RTEMS ARM Applications Supplement

Edition 4.6.2, for RTEMS 4.6.2

30 August 2003

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 2002-11-25.11

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

Table of Contents

Preface

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

This document discusses the ARM architecture dependencies in this port of RTEMS. The ARM family has a wide variety of implementations by a wide range of vendors. Consequently, there are 100's of CPU models within it.

It is highly recommended that the ARM RTEMS application developer obtain and become familiar with the documentation for the processor being used as well as the documentation for the ARM architecture as a whole.

Architecture Documents

For information on the ARM architecture, refer to the following documents available from Arm, Limited ('`http//www.arm.com/`'). There does not appear to be an electronic version of a manual on the architecture in general on that site. The following book is a good resource:

- *David Seal.* "*ARM Architecture Reference Manual.*" *Addison-Wesley.* **ISBN 0-201-73719-1**. *2001.*

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 ARM, SPARC, and PA-RISC are based on an architectural specification which is independent or any particular CPU model or implementation. Older families such as the M68xxx and the iX86 evolved as the manufacturer strived to produce higher performance processor models which maintained binary compatibility with older models.

RTEMS takes advantage of the similarity of the various models within a CPU family. Although the models do vary in significant ways, the high level of compatibility makes it possible to share the bulk of the CPU dependent executive code across the entire family. 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 ARM implementations and are of importance to RTEMS. The set of CPU model feature macros are defined in the file cpukit/score/cpu/arm/rtems/score/arm.h based upon the particular CPU model defined on the compilation command line.

1.2 CPU Model Name

The macro `CPU_MODEL_NAME` is a string which designates the architectural level of this CPU model. The following is a list of the settings for this string based upon `gcc` CPU model predefines:

```
__ARM_ARCH4__   "ARMv4"
__ARM_ARCH4T__  "ARMv4T"
__ARM_ARCH5__   "ARMv5"
__ARM_ARCH5T__  "ARMv5T"
__ARM_ARCH5E__  "ARMv5E"
__ARM_ARCH5TE__ "ARMv5TE"
```

1.3 Count Leading Zeroes Instruction

The macro `ARM_HAS_CLZ` is set to 1 to indicate that the architectural version has the `clz` instruction. On ARM architectural version 5 and above, the count leading zeroes instruction (`clz`) is available and can be used to speed up the find first bit operation. The use of this instruction significantly speeds up the scheduling associated with a thread blocking.

1.4 Floating Point Unit

The macro ARM_HAS_FPU is set to 1 to indicate that this CPU model has a hardware floating point unit and 0 otherwise. It does not matter whether the hardware floating point support is incorporated on-chip or is an external coprocessor.

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 ARM architecture supports a simple yet effective call and return mechanism. A subroutine is invoked via the branch and link (`bl`) instruction. This instruction saves the return address in the `lr` register. Returning from a subroutine only requires that the return address be moved into the program counter (`pc`), possibly with an offset. It is is important to note that the `bl` instruction does not automatically save or restore any registers. It is the responsibility of the high-level language compiler to define the register preservation and usage convention.

2.3 Calling Mechanism

All RTEMS directives are invoked using the `bl` instruction and return to the user application via the mechanism described above.

2.4 Register Usage

As discussed above, the ARM's call and return mechanism dos not automatically save any registers. RTEMS uses the registers `r0`, `r1`, `r2`, and `r3` as scratch registers and per ARM calling convention, the `lr` register is altered as well. 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 ARM calling conventions are followed and that the first four arguments are placed in registers `r0` through `r3`. If there are more arguments, than that, then they are place on the stack.

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

Members of the ARM family newer than Version 3 support a flat 32-bit address space with addresses ranging from 0x00000000 to 0xFFFFFFFF (4 gigabytes). Each address is represented by a 32-bit value and is byte addressable. The address may be used to reference a single byte, word (2-bytes), or long word (4 bytes). Memory accesses within this address space are performed in the endian mode that the processor is configured for. In general, ARM processors are used in little endian mode.

Some of the ARM family members such as the 920 and 720 include an MMU and thus support virtual memory and segmentation. RTEMS does not support virtual memory or segmentation on any of the ARM family members.

4 Interrupt Processing

4.1 Introduction

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

The ARM has 7 exception types:

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

Of these types, only IRQ and FIQ are handled through RTEMS's interrupt vectoring.

4.2 Vectoring of an Interrupt Handler

Unlike many other architectures, the ARM has seperate stacks for each interrupt. When the CPU receives an interrupt, it:

- switches to the exception mode corresponding to the interrupt,
- saves the Current Processor Status Register (CPSR) to the exception mode's Saved Processor Status Register (SPSR),
- masks off the IRQ and if the interrupt source was FIQ, the FIQ is masked off as well,
- saves the Program Counter (PC) to the exception mode's Link Register (LR - same as R14),
- and sets the PC to the exception's vector address.

The vectors for both IRQ and FIQ point to the _ISR_Handler function. _ISR_Handler() calls the BSP specific handler, ExecuteITHandler(). Before calling ExecuteITHandler(), registers R0-R3, R12, and R14(LR) are saved so that it is safe to call C functions. Even ExecuteITHandler() can be written in C.

4.3 Interrupt Levels

The ARM architecture supports two external interrupts - IRQ and FIQ. FIQ has a higher priority than IRQ, and has its own version of register R8 - R14, however RTEMS does not take advantage of them. Both interrupts are enabled through the CPSR.

The RTEMS interrupt level mapping scheme for the AEM is not a numeric level as on most RTEMS ports. It is a bit mapping that corresponds the enable bits's postions in the CPSR:

FIQ Setting bit 6 (0 is least significant bit) disables the FIQ.

IRQ Setting bit 7 (0 is least significant bit) disables the IRQ.

4.4 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables interrupts to level seven (7) before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than TBD microseconds on a 100 Mhz processor with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled is hand calculated. This calculation was last performed for Release ss-20020301.]

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.

4.5 Interrupt Stack

RTEMS expects the interrupt stacks to be set up in bsp_start(). The memory for the stacks is reserved in the linker script.

5 Default Fatal Error Processing

5.1 Introduction

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

5.2 Default Fatal Error Handler Operations

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

- disables processor interrupts,
- places the error code in **r0**, and
- executes an infinite loop (`while(0);` 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 XXX 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 XXX processor is reset. When the XXX is reset, the processor performs the following actions:

- The tracing bits of the status register are cleared to disable tracing.
- The supervisor interrupt state is entered by setting the supervisor (S) bit and clearing the master/interrupt (M) bit of the status register.
- The interrupt mask of the status register is set to level 7 to effectively disable all maskable interrupts.
- The vector base register (VBR) is set to zero.
- The cache control register (CACR) is set to zero to disable and freeze the processor cache.
- The interrupt stack pointer (ISP) is set to the value stored at vector 0 (bytes 0-3) of the exception vector table (EVT).
- The program counter (PC) is set to the value stored at vector 1 (bytes 4-7) of the EVT.
- The processor begins execution at the address stored in the PC.

6.3 Processor Initialization

The address of the application's initialization code should be stored in the first vector of the EVT which will allow the immediate vectoring to the application code. If the application requires that the VBR be some value besides zero, then it should be set to the required value at this point. All tasks share the same XXX's VBR value. Because interrupts are enabled automatically by RTEMS as part of the initialize executive directive, the VBR MUST be set 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.

In addition to the requirements described in the Board Support Packages chapter of the Applications User's Manual for the reset code which is executed before the call to initialize executive, the XXX version has the following specific requirements:

- Must leave the S bit of the status register set so that the XXX remains in the supervisor state.
- Must set the M bit of the status register to remove the XXX from the interrupt state.
- Must set the master stack pointer (MSP) such that a minimum stack size of MINIMUM_STACK_SIZE bytes is provided for the initialize executive directive.
- Must initialize the XXX's vector table.

Note that the BSP is not responsible for allocating or installing the interrupt stack. RTEMS does this automatically as part of initialization. If the BSP does not install an interrupt stack and – for whatever reason – an interrupt occurs before initialize_executive is invoked, then the results are unpredictable.

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 XXX version of the RTEMS CPU Dependent Information Table contains the information required to interface a Board Support Package and RTEMS on the XXX. 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 */

  /* XXX CPU family dependent stuff */
} 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.

`XXX` is where the CPU family dependent stuff goes.

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: na

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: na
- Maximum Configuration: na

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	na	NA	
Initialization	na	NA	
Task	na	NA	
Interrupt	na	NA	
Clock	na	NA	
Timer	na	na	
Semaphore	na	na	
Message	na	na	
Event	na	na	
Signal	na	na	
Partition	na	na	
Region	na	na	
Dual Ported Memory	na	na	
I/O	na	na	
Fatal Error	na	NA	
Rate Monotonic	na	na	
Multiprocessing	na	na	
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	* na =	
maximum_timers	* na =	
maximum_semaphores	* na =	
maximum_message_queues	* na =	
maximum_regions	* na =	
maximum_partitions	* na =	
maximum_ports	* na =	
maximum_periods	* na =	
maximum_extensions	* na =	
Floating Point Tasks	* na =	
Task Stacks	=	
Total Single Processor Requirements		

Description	Equation	Bytes Required
maximum_nodes	* na =	
maximum_global_objects	* na =	
maximum_proxies	* na =	

Total Multiprocessing Requirements		
Fixed System Requirements	na	
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 applica-

tion 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 MYBSP Timing Data

10.1 Introduction

The timing data for the ARM 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 ARM version of RTEMS.

10.2 Hardware Platform

All times reported except for the maximum period interrupts are disabled by RTEMS were measured using a Motorola MYBSP CPU board. The MYBSP is a 100 Mhz board with SDRAM and no numeric coprocessor. A countdown timer on this board was used to measure elapsed time with a 20 nanosecond resolution. All sources of hardware interrupts were disabled, although the interrupt level of the ARM microprocessor 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. The worst case times of the ARM9DTMI microprocessor were used for each instruction. 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 TBD to simulate a TBD Mhz processor. It should be noted that the worst case instruction times assume that the internal cache is disabled and that no instructions overlap.

10.3 Interrupt Latency

The maximum period with interrupts disabled within RTEMS is less than TBD microseconds including the instructions which disable and re-enable interrupts. The time required for the processor to vector an interrupt and for the RTEMS entry overhead before invoking the user's interrupt handler are a total of unavailable microseconds. These combine to yield a worst case interrupt latency of less than TBD + unavailable microseconds at 100 Mhz. [NOTE: The maximum period with interrupts disabled was last determined for Release ss-20020301.]

It should be noted again that the maximum period with interrupts disabled within RTEMS is hand-timed and based upon worst case (i.e. CPU cache disabled and no instruction overlap) times for a 100 Mhz processor. The interrupt vector and entry overhead time was generated on an MYBSP benchmark platform using the Multiprocessing Communications registers to generate as the interrupt source.

10.4 Context Switch

The RTEMS processor context switch time is 11 microseconds on the MYBSP benchmark platform when no floating point context is saved or restored. Additional execution time is required when a TASK_SWITCH user extension is configured. The use of the TASK_SWITCH extension is application dependent. Thus, its execution time is not considered part of the raw context switch time.

The ARM processor benchmarked does not have a floating point unit and consequently no FPU results are reported.

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

No Floating Point Contexts	11
Floating Point Contexts	
restore first FP task	NA
save initialized, restore initialized	NA
save idle, restore initialized	NA
save idle, restore idle	NA

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	43
TASK_IDENT	85
TASK_START	19
TASK_RESTART	
calling task	26
suspended task – returns to caller	23
blocked task – returns to caller	28
ready task – returns to caller	24
suspended task – preempts caller	35
blocked task – preempts caller	64
ready task – preempts caller	64
TASK_DELETE	
calling task	55
suspended task	42
blocked task	43
ready task	43
TASK_SUSPEND	
calling task	21
returns to caller	9
TASK_RESUME	
task readied – returns to caller	10
task readied – preempts caller	18
TASK_SET_PRIORITY	
obtain current priority	7
returns to caller	15
preempts caller	29
TASK_MODE	
obtain current mode	4
no reschedule	4
reschedule – returns to caller	13
reschedule – preempts caller	30
TASK_GET_NOTE	8
TASK_SET_NOTE	7
TASK_WAKE_AFTER	
yield – returns to caller	5
yield – preempts caller	17
TASK_WAKE_WHEN	33

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	unavailable
returns to interrupted task	unavailable
returns to preempting task	unavailable
Interrupt Exit Overhead	
returns to nested interrupt	unavailable
returns to interrupted task	unavailable
returns to preempting task	unavailable

10.8 Clock Manager

CLOCK_SET	21
CLOCK_GET	1
CLOCK_TICK	10

10.9 Timer Manager

TIMER_CREATE	8
TIMER_IDENT	83
TIMER_DELETE	
inactive	11
active	12
TIMER_FIRE_AFTER	
inactive	14
active	15
TIMER_FIRE_WHEN	
inactive	21
active	21
TIMER_RESET	
inactive	14
active	15
TIMER_CANCEL	
inactive	7
active	9

10.10 Semaphore Manager

SEMAPHORE_CREATE	27
SEMAPHORE_IDENT	97
SEMAPHORE_DELETE	24
SEMAPHORE_OBTAIN	
available	5
not available – NO_WAIT	5
not available – caller blocks	28
SEMAPHORE_RELEASE	
no waiting tasks	9
task readied – returns to caller	14
task readied – preempts caller	22

10.11 Message Manager

MESSAGE_QUEUE_CREATE	54
MESSAGE_QUEUE_IDENT	83
MESSAGE_QUEUE_DELETE	32
MESSAGE_QUEUE_SEND	
no waiting tasks	14
task readied – returns to caller	16
task readied – preempts caller	25
MESSAGE_QUEUE_URGENT	
no waiting tasks	14
task readied – returns to caller	16
task readied – preempts caller	25
MESSAGE_QUEUE_BROADCAST	
no waiting tasks	11
task readied – returns to caller	35
task readied – preempts caller	42
MESSAGE_QUEUE_RECEIVE	
available	15
not available – NO_WAIT	10
not available – caller blocks	29
MESSAGE_QUEUE_FLUSH	
no messages flushed	8
messages flushed	9

10.12 Event Manager

EVENT_SEND	
no task readied	7
task readied – returns to caller	13
task readied – preempts caller	22
EVENT_RECEIVE	
obtain current events	1
available	14
not available – NO_WAIT	7
not available – caller blocks	24

10.13 Signal Manager

SIGNAL_CATCH	7
SIGNAL_SEND	
returns to caller	16
signal to self	29
EXIT ASR OVERHEAD	
returns to calling task	22
returns to preempting task	25

10.14 Partition Manager

PARTITION_CREATE	27
PARTITION_IDENT	83
PARTITION_DELETE	18
PARTITION_GET_BUFFER	
available	14
not available	10
PARTITION_RETURN_BUFFER	10

10.15 Region Manager

REGION_CREATE	29
REGION_IDENT	84
REGION_DELETE	17
REGION_GET_SEGMENT	
available	14
not available – NO_WAIT	18
not available – caller blocks	56
REGION_RETURN_SEGMENT	
no waiting tasks	15
task readied – returns to caller	40
task readied – preempts caller	58

10.16 Dual-Ported Memory Manager

PORT_CREATE	18
PORT_IDENT	83
PORT_DELETE	19
PORT_INTERNAL_TO_EXTERNAL	6
PORT_EXTERNAL_TO_INTERNAL	6

10.17 I/O Manager

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

10.18 Rate Monotonic Manager

RATE_MONOTONIC_CREATE	18
RATE_MONOTONIC_IDENT	83
RATE_MONOTONIC_CANCEL	18
RATE_MONOTONIC_DELETE	
active	23
inactive	21
RATE_MONOTONIC_PERIOD	
initiate period – returns to caller	25
conclude period – caller blocks	20
obtain status	13

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.