Improving the reliability of embedded systems as complexity increases: supporting the migration between event-triggered and time-triggered software architectures

FARAH LAKHANI, University of Leicester
ANJALI DAS, TTE Systems
MICHAEL J. PONT, University of Leicester

We can divide the software architectures employed in embedded systems into two categories - time-triggered (TT) and event-triggered (ET) - based on the way in which the various systems tasks are initiated. ET architectures are suitable for use with small systems of limited complexity; as systems grow, it may be necessary to migrate the existing code to a TT architecture. This paper is concerned with techniques which may be used to support the migration between ET and TT architectures.

Categories and Subject Descriptors: C.3 [Special-purpose and application-based systems]: Microprocessor/microcomputer applications – process control systems, real-time and embedded system; J.7 [Computers in other systems]: Command and control, consumer products, industrial control

General Terms: Performance, Design, Reliability, Security, Verification

Additional Key Words and Phrases: Embedded processors, Event-triggered, Time-triggered, Predictability, Design Patterns

1. INTRODUCTION

Embedded processors are found in more and more products with which we interact every day. These products range from toys, mobile phones and household appliances through to cars, planes and medical equipment. In total, it is estimated that people in the “Western world” make use of around 300 embedded processors every day of their lives (a figure which is expected to grow to 1,500 processors over the next few years).

As embedded processors become ever more ubiquitous, the complexity of software running on these processors is also increasing. This presents a number of new challenges to the developers of embedded systems.

The software architectures employed in such systems can be divided into two categories - time-triggered (TT) and event-triggered (ET) - based on the way in which the various systems tasks are initiated. ET architectures are generally considered to be suitable for use with small systems of limited complexity: as systems grow, it may be necessary to migrate the existing code to a TT architecture. This paper is concerned with techniques which may be used to support such a migration.

Migrating complex embedded software – particularly in applications which are (1) safety-critical and / or (2) require real-time behavior - can be a costly, time consuming, and risky process requiring code changes, retesting and even recertification (Oest 2008; Mosley 2006). This paper explores ways in which techniques known as “design patterns” may be able to assist in this process.

This work is supported by an HEC scholarship awarded to Farah Lakhani from the Pakistan government.

Authors’ addresses: Farah Lakhani, Embedded Systems Research Group, University of Leicester, University Road, Leicester LE1 7RH, UK; email: fnl1@le.ac.uk; Anjali Das, TTE Systems Ltd, 106 New walk, Leicester LE1 7EA, UK; email: a.das@tte-systems.com; Michael J. Pont, Embedded Systems Research Group, University of Leicester, University Road Leicester LE1 7RH, UK; email: m.pont@le.ac.uk

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers’ workshop at the 15th European Conference on Pattern Languages of Programs (EuroPLoP 2010), July 7-11, 2010, Insee Monastery, Bavaria, Germany.

Copyright 2010 is held by the author(s). ACM 978-1-4503-0259-3.
The concept of design patterns first emerged from the work of an architect, Christopher Alexander, during the 1960s and 1970s. Alexander and his colleagues published three pioneering texts (Alexander et al. 1975; Alexander et al. 1977; Alexander et al. 1979) between 1975 and 1979 that laid the foundation of use of patterns in the field of architecture. Alexander defines a pattern as “a three part rule which expresses a relation between a certain context, a problem and a solution”. The general nature of this concept makes design patterns a useful tool beyond the architecture. In particular, Alexander’s techniques have been adopted by the software-engineering community.

In the field of embedded systems, most previous work with design patterns has focused on the process of system construction (see, for example (Pont 2001)). Our particular concern in this paper is the situation in which: (1) a system has already been created using an “event-triggered” (ET) architecture; and (2) the developers wish to migrate the design from an ET to one of the possible TT architectures. We have previously introduced a small set of patterns (Lakhani et al. 2009) to support such activities. In the continuation of the same process, this introduces some new patterns.

Following the introduction, the paper is organized as follows: Section 2 provides a brief introduction to the software architectures commonly discussed in the remainder of this paper. In Section 3, a small pattern collection is introduced. Section 4 concludes the paper.

2. KEY SOFTWARE ARCHITECTURES

To ensure that an embedded system meets its requirements (such as response time, battery life, performance), choice of appropriate software architecture is extremely important.

Event-triggered (ET) architectures provide a general-purpose solution. In most ET systems, the software will be designed to respond to a number (possibly a large number) of hardware interrupts. These interrupts may arise at various frequencies and in different combinations. By contrast, in most (but not all) systems with a time-triggered (TT) architecture (Kopetz and Bauer 2002), there is a single periodic interrupt used to drive all system activities.

For small systems with limited CPU loading, ET designs are easy to build (usually easier to build than an equivalent TT design) and they can perform very effectively. However, as complexity grows and/or safety requirements are introduced, ET design become rather less attractive, not least because of the challenges involved in completing test and verification procedures when interrupts can occur “at any time”. One particular reason is that “random” interrupts can result in a vast increase in the potential paths within software when compared to code with no interrupts (Bate 1998).

For the above reasons, it can become necessary to migrate between ET and TT architectures when systems grow in complexity and/or developers require higher levels of reliability from their systems.

In this section, we consider the two possible TT architectures which can be employed when projects are migrated. The two architectures are referred to here as “Time-Triggered Co-operative” (TTC) and “Time-Triggered Hybrid” (TTH).

2.1 Time-Triggered Co-operative (TTC)

In our experience, use of a TTC (Pont 2001) software architecture provides a practical foundation for the creation of a wide range of embedded systems with very predictable behavior (even when computational resources are heavily constrained).

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

As an example, Fig.1 shows two tasks A and B under the control of a TTC scheduler with a “tick interval” of 1 millisecond. This “tick” is derived from a timer overflow: drift or jitter in this timing is, in large part, dependent on the associated computer hardware (and can be very low). Such low levels of jitter are an important requirement in many control and data-acquisition systems (for example).

In university textbooks, it is often assumed that software projects involving an architecture such as that shown in Fig.1 will begin from scratch, following a waterfall model (or similar). In practice, most designs involve re-use of at least some code from a previous system. This re-use presents many challenges. We focus on one such challenge in this paper.

2.2 Time-Triggered Hybrid (TTH)

Though TTC architecture provides a good platform for a wide range of embedded applications, in some cases it is necessary to introduce some form of pre-emption in the system. In such cases “time- triggered hybrid” or TTH schedulers may provide an effective solution (Pont 2001). The TTH scheduler supports (1) any number of co-operatively scheduled tasks (2) a single pre-empting task (which can interrupt co-operative tasks).

The operation of a typical TTH scheduler is shown in Fig. 2.
3. PATTERNS FOR SYSTEM BALANCING

In the process of migrating from ET to TT designs, the issue of "balanced code" is of particular concern. When we say that a piece of code is balanced we mean that the execution time is the same every time it runs.

The concept of balanced code may appear trivial when we consider simple code statements (for example, assignment statements), but when code contains loops or branches (for example), the balancing operation is rather less straightforward.

We are in the process of developing a set of patterns to support the development of balanced code (see Fig. 3). This paper presents three new patterns BALANCED SYSTEM, SINGLE-PATH DELAY and PLANNED PRE-EMPTION, as well as a modified version of the pattern SANDWICH DELAY (Pont et al. 2006).

BALANCED SYSTEM is an abstract pattern. An abstract pattern is an entry point to the understanding of the solution suggested for a problem at hand. The documentation for an abstract pattern consists of high level design considerations that need to be made to solve the problem addressed by the pattern (Kurian and Pont 2005).

In BALANCED SYSTEM, we explain the concept of balanced code and the reasons that could cause a system to be unbalanced.

Fig. 3. Patterns for Balancing System

3.1 BALANCED SYSTEM (Abstract Pattern)

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

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

Background
Only choosing TT architecture does not fully guarantee the system predictability as there are a number of other factors which could make a TT system unpredictable. The parameters of tasks which are running in TT architecture such as release time, execution time, finish time and deadline are required to be known in
Improving the reliability of embedded systems as complexity increases: supporting the migration between event-triggered and time-triggered software architectures.

The prior knowledge of these parameters plays an important role in guaranteeing the overall predictability of the system. However, systems those run in practice generally show considerable variations in these parameters. These variations are termed as jitter.

To understand the concept of jitter more clearly, consider the different instances of a task (Task A) as shown in Fig. 4. For tasks in TT systems, release time can be considered as the point at which we would ideally expect a task to start its execution. In actual practice this is delayed due to factors such as scheduler overhead and variable interrupt response times (Liu 2000; Maaita and Pont 2005). The actual start time of a task is always deviated from its (pre-determined) release time and we can say that tasks always suffer from release jitter - see unequal values of \( x_1, x_2 \) and \( x_3 \) in Fig. 4.

**Fig. 4: Illustration of jitter in different calls of a periodic task**

In real-time systems one important parameter is the upper bound of the execution time for a task, known as worst case execution time (WCET). Unfortunately, determining WCET of tasks is rarely straightforward (Puschner and Burns 2002; Puschner 2002; Puschner 2003a; Puschner 2003b). This is because the program code of a task may contain conditional branches and / or loops and each may take different times to execute (Liu 2000). The decision between one branch and the other during task execution is dependent on the input data. This makes predicting a branch prior to execution a very difficult task (Eisenbeis and Windheiser 1993). All these factors lead to variable execution time of a task and this is known as execution jitter (see imbalanced values of \( y_1, y_2 \) and \( y_3 \) in Fig. 4). The cascading effects of release and execution jitter will result in the deviation of task finish time, shown as \( z_1, z_2 \) and \( z_3 \) in Fig. 4.

Ideally, a predictable system should be jitter free. Considering Fig. 4 once again, we can say that in a jitter free system:

\[
\begin{align*}
x_1 &= x_2 = x_3 \\
y_1 &= y_2 = y_3 \\
z_1 &= z_2 = z_3
\end{align*}
\]

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

For some applications, such as data, speech or music playback (for example) these variations may make no measurable difference to the system. However, for applications in real-time control systems which involve sampling, computation and actuation, such delays in operations are very risky for the overall performance of the system. The presence of jitter can have a degrading impact on the performance of real-time systems or can even lead to critical failure (Phatrapornnant and Pont 2006)).

**Solution**

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

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

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

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

Overall Strengths and Weaknesses
- **A BALANCED SYSTEM** is more robust against the presence of various types of jitters in the system.
- Results in a more predictable timing behaviour of the system.
- Requires (non-exclusive) access to some hardware resources, for example, timers.
- Balancing a system requires extra effort in writing code for balancing which in turn increases CPU utilisation.

3.2 **SANDWICH DELAY (Pattern)**

Context
- You are using the pattern Balanced System.
- In your application you are running two activities one after the other.

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

Background
Suppose we have a system executing two functions periodically using a timer Interrupt service Routine (ISR), as outlined in Listing 1.
Improving the reliability of embedded systems as complexity increases: supporting the migration between event-triggered and time-triggered software architectures.

According to Listing 1, function Do_X() will be executed every 10 ms. Similarly, function Do_Y() will be executed every 10 ms, after Do_X() completes. For many resource-constrained applications (for example, control systems) this architecture may be appropriate. However, in some cases, the risk of jitter in the start times of function Do_Y() may cause problems. Such jitter will arise if there is any variation in the duration of function Do_X(). In Fig. 5, the jitter is reflected in differences between the values of ty1 and ty2 (for example).

![Fig. 5: The impact of variations in the duration of Do_X() on the release jitter of Do_Y()](image)

**Solution**

A **SANDWICH DELAY** can be used to solve this type of problem. More specifically, a **SANDWICH DELAY** provides a simple but highly effective means of ensuring that a particular piece of code always takes the same period of time to execute: this is done using two timer operations to “sandwich” the activity you need to perform. Please refer to code segment in Listing 2.

```
// ISR invoked by timer overflow every 10ms
void Timer_ISR(void)
{
    Set_Sandwich_Timer_overflow(5);
    Do_X();    //WCET approx 4.0ms
    Wait_Sandwich_Timer_Overflow();
    Do_Y();
}
```

Listing 2: Pseudo code for Sandwich delay

The timer is set to overflow after 5 ms (a period slightly longer than the WCET of Do_X()). We then start this timer before we run the function and - after the function is complete - we wait for the timer to reach 5 ms value. In this way, we ensure that as long as Do_X() does not exceed a duration of 5 ms – Do_Y() runs with minimum jitter as shown in Fig. 6.
Sandwich delays are also found to be useful for systems involving pre-emption, for example, TTH designs. In such designs controlling the execution jitter of a pre-emptive task using a delay (slightly bigger than the WCET of the pre-emptive task) showed considerable reduction in the period jitter of the co-operative task - see Fig. 7.

Reliability and Safety Implications

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

1. You need to know the duration WCET of the functions to be sandwiched. If you underestimate this value, the timer will already have reached its overflow value when your function(s) complete, and the level of jitter will not be reduced (indeed the Sandwich Delay is likely to slightly increase the jitter in this case).

2. You must check the code carefully, because the “wait” function may never terminate if the timer is incorrectly set up. In these circumstances a watchdog timer (see (Pont et al 2006) ) or a task guardian (see (Hughes and Pont 2008) ) may help to rescue your system, but relying on such mechanisms to deal with poor design or inadequate testing – of course – is not a good idea.

3. You will rarely manage to remove all jitter using such an approach, because the system cannot react instantly when the timer reaches its maximum value (at the machine-code level, the code used to poll the timer flag is more complex than it may appear, and the time taken to react to the flag change will vary slightly). A useful rule of thumb is that jitter levels of around 1 microsecond will still be seen using a Sandwich Delay.
Overall Strengths and Weaknesses

- A simple way of ensuring that the WCET of a block of code is highly predictable.
- Requires (non-exclusive) access to a timer.
- Will only rarely provide a “jitter free” solution: variations in code duration of around 1 microsecond are representative.

3.3 SINGLE PATH DELAY (Pattern)

Context

- You are using the pattern Balanced System
- You have decided to balance sections of code involving loops and implemented within the application tasks.

Problem

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

Background

Variable execution times of tasks can lead to unpredictable behaviour in systems. To understand this more clearly, consider a system running tasks A, B and C as shown in Fig. 8.

Variable execution times of tasks can lead to unpredictable behaviour in systems. To understand this more clearly, consider a system running tasks A, B and C as shown in Fig. 8.

![Fig. 8: Tasks scheduled to be run in a TT system](image)

If for any reason, task A takes a longer time to run than expected, task C will run before task B (if it has higher priority than task B) and task B will not be able to finish within the system tick as shown in Fig. 9.

![Fig. 9: Illustration of overall change in system behaviour if the execution time of task A takes longer than expected](image)

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

SINGLE PATH DELAY helps to achieve fixed execution time for tasks involving loop statements. The single-path programming approach was introduced by Peter Puschner (Puschner 2002b; Puschner 2003) as part of his extensive research on WCET analysis. According to single-path programming paradigm, programs that involve loops and decision structures (e.g., ‘if-else’) will have a single execution path. This could be achieved at the expense of higher but fixed and predictable execution time as compared to traditional programming. Single-path can be achieved by replacing input-data dependencies in the control flow by predicated code instead of branched code. Thus, the instructions are associated with predicates and get executed if the predicate evaluates to true. In other case (if instruction evaluates to false), the microprocessor replaces the instruction with a NOP (no-operation) instruction.

Translation of Conditionals

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

Using SINGLE PATH DELAY, we assign temporary variables temp1 and temp2 for storing the results of expr1 and expr2 respectively. The conditional move instruction “movt” copies the value of temp1 to the variable result if the test condition evaluates to true, otherwise processor performs a “no operation” (NOP) instruction. On the other hand, if the test condition evaluates to false, “movf” will copy the value of temp2 to result otherwise NOP instruction will be executed.

Listing 3: Sequential code generated from a branching statement using if-conversion [adapted from (Puschner 2003)]

```c
if (cond)
{
    result = expr1;
}
else
{
    result = expr2;
}
```

```c
    temp1 = expr1;
    temp2 = expr2;
    test cond;
    movt result,temp1;
    movf result,temp2;
```

In this way the translation basically generates a sequential code as shown in the right hand side code segment in Listing 3 above.

Overall Strengths and Weaknesses

- Helps to produce constant execution time for code sections involving loop statements.
- Its use is limited to hardware which supports “conditional move” or similar instructions.
- It is likely to increase the power consumption because the CPU will always execute the single-path code for a fixed (maximum) period. During this time, the processor will be in “full power” mode.
3.4 TAKE A NAP (Pattern)

**Context**
- You are using the pattern Balanced System
- Your system is extremely power constrained

**Problem**
How would you ensure the WCET of your application code sections involving loop and decision structures remains constant with reduced power consumption?

**Background**
Sandwich Delay and Single Path Delay provide ways to achieve fixed execution time. In systems where power consumption is a concern, neither a Sandwich Delay nor a Single Path Delay is an attractive solution, because – to achieve balanced code – we need to run the CPU at “full power” at all times. For such systems we need to find out a way to achieve balanced code without any extra power consumption (Gendy and Pont 2007).

**Solution**
TAKE A NAP provides a way to achieve balanced code with reduced power consumption.

Create balanced code by putting the control flow statement within a ‘Sandwich Delay’ (see pattern SANDWICH DELAY). This will ensure that the particular piece of code will always have a constant execution time. For example consider the code segment given in Listing 4.

```c
for (i = 0; i < x; i++)
{
   // body of the loop
}
Listing 4: Simple for loop
```

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

\[ \text{Time}(\text{MAX} - x) = (\text{MAX} - x) \times \frac{\text{Time}(x)}{x} \]  \hspace{1cm} (1)

Once the for loop executes x number of times, the processor is put to sleep for a duration equal to Time (MAX – x). A timer interrupt may be generated when the hardware timer count reaches the value Time(MAX - x) and this can be used to awaken the processor. Using this technique, code segment in Listing 6 is ensured to always – irrespective of the value of x – have a constant execution time equal to the value of Time(MAX) (i.e. the time spent in executing MAX number of iterations of the for loop). Thus, in addition to enabling a ‘power – saving mode of the processor, the resulting ‘balanced’ code with the Sandwich Delay incorporated, provides an additional layer of predictability to the real-time system.

The balanced version of the code segment in Listing 6 may be written as shown in Listing 5.
// Start the timer
Timer_Start();
for (i = 0; i < x; i++)
{
    // body of the loop
}
// Stop the timer
Timer_Stop();

// Store timer count value after x iterations
Time(x) = Timer_Count_Value;

// Determine value of Time(MAX - x)
Time(MAX - x) = (MAX-x) * Time(x)/x;

// Reset the timer
Timer_Reset();

// Timer interrupt to occur after Time(MAX-x)
Set_Interrupt(Time(MAX-x)+"safety margin");

// Put processor to sleep
Processor_Sleep();

Listing 5: Balancing section with reduced power consumption [adapted from Gendy and Pont 2007]

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

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

Overall Strengths and Weaknesses

- A simple technique for improving system reliability by providing an additional layer of predictability is described here.
- Ensures fixed execution time for each task in the system along with reduced power consumption.
- The maximum number of iterations of the control flow statement (i.e. the value of MAX) must be known in advance.
- Requires access to a hardware timer.
3.5 Planned Pre-emption (Pattern)

Context
- You are using the pattern Balanced System
- Your system is based on a TT scheduler specifically TTH architecture.

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

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

During normal operation of the systems using the TTH Scheduler architecture, function main() runs an endless while loop (see Listing 6) from which the function C_Dispatch() is called: this in turn launches the co-operative task(s) currently scheduled to execute. Once these tasks are completed, C_Dispatch() calls Sleep(), placing the processor into a suitable “idle” mode.

```
while(1)
{
    // Dispatch co-operative tasks
    C_Dispatch();
}

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

// Dispatch pre-emptive tasks
void P_Dispatch_ISR(void)
{
    P_Task();
}
```

Listing 6: TTH scheduler

A hybrid scheduler provides limited multi-tasking capabilities to the system. Such systems could exhibit unpredictable behaviour because of two reasons (Maaita 2008)

1. Existence of unbalanced code branches in the timer ISR which leads to variable ISR execution times. This in turn leads to unpredictable scheduler behaviour represented by the appearance of task starting jitter.

2. The existence of CPU instructions with different execution times (i.e. in terms of CPU cycles required to execute the instruction). This leads to variable timer interrupt response times as each of the periodic timer interrupt which take place throughout the life cycle of the application can occur.
while the CPU is in one of the two different states. The CPU may either be running in sleep (idle) mode, or while it is running an instruction and where the interrupt is only serviced once the currently executing instruction is finished shown in Fig.10

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

Solution
By keeping the processor in the same state as all interrupts takes place would likely to reduce the tick jitter (Maaita and Pont 2005). PLANNED PRE-EMPTION makes use of another hardware timer to put the processor to power saving mode before the scheduler timer interrupt occurs thus keeping the processor in the same state every time as shown in Fig. 11.

Power saving mode or sleep/idle mode is available in almost all embedded processor for example ARM7 and 8051 family of processors.
We are naming the extra timer used for this purpose as “PP-timer” being used for Planned Pre-emption. To set the overflow value of the PP-timer it is important to know the WCET in advance so that the processor can have enough time to go to sleep mode before the scheduler timer interrupt occurs. Planned Pre-emption will reduce the tick jitter as the time required to leave the sleep mode and pursue normal execution is a static value (Kopetz and Bauer 2002). Listing 7 illustrates the implementation of PLANNED PRE-EMPTION.

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

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

void P_Dispatch_ISR(void)
{
    // Start idle timer
    ITimer();

    // Dispatch pre-emptive task
    P_Task();
}

// Idle timer ISR
void Idle_Timer_ISR(void)
{
    Sleep();
}

Listing 7: TTH_PP Scheduler [adapted from (Maaita and Pont 2005)]
```

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

- Produces a more predictable TTH system.
- Provides a simple way of getting non variable timer interrupt response times which reduce the tick jitter.
- Makes use of additional hardware timer.
- Slight increase in memory requirements because of increased code size than normal TTH scheduler.

4. CONCLUSIONS

In this paper, we have introduced a small pattern collection which is intended to help the developers of embedded systems to create effective systems with a TT architecture (even if they have started from an ET design).

Further work is underway to expand and refine this initial collection.

5. ACKNOWLEDGEMENTS

We are thankful to Jorge L. Ortega Arjona, our Shepherd at EuroPLoP 2010, for his comments on this paper. Our special thanks to all the members of writers workshop in particular to Uwe Zudun, Veli-Pekka Eloranta and Stefan Sobering for their comments during the conference. This work was supported by an HEC scholarship awarded to Farah Lakhani from the Pakistan government. Additional support was provided by TTE Systems Ltd.

REFERENCES


Martin, T. 2005. The insider’s guide to the Philips ARM7-Based microcontrollers, Coventry, Hitex, UK, Ltd.


Pont, M.J., Kurian, S., and Bautista-Quintero, R. 2006. Meeting real-time constraints using Sandwich Delays. 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Germany.


Improving the reliability of embedded systems as complexity increases: supporting the migration between event-triggered and time-triggered software architectures. Page - 17