EMBEDDED SYSTEMS

THEORY AND DESIGN METHODOLOGY

Edited by **Kiyofumi Tanaka**

INTECH

EMBEDDED SYSTEMS – THEORY AND DESIGN METHODOLOGY

Edited by **Kiyofumi Tanaka**

INTECHWEB.ORG

Embedded Systems – Theory and Design Methodology
Edited by Kiyofumi Tanaka

Published by InTech
Janeza Trdine 9, 51000 Rijeka, Croatia

Publishing Process Manager Marina Jozipovic
Technical Editor Teodora Smiljanic
Cover Designer InTech Design Team

First published February, 2012
Printed in Croatia

A free online edition of this book is available at www.intechopen.com
Additional hard copies can be obtained from orders@intechweb.org

Embedded Systems – Theory and Design Methodology, Edited by Kiyofumi Tanaka
p. cm.
ISBN 978-953-51-0167-3

Preface

Nowadays, embedded systems have permeated various aspects of industry. Therefore, we can hardly discuss our life or society from now on without referring to embedded systems. For wide-ranging embedded systems to continue their growth, a number of high-quality fundamental and applied researches are indispensable.

This book addresses a wide spectrum of research topics on embedded systems, including basic researches, theoretical studies, and practical work. The book consists of nineteen chapters. In Part 1, real-time property, task scheduling, predictability, reliability and safety, which are key factors in real-time embedded systems and will be further treated as important, are introduced by five chapters.

Then, design/evaluation methodology, verification, and development environment, which are indispensable to embedded systems development, are dealt with in Part 2, through ten chapters.

In Part 3, two chapters present high-level synthesis technologies, which can raise design abstraction and make system development periods shorter. The third chapter reveals embedded low-power SRAM cells for future embedded system, and the last one addresses the important issue, energy efficient applications.

Embedded systems are part of products that can be made only after fusing miscellaneous technologies together. I expect that various technologies condensed in this book would be helpful to researchers and engineers around the world.

The editor would like to express his appreciation to the authors of this book for presenting their precious work. The editor would like to thank Ms. Marina Jozipovic, the publishing process manager of this book, and all members of InTech for their editorial assistance.

Kiyofumi Tanaka
School of Information Science
Japan Advanced Institute of Science and Technology
Japan

Contents

Part 1

Real-Time Property, Task Scheduling, Predictability, Reliability, and Safety

Ways for Implementing Highly-Predictable Embedded Systems Using Time-Triggered Co-Operative (TTC) Architectures

Mouaaz Nahas and Ahmed M. Nahhas
Department of Electrical Engineering, College of Engineering and Islamic Architecture, Umm Al-Qura University, Makkah, Saudi Arabia

1. Introduction

Embedded system is a special-purpose computer system which is designed to perform a small number of dedicated functions for a specific application (Sachitanand, 2002; Kamal, 2003). Examples of applications using embedded systems are: microwave ovens, TVs, VCRs, DVDs, mobile phones, MP3 players, washing machines, air conditions, handheld calculators, printers, digital watches, digital cameras, automatic teller machines (ATMs) and medical equipments (Barr, 1999; Bolton, 2000; Fisher et al., 2004; Pop et al., 2004). Besides these applications, which can be viewed as "noncritical" systems, embedded technology has also been used to develop "safety-critical" systems where failures can have very serious impacts on human safety. Examples include aerospace, automotive, railway, military and medical applications (Redmill, 1992; Profeta et al., 1996; Storey, 1996; Konrad et al., 2004).

The utilization of embedded systems in safety-critical applications requires that the system should have real-time operations to achieve correct functionality and/or avoid any possibility for detrimental consequences. Real-time behavior can only be achieved if the system is able to perform *predictable* and *deterministic* processing (Stankovic, 1988; Pont, 2001; Buttazzo, 2005; Phatrapornnant, 2007). As a result, the correct behavior of a real-time system depends on the time at which these results are produced as well as the logical correctness of the output results (Avrunin et al., 1998; Kopetz, 1997). In real-time embedded applications, it is important to predict the timing behavior of the system to guarantee that the system will behave correctly and consequently the life of the people using the system will be saved. Hence, predictability is the key characteristic in real-time embedded systems.

Embedded systems engineers are concerned with all aspects of the system development including hardware and software engineering. Therefore, activities such as specification, design, implementation, validation, deployment and maintenance will all be involved in the development of an embedded application (Fig. 1). A design of any system usually starts with ideas in people's mind. These ideas need to be captured in requirements specification documents that specify the basic functions and the desirable features of the system. The system design process then determines how these functions can be provided by the system components.

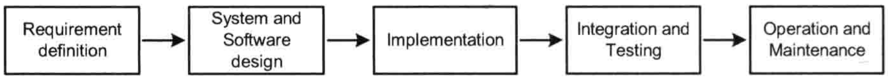

Fig. 1. The system development life cycle (Nahas, 2008).

For successful design, the system requirements have to be expressed and documented in a very clear way. Inevitably, there can be numerous ways in which the requirements for a simple system can be described.

Once the system requirements have been clearly defined and well documented, the first step in the design process is to design the overall system *architecture*. Architecture of a system basically represents an overview of the system components (i.e. sub-systems) and the interrelationships between these different components. Once the software architecture is identified, the process of implementing that architecture should take place. This can be achieved using a lower-level system representation such as an operating system or a *scheduler*. Scheduler is a very simple operating system for an embedded application (Pont, 2001). Building the scheduler would require a *scheduling algorithm* which simply provides the set of rules that determine the order in which the tasks will be executed by the scheduler during the system operating time. It is therefore the most important factor which influences predictability in the system, as it is responsible for satisfying timing and resource requirements (Buttazzo, 2005). However, the actual implementation of the scheduling algorithm on the embedded microcontroller has an important role in determining the functional and temporal behavior of the embedded system.

This chapter is mainly concerned with so-called "Time-Triggered Co-operative" (TTC) schedulers and how such algorithms can be implemented in highly-predictable, resource-constrained embedded applications.

The layout of the chapter is as follows. Section 2 provides a detailed comparison between the two key software architectures used in the design of real-time embedded systems, namely "time-triggered" and "event-triggered". Section 3 introduces and compares the two most known scheduling policies, "co-operative" and "pre-emptive", and highlights the advantages of co-operative over pre-emptive scheduling. Section 4 discusses the relationship between scheduling algorithms and scheduler implementations in practical embedded systems. In Section 5, Time-Triggered Co-operative (TTC) scheduling algorithm is introduced in detail with a particular focus on its strengths and drawbacks and how such drawbacks can be addressed to maintain its reliability and predictability attributes. Section 6 discusses the sources and impact of timing jitter in TTC scheduling algorithm. Section 7 describes various possible ways in which the TTC scheduling algorithm can be implemented on resource-constrained embedded systems that require highly-predictable system behavior. In Section 8, the various scheduler implementations are compared and contrasted in terms of jitter characteristics, error handling capabilities and resource requirements. The overall chapter conclusions are presented in Section 9.

2. Software architectures of embedded systems

Embedded systems are composed of hardware and software components. The success of an embedded design, thus, depends on the right selection of the hardware platform(s) as well as the software environment used in conjunction with the hardware. The selection of hardware and software architectures of an application must take place at early stages in the development process (typically at the design phase). Hardware architecture relates mainly to the type of the processor (or microcontroller) platform(s) used and the structure of the various hardware components that are comprised in the system: see Mwelwa (2006) for further discussion about hardware architectures for embedded systems.

Provided that the hardware architecture is decided, an embedded application requires an appropriate form of software architecture to be implemented. To determine the most appropriate choice for software architecture in a particular system, this condition must be fulfilled (Locke, 1992): *"The [software] architecture must be capable of providing a provable prediction of the ability of the application design to meet all of its time constraints."*

Since embedded systems are usually implemented as collections of *real-time tasks*, the various possible system architectures may then be determined by the characteristics of these tasks. In general, there are two main software architectures which are typically used in the design of embedded systems:

Event-triggered (ET): tasks are invoked as a response to aperiodic events. In this case, the system takes no account of time: instead, the system is controlled purely by the response to external events, typically represented by interrupts which can arrive at anytime (Bannatyne, 1998; Kopetz, 1991b). Generally, ET solution is recommended for applications in which sporadic data messages (with unknown request times) are exchanged in the system (Hsieh and Hsu, 2005).

Time-triggered (TT): tasks are invoked periodically at specific time intervals which are known in advance. The system is usually driven by a global clock which is linked to a hardware timer that overflows at specific time instants to generate periodic interrupts (Bennett, 1994). In distributed systems, where multi-processor hardware architecture is used, the global clock is distributed across the network (via the communication medium) to synchronise the local time base of all processors. In such architectures, time-triggering mechanism is based on time-division multiple access (TDMA) in which each processor-node is allocated a periodic time slot to broadcast its periodic messages (Kopetz, 1991b). TT solution can suit many control applications where the data messages exchanged in the system are periodic (Kopetz, 1997).

Many researchers argue that ET architectures are highly flexible and can provide high resource efficiency (Obermaisser, 2004; Locke, 1992). However, ET architectures allow several interrupts to arrive at the same time, where these interrupts might indicate (for example) that two different faults have been detected at the same time. Inevitably, dealing with an occurrence of several events at the same time will increase the system complexity and reduce the ability to predict the behavior of the ET system (Scheler and Schröder-Preikschat, 2006). In more severe circumstances, the system may fail completely if it is heavily loaded with events that occur at once (Marti, 2002). In contrast, using TT architectures helps to ensure that only a single event is handled at a time and therefore the behavior of the system can be highly-predictable.

Since highly-predictable system behavior is an important design requirement for many embedded systems, TT software architectures have become the subject of considerable attention (e.g. see Kopetz, 1997). In particular, it has been widely accepted that TT architectures are a good match for many safety-critical applications, since they can help to improve the overall safety and reliability (Allworth, 1981; Storey, 1996; Nissanke, 1997; Bates; 2000; Obermaisser, 2004). Liu (2000) highlights that TT systems are easy to validate, test, and certify because the times related to the tasks are deterministic. Detailed comparisons between the TT and ET concepts were performed by Kopetz (1991a and 1991b).

3. Schedulers and scheduling algorithms

Most embedded systems involve several tasks that share the system resources and communicate with one another and/or the environment in which they operate. For many projects, a key challenge is to work out how to schedule tasks so that they can meet their timing constraints. This process requires an appropriate form of *scheduler*[1]. A scheduler can be viewed as a very simple operating system which calls tasks periodically (or aperiodically) during the system operating time. Moreover, as with desktop operating systems, a scheduler has the responsibility to manage the computational and data resources in order to meet all temporal and functional requirements of the system (Mwelwa, 2006).

According to the nature of the operating tasks, any real-time scheduler must fall under one of the following types of scheduling policies:

Pre-emptive scheduling: where a multi-tasking process is allowed. In more details, a task with higher priority is allowed to pre-empt (i.e. interrupt) any lower priority task that is currently running. The lower priority task will resume once the higher priority task finishes executing. For example, suppose that - over a particular period of time - a system needs to execute four tasks (Task A, Task B, Task C, Task D) as illustrated in Fig. 2.

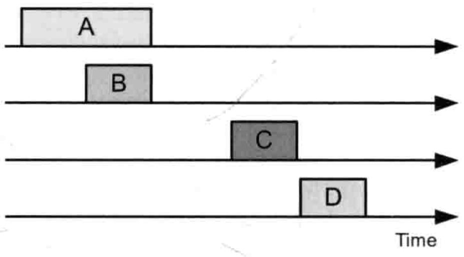

Fig. 2. A schematic representation of four tasks which need to be scheduled for execution on a single-processor embedded system (Nahas, 2008).

Assuming a single-processor system is used, Task C and Task D can run as required where Task B is due to execute before Task A is complete. Since no more than one task can run at the same time on a single-processor, Task A or Task B has to relinquish control of the CPU.

[1] Note that schedulers represent the core components of "Real-Time Operating System" (RTOS) kernels. Examples of commercial RTOSs which are used nowadays are: VxWorks (from Wind River), Lynx (from LynxWorks), RTLinux (from FSMLabs), eCos (from Red Hat), and QNX (from QNX Software Systems). Most of these operating systems require large amount of computational and memory resources which are not readily available in low-cost microcontrollers like the ones targeted in this work.

In pre-emptive scheduling, a higher priority might be assigned to Task B with the consequence that – when Task B is due to run – Task A will be interrupted, Task B will run, and Task A will then resume and complete (Fig. 3).

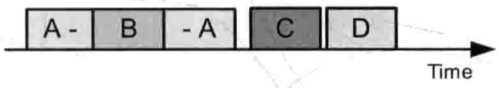

Fig. 3. Pre-emptive scheduling of Task A and Task B in the system shown in Fig. 2: Task B, here, is assigned a higher priority (Nahas, 2008).

Co-operative (or "non-pre-emptive") scheduling: where only a single-tasking process is allowed. In more details, if a higher priority task is ready to run while a lower priority task is running, the former task cannot be released until the latter one completes its execution. For example, assume the same set of tasks illustrated in Fig. 2. In the simplest solution, Task A and Task B can be scheduled co-operatively. In these circumstances, the task which is currently using the CPU is implicitly assigned a high priority: any other task must therefore wait until this task relinquishes control before it can execute. In this case, Task A will complete and then Task B will be executed (Fig. 4).

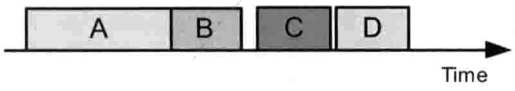

Fig. 4. Co-operative scheduling of Task A and Task B in the system shown in Fig. 2 (Nahas, 2008).

Hybrid scheduling: where a limited, but efficient, multi-tasking capabilities are provided (Pont, 2001). That is, only one task in the whole system is set to be pre-emptive (this task is best viewed as "highest-priority" task), while other tasks are running co-operatively (Fig. 5). In the example shown in the figure, suppose that Task B is a short task which has to execute immediately when it arrives. In this case, Task B is set to be pre-emptive so that it acquires the CPU control to execute whenever it arrives and whether (or not) other task is running.

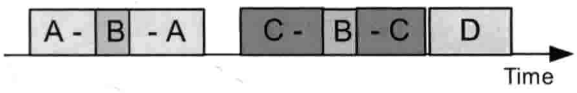

Fig. 5. Hybrid scheduling of four-tasks: Task B is set to be pre-emptive, where Task A, Task C and Task D run co-operatively (Nahas, 2008).

Overall, when comparing co-operative with pre-emptive schedulers, many researchers have argued that co-operative schedulers have many desirable features, particularly for use in safety-related systems (Allworth, 1981; Ward, 1991; Nissanke, 1997; Bates, 2000; Pont, 2001). For example, Bates (2000) identified the following four advantages of co-operative scheduling over pre-emptive alternatives:

- The scheduler is simpler.
- The overheads are reduced.
- Testing is easier.
- Certification authorities tend to support this form of scheduling.

Similarly, Nissanke (1997) noted: *"[Pre-emptive] schedules carry greater runtime overheads because of the need for context switching - storage and retrieval of partially computed results. [Co-operative] algorithms do not incur such overheads. Other advantages of co-operative algorithms include their better understandability, greater predictability, ease of testing and their inherent capability for guaranteeing exclusive access to any shared resource or data."*

Many researchers still, however, believe that pre-emptive approaches are more effective than co-operative alternatives (Allworth, 1981; Cooling, 1991). This can be due to different reasons. As in (Pont, 2001), one of the reasons why pre-emptive approaches are more widely discussed and considered is because of confusion over the options available. Pont gave an example that the basic cyclic scheduling, which is often discussed by many as an alternative to pre-emptive, is not a representative of the wide range of co-operative scheduling architectures that are available.

Moreover, one of the main issues that concern people about the reliability of co-operative scheduling is that long tasks can have a negative impact on the responsiveness of the system. This is clearly underlined by Allworth (1981): *"[The] main drawback with this co-operative approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response [of the] system is not to be impaired."*

However, in many practical embedded systems, the process (task) duration is extremely short. For example, calculations of one of the very complicated algorithms, the "proportional integral differential" (PID) controller, can be carried out on the most basic (8-bit) 8051 microcontroller in around 0.4 ms: this imposes insignificant processor load in most systems - including flight control - where 10 ms sampling rate is adequate (Pont, 2001). Pont has also commented that if the system is designed to run long tasks, *"this is often because the developer is unaware of some simple techniques that can be used to break down these tasks in an appropriate way and – in effect – convert long tasks called infrequently into short tasks called frequently"*: some of these techniques are introduced and discussed in Pont (2001).

Moreover, if the performance of the system is seen slightly poor, it is often advised to update the microcontroller hardware rather than to use a more complex software architecture. However, if changing the task design or microcontroller hardware does not provide the level of performance which is desired for a particular application, then more than one microcontroller can be used. In such cases, long tasks can be easily moved to another processor, allowing the host processor to respond rapidly to other events as required (for further details, see Pont, 2001; Ayavoo et al., 2007).

Please note that the very wide use of pre-emptive schedulers can simply be resulted from a poor understanding and, hence, undervaluation of the co-operative schedulers. For example, a co-operative scheduler can be easily constructed using only a few hundred lines of highly portable code written in a high-level programming language (such as 'C'), while the resulting system is highly-predictable (Pont, 2001).

It is also important to understand that sometimes pre-emptive schedulers are more widely used in RTOSs due to commercial reasons. For example, companies may have commercial benefits from using pre-emptive environments. Consequently, as the complexity of these environments increases, the code size will significantly increase making 'in-house' constructions of such environments too complicated. Such complexity factors lead to the sale of commercial RTOS products at high prices (Pont, 2001). Therefore, further academic research has been conducted in this area to explore alternative solutions. For example, over the last few years, the Embedded Systems Laboratory (ESL) researchers have considered various ways in which simple, highly-predictable, non-pre-emptive (co-operative) schedulers can be implemented in low-cost embedded systems.

4. Scheduling algorithm and scheduler implementation

A key component of the scheduler is the *scheduling algorithm* which basically determines the order in which the tasks will be executed by the scheduler (Buttazzo, 2005). More specifically, a scheduling algorithm is the set of rules that, at every instant while the system is running, determines which task must be allocated the resources to execute.

Developers of embedded systems have proposed various scheduling algorithms that can be used to handle tasks in real-time applications. The selection of appropriate scheduling algorithm for a set of tasks is based upon the capability of the algorithm to satisfy all timing constraints of the tasks: where these constraints are derived from the application requirements. Examples of common scheduling algorithms are: Cyclic Executive (Locke, 1992), Rate Monotonic (Liu & Layland, 1973), Earliest-Deadline-First (Liu & Layland, 1973; Liu, 2000), Least-Laxity-First (Mok, 1983), Deadline Monotonic (Leung, 1982) and Shared-Clock (Pont, 2001) schedulers (see Rao et al., 2008 for a simple classification of scheduling algorithms). This chapter outlines one key example of scheduling algorithms that is widely used in the design of real-time embedded systems when highly-predictable system behavior is an essential requirement: this is the Time Triggered Co-operative scheduler which is a form of cyclic executive.

Note that once the design specifications are converted into appropriate design elements, the system implementation process can take place by translating those designs into software and hardware components. People working on the development of embedded systems are often concerned with the software implementation of the system in which the system specifications are converted into an executable system (Sommerville, 2007; Koch, 1999). For example, Koch interpreted the implementation of a system as the way in which the software program is arranged to meet the system specifications.

The implementation of schedulers is a major problem which faces designers of real-time scheduling systems (for example, see Cho et al., 2005). In their useful publication, Cho and colleges clarified that the well-known term *scheduling* is used to describe the process of finding the optimal schedule for a set of real-time tasks, while the term *scheduler implementation* refers to the process of implementing a physical (software or hardware) scheduler that enforces - at run-time - the task sequencing determined by the designed schedule (Cho et al., 2007).