18

Rate Monotonic Manager

18.1 Introduction

The **rate monotonic manager** provides facilities to implement tasks which execute in a periodic fashion. The directives provided by the rate monotonic manager are:

Name	Directive Description
rate_monotonic_create	Create a rate monotonic period
rate_monotonic_ident	Get ID of a period
rate_monotonic_cancel	Cancel a period
rate_monotonic_delete	Delete a rate monotonic period
rate_monotonic_period	Conclude current/Start next period

18.2 Background

The rate monotonic manager provides facilities to manage the execution of periodic tasks. This manager was designed to support application designers who utilize the **Rate Monotonic Scheduling Algorithm** (RMS) to insure that their periodic tasks will meet their deadlines, even under transient overload conditions. Although designed for hard real-time systems, the services provided by the rate monotonic manager may be used by any application which requires periodic tasks.

18.2.1 Definitions

A **periodic task** is one which must be executed at a regular interval. The interval between successive iterations of the task is referred to as its **period**. Periodic tasks can be characterized by the length of their period and execution time. The period and execution time of a task can be used to determine the processor utilization for that task. **Processor utilization** is the percentage of processor time used and can be calculated on a per-task or system-wide basis. Typically, the task's worst-case execution time will be less than its period. For example, a periodic task's requirements may state that it should execute for 10 milliseconds every 100 milliseconds. Although the execution time may be the average, worst, or best case, the worst-case execution time is more appropriate for use when analyzing system behavior under transient overload conditions.

In contrast, an **aperiodic task** executes at irregular intervals and has only a soft deadline. In other words, the deadlines for aperiodic tasks are not rigid, but adequate response times are desirable. For example, an aperiodic task may process user input from a terminal.

Finally, a **sporadic task** is an aperiodic task with a hard deadline and minimum interarrival time. The **minimum interarrival time** is the minimum period of time which exists between successive iterations of the task. For example, a sporadic task could be used to process the pressing of a fire button on a joystick. The mechanical action of the fire button insures a minimum time period between successive activations, but the missile must be launched by a hard deadline.

18.2.2 Rate Monotonic Scheduling Algorithm

The **Rate Monotonic Scheduling Algorithm** (RMS) is important to real-time systems designers because it allows one to guarantee that a set of tasks is **schedulable**. A set of tasks is said to be **schedulable** if all of the tasks can meet their deadlines. RMS provides a set of rules which can be used to perform a **guaranteed schedulability analysis** for a task set. This analysis determines whether a task set is schedulable under worst-case conditions and emphasizes the predictability of the system's behavior. It has been proven that:

> *RMS is an optimal static priority algorithm for scheduling independent, preemptible, periodic tasks on a single processor.*

RMS is optimal in the sense that if a set of tasks can be scheduled by any static priority algorithm, then RMS will be able to schedule that task set. RMS bases it

schedulability analysis on the processor utilization level below which all deadlines can be met.

RMS calls for the static assignment of task priorities based upon their period. The shorter a task's period, the higher its priority. For example, a task with a 1 millisecond period has higher priority than a task with a 100 millisecond period. If two tasks have the same period, than RMS does not distinguish between the tasks. However, **RTEMS** specifies that when given tasks of equal priority, the task which has been ready longest will execute first. RMS's priority assignment scheme does not provide one with exact numeric values for task priorities. For example, consider the following task set and priority assignments:

Task	Period (in milliseconds)	Priority
1	100	Low
2	50	Medium
3	50	Medium
4	25	High

RMS only calls for task 1 to have the lowest priority, task 4 to have the highest priority, and tasks 2 and 3 to have an equal priority between that of tasks 1 and 4. The actual **RTEMS** priorities assigned to the tasks must only adhere to those guidelines.

Many applications have tasks with both hard and soft deadlines. The tasks with hard deadlines are typically referred to as the **critical task set**, with the soft deadline tasks being the **non-critical task set**. The critical task set can be scheduled using RMS, with the non-critical tasks not executing under transient overload, by simply assigning priorities such that the lowest priority critical task (i.e. longest period) has a higher priority than the highest priority non-critical task. Although RMS may be used to assign priorities to the non-critical tasks, it is not necessary. In this instance, schedulability is only guaranteed for the critical task set.

18.2.3 Schedulability Analysis

RMS allows application designers to insure that tasks can meet all deadlines, even under transient overload, without knowing exactly when any given task will execute by applying proven schedulability analysis rules.

18.2.3.1 Assumptions

The schedulability analysis rules for RMS were developed based on the following assumptions:

- *The requests for all tasks for which hard deadlines exist are periodic, with a constant interval between requests.*
- *Each task must complete before the next request for it occurs.*
- *The tasks are independent in that a task does not depend on the initiation or completion of requests for other tasks.*
- *The execution time for each task without preemption or interruption is constant and does not vary.*
- *Any non-periodic tasks in the system are special. These tasks displace periodic tasks while executing and do not have hard, critical deadlines.*

Once the basic schedulability analysis is understood, some of the above assumptions can be relaxed and the side-effects accounted for.

18.2.3.2 Processor Utilization Rule

The **Processor Utilization Rule** requires that processor utilization be calculated based upon the period and execution time of each task. The fraction of processor time spent executing task **index** is Time(index) / Period(index). The processor utilization can be calculated as follows:

```
Utilization = 0
for index = 1 to maximum_tasks
  Utilization = Utilization + (Time(index)/Period(index))
```

To insure schedulability even under transient overload, the processor utilization must adhere to the following rule:

$$Utilization = maximum_tasks * (2^{(1/maximum_tasks)} - 1)$$

As the number of tasks increases, the above formula approaches ln(2) for a worst-case utilization factor of approximately 0.693. Many tasks sets can be scheduled with a greater utilization factor. In fact, the average processor utilization threshold for a randomly generated task set is approximately 0.88.

18.2.3.3 Processor Utilization Rule Example

This example illustrates the application of the **Processor Utilization Rule** to an application with three critical periodic tasks. The following table details the RMS priority, period, execution time, and processor utilization for each task:

Task	RMS Priority	Period	Execution Time	Processor Utilization
1	High	100	15	0.15
2	Medium	200	50	0.25
3	Low	300	100	0.33

The total processor utilization for this task set is 0.73 which is below the upper bound of $3 * (2^{(1/3)} - 1)$, or 0.779, imposed by the **Processor Utilization Rule**. Therefore, this task set is guaranteed to be schedulable using RMS.

18.2.3.4 First Deadline Rule

If a given set of tasks do exceed the processor utilization upper limit imposed by the **Processor Utilization Rule**, they can still be guaranteed to meet all their deadlines by application of the **First Deadline Rule**. This rule can be stated as follows:

For a given set of independent periodic tasks, if each task meets its first deadline when all tasks are started at the same time, then the deadlines will always be met for any combination of start times.

A key point with this rule is that **ALL** periodic tasks are assumed to start at the exact same instant in time. Although this assumption may seem to be invalid, **RTEMS** makes it quite easy to insure. By having a non-preemptible user initialization task, all application tasks, regardless of priority, can be created and started before the initialization deletes itself. This technique insures that all tasks begin to compete for execution time at the same instant -- when the user initialization task deletes itself.

18.2.3.5 First Deadline Rule Example

The **First Deadline Rule** can insure schedulability even when the **Processor Utilization Rule** fails. The example below is a modification of the **Processor Utilization Rule** example where task execution time has been increased from 15 to 25 units. The following table details the RMS priority, period, execution time, and processor utilization for each task:

Task	RMS Priority	Period	Execution Time	Processor Utilization
1	High	100	25	0.25
2	Medium	200	50	0.25
3	Low	300	100	0.33

The total processor utilization for the modified task set is 0.83 which is above the upper bound of 3 * ($2^{(1/3)}$ - 1), or 0.779, imposed by the **Processor Utilization Rule**. Therefore, this task set is not guaranteed to be schedulable using RMS. However, the **First Deadline Rule** can guarantee the schedulability of this task set. This rule calls for one to examine each occurrence of deadline until either all tasks have met their deadline or one task failed to meet its first deadline. The following table details the time of each deadline occurrence, the maximum number of times each task may have run, the total execution time, and whether all the deadlines have been met.

Deadline Time	Task 1	Task 2	Task 3	Total Execution Time	All Deadlines Met?
100	1	1	1	25 + 50 + 100 = 175	NO
200	2	1	1	50 + 50 + 100 = 200	YES

The key to this analysis is to recognize when each task will execute. For example at time 100, task 1 must have met its first deadline, but tasks 2 and 3 may also have begun execution. In this example, at time 100 tasks 1 and 2 have completed execution and thus have met their first deadline. Tasks 1 and 2 have used (25 + 50) = 75 time units, leaving (100 - 75) = 25 time units for task 3 to begin. Because task 3 takes 100 ticks to execute, it will not have completed execution at time 100. Thus at time 100, all of the tasks except task 3 have met their first deadline.

At time 200, task 1 must have met its second deadline and task 2 its first deadline. As a result, of the first 200 time units, task 1 uses (2 * 25) = 50 and task 2 uses 50, leaving (200 - 100) time units for task 3. Task 3 requires 100 time units to execute, thus it will have completed execution at time 200. Thus, all of the tasks have met their first deadlines at time 200, and the task set is schedulable using the **First Deadline Rule**.

18.2.3.6 Relaxation of Assumptions

The assumptions used to develop the RMS schedulability rules are uncommon in most real-time systems. For example, it was assumed that tasks have constant unvarying execution time. It is possible to relax this assumption, simply by using the worst-case execution time of each task.

Another assumption is that the tasks are independent. This means that the tasks do not wait for one another or contend for resources. This assumption can be relaxed by accounting for the amount of time a task spends waiting to acquire resources. Similarly, each task's execution time must account for any I/O performed and any **RTEMS** directive calls.

In addition, the assumptions did not account for the time spent executing interrupt service routines. This can be accounted for by including all the processor utilization by interrupt service routines in the utilization calculation. Similarly, one should also account for the impact of delays in accessing local memory caused by direct memory access and other processors accessing local dual-ported memory.

The assumption that nonperiodic tasks are used only for initialization or failure-recovery can be relaxed by placing all periodic tasks in the critical task set. This task set can be scheduled and analyzed using RMS. All nonperiodic tasks are placed in the non-critical task set. Although the critical task set can be guaranteed to execute even under transient overload, the non-critical task set is not guaranteed to execute.

In conclusion, the application designer must be fully cognizant of the system and its run-time behavior when performing schedulability analysis for a system using RMS. Every hardware and software factor which impacts the execution time of each task must be accounted for in the schedulability analysis.

18.2.4 Further Reading

For more information on **Rate Monotonic Scheduling** and its schedulability analysis, the reader is referred to the following:

- *C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment."* ***Journal of the Association of Computing Machinery****. January 1973. pp. 46-61.*
- *John Lehoczky, Lui Sha, and Ye Ding. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior."* ***IEEE Real-Time Systems Symposium.*** *1989. pp. 166-171.*

- *Lui Sha and John Goodenough. "Real-Time Scheduling Theory and Ada." **IEEE Computer**. April 1990. pp. 53-62.*
- *Alan Burns. "Scheduling hard real-time systems: a review." **Software Engineering Journal**. May 1991. pp. 116-128.*

18.3 Operations

18.3.1 Creating a Rate Monotonic Period

The **rate_monotonic_create** directive creates a rate monotonic period which is to be used by the calling task to delineate a period. **RTEMS** allocates a **Period Control Block** (PCB) from the PCB free list. This data structure is used by **RTEMS** to manage the newly created rate monotonic period. **RTEMS** returns a unique period ID to the application which is used by other rate monotonic manager directives to access this rate monotonic period.

18.3.2 Manipulating a Period

The **rate_monotonic_period** directive is used to establish and maintain periodic execution utilizing a previously created rate monotonic period. Once initiated by the **rate_monotonic_period** directive, the period is said to run until it either expires or is reinitiated. The state of the rate monotonic period results in one of the following scenarios:

- *If the rate monotonic period is running, the calling task will be blocked for the remainder of the outstanding period and, upon completion of that period, the period will be reinitiated with the specified **period**.*
- *If the rate monotonic period is not currently running and has not expired, it is initiated with a length of **period** ticks and the calling task returns immediately.*
- *If the rate monotonic period has expired before the task invokes the **rate_monotonic_period** directive, the period will be initiated with a length of **period** ticks and the calling task returns immediately with a timeout error status.*

18.3.3 Obtaining a Period's Status

If the **rate_monotonic_period** directive is invoked with a **period** of **PERIOD_STATUS** ticks, the current state of the specified rate monotonic period will be returned. The following table details the relationship between the period's

status and the directive status code returned by the **rate_monotonic_period** directive:

Directive Status	Period State
SUCCESSFUL	**period is running**
TIMEOUT	**period has expired**
NOT_DEFINED	**period has never been initiated**

Obtaining the status of a rate monotonic period does not alter the state or length of that period.

18.3.4 Canceling a Period

The **rate_monotonic_cancel** directive is used to stop the period maintained by the specified rate monotonic period. The period is stopped and the rate monotonic period can be reinitiated using the **rate_monotonic_period** directive.

18.3.5 Deleting a Rate Monotonic Period

The **rate_monotonic_delete** directive is used to delete a rate monotonic period. If the period is running and has not expired, the period is automatically canceled. The rate monotonic period's control block is returned to the PCB free list when it is deleted. A rate monotonic period can be deleted by a task other than the task which created the period.

18.3.6 Examples

The following sections illustrate common uses of rate monotonic periods to construct periodic tasks.

18.3.6.1 Simple Periodic Task

This example consists of a single periodic task which, after initialization, executes every 100 clock ticks.

```
rtems_task Periodic_task()
{
  rtems_name        name;
  rtems_id          period;
  rtems_status_code status;

  name = build_name( 'P', 'E', 'R', 'D' );
  status = rate_monotonic_create( name, &period );

  while ( 1 ) {

    status = rate_monotonic_period( period, 100 );
    if ( status == TIMEOUT )
      break;

    /* Perform some periodic actions */

  }

  /* missed period so delete period and SELF */
  status = rate_monotonic_delete( period );
  status = task_delete( SELF );

}
```

The above task creates a rate monotonic period as part of its initialization. The first time the loop is executed, the **rate_monotonic_period** directive will initiate the period for 100 ticks and return immediately. Subsequent invocations of the **rate_monotonic_period** directive will result in the task blocking for the remainder of the 100 tick period. If, for any reason, the body of the loop takes more than 100 ticks to execute, the **rate_monotonic_period** directive will return the **TIMEOUT** status. If the above task misses its deadline, it will delete the rate monotonic period and itself.

18.3.6.2 Task with Multiple Periods

This example consists of a single periodic task which, after initialization, performs two sets of actions every 100 clock ticks. The first set of actions is performed in the first forty clock ticks of every 100 clock ticks, while the second set of actions is performed between the fortieth and seventieth clock ticks. The last thirty clock ticks are not used by this task.

```
task Periodic_task()
{
  rtems_name        name_1;
  rtems_name        name_2;
  rtems_id          period_1;
  rtems_id          period_2;
  rtems_status_code status;

  name_1 = build_name( 'P', 'E', 'R', '1' );
  name_2 = build_name( 'P', 'E', 'R', '2' );

  status = rate_monotonic_create( name_1, &period_1 );
  status = rate_monotonic_create( name_2, &period_2 );
  while ( 1 ) {
    status = rate_monotonic_period( period_1, 100 );
    if ( status == TIMEOUT )
      break;

    status = rate_monotonic_period( period_2, 40 );
    if ( status == TIMEOUT )
      break;

    /* Perform first set of actions between clock
     * ticks 0 and 39 of every 100 ticks.
     */

    status = rate_monotonic_period( period_2, 30 );
    if ( status == TIMEOUT )
      break;

    /* Perform second set of actions between clock
     * 40 and 69 of every 100 ticks.
     */

    /* Check to make sure we didn't miss
     * the period_2 period.
     */

    status = rate_monotonic_period( period_2, STATUS );
    if ( status == TIMEOUT )
      break;

    rate_monotonic_cancel( period_2 );
  }
  /* missed period so delete period and SELF */
  status = rate_monotonic_delete( period_1 );
  status = rate_monotonic_delete( period_2 );
  status = task_delete( SELF );
}
```

The above task creates two rate monotonic periods as part of its initialization. The first time the loop is executed, the **rate_monotonic_period** directive will initiate the **period_1** period for 100 ticks and return immediately. Subsequent invocations of the **rate_monotonic_period** directive for **period_1** will result in the task blocking for the remainder of the 100 tick period. The **period_2** period is used to

control the execution time of the two sets of actions within each 100 tick period established by **period_1**. The **rate_monotonic_cancel(period_2)** call is performed to insure that the **period_2** period does not expire while the task is blocked on the **period_1** period. If this cancel operation were not performed, every time the **rate_monotonic_period(period_1, 40)** call is executed, except for the initial one, a directive status of **TIMEOUT** is returned. It is important to note that every time this call is made, the **period_1** period will be initiated immediately and the task will not block.

If, for any reason, the task misses any deadline, the **rate_monotonic_period** directive will return the **TIMEOUT** directive status. If the above task misses its deadline, it will delete the rate monotonic periods and itself.

18.4 Directives

This section details the rate monotonic manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

18.4.1 RATE_MONOTONIC_CREATE - Create a rate monotonic period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_create(
  rtems_name  name,
  rtems_id   *id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	rate monotonic period created successfully
INVALID_NAME	invalid task name
TOO_MANY	too many periods created

DESCRIPTION:

This directive creates a rate monotonic period. The assigned rate monotonic id is returned in **id**. This id is used to access the period with other rate monotonic manager directives. For control and maintenance of the rate monotonic period, **RTEMS** allocates a PCB from the local PCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

18.4.2 RATE_MONOTONIC_IDENT - Get ID of a period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_ident(
  rtems_name  name,
  rtems_id   *id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	period identified successfully
INVALID_NAME	period name not found

DESCRIPTION:

This directive obtains the period **id** associated with the period **name** to be acquired. If the period name is not unique, then the period id will match one of the periods with that name. However, this period id is not guaranteed to correspond to the desired period. The period id is used to access this period in other rate monotonic manager directives.

NOTES:

This directive will not cause the running task to be preempted.

18.4.3 RATE_MONOTONIC_CANCEL - Cancel a period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_cancel(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	period canceled successfully
INVALID_ID	invalid rate monotonic period id
NOT_OWNER_OF_RESOURCE	rate monotonic period not created by calling task

DESCRIPTION:

This directive cancels the rate monotonic period **id**. This period will be reinitiated by the next invocation of **rate_monotonic_period** with **id**.

NOTES:

This directive will not cause the running task to be preempted.

The rate monotonic period specified by **id** must have been created by the calling task.

18.4.4 RATE_MONOTONIC_DELETE - Delete a rate monotonic period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	period deleted successfully
INVALID_ID	invalid rate monotonic period id

DESCRIPTION:

This directive deletes the rate monotonic period specified by **id**. If the period is running, it is automatically canceled. The PCB for the deleted period is reclaimed by **RTEMS**.

NOTES:

This directive will not cause the running task to be preempted.

A rate monotonic period can be deleted by a task other than the task which created the period.

18.4.5 RATE_MONOTONIC_PERIOD - Conclude current/Start next period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_period(
  rtems_id       id,
  rtems_interval length
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	period initiated successfully
INVALID_ID	invalid rate monotonic period id
NOT_OWNER_OF_RESOURCE	period not created by calling task
NOT_DEFINED	period has never been initiated
TIMEOUT	period has expired

DESCRIPTION:

This directive initiates the rate monotonic period **id** with a length of **period** ticks. If **id** is running, then the calling task will block for the remainder of the period before reinitiating the period with the specified period. If **id** was not running (either expired or never initiated), the period is immediately initiated and the directive returns immediately.

If invoked with a **period** of **PERIOD_STATUS** ticks, the current state of **id** will be returned. The directive status indicates the current state of the period. This does not alter the state or **period** of the period.

NOTES:

This directive will not cause the running task to be preempted.