RTEMS Hewlett Packard PA-RISC Applications Su

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.

For information on the PA-RISC V1.1 architecture in general, refer to the following documents:

- *PA-RISC 1.1 Architecture and Instruction Set Reference Manual, Third Edition. HP Part Number 09740-90039.*

It is highly recommended that the PA-RISC RTEMS application developer also obtain and become familiar with the Technical Reference Manual for the particular implementation of the PA-RISC being used.

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

This chapter presents the set of features which vary across PA-RISC implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file c/src/exec/score/cpu/hppa1_1/hppa.h based upon the particular CPU model defined on the compilation command line.

1.2 CPU Model Name

The macro CPU_MODEL_NAME is a string which designates the name of this CPU model. For example, for the Hewlett Packard PA-7100 CPU model, this macro is set to the string "hppa 7100".

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.

This chapter describes the calling conventions used by the GNU C and standard HP-UX compilers for the PA-RISC architecture.

2.2 Processor Background

The PA-RISC architecture supports a simple yet effective call and return mechanism for subroutine calls where the caller and callee are both in the same address space. The compiler will not automatically generate subroutine calls which cross address spaces. A subroutine is invoked via the branch and link (bl) or the branch and link register (blr). These instructions save the return address in a caller specified register. By convention, the return address is saved in r2. The callee is responsible for maintaining the return address so it can return to the correct address. The branch vectored (bv) instruction is used to branch to the return address and thus return from the subroutine to the caller. It is is important to note that the PA-RISC subroutine call and return mechanism does not automatically save or restore any registers. It is the responsibility of the high-level language compiler to define the register preservation and usage convention.

2.3 Calling Mechanism

All RTEMS directives are invoked as standard subroutines via a bl or a blr instruction with the return address assumed to be in r2 and return to the user application via the bv instruction.

2.4 Register Usage

As discussed above, the bl and blr instructions do not automatically save any registers. RTEMS uses the registers r1, r19 - r26, and r31 as scratch registers. The PA-RISC calling convention specifies that the first four (4) arguments to subroutines are passed in registers r23 - r26. After the arguments have been used, the contents of these registers may be

altered. Register r31 is the millicode scratch register. Millicode is the set of routines which support high-level languages on the PA-RISC by providing routines which are either too complex or too long for the compiler to generate inline code when these operations are needed. For example, indirect calls utilize a millicode routine. The scratch registers are not preserved by RTEMS directives therefore, the contents of these registers should not be assumed upon return from any RTEMS directive.

Surprisingly, when using the GNU C compiler at least integer multiplies are performed using the floating point registers. This is an important optimization because the PA-RISC does not have otherwise have hardware for multiplies. This has important ramifications in regards to the PA-RISC port of RTEMS. On most processors, the floating point unit is ignored if the code only performs integer operations. This makes it easy for the application developer to predict whether or not any particular task will require floating point operations. This property is taken advantage of by RTEMS on other architectures to minimize the number of times the floating point context is saved and restored. However, on the PA-RISC architecture, every task is implicitly a floating point task. Additionally the state of the floating point unit must be saved and restored as part of the interrupt processing because for all practical purposes it is impossible to avoid the use of the floating point registers. It is unknown if the HP-UX C compiler shares this property.

> **NOTE**: Later versions of the GNU C has a PA-RISC specific option to disable use of the floating point registers. RTEMS currently assumes that this option is not turned on. If the use of this option sets a built-in define, then it should be possible to modify the PA-RISC specific code such that all tasks are considered floating point only when this option is not used.

2.5 Parameter Passing

RTEMS assumes that the first four (4) arguments are placed in the appropriate registers (r26, r25, r24, and r23) and, if needed, additional are placed on the current stack before the directive is invoked via the bl or blr instruction. The first argument is placed in r26, the second is placed in r25, and so forth. The following pseudo-code illustrates the typical sequence used to call a RTEMS directive with three (3) arguments:

```
set r24 to the third argument
set r25 to the second argument
set r26 to the first argument
invoke directive
```

The stack on the PA-RISC grows upward – i.e. "pushing" onto the stack results in the address in the stack pointer becoming numerically larger. By convention, r27 is used as the stack pointer. The standard stack frame consists of a minimum of sixty-four (64) bytes and is the responsibility of the callee to maintain.

2.6 User-Provided Routines

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

RTEMS supports applications in which the application and the executive execute within a single thirty-two bit address space. Thus RTEMS and the application share a common four gigabyte address space within a single space. The PA-RISC automatically converts every address from a logical to a physical address each time it is used. The PA-RISC uses information provided in the page table to perform this translation. The following protection levels are assumed:

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

The PA-RISC space registers and associated stack – including the stack pointer r27 – must be initialized when the initialize_executive directive is invoked. RTEMS treats the space registers as system resources shared by all tasks and does not modify or context switch them.

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

RTEMS does not require that logical addresses map directly to physical addresses, although it is desirable in many applications to do so. RTEMS does not need any additional information when physical addresses do not map directly to physical addresses. By not requiring that logical addresses map directly to physical addresses, the memory space of an RTEMS space can be separated from that of a ROM monitor. For example, a ROM monitor may load application programs into a separate logical address space from itself.

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

4 Interrupt Processing

4.1 Introduction

Different types of processors respond to the occurence of an interrupt in their 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 PA-RISC's interrupt response and control mechanisms as they pertain to RTEMS.

4.2 Vectoring of Interrupt Handler

Upon receipt of an interrupt the PA-RISC automatically performs the following actions:

- The PSW (Program Status Word) is saved in the IPSW (Interrupt Program Status Word).
- The current privilege level is set to 0.
- The following defined bits in the PSW are set:
- E bit is set to the default endian bit
- M bit is set to 1 if the interrupt is a high-priority machine check and 0 otherwise
- Q bit is set to zero thuse freezing the IIA (Instruction Address) queues
- C and D bits are set to zero thus disabling all protection and translation.
- I bit is set to zero this disabling all external, powerfail, and low-priority machine check interrupts.
- All others bits are set to zero.
- General purpose registers r1, r8, r9, r16, r17, r24, and r25 are copied to the shadow registers.
- Execution begins at the address given by the formula: Interruption Vector Address + (32 * interrupt vector number).

Once the processor has completed the actions it is is required to perform for each interrupt, the RTEMS interrupt management code (the beginning of which is stored in the Interruption Vector Table) gains control and performs the following actions upon each interrupt:

- returns the processor to "virtual mode" thus reenabling all code and data address translation.
- saves all necessary interrupt state information
- saves all floating point registers
- saves all integer registers
- switches the current stack to the interrupt stack

- dispatches to the appropriate user provided interrupt service routine (ISR). If the ISR was installed with the interrupt_catch directive, then it will be executed at this. Because, the RTEMS interrupt handler saves all registers which are not preserved according to the calling conventions and invokes the application's ISR, the ISR can easily be written in a high-level language.

RTEMS refers to the combination of the interrupt state information and registers saved when vectoring an interrupt as the Interrupt Stack Frame (ISF). A nested interrupt is processed similarly by the PA-RISC and RTEMS with the exception that the nested interrupt occurred while executing on the interrupt stack and, thus, the current stack need not be switched.

4.3 Interrupt Stack Frame

The PA-RISC architecture does not alter the stack while processing interrupts. However, RTEMS does save information on the stack as part of processing an interrupt. This following shows the format of the Interrupt Stack Frame for the PA-RISC as defined by RTEMS:

```
+------------------------+
|    Interrupt Context   | 0xXXX
+------------------------+
|     Integer Context    | 0xXXX
+------------------------+
| Floating Point Context | 0xXXX
+------------------------+
```

4.4 External Interrupts and Traps

In addition to the thirty-two unique interrupt sources supported by the PA-RISC architecture, RTEMS also supports the installation of handlers for each of the thirty-two external interrupts supported by the PA-RISC architecture. Except for interrupt vector 4, each of the interrupt vectors 0 through 31 may be associated with a user-provided interrupt handler. Interrupt vector 4 is used for external interrupts. When an external interrupt occurs, the RTEMS external interrupt handler is invoked and the actual interrupt source is indicated by status bits in the EIR (External Interrupt Request) register. The RTEMS external interrupt handler (or interrupt vector four) examines the EIR to determine which interrupt source requires servicing.

RTEMS supports sixty-four interrupt vectors for the PA-RISC. Vectors 0 through 31 map to the normal interrupt sources while RTEMS interrupt vectors 32 through 63 are directly associated with the external interrupt sources indicated by bits 0 through 31 in the EIR.

The exact set of interrupt sources which are checked for by the RTEMS external interrupt handler and the order in which they are checked are configured by the user in the CPU Configuration Table. If an external interrupt occurs which does not have a handler configured, then the spurious interrupt handler will be invoked. The spurious interrupt handler may also be specifiec by the user in the CPU Configuration Table.

4.5 Interrupt Levels

Two levels (enabled and disabled) of interrupt priorities are supported by the PA-RISC architecture. Level zero (0) indicates that interrupts are fully enabled (i.e. the I bit of the PSW is 1). Level one (1) indicates that interrupts are disabled (i.e. the I bit of the PSW is 0). Thirty-two independent sources of external interrupts are supported by the PA-RISC architecture. Each of these interrupts sources may be individually enabled or disabled. When processor interrupts are disabled, all sources of external interrupts are ignored. When processor interrupts are enabled, the EIR (External Interrupt Request) register is used to determine which sources are currently allowed to generate interrupts.

Although RTEMS supports 256 interrupt levels, the PA-RISC architecture only supports two. RTEMS interrupt level 0 indicates that interrupts are enabled and level 1 indicates that interrupts are disabled. 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 external interrupts by setting the I bit in the PSW to 0 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 XXX instructions when compiled with GNU CC at optimization level 4. The exact execution time will vary based on the based on the processor implementation, amount of cache, the number of wait states for primary memory, and processor speed present on the target board.

Non-maskable interrupts (NMI) such as high-priority machine checks 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.

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 a user-supplied fatal error handler. If no user-supplied handler is configured, the RTEMS provided default fatal error handler is invoked. If the user-supplied fatal error handler returns 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 (i.e. sets the I bit in the PSW register to 0), places the error code in r1, and executes a break instruction 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 PA-RISC 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 PA-RISC processor is reset. The behavior of a PA-RISC upon reset is implementation defined and thus is beyond the scope of this manual.

6.3 Processor Initialization

The precise requirements for initialization of a particular implementation of the PA-RISC architecture are implementation defined. Thus it is impossible to provide exact details of this procedure in this manual. However, the requirements of RTEMS which must be satisfied by this initialization code can be discussed.

RTEMS assumes that interrupts are disabled when the initialize_executive directive is invoked. Interrupts are enabled automatically by RTEMS as part of the initialize executive directive and device driver initialization occurs after interrupts are enabled. Thus all interrupt sources should be quiescent until the system's device drivers have been initialized and installed their interrupt handlers.

If the processor requires initialization of the cache, then it should be be done during the reset application initialization code.

Finally, the requirements in the Board Support Packages chapter of the Applications User's Manual for the reset code which is executed before the call to initialize executive must be satisfied.

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

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

  hppa_rtems_isr_entry spurious_handler;

      /* itimer_clicks_per_microsecond is for the Clock driver */
  unsigned32   itimer_clicks_per_microsecond;
}   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.

`spurious_handler` is the address of the optional user provided routine which is invoked when a spurious external interrupt occurs. A spurious interrupt is one for which no handler is installed.

`itimer_clicks_per_microsecond`
is the number of countdowns in the on-CPU timer which corresponds to a microsecond. This is a function of the clock speed of the CPU being used.

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: 128

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: xx,129
- Maximum Configuration: xx,130

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	x,131	NA	
Initialization	x,132	NA	
Task	x,133	NA	
Interrupt	x,134	NA	
Clock	x,135	NA	
Timer	x,136	148	
Semaphore	x,137	149	
Message	x,138	150	
Event	x,139	151	
Signal	x,140	152	
Partition	x,141	153	
Region	x,142	154	
Dual Ported Memory	x,143	155	
I/O	x,144	156	
Fatal Error	x,145	NA	
Rate Monotonic	x,146	157	
Multiprocessing	x,147	158	
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	* 159 =	
maximum_timers	* 160 =	
maximum_semaphores	* 161 =	
maximum_message_queues	* 162 =	
maximum_regions	* 163 =	
maximum_partitions	* 164 =	
maximum_ports	* 165 =	
maximum_periods	* 166 =	
maximum_extensions	* 167 =	
Floating Point Tasks	* 168 =	
Task Stacks	=	
Total Single Processor Requirements		
Description	**Equation**	**Bytes Required**
maximum_nodes	* 169 =	
maximum_global_objects	* 170 =	
maximum_proxies	* 171 =	
Total Multiprocessing Requirements		
Fixed System Requirements	x,172	
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 HP-7100 Timing Data

10.1 Introduction

The timing data for the PA-RISC version of RTEMS 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 PA-RISC version of RTEMS.

10.2 Hardware Platform

No directive execution times are reported for the HP-7100 because the target platform was proprietary and executions times could not be released.

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 RTEMS_MAXIMUM_DISABLE_PERIOD microseconds including the instructions which disable and re-enable interrupts. The time required for the HP-7100 to vector an interrupt and for the RTEMS entry overhead before invoking the user's trap handler are a total of RTEMS_INTR_ENTRY_RETURNS_TO_PREEMPTING_TASK microseconds. These combine to yield a worst case interrupt latency of less than RTEMS_MAXIMUM_DISABLE_PERIOD + RTEMS_INTR_ENTRY_RETURNS_TO_PREEMPTING_TASK microseconds at 15 Mhz. [NOTE: The maximum period with interrupts disabled was last determined for Release RTEMS_RELEASE_FOR_MAXIMUM_DISABLE_PERIOD.]

It should be noted again that the maximum period with interrupts disabled within RTEMS for the HP-7100 is hand calculated.

10.4 Context Switch

The RTEMS processor context switch time is RTEMS_NO_FP_CONTEXTS microsections for the HP-7100 when no floating point context switch is saved or restored. Saving and restoring the floating point context adds additional time to the context switch procedure. 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. On many processors, 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. As discussed in the Register Usage section, on the HP-7100 the every task is considered to be floating point registers and , as a rsule, every context switch involves saving and restoring the state of the floating point unit.

The following table summarizes the context switch times for the HP-7100 processor:

```
no times are available for the HP-7100
```

10.5 Directive Times

No execution times are available for the HP-7100 because the target platform was proprietary and no timing information could be released.

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 Name 3

2 Calling Conventions 5

2.1 Introduction 5
2.2 Processor Background 5
2.3 Calling Mechanism 5
2.4 Register Usage 5
2.5 Parameter Passing 6
2.6 User-Provided Routines 6

3 Memory Model 7

3.1 Introduction 7
3.2 Flat Memory Model 7

4 Interrupt Processing 9

4.1 Introduction 9
4.2 Vectoring of Interrupt Handler 9
4.3 Interrupt Stack Frame 10
4.4 External Interrupts and Traps 10
4.5 Interrupt Levels 11
4.6 Disabling of Interrupts by RTEMS 11

5 Default Fatal Error Processing 13

5.1 Introduction 13
5.2 Default Fatal Error Handler Operations 13

6 Board Support Packages 15

6.1 Introduction 15
6.2 System Reset 15
6.3 Processor Initialization 15

7 Processor Dependent Information Table 17

7.1 Introduction 17
7.2 CPU Dependent Information Table 17

8 Memory Requirements 19
8.1 Introduction 19
8.2 Data Space Requirements 19
8.3 Minimum and Maximum Code Space Requirements 19
8.4 RTEMS Code Space Worksheet 19
8.5 RTEMS RAM Workspace Worksheet 21

9 Timing Specification 23
9.1 Introduction 23
9.2 Philosophy 23
9.2.1 Determinancy 23
9.2.2 Interrupt Latency 24
9.2.3 Context Switch Time 25
9.2.4 Directive Times 25
9.3 Methodology 26
9.3.1 Software Platform 26
9.3.2 Hardware Platform 26
9.3.3 What is measured? 26
9.3.4 What is not measured? 27
9.3.5 Terminology 27

10 HP-7100 Timing Data 29
10.1 Introduction 29
10.2 Hardware Platform 29
10.3 Interrupt Latency 29
10.4 Context Switch 29
10.5 Directive Times 30

Command and Variable Index 31

Concept Index 33