



The Concise Handbook of Real-Time Systems

version 1.3

The Concise Handbook Of Real-Time Systems

TimeSys Corporation

Version 1.3

©2002 TimeSys Corporation
Pittsburgh, PA

www.timesys.com

What are Real-Time Systems?	7
Real-Time System Application Domains	8
A Taxonomy of Real-Time Software Architectures	9
Cyclic Executives	10
Software Architecture for Cyclic Executives	11
Event-Driven Systems	12
Pipelined Systems	13
Scheduling and Analyzing a Pipelined System	14
Client-Server Systems	15
State Machine Systems	16
Comparing Real-Time System Architectures	17
Deadlines and Timing Analysis	18
Where Do Timing Requirements Originate?	19
Why Do Timing Analysis?	20
Benefits of Schedulability Analysis and Simulation	21
Real-Time Scheduling Policies	22
Analyzing Periodic Tasks	23
Why is the RM Scheduling Bound Less Than 100%?	24
Dealing with Context Switch Overhead	25
Computing Completion Times Efficiently	26
Analyzing Task Synchronization	27
Priority Inversion	28
Unbounded Priority Inversion	29
Real-Time Synchronization Protocols	30
The Priority Inheritance Protocol	31
The Priority Ceiling Protocol	33
Example of the Priority Ceiling Protocol	34
Priority Ceiling Protocol Emulation	35
Aperiodic Tasks	36
Aperiodic Servers	37
Dealing with a Limited Number of Priority Levels	38
Example Scenario for Dealing with a Limited Number of Priority Levels	39
Dealing with Jitter	41

Other Capabilities of Real-Time System Analysis	42
Recommendations for Real-Time System Builders	43
Object Oriented Techniques in Real-Time Systems	44
CORBA	45
A Real-Time CORBA System	46
The Real-Time CORBA 1.0 Standard	47
Real-Time Java	48
TimeSys Solutions for Real-Time System Developers	49
TimeSys Linux™; A Real-Time OS with All the Benefits of Linux	50
TimeSys Linux Support for Reservations	51
TimeWiz®: An Integrated Design and Simulation Environment for Real-Time Systems	52
TimeStorm™: An Integrated Development Environment for TimeSys Linux	53
TimeTrace®: A Real-Time Profiling Environment	54
Glossary of Terms and Concepts	55
Some Key References on Resource Management for Real-Time Systems	61

What are Real-Time Systems?

Real-time computing systems are systems in which the correctness of a certain computation depends not just on how it is done but on *when* it's done. In order for tasks to get done at exactly the right time, real-time systems must allow you to predict and control when tasks occur.

Such systems play a critical role in an industrialized nation's technological infrastructure. Modern telecommunication systems, automated factories, defense systems, power plants, aircraft, airports, spacecraft, medical instrumentation, supervisory control and data acquisition systems, people movers, railroad switching, and other vital systems cannot operate without them.

A real-time system must demonstrate the following features:

- **Predictably fast response** to urgent events.
- **High degree of schedulability:** The timing requirements of the system must be satisfied at high degrees of resource usage.
- **Stability under transient overload:** When the system is overloaded by events and it is impossible to meet all the deadlines, the deadlines of selected critical tasks must still be guaranteed.

The key criteria for real-time systems differ from those for non-real-time systems. The following chart shows what behavior each type of system emphasizes in several important arenas.

	Non-Real-Time Systems	Real-Time Systems
Capacity	High throughput	Schedulability: the ability of system tasks to meet all deadlines.
Responsiveness	Fast average response	Ensured worst-case latency: latency is the worst-case response time to events.
Overload	Fairness	Stability: under overload conditions, the system can meet its important deadlines even if other deadlines cannot be met.

Real-Time System Application Domains

Potential uses for real-time systems include but are not limited to:

- Telecommunication systems
- Automotive control
- Multimedia servers and workstations
- Signal processing systems
- Radar systems
- Consumer electronics
- Process control
- Automated manufacturing systems
- Supervisory control and data acquisition (SCADA) systems
- Electrical utilities
- Semiconductor fabrication systems
- Defense systems
- Avionics
- Air traffic control
- Autonomous navigation systems
- Vehicle control systems
- Transportation and traffic control systems
- Satellite systems
- Nuclear power control systems

A Taxonomy of Real-Time Software Architectures

Virtually all real-time applications use elements from at least one of five architectural patterns:

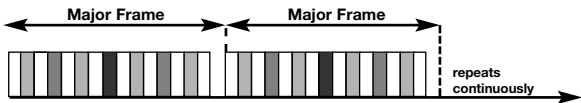
- Cyclic executives (also called “timelines” or frame-based systems) (page 10)
- Event-driven systems with both periodic and aperiodic activities (page 12)
- Pipelined systems (page 13)
- Client-server systems (page 15)
- State machine systems (page 16)

Cyclic Executives

A cyclic executive consists of continuously repeated task sequences, known as major frames. Each major frame consists of a number of small slices of time, known as minor frames; tasks are scheduled into specific minor frames.

- A timeline uses a timer to trigger a task every minor cycle (or frame).
- A non-repeating set of minor cycles makes up a major cycle.
- The operations are implemented as procedures, and are placed in a pre-defined list covering every minor cycle.
- When a minor cycle begins, the timer task calls each procedure in the list.
- Concurrency is not used; long operations must be manually broken to fit frames.

Below is a sample cyclic executive; it consists of minor frames and major frames. Major frames repeat continuously. Within a minor frame, one or more functions execute. Suppose that a minor frame is 10 ms long. Consider 4 functions that must execute at a rate of 50 Hz, 25 Hz, 12.5 Hz, and 6.25 Hz respectively (corresponding to a period of 20 ms, 40 ms, 80 ms, and 160 ms respectively). A cyclic executive can execute them as follows. Note that one minor frame lies idle in the major frame and can lend itself to future expansion.



Minor Frames

□ Function 1 (once every 2 minor frames)

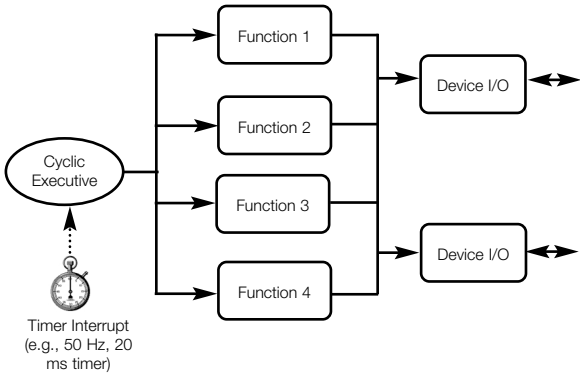
▒ Function 2 (once every 4 minor frames)

■ Function 3 (once every 8 minor frames)

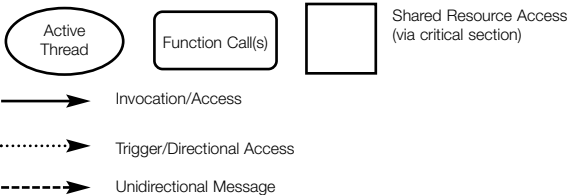
■ Function 4 (once every 16 minor frames)

Software Architecture for Cyclic Executives

Please refer to the above key with the software architectures presented in subsequent sections as well.



Key:



Event-Driven Systems

An event-driven design uses real-time I/O completion or timer events to trigger schedulable tasks. Many real-time Linux systems follow this model.

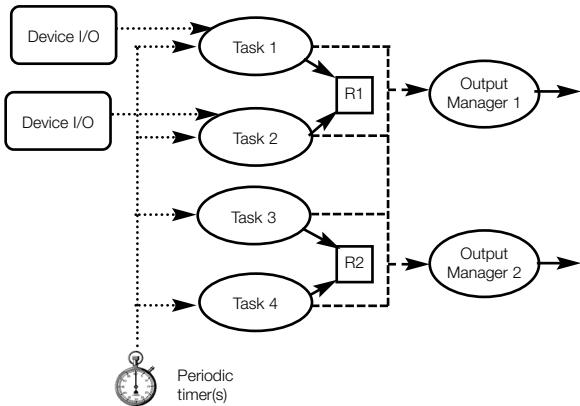
Tasks can be prioritized in the following ways:

- Priorities should be determined by time constraints (e.g., rate-monotonic or deadline-monotonic priority assignment policies).
- Task priority can also depend on semantic importance (but this approach will cause schedulability problems).

The resulting concurrency requires synchronization (e.g., mutex, semaphores, etc.).

- For predictable response, synchronization mechanisms must avoid (i.e. remain free of) unbounded priority inversion.
- To preserve predictable response, aperiodic events must preserve utilization bounds.

All of the rate-monotonic analysis techniques discussed in this handbook apply to event-driven systems without any modifications.



Pipelined Systems

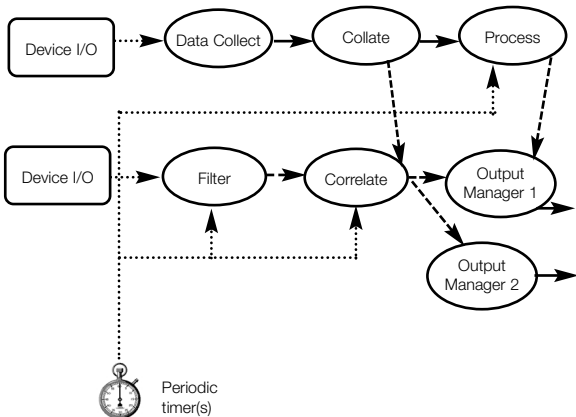
Pipelined systems use inter-task messages (preferably prioritized) in addition to I/O completion and timers to trigger tasks.

Control flow for an event proceeds throughout the system from source to destinations.

Thus, these systems can be described as a set of pipelines of task invocations.

Task priorities play only a minor role:

- If the pipeline is unidirectional, setting increasing task priorities will minimize message queue buildup.
- If the pipeline is bi-directional, it is generally best to set priorities so that they are equal along the pipeline.



Scheduling and Analyzing a Pipelined System

The complexity of pipelined systems makes them relatively difficult to analyze. Engineers must modify rate-monotonic analysis (RMA) techniques to account for the message-driven nature of pipelined systems as well as for precedence constraints.

Model each task as if it were a number of separate tasks, one for every message type that it handles. Since a thread cannot handle one message until it has finished taking appropriate actions on the previous one, these threads are non-preemptible. Although these non-preemptible threads may not really be synchronized, they should be treated as synchronized for analysis purposes. Programmers should use a FIFO synchronization protocol to calculate blocking terms for each separate thread.

The real-time engineer's next challenge is to take into account the sequence in which pipelined threads must proceed. To find out whether a series of threads on a pipelined system can meet its timing constraints, model each task as if it shared a logical resource with all others in the pipe, with FIFO synchronization. If it is then schedulable, at least one order will exist in which a sequence of tasks can meet their deadlines. Therefore, those tasks can always make it through in any order, including the "correct" one. The wise engineer will keep pipelined systems relatively simple to ensure that they meet deadlines in all cases.

Note: In some circumstances, complex pipelines whose precedence graphs include cycles may not lend themselves to analysis at all. In this case, the only option is a simulation.

Client-Server Systems

Client-server systems use inter-task messages in addition to I/O completion and timers to trigger tasks.

Sending tasks, or clients, block until they receive a response from receiving tasks, or servers.

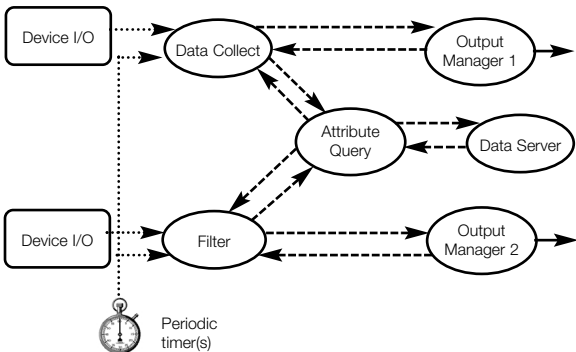
Control for an event remains at a single system node while data flow is distributed

Thus, error processing, checkpointing, and debugging are significantly easier for client-server systems than for pipelined systems.

As with pipelined architectures, task priorities play only a minor role.

- Ideally, server tasks inherit priorities from clients. This is often impractical, so priorities for different tasks are frequently set the same, using prioritized messages to avoid bottlenecks.

For analysis purposes, client-server systems are similar to pipelined systems.

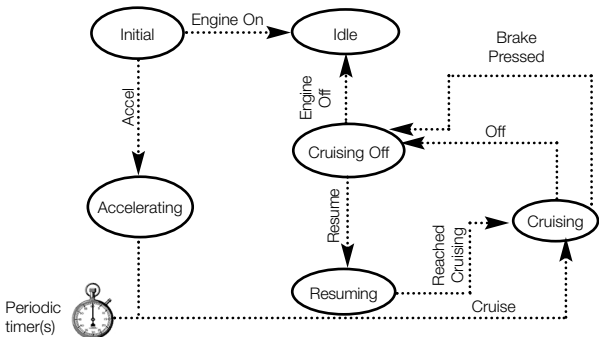


State Machine Systems

In a state machine architecture, the system is broken down into a set of concurrent extended finite state machines. Each such finite state machine is typically used to model the behavior of a reactive or active object. In a state machine, the object resides at any time in one of a finite number of states, waiting for an event. The arrival of an event triggers a transition, which may involve a change of state and execution of some actions associated with the transition. While an extended state machine can be used to model arbitrary behaviors, it is particularly suited to modeling many discrete state-dependent behaviors.

State machines generally follow "run-to-completion" semantics, in which the machine cannot accept an event for processing until it has finished with the previous event. The implementation of a system of concurrent state machines often requires mapping them to a set of concurrently executing tasks. To ensure run-to-completion semantics, a state machine is typically controlled entirely within a single thread that executes an event-loop of receiving (and processing) events.

The timing analysis of a state machine architecture system design depends on the task priorities, the mapping of the state machines to tasks, and the scheduling of events within the event-handling loops of the tasks. One way to make such a system analyzable is to ensure that each task handles either a single timing constraint or a set of similar constraints — this allows task priorities to be assigned based on their timing constraints. Another, albeit more complex, way to make the system analyzable is to view tasks as schedulable resources, using priorities for events and dynamically changing task priorities based on pending events.



Comparing Real-Time Architectures

Each of the five real-time architectures has its own set of benefits and drawbacks. The chart below compares and contrasts the architectural patterns.

	Benefits	Drawbacks
Cyclic Executive	Simple Deterministic Repeatable Easy to understand Most common approach Best for safety-critical systems	Fragile; if you add or change any procedures, the system is likely to break. Very complex to maintain Only good for small, simple systems which don't need dynamic capabilities
Event-Driven	Priority-driven (well-suited to RMA) Relatively simple Good for systems that are statically analyzable (not dynamically changing load)	Not as capable of handling distributed environments as some other designs
Pipelined	Readily used in distributed environments (fully message-based)	Complex to analyze Less predictable than some other designs
Client-Server	Works well with real-time CORBA and other object-oriented standards Simplifies debugging because of bi-directional feedback Good for object-oriented, distributed paradigms	Complex to analyze Uses substantial resources because of extra message traffic
State Machine	Works well with real-time CORBA and other object-oriented standards	Complex to analyze

Deadlines and Timing Analysis

Understanding the timing requirements of an application is important in any application, but it becomes even more crucial when designing real-time systems. One of the first steps to understanding your system's timing requirements is to determine which requirements cannot be missed and which can.

A hard deadline is a deadline that absolutely must be met for the system to function successfully. Failure to meet a hard real-time deadline could lead to loss of resources or even of life.

All other deadlines fall into the category of soft deadlines. If the system misses one of these deadlines, it does not necessarily fail. Most real-time systems contain many soft deadlines and a few hard deadlines.

In order to make the most of scarce processor resources, system designers must determine which deadlines are hard, and schedule processor time so that, no matter what happens with soft deadlines, hard deadlines will always be met.

Where Do Timing Requirements Originate?

Timing constraints originate from two types of sources: explicit and implicit. With top-level, or explicit, requirements, the precise constraints the system needs to meet flow organically from its design. Derived, or implicit, requirements, on the other hand, offer more flexibility. With derived requirements, all the system needs to do is to demonstrate a certain characteristic, and the system designer has to determine how quickly events must take place in order to give the system this characteristic. Generally speaking, explicit requirements tend to correlate with hard deadlines.

Some examples of explicit requirements include:

- Assemble two units every second in a manufacturing plant.
- Satisfy end-to-end display update timing constraint of 2 seconds in an air traffic control system.

Some examples of implicit requirements include:

- **Precision:** e.g., track aircraft position to within 10 meters.
- **Dependability:** e.g., recover from message loss within 500 ms.
- **User-interface requirements,** e.g.,
 - Respond to key presses within 200 milliseconds.
 - Maintain a 30-frames-per-second video frame rate.

In many real-time systems, most requirements are implicit rather than explicit. In other words, a typical system may have only a few requirements that are set in stone.

Why Do Timing Analysis?

Timing analysis provides a framework for scheduling events so the mandatory hardware resources are always available *when they are needed*, in order to make sure that critical tasks meet their timing requirements. Timing analysis provides a number of substantial benefits for your system.

Timing Risk Elimination: Using timing analysis, risks of timing conflicts can be eliminated from your real-time system – while the logic cannot be guaranteed within your system, timing analysis can guarantee that your system timing constraints will be satisfied.

Dramatic Reduction in Integration and Testing Time: Your savings on integration and testing time alone will more than compensate you for applying analytical techniques. These benefits stem from the application of Rate-Monotonic Analysis (RMA), a scientifically proven framework for building analyzable and predictable real-time systems.

Robust Systems Interaction: Your real-time systems are complex and may comprise two or more processors with interconnecting backplane buses and/or network links. These processors work asynchronously with each other. What you want is the assurance that, given all possible working conditions, your system will do the right thing at the right time. The use of a scientifically proven methodology offers this guarantee.

Enhanced System Reliability: The RMA framework and the analyses and simulation that you can perform enhance your system reliability. Since sub-systems and components will behave as expected, there need be no confusion as to whether an inordinately delayed message will cause the failure of a component.

A Priori Testing: You can design and test your system even before it is built, thereby significantly reducing the cost and risk of using the wrong type or number of components.

RMA is much easier with an analysis tool such as TimeWiz.

Benefits of Timing Analysis and Simulation

Timing analysis and simulation can bring your system the following benefits:

- Capture system requirements to use in competitive proposals, which you can then pass as requirements document to your design and development team.
- Visually represent both hardware and software configurations.
- Guarantee predictable behavior.
- Clearly understand worst-case timing behavior.
- Demonstrate competitive average-case timing behavior.
- Perform what-if analyses.
- Avoid costly mistakes.
- Identify better or cheaper configurations with what-if-analyses and automatic binding of software components to hardware components.
- Ensure that sufficient resources remain for future system expansion.
- Obtain certification by capturing and analyzing your system for the benefit of regulatory and certification bodies.

Real-Time Scheduling Policies

Real-time engineers use a number of different schemes for scheduling events. Some popular real-time scheduling policies include:

Fixed Priority Preemptive Scheduling: Every task has a fixed priority that does not change unless the application specifically changes it. A higher-priority task preempts a lower-priority task. Most real-time operating systems support this scheme.

Dynamic-Priority Preemptive Scheduling: The priority of a task can change from instance to instance or within the execution of an instance, in order to meet a specific response time objective. A higher-priority task preempts a lower-priority task. Very few commercial real-time operating systems support such policies.

Rate-Monotonic Scheduling: An optimal fixed-priority preemptive scheduling policy in which, the higher the frequency (inverse of the period) of a periodic task, the higher its priority. This policy assumes that the deadline of a periodic task is the same as its period. It can be implemented in any operating system supporting fixed-priority preemptive scheduling or generalized to aperiodic tasks.

Deadline-Monotonic Scheduling: A generalization of the rate-monotonic scheduling policy in which the deadline of a task is a fixed point in time relative to the beginning of the period. The shorter this (fixed) deadline, the higher the priority. When the deadline time equals the period, this policy is identical to the rate-monotonic scheduling policy.

Earliest-Deadline-First Scheduling: A dynamic-priority preemptive scheduling policy. The deadline of a task instance is the absolute point in time by which the instance must complete. The deadline is computed when the instance is created. The scheduler picks the task with the earliest deadline to run first. A task with an earlier deadline preempts a task with a later deadline. This policy minimizes the maximum lateness of any set of tasks relative to all other scheduling policies.

Least Slack Scheduling: A dynamic-priority non-preemptive scheduling policy. The slack of a task instance is its absolute deadline minus the remaining worst-case execution time for the task instance to complete. The scheduler picks the task with the shortest slack to run first. This policy maximizes the minimum lateness of any set of tasks.

Analyzing Periodic Tasks

1. Consider a set of n periodic tasks, each with a period T_i and a worst-case execution time C_i .
2. Assign a fixed higher priority to a task with a shorter period; i.e., higher rates get higher priorities (rate-monotonic priority assignment).
3. All of these tasks are guaranteed to complete before the end of their periods if:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \leq bound$$

where the *bound* is:

- 1.0 for harmonic task sets.
 - A task set is said to be harmonic if the periods of all its tasks are either integral multiples or sub-multiples of one another.
- 0.88 on the average for random C_i 's and T_i 's.
- $n(2^{1/n} - 1)$.
- 1.0 for $n=1$, $0.69 = \ln 2$ for large n .

The bound varies between 0.88 and 0.98 for most realistic, practical task sets.

$U_i=C_i/T_i$ is called the *utilization* of task i .

Benefits of rate monotonic analysis include simplicity, efficiency, wide support, and practicality.

Many activities in real-time, embedded, and multimedia systems are periodic, including:

- audio sampling in hardware
- audio sample processing
- video capture and processing
- feedback control (sensing and processing)
- navigation
- temperature and speed monitoring

Why is the Rate Monotonic Scheduling Bound Less Than 100%?

Consider two periodic tasks: $\tau_1 = \{C_1 = 41, T_1 = 100\}$ and $\tau_2 = \{C_2 = 59, T_2 = 141\}$. Let both tasks start together and let rate-monotonic scheduling be used. The first instance of task τ_1 arrives at time 0 and the second at time 100. The first instance of task τ_2 arrives at time 0 and the second at time 141. The first instance of task τ_2 must complete within time 100 and the first instance of τ_2 must complete within time 141.

A timeline tracing these tasks would be complete from time 0 to time 141. If C_1 or C_2 is increased by even a very tiny amount, the first instance of τ_2 will miss its deadline at time 141. The total utilization of this task set is $41/100 + 59/141 = 0.41 + 0.4184 = 0.8184$. In other words, for a two-task set, deadlines can be missed at about 82%. With more tasks, this number can drop to 69%, but no lower. But these thresholds represent pathological cases. For example, notice that the utilization of the two tasks is (almost) equal, $C_1 = T_2 - T_1$, and that $T_2/T_1 = 1.414 = \sqrt{2}$. Similarly, the 69% bound is obtained for a large number of tasks with $U_1 = U_2 = \dots = U_n$, $C_i = T_{i+1} - T_i$, and $T_{i+1}/T_i = 2^{1/n}$.

However, in practice, rate-monotonic scheduling can almost always yield at least 88% schedulable utilization. For harmonic task sets, the schedulable utilization is 100%. As a result, task sets with even a few harmonic periods tend to have very high schedulable utilization.

Dealing With Context Switch Overhead

It takes a finite amount of time for the operating system to switch from one running thread to a different running thread. This is referred to as “context switching overhead.”

The worst-case impact of context switching overhead can be completely accounted for by considering that there are, at most, two scheduling actions per task instance, with one context switch when the instance begins to execute and another when it completes. Thus, the utilization of each task now becomes:

$$U_i = C_i/T_i + (2 \cdot CS)/T_i$$

where:

CS = worst-case round-trip context switch time from one task to another.

One can now pose the question “How long should a context switch take?”

The objective of a real-time system builder must be to keep $2 \cdot CS$ a small fraction of T_i , the smallest period of all tasks in the system.

Computing Completion Times Efficiently

The following applies to periodic tasks that are scheduled using any fixed-priority preemptive scheduling policy.

Theorem: Consider a set of independent, periodic tasks. If each task meets its first deadline under the worst-case task phasing, all deadlines of all tasks will always be met.

The worst-case scenario occurs when all tasks arrive simultaneously.

Completion Time (CT) Test: Sort the set of periodic tasks in descending order such that $\text{priority}(\text{task } i) > \text{priority}(\text{task } i+1)$. Suppose that the worst-case computation time, period, and deadline of task i are represented by C_i , T_i , and D_i , with $D_i \leq T_i$.

Let W_i be the worst-case completion time of any instance of task i . W_i may be computed by the fixed-point formula:

$$W_i(0) = 0$$
$$W_i(n+1) = C_i + \sum \left\lceil \frac{W_i(n)}{T_j} \right\rceil C_j$$

Task i is schedulable if its completion time W_i is at or before its deadline D_i (i.e. $W_i \leq D_i$).

Analyzing Task Synchronization

Real-time tasks typically share resources and services for which they must be prepared to wait if they are not available immediately. These resources and services may include:

- Logical resources such as buffers and data.
- Physical resources such as printers and devices.
- Services such as window managers, naming and directory services, transaction services, filesystem services, etc.

Tasks are said to be in a critical section while they are holding a shared resource. This can cause unbounded priority inversion.

Solution: Use any of the priority inheritance protocols:

A priority inheritance protocol bounds and minimizes priority inversion.

$$\frac{C_1+B_1}{T_1} + \frac{C_2+B_2}{T_2} + \dots + \frac{C_n+B_n}{T_n} \leq bound$$

where:

B_i = maximum priority inversion encountered by any instance of task i .

$B_n=0$.

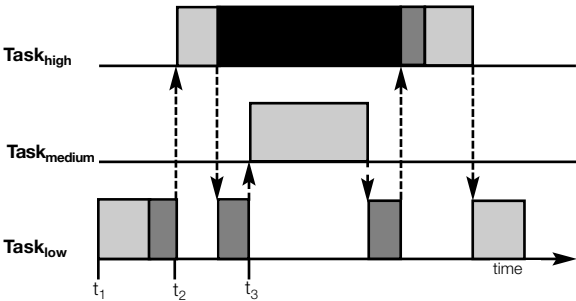
Priority Inversion

Priority inversion is said to occur when a task is forced to wait for a lower-priority task to execute.

Consider three tasks $\text{Task}_{\text{high}}$, $\text{Task}_{\text{medium}}$, and Task_{low} , listed in descending order of priorities. $\text{Task}_{\text{high}}$ and Task_{low} share a logical resource protected by a critical section.

Let $\text{Task}_{\text{high}}$, $\text{Task}_{\text{medium}}$, and Task_{low} arrive at times t_1 , t_2 , and t_3 respectively.

The graph below illustrates what happens to the execution patterns of each of the three tasks:



Key:

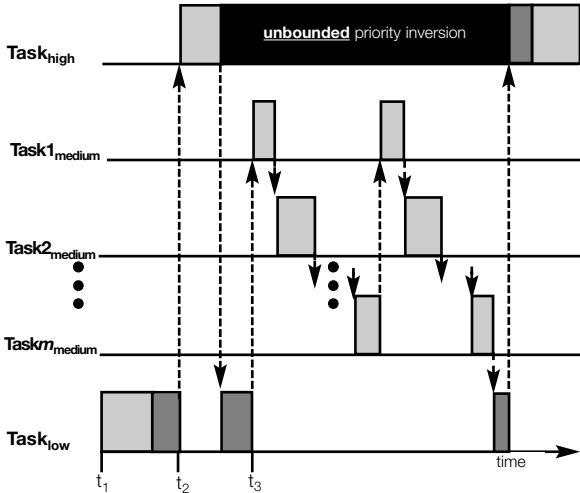
Light gray box: Normal execution

Dark gray box: Execution in critical section

Black box: Priority inversion

Unbounded Priority Inversion

Unbounded priority inversion can happen when there are multiple medium-priority tasks and these tasks are also periodic. As a result, each of these medium-priority tasks can preempt the lowest-priority task holding the critical section. In addition, the medium-priority tasks can recur due to their periodicity, preempting the lower-priority task.



Key:

Normal execution

Execution in critical section

Priority inversion

Real-Time Synchronization Protocols

Real-time synchronization protocols help bound and minimize priority inversion. Different varieties of real-time synchronization protocols include:

- Basic Priority Inheritance Protocol
- Priority Ceiling Protocol
- Critical Section Execution at Priority Ceiling (sometimes called Priority Ceiling Protocol Emulation or Highest Locker Protocol)
- Non-Preemption Protocol: disable preemption within a critical section

	Maximum Number of Critical Sections Waited For Per Period	Deadlock Prevention
Basic Priority Inheritance	Multiple ¹	No
Priority Ceiling Protocol	1	Yes
Critical Section Execution at Priority Ceiling	1	Yes ²
Non-Preemption Protocol	1 (but potentially very large)	Yes ²

Comparison of Synchronization Protocols

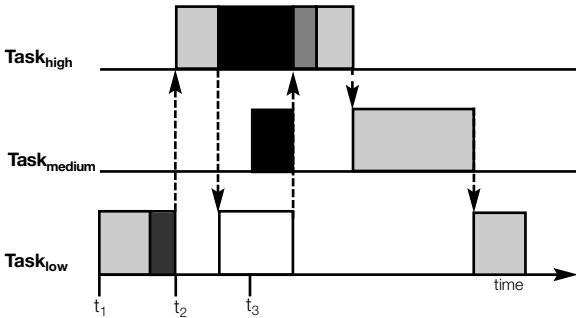
1 A maximum of $\min(m, n)$ critical sections, where n is the number of lower priority tasks and m is the number of distinct locks obtained by them. This assumes that deadlocks are avoided by using other schemes such as "total ordering" of the sequence of locks.

2 Tasks must not suspend within a critical section (e.g., for I/O operations).




The Priority Inheritance Protocol

A task runs at its original priority unless it is blocking one or more higher-priority tasks. In that case, it runs at the priority of the highest-priority task that it blocks.

Note that when a lower-priority task inherits the priority of a higher-priority task, intermediate-priority tasks encounter priority inversion. The higher-priority task also continues to encounter priority inversion in that it *must* still wait for the lower-priority task to exit its critical section. The following diagram provides an example of priority inheritance in action:



Key:

-  Normal execution
-  Execution in critical section
-  Priority inversion
-  Execution in critical section at higher priority

The Priority Inheritance Protocol (cont.)

The Mutual Deadlock Problem

Mutual deadlocks can occur with the basic priority inheritance protocol.

- Task 1 wants to lock L_1 and then L_2 in nested fashion.
- Task 2 tries to lock L_2 and then L_1 in nested fashion.

Task 2 locks L_2 first, before getting preempted by task 1, which then locks L_1 . Now, tasks 1 and 2 will be mutually deadlocked. This scenario can happen with any sequence of 2 or more tasks.

With the basic priority inheritance protocol, one must use a scheme such as “Total Ordering” while attempting to obtain locks. Such a scheme entails numbering each resource uniquely and accessing these resources using a convention such as: “Nested locks may be obtained only in ascending order of resource numbering.”

Not using nested locks is the easiest way to achieve total ordering.

The Priority Ceiling Protocol

Each shared resource has a priority ceiling that is defined as the priority of the highest-priority task that can ever access that shared resource.

The protocol is defined as follows.

- A task runs at its original (sometimes called its base) priority when it is outside a critical section.
- A task can lock a shared resource only if its priority is strictly higher than the priority ceilings of all shared resources currently locked by other tasks. Otherwise, the task must block, and the task which has locked the shared resource with the highest priority ceiling inherits the priority of task t .

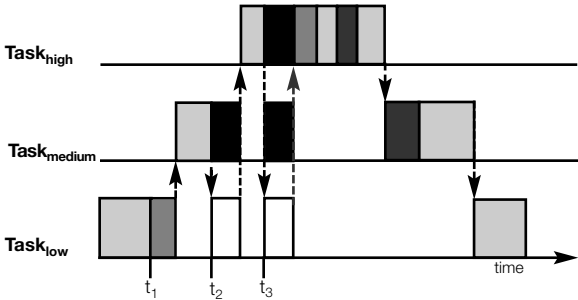
An interesting consequence of the above protocol is that a task may block trying to lock a shared resource, even though the resource is not locked.

The priority ceiling protocol has the interesting and very useful property that no task can be blocked for longer than the duration of the longest critical section of any lower-priority task.

Example of The Priority Ceiling Protocol

Consider tasks $\text{Task}_{\text{high}}$, $\text{Task}_{\text{medium}}$, and Task_{low} in descending order of priority. $\text{Task}_{\text{medium}}$ accesses Lock 2 and Task_{low} accesses Lock 1. $\text{Task}_{\text{high}}$ accesses Lock 1, releases it, then accesses Lock 2. Locks 1 and 2 both have the same priority ceiling, which equals the priority of Task 1.

At time t_1 , Task_{low} can successfully enter Critical Section 1 since there are no other tasks in a critical section. At time t_2 , $\text{Task}_{\text{medium}}$ tries to enter Critical Section 2. But since Task_{low} is already in a critical section locking a shared resource with a priority ceiling equal to the priority of $\text{Task}_{\text{high}}$, $\text{Task}_{\text{medium}}$ must block and Task_{low} starts running at the priority of $\text{Task}_{\text{medium}}$. Later, at time t_3 , when $\text{Task}_{\text{high}}$ tries to enter Critical Section 1, it has to block as well and Task_{low} starts executing at the higher priority of $\text{Task}_{\text{high}}$. When Task_{low} exits its critical section, it resumes its original lower priority. $\text{Task}_{\text{high}}$ can now enter both Critical Sections 1 and 2. Note that $\text{Task}_{\text{high}}$'s priority inversion is bounded by one critical section (that of $\text{Task}_{\text{medium}}$ or that of Task_{low} but not both).



Key:

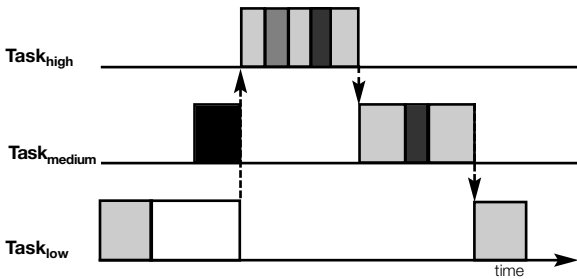
Normal execution	Execution in critical section 1	Priority inversion
Execution in critical section 1 at higher priority	Execution in critical section 2	

Priority Ceiling Protocol Emulation

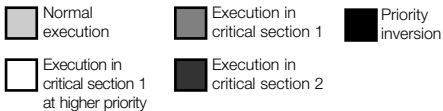
The priority ceiling of a shared resource is defined, as before, to be the priority of the highest-priority task that can ever access that resource.

A task executes at a priority equal to (or higher than) the priority ceiling of a shared resource as soon as it enters a critical section associated with that resource.

Applying the Priority Ceiling Protocol Emulation to the Priority Ceiling Protocol example results in the following sequence:



Key:



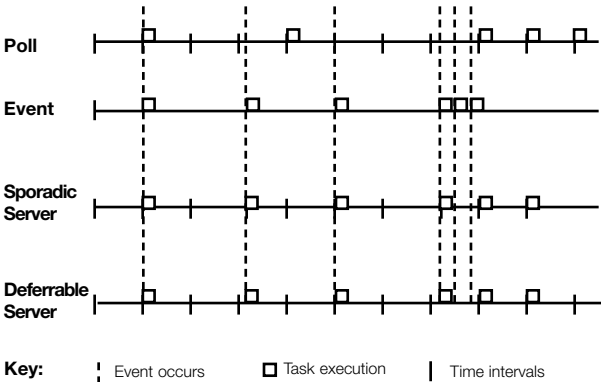
Aperiodic Tasks

Tasks in real-time and embedded systems are not always periodic. Tasks that may be aperiodic include operator requests, emergency message arrivals, threshold crossing notifications, keyboard presses, mouse movements, detection of incoming objects, and dynamic software compilation.

There are three basic ways of dealing with aperiodic tasks:

- **Polling:** The system periodically checks to see if an aperiodic event has occurred, then processes it if it has.
- **Event-interrupt driven:** When an aperiodic event occurs, the system stops what it is doing and processes it.
- **Aperiodic server:** The server deposits “tickets,” which are replenished at expiration of a replenishment period after use. When an aperiodic event occurs, it checks the server to see if any tickets are available. If there are, the system immediately processes the event, then schedules the creation of another ticket based on its ticket creation policies. An aperiodic server imposes predictability on aperiodic tasks, and therefore makes them suitable for analysis with techniques such as RMA. There are two types of aperiodic servers, deferrable and sporadic, each with different ticket creation policies.

The timeline below illustrates each of these policies in action.



Aperiodic Servers

There are two main types of aperiodic servers, the deferrable server and the sporadic server. Of these, the sporadic server has higher schedulable utilization and lends itself more easily to analysis. However, it can be more complex to implement.

In a **deferrable server**, tickets are replenished at regular intervals, completely independent of ticket usage. If an aperiodic task arrives, the system will process it immediately if it has enough tickets, and wait until the tickets are replenished if it does not.

While the deferrable server is simpler to implement, it deviates adversely from the Rate-Monotonic Strict Periodic Execution Model which leads to serious schedulability problems. A system can have at most one deferrable server, which must be at the highest priority in the system.

In a **sporadic server**, the replenishment time depends strictly on ticket usage time. When a ticket is used, the system sets a timer that replaces any used tickets when it goes off.

For example, imagine a system with a timer that goes off n milliseconds after a ticket is used. If this system uses one server ticket at time t , and two more server tickets at time t' , then the first ticket can be replenished n milliseconds after time t , and the other two can be replenished n milliseconds after time t' .

As illustrated by the above example, the sporadic server may have to track multiple ticket usages and their times. Its implementation therefore can be more complex. Simple but more conservative implementations are possible, however.

One benefit of a sporadic server is that a system can have multiple sporadic servers on a single node for different categories of aperiodic events with different base numbers of tickets and different timer intervals. This is because, in the worst case, the sporadic server behaves like a strict rate-monotonic periodic task.

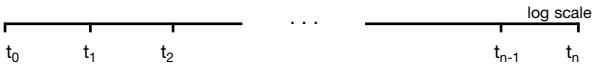
Dealing With A Limited Number of Priority Levels

The original definition of rate-monotonic scheduling algorithms assumed that each task with a different time constraint could be assigned a unique priority. For example, if there were 32 periodic tasks, each with a different time constraint, 32 distinct priority levels would be needed to use rate-monotonic or deadline-monotonic priority assignment.

However, a good approximation of rate-monotonic or deadline-monotonic priority assignments can be used when a sufficient number of priority levels is not available due to limitations from the underlying run-time system or operating system.

Priority Mapping Scheme

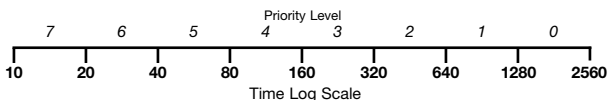
Determine the longest and shortest periods that your system needs to support. Draw the time-spectrum between these two periods on a logarithmic scale, and divide the spectrum equally into n segments, where n is the number of distinct priority levels available.



We, therefore, have $t_1/t_0 = t_2/t_1 = \dots = t_n/t_{n-1} = r$, where t_0 and t_n are the shortest and longest time constraints, respectively, to be supported. Suppose we use rate-monotonic scheduling and the period of a task is T_i . This task is assigned the priority j such that $t_{j-1} < T_i \leq t_j$. Use the relative time constraint instead of the task period T_i in the above context if deadline-monotonic scheduling is used.

Example Scenario for Dealing With A Limited Number of Priority Levels

Suppose that the underlying real-time OS (such as Windows NT) supports only 8 priority levels. Let the smallest period of a real-time task be 10 ms and the longest period be 2.5 seconds. The following priority-mapping scheme can then be used.



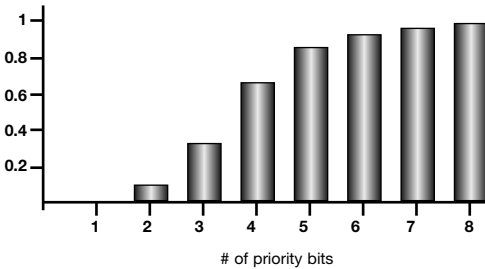
In this example, we assume above that priority level 7 is higher than priority level 6. Some real-time operating systems have the opposite convention, in which a lower value indicates a higher priority level.

This way of assigning priorities with a limited number of priority levels is not optimal, but generally produces a good mapping. For a specific task set, priority assignments with much better schedulability can frequently be obtained manually. This scheme is essentially an analyzable heuristic that works well in a broad range of cases.

Dealing with a Limited Number of Priority Levels (cont.)

Suppose that the shortest period to be supported for a system is 1 ms and the longest period is 100 seconds. We have: $t_r/t_0 = 100/10^{-3} = 10^5$. The loss in schedulability due to the above lumping of tasks with different periods (deadlines) into the same priority level is shown below as the number of priority bits available is varied; e.g., having 4 priority bits means that 16 priority levels are supported.

In general, having 256 distinct priority levels is practically equivalent to having a distinct priority level for each time constraint with a negligible loss of 0.0014 (about one tenth of one-percent). Having 5 priority bits (32 priority levels) is a good compromise for hardware support, where additional priority bits can be too expensive. In software, however, where the additional expenses are minimal, 8 bits (256 priority levels) are recommended.



Dealing with Jitter

Jitter is the size of the variation in the arrival or departure times of a periodic action. Jitter normally causes no problems as long as the actions all stay within the correct period, but certain systems might require that jitter be minimized as much as possible.

Real-time programmers commonly handle tasks with tight jitter requirements in one of two ways:

- If only one or two actions have tight jitter requirements, set those actions to be top priority. Note: This method only works with a very small number of actions.
- If jitter must be minimized for a larger number of tasks, split each task into two, one which computes the output but does not pass it on, and one which passes the output on. Set the second task's priority to be very high and its period to be the same as that of the first task. An action scheduled with this approach will always run one cycle behind schedule, but will have very tight jitter.

Most real-time systems use some combination of these two methods.

Other Capabilities of Real-Time System Analysis

- End-to-end timing analysis
- Network link and backplane bus analysis
- CANbus analysis
- Network switch analysis
- Jitter analysis
- Automatic binding of software to hardware components
- Computation of slack capacity in system for future growth
- RT-CORBA & Real-Time DCOM analysis
- Quality of Service (QoS) management
 - QoS-based Resource Allocation Model that can deal application QoS attributes such as frame size and frame rate, along with timeliness, cryptographic security, and dependability.

Please contact TimeSys Corporation (www.timesys.com) for additional information.

Recommendations for Real-Time System Builders

1. Adopt a proven methodology like RMA, which is:
 - Used by GPS satellites, submarines, shipboard control, air traffic control, medical instrumentation, multimedia cards, communications satellites, consumer electronics, etc.
 - Supported at least in part by commercial OS vendors (Windows 95/NT, AIX, Solaris, OS/2, HP/UX) and virtually all real-time OS vendors (TimeSys Linux, LynxOS, QNX, pSoS, VxWorks, etc.)
 - Supported by standards including Real-Time CORBA, POSIX, Ada 83 and Ada95, and Sun's Java Specification for Real-Time.
 - Adopted by NASA (Space Station) and by the European Space Agency.
2. Apply tools that support the methodology
 - Example: For RMA, use TimeWiz and TimeTrace from TimeSys Corporation (**www.timesys.com**).
 - TimeSys offers a suite of complementary products, including a Linux distribution with full RTOS capability, to serve your real-time system needs.
3. Utilize the experience and knowledge of real-time system experts on such subjects as:
 - How to use OS primitives correctly (e.g., with priority inheritance enabled on message queues and mutexes).
 - How to use middleware services.
 - How to structure applications with object-orientation.

Object-Oriented Techniques in Real-Time Systems

Problems with direct application of traditional object-oriented methodologies to real-time systems include:

- Existing OO methodologies generally push performance issues into the integration and test phase
 - Result: unbounded integration and test-phase, much higher risk and cost.
- Most response-time problems are hidden until late in integration.
- Inheritance and polymorphism should be limited where predictability is critical.

When applying object-oriented techniques to real-time systems, keep the following recommendations in mind:

- Identify concurrency early (perhaps a single thread per object).
- Choose threads early – at architecture definition time.
- Choose threads that do not encapsulate multiple timing constraints.
- Define scheduling techniques before finishing architecture.
- If timing constraints are critical, plan for analytical model (e.g., RMA) in addition to discrete event simulation.

There are important practical considerations for real-time OO systems:

- The usual OO systems underlying OS and infrastructure (e.g., CORBA ORBs, X-Windows) implementations often contain intrinsic priority inversions.

Inheritance and polymorphism are extremely valuable, but can make response time predictability difficult.

- Software architecture must always consider performance.
- For real-time systems, specific architectures have important real-time properties.
- Object-oriented design/programming is usable for real-time systems, but the architecture must consider performance at the highest level.

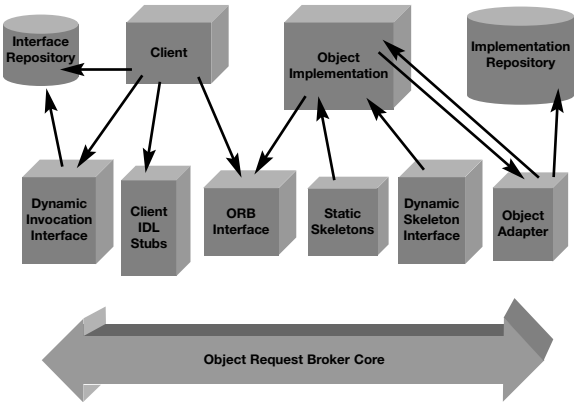
CORBA

CORBA stands for Common Object Request Broker Architecture, and has been standardized by the Object Management Group (OMG) using an open process. CORBA is an interoperable client/server middleware specification that specifies an extensive set of services that are used to produce “made-to-order” components.

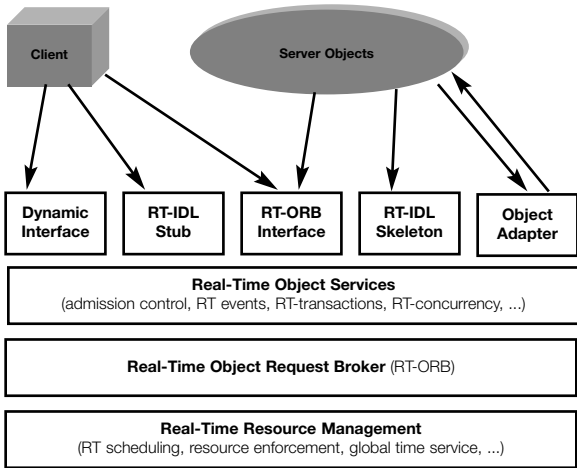
Some of the more than 20 standard services are Naming, Event, Transaction, Event, and Query. CORBA also specifies a neutral Interface Definition Language (IDL), by which all inter-object communication is managed.

A CORBA-based system contains four main components:

- Object Request Broker (ORB)
- CORBA Services
- CORBA Facilities
- Application Objects



A Real-Time CORBA System



The Real-Time CORBA 1.0 Standard

The Real-Time CORBA 1.0 specification supports fixed-priority scheduling. It directly supports the construction of pipelined and client-server-based distributed real-time systems. Pipelined real-time systems are supported by the use of asynchronous one-way messages between a “client” (a message sender) and a “server” (a message receiver).

Real-time operating systems differ in the number of priority levels they support and the convention that determines whether lower values represent higher priority levels or vice versa. As a result, RT-CORBA 1.0 provides a mapping scheme that allows applications to use a homogeneous, portable, and cross-compatible scheme to assign and manipulate priorities.

Secondly, RT-CORBA supports a flexible framework to assign the appropriate priority at which a server must process a client message. In a pipelined system, the “server” may use its own native priority, or inherit the priority of its client (or the highest priority of any waiting client). In a client-server-based system, a remote client request may be processed at a higher priority than any other normal application-processing activity on the server node. This permits the use of the “distributed priority ceiling protocol” and is necessary to minimize the large-duration priority inversion that can otherwise occur. Finally, RT-CORBA provides facilities for pooling and re-using threads and memory.

Real-Time Java

The Real-Time Specification for Java (RTSJ), completed in 2001 under Sun Microsystems' Java Community Process, meets the need for a truly platform-independent real-time programming language. The RTSJ adds to standard Java the following features:

- Real-time threads. These threads offer more carefully defined scheduling attributes than standard Java threads.
- Tools and mechanisms that let developers write code that does not need garbage collection.
- Asynchronous event handlers, and a mechanism that associates asynchronous events with happenings outside the JVM.
- Asynchronous transfer of control, which provides a carefully controlled way for one thread to throw an exception into another thread.
- Mechanisms that let the programmer control where objects will be allocated in memory and access memory at particular addresses.

TimeSys developed the reference implementation for the RTSJ, which is available at www.timesys.com. Further RTSJ information is available at www.rtej.org.

TimeSys Solutions for Real-Time System Developers

TimeSys Linux: TimeSys engineers have developed the first truly real-time version of Linux. TimeSys Linux offers a complete real-time system with the reliability and stability that are hallmarks of Linux. Available in multiple packages, TimeSys Linux can be provided alone, or in combination with TimeTrace™, described below, to capture your application's timing data.

Real-Time Java: TimeSys is in the process of developing a Java virtual machine based on the Real-Time Specification for Java. This product, the first to extend Java's capabilities into the real-time arena, allows real-time system designers to benefit from Java's platform independence and object orientation.

Timing Analysis and Simulation: TimeWiz® is a sophisticated system modeling, analysis, and simulation environment developed and marketed by TimeSys Corporation for real-time systems. The software runs on Windows NT/2000/98/XP.

Application Development: TimeStorm™ is a full-featured integrated development environment that lets you create TimeSys Linux applications on a remote platform. TimeWiz runs on Windows NT/2000/98/XP and allows you to download applications to a wide range of systems running TimeSys Linux.

Architectural Audit: This highly recommended service consists of a comprehensive technical evaluation of your system architecture, including hardware and software by TimeSys experts and Application Engineers. A detailed written report will be produced at the end of this evaluation clearly documenting the conclusions of the audit and recommendations (if any) to ensure that system timing constraints will be satisfied. Both system bottlenecks and resources of low risk will be identified, enabling the customer to focus on critical areas.

Timing Data Collection: TimeTrace™ provides the critical instrumentation needed to see inside your real-time system, collecting all the necessary timing data essential to the successful application of RMA and average-case simulation studies.

TimeSys Linux™: A Real-Time OS with All the Benefits of Linux

TimeSys Linux is the first Linux-based operating system to offer full real-time capabilities. The TimeSys Linux consists of a set of components that, when combined, provide a highly innovative approach to meeting timing constraints. These components include:

- **TimeSys Linux GPL**, a complete Linux kernel, downloadable for free from www.timesys.com, with unique TimeSys modifications to make it easily the lowest-latency Linux kernel anywhere. It is fully preemptible, contains a new priority-based scheduler with support for 2048 priority levels, and makes all interrupt handling and extended interrupt handling fully schedulable and prioritizable. This core is licensed under the GPL, and includes all source code.
- **TimeSys Linux/Real-Time**, a set of real-time modules that transforms TimeSys Linux into a fully-featured real-time OS. It changes all Linux mutexes so that they support priority inheritance and priority ceiling protocol emulation. In addition, it provides clock resolution at the highest level supported by the hardware itself and allows the kernel to declare and control high-resolution periodic tasks.
- **TimeSys Linux/CPU**, which allows threads or groups of threads to get guaranteed access to the CPU to support timely response, no matter what the load on the system.
- **TimeSys Linux/NET**, which allows guaranteed access to incoming and outgoing packets on a network interface.

The list of CPU types and boards supported by TimeSys Linux is constantly growing. See www.timesys.com for an updated list.

TimeSys Linux Support for Reservations

TimeSys Linux offers full support for reservations, allowing you more control in scheduling system tasks. With reservations, you can guarantee the availability of a certain amount of a certain resource, such as a CPU or a network interface.

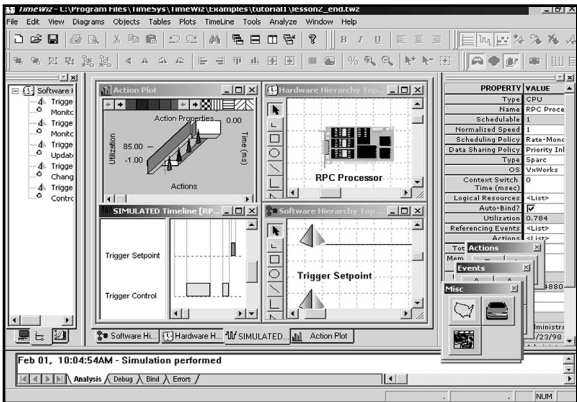
A reservation includes the following parameters:

- T for its period
- C for its uninterrupted computation time (CPU), or number of bytes (Net)
- D for its relative deadline

Each reservation can also be *hard* or *soft*, depending on whether tasks attached to the reservation are allowed to use the reserved resource when the reservation becomes depleted within each reserved period.

- **Hard reserves** deny tasks the use of reserved resources after C is completed for each period.
- **Soft reserves** grant usage of reserved resources to attached tasks after completion of C in a period, but these tasks must compete with all other tasks in the system.

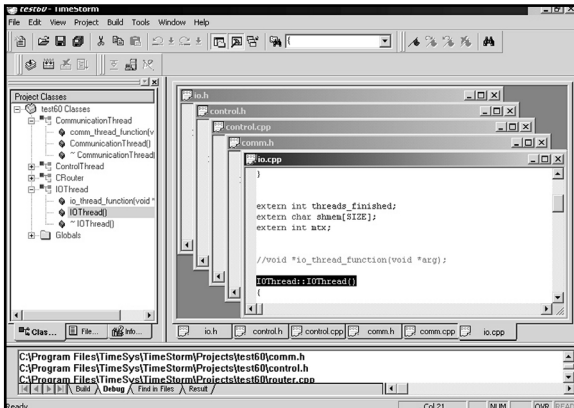
TimeWiz®: An Integrated Design and Simulation Environment for Real-Time Systems



TimeWiz® is a TimeSys Corporation product specifically designed for the construction of simple or complex real-time systems with predictable timing behavior. It lets you:

- Represent your hardware and software configurations.
- Analyze the worst-case timing behavior of your system.
- Simulate its average-case timing behavior.
- Model processors and networks for end-to-end performance.
- Chart your system parameters and generate integrated system reports.

TimeStorm™: An Integrated Development Environment for TimeSys Linux

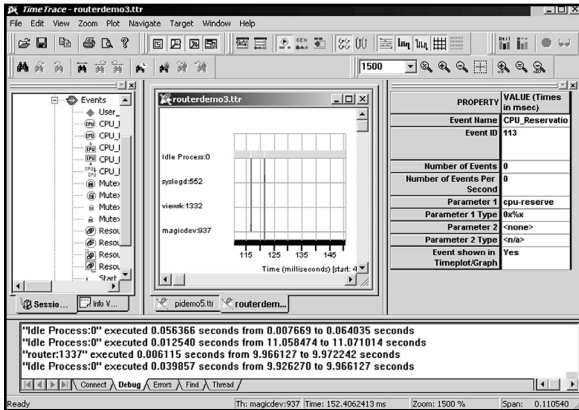


TimeStorm™ is a gcc-based integrated development environment (IDE) that allows you to create, compile, and debug TimeSys Linux applications on a Windows system using cross-compilers for your target board and architecture. With TimeStorm, you can:

- Write and edit code with a powerful editor that features search-and-replace functionality as well as language-specific syntax highlighting.
- Debug your applications with gdb.
- Navigate your project easily with control trees that let you view every file or every class, method, and variable in your project.
- Export applications to a variety of embedded systems running TimeSys Linux.

TimeStorm runs on Windows NT/2000/98/XP.

TimeTrace®: A Real-Time Profiling Environment



TimeTrace® is a productivity enhancement tool from TimeSys Corporation that lets you profile your real-time OS target in real-time. With TimeTrace, you can:

- Capture execution sequence on targets efficiently.
- Display target execution sequences visually to create a "software oscilloscope."
- Feed TimeTrace data into TimeWiz as execution time and period parameters for worst-case analysis and/or average-case simulation.

TimeTrace runs on Windows NT/2000/98/XP.

Glossary of Terms and Concepts

The following definitions apply to terms used throughout this manual, and are derived from the “Handbook of Real-Time Systems.” A clear understanding of these terms is very useful for any designer or developer of real-time systems.

Action	The smallest decomposition of a response; a segment of a response that cannot change system resource allocation. In TimeWiz, an action must be bound to a (physical) RESOURCE before it is analyzed. An action can also use zero, one, or more logical resources.
Aperiodic event	An event sequence whose arrival pattern is not periodic.
Average-case response time	The average response time of a response's jobs within a given interval. In TimeWiz, this is obtained through simulation. It is possible that there is a wide discrepancy between the average- and worst-case response times for a particular task. In many real-time systems (particularly for hard real-time tasks), the worst-case response time <i>must</i> be within a well-specified interval.
Blocking	The act of a lower-priority task delaying the execution of a higher-priority task; more commonly known as priority inversion. Such priority inversion takes more complex forms in distributed and shared memory implementations.
Blocking time	The delay effect (also called the “duration of priority inversion”) caused to events with higher-priority responses by events with lower-priority responses.
Bursty arrivals	An arrival pattern in which events may occur arbitrarily close to a previous event, but over an extended period of time the number of events is restricted by a specific event density; that is, there is a bound on the number of events per time interval. Bursty arrivals are

modeled in TimeWiz using their minimum interarrival time and their resource consumption in that interval.

Critical section

Period during which a real-time task is holding onto a shared resource.

Data-sharing policy

A policy specific to a (physical) resource that determines how logical resources bound to the (physical) resource can be accessed. Some schemes do not provide any protection against priority inversion, while others provide varying degrees of protection. TimeWiz supports multiple data-sharing policies including FIFO (no protection against priority inversion), PRIORITY INHERITANCE PROTOCOL, PRIORITY CEILING PROTOCOL, HIGHEST LOCKER PRIORITY PROTOCOL, and KERNELIZED MONITOR (non-preemptive execution) policies.

Deadline-monotonic scheduling algorithm

A fixed-priority algorithm in which the highest priority is assigned to the task with the earliest *relative* delay constraint (deadline) from each instance of its arrival. The priorities of the remaining tasks are assigned monotonically (or consistently) in order of their deadlines.

This algorithm and the earliest-deadline scheduling algorithm are not the same. In this algorithm, all instances of the same task have the same priority. In the earliest-deadline scheduling algorithm, each instance of the same task has a *different* priority, equal to the absolute deadline (time) by which it must be completed. The rate-monotonic scheduling algorithm and the deadline-monotonic algorithm are one and the same when the relative deadline requirement and periods are equal (which happens often).

Deterministic system

A system in which it is possible to determine exactly what is or will be executing on the processor during system execution. Deterministic systems result from the use of

	certain scheduling policies for groups of processes.
Dynamic-priority scheduling policy	An allocation policy that uses priorities to decide how to assign a resource. Priorities change from instance to instance of the same task (and can also vary during the lifetime of the same instance of a task). The earliest-deadline scheduling algorithm is an example of a dynamic-priority scheduling policy.
Earliest-deadline scheduling	A dynamic-priority assignment policy in which the highest priority is assigned to the task with the most imminent deadline.
Event	A change in state arising from a stimulus within the system or external to the system; or one spurred by the passage of time. An event is typically caused by an interrupt on an input port or a timer expiry. See also TRACE and TRIGGER.
Execution time	Amount of time that a response will consume on a CPU.
Fixed-priority scheduling policy	An allocation policy that uses priorities to decide how to assign a resource. The priority (normally) remains fixed from instance to instance of the same task. Rate-monotonic and deadline-monotonic scheduling policies are fixed-priority scheduling policies.
Hardware-priority scheduling policy	An allocation policy in which the priority of a request for the backplane is determined by a hardware register on each card that plugs into the backplane. Presumably, the hardware priority value reflects the importance of the device that is connected to the adapter.
Highest-locker priority	A DATA-SHARING POLICY in which an action using a logical resource is executed at the highest priority of all actions that use the logical resource (i.e. at the PRIORITY CEILING of the resource). This protocol provides a good level of control over priority inversion.

Input jitter	The deviation in the size of the interval between the arrival times of a periodic action.
Kernelized monitor	A DATA-SHARING POLICY in which an action using a logical resource is executed in non-preemptive fashion (i.e. at kernel priority). This protocol provides a good level of control over priority inversion except when one or more actions using a logical resource has a long execution time (relative to the timing constraints of other higher-priority tasks).
Logical resource	A system entity that is normally shared across multiple tasks. A logical resource must be bound to a physical resource like a processor, and is modeled in TimeWiz as an action with a mutual exclusion requirement. Also, see DATA-SHARING POLICY.
Output jitter	The deviation in the size of the interval between the completion times of a periodic action.
Period	The interarrival interval for a periodic event sequence. Also, see INPUT JITTER.
Periodic event	An event sequence with constant interarrival intervals. Described in terms of the period (the interarrival interval) and a phase value.
Preemption	The act of a higher-priority process taking control of the processor from a lower-priority task.
Priority ceiling	This is associated with each logical resource and corresponds to the priority of the highest-priority action that uses the logical resource.
Priority ceiling protocol	A data-sharing policy in which an action using a logical resource can start only if its priority is higher than the PRIORITY CEILINGS of all logical resources locked by other responses. This protocol provides a good level of control over priority inversion.

Priority inheritance protocol	A DATA-SHARING POLICY in which an action using a logical resource executes at the highest of its own priority or the highest priority of any action waiting to use this resource. This protocol provides an acceptable level of control over priority inversion.
Priority inversion	This is said to occur when a higher-priority action is forced to wait for the execution of a lower-priority action. This is typically caused by the use of logical resources, which must be accessed mutually exclusively by different actions. Uncontrolled priority inversion can lead to timing constraints being violated at relatively low levels of RESOURCE UTILIZATION. Also see BLOCKING and BLOCKING TIME.
Rate-monotonic scheduling algorithm	Algorithm in which highest priority is assigned to the task with the highest rate (in other words, with the shortest period) and the priorities of the remaining tasks are assigned monotonically (or consistently) in order of their rates.
Rate-monotonic scheduling	A special case of fixed-priority scheduling that uses the rate of a periodic task as the basis for assigning priorities to periodic tasks. Tasks with higher rates are assigned higher priorities.
Real-time system	A system that controls an environment by receiving data, processing it, and taking action or returning results quickly enough to affect the functioning of the environment at that time. A system in which the definition of system correctness includes at least one requirement to respond to an event with a time limitation.
Resource	A <i>physical</i> entity such as a processor, a back-plane bus, a network link, or a network router which can be used by one or more actions. A resource may have a resource allocation policy (such as rate-monotonic scheduling) and a data-sharing policy.

Response	A time-ordered sequence of events arising from the same stimulus. In TimeWiz, an event can trigger one or more actions to be executed.
Responses	Multiple time-ordered sequences of events, each arising from a distinct event. Event sequences that result in responses on the same resource often cause resource contention that must be managed through a resource allocation policy.
Task	A schedulable unit of processing composed of one or more actions. Synonymous with <i>process</i> .
Tracer	A stimulus. Synonymous with a single instance of an <code>EVENT</code> within TimeWiz, and is used to represent an end-to-end data flow sequence spanning <i>multiple</i> physical resources. An end-to-end timing constraint is normally associated with a tracer event. TimeWiz computes both worst-case and average-case response times to a tracer using analysis and simulation respectively. Also see <code>TRIGGER</code> .
Trigger	<p>A stimulus with an arrival pattern. Mostly synonymous with the term “<code>EVENT</code>” within TimeWiz but is used to name an event whose response consists of a chain of actions executing on, at most, a <i>single</i> resource.</p> <p>In TimeWiz, a trigger is bound to a (physical) resource when one or more actions in its corresponding response are bound to a (physical) resource. Also see <code>TRACER</code>.</p>
Utilization	The ratio of a response's usage to its period, usually expressed as a percentage. For a CPU resource, this is execution time divided by period.
Worst-case response time	The maximum possible response time of a response's jobs (instances). Also see <code>OUTPUT JITTER</code> .

Some Key References on Resource Management for Real-Time Systems

Baker, T., "Stack-Based Scheduling of Realtime Processes," *Journal of Real-Time Systems*, Vol. 3, No. 1, March 1991, pp. 67-100.

Bollela, G., et al., *The Real-Time Specification for Java*, 2000.

Borger, M. W., and Rajkumar, R., "Implementing Priority Inheritance Algorithms in an Ada Runtime System," *Technical Report*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, February, 1989.

Burns, A., "Scheduling Hard Real-Time Systems: A Review," *Software Engineering Journal*, May 1991, pp. 116-128.

Chen, M., and Lin, K.J., "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," *Journal of Real-Time Systems*, Vol. 2, No. 4, November 1990, pp. 325-346.

Gafford, J. D., "Rate Monotonic Scheduling," *IEEE Micro*, June 1990.

Gagliardi, M., Rajkumar, R., and Sha, L., "Designing for Evolvability: Building Blocks for Evolvable Real-Time Systems," *Proceedings of the IEEE Real-time Technology and Applications Symposium*, June 1996.

Harbour, M. G., Klein M. H., and Lehoczky, J. P., "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority," *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

IEEE Standard P1003.4 (Real-time extensions to POSIX), IEEE, 345 East 47th St., New York, NY 10017, 1991.

Jeffay K., "Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems," *IEEE Real-Time Systems Symposium*, December 1992, pp. 89-99.

Joseph, M., and Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal (British Computing Society)*, Vol. 29, No. 5, October 1986, pp. 390-395.

Lee, C., Lehoczky, J., Rajkumar, R., and Siewiorek, D., "On Quality of Service Optimization with Discrete QoS Options," *Proceedings of the*

IEEE Real-time Technology and Applications Symposium, June 1999.
Lehoczky, J. P., Sha, L., and Strosnider, J., "Enhancing Aperiodic Responsiveness in A Hard Real-Time Environment," *IEEE Real-Time System Symposium*, 1987.

Lehoczky, J.P., Sha, L., and Ding, Y., "The Rate-Monotonic Scheduling Algorithm — Exact Characterization and Average Case Behavior," *Proceedings of IEEE Real-Time System Symposium*, 1989.

Lehoczky, J. P., "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines," *IEEE Real-Time Systems Symposium*, December 1990.

Leung, J., and Whitehead, J., "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation* (2), 1982.

Liu, C. L., and Layland J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM*, Vol. 20 (1), 1973, pp. 46-61.

Mercer, C.W., and Rajkumar, R., "An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management," *Proceedings of the Real-Time Technology and Applications Symposium*, May 1995, pp. 134-139.

Miyoshi, A., and Rajkumar, R., "Protecting Resources with Resource Control Lists," *Proceedings of 7th IEEE Real-Time Technology and Applications Symposium, Taipei, Taiwan*, May 2001.

Molano, A., Juvva, K., and Rajkumar, R., "Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach," *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

de Niz, D. Abeni, L., Saowanee, S., and Rajkumar, R., "Resource Sharing in Reservation-Based Systems," *Work-in-Progress, 7th IEEE Real-Time Technology and Applications Symposium, Taipei, Taiwan*, May 2001.

Oikawa, S., and Rajkumar, R., "Linux/RK: A Portable Resource Kernel in Linux," *IEEE Real-Time Systems Symposium Work-In-Progress*, Madrid, December 1998.

Rajkumar, R., "Real-Time Synchronization Protocols for Shared Memory Multiprocessors," *The Tenth International Conference on*

Distributed Computing Systems, 1990.

Rajkumar, R., "Synchronization in Real-Time Systems: A Priority Inheritance Approach," Kluwer Academic Publishers, 1991, ISBN 0-7923-9211-6.

Rajkumar, R., and Gagliardi, M., "High Availability in The Real-Time Publisher/Subscriber Inter-Process Communication Model," *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

Rajkumar, R., Juvva, K., Molano, A., and Oikawa, S., "Resource Kernels: A Resource-Centric Approach to Real-Time Systems," *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D., "A Resource Allocation Model for QoS Management," *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

Rajkumar, R., Sha, L., and Lehoczky, J.P., "An Experimental Investigation of Synchronization Protocols," *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, May 1988.

Rajkumar, R., Sha, L., and Lehoczky, J. P., "Real-Time Synchronization Protocols for Multiprocessors," *Proceedings of the IEEE Real-Time Systems Symposium*, Huntsville, AL, December 1988, pp. 259-269.

Sha, L., Lehoczky, J. P., and Rajkumar, R., "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *IEEE Real-Time Systems Symposium*, 1986.

Sha, L., and Goodenough, J. B., "Real-Time Scheduling Theory and Ada," *IEEE Computer*, April, 1990.

Sha, L., Rajkumar, R., and Sathaye, S., "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems," *Proceedings of the IEEE*, January 1994.

Sha, L., Rajkumar, R., and Lehoczky, J. P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions On Computers*, September 1990.

Sprunt, H. M. B., Sha, L., and Lehoczky, J. P., "Aperiodic Task Scheduling for Hard Real-Time Systems," *The Journal of Real-Time Systems*, No. 1, 1989, pp. 27-60.

Corporate Headquarters

925 Liberty Avenue
6th Floor
Pittsburgh, PA 15222

888.432.TIME
412.232.3250
Fax: 412.232.0655

www.timesys.com

© 2002 TimeSys Corporation.
All rights reserved.

TimeTrace® and TimeWiz® are registered trademarks of TimeSys Corporation.

TimeSys Linux™, TimeSys Linux/Real Time™, TimeSys Linux/CPU™, TimeSys Linux/NET™, TimeStorm™, and TimeSys™ are trademarks of TimeSys Corporation.

Linux is a trademark of Linus Torvalds.

All other trademarks, registered trademarks, and product names are the property of their respective owners.