Rate Monotonic Analysis

The solution to multitasking problems?

Jim Cooling
Lindentree Associates/Loughborough University

Abstract: A major issue in the design of real-time multitasking systems is that of timing correctness. Signal sampling, control calculations, output commands, etc.: all must be done in a deterministic way to ensure systems meet their performance objectives. In recent years one technique – Rate Monotonic Analysis (RMA) - has frequently been promoted as a way of attaining these goals. Unfortunately the limitations of RMA haven't always been as clearly presented, which can lead to a somewhat over-optimistic belief in its effectiveness.

This paper sets out to describe the background and origins of RMA, highlighting in particular the assumptions underlying its theoretical development. The impact of these assumptions on the performance of real-time (especially hard, fast) systems is discussed in some detail. From this it can be seen that fundamental extensions and changes must be made to RMA scheduling techniques to cope with demanding performance requirements. Moreover, for anything in the least bit complex, tool support is essential to fully examine all aspects of system behaviour.

Rate Monotonic Analysis

The solution to multitasking problems?

1. Setting the scene.

If you look at the software structure of embedded microsystems of early days, certain consistent patterns emerge. Routine low-level activities were performed by a continuous single-loop background task, all other activities being controlled by interrupts. These included functions executed at regular time intervals (synchronous or *periodic* tasks) and those done on demand (asynchronous or *aperiodic* tasks). Examples of periodic tasks include control loop calculations, data monitoring functions, display refreshing, etc. Aperiodic tasks are generally used to deal with events whose occurrences are unpredictable (from a time point of view that is), and, most important, require a fast response. Examples include handling push-button commands (e.g. bomb release), dealing with alarm functions (fuel tank flame detection, for instance) and protecting machinery systems (e.g. machinery running **and** safety guard lifted). This software organization may be regarded as the poor man's multitasking structure.

In developing the software for these systems the designer – and it frequently was a single person - was presented with two major objectives. First, the system had to satisfy its functional objectives, i.e. *functional correctness*. Second, it had to meet its timing requirements, that is, timing or *temporal correctness*. In older (and in particular smaller) embedded designs it wasn't especially difficult to achieve functional correctness (this doesn't imply that it was by any means easy, however). Timing aspects, similarly, didn't usually produce major problems - provided the CPU had adequate spare processing capacity (a most important proviso). The success of real-time embedded processor systems during the 70's and 80's speaks for itself. However, in recent years, the trend in embedded systems is that software has become larger, more complex and team-based. As a way of coping with this, developers have increasingly turned to the use of multitasking techniques in place of simple interrupt-driven structures. This, without a doubt, certainly simplified the task of implementing large, complex systems; it also made it easier to produce designs which are functionally correct. Unfortunately, the same cannot be said for timing correctness; if anything the reverse is true. And that can raise serious problems for the designer of embedded systems, especially the hard/fast ones.

The core mechanism in a multitasking structure is the real-time kernel of the operating systems. Its primary function is the safe and secure scheduling and dispatching of the application tasks (performed by the 'scheduler'). This it does in accordance with a set of rules - an *algorithm* - called the scheduling policy. Where time behaviour is critical, we would dearly love to employ a scheduler whose performance is fully predictable – a *deterministic* scheduler. Now put yourself in the position of a software designer faced with developing a new, technically-challenging, system which has significant performance (time) demands. Moreover, it has been decided to base this on the use of a real-time operating system. The key question is 'what scheduling policy should be used to get the best performance from the system?'. Well, if you were presented with the following comments (taken from different technical articles relating to scheduling and schedulers in real-time systems) I'm sure the decision would be an easy one:

'Rate Monotonic Analysis (RMA) is a simple, practical mathematically sound way to guarantee schedulability in real-time systems.'

'RMA can guarantee if a system is schedulable with a given set of requirements'.

'Thus this result means that the tasks will always meet their deadlines, regardless of how and when the system schedules the task'.

Very impressive indeed; this looks like a sure-fire thing. But are things really so straightforward? The answer, as we will see, is no.

2. The basics of rate monotonic analysis.

Rate monotonic analysis, RMA (known originally for 20 years as rate monotonic scheduling, RMS) is based on the use of a priority pre-emptive task scheduling policy. Task priorities (P) are set according to task periods (Tp), as follows:

$$P = 1/Tp$$

Thus the task with the shortest period has highest priority, followed by the task having the second shortest period; following this is the one with the third shortest period, and so on to the end one. Thus as we go down the set of tasks, periods always increase – a monotonic sequence. A task which is ready to run can always pre-empt a lower priority running task.

The milestone paper on rate monotonic work was that by Liu and Layland in 1973 [1]. They set out to answer the question of how a designer could guarantee, in a multitasking scheme, to produce a feasible task schedule. To put it another way, to guarantee that the system would meet its requirements within specified time frame(s). A key factor in their calculations is that of processor utilization (U), defined by them to be *the fraction of processor time spent in the execution of the task set*. Another term which needs to be understood is that of 'full utilization'. At first sight this could be interpreted as being the case where the processor is fully loaded (i.e. no spare capacity, U = 1.0). Not so, however, for rate monotonic scheduling. Here it describes a very specific situation: where a given task schedule is feasible but *any* increase in execution times leads to failure. We might, for instance, find that only 80% of the processor time is spent executing tasks; yet any attempt to use the spare capacity will cause task(s) to miss deadlines. Take it on trust that for a given processor having:

- A fixed set of tasks and
- A fixed total computation time for all tasks.

the 'full utilization' figure is **not** a fixed value. It depends on individual task timings and the relative activation time of tasks.

Liu and Layland showed that in a system having n tasks, the lowest utilization figure for a guaranteed feasible schedule is:

$$U = n\,(2^{1/n} - 1)$$

For example, with two tasks: $U = 2\,(2^{1/2} - 1) = 2\,(0.414) = 0.828$ or 82.8%.

As n increases the value of U decreases, eventually having a value of 0.693 when n is infinite. This means that if more than 69.3% of the processor time is spent executing tasks, we may not meet our deadlines (i.e. there is no guarantee that the schedule is feasible). More usefully put; a task set scheduled using RMS is guaranteed to be feasible provided its utilization does not exceed 0.693.

This seems fine; unfortunately the underlying assumptions of the theory contain a number of major weaknesses where real-time systems are concerned. These are:

- A1: All tasks are periodic.
- A2: Tasks are independent and non-interacting (no precedence or exclusion relationships are involved).
- A3: Task deadline and period are considered to be synonymous (that is, the task deadline is the time at which it is next due to be re-run).
- A4: The execution time of each task is constant and doesn't vary over time.
- A5: All tasks are equally important – criticality doesn't enter the equation.
- A6: Aperiodic tasks are limited to initialization and error-recovery routines, and do not have hard, critical deadlines.

Let us now look at how these factors impact on the performance of a multitasking system.

3. Periodic and independent tasks – a simple performance evaluation.

To see how rate monotonic scheduling works in practice, let us take an example of a simple (somewhat idealistic) two-task system. The tasks have the following attributes:

Task	Period (Tp)	Computation Time (Tc)	Priority
Task A	5 mSecs.	2 mSecs.	1 (highest)
Task B	10 mSecs	2 mSecs	2

The utilization bound for guaranteed schedulability is 82.8%; the actual utilization is

$$Tc_a/Tp_a + Tc_b/Tp_b = 2/5 + 2/10 = 0.5 \text{ or } 50\%.$$

Thus the schedule is feasible, as demonstrated in fig.1.

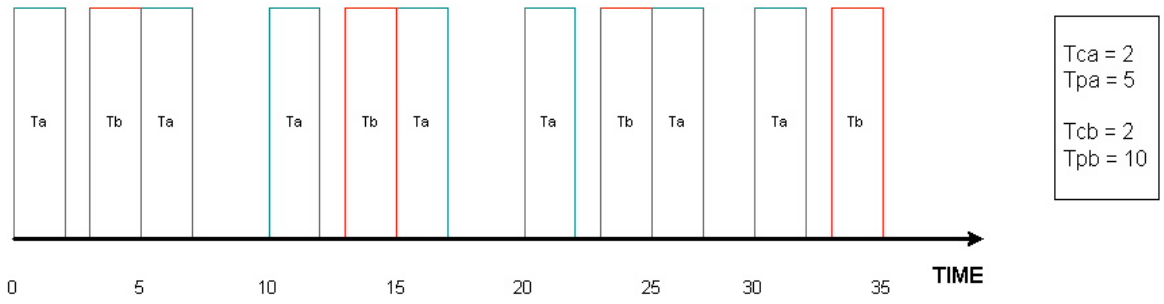

Fig. 1 A simple rate monotonic schedule – Staggered activation of tasks

Observe that the times at which the tasks are activated are staggered. Task A begins executing at time t = 0, completes at t = (0+2), is activated again at t = (0+5), and so on. Task B begins executing at time t = (0+3), completes at t = (0+5), is activated again at t = (0+13), etc. Clearly there is no problem with this schedule.

Now consider the situation where both tasks are readied simultaneously, fig.2.

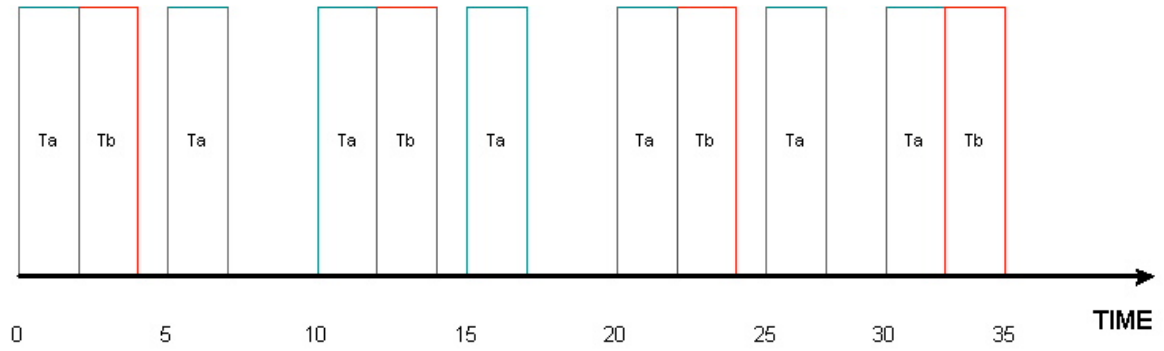

Fig. 2 A simple rate monotonic schedule –Simultaneous activation of tasks

At t = 0, although both tasks become ready ('released'), only task A is executed by the processor – it has highest priority. It finishes execution at t = (0+2), at which time Task B begins executing (completing at t = (0+4)). Simultaneous release of all tasks represents the worst case scheduling condition, but we've still managed to meet our scheduling objectives – to meet the deadlines.

Regrettably, the definition that period equals deadline is may not be acceptable in many situations. To demonstrate this, take a new two-task example:

Task	Period (Tp)	Computation Time (Tc)	Priority
Task A	5 mSecs.	2 mSecs.	1 (highest)
Task B	9 mSecs	2 mSecs	2

In the first example there was a simple integer relationship between the task periods. The result is that, no matter when the tasks are activated, the pattern of their relationship (the

'phasing') remains constant. However, in the second case it is more complex; the phasing changes over time, as shown in fig.3.

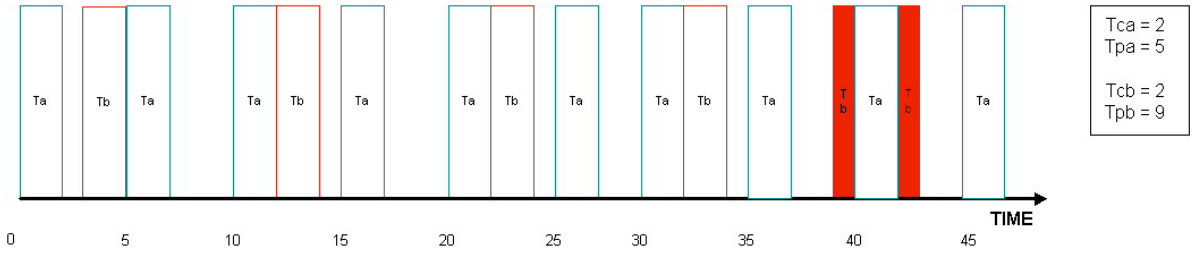

Figure 3 A simple rate monotonic schedule - Task overlap

Fig. 3 A simple rate monotonic schedule – Task overlap

Here, at time t = 0, the processor begins to execute task A, completing at t = (0+2). Note that as this task is the highest priority one, we can immediately mark all its running states on the diagram.

Task B starts to run at t = (0+3). In order to meet its deadlines it must have completed successive executions and be ready to re-run at the following times:

Run	1	2	3	4	5
Deadline	(0+12)	(0+21)	(0+30)	(0+39)	(0+48)

It can be seen that it finishes the first time at t = (0+5), well within its deadline. On its second activation 9 mSec. later (at t = (0+12)), it commences as soon as task A finishes. It is readied for the third time at t = (0+ 21) but unfortunately task 1 happens to be running. Only at t = (0+22) can task B begin to run, completing at t = (0+24). The rest of the diagram is self-explanatory, showing the relative shift in the phasing of the two tasks.

In terms of the basic criteria of RMA the schedule is still a feasible one; task B *always* completes before it is next due to run. But look how the changing time phasing of the tasks has varied task B's start and completion times. You can see that the time interval between successive activations of task B follows the pattern 9, 10, 10, 7, 9, 10, 10, 7, 9 … when nominally it is 9 mSecs. Observe that the actual time between B being readied and delivering its results follows the pattern 2, 2, 3, 4, 4, 2, 2, 3, 4, 4, 2 … This variation is called jitter (more commonly, task jitter is taken to mean variations in the task starting time).

We need to question the significance or importance of jitter in the timing schedule. In effect it depends on the application. For digital closed-loop control systems it may, as pointed out by Rekasius [2], be very important. To quote: *Intermittent computer interruptions result in the deterioration of control quality and may even render the system unstable. There are several reasons for the interruption in computer control …. If the computer has not finished a more important t ask, the control law computation may be postponed.* So what can we do? One solution proposed by Sha and Goodenough [3] is as follows: *Jitter requirements are a special kind of deadline requirement. One way of meeting an output jitter requirement using rate monotonic scheduling is to have a normal periodic task compute a result and place it in a memory-mapped I/O buffer before it is time to send the*

value. RM scheduling theorems can be used to ensure that the value is computed in time. A hardware timer of the operating system clock interrupt handler routine can then initiate the device I/O precisely at the required time. A similar approach can be used for input jitter requirements. While this approach solves one problem it unfortunately introduces others in terms of additional complexity and overheads. Only knowledge of the problem domain will show whether the technique is an appropriate one.

Now, what of the effect in the variation in the time before readying a task and have it complete? In the first four activations of task B the problem is essentially limited that of jitter in the starting times. For each case the actual elapsed time between the task being put into the run state and completing execution (the response time) is 2 mSec. However, the fifth activation is quite different. Here task B starts at t = (0+39) but is suspended after 1 mSec. It resumes after task A completes, 2 mSec. later, finishing at t = (0+43), the response time being 4 mSec. What, we must ask, are the consequences of this effect? And are they truly serious?

The behaviour just described can, in fact, produce at least three distinct problems: dead-time effects in control systems, relative timing errors and loss of system functions. Even worse, these are not mutually exclusive; all three can occur simultaneously. We'll illustrate these with a few simple scenarios, all relating to the scheduling of fig.3.

Example 1: a fast closed-loop control system. Here the computation time is effectively a dead-time in the loop; its effect must be taken into account when devising the control algorithm, taking into account system dynamic behaviour and stability. But, as shown, on the fifth activation, there is a 100% change in the dead-time. This will certainly have an adverse effect on closed-loop performance and in extreme cases may lead to limit cycling or even instability.

Example 2: a process control system. When task B is activated it starts a chemical process, monitoring the operation as it runs. Our design calls for us to measure a parameter (and act on its value) at approximately 1.5 mSec. into the operation. All is fine for the first four runs of the task. However, trouble starts on run five; the parameter will be measured at 3.5 (and not 1.5) mSecs. from the start of task activation. Its value may well be significantly different from that which pertained 2 mSecs. earlier. Subsequent processing could lead to deterioration in the quality of the product, increased wastage or even physical damage.

Example 3: a production engineering application. This involves the checking of components as they move, at speed, through a conveyor system, fig.4(a).

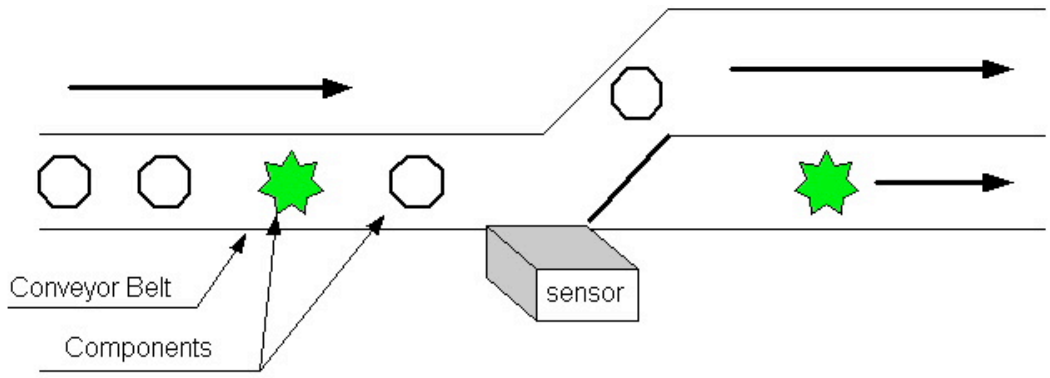

Figure 4 (a)

Fig. 4 Example production engineering application

Task B is concerned with an operation which looks for a component at a particular point in time and – if one is present – diverts it to another transportation belt. The components arrive at 9 mSec intervals (nominally) and take 1 mSec. to pass by the sensor. There are guaranteed time bounds of ± 0.75 mSec. on arrival times, thus giving a measurement 'window' of 1.5 mSec. (fig.4(b)). We design the software having a total window of 2 mSec, where the mid-point coincides with the nominally correct arrival time. Thus the nominal arrival times – referring to fig.3 – are 4, 13, 22, 31, 40, 49, etc. With this in mind, take a look at the operation of task B on its fifth activation, assuming that the component actually arrives at (39+0.25). It will have passed out of detection range 1 mSec. later, at 40.25 mSecs. Unfortunately this coincides with the period of time in which task B is suspended. The result is that the component goes undetected.

These simple examples highlight two factors which are significant in real-time embedded systems:

- the time at which results are delivered and
- the criticality of operations.

We can deduce that, in this case, assumptions A1, A2 and A4 hold; clearly A3 and A5 don't. Moreover, from a practical point of view, we would have to use the worst-case (longest) task times when forming the task schedule.

There are various ways to deal with the problems raised by the failure of assumptions A3 and A5. But first one should check to see ***if*** any assumptions are invalidated. Criticality assessment requires that you have a good, detailed correct knowledge of system operation. It is important to decide early on if a task's deadline and period can be taken to be synonymous. For many applications there isn't a problem if there is variation in *actual* deadlines. Take the case of driving a non-latched alpha-numeric display panel. Provided a sensible update period is maintained, changes in updating times can be tolerated – they won't produce display flicker. However in situations where timing **is** critical, the task schedule will need very careful scrutiny.

I hope you are beginning to appreciate the extent of the difficulty in producing a reliable task schedule. Now extrapolate the discussion to a 10-task system, say. The effort involved to achieve guaranteed schedulability will clearly be considerable, even where assumptions A1, A2 and A4 are satisfied. And remember, so far we have assumed that our system consists of independent tasks only.

4. Interacting tasks.

In real systems tasks are very unlikely to be independent, for three particular reasons:

- Synchronization of operations.

- Communication of information.
- Access to shared resources.

All can lead to so-called *blocking* conditions – one task being delayed by another one (or worse still, being delayed by a number of tasks).

The first two items listed above are pretty clearcut; requirements will become obvious very early on in the design of the tasking software. However, dealing with them is not such an easy aspect, as illustrated in fig.5.

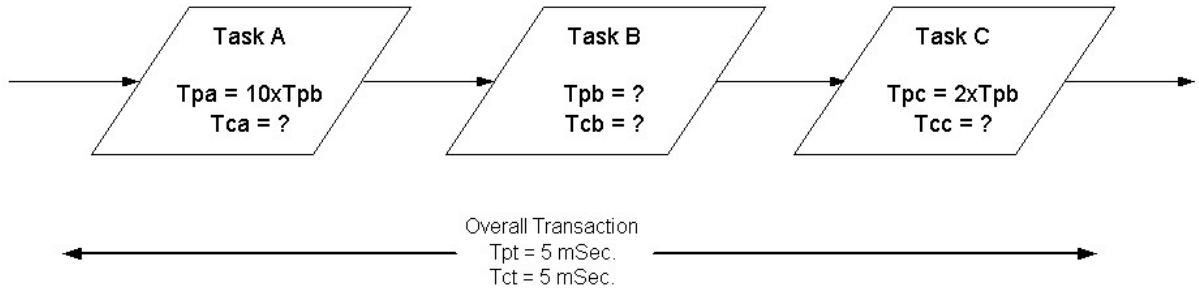

Fig. 5 Overall transaction requirements

This shows a three-task combination which performs some system function or *transaction* (the particular structure is representative of multi-rate control systems, where various tasks operate at different rates). Our primary concern is to ensure that the transaction as a whole meets its requirements; it isn't just a case of the individual parts being satisfactory. Here the transaction requirements are period (Tp_t) and deadline (Td_t), these coming from real-world performance requirements. With this starting point the designer is faced with somewhat of a juggling problem concerning the allocation of times to individual tasks. Difficulties are further compounded because tasks must execute in a particular order.

Unfortunately, where precise results are needed, mathematical predictive techniques are limited to simple cases. RMA in itself doesn't appear to help much. Paper and pencil generated solutions are likely to be tedious and time-consuming and – in large systems – not even feasible.

Now to turn to access control of shared resources (mutual exclusion), a most important topic. Designs must be carried out in a rigorous manner, otherwise serious problems can ensue (if you don't believe this, I recommend that you read Nancy Leveson's account of the Therac-25 fiasco [4]). Mathematical analysis of such interactions tends to be limited to somewhat simplistic situations (perhaps realistic structures are too difficult to analyse?). The basic idea is that we add the time for which a task is blocked to its computation time when calculating processor utilization. For example, suppose that Task A is found to be blocked for a time Tb, then its total time allocation becomes:

$$(Tc_a + Tb_a)$$

Thus its utilization of the processor becomes

$$Ua = (Tc_a + Tb_a) / Tp_a)$$

So far, so good. However, it does require that we get accurate figures for blocking times. Unfortunately, where resource sharing occurs, these aren't always easy to come by. This is particularly the case where unbounded priority inversion occurs (a low-priority task blocking one of higher priority for an indeterminate period [3]). Such behaviour makes it extremely difficult to predict blocking times; worse still it can produce serious performance problems (see 'What really happened on Mars?' [5]).

One way of dealing with this is by use of the so-called ceiling priority protocol [3]. Here each resource is given a priority equal to that of highest priority task (say Task H) which uses that resource. This is defined to be its *priority ceiling*. All other tasks having need of that resource must, by definition, be of lower priority. Should one of these (Task L) acquire the shared item, there is no change in its priority. However, if it suspends whilst holding the resource **and** Task H then tries to use the resource then:

- Task H is suspended.
- Task L has its priority raised to that of Task H and is set running.
- Task L finishes with the resource; it then has its priority reduced to normal.
- Task H preempts Task L and acquires the resource.

As a result a high-priority task has to wait at most for only *one* lower-priority task to finish using a protected resource. Blocking time calculations are very much simplified.

Rate monotonic theory has been extended to cover this situation. If the maximum time that such blocking can occur is Tb (i.e. when the low-priority task has locked the resource), then the modified RMS algorithm is:

$$\sum_{i=1}^{n} [(Tci/Tpi + Tbi/Tpi) \leq n(2^{1/n} - 1)]$$

A number of operating systems and language task models have incorporated the priority ceiling technique (e.g. the protected object of Ada 95). With it, however, comes an overhead as the system must dynamically:

- Keep track of all suspended tasks.
- Note and record the state of each shared resource.
- Change task priorities if and when required.

For hard, fast systems the time overhead may be significant. Unless these are included in the blocking calculations our mathematical predictions will be very suspect indeed.

5. Dealing with aperiodic tasks.

We now come to what is perhaps the most unsettling factor as far as task scheduling is concerned: the presence of aperiodic tasks. Aperiodic (asynchronous) tasks are part and parcel of the embedded world, being activated by the generation of hardware interrupts. By definition the arrival of these cannot be predicted; they can be described in statistical terms

only. So how then can we include them in a deterministic schedule? In the RMA world two techniques predominate:

- Transform aperiodic tasks into periodic ones (using polling techniques).
- Provide time to deal with aperiodic task activations within the timeframe of the schedule (the aperiodic server) [3].

Let us see how these work when applied to an example task set, table 1.

Task	Type	Response Time (milliseconds)	Computation Time (milliseconds)	Period (milliseconds)
A	Periodic	20	10	100
B	Periodic	18	15	120
C	Aperiodic	110 - Deadline	5	-
D	Aperiodic	5 - Deadline	2	-

Table 1. Example task set.

(a) Polling

Instead of using interrupts to trigger tasks we poll I/O devices (under software control) to establish their status. The really difficult question to answer is: how often? Regrettably there is no single answer; it all depends on the application.

For the moment assume that we can produce believable figures. Using these, modify table 1 to show all tasks as periodic ones, table 2.

Task	Type	Response Time (milliseconds)	Computation Time (milliseconds)	Period (milliseconds)
A	Periodic	20	10	100
B	Periodic	18	15	120
C	‘Periodic’	110	5	150
D	‘Periodic’	5	2	250

Table 2. Rate monotonic scheduling – Example task attributes.

This, it can be seen, has assumed that the period or ‘interval time’ of task C is 150 milliseconds. For task D 250 milliseconds has been chosen. Of course we still have to decide on the priorities of the aperiodic tasks. The simplest way is to use the interval time to calculate priorities, as shown in table 3, column 1.

1	**2**	**3**
For aperiodic tasks: • Period ↔ Interval Time • Priority ↔ Periodicity	For aperiodic tasks: • Period ↔ Response Time • Priority ↔ Periodicity	For aperiodic tasks: • Period ↔ Interval Time • Priority ↔ Deadline

Task→Period	Priority Order	Task→Period	Priority Order	Task→Period	Priority Order
A→100	1	A → 100	2	A →100	2
B→120	2	B →120	4	B →120	3
C→150	3	C →110 (Td = 110)	3	C →150 (Td =110)	4
D→250	4	D →5 (Td = 5)	1	D →250 (Td = 5)	1

Table 3. Rate monotonic scheduling – Handling aperiodic tasks.

If we set all tasks ready at time t_o, then task C will complete at $(t_o + 30)$ milliseconds – well within its deadline. Unfortunately task D won't finish until $(t_o + 32)$ milliseconds, much too late. This problem is not just restricted to aperiodic tasks; it is a consequence of treating period and deadline as being the same thing. We have already looked at this, but it's worth bringing it up again in the current context.

Improvements can be attained by using a different set of rules in defining the 'period' of an aperiodic task; use the task response time (or deadline – they're the same in this case) as the period (column 2, table 3). In this particular case the schedule will work, even though task D is activated every 5 milliseconds. Unfortunately such high-rate operation dramatically increases processor utilization; furthermore, in many cases it may generate infeasible schedules.

A technique developed to overcome this drawback is the *deadline monotonic* scheduling policy. With this, aperiodic tasks are treated as periodic tasks as described earlier. This defines their periodicity (column 3, table 3). However priorities are based on deadlines (for periodic tasks, remember, period and deadline are the same thing). As a result task D gets activated only every 250 milliseconds, but when it does it takes on highest priority. As a result it is guaranteed to run to completion without being pre-empted.

Polling, as a technique to handle random inputs, has a number of drawbacks:

- Excessive overheads – polling whether or not an event occurs.
- Staleness of acquired data – time between an event occurring and it being recognised by the software.
- Skew between input signals – simultaneous event signals may, as a result of the polling action, have significantly different time stamps.

Alernative methods such as the periodic server may provide a much better quality of service.

(b) The periodic server.

The periodic server attempts to alleviate these problems by not polling; instead it builds time into the schedule to handle random inputs *should they occur*. One technique allocates a fixed amount of processing time in a predefined time slot for the aperiodic tasks. Once this is used up no further aperiodic processing can be done until the beginning of the next time slot.

By incorporating the periodic server technique, schedules can be analysed mathematically for feasibility. Which is fine as long as you can:

- Defer the handling of inputs should the current time allocation has run out.
- Allow for the overheads involved.

5. Mode changes and related factors.

We've come a long way from the original concepts of rate monotonic priority assignment of Liu and Layland. The basic assumptions, as you've seen, don't hold up in the real world of embedded systems, especially fast, critical ones. Various extensions and modifications have been developed to deal with real-world requirements; these, to use a cliché, succeed to a lesser or greater extent. Now we'll look at something which, in large systems, introduces much complexity into task scheduling: system mode changes.

Take the case of a computer-based integrated missile defence system on a naval vessel. Most of the time it will be in search mode, switching into tracking only if a potential threat is identified. Following this is the target acquisition and lock-on phase, leading into launch and then guidance modes. Thus the functions (here being synonymous with transaction, fig.5) performed by the system are many and varied. However, those executed at any *particular* time depend on the mode of operation. Some will always be performed, some will apply to a number of modes, while others may be carried out only in specific modes.

A second factor is the importance of the function. Here they are categorized into vital, essential and desirable groupings. Vital functions are those which, if they fail, lead to physical damage, injury or loss of life. Essential ones are those which are needed to ensure correct functioning of the system. The third group, desirable, provides convenience facilities; failure is inconvenient but not mission-threatening.

A third major factor is the likelihood of specific functions being activated during operations. The groupings used here are definite, probable, possible and unlikely (a somewhat arbitrary choice, it must be added).

There are also cases where functions must *not* be carried out in particular modes (I know of a case where the leading-edge slats of an aircraft were deployed at Mach 0.9. As a result the wings were torn off and all five crew members killed). These serious error cases need to be explicitly identified so that defensive measures can be implemented in the software.

This information needs to be collated and recorded so that the overall multitasking performance requirements can be established. Begin at the system level design by defining all modes together with their relationships, as for example:

Mode Listing

Current Mode	Event	Response	Next Mode
Search Mode	Target Detected	Activate Tracking	Track Mode

Next, all functions, together with their attributes, need to be identified, e.g.:

Function Listing

Function	Type	Tp	Tc	Td	Importance	Likelihood
Radar Search	Periodic	50 mSec.	2 mSec.	-	Essential	Definite
Missile to manual	Aperiodic	-	1 mSec.	200 mSec	Essential	Possible

Following this, functions need to be allocated to modes, viz:

Targeting Mode Functions

Function	Type	Tp	Tc	Td	Importance	Likelihood
Designate Target	Aperiodic	-	2 mSec.	40 mSec	Essential	Definite
Target Tracking	Periodic	50 mSec.	4 mSec.	-	Essential	Definite

Using this information the processor workload during individual modes can be derived:

Tracking Mode – Processor Workload

Function Type → Function Importance ↓	Periodic	Aperiodic	Total
Vital	0.1	0.05	0.15
Essential	0.6	0.1	0.7
Desirable	0.05	-	0.05
Totals	0.75	0.15	0.9

As the processor tasking model is developed, it must be reconciled (both functionally and temporally) with this system-level information. This will involve the evaluation of all operational scenarios, especially from a time point of view. Only then can the task schedule effectiveness be deduced. A somewhat more onerous job than performing simple rate monotonic analysis, methinks.

6. Closing comments.

You might, at this point, feel this has been to some extent an RMA-bashing article. Not so. It's main target has been those who (deliberately or otherwise) mislead you into thinking that rate monotonic techniques will solve your problems without any BASE input (BASE: Brain Aided Software Engineering).

Rate monotonic scheduling, deadline monotonic scheduling, basic cyclic scheduling, etc.: all have firm foundations and provide a good basis for multitasking design. They bring rigour and coherence to the development process, and should be used as and when appropriate. But it can be seen that these form the foundations; a considerable amount of further work is needed to erect the building. A point often overlooked is that these techniques have an Achilles heel; *everything* depends on the correctness of the time attributes (i.e.

computation time, deadline, period). And note: nothing has so far been said about clock speeds, interrupt latencies, context switch times, and other overheads.

For anything in the least bit complex, tool support ***is essential*** to fully examine all aspects of system behaviour. The academic community recognized this many years ago, producing such tools as part of various research projects [6,7]. More recently commercial products have appeared. Some are provided by RTOS vendors as an integrated part of their development environments; others, such as PERTS [8], are vendor-independent.

One further point; the next time someone tells you that their RTOS is 'fully deterministic', you should have sufficient knowledge to challenge such statements (at present only one scheduler delivers fully deterministic behaviour: the simple safety-critical non-preemptive cyclic scheduler). And even that has to dispense with caching, pipelining, DMA operations and application-level interrupts.

7. References.

[1] Liu,C.L. and Layland,J.W., 'Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment', J.ACM, Vol.20, No.1, 1973, pp 46-61.

[2] Rekasius.Z.V., 'Digital Control with Computer Interruptions', Proceedings 1985 American Control Conf., Vol.3, ISSN 0958 4869, pp1618-1621.

[3] Sha,L. and Goodenough,J.B., 'Real-Time Scheduling Theory and Ada', IEEE Computer, Vol.23, No.4, April 1990, pp 53-63.

[4] Leveson,N.G., 'SAFEWARE – System Safety and Computers', Addison-Wesley Publishing Company, ISBN 9 780201 119725, 1995.

[5] Jones, M., 'What really happened on Mars', http://research.microsoft.com/~mbj/Mars_Pathfinder.html

[6] Cooling,J.E., Calinov,I. and Korousic,B., 'Animation prototyping of real-time multitasking systems', Proceeding IEE Serta'93, September 1993, pp125-130.

[7] Audsley,N.C., Burns,A., Richardson,M.F. and Wellings,A.J. 'STRESS: a simulator for hard real-time systems', Software - Practice and Experience, Vol.24 (6), June 1994, pp 543-564.

[8] PERTS, Tri-Pacific Consulting Group, Alameda, CA 94501.