

Meeting real-time constraints using “Sandwich Delays”

Michael J. Pont, Susan Kurian and Ricardo Bautista

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

M.Pont@le.ac.uk; sk183@le.ac.uk; rb169@le.ac.uk

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

Abstract

This short paper is concerned with the use of patterns to support the development of software for reliable, resource-constrained, embedded systems. The paper introduces one new pattern (SANDWICH DELAY) and describes one possible implementation of this pattern for use with a popular family of ARM-based microcontrollers.

Introduction

In this paper, we are concerned with the development of software for a class of embedded systems in which there are two (sometimes conflicting) constraints. First, we wish to implement the design on a microcontroller with a cost of around \$1.00 (US). This type of platform might include an 8-bit or 16/32-bit microcontroller with very limited memory and CPU performance. Second, we wish to produce a system with extremely predictable behaviour. These behavioural constraints are of a hard real-time nature: for example, we typically require that the interval between the start times of periodic tasks does not vary by more than 1 μ s (preferably less).

To support the development of this type of software, we have previously described a “language” consisting of more than seventy patterns (e.g. see Pont, 2001). 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).

This brief paper describes one new pattern (SANDWICH DELAY) and illustrates – using what we call a “pattern implementation example” (e.g. see Kurian and Pont, 2005) one possible implementation of this pattern for use with a popular family of ARM-based microcontrollers.

Acknowledgements

Many thanks to Bob Hanmer, who provided numerous useful suggestions during the shepherding process.

Copyright

Copyright © 2006 by Michael J. Pont, Susan Kurian and Ricardo Bautista. Permission is granted to copy this paper for use in association with the EuroPLoP 2006 conference.

Context

- You are developing an embedded system.
- Available CPU and / or memory resources are – compared with typical desktop designs – rather limited.
- Predictable system behaviour is a key design requirement.

Problem

How can you ensure that a particular function or activity always takes the same period of time to execute?

Background

In order to make full use of this pattern, an understanding of the operation of timer hardware in a microcontroller is required. The pattern `HARDWARE DELAY` (Pont, 2001, p.194) provides relevant background information: alternatively, please refer to the data sheets for your chosen implementation platform.

Solution

To illustrate the need for a `SANDWICH DELAY`, we will consider a simple example.

Suppose that we have a system executing two functions periodically, as outlined in Listing 1.

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

Listing 1: Using a timer ISR to execute two periodic functions.

According to the code in Listing 1, function `Do_X()` will be executed every 10 ms. Similarly, function `Do_Y()` will be executed every 10 ms, after `Do_X()` completes. Figure 1 illustrates the sequence of function calls over time in a “tick graph”.

* WCET = Worst-Case Execution Time. If we run the task an infinite number of times and measure how long it takes to complete, the WCET will be the longest execution time which we measure.

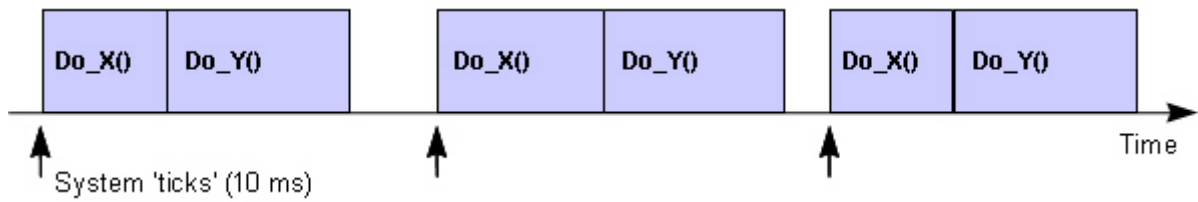


Figure 1: A tick graph showing the sequence of function calls in a simple system.

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 2, the jitter will be reflected in differences between the values of *ty1* and *ty2* (for example).

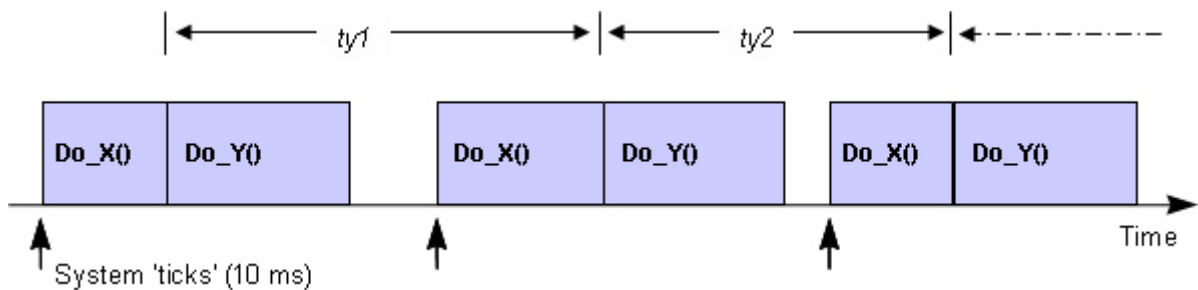


Figure 2: The impact of variations in the duration of `Do_X()` on the jitter in the start times of `Do_Y()`.

See text for details.

A SANDWICH DELAY can be used to solve this type of problem. More generally, 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 “wrap up” the activity you need to perform.

The operation and use of this technique is best illustrated using an example: please refer to Listing 2.

* In many embedded applications (such as those involving control or data acquisition) jitter in task / function start times can have serious implications. For example, Cottet and David 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 (Cottet and David, 1999). Similarly Jerri discusses the serious impact of jitter on applications such as spectrum analysis and filtering (Jerri, 1997). Also, in control systems, jitter can greatly degrade the performance by varying the sampling period (Torngren, 1998; Mart et al., 2001).

```

// ISR invoked by timer overflow every 10 ms
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 ms
  Wait_For_Sandwich_Timer_Overflow(); // Wait for timer to overflow
  // Execute Do_X() in a 'Sandwich Delay' - END

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

```

Listing 2: Employing a SANDWICH DELAY to reduce jitter in the start times of function Do_Y().

In Listing 2, we set a timer to overflow after 5 ms (a period slightly longer than the worst-case execution time 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 the 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 very little jitter.

Figure 3 shows the tick graph from this example, with the sandwich delay included.

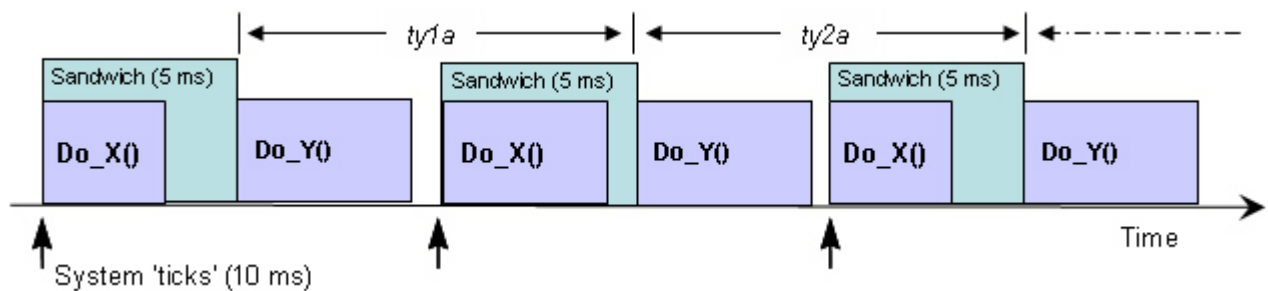


Figure 3: Reducing the impact of variations in the duration of `Do_X()` on the jitter in the start times of `Do_Y()` through use of a Sandwich Delay. See text for details.

Related patterns and alternative solutions

In some cases, you can avoid the use of a SANDWICH DELAY altogether, by altering the system tick interval. For example, if we look again at our `Do_X()` / `Do_Y()` example, the two tasks have the same duration. In this case, we would be better to reduce the tick interval to 5 ms and run the tasks in alternating time slots (Figure 4).

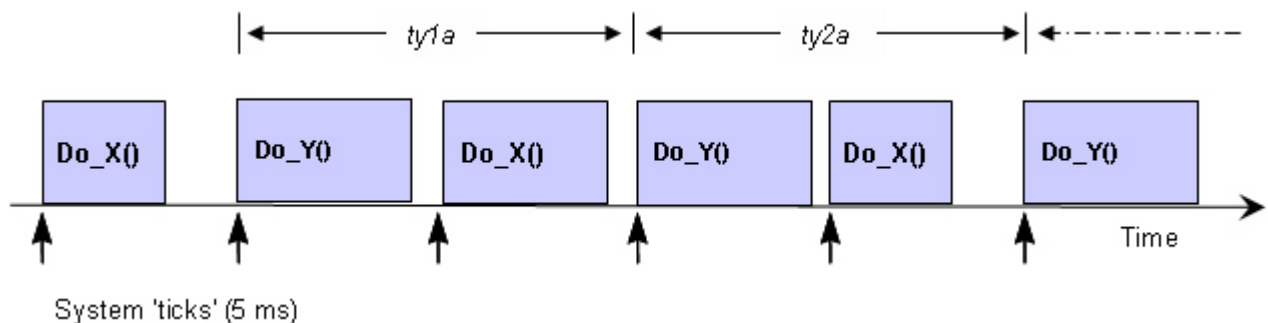


Figure 4: Avoiding the use of Sandwich Delays. See text for details.

Please note that this solution will only work (in general) if the tasks in your system have similar durations.

Reliability and safety implications

Use of a SANDWICH DELAY is generally straightforward, but there are three potential issues of which you should be aware.

First, you **need to know the duration (WCET) of the function(s) 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).

Second, you **must check the code** carefully, because the “wait” function may never terminate if the timer is incorrectly set up. In these circumstances a watchdog timer (e.g. see Pont, 2001; Pont and Ong, 2003) or a “task guardian” (see Hughes and Pont, 2004) may help to rescue your system, but relying on such mechanisms to deal with poor design or inadequate testing is – of course - never a good idea.

Third, 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 μ s 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 μ s are representative.

Example: Control system

Suppose that you need to call a sequence of three functions, as follows:

```
FunctionA();  
FunctionB();  
FunctionC();
```

If FunctionA() and FunctionB() have a variable execution time, then it will be difficult (if not impossible) to determine exactly how long after the start of FunctionA() that FunctionB() and FunctionC() will begin execution. This can be a problem in many applications with hard real-time constraints.

For example, in a control system, FunctionA() might be used to measure the current speed of a system, FunctionB() might be used to execute a control algorithm, and FunctionC() might be used to change an actuator, in order to maintain the speed within safe limits.

In this type of system, the tasks must always execute in the same order, and with precise timing. In particular, many control algorithms are designed on the assumption that the time between taking measurements of the current system state and changing the relevant actuators is fixed. Variations in this timing can have an impact on the performance (or even the stability) of the control system.

To deal with this type of problem, we can use a SANDWICH DELAY. An overview of the basic approach is shown in Listing 3.

```
void ISR(void)
{
    Start_Sandwich_Delay(A); // Set timer to match fn. duration
    FunctionA();
    Wait_For_Sandwich_Delay_To_Complete();

    Start_Sandwich_Delay(B);
    FunctionB();
    Wait_For_Sandwich_Delay_To_Complete();

    Start_Sandwich_Delay(C);
    FunctionC();
    Wait_For_Sandwich_Delay_To_Complete();
}
```

Listing 3: Reducing jitter in a control system using Sandwich Delays. See text for details.

In each case, the function calls are “wrapped” in a SANDWICH DELAY, created using a hardware timer.

Please note that the various SANDWICH DELAYS (and – for example – HARDWARE DELAYS) can often share a single timer, because, in this type of co-operative design, only one delay is active at a time.

Please note also that – where a timer is used to create “scheduler ticks” - it is not generally practical to use the same timer to create SANDWICH DELAYS (or HARDWARE DELAYS).

Example: Application of Dynamic Voltage Scaling

As we note in “Context”, we are concerned in this pattern with the development of software for embedded systems in which (i) the developer must adhere to severe resource constraints, and (ii) there is a need for highly predictable system behaviour. With many mobile designs (for example, mobile medical equipment) we also need to minimise power consumption in order to maximise battery life.

To meet all three constraints, it is sometimes possible to use a system architecture which combines time-triggered co-operative (TTC) task scheduling with a power-reduction technique known as “dynamic voltage scaling” (DVS). To achieve this, use of a SANDWICH DELAY is a crucial part of the implementation (and is used to ensure that the complex DVS operations do not introduce task jitter). The use of SANDWICH DELAYS in this context is described in detail by Phatrapornnant and Pont (2006).

Context

- You wish to implement a SANDWICH DELAY [this paper]
- Your chosen implementation language is C[†].
- Your chosen implementation platform is the Philips LPC2000 family of (ARM7-based) microcontrollers.

Problem

How can you implement a SANDWICH DELAY for the Philips LPC2000 family of microcontrollers?

Background

As with all widely-used microcontrollers, the LPC2000 devices have on-chip timers which are directly accessible by the programmer. More specifically, all members of this family have two 32-bit timers, known as Timer 0 and Timer 1. These can each be set to take actions (such as setting a flag) when a particular time period has elapsed.

In the simplest case, these timers (and other peripheral devices) will be driven by the “peripheral clock” (pclk) which - by default - runs at 25% of the rate of the system oscillator (Figure 5).

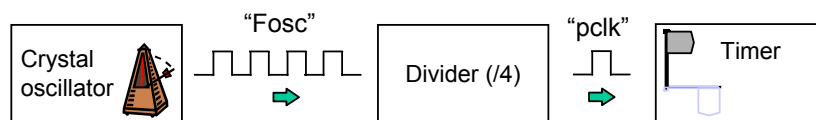


Figure 5: The link between oscillator frequency and timer updates in the LPC2000 devices (default situation).

By taking into account the link between the oscillator frequency and the timer hardware, the timers can be configured so that (for example) a flag is set after a period of 10ms has elapsed. The resulting delay code can be made highly portable.

* As the name might suggest, PIEs are intended to illustrate how a particular pattern can be implemented. This is important (in the embedded systems field) because there are great differences in system environments, caused by variations in the hardware platform (e.g. 8-bit, 16-bit, 32-bit, 64-bit), and programming language (e.g. assembly language, C, C++). The possible implementations are not sufficiently different to be classified as distinct patterns: however, they do contain useful information. We say more about PIEs in another paper at EuroPLoP 2006 (see Kurian and Pont “Restructuring a pattern language which supports time-triggered co-operative software architectures in resource-constrained embedded systems”).

[†] The examples in the pattern were created using the GNU C compiler, hosted in a Keil uVision 3 IDE.

Both Timer 0 and Timer 1 are 32-bit timers, which are preceded by a 32-bit pre-scalar. The pre-scalar is in turn driven by the peripheral clock. This is an extremely flexible combination. As an example, suppose that we wished to generate the longest possible delay using Timer 0 on an LPC2100 device with a 12 MHz oscillator. The delay would be generated as follows:

- Both the pre-scalar and the timer itself begin with a count of 0.
- The pre-scalar would be set to trigger at its maximum value: this is $2^{32}-1$ (=4294967295). With a 12 MHz oscillator (and the default divider of 4), the pre-scalar would take approximately 1432 seconds to reach this value. It would then be reset, and begin counting again.
- When the pre-scalar reached 1432 seconds, Timer 0 would be incremented by 1. To reach its full count (4294967295) would take approximately 200,000 years.

Clearly, this length of delay will not be required in most applications! However, very precise delays (for example, an hour, a day – even a week) can be created using this flexible hardware.

As a more detailed example, suppose that we have a 12 MHz oscillator (again with default divider of 4) and we wish to generate a delay of 1 second. We can omit the prescalar, and simply set the match register on Timer 1 to count to to the required value (3,000,000 – 1).

We can achieve this using the code shown in Listing 4.

Solution

A code example illustrating the implementation of a SANDWICH DELAY for an LPC2000 device is given in Listing 5 and Listing 6.

```

// Prescale is 0 (in effect, prescaler not used)
T1PC = 0;

// Set the "Timer Counter Register" for this timer.
// In this register, Bit 0 is the "Counter Enable" bit.
// When 1, the Timer Counter and Prescale Counter are enabled for counting.
// When 0, the counters are disabled.
T1TCR &= ~0x01; // Stop the timer by clearing Bit 0

// There are three match registers (MR0, MR1, MR2) for each timer.
// The match register values are continuously compared to the Timer Counter value.
// When the two values are equal, actions can be triggered automatically (see below)
T1MR0 = 2999999; // Set the match register (MR0) to required value

// When the match register detects a match, we can choose to:
// Generate an interrupt (not used here),
// Reset the Timer Counter and / or
// Stop the timer.
// These actions are controlled by the settings in the MCR register.
// Here we set a flag on match (no interrupt), reset the count and stop the timer.
T1MCR = 0x07; // Set flag on match, reset count and stop timer

T1TCR |= 0x01; // Start the timer

// Wait for timer to reach count (at which point the IR flag will be set)
while ((T1IR & 0x0001) == 0)
{
;
}

// Reset the timer flag (by writing "1")
T1IR |= 0x01;

```

Listing 4: Configuring Timer1 in the LPC2000 family. See text for details.

```

/*-----*/
Main.C (v1.00)
-----
Simple "Sandwich Delay" demo for Philips LPC2000 devices.
/*-----*/

#include "main.h"
#include "system_init.h"
#include "led_flash.h"
#include "random_loop_delay.h"
#include "sandwich_delay_t1.h"

/*-----*/

int main (void)
/*-----*/
int main(void)
{
// Set up PLL, VPB divider, MAM and interrupt mapping
System_Init();

// Prepare to flash LED
LED_FLASH_Init();

// Prepare for "random" delays
RANDOM_LOOP_DELAY_Init();

while(1)
{
// Set up Timer 1 for 1-second sandwich delay
SANDWICH_DELAY_T1_Start(1000);

// Change the LED state (OFF to ON, or vice versa)
LED_FLASH_Change_State();

// "Random" delay
// (Represents function with variable execution time)
RANDOM_LOOP_DELAY_Wait();

// Wait for the timer to reach the required value
SANDWICH_DELAY_T1_Wait();
}

return 1;
}

/*-----*/
--- END OF FILE -----
/*-----*/

```

Listing 5: Implementing a SANDWICH DELAY for the LPC2000 family (main.c)

```

/*-----*/
sandwich_delay_t1.c (v1.00)
-----

"Sandwich delay" for the LPC2000 family using Timer 1.

/*-----*/
#include "main.h"
/*-----*/

SANDWICH_DELAY_T1_Start()

Parameter is - roughly - delay in milliseconds.

Uses T1 for delay (Timer 0 often used for scheduler)

/*-----*/
void SANDWICH_DELAY_T1_Start(const unsigned int DELAY_MS)
{
    T1PC = 0x00;    // Prescale is 0
    T1TCR &= ~0x01; // Stop timer

    // Set the match register (MR0) to required value
    T1MR0 = ((PCLK / 1000U) * DELAY_MS) - 1;

    // Set flag on match, reset count and stop timer
    T1MCR = 0x07;

    T1TCR |= 0x01; // Start timer
}

/*-----*/

SANDWICH_DELAY_T1_Wait()
Waits (indefinitely) for Sandwich Delay to complete.

/*-----*/

void SANDWICH_DELAY_T1_Wait(void)
{
    // Wait for timer to reach count
    while ((T1IR & 0x01) == 0)
    {
        ;
    }

    // Reset flag (by writing "1")
    T1IR |= 0x01;
}

/*-----*/
---- END OF FILE -----
/*-----*/

```

Listing 6: Implementing a SANDWICH DELAY for the LPC2000 family (example): file (sandwich_delay_t1.c)

References

- 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.
- Furber, S. (2000) "*ARM System-on-Chip Architecture*", Addison-Wesley.
- Jerri, A. J. (1997), "The Shannon sampling theorem: its various extensions and applications a tutorial review", Proc. of the IEEE, Vol. 65, pp. 1565-1596.
- Hughes, Z.H. and Pont, M.J. (2004) "Design and test of a task guardian for use in TTCS embedded systems". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.16-25. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].
- 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].
- Mart, P., Fuertes, J. M., Ramamritham, K. and Fohler, G. (2001), "Jitter Compensation for Real-Time Control Systems", 22nd IEEE Real-Time Systems Symposium (RTSS'01), London, England, pp. 39-48.
- 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 (Special Issue on Design and Test of Systems-On-a-Chip), Feb 2006.
- Philips (2004) "LPC2119 / 2129 / 2194 / 2292 / 2294 User Manual", Philips Semiconductors, 3 February, 2004.
- 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 Hruby, 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., 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.
- Torngren, M. (1998), "Fundamentals of implementing real-time control applications in distributed computer systems", *Real-Time Systems*, 14, pp. 219-250.