



**The application of scheduler agent in time-
triggered embedded systems**

Thesis submitted for the degree of

Master of Philosophy

at the University of Leicester

by

Lei Dong

Department of Engineering

University of Leicester

Leicester, United Kingdom

April 2011

The application of scheduler agents in time-triggered embedded systems

Lei Dong

Abstract

This thesis is concerned with the monitoring of embedded systems in which timing behaviour is the key concern. The focus of this work is on the development of a “scheduler agent” (SA) which is used to monitor the temporal behaviour of embedded systems. It is assumed that the system to be monitored employs a time-triggered software architecture.

This thesis begins by providing a review of timing issues in embedded systems, and followed by a review of previous research on runtime monitoring techniques (including hardware, software and “hybrid” approaches).

The SA is then introduced. It consists of two parts: an internal monitor (IM) and an external analyzer (EA). Both of the IM and the EA have to work cooperatively in order to obtain information from the target system. The communication between them relies on a GPIO interface. However, an encoding technique is required since modern microcontrollers may not have enough GPIO port pins to represent all tasks in the target system. A simple and effective encoding technique has been introduced in this thesis to address this issue.

Two versions of the SA – Passive SA (PSA) and Active SA (ASA) – are implemented. PSA retrieves task information from the “instrumented” target system passively. ASA takes advantages from the TTC architecture employed by the target system, in which the monitoring process collects information from the target system at the time when a task is due to start and end.

We also developed a SA automation tool which can automatically generate the SA code for the external analyzer and the target system. This tool is used in the case study to generate the source code for both PSA and ASA. In the case study presented in this thesis, we confirm that the functionality of SA has in line with its requirements, since it is capable to measure task execution time and detect temporal errors in the target system.

Finally, the conclusions of this thesis with a discussion of the results and some suggestions for further work in this important area are presented.

Acknowledgement

First of all, I would like to express my appreciation to my supervisor Professor Michael Pont. He has provided me with invaluable guidance and constant support throughout this research project.

Next, I would like to thank all friends and colleagues at the embedded systems lab and TTE Systems Ltd. I am gratefully for all their help and support.

Finally yet importantly, I would like to thank my parents. I cannot thank enough for their constant support over the last few years. Thank you for always being there for me.

I dedicate this thesis to my parents

Jianjun Dong and Yonghua Zhang

Table of contents

Table of contents	1
List of figures	3
List of tables	4
List of related publications	5
1 Introduction	1
1.1 Real-time embedded systems	1
1.2 Time-triggered architectures	2
1.3 Embedded systems monitoring	4
1.4 Aims of research	5
1.5 Research contributions	5
1.6 Thesis organization	6
1.7 Conclusion	6
2 Timing issues in real-time embedded systems	7
2.1 Real-time characteristic	7
2.2 Task model	7
2.3 Time-triggered scheduler	10
2.4 Timing issues	16
2.5 Conclusion	20
3 Monitoring techniques	21
3.1 Introduction	21
3.2 Hardware-based monitoring	21
3.3 Software-based monitoring	22
3.4 Hybrid monitoring	23
3.5 Other related works	25
3.6 Agent	26
3.7 Discussion	28
3.8 Conclusion	29
4 Scheduler agent	30
4.1 Introduction	30
4.2 Internal monitor	30
4.3 External analyzer	31
4.4 Communication between the IM and the EA	33
4.5 Using I/O pins with encoding for communication	35
4.6 Conclusion	36
5 Implementation of the SA	38
5.1 Introduction	38
5.2 Target specification	38

5.3	Implementation of the PSA.....	39
5.4	Implementation of the ASA	45
5.5	Conclusion	47
6	SA Automation Tool.....	49
6.1	Introduction.....	49
6.2	Target analyzer.....	50
6.3	SA generator	51
6.4	User interface	53
6.5	Conclusion	57
7	Case study	58
7.1	Introduction.....	58
7.2	SA implementation	58
7.3	Fault injection	59
7.4	Test result for PSA.....	59
7.5	Test result for ASA	62
7.6	Discussion	65
7.7	Conclusion	68
8	Discussion and Conclusion	69
8.1	Introduction.....	69
8.2	Goals and achievements.....	69
8.3	The limitations of the work	71
8.4	Future work	72
8.5	Conclusion	72
	References.....	74
	Appendix.....	84
	The original target code	84
	The adapted target code for PSA	88
	The PSA code.....	91
	The adapted target code for ASA.....	100
	The ASA code	103

List of figures

Figure 2-1 Task model	8
Figure 2-2 TTC Co-operative scheduler described in (Pont 2001).....	11
Figure 2-3 TTC flowchart.....	12
Figure 2-4 The main loop of the TTC	13
Figure 2-5 The dispatch function of TTC	13
Figure 2-6 The interrupt service routine of TTC	14
Figure 2-7 Task overrun effect.....	17
Figure 2-8 Jitter.....	19
Figure 4-1 The abstraction of the SA architecture	30
Figure 4-2 GPIO output	36
Figure 5-1 Code fragment to set the GPIO port.....	40
Figure 5-2 Code fragment to demonstrate the issues of task port pins selection.....	41
Figure 5-3 Software sensors in the ISR	41
Figure 5-4 Software sensors in the dispatch function.....	42
Figure 5-5 Flowchart of TTC scheduler with IM	43
Figure 5-6 Abstract flowchart for the EA of PSA	44
Figure 5-7 Sampling the communication pins at key points.....	45
Figure 5-8 Abstract flowchart for the EA of ASA.....	47
Figure 6-1 Overview of the SA Automation Tool.....	49
Figure 6-2 Code fragment of read “ main.c” file	50
Figure 6-3 The add task function of the TTC scheduler	50
Figure 6-4 Regex rule for task settings	50
Figure 6-5 Code fragment for getting task setting	51
Figure 6-6 Code fragment for showing an inserted locations for software sensors.....	52
Figure 6-7 Code fragment for I/O port pin	53
Figure 6-8 UI-1	54
Figure 6-9 UI-2	55
Figure 6-10 UI-3	56
Figure 6-11 UI-4	57

List of tables

Table 5-1 The specification parameters which are required by the SA.....	38
Table 5-2 The abilities of the SA.....	39
Table 5-3 Functions of the communication pins.....	40
Table 7-1 Fault injection scenarios	59
Table 7-2 PSA output for the normal system - runtime	60
Table 7-3 PSA output for the normal system – tick 10000 th	60
Table 7-4 PSA detected error NO 1	61
Table 7-5 PSA detected error NO 2.....	61
Table 7-6 PSA detected error NO 3.....	61
Table 7-7 PSA detected error NO 4.....	62
Table 7-8 PSA summary-at tick 10000 th	62
Table 7-9 ASA output for the normal system - runtime	63
Table 7-10 ASA output for the normal system – tick 10000 th	63
Table 7-11 ASA detected error NO 1	63
Table 7-12 ASA detected error NO 2	64
Table 7-13 ASA detected error NO 3	64
Table 7-14 ASA detected error NO 4	64
Table 7-15 ASA summary-at tick 10000 th	65
Table 7-16 The actual WCET measured by the PSA and ASA.....	65
Table 7-17 Comparison of the resource requirements	66
Table 7-18 Comparison of the implementation costs of the 4 SA.....	66

List of related publications

Gendy, A.K., Dong, L. and Pont, M.J., 2007. Improving the performance of time-triggered embedded systems by means of a scheduler agent, Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference.

Dong, L. and Pont, M.J., 2007. Improving the reliability of embedded systems using 'scheduler agents'. Poster presentation at "Festival of Postgraduate Research," (University of Leicester), June 29th, 2007.

Dong, L. and Pont, M.J., 2006. The application of 'scheduler agents' in time-triggered embedded systems. Poster presentation at "Festival of Postgraduate Research," (University of Leicester), June 13th, 2006.

1 Introduction

This introductory chapter provides an overview of real-time embedded systems. The research work undertaken in this thesis is briefly presented and the importance of this area is discussed.

1.1 Real-time embedded systems

An embedded system is defined as “a computer system that is part of a larger system and performs some of the requirements of that system” by Standards Coordinating Committee of The IEEE Computer Society in 1990. It combines with software and hardware components to perform dedicated functions with specific requirements (Pont, 2001; Butazzo, 2002; Ganssle and Barr, 2003; Laplante, 2004; Marwedel, 2006).

As embedded system applications are usually enclosed or embedded in a large system, users may not recognize their existence. In fact, embedded systems are ubiquitous. For example, household electric equipment, such as microwave ovens, washing machines, fridges, televisions, DVD players, printers and fax machines, are indispensable in our daily life.

Also, embedded systems are applied to automotive control systems, telecommunications, avionics, defence systems, power control systems, medical applications, etc. Another example, Anti-lock Braking System (ABS) is an embedded system used to control the mechanical parts of cars. Modern cars usually contain approximately hundred microcontrollers(or electronic control units) to control the vehicle's electrical systems and subsystems, including the engine, transmission, airbags, doors, seats, electric windows, courtesy lights, etc (Heath, 2003; Leen et al., 1999; Pop et al., 2004).

Furthermore, embedded systems are applied in safety critical systems or safety-related systems, where failure can lead to loss or injure of life, equipment, or money (). For example, the airbag control system in a car is used for protecting the drivers and the passengers in a crash (Hansson, 2006). In such a system, the airbags need to inflate at an explicit time. If the airbags inflate too early, they will deflate before the occupants hitting the interior of the vehicle. If the airbags inflate too late, the occupants will not be protected, and even suffer from the second impact caused by inflation of the airbags.

Thus, a rigorous and reliable design for such a system is needed to satisfy both the functional and timing requirements.

An embedded system has to satisfy its specifications, including functional and real-time constraints, by using tailored hardware and software. Because of this reason, the designer needs to understand its three common characteristics in order to build such an embedded application (Vahid, 2002).

The first characteristic is that an embedded system is a specific application dedicated for special purposes. Unlike a general-purpose computer which can install a wide variety of software applications, such as word processing and data processing applications, an embedded system is implemented on tailored hardware and software to perform dedicated functions. In some embedded applications, the software is pre-installed into read-only memory (ROM) which is inaccessible to the user.

The second characteristic is that an embedded system has tight resource constraints. A general-purpose computer may have one or more high performance processors with a large amount of memory. It also has hardware interfaces such as a keyboard, a mouse, and a graphical display, to enable user interaction. In contrast, an embedded system is built on variety microprocessors or microcontrollers, which have limited CPU performance and memory size. The system may cooperate with additional sensors, actuators, other peripherals and other control units. The user interfaces could simply be an I/O interface, such as a few led lights, a set of switches, keypad, LCD panel, etc.

The third characteristic is that an embedded system has real-time constraints. It must react to the environment or system changes and response in real-time. Specifically, it has to guarantee the correctness of computation results and the accuracy of the time that the results are generated (Stankovic, 1988). As the example of the airbag system explained above, timing is critical in real-time embedded systems.

1.2 Time-triggered architectures

In order to implement real-time embedded systems, time-triggered (TT) software architectures have been studied (Ward, 1991; Locke, 1992; Kopetz, 1997; Pont, 2001;

Pont, 2003; Pont and Banner, 2004; Phatrapornnant and Pont, 2006; Ayavoo et al., 2007; Kurian and Pont, 2007; Short and Pont, 2007; Gendy and Pont, 2008).

In a TT architecture, a scheduler controls a set of tasks by periodically evaluating the trigger conditions in order to determine which task should be run at that period. Such a TT architecture can provide highly predictable timing behaviour (Allworth, 1981; Nissanke, 1997; Kopeck, 1997; Shaw, 2001; Pop, 2004). This makes it suitable for the implementation of a wide range of low-cost and resource-constrained embedded systems. Especially, it is suitable for safety-related system designs.

A Time triggered co-operative scheduler (TTC) (e.g. Kopetz, 1997; Pont, 2001) which is of interest in this study, provides a pattern of single-tasking systems. It schedules tasks to run in predefined time slots. A timer is used for repeatedly generating an interrupt in a fixed interval. This interval is the system period called a scheduler tick or a tick. At each tick, the scheduler checks and updates the status of each task. If a task is due to run, the scheduler will execute the task. After all tasks are checked and executed (if required), the system goes to idle mode in order to reduce the system power consumption. At the arrival of the next tick, the system will be woken up to perform the same process again.

TTC architecture employs static scheduling with no pre-emption. This gives several advantages of using the TTC scheduler. Firstly, it is simple, predictable, easy to test and less resource conflict (Pont, 2001). Secondly, it does not require context switching, which means less overheads (Locker, 1992). In contrast, pre-emptive schedulers have greater runtime overheads which come from storage or retrieval of partially computed results (Nissanke, 1997). Thirdly, the TTC scheduler demonstrates very low levels of task jitter (Locker, 1992), and can maintain their low-jitter characteristics even when techniques such as dynamic voltage scaling (DVS) are employed to reduce system power consumption (Phatrapornnant and Pont, 2006).

However, the TTC is fragile to task overruns. When a task executes more than its expected time (the end of the tick), we called it a task overrun. This may have a serious impact on the system behaviour. Just as Buttazzo has noted: “Co-operative scheduling is fragile during overload situations, since a task exceeding its predicted execution time could generate (if not aborted) a domino effect on the subsequent tasks” (Buttazzo, 2005).

1.3 Embedded systems monitoring

Embedded systems monitoring is observation of the behaviour of an embedded system to determine whether it complies with the requirements (Tsai et al, 1990; Delgado et al., 2004). The requirements define the expected behaviours of a system. The monitor traces some points of interest during the execution of the system in order to obtain its actual behaviour. Since the monitoring process performs at runtime, it is often called as runtime monitoring.

In runtime monitoring, faults in the system can be detected by comparing the actual behaviour with the expected behaviour of the system. A “fault” in this study is defined as a system defect, i.e. an incorrect state of hardware and software (IEEE, 1990). A deviation of an expected operation is called an error which can be a result of a fault or another error. A “failure” occurs when the system fails to perform its required function. This may be caused by a fault or an error (Storey, 1996; Koren, 2007).

Monitoring an embedded system requires three general activities (Dodd and Ravishankar, 1992):

- i. Adding a monitor to the monitored (or target) system – a monitor could be implemented using a specific hardware, or a piece of code.
- ii. Event detection – identifying an event from the data sent by the monitor.
- iii. Event processing – check if the event is complying with the requirements.

To implement the monitor, there are three basic approaches: hardware-based, software-based and hybrid approaches. Briefly, hardware-based monitor is built on hardware which is independent to the target system. Software-based monitor is a piece of code that is inserted into the target software. Hybrid monitor is a combination of the software and hardware approaches which can minimize the shortcomings of both approaches. Detailed discussions can be seen in chapter 3.

Monitoring embedded systems is not straightforward – many embedded systems are implemented in electronics and sealed in small packages which limit the access from the external world. Also, the resource-constrained nature of embedded systems makes it even more difficult to be monitored; since only very little resource in the system can be

used for the monitor (Cadamuro and Renaux, 2008). For example, the stringent real-time requirements of embedded systems limit the CPU utilisation for the monitor. Additionally, inserting instrumentation codes into the target software may not be possible due to the limitation of the memory size. Even if it is possible, adding a monitor to the target system may alter the system behaviour unintentionally. This adverse outcome is also known as “probe effect”.

1.4 Aims of research

This research study aims to develop a hybrid monitoring solution for detecting timing faults in real-time embedded systems which employ a TTC scheduler.

This monitoring system is called “scheduler agent” (SA). Specifically, the goal of SA is to detect task overruns and task execution ordering errors. We aim to achieve this goal using minimum target resources with as low interference to the target system as possible.

In order to comply with these requirements, the hybrid monitoring approach was selected. This is because hybrid monitoring approach requires less resource from the target system than software-based approach; since the instrumentation code for fault detection is migrated to an external analyzer. Hence, the interference to the target system is reduced.

Besides of the above mentioned aims, we also aim to develop a SA automation tool which can automatically generate the SA code for the external analyzer and the target system.

1.5 Research contributions

The research project described in this thesis made the following contributions to this research area:

1. Design and implementation of a hybrid monitoring technique (i.e. the "scheduler agent") that can examine the temporal behaviour of tasks running on a target embedded system.

Two different versions of SA were developed: Passive AS (PSA) and Active SA (ASA). Briefly, PSA retrieves task information from the “instrumented” target system passively. ASA takes advantages from the TTC architecture in the target

system in which the monitoring process collects information from the target system at the time when a task is due to start and end. This approach reduces the CPU usage significantly without losing performance.

2. Developed a code-generation tool ("SA Automation Tool") to generate a customized SA code for the external analyzer and the target system.
3. Evaluated the SA which was generated by the SA Automation Tool.

The details of the above mentioned contributions are presented in the rest of this thesis.

1.6 Thesis organization

This thesis is organised as follows. Chapter 2 presents the details of TT architecture and the related timing issues that involve in real-time embedded system designs. Chapter 3 presents the literature review of some existing runtime monitoring techniques. In Chapter 4, the concept and the methodology of the SA are introduced. The implementations of two versions of SA (i.e. PSA and ASA) are explained in Chapter 5. Chapter 6 describes the design of the SA Automation tool. Chapter 7 provides a case study to evaluate two different versions of SA which are generated by the SA Automation Tool. Discussions and conclusions are presented in Chapter 8.

1.7 Conclusion

This introductory chapter has presented an overview of real-time embedded systems and emphasized on time-triggered approaches. Despite the advantages of TT, it has real-time constraints and is fragile to task overruns. Runtime monitoring can observe the system behaviours which provide a better understanding of the target system. The remainder of this thesis will describe the work undertaken throughout this project.

2 Timing issues in real-time embedded systems

In the introductory chapter, it was argued that temporal behaviours play a critical role in embedded system design. This chapter provides a further discusses on the related timing issues in time-triggered architecture.

2.1 Real-time characteristic

Embedded systems are widely applied in real-time applications. It is the real-time characteristic that requires i) the correctness of computation results and ii) the accuracy of the time that results are generated (Stankovic, 1988). Thus, the result that is produced too late is considered to be useless and may cause system failure. This time limit is called as deadline.

Based on the consequences of missing the deadline, real-time systems are classified into three categories: hard, soft and firm real-time systems (Laplante, 2004; Buttazzo, 2005; Nolte, 2006).

A hard real-time system is a system that missing a deadline may result in catastrophic consequences.

A soft real-time system is a system that missing a deadline only degrades the performance of the system, and the system could keep running.

A firm real-time system is a system that missing a few deadlines will not lead to a total failure, but missing more than a few may lead to a complete failure.

In embedded systems in general, and in safety-related applications in particular, real-time characteristic must be achieved. In this study, timing related issues are the main focus. The following sections will go through the related subjects in detail.

2.2 Task model

In general, real-time embedded systems can be considered as i) a set of tasks that needs to be performed, ii) a runtime scheduler that controls task states, and iii) shared resources that are used by tasks (Audsley, 1993; Nissanke, 1997; Shaw, 2001).

“A task is the execution of a sequential program. It starts with the reading of the input data and the internal state, and terminates with the production of the results and updating the internal state” defined by Kopetz (1997). A system normally has multiple tasks; each task performs a specific function.

2.2.1 Key parameters of a task

The following parameters are frequently used to represent task’s timing proprieties (Liu and Layland, 1973; Buttazzo, 2002).

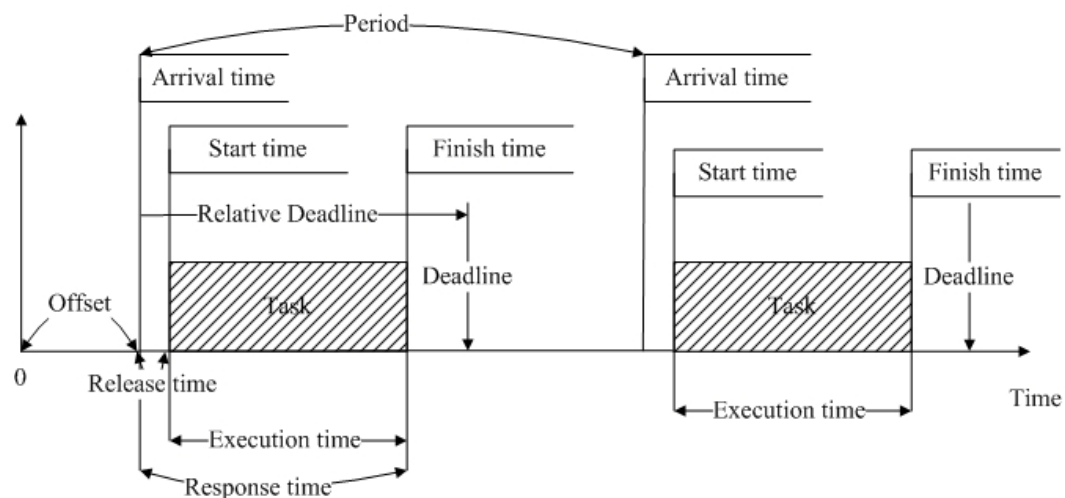


Figure 2-1 Task model

- **Period** (Periodicity) is the time interval between two successive invocations of a task. Invocation is the act of requesting a task to be executed.
- **Offset** is the time from the start of the system to the first arrival of a task.
- **Release time** is the time from the arrival time of a task to the start time of the task.
- **Response time** is the time from the arrival time of a task to the finish time of the task.
- **Execution time** is the time that the processor needs to execute a task.
- **Worst case execution time (WCET)** is the worst possible execution time of a task.
- **Best case execution time (BCET)** is the best possible execution time of a task.
- **Deadline** (also called absolute deadline) is the time which a task needs to complete.
- **Relative deadline** is the time from the arrival time of a task to absolute deadline of the task.

Tasks can be classified into three types: periodic, sporadic and aperiodic (Kopetz 1997). A periodic task infinitely repeats itself in a fixed period. A sporadic task may occur at arbitrary points in time, and has a minimum inter-arrival time. An aperiodic task also occurs at arbitrary points in time but has no minimum inter-arrival time. Usually, an aperiodic task is triggered by an (internal or external) event.

2.2.2 Task constraints

As embedded systems have different requirements, some constraints must be imposed on tasks accordingly. Such task constraints including: timing constraints, precedence constraints, and resource constraints (Buttazzo, 2002).

- **Timing constraints**

Timing constraints, such as deadline, period, response time and jitter, aim to restrict the timing behaviour of tasks. For example, a TTC scheduler contains a set of tasks, $T = (T_1, T_2, T_3 \dots T_n)$ which are periodic and independent. These tasks can be defined with timing constraints as $T_i = (\text{deadline}, \text{period}, \text{offset}, \text{WCET}, \text{BCET}, \text{jitter})$.

In previous section, deadline, period, offset, WCET, BCET has been defined. The task has to be executed within the deadline, and its execution time should be in the boundary of the WCET and BCET. Jitter represents the variability in time. Jitter issue is discussed in section 2.4.3.

- **Precedence constraints**

In some cases, tasks must execute in a fixed order, since certain tasks cannot be done unless their predecessor tasks are completed. Such tasks are called dependent tasks. The specified execution order between the tasks is the precedence constraint (Sandström and Norström, 2002). For example, Task1 is a processing task that requires the data from Task2. Therefore, the Task2 has to be executed before Task1. Such dependent tasks should be taken special care.

- **Resource constraints**

A system resource can be a physical entity such as a processor, a network link, a peripheral device or an external memory, which can be accessed by multiple tasks. A system resource can also be any software structure such as a piece of code, a data structure, a set of variables, a memory area, a set of registers, that used for task execution (Buttazzo, 2002). A shared (or common) resource is a resource that can be used by multiple tasks. Access the shared resource by multiple tasks at the same time, may cause corruption. Instead, they must access the shared resource sequentially.

2.3 Time-triggered scheduler

Tasks in a real-time embedded system usually have their own timing properties. According to these properties, a scheduler is introduced to control these tasks. At every moment of the time, the scheduler determines which task is ready, which task should be executed, which task should be blocked, etc.

A scheduler is a software module which controls task executions and allocates system resources to the tasks according to a pre-defined scheduling algorithm. The aim of a scheduler is to ensure that all tasks in the system meet their timing constraints, or at least meet the most important timing constraints, i.e. the hard deadlines. Scheduling algorithm is the method of deciding feasible tasks sequences. If a task set can be scheduled to meet the given requirements, then this task set is termed feasible.

There are various scheduling algorithms available for different real-time embedded system design (Buttazzo, 2002; Marwedel, 2006; Sha, 2004). Our focus in this study, however, is on the embedded systems built on time-triggered architecture. Time-triggered architecture is developed from cyclic executives. There are similar structures, such as timelines, frame-based systems (Baker, 1989; Locke, 1992; Shaw, 2001; Pop, 2004). In time-triggered architecture, the scheduler periodically evaluates trigger conditions in order to decide which task should be run at that time slot (Kopetz, 1997). So the system is complete in the control of time. The tasks are only activated at the specific time that pre-determined based on their properties.

Therefore, systems which employ a TT architecture can have highly predictable timing behaviour. This makes systems much easier to validate, test and certify (Liu, 2000; Marti, 2002). Especially, it suits for the implementation of safety critical systems.

2.3.1 Time-triggered co-operative scheduler

Time-triggered co-operative scheduler (TTC) (Kopetz, 1997; Pont, 2001) schedules tasks to run in specific time slots. A timer is used for repeatedly generating an interrupt in a fixed interval. This interval is the system period called a scheduler tick or a tick. At each tick, the scheduler checks and updates the task status. If the task is due to run, the scheduler will execute the task. After all tasks are checked and executed (if required), the system goes to idle mode in order to reduce the system power consumption. Then, the scheduler is waiting for the next tick arrives.

The tasks in TTC scheduler are periodical tasks and one-shot tasks. For example, there are three tasks in the TTC scheduler shown in Figure 2-2 where they are scheduled to run non-preemptively. Also, the tasks are assigned with fixed priority.

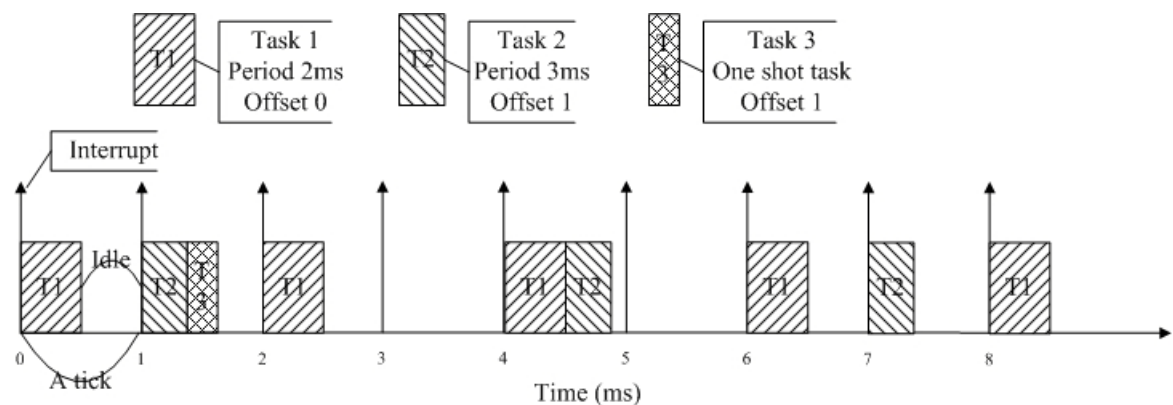


Figure 2-2 TTC Co-operative scheduler described in (Pont 2001)

2.3.2 TTC implementation

A TTC scheduler consists of two parts, i.e. the super loop and the timer interrupt (see Figure 2-3). (Note that the TTC scheduler used in this study, is implemented based on Pont (2001). Also, it has been used in a number of research projects at the Embedded Systems Laboratory at the University of Leicester.)

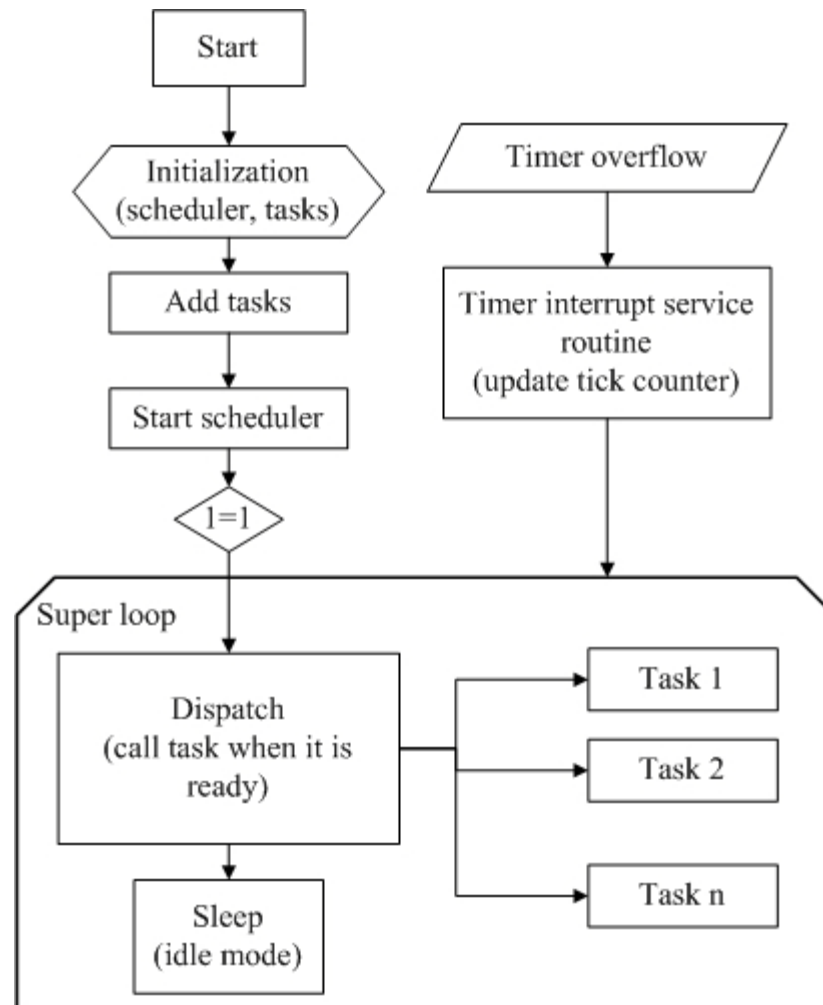


Figure 2-3 TTC flowchart

The super loop makes the system run forever. Inside the super loop, the program will go to dispatch function, firstly. The dispatch function is the core of the scheduler. In each tick, it examines all tasks in the task array, which contains all the predefined task information. Tasks which are ready to run will be executed. After the completion of task executions, the sleep function will send the processor to idle mode.

The timer interrupt is used for creating ticks in a fixed interval. In the beginning of every tick, the timer interrupt will wake up the processor to go through the super loop again.

Note that the scheduler tick interval should be larger than the task durations. A small interval may cause task overrun. (Task overrun will be introduced later in this chapter.)

The pseudo code of the core scheduler is list below.

Main loop

```
{
  Initialize scheduler //set up timer, enable timer interrupt
  Initialize tasks //including peripherals

  Add tasks to the scheduler
  Start scheduler //enable timer

  While (1) //super loop
  {
    Call dispatch function
    Go to sleep
  }
}
```

Figure 2-4 The main loop of the TTC

Dispatch function

```
{
  If < tick counter > 0 >
  {
    Set update_required flag
    Decrease tick counter
  }
  While < update_required flag is set >
  {
    for loop // go through all the tasks
    {
      if <task_n is ready to run> //check task indicator
        run task_n
      Update task_n indicator
    }
    If < tick counter > 0 >
    {
      Set update_required flag
      Decrease tick counter
    }
    Else
    {
      Clear update_required flag
    }
  }
}
```

Figure 2-5 The dispatch function of TTC

Interrupt service routine

```
Scheduler update
{
  Increase tick counter
}
```

```
Reset timer interrupt flag  
}
```

Figure 2-6 The interrupt service routine of TTC

2.3.3 Task and scheduler parameters

From the above section, we understand how the TTC scheduler is implemented. The section gives details on the task and scheduler parameters, and their impact on timing behaviours.

Task Parameters

For each task, its parameters are defined as below.

```
SCH_Add_Task(TASK_NAME, OFFSET, PERIOD);
```

For example:

```
SCH_Add_Task(Task1, 0, 1);  
SCH_Add_Task(Task2, 1, 3);  
SCH_Add_Task(Task3, 2, 3);
```

In the example, Task1 is to be run every tick and Task2 & Task3 is to be run every 3 ticks. Here, the deadline of each task is not its period. We define that the (relative) deadline is the end of the current tick.

The offset defines the task start time, i.e. the offset specifies the start tick of the task. In the example, Task1 is start at first tick, where Task2 is start at the 2nd tick. In case of multiple tasks, start all tasks in the first tick may not be feasible. Therefore, the offset values of the tasks are often set different. In this way, the tasks are allocated in different tick, which can reduce the chances of timing violation.

These parameters are stored in an array in the program memory. The index of this array can represent the order of task. The first added task has an index of 0. Since the dispatch function call the task by index number, i.e. start with 0, and end with the max number of the tasks. The task, which has the index number of 0, has highest priority.

Task sequence

The TTC scheduler schedules task co-operatively. That means each task is run to its completion with no pre-emption. In each tick, the dispatch function checks if the tasks are due to run, which is based on their offset and period.

Note that the tasks that added to the scheduler are assigned with fixed priority. In above example, the first added task Task1 has the highest priority; the last added task Task3 has the lowest task.

Scheduler tick

The scheduler tick is generated by a hardware timer. When the timer overflows (or the timer counter matches a preset value), the timer interrupt service routine will be called.

The tick interval is an important parameter of the scheduler. It has a basic requirement, which is that the tick interval should be large enough to allow all tasks to be completed before the deadline. I.e. inappropriate tick may cause the violations of timing constraints.

Also, the tick interval affects power consumption. Since the dispatch function is called in each tick, a short interval may cause unnecessary CPU load.

However, a long interval may cause the system response slowly, which may be unacceptable. Therefore, the tick interval has to be carefully selected to fit application needs.

Overall, these parameters determine the timing behaviour of the system. In this study, it is assumed that such parameters are known in advance.

2.3.4 Alternative schedulers

Besides the TTC scheduler, there are many other alternative time-triggered schedulers. For example, Time-triggered Rate Monotonic (TTRM) scheduler employs an optimal fixed-priority and pre-emptive scheduling algorithm (Liu, 1973). The priorities are based on task frequency, i.e. the task that has higher occurrence rate (or shorter period) has higher priority.

Another example is the time-triggered hybrid scheduler (TTH) described in (Pont, 2001, 2005, 2007). It is designed for systems that have a long task and short tasks executing in the same tick. It is an adaptive version of TTC scheduler where one of the short tasks is allowed to pre-empt the execution of the long task. The pre-emptive short task has the highest priority for execution. The rest tasks are scheduled co-operatively in the same way of TTC scheduler. As this time-triggered hybrid scheduler can only support one short periodic pre-emptive task; it is much simpler than a fully pre-emptive scheduler. Since it has low resource requirements, it may be useful in resource constrained embedded system design.

2.4 Timing issues

Despite many attractive features, the TTC scheduler has some issues that related to timing.

2.4.1 Task overrun

In TTC, each task is supposed to finish before the end of that tick. If there is a task still running while the next tick occurs, the task is considered to be overrun and may cause problem to the subsequent task executions. When a task executes more than its expected time, we called it task overrun (Cervin et al., 2003; Buttazzo, 2005).

Figure 2-7 shows an example of task overrun. The figure on the top is a TTC scheduler in normal state. The lower figure illustrates the impact of task overrun. At the tick between 2 and 3 ms, both T2's and T3's execution times increase for some reasons. The overrun executions of T2 and T3 lead to the delay of T1 in the subsequent tick. At the tick between 4 and 5 ms, T1 overruns its expected execution time and takes 3.2 ms to complete. This delays the execution of T2 at the same tick. T2 has to wait for the completion of T1 and executes twice sequentially at the tick between 7 and 8 ms.

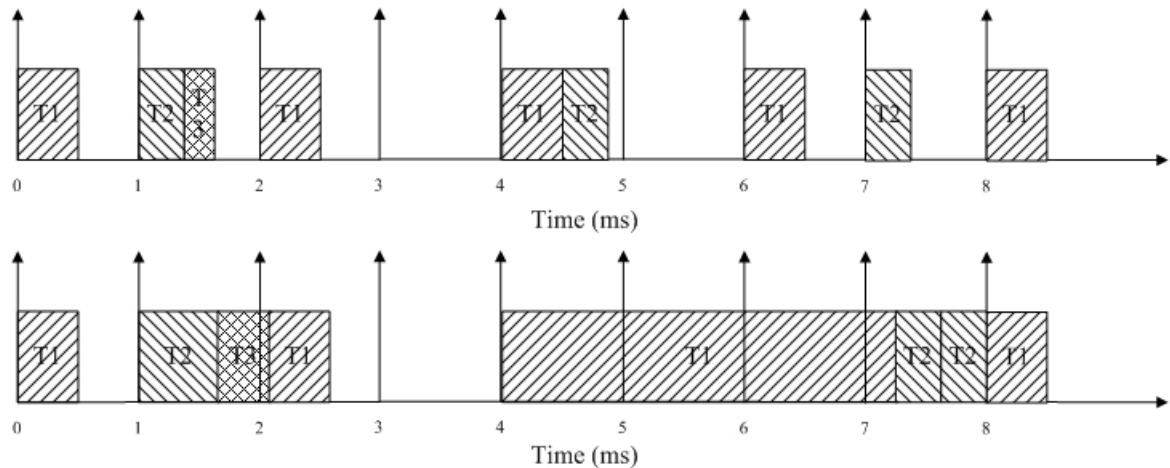


Figure 2-7 Task overrun effect

In this example, we see that a task overrun can cause the subsequent tasks to miss their deadlines. It also can cause the task set to exceed processor available time, which also can be called as transient overload. In situation that the transient overload persists, the system's queue for tasks which are ready to run tends to become longer and task response times tend to increase indefinitely. Since the TTC architecture employs static scheduling and no pre-emption, which means that – in the event of a task overrun – the problem may not even be detected (let alone resolved).

This may have a serious impact on the system behaviour. For systems with 'soft' timing constraints, occasional task overruns can be tolerated. The TTC scheduler may continue working perhaps with performance degradation. For systems with 'hard' timing constraints, task overruns may disrupt the TTC scheduler, which leads to the significant performance deduction or even more serious consequences.

The task overrun could be caused by hardware failures. For example, on-chip hardware failure (e.g. analogue-to-digital conversion), external actuators failure, external sensors failure or power-supply reduction (Pont, 2002). Such unexpected events, may cause tasks behave abnormal.

Also, design errors (e.g. mis-estimation of task WCETs) can cause task overrun. The TT design is based on the assumption on the task behaviour. If such assumptions are incorrect, the system will not perform correctly. The most important assumption is the WCET of tasks. *"Nearly all known real-time scheduling approaches rely on the knowledge*

of WCETs for all tasks of the system", wrote by Gergeleit and Nett (2002). Next section will give a further discuss on this issue.

2.4.2 The WCET issue

The Worst case execution (WCET) issue is an important problem in embedded system design. Since majority scheduling methods are based on the estimates of task WCETs, the prediction of task WCETs is crucial (Kirner, 2003). If a task exceeds its predicted execution time, a domino effect could be generated (if not aborted) on the subsequent tasks. This leads to serious impact on the system behaviour (Buttazzo, 2005).

However, task WCETs are difficult to predict accurately. Many researches are working on this subject (Nett et al., 1996; Domaratsky and Perevozchikov, 2000; Engblom and Ermedahl, 2000; Engblom and Jonsson, 2002; Gergeleit and Nett, 2002; Deverge and Puaut, 2005; Kirner and Puschner, 2007). The methods used to estimate WCET are 1) static analysis and 2) measurement based analysis (Petters, 2007).

The basic idea of static analysis is to analyze the source codes (Lim, 1995; Ferdinand, 1999; Colin, 2000; Engblom, 2001, 2002; Heckmann, 2003, Puschner, 2000). At first, finds the longest execution path of each task by going through the source code instruction by instruction. Then, calculate the WCET of a task using the processor manufacturer provided information about the processing time for each instruction.

However, find the longest execution path is not an easy work. The increasing size of the source code makes the path finding process more challenging (Deverge and Puaut, 2005; Kirner and Puschner, 2007). Nevertheless, the WCET calculation is not straight forward. Because the time for executing a specific instruction, is depends on the state of the processor at the time of executing the instruction (Rochange and Sainrat, 2002; Kirner and Puschner, 2007). Also, modern processors usually use pipelines, caches, branch predictors to increase the performance, which make it difficult to predict (Ferdinand 2001; Deverge and Puaut, 2005).

The measurement based analysis, on the other hand, measures the WCET on a real hardware (or an accurate simulator) (Engblom and Ermedahl, 2000; Deverge and Puaut, 2005). Measuring the execution times requires i) running the code on the actual hardware,

ii) a large set of test input data and iii) a long measurement time (Kirner, 2004; Wenzel, 2005). However, there is no guarantee that the longest execution time is obtained. Thus, a safety margin is often used for ensuring the accuracy of the measured WCET (Vallerio, 2003).

Since the estimation of task WCETs is difficult, the designer not only needs to know the potential risk, but also to understand precisely how the system will behave if an error occurs.

2.4.3 Jitter issues

In many real-time applications, the tasks are required to run at specific time instants. The deviation from the specified timing is defined as jitter (Wavecrest, 2001; Ou et al., 2004). Jitter can have serious impact on systems. For example, in a data acquisition system, the release jitter of the sampling task may cause the sampling result useless (Cottet and David, 1999). Also, in control systems, jitter can greatly degrade the performance by varying the sampling period (Torngren, 1998; Marti et al., 2001).

Jitter can arise from many factors, including oscillator hardware (Schossmaier and Weiss, 1999), electromagnetic interference in a printed-circuit board, or slightly different calculation times for a control algorithm (Kopetz, 1997). Here, we are more interested in the jitters that derived from scheduler and scheduling method.

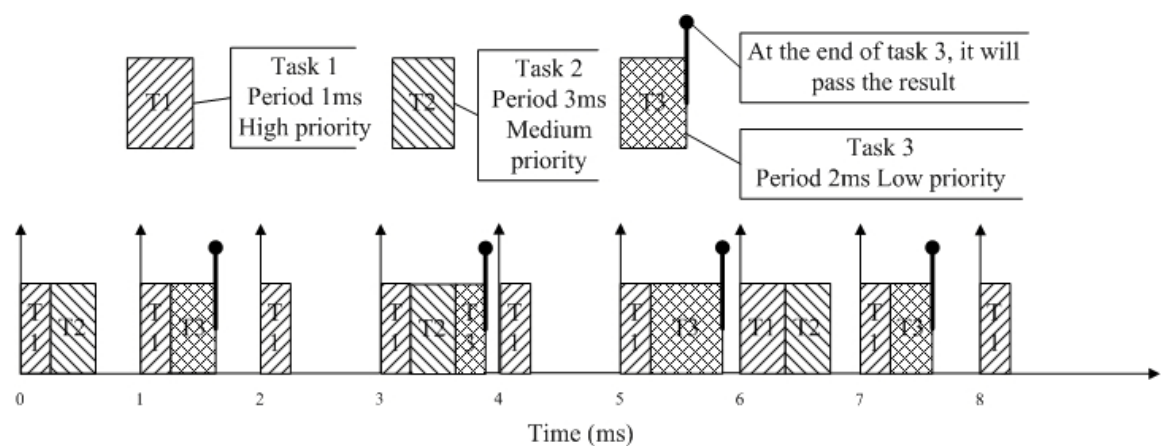


Figure 2-8 Jitter

The first factor is the scheduling, i.e. task sequence. In Figure 2-5, for example, the start time of T3 remains the same in tick 2, 6 and 7 (with the assumption that the execution time of T1 is always the same), since it is executed after T1. However, in tick 4, but T3 is executed after T2 rather than T1. As T2 has its execution time, which is not zero, the start time of T3 has jitter that is caused by the task sequence. Such variation of the task starting time is called release jitter.

Note that the scheduler overhead, which comes from the TTC scheduler, may also cause release jitter. This is mainly caused by context switching (Phatrapornnant and Pont, 2006).

Second factor is the variation of previous task execution time, called execution jitter. Our assumption on T1 that its execution time is always the same, may not hold. For this reason, T3 will also suffer from the start time jitter in tick 2, 6 and 7. It is very common in practice that a task has varied execution time. This may be due to the execution path in if-else, looping, float-point computation, sensor reading, ADC conversion, etc.

The third factor is the variation of a task's own computation time, called finishing jitter. Tasks, which do computation at first and pass the result at the end, usually have such problem. If such tasks have tight jitter constraints, they have to split into two parts (TimeSys, 2002). The first part of the task is used for computation. The second part of the task is used for passing the result. Set these two parts at same period, but give part two high priority. The method may cause the task once cycle delay, but it will have less jitter.

Overall, scheduling method and scheduling setting for each application can affect task release jitter. Also, the variation of the task execution time can affect itself and related task performance.

2.5 Conclusion

This chapter is focused on the timing issues in time-triggered embedded system design. Start with the basic components of real-time embedded systems, i.e. the task model and the scheduler, and end with the discussions of the task overrun problem, the WCET and jitter issues.

3 Monitoring techniques

In Chapter 2, key timing problems in embedded systems were reviewed. This chapter presents the literature review of monitoring techniques.

3.1 Introduction

Runtime monitoring is observation of the target behaviour system to determine whether it complies with the requirements, while the target is executing (Tsai et al, 1990). The monitor could be a software tool or hardware device that operates concurrently with the target (IEEE, 1990). It traces some points of interest and determines if the system consistent with a given specification.

To implement the monitor, there are three basic approaches: hardware-based, software-based and hybrid approaches. Hardware-based monitor is built on hardware which is independent to the target system. Software-based monitor is instrumented in the target software. Hybrid monitor is a combination of the software and hardware approaches. Details of these approaches are discussed in section 3.2-3.4.

3.2 Hardware-based monitoring

Jeffrey J.P. Tsai et al (1990, 1996) proposed a non-invasive architecture to monitoring the execution of distributed real-time system. The target consists of a collection of MC6800 processors and a communication network.

The monitoring system uses additional hardware to build a set of monitoring nodes. The monitoring nodes are connected to the target system to snoop the data, address, and control buses of the target node for specified conditions. Then, the monitoring system processes the information based on predetermined rules, which indicates the state of target system.

The author claimed that the monitoring system does not use any of resources of the target, since all the information is retrieved from the internal buses. However, it is specialised for MC6800 architecture and requires sophisticated hardware and (Dodd and Ravishankar, 1992; Haban and Wybraniec, 1990).

In general, hardware-based monitoring (Plattner, 1984; Liu and Parthasarathi, 1989) often uses dedicated hardware to trace program activity. It observes the data, address, or control bus signals and matches the signals to predefined conditions to obtain the target status.

There are two significant advantages of this approach. Firstly, this approach is completely non-intrusive. It does not add to the complexity of the software on the monitored system. As a result, hardware-based solutions are considered by many people to be “optimal” (Haban and Wybranietz, 1990; Tsai et al., 1990; Dodd and Ravishankar, 1992). Secondly, it is able to detect events which software cannot. For example, it can assert an interrupt occurred by observe the interrupt line.

But there also has several disadvantages. Firstly, as electronics designs become smaller and more integrated access to relevant signal lines becomes increasingly challenging (Thane, 2000; Gergeleit, 2001; Shobaki, 2004). The processor architecture, such as pipeline, memory management unit, instruction and data caches, branch prediction, become more complex. Secondly, hardware monitor are difficult to port from one processor architecture to another, since it is highly dependent on the low level information.

3.3 Software-based monitoring

Aloysius K. Mok and Guangtian Liu (1997) proposed a Java Runtime Timing-Constraint Monitor for distributed real-time system. It is implemented in Java language, so the target needs to support Java.

The Monitor is using in a specification language based on Real Time Logic (RTL) to define timing properties and events. The method is to insert event triggering method calls in the Java program where the event instances occur. This event triggering method calls works like sensors. So when an event is triggered at runtime, the sensor sends a message to the monitor with the occurrence time of the event instance and the event name. The monitor receives the messages and stores them into a queue where the earliest message is at the first. Then, the monitor checks the events with the predefined timing properties later. If a constraint is violated, the monitor will inform the user and target to invoke recovery actions.

The sensors need to reside in the target program, where the monitor can be run inside the target or on a standalone monitoring machine. Implement the monitor on a standalone machine, also can be considered as hybrid approach.

Software-based monitoring approach generally inserts the monitor into the software of target system (Mansouri-samani, 1995; Shobaki, 2004). The instrumentation code, also called software sensor, is inserted at the point of interest to gather information. There are four common instrumentation methods, i.e. binary code modification, assembly code modification, object code modification and source code modification. Source code modification is most popular one. This process could be applied by a programmer, or by an automatic source code instrumentation tool.

This method is considered to be simple and flexible (Obermaisser, 2001). However, the monitor is using target resource and increase software complexity, which will cause inevitably interference. This is also called “probe effect”. There is a trade-off between minimal interference and recording sufficient information (Obermaisser, 2001).

Researches have been done to minimize the interference (Thane, 2000; Mahrenholz, 2001).

3.4 Hybrid monitoring

Hybrid monitoring approach combines the low interference of hardware monitor and the flexibility of software monitor (Mansouri-Samani, 1995; Shobaki, 2004). A hybrid monitor requires an internal software part, which signalling events to the external world. Compare to software monitor is very much simpler. It also requires an external hardware part, which does the rest of monitoring tasks, including event detection and time stamping, data analysis, and record, ect.

3.4.1 FPGA based hybrid monitor

Cadamuro Junior and Douglas P. B. Renaux (2008) present a hybrid-based monitor for embedded systems.

In the target, three types of instrumentation codes are added. i.e. structural, contextual and speculative. They are used to trace program flow (including procedure execution, decision points, and loop construction), variables and third party library functions (with

no source code available) respectively. An instrumentation tool is developed to generate the source code with these instrumentation codes.

The physical connection between the target and the monitor is measurement port, which consists of 16 digital data lines, 1 data valid line and a reference ground signal. The data lines are preferred to be implemented on a 16-bit memory mapped position, i.e. a latch between processor data bus and the measurement port connector is required. Alternatively, GPIO pins may be used.

The external hardware monitor is built on a FPGA (Field-Programmable Gate Array) board. It collects the events and provides it with a timestamp. The timestamp is derived from a 32-bit time, with 20ns resolution. Note that event detection and time stamping is implemented in the hardware state machine. The data processing is implemented in software processor inside the FPGA.

This hybrid monitoring system can extract a large amount of information with adding certain instrumentation codes. To support this, the target needs to have enough resource (e.g. memory, CPU time, I/O ports, etc).

3.4.2 Linux scheduling monitor

Zdenek Slanina and Vilem Srovnal (2008) proposed a process scheduling monitor for embedded systems, which are based on Linux platform. This target is i386 embedded PC and Motorola based machines.

The monitoring system based on hybrid approach. It inserts a “watch module”, as a part of Linux kernel, into a target machine. This module is target to the scheduler and the interrupt source for measuring the scheduling time, context switch and latency of processes. A “watch out” module, which also is inserted in the target, sends the “watch” data to the monitor every second. The connection between the target and the monitor is Ethernet, in order to ensure a fast transmission. The monitor analyzes the received data on a separate machine.

This monitoring system is specified for Linux kernel based application. It add additional modules to the target, requires Ethernet for communication. This method can only be used in limit applications.

3.5 Other related works

The section introduces some temporal fault detection methods.

3.5.1 Scheduler watchdog

Scheduler Watchdog (Pont, 2002) is based on the common watchdog timer. The processor needs to have an on-chip watchdog timer, which is a common block that built-in modern microprocessors.

The watchdog timer will be set to overflow at a period greater than the tick interval. Note the overflow period is depending on the system characteristics. A task, which could update the watchdog timer shortly before it overflows, will be added to the scheduler. So in normal state, the timer will never overflow.

If the watchdog timer overflow occurred, it means the scheduling has disrupted. The monitored processor need to be recovered. The recovery strategies could be simple reset, Fail-Silent recovery or Limp-Home recovery. A risk assessment should be done, before choosing the recovery method.

Overall, scheduler watchdog provides a useful “safety net” in the event that problems in the system disrupt the scheduling. But it has some weaknesses. Firstly, the long timeout period of the Watchdogs is not suitable for some designs. When system crashed, it will continue operate until the watchdog reset. At the time system may behave unpredictable and potentially hazardous. Secondly, the system reliability will reduce, due to improper design. Especially, the recovery mechanisms could cause the system to repeatedly reset itself.

3.5.2 Task guardian

The Task guardian (Hughes, 2004, 2007) does not require any additional hardware support. It can be consider as an updated version of TTCS.

The changes are mainly in timer interrupt update function. The original TTCS update ISR function only identifies tasks, which are due to run in the following tick. The one

with task guardian will detect the task overrun, firstly. If the overrun occurred, it will end the task. So the scheduler will continue normal operation.

As the overrun tasks are shut down directly, a critical system might operate incorrectly or even out of control. To solve this problem, task guardian provides backup tasks. Instead of ending the overrun task, it will run the backup task. A good design of backup task will reduce the impact. Alternatively, if there is no backup task then the task guardian will change the overrunning task to the lowest priority.

Overall, task guardian could immediately handle the overrun, which is quite useful. But the critical systems performance will depend on the design of backup tasks.

3.5.3 Execution timer/clock

Execution-time clock (Harbour, 2003, Puente, 2003) is based on a library-level package: this is intended to detect and handle task overrun for Ada Kernels. It has an execution-time clock function to check the task execution. When task starts, its timer keeps counting down until it exceeds its estimated time. If the task exceeds its estimated WCET, it can be detected through the `Timer_Has_Expired` operation and `Timer_Expired` entry call. Execution-timer servers (Burn, 2006) also have been developed to support for task execution management.

3.6 Agent

3.6.1 Agent definition

“An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives” defined by Jennings (2000). It is a problem solving entities, which embedded in a particular environment. The agent receives the sensory input from the environment, and outputs some actions to achieve the specific purpose.

Agents have the following properties (Wooldridge and Jennings, 1995; Wooldridge, 1997; Wooldridge, 2002):

- **Autonomy:** agents control their internal states, and make their own decisions without the direct intervention of humans or others.

- Social ability: agents interact with other agents (or possibly human) to engage in social activities.
- Reactivity: agents perceive their embedded environment, and respond in a timely fashion to any changes that occur in the environment.
- Pro-activeness: agents do not simply act in response to their environment; they are able to exhibit goal-directed behaviour by taking the initiative.

Other researchers also defined the agent from their own applications. Franklin and Graesser (1996) have analysed many definitions of agents. Weiss (1999) defines agent as *“An autonomous, reactive, pro-active computer system, typically with a central locus of control, that is at least able to communicate with other agents via some kind of communication language.”*

Agent-based systems may contain a single agent, or multiple agents, which is also called Multi-Agent Systems (MAS). Multi-agent system (MAS) is composed of several independent agents, which typically interact with each other to solve complex problems. Thus, the MAS have the potential of better capabilities than a single agent.

3.6.2 The role of agents

Role, in general terms, is related to an application scenario. For example, information agents have the role of managing, or collating the vast amount of information in wide area networks (like the Internet) (Ahmad, Ahmad, et al. 2008).

From the structure level, role is “the functional or social part”, which an agent plays in a (joint) process like problem solving, planning, or learning” (Weiss, 1999). From the implementation level, a role can be defined as “a set of capabilities (i.e. a set of actions that an agent playing such role can perform to achieve its task) and an expected behaviour (i.e. a set of events that an agent is expected to manage in order to “behave” as requested by the role it plays)” (Cabri, Leonardi, et al. 2003). (This concept is used in Behavioural Roles for Agent Interactions framework.)

3.6.3 Agent architectures

To develop an agent, agent architectures have been used to help the hardware/software implementation. Agent Architecture is a particular arrangement of data structures, algorithms, and control flows, which drive the agent behaviour (Weiss, 1999).

Agent architectures can be characterized by the nature of their decision making. Example types of agent architecture include logical-based architectures (in which decision making is achieved via logical deduction), reactive architectures (in which decision making is achieved via simple mapping from perception to action), belief-desire-intention architectures (in which decision making is viewed as practical reasoning of the type that we perform every day in furtherance of our goals), and layered architectures (in which decision making is realized via the interaction of a number of task accomplishing layers) (Weiss, 1999).

3.7 Discussion

Our monitoring system is aim to observe the timing behaviour, and to detect temporal faults of resource constrained real-time embedded systems. Such target system are implemented in electronics and sealed in small packages which limit the access from the external world. The non-invasive/ low interference feature is very attractive. But the decreasing visibility of the microcontroller makes the hardware monitoring difficult to observe. The accessible information is limited. Therefore, hardware monitoring approach is not used.

The software approach, in other hand, can be easily inserted to the target software, which allows the monitor access to low level of execution information. However, the monitor needs to share resource with the target. Only limit the CPU utilisation can be assigned to the monitor. The available memory for monitor also may limit its functionality. The stringent real-time requirements of embedded systems make the software monitoring difficult to operation. Even if it is possible, adding a monitor to the target system may alter the system behaviour unintentionally. Therefore, software approach is not selected.

The hybrid approach combines the benefits of the above two approaches. By inserting miniaturized software sensors into the target, the information can be signal out to an

external hardware. The software sensor requires minimized resource from target to support. Because the low level of intrusion, the sensors could be permanently reside in the processor.

The external hardware is independent to the target. Thus, the likelihood that the monitor will cause disruption to the operation of the target, and the likelihood that both monitor and target will fail simultaneously are reduced.

The FPGA based hybrid monitor and Linux scheduling monitor shows that hybrid monitoring is capable to deliver detailed execution information. Thus, our monitoring system employs the hybrid approach. Also, we employs agent concept in our approach to improve its properties.

3.8 Conclusion

This chapter presents a review of existing monitoring techniques. A general discussion about the monitoring approaches was performed. Next, some existing temporal fault detection tools are presented. Finally, the reason of selecting hybrid monitoring in this study is discussed.

4 Scheduler agent¹

The previous chapter focused on the existing runtime monitoring methods. In this chapter, we present our technique, called the scheduler agent, to monitor the temporal behaviour of time-triggered embedded systems.

4.1 Introduction

SA is a hybrid monitoring technique to monitor the temporal behaviour of embedded systems which employ TTC scheduler. Like other hybrid monitoring technique, SA consists of two parts: an internal monitor (IM) and an external analyzer (EA). Both of the IM and the EA have to work cooperatively in order to obtain information from the target system. The communication between them relies on a coding scheme and a physical data bus which connects the EA and the target system together. The abstraction of the SA architecture is shown in Figure 4-1.

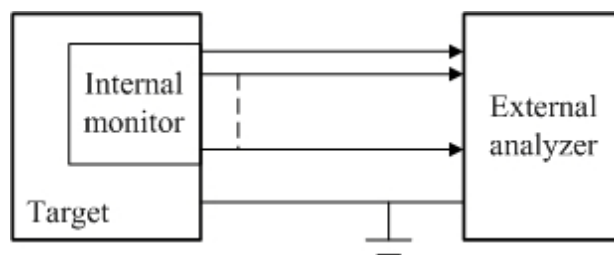


Figure 4-1 The abstraction of the SA architecture

4.2 Internal monitor

The IM is responsible for monitoring the activities of the scheduler and tasks of the target. In order to achieve this, a set of software sensors are inserted at the points of interest in the target software. Information obtained by the IM is then encoded and sent to the EA for further analysis. To be more specific, our interest is to gather information

¹ Part of work has been published in Gendy, A.K., Dong, L. and Pont, M.J., 2007. Improving the performance of time-triggered embedded systems by means of a scheduler agent, Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference.

about 1) the scheduler tick interval, 2) the execution time of each task, and 3) the execution order of the tasks at each scheduler tick. Thus, the software sensors have to be inserted at the start and the end of each task, and at the end of the scheduler interrupt service routine (ISR).

It is important to note that the IM has to keep a balance between sufficient information and minimized interference; since the IM resides in the target software, and hence it consumes the target's resource and CPU time. Therefore, the sensors not only have to be small in term of the code size, but they also have to be capable to deliver required information in time. The details of the sensors are given in the following sections and the next chapter.

We must emphasize that the SA does not attempt to monitor task output, but the focus is solely on timing and scheduling related issues, i.e. faults like computation errors, logical errors are not considered.

4.3 External analyzer

The primary role of the EA is to analyze information gathered by the IM and to report the analytical result to the user.

The EA consists of the four different components:

- 1) Data receiver – it receives data from the IM and puts a timestamp on each received data.
- 2) Event recognizer – it identifies an event from the received data (time-stamped).
- 3) Run-time analyzer – it analyzes the processed data from the event recognizer and determines if there is any deviation from the requirements.
- 4) Reporter – it stores the data and reports the results to the user.

4.3.1 Data receiver

The data receiver receives run-time data of the target system through the communication link and puts a time stamp on the data. In this study, we developed two different types of data receiver – passive data receiver (PDR) and active data receiver (ADR) which are employed in different versions of SA.

The PDR receives data only if new data is arrived. The activation of the PDR is triggered from a signal generated from the IM. If there is no trigger signal present, the EA will continue to perform the current operation. The received data is stored in the “raw data buffer”, together with a timestamp. The data is then processed by the event recognizer.

The ADR works as a data sampler which will be active in a fixed time period. When it is active, it continuously tracks for the changes of the communication port that is connected to the target system at a certain sampling rate. The ADR is activated at specific points in time. Unlike the PDR, the raw data retrieved by the ADR will be directly processed by the event recognizer.

4.3.2 Event recognizer

The event recognizer acts like an interpreter which translates the raw data into a recognizable data format according to a pre-defined coding scheme. It contains a counter to track the current tick number of the target system. The event recognizer converts the counter value together with the raw data and the timestamp provided by the data receiver into an “analyzer comprehensible” format. For example, {Tick number, tick time; new tick start time; Task ID, Task end time; ... ; Task ID, Task end time }. The formatted data will be analyzed by the run-time analyzer subsequently.

4.3.3 Run-time analyzer

The run-time analyzer analyzes the formatted data according to the target system specification. It must be capable to perform the following functions:

- 1) Calculate the scheduler tick interval and detect the scheduler tick error.
- 2) Calculate the execution time of each task and detect if the task exceeds its WCET.
- 3) Detect a missing deadline.
- 4) Detect the task execution sequence error.

The EA calculates the scheduler tick interval of the target system by subtracting the previous and the current tick times. The actual execution time of a task is derived from the difference between the ending times of the previous and the current tasks. By comparing the actual execution time with the target specification, we know whether the

task exceeds the WCET and misses the deadline. The expected task sequence is calculated at run time, based on the specification of the target system. To determine which task should be run at the current tick, we apply all tasks to the formula:

$$(\text{Current_tick} - \text{Offset}_i) \% \text{Period}_i = \text{Run}_i$$

For a task i , if Run_i equal to 0, then task i should run in the current tick.

4.3.4 Reporter

The reporter sends the results that produced by the run-time analyzer to the user. This can be done by applying a serial link to display the result in a host PC. The reporter also can store the result in the memory (or EEPROM) in which it works like a data logging system. This allows the user to retrieve the result later on for further analysis.

4.4 Communication between the IM and the EA

The communication between the IM and the EA plays an important role in the monitoring process. It requires for delivering sampled data in time with low interference to the target system, and allowing sampled data easily to be accessed.

Since a microprocessor is sealed in a small package, this means that the internal memory or bus is difficult to be accessed. Fortunately, modern microcontrollers provide interfaces such as serial communication interfaces and general purpose I/O (GPIO) interfaces, to allow the communication of external devices. As such standard communication interfaces are easy to implement, one of these interfaces will be used in the communication between the IM and the EA. The following sections discuss two interfaces– Serial communication interfaces and GPIO interfaces – which are possible to be used for the implementation of the SA.

4.4.1 Serial communication interfaces

Modern microprocessors usually contain one or several serial communication interfaces, such as UART (Universal Asynchronous Receiver/Transmitter), USART (Universal Synchronous Asynchronous Receiver/Transmitter), SPI (Serial Peripheral Interface), I2C

(Inter-Integrated Circuit), CAN (Controller area network), USB (Universal Serial Bus), etc.

Using a serial interface for the implementation of the SA, the internal status of the target can be accessed easily. However, the IM needs to be tailored in order to pack the data collected by the software sensors into a message and transmit to the EA over the serial link. Thus, additional software is needed for supporting such communication activities. There is also a time stamp issue that needs to be considered. Putting a timestamp on a message can be done in two ways:

1. A message which contains raw data only is sent by the IM immediately after an event happened. The EA receives the message and put a timestamp to the message using its arrival time.
2. The IM takes a timestamp after an event happened. It then sends a message that contains raw data and a time stamp to the EA. Note that the message does not require to be sent immediately, but can be sent when the target system is not busy.

In the first method, special attention is required on the implementation of the communication procedure; since the message requires to be sent to the EA immediately in order to have an accurate timestamp. Clearly, it is not suitable for methods which write the message to a buffer and then transmit the message only when the bus is ready. Additionally, the EA should consider a time compensation that may be caused by the transmission latency.

The second method is to get time stamp from the target system clock, i.e. the raw data is attached with the system timer value. Therefore, unlike the first method, the message does not require to be sent immediately; since a slightly delay of the transmission will not cause a significant problem. However, since the time stamp is based on the target system timer, it is difficult to detect errors when the timer is incorrect. Also, the size of the message is increased because the timer value is included.

The advantage for using a serial communication for the SA is that applications which have already used a serial communication can add the SA into system. However, if the system already has a heavy-load serial communication, the SA may not be appropriate.

4.4.2 General purpose I/O interfaces

GPIO is the most common interface which can be found in any modern microcontrollers. Like a serial communication interface, it allows external devices to communicate with microcontrollers. A GPIO port contains a number of pins which accept digital signals (i.e. either 0 or 1) only. It is very simple and easy to implement.

Using a GPIO interface to implement the SA is simple, because it does not require any additional software to support the communication. The IM in the target system can update the output of the GPIO pins directly according to data collected by the software sensors. This allows the information of the target system to be sent to the EA much quicker than the serial interface.

The initial idea for this study is to assign each task in the target system with a GPIO pin (which we call it as a “task pin”). An example is shown in the middle of Figure 4-2. The task pin is set to high during the task execution and set to low when the task finishes. The rising edge and the falling edge of a signal can therefore be used to respectively identify the start and the end points of the task.

Compared with using the serial communication for SA, it consumes much less resource from the target system. Hence, it has lower interference to the target behaviour. .

Therefore, GPIO interface was employed for the implementation of the SA. However, for systems with a large number of tasks or small numbers of I/O pins, it may not have enough number of pins to assign to all the tasks. Therefore, encoding technique must be employed to make this method feasible for this situation.

4.5 Using I/O pins with encoding for communication

Instead of using a pin to represent a task, a GPIO port is used for representing all tasks. We define a task port which can be (a part of) a normal GPIO port or a combination of a set of pins in different GPIO ports.

We assign each task to a unique task ID (an integer value). Therefore, the task port can output the task ID into a binary value to indicate the task is running. We define that when no task is running (idle or sleep mode), the output is 0; when a task n is running, the output display the ID of task n , where n is an integer.

As mentioned in Chapter 2, each task has its own task index (from 0 to N). The task ID is defined as (task index + 1), since 0 is considered as idle/sleep mode. For example, in Figure 4-2, there are three tasks running in the target system. Only two task pins are required to show which task is currently running. These two pins will be set to 01 (a binary number of 1) if the task with an ID equals to 1 is running. If the task with an ID equals to 2, the pins will be set to 10 (a binary number of 2) and so on. In this way, n task pins can represent $(2^n - 1)$ tasks which reduces the number of pins required for the communication.

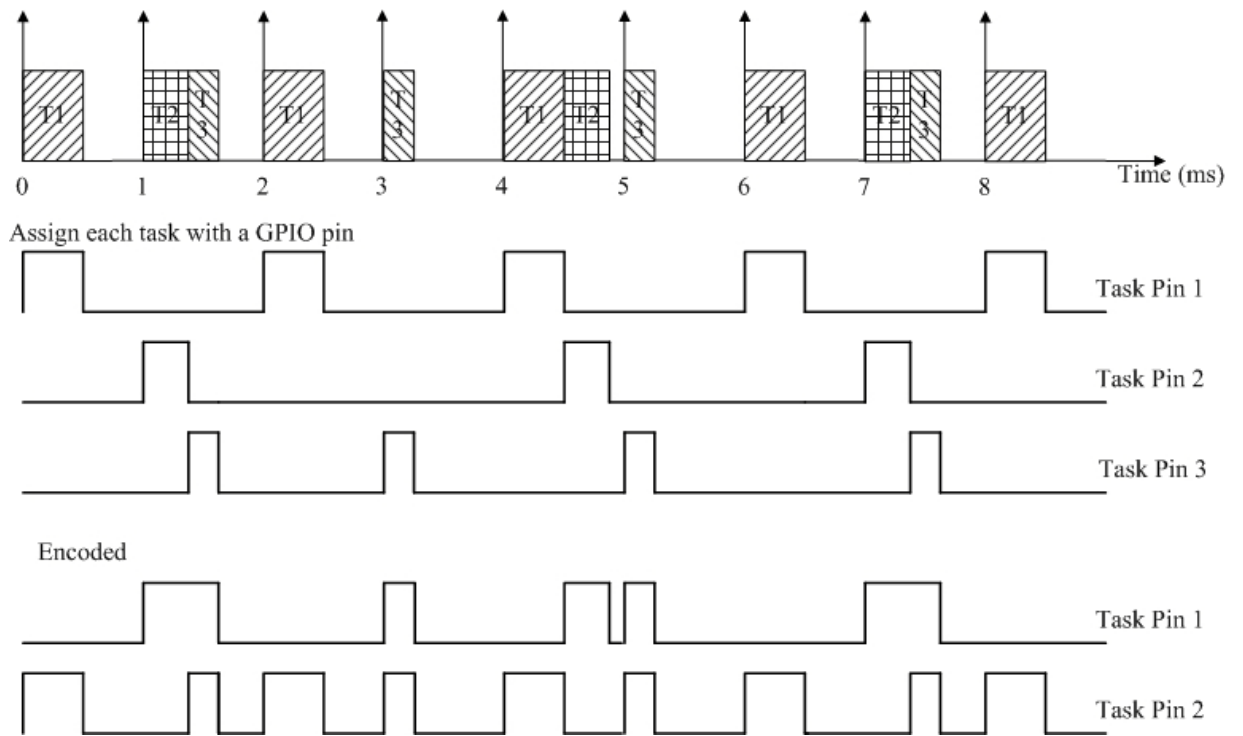


Figure 4-2 GPIO output

The encoding technique introduced in this section is simple and can be extended to represent more information of the target status. However, it may require more resource and computation time which will cause interference to the target behaviour. The balance between more functionalities and low interference has to be considered. Due to the time constraints of this study, the extension of this encoding technique has not been explored.

4.6 Conclusion

The SA consists of the IM and the EA, which are connected together using a GPIO interface, since a GPIO interface is much simpler, quicker and less interference than a

serial communication interface. However, an encoding technique is required since modern microcontrollers may not have enough GPIO port pins to represent all tasks in the target system. A simple and effective encoding technique has been introduced in the chapter to address this shortcoming. In the next chapter, two versions of the SA – Passive SA (PSA) and Active SA (ASA) – are introduced. We also look at the implementation of both SA in details.

5 Implementation of the SA

The previous chapter focuses on the design of the SA. This chapter presents the implementation of the PSA and ASA.

5.1 Introduction

Passive Scheduler Agent (PSA) and Active Scheduler Agent (ASA) are the two forms of scheduler agent. The primary purpose of PSA and ASA is to monitor the temporal behaviour of (and detect temporal errors in) a TTC embedded system in runtime. The difference between them is that PSA employs a passive data receiver, but the ASA employs an active data receiver. The details can be found in Chapter 4. However the steps of their implementation are the same which involves the following steps:

- 1) Obtain the specification of the target.
- 2) Based on the specification, select a set of GPIO pins required for the communication between the target and the SA.
- 3) Implement the IM on the target.
- 4) Implement the EA using a microcontroller which is independent to the target.

These four implementation steps are elucidated in the following sections.

5.2 Target specification

The design specification of the target provides valuable information for system monitoring. In Chapter 2, we explained that the TTC scheduler schedules task executions according to the specification. Table 5-1 lists some of the specification parameters which require for the SA.

Table 5-1 The specification parameters which are required by the SA

The parameters of the scheduler:

- The Max number of the tasks
- Tick interval

The parameters of a task

- Task Index
- Offset
- Period

- WCET
- BCET (if available)

By using these parameters, the SA can predict the behaviours and determine the correctness of the system (see Table 5-2).

Table 5-2 The abilities of the SA

For the scheduler:

- Evaluate the actual timer interrupt arrival time?
- Detect the actual task execution sequence

For a task:

- Evaluate the activation (or release) time
- Evaluate the actual execution time (hence, the SA can detect a task overrun)

5.3 Implementation of the PSA

The implementation of the PSA consists of three parts: 1) the implementation of the communication interface, 2) the implementation of IM and 3) the implementation of the EA.

5.3.1 The implementation of the communication interface

As discussed in last chapter, the IM needs signal the EA for any change detected in the target system. In order to do this, EA must have a dedicated pin (task interrupt pin) for this purpose. This pin is connecting to an external interrupt source pin of the EA. For example, when a task is completed, the IM will generate a rising-edge signal to trigger an interrupt in the EA. This allows the EA to respond the change rapidly.

Apart from the task interrupt pin, additional pins are required to signal the EA about the information of the change in the target. We define a scheduler tick pin to indicate the status of the scheduler. This pin will be toggled at the arrival of a scheduler tick. Therefore, the rising-edge and the falling-edge of the signal in this pin can be used to identify the arrival of a new tick. We also define an overrun pin to indicate the occurrence of a task overrun. This pin will be set to high, when an overrun happens, and it will be reset in the arrival of the next tick after the overrun task is completed.

Together with the task port, these three pins provide all the information about the target system needed by the EA. Table 5-3 shows the functions of these pins. In the rest of this thesis, we refer these pins as communication pins.

Table 5-3 Functions of the communication pins

Name	Number of pins	Functions
Task port	ln(Max task number +1)	Clear the task port when no task is running. Set the task port to the ID which equals to the ID of the running task.
Task interrupt pin	1	Generate a rising-edge signal when there is any change in the target system.
Scheduler tick pin	1	Toggle the pin at the arrival of a new tick.
Overrun pin	1	Clear this pin in the normal state if it is set. Set it to high when overrun occurs.
Common ground	0	Connect the ground of the target and the EA

Note that using the encoding method mentioned in Chapter 4, the selected GPIO pins must be at the same GPIO port, and they must be adjacent to each others. For example, using the pins 0, 1, 2 and 3 in Port 1 as the task port to output a decimal value of 3. It must mask the rest of the pins in Port 1 and set the port register to 0x03. Figure 5-1 shows the code fragment for this example. If the task port pins are not adjacent (or at different ports), each pin has to be assigned individually. Figure 5-2 shows an example of this problem.

```
IOCLR1 |= 0x0F; //clear the Port 1, pin 0, 1, 2, 3
IOSET1 |= 0x03; //set the value to 3
```

Figure 5-1 Code fragment to set the GPIO port

```
Value = 3; //assign 3 to the variable 'Value'

IOCLR0 |= 0x05; //clear the Port 0, pin 1, 3
//assign lsb (least significant bit, bit 0)
if ((Value & 0x01) == 0x01) //Is lsb set to 1?
    IOSET0 |= 0x02; //set the bit 0
//assign bit 1
if ((Value & 0x02) == 0x02) //Is bit 2 set to 1?
    IOSET0 |= 0x04; //set the bit 2

IOCLR1 |= 0x48; //clear the Port 1, pin 4, 7
//assign bit 3
```

```

if ((Value &0x04) == 0x04) //Is bit 3 set to 1 ?
    IOSET1 |= 0x08; //set the bit 3
//assign msb (most significant bit, bit 4)
if ((Value &0x08) == 0x08) //Is bit 4 set to 1 ?
    IOCLR0 |= 0x40; // set the bit 4

```

Figure 5-2 Code fragment to demonstrate the issues of task port pins selection

5.3.2 The implementation of IM

The implementation of the IM involves inserting monitor codes into the software of the target. This includes the codes for the initialisation of the communication pins and the software sensors. The initialisation of the communication pins can be done before the scheduler starts. Since the data flows from the IM to the EA, the communication pins in the IM are initialized as outputs. In the EA, they are initialized as inputs.

The software sensors in the IM are used to control specific communication pins when they are executed. The software sensor for the scheduler tick pin is inserted into the tick timer ISR to sense the arrival of a scheduler tick. This is shown in Figure 5-3.

```

Interrupt service routine
Scheduler update
{
    Increase tick counter
    Toggle scheduler tick pin status
    Clear the task interrupt pin
    Set the task interrupt pin
    Reset timer interrupt flag
}

```

Figure 5-3 Software sensors in the ISR

The software sensor for controlling the task port is inserted in the dispatch function to sense the task runtime information (see Figure 5-4). The sensor can be inserted in each task, but it will increase the target code size significantly.

The software sensor for controlling the overrun pin is also inserted in the dispatch function. It will be set to high if a task overrun occurs. The deployment of this sensor is also shown in Figure 5-4.

The software sensor for controlling the task interrupt pin is inserted in the ISR and the dispatch function (See Figure 5-3 and Figure 5-4).

The simplified flowchart of the TTC scheduler with the IM is shown as Figure 5-5, where the codes for the IM are in bold format.

```

Dispatch function
{
  If < tick counter > 0 >
  {
    Set update_required flag
    Decrease tick counter
  }
  While < update_required flag is set >
  {
    for loop // go through all the tasks
    {
      if <task_n is ready to run> //check task indicator
      {
        Set the task port = task ID
        run task_n
        Clear the task interrupt pin
        Set the task interrupt pin
        Update task_n indicator
      }
      Clear the task port
    }
    If < tick counter > 0 >
    {
      Set update_required flag
      Decrease tick counter
      Set the overrun pin
      Clear the task interrupt pin
      Set the task interrupt pin
    }
    Else
    {
      Clear update_required flag
      Clear the overrun pin
    }
  }
}

```

Figure 5-4 Software sensors in the dispatch function

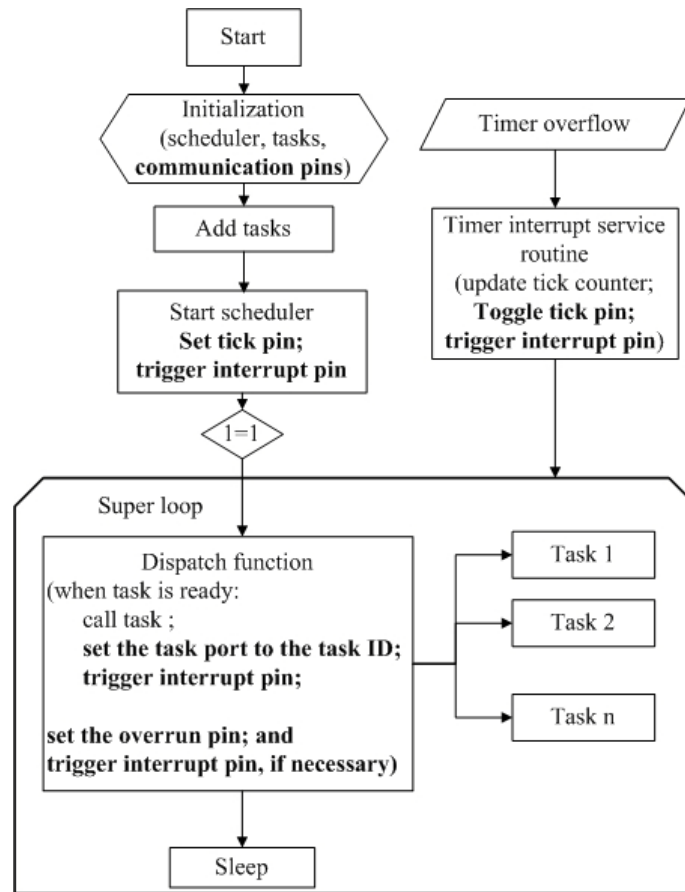


Figure 5-5 Flowchart of TTC scheduler with IM

5.3.3 EA implementation

The hardware platform chosen for the EA was an NXP LPC2129 microcontroller running on a small evaluation board. The LPC2129 is based on an ARM7TDMI core and is typical of modern (low cost) embedded processors. The abstract flowchart diagram of the EA is shown in Figure 5-6.

The passive data receiver reads the communication pins when it is triggered by an external event interrupt. It records the state of the pins with a timestamp that derived from the internal 32-bit system timer. These data are written to a raw data buffer. The size of the raw data buffer is dependent on the number of tasks.

The event recognizer periodically checks the raw data buffer and processes the data into a form of { Tick number, tick time; new tick start time; Task ID, Task end time; ... ; Task ID, Task end time }. After the information of a tick is collected, it sets a flag to inform the runtime analyzer to start analyzing.

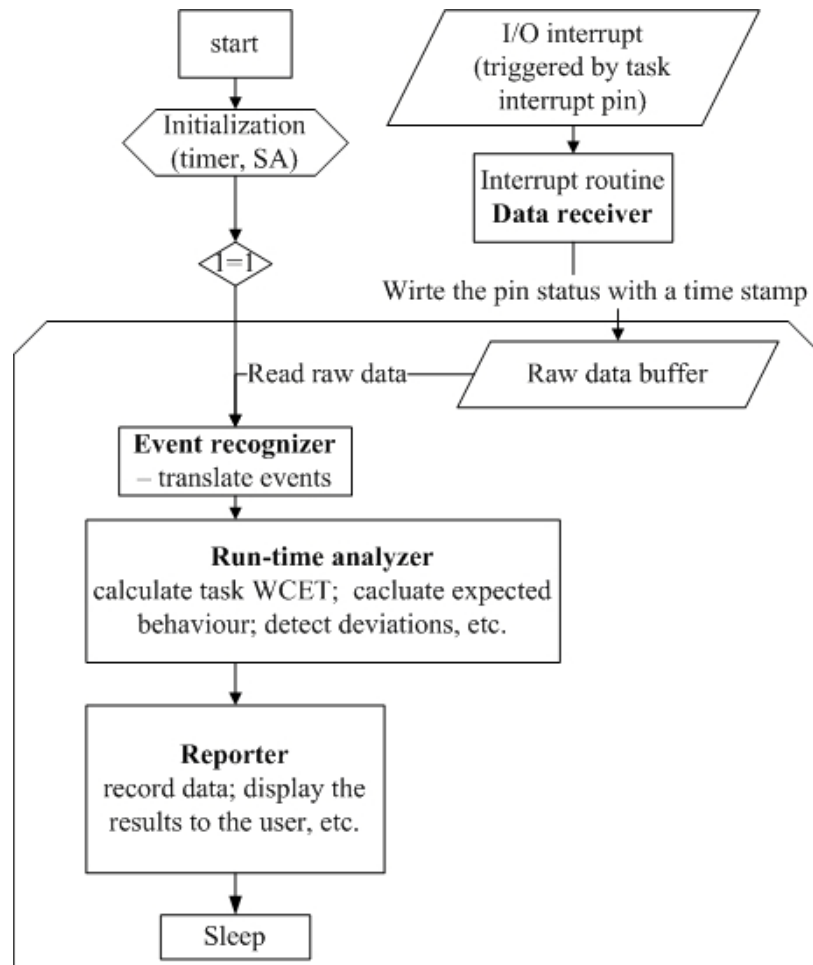


Figure 5-6 Abstract flowchart for the EA of PSA

The runtime analyzer calculates the expected task execution sequence in each scheduler tick based on the target specification. The expected execution sequence is compared actual task execution sequence to verify for the correctness. The analyzer also works out the actual execution time of each task in each scheduler tick, and compares them with the estimated WCETs to check for any mismatch. If any overrun task is identified, an error message will be sent to the user by the reporter. For test proposes, a terminal emulator (e.g. HyperTerminal) is used on a PC to display the error message and task information. Note that the actual WCET and BCET of tasks are recorded for late use.

5.4 Implementation of the ASA

In the PSA design, the data receiver is driven by the interrupt that is generated from the target system. Alternatively, the data receiver can retrieve data from the target system by sampling the communication pins, i.e. active data receiver.

5.4.1 Sampling the communication pins

In applications which requires for sampling data, the sampling task is invoked when it is required, which can be periodically or at specific points in time. The ASA in this study employs the later approach. Sampling key points are defined in runtime to activate the data receiver to retrieve target information. This provides an efficient monitoring solution and is compatible with TT designs. Using this approach, the EA can spend more CPU time to process the data and the rest of the operations, such as transmitting the result to the host computer. Figure 5-7 illustrates the operation of this approach.

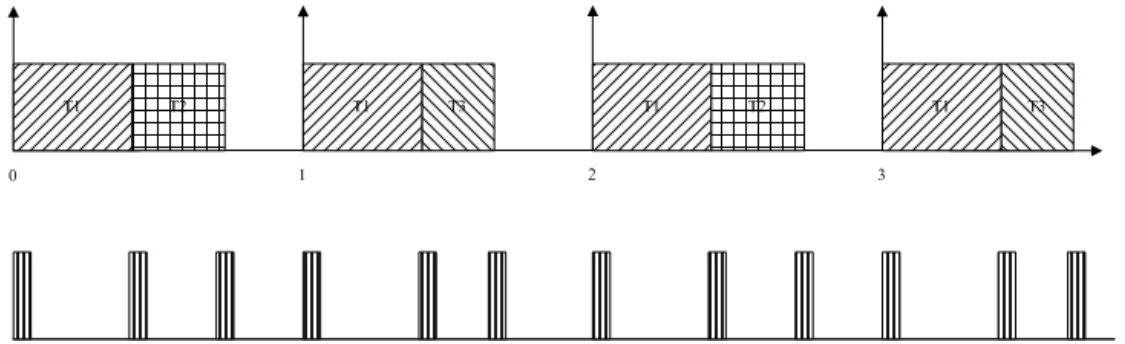


Figure 5-7 Sampling the communication pins at key points

5.4.2 Sampling key points

As the specification of the TTC scheduler is provided, the EA can estimate the sampling key points using the following steps:

- 1) The start of the first task of a scheduler tick is considered at the start of the tick.
- 2) Decide which task should be run in the current tick using the formula given in section 4.3.3:

$$(\text{Current_tick} - \text{Offset}_i) \% \text{Period}_i = \text{Run}_i$$

For task i , if Run_i equal to 0, then task i should run in Current_tick .

- 3) Decide the boundary of the start and end time of each task.

In Current_tick, the start time of task i is between

$$Time_start_i^{lower_bound} = \sum_{n=1}^{i-1} (Estimated_BCET_n * Run_n) \text{ and}$$

$$Time_start_i^{upper_bound} = \sum_{n=1}^{i-1} (Estimated_WCET_n * Run_n) .$$

The end time of task i is between

$$Time_end_i^{lower_bound} = Time_start_i^{lower_bound} + Estimated_BCET_i \text{ and}$$

$$Time_end_i^{upper_bound} = Time_start_i^{upper_bound} + Estimated_WCET_i .$$

These set of data could be calculated at the design stage, and stored in the form of an array in the EA. However, the memory requirements would become significant if the target system contains a large number of tasks. Thus, the ASA performs this calculation at runtime.

The sampling should start before the lower bound of a task start and end time to ensure that the change of a pin state can be captured. The sampling should only stop at the point when the change is detected. Note that if no BCET information of tasks is provide, the ASA needs to sample from the start to the end of a task execution.

In time-triggered designs, “balanced” tasks are sometime employed to fix task execution times (e.g. see Gendy et al., 2007). If the ASA is employed in such a system, there is no need for sampling from the lower bound of a task start and end time.

5.4.3 Modified IM

The IM for ASA is the same as PSA. However, the ASA only requires the signal from the task interrupt pin generated by the IM in the tick timer ISR. This signal is used to trigger an interrupt in the EA for the tick interval error detection.

5.4.4 EA implementation

Figure 5-8 shows the abstract flowchart of the EA. The differences between the ASA and the PSA are that the runtime analyzer in the ASA is spitted into two parts, but the data receiver and the event recognizer are merged together.

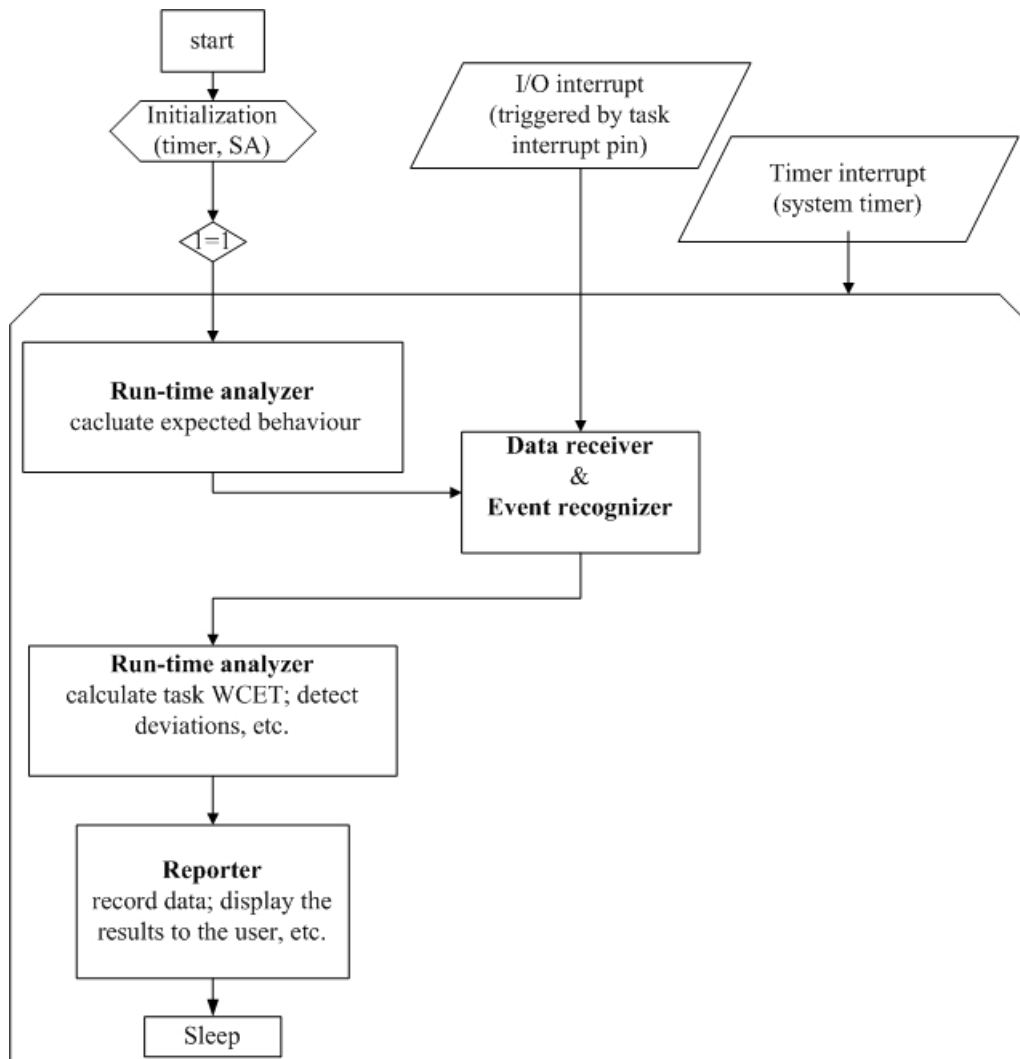


Figure 5-8 Abstract flowchart for the EA of ASA

The first part of the runtime analyzer calculates the expected task execution sequence for the next tick in advance. This information will be used by the event recognizer to organize the activities of the data receiver. The data receiver starts sampling at the lower bound of the key points, and ends at the point when any change in the target system is detected. Additionally, the ISR for the external event, which is triggered at the start of each scheduler tick of the target, can activate the data receiver. The reason for this is to ensure the EA to capture the scheduler tick and the tasks in the tick when the tick interval is incorrect.

5.5 Conclusion

The implementation of the two versions of SA is explained. The PSA passively receives the target information from the target. The ASA takes advantages of the TTC scheduling

behaviour. It collects information from the target system only at the key points - where a task is due to start / end. A case study to evaluate these two versions of SA is presented in Chapter 7.

6 SA Automation Tool

The previous chapter has presented the implementation of the SA. This chapter describes the SA Automation Tool which is designed to help the user implementing the SA.

6.1 Introduction

In Chapter 4, techniques for creating a Scheduler Agent (SA) were discussed. If such an approach is to be practical in real-world systems, a means of automating the creation of the SA would be very useful. Because of the nature of the SA (and its link to statically-scheduled TT systems), it is expected that automatic creation of the SA code would be comparatively straightforward. To test this assumption, a simple SA automation tool was created. The prototype tool is described in this chapter. (This prototype is designed base on ARM LPC21xx family.)

As shown in Figure 6-1, this tool consists of three main modules:

- 1) Target analyzer - it is used to analyze the source code of the target and to retrieve information such as scheduler and task settings.
- 2) SA generator - it is used to generate the source code for the IM and EA.
- 3) User interface - it allows the user to input the target source code and the desired SA settings.

The remainder of this chapter describes the design and implementation of these modules.

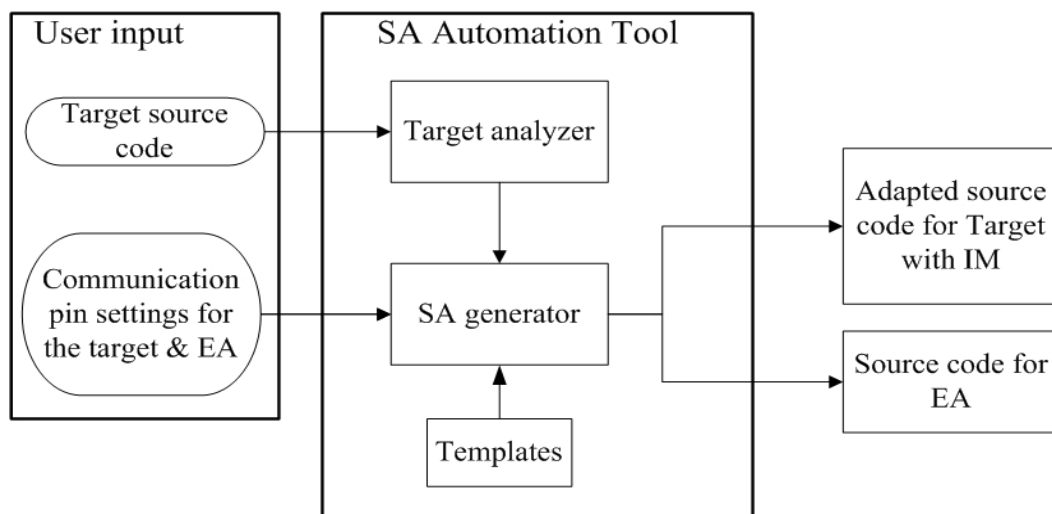


Figure 6-1 Overview of the SA Automation Tool

6.2 Target analyzer

The target analyzer collects the specification of the target from its source code. The key information is stored in the “main” file where the task information is added, and the “scheduler” file where the scheduler is setup. The major task for the analyzer is to analyze these files.

The analysis is done by using the regular expression. The regular expression, also called “regex”, is the way to match specific strings from a set of character strings. In our tool, the regular expression is used to extract information. We use the “main.c” to explain how this tool was implemented.

As Figure 6-2 shown, the analyzer reads the “main” file from the user specified directory as a text file. Then, the text file converts into a set of strings which are stored in an array. Each string in the array is one line of the original code.

```
string MPmainpath = MP_Directory + "\\Main.c";
TextReader MPmainReader = new StreamReader (new
FileStream(MPmainpath, FileMode.Open));
Regex splitter = new Regex(Environment.NewLine);
string[] MPmainlist = splitter.Split(MPmainReader.ReadToEnd());
```

Figure 6-2 Code fragment of read “main.c” file

From the pattern of TTC scheduler, we know that the tasks are added into the scheduler in the form shown in Figure 6-3.

```
SCH_Add_Task(Sampling, 0, 1);
SCH_Add_Task(Computation, 1, 3);
SCH_Add_Task(Actuation, 2, 3);
```

Figure 6-3 The add task function of the TTC scheduler

Therefore, we can define a regex rule (as shown in Figure 6-4) to extract the name, the offset and the period. (Note that, the MP (the monitored processor) is used to represent the target in this study.)

```
Regex MPTaskInforegex = new Regex(@"\"s*SCH_Add_Task\"s*[()]\s*" +
    @"(?<TaskName>\w+)" + @"\"s*,\s*" +
    @"(?<TaskOffset>\d+)" + @"\"s*,\s*" +
    @"(?<TaskPeriod>\d+)");
```

Figure 6-4 Regex rule for task settings

Using the regex rule to check through the entire array, we can get task settings as shown Figure 6-5.

```
for (int i = 0; i < MPmainlist.Length; i++)
{
    if (!Regex.IsMatch(MPmainlist[i], @"^\s*$")) //Is it a common line?
    {
        Match m = MPTaskInforegex.Match(MPmainlist[i]);

        if (m.Success)
        {
            MP_Task_NO++;
            MPTaskInfo[MP_Task_NO].Task_ID = MP_Task_NO;
            MPTaskInfo[MP_Task_NO].Task_Name = m.Groups["TaskName"].Value;
            MPTaskInfo[MP_Task_NO].Period =
            Convert.ToInt32(m.Groups["TaskPeriod"].Value);
            MPTaskInfo[MP_Task_NO].Offset =
            Convert.ToInt32(m.Groups["TaskOffset"].Value);
        }
    }
}
```

Figure 6-5 Code fragment for getting task setting

6.3 SA generator

The SA generator has two tasks: IM and EA code generation.

6.3.1 The IM generation

The SA generator adds the IM to the target. From the last chapter, we showed that the IM is implemented on the scheduler. Therefore, the modifications are solely on the “scheduler” file. The generator follows the steps that described in section 5.3.2. However, there are two problems that need to be solved.

The first problem is to locate the points where the IM should be inserted. To solve this problem, “Regex” is used to match the specific positions. For example, the task port operation should be added before the line of “(*SCH_tasks_G[Index].pTask)();”. A search string is define as @"pTask()]" to match the line. (Since only this line contains pTask() in the whole file.)

In the generator, we define an array of string to locate each position, and match the entire files to insert monitoring codes at these locations as shown in Figure 6-6.

```

string[] Search_String = new string[]
{
    @"include",
    @"VICIntEnable",
    @"T0TCR |= 0x01;",
    @"Tick_count_G",
    @"pTask[][(ID)]",
    @"Update_required = 1",
    @"Update_required = 0" };

for (int i = 0; i < MPSchlist.Length; i++)
{
    MPSchlist_adapted[MPSch_Index++] = MPSchlist[i];
    if (Regex.IsMatch(MPSchlist[i], Search_String[Search_Index]))
    {
        switch (Search_Index)
        {
            case 0:
            {
                //inserting point, add code here
            }
            .....

```

Figure 6-6 Code fragment for showing an inserted locations for software sensors

The second problem is to create the code for the operation of the communication pins. Since the settings vary between different applications; the tool currently can only support LPC21xx processors. However, the method can be extended to support other processors.

In the tool, we created functions for setting up the pins function, direction and value. The code fragment in Figure 6-7 shows the how to set up the direction and generate the value of a pin of Port 0 and 1.

```

public string SetpinDirection(int port, int pin, bool dir)//0-input; 1-output
{
    string st = "";
    if (port == 0)
    {
        if (dir)
            st = "IODIR0 |= 0x" + Setpinvalue(pin).ToString("X8") + ";//output";
        else
            st = "IODIR0 &= ~0x" + Setpinvalue(pin).ToString("X8") + ";//output";
    }
    else if (port == 1)
    {
        if (dir)
            st = "IODIR1 |= 0x" + Setpinvalue(pin).ToString("X8") + ";//output";
        else

```

```

    st = "IODIR1 &= ~0x" + Setpinvalue(pin).ToString("X8")+ ";//output";
}
return st;
}

public string Pinwrite (int port, int pin, bool output)
{
    string st = "";
    if (port == 0)
    {
        if(output)
            st = "IOSET0 |= 0x" + Setpinvalue(pin).ToString("X8")+ ";//set";
        else
            st = "IOCLR0 |= ~0x" + Setpinvalue(pin).ToString("X8")+ ";//clear";
    }
    else if (port == 1)
    {
        if(output)
            st = "IOSET1 |= 0x" + Setpinvalue(pin).ToString("X8")+ ";//set";
        else
            st = "IOCLR1 |= ~0x" + Setpinvalue(pin).ToString("X8")+ ";//clear";
    }
    return st;
}

```

Figure 6-7 Code fragment for I/O port pin

Overall, the generator uses the regex to locate the insertion point, and then uses the above functions to create the code for the user defined pin settings.

6.3.2 The EA generation

The EA is generated from general templates. The templates contain the basic EA code with the target information replaced with special marks. To create an EA, the generator needs to load the template and customizes it according to the actual target information. The process uses the regex to locate the marks and set up the communication port which has been mentioned.

6.4 User interface

This section describes the user interface (UI) of the tool. Figure 6-8 shows the first UI of the tool. It allows the user to specify the file directory of the target firmware and the file location to store the final output.

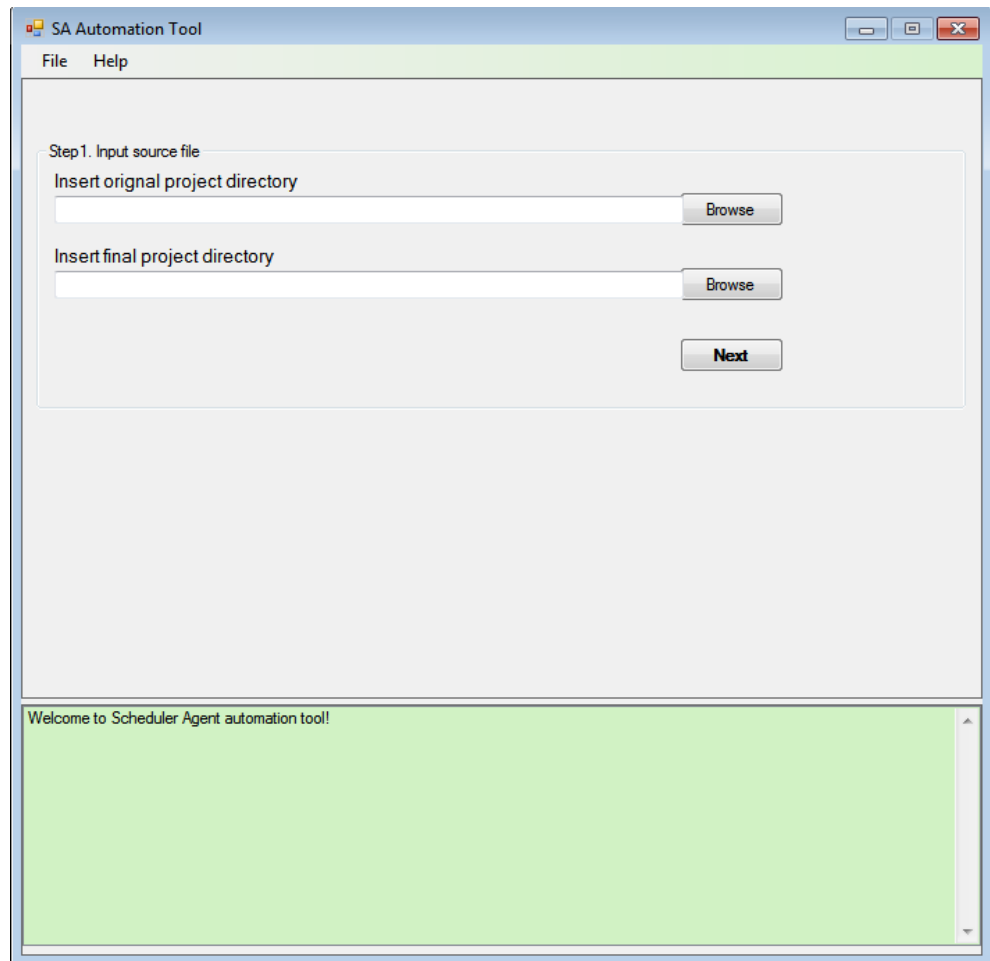
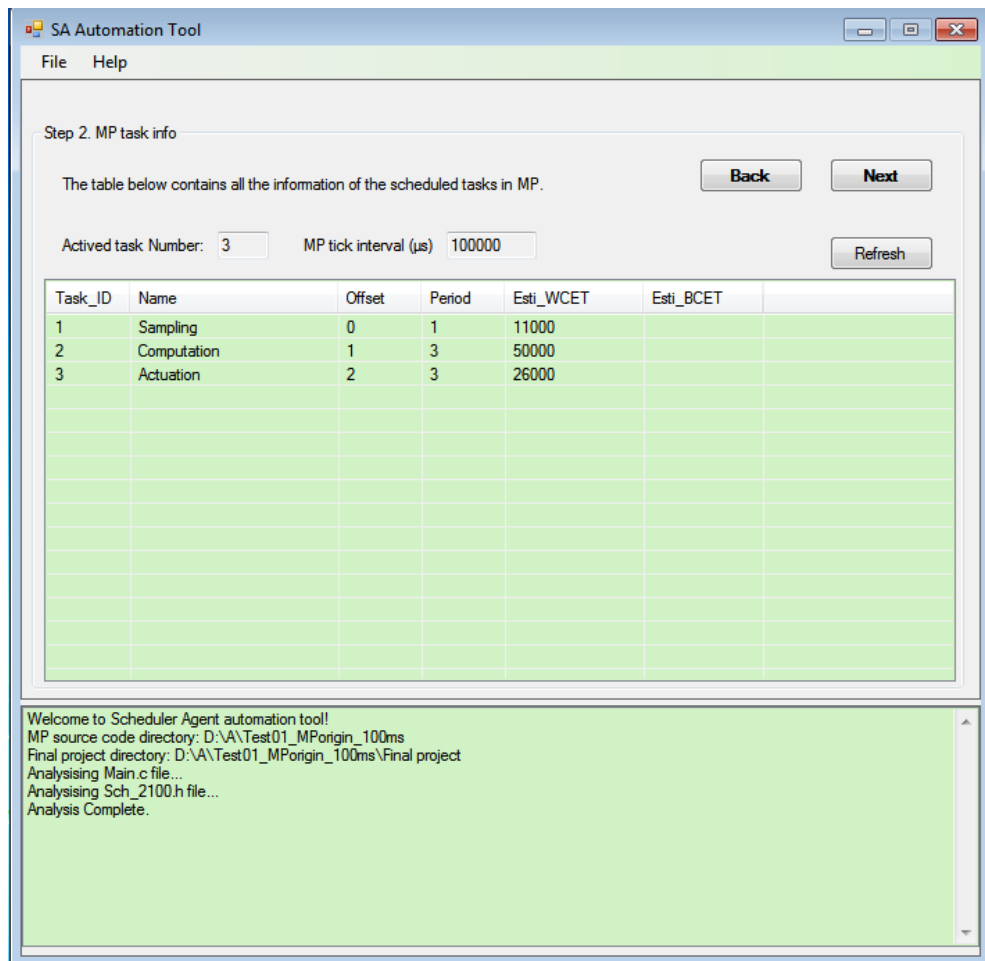


Figure 6-8 UI-1

When the 'next' button is pressed, the tool will start searching the “main” and “scheduler” file in the original directory. If no such file is found, an error message will be displayed in the text box. If the files are found, the target analyzer will start extract the task and scheduler information and display them in the next screen (shown in Figure 6-9).



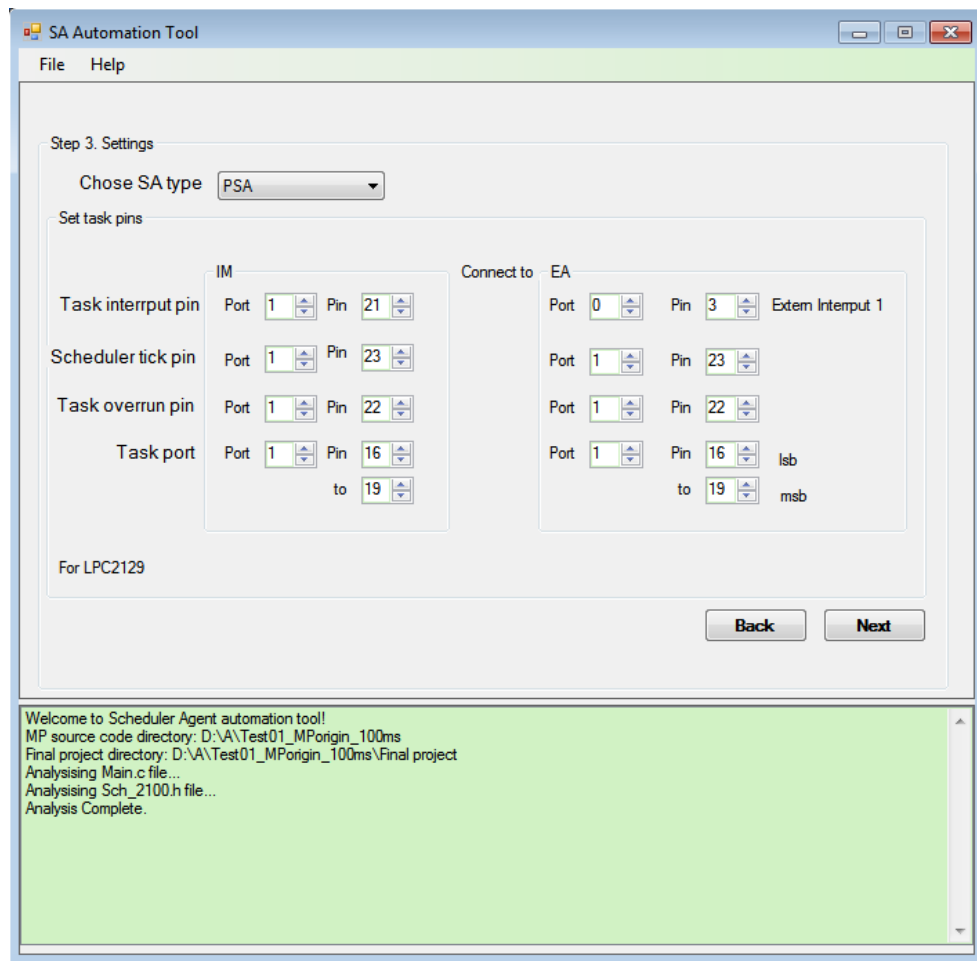


Figure 6-10 UI-3

After successfully generated the code, the screen in Figure 6-11 is shown. By clicking on the 'open final project folder' button, the folder will be opened in windows explorer.

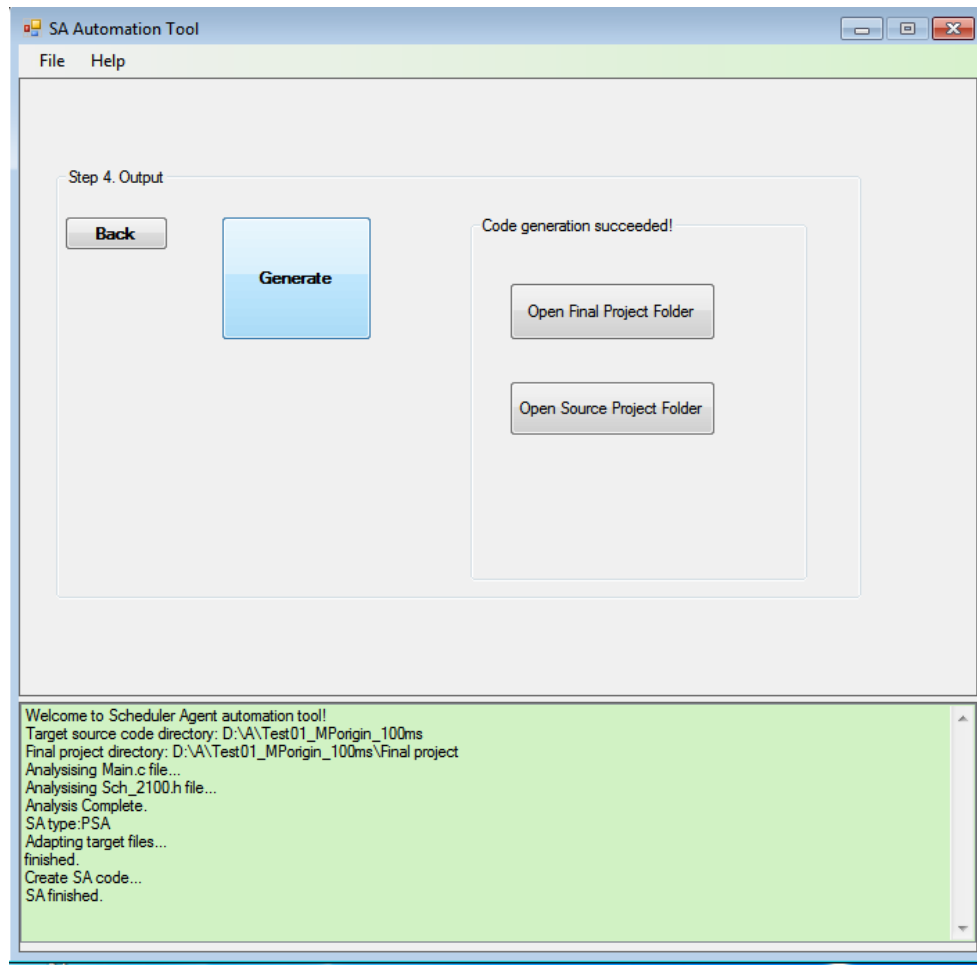


Figure 6-11 UI-4

6.5 Conclusion

The SA Automation Tool is designed to simplify the implementation process of the SA. It has three key modules, the target analyzer, SA generator and user interface. The target analyzer can assist the user to extract target specification. The SA generator can produce customized SA code. The user interface help user to input requirements.

7 Case study

This chapter presents a case study to demonstrate the use of the PSA and the ASA. Both of these versions of SA are created by the SA automation tool described in the previous chapter.

7.1 Introduction

This case study aims to demonstrate that the SA Automation tool can implement working Scheduler Agents (SA), as well as to demonstrate the functions of PSA and ASA.

The target contains a set of 3 tasks (Task Sampling, Task Computation, and Task Actuation). This set of tasks was intended to be the representative tasks of a typical embedded control system.

In this system, Task Sampling is the first task to run and used for data sampling. Task Computation then executes the control algorithm and – finally – Task Actuation controls the actuator(s). In this case study, the target executes these 3 tasks using a TTC scheduler. The NXP LPC2129 microcontrollers were chosen as the hardware platform for the target and the SA.

The case study employed the SA Automation Tool to generate the code for SA. Then, the generated SA is used to monitor the target and detect timing errors. Also, a set of timing errors is injected to the target, which is used to demonstrate the fault detection function of the SA in details.

7.2 SA implementation

The implementations of the SA are performed by the SA Automation Tool. As the Figure 6-9 shown, the target analyzer extracted the task and scheduler information correctly. Then, the communication port is set as the Figure 6-10 shown. The original target code and implemented SA code are attached in the Appendix.

Both PSA and ASA employ a serial port to pass information such as task execution time and error messages to the host computer. Information received by the host will be displayed using the HyperTerminal.

7.3 Fault injection

In order to investigate and compare the behaviour of the SAs, errors were injected to the target as shown in Table 7-1. In this example, the tasks have been added with a delay function, which is controlled by a set of flags. The delay function will be activated only if the fault-injection function sets the flag. (The target without these faults is called normal system.)

Table 7-1 Fault injection scenarios

NO.	Purpose	Description
1	Check for task exceeding “WCET”	Slightly increase Task Sampling execution time at Tick 500.
2	Check for task overrun	Increase Task Computation execution time at Tick 1001. Increase Task Actuation execution time at Tick 2001.
3	Check for errors in task execution sequence	Task Sampling omitted from schedule at Tick 3000. Task Actuation omitted from schedule at Tick 5001.
4	Check for variations in the scheduler tick interval	Change scheduler tick interval from 100ms to 200ms at Tick 8000.

7.4 Test result for PSA

This section contains the test result that obtained from the PSA. The monitored target is operated with the normal system to show the basic function of the PSA. Then, the target is operated with the fault injected system that described above to further demonstrate the fault detection function of the PSA.

7.4.1 Normal system

Monitor the normal target system with the PSA. The PSA is set to display the target states in runtime. The PSA sent out a line of message at each tick. Each line indicates the tick number, and tasks that has executed in this tick, together with their execution time. Table 7-16 is a part of the data that PSA send out.

Table 7-2 PSA output for the normal system - runtime

TICK 1001	T1= 10138	T2= 30036
TICK 1002	T1= 10138	T3= 25290
TICK 1003	T1= 10138	
TICK 1004	T1= 10138	T2= 38194
TICK 1005	T1= 10138	T3= 25290
TICK 1006	T1= 10138	
TICK 1007	T1= 10138	T2= 42271
TICK 1008	T1= 10138	T3= 25290
TICK 1009	T1= 10138	
TICK 1010	T1= 10138	T2= 37180

T1 is Task Sampling, T2 is Task Computation, T3 is Task Actuation. The SA only displays the ID number of tasks, not the actual name of tasks. The unit is in micro-second (μ s), unless otherwise stated.

The PSA also can be configured to display a statistics of the target at a fixed point. In this example, the PSA is configured to send the WCET of each task that recorded from the start tick 10000th, as Table 7-3 shown.

Table 7-3 PSA output for the normal system – tick 10000th

Total task: 3
T1= 10138, T2= 50020, T3= 25290
Total Error:0

7.4.2 System with faults

In this test, we injected errors in the target and observed its behaviour (via the PSA).

1) Check if a task exceeds the estimated WCET

Inject error NO 1: Slightly increased Task Sampling execution time at Tick 500.

Result: The PSA detected the error (as Table 7-4 shown).

Table 7-4 PSA detected error NO 1

ERROR 1	TICK 500	T1 over estimated WCET
TICK 500	T1= 19538	T2= 37181

2) Check if a task misses its deadline (task over run)

Inject error NO 2: Increased Task Computation execution time at Tick 1001.

Increased Task Actuation execution time at Tick 2001.

Result: The PSA detected the error (as Table 7-5 shown).

Table 7-5 PSA detected error NO 2

ERROR 2	TICK 1001	From T2miss deadline
ERROR 3	TICK 1001	T2 over estimated WCET
TICK 1001	T1= 10120	T2= 94380
ERROR 4	TICK 2001	From T3miss deadline
ERROR 5	TICK 2001	T3 over estimated WCET
TICK 2001	T1= 10138	T3=155340

3) Check if there is a task execution sequence error.

Inject error NO 3: Task Sampling missed the deadline at Tick 3000.

Task Actuation missed at tick 5001st.

Result: The PSA detected the error (as Table 7-6 and shown).

Table 7-6 PSA detected error NO 3

ERROR 6	TICK 3000	T1missed
TICK 3000	T3= 25294	
ERROR 7	TICK 5001	T3missed
TICK 5001	T1= 10138	

4) Check for variations in the scheduler tick interval

Inject error NO 4: Change scheduler tick interval from 100ms to 200ms at Tick 8000.

Result: The PSA detected the error (as Table 7-7 shown).

Table 7-7 PSA detected error NO 4

ERROR 8	TICK 8000	199997 μ s target tick interval wrong
TICK 8000 T1= 10138	T2= 37181	

5) PSA sends summary report at Tick 10000

At the tick 10000, the PSA send a summary of the target shown as Table 7-8. It shows there are 8 errors occurred. The PSA has successfully detected all the errors.

Table 7-8 PSA summary-at tick 10000th

Total task:3
T1= 19538, T2= 94337, T3= 155340
Total Error:8

7.5 Test result for ASA

We have repeated the same experiments as previous section with the ASA.

7.5.1 Normal system

Run the normal target system with the ASA. Table 7-9 is a part of the data that ASA send out.

Table 7-9 ASA output for the normal system - runtime

TICK 1001	T1= 10136	T2= 30032
TICK 1002	T1= 10134	T3= 25288
TICK 1003	T1= 10136	
TICK 1004	T1= 10135	T2= 38189
TICK 1005	T1= 10133	T3= 25288
TICK 1006	T1= 10133	
TICK 1007	T1= 10133	T2=42268
TICK 1008	T1= 10135	T3= 25284
TICK 1009	T1= 10135	
TICK 1010	T1= 10135	T2=37178

Table 7-10 shown the summary from the start to tick 10000th .

Table 7-10 ASA output for the normal system – tick 10000th

Total task:3
T1= 10137, T2= 50019, T3= 25288
Total Error:0

7.5.2 System with faults

In this test, we injected errors in the target and observed its behaviour (via the ASA).

1) Check for task exceeding estimated WCET

Inject error NO 1: Slightly increased Task Sampling execution time at Tick 500.

Result: The ASA detected the error (as Table 7-11 shown).

Table 7-11 ASA detected error NO 1

ERROR 1	TICK 500	T1 over estimated WCET
TICK 500	T1= 19536	T2= 37178

2) Check if a task misses its deadline (task over run)

Inject error NO 2: Increased Task Computation execution time at Tick 1001.

Increased Task Actuation execution time at Tick 2001.

Result: The ASA detected the error (as Table 7-12 shown).

Table 7-12 ASA detected error NO 2

ERROR 2	TICK 1001	From T2miss deadline
ERROR 3	TICK 1001	T2 over estimated WCET
TICK 1001	T1= 10136	T2= 94333
ERROR 4	TICK 2001	From T3miss deadline
ERROR 5	TICK 2001	T3 over estimated WCET
TICK 2001	T1= 10136	T3=155337

3) Check for errors in task execution sequence

Inject error NO 3: Task Sampling missed the deadline at Tick 3000.

Task Actuation missed at tick 5001st.

Result: The ASA detected the error (as Table 7-13 and shown).

Table 7-13 ASA detected error NO 3

ERROR 6	TICK 3000	T1missed
TICK 3000	T3= 25289	
ERROR 7	TICK 5001	T3missed
TICK 5001	T1= 10136	

4) Check for variations in the scheduler tick interval

Inject error NO 4: Change scheduler tick interval from 100ms to 200ms at Tick 8000.

Result: The ASA detected the error (as Table 7-14 shown).

Table 7-14 ASA detected error NO 4

ERROR 8	TICK 8000	199996µs MP tick interval wrong
TICK 8000	T1= 10137	T2= 37178

5) ASA sends summary report at Tick 10000

At the tick 10000, the ASA send a summary of the target shown as Table 7-15. It shows there are 8 errors occurred. The ASA has successfully detected all the errors.

Table 7-15 ASA summary-at tick 10000th

Total task:3
T1= 19536, T2= 94333, T3= 155337
Total Error:8

7.6 Discussion

7.6.1 Test Results

The SA Automation tool has successfully implemented the PSA and the ASA for this application. From the results in the sections 7.4 and 7.5, we confirm that both the PSA and the ASA have similar outputs of tasks' WCET and are capable to detect the injected faults.

Table 7-16 shows the actual WCET of the tasks that measured by the PSA and ASA. The values that retrieved from the ASA are slightly lower than the PSA. The maximum discrepancy between them is approximately 2 μ s. This may be caused by the interrupt latency in the PSA data receiver.

Table 7-16 The actual WCET measured by the PSA and ASA

	PSA (μ s)	ASA (μ s)
Task Sampling	10138	10137
Task Co	50020	50019
Task Ac	25290	25288

It is important to mention that, in both version of SA, the scheduler overhead was not considered. This was because the interrupts at the end of a task and the start of the next task are likely to be too close. However, if high accuracy timing is required, the scheduler overhead should be considered.

As the injected faults (including exceed estimated WCET, missing deadline, execution sequence errors, variations in the scheduler tick interval) were detected by both SA in this case study, we may conclude that both versions of the SA satisfy the functional requirements. Since the SA has obtained the details of each task, it is easy for it to identify the source of an error. Note the difference between the monitored and unmonitored system is that the monitored system has the IM added. For an unmonitored system (without SA), it will be difficult to know the occurrences of the timing errors.

7.6.2 Resource requirements

Since the IM resides within the target system, it will consume the target resource. Table 7-17 shows the comparison of the resource requirements for the IM of both versions of SA. It shows that the code size (in the Flash memory) of the IM of the PSA increases by 636 bytes, and the code size of the IM of the ASA increases by 492 bytes. (This data is collected in normal compiler setting. By setting a high optimization level, the code size could be smaller.) The data size (RAM) of the IM of both versions of SA increases by 1 byte. Compared with the LPC2129, which has 16Kbytes of RAM and 128Kbytes of Flash memory, the size of IM is acceptable.

Table 7-17 Comparison of the resource requirements

(bytes)	Original	+IM(PSA)	+ IM(ASA)
Code size	2928	3564	3420
Data size	46	47	47

Note that the IM is inserted to the scheduler. Therefore, when the number of tasks increases, the IM size will not increase unless the required number of communication pins increase.

Table 7-18 shows the implementation costs of the PSA and ASA.

Table 7-18 Comparison of the implementation costs of the 4 SA

(bytes)	PSA	ASA
Code size	10624	11764
Data size	514	630

7.6.3 Interference

During the implementation of the IM, we attempted to minimize its interference to the target system by placing it into the scheduler. The scheduler needs to run through the IM sensor before calling a task. Therefore, the interference caused by the IM, is associated with the scheduler. This will slightly increase the scheduler overhead, but the change is transparent and can be determined by the developer (since no changes are required to individual tasks).

The IM can reside in the target permanently, even if monitoring is not required. In this way, we can bypass the inconsistency of the system behaviour problem that may be caused by removing the monitor from the target.

7.6.4 Issues with the communication pins

Our communication between the IM and the EA relies on a set of wire connections. We assumed that all the wires are firmly connected and there is no signal lost.

If this assumption not holds, the SA may derive incorrect results. One of the solutions to address this problem is using redundant connections. For example, employ three set of connection, and a hardware/software voter to determine the correctness of the signal (Johnson 2000; Shooman 2002). This method works when there is only one set of signal goes wrong. It also requires an additional connection, which may not be available to small pin-out processors.

Another method to solve the broken wire problem is to set the IM to send a test signal cross over the entire communication pins to check for the wire condition. We can set this test signal periodically based on a predefined sequence. If the EA cannot receive the right value, there might be some connection problems.

7.6.5 Limitations of the SA automation tools

The SA automation tools demonstrated that the code for SA can be generated automatically. However, the SA employed the regex to extract target information will limit the target code (the scheduler files) to follow certain standards. By using tool chain, such as RapiDiTTy (produced by TTE systems), the problem could be solved.

7.7 Conclusion

This case study demonstrated that the SA Automation Tool can implement the PSA and ASA. It also demonstrates that the generated PSA and ASA can function as the expected. A discussion is conducted to evaluate their performance.

8 Discussion and Conclusion

This chapter presents a summary and evaluation of the research conducted and presented in this thesis. Limitations and future work of this research work are also discussed.

8.1 Introduction

The research work presented in this thesis is concerned with the monitoring of resource-constrained embedded systems using time-triggered architecture (i.e. TTC scheduler). Such architecture is using a static scheduling with no pre-emption. Systems which employ this architecture can have highly predictable behaviour. Also, it can help to reduce the context switching overhead and maintain at a low level of jitter.

Despite the above attractive features, it is fragile to task overruns. Task overrun happens when a task executes more than its expected time. This may cause a serious impact on the system behaviour. The cause of overrun comes from two aspects. One of the aspects is the hardware failures, such as analogue-to-digital conversion failure, sensors failure or power-supply reduction. Another aspect is the design errors. For example, the WCET of tasks is required for the task schedule at the design stage. However, an accurate WCET estimation is not difficult to achieve.

Monitoring such real-time embedded systems is not straightforward – many embedded systems are implemented in electronics and sealed in small packages which limit the access from the external world. Also, the resource-constrained nature of embedded systems makes it even more difficult to be monitored; since only very little resource in the system can be used for the monitor. For example, the stringent real-time requirements of embedded systems limit the CPU utilisation for the monitor. Additionally, inserting instrumentation codes into the target software may not be possible due to the limitation of the memory size. Even if it is possible, adding a monitor to the target system may alter the system behaviour unintentionally, i.e. causing the “probe effect”.

8.2 Goals and achievements

The work presents in this thesis is to develop a monitoring solution to help for testing and detecting faults of embedded systems. The target system in this study employs a TTC scheduler which has the real-time and resource constraints. The focus of this study

is on monitoring the timing behaviour and detecting temporal faults (e.g. task overruns and task execution ordering errors) in the target system. We aim to develop a monitoring technique to achieve this goal with using minimum target resources and as low interference to the target system as possible.

Our monitoring system is called “scheduler agent” (SA). The approach we used is hybrid monitoring approach. From the discussion of Chapter 3, we understand that hybrid monitoring approach requires less resource from the target system than software-based approach; since the instrumentation code for fault detection is migrated to an external analyzer. Hence, the interference to the target system is reduced. Also, the external monitor will not be affected by the errors caused by the target.

SA consists of two parts: an internal monitor (IM) and an external analyzer (EA). Both of the IM and the EA have to work cooperatively in order to obtain information from the target system. The communication between them relies on a coding scheme and a physical data.

To connect the IM and the EA, a communication link is needed. In chapter 4, serial communication and GPIO interfaces are discussed. Since a GPIO interface is much simpler, quicker and less interference than a serial communication interface, the GPIO interface is selected for the implementation of the SA. However, an encoding technique is required since modern microcontrollers may not have enough GPIO port pins to represent all tasks in the target system. A simple and effective encoding technique is addressed to this issue.

Two versions of the SA – Passive SA (PSA) and Active SA (ASA) – are implemented. PSA retrieves task information from the “instrumented” target system passively. ASA takes advantages from the TTC architecture in the target system in which the monitoring process collects information from the target system at the time when a task is due to start and end. This approach reduces the CPU usage significantly without losing performance.

We also developed a SA automation tool which can automatically generate the SA code for the external analyzer and the target system. This tool is being used in the case study to generate the source code for both PSA and ASA.

From the case study, we showed that the SA can be easily implemented. We also confirmed that the functionality of SA is in line of the requirements; since it is capable to measure task execution time and detect temporal errors (including exceed estimated WCET, missing deadline, execution sequence errors, variations in the scheduler tick interval) in the target system.

The interference to the scheduler caused by the IM is minimized. It is transparent to and can be determined by the developer. The IM can reside in the scheduler which helps to bypass the probe effect. With the help of SA automation tool, the SA can be easily implemented.

In this study, the EA is implemented on an ARM-based microcontroller. The EA can also be implemented on other types of microprocessors, since the connections between IM and EA is based on a GPIO interface. However, the accuracy of the measurement will depend on the microprocessor's frequency and the timer settings..

8.3 The limitations of the work

The monitoring technique presented in this thesis demonstrated the ability to monitor the temporal behaviour of time-triggered embedded systems. However, there are a few limitations on this technique.

- 1) This monitoring approach requires the target resource including GPIO pins, which is dependent on the maximum number of tasks, the memory size and the CPU time to run the IM. For application which has tight resource constraints, this approach may not be used.
- 2) This approach requires an independent microprocessor for the external analyzer. Although it brings the benefits of requiring less resource from the target system, causing less interference to the target behaviour, and will not be affected by the target faults, it also limits its applicability.
- 3) Currently, the SA only supports for TTC scheduler and the SA Automation Tool is only supporting LPC21xx microcontrollers.
- 4) The data retriever of PSA is triggered by an external interrupt. Therefore, the result will be less accuracy for short duration tasks. The ASA, on the other hand, can sampling the task port continuously which can produce a better result.

However, since the time stamping is performed in the software, there may be a slight delay caused by the execution of time stamping.

- 5) Also, in both versions of SA, the scheduler overhead is not measured. This is because the time between the end of one task and the start of the next task are likely to be too close.

8.4 Future work

To improve the SA performance, FPGA could be used to implement the SA in future. By implement the data receiver and event recognizer into hardware, the SA can achieve better accuracy and have more CPU time to processing data. Also, the SA will be able to measure the scheduler overhead and short duration tasks accurately.

The SA currently is only supporting TTC scheduler; works can be done to extend its capability to time triggered hybrid (TTH) and other pre-emptive schedulers.

The SA Automation Tool is only supporting the LPC21xx microcontrollers; further works can be done to cover more processors.

Another potential area for future work is to extend the capability of SA to recovery the target system from faults. The SA can forward fault information to the recovery mechanism, so that the system can be recovered. Also, early fault detection may be possible, since the SA can predict the expected behaviour of the target system. Therefore, when a task is overrun its deadline, the SA can inform the target immediately, i.e. before the task is finished. With such information, the target can has more recovery choices.

Last but not least, further study can be made to combine timing behaviour monitoring with performance monitoring. In the SA design we used a basic encoding technique to send out information. Thus, the required number of pins is reduced.. The encoding technique can be further enhanced to deliver more information.

8.5 Conclusion

The project described in this thesis has made three major contributions to the field of monitoring embedded systems which employs time-triggered architectures. Firstly, design and implementation of a hybrid monitoring technique (i.e. the "scheduler agent")

that can examine the temporal behaviour of tasks running on a target embedded system. Two different versions of SA were developed: Passive AS (PSA) and Active SA (ASA). Secondly, it developed a code-generation tool ("SA Automation Tool") to generate a customized SA code for the PSA and ASA. Finally, it evaluated the SA which was generated by the SA Automation Tool. Future work to extend and improve the efficiency of the monitoring techniques in this study is discussed.

References

- Ahmad, A., M. S. Ahmad, et al. (2008). An exploratory review of software agents. Information Technology, 2008. ITSIm 2008. International Symposium on.
- Allworth, S. T. (1981). An Introduction to Real-Time Software Design, Macmillan, London.
- Audsley, N., K. Tindell, et al. (1993). The end of line for static cyclic scheduling? Fifth Euromicro Workshop on Real-Time Systems.
- Ayavoo, D., M. J. Pont, et al. (2007). "Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems." Microprocessors and Microsystems 31(5): 326-334.
- Baker, T. P. and A. Shaw (1989). "The cyclic executive model and Ada." Real-Time Systems 1(1): 7-25.
- Burns, A. and A. J. Wellings (2006). Programming execution-time servers in ada 2005. Proceedings of the 27th IEEE Real-Time Systems Symposium.
- Buttazzo, G. C. (2002). Hard real-time computing systems predictable scheduling algorithms and applicaions, Kluwer academic publisher.
- Buttazzo, G. C. (2005). "Rate monotonic vs. EDF: judgment day." Real-Time Syst. 29(1): 5-26.
- Cabri, G., L. Leonardi, et al. (2003). Implementing role-based interactions for Internet agents. Applications and the Internet, 2003. Proceedings. 2003 Symposium on.
- Cadamuro, J. J. and D. P. B. Renaux (2008). Efficient Monitoring of Embedded Real-Time Systems. Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on.
- Cervin, A., D. Henriksson, et al. (2003). How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. IEEE Control Systems Magazine. 23: 16 - 30.

- Chan-gun, L., A. K. Mok, et al. (2007). "Monitoring of Timing Constraints with Confidence Threshold Requirements." *Computers, IEEE Transactions on* 56(7): 977-991.
- Colin, A. and I. Puaut (2000). "Worst Case Execution Time Analysis for a Processor with Branch Prediction." *Real-Time Systems* 18(2-3): 249-274.
- Cottet, F. and L. David (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.
- Delgado, N., A. Q. Gates, et al. (2004). "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools." *IEEE Trans. Softw. Eng.* 30(12): 859-872.
- Deverge, J. and I. Puaut (2005). Safe measurement-based WCET estimation. Proc. of the 5th Workshop on Worst-Case Execution Time Analysis, held in conjunction with the 17th Euromicro Conference on Real-Time Systems.
- Dodd, P. S. and C. V. Ravishankar (1992). "Monitoring and debugging distributed real-time programs." *Software-Practice and Experience* 22(10): 863-877.
- Domaratsky, Y. and M. Perevozchikov (2000). Highly Dependable Time-Triggered Operating System. *Dedicated Systems Magazine*. 4: pp. 77-80.
- Douglass, B. P. and D. Harel (1997). *Real-time UML: Developing efficient objects for embedded systems*, Addison Wesley Longman.
- Engblom, J. (2002). *Processor Pipelines and Static Worst-Case Execution Time Analysis*. Department of Information Technology, Uppsala University.
- Engblom, J. and A. Ermedahl (2000). Validating a worst-case execution-time analysis method for an embedded processor. Proc. 21st IEEE Real-time Systems Symposium (RTSS'00), Orlando, Florida, USA.
- Engblom, J., A. Ermedahl, et al. (2001). "Worst-Case Execution-Time Analysis for Embedded Real-Time Systems." *Journal of Software Tools for Technology Transfer* 4(4): 437-455.

Engblom, J. and B. Jonsson (2002). Processor pipelines and their properties for static WCET analysis. Proceedings of the Second International Conference on Embedded Software, London, UK, Springer-Verlag.

Ferdinand, C., R. Heckmann, et al. (2001). Reliable and Precise WCET Determination for a Real-Life Processor. Embedded Software: First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings (EMSOFT 2001), Springer-Verlag. LNCS 2211: 469–485.

Ferdinand, C. and R. Wilhelm (1999). "Efficient and precise cache behavior prediction for real-time systems." Journal of Real-Time Systems 17: 131-181.

Franklin, S. and A. Graesser (1996). Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents. Proc. 3rd Int'l Workshop on Agent Theories, Architectures & Languages, Springer-Verlag.

Ganssle, J. and Barr M. (2003). Embedded systems dictionary, CMP Books.

Gendy, A., L. Dong, et al. (2007). Improving the performance of time-triggered embedded systems by means of a scheduler agent. Proc. of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007), Las Vegas, Nevada, USA.

Gendy, A. K. and M. J. Pont (2008). "Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems." IEEE Transactions on Industrial Informatics 4(1): 37 - 46.

Gergeleit, M. (2001). A Monitoring-Based Approach to Object-Oriented Real-Time Computing. Faculty for computer science. Germany, Otto-von-Guericke University. PhD Thesis.

Gergeleit, M. and E. Nett (2002). Scheduling transient overload with the TAFT scheduler. GI/ITG specialized group of operating systems. Berlin.

- Haban, D. and D. Wybraniec (1990). "A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems." *IEEE Transactions on Software Engineering* 16(2): 197-211.
- Hansson, H., M. Nolin, et al. (2006). Real-time in embedded systems. *embedded systems handbook*. R. Zurawski, Taylor & Francis: 2-1-2-35.
- Harbour, M. G. and M. A. Rivas (2003). "Managing multiple execution-time timers from a single task." *ACM SIGAda Ada Letters XXIII*(4): 28 - 31
- Heath, S. (2003). *Embedded System Design*, Newnes.
- Heckmann, R., M. Langenbach, et al. (2003). "The influence of processor architecture on the design and results of wcet tools." *Proceedings of the IEEE* 91(7): 1038-1054.
- Hughes, Z. H. and M. J. Pont (2004). Design and test of a task guardian for use in TTCS embedded systems. *Proceedings of the UK Embedded Forum*, Birmingham, UK.
- Hughes, Z. M. and M. J. Pont (2007). "Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggeredtime-triggered software architecture is employed." *Trans Institute of Measurement and Control*.
- IEEE (1990). "IEEE standard glossary of software engineering terminology." *IEEE Std* 610.12-1990.
- Jennings, N. R. (2000). "On Agent-Based Software Engineering." *Artificial Intelligence* 117 277-296.
- Johnson, B. W. (2000). Fault Tolerance. *The Electrical Engineering Handbook*. R. C. Dorf, CRC Press.
- Kirner, R. and P. Puschner (2003). Discussion of Misconceptions about WCET Analysis. *3rd Euromicro International Workshop on WCET Analysis*.
- Kirner, R. and P. Puschner (2007). Time-Predictable Task Preemption for Real-Time Systems with Direct-Mapped Instruction Cache. *The 10th IEEE International*

Symposium on Object and Component-Oriented Real-Time Distributed Computing,
IEEE Computer Society - Washington, DC, USA

Kirner, R., P. Puschner, et al. (2004). Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation. Proceedings of the 4th Workshop on Worst-Case Execution Time Analysis.

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

Koren, I. and C. M. Krishna (2007). Fault tolerant systems, Elsevier, Inc.

Kurian, S. and M. J. Pont (2007) . "The maintenance and evolution of resource-constrained embedded systems created using design patterns." Journal of systems and software 8(1): 32 - 41

Laplant, P. A. (2004). Real-time systems design and analysis, IEEE press / John Wiley & sons, Inc., publication.

Leen, G., D. Heffernan, et al. (1999). "Digital networks in the automotive vehicle." Computing and Control 10: 257-266.

Lim, S.-S., Y. H. Bae, et al. (1995). "An accurate worst-case timing analysis for risc processors." IEEE Transactions on Software Engineering 21(7): 593-604.

Liu, A. C. and R. Parthasarathi (1989). "Hardware Monitoring of a Multiprocessor System." IEEE Micro 9(5): 44-51.

Liu, C. L. and J. W. Layland (1973). "Scheduling algorithms for multiprogramming in a hard real-time environment." Journal of the ACM 20(1): 40-61.

Liu, J. W. S. (2000). Real-Time Systems, Prentice Hall.

Locke, C. D. (1992). "Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives." Real-Time Systems 4(4): 37-52.

- Mahrenholz, D. (2001). Minimal Invasive Monitoring. The 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001), Magdeburg, Germany, IEEE Computer Society.
- Mansouri-samani, M. (1995). Monitoring of Distributed Systems. Department of Computing. UK, University of London Imperial College of Science. PhD thesis.
- Marti, P. (2002). Analysis and design of real-time control systems with varying control timing constraints. Automatic Control Department, Technical University of Catalonia. PhD thesis.
- Marti, P., R. Villa, et al. (2001). On Real-Time Control Tasks Schedulability. European Control Conference, Porto, Portugal.
- Marwedel, P. (2006). Embedded system design, Springer.
- Mok, A. K. and G. Liu (1997). Efficient run-time monitoring of timing constraints. Real-Time Technology and Applications Symposium.
- Nett, E., H. Streich, et al. (1996). Adaptive software fault tolerance policies with dynamic real-time guarantees. WORDS 96, IEEE Second Int. Workshop on Object-oriented Real-time Dependable Systems, Laguna Beach, California, U.S.A.
- Nissanke, N. (1997). Realtime systems, Prentice-Hall, Inc.
- Nolte, T. (2006). Real-time in embedded systems. Embedded systems handbook. R. Zurawski, Taylor & Francis: 2-1.
- Obermaisser, R., P. Peti, et al. (2001). Monitoring and configuration in a Smart Transducer Network. Proceedings of the IEEE Workshop on Real-Time Embedded Systems, London, UK.
- Ou, N., T. Farahmand, et al. (2004). "Jitter models for the design and test of Gbps-speed serial interconnects." IEEE Design & Test of Computers 21(4): 302 - 313.
- Petters, S. M., P. Zadarnowski, et al. (2007). Measurements or Static Analysis or Both. Proceedings of the 7th Workshop on Worst-Case Execution-Time Analysis, Pisa, Italy.

- Phatrapornnant, T. and M. J. Pont (2006). "Reducing Jitter in Embedded Systems Employing a Time-Triggered Software Architecture and Dynamic Voltage Scaling." IEEE TRANSACTIONS ON COMPUTERS 55(2): 113-124.
- Philips (2004). Data sheet LPC2119/2129: Single-chip 32-bit microcontrollers, Philips Semiconductors.
- Philips (2004). LPC2119/2129/2194/2292/2294; Single-chip 32-bit microcontrollers user manual.
- Plattner, B. (1984). "Real-Time Execution Monitoring." IEEE Trans. Software Eng SE-10(6): 756-764.
- Pont, M. J. (2001). Patterns for time-triggered embedded systems, Addison-Wesley.
- Pont, M. J. (2003). "Supporting the development of time-triggered co-operatively scheduled (TTC scheduler) embedded software using design patterns." Informatica 27: 81-88.
- Pont, M. J. and M. P. Banner (2004). "Designing embedded systems using patterns: A case study." Journal of Systems and Software 71(3): 201 -213.
- Pont, M. J., S. Kurian, et al. (2005). Restructuring a pattern language for reliable embedded systems. ESL Technical Report 2005-01, Leicester University
- Pont, M. J., S. Kurian, et al. (2007). Selecting an appropriate scheduler for use with time-triggered embedded systems. EuroPLoP 2007, Irsee, Germany
- Pont, M. J. and R. H. L. Ong (2002). Using watchdog timers to improve the reliability of single-processor embedded systems: Seven new patterns and a case study". Hruby, P. and Soressen, K. E. [Eds.] Proceedings of the First Nordic Conference on Pattern Languages of Programs.
- Pop, P., P. Eles, et al. (2004). Analysis and synthesis of distributed real-time embedded systems, Kluwer Academic Publishers, Netherlands.

- Puente, J. A. d. I. and J. Zamorano (2003). Execution-time clocks and Ravenscar kernels. Proceedings of the 12th international workshop on Real-time Ada. Viana do Castelo, Portugal, ACM.
- Puschner, P. and A. Burns (2000). "A Review of Worst-Case Execution-Time Analysis " Real-Time Systems 18(2): 115-128.
- Rochange, C. and P. Sainrat (2002). Difficulties in computing the WCET for processors with speculative execution. 2nd International Workshop on Worst Case Execution Time Analysis, Vienna.
- Sandström, K. and C. Norström (2002). Managing complex temporal requirements in real-time control systems. 9th IEEE Conf. Engineering of Computer-Based Systems, Sweden.
- Schossmaier, K. and B. Weiss (1999). An Algorithm for Fault-Tolerant Clock State&Rate Synchronization. Proc. of the 18th IEEE Symp. on Reliable Distributed Systems, Lausanne.
- Sha, L., T. Abdelzaher, et al. (2004). "Real Time Scheduling Theory: A Historical Perspective." Real-Time Syst. 28(2-3): 101-155.
- Shaw, A. C. (2001). Real-time systems and software. New York, John Wiley & Sons Inc.
- Shobaki, M. E. (2004). On-Chip Monitoring for Non-Intrusive Hardware/Software Observability". Sweden, Mälardalen University.
- Shoorman, M. L. (2002). Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design, JOHN WILEY & SONS, INC.
- Short, M. J. and P. J. (2007). "Fault-tolerant time-triggered communication using CAN." IEEE Transactions on Industrial Informatics 3(2): 131 -142.
- Slanina, Z. and V. Srovnal (2008). Embedded Systems Scheduling Monitoring. Systems, 2008. ICONS 08. Third International Conference on.

- Stankovic, J. A. (1988). "Misconceptions about real-time computing: a serious problem for next-generation systems." *IEEE Computers* 21(10): 10 - 19.
- Storey, N. (1996). *Safety-Critical Computer Systems*, Addison-Wesley.
- Thane, H. (2000). *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. Department of Machine Design. Sweden, Royal Institute of Technology, KTH. PhD thesis.
- TimeSysCorporation (2002). *The Concise Handbook of Real-Time Systems*.
- Törngren, M. (1998). "Fundamentals of implementing real-time control applications in distributed computer systems." *Journal of Real-Time Systems*.
- Tsai, J. J. P., Y. Bi, et al. (1996). *Distributed real-time systems: monitoring, visualization, debugging, and analysis*, John Wiley & Sons, Inc.
- Tsai, J. J. P., K. Y. Fang, et al. (1990). "A noninvasive architecture to monitor real-time distributed systems." *Computer* 23(3): 11-23.
- Vahid, F. and T. Givargis (2002). *Embedded system design* John Wiley & sons, Inc.
- Vallerio, K. S. and N. K. Jha (2003). Task graph extraction for embedded system synthesis. *Proceedings 16th International Conference on VLSI Design concurrently with the 2nd International Conference on Embedded Systems Design*.
- Ward, N. J. (1991). The static analysis of a safety-critical avionics control system. *Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference*, SaRS, Ltd.
- Wavecrest (2001). *Understanding Jitter: Getting Started*, Wavecrest Corporation.
- Weiss, G. (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, The MIT Press.

Wenzel, I., R. Kirner, et al. (2005). Measurement-Based Worst-Case Execution Time Analysis. Proceedings of the 3rd workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05).

Wooldridge, M. (1997). "Agent-based software engineering." Software Engineering. IEE Proceedings- [see also Software, IEE Proceedings] 144(1): 26-37.

Wooldridge, M. (2002). An introduction to multiagent systems, John wiley&sons.

Wooldridge, M. and N. R. Jennings (1995). "Intelligent agents: theory and practice." Knowledge Engineering Review 10: 115-152.

Appendix

This appendix contains the source code for the case study in Chapter 7.

The original target code

This section contains the key “main.c” and “sch_2100.c” file. The “main.c” is where the task information defined. The “sch_2100.c” contains the scheduler settings.

```
/*-----*-
Main.c
-----*/
#include "Main.h"

/*-----*-
main function
-----*/
int Main (void)
{
    System_Init(); //Set up the chip.
    SCH_TTC_Init(100000); //Set up the scheduler

    //Add Tasks here
    SCH_Add_Task(Sampling, 0, 1); //WCET=11000us; DEADLINE=100ms;
    SCH_Add_Task(Computation, 1, 3); //WCET=50000us; DEADLINE=100ms;
    SCH_Add_Task(Actuation, 2, 3); //WCET=26000us; DEADLINE=100ms;

    SCH_Start(); //Start the scheduler.

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
    return 0;
}
/*-----END-----*/

/*-----*-
sch_2100.c (v1.00)
Port of "PTTES" scheduler to ARM (LPC21xx)
-----*/
#include "Main.h"

// ----- Public variable definitions -----
UCHAR Error_code_G;

// ----- Private variable definitions -----
static sTask SCH_tasks_G[SCH_MAX_TASKS]; //Tasks array
static UINT Tick_count_G = 0;

// ----- Private function prototypes -----
static void SCH_Go_To_Sleep(void);

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

```

SCH_Init(unsigned long tick)
Scheduler initialisation function. Prepares scheduler
data structures and sets up timer interrupts at required rate.
Must be called before using the scheduler.
Define CCLKRATIO in Sys_Setup.h file
the timer valude is the tick*CCLKRATIO(us)
-----*/
void SCH_TTC_Init(unsigned long tick)
{
    UCHAR i;

    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;
    TOMRO = tick*CCLKRATIO; // the match register (MRO) to required value
    TOMCR = 0x03; // Interrupt on match, and automatically restart counter
    VICIntSelect |= 0x10; // Assign "Interrupt 4" to the FIQ category
    VICIntEnable |= 0x10; // Enable this interrupt
}

/*-----*/
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)
{
    TOTCR |= 0x01; // Counter enable (Timer Counter Register)
}

/*-----*/
SCH_Update()
This is the scheduler ISR. It is called at a rate
determined by the timer settings in the 'init' function.
Note that an interrupt has occurred.
-----*/
void SCH_Update(void)
{
    Tick_count_G++;
    TOIR = 0x01; //Reset interrupt flag.
}

/*-----*/
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)
{
    UCHAR Index;

```

```

    UCHAR Update_required = 0;

    // Need to check for a timer interuppt since this
    // function was last executed (in case idle mode is not being used)
    VICIntEnClr = 0x10; // Disable timer interrupt
    if (Tick_count_G > 0)
    {
        Tick_count_G--;
        Update_required = 1;
    }
    VICIntEnable = 0x10; // Re-enable timer interrupts

    while (Update_required)
    {
        // Go through the task array
        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 != 0)
                        // 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;
                    }
                }
            }
        }
        VICIntEnClr = 0x10; // Disable timer interrupt
        if (Tick_count_G > 0)
        {
            Tick_count_G--;
            Update_required = 1;
        }
        else
        {
            Update_required = 0;
        }
        VICIntEnable = 0x10; // Re-enable timer interrupts
    } //end of while

    // The scheduler may enter idle mode at this point (if used)
    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().

```
-----*/
UCHAR SCH_Add_Task(void (* pFunction>(),
                    const UINT DELAY,
                    const UINT PERIOD)
{
    UCHAR 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;

        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
-----*/
int SCH_Delete_Task(const UCHAR TASK_INDEX)
{
    int 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_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: May wish to disable this if using a watchdog.
-----*/
void SCH_Go_To_Sleep(void)
{
    PCON = 1;
}
/*-----END-----*/

```

The adapted target code for PSA

To add the IM into the target, only the scheduler file needs to be modified.

```

/*-----*/
sch_2100sa.c (v1.00)
Port of "PTTES" scheduler to ARM (LPC21xx)
This version is modified for safety agent
-----*/
#include "Main.h"

#define MONI_PORT 0X00EF0000 //set P1.16-17 as task moinitor port
#define TASK_MONI_PORT 0X000F0000 //set P1.16-17 as task moinitor port

```

```

#define TICK_PIN 0x00800000 //set P1.23 as scheduler tick pin
#define OVERRUN_PIN 0x00400000 //set P1.23 as scheduler tick pin
#define TASK_INTERRUPT 0x00200000 //set P1.22 as external interrupt

// ----- Public variable definitions -----
UCHAR Error_code_G;

// ----- Private variable definitions -----
static sTask SCH_tasks_G[SCH_MAX_TASKS]; //Tasks array
static UINT Tick_count_G = 0;

// ----- Private function prototypes -----
static void SCH_Go_To_Sleep(void);

/*-----*/
SCH_Init()
Scheduler initialisation function. Prepares scheduler
data structures and sets up timer interrupts at required rate.
Must be called before using the scheduler.
-----
Here:
crystal = 12.0 MHz, PLL = 48.0 MHz ("cclk"),
pclk = cclk (Timer is incremented 48,000,000 times per second.)
So a count of 48,000 (0xBB80) is a 1 ms tick.
75300
So a count of 4,800,000 (0x493E00) is a 100ms tick.
note: 0x493DFF fit best.
Note: here the max task number is 254.
/*-----*/
void SCH_Init(void)
{
    UCHAR i;

    //Set P1.16-24 as task monitor port for safety agent
    PINSEL2 &= ~0x00000004; //Set P1.16-25 as IO
    IODIR1 |= MONI_PORT; //Set P1.16,17,22,23as output
    IOCLR1 |= MONI_PORT; //clear all pins

    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;
    //100ms 4800000
    TOMRO = 4800000; // 0x493DFFSet the match register (MR0) to required value */
    TOMCR = 0x03; // Interrupt on match, and automatically restart counter
    VICIntSelect |= 0x10; // Assign "Interrupt 4" to the FIQ category
    VICIntEnable |= 0x10; // Enable this interrupt
}

/*-----*/
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)
{
    TOTCR |= 0x01; // Counter enable (Timer Counter Register)
}

/*-----*/
SCH_Update()
This is the scheduler ISR. It is called at a rate
determined by the timer settings in the 'init' function.
-----*/
void SCH_Update(void)
{
    static UCHAR Tick = 1;
    //Note that an interrupt has occurred.
    Tick_count_G++;
    if(Tick)
    {
        IOCLR1 |= TICK_PIN;
        Tick = 0;
    }
    else
    {
        IOSET1 |= TICK_PIN;
        Tick = 1;
    }
    IOCLR1 |= TASK_INTERRUPT;
    IOSET1 |= TASK_INTERRUPT; //start tick
    TOIR = 0x01; //Reset interrupt flag.
}

/*-----*/
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)
{
    UCHAR Index;
    UCHAR Update_required = 0;

    // Need to check for a timer interuppt since this
    // function was last executed (in case idle mode is not being used)
    VICIntEnClr = 0x10; // Disable timer interrupt
    if (Tick_count_G > 0)
    {
        Tick_count_G--;
        Update_required = 1;
    }
    VICIntEnable = 0x10; // Re-enable timer interrupts

    while (Update_required)
    {
        // Go through the task array
        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
            {
                IOCLR1 |= TASK_MONI_PORT; //clear output pins
                IOSET1 |= (Index+1)<<16; //output task index
                (*SCH_tasks_G[Index].pTask)();
                IOCLR1 |= TASK_INTERRUPT;
                IOSET1 |= TASK_INTERRUPT;//start tick

                if (SCH_tasks_G[Index].Period != 0)
                    // 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;
                }
            }
    }
    }
    IOCLR1 |= TASK_MONI_PORT; //clear output pins
    VICIntEnClr = 0x10; // Disable timer interrupt
    if (Tick_count_G > 0)
    {
        IOSET1 |= OVERRUN_PIN;
        IOCLR1 |= TASK_INTERRUPT;
        IOSET1 |= TASK_INTERRUPT;//start tick
        Tick_count_G--;
        Update_required = 1;
    }
    else
    {
        IOCLR1 |= OVERRUN_PIN; //need response time
        Update_required = 0;
    }
    VICIntEnable = 0x10; // Re-enable timer interrupts
} //end of while

// The scheduler may enter idle mode at this point (if used)
SCH_Go_To_Sleep();
}
The rest are the same as original file.
/*-----END-----*/

```

The PSA code

This section contains the files of the PSA.

```

/*****
Project name: Scheduler agent

```

Project Description:

The project is to implement a Scheduler Agent, which is intended to provide a mechanism for monitoring the temporal behaviour of tasks in time-triggered embedded systems.

It is has the following function:

- 1) Monitor task execution time on the monitored processor and detect deviations from predicted WCET values.
- 2) Monitor the task execution sequence on the monitored processor and detect variations (e.g. a switch in task execution order).
- 3) Monitor scheduler correctness (e.g. scheduler tick interval change).
- 4) Perform other temporal analyses.

version 2.0 (SA automation tool)

```
-----/
/*****
/*****
Module name: Main.c
Module Description:
-----/
/*-----*/
Main.c
-----*/
#include "Main.h"
/*-----*/
main function
-----*/

int main (void)
{
    System_Init();
    LED_FLASH_Init();
    UART0_IO_Init();
    sEOS_Init();
    Safety_Agent_Init();
    Extint1_Init();

    while (1)
    {
        Process_Data();
        Analysis_Data();
        Report();
        UART0_IO_Update();
        //sEOS_Go_To_Sleep();
    }

    return 0;
}
/*-----END-----*/

/*-----*/
Simple_EOS.c
-----*/
#include "Main.h"

void Extint1_ISR(void)//Fiq
{
    VICIntEnClr = 0x00008000;           // Disable EINT1 Interrupt
    Sample_Task_Port(); //record pin
    EXTINT = 2;           // Clear EINT1 interrupt flag
```

```

    VICIntEnable = 0x00008000;          // Enable EINT1 Interrupt
}

void Extint1_Init(void)
{
    PINSEL0 |= 3 << 6;                  // Enable EINT1 on GPIO_0.3
    VICIntSelect |= 0x8000;             // Assign "Interrupt 15" to the FIQ category
    VPBDIV = 0;
    EXTMODE = 2;                        // Edge sensitive mode on EINT1
    EXTPOLAR = 2;                       // raising edge sensitive
    VPBDIV = 0;
    VPBDIV = 1;
    EXTINT = 2;                         // Clear EINT1 interrupt flag
    VICIntEnable = 0x00008000;          // Enable EINT1 Interrupt
}

/*-----*/
sEOS_Init()
Scheduler initialisation function.
-----
Here:
crystal = 12.0 MHz, PLL = 72.0 MHz ("cclk"),
pclk = cclk (Timer is incremented 48,000,000 times per second.)
So a count of 72 is a 1 us tick.
/*-----*/

void sEOS_Init(void)
{
    TOMRO = 3000000*CCLKRATIO; // 0x1E0Set the match register (MR0) to required value */
    TOMCR = 4; //the TC and PC will be stopped and TCR[0] will
}

void sEOS_Go_To_Sleep(void)
{
    PCON = 1;
}

/*-----END-----*/

/*-----*/
SA.c
retrieve informaion from main processor
/*-----*/

#include "Main.h"

//globe
#define NORMAL 0
#define OVERRUN_TASK 1
#define OVERRUN_TICK_CONTINUE 2

//Raw
Raw_Datast Raw_Data[20]; //recieve from MP
UCHAR Raw_Data_waiting_Index = 0;
UCHAR Raw_Data_Processed_Index = 0;

//temporal data (a tick)
UCHAR tTask_NO = 0;
UCHAR Last_tick_task_NO = 0;
UCHAR Last_tick_complete =0;
Temp_Datast TempTasks_Info_G[TASK_TOTAL_NO];
UCHAR Report_Flag = 0;

```

```

Errorst Error[6]; //store 6errors
UCHAR Error_Flag = 0;
ULONG Task0_Start_time = 0;
ULONG Next_Task0_Start_time = 0;

//long term data
ULONG MPTick = 0;
Datast Tasks_Info_G[TASK_TOTAL_NO+1];
UCHAR Error_NO = 0;

static UCHAR Old_Tick_Pin = TICK_PIN;
static UCHAR Old_Task_Pins = 0X00;
static UCHAR New_Tick_Pin;
static UCHAR New_Task_Pins;
static UCHAR MPTask_State = NORMAL;

//-----
void Safety_Agent_Init(void)
{
    //Set P1.16-24 as task monitor port for safety agent
    PINSEL2 &= ~0x00000004; //Set P1.16-25 as IO
    IODIR1 &= ~TASK_PORT; //Set P1.16-24 as input
    MPTask_State = NORMAL;
    //task 1
    Tasks_Info_G[1].Period = 1;
    Tasks_Info_G[1].Offset = 0; //start at tick 1
    Tasks_Info_G[1].Esti_Wcet = 11000*CCLKRATIO;
    //task 2
    Tasks_Info_G[2].Period = 3;
    Tasks_Info_G[2].Offset = 1;
    Tasks_Info_G[2].Esti_Wcet = 51000*CCLKRATIO;
    //task 3
    Tasks_Info_G[3].Period = 3;
    Tasks_Info_G[3].Offset = 2;
    Tasks_Info_G[3].Esti_Wcet = 26000*CCLKRATIO;
    MPTick = 0;
    Task_Order_Cacl(); //cacl first tick
}

void Sample_Task_Port(void)
{
    static UCHAR Tick_update = TICK_PIN; //TEST SHOULD BE OPPOSITE
    UCHAR New_Task_Port = (IO1PIN >>16) & TASK_PORT;
    UCHAR New_Tick = New_Task_Port & TICK_PIN;
    //UCHAR New_Task = New_Task_Port & TASK_MONI_PORT;

    Raw_Data[Raw_Data_waiting_Index].Time = TOTC;
    if(New_Tick != Tick_update)
    {
        TOTCR = 2; // Counter reset (Timer Counter Register)
        TOTCR = 1; // Counter enable (Timer Counter Register)
        Tick_update = New_Tick;
        LED_FLASH_Change_State();
    }
    Raw_Data[Raw_Data_waiting_Index].Port_Value = New_Task_Port;
    Raw_Data_waiting_Index++;
}

```

```

void Process_Overrun_Task(void)
{
if (New_Task_Pins==Old_Task_Pins)//the overrun task
{
if(New_Tick_Pin!= Old_Tick_Pin)TempTasks_Info_G[tTask_NO].tOverrun_tick++;
else
{
TempTasks_Info_G[tTask_NO].tTask_id = New_Task_Pins;
TempTasks_Info_G[tTask_NO].tEnd = Raw_Data[Raw_Data_Processed_Index].Time;
TempTasks_Info_G[tTask_NO].tOverrun_tick++;
tTask_NO++;
MPTask_State = OVERRUN_TICK_CONTINUE;
}
}
}

char Complete_Overrun_Tick(void)
{
UCHAR Overrun_End_Flag = Raw_Data[Raw_Data_Processed_Index].Port_Value&OVERRUN_PIN;
if(Overrun_End_Flag==OVERRUN_PIN)
{
Task0_Start_time = Next_Task0_Start_time;
Next_Task0_Start_time = Raw_Data[Raw_Data_Processed_Index].Time;
Last_tick_complete = 1;
Last_tick_task_NO = tTask_NO;
tTask_NO = 0;
MPTick++;
MPTask_State = NORMAL;
return 1;
}
else
{
return 0;
}
}

void Process_Normal_Task(void)
{
if (New_Tick_Pin != Old_Tick_Pin)//new tick
{
//check MP tick
if(Raw_Data[Raw_Data_Processed_Index].Time>(MPTICKINTERVAL*CCLKRATIO)) //150,000*48
86400000
{
Error[Error_Flag].Sch_tick = MPTick;
Error[Error_Flag].Tick = Raw_Data[Raw_Data_Processed_Index].Time/CCLKRATIO;//(us)
Error[Error_Flag].Type_code = MPTICKWRONG;
Error_Flag++;
}
if(New_Task_Pins==0) //normal tick
{
Last_tick_complete = 1;
Last_tick_task_NO = tTask_NO;
tTask_NO = 0;
Task0_Start_time = 0;
MPTick++;
Task0_Start_time = Next_Task0_Start_time;
Next_Task0_Start_time = 0;
}
}
}

```



```

else // there is a task overrun
{
    //check miss deadline
    Error[Error_Flag].Sch_tick = MPTick;
    Error[Error_Flag].Task_id = New_Task_Pins;
    Error[Error_Flag].Type_code = MISSDEADLINE;
    Error_Flag++;
    MPTask_State = OVERRUN_TASK;
}
} //END if (New_Task_Pins!= Old_Tick) //new tick
else //New_Tick_Pin == Old_Tick_Pin
{
    if(New_Task_Pins >0)
    {
        TempTasks_Info_G[tTask_NO].tTask_id = New_Task_Pins;
        TempTasks_Info_G[tTask_NO].tEnd = Raw_Data[Raw_Data_Processed_Index].Time;
        tTask_NO++;
    }
}
}

```

```

void Process_Data(void)
{
    char temp;
    if (Raw_Data_Processed_Index<Raw_Data_waiting_Index)
    {
        New_Tick_Pin = Raw_Data[Raw_Data_Processed_Index].Port_Value&TICK_PIN;
        New_Task_Pins = Raw_Data[Raw_Data_Processed_Index].Port_Value&TASK_MONI_PORT;

        if(MPTask_State==OVERRUN_TASK)Process_Overrun_Task();
        else if(MPTask_State==OVERRUN_TICK_CONTINUE)
        {
            temp =Complete_Overrun_Tick();
            if(!temp)Process_Normal_Task();//not finished
        }
        else if(MPTask_State==NORMAL)
        {
            Process_Normal_Task(); //normal
        }

        Old_Tick_Pin=New_Tick_Pin;
        Old_Task_Pins=New_Task_Pins;
        Raw_Data_Processed_Index++;
        if (Raw_Data_Processed_Index == Raw_Data_waiting_Index)
        {
            Raw_Data_waiting_Index = 0;
            Raw_Data_Processed_Index = 0;
        }
    } //END if (Raw_Data_Processed_Index<Raw_Data_waiting_Index)
}

```

```

void Analysis_Data(void)
{
    UCHAR i,temp;
    if (Last_tick_complete)//complete a tick
    {
        //duration tTask_NO
    }
}

```

```

    if(TempTasks_Info_G[0].tOverun_tick == 0)TempTasks_Info_G[0].tDuration =
TempTasks_Info_G[0].tEnd-Task0_Start_time;
    else
    {
        TempTasks_Info_G[0].tDuration =
TempTasks_Info_G[i].tOverun_tick*MPTICKINTERVAL*CCLKRATIO+TempTasks_Info_G[i].tEnd-
Task0_Start_time;
        TempTasks_Info_G[0].tOverun_tick =0;
    }
    for (i=1;i<Last_tick_task_NO; i++)
    {
        if(TempTasks_Info_G[i].tOverun_tick == 0)TempTasks_Info_G[i].tDuration =
TempTasks_Info_G[i].tEnd-TempTasks_Info_G[i-1].tEnd;
        else
        {
            TempTasks_Info_G[i].tDuration =
TempTasks_Info_G[i].tOverun_tick*MPTICKINTERVAL*CCLKRATIO+TempTasks_Info_G[i].tEnd-
TempTasks_Info_G[i-1].tEnd;
            TempTasks_Info_G[i].tOverun_tick =0;
        }
    }

    //check forWCET
    for (i=0;i<Last_tick_task_NO; i++)
    {
        temp = TempTasks_Info_G[i].tTask_id;

        if (TempTasks_Info_G[i].tDuration > Tasks_Info_G[temp].Actu_Wcet)
        {
            Tasks_Info_G[temp].Actu_Wcet = TempTasks_Info_G[i].tDuration;
        }
        //bcet
        if (TempTasks_Info_G[i].tDuration > Tasks_Info_G[temp].Esti_Wcet)
        {
            Error[Error_Flag].Sch_tick = MPTick-1;
            Error[Error_Flag].Task_id = TempTasks_Info_G[i].tTask_id;
            Error[Error_Flag].Type_code = OVERESTIW CET;
            Error_Flag++;//over estim wcet
        }
    }
    //check for sequence
    Task_Sequence_Check();
    Task_Order_Cacl();
    Last_tick_complete =0;
    Report_Flag++;
}

/*-----END-----*/

/*-----*
Sequence_check.c
retrieve informaion from main processor
-----*/
#include "Main.h"

UCHAR Task_Sequence[TASK_TOTAL_NO];
UCHAR Task_Sequence_Index = 0;

```

```

/*-----*/
    Task_Order_Cacl()
    Task_Order_Cacl.
    //TTC, note offset=MPoffset+1
/*-----*/

void Task_Order_Cacl(void)
{
    UCHAR i;

    Task_Sequence_Index = 0;
    for (i = 1; i<= TASK_TOTAL_NO; i++)
    {
        if(MPTick>Tasks_Info_G[i].Offset)
        {
            if ((MPTick - Tasks_Info_G[i].Offset-1) % Tasks_Info_G[i].Period == 0)
            {
                Task_Sequence[Task_Sequence_Index] = i;
                Task_Sequence_Index++;
            }
        }
    }
}

/*
*/
void Task_Sequence_Check(void)
{
    UCHAR i,j;

    if(Last_tick_task_NO>Task_Sequence_Index)//returned
    {
        j =0;
        for (i = 0; i< Task_Sequence_Index; i++)
        {
            if (Task_Sequence[i] !=TempTasks_Info_G[j].tTask_id )
            {
                if (Task_Sequence[i-1] == TempTasks_Info_G[j].tTask_id)//returned
                {
                    Error[Error_Flag].Sch_tick = MPTick-1;
                    Error[Error_Flag].Task_id = TempTasks_Info_G[j].tTask_id;
                    Error[Error_Flag].Type_code = TASKRETURNED;
                    Error_Flag++;
                    j++;
                    i--;
                }
                else
                {
                    Error[Error_Flag].Sch_tick = MPTick-1;
                    Error[Error_Flag].Task_id = TempTasks_Info_G[i].tTask_id;
                    Error[Error_Flag].Type_code = SEQUENCEWRONG;
                    Error_Flag++;
                    j++;
                }
            }
            else j++;
        }
        if(j<Last_tick_task_NO)

```

```

    {
        for(i = j; i<Last_tick_task_NO;i++)
        {
            Error[Error_Flag].Sch_tick = MPTick-1;
            Error[Error_Flag].Task_id = TempTasks_Info_G[i].tTask_id;
            Error[Error_Flag].Type_code = SEQUENCEWRONG;
            Error_Flag++;
        }
    }
}
else if(Last_tick_task_NO<Task_Sequence_Index)//missed
{
    j = 0;
    for (i = 0; i< Last_tick_task_NO; i++)
    {
        if (TempTasks_Info_G[i].tTask_id != Task_Sequence[j])
        {
            if((TempTasks_Info_G[i].tTask_id == Task_Sequence[j+1])&&((j+1)<Task_Sequence_Index))
            {
                Error[Error_Flag].Sch_tick = MPTick-1;
                Error[Error_Flag].Task_id = Task_Sequence[i];
                Error[Error_Flag].Type_code = TASKMISSED;
                Error_Flag++;
                j+=2;
                //i--;
            }
            else
            {
                Error[Error_Flag].Sch_tick = MPTick-1;
                Error[Error_Flag].Task_id = TempTasks_Info_G[i].tTask_id;
                Error[Error_Flag].Type_code = SEQUENCEWRONG;
                Error_Flag++;
                j++;
            }
        }
        else j++;
    }
    if(j<Task_Sequence_Index)
    {
        for(i = j; i<Task_Sequence_Index;i++)
        {
            Error[Error_Flag].Sch_tick = MPTick-1;
            Error[Error_Flag].Task_id = Task_Sequence[i];
            Error[Error_Flag].Type_code = TASKMISSED;
            Error_Flag++;
        }
    }
}
else //==
{
    if((TempTasks_Info_G[i].tTask_id == Task_Sequence[i+1])&&((i+1)<Task_Sequence_Index))
    {
        Error[Error_Flag].Sch_tick = MPTick-1;
        Error[Error_Flag].Task_id = TempTasks_Info_G[i].tTask_id;
        Error[Error_Flag].Type_code = SEQUENCEWRONG;
        Error_Flag++;
    }
}
}
}

```

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

The adapted target code for ASA

To add the IM into the target, the scheduler file needs to be modified.

```
/*-----*/
sch_2100.c (v1.00)
Port of "PTTES" scheduler to ARM (LPC21xx)
This version is modified for passive safety agent
/*-----*/
#include "Main.h"
#define TASK_MONI_PORT 0x000F0000//define P1.16-19 as task pins
#define TICK_PIN 0x00800000//define P1.23 as tick pin
#define OVERRUN_PIN 0x00400000//define P1.22 as overrun pin
#define TASK_INTERRUPT 0x00200000//define P1.21 as tick pin

// ----- Public variable definitions -----
UCHAR Error_code_G;

// ----- Private variable definitions -----
static sTask SCH_tasks_G[SCH_MAX_TASKS]; //Tasks array
static UINT Tick_count_G = 0;

// ----- Private function prototypes -----
static void SCH_Go_To_Sleep(void);

/*-----*/
SCH_Init(unsigned long tick)
Scheduler initialisation function. Prepares scheduler
data structures and sets up timer interrupts at required rate.
Must be called before using the scheduler.
Define CCLKRATIO in Sys_Setup.h file
the timer valude is the tick*CCLKRATIO(us)
/*-----*/
void SCH_TTC_Init(unsigned long tick)
{
    UCHAR i;

    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;
    TOMR0 = tick*CCLKRATIO; // the match register (MR0) to required value
    TOMCR = 0x03; // Interrupt on match, and automatically restart counter
    VICIntSelect |= 0x10; // Assign "Interrupt 4" to the FIQ category
    //set the task pins as GPIO
    PINSEL2 &= ~0x00000008;//set task pins
    PINSEL2 &= ~0x00000008;//set tick pin
    PINSEL2 &= ~0x00000008;//set overrun pin
    PINSEL2 &= ~0x00000008;//set interrput pin
```

```

//set the task pins as output
IODIR1 |= 0x000F0000;//output//set task pins
IODIR1 |= 0x00800000;//output//set tick pin
IODIR1 |= 0x00200000;//output//set overrun pin
IODIR1 |= 0x00400000;//output//set interrupt pin
//clear task pins
IOCLR1 |= ~0x000F0000;//clear//set task pins
IOCLR1 |= ~0x00800000;//clear//set tick pin
IOCLR1 |= ~0x00400000;//clear//set overrun pin
IOCLR1 |= ~0x00200000;//clear//set interrupt pin
VICIntEnable |= 0x10; // Enable this interrupt
}

/*-----*/
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)
{
    IOSET1 |= TICK_PIN;//start tick
    TOTCR |= 0x01; // Counter enable (Timer Counter Register)
}

/*-----*/
SCH_Update()
This is the scheduler ISR. It is called at a rate
determined by the timer settings in the 'init' function.
Note that an interrupt has occurred.
/*-----*/
void SCH_Update(void)
{
    static unsigned char Tick= 1;
    Tick_count_G++;
    if(Tick)
    {
        IOCLR1 |= TICK_PIN;
        Tick = 0;
    }
    else
    {
        IOSET1 |= TICK_PIN;
        Tick = 1;
    }
    IOCLR1 |= TASK_INTERRUPT;
    IOSET1 |= TASK_INTERRUPT;
    TOIR = 0x01; //Reset interrupt flag.
}

/*-----*/
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)

```

```

{
    UCHAR Index;
    UCHAR Update_required = 0;

    // Need to check for a timer interuppt since this
    // function was last executed (in case idle mode is not being used)
    VICIntEnClr = 0x10; // Disable timer interrupt
    if (Tick_count_G > 0)
    {
        Tick_count_G--;
        Update_required = 1;
    }
    VICIntEnable = 0x10; // Re-enable timer interrupts

    while (Update_required)
    {
        // Go through the task array
        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
                {
                    IOSET1 |= (Index + 1) << 16; //ouput task index
                    (*SCH_tasks_G[Index].pTask)(); // Run the task
                    IOCLR1 |= TASK_MONI_PORT; //clear output pins
                    if (SCH_tasks_G[Index].Period != 0)
                    // 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;
                }
            }
        }
        VICIntEnClr = 0x10; // Disable timer interrupt
        if (Tick_count_G > 0)
        {
            Tick_count_G--;
            IOSET1 |= OVERRUN_PIN; //overflow happened
            Update_required = 1;
        }
        else
        {
            IOCLR1 |= OVERRUN_PIN; //clear overrun pin
            Update_required = 0;
        }
        VICIntEnable = 0x10; // Re-enable timer interrupts
    } //end of while

    // The scheduler may enter idle mode at this point (if used)
    SCH_Go_To_Sleep();
}

```

The rest are the same as original file.

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

The ASA code

This section contains the files of the ASA.

```
/*-----
Project name: Scheduler agent

Project Description:
    The project is to implement a Scheduler Agent, which is
    intended to provide a mechanism for monitoring the temporal
    behaviour of tasks in time-triggered embedded systems.
    It is has the following function:
    1) Monitor task execution time on the monitored processor and
    detect deviations from predicted WCET values.
    2) Monitor the task execution sequence on the monitored processor
    and detect variations (e.g. a switch in task execution order).
    3) Monitor scheduler correctness (e.g. scheduler tick interval change).
    4) Perform other temporal analyses.

    version 2.1 (SA automation tool)
-----*/

/*-----
Module name: Main.c
Module Description:
-----*/

//Add all #includes here
#include "Main.h"

/*-----*
int main (void)
-----*/

int main (void)
{
    System_Init();
    LED_FLASH_Init();
    UART0_IO_Init(115200);
    sEOS_Init(3000000);
    Scheduler_Agent_Init();
    Extint1_Init();

    while (1)
    {
        Retrieve_Task_Info();
        Analysis_Data();
        Report();
        UART0_IO_Update();
        UART0_IO_Update();
        // sEOS_Go_To_Sleep();
    }
}
```



```

    return 0;
}

/*-----END-----*/

/*-----*/
Simple_EOS.c
/*-----*/
#include "Main.h"

void Extint1_ISR(void)//Fiq
{
    VICIntEnClr = 0x00008000;           // Disable EINT1 Interrupt
    Sample_Task_Port(); //record pin
    EXTINT = 2;           // Clear EINT1 interrupt flag
    VICIntEnable = 0x00008000;         // Enable EINT1 Interrupt
}

void Extint1_Init(void)
{
    ULONG Tick_Value = IO1PIN & 0x00800000; //TICK_PIN
    PINSELO |= 3 << 6;           // Enable EINT1 on GPIO_0.3
    VICIntSelect |= 0x8000;       // Assign "Interrupt 15" to the FIQ category
    VPBDIV = 0;
    EXTMODE = 2;                 // Edge sensitive mode on EINT1
    EXTPOLAR = 2;                // raising edge sensitive
    VPBDIV = 0;
    VPBDIV = 1;
    EXTINT = 2;           // Clear EINT1 interrupt flag

    while (Tick_Value==0)// TICK_PIN
    {
        Tick_Value = IO1PIN & 0x00800000;
    }

    VICIntEnable = 0x00008000;       // Enable EINT1 Interrupt
}

/*-----*/

sEOS_Init()
Scheduler initialisation function.
-----
Here:
crystal = 12.0 MHz, PLL = 72.0 MHz ("cclk"),
pclk = cclk (Timer is incremented 48,000,000 times per second.)
So a count of 72 is a 1 us tick.

/*-----*/

void sEOS_Init(ULONG us)
{
    TOMR0 = us*CCLKRATIO; //Set the match register (MR0) to required value */
    TOMCR = 4; //the TC and PC will be stopped and TCR[0] will
}

void Delay_Timer1_us_Times(ULONG Multiplier)
{
    TIMR0 = CCLKRATIO * Multiplier; //Set the match register (MR0) to required value
}

```

```

    TIMCR = 6; //the TC and PC will be stopped and TCR[0] will
    T1TCR |= 1; // Counter enable (Timer Counter Register)
                // be set to 0 if MRO matches the TC
    while(((T1TCR & 1) == 1));
    T1TCR = 2; // Counter reset (Timer Counter Register)
    T1TCR = 0; // Counter disable (Timer Counter Register)
}

void sEOS_Go_To_Sleep(void)
{
    PCON = 1;
}

/*-----END-----*/

/*****
Module name: SA.c
Module Description:
    The module is used for Scheduler Agent to
    retrieve informaion from main processor.
    (in this version, SA uses external interrupt to
    trigger the SA, and sample only at the key points)
*****/
#include "Main.h"
#define CHECK_POINT 3000 //50*60=3000

//globe
#define NORMAL 0
#define OVERRUN_TASK 1
#define OVERRUN_TICK_CONTINUE 2

UCHAR tTask_NO;

//temporal data (a tick)
Temp_Datast TempTasks_Info_G[50]; //TASK_TOTAL_NO
UCHAR TempTasks_Info_Index = 0; //for overrun
UCHAR TempTasks_Info_Processed_Index = 0; //for overrun
UCHAR TempTasks_Info_NO_G[5]; //for overrun
UCHAR TempTasks_Info_NO_Waiting_Index = 1; //for overrun
UCHAR TempTasks_Info_NO_Processed_Index = 1;

UCHAR Report_Flag = 0;
Errorst Error[6]; //store 6errors
UCHAR Error_Flag = 0;
UCHAR Overrun_Count = 0;
UCHAR Overrun_Tick = 0;

//long term data
ULONG MPTick = 0;
Datast Tasks_Info_G[TASK_TOTAL_NO+1];
UCHAR Error_NO = 0;

static UCHAR MPTick_update = 0;

//-----
void Scheudler_Agent_Init(void)
{
    //Set P1.16-24 as task monitor port for safety agent
    PINSEL2 &= ~0x00000004; //Set P1.16-25 as IO

```

```

    IODIR1 &= ~(TASK_PORT<<TASK_PORT_LOW); //Set P1.16-24 as input

    //add task info here
    // //task 1
        Tasks_Info_G[1].Period = 1;
        Tasks_Info_G[1].Offset = 0;//start at tick 1
        Tasks_Info_G[1].Esti_Wcet = 11000;// 11000
        Tasks_Info_G[1].Type = 1;
    // //task 2
        Tasks_Info_G[2].Period = 3;
        Tasks_Info_G[2].Offset = 1;
        Tasks_Info_G[2].Esti_Wcet = 51000; //51000
        Tasks_Info_G[2].Type = 0;
    // //task 3
        Tasks_Info_G[3].Period = 3;
        Tasks_Info_G[3].Offset = 2;
        Tasks_Info_G[3].Esti_Wcet = 26000;//26
        Tasks_Info_G[3].Type = 1;
    MPTick = 0;
}

void Sample_Task_Port(void)
{
    ULONG Temp =T0TC/CCLKRATIO;
    UCHAR New_Task_Pins = (IO1PIN >>16)&TASK_MONI_PORT;
    //Temp =T0TC/CCLKRATIO;
    TOTCR = 2; // Counter reset (Timer Counter Register)
    TOTCR = 1; // Counter enable (Timer Counter Register)
    MPTick_update++;
    LED_FLASH_Change_State();

    //check MP tick
    if(Temp>MPTICKINTERVAL) //150,000*48
    {
        Error[Error_Flag].Sch_tick = MPTick;
        Error[Error_Flag].Tick = Temp;
        Error[Error_Flag].Type_code = MPTICKWRONG;
        Error_Flag++;
    }
    //check miss deadline
    if (New_Task_Pins > 0) //overrun task
    {
        Overrun_Count++;
        Overrun_Tick=1;
        //Overrun_task_Count ++;
        Error[Error_Flag].Sch_tick = MPTick+1;
        Error[Error_Flag].Task_id = New_Task_Pins;
        Error[Error_Flag].Type_code = MISSDEADLINE;
        Error_Flag++;
    }
}

void Retrieve_Task_Info(void)
{
    UCHAR Tick_Finished =1;
    UCHAR Update_Required =0;
    UCHAR Overrun_End_Flag;
    UCHAR New_id;

```

```

VICIntEnClr = 0x00008000;           // Disable EINT1 Interrupt
if (MPTick_update > 0)
{
    MPTick_update--;
    Update_Required = 1;
    Tick_Finished =1;
}
VICIntEnable = 0x00008000;         // Enable EINT1 Interrupt

while(Update_Required)
{
    while (Tick_Finished)
    {
        Delay_Timer1_lus_Times(20); //overhead

        if (Overrun_Tick)
        {
            Overrun_End_Flag = (IO1PIN >>16)&OVERRUN_PIN;
            if(Overrun_End_Flag==OVERRUN_PIN) //check overrun fished pin
            {
                Overrun_Tick =0;
                Tick_Finished=0;
            }
        }
        else
        {
            New_id = (IO1PIN >>16)&TASK_MONI_PORT; // 0x0f
            if (New_id>0)
            {
                TempTasks_Info_G[TempTasks_Info_Index].tTask_id = New_id;
                TempTasks_Info_Index++;
                TempTasks_Info_G[TempTasks_Info_Index-1].tStart = T0TC;
                if(Tasks_Info_G[New_id].Type)//fixed duration
                {
                    Delay_Timer1_lus_Times(Tasks_Info_G[New_id].Esti_Wcet - CHECK_POINT); //overhead
                }
                while(New_id == TempTasks_Info_G[TempTasks_Info_Index-1].tTask_id)
                {
                    //Delay_Timer1_lus_Times(100);
                    New_id = (IO1PIN >>16)&TASK_MONI_PORT; //TASK_MONI_PORT
                }
                TempTasks_Info_G[TempTasks_Info_Index-1].tEnd = T0TC;
                TempTasks_Info_G[TempTasks_Info_Index-1].tOverrun_tick = Overrun_Count;
                Overrun_Count = 0;
            }
            else
            {
                TempTasks_Info_G[TempTasks_Info_Index].tEnd = T0TC;
                TempTasks_Info_G[TempTasks_Info_Index].tOverrun_tick = Overrun_Count;
                Overrun_Count = 0;
                Tick_Finished=0;
            }
        }
    } //END while (Tick_Finished)
} //completed

TempTasks_Info_NO_G[TempTasks_Info_NO_Waiting_Index++] = TempTasks_Info_Index;

VICIntEnClr = 0x00008000;           // Disable EINT1 Interrupt

```

```

    if (MPTick_update > 0)
    {
        MPTick_update--;
        Update_Required = 1;
        Tick_Finished =1;
    }
    else
    {
        Update_Required = 0;
    }
    VICIntEnable = 0x00008000;           // Enable EINT1 Interrupt
} //END while(Update_Required)

}

void Analysis_Data(void)
{
    UCHAR i,temp;
    if (TempTasks_Info_NO_Waiting_Index>TempTasks_Info_NO_Processed_Index)//complete a tick
    {
        MPTick++;
        tTask_NO =TempTasks_Info_NO_G[TempTasks_Info_NO_Processed_Index] -
        TempTasks_Info_NO_G[TempTasks_Info_NO_Processed_Index-1];

        //duration
        //task1
        if(TempTasks_Info_G[TempTasks_Info_Processed_Index].tOverrun_tick == 0)
        {
            if(TempTasks_Info_G[TempTasks_Info_Processed_Index].tStart >
            500*CCLKRATIO)TempTasks_Info_G[TempTasks_Info_Processed_Index].tDuration =
            TempTasks_Info_G[TempTasks_Info_Processed_Index].tEnd-
            TempTasks_Info_G[TempTasks_Info_Processed_Index].tStart;
            else TempTasks_Info_G[TempTasks_Info_Processed_Index].tDuration =
            TempTasks_Info_G[TempTasks_Info_Processed_Index].tEnd;
        }
        else
        {
            if(TempTasks_Info_G[TempTasks_Info_Processed_Index].tStart >
            500*CCLKRATIO)TempTasks_Info_G[TempTasks_Info_Processed_Index].tDuration =
            TempTasks_Info_G[TempTasks_Info_Processed_Index].tOverrun_tick*MPTICKINTERVAL*CCLKRATIO+TempTasks_Info_G[TempTasks_Info_Processed_Index].tEnd-
            TempTasks_Info_G[TempTasks_Info_Processed_Index].tStart;
            else TempTasks_Info_G[TempTasks_Info_Processed_Index].tDuration =
            TempTasks_Info_G[TempTasks_Info_Processed_Index].tOverrun_tick*MPTICKINTERVAL*CCLKRATIO+TempTasks_Info_G[TempTasks_Info_Processed_Index].tEnd;
            //TempTasks_Info_G[TempTasks_Info_Processed_Index].tDuration =
            TempTasks_Info_G[TempTasks_Info_Processed_Index].tOverrun_tick*MPTICKINTERVAL*CCLKRATIO+TempTasks_Info_G[TempTasks_Info_Processed_Index].tEnd-
            TempTasks_Info_G[TempTasks_Info_Processed_Index].tStart;
            TempTasks_Info_G[TempTasks_Info_Processed_Index].tOverrun_tick =0;
        }
        for (i=TempTasks_Info_Processed_Index+1;i<(tTask_NO+TempTasks_Info_Processed_Index); i++)
        {
            if(TempTasks_Info_G[i].tOverrun_tick == 0)TempTasks_Info_G[i].tDuration =
            TempTasks_Info_G[i].tEnd-TempTasks_Info_G[i-1].tEnd;
            else
            {

```

```

        TempTasks_Info_G[i].tDuration =
TempTasks_Info_G[i].tOverrun_tick*MPTICKINTERVAL*CCLKRATIO+TempTasks_Info_G[i].tEnd-
TempTasks_Info_G[i-1].tEnd;
        TempTasks_Info_G[i].tOverrun_tick =0;
    }
}

//check forWCET
for (i=TempTasks_Info_Processed_Index;i<tTask_NO; i++)
{
    temp = TempTasks_Info_G[TempTasks_Info_Processed_Index+i].tTask_id;

    if (TempTasks_Info_G[TempTasks_Info_Processed_Index+i].tDuration >
Tasks_Info_G[temp].Actu_Wcet)
    {
        Tasks_Info_G[temp].Actu_Wcet =
TempTasks_Info_G[TempTasks_Info_Processed_Index+i].tDuration;
    }
    //bcet
    if (TempTasks_Info_G[TempTasks_Info_Processed_Index+i].tDuration >
Tasks_Info_G[temp].Esti_Wcet*CCLKRATIO)
    {
        Error[Error_Flag].Sch_tick = MPTick;//-1
        Error[Error_Flag].Task_id = temp;
        Error[Error_Flag].Type_code = OVERESTIWCET;
        Error_Flag++;//over estim wcet
    }
}
//check for sequence
Task_Order_Cacl();
Task_Sequence_Check();
if(MPTick <=10001)Report_Duration(); //
TempTasks_Info_NO_Processed_Index++;
TempTasks_Info_Processed_Index += tTask_NO;
if(TempTasks_Info_NO_Processed_Index==TempTasks_Info_NO_Waiting_Index)
{
    TempTasks_Info_NO_Processed_Index =1;
    TempTasks_Info_NO_Waiting_Index =1;
    TempTasks_Info_Index=0;
    TempTasks_Info_Processed_Index=0;
}
}
}

/*-----END-----*/

/*-----*
Sequence_check.c
retrieve informaion from main processor
-----*/
#include "Main.h"

UCHAR Task_Sequence[TASK_TOTAL_NO];
UCHAR Task_Sequence_Index = 0;

/*-----*
Task_Order_Cacl()

```

```

    Task_Order_Cacl.
    //TTC, note offset=MPoffset+1
-----*/
void Task_Order_Cacl(void)
{
    UCHAR i;

    Task_Sequence_Index = 0;
    for (i = 1; i<= TASK_TOTAL_NO; i++)
    {
        if(MPTick>Tasks_Info_G[i].Offset)
        {
            if ((MPTick - Tasks_Info_G[i].Offset-1) % Tasks_Info_G[i].Period == 0)
            {
                Task_Sequence[Task_Sequence_Index] = i;
                Task_Sequence_Index++;
            }
        }
    }
}

/*
*/
void Task_Sequence_Check(void)
{
    UCHAR i,j;

    if(tTask_NO>Task_Sequence_Index)//returned
    {
        j =0;
        for (i = 0; i< Task_Sequence_Index; i++)
        {
            if (Task_Sequence[i] !=TempTasks_Info_G[j].tTask_id )
            {
                if (Task_Sequence[i-1] == TempTasks_Info_G[j].tTask_id)//returned
                {
                    Error[Error_Flag].Sch_tick = MPTick;
                    Error[Error_Flag].Task_id = TempTasks_Info_G[j].tTask_id;
                    Error[Error_Flag].Type_code = TASKRETURNED;
                    Error_Flag++;
                    j++;
                    i--;
                }
            }
            else
            {
                Error[Error_Flag].Sch_tick = MPTick;
                Error[Error_Flag].Task_id = TempTasks_Info_G[i].tTask_id;
                Error[Error_Flag].Type_code = SEQUENCEWRONG;
                Error_Flag++;
                j++;
            }
        }
        else j++;
    }
    if(j<tTask_NO)
    {
        for(i = j; i<tTask_NO;i++)

```

```

        {
            Error[Error_Flag].Sch_tick = MPTick;
            Error[Error_Flag].Task_id = TempTasks_Info_G[i].tTask_id;
            Error[Error_Flag].Type_code = SEQUENCEWRONG;
            Error_Flag++;
        }
    }
}
else if(tTask_NO<Task_Sequence_Index)//missed
{
    j = 0;
    for (i = 0; i< tTask_NO; i++)
    {
        if (TempTasks_Info_G[i].tTask_id != Task_Sequence[j])
        {
            if((TempTasks_Info_G[i].tTask_id == Task_Sequence[j+1])&&((j+1)<Task_Sequence_Index))
            {
                Error[Error_Flag].Sch_tick = MPTick;
                Error[Error_Flag].Task_id = Task_Sequence[i];
                Error[Error_Flag].Type_code = TASKMISSED;
                Error_Flag++;
                j+=2;
                //i--;
            }
            else
            {
                Error[Error_Flag].Sch_tick = MPTick;
                Error[Error_Flag].Task_id = TempTasks_Info_G[i].tTask_id;
                Error[Error_Flag].Type_code = SEQUENCEWRONG;
                Error_Flag++;
                j++;
            }
        }
        else j++;
    }
    if(j<Task_Sequence_Index)
    {
        for(i = j; i<Task_Sequence_Index;i++)
        {
            Error[Error_Flag].Sch_tick = MPTick;
            Error[Error_Flag].Task_id = Task_Sequence[i];
            Error[Error_Flag].Type_code = TASKMISSED;
            Error_Flag++;
        }
    }
}
else //==
{
    if((TempTasks_Info_G[i].tTask_id == Task_Sequence[i+1])&&((i+1)<Task_Sequence_Index))
    {
        Error[Error_Flag].Sch_tick = MPTick;
        Error[Error_Flag].Task_id = TempTasks_Info_G[i].tTask_id;
        Error[Error_Flag].Type_code = SEQUENCEWRONG;
        Error_Flag++;
    }
}
}

/*-----END-----*/

```