Universitas Esa Unggul

Smart, Creative and Entrepreneurial

www.esaunggul.ac.id

CCR210-REKAYASA PERANGKAT LUNAK

Pertemuan Ke 13

Oleh : MALABAY

Prodi : Teknik Informatika/Sistem Informasi

# REAR-TIME SOFTWARE DESIGN

Disusun oleh : Malabay

**Suber dari :**

***http://www.bernstein-plus-sons.com/.dowling/CSC175/SE_RealTime.html***

**Some notes for Software Engineering -- Real-Time**
by Herbert J. Bernstein

## Real-Time

The design of real-time systems can be fundamentally different from the design of ordinary applications. When so-called real-time demands do not crowd push resource limits, then ordinary software engineering approaches may work. However, when the applications proposed push available pools of time, space or other resources, the handling of those constraints can dominate the design leading to approaches that would never be considered in projects less demanding of resources.

- Real-Time
  - Systems that in which the time to respond is a significant constraint
  - Operatings systems, device drivers, data acquisition systems are examples
  - Interactive systems have such constraints (e.g. 1/2 second response)
  - Data communications and network design raise real-time issues
  - Embedded systems often have such constraints
    - Weapons control systems, avionics, etc. raise real-time issues
  - Hard Real-Time -- mandatory time constraints
  - Soft Real-Time -- desirable time constraints
  - Quality and reliability particularly important
  - Types of Real-Time constraints
    - Event driven
      - Periodic events for which response must be completed before the next event arrives
      - Periodic events for which response must be completed within some given time from event arrival
      - Periodic events for which response must be completed sufficiently rapidly to avoid resource exhaustion
      - Sporadic events for which response must be completed before the next event arrives
      - Sporadic events for which response must be completed within some given time from event arrival
      - Sporadic events for which response must be completed sufficiently rapidly to avoid resource exhaustion
    - Intrinsic
      - Tasks that must be completed sufficiently rapidly avoid resource exhaustion or to leave the machine ready for some expected event (e.g. garbage collection, file system management, etc.)
  - Approaches to real-time software design
    - Polling
    - Interrupts
    - Threads

        - Threaded code (has nothing to do with threads)
        - Hardware support
            - Intelligent controllers
            - Hardware clocks
            - Memory management hardware
            - Hardware event silos
            - ...
    - Issues in real-time design
        - Dead-time (inability to respond to events for some period of time)
        - Buffer overflows, overruns
        - Interlocks
        - Synchronization
        - Shared data structures
        - Error recovery
        - Stack overflows
        - Disk I/O delays
        - DMA lockouts and bus overloads
        - Maintainable system
        - ...
    - Approaches to real-time design
        - Queuing Theory
            - Model based on customers, servers and queueing discipline
            - Qualitative behavior derived from M/M/1 queues
            - Service rate must be signficantly faster than customer arrival rate
        - Graph theory
        - Network (graph with flows) models
        - State trabsition tables, graphs
        - Object-oriented design (often with processor(s) for each object)
        - Design in Ada or java, implement in C++/C/Fortran/assembler
    - Approaches to real-time implementation
        - Throw hardware at it and pray -- works well for soft real-time
        - Use efficient languages (C, assembler)
        - Design a real-time monitor or executive
            - Manage time
            - Manage interrupts
                - Premeptive scheduling
                - Polling
            - Schedule tasks
                - Queueing
                - Time sharing
            - Allocate resources
        - Design an operating-system free application
        - Avoid disk I/O
        -

**Sumber dari :**
***https://www.eventhelix.com/RealtimeMantra/DesigningRealtimeSoftware.htm#.XBRkUOIxXDc***

## Designing Real-time Software

Designing Realtime software involves several steps. The basic steps are listed below:

- Software Architecture Definition
- Co-Design
- Defining Software Subsystems
- Feature Design
- Task Design

### Software Architecture Definition

This is the first stage of Realtime Software design. Here the software team understands the system that is being designed. The team also reviews at the proposed hardware architecture and develops a very basic software architecture. This architecture definition will be further refined in Co-Design.

Use Cases are also used in this stage to analyze the system. Use cases are used to understand the interactions between the system and its users. For example, use cases for a telephone exchange would specify the interactions between the telephone exchange, its subscribers and the operators which maintain the exchange.

### Co-Design

Once the software architecture has been defined, the hardware and software teams should work together to associate software functionality to hardware modules. The software handling is partitioned between different processors and other hardware resources with the following key considerations:

- The software functionality should be partitioned in such a fashion that processors and links in the system do not get overloaded when the system is operating at peak capacity. This involves simulating the system with the proposed software and hardware architecture.
- The system should be designed for future growth by considering a scalable architecture, i.e. system capacity can be increased by adding new hardware modules. The system will not scale very well if some hardware or software module becomes a bottleneck in increasing system capacity. For example, Xenon scalability will be limited if the CAS processor in the system is assigned a lot of work. As this processor is shared, the software running on the CAS card would become a scalability bottleneck.
- Software modules that interact very closely with each other should be placed on the same processor, this will reduce delays in the system. Higher system performance can be achieved by this approach as inter-processor message communication taxes the CPU as well as link resources.

This stage is sometimes referred to as Co-Design as the hardware and software teams work together to define the final system architecture. This is an iterative process. Changes in system architecture might result in changes in hardware and/or software architecture.

The next step in Realtime system design is the careful analysis of the system to define the software modules.

## Defining Software Subsystems

1. Determine all the features that the system needs to support.
2. Group the various features based on the type of work they perform. Identify various sub-systems by assigning one subsystem for one type of features. For example, for the Xenon switch the groups would be Call Handling, System Maintenance, Operator Interface etc.
3. Identify the tasks that will implement the software features. Clearly define the role of each task in its subsystem.
4. Within each subsystem, classify and group the features appropriately and associate the various tasks constituting the subsystem. For example, the Call Handling subsystem in Xenon would support features like:
    - V5.2 Originating to ISUP Outgoing Call
    - V5.2 Originating to V5.2 Terminating Call
    - Conference Call
    - Toll free call

## Feature Design

A typical Realtime system is composed of various task entities distributed across different processors and all the inter-processor communication takes place mainly through messages. Feature Design defines the software features in terms of message interactions between tasks. This involves detailed specification of message interfaces. The feature design is generally carried out in the following steps:

1. Specify the message interactions between different tasks in the system
2. Identify the tasks that would be controlling the feature. The controlling tasks would be keeping track of progress of feature. Generally this is achieved by running timers.
3. The message interfaces are defined in detail. All the fields and their possible values are identified.

## Feature Design Guidelines

- Keep the design simple and provide a clear definition of the system composition.
- Do not involve too many tasks in a feature.
- Disintegrate big and complex features into small sub features.
- Draw message sequence charts for a feature carefully. Classify the legs of a scenario for a feature in such a way that similar message exchanges are performed by taking the common leg.
- Provide a clear and complete definition of each message interface.

- To check possible message loss, design timer based message exchanges.
- Always consider recovery and rollback cases at each stage of feature design. One way of doing this is to keep a timer for each feature at the task that controls the mainline activity of the feature. And then insert the timeout leg in the message sequence charts of the feature.
- To avoid overloading of message links choose design alternatives that include fewer message exchanges.

## Task Design

Designing a task requires that all the interfaces that the task needs to support should be very well defined. Make sure all the message parameters and timer values have been finalized.

### Selecting the Task Type

Once the external interfaces are frozen, select the type of task/tasks that would be most appropriate to handle the interfaces:

- **Single State Machine:** The tasks functionality can be implemented in a single state machine. The V5.2 Call task in Xenon is a good example of a task of this type.
- **Multiple State Machines:** The task manages multiple state machines. Such tasks would typically include a dispatcher to distribute the received messages to an appropriate state machine. Such tasks would create and delete state machine objects as and when required. The E1 Manger in Xenon exemplifies such a task.
- **Multiple Tasks:** This type of tasks are similar to the multiple state machine tasks discussed above. The main difference is that the task now manages multiple tasks. Each of the managed tasks implements a single state machine. The manager task is also responsible for creating and deleting the single state machine tasks. The V5.2 Manager task is a good example of a task of this category.
- **Complex Task:** This type of task would be required in really complex scenarios. Here the manager task manages other tasks which might be managing multiple state machines.

### Selecting the State Machine Design

After choosing the type of task, the designer should consider dividing the message interface handling into a sequence of state transitions. Two different type of state machines can be supported:

- **Flat State Machines:** This is the most frequently used type of state transition. For example, the states of a call would be "Awaiting Digits", "Awaiting Connect", "Awaiting Release", "Awaiting On-hook" etc. This type of state division does not scale very well with increasing complexity. For a complex system this technique results in a state explosion, with the task requiring hundreds of states.
- **Hierarchical State Machines:** Here the states are viewed as a hierarchy. For example the states covered above would map to "Originating : Awaiting Digits",

"Originating: Awaiting Connect", "Releasing : Awaiting Release", "Releasing : Awaiting On-hook". The total number of states is the same here, but the main difference is that some states inherit from an "Originating" base state and others inherit from "Releasing" base state. The total number of message handlers in each state would be reduced drastically, as all the messages that have common handling in all the originating states would be just handled in the "Originating" base state. In addition to this, the handlers of inheriting states can further refine the base state handling by taking additional actions.

**Task Design Guidelines**

- Do not complicate the design by introducing too many states. Such designs are very difficult to understand. Follow a simple rule of thumb, if you are having difficulty choosing the name of state, you may have identified the wrong state.
- Do not complicate the design by having too few states. If all the states in the system have not been captured in the state machine design, you will end up with lot of flags and strange looking variables which will be needed to control the message flow in the jumbo states.
- Keep the data-structure definitions simple. See if a simpler data-structure would do the same job just as well.
- Out of memory conditions should be handled. Tasks can and will run out of memory. So handle the out of memory conditions gracefully. This can lead to a lot of "if clutter" so consider exception handling as an option.
- All of the legs of the defined scenarios should be handled. This is easier said than done. Many times all the scenario legs identified in the feature design stage may not cover all the possible error scenarios for your task.
- Make sure that all the allocated resources are de-allocated at the end. Again it is very easy to miss out on this one. Many times designers forget to release resources in the error legs.
- Consider using a hierarchical state machine to simplify the state machine design.
- Consider using Object Oriented programming languages like C++. Contrary to popular belief, languages like C++ might turn out to be more efficient in runtime performance because of better locality of reference. (Most objects would be referring to data that is contained in the same object, thus improving the locality of reference)

Sumber dari :

***https://dzone.com/articles/what-is-realtime-1***

Many software developers are familiar with realtime, but we believe
that realtime concepts and user experiences are becoming increasingly important for less technical individuals to understand.

At Fanout, we power realtime APIs to instantly push data to endpoints — which can range from the actual endpoints of an API (the technical term) to external businesses or end users. We use the word in this post loosely to refer to any destination for data.

We're here to share our experience with realtime. We'll provide a definition and current examples, peer into the future of realtime, and try and shed some light on the eternal *realtime* vs. *real-time* vs. *real time semantic debate*.

## The Simple Definition

Realtime refers to a synchronous, bi-directional communication channel between endpoints at a speed of less than 100ms.

We'll break that down in plain(er) English:

- **Synchronous** means that both endpoints have access to data at the same time (not to be confused with sync/async programming).
- **Bi-directional** means that endpoints can send data in either direction.
- **Endpoints** are senders or receivers of data; they could be anything from an API endpoint that makes data available to a user chatting on their phone.
- **100ms** is somewhat arbitrary; data cannot be delivered instantly — but under 100ms is pretty close, especially with respect to human perception. Robert Miller proved this in 1986.

## An Example of a Realtime User Experience

A simple example of a realtime user experience is that of a chat app. In a chat app, you "immediately" (sub-100ms) see messages from the person (endpoint) you're chatting with and can receive information about when they read your messages (synchronous, bi-directional).

## Realtime vs. Request-Response

Web experiences are beginning to move from request-response experiences to live, realtime ones. Social feeds don't require a refresh (a request) to update, and you don't need to email documents as attachments that need to be downloaded (request) and sent back with edits (response) — you just use collaboration software that works in realtime.

## More Realtime Experiences

Realtime user experiences are everywhere you look — especially where near-instant access to information is valuable. You'll find realtime in:

- **Collaboration**: Realtime access to internal and external information from your team is becoming the norm. It's accepted that a sales inquiry (data) can be instantaneously relayed from live chat on your website, into your customer service portal, and then into Slack.
- **Finance**: Stock tracking and bitcoin wallets require immediate access to information. Applications like high-frequency trading exist specifically because of the ability of certain parties to access and act on data faster than others.
- **Events**: Second-screen experiences for sports, including live betting with realtime odds updates, are becoming increasingly common.

- **Crowdsourcing**: Distributed collection, analysis, and dissemination of data from distributed endpoints (think reports from WeatherUnderground stations or from the traffic app Waze) is only valuable when it occurs in realtime.

**Realtime in the Future**

As we see it (and admittedly, we are a little biased), realtime is quickly becoming the new normal. Up-to-date information is expected by businesses
and end users. Realtime is the natural complement to trends like:

**Big Data**

As the number of digitally connected businesses, experiences, and devices rises, so does the amount of data generated. Data becomes more valuable as the three Vs of a dataset (velocity, volume, and variety) increase — and realtime transmission is central to the velocity component.

In the past, companies benefitted from hoarding data, but increasingly, data is becoming most valuable when shared (and monetized). The companies that can aggregate and share the most data, as quickly as possible, will be successful.

**Proliferation of APIs**

Businesses sharing data are increasingly going to do so through APIs. Entire businesses are being built on APIs by platform providers like Twillio (they only have an API) or they are coming to comprise substantial portions of existing businesses (like Salesforce's API).

An elegant end-user experience is increasingly the product of data that's being moved through multiple APIs — and the number of APIs is only going to increase as they trend toward becoming less technical and more accessible and interoperable. The APIs that provide access to data or move it through their system as quickly as possible will rise over those that cannot.

**Realtime vs. Real-time vs. Real Time**

The endless debate — what's the correct way to write what we've been discussing? We use realtime because we believe that "real time" refers to something experienced at normal speed and not condensed or sped up. For example, watching grass grow in 'real time' is not very exciting — but a time lapse is.

**Sumber dari:**

***http://ecetutorials.com/control-systems/real-time-system-defination-and-types-of-realtime-system/***

**Real Time System defination and types of realtime system**

**Real time systems**

**Real time:** It is the time span taken by the system to complete all its tasks and provides an output for an input. This time span should be the same for computation of all its tasks.

**Real time system:** Real time systems are those which must produce the correct response within the specified or defined time limit. If it exceeds these time bonds it results in performance degradation and/or malfunction of system.

For example in aircraft engine control system, the real time control system should perform its task within a specified time as the operator/pilot intended and failure of this can cause the loss of control and possibly the loss of many lives.

**Real time program:** A program for which the correctness of operation depends upon the logical output of the computation and the time at which the results are produced. Every real time system must be having real time clock which specifies the time of the execution of the task or interruption of the task.

## Types of real time system:

As per the clock and execution procedure of task the real time systems are divided as follows

- Clock based systems
- Event based systems
- Interactive systems

### Clock based real time system:

In this system the computation of its task has to be completed in the specified time interval called real time clock. Most of plant control systems are in this category. The clock can be in hours for some chemical process or it may be in milli seconds for some control systems. For example of feedback control of tank level, the real time system should read the level of the tank, process it with control algorithm and actuate the valve accordingly to maintain the level. These three tasks should perform in the specified time interval i.e sampling of input, processing and output response.

This clock can be continuous or discrete. In continuous the system will perform the task continuously within a specified time. This is same as above tank level controller where it is a continuous control process. In some chemical industries, The chemicals should be added with some specified intervals these are called discrete control systems.

### Event based real time system:

In plants there are some systems where actions have to be performed in response of some events instead of some particular time intervals. For example the control system has to close the value if the liquid level in the tank reaches its high level. Here this action is not time based, its an event based and these are used extensively to indicate the alarm conditions and initiate alarm actions, for example indicating the liquid level in the tank high or temperature of the liquid high etc. The specification of event based systems usually indicates that the system must respond within specified maximum time to a particular event. These systems uses interrupts to indicate the real time system that the action is required. Some small system uses Polling i.e the system periodically asks the various sensors to see whether the action is required. These systems are basically aperiodic tasks and may have deadlines expressed in terms of start up time or finish time. For example after sensing of level of liquid the the valve closer should start after some interval.

## Interactive systems:

The combination of Clock based system and Event based system which gives the importance of average execution time of the task is called interactive systems. This covers the systems like Automatic teller machine, reservation system for hotels, Airlines booking etc. This systems receive the input from the plant or operator and initiate the task and executes within the average response time. For an example if you want draw cash from ATM when u put your card then it process the task of giving the money out. In this case the response time depends on the network traffic and internal processing time and it does not bother about other atmospheric changes

Sumber dari :

***https://users.ece.cmu.edu/~koopman/des_s99/real_time/***

Carnegie Mellon University
18-849b Dependable Embedded Systems
Spring 1998

Authors: Kanaka Juvva

## Introduction

Real-Time systems span several domains of computer science. They are defense and space systems, networked multimedia systems, embedded automative electronics etc. In a real-time system the correctness of the system behavior depends not only the logical results of the computations, but also on the physical instant at which these results are produced. A real-time system changes its state as a function of physical time, e.g., a chemical reaction continues to change its state even after its controlling computer system has stopped. Based on this a real-time system can be decomposed into a set of subsystems i.e., the controlled object, the real-time computer system and the human operator. A real-time computer system must react to stimuli from the controlled object (or the operator) within time intervals dictated by its environment. The instant at which a result is produced is called a deadline. If the result has utility even after the deadline has passed, the deadline is classified as soft, otherwise it is firm. If a catastrophe could result if a firm deadline is missed, the deadline is hard. Commands and Control systems, Air traffic control systems are examples for hard real-time systems. On-line transaction systems, airline reservation systems are soft real-time systems.

## Classification Of Real-Time Systems

Real-Time systems can be classified [Kopetz97] from different perspectives. The first two classifications, hard real-time versus soft real-time, and fail-safe versus fail-operational, depend on the characteristics of the application, i.e., on factors outside the computer system. The second three classifications, guaranteed-timeliness versus best-effort, resource-adequate versus resource-inadequate, and event-triggered versus time-triggered, depend on the design and implementation, i.e., on factors inside the computer system. However this paper focuses on the differences between hard and soft real-time classification.

## Hard Real-Time versus Soft Real-Time

Tabel 1 shows the major differences between hard and soft real-time systems. The response time requirements of hard real-time systems are in the order of milliseconds or less and can result in a catastrophe if not met. In contrast, the response time requirements of soft real-time systems are higher and not very stringent. In a hard real-time system, the peak-load performance must be predictable and should not violate the predefined deadlines. In a soft real-time system, a degraded operation in a rarely occurring peak load can be tolerated. A hard real-time system must remain synchronous with the state of the environment in all cases. On the otherhand soft real-time systems will slow down their response time if the load is very high. Hard real-time systems are often safety critical. Hard real-time systems have small data files and real-time databases. Temporal accuracy is often the concern here. Soft real-time systems for example, on-line reservation systems have larger databases and require long-term integrity of real-time systems. If an error occurs in a soft real-time system, the computation is rolled back to a previously established checkpoint to initiate a recovery action. In hard real-time systems, roll-back/recovery is of limited use.

## Real-Time Scheduling

A hard real-time system must execute a set of concurrent real-time tasks in a such a way that all time-critical tasks meet their specified deadlines. Every task needs computational and data resources to complete the job. The scheduling problem is concerned with the allocation of the resources to satisfy the timing constraints. Figure 2 given below represents a taxonomy of real-time scheduling algorithms.

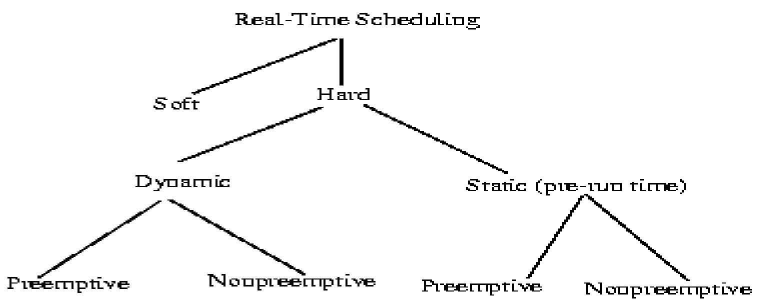

Fig. 2: Taxonomy of Real-Time Scheduling

Real-Time scheduling can be categorized into hard vs soft. Hard real-time scheduling can be used for soft real-time scheduling. Some of the research on QoS [ Klara95] addresses this problem in detail and is not covered here. The present paper focuses on scheduling algorithms for hard real-time.

Hard real-time scheduling can be broadly classifies into two types: static and dynamic. In static scheduling, the scheduling decisions are made at compile time. A run-time schedule is generated off-line based on the prior knowledge of task-set parameters, e.g., maximum execution times, precedence constraints, mutual exclusion constraints, and deadlines. So run-time overhead is small. More details on static scheduling can be found in [ Xu90]. On the otherhand, dynamic scheduling makes its scheduling decisions at run time, selecting one out of the current set of ready tasks. Dynamic schedulers are flexible and adaptive. But they can incur significant overheads because of run-time processing. Preemptive or nonpreemptive scheduling of tasks is possible with static and dynamic scheduling. In preemptive scheduling, the currently executing task will be preempted upon arrival of a higher priority task. In nonpreemptive scheduling, the currently executing task will not be preempted until completion.

## Dynamic Scheduling Algorithms

Schedulability test often used by dynamic schedulers to determine whether a given set of ready tasks can be scheduled to meet their deadlines. Different scheduling algorithms and their schedulability criteria is explained below.

### *Rate Monotonic Algorithm (RMA)*

Rate monotonic algorithm [ Lui94] is a dynamic preemptive algorithm based on static priorities. The rate monotonic algorithm assigns static priorities based on task periods. Here task period is the time after which the tasks repeats and inverse of period is task arrival rate. For example, a task with a period of 10ms repeats itself after every 10ms. The task with the shortest period gets the highest priority, and the task with the longest period gets the lowest static priority. At run time, the dispatcher selects the task with the highest priority for execution. According to RMA a set of periodic, independent task can be scheduled to meet their deadlines, if the sum of their utilization factors of the n tasks is given as below.

### *Ealriest Deadline-First (EDF) Algorithm:*

EDF algorithm is an optimal dynamic preemptive algorithm based on dynamic priorities. In this after any significant event, the task with the earliest deadline is assigned the highest dynamic priority. A significant event in a system can be blocking of a task, invocation of a task, completion of a task etc. The processor utilization can up to 100% with EDF, even when the task periods are not multiples of the smallest period. The dispatcher operates in the same way as the dispatcher for the rate monotonic algorithm.

### *The Priority Ceiling Protocol:*

The priority ceiling protocol [ Lui90] is used to schedule a set dependant periodic tasks that share resources protected by semaphores. The shared resources, e.g., common data structures are used for interprocess communication. The sharing of resources can lead to unbounded priority inversion. The priority ceiling protocols were developed to minimize the priority inversion and blocking time.

### Static Scheduling Algorithms

In static scheduling, scheduling decisions are made during compile time. This assumes parameters of all the tasks is known a priori and builds a schedule based on this. Once a schedule is made, it cannot be modified online. Static scheduling is generally not recommended for dynamic systems. Applications like process control can benefit from this scheduling, where sensor data rates of all tasks are known before hand. There are no explicit static scheduling techniques except that a schedule is made to meet the deadline of the given application under known system configuration. Most often there is no notion of priority in static scheduling. Based on task arriaval pattern a time line is built and embedded into the program and no change in schedules are possible during execution.

---

## Available tools, techniques, and metrics

Real-Time Operating Systems (RTOS) can be used to provide predictable services to the applications. RTOS provide the primitives real-time scheduling policies, inter process communication and run-time monitoring. There a number of RTOSs, e.g. RT-Mach, VxWORKS, Solaris, Lynx.

---

## Relationship to other topics

I/O

Real-Time systems interact with their environment by input/output subsystem. Sensors and actuators are the examples of i/o elements in real-time systems. On the otherhand i/o an important part of real-time systems.

Fault Tolerant Computing

Fault tolerance is important in safety-critical real-time systems because otherwise a single component failure can lead to a catastrophic systems failure.

Quality of Service

With the growth of Internet several multimedia applications like multimedia are merging with real-time systems. Scheduling in these systems is done to provide good quality of service. Some of the real-time systems research is being extended to QoS scheduling to multimedia applications.

Sumber dari:

http://www.mhhe.com/engcs/compsci/pressman/information/olc/RTdesign.html

## 1.1 System Considerations

Like any computer-based system, a real-time system must integrate hardware, software, human, and data base elements to properly achieve a set of functional and performance requirements. In Chapter 10 [SEPA, 5/e], we examined the allocation task for computer-based systems, indicating that the system engineer must allocate function and performance among the system elements. The problem for real-time systems is proper allocation. Real-time performance is often as important as function, yet allocation decisions that relate to performance are often difficult to make with assurance. Can a processing algorithm meet severe timing constraints, or should we build special hardware to do the job Can an off-the-shelf operating system meet our need for efficient interrupt handling, multi-tasking and communication, or should we built a custom executive Can specified hardware coupled with proposed software meet performance criteria These, and many other questions, must be answered by the real-time system engineer.

A comprehensive discussion of all elements of real time systems is beyond the scope of this book. Among a number of good sources of information are [SAV85], [ELL94], and [SEL94]. However, it is important that we understand each of the elements of a real-time system before focusing on software analysis and design issues.

Everett {EVE95] defines three characteristics that differentiate real-time software development from other software engineering efforts:

• *The design of real-time system is resource constrained.* The primary resource for a real-time system is time. It is essential to complete a defined task within a given number of CPU cycles. In addition, other system resources, such as memory size, may be traded against time to achieve system objectives.

• *Real-time systems are compact, yet complex.* Although a sophisticated real-time system may contain well over 1 million lines of code, the time critical portion of the software represents a very small percentage of the total. It is this small percentage of code that is typically the most complex (from an algorithmic point of view).

• *Real-time systems often work without the presence of a human user.* Therefore, real-time software must detect problems that lead to failure and automatically recover from these problems before damage to data and the controlled environment occurs.

In the section that follows, we examine some of the key attributes that differeniate real-time systems from other types of computer software.

## 1.2 Real-Time Systems

Real-time systems generate some action in response to external events. To accomplish this function, they perform high-speed data acquisition and control under severe time and reliability constraints. Because these constraints are so stringent, real-time systems are frequently dedicated to a single application.

Until recently, the major consumer of real-time systems was the military. Today, however, significant decreases in hardware costs make it possible for most companies to afford real-time systems (and products) for diverse applications that include process control, industrial automation, medical and scientific research, computer graphics, local and wide-area communications, aerospace systems, computer-aided testing, and a vast array of industrial instrumentation.

### 1.2.1 Integration and Performance Issues

Putting together a real-time system presents the system engineer with difficult hardware and software decisions. [The allocation issues associated with hardware for real-time systems are beyond the scope of this book (see [SAV85] for additional information)]. Once the software element has been allocated, detailed software requirements are established and a fundamental software design must be developed. Among many real-time design concerns are: coordination between the real-time tasks; processing of system interrupts; I/O handling to ensure that no data is lost; specifying the system's internal and external timing constraints, and ensuring the accuracy of its data base.

Each real-time design concern for software must be applied in the context of system *performance.* In most cases, the performance of a real-time system is measured as one or more time-related characteristics, but other measures such as fault-tolerance may also be used.

Some real-time systems are designed for applications in which only the response time or the data transfer rate is critical. Other real-time applications require optimization of both parameters under peak loading conditions. What's more, real-time systems must handle their peak loads while performing a number of simultaneous tasks.

Since the performance of a real-time system is determined primarily by the system response time and its data transfer rate, it is important to understand these two parameters. System *response time* is the time within which a system must detect an internal or external event and respond with an action. Often, event detection and response generation are simple. It is the processing of the information about the event to determine the appropriate response that may involve complex, time-consuming algorithms.

Among the key parameters that affect the response time are *context switching* and *interrupt latency.*Context switching involves the time and overhead to switch among tasks, and interrupt latency is the time lag before the switch is actually possible. Other parameters that affect response time are the speed of computation and of access to mass storage.

The *data transfer rate* indicates how fast serial or parallel, as well as analog or digital data must be moved into or out of the system. Hardware vendors often quote timing and capacity values for performance characteristics. However, hardware specifications for performance are usually measured in isolation and are often of little value in determining overall real-time system performance. Therefore, I/O device performance, bus latency, buffer size, disk performance, and a host of other factors, although important, are only part of the story of real-time system design.

Real-time systems are often required to process a continuous stream of incoming data. Design must assure that data are not missed. In addition, a real-time system must respond to events that are asynchronous. Therefore, the arrival sequence and data volume cannot be easily predicted in advance.

Although all software applications must be reliable, real-time systems make special demands on reliability, restart, and fault recovery. Because the real-world is being monitored and controlled, loss of monitoring or control (or both) is intolerable in many circumstances (e.g., an air traffic control system). Consequently, real-time systems contain restart and fault-recovery mechanisms and frequently have built-in redundancy to insure backup.

The need for reliability, however, has spurred an on-going debate about whether *on-line* systems, such as airline reservation systems and automatic bank tellers, also qualify as real-time. On one hand, such on-line systems must respond to external interrupts within prescribed response times on the order of one second. On the other hand, nothing catastrophic occurs if an on-line system fails to meet response requirements; instead, only system degradation results.

### 1.2.2 Interrupt Handling

One characteristic that serves to distinguish real-time systems from any other type is *interrupt handling.* A real-time system must respond to external stimulae–*interrupts*– in a time frame dictated by the external world. Because multiple stimulae (interrupts) are often present, priorities and priority interrupts must be established. In other words, the most important task must always be serviced within predefined time constraints regardless of other events.

Interrupt handling entails not only storing information so that the computer can correctly restart the interrupted task, but also avoiding deadlocks and endless loops. The overall approach to interrupt handling is illustrated in Figure 1.1. Normal processing flow is "interrupted" by an event that is detected by processor hardware. An *event* is any occurrence that requires immediate service and may be generated by either hardware or software. The state of the interrupted program is saved (i.e., all register contents, control blocks, etc. are saved) and control is passed to an interrupt service routine that branches to appropriate software for handling the interrupt. Upon completion of interrupt servicing, the state of the machine is restored and normal processing flow continues.

In many situations, interrupt servicing for one event may itself be interrupted by another, higher priority event. Interrupt priority levels (Figure 1.2) may be established. If a lower-priority process is accidentally allowed to interrupt a higher-priority one, it may be difficult to restart the processes in the right order and an endless loop may result.

### 1.2.3 Real-Time Data Bases

Like many data-processing systems, real-time systems often are coupled with a data base management function. However, *distributed data bases* would seem to be a preferred approach in real-time systems because multi-tasking is commonplace and data are often processed in parallel. If the data base is distributed, individual tasks can access their data faster and more reliably, and with fewer bottlenecks than with a centralized data base. The use of a distributed data base for real-time applications divides input/output "traffic" and shortens queues of tasks waiting for access to a data base. Moreover, a failure of one data base will rarely cause the failure of the entire system, if redundancy is built in.

The performance efficiencies achieved through the use of a distributed data base must be weighed against potential problems associated with data partitioning and replication. Although data redundancy improves response time by provided multiple information sources, replication requirements for distributed files also produces logistical and overhead problems, since all the file copies must be updated. In addition, the use of distributed data bases introduces the problem of *concurrency control.* Concurrency control involves synchronizing the data bases so that all copies have the correct, identical information free for access.

The conventional approach to concurrency control is based on what are known as *locking* and *time stamps.* At regular intervals, the following tasks are initiated: (1) the data base is "locked" so that concurrency control is assured; no I/O is permitted; (2) updating occurs as required; (3) the data base is unlocked; (4) files are validated to assure that all updates have been correctly made; (5) the completed update is acknowledged. All locking tasks are monitored by a master clock (i.e., time stamps). The delays involved in these procedures, as well as the problems of avoiding inconsistent updates and deadlock, mitigate against the widespread use of distributed data bases.

Some techniques, however, have been developed to speed updating and to solve the concurrency problem. One of these, called the *exclusive-writer protocol* maintains the consistency of replicated files by allowing only a single, exclusive writing task to update a file. It therefore eliminates the high overhead of locking or time stamp procedures.

### 1.2.4 Real-Time Operating Systems

Choosing a real-time operating system (RTOS) for a specific application is no easy chore. Some operating system classifications are possible, but most do not fit into neat categories with clear-cut advantages and disadvantages. Instead, there is considerable overlap in capabilities, target systems, and other features.

Some real-time operating systems are applicable to a broad range of system configurations, while others are geared to a particular board or even microprocessor, regardless of the surrounding electronic environment. RTOS achieve their capabilities through a combination of software features and (increasingly) a variety of micro-coded capabilities implemented in hardware.

Today, two broad classes of operating systems are used for real-time work: (1) dedicated RTOS designed exclusively for real-time applications and (2) general-purpose operating systems that have been enhanced to provide real-time capability. The use of a *real-time executive* makes real-time performance feasible for a general-purpose operating system. Behaving like application software, the executive performs a number of operating system functions–particularly those that affect real-time performance–faster and more efficiently than the general purpose operating system.

All operating systems must have a priority scheduling mechanism, but RTOS must provide a *priority mechanism* that allows high-priority interrupts to take precedence over less important ones. Moreover, because interrupts occur in response to asynchronous, nonrecurring events, they must be serviced without first taking time to swap in a program from disk storage. Consequently, to guarantee the required response time, a real-time operating system must have a mechanism for *memory locking*–that is, locking at least some programs in main memory so that swapping overhead is avoided.

To determine which kind of real-time operating system best matches an application, measures of RTOS quality can be defined and evaluated. Context switching time and interrupt latency, (discussed earlier) determine interrupt-handling capability, the most important aspect of a real-time system. Context switching time is the time the operating system takes to store the state of the computer and the contents of the registers so that it can return to a processing task after servicing the interrupt.

Interrupt latency, the maximum time lag before the system gets around to switching a task, occurs because in an operating system there are often non-re-entrant or critical processing paths that must be completed before an interrupt can be processed.

The length of these paths (the number of instructions) before the system can service an interrupt indicates the worst-case time lag. The worst case occurs if a high-priority interrupt is generated immediately after the system enters a critical path between an interrupt and interrupt service. If the time is too long, the system may miss an unrecoverable piece of data. It is important that the designer know the time lag so that the system can compensate for it.

Many operating systems perform multitasking [WOO90], or concurrent processing, another major requirement for real-time systems. But to be viable for real-time operation, the system overhead must be low in terms of switching time and memory space used.

### 1.2.5 Real-Time Languages

Because of the special requirements for performance and reliability demanded of real-time systems, the choice of a programming language is important. Many general purpose programming languages (e.g., C, FORTRAN, Modula-2) can be used effectively for real-time applications. However, a class of so-called "real-time languages" (e.g., Ada, Jovial, HAL/S, Chill, and others) is often used in specialized military and communications applications.

A combination of characteristics makes a real-time language different from a general-purpose language. These include the multitasking capability, constructs to directly implement real-time functions, and modern programming features that help ensure program correctness.

A programming language that directly supports multitasking is important because a real-time system must respond to asynchronous events occurring simultaneously. Although many RTOS provide multitasking capabilities, embedded real-time software often exists without an operating system. Instead, embedded applications are written in a language that provides sufficient run-time support for real-time program execution. Run-time support requires less memory than an operating system, and it can be tailored to an application, thus increasing performance.

A real time system that has been designed to accommodate multiple tasks must also accommodate intertask synchronization [KAI83]. A programming language that directly supports synchronization primitives such as SCHEDULE, SIGNAL, and WAIT greatly simplifies the translation from design to code. The SCHEDULE command schedules a process based on time or an event; SIGNAL and WAIT commands manipulate a special flag, called a *semaphore,* that enables concurrent tasks to be synchronized.

Finally, features that facilitate reliable programming are necessary because real-time programs are frequently large and complex. These features include modular programming, strongly enforced data typing, and a host of other control and data definition constructs.

### 1.2.6 Task Synchronization and Communication

A multi-tasking system must furnish a mechanism for the tasks to pass information to each other as well as to ensure their synchronization. For these functions, operating systems and languages with run-time support commonly use queuing semaphores, mailboxes, or message systems. Semaphores supply synchronization and signaling but contain no information. Messages are similar to semaphores except that they carry the associated information. Mailboxes, on the other hand, do not signal information but instead contain it.

*Queuing semaphores* are software primitives that help manage traffic. They provide a method of directing several queues–for example, queues of tasks waiting for resources, data-base access, and devices, as well as queues of the resources and devices. The semaphores coordinate (synchronize) the waiting tasks with whatever they are waiting for without letting tasks or resources interfere with each other.

In a real-time system, semaphores are commonly used to implement and manage *mailboxes.* Mailboxes are temporary storage places (also called a *message pool* or *buffer*) for messages sent from one process to another. One process produces a piece of information, puts it in the mailbox, and then signals a consuming process that there is a piece of information in the mailbox for it to use.

Some approaches to real-time operating systems or run-time support systems view mailboxes as the most efficient way to implement communications between processes. Some real-time operating systems furnish a place to send and receive pointers to mailbox data. This eliminates the need to transfer all of the data–thus saving time and overhead.

A third approach to communication and synchronization among processes is a message system. With a message system, one process sends a message to another. The latter is then automatically activated by the run-time support system or operating system to process the message. Such a system incurs overhead because it transfers the actual information, but it provides greater flexibility and ease of use.

## 1.3 Analysis and Simulation of Real-Time Systems

In the preceding section, we discussed a set of dynamic attributes that cannot be divorced from the functional requirements of a real-time system:

• interrupt handling and context switching
• response time
• data transfer rate and throughput
• resource allocation and priority handling
• task synchronization and intertask communication

Each of these performance attributes can be specified, but it is extremely difficult to verify whether system elements will achieve desired response, system resources will be sufficient to satisfy computational requirements, or processing algorithms will execute with sufficient speed.
The analysis of real time systems requires modeling and simulation that enables the system engineer to assess "timing and sizing" issues. Although a number of analysis techniques have been proposed in the literature (e.g., [LIU90], [WIL90] and [ZUC89]), it is fair to state that analytical approaches for the analysis and design of real-time systems are still in their early stages of development.

## 1.4 Real-Time Design

The design of real-time software must incorporate all of the fundamental concepts and principles (Chapter 13) associated with high quality software. In addition, real-time software poses a set of unique problems for the designer:

• representation of interrupts and context switching;
• concurrency as manifested by multi-tasking and multi-processing;
• intertask communication and synchronization;
• wide variations in data and communication rates;
• representation of timing constraints;
• asynchronous processing;
• necessary and unavoidable coupling with operating systems, hardware, and other external system elements.

Before considering some of these problems, it is worthwhile to address a set of specialized design principles that are particularly relevant during the design of real-time systems. Kurki-Suono [KUR93] discusses the design model for real-time ("reactive") software:

> All reasoning, whether formal or intuitive, is performed with some abstraction. Therefore, it is important to understand which kinds of properties are expressible in the abstraction in question. In connection

> with reactive systems, this is emphasized by the more stringent need for formal methods, and by the fact that no general consensus has been reached about the models that should be used. Rigorous formalisms for reactive systems range from process algebras and temporal logics to concrete state-based models and Petri nets, and different schools keep arguing about their relative merits.

He then defines a number of "modeling principles" that should be considered in the design of real-time software [KUR93]:

**Explicit atomicity.** It is necessary to define "atomic actions" explicitly as part of the real-time design model. An atomic action or event is a well constrained and limited function that can be executed by a single task or executed concurrently by several tasks. An atomic action is invoked only by those tasks ("participants") that require it and the results of its execution affect only those participants; no other parts of the system are affected.

**Interleaving.** Although processing can be concurrent, the history of some computation should be characterized in a way that can be obtained by a linear sequence of actions. Starting with an initial state, a first action is enabled and executed. As a result of this action, the state is modified and a second action occurs. Because several actions can occur in any given state, different results (histories) can be spawned from the same initial state. "This nondeterminism is essential in interleaved modeling of concurrency." [KUR93].

**Nonterminating histories and fairness.** The processing history of a reactive system is assumed to be infinite, By this we mean that processing continues indefinitely or "stutters" until some event causes it to continue processing. *Fairness* requirements prevent a system from stopping at some arbitrary point.

**Closed system principle.** A design model of a real-time system should encompass the software and the environment in which the software resides. "Actions can therefore be partitioned into those for which the system itself is responsible, and to those that are assumed to be executed by the environment." [KUR93]

**Structuring of state.** A real-time system can be modeled as a set of objects each of which has a state of its own.

The software engineer should consider each of the concepts noted above as the design of a real-time system evolves.