Supporting the migration between ‘event triggered’ and ‘time triggered’ software architectures: A small pattern collection intended for use by the developers of reliable embedded systems

Farah Lakhani, Huiyan Wang and Michael J. Pont

Embedded Systems Research Group,
University of Leicester,
University Road,
Leicester LE1 7RH
UK

Technical report
ESRG 2012-07-01 (Version 1.0)

Table of Contents

1. Introduction

Many complex embedded systems (in areas such as aerospace and defence, for example) have a long service life, typically measured in decades. During this life, planned product maintenance and upgrades will inevitably require software changes. Changes to available hardware platforms are also very common over the lifetime of such systems: this too will force some degree of software change. In addition, software for future versions of systems and new products will very rarely be created from scratch: instead, existing software will be adapted to match new requirements (such as a higher "Safety Integrity Level").

In this report, we introduce a small collection of patterns which is intended to support the migration of existing software designs to a "time triggered" architecture, in order to make the system behaviour more predictable and therefore support test and verification activities. The overall goal is to support improvements in system reliability (and – where appropriate – reduce certification effort).

In the next section, we explain (briefly) the meaning of the phrase "design pattern". We then summarise the features of the two software architectures ("event triggered" and "time triggered") which lie at the heart of this pattern collection.

2. Design patterns

The concept of design patterns first emerged from the work of an architect, Christopher Alexander, during the 1960s and 1970s. Alexander and his colleagues published three pioneering texts (Alexander, Silverstein et al., 1975; Alexander, Ishikawa et al., 1977; Alexander, 1979) between 1975 and 1979 that laid the foundation of use of patterns in the field of architecture. He defines a pattern as "a three part rule which expresses a relation between a certain context, a problem and a solution". The general nature of this concept makes design patterns a useful tool beyond the architecture. In particular, Alexander's techniques have been adopted by the software-engineering community.

Ward Cunningham and Kent Beck introduced the first software pattern language (Cunningham, 1987) which consisted of 5 design patterns intended to be used by Smalltalk programmers who were designing graphical user interfaces. The pattern collection by Gamma et al (Gamma, Helm et al., 1995) for object-oriented programming and has been very influential. Examples of patterns for telecommunication includes the work of Hanmer (Hanmer and Stymfal, 2000; Hanmer, 2007) which focuses on fault-tolerant software systems, and the work of Linda Rising (Rising, 2001). There are examples of organisational patterns by Cain, Coplien and Harrison (Cain, Coplien et al., 1996): these seek to document 'best practice' for productive software development from successful organisations. There have also been patterns produced which describe how to introduce new ideas into an organisation (Ramachandran, Fujiwara et al., 2006).

In the field of embedded systems, most previous work with design patterns has focused on the process of system construction (see, for example (Pont, 2001)). In this report, we present a small pattern collection which is intended to support the process of system migration. Our particular concern is to explore ways in which we may be able to help developers of embedded systems to improve system reliability by migrating between "event triggered" (ET) architectures and equivalent “time triggered” (TT) architectures: characteristics of these different architectures are summarised in the next section.

3. ET and TT architectures

In the majority of embedded systems, some form of scheduler will be employed to decide when tasks should be executed. These decisions may be made in an “event-triggered” fashion (i.e. in response to sporadic events) (Kopetz, 1991) or in a “time-triggered” fashion (i.e. in response to pre-determined lapses in time) (Kopetz, 1997). When a task is due to execute, the scheduler can pre-empt the currently executing task or wait for the executing task to relinquish control co-operatively.

There are links between the algorithms used to schedule task execution and the choice between co-operative and pre-emptive task execution: for example, in most (but not all) cases, event-triggered task scheduling is associated with task pre-emption, while many (but not all) time-triggered designs employ co-operative tasks. In this paper, it will be assumed that the event-triggered designs involve task pre-emption. ET and TT architectures have been compared in previous studies (Kopetz, 1991; Albert and Bosch GmbH, 2004; Scarlett and Brennan, 2006; Scheler and Schroder-Preikschat, 2006).

Co-operative schedulers have a number of desirable features, particularly for use in safety-related systems(Buttazzo, 1997; Kopetz, 1997; Pont, 2001) . Compared to a pre-emptive scheduler, co-operative schedulers can be identified as being simpler, having lower overheads, being easier to test and having greater support from certification authorities (Bate, 1998; Liu, 2000). Resource sharing in co-operative schedulers is also a straightforward process, requiring no special design considerations as is the case with pre-emptive systems (Cottet, Delacroix et al., 2002). The simplicity may suggest better predictability while simultaneously necessitating a careful design to realise the theoretical predictions in practice.

One of the simplest implementations of a co-operative scheduler is a cyclic executive (Baker and Shaw, 1988; Locke, 1992; Burns and Wellings, 1994): this is one form of a broad class of time triggered, co-operative (TTC) architectures. With appropriate implementations, TTC architectures are a good match for a wide range of applications, such as automotive applications (Ayavoo, Pont et al., 2005; Ayavoo, 2006), wireless (ECG) monitoring systems (Phatrapornnant and Pont, 2006) , various control applications (Schlindwein, Smith et al., 1988), data acquisition systems, washing-machine control and monitoring of liquid flow rates (Pont 2002).

Figure 1 illustrates the operation of TTC scheduler running two tasks A and B with 1 ms tick interval.

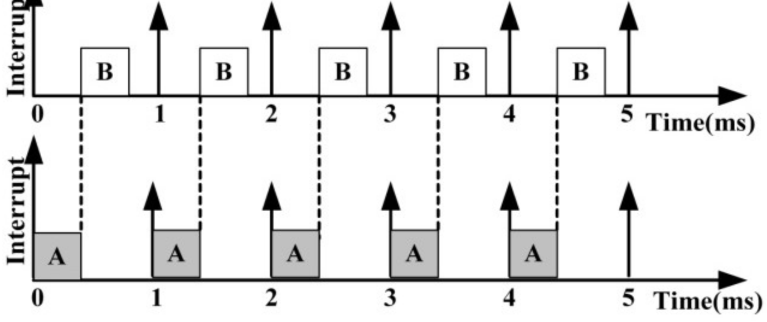


Figure 1: Illustrating the operation of a typical TTC scheduler

In applications where a TTC architecture is not found to be appropriate, a TT hybrid (TTH) design (Pont, 2001) may be an effective alternative (see Figure 2). A TTH design consists of a set of co-operative 'C' tasks (all of the same priority) and a single short pre-empting 'P' task (of higher priority than the C tasks). In many systems the short P task can be used for periodic data acquisition, through an analogue-to-digital converter or similar device (Buttazzo C, 2003).

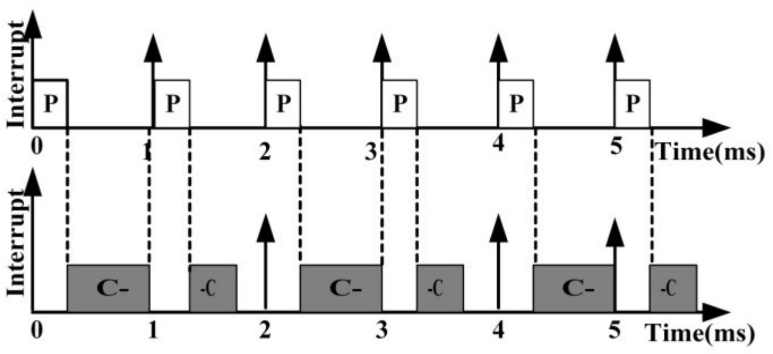


Figure 2: Illustrating the operation of a typical TTH scheduler

4. A small pattern collection

Our goal in this report is to present a small pattern collection which is intended to assist developers in the migration of systems from ET to equivalent systems with a TTC or TTH architecture. In doing this, our expectation is that – provided such a migration is appropriate, and carried out correctly – we should be able to improve system reliability.

The pattern association map is shown in Figure 3. The hierarchy of patterns is divided into abstract patterns and patterns (Kurian and Pont, 2005).

The remaining part of this report will present most of these patterns.

Please note that the patterns CO-OPERATIVE SCHEDULER, HYBRID SCHEDULER, LOOP TIMEOUT and WATCHDOG Timer are part of pattern collection described in (Pont, 2001) and are not included here. However, they can be downloaded (in the form of a complete book from this link http://www.tte-systems.com/books/pttes

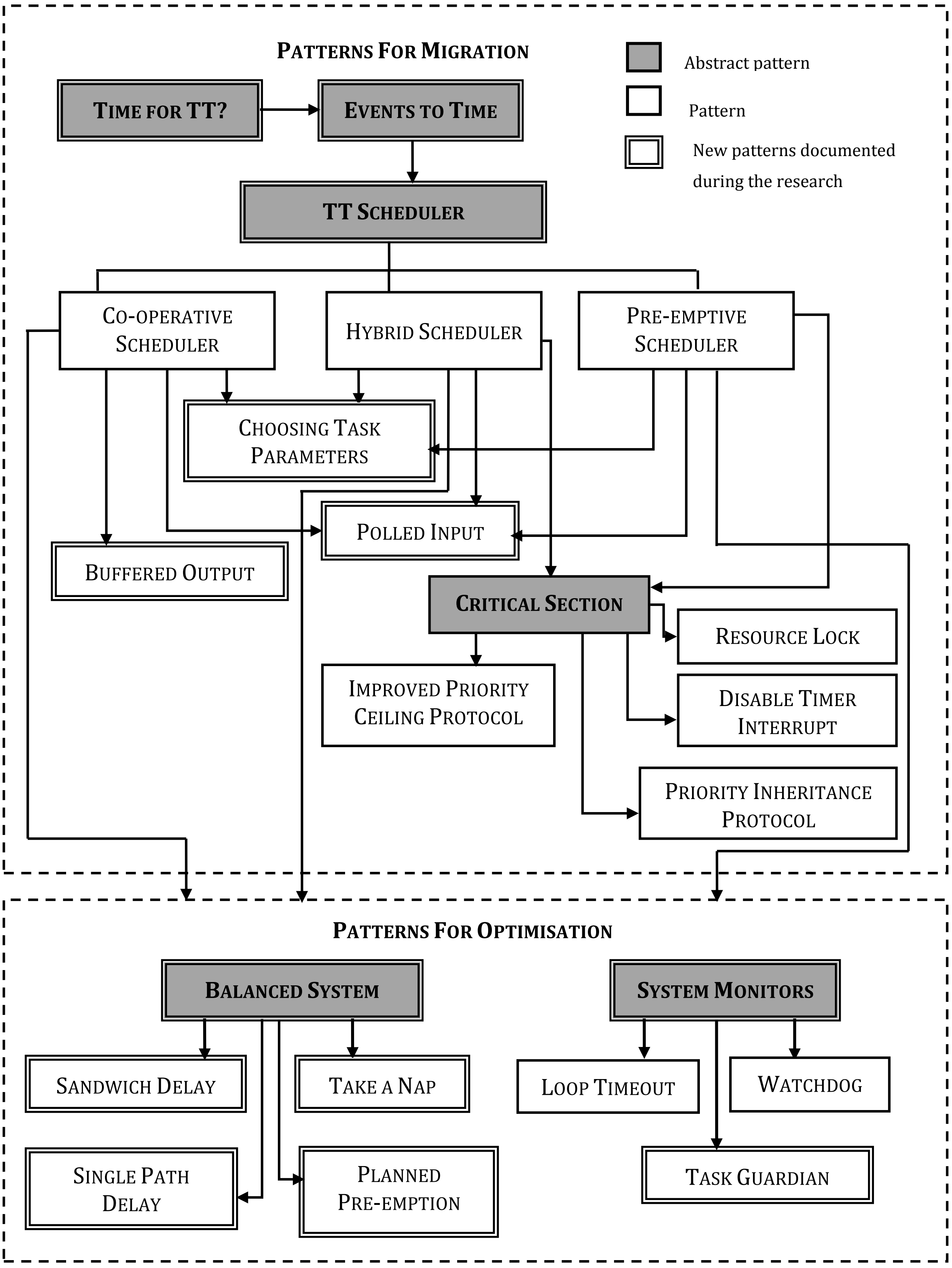


Figure 3: Pattern association map

TIME FOR TT?

Abstract pattern

Context

- You already have at least a design or prototype for your system based on some form of Event-triggered architecture.
- You are in the process of creating or upgrading an embedded system, based on a single processor.
- Reliable system operation is a key design requirement.

Problem

Should you use a TT architecture in your system?

Solution

Some systems are obvious candidates for TT architectures. These systems involve periodic data sampling or data playback, or other periodic activities.

Some simple examples:

- Data acquisition and sensing systems (for example, environmental systems for temperature monitoring) usually involve making data samples on a periodic basis. Some cases (high-frequency systems) may involve making millions of samples per second: other cases (e.g. temperature monitoring at a weather station) may involve making one sample per hour. Whatever the rate, a TT architecture will usually be used in order to guarantee high-quality ("jitter free") data sampling at a known signal-to-noise ratio.
- Control systems (for example, primary flight control in a aircraft or helicopter, cruise control in a passenger car, temperature control in an industrial furnace, control of a hard disk in a computer). Such systems all involve three core – periodic – activities: measuring some aspect of the system to be controlled (e.g. the furnace temperature), calculating changes required to the control system (e.g. calculating new gas burner settings) and applying the changes to the control system (e.g. updating the settings on the gas burner). Use of a TT architecture will ensure high-quality control without jitter in the input or output.
- Data output systems (for example, music players, video playback, head-up displays) are required to generate output signals at precise times (for example, "CD quality" sound will be played back at 44,400 samples per second. Any jitter in the playback will result in degradation of the music quality.

It is important to appreciate that – in many of these cases - use of a TT solution allows the system to perform the above periodic activities **and also perform other functions** (such as reading switches, updating displays, receiving data over serial communication links, performing calculations, etc) **without interfering in any way** with the processing outlined in the above examples. It is the ability to perform multiple tasks and still **guarantee that critical tasks will always execute as required** that makes a TT solution so attractive to developers of high-integrity, safety-related and safety-critical systems.

Clearly, not all systems fall into the “periodic sampling / playback” category. In particular, if your system must respond only to events which may occur at “random” times, it may not be a good match for this architecture.

For example, consider a simple radio transmitter which is used to open your garage doors a few times a week. We could use TT architecture to poll the switch on this system every 20 ms (just in case the switch has been pressed). However, while such a solution would undoubtedly work, it would be likely to have a shorter battery life than a simple event-triggered design which operates in power down mode except when the switch on the unit was pressed. **As there are not likely to be safety concerns with this system and the number of tasks is probably very small, an ET solution will probably be more appropriate in this situation.**

In between these two extremes there are many systems which involve both periodic tasks and the handling of “random” events. Such designs are typically characterised by the use of multiple interrupt service routines (ISRs). In these situations it may not be practical (or necessary) to create a “pure TT” (single interrupt) solution. However, it may well be practical to create a “more TT” solution which reduces the number of interrupts to a level at which it becomes possible to predict the system behaviour sufficiently accurately to meet the needs of the application.

Related patterns

To describe what architectural changes will be required in moving to a TT design, patterns EVENTS TO TIME and TT SCHEDULER provide further details.

Overall strengths and weaknesses

☺ Use of a TT architecture tends to result in a system with highly predictable patterns of behaviour.

☹ Inappropriate system design using this approach can result in applications which have a comparatively slow response to external events and / or shorter battery life.

EVENTS TO TIME

Abstract pattern

Context

- You and / or your development team have programming or design experience with "event-triggered and / or pre-emptive" (ET/P) system architectures: that is, architectures which may involve use of conventional real-time operating system (RTOS) and / or multiple interrupt-service routines (linked to different interrupt sources) and / or task pre-emption.
- You are in the process of creating or upgrading an embedded system, based on a single processor.
- You already have at least a design or prototype for your system based on some form of ET/P architecture.
- Because predictable and highly-reliable system operation is a key design requirement, you have opted to employ a "time-triggered system architecture in your system, if this proves practical.

Problem

How can you convert event triggered / pre-emptive designs and code (and mindsets) to allow effective use of a TT SCHEDULER as the basis of your embedded system?

Background

If we were forced to sum up the difference between "embedded" and "desktop" systems in a single word we'd say "interrupts".

Event triggered behaviour in systems is usually achieved through the use of such interrupts. The system is designed to handle interrupts associated with a range of sources (e.g. switch inputs, CAN interface, RS-232, analogue inputs, etc). Each interrupt source will have an associated priority. Each interrupt source will also require the creation of a corresponding "interrupt service routine" (ISR): this can be viewed as a short task which is triggered "immediately" when the corresponding interrupt is generated.

Creating such (ET/P) systems is – on the surface at least – straightforward. However, challenges often begin to arise (in non-trivial designs) at the testing stage. It is generally impossible to determine what state the system will be in when any interrupt occurs, which makes comprehensive testing almost impossible.

A time-triggered system also requires an understanding of interrupts, but the operation is fundamentally different. At the heart of a TT system is a scheduler which determines when the tasks in the system will be executed. In such a system, there is only a single interrupt source (usually a periodic timer "tick"): is used to drive the scheduler.

Solution

Here's what you need to do to migrate to a TT design:

- You need to ensure that only a single – periodic – timer interrupt is enabled (all other interrupt sources will be converted to flags, which will be polled as required).
- You have to determine an appropriate "tick interval" for your system (that is, you need to determine how frequently the timer interrupt need to take place).
- You have to convert any ET (event-triggered) ISRs into periodic tasks and add these to the schedule.
- You need to decide which TT architecture will best suite your application requirements. Pattern TT SCHEDULER provides comprehensive details. To summarise:
    - Choose Co-operative architecture if your system requirements could be met without any pre-emption involved. All the tasks in the system will be co-operative. For details see pattern CO-OPERATIVE SCHEDULER
    - Choose HYBRID SCHEDULER if limited pre-emption (only a single pre-emptive task) can fulfil the requirements of the system. All the other tasks in the system will be co-operative. Details about implementing such architecture are documented in pattern
    - Choose PRE-EMPTIVE Scheduler for full pre-emption in the system

Related patterns and alternative solutions

The PTTES collection

The PTTES collection (Pont, 2001) describes, in detail, a range of techniques which can be used to implement embedded systems with TTC architecture. This book can now be downloaded (free of charge) from the following WWW site:
http://www.tte-systems.com/books/pttes

TT Schedulers

The pattern TT SCHEDULER provides relevant background information and the situations in which it may be appropriate to use a TT scheduler in your application.

Reliability and safety implications

When compared to pre-emptive schedulers, co-operative schedulers have a number of desirable features, particularly for use in safety-related systems (Allworth, 1981; Ward, 1991; Nissanke, 1997; Bate, 2000)
.

For example, Nissanke (1997, p. 237) notes: "[Pre-emptive] schedules carry greater runtime overheads because of the need for context switching—storage and retrieval of partially computed results. [Co-operative] algorithms do not incur such overheads. Other advantages of [co-operative] algorithms include their better understandability, greater predictability, ease of testing and their inherent capability for guaranteeing exclusive access to any shared resource or data".

Allworth (1981, pp. 53–54) also notes: "Significant advantages are obtained when using this [co-operative] technique. Since the processes are not interruptable, poor synchronisation does not give rise to the problem of shared data. Shared subroutines can be implemented without producing re-entrant code or implementing lock and unlock mechanisms”.

Although not the main focus of this pattern, the advantages of a TT approach also apply in distributed systems: see, for example, (Scarlett and Brennan, 2006).

Overall strengths and weaknesses

☺ Use of TT architecture tends to result in a system with highly predictable patterns of behaviour.

☹ Inappropriate system design using this approach can result in applications which have a comparatively slow response to external events.

Examples

To illustrate part of the translation process, consider a simple ET system (Listing 1) running two interrupts, as a result two tasks X and Y will execute. These tasks are invoked by separate interrupts and implemented by associated ISRs.

```
void X_ISR(void) interrupt IEIndex1
   {
   }

void Y_ISR(void) interrupt IEIndex2
   {
   }

void main(void)
   {
   X_init();
   Y_init();
   EA = 1 ; // Enable all interrupts

   while(1)
      {
      PCON |= 0x01;
      }
   }
```

Listing 1: Possible ET design

There are various possibilities to convert ET designs to TT designs with any of the possible TT architectures listed in the solution. One possible design using TTC scheduler is illustrated in Listing 2.

```
void main(void)
   {
   SCH_Init(); // Set up the scheduler and tasks
   X_Init();
   Y_Init();

   // Add tasks to scheduler
   SCH_Add_Task(X_Update(), 0, 100);
   SCH_Add_Task(Y_Update(), 20, 200);

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }
   }
```

Listing 2: Possible TT design

P lease note that X and Y are two separate tasks created in separate .c files with their init and update functions.

TT SCHEDULER

Abstract pattern

Context

- You already have at least a design or prototype for your system based on some form of ET/P architecture.
- You and / or your development team are using pattern EVENTS TO TIME
- You are in the process of creating or upgrading an embedded system, based on a single processor.
- Because predictable and highly-reliable system operation is a key design requirement, you have opted to employ a "time triggered" system architecture in your system.

Problem

How will you decide which form of time-triggered scheduler should you use for your application?

Background

TT schedulers that we can use can take two forms: Co-operative and Pre-emptive. Both of these types of schedulers provide various options , some of these options are given below:

1. Co-operative Schedulers
    a. Super loop
    b. TTC Dispatch

2. Pre-emptive Schedulers
    a. Full Pre-emption
        i. TTRM Scheduler
    b. Limited Pre-emption
        i. TTH Scheduler

In co-operative scheduling, tasks co-operate with each other and wait for their turn to execute until the currently running task finishes execution.

In pre-emptive scheduling a task of higher priority which is ready to execute can pre-empt a currently running task of lower priority.

Overview of TT Schedulers

TTC-SL Scheduler

The simplest way of implementing a TTC scheduler is by means of a "Super Loop" or "Endless loop" (e.g. Pont, 2001; Kurian and Pont, 2007). A possible implementation of such a scheduler is illustrated in Listing 3.

```
int main(void)
   {
...
   while(1)
      {
      TaskA();
      Delay_6ms();
      TaskB();
      Delay_6ms();
      TaskC();
      Delay_6ms();
   }
   // Should never reach here
   return 1;
}
```

Listing 3: Illustrating a TTC Super Loop Scheduler

Applications based on a TTC-SL SCHEDULER have extremely small resource requirements. Systems based on such a pattern (if used appropriately) can be both reliable and safe, because the overall architecture is extremely simple and easy to understand, and no aspect of the underlying hardware is hidden from the original developer, or from the person who subsequently has to maintain the system.

TTC Dispatch Scheduler

The TTC scheduler implementation referred to here as a "TTC-Dispatch" scheduler provides a more flexible alternative see Listing 4.

The type of TTC scheduler implementation discussed in this pattern is usually implemented using a hardware timer, which is set to generate interrupts on a periodic basis (with "tick intervals" of around 1 ms being typical). In most cases, the tasks will be executed from a "dispatcher" (function), invoked after every scheduler tick. The dispatcher examines each task in its list and executes (in priority order) any tasks which are due to run in this tick interval (see Figure 4). The scheduler then places the processor into an "idle" (power saving) mode, where it will remain until the next tick.

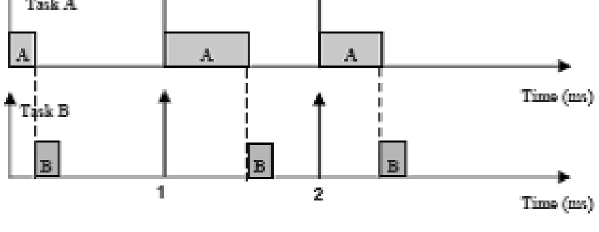


Figure 4: Illustrating TTC design

Provided that an appropriate implementation is used, a time-triggered, co-operative (TTC) architecture is a good match for a wide range of low-cost, resource-constrained applications. TTC architectures also demonstrate very low levels of task jitter (Locke, 1992) and can maintain their low-jitter characteristics even when techniques such as dynamic voltage scaling (DVS) are employed to reduce system power consumption .

```
void main(void)
   {
   // Set up the scheduler
   SCH_Init_T2();

   // Init tasks
   TaskA_Init();
   TaskB_Init();

   // Add tasks (10 ms ticks)
   // Parameters are filename, offset (ticks), period (ticks)
   SCH_Add_Task(TaskA, 0, 3);
   SCH_Add_Task(TaskB, 1, 3);
   SCH_Add_Task(TaskC, 2, 3);

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      SCH_Go_To_Sleep();
      }
   }
```

Listing 4: TTC Implementation

TTRM architectures

Where a TTC architecture is not found to be suitable for use in a particular resource constrained embedded systems, fixed-priority scheduling has been proposed as the most attractive alternative (Audsley, Burns et al., 1991; Bate, 1998).

"Time-triggered rate monotonic" (TTRM) is a well-known fixed-priority scheduling algorithm that was introduced by (Liu and Layland, 1973) in 1973. Technically, TTRM is a pre-emptive scheduling algorithm which is based on a fixed priority assignment (Kopetz, 1997) . In particular, the priorities are assigned to periodic tasks accord to their occurrence rate or, in other words, priorities are inversely proportional to their period, and they do not change through out of the operation (because their periods are constant).

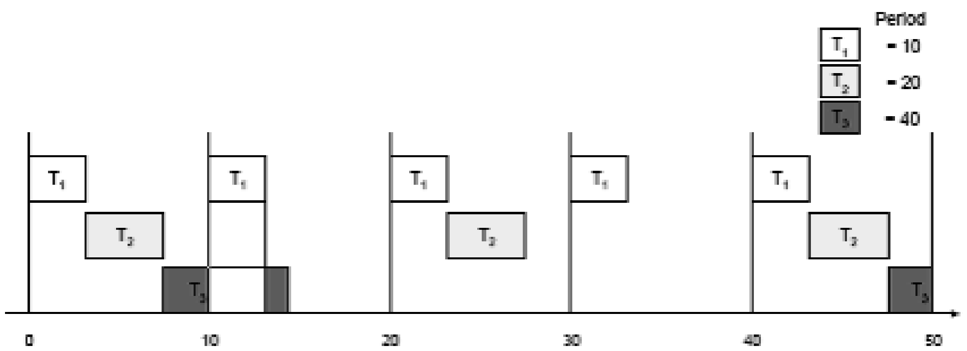


Figure 5: Illustrating TTRM architecture

To illustrate the use of TTRM scheduling, Figure 5 above shows how a set of periodic tasks can be scheduled by this algorithm. Task T1 is executed periodically at the fastest rate, every 10 ms, and is determined to be the highest priority in this scheduling policy, while task T2 and T3, which are run every 20 and 40 ms respectively, have lower priority levels according to their rates. A task scheduled by the TTRM algorithm can be pre-empted by a higher priority task. As illustrated in Figure5, task T3 - which is running - is pre-empted by task T1 is at time 10: it carries on after the completion of task T1.

TTH architectures

Where a TTC architecture is not found to be suitable for a particular system, use of a TTRM design may not be necessary. For example, a single, time-triggered, pre-empting task can be added to a TTC architecture, to give what we have called a "time-triggered hybrid" (TTH) scheduler (Pont, 2001; Maaita and Pont 2005) and others have called a "multi-rate executive with interrupts" (Kalinsky, 2001)

Use of a TTH SCHEDULER allows the system designer to create a static schedule made up of (i) a collection of tasks which operate co-operatively and (ii) a single – short - pre-empting task (see Figure 6). In many of the systems employing a TTH architecture, the pre-empting task will be used for periodic data acquisition, typically through an analogue-to-digital converter or similar device.

Such requirements are common in, for example, control systems (Buttazzo, 2005) and applications which involve data sampling and Fast-Fourier transforms (FFTs) or similar techniques:

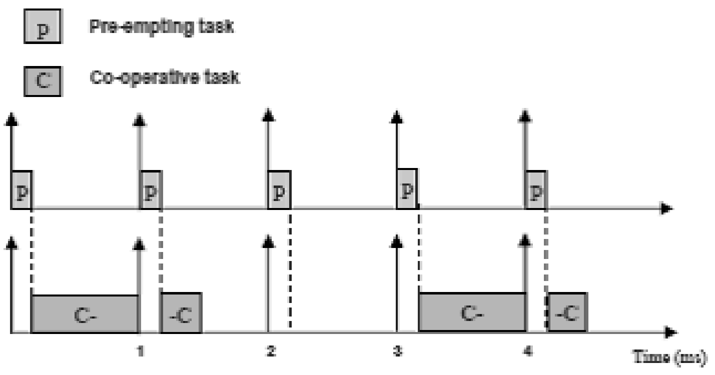


Figure 6: Illustrating TTH design

Solution

Here are the guidelines about choosing appropriate TT architecture.

When to use TTC

Use TTC architecture where ever possible as first choice because of its simple and efficient design. Of course, this architecture is not always appropriate. The main problem is that long tasks will have an impact on the responsiveness of the system. This concern is succinctly summarised by Allworth: "[The] main drawback with this [co-operative] approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response [of the] system is not to be impaired." (Allworth, 1981).

We can express this concern slightly more formally by noting that if the system must execute one of more tasks of duration X and also respond within an interval T to external events (where $T < X$), a pure co-operative scheduler will not generally be suitable. In more simple words duration of a task (execution time) must be less than the tick interval of the system.

In practice, it is sometimes assumed that a TTC architecture is inappropriate because some simple design options have been overlooked, see pattern BUFFERED OUTPUT

When to use TTH

For systems where TTC is not an appropriate choice, avoid jumping to fully pre-emptive architectures as they incur higher overheads because of context switching involved during task pre-emption. Check for TTH solution which provides a limited level of pre-emption. For example, consider a wireless electrocardiogram (ECG) system. An ECG is an electrical recording of the heart that is used for investigating heart disease. In a hospital environment, ECGs normally have 12 leads (standard leads, augmented limb leads and precordial leads) and can plot 250 sample-points per second (at minimum). In the portable ECG system considered here, three standard leads (Lead I, Lead II, and Lead III) were recorded at 500 Hz. The electrical signal were sampled using a (12-bit) ADC and – after compression – the data were passed to a "Bluetooth" module for transmission to a notebook PC, for analysis by a clinician see (Phatrapornnant and Pont, 2006).

In one version of this system, we are required to perform the following tasks:

- Sample the data continuously at a rate of 500 Hz. Sampling takes less than 0.1 ms.
- When we have 10 samples (that is, every 20 ms), compress and transmit the data, a process which takes a total of 6.7 ms.

In this case, we will assume that the compression task cannot be neatly decomposed into a sequence of shorter tasks, and we therefore cannot employ a pure TTC architecture. However, even if you cannot – cleanly - solve the long task / short response time problem, then you can maintain the core co-operative scheduler, and add only the limited degree of pre-emption that is required to meet the needs of your application.

For example, in the case of our ECG system, we can use time-triggered hybrid architecture.

In this case, we allow a single pre-empting task to operate: in our ECG system, this task will be used for data acquisition. This is a time-triggered task, and such tasks will generally be implemented as a function call from the timer ISR which is used to drive the core TTC scheduler. As we have discussed in detail elsewhere (Pont, 2001: Chapter 17) this architecture is extremely easy to implement, and can operate with very high reliability. As such it is one of a number of architectures, based on a TTC scheduler, which are cooperatively based, but also provide a controlled degree of pre-emption.

When to use TTRM

If both TTC and TTH architectures are not appropriate for your application and full pre-emption is a necessary, then TTRM architecture may match your requirements.

Overall, it has been claimed that the main advantage of TTRM scheduling is flexibility during design or maintenance phases, and that such flexibility can reduce the total life cost of the system (Locke, 1992; Bate, 1998). The schedulability of the system can be determined based on the total CPU utilisation of the task set: as a result - when new functionalities are added to the system – it is only necessary to recalculate the new utilisation values. In addition, unlike a TTC design, there is no need to break up long individual tasks in order to meet the length limitations of the minor cycle. The need to employ harmonic frequency relationships among periodic tasks is also avoided. Finally, the scheduling behaviour can be predicted and analysed using a task model proposed by Liu and Layland (1973).

However, the scheduling overheads of TTRM schedulers tend to be larger than those of TTC schedulers because of the additional complexity associated with the context switches when saving and restoring task state (Locke, 1992). This is a concern in embedded systems with limited resources.

Locking mechanisms

If you use any architecture which involves pre-emption (TTH or TTRM), you need to consider ways of preventing more than one task from accessing critical resources at the same time. See pattern CRITICAL SECTION and related patterns for more details.

Overall strengths and weaknesses

☺ Use of a TT scheduler tends to result in a system with highly predictable patterns of behaviour.

☹ Inappropriate system design using this approach can result in applications which have a comparatively slow response to external events.

CHOOSING TASK PARAMETERS

Pattern

Context

- You are in the process of creating or upgrading an embedded system, based on a single processor.
- Because predictable and highly-reliable system operation is a key design requirement, you have opted to employ a “time triggered” system architecture in your system, if this proves practical.

Problem

How can you choose your tasks parameters such as offset and task order to allow effective use of a TT Scheduler as the basis of your embedded system?

Background

Whether a TTC or TTH implementation is used, a number of key scheduler/task parameters must be determined (including the tick interval, task order, and initial delay or phase of each task). Inappropriate choices may mean that a given task set cannot be scheduled at all or inappropriate decisions may still lead to unnecessarily high levels of task jitter. The following parameters are used to characterise each task (Liu and Layland, 1973; Tindell, Burns et al., 1994; Buttazzo, 1997)

1. **Period (p_i):** is the time interval after which task T_i should be repeated, in other words it is the length of time between every two invocations.
2. **Offset (o_i):** is the time, measured from the start of the system power on, after which the first period of task T_i starts.
3. **Start time (s_i):** This is the time at which task starts its execution.
4. **Release time (r_i):** is the time, measured from the start of the task period, after which task T_i becomes ready to run.
5. **Finish time (f_i):** This is the time at which task completes its execution.
6. **Deadline (d_i):** is the time before which task T_i should be completed. Deadline can be measured from the start of the system power on, in which case it is called absolute deadline. Alternatively it can be measured from the start of the task period, in which case it is called relative deadline
7. **Worst-case execution time ($WCET_i$):** This is the longest time taken by the processor to complete the execution of a task T_i.
8. **Best-case execution time ($BCET_i$):** This is the shortest time taken by the processor to finish task T_i.

These parameters are shown in Figure 7.

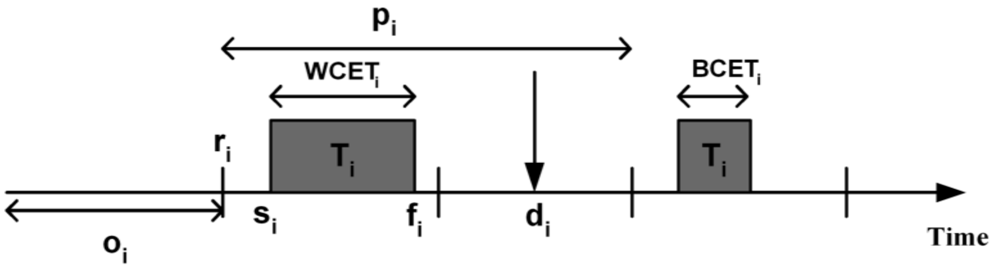


Figure 7: Illustrating task parameters

Another important parameter is the order of tasks in which they are added to the schedule. Inappropriate choice of these parameters may lead to high values of jitter, increased power consumption or a task set which cannot be scheduled at all.

Solution

Choosing the correct offset

A task offset specifies when a task should start or more precisely it specifies the first tick at which the first instance of a task is ready to run. For a task set given with WCET, period and deadline start scheduling all the tasks with offset 0 if the sum of execution times of all the tasks is less than or equal to the length of tick interval.

While assigning task offsets you can take care of the following situations.

For example, the tasks in Table 1 can be scheduled if a tick interval of 3 ms is used and the tasks will meet their deadlines as well.

Table 1: Task specifications for a system in which task offsets are appropriate (all the tasks will meet their deadlines)

Task	WCET(ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	0.5	3	3	0
B	0.75	3	6	0
C	1.5	3	6	0

On the contrary, task set in Table 2 can be scheduled with a tick interval of 5ms but Task C will missed its deadline as shown in Figure8.

Table 2: Task specifications for a system in which task offsets are in appropriate (Task C will not be able to meet its deadline)

Task	WCET(ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	1	5	5	0
B	1.5	5	10	0
C	3	5	10	0

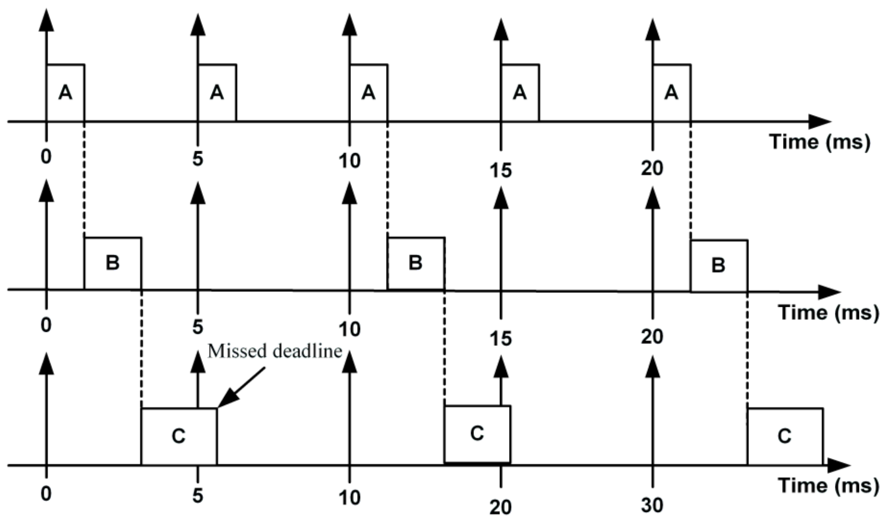


Figure 8: Task C missed deadline because of incorrect offset

By changing the offset to 1, Task C will meet its deadline as shown in Figure 9.

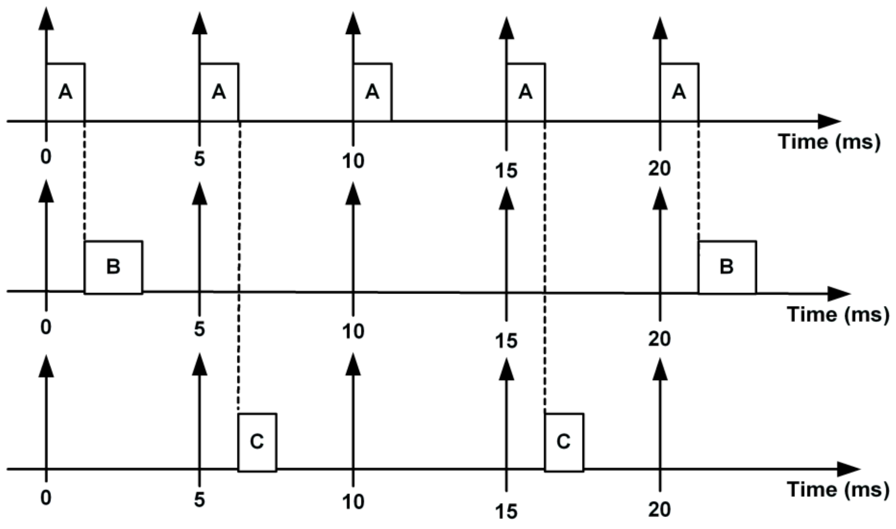


Figure 9: Task C will meet deadline after changing offset to 1

Inappropriate assignment of task offsets can also increase the jitter value in task. For example consider the task set given in Table 3. With the given set of parameters Task C will run after Task A and Task B in some ticks and just after Task A in some other ticks (see Figure 10). This kind of situation poses a kind of unpredictability in system behaviour. This can be adjusted by keeping the jitter constant in Task C in all ticks. Changing the offset of Task C can help to keep the jitter value constant in all the ticks see Table 4 and Figure 11.

Table 3: Task offset that can cause varied jitter in Task C

Task	WCET(ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	1	5	5	0
B	1.5	20	20	0
C	1	10	10	0

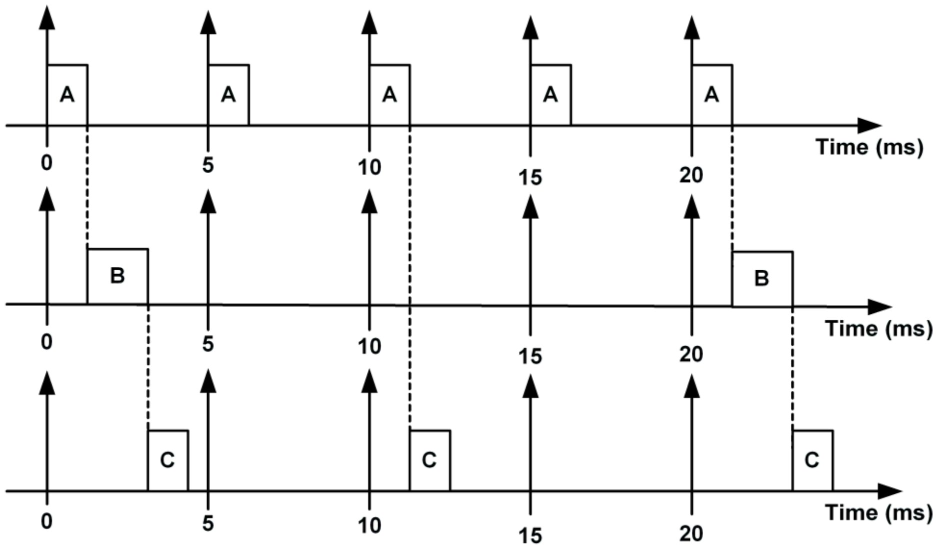


Figure 10: Illustrating Tasks shown in Table 4 above:

Table 4: Changing the offset of Task C to 1 can make the jitter constant for all task instances

Task	WCET(ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	1	5	5	0
B	1.5	20	20	0
C	1	10	10	1

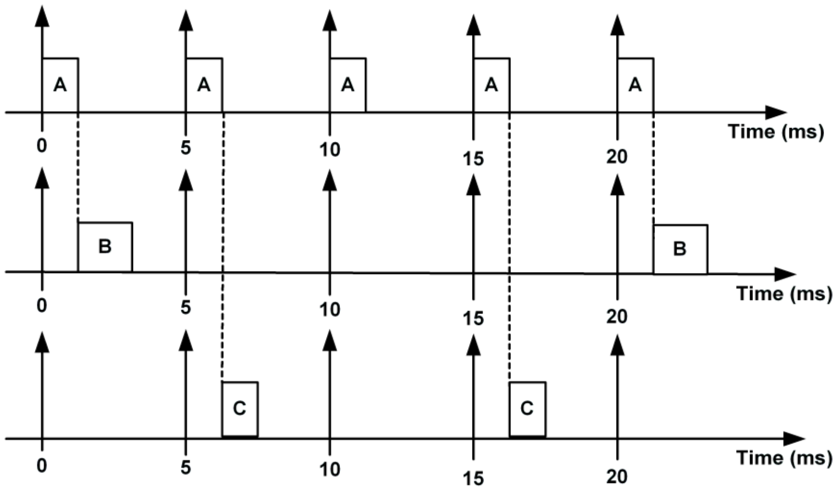


Figure 11: Changing of task offset can keep the jitter constant

Choosing correct task order

Task order can also affect jitter. It is important to consider the task order for jitter sensitive tasks. For example consider the schedule given in Table 5 and Figure 12.

Table 5: Example of incorrect order of task set which can cause varied jitter in task C

Task	WCET(ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	2	10	10	0
B	3	20	20	0
C	4	30	30	0

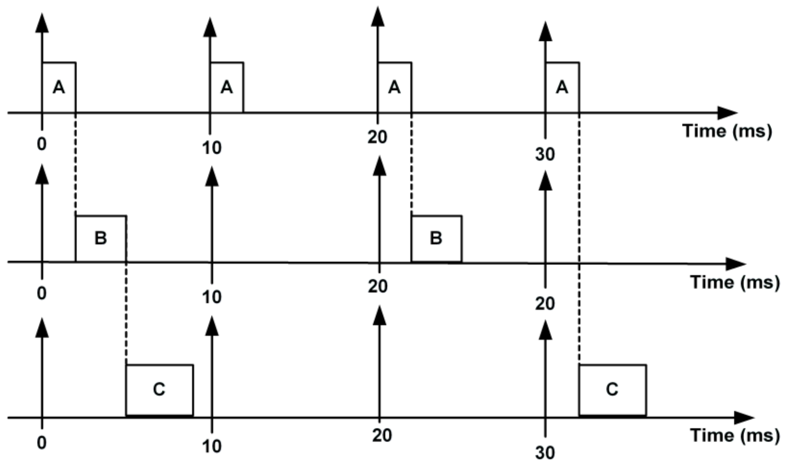


Figure 12: Task C showing variations in jitter because of incorrect task order

Changing the task order can keep the jitter constant in Task C. The new order of tasks is shown in Table 6 and Figure 13.

Table 6: Rearrangement of the task order to keep the jitter constant in task C

Task	WCET(ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	2	10	10	0
C	4	30	30	0
B	3	20	20	0

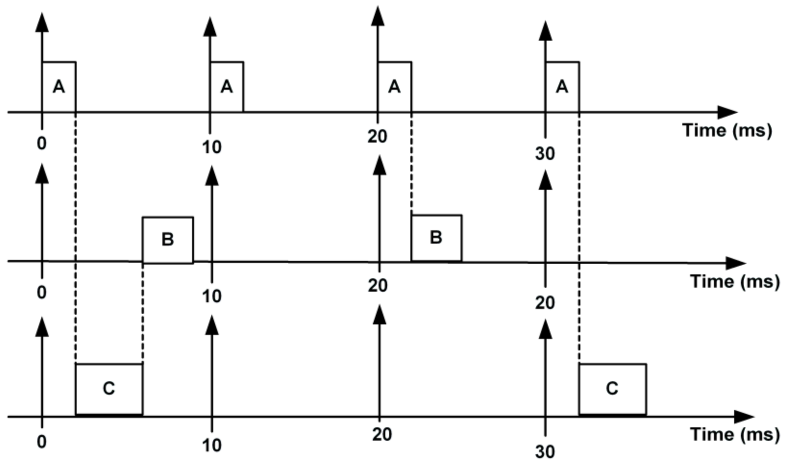


Figure 13: Rearrangement of task order will make task C run after task A every time

Reliability and safety implications

Inappropriate selection of task parameters can make the system unreliable

Overall strengths and weaknesses

- ☺ Uses of appropriate task parameters will result in stable system with all the tasks meet their deadlines and reduced/constant values of jitter.
- ☹ Inappropriate system design using this approach can result in applications which have a comparatively slow response to external events.

Context

- You are applying the pattern EVENTS TO TIME (TTC)
- You need to deal – cleanly – with a "long task" (that is, a task which may have an execution time greater than your chosen tick interval.
- You need to send a significant amount of data between your processor / system and an external device: the data transfer process will take some time.

Problem

How can you structure the data-transfer tasks in your application in a manner which is compatible with TTC architecture?

Background

We illustrate the need for the present pattern with an example.

Suppose we wish to transfer data to a PC at a standard 9600 baud. Transmitting each byte of data, plus stop and start bits, involves the transmission of 10 bits of information (assuming a single stop bit is used). As a result, each byte takes approximately 1 ms to transmit.

Now, suppose we wish to send this information to the PC:

```
Current core temperature is 36.678 degrees
```

If we use a standard function (such as some form of `printf()`) the task sending these 42 characters will take more than 40 milliseconds to complete. In a system supporting task pre-emption, we may be able to treat this as a low-priority task and let it run as required. This approach is not without difficulties (for example, if a high-priority task requires access to the same communication interface while the low-priority task is running). However, with appropriate system design we will be able to make this operate correctly under most circumstances.

Now consider the equivalent TTC design. We can't support task pre-emption and a long data-transmission task (around 40 ms) is likely to cause significant problems. More specifically, if this time is greater than the system tick interval (often 1 ms, rarely greater than 10 ms) then this is likely to present a problem as shown in Figure 14. The RS-232 task is a "long task" has duration greater than the system tick and so is missing the next tick intervals.

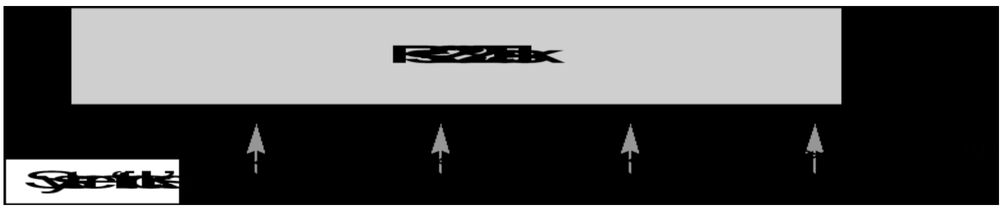


Figure 14: A schematic representation of the problems caused by sending a long character string on an embedded system. In this case, sending the massage takes 42 ms while the System tick interval is 10 ms.

Perhaps the most obvious way of addressing this issue is to increase the baud rate; however, this is not always possible, and - even with very high baud rates - long messages or irregular bursts of data can still cause difficulties.

More generally, the underlying problem here is that the data transfer operation has a duration which depends on the length of the string which we wish to submit. As such, the worst-case execution time (WCET) of the data transfer task is highly variable (and, in a general case, may vary depending on conditions at run time). In a TTC design, we need to know all WCET data for all tasks at design time. We require a different system design. As (Gergeleit and Nett, 2002) have noted " Nearly all known real-time scheduling approaches rely on the knowledge of WCETs for all tasks of the system." The known WCET of tasks will be helpful for developers in designing the offline schedule and preventing task overrun.

Solution

Convert a long data-transfer task (which is called infrequently and may have a variable duration) into a periodic task (which is called comparatively frequently and which has a very short – and known – duration).

A BUFFERED OUTPUT consists of three key components:

- A buffer (usually just an array, implemented in software)
- A function (or small set of functions) which can be used by the tasks in your system to write data to the array.
- A periodic (scheduled) task which checks the buffer and sends a block of data to the receiving device (when there are data to send).

Figure 15 provides an overview of this system architecture. All data to be sent are first moved to a software buffer (a very fast operation). The data is then shifted – one block at a time – to the relevant hardware buffer in the microcontroller (e.g. 1 byte at a time for a UART, 8 bytes at a time for CAN, etc): this software-to-hardware transfer is carried out every 1ms (for example), using a (short) periodic task.

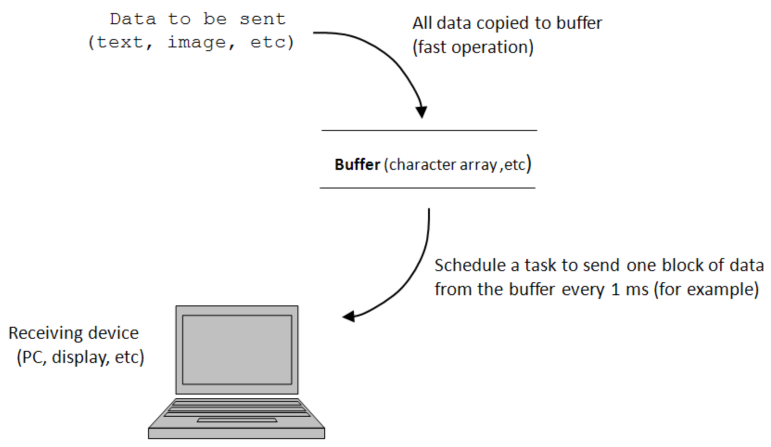

Data to be sent
(text, image, etc)
All data copied to buffer
(fast operation)
Buffer (character array,etc)
Schedule a task to send one block of data
from the buffer every 1 ms (for example)
Receiving device
(PC, display, etc)

Figure 15: An overview of the BUFFERED OUTPUT architecture.

Hardware resource implications

In most cases, the CPU requirements for BUFFERED OUTPUT are very limited, provided we take reasonable care at the design stage. For example, if we are sending message over a CAN bus and we know that each message takes approximately 0.15 ms to transmit; we should schedule the data transmission task to check the buffer at an interval > 0.15 ms. If we do this, the process of copying data from the software buffer to the (CAN) hardware will take very little time (usually a small fraction of a millisecond).

For very small designs (e.g. 8-bit systems) the memory requirements for the software buffer can prove significant. If you can't add external memory in these circumstances, you will need to use a small buffer and send data as frequently as possible (but see the comment above).

In some cases, hardware support can help to reduce both memory requirements and processor load. For example, if using UART-based data transmission, UARTs often have 16-byte hardware buffers: if you have these available, it makes sense to employ them.

Portability

This technique is generic and highly portable.

Reliability and Safety Issues

- Special care must be taken while defining buffer length, the data transfer should not cause any buffer overflow
- Applications that involve high amount of data transfer like video and DSP applications or data acquisition systems the use of buffer might not be a viable solution.

Overall strengths and weaknesses

☺ Use of buffered output is an easy solution for faster data transfer from a task running in an embedded application

☹ One has to be very careful while defining the buffer length, inappropriate buffer definitions may cause buffer overflow and data loss.

POLLED INPUT

Pattern

Context

- You are applying the pattern EVENTS TO TIME
- You need to make your system responsive to external inputs through interface like switches.

Problem

How do you build a time-triggered (TT) system which is equivalent of your event-triggered (ET) system such that it can respond to all (external/internal) input interfaces?

Background

Designing a TT system requires more planning efforts. In a time-triggered co-operative (TTC) design the possible occurrence and the execution times of all the tasks needs to be known in advance. The designer has to plan a task schedule which must execute all the tasks periodically at their allocated time intervals. This effort makes the system more predictable. In contrast to this, in an event-triggered system the scheduler executes the tasks dynamically as the events arrive thus no guarantee that they meet any timeliness constraints. This is the reason that ET designs are not recommended for safety critical applications. The event triggered behaviour in systems is achieved through the use of interrupts.   To support these interrupts, Interrupt Service Routines (ISRs) are provided. Whenever an interrupt occurs it stops the currently running task and ISR executes to respond to the interrupt. This “context switching” is an overhead that sometimes raised serious complications in systems.

The abstract pattern EVENTS TO TIME provides more relevant background information.

Solution

A POLLED INPUT should meet the following specification:

- It should include a periodic task which polls for the occurrence of the event.
- The period of the above task should be set to some value less than or equal to minimum inter-arrival time[1] of the event in question.
- The interrupt associated with this event should not be enabled. In fact only one interrupt associated with the timer responsible for generating system “ticks” should be enabled.

[1] In ET systems the exact arrival time of events is not known so we assume a minimum distance between the arrivals of two consecutive events.

Hardware resource implications

Different interfaces have different implications under various circumstances. Reading a switch input imposes minimal loads on CPU and memory resources whereas scanning the keypad interface imposes both a CPU and memory load.

Reliability and safety issues

One major concern here in migrating from event-triggered to time-triggered is to make systems more predictable. Characteristic for the time-triggered architecture is the treatment of (physical) real time as a first order quantity (Kopetz and Bauer 2002) this implies to the fact that time-triggered systems must be very carefully designed, the task activation rates must be fixed according to the system dynamics i.e. how frequent an input needs to be polled.

Portability

This technique is generic and highly portable.

Overall strengths and weaknesses

☺ A flexible technique, programmer can easily do changes in code for example if auto repeat is required

☺ It is simple and cheap to implement.

☹ Provides no protection against out of range inputs or electrostatic discharge (ESD)

☹ More processor utilisation in polling for tasks

CRITICAL SECTION

Abstract pattern

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can you avoid conflicts over shared resources during the execution of critical sections?

Background

We provide some relevant background material in this section.

Scheduling and TT architectures

For general background information about scheduling (and scheduling of time-triggered systems in particular), please refer to the pattern TT SCHEDULER. TT SCHEDULER provides background information on key concepts such as TTC, TTH and TTRM scheduling.

Shared resources and critical sections

Our focus in this pattern will be on TTH and TTRM (and related) designs in which task pre-emption can occur. Our particular concern will be with the issue of resources which may be accessed by more than one task at the same time. Such "shared resources" may – for example - include areas of memory (for example, two tasks need to access the same global variable) or hardware (for example, two tasks need to access the same analogue-to-digital converter). The code which accesses such shared resources is referred to as a "critical section".

Suppose that there are two tasks, $Task_A$ and $Task_B$ in a system, which are illustrated in Figure 16. There is one shared resource and N represents the normal section and C represents the critical section (that is, the section which involves access to a shared resource). From t1 to t4, $Task_A$ and $Task_B$ are attempting to run "simultaneously".

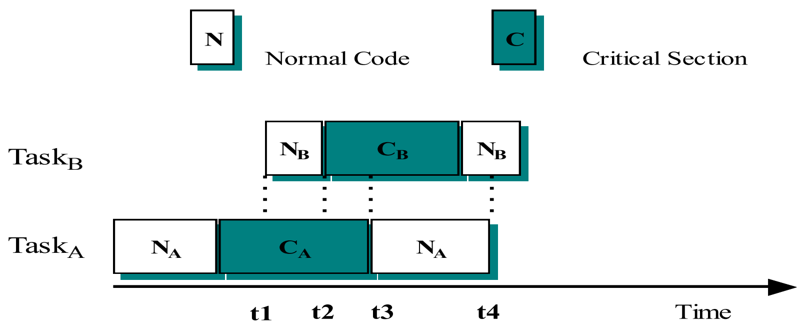


Figure 16: Two Tasks with a shared resource

In a TTC system, a task cannot be pre-empted by another task and the two tasks shown in Figure 16 are scheduled as shown in Figure 17. As in this example, there are no conflicts caused by the shared resources in TTC systems.

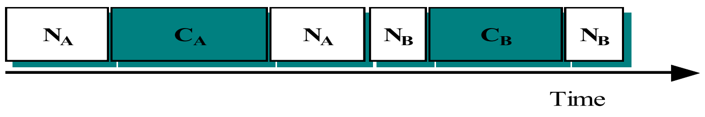


Figure 17: Two tasks scheduled using a TTC scheduler

However, if the same tasks are scheduled in a pre-emptive system, there may be conflicts. For example, suppose the priority of Task$_B$ is higher than that of Task$_A$. Task$_B$ will then pre-empt Task$_A$ at time t1 while it is running the critical section (Figure 18).

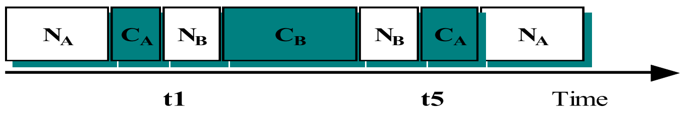


Figure 18: Two tasks scheduled in a pre-emptive system

Assume that the shared resource in the above example is some shared data (for example, numerical data stored in an array). The two tasks write and read the data in the critical section. Task$_B$ pre-empts Task$_A$ at t1 while it is reading the data: we will assume that Task$_A$ has read from half of the array at the time it is interrupted. We will further assume that Task$_B$ Task$_B$ updates all of the data values in the array. After Task$_B$ finishes, Task$_A$ then continues, reading the remaining values from the second half of the array: it then processes a combination of "new" and "old" data, possibly leading to erroneous results (e.g. see (Kalinsky, 2001)).

In general, tasks must share data and / or hardware resources. However, the system designer must ensure that each task has exclusive access to the shared resources to avoid conflicts, data corruption or "hanging tasks" (Labrosse, 2000; Pont, 2001; Laplante, 2004).

What is a resource lock?

A lock is the most common way to protect shared resources.

Before entering a critical section, a semaphore is checked. If it is clear, the resource is available. The task then sets the semaphore and uses the resource. When the task finishes with the resource, the semaphore is cleared.

Resource locking in this way requires care but is comparatively straightforward to implement. and affects only those tasks that need to take the same semaphore (Simon, 2001).

The main drawback is that it causes priority inversion in a priority based system (Sha, Rajkumar et al., 1990; Burns and Wellings, 1997; Renwick, 2004).

What is priority inversion?

In a priority-based system, each task is assigned a priority. In a TTC design, the scheduler will – when deciding which task to run next – always run the task with the highest priority (and this task will then run to completion). In a pre-emptive system, a high-priority task may interrupt a lower-priority task while it is executing.

Priority inversion can occur in pre-emptive designs when resource locks are used. For example, suppose that a low-priority task is using a resource. The resource will be locked. If a high-priority task is then scheduled to run (and use the resource) it will not be able to do so: in effect, the low-priority task will be given greater priority than the high-priority task.

For example, Figure 19 shows an intended operation sequence for two tasks, $Task_H$ and $Task_L$, sharing a critical section C. Figure 20 shows how the priority inversion takes place. When $Task_L$ owns C, and $Task_H$ attempts to access it (at t_3), $Task_H$ is blocked and has to wait until time t_4 before it can run.

Please note that this is sometimes called "bounded priority inversion" (Burns, 2001; Renwick, 2004) or "controlled priority inversion" (Locke, 2002). In this case, the blocking time of $Task_H$ will not exceed the duration of the critical section C of $Task_L$.

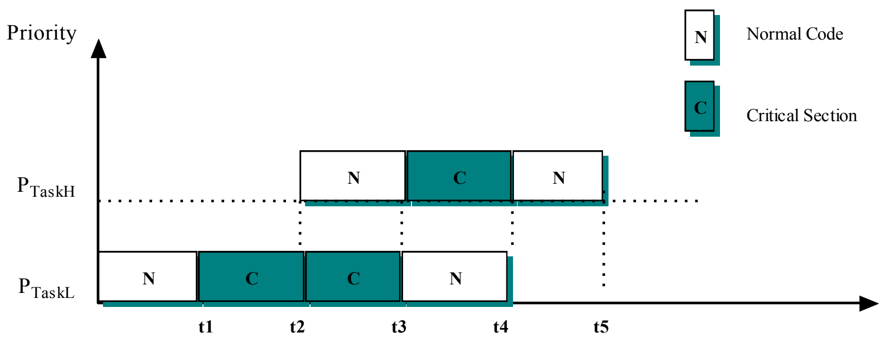


Figure 19: Operation Sequences of TaskH and TaskL

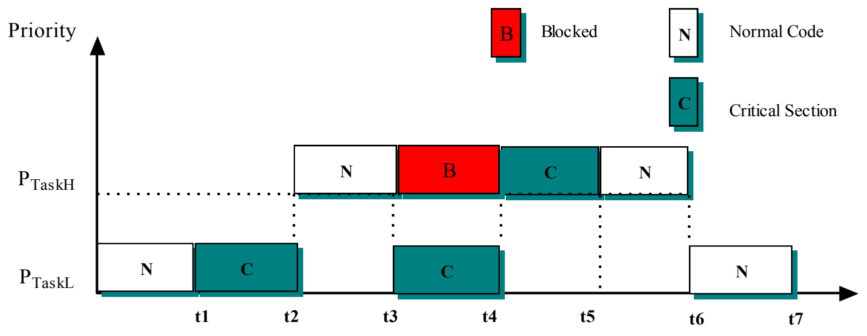


Figure 20: Bounded Priority Inversion

We further suppose that $Task_M$ (with "medium" priority) pre-empts $Task_L$ when $Task_H$ is blocked by $Task_L$, the owner of the shared resource at this time. $Task_H$ then has to wait until $Task_M$ relinquishes control of the processor and $Task_L$ completes the critical section. For example, see Figure 21: here, at t_4, $Task_M$ pre-empts $Task_L$.

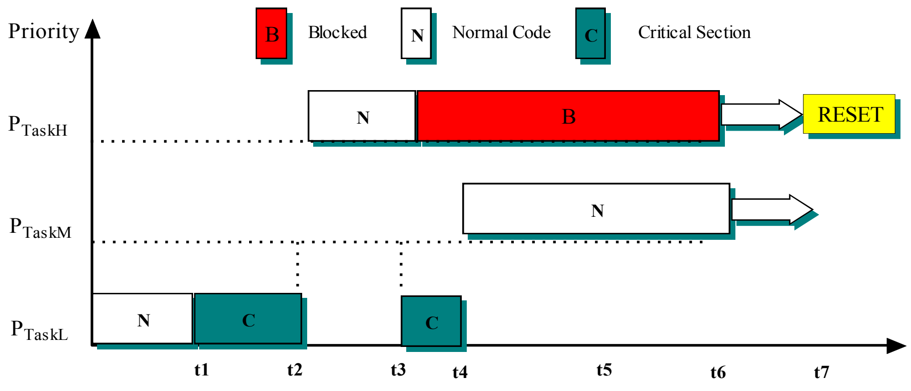


Figure 21: Unbounded Priority Inversion

In these circumstances, the worst-case waiting time for $Task_H$ is the sum of the worst-case execution times of $Task_M$ and the critical section of $Task_L$. This is called unbounded priority inversion (Renwick, 2004). If $Task_M$ runs for a long time (or "for ever"), $Task_H$ is likely to may miss its deadline, with potentially serious consequences (shown as a system reset in Figure 21).

Unbounded priority inversion can be particularly problematic. For example, in 1997, the Mars Pathfinder mission nearly failed because of an undetected (unbounded) priority inversion (Jones, 1997).

What is deadlock?

As noted above, a locking mechanism may lead to priority inversion. However, this is not the only problem which is introduced by the use of locking mechanisms.

For example, suppose that $Task_H$ is waiting for a resource held by $Task_L$, while $Task_L$ is simultaneously waiting for a resource held by $Task_H$: neither task is able to proceed and – as a result - a *deadlock* is formed.

As an example, Figure 22 shows two tasks $Task_H$ and $Task_L$ which share two resources (via C1 and C2): in this case, it is assumed that C1 is nested within C2 in $Task_L$ and that C2 is nested within C1 in $Task_H$

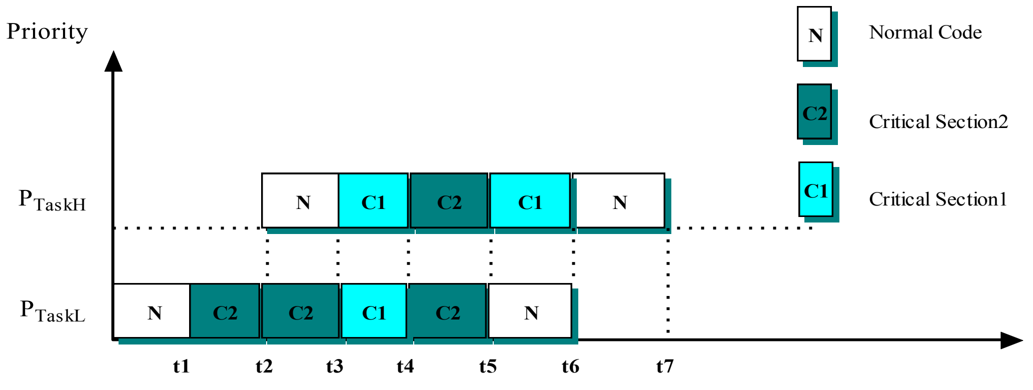


Figure 22 Operation Sequences of TaskH and TaskL

In Figure 23, at t_1, $Task_L$ locks C2, at t_2 $Task_H$ pre-empts $Task_L$ and starts to run, locks C1 at t_3, then requires C2 at t_4. Due to the fact that $Task_L$ has locked C2, $Task_H$ is blocked and $Task_L$ resumes running at t_4. At t_5, $Task_L$ requires C2 which is locked by $Task_H$, and both tasks are blocked.

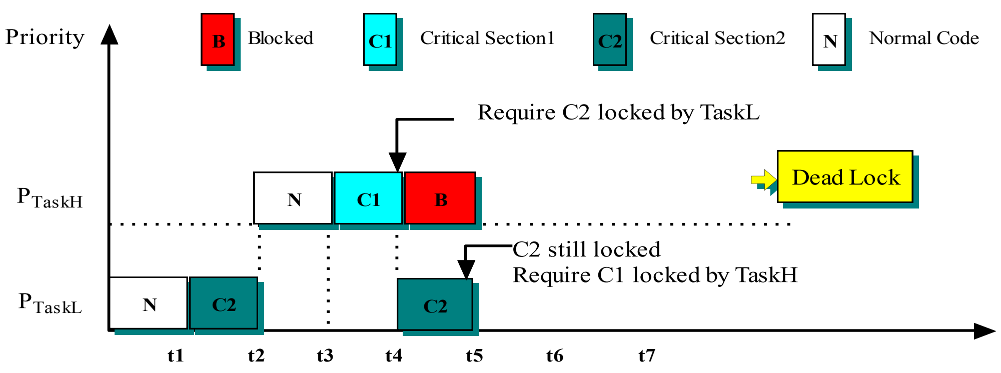


Figure 23: Deadlock, Operation Sequences of Tasks without Priority Protocols

What is chained blocking?

Locking mechanisms can also cause a phenomenon known as chained blocking.

This is best explained by means of an example. Suppose that Task$_H$ is waiting for a resource held by Task$_M$, while Task$_M$ is waiting for a resource held by Task$_L$, and so on (Figure 24). Task$_H$ needs to sequentially access resources C3 and C2. Task$_M$ accesses C2 (with nested C1) and Task$_L$ accesses C1.

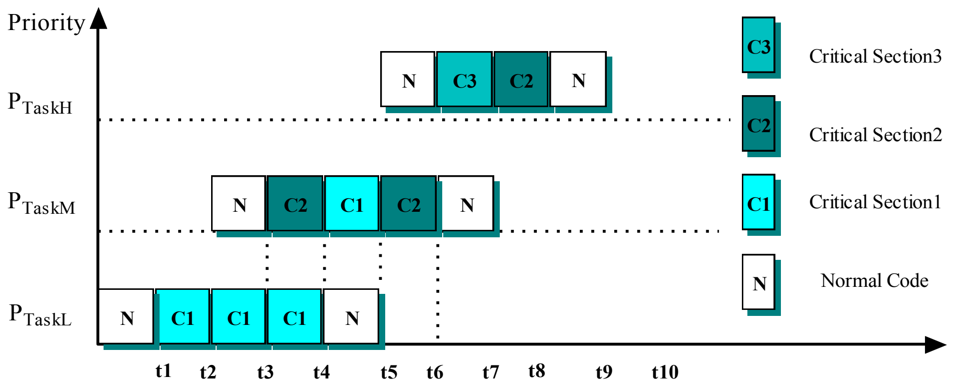


Figure 24 Operation Sequences of Three Tasks

Figure 25 shows that Task$_M$ is blocked by Task$_L$ at t_4 when it requires access to C1 (which is locked by Task$_L$). Task$_H$ is blocked by Task$_M$ at t_7 when it requires access to C2 (which is locked by Task$_M$). Therefore Task$_H$ is blocked for the duration of two critical sections (it has to wait for Task$_L$ to release C1, and then wait for Task$_M$ to release C2). As a result, a *blocking chain* is formed (Sha, Rajkumar et al., 1990).

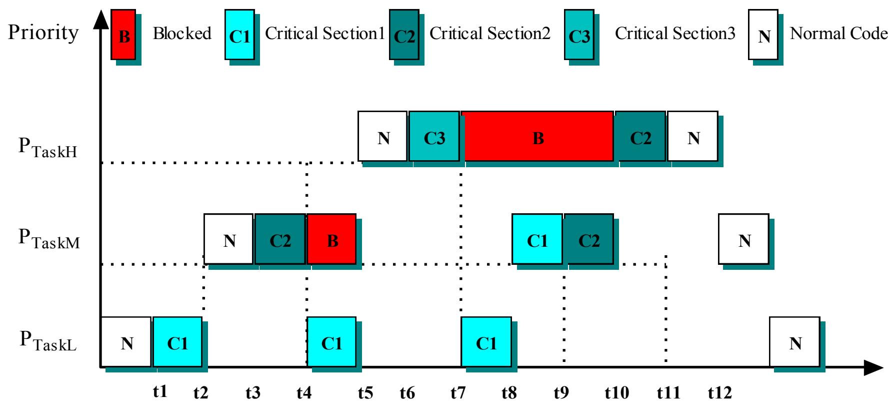


Figure 25 A blocking chain: Operation Sequences of Three Tasks without priority protocols

Solution

This pattern is intended to help answer the question: “How can you avoid conflicts over shared resources during the execution of critical sections?”

In general, the answer to this question is straightforward: you need to ensure that only one task attempts to access each shared resource at any time. There are two common ways of achieving this in a TT system architecture:

1. As noted in “Background”, you can avoid conflicts over shared resources in a time-triggered system if you use a TTC scheduler. This solution avoids the need for any of the mechanisms outlined in “Related patterns and alternative solutions”.
2. You can disable interrupts and / or using a locking mechanism (probably in conjunction with a protocol that will help you avoid priority inversions). These solutions are outlined in “Related patterns and alternative solutions”.

Of these solutions, the first is the simplest and generally the most effective. No matter what you do in a pre-emptive design to protect your shared resources, they will still be shared and only one task can use them at a time. **As such, any form of protection mechanism provides only a partial solution to the problems caused by multi-tasking.**

Consider an example. If the purpose of Task A is to read from an ADC, and Task B has locked the ADC when the Task A is invoked, then Task A cannot carry out its required activity. Use of locks, or any other mechanism, will not solve this problem; however, they may prevent the system from crashing.

Please note that there may – in some circumstances – be two further options for you to consider:

1. Pre-runtime scheduling (e.g. Xu and Parnas, 1990). Use of a “pre-run time schedule design[2]” may allow you to adapt your pre-emptive system schedule in order to ensure that – even with pre-emption – there are never conflicts over shared resources. Such techniques are not trivial to implement and are beyond the scope of the present paper.
2. Planned pre-emption. Adi and Pont (2005) have described an approach called “planned pre-emption” which avoids the need for locking mechanisms in TTH scheduler designs.

Related patterns and alternative solutions

This pattern is an abstract pattern, which provides background knowledge related to shared resources in embedded systems.

The following patterns describe some solutions to avoid shared resources conflicts and priority inversion:

DISABLE TIMER INTERRUPT

Disable interrupt is the simplest and fastest approach considered in this paper. However it may affect the response times of all other tasks in the system.

2 It can be argued that any form of static schedule (e.g. most TTC schedules) could be described as a “pre-runtime schedules”. However, this phrase is usually used to refer to static designs involving pre-emption for which a detailed modelling process is carried out prior to program execution.

Resource Lock

Lock is the most common way to protect shared resources because it affects only those tasks that need to take the same semaphore. However, basic use of locking mechanisms can give rise to problems of priority inversion.

Priority Inheritance Protocol

The Priority Inheritance Protocol is intended to address problems with priority inversion.

Improved Priority Ceiling Protocol

The Improved Priority Ceiling Protocol is intended to address problems with priority inversion, deadlock and chained blocking.

Reliability and safety implications

If a TT system is to allow task pre-emption, appropriate use of the techniques discussed in this pattern can help to make the behaviour of the system more predictable.

Overall strengths and weaknesses

☺ Being aware of the need to safeguard critical sections can help to increase system reliability

☹ Inappropriate use of locking mechanisms and related techniques may increase system complexity without increasing reliability

RESOURCE LOCK

Pattern

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can you implement a resource lock for your embedded system?

Background

We provide some relevant background material in this section.

What is a shared resource?

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION

The role of interrupts in TT systems

For background information about interrupts in TT systems, please see DISABLE TIMER INTERRUPTS

Solution

The pattern is intended to help you answer the question: "How can you implement a resource lock for your embedded system?"

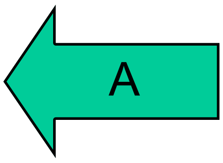
A lock appears, at first inspection, very easy to implement. Before entering the critical section of code, we 'lock' the associated resource; when we have finished with the resource we 'unlock' it. While locked, no other process may enter the critical section.

This is one way we might try to achieve this:

1. Task A checks the 'lock' for Port X it wishes to access.
2. If the section is locked, Task A waits.
3. When the port is unlocked, Task A sets the lock and then uses the port.
4. When Task A has finished with the port, it leaves the critical section and unlocks the port.

Implementing this algorithm in code also seems straightforward, as illustrated in Listing 5.

```
#define UNLOCKED   0
#define LOCKED     1

bit Lock;  // Global lock flag

// ...

// Ready to enter critical section
// - Wait for lock to become clear
// (FOR SIMPLICITY, NO TIMEOUT CAPABILITY IS SHOWN)
while(Lock == LOCKED);

// Lock is clear
// Enter critical section

// Set the lock
Lock = LOCKED;

// CRITICAL CODE HERE //

// Ready to leave critical section
// Release the lock
Lock = UNLOCKED;

// ...
```



Listing 5: Attempting to implement a simple locking mechanism in a pre-emptive scheduler.

However, the above code cannot be guaranteed to work correctly under all circumstances.

Consider the part of the code labelled ‘A’ in Listing 5. If our system is fully pre-emptive, then our task can reach this point at the same time as the scheduler performs a context switch and allows (say) Task B access to the CPU.

If Task B also requires access the Port X, we can then have a situation as follows:

- Task A has check the lock for Port X and found that the port is not locked; Task A has, however, not yet changed the lock flag.
- Task B is then ‘switched in’. Task B checks the lock flag and it is still clear. Task B sets the lock flag and begins to use Port X.
- Task A is ‘switched in’ again. As far as Task A is concerned, the port is not locked; this task therefore sets the flag, and starts to use the port, unaware that Task B is already doing so.

As we can see, this simple lock code violates the principal of mutual exclusion: that is, it allows more than one task to access a critical code section. The problem arises because it is possible for the context switch to occur after a task has checked the lock flag but before the task changes the lock flag. **In other words, the lock ‘check and set code’ (designed to control access to a critical section of code), is itself a critical section.**

This problem can be solved. For example, because it takes little time to ‘check and set’ the lock code, we can disable timer interrupt for this period (see DISABLE TIMER INTERRUPT - see this paper).

Related patterns and alternative solutions

In situations where you have more than two levels of task priority and you use a lock, you will generally need to use an appropriate locking protocol to avoid problems with priority inversion, deadlock and chained blocking. CRITICAL SECTION provides background information on these topics.

The patterns PRIORITY INHERITANCE PROTOCOL and IMPROVED PRIORITY CEILING PROTOCOL describe solutions to some of the problems caused by use of resource locks in systems with more than 2 levels of task priority.

Reliability and safety implications

As discussed in CRITICAL SECTION (this paper), use of a resource lock can give rise to problems of priority inversion. The patterns PRIORITY INHERITANCE PROTOCOL and IMPROVED PRIORITY CEILING PROTOCOL provide (partial) solutions to this problem.

Overall strengths and weaknesses

☺ Easy to implement

☹ May give rise to "priority inversion" if not implemented with care.

Disable Timer Interrupt

Pattern

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER .
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

What is the simplest way of ensuring safe access to shared resources in your system?

Background

We provide some relevant background material in this section.

What is a shared resource?

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION .

The role of interrupts in TT systems

In general, an interrupt is a signal that is used to inform the processor that an event has occurred. Such an event may include a timer overflow, completion of an A/D conversion or arrival of data in a serial port.

In TT systems, we only have a single interrupt source, linked to a timer overflow[3].

Solution

This pattern is intended to describe the simplest way of avoiding conflicts over shared resources in a TT system which involves task pre-emption.

As noted in Background, only a single interrupt is enabled in a time triggered system. This interrupt will be used to drive the scheduler (Pont, 2001). If we disable this interrupt, the scheduler will be disabled.

This gives us a simple mechanism to avoid conflicts over resources, as follows:

- When a task accesses a shared resource, it disables the timer interrupt.
- When the task has finished with the resource it re-enables the timer interrupt.
- During the time that our task is using the shared resource, the scheduler is disabled. This means that no context switch can occur, and no other task can attempt to gain access to the resource.

[3] It is possible – using a Super Loop – to create very simple TTC designs which involve no interrupts (at all). Such architectures are not suitable for use with pre-emptive task sets and are not considered in this set of patterns: see Kurian and Pont (2007) for further details.

Overall, this is a very simple (and fast) way of dealing with issues of shared resources in a TT design. However, it may have an impact on all the tasks in the system. Therefore, interrupts should be disabled as infrequently as possible (and for a very short period of time).

Related patterns and alternative solutions

The pattern CRITICAL SECTION provides general background material on mechanisms for dealing with shared resources in TT systems which involve task pre-emption.

The following patterns describe some alternative ways of handling conflicts over shared resources:

- RESOURCE LOCK
- PRIORITY INHERITANCE PROTOCOL
- IMPROVED PRIORITY CEILING PROTOCOL

Reliability and safety implications

Disabling the interrupt of a system affects the response times of the interrupt routine and of all other tasks in the system. It is not safe if it keeps interrupts disabled for long time. However, if the critical section is very short (e.g. we wish to access a single global variable), it is a fast and easy solution.

Overall strengths and weaknesses

☺ Easy to implement

☺ Faster than other protection mechanisms, such as locks

☹ Increases interrupt latency

☹ May decrease system's ability to respond to external events

☹ Need to carefully recognise the situation in which interrupts should be disabled

PRIORITY INHERITANCE PROTOCOL

Pattern

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER .
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can you ensure that access to shared resources in your system is mutually exclusive and avoids priority inversion?

Background

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION.

Solution

The pattern is intended to help you answer the question: "How can you ensure that access to shared resources in your system is mutually exclusive and avoids priority inversion?"

To avoid unbounded priority inversion, Sha et al (1990) introduced the priority inheritance protocol.

In the priority inheritance protocol, a low priority task inherits the priority of a high priority task if the high priority task requires access to the shared resource owned by the low priority task. The high priority task is blocked and the low priority task can continue executing its critical section until it releases the resource. Then its priority returns to the original and the high priority task starts to run.

This process is illustrated in Figure 26. In this example, the priority of $Task_L$ is raised to the priority of $Task_H$ once the higher-priority task tries to access the critical section (at t_3).

If an medium-priority $Task_M$ pre-empts $Task_L$ while executing the critical section, due to the fact that the priority of $Task_L$ has been raised to the priority of $Task_H$, $Task_M$ has to wait until $Task_H$ completes and $Task_L$ finishes the critical section. Therefore, the highest-priority task $Task_H$ is not pre-empted by the medium-priority $Task_M$.

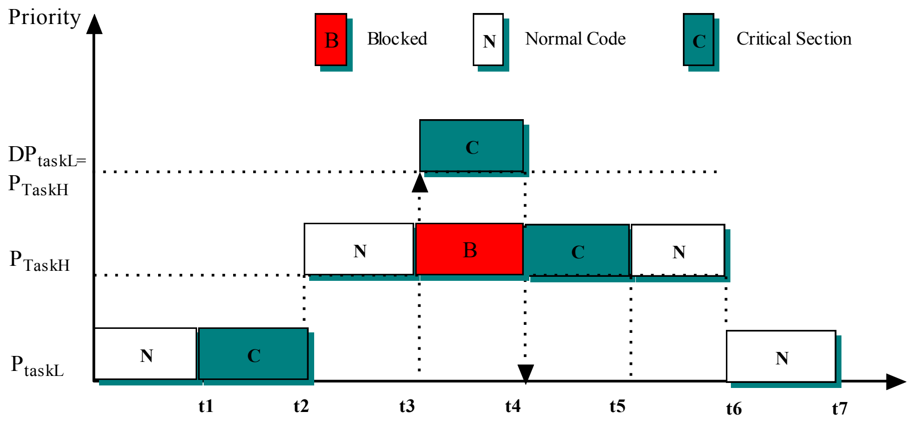


Figure 26: Priority Inheritance Protocol

Related patterns and alternative solutions

The pattern CRITICAL SECTION provides general background material on mechanisms for dealing with shared resources in TT systems which involve task pre-emption.

The following patterns describe some alternative ways of handling conflicts over shared resources:

- DISABLE TIMER INTERRUPT
- RESOURCE LOCK
- IMPROVED PRIORITY CEILING PROTOCOL

Reliability and safety implications

Use of priority inheritance protocol avoids priority inversion, increases the stability of a system. Most of commercial real time operating system support this feature, or as additional package, such as µC/OS-II, eCOS, FreeRTOS and RTLinux etc.

Although priority inheritance protocol is generally found to be an effective and powerful technique to prevent priority inversion, it is not without its critics (e.g. see (Yodaiken, 2002)).

Of particular concern is that this protocol cannot avoid deadlock and blocking chains when tasks have nested shared resources. Therefore, to use PIP safely, an appropriate software architecture design is needed that avoids unnecessary coupling between tasks through shared resources (Locke, 1992), and it is important to avoid nested resources in applications.

Overall strengths and weaknesses

☺ Prevents priority inversion

☺ Has better average–case performance than the Priority Ceiling Protocol. When a critical section is not contended, priorities do not change, there is no context switch and no additional overhead

☹ Difficult to implement when compared with DISABLE TIMER INTERRUPT (see this paper).

☹ Does not prevent deadlock and blocking chains

☹ Wastes processor time if there are not immediate tasks ready to run during the time that a higher-priority task is blocked by a lower-priority task.

☹ Worst-case performance is lower than the worst-case performance for the Priority Ceiling Protocol since nested resource locks increase the wait time.

IMPROVED PRIORITY CEILING PROTOCOL

Pattern

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER (see Pont et al., this conference).
- Your system supports task pre-emption.
- Your tasks may have nested shared resources.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can we ensure that the shared resources are mutually exclusive and that priority inversion, deadlock and blocking chains are avoided?

Background

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION.

Solution

The pattern is intended to help you answer the question: "How can we ensure that the shared resources are mutually exclusive and that priority inversion, deadlock and blocking chains are avoided?"

Nested resource locks are the underlying cause of deadlock and blocking chains. Therefore, the simplest solution is to avoid nested resource locks at the design stage (indeed, some operating systems do not allow use of nested locks).

(Sha, Rajkumar et al., 1990) presented an alternative solution to the priority inheritance protocol: this was the priority ceiling protocol (PCP). However, this original priority ceiling protocol is expensive to implement. A simplified version of the original PCP is widely used (Locke, 2002). In this pattern we refer to this as the "Improved Priority Ceiling Protocol" (IPCP)[4].

In IPCP, each task has an assigned static priority; each resource has also been assigned a priority which is the highest priority of tasks that need access it (i.e. its priority ceiling: (Burns and Wellings, 1997). When a task acquires a shared resource, the task is raised to its ceiling priority.

[4] IPCP is often referred to (incorrectly) as the priority ceiling protocol. What we refer to as IPCP here is also known as the Priority Ceiling Emulation in Real-Time Java, Priority Protect Protocol in POSIX and as the Immediate Ceiling Priority Protocol (Burns and Wellings, 1997).

Therefore, the task will not be pre-empted by any other tasks attempting to access the same resource with the same priority. When the task releases the resource, the task is returned to its original priority.

The deadlock case shown in Figure 23 is illustrated in Figure 27 to explain how IPCP works. $Task_H$ and $Task_L$ access both resources C1 and C2. Thereby the ceiling priorities of C1 (P_{c1}) and C2 (P_{c2}) are the priority of $Task_H$ (P_{TaskH}). $Task_L$ runs first. At t_1, it needs to access C2, according to IPCP, it will be raised to the ceiling priority of C1, which equals to $P_{TaskH.}$ At t2,
$Task_H$ is ready to run. However, its priority is the same as the dynamic priority of $Task_L$. It will not able to pre-empt $Task_L$ until $Task_L$ completes the critical sections and returns to the original priority. Therefore, the deadlock is prevented.

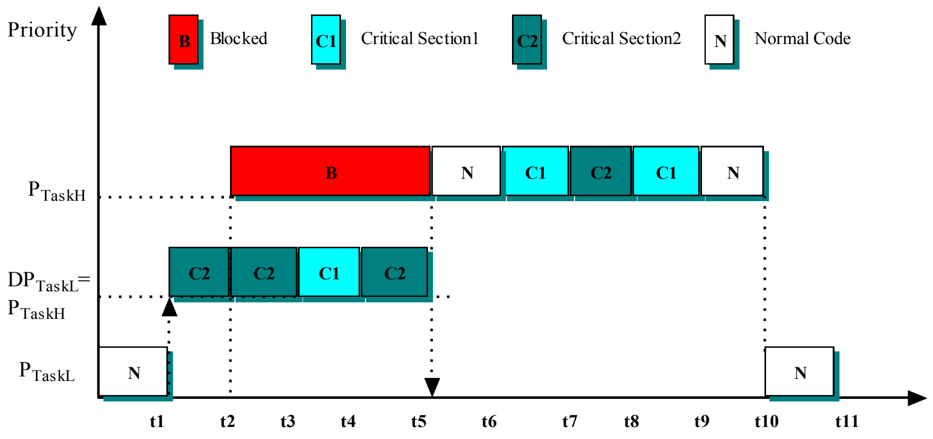


Figure 27 Operation Sequences of Tasks with IPCP

Related patterns and alternative solutions

Related patterns

The pattern CRITICAL SECTION provides general background material on mechanisms for dealing with shared resources in TT systems which involve task pre-emption.

The following patterns describe some alternative ways of handling conflicts over shared resources:
- DISABLE TIMER INTERRUPT
- RESOURCE LOCK
- PRIORITY INHERITANCE PROTOCOL

Alternative solutions: IPCP vs. PCP

It is helpful to understand the differences in behaviour obtained when using PCP and IPCP

When using PCP, each task has an assigned static priority; each resource has also been assigned a priority (which is the highest priority of tasks that need access it, i.e. its priority ceiling).

IPCP operates in a similar same way, but there are two differences. The first difference is that each task's dynamic priority is the maximum of its own static priority and its inheritance priority due to it blocking higher-priority tasks. The second difference is that a task can only lock a resource if its dynamic priority is higher than the ceiling priority of any currently-locked resource (Burns and Wellings, 1997).

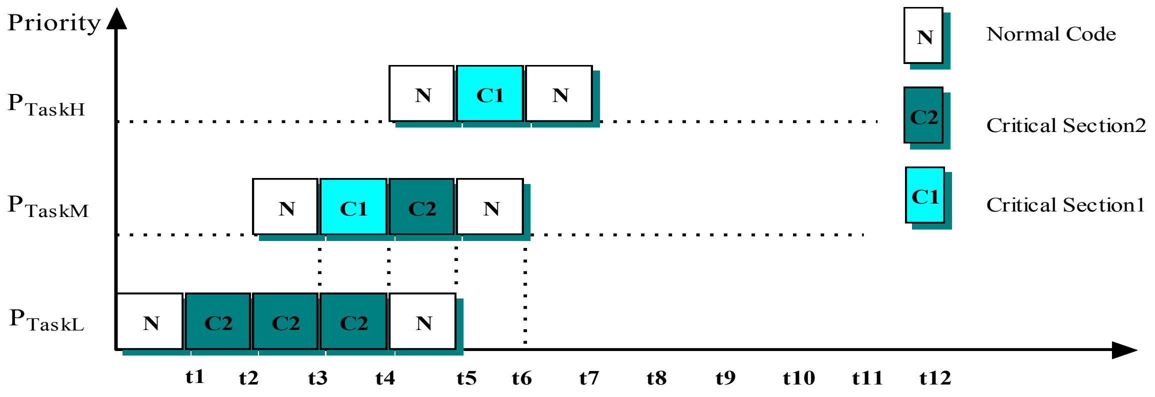


Figure 28 Operational sequences of three tasks with two shared resources

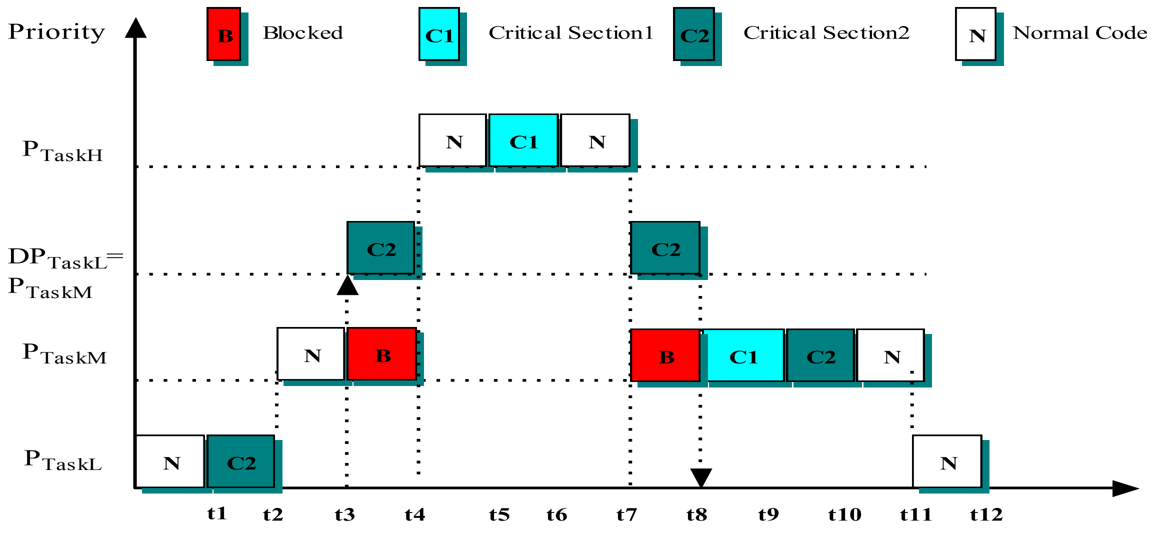


Figure 29 Operational sequences of three tasks with PCP

Figure 29 illustrates how PCP works in a specific example. Three tasks are intend to run as shown in Figure 28. At t2, Task$_M$ pre-empts Task$_L$, at t3 Task$_M$ is attempting to lock the resource C1. However, due to the fact that its dynamic priority P$_{TaskM}$ is not higher than the ceiling priority of C2 (P$_{c2}$ = P$_{TaskM}$) - which is currently locked by Task$_L$ - it cannot lock C1, and is blocked by Task$_L$. Task$_L$ inherits Task$_M$ priority due to it blocking Task$_M$ and continues running at a higher priority. At t4 Task$_H$ starts to run and at t5, it is attempting to lock C1: because its priority is higher than P$_{c2}$, it successfully locks C1 and runs to completion. After Task$_L$ releases C2 and returns to its original priority at t8, Task$_M$ locks C1 and runs to completion.

The behaviour obtained using IPCP in this case is illustrated in Figure 30. From Figure 28 and Figure 29 it is seen that there are 6 context switches using PCP and 4 times context switches when using IPCP. In addition, PCP needs to check blocking information for a task's dynamic priority: this makes PCP is more difficult to implement.

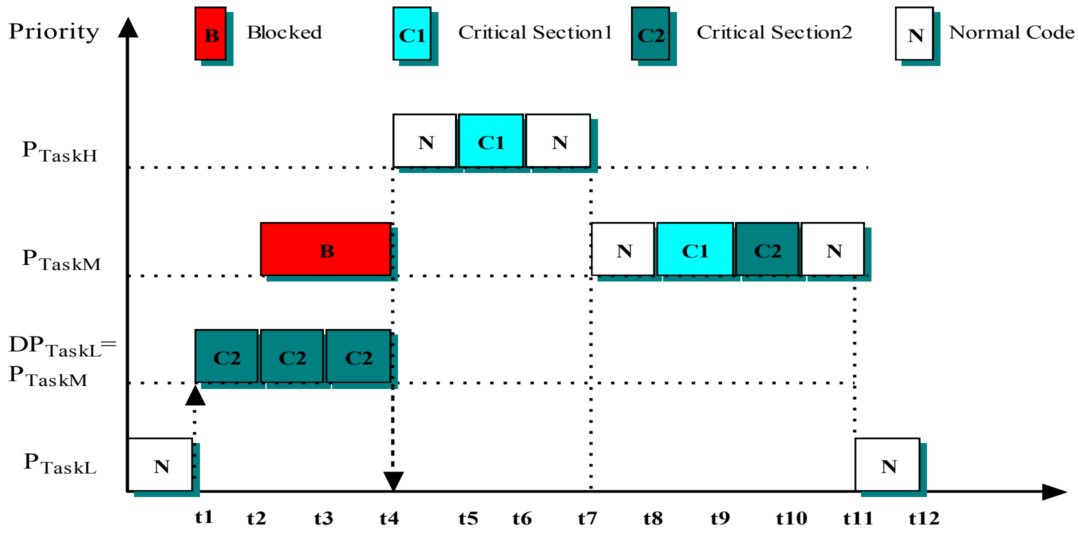


Figure 30 Operational sequences of three tasks with IPCP

Alternative solutions: Combining tasks

Finally, it is worth noting that the problems caused when several tasks use the same resource can – sometimes - be solved most simply by combining the different tasks into a single task (Renwick, 2004).

Reliability and safety implications

IPCP is an attractive choice when there may be nested locks among tasks. Preventing deadlock and blocking chains can increase the stability of a system.

Compared with DISABLE TIMER INTERRUPT, IPCP is more difficult to implement (and test). An appropriate software architecture design is needed that avoids unnecessary coupling between tasks through shared resources (Locke, 2002), and avoids nested resources (if possible).

Overall strengths and weaknesses

☺ Prevents priority inversion

☺ Prevents deadlock and blocking chains

☺ Has better worst-case performance than PIP. The worst-case wait time for a high priority task waiting for a shared resource is limited to the longest critical section of any lower priority tasks that accesses the shared resource.

☹ Difficult to implement

☹ Requires static analysis of a system to find the priority ceiling of each critical section.

☹ Average –case performance is worse than PIP. IPCP changes a task’s priority when it requires a resource, regardless of whether there is contention for the resource or not, resulting in higher overhead and many unnecessary context switches and blocking in unrelated tasks (Locke, 2002)

BALANCED SYSTEM

Abstract pattern

Context

- You are developing an embedded system.
- You have decided to move to or are already working with TT architectures.
- Predictable timing behaviour is the key requirement.

Problem

How can you ensure that your TT system has minimum possible jitter?

Background

To get a guaranteed predictable system, choice of appropriate architecture on top of the application is extremely important. In applications which are based on an ET architecture, tasks run sporadically in response to interrupts whereas in a TT system, tasks are periodic. However, there is a single interrupt which generates "ticks" to control the task periods.

To achieve certification standards it is advisable to avoid the use of arbitrary interrupts in running the tasks because of the increased difficulty in attaining sufficient test coverage. One particular reason is, arbitrary interruptions lead to a vast increase in the potential paths within software when compared to code with no interruptions (Bate, 1998).

From the predictability point of view, this would make TT architecture an appropriate choice for a number of applications.

Only choosing TT architecture does not fully guarantee the system predictability as there are a number of other factors which could make a TT system unpredictable. The parameters of tasks which are running under a TT architecture such as release time, execution time, finish time and deadline are required to be known in advance. The prior knowledge of these parameters plays an important role in guaranteeing the overall predictability of the system. However, systems that run in practice generally show considerable variations in these parameters. These variations are termed as jitter.

Jitter in tasks

To understand the concept of jitter more clearly, consider the different instances of a task (Task A) as shown in Figure 31. For tasks in TT systems, release time can be considered as the point at which we would ideally expect a task to start its execution. In actual practice this is delayed due to factors such as scheduler overhead and variable interrupt response times (Liu, 2000; Maaita and Pont 2005). The actual start time of a task is always deviated from its (pre-determined) release time and we can say that tasks always suffer from release jitter - see unequal values of x1, x2 and x3 in Figure 31

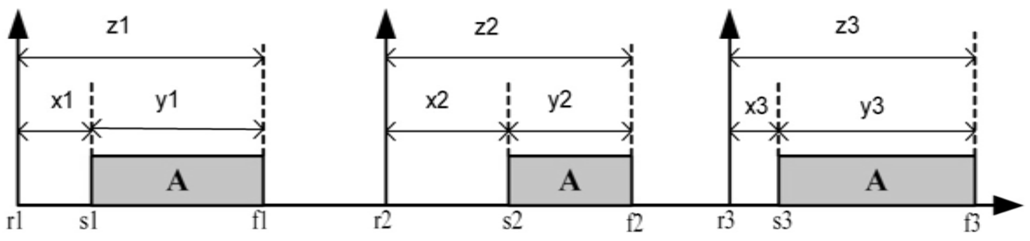


Figure 31: Illustration of jitter in different calls of a periodic task

In real-time systems one important parameter is the upper bound of the execution time for a task, known as worst case execution time (WCET). Unfortunately, determining WCET of tasks is rarely straightforward (Puschner, 2002b; Puschner, 2003). This is because the program code of a task may contain conditional branches and / or loops and each may take different times to execute (Liu, 2000). The decision between one branch and the other during task execution is dependent on the input data. This makes predicting a branch prior to execution a very difficult task . All these factors lead to variable execution time of a task and this is known as execution jitter (see imbalanced values of y1, y2 and y3 in Figure 31). The cascading effects of release and execution jitter will result in the deviation of task finish time, shown as z1, z2 and z3 in Figure 31.

Ideally, a predictable system should be jitter free. Considering Figure 31 once again, we can say that in a zero jitter system:

$$x1 = x2 = x3$$
$$y1 = y2 = y3$$
$$z1 = z2 = z3$$

Some hardware features such as variations in the frequencies of oscillator and use of cache memories (Kirner and Puschner, 2003) also contribute to jitter in tasks.

For some applications, such as data, speech or music playback (for example) these variations may make no measurable difference to the system. However, for applications in real-time control systems which involve sampling, computation and actuation, such delays in operations are very risky for the overall performance of the system. The presence of jitter can have a degrading impact on the performance of real-time systems or can even lead to critical failure (Martin, 2005).

Solution

A BALANCED SYSTEM is one which has minimum values for all types of jitters. One way we can address the challenges discussed in the previous section is to tackle them directly. For example, rather than hoping that we can predict the WCET (for example, through static code analysis or measurement) we can set out from the start to ensure that our code is “balanced” and that the WCET and BCET (best case execution time) are always fixed (and equal). Once we have balanced the code, it becomes comparatively easy to determine (during system testing and during system execution) whether the system tasks actually have a fixed execution time.

Related patterns

In a task, balancing can be required at different levels. For example, we may need to balance the whole task or just sections (for example, areas with loops and conditional code). In some cases, the goal may be to fix the timing of an activity in the task relative to the start of the task (for example, we may wish to ensure that exactly 0.2 ms after the start of a task a sample is taken from a data source).

- SANDWICH DELAY provides a simple solution to balance a task through exclusive use of a hardware timer.
- SINGLE PATH DELAY is a programming approach to ensure that blocks of code involving loops or decision structures will have a single execution path.
- TAKE A NAP is an alternative to achieving balanced code for power constrained systems.
- PLANNED PRE-EMPTION provides a way of achieving balancing for pre-emptive systems.

Reliability and safety implications

Extra care is needed while selecting the tasks/sections of code to be balanced. This is because balancing makes use of additional hardware and software. Devoting resources unnecessarily to balance tasks which are not critical could lead to a fatality rather than a predictable system.

Overall strengths and weaknesses

☺ A BALANCED SYSTEM is more robust against the presence of various types of jitters in the system.

☺ Results in a more predictable timing behaviour of the system.

☹ Requires (non-exclusive) access to some hardware resources, for example, timers.

☹ Balancing a system requires extra effort in writing code for balancing which in turn increases CPU utilisation.

SANDWICH DELAY

Pattern

Context

- You are using the pattern BALANCED SYSTEM.
- In your application you are running two activities, one after the other.

Problem

How can you ensure that the execution time of the tasks is always predictable so that the release time of the two activities is known and fixed?

Background

Suppose we have a system executing two functions periodically using a timer ISR, as outlined in Listing 6.

```
//Interrupt service Routine (ISR) invoked by timer overflow every 10ms
      void Timer_ISR (void)
      {
            Do_X();    //WCET approx 4.0ms
            Do_Y();    //WCET approx 4.0ms
      }
```

Listing 6: System executing two functions using timer ISR

According to Listing 1, function `Do_X()` will be executed every 10ms. Similarly, function `Do_Y()` will be executed every 10 ms, after `Do_X()` completes. For many resource-constrained applications (for example, control systems) this architecture may be appropriate. However, in some cases, the risk of jitter in the start times of function `Do_Y()` may cause problems. Such jitter will arise if there is any variation in the duration of function `Do_X()`. In Figure 32, the jitter is reflected in differences between the values of *ty1* and *ty2* (for example).

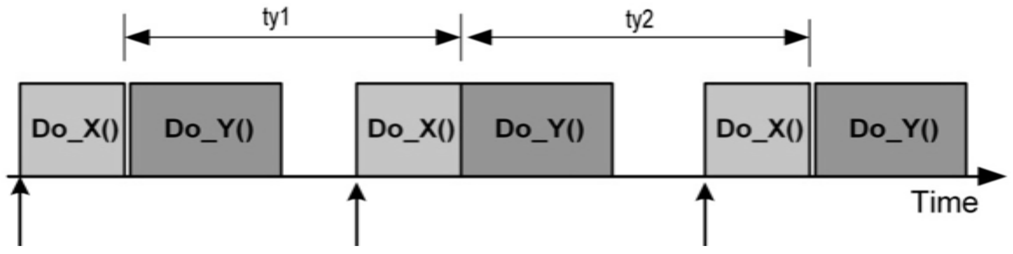


Figure 32: The impact of variations in the duration of `Do_X()` on the release jitter of `Do_Y()`

Solution

A SANDWICH DELAY can be used to solve this type of problem. More specifically, a SANDWICH DELAY provides a simple but highly effective means of ensuring that a particular piece of code always takes the same period of time to execute: this is done using two timer operations to "sandwich" the activity you need to perform. Please refer to code segment in Listing 7.

```
//ISR invoked by timer overflow every 10ms
void Timer_ISR (void)
   {
   //Execute Do_X() in a 'Sandwich Delay'  - BEGIN
   Set_Sandwich_Timer_overflow(5); //Set timer to overflow after 5 ms
   Do_X();    //Execute Do_X()  WCET approx 4.0ms
   Wait_Sandwich_Timer_Overflow(); //Wait for timer to overflow
   //Execute Do_X() in a 'Sandwich Delay'  - END

   Do_Y();    //WCET approx 4.0ms
   }
```

Listing 7: Pseudo code for SANDWICH DELAY

The timer is set to overflow after 5 ms (a period slightly longer than the WCET of `Do_X()`). We then start this timer before we run the function and - after the function is complete - we wait for the timer to reach 5 ms value. In this way, we ensure that as long as `Do_X()` does not exceed a duration of 5 ms – `Do_Y()` runs with minimum jitter as shown in Figure 33.

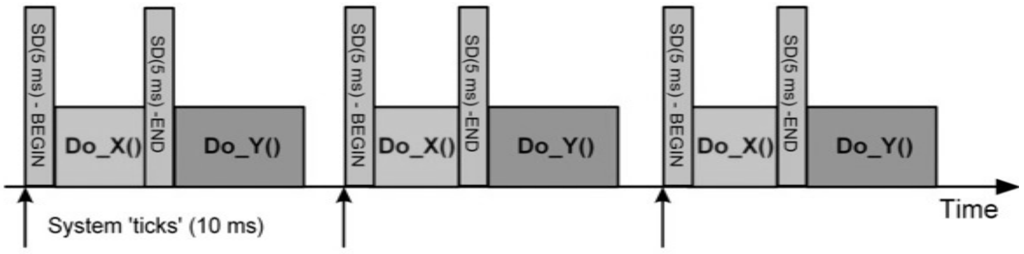


Figure 33: Reducing the impact of variations in the durations of `Do_X()` on the release jitter of `Do_Y()` through the use of SANDWICH DELAY

Sandwich delays are also found to be useful for systems involving pre-emption, for example, TTH designs. In such designs controlling the execution jitter of a pre-emptive task using a delay (slightly bigger than the WCET of the pre-emptive task) showed considerable reduction in the period jitter of the co-operative task - see Figure 34.

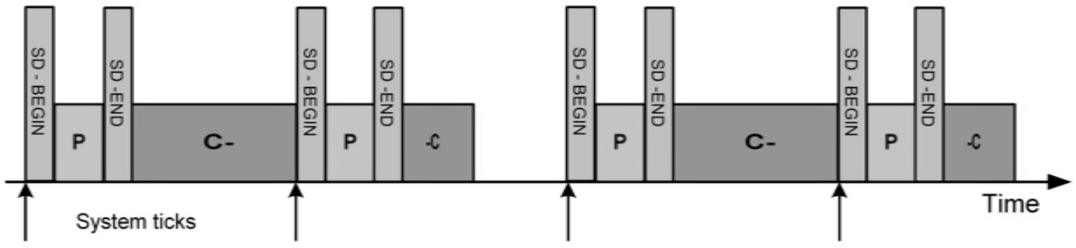


Figure 34: Reducing the impact of variations in the durations of pre-emptive task on the release jitter of co-operative tasks through the use of SANDWICH DELAY in TTH designs

Reliability and safety implications

Use of Sandwich Delay is generally straight forward, but there are three potential issues of which you should be aware.

1. You need to know the duration WCET of the functions to be sandwiched. If you underestimate this value, the timer will already have reached its overflow value when your function(s) complete, and the level of jitter will not be reduced (indeed the Sandwich Delay is likely to slightly increase the jitter in this case)
2. You must check the code carefully, because the "wait" function may never terminate if the timer is incorrectly set up. In these circumstances a monitoring technique may help to rescue the system. See patterns SYSTEM MONITORS, WATCHDOG, LOOP TIMEOUT and TASK GUARDIAN
3. You will rarely manage to remove all jitter using such an approach, because the system cannot react instantly when the timer reaches its maximum value (at the machine-code level, the code used to poll the timer flag is more complex than it may appear, and the time taken to react to the flag change will vary slightly). A useful rule of thumb is that jitter levels of around 1 microsecond will still be seen using a SANDWICH DELAY.

Overall strengths and weaknesses

☺ A simple way of ensuring that the WCET of a block of code is highly predictable.

☹ Requires (non-exclusive) access to a timer.

☹ Will only rarely provide a "jitter free" solution: variations in code duration of around 1 microsecond are representative.

SINGLE PATH

Context

- You are using the pattern BALANCED SYSTEM.
- You have decided to balance sections of code involving loops and decision structures implemented within the application tasks.

Problem

How would you ensure that the execution time of your application code sections involving loop and decision structures will remain fixed every time they run?

Background

Variable execution times of tasks can lead to unpredictable behaviour in systems. To understand this more clearly, consider a system running tasks A, B and C as shown in Figure 35.

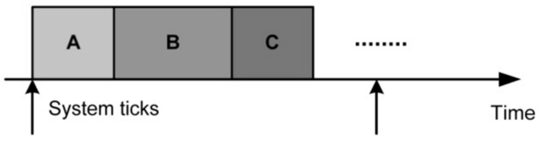


Figure 35: Tasks scheduled to be run in a TT system

If for any reason, task A takes a longer time to run than expected, task C will run before task B (if it has higher priority than task B) and task B will not be able to finish within the system tick as shown in Figure 36.

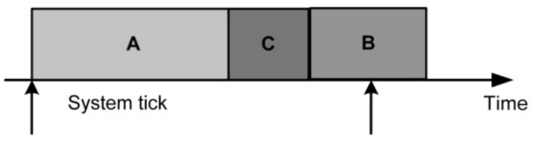


Figure 36: Illustration of overall change in system behaviour if the execution time of task A takes longer than expected

The point to be noted here is, if task A varies in duration it will affect the overall system behaviour. Tasks involving loops and decision structures (e.g., 'if-else', 'switch', etc.) are more likely to have variable execution times. If such tasks can be balanced, we can achieve more stable and predictable system behaviour.

Solution

SINGLE PATH helps to achieve fixed execution time for tasks involving decision structures and loop statements. The single-path programming approach was introduced by Peter Puschner (Puschner, 2003) as part of his extensive research on WCET analysis. According to single-path programming paradigm, programs that involve loops and decision structures (e.g., 'if-else') will have a single execution path. This could be achieved at the expense of higher but fixed and predictable execution time as compared to traditional programming. Single-path can be achieved by replacing input-data dependencies in the control flow by predicated code instead of branched code. Thus, the instructions are associated with predicates and get executed if the predicate evaluates to true. In other case (if instruction evaluates to false), the microprocessor replaces the instruction with a NOP (no-operation) instruction.

Translation of Conditionals

Consider a piece of code where the developer is using an `if` statement to check whether or not a particular condition is true, as shown in the left hand side code segment in Listing 3. If the condition being evaluated (`cond`) is true, the value of the variable `result` is set to `expr1` otherwise the value of `result` is set to `expr2`. As we cannot be sure which of the two expressions (`expr1` or `expr2`) will be calculated, or in other words, which execution path the code will follow, it becomes difficult to predict the execution time of the section of the task with the conditional statement.

Using SINGLE PATH DELAY, we assign temporary variables `temp1` and `temp2` for storing the results of `expr1` and `expr2` respectively. The conditional move instruction "`movt`" copies the value of `temp1` to the variable `result` if the test condition evaluates to true, otherwise processor performs a "`no operation`" (NOP) instruction. On the other hand, if the test condition evaluates to false, "`movf`" will copy the value of `temp2` to `result` otherwise NOP instruction will be executed. In this way the translation basically generates a sequential code as shown in the right hand side code segment in Listing 8.

```
if(cond)                            temp1 = expr1;
{                                   temp2 = expr2;
  result = expr1;
 }                    ⇨             test cond;
else
{                                   movt result, temp1;
  result = expr2;                   movf result, temp2;
}
```

Listing 8: Sequential code generated from a branching statement using if-conversion [adapted from Puschner, 2003]

Translation of loops

Consider a `while` loop as shown in Listing 9 which executes a set of statements based on two conditions being 'true' – a pre-condition, `cond-old` and a condition, `cond-new`.

```
--precondition: cond-old
while cond-new do max expr times

        {
```

```
stmts
}
```

Listing 9: Original `while` loop [adapted from Puschner, 2003]

To translate this loop to have a single path of execution, a boolean variable `finished` is introduced – this variable stores information as to whether the original loop has executed the current iteration or has already terminated. The `while` loop shown in Listing 9 above can be translated as follows (Puschner, 2003) :

1. First the loop is translated to a simple counting loop (e.g., a `for` loop) with the iteration count set to be equal to the maximum iteration count of the original loop (in this case, `expr`).

2. The pre-condition, `cond-old`, is used to build a new branching statement inside the new loop.

3. A new conditional statement that has been generated from the old loop condition (`cond-new`) is transformed into a conditional assignment (using the newly introduced boolean variable `finished`) with constant execution time. As a result, the entire loop executes in constant time.

   This new conditional statement is placed around the body of the original loop and simulates the data dependent termination of the original loop in the newly generated counting loop.

```
finished := false;
for i := 1 to expr do
begin
      if not cond-new
      then finished := true;

      if cond-old and not finished
      then stmts
end
```

Listing 10: `while` loop with a constant execution count [adapted from Puschner, 2003]

Overall strengths and weaknesses

☺ Helps to produce constant execution time for code sections involving loops and conditional statements.

☹ Its use is limited to hardware which supports "conditional move" or similar instructions.

☹ It is likely to increase the power consumption because the CPU will always execute the single-path code for a fixed (maximum) period. During this time, the processor will be in "full power" mode.

TAKE A NAP

Pattern

Context

- You are using the pattern BALANCED SYSTEM.
- You are using a system which is extremely power constrained.

Problem

How would you ensure the WCET of your application code sections involving loop and decision structures remains constant with negligible increase in power consumption?

Background

SANDWICH DELAY and SINGLE PATH DELAY provide ways to achieve fixed execution time. In systems where power consumption is a concern, neither a SANDWICH DELAY nor a SINGLE PATH DELAY is an attractive solution, because – to achieve balanced code – we need to run the CPU at "full power" at all times. For such systems we need to find out a way to achieve balanced code without any extra power consumption.

Solution

For systems which are extremely resource constrained (especially power) TAKE A NAP provides a way to achieve balanced code with reduced power consumption.

Create balanced code by putting the control flow statement within a 'Sandwich Delay' (see pattern SANDWICH DELAY). This will ensure that the particular piece of code will always have a constant execution time. For example consider the code segment given in Listing 11.

```
for (i = 0; i < x;  i++)
{
  // body of the loop
}
```

Listing 11: Simple For Loop

The execution time of the loop is dependent on the value of the variable `x`. Let `MAX` be equal to the maximum number of iterations the loop can execute. Let `Time(x)` be equal to the time spent in executing `x` iterations. The value of `Time(x)` may be measured using hardware timers. Therefore, the time spent in performing `(MAX - x)` iterations may be calculated using the value of `Time(x)` as follows:

```
Time (MAX - x) = (MAX - x) * Time(x)/x    (1)
```

Once the `for` loop executes `x` number of times, the processor is put to sleep for a duration equal to `Time (MAX - x)`. A timer interrupt may be generated when the hardware timer count reaches

the value `Time(MAX - x)` and this can be used to awaken the processor. Using this technique, code segment in Listing 6 is ensured to always – irrespective of the value of `x` – have a constant execution time equal to the value of `Time(MAX)` (i.e. the time spent in executing `MAX` number of iterations of the `for` loop). Thus, in addition to enabling a 'power – saving mode of the processor, the resulting 'balanced' code with the SANDWICH DELAY incorporated, provides an additional layer of predictability to the real-time system.
The balanced version of the code segment in Listing 11 may be written as shown in Listing 12.

```
//start the timer
Timer_Start();

for( i = 0; i < x; i++)
{
   //body of the loop
}
//stop the timer
Timer_Stop();
//Store timer count value after x iterations
Time(x) = Timer_Count_Value;
//Determine value of Time(MAX – x)
Time(MAX – x) = (MAX-x) * Time(x)/x;
//Reset the timer
Timer_Reset();
//Set the timer interrupt to occur after duration Time(MAX–x)
Set_Timer_Intterrupt(Time(MAX-x) + "safety margin");
//Put processor to sleep
Processor_Sleep();
```

Listing 12: Balancing of sections with reduced power consumption

It must be noted that the `for` loop in code segment above must run at least once for the value of `Time(MAX - x)` to be determined. Furthermore, a small 'safety margin' has been added to the calculated time to ensure that there is sufficient time for the processor to enter sleep mode even when the loop is executed for the maximum number of iterations.

TAKE A NAP may also be applied to other control flow and conditional branching statements such as while, if-else and switch.

Overall strengths and weaknesses

☺ A simple technique for improving system reliability by providing an additional layer of predictability is described here.

☺ Ensures fixed execution time for each task in the system along with reduced power consumption.

☹ The maximum number of iterations of the control flow statement (i.e. the value of `MAX`) must be known in advance.

☹ Requires exclusive access to a hardware timer.

PLANNED PRE-EMPTION

Pattern

Context

- You are using the pattern BALANCED SYSTEM.
- Your system is based on a time-triggered scheduler – specifically on TTH architecture.

Problem

How would you ensure the predictable scheduler behaviour in TTH designs?

Background

Some background material related to this pattern is already presented in the introduction of this report (see section TTH design).

During normal operation of the systems using the TTH Scheduler architecture, function `main()` runs an endless `while` loop (see Listing 13) from which the function `C_Dispatch()` is called: this in turn launches the co-operative task(s) currently scheduled to execute. Once these tasks are completed, `C_Dispatch()` calls `Sleep()`, placing the processor into a suitable "idle" mode.

```
while(1)
   {
   C_Dispatch();  // Dispatch Co-op tasks
   }

void C_Dispatch(void)
   {
   // Go through the task array
   // Execute Co-operative tasks as required
   // The scheduler may enter idle mode at this point
   Sleep();
   }

void P_Dispatch_ISR(void)
   {
   P_Task();
   }
```

Listing 13: TTH Scheduler

A hybrid scheduler provides limited multi-tasking capabilities to the system. Such systems could exhibit unpredictable behaviour because of two reasons (Maaita, 2008):
Existence of unbalanced code branches in the timer ISR which leads to variable ISR execution times. This in turn leads to unpredictable scheduler behaviour represented by the appearance of task starting jitter.

1. The existence of CPU instructions with different execution times (i.e. in terms of CPU cycles required to execute the instruction). This leads to variable timer interrupt response times as each of the periodic timer interrupts which take place throughout the life cycle of the application can occur while the CPU is in one of the two different states. The CPU may

either be running in sleep (idle) mode shown in Figure 37, or while it is running an instruction and where the interrupt is only serviced once the currently executing instruction is finished shown in Figure 38.

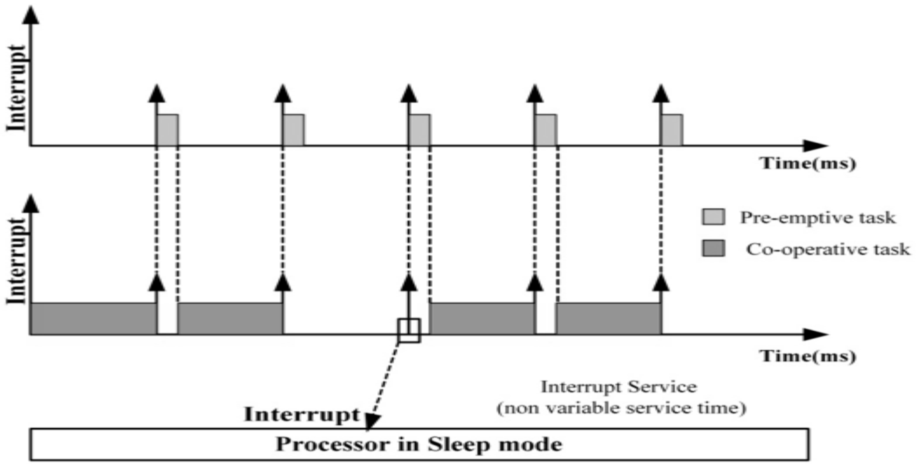


Figure 37: Timer Interrupt when CPU is in sleep mode

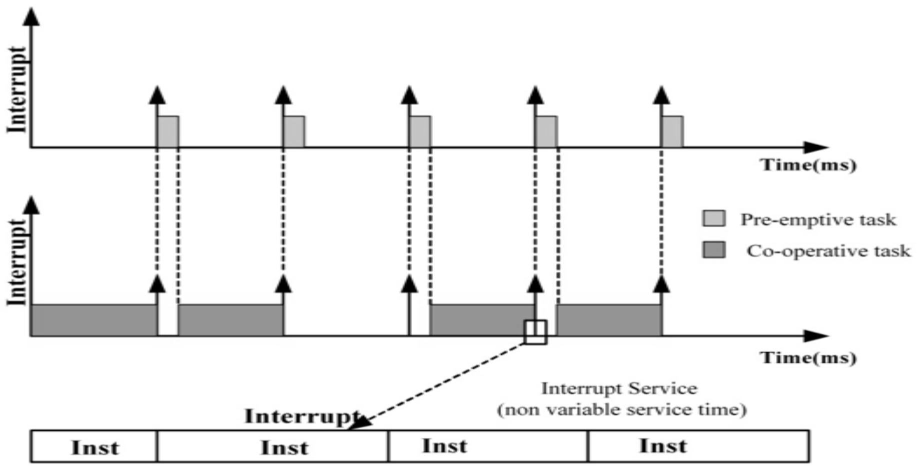


Figure 38: Timer Interrupt during an instruction execution

The possible occurrences of timer interrupts could lead to variable timer ISR response times which translates in to task release jitter. In TTH design this release jitter has the largest impact on tasks which regularly execute after a timer tick has occurred and is, therefore, referred to as "tick jitter".

Solution

By keeping the processor in the same state as all interrupts takes place would be likely to reduce the tick jitter (Maaita and Pont 2005). PLANNED PRE-EMPTION makes use of another hardware timer to put the processor to power saving mode before the scheduler timer interrupt occurs thus keeping the processor in the same state every time. Power saving mode or sleep/idle mode is available in almost all embedded processor for example ARM7 and 8051 family of processors.

We are naming the extra timer used for this purpose as "PP-timer" being use for PLANNED PRE-EMPTION. To set the overflow value of the PP-timer it is important to know the WCET in advance so that the processor can have enough time to go to sleep mode before the scheduler timer interrupt occurs. PLANNED PRE-EMPTION will reduce the tick jitter as the time required to leave the sleep mode and pursue normal execution is a static value (Martin, 2005). Figure 39 illustrates the implementation of PLANNED PRE-EMPTION.

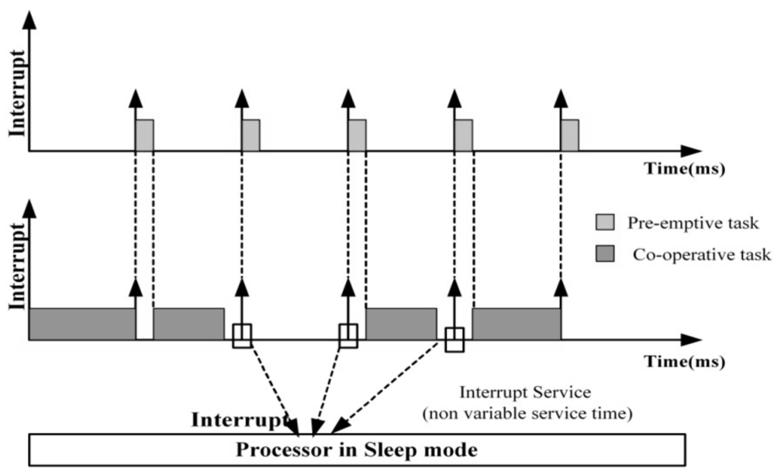


Figure 39: Operation of Planned pre-emption. All interrupts occur when the processor is in sleep mode

```
while(1)
      {
      C_Dispatch();  // Dispatch Co-op tasks
      }
      void C_Dispatch(void)
      {
      //Go through the task array
      //Execute Co-operative tasks as required
      //The scheduler may enter idle mode at this point
      Sleep();
      }
      void P_Dispatch_ISR(void)
      {
      ITimer();  //Start idle timer
      P_Task();  //Dispatch pre-emptive task
      }
      void Idle_Timer_ISR(void)
      {
      Sleep();
      }
```

Listing 14: TTH-PP Scheduler

Reliability and safety implications

Designers have to be careful while using the second timer. The timer should overflow after most of the interval between "pre-emptive ticks" has elapsed. A more efficient implementation in terms of the hardware utilised is to use a second match register on the original scheduler timer. For example, ARM7TDMI supports multiple match registers per timer (UM10211, 2009) .

Overall strengths and weaknesses

☺ Produces a more predictable TTH system.

☺ Provides a simple way of getting non variable timer interrupt response times which reduce the tick jitter.

- ☹ Makes use of exclusive hardware timer.
- ☹ Slight increase in memory requirements because of increased code size than normal TTH scheduler.

SYSTEM MONITOR

Abstract pattern

Context

- You are in the process of creating or upgrading an embedded system, based on a single processor.
- Because predictable and highly-reliable system operation is a key design requirement, you have opted to employ a "time-triggered" system architecture in your system.

Problem

How will you make sure that your system will not 'hang' and will keep on functioning despite unfavourable conditions?

Background

If your application is to be reliable, you need to be able to guarantee that the system should be capable of handling situations which could possibly hang the system. Some of such possibilities are:

- Incorrect initialisation of hardware peripherals or variables associated with hardware for example ADC or DAC
- Hardware devices may be subjected to an excessive input voltage and may not work at all
- Task overrun in the system i.e. if a task exceeds its estimated execution time it will disturb the entire schedule

Solution

In order to ensure the system reliability, SYSTEM MONITORS can be implemented to keep an eye on system functionality. They act like guards to be responsible to make sure that system is working fine and can take appropriate actions if anything unexpected is detected with the system. Such SYSTEM MONITORS can be implemented with use of hardware for example timers (see pattern WATCHDOG) or a separate software task can be design for this purpose (see pattern TASK GUARDIAN). Also, there are some good programming practices which could help to avoid possibilities of hanging system see pattern (LOOP TIMEOUT).

Related patterns and alternative solutions

- WATCHDOG
- LOOP TIMEOUT
- TASK GUARDIANS

Overall strengths and weaknesses

☺ System monitors provide a way of ensuring system reliability

☹ Require use of additional hardware or code to implement monitoring techniques.

☹ Implementing such techniques requires care

Context

- You are using the pattern SYSTEM MONITORS.
- Your application is based on simple TTC design

Problem

How would you ensure that any task overrun in the system should be detected and handled so that system can continue functioning properly?

Background

Despite many advantages, a pure TTC architecture has a failure mode which has the potential to greatly impair system performance: this failure mode relates to the possibility of task overruns (see Figure 40).

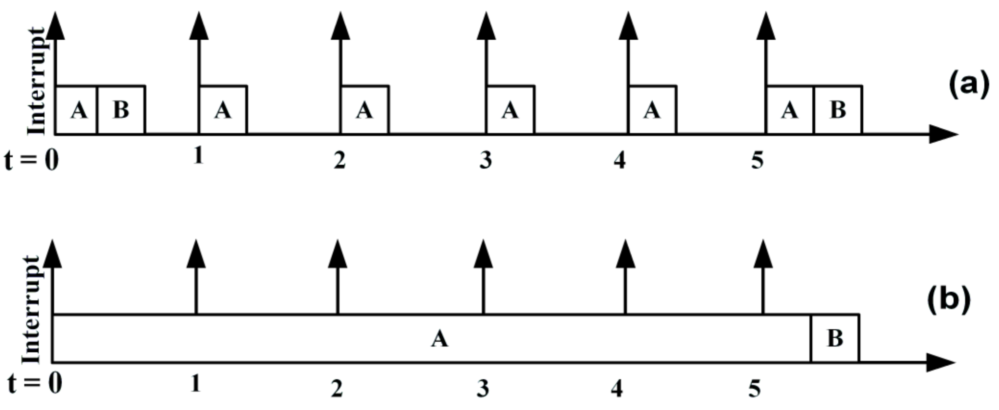


Figure 40: Illustrating the impact of task overrun on TTC based system

Figure 40(a) illustrates a TTCS design running two tasks, A and B. Task A runs every millisecond and Task B runs every 5ms. This system operates as required, since the duration of Task A never exceeds 0.4 ms. Figure 40(b) illustrates the problems that result when Task A overruns: in this case, we assume that the duration of Task A increases to approximately 5.5ms. The co-operative nature of the scheduling in this architecture means that this task overrun has very serious consequences.

In practice, the situation may be even more extreme: in this example, if Task A never completes, then Task B will never run again.

During normal operation of the TTC architecture described by (Pont, 2001), the first function to be run (after the startup code) is Main. Main calls Dispatch which in turn launches any tasks which are currently scheduled to execute. Once these tasks are completed, Dispatch calls Sleep, placing the processor into a suitable "idle" mode. A timer-based interrupt occurs every millisecond (in most implementations) which wakes the processor up from the idle state and invokes the ISR Update. Update identifies tasks which are due to be launched during the next execution of Dispatch. The function calls return all the way back to Main, and Dispatch is called again. The cycle thereby continues (Figure 41).

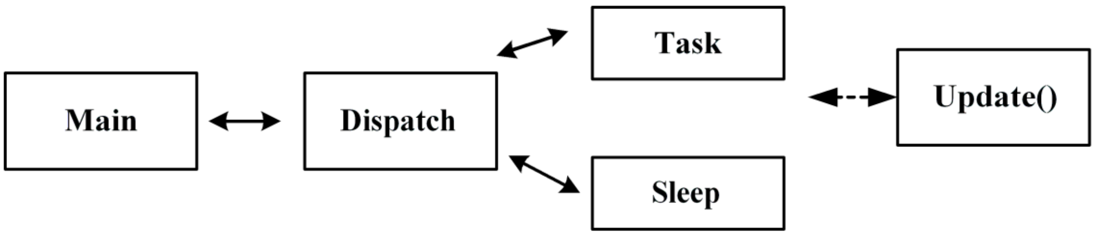


Figure 41: Function call tree for TTC architecture (normal operation)

If a task overrun occurs, then - instead of Sleep being interrupted by the ISR - the overrunning task is interrupted (Figure 42).

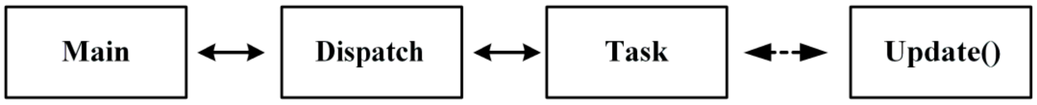


Figure 42: Function call tree for TTC architecture (task overrun)

If Task keeps running then - in the standard TTCS scheduler - it will be periodically (very briefly) interrupted by Update. However, it cannot be shut down.

Solution

The TASK GUARDIAN approach is proposed in (Hughes and Pont 2004) implemented mainly in a revised version of Update - is required to shut down any task which is found to be executing when the Update ISR is invoked.

This will be carried out as follows (see Figure 43):

- The Update ISR will detect the task overrun.
- Update will return control to End_Task (rather than to the problematic task). End_Task will be responsible for unwinding the stack, as required, to locate the return address to Dispatch.
- Control will then be returned to Dispatch, and - as far as possible - normal program operation will continue.

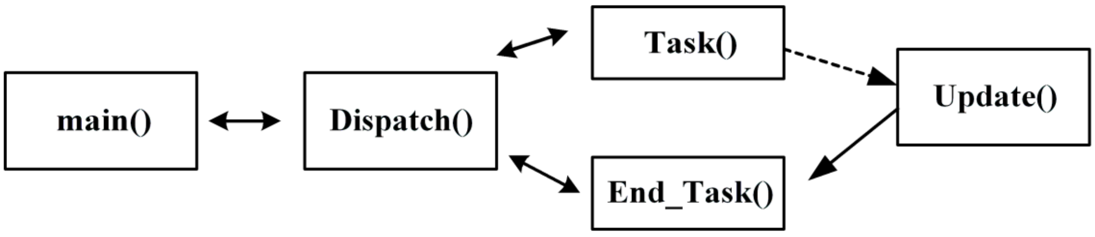


Figure 43: Task Guardian mechanism

Detection of task overrun

In order for the update task function to know that an overrun has occurred and take appropriate action, a simple and reliable method is required to detect overruns. Modifying the code in Dispatch, where the tasks are launched, enables this to be achieved.

Using a variable such as 'Taskover' the task ID can be stored before the task is executed (Listing 15). When the task complete, 'Taskover' is assigned a value of 255, a reserved ID to indicate successful task completion.

```
Taskover = Index                  //Store task ID
(*SCH_tasks_G[Index].pTask)();    //Run the task
Taskover = 255;                   //Task completed
```

Listing 15: Detection of task overrun in dispatch

Returning from update

If a task overrun has occurred then Update must alter its return address so that - instead of returning to the overrunning task - it returns to End_Task (see Figure 43).

Please note that the End_Task function is required because Update is an FIQ (Fast Interrupt Request) ISR which - in the ARM architecture used here - has a separate stack and hence a different set of frames: unlike Update, End_Task runs in User mode and therefore has access to the stack and frames used by the overrunning task. It is the job of End_Task to back-trace and rewind the function calls until the return address to Dispatch is located. The end task function then returns control to Dispatch. Update must determine how the return address is stored and check if the address lies outside the critical code labels, indicating that the task has not returned before setting the 'taskover' variable. The original Update return address is stored so that End_Task can determine where the task overruns. The Update return address is then replaced with the start address of End_Task.

Shutting down the task

Having detected a task overrun in Update and changed the return address, control is transferred to End_Task which must shut down the overrunning task.

End_Task determines whether the overrunning task was a leaf function or a function containing sub functions (with frames in the stack). The frame pointer (fp) register is compared with the saved fp register value in Dispatch: if these values are equal, this indicates that the overrunning task is a leaf function. If the values are different then the function calls are back traced (using the ATPCS standard) until the frame-stored fp register is equal to the fp register value saved in Dispatch. The current processor fp register is then made equal to the stored fp register.

End_Task is then able to check if any registers contents were stored on the stack when the task was called. The store instruction is found by looking at the contents of the address referred to by the frame pointer (which identifies the first line of code in the function). By subtracting 12 from this address, locating it and comparing the contents with the instruction number for a block transfer function, a store instruction can be identified.

The store block transfer instruction is then decoded to recover important settings (such as the names of registers which were stored on the stack and whether the stack is ascending or descending). The required bits of the store instruction are modified to convert it to a load instruction. The new instruction is then pointed to by the address in r13, which is loaded into the program counter. The microprocessor then runs this instruction from RAM: this initiates the return sequence to Dispatch with the saved registers restored. Note that all the important registers are stored before and after the task is called to add an extra level of security from register corruption.

If the overrunning task is a leaf function then the end task function simply returns where the r14 (link) register, which contains the return address of the dispatch task function, is loaded into the PC (program counter) register.

When a task is to be shutdown a flag is set in Update so that Dispatch will loop through the task array again: this avoids the possibility of a tick offset error.

Overall strengths and weaknesses

☺ Task guardians provides a way of detecting and handling task overrun in the system

☹ Addition of task guardians adds to the complexity of the code.

☹ Controlling the transfer of control from Update to End_Task requires care.

5. References

Albert, A. and R. Bosch GmbH (2004). Comparison of Event-Triggered and Time-Triggered Concepts with regard to Distributed Control Systems. Embedded World. Nurnberg**:** 235-252.

Alexander, C. (1979). The Timeless Way of Building, Oxford University Press.

Alexander, C., S. Ishikawa, et al. (1977). A Pattern Language, Oxford University Press.

Alexander, C., M. Silverstein, et al. (1975). The Oregon Experiment, Oxford University Press.

Allworth, S. T. (1981). Introduction to Real-Time Software Design, Macmillan.

Audsley, N. C., A. Burns, et al. (1991). Hard Real-time Scheduling: The Deadline Monotonic Approach. Eight IEEE Workshop on Real-time operating Systems and Softwares., Atlanta, USA.

Ayavoo, D. (2006). The Development of reliable X-by-Wire Systems: Assessing The Effectiveness of a Simulation First Approach. Department of Engineering, University of Leicester. **PhD Thesis**.

Ayavoo, D., M. J. Pont, et al. (2005). A Hardware-in-the-Loop' testbed representing the operation of a cruise-control system in a passenger car. Proceedings of the Second UK Embedded Forum. Birmingham, UK, University of Newcastle upon Tyne.

Baker, T. P. and A. Shaw (1988). "The Cyclic Executive Model and Ada." Real-Time Systems Springer Netherlands **1**(1): 7-25.

Bate, I. J. (1998). Scheduling and timing analysis for safety critical real-time systems. Department of Computer Science, University of York. **PhD Thesis**.

Bate, I. J. (2000). Introduction to Scheduling and Timing Analysis, The use of Ada in Real-Time System. IEEE.

Burns, A. and A. J. Wellings (1994). "HRT-HOOD: a structured design method for hard real-time systems." Real-Time Systems **6**(6): 73-114.

Burns, A. and B. Wellings (1997). Real-Time Systems and Programming Languages, Addison Wesley.

Buttazzo C, G. (2003). Rate Monotonic vs EDF : Judgement Day. LNCS, Springer. **2855/2003:** 67-83.

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

Buttazzo, G. C. (2005). "Rate Monotonic vs EDF : Judgement Day." Real-Time Systems **29**(1): 5-26.

Cain, B. G., J. O. Coplien, et al. (1996). "Social patterns in productive software development organisations." Annals of Software Engineering **2**(1).

Cottet, F., J. Delacroix, et al. (2002). Scheduling in Real-Time Systems, John Wiley & Sons.

Cunningham, W. (1987). The CHECKS pattern language of Information Integrity, Addison Wesley.

Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley.

Gergeleit, M. and E. Nett (2002). Scheduling Transient Overload with the TAFT scheduler. Fall meeting of GI/ITG specialized group of operating systems.

Hanmer, R. (2007). Patterns for Fault Tolerant Software, John Wiley & Sons, Ltd.

Hanmer, S. R. and G. Stymfal (2000). "An Input and Output Pattern Language." Pattern Languages of Program Design **4**: 503-536.

Hughes, Z. M. and M. J. Pont (2004). Design and test of a task guardian for use in TTCS embedded systems. UK Embedded Forum, Published by University of Newcastle.

Jones, M. (1997). "What really happened on Mars?", from http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html.

Kalinsky, D. (2001). Context Switch. Embedded Systems Programming.

Kirner, R. and P. Puschner (2003). Discussion of Misconceptions about WCET Analysis. 3rd Euromicro International workshop on WCET Analysis.

Kopetz, H. (1991). Event-Triggered versus Time-Triggered Real-Time Systems. Workshop on Operating Systems of the 90s and Beyond

Kopetz, H. (1997). Real Time Systems Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers.

Kurian, S. and M. J. Pont (2005). Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples. Second UK Embedded Forum, Birmingham, UK, Published by University of Newcastle upon Tyne.

Labrosse, J. J. (2000). Embedded Systems Building Blocks.

Laplante, P. A. (2004). Real Time Systems Design and Analysis, IEEE Press.

Liu, C. L. and J. W. Layland (1973). "Scheduling Algorithms for Multiprogramming in a Hard-Real Time Environment." Journal of the ACM **20**(1): 46-61.

Liu, J. W. S. (2000). Real-Time Systems, Prentice Hall.

Locke, D. (1992). "Software Architectures for Hard Real-time Applications: Cyclic Executives vs Fixed Priority Executives." Real-Time Systems **4**(1): 37-53.

Locke, D. (2002). "Priority Inheritance: The Real Story." from http://www.linuxdevices.com/articles/AT5698775833.html.

Maaita, A. (2008). Techniques for enhancing the temporal predictability of real-time embedded systems employing a time-triggered software architecture. Department of Engineering, University of Leicester. **PhD Thesis**.

Maaita, A. and M. J. Pont (2005). Using 'planned pre-emption' to reduce levels of task Jitter in a time-triggered hybrid scheduler. Second UK Embedded Forum. Birmingham, UK**:** 18-35.

Martin, T. (2005). The insider's guide to the Philips ARM7 Based microcontrollers, Coventry, Hitex, UK, Ltd.

Nissanke, N. (1997). Real time Systems, Prentice Hall.

Phatrapornnant, T. and M. J. Pont (2006). "Reducing Jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling." IEEE Transactions on Computers **55**(2): 113-124.

Pont, M. J. (2001). Patterns for Time-Triggered Embedded Systems, ACM press.

Pont , M. J. (2002). Embedded C, Addison-Wesley.

Puschner, P. (2002b). Is WCET Analysis a non-problem? Towards new Software and Hardware architectures. 2nd International Workshop on Worst Case Execution Time Analysis, Vienna, Austria.

Puschner, P. (2003). The single-path approach towards WCET-analysable software. IEEE International Conference on Industrial Technology.

Ramachandran, B., K. Fujiwara, et al. (2006). Business process transformation patterns & the business process transformation wizard. Proceedings of the 38th conference on Winter Simulation, Monterey,California.

Renwick, K. (2004). "How to use priority inheritance." from http://www.embedded.com/showArticle.jhtml?articleID=20600062.

Rising, L., Ed. (2001). Design Patterns in Communication Software, Cambridge University Press.

Scarlett, J. J. and R. W. Brennan (2006). Re-evaluating Event-Triggered and Time-Triggered Systems. IEEE conference on Emerging technologies and factory automation**:** 655-661.

Scheler, F. and W. Schroder-Preikschat (2006). Time-triggered vs. event-triggered: A matter of configuration? MMB Workshop Proceedings (GI/ITG Workshop on Non-Functional Properties of Embedded Systems, Nuremberg, Berlin VDE Verlag.

Schlindwein, F. S., M. J. Smith, et al. (1988). "Spectral analysis of doppler signals an computations of the normalized first moment in real time using a digital siganl processor." Medical and Biological Engineering & Computing **26**: 228-232.

Sha, L., R. Rajkumar, et al. (1990). "Priority Inheritance Protocols: an approach to real-time synchronization " IEEE Transactions on Computers **39**(9): 1175-1185.

Simon, D. E. (2001). An Embedded Software Premier, Addison-Wesley/ACM Press.

Tindell, K. W., A. Burns, et al. (1994). "An Extendible Approach for Analyzing Fixed Priority Hard Real-time Tasks." Real-Time Systems **6**: 133-151.

UM10211 (2009). LPC 23XX User manual Rev. 03.

Ward, N. J. (1991). "The static analysis of safety-critical avionics control system" in Air Transport Safety. Safety and Reliability Spring Conference.

Yodaiken, V. (2002). "Against Priority Inheritance." from http://www.linuxdevices.com/articles/AT7168794919.html.