RTEMS PowerPC Applications Supplement

Edition 1, for RTEMS 4.5.0

6 September 2000

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 1999-04-25.15

COPYRIGHT © 1988 - 2000.
On-Line Applications Research Corporation (OAR).

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

Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to either:

On-Line Applications Research Corporation
4910-L Corporate Drive
Huntsville, AL 35805
VOICE: (256) 722-9985
FAX: (256) 722-0985
EMAIL: rtems@OARcorp.com

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 PowerPC architecture dependencies in this port of RTEMS.

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

PowerPC Architecture Documents

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

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

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

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

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

PowerPC Processor Simulator Information

PSIM is a program which emulates the Instruction Set Architecture of the PowerPC microprocessor family. It is reely available in source code form under the terms of the GNU

General Public License (version 2 or later). PSIM can be integrated with the GNU Debugger (gdb) to execute and debug PowerPC executables on non-PowerPC hosts. PSIM supports the addition of user provided device models which can be used to allow one to develop and debug embedded applications using the simulator.

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

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 PowerPC, SPARC, and PA-RISC 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 PowerPC implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file c/src/exec/score/cpu/ppc/ppc.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 PowerPC 603e model, this macro is set to the string "PowerPC 603e".

1.2.2 Floating Point Unit

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

1.2.3 Alignment

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

1.2.4 Cache Alignment

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

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

1.2.5 Maximum Interrupts

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

1.2.6 Has Double Precision Floating Point

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

1.2.7 Critical Interrupts

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

1.2.8 Use Multiword Load/Store Instructions

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

1.2.9 Instruction Cache Size

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

1.2.10 Data Cache Size

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

1.2.11 Debug Model

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

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

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

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

1.2.12 Low Power Model

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

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

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

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.

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

2.2 Programming Model

This section discusses the programming model for the PowerPC architecture.

2.2.1 Non-Floating Point Registers

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

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

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

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

2.2.2 Floating Point Registers

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

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

2.2.3 Special Registers

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

Machine State Register

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

Link Register

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

Count Register

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

2.3 Call and Return Mechanism

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

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

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

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

2.4 Calling Mechanism

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

2.5 Register Usage

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

2.6 Parameter Passing

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

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

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

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

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

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

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

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

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

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

4.2 Synchronous Versus Asynchronous Exceptions

In the PowerPC architecture exceptions can be either precise or imprecise and either synchronous or asynchronous. Asynchronous exceptions occur when an external event interrupts the processor. Synchronous exceptions are caused by the actions of an instruction. During an exception SRR0 is used to calculate where instruction processing should resume. All instructions prior to the resume instruction will have completed execution. SRR1 is used to store the machine status.

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

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

4.3 Vectoring of Interrupt Handler

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

- an instruction address is loaded into SRR0
- bits 33-36 and 42-47 of SRR1 are loaded with information specific to the exception.
- bits 0-32, 37-41, and 48-63 of SRR1 are loaded with corresponding bits from the MSR.

- the MSR is set based upon the exception type.
- instruction fetch and execution resumes, using the new MSR value, at a location specific to the execption type.

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

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

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

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

4.4 Interrupt Levels

The PowerPC architecture supports only a single external asynchronous interrupt source. This interrupt source may be enabled and disabled via the External Interrupt Enable (EE) bit in the Machine State Register (MSR). Thus only two level (enabled and disabled) of external device interrupt priorities are directly supported by the PowerPC architecture.

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

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

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

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

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

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

4.5 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables Critical Interrupts, External Interrupts and

Machine Checks before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than TBD microseconds on a na Mhz PowerPC 603e with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled is hand calculated. This calculation was last performed for Release 4.0.0-lmco.]

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

4.6 Interrupt Stack

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

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

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

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

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

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

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 PowerPC specific BSP issues. For more information on developing a BSP, refer to the chapter titled Board Support Packages in the RTEMS Applications User's Guide.

6.2 System Reset

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

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

6.3 Processor Initialization

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

If this PowerPC implementation supports on-chip caching and this is to be utilized, then it should be enabled during the reset application initialization code. On-chip caching has been observed to prevent some emulators from working properly, so it may be necessary to run with caching disabled to use these emulators.

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

- Must leave the PR bit of the Machine State Register (MSR) set to 0 so the PowerPC remains in the supervisor state.
- Must set stack pointer (sp or r1) such that a minimum stack size of MINIMUM_STACK_SIZE bytes is provided for the `rtems_initialize_executive` directive.
- Must disable all external interrupts (i.e. clear the EI (EE) bit of the machine state register).

- Must enable traps so window overflow and underflow conditions can be properly handled.
- Must initialize the PowerPC's initial Exception Table with default handlers.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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: 428

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: 30,912
- Maximum Configuration: 55,572

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	21,452	NA	
Initialization	1,408	NA	
Task	4,804	NA	
Interrupt	96	NA	
Clock	536	NA	
Timer	1,380	340	
Semaphore	1,928	308	
Message	2,400	532	
Event	1,460	100	
Signal	576	100	
Partition	1,384	244	
Region	1,780	292	
Dual Ported Memory	928	244	
I/O	1,244	NA	
Fatal Error	44	NA	
Rate Monotonic	1,756	336	
Multiprocessing	11,448	612	
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	* 456 =	
maximum_timers	* 68 =	
maximum_semaphores	* 120 =	
maximum_message_queues	* 144 =	
maximum_regions	* 140 =	
maximum_partitions	* 56 =	
maximum_ports	* 36 =	
maximum_periods	* 36 =	
maximum_extensions	* 64 =	
Floating Point Tasks	* 264 =	
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	10,008	
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 uti-

lize 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 PSIM Timing Data

10.1 Introduction

The timing data for RTEMS on the PSIM target 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 PowerPC version of RTEMS.

10.2 Hardware Platform

All times reported in this chapter were measured using the PowerPC Instruction Simulator (PSIM). PSIM simulates a variety of PowerPC 6xx models with the PPC603e being used as the basis for the measurements reported in this chapter.

The PowerPC decrementer register was was used to gather all timing information. In real hardware implementations of the PowerPC architecture, this register would typically count something like CPU cycles or be a function of the clock speed. However, with PSIM each count of the decrementer register represents an instruction. Thus all measurements in this chapter are reported as the actual number of instructions executed. All sources of hardware interrupts were disabled, although traps were enabled and the interrupt level of the PowerPC 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 PowerPC to vector an interrupt and for the RTEMS entry overhead before invoking the user's trap handler are a total of 61 microseconds. These combine to yield a worst case interrupt latency of less than TBD + 61 microseconds at na Mhz. [NOTE: The maximum period with interrupts disabled was last determined for Release 4.0.0-lmco.]

The maximum period with interrupts disabled within RTEMS is hand-timed with some assistance from PSIM. The maximum period with interrupts disabled with RTEMS occurs was not measured on this target.

The interrupt vector and entry overhead time was generated on the PSIM benchmark platform using the PowerPC's decrementer register. This register was programmed to generate an interrupt after one countdown.

10.4 Context Switch

The RTEMS processor context switch time is 214 instructions on the PSIM 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 PSIM benchmark platform:

No Floating Point Contexts	214
Floating Point Contexts	
restore first FP task	255
save initialized, restore initialized	140
save idle, restore initialized	140
save idle, restore idle	290

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	1075
TASK_IDENT	1637
TASK_START	345
TASK_RESTART	
calling task	483
suspended task – returns to caller	396
blocked task – returns to caller	491
ready task – returns to caller	404
suspended task – preempts caller	644
blocked task – preempts caller	709
ready task – preempts caller	686
TASK_DELETE	
calling task	941
suspended task	703
blocked task	723
ready task	729
TASK_SUSPEND	
calling task	403
returns to caller	181
TASK_RESUME	
task readied – returns to caller	191
task readied – preempts caller	803
TASK_SET_PRIORITY	
obtain current priority	147
returns to caller	264
preempts caller	517
TASK_MODE	
obtain current mode	88
no reschedule	110
reschedule – returns to caller	112
reschedule – preempts caller	386
TASK_GET_NOTE	156
TASK_SET_NOTE	155
TASK_WAKE_AFTER	
yield – returns to caller	92
yield – preempts caller	348
TASK_WAKE_WHEN	546

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	60
returns to interrupted task	62
returns to preempting task	61
Interrupt Exit Overhead	
returns to nested interrupt	55
returns to interrupted task	67
returns to preempting task	344

10.8 Clock Manager

CLOCK_SET	340
CLOCK_GET	29
CLOCK_TICK	81

10.9 Timer Manager

TIMER_CREATE	144
TIMER_IDENT	1595
TIMER_DELETE	
inactive	197
active	181
TIMER_FIRE_AFTER	
inactive	252
active	269
TIMER_FIRE_WHEN	
inactive	333
active	334
TIMER_RESET	
inactive	233
active	250
TIMER_CANCEL	
inactive	156
active	140

10.10 Semaphore Manager

SEMAPHORE_CREATE	223
SEMAPHORE_IDENT	1836
SEMAPHORE_DELETE	1836
SEMAPHORE_OBTAIN	
available	175
not available – NO_WAIT	175
not available – caller blocks	530
SEMAPHORE_RELEASE	
no waiting tasks	206
task readied – returns to caller	272
task readied – preempts caller	415

10.11 Message Manager

MESSAGE_QUEUE_CREATE	1022
MESSAGE_QUEUE_IDENT	1596
MESSAGE_QUEUE_DELETE	308
MESSAGE_QUEUE_SEND	
no waiting tasks	421
task readied – returns to caller	434
task readied – preempts caller	581
MESSAGE_QUEUE_URGENT	
no waiting tasks	422
task readied – returns to caller	435
task readied – preempts caller	582
MESSAGE_QUEUE_BROADCAST	
no waiting tasks	244
task readied – returns to caller	482
task readied – preempts caller	630
MESSAGE_QUEUE_RECEIVE	
available	345
not available – NO_WAIT	197
not available – caller blocks	542
MESSAGE_QUEUE_FLUSH	
no messages flushed	142
messages flushed	170

10.12 Event Manager

EVENT_SEND	
no task readied	145
task readied – returns to caller	250
task readied – preempts caller	407
EVENT_RECEIVE	
obtain current events	17
available	133
not available – NO_WAIT	130
not available – caller blocks	442

10.13 Signal Manager

SIGNAL_CATCH	95
SIGNAL_SEND	
returns to caller	165
signal to self	275
EXIT ASR OVERHEAD	
returns to calling task	216
returns to preempting task	329

10.14 Partition Manager

PARTITION_CREATE	320
PARTITION_IDENT	1596
PARTITION_DELETE	168
PARTITION_GET_BUFFER	
available	157
not available	149
PARTITION_RETURN_BUFFER	149

10.15 Region Manager

REGION_CREATE	239
REGION_IDENT	1625
REGION_DELETE	167
REGION_GET_SEGMENT	
available	206
not available – NO_WAIT	190
not available – caller blocks	556
REGION_RETURN_SEGMENT	
no waiting tasks	230
task readied – returns to caller	412
task readied – preempts caller	562

10.16 Dual-Ported Memory Manager

PORT_CREATE	167
PORT_IDENT	1594
PORT_DELETE	165
PORT_INTERNAL_TO_EXTERNAL	133
PORT_EXTERNAL_TO_INTERNAL	134

10.17 I/O Manager

IO_INITIALIZE	23
IO_OPEN	18
IO_CLOSE	18
IO_READ	18
IO_WRITE	18
IO_CONTROL	18

10.18 Rate Monotonic Manager

RATE_MONOTONIC_CREATE	149
RATE_MONOTONIC_IDENT	1595
RATE_MONOTONIC_CANCEL	169
RATE_MONOTONIC_DELETE	
active	212
inactive	186
RATE_MONOTONIC_PERIOD	
initiate period – returns to caller	226
conclude period – caller blocks	362
obtain status	142

11 DMV177 Timing Data

11.1 Introduction

The timing data for RTEMS on the DY-4 DMV177 board 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 PowerPC version of RTEMS.

11.2 Hardware Platform

All times reported in this chapter were measured using a DMV177 board. All data and code caching was disabled. This results in very deterministic times which represent the worst possible performance. Many embedded applications disable caching to insure that execution times are repeatable. Moreover, the JTAG port on certain revisions of the PowerPC 603e does not operate properly if caching is enabled. Thus during development and debug, caching must be off.

The PowerPC decrementer register was was used to gather all timing information. In the PowerPC architecture, this register typically counts something like CPU cycles or is a function of the clock speed. On the PPC603e decrements once for every four (4) bus cycles. On the DMV177, the bus operates at a clock speed of 33 Mhz. This result in a very accurate number since it is a function of the microprocessor itself. Thus all measurements in this chapter are reported as the actual number of decrementer clicks reported.

To convert the numbers reported to microseconds, one should divide the number reported by 8.650752. This number was derived as shown below:

```
((33 * 1048576) / 1000000) / 4 = 8.650752
```

All sources of hardware interrupts were disabled, although traps were enabled and the interrupt level of the PowerPC allows all interrupts.

11.3 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 PowerPC to vector an interrupt and for the RTEMS entry overhead before invoking the user's trap handler are a total of 202 microseconds. These combine to yield a worst case interrupt latency of less than TBD + 202 microseconds at 100.0 Mhz. [NOTE: The maximum period with interrupts disabled was last determined for Release 4.0.0-lmco.]

The maximum period with interrupts disabled within RTEMS is hand-timed with some assistance from the PowerPC simulator. The maximum period with interrupts disabled with RTEMS has not been calculated on this target.

The interrupt vector and entry overhead time was generated on the PSIM benchmark platform using the PowerPC's decrementer register. This register was programmed to generate an interrupt after one countdown.

11.4 Context Switch

The RTEMS processor context switch time is 585 bus cycle on the DMV177 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 DMV177 benchmark platform:

No Floating Point Contexts	585
Floating Point Contexts	
restore first FP task	730
save initialized, restore initialized	478
save idle, restore initialized	825
save idle, restore idle	478

11.5 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.

11.6 Task Manager

TASK_CREATE	2301
TASK_IDENT	2900
TASK_START	794
TASK_RESTART	
calling task	1137
suspended task – returns to caller	906
blocked task – returns to caller	1102
ready task – returns to caller	928
suspended task – preempts caller	1483
blocked task – preempts caller	1640
ready task – preempts caller	1601
TASK_DELETE	
calling task	2117
suspended task	1555
blocked task	1609
ready task	1620
TASK_SUSPEND	
calling task	960
returns to caller	433
TASK_RESUME	
task readied – returns to caller	960
task readied – preempts caller	803
TASK_SET_PRIORITY	
obtain current priority	368
returns to caller	633
preempts caller	1211
TASK_MODE	
obtain current mode	184
no reschedule	213
reschedule – returns to caller	247
reschedule – preempts caller	919
TASK_GET_NOTE	382
TASK_SET_NOTE	383
TASK_WAKE_AFTER	
yield – returns to caller	245
yield – preempts caller	851
TASK_WAKE_WHEN	1275

11.7 Interrupt Manager

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

Interrupt Entry Overhead	
returns to nested interrupt	201
returns to interrupted task	206
returns to preempting task	202
Interrupt Exit Overhead	
returns to nested interrupt	201
returns to interrupted task	213
returns to preempting task	857

11.8 Clock Manager

CLOCK_SET	792
CLOCK_GET	78
CLOCK_TICK	214

11.9 Timer Manager

TIMER_CREATE	357
TIMER_IDENT	2828
TIMER_DELETE	
inactive	432
active	471
TIMER_FIRE_AFTER	
inactive	607
active	646
TIMER_FIRE_WHEN	
inactive	766
active	764
TIMER_RESET	
inactive	552
active	766
TIMER_CANCEL	
inactive	339
active	378

11.10 Semaphore Manager

SEMAPHORE_CREATE	571
SEMAPHORE_IDENT	3243
SEMAPHORE_DELETE	575
SEMAPHORE_OBTAIN	
available	414
not available – NO_WAIT	414
not available – caller blocks	1254
SEMAPHORE_RELEASE	
no waiting tasks	501
task readied – returns to caller	636
task readied – preempts caller	982

11.11 Message Manager

MESSAGE_QUEUE_CREATE	2270
MESSAGE_QUEUE_IDENT	2828
MESSAGE_QUEUE_DELETE	708
MESSAGE_QUEUE_SEND	
no waiting tasks	923
task readied – returns to caller	955
task readied – preempts caller	1322
MESSAGE_QUEUE_URGENT	
no waiting tasks	919
task readied – returns to caller	955
task readied – preempts caller	1322
MESSAGE_QUEUE_BROADCAST	
no waiting tasks	589
task readied – returns to caller	1079
task readied – preempts caller	1435
MESSAGE_QUEUE_RECEIVE	
available	755
not available – NO_WAIT	467
not available – caller blocks	1283
MESSAGE_QUEUE_FLUSH	
no messages flushed	369
messages flushed	431

11.12 Event Manager

EVENT_SEND	
no task readied	354
task readied – returns to caller	571
task readied – preempts caller	946
EVENT_RECEIVE	
obtain current events	43
available	357
not available – NO_WAIT	331
not available – caller blocks	1043

11.13 Signal Manager

SIGNAL_CATCH	267
SIGNAL_SEND	
returns to caller	408
signal to self	607
EXIT ASR OVERHEAD	
returns to calling task	464
returns to preempting task	752

11.14 Partition Manager

PARTITION_CREATE	762
PARTITION_IDENT	2828
PARTITION_DELETE	426
PARTITION_GET_BUFFER	
available	394
not available	376
PARTITION_RETURN_BUFFER	376

11.15 Region Manager

REGION_CREATE	614
REGION_IDENT	2878
REGION_DELETE	425
REGION_GET_SEGMENT	
available	515
not available – NO_WAIT	472
not available – caller blocks	1345
REGION_RETURN_SEGMENT	
no waiting tasks	544
task readied – returns to caller	935
task readied – preempts caller	1296

11.16 Dual-Ported Memory Manager

PORT_CREATE	428
PORT_IDENT	2828
PORT_DELETE	421
PORT_INTERNAL_TO_EXTERNAL	339
PORT_EXTERNAL_TO_INTERNAL	339

11.17 I/O Manager

IO_INITIALIZE	52
IO_OPEN	42
IO_CLOSE	44
IO_READ	42
IO_WRITE	44
IO_CONTROL	42

11.18 Rate Monotonic Manager

RATE_MONOTONIC_CREATE	388
RATE_MONOTONIC_IDENT	2826
RATE_MONOTONIC_CANCEL	427
RATE_MONOTONIC_DELETE	
active	519
inactive	465
RATE_MONOTONIC_PERIOD	
initiate period – returns to caller	556
conclude period – caller blocks	842
obtain status	377

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.

Table of Contents

Preface 1

1 CPU Model Dependent Features 3

1.1 Introduction 3
1.2 CPU Model Feature Flags 3
1.2.1 CPU Model Name 3
1.2.2 Floating Point Unit 3
1.2.3 Alignment 4
1.2.4 Cache Alignment 4
1.2.5 Maximum Interrupts 4
1.2.6 Has Double Precision Floating Point 4
1.2.7 Critical Interrupts 4
1.2.8 Use Multiword Load/Store Instructions 4
1.2.9 Instruction Cache Size 4
1.2.10 Data Cache Size 4
1.2.11 Debug Model 5
1.2.12 Low Power Model 5

2 Calling Conventions 7

2.1 Introduction 7
2.2 Programming Model 7
2.2.1 Non-Floating Point Registers 7
2.2.2 Floating Point Registers 8
2.2.3 Special Registers 8
2.3 Call and Return Mechanism 8
2.4 Calling Mechanism 9
2.5 Register Usage 9
2.6 Parameter Passing 9
2.7 User-Provided Routines 9

3 Memory Model 11

3.1 Introduction 11
3.2 Flat Memory Model 11

4 Interrupt Processing 13

4.1 Introduction 13
4.2 Synchronous Versus Asynchronous Exceptions 13
4.3 Vectoring of Interrupt Handler 13
4.4 Interrupt Levels 14
4.5 Disabling of Interrupts by RTEMS 14
4.6 Interrupt Stack 15

5 Default Fatal Error Processing 17
5.1 Introduction 17
5.2 Default Fatal Error Handler Operations 17

6 Board Support Packages 19
6.1 Introduction 19
6.2 System Reset 19
6.3 Processor Initialization 19

7 Processor Dependent Information Table 21
7.1 Introduction 21
7.2 CPU Dependent Information Table 21

8 Memory Requirements 25
8.1 Introduction 25
8.2 Data Space Requirements 25
8.3 Minimum and Maximum Code Space Requirements 25
8.4 RTEMS Code Space Worksheet 25
8.5 RTEMS RAM Workspace Worksheet 27

9 Timing Specification 29
9.1 Introduction 29
9.2 Philosophy 29
9.2.1 Determinancy 29
9.2.2 Interrupt Latency 30
9.2.3 Context Switch Time 31
9.2.4 Directive Times 31
9.3 Methodology 32
9.3.1 Software Platform 32
9.3.2 Hardware Platform 32
9.3.3 What is measured? 32
9.3.4 What is not measured? 33
9.3.5 Terminology 33

10 PSIM Timing Data 35

10.1 Introduction 35
10.2 Hardware Platform 35
10.3 Interrupt Latency 35
10.4 Context Switch 35
10.5 Directive Times 36
10.6 Task Manager 37
10.7 Interrupt Manager 38
10.8 Clock Manager 38
10.9 Timer Manager 38
10.10 Semaphore Manager 39
10.11 Message Manager 39
10.12 Event Manager 40
10.13 Signal Manager 40
10.14 Partition Manager 40
10.15 Region Manager 41
10.16 Dual-Ported Memory Manager 41
10.17 I/O Manager 41
10.18 Rate Monotonic Manager 41

11 DMV177 Timing Data 43

11.1 Introduction 43
11.2 Hardware Platform 43
11.3 Interrupt Latency 43
11.4 Context Switch 44
11.5 Directive Times 44
11.6 Task Manager 45
11.7 Interrupt Manager 46
11.8 Clock Manager 46
11.9 Timer Manager 46
11.10 Semaphore Manager 47
11.11 Message Manager 47
11.12 Event Manager 48
11.13 Signal Manager 48
11.14 Partition Manager 48
11.15 Region Manager 49
11.16 Dual-Ported Memory Manager 49
11.17 I/O Manager 49
11.18 Rate Monotonic Manager 49

Command and Variable Index 51

Concept Index 53