# Chapter 9
# Real-time facilities



In Chapter 1, it was noted that a language for programming embedded systems requires facilities for real-time control. Indeed, the term 'real-time' has been used as a synonym for this class of system. Given the importance of time in many embedded systems, it may appear strange that consideration of this topic has been postponed until Chapter 9. Facilities for real-time control are, however, generally built upon the concurrency model within the language, and it is therefore necessary to have covered this area first.

The introduction of the notion of time into a programming language can best be described in terms of three largely independent topics.

(1) Interfacing with 'Time'; for example, accessing clocks so that the passage of time can be measured, delaying tasks until some future time, and programming timeouts so that the non-occurrence of some event can be recognized and dealt with.
(2) Representing timing requirements; for example, specifying rates of execution and deadlines.
(3) Satisfying timing requirements.

This chapter is largely concerned with the first two of these topics, although it will commence with some discussion of the notion of time itself. Chapter 11 considers ways of implementing systems such that the worst-case temporal behaviour can be predicted, and hence timing requirements ratified.

## 9.1 The notion of time

Our everyday experiences are so intrinsically linked to notions of past, present and future that it is surprising that the question 'What is time?' is still largely unresolved.

Philosophers, mathematicians, physicists and, more recently, engineers have all studied 'time' in minute detail, but there is still no consensus on a definitive theory of time. As St Augustine stated:

> What, then, is time? If no one asks me, I know what it is. If I wish to explain it to him who asks me, I do not know.

A key question in the philosophical discussions of time can be stated succinctly as 'do we exist in time, or is time part of our existence?' The two mainstream schools of thought are Reductionism and Platonism. All agree that human (and biological) history is made up of events, and that these events are ordered. Platonists believe that time is a fundamental property of nature; it is continuous, non-ending and non-beginning, 'and possesses these properties as a matter of necessity'. Our notion of time is derived from a mapping of historical events onto this external time reference.

Reductionists, by comparison, do without this external reference. Historical time, as made up of historical events, is the only meaningful notion of time. By assuming that certain events are 'regular' (for example, sunrise, winter solstice, vibrations of atoms, and so on), they can invent a useful time reference that enables the passage of time to be measured. But this time reference is a construction, not a given.

A consequence of the Reductionist view is that time cannot progress without change occurring. If the universe started with a 'Big Bang' then this represents the very first historical event and hence time itself started with 'space' at this first epoch. For Platonists, the Big Bang is just one event on an unbounded time line.

Over large distances, Einstein showed that relativity effects impinge not only on time directly but also on the temporal ordering of events. Within the special theory of relativity, the observer of events imposes a frame of references. One observer may place event A before event B; another observer (in a different frame of reference) may reverse the order. Due to such difficulties with temporal ordering, Einstein introduced **causal ordering**. Event A may cause event B if all possible observers see event A occurring first. Another way of defining such causality is to postulate the existence of a signal that travels, at a speed no greater than the speed of light, from event A to event B.

These different themes of time are well illustrated by the notion of simultaneous events. To Platonists, the events are simultaneous if they occur 'at the same time'. For Reductionists, simultaneous events 'happen together'. With Special Relativity, simultaneous events are those for which a causal relationship does not exist.

From a mathematical point of view, there are many different topologies for time. The most common one comes from equating the passage of time with a 'real' line. Hence time is linear, transitive, irreflective and dense:

- linearity: $\forall x, y : x < y \; or \; y < x \; or \; x = y$
- transitivity: $\forall x, y, z : (x < y \; and \; y < z) \Rightarrow x < z$
- irreflexivity: $\forall x : not \; (x < x)$
- density: $\forall x, y : x < y \Rightarrow \exists z : (x < z < y)$

The engineering perspective can largely ignore the philosopher's issue of time. An embedded real-time computer system needs to coordinate its execution with the 'time'

of its environment. The term 'real' is used to draw a distinction with the computer's time. It is real because it is external. Whether this external frame of reference is a Reductionist's construction or an approximative for the Platonists' 'absolute' time frame is not significant. Moreover, for most applications, relativistic effects can also be ignored. In terms of the mathematical topology of real-time systems, there are conflicting opinions as to whether time should be viewed as dense or discrete. Because computers work in discrete time, there is some advantage in constructing computational models based upon discrete time. The significant experience that other branches of engineering have in using, to good effect, dense time models argues the other way. Attempts to integrate both approaches has merely led to a third approach – hybrid systems.

If, by general consensus, a series of events is deemed to be regular then it is possible to define a standard measurement of time. Many such standards have existed in the past. Table 9.1 gives a brief description of some of the more significant ones. This description is taken from Hoogeboom and Halang (1992).

## 9.2 Access to a clock

If a program is going to interact in any meaningful way with the time frame of its environment then it must have access to some method of 'telling the time' or, at least, have some way of measuring the passage of time. This can be done in two distinct ways:

- by having direct access to the environment's time frame;
- by using an internal hardware clock that gives an adequate approximation to the passage of time in the environment.

The first approach is becoming more common and can be achieved in a number of ways. The simplest is for the environment to supply a regular interrupt that is clocked internally. Alternatively the system can be fitted with radio receivers and use one of the international time signals. Universal Coordinated Time (UTC) signals are broadcast by land-based radio stations (on shortwave frequencies) and satellites. For example, GPS (Global Positioning System) incorporates a time signal. Radio signals typically have an accuracy of 0.1–10 milliseconds; GPS provides a much better service with an accuracy of about 1 microsecond. If the system is linked to its environment via a telephone line or a network then these may also provide a time service (with an accuracy of perhaps a few milliseconds). With the Internet, NTP (Network Time Protocol) does indeed provide such a service.

Internal hardware clocks are devices that count the number of oscillations that occur in a quartz crystal. They typically divide this count by a fixed number and store the result in a register (counter) that the system's software can access. Inevitably this 'time' count is not always in perfect synchronization with the external time reference. The error is called **clock drift**. Clock drift may occur due, for example, to temperature changes within the system. A standard quartz crystal may drift by perhaps $10^{-6}$ seconds per second, i.e. one second in 11.6 days. A high-precision clock may have drift values of $10^{-7}$ or $10^{-8}$.

In distributed systems there is not only the drift between internal and external time that must be allowed for but also the **skew** between any of the clocks within the systems.

| Name | Description | Note |
|---|---|---|
| True solar day | Time between two successive culminations (highest point of the Sun) | Varies through the year by 15 minutes (approx.) |
| Temporal hour | One-twelfth part of the time between sunrise and sunset | Varies considerably through the year |
| Universal Time (UT0) | Mean solar time at Greenwich meridian | Defined in 1884 |
| Second (1) | 1/86 400 of a mean solar day | |
| Second (2) | 1/31 566 925.9747 of the tropical year for 1900 | Ephemeris Time defined in 1955 |
| UT1 | Correction to UTO because of polar motion | |
| UT2 | Correction of UT1 because of variation in the speed of rotation of the Earth | |
| Second (3) | Duration of 9 192 631 770 periods of the radiation corresponding to the transition between two hyperfine levels of the ground state of the Caesium 133 atom | Accuracy of current Caesium atomic clocks deemed to be one part in $10^{13}$ (that is, one clock error per 300 000 years) |
| International Atomic Time (IAT) | Based upon Caesium atomic clock | |
| Universal Coordinated Time (UTC) | An IAT clock synchronized to UT2 by the addition of occasional leap ticks | Maximum difference between UT2 (which is based on astronomical measurement) and UTC (which is based upon atomic measurement) is kept to below 0.5 seconds |

**Table 9.1** Time standards.

If each node in the distributed system has its own clock then they will not provide a perfect source of time. If a *global time* service is needed then some form of clock synchronization protocol is needed. Many such protocols exist; they differ in how they deal with node and network failure. Further details on these protocols can be found in the literature on distributed systems – see for example Coulouris et al. (2005).

From the programmer's perspective, access to time can be provided either by a clock primitive in the language or via a device driver for the internal clock, external clock or radio receiver. The programming of device drivers is the topic of Chapter 14; the following subsections illustrate how Ada, Java and C/Real-Time POSIX provide clock abstractions. In general these languages are silent about how these abstractions should be interpreted in distributed systems.

### 9.2.1 The clock packages in Ada

Access to a clock in Ada is provided by a predefined (compulsory) library package called `Calendar` and an optional real-time facility. The `Calendar` package (see Program 9.1) implements an abstract data type for `Time`. It provides a function `Clock` for reading the time and various subprograms for converting between `Time` and humanly understandable units, such as `Years`, `Months`, `Days` and `Seconds`. The first three of these are given as integer subtypes. `Seconds` are, however, defined as a subtype of the primitive type `Duration`.

**Program 9.1** The Ada `Calendar` package.

```
package Ada.Calendar is

  type Time is private;

  subtype Year_Number is Integer range 1901..2099;
  subtype Month_Number is Integer range 1..12;
  subtype Day_Number is Integer range 1..31;
  subtype Day_Duration is Duration range 0.0..86400.0;

  function Clock return Time;

  function Year(Date:Time) return Year_Number;
  function Month(Date:Time) return Month_Number;
  function Day(Date:Time) return Day_Number;
  function Seconds(Date:Time) return Day_Duration;

  procedure Split(Date:in Time; Year:out Year_Number;
                  Month:out Month_Number; Day:out Day_Number;
                  Seconds:out Day_Duration);

  function Time_Of(Year:Year_Number; Month:Month_Number;
                  Day:Day_Number; Seconds:Day_Duration := 0.0)
                  return Time;

  function "+"(Left:Time;Right:Duration) return Time;
  function "+"(Left:Duration;Right:Time) return Time;
  function "-"(Left:Time;Right:Duration) return Time;
  function "-"(Left:Time;Right:Time) return Duration;

  function "<"(Left,Right:Time) return Boolean;
  function "<="(Left,Right:Time) return Boolean;
  function ">"(Left,Right:Time) return Boolean;
  function ">="(Left,Right:Time) return Boolean;

  Time_Error:exception;
  -- Time_Error is raised by Time_Of, Split,"+",and "-"

private
  -- implementation dependent
end Ada.Calendar;
```

Type `Duration` is a predefined fixed-point real that represents an interval of time (relative time). Both its accuracy and its range are implementation-dependent, although its range must be at least –86 400.0 .. 86 400.0, which covers the number of seconds in a day. Its granularity must be no greater than 20 milliseconds. In essence, a value of type `Duration` should be interpreted as a value in seconds. Note that in addition to the above subprograms, the `Calendar` package defines arithmetic operators for combinations of `Duration` and `Time` parameters and comparative operations for `Time` values.

The code required to measure the time taken to perform a computation is quite straightforward. Note the use of the `"-"` operator, which takes two `Time` values but returns a value of type `Duration`.

```
declare
  Old_Time, New_Time : Time;
  Interval : Duration;
begin
  Old_Time := Clock;
  -- other computations
  New_Time := Clock;
  Interval := New_Time - Old_Time;
end;
```

In the latest version of Ada (Ada 2005) some additional packages have been provided that help with:

- constructing code that manipulates time values (including days that may have leap seconds included/excluded);
- formatting time values for input and output; and
- dealing with time zones.

The other language clock is provided by the optional package `Real_Time`. This has a similar form to `Calendar` but is intended to give a finer granularity. The constant `Time_Unit` is the smallest amount of time representable by the `Time` type. The value of `Tick` must be no greater than one millisecond; the range of `Time` (from the epoch that represents the program's start-up) must be at least fifty years.

As well as providing a finer granularity, the `Clock` of `Real_Time` is defined to be **monotonic**. The `Calendar` clock is intended to provide an abstraction for a 'wall clock' and is, therefore, subject to leap years, leap seconds and other adjustments. A monotonic clock has *no* such adjustments. The `Real_Time` package is outlined in Program 9.2.

In addition to these real-time clocks Ada also provides clocks that measure the execution time of task. These facilities are discussed in Chapter 12.

### 9.2.2 Clocks in Real-Time Java

Standard Java supports the notion of a wall clock, and thus has facilities similar to Ada. The current time can be found in Java by calling the static method `System.currentTimeMillis` in the package `java.lang`. This returns the number of milliseconds since midnight, January 1, 1970 GMT. The `Date` and `Calendar` classes in `java.util` use this method as a default when constructing date objects.

**Program 9.2** The Ada `Real_Time` package.

```
package Ada.Real_Time is
  type Time is private;
  Time_First: constant Time;
  Time_Last: constant Time;
  Time_Unit: constant := -- implementation-defined-real-number;

  type Time_Span is private;
  Time_Span_First: constant Time_Span;
  Time_Span_Last: constant Time_Span;
  Time_Span_Zero: constant Time_Span;
  Time_Span_Unit: constant Time_Span;

  Tick: constant Time_Span;
  function Clock return Time;

  function "+" (Left: Time; Right: Time_Span) return Time;
  ...

  function "<" (Left, Right: Time) return Boolean;
  ...

  function "+" (Left, Right: Time_Span) return Time_Span;
  ...

  function "<" (Left, Right: Time_Span) return Boolean;
  ...

  function "abs"(Right : Time_Span) return Time_Span;

  function To_Duration (Ts : Time_Span) return Duration;
  function To_Time_Span (D : Duration) return Time_Span;

  function Nanoseconds  (Ns: Integer) return Time_Span;
  function Microseconds (Us: Integer) return Time_Span;
  function Milliseconds (Ms: Integer) return Time_Span;

  type Seconds_Count is range -- implementation-defined

  procedure Split(T : in Time; Sc: out Seconds_Count;
                  Ts : out Time_Span);
  function Time_Of(Sc: Seconds_Count; Ts: Time_Span) return Time;

private
  -- not specified by the language
end Ada.Real_Time;
```

Real-Time Java adds to these facilities real-time clocks with high-resolution time types. Program 9.3 shows a summary of the base definition of the high-resolution time class. There are methods to read, write and compare time values. This abstract class has two subclasses: one which represents absolute time and one which represents relative time (similar to Ada's `Duration` type). These are given in Program 9.4. Absolute time is actually expressed as a time relative to 1 January 1970, GMT.

**Program 9.3** An extract of the Real-Time Java `HighResolutionTime` class.

```
public abstract class HighResolutionTime
      implements Comparable, Cloneable {
  // methods
  public int compareTo(HighResolutionTime time);
  public boolean equals(HighResolutionTime time);

  public final long getMilliseconds();
  public final int getNanoseconds();

  public void set(HighResolutionTime time);
  public void set(long millis);
  public void set(long millis, int nanos);

  public static void waitForObject(Object target,
      HighResolutionTime time) throws InterruptedException;
  ...
}
```

The `HighResolutionTime`, `AbsoluteTime` and `RelativeTime` classes are all reasonably self-explanatory. However, the method `absolute` does need further discussion. Its role is to convert the encapsulated time (be it absolute or relative) to an absolute time relative to some clock. If an `AbsoluteTime` object is passed as a parameter, the object is updated to reflect the encapsulated time value. Furthermore, the same object is also returned by the function. This is because if a null object is passed as a parameter, a new object is created and returned. In Java, the parameter is copied by value and, therefore, cannot be updated. Consequently, it is necessary for the function to create a new `AbsoluteTime` object and return it.

The Real-Time Java `Clock` class, given in Program 9.5, defines the abstract class from which all clocks are derived. The language allows many different types of clocks; for example, there could be an execution-time clock which measures the amount of execution time being consumed. There is always one real-time clock which advances in sync with the external world. The method `getRealtimeClock` allows this clock to be obtained.[1] Other methods are provided to get the resolution of a clock and, if the hardware permits, to set the resolution of a clock.

The code to measure the time taken to perform a computation is:

```
{
  AbsoluteTime oldTime, newTime;
  RelativeTime interval;
  Clock clock = Clock.getRealtimeClock();

  oldTime = clock.getTime();
  // other computations
  newTime = clock.getTime();

  interval = newTime.subtract(oldTime);

}
```

[1] Note that this is a static method, and therefore it can be called directly without knowledge of any subclasses.

**Program 9.4** The Real-Time Java `Absolute` and `Relative` classes.

```
public class AbsoluteTime extends HighResolutionTime {
  // various constructor methods including
  public AbsoluteTime(AbsoluteTime T);
  public AbsoluteTime(long millis, int nanos);

  public AbsoluteTime absolute(Clock clock, AbsoluteTime dest);

  public AbsoluteTime add(long millis, int nanos);
  public final AbsoluteTime add(RelativeTime time);
  ...
  public final RelativeTime subtract(AbsoluteTime time);
  public final AbsoluteTime subtract(RelativeTime time);

}

public class RelativeTime extends HighResolutionTime {
  // various constructor methods including
  public RelativeTime(long millis, int nanos);
  public RelativeTime(RelativeTime time);

  public AbsoluteTime absolute(Clock clock, AbsoluteTime destination);

  public RelativeTime add(long millis, int nanos);
  public final RelativeTime add(RelativeTime time);

  public final RelativeTime subtract(RelativeTime time);

  ...
}
```

**Program 9.5** The Real-Time Java `Clock` class.

```
public abstract class Clock {
  public Clock();

  public static Clock getRealtimeClock();

  public abstract RelativeTime getResolution();

  public AbsoluteTime getTime();
  public abstract void getTime(AbsoluteTime time);

  public abstract void setResolution(RelativeTime resolution);

}
```

### 9.2.3 Clocks in C/Real-Time POSIX

ANSI C has a standard library for interfacing to 'calendar' time. This defines a basic time type `time_t` and several routines for manipulating objects of type time; Program 9.6 defines some of these functions. C/Real-Time POSIX allows many clocks to be supported by an implementation. Each clock has its own identifier (of type `clockid_t`), and the IEEE standard requires that at least one clock be supported (`CLOCK_REALTIME`). A typical C interface to Real-Time POSIX clocks is illustrated by Program 9.7.

The value returned by a clock (via `clock_gettime`) is given by the structure `timespec`; `tv_sec` represents the number of seconds expired since 1 January 1970 and `tv_nsec` the additional number of nanoseconds (although it is shown as a long integer, the value taken by `tv_nsec` must always be less than 1 000 000 000 and non-negative – C provides no subtyping mechanism). C/Real-Time POSIX requires that the minimum resolution of `CLOCK_REALTIME` is 50 Hz (20 milliseconds); the function `clock_getres` allows the resolution of a clock to be determined.

### 9.2.4 Real-time and monotonic clocks

In the above discussions, two types of clocks have been identified: clocks that allow the absolute external (calendar) time to be obtained and monotonic clocks that allow the passage of time to be measured accurately. Table 9.2 summarizes the difference in the various clocks provided by Ada, C/Real-Time POSIX and Real-Time Java. Note the term 'real-time' clock has a slightly different meaning across the languages.

**Program 9.6** An interface to ANSI C dates and time.

```
typedef ... time_t;

struct tm {
  int tm_sec;     /* seconds after the minute - [0, 61] */
                  /* 61 allows for 2 leap seconds */
  int tm_min;     /* minutes after the hour - [0, 59] */
  int tm_hour;    /* hour since midnight - [0, 23] */
  int tm_mday;    /* day of the month - [1, 31] */
  int tm_mon;     /* months since January - [0, 11] */
  int tm_year;    /* years since 1900 */
  int tm_wday;    /* days since Sunday - [0, 6] */
  int tm_yday;    /* days since January 1 - [0, 365] */
  int tm_isdst;   /* flag for alternate daylight savings time */
}; double difftime(time_t time1, time_t time2);
  /* subtract two time values */
time_t mktime(struct tm *timeptr); /* compose a time value */
time_t time(time_t *timer);
  /* returns the current time and if timer is not null */
  /*it also places the time at that location */
```

**Program 9.7** The C/Real-Time POSIX interface to clocks.

```
#define CLOCK_REALTIME ...;
#define CLOCK_PROCESS_CPUTIME_ID ...;
#define CLOCK_THREAD_CPUTIME_ID ...;

struct timespec {
  time_t tv_sec;   /* number of seconds */
  long   tv_nsec;  /* number of nanoseconds */
};
typedef ... clockid_t;

int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
int clock_getres(clockid_t clock_id, struct timespec *res);

int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
int clock_getcpuclockid(pthread_t_t thread_id, clockid_t *clock_id);

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
/* Note, that a nanosleep return -1 if the sleep is interrupted */
/* by a signal. In this case, rmtp has the remaining sleep time */

int nano_nanosleep(clockid_t clock_id, int flags,
    const struct timespec *rqtp, struct timespec *rmtp);
/* if flag = TIMER_ABSTIME, then the sleep is absolute */
/* using the identified clock */
```

| | Calendar | Monotonic |
|---|---|---|
| Ada | package Calendar | package Real_Time |
| C/Real-Time POSIX | time function<br>CLOCK_REALTIME | CLOCK_MONOTONIC |
| Real-Time Java | currentTimeInMillis function | getRealtimeClock function |

**Table 9.2** Calendar and monotonic clocks.

## 9.3 Delaying a task

In addition to having access to a clock, tasks must also be able to delay their execution either for a relative period of time or until some absolute time in the future.

### 9.3.1 Relative delays

A relative delay enables a task to queue on a future event rather than busy-wait on calls to the clock. For example, the following Ada code shows how a task can loop waiting for 10 seconds to pass:

```
Start := Clock; -- from calendar
loop
  exit when (Clock - Start) > 10.0;
end loop;
```

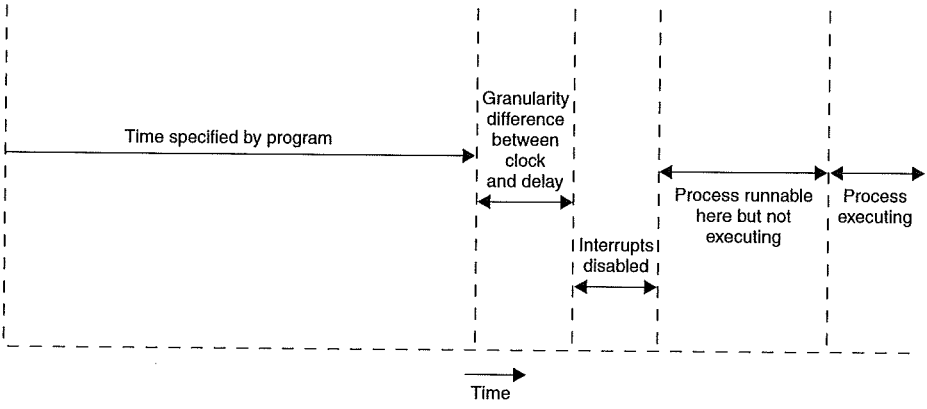



**Figure 9.1** Delay times.

To eliminate the need for these busy-waits, most languages and operating systems provide some form of delay primitive. In Ada, this is a delay statement.

```
delay 10.0;
```

The value after **delay** (of type `Duration`) is relative (that is, the above statement means delay 10 seconds from the current time – a negative value is treated as zero).

In C/Real-Time POSIX, a delay can be obtained by use of the 'sleep' system call if a coarse granularity is required (that is, a delay of 'seconds'), or 'nanosleep' if a finer granularity is required (see Program 9.7), the latter being measured in terms of the `CLOCK_REALTIME`. Java provides similar facilities to C/Real-Time POSIX. The `sleep` method in the `Thread` class allows a thread to delay itself at the milliseconds granularity. The `RealtimeThread` class allows a high-resolution sleep relative to a clock.

It is important to appreciate that a 'delay' or 'sleep' only guarantees that the task is made runnable after the period has expired.[2] The actual delay before the task begins executing is, of course, dependent on the other tasks which are competing for the processor. It should also be noted that the granularity of the delay and the granularity of the clock are not necessarily the same. For example, C/Real-Time POSIX and Real-Time Java allow a granularity down to the nanoseconds; however, few current systems will support this. Moreover, the internal clock may be implemented using an interrupt which could be inhibited for short periods. Figure 9.1 illustrates the factors affecting the delay.

### 9.3.2 Absolute delays

The use of **delay** in Ada (and `sleep` in C/Real-Time POSIX or Real-Time Java) supports a relative time period (for example, 10 seconds from now). If a delay to an absolute time is required, then either the programmer must calculate the period to

[2]In C/Real-Time POSIX, the process can actually be woken early if it receives a signal. In this case an error indication is returned, and the `rmpt` parameter returns the delay time remaining. The situation in Java is similar. If a thread is interrupted then is awoken and the error object `InterruptedException` is thrown. In Ada, a delayed tasks is only woken early if it is within a select-then-abort statement.

delay or an additional primitive is required. For example, if an action should take place 10 seconds after the start of some other action, then the following Ada code could be used (with `Calendar`):

```
Start := Clock;
First_Action;
delay 10.0 - (Clock - Start);
Second_Action;
```

Unfortunately, this might not achieve the desired result. In order for this formulation to have the required behaviour, then

```
delay 10.0 - (Clock - Start);
```

would have to be an uninterruptible (atomic) action, which it is not. For example, if `First_Action` took two seconds to complete then

```
10.0 - (Clock - Start);
```

would equate to eight seconds. However, after having calculated this value, if the task involved is preempted by some other task it could be three seconds (say) before it next executes. At that time it will delay for eight seconds rather than five. To solve this problem, Ada introduces the **`delay until`** statement:

```
Start := Clock;
First_Action;
delay until Start + 10.0;
Second_Action;
```

As with delay, **`delay until`** is accurate only in its lower bound. The task involved will not be released before the current time has reached that specified in the statement, but it may be released later.

The time overrun associated with both relative and absolute delays is called the **local drift** and it it cannot be eliminated. It is possible, however, to eliminate the **cumulative drift** that could arise if local drifts were allowed to superimpose. The following code, in Ada, shows how the computation `Action` is programmed to be executed, on average, every 7 seconds. This code will compensate for any local drift. For example, if two consecutive calls to `Action` were actually 7.4 seconds apart then the subsequent delay would be for only 6.6 seconds (approximately).

```
declare
  Next : Time;
  Interval : constant Duration := 7.0;
begin
  Next := Clock + Interval;
  loop
    Action;
    delay until Next;
    Next := Next + Interval;
  end loop;
end;
```

In Real-Time Java, the `sleep` method (defined in the `RealtimeThread` class) can be used for both relative and absolute delays.

Similarly, an absolute delay can also be constructed in C/Real-Time using the `clock_nanosleep` function (see Program 9.7).

## 9.4 Programming timeouts

Perhaps the simplest time constraint that an embedded system can have is the requirement to recognize, and act upon, the non-occurrence of some external event. For example, a temperature sensor may be required to log a new reading every second, the failure to give a reading within 10 seconds being defined as a fault. In general, a **timeout** is a restriction on the time a task is prepared to wait for a communication. Chapters 5 and 6 have discussed, in detail, inter-task communication facilities. Each one requires a timeout if it is to be used in a real-time environment.

As well as waiting for communication, timeouts are also required on the execution of actions. For example, a programmer might require that a section of code be executed within a certain time. If this does not happen at run-time, then some error recovery might be required.

### 9.4.1 Shared variable communication and timeouts

In Chapter 5, various communication and synchronization mechanisms based on shared variables were discussed. Both mutual exclusive access to a critical section and condition synchronization were seen as being important requirements. When a task attempts to gain access to a critical section, it is blocked if another task is already active in the section. This blocking, however, is bounded by the time taken to execute the code of the section and the number of other tasks that also wish to execute the critical section. For this reason, it is not usually deemed necessary to have a timeout associated with attempting entry. (However, C/Real-Time POSIX does provide such timeouts.) In Section 11.8, the issue of analysing this blocking time is considered in detail.

In contrast, the blocking associated with condition synchronization is not so easily bounded and depends on the application. For example, a producer task attempting to place data in a full buffer must wait for a consumer task to take data out. The consumer tasks might not be prepared to do this for a significant period of time. For this reason, it is important to allow tasks the option of timing out while waiting for condition synchronizations.

The condition synchronization facilities considered in Chapter 5 included:

- semaphores;
- conditional critical regions;
- condition variables in monitors, mutexes or synchronized methods;
- entries in protected objects.

Real-Time Euclid (Kligerman and Stoyenko, 1986) uses semaphores and extends the semantics of `wait` to include a time bound. The following statement illustrates how a process could suspend itself on the semaphore `CALL` with a timeout value of 10 seconds:

```
wait CALL noLongerThan 10 : 200
```

If the process is not signalled within 10 seconds, the exception 200 is raised (exceptions in Real-Time Euclid are numbered). Note, however, that there is no requirement for the process to be actually scheduled and executing within 10 seconds. The same is true for all timeout facilities. They merely indicate that the process should become runnable again.

C/Real-Time POSIX supports an explicit timeout for waiting on a semaphore or on a condition variable (see Programs 5.2 and 5.4). The following statement illustrates how a process could suspend itself on the semaphore `call` with an absolute timeout value of `timeout`:

```
if(sem_timedwait(&call, &timeout) < 0) {
  if ( errno ==  ETIMEDOUT) {
    /* timeout occurred */
  }
  else {
    /* some other error */
  }
} else {
  /* semaphore locked */
};
```

If the semaphore could not be locked by the `timeout`, `errno` is set to the value `ETIMEDOUT`.

With Ada's protected objects, avoidance synchronization is used and, therefore, it is at the point of the protected entry call that the timeout must be applied. As Ada treats a protected entry call the same as a task entry call, an identical mechanism for timeout is used. This is described below in terms of message passing.

With Java, the `wait` method can be used with a timeout either at millisecond granularity or at nanosecond granularity, the latter being provided by the `waitForObject` method in the `HighResolutionTime` class given in Program 9.3.

### 9.4.2 Message passing and timeouts

Chapter 6 has considered the various forms of message passing. All forms require synchronization. Even with asynchronous systems, a task may wish to wait for a message (or a sending task's message buffer might become full). With synchronous message passing, once a task has committed itself to a communication it must wait until such an event has occurred. For these reasons, timeouts are required. To illustrate the programming of timeouts consider a controller task (in Ada) that is called by some other driver task and given a new temperature reading:

```
task Controller is
  entry Call(T : Temperature);
end Controller;

task body Controller is
  -- declarations
begin
  loop
    accept Call(T : Temperature) do
       New_Temp := T;
    end Call;
```

```
    -- other actions
  end loop;
end Controller;
```

It is now required that the controller be modified so that the lack of the entry call is acted upon. This requirement can be provided using the constructs already discussed. A second task is used that delays itself for the timeout period and then calls the controller. However, the need for a timeout is so common that a more concise way of expressing it is desirable. This is usually provided in a real-time language as a special form of alternative in a selective wait. The above example would be coded as follows:

```
task Controller is
  entry Call(T : Temperature);
end Controller;

task body Controller is
  -- declarations
begin
  loop
    select
      accept Call(T : Temperature) do
        New_Temp := T;
      end Call;
    or
      delay 10.0;
      -- action for timeout
    end select;
    -- other actions
  end loop;
end Controller;
```

The delay alternative becomes ready when the time delay has expired. If this alternative is chosen (that is, a `Call` is not registered within 10 seconds) then the statements after the delay are executed.

The above example uses a relative delay, but absolute delays are also possible. Consider the following code that enables a task to accept registrations up to time `Closing_Time`:

```
task Ticket_Agent is
  entry Registration(...);
end Ticket_Agent;

task body Ticket_Agent is
  -- declarations
  Shop_Open : Boolean := True;
begin
  while Shop_Open loop
    select
      accept Registration(...) do
        -- log details
      end Registration;
    or
      delay until Closing_Time;
      Shop_Open := False;
    end select;
```

```
    -- task registrations
  end loop;
end Ticket_Agent;
```

Within the Ada model, it would not make sense to mix an else part, a terminate alternative and delay alternatives. These three structures are, therefore, mutually exclusive; a select statement can have, at most, only one of them. However, if it is a delay alternative, the select can have a number of delays but they must all be of the same kind (that is, all 'delay' statements or all 'delay until' statements). The one with the shortest duration or earliest absolute time is the operative one on each occasion.

A timeout facility is common in message-based concurrent programming languages. Ada actually goes further and allows a timeout on a message send. To illustrate this, consider the device driver that is feeding temperatures to the controller in the above Ada code:

```
loop
  -- get new temperature T
  Controller.Call(T);
end loop;
```

As new temperature readings are available continuously (and there is no point in giving the controller an out-of-date value), the device driver may wish to be suspended waiting for the controller for only half a second before withdrawing the call. This is achieved by using a special form of the select statement which has a single entry call and a single delay alternative:

```
loop
  -- get new temperature T
  select
    Controller.Call(T);
  or
    delay 0.5;
    null;
  end select;
end loop;
```

The `null` is not strictly needed but shows that again the delay can have arbitrary statements following, that are executed if the delay expires before the entry call is accepted. This is a special form of the select. It cannot have more than one entry call and it cannot mix entry calls and accept statements. The action it invokes is called a **timed entry call**. It must be emphasized that the time period specified in the call is a timeout value for the call being accepted; *it is not a timeout on the termination of the associated accept statement.*

When a task wishes only to make an entry call if the called task is immediately prepared to accept the call, then rather than make a timed entry call with time zero, a **conditional entry call** can be made:

```
select
  T.E   -- entry E in task T
else
  -- other actions
end select;
```

The 'other actions' are executed only if T is not prepared to accept E immediately. 'Immediately' means that either T is already suspended on **accept** E or on a select statement with such an open alternative (and it is chosen).

The above examples have used timeouts on inter-task communication; it is also possible, within Ada, to do timed (and conditional) entry call on protected objects:

```
select
  P.E ; -- E is an entry in Protected Object P
or
  delay 0.5;
end select;
```

### 9.4.3 Timeouts on actions

In Section 7.4, mechanisms were discussed that allowed tasks to have their control flows altered by asynchronous notifications. A timeout can be considered such a notification, and therefore if asynchronous notifications are supported, timeouts can also be used. Both a resumption (asynchronous events) and a termination (asynchronous transfer of control) model can be augmented by timeouts. Although for timeouts on actions, it is the termination model that is needed.

For example, in Section 7.6.1, the Ada asynchronous transfer of control (ATC) facility was introduced. With this, an action can be aborted if a 'triggering event' occurs before the action has completed. One of the allowed triggering events is the passage of time. To illustrate this, consider a task that contains an action that must be completed within 100 milliseconds. The following code supports this requirement directly:

```
select
  delay 0.1;
then abort
  -- action
end select;
```

If the action takes too long, the triggering event will be taken and the action will be aborted. This is clearly an effective way of catching code that is stuck in a non-terminating loop or has some other program error.

Timeouts are thus usually associated with error conditions; if a communication has not occurred within $X$ milliseconds then something unfortunate has happened and corrective action must be taken. This is, however, not their only use. Consider a task that has a compulsory component and an optional part. The compulsory computations produce (quickly) an adequate result that is assigned to, say in Ada, a protected object. The task must complete by a fixed time; but if there is time available after the compulsory component has been completed, an optional algorithm can be used to incrementally improve the output value. To program this again requires a timeout on an action.

```
declare
  Precise_Result : Boolean;
begin
  Completion_Time := ...
  --  compulsory part
  Results.Write(...); -- call to procedure in
                      -- external protected object
```

**Program 9.8** The Real-Time Java `Timed` class.

```
public class Timed extends AsynchronouslyInterruptedException
                           implements java.io.Serializable
{
  public Timed(HighResolutionTime time)
       throws IllegalArgumentException;

  public boolean doInterruptible(Interruptible logic);

  public void resetTime(HighResolutionTime time);
}
```

```
  select
    delay until Completion_Time;
    Precise_Result := False;
  then abort
    while Can_Be_Improved loop
      -- improve result
      Results.Write(...);
    end loop;
    Precise_Result := True;
  end select;
end;
```

Note that if the timeout expires during the write to the protected object it will complete its write correctly, as a call to a protected object is an abort-deferred action (that is, the effect of the abort is postponed until the task leaves the protected object).

With Real-Time Java, timeouts on actions are provided by a subclass of `AsynchronouslyInterruptedException` called `Timed`. This class is defined in Program 9.8 (see Section 6.8.4 for information concerning `java.io.Serializable`).

The above Ada example would be fully written in Real-Time Java as follows:

```
public class PreciseResult {
  public resultType value; // the result
  public boolean preciseResult; // indicates if it is imprecise
}

public class ImpreciseComputation {
  private HighResolutionTime CompletionTime;
  private PreciseResult result = new PreciseResult();

  public ImpreciseComputation(HighResolutionTime T)
  {
    CompletionTime = T;  //can be absolute or relative
  }

  private resultType compulsoryPart()
  {
    // function which computes the compulsory part
  }
```

```
  public PreciseResult Service()  // public service
  {
    Interruptible I = new Interruptible()
    {
      public void run(AsynchronouslyInterruptedException exception)
                    throws AsynchronouslyInterruptedException
      {
        // this is the optional function which improves on the
        // compulsory part
        boolean canBeImproved = true;

        while(canBeImproved)
        {
          // improve result
          synchronized(this) {
            // write result --
            // the synchronized statement ensures
            // atomicity of the write operation
          }
        }
        result.preciseResult = true;
      }

      public void interruptAction(
            AsynchronouslyInterruptedException exception)
      {
        result.preciseResult = false;
      }
    };

    Timed t = new Timed(CompletionTime);

    result.value = compulsoryPart();  // compute the compulsory part
    if(t.doInterruptible(I)) {
      // execute the optional part with the timer
      return result;
    } else { ... };
  }
}
```

Timeouts are an important feature of real-time systems; they are, however, far from being the only time constraints of significance. The rest of this chapter, and the following one, deals with the more general topic of time deadlines and how to ensure that they are met.

## 9.5 Specifying timing requirements

For many important real-time systems, it is not sufficient for the software to be logically correct; the programs must also satisfy timing constraints determined by the underlying physical system. These constraints can go far beyond simple timeouts. Unfortunately, existing practices in the engineering of large real-time systems are, in general, still rather

*ad hoc*. Often, a logically correct system is specified, designed and constructed (perhaps as a prototype) and then tested to see if it meets its timing requirements. If it does not, then various fine tunings and rewrites ensue. The result is a system that may be difficult to understand and expensive to maintain and upgrade. A more systematic treatment of time is required.

Work on a more rigorous approach to this aspect of real-time systems has followed two largely distinct paths. One direction of development has concerned the use of formally defined language semantics and timing requirements, together with notations and logics that enable temporal properties to be represented and analysed. The other path has focused on the performance of real-time systems in terms of the feasibility of scheduling the required workload on the available resources (processors and so on).

In this book, attention is focused mainly on the latter work. The reasons for this are three-fold. Firstly, the formal techniques are not yet mature enough to reason about large complex real-time systems. Secondly, there is little reported experience of the use of these techniques in actual real-time systems. Finally, to include a full discussion of such methods would involve a substantial amount of material that is outside the scope of this book. This is not meant to imply that the area is irrelevant to real-time systems. The understanding of, for example, formal techniques based on Communicating Sequential Processes (CSP), temporal logics, real-time logics, model checking and specification techniques that incorporate notions of time is becoming increasingly important.

The success of model checking in verifying functional properties has recently been extended into the real-time domain. A system is modelled as a Timed Automaton (that is, a finite state machine with clocks) and then model checking is used to 'prove' that undesirable states cannot be reached or that desirable states will be entered before some internal clock has reached a critical time (deadline). The latter property is known as *bounded liveliness*. Although the explorations undertaken by model checking are subject to state explosion, a number of tools are now available that can tackle problems of a reasonable size. The technique is likely to become standard industrial practice over the coming years.

The verification of a real-time system can usefully be interpreted as requiring a two-stage process.

(1) Verifying requirements/designs – given an infinitely fast reliable computer, are the temporal requirements coherent and consistent; that is, have they the potential to be satisfied?

(2) Verifying the implementation – with a finite set of (possible unreliable) hardware resources, can the temporal requirements be satisfied?

As indicated above, (1) may require formal reasoning (and/or model checking) to verify that necessary temporal (and causal) orderings are satisfied. For example, if event A must be completed before event B, but is dependent on some event C that occurs after B, then no matter how fast the processor it will never be possible to satisfy these requirements. Early recognition of this difficulty is therefore very useful. The second issue (implementation verification) is the topic of Chapter 11. The remainder of this chapter will concentrate on how temporal requirements can actually be represented in languages.

## 9.6 Temporal scopes

To facilitate the specification of the various timing constraints found in real-time applications, it is useful to introduce the notion of **temporal scopes**. Such scopes identify a collection of statements with an associated timing constraint. The possible attributes of a temporal scope (TS) are illustrated in Figure 9.2, and include:

(1) **deadline** – the time by which the execution of a TS must be finished;
(2) **minimum delay** – the minimum amount of time that must elapse before the start of execution of a TS;
(3) **maximum execution time** – of a TS;
(4) **maximum elapsed time** – of a TS.

Temporal scopes with combinations of these attributes are also possible, and for some timing constraints a combination of sequentially executed temporal scopes is necessary. For example consider a simple control action that reads a sensor, computes a new setting and outputs this setting via an actuator. To get fine control over when the sensor is read, an initial temporal scope with a tight deadline is needed. The output is produced in a second temporal scope which has a minimum delay equal to the first

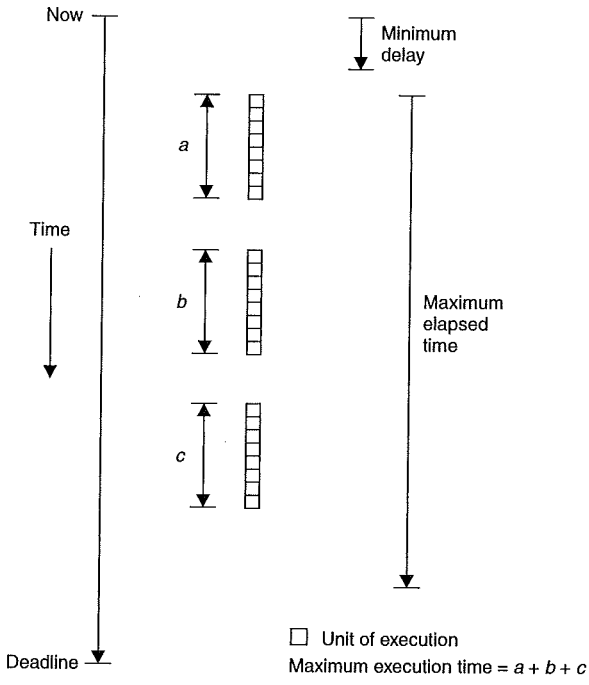



**Figure 9.2** Temporal scopes.

scope's deadline but a later deadline itself. The variation of when a sensor is read is called **input jitter**. If there is a need to also control **output jitter** then a third temporal scope could be added which has a long 'minimum delay' and a short time interval before its deadline.

Temporal scopes can themselves be described as being either **periodic** or **aperiodic**. Typically, periodic temporal scopes sample data or execute a control loop and have explicit deadlines that must be met. Aperiodic, or **sporadic**, temporal scopes usually arise from asynchronous events outside the embedded computer. These scopes have specified response times associated with them.

In general, aperiodic temporal scopes are viewed as being activated randomly, following, for example, a Poisson distribution. Such a distribution allows for bursts of arrivals of external events, but does not preclude any possible concentration of aperiodic activity. It is therefore not possible to do worst-case analysis (there is a non-zero probability of any number of aperiodic events occurring within a given time). To allow worst-case calculations to be made, a minimum period between any two aperiodic events (from the same source) is often defined. If this is the case, the task involved is said to be **sporadic**. In this book, the term 'aperiodic' is used for the general case and 'sporadic' is reserved for situations where a minimum delay between subsequent executions is needed.

In many real-time languages, temporal scopes are, in effect, associated with the tasks that embody them. Tasks can be described as either periodic, aperiodic or sporadic depending on the properties of their internal temporal scope. Most of the timing attributes given in the above list can thus be satisfied by:

(1) running periodic tasks at the correct rate;

(2) completing all tasks by their deadline.

The problem of satisfying timing constraints thus becomes one of scheduling tasks to meet deadlines, or **deadline scheduling**.

Although all computer systems strive to be efficient, and many are described as real-time, further classification is needed to deal adequately with the different levels of importance that time has within applications. As noted in Chapter 1, a system is said to be **hard** real-time if it has deadlines that cannot be missed or else the system fails. By comparison, a system is **soft** if the application is tolerant of missed deadlines. A system is merely **interactive** if it does not have specified deadlines but strives for 'adequate response times'. To give a little more precision, a task with a soft deadline may still deliver its service late – the system will still draw some value from the tardy service. A non-hard task that has a fixed rigid deadline (that is, a tardy service is useless/valueless) is said to be **firm**.

The distinction between hard, firm and soft real-time becomes somewhat blurred in fault-tolerant systems. Nevertheless, it is usually appropriate to use the term 'hard' if a specific error recovery (or fail safe) routine is triggered by a missed deadline, and firm/soft if the nature of the application is tolerant of the occasional missed deadline or deadlines that are not missed by much. Finally, note that many hard real-time systems will have some deadlines which are soft or firm.

### 9.6.1 Specifying tasks and temporal scopes

In real-time systems, it is necessary to deal explicitly with timing requirements, four types of which were given earlier. A general scheme for a periodic task is thus as follows:

```
task Task_T;
  ...
begin
  loop
    IDLE
    start of temporal scope
      ...
    end of temporal scope
  end;
end;
```

The time constraints take the form of a minimum time for IDLE and the requirement that the end of the temporal scope be by some deadline. This deadline can itself be expressed in terms of either:

- absolute time;
- execution time since the start of the temporal scope; or
- elapsed time since the start of the temporal scope.

As noted earlier a task that is sampling data may be composed of a number of temporal scopes:

```
loop
  start of 1st temporal scope
    ...
  end of 1st temporal scope
  start of 2nd temporal scope
    ...
  end of 2nd temporal scope
  IDLE
  start of 3rd temporal scope
    ...
  end of 3rd temporal scope
end;
```

The input activities of this task take place in the 1st temporal scope and hence the deadline at the end of this scope regulates the maximum input jitter for the task. The input data may already be available in buffers for the task, or the task may need to read input registers in the sensor's device interface. The 2nd temporal scope incorporates whatever computations are needed to calculate the task's output values (this may include interacting with other tasks). The 3rd scope is concerned with the output action. Here the IDLE interval is important; it is measured from the beginning of the loop and constrains the time before the output can be produced by the task. The deadline on this final temporal scope places an upper bound on the time of the output phase.

Although it would be possible to incorporate this sequence into a single task, scheduling analysis (see Chapter 11) places restrictions on the structure of a task.

Specifically, a task must only have at most one IDLE statement (at the beginning of the execution sequence) and one deadline (at the end). So, for illustration, assume the control task has a period of 100 ms, a constraint on input jitter of 5 ms, a constraint on output jitter of 10 ms and a deadline of 80 ms. The three necessary tasks would take the form:

```
task periodic_PartA;
  ...
begin
  loop every 100ms
    start of temporal scope
      input operations
      write data to a shared object
    end of temporal scope - deadline 5ms
  end;
end;

task periodic_PartB;
  ...
begin
  loop every 100ms
    IDLE 5ms
    start of temporal scope
      read form shared object
      computations
      write data to a shared object
    end of temporal scope - deadline 65ms
  end;
end;

task periodic_PartC;
  ...
begin
  loop every 100ms
    IDLE 70ms
    start of temporal scope
      read form shared object
      output operations
    end of temporal scope - deadline 10ms
  end;
end;
```

Deadlines are also desirable with aperiodic tasks; here the temporal scope is triggered by an external event that will normally take the form of an interrupt:

```
task aperiodic_P;
  ...
begin
  loop
    wait for interrupt
    start of temporal scope
      ...
    end of temporal scope
  end;
end;
```

Clearly, a periodic task has a defined periodicity (that is, how often the task loop is executed); this measure may also be applied to an aperiodic task in which case it means the maximum rate at which this task will cycle (that is, the fastest rate for interrupt arrivals). As stated earlier, such aperiodic tasks are known as sporadic.

## Summary

The management of time presents a number of difficulties that set embedded systems apart from other computing applications. Current real-time languages are often inadequate in their provisions for this vital area.

The introduction of the notion of time into real-time programming languages has been described in terms of four requirements:

- access to a clock;
- delaying;
- timeouts;
- deadline specification and scheduling.

The sophistication of the means provided to measure the passage of time varies greatly between languages. Ada and Real-Time Java provide two abstract data types for time and a collection of time-related operators. C/Real-Time POSIX provides a comprehensive set of facilities for clocks and timers including periodic timers.

If a task wishes to pause for a period of time, a delay (or sleep) primitive is needed to prevent the task having to busy-wait. Such a primitive always guarantees to suspend the task for at least the designated time, but it cannot force the scheduler to run the task immediately the delay has expired. It is not possible, therefore, to avoid local drift, although it is possible to limit the cumulative drift that could arise from repeated execution of delays.

For many real-time systems, it is not sufficient for the software to be logically correct; the programs must also satisfy timing constraints. Unfortunately, existing practices in the engineering of large real-time systems are, in general, still rather *ad hoc*. To facilitate the specification of timing constraints and requirements, it is useful to introduce the notion of a 'temporal scope'. Possible attributes of temporal scopes include:

- deadline for completion of execution;
- minimum delay before start of execution;
- maximum execution time;
- maximum elapse time.

Temporal scopes can be combined and embedded in tasks that are required to execute in accordance with their timing constraints. Input and output jitter can be controlled and deadlines specified for the actions necessary to produce inputs from outputs.

## Further reading

Buttazzo, G. C. (1997) *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. New York: Kluwer Academic.

Coulouris, G., Dollimore, J. and Kindberg, T. (2005) *Distributed Systems, Concepts and Design* (4th edn). Harlow: Addison-Wesley.

Joseph, M. (ed.) (1996) *Real-Time Systems: Specification, Verification and Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Koptez, H. (1997) *Real-Time Systems*. New York: Kluwer Academic.

Turski, W. M. (1988) Time considered irrelevant for real-time systems. *BIT*, **28**(3), 473–488.

### Exercises

**9.1** Explain how a system can be transformed so that all timing failures manifest themselves as value failures. Can the converse be achieved?

**9.2** Should Ada's timed entry call specify a timeout on the *completion* of the rendezvous, rather than the start of the rendezvous? Give an example of when such an approach might be useful. How might the same effect be obtained?

# Chapter 10
# Programming real-time abstractions



One of the issues raised in the previous chapter was the need to specify timing requirements arising from within the application. The notion of a *temporal scope* was introduced. A temporal scope has several attributes that when combined within a task allow important real-time characteristics to be presented. This chapter illustrates how combinations of these attributes can be used to:

- program periodic tasks;
- program sporadic and aperiodic tasks; and
- control input and output jitter.

Tasks that have any of the above attributes are termed **real-time tasks**.

Languages can support the programming of real-time tasks at various levels of abstraction. For example, Ada and C/Real-Time POSIX provide relatively low-level abstractions that the programmer can combine to implement various real-time models. In contrast, Real-Time Java provides a combination of low- and high-level abstraction. Research-oriented languages tend to focus on particular aspects and explore language support in those areas.

This chapter first considers the relationship between a real-time task and the typical task representation techniques that were considered in Chapter 4. It then illustrates how periodic activities can be programmed in C/Real-Time POSIX, Ada and Real-Time Java. This is followed by a discussion on how event-triggered sporadic and aperiodic activities can be implemented in the same languages. After this, the chapter turns its attention to the control of input and output jitter. Following this, brief consideration is given to the other languages to illustrate the alternative support that could be given.

## 10.1 Real-time tasks

In theory any concurrent activity (be it a result of explicit task creation or the execution of a fork or a cobegin statement) can be considered real-time if real-time attributes are associated with it, and it is scheduled for execution by a real-time scheduler. However, in practice, real-time tasks often have constraints placed on the programming style to ensure that their execution is predictable – if not deterministic. For example, a hard real-time task might be prohibited from executing an unbounded loop statement. Languages like Ada and C/Real-Time POSIX do not syntactically distinguish between a concurrent task/thread and a real-time task/thread. Instead, programming guidelines are applied and tools in the development environment are used to check conformance.

Real-Time Java takes a slightly different approach because it is an extension of Java and consequently must allow the execution of Java programs with standard Java semantics. For this reason, it distinguished between Java threads and those threads that have real-time semantics. The latter are called real-time threads, and are represented by a standard class in the Real-Time Java environment – shown in Programs 10.1 and 10.2. Real-time threads in Java have the following attributes associated with them.

- **Release parameters** – giving, amongst other things, the real-time thread's deadline; if the object is released periodically or sporadically, then subclasses allow an interval to be given. The base class for release parameters is given in Program 10.3. As well as having a deadline, all releases have an associated 'cost'. This is typically the worst-case execution time of the real-time thread (see Section 11.13).

**Program 10.1** An extract of the `RealtimeThread` class.

```
package javax.realtime;
public class RealtimeThread extends Thread
     implements Schedulable {
  // constructors
  public RealtimeThread();
  public RealtimeThread(SchedulingParameters scheduling);
  public RealtimeThread(SchedulingParameters scheduling,
                  ReleaseParameters release);
  public RealtimeThread(SchedulingParameters scheduling,
      ReleaseParameters release, MemoryParameters memory,
      MemoryArea area, ProcessingGroupParameters group,
      Runnable logic);

  // methods
  public void release(); // used for aperiodic execution
  public void start(); // override Thread.start()
  public boolean waitForNextPeriod();
    // used for periodic execution
  public boolean waitForNextRelease();
    // used for aperiodic execution
  ...
}
```

**Program 10.2** An extract of the `NoHeapRealtimeThread` class.

```
package javax.realtime;
public class NoHeapRealtimeThread extends RealtimeThread {
  // constructors

  public NoHeapRealtimeThread(
     SchedulingParameters scheduling, MemoryArea area);
  public NoHeapRealtimeThread(
      SchedulingParameters scheduling,
      ReleaseParameters release, MemoryArea area);
  ...
}
```

**Program 10.3** The `ReleaseParameters` class.

```
package javax.realtime;
public class ReleaseParameters implements Cloneable{
  // constructors
  protected ReleaseParameters();
  protected ReleaseParameters(RelativeTime cost,
          RelativeTime deadline,
          AsyncEventHandler overrunHandler,
          AsyncEventHandler missHandler);

  protected ReleaseParameters(RelativeTime cost,
          RelativeTime deadline,
          RelativeTime blockingTerm,
          AsyncEventHandler overrunHandler,
          AsyncEventHandler missHandler);
  // methods
  public Object clone();
  public RelativeTime getCost();
  public AsyncEventHandler getCostOverrunHandler();
  public RelativeTime getDeadline();
  public AsyncEventHandler getDeadlineMissHandler()
  public void setCost(RelativeTime cost);

  public void setCostOverrunHandler(
            AsyncEventHandler handler);
  public void setDeadline(RelativeTime deadline);

  public void setDeadlineMissHandler(
           AsyncEventHandler handler);
  public boolean setIfFeasible(RelativeTime cost,
                         RelativeTime deadline);

  public RelativeTime getBlockingTerm();
  public void setBlockingTerm(RelativeTime blockingTerm);
  public boolean setIfFeasible(RelativeTime cost,
     RelativeTime deadline, RelativeTime blockingTerm);
}
```

- **Scheduling parameters** – defining the attributes that will be used by the real-time scheduler (see Section 12.7).
- **Memory parameters** – giving the maximum amount of memory used by the object in its default memory area, the maximum amount of memory used in immortal memory, and a maximum allocation rate of heap memory (see Section 14.7.1).
- **Processing group parameters** – this allows several schedulable objects to be treated as a group and to have an associated period, cost and deadline (see Section 13.6.2).

Further information about these parameter classes will be given in due course. For the time being it is adequate to know that there are default values that can be overwritten via the various constructors shown in Program 10.1.

In spite of this explicit identification of a real-time thread, Real-Time Java provides little support to enforce hard real-time programming guidelines. The exception is that it defines a subclass of the `RealtimeThread` class called `NoHeapRealtimeThread` – see Program 10.2. Threads created via this class are not allowed to access the Java heap and therefore cannot be subject to garbage collection delay.[1]

## 10.2 Programming periodic activities

### 10.2.1 Ada and C/Real-Time POSIX

In keeping with many real-time languages, Ada and C/Real-Time POSIX do not support the explicit specification of periodic or sporadic tasks with deadlines; rather low-level mechanisms are provided that can be used for a variety of purposes – e.g. a delay primitive, timers and so on. These can be used within a looping task/thread to provide the same functionality as a periodic activity.

For example, in Ada, a periodic task might take the following form:

```
task body Periodic_T is
  Next_Release : Time;
  Release_Interval : constant Time_Span := Milliseconds(...);
begin
  -- read clock and calculate the next
  -- release time (Next_Release)
  loop
    -- sample data (for example) or
    -- calculate and send a control signal
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
  end loop;
end Periodic_T;
```

[1]Hard real-time systems, particularly those that are safety-critical, are very conservative in their use of dynamic memory, and prefer not to rely on garbage collection due to the potential for unpredictable delays – see Section 14.7.1.

This is comparable to the C/Real-Time POSIX representation:

```
#include <signal.h>
#include <time.h>
#include <pthread.h>
#include <misc/timespec_operations.h>
void periodic_thread() /* destined to be the thread */
{
  struct timespec next_release, remaining_time;
  struct timespec thread_period; /* actual period */

  /* read clock and calculate the next
     release time (next_release) */

  while(1) {
    /* sample data (for example) or
       calculate and send a control signal */

    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                    &next_release, &remaining_time);
    add_timespec(&next_release, &next_release, &thread_period);
  }
}

int init(){
  pthread_attr_t attributes;      /* thread attributes */
  pthread_t PT;                   /* thread pointer */

  pthread_attr_init(&attributes); /* default attributes */
  pthread_create(&PT, &attributes,
                 (void *)periodic_thread, (void *)0);
}
```

The `clock_nanosleep` function is equivalent to Ada's 'delay until' statement. In this case, it uses the real-time clock and an absolute time. Note that the sleep can be interrupted by a signal, in which case `remaining_time` indicates how long is remaining. In this example, it is assumed that no interruption occurs.

As time values in C/Real-Time POSIX are structures, the `add_timespec` function is used to add two time values together and return the result in a third value.

### 10.2.2 Real-Time Java

In Real-Time Java, periodic activities are represented by real-time threads with associated periodic release parameters. The `PeriodicParameters` class is given in Program 10.4. In addition to the release parameter attributes, each periodic activity has a start time (which may be relative or absolute) and a period.

The method `start` is called for the initial release of the thread; once the thread has executed, it calls `waitForNextPeriod` (wFNP) to indicate to the scheduler that

**Program 10.4** An extract of the `PeriodicParameters` class.

```
package javax.realtime;
public class PeriodicParameters extends ReleaseParameters {
  // Constructors: throw IllegalArgumentException if
  // period is null.
  public PeriodicParameters(RelativeTime Period);
  public PeriodicParameters(HighResolutionTime start,
       RelativeTime period);
  public PeriodicParameters(
       HighResolutionTime start, RelativeTime period,
       RelativeTime cost, RelativeTime deadline,
       AsyncEventHandler overrunHandler,
       AsyncEventHandler missHandler);

  public PeriodicParameters(
       HighResolutionTime start, RelativeTime period,
       RelativeTime cost, RelativeTime deadline,
       RelativeTime blockingTerm,
       AsyncEventHandler overrunHandler,
       AsyncEventHandler missHandler);

  // methods
  public RelativeTime getPeriod();
  public HighResolutionTime getStart();
  public void setPeriod(RelativeTime period);
  public void setStart(HighResolutionTime start);
  ...
}
```

it should be made executable again when its next period is due. The following illustrates the approach:

```
public class Periodic extends RealtimeThread {
  public Periodic(PeriodicParameters P)
  { ... }

  public void run() {
    boolean deadlineMet = true;
    while(deadlineMet) {
     // code to be run each period
     ...
     deadlineMet = waitForNextPeriod();
    }
  }
}
```

The actual semantics of wFNP are quite complex and depend on whether the real-time thread has missed its deadline and whether the programmer has requested that an asynchronous event handler be released when this occurs (see Section 13.2.2). Here, it

is assumed that no handler is to be released, in which case the semantics of wFNP are as follows.

- If the deadline was met – wFNP returns true at the next release time.
- If the deadline was missed – wFNP return false immediately it is called. Effectively this allows the programmer to respond to the missed deadline and call wFNP again. The second call to wFNP returns true at the next release time.

The following program fragment illustrates how to create a periodic thread with a 10 millisecond period, a 5 millisecond deadline, whose first execution is delayed until an absolute time A. The thread should consume no more than 1 millisecond of processor time.

```
{
  AbsoluteTime A = new AbsoluteTime(...);
  PeriodicParameters P = new PeriodicParameters(
     A, // start time
     new RelativeTime(10,0), // period
     new RelativeTime(1,0),  // cost
     new RelativeTime(5,0),  // deadline
     null,  null ); // no deadline miss/cost overrun handlers

  Periodic ourThread = new Periodic(P); //create thread
  ourThread.start(); // release it
}
```

Note that as with Ada and C/Real-time POSIX, it is necessary to code the loop. However, unlike these languages, the program does not explicitly have to calculate the required delay.

## 10.3 Programming aperiodic and sporadic activities

As indicated in Section 9.6, the main difference between aperiodic and sporadic activities is that the latter have a defined minimum inter-arrival time (MIT), whereas the former do not. From a programming point of view, the main issue is whether the language/operating system supports the detection of MIT violations. This will be considered in Section 13.7.1. Here the focus is on the structure needed to support general aperiodic tasks.

### 10.3.1 Aperiodic tasks in Ada

In common with Ada's approach to supporting periodic task, the language provides no direct abstractions to support aperiodic or sporadic tasks. Instead, the lower-level mechanisms can be used to program the precise model required. Consider, for example, an aperiodic Ada task that is triggered by an interrupt. Typically, it is necessary to use a protected object to handle the interrupt and release the task for execution:

```
protected Aperiodic_Controller is
  procedure Interrupt; -- mapped onto interrupt
```

# Chapter 12
# Programming schedulable systems



In the previous chapter a number of approaches to scheduling were introduced. These approaches were chosen partly due to their inherent value and maturity, but also because it is possible to use these techniques in programming real systems with current languages and operating systems. This chapter starts by looking briefly at the programming of cyclic executives, and then looks in detail at the support available for priority-based scheduling. EDF is also covered.

## 12.1 Programming cyclic executives

To implement a simple cyclic executive requires no special language support. None of the temporal or scheduling information needs to be represented in the application's code. Rather, a series of procedure calls are linked together with a simple hardware interrupt that is configured to occur at regular intervals. For example, the code for the example given in the previous chapter (see Section 11.1) would have the following simple form (with the interrupt occurring every 25 ms):

```
loop -- MAJOR CYCLE
  wait_for_interrupt;
  -- minor cycle 1
  procedure_for_a;
  procedure_for_b;
  procedure_for_c;
```

```
    wait_for_interrupt;
    -- minor cycle 2
    procedure_for_a;
    procedure_for_b;
    procedure_for_d;
    procedure_for_e;
    wait_for_interrupt;
    -- minor cycle 3
    procedure_for_a;
    procedure_for_b;
    procedure_for_c;
    wait_for_interrupt;
    -- minor cycle 4
    procedure_for_a;
    procedure_for_b;
    procedure_for_d;
end loop;
```

Some protection can be added to this structure to identify an execution time overrun. The last procedure in any minor cycle sets a binary flag to 1. The interrupt handler checks the value of the flag before it switches to the next minor cycle (and assigning 0 to the flag). If there is a minor cycle overflow then recovery can be attempted, or extra time provided, or the system restarted. Of course the recovery code cannot know which procedure in the minor cycle caused the overflow. There is no firewall protection between the procedures.

## 12.2 Programming preemptive priority-based systems

Few programming languages explicitly define priorities as part of their concurrency facilities. Those that do often provide only a rudimentary model.

One language that does attempt to give a more complete provision is Ada – this is, therefore, discussed in detail in the next section. Traditionally, priority-based scheduling has been more an issue for operating systems than languages. Hence, after a discussion of Ada, the facilities of C/Real-Time POSIX are reviewed. Ada and C/Real-Time POSIX assume that any schedulability analysis has been performed offline. More recently, Real-Time Java has attempted to provide the same facilities as Ada and C/Real-Time POSIX but with the option of supporting online feasibility analysis. This is considered in Section 12.7.

It should, perhaps, be noted at this point that although most general-purpose operating systems provide the notion of process or thread priority, their facilities are often inadequate for hard real-time programming. What is minimally required is:

- an effective range of priorities;
- immediate preemptive switching to higher priority processes when they become runnable;
- support for at least priority inheritance, but ideally some form of priority ceiling protocol.

## 12.3 Ada and fixed-priority scheduling

As indicated in the Preface, Ada is defined as a core language plus a number of annexes for specialized application domains. These annexes do not contain any new language features (in the way of new syntax) but define pragmas and library packages that must be supported if that particular annex is to be adhered to. This section considers some of the provisions of the Real-Time Systems Annex, in particular, those that allow priorities to be assigned to tasks (and protected objects).[1]

To indicate that priority-based scheduling is required of the run-time implementation of the application's Ada program, the dispatching policy must be defined. This is done using a pragma:

```
pragma Task_Dispatching_Policy(FIFO_Within_Priority);
```

Where tasks share the same priority, then they are queued in FIFO order. Hence, as tasks become runnable, they are placed at the *back* of a notional **ready queue** for that priority level. One exception to this case is when a task is preempted; here the task is placed at the *front* of the ready queue for that priority level. On a multiprocessor system, it is implementation defined whether this policy is on a per-processor basis or across the entire processor cluster.

To allocate a priority to a task there are, within package `System`, the following declarations:

```
subtype Any_Priority is Integer range
        <implementation-defined>;
subtype Priority is Any_Priority range
        Any_Priority'First .. <implementation-defined>;
subtype Interrupt_Priority is Any_Priority range
        Priority'Last+1 .. Any_Priority'Last;

Default_Priority : constant Priority :=
                   (Priority'First + Priority'Last)/2;
```

An integer range is split between standard priorities and (the higher) interrupt priority range. An implementation must support a range for `System.Priority` of at least 30 values and at least one distinct `System.Interrupt_Priority` value.

A task has its initial priority set by including a pragma in its specification:

```
task Controller is
   pragma Priority(10);
end Controller;
```

If a task-type definition contains such a pragma, then all tasks of that type will have the same priority unless a discriminant is used:

```
task type Servers(Task_Priority : System.Priority) is
  entry Service1(...);
  entry Service2(...);
```

[1] Priority order can also be assigned to entry queues and the operation of the **`select`** statement. This section will, however, focus on task priorities and protected object ceiling priorities.

```
  pragma Priority(Task_Priority);
end Servers;
```

For protected objects acting as interrupt handlers, a special pragma is defined:

```
pragma Interrupt_Priority(Expression);
```

or simply

```
pragma Interrupt_Priority;
```

The definition, and use, of a different pragma for interrupt levels improves the readability of programs and helps to remove errors that can occur if task and interrupt priority levels are confused. However, the expression used in `Interrupt_Priority` evaluates down to `Any_Priority`, and hence it is possible to give a relatively low priority to an interrupt handler. If the expression is actually missing, the highest possible priority is assigned (i.e. `Any_Priority'Last`).

A priority assigned using one of these pragmas is called a **base priority**. A task may also have an **active priority** that may be higher where the `FIFO_Within_Priority` dispatching policy is being used – this will be explained in due course.

The main program, which is executed by a notional environmental task, can have its priority set by placing the `Priority` pragma in the main subprogram. If this is not done, the default value, defined in `System`, is used. Any other task that fails to use the pragma has a default base priority equal to the base priority of the task that created it.

In order to make use of the immediate ceiling priority protocol (ICPP), an Ada program must include the following pragma:

```
pragma Locking_Policy(Ceiling_Locking);
```

An implementation may define other locking policies; only `Ceiling_Locking` is required by the Real-Time Systems Annex. The default policy, if the pragma is missing, is implementation defined. To specify the ceiling priority for each protected object, the `Priority` and `Interrupt_Priority` pragmas defined earlier are used. If the pragma is missing, a ceiling of `System.Priority'Last` is assumed.

The exception `Program_Error` is raised if a task calls a protected object with a priority greater than the defined ceiling. If such a call were allowed, then this could result in the mutually exclusive protection of the object being violated. If it is an interrupt handler that calls in with an inappropriate priority, then the program becomes erroneous. This must ultimately be prevented through adequate testing and/or static analysis of the program.

With `Ceiling_Locking`, an effective implementation will use the thread of the calling task to execute not only the code of the protected call, but also the code of any other task that happens to have been released by the actions of the original call. For example, consider the following simple protected object:

```
protected Gate_Control is
  pragma Priority(28);
  entry Stop_And_Close;
  procedure Open;
```

```
private
  Gate: Boolean := False;
end Gate_Control;

protected body Gate_Control is
  entry Stop_And_Close when Gate is
  begin
    Gate := False;
  end Stop_And_Close;

  procedure Open is
  begin
    Gate := True;
  end Open;
end Gate_Control;
```

Assume a task T, priority 20, calls Stop_And_Close and is blocked. Later, task S (priority 27) calls Open. The thread that implements S will undertake the following actions.

(1) Execute the code of Open for S.
(2) Evaluate the barrier on the entry and note that T can now proceed.
(3) Execute the code of Stop_And_Close for T.
(4) Evaluate the barrier again.
(5) Continue with the execution of S after its call on the protected object.

As a result, there has been no context switch. The alternative is for S to make T runnable at point (2); T now has a higher priority (28) than S (27) and hence the system must switch to T to complete its execution within Gate_Control. As T leaves, a switch back to S is required. This is much more expensive.

As a task enters a protected object, its priority may rise above the base priority level defined by the Priority or Interrupt_Priority pragmas. The priority used to determine the order of dispatching is the **active priority** of a task. This active priority is, when the dispatching policy is FIFO_Within_Priority, the maximum of the task's base priority and any priority it has inherited.

The use of a protected object is one way in which a task can inherit a higher active priority. There are others, for example:

- During activation – a task will inherit the active priority of the parent task that created it; remember (from Section 4.4) the parent task is blocked waiting for its child task to complete, and this could be a source of priority inversion without this inheritance rule.
- During a rendezvous – the task executing the **accept** statement will inherit the active priority of the task making the entry call (if it is greater than its own priority).

Note that the last case does not necessarily remove all possible cases of priority inversion. Consider a server task, S, with entry E and base priority L (low). A high-priority task makes a call on E. Once the rendezvous has started, S will execute with the higher

priority, but before S reaches the accept statement for E, it will execute with priority L (even though the high-priority task is blocked). This, and other candidates for priority inheritance, can be supported by an implementation. The implementation must, however, provide a pragma that the user can employ to select the additional conditions explicitly.

The Real-Time Systems Annex attempts to provide usable, flexible but extensible features. Clearly, this is not easy. Ada 83 suffered from being too prescriptive. However, the lack of a defined dispatching policy would be unfortunate, as it would not assist software development or portability. Hence Ada 95 defined a single policy, FIFO_Within_Priority, in the Annex. Ada 2005 has added to this policy by defining a number of others – some of these are described later in this chapter. First, however, means of restricting the facilities available to the programmer are considered.

## 12.4 The Ada Ravenscar profile

The earlier chapters of this book have demonstrated the extensive set of facilities that Ada provides for the support of real-time and concurrent programming. The expressive power of the full language is clearly extensive. There are, however, situations in which a restricted set of features is desirable. This section looks at ways in which certain restrictions can be identified in an Ada program. It then describes in detail the Ravenscar Profile which is a collection of restrictions aimed at applications that require very efficient implementations, or have high integrity requirements, or both. It provides the minimum language features necessary to program fixed priority-based applications.

Where it is necessary to produce very efficient programs, it would be useful to have run-time systems (kernels) that are tailored to the particular needs of the program actually executing. As this would be impossible to do in general, the language defines a set of restrictions that a run-time system should recognize and 'reward' by giving more effective support. The following example restrictions are identified by the pragma called Restrictions, and are checked and enforced before run-time. Note, however, that there is no requirement on the run-time to tailor itself to the restrictions specified.

- **No_Task_Hierarchy** – all (non-environment) tasks depend directly on the environment task.
- **No_Abort_Statement** – there are no abort statements.
- **No_Terminate_Alternatives** – there are no selective_accepts with terminate alternatives.
- **No_Task_Allocators** – there are no allocators for task types or types containing task subcomponents.
- **No_Dynamic_Priorities** – there are no semantic dependencies on the package Dynamic_Priorities.
- **No_Dynamic_Attachments** – there is no call to any of the operations defined in package Interrupts, e.g. Attach_Handler.
- **No_Local_Protected_Objects** – protected objects shall only be declared at library level.
- **No_Local_Timing_Events** – timing events shall only be declared at library level.

via protected objects is amenable to analysis, the rendezvous and the consequential use of select statements is prohibited. Other restrictions are motivated by the need to have a small and efficient run-time support system – hence the use of abort statements and complex barriers is disallowed. The protocol required for implementing general protected objects is significantly simplified by the restriction of one entry per object and one queued task per entry. Non-static features such as the dynamic attachment of interrupt handlers and task-specific termination handlers are excluded as is the use of package `Calendar`; the real-time clock package is the preferred time base to use. Note that two of the restrictions, `No_Task_Termination` and `Max_Entry_Queue_Length => 1`, result in run-time checks (see the definitions of these restrictions).

An important point to emphasis about Ravenscar is that it is not a substitute for the full tasking model. It has significantly less expressive power, and many applications needs cannot be programmed with this profile.

To complete the definition, the pragma `Detect_Blocking` needs to be defined. Within a protected operation it is illegal to block, for example to call an external procedure that has a delay statement within it. Although illegal it is not possible for the compiler to check that this cannot occur, so it becomes a run-time issue. The program is said to experience a **bounded error** which can result in a number of possible behaviours, one of which is for the exception `Program_Error` to be raised. What the pragma `Detect_Blocking` does is ensure that the error condition is recognized and `Program_Error` raised. Although this checking will add to the overheads, i.e. detract from real-time performance, it is deemed necessary for high-integrity applications.

## 12.5 Dynamic priorities and other Ada facilities

The above discussions have assumed that the base priority of a task remains constant during the entire existence of the task. This is an adequate model for many scheduling approaches. There are, however, situations in which it is necessary to alter base priorities. For example:

- to implement mode changes;
- to implement application-specific scheduling schemes.

In the first example, base priority changes are infrequent and correspond to changes in the relative temporal properties of the tasks after a mode change (for example, a task running more frequently in the new mode). The second use of dynamic priorities allows programmers to construct their own schedulers. An example of this is the programming of execution-time servers that will be illustrated in the Section 13.6.3.

To support dynamic priorities, the language provides a library package – see Program 12.1. The function `Get_Priority` returns the current base priority of the task; this can be changed by the use of `Set_Priority`. A change of base priority takes effect immediately the task is outside a protected object.

Ada 2005 also allows the ceiling priority of a protected object to be changed at run-time. For any protected object, `P`, the attribute `P'Priority` represents a component of `P` of type `System.Any_Priority`. References to the `Priority` attribute can

**Program 12.1** Dynamic priority package.

```
with Ada.Task_Identification;
with System;
package Ada.Dynamic_Priorities is
  procedure Set_Priority(Priority : System.Any_Priority;
                T : Task_Identification.Task_Id :=
                Task_Identification.Current_Task);
    -- raises Program_Error if T is the Null_Task_Id
    -- has no effect if the task has terminated

   function Get_Priority(T : Task_Identification.Task_Id :=
            Task_Identification.Current_Task)
            return System.Any_Priority;
     -- raises Tasking_Error if the task has terminated
     -- or Program_Error if T is the Null_Task_Id
private
  -- not specified by the language
end Ada.Dynamic_Priorities;
```

only occur from within the body of the associated protected body. Such references can be read or write. If the locking policy `Ceiling_Locking` is in effect then a change to the `Priority` attribute results in the ceiling value changing to this new value – but only at the end of the protected action that resulted in the change.

As well as changing the priority of a task, which obviously affects the likelihood of it being scheduled, Ada also allows a task to suspend itself and to suspend other tasks. These two facilities are called **synchronous task control** (see Program 5.1) and **asynchronous task control**. The latter is supported by the following package:

```
with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control is
  procedure Hold(T : Task_Identification.Task_Id);
  procedure Continue(T : Task_Identification.Task_Id);
  function Is_Held(T : Task_Identification.Task_Id)
           return Boolean;
end Ada.Asynchronous_Task_Control;
```

For each processor, there is a conceptual idle task which can always run (but has a priority below any application task). A call of `Hold` lowers the base priority of the designated task to below that of the idle task. It is said to be *held*. If the designated task is not executing with an inherited priority, it will be suspended immediately. A call of `Continue` restores the task's priority. This facility will be used in the next chapter to program a type of execution-time server.

Ada also provides other facilities which are useful for programming a wide variety of real-time systems that are scheduled using fixed priorities, for example, prioritized entry queues, task attributes, and so on. There are also a number of other dispatching policies. One for EDF scheduling is defined later in this chapter (Section 12.8). Another defines non-preemptive scheduling: `Non_Preemptive_FIFO_Within_Priorities`.

This has similar properties to the preemptive version other than the obvious requirement for a task to continue executing even when a higher-priority task becomes runnable. Interrupts can still occur, but task switching is postponed until the task itself executes a blocking operation such as a delay statement. Note the execution of 'delay 0.0' is sufficient to end a period of non-preemptive execution. A further dispatching policy defined in the real-time annex is Round-Robin scheduling. This is the usual algorithm where tasks of the same priority are allocated a quantum of execution time before they are suspended and return to the back of the queue for that priority level. To ensure that the integrity of protected objects is not compromised by this policy then suspension is itself suspended whilst a task executes within such objects.

As well as supporting a number of distinct dispatching policies Ada also allows mixed systems to be specified. This is described later in this chapter once EDF scheduling has been defined (see Section 12.9).

The reader is referred to the Systems Programming and Real-Time Annexes of the Ada Reference Manual and the Further Reading section of this chapter for details on all Ada facilities in the area of real-time programming.

## 12.6 C/Real-Time POSIX and fixed-priority scheduling

C/Real-Time POSIX consists of a variety of related standards. There is the base standard, the real-time extensions, the threads extensions and so on. If implemented in a single system, it would contain a huge amount of software. To help produce more compact versions of operating systems which conform to the C/Real-Time POSIX specifications, a set of application environment **profiles** have been developed, the idea being that implementors can support one or more profiles. For real-time systems, four profiles have been defined in C/Real-Time POSIX:

- **PSE51** – minimal real-time system profile – intended for small single/multiprocessor embedded systems controlling one or more external devices; no operator interaction is required and there is no file system. The main services are threads, fixed-priority scheduling, mutexes with priority inheritance, condition variables, semaphores, simple device I/O and signals. It is analogous to the Ravenscar Profile.
- **PSE52** – real-time controller system profile – an extended PSE51 for potentially multiple processors with a file system interface, message queues and tracing facilities.
- **PSE53** – dedicated real-time system profile – an extension of PSE52 for single or multiprocessor systems; includes multiple multithreaded processes and asynchronous I/O.
- **PSE54** – multipurpose real-time system profile – capable of running a mix of real-time and non real-time processes executing on single/multiprocessor systems with memory management units, mass storage devices, networks and so on.

In general, a C/Real-Time POSIX system is also free not to support any of the optional units of functionality it chooses, and so much finer control over the supported

functionality is possible. All of the real-time and the thread extensions are optional. However, conforming to one of the profiles means that all the required units of functionality must be supported.

To support priority-based scheduling, C/Real-Time POSIX has options to support priority inheritance and ceiling protocols. Priorities may be set dynamically. Within the priority-based facilities, there are four policies.

- **FIFO** – a process/thread runs until it completes or it is blocked; if a process/thread is preempted by a higher-priority process/thread then it is placed at the head of the run queue for its priority.
- **Round-Robin** – a process/thread runs until it completes or it is blocked or its time quantum has expired; if a process/thread is preempted by a higher-priority process then it is placed at the head of the run queue for its priority; however, if its quantum expires it is placed at the back.
- **Sporadic Server** – a process/thread runs as a sporadic server (see Section 11.6.2).
- **OTHER** – an implementation-defined policy (which must be documented).

For each scheduling policy, there is a minimum range of priorities that must be supported; for FIFO and round-robin, this must be at least 32. The scheduling policy can be set on a per process and a per thread basis.

Threads may be created with a 'system contention' option, in which case they compete with other system threads according to their policy and priority. Alternatively, threads can be created with a 'process contention' option; in this case they must compete with other threads (created with a process contention) in the parent process. It is unspecified how such threads are scheduled relative to threads in other processes or to threads with global contention. A specific implementation must decide whether to support 'system contention' or 'process contention' or both.

Programs 12.2 and 12.3 illustrate the C interface to the Real-Time POSIX scheduling facilities. The functions are divided into those which manipulate a process's scheduling policy and parameters, and those that manipulate a thread's scheduling policy and parameters. If a thread modifies the policy and parameters of its owning process, the effect on the thread will depend upon its contention scope (or level). If it is contending at a system level, the change will not affect the thread. If, however, it is contending at the process level, there will be an impact on the thread.

In order to prevent priority inversion, C/Real-Time POSIX allows inheritance protocols to be associated with mutex variables. As well as catering for basic priority inheritance, the immediate ceiling priority protocol (called the priority protect protocol by C/Real-Time POSIX) is also supported.

C/Real-Time POSIX provides other facilities that are useful for real-time systems. For example, it allows:

- message queues to be priority ordered;
- functions for dynamically getting and setting a thread's priority;
- threads to indicate whether their attributes should be inherited by any child thread they create.

**Program 12.2** The C/Real-Time POSIX interface to the Real-Time POSIX scheduling facilities.

```
#define SCHED_FIFO ...      /* preemptive priority scheduling */
#define SCHED_RR ...        /* preemptive priority with quantum */
#define SCHED_SPORADIC ... /* sporadic server */
#define SCHED_OTHER ...     /* implementation-defined scheduler */
#define PTHREAD_SCOPE_SYSTEM ... /* system-wide contention */
#define PTHREAD_SCOPE_PROCESS ...  /* local contention */
#define PTHREAD_PRIO_NONE ... /* no priority inheritance */
#define PTHREAD_PRIO_INHERIT ... /* basic priority inheritance */
#define PTHREAD_PRIO_PROTECT ... /* ICPP */

typedef ... pid_t;

struct sched_param {
  ...
  int sched_priority; /* used for SCHED_FIFO and SCHED_RR */
  ... };

int sched_setparam(pid_t pid, const struct sched_param *param);
int sched_getparam(pid_t pid, struct sched_param *param);
/* set/get the scheduling parameters of process pid */

int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
int sched_getscheduler(pid_t pid);
/* set/get the scheduling policy of process pid */

int sched_yield(void);
/* causes the current thread/process to be placed at the back */
/* of the run queue */

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
/* returns the max/minimum priority for the policy specified */

int sched_rr_get_interval(pid_t pid, struct timespec *t);
/* if pid /= 0, the time quantum for the calling process/thread is
   set in the structure referenced by t
   if pid = 0, the calling process/threads time quantum is set
   in the structure pointed to by t
 */
```

## 12.7 Real-Time Java and fixed-priority scheduling

Real-Time Java has the notion of a schedulable object. This is any object that supports the `Schedulable` interface given in Program 12.4. The classes `RealtimeThread`, `NoHeapRealTimeThread` and `AsyncEventHandler` all support this interface. Objects of these classes all have scheduling parameters (see Program 12.5). Real-Time Java implementations are required to support at least 28 real-time priority levels. As with

**Program 12.3** The C/Real-Time POSIX interface to the Real-Time POSIX scheduling facilities continued.

```
int pthread_attr_setscope(pthread_attr_t *attr,
                          int contentionscope);
int pthread_attr_getscope(const pthread_attr_t *attr,
                          int *contentionscope);
/* set/get the contention scope attribute for a */
/* thread attribute object */

int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                                int *policy);
/* set/get the scheduling policy attribute for a */
/* thread attribute object */

int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                               struct sched_param *param);
/* set/get the scheduling policy attribute for a */
/* thread attribute object */

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                  int protocol);
int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr,
                                  int *protocol);
/* set/get the priority inheritance protocol */

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                     int prioceiling);
int pthread_mutexattr_getprioceiling(pthread_mutexattr_t *attr,
                                     int *prioceiling);
/* set/get the priority ceiling */

/* All the above integer functions return 0 if successful */
```

Ada and C/Real-Time POSIX, the larger the integer value, the higher the priority (and, therefore, the greater the execution eligibility). Non-real-time threads are given priority levels below the minimum real-time priority. Note, scheduling parameters are bound to threads at thread creation time (see Program 10.1). If the parameter objects are changed, they have an immediate impact on the associated threads.

In common with Ada and C/Real-Time POSIX, Real-Time Java supports a preemptive priority-based dispatching policy. However, unlike Ada and C/Real-Time POSIX, Real-Time Java does not require (but does recommend) a preempted thread to be placed at the head of the run queue associated with its priority level. Real-Time Java also supports a high-level scheduler whose goals are to decide whether to admit new schedulable objects according to the resources available and a feasibility algorithm. Hence, while Ada and C/Real-Time POSIX focus on static offline schedulability analysis, Real-Time Java addresses more dynamic systems with the potential for online analysis.

**Program 12.4** An extract of the Real-Time Java `Schedulable` interface.

```
public interface Schedulable extends java.lang.Runnable {
  ...
  public void addToFeasibility();
  public void removeFromFeasibility();

  public MemoryParameters getMemoryParameters();
  public void setMemoryParameters(MemoryParameters memory);

  public ReleaseParameters getReleaseParameters();
  public void setReleaseParameters(ReleaseParameters release);

  public SchedulingParameters getSchedulingParameters();
  public void setSchedulingParameters(
          SchedulingParameters scheduling);

  public Scheduler getScheduler();
  public void setScheduler(Scheduler scheduler);
}
```

**Program 12.5** The Real-Time Java `SchedulingParameters` class and its subclasses.

```
public abstract class SchedulingParameters {
  public SchedulingParameters();
}

public class PriorityParameters extends SchedulingParameters {
  public PriorityParameters(int priority);

  public int getPriority();
  public void setPriority(int priority) throws
                  IllegalArgumentException;
  ...
}

public class ImportanceParameters extends PriorityParameters {
  public ImportanceParameters(int priority, int importance);
  public int getImportance();
  public void setImportance(int importance);
  ...
}
```

The `Scheduler` abstract class is given in Program 12.6. The `isFeasible` method considers only the set of schedulable objects that have been added to its feasibility list (via the `addToFeasibility` and `removeFromFeasibility` methods). The method `changeIfFeasible` checks to see if its set of objects is still feasible if the given object has its release and memory parameters changed. If it is, the parameters are changed. Static methods allow the default scheduler to be queried or set.

**Program 12.6** The Real-Time Java `Scheduler` class.

```
public abstract class Scheduler {
  protected Scheduler();

  protected abstract void addToFeasibility(
                    Schedulable schedulable);
  protected abstract void removeFromFeasibility(
                    Schedulable schedulable);

  public abstract boolean isFeasible();
  // checks the current set of schedulable objects

  public boolean changeIfFeasible(Schedulable schedulable,
                ReleaseParameters release,
                MemoryParameters memory);

  public static Scheduler getDefaultScheduler();
  public static void setDefaultScheduler(Scheduler scheduler);

  public abstract java.lang.String getPolicyName();
}
```

One defined subclass of the `Scheduler` class is the `PriorityScheduler` class (defined in Program 12.7), which implements standard preemptive priority-based scheduling.

*Again, it should be stressed that Real-Time Java does not require an implementation to provide an online feasibility algorithm.* The default algorithm assumes an adequately fast computer. Consequently, it returns `true` – if the application contains only periodic and sporadic schedulable objects; `false` otherwise (that is, if aperiodic schedulable objects are present).

Real-Time Java allows priority inheritance algorithms to be used when accessing synchronized classes. To achieve this, three classes are defined (see Program 12.8). Consider the following class:

```
public class SynchronizeClass {
  public void Method1() { ... };
  public void Method2() { ... };
}
```

An instance of this class can have its control protocol set to immediate priority ceiling inheritance (called priority ceiling emulation by Real-Time Java) with a priority of 10 by the following code:

```
SynchronizeClass SC = new SynchronizeClass();
PriorityCeilingEmulation PCI = new PriorityCeilingEmulation(10);
...
MonitorControl.setMonitorControl(SC, PCI);
```

It is important to note that all queues in Real-Time Java are priority ordered.

**Program 12.7** The Real-Time Java `PriorityScheduler` class.

```
class PriorityScheduler extends Scheduler {
  public PriorityScheduler()

  protected void addToFeasibility(Schedulable s);
  protected void removeFromFeasibility(Schedulable s);

  public boolean isFeasible();
  // checks the current set of schedulable objects

  public boolean changeIfFeasible(Schedulable schedulable,
                ReleaseParameters release, MemoryParameters memory);

  protected void addToFeasibility(Schedulable s);
  protected void removeFromFeasibility(Schedulable s);

  public void fireSchedulable(Schedulable schedulable);

  public int getMaxPriority();
  public int getMinPriority();
  public int getNormPriority();
  public java.lang.String getPolicyName();

  public static PriorityScheduler instance();

  ...
}
```

Like Ada and C/Real-Time POSIX, Real-Time Java has very comprehensive real-time support facilities. Similarly, profiles have been defined. Three such profiles are those developed by the JSR 302 Expert Group.[2] Their goal is to define subsets of Real-Time Java for use in safety-critical systems. This requires a much tighter and smaller set of Java virtual machines and libraries with much more precise performance requirements. The following three profiles have been defined – here the focus is on the scheduling models rather than the memory management models.

- **Level 0** – a Level 0 application's programming model is similar to a cyclic executive model as defined in Section 12.1. A Level 0 application's schedulable objects shall consist only of a set of periodic asynchronous event handlers. Each handler has a period, priority and start time relative to the beginning of a major cycle. A schedule of all handlers is constructed by either the application designer or by an offline tool provided with the implementation. All handlers in a Level 0 implementation execute using a single server thread.
- **Level 1** – a Level 1 application uses a fixed-priority programming model with a set of concurrent computations, each with a priority, running under the control of

[2] See http://jcp.org/en/jsr/detail?id=302.

**Program 12.8** Real-Time Java classes supporting priority inheritance.

```
public abstract class MonitorControl {
  public MonitorControl();

  public static void setMonitorControl(MonitorControl policy);
  // set the default

  public static void setMonitorControl(java.lang.Object monitor,
                                       MonitorControl policy);
  // sets an individual objects policy
}

public class PriorityCeilingEmulation extends MonitorControl {
  public PriorityCeilingEmulation(int ceiling);

  public int getDefaultCeiling();
  // get ceiling for this object
}

public class PriorityInheritance extends MonitorControl {
  public PriorityInheritance();

  public static PriorityInheritance instance();
}
```

a fixed-priority preemptive scheduler. The computations are performed in a set of periodic and sporadic event handlers. Only mutual exclusion is supported; there is no support for condition synchronization (that is, the use of the `wait` and `notify` methods is prohibited).

- **Level 2** – computation at Level 2 is performed in a set of asynchronous event handlers and/or `NoHeapRealtimethreads`. A Level 2 application may use the `wait` and `notify` methods.

As indicated above, the profiles also address the use of dynamic memory allocation. The Ada and C/Real-Time POSIX profiles are silent on this issue.

## 12.8 Programming EDF systems

To support the programming of applications that wish to make use of EDF scheduling, three language features are required:

- a formal representation of the deadline of a task;
- use of deadlines to control dispatching;
- a means of sharing data between tasks that is compatible with EDF.

**Program 12.9** The EDF dispatching package.

```
with Ada.Real_Time; with Ada.Task_Identification;

package Ada.Dispatching.EDF is
  subtype Deadline is Ada.Real_Time.Time;
  Default_Deadline : constant Deadline :=
              Ada.Real_Time.Time_Last;
  procedure Set_Deadline(D : in Deadline;
            T : in Ada.Task_Identification.Task_ID :=
            Ada.Task_Identification.Current_Task);
  procedure Delay_Until_And_Set_Deadline(
            Delay_Until_Time : in Ada.Real_Time.Time;
            TS : in Ada.Real_Time.Time_Span);
  function Get_Deadline(T : in Ada.Task_Identification.Task_ID :=
           Ada.Task_Identification.Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

Of all the mainstream engineering languages discussed in this book only Ada gives direct support to EDF scheduling. Both C/Real-Time POSIX and Real-Time Java allow implementations to support EDF scheduling, but not in a portable way. This section will therefore only discuss Ada, and will cover the three feature just identified. Note Ada's support for deadline-based scheduling does not extend to any notion of deadline inheritance.

### 12.8.1 Representing deadlines in Ada

A task's deadline can be set using the facilities defined in Program 12.9.

The identifier `Deadline` is explicitly introduced even though it is a direct subtype of the time type from `Ada.Real_Time`.

The `Set` and `Get` subprograms have obvious utility. A call of `Delay_Until_And_Set_Deadline` delays the calling task until time `Delay_Until_Time`. When the task becomes runnable again it will have deadline `Delay_Until_Time + TS`. The inclusion of this procedure reflects a common task structure for periodic activity. Consider the example of a periodic task; now assume it has a deadline at the end of its execution (i.e. every time it executes it should finish before its next release):

```
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Dispatching.EDF; use Ada.Dispatching.EDF;
...

  task Periodic_Task;

  task body Periodic_Task is
    Interval : Time_Span := Milliseconds(30);
    -- define the period of the task, 30ms in this example
    Next : Time;
  begin
    Next := Clock;  -- start time
    Set_Deadline(Clock+Interval);
```

```
    loop
      -- undertake the work of the task
      Next := Next + Interval;
      Delay_Until_And_Set_Deadline(Next,Interval);
    end loop;
  end Periodic_Task;
```

If, rather than using just this one procedure call, the task had a set deadline call and a delay until statement then this would most likely result in an extra unwanted task switch (first the deadline is extended and hence a more urgent task will preempt, later the task will execute again just to put itself on the delay queue).

With EDF, all dispatching decisions are based on deadlines, and hence it is necessary for a task to always have a deadline.[3] However, a task must progress though activation before it can get to a position to call `Set_Deadline` and hence a default deadline value is given to all tasks (`Default_Deadline` defined in `Ada.Dispatching.EDF`). However, this default value is well into the future and hence activation will take place with very low urgency (all other task executions will occur before this task's activation). If more urgency is required, the following language-defined pragma is available for inclusion in a task's specification (only):

```
pragma Relative_Deadline(Relative_Deadline_Expression);
```

where the type of the parameter is `Ada.Real_Time.Time_Span`. The initial absolute deadline of a task containing this pragma is the value of `Ada.Real_Time.Clock + Relative_Deadline_Expression`, the call of the clock being made between task creation and the start of its activation.

So the example above should use this pragma rather than include the first call of `Set_Deadline`:

```
task Periodic_Task is
  pragma Relative_Deadline(Milliseconds(30));
end Periodic_Task;
```

A final point to note with the deadline assignment routine concerns when it will take effect. The usual result of a call to `Set_Deadline` is for the associated deadline to be increased into the future and that a task switch is then likely (if EDF dispatching is in force). As this would not be appropriate if the task is currently executing within a protected object the setting of a task's deadline to the new value takes place as soon as is practical but not while the task is performing a protected action. This is similar to the rule that applies to changes to the base priority of a task using the dynamic priority facility.

Once a deadline is set then it becomes possible to check at run-time that the program's execution does indeed meet its deadlines. This is easily accommodated by the asynchronous transfer of control (select-then-abort) feature:

```
loop
  select
    delay until Ada.Dispatching.EDF.Get_Deadline;
```

[3]This is one of the criticisms of EDF – even tasks that have no actual deadline must be given an artificial one so that they will be scheduled.

```
    -- action to be take when deadline missed
  then abort
    -- code
  end select;
end loop;
```

A fuller description of the options available to the programmer when a deadline is missed is given in Chapter 13.

### 12.8.2 Dispatching

To request EDF dispatching, the following use of the dispatching policy pragma is supported:

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

The main result of employing this policy is that the ready queues are now ordered according to deadline (not FIFO). Each ready queue is still associated with a single priority, but at the head of any ready queue is the runnable task with the earliest deadline. Whenever a task is added to a ready queue for its priority it is placed in the position dictated by its current absolute deadline.

The active priority of a task, under EDF dispatching, is no longer directly linked to the base priority of the task. The rules for computing the active priority of a task are somewhat complex and are covered in the next section – they are derived from consideration of each task's use of protected objects. For a simple program with no protected objects the following straightforward rules apply:

- any priorities set by the tasks are ignored;
- all tasks are always placed on the ready queue for priority value `System.Priority'First`.[4]

Hence only one ready queue is used (and that queue is ordered by deadline).

Of course real programs require task interactions, and for Ada real-time programs this usually means the use of protected objects. To complete the definition of the EDF dispatching policy the rules for using protected objects must be considered. This is outlined below.

### 12.8.3 EDF and Baker's algorithm

In Section 11.11.4, Baker's algorithm was introduced as a means of controlling access to shared objects when EDF scheduling was in force. Ada supports Baker's approach by:

- using deadline to represent urgency;
- using base priority to represent each task's preemption level;

[4]If EDF dispatching is defined just for a range of priorities then this priority value is the initial (lowest) value in that range – see Section 12.9.

- using ceiling priorities to represent preemption levels for protected objects;
- using standard `Ceiling_Locking` for access to protected objects.

So tasks have a base priority, but this is not used directly to control dispatching. EDF controls dispatching; the base priority only defines a preemption level.

Baker's algorithm states that a newly released task, T1 say, preempts the currently running task, T2, if and only if:

- the deadline of T1 is earlier than the deadline of T2; and
- the preemption level of T1 is higher than the preemption of any locked protected object (i.e. protected objects that are currently in use by any other task).

To keep track of all locked protected objects is an implementation overhead and hence the rules defined for Ada have the following form. Remember that if `EDF_Across_Priorities` is defined then all ready queues within the range `Priority'First .. Priority'Last` are ordered by deadline. Now rather than always place tasks in the queue for `Priority'First` the following rules apply:

- Whenever a task T is added to a ready queue, other than when it is preempted, it is placed on the ready queue with the highest priority R, if one exists, such that:
  - another task, S, is executing within a protected object with ceiling priority R; and
  - task T has an earlier deadline than task S; and
  - the base priority (preemption level) of task T is greater than R.

  If no such ready queue exists the task is added to the ready queue for `Priority'First`.
- When a task is chosen for execution it runs with the active priority of the ready queue from which the task was taken. If it inherits a higher active priority it will return to its original active priority when it no longer inherits the higher level.

It follows that if no protected objects are in use at the time of the release of T then T will be placed in ready queue at level `Priority'First` at the position dictated by its deadline.

A potential task switch occurs for the currently running task T whenever:

- a change to the deadline of T takes effect; or
- a decrease to the deadline of any task on a ready queue for that processor takes effect and the new deadline is earlier than that of the running task; or
- there is a non-empty ready queue for that processor with a higher priority than the priority of the running task.

So dispatching is preemptive, but it may not be clear that the above rules implement Baker's algorithm. Consider four scenarios. Remember in all of these behaviours, the running task is always returned to its ready queue whenever a task arrives. A task (possibly the same task) is then chosen to become the running task following the rules defined above.

| Task | Relative deadline $D$ | Preemption level $L$ | Uses resources | Arrives at time | Absolute deadline $A$ |
|---|---|---|---|---|---|
| T1 | 100 | 1 | R1,R3 | 0 | 100 |
| T2 | 80 | 2 | R2,R3 | 2 | 82 |
| T3 | 60 | 3 | R2 | 4 | 64 |
| T4 | 40 | 4 | R1 | 8 | 48 |

**Table 12.1** A task set (time attributes in milliseconds).

| Protected object | Ceiling value |
|---|---|
| R1 | 4 |
| R2 | 3 |
| R3 | 2 |

**Table 12.2** Ceiling values.

The system contains four tasks; T1, T2, T3 and T4; and three resources that are implemented as protected objects: R1, R2 and R3. Table 12.1 defines the parameters of these entities.

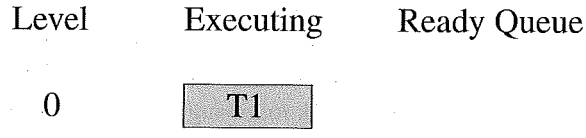
Consider just a single invocation of each task. The arrival times have been chosen so that the tasks arrive in order of lowest preemption level task first, etc. Assume all computation times are sufficient to cause the executions to overlap.

The resources are all used by more than one task, but only one at a time and hence the ceiling values of the resources are straightforward to calculate. For R1, it is used by T1 and T4; hence the ceiling preemption level is 4. For R2, it is used by T2 and T3; hence the ceiling value is 3. Finally, for R3, it is used by T1 and T2; the ceiling equals 2 (see Table 12.2).

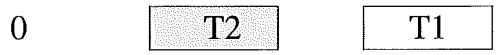
To implement this set of tasks and resources will require ready queues at level 0 (value of `Priority'First` in this example) and values 2, 3 and 4.

### Scenario 1

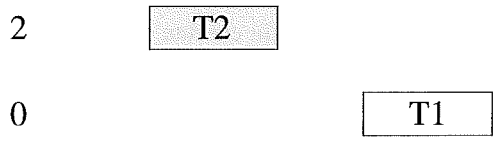
At time 0, T1 arrives. All ready queues are empty and all resources are free so T1 is placed in queue 0. It becomes the running/executing task. This is illustrated in the following where 'Level' is the priority level, 'executing' is the name of the task that is currently executing, and 'Ready Queue' shows the other executable tasks in the system. Again remember that the executing task is not on a ready queue, but is returned to a ready queue before any dispatching decision is taken.



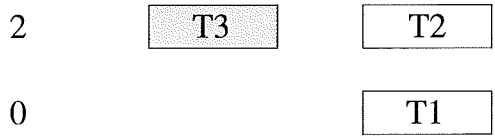
At time 2, T2 arrives and is added to ready queue 0 in front of T1 as it has a shorter absolute deadline. Now T2 is chosen for execution.



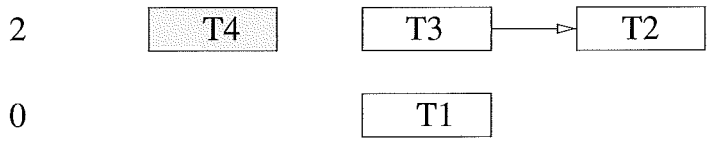

Assume at time 3, T2 calls R3. Its active priority will rise to 2.




At time 4, T3 arrives. Task T2 is joined by T3 on queue 2, as T3 has an earlier deadline and a higher preemption level; T3 is at the head of this queue and becomes the running task.




At time 8, T4 arrives. Tasks T3 and T2 are now joined by T4 as it has a deadline earlier than T3 and a higher preemption level (than 2). Task T4 now becomes the running task, and will execute until it completes; any calls it makes on resource R1 will be allowed immediately as this resource is free.

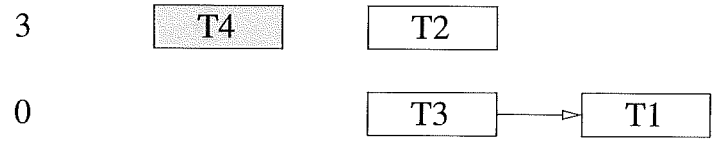



At some time later, T4 will complete then T3 will execute (at priority 2, or 4 if it locks R2) then when it completes, T2 will execute (also at priority 2) until it releases resource R3, at which point its priority will drop to 0 but it will continue to execute. Eventually when T2 completes, T1 will resume (initially at priority 0 – but this will rise if it accesses either of the resources it uses).

### Scenario 2

Here we make the simple change that, at time 3, T2 calls R2 instead of R3. Its active priority will rise to 3. Now when T3 arrives at time 4, it will not have a high enough preemption level to join ready queue 3 and will be placed on the lowest queue at level 0 (but ahead of T1). Task T2 continues to execute.

At time 8, T4 arrives. It passes both elements of the test and is placed on the queue at level 3 ahead of T2 and therefore preempts it.

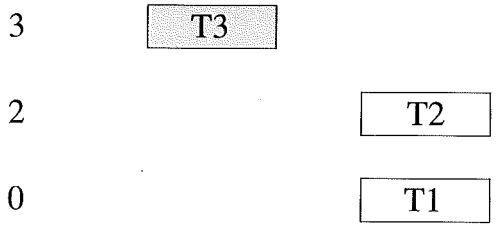



At some time later, T4 will complete then T2 will execute (at priority 3) until it releases resource R2, at which point its priority will drop to 0. Now T3 will preempt and becomes the running task.

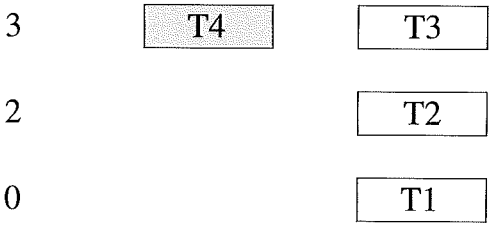
### Scenario 3

For another example, return to the first scenario but assume T3 makes use of resource R2 before T4 arrives.

- At time 0, T1 arrives. All ready queues are empty and all resources are free so T1 is placed in queue 0. It becomes the running task.
- At time 2, T2 arrives and is added to ready queue 0 in front of T1.
- Assume at time 3, T2 calls R3. Its active priority will rise to 2.
- At time 4, T3 arrives and becomes the running task at priority level 2.
- At time 5, T3 calls R2 (note all resource requests are always to resources that are currently free) and thus its active priority rises to 3.

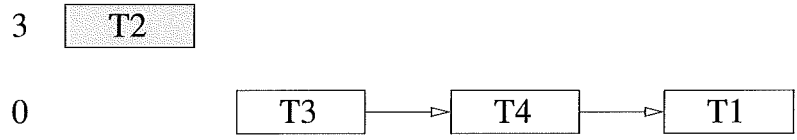



- At time 8, T4 arrives. There is now one task on queue 0; one on queue 2 (T2 holding resource R3) and one task on queue 3 (T3 holding R2). The highest ready queue that the dispatch rules determines is that at level 3, and hence T4 joins T3 on this queue – but at the head and hence becomes the running task.



### Scenario 4

Now consider a simple change to the parameters – let the relative deadline of T4 be 58 so thats its absolute deadline is 66 which is later than T3's absolute deadline. At time 3, T2 calls R2. Its active priority will rise to 3. Now when T3 arrives at time 4, it will not have a high enough preemption level to join ready queue 3 and will be placed on the lowest queue at level 0 (but ahead of T1). Task T2 continues to execute. At time 8, T4 arrives but will fail the preemption rule as its deadline is not earlier than T3's. T4 will be placed on the level 0 queue between T3 and T1.




Task T2 will continue until it completes its execution in the protected object. Its priority will then fall to 0 and T3 will preempt it. All tasks now execute in deadline order.

### Final note

The above rules and descriptions are, unfortunately, not quite complete. The ready queue for `Priority'First` plays a central role in the model as it is, in some senses, the default queue. If a task is not entitled to be put in a higher queue, or if no protected objects are in use, then it is placed in this base queue. Indeed during the execution of a typical program most runnable tasks will be in the `Priority'First` ready queue most of the time. However the protocol only works if there are no protected objects with a ceiling at the `Priority'First` level. Such ceiling values must be prohibited, but this is not a significant constraint.

## 12.8.4 Example of an EDF program

Recap what the programmer needs to do to use EDF. First, preemption levels, are chosen for each task based on the temporal properties of the task. Optimally, preemption levels will be assigned in reverse (relative) deadline order. Preemption levels are assigned to tasks using the 'priority' attribute of each task. These will be *exactly* the same assignments that would be needed with fixed-priority assignment and the deadline-monotonic priority algorithm. Next, shared protected objects are identified and ceiling values assigned. Again this is identical in the fixed-priority and EDF schemes. Finally, the required dispatching policy is asserted.

To illustrate the minimal changes that have to be made to the application code to move from one scheduling paradigm to another consider a simple periodic task scheduled according to the standard fixed priority method. This task has a period and deadline of 10 ms.

```
task Example is
  pragma Priority(5);
end Example;

task body Example is
  Next_Release : Ada.Real_Time.Time;
  Period : Ada.Real_Time.Time_Span
    := Ada.Real_Time.Milliseconds(10);
begin
  Next_Release := Ada.Real_Time.Clock;
  loop
    -- code
    Next_Release := Next_Release + Period;
    delay until Next_Release;
  end loop;
end Example;
```

If EDF dispatching is required (perhaps to make the system schedulable) then very little change is needed. The priority level of the task remains exactly the same. The task's code must, however, now explicitly refer to deadlines (the implicit deadline was always there as it was used to derive the value 5 for the task's priority):

```
task Example is
  pragma Priority(5);
  pragma Relative_Deadline(10); -- gives an initial relative
                                -- deadline of 10 milliseconds
end Example;

task body Example is
  Next_Release: Ada.Real_Time.Time;
  Period : Ada.Real_Time.Time_Span
    := Ada.Real_Time.Milliseconds(10);
begin
  Next_Release := Ada.Real_Time.Clock;
  loop
    -- code
    Next_Release := Next_Release + Period;
    Delay_Until_and_Set_Deadline(Next_Release, Period);
  end loop;
end Example;
```

Finally the dispatching policy must be changed from

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
```

to

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

No other changes are needed – any protected objects used by this or other tasks in the program retain their assigned ceilings.

## 12.9 Mixed scheduling

As was noted earlier, Ada allows a system with a combination of dispatching policies to be utilized. To accomplish this, the system's priority range is split into a number of distinct non-overlapping bands. In each band, a specified dispatching policy is in effect. The bands themselves are ordered by priority. So a runnable task in a high-priority band will take precedence over any other runnable task in a lower-priority band. For example, there could be a band of fixed priorities on top of a band of EDF with a single round-robin level for non-real-time tasks at the bottom. To illustrate this assume a priority range of 1..16:

```
pragma Priority_Specific_Dispatching
               (FIFO_Within_Priorities, 10, 16);

pragma Priority_Specific_Dispatching
               (EDF_Across_Priorities, 2, 9);

pragma Priority_Specific_Dispatching
               (Round_Robin_Within_Priorities, 1, 1);
```

Any task is assigned a dispatching policy by its virtue of its base priority. If a task has base priority 12 it will be dispatched according to the fixed-priority rules; if it has base priority 8 then it is under the control of the EDF rules. In a mixed system all tasks have a preemption level (it is just its base priority) and all tasks have a deadline (it will be `Default_Deadline` if none is assigned in the program). However, this deadline will have no effect if the task in not in an EDF band. With this example a runnable task with priority 14 will always execute before an 'EDF task' even if the 'EDF task' has an earlier deadline.

To achieve any mixture including one or more EDF bands, the properties assigned in the definition of EDF dispatching to level `Priority'First` need to be redefined to use the minimum value of whatever range is used for the EDF tasks. Also note that two adjacent EDF bands are not equivalent to one complete band. Runnable tasks in the upper bands will always take precedence over runnable tasks in the lower bands.

For completeness any priority value not included in the pragma is assumed to be `FIFO_Within_Priorities`. Any of the predefined policies can be mixed apart from the non-preemptive one. It is deemed incompatible to mix non-preemption with any other scheme as non-preemption is a system-wide property; a low-priority preemptive task would impact on a higher-priority task in another band.

Tasks within different bands can communicate using protected objects (and rendezvous if required). The use of `Ceiling_Locking` ensures that protected objects behave as required. For example an 'EDF task' (priority level 7) could share data with a 'round-robin' task (priority level 1) and a 'fixed-priority' task (priority 12). The protected object would have a ceiling value of (at least) 12. When the EDF task accesses the object its active priority will rise from 7 to 12, and while executing this protected action it will prevent any other task executing from within the EDF band. The 'round-robin' task will similarly execute with priority 12 – if its quantum is exhausted inside the object it will continue to execute until it has completed the protected action.

If a task changes its base priority at run-time (using the `Set_Priority` routine) then it may also change its dispatching group. For example, a task in the above illustration with base priority 7 that is changed to have priority 12 will move from EDF to fixed-priority scheduling.

Perhaps one minor weaknesses of Ada's support for mixed dispatching rules is that a task cannot directly ask under what policy it is being scheduled. However, a task can always find out its own priority (using `Get_Priority`) and from that use program constants to ascertain under which policy it is executing.

This ability to mix dispatching policies is unique to Ada. Experience will show whether this level of support for real-time programs proves to be useful, and if implementations are able to deliver this flexibility in an efficient manner.

## Summary

For scheduling theory to be employed in real applications it must be accessible from the programming languages used to implement the applications. Either the language must directly support the scheduling schemes or it must provide a means by which the application can access these schemes as they are supported by the underlying operating system.

The most mature scheduling theory is based upon preemptive fixed-priority dispatching. In this chapter the means by which Ada, C/Real-Time POSIX and Real-Time Java support this dispatching scheme have been described. A common set of primitives is evident in all three languages. This bears witness to the maturing of the theory.

However, fixed-priority dispatching is not the only scheduling scheme available. Earliest Deadline First (EDF) is one of a number of alternative schemes that are, theoretically, at least as important. Unfortunately EDF is not as easily available to application programmers. Unless a bespoke EDF-oriented operating system is constructed, the only way to make use of EDF is via the provisions now available in Ada. This chapter has therefore also described Ada's support for EDF scheduling. This has included discussions of its use of a variety of Baker's algorithms for controlling access to protected objects. Finally in this chapter the means by which mixed dispatching schemes (e.g. fixed-priority, EDF and round-robin) can be programmed in Ada have been included.

## Further reading

Burns, A. and Wellings, A.J. (2007) *Concurrent and Real-Time Programming in Ada 2005*. Cambridge: Cambridge University Press.

Butenhof, D. R. (1997) *Programming With Posix Threads*. Reading, MA: Addison-Wesley.

Hyde, P. (1999) *Java Thread Programming*. Indianapolis, IN: Sams Publishing.

Lea, D. (1999) *Concurrent Programming in Java: Design Principles and Patterns*. Reading, MA: Addison-Wesley.

Oaks, A. and Wong, H. (1997) *Java Threads*. Sebastopol, CA: O'Reilly.

Nichols, B., Buttlar, D. and Farrell, J. (1996) *POSIX Threads Programming*. Sebastopol, CA: O'Reilly.

Wellings, A.J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

## Exercises

**12.1** A real-time systems designer wishes to run a mixture of safety-critical, mission-critical and non-critical periodic and sporadic Ada tasks on the same processor. He or she is using preemptive priority-based scheduling and has used the response time analysis equation to predict that all tasks meet their deadlines. Give reasons why the system might nevertheless fail to meet its deadlines at run-time. What enhancements could be provided to the Ada run-time support systems to help eliminate the problems?

**12.2** Ada allows the base priority of a task to be set dynamically. Using the `Ada.Dynamic_Priorities` package, show how to implement a mode change protocol where a group of tasks must have their priorities changed as a single atomic operation.

**12.3** Explain the pros and cons of supporting dynamic ceiling priorities.

**12.4** Show how earliest deadline first scheduling can be implemented in Real-Time Java with a feasibility test of total process utilization less that 100%.