8

Timing Specification

8.1 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.

8.2 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.

8.2.1 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).

8.2.2 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:

1. *the longest period of time interrupts are disabled by the executive,*
2. *the overhead required by the executive at the beginning of each ISR,*
3. *the time required for the CPU to vector the interrupt, and*
4. *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.

8.2.3 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.

8.2.4 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.

8.3 Methodology

8.3.1 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.

8.3.2 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.

8.3.3 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.

8.3.4 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.

8.3.5 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.