RTEMS Intel i386 Applications Supplement

Edition 4.6.99.3, for RTEMS 4.6.99.3

19 September 2003

On-Line Applications Research Corporation

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

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

Table of Contents

Preface

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

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

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

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

1 CPU Model Dependent Features

1.1 Introduction

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

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

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

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 Intel i386 without an i387 coprocessor, this macro is set to the string `"i386 with i387"`.

1.3 bswap Instruction

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

1.4 Floating Point Unit

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

2 Calling Conventions

2.1 Introduction

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

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

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

2.2 Processor Background

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

2.3 Calling Mechanism

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

2.4 Register Usage

As discussed above, the call instruction does not automatically save any registers. RTEMS uses the registers EAX, ECX, and EDX as scratch registers. These registers are not preserved by RTEMS directives therefore, the contents of these registers should not be assumed upon return from any RTEMS directive.

2.5 Parameter Passing

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

```
push third argument
push second argument
```

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

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

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

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

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

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

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

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

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

4 Interrupt Processing

4.1 Introduction

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

4.2 Vectoring of Interrupt Handler

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

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

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

A nested interrupt is processed similarly by the i386.

4.3 Interrupt Stack Frame

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

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

4.4 Interrupt Levels

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

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

4.5 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables interrupts 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 13.0 microseconds on a 16 Mhz i386 with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled within RTEMS was last calculated for Release 3.1.0.]

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

4.6 Interrupt Stack

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

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

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

5 Default Fatal Error Processing

5.1 Introduction

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

5.2 Default Fatal Error Handler Operations

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

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 i386 specific BSP issues. For more information on developing a BSP, refer to the chapter titled Board Support Packages in the RTEMS Applications User's Guide.

6.2 System Reset

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

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

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

6.3 Processor Initialization

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

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

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

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

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

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

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

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

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

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

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

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

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

`idle_task`
: is the address of the optional user provided routine which is used as the system's IDLE task. If this field is not NULL, then the RTEMS

default IDLE task is not used. This field may be NULL to indicate that the default IDLE is to be used.

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

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

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

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

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

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

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

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

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

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: 833

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: 22,660
- Maximum Configuration: 39,592

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	16,948	NA	
Initialization	916	NA	
Task	3,436	NA	
Interrupt	52	NA	
Clock	296	NA	
Timer	1,084	144	
Semaphore	1,500	136	
Message	1,596	224	
Event	1,036	44	
Signal	396	44	
Partition	1,052	104	
Region	1,392	124	
Dual Ported Memory	664	104	
I/O	676	00	
Fatal Error	20	NA	
Rate Monotonic	1,132	136	
Multiprocessing	6,840	228	
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	* 372 =	
maximum_timers	* 68 =	
maximum_semaphores	* 124 =	
maximum_message_queues	* 148 =	
maximum_regions	* 144 =	
maximum_partitions	* 56 =	
maximum_ports	* 36 =	
maximum_periods	* 36 =	
maximum_extensions	* 64 =	
Floating Point Tasks	* 108 =	
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	6,768	
Total Single Processor Requirements		
Total Multiprocessing Requirements		
Minimum Bytes for RTEMS Workspace		

9 Timing Specification

9.1 Introduction

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

9.2 Philosophy

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

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

9.2.1 Determinancy

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

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

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

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

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

9.2.2 Interrupt Latency

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

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

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

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

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

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

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

9.2.3 Context Switch Time

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

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

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

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

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

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

9.2.4 Directive Times

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

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

9.3 Methodology

9.3.1 Software Platform

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

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

9.3.2 Hardware Platform

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

9.3.3 What is measured?

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

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

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

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

In addition, no user extensions are configured.

9.3.4 What is not measured?

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

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

9.3.5 Terminology

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

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

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

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

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

calling task The task invoking the directive.

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

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

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

no reschedule The directive did not require a rescheduling operation.

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

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

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

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

reschedule The actions of the directive necessitated a rescheduling operation.

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

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

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

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

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

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

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

yield The act of attempting to voluntarily release the CPU.

10 CPU386 Timing Data

10.1 Introduction

The timing data for the i386 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 i386 version of RTEMS.

10.2 Hardware Platform

All times reported except for the maximum period interrupts are disabled by RTEMS were measured using a Force Computers CPU386 board. The CPU386 is a 16 Mhz board with zero wait state dynamic memory and an i80387 numeric coprocessor. One of the countdown timers provided by a Motorola MC68901 was used to measure elapsed time with one microsecond resolution. All sources of hardware interrupts are disabled, although the interrupt level of the i386 allows all interrupts.

The maximum period interrupts are disabled was measured by summing the number of CPU cycles required by each assembly language instruction executed while interrupts were disabled. Zero wait state memory was assumed. The total CPU cycles executed with interrupts disabled, including the instructions to disable and enable interrupts, was divided by 16 to simulate a i386 executing at 16 Mhz.

10.3 Interrupt Latency

The maximum period with interrupts disabled within RTEMS is less than 13.0 microseconds including the instructions which disable and re-enable interrupts. The time required for the i386 to generate an interrupt using the int instruction, vectoring to an interrupt handler, and for the RTEMS entry overhead before invoking the user's interrupt handler are a total of 12 microseconds. These combine to yield a worst case interrupt latency of less 13.0 + 12 microseconds. [NOTE: The maximum period with interrupts disabled within RTEMS was last calculated for Release 3.1.0.]

It should be noted again that the maximum period with interrupts disabled within RTEMS is hand-timed. The interrupt vector and entry overhead time was generated on the Force Computers CPU386 benchmark platform using the int instruction as the interrupt source.

10.4 Context Switch

The RTEMS processor context switch time is 34 microseconds on the Force Computers CPU386 benchmark platform. This time represents the raw context switch time with no user extensions configured. 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 base 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 a 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 exact amount of time required to save and restore floating point context is dependent on the state of the numeric coprocessor. RTEMS places the coprocessor in the initialized state when a task is started or restarted. Once the task has utilized the coprocessor, it is in the idle state when floating point instructions are not executing and the busy state when floating point instructions are executing. The state of the coprocessor is task specific.

The following table summarizes the context switch times for the Force Computers CPU386 benchmark platform:

No Floating Point Contexts	34
Floating Point Contexts	
restore first FP task	57
save initialized, restore initialized	59
save idle, restore initialized	59
save idle, restore idle	83

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	157
TASK_IDENT	748
TASK_START	86
TASK_RESTART	
calling task	118
suspended task – returns to caller	45
blocked task – returns to caller	138
ready task – returns to caller	105
suspended task – preempts caller	149
blocked task – preempts caller	162
ready task – preempts caller	156
TASK_DELETE	
calling task	187
suspended task	147
blocked task	153
ready task	157
TASK_SUSPEND	
calling task	81
returns to caller	45
TASK_RESUME	
task readied – returns to caller	46
task readied – preempts caller	71
TASK_SET_PRIORITY	
obtain current priority	30
returns to caller	67
preempts caller	115
TASK_MODE	
obtain current mode	19
no reschedule	21
reschedule – returns to caller	27
reschedule – preempts caller	66
TASK_GET_NOTE	32
TASK_SET_NOTE	32
TASK_WAKE_AFTER	
yield – returns to caller	18
yield – preempts caller	63
TASK_WAKE_WHEN	128

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	12
returns to interrupted task	13
returns to preempting task	12
Interrupt Exit Overhead	
returns to nested interrupt	10
returns to interrupted task	13
returns to preempting task	58

10.8 Clock Manager

CLOCK_SET	85
CLOCK_GET	2
CLOCK_TICK	16

10.9 Timer Manager

TIMER_CREATE	34
TIMER_IDENT	729
TIMER_DELETE	
inactive	48
active	52
TIMER_FIRE_AFTER	
inactive	65
active	69
TIMER_FIRE_WHEN	
inactive	92
active	92
TIMER_RESET	
inactive	58
active	63
TIMER_CANCEL	
inactive	32
active	37

10.10 Semaphore Manager

SEMAPHORE_CREATE	64
SEMAPHORE_IDENT	787
SEMAPHORE_DELETE	60
SEMAPHORE_OBTAIN	
available	41
not available – NO_WAIT	40
not available – caller blocks	123
SEMAPHORE_RELEASE	
no waiting tasks	47
task readied – returns to caller	70
task readied – preempts caller	95

10.11 Message Manager

MESSAGE_QUEUE_CREATE	294
MESSAGE_QUEUE_IDENT	730
MESSAGE_QUEUE_DELETE	81
MESSAGE_QUEUE_SEND	
no waiting tasks	117
task readied – returns to caller	118
task readied – preempts caller	144
MESSAGE_QUEUE_URGENT	
no waiting tasks	117
task readied – returns to caller	116
task readied – preempts caller	144
MESSAGE_QUEUE_BROADCAST	
no waiting tasks	53
task readied – returns to caller	122
task readied – preempts caller	146
MESSAGE_QUEUE_RECEIVE	
available	93
not available – NO_WAIT	45
not available – caller blocks	127
MESSAGE_QUEUE_FLUSH	
no messages flushed	29
messages flushed	41

10.12 Event Manager

EVENT_SEND	
no task readied	26
task readied – returns to caller	60
task readied – preempts caller	89
EVENT_RECEIVE	
obtain current events	¡1
available	27
not available – NO_WAIT	25
not available – caller blocks	94

10.13 Signal Manager

SIGNAL_CATCH	13
SIGNAL_SEND	
returns to caller	34
signal to self	59
EXIT ASR OVERHEAD	
returns to calling task	39
returns to preempting task	60

10.14 Partition Manager

PARTITION_CREATE	83
PARTITION_IDENT	730
PARTITION_DELETE	40
PARTITION_GET_BUFFER	
available	34
not available	33
PARTITION_RETURN_BUFFER	33

10.15 Region Manager

REGION_CREATE	68
REGION_IDENT	739
REGION_DELETE	39
REGION_GET_SEGMENT	
available	49
not available – NO_WAIT	45
not available – caller blocks	127
REGION_RETURN_SEGMENT	
no waiting tasks	52
task readied – returns to caller	113
task readied – preempts caller	138

10.16 Dual-Ported Memory Manager

PORT_CREATE	39
PORT_IDENT	728
PORT_DELETE	39
PORT_INTERNAL_TO_EXTERNAL	26
PORT_EXTERNAL_TO_INTERNAL	26

10.17 I/O Manager

IO_INITIALIZE	4
IO_OPEN	1
IO_CLOSE	1
IO_READ	¡1
IO_WRITE	1
IO_CONTROL	1

10.18 Rate Monotonic Manager

RATE_MONOTONIC_CREATE	36
RATE_MONOTONIC_IDENT	725
RATE_MONOTONIC_CANCEL	39
RATE_MONOTONIC_DELETE	
active	53
inactive	49
RATE_MONOTONIC_PERIOD	
initiate period – returns to caller	53
conclude period – caller blocks	82
obtain status	30

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.