CHAPTER 12

Software Architectures for Real-Time Embedded Systems

Contents

To create architecture is to put in order. Put what in order? Function and objects.

Le Corbusier

Real-Time Embedded Systems. http://dx.doi.org/10.1016/B978-0-12-801507-0.00012-2

12.1 Real-Time Tasks

The term "task" is a design-time concept. A real-time system typically has multiple tasks, each of which represents a unit of concurrency.

In general, a task is simply a block of instructions (or actions) to be executed by a processor for a specific purpose. A task with real-time constraints is called a real-time task.

A task can be executed multiple times. Each individual run to completion of a task is called a *job* of that task. In such a sense, a task can also be treated as a stream of jobs of the same kind.

Following the classification in [50], we distinguish three types of real-time tasks.

(1) Periodic tasks: A periodic task is a stream of jobs, where the interarrival times between consecutive jobs are almost the same, and are called its period.
(2) Sporadic tasks: A sporadic task is a stream of jobs, where the interarrival times between consecutive jobs may differ widely, and can be arbitrarily small (in short, uneven spurts). A sporadic task is executed in response to events which occur at random instants of time, and the randomness is hard to be characterized by simple probability distribution functions. For example, via the touchpad of an embedded system, one day a person may issue service requests by following the piano notes of Beethoven's Fifth Symphony, but the other day the same person may issue service requests quite differently, say, by following the piano notes of Beethoven's Moonlight Sonata. Both pieces have turbulent yet invigorating parts with rapid progressions from note to note, leading to jobs released in bursts.
(3) Aperiodic tasks: An aperiodic task is a stream of jobs, where the interarrival times between consecutive jobs may follow a known probability distribution function. For example, the serial communication port of an embedded system may receive 3 data packets per second following the Poisson distribution; the interarrival times of the packets can follow the Gaussion distribution with a mean of 100 ms and a standard deviation of 8 ms.

Periodic tasks and sporadic tasks may have hard or soft timing constraints (deadlines). Aperiodic tasks typically have either soft or no deadlines. In the order of importance, periodic tasks with hard deadlines come first, followed by sporadic tasks with hard deadlines, periodic tasks with software deadlines, sporadic tasks with soft deadlines, and lastly aperiodic tasks with soft or no deadlines. For ease of schedulability analysis, if not otherwise specified, the deadlines specified for periodic tasks and sporadic tasks in this book are treated as hard constraints.

12.1.1 Worst-Case Task Execution Time

The execution time of a task (job) is the amount of time required to fully complete its execution, assuming that there is no other task (job) competing for resources. This value

highly depends on the speed of the processor on which the task is running, as well as the time-complexity of the task's instructions.

The actual amount of time to complete a job can vary for many reasons. Hardware features, such as cache and pipelining, can affect the actual execution time of consecutive jobs. Software structures, such as conditional and iterative code blocks, can also contribute to the deviation of the actual execution time. For instance, the instruction block of a task may contain conditional branches. Which branch to take depends on the run-time evaluation of the Boolean expression associated with the conditional instruction. Since different branches may take different amounts of time to finish, the actual execution time of one job can be different from that of another job.

For example, the task as shown on the left in Figure 12.1 has four execution paths, which take 40 ms, 35 ms, 30 ms, and 25 ms, respectively, to finish.[1] The execution time of the task on the

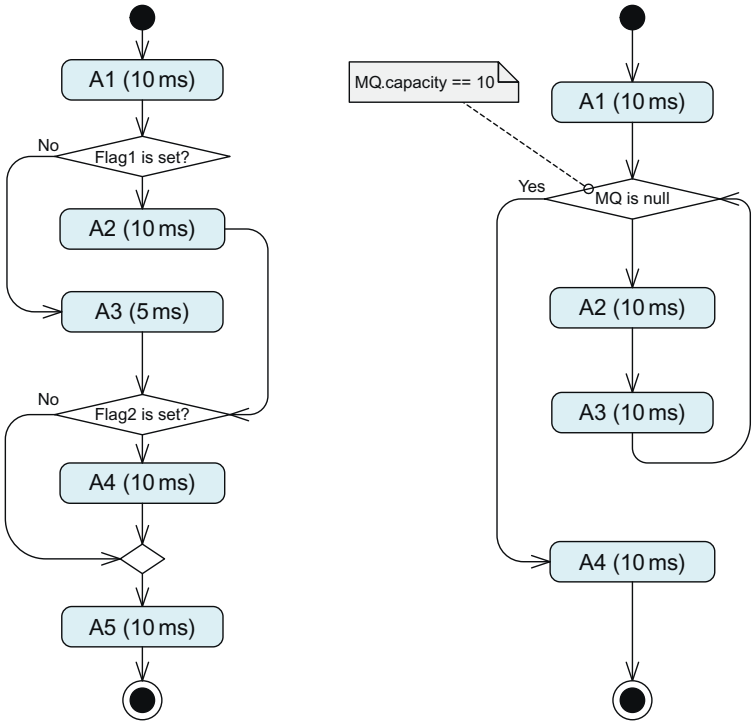

Figure 12.1
The task shown on the left has a worst-case execution time of 40 ms, and the task shown on the right has a worst-case execution time of 220 ms.

[1] These values again are estimated on the basis of the expected number of lines of code, the processor speed, the context switching time, the execution time of system calls, etc.

right can be 20 ms, 40 ms, 60 ms, and up to 220 ms, depending on how many messages are in the message queue.

Obviously, the maximum (or worst-case) execution time of the task shown on the left in Figure 12.1 is 40 ms, and the task on the right has a worst-case execution time of 220 ms (when the message queue is full).

In practice, the worst-case execution time of a task is often estimated beforehand, and is used in schedulability analysis at design time. This, undoubtedly, will lead to conservative design. However, such a design makes it possible to conduct thorough schedulability analysis prior to implementation, which is highly important for safety-critical systems.

12.1.2 Task Specification

A periodic task T_i is specified by a tuple (p_i, r_i, e_i, d_i), where

- p_i is the period, which is the length of inter-release times between two consecutive jobs;
- r_i is the release time (aka. phase), which is the instant of time at which the first job becomes available for execution;
- e_i is the worst-case execution time, indicating the demand for processing time;
- d_i is the deadline, which is the instant of time by which the execution of a job has to be finished.

Given a periodic task $T_i = (p_i, r_i, e_i, d_i)$, its jobs, by the order of occurrence, are denoted by J_{i1}, J_{i2}, etc. Normally, every job of a task T_i is released and becomes ready at the beginning of a period. In such a situation, the task is specified by $T_i = (p_i, e_i, d_i)$. When every job of a task T_i has to be completed by the end of its period, the task is specified by $T_i = (p_i, e_i)$.

For an aperiodic task, only the release time and the execution time are needed in schedulability analysis. Hence, an aperiodic task A_i is specified by a tuple (r_i, e_i).

For a sporadic task, its (hard) deadline also needs to be considered in schedulability analysis. A sporadic task S_i is specified by a tuple (r_i, e_i, d_i).

12.1.3 Task Timing Diagrams

A task timing diagram shows the state changes of one or more tasks over time.

A task timing diagram is given in Figure 12.2. The meanings of the timing attributes marked in Figure 12.2 are listed in Table 12.1.

The timing requirements given as an interval, such as *execution time*, *slack time*, *jitter time*, *rise time*, *initiation time*, *dwell time*, *fall time*, and *period*, are often called *duration*

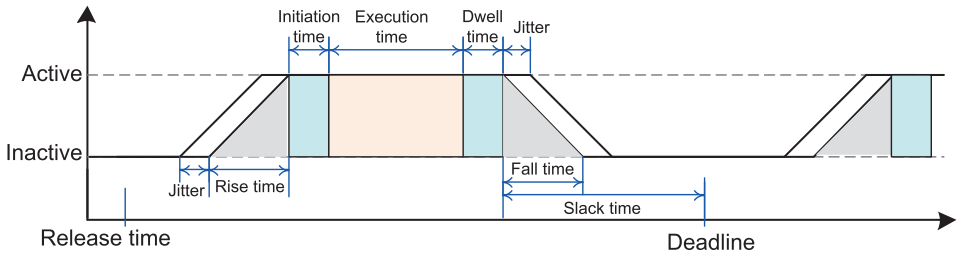

Figure 12.2
A task timing diagram.

constraints. The timing requirements given as an instant, such as *release time*, *deadline*, and state transition instant, are often called *instant constraints*. In Figure 12.2, instant constraints are marked as points on the time axis, and duration constraints are shown by bidirectional arrows with a delimiter line at each end. You may have noticed some similarities between **Table 11.2 and Table 12.1**. Actually a real-time task (or job) comprises a sequence of real-time actions, which may be at multiple levels of granularity. A timing property of a task, say, the execution time, can be the integration of one or more timing priorities of the subordinate actions.

Timing diagrams are very useful to depict the schedule design of a group of tasks. When timing diagrams are used in scheduling analysis, the rise time and fall time of a task are typically suppressed because they are at a much lower magnitude than the other measurements. The attributes such as initiation time and dwell time are also ignored or are treated as part of the execution time of a task.

Table 12.1 Timing information shown in timing diagrams

Attribute	Description
Period	The time interval indicating how often a task is to be performed
Release time	The time instant at which a task becomes eligible for execution
Deadline	The time instant by which a task should be completed
Rise time	The time needed to switch into a new task context
Fall time	The time needed to switch out of the last task context
Jitter time	The time variation for a transition or an event
Initiation time	The time needed to prepare for task execution, i.e., perform initialization actions
Execution time	The time for task execution
Dwell time	The time needed to round up the task execution, e.g., perform management actions (garbage collection). A deadline can be reached during dwell time
Slack time	The time between the end of dwell time and the task deadline

The timing diagram in Figure 12.3 shows the schedule of two periodic tasks and their timing constraints. Every job of the task $T_1 = (20, 6)$ is released at the beginning of a period, and its deadline is at the end of the corresponding period. For task $T_2 = (30, 4, 12, 24)$, its first job J_{21} is released at time 4 (relative to the start of the system), and its second job J_{22} is released at time 34. The inter-release time between jobs J_{21} and J_{22} is equal to the task period. Also note that in general the deadline of a job J_{ij} is calculated by $p_i(j-1) + d_i$.

Another example is given in Figure 12.4. The timing attributes of the three periodic tasks are shown at the top. Note that they are not part of the timing diagram; they are displayed simply for the convenience of the reader. The portion enclosed within the dashed box is the final task schedule design; again the dashed box is merely for clarity. Above the dashed box there are

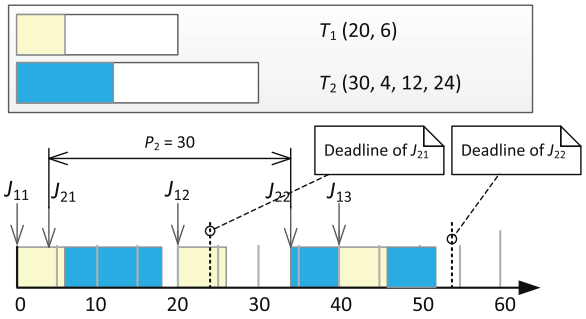

Figure 12.3
Example periodic tasks.

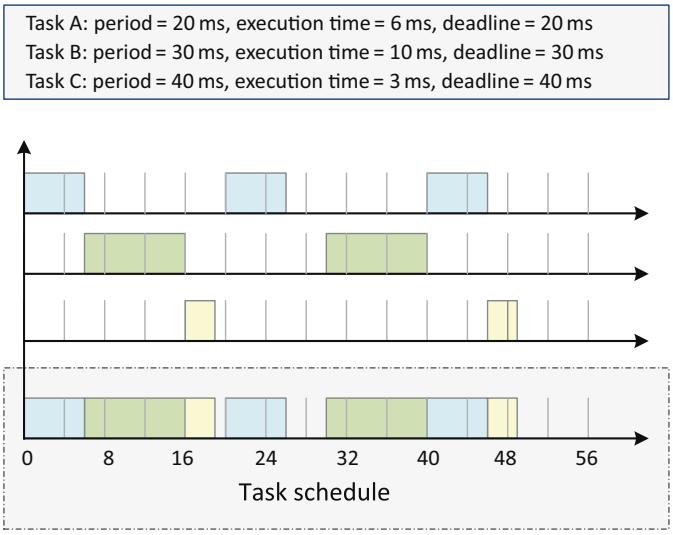

Figure 12.4
A timing diagram showing the design of a task schedule.

three time axes; the schedule of each task is plotted along the corresponding axis. For a system with only one processing unit, special care must be taken to make sure that all the tasks are scheduled interleavingly without breaking any of the timing constraints (task periods and deadlines). We will see many task scheduling diagrams as we learn real-time scheduling later.

12.1.4 Worst-Case Response Time

The performance of a real-time system is typically expressed in terms of the response time. The *response time* of a job is defined as the length of time from the release time of the job to the instant when it finishes.

The response time of a job depends heavily on its execution time, as well as on how jobs are scheduled by the system. For example, consider the job schedule given in Figure 12.5. All jobs of task T_1 are scheduled immediately after they are released; they have the same response time: 10 units of time. Each of the three jobs of task T_2 is scheduled immediately after the completion of a job of task T_1; they have the same response time: 25 units of time. For task T_3,

- its first job J_{31} is released at time 0, scheduled to run at time 25, preempted by job J_{12} at time 30. It resumes its execution at time 40, and finishes at time 55. The response time is thus 55 units of time.
- its second job J_{32} is released at time 90, scheduled to run at time 100, and finishes at time 120. The response time is thus $120 - 90 = 30$ units of time.

The *worst-case response time* of a task is the maximum value among the response times of all its jobs. For the example above, the worst-case response time of task T_3 is 55 units of time.

The concept of response time can be applied to asynchronous events as well. The response time of an event is defined as the length of time from the time instant when it is raised (released) to the instant when it is completely handled. In the rest of this chapter, we will use "worst-case response time" to evaluate several software architectures prevalently used for real-time embedded systems.

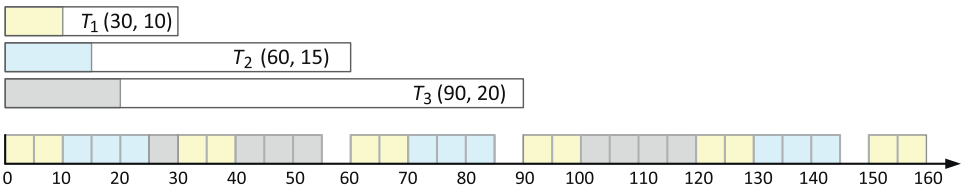

Figure 12.5
Worst-case response time.

12.1.5 Task Implementation

Each job of a task has a single flow of execution. Notice that we here use the word "flow" instead of "thread," because "thread" is an operating system (OS) concept and most real-time embedded systems are implemented without even using an OS.

In the rest of this chapter, we will study three software architectures for real-time embedded systems: round robin, round robin with interrupts, and queue based. In these architectures there is no "thread" concept; the execution flows of many tasks are sequentially organized such that the end of one job is followed by the start of another. In other words, if we overuse the word "thread" here, the whole system has only a single active thread all the time. The system executes those available jobs in a certain order, and repeats this pattern until the system is down.

A real-time OS (RTOS) is the most powerful, and maybe the most expensive, software architecture for real-time embedded systems. We will introduce the thread concept and study RTOS in Chapter 13. For systems built upon an OS, each task can be neatly implemented as a single thread.[2] The run-time behavior of an OS-based system with multiple threads would be in the hands of the OS scheduler, which typically offers acceptable performance to many applications.

12.2 Round-Robin Architecture

Round robin is a well-known principle that is widely adopted in many disciplines. For instance, round robin is one of the simplest scheduling algorithms implemented in many OSs. The rule is quite simple: everything (person, team, task, network node) can take its turn to equally compete for shared resources. As far as real-time embedded systems are concerned, the round-robin principle can be exploited to build software architectures for many simple systems [66].

Below we use a case study to examine its characteristics.

12.2.1 Case Study: Body Thermometer

A digital body thermometer is one of the simplest real-time embedded systems that you will find in almost every house. From a user's perspective, a body thermometer may have the following desired features:

[2] A thread must be rooted in a single *active* object that can not only receive events and dispatch them to the participating object(s) of the thread, but also generate its own stimulus events independently from the rest of the system. Thus, the active object of a thread is also called the executor of the embedded sequence of actions [34].

- a button that allows a user to turn on/off the thermometer;
- a display to show the current measure of body temperature;
- a buzzer that starts buzzing to indicate its readiness for user reading;
- a button that allows a user to stop the buzzing;
- a button that allows a user to switch between degree Fahrenheit (°F) mode and degree Celsius (°C) mode;
- a button that allows a user to start a new measure.

For better user experience, it is also a requirement that the body thermometer should respond to each button press within 1 s.

12.2.1.1 Hardware design

In order to meet the user requirements mentioned above, we consider a simple hardware design as specified below:

- The system has a microprocessor with an analog input interface.
- The system has a temperature sensor.
- The system has a status register `SSR` with a flag bit (*s_flag*) that indicates whether the sensor is on or off. It also has a data register `SDR` for storing the latest measure digitized from the analog input from the sensor.
- The system has an LCD for showing the current body temperature.
- The system has a buzzer circuit. It starts buzzing if the temperature measure has not changed for a while (say, 7 s).
- The system has a group of three pushbuttons. The left button allows a user to toggle between degree Fahrenheit mode and degree Celsius mode. The middle button allows a user to stop the buzzer, if it is buzzing, or to start a new measure, if the buzzer is not buzzing. The right button allows the user to turn on/off the thermometer.
- The three buttons share one status register `BSR`, containing flags that indicate the status of the buttons and the buzzer. The details are given in Figure 12.6. We assume that the flags *l_flag*, *m_flag*, and *r_flag* can be asserted by hardware when the corresponding button is pressed. The other two flags are handled by software.

12.2.1.2 Design of software architecture

As shown in Figure 12.7, a simple software architecture design for the body thermometer is specified in a UML activity diagram.

The system enters into a loop after initialization. The round-robin principle applies to the tasks (activities)[3] inside the loop: they are organized sequentially, taking turns to execute. Task descriptions are given in Table 12.2.

[3] In this example most of the activity steps are rather simple, and it is more appropriate to call them actions. However, we stick with the term 'task' within the context of architecture design.

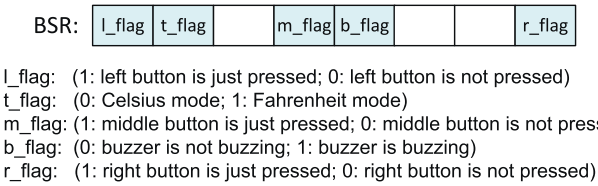

Figure 12.6
Button status register.

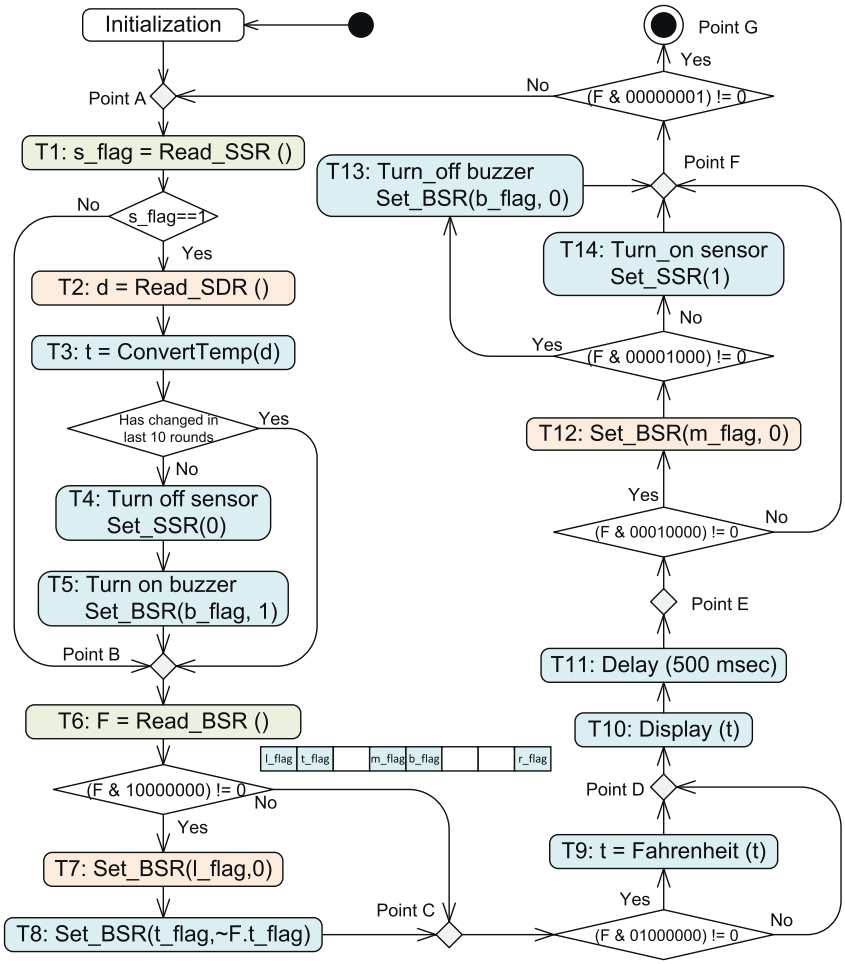

Figure 12.7
A design of a body thermometer.

Table 12.2 Task specification of a body thermometer

Task ID	Task Description	Execution Time	Release Condition
T_1	Read sensor flag	10 ms	Always
T_2	Read sensed temperature	10 ms	Sensor on
T_3	Convert to degree Celsius	50 ms	Sensor on
T_4	Turn off sensor; reset sensor flag	10 ms	No change for 10 rounds
T_5	Turn on buzzer; assert buzzer flag	10 ms	No change for 10 rounds
T_6	Read button status register	10 ms	Always
T_7	Reset l_flag to ack left-button press	10 ms	l_flag asserted
T_8	Toggle t_flag to switch mode	10 ms	l_flag asserted
T_9	Convert temperature to degree Fahrenheit	10 ms	t_flag asserted
T_{10}	Update LCD	100 ms	Always
T_{11}	Delay 500 ms	500 ms	Always
T_{12}	Reset m_flag to ack middle-button press	10 ms	m_flag asserted
T_{13}	Turn off buzzer; reset buzzer flag	10 ms	b_flag asserted
T_{14}	Turn on sensor; assert sensor flag	10 ms	b_flag not asserted

Depending on the run-time evaluation of the branching conditions, some tasks may not execute in every iteration of the loop. We can roughly classify the tasks as follows:

- Tasks T_1, T_6, T_{10}, and T_{11} execute in every iteration. Tasks T_2 and T_3 execute in every iteration when the sensor is on. Task T_9 executes in every iteration when the mode is set to degree Fahrenheit. They can be treated as periodic tasks.
- The execution of T_4 and T_5 is unexpected (no change in the last 10 rounds). They can be taken as aperiodic tasks.
- Tasks T_7, T_8, T_{12}, T_{13}, and T_{14} execute only when a button event happens. For example, T_7 and T_8 execute when a user presses the left button; T_{12} and T_{13} or T_{14} execute when a user presses the middle button. They can be treated as sporadic tasks.

In Table 12.2, we also list the hypothetical worst-case execution time and release conditions for each task. Note that the 'initialization' activity is not listed, because it is outside the round-robin loop and is irrelevant to our analysis.

12.2.1.3 Worst-case task response time

We know from Section 9.1.2 that there are typically three types of events: time events, signal events (internal or external), and change events. All three types of events occur in the design of the body thermometer as given in Figure 12.7. For instance, task T_{11} (delay 500 ms) needs to respond to a time event, and task T_4 raises a change event (turn off sensor) that will affect the system behavior at the decision point before task T_2.

For signal events, there are two subtypes: internal and external. Function calls are internal signals.[4] For example, "Set_SSR(0)" can be taken as a utility function called by task T_4. Although the sensor and the buzzer are hardware devices, they are not supposed to raise events by themselves. Instead, they respond only to software invocations *passively* in our design. Hence, "turn on sensor," "turn off sensor," "turn on buzzer," and "turn off buzzer" are also internal events.

External signal events are those triggered by the external environment. As we know from Chapter 4, external events are asynchronous in the sense that they can occur at any time and typically occur at unanticipated spots of the running program. For example, button presses from a user are external signal events. Devices such as buttons *actively* raise events to the system in response to external triggers (e.g., from the user). External events need to be captured and handled in a timely manner by the system.

Response-time analysis is relatively straightforward for time events, change events, and internal signal events, because they are typically raised and completely processed at fixed locations of the system execution. For example, the sensor can be turned off only in task T_4, and this change is handled immediately in task T_4. It is likely that a timer of 500 ms is launched at the beginning of task T_{11} and the time-out signal from the timer is captured and handled at the end of task T_{11}. Internal signal events are also handled almost immediately.

Thus, in response-time analysis we will focus only on external signal events. For the body thermometer, there is only one type of external event triggered by users: button presses. The task specification given in Table 12.2, together with the UML activity diagram, allows us to examine worst-case scenarios, a few of which are given in Table 12.3.

Take the first entry of Table 12.3 as an example. Suppose a user presses the right button while the system execution is at point A, at which time the context is as follows:

- *s_flag* is set (i.e., sensor is on);
- the temperature measure has not changed for the last 10 rounds; and
- the user also pressed the middle button and the left button while the system execution was in between point C and point A.

For this scenario, starting from point A, the system is turned off after the execution of tasks T_1, T_2, T_3, T_4 (sensor is turned off), T_5 (buzzer is turned on), T_6, T_7, T_8 (switched to degree Fahrenheit mode), T_9, T_{10}, T_{11}, T_{12}, and T_{13}. Thus, the path execution time is $10 + 10 + 50 + 10 + 10 + 10 + 10 + 10 + 10 + 100 + 500 + 10 + 10 = 750$ ms.

Since the timing requirement for a user button press is 1 s, it seems that all the timing requirements (from a user's perspective) can be satisfied by the round-robin-based design.

[4] In systems with interrupts, software interrupts and internal interrupts can also be taken as internal signal events. This category can be further expanded to include OS signals for systems built upon an OS.

Table 12.3 Worst-case event response time

Event	Occurs at	Worst-Case Context	Complete at	Response Time (ms)
Right button pressed (turn off)	Point A	s_flag is set, no change in the last 10 rounds BSR: 1 0 _ 1 0 _ _ 1	Point G	10 + 10 + 50 + 10 + 10 + 10 + 10 + 10 + 10 + 100 + 500 + 10 + 10 = 750
Middle button pressed (stop buzzing)	Point F	Buzzing, s_flag is not set, BSR: 1 0 _ 1 1 _ _ 0	Point F	10 + 10 + 10 + 10 + 10 + 100 + 500 + 10 + 10 = 670
Middle button pressed (restart)	Point F	Not buzzing, s_flag is set, no change in the last 10 rounds BSR: 1 0 _ 1 0 _ _ 0	Point F	10 + 10 + 50 + 10 + 10 + 10 + 10 + 10 + 10 + 100 + 500 + 10 + 10 = 750
Left button pressed (to degree Fahrenheit)	Point D	s_flag is set, no change in the last 10 rounds BSR: 1 0 _ 0 0 _ _ 0	Point D	100 + 500 + 10 + 10 + 10 + 10 + 50 + 10 + 10 + 10 + 10 + 10 + 10 = 750
Left button pressed (to degree Celsius)	Point D	s_flag is set, no change in the last 10 rounds BSR: 1 1 _ 1 0 _ _ 0	Point D	100 + 500 + 10 + 10 + 10 + 10 + 50 + 10 + 10 + 10 + 10 + 10 + 10 = 740

12.2.1.4 Hardware concurrency

A real-time embedded system is typically composed of many devices, each of which may need the system's attention (processing) whenever something important has just happened—say, a button has just been pressed by a user or a serial communication port has received new data. When something such as this happens, a device will notify the system by raising an external event, which is also called a *service request*.

There are four stages in the processing of an external event: occurrence, detection, acknowledgment, and service to completion.

(1) Occurrence. This stage is completely handled at the hardware level. When a device raises an external event (service request), a "footprint" is typically left in hardware—say, special flags and/or some data are set at a certain memory location (register). A service request (event) is called an *outstanding request* from the time instant it is raised until the time instant it is acknowledged by the system. Depending on its nature, a device may become temporarily irresponsive (i.e., operations disabled) while it has an outstanding service request. There is a good reason for this. While having an outstanding request, if the device were still responsive, it might be forced (say, nothing can prevent a user from pressing a button) to raise another service request, and this new footprint would certainly overwrite the footprint of the outstanding request.

(2) Detection. At this stage, the outstanding service request (event) is detected by the system. This can be done by hardware only (say, interrupts detected by a programmable interrupt controller), or by software (say, evaluating control flags, or reading hardware flags from memory or registers).
(3) Acknowledgment. At this stage,
    (a) the footprint of the service request is preprocessed. The footprint information is either recorded by variables or backed up to a safe location. For instance, a variable can be used to capture the hardware flags in a register (such a variable is also called a software flag). Afterward, the software flags and/or the backup copy will be used in any further processing of the current request. In this way, the handling of the current request is isolated from any potential changes to the hardware device.
    (b) the device is reset and re-enabled, if it was disabled for some reason. In particular, hardware flags in the status register are reset.
    (c) at this time point, we say that the request has been *acknowledged*. This means (i) the system will know (from the software flags) that the task for processing the current request is ready for execution, and (ii) the device can resume its normal operations and is ready to capture the next event. It is clear that any new outstanding request from the device will not affect the handling of the current request.
(4) Service to completion. At this stage, the system starts to schedule the corresponding service task to process the request until it has been serviced completely. Once the service task has started, it can clear, if applicable, the software flags set at the acknowledgment stage, so that they can be reused to indicate the readiness of a new service request.

The four stages discussed above are illustrated in Figure 12.8.

The understanding of the four event processing stages allows us to refactor the design in Figure 12.7 into a new design as shown in Figure 12.9.

This new design organizes tasks by their roles played in the event processing stages. Here, for a device i, we use T_i^{d}, T_i^{a}, and T_i^{s} to denote the tasks corresponding to the detection stage, acknowledgment stage, and service stage, respectively, of device i's event processing. Table 12.4 gives the mapping between the tasks of the original design and those of the new design.

Notice the following facts regarding the new design:

- For some tasks such as T_3^{a} and T_4^{d}, there is no counterpart in the original design. Each of these tasks has an execution time of 0. They appear in the new design as placeholders, which are simply for the convenience of our analysis.

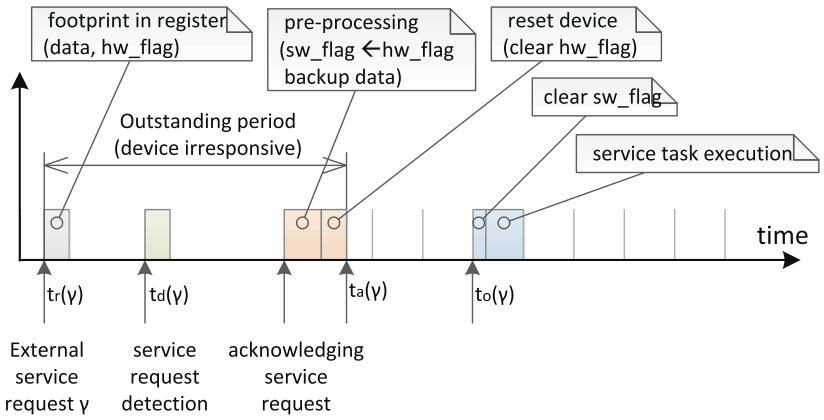

Figure 12.8
Four stages in processing external events.

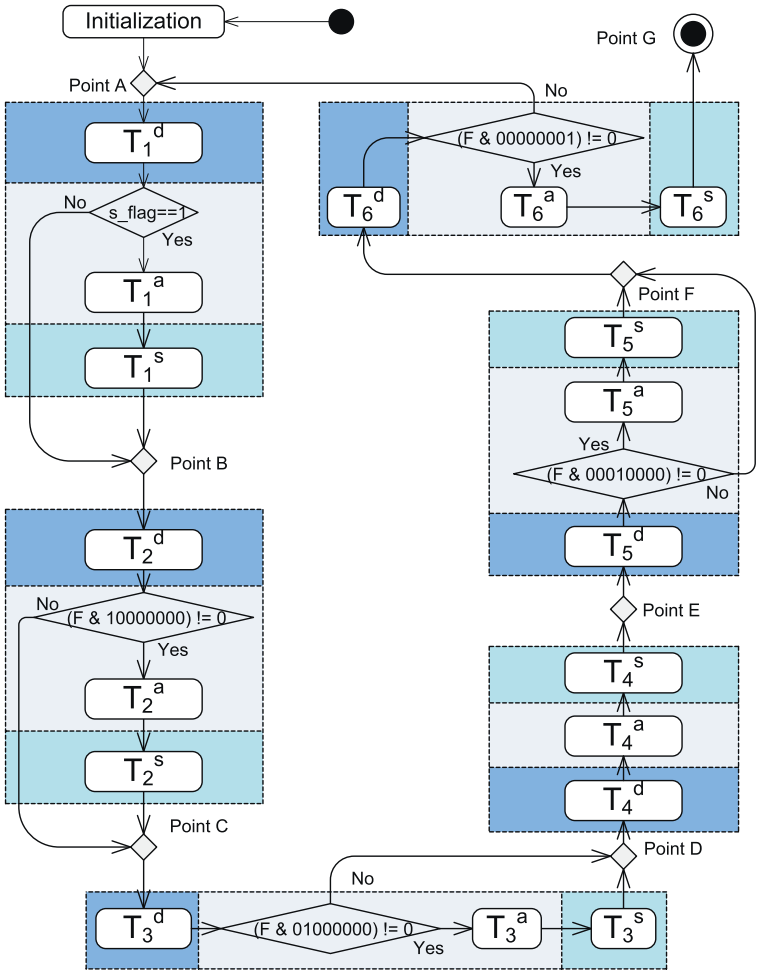

Figure 12.9
Refactored design of the body thermometer.

Table 12.4 Task mapping in structure refactoring

Device	Task ID	Mapped Back to Tasks	Worst-Case Execution Time
Sensor	T_1^{d}	T_1	$e_1^{\mathrm{d}} = 10$ ms
	T_1^{a}	T_2	$e_1^{\mathrm{a}} = 10$ ms
	T_1^{s}	T_3; T_4; T_5	$e_1^{\mathrm{s}} = 70$ ms
Left button	T_2^{d}	T_6	$e_2^{\mathrm{d}} = 10$ ms
	T_2^{a}	T_7	$e_2^{\mathrm{a}} = 10$ ms
	T_2^{s}	T_8	$e_2^{\mathrm{s}} = 10$ ms
Mode filter	T_3^{d}	T_6 (added)	$e_3^{\mathrm{d}} = 10$ ms
	T_3^{a}	Null	$e_3^{\mathrm{a}} = 0$
	T_3^{s}	T_9	$e_3^{\mathrm{s}} = 10$ ms
LCD	T_4^{d}	Null	$e_4^{\mathrm{d}} = 0$
	T_4^{a}	Null	$e_4^{\mathrm{a}} = 0$
	T_4^{s}	T_{10}; T_{11}	$e_4^{\mathrm{s}} = 600$ ms
Middle button	T_5^{d}	T_6 (added)	$e_5^{\mathrm{d}} = 10$ ms
	T_5^{a}	T_{12}	$e_5^{\mathrm{a}} = 10$ ms
	T_5^{s}	(T_{13} | T_{14})	$e_5^{\mathrm{s}} = 10$ ms
Right button	T_6^{d}	T_6 (added)	$e_6^{\mathrm{d}} = 10$ ms
	T_6^{a}	Null	$e_6^{\mathrm{a}} = 0$
	T_6^{s}	Null	$e_6^{\mathrm{s}} = 0$

- Sensor has a task T_1^{a} devoted to the acknowledgment stage. T_1^{a}, which is T_2 in the original design, reads the current temperature measure from the register `SDR` into a variable d.[5] The left button has a task T_2^{a} (i.e., T_7) devoted to the acknowledgment stage. This task simply resets the *l_flag* so that it can be used to capture a new left-button press event. The middle button also has a task T_5^{a} (i.e., T_{12}) devoted to the acknowledgment stage. This task simply resets the *m_flag* so that it can be used to capture a new middle-button press event. There is no acknowledgment task for the right button, because the system is simply turned

[5] To prevent dirty read, it may disable `SDR` for writing beforehand and enable writing again afterward.

off when it is pressed. The mode filter and the LCD do not have an acknowledgment task either; they do not need to be acknowledged.

- Some of the original tasks are merged. For example, T_1^s corresponds to the original block composed of T_3, T_4, and T_5. T_1^s's execution time is the block's worst-case execution time, which is at most $50 + 10 + 10 = 70$ ms. T_5^s's execution time is derived similarly.
- There is no counterpart for tasks T_3^d, T_5^d, and T_6^d in the original design. They are added for a reason that will become clear shortly.

We need to use some notation to refer to the important time instants regarding a service request. For a service request γ, let

- $t_r(\gamma)$ denote the time instant at which γ is raised,
- $t_d(\gamma)$ denote the time instant at which γ is detected,
- $t_a(\gamma)$ denote the time instant at which γ is acknowledged, and
- $t_o(\gamma)$ denote the time instant at which the system starts to execute the service task for γ.

Given that a request γ is raised by a device i, while the release time of γ is $t_r(\gamma)$, the release time of the corresponding service task T_i^s is actually $t_a(\gamma)$, the time instant at which T_i^s is ready for execution.

We call the time interval $[t_r(\gamma), t_a(\gamma)]$ the *outstanding period* of γ. When the service requests of a device have a short outstanding period on average, we say that the device has a *high hardware concurrency*, because it can respond to new external triggers very quickly. Let us examine the three buttons to see their worst-case outstanding periods and best-case outstanding periods.

Now, let us assume that T_3^d was not added. Suppose when the system's execution is at point B, a user presses the left button to switch the mode from degree Celsius to degree Fahrenheit. Since the current mode is degree Celsius, the *t_flag* value read to F by T_2^d is 0. In task T_2^s, *t_flag* is changed to 1. However, this flag change would not be noticed by the mode-filter block, because the *t_flag* in F is still 0. The mode change would happen the next time when `BSR` is read again in T_2^d. By addition of T_3^d, the mode change will take effect immediately in the current execution round.

We have added T_5^d and T_6^d for a similar reason. Suppose a user presses the middle button when the system's execution is at point E. From Table 12.5 we see that this is the best-case scenario, where the middle button has its shortest outstanding period. However, if T_5^d were null, then this button event would not be acknowledged immediately by T_5^a because it is not `BSR` but the variable F that is used at the decision point (F has an obsolete snapshot of `BSR`); instead it would be captured in the next round at T_2^d and then acknowledged by T_5^a. Consequently, this best-case scenario would have an outstanding period of 760 ms! As an exercise, you may want to check the best-case outstanding period of the right button when T_6^d is null.

Table 12.5 Responsiveness of external button events

Device Event		Raised at	Acknowledged by Task	Length of Outstanding Period	Event Response Time
Left button pressed	Best case	Point B	T_2^a	$e_2^d + e_2^a = 20$	$e_2^d + e_2^a + e_2^s = 30$
	Worst case	End of T_2^a	T_2^a	$\sum_{i=1}^{6}(e_i^d + e_i^a + e_i^s) = 780$	790
Middle button pressed	Best case	Point E	T_5^a	$e_5^d + e_5^a = 20$	$e_5^d + e_5^a + e_5^s = 30$
	Worst case	End of T_5^a	T_5^a	$\sum_{i=1}^{6}(e_i^d + e_i^a + e_i^s) = 780$	790
Right button pressed	Best case	Point F	T_6^d	$e_6^d + e_6^a = 10$	$e_6^d + e_6^a + e_6^s = 10$
	Worst case	Point A	T_6^d	$\sum_{i=1}^{6}(e_i^d + e_i^a + e_i^s) = 780$	780

In sum, tasks T_3^d, T_5^d, and T_6^d are added in the new design simply to shorten the outstanding periods of the corresponding devices. However, this will not do anything good to improve the worst-case outstanding period, which is, for all three buttons, the length of a whole execution round! This means, in worst-case scenarios, users may experience irresponsiveness when a button is pressed, which is an indicator of low hardware concurrency. We will see how to improve hardware concurrency by interrupts in Section 12.3.

12.2.2 General Round-Robin Architecture

You may have already noticed that in Figure 12.9 there is a "detect-acknowledge-service" (DAS) task pattern for each device. In general, we assume that each hardware device of a system can raise its service requests by setting a predefined hardware flag at a specific memory location (register), and the system applies the DAS task pattern to each device to repeatedly detect, acknowledge, and service all the external requests.

Figure 12.10 gives the general round-robin architecture. We have the following observations:

(1) In the circular structure as shown in Figure 12.10, the round-robin principle is applied at the device level. In other words, the architecture is a series of DAS task patterns, and each task runs to completion as an integral unit. In Chapter 13 we will learn that the round-robin principle can be applied at a much finer level of granularity to implement a so-called round-robin task scheduler, where time slices (or time slots or quanta) are assigned to each task in equal portions and in circular order. Since the quantum is generally smaller than the task execution time, each task may take many rounds for it to be completed.

(2) In each round of execution, each device i is handled only once, because there is exactly one DAS task pattern dedicated to it. That is, for a device i, the tasks T_i^d, T_i^a, and T_i^s each

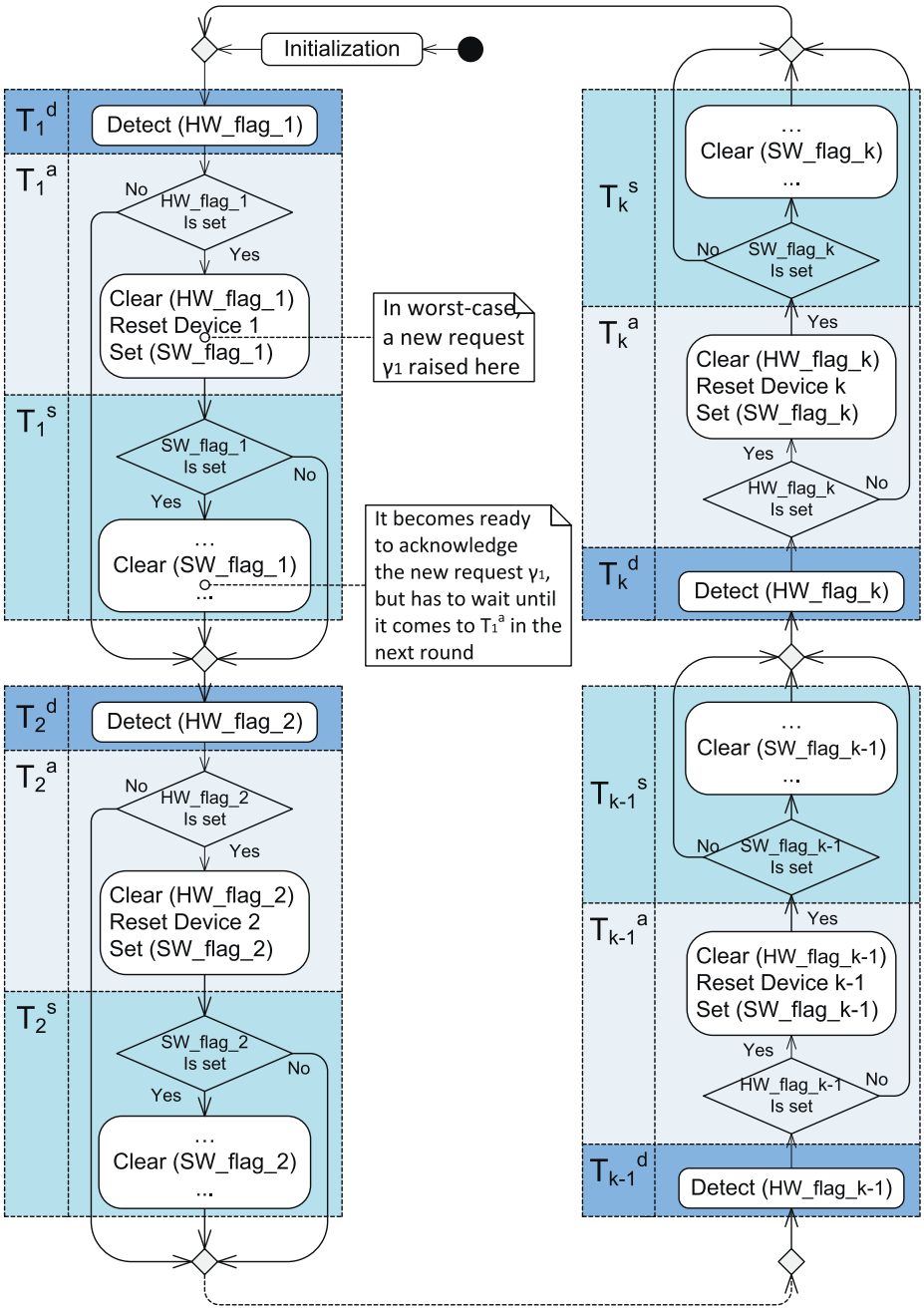

Figure 12.10
Round-robin architecture with a tightly coupled DAS pattern.

appear only once. It is practically possible to have some devices (say, with more urgent deadlines) handled multiple times in each round. As an example, consider a round-robin design with the DAS task pattern appearing twice for a device j. It is easy to show that the worst-case outstanding period and worst-case event response time of device j can be shortened approximately by half. However, such a biased design has to be used very carefully because it would adversely increase the worst-case outstanding periods and worst-case event response times of all the other devices.

(3) A hardware flag `HW_flag_i` and a software flag `SW_flag_i` are reserved for each device i: `HW_flag_i` is asserted by hardware and is cleared by T_i^{a}, while `SW_flag_i` is asserted by T_i^{a} and is cleared by T_i^{s}. The two flags serve as relay batons so that the external events from device i can be processed in an appropriate order. In particular, device i is irresponsive to new triggers until `HW_flag_i` is cleared by T_i^{a}; the system is not ready to handle a new request of device i until `SW_flag_i` is cleared by T_i^{s}.

(4) Owing to the round-robin nature, immediately after `SW_flag_i` has been cleared,[6] even though the system becomes ready to acknowledge a new request from device i, this will not happen until the control comes to T_i^{a} in the next execution round.

(5) For each device i, the corresponding tasks T_i^{d}, T_i^{a}, and T_i^{s} are placed next to each other. However, because of flags `HW_flag_i` and `SW_flag_i`, T_i^{d}, T_i^{a}, and T_i^{s} can be separately arranged as illustrated in Figure 12.11. This will not affect device i's worst-case outstanding period.

12.2.3 Worst-Case Event Response Time

To simplify our analysis, we stick with the architecture given in Figure 12.10, and make the following assumptions explicit. For each device i ($1 \leq i \leq k$), we have the following:

(1) In the circular structure, there is exactly one DAS pattern for i.
(2) The software flag `SW_flag_i` is asserted only in T_i^{a}, and is cleared only in T_i^{s}.
(3) The execution time of a decision point (condition evaluation and branching) is negligible.
(4) It takes no time to execute a null task. That is, $e_i^{\mathrm{d}} = 0$ if T_i^{d} is null, and $e_i^{\mathrm{a}} = 0$ if T_i^{a} is null. Note that we must have $e_i^{\mathrm{s}} > 0$, otherwise device i should not have been considered in the design.

We consider the worst-case scenario as illustrated in Figure 12.12:

- A new request from a device j, denoted by γ_j', is raised immediately after device j is reset in T_j^{a}. This new request can be correctly recorded by the hardware flag `HW_flag_j`, and it will not override the current request, denoted by γ_j, because it is recorded by the software flag `SW_flag_j`.

[6] The service to the current request of device i may be still in progress.

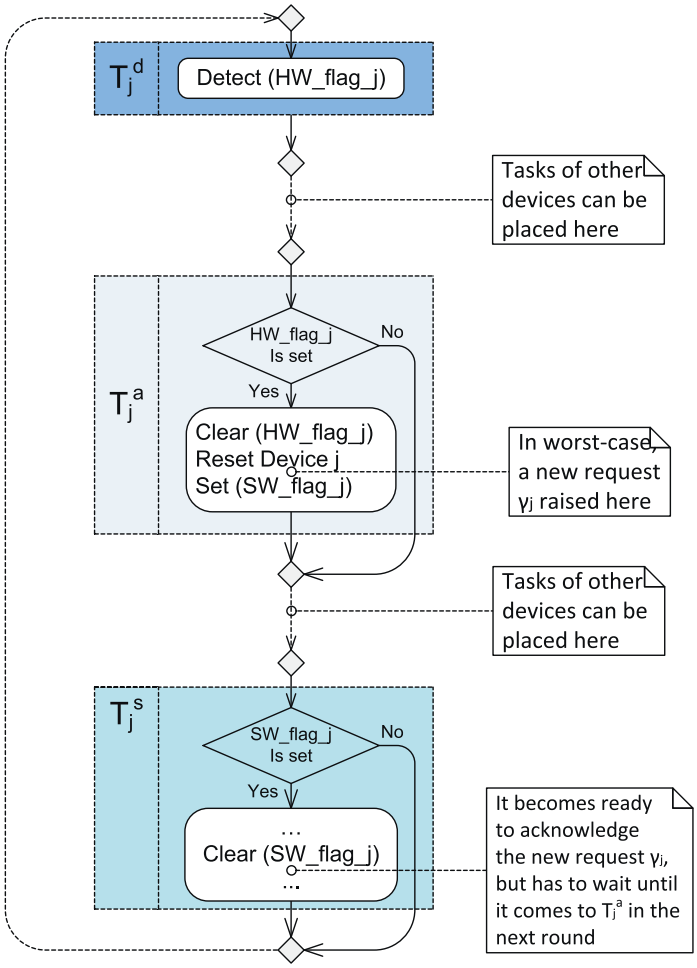

Figure 12.11
Round-robin architecture with loosely coupled DAS patterns.

- T_j^s starts to execute to service the current request γ_j, which takes e_j^s.
- In the worst case, before it comes to T_j^d again, along the execution path the system has to service one request from each of the other devices! For each device i ($1 \leq i \leq k, i \neq j$), the execution time is $(e_i^d + e_i^a + e_i^s)$.
- As it comes to T_j^d again in the next round, T_j^d and T_j^a need to be executed to detect and acknowledge γ_1', and T_j^s needs to be executed to actually service γ_1'. This takes $(e_j^d + e_j^a + e_j^s)$.

Thus, according to the above worst-case analysis, the worst-case outstanding period for a device j is given by

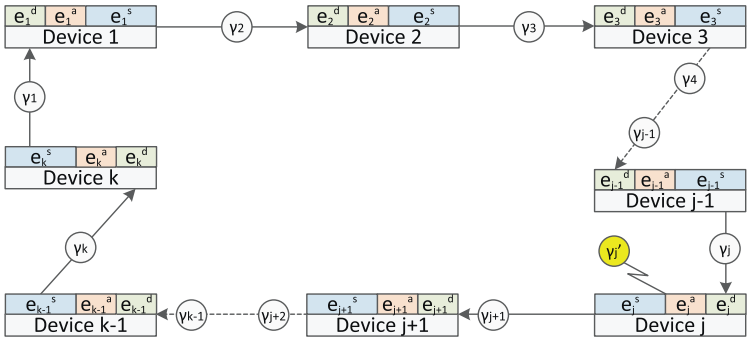

Figure 12.12
Round-robin worst-case response time: a new request γ_j' arrives just after the current request γ_j is acknowledged.

$$\sum_{i=1}^{k}(e_i^{\mathrm{d}} + e_i^{\mathrm{a}} + e_i^{\mathrm{s}}),$$

and the worst-case event response time for a device j is given by

$$e_j^{\mathrm{s}} + \sum_{i=1}^{k}(e_i^{\mathrm{d}} + e_i^{\mathrm{a}} + e_i^{\mathrm{s}}).$$

Obviously, systems based on the round-robin architecture can suffer from low hardware concurrency. This is because the system cannot handle a request from a device immediately; it has to wait until the next time it comes to the DAS task pattern that is dedicated to that device.

12.3 Round Robin with Interrupts

Hardware concurrency is one of the most desirable features in high-performing real-time embedded systems. In order to improve the hardware concurrency of a device, we have to shorten its outstanding period.

One solution is to bring the interrupt concept into play, by which a system could acknowledge external service requests almost immediately. The simplest architecture featuring interrupts is called *round robin with interrupts*. Let us first look at an example.

12.3.1 Case Study: The Simon Game

The Simon game is a memory retention game invented by Ralph H. Baer and Howard J. Morrison to measure and challenge a player's memory retention capacity. The game can

generate a growing sequence of events (colors and sounds) that the player has to repeat. There are several versions of the Simon game on the market: some allow a participant to play against another player, and some may feature adjustable skill levels.

We consider a "simplified" version, where the game system has four colored buttons, each producing a particular tone when it is pressed by a player or activated by the system. When the game starts, the system lights up one or more buttons in a random order, after which the player must accurately reproduce that order by pressing the buttons. For each successive round, the system repeats the latest sequence and adds another button to the sequence. This process is repeated until the participant makes an error, or until the sequence reaches a maximum length.

12.3.1.1 Hardware design

Our "simplified" game system has the following major components:

(1) A PIC microcontroller (say, PIC16F84A or PIC16F628).[7]
(2) A power on/off button.
(3) Four colored pushbuttons (red, yellow, green, and orange), each is lit when pushed, to produce visual cues.
(4) A mini piezo, which can produce a particular tone when it is turned on and off at a definite rate (an audial cue associated with each colored button).
(5) A certain number of byte-width registers.

Suppose the colored buttons are coded such that 0, 1, 2, and 3 represent red, yellow, green, and orange, respectively. Then, it requires two bits to record each step, and 12 registers (byte-width registers) can accommodate a series of up to 48 button presses (steps). One register can be used to keep track of the length (number of steps) of the current series.

12.3.1.2 Software architecture design

In Figure 12.13 we give a simple round-robin design with two interrupts, one for handling button press events and one for handling time-out events. Notice that the two interrupt service routines (ISRs) have the same structure: acknowledging the hardware device (e.g., recording the pressed button) then setting a software flag. The ISRs actually play the same role as played by T_i^a in the round-robin architecture. The big difference is that the execution of T_i^a is performed once in each round and in a fixed order, while the ISRs can be triggered many times and always preempt the execution of the round-robin loop.

The architecture design is quite simple. The system plays welcome music when it is powered on. Then, it clears the current series, randomly selects a colored button, and adds it to the

[7] The original prototype system, built by Baer, included the Texas Instruments TMS 1000 microprocessor chip, which was of low cost and was used by many games in the 1970s.

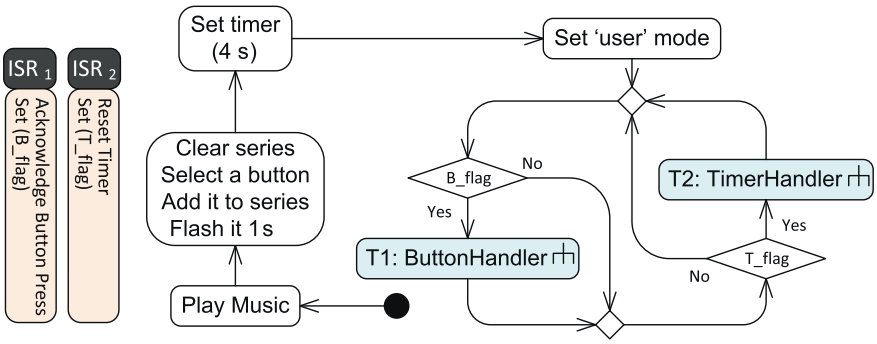

Figure 12.13
An example round-robin architecture with interrupts.

series. The selected button is flashed for 1 s to produce a visual cue (and the corresponding tone is played to produce an audial cue). Then, a timer with a time-out value of 4 s is launched and the "user mode" is set to be the current operation mode. After these initialization steps, the system enters into a round-robin execution loop.

Inside the round-robin execution loop, the system repeatedly checks two flags `B_flag` and `T_flag`, which are asserted by ISR_1 and ISR_2, respectively. In other words, when `B_flag` is true, it indicates that a button has just been pressed; when `T_flag` is true, it indicates that a time-out event has occurred. Depending on the truth values of `B_flag` and `T_flag`, the system executes the two tasks T_1 and T_2 repeatedly in a round-robin manner.

Task T_1 handles button-press events. As shown in Figure 12.14, it clears flag `B_flag` first (so that the next button press can be correctly recorded by ISR_1). T_1 operates only in user mode, so it terminates immediately if the system is currently not in user mode. In user mode, T_1 first disables the timer (it is canceled because the user has responded within the 4 s "thinking" deadline), then checks whether the pressed button is correct. If the player has pressed a wrong button (failed to recall the correct series), the game is restarted after the system has played "failing" music for 2 s. If the player has pressed a correct button, it is flashed for 1 s, then the system checks whether the series is finished. If it has not finished, a 4 s "thinking" period is started for the user to recall the next correct button in the series; if the series has finished, the system enters into a "replay" mode.

Task T_2 handles time-out events. As shown in Figure 12.15, it clears flag `T_flag` first (so that the next time-out can be correctly recorded by ISR_2). If the system is in user mode (the 4 s "thinking" deadline is broken), the game is restarted. If the system is in replay mode (it is time to replay the next button), the next button in the series is flashed for 1 s. Next, the system checks whether the series is finished. If it has not finished, a 1 s timer is launched (when it expires, `T_flag` is set and T_2 is executed again); if it has finished, a new button is randomly selected and is added to the series, and the system enters into user mode.

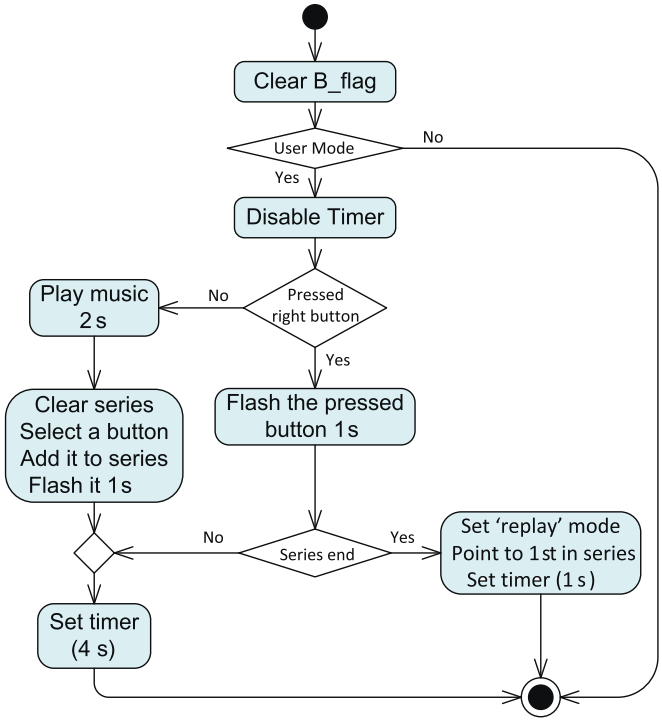

Figure 12.14
Task T_1: handling button press.

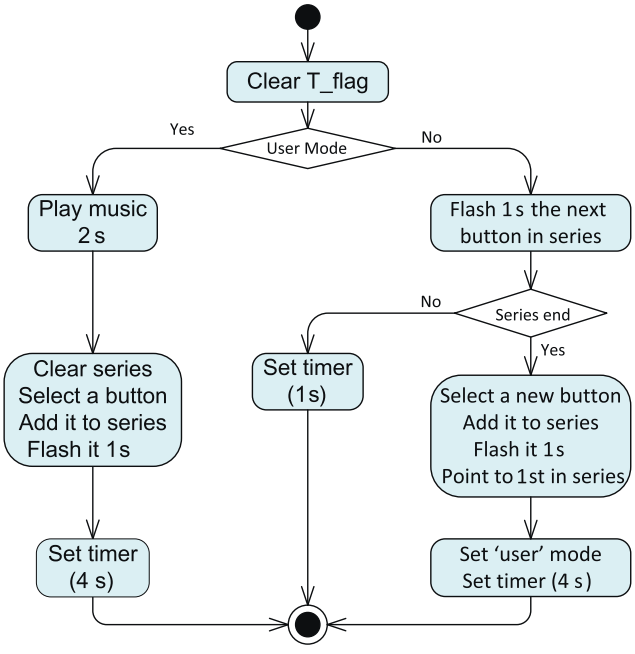

Figure 12.15
Task T_2: handling time-out events.

Obviously, T_1 is an aperiodic task, which is released whenever a user presses a button. It is associated with a soft deadline: a user has to press a button within a 4 s "thinking" time, otherwise the game is restarted. Task T_2 can be viewed as consisting of two subtasks. The user-mode branch is executed whenever a user fails to recall a correct button in 4 s. The replay-mode branch is repeatedly executed every 2 s while the system is replaying a long series.

The worst-case execution time of both tasks is a bit longer than 3 s, which happens when a user has pressed a wrong button.

The worst-case event response time is slightly longer than 6 s; this happens when both a user-mode time-out and a wrong-button press event happen almost at the same time.

12.3.2 General Architecture

In general, the round-robin architecture with interrupts is given in Figure 12.16.

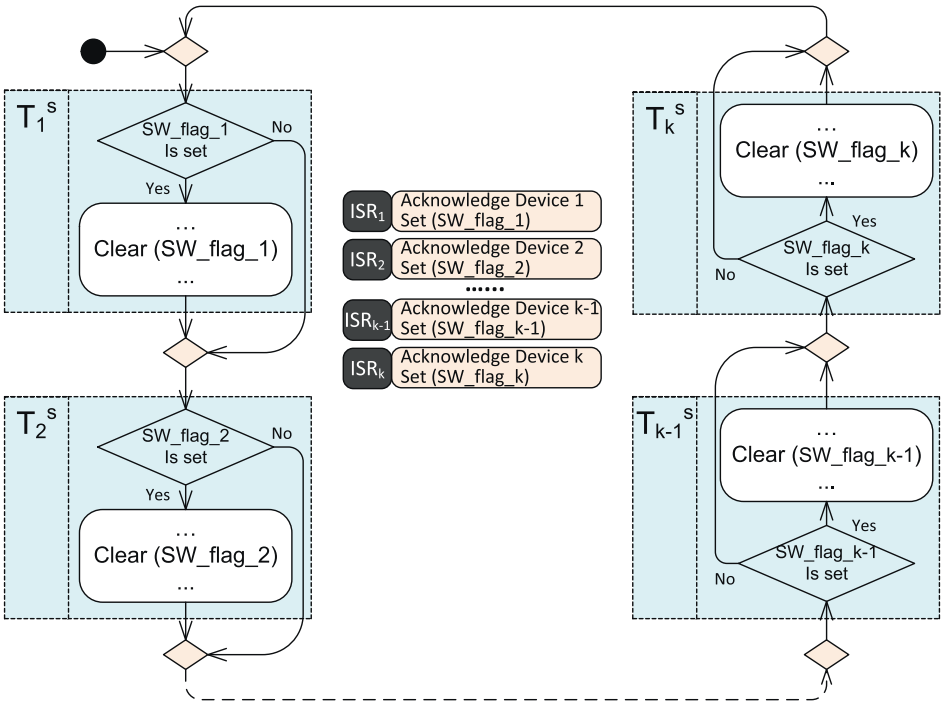

Figure 12.16
Round-robin architecture with interrupts.

This architecture has the following features:

(1) Similarly to the round-robin architecture, the round-robin principle is applied at the device level.

(2) Different from the round-robin architecture, T_i^{d} $(1 \leq i \leq k)$ is no longer needed, because the detection of hardware flags is performed by hardware (i.e., PIC microcontroller). Whenever a request from a device i is raised, it is acknowledged by the corresponding interrupt service routine ISR_i.

(3) ISR_i plays exactly the same role as T_i^{a}. These acknowledgment tasks are no longer placed on the execution path of the round-robin loop. Instead, they are treated separately. Whenever an interrupt request occurs, the execution of the round-robin loop will be preempted by the corresponding ISR.

(4) The worst case happens if device i has the lowest interrupt priority, and each of the other devices has raised a request that preempts the execution of ISR_i. In such a worst case, the outstanding period of the service request from device i would be

$$\sum_{i=1}^{k} e_i^{\mathrm{a}}.$$

Owing to the nature of the interrupt mechanism, an ISR can be triggered and executed multiple times during one round of the system execution. As illustrated in Figure 12.17(a), an execution of ISR_j $(1 \leq j \leq k)$ can appear just after software flag `SW_flag_j` has been cleared by T_j^{s}. This execution of ISR_j will acknowledge the service request by setting `SW_flag_j` again, and this request will not be serviced until it comes to T_j^{s} in the next round. However, along the execution path, ISR_j can be triggered to execute many times before the control once again comes to T_j^{s}. Since each trigger of ISR_j indicates a new service request from device j, the question is, how does the system handle multiple requests when the control comes to T_j^{s}?

There are two solutions. One is to adopt the FIFO policy in the sense that only the first service request is to be serviced by T_j^{s} while all the subsequent requests are ignored by the system. Another solution is to record all the requests by a counter or an array-like structure and for T_j^{s} to process them in a batch mode. A side effect of this solution is that the execution time of each round becomes nondeterministic, which may cause some tasks to miss their deadlines.

To simplify our analysis, we assume that the FIFO policy is used. We can actually apply a design pattern to the ISRs in order to enforce that only the first request is honored. As shown on the left in Figure 12.17(b), ISR_j (i.e., T_j^{a}) simply returns when `SW_flag_j` is asserted. This implies that another request has already been acknowledged prior to this new request.

Then, when would be the earliest opportunity that a request from a device j can be successfully acknowledged? The answer is that the request has to trigger ISR_j immediately after `SW_flag_j` has just been cleared! This is illustrated in Figure 12.17(b) and 12.17(c); they show the same situation except that in Figure 12.17(b) the clearance of the software flag

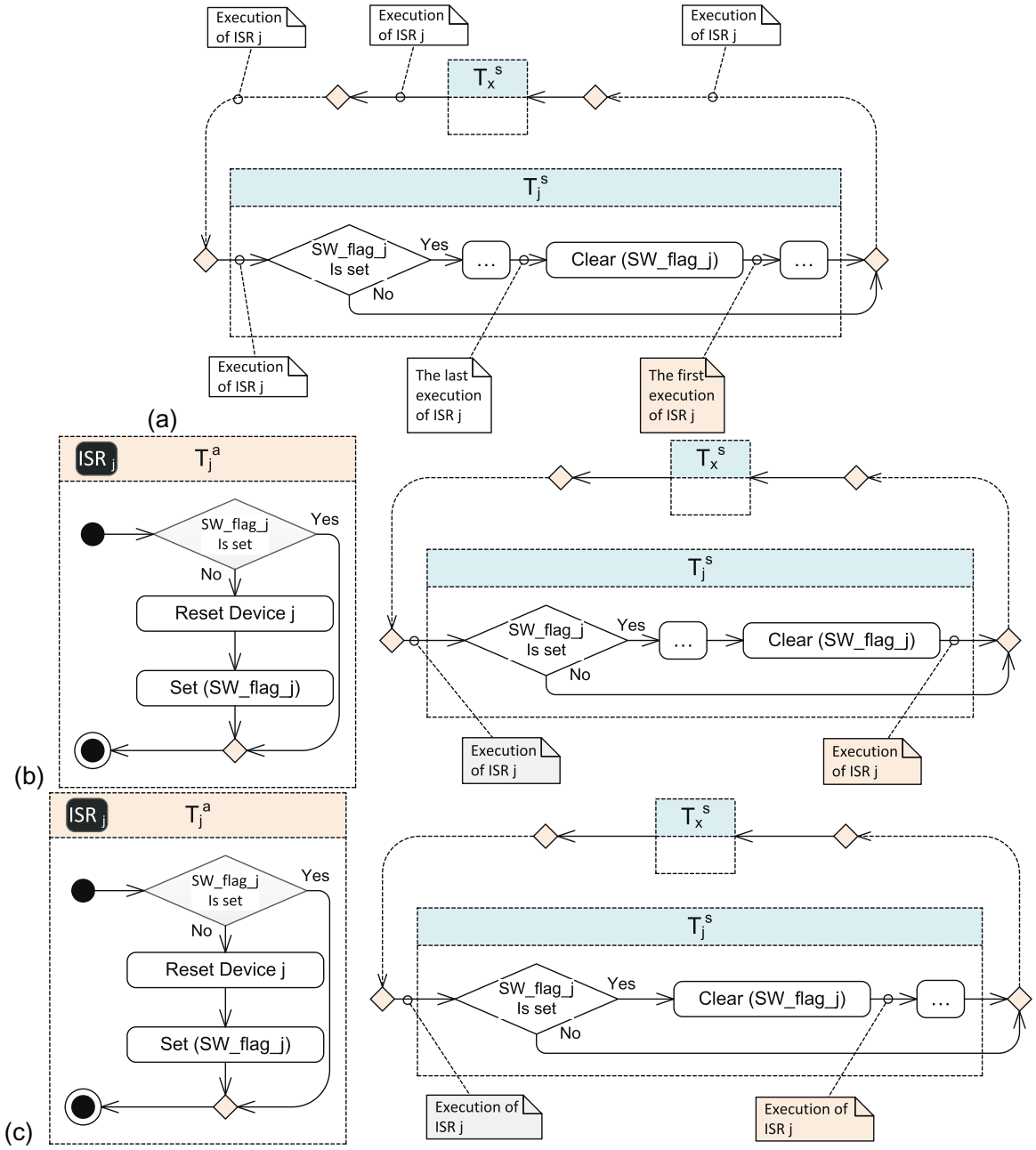

Figure 12.17
Many requests being acknowledged by ISR_j, the system may choose to honor only the first one or all requests in T_j^s (a); flag clearance placed at the end of T_j^s (b); and flag clearance placed at the beginning of T_j^s (c).

is at the end of task T_j^s, while in Figure 12.17(c) the clearance of the software flag is the first thing to do in T_j^s.

12.3.3 Worst-Case Event Response Time

Let us use Figure 12.18 to analyze the worst-time event response time. Our analysis is based on the following assumptions. For each device i ($1 \leq i \leq k$),

(1) task T_i^s appears only once in the circular structure.
(2) software flag `SW_flag_i` is asserted only in ISR_i. In addition, the design pattern as shown in the center of Figure 12.18 is used for ISR_i: `SW_flag_i` is asserted (to acknowledge the triggering request from device i) only when `SW_flag_i` is not currently asserted.
(3) software flag `SW_flag_i` is cleared only in the service task T_i^s. In addition, the clearance of `SW_flag_i` is placed at the beginning of T_i^s (i.e., the case as illustrated in Figure 12.17(c)).
(4) the execution time of a decision point (condition evaluation and branching) is negligible. In particular, e_i^a (the execution time of ISR_i) is 0 when `SW_flag_i` is asserted.

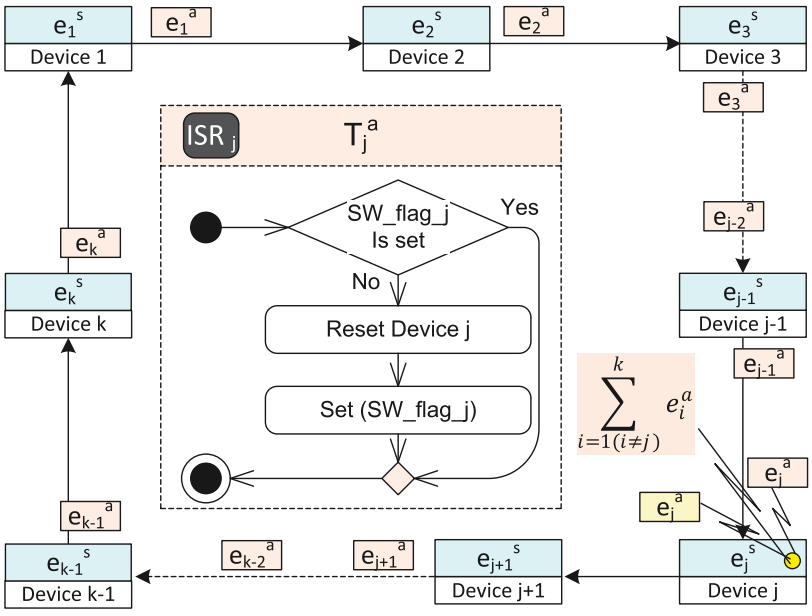

Figure 12.18
Worst-case response time: ISR_j executes as soon as flag `SW_flag_j` has been cleared at the beginning of T_j^s.

One worst-case scenario is illustrated in Figure 12.18:

- T_j^s starts to execute to service the current service request from device j. The first thing to do in T_j^s is to clear `SW_flag_j`.
- As soon as `SW_flag_j` is cleared, a new service request from device j, denoted by γ_j, triggers the execution of ISR_j. In the worst case, device j has the lowest interrupt priority, and each of the other devices has raised a request that preempts the execution of ISR_j. In such a worst case, the total execution time of the ISRs, including the one acknowledging γ_j, is given by $\sum_{i=1}^{k} e_i^a$. After this, the new request γ_j is successfully acknowledged.
- T_j^s completes its service to the "old" request in e_j^s. Now, γ_j is the only request from device j to be serviced.
- Along the execution path, in the worst case, there is at most one request from each device i ($1 \leq i \leq k, i \neq j$) that can be successfully acknowledged (after servicing the "old" request). Thus, for each i ($1 \leq i \leq k, i \neq j$), the execution time is $e_i^s + e_i^a$;
- As the control comes to T_j^s again, it starts to service γ_j.
- As soon as `SW_flag_j` is cleared, another new service request from device j triggers the execution of ISR_j. The amount of execution time is e_j^a, and this new request is successfully acknowledged.
- T_j^s completes its service to γ_j in e_j^s.

Thus, according to the above worst-case analysis, the worst-case event response time for a device j is given by

$$e_j^s + \sum_{i=1}^{k} (2 \times e_i^a + e_i^s).$$

As we have explained before, owing to the introduction of interrupts, the worst-case outstanding period for a device j ($1 \leq j \leq k$) is $\sum_{j=1}^{k} e_j^a$—the execution time of all ISRs. As compared with the round-robin architecture, this is significantly less and leads to much better hardware concurrency.

Table 12.6 summarizes the four processing stages for external events.

Table 12.6 Processing stages for external events

Event from Device *i*	Raised	Detection			Acknowledgment			Service		
	At	By	At	Amount	By	At	Amount	By	At	Amount
Round robin	Any time	Software	T_i^d	$e_i^d > 0$	Software	T_i^a	$e_i^a \geq 0$	Software	T_i^s	$e_i^s > 0$
Round robin with interrupts	Any time	Hardware (in no time)		$e_i^d = 0$	Software	ISR_i	$e_i^a \geq 0$	Software	T_i^s	$e_i^s > 0$

12.4 Queue-Based Architecture

By introducing a queue data structure, we can further refine the round-robin architecture with interrupts as given in Figure 12.16.

A refactored architecture is given in Figure 12.19, where a queue of length k is used by the system to determine when to execute which task.

The queue is static in the sense that exactly one fixed slot in the queue is assigned to each device. Each slot in the queue stores a function pointer—a pointer to the start memory location of the corresponding task code.

The queue is populated by ISRs. Whenever a request from a device i occurs, ISR_i is executed to acknowledge the device and put a function pointer T_i into the ith slot of the queue.

The queue is examined by the main program, shown on the left in Figure 12.19, to determine whether it needs to execute a service task. The main program contains a loop; it uses an index to keep track of the current slot. For each iteration, it retrieves and clears the current slot j. If it contains a function pointer, the system starts to run the corresponding task to its completion. If the current slot contains a null pointer (there is no service request from device j), the index is increased by 1 and the next iteration starts. The index is wrapped to the first position when it reaches the end of the queue.

This queue-based architecture is equivalent to the architecture given in Figure 12.16. The software flags are replaced by the manipulation of function pointers. The round-robin

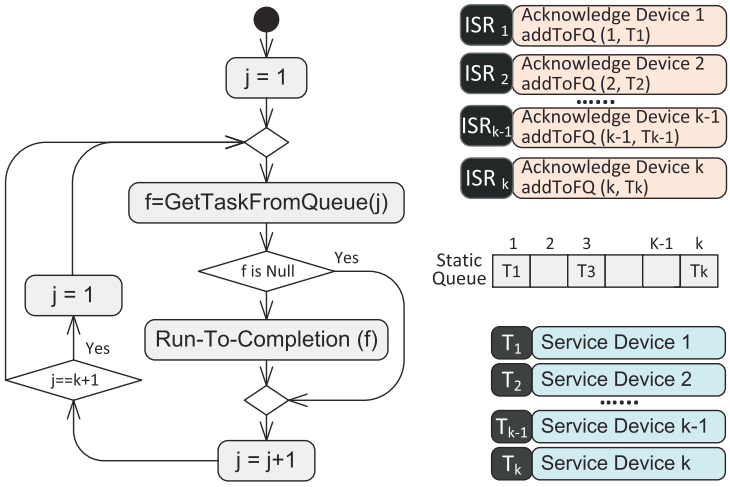

Figure 12.19
Architecture with a static task queue.

structure in Figure 12.16 is implemented here as a static queue which is repeatedly checked by the main program in a fixed order. Consequently, similarly to the round-robin architecture with interrupts, a system built upon this queue-based architecture needs to cycle around the queue to service a new request, and the worst-case event response time for a device j is still $e_j^s + \sum_{i=1}^{k}(2 \times e_i^a + e_i^s)$ (on the basis of assumptions similar to those given in Section 12.3.3).

It is worth noting that the static queue used in this architecture is an exclusive resource shared by the main program and the ISRs. Special care needs to be taken (say, disabling interrupts while the queue is accessed) in the implementation of GetTaskFromQueue() and ISRs.

12.4.1 Nonpreemptive FIFO Queue

The queue used in the architecture given in Figure 12.19 is a static queue, which exhibits at least two limitations. First, the order of service processing is fixed at design time; this is not flexible because at run time it may be necessary to process tasks in an order that is dynamically determined—say, according to how important or urgent they are. Second, a slot j of the static queue can hold at most one service request from device j at the same time. As long as slot j is occupied, any subsequent requests from device j are ignored.

To address these limitations, a dynamic queue can be used instead, so that different requests from the same device can take different positions in the queue. In general, there are two dynamic queue-based architectures: one is based on a FIFO queue and the other is based on a priority queue.

Figure 12.20 gives an architecture based on a FIFO queue. Compared with Figure 12.19, this architecture differs in two main aspects:

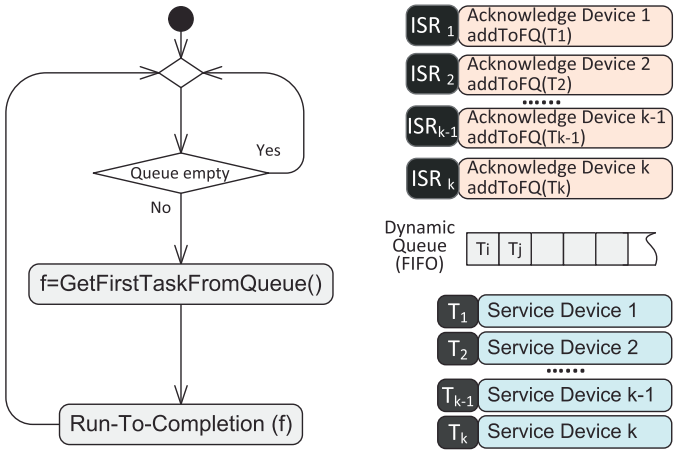

Figure 12.20
An architecture with a FIFO queue.

(1) First, each device no longer has a fixed slot in the queue. Instead, among all the devices, whichever raises the first request, its ISR will add the service task (function pointer) to the first position in the queue, whichever raises the second request, its ISR will add the service task to the second position, and so on. Consequently, all the requests are buffered by the queue in a FIFO manner, and multiple requests from the same device can coexist in the queue.

(2) The main program (shown on the left in Figure 12.20) in each iteration simply removes the first task from the queue and executes the task to its completion. After the execution of GetFirstTaskFromQueue(), the other tasks in the queue are shifted forward by one position (so that the second task becomes the first task and will be executed in the next iteration). In this way, the system executes the tasks from the queue one after the other, serving the external requests in a FIFO order.

12.4.2 Nonpreemptive Priority Queue

The earliest request (task) may not be the most important request (task). A higher priority is typically assigned to a more important task.

Figure 12.21 shows an architecture with a priority queue, which differs from the architecture in Figure 12.20 in two aspects:

(1) First, each ISR needs to add the corresponding task (function pointer) at an appropriate position in the queue such that all the tasks in the queue are ordered decreasingly by their priorities (FIFO for tasks with the same priority).

(2) Since the first position of the queue always contains the task with the highest priority, the main program (shown on the left in Figure 12.21) in each iteration simply removes the

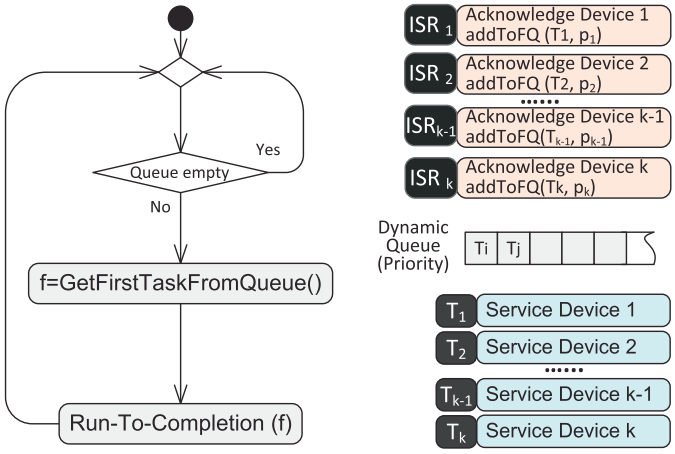

Figure 12.21
Architecture with a nonpreemptive priority queue.

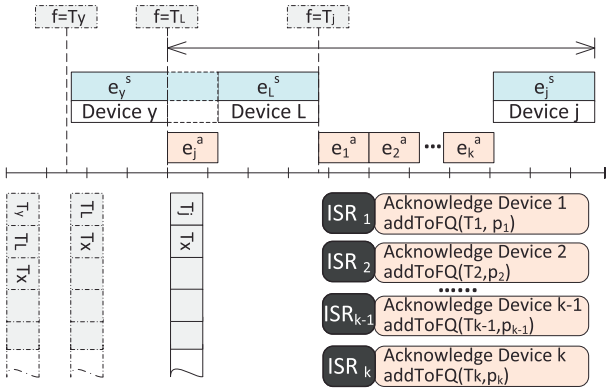

Figure 12.22
Response time for a nonpreemptive priority queue.

first task from the queue and executes the task to its completion. After the execution of GetFirstTaskFromQueue(), the other tasks in the queue are shifted forward by one position. However, the task at the head of the queue might not be executed in the next iteration, because a higher-priority task could be added to the queue during the execution of the current task.

What is the worst-case event response time for the highest-priority task? Figure 12.22 illustrates a worst-case scenario, where device j (i.e., task T_j) has the highest priority:

(1) At the end of T_y, just after T_L has been retrieved from the queue, a request from device j is raised and the main program is preempted by the execution of ISR_j, which takes e_j^a.
(2) After ISR_j finishes, the current task T_L is executed, which takes e_L^s.
(3) Before T_j is scheduled to execute, in the worst case, each device (including device j itself) could raise many requests, and consequently the corresponding ISR could be executed many times. To simplify the analysis, let us assume that a device will not raise another request when it has one request still to be serviced. This can be enforced by applying the design pattern introduced in Figure 12.17(b), where the checking of the software flag is replaced by checking the queue to see the existence of a task from the same device. Then, each ISR_i $(1 \leq i \leq k, i \neq j)$ can be executed once.
(4) Since task T_j has the highest priority, it is the next task to be executed. Its execution time is e_j^s.

Thus, according to the above worst-case analysis, the event response time for the highest-priority device j is given by

$$e_L^s + e_j^s + \sum_{i=1}^{k} e_i^a.$$

The worst case happens when the task T_L happens to be the longest task, that is, $e_L^s = \max_{i=1}^{k}(e_i^s)$. Note that T_L and T_j could be the service task for the same device.

To conclude, queue-based architectures have three limitations. First, regardless of the type of queue (static, FIFO, or priority), once it is adopted by a system, the *queuing policy* is fixed; at run time the whole system has no other choice, and sticks with it. Second, no task preemption is allowed. Although in priority-queue based architecture, the system can immediately acknowledge an urgent service request, the new task cannot be completely serviced until the system finishes the current task, which in the worst case can be the longest and it may cause the urgent task to break its deadline! Third, in priority-queue-based architecture, starvation could happen to lower-priority tasks.

We will cover real-time operating systems (RTOS) in Chapter 13. The kernel of an RTOS typically offers several scheduling policies, and multiple policies can be applied simultaneously to different tasks of a system. By default, an RTOS also allows task preemption for performance reasons.

Problems

12.1 What is the length of the best-case outstanding period of the right button when T_6^d is null?

12.2 Refer to the architecture given in Figure 12.11, and explain why it will not affect the worst-case outstanding period but can increase the worst-case event response time.

12.3 If we keep all the assumptions given in Section 12.3.3 except that the clearance of a software flag is placed at the end of the corresponding service task (the case as illustrated in Figure 12.17(b)), give a general expression for the worst-case response time for a device j.

12.4 Study the UML activity diagram given in Figure 12.23. What is the worst-case response time for a signal `SigA`? As far as the worst-case event response time is concerned, is it more like an architecture with a static task queue, a FIFO queue, or priority queue? If it is priority-based, explain the priority order used by the design.

12.5 Linked list is a dynamic data structure that can be used to implement queues. Implement a task scheduler based on a FIFO queue. You do not need to use interrupts. Instead, you can program a few use cases to test the task scheduler. Each use case can simply add some function pointers to the FIFO queue.

12.6 Repeat the last problem, but use a priority queue instead.

12.7 Conduct a case study, and explain how the system can be implemented using the round-robin architecture.

12.8 Conduct a case study, and explain how the system can be implemented using the round-robin architecture with interrupts.

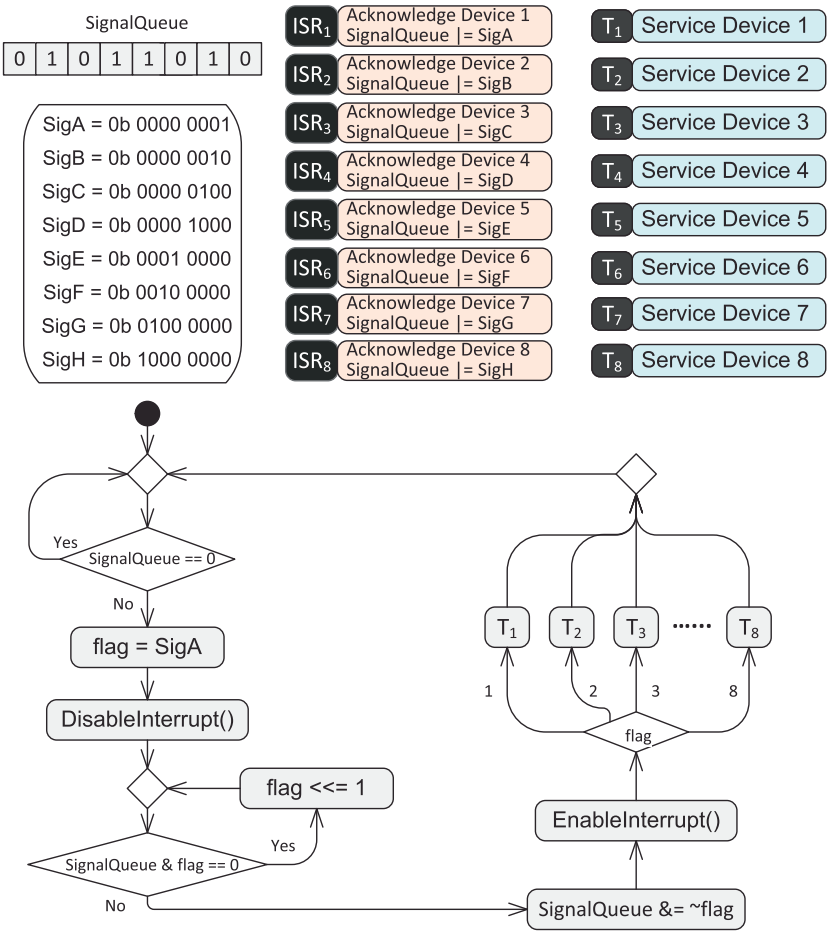

Figure 12.23
Architecture with a signal queue.

12.9 For the round-robin architecture, explain how the DAS pattern is used to handle requests from hardware devices.

12.10 For the round-robin architecture with interrupts, explain how a design pattern is used to enforce that only the first request from a device can be honored.