interrupt Processing

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.

8.4.1 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.

8.4.2 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

8.4.3 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.

8.4.4 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.

8.5 Default Fatal Error Processing

The default fatal error handler for this architecture disables processor interrupts, places the error code in EAX, and executes a HLT instruction to halt the processor.

8.6 Symmetric Multiprocessing

SMP is not supported.

8.7 Thread-Local Storage

Thread-local storage is supported.

8.8 Board Support Packages

8.8.1 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.

8.8.2 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 i386 data structures and their contents, refer to Intel's 386 Programmer's Reference Manual.

CHAPTER

NINE

LATTICE MICO32 SPECIFIC INFORMATION

This chaper discusses the Lattice Mico32 architecture dependencies in this port of RTEMS. The Lattice Mico32 is a 32-bit Harvard, RISC architecture "soft" microprocessor, available for free with an open IP core licensing agreement. Although mainly targeted for Lattice FPGA devices the microprocessor can be implemented on other vendors' FPGAs, too.

Architecture Documents

For information on the Lattice Mico32 architecture, refer to the following documents available from Lattice Semiconductor http://www.latticesemi.com/.

- *"LatticeMico32 Processor Reference Manual"* http://www.latticesemi.com/dynamic/view_document.cfm?document_id=20890

9.1 CPU Model Dependent Features

The Lattice Mico32 architecture allows for different configurations of the processor. This port is based on the assumption that the following options are implemented:

- hardware multiplier
- hardware divider
- hardware barrel shifter
- sign extension instructions
- instruction cache
- data cache
- debug

9.2 Register Architecture

This section gives a brief introduction to the register architecture of the Lattice Mico32 processor.

The Lattice Mico32 is a RISC archictecture processor with a 32-register file of 32-bit registers.

Register Name

Function

r0

holds value zero

r1-r25

general purpose

r26/gp

general pupose / global pointer

r27/fp

general pupose / frame pointer

r28/sp

stack pointer

r29/ra

return address

r30/ea

exception address

r31/ba

breakpoint address

Note that on processor startup all register values are undefined including r0, thus r0 has to be initialized to zero.

9.3 Calling Conventions

9.3.1 Calling Mechanism

A call instruction places the return address to register r29 and a return from subroutine (ret) is actually a branch to r29/ra.

9.3.2 Register Usage

A subroutine may freely use registers r1 to r10 which are *not* preserved across subroutine invocations.

9.3.3 Parameter Passing

When calling a C function the first eight arguments are stored in registers r1 to r8. Registers r1 and r2 hold the return value.

9.4 Memory Model

The Lattice Mico32 processor supports a flat memory model with a 4 Gbyte address space with 32-bit addresses.

The following data types are supported:

Type	Bits	C Compiler Type
unsigned byte	8	unsigned char
signed byte	8	char
unsigned half-word	16	unsigned short
signed half-word	16	short
unsigned word	32	unsigned int / unsigned long
signed word	32	int / long

Data accesses need to be aligned, with unaligned accesses result are undefined.

9.5 Interrupt Processing

The Lattice Mico32 has 32 interrupt lines which are however served by only one exception vector. When an interrupt occurs following happens:

- address of next instruction placed in r30/ea
- IE field of IE CSR saved to EIE field and IE field cleared preventing further exceptions from occuring.
- branch to interrupt exception address EBA CSR + 0xC0

The interrupt exception handler determines from the state of the interrupt pending registers (IP CSR) and interrupt enable register (IE CSR) which interrupt to serve and jumps to the interrupt routine pointed to by the corresponding interrupt vector.

For now there is no dedicated interrupt stack so 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.

Nested interrupts are not supported.

9.6 Default Fatal Error Processing

Upon detection of a fatal error by either the application or RTEMS during initialization the `rtems_fatal_error_occurred` directive supplied by 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 or all of them return without taking action to shutdown the processor or reset, a default fatal error handler is invoked.

Most of the action performed as part of processing the fatal error are described in detail in the Fatal Error Manager chapter in the User's Guide. However, the if no user provided extension or BSP specific fatal error handler takes action, the final default action is to invoke a CPU architecture specific function. Typically this function disables interrupts and halts the processor.

In each of the architecture specific chapters, this describes the precise operations of the default CPU specific fatal error handler.

9.7 Symmetric Multiprocessing

SMP is not supported.

9.8 Thread-Local Storage

Thread-local storage is not implemented.

9.9 Board Support Packages

There are no Lattice Micro32 specific notes on BSPs.

9.9.1 System Reset

An RTEMS based application is initiated or re-initiated when the processor is reset.

CHAPTER

TEN

RENESAS M32C SPECIFIC INFORMATION

The port for this architecture was removed in RTEMS 5.1.

CHAPTER

ELEVEN

M68XXX AND COLDFIRE SPECIFIC INFORMATION

This chapter discusses the Freescale (formerly Motorola) MC68xxx and Coldfire architectural dependencies. The MC68xxx family has a wide variety of CPU models within it based upon different CPU core implementations. Ignoring the Coldfire parts, 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.

Architecture Documents

For information on the MC68xxx and Coldfire architecture, refer to the following documents available from Freescale website (http//www.freescale.com/):

- *M68000 Family Reference, Motorola, FR68K/D*.
- *MC68020 User's Manual, Motorola, MC68020UM/AD*.
- *MC68881/MC68882 Floating-Point Coprocessor User's Manual, Motorola, MC68881UM/AD*.

11.1 CPU Model Dependent Features

This section presents the set of features which vary across m68k/Coldfire implementations that 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 selected on the compilation command line.

11.1.1 BFFFO Instruction

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

11.1.2 Vector Base Register

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

11.1.3 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.

11.1.4 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.

11.1.5 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.

11.2 Calling Conventions

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.

11.2.1 Calling Mechanism

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

11.2.2 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.

11.2.3 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.

11.3 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.

11.4 Interrupt Processing

Discussed in this section are the MC68xxx's interrupt response and control mechanisms as they pertain to RTEMS.

11.4.1 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.

11.4.1.1 Models Without Separate Interrupt Stacks

Upon receipt of an interrupt the MC68xxx family members without separate interrupt stacks automatically use software to switch stacks.

11.4.1.2 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

11.4.2 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
```

(continues on next page)

(continued from previous page)

```
    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 :-)

11.4.3 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.

11.5 Default Fatal Error Processing

The default fatal error handler for this architecture disables processor interrupts to level 7, places the error code in D0, and executes a `stop` instruction to simulate a halt processor instruction.

11.6 Symmetric Multiprocessing

SMP is not supported.

11.7 Thread-Local Storage

Thread-local storage is supported.

11.8 Board Support Packages

11.8.1 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.

11.8.2 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 context switch to the first task, the VBR MUST be set by either RTEMS of the BSP before this occurs ensure 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.

CHAPTER

TWELVE

XILINX MICROBLAZE SPECIFIC INFORMATION

This chapter discusses the dependencies of the *MicroBlaze architecture* (https://en.wikipedia.org/wiki/MicroBlaze).

Architecture Documents

For information on the MicroBlaze architecture, refer to *UG984 MicroBlaze Processor Reference Guide* (https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_2/ug984-vivado-microblaze-ref.pdf).

12.1 CPU Model Dependent Features

There are no CPU model dependent features in this port.

12.2 Calling Conventions

Please refer to “Chapter 4: MicroBlaze Application Binary Interface” of *UG984 MicroBlaze Processor Reference Guide* (https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_2/ug984-vivado-microblaze-ref.pdf).

12.3 Interrupt Processing

Hardware exceptions, interrupts, and user exceptions are all supported. When a hardware exception or user exception occurs, a fatal error will be generated. When an interrupt occurs, the interrupt source is determined by reading the AXI Interrupt Controller's Interrupt Status Register and masking it with the Interrupt Enable Register.

12.3.1 Interrupt Levels

There are exactly two interrupt levels on MicroBlaze with respect to RTEMS. Level zero corresponds to interrupts disabled. Level one corresponds to interrupts enabled. This is the inverse of how most other architectures handle interrupt enable status.

12.3.2 Interrupt Stack

The memory region for the interrupt stack is defined by the BSP.

12.4 Default Fatal Error Processing

The default fatal error is BSP-specific.

12.5 Symmetric Multiprocessing

SMP is not supported.

12.6 Thread-Local Storage

Thread-local storage is supported.

CHAPTER

THIRTEEN

MIPS SPECIFIC INFORMATION

This chapter discusses the MIPS architecture dependencies in this port of RTEMS. The MIPS family has a wide variety of implementations by a wide range of vendors. Consequently, there are many, many CPU models within it.

Architecture Documents

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

13.1 CPU Model Dependent Features

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

13.1.1 Another Optional Feature

The macro XXX

13.2 Calling Conventions

13.2.1 Processor Background

TBD

13.2.2 Calling Mechanism

TBD

13.2.3 Register Usage

TBD

13.2.4 Parameter Passing

TBD

13.3 Memory Model

13.3.1 Flat Memory Model

The MIPS 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 MIPS family members such as the support virtual memory and segmentation. RTEMS does not support virtual memory or segmentation on any of these family members.

13.4 Interrupt Processing

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 MIPS's interrupt response and control mechanisms as they pertain to RTEMS.

13.4.1 Vectoring of an Interrupt Handler

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

- TBD

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.

13.4.2 Interrupt Levels

TBD

13.5 Default Fatal Error Processing

The default fatal error handler for this target architecture disables processor interrupts, places the error code in *XXX*, and executes a``XXX`` instruction to simulate a halt processor instruction.

13.6 Symmetric Multiprocessing

SMP is not supported.

13.7 Thread-Local Storage

Thread-local storage is not implemented.

13.8 Board Support Packages

13.8.1 System Reset

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

- TBD

13.8.2 Processor Initialization

TBD

CHAPTER

FOURTEEN

ALTERA NIOS II SPECIFIC INFORMATION

14.1 Symmetric Multiprocessing

SMP is not supported.

14.2 Thread-Local Storage

Thread-local storage is supported.

CHAPTER

FIFTEEN

OPENRISC 1000 SPECIFIC INFORMATION

This chapter discusses the` OpenRISC 1000 architecture http://opencores.org/or1k/Main_Page dependencies in this port of RTEMS. There are many implementations for OpenRISC like or1200 and mor1kx. Currently RTEMS supports basic features that all implementations should have.

Architecture Documents

For information on the OpenRISC 1000 architecture refer to the` OpenRISC 1000 architecture manual http://openrisc.github.io/or1k.html.

15.1 Calling Conventions

Please refer to the`Function Calling Sequence http://openrisc.github.io/or1k.html#__RefHeading__504887_595890882.

15.1.1 Floating Point Unit

A floating point unit is currently not supported.

15.2 Memory Model

A flat 32-bit memory model is supported.

15.3 Interrupt Processing

OpenRISC 1000 architecture has 13 exception types:

- Reset
- Bus Error
- Data Page Fault
- Instruction Page Fault
- Tick Timer
- Alignment
- Illegal Instruction
- External Interrupt
- D-TLB Miss
- I-TLB Miss
- Range
- System Call
- Floating Point
- Trap

15.3.1 Interrupt Levels

There are only two levels: interrupts enabled and interrupts disabled.

15.3.2 Interrupt Stack

The OpenRISC RTEMS port uses a dedicated software interrupt stack. The stack for interrupts is allocated during interrupt driver initialization. When an interrupt is entered, the _ISR_Handler routine is responsible for switching from the interrupted task stack to RTEMS software interrupt stack.

15.4 Default Fatal Error Processing

The default fatal error handler for this architecture performs the following actions:

- disables operating system supported interrupts (IRQ),
- places the error code in `r0`, and
- executes an infinite loop to simulate a halt processor instruction.

15.5 Symmetric Multiprocessing

SMP is not supported.

CHAPTER

SIXTEEN

POWERPC SPECIFIC INFORMATION

16.1 Multilibs

The following multilibs are available:

1. `.`: 32-bit PowerPC with FPU
2. `nof`: 32-bit PowerPC with software floating point support
3. `m403`: Instruction set for PPC403 with FPU
4. `m505`: Instruction set for MPC505 with FPU
5. `m603e`: Instruction set for MPC603e with FPU
6. `m603e/nof`: Instruction set for MPC603e with software floating point support
7. `m604`: Instruction set for MPC604 with FPU
8. `m604/nof`: Instruction set for MPC604 with software floating point support
9. `m860`: Instruction set for MPC860 with FPU
10. `m7400`: Instruction set for MPC7500 with FPU
11. `m7400/nof`: Instruction set for MPC7500 with software floating point support
12. `m8540`: Instruction set for e200, e500 and e500v2 cores with single-precision FPU and SPE
13. `m8540/gprsdouble`: Instruction set for e200, e500 and e500v2 cores with double-precision FPU and SPE
14. `m8540/nof/nospe`: Instruction set for e200, e500 and e500v2 cores with software floating point support and no SPE
15. `me6500/m32`: 32-bit instruction set for e6500 core with FPU and AltiVec
16. `me6500/m32/nof/noaltivec`: 32-bit instruction set for e6500 core with software floating point support and no AltiVec
17. `me6500/m64`: 64-bit instruction set for e6500 core with FPU and AltiVec
18. `me6500/m64/nof/noaltivec`: 64-bit instruction set for e6500 core with software floating point support and no AltiVec

16.2 Application Binary Interface

In 32-bit PowerPC configurations the ABI defined by Power Architecture 32-bit Application Binary Interface Supplement 1.0 - Embedded is used.

In 64-bit PowerPC configurations the ABI defined by Power Architecture 64-Bit ELF V2 ABI Specification, Version 1.1 is used.

16.3 Special Registers

The following special-purpose registers are used by RTEMS:

Special-Purpose Register General 0 (SPRG0)
In SMP configurations, this register contains the address of the per-CPU control of the processor.

Special-Purpose Register General 1 (SPRG1)
This register contains the interrupt stack pointer for the outer-most interrupt service routine.

Special-Purpose Register General 2 (SPRG2)
This register contains the address of interrupt stack area begin.

16.4 Memory Model

The memory model is flat.

16.5 Interrupt Processing

16.5.1 Interrupt Levels

There are exactly two interrupt levels on PowerPC with respect to RTEMS. Level zero corresponds to interrupts enabled. Level one corresponds to interrupts disabled.

16.5.2 Interrupt Stack

The interrupt stack size can be configured via the `CONFIGURE_INTERRUPT_STACK_SIZE` application configuration option.

16.6 Default Fatal Error Processing

The default fatal error handler is BSP-specific.

16.7 Symmetric Multiprocessing

SMP is supported. Available platforms are the Freescale QorIQ P series (e.g. P1020) and T series (e.g. T2080, T4240).

16.8 Thread-Local Storage

Thread-local storage is supported.

16.9 64-bit Caveats

- The thread pointer is `r13` in contrast to `r2` used in the 32-bit ABI.
- The TOC pointer is `r2`. It must be initialized as part of the C run-time setup. A valid stack pointer is not enough to call C functions. They may use the TOC to get addresses and constants.
- The TOC must be within the first 2GiB of the address space. This simplifies the interrupt prologue, since the `r2` can be set to `.TOC.` via the usual `lis` followed by `ori` combination. The `lis` is subject to sign-extension.
- The `PPC_REG_LOAD`, `PPC_REG_STORE`, `PPC_REG_STORE_UPDATE`, and `PPC_REG_CMP` macros are available for assembly code to provide register size operations selected by the GCC `-m32` and `-m64` options.
- The `MSR[CM]` bit must be set all the time, otherwise the MMU translation my yield unexpected results. The `EPCR[ICM]` or `EPCR[GICM]` bits may be used to enable the 64-bit compute mode for exceptions.

CHAPTER

SEVENTEEN

RISC-V SPECIFIC INFORMATION

17.1 Calling Conventions

Please refer to the RISC-V ELF psABI specification.

17.2 Multilibs

The GCC for RISC-V can generate code for several 32-bit and 64-bit ISA/ABI variants. The following multilibs are available:

- `.`: The default multilib ISA is RV32IMAFDC with ABI ILP32D.
- `rv32i/ilp32`: ISA RV32I with ABI ILP32.
- `rv32im/ilp32`: ISA RV32IM with ABI ILP32.
- `rv32imafd/ilp32d`: ISA RV32IMAFD with ABI ILP32D.
- `rv32iac/ilp32`: ISA RV32IAC with ABI ILP32.
- `rv32imac/ilp32`: ISA RV32IMAC with ABI ILP32.
- `rv32imafc/ilp32f`: ISA RV32IMAFC with ABI ILP32F.
- `rv64imafd/lp64d`: ISA RV64IMAFD with ABI LP64D and code model medlow.
- `rv64imafd/lp64d/medany`: ISA RV64IMAFD with ABI LP64D and code model medany.
- `rv64imac/lp64`: ISA RV64IMAC with ABI LP64 and code model medlow.
- `rv64imac/lp64/medany`: ISA RV64IMAC with ABI LP64 and code model medany.
- `rv64imafdc/lp64d`: ISA RV64IMAFDC with ABI LP64D and code model medlow.
- `rv64imafdc/lp64d/medany`: ISA RV64IMAFDC with ABI LP64D and code model medany.

17.3 Interrupt Processing

Interrupt exceptions are handled via the interrupt extensions API. All other exceptions end up in a fatal error (RTEMS_FATAL_SOURCE_EXCEPTION).

17.3.1 Interrupt Levels

There are exactly two interrupt levels on RISC-V with respect to RTEMS. Level zero corresponds to machine interrupts enabled. Level one corresponds to machine interrupts disabled.

17.3.2 Interrupt Stack

The memory region for the interrupt stack is defined by the BSP.

17.4 Default Fatal Error Processing

The default fatal error is BSP-specific.

17.5 Symmetric Multiprocessing

SMP is supported.

17.6 Thread-Local Storage

Thread-local storage is supported.

CHAPTER

EIGHTEEN

SUPERH SPECIFIC INFORMATION

This chapter discusses the SuperH architecture dependencies in this port of RTEMS. The SuperH family has a wide variety of implementations by a wide range of vendors. Consequently, there are many, many CPU models within it.

Architecture Documents

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

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

18.1 CPU Model Dependent Features

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

18.1.1 Another Optional Feature

The macro XXX

18.2 Calling Conventions

18.2.1 Calling Mechanism

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

18.2.2 Register Usage

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

18.2.3 Parameter Passing

XXX

18.3 Memory Model

18.3.1 Flat Memory Model

The SuperH 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 SuperH family members support virtual memory and segmentation. RTEMS does not support virtual memory or segmentation on any of the SuperH family members. It is the responsibility of the BSP to initialize the mapping for a flat memory model.

18.4 Interrupt Processing

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 MIPS's interrupt response and control mechanisms as they pertain to RTEMS.

18.4.1 Vectoring of an Interrupt Handler

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

- TBD

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.

18.4.2 Interrupt Levels

TBD

18.5 Default Fatal Error Processing

The default fatal error handler for this architecture disables processor interrupts, places the error code in *XXX*, and executes a `XXX` instruction to simulate a halt processor instruction.

18.6 Symmetric Multiprocessing

SMP is not supported.

18.7 Thread-Local Storage

Thread-local storage is not implemented.

18.8 Board Support Packages

18.8.1 System Reset

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

- TBD

18.8.2 Processor Initialization

TBD

CHAPTER

NINETEEN

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. This architectural port is for SPARC Version 7 and 8. Implementations for SPARC V9 are in the sparc64 target.

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. (https://sparc.org/):

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

ERC32 Specific Information

The European Space Agency's ERC32 is a microprocessor 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.

The ERC32 is available from Atmel as the TSC695F.

The RTEMS configuration of GDB enables the SPARC Instruction Simulator (SIS) which can simulate the ERC32 as well as the follow up LEON2 and LEON3 microprocessors.

19.1 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.

19.1.1 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.

19.1.1.1 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".

19.1.1.2 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.

19.1.1.3 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.

19.1.1.4 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.

19.1.1.5 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.

19.1.2 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.

19.2 Calling Conventions

Each high-level language compiler generates subroutine entry and exit code based upon a set of rules known as the application binary interface (ABI) calling convention. These rules address the following issues:

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

An ABI calling convention is of importance when interfacing to subroutines written in another language either assembly or high-level. It determines also the set of registers to be saved or restored during a context switch and interrupt processing.

The ABI relevant for RTEMS on SPARC is defined by SYSTEM V APPLICATION BINARY INTERFACE, SPARC Processor Supplement, Third Edition.

19.2.1 Programming Model

This section discusses the programming model for the SPARC architecture.

19.2.1.1 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 Name	Description
g0	na	reads return 0, writes are ignored
o6	sp	stack pointer
i6	fp	frame pointer
i7	na	return address

The registers g2 through g4 are reserved for applications. GCC uses them as volatile registers by default. So they are treated like volatile registers in RTEMS as well.

The register g6 is reserved for the operating system and contains the address of the per-CPU control block of the current processor. This register is initialized during system start and then remains unchanged. It is not saved/restored by the context switch or interrupt processing code.

The register g7 is reserved for the operating system and contains the thread pointer used for thread-local storage (TLS) as mandated by the SPARC ABI.

19.2.1.2 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 (FSR) specifies the behavior of the floating point unit for rounding, contains its condition codes, version specification, and trap information.

According to the ABI all floating point registers and the floating point status register (FSR) are volatile. Thus the floating point context of a thread is the empty set. The rounding direction is a system global state and must not be modified by threads.

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.

19.2.1.3 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.

19.2.2 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.

19.2.3 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.

In case a floating-point unit is supported, then floating-point return values appear in the floating-point registers. Single-precision values occupy %f0; double-precision values occupy %f0 and %f1. Otherwise, these are scratch registers. Due to this the hardware and software floating-point ABIs are incompatible.

19.2.4 Calling Mechanism

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

19.2.5 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.

19.2.6 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
```

19.2.7 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.

Origin

This SPARC Annul Slot section was originally an email from Jiri Gaisler to Joel Sherrill that explained why sometimes, a single instruction will not be executed, due to the Annul Slot feature.

In SPARC, the default behaviour is to execute instructions after a branch. As with the behaviour of most RISC (Reduced Instruction Set Computer) machines, SPARC uses a branch delay slot. This is because completing an instruction every clock cycle introduces the problem that a branch may not be resolved until the instruction has passed through the pipeline. By inserting stalls, this is prevented. In each cycle, if a stall is inserted, it is considered one branch delay slot.

For example, a regular branch instruction might look like so:

```
cmp %o4, %g4    /* if %o4 is equals to %g4 */
be 200fd06      /* then branch */
mov [%g4], %o4  /* instructions after the branch, this is a */
                /* branch delay slot it is executed regardless */
                /* of whether %o4 is equals to %g4 */
```

However, if marked with ",a", the instructions after the branch will only be executed if the branch is taken. In other words, only if the condition before is true, then it would be executed. Otherwise if would be "annulled".

```
cmp %o4, %g4    /* if %o4 is equals to %g4 */
be,a 200fd06    /* then branch */
mov [%g4], %o4  /* instruction after the branch */
```

The mov instruction is in a branch delay slot and is only executed if the branch is taken (e.g. if %o4 is equals to %g4).

This shows up in analysis of coverage reports when a single instruction is marked unexecuted when the instruction above and below it are executed.

19.3 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.

19.3.1 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.

19.4 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.

19.4.1 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.

19.4.2 Trap Table

A SPARC processor uses a trap table to execute the trap handler associated with a trap. The trap table location is defined by the Trap Base Register (`TBR`). The trap table has 256 entries. Each entry has space for four instructions (16 bytes). RTEMS uses a statically initialized trap table. The start address of the trap table is associated with the `trap_table` global symbol. The first action of the system initialization (entry points `_start` and `hard_reset`) is to set the `TBR` to `trap_table`. The interrupt traps (trap numbers 16 to 31) are connected with the RTEMS interrupt handling. Some traps are connected to standard services defined by the SPARC architecture, for example the window overflow, underflow, and flush handling. Most traps are connected to a fatal error handler. The fatal error trap handler saves the processor context to an exception frame and starts the system termination procedure.

19.4.3 Vectoring of Interrupt Handler

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

- disables traps (sets the PSR.ET bit to 0 in the PSR),
- the PSR.S bit is copied into the Previous Supervisor Mode (PSR.PS) bit in 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 (PSR.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.

19.4.4 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.

19.4.5 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. Level fifteen (15) is a non-maskable interrupt (NMI), which makes it unsuitable for standard usage since it can affect the real-time behaviour by interrupting critical sections and spinlocks. Disabling traps stops also the NMI interrupt from happening. It can however be used for power-down or other critical events.

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.

Many LEON SPARC v7/v8 systems features an extended interrupt controller which adds an extra step of interrupt decoding to allow handling of interrupt 16-31. When such an extended interrupt is generated the CPU traps into a specific interrupt trap level 1-14 and software reads out from the interrupt controller which extended interrupt source actually caused the interrupt.

19.4.6 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 fifteen (15) before the execution of the section and restores them to the previous level upon completion of the section. RTEMS has been optimized to ensure 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.

Interrupts are disabled or enabled by performing a system call to the Operating System reserved software traps 9 (SPARC_SWTRAP_IRQDIS) or 10 (SPARC_SWTRAP_IRQEN). The trap is generated by the software trap (Ticc) instruction or indirectly by calling sparc_disable_interrupts() or sparc_enable_interrupts() functions. Disabling interrupts return the previous interrupt level (on trap entry) in register G1 and sets PSR.PIL to 15 to disable all maskable interrupts. The interrupt level can be restored by trapping into the enable interrupt handler with G1 containing the new interrupt level.

19.4.7 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.

The interrupt stack is statically allocated by RTEMS. There is one interrupt stack for each configured processor. The interrupt stack is used to initialize the system. The amount of memory allocated for the interrupt stack is determined by the `CONFIGURE_INTERRUPT_STACK_SIZE` application configuration option. As part of processing a non-nested interrupt, RTEMS will switch to the interrupt stack before invoking the installed handler.

19.5 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.

19.5.1 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.

If the BSP has been configured with `BSP_POWER_DOWN_AT_FATAL_HALT` set to true, the default handler will disable interrupts and enter power down mode. If power down mode is not available, it goes into an infinite loop to simulate a halt processor instruction.

If `BSP_POWER_DOWN_AT_FATAL_HALT` is set to false, the default handler will place the value `1` in register `g1`, the error source in register `g2`, and the error code in register``g3``. It will then generate a system error which will hand over control to the debugger, simulator, etc.

19.6 Symmetric Multiprocessing

SMP is supported. Available platforms are the Cobham Gaisler GR712RC and GR740.

19.7 Thread-Local Storage

Thread-local storage is supported.

19.8 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.

19.8.1 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.

19.8.2 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 C Applications Users 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.

19.9 Stacks and Register Windows

Credit

The contents of this section were originally written by Peter S. Magnusson and available through a website which is no longer online. Peter graciously granted permission for this information to be included in the RTEMS Documentation.

The SPARC architecture from Sun Microsystems has some "interesting" characteristics. After having to deal with both compiler, interpreter, OS emulator, and OS porting issues for the SPARC, I decided to gather notes and documentation in one place. If there are any issues you don't find addressed by this page, or if you know of any similar Net resources, let me know. This document is limited to the V8 version of the architecture.

19.9.1 General Structure

SPARC has 32 general purpose integer registers visible to the program at any given time. Of these, 8 registers are `global` registers and 24 registers are in a register window. A window consists of three groups of 8 registers, the `out`, `local`, and `in` registers. See table 1. A SPARC implementation can have from 2 to 32 windows, thus varying the number of registers from 40 to 520. Most implementations have 7 or 8 windows. The variable number of registers is the principal reason for the SPARC being "scalable".

At any given time, only one window is visible, as determined by the current window pointer (CWP) which is part of the processor status register (PSR). This is a five bit value that can be decremented or incremented by the `save` and `restore` instructions, respectively. These instructions are generally executed on procedure call and return (respectively). The idea is that the `in` registers contain incoming parameters, the `local` register constitutes scratch registers, the `out` registers contain outgoing parameters, and the `global` registers contain values that vary little between executions. The register windows overlap partially, thus the `out` registers become renamed by `save` to become the `in` registers of the called procedure. Thus, the memory traffic is reduced when going up and down the procedure call. Since this is a frequent operation, performance is improved.

(That was the idea, anyway. The drawback is that upon interactions with the system the registers need to be flushed to the stack, necessitating a long sequence of writes to memory of data that is often mostly garbage. Register windows was a bad idea that was caused by simulation studies that considered only programs in isolation, as opposed to multitasking workloads, and by considering compilers with poor optimization. It also caused considerable problems in implementing high-end SPARC processors such as the SuperSPARC, although more recent implementations have dealt effectively with the obstacles. Register windows are now part of the compatibility legacy and not easily removed from the architecture.)

Table 19.1: Table 1 - Visible Registers

Register Group	Mnemonic	Register Address
global	%g0-%g7	r[0] - r[7]
out	%o0-%o7	r[8] - r[15]
local	%l0-%l7	r[16] - r[23]
in	%i0-%i7	r[24] - r[31]

The overlap of the registers is illustrated in figure 1. The figure shows an implementation with 8 windows, numbered 0 to 7 (labeled w0 to w7 in the figure). Each window corresponds to 24 registers, 16 of which are shared with "neighboring" windows. The windows are arranged in a wrap-around manner, thus window number 0 borders window number 7. The common cause of changing the current window, as pointed to by CWP, is the `restore` and `save` instructions, shown in the middle. Less common is the supervisor `rett` instruction (return from trap) and the trap event (interrupt, exception, or `trap` instruction).

The "WIM" register is also indicated in the top left of Figure 1. The window invalid mask is a bit map of valid windows. It is generally used as a pointer, i.e. exactly one bit is set in the WIM register indicating which window is invalid (in the figure it's window 7). Register windows are generally used to support procedure calls, so they can be viewed as a cache of the stack contents. The WIM "pointer" indicates how many procedure calls in a row can be taken without writing out data to memory. In the figure, the capacity of the register windows is fully utilized. An additional call will thus exceed capacity, triggering a window overflow trap. At the other end, a window underflow trap occurs when the register window "cache" if empty and more data needs to be fetched from memory.

19.9.2 Register Semantics

The SPARC Architecture includes recommended software semantics. These are described in the architecture manual, the SPARC ABI (application binary interface) standard, and, unfortunately, in various other locations as well (including header files and compiler documentation).

Figure 2 shows a summary of register contents at any given time.

```
            %g0  (r00)       always zero
            %g1  (r01)  [1]  temporary value
            %g2  (r02)  [2]  global 2
global      %g3  (r03)  [2]  global 3
            %g4  (r04)  [2]  global 4
            %g5  (r05)       reserved for SPARC ABI
            %g6  (r06)       reserved for SPARC ABI
            %g7  (r07)       reserved for SPARC ABI

            %o0  (r08)  [3]  outgoing parameter 0 / return value from callee
            %o1  (r09)  [1]  outgoing parameter 1
            %o2  (r10)  [1]  outgoing parameter 2
```

(continues on next page)

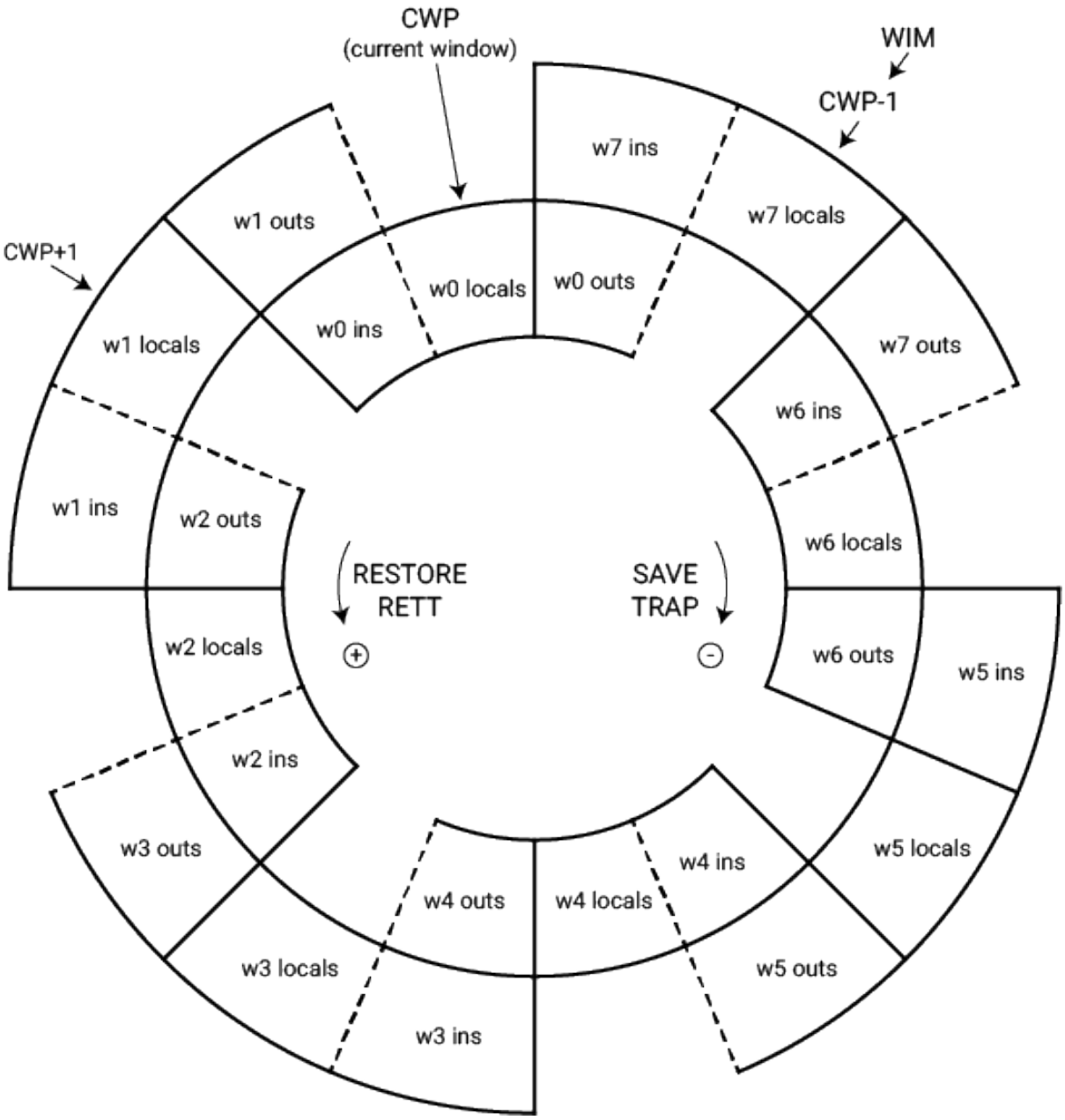

Fig. 19.1: Figure 1 - Windowed Registers

(continued from previous page)

```
out          %o3  (r11)  [1]  outgoing parameter 3
             %o4  (r12)  [1]  outgoing parameter 4
             %o5  (r13)  [1]  outgoing parameter 5
       %sp, %o6  (r14)  [1]  stack pointer
             %o7  (r15)  [1]  temporary value / address of CALL instruction

             %l0  (r16)  [3]  local 0
             %l1  (r17)  [3]  local 1
             %l2  (r18)  [3]  local 2
local        %l3  (r19)  [3]  local 3
             %l4  (r20)  [3]  local 4
             %l5  (r21)  [3]  local 5
             %l6  (r22)  [3]  local 6
             %l7  (r23)  [3]  local 7

             %i0  (r24)  [3]  incoming parameter 0 / return value to caller
             %i1  (r25)  [3]  incoming parameter 1
             %i2  (r26)  [3]  incoming parameter 2
in           %i3  (r27)  [3]  incoming parameter 3
             %i4  (r28)  [3]  incoming parameter 4
             %i5  (r29)  [3]  incoming parameter 5
       %fp, %i6  (r30)  [3]  frame pointer
             %i7  (r31)  [3]  return address - 8
```

Items

[1] assumed by caller to be destroyed (volatile) across a procedure call

[2] should not be used by SPARC ABI library code

[3] assumed by caller to be preserved across a procedure call

Figure 2 - SPARC register semantics

Particular compilers are likely to vary slightly.

Note that globals `%g2-%g4` are reserved for the "application", which includes libraries and compiler. Thus, for example, libraries may overwrite these registers unless they've been compiled with suitable flags. Also, the "reserved" registers are presumed to be allocated (in the future) bottom-up, i.e. `%g7` is currently the "safest" to use.

Optimizing linkers and interpreters are examples that use global registers.

19.9.3 Register Windows and the Stack

The SPARC register windows are, naturally, intimately related to the stack. In particular, the stack pointer (`%sp` or `%o6`) must always point to a free block of 64 bytes. This area is used by the operating system (Solaris, SunOS, and Linux at least) to save the current `local` and `in` registers upon a system interrupt, exception, or `trap` instruction. (Note that this can occur at any time.)

Other aspects of register relations with memory are programming convention. The typical and recommended layout of the stack is shown in figure 3. The figure shows a stack frame.

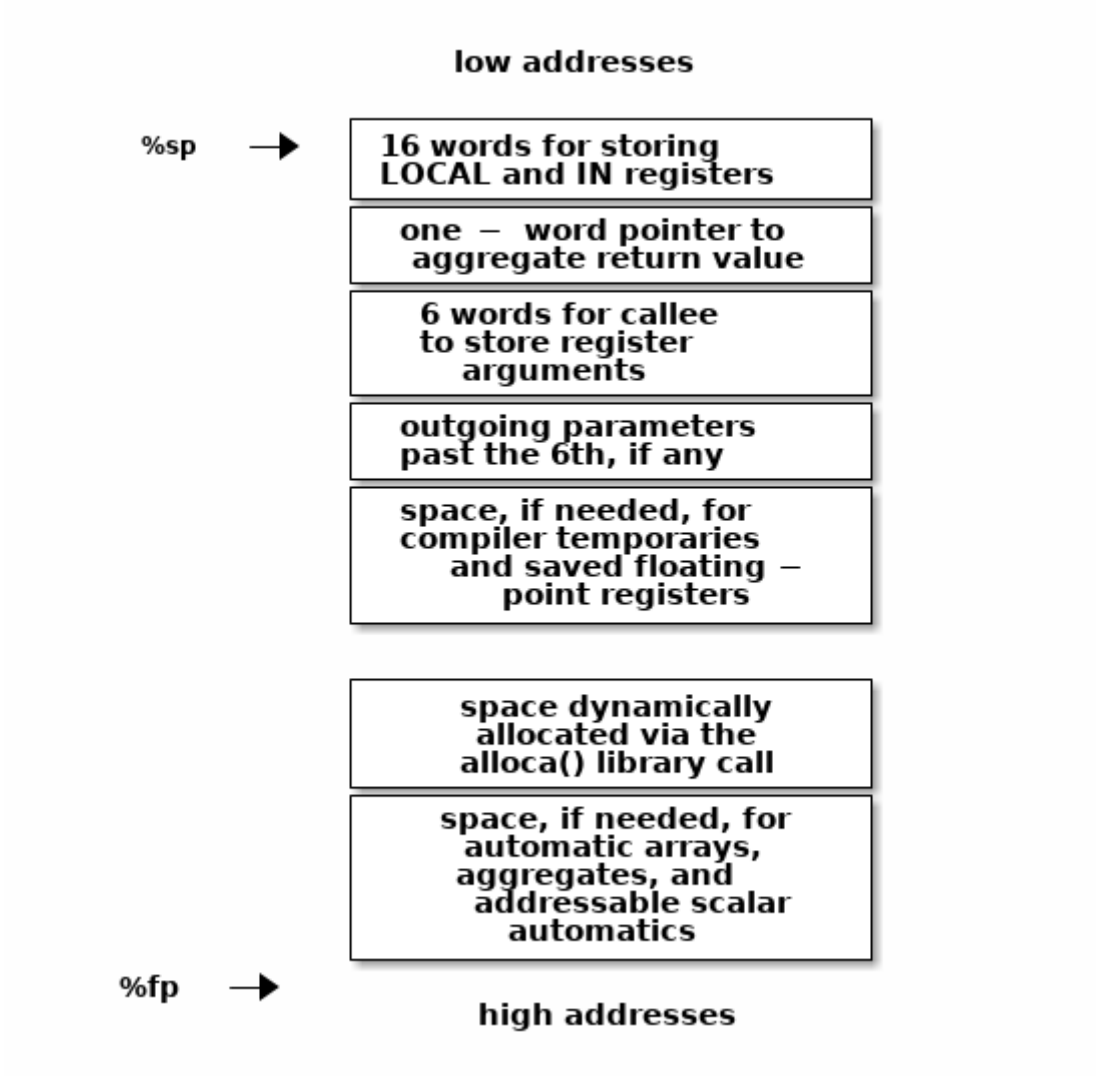

Fig. 19.2: Figure 3 - Stack frame contents

Note that the top boxes of figure 3 are addressed via the stack pointer (`%sp`), as positive offsets (including zero), and the bottom boxes are accessed over the frame pointer using negative offsets (excluding zero), and that the frame pointer is the old stack pointer. This scheme allows the separation of information known at compile time (number and size of local parameters, etc) from run-time information (size of blocks allocated by `alloca()`).

“addressable scalar automatics” is a fancy name for local variables.

The clever nature of the stack and frame pointers is that they are always 16 registers apart in the register windows. Thus, a `save` instruction will make the current stack pointer into the

frame pointer and, since the save instruction also doubles as an add, create a new stack pointer. Figure 4 illustrates what the top of a stack might look like during execution. (The listing is from the pwin command in the SimICS simulator.)

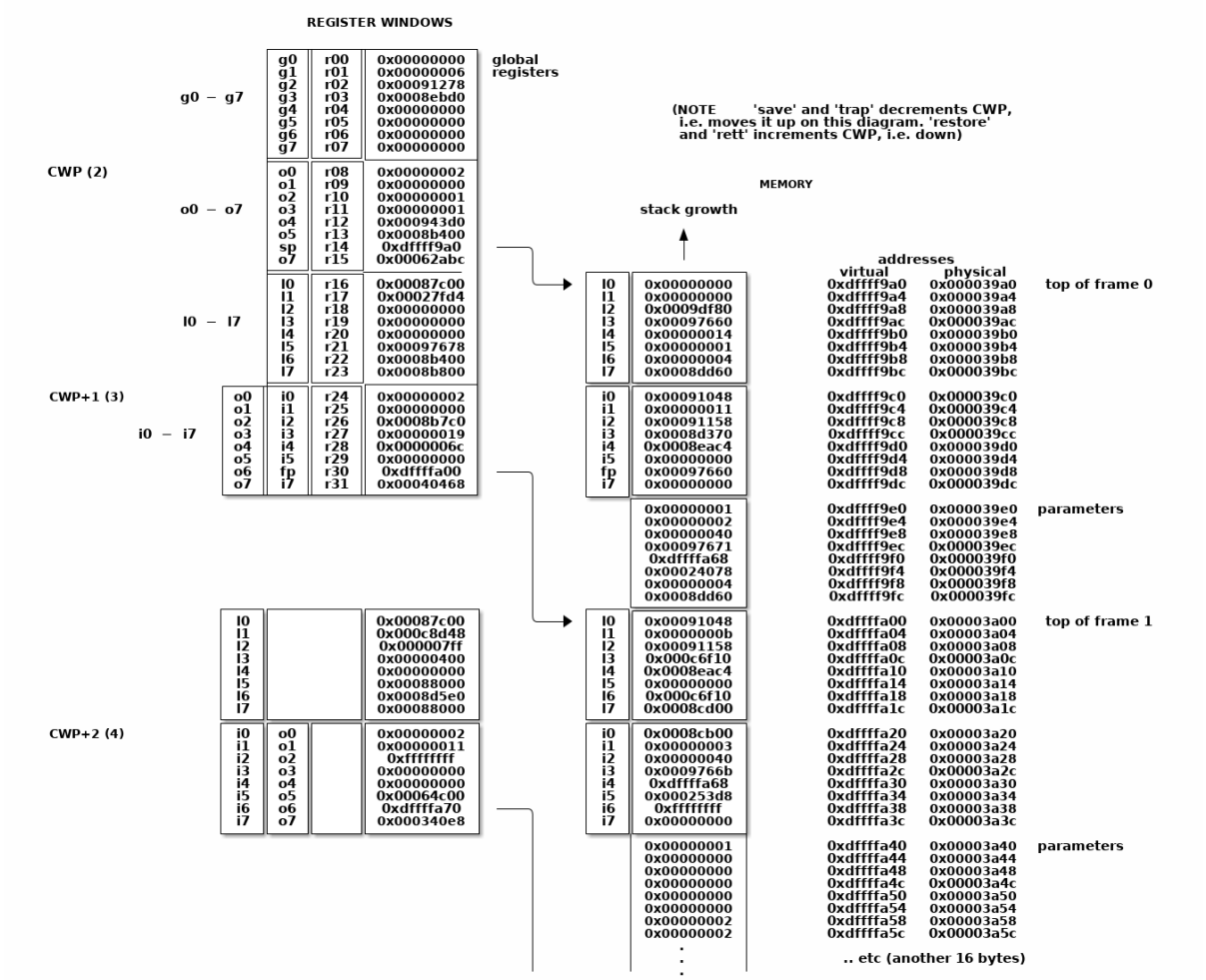

Fig. 19.3: Figure 4 - Sample stack contents

Note how the stack contents are not necessarily synchronized with the registers. Various events can cause the register windows to be “flushed” to memory, including most system calls. A programmer can force this update by using ST_FLUSH_WINDOWS trap, which also reduces the number of valid windows to the minimum of 1.

Writing a library for multithreaded execution is an example that requires explicit flushing, as is longjmp().

19.9.4 Procedure epilogue and prologue

The stack frame described in the previous section leads to the standard entry/exit mechanisms listed in figure 5.

```
function:
  save  %sp, -C, %sp

              ; perform function, leave return value,
              ; if any, in register %i0 upon exit

  ret         ; jmpl %i7+8, %g0
  restore     ; restore %g0,%g0,%g0
```

Figure 5 - Epilogue/prologue in procedures

The `save` instruction decrements the CWP, as discussed earlier, and also performs an addition. The constant `C` that is used in the figure to indicate the amount of space to make on the stack, and thus corresponds to the frame contents in Figure 3. The minimum is therefore the 16 words for the `local` and `in` registers, i.e. (hex) 0x40 bytes.

A confusing element of the `save` instruction is that the source operands (the first two parameters) are read from the old register window, and the destination operand (the rightmost parameter) is written to the new window. Thus, although `%sp` is indicated as both source and destination, the result is actually written into the stack pointer of the new window (the source stack pointer becomes renamed and is now the frame pointer).

The return instructions are also a bit particular. `ret` is a synthetic instruction, corresponding to `jmpl` (jump linked). This instruction jumps to the address resulting from adding 8 to the `%i7` register. The source instruction address (the address of the `ret` instruction itself) is written to the `%g0` register, i.e. it is discarded.

The `restore` instruction is similarly a synthetic instruction and is just a short form for a restore that chooses not to perform an addition.

The calling instruction, in turn, typically looks as follows:

```
call <function>    ; jmpl <address>, %o7
mov 0, %o0
```

Again, the `call` instruction is synthetic, and is actually the same instruction that performs the return. This time, however, it is interested in saving the return address, into register `%o7`. Note that the delay slot is often filled with an instruction related to the parameters, in this example it sets the first parameter to zero.

Note also that the return value is also generally passed in `%o0`.

Leaf procedures are different. A leaf procedure is an optimization that reduces unnecessary work by taking advantage of the knowledge that no `call` instructions exist in many procedures. Thus, the `save`/`restore` couple can be eliminated. The downside is that such a procedure may only use the `out` registers (since the `in` and `local` registers actually belong to the caller). See Figure 6.

```
function:
              ; no save instruction needed upon entry
```

(continues on next page)

(continued from previous page)

```
3
4               ; perform function, leave return value,
5               ; if any, in register %o0 upon exit
6
7   retl        ; jmpl %o7+8, %g0
8   nop         ; the delay slot can be used for something else
```

Figure 6 - Epilogue/prologue in leaf procedures

Note in the figure that there is only one instruction overhead, namely the `retl` instruction. `retl` is also synthetic (return from leaf subroutine), is again a variant of the `jmpl` instruction, this time with `%o7+8` as target.

Yet another variation of epilogue is caused by tail call elimination, an optimization supported by some compilers (including Sun's C compiler but not GCC). If the compiler detects that a called function will return to the calling function, it can replace its place on the stack with the called function. Figure 7 contains an example.

```
1     int
2       foo(int n)
3     {
4       if (n == 0)
5         return 0;
6       else
7         return bar(n);
8     }
9
10      cmp     %o0,0
11      bne     .L1
12      or      %g0,%o7,%g1
13      retl
14      or      %g0,0,%o0
15  .L1:  call    bar
16      or      %g0,%g1,%o7
```

Figure 7 - Example of tail call elimination

Note that the `call` instruction overwrites register `%o7` with the program counter. Therefore the above code saves the old value of `%o7`, and restores it in the delay slot of the `call` instruction. If the function `call` is register indirect, this twiddling with `%o7` can be avoided, but of course that form of `call` is slower on modern processors.

The benefit of tail call elimination is to remove an indirection upon return. It is also needed to reduce register window usage, since otherwise the `foo()` function in Figure 7 would need to allocate a stack frame to save the program counter.

A special form of tail call elimination is tail recursion elimination, which detects functions calling themselves, and replaces it with a simple branch. Figure 8 contains an example.

```
1     int
2       foo(int n)
3     {
```

(continues on next page)

(continued from previous page)

```
        if (n == 0)
          return 1;
        else
          return (foo(n - 1));
      }

      cmp     %o0,0
      be      .L1
      or      %g0,%o0,%g1
      subcc   %g1,1,%g1
.L2:  bne     .L2
      subcc   %g1,1,%g1
.L1:  retl
      or      %g0,1,%o0
```

Figure 8 - Example of tail recursion elimination

Needless to say, these optimizations produce code that is difficult to debug.

19.9.5 Procedures, stacks, and debuggers

When debugging an application, your debugger will be parsing the binary and consulting the symbol table to determine procedure entry points. It will also travel the stack frames "upward" to determine the current call chain.

When compiling for debugging, compilers will generate additional code as well as avoid some optimizations in order to allow reconstructing situations during execution. For example, GCC/GDB makes sure original parameter values are kept intact somewhere for future parsing of the procedure call stack. The live `in` registers other than `%i0` are not touched. `%i0` itself is copied into a free `local` register, and its location is noted in the symbol file. (You can find out where variables reside by using the `info address` command in GDB.)

Given that much of the semantics relating to stack handling and procedure call entry/exit code is only recommended, debuggers will sometimes be fooled. For example, the decision as to whether or not the current procedure is a leaf one or not can be incorrect. In this case a spurious procedure will be inserted between the current procedure and it's "real" parent. Another example is when the application maintains its own implicit call hierarchy, such as jumping to function pointers. In this case the debugger can easily become totally confused.

19.9.6 The window overflow and underflow traps

When the `save` instruction decrements the current window pointer (CWP) so that it coincides with the invalid window in the window invalid mask (WIM), a window overflow trap occurs. Conversely, when the `restore` or `rett` instructions increment the CWP to coincide with the invalid window, a window underflow trap occurs.

Either trap is handled by the operating system. Generally, data is written out to memory and/or read from memory, and the WIM register suitably altered.

The code in Figure 9 and Figure 10 below are bare-bones handlers for the two traps. The text is directly from the source code, and sort of works. (As far as I know, these are minimalistic

handlers for SPARC V8). Note that there is no way to directly access window registers other than the current one, hence the code does additional `save`/`restore` instructions. It's pretty tricky to understand the code, but figure 1 should be of help.

```
      /* a SAVE instruction caused a trap */
window_overflow:
      /* rotate WIM on bit right, we have 8 windows */
      mov %wim,%l3
      sll %l3,7,%l4
      srl %l3,1,%l3
      or  %l3,%l4,%l3
      and %l3,0xff,%l3

      /* disable WIM traps */
      mov %g0,%wim
      nop; nop; nop

      /* point to correct window */
      save

      /* dump registers to stack */
      std %l0, [%sp +  0]
      std %l2, [%sp +  8]
      std %l4, [%sp + 16]
      std %l6, [%sp + 24]
      std %i0, [%sp + 32]
      std %i2, [%sp + 40]
      std %i4, [%sp + 48]
      std %i6, [%sp + 56]

      /* back to where we should be */
      restore

      /* set new value of window */
      mov %l3,%wim
      nop; nop; nop

      /* go home */
      jmp %l1
      rett %l2
```

Figure 9 - window_underflow trap handler

```
      /* a RESTORE instruction caused a trap */
window_underflow:

      /* rotate WIM on bit LEFT, we have 8 windows */
      mov %wim,%l3
      srl %l3,7,%l4
      sll %l3,1,%l3
      or  %l3,%l4,%l3
```

(continues on next page)

(continued from previous page)

```
      and %l3,0xff,%l3

      /* disable WIM traps */
      mov %g0,%wim
      nop; nop; nop

      /* point to correct window */
      restore
      restore

      /* dump registers to stack */
      ldd [%sp +  0], %l0
      ldd [%sp +  8], %l2
      ldd [%sp + 16], %l4
      ldd [%sp + 24], %l6
      ldd [%sp + 32], %i0
      ldd [%sp + 40], %i2
      ldd [%sp + 48], %i4
      ldd [%sp + 56], %i6

      /* back to where we should be */
      save
      save

      /* set new value of window */
      mov %l3,%wim
      nop; nop; nop

      /* go home */
      jmp %l1
      rett %l2
```

Figure 10 - window_underflow trap handler

CHAPTER

TWENTY

SPARC-64 SPECIFIC INFORMATION

This document discusses the SPARC Version 9 (aka SPARC-64, SPARC64 or SPARC V9) architecture dependencies in this port of RTEMS.

The SPARC V9 architecture leaves a lot of undefined implemenation dependencies which are defined by the processor models. Consult the specific CPU model section in this document for additional documents covering the implementation dependent architectural features.

sun4u Specific Information

sun4u is the subset of the SPARC V9 implementations comprising the UltraSPARC I through UltraSPARC IV processors.

The following documents were used in developing the SPARC-64 sun4u port:

- UltraSPARC User's Manual (http://www.sun.com/microelectronics/manuals/ultrasparc/802-7220-02.pdf)
- UltraSPARC IIIi Processor (http://datasheets.chipdb.org/Sun/UltraSparc-IIIi.pdf)

sun4v Specific Information

sun4v is the subset of the SPARC V9 implementations comprising the UltraSPARC T1 or T2 processors.

The following documents were used in developing the SPARC-64 sun4v port:

- UltraSPARC Architecture 2005 Specification (http://opensparc-t1.sunsource.net/specs/UA2005-current-draft-P-EXT.pdf)
- UltraSPARC T1 supplement to UltraSPARC Architecture 2005 Specification (http://opensparc-t1.sunsource.net/specs/UST1-UASuppl-current-draft-P-EXT.pdf)

The defining feature that separates the sun4v architecture from its predecessor is the existence of a super-privileged hypervisor that is responsible for providing virtualized execution environments. The impact of the hypervisor on the real-time guarantees available with sun4v has not yet been determined.

20.1 CPU Model Dependent Features

20.1.1 CPU Model Feature Flags

This section presents the set of features which vary across SPARC-64 implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file cpukit/score/cpu/sparc64/sparc64.h based upon the particular CPU model defined on the compilation command line.

20.1.1.1 CPU Model Name

The macro CPU MODEL NAME is a string which designates the name of this CPU model. For example, for the UltraSPARC T1 SPARC V9 model, this macro is set to the string “sun4v”.

20.1.1.2 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.

20.1.1.3 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 for a maximum of thirty-two register window sets although most implementations only include eight.

20.1.2 CPU Model Implementation Notes

This section describes the implemenation dependencies of the CPU Models sun4u and sun4v of the SPARC V9 architecture.

20.1.2.1 sun4u Notes

XXX

20.1.3 sun4v Notes

XXX

20.2 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 following document also provides some conventions on the global register usage in SPARC V9: http://developers.sun.com/solaris/articles/sparcv9abi.html

20.2.1 Programming Model

This section discusses the programming model for the SPARC architecture.

20.2.1.1 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 Name	Description
g0	na	reads return 0, writes are ignored
o6	sp	stack pointer
i6	fp	frame pointer
i7	na	return address

20.2.1.2 Floating Point Registers

The SPARC V9 architecture includes sixty-four, thirty-two bit registers. These registers may be viewed as follows:

- 32 32-bit single precision floating point or integer registers (f0, f1, ... f31)
- 32 64-bit double precision floating point registers (f0, f2, f4, ... f62)
- 16 128-bit extended precision floating point registers (f0, f4, f8, ... f60)

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

20.2.1.3 Special Registers

The SPARC architecture includes a number of special registers:

``***Ancillary State Registers (ASRs)***``
The ancillary state registers (ASRs) are optional state registers that may be privileged or nonprivileged. ASRs 16-31 are implementation- dependent. The SPARC V9 ASRs include: y, ccr, asi, tick, pc, fprs. The sun4u ASRs include: pcr, pic, dcr, gsr, softint set, softint clr, softint, and tick cmpr. The sun4v ASRs include: pcr, pic, gsr, soft- int set, softint clr, softint, tick cmpr, stick, and stick cmpr.

``***Processor State Register (pstate)***``
The privileged pstate register contains control fields for the proces- sor's current state. Its flag fields include the interrupt enable, privi- leged mode, and enable FPU.

``***Processor Interrupt Level (pil)***``
The PIL specifies the interrupt level above which interrupts will be accepted.

``***Trap Registers***``
The trap handling mechanism of the SPARC V9 includes a number of registers, including: trap program counter (tpc), trap next pc (tnpc), trap state (tstate), trap type (tt), trap base address (tba), and trap level (tl).

``***Alternate Globals***``
The AG bit of the pstate register provides access to an alternate set of global registers. On sun4v, the AG bit is replaced by the global level (gl) register, providing access to at least two and at most eight alternate sets of globals.

``***Register Window registers***``
A number of registers assist in register window management. These include the current window pointer (cwp), savable windows (cansave), restorable windows (canrestore), clean windows (clean- win), other windows (otherwin), and window state (wstate).

20.2.2 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 increments 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 decrements 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 cansave, canrestore, otherwin, and cwp are used in conjunction to manage the finite set of register windows and detect the window overflow and underflow conditions. The first three of these registers must satisfy the invariant cansave + canrestore + otherwin = nwindow - 2, where nwindow is the number of register windows. The cwp contains the index of the register window currently in use. RTEMS does not use the cleanwin and otherwin registers.

The save instruction increments the cwp modulo the number of register windows, and if cansave is 0 then it also generates a window overflow. Similarly, the restore instruction decrements the cwp modulo the number of register windows, and if canrestore is 0 then it also generates a window underflow.

Unlike with the SPARC model, the SPARC-64 port does not assume that a register window is available for a trap. The window overflow and underflow conditions are not detected without hardware generating the trap. (These conditions can be detected by reading the register window registers and doing some simple arithmetic.)

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.

20.2.3 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. This allows for the compiler to generate leaf-optimized functions that utilize the caller's output registers without using save and restore.

20.2.4 Calling Mechanism

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

20.2.5 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.

20.2.6 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
```

20.2.7 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.

20.3 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.

20.3.1 Flat Memory Model

The SPARC-64 architecture supports a flat 64-bit address space with addresses ranging from 0x0000000000000000 to 0xFFFFFFFFFFFFFFFF. Each address is represented by a 64-bit value (and an 8-bit address space identifider or ASI) and is byte addressable. The address may be used to reference a single byte, half-word (2-bytes), word (4 bytes), doubleword (8 bytes), or quad-word (16 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 0x34). 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
quadword	16

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

20.4 Interrupt Processing

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. Note that in the SPARC manuals, interrupts are a subset of the traps that are delivered to software interrupt handlers.

20.4.1 Synchronous Versus Asynchronous Traps

The SPARC architecture includes two classes of traps: synchronous (precise) and asynchronous (deferred). 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.

20.4.2 Vectoring of Interrupt Handler

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

- The trap level is set. This provides access to a fresh set of privileged trap-state registers used to save the current state, in effect, pushing a frame on the trap stack. TL <- TL + 1
- Existing state is preserved - TSTATE[TL].CCR <- CCR - TSTATE[TL].ASI <- ASI - TSTATE[TL].PSTATE <- PSTATE - TSTATE[TL].CWP <- CWP - TPC[TL] <- PC - TNPC[TL] <- nPC
- The trap type is preserved. TT[TL] <- the trap type
- The PSTATE register is updated to a predefined state - PSTATE.MM is unchanged - PSTATE.RED <- 0 - PSTATE.PEF <- 1 if FPU is present, 0 otherwise - PSTATE.AM <- 0 (address masking is turned off) - PSTATE.PRIV <- 1 (the processor enters privileged mode) - PSTATE.IE <- 0 (interrupts are disabled) - PSTATE.AG <- 1 (global regs are replaced with alternate globals) - PSTATE.CLE <- PSTATE.TLE (set endian mode for traps)
- For a register-window trap only, CWP is set to point to the register window that must be accessed by the trap-handler software, that is:
  - If TT[TL] = 0x24 (a clean window trap), then CWP <- CWP + 1.
  - If (0x80 <= TT[TL] <= 0xBF) (window spill trap), then CWP <- CWP + CANSAVE + 2.
  - If (0xC0 <= TT[TL] <= 0xFF) (window fill trap), then CWP <- CWP1.
  - For non-register-window traps, CWP is not changed.
- Control is transferred into the trap table:

  - PC <- TBA<63:15> (TL>0) TT[TL] 0 0000
  - nPC <- TBA<63:15> (TL>0) TT[TL] 0 0100
  - where (TL>0) is 0 if TL = 0, and 1 if TL > 0.

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,
- switches the processor to trap level 0,
- 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 may 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.

20.4.3 Traps and Register Windows

XXX

20.4.4 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.

20.4.5 Disabling of Interrupts by RTEMS

XXX

20.4.6 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.

20.5 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.

20.5.1 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.

20.6 Symmetric Multiprocessing

SMP is not supported.

20.7 Thread-Local Storage

Thread-local storage is supported.

20.8 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.

20.8.1 HelenOS and Open Firmware

The provided BSPs make use of some bootstrap and low-level hardware code of the HelenOS operating system. These files can be found in the shared/helenos directory of the sparc64 bsp directory. Consult the sources for more detailed information.

The shared BSP code also uses the Open Firmware interface to re-use firmware code, primarily for console support and default trap handlers.

BIBLIOGRAPHY

[Dre13] Ulrich Drepper. *ELF Handling For Thread-Local Storage*. 2013. URL: http://www.akkadia.org/drepper/tls.pdf.