University of Leicester

Design and evaluation of flexible time-triggered task schedulers for dynamic control applications

Submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

Musharraf Ahmed Hanif

Department of Engineering
University of Leicester
Leicester, United Kingdom

July 2012

Design and evaluation of flexible time-triggered task schedulers for dynamic control applications

by

Musharaf Ahmed Hanif

Abstract

A statically-scheduled time-triggered (TT) software architecture demonstrates very predictable patterns of temporal behaviour and is – therefore – widely considered to be an appropriate platform for many high integrity and safety-critical embedded applications. However, there remains an important class of highly dynamic control systems for which it is considered that TT architectures are not a good match and for which the use of "event triggered" (ET) designs is usually preferred. These applications include the control systems for internal combustion engines, brushless DC motors and synchronous AC motors. The aim of the research project presented in this thesis was to explore ways in which a static TT architecture could be adapted in order to better meet the requirements of such highly-dynamic control systems.

The project had three main outcomes.

The first project outcome was that a novel "flexible TT architecture" was developed. This architecture differs significantly from conventional TT designs in that – during the system operation – only the timing of the next system interrupt is known in advance (that is, the timing of subsequent interrupts is unknown). This allows for considerable flexibility in the task scheduling while retaining most of the features that make static TT approaches attractive.

The second project outcome was that two novel schedulers were designed and implemented, in order to demonstrate (by means of an "existence proof") that it was possible to construct a practical implementation of the flexible TT architecture.

The third outcome from this project was that a comprehensive evaluation of the flexible TT architecture and the associated scheduler implementations was carried by means of two representative case studies. The case studies involved engine synchronisation and control of a brushless DC motor (BLDCM). In the engine synchronisation case study, the flexible TT architecture was shown to be a viable alternative to ET in conditions where a static TT was unable to cope with the system demands. In the BLDCM case study, while both static TT and flexible TT were viable alternatives, the flexible TT was able to provide similar levels of performance to the static TT solution at a fraction of the resource usage.

I dedicate this thesis to my daughter Eesha,
my wife Sara Alvi

Acknowledgements

The work presented in this thesis is supported by the UK Government (ORSAS Award) and TTE Systems (studentship).

I would like to take this opportunity to express my heartfelt gratitude to my supervisor, Prof. Michael J. Pont. I highly appreciate his guidance and encouragement throughout the course of this research.

I would also like to thank Dr. Michael Short for his advice and guidance on scheduling theory.

My thanks also go to the members of the Embedded Systems Laboratory for their support and the manly lively discussions that we had.

I would also like to thank Mr. Paul Williams for his guidance and support with the Rover engine test bed.

Finally, I would like to thank my wife and family for their support and trust in me.

Table of Contents

List of Figures

List of Flowcharts

List of Tables

List of Related Publications

(Some of the contents of this thesis have been adapted from a paper that was published)

Hanif, M.A., Pont, M.J. and Ayavoo, D. (2008) "Implementing a simple but flexible time triggered architecture for practical deeply embedded applications", In the proceedings of the 4th UK Embedded Forum, September 2008, Southampton, UK.

List of Abbreviations

AC	Alternating current
BCET	Best case execution time
BDC	Bottom dead centre
BLDCM	Brushless DC motor
CPU	Central processing unit
DC	Direct current
DM	Deadline monotonic
DMA	Direct memory access
DOHC	Double overhead cam
ECU	Engine control unit
EDF	Earliest deadline first
ET	Event-triggered
FSM	Finite state machine
FTP	Federal Test Procedure
ISR	Interrupt service routine
LANF	Los Angeles Non Freeway
LCM	Least common multiple
LLF	Lowest laxity first
LOC	Lines of code
MCFTC	Multi Cycle multiphase with Fixed Tick periods Cooperative
MCVTC	Multi Cycle multiphase with Variable Tick period Cooperative
MLF	Minimum laxity first
MUF	Maximum urgency first
NYNF	New York Non Freeway
PC	Personal computer
PID	Proportional integral derivative
RM	Rate monotonic
SCFTC	Single Cycle multiphase with Fixed Tick periods Cooperative
SCVTC	Single Cycle multiphase with Variable Tick period Cooperative
SI	Spark ignited

TDC	Top dead centre
TT	Time-triggered
TTC	Time-triggered cooperative scheduler
TTH	Time-triggered hybrid scheduler
TTMPC	Time-triggered multi phase cooperative scheduler
TTMPVRH	Time-triggered multi phase with variable rate hybrid scheduler
TTP	Time-triggered pre-emptive scheduler
TTSA1	Time-triggered scheduling algorithm 1
TTSA2	Time-triggered scheduling algorithm 2
TTxC	Any cooperative time-triggered scheduler
WCET	Worst case execution time

1 Introduction

1.1 Embedded Systems

The term "embedded system" is used to refer to a wide class of electronic systems that work to help make our life more convenient and safe. A general definition for these systems is given by Ganssle and Barr as:

> *"A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. In some cases, embedded systems are part of a larger system or product, as in the case of an antilock braking system in a car."* (Ganssle, Barr 2003)

Many embedded systems fall under the general category of "real-time systems" which are defined by Laplante as:

> *"A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness."* (Laplante 1997)

Embedded systems serve in roles ranging from improving our quality of life (e.g. automatic washing machines, digital set top boxes, mobile phones and music players, power windows and central locking in a car, etc.) to safety-critical systems (e.g. anti-lock brakes, airbags, medical life support systems, aircraft engine control units, etc.). The failure of a real-time embedded system in a non-safety-critical role results in annoyance and inconvenience. However, for real-time systems being used in safety-critical applications, it is essential that they continue to operate reliably between scheduled maintenance activities, as a failure to do so can have serious consequences, including loss of life.

1.1.1 System architectures for embedded applications

Real-time systems are usually categorized on the basis of the mechanism used to run (or "release" or "trigger") tasks (Kopetz 1991):

1. Time-triggered (TT) or "clock driven" architecture runs tasks based on their temporal criteria (e.g. period, initial delay, etc.) (Kopetz 1991, Liu 2000). TT systems tend to use a timer interrupt to manage the task executions (Liu 2000).

2. Event-triggered (ET) or "event driven" architecture runs tasks in response to internal and external events (Kopetz 1991, Liu 2000, Stewart 2001). An ET system can be implemented directly using interrupt service routines (ISR) or by using a sporadic task server (Stewart 2001).

System architectures can also be categorized by taking into account whether scheduling decisions are made at "design time" or at "run time" (Xu, Parnas 2000, Locke 1992):

1. Static task scheduling systems: the tasks are executed in a pre-determined order set at design time (Locke 1992, Fidge 2002). Typically, research referring to such systems assumes the use of periodic tasks with launch times (for the whole of the "major cycle"; i.e. the period after which the whole cycle repeats itself) stored in a suitable lookup table (Baker, Shaw 1988). In these systems (under normal operating conditions) both the task execution orders and the times at which the tasks are released for execution will be fixed (Locke 1992). By definition, only time-triggered systems can be statically scheduled.

2. Dynamic task scheduling systems: the release time of at least one of the tasks in the system will be determined at run time (Fidge 2002). The dynamic release is done either based on the temporal criteria (e.g. tasks are released periodically based on their periods and initial offsets) or in response to the occurrence of a

software or hardware event (such as the pressing of a switch or the arrival of a message on a communication bus at a time which is not precisely known in advance) (Fidge 2002). The handling of such event may, in turn, have an impact on the processing of other tasks in the system (including periodic tasks, which may suffer from "release jitter").

Scheduling algorithms can also be categorised on the basis of the task execution environment (Pont 2001):

1. In the cooperative execution environment, all tasks are allowed to run to completion without being interrupted by another task (Pont 2001).
2. In the pre-emptive execution environment, the execution of a task can be interrupted by the scheduler to run another task (Pont 2001).

Dynamic scheduling is supported by a great majority of commercial real-time operating systems (RTOSs), including but not limited to VxWorks, LynxOS, µC/OS-II, RT Linux, CHIMERA II, etc. (Stewart, Khosla 1991, Barr 2003). It is seen by many as the "standard" architecture for most embedded real-time systems, as indicated by the level of research being done on dynamic real time systems (e.g. (Liu, Layland 1973, Lehoczky, Sha et al. 1989, Jeffay, Stanat et al. 1991, Stewart, Khosla 1991, Locke 1992, Audsley, Tindell et al. 1993, Spuri, Buttazzo 1996, Liu 2000, Sha, Abdelzaher et al. 2004, Buttazzo 2005a, Buttazzo 2005b, Short 2010)). Incidents like NASA's use of priority driven asynchronous executive on the space shuttle (Martin 1994) and the fixed priority dynamic scheduler in the International Space Station's Freedom module (Sha, Abdelzaher et al. 2004) are also indicative of the present focus in this area.

1.1.2 Desired architecture for high reliability applications

While most of the real world systems tend to have a mixture of sporadic and periodic tasks (Xu, Parnas 2000, Xu 2003), a complex system with only periodic tasks (i.e. a time-triggered system) is easier to predict and analyse (Xu 2003).

In addition to this, it is generally argued that statically scheduled time-triggered systems offer more predictable behaviour than equivalent event-triggered designs, but at a price of reduced flexibility and increased design effort (Locke 1992, Fidge 2002). If all the timing parameters of all tasks (periods, offsets, worst case execution times and deadlines) are known at design time and do not change when the system is running, all the tasks can then be scheduled statically (Xu, Parnas 2000). An added advantage of statically scheduled systems is that most of the tasks can be run cooperatively, and the number and costs of pre-emptions can be minimized (Xu, Parnas 2000, Xu 2003).

1.2 Research Question

As mentioned in the previous section, most of the people involved in real-time system are of the opinion that the static table based cyclic schedules are very limiting and the only other option is to go for the fully pre-emptive dynamic scheduling architectures. Some of the most prominent books on real-time systems (e.g. “Real-Time Systems” by Jane Liu (2000) and “Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications” by Giorgio Buttazzo (2005)) introduce static cyclic executives, list their limitations and present priority based pre-emptive dynamic schedulers as the only way to overcome these limitations.

On the other hand, the level of predictability offered by static time-triggered schedulers makes them the preferred architecture for safety critical applications. This position is further improved by the time-triggered cooperative (TTC) and time-triggered hybrid

(TTH) schedulers, popularized by Pont (Pont 2001), answering some of the arguments raised against static schedulers.

Regardless, there remain some application areas where the time-triggered implementations are considered to be too rigid for the required performance and event-triggered or dynamic scheduling are considered to be the only practical options. These include applications where the software is supposed to keep in sync with and respond to the fast changing dynamics of a system. Examples of such systems include internal combustion engine controllers and synchronised three phase drives for motors.

This raises the question of the possiblity of finding some middle ground between the current static and dynamic scheduling paradimes that would retain most of the predictability of the static scheduling while allowing enough flexibility to permit its use in applications considered being outside of the domain of purely time-triggered systems.

1.3 Scope and Objectives of the Thesis

The overall aim of the research presented in this thesis is to consider the implications of applying a variation of a standard TT approach to a broader class of systems. In particular, the goal is to explore whether it may be advantageous to apply a more dynamic variation of the statically scheduled TT architecture – which will be referred to here as a "flexible TT architecture" – in environments which are considered by many to be a more natural match to an ET solution.

The research presented in this thesis focuses on predictable real time task scheduling on uni-processor architectures. The objectives of this research are as follows:

1. To identify the characteristics that make static time-triggered scheduling the preferred choice for high reliability and safety critical systems.

2. To identify the means of increasing the overall flexibility of a static scheduler while retaining the desired characteristics identified under the first objective.
3. To identify challenging application areas for purely time-triggered architectures and use these as case studies.
4. To test the performance of event-triggered, static time-triggered and flexible time-triggered architectures in a controlled environment by using the results of the case studies.

1.4 Layout of the Thesis

The layout of the thesis is as follows:

Review of relevant literature: Chapter 2 reviews the scheduling theory for real-time embedded systems and provides the justification as to why statically scheduled architectures are preferred for safety critical applications. Chapter 3 presents some of the static scheduling architectures and discusses their pros and cons. Chapter 4 introduces the two real world challenging applications that were studied in the course of this research.

The flexible cooperative architecture and the engine synchronization case study: Chapter 5 explores how the flexibility of the existing Time-Triggered Cooperative (TTC) scheduler can be increased by using some novel techniques like variable tick and cycle periods and multi segment cycles along with the existing idea of a system with multiple operating cycles. Chapter 6 present the Time-Triggered Multi Phase Cooperative (TTMPC) scheduler architecture that implements the desired features highlighted in chapter 5. A discussion of the effects of the enhancements on the predictability of the system is also presented. Chapter 7 presents the internal combustion engine synchronisation case study. It compares the effectiveness with

which the flexible TT, the static TT and the ET implementations can synchronise with an externally generated crank signal.

Limited flexible pre-emption and the brushless motor control case study: Chapter 8 show the limitation of a flexible scheduler based on the TTH and explores how limited flexible pre-emption can be added to a predominantly cooperative execution environment. It also discusses the factors that would affect the predictability of the flexible TT architecture with limited pre-emption. Chapter 9 presents a case study of the brushless motor control. This case study helps to highlight how the flexible TT architecture makes it possible to achieve higher performance by sacrificing some of the predictability of the static TT, while still retaining higher predictability than the ET implementations.

Conclusions and proposals for further research: Chapter 10 presents a discussion on the findings and contributions of the research reported in this thesis and introduces some potential applications that could benefit from the proposed architectures. It then presents a list of areas where future research might be undertaken in continuation of the work presented in this thesis.

2 A Review of the Relevant Scheduling Theory

This chapter reviews previous work on the scheduling of tasks in real time embedded systems. The work considered here forms a basis for discussions throughout the remainder of the thesis.

2.1 Tasks and Their Execution Environments

At the heart of the type of embedded system considered in this thesis, there will be a processor (or network of processors): the processor(s) will run well-defined blocks of software known as "tasks".

2.1.1 Classification of tasks

Tasks can be divided into two classes based on their temporal behaviour (Liu 2000, Buttazzo 2005a):

- **Periodic tasks:** These tasks are specified with a fixed period and initial delay (or "offset") and are run based on these temporal criteria (Liu 2000, Xu, Parnas 2000, Buttazzo 2005a). A large number of tasks in embedded systems are periodic in nature (Baker, Shaw 1988, Xu, Parnas 1993).
- **Aperiodic and Sporadic tasks:** These tasks do not have fixed periods and are run in response to external and internal events (Liu 2000, Xu, Parnas 2000, Buttazzo 2005a). A distinction is made among aperiodic tasks on the basis of the presence of limits on how frequently a task could be released: if a minimum interval between two consecutive requests is specified, it is called a sporadic task (Liu 2000, Buttazzo 2005a).

2.1.2 Converting sporadic and aperiodic tasks to periodic

As only periodic tasks can be scheduled in a purely time-triggered environment (Sha, Abdelzaher et al. 2004), sporadic and aperiodic tasks and not directly supported. However, in some cases, it is possible to use periodic tasks to handle sporadic events (Xu, Parnas 2000, Xu 2003).

For example, suppose a system is required to respond when a certain button is pressed. In event-triggered systems, this event might be handled as follows:

1. An interrupt could be used to launch an interrupt service routine (ISR). The functionality required to respond to the button presses would then be incorporated in the body of the ISR (Stewart 2001).
2. An interrupt could be used to add an appropriate sporadic / aperiodic task to the dynamic scheduler's task queue. The scheduler then controls when the task is actually run. (Sha, Abdelzaher et al. 2004)
3. An interrupt could be used to add an appropriate sporadic task to a sporadic task server's task queue. The relevant task will be run as soon as the system has finished running the periodic tasks. (Spuri, Buttazzo 1996)

A purely time-triggered system will have to run a periodic task to check the state of the button and generate an appropriate response. This period has to be short enough to ensure that the system can respond in a reasonable time (Xu, Parnas 2000, Xu 2003). The use of interrupts and events is a major cause of priority inversion and it is recommended to change them to periodic polling tasks to avoid these conditions (Stewart 2001).

2.1.3 Temporal criteria for tasks

The temporal criteria required by scheduling theory to find and verify a task schedule consists of the following (Jeffay, Stanat et al. 1991, Buttazzo 2005a):

1. Period: This is the period after which a periodic task is released for execution (Baker, Shaw 1988, Xu, Parnas 2000, Buttazzo 2005a). In the case of sporadic tasks, this value is used to indicate the minimum period between two consecutive events (Xu, Parnas 2000).

2. Deadline (relative): This is the time relative to the release of the task before which it has to finish executing to satisfy the real-time specification (Baker, Shaw 1988, Xu, Parnas 2000, Buttazzo 2005a).

3. Worst case execution time (WCET): This is the maximum amount of time that a task could need to finish executing in the absence of pre-emptions (Baker, Shaw 1988, Xu, Parnas 2000, Buttazzo 2005a).

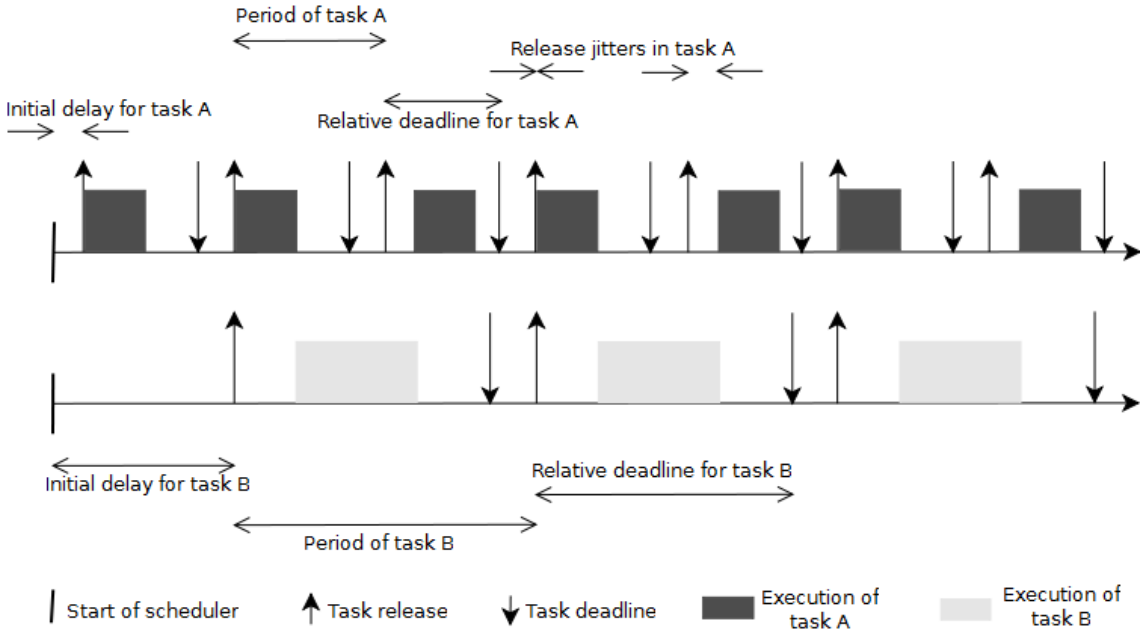


Figure 2-1: Temporal criteria for tasks

4. Initial delay (also referred to as "offset", "release time" or "phase"): The initial delay is the time (relative to the start of the scheduler) after which the first instance of the task is released for execution (Xu, Parnas 2000, Buttazzo 2005a).

Figure 2-1 provides a graphical representation of how some of these temporal criteria come into play in a hypothetical system with two tasks A and B. In this example, the initial delay and period of task A is less than those of task B. The deadlines for both the tasks are less than their respective periods. In this figure, execution time variations are not evident as all instances of a particular task are assumed to require the same time to execute as its WCET.

2.1.4 Jitter in real-time scheduling

Generally, jitter can be defined as deviation from the timing of an event under ideal conditions. Oxford English Dictionary defines jitter as:

> *"Slight irregular movement, variation, or unsteadiness, especially in an electrical signal or electronic device"* (Oxford English Dictionary 1989)

Task release jitter can be a major consideration for real-time systems especially in control system applications where its presence can result in degradation of performance (Proctor, Shackleford 2001, Buttazzo 2005b). The relative task release jitter is defined by Buttazzo as "*the maximum deviation of the start time of two consecutive instances"* (Buttazzo 2005a). It can be expressed in the form of the following equation:

$$\boldsymbol{RRj_i = max_k}\left|\left(\boldsymbol{s_{i,k} - r_{i,k}}\right) - \left(\boldsymbol{s_{i,k-1} - r_{i,k-1}}\right)\right| \qquad \textbf{(2-1)}$$

Where RRj_i is the relative release jitter for task i over multiple executions, $s_{i,k}$ and $r_{i,k}$ are the start time and release time for task i at the k cycle while $s_{i,k-1}$ and $r_{i,k-1}$ are the corresponding start and release times for the previous cycle.

In Figure 2-1, relative release jitter is evident in the executions of task A as its start of execution is delayed in alternate ticks starting from its third iteration due to the execution of task B. Assuming that the execution time for task A remains constant, the relative release jitter for task B will be zero as all of its executions will be delayed by the same amount of time.

2.2 Worst Case Execution Times and Scheduling

Both static and dynamic scheduling techniques require advanced knowledge of the worst case execution times (WCET, i.e. the maximum amount of time that is required by a task to complete) for all tasks in the system (Wilhelm, Engblom et al. 2008). Without access to this information, the static task schedules cannot be assembled and verified (Burns 1995, Gendy, Pont 2008), while schedulability analysis for dynamic scheduling algorithms cannot be calculated (Liu, Layland 1973, Sha, Abdelzaher et al. 2004, Buttazzo 2005a).

2.2.1 Factors affecting Worst Case Execution Time

Depending on the structure of a task and the values of the control variables, different execution paths are possible. Finding the longest path is challenging and might not be possible for complex tasks (Deverge, Puaut 2005, Wilhelm, Engblom et al. 2008).

WCET analysis is further complicated by the effects of speed enhancement techniques that are in use in modern processors and microcontrollers. These include instruction memory caches, virtual memory, pipelines and branch predictions (Deverge, Puaut 2005, Wilhelm, Engblom et al. 2008, Mezzetti, Holsti et al. 2008).

2.2.2 Worst Case Execution Time analysis techniques

The main techniques that are used in WCET analysis are as follows:

1. **Dynamic / measurement based analysis:** This technique relies on timing measurements made on the actual hardware or with accurate simulators. Test data is generated in an attempt to make the execution take the longest path through the task; however, it usually cannot be guaranteed that the observed maximum execution time will not be exceeded at run time. (Wilhelm, Engblom et al. 2008)
2. **Static analysis of the code:** In this technique, the code is analyzed to determine the longest path and the conditions under which this path is chosen. With static analysis, it is possible to determine an upper timing bound which cannot be exceeded at run time. However, in order to cover the variations introduced due to the processor architecture (e.g. branch prediction, caches, etc.), this value can be pessimistic. (Engblom, Puschner, Burns 2002b, Ermedahl et al. 2003, Deverge, Puaut 2005, Mezzetti, Holsti et al. 2008, Wilhelm, Engblom et al. 2008)
3. **Mixed analysis techniques:** Some of the analysis techniques break up the task into smaller blocks of code with fixed execution times and find the overall execution time based on the timing values for the blocks and knowledge of the longest path through the task (Engblom, Ermedahl et al. 2003, Deverge, Puaut 2005, Wilhelm, Engblom et al. 2008).

It is not uncommon to add a safety margin for real-time systems to the WCET values based on measurement techniques (Vallerio, Jha 2003, Gendy, Pont 2008, Gendy 2009).

2.2.3 Minimising execution time variations

An alternate approach to simplifying the WCET analysis is to try to minimize or eliminate the variations:

2.2.3.1 Single path programming

Single path programming is a paradigm that limits the number of execution paths through a task to one (Puschner, Burns 2002a, Puschner, Burns 2002b, Puschner 2003). With the presence of low level support for conditional instruction execution, it is possible to write code that takes the same path regardless of the data that cause conditional execution of some instructions. The conditional instructions are supported by most of the modern architectures like Freescale M-core, Alpha, Pentium P6, ARM7TDMI and Cortex-M3 (Puschner, Burns 2002a, Motorola 2001, ARM 2004, ARM 2010).

Single path programming relies on both the coding practices and tool / architecture support for generation of code with a single flow path. When used properly, it can result in the elimination of execution time variations.

2.2.3.2 Code balancing 1 technique

The code balancing 1 (CB1) technique, proposed by Gendy and Pont, uses a hardware timer and sandwitch delays in an attempt to minimize the execution time variations (Gendy, Pont 2007, Gendy 2009). To stabilize the variations in execution times of a loop structure, a counter is used to measures the number of iterations, while a timer measures the time required for these measurements. After the loop, the system is put into low power mode for the estimated remainder of the time that the loop would have taken had it run for the maximum number of iterations.

The CB1 is a generic technique that can be adapted for use on all architectures. It reduces the amount of jitter in the WCET but might not eliminate it (Gendy, Pont 2007).

2.3 Overview of Scheduling Techniques

Over the years various techniques have been developed and used to schedule tasks (Liu, Layland 1973, Baker, Shaw 1988, Xu, Parnas 1990, Stewart, Khosla 1991, Locke 1992, Kalinsky 2001, Pont 2001, Sha, Abdelzaher et al. 2004). As previously mentioned in section 1.1.1, these techniques can be categorized according to the means by which they launch the tasks into the following groups:

1. Static (offline) Schedulers: The order of task execution is determined at design time. These systems usually rely on a timer interrupt to keep track of the passing of time and dispatch tasks according to the pre-programmed scheduling sequence (Locke 1992, Fidge 2002). Examples of such systems include clock driven cyclic executive schedulers like time line schedulers, time-triggered cooperative (TTC) and time-triggered hybrid (TTH) (Baker, Shaw 1988, Pont 2001, Scheler, Schroder-Preikschat 2006, Pont, Kurian et al. 2007, Wang, Pont 2008).

2. Dynamic (online) Schedulers: The order of task execution is determined at run time based on an online scheduling algorithm or on external and internal events. The tasks are run on the basis of their priorities (Liu 2000). They can be further sub divided according to their priority assignment technique:

    i. Fixed task priority algorithms: The tasks are assigned priorities at design time, and these do not change at run time (Liu 2000, Sha, Abdelzaher et al. 2004). At run time, the resources are allocated to the task with the highest priority. Two commonly used algorithms for assigning priorities are Rate

Monotonic (RM – the priorities are assigned according to the tasks' periods) (Liu, Layland 1973) and Deadline Monotonic (DM – the priorities are assigned according to the deadlines) (Liu 2000).

ii. Dynamic task priority algorithms: The priorities are calculated at run time based on some criteria specified by the algorithm (Liu 2000, Sha, Abdelzaher et al. 2004). Examples of these algorithms include Earliest Deadline First (EDF – the task whose relative deadline is closest is assigned the highest priority) (Liu, Layland 1973) and Minimum / Lowest Laxity First (MLF / LLF – the task that has the least amount of slack - i.e. the difference between time till its deadline and its remaining execution time - gets the highest priority) (Liu 2000).

iii. Mixed task priority algorithms: The cumulative task priorities are composed of a statically assigned part and a dynamically calculated part. One example of a mixed priority algorithm is Maximum Urgency First (MUF – static priorities are assigned to task groups. When multiple tasks are waiting, the task from the highest group gets the priority. Dynamic priorities are used to resolve conflicts if two or more waiting tasks are from the same group) (Stewart, Khosla 1991).

It should be noted that the division between static and dynamic schedulers outlined above is typical but the precise split may depend on the chosen implementation (Gendy 2009): for example, a rate-monotonic or EDF algorithm can be applied at design time in a statically-scheduled system (Xu, Parnas 1990). Similarly, a statically scheduled system could be modified to run non periodic or sporadic tasks in the slack time between the scheduled tasks (Xu, Parnas 1993, Liu 2000).

No matter what combination of algorithm and implementation we choose, none of these techniques offers a panacea to the problems involved in systems design (Scheler, Schroder-Preikschat 2006): they all have their strong and weak points, and the type of scheduler used in an application is a design decision (Scheler, Schroder-Preikschat 2006). It is usually assumed that, for systems with large task sets, dynamic schedulers offer higher CPU loading and ease in developing application software (Kopetz 1991, Fidge 2002, Scheler, Schroder-Preikschat 2006). On the other hand, a statically-scheduled architecture runs tasks in a predetermined order and offers a greater possibility of determining all possible paths in the software and, therefore, of obtaining better reliability (Kopetz 1991, Fidge 2002, Scheler, Schroder-Preikschat 2006).

2.4 Comparison of Scheduling Architectures

In order to understand the strengths and weaknesses of various scheduling architectures, an impartial comparison has to be made between them. Most of the research available on the subject focuses on one particular architecture or on a comparison between two such architectures. In this section, an impartial comparison is presented between the some of the commonly used real-time architecture. This comparison focuses on the following five architectures:

1. Static scheduler (Static): A clock driven table based cyclic executive implementation is assumed.
2. Rate Monotonic (RM): A fully pre-emptive rate monotonic fixed priority implementation is assumed.
3. Earliest Deadline First (EDF): A fully pre-emptive earliest deadline first dynamic priority implementation is assumed.

4. Non pre-emptive Earliest Deadline First (npEDF): A fully cooperative earliest deadline first dynamic priority implementation is assumed.
5. Maximum Urgency First (MUF): A fully pre-emptive maximum urgency first mixed priority implementation is assumed.

2.4.1 Comparison Criteria

The criteria used for a comparison of the above mentioned five architectures is as follows:

2.4.1.1 Scheduler overheads

Scheduler overheads are the load incurred on the processor while running the scheduling algorithm. These scheduler overheads are usually not taken into account when schedulablity analysis is made for a task set (Jeffay, Stanat et al. 1991).

Static scheduler based designs tend to have lower scheduler overheads as compared to pre-emptive dynamic schedulers (Locke 1992, Fidge 2002, Xu 2007). The scheduling decisions are made at design time (Liu 2000, Xu, Parnas 2000) and are stored in the system in the form of a table. The scheduler is invoked at fixed points in time and uses the table to find out which tasks need to be run.

In the case of dynamic schedulers, all tasks waiting to be run are sorted according to their priorities. Each time a new task is added into the system, the queue has to be reordered (Burns 1995). In addition to this, for dynamic priority scheduling algorithms, priorities also have to be updated at run time (Liu 2000, Fidge 2002). Based on this, it may be assumed that the dynamic priority algorithms have higher overheads as compared to the fixed priority algorithms. However, in the case of RM and EDF algorithms, Buttazzo argues that if the number of pre-emptions occurring in the system

are taken into consideration, EDF tends to have fewer overheads than RM (Buttazzo 2005b).

Cooperative systems (both static and dynamic) incur lower over heads as compared to pre-emptive systems because there is no need for context switches or message queues to transfer data between different tasks (Jeffay, Stanat et al. 1991, Short, Pont et al. 2008, Short 2010).

Based on the foregoing, it may be seen that a static cooperative will have the lowest scheduling overheads, followed by static pre-emptive. The RM should be moderate to high, while the EDF should be high to moderate when Buttazzo's observations taken into account. The npEDF should have low to moderate overheads. Finally, the overheads for the MUF should be similar to or slightly higher than those for EDF because of the need to calculate the laxity for the dynamic part of the algorithm.

2.4.1.2 Scheduler memory requirements

All scheduler architectures require some memory to keep the task executions on track. As previously noted, memory is also required for context switches and message passing mechanisms in a pre-emptive environment.

The table based static scheduler requires considerably more memory than other approaches, as they have to store the scheduling table prepared at design time (Liu 2000). The size of this table is dependent on the least common multiple (LCM) of all the task periods, and can be quite large even with a small number of tasks. For example, for a system with two tasks A and B with periods of 3ms and 11ms, the table will need to be large enough to store all the task activations in a 33ms interval.

For a dynamic scheduler, memory will be required for a queue for storing waiting tasks, and for a table with tasks and their timing information including periods, offsets and/or deadlines (Fidge 2002). The size of this task table is proportional to the number of tasks and does not depend on their respective periods.

In general, pre-emptive schedulers need relatively more memory to maintain unified or separate stacks for task execution as compared to cooperative schedulers (Buttazzo, Gai 2006, Short 2010).

From the above it can be inferred that the static schedulers will have the highest memory requirements as compared to dynamic pre-emptive schedulers like RM, EDF and MUF and finally, npEDF should have the lowest requirements because of the absence of pre-emption.

2.4.1.3 Theoretical achievable CPU loading

It is usually considered that architectures which allow higher CPU loading, while ensuring that all the task timing constraints are met, are better as more processing can be done on the same hardware platform.

The debate of which scheduling architecture can offer higher processor loading is ongoing and unresolved. Supporters of dynamic scheduling claim that these systems are able to run more tasks because of higher processor loading (Liu, Layland 1973, Locke 1992, Sha, Abdelzaher et al. 2004). It was shown that the worst case processor loading for a fixed priority dynamic system is approximately 69% (Liu, Layland 1973). This figure, however, is pessimistic and for average real time systems, this value is usually closer to 88% (Lehoczky, Sha et al. 1989). Audsley et al. claim that their worst case response time analysis (for tasks with offset constraints) helps to improve the

schedulablity of task sets with fixed priority scheduling (Audsley, Tindell et al. 1993). However, the actual achievable processor loading is dependent on the specific task set being scheduled.

In the case of pre-empting system architectures with dynamic and mixed priority scheduling, processor loading of 100% is possible (Liu, Layland 1973, Stewart, Khosla 1991, Liu 2000). However, if pre-emption is not allowed, these algorithms are no longer optimal and 100% utilisation cannot be guaranteed (Liu 2000). While a CPU utilization figure cannot be generalized for the cooperative EDF, it is possible to calculate if a particular task set is schedulable (Short 2010).

The work done by Xu has gone a long way to show that static scheduling is still a viable and attractive option for real-time systems (Xu, Parnas 2000, Xu 2003, Wang, Pont 2008). His scheduling algorithm uses the schedule produced by EDF as a starting point and then tries to improve on it (Xu, Parnas 1990). Based on this, and taking pre-emption overheads into consideration, it can be argued that, depending on the task set and the proper scheduling algorithm, statically scheduled systems can have processor utilisation similar to, or better than, dynamic systems. On the other hand, the extra effort required to break up long tasks hinders actual utilisation achieved by static systems.

Based on the above arguments, the achievable processor loading for static schedulers should be moderate but high levels can also be achieved with task segmentation and Xu's scheduling technique. The performance for RM is moderate while EDF and MUF allow for high processor loading. The npEDF's performance should be moderate to high depending on the particular task set. Finally the performance of the MUF architecture should be very high.

2.4.1.4 Ease of system design and maintenance

The choice of system architecture has a large impact on the time and effort that is required to develop and maintain the software.

Statically scheduled systems are usually difficult to construct (Locke 1992, Kalinsky 2001). The most obvious problem is that of the generation of the schedule table. The work done by Burns provides an overview of the various techniques available for schedule generation and compares two of the techniques in a detailed case study (Burns 1995). Also, the addition or removal of a single task, or change in timing parameters of an existing task, can require a recompilation of the schedule table (Locke 1992, Burns 1995, Gendy, Pont 2008). In the statically scheduled systems, the time periods of all tasks must be multiples of the base tick period and should be harmonically related. This could result in the need to change the periods of some tasks and running them at a higher rate. (Locke 1992)

Alternately, systems developed using dynamic scheduling algorithms offer apparent ease in software design (Locke 1992, Fidge 2002). Various techniques have been developed to check the feasibility of a given task set (Liu, Layland 1973, Audsley, Tindell et al. 1993). A change in the task set only requires that the schedulability is recalculated. Also, tasks are not required to have periods that are harmonically related (Locke 1992).

The cooperative dynamic scheduler complicates the design process as long tasks might need to be split in order to make a task set schedulable (Burns 1995). Such code is difficult to write and maintain.

In light of the above arguments, the dynamic pre-empting schedulers including RM, EDF and MUF are easy to use. The npEDF should require moderate effort, while static schedulers require a lot of effort for design and maintanence.

2.4.1.5 Verification procedure

For hard real-time applications, it is essential to test and verify that the system works reliably. Dijkstra's view on system testing highlights the main limitation of trying to prove the correctness of a system through testing:

> *"Program testing can be used to show the presence of bugs, but never to show their absence!"* – (Dijkstra 1970)

A prime example of this limitation of testing is the delay in the first space shuttle launch caused by a transient overload (with a 1 in 67 probability) putting the computers out of synchronization during initialization (John 1988).

Run time behaviour of dynamically scheduled systems is more difficult to analyze and predict as compared to pre-run-time scheduled systems (Xu, Parnas 2000, Xu 2003). This is mainly because of the need to use complex run time mechanisms to achieve process synchronisation and access to shared resources. Coincidently, the cooperative dynamically scheduled systems are easier to test and analyse because locking and synchronization mechanisms are not needed for shared resources (Jeffay, Stanat et al. 1991).

Existing safety certification guidelines (e.g. avionics standard DO-178B) require exhaustive testing of all possible control-flow paths through a program. It is not clear how this can be achieved in a dynamic scheduler that relies to some extent on external events to determine the order of execution of tasks (Fidge 2002).

From the above, it is clear that the verification for the static time-triggered is the easiest because of the set order of task execution. This is followed by the npEDF where the lack of pre-emption eases the verification process. Finally, the pre-emptive dynamic are the hardest to verify because of the potentially large number of execution paths through the software.

2.4.1.6 Code complexity due to shared resources

This metric is an estimation of the additional code complexity due to the presence of shared resources in the system.

One of the basic assumptions behind dynamic pre-emptive scheduling is that the tasks are independent of each other (Liu, Layland 1973). Unfortunately, this assumption does not hold in a lot of real-time systems, where different tasks have to share resources and information with each other and can lead to priority inversion (Xu, Parnas 2000). Priority inversion occurs when the execution of a higher priority task is delayed by the execution of a lower priority task (Babaoglu, Marzullo et al. 1990, Liu 2000, Buttazzo 2005a). Over the years, various locking protocols and techniques have been developed to ensure smooth operations (Sha, Abdelzaher et al. 2004, Scheler, Schroder-Preikschat 2006). These include defining non pre-emptive critical sections, priority ceiling and priority inheritance. Such features can add to system (and design) complexity (Xu, Parnas 2000, Yodaiken 2002).

In the case of statically scheduled systems, all the resource constraints can be handled at design time, thereby, reducing the software's complexity and overheads (Xu, Parnas 1990, Xu, Parnas 2000).

Based on the above, the code complexity added to the user code is the lowest for the statically scheduled, followed by the npEDF and should be the highest for dynamic pre-emptive systems.

2.4.1.7 Temporal flexibility

Temporal flexibility is a measure of the possibility to change task timing parameters at run time. This can be crucial for some applications, making certain scheduling architectures inherently inefficient or unsuitable for them (two such applications are presented in chapter 4).

Existing static scheduling architectures have no temporal flexibility. However, it has been suggested by some that a system could have multiple schedule tables in order to implement different operating modes and conditions (Baker, Shaw 1988, Kopetz, Nossal et al. 1998, Xu, Parnas 2000). While these systems will have limited flexibility, it will not be possible to implement a system in which timing can be varied smoothly within a certain range.

In dynamic priority system analysis, a frequent assumption is that the deadline of any task is less than or equal to its period (e.g. Liu and Layland make this assumption in their seminal paper on rate monotonic and earliest deadline first dynamic schedulers (Liu, Layland 1973)). If such a system is modified to allow task periods to be changed at run time, it can be argued that most analysis techniques should hold if it can be ensured that at any point in time, the static deadlines are less than or equal to the variable periods.

Fixed priority systems should fall somewhere in the middle of the spectrum. In addition to the conditions for dynamic priority systems, designs with variable periods will also

need to ensure that varying the periods of some of the tasks does not result in a priority inversion situation in rate monotonic arrangements (i.e. new period of a lower priority task being shorter than the period of a higher priority task). A deadline monotonic system with fixed deadlines should not be adversely affected by this.

Based on the above discussion, it can be summarised that dynamic and mixed priority pre-empting systems should have the highest level of flexibility. Fixed priority RM and non-pre-emptive dynamic architectures like npEDF should fall in the middle of the spectrum, while statically scheduled permit almost no flexibility.

2.4.1.8 Temporal stability

In real-time systems, the usefulness of some tasks is reduced by timing variation between successive calls (task launch jitter). This is especially true in control systems, where jitter in sampling a data source can reduce the value of collected data or the effectiveness of the control algorithm (Locke 1992, Proctor, Shackleford 2001).

Statically scheduled architectures allow a great level of control on the execution order of tasks. This can be used to minimize task launch jitter for specific tasks (Locke 1992).

In fixed priority architectures, while the high priority tasks tend to have low jitter, this does not hold true for all the tasks in task set. In some cases, when jitter for all tasks is taken into account, EDF was shown to be better than rate monotonic by Buttazzo (Buttazzo 2005b).

Based on the above, it can be inferred that static schedulers offer the highest temporal stability while dynamic pre-empting schedulers offer moderate stability and are adversely affected by shared resources. Finally, the npEDF architecture's performance

is not substantiated by available research but should offer the lowest level of temporal stability because of the cooperative nature of task executions and lack of precise control over the sequence in which the tasks are executed.

2.4.1.9 System predictability

System predictability is a measure of the ease and accuracy with which the behaviour and state of the system can be determined at an arbitrary point in the future.

Generally, statically scheduled systems are inherently predictable (Locke 1992, Xu, Parnas 2000, Fidge 2002). The starting times of all tasks are known at design time and do not vary while the system is operating. Certification authorities tend to support this form of scheduler architecture (Fidge 2002).

In the case of dynamically scheduled systems, because of possible variations in the task execution periods, the sequence of task executions cannot be determined in advance (Kopetz 1991, Fidge 2002). They also do not support replica determinism (a setup where, in the absence of faults and provision of the same initial state and inputs, similar systems generate the same set of outputs at the correct times) by design and require implementation of special techniques to achieve this (Scheler, Schroder-Preikschat 2006).

In the case of cooperative dynamic schedulers, the determinability problem is simplified due to the absence of pre-emptions in the system (Short 2010). For particular task sets, it might be possible to ensure the execution sequence of tasks provided that there is little or no variation in the task execution times.

It can be seen from the above that the statically scheduled provide the highest level of system predictability, followed by npEDF, and finally, the dynamic pre-emptive schedulers offer the lowest level of predictability.

2.4.1.10 System robustness under overloads

As mentioned previously, scheduling theory relies on WCET estimates and measurements for analysis and verification of task sets. Errors in these could invalidate all reaults of analyse based on them.

Statically scheduled systems tend to be fragile as a single task overrunning its scheduled period can cause catastrophic failure of the system (Locke 1992, Burns 1995, Fidge 2002).

Dynamically scheduled systems are better than their statically scheduled counterparts under transient overload conditions (Locke 1992, Fidge 2002). It has been shown by Buttazzo that in a RM setup, all tasks with a priority higher than the offending task (the one causing the overload) will continue to run normally; but in an EDF setup, any task can miss its deadline (Buttazzo 2005b). The performance of RM can lead to a false sense of security as the over running task is not known at design time. Also, as tasks usually interact with other tasks in the system, it is hard to predict the repercussions of the failure of an arbitrary task on the overall performance.

Maximum urgency first (MUF) allows the task set to be sorted into categories. In the case of a failure, the higher priority groups can continue to run while tasks from lower priority groups suffer (Stewart, Khosla 1991). While there are no guaranties that a task in the critical section will not fail, it might be easier to ensure that the tasks in the critical subset of software have built in safeguards to prevent such an occurance.

No analysis is available for npEDF under transient overloads; however, it should not be expected to perform better than EDF.

To summarise, the static scheduler have negligible system robustness under overload conditions. The robustness of RM, EDF and npEDF are very low under such conditions. Finally, MUF could guarantee a Low level of robustness.

2.4.2 Summary of the comparison of various scheduler architectures

The findings of the comparison in this section have been summarised in Table 2-1. For safety critical applications, high level of predictability and ability to verify the correct operation is essential. With reference to the above table, it can be seen that the static scheduling satisfies both the essential requirements. Unfortunatly, this form of task scheduling is limited by large memory requirements, lack of flexibility, and difficulty in system design.

Table 2-1: Qualitative comparison of different scheduler architectures.

	Static	***RM***	***EDF***	***npEDF***	***MUF***
Scheduler overheads	Low	Moderate-High	High-Moderate	Moderate	High
Scheduler memory requirements	High	Moderate	Moderate	Low	Moderate
Theoretical achievable CPU loading	Moderate to high	Moderate	High	Moderate to high	Very high
Ease of system design and maintenance	Hard	Easy	Easy	Moderate	Easy
Verification procedure	Easy	Hard	Hard	Moderate	Hard
Code complexity due to shared resources	Low	High	High	Low	High
Temporal flexibility	None	Low	High	Moderate	High
Temporal stability	High	Moderate	Moderate	Low	Moderate
System predictablity	High	Low	Low	Moderate	Low
System robustness under overloads	None	Very low	Very low	Very low	Low

2.5 Notable Incidents with Real-Time Systems

This section provides an insight into some incidents related to real-time systems in high reliability and safety critical applications and highlights how those issues could have been avoided with proper design, testing and verification processes.

Apollo 11 lunar landing

During the final stages of the Apollo 11 landing, frequent 1202 (computer overload error) and 1201 (effectively an out of memory error) alarms were received from the guidance computer (Martin 1994, Adler 1998, Eyles 2004, Jones 2011). The last landing simulation before the mission (but with another crew) was aborted when a similar alarm was received (Jones 2011).

Initially, the cause of the alarms was attributed to an error in the descent checklist. This was not detected because the hardware simulator used to test the system and train the crew did not have the relevant switch (rendezvous radar mode) hooked to the guidance computer. It was thought the need to process the data from this radar in addition to all the other operations involved in the landing caused an overload in the guidance computer. (Martin 1994, Adler 1998, Eyles 2004)

It was much later discovered that the overloads were caused when two of the power sources for the radar and guidance computer were synchronised out of phase. This would result in the generation of an event in every cycle of the power input as it appeared that the radar antenna was oscillating due to the phase difference. (Eyles 2004)

In either case, this is a strong example for the need for clearly defined (and verified) task sets for different operating modes and the perils of event-triggered systems when there is an underestimation of the rate at which an event might occur.

Therac-25

The Therac-25 was a radiation therapy machine produced by Atomic Energy of Canada Limited. It was based on the earlier Therac-6 and Therac-20 machines that had been operating safely for many years. Between 1985 and 1987, there were six accidents involving the Therac-25s which resulted in massive overdoses in the administered treatment, some of them resulting in the death of the patient being treated (Joinathan 1994, Leveson 1995).

The software for the Therac-25 was based on an earlier model but unlike the older model, there were no hardware protection mechanisms and software was the only line of defence (Joinathan 1994, Leveson 1995).

Some of the failures were caused by a race condition when an operating mode was set and then changed within a very short period resulting in incorrect configuration of the machine, while the others were caused by a logical bug that caused a state variable to overflow and reset (Leveson 1995).

The race condition was occurring because of the inherent event-triggered nature of data and operating mode specification. An implementation based on periodic polling and specific operating modes with specific task sets might have avoided this situation but at the cost of decreased flexibility and increased CPU utilisation for the same capability. Alternatively, this problem might have been avoided by using resource locking techniques to prevent corruption of the settings in a dynamic execution environment. Regardless of the implementation techniques, a detailed analysis should have been performed that might have highlighted the issue with shared resources.

The problem with the overflowing of a state variable only helps to emphasise the need for software verification and testing in safety critical applications. Because of the event-triggered nature of the implemented software, it would have been unlikely to be able to test all possible execution paths through the software.

2.6 Conclusions

This chapter presentes a literature review of the uniprocessor real-time scheduling theory. A detailed impartial comparison of some of the commonly reffered architectures was carried out. Some of the comparison criteria were not covered in sufficient detail in the literature and requied some assumptions for a thorough comparion. The results of this comparison were then summarised in the form of a table that highlighted the pros and cons of the architectures being compared. The literature review presented in this chapter helped to justify the selection of the static time-triggered architecture as the foundation for the predictable flexible TT architectures.

3 Static Scheduling Architectures

This chapter takes a detailed look at the characteristics of existing static scheduling architectures. The work reviewed here forms the basis on which the flexible architectures are based.

3.1 Commonly Used Static Scheduler Architectures

Most of the references to static schedulers imply or specify a table driven cyclic executive scheduler (referred to as timeline schedulers in this thesis) (Baker, Shaw 1988, Xu, Parnas 1993, Xu 2003). In this architecture, a table is used to store the points of time at which various tasks are due to run. A timer interrupt can then be used to launch the tasks at their respective dispatch times. This approach provides a lot of flexibility in controlling the task timings. Also, in such an architecture, jitter in the task launch times can be eliminated as every task is dispatched in its own time slot. However, the size of the table required to store all the scheduling information is proportional to the least common multiple of all the task periods (the major cycle) and can be very large in some cases (Liu 2000, Xu 2003).

An alternate to table based timeline scheduling architecture was presented by Pont in the form of Time-Triggered Cooperative (TTC) and Time-Triggered Hybrid (TTH) schedulers (also referred to as next run time schedulers) (Pont 2001). These schedulers use data structures (referred to as task arrays) to keep track of the time units or ticks remaining till the next execution of each task. The size of the task array is proportional to the number of periodic tasks set to run in a system and not on the relationship of the task periods of all the tasks. In addition to this, the TTC and TTH schedulers are also able to survive transient overloads but at the cost of increased jitter for all affected tasks.

3.2 A Detailed Review of Static Schedulers

This section reviews some of the static time-triggered scheduler implementations and their pros and cons.

3.2.1 Timeline scheduler

While most of the research on static cyclic executives refers to time line schedulers, it is difficult to find references to implementations in practical work. The version by Wang is a very flexible pre-emptive implementation that uses a single timer interrupt to control the execution of all tasks (Wang, Pont 2008).

Key points of Wang's architecture are:

- In the system, only the scheduler's timer interrupt is enabled. All other events have to be polled for in a task.
- The tasks are run by the scheduler's dispatcher in a cooperative or pre-emptive manner depending on the schedule.
- Special functions are used to save and restore the context of the pre-empted tasks.
- The scheduler uses a timeline array to store the points in time at which the tasks are to be run.

As all the task start times are fixed in a timeline scheduler, it is very difficult to modify the system to be able to vary task periods at run time.

3.2.2 Time-triggered cooperative scheduler

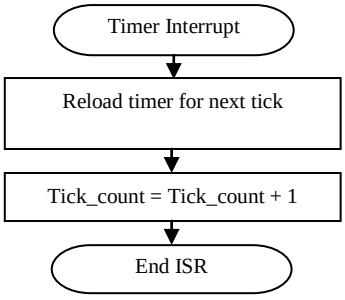
The time-triggered cooperative (TTC) scheduler given by Pont in 2001 is a statically-scheduled cooperative scheduler that uses a single timer interrupt to control the execution of all tasks. Improved versions of this scheduler with reduced jitter have been proposed (Maaita, Pont 2005, Phatrapornnant, Pont 2006).

The key points of this TTC architecture are:

- In the system, only the scheduler's timer interrupt is enabled. All other events have to be polled for in cooperative tasks.
- The tasks are run by the scheduler's dispatcher in a cooperative manner.
- The scheduler uses a task array to keep track of the scheduled tasks, their periods, and the time till their next call.

The key parts of the scheduler are:

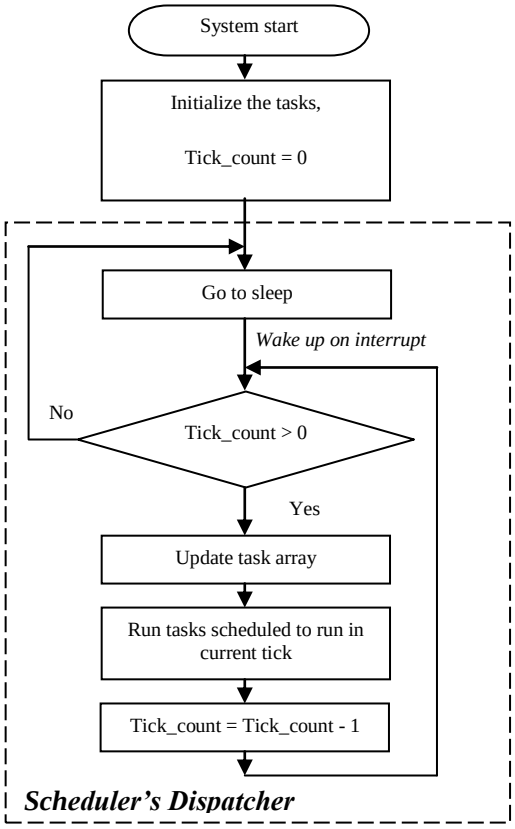
1. TTC scheduler's timer ISR: The timer's ISR indicates the occurrence of a "Tick" to the scheduler's dispatcher (see Flowchart 3-1).



Flowchart 3-1: Flowchart of the TTC scheduler's Timer ISR.

2. TTC scheduler's dispatcher: The scheduler's dispatcher is responsible for updating the task array after every tick, running tasks that are due to run in the current tick, and updating the time till their next call. Once all the tasks have

been completed, the CPU is put into sleep till the next tick. Refer to Flowchart 3-2 for the general structure of the system using the TTC scheduler.



Flowchart 3-2: Flowchart of the TTC system's main structure.

The TTC scheduler suffers from the following short coming:

- Like all cyclic executive algorithms, the response time is not good for a system with long cooperative tasks (Allworth 1981, Locke 1992, Fidge 2002).
- The time periods of all tasks must be multiples of the base tick period.
- Task overruns (tasks which exceed their predicted "worst case" execution time) can have a significant impact on system performance. (Although "task guardians" can be employed, these add significant overheads (Hughes, Pont 2004, Hughes, Pont 2008)).

3.2.3 Time-triggered hybrid scheduler

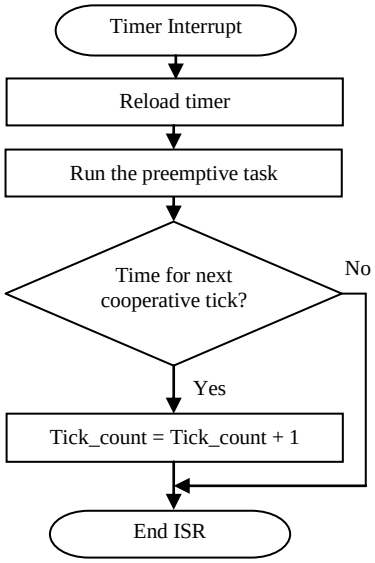
Like the time-triggered cooperative, the time-triggered hybrid scheduler is also taken from Pont's book Patterns for Time-Triggered Embedded Systems (Pont 2001). The TTH scheduler was intended to overcome the problem with scheduling a high priority task with a short time period and one or more cooperative tasks with durations more than the period of the high priority task by allowing limited pre-emption in the system.

The key points of this TTH architecture are:

- In the system, only the scheduler's timer interrupt is enabled. All other events have to be polled for in cooperative or pre-empting tasks.
- The pre-empting task is launched from the scheduler's timer ISR and can pre-empt any currently running cooperative task. The context switch is handled by the ISR and no special coding is needed.
- All pre-empting tasks run cooperatively (i.e. one pre-empting task cannot pre-empt another pre-empting task).
- The cooperative tasks are run by the scheduler's dispatcher in a cooperative manner.
- The scheduler uses a task array to keep track of the scheduled cooperative tasks, their periods, and the time till their next call. The data of the pre-empting task is stored in a separate data structure.

The key parts of the scheduler are:

1. TTH scheduler's timer ISR: The timer's ISR is used to run the pre-empting task, and also indicates the occurrence of a "Tick" to the scheduler's dispatcher after the preset cooperative tick interval has passed (Flowchart 3-3).



Flowchart 3-3: Flowchart of the TTH scheduler's timer ISR.

2. TTH scheduler's dispatcher: The TTH's dispatcher is virtually identical to the TTC's dispatcher. Refer to section 3.2.2 and Flowchart 3-2 for details.

Although the TTH overcomes the problem of scheduling a task with a period shorter than the execution time of some other cooperative tasks that a TTC cannot manage, other problems remain:

- The time periods of all cooperative tasks must remain multiples of the pre-empting task's time period.
- Implementing task guardians (to deal with overruns in co-operative or pre-empting tasks) adds greatly to the scheduler complexity (Hughes, Pont 2008).

3.3 Conclusion

In this chapter the implementation details of some of the available static time-triggered architectures are reviewed. The time-triggered cooperative (TTC) and time-triggered hybrid (TTH) architectures reviewed in this chapter reduce the memory requirements of the table driven schedulers to a level that is comparable to dynamic schedulers. Also, the TTH enhances the capabilities of the TTC architecture by allowing some of the frequent tasks to pre-empt longer tasks so as to meet the requirements that could not be satisfied with a purely cooperative scheduler. Despite these advantages, such architectures are still quite rigid as they do not allow any temporal flexibility. These two architectures form the basis for the flexible time-triggered architectures presented in this thesis.

4 Challenging Real World Applications

This chapter provides background information on two applications that are challenging for static time-triggered architectures. These studies will form the focus of the research in the remainder of this thesis.

4.1 Introduction

In order to test the performance of proposed flexible time-triggered schedulers, challenging case studies were required. It was desired that the applications chosen for the case studies should have wide spread usage and should also provide significant challenges for classical static scheduler based implementations. The two applications that were selected were:

1. Internal combustion engine control: Internal combustion engines are primovers of the modern lifestyle and are used to provide power for applications ranging from small hand held power tools to ocean going ships and power plants.
2. Brushless DC motor control: Brushless motors offer improved speed, efficiency and reliability as compared to brushed DC motors. Their application areas range from servo control in robotics and automation to automotive industry.

The rest of the chapter provides more details about these applications and the challenges that they present for static time-triggered implementations.

4.2 Internal Combustion Engine Control

The reciprocating internal combustion engines have been around for over a century. Nicolaus A. Otto is credited for building the first four stroke internal combustion engine in 1876 (Pulkrabek 1997, Bellis). In the 1880s, internal combustion engines began to appear in automobiles. Since then, these engines have been used to provide

power for applications ranging from lawn mowers and hand tools to ships and electric power plants. In order to understand the challenges presented by these engines, we will be concentrating on four stroke spark ignition petrol engines.

4.2.1 Inner workings of a spark ignited internal combustion engines

The basic section of the engine is a cylinder. The combustion chamber with its openings for mechanical intake and exhaust valves is on one end of the cylinder. The intake valve(s) can be opened to allow induction of the fuel and air mixture into the combustion chamber. A spark plug is used to ignite the compressed fuel air mixture. The exhaust valve(s) can be opened to allow the burnt remains of the chemical combustion to leave the combustion chamber. A piston, connected to a crankshaft, is able to move up and down the cylinder with minimum leakage between the sliding surfaces. In multi cylinder engines, cylinders are arranged in various orientations (I, V, W and radial to name a few), with all the pistons connected to a single crank shaft. (Pulkrabek 1997)

As the name suggests, each cycle of a four stroke engines is composed of four strokes:

1. Induction stroke: Fuel and air mixture enters the combustion chamber when the intake vale is open and the piston is moving down.

2. Compression stroke: The fuel and air mixture is compressed when both valves are closed and the piston moves up.

3. Power stroke: The piston is pushed down after the ignition of the fuel and air mixture.

4. Exhaust stroke: The burnt gases exit the combustion chamber via the open exhaust valve as the piston moves up.

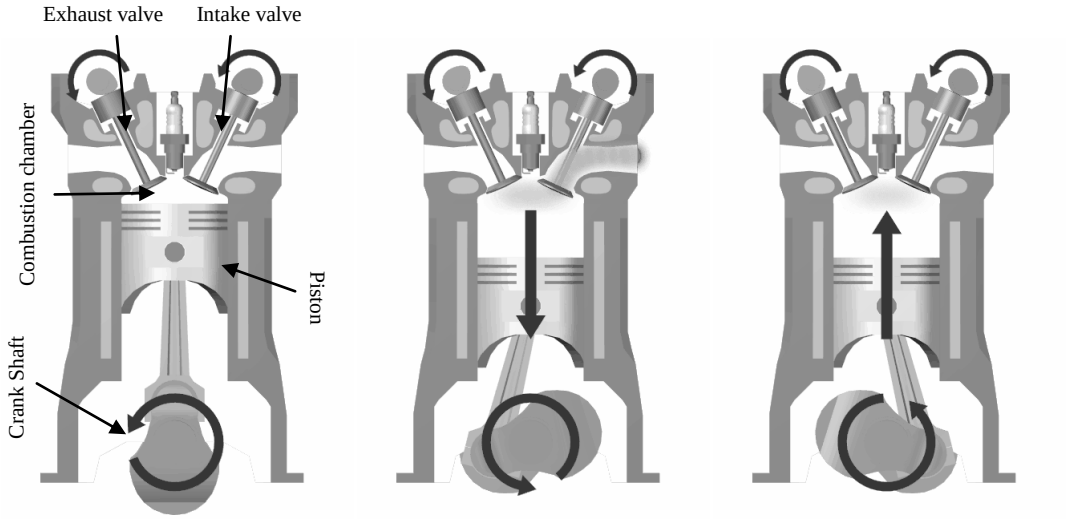


Figure 4-1: First revolution of a four stroke cycle with the starting position, induction stroke and compression stroke. (Illustrations created by Eric Piercing and released under GNU Free Document License)

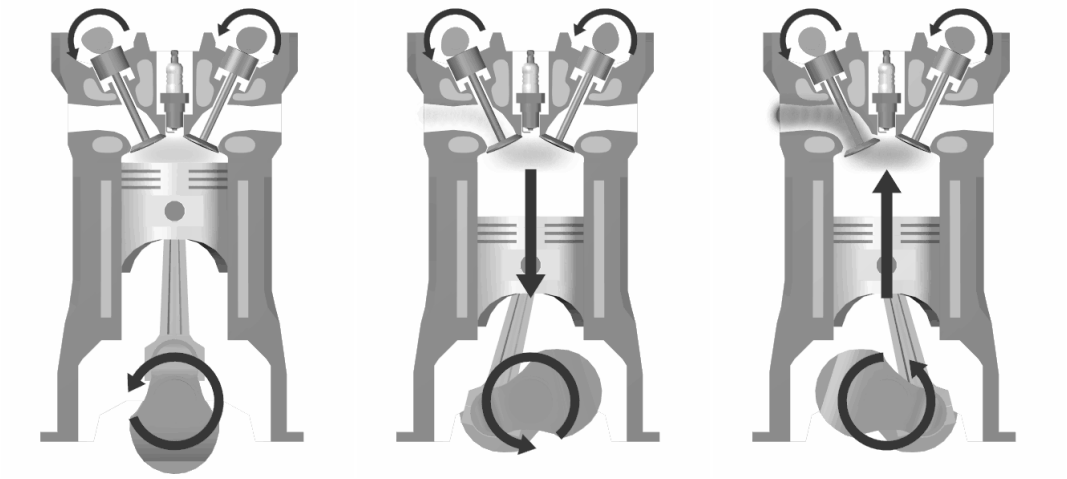


Figure 4-2: Second revolution of a four stroke cycle with the ignition, power stroke and exhaust stroke. (Illustrations created by Eric Piercing and released under GNU Free Document License)

4.2.2 Requirements for smooth engine operation

In order to ensure smooth engine operation over wide range of conditions and varying loads, various parameters have to be governed:

1. Fuel to air ratio: For an automobile engine in normal operation, the fuel to air ratio has to be varied within a range around the stoichiometric mixture ratio (the

stoichiometric ratio is the mixture ratio where after combustion, all of the oxygen reacts with all of the hydro-carbon fuel resulting in mainly carbon dioxide and water molecules) (Pulkrabek 1997). The early engines relied on carburettors to control fuel and air mixture ratios. In modern automobile petrol engines, this task is performed by computer controlled fuel injectors.

2. Ignition timing control: Depending on the speed of the engine, the moment the spark is generated has to be shifted relative to the angle of the crank shaft (Pulkrabek 1997). This is required because for maximum efficiency, the combustion of the fuel air mixture should be completed soon after the end of the compression stroke. This duration is dependent on the speed of the engine. On the other hand, the speed with which the combustion wave progresses through the fuel air mixture remains relatively constant regardless of the engine speed.

3. Valve timing: The opening and closing of the valves has to be closely synchronised with the crank shaft position. Failure to do this results in a drop in efficiency. In most of the existing engines, the valves are controlled through a mechanical linkage (e.g. timing belt, timing chain, etc). This approach has its limitations because the duration of the opening of valves cannot be modified based on the engine speed. Research is ongoing to develop engines with electronically controlled valves instead traditional mechanically controlled valves (Austen 2003).

4.2.3 Digital engine controllers in aviation

While electronic engine controllers have been used in automobiles for a long time, the aviation industry has been very slow in adopting these advances.

Bosch started the series production of Motronic engine management system (integrated fuel injection and spark ignition) in 1979 (Denton 1995, Bosch 2004). Since then they have become an integral part of every new automobile.

A lot the piston engines available for aircraft seem primitive compared to the engines being used in automobiles. It is believed by some that the aviation engine technology is lagging the automotive technology by 20 to 30 years (Dempsey 2011). The Pilot's Handbook of Aeronautical Knowledge published by the Federal Aviation Authority (FAA 2008) of the United States of America states:

> *"Most standard certificated aircraft incorporate a dual ignition system with two individual magnetos, separate sets of wires, and spark plugs to increase reliability of the ignition system."* (FAA 2008, page 6-14)

Some experimental aircraft have been using electronic fuel injection and electronic ignition since the mid 1980s with notable examples including systems and aircraft by Light Speed Engineering, and the Rutan Model 76 Voyager (first plane to circumvent the globe without refuelling) (K Savier 1995, Smithsonian). It is interesting to note that despite experimental engines being used early on, only a small number of aircraft piston engines with Full Authority Digital Engine Control (FADEC) have been certified mainly because of cost of development, certification and production (Smith 2007). The earliest FADEC equipped engines by Teledyne Continental Motors (TCM) were available by 2002 and by 2009 TCM had included a turbocharged model to their range of FADEC equipped engines, bringing the total to three (Continental Motors , Little 2009). Despite the perceived advantages of the FADEC in TCM's engine, they seem to have received a lukewarm response due to the high costs and availability in new airframes only (Bertorelli 2010). It is also interesting to note that unlike the automotive

engine controllers that typically use a single processor to control the entire engine, the TCM PowerLink FADEC utilises one processor per cylinder (TCM 2009). Arguably, this could be to reduce the software complexity running in each node to allow for easier software certification.

Another key player in FADEC equipped spark-ignited aviation engine arena is Lycoming. Their FADEC is more advanced than TCM's product as it also incorporates knock sensing for individual cylinders, making it easier to adapt for alternate fuels (Bertorelli 2010). It is possible that the inherent software complexities of this advanced design might be one of the reasons that are delaying the systems certification. In Aircraft Maintenance Technology's October 2008 edition, it was said that the Lycoming's iE2 should be FAA certified by the end of 2008 or early 2009 (Shearer 2008). A July 2010 press release by Lycoming states that the iE2 technology is still pending certification with no new press releases till the end of July 2011 that give the news of the successful certification from the FAA (Lycoming 2010).

4.2.4 Challenges in developing time-triggered engine controller

The engine control problem requires the controllers to synchronise operations like spark ignition, fuel injection, etc. very closely with the mechanical position of the crank shaft. Failure to do so can result in inefficiency and possibly even damage to the engine itself. This level of synchronisation is difficult to achieve using a fixed period polling of the crank position at low speed. High speed polling system, along with the number of tasks that need to be synchronised, cannot be classed as truly time-triggered systems. This point is demonstrated in the case study.

4.3 Brushless DC Motor Speed Control

To get a better understanding of the challenges involved in implementing any purely time-triggered control scheme for a brushless DC motor (BLDCM), the motor itself had to be studied first.

4.3.1 Motor structure

The permanent magnet brushless DC motor has the permanent magnets on the rotor and electromagnets on the stator. From control point of view, the only difference between the brushed DC motor and the BLDCM is that in addition to controlling the actuation signal (power supplied to the motor), the controller also has to take into consideration the sequence in which the coils of the BLDCM have to be energized to make the motor turn in the right direction (Atmel 2006, Brown 2001, Grasblum 2001, Hanif 2004, Yedamale 2003). In the case of the brushed motor, this excitation sequence for the electromagnets is provided by a mechanical arrangement (commutator and brushes) (Chapman 1985).

4.3.2 Commutation sequence generation

The commutation sequence generation presents unique challenges for time-triggered systems (Hanif 2004). As mentioned before, the brushless motor requires the electromagnets in the stator to be energized in a proper sequence to ensure that the motor continues to turn. In sensor based control, three Hall sensors are used to determine the position of the rotor relative to the stator (Brown 2001, Yedamale 2003). The system has to respond to a change in the position of the rotor sufficiently fast by updating the sequence in which the stator coils are energized. Failure to do this causes discontinuous motion at high speed (Brown 2001, Yedamale 2003).

Figure 4-3 shows the block commutation drive signals for the clock wise rotation of a typical three phase brushless DC motor. The hall sensor transitions are spaced 60 degrees of rotation apart, resulting in 6 transitions per revolution.

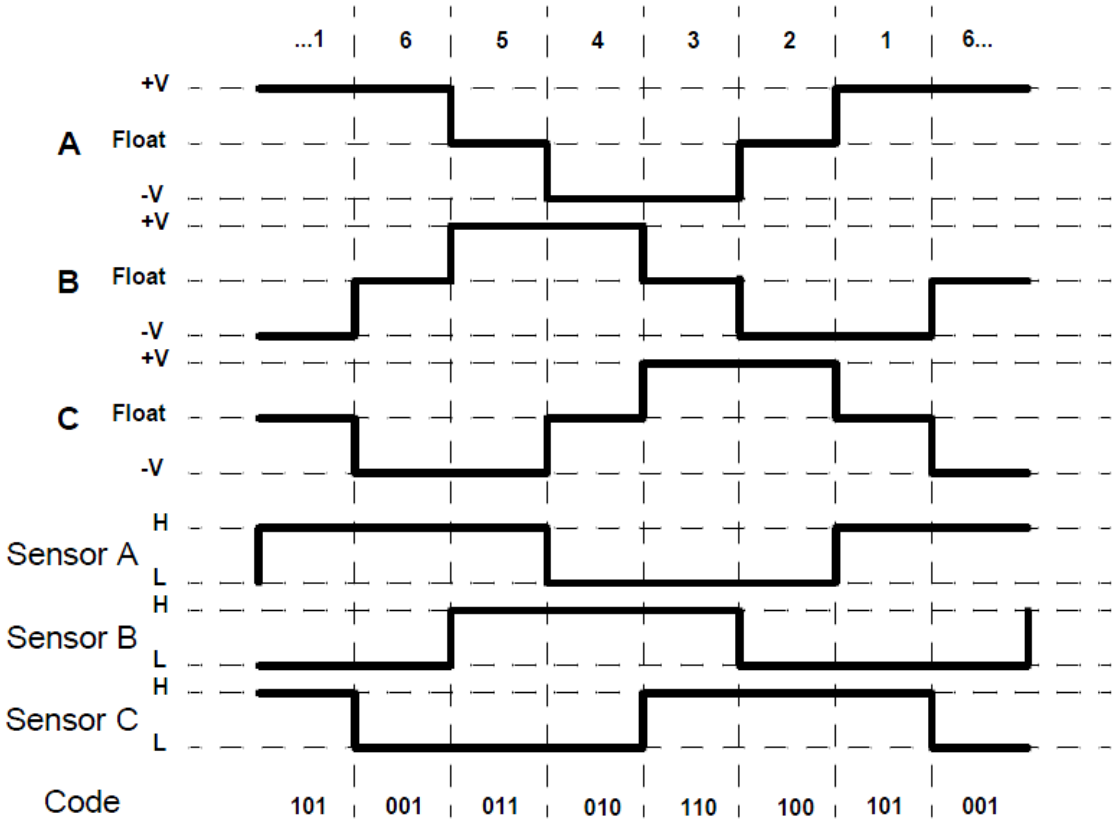


Figure 4-3: 3 phase brushless DC motor drive waveforms in response to hall sensor outputs. (Adapted from (Brown 2001)**).**

4.3.3 Challenges for time-triggered implementation

The speed controller implementation is simplified because the commutation sequence update and actuation signal generation can be treated as two disjoint control problems. Because of the typically high speed of these motors (20,000 to 30,000 RPM), the commutation update needs to be run pre-emptively at a very high rate.

4.4 Conclusion

This chapter presentes two challenging applications that are usually considered to be a better match for event-triggered architectures. Both of these applications require quick responses to events representing changes in the internal configuration of the plant that is being controlled. Implementing the control system with standard TT approach of polling is either inefficient or ineffective. The engine control problem needs the software to run in sync with the actual state of the engine, while the brushless DC motor problem needs a sufficiently high pre-emption rate for commutation update. Failure to meet these requirements can lead to inefficiency and even damage to the plant.

5 Making the Time-Triggered Cooperative Scheduler "More Dynamic"

This chapter examines the behaviour of the time-triggerd cooperative architecture and presents some of the ways in which its flexibility can be increased.

5.1 Classifying the Time-Triggered Cooperative Architecture

It should be noted that while the time-triggered cooperative (TTC) is presented as static cyclic executive scheduler, it can also be considered to be fully time-triggered fixed priority cooperative scheduler, provided tasks in one tick do not run across into the next tick frame.

5.1.1 Dynamic behaviour of the time-triggered cooperative scheduler

In the case of the TTC, a tick event is generated with each timer interrupt. The dispatcher then parses through the task array checking which tasks need to be run in the current tick. As the order of checking is fixed, the tasks in the task array have an implicit priority, with the first task in the array having the highest priority and the last task having the lowest priority. When tasks are added, the scheduler looks up the first available space in the tasks array and adds the new task to that position. Because of this, the task priorities depend on the order in which they are added to the scheduler. Figure 5-1 shows the implied priorities for a system with n cooperative tasks.

Using this knowledge, tasks can be added to conform to rate monotonic, deadline monotonic or arbitrary priority assignment to achieve the desired system behaviour.

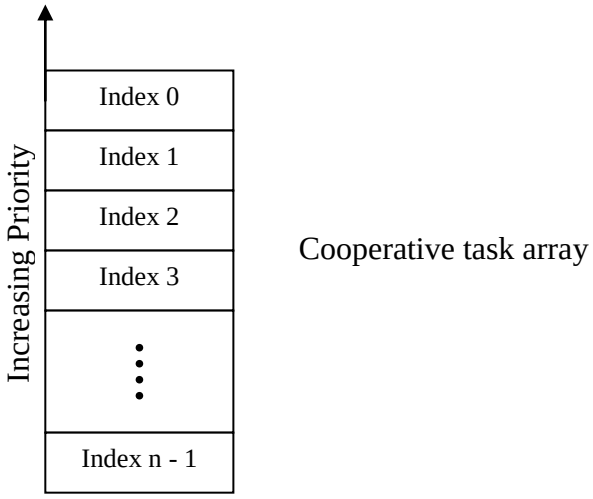


Figure 5-1: Implied fixed priorities in the TTC

5.1.2 Can the time-triggered cooperative scheduler still be called static?

If multiple tasks are set to be run in the same tick, the start time of the subsequent tasks will be effected by the variations in the execution times of the previous tasks in the same tick. Regardless of these variations, the task execution order remains the same. This ensures that all the precedence constraints that were being met at design time will still be met regardless of the level of jitter in the task timings.

In the case of a static table driven scheduler, all tasks have their own unique release times which, when configured properly, can be used to reduce jitter for all tasks. On the other hand, it can be argued that the table driven scheduler will have more fragmented slots of power-down mode if there is variation between the WCET of the tasks and their average case execution times.

These arguments can be clarified with the help of a hypothetical system with three tasks A, B and C. The temporal properties for scheduling these tasks are given in Table 5-1. Table 5-2 and Figure 5-2 give one possible schedule for a TTC architecture (assuming a

10 ms tick period) while Table 5-3 and Figure 5-3 give a similar solution for a table driven scheduler.

Table 5-1: Hypothetical task set with periods and execution times.

Task	Period	BCET	WCET
A	10 ms	2 ms	4 ms
B	20 ms	4 ms	4 ms
C	40 ms	3 ms	3 ms

Table 5-2: Possible scheduling of task set using TTC with a 10 ms tick period.

Task	Period (in ticks)	Initial offset (in ticks)	Implicit priority
A	1	0	High
B	2	0	Medium
C	4	1	Low

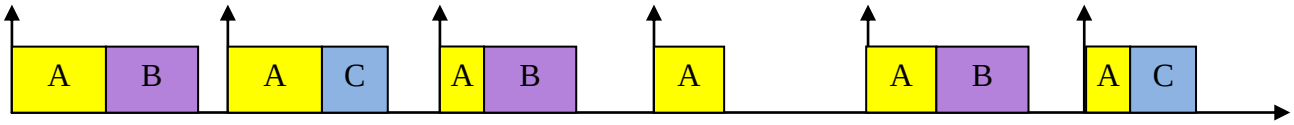


Figure 5-2: Task execution with TTC scheduler.

The assignment of the task priorities for the TTC schedule is in accordance with the task periods (i.e. rate monotonic).Variations in the execution time of A are reflected in the start time jitter of the subsequent tasks (task B in first and third tick and task C in the second tick). The offset is used for task C to avoid the total WCET of tasks in any tick going beyond the tick period.

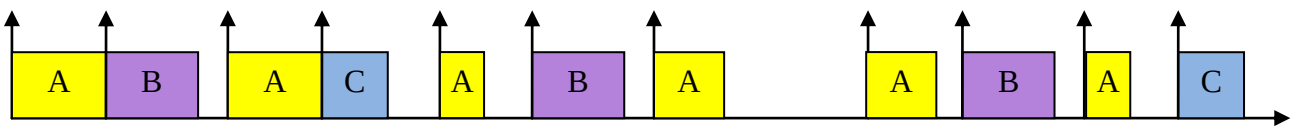


Figure 5-3: Task execution with table driven scheduler

Table 5-3: Possible schedule for table driven scheduler.

#	Start time	Task name
1	0 ms	A
2	4 ms	B
3	10 ms	A
4	14 ms	C
5	20 ms	A
6	24 ms	B
7	30 ms	A

For the table driven schedule, the start times for the tasks are chosen to allow for the variation in the execution times of task A. While this removes start time jitter for all tasks, the number of times the system goes to sleep in each cycle increases each time task A's execution time is less than its worst case estimate.

5.2 Multiple Operating States and Modes

The idea of switching between different statically scheduled task sets to cope with changes in operating modes of the system has been presented before (Xu, Parnas 2000, Baker, Shaw 1988). Systems running finite state machines can benefit from such an approach. For example, a building security system could have the following operating modes or states (see Figure 5-4):

- Standby State: In this state, is waiting to either be reconfigured or armed.
- Configure State: The system's configuration (e.g. access codes, etc) can be modified.

- Alarm Set State: The system is expected to scan various sensors to detect signs of intrusion. In addition to scanning the sensors, the system also needs to check for input to disarm.
- Alarm Triggered State: In this state, the system will be expected to sound an alarm and, optionally, contact authorities about the problem. It might also be desired that the system continue scanning the sensors to assess the level of the intrusion.
- Fault State: It would be expected that the system try to resolve the issue causing the problem or, alternatively, assist in the debugging of the problem by providing a failure report.

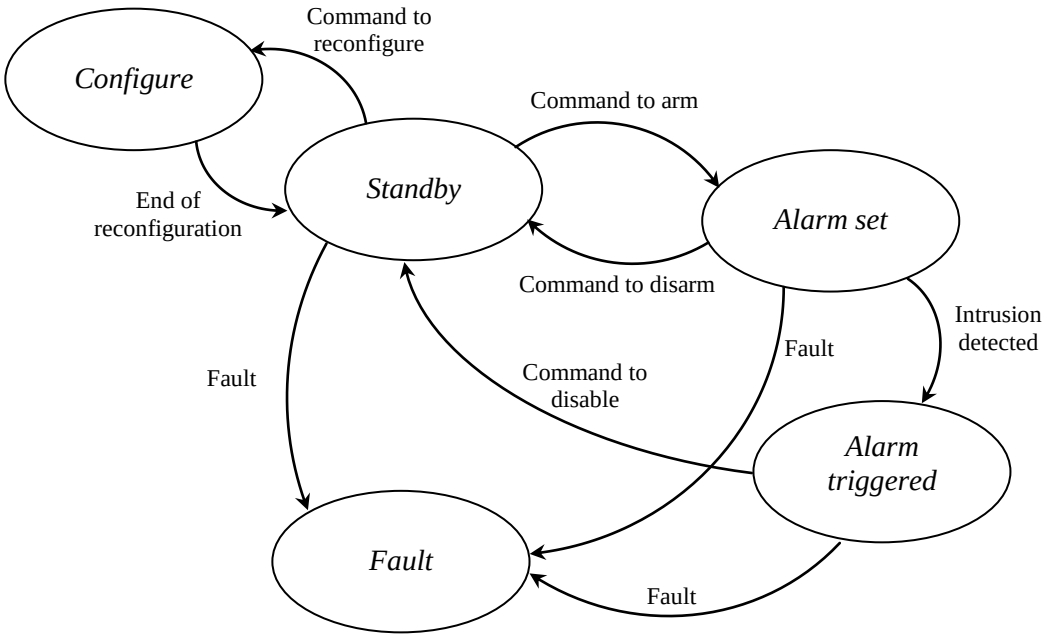


Figure 5-4: Finite state machine representation of a security system

The functionality for multiple states can be added easily to a scheduler derived from the TTC scheduler. A change of mode will effectively require the tick period to be updated along with switching the task array for the corresponding mode. While this mechanism

was initially suggested with table driven schedulers in mind, the memory size requirement to store these multiple tables puts this architecture at a disadvantage.

5.3 Different Segments in the Major Cycle

In embedded applications, it is possible to encounter different segments within the major cycle. These segments will have their own timing requirements and run tasks that are specific to it. In a hypothetical control problem, these segments and their timing requirements might be:

- Sampling: might require multiple ticks with a short tick period for sampling and smoothing some sensor readings.
- Control computation: might require one long tick for calculating the control output value.
- Actuation: might require` one short tick to minimize the task launch jitter for the actuation task.
- Housekeeping and communication with other nodes/systems: might have one or more moderate to long ticks till the next time the control process has to be updated.

Figure 5-5 shows one major cycle of such a control problem which first runs the sampling task ten times with a period of 0.5 ms, calculates the control output in the allocated 6 ms slot, has a single 1 ms slot allocated to the actuation task to allow jitter free in its execution and has the remainder for the 20 ms major cycle devoted to various housekeeping tasks

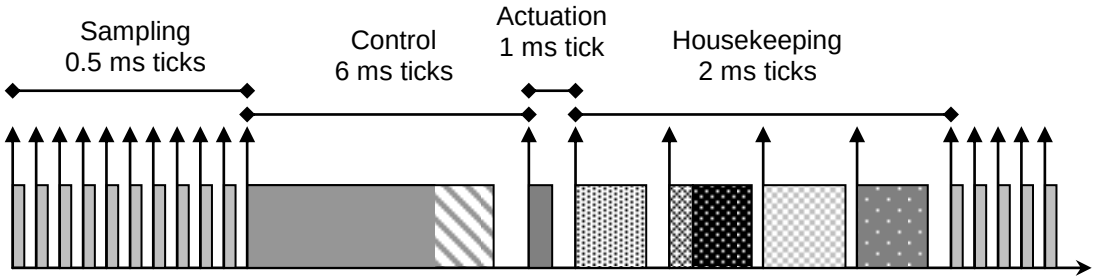


Figure 5-5: Hypothetical phases in a control node.

It is interesting to note that while this static schedule can easily be implemented with a look-up table with 16 entries for the entire major cycle, the TTC implementation will not be so straight forward. Because the tick period has to be the greatest common divisor of all the periods, it will have to be set at 0.5 ms (due to the sampling). In the real world where scheduling overheads cannot be ignored, this imposes significant overheads in the control, actuation and housekeeping segments where the high tick rate is not required or desired. To make the sampling task run in only the first ten ticks of the major cycle, ten instances of this task have to be added to the scheduler with the correct offsets (i.e. first instance with a 0 tick offset, second instance with a 1 tick offset, third instance with a 2 tick offset, etc.). This results in the same number of entries in the TTC's task array as the lookup table implementation.

The TTC can greatly benefit if there is a way to define segments with unique task sets and tick periods.

5.4 Varying the Tick Period and its Effects

Some control problems (e.g. internal combustion engine controllers) require that the execution of the sampling, control computation and actuation tasks remains in sync with the plant being controlled. From the scheduling point of view, it means that the period of the major cycle has to be varied in accordance with cycle period of the plant in question.

In the case of standard look up table based static schedulers, this would require that the entire lookup table is recalculated to ensure that the tasks are distributed evenly in the new major cycle period. This is not just a simple scaling operation as additional checks have to be included to ensure that the WCETs and deadlines of all tasks are met.

In the case of a TTC based architecture, the overall period of the major cycle can be changed relatively easily by just changing the tick period. With the change in the tick period, the release times of all the tasks will change proportionally. The minimum bound on the tick period will be dependent on one or more critical points in the major cycle which have the greatest cumulative WCET of tasks running in that tick. As long as all tasks can meet their deadlines at this critical point with the minimum tick period, the schedule should be valid for any tick period equal to or greater than this value.

Figure 5-6 shows how the tick period in the example in section 5.1.2 can be altered to keep the major cycle of the system synchronised with an external cyclic signal that is illustrated in the form of a ramp signal. In this system, it is assumed that the four ticks of the major cycle have to be keept evenly distributed with a total period that corresponds with an external cycle. The four parts of the figure show the execution of the tasks as the cycle period (both the external ramp signal and the major cycle composed of four ticks) gets progressively shorter, with part "***b***" showing a tick period similar to the 10 ms period in the original example.

In this example, the first and third ticks are the critical points where both task A and B are set to run resulting in a total worst case execution time for these ticks of 8 ms (WCET of 4 ms for both task A and B from).

This example is refered to in subsection 5.4.2 for a discussion of the validity of the four cycles shown in the figure.

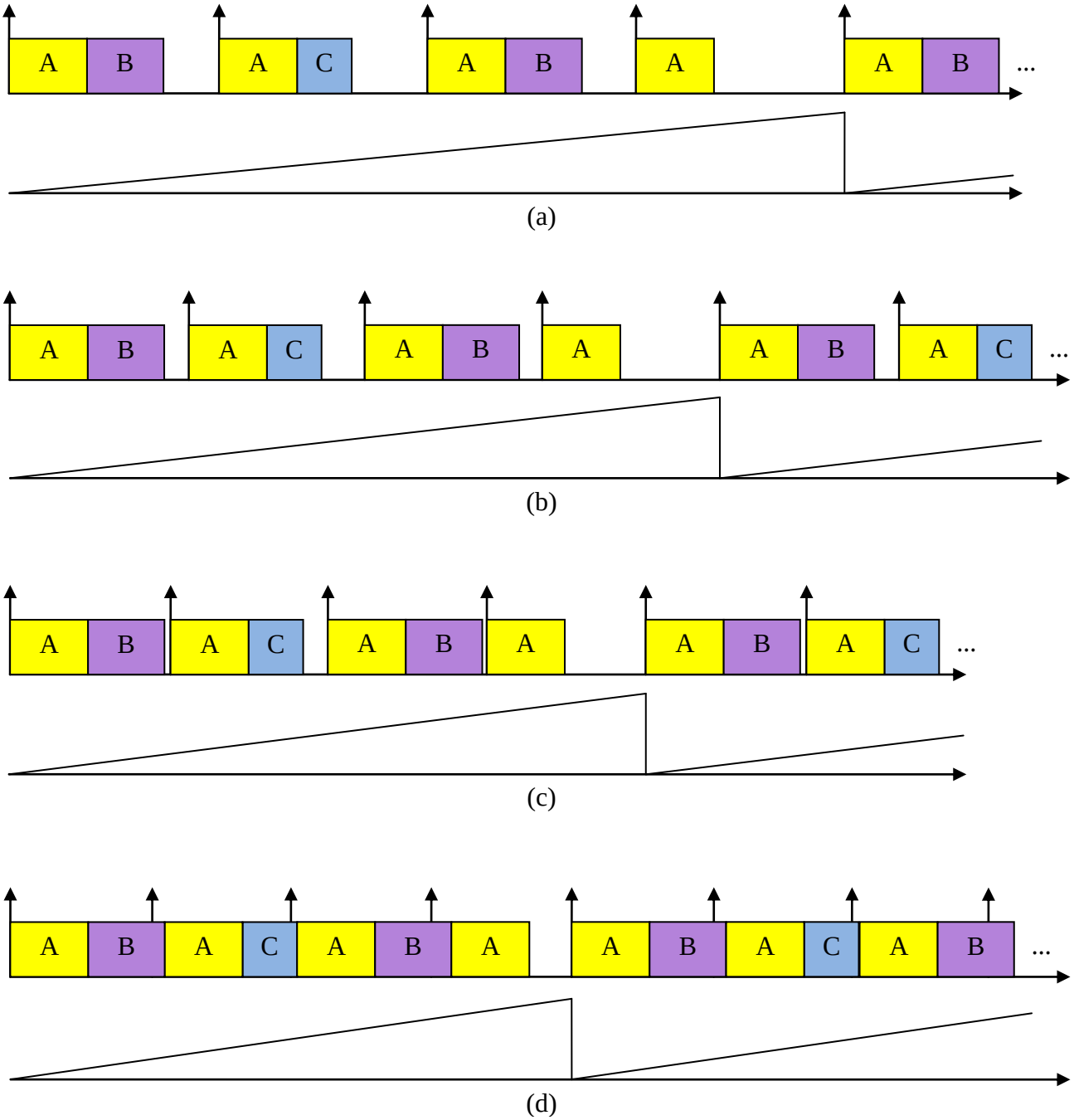


Figure 5-6: Graphical representation of how the major cycle period can be vaired to keep it in sync with an external cycle (represented by a ramp signal) by varying the tick period.

5.4.1 Is it still a static schedule?

Varying the tick period at run time results in a system in which the periods of the tasks change at run time. Such a system fails the classic definition of a statically scheduled time-triggered system as the state of the system cannot be determined at an arbitrary point in the future.

However, it is also not completely dynamic in nature because:

- The changing tick periods have no effect on the order in which the tasks are executed as tasks within a tick are run according to their implicit priorities and the tasks released in a previous tick have to finish before tasks released in the current tick are able to execute.
- As the only interrupt in the system is a timer interrupt, it is possible to determine when the next interrupt will occur.

A combination of these key features along with proper application design can result in a system in which the code execution paths can be traced and analyzed for safety critical applications.

Finally, unlike a purely event triggered system in which events generated by an external signal source drive the execution of tasks, the proposed architecture will continue to run tasks after a failure of the signal source allowing for natural error detection, opening the possibility for various fault recovery schemes.

5.4.2 Verification of a given task-set for a range of periods

Varying the tick period in a safety critical real-time application brings with it the challenge of guaranteeing that all the timing constraints for the tasks can be met for all possible periods.

The fact that the task execution order remains the same despite variation in task execution periods in a TTC architecture works in favour for providing such a guarantee. If a task set meets all the timing criteria in the worst conditions to be encountered, it should work for all conditions that are better than the worst case scenario. The worst case, from the scheduling point of view will occur when the tick period is the shortest as

this is when the schedule will be most tightly packed. Increasing the tick period from this minimum value will only make more slack time available in the major cycle resulting in lower processor utilisation.

Typically the criteria that are used for verification of a schedule in real time systems include:

1. All task deadlines are met: This is a very common criterion that is used for the verification of task set in static and dynamic architectures. If a tasks deadline is the same as its period, then each task should finish executing before its next release.
2. Low jitter tolerance for some / all tasks: In control applications, it is imperative that the jitter in the execution of some tasks is minimized or eliminated. Jitter minimisation imposes a harsher criterion on the scheduling than task deadlines (especially if deadline is the same as the task period).
3. Transient overloads: A transient overload in a TTC is when the tasks set to run in a previous tick are still executing when the next tick occurs. Depending on the application, transient overloads might be acceptable in some parts of the major cycle, resulting in more tolerant scheduling criteria.

All these criteria for the verification of a static schedule can be checked by modelling and analysis at the design time. Once the minimum allowable tick period is found for a task set, it is safe to use that task set at periods greater than this value.

Referring back to the example in Figure 5-6, if the deadline of each task is considered to be equal to its period, then all four cycles in the figure are valid.

If task A has low jitter tolerance, then only cycles "***a***" to "***c***" are valid as there is jitter in cycle "***d***" due to transient overloads. If there is low jitter tolerance for either task B or task C, none of the cycles will be valid unless the execution time of task A remains constant.

For a system with no transient overloads being allowed, the shortest acceptable tick period has to be greater than the sum of the WCETs of all the tasks set to run in the same tick.

5.5 Conclusion

This chapter expands the current state of the field by presenting original work on how the flexibility of the static time-triggered cooperative scheduler could be enhanced. The TTC architecture was analysed to understand why it is considered a static scheduler despite some dynamic tendencies. The concepts explored for increasing the flexibility of the TTC included:

1. Multiple operating modes and states (an existing idea mentioned in literature for the table driven static schedulers)
2. Multi segment cycles (a novel idea to bring a greater level of control on the task start times in a major cycle in a scheduler similar to the TTC)
3. Variable tick and cycle periods (a novel idea that has not been considered for static as well as dynamic time-triggered architectures)

While these features will help boost the flexibility of the statically scheduled architecture, it will result in additional scheduling overheads in the form of:

- Increased memory requirements to store the multiple task-sets and information for the states and segments.

- Increased CPU requirements for the scheduler to ensure that the modes and segments are changed correctly.
- Increased scheduler complexity to incorporate the variable ticks and multiple task-sets.

It can be questioned if an architecture is still static when the task and cycle periods are changed at run time. The guarantee that the task execution sequence remains the same helps to avoid some of the potential problems like task precedence constraints (i.e. where the output of one task is used as an input for another task) that might be faced in a more dynamic system.

6 The Time-Triggered Multi Phase Cooperative Scheduler

This chapter presents the architecture of the flexible time-triggered multiphase cooperative (TTMPC) scheduler and shows how the desired features highlighted in the previous chapter are incorporated in it

6.1 Phases as Building Blocks

The first two desired features outlined at the end of the previous chapter are very similar as both of them require the definition of multiple task sets with unique tick periods. The difference is in how or when the execution of one task set ends and that of the other begins.

The concept of a "phase" allows both these features to be grouped into a single implementation. In the context of the Time-Triggered Multi Phase Cooperative (TTMPC) scheduler, the phase forms the basic building block for static schedules. The TTC can be considered as a system with a single phase with fixed tick period and an indefinite duration. Each phase is configured with four elements (see Figure 6-1):

1. Task set: this includes the list of tasks which have to be run in that phase along with their periods and offset.
2. Tick period: each phase has its own unique tick period. This tick period can be varied at run time between maximum and minimum values set at design time.
3. Length of the phase: the length of the phase can either be a finite number of ticks or can be set as an indefinite length.
4. Next designated phase: for finite duration phases, this specifies the next phase the system should jump to once it reaches the end of the current phase.

While the finite duration phases switch over to the next designated phase automatically, the scheduler also needs a system call that can be used to enforce a phase change at any point during execution to implement finite state machines.

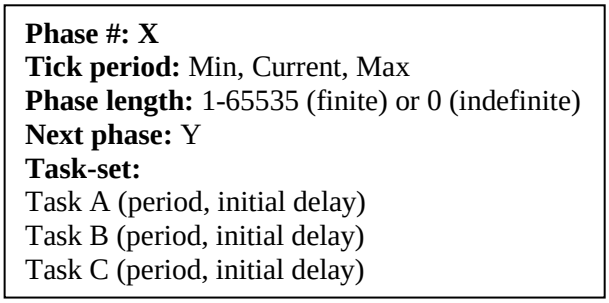


Figure 6-1: Graphical representation of a phase.

Finite length phases can be linked together to form multiple segments of a major cycle. The phase representation of the control example in section 5.3 is shown in Figure 6-2. At the end of each phase, the system should automatically jump to the next phase setting the correct tick period and swapping to the correct task set.

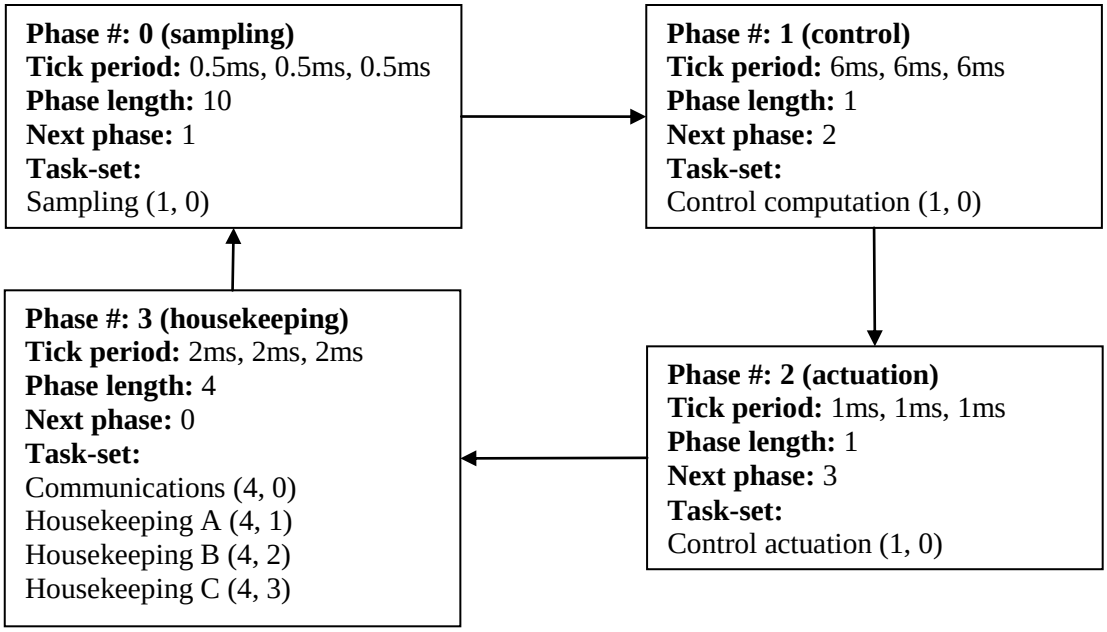


Figure 6-2: Phase representation of the example in Section 5.3.

Figure 6-3 shows the phase representation of the security system example in section 5.2. The phase changes have to be initiated by a system call indicating the phase to which the system should jump to at the end of the current tick.

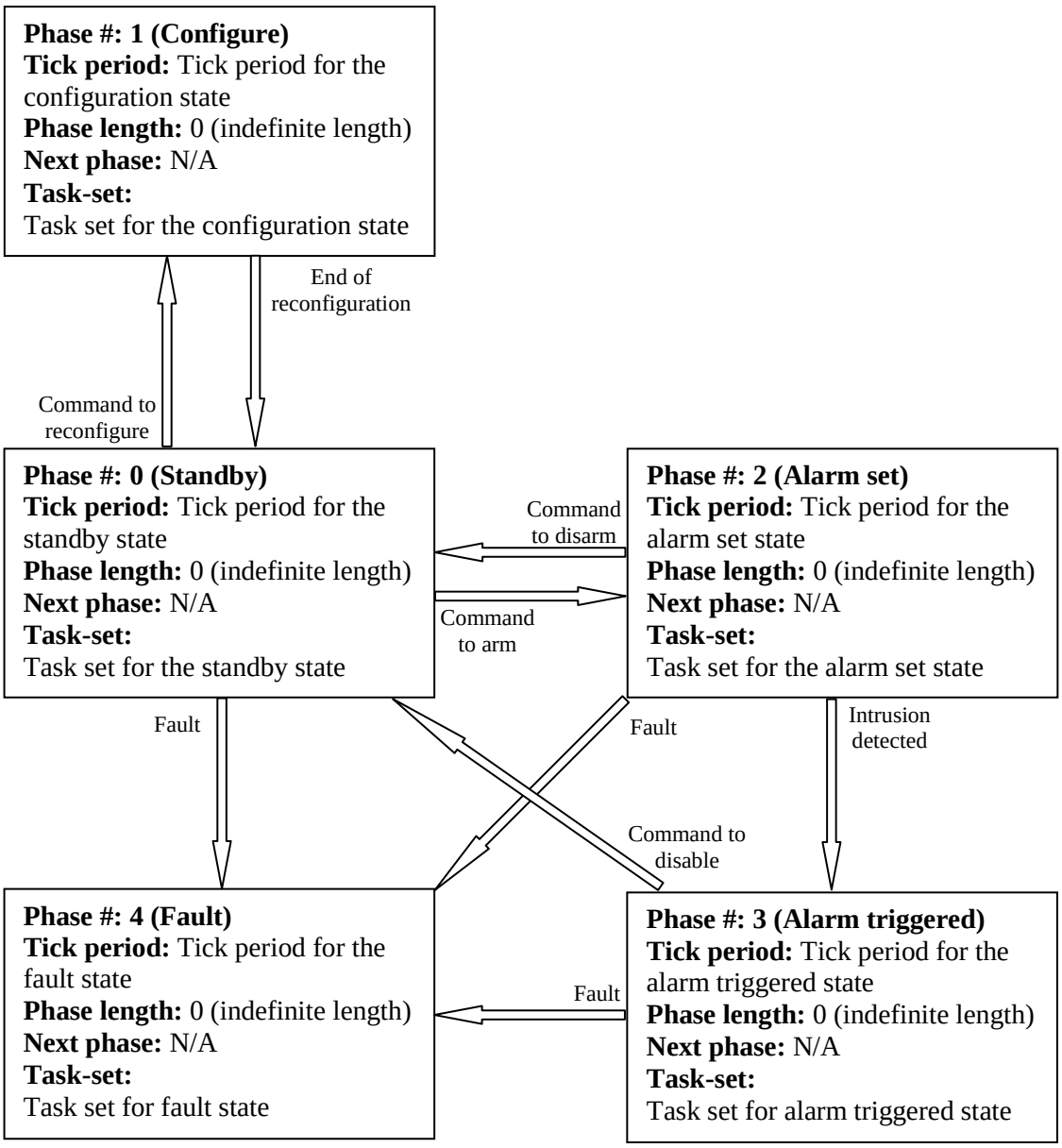


Figure 6-3: Phase representation of multi-state system in Section 5.2.

This system changes phases (and in effect the operating states) in response to external events. When viewed from a high level, the system is inherently event-triggered. The difference between a purely event-triggered and the multi phase time-triggered implementation become clear when the actual task executions are examined. In the

event-triggered implementation, all the tasks are run in response to events. Because of this, the execution sequence of tasks cannot be determined at design time. In the time-triggered multi phase setup, the tasks within a particular state are run in accordance with the phase's static schedule. Only the phase changes occur in response to events, but these are carried out at pre-determined points in the static schedules, allowing smooth transitioning from one operating state to another.

6.2 Phase Transitions under Transient Overload

Special care has to be given to how the transitions between phases are handled under transient overloads. Key questions to be answered are:

- After a forced phase change, should the remaining tasks in the currently executing tick be allowed to run?
- What to do in case of a back log of tasks belonging to more than one tick at a phase change (most probably due to a badly designed scheduled or under estimated task WCET)?
- If tasks from the old phase are still waiting, should the tick period be changed to the one required for the new phase.

6.2.1 Transient overload and automatic phase changes

Automatic phase changes are to be used for implementing segments in the major cycle for cyclic executives. All the tasks scheduled to run in the major cycle have to be executed in the order they are supposed to run and no tasks should be skipped because of the overload.

The tick periods should change to correspond to the new phase that is being delayed due to the transient overload. If this is not done, the timing of the major cycle could be affected.

Some sort of mechanism is needed to keep track of the automatic phase changes to ensure that all the tasks that were delayed are run in the order they were supposed to run despite the overload. This is fairly straightforward because of the periodic nature of the changes, and should allow the actual task execution to lag by more than one phase transition and still have a chance to catch up.

6.2.2 Transient overloads and forced phase changes

Forced phase changes are to be used to jump from the cyclic executive of the old state to the cyclic executive of the new state. Such a change request will usually be in response to a command or change in the operating conditions making it desirable to switch to the new mode as soon as possible. However, in order to limit the points in the cycle at which a change can take place and to make the modelling of forced phase changes easier, the tasks in the current tick should be allowed to finish before the switch to the next phase.

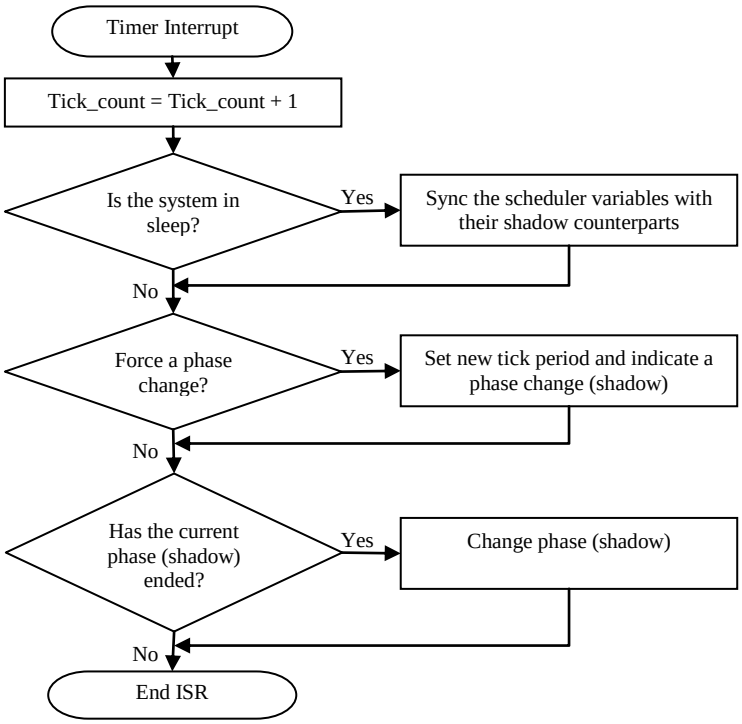
Because of the unpredictability of the transitions (i.e. it might not be known in advance when an event requiring a forced phase change might occur), the only way to keep track of the changes would reqire maintaining some sort of log of these changes.

6.3 Key Parts of the Time-Triggered Multiphase Cooperative Scheduler

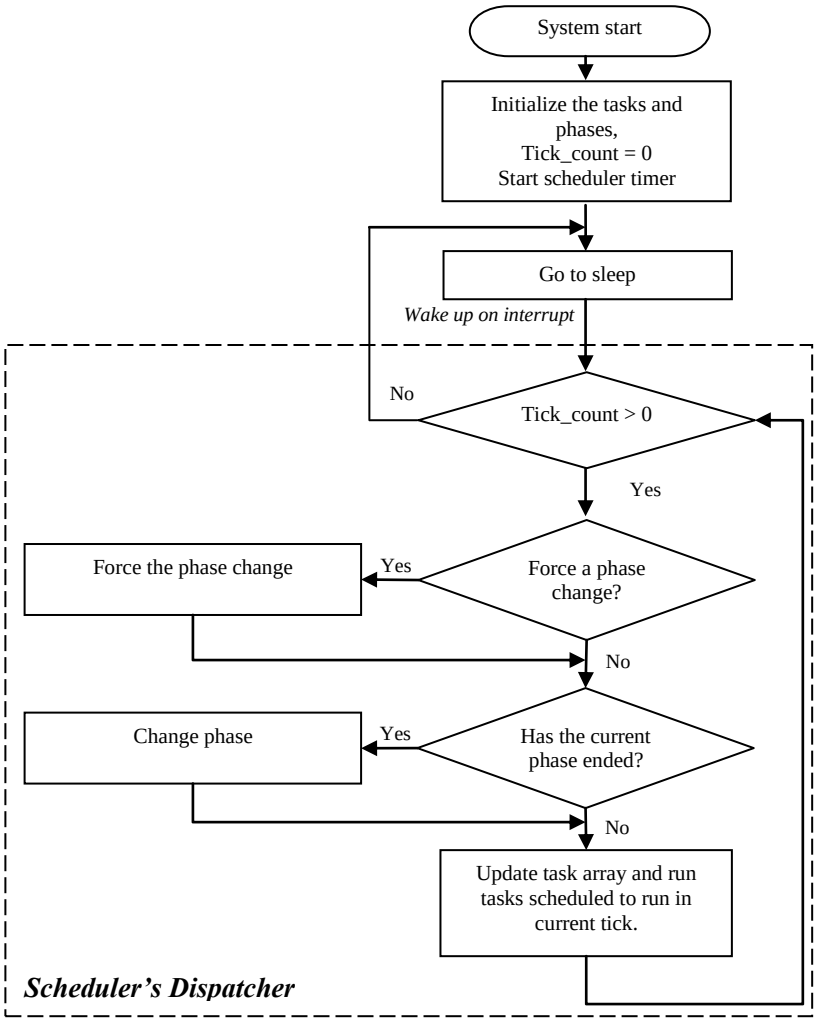
Like the TTC on which it is based (see section 3.2.2), the TTMPC is composed of two parts.

1. Scheduler Timer ISR

The timer's ISR is used to indicate the occurrence of a tick to the scheduler's dispatcher. The timer's ISR also maintains and updates the shadow state of the scheduler. The actual state of the system is synchronised with this shadow state when all tasks are finished within their allocated ticks. This allows tasks overrunning the tick boundary so as not to result in missing any pending task in the previous tick at phase boundaries (see Flowchart 6-1 for details of the Scheduler update function).



Flowchart 6-1: Flowchart of the TTMPC scheduler's Timer ISR.



Flowchart 6-2: Flowchart of the TTMPC system's main structure.

2. Scheduler dispatcher

The TTMPC's dispatcher is derived from the dispatcher of the TTC. The major difference between the two dispatchers is that in the TTMPC, it has to track phase changes in addition to updating the correct task array and running tasks from it. The implementation of TTC from which the TTMPC is derived allows tasks to overrun tick boundaries. This allows the TTC to schedule tasks with a worst case execution time (WCET) larger than the tick period provided another task with a shorter period is not blocked. In order to retain this flexibility, and maintain high determinability (i.e. ensuring that all tasks which are due to run in a tick are

completed regardless of a phase change at its end), the dispatcher has to keep separate track of phase changes.

6.4 Features of the Time-Triggered Multiphase Cooperative Scheduler Implementation

To summarize, the current version of the TTMPC scheduler (version 3) has the following features:

- Ability to define multiple phases with unique task sets, tick rates and phase lengths.
- Automatic and forced phase change mechanisms are implemented.
- Upon the start of a new phase, the originally specified task offsets are restored.
- Indefinite length phases are possible by specifying a phase length of zero.
- A shadow system state is maintained in the scheduler update (timer ISR), while the actual system state is maintained in the dispatcher to allow for transient overloads across phase boundaries.
- In the case of a forced phase change under transient overload, the currently executing tick is allowed to complete before the phase change is enforced. Any additional pending ticks and phases due to the overload are ignored.
- In the case of an automatic phase change under transient overload condition, the scheduler continues to run all the tasks in that major cycle in the order they were supposed to execute in and should catch up if there is sufficient slack time in the major cycle.
- If a forced phase change is requested in the last tick of a finite duration phase, the system jumps to the phase indicated by the forced change request.

- Tick periods of each phase can be varied at run time between the minimum and maximum values specified at design time.

6.5 Limitations of the Design

Despite the flexibility that is obtained by introduction of phases, the system suffers from some key limitations:

- It is not possible to schedule task sets where WCET of the longest task is more than the period of another task (non liquid task sets).

- While the system will be able to recover from a task overshooting its WCET estimate if there is sufficient slack in the schedule, there is no way to recover from a catastrophic overrun (e.g. a hardware or software fault that results in an infinite loop due to the failure of a component of the logical test condition for a software loop).

6.6 Possible Operating Configurations of the Time-Triggered Multiphase Cooperative Scheduler

The configuration of the scheduler and in general, the system, has an impact on the predictability of the end product. The possible operating configurations allowed by the TTMPC are:

1. Single Cycle multiphase with Fixed Tick periods Cooperative (SCFTC): All systems in which there is a single major cycle that has one or more segments with all segments running with fixed tick periods fall under this configuration. The TTC scheduler can be considered a special case of SCFTC with a single segment in the major cycle.

2. Single Cycle multiphase with Variable Tick period Cooperative (SCVTC): The difference between SCCFT and this configuration is that one or more segments in the major cycle have variable tick period.
3. Multi Cycle multiphase with Fixed Tick periods Cooperative (MCFTC): All systems which have multiple major cycles where all the tick periods are fixed fall under this category.
4. Multi Cycle multiphase with Variable Tick period Cooperative (MCVTC): All systems which have multiple major cycles where at least one cycle has variable tick period fall under this category.

6.6.1 Predictability and determinability of configurations

While time-triggered statically scheduled systems are considered highly predictable, the overall predictability is expected to drop with increase in the level of run time flexibility.

Generally, it will be more difficult to predict the operating state of multi state systems at any arbitrary point in the future if the state changes are in response to events. This also applies to a system in which there is one major cycle for all the states, but the code that is executed in the tasks depends on the operating state of the system (e.g. switch case statements in C / C++ etc.).

In the statically scheduled setup, the ability to predict or determine what code the system is running at the current time depends on the amount of information that is available:

For the SCFTC, assuming accurate system timer, only the time at which the system was powered on is required to determine the operating mode of the system and what tasks are being run.

MCFTC systems require the tracking and logging of the events that influence the operating states along with the knowledge of the power up time to determine the operating mode of the system.

While it might not be possible to determine which tasks are being run at what moment in time in variable tick cooperative execution systems (assuming the system is varying tick periods to stay in sync with an external signal), the current operating state can still be predicted if the relevant information is logged (e.g. power on time, events that cause state changes, signals / events that affect the tick period, etc.).

Even in an MCVTC system, the remaining time till the next timer interrupt can be easily calculated. This results in better predictability than an event-triggered system where it might not be possible to determine when the next event would occur.

The forgoing discussion on the predictability and determinability of the various configurations may be summarised as in Table 6-1.

Table 6-1: Summary of the predictability and determinability of statically scheduled cooperative systems.

System	Required info	Predictability and determinability
SCFTC	Power on time.	Current state can be accurately determined. Determinable till any arbitrary point in the future.
MCFTC	Power on time. Log of all events that affect operating state.	Current state can be accurately determined. Determinable till the end of the current cycle / point at which state change might occur due to an event.
SCVTC	Power on time. Log of all events / signals that affect the tick period.	Current state cannot be determined but can be predicted. Predictable till the point at which the new tick periods are calculated. (Predictable beyond that if certain estimates can be made with regard to the new tick periods.)
MCVTC	Power on time. Log of all events that affect operating state. Log of all events / signals that affect the tick period.	Current state cannot be determined but can be predicted. Predictable till the point in the cycle at which the new tick periods are calculated or state change can occur (which ever point comes first)

6.7 Conclusion

This chapter introduces the concept of phases to combine two of the flexibility enhancements identified in the previous chapter into a single construct. These phases can be used as a building block for both multi state systems and multi segment cycles.

The phases and the variable tick and cycle periods are incorporated into the time-triggered multiphase cooperative scheduler (TTMPC) derived from the time-triggered cooperative (TTC) scheduler. A discussion on the predictability of the new architecture concludes that while the predictability is adversely affected by the flexibility, even in the most flexible configuration, the exact moment of the occurance of the next event in the system can always be determined in advance. This results in better predictability than an ET design where, while statistical limits could be placed on some of the event, the exact time till the next event cannot be determined.

7 Engine Synchronisation Case Study

This chapter introduces the software and hardware setup of the engine synchronization test-bed and demonstrates how the flexible time-triggered implementation might be used as an alternative platform for a challenging application.

7.1 Introduction to the Case Study Setup

As mentiond in section 4.2.4, the primary responsibility of the software in the engine control unit is to perform actions (e.g. fuel injection, spark ingintion, etc) at specific points in the engine's cycle. This case study tries to assess the viability of the flexible time-triddered architecture as a foundation for the engine control applications. This is done by guaging the ability of various architectures (both time-triggered and classical event-triggered) to synchronise task executions with an external crank signal.

In the case study setup, shown in Figure 7-1, one microcontroller was used to run one of the three synchronisation architectures that were tested while another microcntroller generated the crank signal against which the synchronisation performance was measured.

The architecture test platform consists of an STM32M103RB microcontroller. For these tests, the microcontroller was clocked at 72 MHz using an 8 MHz external oscillator and the internal PLL. The details of the peripherals of the microcontroller can be found in its reference manual (STMicroelectronics 2011).

The crank signal generator runs on an LPC2129 microcontroller. For this implementation, this controller was clocked at 60MHz with its timers operated at a resolution of either 1µs or 0.1µs using the timer prescalers. The details of the peripherals of the microcontroller can be found in its reference manual (NXP 2008).

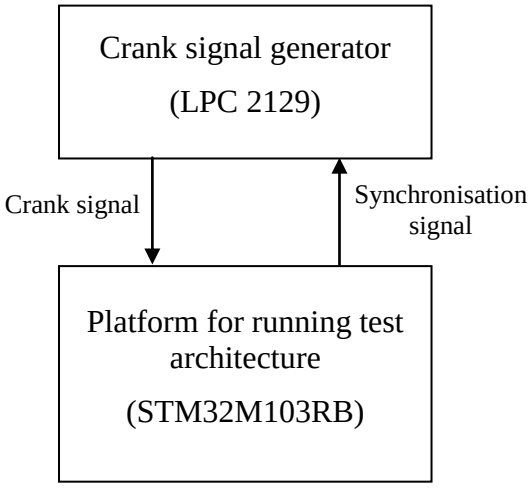


Figure 7-1: Block representation of the case study setup for evaluation of various architectures under identical conditions.

The crank signal gernerated in this case study is modelled on the Rover M series 1994cc DOHC (double overhead cams) engine. This engine has an idling speed of 850 rpm with the red line speed of 6250 rpm (John S Mead 1991).

Most of the information about the operation of this engine and its controller was obtained from notes made when the engine was installed in the departement, a chapeter from an anonymous manual (RoverMEMS - MPi/SPi) and experimental verification of these documents.

In this engine, the engine control unit (ECU) is responsible for the amount of fuel being injected and moment when the fuel air mixture in the cylinders is ignited. These operations have to be synchronised with the internal state of the engine. Failure to do this can result in reduced efficiency, higher emissions, and possibly damage to the engine. (RoverMEMS - MPi/SPi, Jinnelov 2002)

The ECU determines the rough amount of fuel to inject and the moment of spark ignition based on readings from the manifold absolute pressure (indicating the load on

the engine) and speed of the engine and its internal position (derived from the crank sensor). These values are then fine-tuned based on the temperature of the air, fuel and coolant, detection of knocking, etc (RoverMEMS - MPi/SPi).

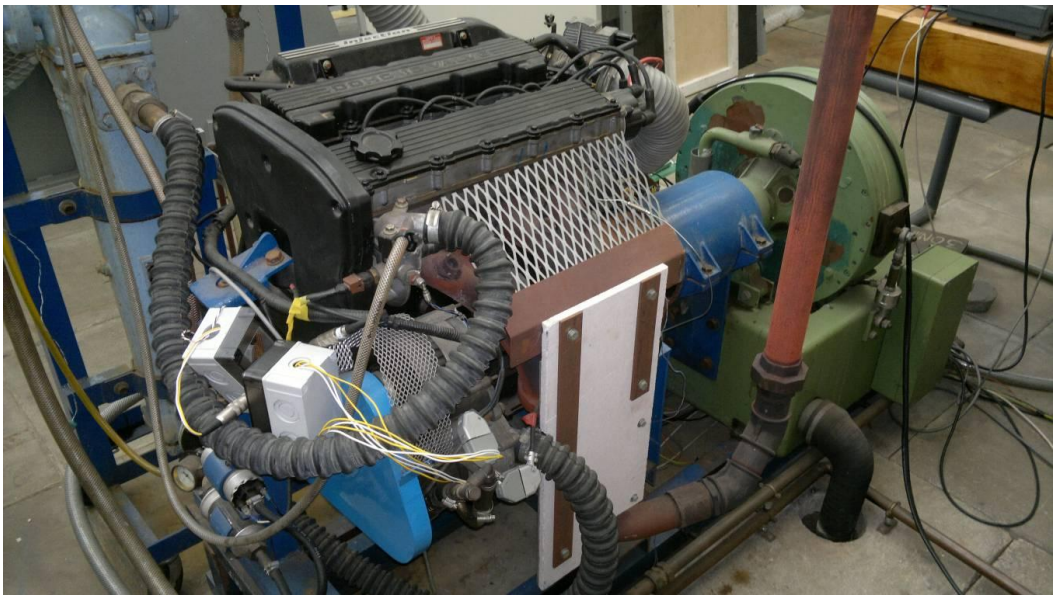


Figure 7-2: The Rover M16 test bed.

The engine test-bed comprises of the above mentioned engine connected to a dynamometer (see Figure 7-2). Most of the data used for this work was extracted by tapping into the circuitry of the engine test bed and logging various sensor values and control actuations at 80,000 samples per second while the engine was operated at various speeds and load combinations. These logs include the following signals:

1. Manifold absolute pressure.
2. Crank sensor output after a basic chopper and comparator circuit.
3. Throttle plate angle.
4. Oxygen sensor.
5. Temperatures of air, fuel and coolant.
6. Knock sensor.
7. Supply voltage.

8. Injector actuations.
9. Ignition coil primary side.

The raw captured data was processed using Matlab script files to extract lookup tables and other modelling information.

7.2 Crank Angle Sensor

The crank angle sensor provides the internal position of the engine. Its output consists of a pulse for every 10 degrees of rotation with missing pulses to indicate the top or bottom dead-centre for the pistons (a total of 17 pulses for every half revolution).

The times at which the signal rises and falls can be used to determine if a top dead-centre condition (TDC) is observed and how many pulses have arrived since the last TDC condition, giving the internal position of the crankshaft.

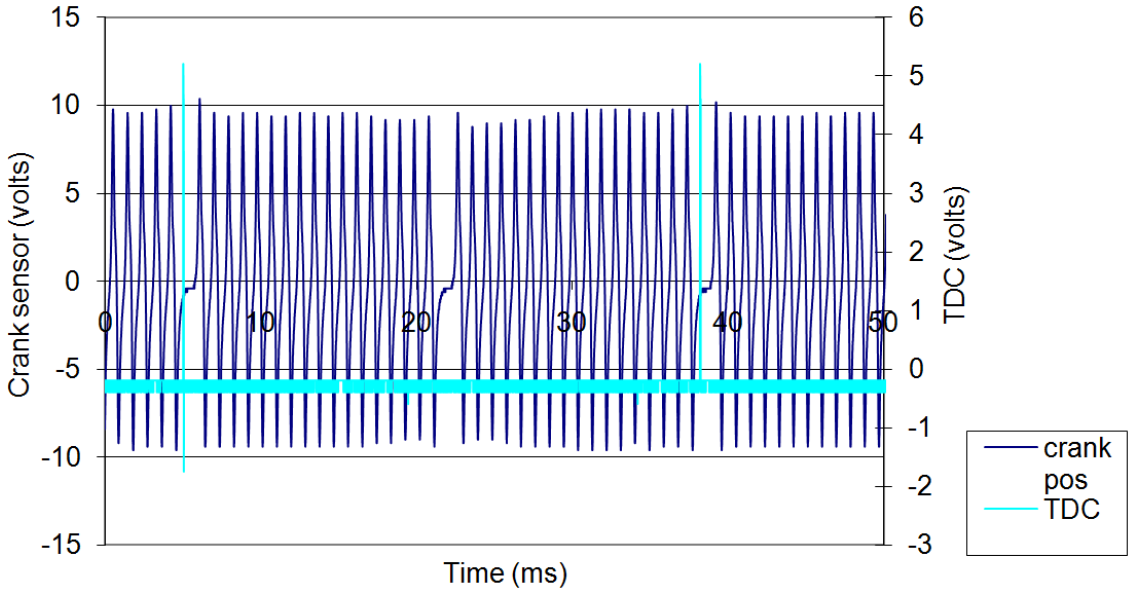


Figure 7-3: Plot of the actual crank sensor (one signal of the differential pair) and an external TDC sensor at 1800 RPM.

Figure 7-3 shows one of the differential pair of signals from the crank sensor at 1800 RPM while the pulse from an external top dead centre sensor for cylinder 1 can also be

seen (note the missing pulse at the TDC and BDC). This signal is passed through a chopper and comparator circuit before it is connected to the microcontroller. Figure 7-4 show the processed signal that was captured at an engine speed of 1500 RPM.

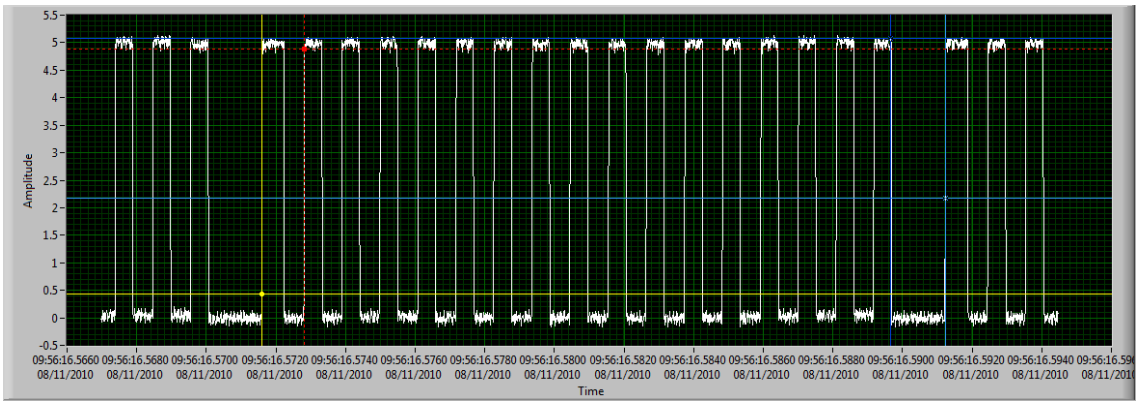


Figure 7-4: Captured crank waveform after chopper and comparator circuit at 1500 RPM.

7.2.1 Implementing the crank sensor interface

In the target controller hardware, the times at which the rising and falling edges occur in are noted using a free running timer and two of its capture inputs. The rising and falling times are formatted into pulse data with the start time, high and low durations or each pulse being stored in a FIFO buffer. Finally, glitch suppression logic is used to merge the glitches into its adjacent pulse. The high and low durations of the contents of the glitch free pulse buffer are then used to update the crank interface state machine. Figure 7-5 shows the data flow diagram for the above mentiond process.

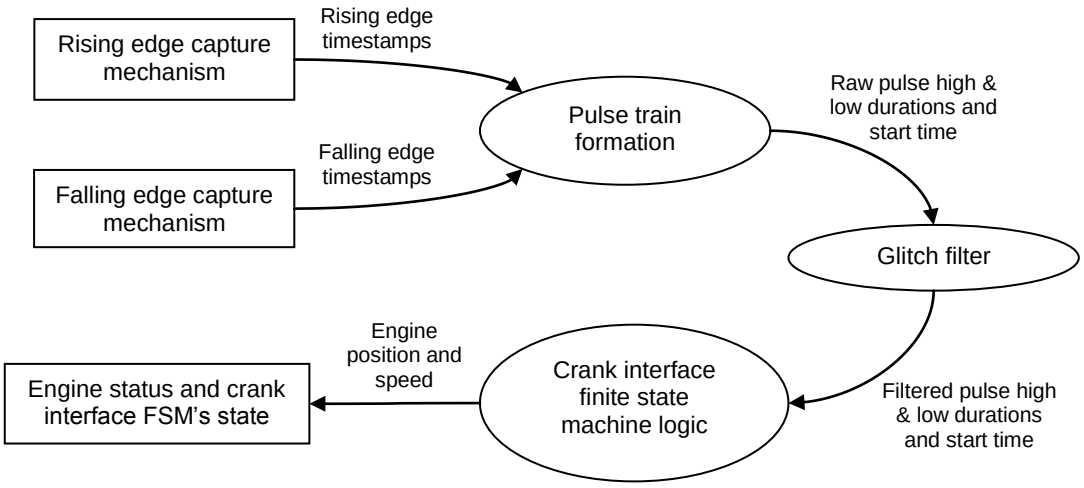


Figure 7-5: Crank interface data flow diagram

The engine status and crank interface's finite state machine (FSM) has the following four states:

1. No Sync state: This is the default state in which the system is waiting for detection of valid pulses with a duty cycle in the 40% to 60% range. This corresponds to the duty cycle of the majority of pulses in a crank cycle and is used to overcome the startup noise enounctered when the crankshaft is stationary or turning below a certain speed.
2. No lock state: After the start / resumption of valid pulses, the system waits for a TDC condition to occur. This is needed to synchronise the controller with the position of the crankshaft.
3. Tentative lock state: This state is used to verify if the controller software is really in sync with the crankshaft. If the TDC condition is encountered again after the correct number of normal pulses, it is assumed that the crank interface logic is in sync with the crank shaft with proper identification of the start of the cycles.
4. Locked state: This state is used for the normal operation of the controller. The control software should inject the fuel and ignite the spark only in this state to

try to ensure that the engine is not damaged by an incorrect synchronisation with the crankshaft.

In order to make the crank interface software modular to enable its use in the event triggered as well as time-triggered architectures with as few modifications as possible, it was split into two tasks:

1. Crank sample: This part is responsible for getting the edge time stamps from the timer capture registers / buffers into the rising and falling edge timestamp buffers.
2. Crank state: This part is responsible for forming the pulse train, applying the glitch filter logic and updating the crank state machine.

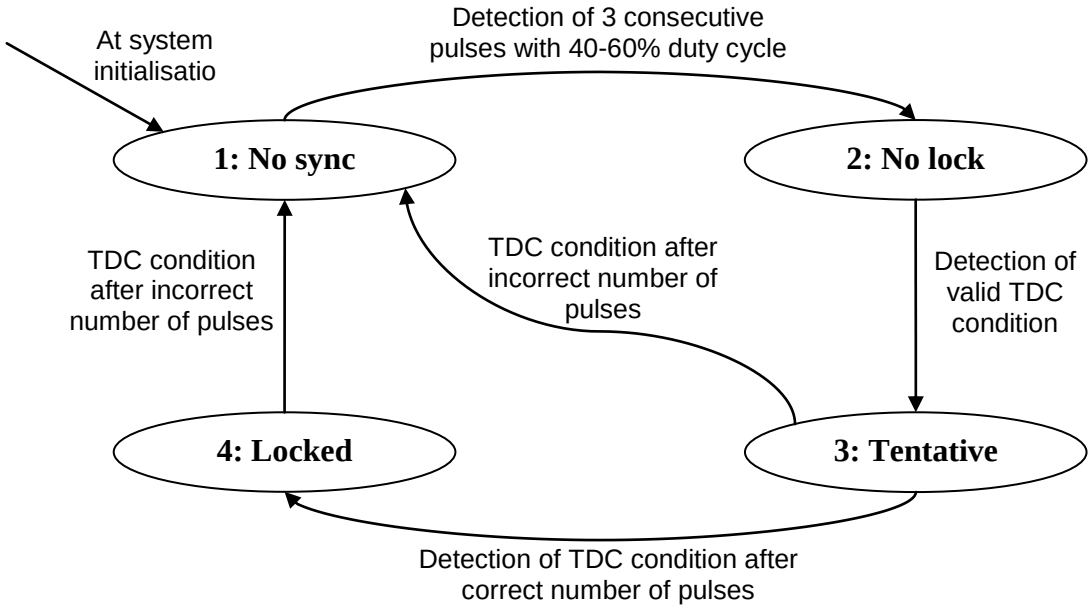


Figure 7-6: State machine of the crank interface logic.

7.2.2 Start of cycle test condition

The detection of the TDC condition is at the heart of the crank interface logic. During analysis of the captured data, it was noted that there were two distinct patterns that

could be observed in the crank signal. The test conditions were derived from empirical data collected from the engine and are as follows:

1. Normal operation: Under normal operation, the captured crank signal consisted of 16 pulses with similar high and low durations followed by one pulse with a low duration significantly greater than the high duration, indicating the missing pulse at the top dead centre. The following test condition is used to check for normal TDC.

   (Current pulse low duration > 2.2 * Previous pulse low duration) AND
   (Current pulse high duration < 1.7 * Previous pulse low duration) AND
   (Current pulse high duration > 0.6 * Previous pulse low duration)

2. During starting / stall at low speeds: When the engine is being cranked at start-up, a different pattern is observed. By the combination of low angular momentum, energy needed to compress the air in the compression stroke and absence of fuel injection and ignition during the cranking stage of start-up, the high duration of the first pulse is significantly longer than the high duration of the previous pulses. The following test condition is used to check for starting TDC:

   (Current pulse high duration > 2.3 * Previous pulse high duration) AND
   (Current pulse low duration < 1.7 * Previous pulse low duration) AND
   (Current pulse low duration > 0.6 * Previous pulse low duration)

With correct fuel injection and spark ignition, the transition from the condition encountered during starting to the normal operating condition usually takes place within one cycle.

7.2.3 Worst case execution time analysis of the crank interface task

WCET measurements were made early on in the development of the crank interface logic. These measurements were required for the proper scheduling of the tasks in the system. In order to conserve the limited resources in the microcontroller, the free running crank interface timer with a pre-scalar of 10 µs was used for these timing measurements by noting its value at the start and end of the tasks to be measured. Over multiple runs of the crank interface logic, the worst case time noted was 170 µs. Taking the scheduling overheads into account, this placed a limit of 180 µs on the lowest tick period that could be used while ensuring that there were no transient overloads in the system.

Despite the WCET being a measured value, this timing can be considered as a very conservative estimate for the final version of the crank interface tasks. This is mainly because of two key differences that are in the code whose timing was measured and the final version of the code:

1. If the validity of the order of rising and falling transitions cannot be verified (i.e. the time stamps indicate that the rising and falling edges are not alternating) in the crank state task. On detection of the erroneous condition, the final version resets all queues and buffers in the crank interface logic while the initial version waited for newer values to be added to allow the erroneous sequences to flush out automatically. The steps taken in the final version allow a quicker recovery from the erroneous condition while significantly reducing the WCET of the crank tasks.
2. A glitch was discovered in the DMAs being used to capture transitions for the time-triggered implementation. Steps taken to resolve the DMA issues resulted

in the elimination of the error that was causing the validity check failure of the rising and falling edges.

7.3 Implementations of Architectures for Performance Comparison

In order to evaluate the performance of the flexible time-triggered architecture, it needs to be compared to other architectures. The three architectures compared in this case study are:

1. Event-triggered.
2. Classic time-triggered polling approach.
3. Flexible time-triggered.

7.3.1 Event-triggered implementation

The event triggered implementation has one interrupt source and has an interrupt driven super-loop architecture. The execution times of the ISR and non-pre-emptable critical sections have been kept to a minimum, with the bulk of the processing being done in the super-loop.

Rising and falling edges in the output of the crank sensor trigger hardware capture events in the timer peripheral. Each capture event causes the value of the timer to be noted at the time of the event, giving a corresponding timestamp. In addition to the capture event, the rising transition also triggers the interrupt service routine (ISR).

The timer capture ISR is used to copy the most recent rising and falling edge transitions into a shadow FIFO buffer of the most recent transitions. After the ISR, the main logic first creates a copy of the shadow buffer or transition timestamps and then processes them to update the crank interface state machine. This is done to allow the ISR to

capture additional transitions into the shadow FIFO buffer while it is in the middle of processing the current batch of transitions.

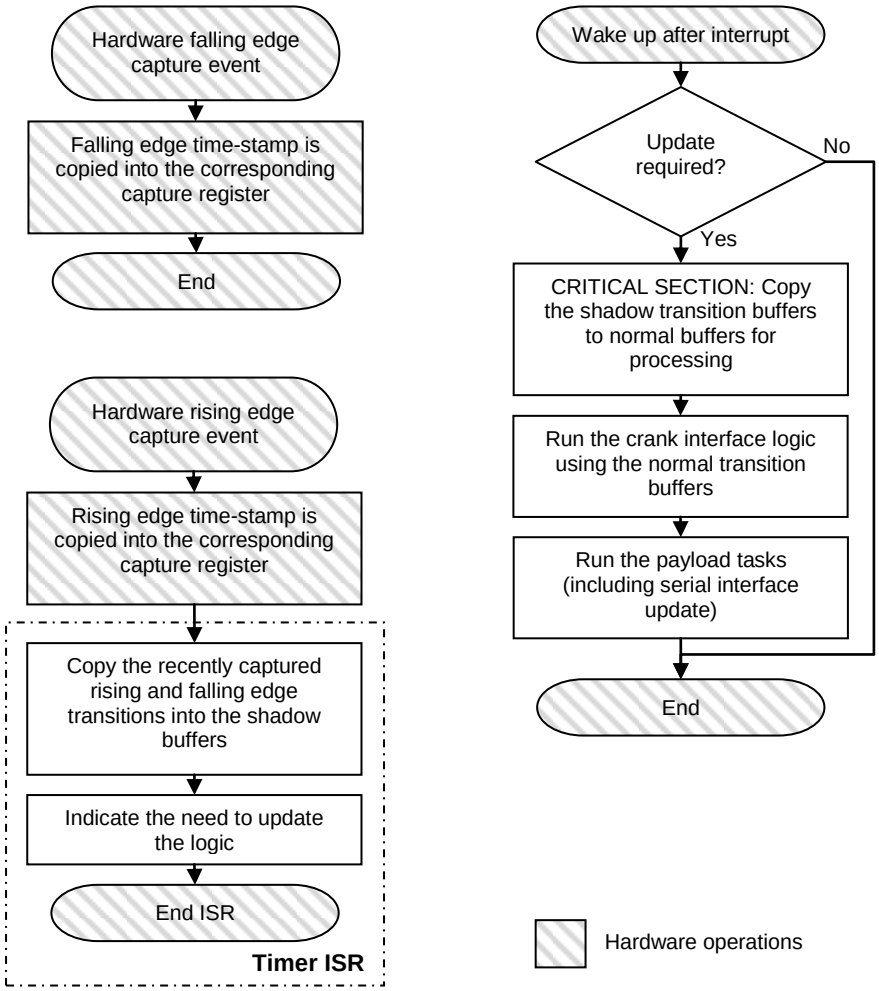


Figure 7-7: Flowcharts for the Event triggered implementation

7.3.2 Static time-triggered implementation

The static time-triggered implementation is used to provide a benchmark of the performance of a typical timer driven polling setup. In both the time-triggered setups, the timer capture is used in conjunction with the onboard DMA to transfer the captured value into buffers in the RAM. It is up to the crank interface task to copy the data from these buffers and process any newly detected transitions.

For the static TT implementation, the crank interface task is run with a period of 1ms and can be expected to process from zero to eight transitions in each execution depending on the speed and position of the engine.

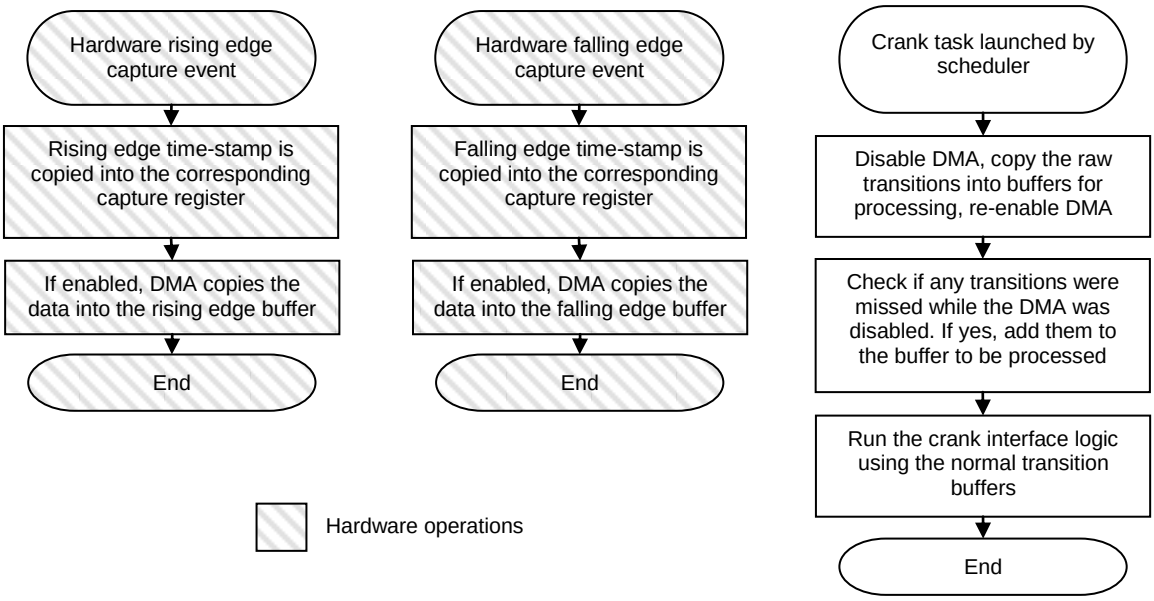


Figure 7-8: Flowchart of the time-triggered implementation of the crank interface

7.3.3 Flexible time-triggered implementation

The flexible time-triggered implementation consists of two major cycles:

1. Default cycle: The default major cycle is similar to the static time-triggered implementation with polling based fixed period control implementation. This mode is used when the engine is not running or when the engine speed is outside the normally expected limits.
2. Variable length cycle: The variable length major cycle is used when the engine is running within normal operating limits. The duration of this major cycle is varied in an attempt to keep it synchronised with a feature of the external signal.

In the case of the crank sensor output, the synchronisation feature can be any pulse out of the 17 pulses in each cycle. Due to the lag introduced by the glitch filter, setting a

system to synchronise with the first pulse of the new cycle will in reality cause it to synchronise at a point after the completion of the second pulse.

Because of the feedback nature of the mechanism for varying the period of the major cycle, it is desired that the crank signal is sampled at a high rate near the time the synchronisation feature is expected to be visible at the output of the glitch filter. At the same time, the DMA transfers allow the crank interface tasks to be run at a rate lower than the rate of arrival of the pulses, freeing up more CPU time for other tasks. To implement these ideas, the variable length major cycle was divided into two parts:

1. Resynchronisation segment: In this segment, only the crank signal is sampled and processed. It consists of four ticks with a tick period of 0.04 of the expected cycle period.
2. Payload segment: this part of the master cycle has five ticks spread evenly over the remaining time in the expected cycle period (tick period of approximately 0.168 of the estimated major cycle period). While the first task in each of these ticks is to sample and process the crank signal, the relatively long tick periods allow the addition of tasks for increased functionality.

The number of ticks and their tick period ratios in the resynchronisation segment are dependent on the following factors:

- Worst case execution time of the tasks to process the crank signal. This should be less than the tick period of the resynchronisation phase when the engine is at its maximum expected speed. With the WCET for the crank sampling and processing with scheduling overheads in the order of 180µs and a tick period of 0.04 of the expected cycle period, this allows the system to operate at engine speeds of up to 6666.66 RPM while ensuring that the sampling task does not

overrun the tick boundaries under the worst execution conditions. It should be noted that the engine has a red line speed of 6250 RPM.

- Maximum rate of change of period expected from cycle to cycle. This dictates the minimum ratio of the cycle period that should be dedicated to the resynchronisation part to capture the worst expected change of rates under normal operating conditions. With a tick period of 0.04 of the cycle period per tick and the estimated synchronisation point, it is able to maintain synchronisation with an error of up to 8 percent between the projected and actual period.

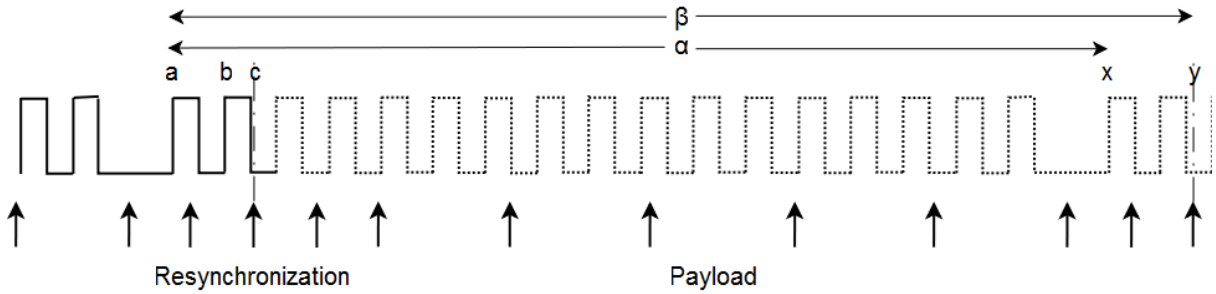


Figure 7-9: Crank signal events and estimation of the next point in time when synchronisation feature will be detected.

Figure 7-9 shows the crank signal, the ticks and various events that are associated with the calculation of the new periods for the next cycle. These events are as follows:

- Event a: start of the first pulse of the cycle
- Event b: point in time after which the start pulse of the cycle will be visible at the output of the glitch filter stage.
- Event c: start of the tick in which the start of the cycle is detected by the logic.
- Event x: estimated point in time at which the next cycle will start.

- Event y: estimated point in time when it would be possible to detect the start of the new cycle.

Calculating the new periods involves:

1. Estimate the period of the new cycle (α) based on the last two cycles (period of the last cycle and the rate of change of periods between the last two cycles).
2. Calculate the new period for the resynchronisation ticks ($0.04 \times \alpha$).
3. Estimate the period from the current start of cycle to the estimated detection possibility of the next cycle ($\beta = 1.1 \times \alpha$).
4. Compensate for the time elapsed since the start of the cycle, the tick in which the start of the cycle is detected and the tick number or the resynchronisation state in which the next cycle should be detected and divide it ($\beta_{remaining}$) over payload ticks.

The flexible time-triggered implementation based on the variable tick TTMPC architecture consists of four phases (see Figure 7-10):

1. Default state: The operations in this state are very similar to the polling implementation of the static TT design.
2. Transition phase: The periods for this phase are equal to the payload phase. It is used only during the transition from the default cycle to the variable length major cycle.
3. Payload segment: This state is designed to allow the user to add additional functionality to the system that is not integral to getting in sync with the crank signal.
4. Resynchronisation segment: This state has short tick periods and runs only tasks which are integral to the crank signal interface and crank synchronisation.

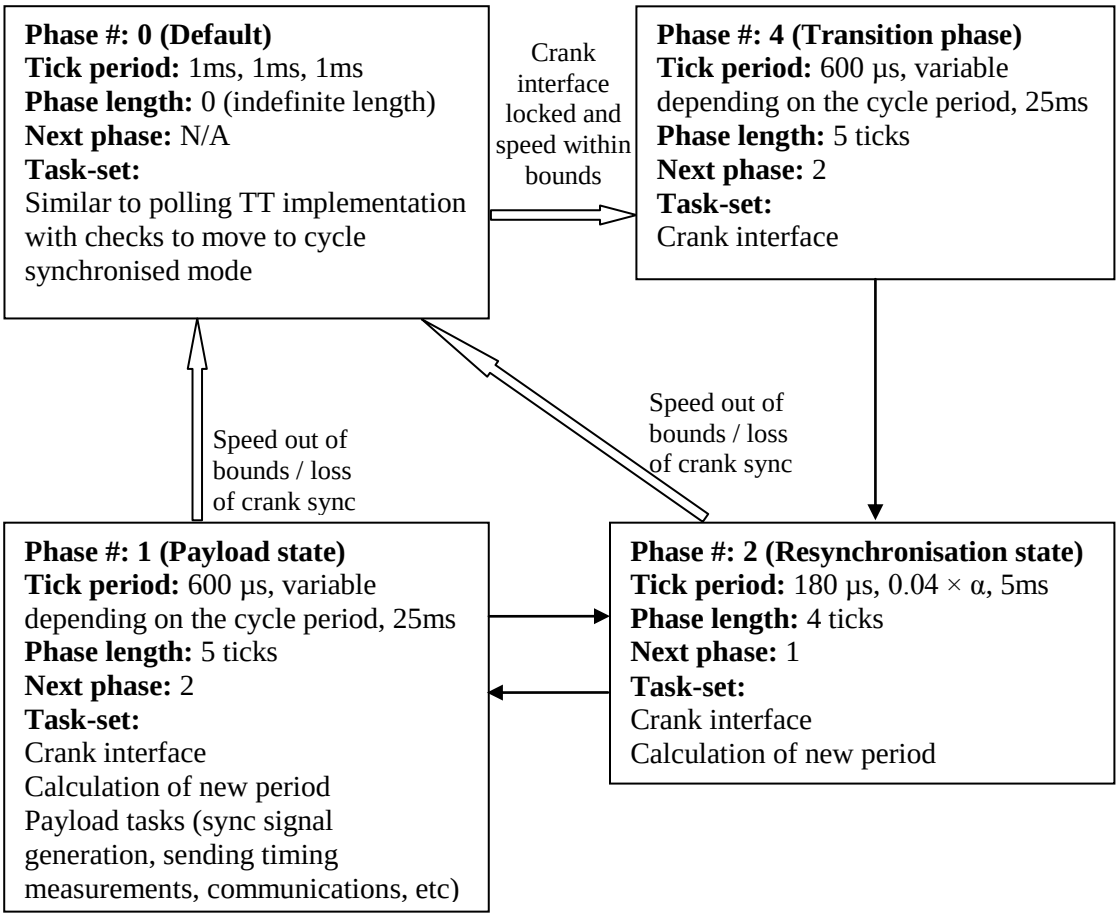


Figure 7-10: Phase diagram for flexible TT implementation

7.4 Test Setup for Synchronisation Performance Comparison

A crank signal generator was implemented to test the synchronisation performance of the three architectures under investigation under similar circumstances. An NXP LPC2129 running at 60MHz is used to simulate the signals expected from the engine. This simulator can be used to output the processed crank signals captured from the engine test-bed or alternately, a simulated signal can be generated to a preset profile.

The signal simulation utilizes a lookup table that contains the ratios of the high and low pulse durations of the actual signal at idling speed and no load. The values in this table are then scaled to generate the signal for any speed.

Table 7-1: Pulse duration ratios used for crank signal simulation.

Pulse #	High duration	Low duration	Pulse #	High duration	Low duration
1	0.0335	0.0297	10	0.0297	0.0260
2	0.0297	0.0297	11	0.0260	0.0297
3	0.0260	0.0297	12	0.0260	0.0297
4	0.0297	0.0297	13	0.0260	0.0260
5	0.0260	0.0297	14	0.0260	0.0297
6	0.0260	0.0297	15	0.0260	0.0260
7	0.0260	0.0297	16	0.0260	0.0297
8	0.0260	0.0297	17	0.0260	0.0781
9	0.0260	0.0260			

For cycles where the initial speed is not the same as the final speed (i.e. there is acceleration or deceleration), intermediate scaling speeds are calculated for each pulse according to the following equation:

$$\boldsymbol{Scaling\_speed_i = \omega_{initial} + \left((i-1) \times \frac{\omega_{final} - \omega_{initial}}{16}\right)} \quad \textbf{(7-1)}$$

Where i is the pulse number for which the scaling speed is being calculated, $\omega_{initial}$ is the initial cycle speed and ω_{final} is the final cycle speed.

In addition to generating the crank signal, the NXP LPC2129 also has an input on which it expects a synchronisation pulse from the system being tested. At the end of each cycle, the crank simulator outputs the period of the next cycle and the point in time at which the pulse was received in the previous cycle (relative to the start of that cycle). This information is sent via an RS232 link at a baud rate of 256 kbps.

To check the dynamic performance of the systems under test, each system generates a sync signal which indicates that the ECU software has reached the point where it will run the control algorithm. The time difference between the start of the cycle and the occurrence of the sync signal are logged by the engine simulator. The ratio between this time to synchronise and the period of the cycle can be plotted to indicate the effectiveness of the setup.

$$\boldsymbol{Sync\ ratio} = \frac{\boldsymbol{\Delta t\ between\ start\ of\ cycle}\ \&\ occurance\ \ of\ \ sync\ honication}{\boldsymbol{Period\ of\ corresponding\ cycle}} \qquad \textbf{(7-2)}$$

For the static time-triggered and event-triggered setups, the sync signal is generated immediately after the start of the new cycle is detected. For the flexible TT, the sync signal is generated when the system is expected to run the control algorithm, i.e. at the start of the first tick of the payload state.

Ideally, due to the crank interface glitch suppression and the interrupt being triggered only on the rising edge, the sync signal in the ET setup should be generated immediately after the end of the second pulse in the crank cycle. The ideal response by the ET implementation for the given test profile can be obtained by the engine simulator logging the start times of the cycle and the third pulse. Using the pulse ratio lookup table, it can be seen that at constant speed, the synchronisation should occur near the 12.27% into the cycle.

The TT implementation's ability to process partially received pulses allows the detection of the first pulse after the falling edge of the second pulse. Assuming no processing overheads and constant speed, this allows for the detection of the new cycle at 9.29% into the cycle. In the case of the static TT, this gives the lower limit on

detection window. The detection window is dependent on the ratio of the tick period and the overall cycle period.

In the case of the flexible TT, the new tick periods are calculated such, that based on the last cycle period and the rate of change of period, the detection of the new cycle should be possible sometime after the middle of the 2nd resynchronisation tick. This gives an offset of approximately 10% period ratio (2.5 × 0.04 cycle period) between the estimated detection of new cycle (from the period $\beta = 1.1 \times \alpha$) and the generation of the synchronisation signal.

Plots of the sync ratio vs speed give an idea of how well an architecture under test is able to remain synchronised with the generated crank signal. As a general rule, a syncrronisation ratio plot that is confined to a narrow range of ratio values over the wide range of test speeds indicates a good synchronisation ability, while a wide range for ratio values could indicate problems with the synchronisation performance.

7.5 Test Case 1: Basic Synchronisation Test

A simple periodic test profile was used to generate a crank signal that comprises of periods at constant speed and transitions of constant rate of change of speed per cycle. This profile was devised to test the ability of the architectures being tested to remain synchronised over the major portion of the engine's operating range. The profile starts with a speed of 600 RPM for a duration of 600 cycles (300 revolutions) to allow the ECU under test to initialize and synchronise with the generated crank signal. After this it generates constant speeds of 500, 1000, 2000, 3000, 4000, 5000 and 6000 RPM for 200 cycles. Between the constant speed periods, the rate of change of speed is 25 RPM per cycle. Figure 7-11 and Figure 7-12 show this test profile with the number of cycles and the time on the x-axis respectively.

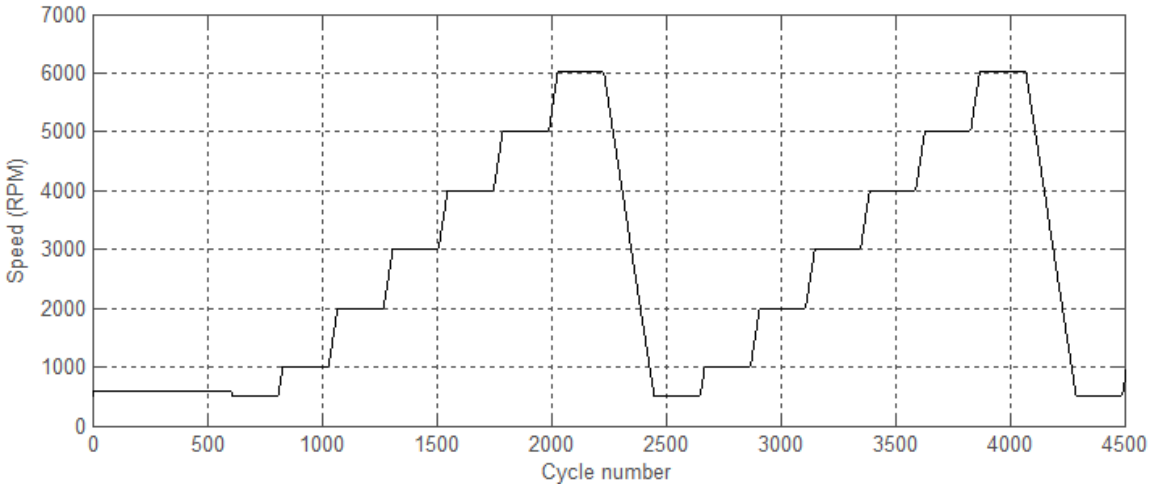


Figure 7-11: Speed vs. cycle number crank simulation test profile used for comparing synchronisation performance.

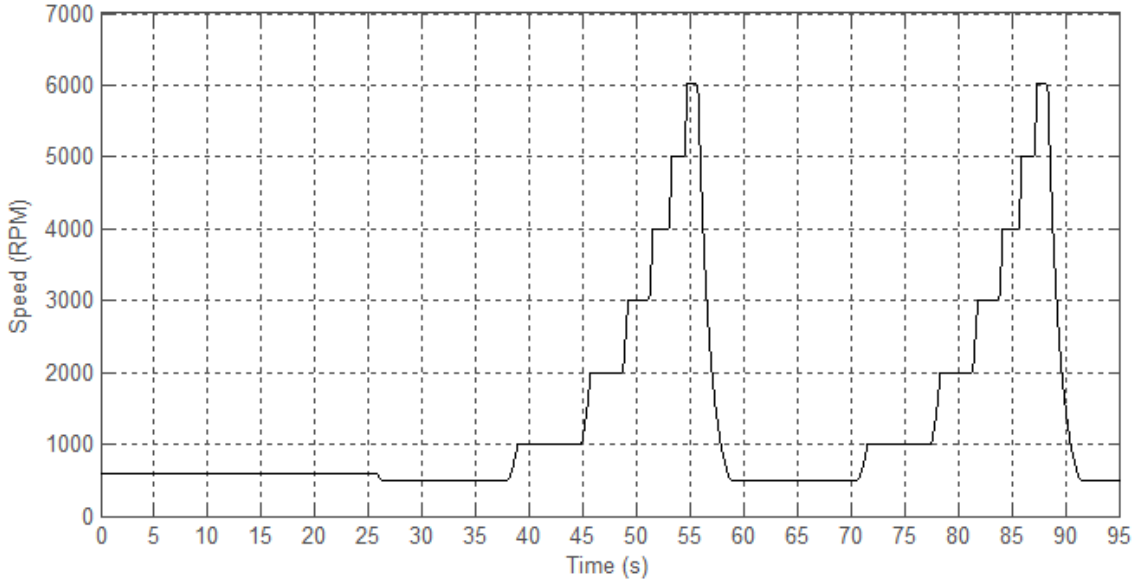


Figure 7-12: Speed vs. time crank simulation test profile used for comparing synchronisation performance.

Figure 7-13, Figure 7-14 and figure 7-15 show the synchronisation performance of the TT static, TT flexible and ET systems respectively for the basic test profile. These plots show the ratio of the synchronisation point to the period of the cycle versus the speed range covered in the test profile (equation 7-2). From figure 7-13, it can be seen that while the static TT allows the earliest sync signal generation in some cycles, the overall jitter in the generation of the sync signal increases with the increase in speed of the

engine. At around 6000 rpm, this jitter ranges from approximately 0.09 to 0.29, giving and overall jitter of about 20% of the cycle period. From these figures, it is evident that the performance of the flexible TT and ET designs is greatly superior to the performance of the static TT design.

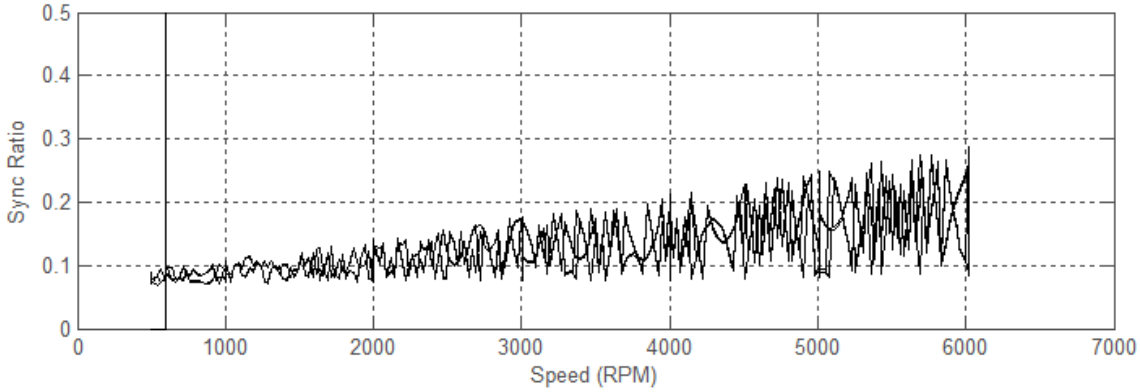


Figure 7-13: Sync performance of static time-triggered system

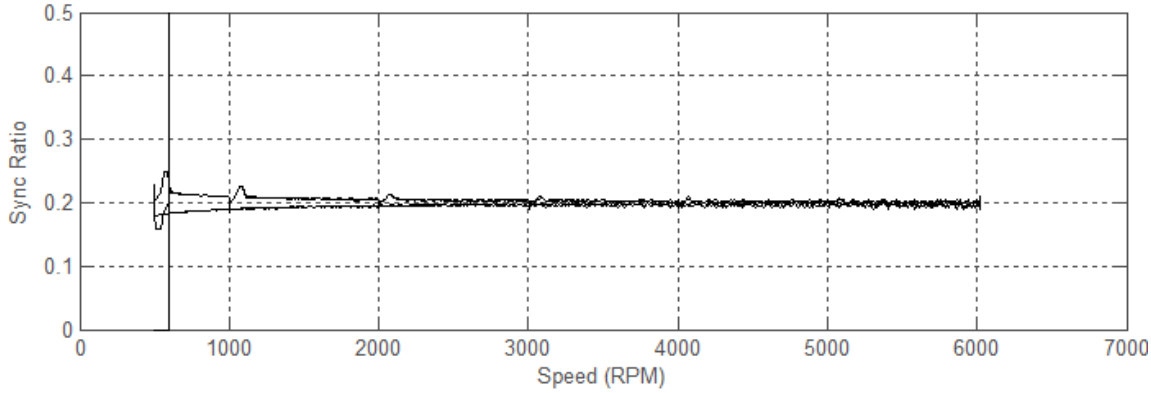


Figure 7-14: Sync performance of flexible time-triggered system

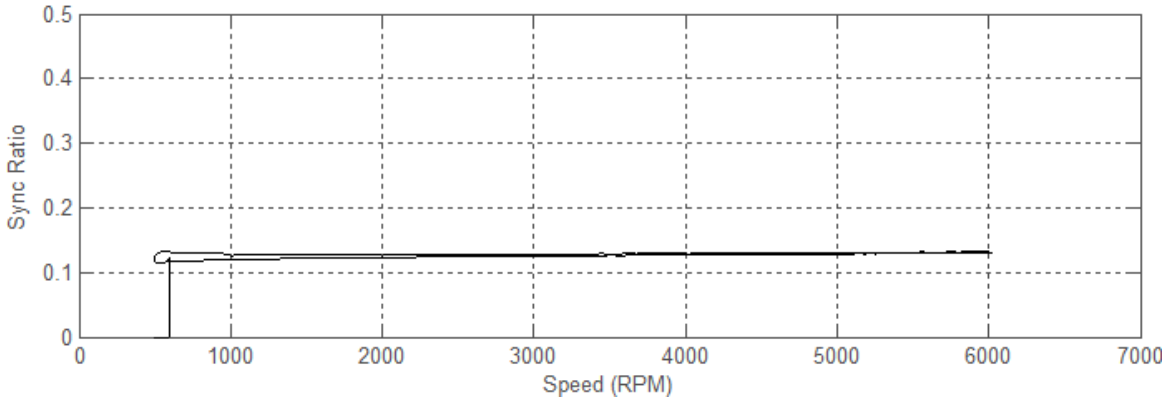


Figure 7-15: Sync performance of event-triggered system

7.5.1 Performance comparison of event-triggered and flexible time-triggered implementations

Figure 7-16 provides a closer comparison between the flexible TT and the ET designs and also provides the ideal case for the ET implementation. Several observations can be made:

- The flexible TT design synchronises at a later point in the cycle as compared to the ET design. This is mainly due to the fundamental differences in the point in the cycle the synchronisation signal is generated in the two designs. It should be noted that the point at which the ET or the flexible TT generate the synchronisation signal can be moved by selecting a different feature / pulse of the crank signal to which these systems synchronise to.
- Increase in jitter in synchronisation of the flexible TT with the increase in the speed of the engine. This is most probably caused by the timer resolution that is used to measure the crank signal period. With a resolution of 10 µs, the quantisation error in the measurement of the period at a speed of 500 RPM results in a reading that could range from 499.917 to 500.083 RPM. At a speed of 6000 RPM, the quantisation error could result in a reading in the range of 5988.024 to 6012.024 RPM. Although the ET design uses the same timer resolution, it is not affected by this as its execution is dictated by the timing of the pulses being generated by the crank signal simulator.

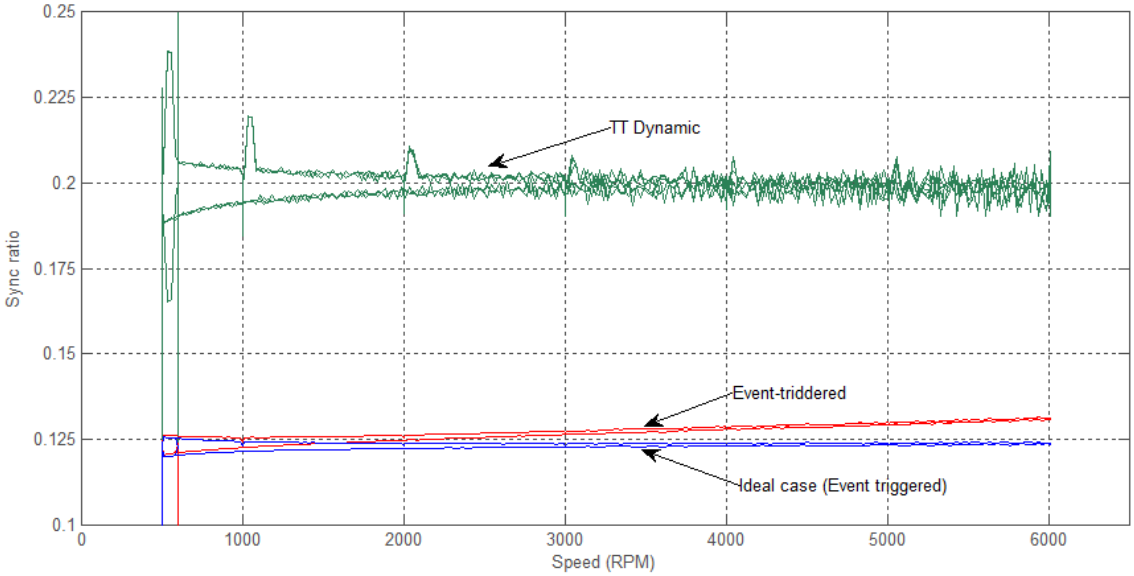


Figure 7-16: Comparison of the synchronisation performance of flexible time-triggered, event-triggered and ideal case.

- Spikes in the synchronisation signal plot of the flexible TT indicate the points when there is a sudden change in speed. The current flexible TT tries to guess the period of the next cycle only on the basis of the past periods that it has observed. When there is a sudden change in the speed of the crank signal, these estimates are thrown off. The spikes are a result of the overshoot because of the prediction mechanism used. In an actual engine, where the speed of the engine is partly affected by the outputs of the controller (e.g. amount of fuel, ignition, etc.); it should be possible to make better predictions of how the future periods will be effected.
- There is a noticeable upward trend in the ET design compared to the ideal case. This is caused by the time it takes to process the new pulses and update the logic. The flexible TT does not exhibit this trend as it compensates for the time it takes to process new pulses.

7.5.2 Implications on code size and CPU usage

In addition to the dynamic performance of the systems, the software code size and CPU usage of the systems was also measured to provide a better comparison of the architectures. Table 7-2 gives a comparison of the code sizes for the designs. The size difference between the ET and static TT is mainly due to the added code complexity for making the system able to log and process multiple pulses per execution. The size difference between the static TT and flexible TT designs is solely because of the scheduler and synchronisation algorithm.

Table 7-2: Code size comparison.

System	Code size	Size relative to ET	Size relative to static TT
Event triggered	20648 bytes	100%	89.23%
TT static	23140 bytes	112.07%	100%
TT flexible	25268 bytes	122.38%	109.2%

Figure 7-17 provides a comparison of the CPU usage per engine cycle for the three systems under test.the CPU usage per cycle for the flexible TT and ET designs remains farily constant regardless of the engine speed. Contrary to this, the CPU usage of the static TT design remains dependent on the ratio between the tick period and the cycle period. At low speeds, the CPU usage of the static TT is fairly higher than the other designs. It has comparable usage to the other systems around 2500 RPM speed, however, at these speeds, its synchronisation performance is significantly worse than both the other designs as evident from the sync performance plots.

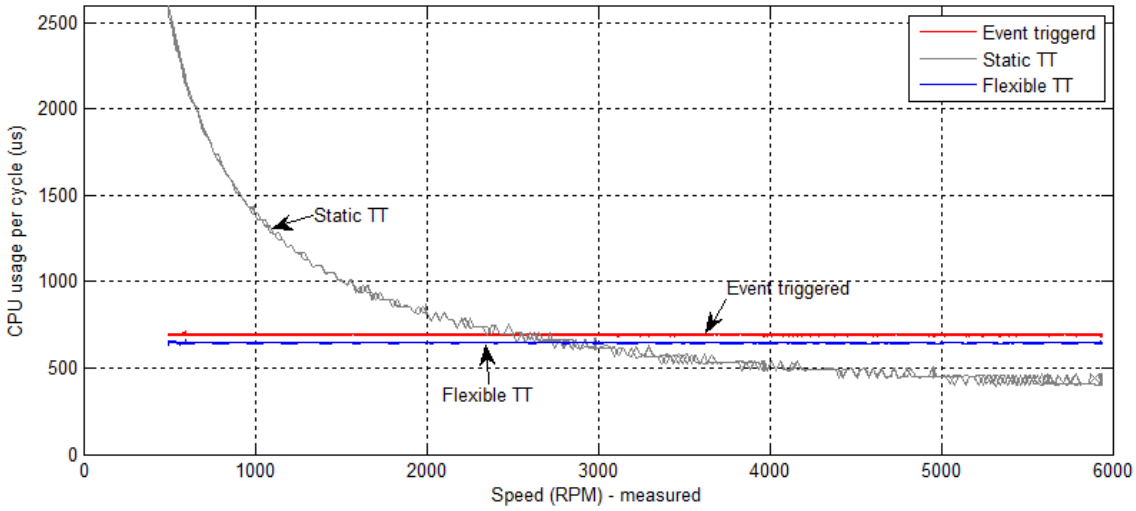


Figure 7-17: CPU usage vs engine speed for the systems under test.

Table 7-3: Comparison of CPU usage per cycle of ET and flexible TT.

	Event-triggered		Flexible time-triggered	
	Overall (µs)	Total ISRs and critical sections (µs)	Overall (µs)	Scheduler overheads (µs)
Min.	681	123	632	83
Max.	711	148	662	98
Avg.	693.56	134.116	646.0836	90.8281
Std. Dev.	3.357	3.7972	2.6174	2.4883

Table 7-3 provides a detailed comparison of the ET and flexible TT designs. It can be seen that despite the schdeuler overheads, the flexible TT has lower overall CPU usage as it can allow for pulses to arrive and be logged by the DMA while processing them in one go. A similar technique can be used in an ET design if there are no glitches in the crank signal. However, the presense of glitches will complicate the ET design's logic as the new cycle cannot be guaranteed to occur after 17 events.

7.6 Effects of Increase in Timing Resolution on Flexible Time-Triggered Performance

Based on the observations on the flexible TT's jitter in the engine synchronisation at high speeds, it was decided to experiment with using a higher timer resolution. The initial decision to use the 10 µs pre-scalar for the timer used to measure the crank signal pulse durations was influenced by the size of the timer (16 bits) and the mechanism that was used to measure the period of the cycle.

- Original method for calculating period of a captured cycle: In order the find the period of a captured cycle, the 16 bit value of the timer at the start of the cycle was subtracted (signed) from the 16 bit value at the end of the cycle. This limited the maximum period that could be measured to a timer count of less than 32767.
- Lowest speeds observed: During engine start, speeds as low as 150 to 200 RPM were encountered. At these speeds, the output of the crank signal was stable enough for the ECU to start fuel injection and spark ignition.

Table 7-4 provides the minimum speed that can be measured by the original method for some timer pre-scalar values. In addition to these, the quantisation error encountered at 2000 and 6500 RPM for these pre-scalars is also included. The original 10 µs pre-scalar was selected as it allowed the detection and correct period measurement of the cycles encountered during engine start up while allowing for the least amount of quantisation error at high speeds.

Table 7-4: Effects of pre-scalar value on the minimum speed measurable and the quantisation errors at 2000 RPM and 6500 RPM.

	Timer pre-scalar			
	1 µs	5 µs	10 µs	20 µs
Minimum speed (RPM)	915.555	457.777	91.555	45.777
Timer count at 2000 RPM cycle	15000	3000	1500	750
Quantisation error at 2000 RPM[1]	0.13 RPM	0.67 RPM	1.33 RPM	2.66 RPM
Timer count at 6500 RPM cycle	4615	923	461	230
Quantisation error at 6500 RPM	1.41 RPM	7.04 RPM	14.08 RPM	28.23 RPM

In order to increase the timer resolution, the following options were possible:

1. Increasing the timer and capture units size: The STM32M103RB controller allows the two or more 16 bit timers to be daisy chained to form higher resolutions (e.g. 32 bits, 48 bits, 64 bits, etc.). While this offers a seemingly easy and straight forward extension with no change in the logical part of the software, the separate DMA handling of the high and low 16-bit capture values for the rising and falling edges of the transitions would have been complex.
2. Alternate mechanism to measure crank cycle period: The crank signal interface task maintains a FIFO of the last 20 filtered pulses that were detected. When an unprocessed pulse is being processed, its period is added to a running total. At the valid detection of the end of cycle / start of new cycle, the running total is noted as the period of the last cycle and is reset. The disadvantage of this

[1] Speed difference corresponding to the count indicated for the speed and the count incremented by 1.

method is the requirement to maintain a running total of the cycle period. This adds an additional step of a 32 bit addition (on a 16-bit architecture) in the processing of the pulses. For a 17 pulse cycle, this translates as seventeen 32-bit additions per cycle compared to the single 16-bit signed subtraction for the original method.

The above mentioned alternate period calculation method was used with a 1 µs timer pre-scalar in the high resolution performance tests. Minimum speed that can be measured by this method is limited by the period of the longest pulse in the cycle. At constant speed, the longest pulse is the 17th pulse with a period approximately 10.41% of the whole cycle (Table 7-1). This give the maximum cycle period bound of 314.764 ms (i.e. 32767 / 10.41%) which translates to a speed of 95.309 RPM.

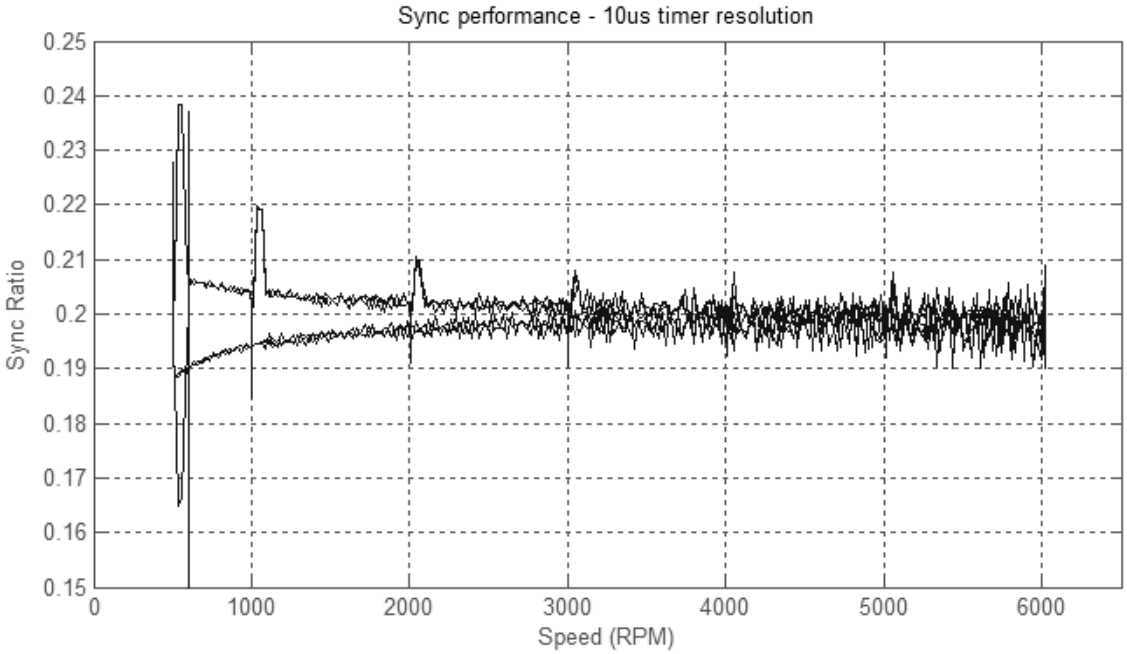


Figure 7-18: Synchronisation performance with a timer resolution of 10 µs

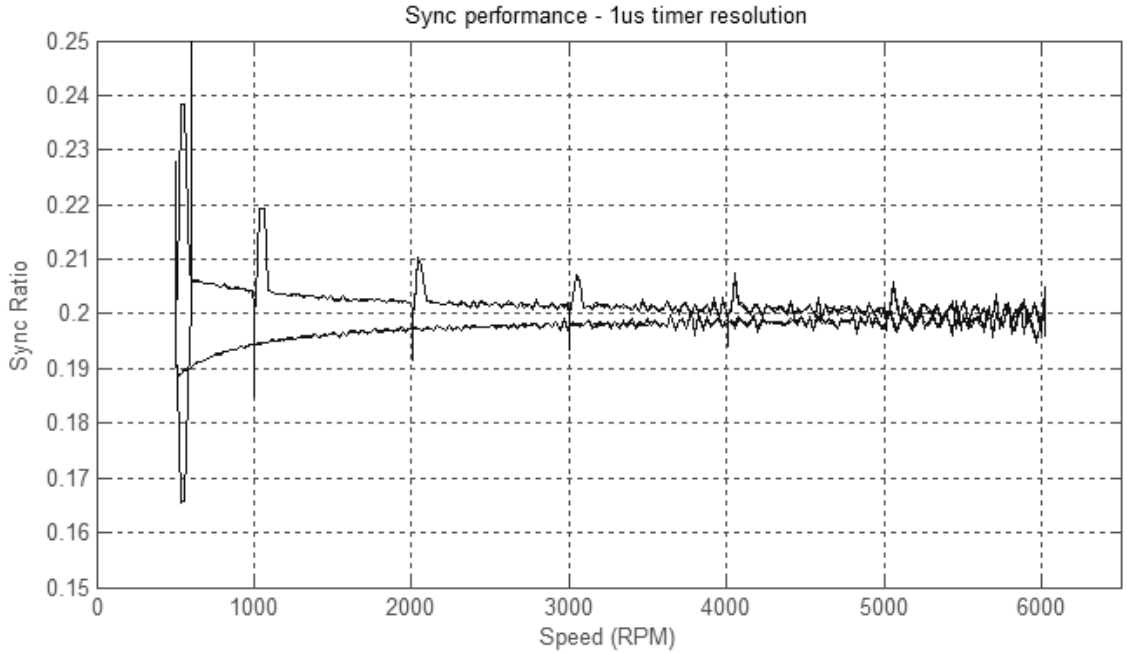


Figure 7-19: Synchronisation performance with a timer resolution of 1 µs

Figure 7-18 and Figure 7-19 show the synchronisation performance with a timer resolution of 10 µs and 1 µs respectively. From these figures, it is evident that there is a significant reduction in the level of jitter at high speeds with the increase in timer resolution. The level of jitter at 6000 RPM has been reduced to one half of that for the original resolution.

Table 7-5: Effect of different period calculation methods on CPU usage.

	10 µs pre-scalar		1 µs pre-scalar	
	Overall execution time (µs)	Scheduler overheads (µs)	Overall execution time (µs)	Scheduler overheads (µs)
Min.	632	83	632	83
Max.	662	98	674	99
Avg.	646.0836	90.8281	647.3241	90.8895
Std. Dev.	2.6174	2.4883	2.9324	2.5735

Table 7-5 shows the comparison of the CPU usage for the two calculation methods. There is a slight increase in the CPU usage for the new method; however, this was to be expected with the increase in complexity of the calculations. In addition to this, the

code size for the new setup has gone up to 25312 bytes from the 25268 bytes for the original method (an insignificant increase of 0.17%).

7.7 Test Case 2: Realistic Driving Cycle

The second test cycle (Figure 7-20 for the test profile) derived from an engine emissions dynamometer test is used to guage how the flexible time-triggered system would perform in a more realistic setting.

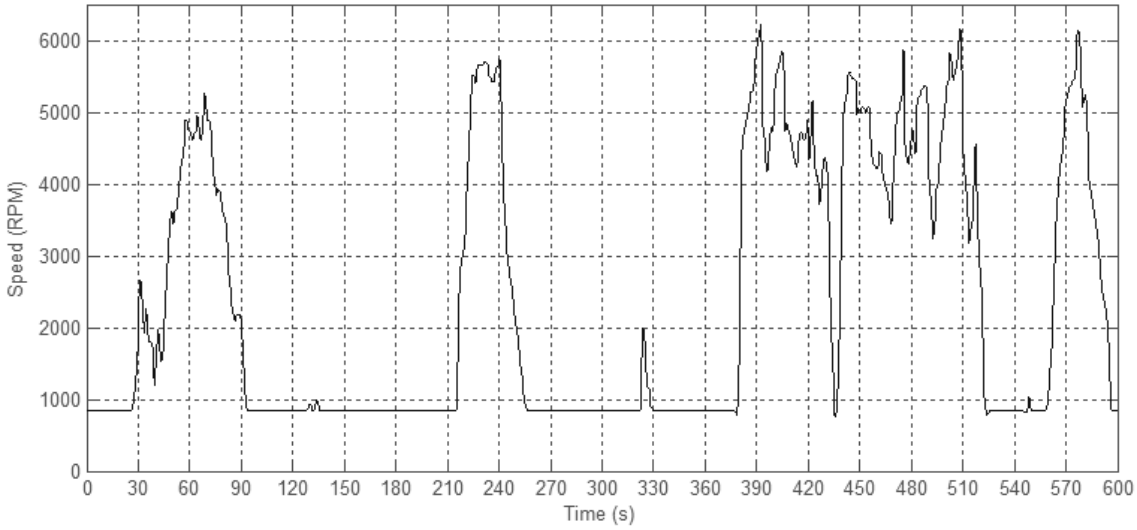


Figure 7-20: Second test profile used to check the synchronisation performance of the various architectures.

The New York Non Freeway (NYNF) and the Los Angeles Non Freeway (LANF) portions of the Federal Test Procedure (FTP) heavy duty transient cycle are used as the starting point for the realistic test cycle. The operations carried out on the original test cycle to make it suitable for the synchronisation testing are:

i. The original cycle was scaled such that the idling speed was 850 RPM and the maximum speed achieved by the engine was 6250 RPM.

ii. The discontinuous step commands of the scaled cycle were smoothed to obtain continuously varying speed profile. Figure 7-21 shows the results of the smoothing process on a portion of the cycle.

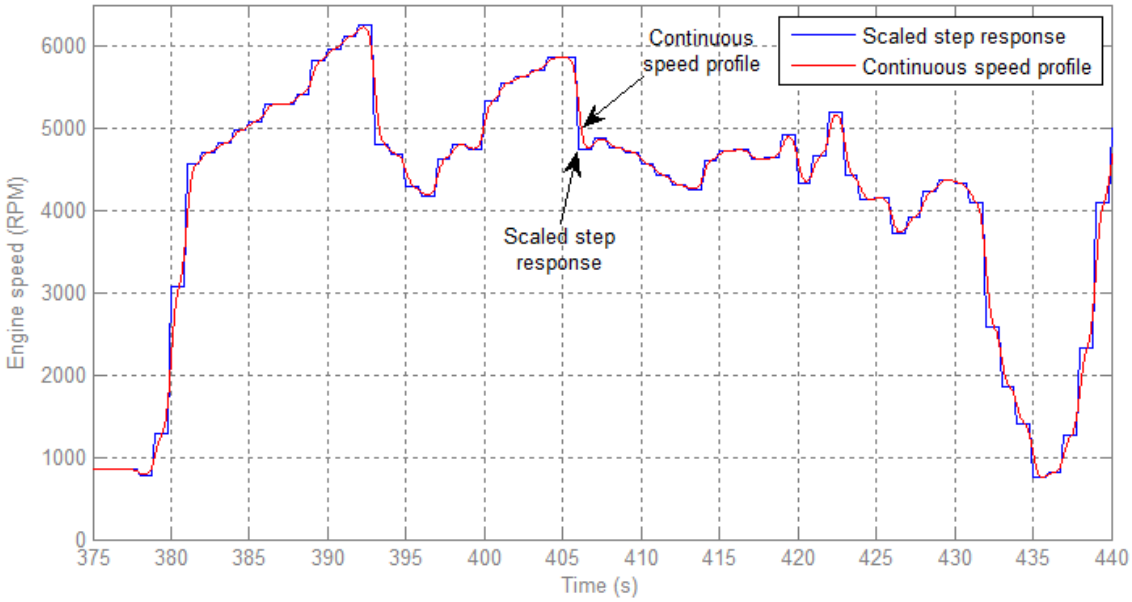


Figure 7-21: The scaled step commands and the output of the smoothing process show from time 375 to 440 seconds of the test profile.

iii. From this continuous profile, initial and final speeds for each cycle of the simulated crank signal were calculated to make the generated crank signal mimic the smoothed profile.

A lookup table with 47805 cycle speed entries is used by the crank signal generator to generate the test crank signal that follows the speed profile. In addition to this, other changes were made between this and the previous test setup. These included increased timing resolution in the crank signal generator from 1µs to 0.1µs to reduce quantisation errors in the generated speed near the top end of the speed spectrum. Also, based on the observations in section 7.6, a 1µs timer resolution is used in the architectures being tested.

Figure 7-22, Figure 7-23 and Figure 7-24 show the synchronisation performance of the static time-triggered, flexible time-triggered and event-triggered architectures over the realistic drive cycle. Similar to the observation made in the basic synchronisation test in section 7.5, the performance of the static time-triggered architecture is significantly inferior to the performance of the other two architectures being tested due to the large amounts of jitter in the generation of its synchronisation signal.

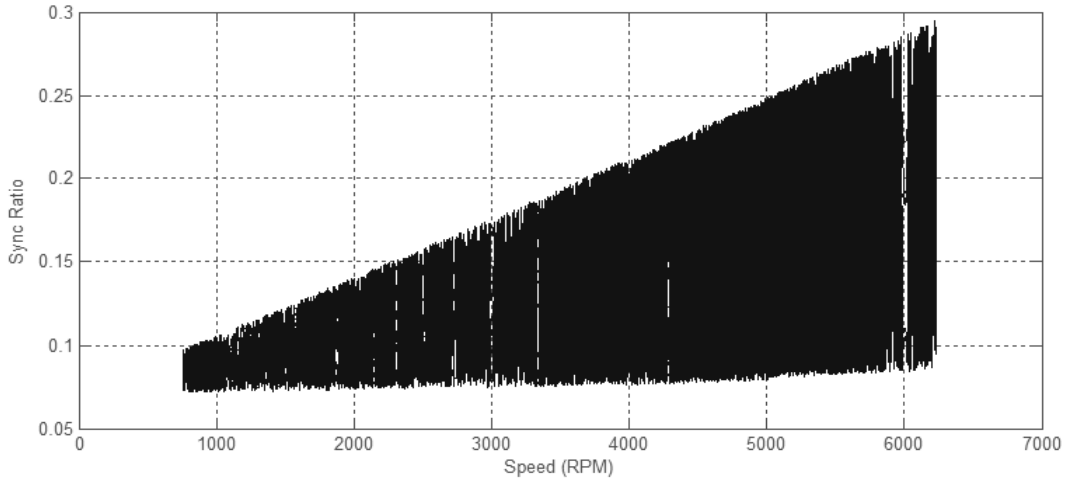


Figure 7-22: Synchronisation performance of static time-triggered architecture on the realistic drive cycle.

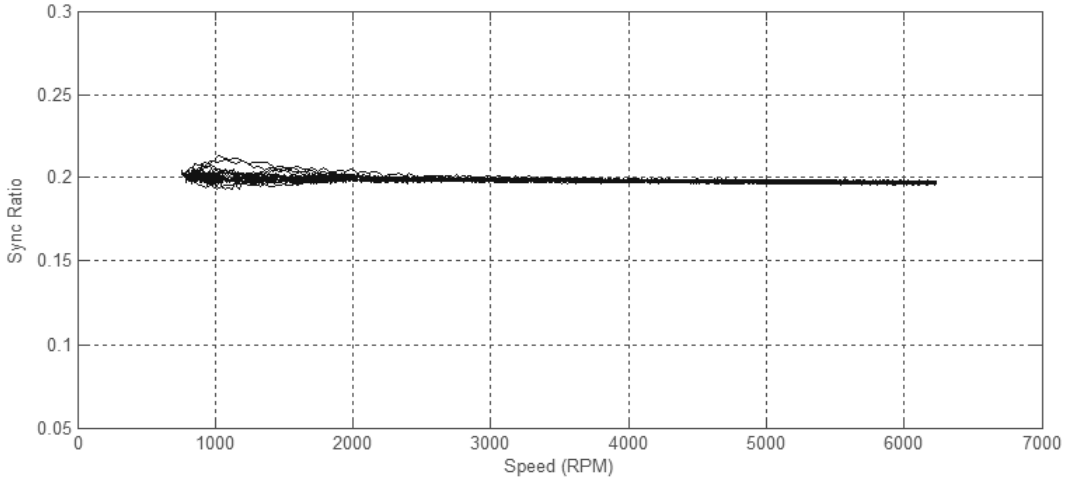


Figure 7-23: Synchronisation performance of flexible time-triggered architecture on the realistic drive cycle.

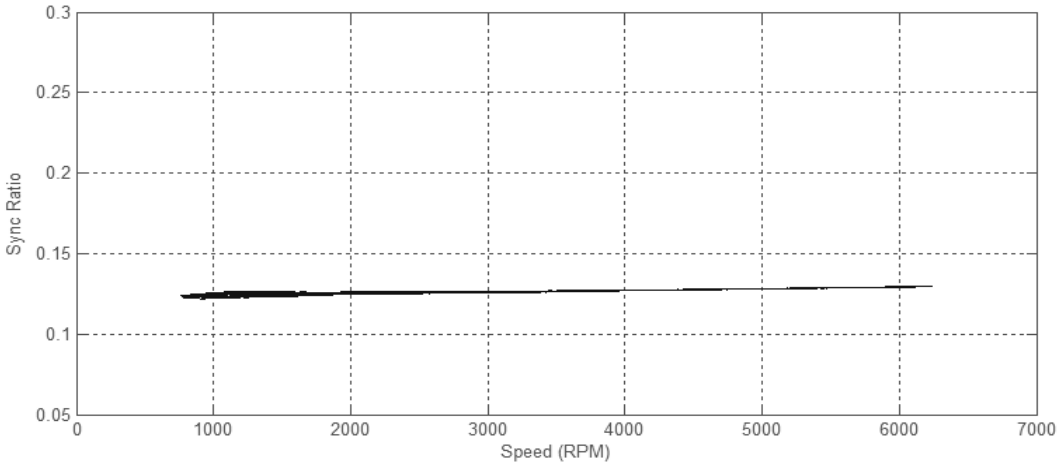


Figure 7-24: Synchronisation performance of event-triggered architecture on the realistic drive cycle.

7.7.1 Performance comparison of event-triggered and flexible time-triggered implementations

Figure 7-25 shows the side-by-side synchronisation performance of the flexible time-triggered, the observed event-triggered and the ideal case event triggered. Some of the observations made in the previous test case (in section 7.5) are still valid, while others are greatly reduced due to increased timing resolutions both in crank signal generation and detection.

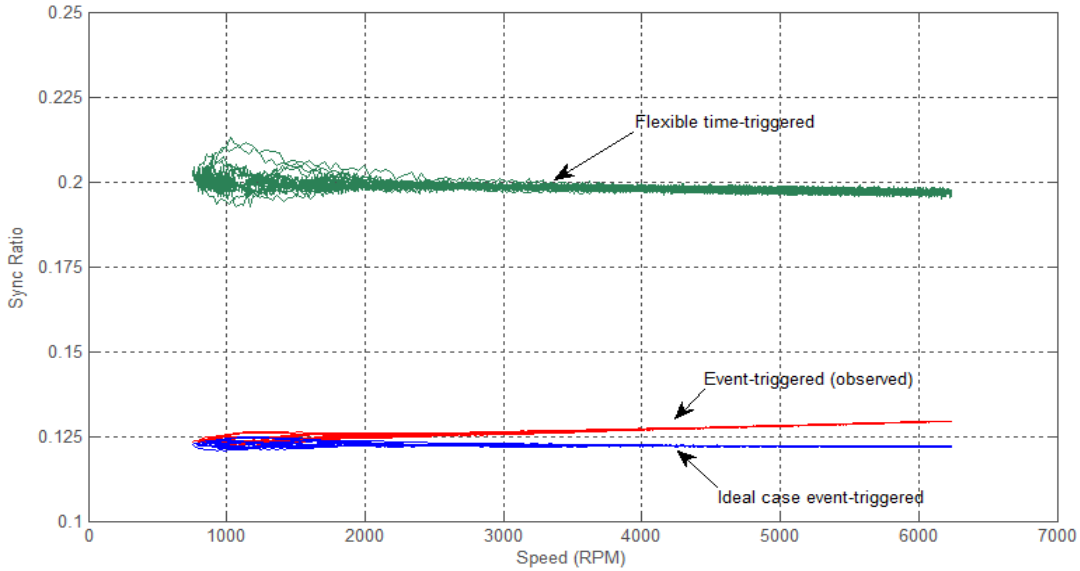


Figure 7-25: Side-by-side synchronisation performance comparison of TT flexible, event-triggered and ideal case.

Like the previous test case, the flexible TT design synchronises at a later point in the cycle as compared to the ET design. This is mainly due to the synchronisation signal being generated at the moment the start of the cycle is detected in the ET and at a fixed point in the staticly scheduled major cycle of the flexible TT design. Also, the noticeable upward trend in the ET design compared to the ideal case is again evident.

Unlike the previous test case in section 7.5, there is no noticeable increase in jitter in the synchronisation of the flexible TT with the increase in the speed of the engine. This is helped by the increase in timing resolution of the crank signal capture as indicated in section 7.6 and the increase in timing resolution of the crank signal generator.

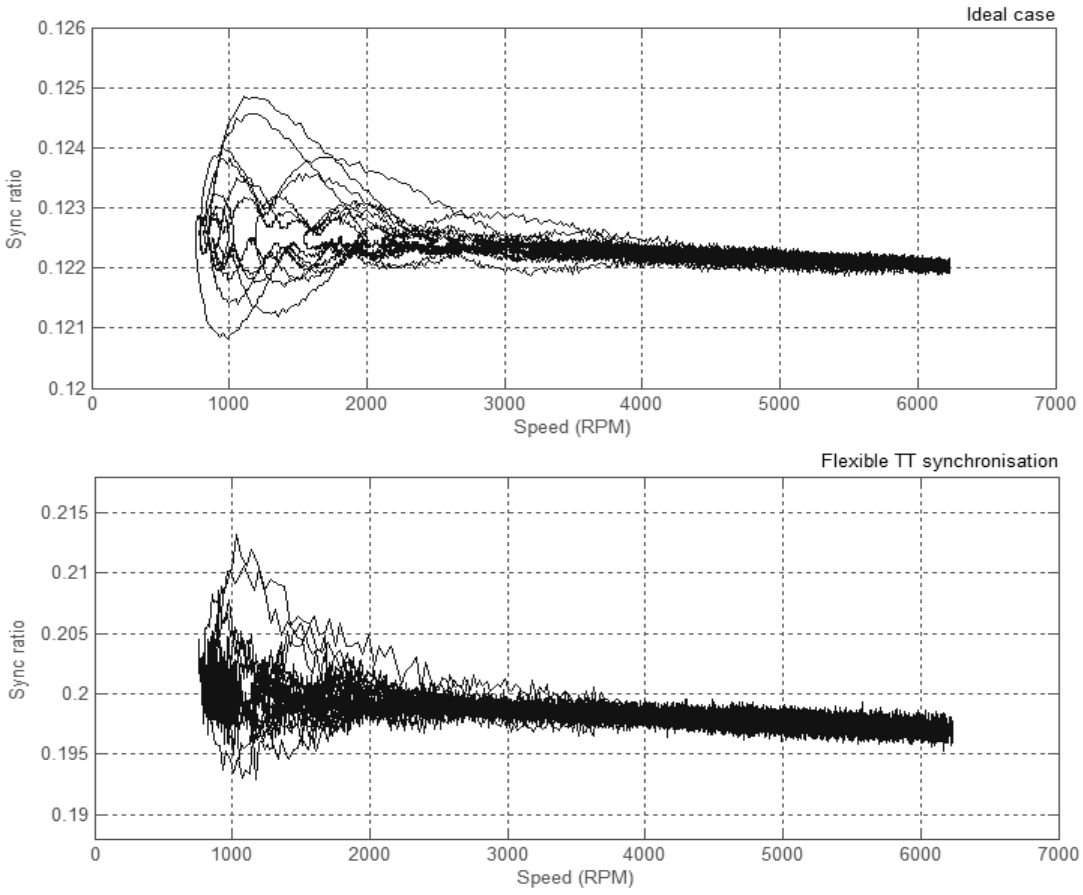


Figure 7-26: Close up view highlighting the similarity of the contours of the ideal case event-triggered (top plot) and the flexible TT (bottom plot)

In Figure 7-26 it can be seen that there are no pronounced spikes in the flexible TT‘s synchronisation plot similar to those encountered in the previous case study in section 7.5. To examin this phenomenon in detail, two metrics for the test profile mut be defined:

i. First order speed difference: This metric provides informantion on how rapidly the speed changes from one cycle to the next. It is given by equation 7-3:

$$\Delta\boldsymbol{\omega_i} = \boldsymbol{\omega_{Fi}} - \boldsymbol{\omega_{Ii}} = \boldsymbol{\omega_{Fi}} - \boldsymbol{\omega_{F(i-1)}} \quad \textbf{(7-3)}$$

Where $\Delta\omega_i$ is the speed difference for cycle i, ω_{Fi} is the final speed for cycle i, ω_{Ii} is the initial speed for cycle i and $\omega_{F(i-1)}$ is the final speed for cycle i-1.

ii. Second order speed difference: This metric provides a correlation between the speed changes in the previous and current cycle, and is given by equation 7-4:

$$\Delta^2\boldsymbol{\omega_i} = \Delta\boldsymbol{\omega_i} - \Delta\boldsymbol{\omega_{i-1}} = (\boldsymbol{\omega_{Fi}} - \boldsymbol{\omega_{Ii}}) - \left(\boldsymbol{\omega_{F(i-1)}} - \boldsymbol{\omega_{I(i-1)}}\right) \quad \textbf{(7-4)}$$

Where $\Delta\omega_i$ is the speed difference for cycle i, $\Delta\omega_{(i-1)}$ is the speed difference for cycle i-1, ω_{Fi} is the final speed for cycle i, ω_{Ii} is the initial speed for cycle i, $\omega_{F(i-1)}$ is the final speed for cycle i-1 and $\omega_{I(i-1)}$ is the initial speed for cycle i-1.

Figure 7-27 show plots of the first order speed difference for the realistic (top) and basic test profiles (bottom) respectively. From these, it can be seen that the maximum value for the realistic test profile is close to 50 RPM difference between the initial and final speeds of a cycle (encountered at cycle number 11275), but does not exceed ±25 RPM for the the basic test profile.

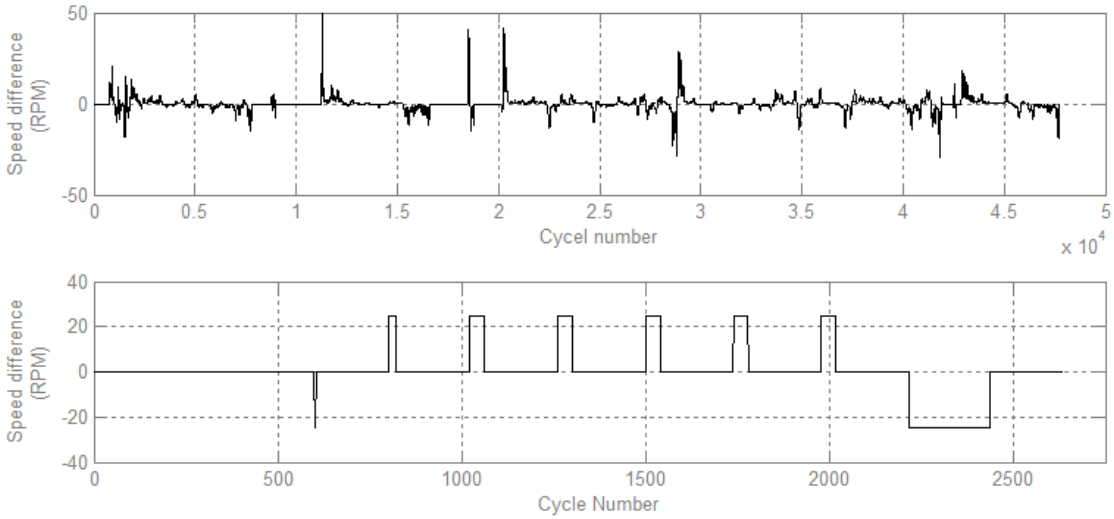


Figure 7-27: First order speed difference for the realistic drive cycle (top) and basic test profile (bottom).

Figure 7-28 shows the plots of the second order speed difference for the realistic (top) and basic test profiles (bottom) respectively. From these, it can be seen that the maxium values for the realistic cycle fall within ±5 RPM but remains ±25 RPM for the basic test cycle. The reason why seond order speed difference is a better indicator of synchronisation performance for a particular test input lies in the current mechanism that is used to estimate the expected period of the next cycle (section 7.3.3).

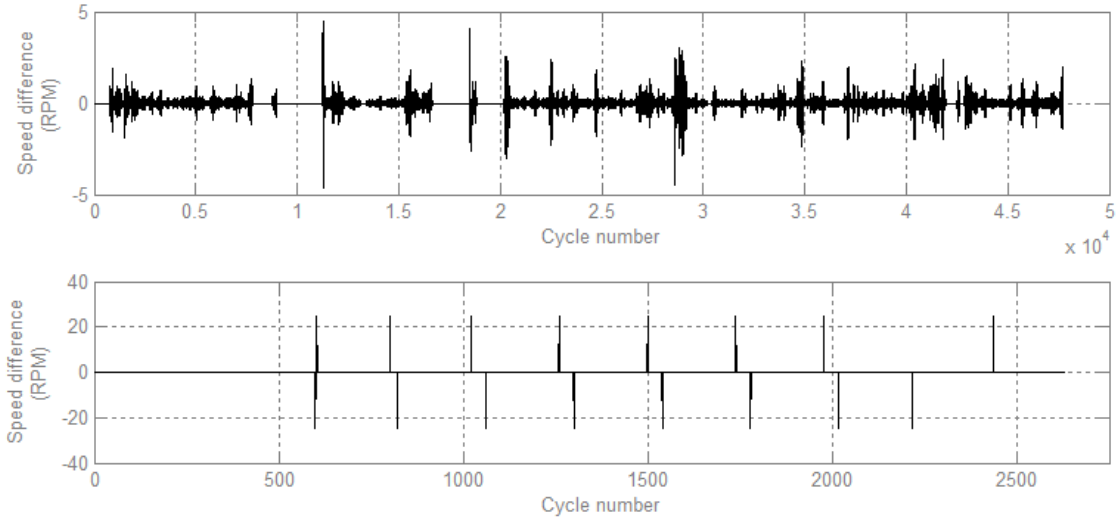


Figure 7-28: Second order speed difference for the realistic drive cycle (top) and basic test profile (bottom).

7.8 Conclusion

The engine synchronisation case study presented in this chapter provides a challenging problem for time-triggered systems. The literature review on the subject failed to provide any references for purely time-triggered architecture being used for engine management of the reciprocating engines.

The work culminated in the implementation of the following solutions:

1. Static time-triggered synchronisation platform that relied on polling the crank signal at a fixed period.
2. Flexible time-triggered synchronisation platform that constantly changed its cycle periods to remain synchronised with the crank signal.
3. An event-triggered platform that relied on events generated by the crank signal to maintain synchronisation.

The two test cases used to test the performance of these architectures managed to highlight the limitations of the static TT for this application and shows why, in the absence of flexible TT, such applications can be considered to be only in the domain of ET architectures. The novel flexible TT approach manages to provide a platform that is capable of remaining synchronised with the internal orientation of the engine's crank shaft, allowing tasks to run at specific points in the cycle.

This flexible TT implementation could form the foundation for an internal combustion engine control setup. Such a setup should have a more predictable nature than event-triggered architectures.

8 Adding Flexible Pre-Emption

This chapter looks at how flexible pre-emption can be added to the multiphase cooperative scheduler. It discusses the behaviour of the Time-Triggered Hybrid (TTH) scheduler and why its extension is not a very flexible alternative. Finally, it presents and discusses the flexible dual scheduler architecture.

8.1 Classifying the Time-Triggered Hybrid Architecture

The Time-Triggered Hybrid (TTH) acts as a fixed priority semi pre-emptive scheduler with two priority groups. While the tasks run cooperatively within these groups, tasks from the higher priority group can pre-empt tasks from the lower priority group. Figure 8-1 shows the implied task priorities and their ability to pre-empt other tasks for a hybrid system with ***m*** pre-emptive tasks and ***n*** cooperative tasks.

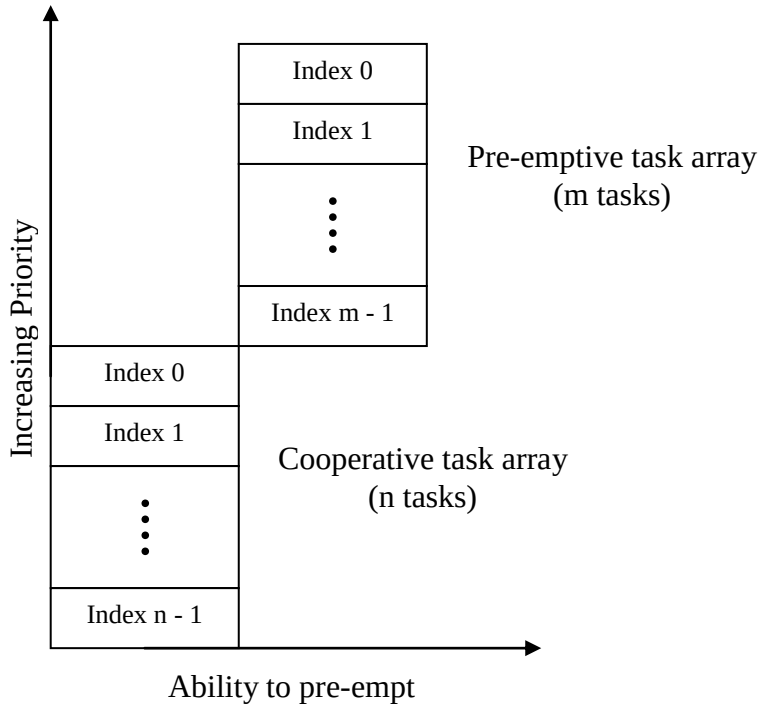


Figure 8-1: Implied fixed priorities in the Time-Triggered Hybrid (TTH) scheduler

While the Time-Triggered Cooperative (TTC) can still be considered a static scheduler as the task execution order is not affected by variations in task timings, the same cannot be said about the TTH. The pre-emptive nature of some tasks along with variations in the timings of the cooperative tasks could mean that the actual number of paths through the program could be infinite. That said, the combination of single path execution or code balancing techniques (described in section 2.2.3) with the advance knowledge of when the next interrupt will occur can be used to limit the number of paths through the program.

8.2 Adding Flexible Limited Pre-Emption

At times, a system needs to run a task periodically with a short period/deadline along with one or more task(s) whose worst case execution time (WCET) is more than this value. As mentioned in section 2.4, this is not possible in a cooperative environment without breaking up the long task. Examples of the tasks that could require short periods include sampling an input signal (e.g. for sampling an audio signal), generating an output pattern in accordance with earlier control computations (e.g. variable frequency sine wave generation using look up tables) or responding to an external event very quickly (e.g. over-current detection to protect a critical or expensive component).

In order to make a highly predictable scheduler architecture that meets the needs of generic applications, some key assumptions have to be made with regards to the potential applications:

1. Most of the processing should be done in cooperative tasks to keep the pre-empting tasks as short as possible.
2. The pre-empting task(s) has to be run periodically regardless of the phase / state, the rest of the system is in.

3. There should be an option to vary the period of the pre-empting task at run time.

8.3 Single Scheduler Architecture with Limited Pre-Emption

Similar to the concept of a hybrid version of the TTC (i.e. TTH) that allows a single frequent task to pre-empt the other (cooperative) tasks, a hybrid version of Time-Triggered Multiphase Cooperative (TTMPC) seems like a good starting point. The Time-Triggered Multiphase cooperative with Variable Rate pre-emption Hybrid (TTMPVRH) scheduler is based on first version of the TTMPC scheduler (support for only finite duration phases, no transient overloads across phase boundaries and fixed tick periods for the phases).

8.3.1 Timing relationships between cooperative and pre-emptive tasks

In order to limit the code complexity caused by the introduction of a single pre-emptive task, some limitations had to be enforced on the timing relationships between the various periods in the system. A new time unit (base time period) was defined such that:

$$base\ time\ \ period = HIGHEST\ COMMON\ FACTOR\left(tick\ \ durations\ of\ \ all\ \ phases\right)$$

Alternately, if an arbitrary base tick period is selected, it limits the cooperative tick periods to values given by the following equation:

$$possible\ cooperative\ tick\ \ period = base\ time\ \ period * n \qquad \left(where\ n = 1, 2, 3, ...\right)$$

The relationship between the base tick period and the pre-emptive task's period is given by the following equation:

$$preemptive\ tick\ \ period = \frac{base\ time\ \ period}{n} \qquad \left(where\ n = 1, 2, 3, ..., 255\right)$$

The pre-emptive task's period (i.e. timer interrupt period) can be changed at run time by calling a scheduler function with the desired configuration parameters.

8.3.2 Code complexity of the flexible time-triggered hybrid scheduler

A significant amount of code complexity is added to the TTMPVRH to support the variable rate hybrid task. Table 8-1 gives the code complexity of the two schedulers.

Table 8-1: Code complexity comparison for TTMPC v1.0 and TTMPVRH v1.0.

	TTMPC v1.0		**TTMPVRH v1.0**	
	Sch_Update (ISR)	***Complete scheduler***	***Sch_Update (ISR)***	***Complete scheduler***
Cyclomatic complexity (McCabe 1976)	4	28	7	33
Lines of code (LOC) - total[2]	27	197	38	230
LOC – execution	17	101	22	117
LOC – declaration	2	31	2	38

It can be seen from Table 8-1 that the complexity of the scheduler timer ISR is increased significantly (40% increase in total LOC, 75% increase in cyclomatic complexity of the scheduler ISR). This problem is compounded if the system is adapted to accept multiple pre-emptive tasks.

Also, because the pre-emptive task is run from within the scheduler update, a catastrophic failure in this task will result in the failure of the whole system.

8.3.3 Pros and cons of a single scheduler architecture

Advantages of such a design include:

- Effectively an extension of the existing TTH architecture, allowing adapting the techniques and tools for the TTH to the TTMPVRH.
- The use of a single time unit (GCD of all periods involved) for the cooperative and pre-empting tasks ensure that there is a high level of synchronisation between all tasks.

[2] The total LOC includes white space lines in the code. These lines are not counted in the other two LOC measures.

- It can be extended to allow for pre-empting tasks that are specific to individual phases.

The limitations of such a design include:

- Restrictions are imposed on the task periods and the tick periods of the different phases. Inappropriate selection of tick periods in different phases can result in excessive overheads (e.g. a two phase system with one phase having a tick period of 1.2 ms and the second phase tick period of 2 ms will require periodic interrupt generation ever 0.2 ms).
- Such a design cannot be used in implementations where the tick periods have to be varied arbitrarily at run time to synchronise the major cycle with an external system.
- Greatly increased code complexity of the scheduler ISR. This value will be further increased if support for multiple pre-empting tasks is required.

8.4 Dual Scheduler Architecture

In order to limit the complexity while increasing the flexibility, a dual scheduler architecture was also evaluated. The resulting system is in effect two disjoint schedulers running on the same machine (Figure 8-2). In this architecture, each scheduler has its own timer and ISR.

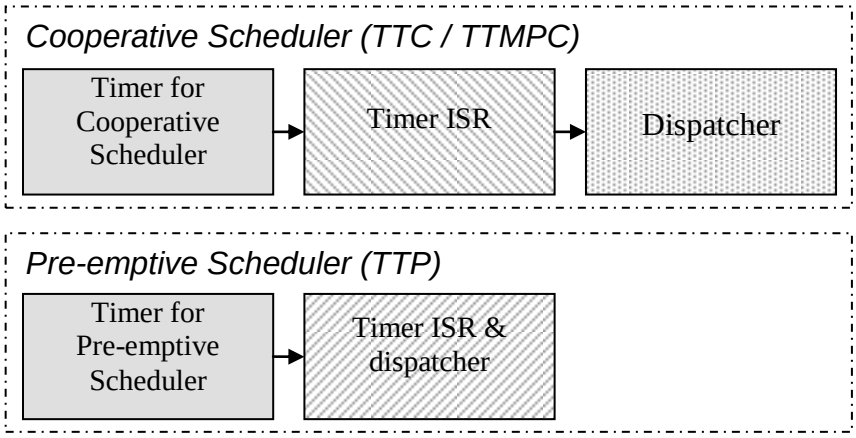


Figure 8-2: Overview of the TTxC + TTP architecture.

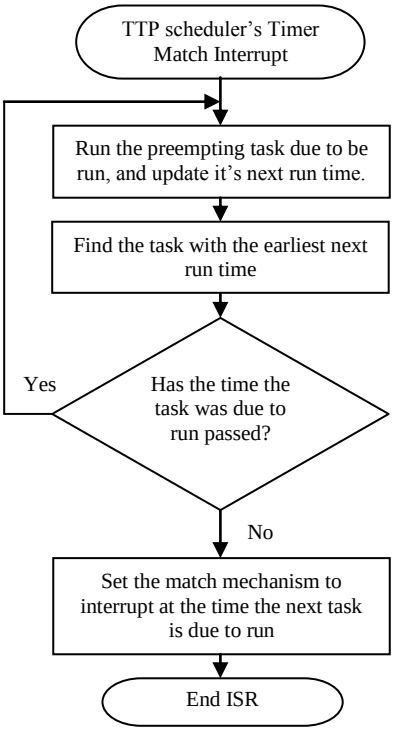
The cooperative scheduler can either be a regular time-triggered cooperative (TTC) scheduler described in section 3.2.2 or the Time-Triggered Multiphase Cooperative (TTMPC) scheduler in chapter 6.

8.4.1 The pre-emptive scheduler

The time-triggered pre-emptive (TTP) scheduler is to be used in parallel with a cooperative scheduler. The key points of the TTP's design are as follows:

- The scheduler uses a free running timer (incrementing from zero to the maximum value and then overflowing back to zero) and uses the timer ISR to pre-empt any cooperative task and run the pre-empting tasks.
- The scheduler keeps track of the time (timer value) at which all the tasks are due to run next, and uses the timer match interrupt to launch the task that is supposed to run next (similar to an EDF scheduler).
- Because of the way the scheduler is designed and implemented, the pre-empting tasks do not require time periods that are multiples of a base time unit. Also, offsets can be used to minimize jitter in multiple pre-empting tasks with the same period.

- Despite the name, all the pre-emptive tasks run cooperatively. If more than one pre-empting tasks are due to run at the same time, an implied static priority is used to determine the order of execution.



Flowchart 8-1: TTP Scheduler's update and dispatch mechanism.

- With provisions to change the periods of existing tasks, deleting them or adding new tasks in the pre-emptive scheduler while the system is running, flexible scheduling can be achieved to quickly respond to the changes. However, this is achieved at the cost of reduced predictablity.

8.4.2 Interactions between the cooperative and pre-emptive schedulers

The two schedulers run concurrently on the target platform. The key points of this hybrid architecture are:

- In the system, only the timer interrupts for the two schedulers are enabled. All other events have to be polled for in user tasks.

- The cooperative and pre-emptive schedulers use two separate timers to allow greater flexibility in the selection of the time periods of the cooperative and pre-empting tasks (Figure 8-2).
- The pre-empting task is launched from the TTP scheduler's timer ISR and can pre-empt any currently running cooperative task (Figure 8-3).
- The cooperative scheduler's timer ISR can pre-empt TTP's timer ISR as well as any pre-empting task (refer to Figure 8-3 for details). This can ease the implementation of a suitable task guardian.

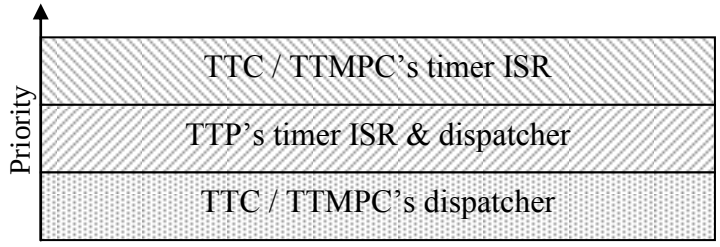


Figure 8-3: Priority levels in the dual scheduler architecture.

- Because of the high level of temporal flexibility offered by the architecture, the responsibility of using appropriate resource sharing techniques is left to the user. This is done so that applications that have limited or no resource sharing between cooperative and pre-emptive tasks do not suffer the associated overheads. However, applications that require resource sharing will be more complicated from the user's point of view.

8.4.3 Alternate configuration for the dual scheduler architecture

An alternate configuration is also possible for the dual scheduler architecture. Instead of using two timers, the Scheduler Update (the timer ISR) for the cooperative scheduler can be scheduled to run as another pre-empting task (see Figure 8-4 and Figure 8-5 for details).

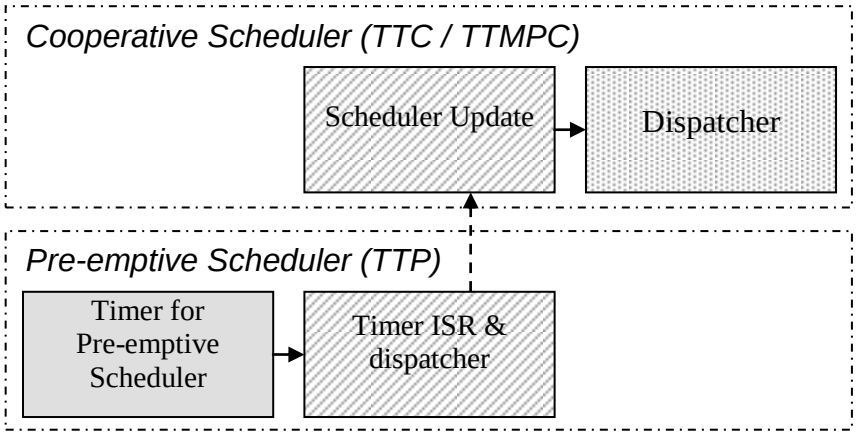


Figure 8-4: Overview of the alternate configuration of the TTxC + TTP architecture.

The pros and cons of this configuration compared to the original configuration are as follows:

- Only one timer is required for the two schedulers, freeing up a valuable peripheral in the system.
- Higher level of synchronisation between the cooperative and pre-empting tasks is possible with fewer overheads as the same time source is used. This is more challenging in the original configuration as there is a slight offset in the start times of the two timers at system initialisation.
- Some modification will be required in the cooperative scheduler to operate in the alternate configuration. The main changes will be in parts of the scheduler code that setup the timer and vary the tick periods.
- As the pre-empting tasks are run from the only ISR in the system, the implementation of the task guardian is not straight forward.

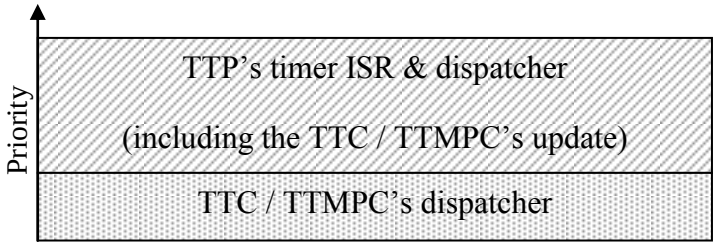


Figure 8-5: Priority levels in the alternate configuration of the dual scheduler architecture.

8.5 Operating Configurations of the Dual Scheduler Architecture

The possible operating configurations of the TTP scheduler architecture are:

1. Fixed Period Hybrid (FPH): All the pre-empting task periods remain fixed at run time.
2. Variable Period Hybrid (VPH): The period of at least one pre-empting task can change at run time.

The overall operating configuration of the combined system is dependent on both the configuration of the pre-empting scheduler and the configuration of the cooperative scheduler (section 6.6).

The level of predictability of the overall system is adversely affected by the level of flexibility that is allowed in it. The highest level of predictability will be achieved if all the task periods and execution times remain constant at run time. Any of the following factors will have a detrimental effect on the predictability of the overall system:

1. Variations in the execution times of the pre-empting or cooperative tasks.
2. Change in the operating mode of the cooperative scheduler
3. Variations in the task periods in the cooperative or pre-empting tasks.

Regardless of the operating configuration, from the software's point of view, it is always possible to find how much time remains till the next pre-emption. In addition to the definition of non-pre-emptable critical sections, this also opens up the possibility of using the Timed Resource Access Protocol (TRAP) to manage resources used by both the cooperative and pre-empting tasks (Maaita 2008).

8.6 Conclusion

This chapter presents the novel dual scheduler architecture for flexible limited pre-emption time-triggered implementations. This architecture, uses seprate timers for the purely cooperative and limited pre-emption schedulers to allow very high level of flexibility that cannot be achieved by a single timer architecture (like the time-triggered hybrid) without significant overheads.

The flexible architecture allows the designer to choose between a ridigd but highly predictable architecture and a flexible architecture at the cost of reduced predictability. This flexibility can be used to tailor an implementation to specific application requirements.

Like in the case of the multi phase cooperative scheduler in chapter 6, even in the most flexible configuration, it can still be calculated when the next event (and in the case of cooperative tasks, the possible pre-emption) will occur.

9 Brushless DC Motor Case Study

This chapter presents the setup used and the results of the brushless DC motor case study[3].

9.1 Target Platform

The brushless DC motor (BLDCM) controller is implemented using ST Microelectronics' STM32F103RB microcontroller. This microcontroller uses ARM's Cortex M3 core as its basis and comes with many peripherals (four 16 bit timers with capture and compare IO, Onboard ADC, etc.) that are of great value in motor control applications.

The brushless motor used for the research was a Maxon Motor 32 volt, 50 watt permanent magnet brushless motor. This motor was fitted with the hall sensors and quadrature incremental encoder for absolute position and speed measurements respectively.

The SGS Thomson L6234 three phase driver IC was used to implement the motor drive. The L6234 operates on a wide range to supply voltages, accepts TTL input signals, and includes inbuilt cross conduction protection and thermal shutdown.

9.2 Components

This section covers the specifics of the implementation of the test bed.

9.2.1 Time-triggered scheduler setup

The TTC + TTP scheduler setup was used for the time-triggered implementation. The Cortex M3's System Tick timer is used to generate the periodic interrupts for the

[3] Parts of this chapter have been published previously in (Hanif, Pont et al. 2008)

cooperative scheduler. The pre-emptive scheduler is driven by the 16 bit TIM2 general purpose timer module. A 10 µs prescaler is used to drive the TIM2 timer.

These two schedulers are used to run all the tasks in the time-triggered implementation of the controllers.

9.2.2 Speed measurement

STM32F103's general purpose timer module 3 (TIM3) is interfaced with the quadrature encoder on the motor. The counter's value is incremented or decremented by the sequence of pulses coming from the encoder.

A cooperative task is used to measure the motor's speed by reading the counter's value and resetting it. It is important that the task for reading the speed is the first task in a tick as jitter in starting time of the task will cause an error in the speed measurement.

9.2.3 Speed controller

A cooperative task can be used to calculate the control outputs. The high speed 32-bit controller allows for reasonably fast calculation of the speed controller algorithm.

A TIM1 output was used as a 30 kHz pulse width modulator signal to control the amount of power going to the motor. The PWM signal is used to modulate the high side transistors in the drive circuit while the low side transistors are kept on in accordance with the lookup table (Table 9-2).

9.2.4 Commutation and drive setup

A pre-empting task checks the hall sensor outputs and updates a six bit output pattern. Combinational logic was used to modulate the generated patterns with the PWM signal to control the order and time for which the phases were energised using the L6234

driver IC. Table 9-1 provides the inputs and corresponding outputs for a single half bridge of the L6234.

Table 9-1: L6234's input combinations and corresponding output configurations.

Enable (ENx)	Input (INx)	Output (OUTx)
0	0	Floating
0	1	Floating
1	0	Low
1	1	High

Table 9-2 shows the possible hall sensor outputs and the corresponding phases drive sequences that have to be generated for clockwise rotation of the motor. It also shows the input combinations needed for the L6234 driver.

Table 9-2: Hall sensor inputs and corresponding drive configuration for clock wise rotation of brushless DC motor.

Hall sequence (ABC)	Motor windings			L6234 inputs		
	A (Out1)	B (Out2)	C (Out3)	EN1, IN1	EN2, IN2	EN3, IN3
000 (Invalid)	Floating	Floating	Floating	0, 0	0, 0	0, 0
001	Floating	Low	High	0, 0	1, 0	1, 1
010	Low	High	Floating	1, 0	1, 1	0, 0
011	Low	Floating	High	1, 0	0, 0	1, 1
100	High	Floating	Low	1, 1	0, 0	1, 0
101	High	Low	Floating	1, 1	1, 0	0, 0
110	Floating	High	Low	0, 0	1, 1	1, 0
111 (invalid)	Floating	Floating	Floating	0, 0	0, 0	0, 0

9.2.5 Data to PC

The measured speed and additional data is sent to the PC over an RS232 serial link. Although the speed is measured every 1ms, it is sent to the pc every 10ms.

9.3 Commutation Generation Techniques

For the implementation of sensor based commutation sequence generator in a purely time-triggered manner using the proposed architecture, the following Time-Triggered systems were tested:

1. By statically scheduling a pre-empting task to update the commutation sequence with very short period (every 50µs). This is close to what can be achieved with existing timeline and TTH architectures. This setup was referred to as the Time-Triggered High pre-emption Rate Static (TT HRS).

2. By statically scheduling a pre-empting task to update the commutation sequence with a moderately short period (every 250µs). This is done to reduce the overheads because of commutation update. This setup was referred to as the Time-Triggered Low pre-emption Rate Static (TT LRS).

3. By using a pre-empting task that changes its period dynamically. The general idea is that once the commutation takes place, the task does not need to run till it is closer to the time at which the next commutation update is expected. The algorithm used for doing this is given in Code listing 9.1. This setup was referred to as the Time-Triggered Mixed pre-emption Rate Flexible (TT MRF).

```
If change in hall sensor sequence
    Update drive circuit's activation pattern
    Calculate the expected time of the next commutation
    Set task's period to: (0.9 * expected  period) % 50 µs
Else
    Change task's period to 50 µs
End
```

Code listing 9.1: Algorithm for pre-emptive variable rate commutation update task.

9.4 Results

It was planned to compare the open-loop, no-load-speed performance of the three time-triggered implementations presented in the previous section against an event-triggered system. However, during implementation of the event-triggered system, spurious interrupts caused damage to the motor's drive circuit. It was decided that the time-triggered systems will be compared to the performance of a simulated event-triggered system.

The simulated event-triggered system (ET sim) was implemented by using two STM32F103 microcontrollers. One was responsible for running only the modified commutation update sequence in a polling loop, while the second controller took care of all the data collection. The first controller was able to run the commutation update sequence every 2.5 µs.

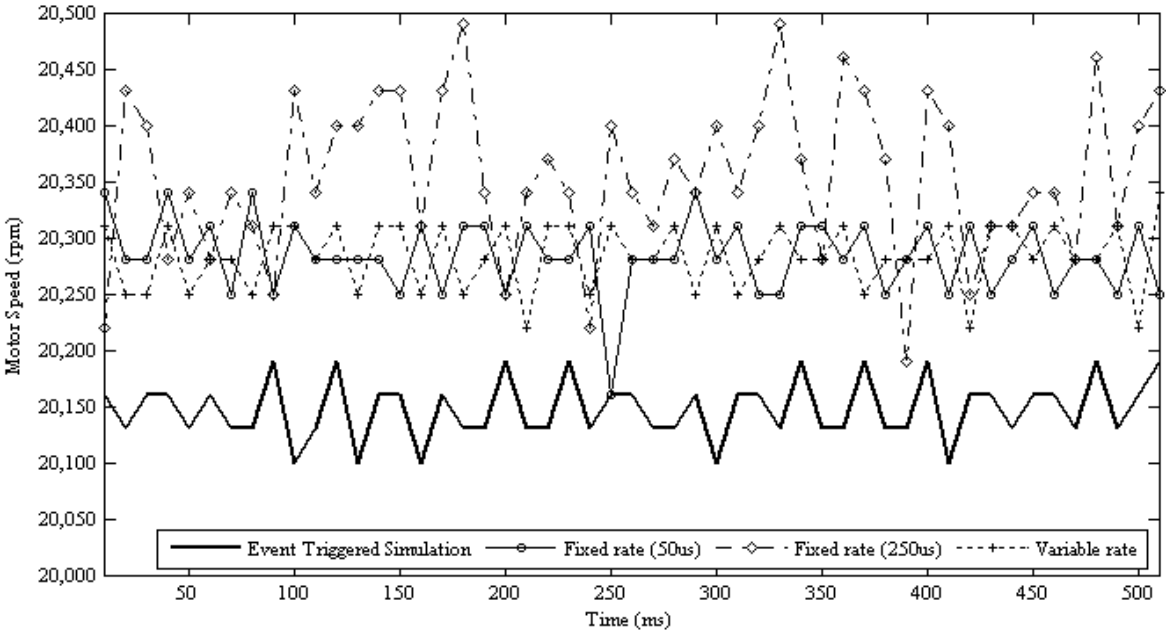


Figure 9-1: Open loop maximum speed plots for different implementations.

For all four test models, the open loop speed, number of pre-emptions per second, the duration of each pre-emption and the current through the drive circuit was measured. A standard deviation and average speed were calculated using 300 speed readings for all

test models (over a duration of 3s). The standard deviation of the speed readings offers a measure of the effectiveness of each implementation. Lower standard deviation tends to indicate smoother and more efficient running of the motor as can seen from the current readings of the models.

Table 9-3: Comparison of different implementation methods.

	ET sim	***TT HRS***	***TT LRS***	***TT MRF***
Speed Minimum (RPM)	20,100	20,160	20,190	20,220
Speed Maximum (RPM)	20,190	20,340	20,520	20,340
Speed Average (RPM)	20,148	20,287	20,362	20,284
Standard deviation in measured speed	27.421	28.861	67.996	30.543
Average number of pre-emptions per second	2014.8	20,000	4,000	5,798.3
Duration of each pre-emption	2.5 µs	8 µs	8 µs	8-12 µs
Effective CPU loading	0.5 %	16 %	3.2 %	5.45 %
Current consumption (A)	0.16	0.17	0.53	0.18
Efficiency measure (Speed (RPM) / Current (A))	125925	119335.3	38418.87	112688.9
Efficiency relative to ET sim	100 %	94.77 %	30.50 %	89.49 %

For the event-triggered simulation's number of pre-emptions and the CPU loading have been calculated using the average speed of the motor, the number of events per revolution and the time to run through the polling loop.

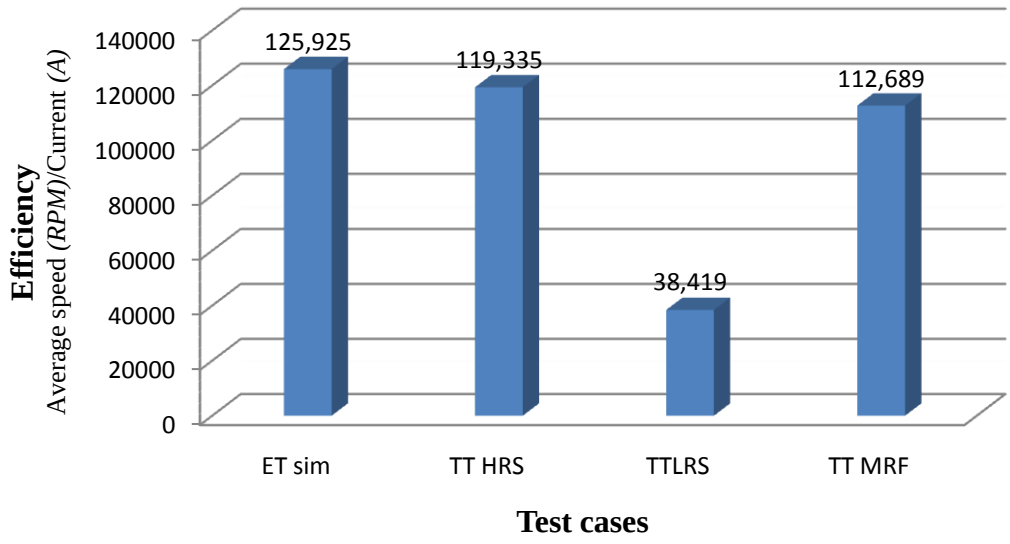


Figure 9-2: A comparison of the efficiency of the test cases.

Figure 9-1 shows the plot of the speed readings from the four models. From both this figure and the data from Table 9-3, it can be seen that the event-triggered implementation offers the best open loop control scheme. In addition to the superior open loop control, the CPU loading, because of the commutation update task for achieving this, is the lowest among all models. However, it must be noted that this superior performance is achieved at the cost of determinability (i.e. at any point in time, it is never known when the next pre-emption will occur).

The TT HRS model offers the second best performance amongst all models. The 50 µs period of the commutation update task proves to be fast enough to be comparable with the event-triggered simulation model. The constant period of the pre-empting task also offers excellent determinability as it is known well in advance when the next pre-emption will occur. However, this performance and predictability is achieved at the cost of a comparatively high CPU loading.

The TT LRS model tries to reduce the CPU loading by lowering the rate at which the commutation update task is called. This results in the expected lower processor loading

but sacrifices the high speed open loop performance. This arrangement can still be used in safety critical applications that require high determinability but do not require the operation of the motor at high speeds.

The TT MRF model tries to reduce the CPU loading by varying the rate of execution of the commutation update task as given by the algorithm in Code listing 9.1. By using the 90% value of the time till the next commutation, the system ensures responsiveness when the motor is accelerating. Although the determinability of this model is not as high as the first two time-triggered models, it is still better than the event-triggered model as at any point in time, the time till the next pre-emption can be easily determined. The determinability of this system is further increased by the modulus operation in the calculations of the new period after a commutation. The modulus operation effectively limits the pre-empting task to a fixed number of time slots in which the pre-emption can occur and thereby greatly limits the permutations of the program execution path.

9.5 Conclusion

The brushless motor control case study demonstrates how the flexible limited pre-emption architecture could be used to provide the foundation software architecture for motor control applications. The use of the flexible TT architecture allows the designer to make trade offs between performance, CPU utilization and system predictability.

The results indicate that similar performance can be achieved at lower CPU overheads if the desired predictability levels can be reduced.

10 Conclusions and Future work

The conclusions drawn from the work presented in this thesis are presented in this chapter along with potential applications and possible extensions. Recommendations for further work are also made.

10.1 Summary of Thesis Contributions

The contributions of this thesis are as follows:

- The concept of flexible time-triggered (TT) scheduling was introduced. At its core lay the novel idea of changing the task and cycle periods without changing the order of execution of tasks at the run time. In this setup, the time at which next event in the system occurs is known in advance. This arrangement brings about the possibility of using statically scheduled architectures to interface with and control pseudo-periodic systems (i.e. systems where the cycle period varies from one cycle to the next instead of remaining fixed at a particular value).
- Two flexible time-triggered scheduler architectures were presented:
    i. The cooperative flexible TT scheduler incorporated the above mentioned technique along with support for multiple task sets that had been suggested by others (section 2.4.1.7).
    ii. A novel dual scheduler arrangement was presented to allow addition of flexible limited pre-emption in a predominantly cooperative environment. This arrangment provided greater flexibility than a simple extension of a single hybrid scheduler architecture like the TTH (section 3.2.3) or its derivatives, while still retaining the ability to determine the time till the next interrupt.

- Two representative case studies were carried out to test the performance of the flexible TT architecture in real world applications:
    i. An engine synchronisation case study was used to test the performance of the cooperative flexible architecture and its underlying theory. The results of this case study clearly demonstrated the ability of the flexible cooperative time-triggered architecture to synchronise task executions with a generated crank signal over a wide range of engine operating speeds.
    ii. A brushless motor control case study was used to test the effectiveness of the limited pre-emption flexible TT scheduler. The results of this case study demonstrated how the dual scheduler architecture allowed trading predictability for performance and vice versa.

10.2 Review of the Contributions

This thesis makes contributions to the fields of real-time scheduling and to the application areas of internal combustion engine control architecture and brushless DC motor control architectures.

10.2.1 Scheduling theory and architectures

In an attempt to overcome the limitations of the cooperative static schedulers while retaining their high levels of predictablity, this thesis introduces the novel concept of flexible static scheduling where the task execution orders remain fixed (i.e. set and verified at design time), but the overall period of the major cycle is changed by varying the tick periods. In addition to this, it was also determined that the flexibility of the

TTC scheduler could be further increased by allowing the major cycle to be composed of segments with their own task sets and tick periods. When coupled with the existing concept of multiple operating modes with unique task sets (Baker, Shaw 1988, Kopetz, Nossal et al. 1998, Xu, Parnas 2000) this gave form to the construct of phases. The Time-Triggered Multi Phase Cooperative (TTMPC) scheduler architecture combines all these concepts in a single cooperative time-triggered scheduler.

In order to cope with situations and applications, where some limited form of pre-emption is required, the following two separate architectures are developed and evaluated (Sections 8.3, 8.4 and 8.5):

1. A hybrid scheduler based on the concept of the Time-triggered Hybrid presented by Pont (Pont 2001).
2. A novel dual scheduler architecture that employs separate schedulers for the cooperative tasks and the pre-emptive tasks.

Based on this evaluation, the dual scheduler architecture was selected due to its greater flexibility.

These proposed architectures (the TTMPC alone, or the TTC / TTMPC in conjunction with the TTP) allow the user to vary the amount of flexibility in the system so as to implement solutions that range from highly predictable but rigid solutions to very flexible solutions at the cost of reduced predictability. It should be noted that even in the most flexible configuration, due to the completely time-triggered basis of the architectures, it is always possible to determine when the next interrupt / pre-emption will occur.

10.2.1.1 Features and limitations of the proposed architectures

The Time-triggered Multi Phase Cooperative (TTMPC) scheduler implements the proposed architecture as follows (see sections 6.4 and 6.5 for more details):

- Support for multiple phases with unique task sets and tick periods.
- Support for automatic phase changes at the end of a finite duration phase. This aids in the formation of segments of a major cycle.
- Support for forced phase change at any point in the cycle. The change is enforced after all the tasks set to run in the tick with the phase change request have executed. In case of a multi mode system, this functionality helps to jump from one operating mode / state to an other.
- Support for changing the tick periods of a phase at run time. This effectively changes the period of the major cycle in which that phase is included. The new period can only be within a range that is specified during system initialisation so as to allow minimum and maximum limits to be imposed at design time.
- While the system's operating mode and tick periods can be changed at run time, the task execution orders in each phase remain the same as what is specified at the design time.
- Support for transient overloads across phase boundaries in a major cycle.

Some of the key points of the cooperative + TTP architecture are as follows (see section 8.4):

- It has the ability to include pre-empting tasks in a predominantly cooperatively scheduled system.
- One pre-empting task cannot pre-empt another pre-empting task.

- The periods and offsets of the pre-empting tasks do not have to be constrained to multiples of a base period or the tick period of the cooperative tasks. This allows fine timing control for these tasks.
- The periods of the tasks can be changed to any arbitrary value limited by the scheduler's timer resolution at run time. However, stricter limits can be imposed by the user to achieve a more predictable behaviour.

10.2.1.2 Predictability of Flexible Time-Triggered Systems

The increased flexibility afforded by the two proposed scheduler architectures has an adverse effect on the predictability of the overall system.

Unlike a simple cyclic executive with a single task set and fixed task periods is highly predictable or even determinable under some conditions (i.e. it can be found what task the system would be running at any point in the future), the flexible time-triggered architecture is not completely determinable (i.e. even in the worst case, at any point during the execution, while it is known when the next tick will occur and what tasks would be run next, it might not be possible to determine the state of the system at an arbitrary point of time in the future). These issues were discussed in sections 6.6 and 8.5 for the cooperative and dual scheduler architectures respectively. In any case, even in the most flexible configuration, in a flexible TT system, it can always be found out when the next event or interrupt will occur in the system. It allows for the user software to prepare for it and utilise techniques like the TRAP protocol to avoid conflicts in the shared resources. This is better than an event-triggered system where even if statistical limits can be placed on when an event might occur, the exact moment an event occurs cannot be determined.

10.2.2 Internal combustion engine synchronisation

Synchronising with and controlling reciprocating internal combustion engines provides a significant challenge to time-triggered systems. The execution of tasks has to be synchronised with the internal position of the crank shaft (provided by the crank angle sensor). Conventionally, this task is performed by using events which are generated on the basis of internal orientation of the various engine parts. The use of the novel flexible time-triggered scheduling with variable tick periods opens up the possibility of using a time-triggered (TT) design in an application area that has seen wide spread use of event-triggered (ET) designs and has no mention of purely time-triggered implementations in the published literature.

A brief description of the contributions of this study is as follows:

- A crank sensor interface that could work efficiently with both event-triggered and time-triggered task scheduling has been developed and tested. This efficiency was obtained by using hardware DMAs to capture the pulses for time-triggered implementation allowing the crank interface tasks to be polled at a rate considerably lower than the expected rate of arrival of pulses.
- Synchronisation tests were carried out by providing the same crank signal pattern to event-triggered, traditional time-triggered and flexible time-triggered architectures. The results of the synchronisation tests indicate that while the traditional TT based design is inadequate for use in this application area, the flexible TT based design was able to keep itself synchronised with simulated crank signal through various speed changes across the engine's operating range.
- Despite this improvement in synchronising ability, the performance of the ET was better than that of the flexible TT (smooth tracking during speed changes).

However, the performance of the TT design is closely linked with its ability to predict the next period. In case the TT design is actually controlling the engine (fuel injection and moment of spark ignition) and in effect, the changes in the crank signal, it should be possible to take this information into account so as to make better predictions for the next cycle period.

- The proposed techinque of using a DMA to capture potentially noisy data gives the flexible TT design an advantage over the ET design which has to capture each transition as it arrives because it does not know, in advance, whether it is a glitch or start of a normal pulse.
- It was also shown that increasing the resolution of the criteria used to predict the next period (in this case, the period of the cycle) has a positive effect on the synchronisation performance.
- The developed flexible TT solution can be used as a foundation for an engine controller with highly predictable software.

10.2.3 Brushless DC motor control

Brushless DC motor (BLDCM) drives and 3 phase motor drives, in general, are challenging for time-triggered systems because of the high rate of polling that is required to quickly detect changes in the position sensors. While it is possible to run such a motor with a static time-triggered architecture (like the Time-Triggered Hybrid scheduler), using a flexible TT design allows the motor to be operated at high speeds with reduced CPU usage compared to a static TT design.

In this case study a sensor based block commutation 3 phase drive was implemented for a brushless DC motor in various architectures for comparison purposes. Only open loop speed tests were carried out to investigate the effects of the design choices of the test

cases on the performance as closed loop tests might have compensated for the performance differences by taking feedback into consideration.

The main contributions of this case study were as follows:

- The efficiency of various architectures was compared. While the efficiency of the simulated ET was the best out of the four cases, the high rate static and mixed rate flexible time-triggered implementations came around 94.8 % and 89.5% of the open loop event triggered implementation.
- Implications of the design choices on the predictability of the test cases were discussed. With the static schedules, it was possible to determine when all the pre-emptions would occur at design time. With the flexible TT, it was still possible to determine how much time remained till the next pre-emption. Also, limiting the period of the pre-empting task to integer factors of the cooperative tick period had the effect of limiting the points in the cooperative tick where pre-emption could occur.

10.3 Alternate Application Areas for Flexible Time-Triggered

Two potential alternate application areas for flexible time-triggered architecture are:

10.3.1 Long term tracking of geographical features

Land and mud slides are a cause of large number of casualties worldwide. The failure mode of these slids may range from a few hours to several days (Rose, Hungr 2006) and conditions that can lead to failures take even longer to come into effect (e.g. for mud slides, water has to reach the failure prone fault surface which can take days after the start of heavy rains). Signs of increased chances of failure include change in rate of movement of the ground. Some possible methods that could be used to track the rate of

movement could be comprised include triangulation of position or the nodes (either with the help of active beacons or by taking measurements of known fixed features), measuring distances between adjacent nodes, strain gauges (for brittle surfaces), etc.

It is possible to use the proposed architectures for the implementation of battery powered long term monitoring networks to monitor geographic slope stability. The sensor nodes used for this application could be loosely based on the shared clock scheduler architecture. Considering that at low risk times, the measurements have to be taken at a very low rate (a couple of readings per day or even slower), keeping the receivers powered on the slaves will be a waste of power. Alternately, the slave nodes could power down their receivers for the majority of the time between the readings, turning them on only a limited time before the synchronising tick message is expected. The operation of the slave nodes can be based on the following steps:

1. Turn on receiver and enter a waiting state.
2. Upon receiving a valid tick message, use variable tick periods of a synchronisation state to compensate for any timing error.
3. After the synchronisation state, take the measurements and send the relevant results to the master for evaluating the situation. After evaluation of the captured data, the master specifies the time till the next reading
4. The slave nodes go into power saving state for the specified duration.

The operation of the master node will differ from the slave nodes as it will have to send out the synchronisation tick messages. In addition the master might keep its receiver on all the time in order to detect reset on a slave and help it to initialize.

10.3.2 Wireless sensor networks

Wireless sensor networks are used to monitor and log physical and environmental data. These are useful in a wide range of applications and are a topic of active research (Werner-Allen, Lorincz et al. 2006, Xu, Rangwala et al. 2004, Cardei, Du 2005, Sohrabi, Gao et al. 2000). Some of the main requirements for the nodes in wireless sensor networks are:

- Battery management and life time maximisation.
- Reliability.
- Flexible configuration.

The flexible time-triggered architectures proposed in this thesis could be used for implementation of decentralized wireless sensor network where the nodes are allowed to run independently but also retain the capability to resynchronise if required. It is difficult to go into more details at an abstract level as the nodes in the network are usually highly specialized for specific applications (e.g. some require high data transmission rates (Werner-Allen, Lorincz et al. 2006, Xu, Rangwala et al. 2004) while others might require multi path data transmission for reliability in hostile environments).

10.4 Future Work

This section outlines the areas where further work will help expand the understanding and usefulness of the work presented in this thesis.

10.4.1 Scheduler architectures

The following items may require further work to expand the usefulness and reliability of the scheduler architectures presented in this thesis:

- Integration of task guardians in the TTMPC and dual scheduler architectures would help provide recovery options in the case of task overruns. The task guardian for the TTC should be easily extendable to the TTMPC architecture.
- Reducing scheduling jitter from the scheduler side using code balancing or single path programming. Single path implementation of the schedulers might help with their certification for high reliability and safety critical applications (as it limits the number of flow paths through the code.).
- Adapting existing automatic schedule determination techniques (such as TTSA1 and TTSA2 (Gendy, Pont 2008, Gendy 2009)) to take advantage of the multi segment cycles. Implementation of efficient techniques could also open the path for systems that are able to compute new static task sets at run time in response to anticipation of changes in an operating environment.
- More work is required to be able to quantify the level predictability of a general architecture and more specifically, an implemented system. This issue has been highlighted but still remains unresolved (John 1988, Halang, Gumzej et al. 2000).
- The scheduler architectures presented in this thesis were limited to single processor architectures. More research may be required to integrate this into

distributed shared clock scheduling. Some thoughts on this issue were presented earlier in this chapter but more work may be undertaken for realization and further analysis.

- The technique of varying the task periods can also be applied to dynamic schedulers and should be studied in detail to understand the effects of such changes on the system behaviour.

10.4.2 Internal combustion engines

The implementation of an engine controller based on the flexible TT architecture is required to verify its feasibility. Work along these lines had to be abandoned due to complications caused by a general absence of manufacturer's specifications and control algorithms and the limited resources available. Without the implementation of the controller, detailed performance comparisons between the conventional and proposed architecture may not be carried out.

10.4.3 3 phase motor drives

The comparison of the efficiency for the different control schemes presented in this thesis was limited due to the low resolution of the current measurements and the lack of ability to apply constant loads at various speeds. These are required to understand in detail the effects of various control schemes on the efficiency of the motor.

10.5 Final Conclusion

The work done in this thesis is aimed at exploring the possibility of applying a variation of the standard time-triggered approach to a class of systems that are usually considered to be outside of the domain of time-triggered solutions. A gap was identified in real-time scheduling practices and accordingly, the implications of varying task and cycle periods at run time were studied.

The two flexible TT scheduler architectures (the purely cooperative – TTMPC and the limited pre-emption dual scheduler TTxC + TTP) which have been developed have opened up the possibility of trading the predictability for performance and vice versa depending on the design choices. This flexibility and its advantages were highlighted in the results and observations of the engine synchronization and brushless DC motor case studies.

The software developed for both the case studies can form the foundation framework for controllers utilising flexible TT architectures.

References

Adler, P. (1998) “Apollo 11 Program Alarms”, NASA, WWW website (last accessed 13th August 2009), URL: http://www.hq.nasa.gov/office/pao/History/alsj/a11/a11.1201-pa.html.

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

ARM (2004) “ARM7TDMI - Technical Reference Manual”, ARM, available online (last accessed 16th September 2011), URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf.

ARM (2010) “Cortex-M3 Devices - Generic User Guide”, ARM, available online (last accessed 16th September 2011), URL: http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/DUI0552A_ cortex_m3_dgug.pdf.

Atmel (2006) “AVR443: Sensor-based control of three phase Brushless DC motor”, Atmel, available online (last accessed 13th August 2009), URL: http://www.atmel.com/dyn/resources/prod_documents/doc2596.pdf.

Atmel (2007) “AVR449: Sinusoidal driving of 3-phase permanent magnet motor using ATtiny261/461/861”, Atmel, available online (last accessed 13th August 2009), URL: http://www.atmel.com/dyn/resources/prod_documents/doc8030.pdf.

Audsley, N., Tindell, K. and Burns, A. (1993) “The end of line for static cyclic scheduling?”, in the Fifth Euromicro Workshop on Real-Time Systems, 22-24 Jun 1993, pp. 36-41.

Austen, I. (2003) “A Chip-Based Challenge to a Car's Spinning Camshaft”, New York Times, WWW website (last accessed 13th August 2009), URL: http://www.nytimes.com/2003/08/21/technology/what-s-next-a-chip-based-challenge-to-a-car-s-spinning-camshaft.html?sec=&spon=&pagewanted=1.

Babaoglu, Ö., Marzullo, K. and Schneider, F.B. (1990). “Priority Inversion and its Prevention”, Department of Computer Science, Cornell University.

Baker, T.P. and Shaw, A. (1988) “The cyclic executive model and Ada”, in the proceedings of the Real-Time Systems Symposium, pp. 120-129.

Barr, M. (2003) “Choosing an RTOS”. *Embedded Systems Programming*, WWW websit (last accessed 12 August 2011), URL: http://www.eetimes.com/discussion/other/4024563/Special-Report-Choosing-an-RTOS.

Bellis, M., “The History of the Automobile”, About.com, WWW website (last accessed 13th August 2009), URL: http://inventors.about.com/library/weekly/aacarsgasa.htm.

Bertorelli, P. (2010) “Lycoming's IE2: Perfect Timing?” AV Web, WWW website (last accessed 30th July 2011), URL: http://www.avweb.com/blogs/insider/AvWebInsider_LycomingEyeEeeTwo_202445-1.html [July 30, 2011].

Bosch (2004) “25 years of Bosch Motronic: Think tank under the bonnet”, Bosch, WWW website (last accessed 28th July 2011) URL: http://www.bosch.com/content/language2/html/3074_3184.htm.

Brown, W. (2001) “AN857: Brushless DC motor control made easy”, Microchip, online application note (last accessed 8th May 2011), URL: http://ww1.microchip.com/downloads/en/AppNotes/00857a.pdf.

Burns, A. (1995) “Generating feasible cyclic schedules”, Control Engineering Practice, volume 3 issue 2, pp. 151-162.

Buttazzo, G.C. (2005a) “Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications” Second edition edn., Springer.

Buttazzo, G.C. (2005b) “Rate monotonic vs. EDF: judgment day”. Real-Time Systems, volume 29 issue 1, pp. 5-26.

Buttazzo, G.C., and Gai, P. (2006) “Efficient EDF Implementation for Small Embedded Systems”, in the proceedings of the 2nd Int. Workshop on Operating System Platform for Embedded Real-Time applications (OSPERT 2006), Dresden, Germany, July 2006

Cardei, M. and Du, D.Z. (2005) “Improving Wireless Sensor Network Lifetime through Power Aware Organization” Wireless Networks, volume 11 issue 3.

Chapman, S.J. (1985) “Electric Machinery Fundamentals” McGraw-Hill.

Continental Motors, “Continental Motors: Company history timeline” homepage of Continental Motors, WWW website (last accessed 28th July 2011), URL: http://www.genuinecontinental.aero/history.aspx.

Dempsey, M.W. (2011) “Aircraft Piston Engines – time for some new technology?”, Get Aviation blog, WWW website (last accessed28th July 2011), URL: http://get-aviation.com/blog/aviation-thoughts/aircraft-piston-engines-%E2%80%93-time-for-some-new-technology.

Denton, T. (1995) “Automobile electrical and electronic systems” Arnold, ISBN 0-340-58604-4

Deverge, J. and Puaut, I. (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, July 2005 2005, pp. 7-10.

Dijkstra, E.W. (1970) “Notes on structure programming”, Technological University of Eindhoven.

Engblom, J., Ermedahl, A., Sjoedin, M., Gustafsson, J. and Hansoon, H. (2003) "Worst-case execution-time analysis for embedded real-time systems", Journal of Software Tools for Technology Transfer (STTT), volume 4 issue 4, pp. 437-455.

Eyles, D. (2004). "Tales from the lunar module guidance computer", Advances in the Astronautical Sciences, volume 118.

FAA (2008) "Pilot's Handbook of Aeronautical Knowledge" U.S. Department of Transport, available online (last accessed 19th July 2011), URL: http://www.faa.gov/library/manuals/aviation/pilot_handbook/media/FAA-H-8083-25A.pdf

Fidge, C.J. (2002) "Real-Time Scheduling Theory" Software verification research center, The University of Queensland.

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

Gendy, A.K. (2009) "Techniques for scheduling time-triggered resource-constrained embedded systems", Ph.D. thesis, University of Leicester.

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

Gendy, A.K. and Pont, M.J. (2007) "Towards a generic 'single-path programming' solution with reduced power consumption", Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007), 4-7 September 2007.

Grasblum, P. (2001) "AN1916D: 3-Phase BLDC Motor Control with Hall Sensors Using DSP56F80x" Motorolla, application note (last accessed 8th May 2008) URL: http://www.motorola.com.cn/semiconductors/mcudsp/forms/appnote/AN1916.pdf.

Halang, W.A., Gumzej, R., Colnaric, M. and Druzovec, M. (2000) "Measuring the performance of real-time systems" Real-Time Systems, volume 18 issue 1, pp. 59.

Hanif, M.A. (2004) "Brushless Motor Speed Controller", M.Sc. dissertation, University of Leicester.

Hanif, M.A., Pont, M.J. and Ayavoo, D. (2008) "Implementing a simple but flexible time triggered architecture for practical deeply embedded applications", 4th UK Embedded Forum, September 2008, Southampton, UK.

Hughes, Z.H. and Pont, M.J. (2008) "Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed", Transactions of the Institute of Measurement and Control, volume 30, pp. 427-450.

Hughes, Z.H. and Pont, M.J. (2004) "Desing and test of a task guardian for use in TTCS embedded systems" UK Embedded Forum, October 2004, University of Newcastle upon Tyne, pp. 16-25.

Jeffay, K., Stanat, D.F. and Martel, C.U. (1991) "On non-preemptive scheduling of periodic and sporadic tasks", *the 12 th IEEE Symposium on Real-Time Systems* 1991, pp. 129-139.

Jinnelov, M. (2002) "Analysis of an Engine Control System in Preparation of a Real-Time Database", M.Sc. thesis, Linköpings University.

John, A.S. (1988) "Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems", Computer, volume 21, issue 10, pp. 10-19.

Joinathan, J. (1994) "Safety-Critical Computing: Hazards, Practices, Standards, and Regulation", University of Washington, WWW website (last accessed 12th September 2011) URL: http://staff.washington.edu/jon/pubs/safety-critical.html.

Jones, E.M. (2011) "The first lunar landing", NASA, WWW website (last accessed 20th September 2011), URL: http://next.nasa.gov/alsj/a11/a11.landing.html.

Kalinsky, D. (2001) "Context switch." Embedded Systems Programming, volume 14 issue 1, pp. 94; 105.

Kopetz, H., Nossal, R., Hexel, R., Kruger, A., Millinger, D., Pallierer, R., Temple, C. and Krug, M. (1998) "Mode handling in the Time-Triggered Architecture." Control Engineering Practice, volume 6, issue 1, pp. 61.

Kopetz, H. (1991) "Event-triggered versus time-triggered real-time systems", Proceedings of the International Workshop on Operating Systems of the 90s and Beyond, Springer-Verlag, pp. 87-101.

Laplante, P.A. (1997) "Real-Time Systems Design and Analysis - An Engineer's Handbook", IEEE Press.

Lehoczky, J., Sha, L. and Ding, Y. (1989) "The rate monotonic scheduling algorithm: exact characterization andaverage case behaviour", Proceedings of Real Time Systems Symposium, pp. 166-171.

Levenson, N. (1995) "Medical Devices: The Therac-25", Software: System safety and computers. Addison-Wesley.

Little, M. (2009) "Continental Motors Announces Turbo FADEC Receives FAA Certification", Continental Motors, press release (last accessed 28th July 2011), URL: http://www.genuinecontinental.aero/DOCUMENTS/CertifiedTurboFADEC.pdf.

Liu, C.L. and Layland, J.W. (1973) "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM, volume 20, issue 1, pp. 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.” The Journal of Real-Time Systems, volume 4, pp. 37-53.

Lycoming (2010) “Lycoming iE2 Technology Offered on Lancair Evolution”, Lycoming, WWW website (last accessed 15th July 2011), URL: http://www.lycoming.com/news-and-events/press-releases/release-07-26-10a.html.

Maaita, A.A. (2008) “Techniques for Enhancing the Temporal Predictability of Real-Time Embedded Systems Employing a Time-Triggered Software Architecture”, Ph.D. thesis, University of Leicester.

Maaita, A. and Pont, M.J. (2005) “Using ‘planned pre-emption’ to reduce levels of task jitter in a time-triggered hybrid scheduler”, Proceedings of the Second UK Embedded Forum, October 2005, University of Newcastle upon Tyne, pp. 18-35.

Martin, F.H. (1994) “Apollo 11: 25 years later”, NASA, WWW website (last accessed 13th September 2011), URL: http://www.hq.nasa.gov/alsj/a11/a11.1201-fm.html.

McCabe, T.J. (1976) “A Complexity Measure”, IEEE Transactions on Software Engineering, volume 2, issue 4, pp. 12-308.

Mead, J.S. (1991) “Rover 820 Owners Workshop Manual”, Haynes Publishing Group.

Mezzetti, E., Holsti, N., Colin, A., Bernat, G. and Vardanega, T. (2008) “Attacking the source of unpredictablity in the instruction cache behaviour”, proceedings of the 16th International Conference on Real-Time and Network Systems, 4 November 2008, pp. 151.

Motorola (2001) “Reference Manual: M-CORE with M210/M210S Specifications”, Motorola Inc.

NXP (2008) “UM10114: LPC21xx and LPC22xx User manual”, NXP, Online user manual (last accessed 8th December 2010), URL: http://www.nxp.com/documents /user_manual/UM10114.pdf

Oxford English Dictionary (1989) “Oxford English Dictionary 2nd edition”, Oxford: Clarendon Press.

Phatrapornnant, T. and Pont, M.J. (2006) “Reducing Jitter in Embedded Systems Employing a Time-Triggered Software Architecture and Dynamic Voltage Scaling”, IEEE Transactions on Computers, volume 55, issue 2, pp. 113-124.

Pont, M.J., Kurian, S., Wang, H. and Phatrapornnant, T. (2007) “Selecting an appropriate scheduler for use with time triggered embedded systems”, 12th European Conference on Pattern Languages of Programs 2007.

Pont, M.J. (2001) “Patterns for Time-Triggered Embedded Systems”. Addison Wesley.

Proctor, F.M. and Shackleford, W.P. (2001) “Real-time Operating System Timing Jitter and its Impact on Motor Control”, proceedings of the 2001 SPIE Conference on Sensors and Controls for Intelligent Manufacturing II, Vol. 4563-02.

Pulkrabek, W.W. (1997) “Engineering Fundamentals of the Internal Combustion Engine”, Prentice-Hall.

Puschner, P. (2003) “The single-path approach towards WCET-analysable software”, Proceedings of IEEE International Conference on Industrial Technology, December 2003, pp. 699-704.

Puschner, P. and Burns, A. (2002a) “Transforming execution-time boundable code into temporally predictable code”, Proceedings of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems (DIPES 2002), pp. 163-172.

Puschner, P. and Burns, A. (2002b) “Writing temporally predictable code”, Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, pp. 85-91.

Rose, N.D. and Hunger, O. (2006) “Forcasting potential slope failure in open pit mines - contingency planning and remediation”, International Journal of Rock Mechanics & Mining Sciences, volume 44, pp. 308-320.

RoverMEMS - MPi/SPi, anonymous online technical manual (last accessed 19th July 2011), URL: http://www.gaima.co.uk/peter/RoverMEMS.pdf.

Savier (1995) “Electronic ignition for aircraft (part 2)”, Sport Aviation, WWW website (last accessed 16 July 2011), URL: http://www.lightspeedengineering.com/ Technicalities/sport_aviation95.html.

Scheler, F. and Schroder-Preikschat, W. (2006) “Time-triggered vs. event-triggered: A matter of configuration?”, GI/ITG Workshop on Non-Functinal Properties of Embedded Systems, March 2006, VDE Verlag GmbH.

Sha, L., Abdelzaher, T., Arzen, K., Cervin, A., Theodore, B., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J. and Mok, A.K. (2004) “Real Time Scheduling Theory: A Historical Perspective”, Real Time Systems, volume 28, pp. 101-155.

Shearer, D. (2008) “Greening Up' Blue Skies”, Aircraft Maintenance Technology, WWW website (last accessed: 16 July 2011), URL: http://www.amtonline.com/ publication/article.jsp?pubId=1&id=6564&pageNum=1.

Short, M.J., Pont, M.J. and Fang, J. (2008) “Exploring the Impact of Task Preemption on Dependability in Time-Triggered Embedded Systems: a Pilot Study”, Euromicro Conference on Real-Time Systems, 2-4 July 2008, pp. 83-91.

Short, M.J. (2010) “The Case For Non-preemptive Deadline-driven Scheduling In Real-time Embedded Systems”, Proceedings of the World Congress on Engineering, June 30 - July 2 2010.

Smith, D. (2007) “General Aviation: FADEC: Making a Piston Engine Act Like a Turbine”, Aviation today, WWW website (last accessed 31st July 2011), URL: http://www.aviationtoday.com/am/categories/bga/General-Aviation-FADEC-Making-a-Piston-Engine-Act-Like-a-Turbine_13506.html.

Smithsonian, “Smithsonian National Air and Space Museum: Rutan Voyager”, WWW website (last accessed 28th July 2011), URL: http://www.nasm.si.edu/collections /artifact.cfm?id=A19880548000.

Sohrabi, K., Gao, J., Ailawadhi, V. and Pottie, G.J. (2000) “Protocols for self-organization of a wireless sensor network”, IEEE Personal Communications, volume 7, issue 5.

Spuri, M. and Buttazzo, G. (1996) “Scheduling aperiodic tasks in dynamic priority systems”, Real-Time Systems, volume 10, issue 2, pp. 179-210.

Stewart, D.B. (2001) “Twenty-Five Most Common Mistakes with Real-Time Software Developement”, International Conference on Embedded Systems, April, 2001.

Stewart, D.B. and Khosla, P.K. (1991) “Real-Time Scheduling of Sensor-Based Control Systems”, Eighth IEEE Workshop on Real-Time Operating Systems and Software in conjunction with 17th IFAC/IFIP Workshop on Real-Time Programming, May 1991, pp. 144-150.

STMicroelectronics (2011) “Reference Manual: STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs”, Online reference manual (last accessed 19th July 2011), URL: http://www. st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/REFERENCE_MANUAL/CD00171190.pdf.

TCM (2009) “Full Authority Digital Engine Controls”, Teledyne Continental Motors, Online document (last accessed 17th September 2011), URL: http://www. genuinecontinental.aero/documents/fadec.ppt.

Vallerio, K.S. and Jha, N.K. (2003) “Task graph extraction for embedded system synthesis”, Proceedings of the 16th Int. Conference on VLSI Design concurrently with the 2nd International Conference on Embedded Systems Design, 4-8 January 2003, IEEE Computer Society, Washington DC, pp. 480-486.

Wang, H. and Pont, M.J. (2008) “Design and Implementation of a Static Pre-emptive Scheduler with Highly-Predictable Behaviour”, 4th UK Embedded Forum 2008, pp. 88-94.

Werner-Allen, G., Lorincz, K., Welsh, M., Marcillo, O., Johnson, J., Ruiz, M. and Lees, J. (2006) “Deploying a Wireless Sensor Network on an Active Volcano”, IEEE Internet Computing, volume 10, issue 2, pp. 18.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J. and Stenström, P. (2008) “The worst-case execution-time problem —

Overview of methods and survey of tools", ACM Transactions on Embedded Computing Systems, volume 7, issue 3.

Xu, J., Parnas, D.L. (1990) "Scheduling processes with release times, deadlines, precedence and exclusion relations", IEEE Transactions on Software Engineering, volume 16, issue 3, pp. 360-369.

Xu, N., Rangwala, S., Chintalapudi, K.K., Ganesan, D., Broad, A., Govindan, R. and Estrin, D. (2004) "A wireless sensor network For structural monitoring", Proceedings of the 2nd international conference on Embedded networked sensor systems, November 03-05, 2004, Baltimore, MD, USA.

Xu, J. (2007) "A software architecture for simplifying verification of system timing properties", Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing.

Xu, J. (2003) "On inspection and verification of software with timing requirements", IEEE Transactions on Software Engineering, volume 29, issue 8, pp. 705-720.

Xu, J. and Parnas, D.L. (1993) "On satisfying timing constraints in hard - real - time systems", IEEE Transactions on Software Engineering, volume 19, issue 1, pp. 70-84.

Xu, J. and Parnas, D.L. (2000) "Priority Scheduling Versus Pre-Run-Time Scheduling", The International Journal of Time-Critical Computing Systems, volume 18, pp. 7-23.

Yedamale, P. (2003) "AN885: Brushless DC (BLDC) Motor Fundamentals", Microchip, Application note (last accessed 12th July 2011), URL: http://ww1. microchip.com/downloads/en/AppNotes/00885a.pdf.

Yodaiken, V. (2002) "Against priority inheritance", Finite State Machine Labs (FSMLabs), URL: http://www.math.unipd.it/~tullio/SCD/2007/Materiale/Yodaiken-200207.pdf.