REAL-TIME OPERATING SYSTEMS

Any discussion of Real-Time Operating Systems (RTOSs) must separate the features of past, current RTOSs and future RTOSs. The tasking management approach used in previous RTOSs and some current RTOSs employed scheduling schemes that had fixed priorities for the applications and offered limited time services → these RTOSs can be broadly classed as *priority driven*.

Priority Driven Systems

A typical example of this class of early RTOS was the Data General RTOS™ product which had five major functional components:

- Device drivers
- System call handlers
- Interrupt handlers
- Task call handlers
- Scheduling mechanism

The operational states were:

- executing an application task
- handling an interrupt
- executing a task call and performing rescheduling

All tasks had fixed priorities and the scheduler maintains two task queues - a *ready task* queue and a *suspended task* queue.

The highest priority task in the ready task queue has control of the processor until a task call in performed by the executing task or by invocation of the scheduler on interrupt exit. While this provided a relatively simple system → there was a complete absence of explicit time notions in control and management structures → any user task could vary its execution time that could adversely influence system behaviour.

Priority Driven Systems with Enhanced Time Services

An example of the class of operating system is provided by Intel who introduced the iRMX 80x86 family of microprocessors. This type of operating system supported a rich set of time services and modern approaches for task management, interrupt handling, synchronization, and communication. Some important features were:

- pre-emptive priority scheduling combined with a round-robin mechanism and periodic reactivation of tasks.
- interrupt handler is fast and predictable
- message communication employs mailbox structures combined with semaphores for intertask communications.
- time management is local and alarm services allowing periodic and aperiodic awakening are provided.

These operating systems still do not include an explicit notion of time for resource allocation and task execution decisions. A standardisation effort for this type of operating system was initiated by Motorola in their RTEID (Real-Time Executive Interface Definition), but was later overtaken by the IEEE POSIX (Portable Operating Systems Interface uniX) standardisation effort.

Operating systems in this class still allow interrupt service routines to pre-empt running tasks with imminent deadlines → can result in a deadline miss. RTEID allowed user programs to alter real-time characteristics of the entire system using priority set primitives. A limited, and not completely satisfactory solution, to this problem was provided by allowing the operating system to restrict task action 'scenarios' → led to research interest in another operating system class.

Time Driven Scheduling based Systems

The primary model used in Time Driven Scheduling (TDS) is a time varying *value function* which is used as a measure of criticality of a task.

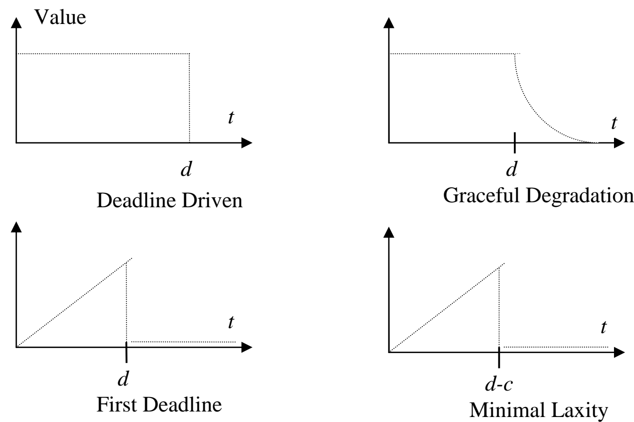
The assumptions used in the TDS model are:

- the computation time of tasks have known stochastic distribution parameters.
- task completion time can be analysed in terms of criticality

The model defines each value function into three domain sections, and over each section it is given by an expression of the form:

$$v(t) = K_1 + K_2t + K_3t^2 + K_4e^{-K_5t}$$

TDS uses 15 coefficients to define each value function and the following diagrams illustrate four of the common time driven scheduling value functions:



- Deadline Driven - hard deadline at d $\rightarrow$ no reason to schedule after this.
- Graceful Degradation - soft deadline $\rightarrow$ exponential drop to zero:

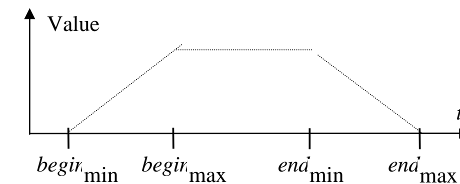
$$\text{i.e. } v(t) = \begin{cases} 0 & 0 \le t \\ K_1 & 0 < t \le d \\ K_4 e^{-K_5(t-d)} & t > d \end{cases}$$

- First deadline - linearly increasing criticality followed by no reason to schedule:

$$\text{i.e. } v(t) = \begin{cases} 0 & 0 \le t \\ K_1 + K_2t & 0 < t \le d \\ 0 & t > d \end{cases}$$

- Minimal Laxity - Maximum criticality is reached before the deadline and after that no reason to schedule.

Combining the TDS model with a general time constraint model (defined in more detail later) permits the following value function to be applied:



and the value function can be represented by:

$$\text{i.e. } v(t) = \begin{cases} K_{1,1} + K_{2,1}t & begin_{max} \le t\,(K_{1,1} + K_{2,1}begin_{min} = 0) \\ K_{1,2} & begin_{max} < t \le end_{min} \\ K_{1,3} - K_{2,3}t & t > end_{min}\ (K_{1,3} - K_{2,3}end_{max} = 0) \end{cases}$$

The value function based model is used with a *best-effort scheduling* algorithm, which has the goal of maximising the total value of all completed tasks over all workloads and given value functions.

In the algorithm it is necessary to compute an *overhead estimation* heuristic at each scheduling point to obtain a probability of overload, so that overloads where tasks cannot meet deadlines are predictable. In the event that a near overload occurs (i.e. this probability exceeds some threshold) then a *load shedding* heuristic is invoked which aborts tasks to decrease the probability below the threshold. The order of task abortion conforms to an expected *value density*. Such heuristic algorithms are

computationally expensive, so most TDS approaches use approximation schemes.
In the design of TDS algorithms, the scheduling policy and scheduling mechanisms are separated → policy modules are located in the operating system kernel and function calls carry out communication between modules and the scheduling mechanisms.
Typical primitives are:

Set_Policy (schedule, policy_module_name)
Set_Attribute (schedule, K_1, K_2, . . .)

The advantage of locating the policy modules in the kernel is to reduce system access overhead, but the disadvantage is an increase in maintainability overhead.

Note that the TDS algorithm does not provide guaranteed deadline compliance, but it can be overlayed on a conventional operating system by creating a global reference for task value functions and using the task abort/restart capability based on the value function of all tasks.

Deadline-Guaranteeing Operating Systems

An example of this class of operating system is the Spring kernel developed by the University of Massachusetts (USA) - primary scheduling is based on explicit time constraints.

The main features are:

- *Architecture support*: the operating environment is a physically distributed multiprocessor system where each multiprocessor has one or more system processors, one or more application processors, and an I/O subsystem. Management activities (e.g. scheduling and allocation) are executed by the system processor and I/O subsystem so that all overhead activities are offloaded from application processes. This improves both performance and temporal determinism of the application processes. Interrupts are handled by the I/O subsystem → they also do not disturb the application processes.

- *Allocation and Scheduling*: a table of guaranteed tasks is maintained with resource allocation and scheduling for execution being implemented in four modules:
  1. Dispatcher - initiates task from the table of already ordered guaranteed tasks.
  2. Local Scheduler - verifies local deadline satisfaction and orders the guaranteed task table.
  3. Global Scheduler - allocates remote processor nodes for task execution that cannot be guaranteed locally.
  4. Meta-level controller - an environment change controller and supporter of user interfaces.

- *Segmentation*: according to applications and their required resources, the OS produces metrics and resource segment sizes. Temporal constraints, data structures and memory modules are treated as resources with segments that can be allocated to computations.
- *Task management*: the Spring kernel provides primitives to manipulate scheduling entities, e.g. task criticalities can be queried and set, off-line tasks can be queued, task execution suspended and resumed, and explicit time constraints can be specified.
- *Memory management*: the Spring kernel makes no use of virtual memory policies due to page fault and page replacement indeterminacy. A resource segmentation approach is followed instead, i.e. memory is pre-allocated → restricts the dynamic behaviour of data structures.
- *Inter-process communication*: message passing via mailboxes is supported with no use of shared memory. Note that no mutual exclusion mechanism is required due to the scheduling approach used, i.e. part of the verification of guarantees is to check for the availability of data resource segments.

Assessment of RTOS approaches

An identifiable trend is the evolution from priority based systems through time-deterministic systems to time-motivated systems.
Several problems areas remain, e.g:

- implementation specific solutions to obtain hard real-time performance → difficult to generalise.
- need to provide for execution begin time constraint and a deadline constraint → aim is to enhance predictability.
- need to include fault tolerance goals → damage containment (graceful degradation) under exception conditions.

SUPPORT ENVIRONMENTS FOR REAL-TIME SYSTEMS

Real-Time Operating Systems (RTOSs) provide a wide range of run-time facilities to support distributed real-time systems, e.g. task scheduling, resource management, fault and interrupt handling, and support for multiple processor operation. There are two common approaches to providing multiple processor support:

1. Shared memory systems - the scheduler determines process distribution to processors as well as process scheduling
2. Loosely-coupled systems - an RTOS on each processor has application programs that use a kernel providing communication primitives

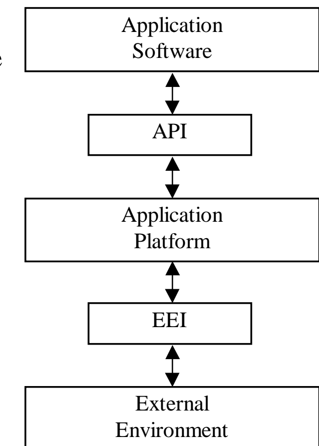
As we are primarily interested in loosely-coupled distributed systems → RTOSs with suitable inter-processor communication services are of most interest - the most common and popular is Real-Time variants of the Unix operating system.

The original Unix OS was not designed to support real-time systems, i.e. aimed at time-sharing → did not have a pre-emptive scheduling mechanism. Two approaches emerged to modify Unix - graft an RT kernel onto Unix or redesign from scratch → led to the effort to retain a Unix style interface and standardise it.

Portable Operating System Interface uniX (POSIX) - IEEE 1003

The POSIX operating systems environment (OSE) has been assembled by working group 1003.0 of the IEEE TC on Operating Systems. It offers a standard set of interfaces to the primary operating system building blocks, providing portability and interoperability standards.

The reference model used in the POSIX OSE has two standard interfaces, an application program interface (API) and an external environment interface (EEI):



The APIs are generate procedure calls made to the application platform (i.e. the host computer operating system). Portability is ensured through standard APIs.

The external environment is typically composed of I/O devices and communication services - standard EEIs provide for interoperability.

POSIX OSE system services include both language and operating systems services.

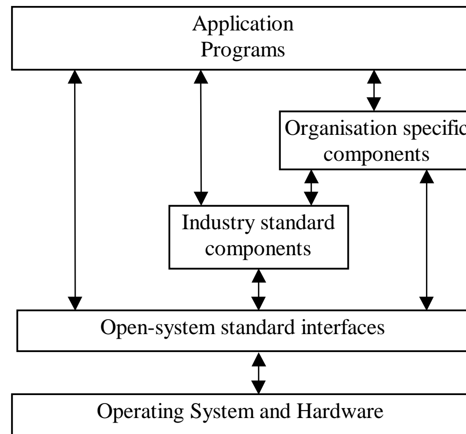
To make services in the OSE accessible from application programs, language bindings (or subprogram calls in specific languages) are provided. The kernel standard (IEEE 1003.1 or ISO 9945.1) is defined in the C-language but FORTRAN (1003.9) and Ada language bindings (1003.5) are also provided.

Other standards in the POSIX OSE include shell and utilities (1003.2) , real-time extensions (1003.4), security extensions (1003.6), system administration (1003.7), network file access (1003.8), and a protocol independent network interface (1003.12).

Additional APIs in the POSIX OSE include non-POSIX standards, e.g. Structured Query Language (ISO 9075) and Network Data Language (ISO 8907).

POSIX based applications are to be constructed is based on a hierarchy of software components:

The lowest level is made up of generic system services, with higher levels provided by APIs specific to industries (e.g. the Interactive Multimedia Association's API for computer aided training systems) and specific to organisations.



The benefits of open-systems standards are well acknowledged by the industry → should lead to more efficient software development on a wider range of hardware platforms.

Real-Time Extensions (IEEE 1003.1b): binary semaphores, memory locking, shared memory, priority scheduling, asynchronous event notification, clocks and timers, interprocess message passing, asynchronous and synhronous I/O, and real time files.

Threads Interface (IEEE 1003.1c): unlike conventional UNIX processes, each POSIX process supports multiple threads of execution in the same process address space, sharing file descriptors, process ID and other context information. Importantly, I/O operations which normally block UNIX processes do not block other threads of execution in the same process. Primatives include thread creation and control, thread scheduling and synchronisation, thread cancellation and cleanup, signals.

Standard Revision: the Austin Common Standards Revision Group was established to produce a common revision of ISO/IEC 9945 and IEEE Std 1003 and the appropriate parts of the Single UNIX Specification. The result was a common IEEE POSIX (IEEE 1003-2001) and Open Group Single UNIX Specification Version 3 released in Jan 2002 (with ISO/IEC approval to follow shortly).

RTOS PERFORMANCE ISSUES

It is important that the real-time services provided by an RTOS are implemented efficiently and that appropriate benchmarks are used to determine suitability for a user's application.

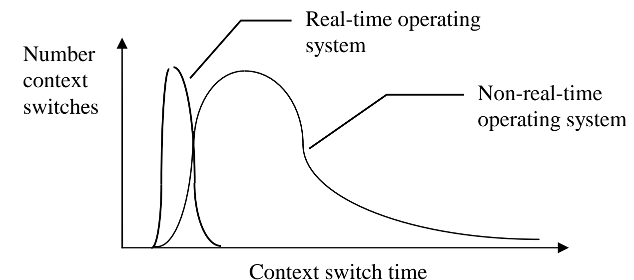
Many RTOS vendors make claims for their implementations which do not necessarily represent worst-case operating conditions.

Some of the basic performance measures are:

- *context switching time*: the time to switch the CPU from executing one process to another.
- *pre-emption latency*: the time delay before a process with a higher priority than the currently executing process will start to execute.
- *interrupt latency*: the time from interrupt reception to the time a handler commences execution.

In general, a "real-time" kernel can achieve context switching in under 100 µsec whereas conventional operating systems often have times exceeding 1 msec.

The actual situation is far more involved, as the following diagram suggests:



It is important that the application context be considered, i.e. if 10 msec laxities on deadlines are acceptable, and the task loading is relatively light, then a non real-time operating system could be acceptable in this application.

The primary cause of the significant variation in context switch times in general purpose operating systems can be identified:

- Their kernels have critical regions which block interrupts for extended periods $\rightarrow$ prevents context switches.
- Memory protection is strongly enforced so that each real-time process (task) is mapped to a Unix (for example) process and sharing memory by executing functions in the memory space of another process is indirect and inefficient.

The approach used in current RTOSs is to provide some form of light-weight process model, e.g:

- VxWorks uses a shared code and data memory space $\rightarrow$ provides for minimal overhead in spawning additional processes.
- LynxOS (and other POSIX RTOS variants) uses threads $\rightarrow$ again minimal overhead in thread creation and switching.

Examples of Real-Time Unix based products (i.e. Unix like shell and utilities but proprietary real-time kernel):

Regulus (Alcyon), pSOS+ (Software Components Group), D-Nix (Diab Systems), RTU (Concurrent/Masscomp), IDRIS (Whitesmiths), VxWorks (Wind River Systems), QNX (Quantum Software Systems), PDOS (Eyring Research), LynxOS (Lynx Real Time Systems), OS9000 (Microware).

THE LYNX REAL-TIME OPERATING SYSTEM

LynxOS is A 'UNIX-like' real-time operating system that is compliant with IEEE P1003.1 Portable Operating Systems Standard for Computer Environments (POSIX) and AT&T System V interface definition (SVID).

LynxOS is also compatible with the real-time extensions (P1003.1b) and the threads extension (P1003.1c) POSIX standards.

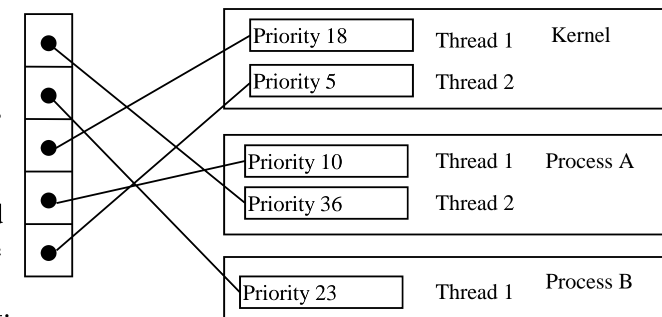
LynxOS is available for the following platforms: i860, 80x86, Sparc, R3000, 680x0, 880x0.

Application areas for LynxOS encompasses both embedded and non-embedded real-time systems - ranging from process control, robotics, manufacturing, defence, aerospace and communications systems.

I/O Interrupt Processing and Kernel Threads

This is a critical performance area for RTOSs as interrupt processing runs at a higher priority than user tasks and some interrupt processing may take 100 µsec $\rightarrow$ 1 msec (e.g. network protocol handling).
Also the number of these interrupts may not be bounded $\rightarrow$ high-priority user tasks may be delayed for unpredictable periods.

LynxOS provides a mechanism to handle this problem with *kernel threads*. Threads are the smallest independently schedulable objects with their own priority and register stack. User threads exist within the address space of a user process, and kernel threads share the kernel's address space. The scheduler treats both thread types in the same way - i.e. on a priority basis, e.g:

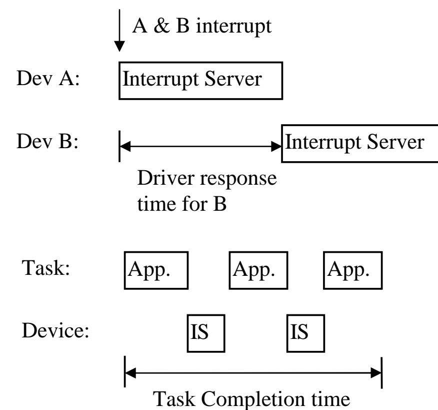


A scheme, referred to as *priority tracking*, is used to set the 'best' priority for a kernel thread:

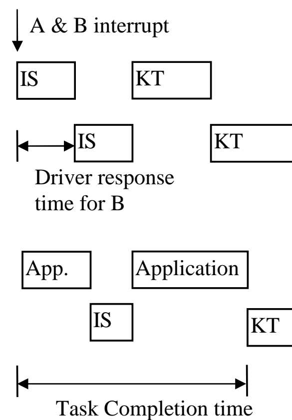
- The priority of the kernel thread is inherited from the highest priority application it serves.
- To improve the resolution of priority levels, 1/2 steps are allowed for kernel threads.
- Example: suppose we have a kernel thread running at priority 18 serving some application, and an application running at priority 23 $\rightarrow$ the kernel thread is raised to priority 23-1/2 so that all applications of priority greater than 23 will run before this thread and this thread will run before all applications with priority of 23 or less.

Kernel threads have a beneficial effect on driver response, task response and task completion times, e.g:

No kernel threads:



With kernel threads:



The task completion time is improved by allowing the low level interrupt handler to disable future interrupts from the device prior to scheduling the kernel thread (which then re-enables them) $\rightarrow$ the thread will not run until higher priority applications complete. Each device is restricted to a single interrupt while higher priority applications are pending.

Benchmarks for Real-Time Operating Systems

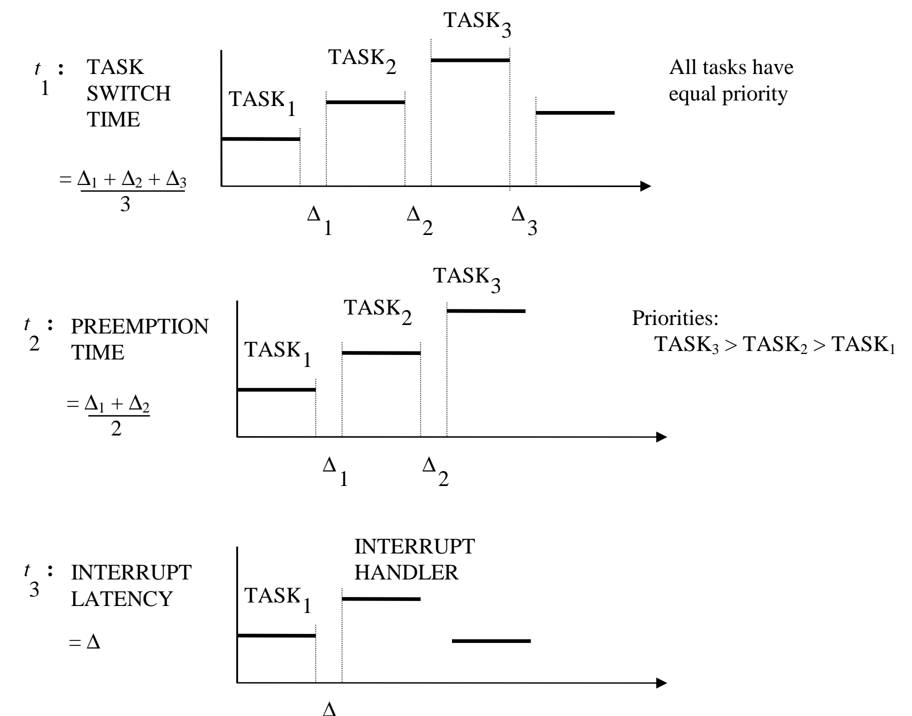
There are two benchmarks which attempt to provide a more balanced set of criteria for performance metrics on RTOSs: *Rhealstones* and *Hartstones*.

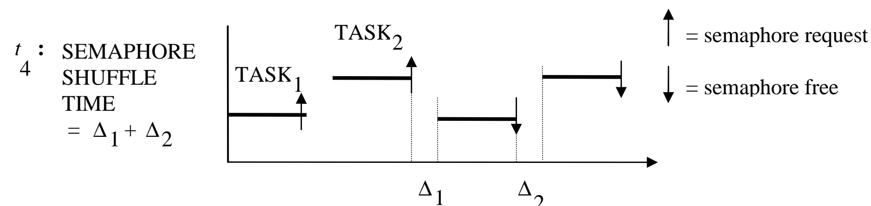
The Rhealstone Benchmark

Defined as the sum of six individual performance measurements:

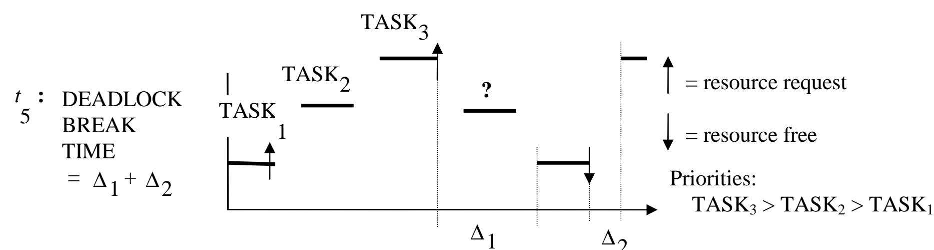
$$\text{Rhealstone} = \frac{6}{t_1 + t_2 + t_3 + t_4 + t_5 + t_6}$$

where $t_1, \ldots, t_6$ are times in seconds obtained from the following measurements:

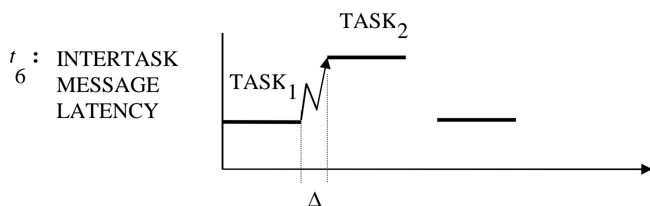




i.e. the *Semaphore Shuffle Time* is the delay between the attempted acquisition of a task's semaphore and the reactivation of the task blocked on a semaphore wait.



i.e. the *Deadlock Break Time* is the time the executive takes to resolve the conflict occurring when a higher-priority task pre-empts a lower-priority task that holds a resource needed by the higher priority task.



i.e. the *Intertask Message Latency* is the time taken to transfer a non-zero length message via pipes, queues, or stream files between tasks.

Weighting factors can be applied to construct an application specific benchmark, e.g:

$$\text{Weighted Rhealstone} = \frac{n_1 + n_2 + n_3 + n_4 + n_5 + n_6}{n_1 t_1 + n_2 t_2 + n_3 t_3 + n_4 t_4 + n_5 t_5 + n_6 t_6}$$

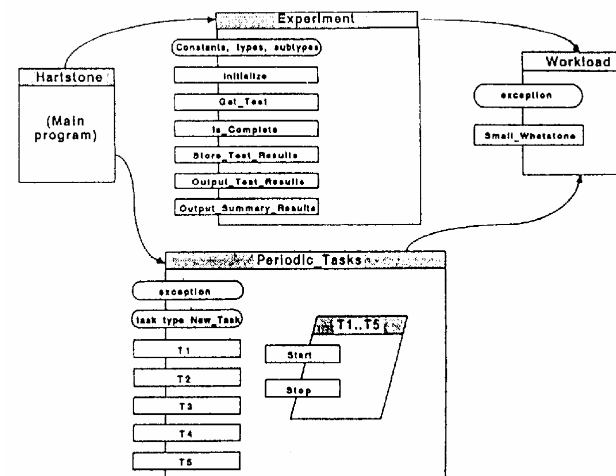
The Hartstone Benchmark

The Hartstone uses a variant of the Whetstone benchmark for the basic computational load imposed on the processor. The benchmark suite is written in the Ada language and is designed to measure the following:

- performance of the Ada runtime environment
- efficiency of handling multiple real-time tasks

The test set is made up of five series of tests; the first is the periodic harmonic (PH) test:

- Consists of five independent periodic Ada tasks which have an assigned *frequency*, *priority* and *workload*. Task frequencies are made harmonic, i.e. are an integral multiple of the frequency of the lowest frequency task. The main loop of each task executes a Kilo-Whetstone.
- The task dependency diagram:



A Hartstone task must execute a specified number of Kilo-Whetstones in its scheduled period, and the rate at which the task performs this is measured in Kilo-Whetstone Instructions per second (KWIPS). The *workload rate* is then the per-period work load multiplied by the task frequency.

The task must also satisfy a *deadline* for completion of the assumed workload - which is the next scheduled activation for the task. Failure to complete on time results in a missed deadline being recorded for that task - *load shedding* is used to skip deadlines and continue operation of the test.

Task priorities are static and are assigned at initialisation of the test using a *rate-monotonic* scheduling discipline (i.e. high frequency tasks are assigned a higher priority than low frequency tasks).

Hartstone PH Tests: A baseline test is performed to obtain an acceptable task workload utilization without any missed deadlines. Utilization in the baseline test can be compared with 100% utilisation in the calibration test done with no tasking and only the computational load applied, e.g. consider a baseline test:

Task No.	Frequency (Hertz)	Kilo-Whets per period	Kilo-Whets per second
1	2.00	32	64.00
2	4.00	16	64.00
3	8.00	8	64.00
4	16.00	4	64.00
5	32.00	2	64.00

			320.00

The first test adjusts the frequency of the highest frequency task (task 5) for each successive cycle of the test until some task misses a deadline. The test continues with this task frequency until a specified number of deadline misses or skips is recorded, e.g. consider the results of a typical first experiment:

```
Experiment:     EXPERIMENT_1
Completion on: Miss/skip 50 deadlines

Raw speed in Kilo-Whetstone Instructions Per Second (KWIPS): 1122.19

Test 21 characteristics:
```

Task No.	Frequency (Hertz)	Kilo-Whets per period	Kilo-Whets per second	Requested Workload Utilization
1	2.00	32	64.00	5.70 %
2	4.00	16	64.00	5.70 %
3	8.00	8	64.00	5.70 %
4	16.00	4	64.00	5.70 %
5	352.00	2	704.00	62.73 %
			-------	--------
			960.00	85.55 %

```
Experiment step size:    2.85 %
```



```
Test 21 results:

Test duration (seconds):  10.0
```

Task No.	Period in msecs	Met Deadlines	Missed Deadlines	Skipped Deadlines	Average Late (msec)
1	500.000	0	7	13	626.683
2	250.000	40	0	0	0.000
3	125.000	80	0	0	0.000
4	62.500	160	0	0	0.000
5	2.841	3520	0	0	0.000

There are three additional "experiments" which are run in the PH test:

1. All task frequencies are harmonically scaled upwards (by 10% on each new test cycle).
2. All task workloads are scaled upwards (by 1 KW per period on each new test cycle).
3. New tasks with the same frequency and workload as task 3 are added on each new test cycle.

The purpose of each additional experiment is to determine the computational load and associated overhead for periodic tasks that exceeds the capability of the run-time environment and forces deadlines to be missed.