The Concise Handbook Of Real-Time Systems

TimeSys Corporation
Real-Time… Real Solutions™

Version 1.1

©2000 TimeSys Corporation
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*
Client-Server Systems. *14*

Why Do Timing Analysis? 15

Where Do Timing Requirements Originate? 16

Why Use Schedulability Analysis or Simulation? 17
Real-Time Scheduling Policies *18*

Analyzing Periodic Tasks 19
Why is the RM Scheduling Bound Less Than 100%. *20*
Dealing with Context Switch Overhead *21*
Computing Completion Times Effeciently *22*
Analyzing Task Synchronization *23*
Priority Inversion *24*
Unbounded Priority Inversion *25*

Real-Time Synchronization Protocols 26
The Priority Inheritance Protocol *27*
The Priority Ceiling Protocol *29*
Example of the Priority Ceiling Protocol *30*
Priority Ceiling Protocol Emulation *31*

Aperiodic Tasks 32
Aperiodic Servers *33*

Dealing with a Limited Number of Priority Levels 34
Example Scenario for Dealing with a Limited Number of Priority Levels *35*

Other Capabilities of Real-Time System Analysis 37

Recommendations for Real-Time System Builders 38

Object-Oriented Techniques in Real-Time Systems 39

CORBA 40
A Real-Time CORBA System *41*
The Real-Time CORBA 1.0 Standard *42*

UNIFIED MODELING LANGUAGE (UML) 43
UML Diagrams 44
Real-Time Extensions to UML 45

TIMESYS SOLUTIONS FOR REAL-TIME SYSTEM DEVELOPERS 46
TimeSys Linux/RT™: A Real-Time OS with All the Benefits of Linux 48
TimeWiz®: An Integrated Design Environment for Real-Time Systems 49
TimeBench™: A Workbench Environment for Design and Visualization of Real-Time Systems 50
TimeTrace™: A Real-Time Profiling Environment 51

FUTURE RELEASES OF THIS HANDBOOK 52

GLOSSARY OF TERMS AND CONCEPTS 53

SOME KEY REFERENCES ON RESOURCE MANAGEMENT FOR REAL-TIME SYSTEMS 60

What are Real-Time Systems?

Real-time computing systems are systems in which it is possible to predict and control when different computations are handled. They are critical components of an industrialized nation's technological infrastructure. Modern telecommunication systems, automated factories, defense systems, power plants, aircraft, airports, spacecraft, medical instrumentation, SCADA systems, people movers, railroad switching, and other vital systems cannot operate without them.

In real-time applications, the correctness of a computation depends not only upon its results but also upon the time at which its outputs are generated. The measures of merit in a real-time system include:

- **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 time-sharing systems. The following chart shows what behavior each type of system emphasizes in several important arenas.

	Time-Shared 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

These 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

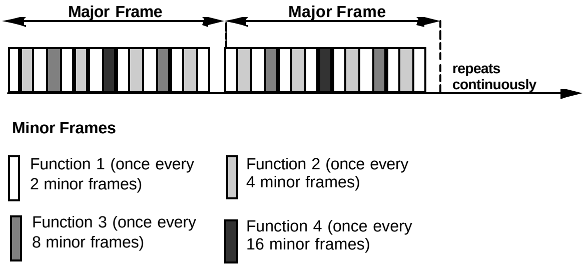
Virtually all real-time applications use one of four (overlapping) architectural patterns:

- A cyclic executive (also called a “timeline” or frame-based system)
- Event-driven systems with both periodic and aperiodic activities
- Pipelined systems
- Client-server systems

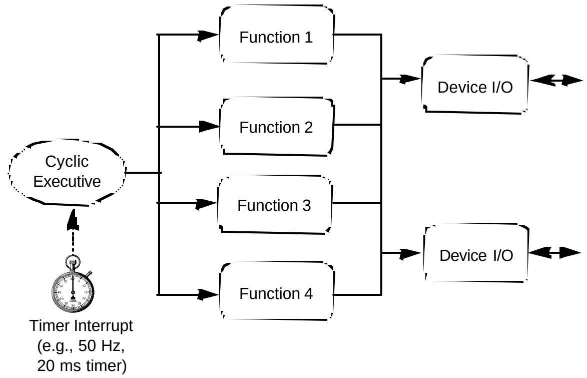
Cyclic Executives

- A timeline uses a timer to trigger a task every minor cycle (or frame).
- A non-repeating set of minor cycles comprises 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.

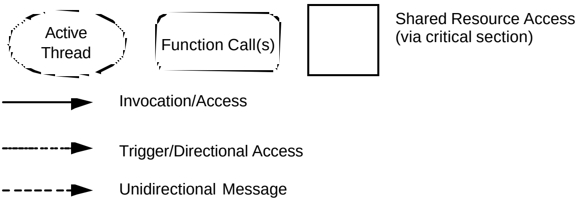
Below is a sample cyclic executive; it consists of minor frames and major frames. Major frames repeat continuously. Within a minor frame, one or more functions execute. Suppose that a minor frame is 10 ms long. Consider 4 functions that must execute at a rate of 50Hz, 25 Hz, 12.5 Hz, and 6.25 Hz respectively (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 is idle in the major frame and can be used for future expansion.


Major Frame
Major Frame
repeats continuously
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



Key:


Active Thread
Function Call(s)
Shared Resource Access (via critical section)
Invocation/Access
Trigger/Directional Access
Unidirectional Message

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

Event-Driven Systems

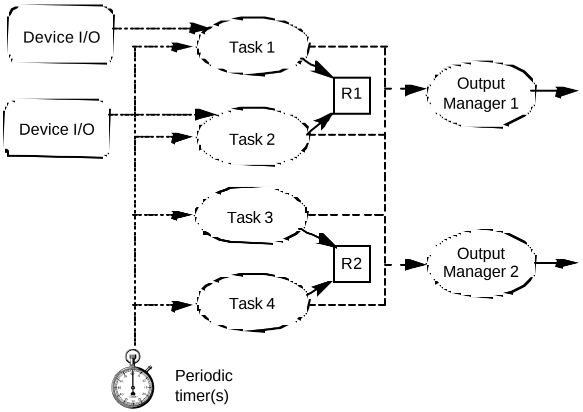
An event-driven design uses real-time I/O completion or timer events to trigger schedulable tasks.

Tasks have priorities:

- •Priorities should be determined by time constraints (e.g., rate-monotonic or deadline-monotonic priority assignment policies).
- •Task priority can also be based on semantic importance (but 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.



Pipelined Systems

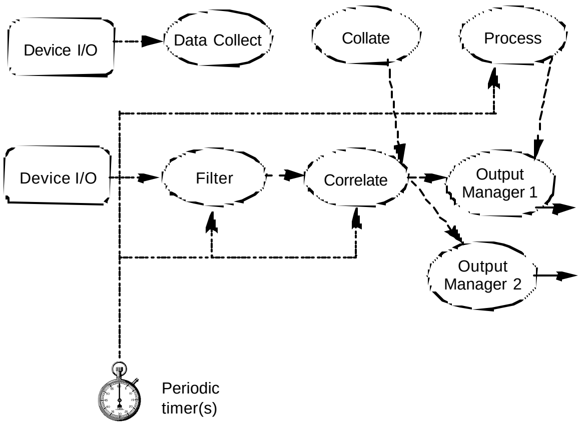
Pipelined systems use possibly prioritized inter-task messages in addition to I/O completion and timers to trigger tasks.

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:

- A unidirectional pipeline with increasing task priorities will minimize message queue buildup.
- If the pipeline is bi-directional, task priorities are usually equal along the pipeline.



Client-Server Systems

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

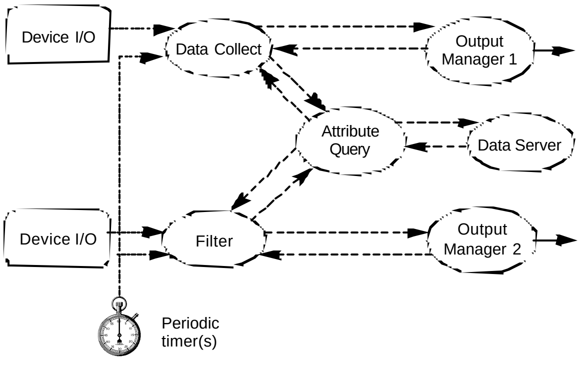
Sending tasks, or clients, block pending response from receiving tasks, or servers.

Control for an event remains at a single system node while dataflow 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 are frequently set the same, using prioritized messages to avoid bottlenecks.



Why Do Timing Analysis?

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), the 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 and the comfort that ,given all possible working conditions, your system will do the right thing at the right time. The use of scientifically proven methodology offers this guarantee.

Enhanced System Reliability: The RMA framework and the analyses and simulation that one 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 choice or number of components.

Where Do Timing Requirements Originate?

Timing constraints originate from two sources:

Top-level (Explicit) Requirements

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

Derived (Implicit) Requirements

- *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.

Real-time system requirements are often specified such that derived timing requirements are often the most common source of timing requirements. That is, at the top level, timing constraints may be sparse.

Why Use Schedulability Analysis or Simulation?

- Capture system requirements to use in competitive proposals, and then to 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/cheaper configurations with what-if-analyses and automatic binding of software components to hardware components.
- Ensure that sufficient resources remain for future system expansion.
- Automatically track component costs.
- Automatically track component development history.
- Customize catalog to track your system-specific component attributes, e.g., vendor address.
- Use report designer to incorporate in-house visual and documentation conventions.
- Write plug-ins to analyze and/or simulate in-house components.
- Distribute configuration and cost options to customers with custom catalog.
- Obtain certification by capturing and analyzing your system for the benefit of regulatory and certification bodies.

Real-Time Scheduling Policies

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 is 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 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.

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:

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

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.

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

Benefits:
Simplicity, efficiency, wide support, practicality.

Examples of Periodic Tasks:
Many activities in real-time, embedded and multimedia systems are periodic.

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

Why is the RM 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 $T_2/T_1 = 1.414 = \text{sqrt}(2)$. Similarly, the 69% bound is obtained for a large number of tasks with $U_1 = U_2 = \ldots = U_n$, $C_i = T_{i+1} - T_i$, 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:

$U_i = C_i / T_i + (2*CS) / T_i$

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*CS a small fraction of T_1, 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 priority(task *i*) > priority(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:

$$W_i(0) = 0$$

$$W_i(n+1) = C_i + \sum_{j<i} \lceil W_i(n) / T_j \rceil C_j$$

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

Analyzing Task Synchronization

Real-time tasks typically share resources and services for which they must be prepared to wait if they are unavailable. 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.

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

where:

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

B_n=0.

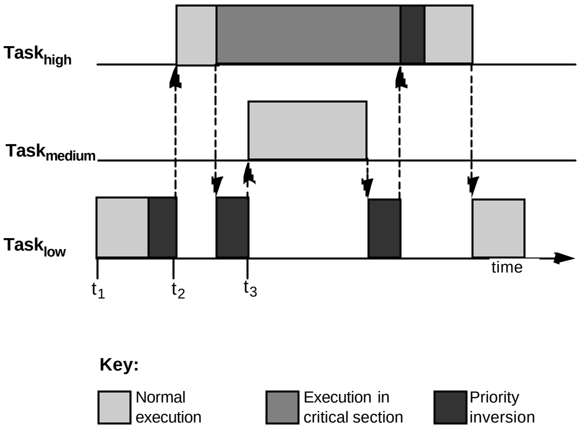
Priority Inversion

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

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

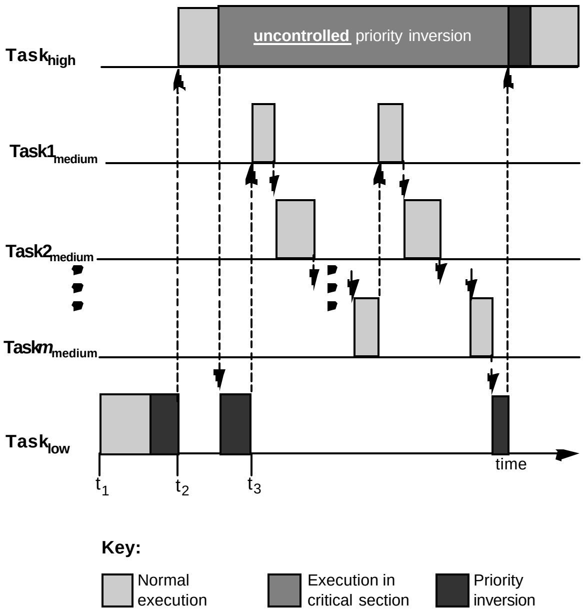
Let Task$_{high}$, Task$_{medium}$, and Task$_{low}$ arrive at times t_1, t_2, and t_3 respectively.

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



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.



Real-Time Synchronization Protocols

Goals:
Bound and minimize priority inversion.

Schemes::

- •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

Comparison of Synchronization Protocols

	Maximum Priority Inversion	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]

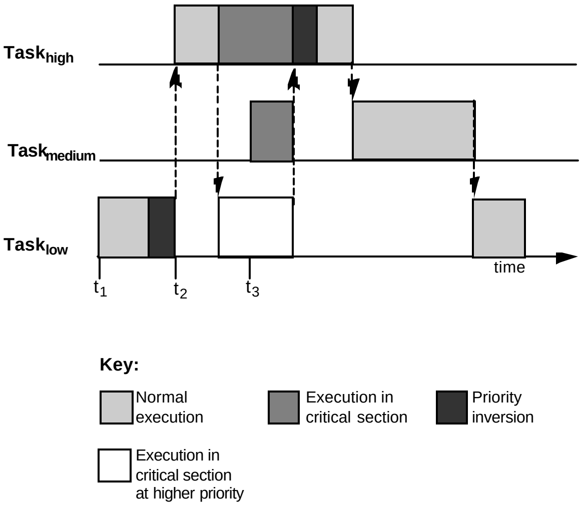
1 A maximum of min(m, n) critical sections, where n is the number of lower priority tasks and m is the number of distinct locks obtained by them. This assumes that deadlocks are avoided by using other schemes such as "total ordering" of the sequence of locks.

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:



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 also happen with sequences of 2 or more tasks.

Avoiding Deadlocks
With the basic priority inheritance protocol, one must use a scheme such as "Total Ordering" while attempting to obtain locks.

That is, number each resource uniquely. Access 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*.

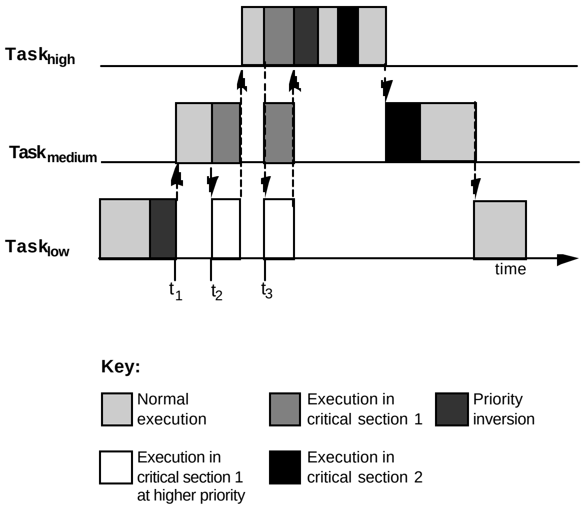
An interesting consequence of the above protocol is that a task may block trying to lock a shared resource, even though the resource is not locked.

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 Task_{high}, Task_{medium}, and Task_{low} in descending order of priority. Task_{medium} accesses Lock 2 and Task_{low} accesses Lock 1. Task_{high} accesses both Lock 1 and 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, Task_{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 Task_{high}, Task_{medium} must block and Task_{low} starts running at the priority of Task_{medium}. Later, at time t_3, when Task_{high} tries to enter Critical Section 1, it has to block as well and Task_{low} starts executing at the higher priority of Task_{high}. When Task_{low} exits its critical section, it resumes its original lower priority. Task_{high} can now enter both Critical Sections 1 and 2. Note that Task_{high}'s priority inversion is bounded by one critical section (that of Task_{medium} or that of Task_{low} but not both).

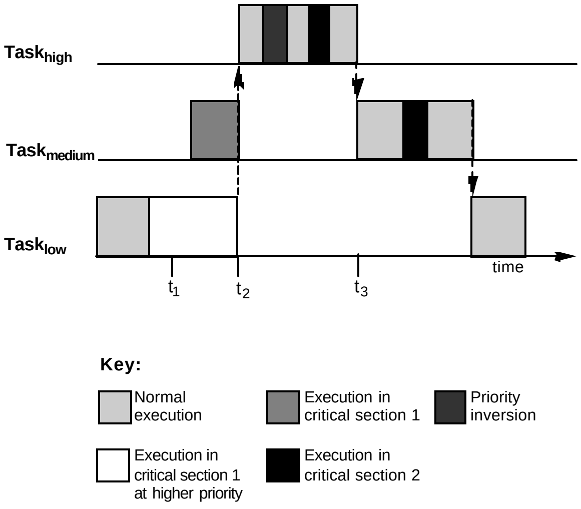


Priority Ceiling Protocol Emulation

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.

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



$\text{Task}_{\text{high}}$
$\text{Task}_{\text{medium}}$
Task_{low}
time
t_1
t_2
t_3
Key:
Normal execution
Execution in critical section 1
Priority inversion
Execution in critical section 1 at higher priority
Execution in critical section 2

Aperiodic Tasks

Tasks in real-time and embedded systems are not always periodic. They can consist of operator requests, emergency message arrivals, threshold crossing notifications, keyboard presses, mouse movements, detection of incoming objects, dynamic software compilation, etc.

The Aperiodic Server Approach
Create a pseudo-periodic task with a period T_{server} and an execution time C_{server}.

The server will get C_{server} worth of "tickets" every successive T_{server} time units.

- Unused tickets are lost when the server's ticket is "replenished".

T_{server} corresponds to a particular priority level based on rate-monotonic priority assignment.

An aperiodic task checks the server on arrival. If the server has a ticket, the task can use the tickets to consume the CPU for the duration of the server at the server's priority level.

Benefits:
Can be used to guarantee hard deadlines for aperiodic ("sporadic") events with minimum inter-arrival times.

Aperiodic Servers

There are two noteworthy 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 is more complex to implement.

Deferrable Server

The replenishment time of tickets is completely independent of ticket usage. C_{server} "tickets" are replenished every T_{server} time units. This is the scheme described above.

Comments:
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.

Sporadic Server

Here, the replenishment time is dependent strictly on ticket usage time.

Used tickets are replenished T_{server} time units following the start of usage.

Example::
If 1 unit of server ticket is (begun to be) consumed at time *t*, and 2 units of server tickets are (begun to be) consumed at *t'*, then the 1 unit can be replenished at time $t+T_{server}$, and the 2 units can be replenished at time $t'+T_{server}$.

Comments:
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.

On the plus side, a system can have multiple sporadic servers on a single node for different categories of aperiodic events with different C's and T's. This is because, in the worst case, the sporadic server behaves like a strict rate-monotonic periodic task.

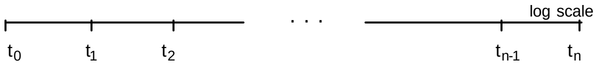
Dealing With A Limited Number of Priority Levels

The original definitions of rate-monotonic and deadline-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

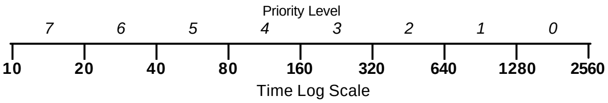
Determine the longest and shortest periods that your system needs to support. Draw the time-spectrum between these two periods on a logarithmic scale, and divide the spectrum equally into *n* segments, where *n* is the number of distinct priority levels available.



We, therefore, have $t_1/t_0 = t_2/t_1 = \ldots = t_n/t_{n-1} = r$, 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.

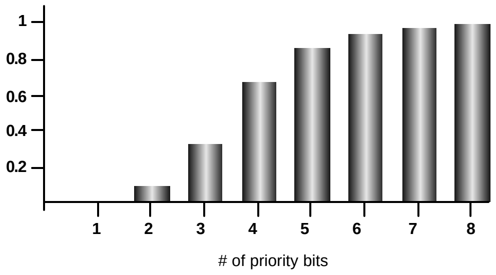
Comments:
This suggested way of assigning priorities with a limited number of priority levels is not optimal, but generally produces a good mapping. For a specific task set, priority assignments with much better schedulability can frequently be obtained manually. This scheme is essentially an analyzable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

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

Schedulability Loss

Suppose that the shortest period to be supported is 1 ms and the longest period is 100 seconds. We have: t_n / t_0 = 100/10-3 = 105. 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.



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 with 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, fighter aircraft, shipboard control, air traffic control, medical instrumentation, multimedia cards, 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/RT, 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 to serve your real-time system needs, as well as a range of consulting services and training courses.
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

- •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.

Recommendations

- •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.

Comments

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

- •The usual OO underlying OS and infrastructure (e.g., CORBA ORBs, X-Windows) implementations usually 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.

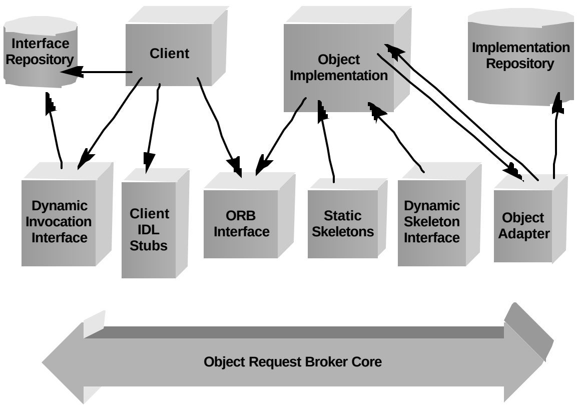
CORBA

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

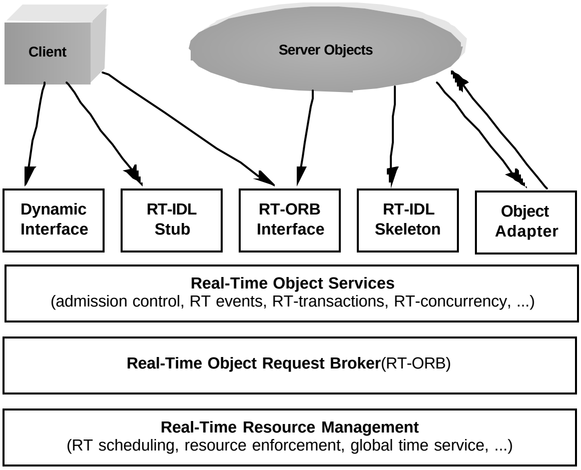
Some of the more than 20 standard services are Naming, Event, Transaction, Event, and Query. CORBA also specifies a neutral Interface Definition Language (IDL), by which all inter-object communication is managed.

A CORBA-based system contains four main components:

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



A Real-Time CORBA System


Client
Server Objects
Dynamic Interface
RT-IDL Stub
RT-ORB Interface
RT-IDL Skeleton
Object Adapter
Real-Time Object Services
(admission control, RT events, RT-transactions, RT-concurrency, ...)
Real-Time Object Request Broker(RT-ORB)
Real-Time Resource Management
(RT scheduling, resource enforcement, global time service, ...)

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.

Unified Modeling Language (UML)

Modeling and analysis of object-oriented systems has had a long and illustrious history. DeMarco and Yourdon developed structured analysis in the late 1970s. The mid-1980s saw behavioral models developed by Ward Mellor and the Harel Charts by Harel. Modeling of object-oriented systems came into vogue in the late 1980s with OMT (Object Modeling Technique), OOSE (Object-Oriented Software Engineering), and the Booch method. The Unified Modeling Language (UML) seeks to bring together the benefits of many of the above techniques under a single unifying umbrella.

The Object Management Group (OMG) is standardizing UML and real-time extensions to UML. OMG released Version 1.1 of UML in 1997. UML supports:

- The visual representation of objects and behavior.
- Design patterns or diagrams.
- A rich set of notations.

UML is useful in requirements and design phases of projects and provides the following basic elements as building blocks:

Objects	Messages	Relationships	Events/ Transitions
name	association	name	name
attributes	aggregation	precursor	parameters
methods	composition	sequence number	guard condition
stereotypes (optional)	generalization	parameters	resulting actions
symbols	refinement	guard condition	other events

UML Diagrams

Diagrams for Requirement Definition

Use-Case Diagram

- This captures a broad view of system functionality, typically from the end-user point of view.

Sequence Diagram

- Illustrates a scenario.
- Scenarios are instances of use cases.
- A complex set of interactions analyzed one scenario at a time
- All messages are enumerated.

Collaboration Diagram

- The collaboration diagram is an alternate way of looking at sequence diagrams.
- Emphasis is on objects and interfaces.
- In contrast, the sequence diagram emphasizes the message sequence between objects.

Diagrams for Design

Deployment Diagram

- Illustrates the system architecture.
- Physical mapping of functionality.

Class Diagram

- Identifies classes and their relationships.

State Chart Diagram

- Shows states, transitions; Harel charts, Moore machines, and Mealy machines.

Activity diagram

- Illustrates procedural flow of control, flow charts.

Real-Time Extensions to UML

Real-time extensions to the Unified Modeling Language (UML) are currently being standardized by the Object Management Group (OMG), which also created CORBA and RT-CORBA. The Real-Time Analysis and Design Working Group of the OMG is currently specifying these real-time extensions.

The real-time extensions to UML, known as RT-UML, will include support for schedulability analysis, stochastic performance analysis, and representation of time and timing constraints, among other aspects still being discussed. Diagrams to represent the timing behavior of real-time and reactive systems are also likely to be in the standard.

TimeSys Solutions for Real-Time System Developers

TimeSys Linux/RT: The first truly real-time version of Linux was created by TimeSys, and is now being distributed. Linux/RT offers a complete real-time system with the reliability and stability that are hallmarks of Linux. Available in multiple packages, Linux/RT 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 and Windows 98. A detailed data sheet, an application example, and a product brochure are enclosed. We also require an annual maintenance contract priced at 15% of the product price, which includes technical support and product upgrades. The product comes with a full 30-day money back guarantee.

TimeBench™ is a friendly and visual software environment for designing, modeling, analyzing, documenting, and visualizing the timing behavior, reusibility, and object-orientation of dynamic real-time and dependable systems. The software incorporates TimeSys-developed extensions to RT-UML and allows users to seamlessly translate models into source code and vice versa.

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.

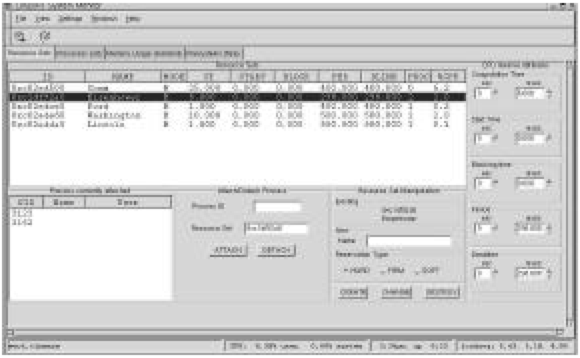
Training: A two-day training course at the customer site will be provided on the use of RMA (Rate-Monotonic Analysis), TimeWiz®, and TimeTrace™.

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. Motivated by our experience with a broad base of customers, we offer TimeTrace™ on a wide variety of platforms ranging from TimeSys Linux/RT to standard real-time operating systems such as VxWorks, pSoS, Windows NT and Windows CE, as well as solutions that can be custom-developed for your environment to assist in collecting the necessary timing data and incorporating them in timing analysis.

Custom Project-Wide Catalogs: This service includes custom creation and centralized storage of objects, properties, analysis, and simulation models for re-use within the organization, using TimeWiz® facilities. This customization is essential so that your architecture is accurately modeled and its timing characteristics are accurately analyzed and simulated in the timing analysis.

Custom Services: We can develop custom solutions that best suit your unique needs on open or proprietary run-time environments. This may include the establishment of a visual and friendly environment for precise performance measurements, or the use of appropriate real-time middleware services such as group communications.

TimeSys Linux/RT^{TM}: A Real-Time OS with All the Benefits of Linux

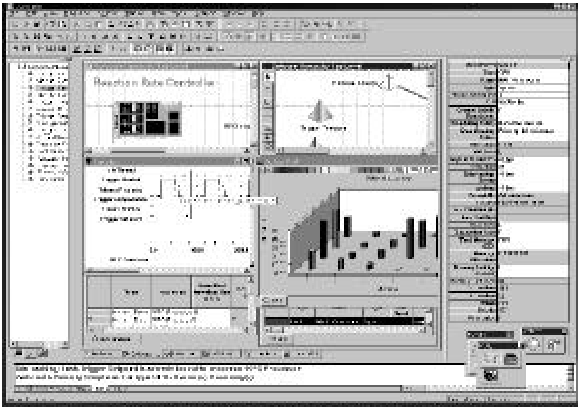


TimeSys Linux/RT is the first Linux-based operating system to offer full real-time capabilities. Linux/RT incorporates the revolutionary Resource Kernel, which offers these features:

- Fixed-priority scheduling with 256 priority levels
- Priority inheritance to avoid unbounded priority inversion
- Quality of Service (QoS) support for resource reservation
- Fine-grained control over which processes run after their reservations expire
- High-resolution clocks and timers
- Ability to execute tasks in periodic fashion
- Stable design

In addition to running on standard computer systems. Linux/RT can also be embedded into systems that lack disks and displays.

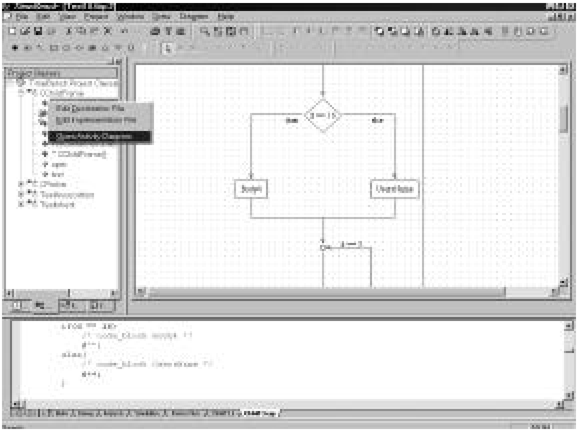
TimeWiz®: An Integrated Design Environment for Real-Time Systems



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.

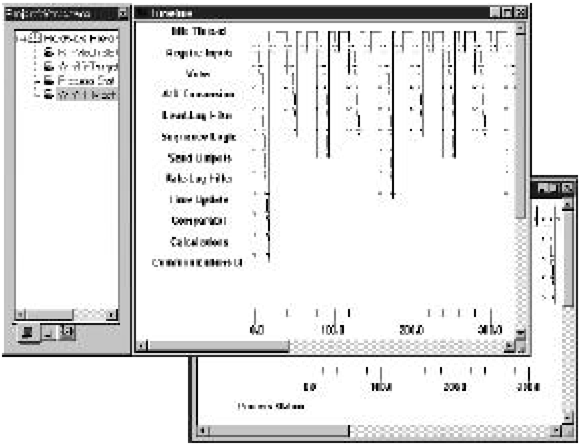
TimeBench™: A Workbench Environment for Design and Visualization of Real-Time Systems



TimeBench is a TimeSys tool to aid you in visualizing your system as it goes through every phase of the development process. It allows you to:

- Represent the static hierarchies and timing information of real-time and embedded systems visually and consistently.
- Use the object hierarchy and timing information representation to automatically generate code for specific targets.
- Reverse-engineer code added by the user to automatically re-generate the static hierarchy and timing information, thereby consistently maintaining the visual and semantic configuration of the system.
- Perform integrated timing analysis of the system taking the object-orientation of the system explicitly into account.

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".
- Monitor multiple targets simultaneously from a single workstation.
- Feed TimeTrace data into TimeWiz as execution time and period parameters for worst-case analysis and/or average-case simulation.

TimeTrace currently supports Linux/RT, LynxOS, Windows NT, Windows CE, and Real-Time Mach, an open source code real-time operating system. Expect additional support in the future.

Future Releases of This Handbook

If you would like a colleague or a group to receive free copies of this handbook, please register their names with TimeSys (*www.timesys.com*) and they will be immediately sent copies of the handbook.

Future versions will include new sections on:

•Dealing with Jitter

•Dealing with Distributed Event-Driven Systems

•Modeling and Analyzing An End-To-End Pipelined System

•Priority Inversion in a Distributed Client-Server System

•Modeling and Analyzing A Distributed Client-Server System

•High Availability Support

•Embedding RT-Java

•RT-Java Tools

•Disk BW Reservations

We encourage you to add your name to the TimeSys mailing list at *www.timesys.com/handbook* to get future versions of this "Concise Handbook of Real-Time Systems".

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/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 *must* 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.

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.

	Determinism is a consequence of the scheduling policies supporting a group 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 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 pri-

	ority 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 *physical* entity such as a processor, a backplane 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 *process*.
Tracer	A stimulus. Synonymous with a single instance of an EVENT within TimeWiz, and is used to represent an end-to-end data flow sequence spanning *multiple* 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 TRIGGER.
Trigger	A stimulus with an arrival pattern. Mostly synonymous with the term "EVENT" within TimeWiz but is used to name an event whose response consists of a chain of actions executing on, at most, a *single* resource. 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 Tracer.
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 OUTPUT JITTER.

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.

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 East47th 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.

J. P. Lehoczky, L. Sha, and Y. Ding, "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.

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.

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.