

## 3 Unit 3 - Real Time Software Design

### 3.1 WHAT IS REAL TIME?

Real time is a quantitative notion of time and is measured using a physical (real) clock. Whenever we quantify time using a physical clock, we deal with real time. An example use of this quantitative notion of time can be observed in a description of an automated chemical plant. Consider this: When the temperature of the chemical reaction chamber attains a certain predetermined temperature, say 250°C, the system automatically switches off the heater within a pre-determined time interval, say within 30 mSec. In this description of a part of the behaviour of a chemical plant, the time value that was referred to denotes the readings of some physical clock present in the plant automation system.

In contrast to real time, **logical time** (also known as virtual time) deals with a qualitative notion of time and is expressed using event ordering relations such as before, after, sometimes, eventually, precedes, succeeds, etc. While dealing with logical time, time readings from a physical clock are not necessary for ordering the events. As an example, consider the following part of the behaviour of a library automation software used to automate the bookkeeping activities of a college library: "After a *query book* command is given by the user, details of all matching books are displayed by the software." In this example, the events "issue of query book command" and "display of results" are logically ordered in terms of which events follow the other. But, no quantitative expression of time was required. Clearly, this example behaviour is devoid of any real-time considerations. We are now in a position to define what is a real-time system:

A system is called a **real-time system**, when we need quantitative expression of time (i.e., real time) to describe the behaviour of the system.

Remember that in this definition of a real-time system, it is implicit that all quantitative time measurements are carried out using a physical clock. A chemical plant, whose part behaviour description is—when temperature of the reaction chamber attains certain predetermined temperature value, say 250°C, the system automatically switches off the heater within say 30 mSec—is clearly a real-time system. It should, however, be remembered that both the computer system and the controlled system (environment) use the same time scale. So far our examples were restricted to the description of partial behaviour of systems. The complete behaviour of a system can be described by listing its response to various external stimuli. It may be noted that all the clauses in the description of the behaviour of a real-time system need not involve quantitative measures of time. That is, large parts of a description of the behaviour of a system may not have any quantitative expressions of time at all, and still qualify as a real-time system. Any system whose behaviour can completely be described without using any quantitative expression of time is not a real-time system.

#### DEFINITION

### 3.2 real-time operating system (RTOS)

A real-time operating system (RTOS) is an operating system that guarantees a certain capability within a specified time constraint. For example, an operating system might be designed to ensure that a certain object was available for a robot on an assembly line. In what is usually called a "hard" real-time operating system, if the calculation could not be performed for making the object available at the designated time, the operating system would terminate with a failure. In a "soft" real-time operating system, the assembly line would continue to function but the production output might be lower as objects failed to appear at their designated time, causing the robot to be temporarily unproductive. Some real-time operating systems are created for a special application

and others are more general purpose. Some existing general purpose operating systems claim to be a real-time operating systems. To some extent, almost any general purpose operating system such as Microsoft's [Windows 2000](#) or IBM's [OS/390](#) can be evaluated for its real-time operating system qualities. That is, even if an operating system doesn't qualify, it may have characteristics that enable it to be considered as a solution to a particular real-time application problem.

In general, real-time operating systems are said to require:

- [Multitasking](#)
- Process [threads](#) that can be prioritized
- A sufficient number of [interrupt](#) levels

Real-time operating systems are often required in small embedded operating systems that are packaged as part of microdevices. Some [kernels](#) can be considered to meet the requirements of a real-time operating system. However, since other components, such as [device drivers](#), are also usually needed for a particular solution, a real-time operating system is usually larger than just the kernel.

### Real-time computing

In [computer science](#), **real-time computing (RTC)**, or **reactive computing**, is the study of [hardware](#) and [software](#) systems that are subject to a "real-time constraint"— e.g. operational deadlines from event to system response. Real-time programs must guarantee response within strict time constraints, often referred to as "deadlines".<sup>[1]</sup> Real-time responses are often understood to be in the order of milliseconds, and sometimes microseconds. Conversely, a system without real-time facilities, cannot *guarantee* a response within any timeframe (regardless of *actual* or *expected* response times).

The use of this word should not be confused with the two other legitimate uses of 'real-time'. In the domain of simulations, the term means that the simulation's clock runs as fast as a real clock. In the processing and enterprise systems domains, the term is used to mean 'without perceivable delay'.

Real-time software may use one or more of the following: [synchronous programming languages](#), [real-time operating systems](#), and real-time networks, each of which provide essential frameworks on which to build a real-time software application.

A real-time system may be one where its application can be considered (within context) to be [mission critical](#). The [anti-lock brakes](#) on a car are a simple example of a real-time computing system — the real-time constraint in this system is the time in which the brakes must be released to prevent the wheel from locking. Real-time computations can be said to have *failed* if they are not completed before their deadline, where their deadline is relative to an event. A real-time deadline must be met, regardless of [system load](#).

Real-time systems, as well as their deadlines, are classified by the consequence of missing a deadline.

#### Hard

Missing a deadline is a total system failure.

#### Firm

Infrequent deadline misses are tolerable, but may degrade the system's quality of service. The usefulness of a result is zero after its deadline.

#### Soft

The usefulness of a result degrades after its deadline, thereby degrading the system's quality of service.

Thus, the goal of a **hard real-time system** is to ensure that all deadlines are met, but for **soft real-time systems** the goal becomes meeting a certain subset of deadlines in order to optimize some application specific criteria. The particular criteria optimized depends on the application, but some typical examples include maximizing the number of deadlines met, minimizing the lateness of tasks and maximizing the number of high priority tasks meeting their deadlines.

Hard real-time systems are used when it is imperative that an event be reacted to within a strict deadline. Such strong guarantees are required of systems for which not reacting in a certain interval of time would cause great loss in some manner, especially damaging the surroundings physically or threatening human lives (although the strict definition is simply that missing the deadline constitutes failure of the system). For example, a [car engine](#) control system is a hard real-time system because a delayed signal may cause engine failure or damage. Other examples of hard real-time embedded systems include medical systems such as heart [pacemakers](#) and industrial process controllers. Hard real-time systems are typically found interacting at a low level with physical hardware, in [embedded systems](#). Early video game systems such as the [Atari 2600](#) and [Cinematronics](#) vector graphics had hard real-time requirements because of the nature of the graphics and timing hardware.

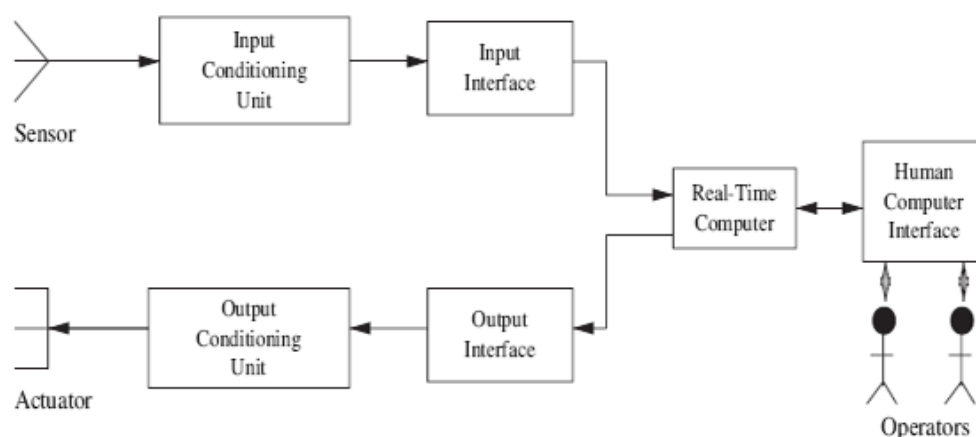
In the context of [multitasking](#) systems the scheduling policy is normally priority driven ([pre-emptive](#) schedulers). Other scheduling algorithms include [Earliest Deadline First](#), which, ignoring the overhead of [context switching](#), is sufficient for system loads of less than 100%.<sup>[2]</sup> New overlay scheduling systems, such as an [Adaptive Partition Scheduler](#) assist in managing large systems with a mixture of hard real-time and non real-time applications.

Soft real-time systems are typically used where there is some issue of concurrent access and the need to keep a number of connected systems up to date with changing situations; for example software that maintains and updates the flight plans for commercial [airliners](#). The flight plans must be kept reasonably current but can operate to a latency of seconds. Live audio-video systems are also usually soft real-time; violation of constraints results in degraded quality, but the system can continue to operate.

### 3.3 A BASIC MODEL OF A REAL-TIME SYSTEM

We have already pointed out that this book confines itself to the software issues in real-time systems. However, in order to be able to see the software issues in a proper perspective, we need to have a basic conceptual understanding of the underlying hardware. Therefore, in this section we try to develop a broad understanding of the high-level issues of the underlying hardware in a real-time system. For a more detailed study of the underlying hardware issues, we refer the reader to [8]. **Figure 1.3** shows a simple model of a real-time system in terms of its important functional blocks. Unless otherwise mentioned, all our subsequent discussions would implicitly assume such a model. Observe that in **Fig. 1.3**, the sensors are interfaced with the input conditioning block, which, in turn, is connected to the input interface. The output interface, output conditioning, and the actuator are interfaced in a complementary manner. In the following, we briefly describe the roles of the different functional blocks of a real-time system:

**FIGURE 1.3. A Model of a Real-Time System**



**Sensor.** A sensor converts some physical characteristic of its environment into electrical signals. An example of a sensor is a photo-voltaic cell which converts light energy into electrical energy. A wide variety of temperature and pressure sensors are also used. Typically, a temperature sensor operates on the principle of a **thermocouple**. Temperature sensors based on many other physical principles also exist. For example, one type of temperature sensor employs the principle of variation of electrical resistance with temperature (called a *varistor*). A pressure sensor typically operates on the *piezoelectricity* principle. Pressure sensors based on other physical principles also exist.

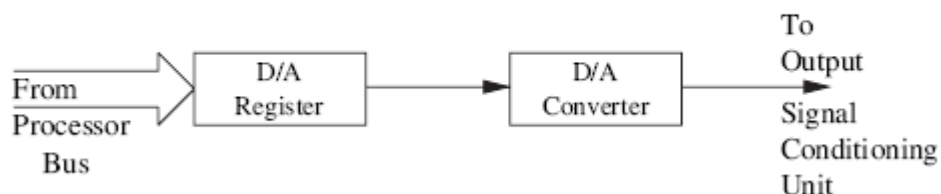
Actuator. An actuator is any device that takes its inputs from the output interface of a computer and converts these electrical signals into some physical actions on its environment. The physical actions may be in the form of motion, change of thermal, electrical, pneumatic, or physical characteristics of some objects. A popular actuator is a motor. Heaters are also very commonly used. Besides, several hydraulic and pneumatic actuators are also popular.

Signal Conditioning Units. The electrical signals produced by a computer can rarely be used to directly drive an actuator. The computer signals usually need conditioning before they can be used by the actuator. This is termed *output conditioning*. Similarly, input conditioning is required to be carried out on sensor signals before they can be accepted by the computer. For example, analog signals generated by a photo-voltaic cell are normally in the millivolts range and need to be conditioned before they can be processed by a computer. The following are some important types of conditioning carried out on raw signals generated by sensors and digital signals generated by computers:

1. **Voltage Amplification:** Voltage amplification is normally required to be carried out to match the full-scale sensor voltage output with the full-scale voltage input to the interface of a computer. For example, a sensor might produce voltage in the millivolts range, whereas the input interface of a computer may require the input signal level to be of the order of a volt.
2. **Voltage Level Shifting:** Voltage level shifting is often required to align the voltage level generated by a sensor with that acceptable to the computer. For example, a sensor may produce voltage in the range  $-0.5$  to  $+0.5$  volt, whereas the input interface of the computer may accept voltage only in the range of 0 to 1 volt. In this case, the sensor voltage must undergo level shifting before it can be used by the computer.
3. **Frequency Range Shifting and Filtering:** Frequency range shifting is often used to reduce the noise components in a signal. Various types of noise occur in narrow bands and the signal must be shifted from the noise bands so that noise can be filtered out.
4. **Signal Mode Conversion:** A type of signal mode conversion that is frequently carried out during signal conditioning involves changing direct current into alternating current and vice versa. Another type of signal mode conversion frequently used is conversion of analog signals to a constant amplitude pulse train such that the pulse rate or pulse width is proportional to the voltage level. Conversion of analog signals to a pulse train is often necessary for input to systems such as **transformer coupled circuits** that do not pass direct current.

Interface Unit. Normally, commands from the CPU are delivered to the actuator through an output interface. An output interface converts the stored voltage into analog form and then outputs this to the actuator circuitry. This would require the value generated to be written on a register (see [Fig. 1.4](#)). In order to produce an analog output, in an output interface, the CPU selects a data register of the output interface and writes the necessary data to it. The two main functional blocks of an output interface are shown in [Fig. 1.4](#). The interface takes care of the buffering and the handshake control aspects. Analog to digital conversion is frequently deployed in an input interface. Similarly, digital to analog conversion is frequently used in an output interface.

**FIGURE 1.4. An Output Interface**



In the following, we discuss the important steps of Analog to Digital Signal Conversion (ADC).

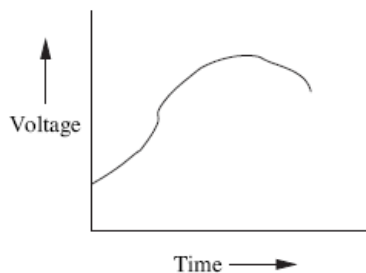
Analog to Digital Conversion. Digital computers can not process analog signals. Therefore, analog signals need to be converted to digital form using a circuitry whose block diagram is shown in [Fig.](#)

**1.7.** Using that block diagram, analog signals are normally converted to digital form through the following two main steps:

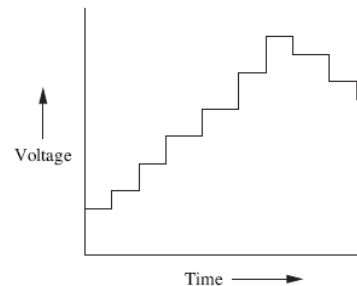
- Sample the analog signal (shown in [Fig. 1.5](#)) at regular intervals. This sampling can be done by a capacitor circuitry that stores the voltage levels. The stored voltage level can be discretized. After sampling the analog signal (shown in [Fig. 1.5](#)), a step waveform as shown in [Fig. 1.6](#) is obtained.
- Convert the stored value to a binary number by using an ADC as shown in [Fig. 1.7](#), and store the digital value in a register.

Digital to analog conversion can be carried out through a complementary set of operations. We leave it as an exercise to the reader to figure out the details of the circuitry that can perform the Digital to Analog Conversion (DAC).

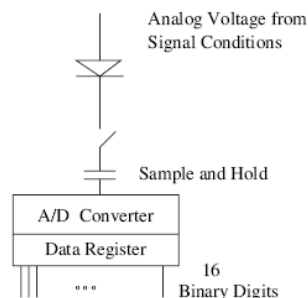
**FIGURE 1.5. Continuous Analog Voltage**



**FIGURE 1.6. Analog Voltage Converted to Discrete Form**



**FIGURE 1.7. Conversion of an Analog Signal to a 16-bit Binary Number**



## 3.4 Real-Time Simulation

### On this page...

[What Is Real-Time Simulation?](#)

[Requirements for Real-Time Simulation](#)

[Simulating Physical Models in Real Time](#)

[Preparing a Model for Real-Time Simulation](#)

[Troubleshooting Real-Time Simulation Problems](#)

### What Is Real-Time Simulation?

Real-time simulation of an engineering system becomes possible when you replace physical devices with virtual devices. This replacement reduces costs and improves the quality of physical and control systems, including their software, by enabling more complete testing of the entire system. It also enables continuous testing, without interruption and under possibly dangerous conditions. Real-time simulation allows you to test even when you have no prototypes.

Real-time simulation becomes a necessity if you want to simulate a system realistically responding to its environment. Such realistic simulation means that the inputs and outputs in the virtual world of simulation must be read or updated synchronously with the real world. When the simulation clock reaches a certain time in real-time simulation, the same amount of time must have passed in the real world.

### **Using Real-Time Simulation to Test Virtual Controllers and Systems**

In desktop simulation, you use models to develop and test control and signal processing algorithms. Once the designs are complete and you have converted these algorithms to embedded code, you must test that code as well as the actual controller. If the model is capable of running in real time, you can use the model created in the design phase to test the embedded code and processor, instead of connecting it directly to a hardware prototype. Such real-to-virtual substitution, simulating in real time, is referred to as *hardware-in-the-loop* (HIL) testing.

#### **Example**

Systems with a human in the simulation loop require real-time simulation. For example, flight simulators that train pilots require real-time simulation of the plane, its control system, the weather, and other environmental conditions.

### **Requirements for Real-Time Simulation**

Configuring a model and a numerical integrator to simulate in real time is often more challenging than ordinary simulation. You simulate with a more restrictive version of the universal computational tradeoff of accuracy versus speed.

The simulation execution time per time step must be consistently short enough to permit any other tasks that the simulation environment must perform, such as reading sensor input or generating output actuator signals. This requirement must be satisfied even if the simulation changes its qualitative character: the system stiffness might change, and discrete components can switch states. Such changes occasionally require more computations to achieve an accurate result.

### **Bounding and Stabilizing Execution Time with Fixed-Step Solvers and Fixed-Cost Simulation**

When real-time simulation is the goal, the execution time per simulation time step must be bounded. Variable-step solvers, which are often used in desktop simulation, take smaller steps to accurately capture events that occur during the simulation. But you cannot vary the step size in a real-time simulation. Instead, you must

- Choose a fixed-step solver that can capture the system dynamics accurately and minimize the amount of computation required per time step, without changing the step size. If the system states are all discrete, the fixed-step solver can be discrete as well.

If you choose a small enough step size, most fixed-step solvers produce the same simulation results as a variable-step solver. However, different fixed-step solvers (implicit/explicit, lower/higher order, and so on) require different step sizes to produce accurate results. They also require different amounts of computation per time step.

- Choose a fixed step size large enough to permit stable real-time simulation. The step size must not be so large that the simulation results are inaccurate, but not so small that real-time simulation is impossible.

You often need trial and error to find the right combination of settings that satisfy both criteria.

Real-time simulation requires not only bounding the execution time, but fixing it to a stable value. This requires a fixed-cost simulation method. For more information, see [Customizing Solvers for Physical Models](#).

### **Simulating Physical Models in Real Time**

Achieving real-time simulation with any Simscape™ model includes:

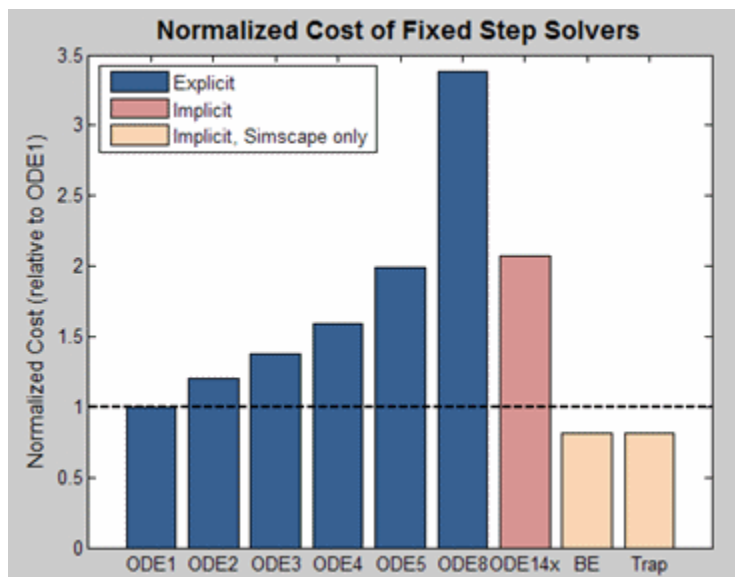
- Enabling simulation with fixed-step, fixed-cost solvers



- Converting the model with [Simulink® Coder™](#) to code for a particular computer hardware target
  - Testing real-time simulation on PC-compatible hardware with [xPC Target™](#), if desired
- For more information, see [Code Generation](#).

Preparation for real-time simulation requires particular choices and adjustment of Simulink variable-step solvers. Actual real-time simulation requires Simulink fixed-step solvers. Certain Simscape features enable and enhance real-time simulation of physical systems with Simulink fixed-step solvers, both explicit and implicit. These features include fixed-cost algorithms and local solvers, with the trapezoidal rule or backward Euler method. See [Customizing Solvers for Physical Models](#).

This figure plots the normalized computational cost of all fixed-step solvers available for Simscape models, obtained for a nonlinear model example with one physical network. For comparison, the step size was kept the same, with similar settings for the total number of solver iterations.



### Preparing a Model for Real-Time Simulation

To move from desktop to real-time simulation on your real-time hardware target, adjust the following simulation properties until the simulation can execute in real time and deliver results close to the results from desktop simulation:

- Solver choice
- Number of solver iterations
- Simulation time step size
- Model size and fidelity

Follow these high-level tasks to prepare a model for real-time simulation. Each task is also a link to specific instructions for that part of the procedure.

1. [Simulate and Converge with Variable-Step Solver](#)
2. [Check Variable Time Steps for Optimal Step Size](#)
3. [Simulate with Fixed-Cost Solver and Compare to Variable-Step Simulation](#)
4. [Adjust Step Size and Iterations to Approximate Variable-Step Simulation Results](#)
5. [Attempt to Simulate in Real Time](#)
6. [Respond to Real-Time Simulation Failures](#)

### Simulate and Converge with Variable-Step Solver

The first task is to obtain a converged set of results with a variable-step solver.

To ensure that the results obtained with the fixed-step solver are accurate, you need a set of reference results. You can obtain these by simulating the system with a variable-step solver.

Ensure that the results converge by tightening the error tolerances until the simulation results do not change significantly.

### Check Variable Time Steps for Optimal Step Size

The second task is to examine the time step sizes during the desktop simulation and determine if the model is likely to run with a large enough step size to permit real-time simulation.

A variable-step solver varies the step size to keep the solution within error tolerances and to react to [zero crossing events](#). If the solver abruptly reduces the step size to a small value (for example,  $1e-15$  s), the solver is trying to accurately identify a zero crossing event. A fixed-step solver might have trouble capturing these events at a step size large enough to permit real-time simulation.

Analysis of these particular variable time steps provides an estimate of a step size that can be used to run the simulation. Modifying or eliminating the effects are causing these events makes it easier to simulate the system with a fixed-step solver at a reasonably large step size and produce results comparable to the variable-step simulation. See [Troubleshooting Real-Time Simulation Problems](#).

### Simulate with Fixed-Cost Solver and Compare to Variable-Step Simulation

The third task is to simulate the system with a fixed-step, fixed-cost solver and compare these results to the reference results from the variable-step simulation.

**Limiting Per-Step Solver Iterations.** Simulating physical systems often requires multiple iterations per time step to converge on a solution. To perform a fixed-cost simulation, you must limit these iterations. In each physical network [Solver Configuration](#) block, select the **Use fixed-cost runtime consistency iterations** check box and enter the number of allowed iterations.

**Switching to Local Solvers.** You can further minimize the computations done per time step by choosing a local solver on each physical network in the model. To switch to a local solver in a physical network, open the Solver Configuration block of that network and select **Use local solver**. By using this option, you can use an implicit fixed-step solver only on the stiff portions of the model and an explicit fixed-step solver on the remainder of the model. This minimizes the computations done per time step, making it more likely that the model can run in real time.

### Adjust Step Size and Iterations to Approximate Variable-Step Simulation Results

The fourth task is to reduce the step size and adjust the number of nonlinear iterations, in order to produce results that are sufficiently close to the reference results from variable-step simulation. The step size must still be large enough for a safety margin to prevent an execution overrun.

During each time step, the real-time simulation must calculate the result for the next time step (simulation execution), and read inputs and write outputs (I/O processing and other tasks). If these actions take less time than the specified time step, the processor remains idle during the remainder of the step. Choosing a computationally more intensive solver, increasing the number of nonlinear iterations, or reducing the step size both increases the simulation accuracy and reduces the amount of idle time, raising the risk that the simulation cannot run in real time. Adjusting these settings in the opposite way increases the amount of idle time but reduce accuracy.

Estimating the budget for the execution time helps ensure that you choose a feasible combination of settings. If you know the amount of time spent processing inputs and outputs and performing other actions, as well as the percentage of idle time that you want, the amount of time available for simulation execution can be calculated as follows:

Simulation Execution Time Budget =

Step Size – [I/O Processing Time + (Desired Percentage of Idle Time)·(Step Size)]



**Estimating Real-Time Execution Time.** You can use the desktop simulation speed to estimate the execution time on a real-time hardware target. Many factors affect the real-time target execution time, so that comparing processor speeds might not be sufficient. A better method is to measure the execution time of desktop simulation and then to determine the average execution time per time step on the real-time target for a particular model. Knowing how these execution times compare for one model means that you can estimate execution time on the real-time target from the desktop simulation execution time when you test other models.

#### **Attempt to Simulate in Real Time**

The fifth task is to use the selected solver, the number of nonlinear iterations, and the step size to simulate on the real-time target and to verify if the simulation can run in real time.

If the simulation does not run in real time on the target hardware, the model might not be real-time capable.

#### **Respond to Real-Time Simulation Failures**

If the simulation does not run in real time on the selected real-time target, perform a sixth, contingent task, described in [Troubleshooting Real-Time Simulation Problems](#).

## **Troubleshooting Real-Time Simulation Problems**

If the simulation does not run in real time on the real-time platform, or if the simulation performance is unacceptable, you should determine the causes and find an appropriate solution. The combination of effects captured in the model and the speed of the real-time platform might make it impossible to find solver settings that permit it to run in real time. Consider the following options to make it real-time capable.

Once you modify your model, return to the third, fourth, and fifth tasks of [Preparing a Model for Real-Time Simulation](#) to identify and implement the appropriate settings to enable real-time simulation.

#### **Speeding Up Real-Time Execution**

You can speed up the real-time simulation by using a faster real-time target computer.

Alternatively, you can achieve the same goal by determining new model settings that permit a larger step size or reduce the execution time (for example, by reducing the number of nonlinear iterations).

#### **Simulating Parts of the System in Parallel**

If possible, configure the model to evaluate multiple physical networks in parallel. You can do this if the networks are not dependent upon one another. You need experience and experimentation with your model, the generated code, and the real-time target to make effective use of this option.

#### **Eliminating Effects That Require Intensive Computation**

Certain effects in your model can prevent real-time simulation. Such effects include instantaneous events and rapid changes in parts of the system with very small time constants. Identify and modify or remove these elements before searching again for a combination of solver settings and step size that permits real-time simulation.

**Identifying Elements Causing Rapid or Instantaneous Changes.** Watch for certain system elements becoming excited to high frequencies. Examine the system eigenmodes to isolate which system states have the highest frequency. Mapping those states to individual components often points to the source of the problem. Because you can only do this at a particular operating point, choose an operating point corresponding to simulation times in the

variable-step simulation that had small step sizes. At such simulation times, the variable-step solver is struggling to simulate a rapid change.

With scripts written in MATLAB®, you can interrogate the model, identify these components quickly, and narrow the search for the effects that you need to modify. You can automate and extend these searches to other models with tools like the [Simulink Model Advisor](#). The troublesome components that you need to locate include:

- Elements that create events and change the solution nearly instantaneously. A fixed-step solver might not be able to step over such rapid changes and find the right solution on the other side of the event. If it fails to find the solution, the solver may become unstable. Examples of elements that create these kinds of events include:
  - Hard stops or backlash
  - Stick-slip friction
  - Switches or clutches
- Elements with very small time constants. The dynamics of these elements require a small step size so that a fixed-step solver can accurately simulate them, perhaps too small for real-time simulation. Examples of systems with a small time constant include:
  - Small masses attached to stiff springs with minimal damping
  - Electrical circuits with small capacitance and inductance and low resistance
  - Hydraulic circuits with small compressible volumes

**Modifying or Removing Elements Causing Rapid or Instantaneous Changes.** Once you have identified these elements, change or eliminate them by:

- Replacing nonlinear components with linearized versions
- Replacing complex equations with lookup tables for their solution
- Replacing complicated components with simplified models by using system identification theory on their input and output data
- Smoothing discontinuous functions (step changes) by using filters, delays, and other techniques.

## What are Real-Time Systems?

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

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

A real-time system must demonstrate the following features:

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

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

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

## **Real-Time System Application Domains**

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

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

## 3.5 Real-time programming

### 3.5.1 Basic concepts

- Hard-real time systems may have to be programmed in assembly language to

ensure that deadlines are met.

- Languages such as C allow efficient programs to be written but do not have

constructs to support concurrency or shared resource management.

#### Java as a real-time language

- ❖ **Java supports lightweight concurrency (threads and synchronized methods) and can be used for some soft real-time systems.**
- ❖ **Java 2.0 is not suitable for hard RT programming but real-time versions of Java are now available that address problems such as**
  - **Not possible to specify thread execution time;**
  - **Different timing in different virtual machines;**
  - **Uncontrollable garbage collection;**
  - **Not possible to discover queue sizes for shared**
  - **resources;**
  - **Not possible to access system hardware;**
  - **Not possible to do space or timing analysis**

## 3.6 System design

- Design both the hardware and the software associated with system. Partition functions to either hardware or software.
- Design decisions should be made on the basis of non-functional system requirements.
- Hardware delivers better performance but potentially longer development and less scope for change.
- Identify the stimuli to be processed and the required responses to these stimuli.
- For each stimulus and response, identify the timing constraints.
- Aggregate the stimulus and response processing into concurrent processes. A process may be associated with each class of stimulus and response.
- Design algorithms to process each class of stimulus and response. These must meet the given timing requirements.

- Design a scheduling system which will ensure that processes are started in time to meet their deadlines.
- Integrate using a real-time operating system.

### **3.7 Timing constraints**

- May require extensive simulation and experiment to ensure that these are met by the system.
- May mean that certain design strategies such as object-oriented design cannot be used because of the additional overhead involved.
- May mean that low-level programming language features have to be used for performance reasons.