

Selecting an appropriate scheduler for use with time-triggered embedded systems

Michael J. Pont, Susan Kurian, Huiyan Wang and Teera Phatrapornnant

*Embedded Systems Laboratory, University of Leicester,
University Road, LEICESTER LE1 7RH, UK.*

M.Pont@le.ac.uk; sk183@le.ac.uk; hw79@le.ac.uk; tp35@le.ac.uk

<http://www.le.ac.uk/eg/embedded/>

Abstract

We have previously described a “language” consisting of more than seventy patterns. This language is intended to support the development of reliable embedded systems: the particular focus of the collection is on systems with a time triggered (TT) system architecture.

We have been assembling this collection for a decade. As our experience with the collection has grown, we have begun to add a number of new patterns and revised some of the existing ones. As we have worked with this collection, we have felt that there were ways in which the overall architecture could be improved in order to make the collection easier to use, and to reduce the impact of future changes.

The introduction to this paper describes the approach that we have taken in order to refactor and refine our original pattern collection. The core of the paper then goes on to describe one new pattern that has resulted from this process.

Acknowledgements

This work is supported by (i) an ORS award to Susan Kurian from the UK Government, Department for Education and Skills (ii) an DTA award to Huiyan Wang from the UK Government, Engineering and Physical Sciences Research Council, and (iii) University of Leicester.

This paper has taken some time to evolve. Early versions of the pattern in this paper were published in a technical report (Pont et al., 2005) and a paper presented at the 2nd UK Embedded Forum (Kurian and Pont, 2005). A later version of this paper was also discussed at EuroPLoP 2006 (but was not published in the final proceedings because we were not yet happy with it).

From “EuroPLoP 2006”, we thank Dietmar Schuetz, who provided numerous useful suggestions on earlier drafts of this paper during the shepherding (and workshop) process. We also thank Neil Harrison, Klaus Meffert and Hedin Meitel who provided useful comments at the workshop. (The problems which remained with this paper were entirely our fault.)

From “EuroPLoP 2007”, we thank Wolfgang Herzner, who asked us some difficult questions about this evolving paper and helped to transform it into the present version.

Copyright

Copyright © 2005-2007 by Michael J. Pont, Susan Kurian, Huiyan Wang and Teera Phatrapornnant. Permission is granted to copy this paper for purposes associated with EuroPLoP 2007.

Introduction

We have previously described a “language” consisting of more than seventy patterns, which will be referred to here as the “PTTES Collection” (see Pont, 2001). This language is intended to support the development of reliable embedded systems: the particular focus of the collection is on systems with a time triggered (TT) system architecture. Work began on these patterns in 1996, and they have since been used in a range of industrial systems, numerous university research projects, as well as in undergraduate and postgraduate teaching on many university courses (e.g. see Pont, 2003; Pont and Banner, 2004; Phatrapornnant and Pont, 2006; Mwelwa et al., 2007; Short and Pont, 2007).

As our experience with the collection has grown, we have begun to add a number of new patterns and revised some of the existing ones (e.g. see Pont and Ong, 2003; Pont *et al.*, 2004; Key *et al.*, 2004; Pont et al., in press). Inevitably, by definition, a pattern language consists of an inter-related set of elements: as a result, it is unlikely that it will ever be possible to refine or extend such a system without causing some side effects. However, as we have worked with this collection, we have felt that there were ways in which the overall architecture could be improved in order to make the collection easier to use, and to reduce the impact of future changes.

In this new structure, we use what we call “abstract patterns” to address common design decisions faced by developers of embedded systems. Often used as an “entry point” to the evolving collection, such patterns do not – directly – tell the user how to construct a piece of software or hardware: instead they are intended to help a developer decide whether use of a particular design solution (perhaps a hardware component, a software algorithm, or some combination of the two) would be an appropriate way of solving a particular design challenge. Note that the problem statements for these patterns typically begin with the phrase “Should you use a ...” (or something similar).

For example, in this paper, we present the abstract pattern TT SCHEDULER. This pattern describes what a TT scheduler is, and discusses situations when it would be appropriate to use such an architecture in a reliable embedded system. If you decide to use a TT architecture, then you have a number of different implementation options available: these different options have varying resource requirements and performance figures. The patterns TTC-SL SCHEDULER, TTC-ISR SCHEDULER and TTC SCHEDULER describe some of the ways in which a TTC SCHEDULER can be implemented. In each of these “full” patterns, we refer back to the abstract pattern for background information.

The remainder of this paper

The remainder of this paper presents the abstract pattern TT SCHEDULER.

References

- Key, S.A., Pont, M.J. and Edwards, S. (2004) "Implementing low-cost TTCS systems using assembly language". Proceedings of the Eighth European conference on Pattern Languages of Programs (EuroPLoP 2003), Germany, June 2003: pp.667-690. Published by Universitätsverlag Konstanz. ISBN 3-87940-788-6.
- Kurian, S. and Pont, M.J. (2005) "Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.36-59. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].
- Mwelwa, C., Athaide, K., Mearns, D., Pont, M.J. and Ward, D. (2007) "Rapid software development for reliable embedded systems using a pattern-based code generation tool". *SAE Transactions: Journal of Passenger Cars (Electronic and Electrical Systems)*, **115**(7): 795-803.
- Phatrapornnant, T. and Pont, M.J. (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: Building Reliable Applications with the 8051 Family of Microcontrollers", Addison-Wesley / ACM Press. ISBN: 0-201-331381.
- Pont, M.J. (2003) "Supporting the development of time-triggered co-operatively scheduled (TTCS) embedded software using design patterns", *Informatica*, **27**: 81-88.
- Pont, M.J. and Banner, M.P. (2004) "Designing embedded systems using patterns: A case study", *Journal of Systems and Software*, **71**(3): 201-213.
- Pont, M.J. and Ong, H.L.R. (2003) "Using watchdog timers to improve the reliability of TTCS embedded systems", in Hrubby, P. and Soressen, K. E. [Eds.] *Proceedings of the First Nordic Conference on Pattern Languages of Programs, September, 2002 ("VikingPloP 2002")*, pp.159-200. Published by Microsoft Business Solutions. ISBN: 87-7849-769-8.
- Pont, M.J., Kurian, S. and Bautista-Quintero, R. (in press) "Meeting real-time constraints using 'Sandwich Delays'". Paper presented at the 11th European Conference on Pattern Languages of Programs (EuroPLoP 11), Germany, July 2006.
- Pont, M.J., Kurian, S., Maaita, A. and Ong, R. (2005) "Restructuring a pattern language for reliable embedded systems", Embedded Systems Laboratory Technical Report ESL05/01.
- Pont, M.J., Norman, A.J., Mwelwa, C. and Edwards, T. (2004) "Prototyping time-triggered embedded systems using PC hardware". Proceedings of the Eighth European conference on Pattern Languages of Programs (EuroPLoP 2003), Germany, June 2003: pp.691-716. Published by Universitätsverlag Konstanz. ISBN 3-87940-788-6.
- Short, M.J. and Pont, M.J. (2007) "Fault-tolerant time-triggered communication using CAN", *IEEE Transactions on Industrial Informatics*, **3**(2): 131-142.

Context

- You are developing an embedded system.
- Your design is likely to employ a single processor.
- You are likely to employ a processor which has – compared with a desktop PC – significant resource constraints (e.g. limited memory, limited resource constraints).
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

Should you use a time-triggered (TT) scheduler as the basis of your embedded system (and, if so, which form of TT scheduler should you use)?

Background

This pattern is concerned with systems which have at their heart a TT scheduler. We will be concerned both with “time-triggered co-operative” (TTC) designs, “time-triggered rate-monotonic” (TTRM) designs and “time-triggered hybrid” (TTH) designs.

We provide some essential background material and definitions in this section.

What is a task?

Tasks are the building blocks of embedded systems. A task is simply a labelled segment of program code: in the systems we will be concerned with in this pattern, a task will generally be implemented using a function in the C programming language*.

Working with periodic tasks

Most embedded systems will be assembled from collections of tasks. We will be concerned with systems implemented using periodic tasks. In our case, such tasks will be implemented as functions which are called – for example – every millisecond or every 100 milliseconds during some or all of the time that the system is active.

Jitter

The term “jitter” is used to refer to variations in the interval between events. For example, suppose a periodic task is due to start at the following times (in ms):

{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, ...}

* A task implemented in this way does not need to be a “leaf” function: that is, a task may call (other) functions.

Suppose, instead, that it runs as follows:

{1.0, 2.1, 2.9, 4.0, 5.1, 6.2, ...}

The specified times have no jitter (the interval between tasks is 1.0 ms). By contrast, the interval between the observed times varies between a minimum of 0.8 ms and a maximum of 1.1 ms (a worst-case variation of 20% of the sample interval).

Jitter can have a serious impact on a range of applications in which a TT architecture can be employed. For example, Cottet and David (1999) show that – during data acquisition tasks – jitter rates of 10% or more can introduce errors which are so significant that any subsequent interpretation of the sampled signal may be rendered meaningless. Similarly Jerri (1977) discusses the detrimental impact of jitter on applications such as spectrum analysis and filtering. Hong (1995) and Stothert and Macleod (1998) have discussed the degradation in performance caused by jitter in control applications. In such systems, jitter can greatly degrade the performance by varying the sampling period (Torgren, 1998; Mart et al., 2001).

Scheduling tasks (overview)

For many projects, a key challenge is to work out how to schedule these tasks so as to meet all of the timing constraints (including jitter constraints).

The scheduler we use can take two forms: pre-emptive and co-operative (or “non-pre-emptive”). The difference between these two forms is - superficially – rather small but has very large implications for our discussions in this pattern.

To illustrate this distinction, suppose that – over a particular period of time – we wish to execute four tasks (Task A, Task B, Task C, Task D) as illustrated in Figure 1.

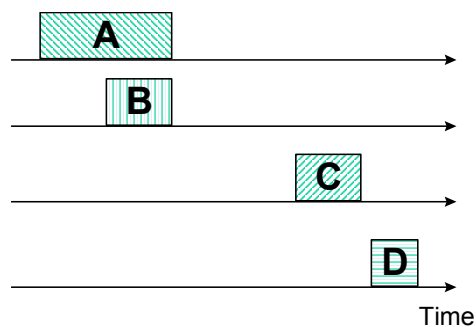


Figure 1: A schematic representation of four tasks (Task A, Task B, Task C, Task D) which we wish to schedule for execution in an embedded system with a single CPU.

We assume that we have a single processor. As a result, what we are attempting to achieve is shown in Figure 2.



Figure 2: Attempting the impossible: Task A and Task B are scheduled to run simultaneously.

In this case, we can run Task C and Task D as required. However, Task B is due to execute before Task A is complete. Since we cannot run more than one task on our single CPU, one of the tasks has to relinquish control of the CPU at this time.

In the simplest solution, we schedule Task A and Task B *co-operatively*. In these circumstances we (implicitly) assign a high priority to any task which is currently using the CPU: any other task must therefore wait until this task relinquishes control before it can execute. In this case, Task A will complete and then Task B will be executed (Figure 3).

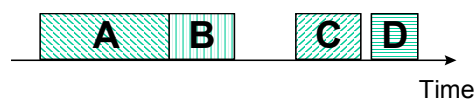


Figure 3: Scheduling Task A and Task B co-operatively.

Alternatively, we may choose a *pre-emptive* solution. For example, we may wish to assign a higher priority to Task B with the consequence that – when Task B is due to run – Task A will be interrupted, Task B will run, and Task A will then resume and complete (Figure 4).

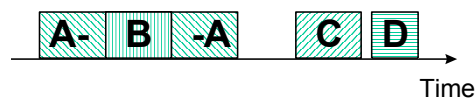


Figure 4: Assigning a high priority to Task B and scheduling the two tasks pre-emptively.

TTC architectures

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 (Phatrapornnant and Pont, 2006).

The type of TTC scheduler implementation discussed in this paper 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. The scheduler then places the processor into an “idle” (power saving) mode, where it will remain until the next tick.

Figure 5 shows an example of two tasks run with TTC scheduler with a tick interval of 1 ms. This “tick” is derived from a timer overflow: drift or jitter in this timing is, in large part, dependent on the associated computer hardware.

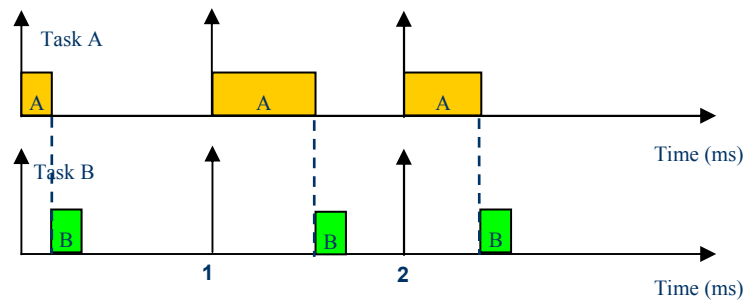


Figure 5: Illustrating the operation of a typical (interrupt-driven) TTC scheduler implementation.

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 *et al.*, 1993; 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 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).

The TTRM algorithm has been proved to be optimal amongst all fixed-priority algorithms (Liu and Layland, 1973): that is, Liu and Layland demonstrated that - if it is possible to schedule a task set using a fixed-priority algorithm and meet all of its timing constraints – then a TTRM algorithm can achieve this. Theoretically, every task can meet its deadline if the total CPU utilization is $\leq 69\%$ and: all tasks are periodic and independent of each other; the deadline of every task is equal to its period; the worst-case execution time of all tasks is known; and, context switching time can be ignored (Liu and Layland, 1973; Locke, 1992; Bate, 1998; Buttazzo, 2004).

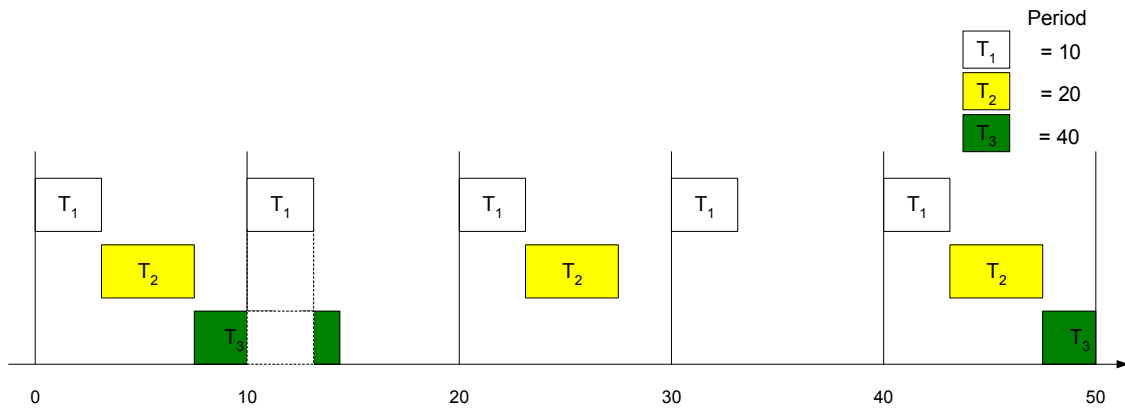


Figure 6: Structure of rate monotonic scheduling (adapted from Locke, 1992, Figure 3).

To illustrate the use of TTRM scheduling, Figure 6 shows how a set of periodic tasks can be scheduled by this algorithm. Task T_1 is executed periodically at the fastest rate, every 10 ms, and is determined to be the highest priority in this scheduling policy, while task T_2 and T_3 , 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 Figure 6, task T_3 - which is running - is pre-empted by task T_1 is at time 10: it carries on after the completion of task T_1 . Generally, the deadline of a task in TTRM scheduling is defined as the period: this assumption may have implications for task jitter levels (as we will discuss further in “Solution”).

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): see Figure 7.

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^{*†}. 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: see, for example, the work by Schlindwein et al. (1988).

* In the TTH architecture, the co-operative tasks all have the same priority (Priority C). The – single – pre-emptive task has Priority P. Priority P > Priority C.

† Please note that, in the TTH architecture, both the “co-operative” task and the (single) “pre-empting” task are periodic. This is in contrast to architectures investigated in some previous studies (e.g. Sandstrom et al., 1988) which have sought to integrate time-triggered task scheduling with the response to aperiodic (event related) interrupts.

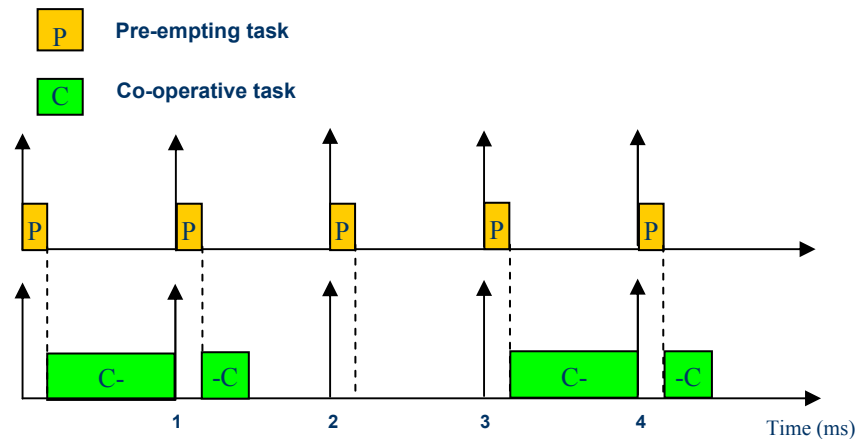


Figure 7: Illustrating the operation of a typical TTH scheduler implementation (adapted from Maaita and Pont, 2005, Figure 1).

Please note that it is not our intention to imply that a TTH architecture has – in terms of its scheduling behaviour – any particularly novel characteristics. Indeed, in many cases, a TTH architecture will be used with a very small number of tasks to implement a TTRM schedule. In addition, it should be emphasised that we support in this architecture only a single pre-empting task (since this is all we require). As a consequence, in terms of a theoretical scheduling analysis, this type of scheduler is of limited interest. However, in a resource-constrained embedded system, it is a very attractive proposition because it allows us to create a scheduler with minimal resource requirements which is precisely matched to the needs of many practical applications.

Solution

This pattern is intended to help answer the questions:

“Should you use a time-triggered (TT) scheduler as the basis of your embedded system (and, if so, which form of TT scheduler should you use)?”

In this section, we will explain how you can determine whether a TT architecture is a good choice for your application, and – for situations where such an architecture is appropriate – we will provide an overview of different scheduler options, to help you select the most appropriate solution.

Overall, our argument will be that – to maximise the reliability of your design – you should use the simplest “appropriate architecture”, and only employ the level of pre-emption that is essential to the needs of your application.

Should you use a TT architecture?

Some systems are “obvious” candidates for TT architectures. These systems include applications which involve data sampling or data playback, or other periodic activities (notably control algorithms).

Good uses for TT architectures include:

- Music players (for example, MP3 players) are required to play back music samples at a fixed – known – rate: the precise the rate will depend on the music quality: full “CD quality” sound will be played back at 44,400 samples per second. Any jitter in the playback times will result in a degradation in the music quality.
- 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 to put the system “to sleep” between samples.
- Control systems (for example, cruise control in your car, temperature control in your central heating or air conditioning system, control of the hard disk in your computer, control of the industrial robots in a local factory or the toy robots you buy for your children). Such systems all involve three core – periodic – activities: measuring some aspect of the system to be controlled (e.g. the room temperature), calculating changes required to the control system (e.g. calculating what new settings are required to your air conditioning system) and applying the changes to the control system (e.g. altering the settings on the air conditioning). In almost every case, a control system will be implemented using a TT architecture.

Of course, not every system is a good match for a TT architecture. In particular, if your system must only respond to aperiodic events, a TT architecture may not be appropriate. For example, a radio transmitter used to open your garage doors may be used only a few times a week. We could use a TT architecture to poll the switch on this system every 20 ms, just in case the switch has been pressed (see Listing 1). However, while such a solution would work, it would be likely to use more energy (and have shorter battery life) than a simple “event triggered” design (which might, for example, operate in power-down mode, except when the “reset switch” on the unit was pressed: see Listing 2).

```
int main(void)
{
    ...
    while(1)
    {
        Check_Switch();
        Control_RF_Transmitter();
        Delay_20ms();
    }

    // Should never reach here
    return 1
}
```

Listing 1: A very simple implementation of a time-triggered co-operative scheduler which is being used to control a radio-frequency transmitter. .

```

// System is reset every time switch is pressed
int main(void)
{
    Switch_On_RF_Transmitter();
    Delay_20ms();
    Switch_Off_RF_Transmitter();

    Enter_Power_Down_Mode();

    // Should never reach here
    while(1);
    return 1
}

```

Listing 2: A one-shot architecture for control of an RF transmitter.

In a similar way (but very different timescale), some forms of engine management designs require responses to events which are highly aperiodic. Such designs may not be a good match for TT architectures.

We will provide some further examples of systems which might best be implemented using a TT architecture in the remainder of this pattern. In considering these possible designs, our argument will be that - if a TT architecture is appropriate for your system - then to maximise the reliability and minimise resource requirements - you should use the simplest “appropriate architecture”, and only employ the level of pre-emption that is essential to the needs of your application.

When is it appropriate (and not appropriate) to use a pure TTC architecture?

Pure TTC architectures are a good match for a wide range of applications. For example, we have previously described in detail how these techniques can be in – for example - data acquisition systems, washing-machine control and monitoring of liquid flow rates (Pont, 2002), in various automotive applications (e.g. Ayavoo et al., 2004), a wireless (ECG) monitoring system (Phatrapornnant and Pont, 2006), and various control applications (e.g. Edwards et al., 2004; Key et al., 2004).

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 practice, it is sometimes assumed that a TTC architecture is inappropriate because some simple design options have been overlooked. We will use two examples to try and illustrate

how – with appropriate design choices – we can meet some of the challenges of TTC development.

Example: Multi-stage tasks

Suppose we wish to transfer data to a PC at a standard 9600 baud; that is, 9600 bits per second. 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. 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 (Figure 8).

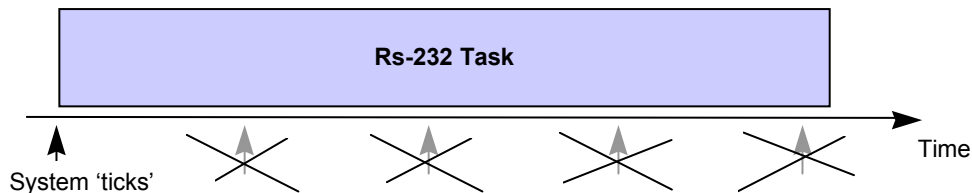


Figure 8: A schematic representation of the problems caused by sending a long character string on an embedded system with a simple operating system. In this case, sending the message takes 42 ms while the OS tick interval is 10 ms.

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.

A complete solution involves a change in the system architecture. Rather than sending all of the data at once, we store the data we want to send to the PC in a buffer (Figure 9). Every ten milliseconds (say) we check the buffer and send the next character (if there is one ready to send). In this way, all of the required 43 characters of data will be sent to the PC within 0.5 seconds. This is often (more than) adequate. However, if necessary, we can reduce this time by checking the buffer more frequently. Note that because we do not have to wait for each character to be sent, the process of sending data from the buffer will be very fast (typically a fraction of a millisecond).

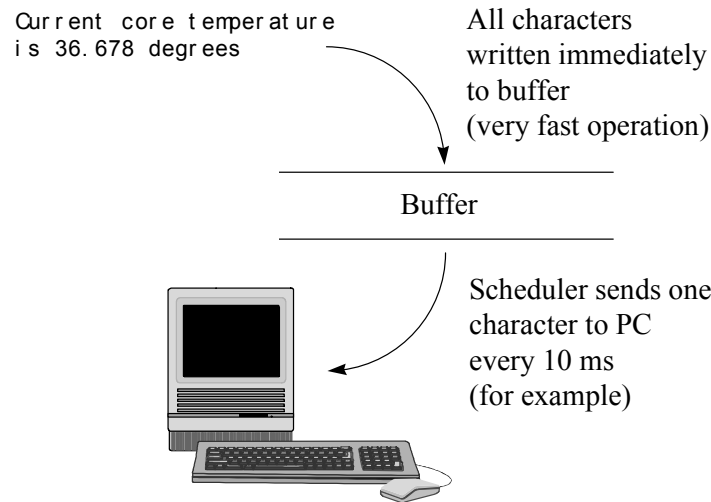


Figure 9: A schematic representation of the software architecture used in the RS-232 library.

This is an example of an effective solution to a widespread problem. The problem is discussed in more detail in the pattern MULTI-STAGE TASK [Pont, 2001].

Example: Rapid data acquisition

The previous example involved sending data to the outside world. To solve the design problem, we opted to send data at a rate of one character every millisecond. In many cases, this type of solution can be effective.

Consider another problem (again taken from a real design). This time suppose we need to receive data from an external source over a serial (RS-232) link. Further suppose that these data are to be transmitted as a packet, 100 ms long, at a rate of 115,200 baud and that one such packet will be sent every second for processing by our embedded system.

At this baud rate, data bytes will arrive approximately every 87 μ s. To avoid losing data, we would – if we used the architecture outlined in the previous example – need to have a system tick interval of around 40 μ s. This is a short tick interval, and would only produce a practical TTC architecture if a powerful processor was used.

However, a pure TTC architecture may still be possible, as follows. First, we set up an interrupt service routine (ISR), set to trigger on receipt of UART interrupts:

```
void UART_ISR(void)
{
    // Get first char

    // Collect data for 100 ms (with timeout)
}
```

These interrupts will be received roughly once per second, and the ISR will run for 100 ms. When the ISR ends, processing continues in the main loop:

```

void main(void)
{
    ...

    while(1)
    {
        Process_UART_Data();
        Go_To_Sleep();
    }
}

```

Here we have up to 0.9 seconds to process the UART data, before the next tick.

Pros and cons of TTRM

If a TTC architecture is not appropriate for your application, then a 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 utilization of the task set: as a result - when new functionalities are added to the system – it is only necessary to recalculate the new utilization 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.*

Of greater concern in this pattern is that RM scheduling seems likely to have more jitter than TTC scheduling, because the pre-emption from higher priority tasks may interrupt or block the lower priority tasks. These interferences may delay the release time of tasks, or interrupt running tasks and then prolong the output of a process residing at the end of a task: this may which result in jitter (Buttazzo, 2004). For example, in Figure 6, the output jitter can take place when task T_3 is pre-empted by task T_1 .

* It has been argued that another popular pre-emptive scheduler (“Earliest Deadline First”, EDF) has a lower runtime overhead than RM approaches (Buttazzo, 2005). Even though EDF always needs to update task deadlines this increased load may be offset by a reduction in the number of preemptions that occur under EDF (with a consequent reduction in context-switching time). Overall, Buttazzo (2005) suggests that the real advantage of TTRM scheduling is its simpler implementation. We would argue that TTRM also has (compared with a dynamic scheduling algorithm like EDF) more predictable behaviour and lower levels of task jitter. We say a little more about EDF in the “Related patterns” section of this pattern.

Overall, in the type of “low jitter” application with which this pattern is concerned, use of an RM algorithm presents two main challenges.

The first challenge is that the RM algorithm is based on the assumption that task deadlines are equal to periods: this means that use of RM guarantees only that a given task will complete its execution before it is due to run again. For short tasks, this means that jitter rates may be in the region of 90% (of the sample period), and the schedule will still be “correct”. In many cases, however, even jitter levels of 10% (of the sampling period) can render sampled data meaningless. Note that use of high task priorities will tend to reduce jitter levels: however – even if the tasks are wholly independent - the only safe assumption is that the highest-priority task will be guaranteed to demonstrate very low jitter levels (Locke, 1992).

The second challenge is that tasks are unlikely to be independent and that more than one task may require access to a mutually-exclusive resource (e.g. serial port, ADC and etc.). Where such critical sections are accessed through semaphores, even the highest-priority task may be blocked by a lower priority task (a process known as priority inversion) and then experience jitter or delay (Buttazzo, 2005). The priority inversion problem can be “solved” by using appropriate protocols (e.g. Priority Inheritance Protocol or Priority Ceiling Protocol, developed by Sha et al., 1990), to control access shared resources: however, such techniques were developed to address problems of deadlock and their impact on jitter is not always easy to predict.

Don't forget the TTH option

Sometimes a TTC architecture cannot meet our needs, but a TTRM architecture may still be “overkill”. For example, consider a wireless electrocardiogram (ECG) system (Figure 10).

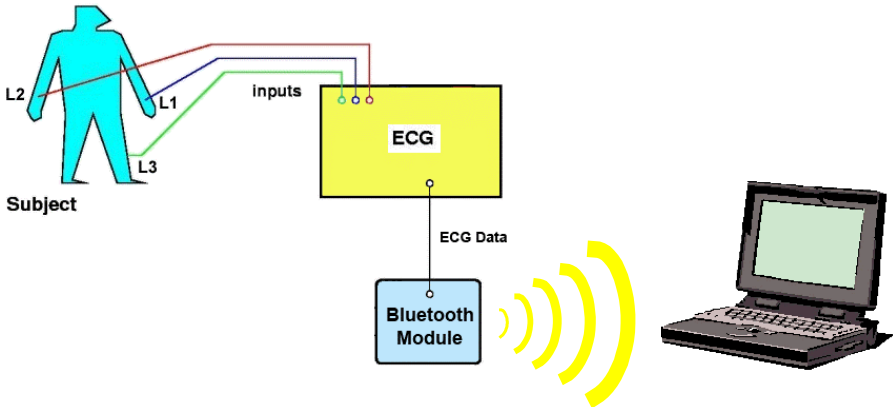


Figure 10: A schematic representation of a system for ECG monitoring. See Phatrapornnant and Pont (2006) for details.

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 a time-triggered hybrid architecture (Figure 11).

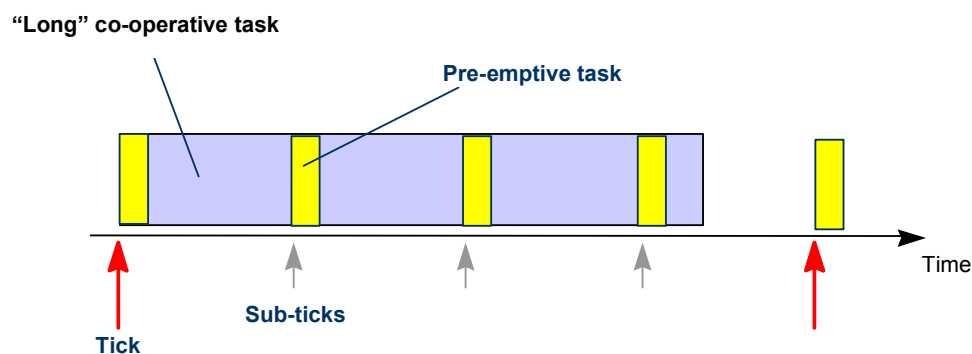


Figure 11: A “hybrid” software architecture. See text for details.

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 co-operatively based, but also provide a controlled degree of pre-emption.

Related patterns and alternative solutions

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: A very simple cyclic executive (time-triggered co-operative scheduler) which executes three periodic tasks, in sequence.

If we assume that the tasks executed in Listing 3 always have a duration of 4 ms, then – through the use of the Super Loop and delay functions, we have created a system which has a 10 ms “tick interval” (Figure 12).

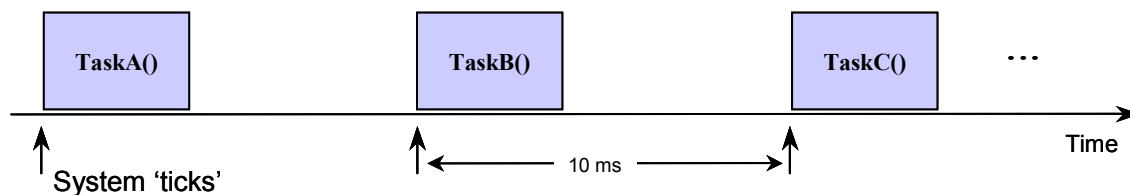


Figure 12: The task executions resulting from the code in Listing 3 (assuming all tasks are of duration 4 ms).

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-ISR Scheduler

The pattern “TTC-ISR SCHEDULER” describes another very simple software architecture for small embedded systems. Like a TTC-SL SCHEDULER, the TTC-ISR implementation this is a “hard wired” table-based scheduler. Unlike TTC-SL SCHEDULER, TTC-ISR SCHEDULER is suitable for use with systems which have hard timing constraints. The particular implementation discussed in this section is based on that described in detail elsewhere (see: Pont, 2002).

The basis of a TTC-ISR SCHEDULER is an interrupt service routine (ISR) linked to the overflow of a hardware timer. For example, see Figure 13. Here we assume that one of the microcontroller’s timers has been set to generate an interrupt once every 10 ms, and thereby call the function `Update()`. When not executing this interrupt service routine (ISR), the

system is “asleep”. The overall result is a system which - like that shown in Listing 3 – has a 10 ms “tick interval” in which three tasks are executed in sequence.

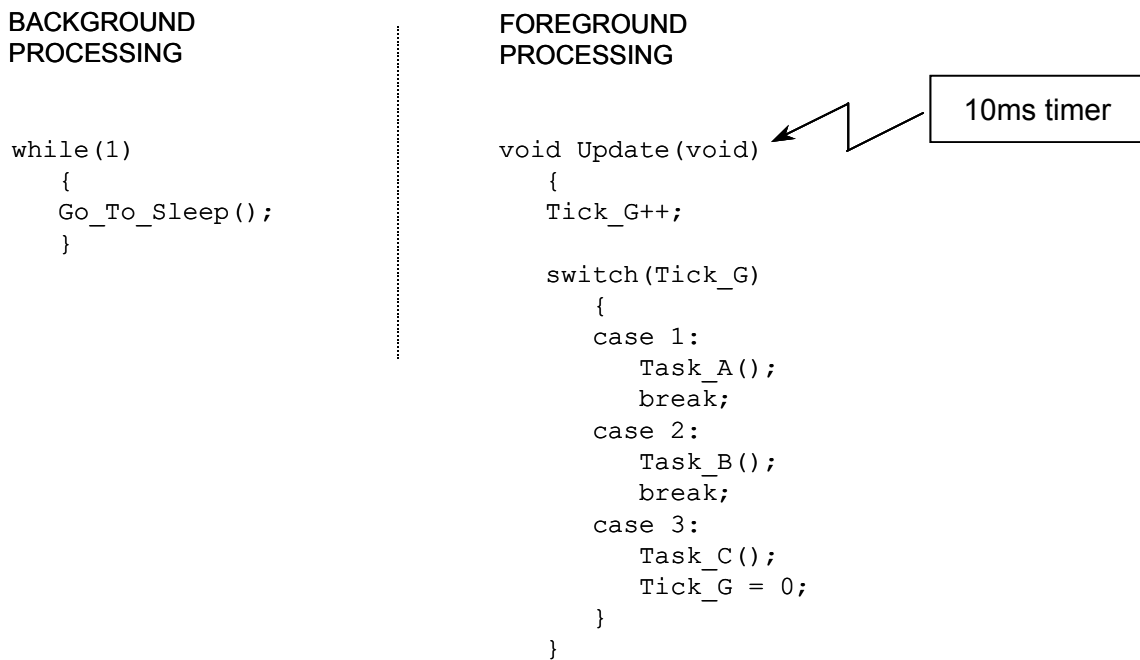


Figure 13: A schematic representation of a simple TTC-ISR Scheduler

Please note that “putting the processor to sleep” means moving it into a low-power (“idle”) mode. Most processors have such modes, and their use can – for example – greatly increase battery life in embedded designs. Use of idle modes is common but not essential.

Whether or not idle mode is used, the timing observed is largely independent of the software used but instead depends on the underlying timer hardware (which will usually mean the accuracy of the crystal oscillator driving the microcontroller). One consequence of this is that (for the system shown in Figure 13, for example), the successive function calls will take place at precisely-defined intervals (Figure 14), even if there are large variations in the duration of Update(). This is very useful behaviour, and is not easily obtained with architectures such as TTC-SL SCHEDULER.

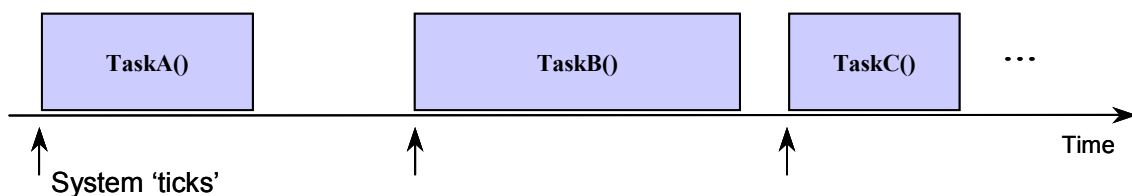


Figure 14: One advantage of the interrupt-driven approach is that the tasks will not normally suffer from “jitter” in their start times.

TTC Dispatch Scheduler

The implementation of a TTC-ISR SCHEDULER is highly system dependent. In addition, the implementation requires a significant amount of hand coding (to control the task timing), and there is no division between the “scheduler” code and the “application” code.

The TTC scheduler implementation referred to here as a “TTC-Dispatch” scheduler provides a more flexible alternative. The particular implementation discussed in this section is based on that described in detail elsewhere (see: Pont, 2001).

The TTC-Dispatch scheduler implementation considered in this section is characterised by distinct and well-defined scheduler functions (see Listing 4).

```
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 in ticks>, <period in 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: An overview of a possible TTC Scheduler implementation: see Pont (2001) for details.

In this paper, we summarise the operation of a TTC-Dispatch Scheduler which has been fully documented (Pont, 2001). We will refer to this implementation here as “TTC-2001”. Please note that this scheduler provides support for “one shot” tasks and dynamic scheduling: these features are not considered in this paper.

The TTC-2001 scheduler is driven by periodic interrupts generated from an on-chip timer. When an interrupt occurs, the processor executes an “Update” function (see Listing 5). In the Update function, the scheduler checks to see if any tasks are due to run and sets appropriate flags. After these checks are complete, a Dispatch function (Listing 6) will be called, and the identified tasks (if any) will be executed. When not executing the Update and Dispatch functions, the system will usually enter a low-power (“idle”) mode (see Pont, 2001 for further details).

```

void SCH_Update(void) interrupt INTERRUPT_Timer_6_Overflow
{
    tByte Index;

    // Clear T6 interrupt request flag
    T6IR = 0;

    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask)
        {
            if (--SCH_tasks_G[Index].Delay == 0)
            {
                // The task is due to run
                SCH_tasks_G[Index].RunMe += 1; // Incr. the 'Run Me' flag

                if (SCH_tasks_G[Index].Period)
                {
                    // Schedule rperiodic tasks to run again
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            }
        }
    }
}

```

Listing 5: “Update” ISR of TTC-2001 scheduler.

```

void SCH_Dispatch_Tasks(void)
{
    tByte Index;

    // Dispatches (runs) the next task (if one is ready)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        if (SCH_tasks_G[Index].RunMe > 0)
        {
            (*SCH_tasks_G[Index].pTask)(); // Run the task

            SCH_tasks_G[Index].RunMe -= 1; // Reset / reduce RunMe flag

            // Periodic tasks will automatically run again
            // - if this is a 'one shot' task, remove it from the array
            if (SCH_tasks_G[Index].Period == 0)
            {
                SCH_Delete_Task(Index);
            }
        }
    }
    // Report system status
    SCH_Report_Status();

    // The processor enters idle mode at this point
    SCH_Go_To_Sleep();
}

```

Listing 6: Dispatch function of TTC-2001 scheduler.

TTH Dispatch Scheduler

There are numerous ways in which a TTH scheduler can be implemented. One possible implementation of a “TTH Dispatch Scheduler” is described by Pont (2001).

Implementing a TTRM scheduler

If you are determined to implement a fully pre-emptive design, then Jean Labrosse (1999) and Anthony Massa (2003) discuss – in detail – the construction of such systems.

Alternative scheduling algorithms

We have considered a range of TT scheduling algorithms in this paper. There are, of course, various other alternatives. Briefly, these include techniques which schedule tasks:

- According to their deadline, with the “earliest deadline first” (EDF). For further details, see Liu and Layland (1973).
- According to their slack - or laxity – time, with the “least Laxity first” (LLF). For further details, see Chen (2002).
- According to their worst-case execution time: usually referred to as “shortest job first” (SJF) scheduling. See Stankovic and Ramamritham (1987) for further details.

We are not aware of patterns which describe how to implement these various schedulers.

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. Huiyan and Pont (this conference) describe a number of patterns which can help you to achieve this. See also SCOPED LOCKING in Buschmann et al. (2007).

Maximising reliability of pre-emptive designs

If using pre-emptive architectures, Jai Xu and David Parnas have worked for a number of years on what they call “pre-runtime scheduling”. This approach has the potential to improve the reliability of TT designs which employ pre-emption. For further details, please see: Xu (1993); Xu and Parnas (1990); Xu and Parnas (1993); Xu and Parnas (2000).

Multi-processor alternatives

Finally, we should note that all of the patterns in this paper assume the use of a single-processor solution. Various time-triggered architectures for multi-processor systems have also been described: see, for example, Kopetz (1997); Herzner et al. (2006); Pont (2001); Ayavoo et al. (2007); Short and Pont (2007).

Reliability and safety implications

For reasons discussed in detail in the previous sections of this pattern, time-triggered co-operative schedulers are generally considered to be a highly appropriate platform on which to construct a reliable (and safe) embedded system.

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.

Further reading

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

Audsley, N., Tindell, K. and Burns, A. (1993), *"The end of the line for static cyclic scheduling?"* Proceedings of the 5th Euromicro Workshop on Real-time Systems, Finland, pp. 36-41.

Ayavoo, D., Pont, M.J. and Parker, S. (2004) "Using simulation to support the design of distributed embedded control systems: A case study". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.

Ayavoo, D., Pont, M.J., Short, M. and Parker, S. (2007) "Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems", *Microprocessors and Microsystems*, **31**(5): 326-334.

Baker, T.P. and Shaw, A. (1989) "The cyclic executive model and Ada", *Real-Time Systems*, 1(1): 7-25.

Bate, I.J. (1998) "Scheduling and timing analysis for safety critical real-time systems", PhD thesis, University of York, UK.

Bate, I.J. (2000) "Introduction to scheduling and timing analysis", in *"The Use of Ada in Real-Time System"* (6 April, 2000). IEE Conference Publication 00/034.

Bennett, S. (1994) *"Real-Time Computer Control"* (Second Edition) Prentice-Hall.

Buschmann, F., Henney, K. and Schmidt, D.C. (2007) "Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing" (Volume 4). Wiley. ISBN: 978-0-470-05902-9

Buttazzo, G. C. (2004), *"Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications"*, 2nd ed, Springer.

Buttazzo, G. C. (2005), *"Rate monotonic vs. EDF: Judgement day"*, *Real-Time Systems*, Vol.29 pp. 5-26.

Cottet, F. and David, L. (1999) "A solution to the time jitter removal in deadline based scheduling of real-time applications", 5th IEEE Real-Time Technology and Applications Symposium - WIP, Vancouver, Canada, pp. 33-38.

Edwards, T., Pont, M.J., Scotson, P. and Crumpler, S. (2004) "A test-bed for evaluating and comparing designs for embedded control systems". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.

- Fohler, G. (1999) "Time Triggered vs. Event Triggered - Towards Predictably Flexible Real-Time Systems", Keynote Address, Brazilian Workshop on Real-Time Systems, May 1999.
- Hartwich F., Muller B., Fuhrer T., Hugel R., Bosh R. GmbH, (2002), Timing in the TTCAN Network, Proceedings 8th International CAN Conference.
- Herzner, W., Kubinger, W. and Gruber, M. (2006) "Triple-T - A system of patterns for reliable communication in hard real-time systems"; in D. Manolescu, M. Völter, J. Noble (eds): Pattern Languages of Program Design 5 (PLOPD5); pp.89-126; Software Engineering/Patterns Series, Addison-Wesley, Boston. ISBN 0-321-32194-4
- Hong, S.H. (1995) "Scheduling algorithm of data sampling times in the integrated communication and control systems". IEEE Transactions on Control Systems Technology, 3(2): 225-230
- Jerri, A.J. (1977) "The Shannon sampling theorem: its various extensions and applications a tutorial review", Proc. of the IEEE, vol. 65, n° 11, p. 1565-1596.
- Kalinsky, D., 2001. Context switch, Embedded Systems Programming, 14(1), 94-105.
- Key, S. and Pont, M.J. (2004) "Implementing PID control systems using resource-limited embedded processors". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.
- Kopetz, H. (1997) "Real-time systems: Design principles for distributed embedded applications", Kluwer Academic.
- Kurian, S. and Pont, M.J. (2007) "Maintenance and evolution of resource-constrained embedded systems created using design patterns", *Journal of Systems and Software*, **80**(1): 32-41.
- Labrosse, J. (1999) "MicroC/OS-II: The real-time kernel", CMP books. ISBN: 0-87930-543-6.
- Liu, C. L. and Layland, J. W. (1973), "Scheduling algorithms for multi-programming in a hard real-time environment", *Journal of the ACM*, **20**(1): 40-61.
- Locke, C.D. (1992) "Software architecture for hard real-time systems: Cyclic executives vs. Fixed priority executives", *The Journal of Real-Time Systems*, 4: 37-53.
- Maaita, A. and Pont, M.J. (2005) "Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid scheduler". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.18-35. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].
- Mart, P., Fuertes, J. M., Villi, R. and Fohler, G. (2001), "On Real-Time Control Tasks Schedulability", European Control Conference (ECC01), Porto, Portugal, pp. 2227-2232.
- Massa, A.J. (2003) "Embedded Software Development with eCOS", Prentice Hall. ISBN: 0-13-035473-2.

- Phatrapornnant, T. and Pont, M.J. (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: Building Reliable Applications with the 8051 Family of Microcontrollers", Addison-Wesley / ACM Press. ISBN: 0-201-331381.
- Pont, M.J. (2002) "Embedded C", Addison-Wesley. ISBN: 0-201-79523-X.
- Proctor, F. M. and Shackleford, W. P. (2001), "Real-time Operating System Timing Jitter and its Impact on Motor Control", proceedings of the 2001 SPIE Conference on Sensors and Controls for Intelligent Manufacturing II, Vol. 4563-02.
- Schindwein, F.S.; Smith, M.J. and Evans, D.H. (1988) "Spectral analysis of Doppler signals and computation of the normalized first moment in real time using a digital signal processor", *Medical & Biological Engineering & Computing*, **26**, pp. 228-232.
- Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990), "*Priority inheritance protocols: an approach to real-time synchronization*", *IEEE Transactions on Computers*, **39**(9): 1175-1185.
- Shaw, A.C. (2001) "Real-time systems and software" John Wiley, New York. [ISBN 0-471-35490-2]
- Short, M.J. and Pont, M.J. (2007) "Fault-tolerant time-triggered communication using CAN", *IEEE Transactions on Industrial Informatics*, **3**(2): 131-142.
- Stankovic, J. and Ramamritham, "The design of the spring kernel," *Proc. of the IEEE real-Time Systems Symposium, 1987*, pp. 146-157
- Stothert A. and Macleod I.M. (1998) "Effect of timing jitter on distributed computer control system performance". Proceedings of 15th IFAC Workshop DCCS'98-Distributed Computer Control Systems, Villa Olmo: Pergamon Press, 1998, pp.25-30.
- Torngren, M. (1998) "Fundamentals of implementing real-time control applications in distributed computer systems", *Real-Time Systems*, vol.14, pp.219-250.
- Ward, N. J. (1991) "The static analysis of a safety-critical avionics control system", in Corbyn, D.E. and Bray, N. P. (Eds.) "*Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference, 1991*" Published by SaRS, Ltd.
- Xu , J. (1993) "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, **19**(2), pp. 139-154.
- Xu , J. and Parnas, D.L. (1990) "Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations," *IEEE Transactions on Software Engineering*, **16**(3), pp. 360-369.
- Xu , J. and Parnas, D.L. (1993) "On Satisfying Timing Constraints in Hard-Real-Time Systems," *IEEE Transactions on Software Engineering*, **19**(1), pp. 70-84.
- Xu , J. and Parnas, D.L. (2000) "Priority Scheduling Versus Pre-Run-Time Scheduling," *International Journal of Time-Critical Systems*, **18**, 7-23, Kluwer Academic Publishers.