Restructuring a pattern language which supports time-triggered co-operative software architectures in resource-constrained embedded systems

Susan Kurian and Michael J. Pont

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

Sk183@le.ac.uk ; M_Pont@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, co-operatively scheduled (TTCS) system architecture. We have been assembling this collection for almost 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. This paper briefly describes the approach that we have taken in order to re-factor and refine our original pattern collection. It goes on to describe some of the new and revised patterns that have resulted from this process.

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, co-operatively scheduled (TTCS) system architecture. Work began on these patterns in 1996, and they have since been used it 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).

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). Inevitably, by definition, a pattern language consists of an inter-related set of components: 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 language, 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.

This paper briefly describes the approach that we have taken in order to re-factor and refine our original pattern collection. It goes on to describe some of the new and revised patterns that have resulted from this process.

This paper is organised as follows. Section 1 discusses the need for a new structure. Section 2 introduces the new layered structure. Section 3 presents the scheduler patterns in the new structure. This section documents the abstract pattern - TTC PLATFORM. It documents the three design patterns – TTC-SL SCHEDULER, TTC-ISR SCHEDULER and the TTC SCHEDULER which are based on the TTC PLATFORM and a Pattern Implementation Example (PIE) for each of these patterns.

1. The need for restructuring

a) Identifying design patterns used in a project

Automatic extraction of design patterns is not a straightforward process. One of the significant differences between the design pattern and source code is the difference in abstraction of both entities. While the program source code lies at a lower level of abstraction, the design patterns used to generate the code lie at the higher abstraction level – that of the system design. It is not possible to directly extract the design from source code.

However pattern implementation examples that form part of the design pattern lie at the same level of abstraction as the source code. By analysing source code to identify usage of a

pattern implementation example we can indirectly identify the patterns used and the design requirements of the project that facilitated the use of a particular design pattern.

Most research in identifying pattern usage in source code has been done on the pattern collection compiled by Erich Gamma and colleagues (Gamma et al., 1995). For example, Keller et al. (1999) and Balanyi (2003) describe their reverse-engineering environments that are intended for use with this collection. Their techniques are based on projects using C++ source code.

When developing embedded systems however, C++ is rarely used, and the C language is a much more popular choice: the reasons for this are discussed, for example, by Pont (2003). Unfortunately, despite many advantages in an embedded setting, C lacks the support for object-oriented design that is assumed in previous tools developed for identifying pattern usage. We approach the problem of pattern identification by restructuring the PTTES pattern language.

b) Restructuring the pattern language

The original PTTES collection (Pont, 2001) labelled everything as a "pattern". In later publications (see Pont et al., 2005; Kurian and Pont, 2005) the need to layer the collection gives us three new layers:

- Abstract patterns

- Patterns, and,

- Pattern implementation examples

In this new structure, the "abstract patterns" are intended to address common design decisions faced by developers of embedded systems. 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. The problem statements for these patterns typically begin with the phrase "Should you use a …" (or something similar).

2. The restructured pattern language

As discussed in Section 1.b)., the restructured pattern language introduces two new layers:-

- Abstract patterns

- Pattern Implementation Examples (PIE)

We discuss each of these patterns in greater detail here.

a) Abstract patterns

Abstract patterns contain very abstract information. This pattern identifies a class of design problems for which design patterns are available in the catalogue. The problem addressed by each abstract pattern can be solved using one of many equivalent design patterns that are related to each abstract pattern. The actual implementation of the design pattern in the system depends on certain software and hardware characteristics of the embedded system. The PIE contains the information relevant to implementation. Each lower level pattern extensively references the patterns in the respective higher levels.
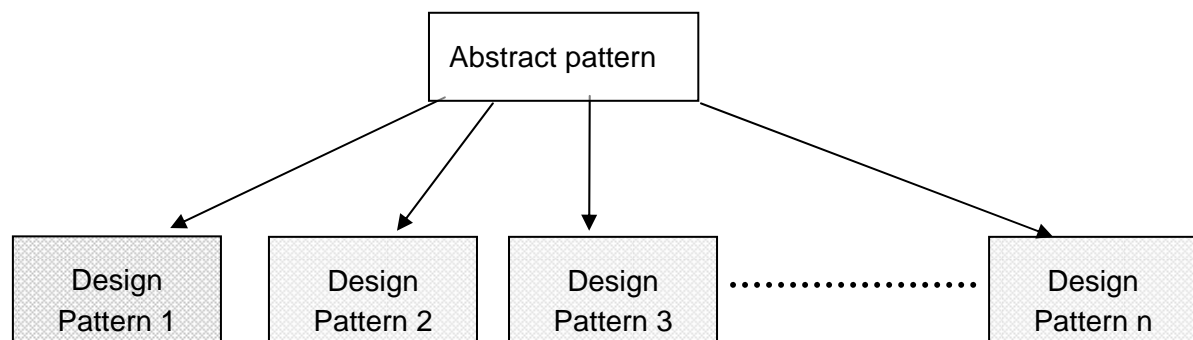

Figure 1: Abstract patterns at a higher level of abstraction compared to design patterns

b) The concept of a Pattern Implementation Example (PIE)

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.

Any PIE has a lot of implementation specific information. This also includes extensive source code examples to illustrate important design considerations. The information in a PIE lies on the same level of abstraction as source code. By identifying use of PIE in project source code, we get an insight into the patterns that have been used in the project. This is by virtue of the fact that PIEs are related to patterns which are in-turn related to abstract patterns in the new re-structured language.

Two important reasons for introducing a new PIE layer are stated below: -

- Some "low level" programming patterns have been labelled as "idioms". It was considered whether it was necessary to introduce yet another new term (i.e. PIE) into this area, or whether the term idiom could be used here. There are many possible ways of implementing any idiom, while each PIE is associated with a single (or small number of) specific implementations and so it was felt that the new term PIE was the

best way to identify this kind of pattern documentation.

- Another alternative to the use of PIEs was to simply extend each pattern with a large numbers of examples. However, this would make the pattern bulky, and difficult to use.

- In addition, new devices appear with great frequency in the embedded sector. By having distinct PIEs, we can add new implementation descriptions when these are useful, without revising the entire pattern each time we do so.

It is easy to see now that the concept of a PIE is very important in embedded systems where new hardware platforms are introduced into the market frequently. PIEs and design patterns can be used effectively in speeding up the developers understanding of a new hardware platform. Since each design pattern has a set of PIEs associated with it we have a one-to-many relationship between design patterns and PIEs. This is depicted in Figure 9.
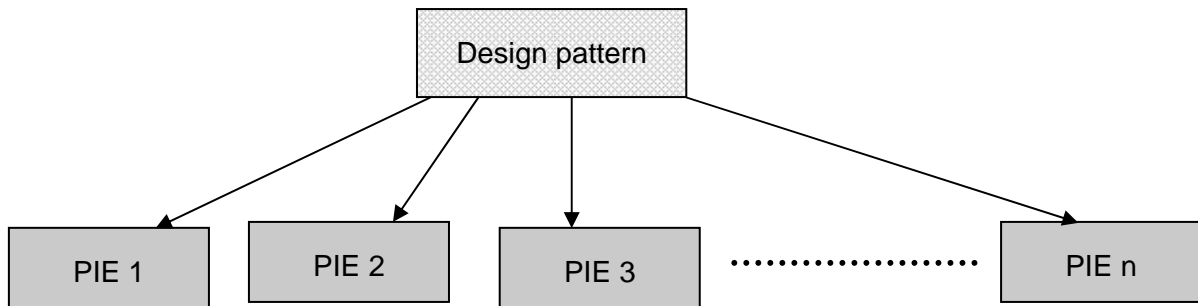

Figure 2: PIEs at a lower abstraction level compared to design patterns

c) An example - The TTC PLATFORM

The **CO-OPERATIVE SCHEDULER** described in PTTES (Pont, 2001) describes a useful design pattern that can be used to build an embedded system using a time-triggered co-operative architecture. This important design pattern provides the basis for the TTC PLATFORM depicted in Figure 3. The TTC Platform is described in Section 3 as an abstract pattern. It describes what a time-triggered co-operative (TTC) scheduler is, and discusses situations when it would be appropriate to use such an architecture in an embedded system.

If you decide to use TTC architecture, then you have a number of different implementation options available: these different options have varying resource requirements and performance figures. The design patterns TTC-SL SCHEDULER, TTC- ISR SCHEDULER (Section 3) and TTC SCHEDULER describe some of the ways in which a TTC Platform can be implemented.

In each of these "full" patterns, we refer back to the abstract pattern for background information. The TTC-SL SCHEDULER[C, 8051] (Section 3) describes how the TTC-SL SCHEDULER can be implemented on an 8051 micro-controller using the C-language.

Figure 3 depicts part of the re-structured language corresponding to the abstract pattern – TTC PLATFORM.
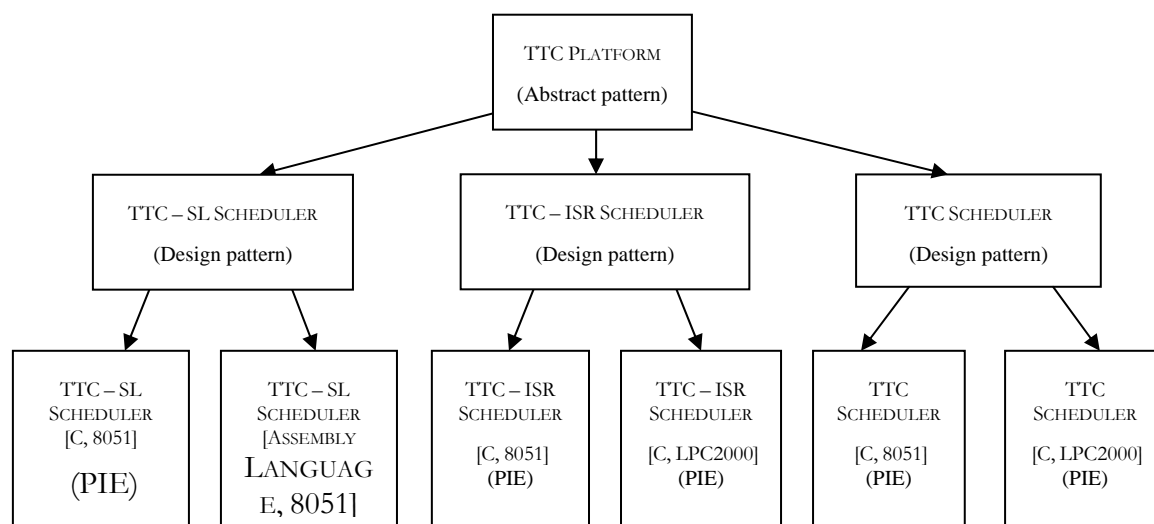
Figure 3: TTC PLATFORM design pattern

3. Different kinds of patterns related to TTC PLATFORM

This section provides the pattern documentation for each kind of pattern. The TTC PLATFORM is an example of an abstract pattern. As described in Figure 3, TTC-SL SCHEDULER, TTC-ISR SCHEDULER and TTC SCHEDULER are patterns that belong to this abstract pattern. A pattern implementation example for each of these patterns – TTC – SL SCHEDULER[C, C167], TTC – ISR SCHEDULER[C, LPC2000] and the TTC SCHEDULER[C, 8051] is also documented.

TTC PLATFORM

Context

- You are developing an embedded system.

- Reliability is a key design requirement.

Problem

Should you use a time-triggered co-operative (TTC) scheduler as the basis of your embedded system?

Background

This pattern is concerned with systems which have at their heart a time-triggered co-operative (TTC) scheduler. We will be concerned both with "pure" TTC designs - sometimes referred to as "cyclic executives" (e.g. Baker and Shaw, 1989; Locke, 1992; Shaw, 2001) – as well as "hybrid" TTC designs (e.g. Pont, 2001; Pont, 2004), which include a single pre-emptive task.

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

Tasks

Tasks are the building blocks of embedded systems. A task is simply a labeled segment of program code: in the systems we will be concerned with in this pattern a task will generally be implemented using a C function[1].

Most embedded systems will be assembled from collections of tasks. When developing systems, it is often helpful to divide these tasks into two broad categories:

- *Periodic* 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.

- *Aperiodic* tasks will be implemented as functions which may be activated if a particular event takes place. For example, an aperiodic task might be activated when a switch is pressed, or a character is received over a serial connection.

Please note that the distinction between calling a periodic task and activating an aperiodic task is significant, because of the different ways in which events may be handled. For example,

---

[1]   A task implemented in this way does not need to be a "leaf" function: that is, a task may call (other) functions.

we might design the system in such a way that the arrival of a character via a serial (e.g. RS-232) interface will generate an interrupt, and thereby call an interrupt service routine (an "ISR task"). Alternatively, we might choose to design the system in such a way that a hardware flag is set when the character arrives, and use a periodic task "wrapper" to check (or poll) this flag: if the flag is found to be set, we can then call an appropriate task.

Basic timing constraints

For both types of tasks, timing constraints are often a key concern. We will use the following loose definitions in this pattern:

- A task will be considered to have **soft timing (ST) constraints** if its execution >= 1 second late (or early) may cause a significant change in system behaviour.

- A task will be considered to have **firm timing (FT) constraints** if its execution >= 1 millisecond late (or early) may cause a significant change in system behaviour.

- A task will be considered to have **hard timing (HT) constraints** if its execution >= 1 microsecond late (or early) may cause a significant change in system behaviour.

Thus, for example, we might have a FT periodic task that is due to execute at times t = {0 ms, 1000 ms, 2000 ms, 3000 ms, …}. If the task executes at times t = {0 ms, 1003 ms, 2000 ms, 2998 ms, …} then the system behaviour will be considered unacceptable.

Jitter

For some periodic tasks, the absolute deadline is less important than variations in the timing of activities. For example, suppose that we intend that some activity should occurs at times:

t = {1.0 ms, 2.0 ms, 3.0 ms, 4.0 ms, 5.0 ms, 6.0 ms, 7.0 ms, …}.

Suppose, instead, that the activity occurs at times:

t = {11.0 ms, 12.0 ms, 13.0 ms, 14.0 ms, 15.0 ms, 16.0 ms, 17.0 ms, …}.

In this case, the activity has been delayed (by 10 ms). **For some applications – such as data, speech or music playback, for example – this delay may make no measurable difference to the user of the system.**

However, suppose that – for a data playback system - same activities were to occur as follows:

t = {1.0 ms, 2.1 ms, 3.0 ms, 3.9 ms, 5.0 ms, 6.1 ms, 7.0 ms, …}.

In this case, there is a variation (or jitter) in the task timings. Jitter can have a very detrimental impact on the performance of many applications, particularly those involving

period sampling and / or data generation (such as data acquisition, data playback and control systems: see Torngren, 1998).

For example, Cottet and David (1999) show that – during data acquisition tasks – jitter rates of 10% or more of the sampling period can introduce errors which are so significant that any subsequent interpretation of the sampled signal may be rendered meaningless. Similarly Jerri (1977) discuss the serious impact of jitter on applications such as spectrum analysis and filtering. Also, in control systems, jitter can greatly degrade the performance by varying the sampling period (Torgren, 1998; Mart et al., 2001). It is easy to see why an incorrectly sampled input signal defeats the working of a controller using this signal.

Transactions

Most systems will consist of several tasks (a large system may have hundreds of tasks, possibly distributed across a number of CPUs). Whatever the system, tasks are rarely independent: for example, we often need to exchange data between tasks. In addition, more than one task (on the same processor) may need to access shared components such as ports, serial interfaces, digital-to-analogue converters, and so forth. The implication of this type of link between tasks varies depending on the method of scheduling that is employed: we discuss this further shortly.

Another important consideration is that tasks are often linked in what are sometimes called *transactions*. Transactions are sequences of tasks which must be invoked in a specific order. For example, we might have a task that records data from a sensor (`TASK_Get_Data()`), and a second task that compresses the data (`TASK_Compress_Data()`), and a third task that stores the data on a Flash disk (`TASK_Store_Data()`). Clearly we cannot compress the data before we have acquired it, and we cannot store the data before we have compressed it: we must therefore always call the tasks in the same order:

```
TASK_Get_Data()
TASK_Compress_Data()
TASK_Store_Data()
```

When a task is included in a transaction it will often inherit timing requirements. For example, in the case of our data storage system, we might have a requirement that the data are acquired every 10 ms. This requirement will be inherited by the other tasks in the transaction, so that all three tasks must complete within 10 ms.

Scheduling tasks

As we have noted, most systems involve more than one task. For many projects, a key challenge is to work out how to schedule these tasks so as to meet all of the timing constraints.

The scheduler we use can take two forms: co-operative and pre-emptive. The difference between these two forms is - superficially – rather small but has very large implications for

our discussions in this pattern. We will therefore look closely at the differences between co-operative and pre-emptive scheduling.

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 4.
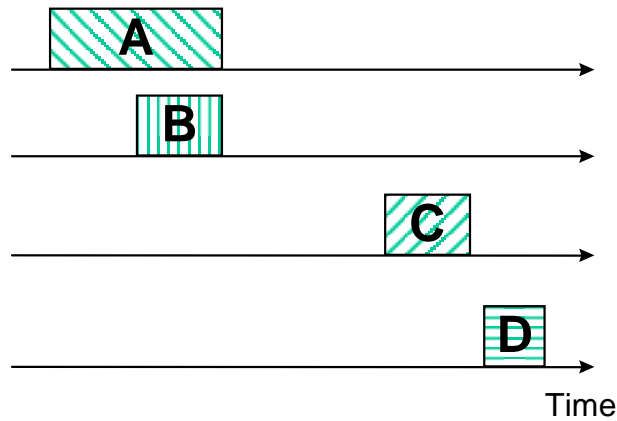
Figure 4: A schematic representation of four tasks (Task A, Task B, Task C, Task D) which we wish to schedule for execution in an embedded system with a single CPU.

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 5).
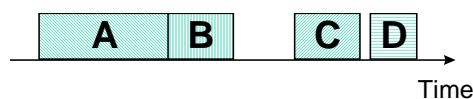
Figure 5: Scheduling Task A and Task B co-operatively.

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 6).
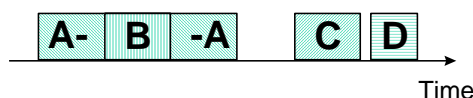
Figure 6: Assigning a high priority to Task B and scheduling the two tasks pre-emptively.

A closer look at co-operative vs. pre-emptive architectures

When compared to pre-emptive schedulers, co-operative schedulers have a number of desirable features, particularly for use in safety-related systems (Allworth, 1981; Ward, 1991; Nissanke, 1997; Bate, 2000). For example, Nissanke (1997, p.237) notes: *"[Pre-emptive] schedules carry greater runtime overheads because of the need for context switching - storage and retrieval of partially computed results. [Co-operative] algorithms do not incur such overheads. Other advantages of [co-operative] algorithms include their better understandability, greater predictability, ease of testing and their inherent capability for guaranteeing exclusive access to any shared resource or data."*. Allworth (1981, 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, Bate (2000) identifies 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.

This matter has also been discussed in the field of distributed systems, where a range of different network protocols have been developed to meet the needs of high-reliability systems (e.g. see Kopetz, 2001; Hartwich *et al.*, 2002). More generally, Fohler has observed that: "Time triggered real-time systems have been shown to be appropriate for a variety of critical applications. They provide verifiable timing behavior and allow distribution, complex application structures, and general requirements." (Fohler, 1999).

Solution

This pattern is intended to help answer the question: "Should you use a time-triggered co-operative (TTC) scheduler as the basis for your reliable embedded system?"

In this section, we will argue that the short answer to this question is "yes". More specifically, we will explain how you can determine whether a TTC architecture is appropriate for your application, and – for situations where such an architecture is inappropriate – we will describe ways in which you can extend the simple TTC architecture to introduce limited degrees of pre-emption into the design.

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.

What is a time-triggered architecture?

Systems using a time-triggered architecture use a periodic interrupt to keep track of time since

system start. Tasks are executed according to a schedule that exploits the periodic nature of interrupt occurrence. We refer to this periodic interrupt as a system tick and the period between two consequtive ticks as a tick-interval.

It is important to note that since the occurrence of an interrupt is an indication of the time used to run a schedule of tasks in the system, the interrupt source is reliable and suitably activated. For reliable tick generation only one interruot source should be used in a pure TTC system. This timer source is usually a timer set to overflow periodically.

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, 2004), and various control applications (e.g. Edwards et al., 2004; Key et al., 2004).

Of course, this architecture 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 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 7).
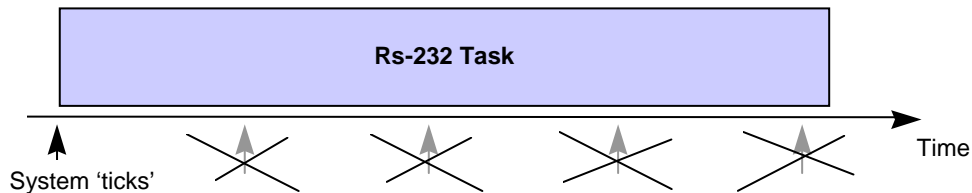
Figure 7: A schematic representation of the problems caused by sending a long character string on an embedded system with a simple operating system. In this case, sending the massage takes 42 ms while the OS tick interval is 10 ms.

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 8). 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).
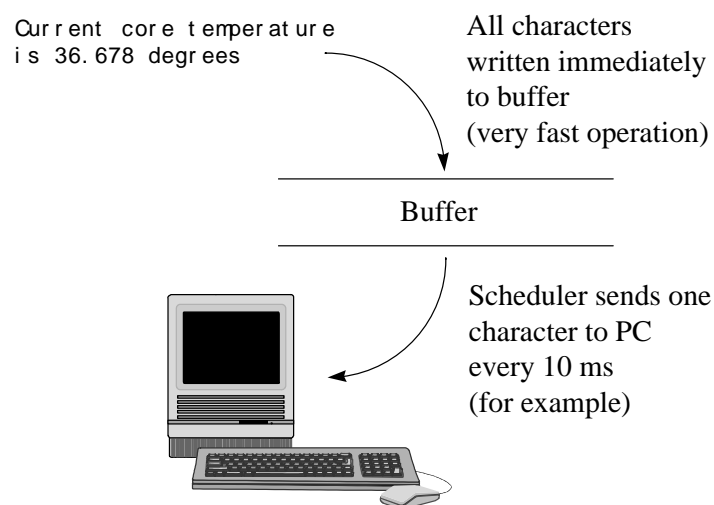
Figure 8: A schematic representation of the software architecture used in the RS-232 library.

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 baud rate of 115 kbaud. One packet will be sent every second for processing by our embedded system.

At this baud rate, data will arrive approximately every 87 μs. 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 μs. 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 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.

What should you do if a pure TTC architecture cannot meet your application needs?

In the previous two examples, we could produce a clean TTC system with appropriate design. This is – of course – not always possible. For example, consider a wireless electrocardiogram (ECG) system (Figure 9).
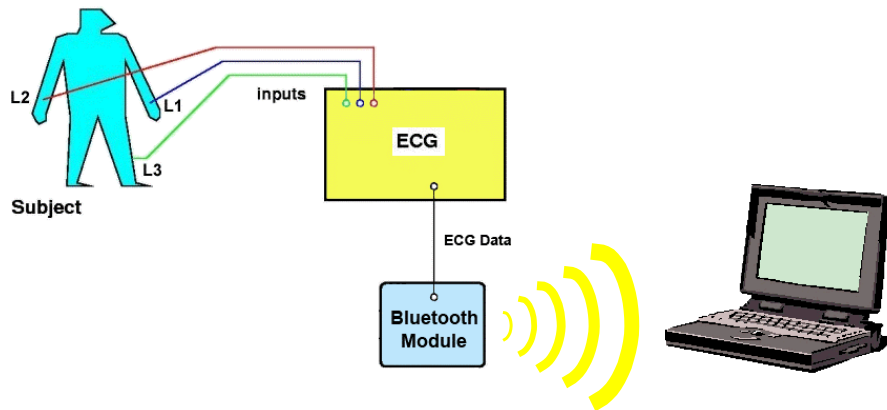
**Figure 9: A schematic representation of a system for ECG monitoring.
See Phatrapornnant and Pont (2004) for details.**

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, 2004)

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.

In such circumstances, it is tempting to opt immediately for a full pre-emptive design. Indeed, many studies seem to suggest that this is the only alternative. For example, Locke (1992) - in a widely cited publication - 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, 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."* More recently Bate (1998) compared cyclic executives and fixed-priority pre-emptive schedulers (exploring, in greater depth, Locke's study from a few years earlier).

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.
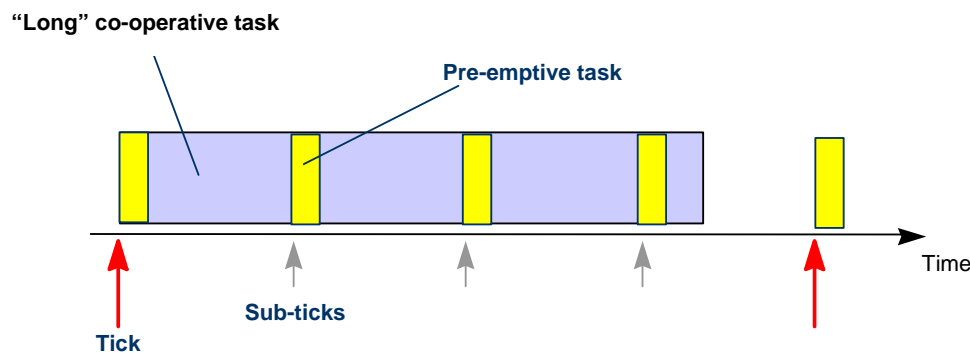
Figure 10: A "hybrid" software architecture. See text for details.

In this case, we allow a single pre-emptive 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.

As we have noted, most discussions of scheduling tend to overlook these "hybrid" architectures in favour of fully pre-emptive alternatives. When considering this issue, it cannot be ignored that the use of (fully) pre-emptive environments can be seen to have clear commercial advantages for some companies. For example, a co-operative scheduler may be easily constructed, entirely in a high-level programming language, in around 300 lines of 'C' code. The code is highly portable, easy to understand and to use and is, in effect, freely available. By contrast, the increased complexity of a pre-emptive operating environment results in a much larger code framework (some ten times the size, even in a simple implementation: Labrosse 1992). 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."*

Related patterns and alternative solutions

We highlight some related patterns and alternative solutions in this section.

Implementing a TTC Scheduler

The following patterns describe different ways of implementing a TTC PLATFORM:

- TTC-SL SCHEDULER

- TTC-ISR SCHEDULER

- TTC SCHEDULER

Alternatives to TTC scheduling

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.

Reliability and safety implications

For reasons discussed in detail in the previous sections of this pattern, 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

☺ 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.

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.

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.

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.

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.

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.

Labrosse, J. (1999) "MicroC/OS-II: The real-time kernel", CMP books. ISBN: 0-87930-543-6.

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.

Mart, P., Fuertes, J. M., Villɩ, 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.

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

Phatrapornnant, T. and Pont, M.J. (2004) "The application of dynamic voltage scaling in embedded systems employing a TTCS software architecture: A case study", Proceedings of the IEE / ACM Postgraduate Seminar on "System-On-Chip Design, Test and Technology", Loughborough, UK, 15 September 2004. Published by IEE. ISBN: 0 86341 460 5 (ISSN: 0537-9989)

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.

Pont, M.J. (2004) "A "Co-operative First" approach to software development for reliable embedded systems", invited presentation at the UK Embedded Systems Show, 13-14 October, 2004. Presentation available here: www.le.ac.uk/eg/embedded

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.

Shaw, A.C. (2001) "Real-time systems and software" John Wiley, New York. [ISBN 0-471-35490-2]

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.

TTC-SL SCHEDULER

Context

- You have decided that a TTC PLATFORM will provide an appropriate basis for your embedded system.

and

- Your application will have a single periodic task (or a single transaction).

- Your task / transaction has soft or firm constraints.

- There is no risk of task overruns (or occasional overruns can be tolerated).

- You need to use a minimum of CPU and memory resources.

Problem

How can you implement a TTC PLATFORM which meets the above requirements?

Background

See TTC PLATFORM for relevant background information.

Solution

A TTC-SL Scheduler allows us to schedule a single periodic task. To implement such a scheduler, we need to do the following:

1. Determine the task period (that is, the interval between task executions).

2. Determine the worst case execution time (WCET) of the task.

3. The required delay value is task period – WCET.

4. Choose an appropriate delay function (e.g. SOFTWARE DELAY or HARDWARE DELAY: Pont, 2001) that meets the delay requirements.

5. Implement a suitable SUPER LOOP (Pont, 2001) containing a task call and a delay call.

For example, suppose that we wish to flash an LED on and off at a frequency of 0.5 Hz (that is, on for one second, off for one second, etc). Further suppose that we have a function - `LED_Flash_Update()` – that changes the LED state every time it is called.

`LED_Flash_Update()` is the task we wish to schedule. It has a WCET of approximately 0,

so we require a delay of 1000 ms. Listing 1 shows a TTC-SL Scheduler framework which will allow us to schedule this task as required.

```
#include "Main.h"
#include "Loop_Del.h"
#include "LED_Flas.h"

void main(void)
   {
   LED_Flash_Init();

   while (1)
      {
      LED_Flash_Update();
      Loop_Delay(1000);     // Delay 1000 ms
      }
   }
```

Listing 1: Implementation of a TTC-SL SCHEDULER

Related patterns and alternative solutions

We highlight some related patterns and alternative solutions in this section.

- SOFTWARE DELAY

- HARDWARE DELAY

- TTC-ISR SCHEDULER

- TTC SCHEDULER

Reliability and safety implications

In this section we consider some of the key reliability and safety implications resulting from the use of this pattern.

Running multiple tasks

TTC-SL SCHEDULERS can be used to run multiple tasks with soft timing requirements. It is important that the worst-case execution time of each task is known before hand to set up the appropriate delay values.

Use of Idle mode and task jitter

The processor does not benefit from using idle mode. There is considerable jitter in scheduling tasks when using a SUPERLOOP in conjunction with a SOFTWARE DELAY.

What happens if a task overruns?

Task overruns are undesirable and can upset the proper functioning of the system.

Overall strengths and weaknesses

☺   Simple design, easy to implement

☺   Very small resource requirements

☹   Not sufficiently reliable for precise timing

☹   Low energy efficiency, due to inefficient use of idle mode

Further reading

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.

Pont, M.J. (2004) "A "Co-operative First" approach to software development for reliable embedded systems", invited presentation at the UK Embedded Systems Show, 13-14 October, 2004. Presentation available here: www.le.ac.uk/eg/embedded

TTC-SL SCHEDULER [C, C167]

Context

- You wish to implement a TTC-SL SCHEDULER [this paper]

- Your chosen implementation language is C^2.

- Your chosen implementation platform is the C167 family of microcontrollers.

Problem

How can you implement a TTC-SL SCHEDULER for the C167 family of microcontrollers?

Background

-

Solution

Listing 2 shows a complete implementation of a "flashing LED" scheduler, based on the example in TTC-SL Scheduler (Solution section).

---

2   The examples in the pattern were created using the Keil C compiler, hosted in a Keil uVision 3 IDE.

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

    LED_167.C (7 November, 2001)

   -------------------------------------------------------------------

    Simple 'Flash LED' test function for C167 scheduler.

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

#include "Main.h"
#include "Port.h"
#include "LED_167.h"

// ------ SFRs ------------------------------------------------------

sfr PICON = 0xF1C4;

// ------ Private variable definitions ------------------------------

static bit LED_state_G;

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

  LED_Flash_Init()

  - See below.
-*-------------------------------------------------------------------*/
void LED_Flash_Init(void)
   {
   LED_state_G = 0;

   PICON = 0x0000;

   P2   = 0xFFFF;  // set port data register
   ODP2 = 0x0000;  // set port open drain control register
   DP2  = 0xFFFF;  // set port direction register
   }

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

  LED_Flash_Update()

  Flashes an LED (or pulses a buzzer, etc) on a specified port pin.

  Must schedule at twice the required flash rate: thus, for 1 Hz
  flash (on for 0.5 seconds, off for 0.5 seconds) must schedule
  at 2 Hz.

-*-------------------------------------------------------------------*/
void LED_Flash_Update(void)
   {
   // Change the LED from OFF to ON (or vice versa)
   if (LED_state_G == 1)
       {
       LED_state_G = 0;
       LED_pin0 = 0;
       }
   else
       {
       LED_state_G = 1;
       LED_pin0 = 1;
       }
   }
```

Listing 2: Implementation of a simple task – LED_Flash_Update on C167 platform

```
//------------------------------------------------------------------
//
//   File: Delay.C (v1.00)
//   Author: M.J.Pont
//   Date: 05/12/2002
//    Description: Simple hardware delays for C167.
//
//------------------------------------------------------------------

#include "hardware_delay_167.h"

//------------------------------------------------------------------
//
//  Hardware_Delay()
//
//  Function to generate N millisecond delay (approx).
//
//  Uses Timer 2 in GPT1.
//
//------------------------------------------------------------------
void Hardware_Delay(const tWord N)
{
   tWord ms;

      // Using GPT1 for hardware delay (Timer 2)
      T2CON = 0x0000;
      T3CON = 0x0000;

      // Delay value is *approximately* 1 ms per loop
      for (ms = 0; ms < N; ms++)
   {
       // 20 MHz, prescalar of 8
       T2 = 0xF63C;  //  Load timer 2 register

       T2IR = 0;          // Clear overflow flag
       T2R = 1;           // Start timer

       while (T2IR == 0); // Wait until timer overflows

       T2R = 0;           // Stop timer
   }
}
```

Listing 3: Hardware Delay implemented on C167 platform.

```
void main(void)
   {
   // Prepare for the 'Flash_LED' task
   LED_Flash_Init();

   while(1)
      {
      LED_Flash_Update();
      Hardware_Delay(1000);
      }
   }
```

Listing 4: Using a simple SUPERLOOP architecture to schedule an LED_Flash_Update task.
The LED continuously flashes on for 1s and off for 1s

Further Reading

-

TTC-ISR SCHEDULER

Context

- You have decided that a TTC PLATFORM will provide an appropriate basis for your embedded system.

and

- Your application will have a single periodic task (or a single transaction).

- Your task / transaction has firm or hard timing constraints.

- There is no risk of task overruns (or occasional overruns can be tolerated).

- You need to use a minimum of CPU and memory resources.

Problem

How can you implement a TTC PLATFORM which meets the above requirements?

Background

See TTC PLATFORM for relevant background information.

Solution

TTC-ISR SCHEDULER is a simple but highly effective (and therefore very popular) implementation of a TTC PLATFORM.

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 11. 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 has a 10 ms "tick interval" (sometimes called a "major cycle") which – in this case – involves execution of a transaction consisting of a sequence of three tasks.

```
BACKGROUND                          FOREGROUND
PROCESSING                          PROCESSING
                                                        ┌──────────────┐
                                                        │  10ms timer  │
while(1)                      void Update(void)         └──────────────┘
    {                             {
    Go_To_Sleep();                TaskA();
    }                             TaskB();
                                  TaskC();
                                  }
```
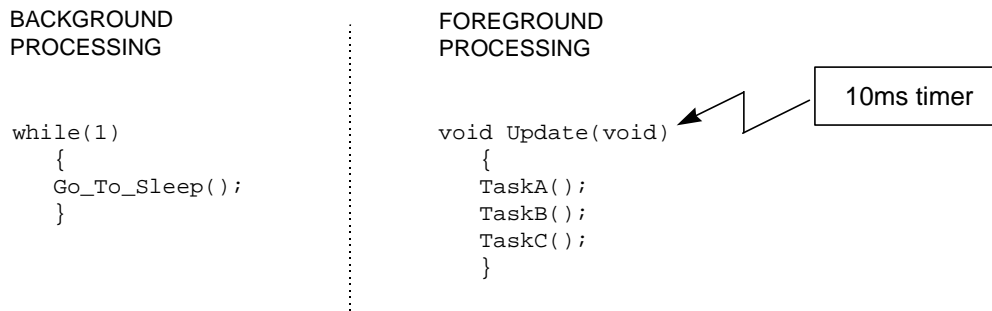
Figure 11: A schematic representation of a simple TTC scheduler ("cyclic executive").

The end result of this activity is the sequence of function calls illustrated in Figure 12.
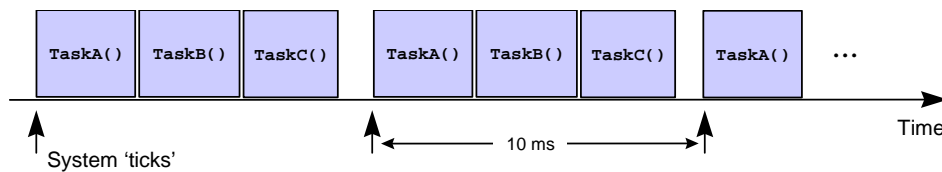
Figure 12: The sequence of task executions resulting from the architecture shown in Figure 11.

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. For example, Figure 13 shows a simple implementation, with a single periodic task implemented directly using the Update function. In this case idle mode is not used.

```
BACKGROUND                          FOREGROUND
PROCESSING                          PROCESSING



while(1)                              Update();
    {
    ; // Do "nothing"
    }
                                        ┌──────────────────┐
                                        │ One-second timer │
                                        └──────────────────┘
```
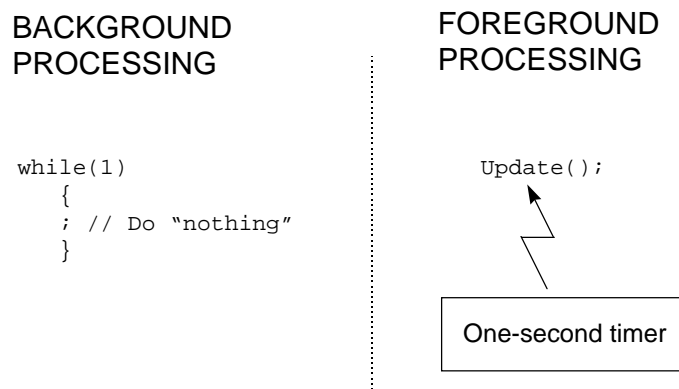
Figure 13: A schematic representation of the processes involved in using interrupts. See text for details.

Please note that – in both implementations - 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 obtained with architectures such as TTC-SL SCHEDULER.
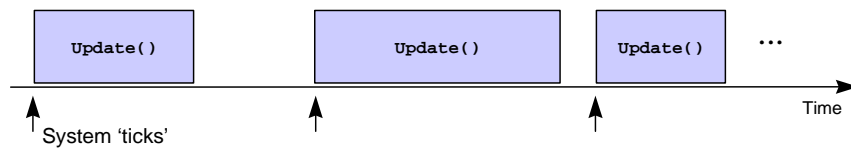
Figure 14: One advantage of the interrupt-driven approach is that the tasks will not normally suffer from "jitter" in their start times.

Hardware resource implications

We consider the hardware resource implications under three main headings: timers, memory and CPU load.

Timer

This pattern requires one hardware timer. If possible, this should be a timer, with "auto-reload" capabilities: such a timer can generate an infinite sequence of precisely-timed interrupts with minimal input from the user program.

Memory and CPU Load

The scheduler will consume no significant CPU resources: short of implementing the application using a TTC - SL SCHEDULER (with all the disadvantages of this rudimentary architecture), there is generally no more efficient way of implementing your application in a high-level language.

Reliability and safety implications

In this section we consider some of the key reliability and safety implications resulting from the use of this pattern.

Running multiple tasks

TTC-ISR SCHEDULER provides an excellent platform for executing a small number of tasks. If you need to run multiple (indirectly related) tasks, particularly tasks with different periods, then you can achieve this with a TTC-ISR SCHEDULER: however, the system will quickly become cumbersome, and may prove difficult to debug and / or maintain.

For systems with multiple tasks, please consider using a more flexible TTC approach, such as that described in CO-OPERATIVE SCHEDULER [Pont, 2001].

Safe use of idle mode

As we discussed in Solution, putting the processor to sleep means moving it into a low-power "idle" mode. Most processors have several power-saving modes: when selecting a suitable mode, make sure you choose one that (a) does not disable the timer you are using to generate the system ticks, and (b) allows the processor to enter the normal operating mode in the event of a timer interrupt.

Note also that changing the processor mode may change the behaviour of other on-chip components (such as watchdog timers). You must ensure that any facilities required by your application remain operational in the idle mode which you choose.

Use of idle mode to reduce task jitter

In addition to saving power, use of idle mode can help to reduce task jitter.

This is the case because – on most processors – instructions take varying numbers of clock cycles to execute, and your processor can only respond to an interrupt when the currently-executing instruction has completed. For example, if your timer interrupt sometimes occurs when your processor is at the start of a 100-cycle instruction, and sometimes occurs at the start of a 2-cycle instruction, then the time taken to respond to the interrupt will vary considerably. By contrast, if you use an idle mode, the time taken to return to the normal operating mode will be longer than the time taken to respond to interrupts if the processor is fully active – but the time will not generally vary.

Overall, using idle mode can usually reduce jitter, and reduce power consumption. The only drawback will be a very slight increase in the time taken to perform the task scheduling.

What happens if a task overruns?

With a TTC-ISR SCHEDULER, there will only be one active interrupt (the timer interrupt), and all tasks are called from the timer ISR. Because ISRs cannot interrupt themselves, there is no possibility that tasks in your system can be pre-empted. This results in highly predictable behaviour.

Although such behaviour is often highly desirable, it is important that you understand what happens if a task overruns. For example, suppose that you have a task that normally takes 1 ms to execute, and has to run every 10 ms. If –infrequently – this task takes 100 ms to execute, then the timer "ticks" that occur in this period will be ignored.

Inevitably, there are some applications for which this is not appropriate behaviour. For example, if you have a periodic task that keeps track of elapsed time (with a millisecond resolution), this task must run 60,000 times every minute, without fail, or your system will lose track of the current time.

To avoid losing ticks, you may need to separate the timer ISR and the process of task execution. Alternatively, you may need to consider using TTH SCHEDULER, or a TASK GUARDIAN.

Strengths and weaknesses

☺   An efficient environment for running a single periodic task or periodic transaction.

☹   Only appropriate for applications which can be implemented cleanly using a single task.

Related patterns and alternative solutions

Please also consider the following implementations of TTC PLATFORM:

- TTC-SL SCHEDULER

- TTC-ISR SCHEDULER

- TTC SCHEDULER

TTC-ISR SCHEDULER can be particularly effective if used in combination with MULTI-STATE TASK.

Further reading

-

TTC-ISR Scheduler [C, LPC2000]

Context

- You wish to implement a TTC-ISR SCHEDULER [this paper]

- Your chosen implementation language is C[3].

- Your chosen implementation platform is the Philips LPC2000 family of (ARM7-based) microcontrollers.

Problem

How can you implement a TTC-ISR SCHEDULER for the Philips LPC2000 family of microcontrollers?

Background

Timers and interrupts on the LPC2000 family

The ARM core at the heart of the LPC2000 family has seven interrupt sources (see Table 1).

Interrupt	Description
Reset	Caused by a chip reset.
Undefined instruction	An attempt has been made to execute an instruction with is not recognised.
Software interrupt	The software interrupt instruction can be used for calls to an operating system (sometimes known as a "supervisor call").
Prefetch abort	Caused by an instruction fetch memory fault.
Data abort	Caused by a data fetch memory fault.
IRQ	Used for programmer-defined interrupts which are not handled in FIQ mode.
FIQ	This provides the fastest way of responding to programmer-defined interrupts. This is generally used for handling a single critical interrupt: in this paper, it will almost always be used for handling timer interrupts.

Table 1: Interrupt sources from the ARM7 core.

Behaviour is as follows (see Furber, 2000):

1. Change to the operating mode corresponding to the exception

---

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

2. Save the address of the next instruction in r14 of the new mode

3. Save the old value of the CPSR in the SPSR of the new mode

4. Disable IRQs by setting bit 7 of the CPSR and, if the exception is a fast interrupt, disable further fast interrupts by setting bit 6 of the CPSR

5. Set the PC to the relevant vector address (above table).

Normally the vector address will contain a branch to the relevant routine.

Return behaviour from exceptions is as follows (again from Furber, 2000):

1. Any modified user registers must be restored from the handler's stack.

2. The CPSR must be restored from the appropriate SPSR.

3. The PC must be changed back to the relevant instruction address in the user instruction stream.

Note that the FIQ mode has additional private registers to give better performance by avoiding the need to save user registers. It is therefore the logical way of handling our timer interrupt in this scheduler.

The process of handling an FIQ interrupt from the timer hardware in this way is summarised in Figure 15.

Example code:

```
void Interrupt_Function(void)
    {
    ...
    }
```
Interrupt function (C language)

```
FIQ_ISR:
    STMFD   R13!, {R0-R7,R14}
    BL      Interrupt_Function
    LDMFD   R13!, {R0-R7,R14}
    SUBS    PC, R14, #4
```
Interrupt wrapper (assembly language)

(Link between timer interrupt and the assembly-language wrapper is set up in the "startip" file.)

Crystal oscillator "Fosc" PLL "cclk" VPB divider "pclk" Timer
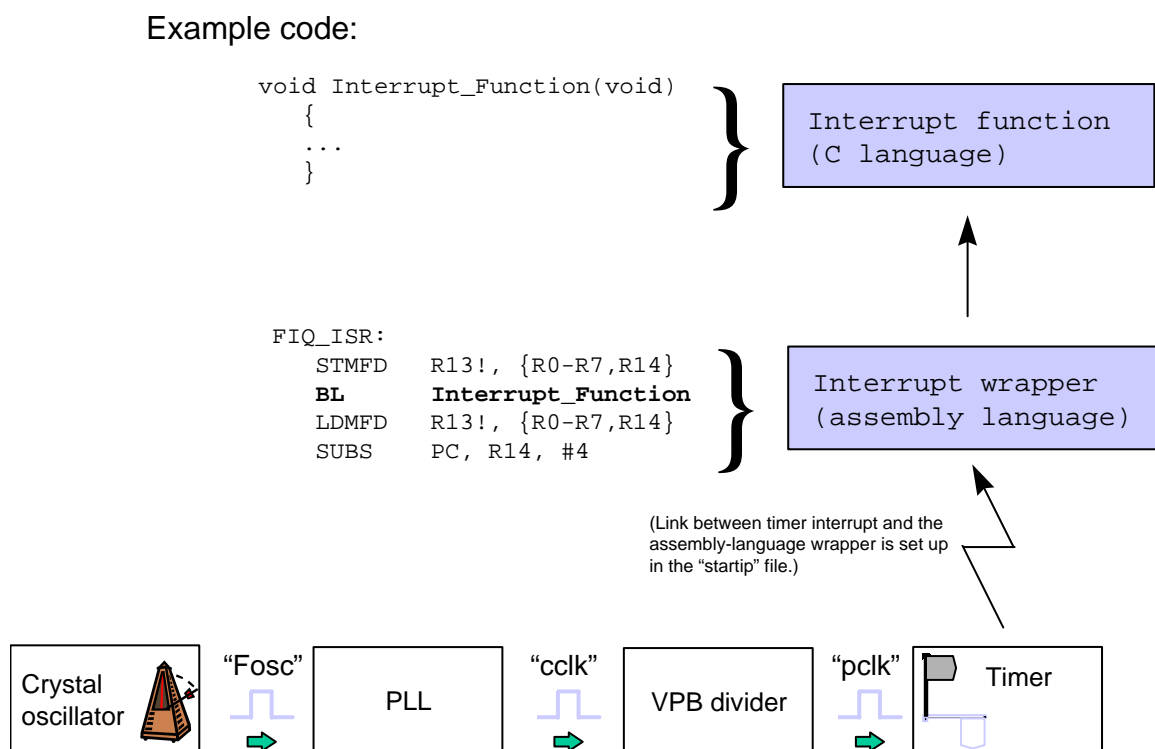
Figure 15: Interrupt handling (timers) in the LPC2000 family.

The operation of the phase-locked loop (PLL) and VLSI Peripheral Bus (VPB) divider may be clear from the code example that follows: if not, Philips (2004) provides further details.

Solution

A complete code example illustrating the implementation of a TTC-ISR SCHEDULER is given in Listing 5.

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

   main.c (v1.00)

  -------------------------------------------------------------------

   A simple "Hello Embedded World" test program for LPC2129 family.

   (P1.16 used for LED output)

-*------------------------------------------------------------------*/
// Device header (from Keil)
#include <lpc21xx.h>

// Oscillator / resonator frequency (in Hz)
// e.g. (10000000UL) when using 10 MHz oscillator
#define FOSC (12000000UL)

// Between 1 and 32
#define PLL_MULTIPLIER (5U)

// 1, 2, 4 or 8
#define PLL_DIVIDER (2U)

// 1, 2 or 4
#define VPB_DIVIDER (1U)

// CPU clock
#define CCLK (FOSC * PLL_MULTIPLIER)

// Peripheral clock
#define PCLK (CCLK / VPB_DIVIDER)

#define PLL_FCCO_MIN (156000000UL)
#define PLL_FCCO_MAX (320000000UL)

#define CCLK_MIN (10000000UL)
#define CCLK_MAX (60000000UL)


// Function prototypes
void LED_FLASH_ISR_Init(void);
void LED_FLASH_ISR_Change_State(void);

void System_Init(void);

int PLL_Init(void);
int VPB_Init(void);

void MAM_Init(void);
void Set_Interrupt_Mapping(void);

/*..................................................................*/

int main()
    {
    // Set up PLL, VPB divider and MAM (disabled)
    System_Init();

    // Prepare to flash LED
    LED_FLASH_ISR_Init();

    while(1) // Super Loop
        {
        // Enter idle mode
        PCON = 1;
        }

    // Should never reach here ...
    return 1;
```

```c
    }

// ------ Private constants -------------------------------------

// Interrupt mapping set through the "target" settings in the IDE
#ifndef RAM
#define MAP 0x01
#else
#define MAP 0x02
#endif



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

  System_Init()

   Configures:
   - PLL
   - VPB divider
   - Memory accelerator module
   - Interrupt mapping

-*-----------------------------------------------------------------*/
void System_Init(void)
    {
    // Set up the PLL
    if (PLL_Init() != 0)
        {
        while(1);  // PLL error - stop
        }

    // Set up the VP bus
    if (VPB_Init() != 0)
        {
        while(1);  // VPB divider error - stop
        }

    // Set up the memory accelerator module
    MAM_Init();

    // Control interrupt mapping
    Set_Interrupt_Mapping();
    }
```

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

   PLL_Init()

   Set up PLL.

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

int PLL_Init(void)
   {
   unsigned int Fcco;
   unsigned int PLL_tmp;

   // Cclk will be PLL_MULTIPLIER * FOSC
   // Fcco will be PLL_MULTIPLIER * FOSC * 2 * PLL_DIVIDER

   // To allow us to check the frequencies
   Fcco = CCLK * PLL_DIVIDER * 2;

   // Check that the cclk frequency is OK
   if ((CCLK > CCLK_MAX) || (CCLK < CCLK_MIN))
      {
      return 1;  // Error
      }

   // Check that the CCO frequency is OK
   if ((Fcco > PLL_FCCO_MAX) || (Fcco < PLL_FCCO_MIN))
      {
      return 1;  // Error
      }

   // Set up PLLCFG register - the divider
   switch (PLL_DIVIDER)
      {
      case 1:
         PLL_tmp = 0;
         break;

      case 2:
         PLL_tmp = 0x20;
         break;

      case 4:
         PLL_tmp = 0x40;
         break;

      case 8:
         PLL_tmp = 0x40;
         break;

      default:
         return 1;  // Error
      }

   // Set up the PLLCFG register - now the multiplier
   PLL_tmp |= PLL_MULTIPLIER - 1;

   // Apply the calculated values
   PLLCFG |= PLL_tmp;

   PLLCON = 0x00000001;  // Enable the PLL

   PLLFEED = 0x000000AA; // Update PLL registers with feed sequence
   PLLFEED = 0x00000055;

   while (!(PLLSTAT & 0x00000400))  // Test Lock bit
      {
      PLLFEED = 0x000000AA; // Update PLL with feed sequence
      PLLFEED = 0x00000055;
      }
```

```
    PLLCON = 0x00000003;  // Connect the PLL

    PLLFEED = 0x000000AA; // Update PLL registers
    PLLFEED = 0x00000055;

    return 0;
    }




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

  VPB_Init()

  Demonstrates setup of VPB divider

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

int VPB_Init(void)
    {
    // Input to VPB divider is output of PLL (cclk)

    // VPB divider consists of two bits
    // 0 0 - VPB bus clock is 25% of processor clock [DEFAULT]
    // 0 1 - VPB bus clock is same as processor clock
    // 1 0 - VPB bus clock is 50% of processor clock
    // 1 1 - Reserved (no effect - previous setting retained)

    switch (VPB_DIVIDER)
        {
        case 1:
           VPBDIV &= 0xFFFFFFFC;
           VPBDIV |= 0x00000001;
           break;

        case 2:
           VPBDIV &= 0xFFFFFFFC;
           VPBDIV |= 0x00000002;
           break;

        case 4:
           VPBDIV &= 0xFFFFFFFC;
           break;

        default:
           return 1;  // Error
        }

    // OK
    return 0;
    }
```

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

  MAM_Init()

  Set up the memory accelerator module.

  NOTE: Here we DISABLE the MAM, for maximum predictability.

  Adapt as needed for your application.

-*-------------------------------------------------------------------*/
void MAM_Init(void)
   {
   // Turn off MAM
   MAMCR = 0;
   }




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

  Set_Interrupt_Mapping()

  Remaps interrupts to RAM or Flash memory, as required.

  For Flash, MAP = 0x01
  For RAM, MAP = 0x02

  Here, value is set through Keil uVision
  (depending on target built).

-*-------------------------------------------------------------------*/
void Set_Interrupt_Mapping(void)
   {
   MEMMAP = MAP;
   }
```

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

  LED_FLASH_ISR_Init()

  Prepare for LED_FLASH_ISR_Change_State() function - see below.

-*------------------------------------------------------------------*/
void LED_FLASH_ISR_Init(void)
   {
   // First, set up the timer
   // We require a "tick" every 1000 ms
   // (Timer is incremented PCLK times every second)
   T0MR0 = PCLK - 1;

   T0MCR = 0x03;  // Interrupt on match, and restart counter
   T0TCR = 0x01;  // Counter enable

   VICIntSelect = 0x10;   // Assign "Interrupt 4" to the FIQ category
   VICIntEnable = 0x10;   // Enable this interrupt

   // Now set the mode of the I/O pin
   // using the appropriate pin function select register

   // Here we assume that Pin 1.16 is being used.

   // First, set up P1.16 as GPIO
   // Clearing Bit 3 in PINSEL2 configures P1.16:25 as GPIO
   PINSEL2 &= ~0x0008;

   // Now set P1.16 to output mode
   // through the appropriate IODIR register
   IODIR1 = 0x00010000;
   }




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

  LED_FLASH_ISR_Update()

  Changes the state of an LED (or pulses a buzzer, etc) on a
  specified port pin.

  Must call at twice the required flash rate: thus, for 1 Hz
  flash (on for 0.5 seconds, off for 0.5 seconds),
  this function must be called twice a second.

-*------------------------------------------------------------------*/
void LED_FLASH_ISR_Update(void)
   {
   static int LED_state = 0;

   // Change the LED from OFF to ON (or vice versa)
   if (LED_state == 1)
      {
      LED_state = 0;
      IOCLR1 = 0x10000;
      }
   else
      {
      LED_state = 1;
      IOSET1 = 0x10000;
      }

   // After interrupt, reset interrupt flag (by writing "1")
   T0IR = 0x01;
   }
```

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

  LOOP_DELAY_Wait()

  Delay duration varies with parameter.

  Parameter is, *ROUGHLY*, the delay, in milliseconds,
  on 12.0 MHz LPC2129 (no PLL used).

  You *WILL* need to adjust the timing for your application!

-*-------------------------------------------------------------------*/
void LOOP_DELAY_Wait(const unsigned int DELAY)
   {
   unsigned int x,y,z;

   for (x = 0; x <= DELAY; x++)
      {
      for (y = 0; y <= 1000; y++)
         {
         z = z + 1;
         }
      }
   }



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

  Trap_Interrupts()

  Interrupt trap - see Chapter 3 (Pont, 2001).

-*-------------------------------------------------------------------*/
void Trap_Interrupts(void)
   {
   // *** Basic behaviour ***
   // DISABLE ALL INTERRUPTS
   VICIntEnClr = 0xFFFFFFFF;
   while(1);
   }


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

Listing 5: Implementing a TTC-ISR SCHEDULER for the LPC2000 family (example).

Further reading

Philips (2004) "LPC2119 / 2129 / 2194 / 2292 / 2294 User Manual", Philips Semiconductors, 3 February, 2004.

Furber, S. (2000) "*ARM System-on-Chip Architecture"*, Addison-Wesley.

TTC SCHEDULER

Context

- You are developing an embedded application using the TTC PLATFORM.

- You need to schedule many tasks in your system

- You need to schedule one or more periodic (co-operative) tasks.

- You may need to schedule one or more aperiodic (co-operative) tasks.

Problem

How can you implement a TTC PLATFORM which meets the above requirements?

Background

See TTC PLATFORM for relevant background information.

Solution

We consider a TTC SCHEDULER made up of the following key components:

- The scheduler data structure.

- An initialisation function.

- A single interrupt service routine (ISR), used to update the scheduler at regular time intervals.

- A function for adding tasks to the scheduler.

- A dispatcher function that causes tasks to be executed when they are due to run.

- A function for removing tasks from the scheduler (not required in all applications).

We consider each of the required components in this section.

Overview

Before looking at the individual components, we consider how the scheduler will typically appear to the user. To do this we will use a simple example: a scheduler used to flash a single LED on and off, repeatedly: on for one second, off for one second, etc.

1. We assume that the LED will be switched on and off by means of a 'task' `LED_Flash_Update()`. Thus, if the LED is initially off and we call `LED_Flash_Update()` twice, we assume that the LED will be switched on, and then switched off again.

   To obtain the required flash rate, we therefore require that the scheduler calls `LED_Flash_Update()` every second *ad infinitum*.

2. We prepare the scheduler using the function `SCH_Init()`.

3. After preparing the scheduler, we add the function `LED_Flash_Update()` to the scheduler task list using he `SCH_Add_Task()` function. At the same time we specify that the LED will be turned on and off at the required rate as follows:

```
// Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
// - timings are in ticks (1 ms tick interval)
// (Max interval / delay is 65535 ticks)
SCH_Add_Task(LED_Flash_Update, 0, 1000);
```

4. The timing of the `LED_Flash_Update()` function will be controlled by the function `SCH_Update()`, an interrupt service routine triggered by the overflow of a timer:

```
void SCH_Update(void) // Timer-related ISR
   {
   // Increment tick variable
   ...
   }
```

5. The 'Update' ISR does not execute the task: it calculates when a task is due to run, and sets a flag. The job of executing `LED_Flash_Update()` falls the dispatcher function (`SCH_Dispatch_Tasks()`), which runs in the main ('super' loop):

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

Before considering these components in detail, we should acknowledge that this is - undoubtedly - a complicated way of flashing an LED: if our intention was to develop an LED flasher application that required minimal memory and minimal code size, then this would **not** be a good solution. However, the key point is that we will be able to use **the same scheduler architecture** for building other systems, including a number of substantial and complex applications, and the effort required to understand the operation of this environment will be rapidly repaid.

It should also be emphasised that the scheduler is a 'low cost' option: it consumes a small percentage of the CPU resources (we will consider precise percentages shortly). In addition, the scheduler itself requires no more than 7 bytes of memory for each task. Since a typical application will require no more than four to six tasks, the task-memory budget (around 40 bytes) is not excessive, even on an 8-bit microcontroller.

The scheduler data structure and task array

At the heart of the scheduler is the scheduler data structure: this is a user-defined data type which collects together the information required about each task. Each task that is scheduled in the system must have a reference to some user code (a function defined by the user). It should also have basic timing information determined by the system designer. For eg: the task interval that it is due to be first executed in and the time period at which the task is called for periodic execution. All this information is stored in a single data structure. The task period can be set to 0 to denote an aperiodic task.

The different tasks in the system that need to be scheduled are then stored as an array of this 'task' element. The scheduler processes the information in the task array, so that the tasks in any TTC system are processed as required. It is however important to store information about the maximum number of tasks that need to be scheduled at any instance. This way we can ensure that unused memory is not allocated to the task array.

A simple C-implementation of such a user defined data type is given in Listing 6.

```
typedef data struct
    {
    // Pointer to the task (must be a 'void (void)' function)
    void (code * pTask)(void);

    // Delay (ticks) until the function will (next) be run
    // - see SCH_Add_Task() for further details
    tWord Delay;

    // Interval (ticks) between subsequent runs.
    // - see SCH_Add_Task() for further details
    tWord Period;

    // Set to 1 (by scheduler) when task is due to execute
    tByte RunMe;
    } sTask;
```

Listing 6: Defining a task object using the 'struct' construct in C

Once the basic task element is defined the queue of tasks is defined as an array of task elements, as shown in Listing 7.

```
// The array of tasks
sTask SCH_tasks_G[SCH_MAX_TASKS];
```

Listing 7: Defining the task array which stores tasks to be scheduled

The initialisation function

Like most of the tasks we wish to schedule, the scheduler itself requires an initialisation function. While this performs various important operations - such as preparing the scheduler array (discussed above) and the error code variable (discussed below) - the main purpose of this function is to set up a timer that will be used to generate the regular 'ticks' that will drive the scheduler.

The designer generally needs to adapt the initialisation code to match the system requirements. In particular, we will need to ensure that:

1. The oscillator / resonator frequency assumed in the initialisation function matches thehardware.

2. The tick interval of the scheduler matches your requirements.

Listing 8 shows the code fragment that initialises the timers in an LPC2000 implementation.

```
// Set up required match register
TIMER0_MR0 = ((PCLK / 1000U) * TICK_MS) - 1;
TIMER0_MCR = 0x03;   // Interrupt on match, and automatically

// 0x10 -> 0b10000; Set bit 4 in these registers ...
VICIntSelect |= 0x10;   // Assign "Interrupt 4" to the FIQ category
VICIntEnable |= 0x10;   // Enable this interrupt
```
Listing 8: Initialising timers and interrupts in the 'Init' function

Guidance on the choice of the tick interval is provided below in the section 'Reliability and safety implications'.

The 'Update' function

The 'Update' function is the scheduler ISR. It is invoked by the overflow of the timer (set up using the 'Init' function, as discussed in the previous section).

```
{
// Note that an interrupt has occured
Tick_count_G++;

// After interrupt, reset interrupt flag (by writing "1")
TIMER0_IR = 0x01;
```
Listing 9: Registering a 'tick' in the Timer -ISR

Like most of the scheduler, the update function is not complex (Listing 8). When it determines that a task is due to run, the update function modifies the timing information of the task instance which in turn indicates that a task is due to be executed in the coming interval: the task will then be executed by the dispatcher, as we discuss below.

The 'Add Task' function

As the name suggests, the 'Add Task' function is used to add tasks to the task array, to ensure that they are called at the required time(s). The basic 'add task' function takes 3 parameters: a reference to a user function (task definition), the initial delay in tick intervals before the task can be executed for the first time and finally the period (in ticks) when the task is called repeatedly.

The parameters for the 'add task' function are described in Figure 16.

Initial_Delay
the delay (in **ticks**)
before task is first
executed. If set to 0,
the task is executed
immediately.

Sch_Add_Task(**Task_Name, Initial_Delay, Period**);

Task_Name
the name of the function
(task) that you wish to
schedule

Period
the interval (in **ticks**)
between repeated
executions of the task.
If set to 0, the task is
executed only once.
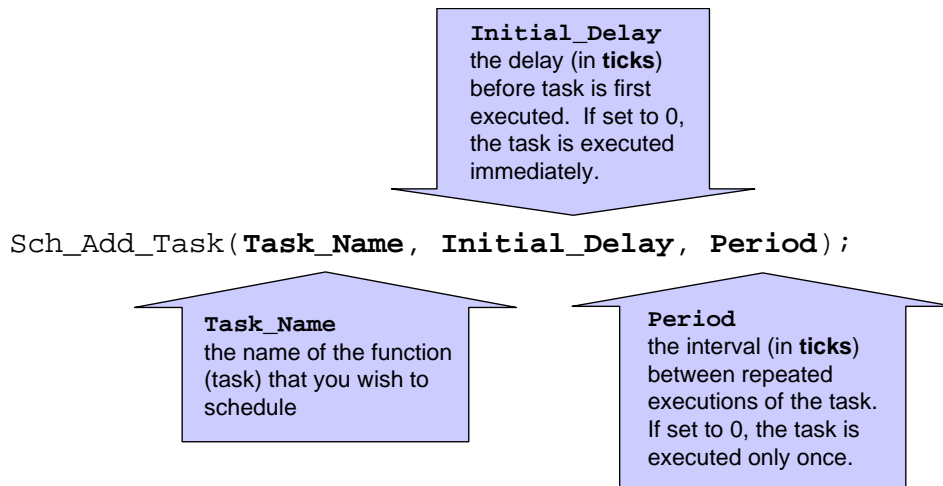
Figure 16: The parameters of the SCH_Add_Task() function

Here are some examples.

This set of parameters causes the function Do_X() to be executed once after 1000 scheduler ticks:

```
SCH_Add_Task(Do_X,1000,0);
```

This does the same, but saves the task ID (the position in the task array) so that the task may be subsequently deleted, if necessary (see 'Delete Task' function for further information about the deleting of tasks):

```
Task_ID = SCH_Add_Task(Do_X,1000,0);
```

This causes the function Do_X() to be executed regularly, every 1000 scheduler ticks; the task will first be executed as soon as the scheduling is started:

```
SCH_Add_Task(Do_X,0,1000);
```

This causes the function Do_X() to be executed regularly, every 1000 scheduler ticks; task will be first executed at T = 300 ticks, then 1300, 2300, etc:

```
SCH_Add_Task(Do_X,300,1000);
```

The 'Dispatcher'

As we have seen above, the 'Update' function does not execute any tasks: the tasks that are due to run are invoked through the 'Dispatcher' function. Suppose we have a scheduler with a tick interval of 1ms and - for whatever reason - a scheduled task sometimes has a duration of 3ms. The tasks are called as shown in

```
                if (--SCH_tasks_G[Index].Delay == 0)
                    {
                    // The task is due to run
                    (*SCH_tasks_G[Index].pTask)();  // Run the task
```

Listing 10: Executing tasks from the 'Dispatch' function

If the Update function runs the functions directly then - all the time the long task is being executed - the tick interrupts are effectively disabled. Specifically, two 'ticks' will be missed. This will mean that all system timing is seriously affected, and may mean that two (or more) tasks are not scheduled to execute at all.

If the Update and Dispatch function are separated, then system ticks can still be processed while the long task is executing. This means that we will suffer task 'jitter' (the 'missing' tasks will not be run at the correct time), but these tasks will, eventually, run.

The 'Start' function

The 'start' function is very simple. After all the tasks have been added, this function is called to begin the scheduling process. (Listing 11) The function achieves this by globally enabling interrupts in an 8051 implementation.

```
void SCH_Start(void)
    {
    TIMER0_TCR |= 0x01;    // Counter enable (Timer Counter Register)
    }
```

Listing 11: Starting the timer (LPC2000 family)

The 'Delete Task' function

When tasks are added to the task array, the 'add task' function returns the position in the task array at which the task has been added:

```
    Task_ID = SCH_Add_Task(Do_X,1000,0);
```

Sometimes it can be necessary to delete tasks from the array. To do so, a 'delete task' function can be used as follows:

```
    SCH_Delete_Task(Task_ID);
```

Reducing power consumption

An important feature of scheduled applications is that they can lend themselves to low-power operation. This is possible because all current members of many controllers provide an 'idle' mode, where the CPU activity is halted, but the state of the processor is maintained. In this mode, the power required to run the processor is typically reduced by around 50%.

```
void SCH_Go_To_Sleep()
   {
   // Lots of further power saving that can be done ...
   // - see user manual
   PCON = 1;
```

Listing 12: Sleep or power-save mode (LPC2000)

Listing 12 shows how the processor sleep mode can be used. This idle mode is particularly effective in scheduled applications because it may be entered under software control and the micro-controller returns to the normal operating mode when any interrupt is received. Because the scheduler generates regular timer interrupts as a matter of course, we can put the system 'to sleep' at the end of every dispatcher call: it will then wake up when the next timer tick occurs.

Reporting errors

Hardware fails; software is never perfect; errors are a fact of life. To report errors at any part of the scheduled application, we could use an (8-bit) error code variable Error_code_G.

To report these error codes, the scheduler has a 'report error' function, which is called from the Update function. Error reporting is an optional feature. The scheduler will work without an error reporting mechanism. Adding an error reporting feature, makes it easier to maintain the system and analyse any faults. The 8-bit error code can be written to a port periodically.

The simplest way of displaying these codes is to attach eight LEDs (with suitable buffers) to the error port, as discussed in IC DRIVER. (PONT, 2001)

Reliability and safety implications

In this section we consider some key reliability and safety implications.

Make sure the task array is large enough

See 'Solution' for details.

Take care with function pointers

See 'Background' and 'Solution' for details.

Dealing with task overlap

Suppose we have two tasks in our application (Task A, Task B). We further assume that Task A is to run every second, and Task B every three seconds. We assume also that each task has duration of around 0.5 ms.

Suppose we schedule the tasks as follows (assuming a 1ms tick interval):

```
SCH_Add_Task(TaskA,0,1000);
SCH_Add_Task(TaskB,0,3000);
```

In this case, the two tasks will sometimes be due to execute at the same time. On these occasions, both tasks will run, but Task B will always execute after Task A (see the code for `SCH_Add_Task()` for details). This will mean that if Task A varies in duration, then Task B will suffer from 'jitter': it will not be called at the correct time when the tasks overlap.

Alternatively, suppose we schedule the tasks as follows:

```
SCH_Add_Task(TaskA,0,1000);
SCH_Add_Task(TaskB,5,3000);
```

Now, both tasks still run every 1000 ms and 3000 ms (respectively), as required. However, Task A is explicitly scheduled to run, always, 5 ms before Task B. As a result, Task B will always run on time.

In many cases, we can avoid all (or most) task overlaps simply by the judicious use of the initial task delays.

Determining the required tick interval

Since, our main focus is in applications which operate on a millisecond timescale. Thus, the various tasks you will be adding to the scheduler will typically have task intervals of (say) 12 ms, 3 ms and 1000 ms.

In most instances, the simplest way of meeting the needs of the various task intervals is to allocate a scheduler tick interval of 1 ms. This is easily done: see HARDWARE DELAY [Page 181] and Chapter 13 (Pont, 2001) for details.

Remember, however, that the scheduler itself will impose a CPU load on the microcontroller, and that this load will increase dramatically at low tick intervals (see 'Hardware resource implications'). To keep the scheduler load as low as possible (and to reduce the power consumption: see below), it can help to use a long tick interval.

If you want to reduce overheads and power consumption to a minimum, the scheduler tick interval should be set to match the 'greatest common factor' of all the task (and offset intervals). This is easily calculated, if you remember some simple high-school mathematics.

Suppose we have three tasks (X,Y,Z), and Task X is to be run every 10 ms, Task Y every 30 ms and Task Z every 25 ms. The scheduler tick interval needs to be set by determining the relevant factors, as follows:

- The factors[4] of the Task X interval (10 ms) are: 1 ms, 2ms, 5 ms, 10 ms.

- Similarly, the factors of the Task Y interval (30 ms) are as follows: 1 ms, 2 ms, 3 ms, 5 ms, 6 ms, 10 ms, 15 ms and 30 ms.

- Finally, the factors of the Task Z interval (25 ms) are as follows: 1 ms, 5 ms and 25 ms.

In this case, therefore, the greatest common factor is 5 ms: this is the required tick interval.

Note that it may seem that if you have task intervals of (say) 5 ms, 25 ms and 1000 ms, then this process will be extremely tedious, because 1000 will have many factors. However, in practice, we are only concerned with the factors up to and including the smallest of the task intervals. In this case, therefore, we would be only interested in the factors of 5, 25 and 1000 between 1 and 5. The largest common factor being, in this case, 5 ms.

The situation becomes slightly more complicated if we consider the initial task delays.

If we go back to the example above, suppose we have decided to use a 5ms scheduler. We are adding three tasks to the scheduler as follows:

```
SCH_Add_Task(X, 0, 2);
SCH_Add_Task(Y, 0, 6);
SCH_Add_Task(Z, 0, 5);
```

Clearly, these tasks are going to frequently overlap. For example, every time Task Y is scheduled to run, so is Task X; on some occasions, all three tasks are due to run simultaneously. To avoid this, we can add some initial task delays, as follows:

- Task X is to be run every 10 ms: we start this task immediately.

- Task Z is to be run every 25 ms: we start this task after 2 ms.

- Task Y is to be run every 30 ms; we start this task after 1 ms.

When determining the required scheduler interval, we must now take into account both the task intervals and the initial delays. This, in this case, we now need to find the greatest common factor of 10, 25, 30, 1 and 2: this suggest a scheduler interval of 1 ms is now required.

---

4   Remember: the factors are integers (between 1 and X) by which we can divide X and obtain a remainder of 0.

Guidelines for predictable and reliable scheduling

1. For precise scheduling, the scheduler tick interval should be set to match the 'greatest common factor' of all the task intervals (see above).

2. All tasks should have a duration less than the schedule tick interval, to ensure that the dispatcher is always free to call any task that is due to execute. Software simulation can often be used to measure the task duration.

3. In order to meet Condition 2, all tasks **must** 'timeout' so that they cannot block the scheduler under any circumstances. Note that this condition can often be met by incorporating, where necessary, a LOOP TIMEOUT [Page 262] or a HARDWARE TIMEOUT [Page 268] in scheduled tasks.

   Please remember that this condition also applies to any functions called from within a scheduled task, including any library code provided by your compiler manufacturer. In many cases, standard functions (like `printf()`) do not include timeout features. They must not be used in situation where predictability is required.

4. The total time required to execute all of the scheduled tasks must be less than the available processor time. Of course, the total processor time must include both this 'task time' and the 'scheduler time' required to execute the scheduler update and dispatcher operations.

5. Tasks should be scheduled so that they are never required to execute simultaneously: that is, task overlaps should be minimised. Note that where **all** tasks are of a duration much less than the scheduler tick interval, and that some task jitter can be tolerated, this problem may not be significant.

Portability

A co-operative scheduler, like that described in this pattern, can be written entirely in 'C', for many different platforms. The TTC Scheduler[C, 8051], describes a pattern implementation example written in C-language for the 8051 family of microcontrollers.

Overall strengths and weaknesses

The overall strengths and weaknesses of a co-operative scheduler may be summarised as follows:

☺ The scheduler is simple, and can be implemented in a small amount of code.

☺ The applications based on the scheduler are inherently predictable, safe and reliable.

☺ The scheduler is written entirely in 'C': it is not a separate application, but becomes part of the developer's code

☺ The scheduler supports team working, since individual tasks can often be developed largely independently and then assembled into the final system.

☹ Obtain rapid responses to external events requires care at the design stage.

☹ The tasks cannot safely use interrupts: the only interrupt that should be used in the application is the timer-related interrupt that drives the scheduler itself.

Related patterns and alternative solutions

For alternative solutions see:

- HYBRID SCHEDULER

- ONE-TASK SCHEDULER

- ONE-YEAR SCHEDULER

- STABLE SCHEDULER

Further reading

-

TTC SCHEDULER [C, 8051]

Context

- You wish to implement a TTC SCHEDULER [this paper]

- Your chosen implementation language is C[5].

- Your chosen implementation platform is the 8051 family of microcontrollers.

Problem

How can you implement a TTC SCHEDULER for the Atmel 8051 family of microcontrollers?

Background

Function pointers and Keil linker options

When we write:

```
SCH_Add_Task(Do_X,1000,0);
```

…the first parameter of the 'Add Task' function is a *pointer* to the function Do_X(). This function pointer is then passed to the Dispatch function and it is through this function that the task is executed:

```
if (SCH_tasks_G[Index].RunMe > 0)
   {
   (*SCH_tasks_G[Index].pTask)();  // Run the task
```

The use of the 'C' function pointers on small microcontrollers presents a particular challenge. This is particularly true when function pointers are used as function arguments.

On desktop systems, function arguments are generally passed on the stack using the push and pop assembly instructions. Since the 8051 has a size limited stack (only 128 bytes at best and as low as 64 bytes on some devices), function arguments must be passed using a different technique: in the case of Keil C51, these arguments are stored in fixed memory locations. When the linker is invoked, it builds a call tree of the program, decides which function arguments are mutually exclusive (that is, which functions cannot be called at the same time), and overlays these arguments.

The linker has difficulty determining the correct call tree when function pointers are used as

---

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

function arguments, as is the case with the 'Add Task' function. To deal with this situation, you have two realistic options:

1. You can prevent the compiler from using the OVERLAY directive by disabling overlays as part of the linker options for your project.

   Note that, compared to applications using overlays, you will generally require more RAM to run your program.

2. You can tell the linker how to create the correct call tree for your application by explicitly providing this information in the linker 'Additional Options' dialogue box.

   This solution generally uses less memory, but the compiler often cannot tell if you provide incorrect information: if you get this option wrong, your program can generate unpredictable results.

The linker options required are not difficult to understand. Suppose we have run our simple flashing LED example presented earlier, and we are scheduling a single task, as follows:

```
void main(void)
   {
   //...

   // Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
   // - timings are in ticks (1 ms tick interval)
   // (Max interval / delay is 65535 ticks)
   SCH_Add_Task(LED_Flash_Update, 0, 1000);

   //...
```

The linker assumes - because the pointer LED_Flash_Update appears in main() - that the function is called from main(). Instead, the function is called from SCH_Dispatch_Tasks.

We make this change explicit using the linker options below:

```
OVERLAY
(main ~ (LED_Flash_Update),
SCH_Dispatch_Tasks ! (LED_Flash_Update))
```

Reporting errors

To report errors at any part of the scheduled application, we use an (8-bit) error code variable Error_code_G, which is defined in Sch51.C as follows:

```
// Used to display the error code
tByte Error_code_G = 0;
```

To record an error we include lines such as:

```
Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
Error_code_G = ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START;
Error_code_G = ERROR_SCH_LOST_SLAVE;
Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;
```

Listing 13: Error Codes in the project header file.

These error codes are given in the file Main.H which is an example of the pattern PROJECT HEADER. (PONT 2001)

To report these error code, the scheduler has a function `SCH_Report_Status()`, which is called from the Update function. Note that error reporting may be disabled via the `Port.H` header file:

```
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS
```

Where error reporting is required, the port on which error codes will be displayed is also determined via `Port.H`:

```
#ifdef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif
```

The simplest way of displaying these codes is to attach eight LEDs (with suitable buffers) to the error port, as discussed in IC DRIVER. (PONT, 2001)

Solution

A complete code example illustrating the implementation of a TTC SCHEDULER is given in Listing 14 and Listing 15.

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

   2_01_10i.c (v1.00)

  -------------------------------------------------------------------

   *** THIS IS A SCHEDULER FOR 80C515C (etc.) ***
   *** For use in single-processor applications  ***

   *** Uses T2 for timing, 16-bit auto reload ***

   *** This version assumes 10 MHz crystal on 515c ***
   *** 1 ms (approx) tick interval ***

   *** Includes display of error codes ***


   COPYRIGHT
   ---------

   This code is adapted from the book:

   PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
   [Pearson Education, 2001; ISBN: 0-201-33138-1].

   This code is copyright (c) 2001 by Michael J. Pont.

   See book for copyright details and other information.

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

#include "Main.h"
#include "2_01_10i.H"

// ------ Public variable declarations ---------------------------

// The array of tasks (see Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// Used to display the error code
// See Main.H for details of error codes
// See Port.H for details of the error port
extern tByte Error_code_G;

//  Used to indicate the number of times that the timer
//  has overflowed
long int Tick_count_G;

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

  SCH_Init_T2()

  Scheduler initialisation function.  Prepares scheduler data
  structures and sets up timer interrupts at required rate.
  Must call this function before using the scheduler.

-*-------------------------------------------------------------------*/
void SCH_Init_T2(void)
   {
   tByte i;

   Tick_count_G = 0;

   // Sort out the tasks
   for (i = 0; i < SCH_MAX_TASKS; i++)
      {
      SCH_Delete_Task(i);
      }

   // Reset the global error variable
   // - SCH_Delete_Task() will generate an error code,
```

```
  //   (because the task array is empty)
  Error_code_G = 0;

  // Now set up Timer 2
  // 16-bit timer function with automatic reload
  // Crystal is assumed to be 10 MHz
  // Using c515c, so timer can be incremented at 1/6 crystal frequency
  // if prescalar is not used

  // Prescaler not used -> Crystal/6
  //T2PS = 0; // No prescaler in AT89C55? -- Pete

  // The Timer 2 resolution is  0.0000006 seconds (0.6 µs)
  // The required Timer 2 overflow is 0.001 seconds (1 ms)
  // - this takes  1666.666666667 timer ticks (can't get precise timing)
  // Reload value is 65536 - 1667 = 63869 (dec) = 0xF97D

 TH2 = 0xF9;
 TL2 = 0x7D;

  RCAP2H = 0xF9;
  RCAP2L = 0x7D;


  // Compare/capture Channel 0
  // Disabled
  // Compare Register CRC on: 0x0000;
  //CRCH = 0xF9;
  //CRCL = 0x7D; // Not available on AT89C55? -- Pete

  //  Mode 0 for all channels
  T2CON = 4; // 0x11; // Needs to be 00000100b -- Pete

  //  timer 2 overflow interrupt is enabled
  ET2 = 1;
  //  timer 2 external reload interrupt is disabled
  EXEN2 = 0;

  //  CC0/ext3 interrupt is disabled
  //EX3 = 0; // Not available on AT89C55? -- Pete

  // Compare/capture Channel 1-3
  // Disabled
  //CCL1 = 0x00;
  //CCH1 = 0x00;
  //CCL2 = 0x00;
  //CCH2 = 0x00;
  //CCL3 = 0x00;
  //CCH3 = 0x00; // Not available on AT89C55? -- Pete

  // Interrupts Channel 1-3
  // Disabled
  //EX4 = 0;
  //EX5 = 0;
  //EX6 = 0; // Not available on AT89C55? -- Pete

  // all above mentioned modes for Channel 0 to Channel 3
  //CCEN = 0x00; // Not available on AT89C55? -- Pete
  // ------ Set up Timer 2 (end) --------------------------------
  }

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

  SCH_Start()

  Starts the scheduler, by enabling interrupts.

  NOTE: Usually called after all regular tasks are added,
  to keep the tasks synchronised.

  NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!
```

```
-*----------------------------------------------------------------*/
void SCH_Start(void)
   {
   // Comment out as required, depending on compiler used
   EA = 1;   // Use with C51 v5.X
   //EAL = 1;  // Use with C51 v6.X
   }

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

  SCH_Update()

  This is the scheduler ISR.  It is called at a rate determined by
  the timer settings in SCH_Init_T2().  This version is
  triggered by Timer 2 interrupts: timer is automatically reloaded.

-*----------------------------------------------------------------*/
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
   {
   TF2 = 0; // Have to manually clear this.

   Tick_count_G++;
   }

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

Listing 14: Scheduler functions as defined in 2_01_10i.C file

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

    SCH51.C (v1.00)

  ------------------------------------------------------------------

    *** THESE ARE THE CORE SCHEDULER FUNCTIONS ***
    --- These functions may be used with all 8051 devices ---

    *** SCH_MAX_TASKS *must* be set by the user ***
    --- see "Sch51.H" ---

    *** Includes (optional) power-saving mode ***
    --- You must ensure that the power-down mode is adapted ---
    --- to match your chosen device (or disabled altogether) ---


    COPYRIGHT
    ---------

    This code is from the book:

    PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
    [Pearson Education, 2001; ISBN: 0-201-33138-1].

    This code is copyright (c) 2001 by Michael J. Pont.

    See book for copyright details and other information.

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

#include "Main.h"
#include "Port.h"

#include "Sch51.h"
// Gives access to Keil _idle_() function
#include "intrins.h"
// ------ Public variable definitions ----------------------------

// The array of tasks
sTask SCH_tasks_G[SCH_MAX_TASKS];

// Used to display the error code
// See Main.H for details of error codes
// See Port.H for details of the error port
tByte Error_code_G = 0;

// ------ Private function prototypes ----------------------------

static void SCH_Go_To_Sleep(void);

// ------ Private variables --------------------------------------

// Keeps track of time since last error was recorded (see below)
static tWord Error_tick_count_G;

// The code of the last error (reset after ~1 minute)
static tByte Last_error_code_G;

extern long int Tick_count_G;


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

  SCH_Dispatch_Tasks()

  This is the 'dispatcher' function.  When a task (function)
  is due to run, SCH_Dispatch_Tasks() will run it.
  This function must be called (repeatedly) from the main loop.
```

```
-*---------------------------------------------------------------*/
void SCH_Dispatch_Tasks(void)
   {
   tByte Index;
   bit Update_again = 0;


   do {
      // 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].pTask)();  // Run the task

               if (SCH_tasks_G[Index].Period)
                  {
                  // Schedule period tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               else
                  {
                  // Delete one-shot tasks
                  SCH_tasks_G[Index].pTask  = 0;
                  }
               }
            }
         }

      // Disable Timer 2 interrupt
      ET2 = 0;

      if (--Tick_count_G > 0)
         {
         Update_again = 1;
         }
      else
         {
         Update_again = 0;
         }

      // Re-enable Timer 2 interrupt
      ET2 = 1;

      } while (Update_again);

   // Report system status
   SCH_Report_Status();

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

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

   SCH_Add_Task()

   Causes a task (function) to be executed at regular intervals
   or after a user-defined delay

   Fn_P   - The name of the function which is to be scheduled.
            NOTE: All scheduled functions must be 'void, void' -
            that is, they must take no parameters, and have
            a void return type.

   DELAY  - The interval (TICKS) before the task is first executed
```

```
    PERIOD - If 'PERIOD' is 0, the function is only called once,
             at the time determined by 'DELAY'.  If PERIOD is non-zero,
             then the function is called repeatedly at an interval
             determined by the value of PERIOD (see below for examples
             which should help clarify this).


    RETURN VALUE:

    Returns the position in the task array at which the task has been
    added.  If the return value is SCH_MAX_TASKS then the task could
    not be added to the array (there was insufficient space).  If the
    return value is < SCH_MAX_TASKS, then the task was added
    successfully.

    Note: this return value may be required, if a task is
    to be subsequently deleted - see SCH_Delete_Task().

    EXAMPLES:

    Task_ID = SCH_Add_Task(Do_X,1000,0);
    Causes the function Do_X() to be executed once after 1000 sch ticks.

    Task_ID = SCH_Add_Task(Do_X,0,1000);
    Causes the function Do_X() to be executed regularly, every 1000 sch ticks.

    Task_ID = SCH_Add_Task(Do_X,300,1000);
    Causes the function Do_X() to be executed regularly, every 1000 ticks.
    Task will be first executed at T = 300 ticks, then 1300, 2300, etc.

-*-------------------------------------------------------------------*/
tByte SCH_Add_Task(void (code * pFunction)(),
                   const tWord DELAY,
                   const tWord PERIOD)
   {
   tByte Index = 0;

   // First find a gap in the array (if there is one)
   while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
      {
      Index++;
      }

   // Have we reached the end of the list?
   if (Index == SCH_MAX_TASKS)
      {
      // Task list is full
      //
      // Set the global error variable
      Error_code_G = ERROR_SCH_TOO_MANY_TASKS;

      // Also return an error code
      return SCH_MAX_TASKS;
      }

   // If we're here, there is a space in the task array
   SCH_tasks_G[Index].pTask  = pFunction;

   SCH_tasks_G[Index].Delay  = DELAY + 1;
   SCH_tasks_G[Index].Period = PERIOD;

   SCH_tasks_G[Index].RunMe  = 0;

   return Index; // return position of task (to allow later deletion)
   }

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

   SCH_Delete_Task()

   Removes a task from the scheduler.  Note that this does
```

```
   *not* delete the associated function from memory:
   it simply means that it is no longer called by the scheduler.

   TASK_INDEX - The task index.  Provided by SCH_Add_Task().

   RETURN VALUE:  RETURN_ERROR or RETURN_NORMAL

-*-------------------------------------------------------------------*/
bit SCH_Delete_Task(const tByte TASK_INDEX)
   {
   bit Return_code;

   if (SCH_tasks_G[TASK_INDEX].pTask == 0)
      {
      // No task at this location...
      //
      // Set the global error variable
      Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;

      // ...also return an error code
      Return_code = RETURN_ERROR;
      }
   else
      {
      Return_code = RETURN_NORMAL;
      }

   SCH_tasks_G[TASK_INDEX].pTask   = 0x0000;
   SCH_tasks_G[TASK_INDEX].Delay   = 0;
   SCH_tasks_G[TASK_INDEX].Period  = 0;

   SCH_tasks_G[TASK_INDEX].RunMe   = 0;

   return Return_code;        // return status
   }


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

   SCH_Report_Status()

   Simple function to display error codes.

   This version displays code on a port with attached LEDs:
   adapt, if required, to report errors over serial link, etc.

   Errors are only displayed for a limited period
   (60000 ticks = 1 minute at 1ms tick interval).
   After this the the error code is reset to 0.

   This code may be easily adapted to display the last
   error 'for ever': this may be appropriate in your
   application.

   See Chapter 10 for further information.

-*-------------------------------------------------------------------*/
void SCH_Report_Status(void)
   {
#ifdef SCH_REPORT_ERRORS
   // ONLY APPLIES IF WE ARE REPORTING ERRORS
   // Check for a new error code
   if (Error_code_G != Last_error_code_G)
      {
      // Negative logic on LEDs assumed
      Error_port = 255 - Error_code_G;

      Last_error_code_G = Error_code_G;

      if (Error_code_G != 0)
         {
```

E5 - 61

```
                Error_tick_count_G = 60000;
                }
            else
                {
                Error_tick_count_G = 0;
                }
            }
        }
    else
        {
        if (Error_tick_count_G != 0)
            {
            if (--Error_tick_count_G == 0)
                {
                Error_code_G = 0; // Reset error code
                }
            }
        }
#endif
    }


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

  SCH_Go_To_Sleep()

  This scheduler enters 'idle mode' between clock ticks
  to save power.  The next clock tick will return the processor
  to the normal operating state.

  Note: a slight performance improvement is possible if this
  function is implemented as a macro, or if the code here is simply
  pasted into the 'dispatch' function.

  However, by making this a function call, it becomes easier
  - during development - to assess the performance of the
  scheduler, using the 'performance analyser' in the Keil
  hardware simulator. See Chapter 14 for examples for this.

  *** May wish to disable this if using a watchdog ***

  *** ADAPT AS REQUIRED FOR YOUR HARDWARE ***

-*-------------------------------------------------------------------*/
void SCH_Go_To_Sleep()
    {
    // Entering idle mode requires TWO consecutive instructions
    // on 80c515 / 80c505 - to avoid accidental triggering
    PCON |= 0x01;     // Enter idle mode (#1)
    //PCON |= 0x20;    // Enter idle mode (#2) not required on AT89C55 -- Pete
    }

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

Listing 15: Core scheduler functions defined in Sch51.C

Further Reading

-

4. Reference

Balanyi, Z.; Ferenc, R., 2003. Mining design patterns from C++ source code, Proceedings of International Conference on Software Maintenance, ICSM 2003, Publisher:IEEE Comput. Soc.

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

Keller, Rudolf K.; Schauer, Reinhard; Robitaille, Sebastien; Page, Patrick, (1999). Pattern-based reverse-engineering of design components, Proceedings - International Conference on Software Engineering, 226-235, IEEE, Los Angeles, CA, USA.

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. (2005) (Eds.) "Proceedings of the Second UK Embedded Forum" (Birmingham, UK, October 2005). Published by University of Newcastle upon Tyne

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, ("VikingPloP 2002"), pp.159-200. Published by Micrsoft Business Solutions.

Pont, M.J., 2001. Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers, ACM Press / Addison-Wesley, UK

Pont, M.J., 2003. An object-oriented approach to software development for embedded systems implemented using C, Transactions of the Institute of Measurement and Control 25(3): 217-238.

Pont, M.J., Kurian, S., Maaita, A. and Ong, R. (2005) "Restructuring a pattern language for reliable embedded systems" ESL, Technical Report 2005-01. Available for download at http://www.le.ac.uk/eg/embedded/tech_reports.htm

Yoder, J. W.; Foote, B.; Riehle, D.; Tilman, M.(1998). Metadata and active objectmodels. In: Workshop Results Submission OOPSLA'98 Addendum, 1998.