MIS-TIC eMbedded Information Systems

MIS-TIC
Rue Galilée 15
CH-1400 YVERDON
Switzerland

Phone +41 (0)24 557 27 86
Fax +41 (0)24 557 64 04

ZottaOS User Manual

May 2012

Disclaimer
MIS-TIC has taken every precaution to provide complete and accurate information in this document. However, due to continuous efforts being made to improve and update *ZottaOS*, MIS-TIC or HEIG-VD or The University of Applied Sciences of Western Switzerland shall not be liable for any technical or editorial errors or omissions contained in this document, or for any damage, direct or indirect, from discrepancies between this document and *ZottaOS*.
The information is provided on an as-is-basis, is subject to change without notice and does not represent a commitment on the part of MIS-TIC or HEIG-VD or The University of Applied Sciences of Western Switzerland.

Contents

Chapter
1 Introduction

What is *ZottaOS*?

Simply put, *ZottaOS* is a family of embedded preemptive real-time multitasking kernels providing hard and soft deadlines with power-aware capabilities. If you are familiar with these terms you may want to skim through the following sections.

Embedded System Designs

Embedded systems are often not recognized as computers because they are hidden inside most of our everyday objects that surround us and help us. These systems do not usually interact with the outside world through familiar interface devices such as keyboard, mouse and graphical user interface, but rather through sensors, actuators and special I/O devices. Unlike general-purpose computers, embedded systems are designed to perform a fixed number of dedicated actions. Embedded systems can therefore be optimized for size and have the least possible hardware in order to reduce its cost.

Embedded systems come in all sizes, ranging from very powerful processors comparable to desktop computers down to small microcontrollers having built-in peripherals with a few kilobytes of available memory. *ZottaOS* kernels were designed and targeted for embedded systems having scarce memory in mind.

Tasks

Embedded software is usually divided into a set of *actions*, *activities* or ***tasks***. A task is an independent stream of control that can execute its instructions independently. Tasks share the code (each can call the same functions and procedures), global data, stack and heap memory. For example, a task may be taking temperature readings while another task, residing in the same microcontroller, is reading pressure from a sensor. Whenever either reading is above a given threshold, the task turns off the apparatus it is controlling. This example illustrates an important feature of embedded software–both tasks execute ***concurrently***, i.e., they are in progress at the same time.

Concurrent tasks may be independent but all run above some common system software and need to share system resources (memory, processor and peripherals) for which they are in competition.

Kernel

Rather than lose ourselves with the details of a particular embedded system, in the following we consider an abstract application. With this simple application, we shall be able to derive the wanted features of the underlying software that provides an abstraction layer above the hardware and that controls the application's tasks–the ***kernel***.

A Simple Application

Suppose that we design an embedded system that fits into a small go-kart and does two things:

1. It monitors the kart's speed;
2. It detects the presence of moving objects in front of the kart.

Without getting lost with the details of the design, suppose that this application is broken down into 2 tasks, called *TaskA* and *TaskB* respectively. Table 1.1 gives the timing for which the tasks

should be executing using some arbitrary system unit called **timer ticks**.

Task	Execution Rate
TaskA	200 ticks
TaskB	800 ticks

Table 1.1: Task execution rate requirements.

An easy way to meet the timing requirement is to have the program execute a loop every 200 ticks. So long as the summed execution times of the tasks require less than 200 ticks, we will be able to meet the required execution rates. However this also means that *TaskB* is executed 3 times more often than necessarily. Moreover, it may be impossible to achieve the required timings with the processor at hand if we had the following execution times of each task:

Task	Execution Time
TaskA	50 ticks
TaskB	200 ticks

Table 1.2: Task execution time requirements.

Running the loop every 200 ticks can clearly not solve the problem: *TaskB* alone takes 200 ticks every 200 ticks! Yet it appears that it should be possible to satisfy the timing requirements if we consider that the processor usage per periods of 800 ticks:

TaskA requires 4 executions, i.e., $4 \times 50 = 200$ ticks

TaskB requires a single execution, i.e., 200 ticks

Total requirement = 400 ticks

The fact that the processor uses only 400 out of 800 ticks, suggests that *TaskB* should be spitted into several parts. Table 1.3 shows one of the many possibilities to meet the required timing. The first column shows the elapsed time since the beginning of an 800-tick period. The second and third columns show the task that is currently running and the number of ticks it holds the processor.

Elapsed Time	Executing Task	Execution Duration
0	*TaskA*	50 ticks
50	*TaskB_1*	100 ticks
150	idle	50 ticks
200	*TaskA*	50 ticks
250	*TaskB_2*	100 ticks
350	Idle	50 ticks
400	*TaskA*	50 ticks
450	Idle	150 ticks
600	*TaskA*	50 ticks
650	idle	150 ticks

Table 1.3: A feasible task schedule.

TaskB has been divided into 2 parts such that the first part fits into the gap left behind the first execution of *TaskA*. Note that splitting *TaskB* is a difficult task because we need to guarantee that its first part executes in no more than 150 ticks.

Even though the system is fairly simple, its design results into a structure that is poor and difficult to maintain. A change in the timing requirements or the use of another processor operating at a different speed would necessitate a restructuring of the source code. Furthermore, if we consider that the processor is used only 50% of the time, we may chose to introduce another functionality to the embedded system and have an additional task. Doing so would require redoing

the whole design rather than incrementally adding the additional feature. What is needed is a mechanism that can switch between tasks.

Multitasking

Cooperative Multitasking

The simplest technique for switching between different tasks is known as ***cooperative multitasking***. Kernels that support it provide the necessary services to keep track of the different tasks and system calls to switch between them; however, the kernel doesn't take any initiative to do so and it is up to the executing task to surrender the processor with an explicit statement.

Although embedded software designs with cooperative multitasking are better than the loop approach illustrated above, they have the same major weakness–the system can become unresponsive for a very long time. Going back to our previous example with 2 tasks, it is up to the designer to chose where *TaskB* should relinquish the processor. As can be easily seen, a cooperative multitasking system is of no help to our application.

Preemptive Multitasking

When switching between the different tasks is under the control of the kernel, the kernel is called a **preemptive multitasking** kernel. *ZottaOS* is an example of a preemptive multitasking kernel. The term *preemptive* simply means that the kernel determines when each task should execute and for how long. Preemptive multitasking prevents one task from taking up all the processor's time and is a fundamental step toward creating fast, reliable and responsive applications.

Writing a *ZottaOS* Application

Periodic Tasks

Now that we have introduced all the needed terminology, we are ready to create a *ZottaOS* application. Our simple go-kart coded in *ZottaOS* is illustrated in Figure 1.1.

```
1   #include "ZottaOS/ZottaOS.h"
2   void TaskA(void *);     /* Task prototype lines. */
3   void TaskB(void *);
4   int main(void)
5   {
6     /* Initialize the application and then create the two
7     ** tasks with their execution times and rates. */
8     OSCreateTask(TaskA,0,200,200,NULL);
9     OSCreateTask(TaskB,0,800,800,NULL);
10    /* Perform other initializations: configure peripherals
11    ** according to the specifications of the application. */
12    /* Initialize the timer. */
13    return OSStartMultitasking(); /* Launch the kernel. */
14  } /* end of main */
15  void TaskA(void *argument)
16  {
17    /* Code of the task. */
18    OSEndTask();
19  } /* end of TaskA */
20  void TaskB(void *argument)
21  {
22    int i;
23    static int j;
24    /* Code of the task. */
25    OSEndTask();
26  } /* end of TaskB */
```

Figure 1.1: Go-kart application skeleton with *ZottaOS.*

This elementary program creates 2 periodic tasks, which are declared on lines 2 and 3. Under *ZottaOS*, tasks are functions that take a single parameter. When a periodic task is created via

`OSCreateTask`, *ZottaOS* creates all the needed resources to execute the task as well as specifying its attributes:

```
BOOL OSCreateTask(void task(void *), UINT16 periodCycles,
                  INT32 periodOffset, INT32 deadline, void *argument)
```

The first parameter indicates the function that is to be considered as a task by *ZottaOS*. The next 2 parameters define the period length of the task. Because certain applications may require a timer with a 1 MHz oscillator, 32-bit periods are insufficient to represent more than a length of half an hour. This is the reason why an additional 16-bit quantity is added.

For Your Information

ZottaOS uses an internal counter to represent the system time and which is a 32-bit quantity. To avoid a wraparound, all internal temporal values are shifted when the internal system clock exceeds 2^{30}.

The period of a task is therefore represented by 16 + 30 bits. With a 1 MHz timer oscillator, a maximum period of 2 years is possible–more than enough for all applications.

The fourth parameter of `OSCreateTask` specifies the deadline of the task, which is always greater than the execution time of the task and must be smaller or equal to the minimum of 2^{30} and the period of the task.

Task Argument

Finally the last parameter of `OSCreateTask` is an argument that is passed to the task, and it receives this argument each time it is executed. For instance,

```
OSCreateTask(TaskC,...,(void *)4);
...
void TaskC(void *argument)
{
  int i = (int)argument;
  /* i has the value 4 */
  ...
  OSEndTask();
} /* end of TaskC */
```

Because task functions define a template of what the task has to do, this argument can be used to distinguish the different tasks that use the same code and need to be differentiated by their argument. For example, if an application uses 2 different UART peripherals to periodically output the same byte, a task function that has the peripheral identifier as argument can be used by both tasks, each directing the byte to the UART specified by the argument. The only caution to keep in mind when using arguments is that the lifetime of the main function ceases to exist once control is transferred to *ZottaOS* by function `OSStartMultitasking`. Hence pointers to local variables declared in `main` must never be passed as arguments to tasks. However pointers to global variables or to data dynamically allocated by `main` are allowed. For instance, the code below illustrates a proper usage of a pointer argument, which also avoids declaring a global data structure:

```
typedef struct {
   ...
} TASKDATA;
int main(void)
{
  TASKDATA *td = (TASKDATA *)OSMalloc(sizeof(TASKDATA));
  /* Initialize td */
  ...
  OSCreateTask(TaskD, ...,td);
  ...
  return OSStartMultitasking(); /* Start the kernel. */
} /* end of main */
void TaskD(void *argument)
{
  TASKDATA *data = (TASKDATA *)argument;
  ...
} /* end of TaskD */
```

Once the tasks have been created, *ZottaOS* is ready to take over and schedule the tasks according to their attributes. This is done via function call `OSStartMultitasking` on line 13.

For Your Information

ZottaOS does not support dynamic task creation. In other words, all tasks must be created before calling `OSStartMultitasking`.

For Your Information

All functions supplied by *ZottaOS* are prefixed with the 2 letters `OS` or by the 3 characters `_OS`. *ZottaOS* is largely written in C but some parts are in assembler. Because these are separate files, they share 6 global references. These are prefixed with the 3 characters `_OS`. We therefore do not recommend application variable and function identifiers that start with either these 2 prefixes.

Task Instances

Notice that both tasks (lines 15 through 26 of Fig. 1.1) have no loops, and yet they are periodic. As explained in Chapter 2, a periodic task is in fact an infinite sequence of **instances**, and each time that a task reaches the beginning of its period, a new instance of the task is released, conceptually created, and then it becomes eligible for execution. Once the instance calls `OSEndTask`, the memory resources taken from the system are freed and the release time of the next instance of the task is scheduled. The task then waits for its next release.

Although the scheme may seem complicated, it does have its advantages:

1. Tasks do not have a private run-time stack that the software designer has to establish and size.
2. Tasks use the minimal amount of stack memory. For example, if there are 2 tasks having the same period, i.e., they are simultaneously released; the second task can only start its execution once the first one has finished. Because the first task can only be released after the second task instance has also finished, the maximum run-time stack size is then equal to the task requiring the most memory, and not the summed maximum memory usage of both tasks.

Variable Declaration

Tasks may also declare and access local variables and constants. For example, variable `i` declared locally in `TaskB` of Figure 1.1 has a lifetime equal to that of an instance of `TaskB`. Every time that a new instance is created, a new instance of the variable `i` is also created on the run-time stack (this may not always be the case if the processor has plenty of registers to hold all or part of these variables). If on the other hand, a task instance needs to have variables that retain their value from one task instance to another, the local variables may be prefixed with the *static* keyword. Variable `j` of `TaskB` illustrates such a variable.

As mentioned, a task can also serve as a template for different tasks. In this case, there would be one task definition (code), for example `TaskB` of Figure 1.1, but several references to the same task function in the many calls of `OSCreateTask`. The created tasks would share the same code, and each may have different periods and deadlines, however each will share the same static variable `j`. We do not recommend static variables in task templates as these may lead to concurrency inconsistencies.

Although the main program of Figure 1.1 seems simple, it lacks 2 microcontroller specific initializations:

1. watchdog disabling,
2. setting the timer source and its rate.

These initializations are not provided by *ZottaOS* as they can change from one application to another and are highly dependent upon the microcontroller at hand. It is a good practice to disable the watchdog as soon as main starts its execution, but the timer initialization can be done anywhere prior to calling `OSStartMultitasking`. The *ZottaOS* distribution comes with samples that can come in handy to see how these 2 initializations are actually done.

Event-Driven Tasks

Besides periodic tasks, *ZottaOS* also supports tasks **event-driven** or **aperiodic** tasks. These are tasks that require an explicit activation and when they have hard deadlines and a minimum interarrival time restriction to guarantee that the deadline of the task can always be met, they are called *sporadic tasks*. Only sporadic tasks are supported by *ZottaOS*, and a minimum interarrival time and a processor load characterize them. The processor load is defined as the processor bandwidth that is set aside to handle the task.

There are 2 ways to have aperiodic processing:

1. Do the processing in an interrupt handler;
2. Trigger an event that schedules a new task instance providing this processing.

The first possibility is self-explicit. The second possibility is slightly more involved but allows extra flexibility. Aperiodic tasks are created by a specific function that indicates which event is associated with the task. When created or when terminated, an aperiodic task becomes blocked until it is signaled. Signaling is done via function `OSScheduleSuspendedTask(eventId)`. Once signaled, a new aperiodic instance is created and scheduled.

The example program given in Figure 1.2 illustrates these function calls. Line 4 defines an opaque event that is used to signal aperiodic task `AperiodicTask` and which is created on line 10 and associated to the aperiodic task at its creation time (line 12) via function `OSCreateSynchronousTask`:

```
BOOL OSCreateSynchronousTask(void task(void *), INT32 workLoad,
                             void *event, void *argument)
```

The first parameter indicates which function is the aperiodic task to create. The next parameter defines the minimum interarrival time of the task. The third parameter defines the event that is associated with the aperiodic task. Every time that the task instance terminates (line 29), the task is enqueued in a FIFO queue that is part of the event. The final parameter of `OSCreateSynchronousTask`, like for `OSCreateTask`, is the argument of the task. Chapter 2 and the reference part of this manual explain in depth this function.

On line 22 of `SignalerTask`, the aperiodic task is signaled through event `Event` and via function `OSScheduleSuspendedTask`. If an aperiodic task is currently suspended for the event, the task is immediately scheduled. Otherwise, if there are no suspended aperiodic tasks associated with the event, the signal is memorized so that the next time that an aperiodic instance terminates, it can immediately be scheduled. An aperiodic task that is signaled can have a higher priority compared to the signaler task. In this case, the aperiodic task takes over and finishes before the signaler task.

```
1   #include "ZottaOS/ZottaOS.h"
2   void SignalerTask(void *); /* Task that wakes-up AperiodicTask. */
3   void AperiodicTask(void *);
4   void *Event;               /* Event associated with AperiodicTask */
5   int main(void)
6   {
7     /* Create a periodic task that triggers an event. */
8     OSCreateTask(SignalerTask,0,800,800,NULL);
9     /* Create an event that will be triggered by SignalerTask */
10    Event = OSCreateEventDescriptor();
11    /* Create an aperiodic task that processes event Event */
12    OSCreateSynchronousTask(AperiodicTask,2000,Event,NULL);
13    /* Perform other initializations: configure peripherals
14    ** according to the specifications of the application. */
15    /* Initialize the timer. */
16    return OSStartMultitasking(); /* Boot up kernel. */
17  } /* end of main */
18  void SignalerTask(void *argument)
19  {
20    /* Do something useful here, i.e., what this task was designed for. */
21    /* Wake-up the aperiodic task. */
22    OSScheduleSuspendedTask(Event);
```

```
23        /* Do something else useful. */
24        OSEndTask();
25      } /* end of SignalerTask */
26      void AperiodicTask(void *argument)
27      {
28        /* Code of the task. */
29        OSSuspendSynchronousTask();
30      } /* end of AperiodicTask */
```

Figure 1.2: Aperiodic tasking skeleton with *ZottaOS.*

For Your Information

Signals are not counters. For example, two consecutive calls to `OSScheduleSuspendedTask` when the event queue is empty memorize only one event occurrence and not two.

Finally, notice that the distinction between a periodic task and an aperiodic task in terms of their code is the way these tasks terminate. An aperiodic instance needs to call `OSSuspendSynchronousTask` so that it can insert itself into the event queue or grab the signal that is intended for it (line 29).

Power-Awareness

Today's embedded systems must necessarily take power consumption into consideration and treat consumption as a primary figure of merit. In this respect, *ZottaOS* provides static and dynamic power management techniques that are self-monitored and exploit new capabilities of cutting-edge microcontrollers that allow voltage and frequency scaling. Although these techniques are fairly complex in nature, one of the key attributes of *ZottaOS* is that these techniques can be applied with virtually no designer assistance.

More About *ZottaOS*

From an application point of view, a task is simply a sequence of code consisting of an activity that is invoked whenever the activity is to be provided. A task may thus be thought as a sequence of actions whose net result is the provision of some function. However from an operating system point of view, a task is a schedulable entity consisting of code and data, and characterized by attributes and dynamic states. In Chapter 5, we describe the inner workings of *ZottaOS* and namely how it keeps track of the tasks and schedules them. That chapter will come in handy to anyone that wishes to fully understand how power-awareness schemes can be implemented. That chapter also provides algorithms that can help software designers develop new concurrent programs that are particularly efficient when interacting between tasks.

Chapter 2

Real-Time Basics

Introduction

Most books on real-time systems begin by defining what is *real-time*, but doing so is not very helpful. Nevertheless, to open this chapter and to keep with tradition, we can give the following definition:

> **real time** (noun)
> the actual time during which a process or event occurs: *recent natural experiments in which creolization by children can be observed **in real time*** | *information updated in real time.*
> 1. [as adj.] Computing of or relating to a system in which input data is processed within milliseconds so that it is available virtually immediately as feedback, e.g., in a missile guidance or airline booking system: *real-time signal processing.*
> 2. informal a two-way conversation, as opposed to the delay of written correspondence: *a place where two people can talk to one another **in real time**.*
>
> (Oxford American Dictionaries)

Timing Requirements

Regardless of the system at hand, we would of course like it to respond virtually immediately. If we are using our computer to compute something, we care about its response time. If we are to control an air bag in a car we care if the air bag isn't activated after the detection of contact but before the crash. The distinction between real-time systems lies with the way we engineer these systems so that they meet their **timing requirements**. Real-time systems can broadly be classified into 2 categories:

1. Those where the typical demands on the system can easily be met without any special technique or analysis. Here, we simply want to go as fast as possible.
2. Others need to be carefully designed so that they meet their demands because to do otherwise would lead to failure.

Real-Time Definition

A more precise definition would state that a real-time system is a system that not only depends on the correctness of its actions and on its ability to timely react to events, but also on the actual time at which these actions are produced. The main difference between real-time systems and non real-time systems is that the former must complete within a **deadline** when an action is to be undertaken. In other words, a deadline is the maximum time permitted to finish its execution. In real-time applications, a result produced after its deadline is not only late, but can be dangerous.

Hard and Soft Real-Time

We can also classify real-time systems into two types based on how important the timing requirements are met. For **hard real-time systems**, all actions must meet their respective deadlines. The other type of real-time systems is called **soft real-time systems** and can occasionally fail to meet to meet their timing requirements without jeopardizing the correct overall system behavior. These systems are more difficult to specify because is it often too difficult to identify which deadlines can be relaxed and how often they can be relaxed. We shall investigate soft real-time systems in the second half of this chapter.

Concurrency

Real world applications are concurrent: there are many events that can occur at the same time. If there is a single microcontroller to handle the events then the processor must execute them concurrently. In other words, each event requires some amount of computation time and has some time-window for which the processor must respond. Because there are different computations to choose when a number of events occur, the obvious question is: Which one should be chosen and when? Consider the following simplified example.

Two events e_1 and e_2 each require a computation time of 10 ms and respectively have deadlines of 20 ms and 15 ms. Assume that both events are triggered at time t. If event e_1 is processed first, it completes its execution by $t + 10$. Event e_2 then starts its execution at time $t + 10$ and finishes at $t + 20$ missing its deadline. But if event e_2 was first processed, e_2 would have finished at time $t + 10$ and e_1 then finishes at $t + 20$. Both deadlines would have been met.

Choosing which event to process is known as **scheduling**, and most real-time engineering deals with scheduling algorithms and their analysis.

Another important question is finding how much computation time an event requires. Answering this question is a whole field of real-time engineering in itself. Because in *ZottaOS* we are essentially dealing with small embedded systems, we find that measuring the code on a test-rig is sufficient provided that worst-case assumptions are taken.

The Scheduling Problem

A scheduling approach can be separated into three activities: **offline configuration**, **run-time dispatching** and **a priori analysis**. Scheduling approaches differ in how much effort is devoted in each activity.

Run-Time Dispatching

Run-time dispatching deals with the actual switching between the computations of the events at run-time. The scheduling algorithms implemented in *ZottaOS* assume that tasks execute code in response to their associated events. In this sense, a task is a thread of control that is invoked by some event. Once an event occurs, its associated task is said to be **ready**, and it stays in that state until it completes its execution at which time it waits until the event reoccurs. So the run-time dispatcher algorithm actually decides when to switch between ready tasks, and to make these decisions, it uses knowledge about the current state of the system and the information determined offline.

Offline Configuration

The offline configuration activity generates static information used by the run-time dispatcher algorithm. Under *ZottaOS* this basically involves providing the task characteristics. All this information is specified at task creation time in the main program prior to calling `OSStartMultitasking`.

A Priori Analysis

With hard real-time systems, we need to meet all task deadlines. But there can be situations when it is impossible for the run-time dispatcher to ensure that all deadlines are actually met. For instance, if two events occur at the same time and both require a computation time equal to their deadlines, obviously it is impossible to schedule the events so that they both meet their deadlines. In this case we say that the schedule is **infeasible** or that the set of events is **unschedulable**. The aim of a priori analysis is to verify the application before it starts running to check that there cannot be any situation where deadlines can be missed. A priori analysis depends of course on the scheduling algorithm that is implemented in the run-time dispatcher. With *ZottaOS-Hard*, there are two such algorithms to choose from: **static** (or **fixed**) **priority scheduling** and **dynamic priority scheduling**. Both of these scheduling algorithms are online schedulers or priority-driven. These algorithms do not pre-compute the schedule of the tasks but instead simply queue the tasks so that they can wait their turn to be executed. The same applies to *ZottaOS-Soft*.

Before presenting these algorithms and their respective a priori analysis, we first refine the notion of periodic tasks on which all these algorithms are based upon.

Periodic Tasks

Periodic activities represent the most significant computational load in a real-time control system. Activities such as actuator regulation, signal acquisition, filtering, sensory data processing and monitoring all need to execute with a frequency rate determined by the applications requirements.

A **periodic task** is characterized by an infinite sequence of **instances**. Each instance is itself characterized by a release time and a deadline. The release time of the ith instance of a task represents the time at which the task becomes ready for execution for the ith time. The interval of time between two consecutive release times is equal to the task's **period**. The absolute deadline of the ith instance, denoted by $d(i)$, represents the time at which the instance has to complete its execution. Letting P denote the period of task i, we have

$$(i-1)\,P < d(i) \le i\,P \qquad (i > 0)$$

Because it is easier to characterize a **relative deadline** D within a period, the previous equation becomes

$$(i-1)\,P < (i-1)\,P + D \le i\,P \quad (i > 0)$$

The above inequality simply states that the ith deadline of a periodic task can only be defined after its ith instance is released and must occur before its next instance is released.

Fixed Priority Scheduling

Task Preemption

In fixed priority scheduling, the run-time dispatcher ensures that at any time, the highest priority task that can be executed is actually running. If we have a task with a low priority running on the processor, and a higher priority task is released (i.e., some event occurred), the low priority task is suspended and the high priority task starts running (i.e., the low priority task is *preempted*). If while the high priority task is running, a medium priority task is released, the dispatcher leaves it unprocessed and the higher priority task carries on running and at some later time finishes its computation. The medium priority task then starts executing to finish at some even later time. Only when both tasks have completed their execution can the low priority task resume its own execution. The low priority task can then continue executing until either a higher priority task arrives or until it has completed its work.

Analysis requires a model that only considers relevant and limited aspects of the system behavior. In the early 1970's, Lui and Layland in an influential paper [LUI73] studied and produced analysis for fixed priority scheduling for the following model: all tasks are periodic and have deadlines equal to their periods; tasks are not allowed to be blocked nor can they suspend themselves. This paper coined the term **rate monotonic** (RM) for how priorities are chosen: priorities are set monotonically with rate, i.e., tasks with shorter periods should be assigned higher priority. Priorities are also assumed to be unique, and when there are tasks that have the same period, these are broken arbitrarily. Since the periods of the tasks are usually kept constant, the RM algorithm implements a static assignment, in the sense that task periods are determined at task creation and remain unchanged for the whole application run. Lui and Layland also assumed that tasks are strictly periodic, but in fact the analysis is also correct if a task instance i is released at most once every P_i. They also assumed that the task execution times are constant, but in fact they only need to be bounded.

Utilization Bound

In their paper, Lui and Layland showed that RM is optimal among all static priority scheduling algorithms, in other words, if a task set is not schedulable by RM, then the task set cannot be feasibly scheduled by any other fixed priority assignment. Another important result proved by the same authors is that a set of n tasks is schedulable by RM if

$$\sum_{1 \le i \le n} \frac{C_i}{P_i} \le n\left(2^{1/n} - 1\right) \qquad \text{(Eq. 2.1)}$$

where C_i and P_i represent respectively the worst-case execution time and the period of task i. The quantity on the left-hand side of the equation represents the **processor utilization factor** and denotes the fraction of time used by the processor to execute the entire task set; the right-hand side of the equation is known as the **utilization bound**.

Table 2.1 shows the utilization bound $n(2^{1/n} - 1)$ for n from 1 to 10. As can be seen, the bound decreases with n and, for large n, it tends to $\ln 2$, which is approximately equal to 0.693.

n	$n(2^{1/n} - 1)$
1	1.000
2	0.828
3	0.780
4	0.757
5	0.743
6	0.735
7	0.729
8	0.724
9	0.721
10	0.718
∞	0.693

Table 2.1: Maximum utilization bound for RM scheduling.

The above analysis only gives a sufficient condition to guarantee a feasible schedule under the RM algorithm. If the task set has a utilization below the bound then all deadlines will be met (i.e., the set of tasks is schedulable). However the above analysis is not exact: if the utilization is higher than the bound, the task set may in fact be schedulable and yet be rejected. For instance the 3-task set below is schedulable as we shall later see.

Task	P	D	C	Priority
1	300	300	100	high
2	400	400	100	medium
3	520	520	120	low

Table 2.2: Rejected task set according to Eq. 2.1.

The task utilization of this set is 81.4% and so the above scheduling test would reject the task set.

Exact Analysis of Fixed Priority Scheduling

Worst-case response time

An exact analysis for RM is given in [JOS86] and is based on finding the worst-case response time of each task. The worst-case response time, R_i, of task i is the longest time ever taken by an instance of task i from the time it is released until the time it completes its required computation.

Given the worst-case response time for each task, the scheduling test is trivial: we need only verify that $R_i \le D_i$ for each task i. This also means that we can lift the restriction that task deadlines need to be equal to their periods. Another important consequence of being able to calculate worst-case response times is that we can handle *sporadic tasks*. Sporadic tasks are those that run in response to some irregular event, but generally have short deadlines compared to their minimum inter-arrival time. With the model we described in the previous section, an infrequent sporadic task would either have to be given a long deadline (which would entail little system reactivity), or it would have to be given a short period (which would be very inefficient since the analysis would most likely reject most task sets when in reality they are schedulable).

The worst-case response time of task i is given by [JOS86]:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil C_j \qquad \text{(Eq. 2.2)}$$

where $hp(i)$ is the set of tasks having higher priority than task i, and the summation is over all the tasks of higher priority than i. The function $\lceil x \rceil$ is the ceiling function and yields the smallest integer greater than or equal to x. For instance, $\lceil 5 \rceil = 5$, and $\lceil 5.1 \rceil = \lceil 5.9 \rceil = 6$. The above equation can be solved by an iteration approach, starting with $R_i^{(0)} = C_i$ and terminating when $R_i^{(s)} = R_i^{(s-1)}$.

With this analysis, a value of R_i is valid only if $R_i \leq P_i$. When the response time is greater than the period of a task and goes beyond the end of the period, the task can also be delayed by a previous instance of itself, something that is not taken into account in the above equation. This also means that the deadline of an instance cannot be greater than the period of the task. Notice also that the analysis depends only on the task characteristics C and P, and of the priority ordering of the task set.

Table 2.3 summarizes the notions we have seen so far.

Notation	Meaning
C_i	Worst-case execution time of task i
D_i	Deadline of task i
P_i	Period of task i
R_i	Worst-case response time of task i
$hp(i)$	Set of tasks having higher priority than i

Table 2.3: Notations

Before concluding this section, let's apply the analysis to the task set given in Table 2.2 and that was found to be infeasible with the Lui and Layland utilization bounds.

Task 1 instances have the highest priority and the set $hp(1)$ is empty, and so $R_1 = C_1 = 100 < D_1 = 300$.

For R_2, $hp(2)$ only contains Task 1. The first approximation is $R_2^{(0)} = C_2 = 100$. And the following iterations are:

$$R_2^{(1)} = C_2 + \left\lceil \frac{R_2^{(0)}}{P_1} \right\rceil C_1 = 100 + \left\lceil \frac{100}{300} \right\rceil 100 = 200$$

$$R_2^{(2)} = C_2 + \left\lceil \frac{R_2^{(1)}}{P_1} \right\rceil C_1 = 100 + \left\lceil \frac{200}{300} \right\rceil 100 = 200$$

Finally for R_3, $hp(3)$ contains Tasks 1 and 2. The first approximation is $R_3^{(0)} = C_3 = 120$, and the following iterations are:

$$R_3^{(1)} = C_3 + \left\lceil \frac{R_3^{(0)}}{P_1} \right\rceil C_1 + \left\lceil \frac{R_3^{(0)}}{P_2} \right\rceil C_2 = 120 + \left\lceil \frac{120}{300} \right\rceil 100 + \left\lceil \frac{120}{400} \right\rceil 100 = 320$$

$$R_3^{(2)} = C_3 + \left\lceil \frac{R_3^{(1)}}{P_1} \right\rceil C_1 + \left\lceil \frac{R_3^{(1)}}{P_2} \right\rceil C_2 = 120 + \left\lceil \frac{320}{300} \right\rceil 100 + \left\lceil \frac{320}{400} \right\rceil 100 = 420$$

$$R_3^{(3)} = C_3 + \left\lceil \frac{R_3^{(2)}}{P_1} \right\rceil C_1 + \left\lceil \frac{R_3^{(2)}}{P_2} \right\rceil C_2 = 120 + \left\lceil \frac{420}{300} \right\rceil 100 + \left\lceil \frac{420}{400} \right\rceil 100 = 520$$

$$R_3^{(4)} = C_3 + \left\lceil \frac{R_3^{(3)}}{P_1} \right\rceil C_1 + \left\lceil \frac{R_3^{(3)}}{P_2} \right\rceil C_2 = 120 + \left\lceil \frac{520}{300} \right\rceil 100 + \left\lceil \frac{520}{400} \right\rceil 100 = 520$$

We can see that the whole task set is schedulable since $R_i \leq P_i = D_i$ for each task i. This nicely illustrates how the simple utilization bound is not exact.

Deadline Monotonic

With the rate monotonic model of Lui and Layland, we had $D = P$ for all tasks. However we mentioned earlier that being able to calculate worst-case response times allows us to have arbitrary deadlines. We also mentioned that the analysis was valid for $R_i \leq P_i$ and hence for $R_i \leq D_i$. As we might expect, the rate monotonic priority ordering isn't very efficient when we have $D_i \leq P_i$, since an infrequent but urgent task would be given a low priority and will likely miss its deadline. For this case, it turns out that another priority ordering policy is better: the **deadline monotonic**

(DM) ordering policy [LEU82]. Just as its name implies, a task with a short deadline has higher priority than a task with longer deadline. The deadline monotonic ordering has been proved to be optimal for systems when deadlines are shorter than periods. By optimal, we mean that if a task set is infeasible under deadline monotonic ordering then it is infeasible with any other fixed priority ordering. With constant D_i for each task, DM is classified as a static scheduling algorithm.

Apart from the problem of handling sporadic tasks with tight deadlines, there are other reasons for wanting DM. In many systems we wish to control input/output jitter. In other words, we may wish to sample an input at a regular rate, but take the sample within a tight time window. In many control systems, if we fail to do this then control system theory may become invalid and the controlled equipment may behave poorly. Because of DM, we can ensure that the input and output tasks are assigned deadlines that are equal to the required time window and if the task set is schedulable then the I/O jitter can be within acceptable tolerances.

Priority-Driven Scheduling under *ZottaOS*

Single Run-Time Stack

Contrary to other RTOSs, in *ZottaOS* a new task instance is created each time a task is released. Proceeding in this fashion, it is possible to have a single run-time stack for the whole task set. Indeed, in priority scheduling, local variables of the currently active task and functions called by this task are pushed onto a run-time stack, and if this task is preempted by a system interrupt or by a higher priority task, the task can only resume its execution once the interrupt or the higher priority task has completely finished. The resumed task then finds the various data, which it pushed, at the same locations prior to its preemption. RAM memory of a microcontroller is thus efficiently used. However this approach does come with some disadvantages:

1. A task cannot be blocked for an event and relinquish the processor to another task, and then later continue its execution as soon as the event occurs without perturbing the stack. It is impossible to start a new task on the stack without completely finishing it by respecting a priority scheduling, and it is not possible to resume the execution of another task which is not at the top of the run-time stack.
2. Inter-task communications and the I/O processing must be asynchronous, i.e., they must be carried out without blocking, or in case synchrony is needed, by busy waiting which is a waste of processor cycles.
3. No task synchronization and coordination by means of locks, semaphores or by some other widespread concurrent statement are permitted. We will deal in Chapter 3 with solutions and techniques to elude this problem.

Memory effectiveness is thus obtained by limiting the possibilities that can be undertaken by the tasks. It is however possible to have data preserving their value from one instance to another. *ZottaOS* provides mechanisms to easily implement asynchronous communications, as we shall see in Chapter 3.

Taking Overheads into Account

In the previous analysis, the equations implicitly assumed that the scheduling overhead is negligible. In this section we describe how fixed priority scheduling is implemented in *ZottaOS* and how analysis can easily be changed to account for the overhead.

The **context** of a task is the set of registers that is used by the task. On microcontrollers these are temporary data registers, the program counter, the stack pointer and the status registers. When the run-time dispatcher switches from one running task to another, it must save the context of the interrupted task and load the context of the new task. A context switch can take some time to execute and there may be a fair amount of context switching during the execution of a task. These overheads can hence mount up and be quite considerable.

Each preemption of a task by another results into two context switches: one switch from the lower priority task to the preempting task and one switch back again when the preempting task

completes. Hence if a higher priority task preempts a task, say twice, there will be 4 context switches due to that task. But we know how many times a high priority task can preempt a lower priority one: Our existing analysis (Eq. 2.2) tells us this and uses it to calculate the total time a high priority task can execute, delaying the lower priority task. Also when an instance is released, it is possible that the release time occurs while a context switch is taking place. In *ZottaOS*, context switching proceeds in two phases. The first phase may be interrupted, whereas the second may not. Both phases demand approximately the same time.

Moreover in *ZottaOS*, the scheduler is implemented by an event scheduling approach. There are 2 queues of tasks maintained by the scheduler. The first is the ready queue, which is sorted by task priorities. The second is the arrival queue and it is ordered by the release time of the task instances. When an interrupt from the timer occurs, all tasks in the arrival queue with release time before or equal to the interrupt time are moved to the ready queue and the next arrival time of these released tasks are reinserted into the arrival queue. Tasks can therefore be at 2 different places at the same time but represent 2 consecutive instances: the current instance, which is in the ready queue, and the instance immediately following it, which has not yet been released. The timer then prepares the next interrupt and switches to the highest priority task in the ready queue. When a task instance invokes `OSEndTask`, the instance removes itself from the ready queue and context switches to the task instance at the head of the ready queue. Therefore, a single context switch to a lower priority task follows each task completion.

Let C_{isw} denote the time to save the context of the interrupted task and to jump to the timer service routine, also let C_{rsw} be the time to restore a context, and let C_{Timer} denote the worst-case execution time of the timer interrupt service routine. The worst-case response time of task i then becomes

$$R_i \le C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \left(C_j + C_{rsw}\right) + \sum_{1 \le j \le n} \left\lceil \frac{R_i}{P_j} \right\rceil \left(C_{isw} + C_{Timer} + C_{rsw}\right) \qquad \text{(Eq. 2.3)}$$

The first summation considers the delays in executing higher priority tasks, while the second summation takes into account the timer interrupts that are generated by the whole task set. Note that in the second summation, there is a context switch to jump into the timer code and another context switch to either proceed with a higher priority task or continue with the interrupted task.

For Your Information

On the MSP430 line of microcontrollers, constants C_{isw} and C_{rsw} are respectively 78 and 63 cycles.

A Final Remark

The strongest point of fixed priority algorithms is their predictability. It is possible to predict the whole process that will be executed. But predictability has a cost, the cost of not achieving 100% processor utilization except for some special cases. One notable exception where 100% utilization can be attained is when the periods of the task set are *harmonic*, i.e., each task has a period that is an exact multiple of every other task that has a shorter period. For example, tasks with periods of 100, 200, and 400 can achieve a 100% utilization if $\Sigma\ C_i / P_i = 1$.

Harmonic Task Set

Dynamic Priority Scheduling

Earliest Deadline First Scheduling

Since the RM and DM are both optimal among fixed priority assignments, the scheduling bounds that can achieve higher processor utilization can only be done by a dynamic priority assignment. The **Earliest Deadline First** (EDF) scheduling algorithm consists in selecting the task instance with the earliest absolute deadline (among all ready task instances). In contrast to fixed priority scheduling, EDF does not have a complete knowledge of the task set and assigns different priorities to individual instances of each task. The priority changes as instances are released and completed, so the task's priority changes in respect to other tasks of the set that is scheduled. The EDF algorithm is typically a preemptive algorithm, in that a newly released task instance can preempt the running task if its absolute deadline is earlier. In *ZottaOS*, the ready

queue is implemented as a sorted list of task TCBs where the head of the queue is the task instance having the earliest absolute deadline.

Liu and Layland [LUI73] proved that a set of n periodic tasks each having relative deadlines equal to their periods is schedulable by EDF if and only if

$$\sum_{1 \le i \le n} \frac{C_i}{P_i} \le 1 \qquad \text{(Eq. 2.4)}$$

It is worth mentioning that the above condition is necessary and sufficient to guarantee a feasible schedule for the task set. In other words, if this condition is not satisfied, no algorithm is able to yield a feasible schedule. Without taking into account system overheads and context switching, the dynamic priority assignment allows EDF to fully exploit the processor, reaching up to 100% of the available processing power. When the task set has a processor utilization factor (the summation in Eq. 2.4) less than 1, the residual fraction of time can be efficiently set aside to handle aperiodic tasks. In addition and in contrast with RM or DM, EDF generates a lower number of context switches, and therefore causes less run-time overhead. Also, it has been shown that EDF is optimal in that, if there exists an algorithm that can successfully schedule a set of tasks with arbitrary deadlines and execution times on a single processor, EDF will succeed in scheduling the task set.

Under EDF, feasibility analysis is much more complex than with fixed priorities when periodic tasks have shorter relative deadlines compared to their periods. In the method proposed in [BAR90], a task set is schedulable by EDF if and only if, in every time interval of length L, the overall computational demand is not greater than the available processing power, i.e.,

$$\forall L > 0,\ H(L) = \sum_{1 \le i \le n} \left\lfloor \frac{L + P_i - D_i}{P_i} \right\rfloor C_i \le L \qquad \text{(Eq. 2.5)}$$

Hyperperiod

The summation in the inequality, $H(L)$, represents the amount of work that must be completed before time L. The values of L that must be verified can be limited to task deadlines no longer than a **hyperperiod**, i.e., the least common multiple (LCM) of the task periods. The function $\lfloor x \rfloor$ is the floor function and it is equal to the largest integer smaller than or equal to x.

Before concluding this section, it is worthwhile illustrating the use of Eq. 2.5 by an example, for instance for the task set given in Table 2.4.

Task	P	D	C
1	200	150	100
2	300	250	125
3	600	580	30

Table 2.4: Schedulable task set for EDF.

Table 2.5 gives the values of $H(L)$ for each deadline during the first hyperperiod from 0 to 600. As can be seen, all $H(L)$ are never greater than L, and hence the task set in Table 2.4 is schedulable.

L	Task Deadlines	$H(L)$
150	1	100
250	2	225
350	1	325
550	1 and 2	550
580	3	580

Table 2.5: Application of Eq. 2.5 for the task set of Table 2.4.

Equation 2.5 can also be completed in regards to the overhead due to scheduling a given task set. Using the same notation describing the overheads that we introduced for RM and DM, and also using the same reasoning that allowed us to establish Eq. 2.3, we can write

$$\forall L > 0,\ H(L) = \sum_{1 \le i \le n} \left\lfloor \frac{L + P_i - D_i}{P_i} \right\rfloor (C_i + C_{rsw}) + \sum_{1 \le i \le n} \left\lceil \frac{L}{P_i} \right\rceil (C_{isw} + C_{Timer} + C_{rsw}) \le L \quad \text{(Eq. 2.6)}$$

In this equation, the second summation represents the overhead due to the timer. The timer interrupts the execution of the tasks at each task release time and for which 2 context switches are necessary. The first is to preempt the currently active task and to jump to the interrupt service routine while the second is to either resume the current task or to begin executing a newly released task. Once a task completes its execution, it too requires a context switch to start the next task in the ready queue.

Sufficient Acceptance Test

The problem with equations 2.5 and 2.6 is that the number of times these equations need to be applied may be excessive, typically the number of deadlines in a hyperperiod. This explains why we chose the period of Task 3 in Table 2.4 to be a multiple of the other 2 tasks. To alleviate this problem, U.C. Devi [DEV03] proposed a fast and accurate sufficient acceptance test, i.e. an accepted task set is certain to be feasible, but a rejected set may actually be schedulable. The procedure is to arrange the task set in order of the non-decreasing relative deadlines and then for each task k, starting with the smallest relative deadline, verify the following condition:

$$\sum_{1 \le i \le k} \frac{C_i}{P_i} + \frac{1}{D_k} \sum_{1 \le i \le k} \frac{P_i - D_i}{P_i} C_i \le 1 \quad \text{(Eq. 2.7)}$$

When the tasks in the set all have their deadlines at the end of their period, Eq. 2.7 reduces to Eq. 2.4. To illustrate Eq. 2.7, we can reuse the task set given in Table 2.4. For $k = 1$,

$$\frac{C_1}{P_1} + \frac{1}{D_1} \frac{P_1 - D_1}{P_1} C_1 = \frac{100}{200} + \frac{200 - 150}{200} \frac{100}{150} = \frac{2}{3} < 1$$

For $k = 2$,

$$\frac{C_1}{P_1} + \frac{C_2}{P_2} + \frac{1}{D_1}\left(\frac{P_1 - D_1}{P_1} C_1 + \frac{P_2 - D_2}{P_2} C_1 \right) = \frac{100}{200} + \frac{125}{300} + \frac{1}{250}\left(\frac{200 - 150}{200} 100 + \frac{300 - 250}{300} 125 \right) = \frac{11}{10} > 1$$

and the test fails even though the task is schedulable.

However if we reduce C_1 to 40, the task set is accepted. For $k = 1$, the test succeeds since Task 1 introduces a lighter load, and for $k = 2$, we have

$$\frac{40}{200} + \frac{125}{300} + \frac{1}{250}\left(\frac{200 - 150}{200} 40 + \frac{300 - 250}{300} 125 \right) = \frac{1}{5} + \frac{5}{12} + \frac{1}{250}\left(10 + \frac{125}{6} \right) = \frac{36}{50} < 1$$

And finally for $k = 3$,

$$\frac{1}{5} + \frac{5}{12} + \frac{C_3}{P_3} + \frac{1}{D_3}\left(10 + \frac{125}{6} + \frac{P_3 - D_3}{P_3} C_3 \right) = \frac{1}{5} + \frac{5}{12} + \frac{30}{600} + \frac{1}{580}\left(10 + \frac{125}{6} + \frac{600 - 580}{600} 30 \right) = \frac{837}{1160} < 1$$

To account for the context switching overhead, we can charge the cost of executing `OSEndTask` directly to the task's execution time. This makes up for the time needed to resume or start a lower priority instance. Next, each task release creates a timer interrupt and which either returns to the preempted instance or instantiates the released task. This compensates for each task arrival and using our previous notation amounts to $C_{isw} + C_{Timer} + C_{rsw}$.

$$\sum_{1 \le i \le k} \frac{C_i + C_{rsw}}{P_i} + \frac{1}{D_k} \sum_{1 \le i \le k} \frac{P_i - D_i}{P_i} (C_i + C_{rsw}) + \sum_{1 \le i \le n} \frac{C_{isw} + C_{Timer} + C_{rsw}}{P_i} \le 1 \quad \text{(Eq. 2.8)}$$

Faster feasibility analyses

Another way of reducing the number of iterations required by equations 2.5 and 2.6 is to lower the upper bound of L [RIP96]:

$$L_a = max\left\{ D_1, D_2, ..., D_n, \frac{\sum_{1 \le i \le n} (P_i - D_i) U_i}{1 - U} \right\} \quad \text{(Eq. 2.9)}$$

where $U_i = C_i / P_i$, and $U = \sum_{1 \le i \le n} U_i$, which must be strictly smaller than 1.

The length the initial busy period of the task set, when the first instance of each task is released all at once, provides yet another upper bound for L [RIP96]. This busy period ends when the processor idles or when the hyperperiod is reached and can be computed recurrently by

$$w^{(a+1)} = \sum_{1 \le i \le n} \left\lceil \frac{w^{(a)}}{P_i} \right\rceil C_i \qquad \text{(Eq. 2.10)}$$

starting with $w^{(0)} = \sum_{1 \le i \le n} C_i$, and stopping when $w^{(a+1)} = w^{(a)}$. This procedure always converges if $U \le 1$. For $U > 1$, the task set is rejected since the amount of processing to do within a hyperperiod exceeds the length of the hyperperiod.

A much less computational intensive test called Quick convergence Processor-demand Analysis (QPA) algorithm [ZHA09] works by starting with a value close to the 2 previous bounds and then iterates back to 0, jumping over deadlines that can safely be ignored. Only a small number of points are checked with this method, and compared to Eq. 2.5 iterating up to the hyperperiod of the task set, [ZHA09] have experimentally shown that the method can exponentially reduce the computational effort.

The complete QPA algorithm is given in pseudocode is given Figure 2.1. The algorithm starts with the smallest computed upper bound L (steps 3 and 4) and progresses until the smallest relative deadline D_i is reached (step 7.1) or until the task set is rejected (step 8). Starting with the latest deadline prior to the initial bound, at each iteration, the workload brought by the tasks is verified against the processor power (step 7), and a new bound is determined. Step 7.2 skips over the deadlines that are guaranteed while step 7.3 proceeds to the deadline immediately before the bound. To complete this algorithm we still need a method to find the deadline immediately preceding a given value and used by steps 6 and 7.3. For task i with $D_i < t$, the latest released instance that can have its deadline at t is $\lfloor (t - D_i) / P_i \rfloor P_i$ and its absolute deadline, d_i, is simply D_i units further. When $d_i = t$, we need to retract to the previous instance of the task, i.e. $d_i - P_i$. Figure 2.2 illustrates the resulting algorithm.

1. Compute the task set utilization U.
2. if $U > 1$ then Reject the task set.
3. Set $L \leftarrow$ busy period (Eq. 2.10).
4. if $U < 1$ then Set $L \leftarrow \min(L_a, L)$ using Eq. 2.9 to compute L_a.
5. Set $d_{min} \leftarrow$ smallest relative deadline D_i.
6. Set $t \leftarrow$ smallest absolute deadline d_i smaller than L.
7. while $H(t) \le t$ do (Eq. 2.5)
   7.1 if $H(t) \le d_{min}$ then Accept the task set and quit.
   7.2 if $H(t) < t$ then Set $t \leftarrow H(t)$.
   7.3 else Set $t \leftarrow$ smallest absolute deadline d_i smaller than t.
8. Reject the task set.

Figure 2.1: QPA algorithm.

1. Set $d_{smallest} \leftarrow 0$.
2. for each *task i* do
   2.1. if $D_i < t$ then
      2.1.1. Set $d_i \leftarrow \lfloor (t - D_i) / P_i \rfloor P_i + D_i$.
      2.1.2. if $d_i = t$ then Set $d_i \leftarrow d_i - P_i$.
      2.1.3. if $d_i > d_{smallest}$ then Set $d_{smallest} \leftarrow d_i$.
3. Return $d_{smallest}$

Figure 2.2: Procedure to find the last task deadline prior to time t.

We can apply the task set given in Table 2.4 to illustrate the algorithm. First we determine the utilization of the processor,

$$U = \frac{100}{200} + \frac{125}{300} + \frac{30}{600} = \frac{29}{30} < 1$$

Next we need L_a,

$$L_a = max\left\{D_1, D_2, ..., D_n, \frac{\sum_{1 \le i \le n} (P_i - D_i) U_i}{1 - U}\right\} = max\{150, 250, 580, 1405\} = 1405$$

The initial busy period can be computed by Eq. 2.10 or by inspecting the last line of Table 2.5 since the processor is continuously occupied with a task until time 580.
Continuing with the QPA algorithm, $d_{min} = 150$, and the first verified bound is for $t = 550$. Note $t = 580$ is not checked. Indeed, if C_3 were greater than 30 or D_3 smaller than 580, the busy period would have been greater than D_3 and the first checked value of t would have been D_3.
The successive values of $H(t)$ verified by the QPA algorithm are

$H(550) = 550$
$H(350) = 325$
$H(325) = 225$
$H(225) = 100$ and the task set is accepted.

In all, 4 calculations are required instead of the 5 shown in Table 2.5. This is not a tremendous saving, but the task characteristics of Table 2.4 were suitably chosen to reduce the number of lines in Table 2.5. (We will later show another example where the computational reduction is much more significant).

Aperiodic Tasking

Although most applications involve periodic events that need to be handled, there are certain situations that must be executed only when an external event occurs and which may arrive at irregular times through interrupts. Because these tasks have non-periodic arrival patterns, they are termed **aperiodic tasks**. Aperiodic tasks are typically used to serve random processing requirements, and the objective of a scheduling algorithm should be to minimize response time of aperiodic tasks while guaranteeing that all periodic tasks complete their executions within their deadlines. Such a guarantee can only be done by assuming that aperiodic requests, although arriving at irregular intervals, do not exceed a maximum rate, i.e., they are separated by a minimum interarrival time. In the literature, an aperiodic task characterized by a worst-case execution time and a minimum interarrival time is called a **sporadic task**.

Aperiodic Task Example

Before explaining how sporadic tasks are implemented in *ZottaOS*, it is worthwhile detailing an example that can justify the constraints we are confronted with. In a data logger application, gathering daily information transmitted at a specific hour, the data transfer task is clearly a periodic task. However, in case that the receiver is temporarily down, the data transfer should try to retransmit the data. Processing this in a loop that repeatedly tries to transfer the data until the transfer succeeds entails several problems:

1. There should be a maximum number of attempts so that the worst-case execution time of the task can be bounded;
2. It may be worthwhile to back off between each attempt.

The informed reader will have noticed that, because of the single run-time stack policy for all tasks, a periodic task cannot release the processor without calling `OSEndTask`. However an aperiodic task and a timer ISR may circumvent the last problem as shown in the code snippets of Figure 2.3.

In the proposed method, periodic task `PeriodicTransmit` attempts a first transfer and if it fails, it starts an interval timer, processed by `TimerISR` and which can schedule the event-driven task `AperiodicRetransmit`. This task can then reattempt a transfer, and if it too fails, it can reschedule itself or abandon if the number of trials becomes excessive. In this example,

```
void *RetransmitEvent; // Event to signal task AperiodicRetransmit

void PeriodicTransmit(void) {
  Attempt to transmit;
  if (fail) {
    Set up interval timer interrupt with delay period;
  }
  OSEndTask();
} /* end of PeriodicTransmit */

void TimerISR(void) {
  Clear interrupt;
  OSScheduleSuspendedTask(RetransmitEvent); // Signal RetransmitEvent
} /* end of TimerISR */

void AperiodicRetransmit(void) { // Task associated with RetransmitEvent
  Attempt to transmit;
  if (fail) {
    Set up interval timer interrupt with delay period;
  }
  OSSuspendSynchronousTask();
} /* end of AperiodicRetransmit */
```

Figure 2.3: Data transfer and retransmissions attempts for data logger example.

1. Retransmissions are not done at the interrupt level but by an application task. Proceeding otherwise would put the retransmission action amongst the highest (and non-preemptive) priority levels, and guaranteeing that all other application tasks met their deadlines under this condition would also imply an average processor utilization that is extremely low.
2. All entities are instances in the sense that none busy wait but relinquish the processor and clean the run-time stack to reappear later if necessary.

Event-Driven and Synchronous Task

3. Aperiodic tasks are event-driven and allow the implementation of a precedence graph of tasks where one task triggers one or more other tasks, which in turn can trigger others. For this reason when alluding to aperiodic tasks in *ZottaOS*, we refer to them as **event-driven** or **synchronous tasks**.

Aperiodic Tasks with RM and DM Scheduling

Under RM or DM scheduling, event-driven tasks are implemented as ordinary periodic tasks. Instead of being placed in the arrival queue, they are placed in a FIFO queue that is associated with an event, and when a call to `OSScheduleSuspendedTask` is invoked, the event-driven task is moved to the ready queue if its last execution period has expired. Otherwise the task is placed in the arrival queue where it is released as if it were a periodic task.

Aperiodic Tasks with EDF Scheduling

A number of scheduling algorithms that allow a mix set of periodic and aperiodic tasks can be found in the literature. These algorithms introduce the notion of a special purpose task, called **server**, whose **capacity** is used to serve all aperiodic requests. The server is usually scheduled by a specific algorithm designed in such a way that periodic tasks do not miss their deadlines and at the same time, the processor is allotted to the server as soon as possible in order to maintain the deadline of aperiodic tasks. Numerous algorithms have appeared differing largely in the manner in which the server is invoked and how its capacity is allocated.

Total Bandwidth Server

The scheme, called **Total Bandwidth Server** (TBS), which we have adopted in *ZottaOS*, allows sharing of the run-time stack for all periodic and event-driven tasks. The main idea behind the TBS [SPU96] is to immediately assign, whenever possible, the total bandwidth of the server each time an event-driven task is signaled. When the ith ($i > 0$) event-driven requests arrives at time t, it is assigned an absolute deadline $d(i)$ given by

$$d(i) = max\left(t, d(i-1)\right) + \frac{C_i}{U_S} \qquad \text{(Eq. 2.11)}$$

where C_i is the maximal execution time of the ith request, U_S is the server utilization factor or bandwidth, and where $d(0) = 0$. Once the request is assigned an absolute deadline, it is inserted into the ready queue and scheduled by EDF like any other task instance. Intuitively the assignment of the absolute deadlines in each interval of time never exceeds the ratio allocated by EDF to the event-driven requests. Deadlines are assigned to event-driven requests based on the rate at which the TBS server can serve them and not at the rate at which they are expected to arrive. Moreover, the aperiodic deadlines are assigned in such a way that a request must complete before the next aperiodic can start its execution. Aperiodic requests are thus processed in a first come, first served manner relative to other event-driven requests.

Letting U_P denote the processor demand of all periodic tasks, i.e., the left-hand side of Eq. 2.4, the whole task set is schedulable under EDF if

$$U_P + U_S \leq 1 \qquad \text{(Eq. 2.12)}$$

To illustrate the use of Eq. 2.11 with a high demand of event-driven tasks, we assume the task set given in Table 2.6.

Task	Aperiodic	$P = D$	C
1	no	800	200
2	no	1200	300
3	yes		100
4	yes		300

Table 2.6: Mixed periodic and event-driven task set.

The total bandwidth of the aperiodic server is set to $1 - 200/800 - 300/1200 = 0.5$, and Figure 2.4 portrays the resulting schedule when there are 2 events for Task 3 occurring at ticks 400 and 800, and a single request for Task 4 at tick 500.

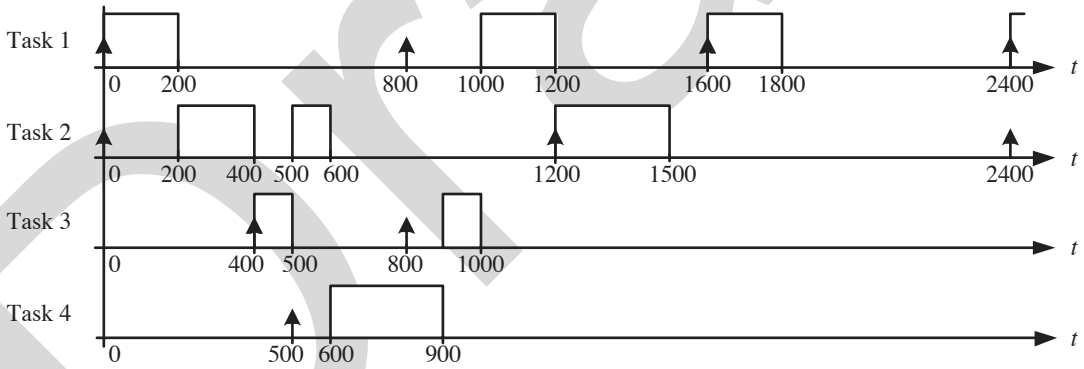


Figure 2.4: Highly prioritized event-driven tasks.
(Upward arrows represent periodic task releases and event-driven task requests)

At tick 400, the first request of Task 3 arrives and is scheduled with an absolute deadline of $400 + 100/0.5 = 600$. Since Task 3 has an earlier deadline than Task 2, the currently active periodic task, Task 3 is immediately executed and terminates at tick 500. At tick 500, the first request of Task 4 is generated. By Eq. 2.7, the absolute deadline of this task is assigned to $max(500, 600) + 300 / 0.5 = 1200$, and the task is inserted into the ready queue to be executed after Task 2. During the execution of Task 4, a second request for Task 3 arrives at tick 800 and its absolute deadline is $1200 + 100/0.5 = 1400$. Notice that Task 1 is delayed when it is released so that Task 3 can execute before it. As this preemption shows, the higher U_S becomes, the higher event-driven tasks become prioritized and hence the smaller their response time becomes.

Summary

The purpose of this chapter was to introduce real-time scheduling concepts and algorithms from a user's perspective and to provide models for analysis and prediction of embedded real-time applications.

Bibliography

[AUD91] N.C. Audsley, A. Burns, M. Richardson, and A.J. Wellings, *Hard Real-Time Scheduling: The Deadline-Monotonic Approach*, Proc. of the 8th IEEE Workshop on Real-Time Operating Systems and Software, pp. 127-132, May 1991.

[BAR90] S.K. Baruah, L.E. Rosier, and R.R. Howell, *Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor*, Real-Time Systems, Vol. 2, No. 4, pp. 301 - 324, Nov. 1990.

[DEV03] U.C. Devi, *An Improved Schedulability Test for Uniprocessor Periodic Task Systems,* Proc. 15th Euromicro Conf. Real-Time Systems (ECRTS'03), pp. 23-30, July 2003.

[JOS86] M. Joseph, and P. Pandya, *Finding Response Times in a Real-Time System*, The Computer Journal, Vol. 29, No. 5, pp. 390-395, Oct. 1986.

[LEU82] J. Leung, and J. Whitehead, *On the Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks*, Performance Evaluation, Vol. 2, No. 4, pp. 237-250, Dec. 1982.

[LUI73] C.L. Lui, and J.W. Layland, *Scheduling algorithms of Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, pp. 46-61, Jan. 1973.

[RIP96] I. Ripoll, A. Crespo, and A.K. Mok, *Improvement in Feasibility Testing for Real-Time Tasks*, Journal Real-Time Systems, Vol. 11, No. 1, pp. 19-39, July 1996.

[SPU96] M. Spuri and G. Buttazzo, *Scheduling Aperiodic Tasks in Dynamic Priority Systems*, The Journal of Real-Time Systems, Vol. 10, No 2, pp. 179-210, Mar. 1996.

[ZHA09] F. Zhang, and A. Burns, *Schedulability Analysis for Real-Time Systems with EDF Scheduling*, IEEE Trans. on Computers, Vol. 58, No. 9, pp. 1250-1258, Sept. 2009.

Chapter 3

Inter-Task Communication

Introduction

Data sharing between tasks is an important component in the design of a real-time system. At first, it may seem that this problem is trivial, but it involves subtleties that can turn a great application into a faulty non-robust one. Several methods can be employed to pass data between tasks. The simplest and fastest among these is the use of global variables. Global variables, although considered contrary to good software practices, are often used for fast operations with very little overhead. The problem with global variables is that tasks executing at higher priority can preempt lower-priority tasks and disturb their execution with incoherent data. As soon as interrupts are included into an application, the interrupt service routines usually need to communicate with some other task for it is usually neither possible nor desirable to process all the work in an interrupt routine. Therefore, interrupt routines and task code can also share one or more variables as a mean to communicate with each other, and this causes the classical *synchronization problem*.

Synchronization Problem Illustrated

Suppose that part of an application requires monitoring of two physical quantities, such as temperatures, pressures and coordinates, that must always be the same, and an alarm is signaled when they differ. The pertinent code extracts are given in Figure 3.1.

```
1  static int measuredValue[2];
2  void ReadValues(void)
3  { // Part of the interrupt service routine or called by it
4    measuredValue[0] = read from some ADC register
5    measuredValue[1] = read from some other ADC register
6  } /* end of ReadValues */
7  void MonitorMeasures(void *arg)
8  {
9    if (measuredValue[0] != measuredValue[1]) {
10     // Set off an alarm
11   }
12   OSEndTask();
13 } /* end of MonitorMeasures */
```

Figure 3.1: Faulty communication between tasks.

Procedure `ReadValues` illustrates part of an interrupt service routine and task `MonitorMeasures` periodically checks for discrepancies between the quantities read by `ReadValues`. Although at first glance there seems to be no problem with the code, `MonitorMeasures` sets off alarms when it shouldn't. This is because the if statement on line 9 is not an indivisible operation as it can be interrupted anywhere after loading the first value into a register and doing the comparison. In fact the if statement may resemble the following assembler instructions:

```
1     mov measuredValue, R14  // R14 ← pointer to table measuredValue
2     mov 0(R14), R15         // R15 ← measuredValue[0]
3     mov 2(R14), R14         // R14 ← measuredValue[1]
4     cmp R14, R15            // Are values equal?
5     jz Done                 // Yes: jump
6     // Set off an alarm
7  Done:                      // Continue with remaining code of the task
```

The compiler translates most C statements into multiple assembly language instructions that you cannot control. With regards to the code generated in assembler, task `MonitorMeasures` closely matches the following C code:

```
 7 void MonitorMeasures(void *arg)
 8 {
 9   register int val0, val1;
10   val0 = measuredValue[0];
11   val1 = measuredValue[1];
12   if (val0 != val1) {
13      // Set off an alarm
14   }
15   OSEndTask();
16 } /* end of MonitorMeasures */
```

Now suppose that `measuredValue` contains the pair of values (103,103) and an interrupt occurs setting the pair to (100,100). If this interrupt takes place between lines 10 and 11 of task `MonitorMeasures`, `MonitorMeasures` compares 2 incoherent measures and sets an alarm since `val0` = 103 and `val1` = 100 differ. *But the two measured values are the same*.

General Context

Generally speaking, tasks are divided into 2 categories: those that produce data and those that read these data. The first tasks are **writers** or **producers** whereas the second are **readers** or **consumers**, see Figure 3.2. Notice that this characterization is not as restrictive as it may seem since writers can also be the peripheral input drivers (peripherals are producers of data), and the readers can be peripheral output drivers (peripherals are data consumers).

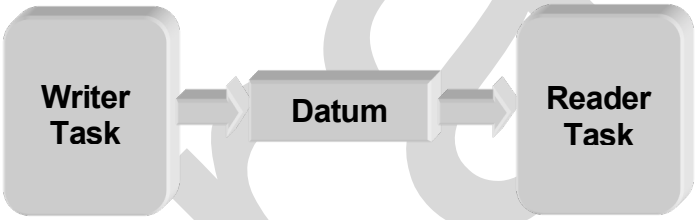


Figure 3.2: Basic communication between a writer and a reader task.

When the shared data consists of a single memory cell, applications can tolerate arbitrary task interleaving without any coordination because both reading and writing a memory location is an indivisible or atomic operation. However, when the shared data is composed of several memory locations, we may no longer assume that reading and writing are indivisible, and it is up to the RTOS to provide synchronization mechanisms that guarantee data coherence. One common approach imitating a synchronous lock is the handshake protocol illustrated below in pseudo-C.

```
TYPE Datum;          /* Shared variable */
BOOL Ready = false;  /* True if Datum holds something useful */
```

```
void Writer(void)
{
  TYPE localDatum;
  /* produce localDatum */
  while (Ready);
  Datum = localDatum;
  Ready = true;
  OSEndTask();
} /* end of Writer */
```

```
void Reader(void)
{
  TYPE localDatum;
  while (not Ready);
  localDatum = Datum;
  Ready = false;
  /* consume localDatum */
  OSEndTask();
} /* end of Reader */
```

Figure 3.3: Handshake protocol between 2 tasks sharing a global datum.

In the example of the handshake protocol, both tasks share a datum having an arbitrary type. The state of this datum is provided by Boolean variable `Ready`. A true value means that the

datum contains information that has not yet been read by the reader, whereas a false value means that the shared datum is yet to be initialized or that it was already read.

Mutual Exclusion

By analyzing the pseudocode one can notice that the shared datum is accessed in **mutual exclusion**: there can be at most one task that accesses the shared datum at any time. One can also note that the behavioral relation of the two tasks is implicitly specified–access to the shared datum strictly alternates between the tasks; the pace of execution is dictated by the slower of the two tasks. If one task must be carried out more frequently than the other, it will not be able to do so because it will have to wait that the other task accesses the datum before it can itself access it again. Both tasks are synchronous.

There are a number of other problems with the above handshake protocol:

1. If the writer is a peripheral driver activated by interruptions, waiting until the shared datum becomes accessible for writing can mean the loss of further interruptions.
2. Busy-waiting exemplified by the 2 loops is not realistic in a RTOS. When a high priority task gains access to the processor, even if it is stuck in a loop, it does not relinquish the processor. When this occurs, the tasks become **deadlocked**.
3. Classical synchronization mechanisms (locks or semaphores) can be provided with the RTOS, but then this entails that the tasks have their own run-time stack. Nevertheless, these mechanisms alone are not sufficient to solve the first identified problem.

The second problem can easily be circumvented if we replace the conditional loops by conditional statements as done in Figure 3.4.

```
TYPE Datum;           /* Shared variable */
BOOL Ready = false;   /* True if Datum holds something useful */

void Writer(void) {                    void Reader(void) {
  TYPE localDatum;                       TYPE localDatum;
  if (!Ready) {                          if (Ready)  {
     /* produce localDatum */               localDatum = Datum;
     Datum = localDatum;                    Ready = false;
     Ready = true;                          /* consume localDatum */
  }                                      }
  OSEndTask();                           OSEndTask();
} /* end of Writer */                  } /* end of Reader */
```

Figure 3.4: Corrected handshake protocol between 2 tasks sharing a global datum.

The scheme portrayed in Figure 3.4 is suitable for 2 tasks having identical periods. To guarantee that the writer is carried out before the reader, an easy workaround is to interfere with the deadlines: it is sufficient that the deadline of the writer be slightly sooner than that of the reader for the writer to the have higher priority and execute before the reader. But what if the writer task is an interrupt service routine such as procedure `ReadValues` from the monitoring example? When introducing that example we have eluded a more important question: Should the task setting off the alarm base its decision on all measures or only on the most recently read measure? Both solutions can easily be implemented in *ZottaOS* through its data sharing API.

Inter-task communication APIs provided in *ZottaOS* implement two different paradigms. The first is a first-in-first-out (FIFO) queue, which allows communication between enqueuers (writers) and dequeuers (readers) in an asynchronous manner. This is a many-to-many communication in that there can be several enqueuer tasks and several dequeuer tasks communicating through the same FIFO queue. The second is an asynchronous reader-writer paradigm where the reader-task obtains only the most recently written data by the writer task.

FIFO Queues

In a First-In-First-Out (FIFO) data structure, the first element added to the queue will be the first

one to be removed. This is equivalent to the requirement that whenever a new element is added, all elements that were added before it have to be removed before the new element can be accessed. FIFO queues can be implemented in a number of different ways. In *ZottaOS*, FIFO queues are *concurrent circular wait-free arrays* that can hold a predetermined number of buffers and have an associated pool of free buffers. The initial size of the free pool corresponds to the number of entries of the circular array.

Terminology The way the term *buffer* is used in the literature is ambiguous. Indeed, the term can mean a single block of data that can be decomposed into independent slots, each slot holding a single datum. Alternatively, the term can indicate a single block of data from a pool of buffers. In the following, *we use the term buffer to indicate a single block of memory that can hold a datum*. When used in connection with a FIFO queue, each buffer of the queue holds distinct data. And when used describing the asynchronous reader-writer paradigm, there will be one buffer (i.e., one datum), which can have multiple instances each in a distinct slot. Hence, a buffer will be synonymous with datum.

The Model An enqueuer task first acquires an empty buffer from the free pool associated with the queue, performs a number of write operations into it until it is full, and then inserts the buffer into the queue. A dequeuer task first acquires a full buffer from the queue, performs a number of reads from it until it is empty. After consuming the buffer, the dequeuer releases the buffer back to the pool of free buffers so that the buffer can be recycled by the enqueuer.

In its normal usage, the inserted buffers originate from the free pool of buffers associated with the FIFO queue. When an enqueuer requests a free buffer and none are available, it can assume that the FIFO queue is full. Because the implementation is wait-free (non-blocking with bounded execution time), the enqueuer should either suspend itself if it is an event-driven task and be awaken by an dequeuer, or terminate itself and check the state of the queue at its next execution if the task is periodic. In both cases, the task instance terminates.

Likewise, a dequeuer wishing to remove a buffer from an empty can either:

1. Do something else;
2. Suspend itself if it is an event-driven task and be awaken by an enqueuer;
3. Terminate itself if it is a periodic task and retrieve a buffer the next time it is released.

In the following we summarize and then illustrate the use of the FIFO queue API with a simple example of a single periodic writer and a single event-driven reader communicating through 3 buffers. Creation and initialization of the FIFO queue is done by means of `OSInitFIFOQueue` where the application specifies the number of entries in the circular array and the size of each buffer. Function `OSGetFreeNodeFIFO` returns a free buffer from the pool of available buffers. The enqueue function is `OSEnqueueFIFO` where an additional parameter can be associated with the enqueued buffer. This parameter can be the useful size of the datum stored in the buffer or a message type. Figure 3.5 illustrates the usefulness of this parameter. Buffers are dequeued with function `OSDequeueFIFO`, which returns a dequeued buffer and, by means of a result parameter, the information that was associated with the buffer when it was enqueued. Finally function `OSReleaseNodeFIFO` allows the dequeuer to return the buffer so that it can be recycled by an enqueuer.

Comments The FIFO queue is given by variable `fifoQueue`, and it is declared locally in the main program but passed as an argument to the tasks. The event to wake up the reader task is however a global variable because only a single argument may be passed to a task. Of course we could have chosen to put `fifoQueue` as a global variable and to hand over the event as the argument to the writer task. In this case, the event would have been declared locally in the main program.

The writer obtains a buffer from the queue at line 24. If the returned buffer is `NULL`, then the FIFO queue is full and the writer quits. Otherwise the writer fills the buffer with the datum it in-

tends to transfer to the reader. Notice that the writer does not check whether its enqueue operation is successful or not. Since there are as many spare buffers as there are entries in the queue, a writer that obtains a buffer from the free pool is guaranteed to find at least one free entry in the array representing the queue. Also note that the writer enqueues the size of the string along with the datum.

```
1  typedef struct DATUM { // Definition of the data transferred between
2    int myData;          // the tasks
3    char string[6];
4  } DATUM;
5  void Writer(void *arg);
6  void Reader(void *arg);
7  void *WakeUpReader;    // Event to wake up the Reader task
8  int main(void)
9  {
10   void *fifoQueue;     // FIFO queue shared between Reader and Writer
11   // 1. Inhibit the watchdog timer
12   // 2. Initialize the system clock
13   // 3. Configure peripherals
14   // 4. Create the application tasks
15   fifoQueue = OSInitFIFOQueue(3,sizeof(DATUM));
16   OSCreateTask(Writer,0,1000,200,fifoQueue);
17   WakeUpReader = OSCreateEventDescriptor();
18   OSCreateSynchronousTask(Reader,0,600,WakeUpReader,fifoQueue);
19   // Do other initializations here
20   return OSStartMultitasking();
21 } /* end of main */
22 void Writer(void *fifoQueue)
23 {
24   DATUM *localDatum = OSGetFreeNodeFIFO(fifoQueue);
25   if (localDatum != NULL) {
26     // Produce localDatum, for example:
27     localDatum->myData = 3;
28     localDatum->array[0] = 'a';
29     OSEnqueueFIFO(fifoQueue,1);
30     OSScheduleSuspendedTask(WakeUpReader);
31   }
32   OSEndTask();
33 } /* end of Writer */
34 void Reader(void *fifoQueue)
35 {
36   DATUM *localDatum;
37   UINT16 s;
38   while (true) {
39     localDatum = OSDequeueFIFO(fifoQueue,&s);
40     if (localDatum == NULL)
41       break;
42     /* Consume field myData with value 3 */
43     /* Consume array in localDatum with size s = 1 */
44      OSReleaseNodeFIFO(fifoQueue,localDatum);
45   }
46   OSSuspendSynchronousTask();
47 } /* end of Reader */
```

Figure 3.5: Single periodic writer and single event-driven reader.

The writer wakes up the reader each time it inserts an item into the queue. When the reader executes, it processes as many buffer items it can in a loop. When dequeueing a buffer on line 39, the reader obtains the number of characters that the writer inserted into array `string` by the result parameter of function `OSDequeueFIFO`.

Alternative FIFO Queue Design

The FIFO queue mechanism in *ZottaOS* is designed for a general context having any number of enqueuers and dequeuers running at any priority. Although robust, its implementation is complex and may even be overkill for a specific application. In the following we give a simple implementation of a FIFO queue that is suitable when there is a single enqueuer task transferring data to a single dequeuer task. The resulting implementation may come in handy in some embedded designs. These functions are given in Figure 3.6.

```
1  #define QUEUE_SIZE  20
2  DATA FifoQueue[QUEUE_SIZE];
3  UINT8 Head = 0, Tail = 0;
4  BOOL EnqueueData(DATA data)
5  { // Queue is full when (1) Head + 1 = Tail or when
6    //    (2) Head + 1 == QUEUE_SIZE and Tail = 0
7    if (Head + 1 != Tail && (Head + 1 != QUEUE_SIZE || Tail != 0)) {
8       FifoQueue[Head] = data;
9       if (Head == QUEUE_SIZE - 1)
10         Head = 0;
11      else
12         Head += 1;
13      return TRUE;
14   }
15   return FALSE;
16 } /* end of EnqueueData */
17 BOOL DequeueData(DATA *data)
18 { // Queue is empty if Tail = Head
19   if (Tail != Head) {
20      *data = FifoQueue[Tail];
21      if (Tail == QUEUE_SIZE - 1)
22         Tail = 0;
23      else
24         Tail += 1;
25      return TRUE;
26   }
27   return FALSE;
28 } /* end of DequeueData */
```

Figure 3.6: Simple FIFO communication between a pair of tasks.

Although this code requires a smaller memory footprint compared to the FIFO mechanism provided by *ZottaOS*, this code is very fragile. For example, if the lines incrementing `Head` (lines 9 through 12) were replaced by

```
9     Head += 1;
10    if (Head == QUEUE_SIZE)
11      Head = 0;
```

the functions would cease to work properly. Although the enqueuer and dequeuer never access the same entry in the array representing the queue, the dequeuer may remove the same datum more than once. Say that entry `QUEUE_SIZE-1` has been filled by the enqueuer and that `Head` is incremented but before testing the condition that `Head` is equal to `QUEUE_SIZE` and adjusting the index, the enqueuer is preempted. Because of line 19 the queue is never seen to be empty since `Tail` can only assume values smaller than `QUEUE_SIZE`, and `Tail` can never be equal to `Head`. If the dequeuer uses a loop to empty a bounded queue, it would loop until the enqueuer has the chance to update its index.

Because the kernel makes use of the same FIFO queues implementation as the provided API to internally trigger event-driven tasks, implementing additional enqueueing and dequeueing functions tailored specifically for 2 tasks brings no real added value in terms of code usage. This is why we strongly recommend using the API supplied with *ZottaOS* when memory is an issue.

Asynchronous Reader Writer Paradigm

A common paradigm in embedded systems involves sensor tasks that read data, perform some preprocessing and distribute the information to various control tasks, which in turn perform computations and set actuators based on this information. In this paradigm, it is important that the control tasks receive and process the *most recently* read data from the sensor tasks.

The simplest method to implement this paradigm is through global shared variables. Albeit its seemingly very low overhead, this method has obvious flaws making it impractical. The method indeed requires an additional mechanism disallowing task preemption while copying data into and out of the shared variables. This can easily be accomplished by disabling and enabling CPU interrupts respectively at the beginning and end of each data access. However, when the shared datum is consequent, interrupt disabling can cause delays that jeopardize the task deadlines. Furthermore, if the shared datum cannot be filled or emptied at once, the sensor and controller tasks require their own local copies of the shared variables and they can only access the shared variables when these are completely filled or emptied. But before explaining how *ZottaOS* can easily solve this kind of problem, it is instructive to view some techniques that can be applied to specific situations.

Sharing Global Variables Between Two Tasks

Although communicating through global variables is a poor software practice, it is nevertheless commonly used. We can of course argue that because it is not an advisable practice that should be banned, it serves no purpose to explain some techniques that can be applied. We feel that an open-minded approach is called for.

Recall from the monitoring example introduced at the beginning of this chapter, an interrupt service routine (ISR) fills 2 global variables that are then later read by a task to check whether they are identical. The flaw of our introductory example was that the monitoring task could infer that the values do not match simply because the task was preempted by the ISR between both reads. Is it possible to atomically read both values at the same time? It depends. If the processor registers are large enough to hold both values then an assignment becomes a single assembler instruction. If both measured values are byte quantities, these can fit into a 16-bit unsigned integer on 16- and 32-bit processors. This method can be used to safely elude the data coherent problem at the expense of some packing and unpacking code. Figure 3.7 illustrates this technique.

```
1  static UINT16 MeasuredValues;
2  void ReadValues(void)
3  { // Part of the interrupt service routine or called by it
4    INT8 val0, val1; // Assume both measures are signed 8-bit quantities
5    val0 = read from some ADC register
6    val1 = read from some other ADC register
7    MeasuredValues = val0 << 8 | val1;  // Pack data
8  } /* end of ReadValues */
9  void MonitorMeasures(void *arg)
10 {
11   INT8 val0, val1;
12   UINT16 values = MeasuredValues;     // Get both measures
13   val0 = values >> 8 & 0xFF;          // Unpack data
14   val1 = values & 0xFF;
15   if (val0 != val1) {
16      // Set off an alarm
17   }
18   OSEndTask();
19 } /* end of MonitorMeasures */
```

Figure 3.7: Packing data to allow an indivisible read or write.

Packing and unpacking data is a general technique that should not be overlooked as it applies to any number of tasks and it is independent of the relative priorities of the involved tasks.

Another Example

Consider an application requiring the number of days, hours, minutes and seconds since it has started. Assuming that the application can run longer than a year and no more than 180 years, the number of days can fit into a 16-bit quantity, while the remaining time units can each readily fit into 8 bits. The common way to increment the time is to first increment the variable representing the number of seconds and to carry cascading overflows onto the variables of higher time units. The code below illustrates this process.

```
1  void AddSecond(void)
2  {
3    if ((Seconds += 1) > 59) {
4       Seconds = 0;
5       if ((Minutes += 1) > 59) {
6          Minutes = 0;
7          if ((Hours += 1) > 23) {
8             Hours = 0;
9             Days += 1;
10         }
11      }
12   }
13 } /* end of AddSecond */
```

How can the 4 time units be read coherently? The answer depends on the period of the task reading the time. If this task is guaranteed to execute at least once every minute it cannot be preempted by higher priority tasks by no more than a minute, and therefore the task can miss at most 59 increments. So if 2 consecutive reads of the second units are identical, the task can assert that the multiple readings of the time are coherent. The code snippet below illustrates this reading operation.

```
1   UINT16 myDays;
2   UINT8 myHours, myMinutes, mySeconds;
3   do {
4      mySeconds = Seconds;  // read from global
5      myMinutes = Minutes;  // read from global
6      myHours = Hours;      // read from global
7      myDays = Days;        // read from global
8   } while (mySeconds != Seconds);
9   // Local time values are coherent at this point.
```

Note that there can be any number of reader tasks but this code fragment only works if `AddSecond` is called from a task having higher priority than those reading the date.

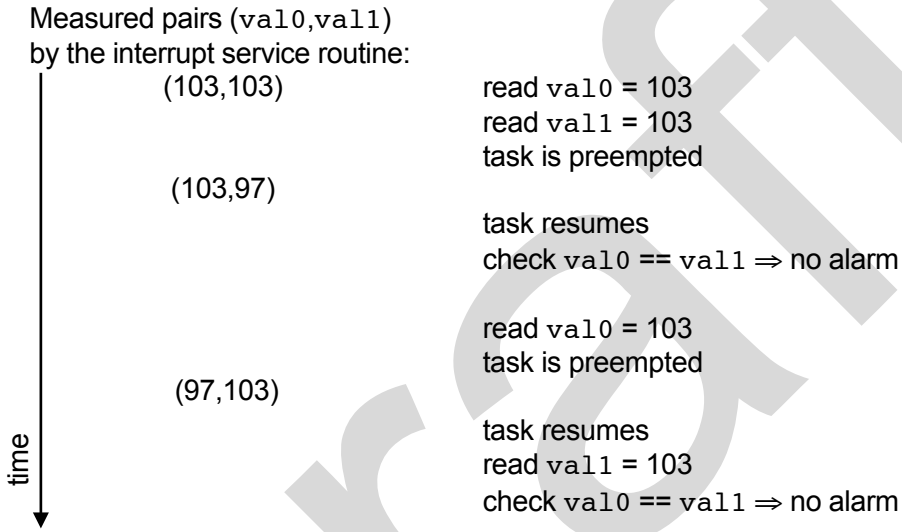
Volatile Keyword

The above code fragment is correct only if the volatile qualifier precedes the data type of the global variable containing the seconds. Optimizing compilers assume that a value stays in memory unless the program explicitly changes it. When the compiler reaches the test on line 8, the optimizer observes that it already read the value of `seconds` into a register at line 4 and instead of re-reading the value from memory, the optimizer deduces that this value cannot change and the re-looping condition of the loop is always false. The optimized code simply becomes lines 4 through 7 and the loop is pushed out of existence. The compiler therefore produces code defeating the purpose of the original loop.

To avoid this, ANSI-C introduced the **volatile qualifier** to warn the compiler that the qualified variable may be changed anytime either within the program or by factors outside the program if the variable is a memory mapped hardware register. With the volatile keyword, all variable references must be accessed regardless of whether they have already been read: the compiler is not allowed to optimize the reads and writes of these variables out of existence.

Going back to the time example, what if the reader task had a period of a couple of minutes? The previous code fragment becomes incorrect. Assume that on day 2, the time is 10:59:23. After reading `mySeconds` = 23 and `myMinutes` = 59, the task can be preempted. Suppose that the time then becomes 11:00:23 when the task resumes its execution. In this case it continues reading the number of hours and days, and then exits the loop since the seconds match those that it previously read before its preemption. The task wrongfully deduces that the current time is 11:59:23. An obvious fix is to pack the second and minute units in a single 16-bit word if the microcontroller has 16-bit registers. But what if an 8-bit processor is used? Even on a 16-bit architecture, the above method fails when the task period exceeds an hour.

It is interesting to note that this technique alone cannot be applied to the monitoring example introduced at the beginning of this chapter. This is because nothing binds the dependences between the 2 measured values. For example, the values can have a transient sequence of (103,103), (103,97), (97,103), and (103,103), which may go unnoticed with this method if the monitoring task is executed at the same rate as the sampling rate:



Data Versioning

Rather than doubling the rate of monitoring task (and doubling the overhead related to the scheduling of the task), a version counter can be introduced.

```
1  static int MeasuredValue[2];
2  volatile UINT8 MeasuredEvents; // Version counter with any initial value
3  void ReadValues(void)
4  { // Part of the interrupt service routine or called by it
5    MeasuredValue[0] = read from some ADC register
6    MeasuredValue[1] = read from some other ADC register
7    MeasuredEvents += 1;  // Set a new version number
8  } /* end of ReadValues */
9  void MonitorMeasures(void *arg)
10 {
11   int val0, val1;
12   UINT8 eventNumber;
13   do {
14      eventNumber = MeasuredEvents; // read from global
15      val0 = MeasuredValue[0];      // read from global
16      val1 = MeasuredValue[1];      // read from global
17   } while (eventNumber!= MeasuredEvents);
18   if (val0 != val1) {
19      // Set off an alarm
20   }
21   OSEndTask();
22 } /* end of MonitorMeasures */
```

Figure 3.8: Binding unrelated data with a version number to allow a coherent read.

It is interesting to note that the version counter `eventNumber` also provides a mean of determining whether the reader task is accessing an already processed datum. All that is needed is to retain its value for the next task instance through the static qualifier provided in C.

Another Example

Say that you wish to implement a function that returns the current time provided by a 16-bit hardware timer. An incorrect implement may look as follows.

```
1  UINT16 TimerHigh = 0;
2  void TimerOverflowISR(void) {  // Timer service routine
3    // reenable interrupts
4    TimerHigh += 1;
5  }
6  UNIT32 GetCurrentTime(void) {  // Function called by application tasks
7    UINT16 low, high;
8    low = timer counter value;
9    high = TimerHigh;
10   return high << 16 + low;
11 }
```

The interrupt service routine (ISR) is invoked whenever the timer overflows and so global variable `TimerHigh` counts the number of times the hardware counter reached 65536. Function `GetCurrentTime` simply returns a value that has the number of timer overflows in its upper 16 bits and the hardware counter in its lower 16 bits. The obvious flaw occurs when a task is interrupted between lines 8 and 9. If we suppose that the timer counter is 0xFFFF, but before reading `TimerHigh` an overflow triggers an interrupt that increments `TimerHigh`. When the task resumes, it blithely concatenates the new `TimerHigh` value with the previously read timer value of 0xFFFF.

One solution would be to disable the interrupts around the reads on lines 8 and 9. This prevents the ISR from gaining control and incrementing `TimerHigh` before the task can read it. This solution is also flawed. Suppose that the timer counter is again 0xFFFF and that `TimerHigh` is 0 immediately after disabling the interrupts. But before reading the timer counter on line 8, a timer tick occurs setting the counter to 0 and creating a pending overflow interrupt. In this case `GetCurrentTime` returns 0 instead of 0x10000 or 0xFFFF, again a huge error.

A correct solution would be to repeatedly read the timer counter until 2 successive reads are identical. The modified `GetCurrentTime` below illustrates.

```
6  UNIT32 GetCurrentTime(void) {
7    UINT16 low, high;
8    do {
9      low = timer counter value;
10     high = TimerHigh;
11   } while (low != timer counter value);
12   return high << 16 + low;
13 }
```

Double Buffering

Double buffering is another technique that can be used when time-correlated data need to be transferred between tasks executing at different but fixed priority levels or when a full set of data is needed by one task but can only be supplied slowly by another task. The basic idea behind double buffering is to maintain 2 sets of data and to keep track of which set is currently being examined.

Going back to the monitoring example illustrating a writer having a higher priority than the reader, we can use 2 sets of measured data. Each task keeps an index corresponding to the entry or set that the task is actualizing (variable `WrtIdx`) or reading from (variable `RdIdx`). The code in Figure 3.9 illustrates.

The code in Figure 3.9 assumes that the task executing `FillValues` may preempt the one calling `MonitorMeasures` but not the converse. Why does this code work? First observe that

```
1  volatile UINT8 WrtIdx = 0; // Most recently available index
2  volatile UINT8 RdIdx = 1;  // Currently accessing index
3  UINT16 MeasuredValues[2][2];
4  void FillValues(void)
5  { // Called by the writer task or ISR
6    if (RdIdx == WrtIdx)
7       WrtIdx = !WrtIdx;     // Switch between alternate sets
8    MeasuredValues[WrtIdx][0] = load from ADC register
9    MeasuredValues[WrtIdx][1] = load from another ADC register
10 } /* end of FillValues */
11 void MonitorMeasures(void)
12 { // Called by a lower priority reader task
13   RdIdx = WrtIdx;          // Post which index is in use
14   if (MeasuredValues[RdIdx][0] != MeasuredValues[RdIdx][1]) {
15      // Set off alarm
16   }
17 } /* end of MonitorMeasures */
```

Figure 3.9: Double buffering with writer task having higher priority than reader task.

function `FillValues` always writes into the set that is not currently being accessed by `MonitorMeasures`. Because `FillValues` always fills a complete set, if `MonitorMeasures` is preempted before line 13, `MonitorMeasures` sees the last filled set; but if `MonitorMeasures` is preempted after line 13, `FillValues` switches sets while `MonitorMeasures` is accessing the previously filled set.

When the reader task has a higher priority compared to the writer task, the basic idea is to let the reader access the most recently filled set even if it has already read it, and to fill another set. The filled set is posted only when the whole set has been filled. Figure 3.10 shows the modified code.

```
1  UINT8 Last = 0;     // Most recently available index
2  UINT16 MeasuredValues[2][2];
3  void FillValues(void)
4  { // Called by the writer task
5    UINT8 next = !Last;
6    MeasuredValues[next][0] = load from ADC register
7    MeasuredValues[next][1] = load from another ADC register
8    Last = next;      // Post most recently filled index
9  } /* end of FillValues */
10 void MonitorMeasures(void)
11 { // Called by a higher priority reader task or an ISR
12   if (MeasuredValues[Last][0] != MeasuredValues[Last][1]) {
13      // Set off alarm
14   }
15 } /* end of MonitorMeasures */
```

Figure 3.10: Double buffering with reader task having higher priority than writer task.

When the relative execution order of the tasks is unknown, the schemes presented so far no longer apply. Two sets of the shared data no longer suffice, and in all, at least 3 sets must be allocated.

Asynchronous Inter-Task Communication API

The API provided by *ZottaOS* to solve this type of communication meets the following requirements:

The Model

1. Data transfer between a writer task and a reader task is carried out by means of a buffer that is composed of slots. The writer inserts data into an available buffer-slot and the reader reads the data from the last written buffer-slot.

2. A writer can overwrite the contents of the slots if the writer is faster than the reader. Some written datum may thus not reach the reader.
3. The reader can repeatedly read the same datum if this task is faster than the writer. This characteristic is avoidable by introducing a state associated with a slot to indicate whether or not the reader has already read its content.
4. The reader must read the data in the order that they were written in the buffer. However, as soon as a datum is read, older data than the one that was read must never be read. Therefore some written data may be lost.
5. Read and write accesses to the shared buffer are asynchronous. In other words, there is no coordination between the tasks to access the buffer.

In what follows, we describe a set of algorithms for this kind of communication. The resulting algorithms also meet the following 3 constraints:

1. The size of the data transferred from the writer to the reader is assumed to be arbitrary and sufficiently large to exclude any particular case where read and write operations become atomic instructions.
2. The algorithms do not make any assumption regarding the relative speed of the tasks nor the time needed to carry out the reading or writing operations.
3. The most recent and complete version of the data written by the writer must always be accessible by a reader.

The second constraint implies that the communication is suitable even between interrupt service routines having no notion of an interrupt priority level, such as for the MSP430 microcontroller. In a general setting and without considering any particular scheduling pattern, it is possible that the reader finishes reading and processing the shared datum and then initiates the reading of a second version of the datum, while the writer is filling up the same buffer. It is thus possible to have several executions of the reader overlapping an execution of the writer. In a similar way, several invocations of the writer may overlap the same reader. The last constraint distinguishes this communication mechanism from the FIFO schemes described previously.

A priori, 3 slots per buffer should be sufficient to comply with our constraints. In what follows we outline a first algorithm then show why it is not correct before proposing correct solutions. The reader may wish to skip this section and to go directly to page 38 where we explain how to use the functions provided in the API. The first algorithm is based on the following (correct) insights:

1. There are two available slots for the writer at all times. When the reader accesses the last version of the datum held in one of the slots, the writer always finds an available slot and when this one is completely filled, the task can pursue with another slot. For example, if the slots are numbered from 1 through 3, and the reader is currently accessing slot 1, the writer alternates between slots 2 and 3.
2. Two control variables shared between the tasks are necessary. One indicates the slot containing the most recently written datum whereas the other indicates the slot that is currently being accessed by the reader.
3. The writer must be able to determine an available slot that does not agree to the most recently completed slot nor to the one used by the reader.

Figure 3.11 shows a possible implementation of the above ideas. The datum transferred from the writer to the reader can be contained in one of the 3 slots. Variable `WrtIdx` indicates the slot holding the most recent and ready to be read copy of the datum; and variable `RdIdx` marks the slot currently being used by the reader. The writer uses array `next` to quickly determine which slot to use. Notice that each entry (1st dimension) refers to 2 indices and does not contain the entry index of the array.

When the writer starts its code, it first chooses a slot that is neither equal to `WrtIdx` nor to `RdIdx` before copying `localData` into the slot. Once the copy is done, global variable `WrtIdx` is updated so that the reader can read it. The reader, before accessing a slot, copies the writer's slot index (`WrtIdx`) into its own (`RdIdx`) to indicate that it is currently using this entry. This as-

```
1  TYPE Buffer[3];                // Shared data-buffer
2  volatile int RdIdx = 0;        // Last read slot by the reader task
3  volatile int WrtIdx = 0;       // Last filled slot by the writer task
4  void Writer(void *arg)
5  {
6    const int next[3][3] = {{1,2,1},{2,2,0},{1,0,0}};
7    int slot;         // Some buffer slot
8    TYPE localDatum; // To create data to insert into the slots
9    /* Produce localDatum */
10   slot = next[RdIdx][WrtIdx]; // Get next slot
11   Buffer[slot] = localDatum;  // Copy the data
12   WrtIdx = slot;              // Indicate last written slot
13   OSEndTask();
14 } /* end of Writer */
15 void Reader(void *arg)
16 {
17   if (WrtIdx != RdIdx) {      // Already read?
18      RdIdx = WrtIdx;          // Indicate current reading slot
19      /* Process the contents of Buffer[RdIdx] */
20   }
21   OSEndTask();
22 } /* end of Reader */
```

Figure 3.11: Asynchronous communication protocol with 3 slots.

signment is not atomic because the reader must first read `WrtIdx` from memory into a register and then copy the contents of that register to the memory location where `RdIdx` is stored. If the reader is preempted after having read `WrtIdx` but before assigning `RdIdx`, it is not impossible that the writer chooses the same slot as the reader. Figure 3.12 illustrates a sample execution path where reader and writer tasks access the same slot.

In the portrayed scenario, the writer posts its first version of the datum and then relinquishes the processor to the reader. This task begins its execution but the writer preempts it before it can update `RdIdx`. The writer regains control of the processor and successively produces 2 new instances of the datum alternating between slots 1 and 2, which it has at its disposal. Notice that while writing into these slots, the writer assumes that the reader can only access slot 0. While filling slot 1, the writer is preempted and the processor is given to the reader so that it can resume its execution. The reader then accesses the slot that is currently being filled by the writer. Although the slot is not corrupted, the reader sees it as such since it sees that the datum starts with the latest instance but ends some other older instance of the datum. If this scenario occurs, the reader obtains incorrect data.

initially `WrtIdx = RdIdx = 0`	
Writer: `slot ← next[0][0] = 1`	
Writer: non atomic write into `Buffer[1]`	
Writer: `WrtIdx ← 1`	
	Reader: reads `WrtIdx = 1`
preemption of Read	
Writer: `slot ← next[0][1] = 2`	
Writer: non atomic write into `Buffer[2]`	
Writer: `WrtIdx ← 2`	
Writer: `slot ← next[0][2] = 1`	
Writer: non atomic write into `Buffer[1]`	
preemption of Writer	
	Reader: assignment of `RdIdx ← 1`
	Reader: reads corrupted `Buffer[1]`

Figure 3.12: Corrupted reading of shared data.

Although this execution has very little chance to occur, it nevertheless shows that the algorithm of Figure 3.11 is not robust. One can also argue, by nature of the priority scheduling implemented in *ZottaOS*, that if the reader first preempts the writer, the reader cannot be later preempted by the writer since the relative priorities of these 2 tasks do not change. This argumentation is however not valid if we consider an interrupt service routine (ISR). In this case, the ISR writer may have several distinct transfers between the peripheral it drives and a buffer slot, and the ISR relinquishes the processor between these transfers. The resulting effect then becomes that of the erroneous scenario we have depicted. Figure 3.13 illustrates the transformation of the writer task into an ISR.

```
1  void ISRWriter(void)
2  {
3    const int next[3][3] = {{1,2,1},{2,2,0},{1,0,0}};
4    static int slot;          // Current buffer slot
5    static bool done = true; // True if previous slot is complete
6    static int i;             // Current slot insertion
7    if (done) {
8       slot = next[RdIdx][WrtIdx]; // Get a slot
9       i = 0;           // Start filling from the beginning
10      done = false;    // Wait until the slot is full
11   }
12   Buffer[slot][i++] = IO_Port_Register; // Add part to the slot
13   if (done = (i == FULL_SLOT))     // Is slot full?
14      WrtIdx = slot;                // Indicate it to the reader
15 } /* end of ISRWriter */
```

Figure 3.13: Multiple instantiation of writer task to complete datum.

There are several published correct algorithms that are immune to temporal subtleties such as the one we have illustrated previously. The first possible solution is to increase the number of slots per shared data. We then obtain an algorithm requiring 4 slots [SIM90]. Another solution consists in re-examining the algorithm of Figure 3.10 to eliminate the temporal dependencies between reader and writer [CHE97]. These 2 solutions are complementary. The algorithm with 4 slots does not use any particular instruction other than load and stores, whereas the corrected solution with 3 slots relies on the existence of a *CompareAndSwap* instruction. In terms of effectiveness, the algorithm with 3 slots is slightly slower, but requires less memory.

In the following, we explain both algorithms starting with the 3-slot scheme. This algorithm is based on the following insights:

1. At all times the reader is in a known state corresponding to one of its processing steps. These states are:
   a. deciding which slot to use,
   b. consuming (i.e., reading) the contents of the slot,
   c. neither of the above states (i.e., doing some other work).
2. In addition to indicating the slot that the reader is currently accessing or the last slot that it read, control variable `RdIdx` could also indicate that the reader is deciding which slot to read.
3. The writer should be able to choose a slot that does not conflict with the reader if the reader indicates which slot it is about to choose.
4. If the reader is preempted or delayed when updating `RdIdx`, the writer can assist the reader in selecting the slot that it should use.
5. Reader and writer can cooperate to update `RdIdx`.

The algorithm with 3 slots resulting from these 5 insights is given in Figure 3.14 [CHE97].

Control variable `RdIdx` takes values 0 through 3; value 3 does not indicate a valid slot but it is used as a state indicator signaling that the reader is choosing the next slot to read.

```
1  TYPE Slot[3];              // A shard data with its 3 slots
2  volatile int RdIdx = 3;    // Last slot read by the reader
3  volatile int WrtIdx = 0;   // Last slot filled by the writer
4  void Writer(void *arg)
5  {
6    const int next[4][3] = {{1,2,1},{2,2,0},{1,0,0},{1,2,0}};
7    int index;                    // A slot index
8    TYPE localDatum;              // To produce some information
9    /* Produce something in localDatum */
10   index = next[RdIdx][WrtIdx]; // Get the next slot
11   Slot[index] = localDatum;     // Copy the information
12   WrtIdx = index;               // Post last filled slot index
13   CAS(&RdIdx,3,index);          // Try to influence reader task
14   OSEndTask();
15 } /* end of Writer */
16 void Reader(void *arg)
17 {
18   RdIdx = 3;                // Post that reader is choosing its slot
19   CAS(&RdIdx,3,WrtIdx);     // Reserve the slot for reading
20   /* Consume contents of Slot[RdIdx] */
21   OSEndTask();
22 } /* end of Reader */
```

Figure 3.14: 3-slot algorithm.

When the writer has a complete datum to hand out to the reader, the task starts by determining a slot that it can fill (line 10). After completing its copying, the task announces which slot it filled by the means of control variable `WrtIdx`, and then tries to influence the reader to use this slot if the reader is currently choosing its slot.

The compare-and-swap (CAS) instruction on lines 13 and 19 is an atomic instruction that can be found on some modern processors. Its purpose is to change the content of a memory location if and only if that location contains a specific value. Hence, `CAS(&RdIdx,3,WrtIdx)` modifies `RdIdx` to `WrtIdx` only if it currently contains the value 3. The important point of this instruction is that it is indivisible. We will have more to say about this instruction in the next section.

The reader starts by indicating that it is in the course of choosing its slot by setting control variable `RdIdx` to 3, it then copies the index of the last slot filled by the writer. An interesting characteristic of the algorithm is how the reader obtains the appropriate slot. Once the slot is found, it does not change, but the 2 tasks contribute to provide the most recent slot.

To obtain an algorithm that does not require any particular atomic instruction, it is necessary to use 4 slots between reader and writer tasks. The resulting algorithm is given in [SIM90] and it is reproduced in Figure 3.15. Several researchers [HEN02, RUS02] have formally checked this algorithm.

The basic idea behind the algorithm is to organize the 4 slots into 2 pairs of 2 slots, with a first pair for the writer and a second for the reader. The writer always chooses a pair opposite to the reader and alternates its writings between the 2 slots of the pair. The selected slot is always the oldest of the 2 slots belonging to the pair. The reader always chooses the last slot filled by the writer. In other words, the reader obtains the last copy of a completely written data by the writer. The algorithm uses 3 control variables to select the suitable slots:

1. `ReaderPair` indicates the last pair of slots currently being accessed by the reader;
2. `WriterPair` indicates the last pair reached by the task writer;
3. `Indices` is a table containing 2 Booleans making it possible to alternate between the slots of a pair.

All these variables are initialized to 0, and it is possible to check whether a slot is completely filled or already read with the following condition:

```
1  TYPE Slot[2][2]       // Communication buffer
2  BOOL ReaderPair;      // Last index read by Reader task
3  BOOL WriterPair;      // Last index filled by Writer task
4  BOOL Indices[2];      // Slot pair to fill or already filled
5  void Writer(void *arg)
6  {
7    TYPE localDatum;               // To produce information
8    BOOL pair, index;
9    /* Produce something in localDatum */
10   pair = !ReaderPair;            // Choose a pair
11   index = !Indices[pair];        // Alternate between used slots
12   Slot[pair][index] = localDatum; // Copy the data
13   Indices[pair] = index;         // Indicate last filled slot
14   WriterPair = pair;             // Indicate last used pair
15   OSEndTask();
16 } /* end of Writer */
17 void Reader(void *arg)
18 {
19   BOOL index;
20   ReaderPair = WriterPair;       // Choose the pair and signal it
21   index = Indices[ReaderPair];   // Choose last filled slot
22   /* Consume the contents of Slot[ReaderPair][index] */
23   OSEndTask();
24 } /* end of Reader */
```

Figure 3.15: Simpson's algorithm with 4 slots.

`ReaderPair` = `WriterPair` *and* `Indices[ReaderPair]` = *last read slot index*.

The writer works on a pair and alternates between the slots of the pair. Variable `WriterPair` indicates the last pair used by the task and `Indices[WriterPair]` indicates the slot containing the most recently filled copy of the `WriterPair` pair. The reader task uses variable `ReaderPair` to indicate to the writer the pair that it is currently reading. Both tasks access different slots and thus avoid reading and writing erroneous data.

How to Use the API

The API provided by *ZottaOS* extends upon the 3- and 4-slotted buffer schemes and incorporates a hidden state indicator to prevent returning uninitialized data to the reader and also indicate already withdrawn data. The API supplies 4 functions: one to create the slotted buffer, one to insert data into a slot and two to remove the last completely filled data. The slotted buffer is created by calling `OSInitBuffer` along with 3 parameters:

1. The size of the slots, which corresponds to the unit of data transferred from the writer to the reader tasks.
2. The algorithm choice, which can either be constant `OS_BUFFER_TYPE_3_SLOT` or `OS_BUFFER_TYPE_4_SLOT`.
3. An optional event, which can be used to signal an event-driven reader task when a slot becomes available for it.

The buffer along with its control variables are created on the heap and can be passed on to the reader and writer tasks as arguments. To illustrate the use of the API, we give a simple example of an asynchronous reader-writer using a 4-slot buffer to transfer a 32-bit unsigned integer from a writer to a reader task. Both tasks are periodic with the writer running twice as much as the reader.

The 4-slot buffer is created on line 10 of Figure 3.16 with a call to `OSInitBuffer` by specifying respectively the datum size (4 bytes), flag `OS_BUFFER_TYPE_4_SLOT` to indicate a 4-slot buffer and finally a NULL event. If the reader was an event-driven task, the event tied to the task could have been passed as the final parameter of the 4-buffer. In this case, once the datum becomes complete (with function `OSWriteBuffer`), the API would signal the event-driven task associated with the event.

```
1  void Writer(void *arg);
2  void Reader(void *arg);
3  int main(void)
4  {
5    void *readerData;    // Reader's datum from Writer
6    // 1. Inhibit the watchdog timer
7    // 2. Initialize the system clock characteristics
8    // 3. Configure peripherals
9    // 4. Create a buffer to store datum from Writer to Reader
10   readerData = OSInitBuffer(sizeof(UINT32),OS_BUFFER_TYPE_4_SLOT,NULL);
11   OSCreateTask(Writer,0,400,400,readerData);
12   OSCreateTask(Reader,0,800,800,readerData);
13   // 5. Configure P10 for output error signals
14   P10SEL = 0x00;  // Set P10 to GPIO
15   P10DIR = 0xff;  // Set P10 to output
16   P10OUT = 0x00;  // Set P10 initially low
17   return OSStartMultitasking();
18 } /* end of main */
19 void Writer(void *readerData)
20 {
21   static UINT32 i = 0;
22   UINT8 dataSize = sizeof(UINT32);
23   // Transfer i to Reader
24   if (dataSize != OSWriteBuffer(readerData,&i,dataSize))
25      P10OUT ^= 0x01; // Signal an error
26   i += 1;            // Prepare for next Writer instance
27   OSEndTask();
28 } /* end of Writer */
29 void Reader(void *readerData)
30 {
31   UINT32 i;
32   UINT8 dataSize = sizeof(UINT32);
33   // Receive datum from Writer
34   if (dataSize == OSGetCopyBuffer(readerData,OS_READ_ONLY_ONCE,&i)) {
35      // Got a complete 4-byte unsigned integer from Writer in i
36   }
37   else
38      P10OUT ^= 0x02; // Signal an error
39   OSEndTask();
40 } /* end of Reader */
```

Figure 3.16: Simple asynchronous reader and writer between 2 periodic tasks.

To keep our example simple, both writer and reader tasks do nothing meaningful. At each instance, the writer transfers the current value of an integer before incrementing it for its next instance. In order to do so, it invokes `OSWriteBuffer` with the buffer that communicates with the reader. This function takes the address of the datum to insert into a slot and the size of the datum. To prevent the reader from obtaining overlapping datum, it is strongly recommended that the size of the buffer be equal to the datum size, as in the case of this example. If the writer is unable to completely transfer its datum, it toggles pin number 1 of digital I/O port 10. This of course should never happen.

The reader task obtains a copy of the 32-bit integer by calling `OSGetCopyBuffer` on line 34. The parameters of this function respectively indicate the buffer from where the datum is sought, a flag specifying that the datum be read only once, and finally the starting address where the copy should be made. Because the writer runs twice as much as the reader, the reader always obtains the very last value transferred by the writer. However, if the reader ran more frequently than the writer, some of the reader's instances would obtain data that a previous instance already processed; flag `OS_READ_ONLY_ONCE` prevents obtaining copies of the same datum. On the other hand, when multiple copies are permitted or desired, this flag can be changed to `OS_READ_MULTIPLE`.

The API also provides another function that does not copy the datum but instead returns a pointer to the slot. This function is called `OSGetReferenceBuffer` and takes the same pa-

rameters except for the 3rd parameter, which becomes a result. The reader gives the address of a pointer and obtains a reference to the datum. For example:

```
UINT32 *pt;
OSGetReferenceBuffer(ReaderData,OS_READ_ONLY_ONCE,&pt);
// available UINT32 in *pt
```

Both reading functions return the number of bytes that were copies or that are accessible.

Finally, to use a 3-slot buffer instead of the 4-buffer, all one needs to do is to change the flag of function `OSInitBuffer` to `OS_BUFFER_TYPE_3_SLOT`.

Advanced Techniques: Concurrent Data Structures

In the preceding sections we looked at the 2 most useful data structures for concurrency: FIFO queues that allow a flow of data blocks from a set of tasks to another, and global data aggregates shared among tasks. We also examined several common techniques avoiding mutual exclusions and locks protecting the data that is shared. Locks are a powerful mechanism to ensure that multiple tasks can safely access shared data without encountering **race conditions**–conditions in which 2 or more tasks try to simultaneously alter the shared data but the overall result becomes different from a serial execution (one after the other) of the involved tasks.

Race Condition

Designing a data structure for concurrency means that any task can access the data structure concurrently and each task will have a coherent view of the data structure. No data is lost or corrupted and all invariants can be substantiated. Such data structures are said to be **thread-safe**. To protect a data structure from race conditions and broken invariants that can ensue, all that is needed is to make all the code areas that access the data structure mutually exclusive. This is precisely what **locks** were designed for. The lock associated with the data structure is captured prior to entering the code area and released at its exit. The code surrounded by the lock is often called a **critical section**, and because only one task can acquire the lock at a time, the data structure is protected by explicitly preventing any concurrent access to the data structure. Tasks are said to be **serialized**, because they take turns when accessing data guarded by the lock.

Thread Safety

Locks

Critical Section

Serialization

Problems with Locks

When reading about locks, it may be tempting to conclude that they can solve all our shared data problems and that they are instrumental in designing concurrent data structures. This is not so. Whenever a task acquires a lock, every other task that subsequently wants that lock must wait until the lock is released. In terms of real-time systems, this means that if a low-priority task captures a lock and is then preempted by a higher priority task also needing the lock, the only way to guarantee that the higher priority task does not miss its deadline is to temporarily give back the processor to the lower priority task until it releases the lock. This has 3 negative consequences. First the RTOS must provide a form of priority inheritance, where the priority of the task holding the lock is temporarily boosted to that of the waiting higher priority task. In addition to an increase in the number of context switches, the RTOS becomes unduly complex for small microcontrollers.

Second, all task instances under *ZottaOS* share the same run-time stack, a scheme that is only viable if higher priority instances completely finish before resuming lower preempted ones. The eventuality of temporarily transferring the processor to lower priority tasks while higher priority tasks are still active implies that each task requires its own run-time stack. This of course leads to a detrimental waste of already scarce memory, since these stacks need to be dimensioned for their worst-case usage.

Third, locks are restricted to application tasks and cannot be used by interrupt service routines (ISR). In fact all commercial RTOSs that we know of all come along with 2 restrictions regarding ISR code. (1) *An ISR may not call a RTOS function that might block the caller*. Therefore ISRs must not require locks and read form queues or mailboxes that might be empty. All these nice and helpful features provided by the RTOS become useless when implementing ISRs. *ZottaOS*

has no such restriction. In fact the FIFO queue and the asynchronous shared-data buffer APIs are non-blocking. (2) *An ISR may not call a RTOS function that might cause the RTOS to task switch unless the RTOS is aware that an ISR and not a task made the call*. Failure to abide to this rule may cause a transfer of the processor to an application task without finishing the ISR and block at least all lower priority interrupts if not all for a very long time. This rule does not apply to *ZottaOS* when signaling an event-driven task because event queues are again non-blocking and task switching is done via a software interrupt that can only be executed after completing the ISR.

Blocking and Non-Blocking

If we can write data structures that are safe for concurrent access without locks then there is the potential to avoid the problems associated with locks. Algorithms for concurrent data structures can be classified as either **blocking** or **non-blocking**. Blocking algorithms are those in which a task trying to read or modify the data structure locks part or all of the data structure to prevent interference from other tasks or interrupt service routines. On the other hand, non-blocking algorithms, also called **lock-free algorithms**, ensure that the data structure is always accessible to all tasks. Such algorithms guarantee that some active task can finish its operation in a finite number of steps even if there are other halted or delayed tasks currently accessing the shared data structure. By definition, non-blocking implementations have no critical sections in which preemption can occur and cannot cause priority inversion–the problem in which lower priority tasks block higher ones or even ISRs. There is another class of progress guarantee other than non-blocking synchronization. **Wait-free synchronization** [HER91] is a stronger progress guarantee since it advocates designing implementations that guarantee that any task can complete any operation in a finite number of steps, irrespective of the execution speed of other tasks.

Lock-Free

Wait-Free

In this section we look at how atomic operations can be used to build non-blocking data structures. We'll start by looking at the reasons for using them before working through some examples and drawing some general guidelines.

Pros and Cons of Non-Blocking Data Structures

Given the difficulty in writing a correct non-blocking data structure, you need pretty good reasons to implement one. With a well-designed non-blocking data structure, every task can progress regardless what the other tasks are doing. The flip side is that tasks cannot be excluded from accessing the data structure and care must be exerted to ensure that the invariants are upheld. Also, special attention must be paid to the ordering of the operations to avoid undefined behaviors associated with race conditions. Modifications are done with atomic operations, but this alone is not enough: these changes must become visible to other tasks in a correct order. All this means that writing thread-safe data structures without using locks is considerably harder than writing them with locks.

Live-Locks

Since there aren't any locks, deadlocks are simply impossible with non-blocking data structures, though there is the possibility of live-locks instead. A **live-lock** occurs when two tasks each try and change the data structure, but for each task the changes made by the other require the operation to be restarted, so both tasks loop and try again. Live-locks are typically short lived because they depend on the exact scheduling of the tasks, and, as we shall shortly see with the methods that we discuss, they can be bounded.

Atomic Instructions

Implementations of concurrent data structures rely on atomic instructions. An atomic instruction is an indivisible instruction. In the following, we survey the 2 most commonly employed instructions and show their relationships as well as their limitations. This discussion is necessary in order to grasp a full understanding of the concurrent data structures that can be developed for *ZottaOS*.

CAS

Most concurrent data structures found in the literature are based on the popular **CompareAndSwap** atomic instruction. The CompareAndSwap (CAS) instruction takes 3 parameters: a mem-

ory address, an expected value and which should be contained in the memory location referred by the first parameter, and a new value that should be stored into the memory location. The new value is written into the memory location if and only if the location holds the expected value and the returned value is a Boolean indicating whether the substitution took place. Semantically, this instruction is equivalent to the atomic execution of the code given below:

```
BOOL CompareAndSwap(WORD *address, WORD expectedValue, WORD newValue)
{
  if (*address == expectedValue) {
     *address = newValue;
     return true;
  }
  else
     return false;
} /* end of CompareAndSwap */
```

This instruction is generally part of the instruction set of modern processors. However, not all microcontrollers for which *ZottaOS* was designed for actually have this instruction. In fact, most do not have any atomic instruction at all. For instance the MSP430 line of microcontrollers have no atomic instruction, whereas the ARM7 family of processors includes only the TestAndSet instruction, which is insufficient to implement non-blocking concurrent algorithms. Given this state of affairs, one would have to emulate this instruction in order to employ the algorithms found in the literature. The two considerations below condition our implementation of CompareAndSwap:

1. To decrease the overhead due to the call of an atomic function, it is preferable to emulate the instruction as a macro, even if macro expansions result into a bigger code size.
2. The CompareAndSwap instruction is generally used as a termination condition of a loop. For example, to increment a variable shared between several tasks, we typically have the following code fragment:

   ```
   do {
     old = globalVariable;  // Local copy for the comparison
     new = old + 1;         // Modified value
   } while (!CompareAndSwap(&globalVariable,old,new));
   ```

   While the increment does not succeed, the loop is not exited.

With these considerations in mind, we propose the implementation given below.

```
#define CAS(address,oldValue,newValue,failLabel) \
{ \
  DISABLE_INTERRUPT; \
  if (*address == oldValue) { \
     *address = newValue; \
     ENABLE_INTERRUPT; \
  } \
  else { \
     ENABLE_INTERRUPT; \
     goto failLabel; \
  } \
}
```

And the preceding example, which consisted in incrementing a global variable, becomes:

```
failure:
  old = globalVariable;   // Local copy for the comparison
  new = new + 1;          // Modified value
  CAS(&globalVariable,old,new,failure);
```

For Your Information

Emulation of a CompareAndSwap instruction on the MSP430 microcontroller takes approximately 38 machine cycles of which 22 are in a critical section for which interrupts are disabled.

LL and SC

Modern instruction-set architectures, such as ARMv6 and above, DEC Alpha, MIPS II and PowerPC processors, do not provide atomic instructions that read and update a memory loca-

tion in a single step because it is much more complex than a typical RISC operation and difficult to pipeline. Instead these processors provide an alternative mechanism based on two atomic instructions, **load-linked** (LL, also called load-lock or load reserve) and **store-conditional** (SC). The usual semantics of LL and SC assumed by most algorithm designers are given in Figure 3.17. A shared variable X accessed by these instructions can be viewed as a variable that has an associated shared set of task identifiers, $valid_X$, which is initially empty. Instruction LL returns the value stored in the shared location X and marks it as reserved by including the calling task identifier in set $valid_X$. This read operation always succeeds. Instruction SC takes the address of a shared location X and a value, newValue, as parameters. It checks if the calling task's identifier is in $valid_X$, and if so, completely clears $valid_X$ and updates the location before returning success; otherwise the instruction returns failure without modifying location X.

```
WORD LL(WORD *addressOfX) {
  validX = validX ∪ {taskID};
  return *addressOfX;
} /* end of LL */

BOOL SC(WORD *addressOfX, WORD newValue) {
  if (taskID ∈ validX) {
     validX = ∅;
     *addressOfX = newValue;
     return true;
  }
  else
     return false;
} /* end of SC */
```

Figure 3.17: Equivalent atomic statements specifying the theoretical semantics of LL/SC.

Unfortunately, all architectures that support LL/SC instructions do not support the full theoretical semantics that we have described above. These architectures have one or several of the following limitations [MOI97]:

1. There can be neither nesting nor interleaving of LL/SC pairs.
2. A single bit per processor replaces the set of task identifiers $valid_X$.
3. The reservation bit typically may also be associated to a set of memory locations and a normal write to an address close to the one that was read by a LL can clear the bit.

Given the LL/SC instruction pair, atomically incrementing a global variable can readily be written as

```
do {
  old = LL(&globalVariable); // Get and reserve the shared variable
  new = new + 1;             // Modified value
} while (!SC(&globalVariable,new));
```

Retry Loops Operations on non-blocking data structures are implemented in a similar way, through the use of retry loops as exemplified by the preceding increment operation on a global variable. For real-time tasks, the number of times that the retry loop is taken must be bounded. To intuitively explain why such a bound exists, let us call an iteration of the retry loop a successful update if the while loop exits and a failed update when an additional iteration must be done. Thus the increment operation on a global variable consists of a number of failed attempts at the SC instruction followed by one successful update. Consider two tasks T_1 and T_2 that increment the shared global variable. Suppose that T_1 causes T_2 to experience a failed update. This can only happen if T_1 preempts T_2. Thus there is a relationship between failed updates and task priorities. The maximum number of task preemptions in a given time interval can be determined from the timing requirements of the tasks. Using this information, it is possible to determine a bound on the number of failed updates in that interval, and intuitively, a task set that shares a global variable is schedulable if there is enough free processor time to accommodate the failed updates that can occur over any interval. Hence under priority-driven scheduling, lock-free implementations may be bounded and may even become wait-free.

A CAS instruction can also be emulated as

```
BOOL CompareAndSwap(WORD *address, WORD expectedValue, WORD newValue)
{
  while (LL(address) == expectedValue)
    if (SC(address,newValue))
       return true;
  return false;
} /* end of CompareAndSwap */
```

Notice how a while loop is necessary to retain the full semantics of the CAS instruction. If a task modifies the memory location to a given value and then re-modifies it back to its previous value after another task has read the contents of the memory by an LL but before it has a chance to set its new value, this latter task should succeed its SC. Hence, the only possible failure from a CAS is when the LL fails to return the expected value that should have been in the memory location.

For Your Information

The LL/SC function calls corresponding to all native data types are implemented in file *ZottaOS_Atomic.c* and their prototypes are defined in the header file corresponding to the kernel version of *ZottaOS*.

LL/SC on Cortex-M3 and M4

Cortex-M3 and Cortex-M4 based microcontrollers, such as STM32L1, STM32F1, STM32F2 [STM11], and STM32F4 [STM12] families from STMicroelectronics provide instructions `LDREX` and `STREX` (load and store register exclusive) that have the sought semantics implementing functions LL and SC. These microcontrollers also have an instruction, called `CLREX`, which forces the failure of the next `STREX` instruction if its paired `LDREX` occurred before the `CLREX`. `CLREX` is executed by *ZottaOS* whenever there is a context switch.

For Your Information

The kernel implementations of *ZottaOS* are completely done with LL/SC calls. Hence on Cortex-M3 and M4 compliant microcontrollers, interrupts are enabled at all times.

LL/SC on AVR32

The semantics of the LL/SC function pair can also be implemented on Atmel's AVR32 line of microcontrollers [ATM11], which has a dedicated lock bit in the core's status register for this purpose. The `stcond` instruction tests this bit to determine whether its second operand can replace the content of the memory location given by its first operand. An SC operation becomes

```
stcond m, val  ; *m ← val if lock bit is set
sreq result    ; result ← lock bit (0 or 1)
```

An LL instruction simply sets the lock bit before loading the content of a memory location into a register:

```
ssrf 5         ; set lock bit
read memory content
```

All that is needed to complete the semantics is to clear the lock bit whenever there is a context switch. This is readily accomplished with the provided `rete` instruction, which is normally called when returning from an interruption but which can also be called when terminating a task.

LL/SC on MSP430 and CC430

On the MSP430 families of microcontrollers, the LL/SC functions need to be emulated, as was done for the CompareAndSwap instruction. This is accomplished by having a global Boolean (called `_OSLLReserveBit` and defined in file *ZottaOS_Atomic.c* for MSP430) that acts as the lock bit found in the AVR32. The table on Figure 3.18 shows the number of machine cycles needed to execute these functions. Of particular interest is the number of cycles in the critical section where the interrupts are disabled. Only *ZottaOS-Hard PA* uses SC invocations for 32 bit data sizes. Therefore the longest period of time during which interrupts are disabled is 18 machine cycles.

In the following we look at two non-blocking data structures that may come in handy when designing an application: stacks and priority queues. These 2 data structures combined with those provided in *ZottaOS*'s API constitute the bulk of the building blocks of an embedded application.

Function	Data Size [bits]	Machine Cycles	Cycles in Critical Section
LL	8 and 16	9 / 11	0
	32	19 / 22	7
SC	8 and 16	23 / 26	15
	32	26 / 29	18

Figure 3.18: Number of machine cycles required by emulation of LL/SC functions on MSP430 and CC430 microcontrollers. (Numbers *A* / *B* found on the 3rd column respectively give the number of machine cycles for 16-bit (*A*) and 20-bit (*B*) address space.)

Concurrent Stack

On their own, stacks are not common in an application but they are adequate to implement a global pool of free blocks, where tasks may acquire a block, fill it up and later give it back as soon as the block no longer becomes useful so that it can be recycled. The basic behavior of a stack is fairly straightforward: nodes are retrieved in reverse order to which they are added, i.e., last in, first out (LIFO) order. The simplest implementation relies on a linked list of nodes where each node contains a link to its successor and a global pointer, called `top`, identifying the starting point of the list and representing the most recently added node that has not yet been consumed.

Adding a node involves 3 steps:

1. obtain a free node,
2. set its successor pointer to the current `top` node,
3. set the `top` pointer to the newly inserted node.

When several tasks are inserting nodes, there is a race condition between steps 2 and 3. Specifically, a task could be preempted between these steps and after it resumes, it would substitute its node as the new top, discarding the added nodes that could have been inserted by the tasks that preempted it. Before we address this race condition, it is also important to note that once the inserted node becomes the new `top`, other tasks can access it. It is therefore indispensable that the new node is thoroughly consumable before `top` is set to point to it: the node cannot be modified afterwards. To correct for the race condition, `top` must be marked with an LL instruction at step 2 and modified with a matching SC instruction at step 3 to ensure that its has not been modified since it was read. If it has, then steps 2 and 3 have to be repeated. Function `PushInteger` in Figure 3.19 shows the implementation of the concurrent push operation without locks. In this example, we implement a concurrent stack of 32-bit integers having 2 operations: a push and a pop operation. These functions respectively insert an integer onto the stack and return the most recently unconsumed integer. The maximum size of the stack is determined by function `InitStack`, which creates a pool of free nodes. This function should be called prior to calling `OSStartMultitasking` or to the first stack operation, whichever is called first. Once the stack is initialized, tasks can push or retrieve integers. To insert integers, tasks first obtain a free node with `GetFreeNode` on line 27. If one is available at the time of the call, the task prepares its insertion on line 29 and 31. Lines 31 and 32 make up the mentioned retry loop circumventing the race condition.

Now that we have a function to add data to the stack, we need a means of popping them off. At first glance, this may seem simple:

1. read the current value of `top` into a local pointer, say `t`,
2. set `top` to `t→next`,
3. return the data from the retrieved node `t`,
4. recycle the retrieved node `t` into the free pool.

```
1  typedef struct NODE {
2    INT32 value;
3    NODE *next;
4  } NODE;
5  static NODE *top;       // stacked nodes, initialized to NULL
6  static NODE *freeNodes; // stack of available nodes
7  BOOL InitStack(UINT8 nodes) {
8    UINT8 i;
9    freeNodes = (NODE *)OSMalloc(nodes*sizeof(NODE));
10   for (i = 0; i < nodes - 1; i += 1)
11      freeNodes[i].next = &freeNodes[i+1];
12   freeNodes[i].next = NULL;
13 } /* end of InitStack */
14 static NODE *GetFreeNode(void) {
15   NODE *node;
16   while ((node = (NODE *)OSUINTPTR_LL((UINTPTR *)&freeNodes)) != NULL)
17      if (OSUINTPTR_SC((UINTPTR *)&freeNodes,(UINTPTR)node->next))
18         break;
19   return node;
20 } /* end of GetFreeNode */
21 static void ReleaseNode(NODE *node) {
22   do {
23      node->next = (NODE *)OSUINTPTR_LL((UINTPTR *)&freeNodes);
24   } while (!OSUINTPTR_SC((UINTPTR *)&freeNodes,(UINTPTR)node));
25 } /* end of ReleaseNode */
26 BOOL PushInteger(INT32 value) {
27   NODE *node = GetFreeNode();
28   if (node != NULL) {
29      node->value = value;
30      do {
31         node->next = (NODE *)OSUINTPTR_LL((UINTPTR *)&top);
32      } while (!OSUINTPTR_SC((UINTPTR *)&top,(UINTPTR)node));
33      return TRUE;
34   }
35   return FALSE;
36 } /* end of PushInteger */
37 BOOL PopInteger(INT32 *value) {
38   NODE *node;
39   while ((node = (NODE *)OSUINTPTR_LL((UINTPTR *)&top)) != NULL)
40      if (OSUINTPTR_SC((UINTPTR *)&top,(UINTPTR)node->next)) {
41         *value = node->value;
42         ReleaseNode(node);
43         return TRUE;
44      }
45   return FALSE;
46 } /* end of PopInteger */
```

Figure 3.19: Concurrent integer stack.

However in the presence of multiple tasks this is not so simple. If there are two tasks removing items from the stack, then both might read the same value of `top` at step 1. If one task then proceeds all the way through step 4 before the other gets to step 2, the latter task will be accessing a memory block that is in the free pool, replacing the stack by the pool of free data blocks. The latter task will also return the same value as the former task, violating the customary requirement that values may only be popped once from the stack. This race condition can be resolved in the same way as for the push operation: tasks compete to replace the top pointer with its successor node with the now familiar LL/SC retry loop, and because only one task can accomplish this, the successful task takes ownership of the popped node, and may then safely execute steps 3 and 4. Function `PopInteger` implements this idea. Lines 39 and 40 carry out the retry loop, and on success, return the stored value before recycling the popped node by calling `ReleaseNode`.

The only thing held in abeyance and that was skimmed over is the memory management of the stacked blocks. *ZottaOS* was designed for systems with scarce memory. As such, dynamically allocating and freeing memory blocks not only is time consuming but it also creates memory holes that are costly to recuperate so that they may be recycled. This is why embedded systems usually allocate all their memory blocks beforehand. For data structures this also means that it is up to the data structure to manage its own blocks, and what simpler way than to manage these blocks by a stack. As shown on Figure 3.19, `InitStack` allocates the memory blocks dynamically in a single chunk and then creates the initial stack of free blocks by linking these blocks as a list. Function `GetFreeNode` returns the first available block from the pool, and `ReleaseNode` inserts the freed node back into the pool.

Wait-Free Concurrent Priority Queue

A priority queue offers a slightly different challenge to a stack. The linked list of nodes is retained but an insertion may take place anywhere in the list rather than only at its end. Consequently, the synchronization needs are different: we need to ensure that all the changes made by an insertion operation are correctly visible to accesses at the dequeueing end.

Priority queues are useful when data should be processed in a specific order, such as scheduling tasks or processing control messages before data messages in a RF channel. In what follows we show such an example and construct the foundation needed to build a time-driven sub-application where the sub-application operates as a statechart accommodating several tasks that suspend themselves after scheduling their next release. Specifically, we wish to be able to have event-driven tasks that when activated carry out operations related to a state, and once these operations are completed, the task prepares its next instance by specifying a transition delay. Task `TimeDrivenTask` in Figure 3.20 illustrates such a task. When an instance is activated it first determines what it should do by inspecting local variable `state`, executes the operations associated with its current state, and then decides its next transition by updating `state` and invoking `SetTaskDelay` with the event that triggers the task and with a delay representing the time it spends in its current state. The instance finally suspends itself.

In the above, `SetTaskDelay` incorporates an insertion algorithm which creates a block containing an event with its time of occurrence and inserts the block into a linked list sorted in the order in which the events take place. The sub-application also includes an event-driven timer module whose duty is to signal the events when their time falls due.

```
1  void TimeDrivenTask(void *argument) {
2    static UINT8 state = 0;  // Current state when activated
3    INT32 nextDelay;         // Time spent in the state
4    switch (state) {
5       case 0:  // initial (first or even) execution
6          /* Do something useful, e.g. set a LED */
7          nextDelay = 2000; state = 1;
8          break;
9       case 1:  // second or odd exections
10      default:
11         /* Do something useful, e.g. clear a LED */
12         nextDelay = 5000; state = 0;
13   }
14   SetTaskDelay(Event,nextDelay);
15   OSSuspendSynchronousTask();
16 } /* end of TimeDrivenTask */
```

Figure 3.20: Structure of a time-driven task triggered by events.

The implementation of our final sub-application is trickier than it appears for we should also address some of its inherent problems:

- How can we store the event time on a finite number of bits without being troubled with integer overflows and wraparounds?

- How do we program the timer module when a new block is inserted at the head of the list?

Before dealing with these problems it is worthwhile giving an implementation of a simple wait-free priority queue sorted by an integral key and providing 2 functions: a dequeue function which retrieves the smallest key at the time of its invocation, and an insertion function. As we shall shortly see, this implementation is not as easy as it may seem. We begin by overviewing some of the difficulties we are faced with.

As a running example, consider a linked list containing two sorted integers, 30 and 100 along with the queue's sentinel head. To simplify the explanation, let `A` and `B` denote the two nodes that are in the queue. Suppose that a task performs an insertion operation in the queue with a node holding integer 40. Call this node `C`. The new node should be inserted between nodes `A` and `B`. Hence, the task must first set the successor link of `C` to point to `B`, and then, in a second step, modify the successor field of node `A` so that it contains the address of `C`. However, between these two operations, the task can be preempted. Figure 3.21 shows the state of the linked list at this moment.

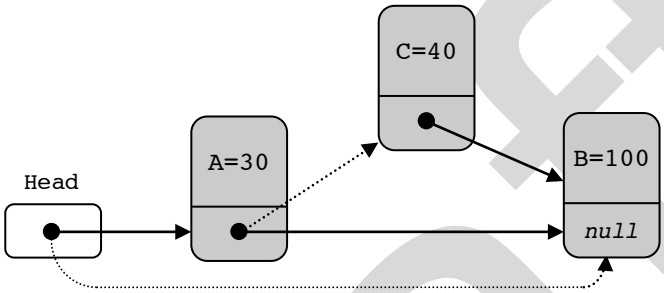


Figure 3.21: Linked list with a concurrent insertion and dequeue. Solid arrows represent links that are established and dotted lines indicate those yet to be done.

Suppose that another task concurrently removes node `A` during the preemption of the inserter. This task establishes a link from the head sentinel to node `B` so as to short-circuit node `A`. Later, when the task, which was inserting node `C`, resumes it still has to complete the link from `A` to `C`. But the inserter task has no way of knowing that node `A` is no longer part of the linked list, and node `C` is lost.

Continuing with this example, suppose that during the preemption of the inserter task, another task removes node `A` and reinserts it but with integer value 90. Node `A` occupies the same position as before. When the inserter task resumes its execution it will wrongly insert node `C` between nodes `A` and `B`. Any valid concurrent sorted linked list algorithm must thus be immune to the described problems. The reader will have noticed that the problems are related to the fact that the concurrent operations (insertions and dequeues) overlap and would not have occurred were they serialized.

Incremental Helping

Our linked list implementation is based on the technique put forward by Anderson et al. [AND97,RAM96], which they call **incremental helping**. Before beginning an operation, a task must first announce its intention by posting information about its operation. However before a task is allowed to do this, it must first help complete any other previously announced operation.

The scheme is based on the following 4 observations:

1. Because of priority-driven scheduling policies in *ZottaOS*, a task instance can only be preempted by a higher priority task, and the preempted instance cannot be given the processor if its preemptor is still active. This last claim is true because an instance cannot dynamically change its priority.
2. When preempted, the information regarding an operation is accessible to all higher priority tasks. Hence, a task can post a local descriptor located on the run-time stack. There is no need to provide global structures that depend upon a priori knowledge of the number of tasks in the application.

3. When a task instance preempts another task, the instance can only encounter a single pending operation posted by another task during its whole execution.
4. When a task relinquishes the processor to another task, it cannot leave behind an unfinished operation.

With the above observations, the number of operations is bounded by those done by the task plus a single pending operation that the task may have to complete on behalf of another lower priority task. The scheme allows wait-free implementations of concurrent data structures.

The general idea of incremental helping is illustrated in Figure 3.22. Assume that the lowest priority task starts a modification operation on the sorted list. It first announces its operation and proceeds with the modification of the list. Before that task can complete its operation, Task 2 preempts it. If Task 2 also needs to modify the list, it must first help Task 1 complete its operation before starting its own. However, as Figure 3.22 illustrates, Task 3 preempts Task 2 before it can complete the operation. Suppose that Task 3 also requires a modification of the shared list. It too must first help the pending operation posted by Task 1 before announcing its own. It then completes both operations–the one requested by Task 1 and its own–before relinquishing the processor to Task 2. Task 2 detects that the pending operation of Task 1 is complete and continues by executing its own operation and finishes it before leaving the processor to Task 1. Task 1 then detects that its operation is complete and returns.

The novelties of our method over all those proposed in the literature are twofold:

1. Tasks announce their operations using only locally declared variables made visible and accessible to other tasks by simply posting an entry-point to the variables via a single global pointer.
2. Our method does not rely on a priori knowledge of the number of tasks that may access the concurrent sorted list.

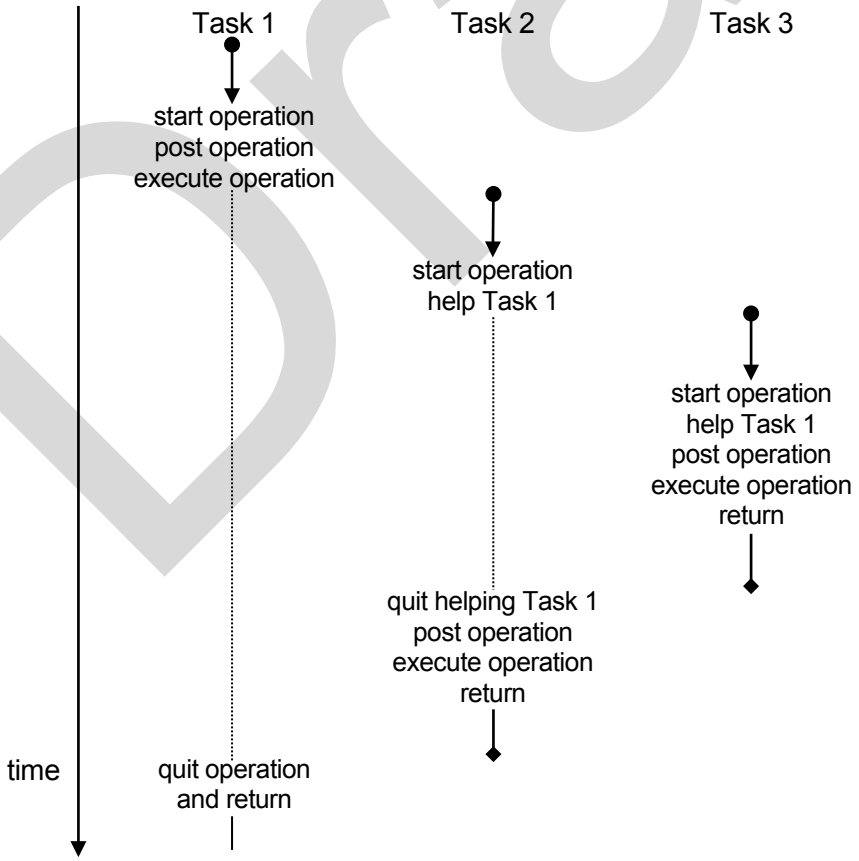


Figure 3.22: Serialization by incremental helping. Dotted end-point arrows denote a task instance preemption and diamond shaped end-points indicate the end of the task instance.

The aim of an insertion algorithm for a sorted linked list is to find two adjacent nodes, *left* and *right*, such that

$$left = head\ sentinel \ \underline{\text{or}}\ TestKey(left,node)$$

and

$$right = null \ \underline{\text{or}}\ TestKey(node,right)$$

where *TestKey* is an ordering function defined as

$$TestKey(a,b) = \begin{cases} true & \text{if } a \text{ is before } b \\ false & \text{otherwise} \end{cases}$$

Figure 3.23 illustrates 3 snapshots of the contents of a posted descriptor at different points in time: while seeking the left and right nodes, when the right node has been established and finally when the new node to insert has been inserted into the sorted list. The different fields of the descriptor are the following. The `left` field holds the last visited node of the list while seeking for the right node and allows tasks joining those that are completing the operation to directly begin with the most recently inspected node. This field is initialized with the head sentinel of the list prior to starting the insertion. Field `insertionNode` denotes the node that is to be inserted. Finally, field `done` indicates the state of the insertion operation and is set to true when the node has been inserted.

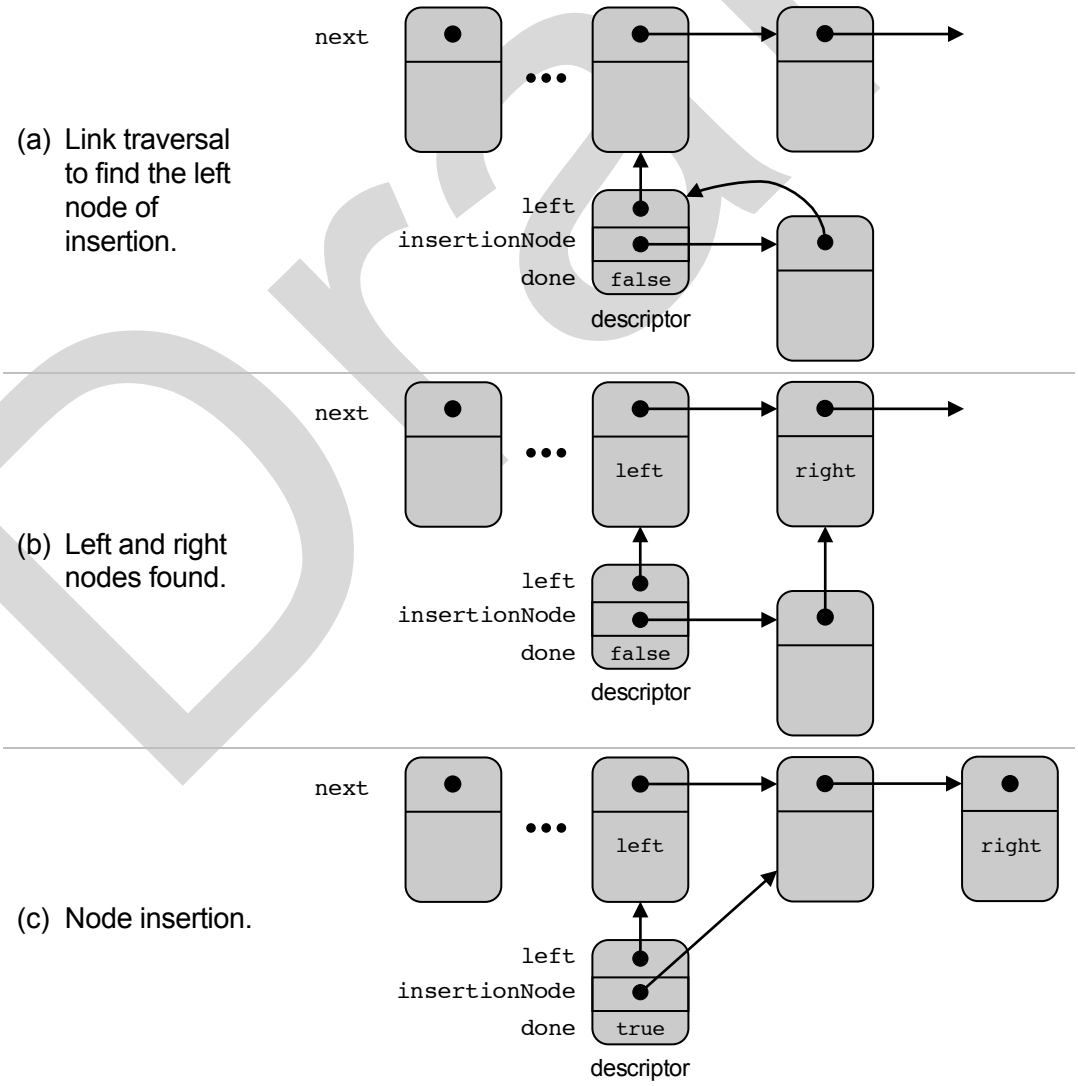


Figure 3.23: Three snapshots of the concurrent insertion algorithm.

To guarantee a correct implementation, the insertion algorithm should provide a means of detecting that an operation is finished and, if a task is preempted in the midst of an insertion operation and then resumed, the actions performed by the task should fail and not modify the list if the operation has been completed by a higher priority task. This can be done in 2 ways. Our insertion algorithm uses both methods. The memory location to modify can be initially marked with a unique sentinel value, and then updated in a retry loop with the atomic instructions LL/SC:

```
while (LL(&memoryLocation) == sentinel)
  if (SC(&memoryLocation,update))
     break;
```

The first task to successfully flip the memory location with the updated value will prevent further modifications of the memory location. This method is used to establish field `next` of the inserted node. Initially this field is assigned to the address of the posted descriptor, and the first task to find the right node establishes the field (see part (b) of Figure 3.23). The second method makes use of a Boolean flag indicating that the operation is already done. This method only works if the first successful task to set the flag updates the memory location with the same value as the tasks it may have preempted in their retry loops. The following code snippet illustrates this method.

```
while (LL(&memoryLocation) == previously read value)
  if (done || SC(&memoryLocation,update))
     break;
done = true;
```

The final operation, making the found left node point to the newly inserted node, uses this method.

Priority Queue Implementation

The complete concurrent implementation is given in Figures 3.24 and 3.25 in a self-explanatory pidgin C notation. For clarity, various type casts are missing but pointer dereferencing follows strict C notation. Following standard practice, all global variables have identifiers beginning with an uppercase letter and variables in lowercase are local to an operation. As well as providing an insertion algorithm, the implementation also includes a dequeueing function, which also relies on incremental helping to retrieve the first node of the linked list. We use a standard programming trick to distinguish between the 2 operations. Each descriptor has a field indicating its operation type (lines 7 and 13), and if a pending operation exists when a task invokes one of the queueing operations, it is an easy manner to extract the pending operation type and to call its associated helper function (lines 29 to 34 and 68 to 73).

As done in the stack implementation of Figure 3.19, we assume that the nodes that are part of the linked list come from a free pool of nodes managed as a stack. Our implementation refers to functions `GetFreeNode` and `ReleaseNode` that have the same definitions as for the stack algorithm except that they operate on nodes that are defined on lines 1 to 4 of Figure 3.24.

The astute reader may have noticed that we have only explained how to insert a node into a non-empty list and have disregarded the fact that the list may be empty. This is because we use a sentinel node so that the queue always appears to have at least a node. But contrary to other linked list implementations, ours doesn't use an explicit node. Going back to Figure 3.23, the `left` field of an insertion descriptor denotes the first node of the list to inspect, and at each step we take `right = left->next`. Now if we initialize `left` to the address of the list, then `right` refers to the list itself. The relevant parts of the insertion with their line numbers and their effects are given below.

```
16 NODE *QueueHead = null;
   …
27 left = (NODE *)&QueueHead;
   …
41 right = left->next;       // right = QueueHead = null
   …
55 left->next = newNode;     // QueueHead = newNode
```

The tricky part is on line 41 where the address contained in `left` is added to an offset of 0 before being referenced and becomes equivalent to `right = *left`. This explains why the successor fields of the nodes in the list must be the first in the structure.

Given these preliminary remarks, we are now in position to overview our implementation. Function `InsertQueue` inserts the integer passed as parameter and posts an insertion operation. The first steps of this function initiate the descriptor so that it corresponds to part (a) of Figure 3.23 (lines 21 to 28). On line 29, the global pointer `PendingQueueOperations` is read and if it is different than *`null`*, the variable designates the descriptor of a pending operation posted by a lower priority task, and the pending operation is completed on lines 32 and 34. The descriptor is posted on line 35 and then processed. Finally, the descriptor is removed on line 37. The actual insertion is done by function `InsertQueueHelper`, which is composed of a loop to traverse the queue. At each iteration, the immediate successor of the last visited node is determined (line 43), and if this node is still to the left of the node to insert, the node becomes the last visited left node by this task (line 61). By saving the left node, a task helping another can continue from the most recently visited node rather than starting from the beginning of the list.

Once the immediate successor of the left node is determined, a succession of tests is done on that node. If the right node corresponds to the inserted node (line 46), then there must have been some other task that successfully inserted the node in the queue (line 55) but was preempted before it had a chance of setting the `done` flag on line 57. In this case, all that needs to be done is to mark the end of the current insertion and exit (lines 47 and 48).

The test on line 50 ascertains whether the right node is to the right of the node to insert. If this is the case, the inserted node should be between the now found left and right nodes. First the successor of the inserted node is established (lines 51 to 53), and the LL/SC operation succeeds if and only if the successor field `next` has never been modified since it was marked with the sentinel at initialization time (line 24).

The final step to insert a node is to establish the link from the left node to its new successor (lines 54 to 56). This step only succeeds if the left and right nodes are still adjacent to each other and if the step has not yet been done.

As mentioned, the dequeueing function is also based on incremental helping, and it requires a descriptor with a single distinctive field called `deletedNode`. The dequeuer task posts its operation and receives, once completed, the node that was at the head of the list when the operation was posted. All tasks that intervene with the dequeuer participate in retrieving the node. Figure 3.25 shows the implementation of the dequeueing function. Likewise to `InsertQueue`, `DeleteQueue` first prepares its descriptor and then completes any pending operation before posting its own operation (line 74). Once `DeleteQueueHelper` finishes the operation, the node that was removed is found in field `deletedNode` (line 77). If there is such a node, the dequeuer waives it to the pool of free nodes (line 81) after retrieving the integer key it contains (line 80).

Function `DeleteQueueHelper` proceeds in two steps. The first is to save the node at the head of the list and the second is to remove it. During the first step, the head of the list is read and stored in local variable `removeNode` (line 86) and if this variable refers to a node, its content is copied into the descriptor (line 88). It is interesting to note that this first step does not rely on any atomic instruction for its correctness. There are 2 cases to consider. If a dequeuer is preempted between lines 86 and 87 or between 87 and 88, the task simply copies the value it read into `removeNode` whose value corresponds to the one already found in field `deletedNode` of the descriptor. This is because the preempting task completing the operation reads the same node value as the preempted task. However if the dequeuer is preempted before line 86, the code's correctness is surprisingly witted. There are again 2 cases to consider. In the first, the pre–empted dequeuer may read a different node than the one found in field `deletedNode`. In this case the dequeuer fails the test on line 87. The second and interesting situation is where the list is currently empty when the operation is posted. In this case the preempted task may read a non-null pointer in its local variable and accordingly update field `deletedNode`. This situation behaves as if the task was preempted immediately before posting its operation and receives the node inserted by a higher priority task.

```
1  typedef struct NODE {    // Blocks that are in the priority queue
2    struct NODE *next;     // Successor node
3    INT32 key;             // Sorting key
4  } NODE;
5  typedef enum {InsertOp,DeleteOp} QUEUEOP;
6  typedef struct {         // Insertion information block
7    QUEUEOP op;            // Equal to InsertOp
8    volatile BOOL done;    // Termination flag
9    NODE *left;            // Current scanning pointer
10   NODE *insertionNode;   // Node to insert
11 } INSERTQUEUE_OP;
12 typedef struct {         // Deletion information block
13   QUEUEOP op;            // Equal to DeleteOp
14   NODE *deletedNode;     // Removed node
15 } DELETEQUEUE_OP;

16 static NODE *QueueHead = null; // First node of the list
17 static INSERTQUEUE_OP *PendingQueueOperations = null;
18 BOOL InsertQueue(INT32 key) {
19   NODE *newNode;
20   INSERTQUEUE_OP des, *pendingOp;
21   if ((newNode = GetFreeNode()) == null) // Defined in Figure 3.19
22      return false;
23   newNode->key = key;
24   newNode->next = (NODE *)&des;
25   des.op = InsertOp;
26   des.done = false;
27   des.left = (NODE *)&QueueHead;
28   des.insertionNode = newNode;
29   pendingOp = PendingQueueOperations;
30   if (pendingOp != null)
31      if (pendingOp->op == InsertOp)
32         InsertQueueHelper(pendingOp);
33      else
34         DeleteQueueHelper((DELETEQUEUE_OP *)pendingOp);
35   PendingQueueOperations = &des;
36   InsertQueueHelper(&des);
37   PendingQueueOperations = null;
38   return true;
39 } /* end of InsertQueue */
40 static void InsertQueueHelper(INSERTQUEUE_OP *des) {
41   NODE *right, *left = des->left;
42   while (true) {           // Loop until done
43      right = left->next;   // Get adjacent node
44      if (des->done)        // Has a higher priority task finished the insertion?
45         break;
46      else if (right == des->insertionNode) { // Is the node already in the queue?
47         des->done = true;                    // Mark that node has been inserted
48         break;
49      }
50      else if (right == null || des->insertionNode->key < right->key) {
51         while (LL(&des->insertionNode->next) == des)
52            if (SC(&des->insertionNode->next,right))
53               break;
54         while (LL(&left->next) == right)
55            if (des->done || SC(&left->next,des->insertionNode))
56               break;
57         des->done = true;   // Mark that node has been inserted
58         break;
59      }
60      else
61         left = des->left = right;
62   }
63 } /* end of InsertQueueHelper */
```

Figure 3.24: Concurrent wait-free priority queue implementation (1st part).

```
64 BOOL DeleteQueue(INT32 *key) {
65   DELETEQUEUE_OP des, *pendingOp;
66   des.op = DeleteOp;
67   des.deletedNode = null;
68   pendingOp = PendingQueueOperations;
69   if (pendingOp != null)
70      if (pendingOp->op == InsertOp)
71         InsertQueueHelper((INSERTQUEUE_OP *)pendingOp);
72      else
73         DeleteQueueHelper(pendingOp);
74   PendingQueueOperations = &des;
75   DeleteQueueHelper(&des);
76   PendingQueueOperations = null;
77   if (des.deletedNode == null)
78      return false;
79   else {
80      *key = des.deletedNode->key;
81      ReleaseNode(des.deletedNode);   // Defined in Figure 3.19
82      return true;
83   }
84 } /* end of DeleteQueue */
85 static void DeleteQueueHelper(DELETEQUEUE_OP *des) {
86   NODE *removeNode = QueueHead;
87   if (removeNode != null && des->deletedNode == null)
88      des->deletedNode = removeNode;
89   if (des->deletedNode != null) {
90      removeNode = des->deletedNode->next;
91      while (LL(&QueueHead) == des->deletedNode)
92         if (SC(&QueueHead,removeNode))
93            break;
94   }
95 } /* end of DeleteQueueHelper */
```

Figure 3.25: Concurrent wait-free priority queue implementation (2nd and last part).

The final step of `DeleteQueueHelper` is to remove the node. This operation is done in a retry loop on lines 91 to 93. Because the nodes that are removed from the list cannot find their way back into the list without being freed by the dequeuer, the test on line 91 can only succeed for a single task. This reasoning is of course only valid in systems with constant priorities.

Wait-free Priority Queue with Scheduled Events for Time-Driven Tasks

Going back to the timer event example at the beginning of this section, and which motivated our need of a priority queue, we are now in position to elaborate on that example, and namely address the synchronization problems when incorporating an autonomous and asynchronous interval timer device interacting with the application through interrupts. There are 2 functions to provide. Function `SetTaskDelay(`*`event,delay`*`)` is called by event-driven tasks and inserts an event into a list sorted by the time of occurrence of the events to schedule. This function basically acts as the enqueueing function that we have expounded and prepares an enqueueing descriptor containing the insertion key.

The second function to implement is the timer-interrupt service routine, which is called whenever an event takes place. Contrary to the dequeueing function that we have detailed, this function can be simplified if we assume that `SetTaskDelay` can only be called from application tasks. Because ISRs can interrupt application tasks and not the converse, there can be no pending dequeueing operation and the timer ISR is free from any intervening actions tampering with the list. This function can then easily remove one or more events at the head of the list without posting a dequeueing operation and schedule these events before setting the time of its next interrupt.

To all appearances, the functions to provide seem to be a straightforward customization of the priority queue presented earlier. However there are 2 new problems that need to be handled and that make this implementation interesting. First the time associated with an event must be on a finite number of bits, and because any time variable is monotonically increasing, an over-

flow will sooner or later happen. Of course one can always add bits to time variables, e.g. 64 or 80, and lower the clock rate so that this overflow takes place in 10, 50 or a 100 years, but in any sorting algorithm, comparisons constitute an important bottleneck, and the clock rate impacts on the reactivity of the application tasks. In our proposed implementation we wish to be independent of the clock rate and allow the event-driven tasks to be very reactive while having fast comparisons that can only be achieved with operands that can fit into a register. We therefore compromise and choose time variables on 32 bits and periodically shift these variables before they can wraparound. Now because the list is sorted by the time that the events occur, an insertion can only begin once the time of occurrence of the event is known. But as stated, time variables need to be shifted to avoid wraparounds, and the timer ISR executes the shift whenever it detects an overflow. So from the standpoint of the enqueuer this can happen any time and namely after setting the occurrence time of the event into the wait-free insertion descriptor but before posting it. Without taking any particular precaution, an event may experience an unwanted additional delay equal to the overflow period.

The second problem to consider is setting the next timer interruption done by the timer ISR whenever it removes the event at the head of the list and by an enqueuer when it inserts an event that occupies the head of the list. In the same way as setting an alarm clock for 7 a.m. when it's past 7, special care must be taken by an enqueuer when setting the interrupt time so that the event occurrence time is not missed.

Abstract Interval Timer Model

Our timer revolves around four memory-mapped registers.

`*Timer_Counter` denotes the content of the 32-bit internal counter register that is programmed to count regularly from 0 to $2^{30}-1$, and once the counter increments to 2^{30} the timer sets an overflow interrupt and the counter rolls back to 0.

Although the counter could count up to $2^{32}-1$ before triggering an interrupt, why halve the range in four? First we need to work with signed values so that it is easy to detect when an event is late and should have occurred before the current time. Second, the delay parameter of `SetTaskDelay` relative to the current time should fit into an integer, and a reasonable a priori compromise between the number of overflows and the highest possible delay is to divide the range in two:

$$\underbrace{2^{30}-1}_{\text{maximum counter value}} + \underbrace{2^{30}}_{\text{maximum delay}} = 2^{31}-1$$

`*Timer_Comparator` contains a 32-bit value which when compared to the counter triggers an interrupt when the values match. It is through this register that events can be scheduled in the future and signaled by the timer ISR when they occur. We assume that this register initially contains the value 0.

`*Timer_Status` contains the cause of the interrupt and is used by the interrupt handler to control its actions. Only counter overflows and compare matches need to be taken into account.

`*Timer_GenerateInterrupt` activates the different interrupts associated with the timer device by setting a specific bit.

The described timer peripheral corresponds to the 32-bit up-counting timers found on STM32 microcontrollers. Like all timer devices, a compare match interrupt is generated when the counter becomes equal to the contents of the comparator. Setting the comparator to a value smaller than or equal to the current value of the counter does not generate an interrupt.

Since interrupt service routines are described later in Chapter 4, we assume that the timer ISR calls function `TimerEventHandler` given in Figure 3.26 whenever it needs to handle an interrupt. As we shall see in Chapter 4, it will be possible to encapsulate all the global variables into a record and to associate this record with the ISR instead of declaring them as we have done here. Function `TimerEventHandler` proceeds in 2 steps. First any pending operation on the sorted list is finalized before continuing to the second step, which investigates each possible interrupt source. The first step should seem quite familiar to the reader. The second step is new.

The status register holds the cause of the interrupt, and a counter overflow is indicated in its LSB (line 23). If the handler is invoked because of an overflow, the handler prepares for its next interrupt by clearing the overflow bit (line 24) and then shifts all time variables by the counter's period (lines 25 and 26). The fact that the time variables have been shifted is stored in a version number (line 27). We will shortly clarify this point when explaining the enqueueing functions.

```
1  typedef struct TIMER_EVENT_NODE { // Blocks that are in the event queue
2    struct TIMER_EVENT_NODE *next;  // Successor node
3    void *event;                    // Event to signal
4    INT32 time;                     // Event occurrence time
5  } TIMER_EVENT_NODE;

6  #define SHIFT_TIME 0x40000000      // Corresponds to 2^30
7  INT16 *Timer_Status;               // Pointers to memory mapped internal registers
8  INT32 *Timer_Counter;              //   Counts until SHIFT_TIME is reached
9  INT32 *Timer_Comparator;
10 INT16 *Timer_GenerateInterrupt;

11 static TIMER_EVENT_NODE *QueueHead = null;
12 static INSERTQUEUE_OP *PendingInsertOperations = null;
13 static UINT16 Version;

14 void TimerEventHandler(void)
15 { // Called by the timer device ISR whenever it gets an interrupt.
16   TIMER_EVENT_NODE *eventNode;
17   INSERTQUEUE_OP *pendingOp;
18   pendingOp = PendingInsertOperations;
19   if (pendingOp != null)           // Is there an incomplete operation under way?
20      InsertQueueHelper(pendingOp,false);
21   PendingInsertOperations = null;
22   do {
23      if (*Timer_Status & 1) {      // Is timer overflow interrupt?
24         *Timer_Status ^= 1;        // Clear interrupt flag
25         for (eventNode = QueueHead; eventNode != NULL; eventNode = eventNode->next)
26            eventNode->time -= SHIFT_TIME;
27         Version += 1;
28      }
29      while (true) {
30         *Timer_Comparator = 0;     // Disable timer comparator
31         *Timer_Status &= ~2;       // Clear comparator interrupt interrupt flag
32         while ((eventNode = QueueHead) != NULL && eventNode->time <= *Timer_Counter) {
33            OSScheduleSuspendedTask(eventNode->event);
34            QueueHead = eventNode->next;
35            ReleaseNode(eventNode); // Defined in Figure 3.19
36         }
37         if (eventNode != null) {   // Program the next timer comparator
38            *Timer_Comparator = eventNode->time;
39            if (*Timer_Comparator > *Timer_Counter)
40               break;
41         }
42         else break;
43      }
44   } while (*Timer_Status != 0);
45 } /* end of TimerEventHandler */
```

Figure 3.26: Scheduling event-driven tasks with a wait-free priority queue.

Compare match interrupts are then processed on lines 29 to 43. This is the second and last interrupt source of the timer device. The comparator register is first cleared so that if there are no longer any events to schedule, the next interrupt will be an overflow which will also correspond to a match between the counter and the comparator. This simplifies the operations in that we do not need to enable or disable compare match interrupts when inserting the first event in the list or removing the last event from the list. Next the compare match flag is cleared in the status register. The order of these 2 operations is not important since the loop ending at line 44 cannot finish if the handler creates new pending interrupts. Each event with an occurrence time smaller than or equal to the current time is then signaled (line 33) and dequeued from the list (line 34). As for our previous implementation, we assume the existence of an initialized stack of free nodes that can supply nodes with function `GetFreeNode` and reclaim nodes with

`ReleaseNode`. Dequeued nodes are returned to the free pool on line 35. At the exit of this loop, we still need to prepare for the next compare interrupt. This is done on line 38 and immediately after a check is done to verify that the counter has not exceeded the next interrupt value. If so, the inner loop emptying the events from the list and signaling them must be reiterated. Note that the comparator may be set to a value that is out of range of the counter. This has no consequence as this event may not be signaled before the handler receives an overflow interrupt that will schedule the event within the range after shifting the occurrence time of this event.

The event insertion functions are given on Figures 3.27 and 3.28. Compared to our previous example inserting an integer, this adapted version inserts events into a time-sorted list and possibly sets the compare register of the timer. Before examining specific parts of this code, a quick skim through its constituents may be helpful. Like our integral priority queue, an insertion is done in 3 stages: posting a descriptor, incrementally helping other insertions and finally inserting the intended event. Function `SetTaskDelay` in Figure 3.27 carries out these operations. Function `InsertQueueHelper` shown on Figure 3.28 also proceeds in 3 stages: first the insertion key is established (lines 80 to 88), followed by a concurrent insertion (lines 89 to 110) and finally setting the compare register (lines 111 to 122).

Because event times may be shifted by the timer handler without the enqueuer's awareness, the insertion key must either be shifted by the timer handler or when a shift takes place, the enqueuer should be notified in some way. The first case is fairly easy. If the enqueueing descriptor is in the path of the timer handler, the hander first inserts the event indicated within the descriptor before shifting its occurrence time. The second case can be dealt with a version number. The handler increments a global version number every time it shifts the time variables. All the enqueuer needs to do is gather all the necessary data that is required to set the insertion key and postpone its determination once the prepared descriptor is in the path of the handler. This explains why we include 2 additional fields to the descriptor of Figure 3.27 (lines 50 and 51) and which are filled by the enqueuer when it collects the initial occurrence time of the event it wishes to insert (lines 59 to 62). Should a shift take place before the enqueuer has the opportunity of

```
46 typedef struct {                    // Insertion information block
47   volatile BOOL done;               // Termination flag
48   TIMER_EVENT_NODE *left;           // Current scanning pointer
49   TIMER_EVENT_NODE *eventNode;      // Node to insert
50   INT32 time;                       // Sorting key
51   UINT16 version;
52   volatile BOOL generatedInterrupt;
53 } INSERTQUEUE_OP;
54 BOOL SetTaskDelay(void *event, INT32 delay) {
55   TIMER_EVENT_NODE *newNode;
56   INSERTQUEUE_OP des, *pendingOp;
57   if ((newNode = GetFreeNode()) == null)
58      return false;
59   do {
60      des.version = Version;
61      des.time = *Timer_Counter + delay;
62   } while (des.version != Version);
63   newNode->time = -1; // Set to an uninitialized sentinel value
64   newNode->next = (TIMER_EVENT_NODE *)&des;
65   des.done = false;
66   des.left = (TIMER_EVENT_NODE *)&QueueHead;
67   des.eventNode = newNode;
68   des.generatedInterrupt = false;
69   pendingOp = PendingInsertOperations;
70   if (pendingOp != null)
71      InsertQueueHelper(pendingOp,true);
72   PendingInsertOperations = &des;
73   InsertQueueHelper(&des,true);
74   PendingInsertOperations = null;
75   return true;
76 } /* end of SetTaskDelay */
```

Figure 3.27: Inserting events into a concurrent priority queue (1st part).

```
77 void InsertQueueHelper(INSERTQUEUE_OP *des, BOOL genInterrupt) {
78   INT32 scheduledTime;
79   TIMER_EVENT_NODE *right, *left;
80   if (LL(&des->eventNode->time) == -1) {
81      if (des->version == Version)
82         scheduledTime = des->time;
83      else
84         scheduledTime = des->time - SHIFT_TIME;
85      while (!SC(&des->eventNode->time,scheduledTime))
86         if (LL(&des->eventNode->time) == -1)
87            break;
88   }
89   left = des->left;
90   while (true) {          // Loop until done
91      right = left->next;  // Get adjacent node
92      if (des->done)       // Has a higher priority task finished the insertion?
93         break;
94      else if (right == des->eventNode) { // Is the event already in the queue?
95         des->done = true;                // Mark that event has been inserted
96         break;
97      }
98      else if (right == null || des->eventNode->time < right->time)) {
99         while (LL(&des->eventNode->next) == des)
100           if (SC(&des->eventNode->next,right))
101              break;
102        while (LL(&left->next) == right)
103           if (des->done || SC(&left->next,des->eventNode))
104              break;
105        des->done = true;  // Mark that event has been inserted
106        break;
107     }
108     else
109        left = des->left = right;
110  }
111  if (genInterrupt && QueueHead == des->eventNode) {
112     do {
113        LL(Timer_Comparator);
114        if (des->generatedInterrupt) return;
115     } while (!SC(Timer_Comparator,des->eventNode->time));
116     if (*Timer_Comparator <= *Timer_Counter)
117        do {
118           LL(Timer_GenerateInterrupt);
119           if (des->generatedInterrupt) return;
120        } while (!SC(Timer_GenerateInterrupt,2))
121  }
122  des->generatedInterrupt = true;
123 } /* end of InsertQueueHelper */
```

Figure 3.28: Inserting events into a concurrent priority queue (2nd and last part).

posting its descriptor, the true insertion key must also be shifted, but only once. This last point is attained by initializing the key to a sentinel value and transferring the correct value only if it still contains the sentinel. This is the first step of function `InsertQueueHelper` where a sentinel value of -1, an impossible value, indicates that the occurrence time of the inserted event has not yet been established. When the version numbers are in agreement, the prepared time is the actual time to transfer (line 82), and otherwise if we assume that the time cannot wraparound more than once between lines 61 and 72, the prepared time needs a single shift (line 84).

An alternative design, but not as good as the one we have presented, would be to store the delay into the insertion descriptor and establish the key in `InsertQueueHelper`. This would eliminate global variable `Version` but adds an extra possibly unwanted delay especially noticeable when the enqueuer is temporarily stalled for some reason.

The last distinctive feature of our new insertion algorithm is to modify the compare register of the timer device and possibly generate an interrupt when the inserted event occupies the head of the list. When describing the timer handler, we have shown that the compare register is always modified before its conclusion. It is therefore useless to generate an interrupt when processing

that interrupt. The second parameter of `InsertQueueHelper` allows the handler to skip this step while all other callers need to modify the next interrupt time. This is done in two steps: first setting the comparator and then verifying that the scheduled interrupt was not missed by comparing the inserted value against the current time. The disposition of these 2 steps is similar (see lines 112 to 115 and 117 to 120). The register is read with an LL instruction so that its corresponding SC fails if there is a context switch. Having an LL without its matching SC has no consequence but allows the task to drop the operation if another task completed it. This is the intention of field `generatedInterrupt` whose purpose is to guarantee that the compare register is updated only once.

Notice that the content of the inserted node is referenced (line 115) even if the node is no longer part of the list. This any happen when a tardy task finishes its insertion but has already been signaled by the timer handler. Again this has no consequence because (1) signals are never lost and the task restarts as soon as possible, and (2) list nodes never disappear. The task may therefore do some idempotent operations but never influence the functioning of the timer device.

Summary

Sharing data is a difficult problem for which there are no general solutions that can take into account all possible interactions and constraints between the involved tasks. The difficulty lies with the order in which the operations need to take place to ensure that there are no race conditions and that each involved task sees a coherent view of the shared data. Three useful techniques come in handy when designing for an embedded application:

- *Data packing* fits the data into a single manipulable word that can be accessed through atomic reads and writes.
- *Data versioning* uniquely tags a set of data at each update so that reader tasks can detect an inconsistent state resulting from a preemption that modified the set in the midst of their reads. This technique can only be applied when a single writer having higher priority than the reader tasks can update the data.
- *Double buffering* allows a flow of information through a pair of buffers in which the writer task fills a buffer while the reader empties the previously filled one, and then the buffers are exchanged. A high throughput with no loss of information is attained when the readers are done with a buffer before the next buffer becomes full.

These techniques fail when the interactions amongst the tasks become more complex and the relative priority order of the involved tasks is unknown or continuously changing. To circumvent this state of affairs, the designer can use atomic instructions to coherently handle primitive data types, or rely on the concurrent data structures provided in the API supplied with *ZottaOS*. These available building blocks solve the two most common communication patterns found in embedded systems: FIFO message queues and a shared buffer with concurrent updates and reads. One great benefit of these built-in data structures is that the concurrent intricacies are encapsulated in the data structure and transparent to the designer.

It is of course impossible to supply every possible data structure within an API and one has to sometimes get one's hands dirty. To this end we have illustrated how to build concurrent stacks and priority queues using lock-free constructs and a wait-free technique called *incremental helping* that requires posting operations so that intervening tasks can concurrently assist ongoing operations.

Through the examples in this chapter we hope that you are better equipped to design your own data structure or implement one from a research paper, although these latter ones can be greatly simplified when taking a particular and distinctive feature of *ZottaOS*: the execution priority of a task instance is constant throughout the lifetime of the instance. In other words, an instance cannot be preempted by an instance that it itself preempted.

Bibliography

[AND97] J.H. Anderson, S. Ramamurthy, and R. Jain, *Implementing Wait-Free Objects in Priority-Based Systems*, Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'97), pp. 229-238, Aug. 1997.

[ATM11] AVR32 Architecture Manual, Atmel, Rev. 32000D, April 2011.

[CHE97] J. Chen and A. Burns, *A Three-Slot Asynchronous Reader/Writer Mechanism for Multiprocessor Real-Time Systems*, University of York Computer Science Report, YCS 286, 1997.

[EVE09] C. Evequoz, Population Oblivious Wait-Free Concurrent Priority Queues Using Single Word Synchronization Primitives, unpublished manuscript.

[HAR01] T.L. Harris, *A Pragmatic Implementation of Non-Blocking Linked-Lists*, Proceedings of the 15th International Conference on Distributed Computing (DISC 2001), pp. 300-314, Oct. 2001.

[HEN02] N. Henderson and S. E. Paynter, *The Formal Classification and Verification of Simpson's 4-Slot Asynchronous Communication Mechanism*, Proceedings of the International Symposium of Formal Methods Europe on Formal Methods (FME'02), No 2391 in Lecture Notes in Computer Science, pages 350-369, Springer, 2002.

[HER91] M.P. Herlihy, *Wait-Free Synchronization*, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 13, No. 1, pp. 124-149, Jan. 1991.

[MOI97] M. Moir, *Practical Implementations of Non-Blocking Synchronization Primitives*, Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'97), pp. 219-228, Aug. 1997.

[RAM96] S. Ramamurthy, M. Moir, and J.H. Anderson, *Real-Time Object Sharing with Minimal System Support*, Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'97), pp. 233-242, May 1996.

[RUS02] J. Rushby, *Model-Checking Simpson's Four-Slot Fully Asynchronous Communication Mechanism*, Technical Report, Computer Science Laboratory, SRI International, July 2002.

[SIM90] H.R. Simpson, *Four-Slot Fully Asynchronous Communication Mechanism*, IEE Proceedings, Vol. 137 Part E(1), pages 17-30, Jan. 1990.

[STM11] STM32F10xxx/20xxx/21xxx/L1xxxx Cortex-M3 Programming Manual, STMicroelectronics, PM0056, Doc ID 15491 Rev 4, March 2011.

[STM12] STM32F4xxx Cortex-M4 Programming Manual, STMicroelectronics, PM0214, Doc ID 022708 Rev 1, February 2012.

Chapter 4

Interrupt Subsystem

Simple device programming can be done by testing the status/control bits associated with an I/O device in a tight loop. When the device becomes free, a task can then set-up the next data-transfer, wait for the transfer to take place, and then retrieve and process the data. Although this is a very simple and reliable method, it is a bad method for time-critical systems because an event might occur while the processor is stuck in the loop. Overall system reactivity is poor. Moreover, while the application is actively looping it could have been put into a low-power mode instead. Power efficiency is also very poor for this method.

Embedded systems tend to be interrupt-driven. This is because their unit cost must be minimized and therefore they tend to use cheap and unsophisticated peripherals that require constant attention by the processor. Also, most microcontrollers can go into a sleep mode while waiting for an event. This is an important implementation optimization that should be considered in all applications. This chapter explains how easily it is to handle interrupts in *ZottaOS*.

ISR

Each interruption source is associated with a memory location that mimics an interrupt vector and for which each entry of the vector contains the address of a descriptor that can handle the interrupt. The minimal content of this descriptor is the address of the **interrupt service routine** (ISR). For example,

```
1  typedef struct myDescriptorDef {
2    void (*myISR)(struct myDescriptorDef *);
3    // other information used by myISR
4  } myDescriptorDef;
```

When an interrupt occurs, further interrupts from the same source are disabled so that an interrupt burst will never trouble the processing of the interrupt. The processor registers are then saved onto the stack, and control is given to the interrupt service routine specified within the descriptor. This is equivalent to the following code snippet:

```
Vector[sourceEntry]->myISR(&Vector[sourceEntry])
```

Because the interrupt service routine receives a pointer to the interrupt descriptor, it can store in that descriptor all the information that it needs to process the interrupt without resorting to global or static variables. When the last instruction of the ISR is reached, control goes back to the interrupt service manager (the caller of the ISR), which restores the saved registers of the interrupted task instance before resuming the instance. When further interrupts are to be received by the ISR, interrupts corresponding to the particular source must be re-enabled before exiting the ISR. The code below provides the skeleton of an ISR.

```
1  void myInterruptHandler(struct myDescriptorDef *myDescriptor)
2  {
3    // process the interrupt
4    // enable interrupt source if needed
5    // exit and return to the interrupted task instance
6  }
```

Multiple-Sourced Devices

Some microcontrollers have peripherals, which also have their own interrupt vector. These are usually accessible via a dedicated register. We call these devices **multiple-sourced devices**. In terms of the definition of the ISR introduced above, the ISR functions along similar lines: when a peripheral interrupt is raised for a particular source within the peripheral, that and only that particular interrupt is disabled prior to calling the ISR. And like for single-sourced interrupts, the interrupt source must be re-enabled in the ISR if further interrupts are expected. Basically, ZottaOS flattens the resulting interrupt matrix introduced by multiple-sourced devices into a single vector.

Binding

ZottaOS provides a function to bind an ISR to a particular interrupt source. This is done via

```
void OSSetISRDescriptor(UINT8 index, void *descriptor)
```

where `index` denotes an entry into the interrupt vector of *ZottaOS* and the second parameter corresponds to the above descriptor. Interrupt binding can be done anywhere: during the application initializing phase or during application task execution. The only precaution to take is with regards to the memory allocation needed to store the descriptor. A descriptor cannot be stored on the stack since it must exist when the ISR is invoked. Typically this descriptor is only used by the ISR and initialized prior to starting the application tasks, the main program can invoke `OSMalloc` to allocate the descriptor and reference the descriptor with the returned address.

Finally, *ZottaOS* also supplies a function to recover the descriptor after it has been bound to an interrupt source:

```
void *OSGetISRDescriptor(UINT8 index)
```

where `index` specifies the entry of the interrupt vector.

Interrupt Programming Sample

In this section we illustrate the coding of an interrupt service routine and the constructs introduced in this chapter. The provided sample is basically the ISR of a circular-buffered UART output port. This example is sufficiently simple to understand how to use the interrupt subsystem and yet sufficiently complex so as to not fall into straightforwardness.

The UART accesses the output port by the internal data register (`UCA0TXBUF`), and it also has an interrupt register to enable of disable interrupts (`UCA0IE`). We assume that the UART also has its own source vector and which can assume the bit values given in the following table.

Value	Meaning
0x00	No pending interrupt
0x02	Data received
0x04	Data register empty

The only interrupt we will be dealing with is the value 0x04.

Our implementation uses a circular (or bounded) buffer of 8 bytes and consists of 2 distinct parts:

1. An interrupt service routine associated with the output UART provides the next byte to transmit once the previous transmitted byte is sent. This is done by copying a byte from the buffer into the data register of the UART. As soon as this byte has been sent by the UART, the UART generates an interruption so that the next byte stored in the buffer can be sent. When the buffer becomes empty, UART interruptions are disabled.
2. A service function that can be invoked by an application task to insert bytes into the buffer accessed by the ISR. Because the output buffer is bounded, there are 2 possible designs for this function. It can insert a single byte into the buffer and return a Boolean indicating whether the byte was inserted, or it can itself take a buffer as parameter and transfer as many bytes from it as possible. For instance, if at the time of the call the output buffer holds 5 bytes, at most 3 bytes can be taken from the parameter and placed into the output buffer. The prototype of this function is

   ```
   UINT8 PutBytes(UINT8 *data, UINT8 size)
   ```

   where `data` holds the bytes to transfer to the output buffer accessed by the ISR. Parameter `size` indicates the maximum number of bytes from `data` that can be transferred. This function returns the number of bytes actually transferred.

The UART descriptor should hold the output buffer, and the indices where the insertion and extraction entries take place. Note that when these two indices are equal, the buffer can either be

full or empty. To differentiate between these 2 states, we shall also need a counter that indicates the number of pending bytes to transmit actually held in the buffer.

ISR Data Definition

```
typedef struct UART_DESCRIPTOR {
  void (*ISR)(struct UART_DESCRIPTOR *);
  UINT8 buffer[8]; // circular buffer with the data to transmit
  UINT8 items;     // number of remaining bytes to send
  UINT8 in;        // buffer index that holds the next free entry
  UINT8 out;       // buffer index that holds the next byte to send
} UART_DESCRIPTOR;
```

The initializations of the data structure should be as follows:

1. The interrupt service routine (i.e., field `ISR`) for the UART must of course correspond to the one that we shall carry out below;
2. The last 3 fields must be set to 0 to respectively indicate that the buffer is empty, that the next insertion and extraction take place for entry 0.

The interrupt service routine is given below.

ISR Sample

```
void UartOutputByte(UART_DESCRIPTOR *uart)
{ // Handles only UART transmits when the data register becomes empty
  if (uart->items > 0) {     // Is there something to send?
     UCA0TXBUF = uart->buffer[uart->out]; // Output data byte
     uart->items -= 1;       // One less byte in the buffer
     uart->out = (uart->out + 1) % 8;    // Prepare for next output
  }
  if (uart->items != 0) {    // Are we done?
     UCA0IE |= UCTXIE;       // No: enable interrupts to come back
  }                          //   once transmit register becomes free.
} /* end of UartOutputByte */
```

Initially UART interrupts are masked. When insertions into the circular buffer take place, they should be unmasked so that an interrupt can be generated and the processor transferred to the ISR which in turn can start to output the data through the UART port. Notice that the ISR only handles interrupts when an empty data register is raised. As mentioned at the beginning of this chapter, our UART is an example of a multiple-sourced device and all its other interrupt sources must be carried-out by other ISRs. Our implementation is nevertheless robust provided that the interrupt service manager filters-out all the interrupt sources, which are not required by the application. Such is the case when using *ZottaOS Configurator Tool* where you can select the specific interrupt sources that are required.

ISR Registration

For `UartOutputByte` to be recognized by *ZottaOS*, we still need to register it and to bind it to the UART peripheral. This is done in the main program given below.

```
int main(void)
{
  UART_DESCRIPTOR *uart; // local UART structure
  // 1. Inhibit the watchdog timer
  // 2. Initialize the system clock
  // 3. Create the application tasks
  // 4. Set the baud rate of the UART
  // 5. Configure other used peripherals
  // 6. Prepare the descriptor of the UART
  uart = (UART_DESCRIPTOR *)OSMalloc(sizeof(UART_DESCRIPTOR));
  uart->ISR = UartOutputByte;
  uart->items = 0;
  uart->in = 0;
  uart->out = 0;
  OSSetISRDescriptor(OS_IO_USCIA0,uart);
  // Do other initializations here
  return OSStartMultitasking();
} /* end of main */
```

Notice that our descriptor is declared locally as a reference and allocated dynamically. Recall that prior to `OSStartMultitasking`, `OSMalloc` permanently allocates memory blocks beneath the stack and each block has a lifetime equal to that of the application. Since `PutBytes` will need to access the circular buffer contained within the descriptor in order to introduce the bytes that are to be transmitted by the UART, the function can retrieve the descriptor via function `OSGetISRDescriptor`.

Filling Output Bytes

The `PutBytes` function and the ISR can concurrently access the UART descriptor, but the only field where there is a possible conflict is the counter field `items`. Indeed, if we assume that only one application task calls the `PutBytes` function, there can only be a conflict between the ISR and this task. Since the ISR and `PutBytes` do not simultaneously access the same entry of the output buffer we do not need to take any precaution regarding the circular buffer. Also, the ISR and the function use their own indices (`in` and `out`).

```
1  UINT8 PutBytes(UINT8 *data, UINT8 size)
2  {
3    UART_DESCRIPTOR *uart;
4    UINT8 outData = 0; // number of bytes actually transferred to the UART
5    UINT8 oldItems, newItems;
6    uart = (UART_DESCRIPTOR *)OSGetISRDescriptor(OS_IO_USCIA0);
7    while (size-- > 0 && uart->items + outData < 8) {
8       uart->buffer[uart->in] = *data++; // Put a byte into the buffer
9       uart->in = (uart->in + 1) % 8;    // Prepare for the next input
10      outData += 1;
11   }
12   // At this point, outData bytes have been inserted and the item count
13   // must be updated. Because this counter is concurrently accessed, the
14   // equivalent of an atomic fetch and add instruction can do the job.
15   do {
16      oldItems = OSUINT8_LL(&uart->items);
17      newItems = oldItems + outData;
18   } while (!OSUINT8_SC(&uart->items,newItems));
19   // Enable interrupts. Note that the first interruption can immediately
20   // take place if the UART device is available.
21   UCA0IE |= UCTXIE; // Enable UART interrupts
22   return outData;
23 } /* end of PutBytes */
```

Non Blocking Concurrent Update

Updates of the `items` field are done by means of an equivalent atomic fetch-and-add instruction, which adds the contents of a memory location with an increment and stores the result of the addition back into the memory location. This update is performed with the load linked/store conditional (LL/SC) pair of instructions. Recall that when a value is loaded into a register with LL, a reservation is placed on that memory address. If that reservation still exists when the store conditional operation (SC) is done, the store will be successfully accomplished. If between the LL and the SC, another task or ISR writes to the memory location, the reservation is cleared, and any other task will fail its store conditional. When this happens, the program must restart and try the operation again, beginning with LL.

On line 16 of `PutBytes`, the current value of the number of unsent bytes is read and reserved into `oldItems`. Local variable `newItems` is then assigned to the new value that the counter should contain. Finally, the SC on line 18 tries to store `newItems` into `uart->items`, and the loop is restarted if the store fails.

Why Is It So Simple and Why Does It Work?

When reading what a designer must complete to have an ISR up and running, one cannot help to ask the obvious question: where's the catch? In what follows, we explain what goes on behind the scene of what we have described in this chapter.

Interrupt Vector Table

An interrupt, whether generated to manage a peripheral event or an exception condition such as an access violation to a memory address, implies a transfer of control to an ISR handler. Because the processor needs to know where the handler is, it must be placed at some predetermined address in memory. Because these addresses are grouped together, these contiguous memory locations constitute an **interrupt vector table**.

The service provided to the interrupt depends on the source of the interrupt. For example, if the interrupt is caused to a power failure, then the required taken action may be to save some values so that these are not lost when the power is restored. Most microcontrollers have different interrupt sources within an interrupt. For example, the MSP430x2xx series has a flag register indicating which I/O pin requests an interrupt and it is up to the ISR handler to check this register in order to find the real source and apply the appropriate action. However on the MSP430x5xx series, a register holds a number corresponding to the highest pending request which can be added to the program counter to immediately jump to the routine associated with the I/O pin. Hence, even for the same product, the nature of the ISR is not orthogonal and hampers producing portable code.

In *ZottaOS*, a specific ISR manager is bound to each address of the interrupt vector table of the microcontroller. This manager simply determines the particular source of its associated peripheral interrupt or exception and which depends on the type of the device. Once found the manager transfers control to an entry in the kernel internal interrupt table that is indexed by the 2nd parameter of function `OSSetISRDescriptor`. Figure 4.1 shows this process. *ZottaOS* basically provides the user with another interrupt vector table so that the intricacies of uncovering the source can be hidden. This has a number of advantages:

- modular and portable processing of each interrupt,
- easy detection of unprocessed interrupt services in debug mode. *ZottaOS* can run under 2 modes. When symbolic flag `DEBUG_MODE` is specified, peripheral devices that have no entry in the kernel's interrupt table fall into an infinite loop that can be identified by a debugger. Without the `DEBUG_MODE` flag, no code is generated.

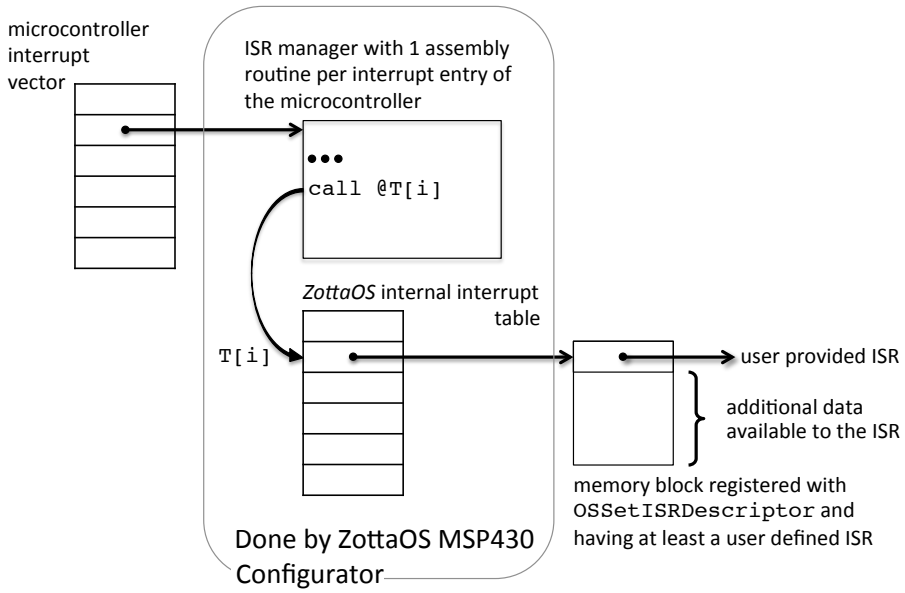


Figure 4.1: Inner working of *ZottaOS* interrupt subsystem.

This solution is only practical if a configuration tool is available to generate the code of the ISR manager and size the table used by the kernel. This is the main pursued goal of *ZottaOS MSP430 Configurator*, which we now describe.

Interrupt Configuration Tool

ZottaOS MSP430 Configurator is a step-driven dialog box application where one or more steps are displayed in a page, and the user progresses page by page until he reaches step 7 where he can generate the files corresponding to his selections. At step 1, you choose the MSP430 or

CC430 that your are working with. Step 3 presents the list of available modules having interrupts. These are grouped into pushbuttons, which when pressed brings up a particular dialog box displaying all the possible interrupt sources of the selected devices. For example, clicking on the button labeled Ports from step 3 for MSP430AFE233 and MSP430FR5738 displays the 2 dialog boxes shown at the bottom of Figure 4.2.

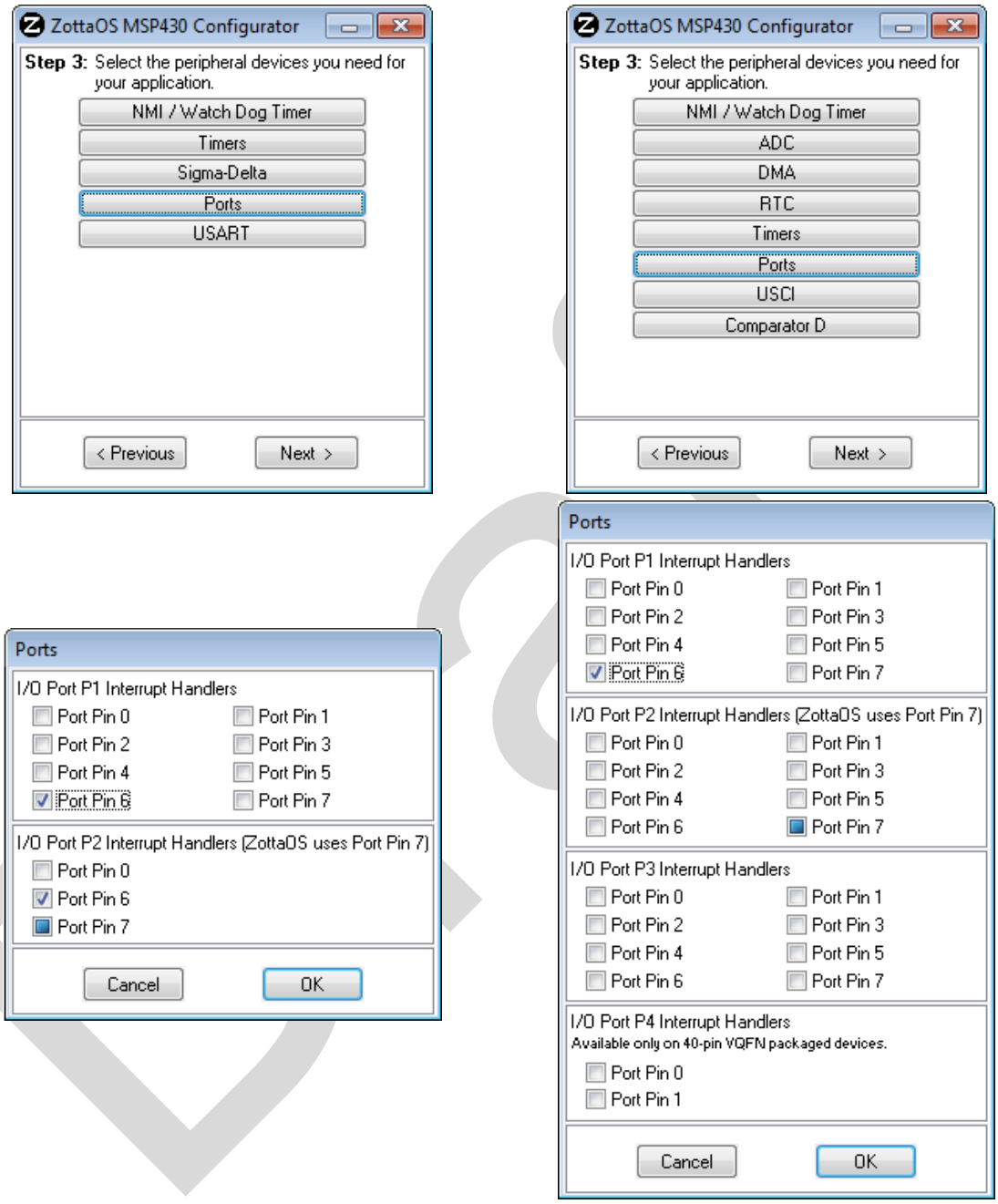


Figure 4.2: Third step and port dialog boxes for 2 different microcontrollers.
The 2 dialog boxes to the left apply to MSP430AFE233
and those to the right correspond to MSP430FR5738.

Continuing this trivial example and selecting only port P1.6 for both microcontrollers and then generating the files at step 7 produces a header file and an assembler file. Albeit their names and the internal timer used by *ZottaOS*, both header files contain the following code snippet:

```
1 /* Entries into the kernel ISR interrupt vector. */
2 #define OS_IO_PORT1_6                0  /* Port 1 Pin 6 */
3
4 /* The next symbol is the size of the interrupt table. */
5 #define OS_IO_MAX 2 /* Defined as the last defined OS_IO_XXX + 2 */
```

The last define fixes the size of the internal interrupt vector which only has a single entry named `OS_IO_PORT1_6`.

The corresponding assembler files are however quite different as the MSP430AFE233 uses an interrupt flag register (*P1IFG*) while the MSP430FR5738 relies on an interrupt vector register to locate the port pin that raised the interrupt. The code snippets below emphasize this difference, but the important point to *keep in mind is that this code is completely transparent to the application*.

```
1  ; I/O Port P1 interrupt handler (int04).
2    default_handler int04
3    mov.b &P1IFG,r13                ; Load P1 interrupt flag register to get pin sources
4    and.b &P1IE,r13                 ; Mask all non-pending sources
5    bic.b r13,&P1IE                 ; Disable all Port 1 pending interrupts
6    push.b &P2IE                    ; Save mask Port 2 (see int04)
7    bic.b #BIT6,&P2IE               ; Disable I/O Port 2 pin 6 interrupts (see int04)
8    eint                            ; Enable all interrupts
9    SaveCtx                         ; Save context of the application task
10   .if $defined(DEBUG_MODE)
11       bit.b #BIT0,r13             ; Test if pin 0 is pending
12       jnz UndefHandlerPort1       ; Error: loop indefinitely
13   .endif
14   .if $defined(DEBUG_MODE)
15       bit.b #BIT1,r13             ; Test if pin 1 is pending
16       jnz UndefHandlerPort1       ; Error: loop indefinitely
17   .endif
18   .if $defined(DEBUG_MODE)
19       bit.b #BIT2,r13             ; Test if pin 2 is pending
20       jnz UndefHandlerPort1       ; Error: loop indefinitely
21   .endif
22   .if $defined(DEBUG_MODE)
23       bit.b #BIT3,r13             ; Test if pin 3 is pending
24       jnz UndefHandlerPort1       ; Error: loop indefinitely
25   .endif
26   .if $defined(DEBUG_MODE)
27       bit.b #BIT4,r13             ; Test if pin 4 is pending
28       jnz UndefHandlerPort1       ; Error: loop indefinitely
29   .endif
30   .if $defined(DEBUG_MODE)
31       bit.b #BIT5,r13             ; Test if pin 5 is pending
32       jnz UndefHandlerPort1       ; Error: loop indefinitely
33   .endif
34   bit.b #BIT6,r13                 ; Test if pin 6 is pending
35   jz Next7PORT1
36   mov #_OSTabDevice,r12           ; Load ZottaOS device handler table in R12
37   mov OS_IO_PORT1_6(r12),r12      ; Get entry to the interrupt
38   push r13                        ; Save working register R13
39   call @r12                       ; Continue with the interrupt routine of pin 6
40   pop r13                         ; Restore working register R13
41 Next7PORT1:
42   .if $defined(DEBUG_MODE)
43       bit.b #BIT7,r13             ; Test if pin 7 is pending
44       jnz UndefHandlerPort1       ; Error: loop indefinitely
45   .endif
46   br #EndIntNoClearNoSaveCtx      ; Restore context
47   .if $defined(DEBUG_MODE)
48 UndefHandlerPort1:
49      dint
50      jmp UndefHandlerPort1        ; Error: loop indefinitely
51   .endif
52 ; end of I/O Port P1 interrupt handler (int04)
```

Figure 4.3: ISR manager for Port 1 device extracted from *ZottaOS_msp430afe2x3.asm*.

```
1  ; I/O Port P1 interrupt handler (int47).
2    default_handler int47
3    push.b &P2IE                      ; Save mask Port 2 (see int47)
4    bic.b #BIT6,&P2IE                 ; Disable I/O Port 2 pin 6 interrupts (see int47)
5    add &P1IV,pc                      ; Add offset to jump table
6    .if $defined(DEBUG_MODE)
7       jmp UndefHandlerPort1          ; Vector 0: No interrupt
8       jmp UndefHandlerPort1          ; Vector 2: Port 1 pin 0
9       jmp UndefHandlerPort1          ; Vector 4: Port 1 pin 1
10      jmp UndefHandlerPort1          ; Vector 6: Port 1 pin 2
11      jmp UndefHandlerPort1          ; Vector 8: Port 1 pin 3
12      jmp UndefHandlerPort1          ; Vector 10: Port 1 pin 4
13      jmp UndefHandlerPort1          ; Vector 12: Port 1 pin 5
14      jmp Port1_6                    ; Vector 14: Port 1 pin 6
15      jmp UndefHandlerPort1          ; Vector 16: Port 1 pin 7
16 Port1_6:
17   .else
18      nop                            ; Vector 0: No interrupt
19      nop                            ; Vector 2: Port 1 pin 0
20      nop                            ; Vector 4: Port 1 pin 1
21      nop                            ; Vector 6: Port 1 pin 2
22      nop                            ; Vector 8: Port 1 pin 3
23      nop                            ; Vector 10: Port 1 pin 4
24      nop                            ; Vector 12: Port 1 pin 5
25      ; Fall into handler for port 1 pin 6 interrupt (no JMP required).
26   .endif
27   bic.b #BIT6,&P1IE                 ; Disable I/O Port 1 pin 6 interrupt
28   eint                              ; Enable all interrupts
29   mov #OS_IO_PORT1_6,r13            ; Load interrupt entry to ZottaOS device handler table in R13
30   SaveCtx                           ; Save context of the application task
31   mov #_OSTabDevice,r12             ; Load ZottaOS device handler table in R12
32   add r13,r12                       ; Add table offset to get the interrupt entry
33   mov @r12,r12                      ; Get entry to the interrupt
34   call @r12                         ; Continue with the interrupt routine
35   br #EndIntNoClearNoSaveCtx        ; Restore context
36   .if $defined(DEBUG_MODE)
37 UndefHandlerPort1:
38      dint
39      jmp UndefHandlerPort1          ; Error: loop indefinitely
40   .endif
41 ; end of I/O Port P1 interrupt handler (int47)
```

Figure 4.3: Matching ISR manager snippet for Port 1 device extracted from *ZottaOS_msp430fr5738.asm*.

Interrupt Overloads

Many interrupt-driven systems are vulnerable to interrupt overload, a condition where interrupts are raised often enough that user tasks are starved. This is mainly caused by the fact that interrupts run at higher priority compared to the tasks. For example, toggle switches are subject to rebounds during the transitions of their states and can create transient interrupts in the range of 500 Hz to 1 kHz. Techniques to debounce switches are well-known among embedded system developers, but since this topic is important and common to most designs, we discuss some of these techniques as they apply to *ZottaOS*.

Preventing interrupt overload boils down to forestalling the processor from handling interrupts when these exceed some given rate. This can be achieved by some low-pass filter in hardware or by tampering the enable/disable bit associated the interrupt. One common technique is to make use of a timer to enforce a minimum interarrival time between interrupts. This is a complete interrupt-driven approach. A timer is started in the ISR of the interrupt that is to be controlled, and when it expires, its handler re-enables the controlled interrupt. This simple solution requires a timer with as many compare registers as there are interrupts to control, where each compare register is allotted to a single interrupt.

When there are more interrupts to control than there are compare registers available, the solution becomes overwhelmingly complex: a queue of time events sorted by their order of occur-

rence must be maintained, and when a interrupt occurs, its ISR must insert its re-enable time into the queue, then check whether it is the first event that occurs and if so modifies the timer's compare register. Although this may seem to be straightforward, there are many subtleties that complicate this design and that must be taken care of when nested interrupts are allowed. For example, what happens when an interrupt preparing the timer's comparator gets preempted by another interrupt with a shorter expiration time? When the first interrupt resumes, it can overwrite the compare register established by the second interrupt.

A simpler solution is to make use of the tasking offered by *ZottaOS*. As a running example, we assume that an external switch button is connected to the interrupt-driven digital port P1.1, and to keep the example simple enough, we assume that a counter is incremented each time it is pressed.

Figure 4.4 illustrates the implementation. Function `InitButton` installs the ISR and a periodic task named `ButtonDebouncer` whose purpose is to re-enable the switch. This function also associates the pointer to the counter with the ISR that gets incremented each time the button is pressed.

```
1  typedef struct ButtonDescriptor {
2    void (*handler)(struct ButtonDescriptor *);
3    UINT16 *counter;
4  } ButtonDescriptor;
5  void InitButton(UINT16 *counter, UINT32 minArrivalTime)
6  {
7    ButtonDescriptor *button;
8    button = (ButtonDescriptor *)OSMalloc(sizeof(ButtonDescriptor));
9    button->handler = ButtonISR;
10   button->counter = counter;
11   OSSetISRDescriptor(OS_IO_PORT1_1,button);
12   OSCreateTask(ButtonDebouncer,0,minArrivalTime,minArrivalTime,NULL);
13   P1OUT |= 0x02;  // Button is pressed when bit 1 is = 0
14   P1DIR &= ~0x02; // Set P1.1 as an input
15   P1SEL &= ~0x02; // Set P1.1 as a general purpose I/O
16   P1IES |= 0x02;  // Allow interrupts on descending edges
17   P1IFG &= ~0x02; // Clear the interrupts flag
18   P1IE |= 0x02;   // Enable interrupts
19 } /* end of InitButton */
20 void ButtonISR(ButtonDescriptor *button)
21 {
22   *button->counter += 1; // Handle the interrupt
23 } /* end of ButtonISR */
24 void ButtonDebouncer(void *arg)
25 {
26   P1IE |= 0x02;          // Re-enable Port 1.1 interrupts
27   OSEndTask();
28 } /* end of ButtonDebouncer */
```

Figure 4.4: Periodic interrupt re-enabling not guaranteeing a minimal time before the next interrupt.

This first solution has a number of flaws. The astute reader may notice that if an interrupt occurs immediately before `ButtonDebouncer` is scheduled, the next interrupt can occur before `minArrivalTime` ticks. Two consecutive interrupts are no longer spaced by at least `minArrivalTime` ticks. This is because immediately after re-enabling the interrupt another one may be pending. To correct this state of affairs, the idea is to execute 2 successive instances of `ButtonDebouncer` before re-enabling the interrupt. Figure 4.5 shows this modification.

While the `done` field associated with the button ISR = 0, task `ButtonDebouncer` does nothing, but as soon as this value is set to 2, the first invocation of `ButtonDebouncer` sets the value to 1 and then the second re-enables the interrupt.

```
1  typedef struct ButtonDescriptor {
2    void (*handler)(struct ButtonDescriptor *);
3    UINT16 *counter;
4    UINT8 done;
5  } ButtonDescriptor;
6  void InitButton(UINT16 *counter, UINT32 minArrivalTime)
7  {
8    ButtonDescriptor *button;
9    button = (ButtonDescriptor *)OSMalloc(sizeof(ButtonDescriptor));
10   button->handler = ButtonISR;
11   button->counter = counter;
12   buton->done = 0;
13   OSSetISRDescriptor(OS_IO_PORT1_1,button);
14   OSCreateTask(ButtonDebouncer,0,minArrivalTime,minArrivalTime,button);
15   // Initialize port P1.1 as in Figure 4.4
16 } /* end of InitButton */
17 void ButtonISR(ButtonDescriptor *button)
18 {
19   *button->counter += 1;  // Handle the interrupt
20   button->done = 2;       // Indicate that an interrupt occurred
21 } /* end of ButtonISR */
22 void ButtonDebouncer(void *arg)
23 {
24   ButtonDescriptor *button = (ButtonDescriptor *)arg;
25   if (button->done > 0)             // Has an interrupt occurred?
26      if ((button->done -= 1) == 0) // Second pass?
27         P1IE |= 0x02;              // Re-enable Port 1.1 interrupts
28   OSEndTask();
29 } /* end of ButtonDebouncer */
```

Figure 4.5: Periodic interrupt re-enabling with minimum interspacing.

The second flaw with the designs shown so far is that the ISR does the processing. Although this is a not a problem in our running example as the ISR is short, the solution cannot scale to situations demanding more computations. This problem can easily be remediated by moving line 19 immediately after line 25. However there is another reason why the resulting implementation beneficial: the interrupt is processed at a rate not exceeding that of the periodic task. Not only is our third solution, the best so far, suitable when there is more substantially more processing than a single addition, it is also simpler. The ISR simply signals the presence of an interrupt by a flag shared with the periodic task as illustration in Figure 4.6 on the next page.

The counter is incremented at a maximum rate given by the period of task `ButtonCounter`. The ISR sets its pending flag to indicate that an interrupt occurred, and task `ButtonCounter` performs the processing. Notice that `ButtonCounter` first resets the pending flag before re-enabling the interrupt in order to avoid missing the next interrupt if it occurs immediately after re-enabling it.

The previous solutions require that the user release the pushbutton before it can create a new event. If we allow counting while the button is pressed, we can slightly modify task `ButtonCounter` to continuously increment the counter at the same rate that interrupts are permitted. Figure 4.7 portrays this version. The main idea behind this design is to clear the pending flag only when the button is no longer pressed.

```
typedef struct ButtonDescriptor {
  void (*handler)(struct ButtonDescriptor *);
  UINT16 *counter;
  BOOL pending;       // Signal an interrupt
} ButtonDescriptor;

void InitButton(UINT16 *counter, UINT32 minArrivalTime)
{
  ButtonDescriptor *button;
  button = (ButtonDescriptor *)OSMalloc(sizeof(ButtonDescriptor));
  button->handler = ButtonISR;
  button->counter = counter;
  buton->pending = FALSE;
  OSSetISRDescriptor(OS_IO_PORT1_1,button);
  OSCreateTask(ButtonDebouncer,0,minArrivalTime,minArrivalTime,button);
  // Initialize port P1.1 as in Figure 4.4
} /* end of InitButton */

void ButtonISR(ButtonDescriptor *button)
{
  button->pending = TRUE;     // Indicate an interrupt occurred
} /* end of ButtonISR */

void ButtonCounter(void *arg)
{
  ButtonDescriptor *button = (ButtonDescriptor *)arg;
  if (button->pending) {      // Has an interrupt occurred?
     *button->counter += 1;   // Process the interrupt
     button->pending = FALSE;
     P1IE |= 0x02;            // Re-enable Port 1.1 interrupts
  }
  OSEndTask();
} /* end of ButtonCounter */
```

Figure 4.6: Periodic interrupt processing with minimum interspacing.

Continuous Interrupt Processing

```
void ButtonCounter(void *arg)
{
  ButtonDescriptor *button = (ButtonDescriptor *)arg;
  if (button->pending) {        // Has an interrupt occurred?
     *button->counter += 1;     // Process the interrupt
     if ((P1IN & 0x02) != 0) {  // Is button no longer pressed?
        button->pending = FALSE;
        P1IE |= 0x02;           // Re-enable Port 1.1 interrupts
     }
  }
  OSEndTask();
} /* end of ButtonCounter */
```

Figure 4.7: Continuous processing of an interrupt at a maximum rate.

The last 2 solutions perform well when the interrupt constantly overloads the processor but introduce useless reactivations of the periodic task during underloads. This is another flaw with the designs presented so far, and which is reminiscent of a polling approach, where a tight loop continuously checks the status of the pending flag. The proposed solutions are nevertheless better because they allow the processor to do valuable work between their successive pollings.

Another solution, introducing lower overhead, can be obtained by regrouping the different interrupts together into a single periodic task. This is a straightforward extension when the processing rates of the interrupts are the same. When they are not, simply picking the least common multiple (LCM) of the arrival rates for all interrupt sources is likely to result in a very high frequency and thus poor overall performance. We can round up the maximum allowed rates to some multiples of a large number, allowing a single slower periodic task service them all and still maintain reasonable protection from overloads. For example, an application with 2 sources with maximum arrival rates of 90, and 310 Hz can be serviced by a single task running at 100 Hz.

The following example illustrates this solution. In this example we assume that there are 2 buttons to control, and the first can be executed 3 times faster than the rate of the second. Each interrupt is associated with a period counter, which is periodically decremented when there is a pending request to service. The pending interrupt is serviced only when this counter reaches 0, at which time further interrupts for the same source are re-enabled.

```
typedef struct ButtonDescriptor {
  void (*handler)(struct ButtonDescriptor *);
  BOOL pending;          // Signal an interrupt
} ButtonDescriptor;
typedef struct {
  BOOL *pending[2];      // Pointer field pending of ButtonDescriptor
  UINT8 macroPeriod[2]; // Number of periods before processing interrupt
  UINT8 period[2];       // Remaining periods after an interrupt
  UINT16 *counter[2];    // Variable processed by an interrupt
} LumpedButtons;
void InitButton(UINT16 *counter1, UINT16 *counter2, UINT32 lcmArrivalTime)
{
  LumpedButtons *buttons = (LumpedButtons *)OSMalloc(sizeof(LumpedButtons));
  // First button, OS_IO_PORT1_1, has 1 interarrival time
  buttons->pending[0] = InitButtonISR(OS_IO_PORT1_1);
  macroPeriod[0] = period[0] = 0;
  counter[0] = counter1;
  // Second button, OS_IO_PORT1_2, has 3 interarrival time
  buttons->pending[1] = InitButtonISR(OS_IO_PORT1_2);
  macroPeriod[1] = period[1] = 2;
  counter[1] = counter2;
  OSCreateTask(LumpedButtonCounters,0,lcmArrivalTime,lcmArrivalTime,buttons);
  // Initialize ports P1.1 and P1.2 as in Figure 4.4
} /* end of InitButton */
UINT8 *InitButtonISR(UINT8 portID)
{
  ButtonDescriptor *button;
  button = (ButtonDescriptor *)OSMalloc(sizeof(ButtonDescriptor));
  button->handler = ButtonISR;
  button->pending = FALSE;
  OSSetISRDescriptor(portID,button);
  return &button->pending;
} /* end of InitButtonISR */
void ButtonISR(ButtonDescriptor *button)
{ // Same ISR specification for both buttons
  button->pending = TRUE;
}
void LumpedButtonCounters(void *arg)
{
  UINT8 i;
  LumpedButtons *buttons = (LumpedButtons *)arg;
  for (i = 0; i < 2; i += 1)
     if (*buttons->pending[i])
        if (buttons->period[i] == 0) {
           *buttons->count[i] += 1; // Process interrupt Port 1 pin i+1
           *buttons->pending[i] = FALSE;
           buttons->period[i] = buttons->macroPeriod[i];
           P1IE |= 0x02 << i;       // Re-enable Port 1 pin i+1 interrupts
        }
        else
           buttons->period[i] -= 1;
  OSEndTask();
} /* end of LumpedButtonCounters */
```

Figure 4.8: Grouped periodic processing to minimize tasking overheads.

Event-Driven Approach

A seemingly better solution involves triggering an event-driven task each time an interrupt occurs. Such a solution introduces no tasking overheads when interrupts are not solicited, and is particularly well suited when interrupts seldom overload the system. Figure 4.9 below illustrates this approach.

```
typedef struct ButtonDescriptor {
  void (*handler)(struct ButtonDescriptor *);
  UINT16 *counter;
  void *event;
} ButtonDescriptor;
void InitButton(UINT16 *counter, UINT32 minArrivalTime)
{
  ButtonDescriptor *button;
  button = (ButtonDescriptor *)OSMalloc(sizeof(ButtonDescriptor));
  button->handler = ButtonISR;
  button->counter = counter;
  button->event = OSCreateEventDescriptor();
  OSSetISRDescriptor(OS_IO_PORT1_1,button);
  OSCreateSynchronousTask(ButtonCounter,minArrivalTime,button->event,button);
  // Initialize port P1.1 as in Figure 4.4
} /* end of InitButton */
void ButtonISR(ButtonDescriptor *button)
{
  OSScheduleSuspendedTask(ButtonCounter); // Schedule ButtonCounter
} /* end of ButtonISR */
void ButtonCounter(void *arg)
{
  static BOOL first = TRUE;
  ButtonDescriptor *button;
  if (first) {
     button = (ButtonDescriptor *)arg;
     *button->count += 1;   // Process the interrupt
     first = FALSE;         // Indicate the next phase
     OSScheduleSuspendedTask(button->event);
  }
  else {
     first = TRUE;          // Reset to the initial phase
     P1IE |= 0x02;          // Re-enable Port 1.1 interrupts
  }
  OSSuspendSynchronousTask();
} /* end of ButtonCounter */
```

Figure 4.9: Event-driven approach to interrupt processing.

Task `ButtonCounter` has 2 alternating states represented by its local remanent variable `first`. `ButtonCounter` processes the interrupt and reschedules itself when it is invoked indirectly by its associated ISR. This is the first state of the task. The purpose of the second is to re-enable the interrupt. The main idea behind this design is to allow `minArrivalTime` ticks between 2 successive invocations of `ButtonCounter`, in other words, to delay for at least `minArrivalTime` ticks before re-enabling interrupts for port 1.1.

Comparing Event-Driven and Periodic Tasks

An event-driven approach requires 2 consecutive executions of the task per raised interrupt. Hence the value of `minArrivalTime` should be half the value used with periodic tasks if an interrupt should sustain the same maximum rate. This puts an extra constraint on the schedulability of the application when DM is in effect. Indeed, event-driven tasks occupy a priority that is proportional to their execution rate, and at higher rates, these tasks have higher priority. This can undermine the schedulability of the periodic tasks. Furthermore, *ZottaOS-Hard PA* under DM cannot achieve as much power saving compared to an application with an identical load but using periodic tasks in place of event-driven ones. The reason is quite simple: a periodic task cannot put the processor at its lowest frequency and yet meet its deadline since a higher event-driven task can preempt it. Frequency settings are thus extremely pessimistic.

This is not the case when EDF is the scheduling algorithm used. Under EDF, event-driven tasks are lumped together and given a percentage of the processor utilization. Periodic tasks are therefore not affected. However the compromise is that event-driven tasks compete for the allotted utilization and scheduled event-driven tasks delay the execution of others. Under EDF, event-driven tasks are dependent and cannot be fine-tuned as for DM.

Which is Best?

If the application requirements must minimize energy at all costs, we strongly suggest designing the application with EDF in mind and to make use of periodic tasks instead of event-driven tasks. When event-driven tasks cannot be avoided, then it is best to lump them together into a single task to minimize their interdependences.

If an accurate and fine-grain timing of an event is required, DM is the reasonable choice, but energy savings will then be relegated to a secondary aim.

Continuous Interrupt Processing

The previous approach can also readily be extended to continuously count while the pushbutton is pressed. Figure 4.10 illustrates.

```
21 void ButtonCounter(void *arg)
22 {
23   static BOOL first = TRUE;
24   ButtonDescriptor *button;
25   if (first || (P1IN & 0x02) == 0)) {
26      button = (ButtonDescriptor *)arg;
27      *button->count += 1;
28      first = FALSE;
29      OSScheduleSuspendedTask(button->event);
30   }
31   else {
32      first = TRUE;          // Reset to the initial phase
33      P1IE |= 0x02;          // Re-enable Port 1.1 interrupts
34   }
35   OSSuspendSynchronousTask();
36 } /* end of ButtonCounter */
```

Figure 4.10: Continuous event-driven processing of an interrupt at a maximum rate.

The first invocation immediately following the interrupt increments the counter. The following invocations depend on the current state of the button. If the button is still pressed, an increment is done; otherwise the button interrupts are re-enabled.

Burst Interrupt Control

Some devices, such as network interfaces, are inherently bursty. It may then be desirable to handle whole bursts rather than considering only the first interrupt. The ISR of these devices should disenable itself only after a burst of interrupt requests has been observed. Bursty interrupts are characterized by two parameters: a maximum burst size and a minimum interarrival time between bursts. Our designs in the previous section can be extended to accommodate bursts. The basic idea is have a counter associated with the ISR and that counts the number of times the ISR is executed. Once this counter exceeds the burst size, the ISR doesn't re-enable itself as it would otherwise. Figure 4.11 shows the essential constituents related to the burst control of an abstract device. This implementation can easily be modified to satisfy bursts of variable length. In this case there should be a way of determining the end of a burst, for example, the last byte of an input or output message when handling a communication through a UART.

Accomplishing a burst controller with a periodic task is even simpler. In this case, the task controlling the ISR periodically would clear the number of interrupt processed by the ISR (`count` field) and re-enable it regardless if its state.

```
typedef struct BurstyDescriptor {
  void (*handler)(struct BurstyDescriptor *);
  UINT8 count;     // Number of interrupts encountered while in a burst
  UINT8 burstSize;
  void *event;
} BurstyDescriptor;
void InitBurstInterrupt(UINT8 device, UINT8 burstSize, UINT32 minArrivalTime)
{
  BurstyDescriptor *des;
  des = (BurstyDescriptor *)OSMalloc(sizeof(BurstyDescriptor));
  des->handler = BurstyISR;
  des->count = 0;
  des->burstSize = burstSize;
  des->event = OSCreateEventDescriptor();
  OSSetISRDescriptor(device,des);
  OSCreateSynchronousTask(BurstController,minArrivalTime,des->event,des);
  // Initialize device here
} /* end of InitBurstInterrupt */
void BurstyISR(BurstyDescriptor *des)
{
  // Process interrupt here
  if (++des->count == des->burstSize) {
     des->count = 0;
     OSScheduleSuspendedTask(des);
  }
  else {
     // Re-enable interrupt
  }
} /* end of BurstyISR */
void BurstController(void *arg)
{
  static BOOL first = TRUE;
  if (first) {
     first = FALSE;
     OSScheduleSuspendedTask(((BurstyDescriptor *)arg)->event);
  }
  else {
     // Re-enable interrupt
  }
  OSSuspendSynchronousTask();
} /* end of BurstController */
```

Figure 4.11: Event-driven approach to burst controlled interrupts.

Chapter 5 Inner Workings of *ZottaOS*

In essence, a task viewed by a kernel is the smallest unit of work individually schedulable. All RTOSs keep track of active tasks and allocate the processor according to some scheduling policy. In this respect, *ZottaOS* is no different. Just how *ZottaOS* knows when to step in, and how it does it, is the topic of this chapter.

Understanding how *ZottaOS* is implemented is useful in its own right but it is definitely helpful to completely apprehend the details related to power management implementations given in Chapter 6. Although this chapter focuses on *ZottaOS-Hard*, the basic operations and the order in which they are carried out do not greatly change between the different kernels of *ZottaOS*.

Run-Time Stack

Usually each task has its own run-time stack in a multitask system. This stack contains local variables of the functions called by the task, and whenever a function is called, its record is pushed onto the stack, and when the function returns, its record is popped off the stack. By having its own private stack, it is possible to execute the tasks in any order and to transfer the processor to any task. But in a priority-driven system, a task can only be preempted by a higher priority task; there is a correlation between the order in which the contexts are stored onto a run-time stack and the sequence of execution followed by the tasks. The context of an active task currently being executed by the processor is always the most recently created record, and if the tasks were in a single run-time stack, a context buried within the stack would only be reactivated once it is again at the top of the stack; in this case, the task is also the task with the highest priority. In a priority-driven system, it is therefore possible to have **a single run-time stack for a whole task set**.

The organization of the run-time stack roughly follows that of modern recursive languages; the only difference is that the some present-day microcontrollers may not have a frame pointer register necessary to implement the scheme in a transparent way and it may become necessary to emulate the missing registers by global variables.

Figure 5.1 illustrates the state of the run-time stack before and after the preemption of a task. Reversing the operation undoes the preemption and resumes a preempted task. Preemption is an operation carried out when an interruption occurs and for which the processor automatically saves some registers before jumping to an interrupt service routine; these are the program counter (PC), the status register (SR) and the stack pointer register (SP). To complete the preemption and to save the whole context of a task so that it may later resume execution, it is also necessary to store the registers of the interrupted task.

Stack Base Pointer

When an interruption ends, it is necessary to determine the memory locations of the saved registers of the preempted task. This is where another register comes into play, and which we call the **stack base pointer** (SBP) and which holds the first memory word belonging to the context of the interruption in progress. This register acts akin to a frame pointer. By positioning the SP register to this address, it becomes possible to find the top of stack experienced by the preempted task at the time of its preemption. In order to allow nested preemptions, it is also necessary to keep track of all the stack base pointers of the tasks in the stack. These can be put anywhere, but it is easier to push them onto the stack, because to handle them by means of variables would also require a stack.

Task instance creation is not more complicated. As we will later see, a task instance can be created only when another instance finishes or when the timer ISR completes its processing and schedules the next highest priority instance. In both cases, an instance creation is accomplished by overwriting the context of the creating entity. It is thus sufficient to push the initial contents of the registers of the new task starting from the base address of the creator and to simulate a return from interruption. Indirectly, the processor is given to the newly created task as if it were this task that was interrupted.

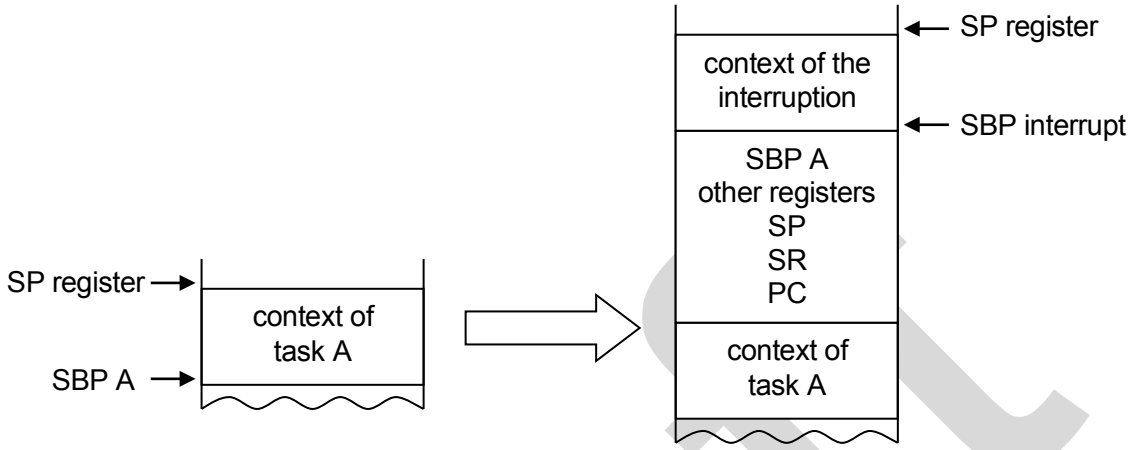


Figure 5.1: State of the run-time stack prior to and during an interruption.

Implementation Model

The kernel maintains two queues. The first queue is called the **ready queue** and holds tasks instances that are eligible for execution. These are the instances that can potentially execute on the processor once the processor becomes free. The instance at the head of this queue is the **active instance**. The second queue is the release or **arrival queue** and holds periodic tasks that are waiting for the beginning of their next period (*next release time*), at which time the task is moved to the ready queue. Tasks in the arrival queue are ordered by their release time. When a task is released, two operations are done: (1) The next release time of the task is computed and the task is reinserted into the arrival queue. (2) The task is then inserted into the ready queue according to its priority or deadline. A periodic task can therefore simultaneously be in two different queues, but these actually represent two consecutive instances of the task. The one in the ready queue is the current instance that needs to complete before its deadline, and the one in the release queue stands for the next instance to arrive. When inserting a new released instance into the ready queue, it is possible that the instance that precedes it is still present. This situation occurs when tasks overrun their deadlines, and indicates that the hard-real time task set is infeasible.

The above description applies to periodic tasks. To provide for event-driven tasks, each event is represented by a concurrent FIFO queue implemented as an array, and whose size corresponds to the number of tasks that can be triggered by the event. The array size is known once all event-driven tasks have been created, and the array is allocated by function `OSStartMultitasking`. Hence, rather than wait for their release time, event-driven tasks are held in the array associated with the event that can trigger them. The actions that are performed on event-driven tasks when they are signaled depend upon the state of the event queue and on the scheduling algorithm that is selected. When an event-driven task is signaled under DM, there are two situations to consider depending on the frequency at which the task is signaled. Recall that under DM, event-driven task are characterized by a workload interval defining the minimum time separating two consecutive executions. If the signaled task has completed its previous workload interval, the task can immediately be started. In this case, the event-driven task should be placed into the ready queue. On the other hand, if the signaled task needs to wait until its previous workload interval is reached, the arrival queue is the most appropriate place to appoint

the task. Although the above scheme may at first glance seem simple enough, there is an inherent difficulty that must be circumvented to achieve an efficient implementation: the arrival and ready queues must support concurrent accesses if we wish to avoid mutual exclusion. Rather than implementing wait-free concurrent insertion and removal operations for these queues, we choose a simpler and faster approach. All accesses to the scheduling queues are enclosed in the timer ISR, and the ISR is the only entity that can manipulate these queues. All other entities place their operations in a list and raise a timer interrupt, which when invoked, empties these operations. Going back to the implementation of event-driven tasks, there is only one operation to implement. The entity that triggers an event removes an event-driven task from the event queue and places the task in a global list before raising the timer interrupt which will complete the action. Once the timer ISR is executed, it empties the tasks from this global list and places the event-driven tasks into the arrival queue if the tasks need to wait or in the ready queue if the tasks are immediately scheduled. The other situation that can occur is a signaling entity that finds no waiting event-driven task for its event. In this case, the signaler marks the event in the event queue. When the first active event-driven task associated with the event completes its execution and relinquishes the processor, rather than inserting itself in the event queue, the task clears the event marker and inserts itself in the global list before setting off the timer interrupt.

Under EDF scheduling, the operations are similar and only the conditions affecting the placement of the task differ. An event-driven task needs to be placed in the arrival queue if its previous instance has not yet reached its deadline. Otherwise a new deadline can be determined and the task can be immediately scheduled. Like for DM, the timer ISR carries out these actions.

The basic difference between *ZottaOS-Hard* and *ZottaOS-Soft* is the organization of the ready queue and the scheduling decision of optional instances. Under DM scheduling, mandatory instances are given a priority ranging from 0 through $n-1$, where n is the total number of application tasks and where higher priority instances have lower numbers. Mandatory instances have the same priority index as for *ZottaOS-Hard*. The priority of the optional instances of a task having its mandatory instances at level i are set to $i+n$. When an instance is released, the instance is removed from the arrival queue and placed into the ready queue as for *ZottaOS-Hard*, but its running priority is first determined. No scheduling decision regarding optional instances is done at this stage. When inserting a new released instance, it is possible that the instance that precedes it is still in the ready queue. This situation occurs for optional instances that were never selected nor discarded by the scheduler. The instance can simply be removed from the ready queue before reinserting it. If an optional instance reaches the head of the ready queue, a feasibility test must be carried out to assert that the optional can indeed be dispatched. This test is done when an instance finishes its execution or at the end of the timer ISR after new tasks instances have been released. If this test is successful, the optional instance is promoted by changing its running priority to that of task's mandatory instances, so that it cannot be preempted by another optional instance once it begins its execution. On the other hand, if the test fails, the optional instance is simply dropped from the ready queue. *ZottaOS-Soft* combined with EDF is more difficult. The ready queue is conceptually structured as a two-level queue where mandatory and event-driven instances occupy the level with the highest priority, while optional instance are inserted into the 2nd queue having a lower priority. When scheduling an optional instance, the instance would then be promoted to the first queue so that it cannot be obstructed by newly released optional instances even if these have an earlier deadline. Implementing this promotion scheme without resorting to an uninterruptable critical section is quite a challenge. In *ZottaOS-Soft*, optional instances are inserted in the same ready queue but after a sentinel. This sentinel is in fact an idle task that ensures the scheduler that there is always some task to schedule (more on this later). The trick is to first mark the optional instance by changing its state and then setting the head of the ready queue to the promoted instance before removing it from the 2nd queue and removing its marker. When accessing the ready queue, any entity can check the state of the instance at the head of the queue, and if the marker is found, the entity can complete the operation.

Task States

A task instance goes through various states during its lifetime and each of these states indicates the actions to be carried out in order to execute the task. *ZottaOS* uses only 4 task states:

- Periodic tasks are in state `STATE_INIT` when they are released from the arrival queue and placed in the ready queue. In this state, the task is ready to begin its execution once it becomes the highest priority task that can execute. Event-driven tasks that have been signaled and are also ready to execute are also placed in the ready queue and set to state `STATE_INIT`.
- When a task instance accesses the processor and begins its execution, the task is set to state `STATE_RUNNING`. In this state, the context of the task should be saved when the task is preempted and restored when it resumes its execution.
- When a task finishes its execution, its state becomes `STATE_TERMINATED`, which indicates that it has completed its work. However, this is a fictitious state, as we shall later explain when we describe the workings of functions `OSEndTask` and `OSSuspendSynchronousTask`. When executing these functions, a task transits through a temporary state called `STATE_ZOMBIE` where the instance is terminated but yet still active. In this state, a task instance can either be preempted, in which case it terminates its execution and its context is not saved, or it can reach the end of the function and remove itself from the run-time stack when it installs the next task to execute on the processor. In both cases there is no need to actually change the task state from `STATE_ZOMBIE` to `STATE_TERMINATED`.

The task state bits also serve to distinguish whether the task is periodic or event-driven.

ZottaOS-Soft combined with EDF also uses an additional state called `STATE_ACTIVATE` to indicate that the task is being promoted and is switching from the low priority ready queue to the high priority queue. Tasks in this state are optional periodic instances that satisfy the feasibility test and are transiting from `STATE_INIT`. If such a task is at the head of the ready queue, its promotion is not complete and must be finalized by the entity accessing the ready queue.

Task Control Block

Like most kernels, *ZottaOS* groups all the information that it needs about a particular task into a data structure called a **task control block** (TCB). Whenever a task is created (`OSCreateTask`, `OSCreateSynchronousTask`), *ZottaOS* creates a corresponding TCB to serve as its run-time description during the lifetime of the application.

By misuse of language, we say that a task is in such-and-such queue, but in fact, tasks do not move. The queues are linked lists of TCBs and to move a task amounts to changing the link assignments of the TCBs. The information contained in a TCB is given on Figures 5.2 through 5.5.

```
typedef struct TCB {
  struct TCB *Next[2];          // Next TCB in the list where this task is located
  UINT8 TaskState;              // Current state of the task
  void (*TaskCodePtr)(void *); // Pointer to the first instruction of the task
  void *argument;               // An instance specific 32-bit value (user input)
  UINT8 Priority;               // Static task priority assignment
  INT32 PeriodLow;              // Inter-arrival period
  UINT16 PeriodHigh;            // Number of full 2^30 cycles of the inter-arrival period
  INT32 NextArrivalTimeLow;     // Remaining time before reappearing
  UINT16 NextArrivalTimeHigh;  // Number of full 2^30 cycles for the next arrival time
} TCB;
```

Figure 5.2: TCB contents of periodic tasks under RM or DM scheduling with *ZottaOS-Hard*.

The TCB of a periodic task is given on Figure 5.2. The first field of the TCB is used to link the TCB in the ready and arrival queues. Two pointers are set aside, because periodic tasks reside in both queues at the same time. Note that even if the implementation would prepare the next release only when an instance completes its execution, the task would also require 2 pointers. This is because to carry out this operation, the finishing instance would have to insert its TCB in the arrival queue before removing the TCB from the ready queue. These operations must be done in this order, because if the instance initially removes itself from the ready queue and an interruption occurs, its TCB no longer becomes accessible and the task disappears forever. The 2nd field stores the state of the task. The 3rd field indicates the code of the task and corresponds to the address of the first instruction to execute when an instance starts its execution. The task's argument comes next and has the size of a pointer. This field acts as an intermediate storage to hold the corresponding value transmitted by function `OSCreateTask` to its instances and through which these instances can access it by referencing their argument. The next field is the task's running priority that is used under RM or DM scheduling to sort the tasks in the ready queue. This priority is determined once all deadlines of the task set are known and becomes a constant as soon as the kernel is launched by function `OSStartMultitasking`. It is interesting to note that the deadline parameter of function `OSCreateTask` is not stored in the TCB of the created tasks. Indeed, under RM or DM scheduling, these deadlines are sorted and then mapped into priority indices before being discarded. The two following fields hold the period of the task decomposed as two stored values. The first does not exceed 2^{30} timer ticks while the second represents the number of full 2^{30} tick-cycles. Together these 2 fields allow the period to be represented on 2^{46} bits. These two values correspond to the period parameters of function `OSCreateTask`. The last 2 fields hold the release time of the next instance of the task and constitute the sorting key of the arrival queue. The first instance of each task is released at time 0, and at each release, the period of the task is added to the arrival-time fields in order to prepare for the next instance of the task. Every time that the system clock is greater or equal to 2^{30}, all arrival-time fields are shifted by 2^{30}: if the `NextArrivalTimeHigh` field is equal to 0, the `NextArrivalTimeLow` field is shifted, otherwise `NextArrivalTimeHigh` is decremented.

The definition of the TCB for periodic tasks under EDF scheduling is given in Figure 5.3. The first 8 fields of the TCB are identical to those for RM or DM except for the priority field, which is inexistent under EDF. Instead, the task deadlines need be explicitly handled at each task release time. The related fields are given in the last 2 fields of the TCB. Field `Deadline` corresponds to the relative deadline of the task and it is provided as a parameter at the creation of the task. The last field, `NextDeadline`, represents the actual deadline of an instance and it is obtained by adding the release time of the instance, `NextArrivalTimeLow`, to the relative deadline of the task. This field also becomes the insertion criterion in the ready queue.

```
typedef struct TCB {
  struct TCB *Next[2];          // Next TCB in the list where this task is located
  UINT8 TaskState;              // Current state of the task
  void (*TaskCodePtr)(void *); // Pointer to the first instruction of the task
  void *argument;               // An instance specific 32-bit value (user input)
  INT32 PeriodLow;              // Inter-arrival period
  UINT16 PeriodHigh;            // Number of full 2^30 cycles of the inter-arrival period
  INT32 NextArrivalTimeLow;     // Remaining time before reappearing
  UINT16 NextArrivalTimeHigh;  // Number of full 2^30 cycles for the next arrival time
  INT32 Deadline;               // Task's deadline <= Period (user input)
  INT32 NextDeadline;           // Current instance deadline
} TCB;
```

Figure 5.3: TCB contents of periodic tasks under EDF scheduling with *ZottaOS-Hard*.

ETCB

The TCBs of event-driven tasks contain less information than their periodic counterparts, and much of their content serves the same purpose. To distinguish these TBCs, the data structure holding the information about a specific aperiodic task is called an **extended task control block** (ETCB) in *ZottaOS*. Figure 5.4 displays the content of an ETCB under RM or DM scheduling. The first 6 fields are exactly identical to a TCB. Recall that an event-driven task has a relative deadline equal to its period. As explained in Chapter 2, an event-driven task behaves like an or-

dinary periodic task with the exception that the task can be immediately schedulable if at least one task-period separates 2 consecutives executions of the task. Otherwise, the event-driven instance waits in the arrival queue until the beginning of its period, at which time it is released. Once scheduled, the release time of the next instance is determined and stored in the `NextArrivalTimeLow` field of the task before the task is inserted into the ready queue. This field allows the scheduler to decide whether the next instance has to wait or not before it can be inserted into the ready queue.

```
typedef struct ETCB {
  struct TCB *Next[2];          // Next TCB in the list where this task is located
  UINT8 TaskState;              // Current state of the task
  void (*TaskCodePtr)(void *);  // Pointer to the first instruction of the task
  void *argument;               // An instance specific 32-bit value (user input)
  UINT8 Priority;               // Static task priority
  INT32 PeriodLow;              // Minimum inter-arrival period
  INT32 NextArrivalTimeLow;     // Remaining time before instance can reappearing
  struct FIFOQUEUE *EventQueue; // Pointer to the event queue of this task
  struct ETCB *NextETCB;        // Link to the next created ETCB
} ETCB;
```

Figure 5.4: ETCB contents of event-driven tasks under RM or DM scheduling with *ZottaOS-Hard*.

The last two fields of the ETCB are particular. An event is associated with a queue of suspended event-driven tasks, and once a task completes its execution it joins the queue indicated by field `EventQueue`. As mentioned for TCBs, once the system clock exceeds $2^{30} - 1$, all temporal values stored in the ETCB (`NextArrivalTimeLow`) need to be shifted, but since the ETCBs are scattered throughout the event queues created for the application, this operation turns out to be time consuming. The ETCBs are therefore grouped together into a list to facilitate this operation, and the field `NextETCB` allows the ETCBs be linked together.

When an event is triggered and an event-driven task is removed from the event queue, or when an event-driven task finishes its current execution only to realize that another event must be dealt with, the ETCB of the task is temporarily placed in a global LIFO queue until the timer ISR is invoked. This queue is a list of unsorted ETCBs that are linked together with the same pointer linking the tasks in the arrival queue. When the timer ISR is executed, it empties the global queue and places the ETCBs either in the arrival queue if the associated task needs to wait or directly in the ready queue if the task can immediately start.

Finally, the different fields of the ETCBs for EDF scheduling are given in Figure 5.5. As explained in Chapter 2, event-driven tasks are scheduled according to the total bandwidth server method, where the deadline of a task depends on the total workload of each event-driven task composing the application. When an event-driven task is inserted into the ready queue, its deadline (field `NextDeadline`) is computed by adding its `Workload` field to the next earliest possible start time of the task, and which is equal to the current time or to the deadline of the last scheduled event-driven task (application of Eq. 2.7 in Chapter 2). The `Workload` field is in essence the equivalent of the relative deadline of periodic tasks but is it used differently. The re

```
typedef struct ETCB {
    struct TCB *Next[2];          // Next TCB in the list where this task is located
    UINT8 TaskState;              // Current state of the task
    void (*TaskCodePtr)(void *);  // Pointer to the first instruction of the task
    void *argument;               // An instance specific 32-bit value (user input)
    INT32 WorkLoad;               // Equal to WCET / (available processor load)
    INT32 NextArrivalTimeLow;     // Remaining time before reappearing
    INT32 NextDeadline;           // Priority criterion for EDF scheduling
    struct FIFOQUEUE *EventQueue; // Pointer to the event queue of this task
    struct ETCB *NextETCB;        // Link to the next created ETCB
  } ETCB;
```

Figure 5.5: ETCB contents of event-driven tasks under EDF scheduling with *ZottaOS-Hard*.

maining fields in the ETCB are functionally equivalent to those under RM or DM, or to those of a TCB under EDF. Namely, the last two fields serve the same purpose as those mentioned for RM or DM scheduling.

Before concluding this section, we summarize the memory occupancy of the task control blocks in Table 5.1. An address is assumed to occupy 2 bytes and a memory block is allocated on even addresses (blocks are multiple of 16 bits). The numbers in Table 5.1 include the overheads due to head and tail sentinels that are part of the arrival and ready queues.

Task Type	RM or DM [bytes]	EDF [bytes]
Periodic	$16 + 22 \times N_p$	$16 + 30 \times N_p$
Event-Driven	$22 \times N_a$	$26 \times N_a$

Table 5.1: TCB size in bytes with 2 byte-addresses including arrival and ready queues overheads as a function of the number of application tasks for *ZottaOS-Hard* (N_p = periodic tasks, N_a = event-driven tasks)

Task Scheduling

In addition to the arrival and ready queues and run-time stack, the scheduler also relies on a timer to maintain the current time and to manage the release times of the periodic tasks. A timer is simply a counter that increments or decrements at a constant rate. One very common timer is the programmable interval timer (PIT), which counts from some specified value down to zero, and raises a processor interrupt once it reaches zero. A PIT may also count upwards until a specified value and then generate an interrupt. Most widespread microcontrollers have timers that operate differently. Typically a timer is composed of a register, which contains the current count, and another register, which can be used to make comparisons. This latter register can be:

- the starting value of the counter and a timer interrupt is then generated when the counter reaches zero (downward count) or when the counter overflows (upward count);
- a target value and a timer interrupt is raised when the counter is equal to the comparator.

Timer Requirements

To be useful by a RTOS scheduler, a timer should have the following minimal functionalities:

1. When the timer raises an interrupt, it must continue counting. In other words, the timer should have a cyclic behavior. This characteristic is essential to maintain a system clock.
2. The comparator must be writable at all times. This point is not provided on all architectures, for example on the Blackfin embedded processor from Analog Devices, a modification of the comparator is taken into account only when the timer reaches its previous value. On these architectures, it is nevertheless possible to set the period of the timer once and for all with the greatest common divisor between the periods of the periodic tasks. However there is will have many useless interruptions.
3. The current value of the counter register must be readable at all times if the actual time must be known.

In the following, we assume that the timer of the microcontroller satisfies all the above characteristics.

Task scheduling can take place at two distinct instants: when a task is released and when a task instance completes its execution. The first case is handled by the timer interrupt service routine (ISR) while the second by explicit function invocations when the tasks reach the end of their processing.

To describe the actions carried out on the data structures representing the priority queues used to schedule the tasks, we introduce the following notations:

- `Head(Q)` refers to the TCB or ETCB which is at the head of queue `Q`; the TCB or ETCB is not withdrawn from `Q` and the queue remains untouched.
- `Dequeue(Q)` removes and returns the TCB or ETCB at the head of queue `Q`.
- `Insert(Q,tcb)` inserts the TCB or ETCB, `tcb`, into queue `Q` according to the type of the queue; this can be by ascending order of the release times when inserting into the arrival queue or according to the priority of the task indicated by the TCB or ETCB when insertion is done in the ready queue.

With these 3 abstract operations, we can give the pseudocode of the essential components of the scheduler.

Although it would appear to the reader that the logical sequence to explain how and when the tasks are actually scheduled would be to begin with the timer ISR. It turns out that task instance terminations require actions that can overlap with the timer ISR and to understand these actions, it is easier to first reveal the functions called by task instances when they reach the end of their execution.

Function `OSEndTask`

When a periodic task instance finishes its execution, it must relinquish the processor to another task; however it doesn't need to prepare its next release as this is done by the timer interrupt when tasks are released. It may seem odd, at first glance, that a released periodic task reinserts itself into the arrival queue in addition to the ready queue, placing itself into 2 different queue at the same time. Assuming that a finishing task also prepares its next arrival would mean that it would also have to fix the timer interrupt if its release time is the earliest. All these operations would be complex if we wish to accomplish these without inhibiting interrupts, the arrival queue would also have to allow concurrent accesses. By centralizing all accesses to the arrival queue in the timer ISR, the operations become easier.

The first steps of function `OSEndTask`, illustrated on Figure 5.6, are to indicate that the active task is in a transitory state where any task or event can complete its operations. Indeed, once a task removes itself from the ready queue, the system state becomes incoherent.

```
step 1   Head(ReadyQ).TaskState ← STATE_ZOMBIE
step 2   _OSNoSaveContext ← true
step 3   Dequeue(ReadyQ)
step 4   if Head(ReadyQ).TaskState = STATE_RUNNING then
             Install new task instance
         else
             Restore context of Head(ReadyQ)
```

Figure 5.6: Execution steps of `OSEndTask`.

To recover from this state of affairs, we use the following set of rules:

1. If an interrupt other than for the timer ISR occurs, the context of the task instance is always saved and restored once the ISR returns. Even though the instance may no longer be in the ready queue, it still can resume its execution and set up the timer comparator if necessary. These ISRs never examine the scheduling queues and always resume the task or the ISR they have preempted.
2. If `Head(ReadyQ).TaskState = STATE_ZOMBIE`, the active task is in the midst of removing itself from the ready queue. If the timer ISR preempts the zombie task, it can finish the actions undertook by the task and since a new task instance will be scheduled (step 6), there is no point in saving the context of the zombie.
3. If `Head(ReadyQ).TaskState` ≠ `STATE_ZOMBIE`, either the active task is before step 2 or after step 5. To distinguish between these 2 locations, an additional global variable,

`_OSNoSaveContext`, is introduced. When *true*, the timer ISR should not save the context of the active task it preempts.

Hence if the state of the instance is `STATE_ZOMBIE` or if `_OSNoSaveContext` is set, the timer ISR does not save the context of the active instance.

Finally, if step 4 is reached, the task relinquishes the processor to the next highest priority task instance in the ready queue. This can be either a new task instance that needs to be installed on the run-time stack (`Head(ReadyQ).TaskState` = `STATE_INIT`) or a preempted task having a smaller priority compared to the terminating zombie instance (`Head(ReadyQ).TaskState` ≠ `STATE_INIT`) and for which the context was saved on the run-time stack when the zombie was scheduled by the timer.

Idle Task

In order to ensure that there is always a task to schedule (step 4), *ZottaOS* has a particular task called **idle**. As its name implies, this task does nothing other than enter an infinite loop, but on processors having sleep modes, the idle task forces the processor to enter the deepest sleep mode that still leaves the internal oscillator and timer peripheral active. Depending on the architecture at hand, this sleep mode can be part of the status register or be a special and dedicated register; however and regardless of the architecture, the sleep mode is part of the context of a task and it must be applied when context switching to that task. When an interruption is raised, the processor should also leave its current sleep mode and enter an active state in order to process the interrupt, saving the sleep mode for when the idle task is rescheduled.

Functions `OSSuspendSynchronousTask` and `OSScheduleSuspendedTask`

Function `OSSuspendSynchronousTask` is invoked when an event-driven task completes its execution and contrasts with `OSEndTask` in that the instance joins a FIFO queue associated with the event that triggered its release. The course of actions taken by a terminating instance differs whether the instance needs to waits for the next event or whether the event is already raised. In the first case, the task joins the FIFO queue of the event. In the first step of `OSSuspendSynchronousTask`, given in Figure 5.7, function `EnqueueEventTask` inserts the ETCB of the event-driven task into the FIFO queue and returns *false*.

As it is possible that at the time at which `OSSuspendSynchronousTask` is called, an event has again been posted by a signaler task and for which there were no awaiting tasks at the time the event was raised, the event-driven task needs to reschedule itself. In this case, function `EnqueueEventTask` clears the transmitted signal and returns *true*. The instance then joins a temporary list that is emptied by the timer ISR and raises a timer interrupt by setting the timer's comparator to its current counter value. The remaining 4 steps of `OSSuspendSynchronousTask` act like those of `OSEndTask`.

```
step 1   if EnqueueEventTask(event,Head(ReadyQ)) then
             EnqueueRescheduleQueue(Head(ReadyQ))
step 2   Head(ReadyQ).TaskState ← STATE_ZOMBIE
step 3   _OSNoSaveContext ← true
step 4   Dequeue(ReadyQ)
step 5   if Head(ReadyQ).TaskState = STATE_RUNNING then
             Restore context of Head(ReadyQ)
         else
             Install new task instance
```

Figure 5.7: Execution steps of `OSSuspendSynchronousTask`.

The last function we wish to expound before explaining the timer functions is `OSScheduleSuspendedTask` and which is invoked to raise an event. This function can be called from any task (even an event-driven task associated with the event) or from an application defined ISR. This function is given in Figure 5.8.

```
step 1   etcb ← DequeueEventTask(event)
step 2   if etcb ≠ null then
            EnqueueRescheduleQueue(etcb)
```

Figure 5.8: Execution steps of `OSScheduleSuspendedTask`.

Function `OSScheduleSuspendedTask` consists of 2 steps where the first dequeues the first aperiodic task in the FIFO queue of the event. Function `DequeueEventTask` does one of two things. When the FIFO queue is empty, the function marks (signals) the fact that an event occurred and returns *null*. The first event-driven instance to terminate clears the signal and reschedules itself when it calls `EnqueueEventTask`.

If the FIFO queue is not empty when invoking `DequeueEventTask`, the dequeued ETCB is returned to the caller so that it may instruct the timer to schedule the event-driven task. This, like for `OSSuspendSynchronousTask`, is done via function `EnqueueRescheduleQueue`.

Timer ISR

One design goal of *ZottaOS* is to allow for fast interrupt processing of peripheral devices, and in order to achieve this, the actions that may interrupt the handling of a peripheral must be limited. One of these actions is the timer ISR, which usually requires a considerable amount of work. One scheme that allows the timer to keep track of the time and yet not interrupt a peripheral handler is by dividing the timer handler into 2 parts. The first part simply backs up the necessary information needed to update the system time, resets the timer so that it continues to count and then propagates the interrupt to another handler, which processes the interruption per se and which requires the largest part of the timer's work. As long as the second part is done when there are no other pending interrupts and peripherals can interrupt its execution any time. Peripheral processing is ensured to have a fast response time.

System Clock

Figure 5.9 details the steps done by the timer ISR, where we assume that the timer peripheral counts from zero until it reaches a comparator value and then continues counting from zero. Step 1 saves the current comparator value so that the system clock can later be updated. Step 2 sets the next interrupt of the timer to occur when the counter overflows, leaving more than enough time to process the entire interrupt. Indeed, it is imperative that the counter register within the timer does not reach a value that triggers a new interrupt before the previous interrupt succeeds in updating the system clock; otherwise the system clock misses some updates and is sure to lag behind. The system clock, T_{sys}, is then updated in step 4. The **system clock** represents a relative reference of time (in timer ticks) and that is used by `_OSTimerInterrupt-Handler` to compare task arrival times and to set the deadline of event-driven tasks. To obtain the actual relative time at any given moment of time, the counter register of the timer peripheral must be read and added to T_{sys}. Step 5 propagates the interrupt to another source that is enabled when there are no pending interrupts to process. And finally step 6 returns from the timer ISR.

```
step 1   Save comparator value
step 2   Set comparator to INFINITY16
step 3   Re-enable interrupts
step 4   Update the system T_sys
step 5   Propagate interrupt to a low-priority source to finish the interrupt
step 6   Return from interrupt
```

Figure 5.9: Timer ISR propagating the interrupt.

The timer ISR does not save the context of the interrupted task and on the MSP430 family of microcontrollers the ISR requires a total of 38 instruction cycles, of which only 16 cycles are done while interrupts are inhibited.

The second part of the timer handler, called `_OSTimerInterruptHandler`, is given in Figure 5.10. This ISR is attached to a software interrupt, and for microcontrollers that do no have this feature, such as the MSP430, the ISR is bound to an I/O port, and while the 9 steps of this ISR is under way its interrupts are masked. There can therefore be no nested calls.

```
step 1  if Head(ReadyQ).TaskState ≠ STATE_ZOMBIE then
           if _OSNoSaveContext then
              SP ← SBP
         else
            Save context of interrupted task
        else
           Dequeue(ReadyQ)
           SP ← SBP
step 2  Mark that a software timer interrupt is in progress
step 3  tcb ← Head(ArrivalQ)
        while tcb.NextArrivalTimeLow ≤ T_sys and
                 (not IsPeriodicTask(tcb) or tcb.NextArrivalTimeHigh = 0) do
           tcb ← Dequeue(ArrivalQ)
           tcb.TaskState ← STATE_INIT
           if IsPeriodicTask(tcb) then
              tcb.NextDeadline ← tcb.NextArrivalTimeLow + tcb.Deadline (only EDF)
              tcb.(NextArrivalTimeLow,NextArrivalTimeHigh) ←
                              tcb.(NextArrivalTimeLow,NextArrivalTimeHigh) +
                              tcb.(PeriodLow,PeriodHigh)
              Insert(ArrivalQ,tcb)
           else
              etcb.NextDeadline ← Eq. 2.7 (only EDF)
              etcb.NextArrivalTimeLow ←
                           etcb.NextArrivalTimeLow + etcb.PeriodLow (only RM or DM)
           Insert(ReadyQ,tcb)
           tcb ← Head(ArrivalQ)
        end while
step 4  if T_sys ≥ 2^30 then
           Shift temporal variables
step 5  Process backlogged aperiodic tasks
step 6  _OSNoSaveContext ← true
step 7  Re-enable this interrupt
step 8  newTimerComparator ← Head(ArrivalQ).ArrivalTime – T_sys
        if newTimerComparator ≤ timer counter then
           Set timer comparator to timer counter
        else if newTimerComparator < INFINITY16 then
           Set timer comparator to newTimerComparator
step 9  if Head(ReadyQ).TaskState = STATE_RUNNING then
           Restore context of Head(ReadyQ)
        else
           Install new task instance
```

Figure 5.10: Execution steps of `_OSTimerInterruptHandler`.

As mentioned in `OSEndTask` and in `OSSuspendSynchronousTask`, an instance that completes its execution needs to remove itself from the ready queue. In doing so, the task first sets its state to `STATE_ZOMBIE`. If the timer interrupts this instance and detects an active zombie task, `_OSTimerInterruptHandler` finalizes the removal of the zombie. However if there is no active zombie, but `_OSNoSaveContext` is set, this means that the zombie successfully removed itself from the ready queue and has not yet scheduled the next active task. In both of these cases, the context of the interrupted task instance must not be saved. Setting the stack pointer register (SP) to base address of the interrupted task (SBP) completely eliminates the task.

Step 2 sets a flag to indicate that a timer interrupt is being processed and no interrupt should be raised. This step needs some clarifications. When an event-driven task is released by function `OSScheduleSuspendedTask`, the task may first have to be postponed in the arrival queue until it can become ready or it can also be immediately scheduled and placed in the ready queue. In both cases a timer interrupt needs to be raised so that one of these actions can be carried out. These actions are done at step 5 and if an entity signals an event-driven task before this step is reached, the event-driven task will not go unnoticed by `_OSTimerInterruptHandler`, and there is no point in raising an interrupt. Releasing an event-driven task is the only circumstance where an explicit timer interrupt is raised.

At step 3, all the tasks that have reached their release time are transferred from the arrival queue into the ready queue. For periodic tasks, this implies preparing the next task arrival and setting the deadline of the released instance if EDF is the scheduling algorithm. Note that the arrival times are 46-bit quantities. Event-driven tasks require less preparations and time values are on 32 bits. Step 4 prevents wraparound of the system clock and shifts all temporal variables by 2^{30} as soon as T_{sys} exceeds $2^{30}-1$. Note that with this shift, all temporal variables preserve their relative orderings. The temporary LIFO list of event-driven tasks that are to be scheduled is next emptied at step 5. Recall that when an event-driven task needs to be scheduled (function `EnqueueRescheduleQueue`), the caller first places the ETCB of the task in a list and then raises a timer interrupt if none is in progress. The dequeued ETCBs are either placed in the ready queue or in the arrival queue if they cannot yet be scheduled. Once the last ETCB has been dequeued, timer interrupts are again allowed, and `_OSTimerInterruptHandler` may be interrupted. Step 6 sets `_OSNoSaveContext` so that if a timer interrupt is raised immediately after step 7 and `_OSTimerInterruptHandler` is invoked, the current call is expelled from the stack during the first step of the new call.

Step 8 prepares the timer peripheral to trigger for the next release time. There are 2 special cases to consider before setting the timer's new comparator value. First, `_OSTimer-InterruptHandler` could have missed the release time, for example if another interrupt preempts it and delays the handler by more than one tick. In this case, the timer is set to trigger immediately. Second, the next release time may occur after a wraparound of the timer's counter. Recall that the timer ISR (first part) sets the timer counter to its maximum value before propagating the interrupt and which `_OSTimerInterruptHandler` handles. Hence the timer is already adjusted. When neither case applies, the comparator is set to the remaining ticks needed to reach the earliest release time.

Finally step 9 finalizes the execution of the interruption and relinquishes the processor to the next highest priority task instance in the ready queue.

Summary

It is common practice to implement an RTOS by temporarily disabling hardware interrupts when the kernel takes over or when application tasks share data. While interrupts are disabled, the executing task is ensured to continue uninterrupted, so no interleaving may occur. Since disabling and enabling interrupts cost only 2 instructions, its use is widespread. For example it is customary to disable interrupts to protect kernel data structures and prevent context switches while scheduling tasks. Despite the cost advantage of disabling interrupts, it has a severe limitation: interrupts cannot remain disabled for long; otherwise peripheral interrupts may be lost or have lengthy latencies.

In the design of *ZottaOS*, we have taken great care to considerably limit the duration where the kernel disables and enables interrupts. In fact, even when a microcontroller does not support any atomic instruction, such as the MSP430, the longest period of time where interrupts are disabled takes 29 machine cycles. The design of the kernel can however become intricate as this chapter demonstrates.

Chapter
6 Power-Awareness

As battery-operated embedded systems proliferate, energy efficiency is becoming an increasingly critical consideration in system design. Moreover, embedded systems are usually designed to provide high peak performance when needed. Opportunities thus naturally arise to reduce power consumption by dynamically varying the performance requirements according to workload variation. A **power-aware** embedded system is one that is able to provide the right power at the right place and at the right time.

Run-time power reduction mechanisms involve shutting down the processing unit and its peripherals when they become idle and scaling the processor's supply voltage and operating frequencies (usually referred as dynamic voltage/frequency scaling (DVFS)). Using these power-saving mechanisms inevitably reduces system performance, and as most real-time applications must satisfy timing requirements, the decreased computing performance should not cause deadline violations. Power-aware kernels like *ZottaOS-Hard PA*, the power-aware version of *ZottaOS-Hard*, thus aim to exploit the benefit of power manageable resources while meeting the requirements of the real-time application.

Background

The key to reduce the energy consumption of an embedded system is to devise and only use electronic components specifically designed to consume minimal energy while meeting the requirements of the design. This is an obvious approach used by embedded hardware designers. Further energy reduction can be achieved by powering down unused peripherals units within or controlled by the microcontroller at hand. The next step in energy reduction consists in optimizing the code of the software that runs on the embedded system. The idea behind this technique is to finish the necessary computation earlier and to go into one of the processor's sleep modes as soon as possible. These approaches have become common practice among embedded system designers.

ZottaOS-Hard PA and *ZottaOS-Soft PA* take a further step towards energy reduction by incorporating dynamic frequency scaling scheduling. Simply put, dynamic frequency scaling works by varying the processor's supply voltage and frequency at run-time to match the workload and time requirements of the application.

The workload of a processor is not constant; it is made of processing and idle periods. During the idle periods, the processor can be put to sleep, but as we shall shortly see, it is more advantageous to lengthen the processing periods to the detriment of idle periods by adapting the processing speed of the processor, and thus its power consumption, with the current workload of the processor while acting on the supply voltage applied to the processor.

To understand how there can be an energy profit by controlling the speed of the processor, we must first understand the technological factors related to the energy consumption of electronic components. For CMOS circuits, power consumption includes dynamic power and leakage power. Dynamic power is due to the switching activities of the transistors, while leakage power is consumed even when no logic operations are performed. For a given integration technology, the dynamic power dissipated (P_{dyn}) by a CMOS circuit is proportional to a technological factor (which takes into account the load capacitance and the switching activity), to the system clock or operating frequency (f), and to the supply voltage (V) roughly squared. By considering only the supply voltage and the operating frequency, we have the following relationship:

$$P_{dyn} \propto f \cdot V^2$$

Supply voltage is thus the term for which a reduction will have the most impact on dynamic power. In the course of time, supply voltages did not cease to decrease. The most widespread value today is 3.3 V, but certain processors achieve even lower values. For example, Silicon Labs' C8051F931 microcontroller can operate at 0.9 V. However this reduction in supply voltage encounters increasing problems with the laws of physics. At small levels, a further lowering of the voltage has as consequence that the leakage currents, and hence the leakage power, become the dominant component influencing the overall power consumption. Also, the lowering of the voltage supply decreases the signal-to-noise ratio and makes the microcontroller more sensitive to external disturbances. The key observation for which energy reduction can be made is related to the fact that a processor that can operate on a range of frequencies requires higher voltages for higher operating frequencies, and lower voltages for lower frequencies. By lowering the supply voltage, and consequently the operating frequency, it is possible to achieve a significant energy saving. However, for this technique to be of any practical use on small low-cost embedded systems, the microcontroller must have a programmable core voltage regulator so that the supply voltage applied to the CPU can be controlled and there must also be a way to independently adjust the operating frequency of the CPU.

Some members of the MSP430F5xx and all members CC430F5xx and CC430F6xx from Texas Instruments are the first ultralow power microcontroller to meet these requirements. *ZottaOS-Hard PA* and *ZottaOS-Soft PA* are specifically intended for these microcontrollers.

Power-Awareness Scheduling

Depending on when the power-aware decisions are made, power-aware scheduling can be classified as either **offline** or **online**. Offline techniques are applied at design time and consider the timing specifications by taking worst-case scenarios and assume that each task will take their worst-case execution time.

One disadvantage of this strategy is that the worst-case execution times can be much longer than the average-case execution time and result into a pessimistic task schedule that can severely limit possible energy savings. To further improve energy efficiency, offline decisions can be combined with an online technique resulting into some hybrid approach.

Online scheduling

Online scheduling techniques are adaptive to the current workload and take into account the non-predictable actual execution times available only at run-time to guide scheduling decisions. As most embedded systems are highly dynamic in nature, an online approach can lead to significant energy savings provided that the computation complexity is kept low. Otherwise the technique may lead into an infeasible schedule or severely compromise the energy efficiency.

To see how power-aware scheduling technique can save energy, we shall illustrate each scheme implemented in the power-aware versions of *ZottaOS* with a common example of 3 periodic tasks having the characteristics shown in Table 6.1. Each task is characterized by its worst-case execution time (WCET), its period (P) and deadline (D). These 3 attributes are known or can be determined at design time. The last column of the table holds the **actual execution time** (AET) of each task that will only be known when the task completes its instance.

Task	WCET	P = D	AET
1	240	400	100
2	120	600	120
3	120	1200	100

Table 6.1: Task set example in ticks of 32.768 kHz

For these examples, we assume that the processor has 3 frequency settings defined for 12, 16 and 20 MHz.

Figure 6.1a shows the predicted scheduling times of the tasks without any power management during a hyper-period of 1200 ticks. One common strategy is to run the tasks at their highest speed and to put the processor into a sleep mode when it becomes idle. Figure 6.1b shows the resulting idle period for the given task set.

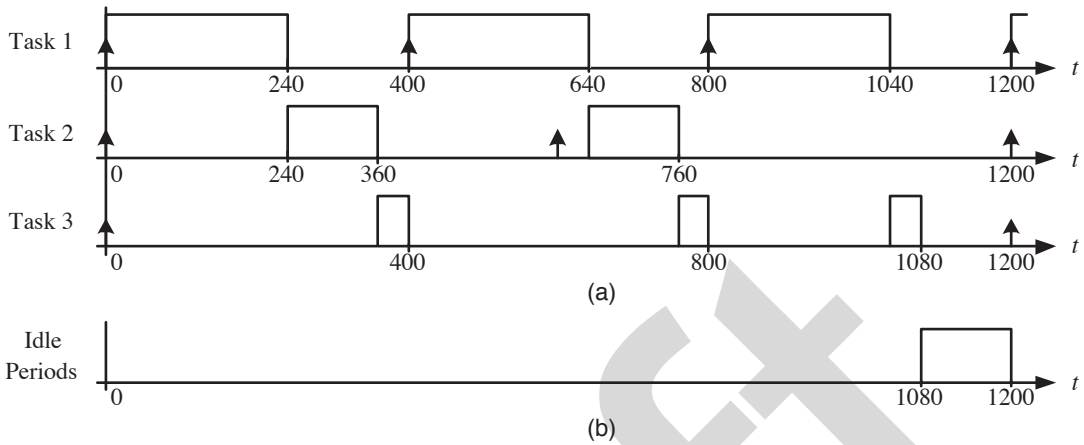


Figure 6.1: Scheduling diagram of the task set of Table 6.1.
(Upward arrows represent task instance releases)

For Your Information

Power mode settings for *ZottaOS* are done via symbolic constant `POWER_MANAGEMENT` defined in *ZottaOSHardPA.h*. There are 5 possible settings of which `POWER_MANAGEMENT` assigned to `NONE` runs each task at the highest speed and puts the processor to sleep when it becomes idle.

Static Frequencies

A better strategy is to compute the offline **slowdowns**, i.e., the reduced frequency settings for the different tasks divided by the maximum operating frequency.

Task	f [MHz]	*slowdown*
1	20	1
2	16	0.8
3	16	0.8

Figure 6.2 shows the resulting scheduling of the task set. As expected, the processor idle periods are greatly reduced. The processor is now idle for 30 ticks instead of 120 ticks.

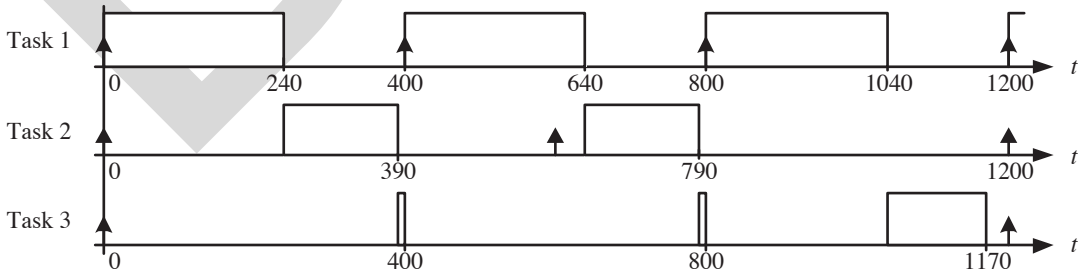


Figure 6.2: Scheduling diagram of the task set of Table 6.1 with static frequencies.

For Your Information

Computing the optimal speeds of the tasks is complex when task deadlines and task periods are different or when RM or DM scheduling is in effect. The *ZottaOS* distribution package includes a program to compute the static frequencies for a given task set under all supported scheduling algorithms.

Online Strategies

Slack

As mentioned previously, online strategies can realize even greater energy savings because task instances typically require fewer processing cycles than indicated by their specified worst-case execution times. Since actual execution times (AET) are only known after the instance has executed, the difference between WCET and AET, also called **slack** or **excess**, can be given to another task instance without changing the behavior of the real-time scheduling algorithms. On-line strategies vary in the way the slacks are distributed among the task instances. Figure 6.3 shows the resulting scheduling diagram when the actual execution times given in Table 6.1 are carried over onto the scheduling diagram of Figure 6.2. As can be seen from Figure 6.3b, a fair amount of idle periods is introduced at run-time and there are slacks that are not exploited.

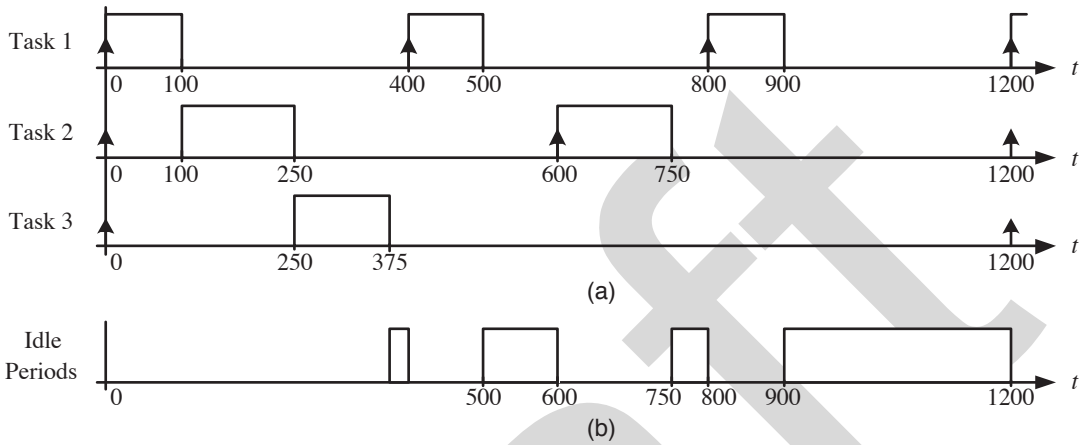


Figure 6.3: Actual execution time of task set of Table 6.1 with static frequencies.

Specifically at tick 400 when the second instance of Task 1 is released, the scheduler knows that there can be no event before tick 600 and therefore that instance can be brought to a lower operating frequency whereby the core voltage can be lowered in order to make additional energy earnings. The 3rd instance of Task 1 can also benefit from an online strategy since the scheduler can readily determine that at tick 800, the instance has until tick 1200 to complete its execution.

One-Time-Extension (OTE)

The easiest algorithm distributes the accumulated slack to the last task instance in the ready queue; the processing time of this task can be extended to its deadline or to the next arrival time of a task instance without changing the behavior of the real-time scheduling algorithms [SHI99]. The OTE power management technique is independent of the scheduling algorithm and can be used with RM, DM or EDF scheduling algorithms.

Figure 6.4 shows the scheduling diagram of our example task set. The numbers inserted in the rectangle representing the duration of the tasks are the slowdowns that are applied to the tasks. At tick 100, the first instance of Task 1 finishes and leaves a slack of 140 ticks (240-100). The 2nd instance of Task 2 is then scheduled with its static frequency and finishes at tick 250 at which time Task 3 is scheduled. Notwithstanding the fact that this task is the last task in the ready queue, its WCET is 120 and the time that it has to execute its work before its deadline or the next arrival is 150 ticks. It therefore runs with a slowdown of 0.8 and finishes at tick 375. At tick 400, a new instance of Task 1 is released and it is the only ready task. Because the next arrival will occur at 600, it cannot benefit from the slack left behind the other tasks because its WCET exceeds the time it has until the next task release. At tick 600 a new instance of Task 2 is released and the next predicted event occurs at tick 800 to complete its work. It can therefore reduce its speed to $120 / (800 - 600) \times 20 = 12$ MHz and run with a slowdown of 0.6 rather than with its static slowdown of 0.8. Finally the 3rd instance of Task 1 is released at tick 800 and has 400 available ticks to accomplish its 240 ticks of work and it too can run at 12 MHz.

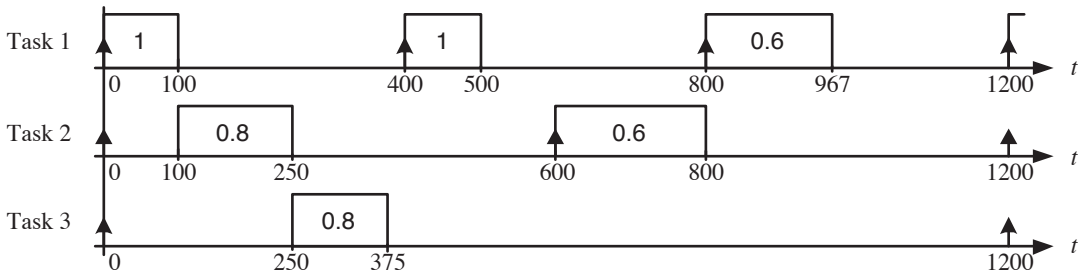


Figure 6.4: Scheduling diagram of task set of Table 6.1 with static frequencies and OTE.

For Your Information

To benefit from OTE, the symbolic constant `POWER_MANAGEMENT` defined in *ZottaOSHardPA.h* needs to be set to `OTE`.

As we've seen in Chapters 2 and 5, the schedulers implemented in *ZottaOS* are based on two priority queues where the first is the arrival queue holding task instances waiting to be released and the second, the ready queue, contains task instances that have been released but have not yet finished their execution. As these queues are sorted, the task instance at the head of the arrival queue is the next task instance to be released, and the one at the head of the ready queue is the currently active task. In *ZottaOS*, the dispatcher is invoked by the timer when a task instance is released or by an application task when it completes its execution. In both cases, the dispatcher can readily determine the latest possible finishing time of a task. This can either be:

1. the next task release time;
2. the deadline of the active task;
3. the earliest possible release time of the next event-driven task if there are event-driven tasks that are part of the application.

Given the amount of work that has to be done by the scheduled task and the amount of time available to the task, the slowdown that can be applied to the task is

$$slowdown = \frac{remaining\ work}{t - current\ time}$$

where t is the time of the next event when scheduling the task.

In terms of the information that has to be kept on a per task basis, the advantages of RM or DM over EDF are lost. Indeed, the deadline of a task is inherently part of the information needed to schedule a task under EDF. This is not so for RM or DM, because once the static priorities of the task set are found, the deadlines of the tasks are no longer used at run-time. However, to determine the time of the earliest event, the deadline of the task needs to be kept and updated. Therefore the TCB fields related to the deadline and used for EDF have to be replicated for RM or DM. Another element of information specifically required for OTE, and which is used to compute the slowdowns, is the amount of work that a task has to accomplish before it can finish. Figure 6.5 gives the fields that must be added to the TCBs of periodic tasks. Other than a static frequency, there are no additional fields to include in the ETCBs of aperiodic tasks. Table 6.2 summarizes the memory usage of the task control blocks used at run-time to schedule the tasks when static frequencies are also in effect. Addresses are assumed to occupy 2 bytes and each field greater than 2 are allocated on even addresses. The figures in the table also include the overheads introduced by the arrival and ready queues.

Task Type	RM or DM [bytes]	EDF [bytes]
Periodic	$16 + 40 \times N_p$	$16 + 40 \times N_p$
Event-Driven	$24 \times N_a$	$28 \times N_a$

Table 6.2: TCB size in bytes with 2 byte-addresses including arrival and ready queues overheads as a function of the number of application tasks for *ZottaOS-Hard PA* with compilation flag `POWER_MANAGEMENT` set to `OTE` and `STATIC_POWER_MANAGEMENT` defined (N_p = periodic tasks, N_a = event-driven tasks)

```
typedef struct TCB {   // RM or DM scheduling
  UINT8 FrequencyIndex;    // Static frequency setting
  INT32 NextDeadline;      // Current instance deadline
  INT32 Deadline;          // Task's deadline <= Period (user input)
  INT32 WCET;              // Worst-case execution time (user input)
  INT32 RemainingWork;     // Amount of WCET yet to be done by the task
} TCB;

typedef struct TCB {   // EDF scheduling
  UINT8 FrequencyIndex;    // Static frequency setting
  INT32 WCET;              // Worst-case execution time (user input)
  INT32 RemainingWork;     // Amount of WCET yet to be done by the task
} TCB;
```

Figure 6.5: Specific fields added to TCBs of periodic tasks.

Based on the inner workings of *ZottaOS* detailed in Chapter 5, we give a rundown of the modifications that are needed to accommodate *ZottaOS Hard* for OTE. Readers not concerned with the implementation of OTE into *ZottaOS* may skip this part and directly go to the next section without any loss of understanding.

We introduce to following notation to describe these modifications:

$S_{processor}$: Calculated speed of the processor and that needs to be applied.

S_{max}: Maximal speed of the processor.

`Head(Q)`: Denotes the task control block which is at the head of queue `Q`.

Figure 6.6 gives the pseudocode of functions `GetProcessorSpeed` and `OSSetProcessorSpeed`, which respectively determine the operating frequency to be applied when dispatching a task and then sets it. As mentioned previously, event-driven tasks are not subject to dynamic speed adjustments and run at their static frequency or at the maximal processor frequency (S_{max}) if no static power management is in effect. The condition of step 1 checks whether or not OTE can be applied to the active task at hand, and if so, the latest possible time that the task has until it must finish is determined in t. Finally, the calculated speed $S_{processor}$ is applied at step 2.

step 1 **if** `IsPeriodicTask(Head(ReadyQ))` **and** `Head(ReadyQ).next == NULL` **then**
  $t \leftarrow$ Next event time
  if `Head(ReadyQ).RemainingWork` $> t - CurrentTime$ **then**
    $S_{processor} \leftarrow$ `Head(ReadyQ).FrequencyIndex`
  else
    $S_{processor} \leftarrow S_{max} \times$ `Head(ReadyQ).RemainingWork` $/ (t - CurrentTime)$
else
  $S_{processor} \leftarrow$ `Head(ReadyQ).FrequencyIndex`

step 2 Set processor speed to $S_{processor}$

Figure 6.6: Execution steps of `GetProcessorSpeed` and `OSSetProcessorSpeed`.

When a periodic task is released (step 4 of `_OSTimerInterruptHandler`), the `RemainingWork` field of its TCB needs to be set to its worst-case execution time (`WCET` field of the TCB). When the timer interrupts a periodic task, the work that the task has accomplished since it was last dispatched needs to be accounted for prior to scheduling a new task (before step 9 of `_OSTimerInterruptHandler`). Because this update involves several steps that should normally be done in a critical section, the update can be done while the timer function is guaranteed to have unique access to the needed information. Recall that in Chapter 5, we mentioned that during all instructions prior to step 7, no timer interrupt can be raised, and because only `_OSTimerInterruptHandler` updates field `RemainingWork`, the most appropriate time to do the update is between steps 4 and 5 of `_OSTimerInterruptHandler`. Figure 6.7 shows the modifications of `_OSTimerInterruptHandler`.

In step 3

```
tcb.RemainingWork ← tcb.WCET
```

After step 4 and before step 5

```
if not SetActiveTaskRemainingTime and IsPeriodicTask(Head(ReadyQ)) then
   t ← (T_sys - LastRemainingWorkUpdate) × S_processor / S_max
   Head(ReadyQ).RemainingWork ← Head(ReadyQ).RemainingWork - t
   LastRemainingWorkUpdate ← T_sys
```

After step 8 and before step 9

```
LastRemainingWorkUpdate ← T_sys
call SetProcessorSpeed with CurrentTime = T_sys
SetActiveTaskRemainingTime ← false
```

Figure 6.7: Modified execution steps of `_OSTimerInterruptHandler`.

Finally, the modification to `OSEndTask` and `OSSuspendSynchronousTask` are given on Figures 6.8 and 6.9 respectively.

Before step 1

```
SetActiveTaskRemainingTime ← true
```

After step 4 and before step 4

```
LastRemainingWorkUpdate ← T_sys + timer counter
call SetProcessorSpeed with CurrentTime = StartExecutionTime
SetActiveTaskRemainingTime ← false
```

Figure 6.8: Modified execution steps of `OSEndTask`.

After step 4 and before step 5

```
LastRemainingWorkUpdate ← T_sys + timer counter
call SetProcessorSpeed with CurrentTime = StartExecutionTime
SetActiveTaskRemainingTime ← false
```

Figure 6.9: Modified execution steps of `OSSuspendSynchronousTask`.

Dynamic Reclaiming Algorithm

Task Earliness

Comparing the worst-case schedule with the actual schedules of the task set, we can see that some tasks can start earlier because those that executed before it have finished earlier. The aim of the dynamic reclaiming algorithm [AYD04] presented in this section determines precisely how much earlier a task actually begins. This additional time can then safely be attributed to the task to schedule and which will allow it to slow down. In the following, this amount of time is referred to as the **earliness** of the task. For instance, the first instance of Task 2 starts at tick 240 in its worst-case scenario, but it actually starts at tick 100 on Figure 6.3. Hence, this task can be slowed down by a factor of

$$WCET_{\text{Task 2}} / (WCET_{\text{Task 2}} + 240 - 100) = 120 / (120 + 140) = 0.46$$

The key idea to determine the earliness of the task is to maintain a simulated ready queue that mirrors the actual ready queue except that all timings are based on worst-case assumptions. Hence, whenever a task is released, the task is inserted in the ready queue and in the simulation queue in its correct priority position.

The simulation queue is composed of TCBs consisting of 5 fields:

(*task*, `ReleaseTime`, `Deadline`, `RemainingWork`, `CompletionTime`)

The first field is used to identify the task. In fact, this field is inexistent in the actual implementation as the simulated TCBs are the actual TCBs, however this field simplifies the explanation and the example given below. The next 2 fields are the release time and relative deadline of the task and are used to sort the TCB in the simulated ready queue. Field `RemainingWork` corre-

sponds to the actual amount of work that the task instance has to accomplish before terminating its worst-case execution time. At each task preemption, field `RemainingWork` is updated considering the slowdown that was given to the task since it was last dispatched. The last field, `CompletionTime`, is similar but corresponds to the work that the task has to accomplish when no slowdown is applied; the field decreases at a constant rate with the passage of time. When the task finishes, this field gives the amount of slack left behind by the instance and will help to determine the earliness of the tasks that are scheduled. Both these last 2 fields are initialized to the task's worst-case execution time.

As time elapses, the fields in the simulation are updated accordingly:

1. Field `CompletionTime` at the head of the simulated ready queue is decreased at the same rate to that of the passage of time, and when the field reaches zero, the simulated TCB is removed from the queue and the update continues with the next simulated TCB in the queue. By proceeding in this fashion, the simulated queue mimics exactly the worst-case behavior of a ready queue when no slowdowns are applied.
2. Field `RemainingWork` however decreases at a rate that corresponds to the slowdown applied to the task.

Earliness Computation

When dispatching a task instance, its TCB is also in the simulated ready queue because otherwise the task would have overran beyond its worst-case execution time and would have been eliminated from the simulated queue. All the tasks that precede the dispatched task in the simulated task have higher priority and have completed their execution earlier than they would have had the worst-case scenario been applied. The sum of field `CompletionTime` of these TCBs corresponds to the earliness, e, of the dispatched task, and which can safely be given to the task to schedule:

$$slowdown = \frac{\texttt{RemainingWork}}{e + \texttt{CompletionTime}}$$

Another important point to consider is the scheduling policy that is applied. To guarantee that the simulated ready queue mimics exactly the events of the ready queue that it simulates, the scheduling policy applied to both queues follows a variant of EDF called EDF*. The tasks in these queues are first sorted according to their deadlines (like EDF), but when the deadlines are equal, the tasks are sorted according to their release time, and when both these fields are equal, the task with the lowest TCB address is considered to have higher priority. With this scheduling policy, both queues handle TCB insertions and removals in the same order.

Figure 6.10 shows the scheduling diagram of our example task set with the dynamic reclamation scheme. At tick 0, Tasks 1 through 3 are released and Task 1 is scheduled with $slowdown_{\text{Task 1}} = 1$. The content of the simulated ready queue is

(Task 1,0,400,240,240) (Task 2,0,600,120,120) (Task 3,0,1200,120,120)

At tick 100 (= $AET_{\text{Task 1}}$), Task 1 finishes and the completed work of Task 1 is updated in the TCB of the simulated queue:

(Task 1,0,400,240,140) (Task 2,0,600,120,120) (Task 3,0,1200,120,120)

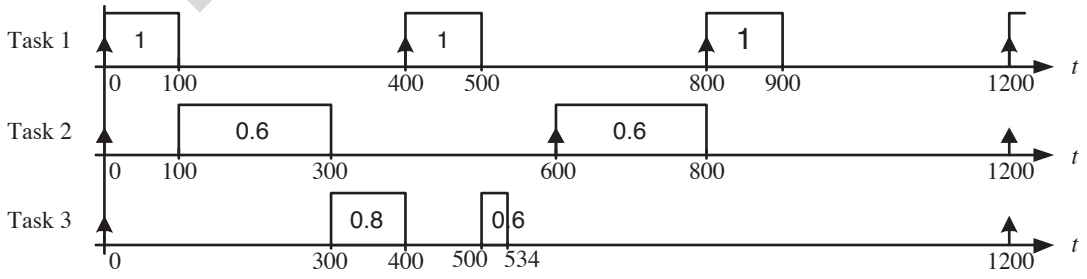


Figure 6.10: Scheduling diagram of task set of Table 6.1 with static frequencies and dynamic reclamation.

Notice that remaining work of Task 1 is not modified since this field is no longer used for this instance.

Task 2 is scheduled with

$$slowdown_{\text{Task 2}} = \frac{120}{140+120} = \frac{6}{13} = 0.46$$

and because our processor only has 3 frequency settings of 12, 16, and 20 MHz, which correspond respectively to slowdowns of 0.6, 0.8 and 1, Task 2 is scheduled with frequency 12 MHz.

At tick 300 (=100 + $AET_{\text{Task 1}}$ / 0.6), Task 2 finishes. The simulated queue is then updated as
(Task 2,0,600,120,60) (Task 3,0,1200,120,120)

Notice that the slack (140) of Task 1 has been entirely consumed and the task is no longer part of the simulated ready queue. Task 3 is then scheduled with

$$slowdown_{\text{Task 3}} = \frac{120}{60+120} = \frac{2}{3} = 0.66$$

and $slowdown_{\text{Task 3}}$ = 0.8 is chosen.

At tick 400, Task 3 is preempted and the second instance of Task 1 is released. The simulated queue is first updated and then the simulated TCB of Task 1 is inserted:
(Task 1,400,800,240,240) (Task 3,0,1200,40,80)

Notice that since Task 1 has an earlier deadline, its simulated TCB is placed before that of Task 3. Also, because Task 3 executed at a slowdown of 0.8 for 100 ticks, its remaining work is decreased by 80, and it needs 40 additional ticks with a slowdown of 1 to complete.

Task 1 is scheduled with a slowdown of 240 / 240 = 1.

At tick 500, Task 1 completes:
(Task 1,400,800,240,140) (Task 3,0,1200,40,80)
and Task 3 is resumed with

$$slowdown_{\text{Task 3}} = \frac{40}{140+80} = \frac{2}{11} = 0.19$$

Hence, the lowest possible slowdown of 0.6 is chosen.

At tick 534, Task 3 terminates and the processor is set into its sleep mode. Since no task is scheduled other than the idle task, the simulated ready queue is not updated.

At tick 600, the second instance of Task 2 is released and the simulated queue becomes
(Task 1,400,800,240,40) (Task 3,0,1200,40,80) (Task 2,600,1200,120,120)

Notice how the TCB of Task 2 is inserted after that of Task 3. Because both tasks have the same deadline, we are free under the EDF policy to insert them in any order as long as the ready queue uses the same criterion. Sorting the TCBs according to their respective release time turns out to be beneficial. Task 2 is then scheduled with

$$slowdown_{\text{Task 2}} = \frac{120}{40+80+120} = \frac{1}{2} = 0.5$$

and $slowdown_{\text{Task 2}}$ = 0.6 is chosen.

At tick 800, Task 2 completes (120 / 0.6 = 200), and the 3rd instance of task is released. The simulated queue is first updated with the completion of Task 2 before inserting the simulated TCB of Task 1:
(Task 2,600,1200,120,40) (Task 1,800,1200,240,240)

And Task 1 is scheduled with

$$slowdown_{\text{Task 1}} = \frac{240}{40+240} = \frac{6}{7} = 0.86$$

and $slowdown_{\text{Task 1}}$ = 1 is chosen.

At tick 900, Task 1 terminates and the simulation queue is left untouched, as there are no application tasks to schedule.

Finally, at tick 1200, the simulation is updated and becomes empty. A new instance of each task is then released and the execution continues with the same pattern as for tick 0.

Additional power savings can be achieved by combining the scheme with OTE. Indeed, at tick 800, the 3rd instance of Task 1 is released and it is the only task in the ready queue, and because it has 400 ticks to complete its work, it can be scheduled with

$$slowdown_{\text{Task 1}} = \frac{240}{400} = \frac{6}{10} = 0.6$$

This is in fact the only task instance from our task set example that can benefit from OTE.

The overheads needed for the dynamic reclaiming algorithm implemented in *ZottaOS-Hard* including the added fields to the task control blocks are given in Table 6.3. Adding OTE to the algorithm does not add any additional memory usage.

Task Type	DRA [bytes]
Periodic	$20 + 44 \times N_p$
Event-Driven	$40 \times N_a$

Table 6.3: TCB size in bytes with 2 byte-addresses including arrival, ready and simulated queues overheads as a function of the number of application tasks for *ZottaOS-Hard PA* with compilation flag `POWER_MANAGEMENT` set to `DRA` (N_p = periodic tasks, N_a = event-driven tasks)

Figure 6.11 shows the measured relative energy savings as a function of the load submitted to the processor obtained by comparing EDF with and without dynamic reclamation combined with OTE on TI MSP-EXP430F5438 evaluation kit equipped with a microcontroller prerelease of MSP430F55438A, the X430F55438. The task set is composed of 3 synthetic tasks and their respective characteristics are given in Table 6.4, where ρ is the load that is applied to the processor.

	WCET	*P* = *D*	*AET*
Task 1	1000	3000	ρ ×1000
Task 2 and 3	ρ × 1000	3000	ρ × 1000

Table 6.4: Measured power saving test task set. (*P* and *D* are respectively the period and deadline of the task, *WCET* and *AET* are respectively the worst-case and actual execution time of the task)

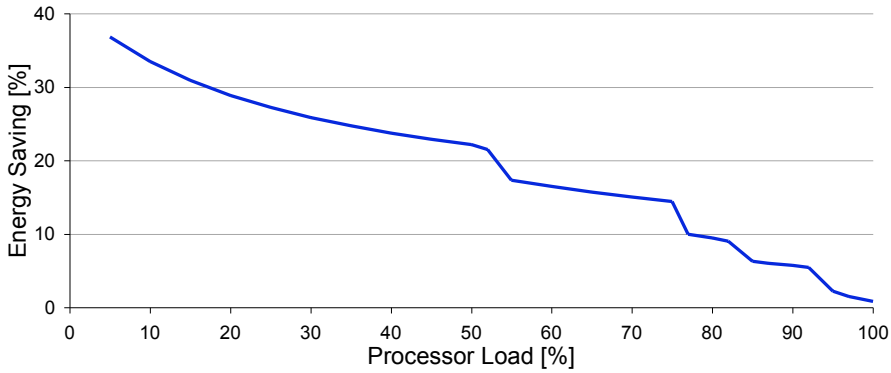


Figure 6.11: Energy savings of 3 identical tasks on prerelease X430F55438 as a function of the processor load.

In the task set, Task 1 is always the first task that is executed when the tasks are released. The values portrayed on Figure 6.11 are obtained by computing (E_{EDF} – E_{DROTE}) / E_{EDF}, where E_{EDF} is the measured consumption of the task set under EDF with no power management other than

applying sleep mode LPM1 when the processor becomes idle, and E_{DROTE} is the same measured consumption but with dynamic reclamation and OTE power management algorithms.

As can be seen from Figure 6.11, there are 5 different task slowdown (or frequency settings) assignments and each transition from one assignment to another produces a step. The corresponding task frequency assignments annotated on the figure are given in Table 6.5.

	Task 1	Task 2	Task 3
A	18 MHz	12 MHz	12 MHz
B	18 MHz	16 MHz	12 MHz
C	18 MHz	16 MHz	16 MHz
D	18 MHz	18 MHz	16 MHz
E	18 MHz	18 MHz	18 MHz

Table 6.5: Frequency assignments at different processor load ranges on Figure 6.11.

Notwithstanding the overhead needed to simulate the ready queue and obtain the earliness of Tasks 2 and 3, this overhead, which also contributes to the energy consumption of the processor, is less than the energy it actually saves the processor.

Hence, when an application has variability in its execution time, dynamic reclamation combined with OTE can bring a significant amount of energy savings. For example, assume that the application has a worst-case execution corresponding to a processor utilization of 80% but its demand on the processor can vary down to 20%. Static frequency assignments corresponding to entry C in Table 6.5 produces an earning of approximately 10% over simple sleep mode schemes. When the application requires less computation power and drops to a load 20%, an additional 20% energy saving is done via dynamic reclamation and OTE. Of course the longer the application stays at its lowest load the better.

For Your Information

There are 2 possible power management settings to benefit from the dynamic reclaiming algorithm. Setting the symbolic constant `POWER_MANAGEMENT`, defined in *ZottaOSHardPA.h*, to `DRA` produces only dynamic reclamations energy savings. However, setting `POWER_MANAGEMENT` to `DR_OTE` also combines the advantages of the one-time-extension algorithm with those of the dynamic reclaiming algorithm.

Deadline Monotonic Slack (DM Slack)

Rather than simulating the ready queue with the worst-case execution times to get the earliness of a task, for Rate Monotonic or Deadline Monotonic scheduling we have designed a specific algorithm that computes this earliness. Recall that a task that finishes sooner than it would under its worst-case assumption allows other tasks to start earlier, and the difference between the task's worst-case execution time (determined offline) and its actual execution time (determined online) called slack, can be distributed to other tasks. However under RM or DM scheduling, the slacks cannot be distributed arbitrarily. Say that a medium task finishes earlier. The fact that this task finished sooner cannot influence higher priority tasks since these preempt the medium task every time they are released. The opposite is not true; lower priority tasks start sooner if higher priority tasks finish earlier. Hence, when a task finishes sooner, the slack that it leaves can only be given to lower priority tasks. This is the key observation of our method. A similar but slightly more complex scheme is proposed by Saewong and Rajkumar [SAE03] and requires more computations.

To compute the slacks, we need to monitor the amount of time that a task still requires in order to finish. Let this quantity be called $remainingWork$. When task i is released, $remainingWork_i$ is set to the task's worst-case execution time and decreases with a rate depending on the slowdown that is applied to the task when it has the processor. Let $slack_j$ be the amount of slack left by a

task running with priority j, and let higher integral numbers represent higher priorities. When task i is scheduled, the $slowdown_i$ that can be applied to it is given by

$$slowdown_i = \begin{cases} \dfrac{remainingWork_i}{slack_j + remainingWork_i} & \text{if } j > i \\ 1 & \text{otherwise} \end{cases}$$

Whenever a task is preempted by another, we need to update its remaining execution time since this quantity is required to compute its slowdown:

$$remainingWork_i = remainingWork_i - slowdown_i \times \left(T_{sys} - lastStartTime\right),$$

where T_{sys} is the current system time and $lastStartTime$ is the time at which the last scheduled task (the one being preempted) was dispatched. The available slack that the task used during its execution also needs to be modified:

$$slack_j = \max\left(0, slack_j - \left(T_{sys} - lastStartTime\right)\right).$$

Finally, when a task finishes its execution, the slack it leaves to other lower priority tasks is simply

$$slack_i = remainingWork_i - slowdown_i \times \left(T_{sys} - lastStartTime\right),$$

which is simply the remaining work that task i did not have to execute.

Figure 6.12 shows the scheduling diagram of our example task set of Table 6.1 with DM slack power saving algorithm. At tick 0, Task 1 is scheduled at the maximal frequency of 20 MHz, and it completes at tick 100. Because in its worst-case scenario the task would still have 140 ticks before terminating, this value constitutes the slack that can be distributed amongst the other tasks of the set:

$slack_{\text{Task 1}} = remainingWork_{\text{Task 1}} - 100 = 240 - 100 = 140$ ticks.

Using this slack, Task 2 is scheduled with

$$slowdown_{\text{Task 2}} = \frac{WCET_{\text{Task 2}}}{slack_{\text{Task 1}} + WCET_{\text{Task 2}}} = \frac{120}{140 + 120} = 0.46.$$

Hence a slowdown of 0.6 is chosen for this task.

At tick 100 + 120 / 0.6 = 300, Task 2 terminates and leaves a slack of

$slack_{\text{Task 2}} = 120 - 0.6 \times 200 = 0$ ticks.

Since there is no slack to attribute to Task 3, its slowdown is 1, but by taking into account the static frequency setting for this task, Task 3 is scheduled with a slowdown of 0.8.

At tick 400, Task 3 is preempted by the release of the 2nd instance of Task 1. Task 3's remaining work is calculated as

$remainingWork_{\text{Task 3}} = 120 - 0.8 \times 100 = 40$ ticks.

Task 1 is then scheduled with a slowdown of 1. At Tick 500, Task 1 finishes and leaves a slack of

$slack_{\text{Task 1}} = 240 - 100 = 140$ ticks.

Task 3 then resumes its execution with a slowdown of

$$slowdown_{\text{Task 3}} = \frac{40}{140 + 40} = 0.22.$$

Since 0.6 is the minimal slowdown, this is the slowdown that is applied to the processor and the task will finish its execution at

$500 + (remainingWork_{\text{Task 3}} - (WCET_{\text{Task 3}} - AET_{\text{Task 3}})) / slowdown_{\text{Task 3}}$
$= 500 + (40 - (120 - 100)) / 0.6 = 534.$

At tick 534, Task 3 finishes and leaves a slack of

$slack_{\text{Task 3}} = 40 - 0.6 \times 34 = 20$ ticks,

which cannot be used by any other task since Task 3 instances have the lowest priority of the whole task set.

At tick 600, the 2nd instance of Task 2 is released and it would be scheduled with a slowdown of 1 if it hadn't a smaller static slowdown of 0.8.

At tick 750, Task 2 terminates and leaves a calculated slack of

$slack_{\text{Task 2}} = WCET_{\text{Task 2}} - 0.8 \times 150 = 0$ ticks.

At tick 800, the 3rd instance of Task 1 is released and completes at tick 900 with a slowdown of 1.

DM Slack does a good job to distribute the slacks but has poor efficiency when there are few and very dispersed tasks in the resulting schedule. However, combining OTE with DM Slack can alleviate this problem. Namely, the 2nd instance of Task 2 and the 3rd instance of Task 1 would have both executed with a slowdown of 0.6, and provided additional power savings. In fact, *ZottaOS-Hard PA* and *ZottaOS-Soft PA* implement only DM Slack combined with OTE as this produces the most energy earnings.

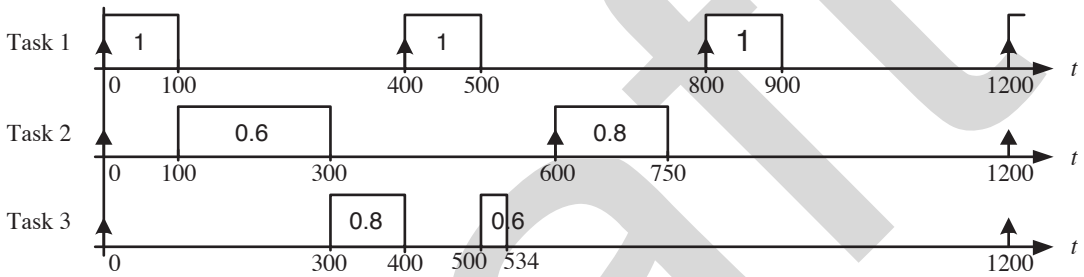


Figure 6.12: Scheduling diagram of task set of Table 6.1 with static frequencies and DM Slack algorithm.

We have also conducted an experiment with the same task settings as those used to evaluate the dynamic reclaiming algorithm with OTE. The results we obtain are exactly the same. This can be explained by the following 2 arguments:

1. There is no scheduling difference between DM, RM or EDF for the given task characteristics. All scheduling algorithms produce the same events and the tasks are executed in the same order.
2. The task demands are such that the difference between the overheads introduced by the dynamic reclaiming algorithm with OTE and DM Slack with OTE are negligible.

In terms of memory occupancy, DM slack is cheaper than the dynamic reclaiming algorithm. Table 6.6 shows the overheads brought by DM slack as implemented in *ZottaOS-Hard PA* with the same assumptions as those given for Table 6.3.

Task Type	DM Slack [bytes]	DM Slack with static frequencies [bytes]
Periodic	$16 + 38 \times N_p$	$16 + 40 \times N_p$
Event-Driven	$30 \times N_a$	$32 \times N_a$

Table 6.6: TCB size in bytes with 2 byte-addresses including arrival, ready and simulated queues overheads as a function of the number of application tasks for *ZottaOS-Hard PA* with compilation flag `POWER_MANAGEMENT` set to `DM_SLACK` and without and with flag `STATIC_POWER_MANAGEMENT` (N_p = periodic tasks, N_a = event-driven tasks)

For Your Information

To benefit from DM Slack combined with OTE when using either DM or RM scheduling, set the symbolic constant `POWER_MANAGEMENT`, defined in *ZottaOSHardPA.h*, to `DM_SLACK`.

Summary

Even on microcontrollers qualified as ultralow power, the methods provided by the power-aware versions *ZottaOS* can achieve additional energy savings if one or both conditions are met:

1. The application is composed of tasks where some require fast response times and are ran at the fastest possible speed, while others simply have to meet their deadlines. In this case, *ZottaOS* takes care of the power adjustments without adding any code complexity to the application. The frequency settings may here be static or dynamic.
2. The application is composed of tasks that have varying computational demands. In this case, dynamic techniques are required to capture the unused processor cycles and to distribute them to tasks, which can then execute at a lower frequency and at a lower core voltage, and yet still meet their deadlines.

We have shown in this chapter that it is possible to achieve from 10 to 30% additional energy earnings on most practical applications.

Bibliography

[AYD04] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, *Power-Aware Scheduling for Periodic Real-Time Tasks*, IEEE Transactions on Computers, Vol. 53, No. 5, pp. 584-600, May 2004.

[SAE03] S. Saewong, and R. Rajkumar, *Practical Voltage-Scaling for Fixed-Priority Real-Time Systems*, Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 106-115, May 2003.

[SHI99] Y. Shin, and K. Choi, *Power-Conscious Fixed Priority Scheduling for Hard Real-Time Systems*, Proceedings of the 36th Design Automation Conference (DAC '99), pp. 134-139, June 1999.

Chapter 7

Reference

Code Size

In terms of code size, *ZottaOS* ranges from 7127 to 9527 bytes, where the minimum size is without any power management, and the maximum is with the dynamic reclaiming algorithm with OTE, which is the scheme that involves the most processing and code. Of these, 440 bytes are for the simple UART API and 2130 bytes to process the interrupt table, branching at particular devices and source within this device. Without this table processing, the code size of the RTOS would range from 4997 to 7397 bytes.

Compilation Switches

There are 3 groups of settings that directly influence an application and that are specified by symbolic constants, which provide conditional compilation directives.

1. Microcontroller Type or Family Member

For Texas Instruments MSP430, the constant MSPSERIES defined in ZottaOS_5xx.h should be set to one of:

`MSP430`	applies to all series other than MSP430F5xx or MSP430F5xxA
`MSP430F5xx`	applies to MSP430F5xx with 3 frequency/voltage settings for power-aware schemes.
`MSP430F5xxA`	applies to MSP430F5xxA with 4 frequency/voltage settings

2. Power Management Schemes

On microcontrollers with dynamic frequency/voltage scaling capabilities such as the MSP430F5xx or MSP430F5xxA, the power management algorithm can be defined by setting the constant `POWER_MANAGEMENT` defined in ZottaOS_5xx.h to one of the 5 settings:

`NONE`	No power management, uses sleep mode only
`OTE`	One Task Extension algorithm (OTE)
`DRA`	Dynamic Reclaiming algorithm (works only with EDF∗)
`DR_OTE`	Dynamic Reclaiming algorithm with OTE (works only with EDF∗)
`DM_SLACK`	Deadline Monotonic Slack with OTE

All the above power management schemes (even `NONE`) can be combined with a static frequency setting. To benefit from this feature, symbolic constant `STATIC_POWER_MANAGEMENT` also needs to be defined. This constant is also defined in ZottaOS_5xx.h.

A separate program that is part of the *ZottaOS* distribution can help you determine static frequencies of all the task of your application.

3. Scheduling Algorithm Selection

ZottaOS can work under 3 possible scheduling algorithms. Under EDF, tasks are scheduled according to their absolute deadlines; tasks with earlier deadlines have higher dynamic priorities than those with later deadlines. When *ZottaOS* is used with dynamic reclaiming power-aware algorithm (`DRA` or `DR_OTE`), the scheduling algorithm must be EDF∗, which is similar to EDF but also takes the release time and addresses of the tasks into account when tasks have identical absolute deadlines. Without power management, it is better not to use EDF∗. The 3rd scheduling algorithm is deadline monotonic scheduling where tasks have a static priority based on their relative deadline. Shorter deadlines have higher priority than longer deadlines.

Selection of the scheduling algorithm is done in ZottaOS_5xx.h by means of the symbolic constant `SCHEDULER_REAL_TIME_MODE`, which must take one of the 3 possible values:

	Used with power-aware schemes
EARLIEST_DEADLINE_FIRST	NONE, or OTE
EARLIEST_DEADLINE_FIRST_STAR	NONE (not recommended), DRA, or DR_OTE
DEADLINE_MONOTONIC_SCHEDULING	NONE, OTE, or DM_SLACK

Function Calls

The remaining of this chapter describes all the functions that can be called to initialize the application (called from main) or from the application tasks or ISRs. *ZottaOS* only provides function calls to assist an application; there are no global variables that are defined by *ZottaOS* that have a meaning for an application. All function identifiers are prefixed by OS, except for functions _OSCreateTask and _OSCreateSynchronousTask. Because of its implementation in C and in assembler, there are 7 global variables that are prefixed by _OS and that should never be referenced from the application. We therefore strongly recommend not prefixing application variables or function identifiers with _OS or OS.

Unless explicitly indicated, all function prototypes are declared in the header file ZottaOS_5xx.h. These 31 functions are apportioned into groups according to their purposes. They are:

1. tasks creation and instance termination,
2. concurrent FIFO queue API for inter-task communication,
3. asynchronous reader-writer functions for input and output processing,
4. interrupt Subsystem Functions
5. simple UART API to easily input and output from a UART peripheral,
6. miscellaneous functions.

Defining Tasks

OSCreateTask: Periodic Task Creation

Prototype

ZottaOS-Hard:

```
BOOL OSCreateTask(void task(void *), UINT16 periodCycles,
   INT32 periodOffset, INT32 deadline, void *argument)
```

ZottaOS-Hard PA:

```
BOOL OSCreateTask(void task(void *), INT32 wcet,
   UINT16 periodCycles, INT32 periodOffset, INT32 deadline,
   void *argument)
```

ZottaOS-Soft and *ZottaOS-Soft PA*:

```
BOOL OSCreateTask(void task(void *), INT32 wcet,
   UINT16 periodCycles, INT32 periodOffset, INT32 deadline,
   UINT8 m, UINT8 k, UINT8 startInstance, void *argument)
```

Scope

Applies to all kernel versions of *ZottaOS* albeit with different parameters.

Description

Creates a periodic task with its timing characteristics expressed in timer ticks. The function call to use depends on the selected *ZottaOS* kernel and for which different or more complete task characteristics are taken into account when scheduling the task set.

The period of a task is defined by 2 parameters. The first is the number of system clock wraparound ticks defined as 2^{30} (`periodCycles`) and the second is the number of ticks within the 2^{30} boundary (`periodOffset`). Together these define the period on 16 + 30 bits. Hence, the actual period of a task is equal to

$$period = \texttt{periodCycles} \times 2^{30} + \texttt{periodOffset}$$

The maximum period that can be assumed by a periodic task is

68 years with a 32 kHz timer crystal or resonator;
2 years with a 1 MHz crystal or resonator.

Parameters

1. Pointer to the code denoting the task template.
2. (`INT32`) `wcet`: Number of ticks needed to execute the task; this parameter is used with power-aware versions of *ZottaOS* when scheduling tasks with DVFS. It is also required with *ZottaOS-Soft* to compute the workload of mandatory tasks when dispatching optional tasks.
3. (`UINT16`) `periodCycles`: Number of full 2^{30} cycles.
4. (`INT32`) `periodOffset`: Remaining period ticks, must be < 2^{30}.
5. (`INT32`) `deadline`: Task's deadline (`wcet` < `deadline` ≤ *period*).
6. `(UINT8) m`: Used for (m,k) QoS guarantees, i.e. out of k consecutive instances, there should be `m` instances that meet their deadlines, `m` must be > 0. This parameter is only defined for *ZottaOS-Soft* versions.
7. `(UINT8) k`: Second component of the QoS tuple (see above). `k` must be ≥ `m`. When `k` = `m`, all instances of the task must meet their deadlines. This parameter is only defined for *ZottaOS-Soft* versions.
8. `(UINT8) startInstance`: Initial task offset delay counted in number of task instances. 0 ≤ `startInstance` < `k`. This parameter is only defined for *ZottaOS-Soft* versions.
9. `(void *) argument`: An instance specific 16-bit value that is accessible when a task instance is active.

Returned Value

(`BOOL`) `TRUE` if the task is successfully created and `FALSE` otherwise. The function can only fail when there is a memory allocation failure.

Restrictions

This function can only be invoked before starting *ZottaOS* and should be called by the

main program or one of the initializing functions called by main. There is no concept of dynamic task creation in *ZottaOS* as it would be impossible to guarantee the deadlines of the tasks without a priori knowledge of the workload.
The task implicitly starts its execution once *ZottaOS* is invoked by calling `OSStart-Multitasking`.
If `STATIC_POWER_MANAGEMENT` is defined for the power-aware versions of *ZottaOS*, this function is identical to calling `_OSCreateTask` with a static frequency setting equal to the processor's fastest speed.
Usually the specified code of a task is univocal in that each created task has its own code. However, there is no restriction in specifying the same function to denote several different tasks. The only limitation is with regard to static variables visible only from within a function. For instance,

```
void Specification(void *arg)
{
  static int i;
  auto int j;
  ...
} /* end of Specification */
```

The static variable `i` is never allocated on the execution stack as its lifetime is greater than that of a task instance. In this respect, static variables behave as global variables. If there are several tasks sharing function `Specification`, all will have their own instance of variable `j` but all will share the same variable `i`. Care must be exerted to maintain the coherence of this latter variable.

Giving Arguments to Tasks

The `OSCreateTask()` routine allows you to pass a single 16-bit argument to the user-defined task. For cases where multiple arguments must be passed to the task, this limitation is easily overcome by using a *statically allocated and initialized* structure containing all the arguments. The reference to this structure (casted to a void pointer) can then be passed when `OSCreateTask()` is called. Once the task is created, this argument is stored in the task's TCB and a copy of the argument is made accessible every time an instance of the task is activated.

Example 1 - Passing a Single Integer to a Task
There are 2 ways to pass a simple integer to tasks. The caller can either use a unique integer for the tasks, insuring that each task's argument remains intact, or the caller can simply type cast the integer value into a pointer. This last solution requires the least amount of overhead.

```
void Template(void *);

int main(void)
{
  int i;
  ...
  /* Create and number tasks from 1 to NUM_TASKS */
  for (i = 1; i <= NUM_TASKS; i += 1)
     OSCreateTask(Template...,(void *)i);
  ...
  return OSStartMultitasking();
} // end of main

int taskId[NUM_TASKS];
void Template(void *);

int main(void)
{
  int i;
  ...
```

```
    /* Create and number tasks from 1 to NUM_TASKS */
    for (i = 0; i < NUM_TASKS; i += 1) {
       taskId[i] = i + 1;
       OSCreateTask(Template...,(void *)&taskId[i]);
    }
    ...
    return OSStartMultitasking();
} // end of main
```

Note that because each task has its own TCB, there can be no confusion as to which argument is received by the task code when the same function template is used for different tasks. In such a case, the task's argument can be used to differentiate between the different tasks.

Example 2 - Incorrect Task Argument Passing
This example performs argument passing incorrectly. It passes the address of variable `taskId`, which is local to `main()` and should become a shared memory space visible to all tasks but doesn't.

```
void Template(void *);

int main(void)
{
  int taskId;
  ...
  /* Create and number tasks from 1 to NUM_TASKS */
  for (taskId = 1; taskId <= NUM_TASKS; taskId += 1)
     OSCreateTask(Template...,(void *)&taskId);
  ...
  return OSStartMultitasking();
} // end of main
```

There are a number of flaws with this code fragment.

(1) The main program no longer exists once `OSStartMultitasking()` is called. The run-time stack is adjusted and all local variables of `main()` are wiped-out from the stack. All tasks can thus access an inexistent variable. To correct this problem variable `taskId` can either be prefixed by static (awkward solution taking advantage of how variables with a limited scope and extended lifetime are actually implemented), or `taskId` can be declared globally.

(2) Notwithstanding the problem stated above, if the intent pursued by the code fragment is to baptize all tasks with the same number, the last correction suggested above accomplishes the desired goal. Otherwise, all tasks obtain the address of `taskId` and each will have the same id. Variable `taskId` becomes a shared variable.

See Also `OSEndTask`, `_OSCreateTask`

_OSCreateTask: Periodic Task Creation with Static Frequency Setting

Prototype

ZottaOS-Hard PA:

```
BOOL _OSCreateTask(void task(void *), INT32 wcet,
   UINT16 periodCycles, INT32 periodOffset, INT32 deadline,
   void *argument, UINT8 frequencyIndex)
```

ZottaOS-Soft PA:

```
BOOL _OSCreateTask(void task(void), INT32 wcet,
   UINT16 periodCycles, INT32 periodOffset, INT32 deadline,
   UINT8 m, UINT8 k, UINT8 startInstance, void *argument,
   UINT8 frequencyIndex)
```

Scope

Applies to all power-aware kernel versions of *ZottaOS* with header files generated by *ZottaOS Configurator Tool*.
Symbolic symbol `STATIC_POWER_MANAGEMENT` must also be defined in ZottaOS–HardPA.h or ZottaOSSoftPA.h.

Description

This function is identical to `OSCreateTask` except that a static frequency setting is specified along with the task. This extra parameter is then used whenever the task is scheduled. Note that this index can be combined with dynamic power management schemes.

Parameters

1. Pointer to the code denoting the task template.
2. (`INT32`) `wcet`: Worst-case execution time of a task instance and which is equal to the number of timer ticks needed to execute the task.
3. (`UINT16`) `periodCycles`: Number of full 2^{30} cycles.
4. (`INT32`) `periodOffset`: Remaining period ticks, must be < 2^{30}.
5. (`INT32`) `deadline`: Task's deadline (`wcet` < `deadline` ≤ *period*).
6. `(UINT8) m`: Used for (m,k) QoS guarantees, i.e. out of k consecutive instances, there should be `m` instances that meet their deadlines, `m` must be > 0. This parameter is only defined for *ZottaOS-Soft PA*.
7. `(UINT8) k`: Second component of the QoS tuple (see above). `k` must be ≥ `m`. When `k` = `m`, all instances of the task must meet their deadlines. This parameter is only defined for *ZottaOS-Soft PA*.
8. `(UINT8) startInstance`: Initial task offset delay counted in number of task instances. 0 ≤ `startInstance` < `k`. This parameter is only defined for *ZottaOS-Soft PA*.
10. `(void *) argument`: An instance specific 16-bit value that is accessible when a task instance is active.
11. (`INT8`) `frequencyIndex`: Static frequency index of the task instances; this index corresponds to the fastest speed that will be applied to the task when its instances are scheduled.

Returned Value

(`BOOL`) `TRUE` if the periodic task was successfully created and `FALSE` otherwise. The function can only fail when there is a memory allocation failure.

Restrictions

All restrictions described for `OSCreateTask` also apply to `_OSCreateTask`.

Note

Each frequency index corresponds to an entry into a system table generated by *ZottaOS Configurator Tool*, and which contains the operating frequency settings along with the minimal voltages that can be applied to the core when task instances are scheduled. Together, these tuples allow slowdowns so that power savings can be realized.
Function `_OSCreateTask` can only be used with microcontrollers having a power manager module. Currently, these microcontrollers are for some members of the MSP430F5xx family and all CC430xxx families.

The frequency indices that can be used are generated by the *ZottaOS Configurator Tool* and can be found in the header file ZottaOS_msp430x*ZZZ*.h or ZottaOS_cc430x*ZZZ*.h, where *ZZZ* is the family member of the microcontroller (for example 54xA).

Index	MSP430F5xx†	CC430xxx
`OS_8MHZ_SPEED`	✓	✓
`OS_12MHZ_SPEED`	✓	✓
`OS_16MHZ_SPEED`	undefined	✓
`OS_20MHZ_SPEED`	✓	✓
`OS_25MHZ_SPEED`	✓	undefined

† Applies for MSP430F54xxA and MSP430F55xx
A checkmark in the table indicates a valid frequency index.

See Also `OSCreateTask`

OSEndTask: Periodic Task Termination

Prototype `void OSEndTask(void)`

Scope Applies to all kernel versions of *ZottaOS.*

Description Very last instruction executed by a periodic task. The purpose of this call is to finish the current instance of the calling task and to clean up the run-time stack for the next task that will be scheduled. Any code appearing after the call will never be executed. Also, without this call, a task instance can never terminate and remains active until the next release time at which time the scheduler will fall into an infinite loop.
This function also schedules the next ready task and sets its operating speed if a power management scheme is in effect.

Parameters None

Returned Value None

Restrictions This function can only be invoked from within a function acting as the code of a periodic task or from a function called by a periodic task.
There is no limit to the number of times this function appears in the task. For example,

```
void Task(void *arg)
{
   if (!globalCondition)
      OSEndTask();      // Do not continue
   /* Continue executing if globalCondition is true. */
   /* Do something useful here. */
   OSEndTask();
}
```

See Also `OSSuspendSynchronousTask`

OSGetTaskInstance: Periodic Task Instance Number

Prototype `UINT8 OSGetTaskInstance(void)`

Scope Only applies to *ZottaOS-Soft* and *ZottaOS-Soft PA* kernel versions of *ZottaOS.*

Description Returns the instance number of the calling task.

Parameters None

Returned Value (`UINT8`) A number between 0 and k – 1, where k is the number of consecutive task instances that was specified when creating the periodic task with QoS guarantees. Because periodic tasks scheduled with an (m,k)-QoS to handle overloaded situations may have up to k – m instances that are dropped by the scheduler, if a task has a periodic update to perform and which depends upon the number of times it is ran, this function can be used to determine the number dropped instances.

Sample In this small example, at least one of 10 consecutive instances requires at particular processing. Because some instances are dropped, we can make use of the instance identifier to count the number of missed periods.

```
void SoftPeriodicTask(void *arg)
{
  static INT8 lastInstance = -1;
  static UINT8 counter; // static variables are initialized with 0
  INT8 inc;
  if ((inc = OSGetTaskInstance() - lastInstance) < 0)
     inc += k;
  counter += inc;
  if (counter >= 10) {
     /* Do something useful here */
     counter = 0;
  }
  lastInstance = OSGetTaskInstance();
}
```

See Also `OSCreateTask`, `_OSCreateTask`

OSCreateSynchronousTask: Event-Driven Task Creation

Prototype

ZottaOS-Hard:

```
BOOL OSCreateSynchronousTask(void task(void *),
                             INT32 workLoad, void *event, void *argument)
```

ZottaOS-Soft, *ZottaOS-Hard PA* and *ZottaOS-Soft PA*:

```
BOOL OSCreateSynchronousTask(void task(void *), INT32 wcet,
   INT32 workLoad, UINT8 aperiodicUtilization, void *event,
   void *argument)
```

Scope

Applies to all kernel versions of ZottaOS albeit with different parameters.

Description

Creates an event-driven task. Compared to periodic tasks that periodically have active instances, event-driven tasks become schedulable when they are explicitly signaled.

Parameters

1. Pointer to the code denoting the task template.
2. (`INT32`) `wcet`: Worst-case execution time of a task instance and which is equal to the number of timer ticks needed to execute the task. This parameter is only used for dynamic power management (*ZottaOS-Soft*, *ZottaOS-Hard PA* or *ZottaOS-Soft PA*).
3. (`INT32`) `workload`: Equal to `wcet`/(available processor load). This parameter is equivalent to the period and deadline of a periodic task.
   Under Deadline Monotonic scheduling and when an event-driven task is signaled but its previous instance has terminated but not yet completed its specified workload, a new instance is inserted in the arrival queue; otherwise a new instance is created and immediately scheduled. Let A_i denote the starting time (in ticks) of task instance i. When signaling this task, instance $i+1$ is scheduled at
   $$A_{i+1} = \min(A_i + \texttt{workload}, \mathit{Time})$$
   where *Time* is the current time. The priority level of an event-driven task is also derived from the workload parameter.
   Under Earliest Deadline First scheduling, the same parameter (`workload`) acts as the relative deadline of an instance from the moment it is scheduled. When the previous instance of this task has terminated, a new instance is created and immediately scheduled. Let D_i denote the deadline of the i th instance of *any* event-driven task, and let $U_{periodic}$ denote the processor utilization of *all* periodic tasks. Then, the deadline assigned to the new instance is
   $$D_{i+1} = \max(D_i, \mathit{Time}) + \texttt{wcet}/(1 - U_{periodic}) = \max(D_i, \mathit{Time}) + \texttt{workload}$$
4. (`UINT8`) `aperiodicUtilization`: Total processor utilization of all event-driven tasks × 256 (equal to $1 - U_{periodic}$ defined above); this parameter is only used for EDF when *ZottasOS-Soft* or a dynamic power management is in effect, and can be set to 0 if Deadline Monotonic scheduling is selected.
5. (`void *`) `event`: Opaque descriptor returned by `OSCreateEventDescriptor` and associated with the task. This parameter is used by `OSScheduleSuspendedTask` to awake and create a new task instance.
6. `(void *) argument`: An instance specific 16-bit value that is accessible when a task instance is active.

Returned Value

(`BOOL`) `TRUE` if the event-driven task was successfully created and `FALSE` otherwise. The function can only fail when there is a memory allocation failure.

Restrictions

This function can only be invoked before starting *ZottaOS* and can be called by the main program or one of the initializing functions called by main. Once *ZottaOS* is invoked by calling `OSStartMultitasking`, task instances have to implicitly start the execution an event-driven task. This is done with a call to `OSScheduleSuspendedTask`.
An event-driven task can only be associated with a single event and it can also signal

itself with `OSScheduleSuspendedTask`.
When used with a power-aware kernel of *ZottaOS* and if `STATIC_POWER_MANAGEMENT` is defined in either ZottaOSHardPA.h or ZottaOSSoftPA.h, this function is identical to calling `_OSCreateSynchronousTask` with a static operating frequency equal to the processor's fastest speed.
No dynamic power management scheme is applied to an event-driven task.
As for periodic tasks, there is no restriction in specifying the same function to denote several different event-driven tasks associated with the same event or with a different event. The only limitation concerns the use of static variables defined in the code (see `OSCreateTask`). Also see `OSCreateTask` on how to pass arguments to the task instances.

See Also `CreateEventDescriptor`, `_OSCreateSynchronousTask`, `OSSuspendSynchronousTask`, `OSScheduleSuspendedTask`

_OSCreateSynchronousTask: Event-Driven Task Creation with Static Frequency

Prototype

```
BOOL _OSCreateSynchronousTask(void task(void *), INT32 wcet,
   INT32 workLoad, UINT8 aperiodicUtilization, void *event,
   void *argument, UINT8 frequencyIndex)
```

Scope

Applies to *ZottaOS-Hard PA* and *ZottaOS-Soft PA* and with header files generated by *ZottaOS Configurator Tool*.
Symbolic symbol `STATIC_POWER_MANAGEMENT` must also be defined in ZottaOS–HardPA.h or ZottaOSSoftPA.h.

Description

This function is identical to `OSCreateSynchronousTask` except that a static frequency setting is specified along with the event-driven task. This extra parameter is then used whenever the task is scheduled.

Parameters

1. Pointer to the code denoting the task template.
2. (`INT32`) `wcet`: Worst-case execution time of a task instance and which is equal to the number of timer ticks needed to execute the task; this parameter is only used for dynamic power management, and can be set to 0 when no dynamic power management is in effect.
3. (`INT32`) `workload`: Equal to `wcet`/(available processor load). This parameter is equivalent to the period and deadline of a periodic task. See `OSCreateSynchronousTask` for a complete description.
4. (**`void`** `*`) `event`: Opaque descriptor returned by `OSCreateEventDescriptor` and associated with the task. This parameter is used by `OSScheduleSuspendedTask` to wake-up a task and create a new task instance.
5. (`UINT8`) `aperiodicUtilization`: Total processor utilization of all event-driven tasks × 256 (see `OSCreateSynchronousTask`); this parameter is only used for EDF when dynamic power management is in effect, and can be set to 0 when no dynamic power management is in effect or if Deadline Monotonic scheduling is selected.
6. (**`void`** `*`) `argument`: An instance specific 16-bit value that is accessible when a task instance is active.
7. (`INT8`) `frequencyIndex`: Static frequency index of the task instances; this index corresponds to the speed that will be applied to the task when its instances are scheduled. This is the only power management scheme that is applied to the task.

Returned Value

(`BOOL`) `TRUE` if the event-driven task was successfully created and `FALSE` otherwise. The function can only fail when there is a memory allocation failure.

Restrictions

All restrictions described for `OSCreateSynchronousTask` also apply to `_OSCreateSynchronousTask`.

Notes

Each frequency index corresponds to an entry into a system table generated by ZottaOS Configurator Tool, and which contains the operating frequency settings along with the minimal voltages that can be applied to the core when task instances are scheduled. Together, these tuples allow slowdowns so that power savings can be realized.
Function `_OSCreateSynchronousTask` can only be used with microcontrollers having a power manager module. Currently, these microcontrollers are for some members of the MSP430F5xx family and all CC430xxx families.
The frequency indices that can be used are generated by the *ZottaOS Configurator Tool* and can be found in the header file ZottaOS_msp430x*ZZZ*.h or ZottaOS_cc430x*ZZZ*.h, where *ZZZ* is the family member of the microcontroller (for example 54xA).

Index	MSP430F5xx†	CC430xxx
OS_8MHZ_SPEED	✓	✓
OS_12MHZ_SPEED	✓	✓
OS_16MHZ_SPEED	undefined	✓
OS_20MHZ_SPEED	✓	✓
OS_25MHZ_SPEED	✓	undefined

† Applies for MSP430F54xxA and MSP430F55xx
A checkmark in the table indicates a valid frequency index.

See Also `CreateEventDescriptor`, `OSCreateSynchronousTask`, `OSSuspendSynchronousTask`, `OSScheduleSuspendedTask`

OSScheduleSuspendedTask: Signals a Suspended Event-Driven Task

Prototype `void OSScheduleSuspendedTask(void *event)`

Scope Applies to all kernel versions of *ZottaOS*.

Description Unblocks and schedules the first suspended event-driven task associated with a specified event. If at the time the function is invoked there are no suspended tasks associated with the event, the signal generated by this function is stored and when the next event-driven task terminates, it is immediately rescheduled.

Parameter `(void *) event`: An vvent descriptor returned by `OSCreateSynchronousTask` and associated to the event-driven tasks that are signaled.

Returned Value None

Restrictions When several event-driven tasks are suspended, the first task that invoked `OSSuspendSynchronousTask` is rescheduled. Task signaling is thus done in a FIFO manner regardless of the task's priorities. Although this may be the expected behavior under EDF scheduling considering the way event-driven task deadlines are determined at run-time (see `OSCreateSynchronousTask`), under DM scheduling it is currently not possible to signal a particular event-driven priority task level.

See Also `OSCreateSynchronousTask`, `_OSCreateSynchronousTask`, `OSSuspendSynchronousTask`

OSSuspendSynchronousTask: Event-Driven Task Termination

Prototype `void OSSuspendSynchronousTask(void)`

Scope Applies to all kernel versions of *ZottaOS*.

Description This is the very last instruction of an event-driven task. The purpose of this call is to finish the current instance of the caller.
This function checks if the calling instance should restart a new instance of itself (an event was signaled) or if it should suspend itself until the event is signaled by another task that calls `OSScheduleSuspendedTask`.
Any code appearing after the call will never be executed. Also, without this call, a task instance can never terminate and will overrun its allotted processor utilization.
This function also schedules the next ready task and sets its operating speed if a power management scheme is in effect.

Parameter None

Returned Value None

Restrictions This function can only be invoked from within a function acting as the code of an event-driven task. It can also be invoked from one of the functions called from an event-driven task.
There is no limit to the number of times this function can appear in the task's code.

```
void SignaledTask(void *arg)
{
   if (!globalCondition)
      OSScheduleSuspendedTask();  // Do not continue
   /* Continue executing if globalCondition is true. */
   /* Do something useful here. */
   OSScheduleSuspendedTask();
}
```

See Also `OSEndTask`, `OSCreateSynchronousTask`, `OSScheduleSuspendedTask`

Miscellaneous Functions: Memory Management, Event Creation, Processor Speed, Starting *ZottaOS*, and Atomic Instructions

OSMalloc: Memory Allocator

Prototype `void *OSMalloc(UINT16 size)`

Scope Applies to all kernel versions of *ZottaOS*.

Description Since most applications do all their dynamic memory allocations at initialization time and these allocations are never reclaimed, the standard `malloc` function provided by a run-time library is overkill because there is no need to manage freed blocks. To correct this state of affairs, *ZottaOS* provides the `OSMalloc` function, which follows a specification that is similar to the standard `malloc` function but works differently. At initialization time, i.e., when main is active and the first task has not yet executed, the run-time stack starts near the middle of the RAM memory and all functions that are called to prepare the application are successively stacked below main. Indeed, the stack expands towards decreasing addresses on most microcontrollers, as witnessed by the `push` instruction. During this time, all memory block allocated by `OSMalloc` to the application occupy successive addresses at the opposite end of RAM memory starting with the highest possible address. Once the kernel begins its execution, the run-time stack is adjusted so that its starting address is immediately below the last allocated memory block.
As well as providing a useful memory allocation function for user applications, this approach has several interesting advantages:

1. The code size is definitively smaller and faster compared to the standard thread-safe `malloc` and `free` pair of functions;
2. There can be no memory corruption in case of a stack overflow;
3. All variables declared in main and used to initialize the application are automatically wiped out as soon as *ZottaOS* starts and no RAM memory is wasted;
4. The stack overflows only when RAM memory is entirely full.

On the MSP430 line of microcontrollers, this is the only memory allocation function that can be provided. Because the MSP430 does not provide a frame pointer, other memory allocation functions such as the standard C `alloca` cannot be supplied.

Parameter (`UINT16`) `size`: Specifies the size (in bytes) of the allocated block of memory.

Returned Value The function returns a null pointer if there is insufficient memory. Otherwise the function returns a pointer to the first allocated memory cell.

Affected By `OSMALLOC_INTERNAL_HEAP_SIZE`: Initial byte size of the RAM memory allotted to dynamic memory allocations. This value can be found in the header file generated by *ZottaOSconf.exe* (either *ZottaOS_msp430XXX.h* or *ZottaOS_cc430XXX.h*, where *XXX* is the family member of the microcontroller at hand).

Restrictions Can only be called prior to `OSStartMultitasking`.

OSCreateEventDescriptor: Event Creation

Prototype `void *OSCreateEventDescriptor(void)`

Scope Applies to all kernel versions of *ZottaOS*.

Description Creates and returns an opaque descriptor with all the needed information to block and wake-up an event-driven task.
When an event-driven task is created or when its instance terminates, the task is inserted into a FIFO queue associated with the event. When function `OSScheduleSuspendedTask` is invoked, the next event-driven task in the queue is scheduled. If at the time of the call, there are no suspended tasks, the first event-driven task associated with the event and that calls `OSSuspendSynchronousTask` is immediately rescheduled.

Parameters None

Returned Value `(void *)` An opaque descriptor or `NULL` when a memory failure occurs.

Restrictions This function can only be invoked before starting *ZottaOS* and is usually called by the main program or one of the initializing functions called by main.
When associated with a FIFO queue or an asynchronous read-writer protocol, an event can be declared local to the function that calls `CreateEventDescriptor`. However, events are usually declared as global variables when called by a signaler task.

Note The descriptor of an event requires 9 + 2 N_a bytes of memory space where N_a represents the number of event-driven tasks associated with the event.

See Also `OSCreateSynchronousTask`, `OSScheduleSuspendedTask`

OSSetMinimalProcessorSpeed: Initialize And Set Minimal Processor Speed

Prototype `void OSSetMinimalProcessorSpeed(UINT8 speed)`

Description Sets the initial operating frequency of the processor and also sets the minimal speed that the processor should run. The function returns only when the specified processor speed is in effect.
This function should only be called prior to starting the kernel, i.e., before `OSStartMultitasking` and should never be called by an application task or by an interrupt ISR.

Parameter (`UINT8`) `speed`: The operating frequency setting. This value should be one of the values given in the following table. Currently this speed is defined for the Texas Instruments MSP430F5xx family of microcontrollers and can be one of the values defined in ZottaOS_5xx.h and given in the following table.

Index	MSP430F5xx	MSP430F5xxA
`OS_12MHZ_SPEED`	12 MHz	25 MHz
`OS_16MHZ_SPEED`	16 MHz	16 MHz
`OS_18MHZ_SPEED`	18 MHz	18 MHz
`OS_25MHZ_SPEED`	undefined	25 MHz

Returned Value None.

Notes The rational behind this function is twofold: (1) to provide a processor speed in order to carry out initializations while setting up the different tasks and initializing the peripherals; (2) to supply a minimum speed to execute while the processor is idle. Of course the smallest operating frequency is preferred for maximal power savings, but in case of an interrupt processing for which the response time may be penalized at the lowest speed, this function provides a mechanism to circumvent this problem.

When static frequencies are provided by the create task functions `_OSCreateTask` and `_OSCreateSynchronousTask`, no control is done at run-time to prevent the processor from slowing down to the specified setting of a task. It is therefore recommended that the final speed setting provided with `OSSetMinimalProcessorSpeed` be coherent with the specified task settings.

This function can be called any number of times prior to calling `OSStartMultitasking`.

See Also `_OSCreateTask`, `_OSCreateSynchronousTask`

OSStartMultitasking: Launching of *ZottaOS*

Prototype `BOOL OSStartMultitasking(void)`

Scope Applies to all kernel versions of *ZottaOS*.

Description This is the very last function to call in main and it gives control of the processor to the kernel and to the defined application tasks.
This function never returns and once the kernel gets control of the processor, it loops indefinitely switching between the user tasks. When no application task is created, an idle task gets control of the processor, and the only means by which an application can be executed is by processing interrupt defined before calling this function. Hence, the returned value is always `FALSE` and can never be `TRUE`.

Parameters None

Returned Value Always `FALSE`.

Load-Link: Load and Reserve Memory Location

Prototypes

```
UINT8 OSUINT8_LL(UINT8 *memAddr)
UINT16 OSUINT16_LL(UINT16 *memAddr)
```

Scope

Applies to all kernel versions of *ZottaOS*.

Description

The *LL* functions are used in conjunction with the *SC* functions call to provide synchronization support for *ZottaOS*. The *LL*/*SC* pair of functions works very much like simple get and set functions.
The *LL* functions, in addition to returning the contents of a memory location, have the effect of setting a user transparent reservation bit. If this bit is still set when an *SC* function is executed, the store of *SC* occurs; otherwise the store fails and the specified memory location is left unchanged (see `OSUINT8_SC` and `OSUINT16_SC`).
The *LL* function is semantically equivalent to the atomic execution of the following code:

```
type OStype_LL(type *memAddr) {
   reserveBit = true;
   return *memAddr;
}
```

where *type* can be one of `UINT8` or `UINT16`.

Parameter

(*type* `*`) `memAddr`: Address to a memory location that holds the value to read, where *type* can be one of `UINT8` or `UINT16`.

Returned Value

The function returns the content of the specified memory location.

Notes and Restrictions

The *LL* and *SC* functions need not be paired. In particular a conditional branch statement may follow an *LL* where on the fall-through path an *SC* is executed, whereas on the taken path no matching *SC* is executed.
If two *LL* functions execute with no intervening *SC* function call, the second *LL* overwrites the state of the first *LL*. If two *SC* function calls execute with no intervening *LL* call, the second *SC* always fails because the first clears the reservation bit.
The kernel clears the reservation bit on each context switch. It does this to ensure that a task doesn't use the reservation flag owned by another task.
On microcontrollers that do not provide *LL*/*SC* function pairs as part of their instruction set, the functions are emulated. On the MSP430 line of microcontrollers, which does not have any atomic instruction, function LL is implemented in assembler and requires 18 machine cycles (including the call and the return) of which 12 machine cycles are in a critical section.

Sample Code

The sequence

```
do {
   x = OSUINT16_LL(mem);
   y = f(x);
} while (!OSUINT16_SC(mem,y));
```

does an atomic read-modify-write of a shared `UINT16` datum until the `OSUINT16_SC` returns `TRUE`. The loop is repeated until the `OSUINT16_SC` succeeds.
Another example is the equivalent of an atomic swap between two memory locations. The following function assumes that the second memory location is not subject to contention.

```
void Swap(UINT16 *mem1, UINT16 *mem2) {
  UINT16 tmp1, tmp2 = *mem2;
  do {
     tmp1 = OSUINT16_LL(mem1);
  } while (!OSUINT16_SC(mem1,tmp2));
  *mem2 = tmp1;
}
```

See Also `OSUINT8_SC`, `OSUINT16_SC`

Store Conditional: Store Memory Location If Reserved

Prototypes

```
BOOL OSUINT8_SC(UINT8 *memAddr, UINT8 newValue)
BOOL OSUINT16_SC(UINT16 *memAddr, UINT16 newValue)
```

Scope

Applies to all kernel versions of *ZottaOS*.

Description

If the reservation bit is set by a previous call to an *LL* function, the second parameter is written into the memory location specified by the first parameter.
The *SC* functions are semantically equivalent to the atomic execution of the code given below:

```
BOOL OStype_SC(type *memAddr, type newValue) {
  if (reserveBit) {
     *memAddr = newValue
     reserveBit = false;
     return true;
  }
  else
     return false;
}
```

where *type* is one of UINT8 or UINT16.

Parameters

1. (*type* *) memAddr: Address to a memory location that holds the value to modify, where *type* can be one of UINT8 or UINT16.
2. (*type*) newValue: Value to insert into the memory location specified by the above parameter if and only if the reservation bit is still set.

Returned Value

(BOOL) The functions return TRUE if the store operation succeeds and FALSE otherwise.

Note

On the MSP430 line of microcontrollers, which does not have any atomic instruction, the SC instruction is implemented as a function in assembler and has a maximal path length of 25 machine cycles (including the call and the return) of which 19 machine cycles are in a critical section.

See Also

OSUINT8_LL, OSUINT16_LL

Concurrent FIFO Queue API

OSInitFIFOQueue: Create Concurrent FIFO Queue

Prototype `void *OSInitFIFOQueue(UINT8 maxNodes, UINT8 maxNodeSize)`

Scope Applies to all kernel versions of *ZottaOS*.

Description This function creates and initializes a concurrent FIFO queue that can be used for inter-task communications following a multiple producer and consumer paradigm. To guarantee that there will be sufficient memory when items are created and filled before being enqueued, a free list of unused buffers is also created and associated with the created queue. To use these, a task should first obtain a buffer via function `OSGetFreeNodeFIFO`, fill the buffer with its data and then enqueue the node via `OSEnqueueFIFO` into the same FIFO queue it got the free buffer from. When dequeued buffers are no longer in use, these should be released via `OSReleaseNodeFIFO` so that the buffer can be recycled.

Parameters
1. (`UINT8`) `maxNodes`: Maximum number of buffers that can be enqueued. This value also corresponds to the initial number of free buffers that are created and inserted into the free list associated with the FIFO queue.
2. (`UINT8`) `maxNodeSize`: Byte size of the buffers that are in the free list.

Returned Value (**`void`** *) On success, the descriptor of the created queue is returned. On memory allocation failure, the function returns `NULL`.

Notes In order to use a FIFO queue, the very first operation is to create it. It is best to do this operation prior to starting *ZottaOS* since the queue should exist prior to performing any action on it and namely from a peripheral handler. Usually, memory blocks are dynamically created by an enqueuer task and later freed by a dequeuer task once the memory block is no longer needed. Memory allocation and reclamation are non-determinist operations and are not recommended for real-time operating systems like *ZottaOS*. Instead, real-time systems favor pre-allocations of memory pools that are created once and for all.
The queue implementation is based on a deterministic wait-free circular array algorithm.

Restrictions When using multiple FIFO queues, an available buffer taken from one queue, must be released to the same queue.
The data content size of a buffer is bounded when the FIFO queue is created.

See Also `OSEnqueueFIFO`, `OSDequeueFIFO`, `OSGetFreeNodeFIFO`, `OSReleaseNodeFIFO`

OSEnqueueFIFO: Insert Buffer into FIFO Queue

Prototype `BOOL OSEnqueueFIFO(void *fifoQueue, void *node, UINT16 size)`

Scope Applies to all kernel versions of *ZottaOS*.

Description This function inserts a buffer obtained from `OSGetFreeNodeFIFO` into a FIFO queue.

Parameters

1. (**`void *`**) `fifoQueue`: Opaque FIFO queue descriptor that was created by function `OSInitFIFOQueue`.
2. (**`void *`**) `node`: Buffer to insert into the queue.
3. (`UINT16`) `size`: Useful buffer byte size. The value of this parameter is returned with `OSDequeueFIFO` and should not be confused with the maximum size of each node created when calling `OSInitFIFOQueue`. Note that if an application uses constant sized information in its buffers or if an identifier can determine the useful information contained within the node, this parameter can also denote a message type.

Returned Value (`BOOL`) `TRUE` if the buffer has been successfully inserted into the queue. Otherwise the function returns `FALSE` to indicate that the queue is full at the time of the call. This last result can only occur if the inserted node was not obtained from the queue it is being inserted into (also see `OSGetFreeNodeFIFO`).

See Also `OSGetFreeNodeFIFO`, `OSDequeueFIFO`

OSDequeueFIFO: Return Buffer Stored in the FIFO Queue

Prototype `void *OSDequeueFIFO(void *fifoQueue, UINT16 *size)`

Scope Applies to all kernel versions of *ZottaOS*.

Description Removes and returns the oldest stored node (buffer) from a FIFO queue.

Parameters

1. (`void *`) `fifoQueue`: Opaque FIFO queue descriptor that was created by function `OSInitFIFOQueue`.
2. (`UINT16 *`) `size`: Output parameter denoting the useful data byte size of the returned node, if the returned value is different than `NULL` (also see `OSEnqueueFIFO`).

Returned Value `NULL` if the queue is empty at the time of the call and no buffer is returned. Otherwise the returned value corresponds to the starting address of the contents held in the buffer.

Note Because the number of nodes is restricted, it is important that the returned buffers are released as soon as possible with `OSReleaseNodeFIFO`.

See Also `OSEnqueueFIFO`, `OSReleaseNodeFIFO`

OSGetFreeNodeFIFO: Get Free Buffer

Prototype `void *OSGetFreeNodeFIFO(void *fifoQueue)`

Scope Applies to all kernel versions of *ZottaOS*.

Description This function returns a free buffer that an application can fill with useful data and then insert into a FIFO queue. The free buffer should come from a pool of buffers that is associated with the same FIFO queue into which the enqueue operation will insert the buffer.

Parameter (`void *`) `fifoQueue`: Opaque FIFO queue descriptor that was created by function `OSInitFIFOQueue`.

Returned Value The function returns `NULL` when all created nodes (buffers) are in use and none is available at the time of the call. Otherwise the function returns the starting address of the buffer that can be filled. This same address can then be transferred to `OSEnqueueFIFO` so that this function can insert the buffer into the FIFO queue, or if can be released to the free pool of buffers with function `OSReleaseNodeFIFO`.

Restriction The free buffer retrieved from the FIFO queue should not be released to another FIFO queue. Failure to comply with this restriction may result into lost buffers.
If enqueue operations are done to the same FIFO queue where the free buffers are retrieved from, enqueue operations never fail.

See Also `OSInitFIFOQueue`, `OSEnqueueFIFO`, `OSReleaseNodeFIFO`

OSReleaseNodeFIFO: Free Consumed Buffer

Prototype `void OSReleaseNodeFIFO(void *fifoQueue, void *releasedNode)`

Scope Applies to all kernel versions of *ZottaOS*.

Description This function releases a buffer and returns it to the pool of available buffers associated with a FIFO queue for subsequent reuse.

Parameters
1. (`void *`) `fifoQueue`: Opaque FIFO queue descriptor that was created by function `OSInitFIFOQueue`.
2. (`void *`) `releasedNode`: Address of the buffer to release.

Returned Value None.

Restrictions The released buffer should be done to the same FIFO that the buffer was retrieved from with `OSGetFreeNodeFIFO`. Failure to do so, results into buffer losses if the pool of free buffers is full at the time of the call.
Once released, the buffer contents must neither be accessed nor modified.

See Also `OSInitFIFOQueue`, `OSGetFreeNodeFIFO`

Asynchronous Reader-Writer Functions

OSInitBuffer: Asynchronous Reader-Writer Initialization

Prototype `void *OSInitBuffer(UINT8 slotSize, UINT8 bufType, void *event)`

Scope Applies to all kernel versions of *ZottaOS*.

Description This function creates a 3- or 4-slot buffer that can be used for I/Os and inter-task communications following a single-writer-single-reader asynchronous protocol. The appropriate time to call this function is in the main program prior to calling `OSStart-Multitasking`, and to save the returned value in a global variable that is then shared by the communicating application tasks.

Parameters

1. (`UINT8`) `slotSize`: Required size of each slot; once the writer has completed a full slot (see function `OSWriteBuffer`), this slot become eligible for reading. It is thus recommended that the specified size of the slots be equal to those that are read as a chunk by one of the reading functions to prevent data overlap over several slots.
2. (`UINT8`) `bufType`: This can be one of the following 2 values:
   - `OS_BUFFER_TYPE_4_SLOT`: Specifies a 4-slot mechanism that is appropriate for small-sized data requiring fast transfer between a writer task and a reader task. This mechanism does not need any atomic instruction and is well suited for microcontrollers without any atomic instruction support.
   - `OS_BUFFER_TYPE_3_SLOT`: Specifies a 3-slot mechanism that is more memory efficient but somewhat slower because of a needed atomic instruction. When no atomic instruction is defined in the instruction set of the microcontroller, these are emulated by ways of masking interrupts.
3. (**`void *`**) `event`: An optional event, which when specified can be used to signal an event-driven reader task as soon as a complete slot becomes available for it. This event must have been created with `OSCreateEventDescriptor`.

Returned Value (**`void *`**) On success, the function returns an opaque descriptor holding all state and buffer information that can be used for device or for task communications.
On memory allocation failure, the function returns `NULL`.

Note The data memory usage is shown in the following table.

Mechanism	Memory Requirements
`OS_BUFFER_TYPE_3_SLOT`	21 + 3 × `slotSize`
`OS_BUFFER_TYPE_4_SLOT`	27 + 4 × `slotSize`

See Also `OSWriteBuffer`, `OSGetCopyBuffer`, `OSGetReferenceBuffer`

OSGetCopyBuffer: Retrieve a Filled Slot

Prototype `UINT8 OSGetCopyBuffer(void *des, UINT8 readMode, UINT8 *data)`

Scope Applies to all kernel versions of *ZottaOS*.

Description Returns a copy of the most recently filled slot completed by `OSWriteBuffer`.

Parameters

1. (`void *`) `des`: Opaque descriptor created and returned by `OSInitBuffer`.
2. (`UINT8`) `readMode`: Specifies whether the latest available slot should be read only once or if multiple reads of the same slot are permitted.
   - `OS_READ_ONLY_ONCE` indicates that only the last and most recently unread slot can be read. Calling `OSGetCopyBuffer` with this flag marks the slot so that the same slot can no longer be read neither with this function nor with `OSGetReferenceBuffer` if this flag is specified. This mode is useful when the reader task needs to consume the data held in the slot only once.
   - `OS_READ_MULTIPLE` indicates that the most recently available slot should be read. Calling `OSGetCopyBuffer` with this flag does not mark the slot and the same slot can be read more than once. If at the time `OSGetCopyBuffer` is invoked, the most recent slot is already marked by a previous call to either `OSGetCopyBuffer` or `OSGetReferenceBuffer` with flag `OS_READ_ONLY_ONCE`, the function still succeeds and returns the data held in the slot. In this case the slot marking remains untouched.
3. (`UINT8 *`) `data`: Data buffer where data is to be copied. This buffer should be greater or equal to the slot size specified by `OSInitBuffer` when the slotted-buffer was created.

Returned Value (`UINT8`) Number of bytes copied into the specified data buffer.
The only time that the function can return 0 when `OS_READ_MULTIPLE` is specified is when `OSWriteBuffer` has not yet completely filled its very first slot.

See Also `OSInitBuffer`, `OSGetReferenceBuffer`

OSGetReferenceBuffer: Retrieve Reference to a Filled Slot

Prototype

```
UINT8 OSGetReferenceBuffer(void *descriptor, UINT8 readMode,
   UINT8 **data)
```

Scope

Applies to all kernel versions of *ZottaOS*.

Description

Returns a pointer to the most recently filled slot that can be read.

Parameters

1. (**void** *) descriptor: Opaque buffer descriptor created and returned by OSInitBuffer.
2. (UINT8) readMode: Specifies one of the 2 possible read modes:
   - OS_READ_ONLY_ONCE indicates that only the last and most recently unread slot can be read. Calling OSGetReferenceBuffer with this flag marks the slot so that the same slot can no longer be read neither with this function nor with OSGetCopyBuffer if this flag is specified. This mode is useful when the reader task needs to consume the data held in the slot only once.
   - OS_READ_MULTIPLE indicates that the most recently available slot should be read. Calling OSGetReferenceBuffer with this flag does not mark the slot and the same slot can be read more than once. If at the time OSGetReferenceBuffer is invoked, the most recent slot is already marked by a previous call to either OSGetReferenceBuffer or OSGetCopyBuffer with flag OS_READ_ONLY_ONCE, the function still succeeds and returns the data held in the slot. In this case the slot marking remains untouched.
3. (UINT8 **) data: Pointer to one of the slots holding the data. This is a result parameter.

Returned Value

(UINT8) The return value is the number of bytes that is available in the returned data buffer. When there is nothing to read (OSWriteBuffer has never completed its first slot) or if the data has already been read (read mode set to OS_READ_ONLY_ONCE), 0 is returned and the data buffer passed as argument is set to NULL.

Notes

The implemented asynchronous read-writer mechanism guarantees that OSWriteBuffer never overwrites the current reader-slot. Hence the returned slot may be modified and accessed by the caller up until the next call to either OSGetReferenceBuffer or OSGetCopyBuffer.
The only time that the function can return 0 when OS_READ_MULTIPLE is specified is when OSWriteBuffer has not yet completely filled its very first slot.

See Also

OSInitBuffer, OSGetCopyBuffer

OSWriteBuffer: Insert Data into Slot

Prototype `UINT8 OSWriteBuffer(void *descriptor, UINT8 *data, UINT8 size)`

Scope Applies to all kernel versions of *ZottaOS*.

Description Copies data into the current writer-slot. Once the current slot becomes full, i.e., when it reaches the size specified with `OSInitBuffer`, the writer-slot becomes available to the reader task, and a new slot is chosen for the next time this function is called.
When the amount of data to be copied exceeds the available space of the current slot, only the first bytes of the data that completely fill the slot are copied. The remaining bytes are not considered.

Parameters

1. (`void *`) `descriptor`: A slotted-buffer descriptor created by `OSInitBuffer`.
2. (`UINT8 *`) `data`: Data to copy into the current writer-slot.
3. (`UINT8`) `size`: Maximum number of data bytes to copy from the 2nd argument.

Returned Value (`UINT8`) This function returns number of bytes actually accepted into the slot. The returned value may not exceed the slot size specified by the first parameter of `OSInitBuffer`.
Unless the slotted-buffer descriptor is `NULL` or the size parameter of this function is equal to 0, the returned value is always greater than or equal to 0.

See Also `OSInitBuffer`

Interrupt Subsystem Functions

OSGetISRDescriptor: Retrieve Registered ISR Descriptor

Prototype `void *OSGetISRDescriptor(UINT8 index)`

Scope Applies to all kernel versions of *ZottaOS*.

Description This function returns the ISR descriptor bound to a device interrupt entry.
This function is made available to application tasks so that inserted descriptors can be returned to the user in order to read or to store application data that can later be used by an interrupt handler.

Parameter (`UINT8`) `index`: Entry of the device interrupt vector where the descriptor is held. A full description of the possible index values is given in ZottaOS_5xx.h and these are defined as `OS_IO_xx` where `xx` denotes the peripheral device.

Returned Value (**`void *`**) The requested ISR descriptor is returned. If no previous call to `OSSetISR-Descriptor` was previously made for the specified interrupt, the returned value is undefined.

See Also File ZottaOS_5xx.h provided with the distribution
`OSSetISRDescriptor`

OSSetISRDescriptor: Bind Interrupt Source with Handler

Prototype `void OSSetISRDescriptor(UINT8 index, void *descriptor)`

Scope Applies to all kernel versions of *ZottaOS*.

Description This function binds an interrupt descriptor to a particular interrupt source, which is usually an I/O device. The very first field of the descriptor is a pointer to a user-defined interrupt service routine that is invoked when the interrupt event specified by the index is signaled.

Parameters

1. (`UINT8`) `index`: Entry to an interrupt vector and identifies the interrupt to bind. A full description of the possible index values is given in ZottaOS_5xx.h and these are defined as `OS_IO_xx` where `xx` denotes the peripheral device.
2. (**`void *`**) `descriptor`: Structure holding all the need information to service the interrupt. The very first field of the structure is always a pointer to the ISR, while the following fields are user-defined and can hold additional information which otherwise would have been declared as global variables.
   The template of this descriptor is

```
typedef struct myDescriptor {
  void (*myISR)(struct myDescriptor *);
  // other information used by myISR
} myDescriptor;
```

   and the ISR has the following prototype is

```
void MyISR(struct myDescriptor *des);
```

Returned Value None

Restrictions A binding can be established anywhere and at anytime. The only precaution to take is that the memory allotted to the descriptor exists. In other words, the descriptor should either be a global (static) variable or dynamically allocated by `OSMalloc` prior to starting *ZottaOS*.

See Also `OSMalloc`, `OSGetISRDescriptor`

Simple UART API

OSInitTransmitUART: Initialize Transmitter Part of a UART

Prototype

```
BOOL OSInitTransmitUART(UINT8 maxNodes, UINT8 maxNodeSize,
                        UINT8 interruptIndex)
```

Scope

Applies to all kernel versions of *ZottaOS*.

Header File

`UART/UART.h` contained in folder `ZottaOS`.

Description

This function is called at task creation time in the main program to initialize the internal data structure associated with the transmitter part of a UART device. The function initializes the kernel internals in order to store the bytes that are to be sent on a specified output port and must be called before doing any I/O related with the UART. It can be called before or after creating the application tasks but never from an application task.
The transmission baud rate of the UART must be done by the application, typically in the main program. The kernel does not provide a function to accomplish this task because low-level hardware settings depend on the particular microcontroller at hand.

Parameters

1. (`UINT8`) `maxNodes`: Maximum number of outstanding messages (or message parts) that can be enqueued at once. This value also corresponds to the initial number of free message buffers that are created and inserted into the free list associated with a particular UART device.
2. (`UINT8`) `maxNodeSize`: Byte size of the message buffers that are in the free list. This value also corresponds to the maximum number of bytes that can be inserted into a message that the UART can transmit.
3. (`UINT8`) `interruptIndex`: Entry to the interrupt table corresponding to the transmitter part of a specific UART.

Returned Value

(`BOOL`) `TRUE` on success and `FALSE` if there is insufficient memory to create the descriptor or the buffers that will be used in connection with the UART.

Notes

The idea behind the implemented API is to associate a FIFO queue of messages to the transmitter part of the UART. With this API, we believe that developing an application requiring UART transmission can be done in record time, as the designer would only need to concentrate on the output protocol.
Although the implementation is fully concurrent, the UART does not prevent message parts from being interleaved. Therefore `maxNodeSize` should be the maximal size of a message that is filled by an application task and transferred to the UART. However as the implementation guarantees a FIFO processing of the messages transferred to the UART, a long message can be broken into smaller parts when there is a single application task that transmits messages through the UART.

Sample

Suppose that we have a task that produces a UINT16 number every second and then transmits it as a string of hexadecimal characters. Without elaborating on how these numbers are actually produced and focusing on the transmission part of this application, the main program initializes a UART with 3 buffers of 4 bytes each and which sends bytes at 9600 Bauds. In this example we also suppose that these characters are sent to a big endian processor such as a PowerPC, then bytes have to be inverted, sent from last to first.

```
#define PENDINGNUMBERS 3 /* Allow up to 3 pending transmissions */

/* Task that actually produces the 16-bit unsigned numbers */
void ProduceNumberTask(void *arg);
```

```
int main(void)
{
  // Do other initializations here

  /* Initialize ZottaOS output UART driver (USCI A0 on MSP430x5xx)*/
  OSInitTransmitUART(PENDINGNUMBERS,4,OS_IO_USCI_A0_TX);

  /* Initialize USCI A0 for transmission */
  UCA0CTL1 |= UCSWRST;   // Put state machine in reset
  UCA0CTL1 |= UCSSEL_1;  // Select ACLK (32,768 kHz) as clock source
  UCA0BR0 = 3;           // 9600 Baud -> UCBRx=3, UCBRSx=3, UCBRFx=0
  UCA0BR1 = 0;
  UCA0MCTL |= UCBRS_3 + UCBRF_0;
  UCA0CTL1 &= ~UCSWRST;  // Initialize USCI state machine

  /* Create a periodic task that produces a number and sends it
  ** every second. */
  OSCreateTask(ProduceNumberTask,0,32768,32768,NULL);
  return OSStartMultitasking();
} /* end of main */

void ProduceNumberTask(void *arg)
{
  UINT16 number;    // Number that is produced
  UINT8 i, j, n;
  UCHAR *convert;   // Allows byte extractions
  char *sendBuffer; // Buffer that will hold the number

  /* Get a free buffer to insert the number */
  sendBuffer = (char *)OSGetFreeNodeUART(OS_IO_USCI_A0_TX);
  if (sendBuffer == NULL) // Skip this number if we can't send it
     OSEndTask();

  // Produce UINT16 number to send

  // Insert number into sendBuffer formatted as hexadecimal chars
  convert = (UCHAR *)&number;
  // Start with most significant byte first (MSP430 is little endian)
  for (i = 1, j = 0; i >= 0; i--) {
     n = (convert[i] >> 4) & 0x0F;  // Insert upper nibble
     sendBuffer[j++] = (n < 10 ? '0':'A'-10) + n;
     n = convert[i] & 0x0F;         // Insert lower nibble
     sendBuffer[j++] = (n < 10 ? '0':'A'-10) + n;
  }
  /* Send the number */
  OSEnqueueUART(sendBuffer,4,OS_IO_USCI_A0_TX);
  OSEndTask();
} /* end of ProduceNumberTask */
```

See Also OSEnqueueUART, OSGetFreeNodeUART, OSReleaseNodeUART

OSEnqueueUART: Add Message to UART Output Transmitter Queue

Prototype

```
void OSEnqueueUART(void *buffer, UINT8 dataSize,
                   UINT8 transmitInterruptIndex)
```

Scope Applies to all kernel versions of *ZottaOS*.

Header File `UART/UART.h` contained in folder `ZottaOS`.

Description This function is called by an application task to output a string of bytes contained in a buffer that was supplied by function `OSGetFreeNodeUART`. The buffer that is passed with the call must not be freed with `OSReleaseNodeUART` after the call as it is given to the transmitter part of the UART and which will later free it once its contents has completely been transferred.

Parameters

1. (**`void`** `*`) `buffer`: Byte contents to output from the UART.
2. (`UINT8`) `dataSize`: Number of bytes to consider from 1st parameter.
3. (`UINT8`) `transmitInterruptIndex`: Entry to the interrupt table corresponding the transmitter part of the UART device for which the bytes are intended. This index is used to extract the transmitter descriptor, which contains the FIFO onto which the buffer is added to, and to initiate the transmission if the device is idle.

Returned Value None. The function always succeeds as memory availability is checked when obtaining the buffer from `OSGetFreeNodeUART`.

Notes It is important that the buffer supplied with this call originally came from `OSGetFreeNodeUART` for the transmitter part of the UART device. Failure to comply with this restriction will result into an erratic behavior.

See Also `OSInitTransmitUART`, `OSGetFreeNodeUART`

OSGetFreeNodeUART: Get Buffer For New Message To Transmit

Prototype `void *OSGetFreeNodeUART(UINT8 transmitInterruptIndex)`

Scope Applies to all kernel versions of *ZottaOS*.

Header File `UART/UART.h` contained in folder `ZottaOS`.

Description This function returns a free buffer that an application can fill with useful data and then transfer it to `OSEnqueueUART`. The free buffer should come from a pool of buffers that is associated with the same UART transmitter queue for which the application will later invoke an `OSEnqueueUART` operation.

Parameter (`UINT8`) `transmitInterruptIndex`: Entry to the interrupt table corresponding to the transmitter part of the UART device for which the buffer will later be transferred to. This index should correspond to the third parameter of function `OSEnqueueUART`.

Returned Value (**`void *`**) If all the pre-allocated buffers created when calling `OSInitTransmitUART` are in use, `OSGetFreeNodeUART` returns `NULL`. Otherwise the function returns the starting address of the buffer that can be filled. This same address can then be transferred to `OSEnqueueUART`, or it can be released to the free pool of buffers by calling function `OSReleaseNodeUART`.

See Also `OSInitTransmitUART`, `OSEnqueueUART`, `OSReleaseNodeUART`

OSReleaseNodeUART: Free Message Buffer

Prototype

```
void OSReleaseNodeUART(void *buffer,
                       UINT8 transmitInterruptIndex)
```

Scope Applies to all kernel versions of *ZottaOS*.

Header File `UART/UART.h` contained in folder `ZottaOS`.

Description This function releases a buffer that was acquired from `OSGetFreeNodeUART` and returns it to the pool of available buffers associated with a UART. This function is provided when the logic of the program first obtains a buffer from `OSGetFreeNodeUART` to later realize that the buffer cannot be used.

Parameters

1. (**`void`** `*`) `buffer`: Pointer to the buffer to free.
2. (`UINT8`) `transmitInterruptIndex`: Entry to the interrupt table corresponding to the transmitter part of the UART device from which the buffer was received. This value must correspond to the `transmitInterruptIndex` parameter of function `OSGetFreeNodeUART`, which provided the buffer.

Returned Value None

Sample Suppose that several tasks can concurrently forward two 16-bit integers to the output port of USART0 but only one of these tasks must actually perform the work. A global flag is used to indicate the state of these integers, where `TRUE` signifies that the integers should be sent and `FALSE` that the integers have already been sent.
To avoid acquiring a buffer after signaling to the other tasks that the integers are being taken care, and then realizing that there is no available buffer to store the integers, the operations done by the task are reversed. A task therefore obtains a buffer, and only then doe it indicate that it is the one that processes that integers. Proceeding in this order guarantees that the integers will be sent by one of the tasks.

```
static BOOL global = FALSE;

void ConcurrentPeriodicTask(void *arg)
{
  INT16 *buffer;
  BOOL continue;
  ...
  buffer = (INT16 *)OSGetFreeNodeUART(OS_IO_USART0TX);
  if (buffer != NULL) {
     /* Indicate to the other tasks that the 2 integers will
     ** be taken care of. */
     while (continue = OSUINT8_LL(&global))
        if (OSUINT8_SC(&global,FALSE)) break;
     /* At this point, continue can either be TRUE or FALSE. */
     if (continue) {
        /* Fill buffer with the 2 integers. */
        /* And then pass it on to OS_IO_USART0TX */
        OSEnqueueUART(buffer,2*sizeof(INT16),OS_IO_USART0TX);
     }
     else // Release unused buffer and quit
        OSReleaseNodeUART(buffer,OS_IO_USART0TX);
  }
  OSEndTask();
}
```

Restriction Once released, the buffer contents must neither be accessed nor modified.

See Also `OSGetFreeNodeUART`

OSInitReceiveUART: Initialize Receiver Part of a UART

Prototype

```
BOOL OSInitReceiveUART(BOOL (*userReceiveHandler)(UINT8),
                       UINT8 interruptIndex)
```

Scope

Applies to all kernel versions of *ZottaOS*.

Header File

`UART/UART.h` contained in folder `ZottaOS`.

Description

This function is called at task creation time from the main program to initialize the internal data structure associated with the receiver part of a UART device. This function must be called before doing any input related with the UART and can be called before or after creating the application tasks but never from an application task. Low-level protocol, e.g. RS232 or I2C, and reception baud rate of an I/O port must be done by the application, typically in the main program. The kernel does not provide a function to accomplish this task because low-level hardware settings depend on the particular microcontroller at hand.

Parameters

1. (`BOOL (*userReceiveHandler) (UINT8)`): An ISR function that is called when the UART receives a new byte from its input port. This function is called every time an interrupt for receiver part of the UART is raised so that it can process the byte that was retrieved from the UART's internal buffer. This function should return `TRUE` to re-enable interrupts from the UART and `FALSE` if interrupts should not be re-enabled.
2. (`UINT8`) `interruptIndex`: Entry to the interrupt table corresponding to the receiver part of a specific UART device.

Returned Value

(`BOOL`) `TRUE` on success and `FALSE` if there's insufficient memory to create the descriptor or the buffers that will be used in connection with the UART.

Sample

Suppose that USART0 receives unsigned integers that are transmitted in ASCII and that these numbers are processed one at a time. Each number ends with the ASCII character end-of-text (ETX), which has the value 0x03, and as soon as a number is received, it is processed by some task called `ProcessNumberTask`.
In this example, task `ProcessNumberTask` operates on the numbers while the next one is being filled by the UART interrupt handler. To simplify the code, we assume that there only digits and ETX can be received by the UART.

```
static UINT32 number = 0;          // Number to process
static void *completeNumberEvent;  // Number received event
/* Handler that receives a number, one byte at the time. */
static BOOL UARTReceiveNumberHandler(UINT8 data);
/* Task that processes the received numbers. */
static void ProcessNumberTask(void *arg);

/* Function that creates the different entities of this example and
** that is called from main. */
BOOL CreateNumberReader(void)
{
  /* Create the event to wake up ProcessNumberTask. */
  completeNumberEvent = OSCreateEventDescriptor();
  if (completeNumberEvent != NULL) {
     // Initialize UART receiver that reads ASCII characters and
     // converts them into a number.
     OSInitReceiveUART(UARTReceiveNumberHandler,OS_IO_USART0RX);
     // Create the task that processes these numbers.
     return OSCreateSynchronousTask(ProcessNumberTask,50,
                                    completeNumberEvent,NULL);
  }
  return FALSE; // Insufficient memory
} /* end of CreateNumberReader */
```

```
BOOL UARTReceiveNumberHandler(UINT8 data)
{
  if (data == 3) {
     /* End of text received: signal that the current number is
     ** ready. */
     OSScheduleSuspendedTask(completeNumberEvent);
     return FALSE; // Disable receive UART interrupts
  }
  else {
     number = number * 10 + (data - '0');
     return TRUE; // Re-enable receive UART interrupts
  }
} /* end of UARTReceiveNumberHandler */

void ProcessNumberTask(void *arg)
{
  UINT32 thisNumber = number; // Make a copy of the current number
  number = 0;                 // Prepare for next integer input
  OSEnableReceiveInterruptUART(OS_IO_USART0RX);
  /* Do something useful here with thisNumber while the next number
  ** is being received concurrently with the processing of the
  ** current one. */
  ...
  /* Wait until next number is ready. */
  OSSuspendSynchronousTask();
} /* end of ProcessNumberTask */
```

Note that global static variable `number` is explicitly initialized to 0. This is because TI Code Composer does not, without any special adjustment, abide by the ANSI C standard of loading a zero value for global static when the variable is declared without an initializer.

See Also `OSEnableReceiveInterruptUART`

OSEnableReceiveInterruptUART: Re-enable UART Receiver Interrupts

Prototype `void OSEnableReceiveInterruptUART(UINT8 receiveInterruptIndex)`

Scope Applies to all kernel versions of *ZottaOS*.

Header File `UART/UART.h` contained in folder `ZottaOS`.

Description This function re-enables receive interrupts for a particular UART.

Parameter (`UINT8`) `receiveInterruptIndex`: Entry to the interrupt table corresponding to the receiver part of the UART device.

Returned Value None.

See Also `OSInitReceiveUART`