Designing embedded systems using patterns: A case study

Michael J. Pont[1] and Mark P. Banner

Control & Instrumentation Research Group
Department of Engineering
University of Leicester
University Road
LEICESTER
LE1 7RH
United Kingdom.

FINAL VERSION (ACCEPTED - JSS 02-104):
Journal of Systems and Software
23 August 2002

Document details

File name: MJP & MPB - Jnl Sys & Soft 2002 v08 - FINAL VERSION August 2002
8015 words

[1] To whom correspondence should be addressed. E-mail: M.Pont@le.ac.uk

Designing embedded systems using patterns: A case study

Keywords

Design patterns, Time-triggered, Co-operating scheduling, Embedded system, Microcontroller

Abstract

If software for embedded processors is based on a time-triggered architecture, using co-operative task scheduling, the resulting system can have very predictable behaviour. Such a system characteristic is highly desirable in many applications, including (but not restricted to) those with safety-related or safety-critical functions. In practice, a time-triggered, co-operatively-scheduled (TTCS) architecture is less widely employed than might be expected, not least because care must be taken during the design and implementation of such systems if the theoretically-predicted behaviour is to be obtained. In this paper, we argue that the use of appropriate 'design patterns' can greatly simplify the process of creating TTCS systems. We briefly explain the origins of design patterns. We then illustrate how an appropriate set of patterns can be used to facilitate the development of a non-trivial embedded system.

1. Introduction

The focus of this paper is on the development of software for time-triggered, co-operatively scheduled (TTCS) embedded systems. Many studies suggest that systems implemented using TTCS techniques have more predictable behaviour than those implemented using alternative architectures (such as those which are event-triggered and / or pre-emptively scheduled). Set against this is the fact that the creation of TTCS architectures requires careful design and implementation if the theoretically-predicted improvements in system reliability are to be realised in practice. To address this problem, we propose the use of appropriate 'design patterns'.

In this paper, we explain the origins of design patterns, and illustrate how such patterns may be used to develop TTCS systems. Our conclusions are presented in Section 7.

2. Background

Embedded systems are often designed and implemented as a collection of communicating tasks (e.g. Nissanke, 1997; Shaw, 2001). The various possible system architectures may then be characterised in terms of these tasks: for example, if the tasks are invoked by aperiodic events (typically implemented as hardware interrupts) the system may be described as 'event triggered' (Nissanke, 1997). Alternatively, if all the tasks are invoked periodically (say every 10 ms), under the control of a timer, then the system may be described as 'time triggered' (Kopetz, 1997). The nature of the tasks themselves is also significant. If the tasks, once invoked, can pre-empt (or interrupt) other tasks, then the system is said to be 'pre-emptive'; if tasks cannot be interrupted, the system is said to be co-operative.[2]

Various studies have demonstrated that, 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, p.53-54) 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"*. Also, in a recent presentation, Bate (2000) identified the following four advantages of co-operative scheduling, compared to pre-emptive alternatives: [1] The scheduler is simpler; [2] The overheads are reduced; [3] Testing is easier; [4] Certification authorities tend to support this form of scheduling.

Despite these observations, all of the authors cited above, and the vast majority of other workers in this area, focus on the use of pre-emptive schedulers. Indeed, this focus is so strong that many

---

2 Note that these categories are not exclusive. For example, a complex system may support both time-triggered and event-triggered tasks; some of these may be co-operative in nature while others are pre-emptive.

authors refer to co-operative schedulers as ‘non-preemptive schedulers’. At least part of the reason why pre-emptive approaches are more widely discussed is because of confusion over the options available. For example, Bennett (1994, p.205) states: *“If we consider the scheduling of time allocation on a single CPU there are two basic alternatives: [1] cyclic, [2] pre-emptive.”* In fact, contrary to Bennett’s assertion, cyclic scheduling is a simple, specialised, form of co-operative algorithm suitable for use in a restricted range of simple applications, in particular those where accurate timing is not a key requirement and limited memory and CPU resources are available: this approach is not representative of the broad range of co-operative scheduling applications which are available. Barnett is, however, not alone: other researchers make similar assumptions. For example, Locke (1992) - in a widely cited paper - suggests that *“traditionally, there have been two basic approaches to the overall design of application systems exhibiting hard real-time deadlines: the cyclic executive … and the fixed priority [pre-emptive] architecture.”* (p.37). Similarly, Cooling (1991, p.292-293) compares co-operative and pre-emptive scheduling approaches. Again, however, his discussion of co-operative schedulers is restricted to a consideration of the special case of cyclic scheduling: as a result, his conclusion that a pre-emptive approach is more effective is unsurprising.

In any discussion of this topic, economic factors must also be considered: in particular, it would be naïve to ignore the fact that the promotion and use of pre-emptive environments offer considerable commercial advantages for some companies. For example, a co-operative scheduler may be easily constructed, entirely in a high-level programming language, in around two hundred lines of ‘C’ code (Pont, 2001). The code is portable, easy to understand and is, in effect, freely available. By contrast, the increased complexity of even a comparatively simple pre-emptive environment results in a much larger code framework (Figure 1). The size and complexity of this code makes it unsuitable for ‘in house’ construction in most situations, and therefore provides the basis for a commercial ‘RTOS’ products to be sold, generally at high prices and often with expensive run-time royalties to be paid. The continued promotion and sale of such environments has, in turn, prompted further academic interest in this area. For example, according to Liu and Ha, (1995): *“[An] objective of reengineering is the adoption of commercial off-the-shelf and standard operating systems. Because they do not support cyclic scheduling, the adoption of these operating systems makes it necessary for us to abandon this traditional approach to scheduling.”*

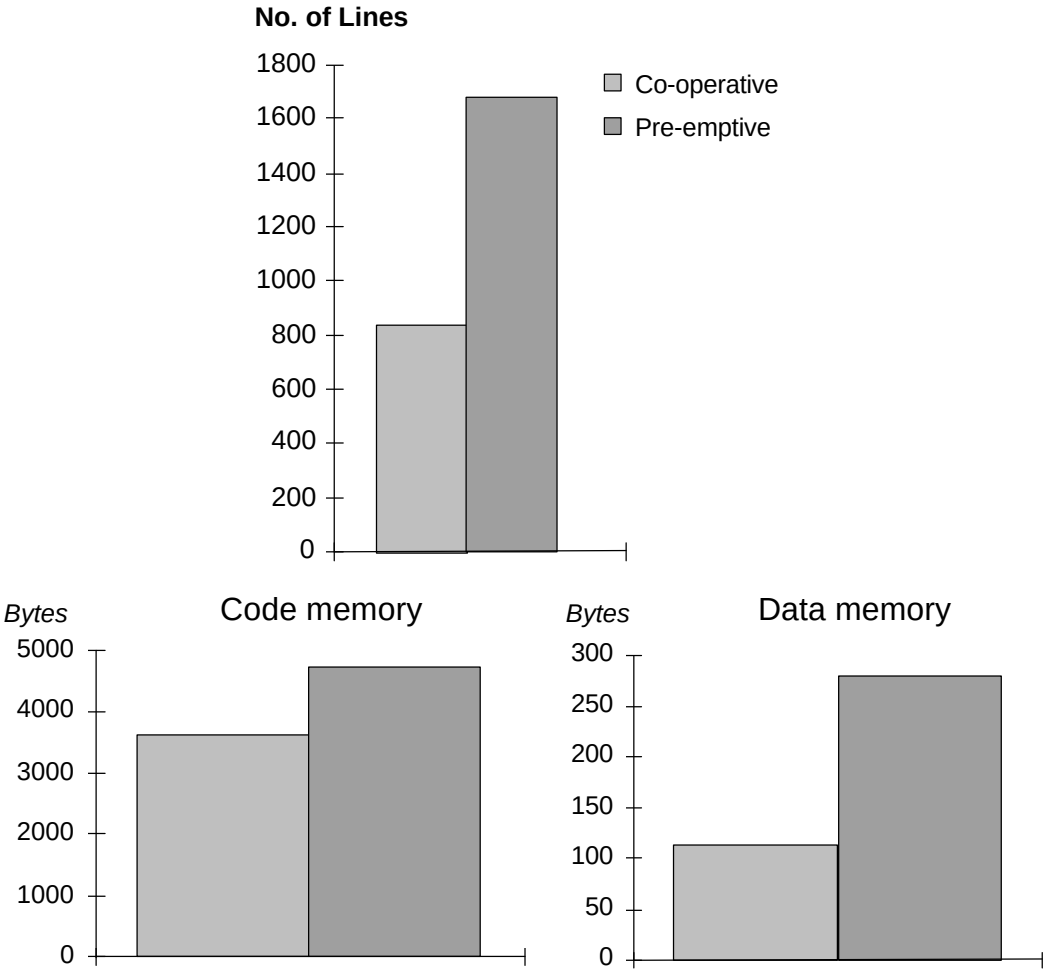

Figure 1: A comparison of some key features of co-operative and pre-emptive schedulers, when used in a embedded cruise-control application. The co-operative scheduler is fully described elsewhere (Pont, 2001). The simple pre-emptive has a similar architecture, but is modified to provide support for multiple pre-emptive tasks, and also has a simple 'locking' mechanism (to reduce conflicts over shared resources). Please note that the pre-emptive scheduler did not support task prioritisation: adding such features would add further to the complexity (and resource requirements) of this solution. Please note that both schedulers were implemented largely in C; however, the context-switch mechanism in the pre-emptive scheduler required the use of assembly language. To simplify the comparisons, the lines-of-code measures were made in assembly language, based on the outputs generated by the (Keil) compiler tools used in this study.

3. Design patterns

Where the different technical characteristics of pre-emptive and co-operative scheduling are compared equitably, the main concern expressed about co-operative approaches is often that long tasks will have an impact on the responsiveness of a co-operative scheduler. 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).

This concern is entirely legitimate, and any co-operative system that has been designed without due consideration of task durations is likely to prove extremely unreliable. However, there are a number of different techniques that may be employed in order to address this problem.

For example, there are some basic 'brute force' solutions:

- By using a faster processor, or a faster system oscillator, we can reduce the duration of 'long' tasks.
- By making use of an additional processor, we can obtain a true multi-tasking capability.

There are also a range of other alternatives:

- By using 'time out' mechanisms, we can ensure that tasks complete within their alloted time.
- By splitting up long tasks (triggered infrequently) into shorter 'multi-stage' tasks (triggered frequently), the processor activity can be more evenly distributed.
- By employing a 'hybrid' scheduler we can retain most of the desirable features of the (pure) co-operative scheduler, but still allow a single long (pre-emptible) task to be executed.

In the right circumstances, each of these ideas can prove useful. However, such observations do not, on their own, make it easier for developers to employ TTCS solutions. Instead, what is needed is a means of what we might call 'recycling design experience': specifically, we would like to find a way of allowing less experienced software engineers to incorporate successful solutions from previous TTCS design in their systems.

Recently, some developers have found that **software patterns** offer a way of achieving this. Current work on software patterns has been inspired by the work of Christopher Alexander and his colleagues (e.g. Alexander *et al.*, 1977; Alexander, 1979). Alexander is an architect who first described what he called 'a pattern language' relating various architectural problems (in buildings) to good design solutions. He defines patterns as "a three-part rule, which expresses a relation between a certain context, a problem, and a solution (Alexander, 1979, p.247).

For example, consider Alexander's **WINDOW PLACE** pattern, summarised briefly in Figure 2. This takes the form of a recognisable problem, linked to a corresponding solution. More specifically, like all good patterns, **WINDOW PLACE** does the following:

- It describes, clearly and concisely, a successful solution to a significant and well-defined problem.
- It describes the circumstances in which it is appropriate to apply this solution.
- It provides a rationale for this solution.
- It describes the consequences of applying the solution.
- It gives the solution a name.

This basic concept of descriptive problem-solution mappings was adopted by Ward Cunningham and Kent Beck who used some of Alexander's techniques as the basis for a small 'pattern language' intended to provide guidance to novice Smalltalk programmers (Cunningham and Beck, 1987). This work was subsequently built upon by Erich Gamma and colleagues who, in 1995, published an influential book on general-purpose object-oriented software patterns (Gamma *et al.*, 1995).

WINDOW PLACE

Context

WINDOW PLACE is an architectural pattern. It is most frequently applied in the design of hotels, offices or substantial houses.

Problem

You need to design a 'living room' in which people will congregate to sit and talk, drink tea, read newspapers, and so forth.

Solution

In developing a solution to this problem, Alexander *et al.* made the following observations:

- During the day, people generally dislike spending time in rooms without windows.
- When a room has windows, then people will be drawn to them. As a result, if the seating area is adjacent to the windows (preferably so that people can see out from their seats), then most people will tend to feel comfortable.
- By contrast, if the windows are on one side of the room and the seats on the other, most people will tend to feel uncomfortable.

Based on these observations (presented in greater detail in the original than is possible here), Alexander *et al.* proposed that, in solving this problem, architects should aim to create a well-lit 'window place', where people can sit comfortably adjacent to the window.

Figure 2: A summary of the **WINDOW PLACE** architectural pattern. Adapted from Alexander et al. (1977).

4. Patterns for time-triggered embedded systems

Since 1995 the development of pattern-based design techniques has become an important area of research in the software engineering community. Gradually, the focus has shifted from the use, assessment and refinement of individual patterns, to the creation of complete pattern languages, in areas including telecommunications systems (see, for example, Rising, 2001) and systems with hardware constraints (Noble and Weir, 2001).

In 1996, we began to assemble a collection of patterns to support the development of time-triggered software for embedded systems. The first version of these patterns were used 'in house', primarily

for teaching and training purposes. We then began to publish and discuss the next versions of the patterns more widely, not just at pattern workshops (see, for example, Pont, 2000) but also at more general technical conferences (see, for example, Pont, 1998; Pont *et al.*, 1999). Through this process we obtained a great deal of useful feedback on the project, and refined the collection again. The end result was the set of more than seventy patterns, which we refer to here as the 'PTTES collection' (see Pont, 2001).

The PTTES patterns are listed in Table 1.

255-TICK SCHEDULER	3-LEVEL PWM	A-A FILTER	ADC PRE-AMP
BJT DRIVER	CERAMIC OSCILLATOR	CO-OPERATIVE SCHEDULER	CRYSTAL OSCILLATOR
CURRENT SENSOR	DAC DRIVER	DAC OUTPUT	DAC SMOOTHER
DATA UNION	DOMINO TASK	EMR DRIVER	EXTENDED 8051
HARDWARE DELAY	HARDWARE PRM	HARDWARE PULSE-COUNT	HARDWARE PWM
HARDWARE TIMEOUT	HARDWARE WATCHDOG	HYBRID SCHEDULER	I2C PERIPHERAL
IC BUFFER	IC DRIVER	KEYPAD INTERFACE	LCD CHARACTER PANEL
LONG TASK	LOOP TIMEOUT	MOSFET DRIVER	MULTI-STAGE TASK
MULTI-STATE SWITCH	MULTI-STATE TASK	MX LED DISPLAY	NAKED LED
NAKED LOAD	OFF-CHIP CODE MEMORY	OFF-CHIP DATA MEMORY	ON-CHIP MEMORY
ONE-SHOT ADC	ONE-TASK SCHEDULER	ONE-YEAR SCHEDULER	ON-OFF SWITCH
PC LINK (RS232)	PID CONTROLLER	PORT HEADER	PORT I/O
PROJECT HEADER	PWM SMOOTHER	RC RESET	ROBUST RESET
SCC SCHEDULER	SCI SCHEDULER (DATA)	SCI SCHEDULER (TICK)	SCU SCHEDULER (LOCAL)
SCU SCHEDULER (RS-232)	SCU SCHEDULER (RS-485)	SEQUENTIAL ADC	SMALL 8051
SOFTWARE DELAY	SOFTWARE PRM	SOFTWARE PULSE-COUNT	SOFTWARE PWM
SPI PERIPHERAL	SSR DRIVER (AC)	SSR DRIVER (DC)	STABLE SCHEDULER
STANDARD 8051	SUPER LOOP	SWITCH INTERFACE (HARDWARE)	SWITCH INTERFACE (SOFTWARE)

Table 1: The 72 patterns we have assembled in order to support the development of embedded systems. See Pont (2001) for further details.

It is important to appreciate that all of the PTTES patterns are intended to support the development of software for systems using a TTCS architecture. For example, as we noted in Section 3, co-operatively-scheduled systems that are designed without due consideration being given to the task durations are likely to prove extremely unreliable. The following patterns address such issues:

- The processor patterns (**STANDARD 8051**, **SMALL 8051**, **EXTENDED 8051**) allow selection of a processor with performance levels appropriate for the application.
- The oscillator patterns (**CRYSTAL OSCILLATOR** and **CERAMIC RESONATOR**) allow an appropriate choice of oscillator type, and oscillator frequency to be made, taking into account system performance (and, hence, task duration), power-supply requirements, and other relevant factors.
- The various Shared-Clock schedulers (**SCC SCHEDULER**, **SCI SCHEDULER (DATA)**, **SCI SCHEDULER (TICK)**, **SCU SCHEDULER (LOCAL)**, **SCU SCHEDULER (RS-232)**, **SCU SCHEDULER (RS-485)**) describe how to schedule tasks on multiple processors, which still maintaining a time-triggered system architecture. Using one of these schedulers as a foundation, the pattern **LONG**

TASK describes how to migrate longer tasks onto another processor without compromising the basic time-triggered architecture.

- **LOOP TIMEOUT** and **HARDWARE TIMEOUT** describe the design of timeout mechanisms which may be used to ensure that tasks complete within their alloted time.
- **MULTI-STAGE TASK** discusses how to split up a long, infrequently-triggered task into a short task, which will be called more frequently. **PC LINK (RS232)** and **LCD CHARACTER PANEL** both implement this architecture.
- **HYBRID SCHEDULER** describes a scheduler that has most of the desirable features of the (pure) co-operative scheduler, but allows a single long (pre-emptible) task to be executed.

5. Example pattern

In this section, we give an example of one of the patterns from the PTTES collection. The pattern chosen is **LOOP TIMEOUT**. To meet the size constraints of this paper, the pattern has been edited: however, the key features of the pattern have been retained (see Figure 3).

As you examine this figure, please bear in mind three general points:

- It is sometimes assumed that a (software) pattern is simply a code library. It should be clear from **LOOP TIMEOUT** that this is not the case. Of course, some code is included: however, the pattern also includes a broad discussion of the problem area, a presentation of a solution, a discussion of the consequences of applying this solution, as well as suggestions about alternative approaches.
- A pattern should not simply describe something which the pattern author thinks is 'a good idea': instead, it should document a successful, tried-and-tested, design solution. **LOOP TIMEOUT** satisfies this condition, and describes a solution which will be recognised by many experienced developers. This might be taken to imply that patterns cannot describe 'novel' ideas, but this is not necessarily the case: for example, if a company has been successfully applying a solution X to problem Y, then - even if no other developer has ever identified the link between X and Y - the solution may well form the basis of a genuine and (for most developers) novel pattern.
- The process of 'mining' the patterns, and the process in which they are refined and enhanced (through structured workshop sessions) is an important part of the pattern development process. For example, as we noted in Section 4, the PTTES collection began life about four years ago. In this period, **LOOP TIMEOUT** has been discussed in a number of pattern sessions, and has been used in a large number of projects by many people. At the end of the day, the published version of **LOOP TIMEOUT** is a great improvement on the original version but it is certainly not perfect: like all patterns, it is best viewed as a 'work in progress'.

Loop TIMEOUT

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you ensure that you system will not 'hang' while waiting for a task (such as a switch read, an analog-to-data conversion, or serial data transfer) to complete?

Background

To understand the need for Loop Timeouts, consider an example.

The Philips 8Xc552 is an Extended 8051 device with an number of on-chip peripherals, including an 8-channel, 10-bit analog-to-digital converter (ADC). Philips provide an application note (AN93017) that describes how to use this microcontroller. This application note includes the following code:

```
// Wait until AD conversion finishes (checking ADCI)
while ((ADCON & ADCI) == 0);
```

Such code is not intended to be of 'production' quality. However, its structure is not unusual in embedded systems. The problem is that there are circumstances under which our application may 'hang'. This might occur for one or more of the following reasons:

- If the ADC has been incorrectly initialised, we cannot be sure that a data conversion will be carried out.
- If the ADC has been subjected to an excessive input voltage, then it may not operate at all.
- If the variable ADCON or ADCI were not correctly initialised, they may not operate as required.

Such problems are not, of course, unique to this particular microcontroller, or even to ADCs. Such code is common in embedded applications.

If your application is to be reliable, you need to be able to guarantee that no task or function will 'hang' in this way. Loop timeouts offer a simple but effective means of providing such a guarantee.

Solution

A loop timeout may be easily created. The basis of the code structure is a loop delay, created as follows:

```
unsigned integer Timeout_loop = 0;
...

while ((++Timeout_loop) > 0);
```

This loop will keep running until the variable `Timeout_loop` reaches its maximum value (assuming 16-bit integers) of 65535, and then overflows (to 0). When the variable overflows, the program will continue. Note that, without some simulation studies or prototyping, we cannot easily determine how long this delay will be. However, we do know that the loop will, eventually, 'time out'.

Such a loop is not terribly useful. However, if we consider again the ADC example given in 'Background', we can easily extend this idea. Recall that the original code was as follows:

```
// Wait until AD conversion finishes (checking ADCI)
while ((ADCON & ADCI) == 0);
```

Here is a modified version of this code, this time incorporating a loop timeout:

```
tWord Timeout_loop = 0;

// Take sample from ADC
// Wait until conversion finishes (checking ADCI)
// - simple loop timeout
while (((ADCON & ADCI) == 0) && (++Timeout_loop != 0));
```

We now know that the loop cannot go on 'for ever'.

Figure 3: The **LOOP TIMEOUT** software pattern (Part 1 of 2). Adapted from Pont (2001).

Note that we can vary the duration of the loop timeout by changing the initial value loaded into loop variable. The file `TimeoutL.H` (below) includes a set of constants that give - very approximately - the specified timeout values.

```
/*-------------------------------------------------------------------*-

   TimeoutL.H (v1.00)

   Simple loop timeout delays for the 8051 family based.

   * THESE VALUES ARE NOT PRECISE - YOU MUST ADAPT TO YOUR SYSTEM *

-*-------------------------------------------------------------------*/

// ------ Public constants -----------------------------------------

// Vary this value to change the loop duration
// THESE ARE APPROX VALUES FOR VARIOUS TIMEOUT DELAYS
// ON 8051, 12 MHz, 12 Osc / cycle

// *** MUST BE FINE TUNED FOR YOUR APPLICATION ***

// *** Timings vary with compiler optimisation settings ***

#define LOOP_TIMEOUT_INIT_001ms 65435
#define LOOP_TIMEOUT_INIT_010ms 64535
#define LOOP_TIMEOUT_INIT_500ms 14535

/*-------------------------------------------------------------------*/
```

Hardware resource implications

LOOP TIMEOUT does not use a timer and imposes an almost negligible CPU and memory load.

Reliability and safety implications

Using a **LOOP TIMEOUT** can result in a huge reliability and safety improvement at minimal cost. However, if practical, **HARDWARE TIMEOUT** (see Pont, 2001) is usually an even better solution.

Portability

Loop timeouts will work in any environment. However, the timings obtained will vary dramatically between microcontrollers and compilers.

Overall strengths and weaknesses

☺ **Much better than executing code without any form of timeout protection.**

☺ **Many applications use a timer for RS232 baud rate generation, and another timer to run the scheduler. In many 8051 devices, this leaves no further timers available to implement a** HARDWARE TIMEOUT **[see Pont, 2001]. In these circumstances, use of a loop is the only practical way of implementing effective timeout behaviour.**
Timings are difficult to calculate and timer values are not portable. **HARDWARE TIMEOUT** is always a better solution, if you have a spare timer available.

Related patterns and alternative solutions

As mentioned under Reliability and Safety Implications, **HARDWARE TIMEOUT** is often a better alternative to **LOOP TIMEOUT**.

In addition, **HARDWARE WATCHDOG** [see Pont, 2001] provides an alternative; however, it is rather crude by comparison, and detects errors at the application (rather than task) level.

Example: Test program for loop timeout code

[See Pont, 2001: Omitted here.]

Example: Loop timeouts in an I^2C library

[See Pont, 2001: Omitted here.]

Figure 3: The **LOOP TIMEOUT** software pattern (Part 2 of 2). Adapted from Pont (2001).

6. Designing with patterns

In order to illustrate how the PTTES patterns - like **LOOP TIMEOUT** - may be used to support the development of TTCS embedded systems, we will present a case study in this section. The focus of this study will be on a common 'white good' application: a washing machine.

6.1 Initial design

Software Engineering is, compared with other branches of the engineering profession, a very young discipline. In the limited time available, much work on software design has focused on the development and use of various graphical notations, including process-oriented notations, such as dataflow diagrams (Yourdon, 1989), and object-oriented notations, such as the 'Unified Modelling Language' (Fowler and Scott, 2000). The use of such notations is supported by 'methodologies': these are collections of 'recipes' for software design, detailing how and when particular notations should be used in a project (see, for example, Pont, 1996). The designs that result from the system of these techniques consist of a set of linked diagrams, each following a standard notation, and accompanied by appropriate supporting documentation (Yourdon, 1989; Booch, 1994; Pont, 1996; Fowler and Scott, 2000).

Although such notations have limitations, they offer a 'standard' way of recording system designs, and they need not be abandoned when a pattern-based design approach is adopted. Typically, any new design project will begin by sketching a high-level representation of the system to be developed: Figure 4 shows a standard 'context diagram' representation of the washing-machine product under discussion here (see Yourdon, 1989; Pont, 1996 for further details of this notation). For our purposes, the key value of the context diagram is that it provides a simple representation of the various interfaces that will be required between the processor at the heart of the system, and the various sensors and actuators in the machine itself.

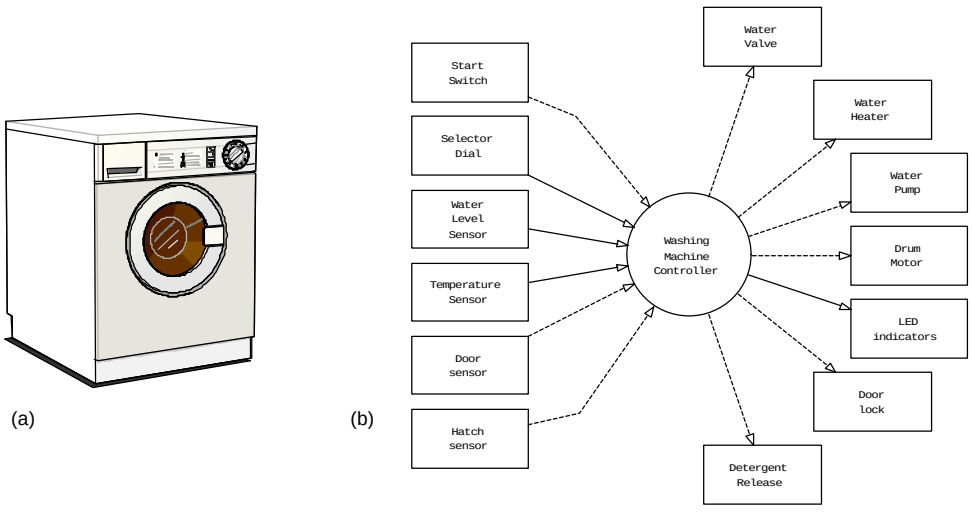

Figure 4: An example of part of the design for a washing machine, where (a) shows the finished product; (b) shows the 'Context diagram'. Please note that the design has been somewhat simplified for the purposes of this discussion.

Note that, in Figure 4, we have shown only a context diagram for this system. Some designers prefer to record more detail about the system requirements at this stage, using further diagrams and / or text. This is entirely appropriate, provided that what is generated is a 'logical' model of the system (specifying what the system is to do), rather than a 'physical model' (specifying how this

required behaviour is to be achieved): please refer to Pont (1996) for further discussion about the distinction between ‘logical’ and ‘physical’ system models.

In this case, here is a brief description of the way in which we expect the system to operate:

1. The user fills the machine with dirty laundry and locks the door.
2. The user fills the detergent tank with appropriate detergent.
3. The user selects a wash program (e.g. ‘Wool’, ‘Cotton’) on the selector dial.
4. The user presses the ‘Start’ switch.
5. The door lock is engaged.
6. The water valve is opened to allow water into the wash drum.
7. If the wash program involves detergent, the detergent hatch is opened. When the detergent has been released, the detergent hatch is closed.
8. When the ‘full water level’ is sensed, the water valve is closed.
9. If the wash program involves warm water, the water heater is switched on. When the water reaches the correct temperature, the water heater is switched off.
10. The washer motor is turned on to rotate the drum. The motor then goes through a series of movements, both forward and reverse (at various speeds) to wash the clothes. (The precise set of movements carried out depends on the wash program that the user has selected.) At the end of the wash cycle, the motor is stopped.
11. The pump is switched on to drain the drum. When the drum is empty, the pump is switched off.

The description is simplified for the purposes of this example, but it will be adequate for our purposes here.

6.2 Locating the appropriate pattern

Having developed an initial logical model of the system, we will want to identify patterns that can assist us in meeting the system requirements. To do this, we will clearly need to locate appropriate patterns.

In this case, we will simply assume that the developer has access to the PTTES collection. In this collection, the patterns are grouped into key areas as follows:

- Software foundations
  The patterns in this section describe how to create basic ‘Super Loop’ architectures, how to control the state of port pins, and how to create delay and ‘watchdog’ components.
- Time-triggered architectures for single-processor systems
  The patterns in this section describe how to create several complete scheduler architectures (both co-operative and hybrid), plus software components (such as **LOOP TIMEOUT***) suitable for use with such ‘operating systems’.*
- Time-triggered architectures for multi-processor systems
  The patterns in this section describe how to link multiple processors (using, for example, RS-485 or CAN), and schedule tasks on the resulting multi-tasking system.

- User-interface components
  The patterns in this section describe how to create switch and keypad interfaces. The also describe how to control both LED and liquid-crystal displays, and how to transfer data to and from desktop (or similar) PCs, using 'RS-232'.
- Serial-peripheral libraries
  Many useful components (such as analog-to-digital converters, temperature sensors and memory components) now have I^2C or SPI serial interfaces. The patterns in this section illustrate how to interface to such components.
- Monitoring and control components
  The monitoring and control patterns describe techniques that may be used to work with various monitoring- and control-related components (such as analog-to-digital converters, digital-to-analog converters, PWM outputs), and to perform a number of other related tasks (such as measuring rotational speed from pulse streams, and generating PID control algorithms).
- Hardware foundations
  The patterns in this section describe how to create basic oscillator and reset circuits, memory circuits, and circuits for the control of AC and DC loads.

Each pattern then has a 'Problem' section (describing, concisely, the problem it addresses). Each pattern also provides links to 'related patterns and alternative solutions': see, for example, **LOOP TIMEOUT**, in Section 5.

Overall, the pattern names, the pattern groupings, along with the 'Problem' and 'Related patterns' sections can help to direct users to the most appropriate patterns.

Of course, the patterns in the PTTES collection may not meet all of your requirements. One general solution will be to create pattern 'catalogues' that list, describe and link patterns from various sources: this is now beginning to happen (see, for example, Rising, 2001).

6.3 Assessing the interface requirements

In the case of the washing machine, starting with the context diagram (Figure 4), we can begin to examine the requirements for the various sensor and actuator components in more detail. Then, using the approach outlined in Section 6.2, we can identify patterns which will help us to construct these interfaces.

We will therefore consider each of these components in turn here.

6.3a LED indicators

We assume that four LEDs will be used in this system to indicate the washer status. We will therefore consider how these LEDs will be connected to the microcontroller.

The port pins on a typical microcontroller can be set at values of either 0V or 5V (or, in a 3V system, 0V and 3V) under software control. Each pin can typically sink (or source) a current of around 10 mA. With care, the port may be used to directly drive low-power DC loads, such as the LEDs we require in this interface (the pattern **NAKED LED** describes how to do this) or, for example, small

warning buzzers (see **NAKED LOAD**). However, while a limited number of such loads may be connected directly to the port, **NAKED LED** makes it clear that connecting four high-intensity LEDs will generally exceed the total port or microcontroller capacity. In these circumstances, use of an IC-based buffer circuit can be a cost-effective solution: see **IC BUFFER**. Indeed, even with small loads, the reliability of the application may be improved through the use of such a buffer.

Note that **NAKED LED** and **IC BUFFER** are concerned only with the hardware aspects of the LED Interface: however, the 'Related Patterns' section of these patterns emphasises the link to the pattern **PORT I/O**, where the relevant software issues are considered.

6.3b Start switch

We next consider the interface required for the 'Start' switch.

As noted at the start of this example, we are assuming that the user will fill the washing machine with laundry, add soap powder, select an appropriate wash programme. He or she will then press the 'Start' switch to begin the wash process.

From developers without experience in embedded systems, the design of a switch interface can seem rather trivial. However, issues such as switch bounce and the need to consider the impact of electrostatic discharge (ESD) can make the design of reliable switch interface rather more involved. There are therefore four different patterns in the PTTES collection to support the design of switch interfaces. Inspection of the various switch patterns will reveal that, of these, **SWITCH INTERFACE (HARDWARE)** will probably prove most appropriate in these circumstances.

6.3c Selector dial

We assume that the selector dial takes the form of a multi-position switch and that one of the switch patterns will be appropriate here (see Section 6.3b).

6.3d Water-level sensor

We assume that some form of piezoresistive pressure sensor will be used to determine the water level, and that the resulting (analog) signal will be most easily measured using some form of analog-to-digital converter (ADC).

There are five patterns in the collection that support the use of ADCs. Of these, **ONE-SHOT ADC** will address this problem most directly. The 'Related patterns' section of **ONE-SHOT ADC** emphasises the link to another potentially-useful pattern **ADC PRE-AMP**, which discusses the design of hardware to pre-process signals before analog-to-digital conversion.

6.3e Temperature sensor

A sensor will be required in order to determine the water temperature. The current pattern collection does not have a pattern directly associated with temperature sensing. However, it does include the patterns **I²C PERIPHERAL** and **SPI PERIPHERAL**. Together, these patterns support the use of a wide range of sensor (and other) components which employ of these two popular serial-interface standards.

6.3f Door and hatch sensors

We assume that the door sensor is required to ensure that the door has been correctly closed, before the wash-programme begins. Similarly, we assume that the hatch sensor is required to ensure that the that the detergent hatch has been fully closed.

We also assume that both of these sensors take the form of a magnetic switch and that one of the switch patterns (see Section 6.3b) will provide guidance on the creation of a suitable interface.

6.3g Water valve, water pump, door lock and hatch release

We assume that both the water valve and water pump will be solenoid-based units, requiring a DC drive voltage. We assume that the door lock and hatch release will be similar (albeit with lower current requirements).

In all cases, the current and voltage requirements of these devices will far exceed the very limited capability of most microcontroller port pins (see Section 6.3a): some form of driver circuit will therefore be required. Seven different patterns for controlling DC loads are presented in the collection: of these, **MOSFET DRIVER** will probably be the most appropriate for use here.

6.3h Water heater

We will assume that the water heater is mains powered (110-250V AC). We will further assume that this has a very simple on / off control system.

To control the heater, we have two main patterns to choose from. In this case, **SSR DRIVER (AC)**, which describes the use of solid-state relays, will probably be an appropriate choice.

6.3i Drum motor

We assume that we will need to have accurate control over the speed of the drum rotation. We also assume that the drum motor will be mains powered.

To control the speed of rotation, a review of the pattern collection would suggest the use of some form of pulse-width modulation interface: of the four PWM patterns included in the collection **HARDWARE PWM** would probably be most useful here.

To create the appropriate drive circuit, **SSR DRIVER (AC)**, would probably be appropriate.

6.4 Processor decisions

We are now in a position to be able to choose a suitable processor platform on which this system will be built.

The PTTES collection is most directly applicable to systems constructed using a member of the 8051 family of microcontrollers. The 8051's profile (price, performance, available memory) match the needs of many embedded systems very well. As a result, the 8051 device - originally developed by Intel - is now produced in more than 400 different forms by a diverse range of companies including Philips, Infineon, Atmel and Dallas. Sales of this vast family are estimated to have the largest share (around 60%) of the microcontroller market as a whole, and to make up more than

50% of the 8-bit microcontroller market. Versions of the 8051 are currently used in a long list of embedded products, from automotive systems to childrens' toys.

To support the selection of an appropriate 8051 device, the PTTES collection divides the family into three broad categories:

- **SMALL 8051**;
- **STANDARD 8051**;
- **EXTENDED 8051**.

In this case, to identify the most appropriate device, we need first to consider the system hardware requirements. These are determined from the discussions in Section 6.3 and summarised in Table 2.

	I/O pin requirements	Additional requirements
LED indicators	4	
Start switch	1	
Selector dial	6	
Water-level sensor	1	ADC
Temperature sensor	2	I^2C
Door sensor	1	
Hatch sensor	1	
Water valve	1	
Water pump	1	
Door lock	1	
Hatch release	1	
Water heater	1	
Drum motor	1	PWM
TOTAL	**22 pins**	**ADC, I^2C and PWM**

Table 2: A summary of the hardware requirements. Please see text for details.

From an examination of Table 2 and an examination of the three processor patterns, it is clear that the need for 22 port pins rules out the Small 8051 devices.

It is also clear that these requirements can be very easily met with an Extended 8051, without the need for any additional hardware. However, reading **EXTENDED 8051** makes it clear that these devices cost considerably more than a Standard 8051: in the white-good market, we wish to keep costs to a minimum.

By reviewing **STANDARD 8051**, we can see that a basic 8051 device (with 32 pins) would meet the need for 22 I/O pins, and provide a very cost-effective solution. Many 8051 devices have on-chip support for PWM, and (as **I^2C PERIPHERAL** makes clear) an I^2C interface can be created in software without imposing a significant processor load. Standard 8051s with on-chip ADCs are rare; however, by using the I^2C interface, an external ADC could be connected without difficulty and at low cost. Given the very limited data transfer requirements in this system, and the need for low system cost, this is probably the most appropriate solution.

6.5 Hardware foundation

All microcontroller-based designs require some form of reset circuit, and some form of oscillator. The patterns **ROBUST RESET** and **CRYSTAL OSCILLATOR** describe how to implement the required hardware foundation.

6.6 'Operating system'

As noted previously, all of the patterns in the PTTES collection are intended to support the development of applications with a time-triggered, co-operatively scheduled architecture.

At the heart of the collection is the pattern **CO-OPERATIVE SCHEDULER**. If hardware constraints permit, this allows up to 255 tasks to be scheduled, co-operatively, on either a 'on-shot' ("run this task after 10ms"), or periodic ("run this task every 2 ms") basis.

We will assume that this 'operating system' will be used in the washer system.

6.7 The software architecture

We can now consider the tasks that will be executed by the scheduler, and to consider the basic system architecture.

We might begin by trying to identify some of the functions that will be required to implement this system. A provisional list might be as shown in Figure 5.

- `Read_Selector_Dial()`
- `Read_Start_Switch()`
- `Read_Water_Level()`
- `Read_Water_Temperature()`
- `Control_Detergent_Hatch()`
- `Control_Door_Lock()`
- `Control_Motor()`
- `Control_Pump()`
- `Control_Water_Heater()`
- `Control_Water_Valve()`

Figure 5: A provisional list of functions that could be used to develop a washing-machine control system.

Now, suppose we wish to identify the **tasks** to be scheduled (co-operatively) in order to implement this system. Based on the above list, it may be tempting to conclude that each of the functions listed in Figure 5 should become a task in the system. While it would be possible to work in this way it would be likely to lead to a complex and cumbersome system implementation.

To see why this is so, take one example: the function `Control_Water_Heater()`. We want to heat the water only at particular times during the wash cycle. Therefore, if we want to treat this as a task and schedule it - say every 100 ms - we need to creation an implementation something like the code shown in Listing 1.

```
void TASK_Control_Water_Heater(void)
   {
   if (Switch_on_water_heater_G == 1)
      {
      Water_heater = ON;
```

```
      return;
      }

   // Switch off heater
   Water_pin = OFF;
   }
```

Listing 1: A possible task for controlling the water heater: for reasons discussed in the text, this approach is not recommended.

The task in Listing 1 checks a flag when it is called: if it is necessary to heat the water, the task switches on the water heater - otherwise, the water heater is switched off.

There are two problems with creating the program in this way:

- It is not at all clear which task should set the flag (`Switch_on_water_heater_G`), or the other similar flags that will be required in the other tasks in this system.
- We are going to end up with large numbers of tasks (very large numbers in a more substantial system), most of which - like this task - actually do very little. For systems without external memory this can create problems, because each task will consume some of the limited memory resources. In addition, this large collection of functions will be cumbersome and difficult to maintain.

These problems stem from the fact that we are confusing the role of "functions" and "tasks" in the system design. In practice, what we require in this and many similar systems is a single 'System Update' task: this, is a task that will be frequently scheduled and will, where necessary, call functions - like `Control_Water_Heater()` when required. This architecture is described fully in the **MULTI-STATE TASK** pattern.

In the washing machine, this system update task may look something like the code in Listing 2.

```
void WASHER_Update(void)
   {
   switch (System_state_G)
      {
      case START:
         {
         // Lock the door
         WASHER_Control_Door_Lock(ON);

         // Start filling the drum
         WASHER_Control_Water_Valve(ON);

         // Release the detergent (if any)
         if (Detergent_G[Program_G] == 1)
            {
            WASHER_Control_Detergent_Hatch(ON);
            }

         // Ready to go to next state
         System_state_G = FILL_DRUM;
         Time_in_state_G = 0;

         break;
         }

      case FILL_DRUM:
         {
```

```
      // Remain in this state until drum is full
      // NOTE: Timeout facility included here
      if (++Time_in_state_G >= MAX_FILL_DURATION)
         {
         // Should have filled the drum by now...
         System_state_G = ERROR;
         }

      // Check the water level
      if (WASHER_Read_Water_Level() == 1)
         {
         // Drum is full

         // Does the program require hot water?
         if (Hot_Water_G[Program_G] == 1)
            {
            WASHER_Control_Water_Heater(ON);

            // Ready to go to next state
            System_state_G = HEAT_WATER;
            Time_in_state_G = 0;
            }
         else
            {
            // Using cold water only
            // Ready to go to next state
            System_state_G = WASH_01;
            Time_in_state_G = 0;
            }
         }
      break;
      }

   case HEAT_WATER:
      {
      // Remain in this state until water is hot
      // NOTE: Timeout facility included here
      if (++Time_in_state_G >= MAX_WATER_HEAT_DURATION)
         {
         // Should have warmed the water by now...
         System_state_G = ERROR;
         }

      // Check the water temperature
      if (WASHER_Read_Water_Temperature() == 1)
         {
         // Water is at required temperature
         // Ready to go to next state
         System_state_G = WASH_01;
         Time_in_state_G = 0;
         }

      break;
      }

   // *** REMAINING WASH PHASES OMITTED HERE ***
   }
}
```

Listing 2: A possible implementation of the single task used to implement a washing-machine control system. Note the difference between the provisional task list (Figure 5) and this final implementation.

We can describe the simplest form of the **MULTI-STATE TASK** architecture as follows:

- The system involves the use of a number of different functions
- The functions are always called in the same sequence.
- The functions are called from a single task, as required.

Note that variations on this theme are also common: for example, the functions may not always be called in the same sequence: the precise sequence followed (and the particular set of functions called) will frequently depend on user preferences, or on some other system inputs.

6.8 Operating system reconsidered

Having opted to implement the basic system architecture using a single Multi-State Task, we may wish to re-consider our choice of operating system. In this case, the pattern **ONE-TASK SCHEDULER** may be an appropriate choice: it runs a single task, on a co-operative basis, and imposes a minimum CPU and memory load.

7. Conclusions

In this paper, we have argued that the use of appropriate 'design patterns' can greatly simplify the process of creating TTCS systems, and we have illustrated how such patterns can be employed in the creation of a non-trivial embedded application.

Patterns are more than simply a collection of design ideas, or a well-documented code library. The process of 'mining' the patterns, and the process in which they are refined and enhanced (through structured workshop sessions) is an important part of the process. However, one might legitimately ask if patterns really 'work'. So far, only a few studies have been conducted in this area: the results are inevitably qualitative, but they are encouraging (Vlissides, 2001).

Does this mean that a developer lacking any experience of embedded systems would be able to create a complete, working washing-machine controller (or similar system), from scratch, without effort? It does not. However, in our experience, developer with even limited experience can provide a means of creating reliable, TTCS applications many times more rapidly, and consistently, than would be possible without such support.

Provided that published patterns continue to represent the results of successful designs (and not simply 'good ideas') then it is difficult to see how this particular form of component-based design can be anything other than a benefit to the software community.

References

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

Alexander, C., Ishikawa, S., Silverstein, M. with Jacobson, M. Fisksdahl-King, I., Angel, S. (1977) *"A pattern language"*, Oxford University Press, NY.

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

Bate, I. (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.

Booch, G. (1994) *"Object-Oriented Analysis and Design,"* Benjamin Cummings.

Cooling, J.E. (1991) *"Software design for real-time systems"*, Chapman & Hall.

Cunningham, W. and Beck, K. (1987) "Using pattern languages for object-oriented programs", Proceedings of OOPSLA'87, Orlando, Florida.

Fowler, M. and Scott, K. (2000) *"UML Distilled" (2nd Edition)*, Addison-Wesley, Reading, MA.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *"Design patterns: Elements of reusable object-oriented software"*, Addison-Wesley, Reading, MA.

Kopetz, H. (1997) "Real-time systems: Design principles for distributed embedded applications", Kluwer Academic.

Liu, J.W.S. and Ha, R. (1995) "Methods for validating real-time constraints", *Journal of Systems and Software*.

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.

Nissanke, N. (1997) *"Realtime Systems"*, Prentice-Hall.

Noble, J. and Weir, C. *"Small Memory Software"*, Addison Wesley, 2001.

Pont, M.J. (1996) *"Software Engineering with C++ and CASE Tools"*, Addison-Wesley.

Pont, M.J. (1998) "Control system design using real-time design patterns", Proceedings of Control '98 (Swansea, UK), September 1998, pp.1078-1083.

Pont, M.J. (2000) "Designing and implementing reliable embedded systems using patterns", in, Dyson, P. and Devos, Martine (Eds.) *"EuroPLoP '99: Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999"*. ISBN 3-87940-774-6, Universitätsverlag Konstanz.

Pont, M.J. (2001) *"Patterns for time-triggered embedded systems: Building reliable systems with the 8051 family of microcontrollers"*, ACM Press, New York. [ISBN 0-201-33138-1].

Pont, M.J., Li, Y., Parikh, C.R. and Wong, C.P. (1999) "The design of embedded systems using software patterns", Proceedings of Condition Monitoring 1999 [Swansea, UK, April 12-15, 1999] pp.221-236.

Rising, L., (Ed.) "*Design Patterns in Communications Software"*, Oxford University Press, 2001.

Shaw, A.C. (2001) *"Real-Time Systems and Software"*, John Wiley & Sons, New York.

Vlissides, J. (1998) *"Pattern Hatching"*, Addison-Wesley.

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.

Yourdon, E.N. (1989) *“Modern Structured Analysis,”* Prentice-Hall.