RTEMS SPARC Applications Supplement

Edition 4.6.99.3, for RTEMS 4.6.99.3

19 September 2003

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TeXinfo 2005-07-05.19

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

Table of Contents

Preface

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

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

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

SPARC Architecture Documents

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

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

ERC32 Specific Information

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

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

- ERC32 System Design Document
- MEC Device Specification

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

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

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

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

1 CPU Model Dependent Features

1.1 Introduction

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.

1.2 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.

1.2.1 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".

1.2.2 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.

1.2.3 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.

1.2.4 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.

1.2.5 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.

1.3 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.

2 Calling Conventions

2.1 Introduction

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

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

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

2.2 Programming Model

This section discusses the programming model for the SPARC architecture.

2.2.1 Non-Floating Point Registers

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

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

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

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

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

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

2.2.2 Floating Point Registers

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

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

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

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

2.2.3 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.

2.3 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.

2.4 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.

2.5 Calling Mechanism

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

2.6 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.

2.7 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
```

2.8 User-Provided Routines

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

3 Memory Model

3.1 Introduction

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

3.2 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.

4 Interrupt Processing

4.1 Introduction

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.

4.2 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.

4.3 Vectoring of Interrupt Handler

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

- disables traps (sets the ET bit of the psr to 0),
- the S bit of the psr is copied into the Previous Supervisor Mode (PS) bit of the psr,
- the cwp is decremented by one (modulo the number of register windows) to activate a trap window,
- the PC and nPC are loaded into local register 1 and 2 (l0 and l1),
- the trap type (tt) field of the Trap Base Register (TBR) is set to the appropriate value, and

- if the trap is not a reset, then the PC is written with the contents of the TBR and the nPC is written with TBR + 4. If the trap is a reset, then the PC is set to zero and the nPC is set to 4.

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

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

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

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

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

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.

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.

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.

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 seven (15) before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than TBD microseconds on a 15.0 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 4.2.0-prerelease.]

[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.

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.

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

5 Default Fatal Error Processing

5.1 Introduction

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

5.2 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.

6 Board Support Packages

6.1 Introduction

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.

6.2 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.

6.3 Processor Initialization

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

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

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

- Must leave the S bit of the status register set so that the SPARC remains in the supervisor state.
- Must set stack pointer (sp) such that a minimum stack size of MINIMUM_STACK_SIZE bytes is provided for the `rtems_initialize_executive` directive.
- Must disable all external interrupts (i.e. set the pil to 15).

- Must enable traps so window overflow and underflow conditions can be properly handled.
- Must initialize the SPARC's initial trap table with at least trap handlers for register window overflow and register window underflow.

7 Processor Dependent Information Table

7.1 Introduction

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

7.2 CPU Dependent Information Table

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

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

} rtems_cpu_table;
```

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

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

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

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

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

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

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

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

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

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

8 Memory Requirements

8.1 Introduction

Memory is typically a limited resource in real-time embedded systems, therefore, RTEMS can be configured to utilize the minimum amount of memory while meeting all of the applications requirements. Worksheets are provided which allow the RTEMS application developer to determine the amount of RTEMS code and RAM workspace which is required by the particular configuration. Also provided are the minimum code space, maximum code space, and the constant data space required by RTEMS.

8.2 Data Space Requirements

RTEMS requires a small amount of memory for its private variables. This data area must be in RAM and is separate from the RTEMS RAM Workspace. The following illustrates the data space required for all configurations of RTEMS:

- Data Space: 9059

8.3 Minimum and Maximum Code Space Requirements

A maximum configuration of RTEMS includes the core and all managers, including the multiprocessing manager. Conversely, a minimum configuration of RTEMS includes only the core and the following managers: initialization, task, interrupt and fatal error. The following illustrates the code space required by these configurations of RTEMS:

- Minimum Configuration: 28,288
- Maximum Configuration: 50,432

8.4 RTEMS Code Space Worksheet

The RTEMS Code Space Worksheet is a tool provided to aid the RTEMS application designer to accurately calculate the memory required by the RTEMS run-time environment. RTEMS allows the custom configuration of the executive by optionally excluding managers which are not required by a particular application. This worksheet provides the included and excluded size of each manager in tabular form allowing for the quick calculation of any custom configuration of RTEMS. The RTEMS Code Space Worksheet is below:

RTEMS Code Space Worksheet

Component	Included	Not Included	Size
Core	20,336	NA	
Initialization	1,408	NA	
Task	4,496	NA	
Interrupt	72	NA	
Clock	576	NA	
Timer	1,336	208	
Semaphore	1,888	192	
Message	2,032	320	
Event	1,696	64	
Signal	664	64	
Partition	1,368	152	
Region	1,736	176	
Dual Ported Memory	872	152	
I/O	1,144	00	
Fatal Error	32	NA	
Rate Monotonic	1,656	208	
Multiprocessing	8,328	408	
Total Code Space Requirements			

8.5 RTEMS RAM Workspace Worksheet

The RTEMS RAM Workspace Worksheet is a tool provided to aid the RTEMS application designer to accurately calculate the minimum memory block to be reserved for RTEMS use. This worksheet provides equations for calculating the amount of memory required based upon the number of objects configured, whether for single or multiple processor versions of the executive. This information is presented in tabular form, along with the fixed system requirements, allowing for quick calculation of any application defined configuration of RTEMS. The RTEMS RAM Workspace Worksheet is provided below:

RTEMS RAM Workspace Worksheet

Description	**Equation**	**Bytes Required**
maximum_tasks	* 488 =	
maximum_timers	* 68 =	
maximum_semaphores	* 124 =	
maximum_message_queues	* 148 =	
maximum_regions	* 144 =	
maximum_partitions	* 56 =	
maximum_ports	* 36 =	
maximum_periods	* 36 =	
maximum_extensions	* 64 =	
Floating Point Tasks	* 136 =	
Task Stacks	=	
Total Single Processor Requirements		
Description	**Equation**	**Bytes Required**
maximum_nodes	* 48 =	
maximum_global_objects	* 20 =	
maximum_proxies	* 124 =	
Total Multiprocessing Requirements		
Fixed System Requirements	10,072	
Total Single Processor Requirements		
Total Multiprocessing Requirements		
Minimum Bytes for RTEMS Workspace		

9 Timing Specification

9.1 Introduction

This chapter provides information pertaining to the measurement of the performance of RTEMS, the methods of gathering the timing data, and the usefulness of the data. Also discussed are other time critical aspects of RTEMS that affect an applications design and ultimate throughput. These aspects include determinancy, interrupt latency and context switch times.

9.2 Philosophy

Benchmarks are commonly used to evaluate the performance of software and hardware. Benchmarks can be an effective tool when comparing systems. Unfortunately, benchmarks can also be manipulated to justify virtually any claim. Benchmarks of real-time executives are difficult to evaluate for a variety of reasons. Executives vary in the robustness of features and options provided. Even when executives compare favorably in functionality, it is quite likely that different methodologies were used to obtain the timing data. Another problem is that some executives provide times for only a small subset of directives, This is typically justified by claiming that these are the only time-critical directives. The performance of some executives is also very sensitive to the number of objects in the system. To obtain any measure of usefulness, the performance information provided for an executive should address each of these issues.

When evaluating the performance of a real-time executive, one typically considers the following areas: determinancy, directive times, worst case interrupt latency, and context switch time. Unfortunately, these areas do not have standard measurement methodologies. This allows vendors to manipulate the results such that their product is favorably represented. We have attempted to provide useful and meaningful timing information for RTEMS. To insure the usefulness of our data, the methodology and definitions used to obtain and describe the data are also documented.

9.2.1 Determinancy

The correctness of data in a real-time system must always be judged by its timeliness. In many real-time systems, obtaining the correct answer does not necessarily solve the problem. For example, in a nuclear reactor it is not enough to determine that the core is overheating. This situation must be detected and acknowledged early enough that corrective action can be taken and a meltdown avoided.

Consequently, a system designer must be able to predict the worst-case behavior of the application running under the selected executive. In this light, it is important that a real-time system perform consistently regardless of the number of tasks, semaphores, or other resources allocated. An important design goal of a real-time executive is that all internal algorithms be fixed-cost. Unfortunately, this goal is difficult to completely meet without sacrificing the robustness of the executive's feature set.

Many executives use the term deterministic to mean that the execution times of their services can be predicted. However, they often provide formulas to modify execution times based upon the number of objects in the system. This usage is in sharp contrast to the notion of deterministic meaning fixed cost.

Almost all RTEMS directives execute in a fixed amount of time regardless of the number of objects present in the system. The primary exception occurs when a task blocks while acquiring a resource and specifies a non-zero timeout interval.

Other exceptions are message queue broadcast, obtaining a variable length memory block, object name to ID translation, and deleting a resource upon which tasks are waiting. In addition, the time required to service a clock tick interrupt is based upon the number of timeouts and other "events" which must be processed at that tick. This second group is composed primarily of capabilities which are inherently non-deterministic but are infrequently used in time critical situations. The major exception is that of servicing a clock tick. However, most applications have a very small number of timeouts which expire at exactly the same millisecond (usually none, but occasionally two or three).

9.2.2 Interrupt Latency

Interrupt latency is the delay between the CPU's receipt of an interrupt request and the execution of the first application-specific instruction in an interrupt service routine. Interrupts are a critical component of most real-time applications and it is critical that they be acted upon as quickly as possible.

Knowledge of the worst case interrupt latency of an executive aids the application designer in determining the maximum period of time between the generation of an interrupt and an interrupt handler responding to that interrupt. The interrupt latency of an system is the greater of the executive's and the applications's interrupt latency. If the application disables interrupts longer than the executive, then the application's interrupt latency is the system's worst case interrupt disable period.

The worst case interrupt latency for a real-time executive is based upon the following components:

- the longest period of time interrupts are disabled by the executive,
- the overhead required by the executive at the beginning of each ISR,
- the time required for the CPU to vector the interrupt, and
- for some microprocessors, the length of the longest instruction.

The first component is irrelevant if an interrupt occurs when interrupts are enabled, although it must be included in a worst case analysis. The third and fourth components are particular to a CPU implementation and are not dependent on the executive. The fourth component is ignored by this document because most applications use only a subset of a microprocessor's instruction set. Because of this the longest instruction actually executed is application dependent. The worst case interrupt latency of an executive is typically defined as the sum of components (1) and (2). The second component includes the time necessry for RTEMS to save registers and vector to the user-defined handler. RTEMS includes the third component, the time required for the CPU to vector the interrupt, because it is a required part of any interrupt.

Many executives report the maximum interrupt disable period as their interrupt latency and ignore the other components. This results in very low worst-case interrupt latency times which are not indicative of actual application performance. The definition used by RTEMS results in a higher interrupt latency being reported, but accurately reflects the longest delay between the CPU's receipt of an interrupt request and the execution of the first application-specific instruction in an interrupt service routine.

The actual interrupt latency times are reported in the Timing Data chapter of this supplement.

9.2.3 Context Switch Time

An RTEMS context switch is defined as the act of taking the CPU from the currently executing task and giving it to another task. This process involves the following components:

- Saving the hardware state of the current task.
- Optionally, invoking the TASK_SWITCH user extension.
- Restoring the hardware state of the new task.

RTEMS defines the hardware state of a task to include the CPU's data registers, address registers, and, optionally, floating point registers.

Context switch time is often touted as a performance measure of real-time executives. However, a context switch is performed as part of a directive's actions and should be viewed as such when designing an application. For example, if a task is unable to acquire a semaphore and blocks, a context switch is required to transfer control from the blocking task to a new task. From the application's perspective, the context switch is a direct result of not acquiring the semaphore. In this light, the context switch time is no more relevant than the performance of any other of the executive's subroutines which are not directly accessible by the application.

In spite of the inappropriateness of using the context switch time as a performance metric, RTEMS context switch times for floating point and non-floating points tasks are provided for comparison purposes. Of the executives which actually support floating point operations, many do not report context switch times for floating point context switch time. This results in a reported context switch time which is meaningless for an application with floating point tasks.

The actual context switch times are reported in the Timing Data chapter of this supplement.

9.2.4 Directive Times

Directives are the application's interface to the executive, and as such their execution times are critical in determining the performance of the application. For example, an application using a semaphore to protect a critical data structure should be aware of the time required to acquire and release a semaphore. In addition, the application designer can utilize the directive execution times to evaluate the performance of different synchronization and communication mechanisms.

The actual directive execution times are reported in the Timing Data chapter of this supplement.

9.3 Methodology

9.3.1 Software Platform

The RTEMS timing suite is written in C. The overhead of passing arguments to RTEMS by C is not timed. The times reported represent the amount of time from entering to exiting RTEMS.

The tests are based upon one of two execution models: (1) single invocation times, and (2) average times of repeated invocations. Single invocation times are provided for directives which cannot easily be invoked multiple times in the same scenario. For example, the times reported for entering and exiting an interrupt service routine are single invocation times. The second model is used for directives which can easily be invoked multiple times in the same scenario. For example, the times reported for semaphore obtain and semaphore release are averages of multiple invocations. At least 100 invocations are used to obtain the average.

9.3.2 Hardware Platform

Since RTEMS supports a variety of processors, the hardware platform used to gather the benchmark times must also vary. Therefore, for each processor supported the hardware platform must be defined. Each definition will include a brief description of the target hardware platform including the clock speed, memory wait states encountered, and any other pertinent information. This definition may be found in the processor dependent timing data chapter within this supplement.

9.3.3 What is measured?

An effort was made to provide execution times for a large portion of RTEMS. Times were provided for most directives regardless of whether or not they are typically used in time critical code. For example, execution times are provided for all object create and delete directives, even though these are typically part of application initialization.

The times include all RTEMS actions necessary in a particular scenario. For example, all times for blocking directives include the context switch necessary to transfer control to a new task. Under no circumstances is it necessary to add context switch time to the reported times.

The following list describes the objects created by the timing suite:

- All tasks are non-floating point.
- All tasks are created as local objects.
- No timeouts are used on blocking directives.
- All tasks wait for objects in FIFO order.

In addition, no user extensions are configured.

9.3.4 What is not measured?

The times presented in this document are not intended to represent best or worst case times, nor are all directives included. For example, no times are provided for the initialize executive and fatal_error_occurred directives. Other than the exceptions detailed in the Determinancy section, all directives will execute in the fixed length of time given.

Other than entering and exiting an interrupt service routine, all directives were executed from tasks and not from interrupt service routines. Directives invoked from ISRs, when allowable, will execute in slightly less time than when invoked from a task because rescheduling is delayed until the interrupt exits.

9.3.5 Terminology

The following is a list of phrases which are used to distinguish individual execution paths of the directives taken during the RTEMS performance analysis:

another task The directive was performed on a task other than the calling task.

available A task attempted to obtain a resource and immediately acquired it.

blocked task The task operated upon by the directive was blocked waiting for a resource.

caller blocks The requested resoure was not immediately available and the calling task chose to wait.

calling task The task invoking the directive.

messages flushed One or more messages was flushed from the message queue.

no messages flushed No messages were flushed from the message queue.

not available A task attempted to obtain a resource and could not immediately acquire it.

no reschedule The directive did not require a rescheduling operation.

NO_WAIT A resource was not available and the calling task chose to return immediately via the NO_WAIT option with an error.

obtain current The current value of something was requested by the calling task.

preempts caller The release of a resource caused a task of higher priority than the calling to be readied and it became the executing task.

ready task The task operated upon by the directive was in the ready state.

reschedule The actions of the directive necessitated a rescheduling operation.

returns to caller The directive succeeded and immediately returned to the calling task.

returns to interrupted task
The instructions executed immediately following this interrupt will be in the interrupted task.

returns to nested interrupt
The instructions executed immediately following this interrupt will be in a previously interrupted ISR.

returns to preempting task
The instructions executed immediately following this interrupt or signal handler will be in a task other than the interrupted task.

signal to self The signal set was sent to the calling task and signal processing was enabled.

suspended task The task operated upon by the directive was in the suspended state.

task readied The release of a resource caused a task of lower or equal priority to be readied and the calling task remained the executing task.

yield The act of attempting to voluntarily release the CPU.

10 ERC32 Timing Data

10.1 Introduction

The timing data for RTEMS on the ERC32 implementation of the SPARC architecture is provided along with the target dependent aspects concerning the gathering of the timing data. The hardware platform used to gather the times is described to give the reader a better understanding of each directive time provided. Also, provided is a description of the interrupt latency and the context switch times as they pertain to the SPARC version of RTEMS.

10.2 Hardware Platform

All times reported in this chapter were measured using the SPARC Instruction Simulator (SIS) developed by the European Space Agency. SIS simulates the ERC32 – a custom low power implementation combining the Cypress 90C601 integer unit, the Cypress 90C602 floating point unit, and a number of peripherals such as counter timers, interrupt controller and a memory controller.

For the RTEMS tests, SIS is configured with the following characteristics:

- 15 Mhz clock speed
- 0 wait states for PROM accesses
- 0 wait states for RAM accesses

The ERC32's General Purpose Timer was used to gather all timing information. This timer was programmed to operate with one microsecond accuracy. All sources of hardware interrupts were disabled, although traps were enabled and the interrupt level of the SPARC allows all interrupts.

10.3 Interrupt Latency

The maximum period with traps disabled or the processor interrupt level set to it's highest value inside RTEMS is less than TBD microseconds including the instructions which disable and re-enable interrupts. The time required for the ERC32 to vector an interrupt and for the RTEMS entry overhead before invoking the user's trap handler are a total of 8 microseconds. These combine to yield a worst case interrupt latency of less than TBD + 8 microseconds at 15.0 Mhz. [NOTE: The maximum period with interrupts disabled was last determined for Release 4.2.0-prerelease.]

The maximum period with interrupts disabled within RTEMS is hand-timed with some assistance from SIS. The maximum period with interrupts disabled with RTEMS occurs during a context switch when traps are disabled to flush all the register windows to memory. The length of time spent flushing the register windows varies based on the number of windows which must be flushed. Based on the information reported by SIS, it takes from 4.0 to 18.0 microseconds (37 to 122 instructions) to flush the register windows. It takes approximately 41 CPU cycles (2.73 microseconds) to flush each register window set to memory. The register window flush operation is heavily memory bound.

[NOTE: All traps are disabled during the register window flush thus disabling both software generate traps and external interrupts. During a normal RTEMS critical section, the processor interrupt level (pil) is raised to level 15 and traps are left enabled. The longest path for a normal critical section within RTEMS is less than 50 instructions.]

The interrupt vector and entry overhead time was generated on the SIS benchmark platform using the ERC32's ability to forcibly generate an arbitrary interrupt as the source of the "benchmark" interrupt.

10.4 Context Switch

The RTEMS processor context switch time is 10 microseconds on the SIS benchmark platform when no floating point context is saved or restored. Additional execution time is required when a TASK_SWITCH user extension is configured. The use of the TASK_SWITCH extension is application dependent. Thus, its execution time is not considered part of the raw context switch time.

Since RTEMS was designed specifically for embedded missile applications which are floating point intensive, the executive is optimized to avoid unnecessarily saving and restoring the state of the numeric coprocessor. The state of the numeric coprocessor is only saved when an FLOATING_POINT task is dispatched and that task was not the last task to utilize the coprocessor. In a system with only one FLOATING_POINT task, the state of the numeric coprocessor will never be saved or restored. When the first FLOATING_POINT task is dispatched, RTEMS does not need to save the current state of the numeric coprocessor.

The following table summarizes the context switch times for the ERC32 benchmark platform:

No Floating Point Contexts	21
Floating Point Contexts	
restore first FP task	26
save initialized, restore initialized	24
save idle, restore initialized	23
save idle, restore idle	33

10.5 Directive Times

This sections is divided into a number of subsections, each of which contains a table listing the execution times of that manager's directives.

10.6 Task Manager

TASK_CREATE	59
TASK_IDENT	163
TASK_START	30
TASK_RESTART	
calling task	64
suspended task – returns to caller	36
blocked task – returns to caller	47
ready task – returns to caller	37
suspended task – preempts caller	77
blocked task – preempts caller	84
ready task – preempts caller	75
TASK_DELETE	
calling task	91
suspended task	47
blocked task	50
ready task	51
TASK_SUSPEND	
calling task	56
returns to caller	16
TASK_RESUME	
task readied – returns to caller	17
task readied – preempts caller	52
TASK_SET_PRIORITY	
obtain current priority	10
returns to caller	25
preempts caller	67
TASK_MODE	
obtain current mode	5
no reschedule	6
reschedule – returns to caller	9
reschedule – preempts caller	42
TASK_GET_NOTE	10
TASK_SET_NOTE	10
TASK_WAKE_AFTER	
yield – returns to caller	6
yield – preempts caller	49
TASK_WAKE_WHEN	75

10.7 Interrupt Manager

It should be noted that the interrupt entry times include vectoring the interrupt handler.

Interrupt Entry Overhead	
returns to nested interrupt	7
returns to interrupted task	8
returns to preempting task	8
Interrupt Exit Overhead	
returns to nested interrupt	5
returns to interrupted task	7
returns to preempting task	14

10.8 Clock Manager

CLOCK_SET	33
CLOCK_GET	4
CLOCK_TICK	6

10.9 Timer Manager

TIMER_CREATE	11
TIMER_IDENT	159
TIMER_DELETE	
inactive	15
active	17
TIMER_FIRE_AFTER	
inactive	21
active	23
TIMER_FIRE_WHEN	
inactive	34
active	34
TIMER_RESET	
inactive	20
active	22
TIMER_CANCEL	
inactive	10
active	13

10.10 Semaphore Manager

SEMAPHORE_CREATE	19
SEMAPHORE_IDENT	171
SEMAPHORE_DELETE	19
SEMAPHORE_OBTAIN	
available	12
not available – NO_WAIT	12
not available – caller blocks	67
SEMAPHORE_RELEASE	
no waiting tasks	14
task readied – returns to caller	23
task readied – preempts caller	57

10.11 Message Manager

MESSAGE_QUEUE_CREATE	114
MESSAGE_QUEUE_IDENT	159
MESSAGE_QUEUE_DELETE	25
MESSAGE_QUEUE_SEND	
no waiting tasks	36
task readied – returns to caller	38
task readied – preempts caller	76
MESSAGE_QUEUE_URGENT	
no waiting tasks	36
task readied – returns to caller	38
task readied – preempts caller	76
MESSAGE_QUEUE_BROADCAST	
no waiting tasks	15
task readied – returns to caller	42
task readied – preempts caller	83
MESSAGE_QUEUE_RECEIVE	
available	30
not available – NO_WAIT	13
not available – caller blocks	67
MESSAGE_QUEUE_FLUSH	
no messages flushed	9
messages flushed	13

10.12 Event Manager

EVENT_SEND	
no task readied	9
task readied – returns to caller	22
task readied – preempts caller	58
EVENT_RECEIVE	
obtain current events	1
available	10
not available – NO_WAIT	9
not available – caller blocks	60

10.13 Signal Manager

SIGNAL_CATCH	6
SIGNAL_SEND	
returns to caller	14
signal to self	22
EXIT ASR OVERHEAD	
returns to calling task	27
returns to preempting task	56

10.14 Partition Manager

PARTITION_CREATE	34
PARTITION_IDENT	159
PARTITION_DELETE	14
PARTITION_GET_BUFFER	
available	12
not available	10
PARTITION_RETURN_BUFFER	10

10.15 Region Manager

REGION_CREATE	22
REGION_IDENT	162
REGION_DELETE	14
REGION_GET_SEGMENT	
available	19
not available – NO_WAIT	19
not available – caller blocks	67
REGION_RETURN_SEGMENT	
no waiting tasks	17
task readied – returns to caller	44
task readied – preempts caller	77

10.16 Dual-Ported Memory Manager

PORT_CREATE	14
PORT_IDENT	159
PORT_DELETE	13
PORT_INTERNAL_TO_EXTERNAL	9
PORT_EXTERNAL_TO_INTERNAL	9

10.17 I/O Manager

IO_INITIALIZE	2
IO_OPEN	1
IO_CLOSE	1
IO_READ	1
IO_WRITE	1
IO_CONTROL	1

10.18 Rate Monotonic Manager

RATE_MONOTONIC_CREATE	12
RATE_MONOTONIC_IDENT	159
RATE_MONOTONIC_CANCEL	14
RATE_MONOTONIC_DELETE	
active	19
inactive	16
RATE_MONOTONIC_PERIOD	
initiate period – returns to caller	20
conclude period – caller blocks	55
obtain status	9

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.